TiDB 中文手册

PingCAP Inc.

20230823

目录

1 文档中心 30

2 关于 TiDB 30

2.1 TiDB 简介 . . . 30

2.1.1 五大核心特性 . . . 30

2.1.2 四大核心应用场景 . . . 30

2.1.3 另请参阅 . . . 31

2.2 TiDB 7.0.0 Release Notes . . . 31

2.2.1 功能详情 . . . 33

2.2.2 兼容性变更 . . . 38

2.2.3 改进提升 . . . 54

2.2.4 错误修复 . . . 55

2.2.5 贡献者 . . . 57

2.3 TiDB 功能概览 . . . 57

2.3.1 数据类型，函数和操作符 . . . 58

2.3.2 索引和约束 . . . 58

2.3.3 SQL 语句 . . . 59

2.3.4 高级 SQL 功能 . . . 60

2.3.5 数据定义语言 (DDL) . . . 60

2.3.6 事务 . . . 60

2.3.7 分区 . . . 61

2.3.8 统计信息 . . . 61

2.3.9 安全 . . . 61

2.3.10 数据导入和导出 . . . 62
2.3.11 管理，可视化和工具 . . . 62
2.4 与 MySQL 兼容性对比 . . . 63
2.4.1 不支持的功能特性 . . . 64
2.4.2 与 MySQL 有差异的特性详细说明 . . . 64
2.5 使用限制 . . . 69
2.5.1 标识符长度限制 . . . 69
2.5.2 Databases、 Tables、 Views、 Connections 总个数限制 . . . 69
2.5.3 单个 Database 的限制 . . . 70
2.5.4 单个 Table 的限制 . . . 70
2.5.5 单行的限制 . . . 70
2.5.6 数据类型限制 . . . 70
2.5.7 SQL Statements 的限制 . . . 70
2.5.8 TiKV 版本的限制 . . . 71
2.6 TiDB 社区荣誉列表 . . . 71
2.6.1 TiDB 开发者 . . . 71
2.6.2 TiDB 文档写作者和译员 . . . 71
2.7 路线图 . . . 72
3 快速上手 72
3.1 TiDB 数据库快速上手指南 . . . 72
3.1.1 部署本地测试集群 . . . 72
3.1.2 在单机上模拟部署生产环境集群 . . . 76
3.1.3 探索更多 . . . 80
3.2 HTAP 快速上手指南 . . . 81
3.2.1 基础概念 . . . 81
3.2.2 体验步骤 . . . 81
3.2.3 探索更多 . . . 85
3.3 SQL 基本操作 . . . 85
3.3.1 分类 . . . 85
3.3.2 查看、创建和删除数据库 . . . 85
3.3.3 创建、查看和删除表 . . . 86
3.3.4 创建、查看和删除索引 . . . 86

3.3.5 记录的增删改 · · · · · 87
3.3.6 查询数据 · · · · · 87
3.3.7 创建、授权和删除用户 · · · · · 88
3.4 HTAP 深入探索指南 · · · · · 88
3.4.1 HTAP 适用场景 · · · · · 89
3.4.2 HTAP 架构 · · · · · 89
3.4.3 HTAP 环境准备 · · · · · 89
3.4.4 HTAP 数据准备 · · · · · 90
3.4.5 HTAP 数据处理 · · · · · 90
3.4.6 HTAP 性能监控 · · · · · 90
3.4.7 HTAP 故障诊断 · · · · · 91
3.4.8 探索更多 · · · · · 91
4 应用开发 91
4.1 开发者手册概览 · · · · · 91
4.1.1 TiDB 基础 · · · · · 91
4.1.2 TiDB 事务机制 · · · · · 91
4.1.3 应用程序与 TiDB 交互的方式 · · · · · 92
4.1.4 扩展阅读 · · · · · 92
4.2 快速开始 · · · · · 92
4.2.1 使用 TiDB Serverless 构建 TiDB 集群 · · · · · 92
4.2.2 使用 TiDB 的增删改查 SQL · · · · · 94
4.3 示例程序 · · · · · 96
4.3.1 TiDB 和 Golang 的简单 CRUD 应用程序 · · · · · 96
4.3.2 使用 Spring Boot 构建 TiDB 应用程序 · · · · · 113
4.3.3 TiDB 和 Java 的简单 CRUD 应用程序 · · · · · 140
4.3.4 使用 Django 构建 TiDB 应用程序 · · · · · 176
4.3.5 TiDB 和 Python 的简单 CRUD 应用程序 · · · · · 193
4.4 连接到 TiDB · · · · · 214
4.4.1 选择驱动或 ORM 框架 · · · · · 214
4.4.2 连接到 TiDB · · · · · 219
4.4.3 连接池与连接参数 · · · · · 221

4.5 数据库模式设计 · · · · · 227
4.5.1 概述 · · · · · 227
4.5.2 创建数据库 · · · · · 229
4.5.3 创建表 · · · · · 230
4.5.4 创建二级索引 · · · · · 240
4.6 数据写入 · · · · · 243
4.6.1 插入数据 · · · · · 243
4.6.2 更新数据 · · · · · 249
4.6.3 删除数据 · · · · · 258
4.6.4 使用 TTL (Time to Live) 定期删除过期数据 · · · · · 266
4.6.5 预处理语句 · · · · · 278
4.7 数据读取 · · · · · 282
4.7.1 单表查询 · · · · · 282
4.7.2 多表连接查询 · · · · · 288
4.7.3 子查询 · · · · · 295
4.7.4 分页查询 · · · · · 298
4.7.5 视图 · · · · · 304
4.7.6 临时表 · · · · · 306
4.7.7 公共表表达式 (CTE) · · · · · 311
4.7.8 读取副本数据 · · · · · 315
4.7.9 HTAP 查询 · · · · · 328
4.8 事务 · · · · · 333
4.8.1 事务概览 · · · · · 333
4.8.2 乐观事务和悲观事务 · · · · · 336
4.8.3 事务限制 · · · · · 365
4.8.4 事务错误处理 · · · · · 383
4.9 优化 SQL 性能 · · · · · 386
4.9.1 概览 · · · · · 386
4.9.2 SQL 性能调优 · · · · · 387
4.9.3 性能调优最佳实践 · · · · · 392
4.9.4 索引的最佳实践 · · · · · 395
4.9.5 其他优化 · · · · · 397

4.10 故障诊断 ····· 400
4.10.1 SQL 或事务问题 ····· 400
4.10.2 结果集不稳定 ····· 401
4.10.3 TiDB 中的各种超时 ····· 405
4.11 引用文档 ····· 406
4.11.1 Bookshop 应用 ····· 406
4.11.2 规范 ····· 411
4.12 云原生开发环境 ····· 413
4.12.1 Gitpod ····· 413
4.13 第三方工具支持 ····· 418
4.13.1 TiDB 支持的第三方工具 ····· 418
4.13.2 已知的第三方工具兼容问题 ····· 426
4.13.3 TiDB 与 ProxySQL 集成 ····· 431
5 部署标准集群 445
5.1 TiDB 软件和硬件环境建议配置 ····· 445
5.1.1 操作系统及平台要求 ····· 445
5.1.2 软件配置要求 ····· 446
5.1.3 服务器建议配置 ····· 447
5.1.4 网络要求 ····· 449
5.1.5 磁盘空间要求 ····· 449
5.1.6 客户端 Web 浏览器要求 ····· 450
5.2 TiDB 环境与系统配置检查 ····· 450
5.2.1 在 TiKV 部署目标机器上添加数据盘 EXT4 文件系统挂载参数 ····· 450
5.2.2 检测及关闭系统 swap ····· 451
5.2.3 设置 TiDB 节点的临时空间（推荐） ····· 452
5.2.4 检测及关闭目标部署机器的防火墙 ····· 453
5.2.5 检测及安装 NTP 服务 ····· 453
5.2.6 检查和配置操作系统优化参数 ····· 455
5.2.7 手动配置 SSH 互信及 sudo 免密码 ····· 460
5.2.8 安装 numactl 工具 ····· 461

5.3 规划集群拓扑 · · · · · 462
5.3.1 最小拓扑架构 · · · · · 462
5.3.2 TiFlash 部署拓扑 · · · · · 463
5.3.3 TiCDC 部署拓扑 · · · · · 464
5.3.4 TiDB Binlog 部署拓扑 · · · · · 465
5.3.5 TiSpark 部署拓扑 · · · · · 468
5.3.6 跨数据中心部署拓扑 · · · · · 471
5.3.7 混合部署拓扑 · · · · · 474
5.4 安装与启动 · · · · · 477
5.4.1 使用 TiUP 部署 TiDB 集群 · · · · · 477
5.4.2 在 Kubernetes 上部署 TiDB · · · · · 491
5.5 验证集群运行状态 · · · · · 492
5.5.1 通过 TiUP 检查集群状态 · · · · · 492
5.5.2 通过 TiDB Dashboard 和 Grafana 检查集群状态 · · · · · 492
5.5.3 登录数据库执行简单 DML/DDL 操作和查询 SQL 语句 · · · · · 493
5.6 测试集群性能 · · · · · 496
5.6.1 如何用 Sysbench 测试 TiDB · · · · · 496
5.6.2 如何对 TiDB 进行 TPC-C 测试 · · · · · 500
5.6.3 如何对 TiDB 进行 CH-benCHmark 测试 · · · · · 502

6 数据迁移 506
6.1 数据迁移概述 · · · · · 506
6.1.1 迁移 Aurora MySQL 到 TiDB · · · · · 506
6.1.2 迁移 MySQL 到 TiDB · · · · · 507
6.1.3 分库分表 MySQL 合并迁移到 TiDB · · · · · 507
6.1.4 从文件迁移数据到 TiDB · · · · · 507
6.1.5 TiDB 集群增量数据同步 · · · · · 507
6.1.6 复杂迁移场景 · · · · · 507
6.2 数据迁移工具概览 · · · · · 508
6.2.1 TiDB Data Migration (DM) · · · · · 508
6.2.2 TiDB Lightning · · · · · 508
6.2.3 Dumpling · · · · · 509
6.2.4 TiCDC · · · · · 509

6.2.5 Backup & Restore (BR) · · · · · 509
6.2.6 sync-diff-inspector · · · · · 510
6.2.7 使用 TiUP 快速安装 · · · · · 510
6.2.8 探索更多 · · · · · 511
6.3 数据迁移场景 · · · · · 512
6.3.1 从 Amazon Aurora 迁移数据到 TiDB · · · · · 512
6.3.2 从小数据量 MySQL 迁移数据到 TiDB · · · · · 518
6.3.3 从大数据量 MySQL 迁移数据到 TiDB · · · · · 522
6.3.4 从小数据量分库分表 MySQL 合并迁移数据到 TiDB · · · · · 529
6.3.5 从大数据量分库分表 MySQL 合并迁移数据到 TiDB · · · · · 534
6.3.6 从 CSV 文件迁移数据到 TiDB · · · · · 542
6.3.7 从 SQL 文件迁移数据到 TiDB · · · · · 547
6.3.8 从 Parquet 文件迁移数据到 TiDB · · · · · 549
6.3.9 从 TiDB 集群迁移数据至另一 TiDB 集群 · · · · · 551
6.3.10 从 TiDB 集群迁移数据至兼容 MySQL 的数据库 · · · · · 557
6.4 复杂迁移场景 · · · · · 562
6.4.1 上游使用 pt-osc/gh-ost 工具的持续同步场景 · · · · · 562
6.4.2 下游存在更多列的迁移场景 · · · · · 563
6.4.3 如何过滤 binlog 事件 · · · · · 567
6.4.4 如何通过 SQL 表达式过滤 DML · · · · · 570

7 数据集成 572
7.1 数据集成概述 · · · · · 572
7.1.1 与 Confluent Cloud 和 Snowflake 进行数据集成 · · · · · 572
7.1.2 与 Apache Kafka 和 Apache Flink 进行数据集成 · · · · · 572
7.2 数据集成场景 · · · · · 572
7.2.1 与 Confluent Cloud 和 Snowflake 进行数据集成 · · · · · 572
7.2.2 与 Apache Kafka 和 Apache Flink 进行数据集成 · · · · · 585

8 运维操作 589
8.1 升级 TiDB 版本 · · · · · 589
8.1.1 使用 TiUP 升级 TiDB · · · · · 589
8.1.2 使用 TiDB Operator · · · · · 595
8.1.3 TiFlash v6.2 升级帮助 · · · · · 595

8.2 扩缩容 · · · · · 597
8.2.1 使用 TiUP 扩容缩容 TiDB 集群 · · · · · 597
8.2.2 使用 TiDB Operator · · · · · 607
8.3 备份与恢复 · · · · · 607
8.3.1 TiDB 备份与恢复概述 · · · · · 607
8.3.2 架构设计 · · · · · 615
8.3.3 使用 BR 进行备份与恢复 · · · · · 627
8.3.4 br cli 命令手册 · · · · · 647
8.3.5 参考指南 · · · · · 662
8.4 集群容灾 · · · · · 671
8.4.1 TiDB 容灾方案概述 · · · · · 671
8.4.2 基于主备集群的容灾方案 · · · · · 681
8.4.3 基于多副本的单集群容灾方案 · · · · · 691
8.4.4 基于备份与恢复的容灾方案 · · · · · 695
8.5 使用资源管控 (Resource Control) 实现资源隔离 · · · · · 697
8.5.1 使用场景 · · · · · 697
8.5.2 使用限制 · · · · · 697
8.5.3 什么是 Request Unit (RU) · · · · · 698
8.5.4 估算 SQL 所消耗的 RU · · · · · 699
8.5.5 相关参数 · · · · · 699
8.5.6 使用方法 · · · · · 699
8.5.7 关闭资源管控特性 · · · · · 701
8.5.8 监控与图表 · · · · · 701
8.5.9 工具兼容性 · · · · · 701
8.5.10 另请参阅 · · · · · 702
8.6 时区支持 · · · · · 702
8.7 日常巡检 · · · · · 703
8.7.1 TiDB Dashboard 关键指标 · · · · · 703
8.8 TiFlash 集群运维 · · · · · 709
8.8.1 查看 TiFlash 版本 · · · · · 709
8.8.2 TiFlash 重要日志介绍 · · · · · 709
8.8.3 TiFlash 系统表 · · · · · 710

8.9 TiUP 常见运维操作 · · · · · 710
8.9.1 查看集群列表 · · · · · 710
8.9.2 启动集群 · · · · · 711
8.9.3 查看集群状态 · · · · · 711
8.9.4 修改配置参数 · · · · · 711
8.9.5 Hotfix 版本替换 · · · · · 712
8.9.6 重命名集群 · · · · · 713
8.9.7 关闭集群 · · · · · 713
8.9.8 清除集群数据 · · · · · 714
8.9.9 销毁集群 · · · · · 714
8.10 在线修改集群配置 · · · · · 715
8.10.1 常用操作 · · · · · 715
8.11 Online Unsafe Recovery 使用文档 · · · · · 749
8.11.1 功能说明 · · · · · 750
8.11.2 适用场景 · · · · · 750
8.11.3 使用步骤 · · · · · 750
8.12 搭建双集群主从复制 · · · · · 754
8.12.1 第 1 步：搭建环境 · · · · · 755
8.12.2 第 2 步：迁移全量数据 · · · · · 756
8.12.3 第 3 步：迁移增量数据 · · · · · 759
8.12.4 第 4 步：模拟主集群故障 · · · · · 760
8.12.5 第 5 步：使用 redo log 确保数据一致性 · · · · · 760
8.12.6 第 6 步：恢复主集群及业务 · · · · · 760

9 监控与告警 761
9.1 TiDB 监控框架概述 · · · · · 761
9.1.1 Prometheus 在 TiDB 中的应用 · · · · · 761
9.1.2 Grafana 在 TiDB 中的应用 · · · · · 762
9.2 TiDB 集群监控 API · · · · · 764
9.2.1 使用状态接口 · · · · · 764
9.2.2 使用 metrics 接口 · · · · · 767

9.3 TiDB 集群监控部署 · · · · · 767
9.3.1 部署 Prometheus 和 Grafana · · · · · 767
9.3.2 配置 Grafana · · · · · 770
9.3.3 查看组件 metrics · · · · · 771
9.4 将 Grafana 监控数据导出成快照 · · · · · 773
9.4.1 使用方法 · · · · · 773
9.4.2 FAQs · · · · · 774
9.5 TiDB 集群报警规则 · · · · · 775
9.5.1 TiDB 报警规则 · · · · · 775
9.5.2 PD 报警规则 · · · · · 778
9.5.3 TiKV 报警规则 · · · · · 783
9.5.4 TiFlash 报警规则 · · · · · 791
9.5.5 TiDB Binlog 报警规则 · · · · · 791
9.5.6 TiCDC 报警规则 · · · · · 791
9.5.7 Node_exporter 主机报警规则 · · · · · 791
9.5.8 Blackbox_exporter TCP、ICMP 和 HTTP 报警规则 · · · · · 794
9.6 TiFlash 报警规则 · · · · · 798
9.6.1 TiFlash_schema_error · · · · · 798
9.6.2 TiFlash_schema_apply_duration · · · · · 799
9.6.3 TiFlash_raft_read_index_duration · · · · · 799
9.6.4 TiFlash_raft_wait_index_duration · · · · · 799
9.7 自定义监控组件的配置 · · · · · 800
9.7.1 自定义 Prometheus 配置 · · · · · 800
9.7.2 自定义 Grafana 配置 · · · · · 801
9.7.3 自定义 Alertmanager 配置 · · · · · 802
9.8 备份恢复监控告警 · · · · · 803
9.8.1 日志备份监控 · · · · · 803

10 故障诊断 806
10.1 故障诊断问题汇总 · · · · · 806
10.1.1 TiDB 集群问题导图 · · · · · 806
10.1.2 TiDB 集群故障诊断 · · · · · 821
10.1.3 TiFlash 常见问题 · · · · · 824

10.2 故障场景 · · · · · 828
10.2.1 慢查询 · · · · · 828
10.2.2 TiDB OOM 故障排查 · · · · · 847
10.2.3 TiDB 热点问题处理 · · · · · 852
10.2.4 读写延迟增加 · · · · · 862
10.2.5 乐观事务模型下写写冲突问题排查 · · · · · 866
10.2.6 TiDB 磁盘 I/O 过高的处理办法 · · · · · 869
10.2.7 TiDB 锁冲突问题处理 · · · · · 871
10.2.8 数据索引一致性报错 · · · · · 881
10.3 故障诊断方法 · · · · · 883
10.3.1 SQL 诊断 · · · · · 883
10.3.2 Statement Summary Tables · · · · · 885
10.3.3 TiDB Dashboard Top SQL 页面 · · · · · 893
10.3.4 定位消耗系统资源多的查询 · · · · · 900
10.3.5 使用 PLAN REPLAYER 保存和恢复集群现场信息 · · · · · 902
10.4 支持资源 · · · · · 907

11 性能调优 908
11.1 优化手册 · · · · · 908
11.1.1 TiDB 性能优化概述 · · · · · 908
11.1.2 TiDB 性能分析和优化 · · · · · 912
11.1.3 OLTP 负载性能优化实践 · · · · · 930
11.1.4 TiFlash 性能分析和优化方法 · · · · · 951
11.1.5 TiCDC 性能分析和优化方法 · · · · · 956
11.1.6 延迟的拆解分析 · · · · · 958
11.2 配置调优 · · · · · 976
11.3 SQL 性能调优 · · · · · 1029
11.3.1 SQL 性能调优 · · · · · 1029
11.3.2 理解 TiDB 执行计划 · · · · · 1029
11.3.3 SQL 优化流程 · · · · · 1101
11.3.4 控制执行计划 · · · · · 1186

12 教程 1225

12.1 单区域多 AZ 部署 TiDB ····· 1225
12.1.1 了解 Raft 协议 ····· 1225
12.1.2 同区域三 AZ 方案 ····· 1225
12.2 双区域多 AZ 部署 TiDB ····· 1231
12.2.1 简介 ····· 1231
12.2.2 架构 ····· 1231
12.2.3 配置 ····· 1233
12.3 单区域双 AZ 部署 TiDB ····· 1237
12.3.1 简介 ····· 1237
12.3.2 部署架构 ····· 1237
12.3.3 配置 ····· 1238
12.4 读取历史数据 ····· 1244
12.4.1 使用 Stale Read 功能读取历史数据（推荐） ····· 1244
12.4.2 通过系统变量 tidb_snapshot 读取历史数据 ····· 1255
12.5 最佳实践 ····· 1260
12.5.1 TiDB 最佳实践 ····· 1260
12.5.2 开发 Java 应用使用 TiDB 的最佳实践 ····· 1265
12.5.3 HAProxy 在 TiDB 中的最佳实践 ····· 1274
12.5.4 TiDB 高并发写入场景最佳实践 ····· 1282
12.5.5 使用 Grafana 监控 TiDB 的最佳实践 ····· 1291
12.5.6 PD 调度策略最佳实践 ····· 1301
12.5.7 海量 Region 集群调优最佳实践 ····· 1309
12.5.8 三节点混合部署的最佳实践 ····· 1314
12.5.9 在三数据中心下就近读取数据 ····· 1318
12.5.10 UUID 最佳实践 ····· 1318
12.5.11 只读存储节点最佳实践 ····· 1320
12.6 Placement Rules 使用文档 ····· 1322
12.6.1 规则系统介绍 ····· 1323
12.6.2 配置规则操作步骤 ····· 1325
12.6.3 典型场景示例 ····· 1329

12.7 Load Base Split · · · · · 1332
12.7.1 场景描述 · · · · · 1332
12.7.2 实现原理 · · · · · 1332
12.7.3 使用方法 · · · · · 1333
12.8 Store Limit · · · · · 1334
12.8.1 实现原理 · · · · · 1334
12.8.2 使用方法 · · · · · 1334
12.9 DDL 语句的执行原理及最佳实践 · · · · · 1335
12.9.1 DDL 执行原理 · · · · · 1335
12.9.2 最佳实践 · · · · · 1339
12.9.3 DDL 相关的命令介绍 · · · · · 1340
12.9.4 常见问题 · · · · · 1340

13 TiDB 工具 1340
13.1 TiDB 工具功能概览 · · · · · 1340
13.1.1 部署运维工具 · · · · · 1340
13.1.2 数据管理工具 · · · · · 1341
13.1.3 OLAP 分析工具 - TiSpark · · · · · 1344
13.2 TiDB 工具的使用场景 · · · · · 1344
13.2.1 在物理机或虚拟机上部署运维 TiDB · · · · · 1344
13.2.2 在 Kubernetes 上部署运维 TiDB · · · · · 1344
13.2.3 从 CSV 导入数据到 TiDB · · · · · 1344
13.2.4 从 MySQL/Aurora 导入全量数据 · · · · · 1345
13.2.5 从 MySQL/Aurora 迁移数据 · · · · · 1345
13.2.6 TiDB 集群备份与恢复 · · · · · 1345
13.2.7 迁出数据到 TiDB · · · · · 1345
13.2.8 TiDB 增量数据订阅 · · · · · 1345
13.3 TiDB 工具下载 · · · · · 1345
13.3.1 TiDB 工具包下载 · · · · · 1345
13.3.2 TiUniManager 下载 · · · · · 1348

13.4 TiUP · · · · · 1350
13.4.1 TiUP 文档地图 · · · · · 1350
13.4.2 TiUP 简介 · · · · · 1350
13.4.3 TiUP 术语及核心概念 · · · · · 1353
13.4.4 使用 TiUP 命令管理组件 · · · · · 1354
13.4.5 TiUP FAQ · · · · · 1357
13.4.6 TiUP 故障排查 · · · · · 1358
13.4.7 TiUP 命令参考手册 · · · · · 1360
13.4.8 通过 TiUP 部署 TiDB 集群的拓扑文件配置 · · · · · 1436
13.4.9 通过 TiUP 部署 DM 集群的拓扑文件配置 · · · · · 1451
13.4.10 TiUP 镜像参考指南 · · · · · 1459
13.4.11 TiUP 组件文档 · · · · · 1466
13.5 TiDB Operator · · · · · 1493
13.6 TiDB Data Migration · · · · · 1493
13.6.1 TiDB Data Migration 简介 · · · · · 1493
13.6.2 Data Migration 架构 · · · · · 1495
13.6.3 TiDB Data Migration 快速上手指南 · · · · · 1497
13.6.4 DM 数据迁移最佳实践 · · · · · 1500
13.6.5 部署 DM 集群 · · · · · 1512
13.6.6 入门指南 · · · · · 1527
13.6.7 进阶教程 · · · · · 1570
13.6.8 运维管理 · · · · · 1611
13.6.9 参考手册 · · · · · 1660
13.7 TiDB Lightning · · · · · 1778
13.7.1 TiDB Lightning 简介 · · · · · 1778
13.7.2 TiDB Lightning 快速上手 · · · · · 1780
13.7.3 部署 TiDB Lightning · · · · · 1782
13.7.4 TiDB Lightning 目标数据库要求 · · · · · 1783
13.7.5 数据源 · · · · · 1785
13.7.6 Physical Import Mode · · · · · 1795
13.7.7 Logical Import Mode · · · · · 1802
13.7.8 TiDB Lightning 前置检查 · · · · · 1805

13.7.9 表库过滤 · · · · · 1812
13.7.10 TiDB Lightning 断点续传 · · · · · 1817
13.7.11 TiDB Lightning 并行导入 · · · · · 1819
13.7.12 TiDB Lightning 错误处理功能 · · · · · 1824
13.7.13 TiDB Lightning 故障处理 · · · · · 1831
13.8 使用 Dumpling 导出数据 · · · · · 1877
13.8.1 从 TiDB/MySQL 导出数据 · · · · · 1878
13.8.2 Dumpling 主要选项表 · · · · · 1884
13.9 TiCDC · · · · · 1897
13.9.1 TiCDC 简介 · · · · · 1897
13.9.2 TiCDC 安装部署与集群运维 · · · · · 1900
13.9.3 Changefeed · · · · · 1904
13.9.4 监控告警 · · · · · 1935
13.9.5 TiCDC 故障处理 · · · · · 2033
13.9.6 TiCDC 常见问题解答 · · · · · 2037
13.9.7 TiCDC 术语表 · · · · · 2046
13.10 TiDB Binlog · · · · · 2047
13.10.1 TiDB Binlog 简介 · · · · · 2047
13.10.2 TiDB Binlog 教程 · · · · · 2049
13.10.3 TiDB Binlog 集群部署 · · · · · 2058
13.10.4 TiDB Binlog 集群运维 · · · · · 2065
13.10.5 TiDB Binlog 配置说明 · · · · · 2068
13.10.6 TiDB Binlog 版本升级方法 · · · · · 2083
13.10.7 TiDB Binlog 集群监控 · · · · · 2086
13.10.8 Reparo 使用文档 · · · · · 2089
13.10.9 binlogctl 工具 · · · · · 2092
13.10.10 Binlog Consumer Client 用户文档 · · · · · 2094
13.10.11 TiDB Binlog Relay Log · · · · · 2098
13.10.12 集群间双向同步 · · · · · 2099
13.10.13 TiDB Binlog 术语表 · · · · · 2103
13.10.14 故障诊断 · · · · · 2104
13.10.15 TiDB Binlog 常见问题 · · · · · 2105

13.11 PingCAP Clinic 诊断服务 ····· 2111
13.11.1 PingCAP Clinic 诊断服务简介 ····· 2111
13.11.2 PingCAP Clinic 快速上手指南 ····· 2113
13.11.3 使用 PingCAP Clinic 诊断集群 ····· 2116
13.11.4 使用 PingCAP Clinic 生成诊断报告 ····· 2124
13.11.5 使用 PingCAP Clinic Diag 采集 SQL 查询计划信息 ····· 2128
13.11.6 PingCAP Clinic 数据采集说明 ····· 2132
13.12 TiSpark ····· 2137
13.12.1 TiSpark 用户指南 ····· 2137
13.13 sync-diff-inspector ····· 2151
13.13.1 sync-diff-inspector 用户文档 ····· 2151
13.13.2 不同库名或表名的数据校验 ····· 2158
13.13.3 分库分表场景下的数据校验 ····· 2159
13.13.4 TiDB 主从集群的数据校验 ····· 2163
13.13.5 基于 DM 同步场景下的数据校验 ····· 2165
13.14 TiUniManager ····· 2165
13.14.1 TiUniManager 概览 ····· 2165
13.14.2 TiUniManager 安装和运维指南 ····· 2169
13.14.3 TiUniManager 快速操作指南 ····· 2183
13.14.4 操作指南 ····· 2187
13.14.5 TiUniManager 常见问题 ····· 2208
13.14.6 发布版本历史 ····· 2212

14 参考指南 2217

14.1 架构 ····· 2217
14.1.1 TiDB 整体架构 ····· 2217
14.1.2 TiDB 数据库的存储 ····· 2219
14.1.3 TiDB 数据库的计算 ····· 2223
14.1.4 TiDB 数据库的调度 ····· 2229

14.2 存储引擎 TiKV · · · · · 2234
14.2.1 TiKV 简介 · · · · · 2234
14.2.2 RocksDB 简介 · · · · · 2235
14.2.3 Titan 介绍 · · · · · 2237
14.2.4 Titan 配置 · · · · · 2244
14.2.5 Partitioned Raft KV · · · · · 2247
14.3 存储引擎 TiFlash · · · · · 2248
14.3.1 TiFlash 简介 · · · · · 2248
14.3.2 构建 TiFlash 副本 · · · · · 2251
14.3.3 使用 TiDB 读取 TiFlash · · · · · 2256
14.3.4 使用 TiSpark 读取 TiFlash · · · · · 2259
14.3.5 使用 MPP 模式 · · · · · 2259
14.3.6 TiFlash 存算分离架构与 S3 支持 · · · · · 2264
14.3.7 使用 FastScan 功能 · · · · · 2269
14.3.8 TiFlash 支持的计算下推 · · · · · 2271
14.3.9 TiFlash 查询结果物化 · · · · · 2278
14.3.10 延迟物化 · · · · · 2280
14.3.11 TiFlash 数据落盘 · · · · · 2282
14.3.12 TiFlash 数据校验 · · · · · 2284
14.3.13 TiFlash 兼容性说明 · · · · · 2285
14.4 系统变量 · · · · · 2286
14.4.1 变量参考 · · · · · 2287
14.5 配置文件参数 · · · · · 2393
14.5.1 TiDB 配置文件描述 · · · · · 2393
14.5.2 TiKV 配置文件描述 · · · · · 2418
14.5.3 TiFlash 配置参数 · · · · · 2475
14.5.4 PD 配置文件描述 · · · · · 2483
14.6 CLI · · · · · 2496
14.6.1 TiKV Control 使用说明 · · · · · 2496
14.6.2 PD Control 使用说明 · · · · · 2510
14.6.3 TiDB Control 使用说明 · · · · · 2536
14.6.4 PD Recover 使用文档 · · · · · 2543

14.7 命令行参数 · · · · · 2546
14.7.1 TiDB 配置参数 · · · · · 2546
14.7.2 TiKV 配置参数 · · · · · 2552
14.7.3 TiFlash 命令行参数 · · · · · 2554
14.7.4 PD 配置参数 · · · · · 2556
14.8 监控指标 · · · · · 2559
14.8.1 Overview 面板重要监控指标详解 · · · · · 2559
14.8.2 Performance Overview 面板重要监控指标详解 · · · · · 2561
14.8.3 TiDB 重要监控指标详解 · · · · · 2567
14.8.4 PD 重要监控指标详解 · · · · · 2572
14.8.5 TiKV 监控指标详解 · · · · · 2580
14.8.6 TiFlash 集群监控 · · · · · 2600
14.8.7 TiCDC 详细监控指标 · · · · · 2602
14.8.8 资源管控 (Resource Control) 监控指标详解 · · · · · 2608
14.9 安全加固 · · · · · 2610
14.9.1 为 TiDB 客户端服务端间通信开启加密传输 · · · · · 2610
14.9.2 为 TiDB 组件间通信开启加密传输 · · · · · 2614
14.9.3 生成自签名证书 · · · · · 2618
14.9.4 静态加密 · · · · · 2620
14.9.5 为 TiDB 落盘文件开启加密 · · · · · 2628
14.9.6 日志脱敏 · · · · · 2628
14.10 权限 · · · · · 2629
14.10.1 与 MySQL 安全特性差异 · · · · · 2629
14.10.2 权限管理 · · · · · 2632
14.10.3 TiDB 用户账户管理 · · · · · 2642
14.10.4 TiDB 密码管理 · · · · · 2646
14.10.5 基于角色的访问控制 · · · · · 2655
14.10.6 TiDB 证书鉴权使用指南 · · · · · 2661
14.11 SQL · · · · · 2669
14.11.1 SQL 语言结构和语法 · · · · · 2669
14.11.2 SQL 语句 · · · · · 2707
14.11.3 数据类型 · · · · · 3048

14.11.4 函数与操作符 · · · · · 3064
14.11.5 聚簇索引 · · · · · 3103
14.11.6 约束 · · · · · 3108
14.11.7 生成列 · · · · · 3115
14.11.8 SQL 模式 · · · · · 3118
14.11.9 表属性 · · · · · 3138
14.11.10 事务 · · · · · 3141
14.11.11 视图 · · · · · 3175
14.11.12 分区表 · · · · · 3180
14.11.13 临时表 · · · · · 3216
14.11.14 缓存表 · · · · · 3222
14.11.15 外键约束 · · · · · 3227
14.11.16 字符集和排序 · · · · · 3235
14.11.17 Placement Rules in SQL · · · · · 3247
14.11.18 系统表 · · · · · 3253
14.11.19 元数据锁 · · · · · 3364
14.12 UI · · · · · 3367
14.12.1 TiDB Dashboard · · · · · 3367
14.13 遥测 · · · · · 3514
14.13.1 哪些使用情况信息会被收集？ · · · · · 3515
14.13.2 开启遥测功能 · · · · · 3516
14.13.3 禁用遥测功能 · · · · · 3518
14.13.4 查看遥测启用状态 · · · · · 3521
14.13.5 使用情况信息合规性 · · · · · 3521
14.14 错误码与故障诊断 · · · · · 3521
14.14.1 错误码 · · · · · 3521
14.14.2 故障诊断 · · · · · 3530
14.15 通过拓扑 label 进行副本调度 · · · · · 3530
14.15.1 根据集群拓扑配置 labels · · · · · 3531
14.15.2 基于拓扑 label 的 PD 调度策略 · · · · · 3535
15 常见问题解答 (FAQ) 3536

15.1 TiDB 产品常见问题解答汇总 · · · · · 3536
15.2 TiDB 产品常见问题 · · · · · 3537
15.2.1 1.1 TiDB 介绍及整体架构 · · · · · 3537
15.2.2 1.2 TiDB 原理 · · · · · 3539
15.3 SQL 操作常见问题 · · · · · 3540
15.3.1 TiDB 是否支持二级键？ · · · · · 3540
15.3.2 TiDB 在对大表执行 DDL 操作时，性能表现如何？ · · · · · 3540
15.3.3 如何选择正确的查询计划？是否需要使用优化器提示？还是可以使用提示？ · · · · · 3540
15.3.4 如何阻止特定的 SQL 语句执行（或者将某个 SQL 语句加入黑名单）? · · · · · 3540
15.3.5 TiDB 对哪些 MySQL variables 兼容？ · · · · · 3541
15.3.6 省略 ORDER BY 条件时 TiDB 中返回结果的顺序与 MySQL 中的不一致 · · · · · 3541
15.3.7 TiDB 是否支持 SELECT FOR UPDATE？ · · · · · 3542
15.3.8 TiDB 的 codec 能保证 UTF8 的字符串是 memcomparable 的吗？我们的 key 需要支持 UTF8，有什么编码建议吗？ · · · · · 3542
15.3.9 一个事务中的语句数量最大是多少？ · · · · · 3543
15.3.10 TiDB 中，为什么出现后插入数据的自增 ID 反而小？ · · · · · 3543
15.3.11 如何在 TiDB 中修改 sql_mode？ · · · · · 3543
15.3.12 用 Sqoop 批量写入 TiDB 数据，虽然配置了 --batch 选项，但还是会遇到 java.sql.BatchUpdateExecption: statement count 5001 exceeds the transaction limitation 的错误，该如何解决？ · · · · · 3543
15.3.13 TiDB 有像 Oracle 那样的 Flashback Query 功能么，DDL 支持么？ · · · · · 3543
15.3.14 TiDB 中删除数据后会立即释放空间吗？ · · · · · 3544
15.3.15 删除数据后查询速度为何会变慢？ · · · · · 3544
15.3.16 对数据做删除操作之后，空间回收比较慢，如何处理？ · · · · · 3544
15.3.17 SHOW PROCESSLIST 是否显示系统进程号？ · · · · · 3544
15.3.18 在 TiDB 中如何控制或改变 SQL 提交的执行优先级？ · · · · · 3544
15.3.19 在 TiDB 中 auto analyze 的触发策略是怎样的？ · · · · · 3545
15.3.20 可以使用 Optimizer Hints 控制优化器行为吗？ · · · · · 3545
15.3.21 DDL 执行 · · · · · 3545
15.3.22 执行 DDL 会慢的可能原因 · · · · · 3546
15.3.23 SQL 优化 · · · · · 3548
15.3.24 数据库优化 · · · · · 3550

15.4 TiDB 安装部署常见问题 · · · · · 3550
15.4.1 软硬件要求 FAQ · · · · · 3550
15.4.2 安装部署 FAQ · · · · · 3551
15.4.3 TiDB 支持在公有云上部署吗？ · · · · · 3556
15.5 迁移常见问题 · · · · · 3557
15.5.1 全量数据导出导入 · · · · · 3557
15.5.2 在线数据同步 · · · · · 3559
15.5.3 业务流量迁入 · · · · · 3559
15.6 升级与升级后常见问题 · · · · · 3561
15.6.1 升级常见问题 · · · · · 3561
15.6.2 升级后常见问题 · · · · · 3561
15.7 TiDB 监控常见问题 · · · · · 3566
15.7.1 目前的监控使用方式及主要监控指标，有没有更好看的监控？ · · · · · 3566
15.7.2 Prometheus 监控数据默认 15 天自动清除一次，可以自己设定成 2 个月或者手动删除吗？ · · · · 3566
15.7.3 Region Health 监控项 · · · · · 3567
15.7.4 Statement Count 监控项中的 selectsimplefull 是什么意思？ · · · · · 3567
15.7.5 监控上的 QPS 和 Statement OPS 有什么区别？ · · · · · 3567
15.8 TiDB 集群管理常见问题 · · · · · 3567
15.8.1 集群日常管理 · · · · · 3567
15.8.2 PD 管理 · · · · · 3570
15.8.3 TiDB server 管理 · · · · · 3571
15.8.4 TiKV 管理 · · · · · 3574
15.8.5 TiDB 测试 · · · · · 3577
15.8.6 TiDB 备份恢复 · · · · · 3577
15.9 高可用常见问题 · · · · · 3578
15.9.1 TiDB 数据是强一致的吗？ · · · · · 3578
15.9.2 TiDB 是否提供三中心跨机房多活部署的推荐方案？ · · · · · 3578
15.10 高可靠常见问题 · · · · · 3578
15.10.1 TiDB 是否支持数据加密？ · · · · · 3578
15.10.2 我们的安全漏洞扫描工具对 MySQL version 有要求，TiDB 是否支持修改 server 版本号呢？ · · · · 3578
15.10.3 TiDB 支持哪些认证协议？过程是怎样的？ · · · · · 3579
15.10.4 如何修改用户名密码和权限？ · · · · · 3579

15.11 备份与恢复常见问题 · · · · · 3579
15.11.1 当误删除或误更新数据后，如何原地快速恢复？ · · · · · 3579
15.11.2 在 TiDB v5.4.0 及后续版本中，当在有负载的集群进行备份时，备份速度为什么会变得很慢？ · 3579
15.11.3 PITR 问题 · · · · · 3580
15.11.4 功能兼容性问题 · · · · · 3582
15.11.5 进行数据恢复的问题 · · · · · 3583
15.11.6 备份恢复功能相关知识 · · · · · 3586
16 版本发布历史 3587
16.1 TiDB 版本发布历史 · · · · · 3587
16.1.1 7.0 · · · · · 3587
16.1.2 6.6 · · · · · 3587
16.1.3 6.5 · · · · · 3587
16.1.4 6.4 · · · · · 3587
16.1.5 6.3 · · · · · 3587
16.1.6 6.2 · · · · · 3588
16.1.7 6.1 · · · · · 3588
16.1.8 6.0 · · · · · 3588
16.1.9 5.4 · · · · · 3588
16.1.10 5.3 · · · · · 3588
16.1.11 5.2 · · · · · 3588
16.1.12 5.1 · · · · · 3589
16.1.13 5.0 · · · · · 3589
16.1.14 4.0 · · · · · 3589
16.1.15 3.1 · · · · · 3590
16.1.16 3.0 · · · · · 3590
16.1.17 2.1 · · · · · 3590
16.1.18 2.0 · · · · · 3591
16.1.19 1.0 · · · · · 3592

16.2 TiDB 版本发布时间线 · · · · · 3592
16.3 TiDB 版本规则 · · · · · 3596
16.3.1 版本命名 · · · · · 3596
16.3.2 长期支持版本 · · · · · 3596
16.3.3 开发里程碑版本 · · · · · 3597
16.3.4 TiDB 工具版本 · · · · · 3597
16.3.5 不再沿用的历史版本号 · · · · · 3597
16.4 TiDB 离线包 · · · · · 3598
16.4.1 延伸阅读 · · · · · 3600
16.5 v7.0 · · · · · 3600
16.5.1 TiDB 7.0.0 Release Notes · · · · · 3600
16.6 v6.6 · · · · · 3626
16.6.1 TiDB 6.6.0 Release Notes · · · · · 3626
16.7 v6.5 · · · · · 3666
16.7.1 TiDB 6.5.3 Release Notes · · · · · 3666
16.7.2 TiDB 6.5.2 Release Notes · · · · · 3669
16.7.3 TiDB 6.5.1 Release Notes · · · · · 3671
16.7.4 TiDB 6.5.0 Release Notes · · · · · 3675
16.8 v6.4 · · · · · 3707
16.8.1 TiDB 6.4.0 Release Notes · · · · · 3707
16.9 v6.3 · · · · · 3732
16.9.1 TiDB 6.3.0 Release Notes · · · · · 3732
16.10 v6.2 · · · · · 3757
16.10.1 TiDB 6.2.0 Release Notes · · · · · 3757
16.11 v6.1 · · · · · 3778
16.11.1 TiDB 6.1.7 Release Notes · · · · · 3778
16.11.2 TiDB 6.1.6 Release Notes · · · · · 3780
16.11.3 TiDB 6.1.5 Release Notes · · · · · 3782
16.11.4 TiDB 6.1.4 Release Notes · · · · · 3783
16.11.5 TiDB 6.1.3 Release Notes · · · · · 3785
16.11.6 TiDB 6.1.2 Release Notes · · · · · 3787
16.11.7 TiDB 6.1.1 Release Notes · · · · · 3789
16.11.8 TiDB 6.1.0 Release Notes · · · · · 3793

16.12 v6.0 · · · · · 3817
16.12.1 TiDB 6.0.0 Release Notes · · · · · 3817
16.13 v5.4 · · · · · 3840
16.13.1 TiDB 5.4.3 Release Notes · · · · · 3840
16.13.2 TiDB 5.4.2 Release Notes · · · · · 3842
16.13.3 TiDB 5.4.1 Release Notes · · · · · 3844
16.13.4 TiDB 5.4 Release Notes · · · · · 3848
16.14 v5.3 · · · · · 3865
16.14.1 TiDB 5.3.4 Release Notes · · · · · 3865
16.14.2 TiDB 5.3.3 Release Note · · · · · 3866
16.14.3 TiDB 5.3.2 Release Notes · · · · · 3866
16.14.4 TiDB 5.3.1 Release Notes · · · · · 3870
16.14.5 TiDB 5.3 Release Notes · · · · · 3873
16.15 v5.2 · · · · · 3887
16.15.1 TiDB 5.2.4 Release Notes · · · · · 3887
16.15.2 TiDB 5.2.3 Release Note · · · · · 3892
16.15.3 TiDB 5.2.2 Release Notes · · · · · 3892
16.15.4 TiDB 5.2.1 Release Notes · · · · · 3894
16.15.5 TiDB 5.2 Release Notes · · · · · 3895
16.16 v5.1 · · · · · 3904
16.16.1 TiDB 5.1.5 Release Notes · · · · · 3904
16.16.2 TiDB 5.1.4 Release Notes · · · · · 3907
16.16.3 TiDB 5.1.3 Release Note · · · · · 3911
16.16.4 TiDB 5.1.2 Release Notes · · · · · 3911
16.16.5 TiDB 5.1.1 Release Notes · · · · · 3914
16.16.6 TiDB 5.1 Release Notes · · · · · 3917
16.17 v5.0 · · · · · 3928
16.17.1 TiDB 5.0.6 Release Notes · · · · · 3928
16.17.2 TiDB 5.0.5 Release Note · · · · · 3931
16.17.3 TiDB 5.0.4 Release Notes · · · · · 3932
16.17.4 TiDB 5.0.3 Release Notes · · · · · 3936
16.17.5 TiDB 5.0.2 Release Notes · · · · · 3939

16.17.6 TiDB 5.0.1 Release Notes · · · · · 3941
16.17.7 What’s New in TiDB 5.0 · · · · · 3943
16.17.8 TiDB 5.0 RC Release Notes · · · · · 3955
16.18 v4.0 · · · · · 3960
16.18.1 TiDB 4.0.16 Release Notes · · · · · 3960
16.18.2 TiDB 4.0.15 Release Notes · · · · · 3962
16.18.3 TiDB 4.0.14 Release Notes · · · · · 3965
16.18.4 TiDB 4.0.13 Release Notes · · · · · 3968
16.18.5 TiDB 4.0.12 Release Notes · · · · · 3971
16.18.6 TiDB 4.0.11 Release Notes · · · · · 3974
16.18.7 TiDB 4.0.10 Release Notes · · · · · 3978
16.18.8 TiDB 4.0.9 Release Notes · · · · · 3980
16.18.9 TiDB 4.0.8 Release Notes · · · · · 3985
16.18.10 TiDB 4.0.7 Release Notes · · · · · 3988
16.18.11 TiDB 4.0.6 Release Notes · · · · · 3990
16.18.12 TiDB 4.0.5 Release Notes · · · · · 3994
16.18.13 TiDB 4.0.4 Release Notes · · · · · 3997
16.18.14 TiDB 4.0.3 Release Notes · · · · · 3997
16.18.15 TiDB 4.0.2 Release Notes · · · · · 4001
16.18.16 TiDB 4.0.1 Release Notes · · · · · 4004
16.18.17 TiDB 4.0 GA Release Notes · · · · · 4005
16.18.18 TiDB 4.0 RC.2 Release Notes · · · · · 4008
16.18.19 TiDB 4.0 RC.1 Release Notes · · · · · 4012
16.18.20 TiDB 4.0 RC Release Notes · · · · · 4015
16.18.21 TiDB 4.0.0 Beta.2 Release Notes · · · · · 4017
16.18.22 TiDB 4.0.0 Beta.1 Release Notes · · · · · 4019
16.18.23 TiDB 4.0 Beta Release Notes · · · · · 4021
16.19 v3.1 · · · · · 4024
16.19.1 TiDB 3.1.2 Release Notes · · · · · 4024
16.19.2 TiDB 3.1.1 Release Notes · · · · · 4024
16.19.3 TiDB 3.1 GA Release Notes · · · · · 4025
16.19.4 TiDB 3.1 RC Release Notes · · · · · 4027

16.19.5 TiDB 3.1 Beta.2 Release Notes · · · · · 4029
16.19.6 TiDB 3.1 Beta.1 Release Notes · · · · · 4031
16.19.7 TiDB 3.1 Beta Release Notes · · · · · 4032
16.20 v3.0 · · · · · 4032
16.20.1 TiDB 3.0.20 Release Notes · · · · · 4032
16.20.2 TiDB 3.0.19 Release Notes · · · · · 4034
16.20.3 TiDB 3.0.18 Release Notes · · · · · 4035
16.20.4 TiDB 3.0.17 Release Notes · · · · · 4035
16.20.5 TiDB 3.0.16 Release Notes · · · · · 4036
16.20.6 TiDB 3.0.15 Release Notes · · · · · 4037
16.20.7 TiDB 3.0.14 Release Notes · · · · · 4038
16.20.8 TiDB 3.0.13 Release Notes · · · · · 4041
16.20.9 TiDB 3.0.12 Release Notes · · · · · 4041
16.20.10 TiDB 3.0.11 Release Notes · · · · · 4042
16.20.11 TiDB 3.0.10 Release Notes · · · · · 4044
16.20.12 TiDB 3.0.9 Release Notes · · · · · 4045
16.20.13 TiDB 3.0.8 Release Notes · · · · · 4047
16.20.14 TiDB 3.0.7 Release Notes · · · · · 4050
16.20.15 TiDB 3.0.6 Release Notes · · · · · 4050
16.20.16 TiDB 3.0.5 Release Notes · · · · · 4053
16.20.17 TiDB 3.0.4 Release Notes · · · · · 4055
16.20.18 TiDB 3.0.3 Release Notes · · · · · 4059
16.20.19 TiDB 3.0.2 Release Notes · · · · · 4060
16.20.20 TiDB 3.0.1 Release Notes · · · · · 4064
16.20.21 TiDB 3.0 GA Release Notes · · · · · 4066
16.20.22 TiDB 3.0.0-rc.3 Release Notes · · · · · 4072
16.20.23 TiDB 3.0.0-rc.2 Release Notes · · · · · 4075
16.20.24 TiDB 3.0.0-rc.1 Release Notes · · · · · 4077
16.20.25 TiDB 3.0.0 Beta.1 Release Notes · · · · · 4081
16.20.26 TiDB 3.0 Beta Release Notes · · · · · 4083

16.21 v2.1 · · · · · 4086
16.21.1 TiDB 2.1.19 Release Notes · · · · · 4086
16.21.2 TiDB 2.1.18 Release Notes · · · · · 4088
16.21.3 TiDB 2.1.17 Release Notes · · · · · 4090
16.21.4 TiDB 2.1.16 Release Notes · · · · · 4093
16.21.5 TiDB 2.1.15 Release Notes · · · · · 4094
16.21.6 TiDB 2.1.14 Release Notes · · · · · 4096
16.21.7 TiDB 2.1.13 Release Notes · · · · · 4097
16.21.8 TiDB 2.1.12 Release Notes · · · · · 4098
16.21.9 TiDB 2.1.11 Release Notes · · · · · 4098
16.21.10 TiDB 2.1.10 Release Notes · · · · · 4099
16.21.11 TiDB 2.1.9 Release Notes · · · · · 4101
16.21.12 TiDB 2.1.8 Release Notes · · · · · 4102
16.21.13 TiDB 2.1.7 Release Notes · · · · · 4104
16.21.14 TiDB 2.1.6 Release Notes · · · · · 4104
16.21.15 TiDB 2.1.5 Release Notes · · · · · 4105
16.21.16 TiDB 2.1.4 Release Notes · · · · · 4107
16.21.17 TiDB 2.1.3 Release Notes · · · · · 4107
16.21.18 TiDB 2.1.2 Release Notes · · · · · 4109
16.21.19 TiDB 2.1.1 Release Notes · · · · · 4110
16.21.20 TiDB 2.1 GA Release Notes · · · · · 4111
16.21.21 TiDB 2.1 RC5 Release Notes · · · · · 4114
16.21.22 TiDB 2.1 RC4 Release Notes · · · · · 4116
16.21.23 TiDB 2.1 RC3 Release Notes · · · · · 4117
16.21.24 TiDB 2.1 RC2 Release Notes · · · · · 4119
16.21.25 TiDB 2.1 RC1 Release Notes · · · · · 4121
16.21.26 TiDB 2.1 Beta Release Notes · · · · · 4125
16.22 v2.0 · · · · · 4127
16.22.1 TiDB 2.0.11 Release Notes · · · · · 4127
16.22.2 TiDB 2.0.10 Release Notes · · · · · 4127
16.22.3 TiDB 2.0.9 Release Notes · · · · · 4128
16.22.4 TiDB 2.0.8 Release Notes · · · · · 4129

16.22.5 TiDB 2.0.7 Release Notes · · · · · 4130
16.22.6 TiDB 2.0.6 Release Notes · · · · · 4131
16.22.7 TiDB 2.0.5 Release Notes · · · · · 4132
16.22.8 TiDB 2.0.4 Release Notes · · · · · 4133
16.22.9 TiDB 2.0.3 Release Notes · · · · · 4133
16.22.10 TiDB 2.0.2 Release Notes · · · · · 4134
16.22.11 TiDB 2.0.1 Release Notes · · · · · 4135
16.22.12 TiDB 2.0 Release Notes · · · · · 4136
16.22.13 TiDB 2.0 RC5 Release Notes · · · · · 4140
16.22.14 TiDB 2.0 RC4 Release Notes · · · · · 4141
16.22.15 TiDB 2.0 RC3 Release Notes · · · · · 4141
16.22.16 TiDB 2.0 RC1 Release Notes · · · · · 4143
16.22.17 TiDB 1.1 Beta Release Notes · · · · · 4143
16.22.18 TiDB 1.1 Alpha Release Notes · · · · · 4145
16.23 v1.0 · · · · · 4146
16.23.1 TiDB 1.0 Release Notes · · · · · 4146
16.23.2 TiDB Pre-GA Release Notes · · · · · 4152
16.23.3 TiDB RC4 Release Notes · · · · · 4153
16.23.4 TiDB RC3 Release Notes · · · · · 4154
16.23.5 TiDB RC2 Release Notes · · · · · 4155
16.23.6 TiDB RC1 Release Notes · · · · · 4156

17 术语表 4157

17.1 A · · · · · 4157
17.1.1 ACID · · · · · 4157
17.2 B · · · · · 4158
17.2.1 BR · · · · · 4158
17.2.2 Batch Create Table · · · · · 4158
17.2.3 Baseline Capturing · · · · · 4158
17.2.4 Bucket · · · · · 4158
17.3 C · · · · · 4158
17.3.1 Cached Table · · · · · 4158
17.3.2 Continuous Profiling · · · · · 4158

17.4 D · · · · 4158
17.4.1 Dynamic Pruning · · · · 4158
17.5 I · · · · 4159
17.5.1 Index Merge · · · · 4159
17.5.2 In-Memory Pessimistic Lock · · · · 4159
17.6 L · · · · 4159
17.6.1 Leader/Follower/Learner · · · · 4159
17.7 O · · · · 4159
17.7.1 Old value · · · · 4159
17.7.2 Operator · · · · 4159
17.7.3 Operator Step · · · · 4159
17.8 P · · · · 4160
17.8.1 Pending/Down · · · · 4160
17.8.2 Point get · · · · 4160
17.8.3 Predicate columns · · · · 4160
17.9 Q · · · · 4160
17.9.1 Quota Limiter · · · · 4160
17.10 R · · · · 4160
17.10.1 Raft Engine · · · · 4160
17.10.2 Region/Peer/Raft Group · · · · 4160
17.10.3 Region Split · · · · 4161
17.10.4 Restore · · · · 4161
17.11 S · · · · 4161
17.11.1 Scheduler · · · · 4161
17.11.2 Store · · · · 4161
17.12 T · · · · 4161
17.12.1 Top SQL · · · · 4161
17.12.2 TSO · · · · 4161
17.12.3 TTL · · · · 4162

1 文档中心

2 关于 TiDB

2.1 TiDB 简介

TiDB 是 PingCAP 公司自主设计、研发的开源分布式关系型数据库，是一款同时支持在线事务处理与在线分析处理 (Hybrid Transactional and Analytical Processing, HTAP) 的融合型分布式数据库产品，具备水平扩容或者缩容、金融级高可用、实时 HTAP、云原生的分布式数据库、兼容 MySQL 5.7 协议和 MySQL 生态等重要特性。目标是为用户提供一站式 OLTP (Online Transactional Processing)、OLAP (Online Analytical Processing)、HTAP 解决方案。TiDB 适合高可用、强一致要求较高、数据规模较大等各种应用场景。

关于 TiDB 的关键技术创新，请观看以下视频。

2.1.1 五大核心特性

- 一键水平扩容或者缩容

  得益于 TiDB 存储计算分离的架构的设计，可按需对计算、存储分别进行在线扩容或者缩容，扩容或者缩容过程中对应用运维人员透明。

- 金融级高可用

  数据采用多副本存储，数据副本通过 Multi-Raft 协议同步事务日志，多数派写入成功事务才能提交，确保数据强一致性且少数副本发生故障时不影响数据的可用性。可按需配置副本地理位置、副本数量等策略满足不同容灾级别的要求。

- 实时 HTAP

  提供行存储引擎TiKV、列存储引擎TiFlash 两款存储引擎，TiFlash 通过 Multi-Raft Learner 协议实时从 TiKV 复制数据，确保行存储引擎 TiKV 和列存储引擎 TiFlash 之间的数据强一致。TiKV、TiFlash 可按需部署在不同的机器，解决 HTAP 资源隔离的问题。

- 云原生的分布式数据库

  专为云而设计的分布式数据库，通过 TiDB Operator 可在公有云、私有云、混合云中实现部署工具化、自动化。

- 兼容 MySQL 5.7 协议和 MySQL 生态

  兼容 MySQL 5.7 协议、MySQL 常用的功能、MySQL 生态，应用无需或者修改少量代码即可从 MySQL 迁移到 TiDB。提供丰富的数据迁移工具帮助应用便捷完成数据迁移。

2.1.2 四大核心应用场景

- 对数据一致性及高可靠、系统高可用、可扩展性、容灾要求较高的金融行业属性的场景

  众所周知，金融行业对数据一致性及高可靠、系统高可用、可扩展性、容灾要求较高。传统的解决方案是同城两个机房提供服务、异地一个机房提供数据容灾能力但不提供服务，此解决方案存在以下缺点：资源利用率低、维护成本高、RTO (Recovery Time Objective) 及 RPO (Recovery Point Objective) 无法真实达到企业

所期望的值。TiDB 采用多副本 + Multi-Raft 协议的方式将数据调度到不同的机房、机架、机器，当部分机器出现故障时系统可自动进行切换，确保系统的 RTO <= 30s 及 RPO = 0。

- 对存储容量、可扩展性、并发要求较高的海量数据及高并发的 OLTP 场景

  随着业务的高速发展，数据呈现爆炸性的增长，传统的单机数据库无法满足因数据爆炸性的增长对数据库的容量要求，可行方案是采用分库分表的中间件产品或者 NewSQL 数据库替代、采用高端的存储设备等，其中性价比最大的是 NewSQL 数据库，例如：TiDB。TiDB 采用计算、存储分离的架构，可对计算、存储分别进行扩容和缩容，计算最大支持 512 节点，每个节点最大支持 1000 并发，集群容量最大支持 PB 级别。

- Real-time HTAP 场景

  随着 5G、物联网、人工智能的高速发展，企业所生产的数据会越来越多，其规模可能达到数百 TB 甚至 PB 级别，传统的解决方案是通过 OLTP 型数据库处理在线联机交易业务，通过 ETL 工具将数据同步到 OLAP 型数据库进行数据分析，这种处理方案存在存储成本高、实时性差等多方面的问题。TiDB 在 4.0 版本中引入列存储引擎 TiFlash 结合行存储引擎 TiKV 构建真正的 HTAP 数据库，在增加少量存储成本的情况下，可以在同一个系统中做联机交易处理、实时数据分析，极大地节省企业的成本。

- 数据汇聚、二次加工处理的场景

  当前绝大部分企业的业务数据都分散在不同的系统中，没有一个统一的汇总，随着业务的发展，企业的决策层需要了解整个公司的业务状况以便及时做出决策，故需要将分散在各个系统的数据汇聚在同一个系统并进行二次加工处理生成 T+0 或 T+1 的报表。传统常见的解决方案是采用 ETL + Hadoop 来完成，但 Hadoop 体系太复杂，运维、存储成本太高无法满足用户的需求。与 Hadoop 相比，TiDB 就简单得多，业务通过 ETL 工具或者 TiDB 的同步工具将数据同步到 TiDB，在 TiDB 中可通过 SQL 直接生成报表。

关于 TiDB 典型应用场景和用户案例的介绍，请观看以下视频。

2.1.3 另请参阅

- TiDB 整体架构
- TiDB 数据库的存储
- TiDB 数据库的计算
- TiDB 数据库的调度

2.2 TiDB 7.0.0 Release Notes

发版日期：2023 年 3 月 30 日

TiDB 版本：7.0.0

试用链接：快速体验 | 下载离线包

在 7.0.0 版本中，你可以获得以下关键特性：

```
<th>分类</th>
<th>功能</th>
<th>描述</th>
```

```
<td rowspan="2">可扩展性与性能</td>
<td>会话级别内<a href="https://docs.pingcap.com/zh/tidb/v7.0/sql-non-prepared-plan-cache" target
    ↪ ="_blank">无需手动准备 SQL 执行计划缓存</a>（实验特性）</td>
<td>支持在会话级别自动重用执行计划缓存，可以减少编译并缩短相同 SQL 查询的时间，而无需事先手动准备
    ↪  Prepare Statement 语句。</td>
```

```
<td>TiFlash 支持<a href="https://docs.pingcap.com/zh/tidb/v7.0/tiflash-disaggregated-and-s3"
    ↪ target="_blank">存储计算分离和 S3 共享存储</a>（实验特性）</td>
<td>TiFlash 增加云原生架构的支持作为可选项：
  <ul>
    <li>支持存算分离架构，提升 HTAP 资源的弹性能力。</li>
    <li>支持基于 S3 的存储引擎，以更低的成本提供共享存储。</li>
  </ul>
</td>
```

```
<td rowspan="2">稳定性与高可用</td>
<td><a href="https://docs.pingcap.com/zh/tidb/v7.0/tidb-resource-control" target="_blank">
    ↪ 增强了资源管控</a>（实验特性）</td>
<td>支持使用资源组来为一个集群中的不同应用或工作负载分配和隔离资源。在这个版本中，TiDB
    ↪ 增加了对不同资源的绑定模式（用户级、会话级、语句级）和用户定义的优先级的支持，
    ↪ 你还可以使用命令来对集群整体资源量进行预估。</td>
```

```
<td>TiFlash 支持<a href="https://docs.pingcap.com/zh/tidb/v7.0/tiflash-spill-disk" target="_blank
    ↪ ">数据落盘</a></td>
<td>TiFlash 支持将中间结果落盘，以缓解数据密集型操作（如聚合、排序和 Hash Join）中的 OOM 问题。</
    ↪ td>
```

```
<td rowspan="2">SQL</td>
<td><a href="https://docs.pingcap.com/zh/tidb/v7.0/time-to-live" target="_blank">行级 TTL</a> (GA
    ↪ )</td>
<td>支持通过后台任务自动删除超过生命周期（Time to live）的数据，
    ↪ 并以此来自动管理数据规模并提高性能。</td>
```

```
<td><a href="https://docs.pingcap.com/zh/tidb/v7.0/partitioned-table#重组分区" target="_blank">
    ↪ 支持 <code>REORGANIZE PARTITION</code> 语法（List/Range 分区表）</a></td>
<td><code>REORGANIZE PARTITION</code> 语句可用于合并相邻分区，或将一个分区拆分为多个分区，
    ↪ 从而提升分区表的易用性。</td>
```

```
<td rowspan="2">数据库管理与可观测性<br/></td>
<td><a href="https://docs.pingcap.com/zh/tidb/v7.0/sql-statement-load-data" target="_blank"><code
    ↪ >LOAD DATA</code> 语句功能增强</a>（实验特性）</td>
<td><code>LOAD DATA</code> 语句功能增强，例如支持从 S3/GCS 导入数据。</td>
```

```
<td>TiCDC 支持<a href="https://docs.pingcap.com/zh/tidb/v7.0/ticdc-sink-to-cloud-storage" target
    ↪ ="_blank">对象存储 Sink</a> (GA)</td>
<td>TiCDC 支持将行变更事件同步到对象存储服务，包括 Amazon S3、GCS、Azure Blob Storage 和 NFS 等
    ↪ 。</td>
```

2.2.1 功能详情

2.2.1.1 可扩展性

- TiFlash 支持存算分离和对象存储（实验特性）#6882 @flowbehappy @JaySon-Huang @breezewish @JinheLin @lidezhu @CalvinNeo

  在 v7.0.0 之前的版本中，TiFlash 为存算一体架构。在此架构下，TiFlash 节点既是存储节点，也是计算节点，TiFlash 的计算能力和存储能力无法独立扩展；同时，TiFlash 节点只能使用本地存储。

  从 v7.0.0 起，TiFlash 新增存算分离架构。在此架构下，TiFlash 节点分为 Compute Node（计算节点）和 Write Node（写入节点）两种类型，并支持兼容 S3 API 的对象存储。这两种节点都可以单独扩缩容，独立调整计算或数据存储能力。TiFlash 的存算分离架构和存算一体架构不能混合使用、相互转换，需要在部署 TiFlash 时进行相应的配置指定使用其中的一种架构。

  更多信息，请参考用户文档。

2.2.1.2 性能

- 实现 Fast Online DDL 和 PITR 的兼容 #38045 @Leavrth

  TiDB v6.5.0 中Fast Online DDL 功能和PITR 未完全兼容。在使用 TiDB v6.5.0 时，建议先停止 PITR 后台备份任务，以 Fast Online DDL 方式快速添加索引，然后再启动 PITR 备份任务实现全量数据备份。

  从 TiDB v7.0.0 开始，Fast Online DDL 和 PITR 已经完全兼容。通过 PITR 恢复集群数据时，系统将自动回放日志备份期间记录的通过 Fast Online DDL 方式添加的索引操作，达到兼容效果。

  更多信息，请参考用户文档。

- TiFlash 引擎支持 Null-Aware Semi Join 和 Null-Aware Anti Semi Join 算子 #6674 @gengliqi

  当使用 `IN`、`NOT IN`、`= ANY` 或 `!= ALL` 算子引导的关联子查询时，TiDB 会将其转化为 Semi Join 或 Anti Semi Join，从而提升计算性能。如果转换后的 Join key 列可能为 `NULL`，则需要具有 Null-Aware 特性的 Join 算法，即Null-Aware Semi Join 和Null-Aware Anti Semi Join 算子。

  在 v7.0.0 之前的版本中，TiFlash 引擎不支持 Null-Aware Semi Join 和 Null-Aware Anti Semi Join 算子，因此无法将这些子查询直接下推至 TiFlash 引擎进行计算。从 TiDB v7.0.0 开始，TiFlash 引擎支持了 Null-Aware Semi Join 和 Null-Aware Anti Semi Join 算子。如果 SQL 包含这几种关联子查询，查询的表包含 TiFlash 副本，并且启用了MPP 模式，优化器将自动判断是否将 Null-Aware Semi Join 和 Null-Aware Anti Semi Join 算子下推至 TiFlash 引擎进行计算以提升整体性能。

  更多信息，请参考用户文档。

- TiFlash 引擎支持 FastScan 功能 (GA) #5252 @hongyunyan

  从 v6.3.0 开始，TiFlash 引擎引入了快速扫描功能 (FastScan) 作为实验特性。在 v7.0.0 中，该功能正式 GA。你可以使用系统变量`tiflash_fastscan` 启用快速扫描功能。通过牺牲强一致性保证，该功能可以大幅提

升扫表性能。如果对应的表仅进行 INSERT 操作，而没有 UPDATE/DELETE 操作，则快速扫描功能在提升扫表性能的同时不会损失强一致性。

更多信息，请参考用户文档。

- TiFlash 查询支持延迟物化功能（实验特性）#5829 @Lloyd-Pottiger

  当 SELECT 语句中包含过滤条件（WHERE 子句）时，TiFlash 默认会先读取该查询所需列的全部数据，然后再根据查询条件对数据进行过滤、聚合等计算任务。延迟物化是一种优化方式，它支持下推部分过滤条件到 TableScan 算子，即先扫描过滤条件相关的列数据，过滤得到符合条件的行后，再扫描这些行的其他列数据，继续后续计算，从而减少 IO 扫描和数据处理的计算量。

  TiFlash 延迟物化默认关闭，可以通过将系统变量tidb_opt_enable_late_materialization 设置为 ON 开启。开启后，TiDB 优化器会根据统计信息和查询的过滤条件，决定哪些过滤条件会被下推到 TableScan 算子。

  更多信息，请参考用户文档。

- 支持缓存非 Prepare 语句的执行计划（实验特性）#36598 @qw4990

  执行计划缓存是提高并发 OLTP 负载能力的重要手段，TiDB 已经支持Prepare 语句的计划缓存。在 v7.0.0 中，非 Prepare 语句的执行计划也能够被缓存，使执行计划缓存能够应用于更广泛的场景，从而提升 TiDB 的并发处理能力。

  这个功能目前默认关闭，你可以通过系统变量tidb_enable_non_prepared_plan_cache 打开。出于稳定性考虑，在当前版本中，TiDB 开辟了一块新的区域用于缓存非 Prepare 语句的执行计划，你可以通过系统变量tidb_non_prepared_plan_cache_size 设置缓存大小。此外，该功能对 SQL 的模式也有一定的限制，具体参见使用限制。

  更多信息，请参考用户文档。

- 解除执行计划缓存对子查询的限制 #40219 @fzzf678

  TiDB v7.0.0 移除了计划缓存对子查询的限制。带有子查询的 SQL 语句的执行计划可以被缓存，例如 `SELECT * FROM t WHERE a > (SELECT ...)`。这进一步扩大了执行计划缓存的应用范围，提高了 SQL 的执行效率。

  更多信息，请参考用户文档。

- TiKV 支持自动生成空的日志文件用于日志回收 #14371 @LykxSassinator

  TiKV 在 v6.3.0 中引入了Raft 日志回收特性，用以减少写负载的长尾延迟。但是，日志回收需要在 Raft 日志文件数量达到一定阈值后才能生效，使得用户无法直观感受到该特性对写负载吞吐的提升。

  为了改善用户体验，v7.0.0 正式引入了raft-engine.prefill-for-recycle 配置项，用以控制 TiKV 是否在进程启动时自动生成空的日志文件用于日志回收。该配置项启用时，TiKV 将在初始化时自动填充一批空日志文件用于日志回收，保证日志回收在初始化后立即生效。

  更多信息，请参考用户文档。

- 支持从窗口函数中推导 TopN 或 Limit 的优化规则，提升窗口函数的性能 #13936 @windtalker

  该功能默认关闭，需要将 session 变量tidb_opt_derive_topn 设置为 ON 开启。

  更多信息，请参考用户文档。

- 支持通过 Fast Online DDL 创建唯一索引 #40730 @tangenta

TiDB v6.5.0 支持通过Fast Online DDL 方式创建普通的二级索引。从 v7.0.0 开始，TiDB 支持通过 Fast Online DDL 方式创建唯一索引。相比于 TiDB v6.1.0，大表添加唯一索引的性能预期将提升数倍。

更多信息，请参考用户文档。

2.2.1.3 稳定性

- 增强了资源管控特性 (实验特性) #38825 @nolouch @BornChanger @glorv @tiancaiamao @Connor1996 @JmPotato @hnes @CabinfeverB @HuSharp

  TiDB 优化了基于资源组的资源管控特性。该特性将会极大地提升 TiDB 集群的资源利用效率和性能表现。资源管控特性的引入对 TiDB 具有里程碑的意义，你可以将一个分布式数据库集群划分成多个逻辑单元，将不同的数据库用户映射到对应的资源组中，并根据需要设置每个资源组的配额。当集群资源紧张时，来自同一个资源组的会话所使用的全部资源将被限制在配额内，避免其中一个资源组过度消耗，从而影响其他资源组中的会话正常运行。

  该特性也可以将多个来自不同系统的中小型应用合入一个 TiDB 集群中，个别应用的负载提升，不会影响其他应用的正常运行。而在系统负载较低的时候，繁忙的应用即使超过设定的读写配额，也仍然可以被分配到所需的系统资源，达到资源的最大化利用。此外，合理利用资源管控特性可以减少集群数量，降低运维难度及管理成本。

  该特性不仅提供了 Grafana 内置的 Resource Control Dashboard 展示资源的实际使用情况，协助你更合理地配置资源，还支持基于会话和语句级别（Hint）的动态资源管控能力。这些功能的引入将帮助你更精确地掌控 TiDB 集群的资源使用情况，并根据实际需要动态调整配额。

  在 v7.0.0 中，支持为资源组设置绝对的调度优先级 (PRIORITY)，保障重要业务能够获取到资源，同时也扩展了资源组的设置方式。

  你可以通过以下方式使用资源组：

  - 用户级别。通过`CREATE USER` 或`ALTER USER` 语句将用户绑定到特定的资源组。绑定后，对应的用户新创建的会话会自动绑定对应的资源组。
  - 会话级别。通过`SET RESOURCE GROUP` 设置当前会话的资源组。
  - 语句级别。通过`RESOURCE_GROUP()` 设置当前语句的资源组。

  更多信息，请参考用户文档。

- 支持 Fast Online DDL 的检查点机制，提升容错性和自动恢复能力 #42164 @tangenta

  TiDB v7.0.0 引入Fast Online DDL 的检查点机制，大幅提升 Fast Online DDL 的容错性和自动恢复能力。通过周期性记录并同步 DDL 进度，即使在 TiDB DDL Owner 故障重启或切换时，正在执行中的 DDL 仍能以 Fast Online DDL 方式继续执行，从而让 DDL 执行更加稳定高效。

  更多信息，请参考用户文档。

- TiFlash 引擎支持数据落盘 (Spill to disk) #6528 @windtalker

  为了提高执行性能，TiFlash 引擎尽可能将数据全部放在内存中运行。当数据量超过内存总大小时，TiFlash 会终止查询，避免内存耗尽导致系统崩溃。因此，TiFlash 可处理的数据量受限于可用的内存大小。

  从 v7.0.0 开始，TiFlash 引擎支持数据落盘功能，通过调整算子内存使用阈值`tidb_max_bytes_before_tiflash_external_grou`
  `↪ `、`tidb_max_bytes_before_tiflash_external_sort`、`tidb_max_bytes_before_tiflash_external_join`，控制对应算子的最大内存使用量。当算子使用内存超过一定阈值时，会自动将数据落盘，牺牲一定的性能，从而处理更多数据。

  更多信息，请参考用户文档。

- 提升统计信息的收集效率 #41930 @xuyifangreeneyes

  在 v7.0.0 中，TiDB 进一步优化了统计信息收集的逻辑，收集时间降低了约 25%。这一优化提升了中大型数据库集群的运行效率和稳定性，减少了统计信息收集对集群性能的影响。

- 新增优化器 Hint 对 MPP 进行干预 #39710 @Reminiscent

  TiDB 在 v7.0.0 中增加了一系列优化器 Hint，来影响 MPP 操作执行计划的生成。

  - `SHUFFLE_JOIN()`：针对 MPP 生效。提示优化器对指定表使用 Shuffle Join 算法。
  - `BROADCAST_JOIN()`：针对 MPP 生效。提示优化器对指定表使用 Broadcast Join 算法。
  - `MPP_1PHASE_AGG()`：针对 MPP 生效。提示优化器对指定查询块中所有聚合函数使用一阶段聚合算法。
  - `MPP_2PHASE_AGG()`：针对 MPP 生效。提示优化器对指定查询块中所有聚合函数使用二阶段聚合算法。

MPP 优化器 Hint 能够协助你干预 HTAP 查询，提升 HTAP 负载下的性能和稳定性。

更多信息，请参考用户文档。

- 优化器 Hint 可兼容指定连接方式与连接顺序 #36600 @Reminiscent

  在 v7.0.0 中，优化器 Hint `LEADING()` 能够和影响连接方式的 Hint 配合使用，两者行为兼容。在多表连接的情况下，可以有效指定最佳的连接方式和连接顺序，提升优化器 Hint 对执行计划的控制能力。

  新的 Hint 行为会有微小的变化。为确保向前兼容，TiDB 引入了变量`tidb_opt_advanced_join_hint`。当此变量为 OFF 时，行为向前兼容。从旧版本升级到 v7.0.0 及之后版本的集群，该变量会被设置成 OFF。为了获取更灵活的 Hint 行为，强烈建议在确保无性能回退的情况下，将该变量切换为 ON。

  更多信息，请参考用户文档。

2.2.1.4 高可用

- TiDB 支持 `prefer-leader` 选项，在网络不稳定的情况下提供更高的读可用性，降低响应延迟 #40905 @LykxSassinator

  你可以通过系统变量`tidb_replica_read` 控制 TiDB 的数据读取行为。在 v7.0.0 中，该变量新增 `prefer-leader` 选项。当设置为 `prefer-leader` 时，TiDB 会优先选择 leader 副本执行读取操作。当 leader 副本的处理速度明显变慢时，例如由于磁盘或网络性能抖动，TiDB 将选择其他可用的 follower 副本来执行读取操作，从而提高可用性并降低响应延迟。

  更多信息，请参考用户文档。

2.2.1.5 SQL 功能

- Time to live (TTL) 已基本可用 (GA) #39262 @lcwangchao @YangKeao

  TTL 提供了行级别的生命周期控制策略。在 TiDB 中，设置了 TTL 属性的表会根据配置自动检查并删除过期的行数据。TTL 的目标是帮助用户周期性且及时地清理不需要的数据，并尽量减少对用户负载的影响。

  更多信息，请参考用户文档。

- 支持 `ALTER TABLE ... REORGANIZE PARTITION` #15000 @mjonss

  TiDB 支持 `ALTER TABLE... REORGANIZE PARTITION` 语法。通过该语法，你可以对表的部分或所有分区进行重新组织，包括合并、拆分、或者其他修改，并且不丢失数据。

  更多信息，请参考用户文档。

- 支持 Key 分区 #41364 @TonsnakeLin

  TiDB 支持 Key 分区。Key 分区与 Hash 分区都可以保证将数据均匀地分散到一定数量的分区里面，区别是 Hash 分区只能根据一个指定的整数表达式或字段进行分区，而 Key 分区可以根据字段列表进行分区，且 Key 分区的分区字段不局限于整数类型。

  更多信息，请参考用户文档。

2.2.1.6 数据库管理

- TiCDC 支持同步变更数据至存储服务 (GA) #6797 @zhaoxinyu

  TiCDC 支持将变更数据同步到兼容 Amazon S3 协议的存储服务、GCS、Azure Blob Storage 以及 NFS 中。存储服务价格便宜，使用方便。对于不使用 Kafka 的用户，可以选择同步变更数据到存储服务。使用该功能，TiCDC 会将变更数据保存到文件，发送到存储服务中。用户自研的消费程序可以定时从存储服务读取新产生的变更数据进行数据处理。目前，TiCDC 支持将格式为 canal-json 和 CSV 的变更数据同步至存储服务。

  更多信息，请参考用户文档。

- TiCDC OpenAPI v2 #8019 @sdojjy

  TiCDC 提供 OpenAPI v2。相比 OpenAPI v1，OpenAPI v2 提供了更完整的同步任务支持。OpenAPI 提供的功能是`cdc cli` 工具的一个子集。你可以通过 OpenAPI v2 对 TiCDC 集群进行查询和运维操作，如获取 TiCDC 节点状态、检查集群健康状态、管理同步任务等。

  更多信息，请参考用户文档。

- DBeaver v23.0.1 默认支持 TiDB #17396 @Icemap

  - 提供独立的 TiDB 模块、Icon 和标识。
  - 默认配置支持 TiDB Serverless，你可以更方便地连接 TiDB Serverless。
  - 支持识别 TiDB 版本，从而显示或隐藏外键 Tab。
  - 支持 Explain SQL 计划显示。
  - 支持 TiDB 语法高亮，如 `PESSIMISTIC`、`OPTIMISTIC`、`AUTO_RANDOM`、`PLACEMENT`、`POLICY`、`REORGANIZE`、`EXCHANGE`、`CACHE`、`NONCLUSTERED`、`CLUSTERED` 等。
  - 支持 TiDB 函数高亮，如 `TIDB_BOUNDED_STALENESS`、`TIDB_DECODE_KEY`、`TIDB_DECODE_PLAN`、`TIDB_IS_DDL_OWNER`、`TIDB_PARSE_TSO`、`TIDB_VERSION`、`TIDB_DECODE_SQL_DIGESTS`、`TIDB_SHARD` 等。

更多信息，请参考 DBeaver 用户文档。

2.2.1.7 数据迁移

- 增强 LOAD DATA 语句功能，支持导入云存储中的数据（实验特性）#40499 @lance6716

  之前，LOAD DATA 语句只能用于导入客户端的数据文件，如果你需要从云存储导入数据，不得不借助 TiDB Lightning 来实现。但是单独部署 TiDB Lightning 又会带来额外的部署成本和管理成本。现在可直接通过LOAD DATA 语句导入云存储中的数据。功能举例说明如下：

  - 支持从 Amazon S3 和 Google Cloud Storage 导入数据到 TiDB，且支持使用通配符一次性匹配多个源文件导入到 TiDB。
  - 支持 DEFINED NULL BY 来定义 null。
  - 支持 CSV、TSV 格式的源文件。

更多信息，请参考用户文档。

- TiDB Lightning 向 TiKV 传输键值对时支持启用压缩传输 #41163 @gozssky

  自 v7.0.0 起，TiDB Lightning 正式支持将本地编码排序后的键值对在网络传输时进行压缩再发送到 TiKV，从而减少网络传输的数据量，降低 50% ~ 80% 网络带宽开销。在 v6.6.0 版本之前不支持该功能，在数据量较大的情况下，TiDB Lightning 对网络带宽要求相对较高，且会产生较高的流量费。相比 v6.6.0, v7.0.0 优化了压缩算法，能以更快的速度将数据导入到 TiKV 中。同样是开启压缩选项，v6.6.0 会增加 2 倍导入时间，而 v7.0.0 只会增加大约 60% 时间。以上的压缩率和导入时长仅供参考，不同的使用场景会有差异。

  该功能默认关闭，你可以通过将 TiDB Lightning 配置项 compress-kv-pairs 设置为 "gzip" 或者 "gz" 开启此功能。

  更多信息，请参考用户文档。

2.2.2 兼容性变更

> **注意：**
>
> 以下为从 v6.6.0 升级至当前版本 (v7.0.0) 所需兼容性变更信息。如果从 v6.5.0 或之前版本升级到当前版本，可能也需要考虑和查看中间版本 release notes 中提到的兼容性变更信息。

2.2.2.1 MySQL 兼容性

- TiDB 移除了自增列必须是索引的约束 #40580 @tiancaiamao

  在 v7.0.0 之前，TiDB 自增列的行为与 MySQL 一致，要求自增列必须是索引或索引前缀。自 TiDB v7.0.0 起，TiDB 移除了自增列必须是索引或索引前缀的限制。这意味着用户现在可以更灵活地定义表的主键，并方便地使用自增列实现排序和分页，同时避免自增列带来的写入热点问题，并通过使用聚簇索引表提升查询性能。现在，你可以使用以下语法创建表：

```
CREATE TABLE test1 (
    `id` int(11) NOT NULL AUTO_INCREMENT,
    `k` int(11) NOT NULL DEFAULT '0',
    `c` char(120) COLLATE utf8mb4_unicode_ci NOT NULL DEFAULT '',
    PRIMARY KEY(`k`, `id`)
);
```

此功能不影响 TiCDC 同步数据。

更多信息，请参考用户文档。

- TiDB 支持 Key 分区类型，如下所示。

```
CREATE TABLE employees (
id INT NOT NULL,
fname VARCHAR(30),
lname VARCHAR(30),
hired DATE NOT NULL DEFAULT '1970-01-01',
separated DATE DEFAULT '9999-12-31',
job_code INT,
store_id INT) PARTITION BY KEY(store_id) PARTITIONS 4;
```

自 TiDB v7.0.0 起，TiDB 支持 Key 分区，并支持解析 MySQL 的 PARTITION BY LINEAR KEY 语法，但会忽略其中的 LINEAR 关键字，只采用非线性 Hash 算法。Key 分区类型暂不支持分区字段列表为空的语句。

更多信息，请参考用户文档。

2.2.2.2 行为变更

- TiCDC 修复了 Avro 编码 FLOAT 类型数据错误的问题 #8490 @3AceShowHand

  在升级 TiCDC 集群到 v7.0.0 时，如果使用 Avro 同步的表包含 FLOAT 类型数据，请在升级前手动调整 Confluent Schema Registry 的兼容性策略为 None，使 changefeed 能够成功更新 schema。否则，在升级之后 changefeed 将无法更新 schema 并进入错误状态。

- 自 v7.0.0 起，tidb_dml_batch_size 对LOAD DATA 语句不再生效。

2.2.2.3 系统变量

变量名	修改类型	描述
tidb_pessimistic_txn_aggressive_locking ↪	删除	更名为tidb_pessimistic_txn_fair_locking ↪ 。

变量名	修改类型	描述
tidb_enable_non_prepared_plan_cache ↪	修改	该变量从 v7.0.0 开始生效，用来控制是否开启非 Prepare 语句执行计划缓存。
tidb_enable_null_aware_anti_join ↪	修改	经进一步的测试后，该变量默认值从 OFF 修改为 ON，表示 TiDB 默认开启对特殊集合算子 NOT IN 和 != ALL 引导的子查询产生的 Anti Join 采用 Null-Aware Hash Join 的执行方式。

变量名	修改类型	描述
tidb_enable_resource_control ↪	修改	该变量默认值从 OFF 修改为 ON，表示默认开启集群按照资源组做资源隔离。资源管控功能在 v7.0.0 默认开启，便于你随时使用此功能。
tidb_non_prepared_plan_cache_size ↪	修改	该变量从 v7.0.0 开始生效，用来控制非 Prepare 语句执行计划缓存最多能够缓存的计划数量。

变量名	修改类型	描述
tidb_rc_read_check_ts	修改	从 v7.0.0 开始，该变量对于使用 Prepared Statement 协议下 cursor fetch read 游标模式不再生效。
tidb_enable_inl_join_inner_multi_pattern	新增	该变量用于控制当内表上有 Selection /Projection 算子时是否支持 Index Join。
tidb_enable_plan_cache_for_subquery	新增	该变量用于控制 Prepared Plan Cache 是否缓存包含子查询的查询。

变量名	修改类型	描述
tidb_enable_plan_replayer_continuous_capture	新增	这个变量用来控制是否开启PLAN REPLAYER CONTINUOUS CAPTURE 功能。默认值 OFF 代表关闭功能。
tidb_load_based_replica_read_threshold	新增	该变量用于设置基于负载的 replica read 的触发阈值。在 v7.0.0，该变量控制的功能尚未完全生效，请保留默认值。

变量名	修改类型	描述
tidb_opt_advanced_join_hint ↪	新增	这个变量用来控制用于控制连接算法的 Join Method Hint 是否会影响 Join Reorder 的优化过程。默认值为 ON，即采用新的兼容控制模式；OFF 则与 v7.0.0 以前的行为保持一致。为了向前兼容，从旧版本升级到 v7.0.0 及之后版本的集群，该变量会被设置成 OFF。

变量名	修改类型	描述
tidb_opt_derive_topn ↪	新增	这个变量用来控制是否开启从窗口函数中推导 TopN 或 Limit 的优化规则。默认值为 OFF，即未开启该优化规则。
tidb_opt_enable_late_materialization ↪	新增	这个变量用来控制是否启用 TiFlash 延迟物化功能。默认值为 OFF，即未开启 TiFlash 延迟物化功能。

变量名	修改类型	描述
tidb_opt_ordering_index_selectivity_threshold ↪	新增	该变量用于当 SQL 中存在 ORDER ↪ BY 和 LIMIT 子句且带有过滤条件时，控制优化器选择索引的行为。
tidb_pessimistic_txn_fair_locking ↪	新增	是否对悲观锁启用加强的悲观锁唤醒模型，以降低单行冲突场景下事务的尾延迟。默认值为 ON，从旧版本升级到 v7.0.0 或之后版本，该变量会被设置成 OFF。

变量名	修改类型	描述
tidb_ttl_running_tasks ↪	新增	这个变量用于限制整个集群内 TTL 任务的并发量。默认值 -1 表示与 TiKV 节点的数量相同。

2.2.2.4 配置文件参数

配置文件	配置项	修改类型	描述
TiKV	server. ↪ snap- ↪ max- ↪ write- ↪ bytes- ↪ per- ↪ sec	删除	更名为server. ↪ snap- ↪ io-max ↪ -bytes ↪ -per- ↪ sec。
TiKV	raft- ↪ engine ↪ . ↪ enable ↪ -log- ↪ recycle ↪	修改	默认值由 false 变更为 true。

配置文件	配置项	修改类型	描述
TiKV	resolved-ts.advance-ts-interval	修改	默认值由 "1s" 变更为 "20s"。该修改可以延长定期推进 Resolved TS 的时间间隔，从而减少 TiKV 节点之间的流量消耗。
TiKV	resource-control.enabled	修改	默认值由 false 变更为 true。
TiKV	raft-engine.prefill-for-recycle	新增	控制 Raft Engine 是否自动生成空的日志文件用于日志回收。默认值为 false。
PD	degraded-mode-wait-duration	新增	PD 中内置的Resource Control 相关配置项。用于配置触发降级模式需要等待的时间。默认值为 "0s"。

配置文件	配置项	修改类型	描述
PD	read-base-cost	新增	PD 中内置的 Resource Control 相关配置项。用于设置每次读请求转换成 RU 的基准系数。默认值为 0.25。
PD	read-cost-per-byte	新增	PD 中内置的 Resource Control 相关配置项。用于设置读流量转换成 RU 的基准系数。默认值为 1/(64 * 1024)。
PD	read-cpu-ms-cost	新增	PD 中内置的 Resource Control 相关配置项。用于设置 CPU 转换成 RU 的基准系数。默认值为 1/3。
PD	write-base-cost	新增	PD 中内置的 Resource Control 相关配置项。用于设置每次写请求转换成 RU 的基准系数。默认值为 1。

配置文件	配置项	修改类型	描述
PD	write- ↪ cost- ↪ per- ↪ byte	新增	PD 中内置的Resource Control 相关配置项。用于设置写流量转换成 RU 的基准系数。默认值为 1/1024。
TiFlash	flash. ↪ disaggregated_mode ↪	新增	在 TiFlash 的存算分离架构中，表示此 TiFlash 节点是 Write Node 还是 Compute Node。可选值为 tiflash_write ↪ 或者 tiflash_compute ↪ 。
TiFlash	storage. ↪ s3. ↪ endpoint ↪	新增	S3 的 endpoint 地址。
TiFlash	storage. ↪ s3. ↪ bucket	新增	TiFlash 的所有数据存储的 bucket。
TiFlash	storage. ↪ s3. ↪ root	新增	S3 bucket 中数据存储的根目录。
TiFlash	storage. ↪ s3. ↪ access_key_id ↪	新增	访问 S3 的 AC-CESS_KEY_ID。

配置文件	配置项	修改类型	描述
TiFlash	storage. ↪ s3. ↪ secret_access_key ↪	新增	访问 S3 的 SECRET_ACCESS_KEY。
TiFlash	storage. ↪ remote ↪ .cache ↪ .dir	新增	TiFlash Compute Node 的本地数据缓存目录。
TiFlash	storage. ↪ remote ↪ .cache ↪ . ↪ capacity ↪	新增	TiFlash Compute Node 的本地数据缓存目录的大小。

配置文件	配置项	修改类型	描述
TiDB Lightning	`add-index-by-sql`	新增	控制 Physical Import Mode 是否通过 SQL 方式添加索引。默认为 `false`，表示 TiDB Lightning 会将行数据以及索引数据都编码成 KV pairs 后一同导入 TiKV，实现机制和历史版本保持一致。通过 SQL 方式添加索引的优点是将导入数据与导入索引分开，可以快速导入数据，即使导入数据后，索引添加失败，也不会影响数据的一致性。

配置文件	配置项	修改类型	描述
TiCDC	enable-table-across-nodes	新增	将表按 Region 个数划分成多个同步范围，这些范围可由多个 TiCDC 节点同步。
TiCDC	region-threshold	新增	开启了 enable-table-across-nodes 后，该功能只对 Region 个数大于 region-threshold 值的表生效。
DM	analyze	新增	配置是否在 CHECKSUM 结束后对所有表逐个执行 ANALYZE TABLE <table> 操作，可配置 "required" "/" optional "/"off" 。默认为 "optional "。

配置文件	配置项	修改类型	描述
DM	range-concurrency	新增	配置 dm-worker 向 TiKV 写入 KV 数据的并发数。
DM	compress-kv-pairs	新增	配置 dm-worker 向 TiKV 发送 KV 数据时是否启用压缩，可配置 "gzip"，默认为空表示不压缩。
DM	pd-addr	新增	配置 Physical Import 时连接下游 PD server 的地址，填一个或多个均可。配置项为空时，默认使用 TiDB 中查询到的 PD 地址信息。

2.2.3 改进提升

- TiDB
  - 引入 EXPAND 算子辅助优化单个 SELECT 中包含多个 DISTINCT 的 SQL 性能 #16581 @AilinKid
  - Index Join 支持更多的 SQL 格式 #40505 @Yisaer
  - 避免某些情况下分区表数据需要在 TiDB 全局排序 #26166 @Defined2014
  - 支持同时使用 fair lock mode 和 lock only if exists 功能 #42068 @MyonKeminta
  - 支持打印事务慢日志以及相关事务内部事件 #41863 @ekexium
  - 支持 ILIKE 操作符 #40943 @xzhangxian1008

- PD
  - 新增监控指标以展示 store limit 限制导致的调度生成失败 #6043 @nolouch
- TiFlash
  - 减少 TiFlash 在写路径上的内存使用量 #7144 @hongyunyan
  - 减少 TiFlash 在有较多表的情况下的重启时间 #7146 @hongyunyan
  - 支持下推 `ILIKE` 操作符 #6740 @xzhangxian1008
- Tools
  - TiCDC
    * 支持在 Kafka 为下游的场景中将单个大表的数据变更分布到多个 TiCDC 节点，从而解决大规模 TiDB 集群的数据集成场景下单表扩展性的问题 #8247 @overvenus
      你可以通过设置 TiCDC 配置项 `enable_table_across_nodes` 为 `true` 来启用这个功能，并通过设置 `region_threshold` 来指定当一张表的 region 个数超过阈值时，TiCDC 开始将对应的表上的数据变更分布到多个 TiCDC 节点。
    * 支持在 redo applier 中拆分事务以提升 apply 吞吐，降低灾难场景的 RTO #8318 @CharlesCheung96
    * 改进表的调度策略，可以将单个表更均匀地拆分到各个 TiCDC 节点上 #8247 @overvenus
    * 在 MQ sink 中添加了 Large Row 监控指标 #8286 @hi-rustin
    * 在一个 Region 包含多个表的数据的场景下，减少了 TiKV 与 TiCDC 节点间的网络流量 #6346 @overvenus
    * 将 Checkpoint TS 和 Resolved TS 的 P99 指标的面板移动到 Lag analyze 面板 #8524 @hi-rustin
    * 支持在 redo log 里 apply DDL 事件 #8361 @CharlesCheung96
    * 支持根据上游写入吞吐来拆分调度表到 TiCDC 节点 #7720 @overvenus
  - TiDB Lightning
    * TiDB Lightning 的物理导入模式 (Physical Import Mode) 支持导入数据和索引分离导入，提升导入速度和稳定性 #42132 @gozssky
      TiDB Lightning 增加 `add-index-by-sql` 参数。默认值为 `false`，表示仍然会用 TiDB Lightning 将行数据以及索引数据编码成 KV pairs 后再一同导入到 TiKV。如果设置为 `true`，表示在物理导入模式 (Physical Import Mode) 下，会在导入数据完成后，通过 `ADD INDEX` 的 SQL 语句帮你建索引，提升导入数据的速度和稳定性。
    * TiDB Lightning 增加 `tikv-importer.keyspace-name` 参数。默认值为空字符串，表示 TiDB Lightning 会去自动获取这次导入对应的 keyspace 名字。如果指定了值，那么使用指定的 keyspace 名字来导入。这个参数使得 TiDB Lightning 导入多租户的 TiDB 集群场景下可以进行灵活配置。#41915 @lichunzhu

2.2.4 错误修复

- TiDB
  - 修复 TiDB 从 v6.5.1 升级到新版本时遗漏部分更新的问题 #41502 @chrysan
  - 修复部分系统变量升级后默认值没有修改的问题 #41423 @crazycs520
  - 修复加索引相关 Coprocessor 请求类型显示为 unknown 的问题 #41400 @tangenta

  - 修复加索引时报 PessimisticLockNotFound 错误的问题 #41515 @tangenta
  - 修复加唯一索引时误报 found duplicate key 错误的问题 #41630 @tangenta
  - 修复加索引时 panic 的问题 #41880 @tangenta
  - 修复 TiFlash 执行中遇到生成列会报错的问题 #40663 @guo-shaoge
  - 修复当存在时间类型时，TiDB 可能无法正确获取统计信息的问题 #41938 @xuyifangreeneyes
  - 修复在开启 Prepared Plan Cache 的情况下，索引全表扫可能会报错的问题 #42150 @fzzf678
  - 修复 IFNULL(NOT NULL COLUMN, ...) 可能返回错误结果的问题 #41734 @LittleFall
  - 修复当分区表的所有数据落在单个 Region 时，TiDB 可能执行得到错误结果的问题 #41801 @Defined2014
  - 修复当同一个 SQL 中出现多个不同的分区表时，TiDB 可能执行得到错误结果的问题 #42135 @mjonss
  - 修复在为分区表添加新的索引之后，该分区表可能无法正确触发统计信息的自动收集的问题 #41638 @xuyifangreeneyes
  - 修复在连续两次收集统计信息后，TiDB 可能读取到错误的列统计信息的问题 #42073 @xuyifangreeneyes
  - 修复在开启 Prepared Plan Cache 时 Index Merge 可能得到错误结果的问题 #41828 @qw4990
  - 修复 IndexMerge 中 goroutine 泄露的问题 #41605 @guo-shaoge
  - 修复非 BIGINT 类型的无符号整数与 STRING/DECIMAL 比较时可能出现错误结果的问题 #41736 @LittleFall
  - 修复 ANALYZE 语句可能会因为当前 session 的前一个 ANALYZE 语句因为内存超限被 kill 导致当前 ANALYZE 语句也被 kill 的问题 #41825 @XuHuaiyu
  - 修复 batch coprocessor 搜集信息过程中存在的数据竞争问题 #41412 @you06
  - 修复 assertion 错误导致无法为分区表打印 MVCC 信息的问题 #40629 @ekexium
  - 修复 fair lock mode 对不存在的 key 加锁处理的问题 #41527 @ekexium
  - 修复 INSERT IGNORE 和 REPLACE 语句对不修改 value 的 key 没有加锁的问题 #42121 @zyguan

- PD

  - 修复了 Scatter Region 操作后 leader 分布不均衡的问题 #6017 @HunDunDM
  - 修复了启动过程中获取 PD 成员时的数据竞争问题 #6069 @rleungx
  - 修复了热点统计信息中的数据竞争问题 #6069 @lhy1024
  - 修复了切换 Placement Rule 时可能存在的 leader 分布不均衡的问题 #6195 @bufferflies

- TiFlash

  - 修复了 Decimal 除法在某些情况下最后一位未进位的问题 #7022 @LittleFall
  - 修复了 Decimal 转换在某些情况下进位错误的问题 #6994 @windtalker
  - 修复了开启 new collation 后 TopN/Sort 算子结果可能出错的问题 #6807 @xzhangxian1008
  - 修复了单个 TiFlash 节点聚合结果集过大（超过 1200 万）时可能会导致 TiFlash 报错的问题 #6993 @windtalker

- Tools

  - Backup & Restore (BR)

    * 修复了在 PITR 恢复过程中等待 split Region 重试的时间不足的问题 #42001 @joccau
    * 修复了在 PITR 恢复过程遇到 memory is limited 错误导致恢复失败的问题 #41983 @joccau
    * 修复了 PD 节点宕机可能导致 PITR 日志备份进度不推进的问题 #14184 @YuJuncen
    * 缓解了 Region leadership 迁移导致 PITR 日志备份进度延迟变高的问题 #13638 @YuJuncen

- TiCDC
  - 修复了重启 changefeed 可能导致数据丢失或者 checkpoint 无法推进的问题 #8242 @overvenus
  - 修复了 DDL sink 出现的数据竞争问题 #8238 @3AceShowHand
  - 修复了状态为 `stopped` 的 changefeed 可能会自动重启的问题 #8330 @sdojjy
  - 修复了当所有 Kafka server 不可访问时会导致 TiCDC server panic 的问题 #8523 @3AceShowHand
  - 修复了下游为 MySQL 且执行语句和 TiDB 行为不兼容时可能导致数据丢失的问题 #8453 @asddongmen
  - 修复了 rolling upgrade 可能导致 TiCDC OOM 或者 checkpoint 卡住的问题 #8329 @overvenus
  - 修复了 Kubernetes 上不能平滑升级 (graceful upgrade) TiCDC 集群的问题 #8484 @overvenus
- TiDB Data Migration (DM)
  - 修复了 DM worker 节点使用 GCP Cloud Storage 时，由于断点续传信息记录过于频繁，达到了 GCP Cloud Storage 的请求频次上限，导致 DM worker 无法把数据写入 GCP Cloud Storage 中，从而导致全量数据加载失败的问题 #8482 @maxshuang
  - 修复了在多个导入任务同时同步同一个下游的数据，并且都使用了下游元数据表来记录断点续传信息时，所有任务的断点续传信息被写入了同一张元数据表，并且使用了相同的任务 ID 的问题 #8500 @maxshuang
- TiDB Lightning
  - 修复了当使用 Physical Import Mode 导入数据时，如果目标表的复合主键中存在 `auto_random` 列，但源数据中没有指定该列的值，TiDB Lightning 不能为 `auto_random` 列自动生成数据的问题 #41454 @D3Hunter
  - 修复了当使用 TiDB Lightning 的 Logical Import Mode 导入数据时，由于目标集群用户没有 `CONFIG` 权限导致导入失败的问题 #41915 @lichunzhu

2.2.5 贡献者

感谢来自 TiDB 社区的贡献者们：

- AntiTopQuark
- blacktear23
- BornChanger
- Dousir9
- erwadba
- HappyUncle
- jiyfhust
- L-maple
- liumengya94
- woofyzhao
- xiaguan

2.3 TiDB 功能概览

本文列出了 TiDB 功能在不同版本中的支持变化情况，包括长期支持版本 (LTS) 和最新的 LTS 版本之后的开发里程碑版本 (DMR)。

注意：

PingCAP 不提供基于 DMR 版本的 bug 修复版本，如有 bug，会在后续版本中修复。如无特殊需求，建议使用最新 LTS 版本。

下表中出现的缩写字母含义如下：

- Y：已 GA 的功能，可以在生产环境中使用。注意即使某个功能在 DMR 版本中 GA，也建议在后续 LTS 版本中将该功能用于生产环境。
- N：不支持该功能。
- E：未 GA 的功能，即实验特性 (experimental)，请注意使用场景限制。实验特性会在未事先通知的情况下发生变化或删除。语法和实现可能会在 GA 前发生变化。如果遇到问题，请在 GitHub 上提交 issue 反馈。

2.3.1 数据类型，函数和操作符

数据类型，函数，操作符	7.0	6.6	6.5	6.1	5.4	5.3	5.2	5.1	5.0	4.0
数值类型	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
日期和时间类型	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
字符串类型	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
JSON 类型	Y	Y	Y	E	E	E	E	E	E	E
控制流程函数	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
字符串函数	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
数值函数与操作符	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
日期和时间函数	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
位函数和操作符	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Cast 函数和操作符	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
加密和压缩函数	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
信息函数	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
JSON 函数	Y	Y	Y	E	E	E	E	E	E	E
聚合函数	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
窗口函数	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
其他函数	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
操作符	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
字符集和排序规则 [1]	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
用户级别锁	Y	Y	Y	Y	N	N	N	N	N	N

2.3.2 索引和约束

[1] TiDB 误将 latin1 处理为 utf8 的子集。见 TiDB #18955。

索引和约束	7.0	6.6	6.5	6.1	5.4	5.3	5.2	5.1	5.0	4.0
表达式索引 [2]	Y	Y	Y	E	E	E	E	E	E	E
列式存储 (TiFlash)	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
使用 FastScan 加速 OLAP 场景下的查询	Y	E	E	N	N	N	N	N	N	N
RocksDB 引擎	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Titan 插件	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Titan Level Merge	E	E	E	E	E	E	E	E	E	E
使用 bucket 提高数据扫描并发度	E	E	E	E	N	N	N	N	N	N
不可见索引	Y	Y	Y	Y	Y	Y	Y	Y	Y	N
复合主键	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
唯一约束	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
整型主键上的聚簇索引	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
复合或非整型主键上的聚簇索引	Y	Y	Y	Y	Y	Y	Y	Y	Y	N
多值索引	E	E	N	N	N	N	N	N	N	N
外键约束	Y	Y	N	N	N	N	N	N	N	N
TiFlash 延迟物化	E	N	N	N	N	N	N	N	N	N

2.3.3 SQL 语句

SQL 语句 [3]	7.0	6.6	6.5	6.1	5.4	5.3	5.2	5.1	5.0	4.0
SELECT, INSERT, UPDATE, DELETE, REPLACE	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
INSERT ON DUPLICATE KEY UPDATE	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
LOAD DATA INFILE	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
SELECT INTO OUTFILE	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
INNER JOIN, LEFT|RIGHT [OUTER] JOIN	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
UNION, UNION ALL	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
EXCEPT 和 INTERSECT 运算符	Y	Y	Y	Y	Y	Y	Y	Y	Y	N
GROUP BY, ORDER BY	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
窗口函数	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
公共表表达式 (CTE)	Y	Y	Y	Y	Y	Y	Y	Y	N	N
START TRANSACTION, COMMIT, ROLLBACK	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
EXPLAIN	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
EXPLAIN ANALYZE	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
用户自定义变量	E	E	E	E	E	E	E	E	E	E
BATCH [ON COLUMN] LIMIT INTEGER DELETE	Y	Y	Y	Y	N	N	N	N	N	N
BATCH [ON COLUMN] LIMIT INTEGER INSERT/UPDATE/REPLACE	Y	Y	Y	N	N	N	N	N	N	N
ALTER TABLE ... COMPACT	Y	Y	Y	E	N	N	N	N	N	N
表级锁 (Table Lock)	E	E	E	E	E	E	E	E	E	E
物化列式存储的查询结果	E	E	E	N	N	N	N	N	N	N

[2] 从 v6.5.0 起，系统变量 tidb_allow_function_for_expression_index 所列出的函数已通过表达式索引的测试，可以在生产环境中创建并使用，未来版本会持续增加。对于没有列出的函数，则不建议在生产环境中使用相应的表达式索引。详情请参考表达式索引。

2.3.4 高级 SQL 功能

高级 SQL 功能	7.0	6.6	6.5	6.1	5.4	5.3	5.2	5.1	5.0	4.0
Prepare 语句执行计划缓存	Y	Y	Y	Y	Y	Y	E	E	E	E
非 Prepare 语句执行计划缓存	E	N	N	N	N	N	N	N	N	N
执行计划管理 (SPM)	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
根据历史执行计划创建绑定	Y	Y	E	N	N	N	N	N	N	N
下推计算结果缓存 (Coprocessor Cache)	Y	Y	Y	Y	Y	Y	Y	Y	Y	E
Stale Read	Y	Y	Y	Y	Y	Y	Y	Y	N	N
Follower Read	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
通过系统变量 `tidb_snapshot` 读取历史数据	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Optimizer hints	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
MPP 执行引擎	Y	Y	Y	Y	Y	Y	Y	Y	Y	N
MPP 执行引擎 - compression exchange	Y	Y	N	N	N	N	N	N	N	N
索引合并	Y	Y	Y	Y	Y	E	E	E	E	E
基于 SQL 的数据放置规则	Y	Y	Y	Y	E	E	N	N	N	N
Cascades Planner	E	E	E	E	E	E	E	E	E	E

2.3.5 数据定义语言 (DDL)

数据定义语言 (DDL)	7.0	6.6	6.5	6.1	5.4	5.3	5.2	5.1	5.0	4.0
`CREATE, DROP, ALTER, RENAME, TRUNCATE`	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
生成列	E	E	E	E	E	E	E	E	E	E
视图	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
序列	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
`AUTO_INCREMENT` 列	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
`AUTO_RANDOM` 列	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
TTL (Time to Live)	Y	E	E	N	N	N	N	N	N	N
DDL 算法断言	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
在单条语句中添加多列	Y	Y	Y	E	E	E	E	E	E	E
更改列类型	Y	Y	Y	Y	Y	Y	Y	Y	N	N
临时表	Y	Y	Y	Y	Y	Y	N	N	N	N
并行 DDL	Y	Y	Y	N	N	N	N	N	N	N
添加索引加速	Y	Y	Y	N	N	N	N	N	N	N
元数据锁	Y	Y	Y	N	N	N	N	N	N	N
`FLASHBACK CLUSTER TO TIMESTAMP`	Y	Y	Y	N	N	N	N	N	N	N

2.3.6 事务

事务	7.0	6.6	6.5	6.1	5.4	5.3	5.2	5.1	5.0	4.0
Async commit	Y	Y	Y	Y	Y	Y	Y	Y	Y	N

事务	7.0	6.6	6.5	6.1	5.4	5.3	5.2	5.1	5.0	4.0
1PC	Y	Y	Y	Y	Y	Y	Y	Y	Y	N
大事务 (10 GB)	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
悲观事务	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
乐观事务	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
可重复读隔离（快照隔离）	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
读已提交隔离	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

2.3.7 分区

分区	7.0	6.6	6.5	6.1	5.4	5.3	5.2	5.1	5.0	4.0
Range 分区	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Hash 分区	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Key 分区	Y	N	N	N	N	N	N	N	N	N
List 分区	Y	Y	Y	Y	E	E	E	E	E	N
List COLUMNS 分区	Y	Y	Y	Y	E	E	E	E	E	N
EXCHANGE PARTITION	Y	Y	Y	E	E	E	E	E	E	N
REORGANIZE PARTITION	Y	N	N	N	N	N	N	N	N	N
动态裁剪	Y	Y	Y	Y	E	E	E	E	N	N
Range COLUMNS 分区	Y	Y	Y	N	N	N	N	N	N	N
Range INTERVAL 分区	E	E	E	N	N	N	N	N	N	N

2.3.8 统计信息

统计信息	7.0	6.6	6.5	6.1	5.4	5.3	5.2
CM-Sketch	默认关闭	默认关闭	默认关闭	默认关闭	默认关闭	默认关闭	Y
直方图	Y	Y	Y	Y	Y	Y	Y
扩展统计信息（多列）	E	E	E	E	E	E	E
统计反馈	N	N	N	已废弃	已废弃	E	E
统计信息自动更新	Y	Y	Y	Y	Y	Y	Y
快速分析	E	E	E	E	E	E	E
动态裁剪	Y	Y	Y	Y	E	E	E
收集部分列的统计信息	E	E	E	E	E	N	N
限制统计信息的内存使用量	E	E	E	E	N	N	N
随机采样约 10000 行数据来快速构建统计信息	E	E	E	E	E	E	E
锁定统计信息	E	E	E	N	N	N	N

2.3.9 安全

安全	7.0	6.6	6.5	6.1	5.4	5.3	5.2	5.1	5.0	4.0
传输层加密 (TLS)	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
静态加密 (TDE)	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
基于角色的访问控制 (RBAC)	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
证书鉴权	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
caching_sha2_password 认证	Y	Y	Y	Y	Y	Y	Y	N	N	N
tidb_sm3_password 认证	Y	Y	Y	N	N	N	N	N	N	N
tidb_auth_token 认证	Y	Y	Y	N	N	N	N	N	N	N
密码管理	Y	Y	Y	N	N	N	N	N	N	N
与 MySQL 兼容的 GRANT 权限管理	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
动态权限	Y	Y	Y	Y	Y	Y	Y	Y	N	N
安全增强模式	Y	Y	Y	Y	Y	Y	Y	Y	N	N
日志脱敏	Y	Y	Y	Y	Y	Y	Y	Y	Y	N

2.3.10 数据导入和导出

数据导入和导出	7.0	6.6	6.5	6.1	5.4	5.3	5.2
快速导入 (TiDB Lightning)	Y	Y	Y	Y	Y	Y	Y
mydumper 逻辑导出	已废弃	已废弃	已废弃	已废弃	已废弃	已废弃	已废弃
Dumpling 逻辑导出	Y	Y	Y	Y	Y	Y	Y
事务 LOAD DATA	E[4]	Y	Y	Y	Y	Y	Y
数据迁移工具	Y	Y	Y	Y	Y	Y	Y
TiDB Binlog	Y	Y	Y	Y	Y	Y	Y
Change data capture (CDC)	Y	Y	Y	Y	Y	Y	Y
TiCDC 支持保存数据到存储服务 (S3/NFS/Azure Blob Storage/GCP)	Y	E	E	N	N	N	N
TiCDC 支持在两个 TiDB 集群之间进行双向复制	Y	Y	Y	N	N	N	N
TiCDC OpenAPI v2	Y	N	N	N	N	N	N

2.3.11 管理，可视化和工具

管理，可视化和工具	7.0	6.6	6.5	6.1	5.4	5.3	5.2	5.1	5.0	4.0
TiDB Dashboard 图形化展示	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
TiDB Dashboard 持续性能分析功能	Y	Y	Y	Y	E	E	N	N	N	N
TiDB Dashboard Top SQL 功能	Y	Y	Y	Y	E	N	N	N	N	N
TiDB Dashboard SQL 诊断功能	Y	Y	Y	E	E	E	E	E	E	E
TiDB Dashboard 集群诊断功能	Y	Y	Y	E	E	E	E	E	E	E
Grafana 中的 TiKV-FastTune 面板	E	E	E	E	E	E	E	E	E	E
Information schema	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Metrics schema	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

[4]对于TiDB v7.0.0，新增参数 FIELDS DEFINED NULL BY 以及新增支持从 S3 和 GCS 导入数据，均为实验特性。
[5]对于 TiDB v4.0，事务 LOAD DATA 不保证原子性。

管理，可视化和工具	7.0	6.6	6.5	6.1	5.4	5.3	5.2	5.1	5.0	4.0
Statements summary tables	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Statements summary tables - 持久化 statements summary	E	E	N	N	N	N	N	N	N	N
慢查询日志	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
TiUP 部署	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Kubernetes operator	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
内置物理备份	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Global Kill	Y	Y	Y	Y	E	E	E	E	E	E
Lock View	Y	Y	Y	Y	Y	Y	Y	E	E	E
SHOW CONFIG	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
SET CONFIG	Y	Y	Y	Y	E	E	E	E	E	E
DM WebUI	E	E	E	E	N	N	N	N	N	N
前台限流	Y	Y	Y	E	N	N	N	N	N	N
后台限流	E	E	E	N	N	N	N	N	N	N
基于 EBS 的备份和恢复	Y	Y	Y	N	N	N	N	N	N	N
PITR	Y	Y	Y	N	N	N	N	N	N	N
全局内存控制	Y	Y	Y	N	N	N	N	N	N	N
RawKV 跨集群复制	E	E	E	N	N	N	N	N	N	N
Green GC	E	E	E	E	E	E	E	E	E	N
资源管控 (Resource Control)	E	E	N	N	N	N	N	N	N	N
TiFlash 存算分离架构与 S3 支持	E	N	N	N	N	N	N	N	N	N

2.4 与 MySQL 兼容性对比

TiDB 高度兼容 MySQL 5.7 协议、MySQL 5.7 常用的功能及语法。MySQL 5.7 生态中的系统工具（PHPMyAdmin、Navicat、MySQL Workbench、 mysqldump、 Mydumper/Myloader ）、客户端等均适用于 TiDB。

但 TiDB 尚未支持一些 MySQL 功能，可能的原因如下：

- 有更好的解决方案，例如 JSON 取代 XML 函数。
- 目前对这些功能的需求度不高，例如存储流程和函数。
- 一些功能在分布式系统上的实现难度较大。

除此以外，TiDB 不支持 MySQL 复制协议，但提供了专用工具用于与 MySQL 复制数据：

- 从 MySQL 复制：TiDB Data Migration (DM) 是将 MySQL/MariaDB 数据迁移到 TiDB 的工具，可用于增量数据的复制。
- 向 MySQL 复制：TiCDC 是一款通过拉取 TiKV 变更日志实现的 TiDB 增量数据同步工具，可通过MySQL sink 将 TiDB 增量数据复制到 MySQL。

> 注意：
>
> 本页内容仅涉及 MySQL 与 TiDB 的总体差异。关于安全特性、悲观事务模式相关的兼容信息请查看各自具体页面。

2.4.1 不支持的功能特性

- 存储过程与函数
- 触发器
- 事件
- 自定义函数
- 全文语法与索引 #1793
- 空间类型的函数（即 GIS/GEOMETRY）、数据类型和索引 #6347
- 非 ascii、latin1、binary、utf8、utf8mb4、gbk 的字符集
- SYS schema
- MySQL 追踪优化器
- XML 函数
- X-Protocol #1109
- 列级权限 #9766
- XA 语法（TiDB 内部使用两阶段提交，但并没有通过 SQL 接口公开）
- CREATE TABLE tblName AS SELECT stmt 语法 #4754
- CHECK TABLE 语法 #4673
- CHECKSUM TABLE 语法 #1895
- REPAIR TABLE 语法
- OPTIMIZE TABLE 语法
- HANDLER 语句
- CREATE TABLESPACE 语句
- “Session Tracker: 将 GTID 上下文信息添加到 OK 包中”

2.4.2 与 MySQL 有差异的特性详细说明

2.4.2.1 自增 ID

- TiDB 的自增列既能保证唯一，也能保证在单个 TiDB server 中自增，使用AUTO_INCREMENT MySQL 兼容模式能保证多个 TiDB server 中自增 ID，但不保证自动分配的值的连续性。不建议将缺省值和自定义值混用，若混用可能会收到 Duplicated Error 的错误信息。
- TiDB 可通过 tidb_allow_remove_auto_inc 系统变量开启或者关闭允许移除列的 AUTO_INCREMENT 属性。删除列属性的语法是：ALTER TABLE MODIFY 或 ALTER TABLE CHANGE。
- TiDB 不支持添加列的 AUTO_INCREMENT 属性，移除该属性后不可恢复。
- 对于 v6.6.0 及更早的 TiDB 版本，TiDB 的行为与 MySQL InnoDB 保持一致，要求自增列必须为主键或者索引前缀。从 v7.0.0 开始，TiDB 移除自增列必须是索引或索引前缀的限制，允许用户更灵活地定义表的主键。关于此更改的详细信息，请参阅 #40580

自增 ID 详情可参阅AUTO_INCREMENT。

> 注意：

若创建表时没有指定主键时，TiDB 会使用 _tidb_rowid 来标识行，该数值的分配会和自增列（如果存在的话）共用一个分配器。如果指定了自增列为主键，则 TiDB 会用该列来标识行。因此会有以下的示例情况：

```
mysql> CREATE TABLE t(id INT UNIQUE KEY AUTO_INCREMENT);
Query OK, 0 rows affected (0.05 sec)

mysql> INSERT INTO t VALUES();
Query OK, 1 rows affected (0.00 sec)

mysql> INSERT INTO t VALUES();
Query OK, 1 rows affected (0.00 sec)

mysql> INSERT INTO t VALUES();
Query OK, 1 rows affected (0.00 sec)

mysql> SELECT _tidb_rowid, id FROM t;
+-------------+------+
| _tidb_rowid | id   |
+-------------+------+
|           2 |    1 |
|           4 |    3 |
|           6 |    5 |
+-------------+------+
3 rows in set (0.01 sec)
```

可以看到，由于共用分配器，id 每次自增步长是 2。在MySQL 兼容模式中改掉了该行为，没有共用分配器，因此不会跳号。

注意：

使用 AUTO_INCREMENT 可能会给生产环境带热点问题，因此推荐使用AUTO_RANDOM 代替。详情请参考TiDB 热点问题处理。

2.4.2.2 Performance schema

TiDB 主要使用 Prometheus 和 Grafana 来存储及查询相关的性能监控指标，所以 Performance schema 部分表是空表。

2.4.2.3 查询计划

TiDB 中，执行计划（EXPLAIN 和 EXPLAIN FOR）在输出格式、内容、权限设置方面与 MySQL 有较大差别。

MySQL 系统变量 optimizer_switch 在 TiDB 中是只读的，对查询计划没有影响。你还可以在optimizer hints 中使用与 MySQL 类似的语法，但可用的 hint 和实现原理可能会有所不同。

详情参见理解 TiDB 执行计划。

2.4.2.4 内建函数

支持常用的 MySQL 内建函数，有部分函数并未支持。可通过执行 SHOW BUILTINS 语句查看可用的内建函数。参考 SQL 语法文档。

2.4.2.5 DDL 的限制

TiDB 中，所有支持的 DDL 变更操作都是在线执行的。与 MySQL 相比，TiDB 中的 DDL 存在以下限制：

- 使用 ALTER TABLE 语句修改一个表的多个模式对象（如列、索引）时，不允许在多个更改中指定同一个模式对象。例如，ALTER TABLE t1 MODIFY COLUMN c1 INT, DROP COLUMN c1 在两个更改中都指定了 c1 列，执行该语句会输出 Unsupported operate same column/index 的错误。
- 不支持使用单个 ALTER TABLE 语句同时修改多个 TiDB 特有的模式对象，包括 TIFLASH REPLICA，SHARD_ROW_ID_BITS，AUTO_ID_CACHE 等。
- ALTER TABLE 不支持少部分类型的变更。比如，TiDB 不支持从 DECIMAL 到 DATE 的变更。当遇到不支持的类型变更时，TiDB 将会报 Unsupported modify column: type %d not match origin %d 的错误。更多细节，请参考ALTER TABLE。
- TiDB 中，ALGORITHM={INSTANT,INPLACE,COPY} 语法只作为一种指定，并不更改 ALTER 算法，详情参阅ALTER TABLE。
- 不支持添加或删除 CLUSTERED 类型的主键。要了解关于 CLUSTERED 主键的详细信息，请参考聚簇索引。
- 不支持指定不同类型的索引 (HASH|BTREE|RTREE|FULLTEXT)。若指定了不同类型的索引，TiDB 会解析并忽略这些索引。
- 分区表支持 HASH、RANGE、LIST 和 KEY 分区类型。KEY 分区类型暂不支持分区字段列表为空的语句。对于不支持的分区类型，TiDB 会返回 Warning: Unsupported partition type %s, treat as normal table，其中 %s 为不支持的具体分区类型。
- 分区表还支持 ADD、DROP、TRUNCATE、REORGANIZE 操作，其他分区操作会被忽略。TiDB 不支持以下分区表语法：
  - SUBPARTITION
  - {CHECK|OPTIMIZE|REPAIR|IMPORT|DISCARD|REBUILD|COALESCE} PARTITION

更多详情，请参考分区表文档。

2.4.2.6 ANALYZE TABLE

TiDB 中的信息统计与 MySQL 中的有所不同：TiDB 中的信息统计会完全重构表的统计数据，语句执行过程较长，但在 MySQL/InnoDB 中，它是一个轻量级语句，执行过程较短。

更多信息统计的差异请参阅ANALYZE TABLE。

2.4.2.7 SELECT 的限制

- 不支持 SELECT ... INTO @变量 语法。
- 不支持 SELECT ... GROUP BY ... WITH ROLLUP 语法。
- TiDB 中的 SELECT .. GROUP BY expr 的返回结果与 MySQL 5.7 并不一致。MySQL 5.7 的结果等价于 GROUP BY expr ORDER BY expr。

详情参见SELECT。

2.4.2.8 UPDATE 语句

详情参见UPDATE。

2.4.2.9 视图

TiDB 中的视图不可更新，不支持 UPDATE、INSERT、DELETE 等写入操作。

2.4.2.10 临时表

详见TiDB 本地临时表与 MySQL 临时表的兼容性。

2.4.2.11 字符集和排序规则

- 关于 TiDB 对字符集和排序规则的支持情况，详见字符集和排序规则。
- 关于 GBK 字符集与 MySQL 的兼容情况，详见GBK 兼容情况。
- TiDB 继承表中使用的字符集作为国家字符集。

2.4.2.12 存储引擎

- 仅在语法上兼容创建表时指定存储引擎，实际上 TiDB 会将元信息统一描述为 InnoDB 存储引擎。TiDB 支持类似 MySQL 的存储引擎抽象，但需要在系统启动时通过--store 配置项来指定存储引擎。

2.4.2.13 SQL 模式

TiDB 支持大部分SQL 模式。不支持的 SQL 模式如下：

- 不支持兼容模式，例如：Oracle 和 PostgreSQL（TiDB 解析但会忽略这两个兼容模式），MySQL 5.7 已弃用兼容模式，MySQL 8.0 已移除兼容模式。
- TiDB 的 ONLY_FULL_GROUP_BY 模式与 MySQL 5.7 相比有细微的语义差别。
- NO_DIR_IN_CREATE 和 NO_ENGINE_SUBSTITUTION 仅用于解决与 MySQL 的兼容性问题，并不适用于 TiDB。

2.4.2.14 默认设置

- 字符集：
  - TiDB 默认：utf8mb4。
  - MySQL 5.7 默认：latin1。
  - MySQL 8.0 默认：utf8mb4。
- 排序规则：
  - TiDB 中 utf8mb4 字符集默认：utf8mb4_bin。
  - MySQL 5.7 中 utf8mb4 字符集默认：utf8mb4_general_ci。
  - MySQL 8.0 中 utf8mb4 字符集默认：utf8mb4_0900_ai_ci。
- SQL mode：
  - TiDB 默认：ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO
    ↪ ,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION。
  - MySQL 5.7 默认与 TiDB 相同。
  - MySQL 8.0 默认 ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,
    ↪ ERROR_FOR_DIVISION_BY_ZERO,NO_ENGINE_SUBSTITUTION。
- lower_case_table_names：
  - TiDB 默认：2，且仅支持设置该值为 2。
  - MySQL 默认如下：
    * Linux 系统中该值为 0，表示表名和数据库名按照在 CREATE TABLE 或 CREATE DATABASE 语句中指定的字母大小写存储在磁盘上，且名称比较时区分大小写。
    * Windows 系统中该值为 1，表示表名按照小写字母存储在磁盘上，名称比较时不区分大小写。MySQL 在存储和查询时将所有表名转换为小写。该行为也适用于数据库名称和表的别名。
    * macOS 系统中该值为 2，表示表名和数据库名按照在 CREATE TABLE 或 CREATE DATABASE 语句中指定的字母大小写存储在磁盘上，但 MySQL 在查询时将它们转换为小写。名称比较时不区分大小写。
- explicit_defaults_for_timestamp：
  - TiDB 默认：ON，且仅支持设置该值为 ON。
  - MySQL 5.7 默认：OFF。
  - MySQL 8.0 默认：ON。

2.4.2.15 日期时间处理的区别

2.4.2.15.1 时区

- TiDB 采用系统当前安装的所有时区规则进行计算（一般为 tzdata 包），不需要导入时区表数据就能使用所有时区名称，无法通过导入时区表数据的形式修改计算规则。
- MySQL 默认使用本地时区，依赖于系统内置的当前的时区规则（例如什么时候开始夏令时等）进行计算；且在未导入时区表数据的情况下不能通过时区名称来指定时区。

2.4.2.16 类型系统

- 不支持 FLOAT4/FLOAT8。
- 不支持 SQL_TSI_*（包括 SQL_TSI_MONTH、SQL_TSI_WEEK、SQL_TSI_DAY、SQL_TSI_HOUR、SQL_TSI_MINUTE 和 SQL_TSI_SECOND，但不包括 SQL_TSI_YEAR）。

2.4.2.17 MySQL 弃用功能导致的不兼容问题

TiDB 不支持 MySQL 中标记为弃用的功能，包括：

- 指定浮点类型的精度。MySQL 8.0 弃用了此功能，建议改用 DECIMAL 类型。
- ZEROFILL 属性。MySQL 8.0 弃用了此功能，建议在业务应用中填充数字值。

2.4.2.18 CREATE RESOURCE GROUP，DROP RESOURCE GROUP 和 ALTER RESOURCE GROUP

TiDB 资源组创建与修改语句的语法与 MySQL 官方不同，详情参见：

- CREATE RESOURCE GROUP
- DROP RESOURCE GROUP
- ALTER RESOURCE GROUP

2.5 使用限制

本文会将详细描述 TiDB 中常见的使用限制，包括：标识符长度，最大支持的数据库、表、索引、分区表、序列等的个数。

2.5.1 标识符长度限制

标识符类型	最大长度（字符）
Database	64
Table	64
Column	64
Index	64
View	64
Sequence	64

2.5.2 Databases、Tables、Views、Connections 总个数限制

标识符类型	最大个数
Databases	unlimited
Tables	unlimited
Views	unlimited

标识符类型	最大个数
Connections	unlimited

2.5.3 单个 Database 的限制

类型	最大限制
Tables	unlimited

2.5.4 单个 Table 的限制

类型	最大限制（默认值）
Columns	默认为 1017，最大可调至 4096
Indexs	默认为 64，最大可调至 512
Rows	无限制
Size	无限制
Partitions	8192

- Columns 的最大限制可通过`table-column-count-limit` 修改。
- Indexs 的最大限制可通过`index-limit` 修改。

2.5.5 单行的限制

类型	最大限制（默认值）
Size	默认为 6 MiB，可通过`txn-entry-size-limit` 配置项调至 120 MiB

2.5.6 数据类型限制

类型	最大限制
CHAR	256 字符
BINARY	256 字节
VARBINARY	65535 字节
VARCHAR	16383 字符
TEXT	默认为 6291456 字节（即 6 MiB），可调至 125829120 字节（即 120 MiB）
BLOB	默认为 6291456 字节（即 6 MiB），可调至 125829120 字节（即 120 MiB）

2.5.7 SQL Statements 的限制

类型	最大限制
单个事务最大语句数	在使用乐观事务并开启事务重试的情况下，默认限制 5000，可通过`stmt-count-limit` 调整

2.5.8 TiKV 版本的限制

在集群中，如果 TiDB 组件的版本为 v6.2.0 及以上，则 TiKV 组件版本不得低于 v6.2.0。

2.6 TiDB 社区荣誉列表

每一位贡献者都是推动 TiDB 健壮发展的重要成员，我们感谢所有为 TiDB 提交代码、撰写或翻译文档的贡献者。

2.6.1 TiDB 开发者

TiDB 开发者为 TiDB 的新功能开发、性能优化、稳定性保障做出了贡献。以下链接包含了 TiDB 相关 repo 的贡献者名单：

- pingcap/tidb
- tikv/tikv
- tikv/pd
- pingcap/tiflash
- pingcap/tidb-operator
- pingcap/tiup
- pingcap/tidb-binlog
- pingcap/tidb-dashboard
- pingcap/tiflow
- pingcap/tidb-tools
- pingcap/tispark
- tikv/client-java
- tidb-incubator/TiBigData
- ti-community-infra

2.6.2 TiDB 文档写作者和译员

TiDB 文档写作者和译员为 TiDB 及相关项目撰写文档、提供翻译。以下链接包含了 TiDB 文档相关 repo 的贡献者名单：

- pingcap/docs-cn
- pingcap/docs
- pingcap/docs-tidb-operator
- tikv/website

2.7 路线图

3 快速上手

3.1 TiDB 数据库快速上手指南

本指南介绍如何快速上手体验 TiDB 数据库。对于非生产环境，你可以选择以下任意一种方式部署 TiDB 数据库：

- 部署本地测试集群（支持 macOS 和 Linux）
- 在单机上模拟部署生产环境集群（支持 Linux）

> 注意：
>
> 本指南中的 TiDB 部署方式仅适用于快速上手体验，不适用于生产环境。
>
> - 如需在生产环境部署 TiDB，请参考在生产环境中部署 TiDB 指南。
> - 如需在 Kubernetes 上部署 TiDB，请参考快速上手 TiDB Operator。
> - 如需在云上管理 TiDB，请参考 TiDB Cloud 快速上手指南。

要快速了解 TiUP 的基本功能、使用 TiUP 快速搭建 TiDB 集群的方法与连接 TiDB 集群并执行 SQL 的方法，建议先观看下面的培训视频（时长 15 分钟）。注意本视频只作为学习参考，如需了解TiUP 的具体使用方法和TiDB 快速上手具体操作步骤，请以文档内容为准。

3.1.1 部署本地测试集群

- 适用场景：利用本地 macOS 或者单机 Linux 环境快速部署 TiDB 测试集群，体验 TiDB 集群的基本架构，以及 TiDB、TiKV、PD、监控等基础组件的运行。

TiDB 是一个分布式系统。最基础的 TiDB 测试集群通常由 2 个 TiDB 实例、3 个 TiKV 实例、3 个 PD 实例和可选的 TiFlash 实例构成。通过 TiUP Playground，可以快速搭建出上述的一套基础测试集群，步骤如下：

1. 下载并安装 TiUP。

```
curl --proto '=https' --tlsv1.2 -sSf https://tiup-mirrors.pingcap.com/install.sh | sh
```

安装完成后会提示如下信息：

```
Successfully set mirror to https://tiup-mirrors.pingcap.com
Detected shell: zsh
Shell profile:  /Users/user/.zshrc
/Users/user/.zshrc has been modified to add tiup to PATH
open a new terminal or source /Users/user/.zshrc to use it
Installed path: /Users/user/.tiup/bin/tiup
```

```
================================================
Have a try:     tiup playground
================================================
```

2. 声明全局环境变量。

> **注意：**
>
> TiUP 安装完成后会提示 Shell profile 文件的绝对路径。在执行以下 `source` 命令前，需要将 `${your_shell_profile}` 修改为 Shell profile 文件的实际位置。

```
source ${your_shell_profile}
```

3. 在当前 session 执行以下命令启动集群。

    - 直接执行 `tiup playground` 命令会运行最新版本的 TiDB 集群，其中 TiDB、TiKV、PD 和 TiFlash 实例各 1 个：

    ```
    tiup playground
    ```

    - 也可以指定 TiDB 版本以及各组件实例个数，命令类似于：

    ```
    tiup playground v7.0.0 --db 2 --pd 3 --kv 3
    ```

    上述命令会在本地下载并启动某个版本的集群（例如 v7.0.0）。最新版本可以通过执行 `tiup list tidb` 来查看。运行结果将显示集群的访问方式：

    ```
    CLUSTER START SUCCESSFULLY, Enjoy it ^-^
    To connect TiDB: mysql --comments --host 127.0.0.1 --port 4001 -u root -p (no password)
    To connect TiDB: mysql --comments --host 127.0.0.1 --port 4000 -u root -p (no password)
    To view the dashboard: http://127.0.0.1:2379/dashboard
    PD client endpoints: [127.0.0.1:2379 127.0.0.1:2382 127.0.0.1:2384]
    To view the Prometheus: http://127.0.0.1:9090
    To view the Grafana: http://127.0.0.1:3000
    ```

> **注意：**
>
> - 支持 v5.2.0 及以上版本的 TiDB 在 Apple M1 芯片的机器上运行 `tiup playground`。
> - 以这种方式执行的 playground，在结束部署测试后 TiUP 会清理掉原集群数据，重新执行该命令后会得到一个全新的集群。
> - 若希望持久化数据，可以执行 TiUP 的 `--tag` 参数：`tiup --tag <your-tag> playground ...`，详情参考TiUP 参考手册。

4. 新开启一个 session 以访问 TiDB 数据库。

    - 使用 TiUP `client` 连接 TiDB：

```
tiup client
```

- 也可使用 MySQL 客户端连接 TiDB：

```
mysql --host 127.0.0.1 --port 4000 -u root
```

5. 通过 http://127.0.0.1:9090 访问 TiDB 的 Prometheus 管理界面。
6. 通过 http://127.0.0.1:2379/dashboard 访问TiDB Dashboard 页面，默认用户名为 root，密码为空。
7. 通过 http://127.0.0.1:3000 访问 TiDB 的 Grafana 界面，默认用户名和密码都为 admin。
8. （可选）将数据加载到 TiFlash 进行分析。
9. 测试完成之后，可以通过执行以下步骤来清理集群：
    1. 按下 Control+C 键停掉上述启用的 TiDB 服务。
    2. 等待服务退出操作完成后，执行以下命令：

```
tiup clean --all
```

> 注意：
>
> TiUP Playground 默认监听 127.0.0.1，服务仅本地可访问；若需要使服务可被外部访问，可使用 --host 参数指定监听网卡绑定外部可访问的 IP。

TiDB 是一个分布式系统。最基础的 TiDB 测试集群通常由 2 个 TiDB 实例、3 个 TiKV 实例、3 个 PD 实例和可选的 TiFlash 实例构成。通过 TiUP Playground，可以快速搭建出上述的一套基础测试集群，步骤如下：

1. 下载并安装 TiUP。

```
curl --proto '=https' --tlsv1.2 -sSf https://tiup-mirrors.pingcap.com/install.sh | sh
```

安装完成后会提示如下信息：

```
Successfully set mirror to https://tiup-mirrors.pingcap.com
Detected shell: bash
Shell profile:  /home/user/.bashrc
/home/user/.bashrc has been modified to add tiup to PATH
open a new terminal or source /home/user/.bashrc to use it
Installed path: /home/user/.tiup/bin/tiup
===============================================
Have a try:     tiup playground
===============================================
```

2. 声明全局环境变量。

> **注意：**
>
> TiUP 安装完成后会提示 Shell profile 文件的绝对路径。在执行以下 `source` 命令前，需要将 `${your_shell_profile}` 修改为 Shell profile 文件的实际位置。

```
source ${your_shell_profile}
```

3. 在当前 session 执行以下命令启动集群。

    - 直接运行 `tiup playground` 命令会运行最新版本的 TiDB 集群，其中 TiDB、 TiKV、 PD 和 TiFlash 实例各 1 个：

    ```
    tiup playground
    ```

    - 也可以指定 TiDB 版本以及各组件实例个数，命令类似于：

    ```
    tiup playground v7.0.0 --db 2 --pd 3 --kv 3
    ```

    上述命令会在本地下载并启动某个版本的集群（例如 v7.0.0）。最新版本可以通过执行 `tiup list tidb` 来查看。运行结果将显示集群的访问方式：

    ```
    CLUSTER START SUCCESSFULLY, Enjoy it ^-^
    To connect TiDB: mysql --host 127.0.0.1 --port 4000 -u root -p (no password) --comments
    To view the dashboard: http://127.0.0.1:2379/dashboard
    PD client endpoints: [127.0.0.1:2379]
    To view the Prometheus: http://127.0.0.1:9090
    To view the Grafana: http://127.0.0.1:3000
    ```

> **注意：**
>
> - 以这种方式执行的 playground，在结束部署测试后 TiUP 会清理掉原集群数据，重新执行该命令后会得到一个全新的集群。
> - 若希望持久化数据，可以执行 TiUP 的 `--tag` 参数：`tiup --tag <your-tag> playground ...`，详情参考TiUP 参考手册。

4. 新开启一个 session 以访问 TiDB 数据库。

    - 使用 TiUP `client` 连接 TiDB：

    ```
    tiup client
    ```

    - 也可使用 MySQL 客户端连接 TiDB：

    ```
    mysql --host 127.0.0.1 --port 4000 -u root
    ```

5. 通过 http://127.0.0.1:9090 访问 TiDB 的 Prometheus 管理界面。
6. 通过 http://127.0.0.1:2379/dashboard 访问TiDB Dashboard 页面，默认用户名为 root，密码为空。
7. 通过 http://127.0.0.1:3000 访问 TiDB 的 Grafana 界面，默认用户名和密码都为 admin。
8. （可选）将数据加载到 TiFlash 进行分析。
9. 测试完成之后，可以通过执行以下步骤来清理集群：
    1. 按下 Control+C 键停掉上述启用的 TiDB 服务。
    2. 等待服务退出操作完成后，执行以下命令：

```
tiup clean --all
```

> 注意：
>
> TiUP Playground 默认监听 127.0.0.1，服务仅本地可访问。若需要使服务可被外部访问，可使用 --host 参数指定监听网卡绑定外部可访问的 IP。

3.1.2 在单机上模拟部署生产环境集群

- 适用场景：希望用单台 Linux 服务器，体验 TiDB 最小的完整拓扑的集群，并模拟生产环境下的部署步骤。

本节介绍如何参照 TiUP 最小拓扑的一个 YAML 文件部署 TiDB 集群。

3.1.2.1 准备环境

准备一台部署主机，确保其软件满足需求：

- 推荐安装 CentOS 7.3 及以上版本
- 运行环境可以支持互联网访问，用于下载 TiDB 及相关软件安装包

最小规模的 TiDB 集群拓扑：

> 注意：
>
> 下表中拓扑实例的 IP 为示例 IP。在实际部署时，请替换为实际的 IP。

实例	个数	IP	配置
TiKV	3	10.0.1.1 10.0.1.1 10.0.1.1	避免端口和目录冲突

实例	个数	IP	配置
TiDB	1	10.0.1.1	默认端口全局目录配置
PD	1	10.0.1.1	默认端口全局目录配置
TiFlash	1	10.0.1.1	默认端口全局目录配置
Monitor	1	10.0.1.1	默认端口全局目录配置

部署主机软件和环境要求：

- 部署需要使用部署主机的 root 用户及密码
- 部署主机关闭防火墙或者开放 TiDB 集群的节点间所需端口
- 目前 TiUP Cluster 支持在 x86_64（AMD64）和 ARM 架构上部署 TiDB 集群
  - 在 AMD64 架构下，建议使用 CentOS 7.3 及以上版本 Linux 操作系统
  - 在 ARM 架构下，建议使用 CentOS 7.6 1810 版本 Linux 操作系统

3.1.2.2 实施部署

> 注意：
> 你可以使用 Linux 系统的任一普通用户或 root 用户登录主机，以下步骤以 root 用户为例。

1. 下载并安装 TiUP：

```
curl --proto '=https' --tlsv1.2 -sSf https://tiup-mirrors.pingcap.com/install.sh | sh
```

2. 声明全局环境变量：

> 注意：
> TiUP 安装完成后会提示对应 Shell profile 文件的绝对路径。在执行以下 `source` 命令前，需要将 `${your_shell_profile}` 修改为 Shell profile 文件的实际位置。

```
source ${your_shell_profile}
```

3. 安装 TiUP 的 cluster 组件：

```
tiup cluster
```

4. 如果机器已经安装 TiUP cluster，需要更新软件版本：

```
tiup update --self && tiup update cluster
```

5. 由于模拟多机部署，需要通过 root 用户调大 sshd 服务的连接数限制：

    1. 修改 `/etc/ssh/sshd_config` 将 `MaxSessions` 调至 20。
    2. 重启 sshd 服务：

    ```
    service sshd restart
    ```

6. 创建并启动集群

    按下面的配置模板，编辑配置文件，命名为 `topo.yaml`，其中：

    - `user: "tidb"`：表示通过 `tidb` 系统用户（部署会自动创建）来做集群的内部管理，默认使用 22 端口通过 ssh 登录目标机器
    - `replication.enable-placement-rules`：设置这个 PD 参数来确保 TiFlash 正常运行
    - `host`：设置为本部署主机的 IP

    配置模板如下：

```
# # Global variables are applied to all deployments and used as the default value of
# # the deployments if a specific deployment value is missing.
global:
 user: "tidb"
 ssh_port: 22
 deploy_dir: "/tidb-deploy"
 data_dir: "/tidb-data"

# # Monitored variables are applied to all the machines.
monitored:
 node_exporter_port: 9100
 blackbox_exporter_port: 9115

server_configs:
 tidb:
   log.slow-threshold: 300
 tikv:
   readpool.storage.use-unified-pool: false
   readpool.coprocessor.use-unified-pool: true
 pd:
   replication.enable-placement-rules: true
   replication.location-labels: ["host"]
 tiflash:
   logger.level: "info"

pd_servers:
 - host: 10.0.1.1

tidb_servers:
 - host: 10.0.1.1
```

```
tikv_servers:
 - host: 10.0.1.1
   port: 20160
   status_port: 20180
   config:
     server.labels: { host: "logic-host-1" }

 - host: 10.0.1.1
   port: 20161
   status_port: 20181
   config:
     server.labels: { host: "logic-host-2" }

 - host: 10.0.1.1
   port: 20162
   status_port: 20182
   config:
     server.labels: { host: "logic-host-3" }

tiflash_servers:
 - host: 10.0.1.1

monitoring_servers:
 - host: 10.0.1.1

grafana_servers:
 - host: 10.0.1.1
```

7. 执行集群部署命令：

```
tiup cluster deploy <cluster-name> <version> ./topo.yaml --user root -p
```

- 参数 <cluster-name> 表示设置集群名称
- 参数 <version> 表示设置集群版本，例如 v6.5.0。可以通过 tiup list tidb 命令来查看当前支持部署的 TiDB 版本
- 参数 -p 表示在连接目标机器时使用密码登录

> 注意：
> 如果主机通过密钥进行 SSH 认证，请使用 -i 参数指定密钥文件路径，-i 与 -p 不可同时使用。

按照引导，输入”y”及 root 密码，来完成部署：

```
Do you want to continue? [y/N]:  y
Input SSH password:
```

8. 启动集群：

```
tiup cluster start <cluster-name>
```

9. 访问集群：

   - 安装 MySQL 客户端。如果已安装 MySQL 客户端则可跳过这一步骤。

     ```
     yum -y install mysql
     ```

   - 访问 TiDB 数据库，密码为空：

     ```
     mysql -h 10.0.1.1 -P 4000 -u root
     ```

   - 访问 TiDB 的 Grafana 监控：

     通过 http://%7Bgrafana-ip%7D:3000 访问集群 Grafana 监控页面，默认用户名和密码均为 admin。

   - 访问 TiDB 的 Dashboard：

     通过 http://%7Bpd-ip%7D:2379/dashboard 访问集群TiDB Dashboard 监控页面，默认用户名为 root，密码为空。

   - 执行以下命令确认当前已经部署的集群列表：

     ```
     tiup cluster list
     ```

   - 执行以下命令查看集群的拓扑结构和状态：

     ```
     tiup cluster display <cluster-name>
     ```

3.1.3 探索更多

- 如果你刚刚部署好一套 TiDB 本地测试集群：
  - 学习TiDB SQL 操作
  - 迁移数据到 TiDB
- 如果你准备好在生产环境部署 TiDB 了：
  - 在线部署：使用 TiUP 部署 TiDB 集群
  - 使用 TiDB Operator 在云上部署 TiDB
- 如果你想使用 TiFlash 作为数据分析的解决方案，可参阅以下文档：
  - 使用 TiFlash
  - TiFlash 简介

3.2 HTAP 快速上手指南

本指南介绍如何快速上手体验 TiDB 的一站式混合型在线事务与在线分析处理 (Hybrid Transactional and Analytical Processing, HTAP) 功能。

> **注意：**
>
> 本指南中的步骤仅适用于快速上手体验，不适用于生产环境。如需探索 HTAP 更多功能，请参考深入探索 HTAP。

3.2.1 基础概念

在试用前，你需要对 TiDB 面向在线事务处理的行存储引擎TiKV 与面向实时分析场景的列存储引擎TiFlash 有一些基本了解：

- HTAP 存储引擎：行存 (Row-store) 与列存 (columnar-store) 同时存在，自动同步，保持强一致性。行存为在线事务处理 OLTP 提供优化，列存则为在线分析处理 OLAP 提供性能优化。
- HTAP 数据一致性：作为一个分布式事务型的键值数据库，TiKV 提供了满足 ACID 约束的分布式事务接口，并通过 Raft 协议保证了多副本数据一致性以及高可用。TiFlash 通过 Multi-Raft Learner 协议实时从 TiKV 复制数据，确保与 TiKV 之间的数据强一致。
- HTAP 数据隔离性：TiKV、TiFlash 可按需部署在不同的机器，解决 HTAP 资源隔离的问题。
- MPP 计算引擎：从 v5.0 版本起，TiFlash 引入了分布式计算框架MPP，允许节点之间的数据交换并提供高性能、高吞吐的 SQL 算法，可以大幅度缩短分析查询的执行时间。

3.2.2 体验步骤

本文的步骤以 TPC-H 数据集为例，通过其中一个查询场景来体验 TiDB HTAP 的便捷性和高性能。TPC-H 是业界较为流行的决策支持（Desision Support）业务 Benchmark。它包含大数据量下，一个业务决策分析系统所需要响应的不同类型高复杂度的即席查询。如果需要体验 TPC-H 完整的 22 条 SQL，可以访问 tidb-bench 仓库 或者阅读 TPC-H 官网说明了解如何生成查询语句以及数据。

3.2.2.1 第 1 步：部署试用环境

在试用 TiDB HTAP 功能前，请按照TiDB 数据库快速上手指南中的步骤准备 TiDB 本地测试环境，执行以下命令启动 TiDB 集群：

```
tiup playground
```

> **注意：**
>
> `tiup playground` 命令仅适用于快速上手体验，不适用于生产环境。

3.2.2.2 第 2 步：准备试用数据

通过以下步骤，将生成一个 TPC-H 数据集用于体验 TiDB HTAP 功能。如果你对 TPC-H 感兴趣，可查看其规格说明。

> 注意：
>
> 如果你想使用自己现有的数据进行分析查询，可以将数据迁移到 TiDB 中；如果你想自己设计并生成数据，可以通过 SQL 语句或相关工具生成。

1. 使用以下命令安装数据生成工具：

```
tiup install bench
```

2. 使用以下命令生成数据：

```
tiup bench tpch --sf=1 prepare
```

当命令行输出 `Finished` 时，表示数据生成完毕。

3. 运行以下 SQL 语句查看生成的数据：

```
SELECT CONCAT(table_schema,'.',table_name) AS 'Table Name', table_rows AS 'Number of Rows',
    ↪ CONCAT(ROUND(data_length/(1024*1024*1024),4),'G') AS 'Data Size', CONCAT(ROUND(
    ↪ index_length/(1024*1024*1024),4),'G') AS 'Index Size', CONCAT(ROUND((data_length+
    ↪ index_length)/(1024*1024*1024),4),'G') AS'Total'FROM information_schema.TABLES WHERE
    ↪  table_schema LIKE 'test';
```

从输出中可以看到，一共生成了八张表，最大的一张表数据量有 600 万行（由于数据是工具随机生成，所以实际的数据生成量以 SQL 实际查询到的值为准）。

```
sql +---------------+----------------+-----------+------------+---------+ | Table Name |
↪ Number of Rows | Data Size | Index Size | Total | +---------------+----------------+-----------+--------
↪  | test.nation | 25 | 0.0000G | 0.0000G | 0.0000G | | test.region | 5 | 0.0000G | 0.0000G
↪  | 0.0000G | | test.part | 200000 | 0.0245G | 0.0000G | 0.0245G | | test.supplier | 10000
↪  | 0.0014G | 0.0000G | 0.0014G | | test.partsupp | 800000 | 0.1174G | 0.0119G | 0.1293
↪ G | | test.customer | 150000 | 0.0242G | 0.0000G | 0.0242G | | test.orders | 1514336 |
↪ 0.1673G | 0.0000G | 0.1673G | | test.lineitem | 6001215 | 0.7756G | 0.0894G | 0.8651G |
↪ +---------------+----------------+-----------+------------+---------+ 8 rows in set (0.06
↪  sec)
```

这是一个商业订购系统的数据库。其中，`test.nation` 表是国家信息、`test.region` 表是地区信息、`test.part` 表是零件信息、`test.supplier` 表是供货商信息、`test.partsupp` 表是供货商的零件信息、`test.customer` 表是消费者信息、`test.orders` 表是订单信息、`test.lineitem` 表是在线商品的信息。

3.2.2.3 第 3 步：使用行存查询数据

执行以下 SQL 语句，你可以体验当只使用行存（大多数数据库）时 TiDB 的表现：

```
SELECT
    l_orderkey,
    SUM(
        l_extendedprice * (1 - l_discount)
    ) AS revenue,
    o_orderdate,
    o_shippriority
FROM
    customer,
    orders,
    lineitem
WHERE
    c_mktsegment = 'BUILDING'
AND c_custkey = o_custkey
AND l_orderkey = o_orderkey
AND o_orderdate < DATE '1996-01-01'
AND l_shipdate > DATE '1996-02-01'
GROUP BY
    l_orderkey,
    o_orderdate,
    o_shippriority
ORDER BY
    revenue DESC,
    o_orderdate
limit 10;
```

这是一个运送优先权查询，用于给出在指定日期之前尚未运送的订单中收入最高订单的优先权和潜在的收入。潜在的收入被定义为 `l_extendedprice * (1-l_discount)` 的和。订单按照收入的降序列出。在本示例中，此查询将列出潜在查询收入在前 10 的尚未运送的订单。

3.2.2.4 第 4 步：同步列存数据

TiFlash 部署完成后并不会自动同步 TiKV 数据，你可以在 MySQL 客户端向 TiDB 发送以下 DDL 命令指定需要同步到 TiFlash 的表。指定后，TiDB 将创建对应的 TiFlash 副本。

```
ALTER TABLE test.customer SET TIFLASH REPLICA 1;
ALTER TABLE test.orders SET TIFLASH REPLICA 1;
ALTER TABLE test.lineitem SET TIFLASH REPLICA 1;
```

如需查询 TiFlash 表的同步状态，请使用以下 SQL 语句：

```
SELECT * FROM information_schema.tiflash_replica WHERE TABLE_SCHEMA = 'test' and TABLE_NAME = '
    ↪ customer';
SELECT * FROM information_schema.tiflash_replica WHERE TABLE_SCHEMA = 'test' and TABLE_NAME = '
    ↪ orders';
```

```
SELECT * FROM information_schema.tiflash_replica WHERE TABLE_SCHEMA = 'test' and TABLE_NAME = '
    ↪ lineitem';
```

以上查询结果中：

- AVAILABLE 字段表示该表的 TiFlash 副本是否可用。1 代表可用，0 代表不可用。副本状态变为可用之后就不再改变。
- PROGRESS 字段代表同步进度，在 0.0~1.0 之间，1 代表 TiFlash 副本已经完成同步。

3.2.2.5 第 5 步：使用 HTAP 更快地分析数据

再次执行第 3 步中的 SQL 语句，你可以感受 TiDB HTAP 的表现。

对于创建了 TiFlash 副本的表，TiDB 优化器会自动根据代价估算选择是否使用 TiFlash 副本。如需查看实际是否选择了 TiFlash 副本，可以使用 desc 或 explain analyze 语句，例如：

```
explain analyze SELECT
    l_orderkey,
    SUM(
        l_extendedprice * (1 - l_discount)
    ) AS revenue,
    o_orderdate,
    o_shippriority
FROM
    customer,
    orders,
    lineitem
WHERE
    c_mktsegment = 'BUILDING'
AND c_custkey = o_custkey
AND l_orderkey = o_orderkey
AND o_orderdate < DATE '1996-01-01'
AND l_shipdate > DATE '1996-02-01'
GROUP BY
    l_orderkey,
    o_orderdate,
    o_shippriority
ORDER BY
    revenue DESC,
    o_orderdate
limit 10;
```

如果结果中出现 ExchangeSender 和 ExchangeReceiver 算子，表明 MPP 已生效。

此外，你也可以指定整个查询的各个计算部分都只使用 TiFlash 引擎，详情请参阅使用 TiDB 读取 TiFlash。

你可以对比两次的查询结果和查询性能。

3.2.3 探索更多

- TiDB HTAP 形态架构
- 深入探索 HTAP
- 使用 TiFlash

3.3 SQL 基本操作

成功部署 TiDB 集群之后，便可以在 TiDB 中执行 SQL 语句了。因为 TiDB 兼容 MySQL，你可以使用 MySQL 客户端连接 TiDB，并且大多数情况下可以直接执行 MySQL 语句。

SQL 是一门声明性语言，它是数据库用户与数据库交互的方式。它更像是一种自然语言，好像在用英语与数据库进行对话。本文档介绍基本的 SQL 操作。完整的 SQL 语句列表，参见 TiDB SQL 语法详解。

3.3.1 分类

SQL 语言通常按照功能划分成以下的 4 个部分：

- DDL (Data Definition Language)：数据定义语言，用来定义数据库对象，包括库、表、视图和索引等。
- DML (Data Manipulation Language)：数据操作语言，用来操作和业务相关的记录。
- DQL (Data Query Language)：数据查询语言，用来查询经过条件筛选的记录。
- DCL (Data Control Language)：数据控制语言，用来定义访问权限和安全级别。

常用的 DDL 功能是对象（如表、索引等）的创建、属性修改和删除，对应的命令分别是 CREATE、ALTER 和 DROP。

3.3.2 查看、创建和删除数据库

TiDB 语境中的 Database 或者说数据库，可以认为是表和索引等对象的集合。

使用 `SHOW DATABASES` 语句查看系统中数据库列表：

```
SHOW DATABASES;
```

使用名为 `mysql` 的数据库：

```
USE mysql;
```

使用 `SHOW TABLES` 语句查看数据库中的所有表。例如：

```
SHOW TABLES FROM mysql;
```

使用 `CREATE DATABASE` 语句创建数据库。语法如下：

```
CREATE DATABASE db_name [options];
```

例如，要创建一个名为 `samp_db` 的数据库，可使用以下语句：

```
CREATE DATABASE IF NOT EXISTS samp_db;
```

添加 `IF NOT EXISTS` 可防止发生错误。

使用 `DROP DATABASE` 语句删除数据库。例如：

```
DROP DATABASE samp_db;
```

3.3.3 创建、查看和删除表

使用 `CREATE TABLE` 语句创建表。语法如下：

```
CREATE TABLE table_name column_name data_type constraint;
```

例如，要创建一个名为 `person` 的表，包括编号、名字、生日等字段，可使用以下语句：

```
CREATE TABLE person (
    id INT(11),
    name VARCHAR(255),
    birthday DATE
    );
```

使用 `SHOW CREATE` 语句查看建表语句，即 DDL。例如：

```
SHOW CREATE TABLE person;
```

使用 `DROP TABLE` 语句删除表。例如：

```
DROP TABLE person;
```

3.3.4 创建、查看和删除索引

索引通常用于加速索引列上的查询。对于值不唯一的列，可使用 `CREATE INDEX` 或 `ALTER TABLE` 语句创建普通索引。例如：

```
CREATE INDEX person_id ON person (id);
```

或者：

```
ALTER TABLE person ADD INDEX person_id (id);
```

对于值唯一的列，可以创建唯一索引。例如：

```
CREATE UNIQUE INDEX person_unique_id ON person (id);
```

或者：

```
ALTER TABLE person ADD UNIQUE person_unique_id (id);
```

使用 SHOW INDEX 语句查看表内所有索引：

```
SHOW INDEX FROM person;
```

使用 ALTER TABLE 或 DROP INDEX 语句来删除索引。与 CREATE INDEX 语句类似，DROP INDEX 也可以嵌入 ALTER TABLE 语句。例如：

```
DROP INDEX person_id ON person;
```

```
ALTER TABLE person DROP INDEX person_unique_id;
```

注意：DDL 操作不是事务，在执行 DDL 时，不需要对应 COMMIT 语句。

常用的 DML 功能是对表记录的新增、修改和删除，对应的命令分别是 INSERT、UPDATE 和 DELETE。

3.3.5 记录的增删改

使用 INSERT 语句向表内插入表记录。例如：

```
INSERT INTO person VALUES(1,'tom','20170912');
```

使用 INSERT 语句向表内插入包含部分字段数据的表记录。例如：

```
INSERT INTO person(id,name) VALUES('2','bob');
```

使用 UPDATE 语句向表内修改表记录的部分字段数据。例如：

```
UPDATE person SET birthday='20180808' WHERE id=2;
```

使用 DELETE 语句向表内删除部分表记录。例如：

```
DELETE FROM person WHERE id=2;
```

注意：UPDATE 和 DELETE 操作如果不带 WHERE 过滤条件是对全表进行操作。

DQL 数据查询语言是从一个表或多个表中检索出想要的数据行，通常是业务开发的核心内容。

3.3.6 查询数据

使用 SELECT 语句检索表内数据。例如：

```
SELECT * FROM person;
```

在 SELECT 后面加上要查询的列名。例如：

```
SELECT name FROM person;
```

```
+------+
| name |
+------+
| tom  |
+------+
1 rows in set (0.00 sec)
```

使用 WHERE 子句，对所有记录进行是否符合条件的筛选后再返回。例如：

```
SELECT * FROM person WHERE id<5;
```

常用的 DCL 功能是创建或删除用户，和对用户权限的管理。

3.3.7 创建、授权和删除用户

使用 CREATE USER 语句创建一个用户 tiuser，密码为 123456：

```
CREATE USER 'tiuser'@'localhost' IDENTIFIED BY '123456';
```

授权用户 tiuser 可检索数据库 samp_db 内的表：

```
GRANT SELECT ON samp_db.* TO 'tiuser'@'localhost';
```

查询用户 tiuser 的权限：

```
SHOW GRANTS for tiuser@localhost;
```

删除用户 tiuser：

```
DROP USER 'tiuser'@'localhost';
```

3.4 HTAP 深入探索指南

本指南介绍如何进一步探索并使用 TiDB 在线事务与在线分析处理 (Hybrid Transactional and Analytical Processing, HTAP) 功能。

> 注意：
>
> 如果你对 TiDB HTAP 功能还不太了解，希望快速试用体验，请参阅快速上手 HTAP。

要快速了解 TiDB 在 HTAP 场景下的体系架构与 HTAP 的适用场景，建议先观看下面的培训视频（时长 15 分钟）。注意本视频只作为学习参考，如需了解详细的 HTAP 相关内容，请参阅下方的文档内容。

3.4.1 HTAP 适用场景

TiDB HTAP 可以满足企业海量数据的增产需求、降低运维的风险成本、与现有的大数据栈无缝缝合，从而实现数据资产价值的实时变现。

以下是三种 HTAP 典型适用场景：

- 混合负载场景

  当将 TiDB 应用于在线实时分析处理的混合负载场景时，开发人员只需要提供一个入口，TiDB 将自动根据业务类型选择不同的处理引擎。

- 实时流处理场景

  当将 TiDB 应用于实时流处理场景时，TiDB 能保证源源不断流入系统的数据实时可查，同时可兼顾高并发数据服务与 BI 查询。

- 数据中枢场景

  当将 TiDB 应用于数据中枢场景时，TiDB 作为数据中枢可以无缝连接数据业务层和数据仓库层，满足不同业务的需求。

如果想了解更多关于 TiDB HTAP 场景信息，请参阅 PingCAP 官网中关于 HTAP 的博客。

3.4.2 HTAP 架构

在 TiDB 中，面向在线事务处理的行存储引擎TiKV 与面向实时分析场景的列存储引擎TiFlash 同时存在，自动同步，保持强一致性。

更多架构信息，请参考TiDB HTAP 形态架构。

3.4.3 HTAP 环境准备

在深入探索 TiDB HTAP 功能前，请依据你的数据场景部署 TiDB 以及对应的数据分析引擎。大数据场景 (100 T) 下，推荐使用 TiFlash MPP 作为 HTAP 的主要方案，TiSpark 作为补充方案。

- TiFlash
  - 如果已经部署 TiDB 集群但尚未部署 TiFlash 节点，请参阅扩容 TiFlash 节点中的步骤在现有 TiDB 集群中添加 TiFlash 节点。
  - 如果尚未部署 TiDB 集群，请使用TiUP 部署 TiDB 集群，并在包含最小拓扑的基础上，同时增加 TiFlash 拓扑架构。
  - 在决定如何选择 TiFlash 节点数量时，请考虑以下几种业务场景：
    * 如果业务场景以 OLTP 为主，做轻量级的 Ad hoc OLAP 计算，通常部署 1 个或几个 TiFlash 节点就会产生明显的加速效果。
    * 当 OLTP 数据吞吐量对节点 I/O 无明显压力时，每个 TiFlash 节点将会使用较多资源用于计算，这样 TiFlash 集群可实现近似线性的扩展能力。TiFlash 节点数量应根据期待的性能和响应时间调整。

    * 当 OLTP 数据吞吐量较高时（例如写入或更新超过千万行/小时），由于网络和物理磁盘的写入能力有限，内部 TiKV 与 TiFlash 之间的 I/O 会成为主要瓶颈，也容易产生读写热点。此时 TiFlash 节点数与 OLAP 计算量有较复杂非线性关系，需要根据具体系统状态调整节点数量。

- TiSpark
    - 如果你的业务需要基于 Spark 进行分析，请部署 TiSpark。具体步骤，请参阅TiSpark 用户指南。

3.4.4 HTAP 数据准备

TiFlash 部署完成后并不会自动同步数据，你需要指定需要同步到 TiFlash 的数据表。指定后，TiDB 将创建对应的 TiFlash 副本。

- 如果 TiDB 集群中还没有数据，请先迁移数据到 TiDB。详情请参阅数据迁移。
- 如果 TiDB 集群中已经有从上游同步过来的数据，TiFlash 部署完成后并不会自动同步数据，而需要手动指定需要同步的表，详情请参阅使用 TiFlash。

3.4.5 HTAP 数据处理

使用 TiDB 时，你只需输入 SQL 语句进行查询或者写入需求。对于创建了 TiFlash 副本的表，TiDB 会依靠前端优化器自由选择最优的执行方式。

> 注意：
>
> TiFlash 的 MPP 模式默认开启。当执行 SQL 语句时，TiDB 会通过优化器自动判断并选择是否以 MPP 模式执行。
>
> - 如需关闭 MPP 模式，请将系统变量tidb_allow_mpp 的值设置为 OFF。
> - 如需强制使用 TiFlash 的 MPP 模式执行查询，请将系统变量tidb_allow_mpp 和tidb_enforce_mpp 的值设置为 ON。
> - 如需查看 TiDB 是否选择以 MPP 模式执行，你可以通过 EXPLAIN 语句查看具体的查询执行计划。如果 EXPLAIN 语句的结果中出现 ExchangeSender 和 ExchangeReceiver 算子，表明 MPP 已生效。

3.4.6 HTAP 性能监控

在 TiDB 的使用过程中，可以选择以下方式监控 TiDB 集群运行情况并查看性能数据。

- TiDB Dashboard：查看集群整体运行概况，分析集群读写流量分布及趋势变化，详细了解耗时较长的 SQL 语句的执行信息。
- 监控系统 (Prometheus & Grafana)：查看 TiDB 集群各组件（包括 PD、TiDB、TiKV、TiFlash、TiCDC、Node_exporter）的相关监控参数。

如需查看 TiDB 和 TiFlash 集群报警规则和处理方法，请查阅TiDB 集群报警规则和TiFlash 报警规则。

3.4.7 HTAP 故障诊断

在使用 TiDB 的过程中如果遇到问题，请参阅以下文档：

- 分析慢查询
- 定位消耗系统资源多的查询
- TiDB 热点问题处理
- TiDB 集群故障诊断
- TiFlash 常见问题

除此之外，你可以在 Github Issues 新建一个 Issue 反馈问题，或者在 AskTUG 提交你的问题。

3.4.8 探索更多

- 如果要查看 TiFlash 版本、TiFlash 重要日志及系统表，请参阅TiFlash 集群运维。
- 如果需要移除某个 TiFlash 节点，请参阅缩容 TiFlash 节点。

4 应用开发

4.1 开发者手册概览

本文是为应用程序开发者所编写的，如果你对 TiDB 的内部原理感兴趣，或希望参与到 TiDB 的开发中来，那么可前往阅读 TiDB Kernel Development Guide 来获取更多 TiDB 的相关信息。

本手册将展示如何使用 TiDB 来快速构建一个应用，并且阐述使用 TiDB 期间可能出现的场景以及可能会遇到的问题。因此，在阅读此页面之前，建议你先行阅读TiDB 数据库快速上手指南。

此外，你还可以通过视频的形式学习免费的 TiDB SQL 开发在线课程。

4.1.1 TiDB 基础

在你开始使用 TiDB 之前，你需要了解一些关于 TiDB 数据库的一些重要工作机制：

- 阅读TiDB 事务概览来了解 TiDB 的事务运作方式或查看为应用开发程序员准备的事务说明查看应用开发程序员需要了解的事务部分。
- 学习免费在线课程 TiDB 架构与特点，了解构建 TiDB 分布式数据库集群的核心组件及其概念。
- 了解应用程序与 TiDB 交互的方式。

4.1.2 TiDB 事务机制

TiDB 支持分布式事务，而且提供乐观事务与悲观事务两种事务模式。TiDB 当前版本中默认采用 悲观事务模式，这让你在 TiDB 事务时可以像使用传统的单体数据库 (如: MySQL) 事务一样。

你可以使用BEGIN 开启一个事务，或者使用 `BEGIN PESSIMISTIC` 显式的指定开启一个悲观事务，使用 `BEGIN OPTIMISTIC` 显式的指定开启一个乐观事务。随后，使用COMMIT 提交事务，或使用ROLLBACK 回滚事务。

TiDB 会为你保证 BEGIN 开始到 COMMIT 或 ROLLBACK 结束间的所有语句的原子性，即在这期间的所有语句全部成功，或者全部失败。用以保证你在应用开发时所需的数据一致性。

若你不清楚乐观事务是什么，请暂时不要使用它。因为使用乐观事务的前提是需要应用程序可以正确的处理 COMMIT 语句所返回的所有错误。如果不确定应用程序如何处理，请直接使用悲观事务。

4.1.3 应用程序与 TiDB 交互的方式

TiDB 高度兼容 MySQL 协议，TiDB 支持大多数 MySQL 的语法及特性，因此大部分的 MySQL 的连接库都与 TiDB 兼容。如果你的应用程序框架或语言无 PingCAP 的官方适配，那么建议你使用 MySQL 的客户端库。同时，也有越来越多的三方数据库主动支持 TiDB 的差异特性。

因为 TiDB 兼容 MySQL 协议，且兼容 MySQL 语法，因此大多数支持 MySQL 的 ORM 也兼容 TiDB。

4.1.4 扩展阅读

- 快速开始
- 选择驱动或 ORM 框架
- 连接到 TiDB
- 数据库模式设计
- 数据写入
- 数据读取
- 事务
- 优化 SQL 性能
- 示例程序

4.2 快速开始

4.2.1 使用 TiDB Serverless 构建 TiDB 集群

本章节将介绍如何以最快的方式开始使用 TiDB。你将使用 TiDB Cloud 创建并启动一个 TiDB Serverless 集群，使用 TiDB SQL 客户端，插入数据。随后将从示例程序读取出数据。

若你需要在本地计算机上启动 TiDB，请参阅本地启动 TiDB。

4.2.1.1 第 1 步：创建 TiDB Serverless 集群

1. 如果你还未拥有 TiDB Cloud 账号，请先在此注册。
2. 使用你的 TiDB Cloud 账号登录。

   登录后，默认进入 Clusters 页面。
3. 对于新注册的用户，TiDB Cloud 会自动为你创建一个 TiDB Serverless 集群 `Cluster0`。你可以使用这个默认集群进行后续操作，也可以自行创建一个新的 TiDB Serverless 集群。

   如果你想创建一个新的 TiDB Serverless 集群，请进行以下操作：

   1. 点击 Create Cluster。

2. Create Cluster 页面默认选择 Serverless。你可以根据需要修改集群名称、选择可用区，然后点击 Create。你的 TiDB Serverless 集群将于 30 秒后创建完毕。

4. 点击目标集群名称，进入集群概览页面，然后点击右上角的 Connect 按钮，弹出连接对话框。
5. 在对话框中，选择你需要的连接方式和操作系统并保存对应的连接字符串。下面连接到集群的步骤将以 MySQL 客户端为例。
6. 点击 Create password 生成随机密码。生成的密码不会再次显示，因此请将密码妥善保存。如果没有设置 root 密码，你将无法连接到集群。

> **注意：**
>
> 在连接到 TiDB Serverless 集群时，你需要给用户名加上前缀并使用单引号包裹用户名。你可以在 TiDB Cloud 用户名前缀 中获得更多信息。

4.2.1.2 第 2 步：连接到集群

1. 若未安装 MySQL 客户端，请选择自己的操作系统，按以下步骤安装。

   对于 macOS 操作系统，如果你没有安装 Homebrew，请参考 Homebrew 官网进行安装。

   ```
   brew install mysql-client
   ```

   在安装完成的命令行输出中，得到以下信息：

   ```
   mysql-client is keg-only, which means it was not symlinked into /opt/homebrew,
   because it conflicts with mysql (which contains client libraries).

   If you need to have mysql-client first in your PATH, run:
   echo 'export PATH="/opt/homebrew/opt/mysql-client/bin:$PATH"' >> ~/.zshrc

   For compilers to find mysql-client you may need to set:
   export LDFLAGS="-L/opt/homebrew/opt/mysql-client/lib"
   export CPPFLAGS="-I/opt/homebrew/opt/mysql-client/include"
   ```

   请运行其中的此行（命令行输出若与此处文档不一致，请以命令行输出为准）：

   ```
   echo 'export PATH="/opt/homebrew/opt/mysql-client/bin:$PATH"' >> ~/.zshrc
   ```

   完成后，生效该配置文件（例如 `~/.zshrc`），并验证 MySQL 客户端是否安装成功：

   ```
   source ~/.zshrc
   mysql --version
   ```

   预期会得到形如以下的输出：

   ```
   mysql  Ver 8.0.28 for macos12.0 on arm64 (Homebrew)
   ```

对于 Linux 操作系统，下面以 CentOS 7 为例：

```
yum install mysql
```

完成后，请验证 MySQL 客户端是否安装成功：

```
mysql --version
```

预期会得到形如以下的输出：

```
mysql  Ver 15.1 Distrib 5.5.68-MariaDB, for Linux (x86_64) using readline 5.1
```

2. 运行第 1 步中得到的连接字符串。

```
mysql --connect-timeout 15 -u '<prefix>.root' -h <host> -P 4000 -D test --ssl-mode=
    ↪ VERIFY_IDENTITY --ssl-ca=/etc/ssl/cert.pem -p
```

> 注意：
>
> - 在连接 TiDB Serverless 集群时，必须使用 TLS 连接。
> - 如果你在连接时遇到问题，可阅读 TiDB Serverless 集群安全连接 来获得更多信息。

3. 填写密码，完成登录。

4.2.1.3 第 3 步：运行 SQL

尝试运行一下你在 TiDB Cloud 上的的第一个 SQL 吧：

```
SELECT 'Hello TiDB Cloud!';
```

你将看到这样的输出：

```
+--------------------+
| Hello TiDB Cloud! |
+--------------------+
| Hello TiDB Cloud! |
+--------------------+
```

如果你的实际输出与预期输出一致，表示你已经在 TiDB Cloud 上成功地运行了 SQL 语句。

4.2.2 使用 TiDB 的增删改查 SQL

本章将简单介绍 TiDB 的增删改查 SQL 的使用方法。

4.2.2.1 在开始之前

请确保你已经连接到 TiDB 集群，若未连接，请参考使用 TiDB Serverless 构建 TiDB 集群来创建一个 TiDB Serverless 集群。

4.2.2.2 基本 SQL 操作

> 注意：
>
> 此处文档引用并简化自 TiDB 文档中的SQL 基本操作，你可直接前往此文档获取更全面、深入的 SQL 基本操作信息。

成功部署 TiDB 集群之后，便可以在 TiDB 中执行 SQL 语句了。因为 TiDB 兼容 MySQL，你可以使用 MySQL 客户端连接 TiDB，并且大多数情况下可以直接执行 MySQL 语句。

SQL 是一门声明性语言，它是数据库用户与数据库交互的方式。它更像是一种自然语言，好像在用英语与数据库进行对话。本文档介绍基本的 SQL 操作。完整的 SQL 语句列表，参见 TiDB SQL 语法详解。

4.2.2.3 分类

SQL 语言通常按照功能划分成以下的 4 个部分：

- DDL (Data Definition Language)：数据定义语言，用来定义数据库对象，包括库、表、视图和索引等。
- DML (Data Manipulation Language)：数据操作语言，用来操作和业务相关的记录。
- DQL (Data Query Language)：数据查询语言，用来查询经过条件筛选的记录。
- DCL (Data Control Language)：数据控制语言，用来定义访问权限和安全级别。

此文档中，主要介绍 DML 和 DQL，即数据操作语言和数据查询语言。其余部分可查看SQL 基本操作或 TiDB SQL 语法详解获得更多信息。

4.2.2.4 DML 数据操作语言

数据操作语言可完成数据的增删改。

使用 INSERT 语句向表内插入表记录。例如：

```
INSERT INTO person VALUES(1,'tom','20170912');
```

使用 INSERT 语句向表内插入包含部分字段数据的表记录。例如：

```
INSERT INTO person(id,name) VALUES('2','bob');
```

使用 UPDATE 语句向表内修改表记录的部分字段数据。例如：

```
UPDATE person SET birthday='20180808' WHERE id=2;
```

使用 DELETE 语句向表内删除部分表记录。例如：

```
DELETE FROM person WHERE id=2;
```

> 注意：
>
> UPDATE 和 DELETE 操作如果不带 WHERE 过滤条件是对全表进行操作。

4.2.2.5 DQL 数据查询语言

数据查询语言是从一个表或多个表中检索出想要的数据行，通常是业务开发的核心内容。

使用 SELECT 语句检索单表内数据。例如：

```
SELECT * FROM person;
```

在 SELECT 后面加上要查询的列名。例如：

```
SELECT name FROM person;
```

运行结果为：

```
+------+
| name |
+------+
| tom  |
+------+
1 rows in set (0.00 sec)
```

使用 WHERE 子句，对所有记录进行是否符合条件的筛选后再返回。例如：

```
SELECT * FROM person WHERE id < 5;
```

4.3 示例程序

4.3.1 TiDB 和 Golang 的简单 CRUD 应用程序

本文档将展示如何使用 TiDB 和 Golang 来构造一个简单的 CRUD 应用程序。

> 注意：
>
> 推荐使用 Golang 1.20 以上版本进行 TiDB 的应用程序的编写。

4.3.1.1 第 1 步：启动你的 TiDB 集群

本节将介绍 TiDB 集群的启动方法。

创建 TiDB Serverless 集群。

你可以部署一个本地测试的 TiDB 集群或正式的 TiDB 集群。详细步骤，请参考：

- 部署本地测试 TiDB 集群
- 部署正式 TiDB 集群

基于 Git 的预配置的开发环境：现在就试试

该环境会自动克隆代码，并通过 TiUP 部署测试集群。

4.3.1.2 第 2 步：获取代码

```
git clone https://github.com/pingcap-inc/tidb-example-golang.git
```

当前开源比较流行的 Golang ORM 为 GORM，此处将以 v1.23.5 版本进行说明。

封装一个用于适配 TiDB 事务的工具包 util，编写以下代码备用：

```
package util

import (
    "gorm.io/gorm"
)

// TiDBGormBegin start a TiDB and Gorm transaction as a block. If no error is returned, the
    ↪ transaction will be committed. Otherwise, the transaction will be rolled back.
func TiDBGormBegin(db *gorm.DB, pessimistic bool, fc func(tx *gorm.DB) error) (err error) {
    session := db.Session(&gorm.Session{})
    if session.Error != nil {
        return session.Error
    }

    if pessimistic {
        session = session.Exec("set @@tidb_txn_mode=pessimistic")
    } else {
        session = session.Exec("set @@tidb_txn_mode=optimistic")
    }

    if session.Error != nil {
        return session.Error
    }
    return session.Transaction(fc)
}
```

进入目录 gorm：

```
cd gorm
```

目录结构如下所示：

```
.
├── Makefile
├── go.mod
├── go.sum
└── gorm.go
```

其中，gorm.go 是 gorm 这个示例程序的主体。使用 gorm 时，相较于 go-sql-driver/mysql，gorm 屏蔽了创建数据库连接时，不同数据库差异的细节，其还封装了大量的操作，如 AutoMigrate、基本对象的 CRUD 等，极大的简化了代码量。

Player 是数据结构体，为数据库表在程序内的映射。Player 的每个属性都对应着 player 表的一个字段。相较于 go-sql-driver/mysql，gorm 的 Player 数据结构体为了给 gorm 提供更多的信息，加入了形如 `gorm:"primaryKey;type:VARCHAR(36);column:id"` 的注解，用来指示映射关系。

```
package main

import (
    "fmt"
    "math/rand"

    "github.com/google/uuid"
    "github.com/pingcap-inc/tidb-example-golang/util"

    "gorm.io/driver/mysql"
    "gorm.io/gorm"
    "gorm.io/gorm/clause"
    "gorm.io/gorm/logger"
)

type Player struct {
    ID    string `gorm:"primaryKey;type:VARCHAR(36);column:id"`
    Coins int    `gorm:"column:coins"`
    Goods int    `gorm:"column:goods"`
}

func (*Player) TableName() string {
    return "player"
}

func main() {
```

```
    // 1. Configure the example database connection.
    db := createDB()

    // AutoMigrate for player table
    db.AutoMigrate(&Player{})

    // 2. Run some simple examples.
    simpleExample(db)

    // 3. Explore more.
    tradeExample(db)
}

func tradeExample(db *gorm.DB) {
    // Player 1: id is "1", has only 100 coins.
    // Player 2: id is "2", has 114514 coins, and 20 goods.
    player1 := &Player{ID: "1", Coins: 100}
    player2 := &Player{ID: "2", Coins: 114514, Goods: 20}

    // Create two players "by hand", using the INSERT statement on the backend.
    db.Clauses(clause.OnConflict{UpdateAll: true}).Create(player1)
    db.Clauses(clause.OnConflict{UpdateAll: true}).Create(player2)

    // Player 1 wants to buy 10 goods from player 2.
    // It will cost 500 coins, but player 1 cannot afford it.
    fmt.Println("\nbuyGoods:\n    => this trade will fail")
    if err := buyGoods(db, player2.ID, player1.ID, 10, 500); err == nil {
        panic("there shouldn't be success")
    }

    // So player 1 has to reduce the incoming quantity to two.
    fmt.Println("\nbuyGoods:\n    => this trade will success")
    if err := buyGoods(db, player2.ID, player1.ID, 2, 100); err != nil {
        panic(err)
    }
}

func simpleExample(db *gorm.DB) {
    // Create a player, who has a coin and a goods..
    if err := db.Clauses(clause.OnConflict{UpdateAll: true}).
        Create(&Player{ID: "test", Coins: 1, Goods: 1}).Error; err != nil {
        panic(err)
    }

    // Get a player.
```

```
    var testPlayer Player
    db.Find(&testPlayer, "id = ?", "test")
    fmt.Printf("getPlayer: %+v\n", testPlayer)

    // Create players with bulk inserts. Insert 1919 players totally, with 114 players per batch.
    bulkInsertPlayers := make([]Player, 1919, 1919)
    total, batch := 1919, 114
    for i := 0; i < total; i++ {
        bulkInsertPlayers[i] = Player{
            ID:    uuid.New().String(),
            Coins: rand.Intn(10000),
            Goods: rand.Intn(10000),
        }
    }

    if err := db.Session(&gorm.Session{Logger: db.Logger.LogMode(logger.Error)}).
        CreateInBatches(bulkInsertPlayers, batch).Error; err != nil {
        panic(err)
    }

    // Count players amount.
    playersCount := int64(0)
    db.Model(&Player{}).Count(&playersCount)
    fmt.Printf("countPlayers: %d\n", playersCount)

    // Print 3 players.
    threePlayers := make([]Player, 3, 3)
    db.Limit(3).Find(&threePlayers)
    for index, player := range threePlayers {
        fmt.Printf("print %d player: %+v\n", index+1, player)
    }
}

func createDB() *gorm.DB {
    dsn := "root:@tcp(127.0.0.1:4000)/test?charset=utf8mb4"
    db, err := gorm.Open(mysql.Open(dsn), &gorm.Config{
        Logger: logger.Default.LogMode(logger.Info),
    })
    if err != nil {
        panic(err)
    }

    return db
}
```

```
func buyGoods(db *gorm.DB, sellID, buyID string, amount, price int) error {
    return util.TiDBGormBegin(db, true, func(tx *gorm.DB) error {
        var sellPlayer, buyPlayer Player
        if err := tx.Clauses(clause.Locking{Strength: "UPDATE"}).
            Find(&sellPlayer, "id = ?", sellID).Error; err != nil {
            return err
        }

        if sellPlayer.ID != sellID || sellPlayer.Goods < amount {
            return fmt.Errorf("sell player %s goods not enough", sellID)
        }

        if err := tx.Clauses(clause.Locking{Strength: "UPDATE"}).
            Find(&buyPlayer, "id = ?", buyID).Error; err != nil {
            return err
        }

        if buyPlayer.ID != buyID || buyPlayer.Coins < price {
            return fmt.Errorf("buy player %s coins not enough", buyID)
        }

        updateSQL := "UPDATE player set goods = goods + ?, coins = coins + ? WHERE id = ?"
        if err := tx.Exec(updateSQL, -amount, price, sellID).Error; err != nil {
            return err
        }

        if err := tx.Exec(updateSQL, amount, -price, buyID).Error; err != nil {
            return err
        }

        fmt.Println("\n[buyGoods]:\n    'trade success'")
        return nil
    })
}
```

使用 go-sql-driver/mysql 时，首先进入目录 `sqldriver`：

```
cd sqldriver
```

目录结构如下所示：

```
.
├── Makefile
├── dao.go
├── go.mod
├── go.sum
```

```
├── sql
│   └── dbinit.sql
├── sql.go
└── sqldriver.go
```

其中，dbinit.sql 为数据表初始化语句：

```
USE test;
DROP TABLE IF EXISTS player;

CREATE TABLE player (
    `id` VARCHAR(36),
    `coins` INTEGER,
    `goods` INTEGER,
   PRIMARY KEY (`id`)
);
```

sqldriver.go 是 sqldriver 这个示例程序的主体。因为 TiDB 与 MySQL 协议兼容，因此，需要初始化一个 MySQL 协议的数据源 db, err := sql.Open("mysql", dsn)，以此连接到 TiDB。并在其后，调用 dao.go 中的一系列方法，用来管理数据对象，进行增删改查等操作。

```
package main

import (
    "database/sql"
    "fmt"

    _ "github.com/go-sql-driver/mysql"
)

func main() {
    // 1. Configure the example database connection.
    dsn := "root:@tcp(127.0.0.1:4000)/test?charset=utf8mb4"
    openDB("mysql", dsn, func(db *sql.DB) {
        // 2. Run some simple examples.
        simpleExample(db)

        // 3. Explore more.
        tradeExample(db)
    })
}

func simpleExample(db *sql.DB) {
    // Create a player, who has a coin and a goods.
    err := createPlayer(db, Player{ID: "test", Coins: 1, Goods: 1})
    if err != nil {
```

```
        panic(err)
    }

    // Get a player.
    testPlayer, err := getPlayer(db, "test")
    if err != nil {
        panic(err)
    }
    fmt.Printf("getPlayer: %+v\n", testPlayer)

    // Create players with bulk inserts. Insert 1919 players totally, with 114 players per batch.

    err = bulkInsertPlayers(db, randomPlayers(1919), 114)
    if err != nil {
        panic(err)
    }

    // Count players amount.
    playersCount, err := getCount(db)
    if err != nil {
        panic(err)
    }
    fmt.Printf("countPlayers: %d\n", playersCount)

    // Print 3 players.
    threePlayers, err := getPlayerByLimit(db, 3)
    if err != nil {
        panic(err)
    }
    for index, player := range threePlayers {
        fmt.Printf("print %d player: %+v\n", index+1, player)
    }
}

func tradeExample(db *sql.DB) {
    // Player 1: id is "1", has only 100 coins.
    // Player 2: id is "2", has 114514 coins, and 20 goods.
    player1 := Player{ID: "1", Coins: 100}
    player2 := Player{ID: "2", Coins: 114514, Goods: 20}

    // Create two players "by hand", using the INSERT statement on the backend.
    if err := createPlayer(db, player1); err != nil {
        panic(err)
    }
    if err := createPlayer(db, player2); err != nil {
```

```
        panic(err)
    }

    // Player 1 wants to buy 10 goods from player 2.
    // It will cost 500 coins, but player 1 cannot afford it.
    fmt.Println("\nbuyGoods:\n    => this trade will fail")
    if err := buyGoods(db, player2.ID, player1.ID, 10, 500); err == nil {
        panic("there shouldn't be success")
    }

    // So player 1 has to reduce the incoming quantity to two.
    fmt.Println("\nbuyGoods:\n    => this trade will success")
    if err := buyGoods(db, player2.ID, player1.ID, 2, 100); err != nil {
        panic(err)
    }
}

func openDB(driverName, dataSourceName string, runnable func(db *sql.DB)) {
    db, err := sql.Open(driverName, dataSourceName)
    if err != nil {
        panic(err)
    }
    defer db.Close()

    runnable(db)
}
```

随后，封装一个用于适配 TiDB 事务的工具包 util，编写以下代码备用：

```
package util

import (
    "context"
    "database/sql"
)

type TiDBSqlTx struct {
    *sql.Tx
    conn        *sql.Conn
    pessimistic bool
}

func TiDBSqlBegin(db *sql.DB, pessimistic bool) (*TiDBSqlTx, error) {
    ctx := context.Background()
    conn, err := db.Conn(ctx)
    if err != nil {
```

```
        return nil, err
    }
    if pessimistic {
        _, err = conn.ExecContext(ctx, "set @@tidb_txn_mode=?", "pessimistic")
    } else {
        _, err = conn.ExecContext(ctx, "set @@tidb_txn_mode=?", "optimistic")
    }
    if err != nil {
        return nil, err
    }
    tx, err := conn.BeginTx(ctx, nil)
    if err != nil {
        return nil, err
    }
    return &TiDBSqlTx{
        conn:        conn,
        Tx:          tx,
        pessimistic: pessimistic,
    }, nil
}

func (tx *TiDBSqlTx) Commit() error {
    defer tx.conn.Close()
    return tx.Tx.Commit()
}

func (tx *TiDBSqlTx) Rollback() error {
    defer tx.conn.Close()
    return tx.Tx.Rollback()
}
```

在 dao.go 中定义一系列数据的操作方法，用来对提供数据的写入能力。这也是本例子中和核心部分。

```
package main

import (
    "database/sql"
    "fmt"
    "math/rand"
    "strings"

    "github.com/google/uuid"
    "github.com/pingcap-inc/tidb-example-golang/util"
)

type Player struct {
```

```
    ID    string
    Coins int
    Goods int
}

// createPlayer create a player
func createPlayer(db *sql.DB, player Player) error {
    _, err := db.Exec(CreatePlayerSQL, player.ID, player.Coins, player.Goods)
    return err
}

// getPlayer get a player
func getPlayer(db *sql.DB, id string) (Player, error) {
    var player Player

    rows, err := db.Query(GetPlayerSQL, id)
    if err != nil {
        return player, err
    }
    defer rows.Close()

    if rows.Next() {
        err = rows.Scan(&player.ID, &player.Coins, &player.Goods)
        if err == nil {
            return player, nil
        } else {
            return player, err
        }
    }

    return player, fmt.Errorf("can not found player")
}

// getPlayerByLimit get players by limit
func getPlayerByLimit(db *sql.DB, limit int) ([]Player, error) {
    var players []Player

    rows, err := db.Query(GetPlayerByLimitSQL, limit)
    if err != nil {
        return players, err
    }
    defer rows.Close()

    for rows.Next() {
        player := Player{}
```

```
        err = rows.Scan(&player.ID, &player.Coins, &player.Goods)
        if err == nil {
            players = append(players, player)
        } else {
            return players, err
        }
    }

    return players, nil
}

// bulk-insert players
func bulkInsertPlayers(db *sql.DB, players []Player, batchSize int) error {
    tx, err := util.TiDBSqlBegin(db, true)
    if err != nil {
        return err
    }

    stmt, err := tx.Prepare(buildBulkInsertSQL(batchSize))
    if err != nil {
        return err
    }

    defer stmt.Close()

    for len(players) > batchSize {
        if _, err := stmt.Exec(playerToArgs(players[:batchSize])...); err != nil {
            tx.Rollback()
            return err
        }

        players = players[batchSize:]
    }

    if len(players) != 0 {
        if _, err := tx.Exec(buildBulkInsertSQL(len(players)), playerToArgs(players)...); err !=
            ↪ nil {
            tx.Rollback()
            return err
        }
    }

    if err := tx.Commit(); err != nil {
        tx.Rollback()
        return err
```

```
    }

    return nil
}

func getCount(db *sql.DB) (int, error) {
    count := 0

    rows, err := db.Query(GetCountSQL)
    if err != nil {
        return count, err
    }

    defer rows.Close()

    if rows.Next() {
        if err := rows.Scan(&count); err != nil {
            return count, err
        }
    }

    return count, nil
}

func buyGoods(db *sql.DB, sellID, buyID string, amount, price int) error {
    var sellPlayer, buyPlayer Player

    tx, err := util.TiDBSqlBegin(db, true)
    if err != nil {
        return err
    }

    buyExec := func() error {
        stmt, err := tx.Prepare(GetPlayerWithLockSQL)
        if err != nil {
            return err
        }
        defer stmt.Close()

        sellRows, err := stmt.Query(sellID)
        if err != nil {
            return err
        }
        defer sellRows.Close()
```

```
if sellRows.Next() {
    if err := sellRows.Scan(&sellPlayer.ID, &sellPlayer.Coins, &sellPlayer.Goods); err !=
        ↪  nil {
        return err
    }
}
sellRows.Close()

if sellPlayer.ID != sellID || sellPlayer.Goods < amount {
    return fmt.Errorf("sell player %s goods not enough", sellID)
}

buyRows, err := stmt.Query(buyID)
if err != nil {
    return err
}
defer buyRows.Close()

if buyRows.Next() {
    if err := buyRows.Scan(&buyPlayer.ID, &buyPlayer.Coins, &buyPlayer.Goods); err != nil
        ↪  {
        return err
    }
}
buyRows.Close()

if buyPlayer.ID != buyID || buyPlayer.Coins < price {
    return fmt.Errorf("buy player %s coins not enough", buyID)
}

updateStmt, err := tx.Prepare(UpdatePlayerSQL)
if err != nil {
    return err
}
defer updateStmt.Close()

if _, err := updateStmt.Exec(-amount, price, sellID); err != nil {
    return err
}

if _, err := updateStmt.Exec(amount, -price, buyID); err != nil {
    return err
}

return nil
```

```
    }

    err = buyExec()
    if err == nil {
        fmt.Println("\n[buyGoods]:\n    'trade success'")
        tx.Commit()
    } else {
        tx.Rollback()
    }

    return err
}

func playerToArgs(players []Player) []interface{} {
    var args []interface{}
    for _, player := range players {
        args = append(args, player.ID, player.Coins, player.Goods)
    }
    return args
}

func buildBulkInsertSQL(amount int) string {
    return CreatePlayerSQL + strings.Repeat(",(?,?,?)", amount-1)
}

func randomPlayers(amount int) []Player {
    players := make([]Player, amount, amount)
    for i := 0; i < amount; i++ {
        players[i] = Player{
            ID:    uuid.New().String(),
            Coins: rand.Intn(10000),
            Goods: rand.Intn(10000),
        }
    }

    return players
}
```

sql.go 中存放了 SQL 语句的常量。

```
package main

const (
    CreatePlayerSQL      = "INSERT INTO player (id, coins, goods) VALUES (?, ?, ?)"
    GetPlayerSQL         = "SELECT id, coins, goods FROM player WHERE id = ?"
    GetCountSQL          = "SELECT count(*) FROM player"
```

```
    GetPlayerWithLockSQL = GetPlayerSQL + " FOR UPDATE"
    UpdatePlayerSQL      = "UPDATE player set goods = goods + ?, coins = coins + ? WHERE id = ?"
    GetPlayerByLimitSQL  = "SELECT id, coins, goods FROM player LIMIT ?"
)
```

4.3.1.3 第 3 步：运行代码

本节将逐步介绍代码的运行方法。

4.3.1.3.1 第 3 步第 1 部分：go-sql-driver/mysql 表初始化

> 注意：
>
> 在 Gitpod Playground 中尝试 GORM: 现在就试试

无需手动初始化表。

> 注意：
>
> 在 Gitpod Playground 中尝试 go-sql-driver/mysql: 现在就试试

使用 go-sql-driver/mysql 时，需手动初始化数据库表，若你本地已经安装了 `mysql-client`，且使用本地集群，可直接在 `sqldriver` 目录下运行：

```
make mysql
```

或直接执行：

```
mysql --host 127.0.0.1 --port 4000 -u root<sql/dbinit.sql
```

若你不使用本地集群，或未安装 mysql-client，请直接登录你的集群，并运行 `sql/dbinit.sql` 文件内的 SQL 语句。

4.3.1.3.2 第 3 步第 2 部分：TiDB Cloud 更改参数

若你使用 TiDB Serverless 集群，更改 `gorm.go` 内 `dsn` 参数值：

```
dsn := "root:@tcp(127.0.0.1:4000)/test?charset=utf8mb4"
```

若你设定的密码为 `123456`，而且从 TiDB Serverless 集群面板中得到的连接信息为：

- Endpoint: `xxx.tidbcloud.com`

- Port: 4000
- User: 2aEp24QWEDLqRFs.root

那么此处应将 dsn 更改为：

```
dsn := "2aEp24QWEDLqRFs.root:123456@tcp(xxx.tidbcloud.com:4000)/test?charset=utf8mb4&tls=true"
```

若你使用 TiDB Serverless 集群，更改 sqldriver.go 内 dsn 参数的值：

```
dsn := "root:@tcp(127.0.0.1:4000)/test?charset=utf8mb4"
```

若你设定的密码为 123456，而且从 TiDB Serverless 集群面板中得到的连接信息为：

- Endpoint: xxx.tidbcloud.com
- Port: 4000
- User: 2aEp24QWEDLqRFs.root

那么此处应将 mysql.RegisterTLSConfig 和 dsn 更改为：

```
mysql.RegisterTLSConfig("register-tidb-tls", &tls.Config {
    MinVersion: tls.VersionTLS12,
    ServerName: "xxx.tidbcloud.com",
})

dsn := "2aEp24QWEDLqRFs.root:123456@tcp(xxx.tidbcloud.com:4000)/test?charset=utf8mb4&tls=register
    ↪ -tidb-tls"
```

4.3.1.3.3 第 3 步第 3 部分：运行

运行 make all，这是以下两个操作的组合：

- 构建二进制 (make build)：go build -o bin/gorm-example
- 运行 (make run)：./bin/gorm-example

你也可以单独运行这两个 make 命令或原生命令。

运行 make all，这是以下三个操作的组合：

- 创建表 (make mysql)：mysql --host 127.0.0.1 --port 4000 -u root<sql/dbinit.sql
- 构建二进制 (make build)：go build -o bin/sql-driver-example
- 运行 (make run)：./bin/sql-driver-example

你也可以单独运行这三个 make 命令或原生命令。

4.3.1.4 第 4 步：预期输出

GORM 预期输出

go-sql-driver/mysql 预期输出

4.3.2 使用 Spring Boot 构建 TiDB 应用程序

本教程向你展示如何使用 TiDB 构建 Spring Boot Web 应用程序。使用 Spring Data JPA 模块作为数据访问能力的框架。此示例应用程序的代码仓库可在 Github 下载。

这是一个较为完整的构建 Restful API 的示例应用程序，展示了一个使用 TiDB 作为数据库的通用 Spring Boot 后端服务。设计了以下过程，用于还原一个现实场景：

这是一个关于游戏的例子，每个玩家有两个属性：金币数 `coins` 和货物数 `goods`。且每个玩家都拥有一个字段 `id`，作为玩家的唯一标识。玩家在金币数和货物数充足的情况下，可以自由的交易。

你可以以此示例为基础，构建自己的应用程序。

> 建议：
>
> 在云原生开发环境中尝试 Spring Boot 构建 TiDB 应用程序。预配置完成的环境，自动启动 TiDB 集群，获取和运行代码，只需要一个链接。
>
> 现在就试试

4.3.2.1 第 1 步：启动你的 TiDB 集群

本节将介绍 TiDB 集群的启动方法。

创建 TiDB Serverless 集群。

你可以部署一个本地测试的 TiDB 集群或正式的 TiDB 集群。详细步骤，请参考：

- 部署本地测试 TiDB 集群
- 部署正式 TiDB 集群

基于 Git 的预配置的开发环境：现在就试试

该环境会自动克隆代码，并通过 TiUP 部署测试集群。

4.3.2.2 第 2 步：安装 JDK

请在你的计算机上下载并安装 Java Development Kit (JDK)，这是 Java 开发的必备工具。Spring Boot 支持 Java 版本 8 以上的 JDK，由于 Hibernate 版本的缘故，推荐使用 Java 版本 11 以上的 JDK。

示例应用程序同时支持 Oracle JDK 和 OpenJDK，请自行选择，本教程将使用版本 17 的 OpenJDK。

4.3.2.3 第 3 步：安装 Maven

此示例应用程序使用 Maven 来管理应用程序的依赖项。Spring 支持的 Maven 版本为 3.2 以上，作为依赖管理软件，推荐使用当前最新稳定版本的 Maven。

这里给出命令行安装 Maven 的办法：

- macOS 安装：

```
brew install maven
```

- 基于 Debian 的 Linux 发行版上安装（如 Ubuntu 等）：

```
apt-get install maven
```

- 基于 Red Hat 的 Linux 发行版上安装（如 Fedora、CentOS 等）：
- dnf 包管理器

```
dnf install maven
```

- yum 包管理器

```
yum install maven
```

其他安装方法，请参考 Maven 官方文档。

4.3.2.4 第 4 步：获取应用程序代码

> **建议：**
>
> 如果你希望得到一个与本示例相同依赖的空白程序，而无需示例代码，可参考创建相同依赖空白程序（可选）一节。

请下载或克隆示例代码库 pingcap-inc/tidb-example-java，并进入到目录 `spring-jpa-hibernate` 中。

4.3.2.5 第 5 步：运行应用程序

接下来运行应用程序代码，将会生成一个 Web 应用程序。Hibernate 将在数据库 `test` 中创建一个表 `player_jpa`。如果你向应用程序的 Restful API 发送请求，这些请求将会在 TiDB 集群上运行数据库事务。

如果你想了解有关此应用程序的代码的详细信息，可参阅实现细节部分。

4.3.2.5.1 第 5 步第 1 部分：TiDB Cloud 更改参数

若你使用 TiDB Serverless 集群，更改 `application.yml`（位于 `src/main/resources` 内）关于 `spring.datasource.url`、`spring.datasource.username`、`spring.datasource.password` 的参数：

```
spring:
  datasource:
    url: jdbc:mysql://localhost:4000/test
    username: root
    #    password: xxx
```

```
    driver-class-name: com.mysql.cj.jdbc.Driver
  jpa:
    show-sql: true
    database-platform: org.hibernate.dialect.TiDBDialect
    hibernate:
      ddl-auto: create-drop
```

若你设定的密码为 123456，而且从 TiDB Serverless 集群面板中得到的连接信息为：

- Endpoint: xxx.tidbcloud.com
- Port: 4000
- User: 2aEp24QWEDLqRFs.root

那么此处应将参数更改为：

```
spring:
  datasource:
    url: jdbc:mysql://xxx.tidbcloud.com:4000/test?sslMode=VERIFY_IDENTITY&enabledTLSProtocols=
        ↪ TLSv1.2,TLSv1.3
    username: 2aEp24QWEDLqRFs.root
    password: 123456
    driver-class-name: com.mysql.cj.jdbc.Driver
  jpa:
    show-sql: true
    database-platform: org.hibernate.dialect.TiDBDialect
    hibernate:
      ddl-auto: create-drop
```

4.3.2.5.2 第 5 步第 2 部分：运行

打开终端，进入 tidb-example-java/spring-jpa-hibernate 代码示例目录：

```
cd <path>/tidb-example-java/spring-jpa-hibernate
```

使用 Make 构建并运行（推荐）

```
make
```

手动构建并运行

推荐你使用 Make 方式进行构建并运行，当然，若你希望手动进行构建，请依照以下步骤逐步运行，可以得到相同的结果：

清除缓存并打包：

```
mvn clean package
```

运行应用程序的 JAR 文件：

```
java -jar target/spring-jpa-hibernate-0.0.1.jar
```

4.3.2.5.3 第 5 步第 3 部分：输出

输出的最后部分应如下所示：

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v3.0.1)

2023-01-05T14:06:54.427+08:00  INFO 22005 --- [           main] com.pingcap.App
    ↪                           : Starting App using Java 17.0.2 with PID 22005 (/Users/cheese/
    ↪ IdeaProjects/tidb-example-java/spring-jpa-hibernate/target/classes started by cheese in /
    ↪ Users/cheese/IdeaProjects/tidb-example-java)
2023-01-05T14:06:54.428+08:00  INFO 22005 --- [           main] com.pingcap.App
    ↪                           : No active profile set, falling back to 1 default profile: "
    ↪ default"
2023-01-05T14:06:54.642+08:00  INFO 22005 --- [           main] .s.d.r.c.
    ↪ RepositoryConfigurationDelegate : Bootstrapping Spring Data JPA repositories in DEFAULT
    ↪ mode.
2023-01-05T14:06:54.662+08:00  INFO 22005 --- [           main] .s.d.r.c.
    ↪ RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 17 ms.
    ↪ Found 1 JPA repository interfaces.
2023-01-05T14:06:54.830+08:00  INFO 22005 --- [           main] o.s.b.w.embedded.tomcat.
    ↪ TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
2023-01-05T14:06:54.833+08:00  INFO 22005 --- [           main] o.apache.catalina.core.
    ↪ StandardService   : Starting service [Tomcat]
2023-01-05T14:06:54.833+08:00  INFO 22005 --- [           main] o.apache.catalina.core.
    ↪ StandardEngine    : Starting Servlet engine: [Apache Tomcat/10.1.4]
2023-01-05T14:06:54.865+08:00  INFO 22005 --- [           main] o.a.c.c.C.[Tomcat].[localhost
    ↪ ].[/]       : Initializing Spring embedded WebApplicationContext
2023-01-05T14:06:54.865+08:00  INFO 22005 --- [           main] w.s.c.
    ↪ ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed
    ↪  in 421 ms
2023-01-05T14:06:54.916+08:00  INFO 22005 --- [           main] o.hibernate.jpa.internal.util.
    ↪ LogHelper  : HHH000204: Processing PersistenceUnitInfo [name: default]
2023-01-05T14:06:54.929+08:00  INFO 22005 --- [           main] org.hibernate.Version
    ↪                    : HHH000412: Hibernate ORM core version 6.1.6.Final
2023-01-05T14:06:54.969+08:00  WARN 22005 --- [           main] org.hibernate.orm.deprecation
    ↪            : HHH90000021: Encountered deprecated setting [javax.persistence.sharedCache.
    ↪ mode], use [jakarta.persistence.sharedCache.mode] instead
2023-01-05T14:06:55.005+08:00  INFO 22005 --- [           main] com.zaxxer.hikari.
    ↪ HikariDataSource       : HikariPool-1 - Starting...
2023-01-05T14:06:55.074+08:00  INFO 22005 --- [           main] com.zaxxer.hikari.pool.HikariPool
```

```
    ↪         : HikariPool-1 - Added connection com.mysql.cj.jdbc.ConnectionImpl@5e905f2c
2023-01-05T14:06:55.075+08:00  INFO 22005 --- [           main] com.zaxxer.hikari.
    ↪ HikariDataSource       : HikariPool-1 - Start completed.
2023-01-05T14:06:55.089+08:00  INFO 22005 --- [           main] SQL dialect
    ↪                              : HHH000400: Using dialect: org.hibernate.dialect.
    ↪ TiDBDialect
Hibernate: drop table if exists player_jpa
Hibernate: drop sequence player_jpa_id_seq
Hibernate: create sequence player_jpa_id_seq start with 1 increment by 1
Hibernate: create table player_jpa (id bigint not null, coins integer, goods integer, primary key
    ↪  (id)) engine=InnoDB
2023-01-05T14:06:55.332+08:00  INFO 22005 --- [           main] o.h.e.t.j.p.i.
    ↪ JtaPlatformInitiator       : HHH000490: Using JtaPlatform implementation: [org.hibernate.
    ↪ engine.transaction.jta.platform.internal.NoJtaPlatform]
2023-01-05T14:06:55.335+08:00  INFO 22005 --- [           main] j.
    ↪ LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for
    ↪ persistence unit 'default'
2023-01-05T14:06:55.579+08:00  WARN 22005 --- [           main]
    ↪ JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default.
    ↪  Therefore, database queries may be performed during view rendering. Explicitly configure
    ↪  spring.jpa.open-in-view to disable this warning
2023-01-05T14:06:55.710+08:00  INFO 22005 --- [           main] o.s.b.w.embedded.tomcat.
    ↪ TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
2023-01-05T14:06:55.714+08:00  INFO 22005 --- [           main] com.pingcap.App
    ↪                           : Started App in 1.432 seconds (process running for 1.654)
```

输出日志中，提示应用程序在启动过程中做了什么，这里显示应用程序使用 Tomcat 启动了一个 Servlet，使用 Hibernate 作为 ORM，HikariCP 作为数据库连接池的实现，使用了 `org.hibernate.dialect.TiDBDialect` 作为数据库方言。启动后，Hibernate 删除并重新创建了表 `player_jpa`，及序列 `player_jpa_id_seq`。在启动的最后，监听了 8080 端口，对外提供 HTTP 服务。

如果你想了解有关此应用程序的代码的详细信息，可参阅本教程下方的实现细节。

4.3.2.6 第 6 步：HTTP 请求

在运行应用程序后，你可以通过访问根地址 `http://localhost:8000` 向后端程序发送 HTTP 请求。下面将给出一些示例请求来演示如何使用该服务。

1. 将配置文件 `Player.postman_collection.json` 导入 Postman。
2. 导入后 Collections > Player 如图所示：

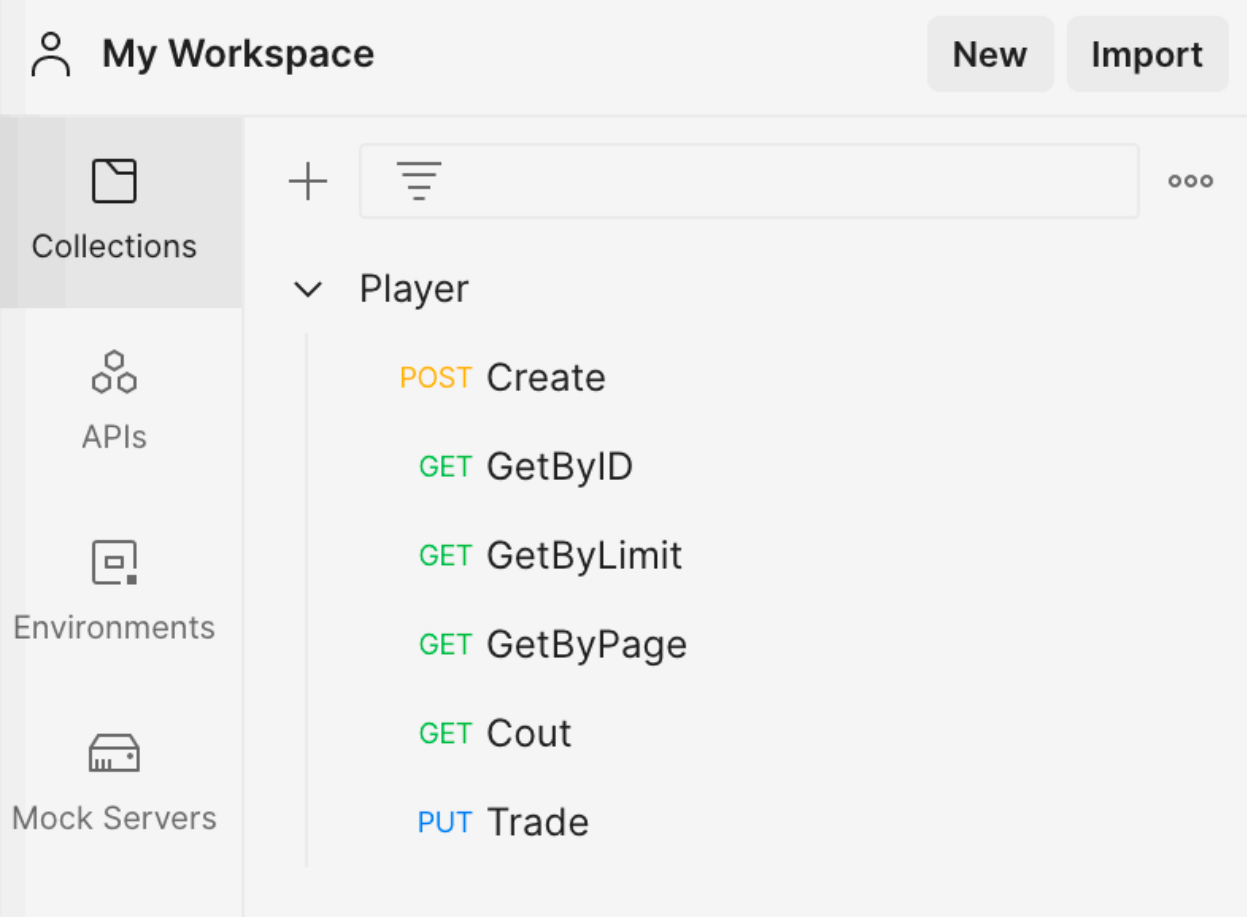


图 1: postman import

3. 发送请求：

   - 增加玩家

     点击 Create 标签，点击 Send 按钮，发送 POST 形式的 `http://localhost:8000/player/` 请求。返回值为增加的玩家个数，预期为 1。

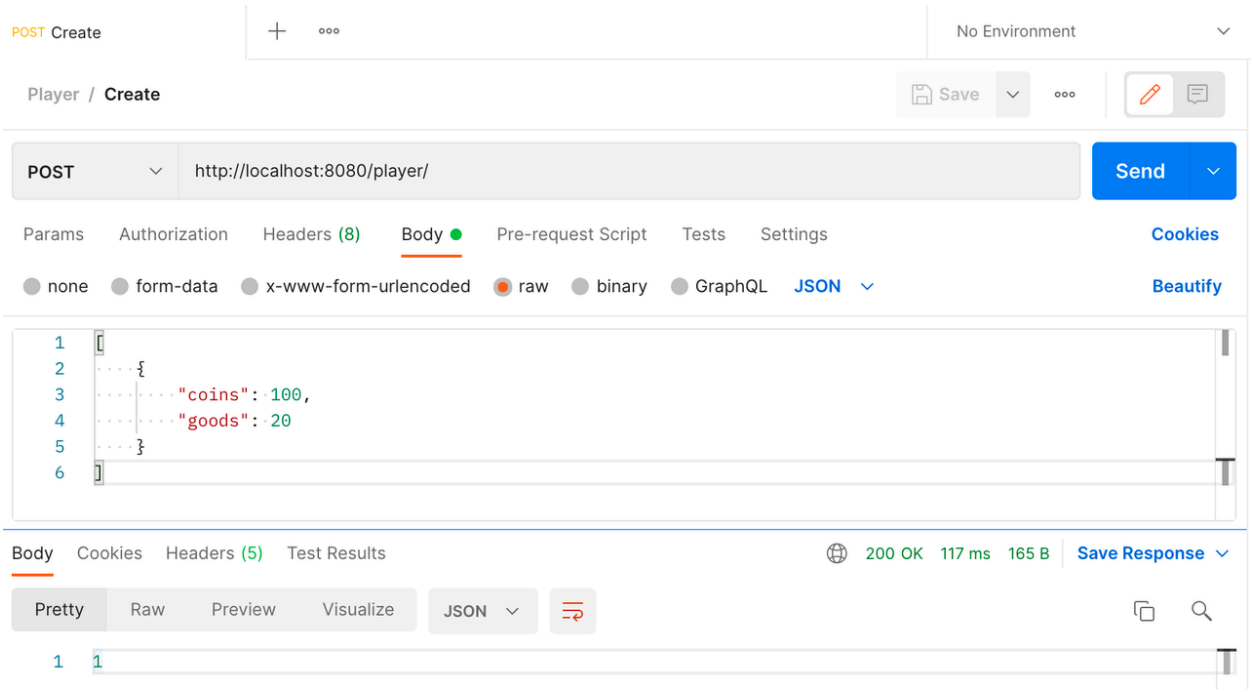


图 2: Postman-Create

- 使用 ID 获取玩家信息

  点击 GetByID 标签，点击 Send 按钮，发送 GET 形式的 `http://localhost:8000/player/1` 请求。返回值为 ID 为 1 的玩家信息。

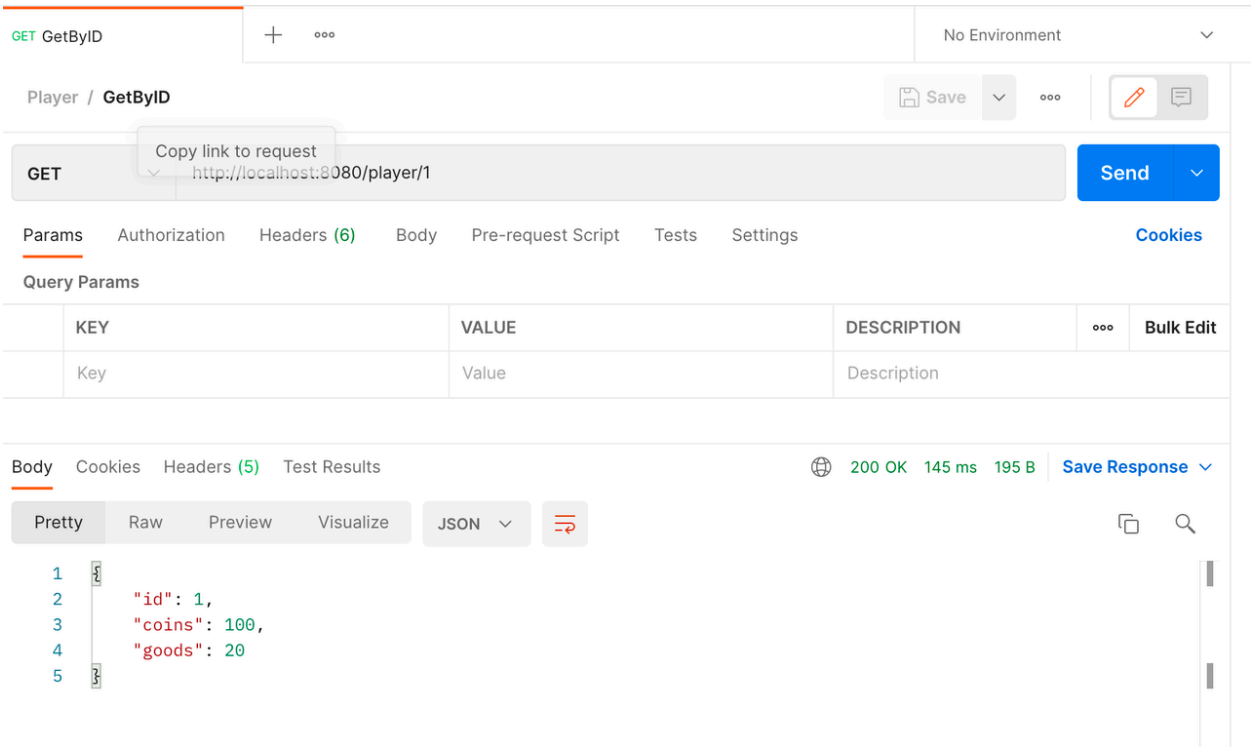


图 3: Postman-GetByID

- 使用 Limit 批量获取玩家信息

  点击 GetByLimit 标签，点击 Send 按钮，发送 GET 形式的 `http://localhost:8000/player/limit/3` 请求。返回值为最多 3 个玩家的信息列表。

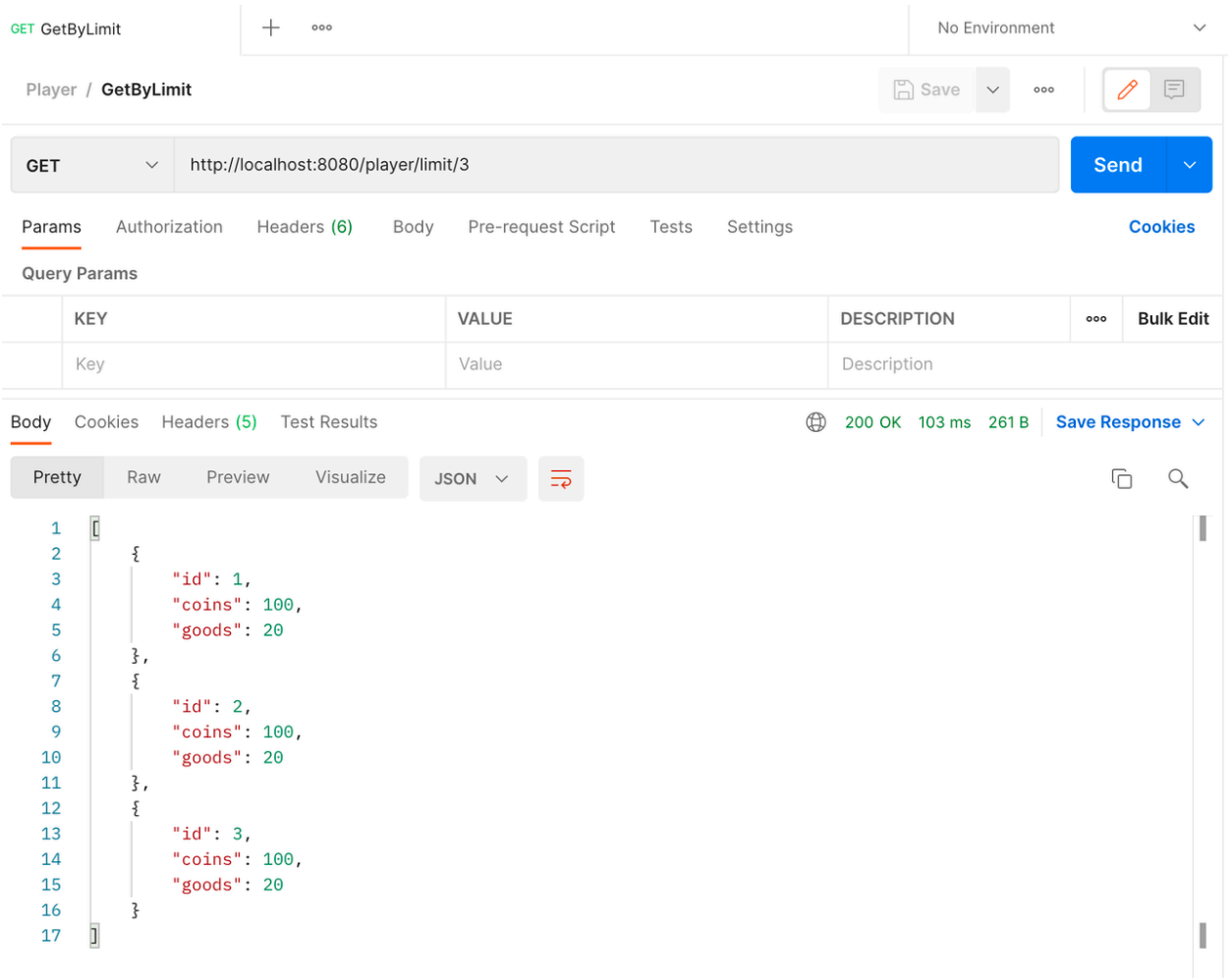


图 4: Postman-GetByLimit

- 分页获取玩家信息

  点击 GetByPage 标签，点击 Send 按钮，发送 GET 形式的 `http://localhost:8080/player/page?index=0&size=2` 请求。返回值为 index 为 0 的页，每页有 2 个玩家信息列表。此外，还包含了分页信息，如偏移量、总页数、是否排序等。

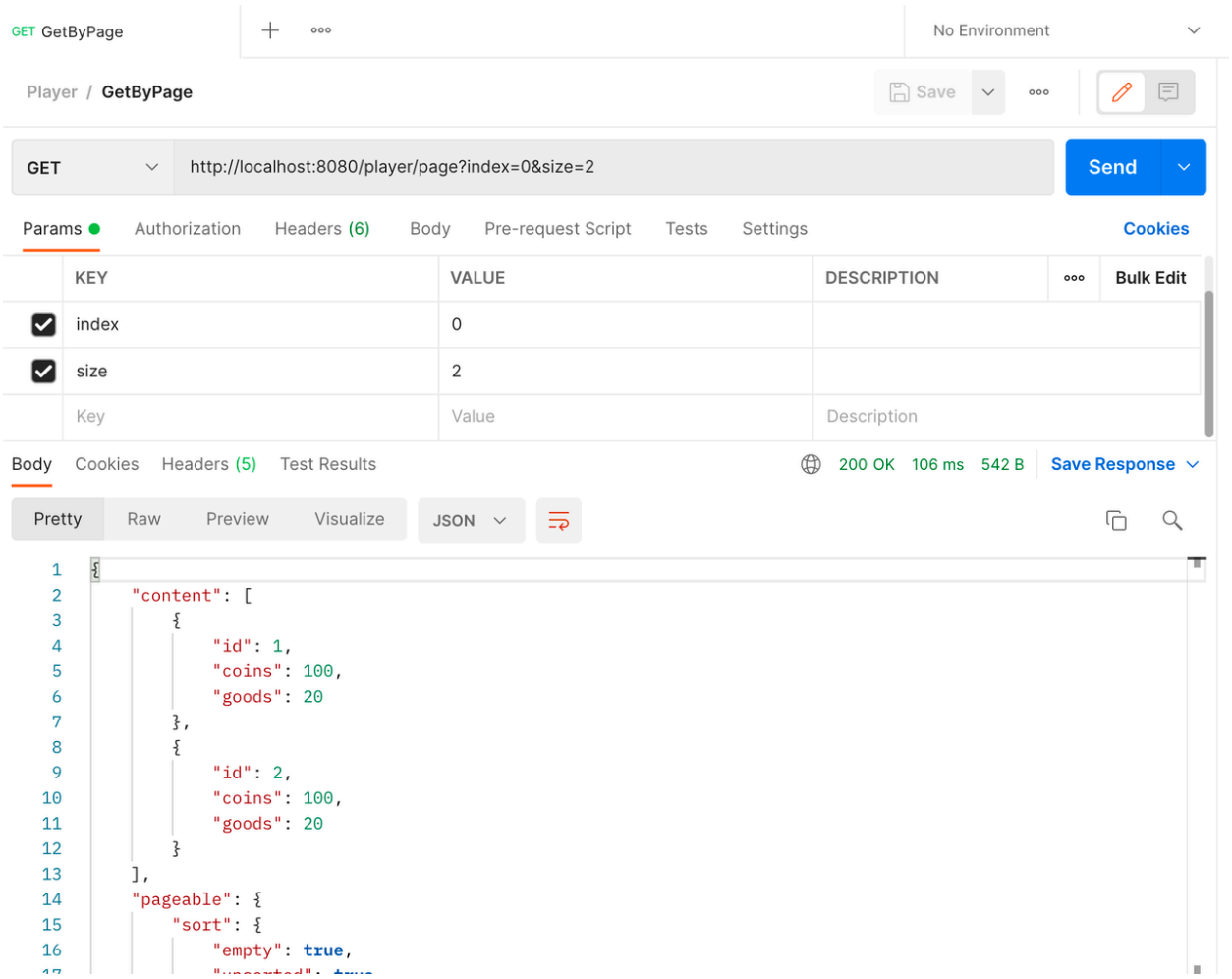


图 5: Postman-GetByPage

- 获取玩家个数

  点击 Count 标签，点击 Send 按钮，发送 GET 形式的 `http://localhost:8000/player/count` 请求。返回值为玩家个数。

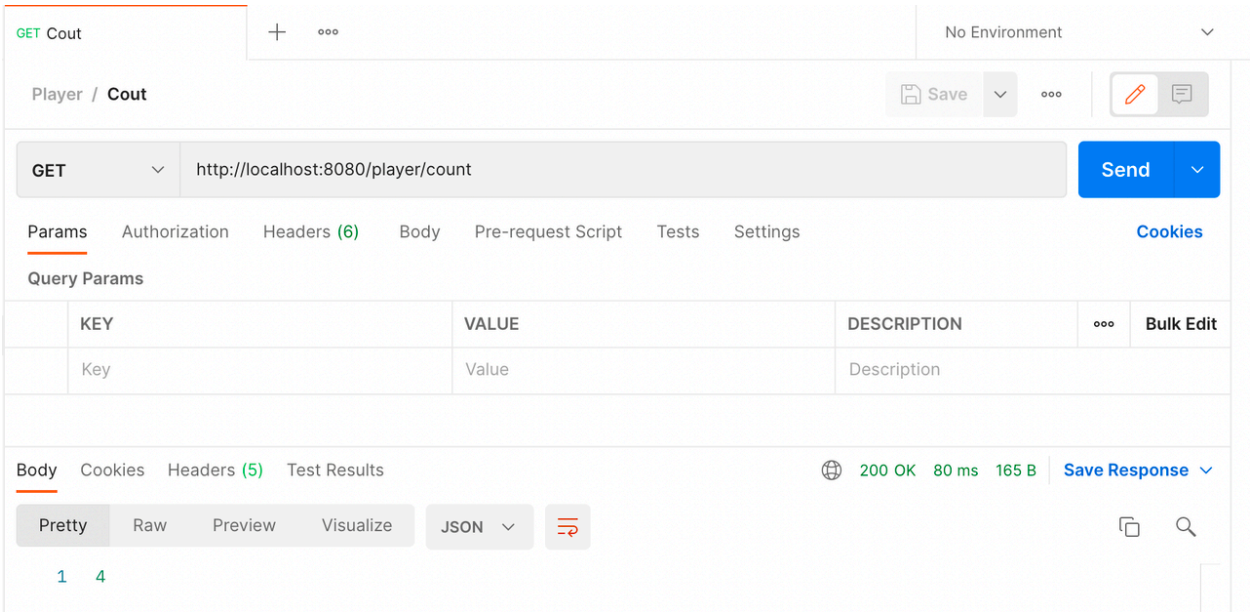


图 6: Postman-Count

- 玩家交易

  点击 Trade 标签，点击 Send 按钮，发送 PUT 形式的 `http://localhost:8000/player/trade` 请求。请求参数为售卖玩家 ID `sellID`、购买玩家 ID `buyID`、购买货物数量 `amount` 以及购买消耗金币数 `price`。返回值为交易是否成功。当出现售卖玩家货物不足、购买玩家金币不足或数据库错误时，交易将不成功。并且由于数据库事务保证，不会有玩家的金币或货物丢失的情况。

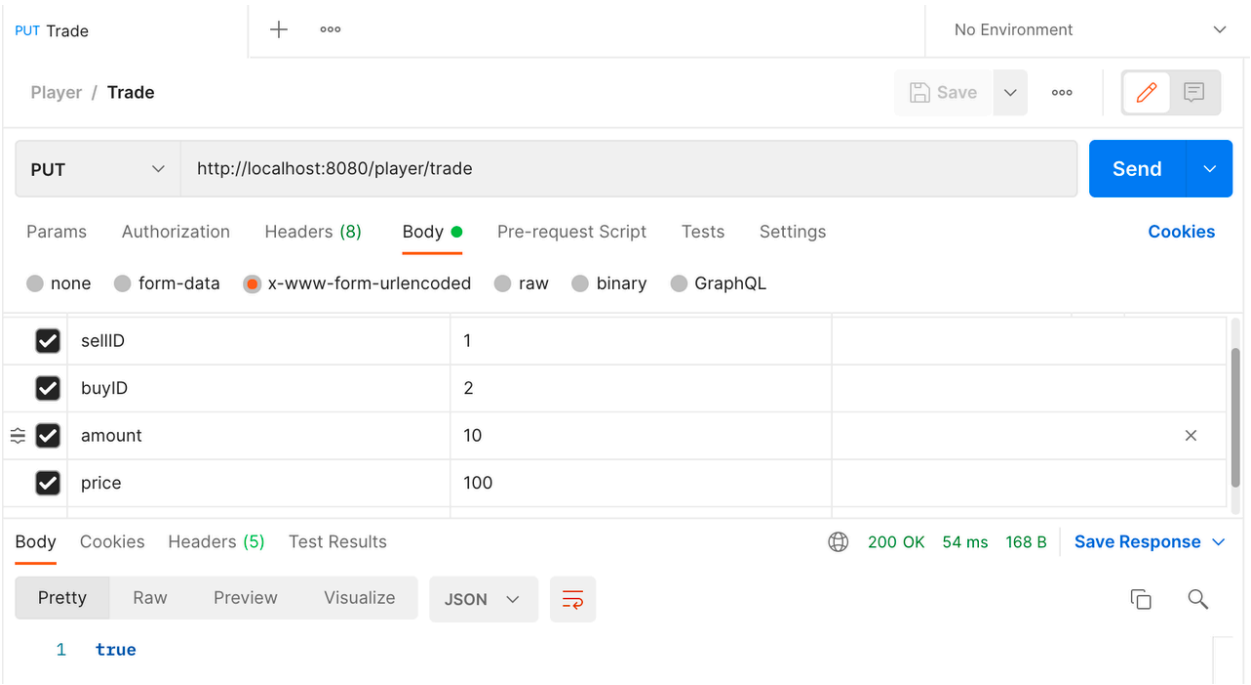


图 7: Postman-Trade

下面使用 curl 请求服务端。

- 增加玩家

  使用 POST 方法向 /player 端点发送请求来增加玩家，例如：

```
curl --location --request POST 'http://localhost:8080/player/' --header 'Content-Type:
    ↪ application/json' --data-raw '[{"coins":100,"goods":20}]'
```

  这里使用 JSON 作为信息的载荷。表示需要创建一个金币数 coins 为 100，货物数 goods 为 20 的玩家。返回值为创建的玩家信息：

```
1
```

- 使用 ID 获取玩家信息

  使用 GET 方法向 /player 端点发送请求来获取玩家信息。此外，还需要在路径上给出玩家的 ID 参数，即 /player/{id}。例如，在请求 ID 为 1 的玩家时：

```
curl --location --request GET 'http://localhost:8080/player/1'
```

  返回值为 ID 为 1 的玩家的信息：

```
{
"coins": 200,
"goods": 10,
"id": 1
}
```

- 使用 Limit 批量获取玩家信息

  使用 GET 方法向 /player/limit 端点发送请求来获取玩家信息。此外，还需要在路径上给出限制查询的玩家信息的总数，即 /player/limit/{limit}。例如，在请求最多 3 个玩家的信息时：

```
curl --location --request GET 'http://localhost:8080/player/limit/3'
```

  返回值为玩家信息的列表：

```
[
{
    "coins": 200,
    "goods": 10,
    "id": 1
},
{
    "coins": 0,
    "goods": 30,
    "id": 2
},
{
```

```
    "coins": 100,
    "goods": 20,
    "id": 3
}
]
```

- 分页获取玩家信息

  使用 GET 方法向 /player/page 端点发送请求来分页获取玩家信息。额外地需要使用 URL 参数，例如在请求页面序号 index 为 0，每页最大请求量 size 为 2 时：

```
curl --location --request GET 'http://localhost:8080/player/page?index=0&size=2'
```

  返回值为 index 为 0 的页，每页有 2 个玩家信息列表。此外，还包含了分页信息，如偏移量、总页数、是否排序等。

```
{
"content": [
    {
    "coins": 200,
    "goods": 10,
    "id": 1
    },
    {
    "coins": 0,
    "goods": 30,
    "id": 2
    }
],
"empty": false,
"first": true,
"last": false,
"number": 0,
"numberOfElements": 2,
"pageable": {
    "offset": 0,
    "pageNumber": 0,
    "pageSize": 2,
    "paged": true,
    "sort": {
    "empty": true,
    "sorted": false,
    "unsorted": true
    },
    "unpaged": false
},
"size": 2,
```

```
"sort": {
    "empty": true,
    "sorted": false,
    "unsorted": true
},
"totalElements": 4,
"totalPages": 2
}
```

- 获取玩家个数

  使用 GET 方法向 /player/count 端点发送请求来获取玩家个数：

```
curl --location --request GET 'http://localhost:8080/player/count'
```

  返回值为玩家个数：

```
4
```

- 玩家交易

  使用 PUT 方法向 /player/trade 端点发送请求来发起玩家间的交易，例如：

```
curl --location --request PUT 'http://localhost:8080/player/trade' \
--header 'Content-Type: application/x-www-form-urlencoded' \
--data-urlencode 'sellID=1' \
--data-urlencode 'buyID=2' \
--data-urlencode 'amount=10' \
--data-urlencode 'price=100'
```

  这里使用 Form Data 作为信息的载荷。表示售卖玩家 ID sellID 为 1、购买玩家 ID buyID 为 2、购买货物数量 amount 为 10、购买消耗金币数 price 为 100。

  返回值为交易是否成功：

```
true
```

  当出现售卖玩家货物不足、购买玩家金币不足或数据库错误时，交易将不成功。并且由于数据库事务保证，不会有玩家的金币或货物丢失的情况。

为方便测试，你可以使用 request.sh 脚本依次发送以下请求：

1. 循环创建 10 名玩家
2. 获取 ID 为 1 的玩家信息
3. 获取至多 3 名玩家信息列表
4. 获取 index 为 0，size 为 2 的一页玩家信息
5. 获取玩家总数
6. ID 为 1 的玩家作为售出方，ID 为 2 的玩家作为购买方，购买 10 个货物，耗费 100 金币

使用 make request 或 ./request.sh 命令运行此脚本，运行结果如下所示：

```
> make request
./request.sh
loop to create 10 players:
1111111111

get player 1:
{"id":1,"coins":200,"goods":10}

get players by limit 3:
[{"id":1,"coins":200,"goods":10},{"id":2,"coins":0,"goods":30},{"id":3,"coins":100,"goods":20}]

get first players:
{"content":[{"id":1,"coins":200,"goods":10},{"id":2,"coins":0,"goods":30}],"pageable":{"sort":{"
    ↪ empty":true,"unsorted":true,"sorted":false},"offset":0,"pageNumber":0,"pageSize":2,"paged
    ↪ ":true,"unpaged":false},"last":false,"totalPages":7,"totalElements":14,"first":true,"size
    ↪ ":2,"number":0,"sort":{"empty":true,"unsorted":true,"sorted":false},"numberOfElements
    ↪ ":2,"empty":false}

get players count:
14

trade by two players:
false
```

4.3.2.7 实现细节

本小节介绍示例应用程序项目中的组件。

4.3.2.7.1 总览

本示例项目的大致目录树如下所示（删除了有碍理解的部分）:

```
.
├── pom.xml
└── src
    └── main
        ├── java
        │   └── com
        │       └── pingcap
        │           ├── App.java
        │           ├── controller
        │           │   └── PlayerController.java
        │           ├── dao
        │           │   ├── PlayerBean.java
        │           │   └── PlayerRepository.java
```

```
        |           └── service
        |               ├── PlayerService.java
        |               └── impl
        |                   └── PlayerServiceImpl.java
        └── resources
            └── application.yml
```

其中：

- pom.xml 内声明了项目的 Maven 配置，如依赖，打包等
- application.yml 内声明了项目的用户配置，如数据库地址、密码、使用的数据库方言等
- App.java 是项目的入口
- controller 是项目对外暴露 HTTP 接口的包
- service 是项目实现接口与逻辑的包
- dao 是项目实现与数据库连接并完成数据持久化的包

4.3.2.7.2 配置

本节将简要介绍 pom.xml 文件中的 Maven 配置，及 application.yml 文件中的用户配置。

Maven 配置

pom.xml 文件为 Maven 配置，在文件内声明了项目的 Maven 依赖，打包方法，打包信息等，你可以通过创建相同依赖空白程序 这一节来复刻此配置文件的生成流程，当然，也可直接复制至你的项目来使用。

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    ↪ instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven
        ↪ -4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.0.1</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

    <groupId>com.pingcap</groupId>
    <artifactId>spring-jpa-hibernate</artifactId>
    <version>0.0.1</version>
    <name>spring-jpa-hibernate</name>
    <description>an example for spring boot, jpa, hibernate and TiDB</description>

    <properties>
        <java.version>17</java.version>
        <maven.compiler.source>17</maven.compiler.source>
```

```
        <maven.compiler.target>17</maven.compiler.target>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <scope>runtime</scope>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

用户配置

`application.yml` 此配置文件声明了用户配置，如数据库地址、密码、使用的数据库方言等。

```
spring:
  datasource:
    url: jdbc:mysql://localhost:4000/test
    username: root
    #    password: xxx
```

```
    driver-class-name: com.mysql.cj.jdbc.Driver
  jpa:
    show-sql: true
    database-platform: org.hibernate.dialect.TiDBDialect
    hibernate:
      ddl-auto: create-drop
```

此配置格式为 YAML 格式。其中：

- `spring.datasource.url`：数据库连接的 URL。
- `spring.datasource.url`：数据库用户名。
- `spring.datasource.password`：数据库密码，此项为空，需注释或删除。
- `spring.datasource.driver-class-name`：数据库驱动，因为 TiDB 与 MySQL 兼容，则此处使用与 mysql-connector-java 适配的驱动类 `com.mysql.cj.jdbc.Driver`。
- `jpa.show-sql`：为 true 时将打印 JPA 运行的 SQL。
- `jpa.database-platform`：选用的数据库方言，此处连接了 TiDB，自然选择 TiDB 方言，注意，此方言在 6.0.0.Beta2 版本后的 Hibernate 中才可选择，请注意依赖版本。
- `jpa.hibernate.ddl-auto`：此处选择的 create-drop 将会在程序开始时创建表，退出时删除表。请勿在正式环境使用，但此处为示例程序，希望尽量不影响数据库数据，因此选择了此选项。

4.3.2.7.3 入口文件

入口文件 `App.java`：

```
package com.pingcap;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.context.ApplicationPidFileWriter;

@SpringBootApplication
public class App {
   public static void main(String[] args) {
      SpringApplication springApplication = new SpringApplication(App.class);
      springApplication.addListeners(new ApplicationPidFileWriter("spring-jpa-hibernate.pid"));
      springApplication.run(args);
   }
}
```

入口类比较简单，首先，有一个 Spring Boot 应用程序的标准配置注解 [@SpringBootApplication](https://docs.spring.io/spring-boot/docs/current/api/org/springframework/boot/autoconfigure/SpringBootApplication.html)。有关详细信息，请参阅 Spring Boot 官方文档中的 [Using the @SpringBootApplication Annotation](https://docs.spring.io/spring-boot/docs/current/reference/html/using-spring-boot.html#using-boot-using-springbootapplication-annotation)。随后，使用 `ApplicationPidFileWriter` 在程序启动过程中，写下一个名为 `spring-jpa-hibernate.pid` 的 PID (process identification number) 文件，可从外部使用此 PID 文件关闭此应用程序。

4.3.2.7.4 数据库持久层

数据库持久层，即 dao 包内，实现了数据对象的持久化。

实体对象

PlayerBean.java 文件为实体对象，这个对象对应了数据库的一张表。

```
package com.pingcap.dao;

import jakarta.persistence.*;

/**
 * it's core entity in hibernate
 * @Table appoint to table name
 */
@Entity
@Table(name = "player_jpa")
public class PlayerBean {
    /**
     * @ID primary key
     * @GeneratedValue generated way. this field will use generator named "player_id"
     * @SequenceGenerator using `sequence` feature to create a generator,
     *    and it named "player_jpa_id_seq" in database, initial form 1 (by `initialValue`
     *    parameter default), and every operator will increase 1 (by `allocationSize`)
     */
    @Id
    @GeneratedValue(generator="player_id")
    @SequenceGenerator(name="player_id", sequenceName="player_jpa_id_seq", allocationSize=1)
    private Long id;

    /**
     * @Column field
     */
    @Column(name = "coins")
    private Integer coins;
    @Column(name = "goods")
    private Integer goods;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public Integer getCoins() {
```

```
        return coins;
    }

    public void setCoins(Integer coins) {
        this.coins = coins;
    }

    public Integer getGoods() {
        return goods;
    }

    public void setGoods(Integer goods) {
        this.goods = goods;
    }
}
```

这里可以看到，实体类中有很多注解，这些注解给了 Hibernate 额外的信息，用以绑定实体类和表：

- `@Entity` 声明 `PlayerBean` 是一个实体类。
- `@Table` 使用注解属性 `name` 将此实体类和表 `player_jpa` 关联。
- `@Id` 声明此属性关联表的主键列。
- `@GeneratedValue` 表示自动生成该列的值，而不应手动设置，使用属性 `generator` 指定生成器的名称为 `player_id`。
- `@SequenceGenerator` 声明一个使用序列的生成器，使用注解属性 `name` 声明生成器的名称为 `player_id`（与 `@GeneratedValue` 中指定的名称需保持一致）。随后使用注解属性 `sequenceName` 指定数据库中序列的名称。最后，使用注解属性 `allocationSize` 声明序列的步长为 1。
- `@Column` 将每个私有属性声明为表 `player_jpa` 的一列，使用注解属性 `name` 确定属性对应的列名。

存储库

为了抽象数据库层，Spring 应用程序使用 Repository 接口，或者 Repository 的子接口。这个接口映射到一个数据库对象，常见的，比如会映射到一个表上。JPA 会实现一些预制的方法，比如INSERT，或使用主键的SELECT 等。

```
package com.pingcap.dao;

import jakarta.persistence.LockModeType;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Lock;
import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.query.Param;
import org.springframework.stereotype.Repository;

import java.util.List;
```

```
@Repository
public interface PlayerRepository extends JpaRepository<PlayerBean, Long> {
    /**
     * use HQL to query by page
     * @param pageable a pageable parameter required by hibernate
     * @return player list package by page message
     */
    @Query(value = "SELECT player_jpa FROM PlayerBean player_jpa")
    Page<PlayerBean> getPlayersByPage(Pageable pageable);

    /**
     * use SQL to query by limit, using named parameter
     * @param limit sql parameter
     * @return player list (max size by limit)
     */
    @Query(value = "SELECT * FROM player_jpa LIMIT :limit", nativeQuery = true)
    List<PlayerBean> getPlayersByLimit(@Param("limit") Integer limit);

    /**
     * query player and add a lock for update
     * @param id player id
     * @return player
     */
    @Lock(value = LockModeType.PESSIMISTIC_WRITE)
    @Query(value = "SELECT player FROM PlayerBean player WHERE player.id = :id")
    // @Query(value = "SELECT * FROM player_jpa WHERE id = :id FOR UPDATE", nativeQuery = true)
    PlayerBean getPlayerAndLock(@Param("id") Long id);
}
```

PlayerRepository 拓展了 Spring 用于 JPA 数据访问所使用的接口 JpaRepository。使用 @Query 注解，告诉 Hibernate 此接口如何实现查询。在此处使用了两种查询语句的语法，其中，在接口 getPlayersByPage 中的查询语句使用的是一种被 Hibernate 称为 HQL (Hibernate Query Language) 的语法。而接口 getPlayersByLimit 中使用的是普通的 SQL，在使用 SQL 语法时，需要将 @Query 的注解参数 nativeQuery 设置为 true。

在 getPlayersByLimit 注解的 SQL 中，:limit 在 Hibernate 中被称为命名参数，Hibernate 将按名称自动寻找并拼接注解所在接口内的参数。你也可以使用 @Param 来指定与参数不同的名称用于注入。

在 getPlayerAndLock 中,使用了一个注解 [@Lock](https://docs.spring.io/spring-data/jpa/docs/current/api/org/springframework/data/jpa/repos 此注解声明此处使用悲观锁进行锁定，如需了解更多其他锁定方式，可查看实体锁定文档。此处的 @Lock 仅可与 HQL 搭配使用，否则将会产生错误。当然，如果你希望直接使用 SQL 进行锁定，可直接使用注释部分的注解：

```
@Query(value = "SELECT * FROM player_jpa WHERE id = :id FOR UPDATE", nativeQuery = true)
```

直接使用 SQL 的 FOR UPDATE 来增加锁。你也可通过 TiDB SELECT 文档 进行更深层次的原理学习。

4.3.2.7.5 逻辑实现

逻辑实现层，即 service 包，内含了项目实现的接口与逻辑

接口

PlayerService.java 文件内定义了逻辑接口，实现接口，而不是直接编写一个类的原因，是尽量使例子贴近实际使用，体现设计的开闭原则。你也可以省略掉此接口，在依赖类中直接注入实现类，但并不推荐这样做。

```
package com.pingcap.service;

import com.pingcap.dao.PlayerBean;
import org.springframework.data.domain.Page;

import java.util.List;

public interface PlayerService {
    /**
     * create players by passing in a List of PlayerBean
     *
     * @param players will create players list
     * @return The number of create accounts
     */
    Integer createPlayers(List<PlayerBean> players);

    /**
     * buy goods and transfer funds between one player and another in one transaction
     * @param sellId sell player id
     * @param buyId buy player id
     * @param amount goods amount, if sell player has not enough goods, the trade will break
     * @param price price should pay, if buy player has not enough coins, the trade will break
     */
    void buyGoods(Long sellId, Long buyId, Integer amount, Integer price) throws RuntimeException
        ↪ ;

    /**
     * get the player info by id.
     *
     * @param id player id
     * @return the player of this id
     */
    PlayerBean getPlayerByID(Long id);

    /**
     * get a subset of players from the data store by limit.
     *
     * @param limit return max size
     * @return player list
     */
```

```
    List<PlayerBean> getPlayers(Integer limit);

    /**
     * get a page of players from the data store.
     *
     * @param index page index
     * @param size page size
     * @return player list
     */
    Page<PlayerBean> getPlayersByPage(Integer index, Integer size);

    /**
     * count players from the data store.
     *
     * @return all players count
     */
    Long countPlayers();
}
```

实现（重要）

`PlayerService.java` 文件内实现了 `PlayerService` 接口，所有数据操作逻辑都编写在这里。

```
package com.pingcap.service.impl;

import com.pingcap.dao.PlayerBean;
import com.pingcap.dao.PlayerRepository;
import com.pingcap.service.PlayerService;
import jakarta.transaction.Transactional;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
import org.springframework.stereotype.Service;

import java.util.List;

/**
 * PlayerServiceImpl implements PlayerService interface
 * @Transactional it means every method in this class, will package by a pair of
 *     transaction.begin() and transaction.commit(). and it will be call
 *     transaction.rollback() when method throw an exception
 */
@Service
@Transactional
public class PlayerServiceImpl implements PlayerService {
    @Autowired
```

```
    private PlayerRepository playerRepository;

    @Override
    public Integer createPlayers(List<PlayerBean> players) {
        return playerRepository.saveAll(players).size();
    }

    @Override
    public void buyGoods(Long sellId, Long buyId, Integer amount, Integer price) throws
        ↪ RuntimeException {
        PlayerBean buyPlayer = playerRepository.getPlayerAndLock(buyId);
        PlayerBean sellPlayer = playerRepository.getPlayerAndLock(sellId);
        if (buyPlayer == null || sellPlayer == null) {
            throw new RuntimeException("sell or buy player not exist");
        }

        if (buyPlayer.getCoins() < price || sellPlayer.getGoods() < amount) {
            throw new RuntimeException("coins or goods not enough, rollback");
        }

        buyPlayer.setGoods(buyPlayer.getGoods() + amount);
        buyPlayer.setCoins(buyPlayer.getCoins() - price);
        playerRepository.save(buyPlayer);

        sellPlayer.setGoods(sellPlayer.getGoods() - amount);
        sellPlayer.setCoins(sellPlayer.getCoins() + price);
        playerRepository.save(sellPlayer);
    }

    @Override
    public PlayerBean getPlayerByID(Long id) {
        return playerRepository.findById(id).orElse(null);
    }

    @Override
    public List<PlayerBean> getPlayers(Integer limit) {
        return playerRepository.getPlayersByLimit(limit);
    }

    @Override
    public Page<PlayerBean> getPlayersByPage(Integer index, Integer size) {
        return playerRepository.getPlayersByPage(PageRequest.of(index, size));
    }

    @Override
```

```
    public Long countPlayers() {
        return playerRepository.count();
    }
}
```

这里使用了 @Service 这个注解，声明此对象的生命周期交由 Spring 管理。

注意，除了有 @Service 注解之外，PlayerServiceImpl 实现类还有一个 [@Transactional](https://docs.spring.io/spring-framework/docs/current/reference/html/data-access.html#transaction-declarative-annotations) 注解。当在应用程序中启用事务管理时 (可使用 [@EnableTransactionManagement](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/transaction/annotation/EnableTransactionManagement.html) 打开，但 Spring Boot 默认开启，无需再次手动配置)，Spring 会自动将所有带有 @Transactional 注释的对象包装在一个代理中，使用该代理对对象的调用进行处理。

你可以简单的认为，代理在带有 @Transactional 注释的对象内的函数调用时：在函数顶部将使用 transaction.begin() 开启事务，函数返回后，调用 transaction.commit() 进行事务提交，而出现任何运行时错误时，代理将会调用 transaction.rollback() 来回滚。

你可参阅数据库事务来获取更多有关事务的信息，或者阅读 Spring 官网中的文章理解 Spring 框架的声明式事务实现。

整个实现类中，buyGoods 函数需重点关注，其在不符合逻辑时将抛出异常，引导 Hibernate 进行事务回滚，防止出现错误数据。

4.3.2.7.6 外部接口

controller 包对外暴露 HTTP 接口，可以通过 REST API 来访问服务。

```
package com.pingcap.controller;

import com.pingcap.dao.PlayerBean;
import com.pingcap.service.PlayerService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Page;
import org.springframework.lang.NonNull;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/player")
public class PlayerController {
    @Autowired
    private PlayerService playerService;

    @PostMapping
    public Integer createPlayer(@RequestBody @NonNull List<PlayerBean> playerList) {
        return playerService.createPlayers(playerList);
```

```
    }

    @GetMapping("/{id}")
    public PlayerBean getPlayerByID(@PathVariable Long id) {
        return playerService.getPlayerByID(id);
    }

    @GetMapping("/limit/{limit_size}")
    public List<PlayerBean> getPlayerByLimit(@PathVariable("limit_size") Integer limit) {
        return playerService.getPlayers(limit);
    }

    @GetMapping("/page")
    public Page<PlayerBean> getPlayerByPage(@RequestParam Integer index, @RequestParam("size")
        ↪ Integer size) {
        return playerService.getPlayersByPage(index, size);
    }

    @GetMapping("/count")
    public Long getPlayersCount() {
        return playerService.countPlayers();
    }

    @PutMapping("/trade")
    public Boolean trade(@RequestParam Long sellID, @RequestParam Long buyID, @RequestParam
        ↪ Integer amount, @RequestParam Integer price) {
        try {
            playerService.buyGoods(sellID, buyID, amount, price);
        } catch (RuntimeException e) {
            return false;
        }

        return true;
    }
}
```

`PlayerController` 中使用了尽可能多的注解方式来作为示例展示功能，在实际项目中，请尽量保持风格的统一，同时遵循你公司或团体的规则。`PlayerController` 有许多注解，下方将进行逐一解释：

- [@RestController](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/RestController
  将 `PlayerController` 声明为一个 Web Controller，且将返回值序列化为 JSON 输出。
- [@RequestMapping](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/RequestMa
  映射 URL 端点为 `/player`，即此 `Web Controller` 仅监听 `/player` URL 下的请求。
- `@Autowired` 用于 Spring 的自动装配，可以看到，此处声明需要一个 `PlayerService` 对象，此对象为接口，并未指定使用哪一个实现类，这是由 Spring 自动装配的，有关此装配规则，可查看 Spirng 官网中的 The

IoC container 一文。

- [@PostMapping](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/PostMapping.h
  声明此函数将响应 HTTP 中的 POST 类型请求。
  - `@RequestBody` 声明此处将 HTTP 的整个载荷解析到参数 `playerList` 中。
  - `@NonNull` 声明参数不可为空，否则将校验并返回错误。
- [@GetMapping](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/GetMapping.htm
  声明此函数将响应 HTTP 中的 GET 类型请求。
  - [@PathVariable](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/PathVaria
    可以看到注解中有形如 `{id}`、`{limit_size}` 这样的占位符，这种占位符将被绑定到 `@PathVariable` 注释的变量中，绑定的依据是注解中的注解属性 `name`（变量名可省略，即 `@PathVariable(name ="limit_size")` 可写成 `@PathVariable("limit_size")`），不特殊指定时，与变量名名称相同。
- [@PutMapping](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/PutMapping.htm
  声明此函数将响应 HTTP 中的 PUT 类型请求。
- [@RequestParam](https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/RequestParam
  此声明将解析请求中的 URL 参数、表单参数等参数，绑定至注解的变量中。

4.3.2.8 创建相同依赖空白程序（可选）

本程序使用 Spring Initializr 构建。你可以在这个网页上通过点选以下选项并更改少量配置，来快速得到一个与本示例程序相同依赖的空白应用程序，配置项如下：

Project

- Maven Project

Language

- Java

Spring Boot

- 最新稳定版本

Project Metadata

- Group: com.pingcap
- Artifact: spring-jpa-hibernate
- Name: spring-jpa-hibernate
- Package name: com.pingcap
- Packaging: Jar
- Java: 17

Dependencies

- Spring Web
- Spring Data JPA
- MySQL Driver

> **注意：**
>
> 尽管 SQL 相对标准化，但每个数据库供应商都使用 ANSI SQL 定义语法的子集和超集。这被称为数据库的方言。Hibernate 通过其 org.hibernate.dialect.Dialect 类和每个数据库供应商的各种子类来处理这些方言的变化。
>
> 在大多数情况下，Hibernate 将能够通过在启动期间通过 JDBC 连接的一些返回值来确定要使用的正确方言。有关 Hibernate 确定要使用的正确方言的能力（以及你影响该解析的能力）的信息，请参阅方言解析。
>
> 如果由于某种原因无法确定正确的方言，或者你想使用自定义方言，则需要设置 hibernate.dialect 配置项。
>
> —— 节选自 Hibernate 官方文档：Database Dialect

随后，即可获取一个拥有与示例程序相同依赖的空白 Spring Boot 应用程序。

4.3.3 TiDB 和 Java 的简单 CRUD 应用程序

本文档将展示如何使用 TiDB 和 Java 来构造一个简单的 CRUD 应用程序。

> **注意：**
>
> 推荐使用 Java 8 及以上版本进行 TiDB 的应用程序的编写。

4.3.3.1 拓展学习视频

- 使用 Connector/J - TiDB v6
- 在 TiDB 上开发应用的最佳实践 - TiDB v6

> **建议：**
>
> 如果你希望使用 Spring Boot 进行 TiDB 应用程序的编写，可以查看Build the TiDB Application using Spring Boot。

4.3.3.2 第 1 步：启动你的 TiDB 集群

本节将介绍 TiDB 集群的启动方法。

创建 TiDB Serverless 集群。

你可以部署一个本地测试的 TiDB 集群或正式的 TiDB 集群。详细步骤，请参考：

- 部署本地测试 TiDB 集群
- 部署正式 TiDB 集群

基于 Git 的预配置的开发环境：现在就试试

该环境会自动克隆代码，并通过 TiUP 部署测试集群。

4.3.3.3 第 2 步：获取代码

```
git clone https://github.com/pingcap-inc/tidb-example-java.git
```

Mybatis 是当前比较流行的开源 Java 应用持久层框架，本文将以 Maven 插件的方式使用 MyBatis Generator 生成部分持久层代码。

进入目录 `plain-java-mybatis`：

```
cd plain-java-mybatis
```

目录结构如下所示：

```
.
├── Makefile
├── pom.xml
└── src
    └── main
        ├── java
        │   └── com
        │       └── pingcap
        │           ├── MybatisExample.java
        │           ├── dao
        │           │   └── PlayerDAO.java
        │           └── model
        │               ├── Player.java
        │               ├── PlayerMapper.java
        │               └── PlayerMapperEx.java
        └── resources
            ├── dbinit.sql
            ├── log4j.properties
            ├── mapper
            │   ├── PlayerMapper.xml
            │   └── PlayerMapperEx.xml
```

```
            ├── mybatis-config.xml
            └── mybatis-generator.xml
```

其中，自动生成的文件有：

- `src/main/java/com/pingcap/model/Player.java`：Player 实体类文件
- `src/main/java/com/pingcap/model/PlayerMapper.java`：Player Mapper 的接口文件
- `src/main/resources/mapper/PlayerMapper.xml`：Player Mapper 的 XML 映射，它是 Mybatis 用于生成 Player Mapper 接口的实现类的配置

这些文件的生成策略被写在了 `mybatis-generator.xml` 配置文件内，它是 Mybatis Generator 的配置文件，下面配置文件中添加了使用方法的说明：

```
<!DOCTYPE generatorConfiguration PUBLIC
 "-//mybatis.org//DTD MyBatis Generator Configuration 1.0//EN"
 "http://mybatis.org/dtd/mybatis-generator-config_1_0.dtd">

<generatorConfiguration>
    <!--
        <context/> entire document: https://mybatis.org/generator/configreference/context.html

        context.id: A unique identifier you like
        context.targetRuntime: Used to specify the runtime target for generated code.
            It has MyBatis3DynamicSql / MyBatis3Kotlin / MyBatis3 / MyBatis3Simple 4 selection to
                ↪  choice.
    -->
    <context id="simple" targetRuntime="MyBatis3">
        <!--
            <commentGenerator/> entire document: https://mybatis.org/generator/configreference/
                ↪ commentGenerator.html

            commentGenerator:
                - property(suppressDate): remove timestamp in comments
                - property(suppressAllComments): remove all comments
        -->
        <commentGenerator>
            <property name="suppressDate" value="true"/>
            <property name="suppressAllComments" value="true" />
        </commentGenerator>

        <!--
            <jdbcConnection/> entire document: https://mybatis.org/generator/configreference/
                ↪ jdbcConnection.html

            jdbcConnection.driverClass: The fully qualified class name for the JDBC driver used
                ↪ to access the database.
```

```
        Used mysql-connector-java:5.1.49, should specify JDBC is com.mysql.jdbc.Driver
    jdbcConnection.connectionURL: The JDBC connection URL used to access the database.
-->
<jdbcConnection driverClass="com.mysql.jdbc.Driver"
    connectionURL="jdbc:mysql://localhost:4000/test?user=root" />

<!--
    <javaModelGenerator/> entire document: https://mybatis.org/generator/configreference/
        ↪ javaModelGenerator.html
    Model code file will be generated at ${targetProject}/${targetPackage}

    javaModelGenerator:
        - property(constructorBased): If it's true, generator will create constructor
            ↪ function in model
-->
<javaModelGenerator targetPackage="com.pingcap.model" targetProject="src/main/java">
    <property name="constructorBased" value="true"/>
</javaModelGenerator>

<!--
    <sqlMapGenerator/> entire document: https://mybatis.org/generator/configreference/
        ↪ sqlMapGenerator.html
    XML SQL mapper file will be generated at ${targetProject}/${targetPackage}
-->
<sqlMapGenerator targetPackage="." targetProject="src/main/resources/mapper"/>

<!--
    <javaClientGenerator/> entire document: https://mybatis.org/generator/configreference
        ↪ /javaClientGenerator.html
    Java code mapper interface file will be generated at ${targetProject}/${targetPackage
        ↪ }

    javaClientGenerator.type (context.targetRuntime is MyBatis3):
        This attribute indicated Mybatis how to implement interface.
        It has ANNOTATEDMAPPER / MIXEDMAPPER / XMLMAPPER 3 selection to choice.
-->
<javaClientGenerator type="XMLMAPPER" targetPackage="com.pingcap.model" targetProject="
    ↪ src/main/java"/>

<!--
    <table/> entire document: https://mybatis.org/generator/configreference/table.html

    table.tableName: The name of the database table.
    table.domainObjectName: The base name from which generated object names will be
        ↪ generated. If not specified, MBG will generate a name automatically based on
```

```
                ↪ the tableName.
            table.enableCountByExample: Signifies whether a count by example statement should be
                ↪ generated.
            table.enableUpdateByExample: Signifies whether an update by example statement should
                ↪ be generated.
            table.enableDeleteByExample: Signifies whether a delete by example statement should
                ↪ be generated.
            table.enableSelectByExample: Signifies whether a select by example statement should
                ↪ be generated.
            table.selectByExampleQueryId: This value will be added to the select list of the
                ↪ select by example statement in this form: "'<value>' as QUERYID".
        -->
        <table tableName="player" domainObjectName="Player"
               enableCountByExample="false" enableUpdateByExample="false"
               enableDeleteByExample="false" enableSelectByExample="false"
               selectByExampleQueryId="false"/>
    </context>
</generatorConfiguration>
```

mybatis-generator.xml 在 pom.xml 中，以 mybatis-generator-maven-plugin 插件配置的方式被引入：

```
<plugin>
    <groupId>org.mybatis.generator</groupId>
    <artifactId>mybatis-generator-maven-plugin</artifactId>
    <version>1.4.1</version>
    <configuration>
        <configurationFile>src/main/resources/mybatis-generator.xml</configurationFile>
        <verbose>true</verbose>
        <overwrite>true</overwrite>
    </configuration>

    <dependencies>
        <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
        <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.49</version>
        </dependency>
    </dependencies>
</plugin>
```

在 Maven 插件内引入后，可删除旧的生成文件后，通过命令 mvn mybatis-generate 生成新的文件。或者你也可以使用已经编写好的 make 命令，通过 make gen 来同时删除旧文件，并生成新文件。

> **注意：**
>
> `mybatis-generator.xml` 中的属性 `configuration.overwrite` 仅可控制新生成的 Java 代码文件使用覆盖方式被写入，但 XML 映射文件仍会以追加方式写入。因此，推荐在 Mybaits Generator 生成新的文件前，先删除掉旧的文件。

`Player.java` 是使用 Mybatis Generator 生成出的数据实体类文件，为数据库表在程序内的映射。`Player` 类的每个属性都对应着 `player` 表的一个字段。

```
package com.pingcap.model;

public class Player {
    private String id;

    private Integer coins;

    private Integer goods;

    public Player(String id, Integer coins, Integer goods) {
        this.id = id;
        this.coins = coins;
        this.goods = goods;
    }

    public Player() {
        super();
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public Integer getCoins() {
        return coins;
    }

    public void setCoins(Integer coins) {
        this.coins = coins;
    }
```

```
    public Integer getGoods() {
        return goods;
    }

    public void setGoods(Integer goods) {
        this.goods = goods;
    }
}
```

PlayerMapper.java 是使用 Mybatis Generator 生成出的映射接口文件，它仅规定了接口，接口的实现类是由 Mybatis 来通过 XML 或注解自动生成的：

```
package com.pingcap.model;

import com.pingcap.model.Player;

public interface PlayerMapper {
    int deleteByPrimaryKey(String id);

    int insert(Player row);

    int insertSelective(Player row);

    Player selectByPrimaryKey(String id);

    int updateByPrimaryKeySelective(Player row);

    int updateByPrimaryKey(Player row);
}
```

PlayerMapper.xml 是使用 Mybatis Generator 生成出的映射 XML 文件，Mybatis 将使用这个文件自动生成 PlayerMapper 接口的实现类：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-
    ↪ mapper.dtd">
<mapper namespace="com.pingcap.model.PlayerMapper">
  <resultMap id="BaseResultMap" type="com.pingcap.model.Player">
    <constructor>
      <idArg column="id" javaType="java.lang.String" jdbcType="VARCHAR" />
      <arg column="coins" javaType="java.lang.Integer" jdbcType="INTEGER" />
      <arg column="goods" javaType="java.lang.Integer" jdbcType="INTEGER" />
    </constructor>
  </resultMap>
  <sql id="Base_Column_List">
```

```
  id, coins, goods
</sql>
<select id="selectByPrimaryKey" parameterType="java.lang.String" resultMap="BaseResultMap">
  select
  <include refid="Base_Column_List" />
  from player
  where id = #{id,jdbcType=VARCHAR}
</select>
<delete id="deleteByPrimaryKey" parameterType="java.lang.String">
  delete from player
  where id = #{id,jdbcType=VARCHAR}
</delete>
<insert id="insert" parameterType="com.pingcap.model.Player">
  insert into player (id, coins, goods
    )
  values (#{id,jdbcType=VARCHAR}, #{coins,jdbcType=INTEGER}, #{goods,jdbcType=INTEGER}
    )
</insert>
<insert id="insertSelective" parameterType="com.pingcap.model.Player">
  insert into player
  <trim prefix="(" suffix=")" suffixOverrides=",">
    <if test="id != null">
      id,
    </if>
    <if test="coins != null">
      coins,
    </if>
    <if test="goods != null">
      goods,
    </if>
  </trim>
  <trim prefix="values (" suffix=")" suffixOverrides=",">
    <if test="id != null">
      #{id,jdbcType=VARCHAR},
    </if>
    <if test="coins != null">
      #{coins,jdbcType=INTEGER},
    </if>
    <if test="goods != null">
      #{goods,jdbcType=INTEGER},
    </if>
  </trim>
</insert>
<update id="updateByPrimaryKeySelective" parameterType="com.pingcap.model.Player">
  update player
```

```
    <set>
      <if test="coins != null">
        coins = #{coins,jdbcType=INTEGER},
      </if>
      <if test="goods != null">
        goods = #{goods,jdbcType=INTEGER},
      </if>
    </set>
    where id = #{id,jdbcType=VARCHAR}
  </update>
  <update id="updateByPrimaryKey" parameterType="com.pingcap.model.Player">
    update player
    set coins = #{coins,jdbcType=INTEGER},
      goods = #{goods,jdbcType=INTEGER}
    where id = #{id,jdbcType=VARCHAR}
  </update>
</mapper>
```

由于 Mybatis Generator 需要逆向生成源码，因此，数据库中需先行有此表结构，可使用 `dbinit.sql` 生成表结构：

```
USE test;
DROP TABLE IF EXISTS player;

CREATE TABLE player (
    `id` VARCHAR(36),
    `coins` INTEGER,
    `goods` INTEGER,
    PRIMARY KEY (`id`)
);
```

额外拆分接口 `PlayerMapperEx` 继承 `PlayerMapper`，并且编写与之匹配的 `PlayerMapperEx.xml`。避免直接更改 `PlayerMapper.java` 和 `PlayerMapper.xml`。这是为了规避 Mybatis Generator 的反复生成，影响到自行编写的代码。

在 `PlayerMapperEx.java` 中定义自行增加的接口：

```
package com.pingcap.model;

import java.util.List;

public interface PlayerMapperEx extends PlayerMapper {
    Player selectByPrimaryKeyWithLock(String id);

    List<Player> selectByLimit(Integer limit);

    Integer count();
```

```
}
```

在 PlayerMapperEx.xml 中定义映射规则：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "http://mybatis.org/dtd/mybatis-3-
    ↪ mapper.dtd">
<mapper namespace="com.pingcap.model.PlayerMapperEx">
  <resultMap id="BaseResultMap" type="com.pingcap.model.Player">
    <constructor>
      <idArg column="id" javaType="java.lang.String" jdbcType="VARCHAR" />
      <arg column="coins" javaType="java.lang.Integer" jdbcType="INTEGER" />
      <arg column="goods" javaType="java.lang.Integer" jdbcType="INTEGER" />
    </constructor>
  </resultMap>
  <sql id="Base_Column_List">
    id, coins, goods
  </sql>

  <select id="selectByPrimaryKeyWithLock" parameterType="java.lang.String" resultMap="
      ↪ BaseResultMap">
    select
    <include refid="Base_Column_List" />
    from player
    where `id` = #{id,jdbcType=VARCHAR}
    for update
  </select>

  <select id="selectByLimit" parameterType="java.lang.Integer" resultMap="BaseResultMap">
    select
    <include refid="Base_Column_List" />
    from player
    limit #{id,jdbcType=INTEGER}
  </select>

  <select id="count" resultType="java.lang.Integer">
    select count(*) from player
  </select>

</mapper>
```

PlayerDAO.java 是程序用来管理数据对象的类。其中 DAO 是 Data Access Object 的缩写。在其中定义了一系列数据的操作方法，用于数据的写入。

```
package com.pingcap.dao;
```

```
import com.pingcap.model.Player;
import com.pingcap.model.PlayerMapperEx;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;

import java.util.List;
import java.util.function.Function;

public class PlayerDAO {
    public static class NotEnoughException extends RuntimeException {
        public NotEnoughException(String message) {
            super(message);
        }
    }

    // Run SQL code in a way that automatically handles the
    // transaction retry logic, so we don't have to duplicate it in
    // various places.
    public Object runTransaction(SqlSessionFactory sessionFactory, Function<PlayerMapperEx,
        ↪ Object> fn) {
        Object resultObject = null;
        SqlSession session = null;

        try {
            // open a session with autoCommit is false
            session = sessionFactory.openSession(false);

            // get player mapper
            PlayerMapperEx playerMapperEx = session.getMapper(PlayerMapperEx.class);

            resultObject = fn.apply(playerMapperEx);
            session.commit();
            System.out.println("APP: COMMIT;");
        } catch (Exception e) {
            if (e instanceof NotEnoughException) {
                System.out.printf("APP: ROLLBACK BY LOGIC; \n%s\n", e.getMessage());
            } else {
                System.out.printf("APP: ROLLBACK BY ERROR; \n%s\n", e.getMessage());
            }

            if (session != null) {
                session.rollback();
            }
        } finally {
            if (session != null) {
```

```
            session.close();
        }
    }

    return resultObject;
}

public Function<PlayerMapperEx, Object> createPlayers(List<Player> players) {
    return playerMapperEx -> {
        Integer addedPlayerAmount = 0;
        for (Player player: players) {
            playerMapperEx.insert(player);
            addedPlayerAmount ++;
        }
        System.out.printf("APP: createPlayers() --> %d\n", addedPlayerAmount);
        return addedPlayerAmount;
    };
}

public Function<PlayerMapperEx, Object> buyGoods(String sellId, String buyId, Integer amount,
    ↪  Integer price) {
    return playerMapperEx -> {
        Player sellPlayer = playerMapperEx.selectByPrimaryKeyWithLock(sellId);
        Player buyPlayer = playerMapperEx.selectByPrimaryKeyWithLock(buyId);

        if (buyPlayer == null || sellPlayer == null) {
            throw new NotEnoughException("sell or buy player not exist");
        }

        if (buyPlayer.getCoins() < price || sellPlayer.getGoods() < amount) {
            throw new NotEnoughException("coins or goods not enough, rollback");
        }

        int affectRows = 0;
        buyPlayer.setGoods(buyPlayer.getGoods() + amount);
        buyPlayer.setCoins(buyPlayer.getCoins() - price);
        affectRows += playerMapperEx.updateByPrimaryKey(buyPlayer);

        sellPlayer.setGoods(sellPlayer.getGoods() - amount);
        sellPlayer.setCoins(sellPlayer.getCoins() + price);
        affectRows += playerMapperEx.updateByPrimaryKey(sellPlayer);

        System.out.printf("APP: buyGoods --> sell: %s, buy: %s, amount: %d, price: %d\n",
            ↪ sellId, buyId, amount, price);
        return affectRows;
```

```
        };
    }

    public Function<PlayerMapperEx, Object> getPlayerByID(String id) {
        return playerMapperEx -> playerMapperEx.selectByPrimaryKey(id);
    }

    public Function<PlayerMapperEx, Object> printPlayers(Integer limit) {
        return playerMapperEx -> {
            List<Player> players = playerMapperEx.selectByLimit(limit);

            for (Player player: players) {
                System.out.println("\n[printPlayers]:\n" + player);
            }
            return 0;
        };
    }

    public Function<PlayerMapperEx, Object> countPlayers() {
        return PlayerMapperEx::count;
    }
}
```

MybatisExample 是 plain-java-mybatis 这个示例程序的主类。其中定义了入口函数：

```
package com.pingcap;

import com.pingcap.dao.PlayerDAO;
import com.pingcap.model.Player;
import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;

import java.io.IOException;
import java.io.InputStream;
import java.util.Arrays;
import java.util.Collections;

public class MybatisExample {
    public static void main( String[] args ) throws IOException {
        // 1. Create a SqlSessionFactory based on our mybatis-config.xml configuration
        // file, which defines how to connect to the database.
        InputStream inputStream = Resources.getResourceAsStream("mybatis-config.xml");
        SqlSessionFactory sessionFactory = new SqlSessionFactoryBuilder().build(inputStream);

        // 2. And then, create DAO to manager your data
```

```
PlayerDAO playerDAO = new PlayerDAO();

// 3. Run some simple examples.

// Create a player who has 1 coin and 1 goods.
playerDAO.runTransaction(sessionFactory, playerDAO.createPlayers(
        Collections.singletonList(new Player("test", 1, 1))));

// Get a player.
Player testPlayer = (Player)playerDAO.runTransaction(sessionFactory, playerDAO.
    ↪ getPlayerByID("test"));
System.out.printf("PlayerDAO.getPlayer:\n    => id: %s\n    => coins: %s\n    => goods: %
    ↪ s\n",
        testPlayer.getId(), testPlayer.getCoins(), testPlayer.getGoods());

// Count players amount.
Integer count = (Integer)playerDAO.runTransaction(sessionFactory, playerDAO.countPlayers
    ↪ ());
System.out.printf("PlayerDAO.countPlayers:\n    => %d total players\n", count);

// Print 3 players.
playerDAO.runTransaction(sessionFactory, playerDAO.printPlayers(3));

// 4. Getting further.

// Player 1: id is "1", has only 100 coins.
// Player 2: id is "2", has 114514 coins, and 20 goods.
Player player1 = new Player("1", 100, 0);
Player player2 = new Player("2", 114514, 20);

// Create two players "by hand", using the INSERT statement on the backend.
int addedCount = (Integer)playerDAO.runTransaction(sessionFactory,
        playerDAO.createPlayers(Arrays.asList(player1, player2)));
System.out.printf("PlayerDAO.createPlayers:\n    => %d total inserted players\n",
    ↪ addedCount);

// Player 1 wants to buy 10 goods from player 2.
// It will cost 500 coins, but player 1 cannot afford it.
System.out.println("\nPlayerDAO.buyGoods:\n    => this trade will fail");
Integer updatedCount = (Integer)playerDAO.runTransaction(sessionFactory,
        playerDAO.buyGoods(player2.getId(), player1.getId(), 10, 500));
System.out.printf("PlayerDAO.buyGoods:\n    => %d total update players\n", updatedCount);

// So player 1 has to reduce the incoming quantity to two.
System.out.println("\nPlayerDAO.buyGoods:\n    => this trade will success");
```

```
        updatedCount = (Integer)playerDAO.runTransaction(sessionFactory,
                playerDAO.buyGoods(player2.getId(), player1.getId(), 2, 100));
        System.out.printf("PlayerDAO.buyGoods:\n    => %d total update players\n", updatedCount);
    }
}
```

当前开源比较流行的 Java ORM 为 Hibernate，且 Hibernate 在版本 6.0.0.Beta2 及以后支持了 TiDB 方言。完美适配了 TiDB 的特性。因此，此处将以 6.0.0.Beta2 + 版本进行说明。

进入目录 `plain-java-hibernate`：

```
cd plain-java-hibernate
```

目录结构如下所示：

```
.
├── Makefile
├── plain-java-hibernate.iml
├── pom.xml
└── src
    └── main
        ├── java
        │   └── com
        │       └── pingcap
        │           └── HibernateExample.java
        └── resources
            └── hibernate.cfg.xml
```

其中，`hibernate.cfg.xml` 为 Hibernate 配置文件，定义了：

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>

        <!-- Database connection settings -->
        <property name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property>
        <property name="hibernate.dialect">org.hibernate.dialect.TiDBDialect</property>
        <property name="hibernate.connection.url">jdbc:mysql://localhost:4000/test</property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.connection.password"></property>
        <property name="hibernate.connection.autocommit">false</property>

        <!-- Required so a table can be created from the 'PlayerDAO' class -->
        <property name="hibernate.hbm2ddl.auto">create-drop</property>
```

```
        <!-- Optional: Show SQL output for debugging -->
        <property name="hibernate.show_sql">true</property>
        <property name="hibernate.format_sql">true</property>
    </session-factory>
</hibernate-configuration>
```

HibernateExample.java 是 plain-java-hibernate 这个示例程序的主体。使用 Hibernate 时，相较于 JDBC，这里仅需写入配置文件地址，Hibernate 屏蔽了创建数据库连接时，不同数据库差异的细节。

PlayerDAO 是程序用来管理数据对象的类。其中 DAO 是 Data Access Object 的缩写。其中定义了一系列数据的操作方法，用来提供数据的写入能力。相较于 JDBC，Hibernate 封装了大量的操作，如对象映射、基本对象的 CRUD 等，极大的简化了代码量。

PlayerBean 是数据实体类，为数据库表在程序内的映射。PlayerBean 的每个属性都对应着 player 表的一个字段。相较于 JDBC，Hibernate 的 PlayerBean 实体类为了给 Hibernate 提供更多的信息，加入了注解，用来指示映射关系。

```
package com.pingcap;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
import org.hibernate.JDBCException;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
import org.hibernate.query.NativeQuery;
import org.hibernate.query.Query;

import java.util.Arrays;
import java.util.Collections;
import java.util.List;
import java.util.function.Function;

@Entity
@Table(name = "player_hibernate")
class PlayerBean {
    @Id
    private String id;
    @Column(name = "coins")
    private Integer coins;
    @Column(name = "goods")
    private Integer goods;
```

```
    public PlayerBean() {
    }

    public PlayerBean(String id, Integer coins, Integer goods) {
        this.id = id;
        this.coins = coins;
        this.goods = goods;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public Integer getCoins() {
        return coins;
    }

    public void setCoins(Integer coins) {
        this.coins = coins;
    }

    public Integer getGoods() {
        return goods;
    }

    public void setGoods(Integer goods) {
        this.goods = goods;
    }

    @Override
    public String toString() {
        return String.format("    %-8s => %10s\n    %-8s => %10s\n    %-8s => %10s\n",
                "id", this.id, "coins", this.coins, "goods", this.goods);
    }
}

/**
 * Main class for the basic Hibernate example.
 **/
public class HibernateExample
{
```

```
public static class PlayerDAO {
    public static class NotEnoughException extends RuntimeException {
        public NotEnoughException(String message) {
            super(message);
        }
    }

    // Run SQL code in a way that automatically handles the
    // transaction retry logic so we don't have to duplicate it in
    // various places.
    public Object runTransaction(Session session, Function<Session, Object> fn) {
        Object resultObject = null;

        Transaction txn = session.beginTransaction();
        try {
            resultObject = fn.apply(session);
            txn.commit();
            System.out.println("APP: COMMIT;");
        } catch (JDBCException e) {
            System.out.println("APP: ROLLBACK BY JDBC ERROR;");
            txn.rollback();
        } catch (NotEnoughException e) {
            System.out.printf("APP: ROLLBACK BY LOGIC; %s", e.getMessage());
            txn.rollback();
        }
        return resultObject;
    }

    public Function<Session, Object> createPlayers(List<PlayerBean> players) throws
        ↪ JDBCException {
        return session -> {
            Integer addedPlayerAmount = 0;
            for (PlayerBean player: players) {
                session.persist(player);
                addedPlayerAmount ++;
            }
            System.out.printf("APP: createPlayers() --> %d\n", addedPlayerAmount);
            return addedPlayerAmount;
        };
    }

    public Function<Session, Object> buyGoods(String sellId, String buyId, Integer amount,
        ↪ Integer price) throws JDBCException {
        return session -> {
            PlayerBean sellPlayer = session.get(PlayerBean.class, sellId);
```

```
            PlayerBean buyPlayer = session.get(PlayerBean.class, buyId);

            if (buyPlayer == null || sellPlayer == null) {
                throw new NotEnoughException("sell or buy player not exist");
            }

            if (buyPlayer.getCoins() < price || sellPlayer.getGoods() < amount) {
                throw new NotEnoughException("coins or goods not enough, rollback");
            }

            buyPlayer.setGoods(buyPlayer.getGoods() + amount);
            buyPlayer.setCoins(buyPlayer.getCoins() - price);
            session.persist(buyPlayer);

            sellPlayer.setGoods(sellPlayer.getGoods() - amount);
            sellPlayer.setCoins(sellPlayer.getCoins() + price);
            session.persist(sellPlayer);

            System.out.printf("APP: buyGoods --> sell: %s, buy: %s, amount: %d, price: %d\n",
                ↪  sellId, buyId, amount, price);
            return 0;
        };
    }

    public Function<Session, Object> getPlayerByID(String id) throws JDBCException {
        return session -> session.get(PlayerBean.class, id);
    }

    public Function<Session, Object> printPlayers(Integer limit) throws JDBCException {
        return session -> {
            NativeQuery<PlayerBean> limitQuery = session.createNativeQuery("SELECT * FROM
                ↪ player_hibernate LIMIT :limit", PlayerBean.class);
            limitQuery.setParameter("limit", limit);
            List<PlayerBean> players = limitQuery.getResultList();

            for (PlayerBean player: players) {
                System.out.println("\n[printPlayers]:\n" + player);
            }
            return 0;
        };
    }

    public Function<Session, Object> countPlayers() throws JDBCException {
        return session -> {
```

```
            Query<Long> countQuery = session.createQuery("SELECT count(player_hibernate) FROM
                ↪  PlayerBean player_hibernate", Long.class);
            return countQuery.getSingleResult();
        };
    }
}

public static void main(String[] args) {
    // 1. Create a SessionFactory based on our hibernate.cfg.xml configuration
    // file, which defines how to connect to the database.
    SessionFactory sessionFactory
            = new Configuration()
            .configure("hibernate.cfg.xml")
            .addAnnotatedClass(PlayerBean.class)
            .buildSessionFactory();

    try (Session session = sessionFactory.openSession()) {
        // 2. And then, create DAO to manager your data.
        PlayerDAO playerDAO = new PlayerDAO();

        // 3. Run some simple examples.

        // Create a player who has 1 coin and 1 goods.
        playerDAO.runTransaction(session, playerDAO.createPlayers(Collections.singletonList(
                new PlayerBean("test", 1, 1))));

        // Get a player.
        PlayerBean testPlayer = (PlayerBean)playerDAO.runTransaction(session, playerDAO.
            ↪ getPlayerByID("test"));
        System.out.printf("PlayerDAO.getPlayer:\n    => id: %s\n    => coins: %s\n    =>
            ↪ goods: %s\n",
                testPlayer.getId(), testPlayer.getCoins(), testPlayer.getGoods());

        // Count players amount.
        Long count = (Long)playerDAO.runTransaction(session, playerDAO.countPlayers());
        System.out.printf("PlayerDAO.countPlayers:\n    => %d total players\n", count);

        // Print 3 players.
        playerDAO.runTransaction(session, playerDAO.printPlayers(3));

        // 4. Explore more.

        // Player 1: id is "1", has only 100 coins.
        // Player 2: id is "2", has 114514 coins, and 20 goods.
        PlayerBean player1 = new PlayerBean("1", 100, 0);
```

```
            PlayerBean player2 = new PlayerBean("2", 114514, 20);

            // Create two players "by hand", using the INSERT statement on the backend.
            int addedCount = (Integer)playerDAO.runTransaction(session,
                    playerDAO.createPlayers(Arrays.asList(player1, player2)));
            System.out.printf("PlayerDAO.createPlayers:\n    => %d total inserted players\n",
                ↪ addedCount);

            // Player 1 wants to buy 10 goods from player 2.
            // It will cost 500 coins, but player 1 cannot afford it.
            System.out.println("\nPlayerDAO.buyGoods:\n    => this trade will fail");
            Integer updatedCount = (Integer)playerDAO.runTransaction(session,
                    playerDAO.buyGoods(player2.getId(), player1.getId(), 10, 500));
            System.out.printf("PlayerDAO.buyGoods:\n    => %d total update players\n",
                ↪ updatedCount);

            // So player 1 has to reduce the incoming quantity to two.
            System.out.println("\nPlayerDAO.buyGoods:\n    => this trade will success");
            updatedCount = (Integer)playerDAO.runTransaction(session,
                    playerDAO.buyGoods(player2.getId(), player1.getId(), 2, 100));
            System.out.printf("PlayerDAO.buyGoods:\n    => %d total update players\n",
                ↪ updatedCount);
        } finally {
            sessionFactory.close();
        }
    }
}
```

进入目录 `plain-java-jdbc`：

```
cd plain-java-jdbc
```

目录结构如下所示：

```
.
├── Makefile
├── plain-java-jdbc.iml
├── pom.xml
└── src
    └── main
        ├── java
        │   └── com
        │       └── pingcap
        │           └── JDBCExample.java
        └── resources
            └── dbinit.sql
```

其中，dbinit.sql 为数据表初始化语句：

```
USE test;
DROP TABLE IF EXISTS player;

CREATE TABLE player (
    `id` VARCHAR(36),
    `coins` INTEGER,
    `goods` INTEGER,
   PRIMARY KEY (`id`)
);
```

JDBCExample.java 是 plain-java-jdbc 这个示例程序的主体。因为 TiDB 与 MySQL 协议兼容，因此，需要初始化一个 MySQL 协议的数据源 MysqlDataSource，以此连接到 TiDB。并在其后，初始化 PlayerDAO，用来管理数据对象，进行增删改查等操作。

PlayerDAO 是程序用来管理数据对象的类。其中 DAO 是 Data Access Object 的缩写。在其中定义了一系列数据的操作方法，用来对提供数据的写入能力。

PlayerBean 是数据实体类，为数据库表在程序内的映射。PlayerBean 的每个属性都对应着 player 表的一个字段。

```
package com.pingcap;

import com.mysql.cj.jdbc.MysqlDataSource;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.*;

/**
 * Main class for the basic JDBC example.
 **/
public class JDBCExample
{
    public static class PlayerBean {
        private String id;
        private Integer coins;
        private Integer goods;

        public PlayerBean() {
        }

        public PlayerBean(String id, Integer coins, Integer goods) {
            this.id = id;
            this.coins = coins;
```

```
        this.goods = goods;
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public Integer getCoins() {
        return coins;
    }

    public void setCoins(Integer coins) {
        this.coins = coins;
    }

    public Integer getGoods() {
        return goods;
    }

    public void setGoods(Integer goods) {
        this.goods = goods;
    }

    @Override
    public String toString() {
        return String.format("    %-8s => %10s\n    %-8s => %10s\n    %-8s => %10s\n",
                "id", this.id, "coins", this.coins, "goods", this.goods);
    }
}

/**
 * Data access object used by 'ExampleDataSource'.
 * Example for CURD and bulk insert.
 */
public static class PlayerDAO {
    private final MysqlDataSource ds;
    private final Random rand = new Random();

    PlayerDAO(MysqlDataSource ds) {
        this.ds = ds;
    }
```

```
/**
 * Create players by passing in a List of PlayerBean.
 *
 * @param players Will create players list
 * @return The number of create accounts
 */
public int createPlayers(List<PlayerBean> players){
    int rows = 0;

    Connection connection = null;
    PreparedStatement preparedStatement = null;
    try {
        connection = ds.getConnection();
        preparedStatement = connection.prepareStatement("INSERT INTO player (id, coins,
            ↪ goods) VALUES (?, ?, ?)");
    } catch (SQLException e) {
        System.out.printf("[createPlayers] ERROR: { state => %s, cause => %s, message =>
            ↪ %s }\n",
                e.getSQLState(), e.getCause(), e.getMessage());
        e.printStackTrace();

        return -1;
    }

    try {
        for (PlayerBean player : players) {
            preparedStatement.setString(1, player.getId());
            preparedStatement.setInt(2, player.getCoins());
            preparedStatement.setInt(3, player.getGoods());

            preparedStatement.execute();
            rows += preparedStatement.getUpdateCount();
        }
    } catch (SQLException e) {
        System.out.printf("[createPlayers] ERROR: { state => %s, cause => %s, message =>
            ↪ %s }\n",
                e.getSQLState(), e.getCause(), e.getMessage());
        e.printStackTrace();
    } finally {
        try {
            connection.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
```

```
        }

        System.out.printf("\n[createPlayers]:\n    '%s'\n", preparedStatement);
        return rows;
    }

    /**
     * Buy goods and transfer funds between one player and another in one transaction.
     * @param sellId Sell player id.
     * @param buyId Buy player id.
     * @param amount Goods amount, if sell player has not enough goods, the trade will break.
     * @param price Price should pay, if buy player has not enough coins, the trade will
         ↪ break.
     *
     * @return The number of effected players.
     */
    public int buyGoods(String sellId, String buyId, Integer amount, Integer price) {
        int effectPlayers = 0;

        Connection connection = null;
        try {
            connection = ds.getConnection();
        } catch (SQLException e) {
            System.out.printf("[buyGoods] ERROR: { state => %s, cause => %s, message => %s }\
                ↪ n",
                    e.getSQLState(), e.getCause(), e.getMessage());
            e.printStackTrace();
            return effectPlayers;
        }

        try {
            connection.setAutoCommit(false);

            PreparedStatement playerQuery = connection.prepareStatement("SELECT * FROM player
                ↪  WHERE id=? OR id=? FOR UPDATE");
            playerQuery.setString(1, sellId);
            playerQuery.setString(2, buyId);
            playerQuery.execute();

            PlayerBean sellPlayer = null;
            PlayerBean buyPlayer = null;

            ResultSet playerQueryResultSet = playerQuery.getResultSet();
            while (playerQueryResultSet.next()) {
                PlayerBean player =  new PlayerBean(
```

```
                playerQueryResultSet.getString("id"),
                playerQueryResultSet.getInt("coins"),
                playerQueryResultSet.getInt("goods")
        );

        System.out.println("\n[buyGoods]:\n    'check goods and coins enough'");
        System.out.println(player);

        if (sellId.equals(player.getId())) {
            sellPlayer = player;
        } else {
            buyPlayer = player;
        }
    }

    if (sellPlayer == null || buyPlayer == null) {
        throw new SQLException("player not exist.");
    }

    if (sellPlayer.getGoods().compareTo(amount) < 0) {
        throw new SQLException(String.format("sell player %s goods not enough.",
            ↪ sellId));
    }

    if (buyPlayer.getCoins().compareTo(price) < 0) {
        throw new SQLException(String.format("buy player %s coins not enough.", buyId
            ↪ ));
    }

    PreparedStatement transfer = connection.prepareStatement("UPDATE player set goods
        ↪  = goods + ?, coins = coins + ? WHERE id=?");
    transfer.setInt(1, -amount);
    transfer.setInt(2, price);
    transfer.setString(3, sellId);
    transfer.execute();
    effectPlayers += transfer.getUpdateCount();

    transfer.setInt(1, amount);
    transfer.setInt(2, -price);
    transfer.setString(3, buyId);
    transfer.execute();
    effectPlayers += transfer.getUpdateCount();

    connection.commit();
```

```
            System.out.println("\n[buyGoods]:\n    'trade success'");
        } catch (SQLException e) {
            System.out.printf("[buyGoods] ERROR: { state => %s, cause => %s, message => %s }\
                ↪ n",
                    e.getSQLState(), e.getCause(), e.getMessage());

            try {
                System.out.println("[buyGoods] Rollback");

                connection.rollback();
            } catch (SQLException ex) {
                // do nothing
            }
        } finally {
            try {
                connection.close();
            } catch (SQLException e) {
                // do nothing
            }
        }

        return effectPlayers;
    }

    /**
     * Get the player info by id.
     *
     * @param id Player id.
     * @return The player of this id.
     */
    public PlayerBean getPlayer(String id) {
        PlayerBean player = null;

        try (Connection connection = ds.getConnection()) {
            PreparedStatement preparedStatement = connection.prepareStatement("SELECT * FROM
                ↪ player WHERE id = ?");
            preparedStatement.setString(1, id);
            preparedStatement.execute();

            ResultSet res = preparedStatement.executeQuery();
            if(!res.next()) {
                System.out.printf("No players in the table with id %s", id);
            } else {
                player = new PlayerBean(res.getString("id"), res.getInt("coins"), res.getInt(
                    ↪ "goods"));
```

```
            }
        } catch (SQLException e) {
            System.out.printf("PlayerDAO.getPlayer ERROR: { state => %s, cause => %s, message
                ↪  => %s }\n",
                    e.getSQLState(), e.getCause(), e.getMessage());
        }

        return player;
    }

    /**
     * Insert randomized account data (id, coins, goods) using the JDBC fast path for
     * bulk inserts.  The fastest way to get data into TiDB is using the
     * TiDB Lightning(https://docs.pingcap.com/tidb/stable/tidb-lightning-overview).
     * However, if you must bulk insert from the application using INSERT SQL, the best
     * option is the method shown here. It will require the following:
     *
     *    Add `rewriteBatchedStatements=true` to your JDBC connection settings.
     *    Setting rewriteBatchedStatements to true now causes CallableStatements
     *    with batched arguments to be re-written in the form "CALL (...); CALL (...); ..."
     *    to send the batch in as few client/server round trips as possible.
     *    https://dev.mysql.com/doc/relnotes/connector-j/5.1/en/news-5-1-3.html
     *
     *    You can see the `rewriteBatchedStatements` param effect logic at
     *    implement function: `com.mysql.cj.jdbc.StatementImpl.executeBatchUsingMultiQueries`
     *
     * @param total Add players amount.
     * @param batchSize Bulk insert size for per batch.
     *
     * @return The number of new accounts inserted.
     */
    public int bulkInsertRandomPlayers(Integer total, Integer batchSize) {
        int totalNewPlayers = 0;

        try (Connection connection = ds.getConnection()) {
            // We're managing the commit lifecycle ourselves, so we can
            // control the size of our batch inserts.
            connection.setAutoCommit(false);

            // In this example we are adding 500 rows to the database,
            // but it could be any number.  What's important is that
            // the batch size is 128.
            try (PreparedStatement pstmt = connection.prepareStatement("INSERT INTO player (
                ↪ id, coins, goods) VALUES (?, ?, ?)")) {
                for (int i=0; i<=(total/batchSize);i++) {
```

```
                for (int j=0; j<batchSize; j++) {
                    String id = UUID.randomUUID().toString();
                    pstmt.setString(1, id);
                    pstmt.setInt(2, rand.nextInt(10000));
                    pstmt.setInt(3, rand.nextInt(10000));
                    pstmt.addBatch();
                }

                int[] count = pstmt.executeBatch();
                totalNewPlayers += count.length;
                System.out.printf("\nPlayerDAO.bulkInsertRandomPlayers:\n    '%s'\n",
                    ↪ pstmt);
                System.out.printf("    => %s row(s) updated in this batch\n", count.
                    ↪ length);
            }
            connection.commit();
        } catch (SQLException e) {
            System.out.printf("PlayerDAO.bulkInsertRandomPlayers ERROR: { state => %s,
                ↪ cause => %s, message => %s }\n",
                e.getSQLState(), e.getCause(), e.getMessage());
        }
    } catch (SQLException e) {
        System.out.printf("PlayerDAO.bulkInsertRandomPlayers ERROR: { state => %s, cause
            ↪ => %s, message => %s }\n",
            e.getSQLState(), e.getCause(), e.getMessage());
    }
    return totalNewPlayers;
}


/**
 * Print a subset of players from the data store by limit.
 *
 * @param limit Print max size.
 */
public void printPlayers(Integer limit) {
    try (Connection connection = ds.getConnection()) {
        PreparedStatement preparedStatement = connection.prepareStatement("SELECT * FROM
            ↪ player LIMIT ?");
        preparedStatement.setInt(1, limit);
        preparedStatement.execute();

        ResultSet res = preparedStatement.executeQuery();
        while (!res.next()) {
            PlayerBean player = new PlayerBean(res.getString("id"),
```

```
                        res.getInt("coins"), res.getInt("goods"));
                System.out.println("\n[printPlayers]:\n" + player);
            }
        } catch (SQLException e) {
            System.out.printf("PlayerDAO.printPlayers ERROR: { state => %s, cause => %s,
                ↪ message => %s }\n",
                    e.getSQLState(), e.getCause(), e.getMessage());
        }
    }


    /**
     * Count players from the data store.
     *
     * @return All players count
     */
    public int countPlayers() {
        int count = 0;

        try (Connection connection = ds.getConnection()) {
            PreparedStatement preparedStatement = connection.prepareStatement("SELECT count
                ↪ (*) FROM player");
            preparedStatement.execute();

            ResultSet res = preparedStatement.executeQuery();
            if(res.next()) {
                count = res.getInt(1);
            }
        } catch (SQLException e) {
            System.out.printf("PlayerDAO.countPlayers ERROR: { state => %s, cause => %s,
                ↪ message => %s }\n",
                    e.getSQLState(), e.getCause(), e.getMessage());
        }

        return count;
    }
}

public static void main(String[] args) {
    // 1. Configure the example database connection.

    // 1.1 Create a mysql data source instance.
    MysqlDataSource mysqlDataSource = new MysqlDataSource();

    // 1.2 Set server name, port, database name, username and password.
```

```
        mysqlDataSource.setServerName("localhost");
        mysqlDataSource.setPortNumber(4000);
        mysqlDataSource.setDatabaseName("test");
        mysqlDataSource.setUser("root");
        mysqlDataSource.setPassword("");

        // Or you can use jdbc string instead.
        // mysqlDataSource.setURL("jdbc:mysql://{host}:{port}/test?user={user}&password={password
            ↪ }");

        // 2. And then, create DAO to manager your data.
        PlayerDAO dao = new PlayerDAO(mysqlDataSource);

        // 3. Run some simple examples.

        // Create a player, who has a coin and a goods..
        dao.createPlayers(Collections.singletonList(new PlayerBean("test", 1, 1)));

        // Get a player.
        PlayerBean testPlayer = dao.getPlayer("test");
        System.out.printf("PlayerDAO.getPlayer:\n    => id: %s\n    => coins: %s\n    => goods: %
            ↪ s\n",
                testPlayer.getId(), testPlayer.getCoins(), testPlayer.getGoods());

        // Create players with bulk inserts. Insert 1919 players totally, with 114 players per
            ↪ batch.
        int addedCount = dao.bulkInsertRandomPlayers(1919, 114);
        System.out.printf("PlayerDAO.bulkInsertRandomPlayers:\n    => %d total inserted players\n
            ↪ ", addedCount);

        // Count players amount.
        int count = dao.countPlayers();
        System.out.printf("PlayerDAO.countPlayers:\n    => %d total players\n", count);

        // Print 3 players.
        dao.printPlayers(3);

        // 4. Explore more.

        // Player 1: id is "1", has only 100 coins.
        // Player 2: id is "2", has 114514 coins, and 20 goods.
        PlayerBean player1 = new PlayerBean("1", 100, 0);
        PlayerBean player2 = new PlayerBean("2", 114514, 20);

        // Create two players "by hand", using the INSERT statement on the backend.
```

```
        addedCount = dao.createPlayers(Arrays.asList(player1, player2));
        System.out.printf("PlayerDAO.createPlayers:\n    => %d total inserted players\n",
            ↪ addedCount);

        // Player 1 wants to buy 10 goods from player 2.
        // It will cost 500 coins, but player 1 cannot afford it.
        System.out.println("\nPlayerDAO.buyGoods:\n    => this trade will fail");
        int updatedCount = dao.buyGoods(player2.getId(), player1.getId(), 10, 500);
        System.out.printf("PlayerDAO.buyGoods:\n    => %d total update players\n", updatedCount);

        // So player 1 has to reduce the incoming quantity to two.
        System.out.println("\nPlayerDAO.buyGoods:\n    => this trade will success");
        updatedCount = dao.buyGoods(player2.getId(), player1.getId(), 2, 100);
        System.out.printf("PlayerDAO.buyGoods:\n    => %d total update players\n", updatedCount);
    }
}
```

4.3.3.4 第 3 步：运行代码

本节将逐步介绍代码的运行方法。

4.3.3.4.1 第 3 步第 1 部分：JDBC 表初始化

> 建议：
>
> 在 Gitpod Playground 中尝试 Mybatis：现在就试试

使用 Mybatis 时，需手动初始化数据库表。若你本地已经安装了 `mysql-client`，且使用本地集群，可直接在 `plain-java-mybatis` 目录下通过 `make prepare` 运行：

```
make prepare
```

或直接执行：

```
mysql --host 127.0.0.1 --port 4000 -u root < src/main/resources/dbinit.sql
```

若你不使用本地集群，或未安装 `mysql-client`，请直接登录你的集群，并运行 `src/main/resources/dbinit.sql` 文件内的 SQL 语句。

> 建议：
>
> 在 Gitpod Playground 中尝试 Hibernate：现在就试试

无需手动初始化表。

> 建议：
>
> 在 Gitpod Playground 中尝试 JDBC：现在就试试

使用 JDBC 时，需手动初始化数据库表，若你本地已经安装了 `mysql-client`，且使用本地集群，可直接在 `plain-java-jdbc` 目录下运行：

```
make mysql
```

或直接执行：

```
mysql --host 127.0.0.1 --port 4000 -u root<src/main/resources/dbinit.sql
```

若你不使用本地集群，或未安装 mysql-client，请直接登录你的集群，并运行 `src/main/resources/dbinit.sql` 文件内的 SQL 语句。

4.3.3.4.2 第 3 步第 2 部分：TiDB Cloud 更改参数

若你使用 TiDB Serverless 集群，更改 `mybatis-config.xml` 内关于 `dataSource.url`、`dataSource.username`、`dataSource.password` 的参数：

```
<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">

<configuration>
    <settings>
        <setting name="cacheEnabled" value="true"/>
        <setting name="lazyLoadingEnabled" value="false"/>
        <setting name="aggressiveLazyLoading" value="true"/>
        <setting name="logImpl" value="LOG4J"/>
    </settings>

    <typeAliases>
        <package name="com.pingcap.dao"/>
    </typeAliases>

    <environments default="development">
        <environment id="development">
            <!-- JDBC transaction manager -->
            <transactionManager type="JDBC"/>
```

```
            <!-- Database pool -->
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.jdbc.Driver"/>
                <property name="url" value="jdbc:mysql://127.0.0.1:4000/test"/>
                <property name="username" value="root"/>
                <property name="password" value=""/>
            </dataSource>
        </environment>
    </environments>

    <mappers>
        <mapper resource="mapper/PlayerMapper.xml"/>
        <mapper resource="mapper/PlayerMapperEx.xml"/>
    </mappers>

</configuration>
```

若你设定的密码为 123456，而且从 TiDB Serverless 集群面板中得到的连接信息为：

- Endpoint: xxx.tidbcloud.com
- Port: 4000
- User: 2aEp24QWEDLqRFs.root

那么此处应将配置文件中 dataSource 节点内更改为：

```
<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE configuration
        PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-config.dtd">

        ...
            <!-- Database pool -->
            <dataSource type="POOLED">
                <property name="driver" value="com.mysql.jdbc.Driver"/>
                <property name="url" value="jdbc:mysql://xxx.tidbcloud.com:4000/test?sslMode=
                    ↪ VERIFY_IDENTITY&amp;enabledTLSProtocols=TLSv1.2,TLSv1.3"/>
                <property name="username" value="2aEp24QWEDLqRFs.root"/>
                <property name="password" value="123456"/>
            </dataSource>
        ...

</configuration>
```

若你使用 TiDB Serverless 集群，更改 hibernate.cfg.xml 内关于 hibernate.connection.url、hibernate.
↪ connection.username、hibernate.connection.password 的参数：

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>

        <!-- Database connection settings -->
        <property name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property>
        <property name="hibernate.dialect">org.hibernate.dialect.TiDBDialect</property>
        <property name="hibernate.connection.url">jdbc:mysql://localhost:4000/test</property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.connection.password"></property>
        <property name="hibernate.connection.autocommit">false</property>

        <!-- Required so a table can be created from the 'PlayerDAO' class -->
        <property name="hibernate.hbm2ddl.auto">create-drop</property>

        <!-- Optional: Show SQL output for debugging -->
        <property name="hibernate.show_sql">true</property>
        <property name="hibernate.format_sql">true</property>
    </session-factory>
</hibernate-configuration>
```

若你设定的密码为 123456，而且从 TiDB Serverless 集群面板中得到的连接信息为：

- Endpoint: xxx.tidbcloud.com
- Port: 4000
- User: 2aEp24QWEDLqRFs.root

那么此处应将配置文件更改为：

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>

        <!-- Database connection settings -->
        <property name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property>
        <property name="hibernate.dialect">org.hibernate.dialect.TiDBDialect</property>
        <property name="hibernate.connection.url">jdbc:mysql://xxx.tidbcloud.com:4000/test?
            ↪ sslMode=VERIFY_IDENTITY&amp;enabledTLSProtocols=TLSv1.2,TLSv1.3</property>
        <property name="hibernate.connection.username">2aEp24QWEDLqRFs.root</property>
```

```
        <property name="hibernate.connection.password">123456</property>
        <property name="hibernate.connection.autocommit">false</property>

        <!-- Required so a table can be created from the 'PlayerDAO' class -->
        <property name="hibernate.hbm2ddl.auto">create-drop</property>

        <!-- Optional: Show SQL output for debugging -->
        <property name="hibernate.show_sql">true</property>
        <property name="hibernate.format_sql">true</property>
    </session-factory>
</hibernate-configuration>
```

若你使用 TiDB Serverless 集群，更改 JDBCExample.java 内关于 Host、 Port、 User、 Password 的参数：

```
mysqlDataSource.setServerName("localhost");
mysqlDataSource.setPortNumber(4000);
mysqlDataSource.setDatabaseName("test");
mysqlDataSource.setUser("root");
mysqlDataSource.setPassword("");
```

若你设定的密码为 123456，而且从 TiDB Serverless 集群面板中得到的连接信息为：

- Endpoint: xxx.tidbcloud.com
- Port: 4000
- User: 2aEp24QWEDLqRFs.root

那么此处应将参数更改为：

```
mysqlDataSource.setServerName("xxx.tidbcloud.com");
mysqlDataSource.setPortNumber(4000);
mysqlDataSource.setDatabaseName("test");
mysqlDataSource.setUser("2aEp24QWEDLqRFs.root");
mysqlDataSource.setPassword("123456");
mysqlDataSource.setSslMode(PropertyDefinitions.SslMode.VERIFY_IDENTITY.name());
mysqlDataSource.setEnabledTLSProtocols("TLSv1.2,TLSv1.3");
```

4.3.3.4.3 第 3 步第 3 部分：运行

运行 make，这是以下四个操作的组合：

- 创建表 (make prepare)：

  ```
  mysql --host 127.0.0.1 --port 4000 -u root < src/main/resources/dbinit.sql
  mysql --host 127.0.0.1 --port 4000 -u root -e "TRUNCATE test.player"
  ```

- 清理并构建 (make gen)：

```
rm -f src/main/java/com/pingcap/model/Player.java
rm -f src/main/java/com/pingcap/model/PlayerMapper.java
rm -f src/main/resources/mapper/PlayerMapper.xml
mvn mybatis-generator:generate
```

- 清理并构建 (make build)：mvn clean package
- 运行 (make run)：java -jar target/plain-java-mybatis-0.0.1-jar-with-dependencies.jar

你也可以单独运行这四个 make 命令或原生命令。

运行 make，这是以下两个操作的组合：

- 清理并构建 (make build)：mvn clean package
- 运行 (make run)：java -jar target/plain-java-hibernate-0.0.1-jar-with-dependencies.jar

你也可以单独运行这两个 make 命令或原生命令。

运行 make，这是以下两个操作的组合：

- 清理并构建 (make build)：mvn clean package
- 运行 (make run)：java -jar target/plain-java-jdbc-0.0.1-jar-with-dependencies.jar

你也可以单独运行这两个 make 命令或原生命令。

4.3.3.5 第 4 步：预期输出

Mybatis 预期输出

Hibernate 预期输出

JDBC 预期输出

4.3.4 使用 Django 构建 TiDB 应用程序

本文档将展示如何使用 Django 构建一个 TiDB Web 应用程序。使用 django-tidb 模块作为数据访问能力的框架。示例应用程序的代码可从 Github 下载。

这是一个较为完整的构建 Restful API 的示例应用程序，展示了一个使用 TiDB 作为数据库的通用 Django 后端服务。该示例设计了以下过程，用于还原一个现实场景：

这是一个关于游戏的例子，每个玩家有两个属性：金币数 coins 和货物数 goods。且每个玩家都拥有一个字段 id，作为玩家的唯一标识。玩家在金币数和货物数充足的情况下，可以自由地交易。

你可以以此示例为基础，构建自己的应用程序。

> 建议：
>
> 在云原生开发环境中尝试 Django 构建 TiDB 应用程序。
>
> 预配置完成的环境将自动启动 TiDB 集群、获取和运行代码，只需要一个链接：现在就试试。

4.3.4.1 第 1 步：启动你的 TiDB 集群

本节将介绍 TiDB 集群的启动方法。

创建 TiDB Serverless 集群。

你可以部署一个本地测试的 TiDB 集群或正式的 TiDB 集群。详细步骤，请参考：

- 部署本地测试 TiDB 集群
- 部署正式 TiDB 集群

基于 Git 的预配置的开发环境：现在就试试

该环境会自动克隆代码，并通过 TiUP 部署测试集群。

4.3.4.2 第 2 步：安装 Python

请在你的计算机上下载并安装 Python。本文的示例使用 Django 3.2.16 版本。根据 Django 文档，Django 3.2.16 版本支持 Python 3.6、3.7、3.8、3.9 和 3.10 版本，推荐使用 Python 3.10 版本。

4.3.4.3 第 3 步：获取应用程序代码

> 建议：
>
> 如果你希望得到一个与本示例相同依赖的空白程序，而无需示例代码，可参考创建相同依赖空白程序（可选）一节。

请下载或克隆示例代码库 pingcap-inc/tidb-example-python，并进入到目录 `django_example` 中。

4.3.4.4 第 4 步：运行应用程序

接下来运行应用程序代码，将会生成一个 Web 应用程序。你可以使用 `python manage.py migrate` 命令，要求 Django 在数据库 `django` 中创建一个表 `player`。如果你向应用程序的 Restful API 发送请求，这些请求将会在 TiDB 集群上运行数据库事务。

如果你想了解有关此应用程序的代码的详细信息，可参阅实现细节部分。

4.3.4.4.1 第 4 步第 1 部分：TiDB Cloud 更改参数

若你使用 TiDB Serverless 集群，更改 `example_project/settings.py` 中的 DATABASES 参数：

```
DATABASES = {
    'default': {
        'ENGINE': 'django_tidb',
        'NAME': 'django',
        'USER': 'root',
        'PASSWORD': '',
        'HOST': '127.0.0.1',
        'PORT': 4000,
    },
}
```

另外，由于 TiDB Serverless 需要使用 SSL 连接。因此，需要提供 CA 证书路径。你可以在 TiDB Serverless 安全连接文档 中查看不同操作系统的 CA 证书路径。

若你设定的密码为 123456，而且从 TiDB Serverless 集群面板中得到的连接信息为：

- Endpoint: `xxx.tidbcloud.com`
- Port: 4000
- User: `2aEp24QWEDLqRFs.root`

下面以 macOS 为例，应将参数更改为：

```
DATABASES = {
    'default': {
        'ENGINE': 'django_tidb',
        'NAME': 'django',
        'USER': '2aEp24QWEDLqRFs.root',
        'PASSWORD': '123456',
        'HOST': 'xxx.tidbcloud.com',
        'PORT': 4000,
        'OPTIONS': {
            'ssl': {
                "ca": "/etc/ssl/cert.pem"
            },
        },
    },
}
```

4.3.4.4.2 第 4 步第 2 部分：运行

1. 打开终端，进入 `tidb-example-python` 代码示例目录：

```
cd <path>/tidb-example-python
```

2. 安装项目依赖并进入 django_example 目录：

```
pip install -r requirement.txt
cd django_example
```

3. 运行数据模型迁移：

> 注意：
>
> - 此步骤假定已经存在 django 数据库。
> - 若未创建 django 数据库，可通过 CREATE DATABASE django 语句进行创建。关于创建数据库语句的详细信息，参考CREATE DATABASE。
> - 数据库名称 NAME 可在 example_project/settings.py 的 DATABASES 属性中更改。

这将在你连接的数据库内生成 Django 所需的相应数据表。

```
python manage.py migrate
```

4. 运行应用程序：

```
python manage.py runserver
```

4.3.4.4.3 第 4 步第 3 部分：输出

输出的最后部分应如下所示：

```
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
December 12, 2022 - 08:21:50
Django version 3.2.16, using settings 'example_project.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

如果你想了解有关此应用程序的代码的详细信息，可参阅实现细节部分。

4.3.4.5 第 5 步：HTTP 请求

在运行应用程序后，你可以通过访问根地址 http://localhost:8000 向后端程序发送 HTTP 请求。下面将给出一些示例请求来演示如何使用该服务。

1. 将配置文件 Player.postman_collection.json 导入 Postman。

2. 导入后 Collections > Player 如图所示：

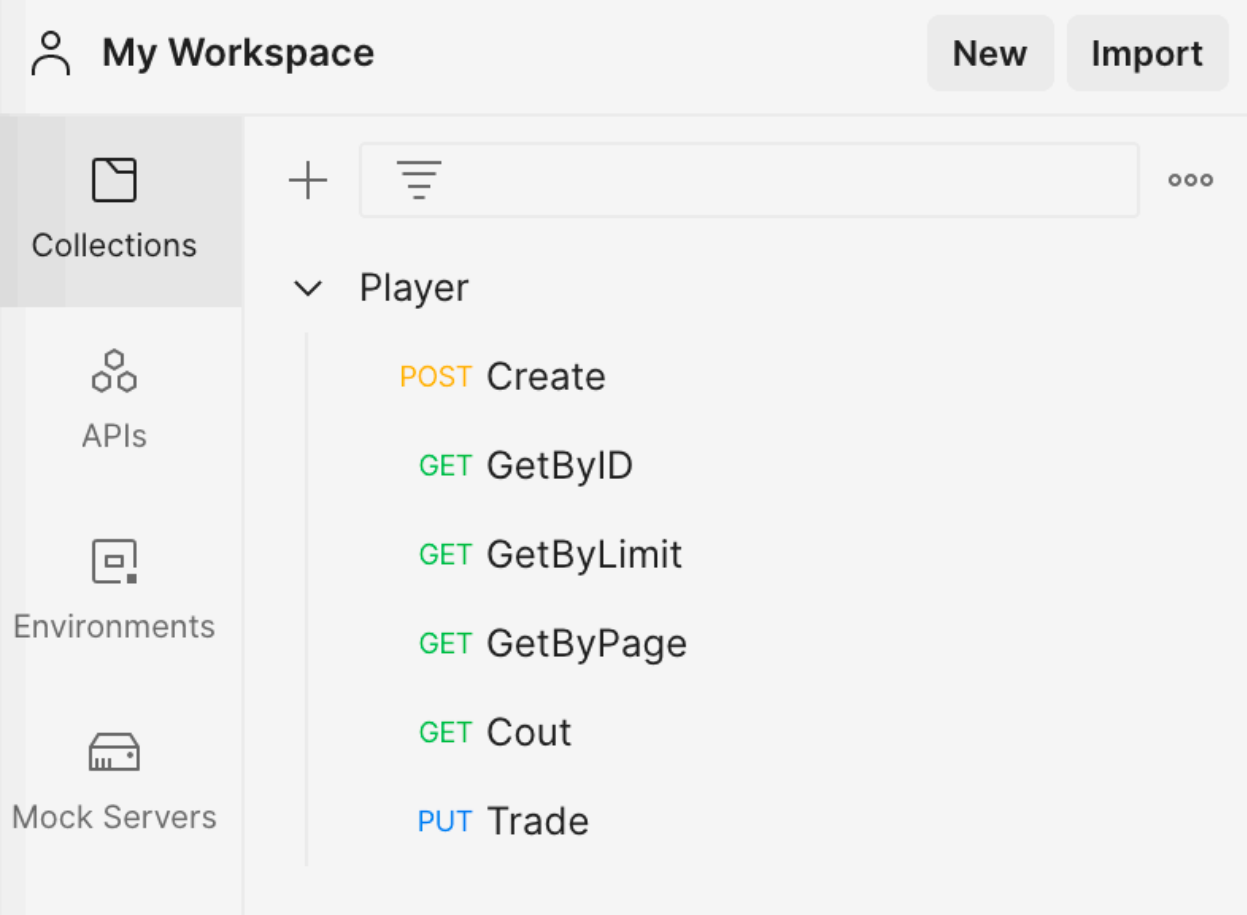


图 8: postman import

3. 发送请求：

- 增加玩家

  点击 Create 标签，点击 Send 按钮，发送 POST 形式的 `http://localhost:8000/player/` 请求。返回值为增加的玩家个数，预期为 1。

- 使用 ID 获取玩家信息

  点击 GetByID 标签，点击 Send 按钮，发送 GET 形式的 `http://localhost:8000/player/1` 请求。返回值为 ID 为 1 的玩家信息。

- 使用 Limit 批量获取玩家信息

  点击 GetByLimit 标签，点击 Send 按钮，发送 GET 形式的 `http://localhost:8000/player/limit/3` 请求。返回值为最多 3 个玩家的信息列表。

- 获取玩家个数

  点击 Count 标签，点击 Send 按钮，发送 GET 形式的 `http://localhost:8000/player/count` 请求。返回值为玩家个数。

- 玩家交易

  点击 Trade 标签，点击 Send 按钮，发送 POST 形式的 `http://localhost:8000/player/trade` 请求。请求参数为售卖玩家 ID `sellID`、购买玩家 ID `buyID`、购买货物数量 `amount` 以及购买消耗金币数 `price`。返回值为交易是否成功。当出现售卖玩家货物不足、购买玩家金币不足或数据库错误时，交易将不成功。并且由于数据库事务保证，不会有玩家的金币或货物丢失的情况。

下面使用 curl 请求服务端。

- 增加玩家

  使用 POST 方法向 `/player` 端点发送请求来增加玩家，例如：

```
curl --location --request POST 'http://localhost:8000/player/' --header 'Content-Type:
    ↪ application/json' --data-raw '[{"coins":100,"goods":20}]'
```

  这里使用 JSON 作为信息的载荷。表示需要创建一个金币数 `coins` 为 100，货物数 `goods` 为 20 的玩家。返回值为创建的玩家信息：

```
create 1 players.
```

- 使用 ID 获取玩家信息

  使用 GET 方法向 `/player` 端点发送请求来获取玩家信息。此外，还需要在路径上给出玩家的 ID 参数，即 `/player/{id}`。例如，在请求 ID 为 1 的玩家时：

```
curl --location --request GET 'http://localhost:8000/player/1'
```

  返回值为 ID 为 1 的玩家的信息：

```
{
"coins": 200,
"goods": 10,
"id": 1
}
```

- 使用 Limit 批量获取玩家信息

  使用 GET 方法向 `/player/limit` 端点发送请求来获取玩家信息。此外，还需要在路径上给出限制查询的玩家信息的总数，即 `/player/limit/{limit}`。例如，在请求最多 3 个玩家的信息时：

```
curl --location --request GET 'http://localhost:8000/player/limit/3'
```

  返回值为玩家信息的列表：

```
[
{
    "coins": 200,
    "goods": 10,
    "id": 1
},
```

```
{
    "coins": 0,
    "goods": 30,
    "id": 2
},
{
    "coins": 100,
    "goods": 20,
    "id": 3
}
]
```

- 获取玩家个数

  使用 GET 方法向 /player/count 端点发送请求来获取玩家个数：

```
curl --location --request GET 'http://localhost:8000/player/count'
```

  返回值为玩家个数：

```
4
```

- 玩家交易

  使用 POST 方法向 /player/trade 端点发送请求来发起玩家间的交易，例如：

```
curl --location --request POST 'http://localhost:8000/player/trade' \
--header 'Content-Type: application/x-www-form-urlencoded' \
--data-urlencode 'sellID=1' \
--data-urlencode 'buyID=2' \
--data-urlencode 'amount=10' \
--data-urlencode 'price=100'
```

  这里使用 Form Data 作为信息的载荷。表示售卖玩家 ID sellID 为 1、购买玩家 ID buyID 为 2、购买货物数量 amount 为 10、购买消耗金币数 price 为 100。

  返回值为交易是否成功：

```
true
```

  当出现售卖玩家货物不足、购买玩家金币不足或数据库错误时，交易将不成功。并且由于数据库事务保证，不会有玩家的金币或货物丢失的情况。

为方便测试，你可以使用 request.sh 脚本依次发送以下请求：

1. 循环创建 10 名玩家
2. 获取 ID 为 1 的玩家信息
3. 获取至多 3 名玩家信息列表
4. 获取玩家总数

5. ID 为 1 的玩家作为售出方，ID 为 2 的玩家作为购买方，购买 10 个货物，耗费 100 金币

使用 `./request.sh` 命令运行此脚本，运行结果如下所示：

```
> ./request.sh
loop to create 10 players:
create 1 players.create 1 players.create 1 players.create 1 players.create 1 players.create 1
    ↪ players.create 1 players.create 1 players.create 1 players.create 1 players.

get player 1:
{"id": 1, "coins": 100, "goods": 20}

get players by limit 3:
[{"id": 1, "coins": 100, "goods": 20}, {"id": 2, "coins": 100, "goods": 20}, {"id": 3, "coins":
    ↪ 100, "goods": 20}]

get players count:
10

trade by two players:
trade successful
```

4.3.4.6 实现细节

本小节介绍示例应用程序项目中的组件。

4.3.4.6.1 总览

本示例项目的目录树大致如下所示：

```
.
├── example_project
|   ├── __init__.py
|   ├── asgi.py
|   ├── settings.py
|   ├── urls.py
|   └── wsgi.py
├── player
|   ├── __init__.py
|   ├── admin.py
|   ├── apps.py
|   ├── migrations
|   |   ├── 0001_initial.py
|   |   └── __init__.py
|   ├── models.py
|   ├── tests.py
```

```
|   ├── urls.py
|   └── views.py
└── manage.py
```

其中：

- 每一个文件夹中的 `__init__.py` 文件声明了该文件夹是一个 Python 包。
- `manage.py` 为 Django 自动生成的用于管理项目的脚本。
- `example_project` 包含项目级别的代码：
  - `settings.py` 声明了项目的配置，如数据库地址、密码、使用的数据库方言等。
  - `urls.py` 配置了项目的根路由。
- `player` 是项目中提供对 `Player` 数据模型管理、数据查询的包，这在 Django 中被称为应用。你可以使用 `python manage.py startapp player` 来创建一个空白的 `player` 应用。
  - `models.py` 定义了 `Player` 数据模型。
  - `migrations` 是一组数据模型迁移脚本。你可以使用 `python manage.py makemigrations player` 命令自动分析 `models.py` 文件中定义的数据对象，并生成迁移脚本。
  - `urls.py` 定义了应用的路由。
  - `views.py` 提供了应用的逻辑代码。

> 注意：
>
> 由于 Django 的设计采用了可插拔模式，因此，你需要在创建应用后，在项目中进行注册。在本示例中，注册过程就是在 `example_project/settings.py` 文件中，在 `INSTALLED_APPS` 对象内添加 `'player.apps.PlayerConfig'` 条目。你可以参考示例代码 `settings.py` 以获得更多信息。

4.3.4.6.2 项目配置

本节将简要介绍 `example_project` 包内 `settings.py` 的重要配置。这个文件包含了 Django 项目的配置，声明了项目包含的应用、中间件、连接的数据库等信息。你可以通过创建相同依赖空白程序这一节来了解此配置文件的生成流程，也可直接在项目中使用 `settings.py` 文件。关于 Django 配置的更多信息，参考 Django 配置文档。

```
...

### Application definition

INSTALLED_APPS = [
    'player.apps.PlayerConfig',
    'django.contrib.admin',
    'django.contrib.auth',
```

```
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]

MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.middleware.common.CommonMiddleware',
    # 'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]

...

### Database
### https://docs.djangoproject.com/en/3.2/ref/settings/#databases

DATABASES = {
    'default': {
        'ENGINE': 'django_tidb',
        'NAME': 'django',
        'USER': 'root',
        'PASSWORD': '',
        'HOST': '127.0.0.1',
        'PORT': 4000,
    },
}
DEFAULT_AUTO_FIELD = 'django.db.models.AutoField'

...
```

其中：

- INSTALLED_APPS：启用的应用全限定名称列表。
- MIDDLEWARE：启用的中间件列表。由于本示例无需 CsrfViewMiddleware 中间件，因此其被注释。
- DATABASES：数据库配置。其中，ENGINE 一项被配置为 django_tidb，这遵循了 django-tidb 的配置要求。

4.3.4.6.3 根路由

在 example_project 包中的 urls.py 文件中编写了根路由：

```
from django.contrib import admin
```

```
from django.urls import include, path

urlpatterns = [
    path('player/', include('player.urls')),
    path('admin/', admin.site.urls),
]
```

在上面的示例中，根路由将 player/ 路径指向 player.urls。即，player 包下的 urls.py 将负责处理所有以 player/ 开头的 URL 请求。关于更多 Django URL 调度器的信息，请参考 Django URL 调度器文档。

4.3.4.6.4 player 应用

player 应用实现了对 Player 对象的数据模型迁移、对象持久化、接口实现等功能。

数据模型

models.py 文件内包含 Player 数据模型，这个模型对应了数据库的一张表。

```
from django.db import models

### Create your models here.


class Player(models.Model):
    id = models.AutoField(primary_key=True)
    coins = models.IntegerField()
    goods = models.IntegerField()

    objects = models.Manager()

    class Meta:
        db_table = "player"

    def as_dict(self):
        return {
            "id": self.id,
            "coins": self.coins,
            "goods": self.goods,
        }
```

在上面的示例中，数据模型中有一个子类 Meta，这些子类给了 Django 额外的信息，用以指定数据模型的元信息。其中，db_table 声明此数据模型对应的表名为 player。关于模型元信息的全部选项可查看 Django 模型 Meta 选项文档。

此外，数据模型中定义了 id、coins 及 goods 三个属性：

- id：models.AutoField(primary_key=True) 表示其为一个自动递增的主键。
- coins：models.IntegerField() 表示其为一个 Integer 类型的字段。

- goods：models.IntegerField() 表示其为一个 Integer 类型的字段。

关于数据模型的详细信息，可查看 Django 模型文档。

数据模型迁移

Django 以 Python 数据模型定义代码为依赖，对数据库模型进行迁移。因此，它会生成一系列数据库模型迁移脚本，用于解决代码与数据库之间的差异。在 models.py 中定义完 Player 数据模型后，你可以使用 python manage.py makemigrations player 生成迁移脚本。在本文示例中，migrations 包内的 0001_initial.py 就是自动生成的迁移脚本。

```
### Generated by Django 3.2.16 on 2022-11-16 11:09

from django.db import migrations, models


class Migration(migrations.Migration):

    initial = True

    dependencies = [
    ]

    operations = [
        migrations.CreateModel(
            name='Player',
            fields=[
                ('id', models.AutoField(primary_key=True, serialize=False)),
                ('coins', models.IntegerField()),
                ('goods', models.IntegerField()),
            ],
            options={
                'db_table': 'player',
            },
        ),
    ]
```

你可以使用 python manage.py sqlmigrate ... 来预览迁移脚本最终将运行的 SQL 语句。这将极大地减少迁移脚本运行你意料之外的 SQL 语句的可能性。在生成迁移脚本后，推荐至少使用一次此命令预览并仔细检查生成的 SQL 语句。在本示例中，你可以运行 python manage.py sqlmigrate player 0001，其输出为可读的 SQL 语句，有助于开发者对语句进行审核：

```
--
-- Create model Player
--
CREATE TABLE `player` (`id` integer AUTO_INCREMENT NOT NULL PRIMARY KEY, `coins` integer NOT NULL
    ↪ , `goods` integer NOT NULL);
```

生成迁移脚本后，你可以使用 `python manage.py migrate` 实施数据迁移。此命令拥有幂等性，其运行后将在数据库内保存一条运行记录以完成幂等保证。因此，你可以多次运行此命令，而无需担心重复运行 SQL 语句。

应用路由

在根路由一节中，示例程序将 `player/` 路径指向了 `player.urls`。本节将展开叙述 `player` 包下的 `urls.py` 应用路由：

```
from django.urls import path

from . import views

urlpatterns = [
    path('', views.create, name='create'),
    path('count', views.count, name='count'),
    path('limit/<int:limit>', views.limit_list, name='limit_list'),
    path('<int:player_id>', views.get_by_id, name='get_by_id'),
    path('trade', views.trade, name='trade'),
]
```

应用路由注册了 5 个路径：

- `''`：被指向了 `views.create` 函数。
- `'count'`：被指向了 `views.count` 函数。
- `'limit/<int:limit>'`：被指向了 `views.limit_list` 函数。此处路径包含一个 `<int:limit>` 路径变量，其中：
  - `int` 是指这个参数其将被验证是否为 `int` 类型。
  - `limit` 是指此参数的值将被映射至名为 `limit` 的函数入参中。
- `'<int:player_id>'`：被指向了 `views.get_by_id` 函数，此处路径包含一个 `<int:player_id>` 路径变量。
- `'trade'`：被指向了 `views.trade` 函数。

此外，应用路由是根路由转发而来的，因此将在 URL 匹配时包含根路由配置的路径。如上面示例所示，根路由配置为 `player/` 转发至此应用路由，那么，应用路由中的：

- `''` 在实际的请求中为 `http(s)://<host>(:<port>)/player`。
- `'count'` 在实际的请求中为 `http(s)://<host>(:<port>)/player/count`。
- `'limit/<int:limit>'` 以 `limit` 为 3 为例，在实际的请求中为 `http(s)://<host>(:<port>)/player/limit/3`。

逻辑实现

逻辑实现代码，在 `player` 包下的 `views.py` 内，这在 Django 中被称为视图。关于 Django 视图的更多信息，参考 Django 视图文档。

```
from django.db import transaction
from django.db.models import F
from django.shortcuts import get_object_or_404

from django.http import HttpResponse, JsonResponse
from django.views.decorators.http import *
from .models import Player
import json


@require_POST
def create(request):
    dict_players = json.loads(request.body.decode('utf-8'))
    players = list(map(
        lambda p: Player(
            coins=p['coins'],
            goods=p['goods']
        ), dict_players))
    result = Player.objects.bulk_create(objs=players)
    return HttpResponse(f'create {len(result)} players.')


@require_GET
def count(request):
    return HttpResponse(Player.objects.count())


@require_GET
def limit_list(request, limit: int = 0):
    if limit == 0:
        return HttpResponse("")
    players = set(Player.objects.all()[:limit])
    dict_players = list(map(lambda p: p.as_dict(), players))
    return JsonResponse(dict_players, safe=False)


@require_GET
def get_by_id(request, player_id: int):
    result = get_object_or_404(Player, pk=player_id).as_dict()
    return JsonResponse(result)


@require_POST
@transaction.atomic
```

```
def trade(request):
    sell_id, buy_id, amount, price = int(request.POST['sellID']), int(request.POST['buyID']), \
                                     int(request.POST['amount']), int(request.POST['price'])
    sell_player = Player.objects.select_for_update().get(id=sell_id)
    if sell_player.goods < amount:
        raise Exception(f'sell player {sell_player.id} goods not enough')

    buy_player = Player.objects.select_for_update().get(id=buy_id)
    if buy_player.coins < price:
        raise Exception(f'buy player {buy_player.id} coins not enough')

    Player.objects.filter(id=sell_id).update(goods=F('goods') - amount, coins=F('coins') + price)
    Player.objects.filter(id=buy_id).update(goods=F('goods') + amount, coins=F('coins') - price)

    return HttpResponse("trade successful")
```

下面将逐一解释代码中的重点部分：

- 装饰器：
  - `@require_GET`：代表此函数仅接受 GET 类型的 HTTP 请求。
  - `@require_POST`：代表此函数仅接受 POST 类型的 HTTP 请求。
  - `@transaction.atomic`：代表此函数内的所有数据库操作将被包含于同一个事务中运行。关于在 Django 中使用事务的更多信息，可参考 Django 数据库事务文档。关于 TiDB 中事物的详细信息，可参考TiDB 事务概览。
- `create` 函数：
  - 获取 `request` 对象中 `body` 的 Payload，并用 `utf-8` 解码：

    ```
    dict_players = json.loads(request.body.decode('utf-8'))
    ```

  - 使用 lambda 中的 `map` 函数，将 dict 类型的 `dict_players` 对象转换为 `Player` 数据模型的列表：

    ```
    players = list(map(
        lambda p: Player(
            coins=p['coins'],
            goods=p['goods']
        ), dict_players))
    ```

  - 调用 `Player` 数据模型的 `bulk_create` 函数，批量添加 `players` 列表，并返回添加的数据条目：

    ```
    result = Player.objects.bulk_create(objs=players)
    return HttpResponse(f'create {len(result)} players.')
    ```

- `count` 函数：调用 `Player` 数据模型的 `count` 函数，并返回所有的数据条目。
- `limit_list` 函数：

  - 短路逻辑，`limit` 为 0 时不发送数据库请求：

```
if limit == 0:
    return HttpResponse("")
```

  - 调用 `Player` 数据模型的 `all` 函数，并使用切片操作符获取前 `limit` 个数据。需要注意的是，Django 不是获取所有数据并在内存中切分前 `limit` 个数据，而是在使用时请求数据库的前 `limit` 个数据。这是由于 Django 重写了切片操作符，并且 QuerySet 对象是惰性的。这意味着对一个未执行的 QuerySet 进行切片，将继续返回一个未执行的 QuerySet，直到你第一次真正的请求 QuerySet 内的数据。例如此处使用 `set` 函数对其进行迭代并返回整个集合。关于 Django QuerySet 的更多信息，你可以参考 Django QuerySet API 文档。

```
players = set(Player.objects.all()[:limit])
```

  - 将返回的 `Player` 数据模型的列表，转为对象为 dict 的列表，并使用 `JsonResponse` 输出。

```
dict_players = list(map(lambda p: p.as_dict(), players))
return JsonResponse(dict_players, safe=False)
```

- `get_by_id` 函数：

  - 使用 `get_object_or_404` 语法糖传入 `player_id`，并将 `Player` 对象转为 dict。如数据不存在，将由此函数返回 404 状态码：

```
result = get_object_or_404(Player, pk=player_id).as_dict()
```

  - 使用 `JsonResponse` 返回数据：

```
return JsonResponse(result)
```

- `trade` 函数：

  - 从 POST Payload 中接收 Form 形式的数据：

```
sell_id, buy_id, amount, price = int(request.POST['sellID']), int(request.POST['buyID'
    ↪ ]), \
                                 int(request.POST['amount']), int(request.POST['price'])
```

  - 调用 `Player` 数据模型的 `select_for_update` 函数对卖家和买家的数据进行加锁，并检查卖家的货物数量和买家的货币数量是否足够。该函数使用了 `@transaction.atomic` 装饰器，任意异常都会导致事务回滚。可以利用这个机制，在任意检查失败的时候，抛出异常，由 Django 进行事务回滚。

```
sell_player = Player.objects.select_for_update().get(id=sell_id)
if sell_player.goods < amount:
    raise Exception(f'sell player {sell_player.id} goods not enough')

buy_player = Player.objects.select_for_update().get(id=buy_id)
if buy_player.coins < price:
    raise Exception(f'buy player {buy_player.id} coins not enough')
```

- 更新卖家与买家的数据。由于这里使用了 `@transaction.atomic` 装饰器，任何异常都将由 Django 回滚事务。因此，请不要在此处使用 `try-except` 语句进行异常处理。如果一定需要处理，请在 except 块中将异常继续抛向上层，以防止因 Django 误认为函数运行正常而提交事务，导致数据错误。

```
Player.objects.filter(id=sell_id).update(goods=F('goods') - amount, coins=F('coins') +
    ↪ price)
Player.objects.filter(id=buy_id).update(goods=F('goods') + amount, coins=F('coins') -
    ↪ price)
```

- 返回交易成功字符串，因为其他情况将导致异常抛出返回：

```
return HttpResponse("trade successful")
```

4.3.4.7 创建相同依赖空白程序（可选）

本程序使用 Django Admin CLI django-admin 构建。你可以安装并使用 `django-admin` 来快速完成 Django 项目的初始化。如果需要快速获得与示例程序 `django_example` 相同的可运行空白应用程序，可以按照以下步骤操作：

1. 初始化 Django 项目 `copy_django_example`：

```
pip install -r requirement.txt
django-admin startproject copy_django_example
cd copy_django_example
```

2. 更改 `DATABASES` 配置：

    1. 打开 `copy_django_example/settings.py` 配置文件
    2. 将 `DATABASES` 部分从指向本地 SQLite 的配置更改为 TiDB 集群的信息：

```
DATABASES = {
    'default': {
        'ENGINE': 'django_tidb',
        'NAME': 'django',
        'USER': 'root',
        'PASSWORD': '',
        'HOST': '127.0.0.1',
        'PORT': 4000,
    },
}
DEFAULT_AUTO_FIELD = 'django.db.models.AutoField'
```

    3. 由于本示例不需要跨域校验，因此你需要注释或删除 `MIDDLEWARE` 中的 `CsrfViewMiddleware`。修改后的 `MIDDLEWARE` 为：

```
MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
```

```
    'django.middleware.common.CommonMiddleware',
    # 'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]
```

至此，你已经完成了一个空白的应用程序，此应用程序与示例应用程序的依赖完全相同。如果需要进一步了解 Django 的使用方法，参考：

- Django 文档
- Django 入门教程

4.3.5 TiDB 和 Python 的简单 CRUD 应用程序

本文档将展示如何使用 TiDB 和 Python 来构造一个简单的 CRUD 应用程序。

> **注意：**
>
> 推荐使用 Python 3.10 及以上版本进行 TiDB 的应用程序的编写。

4.3.5.1 第 1 步：启动你的 TiDB 集群

本节将介绍 TiDB 集群的启动方法。

创建 TiDB Serverless 集群。

你可以部署一个本地测试的 TiDB 集群或正式的 TiDB 集群。详细步骤，请参考：

- 部署本地测试 TiDB 集群
- 部署正式 TiDB 集群

基于 Git 的预配置的开发环境：现在就试试

该环境会自动克隆代码，并通过 TiUP 部署测试集群。

4.3.5.2 第 2 步：获取代码

```
git clone https://github.com/pingcap-inc/tidb-example-python.git
```

SQLAlchemy 为当前比较流行的开源 Python ORM 之一。此处将以 SQLAlchemy 1.4.44 版本进行说明。

```python
import uuid
from typing import List

from sqlalchemy import create_engine, String, Column, Integer, select, func
from sqlalchemy.orm import declarative_base, sessionmaker

engine = create_engine('mysql://root:@127.0.0.1:4000/test')
Base = declarative_base()
Base.metadata.create_all(engine)
Session = sessionmaker(bind=engine)


class Player(Base):
    __tablename__ = "player"

    id = Column(String(36), primary_key=True)
    coins = Column(Integer)
    goods = Column(Integer)

    def __repr__(self):
        return f'Player(id={self.id!r}, coins={self.coins!r}, goods={self.goods!r})'


def random_player(amount: int) -> List[Player]:
    players = []
    for _ in range(amount):
        players.append(Player(id=uuid.uuid4(), coins=10000, goods=10000))

    return players


def simple_example() -> None:
    with Session() as session:
        # create a player, who has a coin and a goods.
        session.add(Player(id="test", coins=1, goods=1))

        # get this player, and print it.
        get_test_stmt = select(Player).where(Player.id == "test")
        for player in session.scalars(get_test_stmt):
            print(player)

        # create players with bulk inserts.
        # insert 1919 players totally, with 114 players per batch.
        # each player has a random UUID
```

```
        player_list = random_player(1919)
        for idx in range(0, len(player_list), 114):
            session.bulk_save_objects(player_list[idx:idx + 114])

        # print the number of players
        count = session.query(func.count(Player.id)).scalar()
        print(f'number of players: {count}')

        # print 3 players.
        three_players = session.query(Player).limit(3).all()
        for player in three_players:
            print(player)

        session.commit()


def trade_check(session: Session, sell_id: str, buy_id: str, amount: int, price: int) -> bool:
    # sell player goods check
    sell_player = session.query(Player.goods).filter(Player.id == sell_id).with_for_update().one
        ↪ ()
    if sell_player.goods < amount:
        print(f'sell player {sell_id} goods not enough')
        return False

    # buy player coins check
    buy_player = session.query(Player.coins).filter(Player.id == buy_id).with_for_update().one()
    if buy_player.coins < price:
        print(f'buy player {buy_id} coins not enough')
        return False


def trade(sell_id: str, buy_id: str, amount: int, price: int) -> None:
    with Session() as session:
        if trade_check(session, sell_id, buy_id, amount, price) is False:
            return

        # deduct the goods of seller, and raise his/her the coins
        session.query(Player).filter(Player.id == sell_id). \
            update({'goods': Player.goods - amount, 'coins': Player.coins + price})
        # deduct the coins of buyer, and raise his/her the goods
        session.query(Player).filter(Player.id == buy_id). \
            update({'goods': Player.goods + amount, 'coins': Player.coins - price})

        session.commit()
        print("trade success")
```

```
def trade_example() -> None:
    with Session() as session:
        # create two players
        # player 1: id is "1", has only 100 coins.
        # player 2: id is "2", has 114514 coins, and 20 goods.
        session.add(Player(id="1", coins=100, goods=0))
        session.add(Player(id="2", coins=114514, goods=20))
        session.commit()

    # player 1 wants to buy 10 goods from player 2.
    # it will cost 500 coins, but player 1 cannot afford it.
    # so this trade will fail, and nobody will lose their coins or goods
    trade(sell_id="2", buy_id="1", amount=10, price=500)

    # then player 1 has to reduce the incoming quantity to 2.
    # this trade will be successful
    trade(sell_id="2", buy_id="1", amount=2, price=100)

    with Session() as session:
        traders = session.query(Player).filter(Player.id.in_(("1", "2"))).all()
        for player in traders:
            print(player)
        session.commit()


simple_example()
trade_example()
```

相较于直接使用 Driver，SQLAlchemy 屏蔽了创建数据库连接时，不同数据库差异的细节。SQLAlchemy 还封装了大量的操作，如会话管理、基本对象的 CRUD 等，极大地简化了代码量。

`Player` 类为数据库表在程序内的映射。`Player` 的每个属性都对应着 `player` 表的一个字段。SQLAlchemy 使用 `Player` 类为了给 SQLAlchemy 提供更多的信息，使用了形如以上示例中的 `id = Column(String(36), primary_key=True)` 的类型定义，用来指示字段类型和其附加属性。`id = Column(String(36), primary_key=True)` 表示 `id` 字段为 `String` 类型，对应数据库类型为 `VARCHAR`，长度为 `36`，且为主键。

关于 SQLAlchemy 的更多使用方法，你可以参考 SQLAlchemy 官网。

peewee 为当前比较流行的开源 Python ORM 之一。此处将以 peewee 3.15.4 版本进行说明。

```
import os
import uuid
from typing import List

from peewee import *
```

```
from playhouse.db_url import connect

db = connect('mysql://root:@127.0.0.1:4000/test')


class Player(Model):
    id = CharField(max_length=36, primary_key=True)
    coins = IntegerField()
    goods = IntegerField()

    class Meta:
        database = db
        table_name = "player"


def random_player(amount: int) -> List[Player]:
    players = []
    for _ in range(amount):
        players.append(Player(id=uuid.uuid4(), coins=10000, goods=10000))

    return players


def simple_example() -> None:
    # create a player, who has a coin and a goods.
    Player.create(id="test", coins=1, goods=1)

    # get this player, and print it.
    test_player = Player.select().where(Player.id == "test").get()
    print(f'id:{test_player.id}, coins:{test_player.coins}, goods:{test_player.goods}')

    # create players with bulk inserts.
    # insert 1919 players totally, with 114 players per batch.
    # each player has a random UUID
    player_list = random_player(1919)
    Player.bulk_create(player_list, 114)

    # print the number of players
    count = Player.select().count()
    print(f'number of players: {count}')

    # print 3 players.
    three_players = Player.select().limit(3)
    for player in three_players:
        print(f'id:{player.id}, coins:{player.coins}, goods:{player.goods}')
```

```
def trade_check(sell_id: str, buy_id: str, amount: int, price: int) -> bool:
    sell_goods = Player.select(Player.goods).where(Player.id == sell_id).get().goods
    if sell_goods < amount:
        print(f'sell player {sell_id} goods not enough')
        return False

    buy_coins = Player.select(Player.coins).where(Player.id == buy_id).get().coins
    if buy_coins < price:
        print(f'buy player {buy_id} coins not enough')
        return False

    return True


def trade(sell_id: str, buy_id: str, amount: int, price: int) -> None:
    with db.atomic() as txn:
        try:
            if trade_check(sell_id, buy_id, amount, price) is False:
                txn.rollback()
                return

            # deduct the goods of seller, and raise his/her the coins
            Player.update(goods=Player.goods - amount, coins=Player.coins + price).where(Player.
                ↪ id == sell_id).execute()
            # deduct the coins of buyer, and raise his/her the goods
            Player.update(goods=Player.goods + amount, coins=Player.coins - price).where(Player.
                ↪ id == buy_id).execute()

        except Exception as err:
            txn.rollback()
            print(f'something went wrong: {err}')
        else:
            txn.commit()
            print("trade success")


def trade_example() -> None:
    # create two players
    # player 1: id is "1", has only 100 coins.
    # player 2: id is "2", has 114514 coins, and 20 goods.
    Player.create(id="1", coins=100, goods=0)
    Player.create(id="2", coins=114514, goods=20)
```

```
    # player 1 wants to buy 10 goods from player 2.
    # it will cost 500 coins, but player 1 cannot afford it.
    # so this trade will fail, and nobody will lose their coins or goods
    trade(sell_id="2", buy_id="1", amount=10, price=500)

    # then player 1 has to reduce the incoming quantity to 2.
    # this trade will be successful
    trade(sell_id="2", buy_id="1", amount=2, price=100)

    # let's take a look for player 1 and player 2 currently
    after_trade_players = Player.select().where(Player.id.in_(["1", "2"]))
    for player in after_trade_players:
        print(f'id:{player.id}, coins:{player.coins}, goods:{player.goods}')


db.connect()

### recreate the player table
db.drop_tables([Player])
db.create_tables([Player])

simple_example()
trade_example()
```

相较于直接使用 Driver，peewee 屏蔽了创建数据库连接时，不同数据库差异的细节。peewee 还封装了大量的操作，如会话管理、基本对象的 CRUD 等，极大地简化了代码量。

`Player` 类为数据库表在程序内的映射。`Player` 的每个属性都对应着 `player` 表的一个字段。peewee 使用 `Player` 类为了给 peewee 提供更多的信息，使用了形如以上示例中的 `id = CharField(max_length=36, primary_key=True)` 的类型定义，用来指示字段类型和其附加属性。`id = CharField(max_length=36, primary_key=True)` 表示 `id` 字段为 `CharField` 类型，对应数据库类型为 `VARCHAR`，长度为 `36`，且为主键。

关于 peewee 的更多使用方法，你可以参考 peewee 官网。

mysqlclient 为当前比较流行的开源 Python Driver 之一。此处将以 mysqlclient 2.1.1 版本进行说明。虽然 Python 的 Driver 相较其他语言，使用也极其方便。但因其不可屏蔽底层实现，需手动管控事务的特性，如果没有大量必须使用 SQL 的场景，仍然推荐使用 ORM 进行程序编写。这可以降低程序的耦合性。

```
import uuid
from typing import List

import MySQLdb
from MySQLdb import Connection
from MySQLdb.cursors import Cursor

def get_connection(autocommit: bool = True) -> MySQLdb.Connection:
    return MySQLdb.connect(
        host="127.0.0.1",
```

```
        port=4000,
        user="root",
        password="",
        database="test",
        autocommit=autocommit
    )


def create_player(cursor: Cursor, player: tuple) -> None:
    cursor.execute("INSERT INTO player (id, coins, goods) VALUES (%s, %s, %s)", player)


def get_player(cursor: Cursor, player_id: str) -> tuple:
    cursor.execute("SELECT id, coins, goods FROM player WHERE id = %s", (player_id,))
    return cursor.fetchone()


def get_players_with_limit(cursor: Cursor, limit: int) -> List[tuple]:
    cursor.execute("SELECT id, coins, goods FROM player LIMIT %s", (limit,))
    return cursor.fetchall()


def random_player(amount: int) -> List[tuple]:
    players = []
    for _ in range(amount):
        players.append((uuid.uuid4(), 10000, 10000))

    return players


def bulk_create_player(cursor: Cursor, players: List[tuple]) -> None:
    cursor.executemany("INSERT INTO player (id, coins, goods) VALUES (%s, %s, %s)", players)


def get_count(cursor: Cursor) -> None:
    cursor.execute("SELECT count(*) FROM player")
    return cursor.fetchone()[0]


def trade_check(cursor: Cursor, sell_id: str, buy_id: str, amount: int, price: int) -> bool:
    get_player_with_lock_sql = "SELECT coins, goods FROM player WHERE id = %s FOR UPDATE"

    # sell player goods check
    cursor.execute(get_player_with_lock_sql, (sell_id,))
    _, sell_goods = cursor.fetchone()
```

```
    if sell_goods < amount:
        print(f'sell player {sell_id} goods not enough')
        return False

    # buy player coins check
    cursor.execute(get_player_with_lock_sql, (buy_id,))
    buy_coins, _ = cursor.fetchone()
    if buy_coins < price:
        print(f'buy player {buy_id} coins not enough')
        return False


def trade_update(cursor: Cursor, sell_id: str, buy_id: str, amount: int, price: int) -> None:
    update_player_sql = "UPDATE player set goods = goods + %s, coins = coins + %s WHERE id = %s"

    # deduct the goods of seller, and raise his/her the coins
    cursor.execute(update_player_sql, (-amount, price, sell_id))
    # deduct the coins of buyer, and raise his/her the goods
    cursor.execute(update_player_sql, (amount, -price, buy_id))


def trade(connection: Connection, sell_id: str, buy_id: str, amount: int, price: int) -> None:
    with connection.cursor() as cursor:
        if trade_check(cursor, sell_id, buy_id, amount, price) is False:
            connection.rollback()
            return

        try:
            trade_update(cursor, sell_id, buy_id, amount, price)
        except Exception as err:
            connection.rollback()
            print(f'something went wrong: {err}')
        else:
            connection.commit()
            print("trade success")


def simple_example() -> None:
    with get_connection(autocommit=True) as conn:
        with conn.cursor() as cur:
            # create a player, who has a coin and a goods.
            create_player(cur, ("test", 1, 1))

            # get this player, and print it.
            test_player = get_player(cur, "test")
```

```
            print(f'id:{test_player[0]}, coins:{test_player[1]}, goods:{test_player[2]}')

            # create players with bulk inserts.
            # insert 1919 players totally, with 114 players per batch.
            # each player has a random UUID
            player_list = random_player(1919)
            for idx in range(0, len(player_list), 114):
                bulk_create_player(cur, player_list[idx:idx + 114])

            # print the number of players
            count = get_count(cur)
            print(f'number of players: {count}')

            # print 3 players.
            three_players = get_players_with_limit(cur, 3)
            for player in three_players:
                print(f'id:{player[0]}, coins:{player[1]}, goods:{player[2]}')


def trade_example() -> None:
    with get_connection(autocommit=False) as conn:
        with conn.cursor() as cur:
            # create two players
            # player 1: id is "1", has only 100 coins.
            # player 2: id is "2", has 114514 coins, and 20 goods.
            create_player(cur, ("1", 100, 0))
            create_player(cur, ("2", 114514, 20))
            conn.commit()

        # player 1 wants to buy 10 goods from player 2.
        # it will cost 500 coins, but player 1 cannot afford it.
        # so this trade will fail, and nobody will lose their coins or goods
        trade(conn, sell_id="2", buy_id="1", amount=10, price=500)

        # then player 1 has to reduce the incoming quantity to 2.
        # this trade will be successful
        trade(conn, sell_id="2", buy_id="1", amount=2, price=100)

        # let's take a look for player 1 and player 2 currently
        with conn.cursor() as cur:
            _, player1_coin, player1_goods = get_player(cur, "1")
            print(f'id:1, coins:{player1_coin}, goods:{player1_goods}')
            _, player2_coin, player2_goods = get_player(cur, "2")
            print(f'id:2, coins:{player2_coin}, goods:{player2_goods}')
```

```
simple_example()
trade_example()
```

Driver 有着更低的封装程度，因此我们可以在程序内见到大量的 SQL。程序内查询到的 `Player`，与 ORM 不同，因为没有数据对象的存在，`Player` 将以元组 (tuple) 进行表示。

关于 mysqlclient 的更多使用方法，你可以参考 mysqlclient 官方文档。

PyMySQL 为当前比较流行的开源 Python Driver 之一。此处将以 PyMySQL 1.0.2 版本进行说明。虽然 Python 的 Driver 相较其他语言，使用也极其方便。但因其不可屏蔽底层实现，需手动管控事务的特性，如果没有大量必须使用 SQL 的场景，仍然推荐使用 ORM 进行程序编写。这可以降低程序的耦合性。

```
import uuid
from typing import List

import pymysql.cursors
from pymysql import Connection
from pymysql.cursors import DictCursor


def get_connection(autocommit: bool = False) -> Connection:
    return pymysql.connect(host='127.0.0.1',
                           port=4000,
                           user='root',
                           password='',
                           database='test',
                           cursorclass=DictCursor,
                           autocommit=autocommit)


def create_player(cursor: DictCursor, player: tuple) -> None:
    cursor.execute("INSERT INTO player (id, coins, goods) VALUES (%s, %s, %s)", player)


def get_player(cursor: DictCursor, player_id: str) -> dict:
    cursor.execute("SELECT id, coins, goods FROM player WHERE id = %s", (player_id,))
    return cursor.fetchone()


def get_players_with_limit(cursor: DictCursor, limit: int) -> tuple:
    cursor.execute("SELECT id, coins, goods FROM player LIMIT %s", (limit,))
    return cursor.fetchall()


def random_player(amount: int) -> List[tuple]:
    players = []
```

```
    for _ in range(amount):
        players.append((uuid.uuid4(), 10000, 10000))

    return players


def bulk_create_player(cursor: DictCursor, players: List[tuple]) -> None:
    cursor.executemany("INSERT INTO player (id, coins, goods) VALUES (%s, %s, %s)", players)


def get_count(cursor: DictCursor) -> int:
    cursor.execute("SELECT count(*) as count FROM player")
    return cursor.fetchone()['count']


def trade_check(cursor: DictCursor, sell_id: str, buy_id: str, amount: int, price: int) -> bool:
    get_player_with_lock_sql = "SELECT coins, goods FROM player WHERE id = %s FOR UPDATE"

    # sell player goods check
    cursor.execute(get_player_with_lock_sql, (sell_id,))
    seller = cursor.fetchone()
    if seller['goods'] < amount:
        print(f'sell player {sell_id} goods not enough')
        return False

    # buy player coins check
    cursor.execute(get_player_with_lock_sql, (buy_id,))
    buyer = cursor.fetchone()
    if buyer['coins'] < price:
        print(f'buy player {buy_id} coins not enough')
        return False


def trade_update(cursor: DictCursor, sell_id: str, buy_id: str, amount: int, price: int) -> None:
    update_player_sql = "UPDATE player set goods = goods + %s, coins = coins + %s WHERE id = %s"

    # deduct the goods of seller, and raise his/her the coins
    cursor.execute(update_player_sql, (-amount, price, sell_id))
    # deduct the coins of buyer, and raise his/her the goods
    cursor.execute(update_player_sql, (amount, -price, buy_id))


def trade(connection: Connection, sell_id: str, buy_id: str, amount: int, price: int) -> None:
    with connection.cursor() as cursor:
        if trade_check(cursor, sell_id, buy_id, amount, price) is False:
```

```
            connection.rollback()
            return

        try:
            trade_update(cursor, sell_id, buy_id, amount, price)
        except Exception as err:
            connection.rollback()
            print(f'something went wrong: {err}')
        else:
            connection.commit()
            print("trade success")


def simple_example() -> None:
    with get_connection(autocommit=True) as connection:
        with connection.cursor() as cur:
            # create a player, who has a coin and a goods.
            create_player(cur, ("test", 1, 1))

            # get this player, and print it.
            test_player = get_player(cur, "test")
            print(test_player)

            # create players with bulk inserts.
            # insert 1919 players totally, with 114 players per batch.
            # each player has a random UUID
            player_list = random_player(1919)
            for idx in range(0, len(player_list), 114):
                bulk_create_player(cur, player_list[idx:idx + 114])

            # print the number of players
            count = get_count(cur)
            print(f'number of players: {count}')

            # print 3 players.
            three_players = get_players_with_limit(cur, 3)
            for player in three_players:
                print(player)


def trade_example() -> None:
    with get_connection(autocommit=False) as connection:
        with connection.cursor() as cur:
            # create two players
            # player 1: id is "1", has only 100 coins.
```

```
            # player 2: id is "2", has 114514 coins, and 20 goods.
            create_player(cur, ("1", 100, 0))
            create_player(cur, ("2", 114514, 20))
            connection.commit()

        # player 1 wants to buy 10 goods from player 2.
        # it will cost 500 coins, but player 1 cannot afford it.
        # so this trade will fail, and nobody will lose their coins or goods
        trade(connection, sell_id="2", buy_id="1", amount=10, price=500)

        # then player 1 has to reduce the incoming quantity to 2.
        # this trade will be successful
        trade(connection, sell_id="2", buy_id="1", amount=2, price=100)

        # let's take a look for player 1 and player 2 currently
        with connection.cursor() as cur:
            print(get_player(cur, "1"))
            print(get_player(cur, "2"))


simple_example()
trade_example()
```

Driver 有着更低的封装程度，因此我们可以在程序内见到大量的 SQL。程序内查询到的 Player，与 ORM 不同，因为没有数据对象的存在，Player 将以 dict 进行表示。

关于 PyMySQL 的更多使用方法，你可以参考 PyMySQL 官方文档。

mysql-connector-python 为当前比较流行的开源 Python Driver 之一。此处将以 mysql-connector-python 8.0.31 版本进行说明。虽然 Python 的 Driver 相较其他语言，使用也极其方便。但因其不可屏蔽底层实现，需手动管控事务的特性，如果没有大量必须使用 SQL 的场景，仍然推荐使用 ORM 进行程序编写。这可以降低程序的耦合性。

```
import uuid
from typing import List

from mysql.connector import connect, MySQLConnection
from mysql.connector.cursor import MySQLCursor


def get_connection(autocommit: bool = True) -> MySQLConnection:
    connection = connect(host='127.0.0.1',
                         port=4000,
                         user='root',
                         password='',
                         database='test')
    connection.autocommit = autocommit
    return connection
```

```python
def create_player(cursor: MySQLCursor, player: tuple) -> None:
    cursor.execute("INSERT INTO player (id, coins, goods) VALUES (%s, %s, %s)", player)


def get_player(cursor: MySQLCursor, player_id: str) -> tuple:
    cursor.execute("SELECT id, coins, goods FROM player WHERE id = %s", (player_id,))
    return cursor.fetchone()


def get_players_with_limit(cursor: MySQLCursor, limit: int) -> List[tuple]:
    cursor.execute("SELECT id, coins, goods FROM player LIMIT %s", (limit,))
    return cursor.fetchall()


def random_player(amount: int) -> List[tuple]:
    players = []
    for _ in range(amount):
        players.append((str(uuid.uuid4()), 10000, 10000))

    return players


def bulk_create_player(cursor: MySQLCursor, players: List[tuple]) -> None:
    cursor.executemany("INSERT INTO player (id, coins, goods) VALUES (%s, %s, %s)", players)


def get_count(cursor: MySQLCursor) -> int:
    cursor.execute("SELECT count(*) FROM player")
    return cursor.fetchone()[0]


def trade_check(cursor: MySQLCursor, sell_id: str, buy_id: str, amount: int, price: int) -> bool:
    get_player_with_lock_sql = "SELECT coins, goods FROM player WHERE id = %s FOR UPDATE"

    # sell player goods check
    cursor.execute(get_player_with_lock_sql, (sell_id,))
    _, sell_goods = cursor.fetchone()
    if sell_goods < amount:
        print(f'sell player {sell_id} goods not enough')
        return False

    # buy player coins check
    cursor.execute(get_player_with_lock_sql, (buy_id,))
```

```
    buy_coins, _ = cursor.fetchone()
    if buy_coins < price:
        print(f'buy player {buy_id} coins not enough')
        return False


def trade_update(cursor: MySQLCursor, sell_id: str, buy_id: str, amount: int, price: int) -> None
    ↪ :
    update_player_sql = "UPDATE player set goods = goods + %s, coins = coins + %s WHERE id = %s"

    # deduct the goods of seller, and raise his/her the coins
    cursor.execute(update_player_sql, (-amount, price, sell_id))
    # deduct the coins of buyer, and raise his/her the goods
    cursor.execute(update_player_sql, (amount, -price, buy_id))


def trade(connection: MySQLConnection, sell_id: str, buy_id: str, amount: int, price: int) ->
    ↪ None:
    with connection.cursor() as cursor:
        if trade_check(cursor, sell_id, buy_id, amount, price) is False:
            connection.rollback()
            return

        try:
            trade_update(cursor, sell_id, buy_id, amount, price)
        except Exception as err:
            connection.rollback()
            print(f'something went wrong: {err}')
        else:
            connection.commit()
            print("trade success")


def simple_example() -> None:
    with get_connection(autocommit=True) as connection:
        with connection.cursor() as cur:
            # create a player, who has a coin and a goods.
            create_player(cur, ("test", 1, 1))

            # get this player, and print it.
            test_player = get_player(cur, "test")
            print(f'id:{test_player[0]}, coins:{test_player[1]}, goods:{test_player[2]}')

            # create players with bulk inserts.
            # insert 1919 players totally, with 114 players per batch.
```

```
            # all players have random uuid
            print(f'start to insert one by one, it will take a long time')
            player_list = random_player(1919)
            for idx in range(0, len(player_list), 114):
                print(f'inserted {idx} players')
                bulk_create_player(cur, player_list[idx:idx + 114])

            # print the number of players
            count = get_count(cur)
            print(f'number of players: {count}')

            # print 3 players.
            three_players = get_players_with_limit(cur, 3)
            for player in three_players:
                print(f'id:{player[0]}, coins:{player[1]}, goods:{player[2]}')


def trade_example() -> None:
    with get_connection(autocommit=False) as conn:
        with conn.cursor() as cur:
            # create two players
            # player 1: id is "1", has only 100 coins.
            # player 2: id is "2", has 114514 coins, and 20 goods.
            create_player(cur, ("1", 100, 0))
            create_player(cur, ("2", 114514, 20))
            conn.commit()

        # player 1 wants to buy 10 goods from player 2.
        # it will cost 500 coins, but player 1 cannot afford it.
        # so this trade will fail, and nobody will lose their coins or goods
        trade(conn, sell_id="2", buy_id="1", amount=10, price=500)

        # then player 1 has to reduce the incoming quantity to 2.
        # this trade will be successful
        trade(conn, sell_id="2", buy_id="1", amount=2, price=100)

        # let's take a look for player 1 and player 2 currently
        with conn.cursor() as cur:
            _, player1_coin, player1_goods = get_player(cur, "1")
            print(f'id:1, coins:{player1_coin}, goods:{player1_goods}')
            _, player2_coin, player2_goods = get_player(cur, "2")
            print(f'id:2, coins:{player2_coin}, goods:{player2_goods}')


simple_example()
```

```
trade_example()
```

Driver 有着更低的封装程度，因此我们可以在程序内见到大量的 SQL。程序内查询到的 `Player`，与 ORM 不同，因为没有数据对象的存在，`Player` 将以 tuple 进行表示。

关于 mysql-connector-python 的更多使用方法，你可以参考 mysql-connector-python 官方文档。

4.3.5.3 第 3 步：运行代码

本节将逐步介绍代码的运行方法。

4.3.5.3.1 第 3 步第 1 部分：表初始化

> **建议：**
>
> 在 Gitpod Playground 中尝试 Python 与 TiDB 的连接：现在就试试

本示例需手动初始化表，若你使用本地集群，可直接运行：

```
mysql --host 127.0.0.1 --port 4000 -u root < player_init.sql
```

```
mycli --host 127.0.0.1 --port 4000 -u root --no-warn < player_init.sql
```

若不使用本地集群，或未安装命令行客户端，请用喜欢的方式（如 Navicat、DBeaver 等 GUI 工具）直接登录集群，并运行 `player_init.sql` 文件内的 SQL 语句。

4.3.5.3.2 第 3 步第 2 部分：TiDB Cloud 更改参数

若你使用了 TiDB Serverless 集群，此处需使用系统本地的 CA 证书，并将证书路径记为 `<ca_path>` 以供后续指代。请参考以下系统相关的证书路径地址：

```
/etc/ssl/cert.pem
```

```
/etc/ssl/certs/ca-certificates.crt
```

```
/etc/pki/tls/certs/ca-bundle.crt
```

```
/etc/ssl/ca-bundle.pem
```

若设置后仍有证书错误，请查阅 TiDB Serverless 安全连接文档。

若你使用 TiDB Serverless 集群，更改 `sqlalchemy_example.py` 内 `create_engine` 函数的入参：

```
engine = create_engine('mysql://root:@127.0.0.1:4000/test')
```

若你设定的密码为 `123456`，而且从 TiDB Serverless 集群面板中得到的连接信息为：

- Endpoint: `xxx.tidbcloud.com`

- Port: 4000
- User: 2aEp24QWEDLqRFs.root

那么此处应将 `create_engine` 更改为：

```
engine = create_engine('mysql://2aEp24QWEDLqRFs.root:123456@xxx.tidbcloud.com:4000/test',
    ↪ connect_args={
    "ssl_mode": "VERIFY_IDENTITY",
    "ssl": {
        "ca": "<ca_path>"
    }
})
```

若你使用 TiDB Serverless 集群，更改 `peewee_example.py` 内 `connect` 函数的入参：

```
db = connect('mysql://root:@127.0.0.1:4000/test')
```

若你设定的密码为 123456，而且从 TiDB Serverless 集群面板中得到的连接信息为：

- Endpoint: xxx.tidbcloud.com
- Port: 4000
- User: 2aEp24QWEDLqRFs.root

那么此处应将 `connect` 更改为：

- peewee 将 PyMySQL 作为 Driver 时：

```
db = connect('mysql://2aEp24QWEDLqRFs.root:123456@xxx.tidbcloud.com:4000/test',
    ssl_verify_cert=True, ssl_ca="<ca_path>")
```

- peewee 将 mysqlclient 作为 Driver 时：

```
db = connect('mysql://2aEp24QWEDLqRFs.root:123456@xxx.tidbcloud.com:4000/test',
    ssl_mode="VERIFY_IDENTITY", ssl={"ca": "<ca_path>"})
```

由于 peewee 会将参数透传至 Driver 中，使用 peewee 时请注意 Driver 的使用类型。

若你使用 TiDB Serverless 集群，更改 `mysqlclient_example.py` 内 `get_connection` 函数：

```
def get_connection(autocommit: bool = True) -> MySQLdb.Connection:
    return MySQLdb.connect(
        host="127.0.0.1",
        port=4000,
        user="root",
        password="",
        database="test",
        autocommit=autocommit
    )
```

若你设定的密码为 123456，而且从 TiDB Serverless 集群面板中得到的连接信息为：

- Endpoint: xxx.tidbcloud.com
- Port: 4000
- User: 2aEp24QWEDLqRFs.root

那么此处应将 get_connection 更改为：

```
def get_connection(autocommit: bool = True) -> MySQLdb.Connection:
    return MySQLdb.connect(
        host="xxx.tidbcloud.com",
        port=4000,
        user="2aEp24QWEDLqRFs.root",
        password="123456",
        database="test",
        autocommit=autocommit,
        ssl_mode="VERIFY_IDENTITY",
        ssl={
            "ca": "<ca_path>"
        }
    )
```

若你使用 TiDB Serverless 集群，更改 pymysql_example.py 内 get_connection 函数：

```
def get_connection(autocommit: bool = False) -> Connection:
    return pymysql.connect(host='127.0.0.1',
                           port=4000,
                           user='root',
                           password='',
                           database='test',
                           cursorclass=DictCursor,
                           autocommit=autocommit)
```

若你设定的密码为 123456，而且从 TiDB Serverless 集群面板中得到的连接信息为：

- Endpoint: xxx.tidbcloud.com
- Port: 4000
- User: 2aEp24QWEDLqRFs.root

那么此处应将 get_connection 更改为：

```
def get_connection(autocommit: bool = False) -> Connection:
    return pymysql.connect(host='xxx.tidbcloud.com',
                           port=4000,
                           user='2aEp24QWEDLqRFs.root',
                           password='123546',
                           database='test',
```

```
                            cursorclass=DictCursor,
                            autocommit=autocommit,
                            ssl_ca='<ca_path>',
                            ssl_verify_cert=True,
                            ssl_verify_identity=True)
```

若你使用 TiDB Serverless 集群，更改 `mysql_connector_python_example.py` 内 `get_connection` 函数：

```
def get_connection(autocommit: bool = True) -> MySQLConnection:
    connection = connect(host='127.0.0.1',
                         port=4000,
                         user='root',
                         password='',
                         database='test')
    connection.autocommit = autocommit
    return connection
```

若你设定的密码为 123456，而且从 TiDB Serverless 集群面板中得到的连接信息为：

- Endpoint: `xxx.tidbcloud.com`
- Port: 4000
- User: `2aEp24QWEDLqRFs.root`

那么此处应将 `get_connection` 更改为：

```
def get_connection(autocommit: bool = True) -> MySQLConnection:
    connection = connect(
        host="xxx.tidbcloud.com",
        port=4000,
        user="2aEp24QWEDLqRFs.root",
        password="123456",
        database="test",
        autocommit=autocommit,
        ssl_ca='<ca_path>',
        ssl_verify_identity=True
    )
    connection.autocommit = autocommit
    return connection
```

4.3.5.3.3 第 3 步第 3 部分：运行

运行前请先安装依赖：

```
pip3 install -r requirement.txt
```

当以后需要多次运行脚本时，请在每次运行前先依照表初始化一节再次进行表初始化。

```
python3 sqlalchemy_example.py
```

```
python3 peewee_example.py
```

```
python3 mysqlclient_example.py
```

```
python3 pymysql_example.py
```

```
python3 mysql_connector_python_example.py
```

4.3.5.4 第 4 步：预期输出

SQLAlchemy 预期输出

peewee 预期输出

mysqlclient 预期输出

PyMySQL 预期输出

mysql-connector-python 预期输出

4.4 连接到 TiDB

4.4.1 选择驱动或 ORM 框架

> **注意：**
>
> TiDB 支持等级说明：
>
> - Full：表明 TiDB 已经兼容该工具的绝大多数功能，并且在该工具的新版本中对其保持兼容。PingCAP 将定期地对TiDB 支持的第三方工具中的新版本进行兼容性测试。
> - Compatible：表明由于该工具已适配 MySQL，而 TiDB 高度兼容 MySQL 协议，因此 TiDB 可以兼容该工具的大部分功能。但 PingCAP 并未对该工具作出完整的兼容性验证，有可能出现一些意外的行为。
>
> 关于更多 TiDB 支持的第三方工具，你可以查看TiDB 支持的第三方工具。

TiDB 兼容 MySQL 的协议，但存在部分与 MySQL 不兼容或有差异的特性，具体信息可查看与 MySQL 兼容性对比。

4.4.1.1 Java

本节介绍 Java 语言的 Driver 及 ORM 的使用方式。

4.4.1.1.1 Java Drivers

支持等级：Full

按照 MySQL 文档中的说明下载并配置 Java JDBC 驱动程序即可使用。对于 TiDB v6.3.0 及以上版本，建议使用 MySQL Connector/J 8.0.33 及以上版本。

> **建议：**
>
> 在 8.0.32 之前的 MySQL Connector/J 8.0 版本中存在一个 bug，当与 TiDB v6.3.0 之前的版本一起使用时，可能会导致线程卡死。为了避免此问题，建议使用 MySQL Connector/J 8.0.32 或更高版本，或者使用 TiDB JDBC（见 TiDB-JDBC 标签）。

有关一个完整的实例应用程序，可参阅使用TiDB 和 JDBC 构建一个 Java 应用。

支持等级：Full

TiDB-JDBC 是基于 MySQL 8.0.29 的定制版本。TiDB-JDBC 基于 MySQL 官方 8.0.29 版本编译，修复了原 JDBC 在 prepare 模式下多参数、多字段 EOF 的错误，并新增 TiCDC snapshot 自动维护和 SM3 认证插件等功能。

基于 SM3 的认证仅在 TiDB 版本的 MySQL Connector/J 中支持。

如果你使用的是 Maven，请将以下内容添加到你的 <dependencies></dependencies>：

```
<dependency>
  <groupId>io.github.lastincisor</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.29-tidb-1.0.0</version>
</dependency>
```

如果你需要使用 SM3 认证，请将以下内容添加到你的 <dependencies></dependencies>：

```
<dependency>
  <groupId>io.github.lastincisor</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.29-tidb-1.0.0</version>
</dependency>
<dependency>
    <groupId>org.bouncycastle</groupId>
    <artifactId>bcprov-jdk15on</artifactId>
    <version>1.67</version>
</dependency>
<dependency>
    <groupId>org.bouncycastle</groupId>
    <artifactId>bcpkix-jdk15on</artifactId>
    <version>1.67</version>
</dependency>
```

如果你使用的是 Gradle，请将以下内容添加到你的 dependencies：

```
implementation group: 'io.github.lastincisor', name: 'mysql-connector-java', version: '8.0.29-
    ↪ tidb-1.0.0'
implementation group: 'org.bouncycastle', name: 'bcprov-jdk15on', version: '1.67'
implementation group: 'org.bouncycastle', name: 'bcpkix-jdk15on', version: '1.67'
```

4.4.1.1.2 Java ORM 框架

> 注意：
>
> - Hibernate 当前不支持嵌套事务。
> - TiDB 从 v6.2.0 版本开始支持Savepoint。如需在 `@Transactional` 中使用 `Propagation.NESTED` 事务传播选项，即 `@Transactional(propagation = Propagation.NESTED)`，请确认你的 TiDB 版本为 v6.2.0 或以上。

支持等级：Full

你可以使用 Gradle 或 Maven 获取你的应用程序的所有依赖项，且会帮你下载依赖项的间接依赖，而无需你手动管理复杂的依赖关系。注意，只有 Hibernate `6.0.0.Beta2` 及以上版本才支持 TiDB 方言。

如果你使用的是 Maven，请将以下内容添加到你的 `<dependencies></dependencies>`：

```
<dependency>
    <groupId>org.hibernate.orm</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>6.0.0.CR2</version>
</dependency>

<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.49</version>
</dependency>
```

如果你使用的是 `Gradle`，请将以下内容添加到你的 dependencies：

```
implementation 'org.hibernate:hibernate-core:6.0.0.CR2'
implementation 'mysql:mysql-connector-java:5.1.49'
```

- 有关原生 Java 使用 Hibernate 进行 TiDB 应用程序构建的例子，可参阅TiDB 和 Java 的简单 CRUD 应用程序 - 使用 Hibernate。
- 有关 Spring 使用 Spring Data JPA、Hibernate 进行 TiDB 应用程序构建的例子，可参阅使用 Spring Boot 构建 TiDB 应用程序。

额外的，你需要在 Hibernate 配置文件中指定 TiDB 方言 `org.hibernate.dialect.TiDBDialect`，此方言在 Hibernate `6.0.0.Beta2` 以上才可支持。若你无法升级 Hibernate 版本，那么请你直接使用 MySQL 5.7 的方言 `org.hibernate.dialect.MySQL57Dialect`。但这可能造成不可预料的使用结果，及部分 TiDB 特有特性的缺失，如：序列等。

支持等级：Full

你可以使用 Gradle 或 Maven 获取应用程序的所有依赖项包括间接依赖，无需手动管理复杂的依赖关系。

如果你使用的是 Maven，请将以下内容添加到你的 `<dependencies></dependencies>`：

```
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>3.5.9</version>
</dependency>

<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.49</version>
</dependency>
```

如果你使用的是 Gradle，请将以下内容添加到你的 dependencies：

```
implementation 'org.mybatis:mybatis:3.5.9'
implementation 'mysql:mysql-connector-java:5.1.49'
```

使用 MyBatis 进行 TiDB 应用程序构建的例子，可参阅TiDB 和 Java 的简单 CRUD 应用程序 - 使用 Mybatis。

4.4.1.1.3 Java 客户端负载均衡

tidb-loadbalance

支持等级：Full

tidb-loadbalance 是应用端的负载均衡组件。通过 tidb-loadbalance，你可以实现自动维护 TiDB server 的节点信息，根据节点信息使用 tidb-loadbalance 策略在客户端分发 JDBC 连接。客户端应用与 TiDB server 之间使用 JDBC 直连，性能高于使用负载均衡组件。

目前 tidb-loadbalance 已实现轮询、随机、权重等负载均衡策略。

> 注意：
>
> tidb-loadbalance 需配合 mysql-connector-j 一起使用。

如果你使用的是 Maven，请将以下内容添加到你的 `<dependencies></dependencies>`：

```
<dependency>
  <groupId>io.github.lastincisor</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.29-tidb-1.0.0</version>
</dependency>
<dependency>
  <groupId>io.github.lastincisor</groupId>
  <artifactId>tidb-loadbalance</artifactId>
  <version>0.0.5</version>
</dependency>
```

如果你使用的是 Gradle，请将以下内容添加到你的 dependencies：

```
implementation group: 'io.github.lastincisor', name: 'mysql-connector-java', version: '8.0.29-
    ↪ tidb-1.0.0'
implementation group: 'io.github.lastincisor', name: 'tidb-loadbalance', version: '0.0.5'
```

4.4.1.2 Golang

本节介绍 Golang 语言的 Driver 及 ORM 的使用方式。

4.4.1.2.1 Golang Drivers

go-sql-driver/mysql

支持等级：Full

按照 go-sql-driver/mysql 文档中的说明获取并配置 Golang 驱动程序即可使用。

有关一个完整的实例应用程序，可参阅使用TiDB 和 go-sql-driver/mysql 构建一个 Golang 应用。

4.4.1.2.2 Golang ORM 框架

GORM

支持等级：Full

GORM 是一个流行的 Golang 的 ORM 框架，你可以使用 go get 获取你的应用程序的所有依赖项。

```
go get -u gorm.io/gorm
go get -u gorm.io/driver/mysql
```

使用 GORM 进行 TiDB 应用程序构建的例子，可参阅TiDB 和 Golang 的简单 CRUD 应用程序 - 使用 GORM。

4.4.1.3 Python

本节介绍 Python 语言的 Driver 及 ORM 的使用方式。

4.4.1.3.1 Python Drivers

支持等级：Compatible

按照 PyMySQL 文档中的说明下载并配置驱动程序即可使用。建议使用 1.0.2 及以上版本。

使用 PyMySQL 构建 TiDB 应用程序的例子，可参阅TiDB 和 Python 的简单 CRUD 应用程序 - 使用 PyMySQL。

支持等级：Compatible

按照 mysqlclient 文档中的说明下载并配置驱动程序即可使用。建议使用 2.1.1 及以上版本。

使用 mysqlclient 构建 TiDB 应用程序的例子，可参阅TiDB 和 Python 的简单 CRUD 应用程序 - 使用 mysqlclient。

支持等级：Compatible

按照 mysql-connector-python 文档中的说明下载并配置驱动程序即可使用。建议使用 8.0.31 及以上版本。

使用 mysql-connector-python 构建 TiDB 应用程序的例子，可参阅TiDB 和 Python 的简单 CRUD 应用程序 - 使用 mysql-connector-python。

4.4.1.3.2 Python ORM 框架

支持等级：Full

Django 是一个流行的 Python 的开发框架，你可以使用 `pip install Django==3.2.16 django-tidb>=3.0.0` 获取你的应用程序的所有依赖项。建议使用 Django 3.2.16 及以上版本。

使用 Django 构建 TiDB 应用程序的例子，可参阅使用 Django 构建 TiDB 应用程序。

支持等级：Full

SQLAlchemy 是一个流行的 Python 的 ORM 框架，你可以使用 `pip install SQLAlchemy==1.4.44` 获取你的应用程序的所有依赖项。建议使用 1.4.44 及以上版本。

使用 SQLAlchemy 构建 TiDB 应用程序的例子，可参阅TiDB 和 Python 的简单 CRUD 应用程序 - 使用 SQLAlchemy。

支持等级：Compatible

peewee 是一个流行的 Python 的 ORM 框架，你可以使用 `pip install peewee==3.15.4` 获取你的应用程序的所有依赖项。建议使用 3.15.4 及以上版本。

使用 peewee 构建 TiDB 应用程序的例子，可参阅TiDB 和 Python 的简单 CRUD 应用程序 - 使用 peewee。

4.4.2 连接到 TiDB

TiDB 高度兼容 MySQL 协议，全量的客户端连接参数列表，请参阅 MySQL Client Options。

TiDB 支持 MySQL 客户端/服务器协议。这使得大多数客户端驱动程序和 ORM 框架可以像连接到 MySQL 一样地连接到 TiDB。

4.4.2.1 MySQL

你可以选择使用 MySQL Client 或 MySQL Shell 连接到 TiDB。

你可以使用 MySQL Client 作为 TiDB 的命令行工具连接到 TiDB。下面以基于 YUM 的 Linux 发行版为例，介绍如何安装 MySQL Client。

```
sudo yum install mysql
```

安装完成后，你可以使用如下命令连接到 TiDB：

```
mysql --host <tidb_server_host> --port 4000 -u root -p --comments
```

你可以使用 MySQL Shell 作为 TiDB 的命令行工具连接到 TiDB。参考 MySQL Shell 文档进行安装。安装完成后，你可以使用如下命令连接到 TiDB：

```
mysqlsh --sql mysql://root@<tidb_server_host>:4000
```

4.4.2.2 JDBC

你可以使用 JDBC 驱动连接到 TiDB，这需要创建一个 `MysqlDataSource` 或 `MysqlConnectionPoolDataSource` 对象 (它们都实现了 `DataSource` 接口)，并使用 `setURL` 函数设置连接字符串。

例如：

```
MysqlDataSource mysqlDataSource = new MysqlDataSource();
mysqlDataSource.setURL("jdbc:mysql://{host}:{port}/{database}?user={username}&password={password}
    ↪ ");
```

有关 JDBC 连接的更多信息，可参考 JDBC 官方文档。

连接参数

参数名	描述
{username}	需要连接到 TiDB 集群的SQL 用户
{password}	需要连接到 TiDB 集群的 SQL 用户的密码
{host}	TiDB 节点运行的 Host
{port}	TiDB 节点正在监听的端口
{database}	(已经存在的) 数据库的名称

4.4.2.3 Hibernate

你可以使用 Hibernate ORM 连接到 TiDB，请将 Hibernate 的配置中的 `hibernate.connection.url` 设置为合法的 TiDB 连接字符串。

例如，你的配置被写在 `hibernate.cfg.xml` 文件中，那么你的配置文件应该为：

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property>
```

```
        <property name="hibernate.dialect">org.hibernate.dialect.TiDBDialect</property>
        <property name="hibernate.connection.url">jdbc:mysql://{host}:{port}/{database}?user={
            ↪ user}&amp;password={password}</property>
    </session-factory>
</hibernate-configuration>
```

随后，使用代码读取配置文件，从而获得 SessionFactory 对象：

```
SessionFactory sessionFactory = new Configuration().configure("hibernate.cfg.xml").
    ↪ buildSessionFactory();
```

这里有几个需要注意的点：

1. 因为使用的配置文件 hibernate.cfg.xml 为 XML 格式，而 & 字符，在 XML 中属于特殊字符，因此，需将 & 更改为 &。即，连接字符串 hibernate.connection.url 由 jdbc:mysql://{host}:{port}/{database}?user={user}&password={password} 改为了 jdbc:mysql://{host}:{port}/{database}?user={user}&password={password}。
2. 在你使用 Hibernate 时，建议使用 TiDB 方言，即 hibernate.dialect 设置为 org.hibernate.dialect.TiDBDialect。
3. Hibernate 在版本 6.0.0.Beta2 及以上可支持 TiDB 方言，因此推荐使用 6.0.0.Beta2 及以上版本的 Hibernate。

更多有关 Hibernate 连接参数的信息，请参阅 Hibernate 官方文档。

连接参数

参数名	描述
{username}	需要连接到 TiDB 集群的SQL 用户
{password}	需要连接到 TiDB 集群的 SQL 用户的密码
{host}	TiDB 节点运行的 Host
{port}	TiDB 节点正在监听的端口
{database}	(已经存在的) 数据库的名称

4.4.3 连接池与连接参数

- 连接池参数 - 连接数配置、探活配置两节摘自开发 Java 应用使用 TiDB 的最佳实践 - 连接池。
- 连接参数摘自开发 Java 应用使用 TiDB 的最佳实践 - JDBC。

4.4.3.1 连接池参数

TiDB (MySQL) 连接建立是比较昂贵的操作（至少对于 OLTP 来讲），除了建立 TCP 连接外还需要进行连接鉴权操作，所以客户端通常会把 TiDB (MySQL) 连接保存到连接池中进行复用。

Java 的连接池实现很多 (HikariCP, tomcat-jdbc, druid, c3p0, dbcp)，TiDB 不会限定使用的连接池，应用可以根据业务特点自行选择连接池实现。

4.4.3.1.1 连接数配置

比较常见的是应用需要根据自身情况配置合适的连接池大小，以 HikariCP 为例：

maximumPoolSize：连接池最大连接数，配置过大会导致 TiDB 消耗资源维护无用连接，配置过小则会导致应用获取连接变慢，所以需根据应用自身特点配置合适的值，可参考这篇文章。

minimumIdle：连接池最小空闲连接数，主要用于在应用空闲时存留一些连接以应对突发请求，同样是需要根据业务情况进行配置。

应用在使用连接池时，需要注意连接使用完成后归还连接，推荐应用使用对应的连接池相关监控（如 metricRegistry），通过监控能及时定位连接池问题。

4.4.3.1.2 探活配置

连接池维护到 TiDB 的长连接，TiDB 默认不会主动关闭客户端连接（除非报错），但一般客户端到 TiDB 之间还会有 LVS 或 HAProxy 之类的网络代理，它们通常会在连接空闲一定时间后主动清理连接。除了注意代理的 idle 配置外，连接池还需要进行保活或探测连接。

如果常在 Java 应用中看到以下错误：

```
The last packet sent successfully to the server was 3600000 milliseconds ago. The driver has not
    ↪ received any packets from the server. com.mysql.jdbc.exceptions.jdbc4.
    ↪ CommunicationsException: Communications link failure
```

如果 `n milliseconds ago` 中的 n 如果是 0 或很小的值，则通常是执行的 SQL 导致 TiDB 异常退出引起的报错，推荐查看 TiDB stderr 日志；如果 n 是一个非常大的值（比如这里的 3600000），很可能是因为这个连接空闲太久然后被中间 proxy 关闭了，通常解决方式除了调大 proxy 的 idle 配置，还可以让连接池执行以下操作：

- 每次使用连接前检查连接是否可用。
- 使用单独线程定期检查连接是否可用。
- 定期发送 test query 保活连接。

不同的连接池实现可能会支持其中一种或多种方式，可以查看所使用的连接池文档来寻找对应配置。

4.4.3.1.3 经验公式

在 HikariCP 的 About Pool Sizing 一文中可以了解到，在完全不知道如何设置数据库连接池大小的时候，可以考虑以以下经验公式为起点，在此基础上，围绕该结果进行尝试，以得到最高性能的连接池大小。

该经验公式描述如下：

```
connections = ((core_count * 2) + effective_spindle_count)
```

解释一下参数含义：

- connections: 得出的连接数大小。

- core_count: CPU 核心数。
- effective_spindle_count: 直译为有效主轴数，实际上是说你有多少个硬盘（非 SSD），因为每个旋转的硬盘可以被称为是一个旋转轴。例如，你使用的是一个有 16 个磁盘组成的 RAID 阵列的服务器，那么 effective_spindle_count 应为 16。此处经验公式，实际上是衡量你的服务器可以管理多少 I/O 并发请求，因为 HDD 通常只能串行请求。

要特别说明的是，在这个经验公式的的下方，也看到了一处说明：

```
A formula which has held up pretty well across a lot of benchmarks for years is
that for optimal throughput the number of active connections should be somewhere
near ((core_count * 2) + effective_spindle_count). Core count should not include
HT threads, even if hyperthreading is enabled. Effective spindle count is zero if
the active data set is fully cached, and approaches the actual number of spindles
as the cache hit rate falls. ... There hasn't been any analysis so far regarding
how well the formula works with SSDs.
```

这个说明指出：

1. core_count 就是 _ 物理核心数 _，与你是否开启超线程无关。
2. 数据被全量缓存时，effective_spindle_count 应被设置为 0，随着命中率的下降，会更加接近实际的 HDD 个数。
3. 这里没有任何基于 SSD 的经验公式。

这里的说明让你在使用 SSD 时，需探求其他的经验公式。

可以参考 CockroachDB 对数据库连接池中的描述，推荐的连接数大小公式为：

```
connections = (number of cores * 4)
```

因此，你在使用 SSD 的情况下可以将连接数设置为 CPU 核心数 * 4。以此来达到初始的连接池最大连接数大小，并以此数据周围进行进一步的调整。

4.4.3.1.4 调整方向

可以看到，在上方的经验公式中得到的，是一个推荐的初始值，若需得到某台具体机器上的最佳值，需在推荐值周围，通过尝试，得到最佳值。

此最佳值的获取，会有一些基本规律，此处罗列如下：

1. 如果你的网络或存储延迟较大，请增大你的最大连接数，可以进行等待，从而让线程在被阻塞时，其他的线程可继续进行处理。
2. 如果你的服务器上部署了多个服务，并且每个服务拥有独立的连接池时，请关注它们的连接池的最大连接数总和。

4.4.3.2 连接参数

Java 应用尽管可以选择在不同的框架中封装，但在最底层一般会通过调用 JDBC 来与数据库服务器进行交互。对于 JDBC，需要关注的主要有：API 的使用选择和 API Implementer 的参数配置。

4.4.3.2.1 JDBC API

对于基本的 JDBC API 使用可以参考 JDBC 官方教程，本文主要强调几个比较重要的 API 选择。

4.4.3.2.2 使用 Prepare API

对于 OLTP 场景，程序发送给数据库的 SQL 语句在去除参数变化后都是可穷举的某几类，因此建议使用预处理语句 (Prepared Statements) 代替普通的文本执行，并复用预处理语句来直接执行，从而避免 TiDB 重复解析和生成 SQL 执行计划的开销。

目前多数上层框架都会调用 Prepare API 进行 SQL 执行，如果直接使用 JDBC API 进行开发，注意选择使用 Prepare API。

另外需要注意 MySQL Connector/J 实现中默认只会做客户端的语句预处理，会将 ? 在客户端替换后以文本形式发送到服务端，所以除了要使用 Prepare API，还需要在 JDBC 连接参数中配置 `useServerPrepStmts = true`，才能在 TiDB 服务器端进行语句预处理（下面参数配置章节有详细介绍）。

4.4.3.2.3 使用 Batch 批量插入更新

对于批量插入更新，如果插入记录较多，可以选择使用 addBatch/executeBatch API。通过 addBatch 的方式将多条 SQL 的插入更新记录先缓存在客户端，然后在 executeBatch 时一起发送到数据库服务器。

> 注意：
>
> 对于 MySQL Connector/J 实现，默认 Batch 只是将多次 addBatch 的 SQL 发送时机延迟到调用 executeBatch 的时候，但实际网络发送还是会一条条的发送，通常不会降低与数据库服务器的网络交互次数。
>
> 如果希望 Batch 网络发送，需要在 JDBC 连接参数中配置 `rewriteBatchedStatements = true`（下面参数配置章节有详细介绍）。

4.4.3.2.4 使用 StreamingResult 流式获取执行结果

一般情况下，为提升执行效率，JDBC 会默认提前获取查询结果并将其保存在客户端内存中。但在查询返回超大结果集的场景中，客户端会希望数据库服务器减少向客户端一次返回的记录数，等客户端在有限内存处理完一部分后再去向服务器要下一批。

在 JDBC 中通常有以下两种处理方式：

- 设置 FetchSize 为 `Integer.MIN_VALUE` 让客户端不缓存，客户端通过 StreamingResult 的方式从网络连接上流式读取执行结果。
- 使用 Cursor Fetch，首先需设置 FetchSize 为正整数，且在 JDBC URL 中配置 `useCursorFetch = true`。

TiDB 中同时支持两种方式，但更推荐使用第一种将 FetchSize 设置为 `Integer.MIN_VALUE` 的方式，比第二种功能实现更简单且执行效率更高。

4.4.3.2.5 MySQL JDBC 参数

JDBC 实现通常通过 JDBC URL 参数的形式来提供实现相关的配置。这里以 MySQL 官方的 Connector/J 来介绍参数配置（如果使用的是 MariaDB，可以参考 MariaDB 的类似配置）。因为配置项较多，这里主要关注几个可能影响到性能的参数。

Prepare 相关参数

- useServerPrepStmts

  默认情况下，useServerPrepStmts 的值为 `false`，即尽管使用了 Prepare API，也只会在客户端做“prepare”。因此为了避免服务器重复解析的开销，如果同一条 SQL 语句需要多次使用 Prepare API，则建议设置该选项为 `true`。

  在 TiDB 监控中可以通过 Query Summary > CPS By Instance 查看请求命令类型，如果请求中 COM_QUERY 被 COM_STMT_EXECUTE 或 COM_STMT_PREPARE 代替即生效。

- cachePrepStmts

  虽然 `useServerPrepStmts = true` 能让服务端执行预处理语句，但默认情况下客户端每次执行完后会 close 预处理语句，并不会复用，这样预处理的效率甚至不如文本执行。所以建议开启 `useServerPrepStmts = true` 后同时配置 `cachePrepStmts = true`，这会让客户端缓存预处理语句。

  在 TiDB 监控中可以通过 Query Summary > CPS By Instance 查看请求命令类型，如果请求中 COM_STMT_EXECUTE 数目远远多于 COM_STMT_PREPARE 即生效。

  另外，通过 `useConfigs = maxPerformance` 配置会同时配置多个参数，其中也包括 `cachePrepStmts = true`。

- prepStmtCacheSqlLimit

  在配置 cachePrepStmts 后还需要注意 prepStmtCacheSqlLimit 配置（默认为 256），该配置控制客户端缓存预处理语句的最大长度，超过该长度将不会被缓存。

  在一些场景 SQL 的长度可能超过该配置，导致预处理 SQL 不能复用，建议根据应用 SQL 长度情况决定是否需要调大该值。

  在 TiDB 监控中通过 Query Summary > CPS By Instance 查看请求命令类型，如果已经配置了 `cachePrepStmts = true`，但 COM_STMT_PREPARE 还是和 COM_STMT_EXECUTE 基本相等且有 COM_STMT_CLOSE，需要检查这个配置项是否设置得太小。

- prepStmtCacheSize

  控制缓存的预处理语句数目（默认为 25），如果应用需要预处理的 SQL 种类很多且希望复用预处理语句，可以调大该值。

  和上一条类似，在监控中通过 Query Summary > CPS By Instance 查看请求中 COM_STMT_EXECUTE 数目是否远远多于 COM_STMT_PREPARE 来确认是否正常。

Batch 相关参数

在进行 batch 写入处理时推荐配置 `rewriteBatchedStatements = true`，在已经使用 addBatch 或 executeBatch 后默认 JDBC 还是会一条条 SQL 发送，例如：

```
pstmt = prepare("INSERT INTO `t` (`a`) VALUES(?)");
pstmt.setInt(1, 10);
pstmt.addBatch();
pstmt.setInt(1, 11);
pstmt.addBatch();
pstmt.setInt(1, 12);
pstmt.executeBatch();
```

虽然使用了 batch 但发送到 TiDB 语句还是单独的多条 insert：

```
INSERT INTO `t` (`a`) VALUES(10);
INSERT INTO `t` (`a`) VALUES(11);
INSERT INTO `t` (`a`) VALUES(12);
```

如果设置 `rewriteBatchedStatements = true`，发送到 TiDB 的 SQL 将是：

```
INSERT INTO `t` (`a`) VALUES(10),(11),(12);
```

需要注意的是，insert 语句的改写，只能将多个 values 后的值拼接成一整条 SQL, insert 语句如果有其他差异将无法被改写。例如：

```
INSERT INTO `t` (`a`) VALUES (10) ON DUPLICATE KEY UPDATE `a` = 10;
INSERT INTO `t` (`a`) VALUES (11) ON DUPLICATE KEY UPDATE `a` = 11;
INSERT INTO `t` (`a`) VALUES (12) ON DUPLICATE KEY UPDATE `a` = 12;
```

上述 insert 语句将无法被改写成一条语句。该例子中，如果将 SQL 改写成如下形式：

```
INSERT INTO `t` (`a`) VALUES (10) ON DUPLICATE KEY UPDATE `a` = values(`a`);
INSERT INTO `t` (`a`) VALUES (11) ON DUPLICATE KEY UPDATE `a` = values(`a`);
INSERT INTO `t` (`a`) VALUES (12) ON DUPLICATE KEY UPDATE `a` = values(`a`);
```

即可满足改写条件，最终被改写成：

```
INSERT INTO `t` (`a`) VALUES (10), (11), (12) ON DUPLICATE KEY UPDATE `a` = values(`a`);
```

批量更新时如果有 3 处或 3 处以上更新，则 SQL 语句会改写为 multiple-queries 的形式并发送，这样可以有效减少客户端到服务器的请求开销，但副作用是会产生较大的 SQL 语句，例如这样：

```
UPDATE `t` SET `a` = 10 WHERE `id` = 1;
UPDATE `t` SET `a` = 11 WHERE `id` = 2;
UPDATE `t` SET `a` = 12 WHERE `id` = 3;
```

另外，因为一个客户端 bug，批量更新时如果要配置 `rewriteBatchedStatements = true` 和 `useServerPrepStmts = true`，推荐同时配置 `allowMultiQueries = true` 参数来避免这个 bug。

集成参数

通过监控可能会发现，虽然业务只向集群进行 insert 操作，却看到有很多多余的 select 语句。通常这是因为 JDBC 发送了一些查询设置类的 SQL 语句（例如 `select @@session.transaction_read_only`）。这些 SQL 对 TiDB 无用，推荐配置 `useConfigs = maxPerformance` 来避免额外开销。

useConfigs = maxPerformance 会包含一组配置，可查看 MySQL Connector/J 8.0 版本 或 5.1 版本 来确认当前 MySQL Connector/J 中 maxPerformance 包含的具体配置。

配置后查看监控，可以看到多余语句减少。

超时参数

TiDB 提供两个与 MySQL 兼容的超时控制参数，wait_timeout 和max_execution_time。这两个参数分别控制与 Java 应用连接的空闲超时时间和连接中 SQL 执行的超时时间，即控制 TiDB 与 Java 应用的连接最长闲多久和最长忙多久。在 TiDB v5.4 及以上版本中，wait_timeout 参数默认值为 28800 秒，即空闲超时为 8 小时。在 v5.4 之前，wait_timeout 参数的默认值为 0，即没有时间限制。max_execution_time 参数的默认值为 0，即不限制一条 SQL 语句的执行时间。

但在实际生产环境中，空闲连接和一直无限执行的 SQL 对数据库和应用都有不好的影响。你可以通过在应用的连接字符串中配置这两个参数来避免空闲连接和执行时间过长的 SQL 语句。例如，设置 sessionVariables=wait_timeout=3600（1 小时）和 sessionVariables=max_execution_time=300000（5 分钟）。

4.5 数据库模式设计

4.5.1 概述

本页概述了 TiDB 中的数据库模式。将从本页开始围绕Bookshop 这个应用程序来对 TiDB 的设计数据库部分展开介绍。并使用此数据库做后续数据的写入、读取示例。

4.5.1.1 术语歧义

此处术语会有歧义，为消除歧义，在此作出数据库模式设计文档部分中的术语简要约定：

为避免和通用术语数据库 (Database) 混淆，因此将逻辑对象称为数据库 (Database)，TiDB 仍使用原名称，并将 TiDB 的部署实例称为集群 (Cluster)。

因为 TiDB 使用与 MySQL 兼容的语法，在此语法下，模式 (Schema) 仅代表通用术语定义，并无逻辑对象定义，可参考此官方文档。若你从其他拥有 Schema 逻辑对象的数据库（如：PostgreSQL、Oracle、Microsoft SQL Server 等）迁移而来，请注意此区别。

4.5.1.2 数据库 Database

TiDB 语境中的 Database 或者说数据库，可以认为是表和索引等对象的集合。

TiDB 集群包含一个名为 test 的数据库。但建议你自行创建数据库，而不是使用 test 数据库。

4.5.1.3 表 Table

TiDB 语境中的 Table 或者说表，从属于某个数据库。

表包含数据行。每行数据中的每个值都属于一个特定的列。每列都只允许单一数据类型的数据值。列可添加约束来进一步限定。你还可以添加生成列（实验特性）用于计算。

4.5.1.4 索引 Index

索引是单个表中行的副本，按列或列集排序。TiDB 查询使用索引来更有效的查找表内的数据，同时可以给出特定列的值。每个索引都是从属于某个表的。

索引有两种常见的类型，分别为：

- Primary Key: 即主键索引，即标识在主键列上的索引。
- Secondary Index: 即二级索引，即在非主键上标识的索引。

> 注意：
>
> TiDB 中，关于 Primary Key 的默认定义与 MySQL 常用存储引擎 InnoDB 不一致。InnoDB 中，Primary Key 的语义为：唯一，不为空，且为聚簇索引。
>
> 而在 TiDB 中，Primary Key 的定义为：唯一，不为空。但主键不保证为聚簇索引。而是由另一组关键字 CLUSTERED、NONCLUSTERED 额外控制 Primary Key 是否为聚簇索引，若不指定，则由系统变量 `@@global.tidb_enable_clustered_index` 影响，具体说明请看聚簇索引。

4.5.1.4.1 专用索引

TiDB 支持一些特殊场景专用的索引，用以提高特定用例中的查询性能。具体请参考索引和约束。

4.5.1.5 其他对象

TiDB 支持一些和表同级的对象：

- 视图: 视图是一张虚拟表，该虚拟表的结构由创建视图时的 SELECT 语句定义，TiDB 目前不支持物化视图。
- 序列: 创建和存储顺序数据。
- 临时表: 临时表是数据不持久化的表。

4.5.1.6 访问控制

TiDB 支持基于用户或角色的访问控制。你可以通过角色或直接指向用户，从而授予用户查看、修改或删除数据对象和数据模式的权限。

4.5.1.7 执行数据库模式更改

不推荐使用客户端的 Driver 或 ORM 来执行数据库模式的更改。以经验来看，作为最佳实践，建议使用 MySQL 客户端或使用任意你喜欢的 GUI 客户端来进行数据库模式的更改。本文档中，将在大多数场景下，使用 MySQL 客户端传入 SQL 文件来执行数据库模式的更改。

4.5.1.8 对象大小限制

具体限制请参考TiDB 使用限制。

4.5.2 创建数据库

在这个章节当中，将开始介绍如何使用 SQL 来创建数据库，及创建数据库时应遵守的规则。将在这个章节中围绕Bookshop 这个应用程序来对 TiDB 的创建数据库部分展开介绍。

> **注意：**
>
> 此处仅对 CREATE DATABASE 语句进行简单描述，详细参考文档（包含其他示例），可参阅CREATE DATABASE 文档。

4.5.2.1 在开始之前

在阅读本页面之前，你需要准备以下事项：

- 使用 TiDB Serverless 构建 TiDB 集群。
- 阅读数据库模式概览。

4.5.2.2 什么是数据库

在 TiDB 中数据库对象可以包含表、视图、序列等对象。

4.5.2.3 创建数据库过程

可使用 CREATE DATABASE 语句来创建数据库。

```
CREATE DATABASE IF NOT EXISTS `bookshop`;
```

此语句会创建一个名为 bookshop 的数据库（如果尚不存在）。请以 root 用户身份执行文件中的建库语句，运行以下命令：

```
mysql
    -u root \
    -h {host} \
    -P {port} \
    -p {password} \
    -e "CREATE DATABASE IF NOT EXISTS bookshop;"
```

要查看集群中的数据库，可在命令行执行一条 SHOW DATABASES 语句：

```
mysql
    -u root \
    -h {host} \
    -P {port} \
    -p {password} \
    -e "SHOW DATABASES;"
```

运行结果为：

```
+--------------------+
| Database           |
+--------------------+
| INFORMATION_SCHEMA |
| PERFORMANCE_SCHEMA |
| bookshop           |
| mysql              |
| test               |
+--------------------+
```

4.5.2.4 数据库创建时应遵守的规则

- 遵循数据库命名规范，给你的数据库起一个有意义的名字。
- `test` 数据库是 TiDB 提供的一个默认数据库。如果没有必要，尽量不要在生产环境使用它。你可以自行使用 `CREATE DATABASE` 语句来创建数据库，并且在 SQL 会话中使用 `USE {databasename};` 语句来更改当前数据库。
- 使用 root 用户创建数据库、角色、用户等，并只赋予必要的权限。
- 作为通用的规则，不推荐使用 Driver、 ORM 进行数据库模式的定义与更改。相反，请使用 MySQL 命令行客户端或其他你喜欢的 MySQL GUI 客户端来进行操作。

4.5.2.5 更进一步

至此，你已经准备完毕 bookshop 数据库，可以将表添加到该数据库中。

你可继续阅读创建表文档获得相关指引。

4.5.3 创建表

本文档介绍如何使用 SQL 语句来创建表以及创建表的最佳实践。本文档提供了一个基于 TiDB 的bookshop 数据库的示例加以说明。

> **注意：**
>
> 此处仅对 `CREATE TABLE` 语句进行简单描述，详细参考文档（包含其他示例），可参阅CREATE TABLE 文档。

4.5.3.1 在开始之前

在阅读本页面之前，你需要准备以下事项：

- 使用 TiDB Serverless 构建 TiDB 集群。
- 阅读数据库模式概览。
- 创建一个数据库。

4.5.3.2 什么是表

表是集群中的一种逻辑对象，它从属于数据库，用于保存从 SQL 中发送的数据。表以行和列的形式组织数据记录, 一张表至少有一列。若在表中定义了 n 个列，那么每一行数据都将拥有与这 n 个列中完全一致的字段。

4.5.3.3 命名表

创建表的第一步，就是给你的表起个名字。请不要使用无意义的表名，将给未来的你或者你的同事带来极大的困扰。推荐你遵循公司或组织的表命名规范。

CREATE TABLE 语句通常采用以下形式：

```
CREATE TABLE {table_name} ( {elements} );
```

参数描述

- {table_name}: 表名。
- {elements}: 以逗号分隔的表元素列表，比如列定义，主键定义等。

假设你需要创建一个表来存储 bookshop 库中的用户信息。

注意，此时因为一个列都没被添加，所以下方这条 SQL 暂时还不能被运行：

```
CREATE TABLE `bookshop`.`users` (
);
```

4.5.3.4 定义列

列从属于表，每张表都至少有一列。列通过将每行中的值分成一个个单一数据类型的小单元来为表提供结构。

列定义通常使用以下形式：

```
{column_name} {data_type} {column_qualification}
```

参数描述

- {column_name}: 列名。
- {data_type}: 列的数据类型。
- {column_qualification}: 列的限定条件，如列级约束或生成列（实验功能）子句。

可以为 users 表添加一些列，如他们的唯一标识 id，余额 balance 及昵称 nickname。

```
CREATE TABLE `bookshop`.`users` (
  `id` bigint,
  `nickname` varchar(100),
  `balance` decimal(15,2)
);
```

其中，定义了一个字段名为 id，类型为bigint 的字段。用以表示用户唯一标识。这意味着，所有的用户标识都应该是 bigint 类型的。

而在其后，又定义了一个字段名为 nickname，类型为varchar，且长度不得超过 100 字符的字段。用以表示用户的昵称。这意味着，所用用户的昵称都是 varchar 类型，且不超过 100 字符的。

最后，又加入了一个字段名为 balance 用以表示用户的余额，类型为decimal，且其精度为 15，比例为 2。简单的说明一下精度和比例代表的含义，精度代表字段数值的总位数，而比例代表小数点后有多少位。例如: decimal(5,2)，即精度为 5，比例为 2 时，其取值范围为 -999.99 到 999.99。decimal(6,1)，即精度为 6，比例为 1 时，其取值范围为 -99999.9 到 99999.9。decimal 类型为定点数，可精确保存数字，在需要精确数字的场景（如用户财产相关）中，请确保使用定点数类型。

TiDB 支持许多其他的列数据类型，包含整数、浮点数、定点数、时间、枚举 等，可参考支持的列的数据类型，并使用与你准备保存在数据库内的数据匹配的数据类型。

稍微提升一下复杂度，例如选择定义一张 books 表，这张表将是 bookshop 数据的核心。它包含书的唯一标识、名称、书籍类型（如：杂志、动漫、教辅等）、库存、价格、出版时间等字段。

```
CREATE TABLE `bookshop`.`books` (
  `id` bigint NOT NULL,
  `title` varchar(100),
  `type` enum('Magazine', 'Novel', 'Life', 'Arts', 'Comics', 'Education & Reference', 'Humanities
      ↪  & Social Sciences', 'Science & Technology', 'Kids', 'Sports'),
  `published_at` datetime,
  `stock` int,
  `price` decimal(15,2)
);
```

这张表比 users 表包含更多的数据类型：

- int: 推荐使用合适大小的类型，防止使用过量的硬盘甚至影响性能 (类型范围过大) 或数据溢出 (类型范围过小)。
- datetime: 可以使用 datetime 类型保存时间值。
- enum: 可以使用 enum 类型的枚举来保存有限选择的值。

4.5.3.5 选择主键

主键是一个或一组列，这个由所有主键列组合起来的值是数据行的唯一标识。

> 注意：
>
> TiDB 中，关于 Primary Key 的默认定义与 MySQL 常用存储引擎 InnoDB 不一致。InnoDB 中，Primary Key 的语义为：唯一，不为空，且为聚簇索引。
>
> 而在 TiDB 中，Primary Key 的定义为：唯一，不为空。但主键不保证为聚簇索引。而是由另一组关键字 CLUSTERED、NONCLUSTERED 额外控制 Primary Key 是否为聚簇索引，若不指定，则由系统变量 @@global.tidb_enable_clustered_index 影响，具体说明请看此文档。

主键在 CREATE TABLE 语句中定义。主键约束要求所有受约束的列仅包含非 NULL 值。

一个表可以没有主键，主键也可以是非整数类型。但此时 TiDB 就会创建一个 _tidb_rowid 作为隐式主键。隐式主键 _tidb_rowid 因为其单调递增的特性，可能在大批量写入场景下会导致写入热点，如果你写入量密集，可考虑通过SHARD_ROW_ID_BITS 和PRE_SPLIT_REGIONS 两参数控制打散。但这可能导致读放大，请自行取舍。

表的主键为整数类型且使用了 AUTO_INCREMENT 时，无法使用 SHARD_ROW_ID_BITS 消除热点。需解决此热点问题，且无需使用主键的连续和递增时，可使用AUTO_RANDOM 替换 AUTO_INCREMENT 属性来消除行 ID 的连续性。

更多有关热点问题的处理办法，请参考TiDB 热点问题处理。

需遵循选择主键时应遵守的规则，举一个 users 表中定义 AUTO_RANDOM 主键的例子：

```
CREATE TABLE `bookshop`.`users` (
  `id` bigint AUTO_RANDOM,
  `balance` decimal(15,2),
  `nickname` varchar(100),
  PRIMARY KEY (`id`)
);
```

4.5.3.6 选择聚簇索引

聚簇索引 (clustered index) 是 TiDB 从 v5.0 开始支持的特性，用于控制含有主键的表数据的存储方式。通过使用聚簇索引，TiDB 可以更好地组织数据表，从而提高某些查询的性能。有些数据库管理系统也将聚簇索引称为“索引组织表” (index-organized tables)。

目前 TiDB 中 含有主键的表分为以下两类：

- NONCLUSTERED，表示该表的主键为非聚簇索引。在非聚簇索引表中，行数据的键由 TiDB 内部隐式分配的 _tidb_rowid 构成，而主键本质上是唯一索引，因此非聚簇索引表存储一行至少需要两个键值对，分别为：
  - _tidb_rowid（键）- 行数据（值）
  - 主键列数据（键）- _tidb_rowid（值）
- CLUSTERED，表示该表的主键为聚簇索引。在聚簇索引表中，行数据的键由用户给定的主键列数据构成，因此聚簇索引表存储一行至少只要一个键值对，即：
  - 主键列数据（键）- 行数据（值）

如主键中所述，聚簇索引在 TiDB 中，使用关键字 CLUSTERED、NONCLUSTERED 进行控制。

> 注意：
>
> TiDB 仅支持根据表的主键来进行聚簇操作。聚簇索引启用时，“主键”和“聚簇索引”两个术语在一些情况下可互换使用。主键指的是约束（一种逻辑属性），而聚簇索引描述的是数据存储的物理实现。

需遵循选择聚簇索引时应遵守的规则，假设需要建立一张 books 和 users 之间关联的表，代表用户对某书籍的评分。使用表名 ratings 来创建该表，并使用 book_id 和 user_id 构建复合主键，并在该主键上建立聚簇索引：

```
CREATE TABLE `bookshop`.`ratings` (
  `book_id` bigint,
  `user_id` bigint,
  `score` tinyint,
  `rated_at` datetime,
  PRIMARY KEY (`book_id`,`user_id`) CLUSTERED
);
```

4.5.3.7 添加列约束

除主键约束外，TiDB 还支持其他的列约束，如：非空约束 NOT NULL、唯一约束 UNIQUE KEY、默认值 DEFAULT 等。完整约束，请查看TiDB 约束文档。

4.5.3.7.1 填充默认值

如需在列上设置默认值，请使用 DEFAULT 约束。默认值将可以使你无需指定每一列的值，就可以插入数据。

你可以将 DEFAULT 与支持的 SQL 函数结合使用，将默认值的计算移出应用层，从而节省应用层的资源（当然，计算所消耗的资源并不会凭空消失，只是被转移到了 TiDB 集群中）。常见的，希望实现数据插入时，可默认填充默认的时间。还是使用 ratings 作为示例，可使用以下语句：

```
CREATE TABLE `bookshop`.`ratings` (
  `book_id` bigint,
  `user_id` bigint,
  `score` tinyint,
  `rated_at` datetime DEFAULT NOW(),
  PRIMARY KEY (`book_id`,`user_id`) CLUSTERED
);
```

额外的，如果需更新时也默认填入当前时间，可使用以下语句（但 ON UPDATE 后仅可填入当前时间相关语句，DEFAULT 后支持更多选择）：

```
CREATE TABLE `bookshop`.`ratings` (
  `book_id` bigint,
  `user_id` bigint,
  `score` tinyint,
  `rated_at` datetime DEFAULT NOW() ON UPDATE NOW(),
  PRIMARY KEY (`book_id`,`user_id`) CLUSTERED
);
```

4.5.3.7.2 防止重复

如果你需要防止列中出现重复值，那你可以使用 UNIQUE 约束。

例如，你需要确保用户的昵称唯一，可以这样改写 users 表的创建 SQL：

```
CREATE TABLE `bookshop`.`users` (
  `id` bigint AUTO_RANDOM,
  `balance` decimal(15,2),
  `nickname` varchar(100) UNIQUE,
  PRIMARY KEY (`id`)
);
```

如果你在 users 表中尝试插入相同的 nickname，将返回错误。

4.5.3.7.3 防止空值

如果你需要防止列中出现空值，那就可以使用 NOT NULL 约束。

还是使用用户昵称来举例子，除了昵称唯一，还希望昵称不可为空，于是此处可以这样改写 users 表的创建 SQL：

```
CREATE TABLE `bookshop`.`users` (
  `id` bigint AUTO_RANDOM,
  `balance` decimal(15,2),
  `nickname` varchar(100) UNIQUE NOT NULL,
  PRIMARY KEY (`id`)
);
```

4.5.3.8 使用 HTAP 能力

> 注意：
>
> 本指南中有关 HTAP 的步骤仅适用于快速上手体验，不适用于生产环境。如需探索 HTAP 更多功能，请参考深入探索 HTAP。

假设 bookshop 应用程序，有对用户评价的 ratings 表进行 OLAP 分析查询的需求，例如需查询: 书籍的评分，是否和评价的时间具有显著的相关性的需求，用以分析用户的书籍评分是否客观。那么会要求查询整个 ratings 表中的 score 和 rated_at 字段。这对普通仅支持的 OLTP 的数据库来说，是一个非常消耗资源的操作。或者使用一些 ETL 或其他数据同步工具，将 OLTP 数据库中的数据，导出到专用的 OLAP 数据库，再进行分析。

这种场景下，TiDB 就是一个比较理想的一站式数据库解决方案，TiDB 是一个 HTAP (Hybrid Transactional and Analytical Processing) 数据库，同时支持 OLTP 和 OLAP 场景。

4.5.3.8.1 同步列存数据

当前，TiDB 支持两种数据分析引擎：TiFlash 和 TiSpark。大数据场景 (100 T) 下，推荐使用 TiFlash MPP 作为 HTAP 的主要方案，TiSpark 作为补充方案。希望了解更多关于 TiDB 的 HTAP 能力，可参考以下文章：快速上手 HTAP 和深入探索 HTAP。

此处选用TiFlash 为 bookshop 数据库的数据分析引擎。

TiFlash 部署完成后并不会自动同步数据，而需要手动指定需要同步的表，开启同步副本仅需一行 SQL，如下所示：

```
ALTER TABLE {table_name} SET TIFLASH REPLICA {count};
```

参数描述

- {table_name}: 表名。
- {count}: 同步副本数，若为 0，则表示删除同步副本。

随后，TiFlash 将同步该表，查询时，TiDB 将会自动基于成本优化，考虑使用 TiKV (行存) 或 TiFlash (列存) 进行数据查询。当然，除了自动的方法，你也可以直接指定查询是否使用 TiFlash 副本，使用方法可查看使用 TiDB 读取 TiFlash 文档。

4.5.3.8.2 使用 HTAP 的示例

ratings 表开启 1 个 TiFlash 副本：

```
ALTER TABLE `bookshop`.`ratings` SET TIFLASH REPLICA 1;
```

> 注意：
>
> 如果你的集群，不包含 TiFlash 节点，此 SQL 语句将会报错：1105 - the tiflash replica count ↪ : 1 should be less than the total tiflash server count: 0 你可以使用 TiDB Serverless 构建 TiDB 集群 来创建一个含有 TiFlash 的集群。

随后正常进行查询即可：

```
SELECT HOUR(`rated_at`), AVG(`score`) FROM `bookshop`.`ratings` GROUP BY HOUR(`rated_at`);
```

也可使用EXPLAIN ANALYZE 语句查看此语句是否使用了 TiFlash 引擎：

```
EXPLAIN ANALYZE SELECT HOUR(`rated_at`), AVG(`score`) FROM `bookshop`.`ratings` GROUP BY HOUR(`
    ↪ rated_at`);
```

运行结果为：

```
+----------------------------+----------+---------+--------------+---------------+-------------------------------
    ↪
| id                         | estRows  | actRows | task         | access object | execution
    ↪ info
    ↪
    ↪ | operator info
    ↪
    ↪ | memory   | disk |
```

```
+-----------------------------+----------+---------+-------------+---------------+-------------------------------
    ↪
| Projection_4                | 299821.99 | 24      | root        |               | time:60.8ms,
    ↪  loops:6, Concurrency:5
    ↪
    ↪ | hour(cast(bookshop.ratings.rated_at, time))->Column#6, Column#5
    ↪                                                                                | 17.7 KB
    ↪  | N/A  |
| └─HashAgg_5                 | 299821.99 | 24      | root        |               | time:60.7ms
    ↪ , loops:6, partial_worker:{wall_time:60.660079ms, concurrency:5, task_num:293, tot_wait
    ↪ :262.536669ms, tot_exec:40.171833ms, tot_time:302.827753ms, max:60.636886ms, p95
    ↪ :60.636886ms}, final_worker:{wall_time:60.701437ms, concurrency:5, task_num:25, tot_wait
    ↪ :303.114278ms, tot_exec:176.564µs, tot_time:303.297475ms, max:60.69326ms, p95:60.69326ms}
    ↪   | group by:Column#10, funcs:avg(Column#8)->Column#5, funcs:firstrow(Column#9)->bookshop
    ↪ .ratings.rated_at                                          | 714.0 KB | N/A  |
|   └─Projection_15           | 300000.00 | 300000  | root        |               | time:58.5ms
    ↪ , loops:294, Concurrency:5
    ↪
    ↪ | cast(bookshop.ratings.score, decimal(8,4) BINARY)->Column#8, bookshop.ratings.rated_at,
    ↪  hour(cast(bookshop.ratings.rated_at, time))->Column#10 | 366.2 KB | N/A  |
|     └─TableReader_10        | 300000.00 | 300000  | root        |               | time:43.5ms
    ↪ , loops:294, cop_task: {num: 1, max: 43.1ms, proc_keys: 0, rpc_num: 1, rpc_time: 43ms,
    ↪ copr_cache_hit_ratio: 0.00}
    ↪
    ↪ | data:TableFullScan_9
    ↪
    ↪ | 4.58 MB  | N/A  |
|       └─TableFullScan_9     | 300000.00 | 300000  | cop[tiflash] | table:ratings |
    ↪ tiflash_task:{time:5.98ms, loops:8, threads:1}, tiflash_scan:{dtfile:{total_scanned_packs
    ↪ :45, total_skipped_packs:1, total_scanned_rows:368640, total_skipped_rows:8192,
    ↪ total_rs_index_load_time: 1ms, total_read_time: 1ms},total_create_snapshot_time:1ms}
    ↪
    ↪ | keep order:false
    ↪
    ↪ | N/A      | N/A  |
+-----------------------------+----------+---------+-------------+---------------+-------------------------------
    ↪
```

在出现 `cop[tiflash]` 字样时，表示该任务发送至 TiFlash 进行处理。

4.5.3.9 执行 CREATE TABLE 语句

按以上步骤创建所有表后，数据库初始化脚本应该如此所示。若需查看表信息详解，请参阅数据表详解。

如果将数据库初始化脚本命名为 `init.sql` 并保存，可使用以下语句来执行数据库初始化：

```
mysql
    -u root \
    -h {host} \
    -P {port} \
    -p {password} \
    < init.sql
```

需查看 bookshop 数据库下的所有表，可使用SHOW TABLES 语句：

```
SHOW TABLES IN `bookshop`;
```

运行结果为：

```
+--------------------+
| Tables_in_bookshop |
+--------------------+
| authors            |
| book_authors       |
| books              |
| orders             |
| ratings            |
| users              |
+--------------------+
```

4.5.3.10 创建表时应遵守的规则

本小节给出了一些在创建表时应遵守的规则。

4.5.3.10.1 命名表时应遵守的规则

- 使用完全限定的表名称（例如：CREATE TABLE {database_name}.{table_name}）。这是因为你在不指定数据库名称时，TiDB 将使用你 SQL 会话中的当前数据库。若你未在 SQL 会话中使用 USE {databasename}; 来指定数据库，TiDB 将会返回错误。
- 请使用有意义的表名，例如，若你需要创建一个用户表，你可以使用名称：user, t_user, users 等，或遵循你公司或组织的命名规范。如果你的公司或组织没有相应的命名规范，可参考表命名规范。请勿使用这样的表名，如：t1, table1 等。
- 多个单词以下划线分隔，不推荐超过 32 个字符。
- 不同业务模块的表单独建立 DATABASE，并增加相应注释。

4.5.3.10.2 定义列时应遵守的规则

- 查看支持的列的数据类型，并按照数据类型的限制来组织你的数据。为你计划被存在列中的数据选择合适的类型。
- 查看选择主键时应遵守的规则，决定是否使用主键列。
- 查看选择聚簇索引时应遵守的规则，决定是否指定聚簇索引。

- 查看添加列约束，决定是否添加约束到列中。
- 请使用有意义的列名，推荐你遵循公司或组织的表命名规范。如果你的公司或组织没有相应的命名规范，可参考列命名规范。

4.5.3.10.3 选择主键时应遵守的规则

- 在表内定义一个主键或唯一索引。
- 尽量选择有意义的列作为主键。
- 出于为性能考虑，尽量避免存储超宽表，表字段数不建议超过 60 个，建议单行的总数据大小不要超过 64K，数据长度过大字段最好拆到另外的表。
- 不推荐使用复杂的数据类型。
- 需要 JOIN 的字段，数据类型保障绝对一致，避免隐式转换。
- 避免在单个单调数据列上定义主键。如果你使用单个单调数据列（例如：AUTO_INCREMENT 的列）来定义主键，有可能会对写性能产生负面影响。可能的话，使用 AUTO_RANDOM 替换 AUTO_INCREMENT（这会失去主键的连续和递增特性）。
- 如果你 必须在单个单调数据列上创建索引，且有大量写入的话。请不要将这个单调数据列定义为主键，而是使用 AUTO_RANDOM 创建该表的主键，或使用SHARD_ROW_ID_BITS 和PRE_SPLIT_REGIONS 打散 _tidb_rowid。

4.5.3.10.4 选择聚簇索引时应遵守的规则

- 遵循选择主键时应遵守的规则：

  聚簇索引将基于主键建立，请遵循选择主键时应遵守的规则，此为选择聚簇索引时应遵守规则的基础。

- 在以下场景中，尽量使用聚簇索引，将带来性能和吞吐量的优势：

  - 插入数据时会减少一次从网络写入索引数据。
  - 等值条件查询仅涉及主键时会减少一次从网络读取数据。
  - 范围条件查询仅涉及主键时会减少多次从网络读取数据。
  - 等值或范围条件查询仅涉及主键的前缀时会减少多次从网络读取数据。

- 在以下场景中，尽量避免使用聚簇索引，将带来性能劣势：

  - 批量插入大量取值相邻的主键时，可能会产生较大的写热点问题，请遵循选择主键时应遵守的规则。
  - 当使用大于 64 位的数据类型作为主键时，可能导致表数据需要占用更多的存储空间。该现象在存在多个二级索引时尤为明显。

- 显式指定是否使用聚簇索引，而非使用系统变量 @@global.tidb_enable_clustered_index 及配置项 alter-primary-key 控制是否使用聚簇索引的默认行为。

4.5.3.10.5 CREATE TABLE 执行时应遵守的规则

- 不推荐使用客户端的 Driver 或 ORM 来执行数据库模式的更改。基于过往经验，建议使用 MySQL 客户端或使用任意你喜欢的 GUI 客户端来进行数据库模式的更改。本文档中，将在大多数场景下，使用 MySQL 客户端传入 SQL 文件来执行数据库模式的更改。
- 遵循 SQL 开发规范中的建表删表规范，建议业务应用内部封装建表删表语句增加判断逻辑。

4.5.3.11 更进一步

请注意，到目前为止，创建的所有表都不包含二级索引。添加二级索引的指南，请参考创建二级索引。

4.5.4 创建二级索引

在这个章节当中，将开始介绍如何使用 SQL 来创建二级索引，及创建二级索引时应遵守的规则。将在这个章节中围绕Bookshop 这个应用程序来对 TiDB 的创建二级索引部分展开介绍。

4.5.4.1 在开始之前

在阅读本页面之前，你需要准备以下事项：

- 使用 TiDB Serverless 构建 TiDB 集群。
- 阅读数据库模式概览。
- 创建一个数据库。
- 创建表。

4.5.4.2 什么是二级索引

二级索引是集群中的逻辑对象，你可以简单地认为它就是一种对数据的排序，TiDB 使用这种有序性来加速查询。TiDB 的创建二级索引的操作为在线操作，不会阻塞表中的数据读写。TiDB 会创建表中各行的引用，并按选择的列进行排序。而并非对表本身的数据进行排序。可在二级索引中查看更多信息。二级索引可跟随表进行创建，也可在已有的表上进行添加。

4.5.4.3 在已有表中添加二级索引

如果需要对已有表中添加二级索引，可使用CREATE INDEX 语句。在 TiDB 中，CREATE INDEX 为在线操作，不会阻塞表中的数据读写。二级索引创建一般如以下形式：

```
CREATE INDEX {index_name} ON {table_name} ({column_names});
```

参数描述

- {index_name}: 二级索引名。
- {table_name}: 表名。
- {column_names}: 将需要索引的列名列表，以半角逗号分隔。

4.5.4.4 新建表的同时创建二级索引

如果你希望在创建表的同时，同时创建二级索引，可在CREATE TABLE 的末尾使用包含 KEY 关键字的子句来创建二级索引：

```
KEY `{index_name}` (`{column_names}`)
```

参数描述

- {index_name}: 二级索引名。
- {column_names}: 将需要索引的列名列表，以半角逗号分隔。

4.5.4.5 创建二级索引时应遵守的规则

见索引的最佳实践。

4.5.4.6 例子

假设你希望 bookshop 应用程序有 查询某个年份出版的所有书籍的功能。books 表如下所示:

字段名	类型	含义
id	bigint(20)	书籍的唯一标识
title	varchar(100)	书籍名称
type	enum	书籍类型（如：杂志、动漫、教辅等）
stock	bigint(20)	库存
price	decimal(15,2)	价格
published_at	datetime	出版时间

```
CREATE TABLE `bookshop`.`books` (
  `id` bigint(20) AUTO_RANDOM NOT NULL,
  `title` varchar(100) NOT NULL,
  `type` enum('Magazine', 'Novel', 'Life', 'Arts', 'Comics', 'Education & Reference', 'Humanities
    ↪  & Social Sciences', 'Science & Technology', 'Kids', 'Sports') NOT NULL,
  `published_at` datetime NOT NULL,
  `stock` int(11) DEFAULT '0',
  `price` decimal(15,2) DEFAULT '0.0',
  PRIMARY KEY (`id`) CLUSTERED
) DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin;
```

因此，就需要对 查询某个年份出版的所有书籍的 SQL 进行编写，以 2022 年为例，如下所示：

```
SELECT * FROM `bookshop`.`books` WHERE `published_at` >= '2022-01-01 00:00:00' AND `published_at`
    ↪  < '2023-01-01 00:00:00';
```

可以使用EXPLAIN 进行 SQL 语句的执行计划检查：

```
EXPLAIN SELECT * FROM `bookshop`.`books` WHERE `published_at` >= '2022-01-01 00:00:00' AND `
    ↪ published_at` < '2023-01-01 00:00:00';
```

运行结果为：

```
+-------------------------+----------+-----------+--------------+--------------------------------------------------
    ↪
| id                      | estRows  | task      | access object | operator info
    ↪
    ↪ |
+-------------------------+----------+-----------+--------------+--------------------------------------------------
    ↪
```

```
| TableReader_7           | 346.32   | root      |               | data:Selection_6
    ↪
    ↪ |
| └─Selection_6           | 346.32   | cop[tikv] |               | ge(bookshop.books.
    ↪ published_at, 2022-01-01 00:00:00.000000), lt(bookshop.books.published_at, 2023-01-01
    ↪ 00:00:00.000000) |
|   └─TableFullScan_5     | 20000.00 | cop[tikv] | table:books   | keep order:false
    ↪
    ↪ |
+-------------------------+----------+-----------+---------------+-----------------------------------------------
    ↪
3 rows in set (0.61 sec)
```

可以看到返回的计划中，出现了类似 TableFullScan 的字样，这代表 TiDB 准备在这个查询中对 books 表进行全表扫描，这在数据量较大的情况下，几乎是致命的。

在 books 表增加一个 published_at 列的索引：

```
CREATE INDEX `idx_book_published_at` ON `bookshop`.`books` (`bookshop`.`books`.`published_at`);
```

添加索引后，再次运行 EXPLAIN 语句检查执行计划：

```
+-------------------------------+---------+-----------+-------------------------------------------------------+
    ↪
| id                            | estRows | task      | access object
    ↪                                       | operator info
    ↪                                                 |
+-------------------------------+---------+-----------+-------------------------------------------------------+
    ↪
| IndexLookUp_10                | 146.01  | root      |
    ↪                                                       |
    ↪                                                                 |
| ├─IndexRangeScan_8(Build)     | 146.01  | cop[tikv] | table:books, index:idx_book_published_at
    ↪ (published_at) | range:[2022-01-01 00:00:00,2023-01-01 00:00:00), keep order:false |
| └─TableRowIDScan_9(Probe)     | 146.01  | cop[tikv] | table:books
    ↪                                             | keep order:false
    ↪                                                   |
+-------------------------------+---------+-----------+-------------------------------------------------------+
    ↪
3 rows in set (0.18 sec)
```

可以看到执行计划中没有了 TableFullScan 的字样，取而代之的是 IndexRangeScan，这代表已经 TiDB 在进行这个查询时准备使用索引。

> 注意：

> 上方执行计划中的的 TableFullScan、IndexRangeScan 等在 TiDB 内被称为算子。这里对执行计划的解读及算子等不做进一步的展开，若你对此感兴趣，可前往TiDB 执行计划概览文档查看更多关于执行计划与 TiDB 算子的相关知识。
>
> 执行计划并非每次返回使用的算子都相同，这是由于 TiDB 使用的优化方式为 基于代价的优化方式 (CBO)，执行计划不仅与规则相关，还和数据分布相关。你可以前往SQL 性能调优文档查看更多 TiDB SQL 性能的描述。
>
> TiDB 在查询时，还支持显式地使用索引，你可以使用Optimizer Hints 或执行计划管理 (SPM) 来人为的控制索引的使用。但如果你不了解它内部发生了什么，请你暂时先不要使用它。

可以使用SHOW INDEXES 语句查询表中的索引：

```
SHOW INDEXES FROM `bookshop`.`books`;
```

运行结果为：

```
+-------+------------+----------------------+--------------+--------------+-----------+-------------+----------
    ↪
| Table | Non_unique | Key_name             | Seq_in_index | Column_name  | Collation |
    ↪ Cardinality | Sub_part | Packed | Null | Index_type | Comment | Index_comment | Visible |
    ↪  Expression | Clustered |
+-------+------------+----------------------+--------------+--------------+-----------+-------------+----------
    ↪
| books |          0 | PRIMARY              |            1 | id           | A         |
    ↪           0 |     NULL | NULL   |      | BTREE      |         |               | YES     |
    ↪  NULL       | YES       |
| books |          1 | idx_book_published_at |            1 | published_at | A         |
    ↪           0 |     NULL | NULL   |      | BTREE      |         |               | YES     |
    ↪  NULL       | NO        |
+-------+------------+----------------------+--------------+--------------+-----------+-------------+----------
    ↪
2 rows in set (1.63 sec)
```

4.5.4.7 更进一步

至此，你已经完成数据库、表及二级索引的创建，接下来，数据库模式已经准备好给你的应用程序提供写入和读取的能力了。

4.6 数据写入

4.6.1 插入数据

此页面将展示使用 SQL 语言，配合各种编程语言将数据插入到 TiDB 中。

4.6.1.1 在开始之前

在阅读本页面之前，你需要准备以下事项：

- 使用 TiDB Serverless 构建 TiDB 集群。
- 阅读数据库模式概览，并创建数据库、创建表、创建二级索引。

4.6.1.2 插入行

假设你需要插入多行数据，那么会有两种插入的办法，假设需要插入 3 个玩家数据：

- 一个多行插入语句：

```
INSERT INTO `player` (`id`, `coins`, `goods`) VALUES (1, 1000, 1), (2, 230, 2), (3, 300, 5);
```

- 多个单行插入语句：

```
INSERT INTO `player` (`id`, `coins`, `goods`) VALUES (1, 1000, 1);
INSERT INTO `player` (`id`, `coins`, `goods`) VALUES (2, 230, 2);
INSERT INTO `player` (`id`, `coins`, `goods`) VALUES (3, 300, 5);
```

一般来说使用一个多行插入语句，会比多个单行插入语句快。

在 SQL 中插入多行数据的示例：

```
CREATE TABLE `player` (`id` INT, `coins` INT, `goods` INT);
INSERT INTO `player` (`id`, `coins`, `goods`) VALUES (1, 1000, 1), (2, 230, 2);
```

有关如何使用此 SQL，可查阅连接到 TiDB 集群文档部分，按文档步骤使用客户端连接到 TiDB 集群后，输入 SQL 语句即可。

在 Java 中插入多行数据的示例：

```
// ds is an entity of com.mysql.cj.jdbc.MysqlDataSource
try (Connection connection = ds.getConnection()) {
    connection.setAutoCommit(false);

    PreparedStatement pstmt = connection.prepareStatement("INSERT INTO player (id, coins, goods)
        ↪ VALUES (?, ?, ?)"))

    // first player
    pstmt.setInt(1, 1);
    pstmt.setInt(2, 1000);
    pstmt.setInt(3, 1);
    pstmt.addBatch();

    // second player
    pstmt.setInt(1, 2);
```

```
    pstmt.setInt(2, 230);
    pstmt.setInt(3, 2);
    pstmt.addBatch();

    pstmt.executeBatch();
    connection.commit();
} catch (SQLException e) {
    e.printStackTrace();
}
```

另外，由于 MySQL JDBC Driver 默认设置问题，你需更改部分参数，以获得更好的批量插入性能：

参数	作用	推荐场景	推荐配置
useServerPrepStmts	是否使用服务端开启预处理语句支持	在需要多次使用预处理语句时	true
cachePrepStmts	客户端是否缓存预处理语句	useServerPrepStmts=true 时	true
prepStmtCacheSqlLimit	预处理语句最大大小（默认 256 字符）	预处理语句大于 256 字符时	按实际预处理语句大小配置
prepStmtCacheSize	预处理语句最大缓存数量（默认 25 条）	预处理语句数量大于 25 条时	按实际预处理语句数量配置
rewriteBatchedStatements	是否重写 batch 语句	需要批量操作时	true
allowMultiQueries	开启批量操作	因为一个客户端 Bug 在 rewriteBatchedStatements = true 和 useServerPrepStmts = true 时，需设置此项	true

MySQL JDBC Driver 还提供了一个集成配置项：useConfigs。当它配置为 maxPerformance 时，相当于配置了一组配置，以 mysql:mysql-connector-java:8.0.28 为例，useConfigs=maxPerformance 包含：

```
cachePrepStmts=true
cacheCallableStmts=true
cacheServerConfiguration=true
useLocalSessionState=true
elideSetAutoCommits=true
alwaysSendSetIsolation=false
enableQueryTimeouts=false
connectionAttributes=none
useInformationSchema=true
```

你可以自行查看 mysql-connector-java-{version}.jar!/com/mysql/cj/configurations/maxPerformance.properties 来获得对应版本 MySQL JDBC Driver 的 useConfigs=maxPerformance 包含配置。

在此处给出一个较为的通用场景的 JDBC 连接字符串配置，以 Host: 127.0.0.1，Port: 4000，用户名: root，密码: 空，默认数据库: test为例：

```
jdbc:mysql://127.0.0.1:4000/test?user=root&useConfigs=maxPerformance&useServerPrepStmts=true&
    ↪ prepStmtCacheSqlLimit=2048&prepStmtCacheSize=256&rewriteBatchedStatements=true&
    ↪ allowMultiQueries=true
```

有关 Java 的完整示例，可参阅：

- TiDB 和 Java 的简单 CRUD 应用程序 - 使用 JDBC
- TiDB 和 Java 的简单 CRUD 应用程序 - 使用 Hibernate
- Build the TiDB Application using Spring Boot

在 Golang 中插入多行数据的示例：

```
package main

import (
    "database/sql"
    "strings"

    _ "github.com/go-sql-driver/mysql"
)

type Player struct {
    ID    string
    Coins int
    Goods int
}

func bulkInsertPlayers(db *sql.DB, players []Player, batchSize int) error {
    tx, err := db.Begin()
    if err != nil {
        return err
    }

    stmt, err := tx.Prepare(buildBulkInsertSQL(batchSize))
    if err != nil {
        return err
    }

    defer stmt.Close()

    for len(players) > batchSize {
        if _, err := stmt.Exec(playerToArgs(players[:batchSize])...); err != nil {
            tx.Rollback()
```

```
            return err
        }

        players = players[batchSize:]
    }

    if len(players) != 0 {
        if _, err := tx.Exec(buildBulkInsertSQL(len(players)), playerToArgs(players)...); err !=
            ↪ nil {
            tx.Rollback()
            return err
        }
    }

    if err := tx.Commit(); err != nil {
        tx.Rollback()
        return err
    }

    return nil
}

func playerToArgs(players []Player) []interface{} {
    var args []interface{}
    for _, player := range players {
        args = append(args, player.ID, player.Coins, player.Goods)
    }
    return args
}

func buildBulkInsertSQL(amount int) string {
    return "INSERT INTO player (id, coins, goods) VALUES (?, ?, ?)" + strings.Repeat(",(?,?,?)",
        ↪ amount-1)
}
```

有关 Golang 的完整示例，可参阅：

- TiDB 和 Golang 的简单 CRUD 应用程序 - 使用 go-sql-driver/mysql
- TiDB 和 Golang 的简单 CRUD 应用程序 - 使用 GORM

在 Python 中插入多行数据的示例：

```
import MySQLdb

connection = MySQLdb.connect(
    host="127.0.0.1",
```

```
    port=4000,
    user="root",
    password="",
    database="bookshop",
    autocommit=True
)
with get_connection(autocommit=True) as connection:

    with connection.cursor() as cur:
        player_list = random_player(1919)
        for idx in range(0, len(player_list), 114):
            cur.executemany("INSERT INTO player (id, coins, goods) VALUES (%s, %s, %s)",
                ↪ player_list[idx:idx + 114])
```

有关 Python 的完整示例，可参阅：

- TiDB 和 Python 的简单 CRUD 应用程序 - 使用 PyMySQL
- TiDB 和 Python 的简单 CRUD 应用程序 - 使用 mysqlclient
- TiDB 和 Python 的简单 CRUD 应用程序 - 使用 mysql-connector-python
- TiDB 和 Python 的简单 CRUD 应用程序 - 使用 SQLAlchemy
- TiDB 和 Python 的简单 CRUD 应用程序 - 使用 peewee

4.6.1.3 批量插入

如果你需要快速地将大量数据导入 TiDB 集群，最好的方式并不是使用 INSERT 语句，这并不是最高效的方法，而且需要你自行处理异常等问题。推荐使用 PingCAP 提供的一系列工具进行数据迁移：

- 数据导出工具：Dumpling。可以导出 MySQL 或 TiDB 的数据到本地或 Amazon S3 中。
- 数据导入工具：TiDB Lightning。可以导入 Dumpling 导出的数据、CSV 文件，或者Amazon Aurora 生成的 Apache Parquet 文件。同时支持在本地盘或 Amazon S3 云盘读取数据。
- 数据同步工具：TiDB Data Migration。可同步 MySQL、MariaDB、Amazon Aurora 数据库到 TiDB 中。且支持分库分表数据库的迁移。
- 数据备份恢复工具：Backup & Restore (BR)。相对于 Dumpling，BR 更适合大数据量的场景。

4.6.1.4 避免热点

在设计表时需要考虑是否存在大量插入行为，若有，需在表设计期间对热点进行规避。请查看创建表 - 选择主键部分，并遵从选择主键时应遵守的规则。

更多有关热点问题的处理办法，请参考TiDB 热点问题处理文档。

4.6.1.5 主键为 AUTO_RANDOM 表插入数据

在插入的表主键为 AUTO_RANDOM 时，这时默认情况下，不能指定主键。例如bookshop 数据库中，可以看到users 表 的 id 字段含有 AUTO_RANDOM 属性。

此时，不可使用类似以下 SQL 进行插入：

```
INSERT INTO `bookshop`.`users` (`id`, `balance`, `nickname`) VALUES (1, 0.00, 'nicky');
```

将会产生错误：

```
ERROR 8216 (HY000): Invalid auto random: Explicit insertion on auto_random column is disabled.
    ↪ Try to set @@allow_auto_random_explicit_insert = true.
```

这是旨在提示你，不建议在插入时手动指定 AUTO_RANDOM 的列。这时，你有两种解决办法处理此错误：

- (推荐) 插入语句中去除此列，使用 TiDB 帮你初始化的 AUTO_RANDOM 值。这样符合 AUTO_RANDOM 的语义。

  ```
  INSERT INTO `bookshop`.`users` (`balance`, `nickname`) VALUES (0.00, 'nicky');
  ```

- 如果你确认一定需要指定此列，那么可以使用SET 语句通过更改用户变量的方式，允许在插入时，指定 AUTO_RANDOM 的列。

  ```
  SET @@allow_auto_random_explicit_insert = true;
  INSERT INTO `bookshop`.`users` (`id`, `balance`, `nickname`) VALUES (1, 0.00, 'nicky');
  ```

4.6.1.6 使用 HTAP

在 TiDB 中，使用 HTAP 能力无需你在插入数据时进行额外操作。不会有任何额外的插入逻辑，由 TiDB 自动进行数据的一致性保证。你只需要在创建表后，开启列存副本同步，就可以直接使用列存副本来加速你的查询。

4.6.2 更新数据

此页面将展示以下 SQL 语句，配合各种编程语言 TiDB 中的数据进行更新：

- UPDATE: 用于修改指定表中的数据。
- INSERT ON DUPLICATE KEY UPDATE: 用于插入数据，在有主键或唯一键冲突时，更新此数据。注意，不建议在有多个唯一键 (包含主键) 的情况下使用此语句。这是因为此语句在检测到任何唯一键 (包括主键) 冲突时，将更新数据。在不止匹配到一行冲突时，将只会更新一行数据。

4.6.2.1 在开始之前

在阅读本页面之前，你需要准备以下事项：

- 使用 TiDB Serverless 构建 TiDB 集群
- 阅读数据库模式概览，并创建数据库、创建表、创建二级索引
- 若需使用 UPDATE 语句更新数据，需先插入数据

4.6.2.2 使用 UPDATE

需更新表中的现有行，需要使用带有 WHERE 子句的UPDATE 语句，即需要过滤列进行更新。

> 注意：
>
> 如果您需要更新大量的行，比如数万甚至更多行，那么建议不要一次性进行完整的更新，而是每次迭代更新一部分，直到所有行全部更新。您可以编写脚本或程序，使用循环完成此操作。您可参考批量更新获得指引。

4.6.2.2.1 SQL 语法

在 SQL 中，UPDATE 语句一般为以下形式：

```
UPDATE {table} SET {update_column} = {update_value} WHERE {filter_column} = {filter_value}
```

参数	描述
{table}	表名
{update_column}	需更新的列名
{update_value}	需更新的此列的值
{filter_column}	匹配条件过滤器的列名
{filter_value}	匹配条件过滤器的列值

此处仅展示 UPDATE 的简单用法，详细文档可参考 TiDB 的UPDATE 语法页。

4.6.2.2.2 UPDATE 最佳实践

以下是更新行时需要遵循的一些最佳实践：

- 始终在更新语句中指定 WHERE 子句。如果 UPDATE 没有 WHERE 子句，TiDB 将更新这个表内的所有行。
- 需要更新大量行 (数万或更多) 的时候，使用批量更新，这是因为 TiDB 单个事务大小限制为txn-total-size-limit (默认为 100MB)，且一次性过多的数据更新，将导致持有锁时间过长 (悲观事务)，或产生大量冲突 (乐观事务)。

4.6.2.2.3 UPDATE 例子

假设某位作者改名为 Helen Haruki，需要更改authors 表。假设他的唯一标识 id 为 1，即过滤器应为：id = 1。

在 SQL 中更改作者姓名的示例为：

```
UPDATE `authors` SET `name` = "Helen Haruki" WHERE `id` = 1;
```

在 Java 中更改作者姓名的示例为：

```
// ds is an entity of com.mysql.cj.jdbc.MysqlDataSource
try (Connection connection = ds.getConnection()) {
    PreparedStatement pstmt = connection.prepareStatement("UPDATE `authors` SET `name` = ? WHERE
        ↪ `id` = ?");
```

```
    pstmt.setString(1, "Helen Haruki");
    pstmt.setInt(2, 1);
    pstmt.executeUpdate();
} catch (SQLException e) {
    e.printStackTrace();
}
```

4.6.2.3 使用 INSERT ON DUPLICATE KEY UPDATE

如果你需要将新数据插入表中，但如果有任何唯一键（主键也是一种唯一键）发生冲突，则会更新第一条冲突数据，可使用 INSERT ... ON DUPLICATE KEY UPDATE ... 语句进行插入或更新。

4.6.2.3.1 SQL 语法

在 SQL 中，INSERT ... ON DUPLICATE KEY UPDATE ... 语句一般为以下形式：

```
INSERT INTO {table} ({columns}) VALUES ({values})
    ON DUPLICATE KEY UPDATE {update_column} = {update_value};
```

参数	描述
{table}	表名
{columns}	需插入的列名
{values}	需插入的此列的值
{update_column}	需更新的列名
{update_value}	需更新的此列的值

4.6.2.3.2 INSERT ON DUPLICATE KEY UPDATE 最佳实践

- 在仅有一个唯一键的表上使用 INSERT ON DUPLICATE KEY UPDATE。此语句在检测到任何 唯一键 (包括主键) 冲突时，将更新数据。在不止匹配到一行冲突时，将只会一行数据。因此，除非能保证仅有一行冲突，否则不建议在有多个唯一键的表中使用 INSERT ON DUPLICATE KEY UPDATE 语句。
- 在创建或更新的场景中使用此语句。

4.6.2.3.3 INSERT ON DUPLICATE KEY UPDATE 例子

例如，需要更新ratings 表来写入用户对书籍的评价，如果用户还未评价此书籍，将新建一条评价，如果用户已经评价过，那么将会更新他之前的评价。

此处主键为 book_id 和 user_id 的联合主键。user_id 为 1 的用户，给 book_id 为 1000 的书籍，打出的 5 分的评价。

在 SQL 中更新书籍评价的示例为：

```
INSERT INTO `ratings`
    (`book_id`, `user_id`, `score`, `rated_at`)
```

```
VALUES
    (1000, 1, 5, NOW())
ON DUPLICATE KEY UPDATE `score` = 5, `rated_at` = NOW();
```

在 Java 中更新书籍评价的示例为：

```
// ds is an entity of com.mysql.cj.jdbc.MysqlDataSource

try (Connection connection = ds.getConnection()) {
    PreparedStatement p = connection.prepareStatement("INSERT INTO `ratings` (`book_id`, `user_id
        ↪ `, `score`, `rated_at`)
VALUES (?, ?, ?, NOW()) ON DUPLICATE KEY UPDATE `score` = ?, `rated_at` = NOW()");
    p.setInt(1, 1000);
    p.setInt(2, 1);
    p.setInt(3, 5);
    p.setInt(4, 5);
    p.executeUpdate();
} catch (SQLException e) {
    e.printStackTrace();
}
```

4.6.2.4 批量更新

需要更新表中多行的数据，可选择使用 UPDATE，并使用 WHERE 子句过滤需要更新的数据。

但如果你需要更新大量行 (数万或更多) 的时候，建议使用一个迭代，每次都只更新一部分数据，直到更新全部完成。这是因为 TiDB 单个事务大小限制为txn-total-size-limit（默认为 100MB），且一次性过多的数据更新，将导致持有锁时间过长（悲观事务），或产生大量冲突（乐观事务）。你可以在程序或脚本中使用循环来完成操作。

本页提供了编写脚本来处理循环更新的示例，该示例演示了应如何进行 SELECT 和 UPDATE 的组合，完成循环更新。

4.6.2.4.1 编写批量更新循环

首先，你应在你的应用或脚本的循环中，编写一个 SELECT 查询。这个查询的返回值可以作为需要更新的行的主键。需要注意的是，定义这个 SELECT 查询时，需要注意使用 WHERE 子句过滤需要更新的行。

4.6.2.4.2 例子

假设在过去的一年里，用户在 bookshop 网站进行了大量的书籍打分，但是原本设计为 5 分制的评分导致书籍评分的区分度不够，大量书籍评分集中在 3 分附近，因此，决定将 5 分制改为 10 分制。用来增大书籍评分的区分度。

这时需要对 ratings 表内之前 5 分制的数据进行乘 2 操作，同时需向 ratings 表内添加一个新列，以指示行是否已经被更新了。使用此列，可以在 SELECT 中过滤掉已经更新的行，这将防止脚本崩溃时对行进行多次更新，导致不合理的数据出现。

例如，你可以创建一个名为 ten_point，数据类型为BOOL 的列作为是否为 10 分制的标识：

```
ALTER TABLE `bookshop`.`ratings` ADD COLUMN `ten_point` BOOL NOT NULL DEFAULT FALSE;
```

> **注意：**
>
> 此批量更新程序将使用 DDL 语句将进行数据表的模式更改。TiDB 的所有 DDL 变更操作全部都是在线进行的，可查看此处，了解此处使用的ADD COLUMN 语句。

在 Golang 中，批量更新程序类似于以下内容：

```
package main

import (
    "database/sql"
    "fmt"
    _ "github.com/go-sql-driver/mysql"
    "strings"
    "time"
)

func main() {
    db, err := sql.Open("mysql", "root:@tcp(127.0.0.1:4000)/bookshop")
    if err != nil {
        panic(err)
    }
    defer db.Close()

    bookID, userID := updateBatch(db, true, 0, 0)
    fmt.Println("first time batch update success")
    for {
        time.Sleep(time.Second)
        bookID, userID = updateBatch(db, false, bookID, userID)
        fmt.Printf("batch update success, [bookID] %d, [userID] %d\n", bookID, userID)
    }
}

// updateBatch select at most 1000 lines data to update score
func updateBatch(db *sql.DB, firstTime bool, lastBookID, lastUserID int64) (bookID, userID int64)
    ↪  {
    // select at most 1000 primary keys in five-point scale data
    var err error
    var rows *sql.Rows

    if firstTime {
```

```
        rows, err = db.Query("SELECT `book_id`, `user_id` FROM `bookshop`.`ratings` " +
            "WHERE `ten_point` != true ORDER BY `book_id`, `user_id` LIMIT 1000")
    } else {
        rows, err = db.Query("SELECT `book_id`, `user_id` FROM `bookshop`.`ratings` "+
            "WHERE `ten_point` != true AND `book_id` > ? AND `user_id` > ? "+
            "ORDER BY `book_id`, `user_id` LIMIT 1000", lastBookID, lastUserID)
    }

    if err != nil || rows == nil {
        panic(fmt.Errorf("error occurred or rows nil: %+v", err))
    }

    // joint all id with a list
    var idList []interface{}
    for rows.Next() {
        var tempBookID, tempUserID int64
        if err := rows.Scan(&tempBookID, &tempUserID); err != nil {
            panic(err)
        }
        idList = append(idList, tempBookID, tempUserID)
        bookID, userID = tempBookID, tempUserID
    }

    bulkUpdateSql := fmt.Sprintf("UPDATE `bookshop`.`ratings` SET `ten_point` = true, "+
        "`score` = `score` * 2 WHERE (`book_id`, `user_id`) IN (%s)", placeHolder(len(idList)))
    db.Exec(bulkUpdateSql, idList...)

    return bookID, userID
}

// placeHolder format SQL place holder
func placeHolder(n int) string {
    holderList := make([]string, n/2, n/2)
    for i := range holderList {
        holderList[i] = "(?,?)"
    }
    return strings.Join(holderList, ",")
}
```

每次迭代中，SELECT 按主键顺序进行查询，最多选择 1000 行未更新到 10 分制（`ten_point` 为 `false`）数据的主键值。每次 SELECT 都会选择比上一次 SELECT 结果的最大主键还要大的数据，防止重复。然后，使用批量更新的方式，对其 `score` 列乘 2，并且将 `ten_point` 设为 `true`，更新 `ten_point` 的意义是在于防止更新程序崩溃重启后，反复更新同一行数据，导致数据损坏。每次循环中的 `time.Sleep(time.Second)` 将使得更新程序暂停 1 秒，防止批量更新程序占用过多的硬件资源。

在 Java (JDBC) 中，批量更新程序类似于以下内容：

Java 代码部分：

```
package com.pingcap.bulkUpdate;

import com.mysql.cj.jdbc.MysqlDataSource;

import java.sql.*;
import java.util.LinkedList;
import java.util.List;
import java.util.concurrent.TimeUnit;

public class BatchUpdateExample {
    static class UpdateID {
        private Long bookID;
        private Long userID;

        public UpdateID(Long bookID, Long userID) {
            this.bookID = bookID;
            this.userID = userID;
        }

        public Long getBookID() {
            return bookID;
        }

        public void setBookID(Long bookID) {
            this.bookID = bookID;
        }

        public Long getUserID() {
            return userID;
        }

        public void setUserID(Long userID) {
            this.userID = userID;
        }

        @Override
        public String toString() {
            return "[bookID] " + bookID + ", [userID] " + userID ;
        }
    }

    public static void main(String[] args) throws InterruptedException {
        // Configure the example database connection.
```

```
        // Create a mysql data source instance.
        MysqlDataSource mysqlDataSource = new MysqlDataSource();

        // Set server name, port, database name, username and password.
        mysqlDataSource.setServerName("localhost");
        mysqlDataSource.setPortNumber(4000);
        mysqlDataSource.setDatabaseName("bookshop");
        mysqlDataSource.setUser("root");
        mysqlDataSource.setPassword("");

        UpdateID lastID = batchUpdate(mysqlDataSource, null);

        System.out.println("first time batch update success");
        while (true) {
            TimeUnit.SECONDS.sleep(1);
            lastID = batchUpdate(mysqlDataSource, lastID);
            System.out.println("batch update success, [lastID] " + lastID);
        }
    }

    public static UpdateID batchUpdate (MysqlDataSource ds, UpdateID lastID) {
        try (Connection connection = ds.getConnection()) {
            UpdateID updateID = null;

            PreparedStatement selectPs;

            if (lastID == null) {
                selectPs = connection.prepareStatement(
                        "SELECT `book_id`, `user_id` FROM `bookshop`.`ratings` " +
                        "WHERE `ten_point` != true ORDER BY `book_id`, `user_id` LIMIT 1000");
            } else {
                selectPs = connection.prepareStatement(
                        "SELECT `book_id`, `user_id` FROM `bookshop`.`ratings` "+
                            "WHERE `ten_point` != true AND `book_id` > ? AND `user_id` > ? "+
                            "ORDER BY `book_id`, `user_id` LIMIT 1000");

                selectPs.setLong(1, lastID.getBookID());
                selectPs.setLong(2, lastID.getUserID());
            }

            List<Long> idList = new LinkedList<>();
            ResultSet res = selectPs.executeQuery();
            while (res.next()) {
                updateID = new UpdateID(
                        res.getLong("book_id"),
```

```
                        res.getLong("user_id")
                );
                idList.add(updateID.getBookID());
                idList.add(updateID.getUserID());
            }

            if (idList.isEmpty()) {
                System.out.println("no data should update");
                return null;
            }

            String updateSQL = "UPDATE `bookshop`.`ratings` SET `ten_point` = true, "+
                    "`score` = `score` * 2 WHERE (`book_id`, `user_id`) IN (" +
                    placeHolder(idList.size() / 2) + ")";
            PreparedStatement updatePs = connection.prepareStatement(updateSQL);
            for (int i = 0; i < idList.size(); i++) {
                updatePs.setLong(i + 1, idList.get(i));
            }
            int count = updatePs.executeUpdate();
            System.out.println("update " + count + " data");

            return updateID;
        } catch (SQLException e) {
            e.printStackTrace();
        }

        return null;
    }

    public static String placeHolder(int n) {
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < n ; i++) {
            sb.append(i == 0 ? "(?,?)" : ",(?,?)");
        }

        return sb.toString();
    }
}
```

hibernate.cfg.xml 配置部分：

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
```

```
    <session-factory>

        <!-- Database connection settings -->
        <property name="hibernate.connection.driver_class">com.mysql.cj.jdbc.Driver</property>
        <property name="hibernate.dialect">org.hibernate.dialect.TiDBDialect</property>
        <property name="hibernate.connection.url">jdbc:mysql://localhost:4000/movie</property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.connection.password"></property>
        <property name="hibernate.connection.autocommit">false</property>
        <property name="hibernate.jdbc.batch_size">20</property>

        <!-- Optional: Show SQL output for debugging -->
        <property name="hibernate.show_sql">true</property>
        <property name="hibernate.format_sql">true</property>
    </session-factory>
</hibernate-configuration>
```

每次迭代中，SELECT 按主键顺序进行查询，最多选择 1000 行未更新到 10 分制（ten_point 为 false）数据的主键值。每次 SELECT 都会选择比上一次 SELECT 结果的最大主键还要大的数据，防止重复。然后，使用批量更新的方式，对其 score 列乘 2，并且将 ten_point 设为 true，更新 ten_point 的意义是在于防止更新程序崩溃重启后，反复更新同一行数据，导致数据损坏。每次循环中的 TimeUnit.SECONDS.sleep(1); 将使得更新程序暂停 1 秒，防止批量更新程序占用过多的硬件资源。

4.6.3 删除数据

此页面将使用DELETE SQL 语句，对 TiDB 中的数据进行删除。如果需要周期性地删除过期数据，可以考虑使用 TiDB 的TTL 功能。

4.6.3.1 在开始之前

在阅读本页面之前，你需要准备以下事项：

- 使用 TiDB Serverless 构建 TiDB 集群。
- 阅读数据库模式概览，并创建数据库、创建表、创建二级索引。
- 需先插入数据才可删除。

4.6.3.2 SQL 语法

在 SQL 中，DELETE 语句一般为以下形式：

```
DELETE FROM {table} WHERE {filter}
```

参数	描述
{table}	表名
{filter}	过滤器匹配条件

此处仅展示 DELETE 的简单用法，详细文档可参考 TiDB 的DELETE 语法。

4.6.3.3 最佳实践

以下是删除行时需要遵循的一些最佳实践：

- 始终在删除语句中指定 WHERE 子句。如果 DELETE 没有 WHERE 子句，TiDB 将删除这个表内的所有行。
- 需要删除大量行(数万或更多)的时候，使用批量删除，这是因为 TiDB 单个事务大小限制为txn-total-size-limit (默认为 100MB)。
- 如果你需要删除表内的所有数据，请勿使用 DELETE 语句，而应该使用TRUNCATE 语句。
- 查看性能注意事项。
- 在需要大批量删除数据的场景下，非事务批量删除对性能的提升十分明显。但与之相对的，这将丢失删除的事务性，因此无法进行回滚，请务必正确进行操作选择。

4.6.3.4 例子

假设在开发中发现在特定时间段内，发生了业务错误，需要删除这期间内的所有rating 的数据，例如，2022-04-15 00:00:00 至 2022-04-15 00:15:00 的数据。此时，可使用 SELECT 语句查看需删除的数据条数：

```
SELECT COUNT(*) FROM `ratings` WHERE `rated_at` >= "2022-04-15 00:00:00" AND  `rated_at` <= "
    ↪ 2022-04-15 00:15:00";
```

- 若返回数量大于 1 万条，请参考批量删除。
- 若返回数量小于 1 万条，可参考下面的示例进行删除：

在 SQL 中，删除数据的示例如下：

```
DELETE FROM `ratings` WHERE `rated_at` >= "2022-04-15 00:00:00" AND  `rated_at` <= "2022-04-15
    ↪ 00:15:00";
```

在 Java 中，删除数据的示例如下：

```
// ds is an entity of com.mysql.cj.jdbc.MysqlDataSource

try (Connection connection = ds.getConnection()) {
    String sql = "DELETE FROM `bookshop`.`ratings` WHERE `rated_at` >= ? AND  `rated_at` <= ?";
    PreparedStatement preparedStatement = connection.prepareStatement(sql);
    Calendar calendar = Calendar.getInstance();
    calendar.set(Calendar.MILLISECOND, 0);

    calendar.set(2022, Calendar.APRIL, 15, 0, 0, 0);
    preparedStatement.setTimestamp(1, new Timestamp(calendar.getTimeInMillis()));

    calendar.set(2022, Calendar.APRIL, 15, 0, 15, 0);
    preparedStatement.setTimestamp(2, new Timestamp(calendar.getTimeInMillis()));
```

```
    preparedStatement.executeUpdate();
} catch (SQLException e) {
    e.printStackTrace();
}
```

在 Golang 中，删除数据的示例如下：

```
package main

import (
    "database/sql"
    "fmt"
    "time"

    _ "github.com/go-sql-driver/mysql"
)

func main() {
    db, err := sql.Open("mysql", "root:@tcp(127.0.0.1:4000)/bookshop")
    if err != nil {
        panic(err)
    }
    defer db.Close()

    startTime := time.Date(2022, 04, 15, 0, 0, 0, 0, time.UTC)
    endTime := time.Date(2022, 04, 15, 0, 15, 0, 0, time.UTC)

    bulkUpdateSql := fmt.Sprintf("DELETE FROM `bookshop`.`ratings` WHERE `rated_at` >= ? AND  `
        ↪ rated_at` <= ?")
    result, err := db.Exec(bulkUpdateSql, startTime, endTime)
    if err != nil {
        panic(err)
    }
    _, err = result.RowsAffected()
    if err != nil {
        panic(err)
    }
}
```

在 Python 中，删除数据的示例如下：

```
import MySQLdb
import datetime
import time

connection = MySQLdb.connect(
```

```
    host="127.0.0.1",
    port=4000,
    user="root",
    password="",
    database="bookshop",
    autocommit=True
)

with connection:
    with connection.cursor() as cursor:
        start_time = datetime.datetime(2022, 4, 15)
        end_time = datetime.datetime(2022, 4, 15, 0, 15)
        delete_sql = "DELETE FROM `bookshop`.`ratings` WHERE `rated_at` >= %s AND `rated_at` <= %
            ↪ s"
        affect_rows = cursor.execute(delete_sql, (start_time, end_time))
        print(f'delete {affect_rows} data')
```

> 注意：
>
> rated_at 字段为日期和时间类型 中的 DATETIME 类型，你可以认为它在 TiDB 保存时，存储为一个字面量，与时区无关。而 TIMESTAMP 类型，将会保存一个时间戳，从而在不同的时区配置时，展示不同的时间字符串。
>
> 另外，和 MySQL 一样，TIMESTAMP 数据类型受 2038 年问题的影响。如果存储的值大于 2038，建议使用 DATETIME 类型。

4.6.3.5 性能注意事项

4.6.3.5.1 TiDB GC 机制

DELETE 语句运行之后 TiDB 并非立刻删除数据，而是将这些数据标记为可删除。然后等待 TiDB GC (Garbage Collection) 来清理不再需要的旧数据。因此，你的 DELETE 语句并不会立即减少磁盘用量。

GC 在默认配置中，为 10 分钟触发一次，每次 GC 都会计算出一个名为 safe_point 的时间点，这个时间点前的数据，都不会再被使用到，因此，TiDB 可以安全的对数据进行清除。

GC 的具体实现方案和细节此处不再展开，请参考GC 机制简介 了解更详细的 GC 说明。

4.6.3.5.2 更新统计信息

TiDB 使用统计信息来决定索引的选择，因此，在大批量的数据删除之后，很有可能会导致索引选择不准确的情况发生。你可以使用手动收集的办法，更新统计信息。用以给 TiDB 优化器以更准确的统计信息来提供 SQL 性能优化。

4.6.3.6 批量删除

需要删除表中多行的数据，可选择DELETE 示例，并使用 WHERE 子句过滤需要删除的数据。

但如果你需要删除大量行（数万或更多）的时候，建议使用一个迭代，每次都只删除一部分数据，直到删除全部完成。这是因为 TiDB 单个事务大小限制为txn-total-size-limit（默认为 100MB）。你可以在程序或脚本中使用循环来完成操作。

本页提供了编写脚本来处理循环删除的示例，该示例演示了应如何进行 SELECT 和 DELETE 的组合，完成循环删除。

4.6.3.6.1 编写批量删除循环

在你的应用或脚本的循环中，编写一个 DELETE 语句，使用 WHERE 子句过滤需要删除的行，并使用 LIMIT 限制单次删除的数据条数。

4.6.3.6.2 批量删除例子

假设发现在特定时间段内，发生了业务错误，需要删除这期间内的所有rating 的数据，例如，2022-04-15 ↪ 00:00:00 至 2022-04-15 00:15:00 的数据。并且在 15 分钟内，有大于 1 万条数据被写入，此时请使用循环删除的方式进行删除：

在 Java 中，批量删除程序类似于以下内容：

```
package com.pingcap.bulkDelete;

import com.mysql.cj.jdbc.MysqlDataSource;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.sql.Timestamp;
import java.util.Calendar;
import java.util.concurrent.TimeUnit;

public class BatchDeleteExample
{
    public static void main(String[] args) throws InterruptedException {
        // Configure the example database connection.

        // Create a mysql data source instance.
        MysqlDataSource mysqlDataSource = new MysqlDataSource();

        // Set server name, port, database name, username and password.
        mysqlDataSource.setServerName("localhost");
        mysqlDataSource.setPortNumber(4000);
        mysqlDataSource.setDatabaseName("bookshop");
        mysqlDataSource.setUser("root");
        mysqlDataSource.setPassword("");
```

```
        Integer updateCount = -1;
        while (updateCount != 0) {
            updateCount = batchDelete(mysqlDataSource);
        }
    }

    public static Integer batchDelete (MysqlDataSource ds) {
        try (Connection connection = ds.getConnection()) {
            String sql = "DELETE FROM `bookshop`.`ratings` WHERE `rated_at` >= ? AND  `rated_at`
                ↪ <= ? LIMIT 1000";
            PreparedStatement preparedStatement = connection.prepareStatement(sql);
            Calendar calendar = Calendar.getInstance();
            calendar.set(Calendar.MILLISECOND, 0);

            calendar.set(2022, Calendar.APRIL, 15, 0, 0, 0);
            preparedStatement.setTimestamp(1, new Timestamp(calendar.getTimeInMillis()));

            calendar.set(2022, Calendar.APRIL, 15, 0, 15, 0);
            preparedStatement.setTimestamp(2, new Timestamp(calendar.getTimeInMillis()));

            int count = preparedStatement.executeUpdate();
            System.out.println("delete " + count + " data");

            return count;
        } catch (SQLException e) {
            e.printStackTrace();
        }

        return -1;
    }
}
```

每次迭代中，DELETE 最多删除 1000 行时间段为 2022-04-15 00:00:00 至 2022-04-15 00:15:00 的数据。

在 Golang 中，批量删除程序类似于以下内容：

```
package main

import (
    "database/sql"
    "fmt"
    "time"

    _ "github.com/go-sql-driver/mysql"
)
```

```
func main() {
    db, err := sql.Open("mysql", "root:@tcp(127.0.0.1:4000)/bookshop")
    if err != nil {
        panic(err)
    }
    defer db.Close()

    affectedRows := int64(-1)
    startTime := time.Date(2022, 04, 15, 0, 0, 0, 0, time.UTC)
    endTime := time.Date(2022, 04, 15, 0, 15, 0, 0, time.UTC)

    for affectedRows != 0 {
        affectedRows, err = deleteBatch(db, startTime, endTime)
        if err != nil {
            panic(err)
        }
    }
}

// deleteBatch delete at most 1000 lines per batch
func deleteBatch(db *sql.DB, startTime, endTime time.Time) (int64, error) {
    bulkUpdateSql := fmt.Sprintf("DELETE FROM `bookshop`.`ratings` WHERE `rated_at` >= ? AND  `
        ↪ rated_at` <= ? LIMIT 1000")
    result, err := db.Exec(bulkUpdateSql, startTime, endTime)
    if err != nil {
        return -1, err
    }
    affectedRows, err := result.RowsAffected()
    if err != nil {
        return -1, err
    }

    fmt.Printf("delete %d data\n", affectedRows)
    return affectedRows, nil
}
```

每次迭代中，DELETE 最多删除 1000 行时间段为 2022-04-15 00:00:00 至 2022-04-15 00:15:00 的数据。

在 Python 中，批量删除程序类似于以下内容：

```
import MySQLdb
import datetime
import time

connection = MySQLdb.connect(
```

```
    host="127.0.0.1",
    port=4000,
    user="root",
    password="",
    database="bookshop",
    autocommit=True
)

with connection:
    with connection.cursor() as cursor:
        start_time = datetime.datetime(2022, 4, 15)
        end_time = datetime.datetime(2022, 4, 15, 0, 15)
        affect_rows = -1
        while affect_rows != 0:
            delete_sql = "DELETE FROM `bookshop`.`ratings` WHERE `rated_at` >= %s AND  `rated_at`
                ↪  <= %s LIMIT 1000"
            affect_rows = cursor.execute(delete_sql, (start_time, end_time))
            print(f'delete {affect_rows} data')
            time.sleep(1)
```

每次迭代中，DELETE 最多删除 1000 行时间段为 2022-04-15 00:00:00 至 2022-04-15 00:15:00 的数据。

4.6.3.7 非事务批量删除

> **注意：**
>
> TiDB 从 v6.1.0 版本开始支持非事务 DML 语句特性。在 TiDB v6.1.0 以下版本中无法使用此特性。

4.6.3.7.1 使用前提

在使用非事务批量删除前，请先仔细阅读非事务 DML 语句。非事务批量删除，本质是以牺牲事务的原子性、隔离性为代价，增强批量数据处理场景下的性能和易用性。

因此在使用过程中，需要极为小心，否则，因为操作的非事务特性，在误操作时会导致严重的后果（如数据丢失等）。

4.6.3.7.2 非事务批量删除 SQL 语法

非事务批量删除的 SQL 语法如下：

```
BATCH ON {shard_column} LIMIT {batch_size} {delete_statement};
```

参数	描述
{shard_column}	非事务批量删除的划分列
{batch_size}	非事务批量删除的每批大小
{delete_statement}	删除语句

此处仅展示非事务批量删除的简单用法，详细文档可参考 TiDB 的非事务 DML 语句。

4.6.3.7.3 非事务批量删除使用示例

以上方批量删除例子场景为例，可使用以下 SQL 语句进行非事务批量删除：

```
BATCH ON `rated_at` LIMIT 1000 DELETE FROM `ratings` WHERE `rated_at` >= "2022-04-15 00:00:00"
    ↪ AND `rated_at` <= "2022-04-15 00:15:00";
```

4.6.4 使用 TTL (Time to Live) 定期删除过期数据

Time to Live (TTL) 提供了行级别的生命周期控制策略。通过为表设置 TTL 属性，TiDB 可以周期性地自动检查并清理表中的过期数据。此功能在一些场景可以有效节省存储空间、提升性能。

TTL 常见的使用场景：

- 定期删除验证码、短网址记录
- 定期删除不需要的历史订单
- 自动删除计算的中间结果

TTL 设计的目标是在不影响在线读写负载的前提下，帮助用户周期性且及时地清理不需要的数据。TTL 会以表为单位，并发地分发不同的任务到不同的 TiDB Server 节点上，进行并行删除处理。TTL 并不保证所有过期数据立即被删除，也就是说即使数据过期了，客户端仍然有可能在这之后的一段时间内读到过期的数据，直到其真正的被后台处理任务删除。

4.6.4.1 语法

你可以通过CREATE TABLE 或ALTER TABLE 语句来配置表的 TTL 功能。

4.6.4.1.1 创建具有 TTL 属性的表

- 创建一个具有 TTL 属性的表：

```
CREATE TABLE t1 (
    id int PRIMARY KEY,
    created_at TIMESTAMP
) TTL = `created_at` + INTERVAL 3 MONTH;
```

上面的例子创建了一张表 t1，并指定了 created_at 为 TTL 的时间列，表示数据的创建时间。同时，它还通过 INTERVAL 3 MONTH 设置了表中行的最长存活时间为 3 个月。超过了此时长的过期数据会在之后被删除。

- 设置 TTL_ENABLE 属性来开启或关闭清理过期数据的功能：

```
CREATE TABLE t1 (
    id int PRIMARY KEY,
    created_at TIMESTAMP
) TTL = `created_at` + INTERVAL 3 MONTH TTL_ENABLE = 'OFF';
```

如果 TTL_ENABLE 被设置成了 OFF，则即使设置了其他 TTL 选项，当前表也不会自动清理过期数据。对于一个设置了 TTL 属性的表，TTL_ENABLE 在缺省条件下默认为 ON。

- 为了与 MySQL 兼容，你也可以使用注释语法来设置 TTL：

```
CREATE TABLE t1 (
    id int PRIMARY KEY,
    created_at TIMESTAMP
) /*T![ttl] TTL = `created_at` + INTERVAL 3 MONTH TTL_ENABLE = 'OFF'*/;
```

在 TiDB 环境中，使用表的 TTL 属性和注释语法来配置 TTL 是等价的。在 MySQL 环境中，会自动忽略注释中的内容，并创建普通的表。

4.6.4.1.2 修改表的 TTL 属性

- 修改表的 TTL 属性：

```
ALTER TABLE t1 TTL = `created_at` + INTERVAL 1 MONTH;
```

上面的语句既支持修改已配置 TTL 属性的表，也支持为一张非 TTL 的表添加 TTL 属性。

- 单独修改 TTL 表的 TTL_ENABLE 值：

```
ALTER TABLE t1 TTL_ENABLE = 'OFF';
```

- 清除一张表的所有 TTL 属性：

```
ALTER TABLE t1 REMOVE TTL;
```

4.6.4.1.3 TTL 和数据类型的默认值

TTL 可以和数据类型的默认值一起使用。以下是两种常见的用法示例：

- 使用 DEFAULT CURRENT_TIMESTAMP 来指定某一列的默认值为该行的创建时间，并用这一列作为 TTL 的时间列，创建时间超过 3 个月的数据将被标记为过期：

```
CREATE TABLE t1 (
    id int PRIMARY KEY,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
) TTL = `created_at` + INTERVAL 3 MONTH;
```

- 指定某一列的默认值为该行的创建时间或更新时间，并用这一列作为 TTL 的时间列，创建时间或更新时间超过 3 个月的数据将被标记为过期：

```
CREATE TABLE t1 (
    id int PRIMARY KEY,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
) TTL = `created_at` + INTERVAL 3 MONTH;
```

4.6.4.1.4 TTL 和生成列

TTL 可以和生成列（实验特性）一起使用，用来表达更加复杂的过期规则。例如：

```
CREATE TABLE message (
    id int PRIMARY KEY,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    image bool,
    expire_at TIMESTAMP AS (IF(image,
            created_at + INTERVAL 5 DAY,
            created_at + INTERVAL 30 DAY
    ))
) TTL = `expire_at` + INTERVAL 0 DAY;
```

上述语句的消息以 expire_at 列来作为过期时间，并按照消息类型来设定。如果是图片，则 5 天后过期，不然就 30 天后过期。

TTL 还可以和JSON 类型 一起使用。例如：

```
CREATE TABLE orders (
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    order_info JSON,
    created_at DATE AS (JSON_EXTRACT(order_info, '$.created_at')) VIRTUAL
) TTL = `created_at` + INTERVAL 3 month;
```

4.6.4.2 TTL 任务

对于每张设置了 TTL 属性的表，TiDB 内部会定期调度后台任务来清理过期的数据。你可以通过给表设置 TTL_JOB_INTERVAL 属性来自定义任务的执行周期，比如通过下面的语句将后台清理任务设置为每 24 小时执行一次：

```
ALTER TABLE orders TTL_JOB_INTERVAL = '24h';
```

TTL_JOB_INTERVAL 的默认值是 1h。

在执行 TTL 任务时，TiDB 会基于 Region 的数量将表拆分为最多 64 个子任务。这些子任务会被分发到不同的 TiDB 节点中执行。你可以通过设置系统变量tidb_ttl_running_tasks 来限制整个集群中同时执行的 TTL 子任务数量。然而，并非所有表的 TTL 任务都可以被拆分为子任务。请参考使用限制以了解哪些表的 TTL 任务不能被拆分。

如果想禁止 TTL 任务的执行，除了可以设置表属性 TTL_ENABLE='OFF' 外，也可以通过设置全局变量 tidb_ttl_job_enable 关闭整个集群的 TTL 任务的执行。

```
SET @@global.tidb_ttl_job_enable = OFF;
```

在某些场景下，你可能希望只允许在每天的某个时间段内调度后台的 TTL 任务，此时可以设置全局变量tidb_ttl_job_schedule_window_start_time 和tidb_ttl_job_schedule_window_end_time 来指定时间窗口，比如：

```
SET @@global.tidb_ttl_job_schedule_window_start_time = '01:00 +0000';
SET @@global.tidb_ttl_job_schedule_window_end_time = '05:00 +0000';
```

上述语句只允许在 UTC 时间的凌晨 1 点到 5 点调度 TTL 任务。默认情况下的时间窗口设置为 00:00 +0000 到 23:59 +0000，即允许所有时段的任务调度。

4.6.4.3 TTL 的可观测性

TiDB 会定时采集 TTL 的运行时信息，并在 Grafana 中提供了相关指标的可视化图表。你可以在 TiDB -> TTL 的面板下看到这些信息。指标详情见TiDB 重要监控指标详解 中的 TTL 部分。

同时，可以通过以下三个系统表获得 TTL 任务执行的更多信息：

- mysql.tidb_ttl_table_status 表中包含了所有 TTL 表的上一次执行与正在执行的 TTL 任务的信息。以其中一行为例：

```
MySQL [(none)]> SELECT * FROM mysql.tidb_ttl_table_status LIMIT 1\G;
*************************** 1. row ***************************
                      table_id: 85
                 parent_table_id: 85
                 table_statistics: NULL
                     last_job_id: 0b4a6d50-3041-4664-9516-5525ee6d9f90
           last_job_start_time: 2023-02-15 20:43:46
           last_job_finish_time: 2023-02-15 20:44:46
           last_job_ttl_expire: 2023-02-15 19:43:46
               last_job_summary: {"total_rows":4369519,"success_rows":4369519,"error_rows":0,
                   ↪ "total_scan_task":64,"scheduled_scan_task":64,"finished_scan_task":64}
                 current_job_id: NULL
           current_job_owner_id: NULL
         current_job_owner_addr: NULL
    current_job_owner_hb_time: NULL
         current_job_start_time: NULL
         current_job_ttl_expire: NULL
             current_job_state: NULL
             current_job_status: NULL
current_job_status_update_time: NULL
1 row in set (0.040 sec)
```

其中列 table_id 为分区表 ID，而 parent_table_id 为表的 ID，与 infomation_schema.tables 表中的 ID 对应。如果表不是分区表，则 table_id 与 parent_table_id 总是相等。

列 {last, current}_job_{start_time, finish_time, ttl_expire} 分别描述了过去和当前 TTL 任务的开始时间、结束时间和过期时间。last_job_summary 列描述了上一次 TTL 任务的执行情况，包括总行数、成功行数、失败行数。

- mysql.tidb_ttl_task 表中包含了正在执行的 TTL 子任务。单个 TTL 任务会被拆分为多个子任务，该表中记录了正在执行的这些子任务的信息。
- mysql.tidb_ttl_job_history 表中记录了 TTL 任务的执行历史。TTL 任务的历史记录将被保存 90 天。以一行为例：

```
MySQL [(none)]> SELECT * FROM mysql.tidb_ttl_job_history LIMIT 1\G;
*************************** 1. row ***************************
          job_id: f221620c-ab84-4a28-9d24-b47ca2b5a301
        table_id: 85
  parent_table_id: 85
    table_schema: test_schema
      table_name: TestTable
  partition_name: NULL
     create_time: 2023-02-15 17:43:46
     finish_time: 2023-02-15 17:45:46
     ttl_expire: 2023-02-15 16:43:46
    summary_text: {"total_rows":9588419,"success_rows":9588419,"error_rows":0,"
        ↪ total_scan_task":63,"scheduled_scan_task":63,"finished_scan_task":63}
    expired_rows: 9588419
    deleted_rows: 9588419
error_delete_rows: 0
          status: finished
```

其中列 table_id 为分区表 ID，而 parent_table_id 为表的 ID，与 infomation_schema.tables 表中的 ID 对应。table_schema、table_name、partition_name 分别对应表示数据库、表名、分区名。create_time、finish_time、ttl_expire 分别表示 TTL 任务的创建时间、结束时间和过期时间。expired_rows 与 deleted_rows 表示过期行数与成功删除的行数。

4.6.4.4 TiDB 数据迁移工具兼容性

TTL 功能能够与 TiDB 的迁移、备份、恢复工具一同使用。

工具名称	最低兼容版本	说明
Backup & Restore (BR)	v6.6.0	恢复时会自动将表的 TTL_ENABLE ↪ 属性设置为 OFF，关闭 TTL。这样可以防止 TiDB 在备份恢复后立即删除过期的数据。此时你需要手动重新配置 TTL_ENABLE ↪ 属性来重新开启各个表的 TTL。

工具名称	最低兼容版本	说明
TiDB Light-ning	v6.6.0	导入后如果表中有 TTL 属性，会自动将表的 TTL_ENABLE ↪ 属性设置为 OFF，关闭 TTL。这样可以防止 TiDB 在导入后立即删除过期的数据。此时你需要手动重新配置 TTL_ENABLE ↪ 属性来重新开启各个表的 TTL。

工具名称	最低兼容版本	说明
TiCDC	v7.0.0	上游的 TTL 删除将会同步至下游。因此，为了防止重复删除，下游表的 TTL_ENABLE ↪ 属性将被强制设置为 OFF。

4.6.4.5 与 TiDB 其他特性的兼容性

特性名称	说明
FLASHBACK TABLE	FLASHBACK TABLE 语句会将每个表的 TTL_ENABLE 属性强制设置为 OFF。这样可以防止 TiDB 在 FLASH-BACK 后立即删除过期的数据。此时你需要手动重新配置 TTL_ENABLE 属性来重新开启各个表的 TTL。

特性名称	说明
FLASHBACK DATABASE	FLASHBACK DATABASE 语句会将每个表的 TTL_ENABLE 属性强制设置为 OFF。这样可以防止 TiDB 在 FLASHBACK 后立即删除过期的数据。此时你需要手动重新配置 TTL_ENABLE 属性来重新开启各个表的 TTL。

特性名称	说明
FLASHBACK CLUSTER TO TIMESTAMP	FLASHBACK CLUSTER TO TIMESTAMP 会将 TIDB_TTL_JOB_ENABLE 系统变量设置为 OFF，同时表的 TTL_ENABLE 属性将保持原样。

4.6.4.6 使用限制

目前，TTL 特性具有以下限制:

- 不允许在临时表上设置 TTL 属性，包括本地临时表和全局临时表。
- 具有 TTL 属性的表不支持作为外键约束的主表被其他表引用。
- 不保证所有过期数据立即被删除，过期数据被删除的时间取决于后台清理任务的调度周期和调度窗口。
- 对于使用聚簇索引的表，如果主键的类型不是整数类型或二进制字符串类型，TTL 任务将无法被拆分成多个子任务。这将导致 TTL 任务只能在一个 TiDB 节点上按顺序执行。如果表中的数据量较大，TTL 任务的执行可能会变得缓慢。
- TTL 无法在 TiDB Serverless 集群上使用。

4.6.4.7 常见问题

- 如何判断删除的速度是否够快，能够保持数据总量相对稳定？

  在Grafana TiDB 面板中，监控项 `TTL Insert Rows Per Hour` 记录了前一小时总共插入数据的数量。相应的 `TTL Delete Rows Per Hour` 记录了前一小时 TTL 任务总共删除的数据总量。如果 `TTL Insert Rows Per Hour` 长期高于 `TTL Delete Rows Per Hour`，说明插入的速度高于删除的速度，数据总量将会上升。例如：

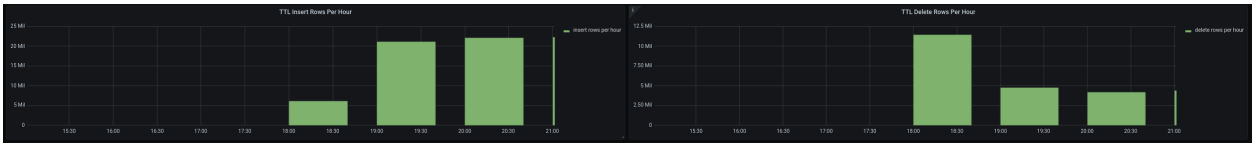


图 9: insert fast example

值得注意的是，由于 TTL 并不能保证数据立即被删除，且当前插入的数据将会在将来的 TTL 任务中才会被删除，哪怕短时间内 TTL 删除的速度低于插入的速度，也不能说明 TTL 的效率一定过慢。需要结合具体情况分析。

- 如何判断 TTL 任务的瓶颈在扫描还是删除？

  观察面板中 `TTL Scan Worker Time By Phase` 与 `TTL Delete Worker Time By Phase` 监控项。如果 scan worker 处于 `dispatch` 状态的时间有很大占比，且 delete worker 很少处于 `idle` 状态，那么说明 scan worker 在等待 delete worker 完成删除工作，如果此时集群资源仍然较为宽松，可以考虑提高 `tidb_ttl_delete_worker_count` 来提高删除的 worker 数量。例如：

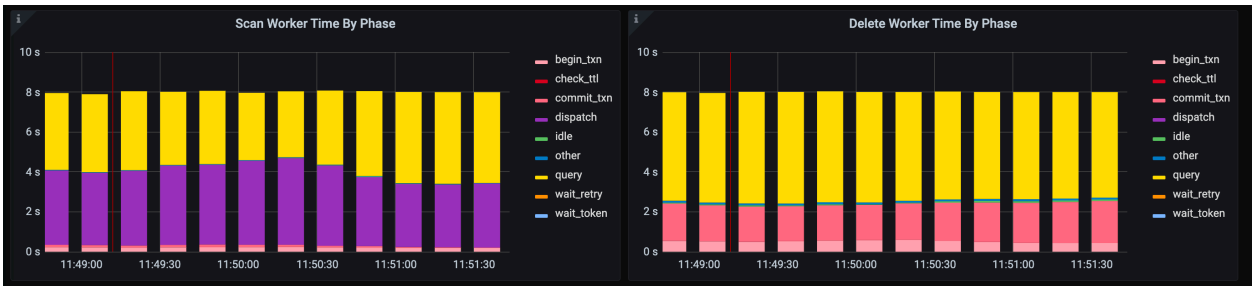


图 10: scan fast example

与之相对，如果 scan worker 很少处于 `dispatch` 的状态，且 delete worker 长期处于 `idle` 阶段，那么说明 delete worker 闲置，且 scan worker 较为忙碌。例如：



图 11: delete fast example

TTL 任务中扫描与删除的占比与机器配置、数据分布都有关系，所以每一时刻的数据只能代表正在执行的 TTL Job 的情况。用户可以通过查询表 `mysql.tidb_ttl_job_history` 来判断某一时刻运行的 TTL Job 对应哪一张表。

- 如何合理配置 `tidb_ttl_scan_worker_count` 和 `tidb_ttl_delete_worker_count`？

  1. 可以参考问题“如何判断 TTL 任务的瓶颈在扫描还是删除？”来考虑提升 `tidb_ttl_scan_worker_count` ↪ 还是 `tidb_ttl_delete_worker_count`。

2. 如果 TiKV 节点数量较多，提升 `tidb_ttl_scan_worker_count` 能够使 TTL 任务负载更加均匀。

由于过高的 TTL worker 数量将会造成较大的压力，所以需要综合观察 TiDB 的 CPU 水平与 TiKV 的磁盘与 CPU 使用量。根据不同场景和需求（需要尽量加速 TTL，或是需要减少 TTL 对其他请求的影响）来调整 `tidb_ttl_scan_worker_count` 与 `tidb_ttl_delete_worker_count`，从而提升 TTL 扫描和删除数据的速度，或降低 TTL 任务对性能的影响。

4.6.5 预处理语句

预处理语句是一种将多个仅有参数不同的 SQL 语句进行模板化的语句，它让 SQL 语句与参数进行了分离。可以用它提升 SQL 语句的：

- 安全性：因为参数和语句已经分离，所以避免了 SQL 注入攻击的风险。
- 性能：因为语句在 TiDB 端被预先解析，后续执行只需要传递参数，节省了完整 SQL 解析、拼接 SQL 语句字符串以及网络传输的代价。

在大部分的应用程序中，SQL 语句是可以被枚举的，可以使用有限个 SQL 语句来完成整个应用程序的数据查询，所以使用预处理语句是最佳实践之一。

4.6.5.1 SQL 语法

本节将介绍创建、使用及删除预处理语句的 SQL 语法。

4.6.5.1.1 创建预处理语句

```
PREPARE {prepared_statement_name} FROM '{prepared_statement_sql}';
```

参数	描述
{prepared_statement_name}	预处理语句名称
{prepared_statement_sql}	预处理语句 SQL，以英文半角问号做占位符

你可查看PREPARE 语句 获得更多信息。

4.6.5.1.2 使用预处理语句

预处理语句仅可使用用户变量作为参数，因此，需先使用SET 语句 设置变量后，供EXECUTE 语句 调用预处理语句。

```
SET @{parameter_name} = {parameter_value};
EXECUTE {prepared_statement_name} USING @{parameter_name};
```

参数	描述
{parameter_name}	用户参数名
{parameter_value}	用户参数值

参数	描述
{prepared_statement_name}	预处理语句名称，需和创建预处理语句中定义的名称一致

你可查看EXECUTE 语句 获得更多信息。

4.6.5.1.3 删除预处理语句

```
DEALLOCATE PREPARE {prepared_statement_name};
```

参数	描述
{prepared_statement_name}	预处理语句名称，需和创建预处理语句中定义的名称一致

你可查看DEALLOCATE 语句 获得更多信息。

4.6.5.2 例子

本节以使用预处理语句，完成查询数据和插入数据两个场景的示例。

4.6.5.2.1 查询示例

例如，需要查询Bookshop 应用 中，id 为 1 的书籍信息。

使用 SQL 查询示例：

```
PREPARE `books_query` FROM 'SELECT * FROM `books` WHERE `id` = ?';
```

运行结果为：

```
Query OK, 0 rows affected (0.01 sec)
```

```
SET @id = 1;
```

运行结果为：

```
Query OK, 0 rows affected (0.04 sec)
```

```
EXECUTE `books_query` USING @id;
```

运行结果为：

```
+---------+---------------------------------+--------+---------------------+-------+--------+
| id      | title                           | type   | published_at        | stock | price  |
+---------+---------------------------------+--------+---------------------+-------+--------+
| 1       | The Adventures of Pierce Wehner | Comics | 1904-06-06 20:46:25 |   586 | 411.66 |
+---------+---------------------------------+--------+---------------------+-------+--------+
1 row in set (0.05 sec)
```

使用 Java 查询示例：

```
// ds is an entity of com.mysql.cj.jdbc.MysqlDataSource
try (Connection connection = ds.getConnection()) {
    PreparedStatement preparedStatement = connection.prepareStatement("SELECT * FROM `books`
        ↪ WHERE `id` = ?");
    preparedStatement.setLong(1, 1);

    ResultSet res = preparedStatement.executeQuery();
    if(!res.next()) {
        System.out.println("No books in the table with id 1");
    } else {
        // got book's info, which id is 1
        System.out.println(res.getLong("id"));
        System.out.println(res.getString("title"));
        System.out.println(res.getString("type"));
    }
} catch (SQLException e) {
    e.printStackTrace();
}
```

4.6.5.2.2 插入示例

还是使用books 表 为例，需要插入一个 `title` 为 `TiDB Developer Guide`, `type` 为 `Science & Technology`, `stock` ↪ 为 `100`, `price` 为 `0.0`, `published_at` 为 插入的当前时间 的书籍信息。需要注意的是，`books` 表的主键包含 `AUTO_RANDOM` 属性，无需指定它。如果你对插入数据还不了解，可以在插入数据一节了解更多数据插入的相关信息。

使用 SQL 插入数据示例如下：

```
PREPARE `books_insert` FROM 'INSERT INTO `books` (`title`, `type`, `stock`, `price`, `
    ↪ published_at`) VALUES (?, ?, ?, ?, ?);';
```

运行结果为：

```
Query OK, 0 rows affected (0.03 sec)
```

```
SET @title = 'TiDB Developer Guide';
SET @type = 'Science & Technology';
SET @stock = 100;
SET @price = 0.0;
SET @published_at = NOW();
```

运行结果为：

```
Query OK, 0 rows affected (0.04 sec)
```

```
EXECUTE `books_insert` USING @title, @type, @stock, @price, @published_at;
```

运行结果为：

```
Query OK, 1 row affected (0.03 sec)
```

使用 Java 插入数据示例如下：

```
try (Connection connection = ds.getConnection()) {
    String sql = "INSERT INTO `books` (`title`, `type`, `stock`, `price`, `published_at`) VALUES
        ↪ (?, ?, ?, ?, ?);";
    PreparedStatement preparedStatement = connection.prepareStatement(sql);

    preparedStatement.setString(1, "TiDB Developer Guide");
    preparedStatement.setString(2, "Science & Technology");
    preparedStatement.setInt(3, 100);
    preparedStatement.setBigDecimal(4, new BigDecimal("0.0"));
    preparedStatement.setTimestamp(5, new Timestamp(Calendar.getInstance().getTimeInMillis()));

    preparedStatement.executeUpdate();
} catch (SQLException e) {
    e.printStackTrace();
}
```

可以看到，JDBC 帮你管控了预处理语句的生命周期，而无需你在应用程序里手动使用预处理语句的创建、使用、删除等。但值得注意的是，因为 TiDB 兼容 MySQL 协议，在客户端使用 MySQL JDBC Driver 的过程中，其默认配置并非开启 服务端的预处理语句选项，而是使用客户端的预处理语句。你需要关注以下配置项，来获得在 JDBC 下 TiDB 服务端预处理语句的支持，及在你的使用场景下的最佳配置：

参数	作用	推荐场景	推荐配置
useServerPrepStmts	是否使用服务端开启预处理语句支持	在需要多次使用预处理语句时	true
cachePrepStmts	客户端是否缓存预处理语句	useServerPrepStmts=true 时	true
prepStmtCacheSqlLimit	预处理语句最大大小（默认 256 字符）	预处理语句大于 256 字符时	按实际预处理语句大小配
prepStmtCacheSize	预处理语句最大缓存数量（默认 25 条）	预处理语句数量大于 25 条时	按实际预处理语句数量配

在此处给出一个较为的通用场景的 JDBC 连接字符串配置，以 Host: 127.0.0.1，Port: 4000，用户: root，密码: 空，默认数据库: test为例：

```
jdbc:mysql://127.0.0.1:4000/test?user=root&useConfigs=maxPerformance&useServerPrepStmts=true&
    ↪ prepStmtCacheSqlLimit=2048&prepStmtCacheSize=256&rewriteBatchedStatements=true&
    ↪ allowMultiQueries=true
```

你也可以查看插入行一章，来查看是否需要在插入数据场景下更改其他 JDBC 的参数。

有关 Java 的完整示例，可参阅：

- TiDB 和 Java 的简单 CRUD 应用程序 - 使用 JDBC。
- TiDB 和 Java 的简单 CRUD 应用程序 - 使用 Hibernate。
- Build the TiDB Application using Spring Boot。

4.7 数据读取

4.7.1 单表查询

在这个章节当中，将开始介绍如何使用 SQL 来对数据库中的数据进行查询。

4.7.1.1 开始之前

下面将围绕Bookshop 这个应用程序来对 TiDB 的数据查询部分展开介绍。

在阅读本章节之前，你需要做以下准备工作：

1. 构建 TiDB 集群（推荐使用TiDB Cloud 或TiUP ）。
2. 导入 Bookshop 应用程序的表结构和示例数据。
3. 连接到 TiDB。

4.7.1.2 简单的查询

在 Bookshop 应用程序的数据库当中，`authors` 表存放了作家们的基础信息，可以通过 `SELECT ... FROM ...` 语句将数据从数据库当中调取出去。

在 MySQL Client 等客户端输入并执行如下 SQL 语句：

```
SELECT id, name FROM authors;
```

输出结果如下：

```
+------------+--------------------------+
| id         | name                     |
+------------+--------------------------+
|       6357 | Adelle Bosco             |
|     345397 | Chanelle Koepp           |
|     807584 | Clementina Ryan          |
|     839921 | Gage Huel                |
|     850070 | Ray Armstrong            |
|     850362 | Ford Waelchi             |
|     881210 | Jayme Gutkowski          |
|    1165261 | Allison Kuvalis          |
|    1282036 | Adela Funk               |
...
| 4294957408 | Lyla Nitzsche            |
+------------+--------------------------+
20000 rows in set (0.05 sec)
```

在 Java 语言当中，可以通过声明一个 Author 类来定义如何存放作者的基础信息，根据数据的类型和取值范围从 Java 语言当中选择合适的数据类型来存放对应的数据，例如：

- 使用 Int 类型变量存放 int 类型的数据。
- 使用 Long 类型变量存放 bigint 类型的数据。
- 使用 Short 类型变量存放 tinyint 类型的数据。
- 使用 String 类型变量存放 varchar 类型的数据。
- ...

```
public class Author {
    private Long id;
    private String name;
    private Short gender;
    private Short birthYear;
    private Short deathYear;

    public Author() {}

     // Skip the getters and setters.
}
```

```
public class AuthorDAO {

    // Omit initialization of instance variables.

    public List<Author> getAuthors() throws SQLException {
        List<Author> authors = new ArrayList<>();

        try (Connection conn = ds.getConnection()) {
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery("SELECT id, name FROM authors");
            while (rs.next()) {
                Author author = new Author();
                author.setId(rs.getLong("id"));
                author.setName(rs.getString("name"));
                authors.add(author);
            }
        }
        return authors;
    }
}
```

- 在获得数据库连接之后，你可以通过 conn.createStatement() 语句创建一个 Statement 实例对象。
- 然后调用 stmt.executeQuery("query_sql") 方法向 TiDB 发起一个数据库查询请求。
- 数据库返回的查询结果将会存放到 ResultSet 当中，通过遍历 ResultSet 对象可以将返回结果映射到此前准备的 Author 类对象当中。

4.7.1.3 对结果进行筛选

查询得到的结果非常多，但是并不都是你想要的？可以通过 WHERE 语句对查询的结果进行过滤，从而找到想要查询的部分。

例如，想要查找众多作家当中找出在 1998 年出生的作家：

在 SQL 中，可以使用 WHERE 子句添加筛选的条件：

```
SELECT * FROM authors WHERE birth_year = 1998;
```

对于 Java 程序而言，可以通过同一个 SQL 来处理带有动态参数的数据查询请求。

将参数拼接到 SQL 语句当中也许是一种方法，但是这可能不是一个好的主意，因为这会给应用程序带来潜在的 SQL 注入风险。

在处理这类查询时，应该使用PreparedStatement 来替代普通的 Statement。

```
public List<Author> getAuthorsByBirthYear(Short birthYear) throws SQLException {
    List<Author> authors = new ArrayList<>();
    try (Connection conn = ds.getConnection()) {
        PreparedStatement stmt = conn.prepareStatement("""
        SELECT * FROM authors WHERE birth_year = ?;
        """);
        stmt.setShort(1, birthYear);
        ResultSet rs = stmt.executeQuery();
        while (rs.next()) {
            Author author = new Author();
            author.setId(rs.getLong("id"));
            author.setName(rs.getString("name"));
            authors.add(author);
        }
    }
    return authors;
}
```

4.7.1.4 对结果进行排序

使用 ORDER BY 语句可以让查询结果按照期望的方式进行排序。

例如，可以通过下面的 SQL 语句对 authors 表的数据按照 birth_year 列进行降序 (DESC) 排序，从而得到最年轻的作家列表。

```
SELECT id, name, birth_year
FROM authors
ORDER BY birth_year DESC;
```

```
public List<Author> getAuthorsSortByBirthYear() throws SQLException {
    List<Author> authors = new ArrayList<>();
    try (Connection conn = ds.getConnection()) {
```

```
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery("""
            SELECT id, name, birth_year
            FROM authors
            ORDER BY birth_year DESC;
            """);

        while (rs.next()) {
            Author author = new Author();
            author.setId(rs.getLong("id"));
            author.setName(rs.getString("name"));
            author.setBirthYear(rs.getShort("birth_year"));
            authors.add(author);
        }
    }
    return authors;
}
```

查询结果如下：

```
+-----------+------------------------+------------+
| id        | name                   | birth_year |
+-----------+------------------------+------------+
| 83420726  | Terrance Dach          | 2000       |
| 57938667  | Margarita Christiansen | 2000       |
| 77441404  | Otto Dibbert           | 2000       |
| 61338414  | Danial Cormier         | 2000       |
| 49680887  | Alivia Lemke           | 2000       |
| 45460101  | Itzel Cummings         | 2000       |
| 38009380  | Percy Hodkiewicz       | 2000       |
| 12943560  | Hulda Hackett          | 2000       |
| 1294029   | Stanford Herman        | 2000       |
| 111453184 | Jeffrey Brekke         | 2000       |
...
300000 rows in set (0.23 sec)
```

4.7.1.5 限制查询结果数量

如果希望 TiDB 只返回部分结果，可以使用 `LIMIT` 语句限制查询结果返回的记录数。

```
SELECT id, name, birth_year
FROM authors
ORDER BY birth_year DESC
LIMIT 10;
```

```
public List<Author> getAuthorsWithLimit(Integer limit) throws SQLException {
    List<Author> authors = new ArrayList<>();
    try (Connection conn = ds.getConnection()) {
        PreparedStatement stmt = conn.prepareStatement("""
            SELECT id, name, birth_year
            FROM authors
            ORDER BY birth_year DESC
            LIMIT ?;
            """);
        stmt.setInt(1, limit);
        ResultSet rs = stmt.executeQuery();
        while (rs.next()) {
            Author author = new Author();
            author.setId(rs.getLong("id"));
            author.setName(rs.getString("name"));
            author.setBirthYear(rs.getShort("birth_year"));
            authors.add(author);
        }
    }
    return authors;
}
```

查询结果如下：

```
+-----------+------------------------+------------+
| id        | name                   | birth_year |
+-----------+------------------------+------------+
| 83420726  | Terrance Dach          | 2000       |
| 57938667  | Margarita Christiansen | 2000       |
| 77441404  | Otto Dibbert           | 2000       |
| 61338414  | Danial Cormier         | 2000       |
| 49680887  | Alivia Lemke           | 2000       |
| 45460101  | Itzel Cummings         | 2000       |
| 38009380  | Percy Hodkiewicz       | 2000       |
| 12943560  | Hulda Hackett          | 2000       |
| 1294029   | Stanford Herman        | 2000       |
| 111453184 | Jeffrey Brekke         | 2000       |
+-----------+------------------------+------------+
10 rows in set (0.11 sec)
```

通过观察查询结果你会发现，在使用 LIMIT 语句之后，查询的时间明显缩短，这是 TiDB 对 LIMIT 子句进行优化后的结果，你可以通过TopN 和 Limit 下推章节了解更多细节。

4.7.1.6 聚合查询

如果你想要关注数据整体的情况，而不是部分数据，你可以通过使用 `GROUP BY` 语句配合聚合函数，构建一个聚合查询来帮助你对数据的整体情况有一个更好的了解。

比如说，你希望知道哪些年出生的作家比较多，你可以将作家基本信息按照 `birth_year` 列进行分组，然后分别统计在当年出生的作家数量：

```
SELECT birth_year, COUNT(DISTINCT id) AS author_count
FROM authors
GROUP BY birth_year
ORDER BY author_count DESC;
```

```
public class AuthorCount {
    private Short birthYear;
    private Integer authorCount;

    public AuthorCount() {}

     // Skip the getters and setters.
}

public List<AuthorCount> getAuthorCountsByBirthYear() throws SQLException {
    List<AuthorCount> authorCounts = new ArrayList<>();
    try (Connection conn = ds.getConnection()) {
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery("""
            SELECT birth_year, COUNT(DISTINCT id) AS author_count
            FROM authors
            GROUP BY birth_year
            ORDER BY author_count DESC;
            """);

        while (rs.next()) {
            AuthorCount authorCount = new AuthorCount();
            authorCount.setBirthYear(rs.getShort("birth_year"));
            authorCount.setAuthorCount(rs.getInt("author_count"));
            authorCounts.add(authorCount);
        }
    }
    return authorCount;
}
```

查询结果如下：

```
+------------+--------------+
| birth_year | author_count |
+------------+--------------+
|       1932 |          317 |
```

```
|        1947 |           290 |
|        1939 |           282 |
|        1935 |           289 |
|        1968 |           291 |
|        1962 |           261 |
|        1961 |           283 |
|        1986 |           289 |
|        1994 |           280 |
...
|        1972 |           306 |
+------------+--------------+
71 rows in set (0.00 sec)
```

除了 COUNT 函数外，TiDB 还支持了其他聚合函数。详情请参考GROUP BY 聚合函数。

4.7.2 多表连接查询

很多时候，应用程序需要在一个查询当中使用到多张表的数据，这个时候可以通过 JOIN 语句将两张或多张表的数据组合在一起。

4.7.2.1 Join 类型

此节将详细叙述 Join 的连接类型。

4.7.2.1.1 内连接 INNER JOIN

内连接的连接结果只返回匹配连接条件的行。

例如，想要知道编写过最多书的作家是谁，需要将作家基础信息表 authors 与书籍作者表 book_authors 进行连接。

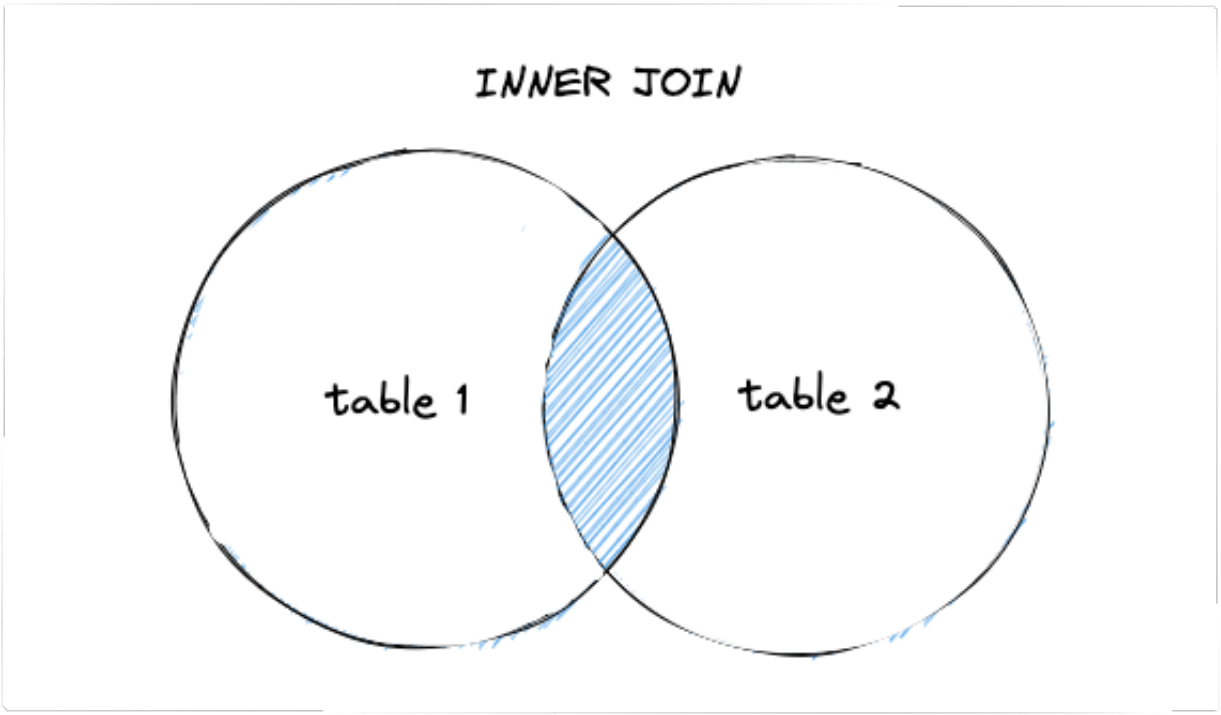


图 12: Inner Join

在下面的 SQL 语句当中，通过关键字 JOIN 声明要将左表 authors 和右表 book_authors 的数据行以内连接的方式进行连接，连接条件为 a.id = ba.author_id，那么连接的结果集当中将只会包含满足连接条件的行。假设有一个作家没有编写过任何书籍，那么他在 authors 表当中的记录将无法满足连接条件，因此也不会出现在结果集当中。

```
SELECT ANY_VALUE(a.id) AS author_id, ANY_VALUE(a.name) AS author_name, COUNT(ba.book_id) AS books
FROM authors a
JOIN book_authors ba ON a.id = ba.author_id
GROUP BY ba.author_id
ORDER BY books DESC
LIMIT 10;
```

查询结果如下：

```
+------------+----------------+-------+
| author_id  | author_name    | books |
+------------+----------------+-------+
|  431192671 | Emilie Cassin  |     7 |
|  865305676 | Nola Howell    |     7 |
|  572207928 | Lamar Koch     |     6 |
| 3894029860 | Elijah Howe    |     6 |
| 1150614082 | Cristal Stehr  |     6 |
| 4158341032 | Roslyn Rippin  |     6 |
```

```
| 2430691560 | Francisca Hahn |     6 |
| 3346415350 | Leta Weimann   |     6 |
| 1395124973 | Albin Cole     |     6 |
| 2768150724 | Caleb Wyman    |     6 |
+------------+----------------+-------+
10 rows in set (0.01 sec)
```

在 Java 中内连接的示例如下：

```
public List<Author> getTop10AuthorsOrderByBooks() throws SQLException {
    List<Author> authors = new ArrayList<>();
    try (Connection conn = ds.getConnection()) {
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery("""
        SELECT ANY_VALUE(a.id) AS author_id, ANY_VALUE(a.name) AS author_name, COUNT(ba.book_id)
            ↪ AS books
        FROM authors a
        JOIN book_authors ba ON a.id = ba.author_id
        GROUP BY ba.author_id
        ORDER BY books DESC
        LIMIT 10;
        """);
        while (rs.next()) {
            Author author = new Author();
            author.setId(rs.getLong("author_id"));
            author.setName(rs.getString("author_name"));
            author.setBooks(rs.getInt("books"));
            authors.add(author);
        }
    }
    return authors;
}
```

4.7.2.1.2 左外连接 LEFT OUTER JOIN

左外连接会返回左表中的所有数据行，以及右表当中能够匹配连接条件的值，如果在右表当中没有找到能够匹配的行，则使用 NULL 填充。

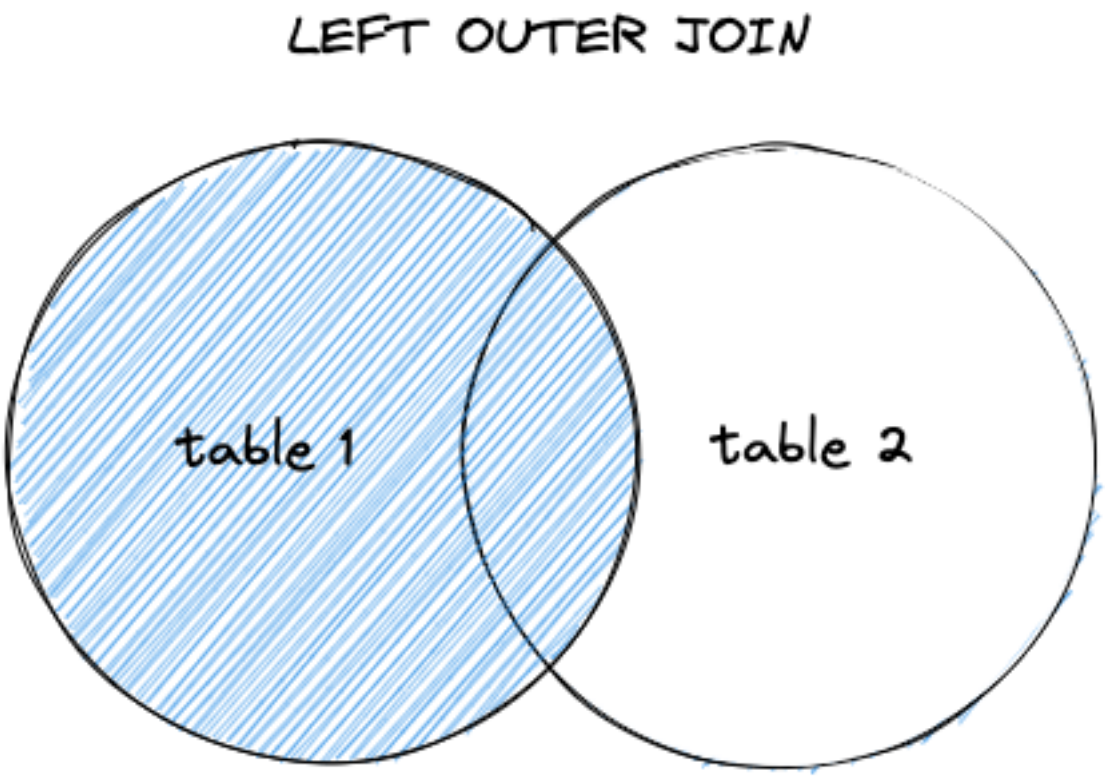


图 13: Left Outer Join

在一些情况下，希望使用多张表来完成数据的查询，但是并不希望因为不满足连接条件而导致数据集变小。

例如，在 Bookshop 应用的首页，希望展示一个带有平均评分的最新书籍列表。在这种情况下，最新的书籍可能是还没有经过任何人评分的，如果使用内连接就会导致这些无人评分的书籍信息被过滤掉，而这并不是期望的结果。

在下面的 SQL 语句当中，通过 `LEFT JOIN` 关键字声明左表 `books` 将以左外连接的方式与右表 `ratings` 进行连接，从而确保 `books` 表当中的所有记录都能得到返回。

```
SELECT b.id AS book_id, ANY_VALUE(b.title) AS book_title, AVG(r.score) AS average_score
FROM books b
LEFT JOIN ratings r ON b.id = r.book_id
GROUP BY b.id
ORDER BY b.published_at DESC
LIMIT 10;
```

查询结果如下：

```
+------------+---------------------------------+---------------+
| book_id    | book_title                      | average_score |
+------------+---------------------------------+---------------+
| 3438991610 | The Documentary of lion         |        2.7619 |
| 3897175886 | Torey Kuhn                      |        3.0000 |
```

```
| 1256171496 | Elmo Vandervort                  |        2.5500 |
| 1036915727 | The Story of Munchkin            |        2.0000 |
|  270254583 | Tate Kovacek                     |        2.5000 |
| 1280950719 | Carson Damore                    |        3.2105 |
| 1098041838 | The Documentary of grasshopper   |        2.8462 |
| 1476566306 | The Adventures of Vince Sanford  |        2.3529 |
| 4036300890 | The Documentary of turtle        |        2.4545 |
| 1299849448 | Antwan Olson                     |        3.0000 |
+------------+----------------------------------+---------------+
10 rows in set (0.30 sec)
```

看起来最新出版的书籍已经有了很多评分，为了验证上面所说的，通过 SQL 语句把 The Documentary of lion 这本书的所有评分给删掉：

```
DELETE FROM ratings WHERE book_id = 3438991610;
```

再次查询，你会发现 The Documentary of lion 这本书依然出现在结果集当中，但是通过右表 `ratings` 的 `score` 列计算得到的 `average_score` 列被填上了 `NULL`。

```
+------------+----------------------------------+---------------+
| book_id    | book_title                       | average_score |
+------------+----------------------------------+---------------+
| 3438991610 | The Documentary of lion          |          NULL |
| 3897175886 | Torey Kuhn                       |        3.0000 |
| 1256171496 | Elmo Vandervort                  |        2.5500 |
| 1036915727 | The Story of Munchkin            |        2.0000 |
|  270254583 | Tate Kovacek                     |        2.5000 |
| 1280950719 | Carson Damore                    |        3.2105 |
| 1098041838 | The Documentary of grasshopper   |        2.8462 |
| 1476566306 | The Adventures of Vince Sanford  |        2.3529 |
| 4036300890 | The Documentary of turtle        |        2.4545 |
| 1299849448 | Antwan Olson                     |        3.0000 |
+------------+----------------------------------+---------------+
10 rows in set (0.30 sec)
```

如果改成使用的是内连接 `JOIN` 结果会怎样？这就交给你来尝试了。

在 Java 中左外连接的示例如下：

```
public List<Book> getLatestBooksWithAverageScore() throws SQLException {
    List<Book> books = new ArrayList<>();
    try (Connection conn = ds.getConnection()) {
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery("""
        SELECT b.id AS book_id, ANY_VALUE(b.title) AS book_title, AVG(r.score) AS average_score
        FROM books b
        LEFT JOIN ratings r ON b.id = r.book_id
```

```
        GROUP BY b.id
        ORDER BY b.published_at DESC
        LIMIT 10;
        """);
        while (rs.next()) {
            Book book = new Book();
            book.setId(rs.getLong("book_id"));
            book.setTitle(rs.getString("book_title"));
            book.setAverageScore(rs.getFloat("average_score"));
            books.add(book);
        }
    }
    return books;
}
```

4.7.2.1.3 右外连接 RIGHT OUTER JOIN

右外连接返回右表中的所有记录，以及左表当中能够匹配连接条件的值，没有匹配的值则使用 NULL 填充。

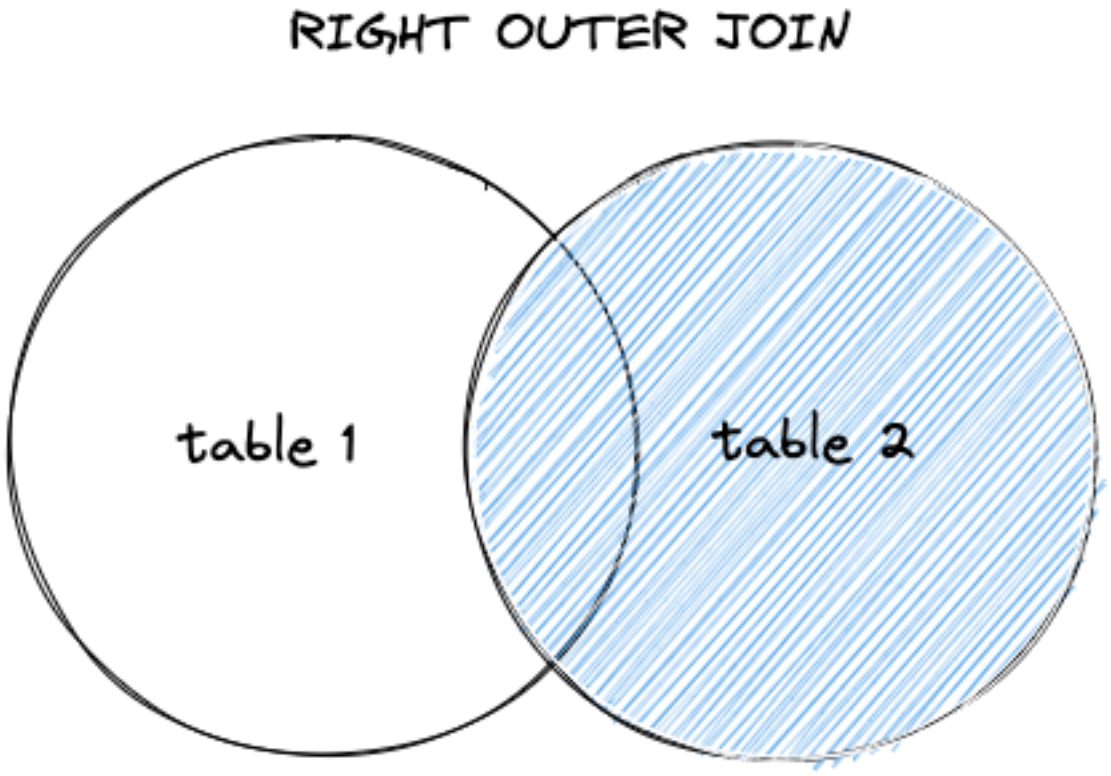


图 14: Right Outer Join

4.7.2.1.4 交叉连接 CROSS JOIN

当连接条件恒成立时，两表之间的内连接称为交叉连接（又被称为“笛卡尔连接”）。交叉连接会把左表的每一条记录和右表的所有记录相连接，如果左表的记录数为 m，右表的记录数为 n，则结果集中会产生 m * n 条记录。

4.7.2.1.5 左半连接 LEFT SEMI JOIN

TiDB 在 SQL 语法层面上不支持 `LEFT SEMI JOIN table_name`，但是在执行计划层面，子查询相关的优化会将 `semi join` 作为改写后的等价 JOIN 查询默认的连接方式。

4.7.2.2 隐式连接

在显式声明连接的 JOIN 语句作为 SQL 标准出现之前，在 SQL 语句当中可以通过 `FROM t1, t2` 子句来连接两张或多张表，通过 `WHERE t1.id = t2.id` 子句来指定连接的条件。你可以将其理解为隐式声明的连接，隐式连接会使用内连接的方式进行连接。

4.7.2.3 Join 相关算法

TiDB 支持下列三种常规的表连接算法，优化器会根据所连接表的数据量等因素来选择合适的 Join 算法去执行。你可以通过 EXPLAIN 语句来查看查询使用了何种算法进行 Join。

- Index Join
- Hash Join
- Merge Join

如果发现 TiDB 的优化器没有按照最佳的 Join 算法去执行。你也可以通过Optimizer Hints 强制 TiDB 使用更好的 Join 算法去执行。

例如，假设上文当中的左连接查询的示例 SQL 使用 Hash Join 算法执行更快，而优化器并没有选择这种算法，你可以在 SELECT 关键字后面加上 Hint `/*+ HASH_JOIN(b, r)*/`（注意：如果表名添加了别名，Hint 当中也应该使用表别名）。

```
EXPLAIN SELECT /*+ HASH_JOIN(b, r) */ b.id AS book_id, ANY_VALUE(b.title) AS book_title, AVG(r.
    ↪ score) AS average_score
FROM books b
LEFT JOIN ratings r ON b.id = r.book_id
GROUP BY b.id
ORDER BY b.published_at DESC
LIMIT 10;
```

Join 算法相关的 Hints：

- MERGE_JOIN(t1_name [, tl_name …])
- INL_JOIN(t1_name [, tl_name …])
- INL_HASH_JOIN(t1_name [, tl_name …])
- HASH_JOIN(t1_name [, tl_name …])

4.7.2.4 Join 顺序

在实际的业务场景中，多个表的 Join 语句是很常见的，而 Join 的执行效率和各个表参与 Join 的顺序有关。TiDB 使用 Join Reorder 算法来确定多个表进行 Join 的顺序。

当优化器选择的 Join 顺序并不够好时，你可以使用 STRAIGHT_JOIN 语法让 TiDB 强制按照 FROM 子句中所使用的表的顺序做联合查询。

```
EXPLAIN SELECT *
FROM authors a STRAIGHT_JOIN book_authors ba STRAIGHT_JOIN books b
WHERE b.id = ba.book_id AND ba.author_id = a.id;
```

关于该算法的实现细节和限制你可以通过查看Join Reorder 算法简介章节进行了解。

4.7.2.5 扩展阅读

- 用 EXPLAIN 查看 JOIN 查询的执行计划
- Join Reorder 算法简介

4.7.3 子查询

本章将介绍 TiDB 中的子查询功能。

4.7.3.1 概述

子查询是嵌套在另一个查询中的 SQL 表达式，借助子查询，可以在一个查询当中使用另外一个查询的查询结果。

下面将以Bookshop 应用为例对子查询展开介绍：

4.7.3.2 子查询语句

通常情况下，子查询语句分为如下几种形式：

- 标量子查询（Scalar Subquery），如 `SELECT (SELECT s1 FROM t2)FROM t1`。
- 派生表（Derived Tables），如 `SELECT t1.s1 FROM (SELECT s1 FROM t2)t1`。
- 存在性测试（Existential Test），如 `WHERE NOT EXISTS(SELECT ... FROM t2)`，`WHERE t1.a IN (SELECT ... FROM t2)`。
- 集合比较（Quantified Comparison），如 `WHERE t1.a = ANY(SELECT ... FROM t2)`。
- 作为比较运算符操作数的子查询，如 `WHERE t1.a > (SELECT ... FROM t2)`。

4.7.3.3 子查询的分类

一般来说，可以将子查询分为关联子查询（Correlated Subquery）和无关联子查询 (Self-contained Subquery) 两大类，TiDB 对于这两类子查询的处理方式是不一样的。

判断是否为关联子查询的依据在于子查询当中是否引用了外层查询的列。

4.7.3.3.1 无关联子查询

对于将子查询作为比较运算符 (> / >=/ < / <= / = / !=) 操作数的这类无关联子查询而言，内层子查询只需要进行一次查询，TiDB 在生成执行计划阶段会将内层子查询改写为常量。

例如，想要查找 authors 表当中年龄大于总体平均年龄的作家，可以通过将子查询作为比较操作符的操作数来实现：

```
SELECT * FROM authors a1 WHERE (IFNULL(a1.death_year, YEAR(NOW())) - a1.birth_year) > (
    SELECT
        AVG(IFNULL(a2.death_year, YEAR(NOW())) - a2.birth_year) AS average_age
    FROM
        authors a2
)
```

在 TiDB 执行上述查询的时候会先执行一次内层子查询：

```
SELECT AVG(IFNULL(a2.death_year, YEAR(NOW())) - a2.birth_year) AS average_age FROM authors a2;
```

假设查询得到的结果为 34，即总体平均年龄为 34，34 将作为常量替换掉原来的子查询。

```
SELECT * FROM authors a1
WHERE (IFNULL(a1.death_year, YEAR(NOW())) - a1.birth_year) > 34;
```

运行结果为：

```
+--------+-------------------+--------+------------+------------+
| id     | name              | gender | birth_year | death_year |
+--------+-------------------+--------+------------+------------+
| 13514  | Kennith Kautzer   | 1      | 1956       | 2018       |
| 13748  | Dillon Langosh    | 1      | 1985       | NULL       |
| 99184  | Giovanny Emmerich | 1      | 1954       | 2012       |
| 180191 | Myrtie Robel      | 1      | 1958       | 2009       |
| 200969 | Iva Renner        | 0      | 1977       | NULL       |
| 209671 | Abraham Ortiz     | 0      | 1943       | 2016       |
| 229908 | Wellington Wiza   | 1      | 1932       | 1969       |
| 306642 | Markus Crona      | 0      | 1969       | NULL       |
| 317018 | Ellis McCullough  | 0      | 1969       | 2014       |
| 322369 | Mozelle Hand      | 0      | 1942       | 1977       |
| 325946 | Elta Flatley      | 0      | 1933       | 1986       |
| 361692 | Otho Langosh      | 1      | 1931       | 1997       |
| 421294 | Karelle VonRueden | 0      | 1977       | NULL       |
...
```

对于存在性测试和集合比较两种情况下的无关联列子查询，TiDB 会将其进行改写和等价替换以获得更好的执行性能，你可以通过阅读子查询相关的优化章节来了解更多的实现细节。

4.7.3.4 关联子查询

对于关联子查询而言，由于内层的子查询引用外层查询的列，子查询需要对外层查询得到的每一行都执行一遍，也就是说假设外层查询得到一千万的结果，那么子查询也会被执行一千万次，这会导致查询需要消耗更多的时间和资源。

因此在处理过程中，TiDB 会尝试对关联子查询去关联，以从执行计划层面上提高查询效率。

例如，假设想要查找那些大于其它相同性别作家的平均年龄的的作家，SQL 语句可以这样写：

```
SELECT * FROM authors a1 WHERE (IFNULL(a1.death_year, YEAR(NOW())) - a1.birth_year) > (
    SELECT
        AVG(
            IFNULL(a2.death_year, YEAR(NOW())) - IFNULL(a2.birth_year, YEAR(NOW()))
        ) AS average_age
    FROM
        authors a2
    WHERE a1.gender = a2.gender
);
```

TiDB 在处理该 SQL 语句是会将其改写为等价的 Join 查询：

```
SELECT *
FROM
    authors a1,
    (
        SELECT
            gender, AVG(
                IFNULL(a2.death_year, YEAR(NOW())) - IFNULL(a2.birth_year, YEAR(NOW()))
            ) AS average_age
        FROM
            authors a2
        GROUP BY gender
    ) a2
WHERE
    a1.gender = a2.gender
    AND (IFNULL(a1.death_year, YEAR(NOW())) - a1.birth_year) > a2.average_age;
```

作为最佳实践，在实际开发当中，建议在明确知道有更好的等价写法时，尽量避免通过关联子查询来进行查询。

4.7.3.5 扩展阅读

- 子查询相关的优化
- 关联子查询去关联
- TiDB 中的子查询优化技术

4.7.4 分页查询

当查询结果数据量较大时，往往希望以“分页”的方式返回所需要的部分。

4.7.4.1 对查询结果进行分页

在 TiDB 当中，可以利用 LIMIT 语句来实现分页功能，常规的分页语句写法如下所示：

```
SELECT * FROM table_a t ORDER BY gmt_modified DESC LIMIT offset, row_count;
```

offset 表示起始记录数，row_count 表示每页记录数。除此之外，TiDB 也支持 LIMIT row_count OFFSET offset 语法。

除非明确要求不要使用任何排序来随机展示数据，使用分页查询语句时都应该通过 ORDER BY 语句指定查询结果的排序方式。

例如，在Bookshop 应用当中，希望将最新书籍列表以分页的形式返回给用户。通过 LIMIT 0, 10 语句，便可以得到列表第 1 页的书籍信息，每页中最多有 10 条记录。获取第 2 页信息，则改成可以改成 LIMIT 10, 10，如此类推。

```
SELECT *
FROM books
ORDER BY published_at DESC
LIMIT 0, 10;
```

在使用 Java 开发应用程序时，后端程序从前端接收到的参数页码 page_number 和每页的数据条数 page_size，而不是起始记录数 offset，因此在进行数据库查询前需要对其进行一些转换。

```
public List<Book> getLatestBooksPage(Long pageNumber, Long pageSize) throws SQLException {
    pageNumber = pageNumber < 1L ? 1L : pageNumber;
    pageSize = pageSize < 10L ? 10L : pageSize;
    Long offset = (pageNumber - 1) * pageSize;
    Long limit = pageSize;
    List<Book> books = new ArrayList<>();
    try (Connection conn = ds.getConnection()) {
        PreparedStatement stmt = conn.prepareStatement("""
        SELECT id, title, published_at
        FROM books
        ORDER BY published_at DESC
        LIMIT ?, ?;
        """);
        stmt.setLong(1, offset);
        stmt.setLong(2, limit);
        ResultSet rs = stmt.executeQuery();
        while (rs.next()) {
            Book book = new Book();
            book.setId(rs.getLong("id"));
            book.setTitle(rs.getString("title"));
```

```
            book.setPublishedAt(rs.getDate("published_at"));
            books.add(book);
        }
    }
    return books;
}
```

4.7.4.2 单字段主键表的分页批处理

常规的分页更新 SQL 一般使用主键或者唯一索引进行排序，再配合 LIMIT 语法中的 `offset`，按固定行数拆分页面。然后把页面包装进独立的事务中，从而实现灵活的分页更新。但是，劣势也很明显：由于需要对主键或者唯一索引进行排序，越靠后的页面参与排序的行数就会越多，尤其当批量处理涉及的数据体量较大时，可能会占用过多计算资源。

下面将介绍一种更为高效的分页批处理方案：

使用 SQL 实现分页批处理，可以按照如下步骤进行：

首先将数据按照主键排序，然后调用窗口函数 `row_number()` 为每一行数据生成行号，接着调用聚合函数按照设置好的页面大小对行号进行分组，最终计算出每页的最小值和最大值。

```
SELECT
    floor((t.row_num - 1) / 1000) + 1 AS page_num,
    min(t.id) AS start_key,
    max(t.id) AS end_key,
    count(*) AS page_size
FROM (
    SELECT id, row_number() OVER (ORDER BY id) AS row_num
    FROM books
) t
GROUP BY page_num
ORDER BY page_num;
```

查询结果如下：

```
+----------+------------+------------+-----------+
| page_num | start_key  | end_key    | page_size |
+----------+------------+------------+-----------+
|        1 |     268996 |  213168525 |      1000 |
|        2 |  213210359 |  430012226 |      1000 |
|        3 |  430137681 |  647846033 |      1000 |
|        4 |  647998334 |  848878952 |      1000 |
|        5 |  848899254 | 1040978080 |      1000 |
...
|       20 | 4077418867 | 4294004213 |      1000 |
+----------+------------+------------+-----------+
20 rows in set (0.01 sec)
```

接下来，只需要使用 `WHERE id BETWEEN start_key AND end_key` 语句查询每个分片的数据即可。修改数据时，也可以借助上面计算好的分片信息，实现高效的数据更新。

例如，假如想要删除第 1 页上的所有书籍的基本信息，可以将上表第 1 页所对应的 `start_key` 和 `end_key` 填入 SQL 语句当中。

```
DELETE FROM books
WHERE
    id BETWEEN 268996 AND 213168525
ORDER BY id;
```

在 Java 语言当中，可以定义一个 `PageMeta` 类来存储分页元信息。

```
public class PageMeta<K> {
    private Long pageNum;
    private K startKey;
    private K endKey;
    private Long pageSize;

    // Skip the getters and setters.

}
```

定义一个 `getPageMetaList()` 方法获取到分页元信息列表，然后定义一个可以根据页面元信息批量删除数据的方法 `deleteBooksByPageMeta()`。

```
public class BookDAO {
    public List<PageMeta<Long>> getPageMetaList() throws SQLException {
        List<PageMeta<Long>> pageMetaList = new ArrayList<>();
        try (Connection conn = ds.getConnection()) {
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery("""
            SELECT
                floor((t.row_num - 1) / 1000) + 1 AS page_num,
                min(t.id) AS start_key,
                max(t.id) AS end_key,
                count(*) AS page_size
            FROM (
                SELECT id, row_number() OVER (ORDER BY id) AS row_num
                FROM books
            ) t
            GROUP BY page_num
            ORDER BY page_num;
            """);
            while (rs.next()) {
                PageMeta<Long> pageMeta = new PageMeta<>();
                pageMeta.setPageNum(rs.getLong("page_num"));
```

```
                pageMeta.setStartKey(rs.getLong("start_key"));
                pageMeta.setEndKey(rs.getLong("end_key"));
                pageMeta.setPageSize(rs.getLong("page_size"));
                pageMetaList.add(pageMeta);
            }
        }
        return pageMetaList;
    }

    public void deleteBooksByPageMeta(PageMeta<Long> pageMeta) throws SQLException {
        try (Connection conn = ds.getConnection()) {
            PreparedStatement stmt = conn.prepareStatement("DELETE FROM books WHERE id >= ? AND
                ↪ id <= ?");
            stmt.setLong(1, pageMeta.getStartKey());
            stmt.setLong(2, pageMeta.getEndKey());
            stmt.executeUpdate();
        }
    }
}
```

如果想要删除第 1 页的数据，可以这样写：

```
List<PageMeta<Long>> pageMetaList = bookDAO.getPageMetaList();
if (pageMetaList.size() > 0) {
    bookDAO.deleteBooksByPageMeta(pageMetaList.get(0));
}
```

如果希望通过分页分批地删除所有书籍数据，可以这样写：

```
List<PageMeta<Long>> pageMetaList = bookDAO.getPageMetaList();
pageMetaList.forEach((pageMeta) -> {
    try {
        bookDAO.deleteBooksByPageMeta(pageMeta);
    } catch (SQLException e) {
        e.printStackTrace();
    }
});
```

改进方案由于规避了频繁的数据排序操作造成的性能损耗，显著改善了批量处理的效率。

4.7.4.3 复合主键表的分页批处理

4.7.4.3.1 非聚簇索引表

对于非聚簇索引表（又被称为“非索引组织表”）而言，可以使用隐藏字段 `_tidb_rowid` 作为分页的 key，分页的方法与单列主键表中所介绍的方法相同。

> **建议：**
>
> 你可以通过 `SHOW CREATE TABLE users;` 语句查看表主键是否使用了聚簇索引。

例如：

```
SELECT
    floor((t.row_num - 1) / 1000) + 1 AS page_num,
    min(t._tidb_rowid) AS start_key,
    max(t._tidb_rowid) AS end_key,
    count(*) AS page_size
FROM (
    SELECT _tidb_rowid, row_number() OVER (ORDER BY _tidb_rowid) AS row_num
    FROM users
) t
GROUP BY page_num
ORDER BY page_num;
```

查询结果如下：

```
+----------+-----------+---------+-----------+
| page_num | start_key | end_key | page_size |
+----------+-----------+---------+-----------+
|        1 |         1 |    1000 |      1000 |
|        2 |      1001 |    2000 |      1000 |
|        3 |      2001 |    3000 |      1000 |
|        4 |      3001 |    4000 |      1000 |
|        5 |      4001 |    5000 |      1000 |
|        6 |      5001 |    6000 |      1000 |
|        7 |      6001 |    7000 |      1000 |
|        8 |      7001 |    8000 |      1000 |
|        9 |      8001 |    9000 |      1000 |
|       10 |      9001 |    9990 |       990 |
+----------+-----------+---------+-----------+
10 rows in set (0.00 sec)
```

4.7.4.3.2 聚簇索引表

对于聚簇索引表（又被称为“索引组织表”），可以利用 `concat` 函数将多个列的值连接起来作为一个 `key`，然后使用窗口函数获取分页信息。

需要注意的是，这时候 `key` 是一个字符串，你必须确保这个字符串长度总是相等的，才能够通过 `min` 和 `max` 聚合函数得到分页内正确的 `start_key` 和 `end_key`。如果进行字符串连接的字段长度不固定，你可以通过 `LPAD` 函数进行补全。

例如，想要对 ratings 表里的数据进行分页批处理。

先可以通过下面的 SQL 语句来在制造元信息表。因为组成 key 的 book_id 列和 user_id 列都是 bigint 类型，转换为字符串是并不是等宽的，因此需要根据 bigint 类型的最大位数 19，使用 LPAD 函数在长度不够时用 0 补齐。

```
SELECT
    floor((t1.row_num - 1) / 10000) + 1 AS page_num,
    min(mvalue) AS start_key,
    max(mvalue) AS end_key,
    count(*) AS page_size
FROM (
    SELECT
        concat('(', LPAD(book_id, 19, 0), ',', LPAD(user_id, 19, 0), ')') AS mvalue,
        row_number() OVER (ORDER BY book_id, user_id) AS row_num
    FROM ratings
) t1
GROUP BY page_num
ORDER BY page_num;
```

> **注意：**
>
> 该 SQL 会以全表扫描 (TableFullScan) 方式执行，当数据量较大时，查询速度会变慢，此时可以使用 TiFlash 进行加速。

查询结果如下：

```
+----------+-----------------------------------------+-----------------------------------------+-----------+
    ↪
| page_num | start_key                               | end_key
    ↪                                | page_size |
+----------+-----------------------------------------+-----------------------------------------+-----------+
    ↪
|        1 | (0000000000000268996,0000000000092104804) |
    ↪ (0000000000140982742,0000000000374645100) |     10000 |
|        2 | (0000000000140982742,0000000000456757551) |
    ↪ (0000000000287195082,0000000004053200550) |     10000 |
|        3 | (0000000000287196791,0000000000191962769) |
    ↪ (0000000000434010216,0000000000237646714) |     10000 |
|        4 | (0000000000434010216,0000000000375066168) |
    ↪ (0000000000578893327,0000000002167504460) |     10000 |
|        5 | (0000000000578893327,0000000002457322286) |
    ↪ (0000000000718287668,0000000001502744628) |     10000 |
...
```

```
|        29 | (0000000004002523918,0000000000902930986) |
    ↪ (0000000004147203315,0000000004090920746) |      10000 |
|        30 | (0000000004147421329,0000000000319181561) |
    ↪ (0000000004294004213,0000000003586311166) |       9972 |
+-----------+-------------------------------------------+-------------------------------------------+-----------+
    ↪
30 rows in set (0.28 sec)
```

假如想要删除第 1 页上的所有评分记录，可以将上表第 1 页所对应的 start_key 和 end_key 填入 SQL 语句当中。

```
SELECT * FROM ratings
WHERE
    (book_id > 268996 AND book_id < 140982742)
    OR (
        book_id = 268996 AND user_id >= 92104804
    )
    OR (
        book_id = 140982742 AND user_id <= 374645100
    )
ORDER BY book_id, user_id;
```

4.7.5 视图

本章将介绍 TiDB 中的视图功能。

4.7.5.1 概述

TiDB 支持视图，视图是一张虚拟表，该虚拟表的结构由创建视图时的 SELECT 语句定义。

- 通过视图可以对用户只暴露安全的字段及数据，进而保证底层表的敏感字段及数据的安全。
- 将频繁出现的复杂查询定义为视图，可以使复杂查询更加简单便捷。

4.7.5.2 创建视图

在 TiDB 当中，可以通过 CREATE VIEW 语句来将某个较为复杂的查询定义为视图，其语法如下：

```
CREATE VIEW view_name AS query;
```

请注意，创建的视图名称不能与已有的视图或表重名。

例如，在多表连接查询 章节当中，通过 JOIN 语句连接 books 表和 ratings 表查询到了带有平均评分的书籍列表。为了方便后续查询，可以将该查询语句定义为一个视图，SQL 语句如下所示：

```
CREATE VIEW book_with_ratings AS
SELECT b.id AS book_id, ANY_VALUE(b.title) AS book_title, AVG(r.score) AS average_score
FROM books b
LEFT JOIN ratings r ON b.id = r.book_id
GROUP BY b.id;
```

4.7.5.3 查询视图

视图创建完成后，便可以使用 SELECT 语句像查询一般数据表一样查询视图。

```
SELECT * FROM book_with_ratings LIMIT 10;
```

TiDB 在执行查询视图语句时，会将视图展开成创建视图时定义的 SELECT 语句，进而执行展开后的查询语句。

4.7.5.4 更新视图

目前 TiDB 中的视图不支持 `ALTER VIEW view_name AS query;` 语法，你可以通过以下两种方式实现视图的“更新”：

- 先 `DROP VIEW view_name;` 语句删除旧视图，再通过 `CREATE VIEW view_name AS query;` 语句创建新视图的方式来更新视图。
- 使用 `CREATE OR REPLACE VIEW view_name AS query;` 语句覆盖已存在的同名视图。

```
CREATE OR REPLACE VIEW book_with_ratings AS
SELECT b.id AS book_id, ANY_VALUE(b.title), ANY_VALUE(b.published_at) AS book_title, AVG(r.score)
    ↪  AS average_score
FROM books b
LEFT JOIN ratings r ON b.id = r.book_id
GROUP BY b.id;
```

4.7.5.5 获取视图相关信息

4.7.5.5.1 使用 SHOW CREATE TABLE|VIEW view_name 语句

```
SHOW CREATE VIEW book_with_ratings\G
```

运行结果为：

```
*************************** 1. row ***************************
                View: book_with_ratings
         Create View: CREATE ALGORITHM=UNDEFINED DEFINER=`root`@`%` SQL SECURITY DEFINER VIEW `
            ↪ book_with_ratings` (`book_id`, `ANY_VALUE(b.title)`, `book_title`, `
            ↪ average_score`) AS SELECT `b`.`id` AS `book_id`,ANY_VALUE(`b`.`title`) AS `
            ↪ ANY_VALUE(b.title)`,ANY_VALUE(`b`.`published_at`) AS `book_title`,AVG(`r`.`score
            ↪ `) AS `average_score` FROM `bookshop`.`books` AS `b` LEFT JOIN `bookshop`.`
            ↪ ratings` AS `r` ON `b`.`id`=`r`.`book_id` GROUP BY `b`.`id`
character_set_client: utf8mb4
collation_connection: utf8mb4_general_ci
1 row in set (0.00 sec)
```

4.7.5.5.2 查询 INFORMATION_SCHEMA.VIEWS 表

```
SELECT * FROM information_schema.views WHERE TABLE_NAME = 'book_with_ratings'\G
```

运行结果为：

```
*************************** 1. row ***************************
       TABLE_CATALOG: def
        TABLE_SCHEMA: bookshop
          TABLE_NAME: book_with_ratings
     VIEW_DEFINITION: SELECT `b`.`id` AS `book_id`,ANY_VALUE(`b`.`title`) AS `ANY_VALUE(b.title)
        ↪ `,ANY_VALUE(`b`.`published_at`) AS `book_title`,AVG(`r`.`score`) AS `average_score`
        ↪ FROM `bookshop`.`books` AS `b` LEFT JOIN `bookshop`.`ratings` AS `r` ON `b`.`id`=`r
        ↪ `.`book_id` GROUP BY `b`.`id`
        CHECK_OPTION: CASCADED
        IS_UPDATABLE: NO
             DEFINER: root@%
       SECURITY_TYPE: DEFINER
CHARACTER_SET_CLIENT: utf8mb4
COLLATION_CONNECTION: utf8mb4_general_ci
1 row in set (0.00 sec)
```

4.7.5.6 删除视图

通过 DROP VIEW view_name; 语句可以删除已经创建的视图。

```
DROP VIEW book_with_ratings;
```

4.7.5.7 局限性

关于局限性，你可以通过阅读参考文档当中的视图章节进行了解。

4.7.5.8 扩展阅读

- 视图
- CREATE VIEW 语句
- DROP VIEW 语句
- 用 EXPLAIN 查看带视图的 SQL 执行计划
- TiFlink: 使用 TiKV 和 Flink 实现强一致的物化视图

4.7.6 临时表

临时表可以被认为是一种复用查询结果的技术。

假设希望知道Bookshop 应用当中最年长的作家们的一些情况，可能需要编写多个查询，而这些查询都需要使用到这个最年长作家列表。可以通过下面的 SQL 语句从 authors 表当中找出最年长的前 50 位作家作为研究对象。

```
SELECT a.id, a.name, (IFNULL(a.death_year, YEAR(NOW())) - a.birth_year) AS age
FROM authors a
ORDER BY age DESC
LIMIT 50;
```

查询结果如下：

```
+------------+---------------------+------+
| id         | name                | age  |
+------------+---------------------+------+
| 4053452056 | Dessie Thompson     |   80 |
| 2773958689 | Pedro Hansen        |   80 |
| 4005636688 | Wyatt Keeling       |   80 |
| 3621155838 | Colby Parker        |   80 |
| 2738876051 | Friedrich Hagenes   |   80 |
| 2299112019 | Ray Macejkovic      |   80 |
| 3953661843 | Brandi Williamson   |   80 |
...
| 4100546410 | Maida Walsh         |   80 |
+------------+---------------------+------+
50 rows in set (0.01 sec)
```

在找到这 50 位最年长的作家后，希望缓存这个查询结果，以便后续的查询能够方便地使用到这组数据。如果使用一般的数据库表进行存储的话，在创建这些表时，需要考虑如何避免不同会话之间的表重名问题，而且可能在一批查询结束之后就不再需要这些表了，还需要及时地对这些中间结果表进行清理。

4.7.6.1 创建临时表

为了满足这类缓存中间结果的需求，TiDB 在 v5.3.0 版本中引入了临时表功能，对于临时表当中的本地临时表而言，TiDB 将会在会话结束的一段时间后自动清理这些已经没用的临时表，用户无需担心中间结果表的增多会带来管理上的麻烦。

4.7.6.1.1 临时表类型

TiDB 的临时表分为本地临时表和全局临时表：

- 本地临时表的表定义和表内数据只对当前会话可见，适用于暂存会话内的中间数据。
- 全局临时表的表定义对整个 TiDB 集群可见，表内数据只对当前事务可见，适用于暂存事务内的中间数据。

4.7.6.1.2 创建本地临时表

在创建本地临时表前，你需要给当前数据库用户添加上 `CREATE TEMPORARY TABLES` 权限。

在 SQL 中，通过 `CREATE TEMPORARY TABLE <table_name>` 语句创建临时表，默认临时表的类型为本地临时表，它只能被当前会话所访问。

```
CREATE TEMPORARY TABLE top_50_eldest_authors (
    id BIGINT,
    name VARCHAR(255),
    age INT,
    PRIMARY KEY(id)
);
```

在创建完临时表后，你可以通过 `INSERT INTO table_name SELECT ...` 语句，将上述查询得到的结果导入到刚刚创建的临时表当中。

```
INSERT INTO top_50_eldest_authors
SELECT a.id, a.name, (IFNULL(a.death_year, YEAR(NOW())) - a.birth_year) AS age
FROM authors a
ORDER BY age DESC
LIMIT 50;
```

运行结果为：

```
Query OK, 50 rows affected (0.03 sec)
Records: 50  Duplicates: 0  Warnings: 0
```

在 Java 中创建本地临时表的示例如下：

```
public List<Author> getTop50EldestAuthorInfo() throws SQLException {
    List<Author> authors = new ArrayList<>();
    try (Connection conn = ds.getConnection()) {
        Statement stmt = conn.createStatement();
        stmt.executeUpdate("""
            CREATE TEMPORARY TABLE top_50_eldest_authors (
                id BIGINT,
                name VARCHAR(255),
                age INT,
                PRIMARY KEY(id)
            );
        """);

        stmt.executeUpdate("""
            INSERT INTO top_50_eldest_authors
            SELECT a.id, a.name, (IFNULL(a.death_year, YEAR(NOW())) - a.birth_year) AS age
            FROM authors a
            ORDER BY age DESC
            LIMIT 50;
        """);

        ResultSet rs = stmt.executeQuery("""
            SELECT id, name FROM top_50_eldest_authors;
```

```
        """);

        while (rs.next()) {
            Author author = new Author();
            author.setId(rs.getLong("id"));
            author.setName(rs.getString("name"));
            authors.add(author);
        }
    }
    return authors;
}
```

4.7.6.1.3 创建全局临时表

在 SQL 中，你可以通过加上 GLOBAL 关键字来声明你所创建的是全局临时表。创建全局临时表时必须在末尾 ON COMMIT DELETE ROWS 修饰，这表明该全局数据表的所有数据行将在事务结束后被删除。

```
CREATE GLOBAL TEMPORARY TABLE IF NOT EXISTS top_50_eldest_authors_global (
    id BIGINT,
    name VARCHAR(255),
    age INT,
    PRIMARY KEY(id)
) ON COMMIT DELETE ROWS;
```

在对全局临时表导入数据时，你需要特别注意，你必须通过 BEGIN 显式声明事务的开始。否则导入的数据在 INSERT INTO 语句执行后就清除掉，因为 Auto Commit 模式下，INSERT INTO 语句的执行结束，事务会自动被提交，事务结束，全局临时表的数据便被清空了。

在 Java 中使用全局临时表时，你需要将 Auto Commit 模式先关闭。在 Java 语言当中，你可以通过 conn.setAutoCommit(false); 语句来实现，当你使用完成后，可以通过 conn.commit(); 显式地提交事务。事务在提交或取消后，在事务过程中对全局临时表添加的数据将会被清除。

```
public List<Author> getTop50EldestAuthorInfo() throws SQLException {
    List<Author> authors = new ArrayList<>();
    try (Connection conn = ds.getConnection()) {
        conn.setAutoCommit(false);

        Statement stmt = conn.createStatement();
        stmt.executeUpdate("""
            CREATE GLOBAL TEMPORARY TABLE IF NOT EXISTS top_50_eldest_authors (
                id BIGINT,
                name VARCHAR(255),
                age INT,
                PRIMARY KEY(id)
            ) ON COMMIT DELETE ROWS;
        """);
```

```
        stmt.executeUpdate("""
            INSERT INTO top_50_eldest_authors
            SELECT a.id, a.name, (IFNULL(a.death_year, YEAR(NOW())) - a.birth_year) AS age
            FROM authors a
            ORDER BY age DESC
            LIMIT 50;
        """);

        ResultSet rs = stmt.executeQuery("""
            SELECT id, name FROM top_50_eldest_authors;
        """);

        conn.commit();
        while (rs.next()) {
            Author author = new Author();
            author.setId(rs.getLong("id"));
            author.setName(rs.getString("name"));
            authors.add(author);
        }
    }
    return authors;
}
```

4.7.6.2 查看临时表信息

通过 `SHOW [FULL] TABLES` 语句可以查看到已经创建的全局临时表，但是无法看到本地临时表的信息，TiDB 暂时也没有类似的 `information_schema.INNODB_TEMP_TABLE_INFO` 系统表存放临时表的信息。

例如，你可以在 table 列表当中查看到全局临时表 `top_50_eldest_authors_global`，但是无法查看到 `top_50_eldest_authors` 表。

```
+-------------------------------+------------+
| Tables_in_bookshop            | Table_type |
+-------------------------------+------------+
| authors                       | BASE TABLE |
| book_authors                  | BASE TABLE |
| books                         | BASE TABLE |
| orders                        | BASE TABLE |
| ratings                       | BASE TABLE |
| top_50_eldest_authors_global  | BASE TABLE |
| users                         | BASE TABLE |
+-------------------------------+------------+
9 rows in set (0.00 sec)
```

4.7.6.3 查询临时表

在临时表准备就绪之后，你便可以像对一般数据表一样对临时表进行查询：

```
SELECT * FROM top_50_eldest_authors;
```

你可以通过表连接将临时表中的数据引用到你的查询当中：

```
EXPLAIN SELECT ANY_VALUE(ta.id) AS author_id, ANY_VALUE(ta.age), ANY_VALUE(ta.name), COUNT(*) AS
    ↪ books
FROM top_50_eldest_authors ta
LEFT JOIN book_authors ba ON ta.id = ba.author_id
GROUP BY ta.id;
```

与视图有所不同，在对临时表进行查询时，不会再执行导入数据时所使用的原始查询，而是直接从临时表中获取数据。在一些情况下，这会帮助你提高查询的效率。

4.7.6.4 删除临时表

本地临时表会在会话结束后连同数据和表结构都进行自动清理。全局临时表在事务结束后会自动清除数据，但是表结构依然保留，需要手动删除。

你可以通过 DROP TABLE 或 DROP TEMPORARY TABLE 语句手动删除本地临时表。例如：

```
DROP TEMPORARY TABLE top_50_eldest_authors;
```

你还可以通过 DROP TABLE 或 DROP GLOBAL TEMPORARY TABLE 语句手动删除全局临时表。例如：

```
DROP GLOBAL TEMPORARY TABLE top_50_eldest_authors_global;
```

4.7.6.5 限制

关于 TiDB 在临时表功能上的一些限制，你可以通过阅读参考文档中的临时表与其他 TiDB 功能的兼容性限制小节进行了解。

4.7.6.6 扩展阅读

- 临时表

4.7.7 公共表表达式 (CTE)

由于业务的客观复杂性，有时候会写出长达 2000 行的单条 SQL 语句，其中包含大量的聚合和多层子查询嵌套，维护此类 SQL 堪称开发人员的噩梦。

在前面的小节当中已经介绍了如何使用视图简化查询，也介绍了如何使用临时表来缓存中间查询结果。

在这一小节当中，将介绍 TiDB 当中的公共表表达式（CTE）语法，它是一种更加便捷的复用查询结果的方法。

TiDB 从 5.1 版本开始支持 ANSI SQL 99 标准的 CTE 及其递归的写法，极大提升开发人员和 DBA 编写复杂业务逻辑 SQL 的效率，增强代码的可维护性。

4.7.7.1 基本使用

公共表表达式 (CTE) 是一个临时的中间结果集，能够在 SQL 语句中引用多次，提高 SQL 语句的可读性与执行效率。在 TiDB 中可以通过 WITH 语句使用公共表表达式。

公共表表达式可以分为非递归和递归两种类型。

4.7.7.1.1 非递归的 CTE

非递归的 CTE 使用如下语法进行定义：

```
WITH <query_name> AS (
    <query_definition>
)
SELECT ... FROM <query_name>;
```

例如，假设还想知道最年长的 50 位作家分别编写过多少书籍。

在 SQL 中，可以将临时表小节当中的例子改为以下 SQL 语句：

```
WITH top_50_eldest_authors_cte AS (
    SELECT a.id, a.name, (IFNULL(a.death_year, YEAR(NOW())) - a.birth_year) AS age
    FROM authors a
    ORDER BY age DESC
    LIMIT 50
)
SELECT
    ANY_VALUE(ta.id) AS author_id,
    ANY_VALUE(ta.age) AS author_age,
    ANY_VALUE(ta.name) AS author_name,
    COUNT(*) AS books
FROM top_50_eldest_authors_cte ta
LEFT JOIN book_authors ba ON ta.id = ba.author_id
GROUP BY ta.id;
```

查询结果如下：

```
+------------+------------+--------------------+-------+
| author_id  | author_age | author_name        | books |
+------------+------------+--------------------+-------+
| 1238393239 |         80 | Araceli Purdy      |     1 |
|  817764631 |         80 | Ivory Davis        |     3 |
| 3093759193 |         80 | Lysanne Harris     |     1 |
| 2299112019 |         80 | Ray Macejkovic     |     4 |
...
+------------+------------+--------------------+-------+
50 rows in set (0.01 sec)
```

在 Java 中的示例如下：

```
public List<Author> getTop50EldestAuthorInfoByCTE() throws SQLException {
    List<Author> authors = new ArrayList<>();
    try (Connection conn = ds.getConnection()) {
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery("""
            WITH top_50_eldest_authors_cte AS (
                SELECT a.id, a.name, (IFNULL(a.death_year, YEAR(NOW())) - a.birth_year) AS age
                FROM authors a
                ORDER BY age DESC
                LIMIT 50
            )
            SELECT
                ANY_VALUE(ta.id) AS author_id,
                ANY_VALUE(ta.name) AS author_name,
                ANY_VALUE(ta.age) AS author_age,
                COUNT(*) AS books
            FROM top_50_eldest_authors_cte ta
            LEFT JOIN book_authors ba ON ta.id = ba.author_id
            GROUP BY ta.id;
        """);
        while (rs.next()) {
            Author author = new Author();
            author.setId(rs.getLong("author_id"));
            author.setName(rs.getString("author_name"));
            author.setAge(rs.getShort("author_age"));
            author.setBooks(rs.getInt("books"));
            authors.add(author);
        }
    }
    return authors;
}
```

这时，可以发现名为 “Ray Macejkovic” 的作者写了 4 本书，继续通过 CTE 查询来了解这 4 本书的销量和评分：

```
WITH books_authored_by_rm AS (
    SELECT *
    FROM books b
    LEFT JOIN book_authors ba ON b.id = ba.book_id
    WHERE author_id = 2299112019
), books_with_average_ratings AS (
    SELECT
        b.id AS book_id,
        AVG(r.score) AS average_rating
    FROM books_authored_by_rm b
    LEFT JOIN ratings r ON b.id = r.book_id
```

```
    GROUP BY b.id
), books_with_orders AS (
    SELECT
        b.id AS book_id,
        COUNT(*) AS orders
    FROM books_authored_by_rm b
    LEFT JOIN orders o ON b.id = o.book_id
    GROUP BY b.id
)
SELECT
    b.id AS `book_id`,
    b.title AS `book_title`,
    br.average_rating AS `average_rating`,
    bo.orders AS `orders`
FROM
    books_authored_by_rm b
    LEFT JOIN books_with_average_ratings br ON b.id = br.book_id
    LEFT JOIN books_with_orders bo ON b.id = bo.book_id
;
```

查询结果如下：

```
+------------+-------------------------+----------------+--------+
| book_id    | book_title              | average_rating | orders |
+------------+-------------------------+----------------+--------+
|  481008467 | The Documentary of goat |         2.0000 |     16 |
| 2224531102 | Brandt Skiles           |         2.7143 |     17 |
| 2641301356 | Sheridan Bashirian      |         2.4211 |     12 |
| 4154439164 | Karson Streich          |         2.5833 |     19 |
+------------+-------------------------+----------------+--------+
4 rows in set (0.06 sec)
```

在这个 SQL 语句，定义了三个 CTE 块，CTE 块之间使用 , 进行分隔。

先在 CTE 块 books_authored_by_rm 当中将该作者（作者 ID 为 2299112019）所编写的书查出来，然后在 books_with_average_ratings 和 books_with_orders 中分别查出这些书的平均评分和订单数，最后通过 JOIN 语句进行汇总。

值得注意的是，books_authored_by_rm 中的查询只会执行一次，TiDB 会开辟一块临时空间对查询的结果进行缓存，当 books_with_average_ratings 和 books_with_orders 引用时会直接从该临时空间当中获取数据。

> 建议：
>
> 当默认的 CTE 查询执行效率不高时，你可以使用MERGE() hint，将 CTE 子查询拓展到外部查询，以此提高执行效率。

4.7.7.1.2 递归的 CTE

递归的公共表表达式可以使用如下语法进行定义：

```
WITH RECURSIVE <query_name> AS (
    <query_definition>
)
SELECT ... FROM <query_name>;
```

比较经典的例子是通过递归的 CTE 生成一组斐波那契数：

```
WITH RECURSIVE fibonacci (n, fib_n, next_fib_n) AS
(
  SELECT 1, 0, 1
  UNION ALL
  SELECT n + 1, next_fib_n, fib_n + next_fib_n FROM fibonacci WHERE n < 10
)
SELECT * FROM fibonacci;
```

查询结果如下：

```
+------+-------+------------+
| n    | fib_n | next_fib_n |
+------+-------+------------+
|    1 |     0 |          1 |
|    2 |     1 |          1 |
|    3 |     1 |          2 |
|    4 |     2 |          3 |
|    5 |     3 |          5 |
|    6 |     5 |          8 |
|    7 |     8 |         13 |
|    8 |    13 |         21 |
|    9 |    21 |         34 |
|   10 |    34 |         55 |
+------+-------+------------+
10 rows in set (0.00 sec)
```

4.7.7.2 扩展阅读

- WITH

4.7.8 读取副本数据

4.7.8.1 Follower Read

本章将介绍使用 Follower Read 在特定情况下加速查询的方法。

4.7.8.1.1 简介

在 TiDB 当中，数据是以Region 为单位，分散在集群中所有的节点上进行存储的。一个 Region 可以存在多个副本，副本又分为一个 leader 和多个 follower。当 leader 上的数据发生变化时，TiDB 会将数据同步更新到 follower。

默认情况下，TiDB 只会在同一个 Region 的 leader 上读写数据。当系统中存在读取热点 Region 导致 leader 资源紧张成为整个系统读取瓶颈时，启用 Follower Read 功能可明显降低 leader 的负担，并且通过在多个 follower 之间均衡负载，显著地提升整体系统的吞吐能力。

4.7.8.1.2 何时使用

优化读热点

你可以在TiDB Dashboard 流量可视化页面当中通过可视化的方法分析你的应用程序是否存在热点 Region。你可以通过将「指标选择框」选择到 `Read (bytes)` 或 `Read (keys)` 查看是否存在读取热点 Region。

如果发现确实存在热点问题，你可以通过阅读TiDB 热点问题处理章节进行逐一排查，以便从应用程序层面上避免热点的产生。

如果读取热点的确无法避免或者改动的成本很大，你可以尝试通过 Follower Read 功能将读取请求更好的负载均衡到 follower region。

优化跨数据中心部署的延迟

如果 TiDB 集群是跨地区或跨数据中心部署的，一个 Region 的不同副本分布在不同的地区或数据中心，此时可以通过配置 Follower Read 为 `closest-adaptive` 或 `closest-replicas` 让 TiDB 优先从当前的数据中心执行读操作，这样可以大幅降低读操作的延迟和流量开销。具体原理可参考Follower Read。

4.7.8.1.3 开启 Follower Read

在 SQL 中，你可以将变量 `tidb_replica_read` 的值（默认为 `leader`）设置为 `follower`、`leader-and-follower`、`prefer-leader`、`closest-replicas` 或 `closest-adaptive` 开启 TiDB 的 Follower Read 功能：

```
SET [GLOBAL] tidb_replica_read = 'follower';
```

你可以通过Follower Read 使用方式 了解该变量的更多细节。

在 Java 语言当中，可以定义一个 `FollowerReadHelper` 类用于开启 Follower Read 功能：

```
public enum FollowReadMode {
    LEADER("leader"),
    FOLLOWER("follower"),
    LEADER_AND_FOLLOWER("leader-and-follower"),
    CLOSEST_REPLICA("closest-replica"),
    CLOSEST_ADAPTIVE("closest-adaptive"),
    PREFER_LEADER("prefer-leader");

    private final String mode;

    FollowReadMode(String mode) {
        this.mode = mode;
```

```
    }

    public String getMode() {
        return mode;
    }
}

public class FollowerReadHelper {

    public static void setSessionReplicaRead(Connection conn, FollowReadMode mode) throws
        ↪ SQLException {
        if (mode == null) mode = FollowReadMode.LEADER;
        PreparedStatement stmt = conn.prepareStatement(
            "SET @@tidb_replica_read = ?;"
        );
        stmt.setString(1, mode.getMode());
        stmt.execute();
    }

    public static void setGlobalReplicaRead(Connection conn, FollowReadMode mode) throws
        ↪ SQLException {
        if (mode == null) mode = FollowReadMode.LEADER;
        PreparedStatement stmt = conn.prepareStatement(
            "SET GLOBAL @@tidb_replica_read = ?;"
        );
        stmt.setString(1, mode.getMode());
        stmt.execute();
    }

}
```

在需要使用从 Follower 节点读取数据时，通过 `setSessionReplicaRead(conn, FollowReadMode.LEADER_AND_FOLLOWER ↪ )` 方法在当前 Session 开启能够在 Leader 节点和 Follower 节点进行负载均衡的 Follower Read 功能，当连接断开时，会恢复到原来的模式。

```
public static class AuthorDAO {

    // Omit initialization of instance variables...

    public void getAuthorsByFollowerRead() throws SQLException {
        try (Connection conn = ds.getConnection()) {
            // Enable the follower read feature.
            FollowerReadHelper.setSessionReplicaRead(conn, FollowReadMode.LEADER_AND_FOLLOWER);

            // Read the authors list for 100000 times.
            Random random = new Random();
```

```
            for (int i = 0; i < 100000; i++) {
                Integer birthYear = 1920 + random.nextInt(100);
                List<Author> authors = this.getAuthorsByBirthYear(birthYear);
                System.out.println(authors.size());
            }
        }
    }

    public List<Author> getAuthorsByBirthYear(Integer birthYear) throws SQLException {
        List<Author> authors = new ArrayList<>();
        try (Connection conn = ds.getConnection()) {
            PreparedStatement stmt = conn.prepareStatement("SELECT id, name FROM authors WHERE
                ↪ birth_year = ?");
            stmt.setInt(1, birthYear);
            ResultSet rs = stmt.executeQuery();
            while (rs.next()) {
                Author author = new Author();
                author.setId( rs.getLong("id"));
                author.setName(rs.getString("name"));
                authors.add(author);
            }
        }
        return authors;
    }
}
```

4.7.8.1.4 扩展阅读

- Follower Read
- TiDB 热点问题处理
- TiDB Dashboard 流量可视化页面

4.7.8.2 Stale Read

Stale Read 是一种读取历史数据版本的机制，通过 Stale Read 功能，你能从指定时间点或时间范围内读取对应的历史数据，从而在数据强一致需求没那么高的场景降低读取数据的延迟。当使用 Stale Read 时，TiDB 默认会随机选择一个副本来读取数据，因此能利用所有保存有副本的节点的处理能力。

在实际的使用当中，请根据具体的场景判断是否适合在 TiDB 当中开启 Stale Read 功能。如果你的应用程序不能容忍读到非实时的数据，请勿使用 Stale Read，否则读到的数据可能不是最新成功写入的数据。

TiDB 提供了语句级别、事务级别、会话级别三种级别的 Stale Read 功能，接下来将逐一进行介绍：

4.7.8.2.1 引入

在Bookshop 应用程序当中，你可以通过下面的 SQL 语句查询出最新出版的书籍以及它们的价格：

```
SELECT id, title, type, price FROM books ORDER BY published_at DESC LIMIT 5;
```

运行结果为：

```
+------------+------------------------------+----------------------+--------+
| id         | title                        | type                 | price  |
+------------+------------------------------+----------------------+--------+
| 3181093216 | The Story of Droolius Caesar | Novel                | 100.00 |
| 1064253862 | Collin Rolfson               | Education & Reference |  92.85 |
| 1748583991 | The Documentary of cat       | Magazine             | 159.75 |
|  893930596 | Myrl Hills                   | Education & Reference | 356.85 |
| 3062833277 | Keven Wyman                  | Life                 | 477.91 |
+------------+------------------------------+----------------------+--------+
5 rows in set (0.02 sec)
```

看到此时（2022-04-20 15:20:00）的列表中，The Story of Droolius Caesar 这本小说的价格为 100.0 元。

于此同时，卖家发现这本书很受欢迎，于是他通过下面的 SQL 语句将这本书的价格高到了 150.0 元。

```
UPDATE books SET price = 150 WHERE id = 3181093216;
```

运行结果为：

```
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

当再次查询最新书籍列表时，发现这本书确实涨价了。

```
+------------+------------------------------+----------------------+--------+
| id         | title                        | type                 | price  |
+------------+------------------------------+----------------------+--------+
| 3181093216 | The Story of Droolius Caesar | Novel                | 150.00 |
| 1064253862 | Collin Rolfson               | Education & Reference |  92.85 |
| 1748583991 | The Documentary of cat       | Magazine             | 159.75 |
|  893930596 | Myrl Hills                   | Education & Reference | 356.85 |
| 3062833277 | Keven Wyman                  | Life                 | 477.91 |
+------------+------------------------------+----------------------+--------+
5 rows in set (0.01 sec)
```

如果不要求必须使用最新的数据，可以让 TiDB 通过 Stale Read 功能直接返回可能已经过期的历史数据，避免使用强一致性读时数据同步带来的延迟。

假设在 Bookshop 应用程序当中，在用户浏览书籍列表页时，不对书籍价格的实时性进行要求，只有用户在点击查看书籍详情页或下单时才去获取实时的价格信息，可以借助 Stale Read 能力来进一步提升应用的吞吐量。

4.7.8.2.2 语句级别

在 SQL 中，你可以在上述价格的查询语句当中添加上 `AS OF TIMESTAMP <datetime>` 语句查看到固定时间点之前这本书的价格。

```
SELECT id, title, type, price FROM books AS OF TIMESTAMP '2022-04-20 15:20:00' ORDER BY
    ↪ published_at DESC LIMIT 5;
```

运行结果为：

```
+------------+------------------------------+----------------------+--------+
| id         | title                        | type                 | price  |
+------------+------------------------------+----------------------+--------+
| 3181093216 | The Story of Droolius Caesar | Novel                | 100.00 |
| 1064253862 | Collin Rolfson               | Education & Reference |  92.85 |
| 1748583991 | The Documentary of cat       | Magazine             | 159.75 |
|  893930596 | Myrl Hills                   | Education & Reference | 356.85 |
| 3062833277 | Keven Wyman                  | Life                 | 477.91 |
+------------+------------------------------+----------------------+--------+
5 rows in set (0.01 sec)
```

除了指定精确的时间点外，你还可以通过：

- `AS OF TIMESTAMP NOW()- INTERVAL 10 SECOND` 表示读取 10 秒前最新的数据。
- `AS OF TIMESTAMP TIDB_BOUNDED_STALENESS('2016-10-08 16:45:26', '2016-10-08 16:45:29')` 表示读取在 2016 年 10 月 8 日 16 点 45 分 26 秒到 29 秒的时间范围内尽可能新的数据。
- `AS OF TIMESTAMP TIDB_BOUNDED_STALENESS(NOW()- INTERVAL 20 SECOND, NOW())` 表示读取 20 秒前到现在的时间范围内尽可能新的数据。

需要注意的是，设定的时间戳或时间戳的范围不能过早或晚于当前时间。

过期的数据在 TiDB 当中会由垃圾回收器进行回收，数据在被清除之前会被保留一小段时间，这段时间被称为GC Life Time (默认 10 分钟)。每次进行 GC 时，将以当前时间减去该时间周期的值作为 GC Safe Point。如果尝试读取 GC Safe Point 之前数据，TiDB 会报如下错误：

```
ERROR 9006 (HY000): GC life time is shorter than transaction duration...
```

如果给出的时间戳是一个未来的时间节点，TiDB 会报如下错误：

```
ERROR 9006 (HY000): cannot set read timestamp to a future time.
```

在 Java 中的示例如下：

```
public class BookDAO {

    // Omit some code...

    public List<Book> getTop5LatestBooks() throws SQLException {
        List<Book> books = new ArrayList<>();
        try (Connection conn = ds.getConnection()) {
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery("""
```

```
            SELECT id, title, type, price FROM books ORDER BY published_at DESC LIMIT 5;
            """);
            while (rs.next()) {
                Book book = new Book();
                book.setId(rs.getLong("id"));
                book.setTitle(rs.getString("title"));
                book.setType(rs.getString("type"));
                book.setPrice(rs.getDouble("price"));
                books.add(book);
            }
        }
        return books;
    }

    public void updateBookPriceByID(Long id, Double price) throws SQLException {
        try (Connection conn = ds.getConnection()) {
            PreparedStatement stmt = conn.prepareStatement("""
            UPDATE books SET price = ? WHERE id = ?;
            """);
            stmt.setDouble(1, price);
            stmt.setLong(2, id);
            int affects = stmt.executeUpdate();
            if (affects == 0) {
                throw new SQLException("Failed to update the book with id: " + id);
            }
        }
    }

    public List<Book> getTop5LatestBooksWithStaleRead(Integer seconds) throws SQLException {
        List<Book> books = new ArrayList<>();
        try (Connection conn = ds.getConnection()) {
            PreparedStatement stmt = conn.prepareStatement("""
            SELECT id, title, type, price FROM books AS OF TIMESTAMP NOW() - INTERVAL ? SECOND
                ↪ ORDER BY published_at DESC LIMIT 5;
            """);
            stmt.setInt(1, seconds);
            ResultSet rs = stmt.executeQuery();
            while (rs.next()) {
                Book book = new Book();
                book.setId(rs.getLong("id"));
                book.setTitle(rs.getString("title"));
                book.setType(rs.getString("type"));
                book.setPrice(rs.getDouble("price"));
                books.add(book);
            }
```

```
        } catch (SQLException e) {
            if ("HY000".equals(e.getSQLState()) && e.getErrorCode() == 1105) {
                System.out.println("WARN: cannot set read timestamp to a future time.");
            } else if ("HY000".equals(e.getSQLState()) && e.getErrorCode() == 9006) {
                System.out.println("WARN: GC life time is shorter than transaction duration.");
            } else {
                throw e;
            }
        }
        return books;
    }
}
```

```
List<Book> top5LatestBooks = bookDAO.getTop5LatestBooks();

if (top5LatestBooks.size() > 0) {
    System.out.println("The latest book price (before update): " + top5LatestBooks.get(0).
        ↪ getPrice());

    Book book = top5LatestBooks.get(0);
    bookDAO.updateBookPriceByID(book.getId(), book.price + 10);

    top5LatestBooks = bookDAO.getTop5LatestBooks();
    System.out.println("The latest book price (after update): " + top5LatestBooks.get(0).getPrice
        ↪ ());

    // Use the stale read.
    top5LatestBooks = bookDAO.getTop5LatestBooksWithStaleRead(5);
    System.out.println("The latest book price (maybe stale): " + top5LatestBooks.get(0).getPrice
        ↪ ());

    // Try to stale read the data at the future time.
    bookDAO.getTop5LatestBooksWithStaleRead(-5);

    // Try to stale read the data before 20 minutes.
    bookDAO.getTop5LatestBooksWithStaleRead(20 * 60);
}
```

通过结果可以看到通过 Stale Read 读取到了更新之前的价格 100.00 元。

```
The latest book price (before update): 100.00
The latest book price (after update): 150.00
The latest book price (maybe stale): 100.00
WARN: cannot set read timestamp to a future time.
WARN: GC life time is shorter than transaction duration.
```

4.7.8.2.3 事务级别

通过 START TRANSACTION READ ONLY AS OF TIMESTAMP 语句，你可以开启一个基于历史时间的只读事务，该事务基于所提供的历史时间来读取历史数据。

在 SQL 中的示例如下：

```
START TRANSACTION READ ONLY AS OF TIMESTAMP NOW() - INTERVAL 5 SECOND;
```

尝试通过 SQL 查询最新书籍的价格，发现 The Story of Droolius Caesar 这本书的价格还是更新之前的价格 100.0 元。

```
SELECT id, title, type, price FROM books ORDER BY published_at DESC LIMIT 5;
```

运行结果为：

```
+------------+------------------------------+----------------------+--------+
| id         | title                        | type                 | price  |
+------------+------------------------------+----------------------+--------+
| 3181093216 | The Story of Droolius Caesar | Novel                | 100.00 |
| 1064253862 | Collin Rolfson               | Education & Reference |  92.85 |
| 1748583991 | The Documentary of cat       | Magazine             | 159.75 |
|  893930596 | Myrl Hills                   | Education & Reference | 356.85 |
| 3062833277 | Keven Wyman                  | Life                 | 477.91 |
+------------+------------------------------+----------------------+--------+
5 rows in set (0.01 sec)
```

随后通过 COMMIT; 语句提交事务，当事务结束后，又可以重新读取到最新数据：

```
+------------+------------------------------+----------------------+--------+
| id         | title                        | type                 | price  |
+------------+------------------------------+----------------------+--------+
| 3181093216 | The Story of Droolius Caesar | Novel                | 150.00 |
| 1064253862 | Collin Rolfson               | Education & Reference |  92.85 |
| 1748583991 | The Documentary of cat       | Magazine             | 159.75 |
|  893930596 | Myrl Hills                   | Education & Reference | 356.85 |
| 3062833277 | Keven Wyman                  | Life                 | 477.91 |
+------------+------------------------------+----------------------+--------+
5 rows in set (0.01 sec)
```

在 Java 中，可以先定义一个事务的工具类，将开启事务级别 Stale Read 的命令封装成工具方法。

```
public static class StaleReadHelper {

    public static void startTxnWithStaleRead(Connection conn, Integer seconds) throws
        ↪ SQLException {
        conn.setAutoCommit(false);
        PreparedStatement stmt = conn.prepareStatement(
            "START TRANSACTION READ ONLY AS OF TIMESTAMP NOW() - INTERVAL ? SECOND;"
        );
```

```
        stmt.setInt(1, seconds);
        stmt.execute();
    }

}
```

然后在 BookDAO 类当中定义一个通过事务开启 Stale Read 功能的方法，在方法内查询最新的书籍列表，但是不再在查询语句中添加 `AS OF TIMESTAMP`。

```
public class BookDAO {

    // Omit some code...

    public List<Book> getTop5LatestBooksWithTxnStaleRead(Integer seconds) throws SQLException {
        List<Book> books = new ArrayList<>();
        try (Connection conn = ds.getConnection()) {
            // Start a read only transaction.
            TxnHelper.startTxnWithStaleRead(conn, seconds);

            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery("""
            SELECT id, title, type, price FROM books ORDER BY published_at DESC LIMIT 5;
            """);
            while (rs.next()) {
                Book book = new Book();
                book.setId(rs.getLong("id"));
                book.setTitle(rs.getString("title"));
                book.setType(rs.getString("type"));
                book.setPrice(rs.getDouble("price"));
                books.add(book);
            }

            // Commit transaction.
            conn.commit();
        } catch (SQLException e) {
            if ("HY000".equals(e.getSQLState()) && e.getErrorCode() == 1105) {
                System.out.println("WARN: cannot set read timestamp to a future time.");
            } else if ("HY000".equals(e.getSQLState()) && e.getErrorCode() == 9006) {
                System.out.println("WARN: GC life time is shorter than transaction duration.");
            } else {
                throw e;
            }
        }
        return books;
    }
}
```

```
List<Book> top5LatestBooks = bookDAO.getTop5LatestBooks();

if (top5LatestBooks.size() > 0) {
    System.out.println("The latest book price (before update): " + top5LatestBooks.get(0).
        ↪ getPrice());

    Book book = top5LatestBooks.get(0);
    bookDAO.updateBookPriceByID(book.getId(), book.price + 10);

    top5LatestBooks = bookDAO.getTop5LatestBooks();
    System.out.println("The latest book price (after update): " + top5LatestBooks.get(0).getPrice
        ↪ ());

    // Use the stale read.
    top5LatestBooks = bookDAO.getTop5LatestBooksWithTxnStaleRead(5);
    System.out.println("The latest book price (maybe stale): " + top5LatestBooks.get(0).getPrice
        ↪ ());

    // After the stale read transaction is committed.
    top5LatestBooks = bookDAO.getTop5LatestBooks();
    System.out.println("The latest book price (after the transaction commit): " + top5LatestBooks
        ↪ .get(0).getPrice());
}
```

输出结果：

```
The latest book price (before update): 100.00
The latest book price (after update): 150.00
The latest book price (maybe stale): 100.00
The latest book price (after the transaction commit): 150
```

通过 SET TRANSACTION READ ONLY AS OF TIMESTAMP 语句，你可以将当前事务或下一个事务设置为基于指定历史时间的只读事务。该事务将会基于所提供的历史时间来读取历史数据。

例如，可以通过下面这个 SQL 将已开启的事务切换到只读模式，通过 AS OF TIMESTAMP 语句开启能够读取 5 秒前的历史数据 Stale Read 功能。

```
SET TRANSACTION READ ONLY AS OF TIMESTAMP NOW() - INTERVAL 5 SECOND;
```

可以先定义一个事务的工具类，将开启事务级别 Stale Read 的命令封装成工具方法。

```
public static class TxnHelper {

    public static void setTxnWithStaleRead(Connection conn, Integer seconds) throws SQLException
        ↪ {
        PreparedStatement stmt = conn.prepareStatement(
```

```
            "SET TRANSACTION READ ONLY AS OF TIMESTAMP NOW() - INTERVAL ? SECOND;"
        );
        stmt.setInt(1, seconds);
        stmt.execute();
    }

}
```

然后在 BookDAO 类当中定义一个通过事务开启 Stale Read 功能的方法，在方法内查询最新的书籍列表，但是不再在查询语句中添加 `AS OF TIMESTAMP`。

```
public class BookDAO {

    // Omit some code...

    public List<Book> getTop5LatestBooksWithTxnStaleRead2(Integer seconds) throws SQLException {
        List<Book> books = new ArrayList<>();
        try (Connection conn = ds.getConnection()) {
            // Start a read only transaction.
            conn.setAutoCommit(false);
            StaleReadHelper.setTxnWithStaleRead(conn, seconds);

            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery("""
            SELECT id, title, type, price FROM books ORDER BY published_at DESC LIMIT 5;
            """);
            while (rs.next()) {
                Book book = new Book();
                book.setId(rs.getLong("id"));
                book.setTitle(rs.getString("title"));
                book.setType(rs.getString("type"));
                book.setPrice(rs.getDouble("price"));
                books.add(book);
            }

            // Commit transaction.
            conn.commit();
        } catch (SQLException e) {
            if ("HY000".equals(e.getSQLState()) && e.getErrorCode() == 1105) {
                System.out.println("WARN: cannot set read timestamp to a future time.");
            } else if ("HY000".equals(e.getSQLState()) && e.getErrorCode() == 9006) {
                System.out.println("WARN: GC life time is shorter than transaction duration.");
            } else {
                throw e;
            }
        }
```

```
        return books;
    }
}
```

4.7.8.2.4 会话级别

为支持读取历史版本数据，TiDB 从 5.4 版本起引入了一个新的系统变量 tidb_read_staleness。系统变量 tidb_read_staleness 用于设置当前会话允许读取的历史数据范围，其数据类型为 int，作用域为 SESSION。

在会话中开启 Stale Read：

```
SET @@tidb_read_staleness="-5";
```

比如，如果该变量的值设置为 -5，TiDB 会在 5 秒时间范围内，保证 TiKV 或者 TiFlash 拥有对应历史版本数据的情况下，选择尽可能新的一个时间戳。

关闭会话当中的 Stale Read：

```
set @@tidb_read_staleness="";
```

在 Java 中示例如下：

```
public static class StaleReadHelper{

    public static void enableStaleReadOnSession(Connection conn, Integer seconds) throws
        ↪ SQLException {
        PreparedStatement stmt = conn.prepareStatement(
            "SET @@tidb_read_staleness= ?;"
        );
        stmt.setString(1, String.format("-%d", seconds));
        stmt.execute();
    }

    public static void disableStaleReadOnSession(Connection conn) throws SQLException {
        PreparedStatement stmt = conn.prepareStatement(
            "SET @@tidb_read_staleness=\"\";"
        );
        stmt.execute();
    }

}
```

4.7.8.2.5 扩展阅读

- Stale Read 功能的使用场景
- 使用 AS OF TIMESTAMP 语法读取历史数据
- 通过系统变量 tidb_read_staleness 读取历史数据

4.7.9 HTAP 查询

HTAP 是 Hybrid Transactional / Analytical Processing 的缩写。传统意义上，数据库往往专为交易或者分析场景设计，因而数据平台往往需要被切分为 Transactional Processing 和 Analytical Processing 两个部分，而数据需要从交易库复制到分析型数据库以便快速响应分析查询。而 TiDB 数据库则可以同时承担交易和分析两种职能，这大大简化了数据平台的建设，也能让用户使用更新鲜的数据进行分析。

在 TiDB 当中，同时拥有面向在线事务处理的行存储引擎 TiKV 与面向实时分析场景的列存储引擎 TiFlash 两套存储引擎。数据在行存 (Row-Store) 与列存 (Columnar-Store) 同时存在，自动同步，保持强一致性。行存为在线事务处理 OLTP 提供优化，列存则为在线分析处理 OLAP 提供性能优化。

在创建数据库章节当中，已经介绍了如何开启 TiDB 的 HTAP 能力。下面将进一步介绍如何使用 HTAP 能力更快地分析数据。

4.7.9.1 数据准备

在开始之前，你可以通过 `tiup demo` 命令导入更加大量的示例数据，例如：

```
tiup demo bookshop prepare --users=200000 --books=500000 --authors=100000 --ratings=1000000 --
    ↪ orders=1000000 --host 127.0.0.1 --port 4000 --drop-tables
```

或使用 TiDB Cloud 的 Import 功能导入预先准备好的示例数据。

4.7.9.2 窗口函数

在使用数据库时，除了希望它能够存储想要记录的数据，能够实现诸如下单买书、给书籍评分等业务功能外，可能还需要对已有的数据进行分析，以便根据数据作出进一步的运营和决策。

在单表读取章节当中，已经介绍了如何使用聚合查询来分析数据的整体情况，在更为复杂的使用场景下，你可能希望多个聚合查询的结果汇总在一个查询当中。例如：你想要对某一本书的订单量的历史趋势有所了解，就需要在每个月都对所有订单数据进行一次聚合求 `sum`，然后将 `sum` 结果汇总在一起才能够得到历史的趋势变化数据。

为了方便用户进行此类分析，TiDB 从 3.0 版本开始便支持了窗口函数功能，窗口函数为每一行数据提供了跨行数据访问的能力，不同于常规的聚合查询，窗口函数在对数据行进行聚合时不会导致结果集被合并成单行数据。

与聚合函数类似，窗口函数在使用时也需要搭配一套固定的语法：

```
SELECT
    window_function() OVER ([partition_clause] [order_clause] [frame_clause]) AS alias
FROM
    table_name
```

4.7.9.2.1 ORDER BY 子句

例如：可以利用聚合窗口函数 `sum()` 函数的累加效果来实现对某一本书的订单量的历史趋势的分析:

```
WITH orders_group_by_month AS (
  SELECT DATE_FORMAT(ordered_at, '%Y-%c') AS month, COUNT(*) AS orders
  FROM orders
  WHERE book_id = 3461722937
  GROUP BY 1
)
SELECT
month,
SUM(orders) OVER(ORDER BY month ASC) as acc
FROM orders_group_by_month
ORDER BY month ASC;
```

sum() 函数会在 OVER 子句当中通过 ORDER BY 子句指定的排序方式按顺序对数据进行累加，累加的结果如下：

```
+---------+-------+
| month   | acc   |
+---------+-------+
| 2011-5  |     1 |
| 2011-8  |     2 |
| 2012-1  |     3 |
| 2012-2  |     4 |
| 2013-1  |     5 |
| 2013-3  |     6 |
| 2015-11 |     7 |
| 2015-4  |     8 |
| 2015-8  |     9 |
| 2017-11 |    10 |
| 2017-5  |    11 |
| 2019-5  |    13 |
| 2020-2  |    14 |
+---------+-------+
13 rows in set (0.01 sec)
```

将得到的数据通过一个横轴为时间，纵轴为累计订单量的折线图进行可视化，便可以轻松地通过折线图的斜率变化宏观地了解到这本书的历史订单的增长趋势。

4.7.9.2.2 PARTITION BY 子句

把需求变得更复杂一点，假设想要分析不同类型书的历史订单增长趋势，并且希望将这些数据通过同一个多系列折线图进行呈现。

可以利用 PARTITION BY 子句根据书的类型进行分组，对不同类型的书籍分别统计它们的订单历史订单累计量。

```
WITH orders_group_by_month AS (
    SELECT
```

```
        b.type AS book_type,
        DATE_FORMAT(ordered_at, '%Y-%c') AS month,
        COUNT(*) AS orders
    FROM orders o
    LEFT JOIN books b ON o.book_id = b.id
    WHERE b.type IS NOT NULL
    GROUP BY book_type, month
), acc AS (
    SELECT
        book_type,
        month,
        SUM(orders) OVER(PARTITION BY book_type ORDER BY book_type, month ASC) as acc
    FROM orders_group_by_month
    ORDER BY book_type, month ASC
)
SELECT * FROM acc;
```

查询结果如下：

```
+------------------------------+---------+------+
| book_type                    | month   | acc  |
+------------------------------+---------+------+
| Magazine                     | 2011-10 |    1 |
| Magazine                     | 2011-8  |    2 |
| Magazine                     | 2012-5  |    3 |
| Magazine                     | 2013-1  |    4 |
| Magazine                     | 2013-6  |    5 |
...
| Novel                        | 2011-3  |   13 |
| Novel                        | 2011-4  |   14 |
| Novel                        | 2011-6  |   15 |
| Novel                        | 2011-8  |   17 |
| Novel                        | 2012-1  |   18 |
| Novel                        | 2012-2  |   20 |
...
| Sports                       | 2021-4  |   49 |
| Sports                       | 2021-7  |   50 |
| Sports                       | 2022-4  |   51 |
+------------------------------+---------+------+
1500 rows in set (1.70 sec)
```

4.7.9.2.3 非聚合窗口函数

除此之外，TiDB 还提供了一些非聚合的窗口函数，可以借助这些函数实现更加丰富分析查询。

例如，在前面的分页查询章节当中，已经介绍了如何巧妙地利用 row_number() 函数实现高效的分页批处理能力。

4.7.9.3 混合负载

当将 TiDB 应用于在线实时分析处理的混合负载场景时，开发人员只需要提供一个入口，TiDB 将自动根据业务类型选择不同的处理引擎。

4.7.9.3.1 开启列存副本

TiDB 默认使用的存储引擎 TiKV 是行存的，你可以通过阅读开启 HTAP 能力章节，在进行后续步骤前，先通过如下 SQL 对 books 与 orders 表添加 TiFlash 列存副本：

```
ALTER TABLE books SET TIFLASH REPLICA 1;
ALTER TABLE orders SET TIFLASH REPLICA 1;
```

通过执行下面的 SQL 语句可以查看到 TiDB 创建列存副本的进度：

```
SELECT * FROM information_schema.tiflash_replica WHERE TABLE_SCHEMA = 'bookshop' and TABLE_NAME =
    ↪  'books';
SELECT * FROM information_schema.tiflash_replica WHERE TABLE_SCHEMA = 'bookshop' and TABLE_NAME =
    ↪  'orders';
```

当 PROGRESS 列为 1 时表示同步进度完成度达到 100%，AVAILABLE 列为 1 表示副本当前可用。

```
+--------------+------------+----------+---------------+-----------------+-----------+----------+
| TABLE_SCHEMA | TABLE_NAME | TABLE_ID | REPLICA_COUNT | LOCATION_LABELS | AVAILABLE | PROGRESS |
+--------------+------------+----------+---------------+-----------------+-----------+----------+
| bookshop     | books      |      143 |             1 |                 |         1 |        1 |
+--------------+------------+----------+---------------+-----------------+-----------+----------+
1 row in set (0.07 sec)
+--------------+------------+----------+---------------+-----------------+-----------+----------+
| TABLE_SCHEMA | TABLE_NAME | TABLE_ID | REPLICA_COUNT | LOCATION_LABELS | AVAILABLE | PROGRESS |
+--------------+------------+----------+---------------+-----------------+-----------+----------+
| bookshop     | orders     |      147 |             1 |                 |         1 |        1 |
+--------------+------------+----------+---------------+-----------------+-----------+----------+
1 row in set (0.07 sec)
```

副本添加完成之后，你可以通过使用 EXPLAIN 语句查看上面窗口函数示例 SQL 的执行计划。你会发现执行计划当中已经出现了 cop[tiflash] 字样，说明 TiFlash 引擎已经开始发挥作用了。

再次执行示例 SQL，查询结果如下：

```
+------------------------------+---------+------+
| book_type                    | month   | acc  |
+------------------------------+---------+------+
| Magazine                     | 2011-10 |    1 |
| Magazine                     | 2011-8  |    2 |
```

```
| Magazine                     | 2012-5  |    3 |
| Magazine                     | 2013-1  |    4 |
| Magazine                     | 2013-6  |    5 |
...
| Novel                        | 2011-3  |   13 |
| Novel                        | 2011-4  |   14 |
| Novel                        | 2011-6  |   15 |
| Novel                        | 2011-8  |   17 |
| Novel                        | 2012-1  |   18 |
| Novel                        | 2012-2  |   20 |
...
| Sports                       | 2021-4  |   49 |
| Sports                       | 2021-7  |   50 |
| Sports                       | 2022-4  |   51 |
+------------------------------+---------+------+
1500 rows in set (0.79 sec)
```

通过对比前后两次的执行结果，你会发现使用 TiFlash 处理有查询速度有了较为明显的提升（当数据量更大时，提升会更为显著）。这是因为在使用窗口函数时往往需要对某些列的数据进行全表扫描，相比行存的 TiKV，列存的 TiFlash 更加适合来处理这类分析型任务的负载。而对于 TiKV 来说，如果能够通过主键或索引快速地将所要查询的行数减少，往往查询速度也会非常快，而且所消耗的资源一般相对 TiFlash 而言会更少。

4.7.9.3.2 指定查询引擎

尽管 TiDB 会使用基于成本的优化器（CBO）自动地根据代价估算选择是否使用 TiFlash 副本。但是在实际使用当中，如果你非常确定查询的类型，推荐你使用Optimizer Hints 明确的指定查询所使用的执行引擎，避免因为优化器的优化结果不同，导致应用程序性能出现波动。

你可以像下面的 SQL 一样在 SELECT 语句中通过 Hint `/*+ read_from_storage(engine_name[table_name])*/` 指定查询时需要使用的查询引擎。

> 注意：
>
> 1. 如果你的表使用了别名，你应该将 Hints 当中的 table_name 替代为 alias_name，否则 Hints 会失效。
> 2. 另外，对公共表表达式设置 read_from_storage Hint 是不起作用的。

```
WITH orders_group_by_month AS (
    SELECT
        /*+ read_from_storage(tikv[o]) */
        b.type AS book_type,
        DATE_FORMAT(ordered_at, '%Y-%c') AS month,
        COUNT(*) AS orders
    FROM orders o
```

```
    LEFT JOIN books b ON o.book_id = b.id
    WHERE b.type IS NOT NULL
    GROUP BY book_type, month
), acc AS (
    SELECT
        book_type,
        month,
        SUM(orders) OVER(PARTITION BY book_type ORDER BY book_type, month ASC) as acc
    FROM orders_group_by_month mo
    ORDER BY book_type, month ASC
)
SELECT * FROM acc;
```

如果你通过 EXPLAIN 语句查看上面 SQL 的执行计划，你会发现 task 列中会同时出现 cop[tiflash] 和 cop[tikv]，这意味着 TiDB 在处理这个查询的时候会同时调度行存查询引擎和列存查询引擎来完成查询任务。需要指出的是，因为 tiflash 和 tikv 存储引擎通常属于不同的计算节点，所以两种查询类型互相之间不受影响。

你可以通过阅读使用 TiDB 读取 TiFlash 小节进一步了解 TiDB 如何选择使用 TiFlash 作为查询引擎。

4.7.9.4 扩展阅读

- HTAP 快速上手指南
- HTAP 深入探索指南
- 窗口函数
- 使用 TiFlash

4.8 事务

4.8.1 事务概览

TiDB 支持完整的分布式事务，提供乐观事务与悲观事务（TiDB 3.0 中引入）两种事务模型。本文主要介绍涉及到事务的语句、乐观事务和悲观事务、事务的隔离级别，以及乐观事务应用端重试和错误处理。

4.8.1.1 拓展学习视频

TiDB 特有功能与事务控制 - TiDB v6：了解可用于应用程序的 TiDB 独特功能，如 AUTO_RANDOM 及 AUTO_INCREMENT 特别注意事项、全局临时表、如何使用 TiFlash 启用 HTAP 以及放置策略等。

4.8.1.2 通用语句

本章介绍在 TiDB 中如何使用事务。将使用下面的示例来演示一个简单事务的控制流程：

Bob 要给 Alice 转账 20 元钱，当中至少包括两个操作：

- Bob 账户减少 20 元。
- Alice 账户增加 20 元。

事务可以确保以上两个操作要么都执行成功，要么都执行失败，不会出现钱平白消失或出现的情况。

使用bookshop 数据库中的 users 表，在表中插入一些示例数据：

```
INSERT INTO users (id, nickname, balance)
  VALUES (2, 'Bob', 200);
INSERT INTO users (id, nickname, balance)
  VALUES (1, 'Alice', 100);
```

现在，运行以下事务并解释每个语句的含义：

```
BEGIN;
  UPDATE users SET balance = balance - 20 WHERE nickname = 'Bob';
  UPDATE users SET balance = balance + 20 WHERE nickname= 'Alice';
COMMIT;
```

上述事务成功后，表应如下所示：

```
+----+--------------+---------+
| id | account_name | balance |
+----+--------------+---------+
|  1 | Alice        |  120.00 |
|  2 | Bob          |  180.00 |
+----+--------------+---------+
```

4.8.1.2.1 开启事务

要显式地开启一个新事务，既可以使用 BEGIN 语句，也可以使用 START TRANSACTION 语句，两者效果相同。语法：

```
BEGIN;
```

```
START TRANSACTION;
```

TiDB 的默认事务模式是悲观事务，你也可以明确指定开启乐观事务：

```
BEGIN OPTIMISTIC;
```

开启悲观事务：

```
BEGIN PESSIMISTIC;
```

如果执行以上语句时，当前 Session 正处于一个事务的中间过程，那么系统会先自动提交当前事务，再开启一个新的事务。

4.8.1.2.2 提交事务

COMMIT 语句用于提交 TiDB 在当前事务中进行的所有修改。语法：

```
COMMIT;
```

启用乐观事务前，请确保应用程序可正确处理 COMMIT 语句可能返回的错误。如果不确定应用程序将会如何处理，建议改为使用悲观事务。

4.8.1.2.3 回滚事务

ROLLBACK 语句用于回滚并撤销当前事务的所有修改。语法：

```
ROLLBACK;
```

回到之前转账示例，使用 ROLLBACK 回滚整个事务之后，Alice 和 Bob 的余额都未发生改变，当前事务的所有修改一起被取消。

```
TRUNCATE TABLE `users`;

INSERT INTO `users` (`id`, `nickname`, `balance`) VALUES (1, 'Alice', 100), (2, 'Bob', 200);

SELECT * FROM `users`;
+----+--------------+---------+
| id | nickname     | balance |
+----+--------------+---------+
|  1 | Alice        |  100.00 |
|  2 | Bob          |  200.00 |
+----+--------------+---------+

BEGIN;
  UPDATE `users` SET `balance` = `balance` - 20 WHERE `nickname`='Bob';
  UPDATE `users` SET `balance` = `balance` + 20 WHERE `nickname`='Alice';
ROLLBACK;

SELECT * FROM `users`;
+----+--------------+---------+
| id | nickname     | balance |
+----+--------------+---------+
|  1 | Alice        |  100.00 |
|  2 | Bob          |  200.00 |
+----+--------------+---------+
```

如果客户端连接中止或关闭，也会自动回滚该事务。

4.8.1.3 事务隔离级别

事务隔离级别是数据库事务处理的基础，ACID 中的 “I”，即 Isolation，指的就是事务的隔离性。

SQL-92 标准定义了 4 种隔离级别：读未提交 (READ UNCOMMITTED)、读已提交 (READ COMMITTED)、可重复读 (REPEATABLE READ)、串行化 (SERIALIZABLE)。详见下表：

Isolation Level	Dirty Write	Dirty Read	Fuzzy Read	Phantom
READ UNCOMMITTED	Not Possible	Possible	Possible	Possible
READ COMMITTED	Not Possible	Not possible	Possible	Possible
REPEATABLE READ	Not Possible	Not possible	Not possible	Possible
SERIALIZABLE	Not Possible	Not possible	Not possible	Not possible

TiDB 语法上支持设置 READ COMMITTED 和 REPEATABLE READ 两种隔离级别：

```
mysql> SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
ERROR 8048 (HY000): The isolation level 'READ-UNCOMMITTED' is not supported. Set
    ↪ tidb_skip_isolation_level_check=1 to skip this error
mysql> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
Query OK, 0 rows affected (0.00 sec)

mysql> SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
Query OK, 0 rows affected (0.00 sec)

mysql> SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
ERROR 8048 (HY000): The isolation level 'SERIALIZABLE' is not supported. Set
    ↪ tidb_skip_isolation_level_check=1 to skip this error
```

TiDB 实现了快照隔离 (Snapshot Isolation, SI) 级别的一致性。为与 MySQL 保持一致，又称其为“可重复读”。该隔离级别不同于ANSI 可重复读隔离级别 和MySQL 可重复读隔离级别。更多细节请阅读TiDB 事务隔离级别。

4.8.2 乐观事务和悲观事务

简单的讲，乐观事务模型就是直接提交，遇到冲突就回滚，悲观事务模型就是在真正提交事务前，先尝试对需要修改的资源上锁，只有在确保事务一定能够执行成功后，才开始提交。

对于乐观事务模型来说，比较适合冲突率不高的场景，因为直接提交大概率会成功，冲突是小概率事件，但是一旦遇到事务冲突，回滚的代价会比较大。

悲观事务的好处是对于冲突率高的场景，提前上锁的代价小于事后回滚的代价，而且还能以比较低的代价解决多个并发事务互相冲突导致谁也成功不了的场景。不过悲观事务在冲突率不高的场景并没有乐观事务处理高效。

从应用端实现的复杂度而言，悲观事务更直观，更容易实现。而乐观事务需要复杂的应用端重试机制来保证。

下面用bookshop 数据库中的表实现一个购书的例子来演示乐观事务和悲观事务的区别以及优缺点。购书流程主要包括：

1. 更新库存数量
2. 创建订单
3. 付款

这三个操作需要保证全部成功或者全部失败，并且在并发情况下要保证不超卖。

4.8.2.1 悲观事务

下面代码以悲观事务的方式，用两个线程模拟了两个用户并发买同一本书的过程，书店剩余 10 本，Bob 购买了 6 本，Alice 购买了 4 本。两个人几乎同一时间完成订单，最终，这本书的剩余库存为零。

当使用多个线程模拟多用户同时插入的情况时，需要使用一个线程安全的连接对象，这里使用 Java 当前较流行的连接池 HikariCP。

Golang 的 `sql.DB` 是并发安全的，无需引入外部包。

封装一个用于适配 TiDB 事务的工具包 util，编写以下代码备用：

```
package util

import (
    "context"
    "database/sql"
)

type TiDBSqlTx struct {
    *sql.Tx
    conn        *sql.Conn
    pessimistic bool
}

func TiDBSqlBegin(db *sql.DB, pessimistic bool) (*TiDBSqlTx, error) {
    ctx := context.Background()
    conn, err := db.Conn(ctx)
    if err != nil {
        return nil, err
    }
    if pessimistic {
        _, err = conn.ExecContext(ctx, "set @@tidb_txn_mode=?", "pessimistic")
    } else {
        _, err = conn.ExecContext(ctx, "set @@tidb_txn_mode=?", "optimistic")
    }
    if err != nil {
        return nil, err
    }
    tx, err := conn.BeginTx(ctx, nil)
    if err != nil {
        return nil, err
    }
    return &TiDBSqlTx{
        conn:        conn,
        Tx:          tx,
```

```
        pessimistic: pessimistic,
    }, nil
}

func (tx *TiDBSqlTx) Commit() error {
    defer tx.conn.Close()
    return tx.Tx.Commit()
}

func (tx *TiDBSqlTx) Rollback() error {
    defer tx.conn.Close()
    return tx.Tx.Rollback()
}
```

使用 Python 的 mysqlclient Driver 开启多个连接对象进行交互，线程之间不共享连接，以保证其线程安全。

4.8.2.1.1 1. 编写悲观事务示例

配置文件

在 Java 中，如果你使用 Maven 作为包管理，在 `pom.xml` 中的 `<dependencies>` 节点中，加入以下依赖来引入 HikariCP，同时设定打包目标，及 JAR 包启动的主类，完整的 `pom.xml` 如下所示:

```
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    ↪ instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.
      ↪ xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.pingcap</groupId>
  <artifactId>plain-java-txn</artifactId>
  <version>0.0.1</version>

  <name>plain-java-jdbc</name>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>17</maven.compiler.source>
    <maven.compiler.target>17</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
```

```
    <version>4.13.2</version>
    <scope>test</scope>
  </dependency>

  <!-- https://mvnrepository.com/artifact/mysql/mysql-connector-java -->
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.28</version>
  </dependency>

  <dependency>
    <groupId>com.zaxxer</groupId>
    <artifactId>HikariCP</artifactId>
    <version>5.0.1</version>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-assembly-plugin</artifactId>
      <version>3.3.0</version>
      <configuration>
        <descriptorRefs>
          <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
        <archive>
          <manifest>
            <mainClass>com.pingcap.txn.TxnExample</mainClass>
          </manifest>
        </archive>

      </configuration>
      <executions>
        <execution>
          <id>make-assembly</id>
          <phase>package</phase>
          <goals>
            <goal>single</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
```

```
    </plugins>
  </build>

</project>
```

代码

随后编写代码：

```
package com.pingcap.txn;

import com.zaxxer.hikari.HikariDataSource;

import java.math.BigDecimal;
import java.sql.*;
import java.util.Arrays;
import java.util.concurrent.*;

public class TxnExample {
    public static void main(String[] args) throws SQLException, InterruptedException {
        System.out.println(Arrays.toString(args));
        int aliceQuantity = 0;
        int bobQuantity = 0;

        for (String arg: args) {
            if (arg.startsWith("ALICE_NUM")) {
                aliceQuantity = Integer.parseInt(arg.replace("ALICE_NUM=", ""));
            }

            if (arg.startsWith("BOB_NUM")) {
                bobQuantity = Integer.parseInt(arg.replace("BOB_NUM=", ""));
            }
        }

        HikariDataSource ds = new HikariDataSource();
        ds.setJdbcUrl("jdbc:mysql://localhost:4000/bookshop?useServerPrepStmts=true&
            ↪ cachePrepStmts=true");
        ds.setUsername("root");
        ds.setPassword("");

        // prepare data
        Connection connection = ds.getConnection();
        createBook(connection, 1L, "Designing Data-Intensive Application", "Science & Technology"
            ↪ ,
                Timestamp.valueOf("2018-09-01 00:00:00"), new BigDecimal(100), 10);
        createUser(connection, 1L, "Bob", new BigDecimal(10000));
```

```
        createUser(connection, 2L, "Alice", new BigDecimal(10000));

        CountDownLatch countDownLatch = new CountDownLatch(2);
        ExecutorService threadPool = Executors.newFixedThreadPool(2);

        final int finalBobQuantity = bobQuantity;
        threadPool.execute(() -> {
            buy(ds, 1, 1000L, 1L, 1L, finalBobQuantity);
            countDownLatch.countDown();
        });
        final int finalAliceQuantity = aliceQuantity;
        threadPool.execute(() -> {
            buy(ds, 2, 1001L, 1L, 2L, finalAliceQuantity);
            countDownLatch.countDown();
        });

        countDownLatch.await(5, TimeUnit.SECONDS);
    }

    public static void createUser(Connection connection, Long id, String nickname, BigDecimal
        ↪ balance) throws SQLException  {
        PreparedStatement insert = connection.prepareStatement(
                "INSERT INTO `users` (`id`, `nickname`, `balance`) VALUES (?, ?, ?)");
        insert.setLong(1, id);
        insert.setString(2, nickname);
        insert.setBigDecimal(3, balance);
        insert.executeUpdate();
    }

    public static void createBook(Connection connection, Long id, String title, String type,
        ↪ Timestamp publishedAt, BigDecimal price, Integer stock) throws SQLException {
        PreparedStatement insert = connection.prepareStatement(
                "INSERT INTO `books` (`id`, `title`, `type`, `published_at`, `price`, `stock`)
                    ↪ values (?, ?, ?, ?, ?, ?)");
        insert.setLong(1, id);
        insert.setString(2, title);
        insert.setString(3, type);
        insert.setTimestamp(4, publishedAt);
        insert.setBigDecimal(5, price);
        insert.setInt(6, stock);

        insert.executeUpdate();
    }

    public static void buy (HikariDataSource ds, Integer threadID,
```

```
                    Long orderID, Long bookID, Long userID, Integer quantity) {
String txnComment = "/* txn " + threadID + " */ ";

try (Connection connection = ds.getConnection()) {
    try {
        connection.setAutoCommit(false);
        connection.createStatement().executeUpdate(txnComment + "begin pessimistic");

        // waiting for other thread ran the 'begin pessimistic' statement
        TimeUnit.SECONDS.sleep(1);

        BigDecimal price = null;

        // read price of book
        PreparedStatement selectBook = connection.prepareStatement(txnComment + "select
            ↪ price from books where id = ? for update");
        selectBook.setLong(1, bookID);
        ResultSet res = selectBook.executeQuery();
        if (!res.next()) {
            throw new RuntimeException("book not exist");
        } else {
            price = res.getBigDecimal("price");
        }

        // update book
        String updateBookSQL = "update `books` set stock = stock - ? where id = ? and
            ↪ stock - ? >= 0";
        PreparedStatement updateBook = connection.prepareStatement(txnComment +
            ↪ updateBookSQL);
        updateBook.setInt(1, quantity);
        updateBook.setLong(2, bookID);
        updateBook.setInt(3, quantity);
        int affectedRows = updateBook.executeUpdate();

        if (affectedRows == 0) {
            // stock not enough, rollback
            connection.createStatement().executeUpdate(txnComment + "rollback");
            return;
        }

        // insert order
        String insertOrderSQL = "insert into `orders` (`id`, `book_id`, `user_id`, `
            ↪ quality`) values (?, ?, ?, ?)";
        PreparedStatement insertOrder = connection.prepareStatement(txnComment +
            ↪ insertOrderSQL);
```

```
                insertOrder.setLong(1, orderID);
                insertOrder.setLong(2, bookID);
                insertOrder.setLong(3, userID);
                insertOrder.setInt(4, quantity);
                insertOrder.executeUpdate();

                // update user
                String updateUserSQL = "update `users` set `balance` = `balance` - ? where id = ?
                    ↪ ";
                PreparedStatement updateUser = connection.prepareStatement(txnComment +
                    ↪ updateUserSQL);
                updateUser.setBigDecimal(1, price.multiply(new BigDecimal(quantity)));
                updateUser.setLong(2, userID);
                updateUser.executeUpdate();

                connection.createStatement().executeUpdate(txnComment + "commit");
            } catch (Exception e) {
                connection.createStatement().executeUpdate(txnComment + "rollback");
                e.printStackTrace();
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

首先编写一个封装了所需的数据库操作的 `helper.go` 文件：

```
package main

import (
    "context"
    "database/sql"
    "fmt"
    "time"

    "github.com/go-sql-driver/mysql"
    "github.com/pingcap-inc/tidb-example-golang/util"
    "github.com/shopspring/decimal"
)

type TxnFunc func(txn *util.TiDBSqlTx) error

const (
    ErrWriteConflict      = 9007 // Transactions in TiKV encounter write conflicts.
    ErrInfoSchemaChanged  = 8028 // table schema changes
```

```
    ErrForUpdateCantRetry = 8002 // "SELECT FOR UPDATE" commit conflict
    ErrTxnRetryable       = 8022 // The transaction commit fails and has been rolled back
)

const retryTimes = 5

var retryErrorCodeSet = map[uint16]interface{}{
    ErrWriteConflict:      nil,
    ErrInfoSchemaChanged:  nil,
    ErrForUpdateCantRetry: nil,
    ErrTxnRetryable:       nil,
}

func runTxn(db *sql.DB, optimistic bool, optimisticRetryTimes int, txnFunc TxnFunc) {
    txn, err := util.TiDBSqlBegin(db, !optimistic)
    if err != nil {
        panic(err)
    }

    err = txnFunc(txn)
    if err != nil {
        txn.Rollback()
        if mysqlErr, ok := err.(*mysql.MySQLError); ok && optimistic && optimisticRetryTimes != 0
            ↪ {
            if _, retryableError := retryErrorCodeSet[mysqlErr.Number]; retryableError {
                fmt.Printf("[runTxn] got a retryable error, rest time: %d\n",
                    ↪ optimisticRetryTimes-1)
                runTxn(db, optimistic, optimisticRetryTimes-1, txnFunc)
                return
            }
        }

        fmt.Printf("[runTxn] got an error, rollback: %+v\n", err)
    } else {
        err = txn.Commit()
        if mysqlErr, ok := err.(*mysql.MySQLError); ok && optimistic && optimisticRetryTimes != 0
            ↪ {
            if _, retryableError := retryErrorCodeSet[mysqlErr.Number]; retryableError {
                fmt.Printf("[runTxn] got a retryable error, rest time: %d\n",
                    ↪ optimisticRetryTimes-1)
                runTxn(db, optimistic, optimisticRetryTimes-1, txnFunc)
                return
            }
        }
```

```
        if err == nil {
            fmt.Println("[runTxn] commit success")
        }
    }
}

func prepareData(db *sql.DB, optimistic bool) {
    runTxn(db, optimistic, retryTimes, func(txn *util.TiDBSqlTx) error {
        publishedAt, err := time.Parse("2006-01-02 15:04:05", "2018-09-01 00:00:00")
        if err != nil {
            return err
        }

        if err = createBook(txn, 1, "Designing Data-Intensive Application",
            "Science & Technology", publishedAt, decimal.NewFromInt(100), 10); err != nil {
            return err
        }

        if err = createUser(txn, 1, "Bob", decimal.NewFromInt(10000)); err != nil {
            return err
        }

        if err = createUser(txn, 2, "Alice", decimal.NewFromInt(10000)); err != nil {
            return err
        }

        return nil
    })
}

func buyPessimistic(db *sql.DB, goroutineID, orderID, bookID, userID, amount int) {
    txnComment := fmt.Sprintf("/* txn %d */ ", goroutineID)
    if goroutineID != 1 {
        txnComment = "\t" + txnComment
    }

    fmt.Printf("\nuser %d try to buy %d books(id: %d)\n", userID, amount, bookID)

    runTxn(db, false, retryTimes, func(txn *util.TiDBSqlTx) error {
        time.Sleep(time.Second)

        // read the price of book
        selectBookForUpdate := "select `price` from books where id = ? for update"
        bookRows, err := txn.Query(selectBookForUpdate, bookID)
        if err != nil {
```

```
        return err
    }
    fmt.Println(txnComment + selectBookForUpdate + " successful")
    defer bookRows.Close()

    price := decimal.NewFromInt(0)
    if bookRows.Next() {
        err = bookRows.Scan(&price)
        if err != nil {
            return err
        }
    } else {
        return fmt.Errorf("book ID not exist")
    }
    bookRows.Close()

    // update book
    updateStock := "update `books` set stock = stock - ? where id = ? and stock - ? >= 0"
    result, err := txn.Exec(updateStock, amount, bookID, amount)
    if err != nil {
        return err
    }
    fmt.Println(txnComment + updateStock + " successful")

    affected, err := result.RowsAffected()
    if err != nil {
        return err
    }

    if affected == 0 {
        return fmt.Errorf("stock not enough, rollback")
    }

    // insert order
    insertOrder := "insert into `orders` (`id`, `book_id`, `user_id`, `quality`) values (?,
        ↪ ?, ?, ?)"
    if _, err := txn.Exec(insertOrder,
        orderID, bookID, userID, amount); err != nil {
        return err
    }
    fmt.Println(txnComment + insertOrder + " successful")

    // update user
    updateUser := "update `users` set `balance` = `balance` - ? where id = ?"
    if _, err := txn.Exec(updateUser,
```

```
            price.Mul(decimal.NewFromInt(int64(amount))), userID); err != nil {
            return err
        }
        fmt.Println(txnComment + updateUser + " successful")

        return nil
    })
}

func buyOptimistic(db *sql.DB, goroutineID, orderID, bookID, userID, amount int) {
    txnComment := fmt.Sprintf("/* txn %d */ ", goroutineID)
    if goroutineID != 1 {
        txnComment = "\t" + txnComment
    }

    fmt.Printf("\nuser %d try to buy %d books(id: %d)\n", userID, amount, bookID)

    runTxn(db, true, retryTimes, func(txn *util.TiDBSqlTx) error {
        time.Sleep(time.Second)

        // read the price and stock of book
        selectBookForUpdate := "select `price`, `stock` from books where id = ? for update"
        bookRows, err := txn.Query(selectBookForUpdate, bookID)
        if err != nil {
            return err
        }
        fmt.Println(txnComment + selectBookForUpdate + " successful")
        defer bookRows.Close()

        price, stock := decimal.NewFromInt(0), 0
        if bookRows.Next() {
            err = bookRows.Scan(&price, &stock)
            if err != nil {
                return err
            }
        } else {
            return fmt.Errorf("book ID not exist")
        }
        bookRows.Close()

        if stock < amount {
            return fmt.Errorf("book not enough")
        }

        // update book
```

```
        updateStock := "update `books` set stock = stock - ? where id = ? and stock - ? >= 0"
        result, err := txn.Exec(updateStock, amount, bookID, amount)
        if err != nil {
            return err
        }
        fmt.Println(txnComment + updateStock + " successful")

        affected, err := result.RowsAffected()
        if err != nil {
            return err
        }

        if affected == 0 {
            return fmt.Errorf("stock not enough, rollback")
        }

        // insert order
        insertOrder := "insert into `orders` (`id`, `book_id`, `user_id`, `quality`) values (?,
            ↪ ?, ?, ?)"
        if _, err := txn.Exec(insertOrder,
            orderID, bookID, userID, amount); err != nil {
            return err
        }
        fmt.Println(txnComment + insertOrder + " successful")

        // update user
        updateUser := "update `users` set `balance` = `balance` - ? where id = ?"
        if _, err := txn.Exec(updateUser,
            price.Mul(decimal.NewFromInt(int64(amount))), userID); err != nil {
            return err
        }
        fmt.Println(txnComment + updateUser + " successful")

        return nil
    })
}

func createBook(txn *util.TiDBSqlTx, id int, title, bookType string,
    publishedAt time.Time, price decimal.Decimal, stock int) error {
    _, err := txn.ExecContext(context.Background(),
        "INSERT INTO `books` (`id`, `title`, `type`, `published_at`, `price`, `stock`) values (?,
            ↪  ?, ?, ?, ?, ?)",
        id, title, bookType, publishedAt, price, stock)
    return err
}
```

```
func createUser(txn *util.TiDBSqlTx, id int, nickname string, balance decimal.Decimal) error {
    _, err := txn.ExecContext(context.Background(),
        "INSERT INTO `users` (`id`, `nickname`, `balance`) VALUES (?, ?, ?)",
        id, nickname, balance)
    return err
}
```

再编写一个包含 `main` 函数的 `txn.go` 来调用 `helper.go`，同时处理传入的命令行参数：

```
package main

import (
    "database/sql"
    "flag"
    "fmt"
    "sync"
)

func main() {
    optimistic, alice, bob := parseParams()

    openDB("mysql", "root:@tcp(127.0.0.1:4000)/bookshop?charset=utf8mb4", func(db *sql.DB) {
        prepareData(db, optimistic)
        buy(db, optimistic, alice, bob)
    })
}

func buy(db *sql.DB, optimistic bool, alice, bob int) {
    buyFunc := buyOptimistic
    if !optimistic {
        buyFunc = buyPessimistic
    }

    wg := sync.WaitGroup{}
    wg.Add(1)
    go func() {
        defer wg.Done()
        buyFunc(db, 1, 1000, 1, 1, bob)
    }()

    wg.Add(1)
    go func() {
        defer wg.Done()
        buyFunc(db, 2, 1001, 1, 2, alice)
    }()
```

```
    wg.Wait()
}

func openDB(driverName, dataSourceName string, runnable func(db *sql.DB)) {
    db, err := sql.Open(driverName, dataSourceName)
    if err != nil {
        panic(err)
    }
    defer db.Close()

    runnable(db)
}

func parseParams() (optimistic bool, alice, bob int) {
    flag.BoolVar(&optimistic, "o", false, "transaction is optimistic")
    flag.IntVar(&alice, "a", 4, "Alice bought num")
    flag.IntVar(&bob, "b", 6, "Bob bought num")

    flag.Parse()

    fmt.Println(optimistic, alice, bob)

    return optimistic, alice, bob
}
```

Golang 的例子中，已经包含乐观事务。

```
import time

import MySQLdb
import os
import datetime
from threading import Thread

REPEATABLE_ERROR_CODE_SET = {
    9007,  # Transactions in TiKV encounter write conflicts.
    8028,  # table schema changes
    8002,  # "SELECT FOR UPDATE" commit conflict
    8022   # The transaction commit fails and has been rolled back
}


def create_connection():
    return MySQLdb.connect(
        host="127.0.0.1",
```

```
        port=4000,
        user="root",
        password="",
        database="bookshop",
        autocommit=False
    )


def prepare_data() -> None:
    connection = create_connection()
    with connection:
        with connection.cursor() as cursor:
            cursor.execute("INSERT INTO `books` (`id`, `title`, `type`, `published_at`, `price`,
                ↪ `stock`) "
                           "values (%s, %s, %s, %s, %s, %s)",
                           (1, "Designing Data-Intensive Application", "Science & Technology",
                            datetime.datetime(2018, 9, 1), 100, 10))

            cursor.executemany("INSERT INTO `users` (`id`, `nickname`, `balance`) VALUES (%s, %s,
                ↪  %s)",
                               [(1, "Bob", 10000), (2, "ALICE", 10000)])
            connection.commit()


def buy_optimistic(thread_id: int, order_id: int, book_id: int, user_id: int, amount: int,
                   optimistic_retry_times: int = 5) -> None:
    connection = create_connection()

    txn_log_header = f"/* txn {thread_id} */"
    if thread_id != 1:
        txn_log_header = "\t" + txn_log_header

    with connection:
        with connection.cursor() as cursor:
            cursor.execute("BEGIN OPTIMISTIC")
            print(f'{txn_log_header} BEGIN OPTIMISTIC')
            time.sleep(1)

            try:
                # read the price of book
                select_book_for_update = "SELECT `price`, `stock` FROM books WHERE id = %s FOR
                    ↪ UPDATE"
                cursor.execute(select_book_for_update, (book_id,))
                book = cursor.fetchone()
                if book is None:
```

```
                raise Exception("book_id not exist")
            price, stock = book
            print(f'{txn_log_header} {select_book_for_update} successful')

            if stock < amount:
                raise Exception("book not enough, rollback")

            # update book
            update_stock = "update `books` set stock = stock - %s where id = %s and stock - %
                ↪ s >= 0"
            rows_affected = cursor.execute(update_stock, (amount, book_id, amount))
            print(f'{txn_log_header} {update_stock} successful')

            if rows_affected == 0:
                raise Exception("stock not enough, rollback")

            # insert order
            insert_order = "insert into `orders` (`id`, `book_id`, `user_id`, `quality`)
                ↪ values (%s, %s, %s, %s)"
            cursor.execute(insert_order, (order_id, book_id, user_id, amount))
            print(f'{txn_log_header} {insert_order} successful')

            # update user
            update_user = "update `users` set `balance` = `balance` - %s where id = %s"
            cursor.execute(update_user, (amount * price, user_id))
            print(f'{txn_log_header} {update_user} successful')

        except Exception as err:
            connection.rollback()

            print(f'something went wrong: {err}')
        else:
            # important here! you need deal the Exception from the TiDB
            try:
                connection.commit()
            except MySQLdb.MySQLError as db_err:
                code, desc = db_err.args
                if code in REPEATABLE_ERROR_CODE_SET and optimistic_retry_times > 0:
                    print(f'retry, rest {optimistic_retry_times - 1} times, for {code} {desc}
                        ↪ ')
                    buy_optimistic(thread_id, order_id, book_id, user_id, amount,
                        ↪ optimistic_retry_times - 1)


def buy_pessimistic(thread_id: int, order_id: int, book_id: int, user_id: int, amount: int) ->
```

```
    ↪ None:
    connection = create_connection()

    txn_log_header = f"/* txn {thread_id} */"
    if thread_id != 1:
        txn_log_header = "\t" + txn_log_header

    with connection:
        with connection.cursor() as cursor:
            cursor.execute("BEGIN PESSIMISTIC")
            print(f'{txn_log_header} BEGIN PESSIMISTIC')
            time.sleep(1)

            try:
                # read the price of book
                select_book_for_update = "SELECT `price` FROM books WHERE id = %s FOR UPDATE"
                cursor.execute(select_book_for_update, (book_id,))
                book = cursor.fetchone()
                if book is None:
                    raise Exception("book_id not exist")
                price = book[0]
                print(f'{txn_log_header} {select_book_for_update} successful')

                # update book
                update_stock = "update `books` set stock = stock - %s where id = %s and stock - %
                    ↪ s >= 0"
                rows_affected = cursor.execute(update_stock, (amount, book_id, amount))
                print(f'{txn_log_header} {update_stock} successful')

                if rows_affected == 0:
                    raise Exception("stock not enough, rollback")

                # insert order
                insert_order = "insert into `orders` (`id`, `book_id`, `user_id`, `quality`)
                    ↪ values (%s, %s, %s, %s)"
                cursor.execute(insert_order, (order_id, book_id, user_id, amount))
                print(f'{txn_log_header} {insert_order} successful')

                # update user
                update_user = "update `users` set `balance` = `balance` - %s where id = %s"
                cursor.execute(update_user, (amount * price, user_id))
                print(f'{txn_log_header} {update_user} successful')

            except Exception as err:
                connection.rollback()
```

```
                print(f'something went wrong: {err}')
            else:
                connection.commit()


optimistic = os.environ.get('OPTIMISTIC')
alice = os.environ.get('ALICE')
bob = os.environ.get('BOB')

if not (optimistic and alice and bob):
    raise Exception("please use \"OPTIMISTIC=<is_optimistic> ALICE=<alice_num> "
                    "BOB=<bob_num> python3 txn_example.py\" to start this script")

prepare_data()

if bool(optimistic) is True:
    buy_func = buy_optimistic
else:
    buy_func = buy_pessimistic

bob_thread = Thread(target=buy_func, kwargs={
    "thread_id": 1, "order_id": 1000, "book_id": 1, "user_id": 1, "amount": int(bob)})
alice_thread = Thread(target=buy_func, kwargs={
    "thread_id": 2, "order_id": 1001, "book_id": 1, "user_id": 2, "amount": int(alice)})

bob_thread.start()
alice_thread.start()
bob_thread.join(timeout=10)
alice_thread.join(timeout=10)
```

Python 的例子中，已经包含乐观事务。

4.8.2.1.2 2. 运行不涉及超卖的例子

运行示例程序：

在 Java 中运行示例程序：

```
mvn clean package
java -jar target/plain-java-txn-0.0.1-jar-with-dependencies.jar ALICE_NUM=4 BOB_NUM=6
```

在 Golang 中运行示例程序：

```
go build -o bin/txn
./bin/txn -a 4 -b 6
```

在 Python 中运行示例程序：

```
OPTIMISTIC=False ALICE=4 BOB=6 python3 txn_example.py
```

SQL 日志：

```
/* txn 1 */ BEGIN PESSIMISTIC
    /* txn 2 */ BEGIN PESSIMISTIC
    /* txn 2 */ SELECT * FROM `books` WHERE `id` = 1 FOR UPDATE
    /* txn 2 */ UPDATE `books` SET `stock` = `stock` - 4 WHERE `id` = 1 AND `stock` - 4 >= 0
    /* txn 2 */ INSERT INTO `orders` (`id`, `book_id`, `user_id`, `quality`) VALUES (1001, 1, 1,
        ↪ 4)
    /* txn 2 */ UPDATE `users` SET `balance` = `balance` - 400.0 WHERE `id` = 2
    /* txn 2 */ COMMIT
/* txn 1 */ SELECT * FROM `books` WHERE `id` = 1 FOR UPDATE
/* txn 1 */ UPDATE `books` SET `stock` = `stock` - 6 WHERE `id` = 1 AND `stock` - 6 >= 0
/* txn 1 */ INSERT INTO `orders` (`id`, `book_id`, `user_id`, `quality`) VALUES (1000, 1, 1, 6)
/* txn 1 */ UPDATE `users` SET `balance` = `balance` - 600.0 WHERE `id` = 1
/* txn 1 */ COMMIT
```

最后，检验一下订单创建、用户余额扣减、图书库存扣减情况，都符合预期。

```
mysql> SELECT * FROM `books`;
+--
    ↪ --+--------------------------------------+----------------------+----------------------+-------+--------+
    ↪
| id | title                                | type                 | published_at         | stock
    ↪ | price  |
+--
    ↪ --+--------------------------------------+----------------------+----------------------+-------+--------+
    ↪
|  1 | Designing Data-Intensive Application | Science & Technology | 2018-09-01 00:00:00 |     0
    ↪ | 100.00 |
+--
    ↪ --+--------------------------------------+----------------------+----------------------+-------+--------+
    ↪
1 row in set (0.00 sec)

mysql> SELECT * FROM orders;
+------+---------+---------+---------+---------------------+
| id   | book_id | user_id | quality | ordered_at          |
+------+---------+---------+---------+---------------------+
| 1000 |       1 |       1 |       6 | 2022-04-19 10:58:12 |
| 1001 |       1 |       1 |       4 | 2022-04-19 10:58:11 |
+------+---------+---------+---------+---------------------+
2 rows in set (0.01 sec)

mysql> SELECT * FROM users;
```

```
+----+---------+----------+
| id | balance | nickname |
+----+---------+----------+
|  1 | 9400.00 | Bob      |
|  2 | 9600.00 | Alice    |
+----+---------+----------+
2 rows in set (0.00 sec)
```

4.8.2.1.3 3. 运行防止超卖的例子

可以再把难度加大，如果图书的库存剩余 10 本，Bob 购买 7 本，Alice 购买 4 本，两人几乎同时下单，结果会是怎样？继续复用上个例子里的代码来解决这个需求，只不过把 Bob 购买数量从 6 改成 7：

运行示例程序：

在 Java 中运行示例程序：

```
mvn clean package
java -jar target/plain-java-txn-0.0.1-jar-with-dependencies.jar ALICE_NUM=4 BOB_NUM=7
```

在 Golang 中运行示例程序：

```
go build -o bin/txn
./bin/txn -a 4 -b 7
```

在 Python 中运行示例程序：

```
OPTIMISTIC=False ALICE=4 BOB=7 python3 txn_example.py
```

```
/* txn 1 */ BEGIN PESSIMISTIC
    /* txn 2 */ BEGIN PESSIMISTIC
    /* txn 2 */ SELECT * FROM `books` WHERE `id` = 1 FOR UPDATE
    /* txn 2 */ UPDATE `books` SET `stock` = `stock` - 4 WHERE `id` = 1 AND `stock` - 4 >= 0
    /* txn 2 */ INSERT INTO `orders` (`id`, `book_id`, `user_id`, `quality`) values (1001, 1, 1,
        ↪ 4)
    /* txn 2 */ UPDATE `users` SET `balance` = `balance` - 400.0 WHERE `id` = 2
    /* txn 2 */ COMMIT
/* txn 1 */ SELECT * FROM `books` WHERE `id` = 1 FOR UPDATE
/* txn 1 */ UPDATE `books` SET `stock` = `stock` - 7 WHERE `id` = 1 AND `stock` - 7 >= 0
/* txn 1 */ ROLLBACK
```

由于 txn 2 抢先获得锁资源，更新了 stock，txn 1 里面 affected_rows 返回值为 0，进入了 rollback 流程。

再检验一下订单创建、用户余额扣减、图书库存扣减情况。Alice 下单 4 本书成功，Bob 下单 7 本书失败，库存剩余 6 本符合预期。

```
mysql> SELECT * FROM books;
```

```
+--
    ↪ --+--------------------------------------+---------------------+---------------------+--------+--------+
    ↪
| id | title                                | type                | published_at        | stock
    ↪ | price  |
+--
    ↪ --+--------------------------------------+---------------------+---------------------+--------+--------+
    ↪
|  1 | Designing Data-Intensive Application | Science & Technology | 2018-09-01 00:00:00 |     6
    ↪ | 100.00 |
+--
    ↪ --+--------------------------------------+---------------------+---------------------+--------+--------+
    ↪
1 row in set (0.00 sec)

mysql> SELECT * FROM orders;
+------+---------+---------+---------+---------------------+
| id   | book_id | user_id | quality | ordered_at          |
+------+---------+---------+---------+---------------------+
| 1001 |       1 |       1 |       4 | 2022-04-19 11:03:03 |
+------+---------+---------+---------+---------------------+
1 row in set (0.00 sec)

mysql> SELECT * FROM users;
+----+----------+----------+
| id | balance  | nickname |
+----+----------+----------+
|  1 | 10000.00 | Bob      |
|  2 |  9600.00 | Alice    |
+----+----------+----------+
2 rows in set (0.01 sec)
```

4.8.2.2 乐观事务

下面代码以乐观事务的方式，用两个线程模拟了两个用户并发买同一本书的过程，和悲观事务的示例一样。书店剩余 10 本，Bob 购买了 6 本，Alice 购买了 4 本。两个人几乎同一时间完成订单，最终，这本书的剩余库存为零。

4.8.2.2.1 1. 编写乐观事务示例

使用 Java 编写乐观事务示例：

代码编写

```
package com.pingcap.txn.optimistic;
```

```
import com.zaxxer.hikari.HikariDataSource;

import java.math.BigDecimal;
import java.sql.*;
import java.util.Arrays;
import java.util.concurrent.*;

public class TxnExample {
    public static void main(String[] args) throws SQLException, InterruptedException {
        System.out.println(Arrays.toString(args));
        int aliceQuantity = 0;
        int bobQuantity = 0;

        for (String arg: args) {
            if (arg.startsWith("ALICE_NUM")) {
                aliceQuantity = Integer.parseInt(arg.replace("ALICE_NUM=", ""));
            }

            if (arg.startsWith("BOB_NUM")) {
                bobQuantity = Integer.parseInt(arg.replace("BOB_NUM=", ""));
            }
        }

        HikariDataSource ds = new HikariDataSource();
        ds.setJdbcUrl("jdbc:mysql://localhost:4000/bookshop?useServerPrepStmts=true&
            ↪ cachePrepStmts=true");
        ds.setUsername("root");
        ds.setPassword("");

        // prepare data
        Connection connection = ds.getConnection();
        createBook(connection, 1L, "Designing Data-Intensive Application", "Science & Technology"
            ↪ ,
                Timestamp.valueOf("2018-09-01 00:00:00"), new BigDecimal(100), 10);
        createUser(connection, 1L, "Bob", new BigDecimal(10000));
        createUser(connection, 2L, "Alice", new BigDecimal(10000));

        CountDownLatch countDownLatch = new CountDownLatch(2);
        ExecutorService threadPool = Executors.newFixedThreadPool(2);

        final int finalBobQuantity = bobQuantity;
        threadPool.execute(() -> {
            buy(ds, 1, 1000L, 1L, 1L, finalBobQuantity, 5);
            countDownLatch.countDown();
        });
```

```
        final int finalAliceQuantity = aliceQuantity;
        threadPool.execute(() -> {
            buy(ds, 2, 1001L, 1L, 2L, finalAliceQuantity, 5);
            countDownLatch.countDown();
        });

        countDownLatch.await(5, TimeUnit.SECONDS);
    }

    public static void createUser(Connection connection, Long id, String nickname, BigDecimal
        ↪ balance) throws SQLException  {
        PreparedStatement insert = connection.prepareStatement(
                "INSERT INTO `users` (`id`, `nickname`, `balance`) VALUES (?, ?, ?)");
        insert.setLong(1, id);
        insert.setString(2, nickname);
        insert.setBigDecimal(3, balance);
        insert.executeUpdate();
    }

    public static void createBook(Connection connection, Long id, String title, String type,
        ↪ Timestamp publishedAt, BigDecimal price, Integer stock) throws SQLException {
        PreparedStatement insert = connection.prepareStatement(
                "INSERT INTO `books` (`id`, `title`, `type`, `published_at`, `price`, `stock`)
                    ↪ values (?, ?, ?, ?, ?, ?)");
        insert.setLong(1, id);
        insert.setString(2, title);
        insert.setString(3, type);
        insert.setTimestamp(4, publishedAt);
        insert.setBigDecimal(5, price);
        insert.setInt(6, stock);

        insert.executeUpdate();
    }

    public static void buy (HikariDataSource ds, Integer threadID, Long orderID, Long bookID,
                            Long userID, Integer quantity, Integer retryTimes) {
        String txnComment = "/* txn " + threadID + " */ ";

        try (Connection connection = ds.getConnection()) {
            try {

                connection.setAutoCommit(false);
                connection.createStatement().executeUpdate(txnComment + "begin optimistic");

                // waiting for other thread ran the 'begin optimistic' statement
```

```
TimeUnit.SECONDS.sleep(1);

BigDecimal price = null;

// read price of book
PreparedStatement selectBook = connection.prepareStatement(txnComment + "SELECT *
    ↪  FROM books where id = ? for update");
selectBook.setLong(1, bookID);
ResultSet res = selectBook.executeQuery();
if (!res.next()) {
    throw new RuntimeException("book not exist");
} else {
    price = res.getBigDecimal("price");
    int stock = res.getInt("stock");
    if (stock < quantity) {
        throw new RuntimeException("book not enough");
    }
}

// update book
String updateBookSQL = "update `books` set stock = stock - ? where id = ? and
    ↪ stock - ? >= 0";
PreparedStatement updateBook = connection.prepareStatement(txnComment +
    ↪ updateBookSQL);
updateBook.setInt(1, quantity);
updateBook.setLong(2, bookID);
updateBook.setInt(3, quantity);
updateBook.executeUpdate();

// insert order
String insertOrderSQL = "insert into `orders` (`id`, `book_id`, `user_id`, `
    ↪ quality`) values (?, ?, ?, ?)";
PreparedStatement insertOrder = connection.prepareStatement(txnComment +
    ↪ insertOrderSQL);
insertOrder.setLong(1, orderID);
insertOrder.setLong(2, bookID);
insertOrder.setLong(3, userID);
insertOrder.setInt(4, quantity);
insertOrder.executeUpdate();

// update user
String updateUserSQL = "update `users` set `balance` = `balance` - ? where id = ?
    ↪ ";
PreparedStatement updateUser = connection.prepareStatement(txnComment +
    ↪ updateUserSQL);
```

```
                updateUser.setBigDecimal(1, price.multiply(new BigDecimal(quantity)));
                updateUser.setLong(2, userID);
                updateUser.executeUpdate();

                connection.createStatement().executeUpdate(txnComment + "commit");
            } catch (Exception e) {
                connection.createStatement().executeUpdate(txnComment + "rollback");
                System.out.println("error occurred: " + e.getMessage());

                if (e instanceof SQLException sqlException) {
                    switch (sqlException.getErrorCode()) {
                        // You can get all error codes at https://docs.pingcap.com/tidb/stable/
                            ↪ error-codes
                        case 9007: // Transactions in TiKV encounter write conflicts.
                        case 8028: // table schema changes
                        case 8002: // "SELECT FOR UPDATE" commit conflict
                        case 8022: // The transaction commit fails and has been rolled back
                            if (retryTimes != 0) {
                                System.out.println("rest " + retryTimes + " times. retry for " +
                                    ↪ e.getMessage());
                                buy(ds, threadID, orderID, bookID, userID, quantity, retryTimes -
                                    ↪  1);
                            }
                    }
                }
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

配置更改

此处，需将 `pom.xml` 中启动类：

```
<mainClass>com.pingcap.txn.TxnExample</mainClass>
```

更改为：

```
<mainClass>com.pingcap.txn.optimistic.TxnExample</mainClass>
```

来指向乐观事务的例子。

Golang 在编写悲观事务示例章节中的例子已经支持了乐观事务，无需更改，可直接使用。

Python 在编写悲观事务示例章节中的例子已经支持了乐观事务，无需更改，可直接使用。

4.8.2.2.2 2. 运行不涉及超卖的例子

运行示例程序：

在 Java 中运行示例程序：

```
mvn clean package
java -jar target/plain-java-txn-0.0.1-jar-with-dependencies.jar ALICE_NUM=4 BOB_NUM=6
```

在 Golang 中运行示例程序：

```
go build -o bin/txn
./bin/txn -a 4 -b 6 -o true
```

在 Python 中运行示例程序：

```
OPTIMISTIC=True ALICE=4 BOB=6 python3 txn_example.py
```

SQL 语句执行过程：

```
    /* txn 2 */ BEGIN OPTIMISTIC
/* txn 1 */ BEGIN OPTIMISTIC
    /* txn 2 */ SELECT * FROM `books` WHERE `id` = 1 FOR UPDATE
    /* txn 2 */ UPDATE `books` SET `stock` = `stock` - 4 WHERE `id` = 1 AND `stock` - 4 >= 0
    /* txn 2 */ INSERT INTO `orders` (`id`, `book_id`, `user_id`, `quality`) VALUES (1001, 1, 1,
        ↪ 4)
    /* txn 2 */ UPDATE `users` SET `balance` = `balance` - 400.0 WHERE `id` = 2
    /* txn 2 */ COMMIT
/* txn 1 */ SELECT * FROM `books` WHERE `id` = 1 for UPDATE
/* txn 1 */ UPDATE `books` SET `stock` = `stock` - 6 WHERE `id` = 1 AND `stock` - 6 >= 0
/* txn 1 */ INSERT INTO `orders` (`id`, `book_id`, `user_id`, `quality`) VALUES (1000, 1, 1, 6)
/* txn 1 */ UPDATE `users` SET `balance` = `balance` - 600.0 WHERE `id` = 1
retry 1 times for 9007 Write conflict, txnStartTS=432618733006225412, conflictStartTS
    ↪ =432618733006225411, conflictCommitTS=432618733006225414, key={tableID=126, handle=1}
    ↪ primary={tableID=114, indexID=1, indexValues={1, 1000, }} [try again later]
/* txn 1 */ BEGIN OPTIMISTIC
/* txn 1 */ SELECT * FROM `books` WHERE `id` = 1 FOR UPDATE
/* txn 1 */ UPDATE `books` SET `stock` = `stock` - 6 WHERE `id` = 1 AND `stock` - 6 >= 0
/* txn 1 */ INSERT INTO `orders` (`id`, `book_id`, `user_id`, `quality`) VALUES (1000, 1, 1, 6)
/* txn 1 */ UPDATE `users` SET `balance` = `balance` - 600.0 WHERE `id` = 1
/* txn 1 */ COMMIT
```

在乐观事务模式下，由于中间状态不一定正确，不能像悲观事务模式一样，通过 `affected_rows` 来判断某个语句是否执行成功。需要把事务看做一个整体，通过最终的 COMMIT 语句是否返回异常来判断当前事务是否发生写冲突。

从上面 SQL 日志可以看出，由于两个事务并发执行，并且对同一条记录做了修改，`txn 1` COMMIT 之后抛出了 `9007 Write conflict` 异常。对于乐观事务写冲突，在应用端可以进行安全的重试，重试一次之后提交成功，最终执行结果符合预期：

```
mysql> SELECT * FROM books;
+--
    ↪ --+-------------------------------------+---------------------+---------------------+-------+--------+
    ↪
| id | title                                | type                | published_at        | stock
    ↪ | price  |
+--
    ↪ --+-------------------------------------+---------------------+---------------------+-------+--------+
    ↪
|  1 | Designing Data-Intensive Application | Science & Technology | 2018-09-01 00:00:00 |     0
    ↪ | 100.00 |
+--
    ↪ --+-------------------------------------+---------------------+---------------------+-------+--------+
    ↪
1 row in set (0.01 sec)

mysql> SELECT * FROM orders;
+------+---------+---------+---------+---------------------+
| id   | book_id | user_id | quality | ordered_at          |
+------+---------+---------+---------+---------------------+
| 1000 |       1 |       1 |       6 | 2022-04-19 03:18:19 |
| 1001 |       1 |       1 |       4 | 2022-04-19 03:18:17 |
+------+---------+---------+---------+---------------------+
2 rows in set (0.01 sec)

mysql> SELECT * FROM users;
+----+---------+----------+
| id | balance | nickname |
+----+---------+----------+
|  1 | 9400.00 | Bob      |
|  2 | 9600.00 | Alice    |
+----+---------+----------+
2 rows in set (0.00 sec)
```

4.8.2.2.3 3. 运行防止超卖的例子

再来看一下用乐观事务防止超卖的例子，如果图书的库存剩余 10 本，Bob 购买 7 本，Alice 购买 4 本，两人几乎同时下单，结果会是怎样？继续复用乐观事务例子里的代码来解决这个需求，只不过把 Bob 购买数量从 6 改成 7：

运行示例程序：

在 Java 中运行示例程序：

```
mvn clean package
java -jar target/plain-java-txn-0.0.1-jar-with-dependencies.jar ALICE_NUM=4 BOB_NUM=7
```

在 Golang 中运行示例程序：

```
go build -o bin/txn
./bin/txn -a 4 -b 7 -o true
```

在 Python 中运行示例程序：

```
OPTIMISTIC=True ALICE=4 BOB=7 python3 txn_example.py
```

```
/* txn 1 */ BEGIN OPTIMISTIC
    /* txn 2 */ BEGIN OPTIMISTIC
    /* txn 2 */ SELECT * FROM `books` WHERE `id` = 1 FOR UPDATE
    /* txn 2 */ UPDATE `books` SET `stock` = `stock` - 4 WHERE `id` = 1 AND `stock` - 4 >= 0
    /* txn 2 */ INSERT INTO `orders` (`id`, `book_id`, `user_id`, `quality`) VALUES (1001, 1, 1,
        ↪ 4)
    /* txn 2 */ UPDATE `users` SET `balance` = `balance` - 400.0 WHERE `id` = 2
    /* txn 2 */ COMMIT
/* txn 1 */ SELECT * FROM `books` WHERE `id` = 1 FOR UPDATE
/* txn 1 */ UPDATE `books` SET `stock` = `stock` - 7 WHERE `id` = 1 AND `stock` - 7 >= 0
/* txn 1 */ INSERT INTO `orders` (`id`, `book_id`, `user_id`, `quality`) VALUES (1000, 1, 1, 7)
/* txn 1 */ UPDATE `users` SET `balance` = `balance` - 700.0 WHERE `id` = 1
retry 1 times for 9007 Write conflict, txnStartTS=432619094333980675, conflictStartTS
    ↪ =432619094333980676, conflictCommitTS=432619094333980678, key={tableID=126, handle=1}
    ↪ primary={tableID=114, indexID=1, indexValues={1, 1000, }} [try again later]
/* txn 1 */ BEGIN OPTIMISTIC
/* txn 1 */ SELECT * FROM `books` WHERE `id` = 1 FOR UPDATE
Fail -> out of stock
/* txn 1 */ ROLLBACK
```

从上面的 SQL 日志可以看出，第一次执行由于写冲突，`txn 1` 在应用端进行了重试，从获取到的最新快照对比发现，剩余库存不够，应用端抛出 `out of stock` 异常结束。

```
mysql> SELECT * FROM books;
+--
    ↪ --+--------------------------------------+---------------------+---------------------+-------+--------+
    ↪
| id | title                                | type                | published_at        | stock
    ↪ | price  |
+--
    ↪ --+--------------------------------------+---------------------+---------------------+-------+--------+
    ↪
|  1 | Designing Data-Intensive Application | Science & Technology | 2018-09-01 00:00:00 |     6
    ↪ | 100.00 |
+--
    ↪ --+--------------------------------------+---------------------+---------------------+-------+--------+
    ↪
1 row in set (0.00 sec)
```

```
mysql> SELECT * FROM orders;
+------+---------+---------+---------+---------------------+
| id   | book_id | user_id | quality | ordered_at          |
+------+---------+---------+---------+---------------------+
| 1001 |       1 |       1 |       4 | 2022-04-19 03:41:16 |
+------+---------+---------+---------+---------------------+
1 row in set (0.00 sec)

mysql> SELECT * FROM users;
+----+----------+----------+
| id | balance  | nickname |
+----+----------+----------+
|  1 | 10000.00 | Bob      |
|  2 |  9600.00 | Alice    |
+----+----------+----------+
2 rows in set (0.00 sec)
```

4.8.3 事务限制

本章将简单介绍 TiDB 中的事务限制。

4.8.3.1 隔离级别

TiDB 支持的隔离级别是 RC（Read Committed）与 SI（Snapshot Isolation），其中 SI 与 RR（Repeatable Read）隔离级别基本等价。

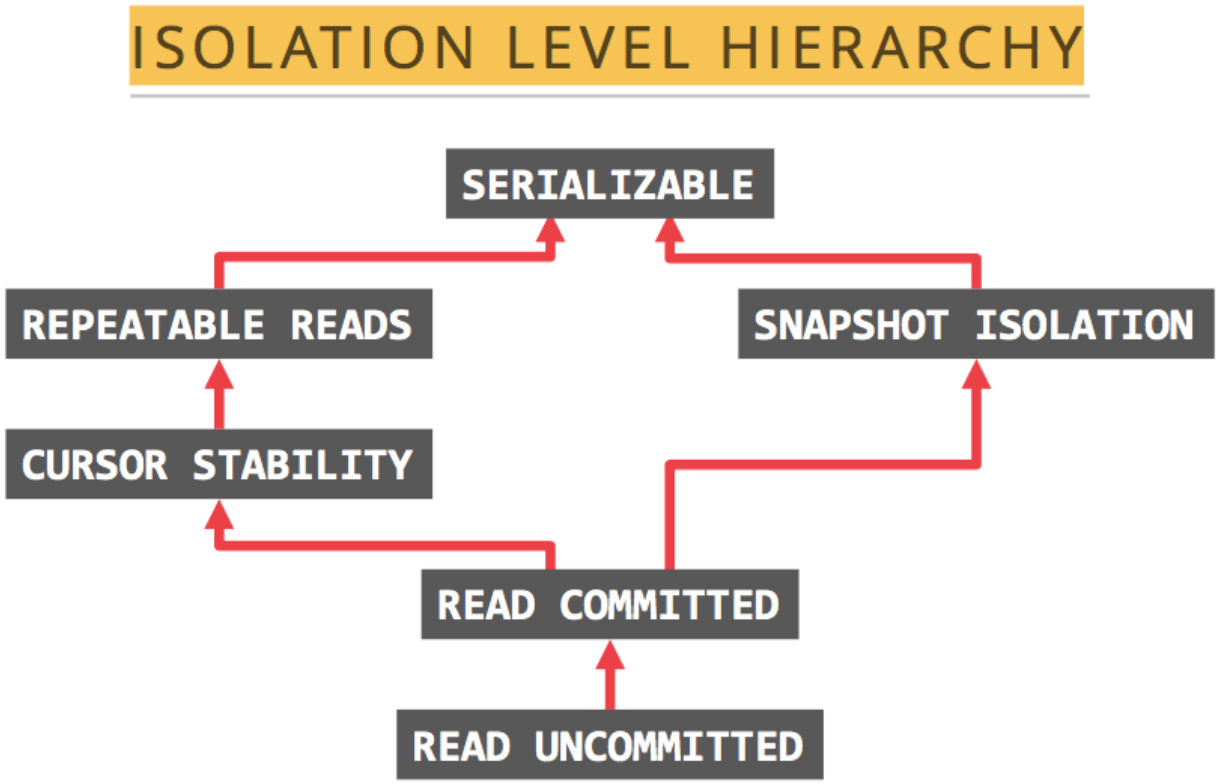


图 15: 隔离级别

4.8.3.2 SI 可以克服幻读

TiDB 的 SI 隔离级别可以克服幻读异常 (Phantom Reads)，但 ANSI/ISO SQL 标准中的 RR 不能。

所谓幻读是指：事务 A 首先根据条件查询得到 n 条记录，然后事务 B 改变了这 n 条记录之外的 m 条记录或者增添了 m 条符合事务 A 查询条件的记录，导致事务 A 再次发起请求时发现有 n+m 条符合条件记录，就产生了幻读。

例如：系统管理员 A 将数据库中所有学生的成绩从具体分数改为 ABCDE 等级，但是系统管理员 B 就在这个时候插入了一条具体分数的记录，当系统管理员 A 改结束后发现还有一条记录没有改过来，就好像发生了幻觉一样，这就叫幻读。

4.8.3.3 SI 不能克服写偏斜

TiDB 的 SI 隔离级别不能克服写偏斜异常（ Write Skew ），需要使用 Select for update 语法来克服写偏斜异常。

写偏斜异常是指两个并发的事务读取了不同但相关的记录，接着这两个事务各自更新了自己读到的数据，并最终都提交了事务，如果这些相关的记录之间存在着不能被多个事务并发修改的约束，那么最终结果将是违反约束的。

举个例子，假设你正在为医院写一个医生轮班管理程序。医院通常会同时要求几位医生待命，但底线是至少有一位医生在待命。医生可以放弃他们的班次（ 例如，如果他们自己生病了 ），只要至少有一个同事在这一班中继续工作。

现在出现这样一种情况，Alice 和 Bob 是两位值班医生。两人都感到不适，所以他们都决定请假。不幸的是，他们恰好在同一时间点击按钮下班。下面用程序来模拟一下这个过程。

Java 程序示例如下：

```
package com.pingcap.txn.write.skew;

import com.zaxxer.hikari.HikariDataSource;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;

public class EffectWriteSkew {
    public static void main(String[] args) throws SQLException, InterruptedException {
        HikariDataSource ds = new HikariDataSource();
        ds.setJdbcUrl("jdbc:mysql://localhost:4000/test?useServerPrepStmts=true&cachePrepStmts=
            ↪ true");
        ds.setUsername("root");

        // prepare data
        Connection connection = ds.getConnection();
        createDoctorTable(connection);
        createDoctor(connection, 1, "Alice", true, 123);
        createDoctor(connection, 2, "Bob", true, 123);
        createDoctor(connection, 3, "Carol", false, 123);

        Semaphore txn1Pass = new Semaphore(0);
        CountDownLatch countDownLatch = new CountDownLatch(2);
        ExecutorService threadPool = Executors.newFixedThreadPool(2);

        threadPool.execute(() -> {
            askForLeave(ds, txn1Pass, 1, 1);
            countDownLatch.countDown();
        });

        threadPool.execute(() -> {
            askForLeave(ds, txn1Pass, 2, 2);
            countDownLatch.countDown();
        });
```

```
        countDownLatch.await();
    }

    public static void createDoctorTable(Connection connection) throws SQLException {
        connection.createStatement().executeUpdate("CREATE TABLE `doctors` (" +
                "    `id` int(11) NOT NULL," +
                "    `name` varchar(255) DEFAULT NULL," +
                "    `on_call` tinyint(1) DEFAULT NULL," +
                "    `shift_id` int(11) DEFAULT NULL," +
                "    PRIMARY KEY (`id`)," +
                "    KEY `idx_shift_id` (`shift_id`)" +
                "  ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin");
    }

    public static void createDoctor(Connection connection, Integer id, String name, Boolean
        ↪ onCall, Integer shiftID) throws SQLException {
        PreparedStatement insert = connection.prepareStatement(
                "INSERT INTO `doctors` (`id`, `name`, `on_call`, `shift_id`) VALUES (?, ?, ?, ?)"
                    ↪ );
        insert.setInt(1, id);
        insert.setString(2, name);
        insert.setBoolean(3, onCall);
        insert.setInt(4, shiftID);
        insert.executeUpdate();
    }

    public static void askForLeave(HikariDataSource ds, Semaphore txn1Pass, Integer txnID,
        ↪ Integer doctorID) {
        try(Connection connection = ds.getConnection()) {
            try {
                connection.setAutoCommit(false);

                String comment = txnID == 2 ? "    " : "" + "/* txn #{txn_id} */ ";
                connection.createStatement().executeUpdate(comment + "BEGIN");

                // Txn 1 should be waiting until txn 2 is done.
                if (txnID == 1) {
                    txn1Pass.acquire();
                }

                PreparedStatement currentOnCallQuery = connection.prepareStatement(comment +
                        "SELECT COUNT(*) AS `count` FROM `doctors` WHERE `on_call` = ? AND `
                            ↪ shift_id` = ?");
                currentOnCallQuery.setBoolean(1, true);
                currentOnCallQuery.setInt(2, 123);
```

```
                ResultSet res = currentOnCallQuery.executeQuery();

                if (!res.next()) {
                    throw new RuntimeException("error query");
                } else {
                    int count = res.getInt("count");
                    if (count >= 2) {
                        // If current on-call doctor has 2 or more, this doctor can leave
                        PreparedStatement insert = connection.prepareStatement( comment +
                                "UPDATE `doctors` SET `on_call` = ? WHERE `id` = ? AND `shift_id`
                                    ↪  = ?");
                        insert.setBoolean(1, false);
                        insert.setInt(2, doctorID);
                        insert.setInt(3, 123);
                        insert.executeUpdate();

                        connection.commit();
                    } else {
                        throw new RuntimeException("At least one doctor is on call");
                    }
                }

                // Txn 2 is done. Let txn 1 run again.
                if (txnID == 2) {
                    txn1Pass.release();
                }
            } catch (Exception e) {
                // If got any error, you should roll back, data is priceless
                connection.rollback();
                e.printStackTrace();
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

在 Golang 中，首先，封装一个用于适配 TiDB 事务的工具包 util，随后编写以下代码：

```
package main

import (
    "database/sql"
    "fmt"
    "sync"
```

```
    "github.com/pingcap-inc/tidb-example-golang/util"

    _ "github.com/go-sql-driver/mysql"
)

func main() {
    openDB("mysql", "root:@tcp(127.0.0.1:4000)/test", func(db *sql.DB) {
        writeSkew(db)
    })
}

func openDB(driverName, dataSourceName string, runnable func(db *sql.DB)) {
    db, err := sql.Open(driverName, dataSourceName)
    if err != nil {
        panic(err)
    }
    defer db.Close()

    runnable(db)
}

func writeSkew(db *sql.DB) {
    err := prepareData(db)
    if err != nil {
        panic(err)
    }

    waitingChan, waitGroup := make(chan bool), sync.WaitGroup{}

    waitGroup.Add(1)
    go func() {
        defer waitGroup.Done()
        err = askForLeave(db, waitingChan, 1, 1)
        if err != nil {
            panic(err)
        }
    }()

    waitGroup.Add(1)
    go func() {
        defer waitGroup.Done()
        err = askForLeave(db, waitingChan, 2, 2)
        if err != nil {
            panic(err)
        }
```

```
    }()

    waitGroup.Wait()
}

func askForLeave(db *sql.DB, waitingChan chan bool, goroutineID, doctorID int) error {
    txnComment := fmt.Sprintf("/* txn %d */ ", goroutineID)
    if goroutineID != 1 {
        txnComment = "\t" + txnComment
    }

    txn, err := util.TiDBSqlBegin(db, true)
    if err != nil {
        return err
    }
    fmt.Println(txnComment + "start txn")

    // Txn 1 should be waiting until txn 2 is done.
    if goroutineID == 1 {
        <-waitingChan
    }

    txnFunc := func() error {
        queryCurrentOnCall := "SELECT COUNT(*) AS `count` FROM `doctors` WHERE `on_call` = ? AND
            ↪ `shift_id` = ?"
        rows, err := txn.Query(queryCurrentOnCall, true, 123)
        if err != nil {
            return err
        }
        defer rows.Close()
        fmt.Println(txnComment + queryCurrentOnCall + " successful")

        count := 0
        if rows.Next() {
            err = rows.Scan(&count)
            if err != nil {
                return err
            }
        }
        rows.Close()

        if count < 2 {
            return fmt.Errorf("at least one doctor is on call")
        }
```

```
        shift := "UPDATE `doctors` SET `on_call` = ? WHERE `id` = ? AND `shift_id` = ?"
        _, err = txn.Exec(shift, false, doctorID, 123)
        if err == nil {
            fmt.Println(txnComment + shift + " successful")
        }
        return err
    }

    err = txnFunc()
    if err == nil {
        txn.Commit()
        fmt.Println("[runTxn] commit success")
    } else {
        txn.Rollback()
        fmt.Printf("[runTxn] got an error, rollback: %+v\n", err)
    }

    // Txn 2 is done. Let txn 1 run again.
    if goroutineID == 2 {
        waitingChan <- true
    }

    return nil
}

func prepareData(db *sql.DB) error {
    err := createDoctorTable(db)
    if err != nil {
        return err
    }

    err = createDoctor(db, 1, "Alice", true, 123)
    if err != nil {
        return err
    }
    err = createDoctor(db, 2, "Bob", true, 123)
    if err != nil {
        return err
    }
    err = createDoctor(db, 3, "Carol", false, 123)
    if err != nil {
        return err
    }
    return nil
}
```

```
func createDoctorTable(db *sql.DB) error {
    _, err := db.Exec("CREATE TABLE IF NOT EXISTS `doctors` (" +
        "    `id` int(11) NOT NULL," +
        "    `name` varchar(255) DEFAULT NULL," +
        "    `on_call` tinyint(1) DEFAULT NULL," +
        "    `shift_id` int(11) DEFAULT NULL," +
        "    PRIMARY KEY (`id`)," +
        "    KEY `idx_shift_id` (`shift_id`)" +
        "  ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin")
    return err
}

func createDoctor(db *sql.DB, id int, name string, onCall bool, shiftID int) error {
    _, err := db.Exec("INSERT INTO `doctors` (`id`, `name`, `on_call`, `shift_id`) VALUES (?, ?,
        ↪ ?, ?)",
        id, name, onCall, shiftID)
    return err
}
```

SQL 日志：

```
/* txn 1 */ BEGIN
    /* txn 2 */ BEGIN
    /* txn 2 */ SELECT COUNT(*) as `count` FROM `doctors` WHERE `on_call` = 1 AND `shift_id` =
        ↪ 123
    /* txn 2 */ UPDATE `doctors` SET `on_call` = 0 WHERE `id` = 2 AND `shift_id` = 123
    /* txn 2 */ COMMIT
/* txn 1 */ SELECT COUNT(*) AS `count` FROM `doctors` WHERE `on_call` = 1 and `shift_id` = 123
/* txn 1 */ UPDATE `doctors` SET `on_call` = 0 WHERE `id` = 1 AND `shift_id` = 123
/* txn 1 */ COMMIT
```

执行结果：

```
mysql> SELECT * FROM doctors;
+----+-------+---------+----------+
| id | name  | on_call | shift_id |
+----+-------+---------+----------+
|  1 | Alice |       0 |      123 |
|  2 | Bob   |       0 |      123 |
|  3 | Carol |       0 |      123 |
+----+-------+---------+----------+
```

在两个事务中，应用首先检查是否有两个或以上的医生正在值班；如果是的话，它就假定一名医生可以安全地休班。由于数据库使用快照隔离，两次检查都返回 2，所以两个事务都进入下一个阶段。Alice 更新自己的记录休班了，而 Bob 也做了一样的事情。两个事务都成功提交了，现在没有医生值班了。违反了至少有一名

医生在值班的要求。下图 (引用自《Designing Data-Intensive Application》) 说明了实际发生的情况：

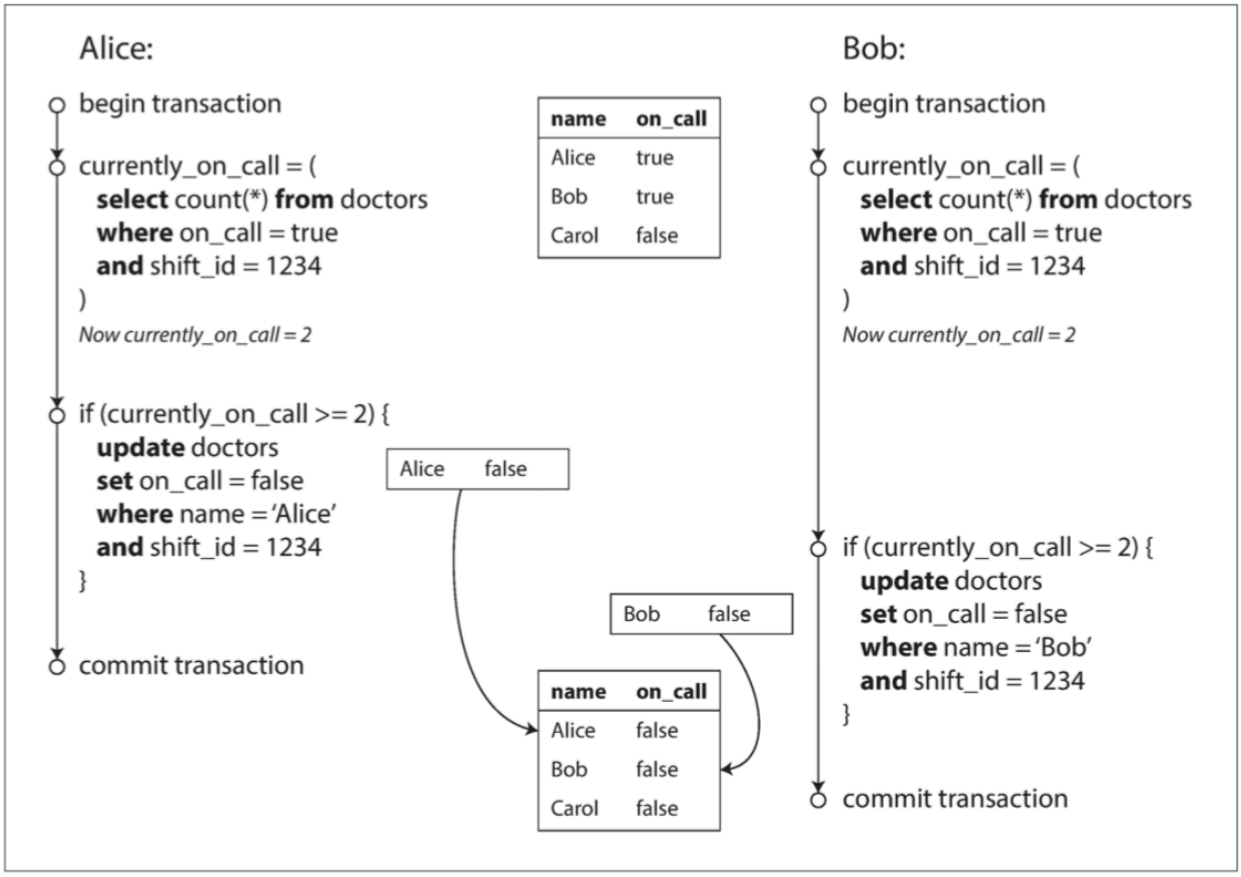


图 16: Write Skew

现在更改示例程序，使用 `SELECT FOR UPDATE` 来克服写偏斜问题：

Java 中使用 `SELECT FOR UPDATE` 来克服写偏斜问题的示例如下：

```
package com.pingcap.txn.write.skew;

import com.zaxxer.hikari.HikariDataSource;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Semaphore;

public class EffectWriteSkew {
```

```
public static void main(String[] args) throws SQLException, InterruptedException {
    HikariDataSource ds = new HikariDataSource();
    ds.setJdbcUrl("jdbc:mysql://localhost:4000/test?useServerPrepStmts=true&cachePrepStmts=
        ↪ true");
    ds.setUsername("root");

    // prepare data
    Connection connection = ds.getConnection();
    createDoctorTable(connection);
    createDoctor(connection, 1, "Alice", true, 123);
    createDoctor(connection, 2, "Bob", true, 123);
    createDoctor(connection, 3, "Carol", false, 123);

    Semaphore txn1Pass = new Semaphore(0);
    CountDownLatch countDownLatch = new CountDownLatch(2);
    ExecutorService threadPool = Executors.newFixedThreadPool(2);

    threadPool.execute(() -> {
        askForLeave(ds, txn1Pass, 1, 1);
        countDownLatch.countDown();
    });

    threadPool.execute(() -> {
        askForLeave(ds, txn1Pass, 2, 2);
        countDownLatch.countDown();
    });

    countDownLatch.await();
}

public static void createDoctorTable(Connection connection) throws SQLException {
    connection.createStatement().executeUpdate("CREATE TABLE `doctors` (" +
            "    `id` int(11) NOT NULL," +
            "    `name` varchar(255) DEFAULT NULL," +
            "    `on_call` tinyint(1) DEFAULT NULL," +
            "    `shift_id` int(11) DEFAULT NULL," +
            "    PRIMARY KEY (`id`)," +
            "    KEY `idx_shift_id` (`shift_id`)" +
            "  ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin");
}

public static void createDoctor(Connection connection, Integer id, String name, Boolean
    ↪ onCall, Integer shiftID) throws SQLException {
    PreparedStatement insert = connection.prepareStatement(
            "INSERT INTO `doctors` (`id`, `name`, `on_call`, `shift_id`) VALUES (?, ?, ?, ?)"
```

```
                ↪ );
        insert.setInt(1, id);
        insert.setString(2, name);
        insert.setBoolean(3, onCall);
        insert.setInt(4, shiftID);
        insert.executeUpdate();
    }

    public static void askForLeave(HikariDataSource ds, Semaphore txn1Pass, Integer txnID,
        ↪ Integer doctorID) {
        try(Connection connection = ds.getConnection()) {
            try {
                connection.setAutoCommit(false);

                String comment = txnID == 2 ? "    " : "" + "/* txn #{txn_id} */ ";
                connection.createStatement().executeUpdate(comment + "BEGIN");

                // Txn 1 should be waiting until txn 2 is done.
                if (txnID == 1) {
                    txn1Pass.acquire();
                }

                PreparedStatement currentOnCallQuery = connection.prepareStatement(comment +
                        "SELECT COUNT(*) AS `count` FROM `doctors` WHERE `on_call` = ? AND `
                            ↪ shift_id` = ? FOR UPDATE");
                currentOnCallQuery.setBoolean(1, true);
                currentOnCallQuery.setInt(2, 123);
                ResultSet res = currentOnCallQuery.executeQuery();

                if (!res.next()) {
                    throw new RuntimeException("error query");
                } else {
                    int count = res.getInt("count");
                    if (count >= 2) {
                        // If current on-call doctor has 2 or more, this doctor can leave
                        PreparedStatement insert = connection.prepareStatement( comment +
                                "UPDATE `doctors` SET `on_call` = ? WHERE `id` = ? AND `shift_id`
                                    ↪  = ?");
                        insert.setBoolean(1, false);
                        insert.setInt(2, doctorID);
                        insert.setInt(3, 123);
                        insert.executeUpdate();

                        connection.commit();
                    } else {
```

```
                        throw new RuntimeException("At least one doctor is on call");
                    }
                }

                // Txn 2 is done. Let txn 1 run again.
                if (txnID == 2) {
                    txn1Pass.release();
                }
            } catch (Exception e) {
                // If got any error, you should roll back, data is priceless
                connection.rollback();
                e.printStackTrace();
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Golang 中使用 `SELECT FOR UPDATE` 来克服写偏斜问题的示例如下：

```
package main

import (
    "database/sql"
    "fmt"
    "sync"

    "github.com/pingcap-inc/tidb-example-golang/util"

    _ "github.com/go-sql-driver/mysql"
)

func main() {
    openDB("mysql", "root:@tcp(127.0.0.1:4000)/test", func(db *sql.DB) {
        writeSkew(db)
    })
}

func openDB(driverName, dataSourceName string, runnable func(db *sql.DB)) {
    db, err := sql.Open(driverName, dataSourceName)
    if err != nil {
        panic(err)
    }
    defer db.Close()
```

```
    runnable(db)
}

func writeSkew(db *sql.DB) {
    err := prepareData(db)
    if err != nil {
        panic(err)
    }

    waitingChan, waitGroup := make(chan bool), sync.WaitGroup{}

    waitGroup.Add(1)
    go func() {
        defer waitGroup.Done()
        err = askForLeave(db, waitingChan, 1, 1)
        if err != nil {
            panic(err)
        }
    }()

    waitGroup.Add(1)
    go func() {
        defer waitGroup.Done()
        err = askForLeave(db, waitingChan, 2, 2)
        if err != nil {
            panic(err)
        }
    }()

    waitGroup.Wait()
}

func askForLeave(db *sql.DB, waitingChan chan bool, goroutineID, doctorID int) error {
    txnComment := fmt.Sprintf("/* txn %d */ ", goroutineID)
    if goroutineID != 1 {
        txnComment = "\t" + txnComment
    }

    txn, err := util.TiDBSqlBegin(db, true)
    if err != nil {
        return err
    }
    fmt.Println(txnComment + "start txn")

    // Txn 1 should be waiting until txn 2 is done.
```

```
if goroutineID == 1 {
    <-waitingChan
}

txnFunc := func() error {
    queryCurrentOnCall := "SELECT COUNT(*) AS `count` FROM `doctors` WHERE `on_call` = ? AND
        ↪ `shift_id` = ?"
    rows, err := txn.Query(queryCurrentOnCall, true, 123)
    if err != nil {
        return err
    }
    defer rows.Close()
    fmt.Println(txnComment + queryCurrentOnCall + " successful")

    count := 0
    if rows.Next() {
        err = rows.Scan(&count)
        if err != nil {
            return err
        }
    }
    rows.Close()

    if count < 2 {
        return fmt.Errorf("at least one doctor is on call")
    }

    shift := "UPDATE `doctors` SET `on_call` = ? WHERE `id` = ? AND `shift_id` = ?"
    _, err = txn.Exec(shift, false, doctorID, 123)
    if err == nil {
        fmt.Println(txnComment + shift + " successful")
    }
    return err
}

err = txnFunc()
if err == nil {
    txn.Commit()
    fmt.Println("[runTxn] commit success")
} else {
    txn.Rollback()
    fmt.Printf("[runTxn] got an error, rollback: %+v\n", err)
}

// Txn 2 is done. Let txn 1 run again.
```

```
    if goroutineID == 2 {
        waitingChan <- true
    }

    return nil
}

func prepareData(db *sql.DB) error {
    err := createDoctorTable(db)
    if err != nil {
        return err
    }

    err = createDoctor(db, 1, "Alice", true, 123)
    if err != nil {
        return err
    }
    err = createDoctor(db, 2, "Bob", true, 123)
    if err != nil {
        return err
    }
    err = createDoctor(db, 3, "Carol", false, 123)
    if err != nil {
        return err
    }
    return nil
}

func createDoctorTable(db *sql.DB) error {
    _, err := db.Exec("CREATE TABLE IF NOT EXISTS `doctors` (" +
        "    `id` int(11) NOT NULL," +
        "    `name` varchar(255) DEFAULT NULL," +
        "    `on_call` tinyint(1) DEFAULT NULL," +
        "    `shift_id` int(11) DEFAULT NULL," +
        "    PRIMARY KEY (`id`)," +
        "    KEY `idx_shift_id` (`shift_id`)" +
        "  ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin")
    return err
}

func createDoctor(db *sql.DB, id int, name string, onCall bool, shiftID int) error {
    _, err := db.Exec("INSERT INTO `doctors` (`id`, `name`, `on_call`, `shift_id`) VALUES (?, ?,
        ↪ ?, ?)",
        id, name, onCall, shiftID)
    return err
```

```
}
```

SQL 日志：

```
/* txn 1 */ BEGIN
    /* txn 2 */ BEGIN
    /* txn 2 */ SELECT COUNT(*) AS `count` FROM `doctors` WHERE on_call = 1 AND `shift_id` = 123
        ↪ FOR UPDATE
    /* txn 2 */ UPDATE `doctors` SET on_call = 0 WHERE `id` = 2 AND `shift_id` = 123
    /* txn 2 */ COMMIT
/* txn 1 */ SELECT COUNT(*) AS `count` FROM `doctors` WHERE `on_call` = 1 FOR UPDATE
At least one doctor is on call
/* txn 1 */ ROLLBACK
```

执行结果：

```
mysql> SELECT * FROM doctors;
+----+-------+---------+----------+
| id | name  | on_call | shift_id |
+----+-------+---------+----------+
|  1 | Alice |       1 |      123 |
|  2 | Bob   |       0 |      123 |
|  3 | Carol |       0 |      123 |
+----+-------+---------+----------+
```

4.8.3.4 对 savepoint 和嵌套事务的支持

Spring 支持的 PROPAGATION_NESTED 传播行为会启动一个嵌套的事务，它是当前事务之上独立启动的一个子事务。嵌套事务开始时会记录一个 savepoint，如果嵌套事务执行失败，事务将会回滚到 savepoint 的状态。嵌套事务是外层事务的一部分，它将会在外层事务提交时一起被提交。下面案例展示了 savepoint 机制：

```
mysql> BEGIN;
mysql> INSERT INTO T2 VALUES(100);
mysql> SAVEPOINT svp1;
mysql> INSERT INTO T2 VALUES(200);
mysql> ROLLBACK TO SAVEPOINT svp1;
mysql> RELEASE SAVEPOINT svp1;
mysql> COMMIT;
mysql> SELECT * FROM T2;
+------+
|  ID  |
+------+
|  100 |
+------+
```

> **注意：**
>
> TiDB 从 v6.2.0 版本开始支持savepoint 特性。因此低于 v6.2.0 版本的 TiDB 不支持 PROPAGATION_NESTED 传播行为。基于 Java Spring 框架的应用如果使用了 PROPAGATION_NESTED 传播行为，需要在应用端做出调整，将嵌套事务的逻辑移除。

4.8.3.5 大事务限制

基本原则是要限制事务的大小。TiDB 对单个事务的大小有限制，这层限制是在 KV 层面。反映在 SQL 层面的话，简单来说一行数据会映射为一个 KV entry，每多一个索引，也会增加一个 KV entry。所以这个限制反映在 SQL 层面是：

- 最大单行记录容量为 120MB（TiDB v5.0 及更高的版本可通过 tidb-server 配置项 `performance.txn-entry-size-limit` 调整，低于 TiDB v5.0 的版本支持的单行容量为 6MB）。
- 支持的最大单个事务容量为 10GB（TiDB v4.0 及更高版本可通过 tidb-server 配置项 `performance.txn-total-size-limit` 调整，低于 TiDB v4.0 的版本支持的最大单个事务容量为 100MB）。

另外注意，无论是大小限制还是行数限制，还要考虑事务执行过程中，TiDB 做编码以及事务额外 Key 的开销。在使用的时候，为了使性能达到最优，建议每 100 ~ 500 行写入一个事务。

4.8.3.6 自动提交的 SELECT FOR UPDATE 语句不会等锁

自动提交下的 SELECT FOR UPDATE 目前不会加锁。效果如下图所示：

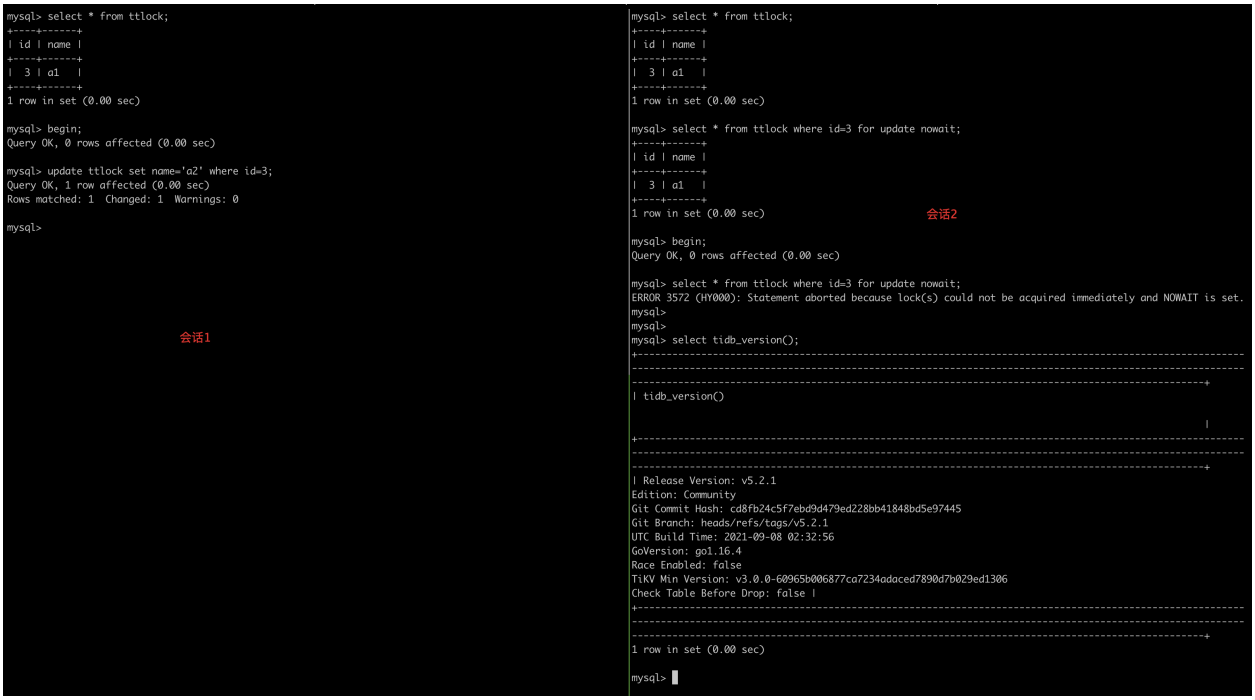


图 17: TiDB 中的情况

这是已知的与 MySQL 不兼容的地方。

可以通过使用显式的 `BEGIN;COMMIT;` 解决该问题。

4.8.4 事务错误处理

本章介绍使用事务时可能会遇到的错误和处理办法。

4.8.4.1 死锁

如果应用程序遇到下面错误时，说明遇到了死锁问题：

```
ERROR 1213: Deadlock found when trying to get lock; try restarting transaction
```

当两个及以上的事务，双方都在等待对方释放已经持有的锁或因为加锁顺序不一致，造成循环等待锁资源，就会出现“死锁”。这里以bookshop 数据库中的 books 表为示例演示死锁：

先给 books 表中写入 2 条数据：

```
INSERT INTO books (id, title, stock, published_at) VALUES (1, 'book-1', 10, now()), (2, 'book-2',
    ↪  10, now());
```

在 TiDB 悲观事务模式下，用 2 个客户端分别执行以下语句，就会遇到死锁：

客户端 -A	客户端 -B
BEGIN;	
	BEGIN;
UPDATE books SET stock=stock-1 WHERE id=1;	
	UPDATE books SET stock=stock-1 WHERE id=2;
UPDATE books SET stock=stock-1 WHERE id=2; – 执行会被阻塞	
	UPDATE books SET stock=stock-1 WHERE id=1; – 遇到 Deadlock 错误

在客户端 -B 遇到死锁错误后，TiDB 会自动 ROLLBACK 客户端 -B 中的事务，然后客户端 -A 中购买 `id=2` 的操作就会执行成功，再执行 `COMMIT` 即可完成购买的事务流程。

4.8.4.1.1 解决方案 1：避免死锁

为了应用程序有更好的性能，可以通过调整业务逻辑或者 Schema 设计，尽量从应用层面避免死锁。例如上面示例中，如果客户端 -B 也和客户端 -A 用同样的购买顺序，即都先买 `id=1` 的书，再买 `id=2` 的书，就可以避免死锁了：

客户端 -A	客户端 -B
BEGIN;	
	BEGIN;
UPDATE books SET stock=stock-1 WHERE id=1;	
	UPDATE books SET stock=stock-1 WHERE id=1; – 执行会被阻塞，当事务 A 完成后再继续执行
UPDATE books SET stock=stock-1 WHERE id=2;	
	UPDATE books SET stock=stock-1 WHERE id=2;
COMMIT;	
	COMMIT;

或者直接用 1 条 SQL 购买 2 本书，也能避免死锁，而且执行效率更高：

```
UPDATE books SET stock=stock-1 WHERE id IN (1, 2);
```

4.8.4.1.2 解决方案 2：减小事务粒度

如果每次购书都是一个单独的事务，也能避免死锁。但需要权衡的是，事务粒度太小不符合性能上的最佳实践。

4.8.4.1.3 解决方案 3：使用乐观事务

乐观事务模型下，并不会有死锁问题，但应用端需要加上乐观事务在失败后的重试逻辑，具体重试逻辑见应用端重试和错误处理。

4.8.4.1.4 解决方案 4：重试

正如错误信息中提示的那样，在应用代码中加入重试逻辑即可。具体重试逻辑见应用端重试和错误处理。

4.8.4.2 应用端重试和错误处理

尽管 TiDB 尽可能地与 MySQL 兼容，但其分布式系统的性质导致了某些差异，其中之一就是事务模型。

开发者用来与数据库通信的 Adapter 和 ORM 都是为 MySQL 和 Oracle 等传统数据库量身定制的，在这些数据库中，提交很少在默认隔离级别失败，因此不需要重试机制。对于这些客户端，当提交失败时，它们会因错误而中止，因为这在这些数据库中被呈现为罕见的异常。

与 MySQL 等传统数据库不同的是，在 TiDB 中，如果采用乐观事务模型，想要避免提交失败，需要在自己的应用程序的业务逻辑中添加机制来处理相关的异常。

下面的类似 Python 的伪代码展示了如何实现应用程序级的重试。它不要求您的驱动程序或 ORM 来实现高级重试处理逻辑，因此可以在任何编程语言或环境中使用。

特别是，您的重试逻辑必须：

- 如果失败重试的次数达到 `max_retries` 限制，则抛出错误
- 使用 `try ... catch ...` 语句捕获 SQL 执行异常，当遇到下面这些错误时进行失败重试，遇到其它错误则进行回滚。详细信息请参考：错误码与故障诊断。
  - `Error 8002: can not retry select for update statement`：SELECT FOR UPDATE 写入冲突报错。
  - `Error 8022: Error: KV error safe to retry`：事务提交失败报错。
  - `Error 8028: Information schema is changed during the execution of the statement`：表的 Schema 结构因为完成了 DDL 变更，导致事务提交时报错。
  - `Error 9007: Write conflict`：写冲突报错，一般是采用乐观事务模式时，多个事务都对同一行数据进行修改时遇到的写冲突报错。
- 在 try 块结束时使用 COMMIT 提交事务：

```
while True:
    n++
    if n == max_retries:
        raise("did not succeed within #{n} retries")
    try:
        connection.execute("your sql statement here")
        connection.exec('COMMIT')
        break
    catch error:
        if (error.code != "9007" && error.code != "8028" && error.code != "8002" && error.code !=
            ↪  "8022"):
            raise error
        else:
            connection.exec('ROLLBACK')

            # Capture the error types that require application-side retry,
            # wait for a short period of time,
            # and exponentially increase the wait time for each transaction failure
            sleep_ms = int(((1.5 ** n) + rand) * 100)
            sleep(sleep_ms) # make sure your sleep() takes milliseconds
```

> **注意：**
>
> 如果你经常遇到 `Error 9007: Write conflict` 错误，你可能需要进一步评估你的 Schema 设计和数据存取模型，找到冲突的根源并从设计上避免冲突。关于如何定位和解决事务冲突，请参考TiDB 锁冲突问题处理。

4.8.4.3 推荐阅读

- TiDB 锁冲突问题处理
- 乐观事务模型下写写冲突问题排查

4.9 优化 SQL 性能

4.9.1 概览

本章内容描述了如何在 TiDB 中优化 SQL 语句的性能。为了获得更好的性能，你可以从以下方面入手：

- SQL 性能调优。
- Schema 设计：根据你的业务负载类型，为了避免事务冲突或者是热点，你可能需要对表的 Schema 做出一些调整。

4.9.1.1 SQL 性能调优

为了让 SQL 语句的性能更好，可以遵循以下原则：

- 扫描的数据越少越好，最好能只扫描需要的数据，避免扫描多余的数据。
- 使用合适的索引，对于 SQL 中的 WHERE 子句中的 Column，需要保证有相应索引，否则这将是一个全表扫的语句，性能会很差。
- 使用合适的 Join 类型。根据查询中各个表的大小和关联性，选择合适的 Join 类型也会非常重要。一般情况下，TiDB 的 cost-based 优化器会自动选择最优的 Join 类型。但在少数情况下，用户手动指定 Join 类型可能会更好。
- 使用合适的存储引擎。对于 OLTP 和 OLAP 混合类型的负载，推荐使用 TiFlash 查询引擎，具体可以参考HTAP 查询。

4.9.1.2 Schema 设计

如果根据SQL 性能调优调优后任然无法获得较好的性能，你可能需要检查你的 Schema 设计和数据读取模型，来确保避免以下问题：

- 事务冲突。关于如何定位和解决事务冲突，请参考TiDB 锁冲突问题处理。
- 热点。关于如何定位和解决热点，请参考TiDB 热点问题处理。

4.9.1.3 推荐阅读

- SQL 性能调优。

4.9.2 SQL 性能调优

本章介绍常见的 SQL 性能调优，你将会了解导致 SQL 执行慢的常见的原因。

4.9.2.1 准备工作

在开始之前，你可以通过 `tiup demo` 命令导入示例数据：

```
tiup demo bookshop prepare --books 1000000 --host 127.0.0.1 --port 4000
```

或使用 TiDB Cloud 的 Import 功能导入预先准备好的示例数据。

4.9.2.2 问题：全表扫描

慢查询最常见的原因就是 SELECT 语句执行是全表扫描，或者是用了不合适的索引。

当基于不在主键或任何二级索引中的列从大表中检索少量行时，通常会获得较差的性能：

```
SELECT * FROM books WHERE title = 'Marian Yost';
```

```
+------------+-------------+-----------------------+---------------------+-------+--------+
| id         | title       | type                  | published_at        | stock | price  |
+------------+-------------+-----------------------+---------------------+-------+--------+
| 65670536   | Marian Yost | Arts                  | 1950-04-09 06:28:58 | 542   | 435.01 |
| 1164070689 | Marian Yost | Education & Reference | 1916-05-27 12:15:35 | 216   | 328.18 |
| 1414277591 | Marian Yost | Arts                  | 1932-06-15 09:18:14 | 303   | 496.52 |
| 2305318593 | Marian Yost | Arts                  | 2000-08-15 19:40:58 | 398   | 402.90 |
| 2638226326 | Marian Yost | Sports                | 1952-04-02 12:40:37 | 191   | 174.64 |
+------------+-------------+-----------------------+---------------------+-------+--------+
5 rows in set
Time: 0.582s
```

可以使用 EXPLAIN 来查看这个查询的执行计划，看看为什么查询这么慢：

```
EXPLAIN SELECT * FROM books WHERE title = 'Marian Yost';
```

```
+--
    ↪ ------------------+------------+-----------+---------------+-----------------------------------+
    ↪
| id                  | estRows    | task      | access object | operator info
    ↪                           |
+--
    ↪ ------------------+------------+-----------+---------------+-----------------------------------+
    ↪
```

```
| TableReader_7       | 1.27       | root      |               | data:Selection_6
    ↪                          |
| └─Selection_6       | 1.27       | cop[tikv] |               | eq(bookshop.books.title, "
    ↪ Marian Yost") |
|   └─TableFullScan_5 | 1000000.00 | cop[tikv] | table:books   | keep order:false
    ↪                          |
+--
    ↪ ------------------+------------+-----------+---------------+--------------------------------+
    ↪
```

从执行计划中的 TableFullScan_5 可以看出，TiDB 将会对表 books 进行全表扫描，然后对每一行都判断 title 是否满足条件。TableFullScan_5 的 estRows 值为 1000000.00，说明优化器估计这个全表扫描会扫描 1000000.00 行数据。

更多关于 EXPLAIN 的使用介绍，可以阅读使用 EXPLAIN 解读执行计划。

4.9.2.2.1 解决方案：使用索引过滤数据

为了加速上面的查询，可以在 books.title 列创建一个索引：

```
CREATE INDEX title_idx ON books (title);
```

现在再执行这个查询将会快很多：

```
SELECT * FROM books WHERE title = 'Marian Yost';
```

```
+------------+-------------+----------------------+---------------------+-------+--------+
| id         | title       | type                 | published_at        | stock | price  |
+------------+-------------+----------------------+---------------------+-------+--------+
| 1164070689 | Marian Yost | Education & Reference | 1916-05-27 12:15:35 | 216   | 328.18 |
| 1414277591 | Marian Yost | Arts                 | 1932-06-15 09:18:14 | 303   | 496.52 |
| 2305318593 | Marian Yost | Arts                 | 2000-08-15 19:40:58 | 398   | 402.90 |
| 2638226326 | Marian Yost | Sports               | 1952-04-02 12:40:37 | 191   | 174.64 |
| 65670536   | Marian Yost | Arts                 | 1950-04-09 06:28:58 | 542   | 435.01 |
+------------+-------------+----------------------+---------------------+-------+--------+
5 rows in set
Time: 0.007s
```

可以使用 EXPLAIN 来查看这个查询的执行计划，看看为什么查询变快了：

```
EXPLAIN SELECT * FROM books WHERE title = 'Marian Yost';
```

```
+--
    ↪ -------------------------+---------+-----------+-------------------------------------+-----------------
    ↪
| id                        | estRows | task      | access object                       |
    ↪ operator info                                        |
```

```
+--
    ↪ ------------------------+---------+-----------+-----------------------------------+-----------------
    ↪
| IndexLookUp_10            | 1.27    | root      |                                   |
    ↪                                                 |
| ├─IndexRangeScan_8(Build) | 1.27    | cop[tikv] | table:books, index:title_idx(title) | range
    ↪ :["Marian Yost","Marian Yost"], keep order:false |
| └─TableRowIDScan_9(Probe) | 1.27    | cop[tikv] | table:books                       | keep
    ↪ order:false                                     |
+--
    ↪ ------------------------+---------+-----------+-----------------------------------+-----------------
    ↪
```

从执行计划中的 IndexLookUp_10 可以看出，TiDB 将会通过索引 `title_idx` 来查询数据，其 `estRows` 值为 `1.27`，说明优化器估计只会扫描 `1.27` 行数据，远远小于之前全表扫的 `1000000.00` 行数据。

IndexLookUp_10 执行计划的执行流程是先用 IndexRangeScan_8 算子通过 `title_idx` 索引获取符合条件的索引数据，然后 TableRowIDScan_9 再更据索引数据里面的 Row ID 回表查询相应的行数据。

更多关于 TiDB 执行计划的内容，可以阅读TiDB 执行计划概览。

4.9.2.2.2 解决方案：使用索引查询数据

上述解决方案中，需要先读取索引信息，再回表查询对应的行数据。但如果索引数据中包含了 SQL 查询所需的所有信息，就可以省去回表查询这个步骤。

例如下面查询中，仅需要根据 `title` 查询对应的 `price`：

```
SELECT title, price FROM books WHERE title = 'Marian Yost';
```

```
+-------------+--------+
| title       | price  |
+-------------+--------+
| Marian Yost | 435.01 |
| Marian Yost | 328.18 |
| Marian Yost | 496.52 |
| Marian Yost | 402.90 |
| Marian Yost | 174.64 |
+-------------+--------+
5 rows in set
Time: 0.007s
```

由于索引 `title_idx` 仅包含 `title` 列的信息，所以 TiDB 还是需要扫描索引数据，然后回表查询 `price` 数据：

```
EXPLAIN SELECT title, price FROM books WHERE title = 'Marian Yost';
```

```
+--
    ↪ ------------------------+---------+-----------+-----------------------------------+-----------------
    ↪
```

```
| id                        | estRows | task      | access object                        |
    ↪ operator info                                          |
+--
    ↪ -------------------------+---------+-----------+-------------------------------------+-------------------
    ↪
| IndexLookUp_10            | 1.27    | root      |                                      |
    ↪                                                        |
| ├─IndexRangeScan_8(Build) | 1.27    | cop[tikv] | table:books, index:title_idx(title) | range
    ↪ :["Marian Yost","Marian Yost"], keep order:false |
| └─TableRowIDScan_9(Probe) | 1.27    | cop[tikv] | table:books                         | keep
    ↪ order:false                                      |
+--
    ↪ -------------------------+---------+-----------+-------------------------------------+-------------------
    ↪
```

删除 title_idx 索引，并新建一个 title_price_idx 索引：

```
ALTER TABLE books DROP INDEX title_idx;
```

```
CREATE INDEX title_price_idx ON books (title, price);
```

现在，price 数据已经存储在索引 title_price_idx 中了，所以下面查询仅需扫描索引数据，无需回表查询了。这种索引通常被叫做覆盖索引：

```
EXPLAIN SELECT title, price FROM books WHERE title = 'Marian Yost';
```

```
--
    ↪ ------------------+---------+-----------+-------------------------------------------------+-------------
    ↪
| id                  | estRows | task      | access object                                   |
    ↪ operator info                                          |
+--
    ↪ ------------------+---------+-----------+-------------------------------------------------+-------------
    ↪
| IndexReader_6       | 1.27    | root      |                                                 |
    ↪ index:IndexRangeScan_5                                 |
| └─IndexRangeScan_5 | 1.27    | cop[tikv] | table:books, index:title_price_idx(title, price) |
    ↪ range:["Marian Yost","Marian Yost"], keep order:false |
+--
    ↪ ------------------+---------+-----------+-------------------------------------------------+-------------
    ↪
```

现在这条查询的速度将会更快：

```
SELECT title, price FROM books WHERE title = 'Marian Yost';
```

```
+-------------+--------+
| title       | price  |
+-------------+--------+
| Marian Yost | 174.64 |
| Marian Yost | 328.18 |
| Marian Yost | 402.90 |
| Marian Yost | 435.01 |
| Marian Yost | 496.52 |
+-------------+--------+
5 rows in set
Time: 0.004s
```

由于后面的示例还会用到这个库，删除 `title_price_idx` 索引。

```
ALTER TABLE books DROP INDEX title_price_idx;
```

4.9.2.2.3 解决方案：使用主键查询数据

如果查询中使用主键过滤数据，这条查询的执行速度会非常快，例如表 `books` 的主键是列 `id`，使用列 `id` 来查询数据：

```
SELECT * FROM books WHERE id = 896;
```

```
+-----+----------------+----------------------+---------------------+-------+--------+
| id  | title          | type                 | published_at        | stock | price  |
+-----+----------------+----------------------+---------------------+-------+--------+
| 896 | Kathryne Doyle | Science & Technology | 1969-03-18 01:34:15 | 468   | 281.32 |
+-----+----------------+----------------------+---------------------+-------+--------+
1 row in set
Time: 0.004s
```

使用 `EXPLAIN` 查看执行计划:

```
EXPLAIN SELECT * FROM books WHERE id = 896;
```

```
+-------------+---------+------+---------------+---------------+
| id          | estRows | task | access object | operator info |
+-------------+---------+------+---------------+---------------+
| Point_Get_1 | 1.00    | root | table:books   | handle:896    |
+-------------+---------+------+---------------+---------------+
```

Point_Get，又名“点查”，它的执行速度也非常快。

4.9.2.3 选择合适的 Join 执行计划

见JOIN 查询的执行计划。

4.9.2.4 推荐阅读

- 使用 EXPLAIN 解读执行计划。
- 用 EXPLAIN 查看索引查询的执行计划。

4.9.3 性能调优最佳实践

本章将介绍在使用 TiDB 数据库的一些最佳实践。

4.9.3.1 DML 最佳实践

以下将介绍使用 TiDB 的 DML 时所涉及到的最佳实践。

4.9.3.1.1 使用单个语句多行数据操作

当需要修改多行数据时，推荐使用单个 SQL 多行数据的语句：

```
INSERT INTO t VALUES (1, 'a'), (2, 'b'), (3, 'c');

DELETE FROM t WHERE id IN (1, 2, 3);
```

不推荐使用多个 SQL 单行数据的语句：

```
INSERT INTO t VALUES (1, 'a');
INSERT INTO t VALUES (2, 'b');
INSERT INTO t VALUES (3, 'c');

DELETE FROM t WHERE id = 1;
DELETE FROM t WHERE id = 2;
DELETE FROM t WHERE id = 3;
```

4.9.3.1.2 使用 PREPARE

当需要多次执行某个 SQL 语句时，推荐使用 PREPARE 语句，可以避免重复解析 SQL 语法的开销。

在 Golang 中使用 PREPARE 语句：

```
func BatchInsert(db *sql.DB) error {
  stmt, err := db.Prepare("INSERT INTO t (id) VALUES (?), (?), (?), (?), (?)")
  if err != nil {
    return err
  }
  for i := 0; i < 1000; i += 5 {
    values := []interface{}{i, i + 1, i + 2, i + 3, i + 4}
    _, err = stmt.Exec(values...)
    if err != nil {
      return err
```

```
    }
  }
  return nil
}
```

在 Java 中使用 PREPARE 语句：

```
public void batchInsert(Connection connection) throws SQLException {
  PreparedStatement statement = connection.prepareStatement(
      "INSERT INTO `t` (`id`) VALUES (?), (?), (?), (?), (?)");
  for (int i = 0; i < 1000; i ++) {
    statement.setInt(i % 5 + 1, i);

    if (i % 5 == 4) {
      statement.executeUpdate();
    }
  }
}
```

在 Python 中使用 PREPARE 语句时，并不需要显式指定。在你使用参数化查询时，mysqlclient 等 Driver 将自动转用执行计划。

注意不要重复执行 PREPARE 语句，否则并不能提高执行效率。

4.9.3.1.3 避免查询不必要的信息

如非必要，不要总是用 SELECT * 返回所以列的数据，下面查询是低效的：

```
SELECT * FROM books WHERE title = 'Marian Yost';
```

应该仅查询需要的列信息，例如：

```
SELECT title, price FROM books WHERE title = 'Marian Yost';
```

4.9.3.1.4 使用批量删除

当需要删除大量的数据，推荐使用批量删除，见批量删除

4.9.3.1.5 使用批量更新

当需要更新大量的数据时，推荐使用批量更新，见批量更新

4.9.3.1.6 使用 TRUNCATE 语句代替 DELETE 全表数据

当需要删除一个表的所有数据时，推荐使用 TRUNCATE 语句：

```
TRUNCATE TABLE t;
```

不推荐使用 DELETE 全表数据：

```
DELETE FROM t;
```

4.9.3.2 DDL 最佳实践

以下将介绍使用 TiDB 的 DDL 时所涉及到的最佳实践。

4.9.3.2.1 主键选择的最佳实践

见选择主键时应遵守的规则。

4.9.3.3 索引的最佳实践

见索引的最佳实践。

4.9.3.3.1 添加索引性能最佳实践

TiDB 支持在线添加索引操作，可通过ADD INDEX 或CREATE INDEX 完成索引添加操作。添加索引不会阻塞表中的数据读写。可以通过修改下面的系统变量来调整 DDL 操作 re-organize 阶段的并行度与回填索引的单批数量大小：

- tidb_ddl_reorg_worker_cnt
- tidb_ddl_reorg_batch_size

为了减少对在线业务的影响，添加索引的默认速度会比较保守。当添加索引的目标列仅涉及查询负载，或者与线上负载不直接相关时，可以适当调大上述变量来加速添加索引：

```
SET @@global.tidb_ddl_reorg_worker_cnt = 16;
SET @@global.tidb_ddl_reorg_batch_size = 4096;
```

当添加索引操作的目标列被频繁更新（包含 UPDATE、 INSERT 和 DELETE）时，调大上述配置会造成较为频繁的写冲突，使得在线负载较大；同时添加索引操作也可能由于不断地重试，需要很长的时间才能完成。此时建议调小上述配置来避免和在线业务的写冲突：

```
SET @@global.tidb_ddl_reorg_worker_cnt = 4;
SET @@global.tidb_ddl_reorg_batch_size = 128;
```

4.9.3.4 事务冲突

关于如何定位和解决事务冲突，请参考TiDB 锁冲突问题处理。

4.9.3.5 Java 数据库应用开发最佳实践

TiDB 最佳实践系列（五）Java 数据库应用开发指南。

4.9.3.5.1 推荐阅读

- TiDB 最佳实践系列（一）高并发写入常见热点问题及规避方法。

4.9.4 索引的最佳实践

本章会介绍在 TiDB 中使用索引的一些最佳实践。

4.9.4.1 准备工作

本章内容将会用bookshop 数据库中的 books 表作为示例。

```
CREATE TABLE `books` (
  `id` bigint(20) AUTO_RANDOM NOT NULL,
  `title` varchar(100) NOT NULL,
  `type` enum('Magazine', 'Novel', 'Life', 'Arts', 'Comics', 'Education & Reference', 'Humanities
      ↪  & Social Sciences', 'Science & Technology', 'Kids', 'Sports') NOT NULL,
  `published_at` datetime NOT NULL,
  `stock` int(11) DEFAULT '0',
  `price` decimal(15,2) DEFAULT '0.0',
  PRIMARY KEY (`id`) CLUSTERED
) DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin;
```

4.9.4.2 创建索引的最佳实践

- 建立你需要使用的数据的所有列的组合索引，这种优化技巧被称为覆盖索引优化 (covering index optimization)。覆盖索引优化将使得 TiDB 可以直接在索引上得到该查询所需的所有数据，可以大幅提升性能。
- 避免创建你不需要的二级索引，有用的二级索引能加速查询，但是要注意新增一个索引是有副作用的。每增加一个索引，在插入一条数据的时候，就要额外新增一个 Key-Value，所以索引越多，写入越慢，并且空间占用越大。另外过多的索引也会影响优化器运行时间，并且不合适的索引会误导优化器。所以索引并不是越多越好。
- 根据具体的业务特点创建合适的索引。原则上需要对查询中需要用到的列创建索引，目的是提高性能。下面几种情况适合创建索引：
  - 区分度比较大的列，通过索引能显著地减少过滤后的行数。例如推荐在人的身份证号码这一列上创建索引，但不推荐在人的性别这一列上创建索引。
  - 有多个查询条件时，可以选择组合索引，注意需要把等值条件的列放在组合索引的前面。这里举一个例子，假设常用的查询是 `SELECT * FROM t where c1 = 10 and c2 = 100 and c3 > 10`，那么可以考虑建立组合索引 `Index cidx (c1, c2, c3)`，这样可以用查询条件构造出一个索引前缀进行 Scan。
- 请使用有意义的二级索引名，推荐你遵循公司或组织的表命名规范。如果你的公司或组织没有相应的命名规范，可参考索引命名规范。

4.9.4.3 使用索引的最佳实践

- 建立索引的目的是为了加速查询，所以请确保索引能在一些查询中被用上。如果一个索引不会被任何查询语句用到，那这个索引是没有意义的，请删除这个索引。
- 使用组合索引时，需要满足最左前缀原则。

  例如假设在列 `title, published_at` 上新建一个组合索引索引：

```
CREATE INDEX title_published_at_idx ON books (title, published_at);
```

  下面这个查询依然能用上这个组合索引：

```
SELECT * FROM books WHERE title = 'database';
```

  但下面这个查询由于未指定组合索引中最左边第一列的条件，所以无法使用组合索引：

```
SELECT * FROM books WHERE published_at = '2018-08-18 21:42:08';
```

- 在查询条件中使用索引列作为条件时，不要在索引列上做计算，函数，或者类型转换的操作，会导致优化器无法使用该索引。

  例如假设在时间类型的列 `published_at` 上新建一个索引：

```
CREATE INDEX published_at_idx ON books (published_at);
```

  但下面查询是无法使用 `published_at` 上的索引的：

```
SELECT * FROM books WHERE YEAR(published_at)=2022;
```

  可以改写成下面查询，避免在索引列上做函数计算后，即可使用 `published_at` 上的索引:

```
SELECT * FROM books WHERE published_at >= '2022-01-01' AND published_at < '2023-01-01';
```

  也可以使用表达式索引，例如对查询条件中的 `YEAR(published_at)` 创建一个表达式索引：

```
CREATE INDEX published_year_idx ON books ((YEAR(published_at)));
```

  然后通过 `SELECT * FROM books WHERE YEAR(published_at)=2022;` 查询就能使用 `published_year_idx` 索引来加速查询了。

> 注意：
> 表达式索引目前是 TiDB 的实验特性，需要在 TiDB 配置文件中开启表达式索引特性，详情可以参考表达式索引文档。

- 尽量使用覆盖索引，即索引列包含查询列，避免总是 `SELECT *` 查询所有列的语句。

  例如下面查询只需扫描索引 `title_published_at_idx` 数据即可获取查询列的数据：

```
SELECT title, published_at FROM books WHERE title = 'database';
```

但下面查询语句虽然能用上组合索引 (title, published_at)，但会多一个回表查询非索引列数据的额外开销，回表查询是指根据索引数据中存储的引用（一般是主键信息），到表中查询相应行的数据。

```
SELECT * FROM books WHERE title = 'database';
```

- 查询条件使用 !=，NOT IN 时，无法使用索引。例如下面查询无法使用任何索引：

```
SELECT * FROM books WHERE title != 'database';
```

- 使用 LIKE 时如果条件是以通配符 % 开头，也无法使用索引。例如下面查询无法使用任何索引：

```
SELECT * FROM books WHERE title LIKE '%database';
```

- 当查询条件有多个索引可供使用，但你知道用哪一个索引是最优的时，推荐使用优化器 Hint 来强制优化器使用这个索引，这样可以避免优化器因为统计信息不准或其他问题时，选错索引。

  例如下面查询中，假设在列 id 和列 title 上都各自有索引 id_idx 和 title_idx，你知道 id_idx 的过滤性更好，就可以在 SQL 中使用 USE INDEX Hint 来强制优化器使用 id_idx 索引。

```
SELECT * FROM t USE INDEX(id_idx) WHERE id = 1 and title = 'database';
```

- 查询条件使用 IN 表达式时，后面匹配的条件数量建议不要超过 300 个，否则执行效率会较差。

4.9.5 其他优化

4.9.5.1 避免隐式类型转换

本章内容将介绍 TiDB 中的隐式类型转换规则、可能带来的后果及避免方法。

4.9.5.1.1 转换规则

当 SQL 中谓词两侧的数据类型不一致时，TiDB 将隐式地将一侧或两侧的数据类型进行转换，将其变为兼容的数据类型，以进行谓词运算。

TiDB 中隐式类型转换规则如下：

- 如果一个或两个参数都是 NULL，比较的结果是 NULL（NULL 安全的 <=> 相等比较运算符除外，对于 NULL <=> NULL，结果为 true，不需要转换）。
- 如果比较操作中的两个参数都是字符串，则将它们作为字符串进行比较。
- 如果两个参数都是整数，则将它们作为整数进行比较。
- 如果不与数字进行比较，则将十六进制值视为二进制字符串。
- 如果其中一个参数是十进制值，则比较取决于另一个参数。如果另一个参数是十进制或整数值，则将参数与十进制值进行比较，如果另一个参数是浮点值，则将参数与浮点值进行比较。
- 如果其中一个参数是 TIMESTAMP 或 DATETIME 列，另一个参数是常量，则在执行比较之前将常量转换为时间戳。
- 在所有其他情况下，参数都是作为浮点数（DOUBLE 类型）比较的。

4.9.5.1.2 隐式类型转换引起的后果

隐式类型转换增强了人机交互的易用性，但在应用代码中，应尽量避免隐式类型转换出现，这是由于隐式类型转换会导致：

- 索引失效
- 精度丢失

索引失效

如下案例，account_id 为主键，其数据类型为 varchar。通过执行计划可见，该 SQL 发生了隐式类型转换，无法使用索引。

```
DESC SELECT * FROM `account` WHERE `account_id`=6010000000009801;
+--
    ↪ ----------------------+---------------+-----------+--------------+--------------------------------
    ↪
| id                     | estRows        | task      | access object | operator info
    ↪                                              |
+--
    ↪ ----------------------+---------------+-----------+--------------+--------------------------------
    ↪
| TableReader_7           | 8000628000.00  | root      |               | data:Selection_6
    ↪                                           |
| └─Selection_6           | 8000628000.00  | cop[tikv] |               | eq(cast(findpt.account.
    ↪ account_id), 6.010000000009801e+15) |
|   └─TableFullScan_5     | 10000785000.00 | cop[tikv] | table:account | keep order:false
    ↪                                           |
+--
    ↪ ----------------------+---------------+-----------+--------------+--------------------------------
    ↪
3 rows in set (0.00 sec)
```

运行结果简述：从以上执行计划中，可见 Cast 算子。

精度丢失

如下案例，字段 a 的数据类型为 decimal(32,0)，从执行计划可以得知，出现了隐式类型转换，decimal 字段和字符串常值都被转换为 double 类型，而 double 类型的精度没有 decimal 高，出现了精度丢失，在这个 case 中，造成了筛选出范围之外的结果集的错误。

```
DESC SELECT * FROM `t1` WHERE `a` BETWEEN '12123123' AND '1111222211111111200000';
+--
    ↪ ----------------------+---------+-----------+--------------+--------------------------------
    ↪
| id                     | estRows | task      | access object | operator info
    ↪                                                                  |
+--
    ↪ ----------------------+---------+-----------+--------------+--------------------------------
    ↪
```

```
| TableReader_7           | 0.80    | root      |                | data:Selection_6
    ↪                                                                  |
| └─Selection_6           | 0.80    | cop[tikv] |                | ge(cast(findpt.t1.a),
    ↪ 1.2123123e+07), le(cast(findpt.t1.a), 1.1112222111111112e+21) |
|   └─TableFullScan_5     | 1.00    | cop[tikv] | table:t1       | keep order:false, stats:pseudo
    ↪                                                        |
+--
    ↪ -----------------------+---------+-----------+---------------+-------------------------------------------
    ↪
3 rows in set (0.00 sec)
```

运行结果简述：从以上执行计划中，可见 Cast 算子。

```
SELECT * FROM `t1` WHERE `a` BETWEEN '12123123' AND '1111222211111111200000';
+------------------------+
| a                      |
+------------------------+
| 1111222211111111222211 |
+------------------------+
1 row in set (0.01 sec)
```

运行结果简述：以上执行出现了错误结果。

4.9.5.2 唯一序列号生成方案

本章将介绍唯一序列号生成方案，为自行生成唯一 ID 的开发者提供帮助。

4.9.5.2.1 自增列

自增（auto_increment）是大多数兼容 MySQL 协议的 RDBMS 上列的一种属性，通过配置该属性来使数据库为该列的值自动赋值，用户不需要为该列赋值，该列的值随着表内记录增加会自动增长，并确保唯一性。在大多数场景中，自增列并未拥有业务属性，仅仅代表了这一行数据，即被作为无业务含义的代理主键使用。自增列的局限性在于：自增列只能采用整型字段，所赋的值也只能为整型。假设业务所需要的序列号由字母、数字及其他字符拼接而成，用户将难以通过自增列来获取序列号中所需的数字自增值。

4.9.5.2.2 序列（Sequence）

序列是一种数据库对象，应用程序通过调用某个序列可以产生递增的序列值，应用程序可以灵活的使用这个序列值为一张表或多张表赋值，也可以使用序列值进行更复杂的加工，来实现文本和数字的组合，来赋予代理键以一定的跟踪和分类的意义。从 TiDB v4.0 版本开始提供序列功能，详情请参考CREATE SEQUENCE。

4.9.5.2.3 类 Snowflake 方案

Snowflake 是 Twitter 提出的分布式 ID 生成方案。目前有多种实现，较流行的是百度的 uid-generator 和美团的 leaf。下面以 uid-generator 为例展开说明。

uid-generator 生成的 64 位 ID 结构如下：

```
| sign | delta seconds | worker node id | sequencs |
|------|---------------|----------------|----------|
| 1bit |     28bits    | 22bits         | 13bits   |
```

- sign：长度固定为 1 位。固定为 0，表示生成的 ID 始终为正数。
- delta seconds：默认 28 位。当前时间，表示为相对于某个预设时间基点 (默认“2016-05-20”) 的增量值，单位为秒。28 位最多可支持约 8.7 年。
- worker node id：默认 22 位。表示机器 id，通常在应用程序进程启动时从一个集中式的 ID 生成器取得。常见的集中式 ID 生成器是数据库自增列或者 Zookeeper。默认分配策略为用后即弃，进程重启时会重新获取一个新的 worker node id，22 位最多可支持约 420 万次启动。
- sequence：默认 13 位。表示每秒的并发序列，13 位可支持每秒 8192 个并发。

4.9.5.2.4 号段分配方案

号段分配方案可以理解为从数据库批量获取自增 ID。本方案需要一张序列号生成表，每行记录表示一个序列对象。表定义示例如下：

字段名	字段类型	字段说明
SEQ_NAME	varchar(128)	序列名称，用来区分不同业务
MAX_ID	bigint(20)	当前序列已被分配出去的最大值
STEP	int(11)	步长，表示每次分配的号段长度

应用程序每次按配置好的步长获取一段序列号，并同时更新数据库以持久化保存当前序列已被分配出去的最大值，然后在应用程序内存中即可完成序列号加工及分配动作。待一段号码耗尽之后，应用程序才会去获取新的号段，这样就有效降低了数据库写入压力。实际使用过程中，还可以适度调节步长以控制数据库记录的更新频度。

最后，需要注意的是，上述两种方案生成的 ID 都不够随机，不适合直接作为 TiDB 表的主键。实际使用过程中可以对生成的 ID 进行位反转（bit-reverse）后得到一个较为随机的新 ID。

4.10 故障诊断

4.10.1 SQL 或事务问题

本章介绍在开发应用过程中可能遇到的常见问题的诊断处理方法。

4.10.1.1 SQL 操作常见问题

如果你想提高 SQL 的性能，可以阅读SQL 性能优化来避免一些常见的性能问题。然后如果依然存在性能问题，推荐阅读：

- 分析慢查询
- 使用 Top SQL 定位系统资源消耗过多的查询

如果你遇到了一些关于 SQL 操作的问题，可以阅读SQL 操作常见问题。

4.10.1.2 事务错误处理

见事务错误处理。

4.10.1.3 推荐阅读

- 不支持的功能特性
- 集群管理 FAQ
- TiDB 产品 FAQ

4.10.2 结果集不稳定

本章将叙述结果集不稳定错误的处理办法。

4.10.2.1 group by

出于便捷的考量，MySQL“扩展”了 group by 语法，使 select 子句可以引用未在 group by 子句中声明的非聚集字段，也就是 non-full group by 语法。在其他数据库中，这被认为是一种语法错误，因为这会导致结果集不稳定。

在下例的 3 条 SQL 语句中，第一条 SQL 使用了 full group by 语法，所有在 select 子句中引用的字段，都在 group by 子句中有所声明，所以它的结果集是稳定的，可以看到 class 与 stuname 的全部组合共有三种；第二条与第三条是同一个 SQL，但它在两次执行时得到了不同的结果，这条 SQL 的 group by 子句中仅声明了一个 class 字段，因此结果集只会针对 class 进行聚集，class 的唯一值有两个，也就是说结果集中只会包含两行数据，而 class 与 stuname 的全部组合共有三种，班级 2018_CS_03 有两位同学，每次执行时返回哪位同学是没有语义上的限制的，都是符合语义的结果。

```
mysql> SELECT a.class, a.stuname, max(b.courscore) from stu_info a join stu_score b on a.stuno=b.
    ↪ stuno group by a.class, a.stuname order by a.class, a.stuname;
+------------+--------------+------------------+
| class      | stuname      | max(b.courscore) |
+------------+--------------+------------------+
| 2018_CS_01 | MonkeyDLuffy |             95.5 |
| 2018_CS_03 | PatrickStar  |             99.0 |
| 2018_CS_03 | SpongeBob    |             95.0 |
+------------+--------------+------------------+
3 rows in set (0.00 sec)

mysql> select a.class, a.stuname, max(b.courscore) from stu_info a join stu_score b on a.stuno=b.
    ↪ stuno group by a.class order by a.class, a.stuname;
+------------+--------------+------------------+
| class      | stuname      | max(b.courscore) |
+------------+--------------+------------------+
| 2018_CS_01 | MonkeyDLuffy |             95.5 |
| 2018_CS_03 | SpongeBob    |             99.0 |
+------------+--------------+------------------+
2 rows in set (0.01 sec)
```

```
mysql> select a.class, a.stuname, max(b.courscore) from stu_info a join stu_score b on a.stuno=b.
    ↪ stuno group by a.class order by a.class, a.stuname;
+------------+--------------+------------------+
| class      | stuname      | max(b.courscore) |
+------------+--------------+------------------+
| 2018_CS_01 | MonkeyDLuffy |             95.5 |
| 2018_CS_03 | PatrickStar  |             99.0 |
+------------+--------------+------------------+
2 rows in set (0.01 sec)
```

因此，想保障 group by 语句结果集的稳定，请使用 full group by 语法。

MySQL 提供了一个 SQL_MODE 开关 ONLY_FULL_GROUP_BY 来控制是否进行 full group by 语法的检查，TiDB 也兼容了这个 SQL_MODE 开关：

```
mysql> select a.class, a.stuname, max(b.courscore) from stu_info a join stu_score b on a.stuno=b.
    ↪ stuno group by a.class order by a.class, a.stuname;
+------------+--------------+------------------+
| class      | stuname      | max(b.courscore) |
+------------+--------------+------------------+
| 2018_CS_01 | MonkeyDLuffy |             95.5 |
| 2018_CS_03 | PatrickStar  |             99.0 |
+------------+--------------+------------------+
2 rows in set (0.01 sec)

mysql> set @@sql_mode='STRICT_TRANS_TABLES,NO_ENGINE_SUBSTITUTION,ONLY_FULL_GROUP_BY';
Query OK, 0 rows affected (0.01 sec)

mysql> select a.class, a.stuname, max(b.courscore) from stu_info a join stu_score b on a.stuno=b.
    ↪ stuno group by a.class order by a.class, a.stuname;
ERROR 1055 (42000): Expression #2 of ORDER BY is not in GROUP BY clause and contains
    ↪ nonaggregated column '' which is not functionally dependent on columns in GROUP BY clause
    ↪ ; this is incompatible with sql_mode=only_full_group_by
```

运行结果简述：上例为 sql_mode 设置了 ONLY_FULL_GROUP_BY 的效果。

4.10.2.2 order by

在 SQL 的语义中，只有使用了 order by 语法才会保障结果集的顺序输出。而单机数据库由于数据都存储在一台服务器上，在不进行数据重组时，多次执行的结果往往是稳定的，有些数据库 (尤其是 MySQL InnoDB 存储引擎) 还会按照主键或索引的顺序进行结果集的输出。TiDB 是分布式数据库，数据被存储在多台服务器上，另外 TiDB 层不缓存数据页，因此不含 order by 的 SQL 语句的结果集展现顺序容易被感知到不稳定。想要按顺序输出的结果集，需明确地把要排序的字段添加到 order by 子句中，这符合 SQL 的语义。

在下面的案例中，用户只在 order by 子句中添加了一个字段，TiDB 只会按照这一个字段进行排序。

```
mysql> select a.class, a.stuname, b.course, b.courscore from stu_info a join stu_score b on a.
    ↪ stuno=b.stuno order by a.class;
```

```
+------------+--------------+-------------------------+-----------+
| class      | stuname      | course                  | courscore |
+------------+--------------+-------------------------+-----------+
| 2018_CS_01 | MonkeyDLuffy | PrinciplesofDatabase    |      60.5 |
| 2018_CS_01 | MonkeyDLuffy | English                 |      43.0 |
| 2018_CS_01 | MonkeyDLuffy | OpSwimming              |      67.0 |
| 2018_CS_01 | MonkeyDLuffy | OpFencing               |      76.0 |
| 2018_CS_01 | MonkeyDLuffy | FundamentalsofCompiling |      88.0 |
| 2018_CS_01 | MonkeyDLuffy | OperatingSystem         |      90.5 |
| 2018_CS_01 | MonkeyDLuffy | PrincipleofStatistics   |      69.0 |
| 2018_CS_01 | MonkeyDLuffy | ProbabilityTheory       |      76.0 |
| 2018_CS_01 | MonkeyDLuffy | Physics                 |      63.5 |
| 2018_CS_01 | MonkeyDLuffy | AdvancedMathematics     |      95.5 |
| 2018_CS_01 | MonkeyDLuffy | LinearAlgebra           |      92.5 |
| 2018_CS_01 | MonkeyDLuffy | DiscreteMathematics     |      89.0 |
| 2018_CS_03 | SpongeBob    | PrinciplesofDatabase    |      88.0 |
| 2018_CS_03 | SpongeBob    | English                 |      79.0 |
| 2018_CS_03 | SpongeBob    | OpBasketball            |      92.0 |
| 2018_CS_03 | SpongeBob    | OpTennis                |      94.0 |
| 2018_CS_03 | PatrickStar  | LinearAlgebra           |       6.5 |
| 2018_CS_03 | PatrickStar  | AdvancedMathematics     |       5.0 |
| 2018_CS_03 | SpongeBob    | DiscreteMathematics     |      72.0 |
| 2018_CS_03 | PatrickStar  | ProbabilityTheory       |      12.0 |
| 2018_CS_03 | PatrickStar  | PrincipleofStatistics   |      20.0 |
| 2018_CS_03 | PatrickStar  | OperatingSystem         |      36.0 |
| 2018_CS_03 | PatrickStar  | FundamentalsofCompiling |       2.0 |
| 2018_CS_03 | PatrickStar  | DiscreteMathematics     |      14.0 |
| 2018_CS_03 | PatrickStar  | PrinciplesofDatabase    |       9.0 |
| 2018_CS_03 | PatrickStar  | English                 |      60.0 |
| 2018_CS_03 | PatrickStar  | OpTableTennis           |      12.0 |
| 2018_CS_03 | PatrickStar  | OpPiano                 |      99.0 |
| 2018_CS_03 | SpongeBob    | FundamentalsofCompiling |      43.0 |
| 2018_CS_03 | SpongeBob    | OperatingSystem         |      95.0 |
| 2018_CS_03 | SpongeBob    | PrincipleofStatistics   |      90.0 |
| 2018_CS_03 | SpongeBob    | ProbabilityTheory       |      87.0 |
| 2018_CS_03 | SpongeBob    | Physics                 |      65.0 |
| 2018_CS_03 | SpongeBob    | AdvancedMathematics     |      55.0 |
| 2018_CS_03 | SpongeBob    | LinearAlgebra           |      60.5 |
| 2018_CS_03 | PatrickStar  | Physics                 |       6.0 |
+------------+--------------+-------------------------+-----------+
36 rows in set (0.01 sec)
```

当遇到相同的 order by 值时，排序结果不稳定。为减少随机性，应当尽可能保持 order by 值的唯一性。不能保证唯一的继续加，保证 order by 的字段组合是唯一时，结果才能唯一。

4.10.2.3 由于 group_concat() 中没有使用 order by 导致结果集不稳定

结果集不稳定是因为 TiDB 是并行地从存储层读取数据，所以 `group_concat()` 在不加 order by 的情况下得到的结果集展现顺序容易被感知到不稳定。

`group_concat()` 要获取到按顺序输出的结果集，需要把用于排序的字段添加到 order by 子句中，这样才符合 SQL 的语义。在下面的案例中，使用 `group_concat()` 不加 order by 的情况下拼接 customer_id，造成结果集不稳定：

1. 不加 order by

第一次查询：

```
sql  mysql> select GROUP_CONCAT( customer_id SEPARATOR ',' )FROM customer where customer_id like
↪  '200002%'; +----------------------------------------------------------------------------+ |
↪ GROUP_CONCAT(customer_id SEPARATOR ',')| +--------------------------------------------------------------------------
↪  | 20000200992,20000200993,20000200994,20000200995,20000200996,20000200... | +--------------------------
↪
```

第二次查询：

```
sql  mysql> select GROUP_CONCAT( customer_id SEPARATOR ',' )FROM customer where customer_id like
↪  '200002%'; +----------------------------------------------------------------------------+ |
↪ GROUP_CONCAT(customer_id SEPARATOR ',')| +--------------------------------------------------------------------------
↪  | 20000203040,20000203041,20000203042,20000203043,20000203044,20000203... | +--------------------------
↪
```

2. 加 order by

第一次查询：

```
sql  mysql> select GROUP_CONCAT( customer_id order by customer_id SEPARATOR ',' )FROM customer
↪ where customer_id like '200002%'; +-----------------------------------------------------------------------------+
↪  | GROUP_CONCAT(customer_id SEPARATOR ',')| +--------------------------------------------------------------------------
↪  | 20000200000,20000200001,20000200002,20000200003,20000200004,20000200... | +--------------------------
↪
```

第二次查询：

```
sql  mysql> select GROUP_CONCAT( customer_id order by customer_id SEPARATOR ',' )FROM customer
↪ where customer_id like '200002%'; +-----------------------------------------------------------------------------+
↪  | GROUP_CONCAT(customer_id SEPARATOR ',')| +--------------------------------------------------------------------------
↪  | 20000200000,20000200001,20000200002,20000200003,20000200004,20000200... | +--------------------------
↪
```

4.10.2.4 select * from t limit n 的结果不稳定

返回结果与数据在存储节点 (TiKV) 上的分布有关。如果进行了多次查询，存储节点 (TiKV) 不同存储单元 (Region) 返回结果的速度不同，会造成结果不稳定。

4.10.3 TiDB 中的各种超时

本章将介绍 TiDB 中的各种超时，为排查错误提供依据。

4.10.3.1 GC 超时

TiDB 的事务的实现采用了 MVCC（多版本并发控制）机制，当新写入的数据覆盖旧的数据时，旧的数据不会被替换掉，而是与新写入的数据同时保留，并以时间戳来区分版本。TiDB 通过定期 GC 的机制来清理不再需要的旧数据。

默认配置下 TiDB 可以保障每个 MVCC 版本（一致性快照）保存 10 分钟，读取时间超过 10 分钟的事务，会收到报错 `GC life time is shorter than transaction duration`。

当用户确信自己需要更长的读取时间时，比如在使用了 Mydumper 做全量备份的场景中（Mydumper 备份的是一致性的快照），可以通过调整 TiDB 中`mysql.tidb` 表中的 `tikv_gc_life_time` 的值来调大 MVCC 版本保留时间，需要注意的是 tikv_gc_life_time 的配置是立刻影响全局的，调大它会为当前所有存在的快照增加生命时长，调小它会立即缩短所有快照的生命时长。过多的 MVCC 版本会拖慢 TiKV 的处理效率，在使用 Mydumper 做完全量备份后需要及时把 tikv_gc_life_time 调整回之前的设置。

更多关于 GC 的信息，请参考 GC 机制简介文档。

4.10.3.2 事务超时

垃圾回收 (GC) 不会影响到正在执行的事务。但悲观事务的运行仍有上限，有基于事务超时的限制（TiDB 配置文件performance 类别下的 `max-txn-ttl` 修改，默认为 60 分钟）和基于事务使用内存的限制。

形如 `INSERT INTO t10 SELECT * FROM t1` 的 SQL 语句，不会受到 GC 的影响，但超过了 `max-txn-ttl` 的时间后，会由于超时而回滚。

4.10.3.3 SQL 执行时间超时

TiDB 还提供了一个系统变量来限制单条 SQL 语句的执行时间，仅对“只读”语句生效：`max_execution_time`，它的默认值为 0，表示无限制。`max_execution_time` 的单位为 ms，但实际精度在 100ms 级别，而非更准确的毫秒级别。

4.10.3.4 JDBC 查询超时

MySQL jdbc 的查询超时设置 `setQueryTimeout()` 对 TiDB 不起作用。这是因为现实客户端感知超时时，向数据库发送一个 KILL 命令。但是由于 tidb-server 是负载均衡的，为防止在错误的 tidb-server 上终止连接，tidb-server 不会执行这个 KILL。这时就要用 `MAX_EXECUTION_TIME` 实现查询超时的效果。

TiDB 提供了三个与 MySQL 兼容的超时控制参数：

- wait_timeout，控制与 Java 应用连接的非交互式空闲超时时间。在 TiDB v5.4 及以上版本中，默认值为 28800 秒，即空闲超时为 8 小时。在 v5.4 之前，默认值为 0，即没有时间限制。
- interactive_timeout，控制与 Java 应用连接的交互式空闲超时时间，默认值为 8 小时。
- max_execution_time，控制连接中 SQL 执行的超时时间，仅对“只读”语句生效，默认值是 0，即允许连接无限忙碌（一个 SQL 语句执行无限的长的时间）。

但在实际生产环境中，空闲连接和一直无限执行的 SQL 对数据库和应用都有不好的影响。你可以通过在应用的连接字符串中配置这两个 session 级的变量来避免空闲连接和执行时间过长的 SQL 语句。例如，设置 sessionVariables=wait_timeout=3600（1 小时）和 sessionVariables=max_execution_time=300000（5 分钟）。

4.11 引用文档

4.11.1 Bookshop 应用

Bookshop 是一个虚拟的在线书店应用，你可以在 Bookshop 当中便捷地购买到各种类别的书，也可以对你看过的书进行点评。

为了方便你阅读应用开发指南中的内容，我们将以 Bookshop 应用的数据表结构和数据为基础来编写示例 SQL。本章节将为你介绍如何导入该应用的表结构和数据，以及其数据表结构的定义。

4.11.1.1 导入表结构和数据

你可以通过 TiUP 或通过 TiDB Cloud Import 两种方式导入 Bookshop 应用的表结构和数据。

4.11.1.1.1 方法一：通过 tiup demo 命令行

如果你使用TiUP 部署 TiDB 集群或者你可以直接连接到你的 TiDB 服务器，你可以通过如下命令快速生成并导入 Bookshop 应用的示例数据：

```
tiup demo bookshop prepare
```

该命令默认会连接到 127.0.0.1 地址上的 4000 端口，使用 root 用户名进行无密码登录，默认在名为 bookshop 的数据库中创建表结构。

配置连接信息

你可以通过如下参数修改默认的连接信息：

参数	简写	默认值	解释
--host	-H	127.0.0.1	数据库地址
--port	-P	4000	数据库端口
--user	-U	root	数据库用户
--password	-p	无	数据库用户密码
--db	-D	bookshop	数据库名称

例如，你想要连接到 TiDB Cloud 上的数据库，你可以如下命令指定连接信息进行连接：

```
tiup demo bookshop prepare -U <username> -H <endpoint> -P 4000 -p <password>
```

设置数据量

另外，你还可以通过如下参数指定各个数据库表生成的数据量：

参数	默认值	解释
`--users`	10000	指定在 `users` 表生成的数据行数
`--authors`	20000	指定在 `authors` 表生成的数据行数
`--books`	20000	指定在 `books` 表生成的数据行数
`--orders`	300000	指定在 `orders` 表生成的数据行数
`--ratings`	300000	指定在 `ratings` 表生成的数据行数

例如，以下命令通过 `--users` 参数指定生成 20 万行用户信息，通过 `--books` 参数指定生成 50 万行书籍的基本信息，通过 `--authors` 参数指定生成 10 万的作者信息，通过 `--ratings` 参数指定生成 100 万的评分记录，通过 `--orders` 参数指定生成 100 万的订单记录。

```
tiup demo bookshop prepare --users=200000 --books=500000 --authors=100000 --ratings=1000000 --
    ↪ orders=1000000 --drop-tables
```

通过 `--drop-tables` 参数你可以删除原有的表结构，更多的参数说明你可以通过命令 `tiup demo bookshop --help` 进行了解。

4.11.1.1.2 方法二：通过 TiDB Cloud Import 功能

在 TiDB Cloud 的集群详情页面，你可以通过切换到 Import 标签页，点击 Import Data 按钮进入到 Data Import 页面。在该页面当中，按照以下步骤将 Bookshop 示例数据从 AWS S3 中导入到你的 TiDB Cloud 集群：

1. 选择 Data Format 为 SQL File。
2. 将以下 Bucket URI 和 Role ARN 复制到页面上对应的输入框当中：

   Bucket URI:

   ```
   s3://developer.pingcap.com/bookshop/
   ```

   Role ARN:

   ```
   arn:aws:iam::494090988690:role/s3-tidb-cloud-developer-access
   ```

3. 点击 Next 按钮切换到 Choose the tables to be imported 步骤确认将要导入的文件的信息。
4. 点击 Next 按钮切换到 Preview 步骤确认将要导入的示例数据是否正确。

   在这个示例数据当中，预先生成了 20 万的用户信息、50 万条书籍信息、10 万条作者信息、100 万条评分记录以及 100 万条订单信息。
5. 点击 Start Import 按钮开始导入数据，等待 TiDB Cloud 完成数据导入。

你可以通过 TiDB Cloud Migration Overview 文档获取更多有关 TiDB Cloud 数据导入和迁移的信息。

4.11.1.1.3 查看数据导入情况

导入完成后，你可以通过下面的 SQL 语句查看各个表的数据量信息：

```
SELECT
    CONCAT(table_schema,'.',table_name) AS 'Table Name',
    table_rows AS 'Number of Rows',
    CONCAT(ROUND(data_length/(1024*1024*1024),4),'G') AS 'Data Size',
    CONCAT(ROUND(index_length/(1024*1024*1024),4),'G') AS 'Index Size',
    CONCAT(ROUND((data_length+index_length)/(1024*1024*1024),4),'G') AS 'Total'
FROM
    information_schema.TABLES
WHERE table_schema LIKE 'bookshop';
```

运行结果为：

```
+----------------------+----------------+-----------+------------+---------+
| Table Name           | Number of Rows | Data Size | Index Size | Total   |
+----------------------+----------------+-----------+------------+---------+
| bookshop.orders      |        1000000 | 0.0373G   | 0.0075G    | 0.0447G |
| bookshop.book_authors |        1000000 | 0.0149G   | 0.0149G    | 0.0298G |
| bookshop.ratings     |        4000000 | 0.1192G   | 0.1192G    | 0.2384G |
| bookshop.authors     |         100000 | 0.0043G   | 0.0000G    | 0.0043G |
| bookshop.users       |         195348 | 0.0048G   | 0.0021G    | 0.0069G |
| bookshop.books       |        1000000 | 0.0546G   | 0.0000G    | 0.0546G |
+----------------------+----------------+-----------+------------+---------+
6 rows in set (0.03 sec)
```

4.11.1.2 数据表详解

以下将详细介绍 Bookshop 应用程序的数据库表结构：

4.11.1.2.1 books 表

该表用于存储书籍的基本信息。

字段名	类型	含义
id	bigint(20)	书籍的唯一标识
title	varchar(100)	书籍名称
type	enum	书籍类型（如：杂志、动漫、教辅等）
stock	bigint(20)	库存
price	decimal(15,2)	价格
published_at	datetime	出版时间

4.11.1.2.2 authors 表

该表用于存储作者的基本信息。

字段名	类型	含义
id	bigint(20)	作者的唯一标识
name	varchar(100)	姓名
gender	tinyint(1)	生理性别 (0: 女, 1: 男，NULL: 未知)
birth_year	smallint(6)	生年
death_year	smallint(6)	卒年

4.11.1.2.3 `users` 表

该表用于存储使用 Bookshop 应用程序的用户。

字段名	类型	含义
id	bigint(20)	用户的唯一标识
balance	decimal(15,2)	余额
nickname	varchar(100)	昵称

4.11.1.2.4 `ratings` 表

该表用于存储用户对书籍的评分记录。

字段名	类型	含义
book_id	bigint	书籍的唯一标识（关联至books）
user_id	bigint	用户的唯一标识（关联至users）
score	tinyint	用户评分 (1-5)
rated_at	datetime	评分时间

4.11.1.2.5 `book_authors` 表

一个作者可能会编写多本书，一本书可能需要多个作者同时编写，该表用于存储书籍与作者之间的对应关系。

字段名	类型	含义
book_id	bigint(20)	书籍的唯一标识（关联至books）
author_id	bigint(20)	作者的唯一标识（关联至authors）

4.11.1.2.6 `orders` 表

该表用于存储用户购买书籍的订单信息。

字段名	类型	含义
id	bigint(20)	订单的唯一标识
book_id	bigint(20)	书籍的唯一标识（关联至books）

字段名	类型	含义
user_id	bigint(20)	用户唯一标识（关联至users）
quantity	tinyint(4)	购买数量
ordered_at	datetime	购买时间

4.11.1.3 数据库初始化 dbinit.sql 脚本

如果你希望手动创建 Bookshop 应用的数据库表结构，你可以运行以下 SQL 语句：

```
CREATE DATABASE IF NOT EXISTS `bookshop`;

DROP TABLE IF EXISTS `bookshop`.`books`;
CREATE TABLE `bookshop`.`books` (
  `id` bigint(20) AUTO_RANDOM NOT NULL,
  `title` varchar(100) NOT NULL,
  `type` enum('Magazine', 'Novel', 'Life', 'Arts', 'Comics', 'Education & Reference', 'Humanities
    ↪  & Social Sciences', 'Science & Technology', 'Kids', 'Sports') NOT NULL,
  `published_at` datetime NOT NULL,
  `stock` int(11) DEFAULT '0',
  `price` decimal(15,2) DEFAULT '0.0',
  PRIMARY KEY (`id`) CLUSTERED
) DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin;

DROP TABLE IF EXISTS `bookshop`.`authors`;
CREATE TABLE `bookshop`.`authors` (
  `id` bigint(20) AUTO_RANDOM NOT NULL,
  `name` varchar(100) NOT NULL,
  `gender` tinyint(1) DEFAULT NULL,
  `birth_year` smallint(6) DEFAULT NULL,
  `death_year` smallint(6) DEFAULT NULL,
  PRIMARY KEY (`id`) CLUSTERED
) DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin;

DROP TABLE IF EXISTS `bookshop`.`book_authors`;
CREATE TABLE `bookshop`.`book_authors` (
  `book_id` bigint(20) NOT NULL,
  `author_id` bigint(20) NOT NULL,
  PRIMARY KEY (`book_id`,`author_id`) CLUSTERED
) DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin;

DROP TABLE IF EXISTS `bookshop`.`ratings`;
CREATE TABLE `bookshop`.`ratings` (
  `book_id` bigint NOT NULL,
  `user_id` bigint NOT NULL,
  `score` tinyint NOT NULL,
```

```
  `rated_at` datetime NOT NULL DEFAULT NOW() ON UPDATE NOW(),
  PRIMARY KEY (`book_id`,`user_id`) CLUSTERED,
  UNIQUE KEY `uniq_book_user_idx` (`book_id`,`user_id`)
) DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin;
ALTER TABLE `bookshop`.`ratings` SET TIFLASH REPLICA 1;

DROP TABLE IF EXISTS `bookshop`.`users`;
CREATE TABLE `bookshop`.`users` (
  `id` bigint AUTO_RANDOM NOT NULL,
  `balance` decimal(15,2) DEFAULT '0.0',
  `nickname` varchar(100) UNIQUE NOT NULL,
  PRIMARY KEY (`id`)
) DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin;

DROP TABLE IF EXISTS `bookshop`.`orders`;
CREATE TABLE `bookshop`.`orders` (
  `id` bigint(20) AUTO_RANDOM NOT NULL,
  `book_id` bigint(20) NOT NULL,
  `user_id` bigint(20) NOT NULL,
  `quality` tinyint(4) NOT NULL,
  `ordered_at` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
  PRIMARY KEY (`id`) CLUSTERED,
  KEY `orders_book_id_idx` (`book_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin
```

4.11.2 规范

4.11.2.1 对象命名规范

用于规范数据库对象的命名，如数据库（DATABASE）、表（TABLE）、索引（INDEX）、用户（USER）等的命名约定。

4.11.2.1.1 原则

- 命名建议使用具有意义的英文词汇，词汇中间以下划线分隔。
- 命名只能使用英文字母、数字、下划线。
- 避免用 TiDB 的保留字如：group，order 等作为单个字段名。
- 建议所有数据库对象使用小写字母。

4.11.2.1.2 数据库命名规范

建议按照业务、产品线或者其它指标进行区分，一般不要超过 20 个字符。如：临时库 (tmp_crm)、测试库 (test_crm)。

4.11.2.1.3 表命名规范

- 同一业务或者模块的表尽可能使用相同的前缀，表名称尽可能表达含义。
- 多个单词以下划线分隔，不推荐超过 32 个字符。
- 建议对表的用途进行注释说明，以便于统一认识。如：
  - 临时表（tmp_t_crm_relation_0425）
  - 备份表（bak_t_crm_relation_20170425）
  - 业务运营临时统计表（`tmp_st_{business code}_{creator abbreviation}_{date}`）
  - 账期归档表（`t_crm_ec_record_YYYY{MM}{dd}`）
- 不同业务模块的表单独建立 DATABASE，并增加相应注释。

4.11.2.1.4 字段命名规范

- 字段命名需要表示其实际含义的英文单词或简写。
- 建议各表之间相同意义的字段应同名。
- 字段也尽量添加注释，枚举型需指明主要值的含义，如”0 - 离线，1 - 在线”。
- 布尔值列命名为 `is_{description}`。如 member 表上表示为 enabled 的会员的列命名为 is_enabled。
- 字段名不建议超过 30 个字符，字段个数不建议大于 60。
- 尽量避免使用保留字，如 order、 from、 desc 等，请参考附录部分的官方保留字。

4.11.2.1.5 索引命名规范

- 主键索引：`pk_{表名称简写}_{字段名简写}`
- 唯一索引：`uk_{表名称简写}_{字段名简写}`
- 普通索引：`idx_{表名称简写}_{字段名简写}`
- 多单词组成的 column_name，取尽可能代表意义的缩写。

4.11.2.2 SQL 开发规范

本章将介绍一些使用 SQL 的一般化开发规范。

4.11.2.2.1 建表删表规范

- 基本原则：表的建立在遵循表命名规范前提下，建议业务应用内部封装建表删表语句增加判断逻辑，防止业务流程异常中断。
- 详细说明：`create table if not exists table_name` 或者 `drop table if exists table_name` 语句建议增加 if 判断，避免应用侧由于 SQL 命令运行异常造成的异常中断。

4.11.2.2.2 SELECT * 使用规范

- 基本原则：避免使用 SELECT * 进行查询。
- 详细说明：按需求选择合适的字段列，避免盲目地 SELECT * 读取全部字段，因为其会消耗网络带宽。考虑将被查询的字段也加入到索引中，以有效利用覆盖索引功能。

4.11.2.2.3 字段上使用函数规范

- 基本原则：在取出字段上可以使用相关函数，但是在 Where 条件中的过滤条件字段上避免使用任何函数，包括数据类型转换函数，以避免索引失效。或者可以考虑使用表达式索引功能。
- 详细说明：

  不推荐的写法：

```
SELECT gmt_create
FROM ...
WHERE DATE_FORMAT(gmt_create, '%Y%m%d %H:%i:%s') = '20090101 00:00:00'
```

  推荐的写法：

```
SELECT DATE_FORMAT(gmt_create, '%Y%m%d %H:%i:%s')
FROM ...
WHERE gmt_create = str_to_date('20090101 00:00:00', '%Y%m%d %H:%i:%s')
```

4.11.2.2.4 其他规范

- WHERE 条件中不要在索引列上进行数学运算或函数运算。
- 用 in/union 替换 or，并注意 in 的个数小于 300。
- 避免使用% 前缀进行模糊前缀查询。
- 如应用使用 Multi Statements 执行 SQL，即将多个 SQL 使用分号连接，一次性地发给客户端执行，TiDB 只会返回第一个 SQL 的执行结果。
- 当使用表达式时，检查其是否支持计算下推到存储层的功能 (TiKV、TiFlash)，否则应有预期在 TiDB 层需要消耗更多内存、甚至 OOM。计算下推到存储层的功能列表如下：
  - TiFlash 支持的计算下推清单。
  - 下推到 TiKV 的表达式列表。
  - 谓词下推。

4.12 云原生开发环境

4.12.1 Gitpod

使用 Gitpod，只需单击一个按钮或链接即可在浏览器中获得完整的开发环境，并且可以立即编写代码。

Gitpod 是一个开源 Kubernetes 应用程序（GitHub 仓库地址 https://github.com/gitpod-io/gitpod），适用于可直接编写代码的开发环境，可为云中的每个任务提供全新的自动化开发环境，非常迅速。此外，Gitpod 能够将你的开发环境描述为代码，并直接从你的浏览器或桌面 IDE 启动即时、远程和基于云的开发环境。

4.12.1.1 快速开始

1. Fork 出 TiDB 应用开发的示例代码仓库 pingcap-inc/tidb-example-java。

2. 通过浏览器的地址栏，在示例代码仓库的 URL 前加上 `https://gitpod.io/#` 来启动你的 gitpod 工作区。

    - 例如，`https://gitpod.io/#https://github.com/pingcap-inc/tidb-example-java`。
    - 支持在 URL 中配置环境变量。例如，`https://gitpod.io/#targetFile=spring-jpa-hibernate_Makefile,targetMode=spring-jpa-hibernate/https://github.com/pingcap-inc/tidb-example-java`。

3. 使用列出的提供商之一登录并启动工作区，例如，`Github`。

4.12.1.2 使用默认的 Gitpod 配置和环境

完成快速开始 的步骤之后，Gitpod 会需要一段时间来设置你的工作区。

以Spring Boot Web 应用程序为例，通过 URL `https://gitpod.io/#targetFile=spring-jpa-hibernate_Makefile,targetMode=spring-jpa-hibernate/https://github.com/pingcap-inc/tidb-example-java` 可以创建一个新工作区。

完成后，你将看到如下所示的页面。

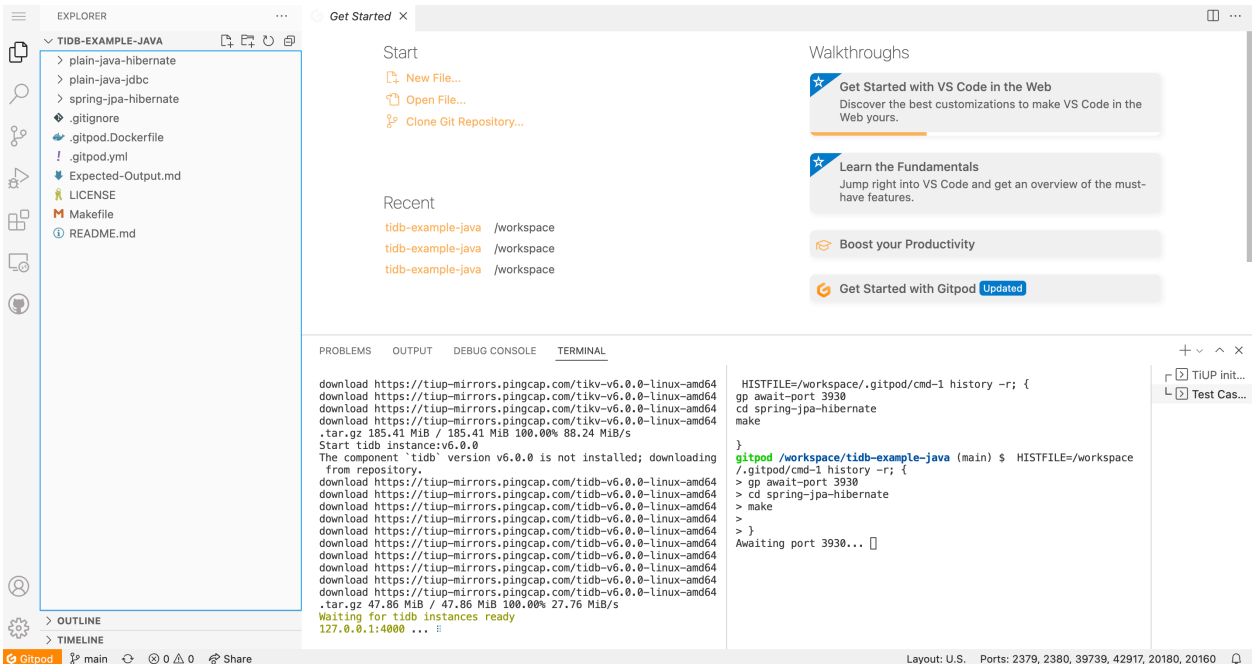


图 18: playground gitpod workspace init

页面中的这个场景使用了 TiUP 来搭建一个 TiDB Playground。你可以在终端的左侧查看进度。

一旦 TiDB Playground 准备就绪，另一个 `Spring JPA Hibernate` 任务将运行。你可以在终端的右侧查看进度。

完成所有任务后，你可以看到如下所示的页面，并在左侧导航栏的 `REMOTE EXPLORER` 中找到你的端口 8080 URL（Gitpod 支持基于 URL 的端口转发）。

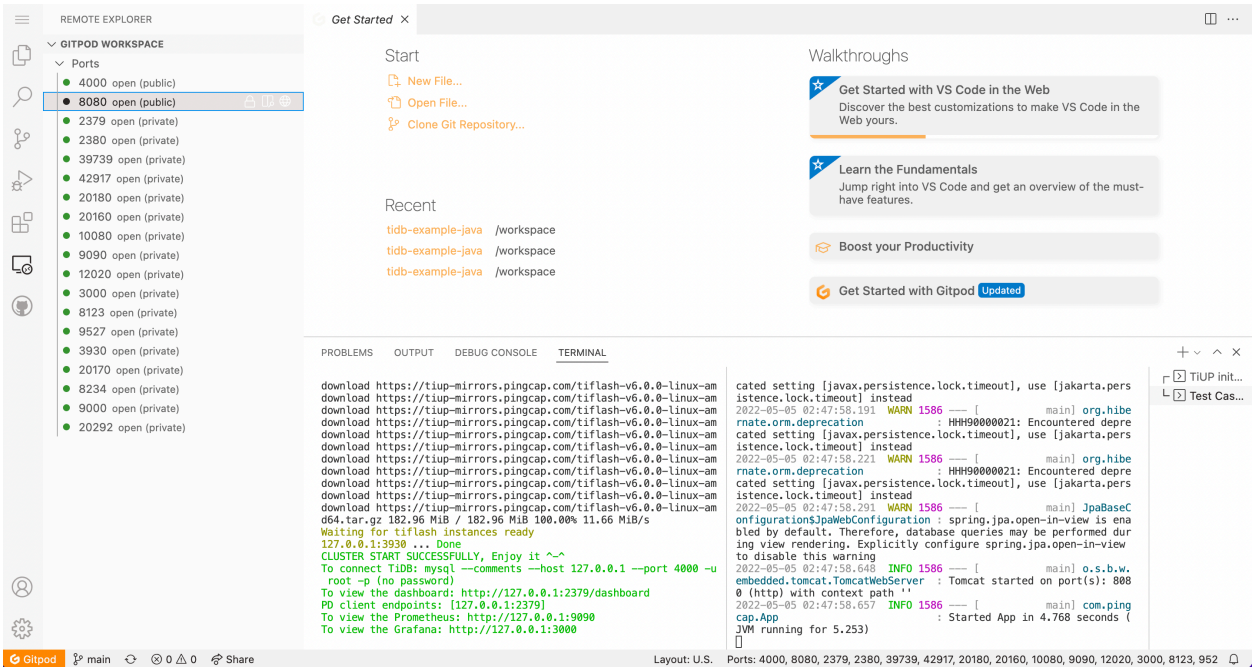


图 19: playground gitpod workspace ready

你可以按照该指南测试 API。注意请将 URL `http://localhost:8080` 替换为你在 `REMOTE EXPLORER` 中找到的那个。

4.12.1.3 使用自定义的 Gitpod 配置和 Docker 镜像

4.12.1.3.1 自定义 Gitpod 配置

在项目的根目录中，参考示例.gitpod.yml，创建一个 `.gitpod.yml` 文件用于配置 Gitpod 工作空间。

```
### This configuration file was automatically generated by Gitpod.
### Please adjust to your needs (see https://www.gitpod.io/docs/config-gitpod-file)
### and commit this file to your remote git repository to share the goodness with others.

### image:
###   file: .gitpod.Dockerfile

tasks:
  - name: Open Target File
    command: |
      if [ -n "$targetFile" ]; then code ${targetFile//[_]//};  fi
  - name: TiUP init playground
    command: |
      $HOME/.tiup/bin/tiup playground
  - name: Test Case
```

```
    openMode: split-right
    init: echo "*** Waiting for TiUP Playground Ready! ***"
    command: |
      gp await-port 3930
      if [ "$targetMode" == "plain-java-jdbc" ]
      then
        cd plain-java-jdbc
        code src/main/resources/dbinit.sql
        code src/main/java/com/pingcap/JDBCExample.java
        make mysql
      elif [ "$targetMode" == "plain-java-hibernate" ]
      then
        cd plain-java-hibernate
        make
      elif [ "$targetMode" == "spring-jpa-hibernate" ]
      then
        cd spring-jpa-hibernate
        make
      fi
ports:
  - port: 8080
    visibility: public
  - port: 4000
    visibility: public
  - port: 2379-36663
    onOpen: ignore
```

4.12.1.3.2 自定义 Gitpod Docker 镜像

默认情况下，Gitpod 使用名为 Workspace-Full 的标准 Docker 镜像作为工作空间的基础。基于此默认镜像启动的工作区预装了 Docker、Go、Java、Node.js、C/C++、Python、Ruby、Rust、PHP 以及 Homebrew、Tailscale、Nginx 等工具。

你可以提供公共 Docker 镜像或 Dockerfile。并为你的项目安装所需的任何依赖项。

这是一个 Dockerfile 示例：示例.gitpod.Dockerfile

```
FROM gitpod/workspace-java-17

RUN sudo apt install mysql-client -y
RUN curl --proto '=https' --tlsv1.2 -sSf https://tiup-mirrors.pingcap.com/install.sh | sh
```

然后需要更新.gitpod.yml：

```
### This configuration file was automatically generated by Gitpod.
### Please adjust to your needs (see https://www.gitpod.io/docs/config-gitpod-file)
### and commit this file to your remote git repository to share the goodness with others.
```

```
image:
  # 在这里导入你的 Dockerfile
  file: .gitpod.Dockerfile

tasks:
  - name: Open Target File
    command: |
      if [ -n "$targetFile" ]; then code ${targetFile//[_]//};  fi
  - name: TiUP init playground
    command: |
      $HOME/.tiup/bin/tiup playground
  - name: Test Case
    openMode: split-right
    init: echo "*** Waiting for TiUP Playground Ready! ***"
    command: |
      gp await-port 3930
      if [ "$targetMode" == "plain-java-jdbc" ]
      then
        cd plain-java-jdbc
        code src/main/resources/dbinit.sql
        code src/main/java/com/pingcap/JDBCExample.java
        make mysql
      elif [ "$targetMode" == "plain-java-hibernate" ]
      then
        cd plain-java-hibernate
        make
      elif [ "$targetMode" == "spring-jpa-hibernate" ]
      then
        cd spring-jpa-hibernate
        make
      fi
ports:
  - port: 8080
    visibility: public
  - port: 4000
    visibility: public
  - port: 2379-36663
    onOpen: ignore
```

4.12.1.3.3 应用更改

完成对 `.gitpod.yml` 文件配置后，请保证最新的代码已在你对应的 GitHub 代码仓库中可用。

访问 `https://gitpod.io/#<YOUR_REPO_URL>` 以建立新的 Gitpod 工作区，新工作区会应用最新的代码。

访问 `https://gitpod.io/workspaces` 以获取所有建立的工作区。

4.12.1.4 总结

Gitpod 提供了完整的、自动化的、预配置的云原生开发环境。无需本地配置，你可以直接在浏览器中开发、运行、测试代码。

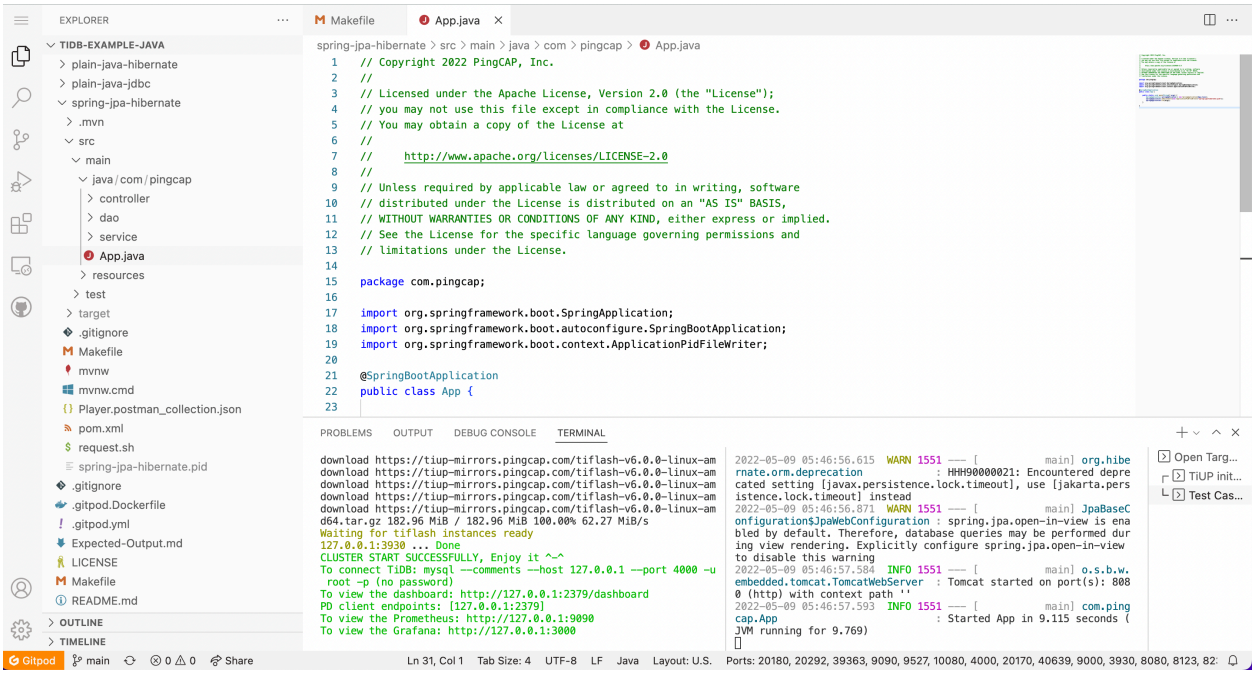


图 20: playground gitpod summary

4.13 第三方工具支持

4.13.1 TiDB 支持的第三方工具

> **注意：**
>
> 本文档仅列举了常见的 TiDB 支持的第三方工具，未被列入其中的第三方工具并非代表不支持，但 PingCAP 无法了解其是否使用到 TiDB 不支持的特性，从而无法保证兼容性。

TiDB 高度兼容 MySQL 协议，使得大部分适配 MySQL 的 Driver、ORM 及其他工具与 TiDB 兼容。本文主要介绍这些工具和它们的支持等级。

4.13.1.1 支持等级

PingCAP 与开源社区合作，通过三方工具提供以下支持：

- Full：表明 PingCAP 已经支持该工具的绝大多数功能兼容性，并且在新版本中对其保持兼容，将定期地对下表中记录的新版本进行兼容性测试。

- Compatible：表明由于该工具已适配 MySQL，而 TiDB 高度兼容 MySQL 协议，因此可以使用此工具的大部分功能。但 PingCAP 并未对该工具作出完整的兼容性验证，有可能出现一些意外的行为。

> **注意：**
>
> 除非明确说明，否则对于支持的 Driver 或者 ORM 框架并不包括应用端事务重试和错误处理。

如果在使用本文列出的工具连接 TiDB 时出现问题，请在 GitHub 上提交包含详细信息的 issue，以帮助在此工具的支持上得到进展。

4.13.1.2 Driver

```
<tr>
   <th>编程语言</th>
   <th>驱动</th>
   <th>最新已测试版本</th>
   <th>支持等级</th>
   <th>TiDB 适配器</th>
   <th>教程</th>
</tr>
```

```
<tr>
   <td>C</td>
   <td><a href="https://dev.mysql.com/doc/c-api/8.0/en/c-api-introduction.html" target="_blank"
       ↪  referrerpolicy="no-referrer-when-downgrade">libmysqlclient</a></td>
   <td>8.0</td>
   <td>Compatible</td>
   <td>N/A</td>
   <td>N/A</td>
</tr>
<tr>
   <td>C#(.Net)</td>
   <td><a href="https://downloads.mysql.com/archives/c-net/" target="_blank" referrerpolicy="no
       ↪ -referrer-when-downgrade">MySQL Connector/NET</a></td>
   <td>8.0</td>
   <td>Compatible</td>
   <td>N/A</td>
   <td>N/A</td>
</tr>
<tr>
   <td>ODBC</td>
   <td><a href="https://downloads.mysql.com/archives/c-odbc/" target="_blank" referrerpolicy="
       ↪ no-referrer-when-downgrade">MySQL Connector/ODBC</a></td>
```

```
    <td>8.0</td>
    <td>Compatible</td>
    <td>N/A</td>
    <td>N/A</td>
</tr>
<tr>
    <td>Go</td>
    <td><a href="https://github.com/go-sql-driver/mysql" target="_blank" referrerpolicy="no-
        ↪ referrer-when-downgrade">go-sql-driver/mysql</a></td>
    <td>v1.6.0</td>
    <td>Full</td>
    <td>N/A</td>
    <td><a href="/zh/tidb/v7.0/dev-guide-sample-application-golang">TiDB 和 Golang 的简单 CRUD
        ↪ 应用程序</a></td>
</tr>
<tr>
    <td>Java</td>
    <td><a href="https://dev.mysql.com/downloads/connector/j/" target="_blank" referrerpolicy="
        ↪ no-referrer-when-downgrade">JDBC</a></td>
    <td>8.0</td>
    <td>Full</td>
    <td>
       <ul>
          <li><a href="/zh/tidb/v7.0/dev-guide-choose-driver-or-orm#java-drivers" data-href="/zh
              ↪ /tidb/v7.0/dev-guide-choose-driver-or-orm#java-drivers">pingcap/mysql-
              ↪ connector-j</a></li>
          <li><a href="/zh/tidb/v7.0/dev-guide-choose-driver-or-orm#tidb-loadbalance" data-href
              ↪ ="/zh/tidb/v7.0/dev-guide-choose-driver-or-orm#tidb-loadbalance">pingcap/tidb-
              ↪ loadbalance</a></li>
       </ul>
    </td>
    <td><a href="/zh/tidb/v7.0/dev-guide-sample-application-java">TiDB 和 Java 的简单 CRUD
        ↪ 应用程序</a></td>
</tr>
<tr>
    <td>JavaScript</td>
    <td><a href="https://github.com/mysqljs/mysql" target="_blank" referrerpolicy="no-referrer-
        ↪ when-downgrade">mysql</a></td>
    <td>v2.18.1</td>
    <td>Compatible</td>
    <td>N/A</td>
    <td>N/A</td>
</tr>
<tr>
    <td>PHP</td>
```

```
    <td><a href="https://dev.mysql.com/downloads/connector/php-mysqlnd/" target="_blank"
        ↪ referrerpolicy="no-referrer-when-downgrade">mysqlnd</a></td>
    <td>PHP 5.4+</td>
    <td>Compatible</td>
    <td>N/A</td>
    <td>N/A</td>
</tr>
<tr>
    <td rowspan="3">Python</td>
    <td><a href="https://dev.mysql.com/doc/connector-python/en/" target="_blank" referrerpolicy
        ↪ ="no-referrer-when-downgrade">mysql-connector-python</a></td>
    <td>8.0</td>
    <td>Compatible</td>
    <td>N/A</td>
    <td><a href="/zh/tidb/v7.0/dev-guide-sample-application-python">TiDB 和 Python 的简单 CRUD
        ↪ 应用程序</a></td>
</tr>
<tr>
    <td><a href="https://mysqlclient.readthedocs.io/" target="_blank" referrerpolicy="no-
        ↪ referrer-when-downgrade">mysqlclient</a></td>
    <td>2.1.1</td>
    <td>Compatible</td>
    <td>N/A</td>
    <td><a href="/zh/tidb/v7.0/dev-guide-sample-application-python">TiDB 和 Python 的简单 CRUD
        ↪ 应用程序</a></td>
</tr>
<tr>
    <td><a href="https://pypi.org/project/PyMySQL/" target="_blank" referrerpolicy="no-referrer-
        ↪ when-downgrade">PyMySQL</a></td>
    <td>1.0.2</td>
    <td>Compatible</td>
    <td>N/A</td>
    <td><a href="/zh/tidb/v7.0/dev-guide-sample-application-python">TiDB 和 Python 的简单 CRUD
        ↪ 应用程序</a></td>
</tr>
```

4.13.1.3 ORM

```
<tr>
    <th>编程语言</th>
    <th>ORM 框架</th>
    <th>最新已测试版本</th>
    <th>支持等级</th>
    <th>TiDB 适配器</th>
    <th>教程</th>
```

```
</tr>
```

```
<tr>
   <td rowspan="5">Go</td>
   <td><a href="https://github.com/go-gorm/gorm" target="_blank" referrerpolicy="no-referrer-
       ↪ when-downgrade">gorm</a></td>
   <td>v1.23.5</td>
   <td>Full</td>
   <td>N/A</td>
   <td><a href="/zh/tidb/v7.0/dev-guide-sample-application-golang">TiDB 和 Golang 的简单 CRUD
       ↪ 应用程序</a></td>
</tr>
<tr>
   <td><a href="https://github.com/beego/beego" target="_blank" referrerpolicy="no-referrer-
       ↪ when-downgrade">beego</a></td>
   <td>v2.0.3</td>
   <td>Full</td>
   <td>N/A</td>
   <td>N/A</td>
</tr>
<tr>
   <td><a href="https://github.com/upper/db" target="_blank" referrerpolicy="no-referrer-when-
       ↪ downgrade">upper/db</a></td>
   <td>v4.5.2</td>
   <td>Full</td>
   <td>N/A</td>
   <td>N/A</td>
</tr>
<tr>
   <td><a href="https://gitea.com/xorm/xorm" target="_blank" referrerpolicy="no-referrer-when-
       ↪ downgrade">xorm</a></td>
   <td>v1.3.1</td>
   <td>Full</td>
   <td>N/A</td>
   <td>N/A</td>
</tr>
<tr>
   <td><a href="https://github.com/ent/ent" target="_blank" referrerpolicy="no-referrer-when-
       ↪ downgrade">ent</a></td>
   <td>v0.11.0</td>
   <td>Compatible</td>
   <td>N/A</td>
   <td>N/A</td>
</tr>
<tr>
```

```
    <td rowspan="4">Java</td>
    <td><a href="https://hibernate.org/orm/" target="_blank" referrerpolicy="no-referrer-when-
        ↪ downgrade">Hibernate</a></td>
    <td>6.1.0.Final</td>
    <td>Full</td>
    <td>N/A</td>
    <td><a href="/zh/tidb/v7.0/dev-guide-sample-application-java">TiDB 和 Java 的简单 CRUD
        ↪ 应用程序</a></td>
</tr>
<tr>
    <td><a href="https://mybatis.org/mybatis-3/" target="_blank" referrerpolicy="no-referrer-
        ↪ when-downgrade">MyBatis</a></td>
    <td>v3.5.10</td>
    <td>Full</td>
    <td>N/A</td>
    <td><a href="/zh/tidb/v7.0/dev-guide-sample-application-java">TiDB 和 Java 的简单 CRUD
        ↪ 应用程序</a></td>
</tr>
<tr>
    <td><a href="https://spring.io/projects/spring-data-jpa/" target="_blank" referrerpolicy="no
        ↪ -referrer-when-downgrade">Spring Data JPA</a></td>
    <td>2.7.2</td>
    <td>Full</td>
    <td>N/A</td>
    <td><a href="/zh/tidb/v7.0/dev-guide-sample-application-spring-boot">使用 Spring Boot 构建
        ↪ TiDB 应用程序</a></td>
</tr>
<tr>
    <td><a href="https://github.com/jOOQ/jOOQ" target="_blank" referrerpolicy="no-referrer-when-
        ↪ downgrade">jOOQ</a></td>
    <td>v3.16.7 (Open Source)</td>
    <td>Full</td>
    <td>N/A</td>
    <td>N/A</td>
</tr>
<tr>
    <td>Ruby</td>
    <td><a href="https://guides.rubyonrails.org/active_record_basics.html" target="_blank"
        ↪ referrerpolicy="no-referrer-when-downgrade">Active Record</a></td>
    <td>v7.0</td>
    <td>Full</td>
    <td>N/A</td>
    <td>N/A</td>
</tr>
<tr>
```

```
    <td rowspan="4">JavaScript / TypeScript</td>
    <td><a href="https://www.npmjs.com/package/sequelize" target="_blank" referrerpolicy="no-
        ↪ referrer-when-downgrade">sequelize</a></td>
    <td>v6.20.1</td>
    <td>Full</td>
    <td>N/A</td>
    <td>N/A</td>
</tr>
<tr>
    <td><a href="https://knexjs.org/" target="_blank" referrerpolicy="no-referrer-when-downgrade
        ↪ ">Knex.js</a></td>
    <td>v1.0.7</td>
    <td>Compatible</td>
    <td>N/A</td>
    <td>N/A</td>
</tr>
<tr>
    <td><a href="https://www.prisma.io/" target="_blank" referrerpolicy="no-referrer-when-
        ↪ downgrade">Prisma Client</a></td>
    <td>4.16.2</td>
    <td>Full</td>
    <td>N/A</td>
    <td>N/A</td>
</tr>
<tr>
    <td><a href="https://www.npmjs.com/package/typeorm" target="_blank" referrerpolicy="no-
        ↪ referrer-when-downgrade">TypeORM</a></td>
    <td>v0.3.6</td>
    <td>Compatible</td>
    <td>N/A</td>
    <td>N/A</td>
</tr>
<tr>
    <td>PHP</td>
    <td><a href="https://laravel.com/" target="_blank" referrerpolicy="no-referrer-when-
        ↪ downgrade">laravel</a></td>
    <td>v9.1.10</td>
    <td>Compatible</td>
    <td><a href="https://github.com/colopl/laravel-tidb" target="_blank" referrerpolicy="no-
        ↪ referrer-when-downgrade">laravel-tidb</a></td>
    <td>N/A</td>
</tr>
<tr>
    <td rowspan="4">Python</td>
```

```
    <td><a href="https://pypi.org/project/Django/" target="_blank" referrerpolicy="no-referrer-
        ↪ when-downgrade">Django</a></td>
    <td>v4.1</td>
    <td>Full</td>
    <td><a href="https://github.com/pingcap/django-tidb" target="_blank" referrerpolicy="no-
        ↪ referrer-when-downgrade">django-tidb</a></td>
    <td><a href="/zh/tidb/v7.0/dev-guide-sample-application-django">TiDB 和 Django 的简单 CRUD
        ↪ 应用程序</a></td>
</tr>
<tr>
    <td><a href="https://www.sqlalchemy.org/" target="_blank" referrerpolicy="no-referrer-when-
        ↪ downgrade">SQLAlchemy</a></td>
    <td>v1.4.37</td>
    <td>Full</td>
    <td>N/A</td>
    <td><a href="/zh/tidb/v7.0/dev-guide-sample-application-python">TiDB 和 Python 的简单 CRUD
        ↪ 应用程序</a></td>
</tr>
<tr>
    <td><a href="https://github.com/coleifer/peewee/" target="_blank" referrerpolicy="no-
        ↪ referrer-when-downgrade">peewee</a></td>
    <td>v3.14.10</td>
    <td>Compatible</td>
    <td>N/A</td>
    <td><a href="/zh/tidb/v7.0/dev-guide-sample-application-python">TiDB 和 Python 的简单 CRUD
        ↪ 应用程序</a></td>
</tr>
```

4.13.1.4 GUI

GUI	最新已测试版本	支持等级	教程
DBeaver	22.1.0	Compatible	N/A
Navicat for MySQL	16.0.14	Compatible	N/A
MySQL Workbench	8.0	Compatible	N/A

```
<tr>
    <th>IDE</th>
    <th>插件</th>
    <th>支持等级</th>
    <th>教程</th>
</tr>
```

```
<tr>
```

```
    <td><a href="https://www.jetbrains.com/datagrip/" target="_blank" referrerpolicy="no-
        ↪ referrer-when-downgrade">DataGrip</a></td>
    <td>N/A</td>
    <td>Compatible</td>
    <td>N/A</td>
</tr>
<tr>
    <td><a href="https://www.jetbrains.com/idea/" target="_blank" referrerpolicy="no-referrer-
        ↪ when-downgrade">IntelliJ IDEA</a></td>
    <td>N/A</td>
    <td>Compatible</td>
    <td>N/A</td>
</tr>
<tr>
    <td rowspan="2"><a href="https://code.visualstudio.com/" target="_blank" referrerpolicy="no-
        ↪ referrer-when-downgrade">Visual Studio Code</a></td>
    <td><a href="https://marketplace.visualstudio.com/items?itemName=dragonly.ticode" target="
        ↪ _blank" referrerpolicy="no-referrer-when-downgrade">TiDE</a></td>
    <td>Compatible</td>
    <td>N/A</td>
</tr>
<tr>
    <td><a href="https://marketplace.visualstudio.com/items?itemName=formulahendry.vscode-mysql"
        ↪  target="_blank" referrerpolicy="no-referrer-when-downgrade">MySQL</a></td>
    <td>Compatible</td>
    <td>N/A</td>
</tr>
```

4.13.2 已知的第三方工具兼容问题

注意：

TiDB 已列举不支持的功能特性，典型的不支持特性有：

- 存储过程与函数
- 触发器
- 事件
- 自定义函数
- 空间类型的函数、数据类型和索引
- XA 语法

这些不支持的功能不兼容将被视为预期行为，不再重复叙述。关于更多 TiDB 与 MySQL 的兼容性对比，你可以查看与 MySQL 兼容性对比。

本文列举的兼容性问题是在一些TiDB 支持的第三方工具中发现的。

4.13.2.1 通用

4.13.2.1.1 TiDB 中 SELECT CONNECTION_ID() 返回结果类型为 64 位整型

描述

TiDB 中 SELECT CONNECTION_ID() 的返回值为 64 位，如 2199023260887。而 MySQL 中返回值为 32 位，如 391650。

规避方法

在 TiDB 应用程序中，请注意使用各语言的 64 位整型类型（或字符串类型）存储 SELECT CONNECTION_ID() 的结果，防止溢出。如 Java 应使用 Long 或 String 进行接收，JavaScript/TypeScript 应使用 string 类型进行接收。

4.13.2.1.2 TiDB 未设置 Com_* 计数器

描述

MySQL 维护了一系列 Com_ 开头的服务端变量来记录你对数据库的操作总数，如 Com_select 记录了 MySQL 数据库从上次启动开始，总共发起的 SELECT 语句数（即使语句并未成功执行）。而 TiDB 并未维护此变量。你可以使用语句SHOW GLOBAL STATUS LIKE 'Com_%' 观察 TiDB 与 MySQL 的差异。

规避方法

请勿使用这样的变量。在 MySQL 中 Com_* 常见的使用场景之一是监控。TiDB 的可观测性较为完善，无需从服务端变量进行查询。如需定制监控工具，可阅读TiDB 监控框架概述来获得更多信息。

4.13.2.1.3 TiDB 错误日志区分 TIMESTAMP 与 DATETIME 类型

描述

TiDB 错误日志区分 TIMESTAMP 与 DATETIME，而 MySQL 不区分，全部返回为 DATETIME。即 MySQL 会将 TIMESTAMP 类型的报错信息错误地写为 DATETIME 类型。

规避方法

请勿使用错误日志进行字符串匹配，要使用错误码进行故障诊断。

4.13.2.1.4 TiDB 不支持 CHECK TABLE 语句

描述

TiDB 不支持 CHECK TABLE 语句。

规避方法

在 TiDB 中使用ADMIN CHECK [TABLE|INDEX] 语句进行表中数据和对应索引的一致性校验。

4.13.2.2 与 MySQL JDBC 的兼容性

测试版本为 MySQL Connector/J 8.0.29。

4.13.2.2.1 默认排序规则不一致

描述

MySQL Connector/J 的排序规则保存在客户端内，通过获取的服务端版本进行判别。

下表列出了已知的客户端与服务端排序规则不一致的字符集：

字符集	客户端默认排序规则	服务端默认排序规则
ascii	ascii_general_ci	ascii_bin
latin1	latin1_swedish_ci	latin1_bin
utf8mb4	utf8mb4_0900_ai_ci	utf8mb4_bin

规避方法

在 TiDB 中手动设置排序规则，不要依赖客户端默认排序规则。客户端默认排序规则由 MySQL Connector/J 配置文件保存。

4.13.2.2.2 参数 NO_BACKSLASH_ESCAPES 不生效

描述

TiDB 中无法使用 NO_BACKSLASH_ESCAPES 参数从而不进行 \ 字符的转义。已提 issue。

规避方法

在 TiDB 中不搭配使用 NO_BACKSLASH_ESCAPES 与 \，而是使用 \\ 编写 SQL 语句。

4.13.2.2.3 未设置索引使用情况参数

描述

TiDB 在通讯协议中未设置 SERVER_QUERY_NO_GOOD_INDEX_USED 与 SERVER_QUERY_NO_INDEX_USED 参数。这将导致以下参数返回与实际不一致：

- com.mysql.cj.protocol.ServerSession.noIndexUsed()
- com.mysql.cj.protocol.ServerSession.noGoodIndexUsed()

规避方法

不使用 noIndexUsed() 与 noGoodIndexUsed() 函数。

4.13.2.2.4 不支持 enablePacketDebug 参数

描述

TiDB 不支持 enablePacketDebug 参数，这是一个 MySQL Connector/J 用于调试的参数，将保留数据包的 Buffer。这将导致连接的意外关闭，请勿打开。

规避方法

不设置 enablePacketDebug 参数。

4.13.2.2.5 不支持 UpdatableResultSet

描述

TiDB 暂不支持 `UpdatableResultSet`，即请勿指定 `ResultSet.CONCUR_UPDATABLE` 参数，也不要在 `ResultSet` 内部进行数据更新。

规避方法

使用 `UPDATE` 语句进行数据更新，可使用事务保证数据一致性。

4.13.2.3 MySQL JDBC Bug

4.13.2.3.1 `useLocalTransactionState` 和 `rewriteBatchedStatements` 同时开启将导致事务无法提交或回滚

描述

`useLocalTransactionState` 和 `rewriteBatchedStatements` 两参数同时开启时，将导致事务无法提交。你可以使用代码复现。

规避方法

> **注意：**
>
> 已向 MySQL JDBC 报告此 Bug，可关注此 Bug Report 进行最新消息的跟踪。

请勿开启 `useLocalTransactionState`，这有可能导致事务无法提交或回滚。

4.13.2.3.2 Connector 无法兼容 5.7.5 版本以下的服务端

描述

MySQL Connector/J 8.0.29 在与 5.7.5 版本以下的 MySQL 服务端，或使用 5.7.5 版本以下 MySQL 服务端协议的数据库（如 TiDB 6.3.0 版本以下）同时使用时，将在某些情况下导致数据库连接的挂起。关于更多细节信息，可查看此 Bug Report。

规避方法

这是一个已知的问题，截至 2022 年 10 月 12 日，MySQL Connector/J 未合并修复代码。

TiDB 对其进行了两个维度的修复：

- 客户端方面：pingcap/mysql-connector-j 中修复了该 Bug，你可以使用 pingcap/mysql-connector-j 替换官方的 MySQL Connector/J。
- 服务端方面：TiDB v6.3.0 修复了此兼容性问题，你可以升级服务端至 v6.3.0 或以上版本。

4.13.2.4 与 Sequelize 的兼容性

本小节描述的兼容性信息基于 Sequelize v6.32.1 测试。

根据测试结果，TiDB 支持绝大部分 Sequelize 功能（使用 MySQL 作为方言），不支持的功能有：

- 不支持 GEOMETRY 相关。
- 不支持修改整数主键。
- 不支持 PROCEDURE 相关。
- 不支持 READ-UNCOMMITTED 和 SERIALIZABLE 隔离级别。
- 默认不允许修改列的 AUTO_INCREMENT 属性。
- 不支持 FULLTEXT、HASH 和 SPATIAL 索引。
- 不支持 `sequelize.queryInterface.showIndex(Model.tableName);`。
- 不支持 `sequelize.options.databaseVersion`。
- 不支持使用 `queryInterface.addColumn` 添加外键引用。

4.13.2.4.1 不支持修改整数主键

描述

不支持修改整数类型的主键，这是由于当主键为整数类型时，TiDB 使用其作为数据组织的索引。你可以在此 Issue 或聚簇索引一节中获取更多信息。

4.13.2.4.2 不支持 READ-UNCOMMITTED 和 SERIALIZABLE 隔离级别

描述

TiDB 不支持 READ-UNCOMMITTED 和 SERIALIZABLE 隔离级别。设置事务隔离级别为 READ-UNCOMMITTED 或 SERIALIZABLE 时将报错。

规避方法

仅使用 TiDB 支持的 REPEATABLE-READ 或 READ-COMMITTED 隔离级别。

如果你的目的是兼容其他设置 SERIALIZABLE 隔离级别的应用，但不依赖于 SERIALIZABLE，你可以设置`tidb_skip_isolation_level_check` 为 1，此后如果对 `tx_isolation`（`transaction_isolation` 别名）赋值一个 TiDB 不支持的隔离级别（READ-UNCOMMITTED 和 SERIALIZABLE），不会报错。

4.13.2.4.3 默认不允许修改列的 AUTO_INCREMENT 属性

描述

默认不允许通过 ALTER TABLE MODIFY 或 ALTER TABLE CHANGE 来增加或移除某个列的 AUTO_INCREMENT 属性。

规避方法

参考AUTO_INCREMENT 的使用限制。

设置 `@@tidb_allow_remove_auto_inc` 为 `true`，即可允许移除 AUTO_INCREMENT 属性。

4.13.2.4.4 不支持 FULLTEXT、HASH 和 SPATIAL 索引

描述

TiDB 不支持 FULLTEXT、HASH 和 SPATIAL 索引。

4.13.3 TiDB 与 ProxySQL 集成

本文以 CentOS 7 为例，简单介绍 TiDB 与 ProxySQL 的集成方法。如果你有其他系统的集成需求，可参考快速体验使用 Docker 及 Docker Compose 部署测试集成环境。你也可以参考以下链接，以获得更多信息：

- TiDB 文档
- TiDB 应用开发文档
- ProxySQL 官方文档
- TiDB 与 ProxySQL 的集成测试

4.13.3.1 1. 启动 TiDB

4.13.3.1.1 测试环境

请参考使用 TiDB Serverless 构建 TiDB 集群。

1. 下载 TiDB 源码，进入 `tidb-server` 目录后，使用 `go build` 进行编译。

```
git clone git@github.com:pingcap/tidb.git
cd tidb/tidb-server
go build
```

2. 使用配置文件 `tidb-config.toml` 来启动 TiDB，命令如下所示：

```
${TIDB_SERVER_PATH} -config ./tidb-config.toml -store unistore -path "" -lease 0s > ${
    ↪ LOCAL_TIDB_LOG} 2>&1 &
```

> 注意：
>
> - 此处使用 `unistore` 作为存储引擎，这是 TiDB 的测试存储引擎，请仅在测试时使用它。
> - `TIDB_SERVER_PATH`：上一步中使用 `go build` 编译的二进制文件位置，如你在 `/usr/local` 下进行上一步操作，那么此处的 `TIDB_SERVER_PATH` 应为 `/usr/local/tidb/tidb-server/tidb-server`。
> - `LOCAL_TIDB_LOG`：输出 TiDB 日志的位置。

TiUP 在 TiDB 中承担着包管理器的角色，管理着 TiDB 生态下众多的组件，如 TiDB、PD、TiKV 等。

1. 安装 TiUP

```
curl --proto '=https' --tlsv1.2 -sSf https://tiup-mirrors.pingcap.com/install.sh | sh
```

2. 启动测试环境 TiDB

```
tiup playground
```

4.13.3.1.2 正式环境

在需要托管 TiDB 服务的前提下（如无法自行运维、需要云原生环境等），建议直接使用 TiDB Cloud。你可以参考 TiDB Cloud 的 Create a TiDB Cluster 在 TiDB Cloud 中部署正式环境下的 TiDB。

正式环境相对测试环境会复杂许多，建议参考使用 TiUP 部署 TiDB 集群并根据硬件条件部署。

4.13.3.2 2. 启动 ProxySQL

4.13.3.2.1 yum 安装

1. 添加 RPM 仓库：

```
cat > /etc/yum.repos.d/proxysql.repo << EOF
[proxysql]
name=ProxySQL YUM repository
baseurl=https://repo.proxysql.com/ProxySQL/proxysql-2.4.x/centos/\$releasever
gpgcheck=1
gpgkey=https://repo.proxysql.com/ProxySQL/proxysql-2.4.x/repo_pub_key
EOF
```

2. 安装：

```
yum install proxysql
```

3. 启动：

```
systemctl start proxysql
```

4.13.3.2.2 其他安装方式

参考 ProxySQL 的 Github 页面或 ProxySQL 官方文档进行安装。

4.13.3.3 3. 配置 ProxySQL

需要将 ProxySQL 内的配置指向 TiDB，以此将 ProxySQL 作为 TiDB 的代理。下面列举必需的配置项，其余配置项可参考 ProxySQL 官方文档。

4.13.3.3.1 ProxySQL 配置简介

ProxySQL 使用一个单独的端口进行配置管理，另一个端口进行代理。其中，配置管理的入口称为 ProxySQL Admin interface，代理的入口称为 ProxySQL MySQL Interface。

- ProxySQL Admin interface：可以使用具有 `admin` 权限的用户连接到管理界面，以读取和写入配置，或者使用具有 `stats` 权限的用户，只能读取某些统计数据（不读取或写入配置）。默认凭证是 `admin:admin` 和 `stats:stats`，但出于安全考虑，可以使用默认凭证进行本地连接。要远程连接，需要配置一个新的用户，通常它被命名为 `radmin`。

- ProxySQL MySQL Interface：用于代理，将 SQL 转发到配置的服务中。

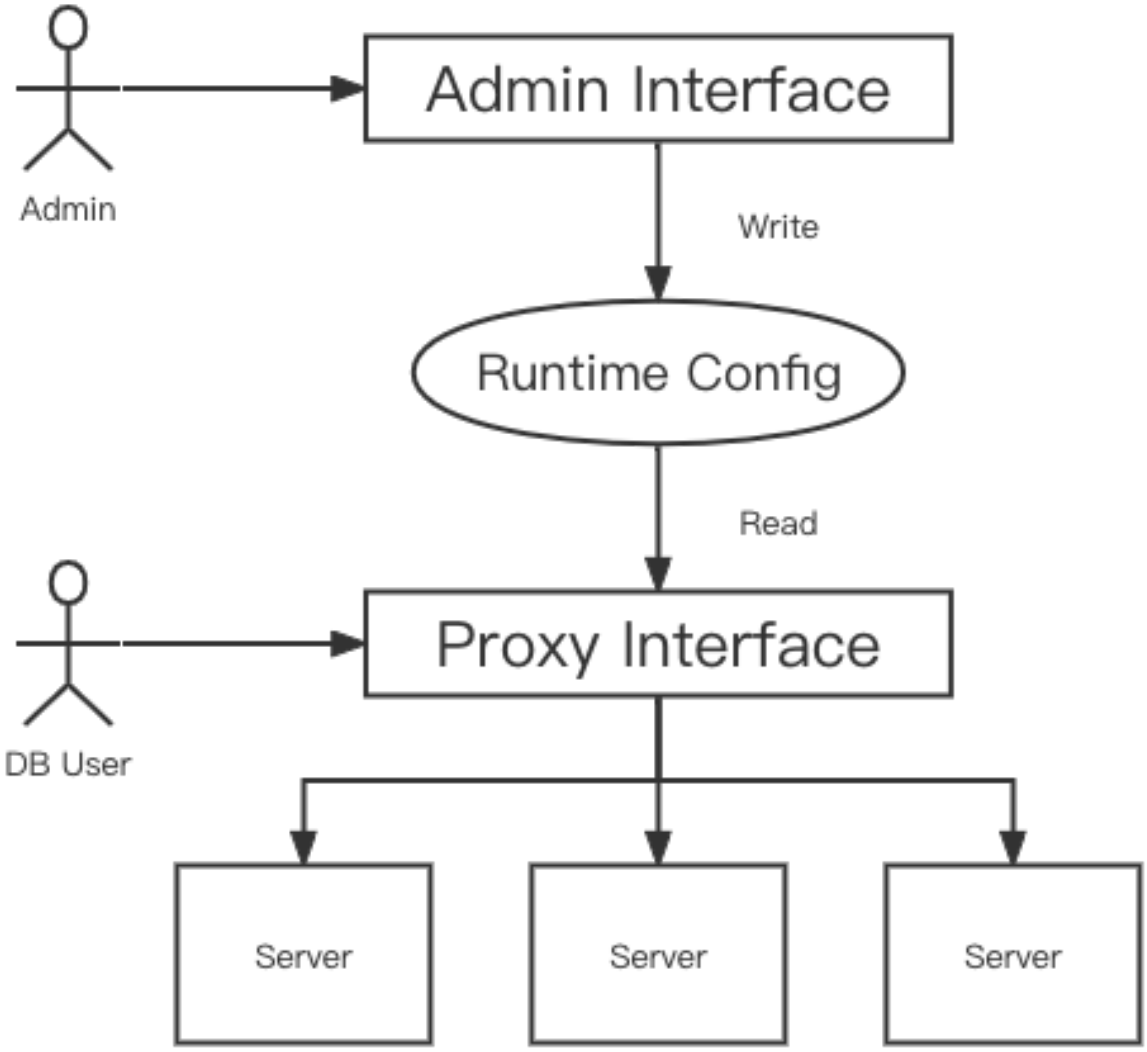


图 21: proxysql config flow

ProxySQL 有三层配置：runtime、memory、disk。你仅能更改 memory 层的配置。在更改配置后，可以使用 LOAD xxx TO runtime 来生效这个配置，也可以使用 SAVE xxx TO DISK 落盘，防止配置丢失。

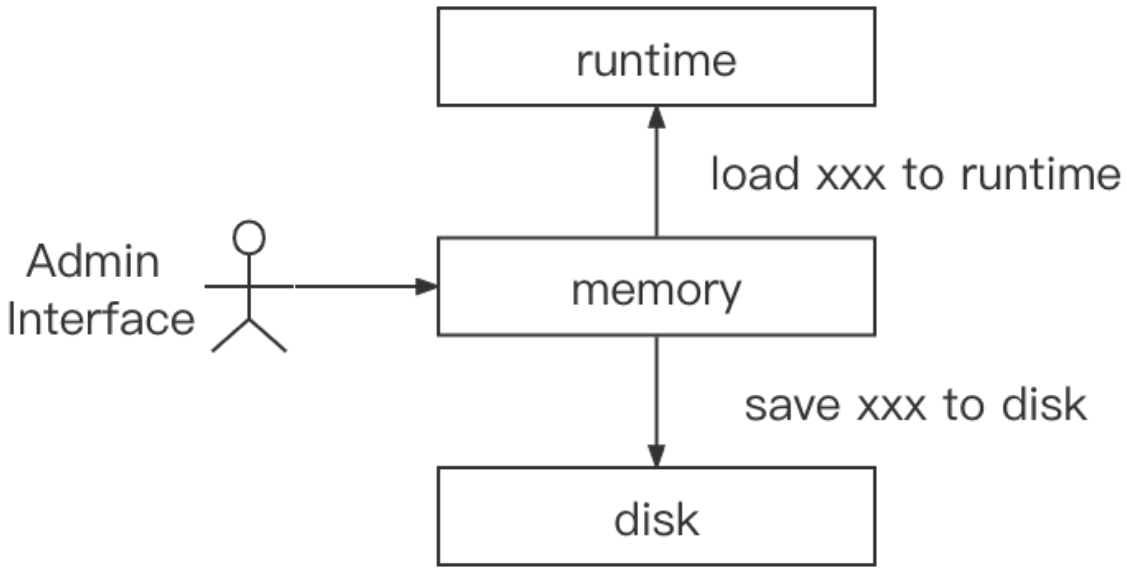


图 22: proxysql config layer

4.13.3.3.2 配置 TiDB 后端

在 ProxySQL 中添加 TiDB 后端，此处如果有多个 TiDB 后端，可以添加多条。请在 ProxySQL Admin interface 进行此操作：

```
INSERT INTO mysql_servers(hostgroup_id, hostname, port) VALUES (0, '127.0.0.1', 4000);
LOAD mysql servers TO runtime;
SAVE mysql servers TO DISK;
```

字段解释：

- `hostgroup_id`：ProxySQL 是以 hostgroup 为单位管理后端服务的，可以将需要负载均衡的几个服务配置为同一个 hostgroup，这样 ProxySQL 将均匀地分发 SQL 到这些服务上。而在需要区分不同后端服务时（如读写分离场景等），可将其配置为不同的 hostgroup，以此配置不同的代理条件。
- `hostname`：后端服务的 IP 或域名。
- `port`：后端服务的端口。

4.13.3.3.3 配置 Proxy 登录账号

在 ProxySQL 中添加 TiDB 后端的登录账号。ProxySQL 将允许此账号来登录 ProxySQL MySQL Interface，而且 ProxySQL 将以此创建与 TiDB 之间的连接，因此，请确保此账号在 TiDB 中拥有相应权限。请在 ProxySQL Admin interface 进行此操作：

```
INSERT INTO mysql_users(username, password, active, default_hostgroup, transaction_persistent)
    ↪ VALUES ('root', '', 1, 0, 1);
LOAD mysql users TO runtime;
SAVE mysql users TO DISK;
```

字段解释：

- username：用户名。
- password：密码。
- active：是否生效。1 为生效，0 为不生效，仅 active = 1 的用户可登录。
- default_hostgroup：此账号默认使用的 hostgroup，SQL 将被发送至此 hostgroup 中，除非查询规则将流量发送到不同的 hostgroup。
- transaction_persistent：值为 1 时，表示事务持久化，即：当该用户在连接中开启了一个事务后，那么在事务提交或回滚之前，所有的语句都路由到同一个 hostgroup 中，避免语句分散到不同 hostgroup。

4.13.3.3.4 配置文件配置

除了使用 ProxySQL Admin interface 配置，也可以使用配置文件进行配置。ProxySQL 文档中，配置文件仅应该被视为是一种辅助初始化的方式，而并非主要配置的手段。配置文件仅在 SQLite 数据库未被创建时读取，后续将不会继续读取配置文件。因此，使用配置文件配置时，你应进行 SQLite 数据库的删除，这将丢失你在 ProxySQL Admin interface 中对配置进行的更改：

```
rm /var/lib/proxysql/proxysql.db
```

另外，也可以运行 LOAD xxx FROM CONFIG，用配置文件中的配置覆盖当前内存中的配置。

配置文件的位置为 /etc/proxysql.cnf，我们将上方的必需配置转换为配置文件方式，仅更改 mysql_servers、mysql_users 这两个配置节点，其余配置可自行查看 /etc/proxysql.cnf：

```
mysql_servers =
(
    {
        address="127.0.0.1"
        port=4000
        hostgroup=0
        max_connections=2000
    }
)

mysql_users:
(
    {
        username = "root"
        password = ""
        default_hostgroup = 0
        max_connections = 1000
        default_schema = "test"
        active = 1
        transaction_persistent = 1
    }
)
```

随后使用 `systemctl restart proxysql` 进行服务重启后即可生效，配置生效后将自动创建 SQLite 数据库，后续将不会再次读取配置文件。

4.13.3.3.5 其余配置项

仅以上配置为必需配置项，其余配置项并非必需。你可在 ProxySQL Global Variables 中获取全部配置项的名称及作用。

4.13.3.4 4. 快速体验

在测试环境中，你可以使用 Docker 及 Docker Compose 快速进行集成后的环境体验，请确认 4000、 6033 端口未被占用，然后执行如下命令：

```
git clone https://github.com/Icemap/tidb-proxysql-integration-test.git
cd tidb-proxysql-integration-test && docker-compose pull # Get the latest Docker images
sudo setenforce 0 # Only on Linux
docker-compose up -d
```

> **警告：**
>
> 请勿在生产环境使用此快速体验方式创建集成环境。

这样就已经完成了一个集成了 TiDB 与 ProxySQL 环境的启动，这将启动两个容器。你可以使用用户名为 `root`，密码为空的账号，登录到本机的 6033 端口 (ProxySQL)。容器具体配置可见 `docker-compose.yaml`，ProxySQL 具体配置可见 proxysql-docker.cnf。

运行如下命令：

```
mysql -u root -h 127.0.0.1 -P 6033 -e "SELECT VERSION()"
```

运行结果：

```
+--------------------+
| VERSION()          |
+--------------------+
| 5.7.25-TiDB-v6.1.0 |
+--------------------+
```

4.13.3.5 5. 配置示例

配置示例的前提条件：

- Docker
- Docker Compose
- MySQL Client

下载示例源码并进入目录：

```
git clone https://github.com/Icemap/tidb-proxysql-integration-test.git
cd tidb-proxysql-integration-test
```

下面的示例均以 `tidb-proxysql-integration-test` 目录做为根目录。

4.13.3.5.1 使用 Admin Interface 配置负载均衡

进入本示例目录：

```
cd example/proxy-rule-admin-interface
```

脚本运行

以 ProxySQL Admin Interface 为配置入口，配置负载均衡场景为例。可使用以下命令运行脚本：

```
./test-load-balance.sh
```

逐步运行

1. 通过 Docker Compose 启动三个 TiDB 容器实例，容器内部端口均为 4000，映射宿主机端口为 4001、4002、4003。TiDB 容器实例启动后，再启动一个 ProxySQL 实例，容器内部 ProxySQL MySQL Interface 端口为 6033，映射宿主机端口为 6034。不暴露 ProxySQL Admin Interface 端口，因为其仅可在本地（即容器内）登录 ProxySQL Admin Interface。此流程被写在 docker-compose.yaml 中。

   ```
   docker-compose up -d
   ```

2. 在 3 个 TiDB 实例内，创建相同的表结构，但写入不同的数据：'tidb-0'、'tidb-1'、'tidb-2'，以便分辨不同的数据库实例：

   ```
   mysql -u root -h 127.0.0.1 -P 4001 << EOF
   DROP TABLE IF EXISTS test.test;
   CREATE TABLE test.test (db VARCHAR(255));
   INSERT INTO test.test (db) VALUES ('tidb-0');
   EOF

   mysql -u root -h 127.0.0.1 -P 4002 << EOF
   DROP TABLE IF EXISTS test.test;
   CREATE TABLE test.test (db VARCHAR(255));
   INSERT INTO test.test (db) VALUES ('tidb-1');
   EOF

   mysql -u root -h 127.0.0.1 -P 4003 << EOF
   DROP TABLE IF EXISTS test.test;
   CREATE TABLE test.test (db VARCHAR(255));
   INSERT INTO test.test (db) VALUES ('tidb-2');
   EOF
   ```

3. 使用 `docker-compose exec` 命令，在 ProxySQL Admin Interface 中运行事先准备好的配置 ProxySQL 的 SQL 文件：

```
docker-compose exec proxysql sh -c "mysql -uadmin -padmin -h127.0.0.1 -P6032 < ./proxysql-
    ↪ prepare.sql"
```

此 SQL 文件将会运行：

1. 添加 3 个 TiDB 后端的地址，并且 `hostgroup_id` 均为 0。
2. 令 TiDB 后端配置生效，并落盘保存。
3. 添加用户 `root`，密码为空，`default_hostgroup` 为 0，对应上方的 TiDB 后端 `hostgroup_id`。
4. 生效用户配置，并落盘保存。

4. 使用 `root` 用户登录 ProxySQL MySQL Interface，连续查询 5 次数据，预期结果将有 `'tidb-0'`、`'tidb-1'`、`'tidb-2'` 三种不同的返回。

```
mysql -u root -h 127.0.0.1 -P 6034 -t << EOF
SELECT * FROM test.test;
SELECT * FROM test.test;
SELECT * FROM test.test;
SELECT * FROM test.test;
SELECT * FROM test.test;
EOF
```

5. 停止并清除 Docker Compose 启动的容器、网络拓扑等资源。

```
docker-compose down
```

预期输出

因为负载均衡的原因，预期输出将有 `'tidb-0'`、`'tidb-1'`、`'tidb-2'` 三种不同的返回。但具体顺序未知。其中一种预期输出为：

```
### ./test-load-balance.sh
Creating network "load-balance-admin-interface_default" with the default driver
Creating load-balance-admin-interface_tidb-1_1 ... done
Creating load-balance-admin-interface_tidb-2_1 ... done
Creating load-balance-admin-interface_tidb-0_1 ... done
Creating load-balance-admin-interface_proxysql_1 ... done
+--------+
| db     |
+--------+
| tidb-2 |
+--------+
+--------+
| db     |
+--------+
| tidb-0 |
```

```
+--------+
+--------+
| db     |
+--------+
| tidb-1 |
+--------+
+--------+
| db     |
+--------+
| tidb-1 |
+--------+
+--------+
| db     |
+--------+
| tidb-1 |
+--------+
Stopping load-balance-admin-interface_proxysql_1 ... done
Stopping load-balance-admin-interface_tidb-0_1   ... done
Stopping load-balance-admin-interface_tidb-2_1   ... done
Stopping load-balance-admin-interface_tidb-1_1   ... done
Removing load-balance-admin-interface_proxysql_1 ... done
Removing load-balance-admin-interface_tidb-0_1   ... done
Removing load-balance-admin-interface_tidb-2_1   ... done
Removing load-balance-admin-interface_tidb-1_1   ... done
Removing network load-balance-admin-interface_default
```

4.13.3.5.2 使用 Admin Interface 配置用户分离

进入本示例目录：

```
cd example/proxy-rule-admin-interface
```

脚本运行

以 ProxySQL Admin Interface 为配置入口，配置负载均衡配置用户分离场景为例，不同用户将使用不同的 TiDB 后端。可使用以下命令运行脚本：

```
./test-user-split.sh
```

逐步运行

1. 通过 Docker Compose 启动两个 TiDB 容器实例，容器内部端口均为 4000，映射宿主机端口为 4001、4002。TiDB 实例启动后，再启动一个 ProxySQL 实例，容器内部 ProxySQL MySQL Interface 端口为 6033，映射宿主机端口为 6034。不暴露 ProxySQL Admin Interface 端口，因为其仅可在本地（即容器内）登录 ProxySQL Admin Interface。此流程被写在 docker-compose.yaml 中。

   ```
   docker-compose up -d
   ```

2. 在 2 个 TiDB 实例内，创建相同的表结构，但写入不同的数据'tidb-0'、'tidb-1'，以便分辨不同的数据库实例：

```
mysql -u root -h 127.0.0.1 -P 4001 << EOF
DROP TABLE IF EXISTS test.test;
CREATE TABLE test.test (db VARCHAR(255));
INSERT INTO test.test (db) VALUES ('tidb-0');
EOF

mysql -u root -h 127.0.0.1 -P 4002 << EOF
DROP TABLE IF EXISTS test.test;
CREATE TABLE test.test (db VARCHAR(255));
INSERT INTO test.test (db) VALUES ('tidb-1');
EOF
```

3. 为 ProxySQL 在 tidb-1 实例中新建一个用户：

```
mysql -u root -h 127.0.0.1 -P 4002 << EOF
CREATE USER 'root1' IDENTIFIED BY '';
GRANT ALL PRIVILEGES ON *.* TO 'root1'@'%';
FLUSH PRIVILEGES;
EOF
```

4. 使用 docker-compose exec 命令，在 ProxySQL Admin Interface 中运行事先准备好的配置 ProxySQL 的 SQL 文件：

```
docker-compose exec proxysql sh -c "mysql -uadmin -padmin -h127.0.0.1 -P6032 < ./proxysql-
    ↪ prepare.sql"
```

   此 SQL 文件将会运行：

   1. 添加 2 个 TiDB 后端的地址，其中，tidb-0 的 hostgroup_id 为 0，tidb-1 的 hostgroup_id 为 1。
   2. 生效 TiDB 后端配置，并落盘保存。
   3. 添加用户 root，密码为空，default_hostgroup 为 0，即默认将路由至 tidb-0。
   4. 添加用户 root1，密码为空，default_hostgroup 为 1，即默认将路由至 tidb-1。
   5. 生效用户配置，并落盘保存。

5. 分别使用 root 用户及 root1 用户登录 ProxySQL MySQL Interface，预期结果将为 'tidb-0'、'tidb-1'。

```
mysql -u root -h 127.0.0.1 -P 6034 -e "SELECT * FROM test.test;"
mysql -u root1 -h 127.0.0.1 -P 6034 -e "SELECT * FROM test.test;"
```

6. 停止并清除 Docker Compose 启动的容器、网络拓扑等资源。

```
docker-compose down
```

预期输出

```
### ./test-user-split.sh
Creating network "user-split-admin-interface_default" with the default driver
Creating user-split-admin-interface_tidb-1_1 ... done
Creating user-split-admin-interface_tidb-0_1 ... done
Creating user-split-admin-interface_proxysql_1 ... done
+--------+
| db     |
+--------+
| tidb-0 |
+--------+
+--------+
| db     |
+--------+
| tidb-1 |
+--------+
Stopping user-split-admin-interface_proxysql_1 ... done
Stopping user-split-admin-interface_tidb-0_1   ... done
Stopping user-split-admin-interface_tidb-1_1   ... done
Removing user-split-admin-interface_proxysql_1 ... done
Removing user-split-admin-interface_tidb-0_1   ... done
Removing user-split-admin-interface_tidb-1_1   ... done
Removing network user-split-admin-interface_default
```

4.13.3.5.3 使用 Admin Interface 配置代理规则

进入本示例目录：

```
cd example/proxy-rule-admin-interface
```

脚本运行

以 ProxySQL Admin Interface 为配置入口，代理规则场景中，常见的读写分离配置为例，将使用规则匹配将要运行的 SQL，从而将读、写 SQL 转发至不同的 TiDB 后端（若均未匹配，则使用用户的 `default_hostgroup`）。可使用以下命令运行脚本：

```
./proxy-rule-split.sh
```

逐步运行

1. 通过 Docker Compose 启动两个 TiDB 容器实例，容器内部端口均为 4000，映射宿主机端口为 4001、4002。TiDB 实例启动后，再启动一个 ProxySQL 实例，容器内部 ProxySQL MySQL Interface 端口为 6033，映射宿主机端口为 6034。不暴露 ProxySQL Admin Interface 端口，因为其仅可在本地（即容器内）登录 ProxySQL Admin Interface。此流程被写在 `docker-compose.yaml` 中。

   ```
   docker-compose up -d
   ```

2. 在 2 个 TiDB 实例内，创建相同的表结构，但写入不同的数据 'tidb-0'、'tidb-1'，以便分辨不同的数据库实例。此处展示向其中一个 TiDB 实例写入数据的命令，另一实例同理：

```
mysql -u root -h 127.0.0.1 -P 4001 << EOF
DROP TABLE IF EXISTS test.test;
CREATE TABLE test.test (db VARCHAR(255));
INSERT INTO test.test (db) VALUES ('tidb-0');
EOF

mysql -u root -h 127.0.0.1 -P 4002 << EOF
DROP TABLE IF EXISTS test.test;
CREATE TABLE test.test (db VARCHAR(255));
INSERT INTO test.test (db) VALUES ('tidb-1');
EOF
```

3. 使用 `docker-compose exec` 命令，在 ProxySQL Admin Interface 中运行事先准备好的配置 ProxySQL 的 SQL 文件：

```
docker-compose exec proxysql sh -c "mysql -uadmin -padmin -h127.0.0.1 -P6032 < ./proxysql-
    ↪ prepare.sql"
```

此 SQL 文件将会运行：

1. 添加 2 个 TiDB 后端的地址，其中，`tidb-0` 的 `hostgroup_id` 为 `0`，`tidb-1` 的 `hostgroup_id` 为 `1`。
2. 生效 TiDB 后端配置，并落盘保存。
3. 添加用户 `root`，密码为空，`default_hostgroup` 为 `0`，即默认将路由至 `tidb-0`。
4. 生效用户配置，并落盘保存。
5. 添加规则 `^SELECT.*FOR UPDATE$`，`rule_id` 为 `1`，`destination_hostgroup` 为 `0`，即匹配此规则的 SQL 语句将被转发至 `hostgroup` 为 `0` 的 TiDB 中（这条规则是为了将 `SELECT ... FOR UPDATE` 语句转发至写的数据库中）。
6. 添加规则 `^SELECT`，`rule_id` 为 `2`，`destination_hostgroup` 为 `1`，即匹配此规则的 SQL 语句将被转发至 `hostgroup` 为 `1` 的 TiDB 中。
7. 生效规则配置，并落盘保存。

> 注意：
>
> 关于匹配规则：
>
> - ProxySQL 将按照 `rule_id` 从小到大的顺序逐一尝试匹配规则。
> - `^` 匹配 SQL 语句的开头，`$` 匹配 SQL 语句的结尾。
> - 此处使用的 `match_digest` 进行匹配，用于匹配参数化后的 SQL 语句，语法见 query_processor_regex。
> - 重要参数：
>   - `digest`：用于匹配参数化后的 Hash 值。
>   - `match_pattern`：用于匹配原始 SQL 语句。
>   - `negate_match_pattern`：设置为 `1` 时，对 `match_digest` 或 `match_pattern` 匹配取反。

  - `log`：将记录查询日志。
  - `replace_pattern`：将匹配到的内容，替换为此字段的值，如为空，则不做替换。
- 完整参数，请见 mysql_query_rules。

4. 使用 `root` 用户登录 ProxySQL MySQL Interface：

```
mysql -u root -h 127.0.0.1 -P 6034
```

登录后可运行以下语句：

- SELECT 语句：

```
SELECT * FROM test.test;
```

预计匹配 `rule_id` 为 2 的规则，从而转发至 `hostgroup` 为 1 的 TiDB 后端 `tidb-1` 中。

- SELECT ... FOR UPDATE 语句：

```
SELECT * FROM test.test for UPDATE;
```

预计匹配 `rule_id` 为 1 的规则，从而转发至 `hostgroup` 为 0 的 TiDB 后端 `tidb-0` 中。

- 事务语句：

```
BEGIN;
INSERT INTO test.test (db) VALUES ('insert this and rollback later');
SELECT * FROM test.test;
ROLLBACK;
```

`BEGIN` 语句预计不会匹配所有规则，因此将使用用户的 `default_hostgroup`（为 0），从而转发至 `hostgroup` 为 0 的 TiDB 后端 `tidb-0` 中。而因为 ProxySQL 默认开启用户的 `transaction_persistent` ↪，这将使同一个事务内的所有语句运行在同一个 `hostgroup` 中，因此，这里的 `INSERT` 语句和 `SELECT * FROM test.test;` 也将转发至 `hostgroup` 为 0 的 TiDB 后端 `tidb-0` 中。

5. 停止并清除 Docker Compose 启动的容器、网络拓扑等资源。

```
docker-compose down
```

预期输出

```
### ./proxy-rule-split.sh
Creating network "proxy-rule-admin-interface_default" with the default driver
Creating proxy-rule-admin-interface_tidb-1_1 ... done
Creating proxy-rule-admin-interface_tidb-0_1 ... done
Creating proxy-rule-admin-interface_proxysql_1 ... done
+--------+
| db     |
+--------+
| tidb-1 |
```

```
+--------+
+--------+
| db     |
+--------+
| tidb-0 |
+--------+
+--------------------------------+
| db                             |
+--------------------------------+
| tidb-0                         |
| insert this and rollback later |
+--------------------------------+
Stopping proxy-rule-admin-interface_proxysql_1 ... done
Stopping proxy-rule-admin-interface_tidb-0_1   ... done
Stopping proxy-rule-admin-interface_tidb-1_1   ... done
Removing proxy-rule-admin-interface_proxysql_1 ... done
Removing proxy-rule-admin-interface_tidb-0_1   ... done
Removing proxy-rule-admin-interface_tidb-1_1   ... done
Removing network proxy-rule-admin-interface_default
```

4.13.3.5.4 使用配置文件配置负载均衡

以配置文件为配置入口，配置负载均衡场景为例，运行如下命令：

```
cd example/load-balance-config-file
./test-load-balance.sh
```

此配置实现效果与使用 Admin Interface 配置负载均衡完全一致，仅改为使用配置文件进行 ProxySQL 初始化配置。

> 注意：
>
> - ProxySQL 的配置保存在 SQLite 中。配置文件仅在 SQLite 不存在时读取。
> - ProxySQL 不建议使用配置文件进行配置更改，仅作为初始化配置时使用，请勿过度依赖配置文件。这是由于使用 ProxySQL Admin Interface 配置时，会有以下优点：
>   - 输入校验。
>   - 可使用任意 MySQL Client 进行配置更改。
>   - 更高的可用性（因为无需重启）。
>   - 在使用 ProxySQL Cluster 时将自动同步至其他 ProxySQL 节点。

5 部署标准集群

5.1 TiDB 软件和硬件环境建议配置

TiDB 作为一款开源一栈式实时 HTAP 数据库，可以很好地部署和运行在 Intel 架构服务器环境、ARM 架构的服务器环境及主流虚拟化环境，并支持绝大多数的主流硬件网络。作为一款高性能数据库系统，TiDB 支持主流的 Linux 操作系统环境。

5.1.1 操作系统及平台要求

操作系统	支持的 CPU 架构
Red Hat Enterprise Linux 8.4 及以上的 8.x 版本	x86_64 ARM 64
Red Hat Enterprise Linux 7.3 及以上的 7.x 版本 CentOS 7.3 及以上的 7.x 版本	x86_64 ARM 64
Amazon Linux 2	x86_64 ARM 64
麒麟欧拉版 V10 SP1/SP2	x86_64 ARM 64
UOS V20	x86_64 ARM 64
macOS 12 (Monterey) 及以上的版本	x86_64 ARM 64

| | Oracle Enterprise Linux 7.3 及以上的 7.x 版本 | x86_64 | | Ubuntu LTS 18.04 及以上的版本 | x86_64 | | CentOS 8 Stream |

x86_64

ARM 64

| | Debian 9 (Stretch) 及以上的版本 | x86_64 | | Fedora 35 及以上的版本 | x86_64 | | openSUSE Leap 15.3 以上的版本（不包含 Tumbleweed）| x86_64 | | SUSE Linux Enterprise Server 15 | x86_64 |

> **注意：**
>
> - TiDB 只支持 Red Hat 兼容内核 (RHCK) 的 Oracle Enterprise Linux，不支持 Oracle Enterprise Linux 提供的 Unbreakable Enterprise Kernel。
> - 根据 CentOS Linux EOL，CentOS Linux 8 的上游支持已于 2021 年 12 月 31 日终止，但 CentOS 将继续提供对 CentOS Stream 8 的支持。
> - TiDB 将不再支持 Ubuntu 16.04。强烈建议升级到 Ubuntu 18.04 或更高版本。
> - 对于以上表格中所列操作系统的 32 位版本，TiDB 在这些 32 位操作系统以及对应的 CPU 架构上不保障可编译、可构建以及可部署，或 TiDB 不主动适配这些 32 位的操作系统。
> - 以上未提及的操作系统版本也许可以运行 TiDB，但尚未得到 TiDB 官方支持。

5.1.1.1 编译和运行 TiDB 所依赖的库

编译和构建 TiDB 所需的依赖库	版本
Golang	1.20 及以上版本
Rust	nightly-2022-07-31 及以上版本
GCC	7.x
LLVM	13.0 及以上版本

运行时所需的依赖库：glibc（2.28-151.el8 版本）

5.1.1.2 Docker 镜像依赖

支持的 CPU 架构如下：

- x86_64，从 TiDB v6.6.0 开始，需要 x84-64-v2 指令集
- ARM 64

5.1.2 软件配置要求

5.1.2.1 中控机软件配置

软件	版本
sshpass	1.06 及以上
TiUP	1.5.0 及以上

注意：

中控机需要部署TiUP 软件来完成 TiDB 集群运维管理。

5.1.2.2 目标主机建议配置软件

软件	版本
sshpass	1.06 及以上
numa	2.0.12 及以上
tar	任意

5.1.3 服务器建议配置

TiDB 支持部署和运行在 Intel x86-64 架构的 64 位通用硬件服务器平台或者 ARM 架构的硬件服务器平台。对于开发、测试及生产环境的服务器硬件配置（不包含操作系统 OS 本身的占用）有以下要求和建议：

5.1.3.1 开发及测试环境

组件	CPU	内存	本地存储	网络	实例数量 (最低要求)
TiDB	8 核 +	16 GB+	无特殊要求	千兆网卡	1（可与 PD 同机器）
PD	4 核 +	8 GB+	SAS, 200 GB+	千兆网卡	1（可与 TiDB 同机器）
TiKV	8 核 +	32 GB+	SSD, 200 GB+	千兆网卡	3
TiFlash	32 核 +	64 GB+	SSD, 200 GB+	千兆网卡	1
TiCDC	8 核 +	16 GB+	SAS, 200 GB+	千兆网卡	1

注意：

- 验证测试环境中的 TiDB 和 PD 可以部署在同一台服务器上。
- 如进行性能相关的测试，避免采用低性能存储和网络硬件配置，防止对测试结果的正确性产生干扰。
- TiKV 的 SSD 盘推荐使用 NVME 接口以保证读写更快。
- 如果仅验证功能，建议使用TiDB 数据库快速上手指南进行单机功能测试。
- TiDB 对于磁盘的使用以存放日志为主，因此在测试环境中对于磁盘类型和容量并无特殊要求。

- 从 v6.3.0 开始，在 Linux AMD64 架构的硬件平台部署 TiFlash 时，CPU 必须支持 AVX2 指令集。确保命令 `cat /proc/cpuinfo | grep avx2` 有输出。而在 Linux ARM64 架构的硬件平台部署 TiFlash 时，CPU 必须支持 ARMv8 架构。确保命令 `cat /proc/cpuinfo | grep 'crc32' | grep 'asimd'` 有输出。通过使用向量扩展指令集，TiFlash 的向量化引擎能提供更好的性能。

5.1.3.2 生产环境

组件	CPU	内存	硬盘类型	网络	实例数量 (最低要求)
TiDB	16 核 +	48 GB+	SSD	万兆网卡（2 块最佳）	2
PD	8 核 +	16 GB+	SSD	万兆网卡（2 块最佳）	3
TiKV	16 核 +	64 GB+	SSD	万兆网卡（2 块最佳）	3
TiFlash	48 核 +	128 GB+	1 or more SSDs	万兆网卡（2 块最佳）	2
TiCDC	16 核 +	64 GB+	SSD	万兆网卡（2 块最佳）	2
监控	8 核 +	16 GB+	SAS	千兆网卡	1

注意：

- 生产环境中的 TiDB 和 PD 可以部署和运行在同一台服务器上，如对性能和可靠性有更高的要求，应尽可能分开部署。
- 生产环境强烈推荐使用更高的配置。
- TiKV 硬盘大小配置建议 PCI-E SSD 不超过 2 TB，普通 SSD 不超过 1.5 TB。
- TiFlash 支持多盘部署。
- TiFlash 数据目录的第一块磁盘推荐用高性能 SSD 来缓冲 TiKV 同步数据的实时写入，该盘性能应不低于 TiKV 所使用的磁盘，比如 PCI-E SSD。并且该磁盘容量建议不小于总容量的 10%，否则它可能成为这个节点的能承载的数据量的瓶颈。而其他磁盘可以根据需求部署多块普通 SSD，当然更好的 PCI-E SSD 硬盘会带来更好的性能。
- TiFlash 推荐与 TiKV 部署在不同节点，如果条件所限必须将 TiFlash 与 TiKV 部署在相同节点，则需要适当增加 CPU 核数和内存，且尽量将 TiFlash 与 TiKV 部署在不同的磁盘，以免互相干扰。
- TiFlash 硬盘总容量大致为：整个 TiKV 集群的需同步数据容量 / TiKV 副本数 * TiFlash 副本数。例如整体 TiKV 的规划容量为 1 TB、TiKV 副本数为 3、TiFlash 副本数为 2，则 TiFlash 的推荐总容量为 1024 GB / 3 * 2。用户可以选择同步部分表数据而非全部，具体容量可以根据需要同步的表的数据量具体分析。
- TiCDC 硬盘配置建议 1 TB+ PCIE-SSD。

5.1.4 网络要求

TiDB 作为开源一栈式实时 HTAP 数据库，其正常运行需要网络环境提供如下的网络端口配置要求，管理员可根据实际环境中 TiDB 组件部署的方案，在网络侧和主机侧开放相关端口：

组件	默认端口	说明
TiDB	4000	应用及 DBA 工具访问通信端口
TiDB	10080	TiDB 状态信息上报通信端口
TiKV	20160	TiKV 通信端口
TiKV	20180	TiKV 状态信息上报通信端口
PD	2379	提供 TiDB 和 PD 通信端口
PD	2380	PD 集群节点间通信端口
TiFlash	9000	TiFlash TCP 服务端口
TiFlash	8123	TiFlash HTTP 服务端口
TiFlash	3930	TiFlash RAFT 服务和 Coprocessor 服务端口
TiFlash	20170	TiFlash Proxy 服务端口
TiFlash	20292	Prometheus 拉取 TiFlash Proxy metrics 端口
TiFlash	8234	Prometheus 拉取 TiFlash metrics 端口
Pump	8250	Pump 通信端口
Drainer	8249	Drainer 通信端口
CDC	8300	CDC 通信接口
Monitoring	9090	Prometheus 服务通信端口
Monitoring	12020	NgMonitoring 服务通信端口
Node_exporter	9100	TiDB 集群每个节点的系统信息上报通信端口
Blackbox_exporter	9115	Blackbox_exporter 通信端口，用于 TiDB 集群端口监控
Grafana	3000	Web 监控服务对外服务和客户端 (浏览器) 访问端口
Alertmanager	9093	告警 web 服务端口
Alertmanager	9094	告警通信端口

5.1.5 磁盘空间要求

组件	磁盘空间要求	健康水位使用率

| TiDB |

日志盘建议最少预留 30 GB。

v6.5.0 及以上版本默认启用了 Fast Online DDL 对添加索引等 DDL 操作进行加速（通过变量`tidb_ddl_enable_fast_reorg` ↪ 控制）。如果业务中可能存在针对大对象的 DDL 操作，推荐为 TiDB 准备额外的 SSD 磁盘空间（建议 100 GB+）。配置方式详见设置 TiDB 节点的临时空间。

| 低于 90% | | PD | 数据盘和日志盘建议最少各预留 20 GB | 低于 90% | | TiKV | 数据盘和日志盘建议最少各预留 100 GB | 低于 80% | | TiFlash | 数据盘建议最少预留 100 GB，日志盘建议最少预留 30 GB | 低于 80% | | TiUP |

中控机：部署一个版本的 TiDB 集群占用不超过 1 GB 空间，部署多个版本集群所占用的空间会相应增加

部署服务器（实际运行 TiDB 各组件的机器）：TiFlash 占用约 700 MB 空间，其他组件（PD、TiDB、TiKV 等）各占用约 200 MB 空间。同时，部署过程会占用小于 1 MB 临时空间（/tmp）存放临时文件

| 不涉及 | | Ngmonitoring |

Conprof：3 x 1 GB x 组件数量（表示每个组件每天占用约 1 GB，总共 3 天）+ 20 GB 预留空间

Top SQL：30 x 50 MB x 组件数量（每个组件每天占用约 50 MB，总共 30 天）

Top SQL 和 Conprof 共享预留空间

| 不涉及 |

5.1.6 客户端 Web 浏览器要求

TiDB 提供了基于 Grafana 的技术平台，对数据库集群的各项指标进行可视化展现。采用支持 Javascript 的微软 IE、Google Chrome、 Mozilla Firefox 的较新版本即可访问监控入口。

5.2 TiDB 环境与系统配置检查

本文介绍部署 TiDB 前的环境检查操作，以下各项操作按优先级排序。

5.2.1 在 TiKV 部署目标机器上添加数据盘 EXT4 文件系统挂载参数

生产环境部署，建议使用 EXT4 类型文件系统的 NVME 类型的 SSD 磁盘存储 TiKV 数据文件。这个配置方案为最佳实施方案，其可靠性、安全性、稳定性已经在大量线上场景中得到证实。

使用 root 用户登录目标机器，将部署目标机器数据盘格式化成 ext4 文件系统，挂载时添加 nodelalloc 和 noatime 挂载参数。nodelalloc 是必选参数，否则 TiUP 安装时检测无法通过；noatime 是可选建议参数。

> **注意：**
>
> 如果你的数据盘已经格式化成 ext4 并挂载了磁盘，可先执行 umount /dev/nvme0n1p1 命令卸载，从编辑 /etc/fstab 文件步骤开始执行，添加挂载参数重新挂载即可。

以 /dev/nvme0n1 数据盘为例，具体操作步骤如下：

1. 查看数据盘。

```
fdisk -l
```

```
Disk /dev/nvme0n1: 1000 GB
```

2. 创建分区。

```
parted -s -a optimal /dev/nvme0n1 mklabel gpt -- mkpart primary ext4 1 -1
```

> **注意：**
>
> 使用 `lsblk` 命令查看分区的设备号：对于 nvme 磁盘，生成的分区设备号一般为 `nvme0n1p1`；对于普通磁盘（例如 `/dev/sdb`），生成的分区设备号一般为 `sdb1`。

3. 格式化文件系统。

```
mkfs.ext4 /dev/nvme0n1p1
```

4. 查看数据盘分区 UUID。

   本例中 `nvme0n1p1` 的 UUID 为 `c51eb23b-195c-4061-92a9-3fad812cc12f`。

```
lsblk -f
```

```
NAME    FSTYPE LABEL UUID                                 MOUNTPOINT
sda
├─sda1  ext4         237b634b-a565-477b-8371-6dff0c41f5ab /boot
├─sda2  swap         f414c5c0-f823-4bb1-8fdf-e531173a72ed
└─sda3  ext4         547909c1-398d-4696-94c6-03e43e317b60 /
sr0
nvme0n1
└─nvme0n1p1 ext4         c51eb23b-195c-4061-92a9-3fad812cc12f
```

5. 编辑 `/etc/fstab` 文件，添加 `nodelalloc` 挂载参数。

```
vi /etc/fstab
```

```
UUID=c51eb23b-195c-4061-92a9-3fad812cc12f /data1 ext4 defaults,nodelalloc,noatime 0 2
```

6. 挂载数据盘。

```
mkdir /data1 && \
mount -a
```

7. 执行以下命令，如果文件系统为 ext4，并且挂载参数中包含 `nodelalloc`，则表示已生效。

```
mount -t ext4
```

```
/dev/nvme0n1p1 on /data1 type ext4 (rw,noatime,nodelalloc,data=ordered)
```

5.2.2 检测及关闭系统 swap

TiDB 运行需要有足够的内存。如果内存不足，不建议使用 swap 作为内存不足的缓冲，因为这会降低性能。建议永久关闭系统 swap。

要永久关闭 swap，可执行以如下命令:

```
echo "vm.swappiness = 0">> /etc/sysctl.conf
swapoff -a && swapon -a
sysctl -p
```

注意：

- 一起执行 swapoff -a 和 swapon -a 命令是为了刷新 swap，将 swap 里的数据转储回内存，并清空 swap 里的数据。不可省略 swappiness 设置而只执行 swapoff -a；否则，重启后 swap 会再次自动打开，使得操作失效。
- 执行 sysctl -p 命令是为了在不重启的情况下使配置生效。

5.2.3 设置 TiDB 节点的临时空间（推荐）

TiDB 的部分操作需要向服务器写入临时文件，因此需要确保运行 TiDB 的操作系统用户具有足够的权限对目标目录进行读写。如果 TiDB 实例不是以 root 权限启动，则需要检查目录权限并进行正确设置。

- TiDB 临时工作区

  哈希表构建、排序等内存消耗较大的操作可能会向磁盘写入临时数据，用来减少内存消耗，提升稳定性。写入的磁盘位置由配置项tmp-storage-path 定义。在默认设置下，确保运行 TiDB 的用户对操作系统临时文件夹（通常为 /tmp）有读写权限。

- Fast Online DDL 工作区

  当变量tidb_ddl_enable_fast_reorg 被设置为 ON（v6.5.0 及以上版本中默认值为 ON）时，会激活 Fast Online DDL，这时部分 DDL 要对临时文件进行读写。临时文件位置由配置temp-dir 定义，需要确保运行 TiDB 的用户对操作系统中该目录有读写权限。以默认目录 /tmp/tidb 为例：

注意：
如果业务中可能存在针对大对象的 DDL 操作，推荐为temp-dir 配置独立文件系统及更大的临时空间。

```
sudo mkdir /tmp/tidb
```

如果目录 /tmp/tidb 已经存在，需确保有写入权限。

```
sudo chmod -R 777 /tmp/tidb
```

注意：
如果目录不存在，TiDB 在启动时会自动创建该目录。如果目录创建失败，或者 TiDB 对该目录没有读写权限，Fast Online DDL 在运行时可能产生不可预知的问题。

5.2.4 检测及关闭目标部署机器的防火墙

本段介绍如何关闭目标主机防火墙配置，因为在 TiDB 集群中，需要将节点间的访问端口打通才可以保证读写请求、数据心跳等信息的正常的传输。在普遍线上场景中，数据库到业务服务和数据库节点的网络联通都是在安全域内完成数据交互。如果没有特殊安全的要求，建议将目标节点的防火墙进行关闭。否则建议按照端口使用规则，将端口信息配置到防火墙服务的白名单中。

1. 检查防火墙状态（以 CentOS Linux release 7.7.1908 (Core) 为例）

```
sudo firewall-cmd --state
sudo systemctl status firewalld.service
```

2. 关闭防火墙服务

```
sudo systemctl stop firewalld.service
```

3. 关闭防火墙自动启动服务

```
sudo systemctl disable firewalld.service
```

4. 检查防火墙状态

```
sudo systemctl status firewalld.service
```

5.2.5 检测及安装 NTP 服务

TiDB 是一套分布式数据库系统，需要节点间保证时间的同步，从而确保 ACID 模型的事务线性一致性。目前解决授时的普遍方案是采用 NTP 服务，可以通过互联网中的 pool.ntp.org 授时服务来保证节点的时间同步，也可以使用离线环境自己搭建的 NTP 服务来解决授时。

采用如下步骤检查是否安装 NTP 服务以及与 NTP 服务器正常同步：

1. 执行以下命令，如果输出 running 表示 NTP 服务正在运行：

```
sudo systemctl status ntpd.service
```

```
ntpd.service - Network Time Service
Loaded: loaded (/usr/lib/systemd/system/ntpd.service; disabled; vendor preset: disabled)
Active: active (running) since 一 2017-12-18 13:13:19 CST; 3s ago
```

   - 若返回报错信息 Unit ntpd.service could not be found.，请尝试执行以下命令，以查看与 NTP 进行时钟同步所使用的系统配置是 chronyd 还是 ntpd：

```
sudo systemctl status chronyd.service
```

```
chronyd.service - NTP client/server
Loaded: loaded (/usr/lib/systemd/system/chronyd.service; enabled; vendor preset:
    ↪ enabled)
Active: active (running) since Mon 2021-04-05 09:55:29 EDT; 3 days ago
```

若发现系统既没有配置 chronyd 也没有配置 ntpd，则表示系统尚未安装任一服务。此时，应先安装其中一个服务，并保证它可以自动启动，默认使用 ntpd。

如果你使用的系统配置是 `chronyd`，请直接执行步骤 3。

2. 执行 ntpstat 命令检测是否与 NTP 服务器同步：

> **注意：**
>
> Ubuntu 系统需安装 ntpstat 软件包。

```
ntpstat
```

- 如果输出 synchronised to NTP server，表示正在与 NTP 服务器正常同步：

```
synchronised to NTP server (85.199.214.101) at stratum 2
time correct to within 91 ms
polling server every 1024 s
```

- 以下情况表示 NTP 服务未正常同步：

```
unsynchronised
```

- 以下情况表示 NTP 服务未正常运行：

```
Unable to talk to NTP daemon. Is it running?
```

3. 执行 chronyc tracking 命令查看 Chrony 服务是否与 NTP 服务器同步。

> **注意：**
>
> 该操作仅适用于使用 Chrony 的系统，不适用于使用 NTPd 的系统。

```
chronyc tracking
```

- 如果该命令返回结果为 Leap status : Normal，则代表同步过程正常。

```
Reference ID    : 5EC69F0A (ntp1.time.nl)
Stratum         : 2
Ref time (UTC)  : Thu May 20 15:19:08 2021
System time     : 0.000022151 seconds slow of NTP time
Last offset     : -0.000041040 seconds
RMS offset      : 0.000053422 seconds
Frequency       : 2.286 ppm slow
Residual freq   : -0.000 ppm
Skew            : 0.012 ppm
Root delay      : 0.012706812 seconds
Root dispersion : 0.000430042 seconds
Update interval : 1029.8 seconds
Leap status     : Normal
```

- 如果该命令返回结果如下，则表示同步过程出错：

```
Leap status    : Not synchronised
```

- 如果该命令返回结果如下，则表示 Chrony 服务未正常运行：

```
506 Cannot talk to daemon
```

如果要使 NTP 服务尽快开始同步，执行以下命令。可以将 pool.ntp.org 替换为你的 NTP 服务器：

```
sudo systemctl stop ntpd.service && \
sudo ntpdate pool.ntp.org && \
sudo systemctl start ntpd.service
```

如果要在 CentOS 7 系统上手动安装 NTP 服务，可执行以下命令：

```
sudo yum install ntp ntpdate && \
sudo systemctl start ntpd.service && \
sudo systemctl enable ntpd.service
```

5.2.6 检查和配置操作系统优化参数

在生产系统的 TiDB 中，建议对操作系统进行如下的配置优化：

1. 关闭透明大页（即 Transparent Huge Pages，缩写为 THP）。数据库的内存访问模式往往是稀疏的而非连续的。当高阶内存碎片化比较严重时，分配 THP 页面会出现较高的延迟。
2. 将存储介质的 I/O 调度器设置为 noop。对于高速 SSD 存储介质，内核的 I/O 调度操作会导致性能损失。将调度器设置为 noop 后，内核不做任何操作，直接将 I/O 请求下发给硬件，以获取更好的性能。同时，noop 调度器也有较好的普适性。
3. 为调整 CPU 频率的 cpufreq 模块选用 performance 模式。将 CPU 频率固定在其支持的最高运行频率上，不进行动态调节，可获取最佳的性能。

采用如下步骤检查操作系统的当前配置，并配置系统优化参数：

1. 执行以下命令查看透明大页的开启状态。

```
cat /sys/kernel/mm/transparent_hugepage/enabled
```

```
[always] madvise never
```

> 注意：
>
> `[always] madvise never` 表示透明大页处于启用状态，需要关闭。

2. 执行以下命令查看数据目录所在磁盘的 I/O 调度器。假设在 sdb、sdc 两个磁盘上创建了数据目录。

```
cat /sys/block/sd[bc]/queue/scheduler
```

```
noop [deadline] cfq
noop [deadline] cfq
```

> 注意：
>
> `noop [deadline] cfq` 表示磁盘的 I/O 调度器使用 `deadline`，需要进行修改。

3. 执行以下命令查看磁盘的唯一标识 `ID_SERIAL`。

```
udevadm info --name=/dev/sdb | grep ID_SERIAL
```

```
E: ID_SERIAL=36d0946606d79f90025f3e09a0c1f9e81
E: ID_SERIAL_SHORT=6d0946606d79f90025f3e09a0c1f9e81
```

> 注意：
>
> 如果多个磁盘都分配了数据目录，需要多次执行以上命令，记录所有磁盘各自的唯一标识。

4. 执行以下命令查看 cpufreq 模块选用的节能策略。

```
cpupower frequency-info --policy
```

```
analyzing CPU 0:
current policy: frequency should be within 1.20 GHz and 3.10 GHz.
              The governor "powersave" may decide which speed to use within this range.
```

> **注意：**
>
> The governor "powersave" 表示 cpufreq 的节能策略使用 powersave，需要调整为 performance 策略。如果是虚拟机或者云主机，则不需要调整，命令输出通常为 Unable to determine current policy。

5. 配置系统优化参数

    - 方法一：使用 tuned（推荐）

        1. 执行 tuned-adm list 命令查看当前操作系统的 tuned 策略。

```
tuned-adm list
```

```
Available profiles:
- balanced                    - General non-specialized tuned profile
- desktop                     - Optimize for the desktop use-case
- hpc-compute                 - Optimize for HPC compute workloads
- latency-performance         - Optimize for deterministic performance at the cost
    ↪ of increased power consumption
- network-latency             - Optimize for deterministic performance at the cost
    ↪ of increased power consumption, focused on low latency network performance
- network-throughput          - Optimize for streaming network throughput,
    ↪ generally only necessary on older CPUs or 40G+ networks
- powersave                   - Optimize for low power consumption
- throughput-performance      - Broadly applicable tuning that provides excellent
    ↪ performance across a variety of common server workloads
- virtual-guest               - Optimize for running inside a virtual guest
- virtual-host                - Optimize for running KVM guests
Current active profile: balanced
```

        Current active profile: balanced 表示当前操作系统的 tuned 策略使用 balanced，建议在当前策略的基础上添加操作系统优化配置。

        2. 创建新的 tuned 策略。

```
mkdir /etc/tuned/balanced-tidb-optimal/
vi /etc/tuned/balanced-tidb-optimal/tuned.conf
```

```
[main]
include=balanced

[cpu]
governor=performance

[vm]
transparent_hugepages=never
```

```
[disk]
devices_udev_regex=(ID_SERIAL=36d0946606d79f90025f3e09a0c1fc035)|(ID_SERIAL=36
    ↪ d0946606d79f90025f3e09a0c1f9e81)
elevator=noop
```

include=balanced 表示在现有的 balanced 策略基础上添加操作系统优化配置。

3. 应用新的 tuned 策略。

```
tuned-adm profile balanced-tidb-optimal
```

- 方法二：使用脚本方式。如果已经使用 tuned 方法，请跳过本方法。

1. 执行 grubby 命令查看默认内核版本。

> 注意：
> 需安装 grubby 软件包。

```
grubby --default-kernel
```

```
/boot/vmlinuz-3.10.0-957.el7.x86_64
```

2. 执行 grubby --update-kernel 命令修改内核配置。

```
grubby --args="transparent_hugepage=never" --update-kernel /boot/vmlinuz
    ↪ -3.10.0-957.el7.x86_64
```

> 注意：
> --update-kernel 后需要使用实际的默认内核版本。

3. 执行 grubby --info 命令查看修改后的默认内核配置。

```
grubby --info /boot/vmlinuz-3.10.0-957.el7.x86_64
```

> 注意：
> --info 后需要使用实际的默认内核版本。

```
index=0
kernel=/boot/vmlinuz-3.10.0-957.el7.x86_64
args="ro crashkernel=auto rd.lvm.lv=centos/root rd.lvm.lv=centos/swap rhgb quiet
    ↪ LANG=en_US.UTF-8 transparent_hugepage=never"
root=/dev/mapper/centos-root
initrd=/boot/initramfs-3.10.0-957.el7.x86_64.img
title=CentOS Linux (3.10.0-957.el7.x86_64) 7 (Core)
```

4. 修改当前的内核配置立即关闭透明大页。

```
echo never > /sys/kernel/mm/transparent_hugepage/enabled
echo never > /sys/kernel/mm/transparent_hugepage/defrag
```

5. 配置 udev 脚本应用 IO 调度器策略。

```
vi /etc/udev/rules.d/60-tidb-schedulers.rules
```

```
ACTION=="add|change", SUBSYSTEM=="block", ENV{ID_SERIAL}=="36
    ↪ d0946606d79f90025f3e09a0c1fc035", ATTR{queue/scheduler}="noop"
ACTION=="add|change", SUBSYSTEM=="block", ENV{ID_SERIAL}=="36
    ↪ d0946606d79f90025f3e09a0c1f9e81", ATTR{queue/scheduler}="noop"
```

6. 应用 udev 脚本。

```
udevadm control --reload-rules
udevadm trigger --type=devices --action=change
```

7. 创建 CPU 节能策略配置服务。

```
cat  >> /etc/systemd/system/cpupower.service << EOF
[Unit]
Description=CPU performance
[Service]
Type=oneshot
ExecStart=/usr/bin/cpupower frequency-set --governor performance
[Install]
WantedBy=multi-user.target
EOF
```

8. 应用 CPU 节能策略配置服务。

```
systemctl daemon-reload
systemctl enable cpupower.service
systemctl start cpupower.service
```

6. 执行以下命令验证透明大页的状态。

```
cat /sys/kernel/mm/transparent_hugepage/enabled
```

```
always madvise [never]
```

7. 执行以下命令验证数据目录所在磁盘的 I/O 调度器。

```
cat /sys/block/sd[bc]/queue/scheduler
```

```
[noop] deadline cfq
[noop] deadline cfq
```

8. 执行以下命令查看 cpufreq 模块选用的节能策略。

```
cpupower frequency-info --policy
```

```
analyzing CPU 0:
current policy: frequency should be within 1.20 GHz and 3.10 GHz.
              The governor "performance" may decide which speed to use within this range.
```

9. 执行以下命令修改 sysctl 参数。

```
echo "fs.file-max = 1000000">> /etc/sysctl.conf
echo "net.core.somaxconn = 32768">> /etc/sysctl.conf
echo "net.ipv4.tcp_tw_recycle = 0">> /etc/sysctl.conf
echo "net.ipv4.tcp_syncookies = 0">> /etc/sysctl.conf
echo "vm.overcommit_memory = 1">> /etc/sysctl.conf
sysctl -p
```

10. 执行以下命令配置用户的 limits.conf 文件。

```
cat << EOF >>/etc/security/limits.conf
tidb           soft    nofile          1000000
tidb           hard    nofile          1000000
tidb           soft    stack          32768
tidb           hard    stack          32768
EOF
```

5.2.7 手动配置 SSH 互信及 sudo 免密码

对于有需求，通过手动配置中控机至目标节点互信的场景，可参考本段。通常推荐使用 TiUP 部署工具会自动配置 SSH 互信及免密登录，可忽略本段内容。

1. 以 root 用户依次登录到部署目标机器创建 tidb 用户并设置登录密码。

```
useradd tidb && \
passwd tidb
```

2. 执行以下命令，将 tidb ALL=(ALL)NOPASSWD: ALL 添加到文件末尾，即配置好 sudo 免密码。

```
visudo
```

```
tidb ALL=(ALL) NOPASSWD: ALL
```

3. 以 tidb 用户登录到中控机，执行以下命令。将 10.0.1.1 替换成你的部署目标机器 IP，按提示输入部署目标机器 tidb 用户密码，执行成功后即创建好 SSH 互信，其他机器同理。新建的 tidb 用户下没有 .ssh 目录，需要执行生成 rsa 密钥的命令来生成 .ssh 目录。如果要在中控机上部署 TiDB 组件，需要为中控机和中控机自身配置互信。

```
ssh-keygen -t rsa
ssh-copy-id -i ~/.ssh/id_rsa.pub 10.0.1.1
```

4. 以 tidb 用户登录中控机，通过 ssh 的方式登录目标机器 IP。如果不需要输入密码并登录成功，即表示 SSH 互信配置成功。

```
ssh 10.0.1.1
```

```
[tidb@10.0.1.1 ~]$
```

5. 以 tidb 用户登录到部署目标机器后，执行以下命令，不需要输入密码并切换到 root 用户，表示 tidb 用户 sudo 免密码配置成功。

```
sudo -su root
```

```
[root@10.0.1.1 tidb]#
```

5.2.8 安装 numactl 工具

本段主要介绍如何安装 NUMA 工具。在生产环境中，因为硬件机器配置往往高于需求，为了更合理规划资源，会考虑单机多实例部署 TiDB 或者 TiKV。NUMA 绑核工具的使用，主要为了防止 CPU 资源的争抢，引发性能衰退。

> **注意：**
>
> - NUMA 绑核是用来隔离 CPU 资源的一种方法，适合高配置物理机环境部署多实例使用。
> - 通过 tiup cluster deploy 完成部署操作，就可以通过 exec 命令来进行集群级别管理工作。

安装 NUMA 工具有两种方法：

方法 1：登录到目标节点进行安装（以 CentOS Linux release 7.7.1908 (Core) 为例）。

```
sudo yum -y install numactl
```

方法 2：通过 tiup cluster exec 在集群上批量安装 NUMA。

1. 使用 TiUP 安装 TiDB 集群，参考使用 TiUP 部署 TiDB 集群完成 tidb-test 集群的部署。如果本地已有集群，可跳过这一步。

```
tiup cluster deploy tidb-test v6.1.0 ./topology.yaml --user root [-p] [-i /home/root/.ssh/
    ↪ gcp_rsa]
```

2. 执行 tiup cluster exec 命令，以 sudo 权限在 tidb-test 集群所有目标主机上安装 NUMA。

```
tiup cluster exec tidb-test --sudo --command "yum -y install numactl"
```

你可以执行 tiup cluster exec --help 查看的 tiup cluster exec 命令的说明信息。

5.3 规划集群拓扑

5.3.1 最小拓扑架构

本文档介绍 TiDB 集群最小部署的拓扑架构。

5.3.1.1 拓扑信息

实例	个数	物理机配置	IP	配置
TiDB	2	16 VCore 32 GiB 100 GiB 用于存储	10.0.1.1 10.0.1.2	默认端口全局目录配置
PD	3	4 VCore 8 GiB 100 GiB 用于存储	10.0.1.4 10.0.1.5 10.0.1.6	默认端口全局目录配置
TiKV	3	16 VCore 32 GiB 2 TiB (NVMe SSD) 用于存储	10.0.1.7 10.0.1.8 10.0.1.9	默认端口全局目录配置

实例	个数	物理机配置	IP	配置
Monitoring & Grafana	1	4 VCore 8 GiB 500 GiB (SSD) 用于存储	10.0.1.11	默认端口全局目录配置

5.3.1.1.1 拓扑模版

简单最小配置模板

详细最小配置模板

以上 TiDB 集群拓扑文件中，详细的配置项说明见通过 TiUP 部署 TiDB 集群的拓扑文件配置。

> 注意：
>
> - 无需手动创建配置文件中的 `tidb` 用户，TiUP cluster 组件会在目标主机上自动创建该用户。可以自定义用户，也可以和中控机的用户保持一致。
> - 如果部署目录配置为相对路径，会部署在用户的 Home 目录下。

5.3.2 TiFlash 部署拓扑

本文介绍在部署最小拓扑集群的基础上，部署TiFlash 的拓扑结构。TiFlash 是列式的存储引擎，已经成为集群拓扑的标配，适合 Real-Time HTAP 业务。

5.3.2.1 拓扑信息

实例	个数	物理机配置	IP	配置
TiDB	3	16 VCore 32GB * 1	10.0.1.7 10.0.1.8 10.0.1.9	默认端口全局目录配置
PD	3	4 VCore 8GB * 1	10.0.1.4 10.0.1.5 10.0.1.6	默认端口全局目录配置
TiKV	3	16 VCore 32GB 2TB (nvme ssd) * 1	10.0.1.1 10.0.1.2 10.0.1.3	默认端口全局目录配置

实例	个数	物理机配置	IP	配置
TiFlash	1	32 VCore 64 GB 2TB (nvme ssd) * 1	10.0.1.11	默认端口全局目录配置
Monitoring & Grafana	1	4 VCore 8GB * 1 500GB (ssd)	10.0.1.10	默认端口全局目录配置

5.3.2.1.1 拓扑模版

- 简单 TiFlash 配置模版
- 详细 TiFlash 配置模版

以上 TiDB 集群拓扑文件中，详细的配置项说明见通过 TiUP 部署 TiDB 集群的拓扑文件配置。

5.3.2.1.2 关键参数介绍

- 需要将配置模板中 `replication.enable-placement-rules` 设置为 `true`，以开启 PD 的Placement Rules 功能。
- `tiflash_servers` 实例级别配置 `"-host"` 目前只支持 IP，不支持域名。
- TiFlash 具体的参数配置介绍可参考TiFlash 参数配置。

> **注意：**
>
> - 无需手动创建配置文件中的 `tidb` 用户，TiUP cluster 组件会在目标主机上自动创建该用户。可以自定义用户，也可以和中控机的用户保持一致。
> - 如果部署目录配置为相对路径，会部署在用户的 Home 目录下。

5.3.3 TiCDC 部署拓扑

> **注意：**
>
> TiCDC 从 v4.0.6 起成为正式功能，可用于生产环境。

本文介绍TiCDC 部署的拓扑，以及如何在最小拓扑的基础上同时部署 TiCDC。TiCDC 是 4.0 版本开始支持的 TiDB 增量数据同步工具，支持多种下游 (TiDB/MySQL/MQ)。相比于 TiDB Binlog，TiCDC 有延迟更低、天然高可用等优点。

实例	个数	物理机配置	IP	配置

5.3.3.1 拓扑信息

实例	个数	物理机配置	IP	配置
TiDB	3	16 VCore 32GB * 1	10.0.1.1 10.0.1.2 10.0.1.3	默认端口全局目录配置
PD	3	4 VCore 8GB * 1	10.0.1.4 10.0.1.5 10.0.1.6	默认端口全局目录配置
TiKV	3	16 VCore 32GB 2TB (nvme ssd) * 1	10.0.1.7 10.0.1.8 10.0.1.9	默认端口全局目录配置
CDC	3	8 VCore 16GB * 1	10.0.1.11 10.0.1.12 10.0.1.13	默认端口全局目录配置
Monitoring & Grafana	1	4 VCore 8GB * 1 500GB (ssd)	10.0.1.11	默认端口全局目录配置

5.3.3.1.1 拓扑模版

简单 TiCDC 配置模板

详细 TiCDC 配置模板

以上 TiDB 集群拓扑文件中，详细的配置项说明见通过 TiUP 部署 TiDB 集群的拓扑文件配置。

> 注意：
>
> - 无需手动创建配置文件中的 `tidb` 用户，TiUP cluster 组件会在目标主机上自动创建该用户。可以自定义用户，也可以和中控机的用户保持一致。
> - 如果部署目录配置为相对路径，会部署在用户的 Home 目录下。

5.3.4 TiDB Binlog 部署拓扑

本文介绍在部署最小拓扑集群的基础上，同时部署TiDB Binlog。TiDB Binlog 是目前广泛使用的增量同步组件，可提供准实时备份和同步功能。

5.3.4.1 拓扑信息

实例	个数	物理机配置	IP	配置
TiDB	3	16 VCore 32 GB	10.0.1.1 10.0.1.2 10.0.1.3	默认端口配置；开启 enable_binlog；开启 ignore-error
PD	3	4 VCore 8 GB	10.0.1.4 10.0.1.5 10.0.1.6	默认端口配置
TiKV	3	16 VCore 32 GB	10.0.1.7 10.0.1.8 10.0.1.9	默认端口配置
Pump	3	8 VCore 16GB	10.0.1.1 10.0.1.7 10.0.1.8	默认端口配置；设置 GC 时间 7 天

实例	个数	物理机配置	IP	配置
Drainer	1	8 VCore 16GB	10.0.1.11	默认端口配置；设置默认初始化 commitTS -1 为最近的时间戳配置下游目标 TiDB 10.0.1.12:4000

5.3.4.1.1 拓扑模版

简单 TiDB Binlog 配置模板（下游为 MySQL）

简单 TiDB Binlog 配置模板（下游为 file）

详细 TiDB Binlog 配置模板

以上 TiDB 集群拓扑文件中，详细的配置项说明见通过 TiUP 部署 TiDB 集群的拓扑文件配置。

5.3.4.1.2 关键参数介绍

拓扑配置模版的关键参数如下：

- `server_configs.tidb.binlog.enable: true`

  开启 TiDB Binlog 服务，默认为 false。

- `server_configs.tidb.binlog.ignore-error: true`

  高可用场景建议开启，如果设置为 true，发生错误时，TiDB 会停止写入 TiDB Binlog，并且在监控项 `tidb_server_critical_error_total` 上计数加 1；如果设置为 false，一旦写入 TiDB Binlog 失败，会停止整个 TiDB 的服务。

- `drainer_servers.config.syncer.db-type`

  TiDB Binlog 的下游类型，目前支持 `mysql`、`tidb`、`kafka` 和 `file`。

- `drainer_servers.config.syncer.to`

  TiDB Binlog 的下游配置。根据 `db-type` 的不同，该选项可配置下游数据库的连接参数、Kafka 的连接参数、文件保存路径。详细说明可参见TiDB Binlog 配置说明。

> 注意：
>
> - 编辑配置文件模版时，如无需自定义端口或者目录，仅修改 IP 即可。
> - 无需手动创建配置文件中的 `tidb` 用户，TiUP cluster 组件会在目标主机上自动创建该用户。可以自定义用户，也可以和中控机的用户保持一致。
> - 如果部署目录配置为相对路径，会部署在用户的 Home 目录下。

5.3.5 TiSpark 部署拓扑

本文介绍 TiSpark 部署的拓扑，以及如何在最小拓扑的基础上同时部署 TiSpark。TiSpark 是 PingCAP 为解决用户复杂 OLAP 需求而推出的产品。它借助 Spark 平台，同时融合 TiKV 分布式集群的优势，和 TiDB 一起为用户一站式解决 HTAP (Hybrid Transactional/Analytical Processing) 的需求。

关于 TiSpark 的架构介绍与使用，参见TiSpark 用户指南。

> 警告：
>
> TiUP Cluster 的 TiSpark 支持目前为废弃状态，不建议使用。

5.3.5.1 拓扑信息

实例	个数	物理机配置	IP	配置
TiDB	3	16 VCore 32GB * 1	10.0.1.1 10.0.1.2 10.0.1.3	默认端口全局目录配置
PD	3	4 VCore 8GB * 1	10.0.1.4 10.0.1.5 10.0.1.6	默认端口全局目录配置
TiKV	3	16 VCore 32GB 2TB (nvme ssd) * 1	10.0.1.7 10.0.1.8 10.0.1.9	默认端口全局目录配置

实例	个数	物理机配置	IP	配置
TiSpark	3	8 VCore 16GB * 1	10.0.1.21 (master) 10.0.1.22 (worker) 10.0.1.23 (worker)	默认端口 全局目录配置
Monitoring & Grafana	1	4 VCore 8GB * 1 500GB (ssd)	10.0.1.11	默认端口 全局目录配置

5.3.5.1.1 拓扑模版

简单 TiSpark 配置模板

详细 TiSpark 配置模板

以上 TiDB 集群拓扑文件中，详细的配置项说明见通过 TiUP 部署 TiDB 集群的拓扑文件配置。

> **注意：**
>
> - 无需手动创建配置文件中的 `tidb` 用户，TiUP cluster 组件会在目标主机上自动创建该用户。可以自定义用户，也可以和中控机的用户保持一致。
> - 如果部署目录配置为相对路径，会部署在用户的 Home 目录下。

5.3.5.2 环境要求

由于 TiSpark 基于 Apache Spark 集群，在启动包含 TiSpark 组件的 TiDB 集群前，需要在部署了 TiSpark 组件的服务器上安装 Java 运行时环境 (JRE) 8，否则将无法启动相关组件。

TiUP 不提供自动安装 JRE 的支持，该操作需要用户自行完成。JRE 8 的安装方法可以参考 OpenJDK 的文档说明。

如果部署服务器上已经安装有 JRE 8，但不在系统的默认包管理工具路径中，可以通过在拓扑配置中设置 java_home 参数来指定要使用的 JRE 环境所在的路径。该参数对应系统环境变量 JAVA_HOME。

5.3.6 跨数据中心部署拓扑

本文以典型的两地三中心为例，介绍跨数据中心部署的拓扑以及关键参数。本文示例所涉及的城市是上海（即 sha）和北京（即 bja 和 bjb）。

5.3.6.1 拓扑信息

实例	个数	物理机配置	BJ IP	SH IP	配置
TiDB	5	16 VCore 32GB * 1	10.0.1.1 10.0.1.2 10.0.1.3 10.0.1.4	10.0.1.5	默认端口全局目录配置
PD	5	4 VCore 8GB * 1	10.0.1.6 10.0.1.7 10.0.1.8 10.0.1.9	10.0.1.10	默认端口全局目录配置

实例	个数	物理机配置	BJ IP	SH IP	配置
TiKV	5	16 VCore 32GB 2TB (nvme ssd) * 1	10.0.1.11 10.0.1.12 10.0.1.13 10.0.1.14	10.0.1.15	默认端口全局目录配置
Monitoring & Grafana		4 VCore 8GB * 1 500GB (ssd)	10.0.1.16		默认端口全局目录配置

5.3.6.1.1 拓扑模版

跨机房配置模板

以上 TiDB 集群拓扑文件中，详细的配置项说明见通过 TiUP 部署 TiDB 集群的拓扑文件配置。

5.3.6.1.2 关键参数配置

本节介绍跨数据中心部署 TiDB 集群的关键参数配置。

TiKV 参数

- 设置 gRPC 的压缩格式，默认为 none。为提高跨机房部署场景的目标节点间 gRPC 包的传输速度，建议设置为 gzip 格式。

```
server.grpc-compression-type: gzip
```

- label 配置

  由于采用跨机房部署 TiKV，为了避免物理机宕机导致 Raft Group 默认的 5 副本中丢失 3 副本，使集群不可用的问题，可以通过 label 来实现 PD 智能调度，保证同中心、同机柜、同机器 TiKV 实例不会出现 Raft Group 有 3 副本的情况。

- TiKV 配置

  相同物理机配置相同的 host 级别 label 信息：

```
config:
  server.labels:
    zone: bj
    dc: bja
    rack: rack1
    host: host2
```

- 防止异地 TiKV 节点发起不必要的 Raft 选举，需要将异地 TiKV 节点发起选举时经过最少的 tick 个数和最多经过的 tick 个数都调大，这两个参数默认设置均为 0。

```
raftstore.raft-min-election-timeout-ticks: 1000
raftstore.raft-max-election-timeout-ticks: 1020
```

PD 参数

- PD 元数据信息记录 TiKV 集群的拓扑信息，根据四个维度调度 Raft Group 副本。

```
replication.location-labels: ["zone","dc","rack","host"]
```

- 调整 Raft Group 的副本数据量为 5，保证集群的高可用性。

```
replication.max-replicas: 5
```

- 拒绝异地机房 TiKV 的 Raft 副本选举为 Leader。

```
label-property:
      reject-leader:
        - key: "dc"
          value: "sha"
```

> 注意：
>
> TiDB 5.2 及以上版本默认不支持 `label-property` 配置。若要设置副本策略，请使用Placement Rules。

有关 Label 的使用和 Raft Group 副本数量，详见通过拓扑 label 进行副本调度。

> 注意：
>
> - 无需手动创建配置文件中的 `tidb` 用户，TiUP cluster 组件会在目标主机上自动创建该用户。可以自定义用户，也可以和中控机的用户保持一致。
> - 如果部署目录配置为相对路径，会部署在用户的 Home 目录下。

5.3.7 混合部署拓扑

本文介绍 TiDB 集群的 TiKV 和 TiDB 混合部署拓扑以及主要参数。常见的场景为，部署机为多路 CPU 处理器，内存也充足，为提高物理机资源利用率，可单机多实例部署，即 TiDB、TiKV 通过 numa 绑核，隔离 CPU 资源。PD 和 Prometheus 混合部署，但两者的数据目录需要使用独立的文件系统。

5.3.7.1 拓扑信息

实例	个数	物理机配置	IP	配置
TiDB	6	32 VCore 64GB	10.0.1.1 10.0.1.2 10.0.1.3	配置 numa 绑核操作
PD	3	16 VCore 32 GB	10.0.1.4 10.0.1.5 10.0.1.6	配置 location_labels 参数

实例	个数	物理机配置	IP	配置
TiKV	6	32 VCore 64GB	10.0.1.7 10.0.1.8 10.0.1.9	1. 区分实例级别的 port、status_port； 2. 配置全局参数 readpool、storage 以及 raftstore； 3. 配置实例级别 host 维度的 labels； 4. 配置 numa 绑核

实例	个数	物理机配置	IP	配置
Monitoring & Grafana	1	4 VCore 8GB * 1 500GB (ssd)	10.0.1.11	默认配置

5.3.7.1.1 拓扑模版

简单混部配置模板

详细混部配置模板

以上 TiDB 集群拓扑文件中，详细的配置项说明见通过 TiUP 部署 TiDB 集群的拓扑文件配置。

5.3.7.1.2 混合部署的关键参数介绍

本节介绍单机多实例的关键参数，主要用于 TiDB、TiKV 的单机多实例部署场景。你需要按照提供的计算公式，将结果填写至上一步的配置模板中。

- TiKV 进行配置优化
  - readpool 线程池自适应，配置 `readpool.unified.max-thread-count` 参数可以使 `readpool.storage` 和 `readpool.coprocessor` 共用统一线程池，同时要分别设置自适应开关。
    * 开启 `readpool.storage` 和 `readpool.coprocessor`：

```
readpool.storage.use-unified-pool: true
readpool.coprocessor.use-unified-pool: true
```

    * 计算公式如下：

```
readpool.unified.max-thread-count = cores * 0.8 / TiKV 数量
```

  - storage CF (all RocksDB column families) 内存自适应，配置 `storage.block-cache.capacity` 参数即可实现 CF 之间自动平衡内存使用。
    * 计算公式如下：

```
storage.block-cache.capacity = (MEM_TOTAL * 0.5 / TiKV 实例数量)
```

  - 如果多个 TiKV 实例部署在同一块物理磁盘上，需要在 tikv 配置中添加 capacity 参数：

```
raftstore.capacity = 磁盘总容量 / TiKV 实例数量
```

- label 调度配置

  由于采用单机多实例部署 TiKV，为了避免物理机宕机导致 Region Group 默认 3 副本的 2 副本丢失，导致集群不可用的问题，可以通过 label 来实现 PD 智能调度，保证同台机器的多 TiKV 实例不会出现 Region Group 只有 2 副本的情况。

  - TiKV 配置

    相同物理机配置相同的 host 级别 label 信息：

    ```
    config:
      server.labels:
        host: tikv1
    ```

  - PD 配置

    PD 需要配置 labels 类型来识别并调度 Region：

    ```
    pd:
      replication.location-labels: ["host"]
    ```

- numa_node 绑核

  - 在实例参数模块配置对应的 numa_node 参数，并添加对应的物理 CPU 的核数；
  - numa 绑核使用前，确认已经安装 numactl 工具，以及物理机对应的物理机 CPU 的信息后，再进行参数配置；
  - numa_node 这个配置参数与 `numactl --membind` 配置对应。

注意：

- 编辑配置文件模版时，注意修改必要参数、IP、端口及目录。
- 各个组件的 deploy_dir，默认会使用 global 中的 `<deploy_dir>/<components_name>-<port>`。例如 tidb 端口指定 4001，则 deploy_dir 默认为 ‘/tidb-deploy/tidb-4001’。因此，在多实例场景下指定非默认端口时，无需再次指定目录。
- 无需手动创建配置文件中的 `tidb` 用户，TiUP cluster 组件会在部署主机上自动创建该用户。可以自定义用户，也可以和中控机的用户保持一致。
- 如果部署目录配置为相对路径，会部署在用户的 Home 目录下。

5.4 安装与启动

5.4.1 使用 TiUP 部署 TiDB 集群

TiUP 是 TiDB 4.0 版本引入的集群运维工具，TiUP cluster 是 TiUP 提供的使用 Golang 编写的集群管理组件，通过 TiUP cluster 组件就可以进行日常的运维工作，包括部署、启动、关闭、销毁、弹性扩缩容、升级 TiDB 集群，以及管理 TiDB 集群参数。

目前 TiUP 可以支持部署 TiDB、TiFlash、TiDB Binlog、TiCDC 以及监控系统。本文将介绍不同集群拓扑的具体部署步骤。

5.4.1.1 第 1 步：软硬件环境需求及前置检查

软硬件环境需求

环境与系统配置检查

5.4.1.2 第 2 步：在中控机上部署 TiUP 组件

在中控机上部署 TiUP 组件有两种方式：在线部署和离线部署。

5.4.1.2.1 在线部署

以普通用户身份登录中控机。以 tidb 用户为例，后续安装 TiUP 及集群管理操作均通过该用户完成：

1. 执行如下命令安装 TiUP 工具：

```
curl --proto '=https' --tlsv1.2 -sSf https://tiup-mirrors.pingcap.com/install.sh | sh
```

2. 按如下步骤设置 TiUP 环境变量：

    1. 重新声明全局环境变量：

    ```
    shell source .bash_profile
    ```

    2. 确认 TiUP 工具是否安装：

    ```
    which tiup
    ```

3. 安装 TiUP cluster 组件：

```
tiup cluster
```

4. 如果已经安装，则更新 TiUP cluster 组件至最新版本：

```
tiup update --self && tiup update cluster
```

预期输出 “Update successfully!” 字样。

5. 验证当前 TiUP cluster 版本信息。执行如下命令查看 TiUP cluster 组件版本：

```
tiup --binary cluster
```

5.4.1.2.2 离线部署

离线部署 TiUP 组件的操作步骤如下。

准备 TiUP 离线组件包

方式一：在官方下载页面选择对应版本的 TiDB server 离线镜像包（包含 TiUP 离线组件包）。需要同时下载 TiDB-community-server 软件包和 TiDB-community-toolkit 软件包。

方式二：使用 tiup mirror clone 命令手动打包离线组件包。步骤如下：

1. 在在线环境中安装 TiUP 包管理器工具

    1. 执行如下命令安装 TiUP 工具：

    ```
    curl --proto '=https' --tlsv1.2 -sSf https://tiup-mirrors.pingcap.com/install.sh | sh
    ```

    2. 重新声明全局环境变量：

    ```
    source .bash_profile
    ```

    3. 确认 TiUP 工具是否安装：

    ```
    which tiup
    ```

2. 使用 TiUP 制作离线镜像

    1. 在一台和外网相通的机器上拉取需要的组件：

    ```
    tiup mirror clone tidb-community-server-${version}-linux-amd64 ${version} --os=linux --
        ↪ arch=amd64
    ```

    该命令会在当前目录下创建一个名叫 `tidb-community-server-${version}-linux-amd64` 的目录，里面包含 TiUP 管理的组件包。

    2. 通过 tar 命令将该组件包打包然后发送到隔离环境的中控机：

    ```
    tar czvf tidb-community-server-${version}-linux-amd64.tar.gz tidb-community-server-${
        ↪ version}-linux-amd64
    ```

    此时，`tidb-community-server-${version}-linux-amd64.tar.gz` 就是一个独立的离线环境包。

3. 自定义制作的离线镜像，或调整已有离线镜像中的内容

    如果从官网下载的离线镜像不满足你的具体需求，或者希望对已有的离线镜像内容进行调整，例如增加某个组件的新版本等，可以采取以下步骤进行操作：

    1. 在制作离线镜像时，可通过参数指定具体的组件和版本等信息，获得不完整的离线镜像。例如，要制作一个只包括 v1.11.3 版本 TiUP 和 TiUP Cluster 的离线镜像，可执行如下命令：

    ```
    tiup mirror clone tiup-custom-mirror-v1.11.3 --tiup v1.11.3 --cluster v1.11.3
    ```

    如果只需要某一特定平台的组件，也可以通过 `--os` 和 `--arch` 参数来指定。

    2. 参考上文“使用 TiUP 制作离线镜像”第 2 步的方式，将此不完整的离线镜像传输到隔离环境的中控机。

    3. 在隔离环境的中控机上，查看当前使用的离线镜像路径。较新版本的 TiUP 可以直接通过命令获取当前的镜像地址：

    ```
    tiup mirror show
    ```

    以上命令如果提示 show 命令不存在，可能当前使用的是较老版本的 TiUP。此时可以通过查看 `$HOME/.tiup/tiup.toml` 获得正在使用的镜像地址。将此镜像地址记录下来，后续步骤中将以变量 `${base_mirror}` 指代此镜像地址。

4. 将不完整的离线镜像合并到已有的离线镜像中：

   首先将当前离线镜像中的 keys 目录复制到 $HOME/.tiup 目录中：

   ```
   cp -r ${base_mirror}/keys $HOME/.tiup/
   ```

   然后使用 TiUP 命令将不完整的离线镜像合并到当前使用的镜像中：

   ```
   tiup mirror merge tiup-custom-mirror-v1.11.3
   ```

5. 上述步骤完成后，通过 tiup list 命令检查执行结果。在本文例子中，使用 tiup list tiup 和 tiup list cluster 均应能看到对应组件的 v1.11.3 版本出现在结果中。

部署离线环境 TiUP 组件

将离线包发送到目标集群的中控机后，执行以下命令安装 TiUP 组件：

```
tar xzvf tidb-community-server-${version}-linux-amd64.tar.gz && \
sh tidb-community-server-${version}-linux-amd64/local_install.sh && \
source /home/tidb/.bash_profile
```

local_install.sh 脚本会自动执行 tiup mirror set tidb-community-server-${version}-linux-amd64 命令将当前镜像地址设置为 tidb-community-server-${version}-linux-amd64。

合并离线包

如果是通过官方下载页面下载的离线软件包，需要将 TiDB-community-server 软件包和 TiDB-community-toolkit 软件包合并到离线镜像中。如果是通过 tiup mirror clone 命令手动打包的离线组件包，不需要执行此步骤。

执行以下命令合并离线组件到 server 目录下。

```
tar xf tidb-community-toolkit-${version}-linux-amd64.tar.gz
ls -ld tidb-community-server-${version}-linux-amd64 tidb-community-toolkit-${version}-linux-amd64
cd tidb-community-server-${version}-linux-amd64/
cp -rp keys ~/.tiup/
tiup mirror merge ../tidb-community-toolkit-${version}-linux-amd64
```

若需将镜像切换到其他目录，可以通过手动执行 tiup mirror set <mirror-dir> 进行切换。如果需要切换到在线环境，可执行 tiup mirror set https://tiup-mirrors.pingcap.com。

5.4.1.3 第 3 步：初始化集群拓扑文件

执行如下命令，生成集群初始化配置文件：

```
tiup cluster template > topology.yaml
```

针对两种常用的部署场景，也可以通过以下命令生成建议的拓扑模板：

- 混合部署场景：单台机器部署多个实例，详情参见混合部署拓扑架构。

  ```
  tiup cluster template --full > topology.yaml
  ```

- 跨机房部署场景：跨机房部署 TiDB 集群，详情参见跨机房部署拓扑架构。

```
tiup cluster template --multi-dc > topology.yaml
```

执行 vi topology.yaml，查看配置文件的内容：

```
global:
  user: "tidb"
  ssh_port: 22
  deploy_dir: "/tidb-deploy"
  data_dir: "/tidb-data"
server_configs: {}
pd_servers:
  - host: 10.0.1.4
  - host: 10.0.1.5
  - host: 10.0.1.6
tidb_servers:
  - host: 10.0.1.7
  - host: 10.0.1.8
  - host: 10.0.1.9
tikv_servers:
  - host: 10.0.1.1
  - host: 10.0.1.2
  - host: 10.0.1.3
monitoring_servers:
  - host: 10.0.1.4
grafana_servers:
  - host: 10.0.1.4
alertmanager_servers:
  - host: 10.0.1.4
```

下表列出了常用的 7 种场景，请根据链接中的拓扑说明以及配置文件模板配置topology.yaml。如果有其他组合场景的需求，请根据多个模板自行调整。

场景	配置任务	配置文件模板	拓扑说明
OLTP 业务	部署最小拓扑架构	简单最小配置模板详细最小配置模板	最小集群拓扑，包括 tidb-server、tikv-server、pd-server。

场景	配置任务	配置文件模板	拓扑说明
HTAP 业务	部署 Ti-Flash 拓扑架构	简单 Ti-Flash 配置模版 详细 Ti-Flash 配置模版	在最小拓扑的基础上部署 Ti-Flash。Ti-Flash 是列式存储引擎，已经逐步成为集群拓扑的标配。

场景	配置任务	配置文件模板	拓扑说明
使用 TiCDC 进行增量同步	部署 TiCDC 拓扑架构	简单 TiCDC 配置模板 详细 TiCDC 配置模板	在最小拓扑的基础上部署 TiCDC。TiCDC 支持多种下游 (TiDB/MySQL/MQ)。

场景	配置任务	配置文件模板	拓扑说明
使用 TiDB Binlog 进行增量同步	部署 TiDB Binlog 拓扑架构	简单 TiDB Binlog 配置模板（下游为 MySQL） 简单 TiDB Binlog 配置模板（下游为 file） 详细 TiDB Binlog 配置模板	在最小拓扑的基础上部署 TiDB Binlog。

场景	配置任务	配置文件模板	拓扑说明
使用 Spark 的 OLAP 业务	部署 TiSpark 拓扑架构	简单 TiSpark 配置模板 详细 TiSpark 配置模板	在最小拓扑的基础上部署 TiSpark 组件。TiSpark 是 PingCAP 为解决用户复杂 OLAP 需求而推出的产品。TiUP cluster 组件对 TiSpark 的支持

场景	配置任务	配置文件模板	拓扑说明
单台机器，多个实例	混合部署拓扑架构	简单混部配置模板详细混部配置模板	也适用于单机多实例需要额外增加目录、端口、资源配比、label 等配置的场景。

场景	配置任务	配置文件模板	拓扑说明
跨机房部署 TiDB 集群	跨机房部署拓扑架构	跨机房配置模板	以典型的两地三中心架构为例，介绍跨机房部署架构，以及需要注意的关键设置。

注意：

- 对于需要全局生效的参数，请在配置文件中 `server_configs` 的对应组件下配置。

- 对于需要某个节点生效的参数，请在具体节点的 config 中配置。
- 配置的层次结构使用 . 表示。如：log.slow-threshold。更多格式参考 TiUP 配置参数模版。
- 如果需要指定在目标机创建的用户组名，可以参考这个例子。

更多参数说明，请参考：

- TiDB config.toml.example
- TiKV config.toml.example
- PD config.toml.example
- TiFlash config.toml.example

5.4.1.4 第 4 步：执行部署命令

注意：

通过 TiUP 进行集群部署可以使用密钥或者交互密码方式来进行安全认证：

- 如果是密钥方式，可以通过 -i 或者 –identity_file 来指定密钥的路径。
- 如果是密码方式，可以通过 -p 进入密码交互窗口。
- 如果已经配置免密登录目标机，则不需填写认证。

一般情况下 TiUP 会在目标机器上创建 topology.yaml 中约定的用户和组，以下情况例外：

- topology.yaml 中设置的用户名在目标机器上已存在。
- 在命令行上使用了参数 –skip-create-user 明确指定跳过创建用户的步骤。

执行部署命令前，先使用 check 及 check --apply 命令检查和自动修复集群存在的潜在风险：

1. 检查集群存在的潜在风险：

```
tiup cluster check ./topology.yaml --user root [-p] [-i /home/root/.ssh/gcp_rsa]
```

2. 自动修复集群存在的潜在风险：

```
tiup cluster check ./topology.yaml --apply --user root [-p] [-i /home/root/.ssh/gcp_rsa]
```

3. 部署 TiDB 集群：

```
tiup cluster deploy tidb-test v7.0.0 ./topology.yaml --user root [-p] [-i /home/root/.ssh/
    ↪ gcp_rsa]
```

以上部署示例中：

- `tidb-test` 为部署的集群名称。
- `v7.0.0` 为部署的集群版本，可以通过执行 `tiup list tidb` 来查看 TiUP 支持的最新可用版本。
- 初始化配置文件为 `topology.yaml`。
- `--user root` 表示通过 root 用户登录到目标主机完成集群部署，该用户需要有 ssh 到目标机器的权限，并且在目标机器有 sudo 权限。也可以用其他有 ssh 和 sudo 权限的用户完成部署。
- [-i] 及 [-p] 为可选项，如果已经配置免密登录目标机，则不需填写。否则选择其一即可，[-i] 为可登录到目标机的 root 用户（或 –user 指定的其他用户）的私钥，也可使用 [-p] 交互式输入该用户的密码。

预期日志结尾输出 `Deployed cluster \`tidb-test\` successfully` 关键词，表示部署成功。

5.4.1.5 第 5 步：查看 TiUP 管理的集群情况

```
tiup cluster list
```

TiUP 支持管理多个 TiDB 集群，该命令会输出当前通过 TiUP cluster 管理的所有集群信息，包括集群名称、部署用户、版本、密钥信息等。

5.4.1.6 第 6 步：检查部署的 TiDB 集群情况

例如，执行如下命令检查 `tidb-test` 集群情况：

```
tiup cluster display tidb-test
```

预期输出包括 `tidb-test` 集群中实例 ID、角色、主机、监听端口和状态（由于还未启动，所以状态为 Down/inactive）、目录信息。

5.4.1.7 第 7 步：启动集群

安全启动是 TiUP cluster 从 v1.9.0 起引入的一种新的启动方式，采用该方式启动数据库可以提高数据库安全性。推荐使用安全启动。

安全启动后，TiUP 会自动生成 TiDB root 用户的密码，并在命令行界面返回密码。

> 注意：
>
> - 使用安全启动方式后，不能通过无密码的 root 用户登录数据库，你需要记录命令行返回的密码进行后续操作。
> - 该自动生成的密码只会返回一次，如果没有记录或者忘记该密码，请参照忘记 root 密码修改密码。

方式一：安全启动

```
tiup cluster start tidb-test --init
```

预期结果如下，表示启动成功。

```
Started cluster `tidb-test` successfully.
The root password of TiDB database has been changed.
The new password is: 'y_+3Hwp=*AWz8971s6'.
Copy and record it to somewhere safe, it is only displayed once, and will not be stored.
The generated password can NOT be got again in future.
```

方式二：普通启动

```
tiup cluster start tidb-test
```

预期结果输出 `Started cluster `tidb-test` successfully`，表示启动成功。使用普通启动方式后，可通过无密码的 root 用户登录数据库。

5.4.1.8 第 8 步：验证集群运行状态

```
tiup cluster display tidb-test
```

预期结果输出：各节点 Status 状态信息为 Up 说明集群状态正常。

5.4.1.9 探索更多

如果你已同时部署了TiFlash，接下来可参阅以下文档：

- 使用 TiFlash
- TiFlash 集群运维
- TiFlash 报警规则与处理方法
- TiFlash 常见问题

如果你已同时部署了TiCDC，接下来可参阅以下文档：

- Changefeed 概述
- 管理 Changefeed
- TiCDC 故障处理
- TiCDC 常见问题

5.4.2 在 Kubernetes 上部署 TiDB

你可以使用 TiDB Operator 在 Kubernetes 上部署 TiDB。TiDB Operator 是 Kubernetes 上的 TiDB 集群自动运维系统，提供包括部署、升级、扩缩容、备份恢复、配置变更的 TiDB 全生命周期管理。借助 TiDB Operator，TiDB 可以无缝运行在公有云或自托管的 Kubernetes 集群上。

TiDB Operator 的文档目前独立于 TiDB 文档。要查看如何在 Kubernetes 上部署 TiDB 的详细步骤，请参阅对应版本的 TiDB Operator 文档：

- TiDB on Kubernetes 用户文档

5.5 验证集群运行状态

在部署完一套 TiDB 集群后，需要检查集群是否正常运行。本文介绍如何通过 TiUP 命令、TiDB Dashboard 和 Grafana 检查集群状态，以及如何登录 TiDB 数据库执行简单的 SQL 操作。

5.5.1 通过 TiUP 检查集群状态

检查集群状态的命令是 `tiup cluster display <cluster-name>`，例如：

```
tiup cluster display tidb-test
```

预期结果输出：各节点 Status 状态信息为 Up 说明集群状态正常。

5.5.2 通过 TiDB Dashboard 和 Grafana 检查集群状态

本节介绍如何通过TiDB Dashboard 和 Grafana 检查集群状态。

5.5.2.1 查看 TiDB Dashboard 检查 TiDB 集群状态

1. 通过 `{pd-ip}:{pd-port}/dashboard` 登录 TiDB Dashboard，登录用户和口令为 TiDB 数据库 `root` 用户和口令。如果你修改过数据库的 `root` 密码，则以修改后的密码为准，默认密码为空。

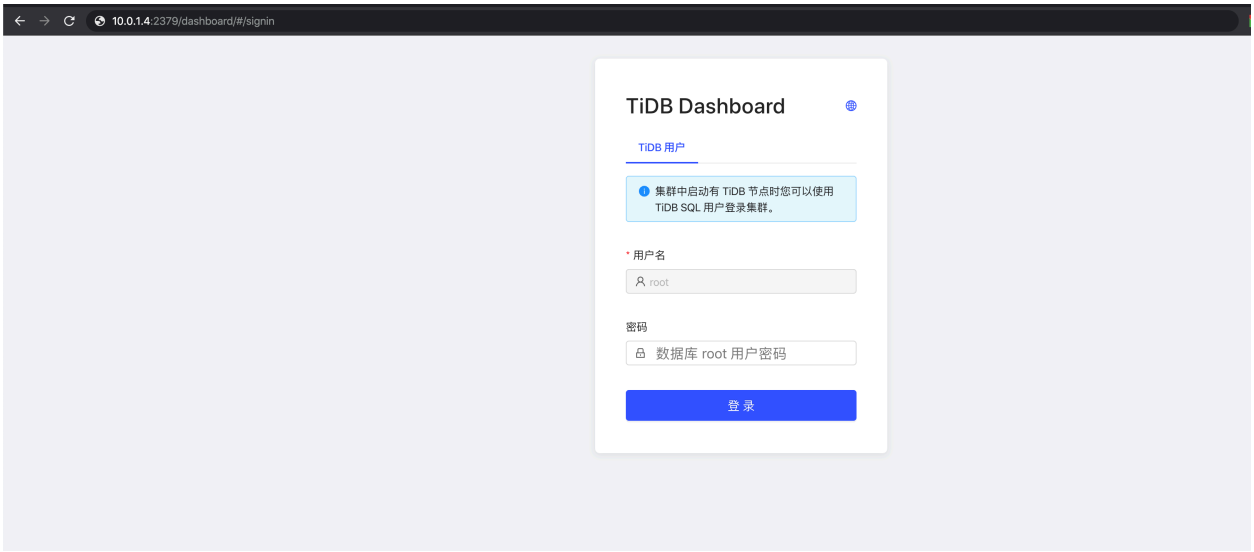


图 23: TiDB-Dashboard

2. 主页面显示 TiDB 集群中节点信息

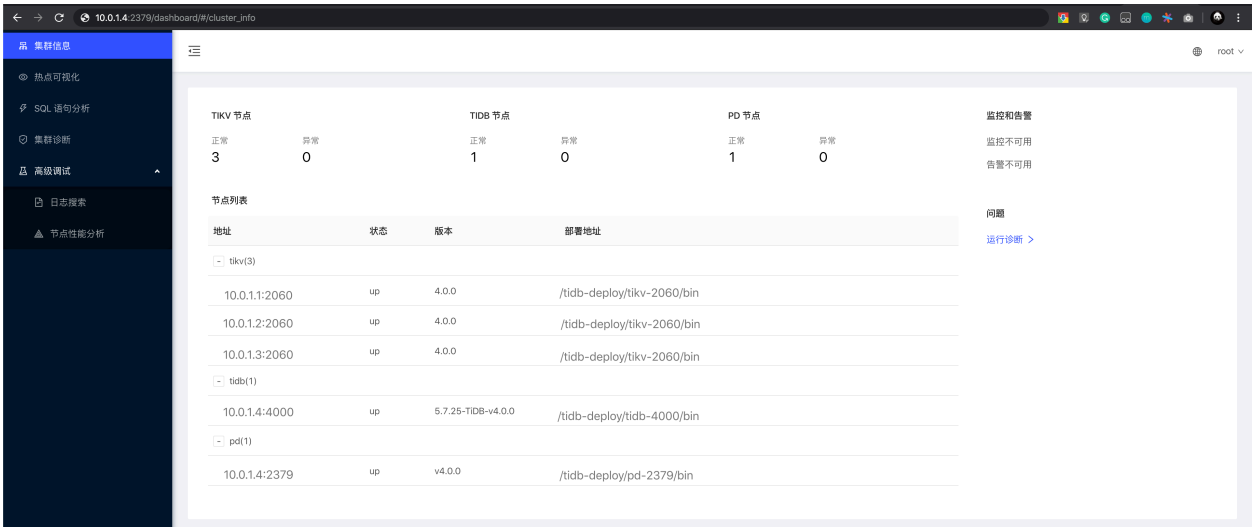


图 24: TiDB-Dashboard-status

5.5.2.2 查看 Grafana 监控 Overview 页面检查 TiDB 集群状态

- 通过 `{Grafana-ip}:3000` 登录 Grafana 监控，默认用户名及密码为 `admin/admin`。
- 点击 Overview 监控页面检查 TiDB 端口和负载监控信息。

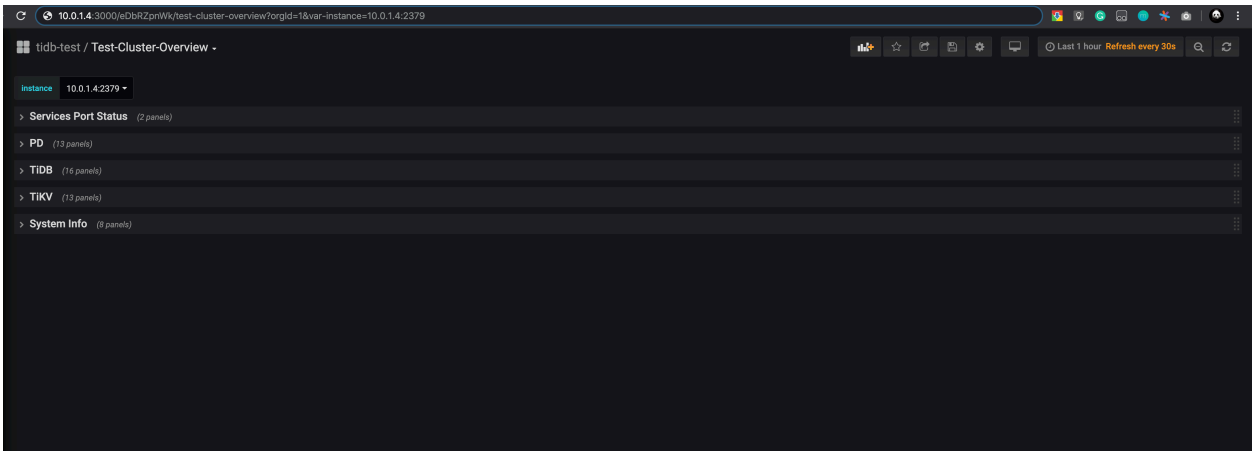


图 25: Grafana-overview

5.5.3 登录数据库执行简单 DML/DDL 操作和查询 SQL 语句

> **注意：**
>
> 登录数据库前，你需要安装 MySQL 客户端。

执行以下命令登录数据库：

```
mysql -u root -h ${tidb_server_host_IP_address} -P 4000
```

其中，${tidb_server_host_IP_address} 是在初始化集群拓扑文件时为 tidb_servers 配置的 IP 地址之一，例如 10.0.1.7。

输出下列信息表示登录成功：

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3
Server version: 5.7.25-TiDB-v5.0.0 TiDB Server (Apache License 2.0) Community Edition, MySQL 5.7
    ↪ compatible

Copyright (c) 2000, 2015, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
```

5.5.3.1 数据库操作

- 检查 TiDB 版本

```
select tidb_version()\G
```

预期结果输出：

```
*************************** 1. row ***************************
tidb_version(): Release Version: v5.0.0
Edition: Community
Git Commit Hash: 689a6b6439ae7835947fcaccf329a3fc303986cb
Git Branch: HEAD
UTC Build Time: 2020-05-28 11:09:45
GoVersion: go1.13.4
Race Enabled: false
TiKV Min Version: v3.0.0-60965b006877ca7234adaced7890d7b029ed1306
Check Table Before Drop: false
1 row in set (0.00 sec)
```

- 创建 PingCAP database

```
create database pingcap;
```

```
Query OK, 0 rows affected (0.10 sec)
```

```
use pingcap;
```

预期输出

```
Database changed
```

- 创建 tab_tidb 表

```
CREATE TABLE `tab_tidb` (
`id` int(11) NOT NULL AUTO_INCREMENT,
`name` varchar(20) NOT NULL DEFAULT '',
`age` int(11) NOT NULL DEFAULT 0,
`version` varchar(20) NOT NULL DEFAULT '',
PRIMARY KEY (`id`),
KEY `idx_age` (`age`));
```

预期输出

```
Query OK, 0 rows affected (0.11 sec)
```

- 插入数据

```
insert into `tab_tidb` values (1,'TiDB',5,'TiDB-v5.0.0');
```

预期输出

```
Query OK, 1 row affected (0.03 sec)
```

- 查看 tab_tidb 结果

```
select * from tab_tidb;
```

预期输出

```
+----+------+-----+-------------+
| id | name | age | version     |
+----+------+-----+-------------+
|  1 | TiDB |   5 | TiDB-v5.0.0 |
+----+------+-----+-------------+
1 row in set (0.00 sec)
```

- 查看 TiKV store 状态、store_id、存储情况以及启动时间

```
select STORE_ID,ADDRESS,STORE_STATE,STORE_STATE_NAME,CAPACITY,AVAILABLE,UPTIME from
    ↪ INFORMATION_SCHEMA.TIKV_STORE_STATUS;
```

预期输出

```
+--
    ↪ --------+--------------------+-------------+-----------------+----------+-----------+--------------
    ↪
| STORE_ID | ADDRESS            | STORE_STATE | STORE_STATE_NAME | CAPACITY | AVAILABLE |
    ↪ UPTIME              |
+--
    ↪ --------+--------------------+-------------+-----------------+----------+-----------+--------------
    ↪
|        1 | 10.0.1.1:20160 |           0 | Up               | 49.98GiB | 46.3GiB   | 5
    ↪ h21m52.474864026s |
|        4 | 10.0.1.2:20160 |           0 | Up               | 49.98GiB | 46.32GiB  | 5
    ↪ h21m52.522669177s |
|        5 | 10.0.1.3:20160 |           0 | Up               | 49.98GiB | 45.44GiB  | 5
    ↪ h21m52.713660541s |
+--
    ↪ --------+--------------------+-------------+-----------------+----------+-----------+--------------
    ↪
3 rows in set (0.00 sec)
```

- 退出

```
exit
```

预期输出

```
Bye
```

5.6 测试集群性能

5.6.1 如何用 Sysbench 测试 TiDB

建议使用 Sysbench 1.0 或之后的更新版本，可在 Sysbench Release 1.0.20 页面下载。

5.6.1.1 测试方案

5.6.1.1.1 TiDB 配置

升高日志级别，可以减少打印日志数量，对 TiDB 的性能有积极影响。具体在 TiUP 配置文件中加入：

```
server_configs:
  tidb:
    log.level: "error"
```

同时，推荐启用tidb_enable_prepared_plan_cache，并保证 --db-ps-mode 设置为 auto，这样 Sysbench 就可以使用预处理语句。关于 SQL 执行计划缓存的功能及监控，请参考执行计划缓存。

> **注意：**
>
> 不同版本 Sysbench 的 db-ps-mode 参数默认值可能会不同，建议在命令中显式指定。

5.6.1.1.2 TiKV 配置

升高 TiKV 的日志级别同样有利于提高性能表现。

TiKV 集群存在多个 Column Family，包括 Default CF、Write CF 和 LockCF，主要用于存储不同类型的数据。对于 Sysbench 测试，需要关注 Default CF 和 Write CF，导入数据的 Column Family 在 TiDB 集群中的比例是固定的。这个比例是：

Default CF : Write CF = 4 : 1

在 TiKV 中需要根据机器内存大小配置 RocksDB 的 block cache，以充分利用内存。以 40 GB 内存的虚拟机部署一个 TiKV 为例，其 block cache 建议配置如下：

```
server_configs:
  tikv:
    log-level: "error"
    rocksdb.defaultcf.block-cache-size: "24GB"
    rocksdb.writecf.block-cache-size: "6GB"
```

还可以使用共享 block cache 的方式进行设置：

```
server_configs:
  tikv:
    storage.block-cache.capacity: "30GB"
```

更详细的 TiKV 参数调优请参考TiKV 内存参数性能调优。

5.6.1.2 测试过程

> **注意：**
>
> 此文档中的测试并没有使用如 HAproxy 等负载均衡工具。在 TiDB 单一节点上进行 Sysbench 测试，并把结果相加。负载均衡工具和不同版本参数也会影响性能表现。

5.6.1.2.1 Sysbench 配置

以下为 Sysbench 配置文件样例：

```
mysql-host={TIDB_HOST}
mysql-port=4000
```

```
mysql-user=root
mysql-password=password
mysql-db=sbtest
time=600
threads={8, 16, 32, 64, 128, 256}
report-interval=10
db-driver=mysql
```

可根据实际需求调整其参数，其中 TIDB_HOST 为 TiDB server 的 IP 地址（配置文件中不能写多个地址），threads 为测试中的并发连接数，可在“8, 16, 32, 64, 128, 256”中调整，导入数据时，建议设置 threads = 8 或者 16。调整后，将该文件保存为名为 config 的文件。

配置文件参考示例如下：

```
mysql-host=172.16.30.33
mysql-port=4000
mysql-user=root
mysql-password=password
mysql-db=sbtest
time=600
threads=16
report-interval=10
db-driver=mysql
```

5.6.1.2.2 数据导入

> 注意：
>
> 如果 TiDB 启用了乐观事务模型（默认为悲观锁模式），当发现并发冲突时，会回滚事务。将 tidb_disable_txn_auto_retry 设置为 off 会开启事务冲突后的自动重试机制，可以尽可能避免事务冲突报错导致 Sysbench 程序退出的问题。

在数据导入前，需要对 TiDB 进行简单设置。在 MySQL 客户端中执行如下命令：

```
set global tidb_disable_txn_auto_retry = off;
```

然后退出客户端。

重新启动 MySQL 客户端执行以下 SQL 语句，创建数据库 sbtest：

```
create database sbtest;
```

调整 Sysbench 脚本创建索引的顺序。Sysbench 按照“建表 -> 插入数据 -> 创建索引”的顺序导入数据。对于 TiDB 而言，该方式会花费更多的导入时间。你可以通过调整顺序来加速数据的导入。

假设使用的 Sysbench 版本为 1.0.20，可以通过以下两种方式来修改：

1. 直接下载为 TiDB 修改好的 oltp_common.lua 文件，覆盖 /usr/share/sysbench/oltp_common.lua 文件。
2. 将 /usr/share/sysbench/oltp_common.lua 的第 235-240 行移动到第 198 行以后。

> **注意：**
>
> 此操作为可选操作，仅节约了数据导入时间。

命令行输入以下命令，开始导入数据，config 文件为上一步中配置的文件：

```
sysbench --config-file=config oltp_point_select --tables=32 --table-size=10000000 prepare
```

5.6.1.2.3 数据预热与统计信息收集

数据预热可将磁盘中的数据载入内存的 block cache 中，预热后的数据对系统整体的性能有较大的改善，建议在每次重启集群后进行一次数据预热。

```
sysbench --config-file=config oltp_point_select --tables=32 --table-size=10000000 prewarm
```

5.6.1.2.4 Point select 测试命令

```
sysbench --config-file=config oltp_point_select --tables=32 --table-size=10000000 --db-ps-mode=
    ↪ auto --rand-type=uniform run
```

5.6.1.2.5 Update index 测试命令

```
sysbench --config-file=config oltp_update_index --tables=32 --table-size=10000000 --db-ps-mode=
    ↪ auto --rand-type=uniform run
```

5.6.1.2.6 Read-only 测试命令

```
sysbench --config-file=config oltp_read_only --tables=32 --table-size=10000000 --db-ps-mode=auto
    ↪ --rand-type=uniform run
```

5.6.1.3 常见问题

5.6.1.3.1 在高并发压力下，TiDB、TiKV 的配置都合理，为什么整体性能还是偏低？

这种情况可能与使用了 proxy 有关。可以尝试直接对单个 TiDB 加压，将求和后的结果与使用 proxy 的情况进行对比。

以 HAproxy 为例。nbproc 参数可以增加其最大启动的进程数，较新版本的 HAproxy 还支持 nbthread 和 cpu-map 等。这些都可以减少对其性能的不利影响。

5.6.1.3.2 在高并发压力下，为什么 TiKV 的 CPU 利用率依然很低？

TiKV 虽然整体 CPU 偏低，但部分模块的 CPU 可能已经达到了很高的利用率。

TiKV 的其他模块，如 storage readpool、coprocessor 和 gRPC 的最大并发度限制是可以通过 TiKV 的配置文件进行调整的。

通过 Grafana 的 TiKV Thread CPU 监控面板可以观察到其实际使用率。如出现多线程模块瓶颈，可以通过增加该模块并发度进行调整。

5.6.1.3.3 在高并发压力下，TiKV 也未达到 CPU 使用瓶颈，为什么 TiDB 的 CPU 利用率依然很低？

在某些高端设备上，使用的是 NUMA 架构的 CPU，跨 CPU 访问远端内存将极大降低性能。TiDB 默认将使用服务器所有 CPU，goroutine 的调度不可避免地会出现跨 CPU 内存访问。

因此，建议在 NUMA 架构服务器上，部署 n 个 TiDB（n = NUMA CPU 的个数），同时将 TiDB 的 `max-procs` 变量的值设置为与 NUMA CPU 的核数相同。

5.6.2 如何对 TiDB 进行 TPC-C 测试

本文介绍如何对 TiDB 进行 TPC-C 测试。

TPC-C 是一个对 OLTP（联机交易处理）系统进行测试的规范，使用一个商品销售模型对 OLTP 系统进行测试，其中包含五类事务：

- NewOrder – 新订单的生成
- Payment – 订单付款
- OrderStatus – 最近订单查询
- Delivery – 配送
- StockLevel – 库存缺货状态分析

在测试开始前，TPC-C Benchmark 规定了数据库的初始状态，也就是数据库中数据生成的规则，其中 ITEM 表中固定包含 10 万种商品，仓库的数量可进行调整，假设 WAREHOUSE 表中有 W 条记录，那么：

- STOCK 表中应有 W * 10 万条记录（每个仓库对应 10 万种商品的库存数据）
- DISTRICT 表中应有 W * 10 条记录（每个仓库为 10 个地区提供服务）
- CUSTOMER 表中应有 W * 10 * 3000 条记录（每个地区有 3000 个客户）
- HISTORY 表中应有 W * 10 * 3000 条记录（每个客户一条交易历史）
- ORDER 表中应有 W * 10 * 3000 条记录（每个地区 3000 个订单），并且最后生成的 900 个订单被添加到 NEW-ORDER 表中，每个订单随机生成 5 ~ 15 条 ORDER-LINE 记录。

我们将以 1000 WAREHOUSE 为例进行测试。

TPC-C 使用 tpmC 值 (Transactions per Minute) 来衡量系统最大有效吞吐量 (MQTh, Max Qualified Throughput)，其中 Transactions 以 NewOrder Transaction 为准，即最终衡量单位为每分钟处理的新订单数。

本文使用 go-tpc 作为 TPC-C 测试实现，可以通过TiUP 命令下载测试程序：

```
tiup install bench
```

关于 TiUP Bench 组件的详细用法可参考TiUP Bench。

假设已部署 TiDB 集群，其中 TiDB 节点部署在 172.16.5.140、172.16.5.141 实例上，端口都为 4000，可按如下步骤进行 TPC-C 测试。

5.6.2.1 导入数据

导入数据通常是整个 TPC-C 测试中最耗时，也是最容易出问题的阶段。

在 shell 中运行 TiUP 命令：

```
tiup bench tpcc -H 172.16.5.140,172.16.5.141 -P 4000 -D tpcc --warehouses 1000 --threads 20
    ↪ prepare
```

基于不同的机器配置，这个过程可能会持续几个小时。如果是小型集群，可以使用较小的 WAREHOUSE 值进行测试。

数据导入完成后，可以通过命令 `tiup bench tpcc -H 172.16.5.140 -P 4000 -D tpcc --warehouses 4 check` 验证数据正确性。

5.6.2.2 运行测试

运行测试的命令是：

```
tiup bench tpcc -H 172.16.5.140,172.16.5.141 -P 4000 -D tpcc --warehouses 1000 --threads 100 --
    ↪ time 10m run
```

运行过程中控制台上会持续打印测试结果：

```
[Current] NEW_ORDER - Takes(s): 4.6, Count: 5, TPM: 65.5, Sum(ms): 4604, Avg(ms): 920, 90th(ms):
    ↪ 1500, 99th(ms): 1500, 99.9th(ms): 1500
[Current] ORDER_STATUS - Takes(s): 1.4, Count: 1, TPM: 42.2, Sum(ms): 256, Avg(ms): 256, 90th(ms)
    ↪ : 256, 99th(ms): 256, 99.9th(ms): 256
[Current] PAYMENT - Takes(s): 6.9, Count: 5, TPM: 43.7, Sum(ms): 2208, Avg(ms): 441, 90th(ms):
    ↪ 512, 99th(ms): 512, 99.9th(ms): 512
[Current] STOCK_LEVEL - Takes(s): 4.4, Count: 1, TPM: 13.8, Sum(ms): 224, Avg(ms): 224, 90th(ms):
    ↪  256, 99th(ms): 256, 99.9th(ms): 256
...
```

运行结束后，会打印测试统计结果：

```
[Summary] DELIVERY - Takes(s): 455.2, Count: 32, TPM: 4.2, Sum(ms): 44376, Avg(ms): 1386, 90th(ms
    ↪ ): 2000, 99th(ms): 4000, 99.9th(ms): 4000
[Summary] DELIVERY_ERR - Takes(s): 455.2, Count: 1, TPM: 0.1, Sum(ms): 953, Avg(ms): 953, 90th(ms
    ↪ ): 1000, 99th(ms): 1000, 99.9th(ms): 1000
[Summary] NEW_ORDER - Takes(s): 487.8, Count: 314, TPM: 38.6, Sum(ms): 282377, Avg(ms): 899, 90th
    ↪ (ms): 1500, 99th(ms): 1500, 99.9th(ms): 1500
[Summary] ORDER_STATUS - Takes(s): 484.6, Count: 29, TPM: 3.6, Sum(ms): 8423, Avg(ms): 290, 90th(
    ↪ ms): 512, 99th(ms): 1500, 99.9th(ms): 1500
```

```
[Summary] PAYMENT - Takes(s): 490.1, Count: 321, TPM: 39.3, Sum(ms): 144708, Avg(ms): 450, 90th(
    ↪ ms): 512, 99th(ms): 1000, 99.9th(ms): 1500
[Summary] STOCK_LEVEL - Takes(s): 487.6, Count: 41, TPM: 5.0, Sum(ms): 9318, Avg(ms): 227, 90th(
    ↪ ms): 512, 99th(ms): 1000, 99.9th(ms): 1000
```

测试完成之后，也可以运行 `tiup bench tpcc -H 172.16.5.140 -P 4000 -D tpcc --warehouses 4 check` 进行数据正确性验证。

5.6.2.3 清理测试数据

```
tiup bench tpcc -H 172.16.5.140 -P 4000 -D tpcc --warehouses 4 cleanup
```

5.6.3 如何对 TiDB 进行 CH-benCHmark 测试

本文介绍如何对 TiDB 进行 CH-benCHmark 测试。

CH-benCHmark 是包含 TPC-C 和 TPC-H 的混合负载，也是用于测试 HTAP 系统的最常见负载。更多信息，请参考 The mixed workload CH-benCHmark。

在进行 CH-benCHmark 测试前，你需要先部署 TiDB 的 HTAP 组件TiFlash。部署 TiFlash 并创建 TiFlash 副本后，对于 TPC-C 联机交易数据，系统将实时同步最新的数据到 TiFlash 组件；TiDB 优化器会自动将 TPC-H 负载的 OLAP 查询下推到 TiFlash MPP 引擎进行高效执行。

本文使用 go-tpc 作为 CH 测试实现，可以通过TiUP 命令下载测试程序：

```
tiup install bench
```

关于 TiUP Bench 组件的详细用法可参考TiUP Bench。

5.6.3.1 导入数据

5.6.3.1.1 导入 TPC-C 数据

导入数据通常是整个 TPC-C 测试中最耗时、也是最容易出问题的阶段。

本文以 1000 WAREHOUSE 为例，在 shell 中运行以下 TiUP 命令进行数据导入和测试。注意你需要将本文中的 172.16.5.140 和 4000 替换为你实际的 TiDB host 和 port 值。

```
tiup bench tpcc -H 172.16.5.140 -P 4000 -D tpcc --warehouses 1000 prepare -T 32
```

基于不同的机器配置，数据导入过程可能会持续几个小时。如果是小型集群，可以使用较小的 WAREHOUSE 值进行测试。

数据导入完成后，可以通过命令 `tiup bench tpcc -H 172.16.5.140 -P 4000 -D tpcc --warehouses 1000 check` 验证数据正确性。

5.6.3.1.2 导入 TPC-H 所需额外的表和视图

在 shell 中运行 TiUP 命令：

```
tiup bench ch -H 172.16.5.140 -P 4000 -D tpcc prepare
```

日志输出如下：

```
creating nation
creating region
creating supplier
generating nation table
generate nation table done
generating region table
generate region table done
generating suppliers table
generate suppliers table done
creating view revenue1
```

5.6.3.2 创建 TiFlash 副本

部署 TiFlash 后，TiFlash 并不会自动同步 TiKV 数据，你需要执行以下 SQL 语句创建整库的 TiFlash 副本。创建 TiFlash 副本后，系统自动实时同步最新数据到 TiFlash 组件。例如，当集群中部署了两个 TiFlash 节点时，如果将 replica 设置为 2，执行以下 SQL 语句将创建两个 TiFlash 副本。

```
ALTER DATABASE tpcc SET TIFLASH REPLICA 2;
```

可通过如下 SQL 语句确认所有表（通过 WHERE 语句可以指定需要确认的表，去掉 WHERE 语句则查看所有表）的 TiFlash 副本的状态是否完成同步：

```
SELECT * FROM information_schema.tiflash_replica WHERE TABLE_SCHEMA = 'tpcc';
```

查询结果中：

- AVAILABLE 字段表示该表的 TiFlash 副本是否可用。1 代表可用，0 代表不可用。副本状态变为可用之后就不再改变。
- PROGRESS 字段代表同步进度，进度值在 0 到 1 之间，1 代表 TiFlash 副本已经完成同步。

5.6.3.3 搜集统计信息

为了确保优化器能生成最优的执行计划，请执行以下 SQL 语句提前搜集统计信息：

```
analyze table customer;
analyze table district;
analyze table history;
analyze table item;
analyze table new_order;
analyze table order_line;
```

```
analyze table orders;
analyze table stock;
analyze table warehouse;
analyze table nation;
analyze table region;
analyze table supplier;
```

5.6.3.4 运行测试

以 50 TP 并发，1 AP 并发为例，运行以下测试命令：

```
tiup bench ch --host 172.16.5.140 -P4000 --warehouses 1000 run -D tpcc -T 50 -t 1 --time 1h
```

命令运行过程中，控制台上会持续打印测试结果。例如：

```
[Current] NEW_ORDER - Takes(s): 10.0, Count: 13524, TPM: 81162.0, Sum(ms): 998317.6, Avg(ms):
    ↪ 73.9, 50th(ms): 71.3, 90th(ms): 100.7, 95th(ms): 113.2, 99th(ms): 159.4, 99.9th(ms):
    ↪ 209.7, Max(ms): 243.3
[Current] ORDER_STATUS - Takes(s): 10.0, Count: 1132, TPM: 6792.7, Sum(ms): 16196.6, Avg(ms):
    ↪ 14.3, 50th(ms): 13.1, 90th(ms): 24.1, 95th(ms): 27.3, 99th(ms): 37.7, 99.9th(ms): 50.3,
    ↪ Max(ms): 52.4
[Current] PAYMENT - Takes(s): 10.0, Count: 12977, TPM: 77861.1, Sum(ms): 773982.0, Avg(ms): 59.7,
    ↪  50th(ms): 56.6, 90th(ms): 88.1, 95th(ms): 100.7, 99th(ms): 151.0, 99.9th(ms): 201.3, Max
    ↪ (ms): 243.3
[Current] STOCK_LEVEL - Takes(s): 10.0, Count: 1134, TPM: 6806.0, Sum(ms): 31220.9, Avg(ms):
    ↪ 27.5, 50th(ms): 25.2, 90th(ms): 37.7, 95th(ms): 44.0, 99th(ms): 71.3, 99.9th(ms): 117.4,
    ↪ Max(ms): 125.8
[Current] Q11    - Count: 1, Sum(ms): 3682.9, Avg(ms): 3683.6
[Current] DELIVERY - Takes(s): 10.0, Count: 1167, TPM: 7002.6, Sum(ms): 170712.9, Avg(ms): 146.3,
    ↪  50th(ms): 142.6, 90th(ms): 192.9, 95th(ms): 209.7, 99th(ms): 251.7, 99.9th(ms): 335.5,
    ↪ Max(ms): 385.9
[Current] NEW_ORDER - Takes(s): 10.0, Count: 13238, TPM: 79429.5, Sum(ms): 1010795.3, Avg(ms):
    ↪ 76.4, 50th(ms): 75.5, 90th(ms): 104.9, 95th(ms): 117.4, 99th(ms): 159.4, 99.9th(ms):
    ↪ 234.9, Max(ms): 352.3
[Current] ORDER_STATUS - Takes(s): 10.0, Count: 1224, TPM: 7350.6, Sum(ms): 17874.1, Avg(ms):
    ↪ 14.6, 50th(ms): 13.6, 90th(ms): 23.1, 95th(ms): 27.3, 99th(ms): 37.7, 99.9th(ms): 54.5,
    ↪ Max(ms): 60.8
[Current] PAYMENT - Takes(s): 10.0, Count: 12650, TPM: 75901.1, Sum(ms): 761981.3, Avg(ms): 60.3,
    ↪  50th(ms): 56.6, 90th(ms): 88.1, 95th(ms): 104.9, 99th(ms): 159.4, 99.9th(ms): 218.1, Max
    ↪ (ms): 318.8
[Current] STOCK_LEVEL - Takes(s): 10.0, Count: 1179, TPM: 7084.9, Sum(ms): 32829.8, Avg(ms):
    ↪ 27.9, 50th(ms): 26.2, 90th(ms): 37.7, 95th(ms): 44.0, 99th(ms): 71.3, 99.9th(ms): 100.7,
    ↪ Max(ms): 117.4
[Current] Q12    - Count: 1, Sum(ms): 9945.8, Avg(ms): 9944.7
[Current] Q13    - Count: 1, Sum(ms): 1729.6, Avg(ms): 1729.6
...
```

命令运行结束后，控制台会打印测试统计结果。例如：

```
Finished: 50 OLTP workers, 1 OLAP workers
[Summary] DELIVERY - Takes(s): 3599.7, Count: 501795, TPM: 8363.9, Sum(ms): 63905178.8, Avg(ms):
    ↪ 127.4, 50th(ms): 125.8, 90th(ms): 167.8, 95th(ms): 184.5, 99th(ms): 226.5, 99.9th(ms):
    ↪ 318.8, Max(ms): 604.0
[Summary] DELIVERY_ERR - Takes(s): 3599.7, Count: 14, TPM: 0.2, Sum(ms): 1027.7, Avg(ms): 73.4,
    ↪ 50th(ms): 71.3, 90th(ms): 109.1, 95th(ms): 109.1, 99th(ms): 113.2, 99.9th(ms): 113.2, Max
    ↪ (ms): 113.2
[Summary] NEW_ORDER - Takes(s): 3599.7, Count: 5629221, TPM: 93826.9, Sum(ms): 363758020.7, Avg(
    ↪ ms): 64.6, 50th(ms): 62.9, 90th(ms): 88.1, 95th(ms): 100.7, 99th(ms): 130.0, 99.9th(ms):
    ↪ 184.5, Max(ms): 570.4
[Summary] NEW_ORDER_ERR - Takes(s): 3599.7, Count: 20, TPM: 0.3, Sum(ms): 404.2, Avg(ms): 20.2,
    ↪ 50th(ms): 18.9, 90th(ms): 37.7, 95th(ms): 50.3, 99th(ms): 56.6, 99.9th(ms): 56.6, Max(ms)
    ↪ : 56.6
[Summary] ORDER_STATUS - Takes(s): 3599.8, Count: 500318, TPM: 8339.0, Sum(ms): 7135956.6, Avg(ms
    ↪ ): 14.3, 50th(ms): 13.1, 90th(ms): 24.1, 95th(ms): 27.3, 99th(ms): 37.7, 99.9th(ms):
    ↪ 50.3, Max(ms): 385.9
[Summary] PAYMENT - Takes(s): 3599.8, Count: 5380815, TPM: 89684.8, Sum(ms): 269863092.5, Avg(ms)
    ↪ : 50.2, 50th(ms): 48.2, 90th(ms): 75.5, 95th(ms): 88.1, 99th(ms): 125.8, 99.9th(ms):
    ↪ 184.5, Max(ms): 1073.7
[Summary] PAYMENT_ERR - Takes(s): 3599.8, Count: 11, TPM: 0.2, Sum(ms): 313.0, Avg(ms): 28.5, 50
    ↪ th(ms): 10.0, 90th(ms): 67.1, 95th(ms): 67.1, 99th(ms): 88.1, 99.9th(ms): 88.1, Max(ms):
    ↪ 88.1
[Summary] STOCK_LEVEL - Takes(s): 3599.8, Count: 500467, TPM: 8341.5, Sum(ms): 13208726.4, Avg(ms
    ↪ ): 26.4, 50th(ms): 25.2, 90th(ms): 37.7, 95th(ms): 44.0, 99th(ms): 62.9, 99.9th(ms):
    ↪ 96.5, Max(ms): 570.4
[Summary] STOCK_LEVEL_ERR - Takes(s): 3599.8, Count: 2, TPM: 0.0, Sum(ms): 7.6, Avg(ms): 3.7, 50
    ↪ th(ms): 3.1, 90th(ms): 4.7, 95th(ms): 4.7, 99th(ms): 4.7, 99.9th(ms): 4.7, Max(ms): 4.7
tpmC: 93826.9, efficiency: 729.6%
[Summary] Q1      - Count: 11, Sum(ms): 42738.2, Avg(ms): 3885.3
[Summary] Q10     - Count: 11, Sum(ms): 440370.3, Avg(ms): 40034.3
[Summary] Q11     - Count: 11, Sum(ms): 44208.6, Avg(ms): 4018.7
[Summary] Q12     - Count: 11, Sum(ms): 105320.3, Avg(ms): 9574.6
[Summary] Q13     - Count: 11, Sum(ms): 19199.5, Avg(ms): 1745.4
[Summary] Q14     - Count: 11, Sum(ms): 84582.1, Avg(ms): 7689.5
[Summary] Q15     - Count: 11, Sum(ms): 271944.8, Avg(ms): 24722.8
[Summary] Q16     - Count: 11, Sum(ms): 183894.9, Avg(ms): 16718.1
[Summary] Q17     - Count: 11, Sum(ms): 89018.9, Avg(ms): 8092.7
[Summary] Q18     - Count: 10, Sum(ms): 767814.5, Avg(ms): 76777.6
[Summary] Q19     - Count: 10, Sum(ms): 17099.1, Avg(ms): 1709.8
[Summary] Q2      - Count: 11, Sum(ms): 53513.6, Avg(ms): 4865.2
[Summary] Q20     - Count: 10, Sum(ms): 73717.7, Avg(ms): 7372.1
[Summary] Q21     - Count: 10, Sum(ms): 166001.4, Avg(ms): 16601.1
[Summary] Q22     - Count: 10, Sum(ms): 48268.4, Avg(ms): 4827.7
[Summary] Q3      - Count: 11, Sum(ms): 31110.1, Avg(ms): 2828.5
```

```
[Summary] Q4      - Count: 11, Sum(ms): 83814.2, Avg(ms): 7619.3
[Summary] Q5      - Count: 11, Sum(ms): 368301.0, Avg(ms): 33483.5
[Summary] Q6      - Count: 11, Sum(ms): 61702.5, Avg(ms): 5608.9
[Summary] Q7      - Count: 11, Sum(ms): 158928.2, Avg(ms): 14446.3
```

测试完成之后，也可以运行 `tiup bench tpcc -H 172.16.5.140 -P 4000 -D tpcc --warehouses 1000 check` 验证数据正确性。

6 数据迁移

6.1 数据迁移概述

本文档总体介绍可用于 TiDB 的数据迁移方案。数据迁移方案如下：

- 全量数据迁移。
  - 数据导入：使用 TiDB Lightning 将 Aurora Snapshot，CSV 文件或 Mydumper SQL 文件的数据全量导入到 TiDB 集群。
  - 数据导出：使用 Dumpling 将 TiDB 集群的数据全量导出为 CSV 文件或 Mydumper SQL 文件，从而更好地配合从 MySQL 数据库或 MariaDB 数据库进行数据迁移。
  - TiDB DM (Data migration) 也提供了适合小规模数据量数据库（例如小于 1 TiB）的全量数据迁移功能。
- 快速初始化 TiDB 集群：TiDB Lightning 提供的快速导入功能可以实现快速初始化 TiDB 集群的指定表的效果。请注意，使用快速初始化 TiDB 集群的功能对 TiDB 集群的影响极大，在进行初始化的过程中，TiDB 集群不支持对外访问。
- 增量数据迁移：使用 TiDB DM 从 MySQL，MariaDB 或 Aurora 同步 Binlog 到 TiDB，该功能可以极大降低业务迁移过程中停机窗口时间。
- TiDB 集群复制：TiDB 支持备份恢复功能，该功能可以实现将 TiDB 的某个快照初始化到另一个全新的 TiDB 集群。
- TiDB 集群增量数据同步：TiCDC 支持同构数据库之间的灾备场景，能够在灾难发生时保证主备集群数据的最终一致性。目前该场景仅支持 TiDB 作为主备集群。

根据迁移数据所在数据库类型、部署位置、业务数据规模大小、业务需求等因素，会有不同数据迁移选择。下面展示一些常用的数据迁移场景，方便用户依据这些线索选择到最适合自己的数据迁移方案。

6.1.1 迁移 Aurora MySQL 到 TiDB

从 Aurora 迁移数据到部署在 AWS 的 TiDB 集群，数据迁移可以分为全量迁移和增量迁移两个步骤进行。请根据你的业务需求选择相应的步骤。

- 从 Aurora 迁移数据到 TiDB

6.1.2 迁移 MySQL 到 TiDB

如果你没有使用 Cloud storage (S3) 服务，而且网络联通和延迟情况良好，那么从 MySQL 迁移数据到 TiDB 时可以参照下面的方案。

- 从小数据量 MySQL 迁移数据到 TiDB

如果你对数据迁移速度有要求，或者数据规模特别大（例如大于 1 TiB），并且禁止 TiDB 集群在迁移期间有其他业务写入，那么你可以先使用 Lightning 进行快速导入，然后根据业务需要选择是否使用 DM 进行增量数据 (Binlog) 同步。

- 从大数据量 MySQL 迁移数据到 TiDB

6.1.3 分库分表 MySQL 合并迁移到 TiDB

如果你的业务使用了基于 MySQL 分库的方案来存储数据，业务数据从 MySQL 迁移到 TiDB 后，合并这些分表数据到一张合并，那么你可以使用 DM 进行分表合并迁移。

- 从小数据量分库分表 MySQL 合并迁移数据到 TiDB

如果分表数据总规模特别大（例如大于 1 TiB），并且禁止 TiDB 集群在迁移期间有其他业务写入，那么你可以使用 Lightning 对分表数据进行快速合并导入，然后根据业务需要选择是否使用 DM 进行增量数据 (Binlog) 的分表同步。

- 从大数据量分库分表 MySQL 合并迁移数据到 TiDB

6.1.4 从文件迁移数据到 TiDB

- 从 CSV 文件迁移数据到 TiDB
- 从 SQL 文件迁移数据到 TiDB
- 从 Parquet 文件迁移数据到 TiDB

6.1.5 TiDB 集群增量数据同步

可以使用 TiCDC 进行 TiDB 集群间的增量数据同步。详情请参考TiCDC 简介。

6.1.6 复杂迁移场景

DM 在实时同步过程中，多个已有特性可以使得同步过程更加灵活，适应各类业务需求：

- 上游使用 pt/gh-ost 工具的持续同步场景
- 下游存在更多列的迁移场景
- 如何过滤 binlog 事件
- 如何通过 SQL 表达式过滤 binlog

6.2 数据迁移工具概览

TiDB 提供了丰富的数据迁移相关的工具，用于全量迁移、增量迁移、备份恢复、数据同步等多种场景。

本文介绍了使用这些工具的场景、支持的上下游、优势和相关限制等信息。请根据你的需求选择合适的工具。

6.2.1 TiDB Data Migration (DM)

使用场景	用于将数据从与 MySQL 协议兼容的数据库迁移到 TiDB
上游	MySQL，MariaDB，Aurora
下游	TiDB

| 主要优势 |

一体化的数据迁移任务管理工具，支持全量迁移和增量同步

支持对表与操作进行过滤

支持分库分表的合并迁移

使用限制 | 数据导入速度与 TiDB Lightning 的Logical Import Mode 大致相同，而比 TiDB Lightning 的Physical Import Mode 低很多。建议用于 1 TB 以内的存量数据迁移。 |

6.2.2 TiDB Lightning

使用场景	用于将数据全量导入到 TiDB

| 上游（输入源文件） |

Dumpling 输出的文件

从 Amazon Aurora 或 Apache Hive 导出的 Parquet 文件

CSV 文件

从本地盘或 Amazon S3 云盘读取数据

| | 下游 | TiDB | | 主要优势 |

支持快速导入大量数据，实现快速初始化 TiDB 集群的指定表

支持断点续传

支持数据过滤

| | 使用限制 |

如果使用Physical Import Mode 进行数据导入，TiDB Lightning 运行后，TiDB 集群将无法正常对外提供服务。

如果你不希望 TiDB 集群的对外服务受到影响，可以参考 TiDB Lightning Logical Import Mode 中的硬件需求与部署方式进行数据导入。

|

6.2.3 Dumping

使用场景	用于将数据从 MySQL/TiDB 进行全量导出
上游	MySQL，TiDB
下游（输出文件）	SQL，CSV

| 主要优势 |

支持全新的 table-filter，筛选数据更加方便

支持导出到 Amazon S3 云盘

| | 使用限制 |

如果导出后计划往非 TiDB 的数据库恢复，建议使用 Dumpling。

如果导出后计划往另一个 TiDB 恢复，建议使用BR。

|

6.2.4 TiCDC

使用场景	通过拉取 TiKV 变更日志实现的 TiDB 增量数据同步工具，具有将数据还原到与上游任意 TSO 一致状态的能力，支持
上游	TiDB
下游	TiDB，MySQL，Kafka，Confluent
主要优势	提供开放数据协议 (TiCDC Open Protocol)。

| 使用限制 | TiCDC 只能同步至少存在一个有效索引的表。暂不支持以下场景：

单独使用 RawKV 的 TiKV 集群。

在 TiDB 中创建 SEQUENCE 的 DDL 操作和 SEQUENCE 函数。

|

6.2.5 Backup & Restore (BR)

使用场景	通过对大数据量的 TiDB 集群进行数据备份和恢复，实现数据迁移
上游	TiDB
下游（输出文件）	SST，backup.meta 文件，backup.lock 文件

| 主要优势 |

适用于向另一个 TiDB 迁移数据。

支持数据冷备份到外部存储，可以用于灾备恢复。

| | 使用限制 |

BR 恢复到 TiCDC / Drainer 的上游集群时，恢复数据无法由 TiCDC / Drainer 同步到下游。

BR 只支持在 new_collations_enabled_on_first_bootstrap 开关值相同的集群之间进行操作。

|

6.2.6 sync-diff-inspector

使用场景	用于校验 MySQL/TiDB 中两份数据的一致性
上游	TiDB，MySQL
下游	TiDB，MySQL
主要优势	提供了修复数据的功能，适用于修复少量不一致的数据。

| 使用限制 |

对于 MySQL 和 TiDB 之间的数据同步不支持在线校验。

不支持 JSON、 BIT、 BINARY、 BLOB 等类型的数据。

|

6.2.7 使用 TiUP 快速安装

从 TiDB 4.0 开始，TiUP 作为软件包管理器，帮助你轻松管理 TiDB 生态系统中的不同集群组件。现在你可以只用一个 TiUP 命令行来管理任何组件。

6.2.7.1 第 1 步：安装 TiUP

```
curl --proto '=https' --tlsv1.2 -sSf https://tiup-mirrors.pingcap.com/install.sh | sh
```

重新声明全局环境变量：

```
source ~/.bash_profile
```

6.2.7.2 第 2 步：安装组件

你可以通过以下命令查看所有可用组件:

```
tiup list
```

以下输出为所有可用组件：

```
Available components:
Name            Owner    Description
----            -----    -----------
bench           pingcap  Benchmark database with different workloads
br              pingcap  TiDB/TiKV cluster backup restore tool
cdc             pingcap  CDC is a change data capture tool for TiDB
client          pingcap  Client to connect playground
cluster         pingcap  Deploy a TiDB cluster for production
ctl             pingcap  TiDB controller suite
dm              pingcap  Data Migration Platform manager
dmctl           pingcap  dmctl component of Data Migration Platform
errdoc          pingcap  Document about TiDB errors
pd-recover      pingcap  PD Recover is a disaster recovery tool of PD, used to recover the PD
    ↪ cluster which cannot start or provide services normally
playground      pingcap  Bootstrap a local TiDB cluster for fun
tidb            pingcap  TiDB is an open source distributed HTAP database compatible with the
    ↪ MySQL protocol
tidb-lightning  pingcap  TiDB Lightning is a tool used for fast full import of large amounts of
    ↪ data into a TiDB cluster
tiup            pingcap  TiUP is a command-line component management tool that can help to
    ↪ download and install TiDB platform components to the local system
```

选择所需要的组件进行安装

```
tiup install dumpling tidb-lightning
```

> Note:
>
> 如果需要安装特定版本，可以使用 `tiup install <component>[:version]` 命令.

6.2.7.3 第 3 步：更新 TiUP 及组件 (可选)

建议先查看新版本的更新日志及兼容性说明：

```
tiup update --self && tiup update dm
```

6.2.8 探索更多

- 离线方式安装 TiUP
- 以二进制包形式安装各工具

6.3 数据迁移场景

6.3.1 从 Amazon Aurora 迁移数据到 TiDB

本文档介绍如何从 Amazon Aurora 迁移数据到 TiDB，迁移过程采用 DB snapshot，可以节约大量的空间和时间成本。整个迁移包含两个过程：

- 使用 Lightning 导入全量数据到 TiDB
- 使用 DM 持续增量同步到 TiDB（可选）

6.3.1.1 前提条件

- 安装 Dumpling 和 Lightning。
- 获取 Dumpling 所需上游数据库权限。
- 获取 Lightning 所需下游数据库权限。

6.3.1.2 导入全量数据到 TiDB

6.3.1.2.1 第 1 步：导出 Aurora 快照文件到 Amazon S3

1. 在 Aurora 上，执行以下命令，查询并记录当前 binlog 位置：

```
mysql> SHOW MASTER STATUS;
```

你将得到类似以下的输出，请记录 binlog 名称和位置，供后续步骤使用：

```
+------------------+----------+--------------+------------------+-------------------+
| File             | Position | Binlog_Do_DB | Binlog_Ignore_DB | Executed_Gtid_Set |
+------------------+----------+--------------+------------------+-------------------+
| mysql-bin.000002 |    52806 |              |                  |                   |
+------------------+----------+--------------+------------------+-------------------+
1 row in set (0.012 sec)
```

2. 导出 Aurora 快照文件。具体方式请参考 Aurora 的官方文档：Exporting DB snapshot data to Amazon S3.

请注意，上述两步的时间间隔建议不要超过 5 分钟，否则记录的 binlog 位置过旧可能导致增量同步时产生数据冲突。

完成上述两步后，你需要准备好以下信息：

- 创建快照点时，Aurora binlog 的名称及位置。
- 快照文件的 S3 路径，以及具有访问权限的 SecretKey 和 AccessKey。

6.3.1.2.2 第 2 步：导出 schema

因为 Aurora 生成的快照文件并不包含建表语句文件，所以你需要使用 Dumpling 自行导出 schema 并使用 Lightning 在下游创建 schema。你也可以跳过此步骤，并以手动方式在下游自行创建 schema。

运行以下命令，建议使用 `--filter` 参数仅导出所需表的 schema：

```
tiup dumpling --host ${host} --port 3306 --user root --password ${password} --filter 'my_db1.
    ↪ table[12],mydb.*' --consistency none --no-data --output 's3://my-bucket/schema-backup'
```

命令中所用参数描述如下。如需更多信息可参考Dumpling overview。

参数	说明
-u 或 –user	Aurora MySQL 数据库的用户
-p 或 –password	MySQL 数据库的用户密码
-P 或 –port	MySQL 数据库的端口
-h 或 –host	MySQL 数据库的 IP 地址
-t 或 –thread	导出的线程数
-o 或 –output	存储导出文件的目录，支持本地文件路径或外部存储 URI 格式
-r 或 –row	单个文件的最大行数
-F	指定单个文件的最大大小，单位为 MiB，建议值 256 MiB
-B 或 –database	导出指定数据库
-T 或 –tables-list	导出指定数据表
-d 或 –no-data	不导出数据，仅导出 schema
-f 或 –filter	导出能匹配模式的表，不可与 -T 一起使用，语法可参考table filter
–consistency	导出期间的一致性模式，Aurora 只能使用 none，即不加锁 dump，不保证一致性

6.3.1.2.3 第 3 步：编写 Lightning 配置文件

根据以下内容创建 `tidb-lightning.toml` 配置文件：

```
vim tidb-lightning.toml
```

```
[tidb]

### 目标 TiDB 集群信息.
host = ${host}                # 例如：172.16.32.1
port = ${port}                # 例如：4000
user = "${user_name}"         # 例如："root"
password = "${password}"      # 例如："rootroot"
status-port = ${status-port}  # 表结构信息在从 TiDB 的“状态端口”获取例如：10080
pd-addr = "${ip}:${port}"     # 集群 PD 的地址，lightning 通过 PD 获取部分信息，例如
    ↪ 172.16.31.3:2379。当 backend = "local" 时 status-port 和 pd-addr 必须正确填写，
    ↪ 否则导入将出现异常。

[tikv-importer]
### "local"：默认使用该模式，适用于 TB 级以上大数据量，但导入期间下游 TiDB 无法对外提供服务。
```

```
### "tidb": TB 级以下数据量也可以采用`tidb`后端模式，下游 TiDB 可正常提供服务。
    ↪ 关于后端模式更多信息请参阅：https://docs.pingcap.com/tidb/stable/tidb-lightning-backends
backend = "local"

### 设置排序的键值对的临时存放地址，目标路径必须是一个空目录，目录空间须大于待导入数据集的大小，
    ↪ 建议设为与 `data-source-dir` 不同的磁盘目录并使用闪存介质，独占 IO 会获得更好的导入性能。
sorted-kv-dir = "${path}"

[mydumper]
### 快照文件的地址
data-source-dir = "${s3_path}"  # eg: s3://my-bucket/sql-backup

[[mydumper.files]]
### 解析 parquet 文件所需的表达式
pattern = '(?i)^(?:[^/]*/)*([a-z0-9_]+)\.([a-z0-9_]+)/(?:[^/]*/)*(?:[a-z0-9\-_.]+\.(parquet))$'
schema = '$1'
table = '$2'
type = '$3'
```

如果需要在 TiDB 开启 TLS，请参考TiDB Lightning Configuration。

6.3.1.2.4 第 4 步：导入全量数据到 TiDB

1. 使用 Lightning 在下游 TiDB 建表:

   ```
   tiup tidb-lightning -config tidb-lightning.toml -d 's3://my-bucket/schema-backup'
   ```

2. 运行 tidb-lightning。如果直接在命令行中启动程序，可能会因为 SIGHUP 信号而退出，建议配合 nohup 或 screen 等工具，如：

   将有权限访问该 Amazon S3 后端存储的账号的 SecretKey 和 AccessKey 作为环境变量传入 Lightning 节点。同时还支持从 ~/.aws/credentials 读取凭证文件。

   ```
   export AWS_ACCESS_KEY_ID=${access_key}
   export AWS_SECRET_ACCESS_KEY=${secret_key}
   nohup tiup tidb-lightning -config tidb-lightning.toml > nohup.out 2>&1 &
   ```

3. 导入开始后，可以采用以下任意方式查看进度：

   - 通过 grep 日志关键字 progress 查看进度，默认 5 分钟更新一次。
   - 通过监控面板查看进度，请参考TiDB Lightning 监控。
   - 通过 Web 页面查看进度，请参考Web 界面。

4. 导入完毕后，TiDB Lightning 会自动退出。查看 tidb-lightning.log 日志末尾是否有 the whole procedure completed 信息，如果有，表示导入成功。如果没有，则表示导入遇到了问题，可根据日志中的 error 提示解决遇到的问题。

> **注意：**
>
> 无论导入成功与否，最后一行都会显示 `tidb lightning exit`。它只是表示 TiDB Lightning 正常退出，不代表任务完成。

如果导入过程中遇到问题，请参见TiDB Lightning 常见问题。

6.3.1.3 持续增量同步数据到 TiDB（可选）

6.3.1.3.1 前提条件

- 安装 DM 集群
- 获取 DM 所需上下游数据库权限

6.3.1.3.2 第 1 步：创建数据源

1. 新建 `source1.yaml` 文件，写入以下内容：

```
# 唯一命名，不可重复。
source-id: "mysql-01"

# DM-worker 是否使用全局事务标识符（GTID）拉取 binlog。使用前提是上游 MySQL 已开启 GTID 模式
    ↪ 。若上游存在主从自动切换，则必须使用 GTID 模式。
enable-gtid: false

from:
  host: "${host}"         # 例如：172.16.10.81
  user: "root"
  password: "${password}" # 支持但不推荐使用明文密码，建议使用 dmctl encrypt
      ↪ 对明文密码进行加密后使用
  port: 3306
```

2. 在终端中执行下面的命令，使用 `tiup dmctl` 将数据源配置加载到 DM 集群中:

```
tiup dmctl --master-addr ${advertise-addr} operate-source create source1.yaml
```

该命令中的参数描述如下：

参数	描述
`--master-addr`	dmctl 要连接的集群的任意 DM-master 节点的 `{advertise-addr}`，例如：172.16.10.71:8261
`operate-source create`	向 DM 集群加载数据源

6.3.1.3.3 第 2 步：创建迁移任务

新建 task1.yaml 文件，写入以下内容：

```
### 任务名，多个同时运行的任务不能重名。
name: "test"
### 任务模式，可设为
### full：只进行全量数据迁移
### incremental： binlog 实时同步
### all： 全量 + binlog 迁移
task-mode: "incremental"
### 下游 TiDB 配置信息。
target-database:
  host: "${host}"                  # 例如：172.16.10.83
  port: 4000
  user: "root"
  password: "${password}"          # 支持但不推荐使用明文密码，建议使用 dmctl encrypt
      ↪ 对明文密码进行加密后使用

### 黑白名单全局配置，各实例通过配置项名引用。
block-allow-list:                    # 如果 DM 版本早于 v2.0.0-beta.2 则使用 black-white-list。
  listA:                             # 名称
    do-tables:                       # 需要迁移的上游表的白名单。
    - db-name: "test_db"             # 需要迁移的表的库名。
      tbl-name: "test_table"         # 需要迁移的表的名称。

### 配置数据源
mysql-instances:
  - source-id: "mysql-01"              # 数据源 ID，即 source1.yaml 中的 source-id
    block-allow-list: "listA"          # 引入上面黑白名单配置。
###    syncer-config-name: "global"        # syncer 配置的名称
    meta:                              # `task-mode` 为 `incremental` 且下游数据库的 `checkpoint
        ↪ ` 不存在时 binlog 迁移开始的位置；如果 checkpoint 存在，则以 `checkpoint` 为准。如果
        ↪ `meta` 项和下游数据库的 `checkpoint` 都不存在，则从上游当前最新的 binlog 位置开始迁移
        ↪ 。
      binlog-name: "mysql-bin.000004"   # “Step 1. 导出 Aurora 快照文件到 Amazon S3”
          ↪ 中记录的日志位置，当上游存在主从切换时，必须使用 gtid。
      binlog-pos: 109227
      # binlog-gtid: "09bec856-ba95-11ea-850a-58f2b4af5188:1-9"

### 【可选配置】 如果增量数据迁移需要重复迁移已经在全量数据迁移中完成迁移的数据，则需要开启 safe
    ↪ mode 避免增量数据迁移报错。
   ##  该场景多见于以下情况：全量迁移的数据不属于数据源的一个一致性快照，
       ↪ 随后从一个早于全量迁移数据之前的位置开始同步增量数据。
   # syncers:            # sync 处理单元的运行配置参数。
   #  global:           # 配置名称。
```

```
#    safe-mode: true # 设置为 true，会将来自数据源的 INSERT 改写为 REPLACE，将 UPDATE 改写为
    ↪ DELETE 与 REPLACE，从而保证在表结构中存在主键或唯一索引的条件下迁移数据时可以重复导入
    ↪ DML。在启动或恢复增量复制任务的前 1 分钟内 TiDB DM 会自动启动 safe mode。
```

以上内容为执行迁移的最小任务配置。关于任务的更多配置项，可以参考DM 任务完整配置文件介绍

6.3.1.3.4 第 3 步：启动任务

在你启动数据迁移任务之前，建议使用 check-task 命令检查配置是否符合 DM 的配置要求，以降低后期报错的概率：

```
tiup dmctl --master-addr ${advertise-addr} check-task task.yaml
```

使用 tiup dmctl 执行以下命令启动数据迁移任务。

```
tiup dmctl --master-addr ${advertise-addr} start-task task.yaml
```

该命令中的参数描述如下：

参数	描述
--master-addr	dmctl 要连接的集群的任意 DM-master 节点的 {advertise-addr}，例如：172.16.10.71:8261
start-task	命令用于创建数据迁移任务

如果任务启动失败，可根据返回结果的提示进行配置变更后，再次执行上述命令，重新启动任务。遇到问题请参考故障及处理方法以及常见问题。

6.3.1.3.5 第 4 步：查看任务状态

如需了解 DM 集群中是否存在正在运行的迁移任务及任务状态等信息，可使用 tiup dmctl 执行 query-status 命令进行查询：

```
tiup dmctl --master-addr ${advertise-addr} query-status ${task-name}
```

关于查询结果的详细解读，请参考查询状态。

6.3.1.3.6 第 5 步：监控任务与查看日志

要查看迁移任务的历史状态以及更多的内部运行指标，可参考以下步骤。

如果使用 TiUP 部署 DM 集群时，正确部署了 Prometheus、Alertmanager 与 Grafana，则使用部署时填写的 IP 及端口进入 Grafana，选择 DM 的 dashboard 查看 DM 相关监控项。

DM 在运行过程中，DM-worker、DM-master 及 dmctl 都会通过日志输出相关信息。各组件的日志目录如下：

- DM-master 日志目录：通过 DM-master 进程参数 --log-file 设置。如果使用 TiUP 部署 DM，则日志目录默认位于 /dm-deploy/dm-master-8261/log/。
- DM-worker 日志目录：通过 DM-worker 进程参数 --log-file 设置。如果使用 TiUP 部署 DM，则日志目录默认位于 /dm-deploy/dm-worker-8262/log/。

6.3.1.4 探索更多

- 暂停数据迁移任务
- 恢复数据迁移任务
- 停止数据迁移任务
- 导出和导入集群的数据源和任务配置
- 处理出错的 DDL 语句

6.3.2 从小数据量 MySQL 迁移数据到 TiDB

本文档介绍如何使用 TiDB DM （以下简称 DM ）以全量 + 增量的模式数据到 TiDB。本文所称 “小数据量” 通常指 TiB 级别以下。

一般而言，受到表结构索引数目等信息、硬件以及网络环境影响，迁移速率在 30 ~ 50GB/h 不等。使用 TiDB DM 迁移的流程如下图所示。

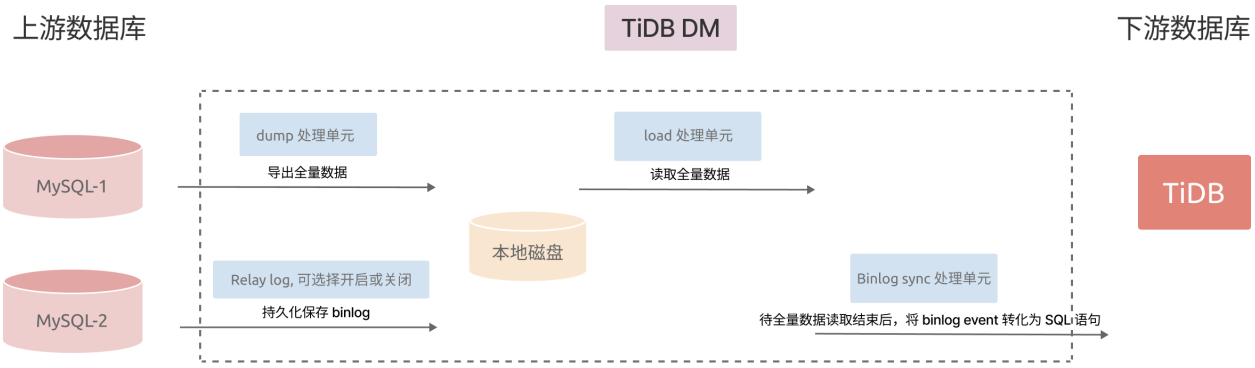


图 26: dm

6.3.2.1 前提条件

- 使用 TiUP 安装 DM 集群
- DM 所需上下游数据库权限

6.3.2.2 第 1 步：创建数据源

首先，新建 `source1.yaml` 文件，写入以下内容：

```
### 唯一命名，不可重复。
source-id: "mysql-01"

### DM-worker 是否使用全局事务标识符 (GTID) 拉取 binlog。使用前提是上游 MySQL 已开启 GTID 模式。
    ↪ 若上游存在主从自动切换，则必须使用 GTID 模式。
enable-gtid: true

from:
```

```
  host: "${host}"         # 例如：172.16.10.81
  user: "root"
  password: "${password}" # 支持但不推荐使用明文密码，建议使用 dmctl encrypt
      ↪ 对明文密码进行加密后使用
  port: 3306
```

其次，在终端中执行下面的命令后，使用 tiup dmctl 将数据源配置加载到 DM 集群中:

```
tiup dmctl --master-addr ${advertise-addr} operate-source create source1.yaml
```

该命令中的参数描述如下：

参数	描述
--master-addr	dmctl 要连接的集群的任意 DM-master 节点的 {advertise-addr}，例如：172.16.10.71:8261
operate-source create	向 DM 集群加载数据源

6.3.2.3 第 2 步：创建迁移任务

新建 task1.yaml 文件，写入以下内容：

```
### 任务名，多个同时运行的任务不能重名。
name: "test"
### 任务模式，可设为
### full：只进行全量数据迁移
### incremental： binlog 实时同步
### all： 全量 + binlog 迁移
task-mode: "all"
### 下游 TiDB 配置信息。
target-database:
  host: "${host}"                  # 例如：172.16.10.83
  port: 4000
  user: "root"
  password: "${password}"          # 支持但不推荐使用明文密码，建议使用 dmctl encrypt
      ↪ 对明文密码进行加密后使用

### 当前数据迁移任务需要的全部上游 MySQL 实例配置。
mysql-instances:
-
  # 上游实例或者复制组 ID。
  source-id: "mysql-01"
  # 需要迁移的库名或表名的黑白名单的配置项名称，用于引用全局的黑白名单配置，全局配置见下面的 `
      ↪ block-allow-list` 的配置。
  block-allow-list: "listA"


### 黑白名单全局配置，各实例通过配置项名引用。
```

```
block-allow-list:
  listA:                                # 名称
    do-tables:                          # 需要迁移的上游表的白名单。
    - db-name: "test_db"                # 需要迁移的表的库名。
      tbl-name: "test_table"            # 需要迁移的表的名称。
```

以上内容为执行迁移的最小任务配置。关于任务的更多配置项，可以参考DM 任务完整配置文件介绍。

6.3.2.4 第 3 步：启动任务

在你启动数据迁移任务之前，建议使用 `check-task` 命令检查配置是否符合 DM 的配置要求，以避免后期报错。

```
tiup dmctl --master-addr ${advertise-addr} check-task task.yaml
```

使用 `tiup dmctl` 执行以下命令启动数据迁移任务。

```
tiup dmctl --master-addr ${advertise-addr} start-task task.yaml
```

该命令中的参数描述如下：

参数	描述
--master-addr	要连接的集群的任意 DM-master 节点的 {advertise-addr}，例如：172.16.10.71:8261
start-task task.yaml	参数用于启动数据迁移任务

如果任务启动失败，可根据返回结果的提示进行配置变更后执行 start-task task.yaml 命令重新启动任务。遇到问题请参考故障及处理方法以及常见问题。

6.3.2.5 第 4 步：查看任务状态

如需了解 DM 集群中是否存在正在运行的迁移任务及任务状态等信息，可使用 `tiup dmctl` 执行 `query-status` 命令进行查询：

```
tiup dmctl --master-addr ${advertise-addr} query-status ${task-name}
```

关于查询结果的详细解读，请参考查询状态。

6.3.2.6 第 5 步：监控任务与查看日志（可选）

要查看迁移任务的历史状态以及更多的内部运行指标，可参考以下步骤。

如果使用 TiUP 部署 DM 集群时，正确部署了 Prometheus、Alertmanager 与 Grafana，则使用部署时填写的 IP 及端口进入 Grafana，选择 DM 的 Dashboard 查看 DM 相关监控项。

DM 在运行过程中，DM-worker、DM-master 及 dmctl 都会通过日志输出相关信息。各组件的日志目录如下：

- DM-master 日志目录：通过 DM-master 进程参数 `--log-file`设置。如果使用 TiUP 部署 DM，则日志目录默认位于 `/dm-deploy/dm-master-8261/log/`。
- DM-worker 日志目录：通过 DM-worker 进程参数 `--log-file` 设置。如果使用 TiUP 部署 DM，则日志目录默认位于 `/dm-deploy/dm-worker-8262/log/`。

6.3.2.7 探索更多

- 暂停数据迁移任务
- 恢复数据迁移任务
- 停止数据迁移任务
- 导出和导入集群的数据源和任务配置
- 处理出错的 DDL 语句

6.3.3 从大数据量 MySQL 迁移数据到 TiDB

通常数据量较低时，使用 DM 进行迁移较为简单，可直接完成全量 + 持续增量迁移工作。但当数据量较大时，DM 较低的数据导入速度 (30~50 GiB/h) 可能令整个迁移周期过长。本文所称“大数据量”通常指 TiB 级别以上。

因此，本文档介绍使用 Dumpling 和 TiDB Lightning 进行全量数据迁移，其本地导入 (local backend) 模式导入速度可达每小时 500 GiB。完成全量数据迁移后，再使用 DM 完成增量数据迁移。

6.3.3.1 前提条件

- 部署 DM 集群。
- 安装 Dumpling 和 TiDB Lightning。
- 配置 DM 所需上下游数据库权限。
- 获取 TiDB Lightning 所需下游数据库权限。
- 获取 Dumpling 所需上游数据库权限。

6.3.3.2 资源要求

操作系统：本文档示例使用的是若干新的、纯净版 CentOS 7 实例，你可以在本地虚拟化一台主机，或在供应商提供的平台上部署一台小型的云虚拟主机。TiDB Lightning 运行过程中，默认会占满 CPU，建议单独部署在一台主机上。如果条件不允许，你可以将 TiDB Lightning 和其他组件（比如 `tikv-server`）部署在同一台机器上，然后设置 `region-concurrency` 配置项的值为逻辑 CPU 数的 75%，以限制 TiDB Lightning 对 CPU 资源的使用。

内存和 CPU：因为 TiDB Lightning 对计算机资源消耗较高，建议分配 64 GB 以上的内存以及 32 核以上的 CPU，而且确保 CPU 核数和内存（GB）比为 1:2 以上，以获取最佳性能。

磁盘空间：

- Dumpling 需要能够储存整个数据源的存储空间，即可以容纳要导出的所有上游表的空间。计算方式参考下游数据库所需空间。
- TiDB Lightning 导入期间，需要临时空间来存储排序键值对，磁盘空间需要至少能存储数据源的最大单表。
- 若全量数据量较大，可适当加长上游 binlog 保存时间，以避免增量同步时缺必要 binlog 导致重做。

说明：目前无法精确计算 Dumpling 从 MySQL 导出的数据大小，但你可以用下面 SQL 语句统计信息表的 `data_length` 字段估算数据量：

```
/* 统计所有 schema 大小，单位 MiB，注意修改 ${schema_name} */
SELECT table_schema,SUM(data_length)/1024/1024 AS data_length,SUM(index_length)/1024/1024 AS
    ↪ index_length,SUM(data_length+index_length)/1024/1024 AS SUM FROM information_schema.
    ↪ tables WHERE table_schema = "${schema_name}" GROUP BY table_schema;

/* 统计最大单表，单位 MiB，注意修改 ${schema_name} */
SELECT table_name,table_schema,SUM(data_length)/1024/1024 AS data_length,SUM(index_length)
    ↪ /1024/1024 AS index_length,SUM(data_length+index_length)/1024/1024 AS SUM from
    ↪ information_schema.tables WHERE table_schema = "${schema_name}" GROUP BY table_name,
    ↪ table_schema ORDER BY SUM DESC LIMIT 5;
```

6.3.3.2.1 目标 TiKV 集群的磁盘空间要求

目标 TiKV 集群必须有足够空间接收新导入的数据。除了标准硬件配置以外，目标 TiKV 集群的总存储空间必须大于数据源大小 × 副本数量 × 2。例如，集群默认使用 3 副本，那么总存储空间需为数据源大小的 6 倍以上。公式中的 2 倍可能难以理解，其依据是以下因素的估算空间占用：

- 索引会占据额外的空间。
- RocksDB 的空间放大效应。

6.3.3.3 第 1 步：从 MySQL 导出全量数据

1. 运行以下命令，从 MySQL 导出全量数据：

```
tiup dumpling -h ${ip} -P 3306 -u root -t 16 -r 200000 -F 256MiB -B my_db1 -f 'my_db1.table
    ↪ [12]' -o 's3://my-bucket/sql-backup'
```

Dumpling 默认导出数据格式为 SQL 文件，你也可以通过设置 `--filetype` 指定导出文件的类型。

以上命令行中用到的参数描述如下。要了解更多 Dumpling 参数，请参考Dumpling 使用文档。

参数	说明
-u 或 --user	MySQL 数据库的用户
-p 或 --password	MySQL 数据库的用户密码
-P 或 --port	MySQL 数据库的端口
-h 或 --host	MySQL 数据库的 IP 地址
-t 或 --thread	导出的线程数。增加线程数会增加 Dumpling 并发度提高导出速度，但也会加大数据库内存消耗，因此不宜
-o 或 --output	存储导出文件的目录，支持本地文件路径或外部存储 URI 格式
-r 或 --row	用于指定单个文件的最大行数，指定该参数后 Dumpling 会开启表内并发加速导出，同时减少内存使用
-F	指定单个文件的最大大小，单位为 MiB。强烈建议使用 -F 参数以避免单表过大导致备份过程中断
-B 或 --database	导出指定数据库
-f 或 --filter	导出能匹配模式的表，语法可参考table-filter

请确保 ${data-path} 的空间可以容纳要导出的所有上游表，计算方式参考下游数据库所需空间。强烈建议使用 -F 参数以避免单表过大导致备份过程中断。

2. 查看在 ${data-path} 目录下的 metadata 文件，这是 Dumpling 自动生成的元信息文件，请记录其中的 binlog 位置信息，这将在第 3 步增量同步的时候使用。

```
SHOW MASTER STATUS:
Log: mysql-bin.000004
Pos: 109227
GTID:
```

6.3.3.4 第 2 步：导入全量数据到 TiDB

1. 编写配置文件 tidb-lightning.toml：

```
[lightning]
# 日志
level = "info"
file = "tidb-lightning.log"

[tikv-importer]
# "local"：默认使用该模式，适用于 TB 级以上大数据量，但导入期间下游 TiDB 无法对外提供服务。
# "tidb"：TB 级以下数据量也可以采用 `tidb` 后端模式，下游 TiDB 可正常提供服务。
    ↪ 关于后端模式更多信息请参阅：https://docs.pingcap.com/tidb/stable/tidb-lightning-
    ↪ backends
backend = "local"
# 设置排序的键值对的临时存放地址，目标路径必须是一个空目录，目录空间须大于待导入数据集的大小
    ↪ 。建议设为与 `data-source-dir` 不同的磁盘目录并使用闪存介质，独占 IO
    ↪ 会获得更好的导入性能
sorted-kv-dir = "${sorted-kv-dir}"

[mydumper]
# 源数据目录，即第 1 步中 Dumpling 保存数据的路径。
```

```
data-source-dir = "${data-path}" # 本地或 S3 路径，例如：'s3://my-bucket/sql-backup'

[tidb]
# 目标集群的信息
host = ${host}                # 例如：172.16.32.1
port = ${port}                # 例如：4000
user = "${user_name}"         # 例如："root"
password = "${password}"      # 例如："rootroot"
status-port = ${status-port}  # 导入过程 Lightning 需要在从 TiDB 的“状态端口”
    ↪ 获取表结构信息，例如：10080
pd-addr = "${ip}:${port}"     # 集群 PD 的地址，Lightning 通过 PD 获取部分信息，例如
    ↪ 172.16.31.3:2379。当 backend = "local" 时 status-port 和 pd-addr 必须正确填写，
    ↪ 否则导入将出现异常。
```

关于更多 TiDB Lightning 的配置，请参考TiDB Lightning 配置参数。

2. 运行 tidb-lightning。如果直接在命令行中启动程序，可能会因为 SIGHUP 信号而退出，建议配合 nohup 或 screen 等工具，如：

   若从 Amazon S3 导入，则需将有权限访问该 S3 后端存储的账号的 SecretKey 和 AccessKey 作为环境变量传入 Lightning 节点。同时还支持从 ~/.aws/credentials 读取凭证文件。

```
export AWS_ACCESS_KEY_ID=${access_key}
export AWS_SECRET_ACCESS_KEY=${secret_key}
nohup tiup tidb-lightning -config tidb-lightning.toml > nohup.out 2>&1 &
```

3. 导入开始后，可以采用以下任意方式查看进度：

   - 通过 grep 日志关键字 progress 查看进度，默认 5 分钟更新一次。
   - 通过监控面板查看进度，请参考TiDB Lightning 监控。
   - 通过 Web 页面查看进度，请参考Web 界面。

4. 导入完毕后，TiDB Lightning 会自动退出。查看 tidb-lightning.log 日志末尾是否有 the whole procedure completed 信息，如果有，表示导入成功。如果没有，则表示导入遇到了问题，可根据日志中的 error 提示解决遇到的问题。

> **注意：**
>
> 无论导入成功与否，最后一行都会显示 tidb lightning exit。它只是表示 TiDB Lightning 正常退出，不代表任务完成。

如果导入过程中遇到问题，请参见TiDB Lightning 常见问题。

6.3.3.5 第 3 步：使用 DM 持续复制增量数据到 TiDB

6.3.3.5.1 添加数据源

1. 新建 source1.yaml 文件，写入以下内容：

```
# 唯一命名，不可重复。
source-id: "mysql-01"

# DM-worker 是否使用全局事务标识符（GTID）拉取 binlog。使用前提是上游 MySQL 已开启 GTID 模式
    ↪ 。若上游存在主从自动切换，则必须使用 GTID 模式。
enable-gtid: true

from:
  host: "${host}"           # 例如：172.16.10.81
  user: "root"
  password: "${password}"   # 支持但不推荐使用明文密码，建议使用 dmctl encrypt
      ↪ 对明文密码进行加密后使用
  port: 3306
```

2. 在终端中执行下面的命令，使用 tiup dmctl 将数据源配置加载到 DM 集群中:

```
tiup dmctl --master-addr ${advertise-addr} operate-source create source1.yaml
```

该命令中的参数描述如下：

参数	描述
--master-addr	dmctl 要连接的集群的任意 DM-master 节点的 {advertise-addr}，例如：172.16.10.71:8261
operate-source create	向 DM 集群加载数据源

6.3.3.5.2 添加同步任务

1. 编辑 task.yaml，配置增量同步模式，以及每个数据源的同步起点：

```
name: task-test                     # 任务名称，需要全局唯一。
task-mode: incremental              # 任务模式，设为 "incremental" 即只进行增量数据迁移。

# 配置下游 TiDB 数据库实例访问信息
target-database:                    # 下游数据库实例配置。
  host: "${host}"                   # 例如：127.0.0.1
  port: 4000
  user: "root"
  password: "${password}"           # 推荐使用经过 dmctl 加密的密文。

#  使用黑白名单配置需要同步的表
block-allow-list:                   # 数据源数据库实例匹配的表的 block-allow-list
    ↪ 过滤规则集，如果 DM 版本早于 v2.0.0-beta.2 则使用 black-white-list。
  bw-rule-1:                        # 黑白名单配置项 ID。
```

```
    do-dbs: ["${db-name}"]           # 迁移哪些库。

# 配置数据源
mysql-instances:
  - source-id: "mysql-01"            # 数据源 ID，即 source1.yaml 中的 source-id
    block-allow-list: "bw-rule-1"    # 引入上面黑白名单配置。
    # syncer-config-name: "global"    # 引用下面的 syncers 增量数据配置。
    meta:                            # `task-mode` 为 `incremental` 且下游数据库的 `
        ↪ checkpoint` 不存在时 binlog 迁移开始的位置；如果 checkpoint 存在，则以 `
        ↪ checkpoint` 为准。如果 `meta` 项和下游数据库的 `checkpoint` 都不存在，
        ↪ 则从上游当前最新的 binlog 位置开始迁移。
      # binlog-name: "mysql-bin.000004"  # 第 1 步中记录的日志位置，当上游存在主从切换时，
          ↪ 必须使用 gtid。
      # binlog-pos: 109227
      binlog-gtid: "09bec856-ba95-11ea-850a-58f2b4af5188:1-9"

# 【可选配置】 如果增量数据迁移需要重复迁移已经在全量数据迁移中完成迁移的数据，则需要开启
    ↪ safe mode 避免增量数据迁移报错。
#  该场景多见于以下情况：全量迁移的数据不属于数据源的一个一致性快照，
    ↪ 随后从一个早于全量迁移数据之前的位置开始同步增量数据。
# syncers:            # sync 处理单元的运行配置参数。
#  global:           # 配置名称。
#    safe-mode: true # 设置为 true，会将来自数据源的 INSERT 改写为 REPLACE，将 UPDATE 改写为
    ↪  DELETE 与 REPLACE，
    ↪ 从而保证在表结构中存在主键或唯一索引的条件下迁移数据时可以重复导入 DML。
    ↪ 在启动或恢复增量复制任务的前 1 分钟内 TiDB DM 会自动启动 safe mode。
```

以上内容为执行迁移的最小任务配置。关于任务的更多配置项，可以参考DM 任务完整配置文件介绍。

在你启动数据迁移任务之前，建议使用check-task命令检查配置是否符合 DM 的配置要求，以降低后期报错的概率。

```
tiup dmctl --master-addr ${advertise-addr} check-task task.yaml
```

2. 使用 tiup dmctl 执行以下命令启动数据迁移任务：

```
tiup dmctl --master-addr ${advertise-addr} start-task task.yaml
```

该命令中的参数描述如下：

参数	描述
--master-addr	dmctl 要连接的集群的任意 DM-master 节点的 {advertise-addr}，例如：172.16.10.71:8261
start-task	命令用于创建数据迁移任务

如果任务启动失败，可根据返回结果的提示进行配置变更，再执行上述命令重新启动任务。遇到问题请参考故障及处理方法以及常见问题。

6.3.3.5.3 查看任务状态

如需了解 DM 集群中是否存在正在运行的迁移任务及任务状态等信息，可使用 `tiup dmctl` 执行 `query-status` 命令进行查询：

```
tiup dmctl --master-addr ${advertise-addr} query-status ${task-name}
```

关于查询结果的详细解读，请参考查询状态。

6.3.3.5.4 监控任务与查看日志

要查看迁移任务的历史状态以及更多的内部运行指标，可参考以下步骤。

如果使用 TiUP 部署 DM 集群时，正确部署了 Prometheus、Alertmanager 与 Grafana，则使用部署时填写的 IP 及端口进入 Grafana，选择 DM 的 dashboard 查看 DM 相关监控项。

DM 在运行过程中，DM-worker、DM-master 及 dmctl 都会通过日志输出相关信息。各组件的日志目录如下：

- DM-master 日志目录：通过 DM-master 进程参数 `--log-file` 设置。如果使用 TiUP 部署 DM，则日志目录默认位于 `/dm-deploy/dm-master-8261/log/`。
- DM-worker 日志目录：通过 DM-worker 进程参数 `--log-file` 设置。如果使用 TiUP 部署 DM，则日志目录默认位于 `/dm-deploy/dm-worker-8262/log/`。

6.3.3.6 探索更多

- 暂停数据迁移任务
- 恢复数据迁移任务
- 停止数据迁移任务
- 导出和导入集群的数据源和任务配置
- 处理出错的 DDL 语句

6.3.4 从小数据量分库分表 MySQL 合并迁移数据到 TiDB

如果你想把上游多个 MySQL 数据库实例合并迁移到下游的同一个 TiDB 数据库中，且数据量较小，你可以使用 DM 工具进行分库分表的合并迁移。本文所称“小数据量”通常指 TiB 级别以下。本文举例介绍了合并迁移的操作步骤、注意事项、故障排查等。本文档适用于:

- TiB 级以内的分库分表数据合并迁移
- 基于 MySQL binlog 的增量、持续分库分表合并迁移

若要迁移分表总和 1 TiB 以上的数据，则 DM 工具耗时较长，可参考从大数据量分库分表 MySQL 合并迁移数据到 TiDB。

在本文档的示例中，数据源 MySQL 实例 1 和实例 2 均使用以下表结构，计划将 store_01 和 store_02 中 sale 开头的表合并导入下游 store.sale 表。

Schema	Tables
store_01	sale_01, sale_02
store_02	sale_01, sale_02

迁移目标库的结构如下：

Schema	Tables
store	sale

6.3.4.1 前提条件

- 使用 TiUP 安装 DM 集群
- DM 所需上下游数据库权限

6.3.4.2 分表数据冲突检查

迁移中如果涉及合库合表，来自多张分表的数据可能引发主键或唯一索引的数据冲突。因此在迁移之前，需要检查各分表数据的业务特点。详情请参考跨分表数据在主键或唯一索引冲突处理。

在本示例中：`sale_01` 和 `sale_02` 具有相同的表结构如下：

```
CREATE TABLE `sale_01` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `sid` bigint(20) NOT NULL,
  `pid` bigint(20) NOT NULL,
  `comment` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `sid` (`sid`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

其中 `id` 列为主键，`sid` 列为分片键，具有全局唯一性。`id` 列具有自增属性，多个分表范围重复会引发数据冲突。`sid` 可以保证全局满足唯一索引，因此可以按照参考去掉自增主键的主键属性中介绍的操作绕过 `id` 列。在下游创建 `sale` 表时移除 `id` 列的唯一键属性：

```
CREATE TABLE `sale` (
  `id` bigint(20) NOT NULL,
  `sid` bigint(20) NOT NULL,
  `pid` bigint(20) NOT NULL,
  `comment` varchar(255) DEFAULT NULL,
  INDEX (`id`),
  UNIQUE KEY `sid` (`sid`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

6.3.4.3 第 1 步：创建数据源

新建 source1.yaml 文件，写入以下内容：

```
### 唯一命名，不可重复。
source-id: "mysql-01"

### DM-worker 是否使用全局事务标识符 (GTID) 拉取 binlog。使用前提是上游 MySQL 已开启 GTID 模式。
    ↪ 若上游存在主从自动切换，则必须使用 GTID 模式。
enable-gtid: true

from:
  host: "${host}" # 例如：172.16.10.81
  user: "root"
  password: "${password}" # 支持但不推荐使用明文密码，建议使用 dmctl encrypt
      ↪ 对明文密码进行加密后使用
  port: 3306
```

在终端中执行下面的命令，使用 tiup dmctl 将数据源配置加载到 DM 集群中:

```
tiup dmctl --master-addr ${advertise-addr} operate-source create source1.yaml
```

该命令中的参数描述如下：

参数	描述
--master-addr	dmctl 要连接的集群的任意 DM-master 节点的 {advertise-addr}，例如：172.16.10.71:8261
operate-source create	向 DM 集群加载数据源

重复以上操作直至所有数据源均添加完成。

6.3.4.4 第 2 步：创建迁移任务

新建task1.yaml文件，写入以下内容：

```
name: "shard_merge"
### 任务模式，可设为
### full：只进行全量数据迁移
### incremental： binlog 实时同步
### all： 全量 + binlog 迁移
task-mode: all
### 分库分表合并任务则需要配置 shard-mode。默认使用悲观协调模式 "pessimistic"，
    ↪ 在深入了解乐观协调模式的原理和使用限制后，也可以设置为乐观协调模式 "optimistic"
### 详细信息可参考：https://docs.pingcap.com/zh/tidb/v7.0/feature-shard-merge/
shard-mode: "pessimistic"
meta-schema: "dm_meta"                          # 将在下游数据库创建 schema 用于存放元数据
ignore-checking-items: ["auto_increment_ID"]    # 本示例中上游存在自增主键，
    ↪ 因此需要忽略掉该检查项
```

```
target-database:
  host: "${host}"                                  # 例如：192.168.0.1
  port: 4000
  user: "root"
  password: "${password}"                          # 支持但不推荐使用明文密码，建议使用 dmctl
      ↪ encrypt 对明文密码进行加密后使用

mysql-instances:
  -
    source-id: "mysql-01"                                      # 数据源 ID，即 source1.yaml 中的
        ↪ source-id
    route-rules: ["sale-route-rule"]                           # 应用于该数据源的 table route 规则
    filter-rules: ["store-filter-rule", "sale-filter-rule"] # 应用于该数据源的 binlog event
        ↪ filter 规则
    block-allow-list:  "log-bak-ignored"                       # 应用于该数据源的 Block & Allow
        ↪ Lists 规则
  -
    source-id: "mysql-02"
    route-rules: ["sale-route-rule"]
    filter-rules: ["store-filter-rule", "sale-filter-rule"]
    block-allow-list:  "log-bak-ignored"

### 分表合并配置
routes:
  sale-route-rule:
    schema-pattern: "store_*"                                  # 合并 store_01 和 store_02 库到下游
        ↪ store 库
    table-pattern: "sale_*"                                    # 合并上述库中的 sale_01 和 sale_02
        ↪ 表到下游 sale 表
    target-schema: "store"
    target-table:  "sale"
    # 可选配置：提取各分库分表的源信息，并写入下游用户自建的列，用于标识合表中各行数据的来源。
        ↪ 如果配置该项，需要提前在下游手动创建合表，具体可参考下面 Table routing 的用法
    # extract-table:                                           # 提取分表去除 sale_ 的后缀信息，
        ↪ 并写入下游合表 c_table 列，例如，sale_01 分表的数据会提取 01 写入下游 c_table 列
    #   table-regexp: "sale_(.*)"
    #   target-column: "c_table"
    # extract-schema:                                          # 提取分库去除 store_ 的后缀信息，
        ↪ 并写入下游合表 c_schema 列，例如，store_02 分库的数据会提取 02 写入下游 c_schema 列
    #   schema-regexp: "store_(.*)"
    #   target-column: "c_schema"
    # extract-source:                                          # 提取数据库源实例信息写入 c_source
        ↪ 列，例如，mysql-01 数据源实例的数据会提取 mysql-01 写入下游 c_source 列
    #   source-regexp: "(.*)"
```

```
    #   target-column: "c_source"

### 过滤部分 DDL 事件
filters:
  sale-filter-rule:
    schema-pattern: "store_*"
    table-pattern: "sale_*"
    events: ["truncate table", "drop table", "delete"]
    action: Ignore
  store-filter-rule:
    schema-pattern: "store_*"
    events: ["drop database"]
    action: Ignore

### 黑白名单
block-allow-list:
  log-bak-ignored:
    do-dbs: ["store_*"]
```

以上内容为执行迁移的最小任务配置。关于任务的更多配置项，可以参考DM 任务完整配置文件介绍。

若想了解配置文件中 routes、filters 等更多用法，请参考：

- Table routing
- Block & Allow Table Lists
- 如何过滤 binlog 事件
- 如何通过 SQL 表达式过滤 DML

6.3.4.5 第 3 步：启动任务

在你启动数据迁移任务之前，建议使用 check-task 命令检查配置是否符合 DM 的配置要求，以降低后期报错的概率。

```
tiup dmctl --master-addr ${advertise-addr} check-task task.yaml
```

使用 tiup dmctl 执行以下命令启动数据迁移任务。

```
tiup dmctl --master-addr ${advertise-addr} start-task task.yaml
```

该命令中的参数描述如下：

参数	描述
--master-addr	dmctl 要连接的集群的任意 DM-master 节点的 {advertise-addr}，例如：172.16.10.71:8261
start-task	命令用于创建数据迁移任务

如果任务启动失败，可根据返回结果的提示进行配置变更后执行 start-task task.yaml 命令重新启动任务。遇到

问题请参考故障及处理方法以及常见问题。

6.3.4.6 第 4 步：查看任务状态

如需了解 DM 集群中是否存在正在运行的迁移任务及任务状态等信息，可使用 `tiup dmctl` 执行 `query-status` 命令进行查询：

```
tiup dmctl --master-addr ${advertise-addr} query-status ${task-name}
```

关于查询结果的详细解读，请参考查询状态。

6.3.4.7 第 5 步：监控任务与查看日志 (可选)

你可以通过 Grafana 或者日志查看迁移任务的历史状态以及各种内部运行指标。

- 通过 Grafana 查看

  如果使用 TiUP 部署 DM 集群时，正确部署了 Prometheus、Alertmanager 与 Grafana，则使用部署时填写的 IP 及端口进入 Grafana，选择 DM 的 dashboard 查看 DM 相关监控项。

- 通过日志查看

  DM 在运行过程中，DM-worker、DM-master 及 dmctl 都会通过日志输出相关信息，其中包含迁移任务的相关信息。各组件的日志目录如下：

  - DM-master 日志目录：通过 DM-master 进程参数 `--log-file` 设置。如果使用 TiUP 部署 DM，则日志目录默认位于 `/dm-deploy/dm-master-8261/log/`。
  - DM-worker 日志目录：通过 DM-worker 进程参数 `--log-file` 设置。如果使用 TiUP 部署 DM，则日志目录默认位于 `/dm-deploy/dm-worker-8262/log/`。

6.3.4.8 探索更多

- 分库分表合并中的悲观/乐观模式
- 分表合并数据迁移最佳实践
- 故障及处理方法
- 性能问题及处理方法
- 常见问题

6.3.5 从大数据量分库分表 MySQL 合并迁移数据到 TiDB

如果分表数据总规模特别大（例如大于 1 TiB），并且允许 TiDB 集群在迁移期间无其他业务写入，那么你可以使用 TiDB Lightning 对分表数据进行快速合并导入，然后根据业务需要选择是否使用 TiDB DM 进行增量数据的分表同步。本文所称“大数据量”通常指 TiB 级别以上。本文档举例介绍了导入数据的操作步骤。

如果分库分表合并迁移在 1 TiB 以内，请参考从小数据量分库分表 MySQL 合并迁移数据到 TiDB，支持全量和增量且更为简单。

在本文的示例中，假设有两个数据库 my_db1 和 my_db2，使用 Dumpling 分别从 my_db1 中导出 table1 和 table2 两个表，从 my_db2 中导出 table3 和 table4 两个表，然后再用 TiDB Lightning 把导出的 4 个表合并导入到下游 TiDB 中的同一个库 my_db 的同一个表格 table5 中。

本文将以三个步骤演示导入流程：

1. 使用 Dumpling 导出全量数据备份。在本文档示例中，分别从两个源数据库中各导出两个表：
    - 从实例 1 MySQL 的 my_db1 导出 table1、 table2
    - 从实例 2 MySQL 的 my_db2 导出 table3、 table4
2. 启动 TiDB Lightning 执行导入 TiDB 中的 mydb.table5
3. 使用 DM 进行增量数据迁移（可选）

6.3.5.1 前提条件

- 使用 TiUP 安装 DM 集群
- 使用 TiUP 安装 Dumpling 和 Lightning
- Dumpling 所需上游数据库权限
- TiDB Lightning 所需下游数据库权限
- TiDB Lightning 下游数据库所需空间
- DM 所需上下游数据库权限

6.3.5.1.1 分表数据冲突检查

迁移中如果涉及合库合表，来自多张分表的数据可能引发主键或唯一索引的数据冲突。因此在迁移之前，需要检查各分表数据的业务特点。详情请参考跨分表数据在主键或唯一索引冲突处理，这里做简要描述：

假设 table1~4 具有相同的表结构如下：

```
CREATE TABLE `table1` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `sid` bigint(20) NOT NULL,
  `pid` bigint(20) NOT NULL,
  `comment` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `sid` (`sid`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

其中 `id` 列为主键，具有自增属性，多个分表范围重复会引发数据冲突。`sid` 列为分片键，可以保证全局满足唯一索引。因此可以移除下游 `table5` 表 `id` 列的唯一键属性：

```
CREATE TABLE `table5` (
  `id` bigint(20) NOT NULL,
  `sid` bigint(20) NOT NULL,
  `pid` bigint(20) NOT NULL,
  `comment` varchar(255) DEFAULT NULL,
  INDEX (`id`),
  UNIQUE KEY `sid` (`sid`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

6.3.5.2 第 1 步：用 Dumpling 导出全量数据备份

如果需要导出的多个分表属于同一个上游 MySQL 实例，建议直接使用 Dumpling 的 `-f` 参数一次导出多个分表的结果。如果多个分表分布在不同的 MySQL 实例，可以使用 Dumpling 分两次导出，并将两次导出的结果放置在相同的父目录下即可。下面的例子中同时用到了上述两种方式，然后将导出的数据存放在同一父目录下。

首先使用 Dumpling 从 my_db1 中导出表 table1 和 table2，如下：

```
tiup dumpling -h ${ip} -P 3306 -u root -t 16 -r 200000 -F 256MB -B my_db1 -f 'my_db1.table[12]' -
    ↪ o ${data-path}/my_db1
```

以上命令行中用到的参数描述如下。要了解更多 Dumpling 参数，请参考Dumpling 使用文档。

参数	描述
`-u` 或 `--user`	MySQL 数据库的用户
`-p` 或 `--password`	MySQL 数据库的用户密码
`-P` 或 `--port`	MySQL 数据库的端口
`-h` 或 `--host`	MySQL 数据库的 IP 地址
`-t` 或 `--thread`	导出的线程数。增加线程数会增加 Dumpling 并发度提高导出速度，但也会加大数据库内存消耗，因此不宜
`-o` 或 `--output`	存储导出文件的目录，支持本地文件路径或外部存储 URI 格式
`-r` 或 `--row`	用于指定单个文件的最大行数，指定该参数后 Dumpling 会开启表内并发加速导出，同时减少内存使用
`-F`	指定单个文件的最大大小，单位为 MiB。强烈建议使用`-F`参数以避免单表过大导致备份过程中断
`-B` 或 `--database`	导出指定数据库
`-f` 或 `--filter`	导出能匹配模式的表，语法可参考table-filter

然后使用同样的方式从 my_db2 中导出表 table3 和 table4。注意路径在相同 `${data-path}` 下的不同子目录 my_db2。

```
tiup dumpling -h ${ip} -P 3306 -u root -t 16 -r 200000 -F 256MB -B my_db2 -f 'my_db2.table[34]' -
    ↪ o ${data-path}/my_db2
```

这样所需的全量备份数据就全部导出到了 `${data-path}` 目录中。将所有源数据表格存储在一个目录中，是为了后续方便用 TiDB Lightning 导入。

第 3 步增量同步的时候所需的起始位点信息，在 `${data-path}` 目录下，`my_db1` 和 `my_db2` 的 `metadata` 文件中，这是 Dumpling 自动生成的元信息文件，请记录其中的 binlog 位置信息。

6.3.5.3 第 2 步：启动 TiDB Lightning 进行导入

在启动 TiDB Lightning 进行迁移之前，建议先了解如何处理检查点，然后根据需要选择合适的方式进行迁移。

6.3.5.3.1 断点续传

大量数据导入一般耗时数小时甚至数天，长时间运行的进程会有一定机率发生非正常中断。如果每次重启都从头开始，之前已成功导入的数据就会前功尽弃。为此，TiDB Lightning 提供了断点续传的功能，即使 TiDB Lightning 崩溃，在重启时仍然从断点开始继续工作。

若 TiDB Lightning 因不可恢复的错误而退出，例如数据出错，在重启时不会使用断点，而是直接报错离开。为保证已导入的数据安全，必须先解决掉这些错误才能继续。你可以使用 tidb-lightning-ctl 命令控制导入出错后的行为。该命令的选项有：

- –checkpoint-error-destroy：出现错误后，让失败的表从头开始整个导入过程。
- –checkpoint-error-ignore：如果导入表曾经出错，该命令会清除出错状态，如同错误没有发生过一样。
- –checkpoint-remove：无论是否有出错，把表的断点清除。

关于断点续传的更多信息，请参考TiDB Lightning 断点续传。

6.3.5.3.2 在下游创建 schema

在下游创建 mydb.table5。

```
CREATE TABLE `table5` (
  `id` bigint(20) NOT NULL,
  `sid` bigint(20) NOT NULL,
  `pid` bigint(20) NOT NULL,
  `comment` varchar(255) DEFAULT NULL,
  INDEX (`id`),
  UNIQUE KEY `sid` (`sid`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

6.3.5.3.3 执行导入操作

启动 tidb-lightning 的步骤如下：

1. 编写配置文件 tidb-lightning.toml。

```

[lightning]
# 日志
level = "info"
file = "tidb-lightning.log"

[mydumper]
data-source-dir = ${data-path}

[tikv-importer]
# "local"：默认使用该模式，适用于 TB 级以上大数据量，但导入期间下游 TiDB 无法对外提供服务。
# "tidb"：TB 级以下数据量也可以采用`tidb`后端模式，下游 TiDB 可正常提供服务。
    ↪ 关于后端模式更多信息请参阅：https://docs.pingcap.com/tidb/stable/tidb-lightning-
    ↪ backends
backend = "local"
# 设置排序的键值对的临时存放地址，目标路径必须是一个空目录，目录空间须大于待导入数据集的大小
    ↪ 。建议设为与 `data-source-dir` 不同的磁盘目录，独占 IO 会获得更好的导入性能
```

```
sorted-kv-dir = "${sorted-kv-dir}"

# 设置分库分表合并规则，将 my_db1 中的 table1、table2 两个表,以及 my_db2 中的 table3、table4
    ↪ 两个表，共计 2 个数据库中的 4 个表都导入到目的数据库 my_db 中的 table5 表中。
[[mydumper.files]]
pattern = '(^|/)my_db1\.table[1-2]\..*\.sql$'
schema = "my_db"
table = "table5"
type = "sql"

[[mydumper.files]]
pattern = '(^|/)my_db2\.table[3-4]\..*\.sql$'
schema = "my_db"
table = "table5"
type = "sql"

# 目标集群的信息，示例仅供参考。请把 IP 地址等信息替换成真实的信息。
[tidb]
# 目标集群的信息
host = ${host}              # 例如：172.16.32.1
port = ${port}              # 例如：4000
user = "${user_name}"       # 例如："root"
password = "${password}"    # 例如："rootroot"
status-port = ${status-port} # 导入过程 Lightning 需要在从 TiDB 的“状态端口”获取表结构信息
    ↪ ，例如：10080
# PD 集群的地址，Lightning 通过 PD 获取部分信息。
pd-addr = "${ip}:${port}"   # 例如 172.16.31.3:2379。当 backend = "local" 时 status-port 和
    ↪ pd-addr 必须正确填写，否则导入将出现异常。
```

2. 运行 tidb-lightning。如果直接在命令行中启动程序，可能会因为 SIGHUP 信号而退出，建议配合 nohup 或 screen 等工具，如：

   若从 Amazon S3 导入，则需将有权限访问该 S3 后端存储的账号的 SecretKey 和 AccessKey 作为环境变量传入 TiDB Lightning 节点。同时还支持从 ~/.aws/credentials 读取凭证文件。

```
export AWS_ACCESS_KEY_ID=${access_key}
export AWS_SECRET_ACCESS_KEY=${secret_key}
nohup tiup tidb-lightning -config tidb-lightning.toml > nohup.out 2>&1 &
```

3. 导入开始后，可以采用以下任意方式查看进度：

   - 通过 grep 日志关键字 progress 查看进度，默认 5 分钟更新一次。
   - 通过监控面板查看进度，请参考TiDB Lightning 监控。
   - 通过 Web 页面查看进度，请参考Web 界面。

4. 导入完毕后，TiDB Lightning 会自动退出。查看 tidb-lightning.log 日志末尾是否有 the whole procedure completed 信息，如果有，表示导入成功。如果没有，则表示导入遇到了问题，可根据日志中的 error 提示解决遇到的问题。

> **注意：**
>
> 无论导入成功与否，最后一行都会显示 `tidb lightning exit`。它只是表示 TiDB Lightning 正常退出，不代表任务完成。

如果导入过程中遇到问题，请参见TiDB Lightning 常见问题。

6.3.5.4 第 3 步：使用 DM 持续复制增量数据到 TiDB (可选)

基于 binlog 从指定位置同步数据库到 TiDB，可以使用 DM 来执行增量复制

6.3.5.4.1 添加数据源

新建 `source1.yaml` 文件，写入以下内容：

```
### 唯一命名，不可重复。
source-id: "mysql-01"

### DM-worker 是否使用全局事务标识符 (GTID) 拉取 binlog。使用前提是上游 MySQL 已开启 GTID 模式。
    ↪ 若上游存在主从自动切换，则必须使用 GTID 模式。
enable-gtid: true

from:
  host: "${host}"           # 例如：172.16.10.81
  user: "root"
  password: "${password}"   # 支持但不推荐使用明文密码，建议使用 dmctl encrypt
      ↪ 对明文密码进行加密后使用
  port: 3306
```

在终端中执行下面的命令，使用 `tiup dmctl` 将数据源配置加载到 DM 集群中:

```
tiup dmctl --master-addr ${advertise-addr} operate-source create source1.yaml
```

该命令中的参数描述如下：

参数	描述
`--master-addr`	dmctl 要连接的集群的任意 DM-master 节点的 `{advertise-addr}`
`operate-source create`	向 DM 集群加载数据源

重复以上步骤直至所有 MySQL 实例被加入 DM。

6.3.5.4.2 添加同步任务

编辑 task.yaml，配置增量同步模式，以及每个数据源的同步起点：

```
name: task-test                 # 任务名称，需要全局唯一。
task-mode: incremental          # 任务模式，设为 "incremental" 即只进行增量数据迁移。
### 分库分表合并任务则需要配置 shard-mode。默认使用悲观协调模式 "pessimistic"，
    ↪ 在深入了解乐观协调模式的原理和使用限制后，也可以设置为乐观协调模式 "optimistic"
### 详细信息可参考：https://docs.pingcap.com/zh/tidb/v7.0/feature-shard-merge/
shard-mode: "pessimistic"

#### 配置下游 TiDB 数据库实例访问信息
target-database:                # 下游数据库实例配置。
  host: "${host}"               # 例如：127.0.0.1
  port: 4000
  user: "root"
  password: "${password}"       # 推荐使用经过 dmctl 加密的密文。

####  使用黑白名单配置需要同步的表
block-allow-list:               # 数据源数据库实例匹配的表的 block-allow-list 过滤规则集，如果 DM
    ↪ 版本早于 v2.0.0-beta.2 则使用 black-white-list。
  bw-rule-1:                    # 黑白名单配置项 ID。
    do-dbs: ["my_db1"]          # 迁移哪些库。这里将实例 1 的 my_db1 和实例 2 的 my_db2
        ↪ 分别配置为两条 rule，以示例如何避免实例 1 的 my_db2 被同步。
  bw-rule-2:
    do-dbs: ["my_db2"]

routes:                                 # 上游和下游表之间的路由 table routing 规则集
  route-rule-1:                         # 配置名称。将 my_db1 中的 table1 和 table2 合并导入下游
      ↪ my_db.table5
    schema-pattern: "my_db1"            # 库名匹配规则，支持通配符 "*" 和 "?"
    table-pattern: "table[1-2]"         # 表名匹配规则，支持通配符 "*" 和 "?"
    target-schema: "my_db"              # 目标库名称
    target-table: "table5"              # 目标表名称
  route-rule-2:                         # 配置名称。将 my_db2 中的 table3 和 table4 合并导入下游
      ↪ my_db.table5
    schema-pattern: "my_db2"
    table-pattern: "table[3-4]"
    target-schema: "my_db"
    target-table: "table5"

#### 配置数据源，以两个数据源为例
mysql-instances:
  - source-id: "mysql-01"               # 数据源 ID，即 source1.yaml 中的 source-id
    block-allow-list: "bw-rule-1"       # 引入上面黑白名单配置。同步实例1的 my_db1
    route-rules: ["route-rule-1"]       # 引入上面表合并配置。
###       syncer-config-name: "global"  # 引用后面的 syncers 增量数据配置。
```

```
    meta:                               # `task-mode` 为 `incremental` 且下游数据库的 `checkpoint`
        ↪ 不存在时 binlog 迁移开始的位置；如果 checkpoint 存在，则以 `checkpoint` 为准。如果 `
        ↪ meta` 项和下游数据库的 `checkpoint` 都不存在，则从上游当前最新的 binlog 位置开始迁移
        ↪ 。
      binlog-name: "${binlog-name}"   # 第 1 步中 ${data-path}/my_db1/metadata 记录的日志位置，
          ↪ 当上游存在主从切换时，必须使用 gtid。
      binlog-pos: ${binlog-position}
      # binlog-gtid:                  " 例如：09bec856-ba95-11ea-850a-58f2b4af5188:1-9"
  - source-id: "mysql-02"             # 数据源 ID，即 source1.yaml 中的 source-id
    block-allow-list: "bw-rule-2"     # 引入上面黑白名单配置。实例2的 my_db2
    route-rules: ["route-rule-2"]     # 引入上面表合并配置。

###       syncer-config-name: "global"  # 引用后面的 syncers 增量数据配置。
    meta:                               # task-mode 为 incremental 且下游数据库的 checkpoint
        ↪ 不存在时 binlog 迁移开始的位置；如果 checkpoint 存在，则以 checkpoint 为准。
      # binlog-name: "${binlog-name}"   # 第 1 步中 ${data-path}/my_db2/metadata 记录的日志位置，
          ↪ 当上游存在主从切换时，必须使用 gtid。
      # binlog-pos: ${binlog-position}
      binlog-gtid: "09bec856-ba95-11ea-850a-58f2b4af5188:1-9"

#### 【可选配置】 如果增量数据迁移需要重复迁移已经在全量数据迁移中完成迁移的数据，则需要开启 safe
    ↪  mode 避免增量数据迁移报错。
####  该场景多见于以下情况：全量迁移的数据不属于数据源的一个一致性快照，
    ↪ 随后从一个早于全量迁移数据之前的位置开始同步增量数据。
### syncers:            # sync 处理单元的运行配置参数。
###  global:           # 配置名称。
###    safe-mode: true # 设置为 true，会将来自数据源的 INSERT 改写为 REPLACE，将 UPDATE 改写为
    ↪ DELETE 与 REPLACE，从而保证在表结构中存在主键或唯一索引的条件下迁移数据时可以重复导入 DML
    ↪ 。在启动或恢复增量复制任务的前 1 分钟内 TiDB DM 会自动启动 safe mode。
```

关于任务的更多配置项，可以参考DM 任务完整配置文件介绍。

在你启动数据迁移任务之前，建议使用 check-task 命令检查配置是否符合 DM 的配置要求，以降低后期报错的概率。

```
tiup dmctl --master-addr ${advertise-addr} check-task task.yaml
```

使用 tiup dmctl 执行以下命令启动数据迁移任务。

```
tiup dmctl --master-addr ${advertise-addr} start-task task.yaml
```

该命令中的参数描述如下：

参数	描述
--master-addr	dmctl 要连接的集群的任意 DM-master 节点的 {advertise-addr}，例如：172.16.10.71:8261
start-task	命令用于创建数据迁移任务

如果任务启动失败，可根据返回结果的提示进行配置变更后执行 start-task task.yaml 命令重新启动任务。遇到问题请参考故障及处理方法以及常见问题。

6.3.5.4.3 查看任务状态

如需了解 DM 集群中是否存在正在运行的迁移任务及任务状态等信息，可使用 `tiup dmctl` 执行 `query-status` 命令进行查询：

```
tiup dmctl --master-addr ${advertise-addr} query-status ${task-name}
```

关于查询结果的详细解读，请参考查询状态。

6.3.5.4.4 监控任务与查看日志

你可以通过 Grafana 或者日志查看迁移任务的历史状态以及各种内部运行指标。

- 通过 Grafana 查看

  如果使用 TiUP 部署 DM 集群时，正确部署了 Prometheus、Alertmanager 与 Grafana，则使用部署时填写的 IP 及端口进入 Grafana，选择 DM 的 dashboard 查看 DM 相关监控项。

- 通过日志查看

  DM 在运行过程中，DM-worker、DM-master 及 dmctl 都会通过日志输出相关信息，其中包含迁移任务的相关信息。各组件的日志目录如下：

  - DM-master 日志目录：通过 DM-master 进程参数 `--log-file` 设置。如果使用 TiUP 部署 DM，则日志目录默认位于 `/dm-deploy/dm-master-8261/log/`。
  - DM-worker 日志目录：通过 DM-worker 进程参数 `--log-file` 设置。如果使用 TiUP 部署 DM，则日志目录默认位于 `/dm-deploy/dm-worker-8262/log/`。

6.3.5.5 探索更多

- 关于 Dumpling
- 关于 Lightning
- 分库分表合并中的悲观/乐观模式
- 暂停数据迁移任务
- 恢复数据迁移任务
- 停止数据迁移任务
- 导出和导入集群的数据源和任务配置
- 处理出错的 DDL 语句
- 故障及处理方法

6.3.6 从 CSV 文件迁移数据到 TiDB

本文档介绍如何从 CSV 文件迁移数据到 TiDB。

TiDB Lightning 支持读取 CSV 格式的文件，以及其他定界符格式，如 TSV（制表符分隔值）。对于其他“平面文件”类型的数据导入，也可以参考本文档进行。

6.3.6.1 前提条件

- 安装 TiDB Lightning。
- 获取 Lightning 所需下游数据库权限。

6.3.6.2 第 1 步：准备 CSV 文件

将所有要导入的 CSV 文件放在同一目录下，若要 TiDB Lightning 识别所有 CSV 文件，文件名必须满足以下格式：

- 包含整张表数据的 CSV 文件，需命名为 `${db_name}.${table_name}.csv`。
- 如果一张表分布于多个 CSV 文件，这些 CSV 文件命名需加上文件编号的后缀，如 `${db_name}.${table_name}.003.csv`。数字部分不需要连续，但必须递增，并且需要用零填充数字部分，保证后缀为同样长度。

6.3.6.3 第 2 步：创建目标表结构

CSV 文件自身未包含表结构信息。要将 CSV 数据导入 TiDB，就必须为数据提供表结构。可以通过以下任一方法创建表结构：

- 方法一：使用 TiDB Lightning 创建表结构。

  编写包含 DDL 语句的 SQL 文件如下：

  - 文件名格式为 `${db_name}-schema-create.sql`，其内容需包含 `CREATE DATABASE` 语句。
  - 文件名格式为 `${db_name}.${table_name}-schema.sql`，其内容需包含 `CREATE TABLE` 语句。

- 方法二：手动在下游 TiDB 建库和表。

6.3.6.4 第 3 步：编写配置文件

新建文件 `tidb-lightning.toml`，包含以下内容：

```
[lightning]
### 日志
level = "info"
file = "tidb-lightning.log"

[tikv-importer]
### "local"：默认使用该模式，适用于 TB 级以上大数据量，但导入期间下游 TiDB 无法对外提供服务。
### "tidb"：TB 级以下数据量也可以采用 `tidb` 后端模式，下游 TiDB 可正常提供服务。
    ↪ 关于导入模式更多信息请参阅：https://docs.pingcap.com/zh/tidb/stable/tidb-lightning-
    ↪ overview#tidb-lightning-整体架构
backend = "local"
### 设置排序的键值对的临时存放地址，目标路径必须是一个空目录，目录空间须大于待导入数据集的大小，
    ↪ 建议设为与 `data-source-dir` 不同的磁盘目录并使用闪存介质，独占 IO 会获得更好的导入性能
sorted-kv-dir = "/mnt/ssd/sorted-kv-dir"

[mydumper]
```

```
### 源数据目录。
data-source-dir = "${data-path}" # 本地或 S3 路径，例如：'s3://my-bucket/sql-backup'

### 定义 CSV 格式
[mydumper.csv]
### 字段分隔符，必须不为空。如果源文件中包含非字符串或数值类型的字段（如 binary, blob, bit 等），
    ↪ 则不建议源文件使用默认的“,”简单分隔符，推荐“|+|”等非常见字符组合
separator = ','
### 引用定界符，可以为零或多个字符。
delimiter = '"'
### CSV 文件是否包含表头。
### 如果为 true，则 lightning 会使用首行内容解析字段的对应关系。
header = true
### CSV 是否包含 NULL。
### 如果为 true，CSV 文件的任何列都不能解析为 NULL。
not-null = false
### 如果 `not-null` 为 false（即 CSV 可以包含 NULL），
### 为以下值的字段将会被解析为 NULL。
null = '\N'
### 是否将字符串中包含的反斜杠（'\'）字符作为转义字符处理
backslash-escape = true
### 是否移除行尾的最后一个分隔符。
trim-last-separator = false

[tidb]
### 目标集群的信息
host = ${host}               # 例如：172.16.32.1
port = ${port}               # 例如：4000
user = "${user_name}"        # 例如："root"
password = "${password}"     # 例如："rootroot"
status-port = ${status-port}  # 导入过程 Lightning 需要在从 TiDB 的“状态端口”获取表结构信息，
    ↪ 例如：10080
pd-addr = "${ip}:${port}"     # 集群 PD 的地址，Lightning 通过 PD 获取部分信息，例如
    ↪ 172.16.31.3:2379。当 backend = "local" 时 status-port 和 pd-addr 必须正确填写，
    ↪ 否则导入将出现异常。
```

关于配置文件更多信息，可参阅TiDB Lightning 配置参数。

6.3.6.5 第 4 步：导入性能优化（可选）

导入文件的大小统一约为 256 MiB 时，TiDB Lightning 可达到最佳工作状态。如果导入单个 CSV 大文件，TiDB Lightning 在默认配置下只能使用一个线程来处理，这会降低导入速度。

要解决此问题，可先将 CSV 文件分割为多个文件。对于通用格式的 CSV 文件，在没有读取整个文件的情况下，无法快速确定行的开始和结束位置。因此，默认情况下 TiDB Lightning 不会自动分割 CSV 文件。但如果你确定待导入的 CSV 文件符合特定的限制要求，则可以启用 `strict-format` 模式。启用后，TiDB Lightning 会将单个 CSV

大文件分割为单个大小为 256 MiB 的多个文件块进行并行处理。

> **注意：**
>
> 如果 CSV 文件不是严格格式，但 `strict-format` 被误设为 `true`，跨多行的单个完整字段会被分割成两部分，导致解析失败，甚至不报错地导入已损坏的数据。

严格格式的 CSV 文件中，每个字段仅占一行，即必须满足以下条件之一：

- delimiter 为空；
- 每个字段不包含 CR (\r) 或 LF (\n)。

如果你确认满足条件，可按如下配置开启 `strict-format` 模式以加快导入速度。

```
[mydumper]
strict-format = true
```

6.3.6.6 第 5 步：执行导入

运行 `tidb-lightning`。如果直接在命令行中启动程序，可能会因为 `SIGHUP` 信号而退出，建议配合 nohup 或 `screen` 等工具，如：

```
nohup tiup tidb-lightning -config tidb-lightning.toml > nohup.out 2>&1 &
```

导入开始后，可以采用以下任意方式查看进度：

- 通过 `grep` 日志关键字 `progress` 查看进度，默认 5 分钟更新一次。
- 通过监控面板查看进度，请参考TiDB Lightning 监控。
- 通过 Web 页面查看进度，请参考Web 界面。

导入完毕后，TiDB Lightning 会自动退出。查看 `tidb-lightning.log` 日志末尾是否有 `the whole procedure completed` 信息，如果有，表示导入成功。如果没有，则表示导入遇到了问题，可根据日志中的 error 提示解决遇到的问题。

> **注意：**
>
> 无论导入成功与否，最后一行都会显示 `tidb lightning exit`。它只是表示 TiDB Lightning 正常退出，不代表任务完成。

如果导入过程中遇到问题，请参见TiDB Lightning 常见问题。

6.3.6.7 其他格式的文件

若数据源为其他格式，除文件名仍必须以 `.csv` 结尾外，配置文件 `tidb-lightning.toml` 的 `[mydumper.csv]` 格式定义同样需要做相应修改。常见的格式修改如下：

TSV：

```
### 格式示例
### ID    Region    Count
### 1     East      32
### 2     South     NULL
### 3     West      10
### 4     North     39

### 格式配置
[mydumper.csv]
separator = "\t"
delimiter = ''
header = true
not-null = false
null = 'NULL'
backslash-escape = false
trim-last-separator = false
```

TPC-H DBGEN：

```
### 格式示例
### 1|East|32|
### 2|South|0|
### 3|West|10|
### 4|North|39|

### 格式配置
[mydumper.csv]
separator = '|'
delimiter = ''
header = false
not-null = true
backslash-escape = false
trim-last-separator = true
```

6.3.6.8 探索更多

- CSV 支持与限制

6.3.7 从 SQL 文件迁移数据到 TiDB

本文介绍如何使用 TiDB Lightning 从 MySQL SQL 文件迁移数据到 TiDB。关于如何生成 MySQL SQL 文件，请参考 Dumpling 文档中的导出为 SQL 文件。

6.3.7.1 前提条件

- 使用 TiUP 安装 TiDB Lightning
- Lightning 所需下游数据库权限

6.3.7.2 第 1 步：准备 SQL 文件

将所有 SQL 文件放到统一目录下，例如 `/data/my_datasource/` 或 `s3://my-bucket/sql-backup`。Lightning 将递归地寻找该目录下及其子目录内的所有 `.sql` 文件。

6.3.7.3 第 2 步：定义目标表结构

要导入 TiDB，就必须为 SQL 文件提供表结构。

如果使用 Dumpling 工具导出数据，则会自动导出表结构文件。此外，其他方式导出的数据可以通过以下任一方法创建表结构：

- 方法一：使用 TiDB Lightning 创建表结构。

  编写包含 DDL 语句的 SQL 文件：

  - 文件名格式为 `${db_name}-schema-create.sql`，其内容需包含 `CREATE DATABASE` 语句。
  - 文件名格式为 `${db_name}.${table_name}-schema.sql`，其内容需包含 `CREATE TABLE` 语句。

- 方法二：手动在下游 TiDB 建库和表。

6.3.7.4 第 3 步：编写配置文件

新建文件 `tidb-lightning.toml`，包含以下内容：

```
[lightning]
### 日志
level = "info"
file = "tidb-lightning.log"

[tikv-importer]
### "local"：默认使用该模式，适用于 TiB 级以上大数据量，但导入期间下游 TiDB 无法对外提供服务。
backend = "local"
### # "tidb"：TiB 级以下数据量也可以采用 `tidb` 后端模式，下游 TiDB 可正常提供服务。
    ↪ 关于后端模式更多信息请参考 https://docs.pingcap.com/zh/tidb/stable/tidb-lightning-
    ↪ backends 。
### 设置排序的键值对的临时存放地址，目标路径必须是一个空目录，目录空间须大于待导入数据集的大小。
    ↪ 建议设为与 `data-source-dir` 不同的磁盘目录并使用闪存介质，独占 I/O 会获得更好的导入性能
    ↪ 。
```

```
sorted-kv-dir = "${sorted-kv-dir}"

[mydumper]
### 源数据目录
data-source-dir = "${data-path}" # 本地或 S3 路径，例如：'s3://my-bucket/sql-backup'

[tidb]
### 目标集群的信息
host = ${host}                # 例如：172.16.32.1
port = ${port}                # 例如：4000
user = "${user_name}"         # 例如："root"
password = "${password}"      # 例如："rootroot"
status-port = ${status-port}  # 导入过程 Lightning 需要在从 TiDB 的“状态端口”获取表结构信息，
    ↪ 例如：10080
pd-addr = "${ip}:${port}"     # 集群 PD 的地址，Lightning 通过 PD 获取部分信息，例如
    ↪ 172.16.31.3:2379。当 backend = "local" 时 status-port 和 pd-addr 必须正确填写，
    ↪ 否则导入将出现异常。
```

关于配置文件更多信息，可参阅TiDB Lightning Configuration。

6.3.7.5 第 4 步：执行导入

运行 tidb-lightning。如果直接在命令行中启动程序，可能会因为 SIGHUP 信号而退出，建议配合 nohup 或 screen 等工具。

若从 Amazon S3 导入，则需将有权限访问该 S3 后端存储的账号的 SecretKey 和 AccessKey 作为环境变量传入 Lightning 节点。

```
export AWS_ACCESS_KEY_ID=${access_key}
export AWS_SECRET_ACCESS_KEY=${secret_key}
nohup tiup tidb-lightning -config tidb-lightning.toml > nohup.out 2>&1 &
```

同时，TiDB Lightning 还支持从 ~/.aws/credentials 读取凭证文件。

导入开始后，可以采用以下任意方式查看进度：

- 通过 grep 日志关键字 progress 查看进度，默认 5 分钟更新一次。
- 通过监控面板查看进度，请参考TiDB Lightning 监控。
- 通过 Web 页面查看进度，请参考Web 界面。

导入完毕后，TiDB Lightning 会自动退出。查看 tidb-lightning.log 日志末尾是否有 the whole procedure ↪ completed 信息，如果有，表示导入成功。如果没有，则表示导入遇到了问题，可根据日志中的 error 提示解决遇到的问题。

> **注意：**
>
> 无论导入成功与否，最后一行都会显示 `tidb lightning exit`。它只是表示 TiDB Lightning 正常退出，不代表任务完成。

如果导入过程中遇到问题，请参见TiDB Lightning 常见问题。

6.3.8 从 Parquet 文件迁移数据到 TiDB

本文介绍如何从 Apache Hive 中生成 Parquet 文件以及如何使用 TiDB Lightning 从 Parquet 文件迁移数据到 TiDB。

如果你从 Amazon Aurora 中导出 Parquet 文件，请参照从 Amazon Aurora 迁移数据到 TiDB。

6.3.8.1 前提条件

- 使用 TiUP 安装 TiDB Lightning
- 获取 TiDB Lightning 所需下游数据库权限

6.3.8.2 第 1 步：准备 Parquet 文件

本节描述如何从 Hive 中导出能被 TiDB Lightning 读取的 Parquet 文件。

Hive 中每个表都能通过标注 `STORED AS PARQUET LOCATION '/path/in/hdfs'` 的形式将表数据导出到 Parquet 文件中。因此，如果你需要导出一张名叫 `test` 的表，请执行以下步骤：

1. 在 Hive 中执行如下 SQL 语句：

    ```
    CREATE TABLE temp STORED AS PARQUET LOCATION '/path/in/hdfs'
    AS SELECT * FROM test;
    ```

    执行上述语句后，表数据就成功导出到 HDFS 系统里。

2. 使用 `hdfs dfs -get` 命令将 Parquet 文件导出到本地：

    ```
    hdfs dfs -get /path/in/hdfs /path/in/local
    ```

    完成导出后，如果你需要将 HDFS 里导出的 Parquet 文件删除，可以直接将这个临时表 (`temp`) 删掉：

    ```
    DROP TABLE temp;
    ```

3. 从 Hive 导出的 Parquet 文件可能不带有 `.parquet` 的后缀，无法被 TiDB Lightning 正确识别。因此，在进行导入之前，需要对导出的文件进行重命名，添加 `.parquet` 后缀。

4. 将所有 Parquet 文件放到统一目录下，例如 `/data/my_datasource/` 或 `s3://my-bucket/sql-backup`。TiDB Lightning 将递归地寻找该目录及其子目录内的所有 `.parquet` 文件。

6.3.8.3 第 2 步：创建目标表结构

在将 Parquet 文件导入 TiDB 前，你必须为 Parquet 文件提供表结构。你可以通过以下任一方法创建表结构：

- 方法一：使用 TiDB Lightning 创建表结构。

  编写包含 DDL 语句的 SQL 文件：

  - 文件名格式为 `${db_name}-schema-create.sql`，其内容需包含 `CREATE DATABASE` 语句。
  - 文件名格式为 `${db_name}.${table_name}-schema.sql`，其内容需包含 `CREATE TABLE` 语句。

- 方法二：手动在下游 TiDB 建库和表。

6.3.8.4 第 3 步：编写配置文件

新建文件 `tidb-lightning.toml`，包含以下内容：

```
[lightning]
### 日志
level = "info"
file = "tidb-lightning.log"

[tikv-importer]
### "local"：默认使用该模式，适用于 TiB 级以上大数据量，但导入期间下游 TiDB 无法对外提供服务。
backend = "local"
### # "tidb"：TiB 级以下数据量也可以采用 `tidb` 后端模式，下游 TiDB 可正常提供服务。
    ↪ 关于导入模式更多信息请参阅：https://docs.pingcap.com/zh/tidb/stable/tidb-lightning-
    ↪ overview#tidb-lightning-整体架构
### 设置排序的键值对的临时存放地址，目标路径必须是一个空目录，目录空间须大于待导入数据集的大小。
    ↪ 建议设为与 `data-source-dir` 不同的磁盘目录并使用闪存介质，独占 I/O 会获得更好的导入性能
    ↪ 。
sorted-kv-dir = "${sorted-kv-dir}"

[mydumper]
### 源数据目录
data-source-dir = "${data-path}" # 本地或 S3 路径，例如：'s3://my-bucket/sql-backup'

[tidb]
### 目标集群的信息
host = ${host}               # 例如：172.16.32.1
port = ${port}               # 例如：4000
user = "${user_name}"        # 例如："root"
password = "${password}"     # 例如："rootroot"
status-port = ${status-port}  # 导入过程 Lightning 需要在从 TiDB 的“状态端口”获取表结构信息，
    ↪ 例如：10080
pd-addr = "${ip}:${port}"     # 集群 PD 的地址，Lightning 通过 PD 获取部分信息，例如
    ↪ 172.16.31.3:2379。当 backend = "local" 时 status-port 和 pd-addr 必须正确填写，
    ↪ 否则导入将出现异常。
```

关于配置文件更多信息，可参阅TiDB Lightning 配置参数。

6.3.8.5 第 4 步：执行导入

1. 运行 tidb-lightning。

    - 如果从 Amazon S3 导入，需先将有权限访问该 S3 后端存储的账号的 SecretKey 和 AccessKey 作为环境变量传入 Lightning 节点。

    ```
    export AWS_ACCESS_KEY_ID=${access_key}
    export AWS_SECRET_ACCESS_KEY=${secret_key}
    ```

    此外，TiDB Lightning 还支持从 ~/.aws/credentials 读取凭证文件。

    - 如果直接在命令行中启动程序，可能会因为 SIGHUP 信号而退出，建议配合 nohup 或 screen 等工具运行 tidb-lightning：

    ```
    nohup tiup tidb-lightning -config tidb-lightning.toml > nohup.out 2>&1 &
    ```

2. 导入开始后，可以采用以下任意方式查看进度：

    - 通过 grep 日志关键字 progress 查看进度，默认 5 分钟更新一次。
    - 通过监控面板查看进度，请参考TiDB Lightning 监控。
    - 通过 Web 页面查看进度，请参考Web 界面。

    导入完毕后，TiDB Lightning 会自动退出。

3. 检查导入是否成功。

    查看 tidb-lightning.log 日志末尾是否有 the whole procedure completed 信息，如果有，表示导入成功。如果没有，则表示导入遇到了问题，可根据日志中的 error 提示解决遇到的问题。

> 注意：
> 无论导入成功与否，最后一行都会显示 tidb lightning exit。它只是表示 TiDB Lightning 正常退出，不代表任务完成。

如果导入过程中遇到问题，请参见TiDB Lightning 常见问题。

6.3.9 从 TiDB 集群迁移数据至另一 TiDB 集群

本文档介绍如何将数据从一个 TiDB 集群迁移至另一 TiDB。在如下场景中，你可以将数据从一个 TiDB 集群迁移至另一个 TiDB 集群：

- 拆库：原 TiDB 集群体量过大，或者为了避免原有的 TiDB 集群所承载的数个业务之间互相影响，将原 TiDB 集群中的部分表迁到另一个 TiDB 集群。
- 迁库：是对数据库的物理位置进行迁移，比如更换数据中心。

- 升级：在对数据正确性要求严苛的场景下，可以将数据迁移到一个更高版本的 TiDB 集群，确保数据安全。

本文将模拟整个迁移过程，具体包括以下四个步骤：

1. 搭建环境
2. 迁移全量数据
3. 迁移增量数据
4. 平滑切换业务

6.3.9.1 第 1 步：搭建环境

1. 部署集群。

   使用 TiUP Playground 快速部署上下游测试集群。更多部署信息，请参考TiUP 官方文档。

```
# 创建上游集群
tiup --tag upstream playground --host 0.0.0.0 --db 1 --pd 1 --kv 1 --tiflash 0 --ticdc 1
# 创建下游集群
tiup --tag downstream playground --host 0.0.0.0 --db 1 --pd 1 --kv 1 --tiflash 0 --ticdc 1
# 查看集群状态
tiup status
```

2. 初始化数据。

   测试集群中默认创建了 test 数据库，因此可以使用 sysbench 工具生成测试数据，用以模拟真实集群中的历史数据。

```
sysbench oltp_write_only --config-file=./tidb-config --tables=10 --table-size=10000 prepare
```

   这里通过 sysbench 运行 oltp_write_only 脚本，其将在测试数据库中生成 10 张表，每张表包含 10000 行初始数据。tidb-config 的配置如下：

```
mysql-host=172.16.6.122 # 这里需要替换为实际上游集群 ip
mysql-port=4000
mysql-user=root
mysql-password=
db-driver=mysql         # 设置数据库驱动为 mysql
mysql-db=test           # 设置测试数据库为 test
report-interval=10      # 设置定期统计的时间间隔为 10 秒
threads=10              # 设置 worker 线程数量为 10
time=0                  # 设置脚本总执行时间，0 表示不限制
rate=100                # 设置平均事务速率 tps = 100
```

3. 模拟业务负载。

   实际生产集群的数据迁移过程中，通常原集群还会写入新的业务数据，本文中可以通过 sysbench 工具模拟持续的写入负载，下面的命令会使用 10 个 worker 在数据库中的 sbtest1、sbtest2 和 sbtest3 三张表中持续写入数据，其总 tps 限制为 100。

```
sysbench oltp_write_only --config-file=./tidb-config --tables=3 run
```

4. 准备外部存储。

   在全量数据备份中，上下游集群均需访问备份文件，因此推荐使用备份存储存储备份文件，本文中通过 Minio 模拟兼容 S3 的存储服务：

```
wget https://dl.min.io/server/minio/release/linux-amd64/minio
chmod +x minio
# 配置访问 minio 的 access-key access-screct-id
export HOST_IP='172.16.6.122' # 替换为实际上游集群 ip
export MINIO_ROOT_USER='minio'
export MINIO_ROOT_PASSWORD='miniostorage'
# 创建数据目录， 其中 backup 为 bucket 的名称
mkdir -p data/backup
# 启动 minio，暴露端口在 6060
./minio server ./data --address :6060 &
```

   上述命令行启动了一个单节点的 minio server 模拟 S3 服务，其相关参数为：

   - Endpoint: http://$%7BHOST_IP%7D:6060/
   - Access-key: minio
   - Secret-access-key: miniostorage
   - Bucket: backup

   相应的访问链接为：

```
s3://backup?access-key=minio&secret-access-key=miniostorage&endpoint=http://${HOST_IP}:6060&
    ↪ force-path-style=true
```

6.3.9.2 第 2 步：迁移全量数据

搭建好测试环境后，可以使用 BR 工具的备份和恢复功能迁移全量数据。BR 工具有多种使用方式，本文中使用 SQL 语句BACKUP 和RESTORE 进行备份恢复。

> 注意：
>
> - 在生产集群中，关闭 GC 机制和备份操作会一定程度上降低集群的读性能，建议在业务低峰期进行备份，并设置合适的 RATE_LIMIT 限制备份操作对线上业务的影响。
> - 上下游集群版本不一致时，应检查 BR 工具的兼容性。本文假设上下游集群版本相同。

1. 关闭 GC。

   为了保证增量迁移过程中新写入的数据不丢失，在开始备份之前，需要关闭上游集群的垃圾回收 (GC) 机制，以确保系统不再清理历史数据。

执行如下命令关闭 GC：

```
MySQL [test]> SET GLOBAL tidb_gc_enable=FALSE;
```

```
Query OK, 0 rows affected (0.01 sec)
```

查询 tidb_gc_enable 的取值，判断 GC 是否已关闭：

```
MySQL [test]> SELECT @@global.tidb_gc_enable;
```

```
+-------------------------+:
| @@global.tidb_gc_enable |
+-------------------------+
|                       0 |
+-------------------------+
1 row in set (0.00 sec)
```

2. 备份数据。

   在上游集群中执行 BACKUP 语句备份数据：

```
MySQL [(none)]> BACKUP DATABASE * TO 's3://backup?access-key=minio&secret-access-key=
    ↪ miniostorage&endpoint=http://${HOST_IP}:6060&force-path-style=true' RATE_LIMIT = 120
    ↪  MB/SECOND;
```

```
+---------------+----------+--------------------+--------------------+--------------------+
    ↪
| Destination   | Size     | BackupTS           | Queue Time         | Execution Time
    ↪ |
+---------------+----------+--------------------+--------------------+--------------------+
    ↪
| s3://backup   | 10315858 | 431434047157698561 | 2022-02-25 19:57:59 | 2022-02-25 19:57:59
    ↪ |
+---------------+----------+--------------------+--------------------+--------------------+
    ↪
1 row in set (2.11 sec)
```

   备份语句提交成功后，TiDB 会返回关于备份数据的元信息，这里需要重点关注 BackupTS，它意味着该时间点之前数据会被备份，后边的教程中，本文将使用 BackupTS 作为数据校验截止时间和 TiCDC 增量扫描的开始时间。

3. 恢复数据。

   在下游集群中执行 RESTORE 语句恢复数据：

```
mysql> RESTORE DATABASE * FROM 's3://backup?access-key=minio&secret-access-key=miniostorage&
    ↪ endpoint=http://${HOST_IP}:6060&force-path-style=true';
```

```
+--------------+-----------+--------------------+--------------------+--------------------+
    ↪
| Destination  | Size      | BackupTS           | Queue Time         | Execution Time
    ↪ |
+--------------+-----------+--------------------+--------------------+--------------------+
    ↪
| s3://backup  | 10315858  | 431434141450371074 | 2022-02-25 20:03:59 | 2022-02-25 20:03:59
    ↪ |
+--------------+-----------+--------------------+--------------------+--------------------+
    ↪
1 row in set (41.85 sec)
```

4. （可选）校验数据。

   通过sync-diff-inspector 工具，可以验证上下游数据在某个时间点的一致性。从上述备份和恢复命令的输出可以看到，上游集群备份的时间点为 431434047157698561，下游集群完成数据恢复的时间点为 431434141450371074。

```
sync_diff_inspector -C ./config.yaml
```

   关于 sync-diff-inspector 的配置方法，请参考配置文件说明，在本文中，相应的配置如下：

```
# Diff Configuration.
######################### Datasource config #########################
[data-sources]
[data-sources.upstream]
        host = "172.16.6.122" # 需要替换为实际上游集群 ip
        port = 4000
        user = "root"
        password = ""
        snapshot = "431434047157698561" # 配置为实际的备份时间点（参见「备份」小节的
            ↪ BackupTS）
[data-sources.downstream]
        host = "172.16.6.125" # 需要替换为实际下游集群 ip
        port = 4000
        user = "root"
        password = ""

######################### Task config #########################
[task]
        output-dir = "./output"
        source-instances = ["upstream"]
        target-instance = "downstream"
        target-check-tables = ["*.*"]
```

6.3.9.3 第 3 步：迁移增量数据

1. 部署 TiCDC。

   完成全量数据迁移后，就可以部署并配置 TiCDC 集群同步增量数据，实际生产集群中请参考TiCDC 部署。本文在创建测试集群时，已经启动了一个 TiCDC 节点，因此可以直接进行 changefeed 的配置。

2. 创建同步任务。

   在上游集群中，执行以下命令创建从上游到下游集群的同步链路：

   ```
   tiup cdc cli changefeed create --server=http://172.16.6.122:8300 --sink-uri="mysql://root:
       ↪ @172.16.6.125:4000" --changefeed-id="upstream-to-downstream" --start-ts
       ↪ ="431434047157698561"
   ```

   以上命令中：

   - `--server`：TiCDC 集群中任意一个节点的地址
   - `--sink-uri`：同步任务下游的地址
   - `--changefeed-id`：同步任务的 ID，格式需要符合正则表达式 [6]+(-[a-zA-Z0-9]+)*$
   - `--start-ts`：TiCDC 同步的起点，需要设置为实际的备份时间点，也就是第 2 步：迁移全量数据中“备份数据”提到的 BackupTS

   更多关于 changefeed 的配置，请参考TiCDC Changefeed 配置参数。

3. 重新开启 GC。

   TiCDC 可以保证 GC 只回收已经同步的历史数据。因此，创建完从上游到下游集群的 changefeed 之后，就可以执行如下命令恢复集群的垃圾回收功能。详情请参考TiCDC GC safepoint 的完整行为。

执行如下命令打开 GC：

```
```sql
MySQL [test]> SET GLOBAL tidb_gc_enable=TRUE;
```


```
Query OK, 0 rows affected (0.01 sec)
```

查询 `tidb_gc_enable` 的取值，判断 GC 是否已开启：

```sql
MySQL [test]> SELECT @@global.tidb_gc_enable;
```


```
+--------------------------+
| @@global.tidb_gc_enable |
```

[6] a-zA-Z0-9
```

```
+-------------------------+
|                       1 |
+-------------------------+
1 row in set (0.00 sec)
```

6.3.9.4 第 4 步：平滑切换业务

通过 TiCDC 创建上下游的同步链路后，原集群的写入数据会以非常低的延迟同步到新集群，此时可以逐步将读流量迁移到新集群了。观察一段时间，如果新集群表现稳定，就可以将写流量接入新集群，步骤如下：

1. 停止上游集群的写业务。确认上游数据已全部同步到下游后，停止上游到下游集群的 changefeed。

```
# 停止旧集群到新集群的 changefeed
tiup cdc cli changefeed pause -c "upstream-to-downstream" --server=http://172.16.6.122:8300

# 查看 changefeed 状态
tiup cdc cli changefeed list
```

```
[
  {
    "id": "upstream-to-downstream",
    "summary": {
    "state": "stopped",  # 需要确认这里的状态为 stopped
    "tso": 431747241184329729,
    "checkpoint": "2022-03-11 15:50:20.387", # 确认这里的时间晚于停写的时间
    "error": null
    }
  }
]
```

2. 创建下游到上游集群的 changefeed。由于此时上下游数据是一致的，且没有新数据写入，因此可以不指定 start-ts，默认为当前时间：

```
tiup cdc cli changefeed create --server=http://172.16.6.125:8300 --sink-uri="mysql://root:
    ↪ @172.16.6.122:4000" --changefeed-id="downstream -to-upstream"
```

3. 将写业务迁移到下游集群，观察一段时间后，等新集群表现稳定，便可以弃用原集群。

6.3.10 从 TiDB 集群迁移数据至兼容 MySQL 的数据库

本文档介绍如何将数据从 TiDB 集群迁移至兼容 MySQL 的数据库，如 Aurora、MySQL、MariaDB 等。本文将模拟整个迁移过程，具体包括以下四个步骤：

1. 搭建环境

2. 迁移全量数据
3. 迁移增量数据
4. 平滑切换业务

6.3.10.1 第 1 步：搭建环境

1. 部署上游 TiDB 集群。

   使用 TiUP Playground 快速部署上下游测试集群。更多部署信息，请参考TiUP 官方文档。

   ```
   # 创建上游集群
   tiup playground --db 1 --pd 1 --kv 1 --tiflash 0 --ticdc 1
   # 查看集群状态
   tiup status
   ```

2. 部署下游 MySQL 实例。

   - 在实验环境中，可以使用 Docker 快速部署 MySQL 实例，执行如下命令：

     ```
     docker run --name some-mysql -e MYSQL_ROOT_PASSWORD=my-secret-pw -p 3306:3306 -d mysql
     ```

   - 在生产环境中，可以参考 Installing MySQL 来部署 MySQL 实例。

3. 模拟业务负载。

   在测试实验环境下，可以使用 go-tpc 向上游 TiDB 集群写入数据，以让 TiDB 产生事件变更数据。执行如下命令，将首先在上游 TiDB 创建名为 tpcc 的数据库，然后使用 TiUP bench 写入数据到刚创建的 tpcc 数据库中。

   ```
   tiup bench tpcc -H 127.0.0.1 -P 4000 -D tpcc --warehouses 4 prepare
   tiup bench tpcc -H 127.0.0.1 -P 4000 -D tpcc --warehouses 4 run --time 300s
   ```

   关于 go-tpc 的更多详细内容，可以参考如何对 TiDB 进行 TPC-C 测试。

6.3.10.2 第 2 步：迁移全量数据

搭建好测试环境后，可以使用Dumpling 工具导出上游集群的全量数据。

> 注意：
>
> 在生产集群中，关闭 GC 机制和备份操作会一定程度上降低集群的读性能，建议在业务低峰期进行备份，并设置合适的 `RATE_LIMIT` 限制备份操作对线上业务的影响。

1. 关闭 GC (Garbage Collection)。

   为了保证增量迁移过程中新写入的数据不丢失，在开始全量导出之前，需要关闭上游集群的垃圾回收 (GC) 机制，以确保系统不再清理历史数据。

   执行如下命令关闭 GC：

```
MySQL [test]> SET GLOBAL tidb_gc_enable=FALSE;
```

```
Query OK, 0 rows affected (0.01 sec)
```

查询 tidb_gc_enable 的取值，判断 GC 是否已关闭：

```
MySQL [test]> SELECT @@global.tidb_gc_enable;
```

```
+--------------------------+:
| @@global.tidb_gc_enable |
+--------------------------+
|                        0 |
+--------------------------+
1 row in set (0.00 sec)
```

2. 备份数据。

    1. 使用 Dumpling 导出 SQL 格式的数据：

    ```
    tiup dumpling -u root -P 4000 -h 127.0.0.1 --filetype sql -t 8 -o ./dumpling_output -r
        ↪ 200000 -F256MiB
    ```

    2. 导出完毕后，执行如下命令查看导出数据的元信息，metadata 文件中的 Pos 就是导出快照的 TSO，将其记录为 BackupTS：

    ```
    cat dumpling_output/metadata
    ```

    ```
    Started dump at: 2022-06-28 17:49:54
    SHOW MASTER STATUS:
            Log: tidb-binlog
            Pos: 434217889191428107
            GTID:

    Finished dump at: 2022-06-28 17:49:57
    ```

3. 恢复数据。

    使用开源工具 MyLoader 导入数据到下游 MySQL。MyLoader 的安装和详细用例参见 MyDumpler/MyLoader。执行以下指令，将 Dumpling 导出的上游全量数据导入到下游 MySQL 实例：

    ```
    myloader -h 127.0.0.1 -P 3306 -d ./dumpling_output/
    ```

4. （可选）校验数据。

    通过sync-diff-inspector 工具，可以验证上下游数据在某个时间点的一致性。

    ```
    sync_diff_inspector -C ./config.yaml
    ```

关于 sync-diff-inspector 的配置方法，请参考配置文件说明。在本文中，相应的配置如下：

```
# Diff Configuration.
######################### Datasource config #########################
[data-sources]
[data-sources.upstream]
        host = "127.0.0.1" # 需要替换为实际上游集群 ip
        port = 4000
        user = "root"
        password = ""
        snapshot = "434217889191428107" # 配置为实际的备份时间点（参见「备份」小节的
            ↪ BackupTS）
[data-sources.downstream]
        host = "127.0.0.1" # 需要替换为实际下游集群 ip
        port = 3306
        user = "root"
        password = ""

######################### Task config #########################
[task]
        output-dir = "./output"
        source-instances = ["upstream"]
        target-instance = "downstream"
        target-check-tables = ["*.*"]
```

6.3.10.3 第 3 步：迁移增量数据

1. 部署 TiCDC。

   完成全量数据迁移后，就可以部署并配置 TiCDC 集群同步增量数据，实际生产集群中请参考TiCDC 部署。本文在创建测试集群时，已经启动了一个 TiCDC 节点，因此可以直接进行 changefeed 的配置。

2. 创建同步任务。

   在上游集群中，执行以下命令创建从上游到下游集群的同步链路：

```
tiup ctl:v<CLUSTER_VERSION> cdc changefeed create --server=http://127.0.0.1:8300 --sink-uri
    ↪ ="mysql://root:@127.0.0.1:3306" --changefeed-id="upstream-to-downstream" --start-ts
    ↪ ="434217889191428107"
```

   以上命令中：

   - `--server`：TiCDC 集群任意一节点地址
   - `--sink-uri`：同步任务下游的地址
   - `--changefeed-id`：同步任务的 ID，格式需要符合正则表达式 `^[a-zA-Z0-9]+(\-[a-zA-Z0-9]+)*$`
   - `--start-ts`：TiCDC 同步的起点，需要设置为实际的备份时间点，也就是第 2 步：迁移全量数据中“备份数据”提到的 BackupTS

更多关于 changefeed 的配置，请参考TiCDC Changefeed 配置参数。

3. 重新开启 GC。

   TiCDC 可以保证 GC 只回收已经同步的历史数据。因此，创建完从上游到下游集群的 changefeed 之后，就可以执行如下命令恢复集群的垃圾回收功能。详情请参考TiCDC GC safepoint 的完整行为。

   执行如下命令打开 GC：

```
MySQL [test]> SET GLOBAL tidb_gc_enable=TRUE;
```

```
Query OK, 0 rows affected (0.01 sec)
```

   查询 tidb_gc_enable 的取值，判断 GC 是否已开启：

```
MySQL [test]> SELECT @@global.tidb_gc_enable;
```

```
+-------------------------+
| @@global.tidb_gc_enable |
+-------------------------+
|                       1 |
+-------------------------+
1 row in set (0.00 sec)
```

6.3.10.4 第 4 步：平滑切换业务

通过 TiCDC 创建上下游的同步链路后，原集群的写入数据会以非常低的延迟同步到新集群，此时可以逐步将读流量迁移到新集群了。观察一段时间，如果新集群表现稳定，就可以将写流量接入新集群，步骤如下：

1. 停止上游集群的写业务。确认上游数据已全部同步到下游后，停止上游到下游集群的 changefeed。

```
# 停止旧集群到新集群的 changefeed
tiup cdc cli changefeed pause -c "upstream-to-downstream" --pd=http://172.16.6.122:2379

# 查看 changefeed 状态
tiup cdc cli changefeed list
```

```
[
  {
    "id": "upstream-to-downstream",
    "summary": {
    "state": "stopped",  # 需要确认这里的状态为 stopped
    "tso": 434218657561968641,
    "checkpoint": "2022-06-28 18:38:45.685", # 确认这里的时间晚于停写的时间
    "error": null
    }
  }
]
```

2. 将写业务迁移到下游集群，观察一段时间后，等新集群表现稳定，便可以弃用原集群。

6.4 复杂迁移场景

6.4.1 上游使用 pt-osc/gh-ost 工具的持续同步场景

在生产业务中执行 DDL 时，产生的锁表操作会一定程度阻塞数据库的读取或者写入。为了把对读写的影响降到最低，用户往往会选择 online DDL 工具执行 DDL。常见的 Online DDL 工具有 gh-ost 和 pt-osc。

在使用 DM 完成 MySQL 到 TiDB 的数据迁移时，可以开启`online-ddl`配置，实现 DM 工具与 gh-ost 或 pt-osc 的协同。

具体迁移操作可参考已有数据迁移场景：

- 从小数据量 MySQL 迁移数据到 TiDB
- 从大数据量 MySQL 迁移数据到 TiDB
- 从小数据量分库分表 MySQL 合并迁移数据到 TiDB
- 从大数据量分库分表 MySQL 合并迁移数据到 TiDB

6.4.1.1 开启 DM 的 online-ddl 特性

配置 DM 的任务配置文件时，将全局参数的`online-ddl`设置为 true，具体配置示例如下图：

```
### ----------- 全局配置 -----------
#### ********* 基本信息配置 *********
name: test                      # 任务名称，需要全局唯一
task-mode: all                  # 任务模式，可设为 "full"、"incremental"、"all"
shard-mode: "pessimistic"       # 如果为分库分表合并任务则需要配置该项。默认使用悲观协调模式 "
    ↪ pessimistic"，在深入了解乐观协调模式的原理和使用限制后，也可以设置为乐观协调模式 "
    ↪ optimistic"
meta-schema: "dm_meta"          # 下游储存 `meta` 信息的数据库
online-ddl: true                # 开启 DM 的 online DDL 支持特性。兼容上游使用 gh-ost 、pt-osc
    ↪ 两种工具的自动处理
```

6.4.1.2 开启 online-ddl 的影响

当开启`online-ddl`特性后，DM 同步 gh-ost 或 pt-osc 工具所产生的 DDL 语句将会发生一些变化：

上游 gh-ost 或 pt-osc 工具工作流如下：

- 根据 DDL 目标表 (real table) 的表结构新建一张镜像表 (ghost table)；
- 在镜像表上应用变更 DDL；
- 将 DDL 目标表的数据同步到镜像表；
- 在目标表与镜像表数据一致后，通过 RENAME 语句使镜像表替换掉目标表。

`online-ddl=true` 时 DM 的同步方式：

- 不在下游新建镜像表 (ghost table)；
- 记录变更 DDL；

- 仅从镜像表同步数据；
- 在下游执行 DDL 变更。

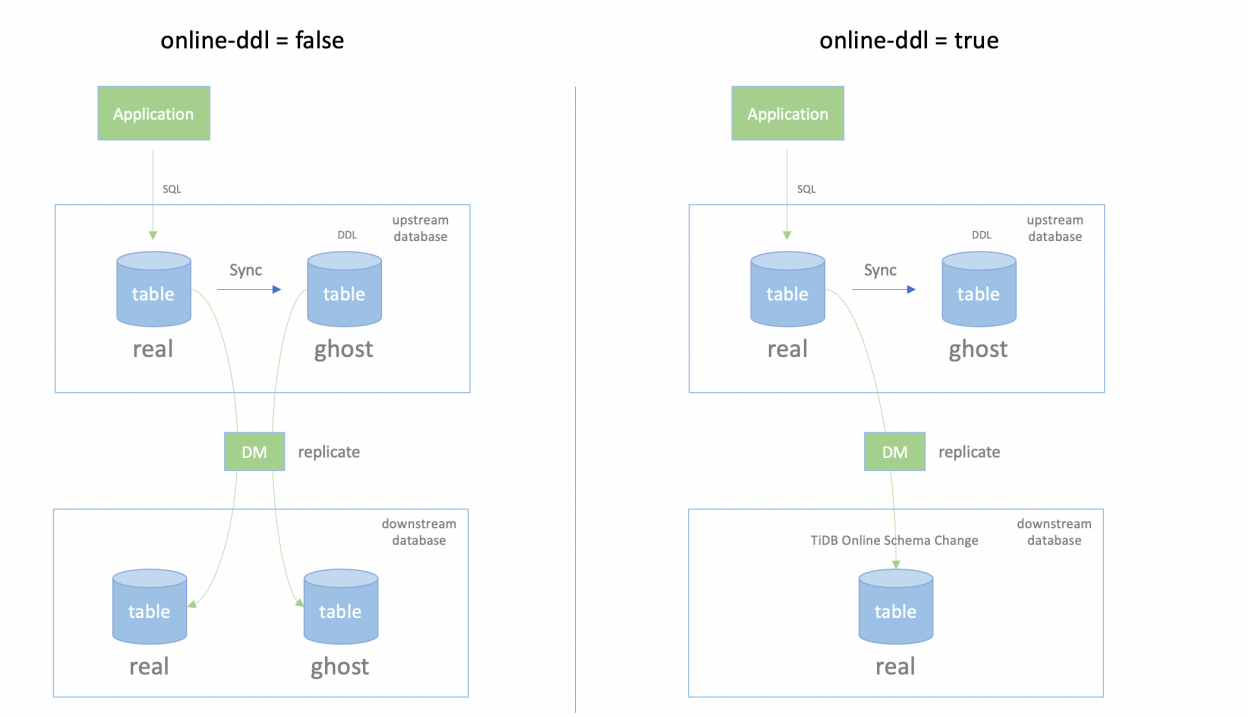


图 27: dm-online-ddl

这些变化将带来一些好处：

- 下游 TiDB 无需创建和同步镜像表，节约相应存储空间和网络传输等开销；
- 在分库分表合并场景下，自动忽略各分表镜像表的 RENAME 操作，保证同步正确性。

如果您想深入了解其实现原理，请阅读以下两篇技术博客：

- DM 源码阅读系列文章（八）Online Schema Change 迁移支持
- TiDB Online Schema Change 原理

6.4.1.3 探索更多

- DM 与 online DDL 工具协作细节

6.4.2 下游存在更多列的迁移场景

本文档介绍数据同步时，下游存在更多列的迁移场景需要的注意事项。具体迁移操作可参考已有数据迁移场景：

- 从小数据量 MySQL 迁移数据到 TiDB
- 从大数据量 MySQL 迁移数据到 TiDB
- 从小数据量分库分表 MySQL 合并迁移数据到 TiDB
- 从大数据量分库分表 MySQL 合并迁移数据到 TiDB

6.4.2.1 使用 DM 迁移至存在更多列的下游

DM 同步上游的 binlog 时，会尝试使用下游当前的表结构来解析 binlog 并生成相应的 DML 语句。如果上游的 binlog 里数据表的列数与下游表结构的列数不一致，则会产生如下错误：

```
"errors": [
    {
        "ErrCode": 36027,
        "ErrClass": "sync-unit",
        "ErrScope": "internal",
        "ErrLevel": "high",
        "Message": "startLocation: [position: (mysql-bin.000001, 2022), gtid-set:09bec856-ba95-11
            ↪ ea-850a-58f2b4af5188:1-9 ], endLocation: [position: (mysql-bin.000001, 2022),
            ↪ gtid-set: 09bec856-ba95-11ea-850a-58f2b4af5188:1-9]: gen insert sqls failed,
            ↪ schema: log, table: messages: Column count doesn't match value count: 3 (columns)
            ↪  vs 2 (values)",
        "RawCause": "",
        "Workaround": ""
    }
]
```

例如上游表结构为：

```
### 上游表结构
CREATE TABLE `messages` (
  `id` int(11) NOT NULL,
  PRIMARY KEY (`id`)
)
```

例如下游表结构为：

```
### 下游表结构
CREATE TABLE `messages` (
  `id` int(11) NOT NULL,
  `message` varchar(255) DEFAULT NULL, # 下游比上游多出的列。
  PRIMARY KEY (`id`)
)
```

当 DM 尝试使用下游表结构解析上游产生的 binlog event 时，DM 会报出上述 `Column count doesn't match` 错误。

此时，你可以使用 `binlog-schema` 命令来为数据源中需要迁移的表指定表结构，表结构需要对应 DM 将要开始同步的 binlog event 的数据。如果你在进行分表合并的数据迁移，那么需要为每个分表按照如下步骤在 DM 中设置用于解析 binlog event 的表结构。具体操作为：

1. 在 DM 中，新建一个 .sql 文件，并将上游表结构对应的 CREATE TABLE 语句添加到该文件。例如，将以下表结构保存到 log.messages.sql 中。如果是 6.0 及以上版本，可以直接通过 --from-source/--from-target 更新，无需创建 SQL 文件。可参考管理迁移表的表结构。

```
# 上游表结构
CREATE TABLE `messages` (
`id` int(11) NOT NULL,
PRIMARY KEY (`id`)
)
```

2. 使用 binlog-schema 命令为数据源中需要迁移的表设置表结构（此时数据迁移任务应该由于上述 Column count doesn't match 错误而处于 Paused 状态）。

```
tiup dmctl --master-addr ${advertise-addr} binlog-schema update -s ${source-id} ${task-name}
    ↪  ${database-name} ${table-name} ${schema-file}
```

该命令中的参数描述如下：

参数	描述
--master-addr	指定 dmctl 要连接的集群的任意 DM-master 节点的 ${advertise-addr}。${advertise-addr} 表示 DM-master 向外界宣告的地址。

参数	描述
binlog ↪ -schema ↪ update ↪	手动更新schema信息
-s	指定source。
${ ↪ task ↪ - ↪ name ↪ }	指定task。表示数据同步任务配置文件task.↪ yaml ↪ 中定义的同步任务名称。
${ ↪ database ↪ - ↪ name ↪ }	指定database。表示上游数据库名。

参数	描述
${ ↪ table ↪ - ↪ name ↪ }	指定 table。表示上游数据表名。
${ ↪ schema ↪ - ↪ file ↪ }	指定表的 schema 文件。表示将被设置的表结构文件。

例如：

```
tiup dmctl --master-addr 172.16.10.71:8261 binlog-schema update -s mysql-01 task-test -d log
    ↪  -t message log.message.sql
```

3. 使用 `resume-task` 命令恢复处于 Paused 状态的同步任务。

```
tiup dmctl --master-addr ${advertise-addr} resume-task ${task-name}
```

4. 使用 `query-status` 命令确认数据迁移任务是否运行正常。

```
tiup dmctl --master-addr ${advertise-addr} query-status resume-task ${task-name}
```

6.4.3 如何过滤 binlog 事件

本文档介绍使用 DM 持续增量数据同步时，如何过滤 binlog 事件。具体迁移操作可参考已有数据迁移场景：

- 从小数据量 MySQL 迁移数据到 TiDB
- 从大数据量 MySQL 迁移数据到 TiDB
- 从小数据量分库分表 MySQL 合并迁移数据到 TiDB
- 从大数据量分库分表 MySQL 合并迁移数据到 TiDB

6.4.3.1 配置方式

配置 DM 的任务配置文件时，增加如下`filter`，具体配置示例如下图：

```
filters:
  rule-1:
    schema-pattern: "test_*"
    table-pattern: "t_*"
    events: ["truncate table", "drop table"]
    sql-pattern: ["^DROP\\s+PROCEDURE", "^CREATE\\s+PROCEDURE"]
    action: Ignore
```

- `schema-pattern/table-pattern`：对匹配上的 schema 或 table 进行过滤
- `events`：binlog events，支持的 Event 如下表所示:

Event	分类	说明
all		匹配所有 events
all dml		匹配所有 DML events
all ddl		匹配所有 DDL events
none		不匹配任何 events
none ddl		不包含任何 DDL events
none dml		不包含任何 DML events
insert	DML	匹配 insert DML event
update	DML	匹配 update DML event
delete	DML	匹配 delete DML event
create database	DDL	匹配 create database event
drop database	DDL	匹配 drop database event
create table	DDL	匹配 create table event
create index	DDL	匹配 create index event
drop table	DDL	匹配 drop table event
truncate table	DDL	匹配 truncate table event
rename table	DDL	匹配 rename table event
drop index	DDL	匹配 drop index event
alter table	DDL	匹配 alter table event

- `sql-pattern`：匹配指定的 DDL SQL 语句，支持正则表达式匹配。
- `action`：可取值 Do 或 Ignore。
  - `Do`：白名单。binlog event 如果满足下面两个条件之一将会被同步：
    * 符合 events 条件；
    * sql-pattern 不为空，且对应的 SQL 可以匹配上 sql-pattern 中任意一项。
  - `Ignore`：黑名单。如果满足下面两个条件之一就会被过滤掉：
    * 符合 events 条件；
    * sql-pattern 不为空，且对应的 SQL 可以匹配上 sql-pattern 中任意一项

注意：如果同时配置 `Do/Ignore`，则 `Ignore` 优先级更高。`binlog event` 不匹配白名单或者匹配黑名单都将被直接过滤。

6.4.3.2 使用场景举例

6.4.3.2.1 过滤分库分表的所有删除操作

设置 `filter-table-rule` 和 `filter-schema-rule` 两个过滤规则，具体如下：

```
filters:
  filter-table-rule:
    schema-pattern: "test_*"
    table-pattern: "t_*"
    events: ["truncate table", "drop table", "delete"]
    action: Ignore
  filter-schema-rule:
    schema-pattern: "test_*"
    events: ["drop database"]
    action: Ignore
```

6.4.3.2.2 只迁移分库分表的 DML 操作

设置两个 `Binlog event filter rule`：

```
filters:
  do-table-rule:
    schema-pattern: "test_*"
    table-pattern: "t_*"
    events: ["create table", "all dml"]
    action: Do
  do-schema-rule:
    schema-pattern: "test_*"
    events: ["create database"]
    action: Do
```

6.4.3.2.3 过滤 TiDB 不支持的 SQL 语句

```
filters:
  filter-procedure-rule:
    schema-pattern: "*"
    sql-pattern: [".*\\s+DROP\\s+PROCEDURE", ".*\\s+CREATE\\s+PROCEDURE", "ALTER\\s+TABLE[\\s\\S
        ↪ ]*ADD\\s+PARTITION", "ALTER\\s+TABLE[\\s\\S]*DROP\\s+PARTITION"]
    action: Ignore
```

> **注意：**
>
> 全局过滤规则的设置必须尽可能严格，以避免过滤掉需要迁移的数据。

6.4.3.3 探索更多

- 如何通过 SQL 表达式过滤 binlog

6.4.4 如何通过 SQL 表达式过滤 DML

本文档介绍使用 DM 持续增量数据同步时，如何更加精细的过滤 binlog 事件。具体迁移操作可参考已有数据迁移场景：

- 从小数据量 MySQL 迁移数据到 TiDB
- 从大数据量 MySQL 迁移数据到 TiDB
- 从小数据量分库分表 MySQL 合并迁移数据到 TiDB
- 从大数据量分库分表 MySQL 合并迁移数据到 TiDB

在进行增量数据迁移时，可以通过如何过滤 binlog 事件功能过滤某些类型的 binlog event，例如不向下游迁移 DELETE 事件以达到归档、审计等目的。但是 binlog event filter 无法以更细粒度判断某一行的 DELETE 事件是否要被过滤。

为了解决上述问题，从 v2.0.5 起，DM 支持在增量数据同步阶段使用`binlog value filter`过滤迁移数据。DM 支持的 ROW 格式的 binlog 中，binlog event 带有所有列的值。你可以基于这些值配置 SQL 表达式。如果该表达式对于某条行变更的计算结果是 TRUE，DM 就不会向下游迁移该条行变更。

与如何过滤 binlog 事件类似，表达式过滤需要在数据迁移任务配置文件里配置，详见下面配置样例。完整的配置及意义，可以参考DM 完整配置文件示例：

```
name: test
task-mode: all

mysql-instances:
  - source-id: "mysql-replica-01"
    expression-filters: ["even_c"]

expression-filter:
  even_c:
    schema: "expr_filter"
    table: "tbl"
    insert-value-expr: "c % 2 = 0"
```

上面的示例配置了 even_c 规则，并让 source ID 为 `mysql-replica-01` 的数据源引用了该规则。even_c 规则的含义是：对于 `expr_filter` 库下的 `tbl` 表，当插入的 c 的值为偶数 (`c % 2 = 0`) 时，不将这条插入语句迁移到下游。下面展示该规则的使用效果。

在上游数据源增量插入以下数据：

```
INSERT INTO tbl(id, c) VALUES (1, 1), (2, 2), (3, 3), (4, 4);
```

随后在下游查询 `tbl` 表，可见只有 c 的值为单数的行迁移到了下游：

```
MySQL [test]> select * from tbl;

+------+------+
| id   | c    |
+------+------+
|    1 |    1 |
|    3 |    3 |
+------+------+

2 rows in set (0.001 sec)
```

6.4.4.1 配置参数及规则说明

- schema：要匹配的上游数据库库名，不支持通配符匹配或正则匹配。
- table: 要匹配的上游表名，不支持通配符匹配或正则匹配。
- insert-value-expr：配置一个表达式，对 INSERT 类型的 binlog event (WRITE_ROWS_EVENT) 带有的值生效。不能与 update-old-value-expr、update-new-value-expr、delete-value-expr 出现在一个配置项中。
- update-old-value-expr：配置一个表达式，对 UPDATE 类型的 binlog event (UPDATE_ROWS_EVENT) 更新对应的旧值生效。不能与 insert-value-expr、delete-value-expr 出现在一个配置项中。
- update-new-value-expr：配置一个表达式，对 UPDATE 类型的 binlog event (UPDATE_ROWS_EVENT) 更新对应的新值生效。不能与 insert-value-expr、delete-value-expr 出现在一个配置项中。
- delete-value-expr：配置一个表达式，对 DELETE 类型的 binlog event (DELETE_ROWS_EVENT) 带有的值生效。不能与 insert-value-expr、update-old-value-expr、update-new-value-expr 出现在一个配置项中。

> 注意：
>
> - update-old-value-expr 可以与 update-new-value-expr 同时配置。
> - 当二者同时配置时，会将“更新 + 旧值“满足update-old-value-expr 且” 更新 + 新值“满足 update-new-value-expr 的行过滤掉。
> - 当只配置一者时，配置的这条表达式会决定是否过滤整个行变更，即旧值的删除和新值的插入会作为一个整体被过滤掉。

SQL 表达式可以涉及一列或多列，也可使用 TiDB 支持的 SQL 函数，例如 `c % 2 = 0`、`a*a + b*b = c*c`、`ts > NOW()`。

TIMESTAMP 类型的默认时区是任务配置文件中指定的时区，默认值是下游时区。可以使用 `c_timestamp = '2021-01-01 12:34:56.5678+08:00'` 的方式显式指定时区。

配置项 expression-filter 下可以定义多条过滤规则，上游数据源在其 expression-filters 配置项中引用需要的规则使其生效。当有多条规则生效时，匹配到任意一条规则即会导致某个行变更被过滤。

> **注意：**
>
> 为某张表设置过多的表达式过滤会增加 DM 的计算开销，可能导致数据迁移速度变慢。

7 数据集成

7.1 数据集成概述

数据集成一般是指数据在各个独立的数据源之间流动、转换和汇集。随着数据量的爆炸式增长和数据价值被深度挖掘，对数据集成的需求越来越普遍和迫切。为了避免 TiDB 成为数据孤岛，顺利与各个数据系统进行集成，TiCDC 提供将 TiDB 增量数据变更日志实时同步到其他数据系统的能力。本文介绍一些常用的数据集成场景，你可以依据这些场景选择最适合自己的数据集成方案。

7.1.1 与 Confluent Cloud 和 Snowflake 进行数据集成

你可以使用 TiCDC 将 TiDB 的增量数据同步到 Confluent Cloud，并借助 Confluent Cloud 的能力最终将数据分别同步到 Snowflake、 ksqlDB、 SQL Server。 参见与 Confluent Cloud 和 Snowflake 进行数据集成。

7.1.2 与 Apache Kafka 和 Apache Flink 进行数据集成

你可以使用 TiCDC 将 TiDB 的增量数据同步到 Apache Kafka，并使用 Apache Flink 消费 Kafka 中的数据。参见与 Apache Kafka 和 Apache Flink 进行数据集成。

7.2 数据集成场景

7.2.1 与 Confluent Cloud 和 Snowflake 进行数据集成

Confluent 是一个兼容 Apache Kafka 的数据流平台，能够访问、存储和管理连续的实时流数据，具备丰富的数据集成能力。自 v6.1.0 开始，TiCDC 支持将增量变更数据以 Avro 格式输出到 Confluent。本文档介绍如何使用TiCDC 将 TiDB 的增量数据同步到 Confluent Cloud，并借助 Confluent Cloud 的能力最终将数据分别同步到 Snowflake、 ksqlDB、 SQL Server。 主要内容包括：

- 快速搭建包含 TiCDC 的 TiDB 集群
- 创建将数据输出到 Confluent Cloud 的 changefeed
- 创建将数据从 Confluent Cloud 输出到 Snowflake、 ksqlDB 和 SQL Server 的连接器 (Connector)
- 使用 go-tpc 写入数据到上游 TiDB，并观察 Snowflake、 ksqlDB 和 SQL Server 中的数据

上述过程将会基于实验环境进行，你也可以参考上述执行步骤，搭建生产级别的集群。

7.2.1.1 输出增量数据到 Confluent Cloud

7.2.1.1.1 第 1 步：搭建环境

1. 部署包含 TiCDC 的 TiDB 集群。

   在实验或测试环境中，可以使用 TiUP Playground 功能快速部署 TiCDC，命令如下：

   ```
   tiup playground --host 0.0.0.0 --db 1 --pd 1 --kv 1 --tiflash 0 --ticdc 1
   # 查看集群状态
   tiup status
   ```

   如果尚未安装 TiUP，可以参考安装 TiUP。在生产环境下，可以参考使用 TiUP 安装部署 TiCDC 集群，完成 TiCDC 集群部署工作。

2. 注册 Confluent Cloud 并创建 Confluent 集群。

   创建 Basic 集群并开放 Internet 访问，详见 Quick Start for Confluent Cloud。

7.2.1.1.2 第 2 步：创建 Access Key Pair

1. 创建 Cluster API Key。

   在 Confluent 集群控制面板中依次点击 Data integration > API keys > Create key。在弹出的 Select scope for API key 页面，选择 Global access。

   创建成功后会得到一个 Key Pair 文件，内容如下：

   ```
   === Confluent Cloud API key: xxx-xxxxx ===

   API key:
   L5WWA4GK4NAT2EQV

   API secret:
   xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

   Bootstrap server:
   xxx-xxxxx.ap-east-1.aws.confluent.cloud:9092
   ```

2. 记录 Schema Registry Endpoints。

   在 Confluent 集群控制面板中，选择 Schema Registry > API endpoint，记录 Schema Registry Endpoints，如下：

   ```
   https://yyy-yyyyy.us-east-2.aws.confluent.cloud
   ```

3. 创建 Schema Registry API key。

   在 Confluent 集群控制面板中，选择 Schema Registry > API credentials，点击 Edit 和 Create key。

   创建成功后会得到一个 Key Pair 文件，内容如下：

   ```
   === Confluent Cloud API key: yyy-yyyyy ===

   API key:
   ```

```
7NBH2CAFM2LMGTH7

API secret:
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

以上步骤也可以通过 Confluent CLI 实现，详见 Connect Confluent CLI to Confluent Cloud Cluster。

7.2.1.1.3 第 3 步：创建 Kafka changefeed

1. 创建 changefeed 配置文件。

   根据 Avro 协议和 Confluent Connector 的要求和规范，每张表的增量数据需要发送到独立的 Topic 中，并且每个事件需要按照主键值分发 Partition。因此，需要创建一个名为 `changefeed.conf` 的配置文件，填写如下内容：

   ```
   [sink]
   dispatchers = [
   {matcher = ['*.*'], topic = "tidb_{schema}_{table}", partition="index-value"},
   ]
   ```

   关于配置文件中 `dispatchers` 的详细解释，参考自定义 Kafka Sink 的 Topic 和 Partition 的分发规则。

2. 创建一个 changefeed，将增量数据输出到 Confluent Cloud：

   ```
   tiup ctl:v<CLUSTER_VERSION> cdc changefeed create --server="http://127.0.0.1:8300" --sink-
       ↪ uri="kafka://<broker_endpoint>/ticdc-meta?protocol=avro&replication-factor=3&enable-
       ↪ tls=true&auto-create-topic=true&sasl-mechanism=plain&sasl-user=<broker_api_key>&sasl
       ↪ -password=<broker_api_secret>" --schema-registry="https://<schema_registry_api_key
       ↪ >:<schema_registry_api_secret>@<schema_registry_endpoint>" --changefeed-id="
       ↪ confluent-changefeed" --config changefeed.conf
   ```

   将如下字段替换为第 2 步：创建 Access Key Pair中创建和记录的值：

   - `<broker_endpoint>`
   - `<broker_api_key>`
   - `<broker_api_secret>`
   - `<schema_registry_api_key>`
   - `<schema_registry_api_secret>`
   - `<schema_registry_endpoint>`

   其中 `<schema_registry_api_secret>` 需要经过 HTML URL 编码后再替换，替换完毕后示例如下：

   ```
   tiup ctl:v<CLUSTER_VERSION> cdc changefeed create --server="http://127.0.0.1:8300" --sink-
       ↪ uri="kafka://xxx-xxxxx.ap-east-1.aws.confluent.cloud:9092/ticdc-meta?protocol=avro&
       ↪ replication-factor=3&enable-tls=true&auto-create-topic=true&sasl-mechanism=plain&
       ↪ sasl-user=L5WWA4GK4NAT2EQV&sasl-password=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx" --
       ↪ schema-registry="https://7NBH2CAFM2LMGTH7:xxxxxxxxxxxxxxxxx@yyy-yyyyy.us-east-2.aws
       ↪ .confluent.cloud" --changefeed-id="confluent-changefeed" --config changefeed.conf
   ```

- 如果命令执行成功，将会返回被创建的 changefeed 的相关信息，包含被创建的 changefeed 的 ID 以及相关信息，内容如下：

```
Create changefeed successfully!
ID: confluent-changefeed
Info: {... changfeed info json struct ...}
```

- 如果命令长时间没有返回，请检查当前执行命令所在服务器到 Confluent Cloud 之间网络可达性，参考 Test connectivity to Confluent Cloud。

3. Changefeed 创建成功后，执行如下命令，查看 changefeed 的状态：

```
tiup ctl:v<CLUSTER_VERSION> cdc changefeed list --server="http://127.0.0.1:8300"
```

可以参考管理 Changefeed，对 changefeed 状态进行管理。

7.2.1.1.4 第 4 步：写入数据以产生变更日志

完成以上步骤后，TiCDC 会将上游 TiDB 的增量数据变更日志发送到 Confluent Cloud。本小节将对 TiDB 写入数据，以产生增量数据变更日志。

1. 模拟业务负载。

在测试实验环境下，可以使用 go-tpc 向上游 TiDB 集群写入数据，以让 TiDB 产生事件变更数据。执行以下命令，会首先在上游 TiDB 创建名为 tpcc 的数据库，然后使用 TiUP bench 写入数据到这个数据库中。

```
tiup bench tpcc -H 127.0.0.1 -P 4000 -D tpcc --warehouses 4 prepare
tiup bench tpcc -H 127.0.0.1 -P 4000 -D tpcc --warehouses 4 run --time 300s
```

关于 go-tpc 的更多详细内容，可以参考如何对 TiDB 进行 TPC-C 测试。

2. 观察 Confluent 中数据传输情况。

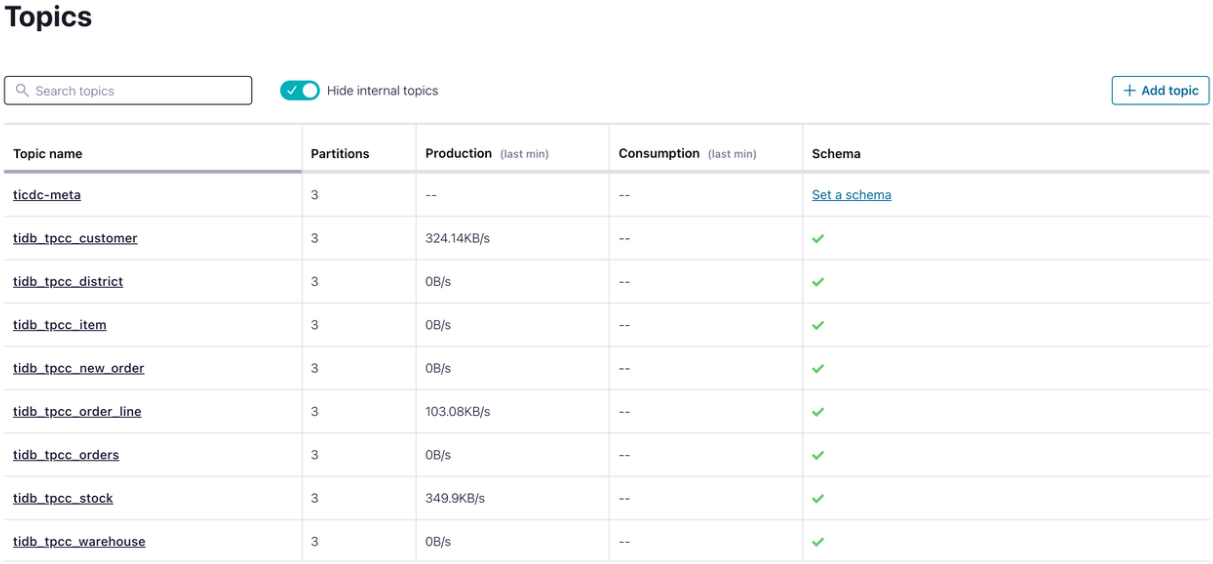


图 28: Confluent topics

在 Confluent 集群控制面板中，可以观察到相应的 Topic 已经被自动创建，并有数据正在写入。至此，TiDB 数据库中的增量数据就被成功输出到了 Confluent Cloud。

7.2.1.2 与 Snowflake 进行数据集成

Snowflake 是一种云原生数据仓库。借助 Confluent 的能力，你只需要创建 Snowflake Sink Connector，就可以将 TiDB 的增量数据输出到 Snowflake。

7.2.1.2.1 准备工作

- 注册和创建 Snowflake 集群，参考 Getting Started with Snowflake。
- 连接到 Snowflake 前，为 Snowflake 添加 Private Key，参考 Key Pair Authentication & Key Pair Rotation。

7.2.1.2.2 集成步骤

1. 在 Snowflake 中创建 Database 和 Schema。

   在 Snowflake 控制面板中，选择 Data > Database。创建名为 TPCC 的 Database 和名为 TiCDC 的 Schema。

2. 在 Confluent 集群控制面板中，选择 Data integration > Connectors > Snowflake Sink，进入如下页面：

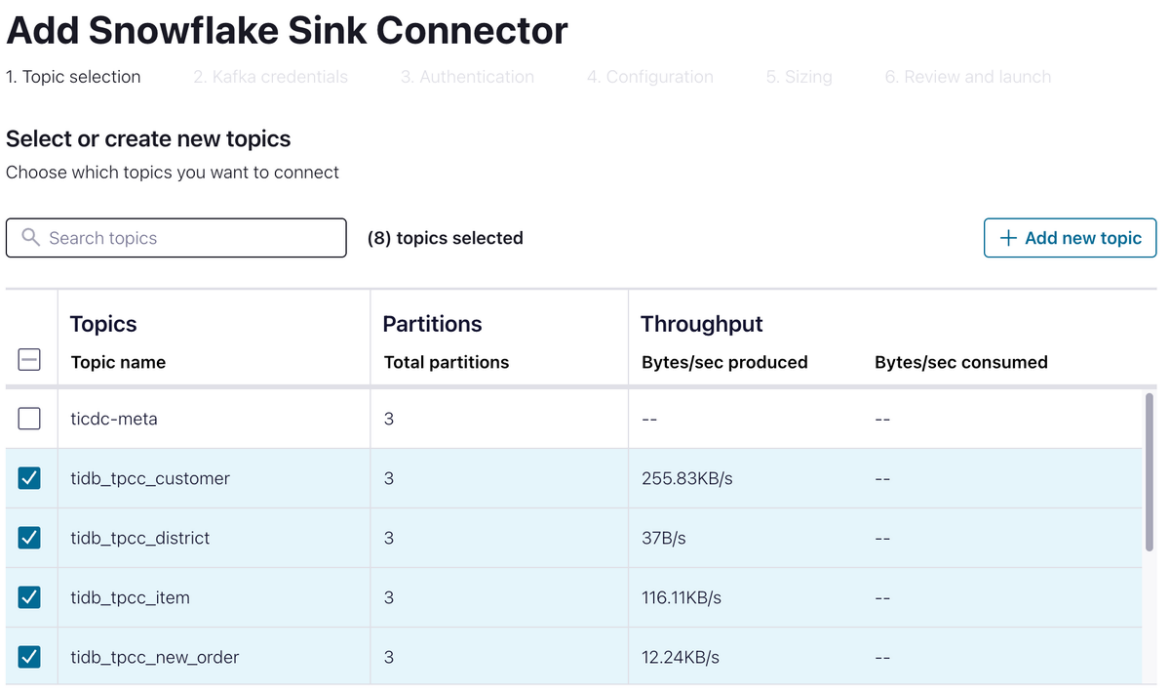


图 29: Add snowflake sink connector

3. 选择需要同步到 Snowflake 的 Topic 后，进入下一页面：

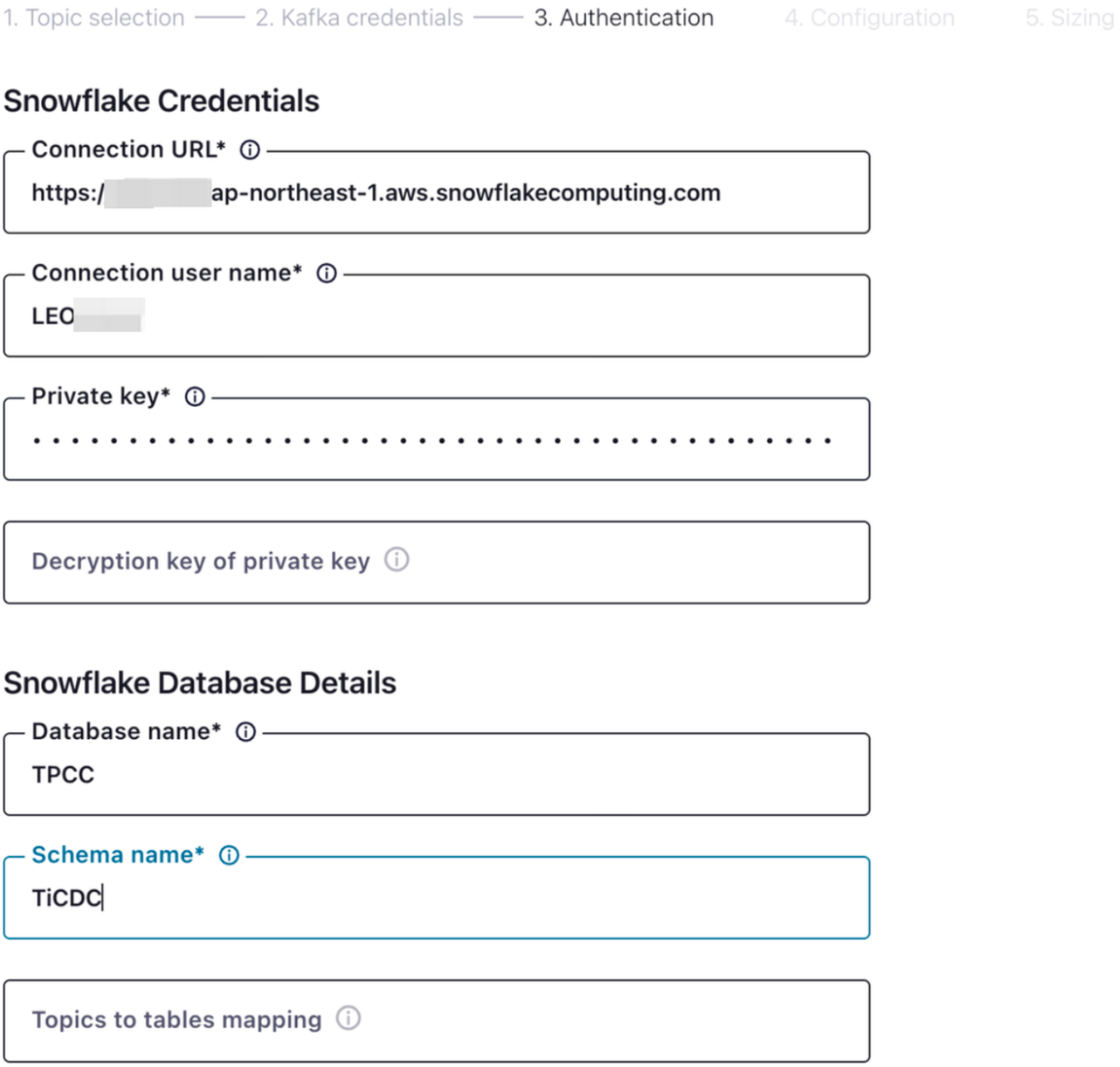


图 30: Credentials

4. 填写 Snowflake 连接认证信息，其中 Database name 和 Schema name 填写在上一步创建的 Database 和 Schema 名，随后进入下一页面：

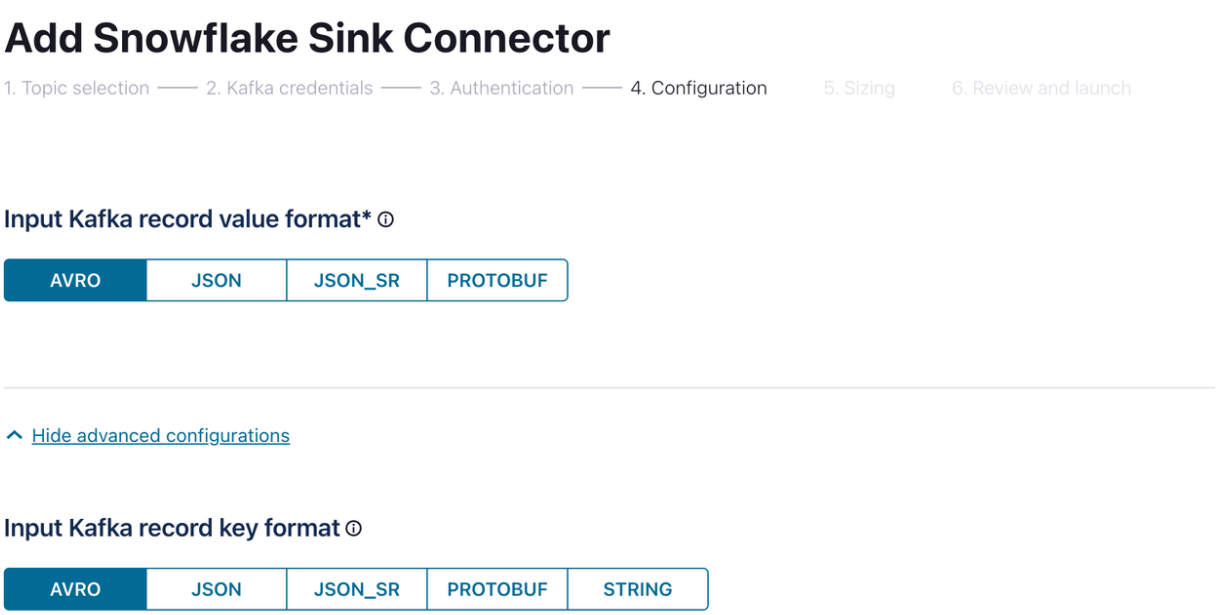


图 31: Configuration

5. 在 Configuration 页面中，`record value format` 和 `record key format` 都选择 AVRO，点击 Continue，直到 Connector 创建完成。等待 Connector 状态变为 `RUNNING`，这个过程可能持续数分钟。

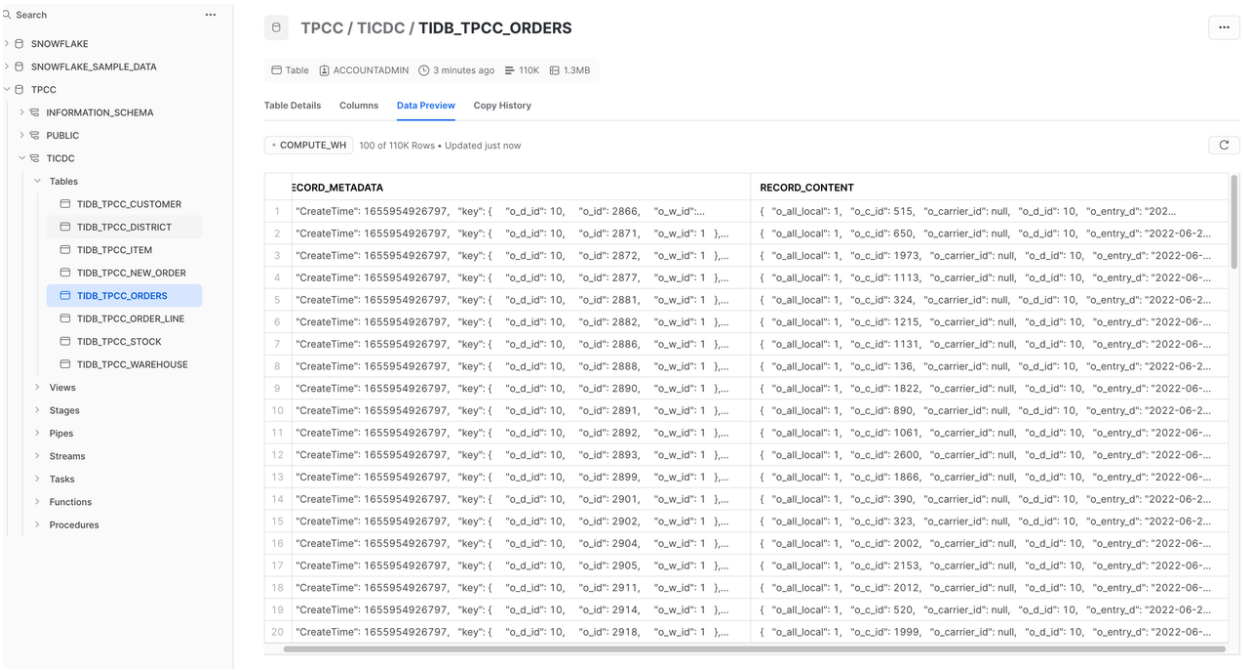


图 32: Data preview

6. 在 Snowflake 控制面板中，选择 Data > Database > TPCC > TiCDC，可以观察到 TiDB 中的增量数据实时同步到了 Snowflake，如上图。但 Snowflake 中的表结构和 TiDB 中的表结构不同，数据也以“追加”的方式插入 Snowflake 表。在大多数业务场景中，都希望 Snowflake 中的表数据是 TiDB 表的一个副本，而不是存储 TiDB 表的变更日志。该问题将在下一章节解决。

7.2.1.2.3 在 Snowflake 中创建 TiDB 表对应的数据副本

在上一章节，TiDB 的增量变更日志已经被同步到 Snowflake 中，本章节将介绍如何借助 Snowflake 的 TASK 和 STREAM 功能，将实时写入 Snowflake 的数据变更日志根据 INSERT、UPDATE 和 DELETE 等事件类型分别处理，写入一个与上游 TiDB 结构相同的表中，从而在 Snowflake 中创建一个数据副本。下面以 ITEM 表为例。

ITEM 表结构为：

```
CREATE TABLE `item` (
  `i_id` int(11) NOT NULL,
  `i_im_id` int(11) DEFAULT NULL,
  `i_name` varchar(24) DEFAULT NULL,
  `i_price` decimal(5,2) DEFAULT NULL,
  `i_data` varchar(50) DEFAULT NULL,
  PRIMARY KEY (`i_id`)
);
```

Snowflake 中存在一张名为 TIDB_TEST_ITEM 的表，这张表是 Confluent Snowflake Sink Connector 自动创建的，表结构如下：

```
create or replace TABLE TIDB_TEST_ITEM (
        RECORD_METADATA VARIANT,
        RECORD_CONTENT VARIANT
);
```

1. 根据 TiDB 中的表结构，在 Snowflake 中创建结构相同的表：

    ```
    create or replace table TEST_ITEM (
        i_id INTEGER primary key,
        i_data VARCHAR,
        i_im_id INTEGER,
        i_name VARCHAR,
        i_price DECIMAL(36,2)
    );
    ```

2. 为 TIDB_TEST_ITEM 创建一个 STREAM，将 append_only 设为 true，表示仅接收 INSERT 事件。创建的 STREAM 可以实时捕获 TIDB_TEST_ITEM 的 INSERT 事件，也就是说，当 TiDB 中 ITEM 有新的变更日志时，变更日志将会被插入到 TIDB_TEST_ITEM 表，然后被 STREAM 捕获。

    ```
    create or replace stream TEST_ITEM_STREAM on table TIDB_TEST_ITEM append_only=true;
    ```

3. 处理 STREAM 中的数据，根据不同的事件类型，在 TEST_ITEM 表中插入、更新或删除 STREAM 数据。

```
--将数据合并到 TEST_ITEM 表
merge into TEST_ITEM n
  using
      -- 查询 TEST_ITEM_STREAM
      (SELECT RECORD_METADATA:key as k, RECORD_CONTENT:val as v from TEST_ITEM_STREAM) stm
      -- 以 i_id 相等为条件将流和表做匹配
      on k:i_id = n.i_id
  -- 如果 TEST_ITEM 表中存在匹配 i_id 的记录，并且 v 为空，则删除这条记录
  when matched and IS_NULL_VALUE(v) = true then
      delete

  -- 如果 TEST_ITEM 表中存在匹配 i_id 的记录，并且 v 不为空，则更新这条记录
  when matched and IS_NULL_VALUE(v) = false then
      update set n.i_data = v:i_data, n.i_im_id = v:i_im_id, n.i_name = v:i_name, n.i_price
          ↪ = v:i_price

  -- 如果 TEST_ITEM 表中不存在匹配 i_id 的记录，则插入这条记录
  when not matched then
      insert
          (i_data, i_id, i_im_id, i_name, i_price)
      values
          (v:i_data, v:i_id, v:i_im_id, v:i_name, v:i_price)
;
```

在上面的语句中，我们使用了 Snowflake 的 MERGE INTO 语句，这个语句可以根据条件将流和表做匹配，然后根据不同的匹配结果，执行不同的操作，比如删除、更新或插入。在这个例子中，我们使用了三个 WHEN 子句，分别对应了三种情况：

- 当流和表匹配时，且流中的数据为空，则删除表中的记录
- 当流和表匹配时，且流中的数据不为空，则更新表中的记录
- 当流和表不匹配时，则插入流中的数据

4. 周期性执行第三步中的语句，以保证数据的实时性。可通过 Snowflake 的 SCHEDULED TASK 来实现：

```
-- 创建一个 TASK，周期性执行 MERGE INTO 语句
create or replace task STREAM_TO_ITEM
    warehouse = test
    -- 每分钟执行一次
    schedule = '1 minute'
when
    -- 当 TEST_ITEM_STREAM 中无数据时跳过
    system$stream_has_data('TEST_ITEM_STREAM')
as
-- 将数据合并到 TEST_ITEM 表，和上文中的 merge into 语句相同
merge into TEST_ITEM n
  using
```

```
    (select RECORD_METADATA:key as k, RECORD_CONTENT:val as v from TEST_ITEM_STREAM) stm
    on k:i_id = n.i_id
when matched and IS_NULL_VALUE(v) = true then
    delete
when matched and IS_NULL_VALUE(v) = false then
    update set n.i_data = v:i_data, n.i_im_id = v:i_im_id, n.i_name = v:i_name, n.i_price
        ↪ = v:i_price
when not matched then
    insert
        (i_data, i_id, i_im_id, i_name, i_price)
    values
        (v:i_data, v:i_id, v:i_im_id, v:i_name, v:i_price)
;
```

至此，你就建立了一条具备一定 ETL 能力的数据通路，使得 TiDB 的增量数据变更日志能够被输出到 Snowflake，并且维护一个 TiDB 表的数据副本，实现在 Snowflake 中使用 TiDB 表的数据。最后一步操作是定期清理 TIDB_TEST_ITEM 表中的无用数据：

```
-- 每两小时清空表 TIDB_TEST_ITEM
create or replace task TRUNCATE_TIDB_TEST_ITEM
    warehouse = test
    schedule = '120 minute'
when
    system$stream_has_data('TIDB_TEST_ITEM')
as
    TRUNCATE table TIDB_TEST_ITEM;
```

7.2.1.3 与 ksqlDB 进行数据集成

ksqlDB 是一种面向流式数据处理的数据库。你可以直接在 Confluent Cloud 上创建 ksqlDB 集群，并且直接读取 TiCDC 输出到 Confluent 的增量数据。

1. 在 Confluent 集群控制面板中选择 ksqlDB，按照引导创建 ksqlDB 集群。

   等待集群状态为 Running 后，进入下一步操作，这个过程可能持续数分钟。

2. 在 ksqlDB Editor 中执行如下命令，创建一个用于读取 `tidb_tpcc_orders` Topic 的 STREAM。

   ```
   CREATE STREAM orders (o_id INTEGER, o_d_id INTEGER, o_w_id INTEGER, o_c_id INTEGER,
       ↪ o_entry_d STRING, o_carrier_id INTEGER, o_ol_cnt INTEGER, o_all_local INTEGER) WITH
       ↪ (kafka_topic='tidb_tpcc_orders', partitions=3, value_format='AVRO');
   ```

3. 执行如下命令查询 orders STREAM 数据：

   ```
   SELECT * FROM ORDERS EMIT CHANGES;
   ```

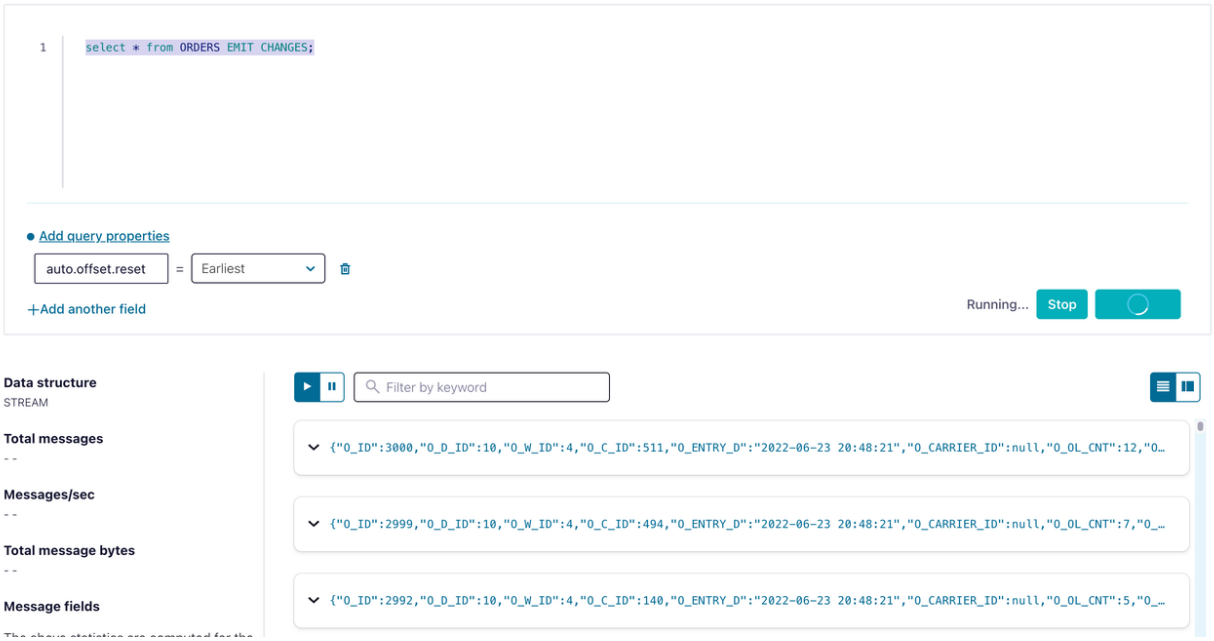


图 33: Select from orders

可以观察到 TiDB 中的增量数据实时同步到了 ksqlDB，如上图。至此，就完成了 TiDB 与 ksqlDB 的数据集成。

7.2.1.4 与 SQL Server 进行数据集成

SQL Server 是 Microsoft 推出的关系型数据库软件。借助 Confluent 的能力，你只需要创建 SQL Server Sink Connector，就可以将 TiDB 的增量数据输出到 SQL Server。

1. 连接 SQL Server 服务器，创建名为 `tpcc` 的数据库：

```
[ec2-user@ip-172-1-1-1 bin]$ sqlcmd -S 10.61.43.14,1433 -U admin
Password:
1> create database tpcc
2> go
1> select name from master.dbo.sysdatabases
2> go
name
--------------------------------------------------------------------------------
master
tempdb
model
msdb
rdsadmin
tpcc

(6 rows affected)
```

2. 在 Confluent 集群控制面板中，选择 Data integration > Connectors > Microsoft SQL Server Sink，进入如下页面：

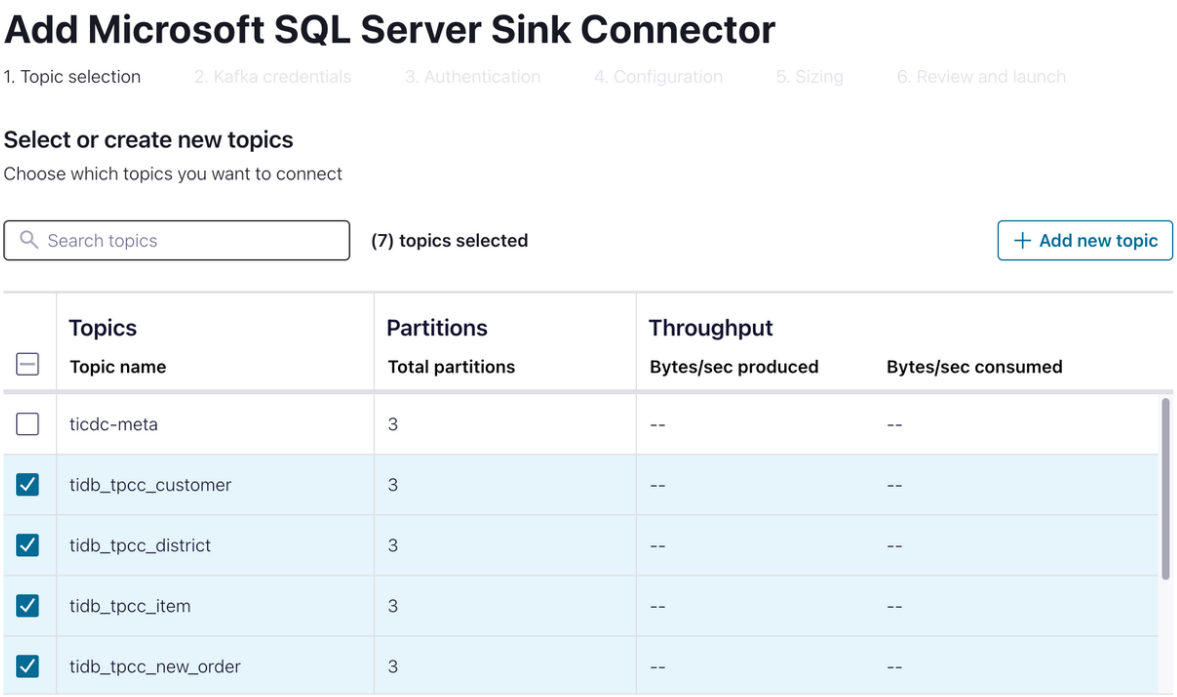


图 34: Topic selection

3. 选择需要同步到 SQL Server 的 Topic 后，进入下一页面：

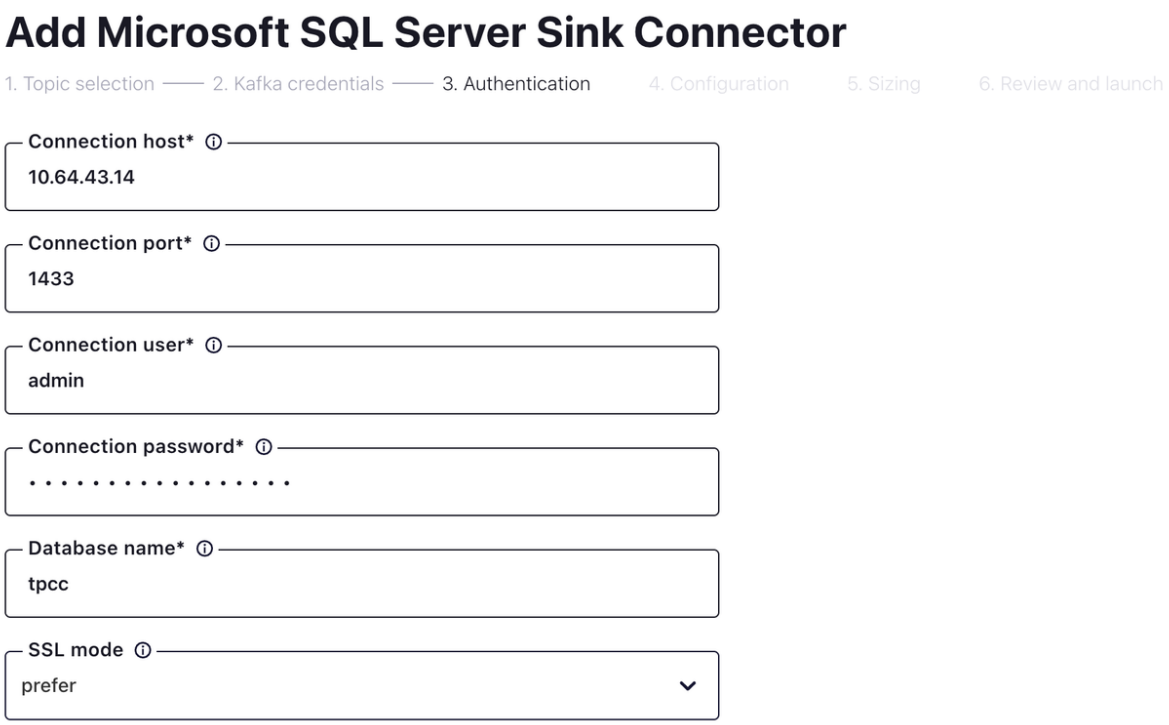


图 35: Authentication

4. 在填写 SQL Server 的连接和认证信息后，进入下一页面。
5. 在 Configuration 界面，按下表进行配置：

字段	取值
Input Kafka record value format	AVRO
Insert mode	UPSERT
Auto create table	true
Auto add columns	true
PK mode	record_key
Input Kafka record key format	AVRO
Delete on null	true

6. 配置完成后，选择 Continue，等待 Connector 状态变为 RUNNING，这个过程可能持续数分钟。

Run Cancel | Disconnect Change Connection | tpcc

```
SELECT TOP (1000) [o_id]
      ,[o_d_id]
      ,[o_w_id]
      ,[o_c_id]
      ,[o_entry_d]
      ,[o_carrier_id]
      ,[o_ol_cnt]
      ,[o_all_local]
  FROM [tpcc].[dbo].[tidb_tpcc_orders]
```

Results Messages

	o_id	o_d_id	o_w_id	o_c_id	o_entry_d	o_carrier_id	o_ol_cnt	o_all_local
1	1	1	1	1860	2022-06-22 22:08:06	3	6	1
2	1	3	1	884	2022-06-22 22:08:06	10	5	1
3	2	1	1	1247	2022-06-22 22:08:06	7	10	1
4	2	3	1	752	2022-06-22 22:08:06	2	12	1
5	3	1	1	902	2022-06-22 22:08:06	5	11	1
6	3	2	1	1250	2022-06-22 22:08:06	8	7	1
7	4	1	1	1468	2022-06-22 22:08:06	7	6	1
8	4	2	1	2246	2022-06-22 22:08:06	3	9	1
9	4	3	1	358	2022-06-22 22:08:06	4	13	1
10	5	1	1	639	2022-06-22 22:08:06	9	10	1

图 36: Results

7. 连接 SQL Server。观察 TiDB 中的增量数据实时同步到了 SQL Server，如上图。至此，就完成了 TiDB 与 SQL Server 的数据集成。

7.2.2 与 Apache Kafka 和 Apache Flink 进行数据集成

本文档介绍如何使用TiCDC 将 TiDB 的数据同步到 Apache Kafka。主要包含以下内容：

1. 快速搭建 TiCDC 集群、 Kafka 集群和 Flink 集群
2. 创建 changefeed，将 TiDB 增量数据输出至 Kafka
3. 使用 go-tpc 写入数据到上游 TiDB
4. 使用 Kafka console consumer 观察数据被写入到指定的 Topic
5. （可选）配置 Flink 集群消费 Kafka 内数据

上述过程将会基于实验环境进行。你也可以参考上述执行步骤，搭建生产级别的集群。

7.2.2.1 第 1 步：搭建环境

1. 部署包含 TiCDC 的 TiDB 集群。

   在实验或测试环境中，可以使用 TiUP Playground 功能，快速部署 TiCDC，命令如下：

```
tiup playground --host 0.0.0.0 --db 1 --pd 1 --kv 1 --tiflash 0 --ticdc 1
# 查看集群状态
tiup status
```

如果尚未安装 TiUP，可以参考安装 TiUP。在生产环境下，可以参考TiUP 安装部署 TiCDC 集群，完成 TiCDC 集群部署工作。

2. 部署 Kafka 集群。

    - 实验环境，可以参考 Apache Kakfa Quickstart 启动 Kafka 集群。
    - 生产环境，可以参考 Running Kafka in Production 完成 Kafka 集群搭建。

3. （可选）部署 Flink 集群。

    - 实验环境，可以参考 Apache Flink First steps 启动 Flink 集群。
    - 生产环境，可以参考 Apache Kafka Deployment 部署 Flink 生产集群。

7.2.2.2 第 2 步：创建 Kafka changefeed

1. 创建 changefeed 配置文件。

    根据 Flink 的要求和规范，每张表的增量数据需要发送到独立的 Topic 中，并且每个事件需要按照主键值分发 Partition。因此，需要创建一个名为 changefeed.conf 的配置文件，填写如下内容：

```
[sink]
dispatchers = [
{matcher = ['*.*'], topic = "tidb_{schema}_{table}", partition="index-value"},
]
```

    关于配置文件中 dispatchers 的详细解释，参考自定义 Kafka Sink 的 Topic 和 Partition 的分发规则。

2. 创建一个 changefeed，将增量数据输出到 Kafka：

```
tiup ctl:v<CLUSTER_VERSION> cdc changefeed create --server="http://127.0.0.1:8300" --sink-
    ↪ uri="kafka://127.0.0.1:9092/kafka-topic-name?protocol=canal-json" --changefeed-id="
    ↪ kafka-changefeed" --config="changefeed.conf"
```

    - 如果命令执行成功，将会返回被创建的 changefeed 的相关信息，包含被创建的 changefeed 的 ID 以及相关信息，内容如下：

```
Create changefeed successfully!
ID: kafka-changefeed
Info: {... changfeed info json struct ...}
```

    - 如果命令长时间没有返回，你需要检查当前执行命令所在服务器到 sink-uri 中指定的 Kafka 机器的网络可达性，保证二者之间的网络连接正常。

生产环境下 Kafka 集群通常有多个 broker 节点，你可以在 sink-uri 中配置多个 broker 的访问地址，这有助于提升 changefeed 到 Kafka 集群访问的稳定性，当部分被配置的 Kafka 节点故障的时候，changefeed 依旧可以正常工作。假设 Kafka 集群中有 3 个 broker 节点，地址分别为 127.0.0.1:9092 / 127.0.0.2:9092 / 127.0.0.3:9092，可以参考如下 sink-uri 创建 changefeed：

```
tiup ctl:v<CLUSTER_VERSION> cdc changefeed create --server="http://127.0.0.1:8300" --sink-
    ↪ uri="kafka://127.0.0.1:9092,127.0.0.2:9092,127.0.0.3:9092/kafka-topic-name?protocol=
    ↪ canal-json&partition-num=3&replication-factor=1&max-message-bytes=1048576" --config
    ↪ ="changefeed.conf"
```

3. Changefeed 创建成功后，执行如下命令，查看 changefeed 的状态：

```
tiup ctl:v<CLUSTER_VERSION> cdc changefeed list --server="http://127.0.0.1:8300"
```

可以参考管理 Changefeed，对 changefeed 状态进行管理。

7.2.2.3 第 3 步：写入数据以产生变更日志

完成以上步骤后，TiCDC 会将上游 TiDB 的增量数据变更日志发送到 Kafka，下面对 TiDB 写入数据，以产生增量数据变更日志。

1. 模拟业务负载。

   在测试实验环境下，可以使用 go-tpc 向上游 TiDB 集群写入数据，以让 TiDB 产生事件变更数据。如下命令，首先在上游 TiDB 创建名为 tpcc 的数据库，然后使用 TiUP bench 写入数据到这个数据库中。

   ```
   tiup bench tpcc -H 127.0.0.1 -P 4000 -D tpcc --warehouses 4 prepare
   tiup bench tpcc -H 127.0.0.1 -P 4000 -D tpcc --warehouses 4 run --time 300s
   ```

   关于 go-tpc 的更多详细内容，可以参考如何对 TiDB 进行 TPC-C 测试。

2. 消费 Kafka Topic 中的数据。

   changefeed 正常运行时，会向 Kafka Topic 写入数据，你可以通过由 Kafka 提供的 kafka-console-consumer.sh，观测到数据成功被写入到 Kafka Topic 中：

   ```
   ./bin/kafka-console-consumer.sh --bootstrap-server 127.0.0.1:9092 --from-beginning --topic `
       ↪ ${topic-name}`
   ```

至此，TiDB 的增量数据变更日志就实时地复制到了 Kafka。下一步，你可以使用 Flink 消费 Kafka 数据。当然，你也可以自行开发适用于业务场景的 Kafka 消费端。

7.2.2.4 第 4 步：配置 Flink 消费 Kafka 数据（可选）

1. 安装 Flink Kafka Connector。

   在 Flink 生态中，Flink Kafka Connector 用于消费 Kafka 中的数据并输出到 Flink 中。Flink Kafka Connector 并不是内建的，因此在 Flink 安装完毕后，还需要将 Flink Kafka Connector 及其依赖项添加到 Flink 安装目录中。下载下列 jar 文件至 Flink 安装目录下的 lib 目录中，如果你已经运行了 Flink 集群，请重启集群以加载新的插件。

- flink-connector-kafka-1.15.0.jar
- flink-sql-connector-kafka-1.15.0.jar
- kafka-clients-3.2.0.jar

2. 创建一个表。

   你可以在 Flink 的安装目录执行如下命令，启动 Flink SQL 交互式客户端：

```
[root@flink flink-1.15.0]# ./bin/sql-client.sh
```

   随后，执行如下语句创建一个名为 `tpcc_orders` 的表：

```
CREATE TABLE tpcc_orders (
    o_id INTEGER,
    o_d_id INTEGER,
    o_w_id INTEGER,
    o_c_id INTEGER,
    o_entry_d STRING,
    o_carrier_id INTEGER,
    o_ol_cnt INTEGER,
    o_all_local INTEGER
) WITH (
'connector' = 'kafka',
'topic' = 'tidb_tpcc_orders',
'properties.bootstrap.servers' = '127.0.0.1:9092',
'properties.group.id' = 'testGroup',
'format' = 'canal-json',
'scan.startup.mode' = 'earliest-offset',
'properties.auto.offset.reset' = 'earliest'
)
```

   请将 `topic` 和 `properties.bootstrap.servers` 参数替换为环境中的实际值。

3. 查询表内容。

   执行如下命令，查询 `tpcc_orders` 表中的数据：

```
SELECT * FROM tpcc_orders;
```

   执行成功后，可以观察到有数据输出，如下图：

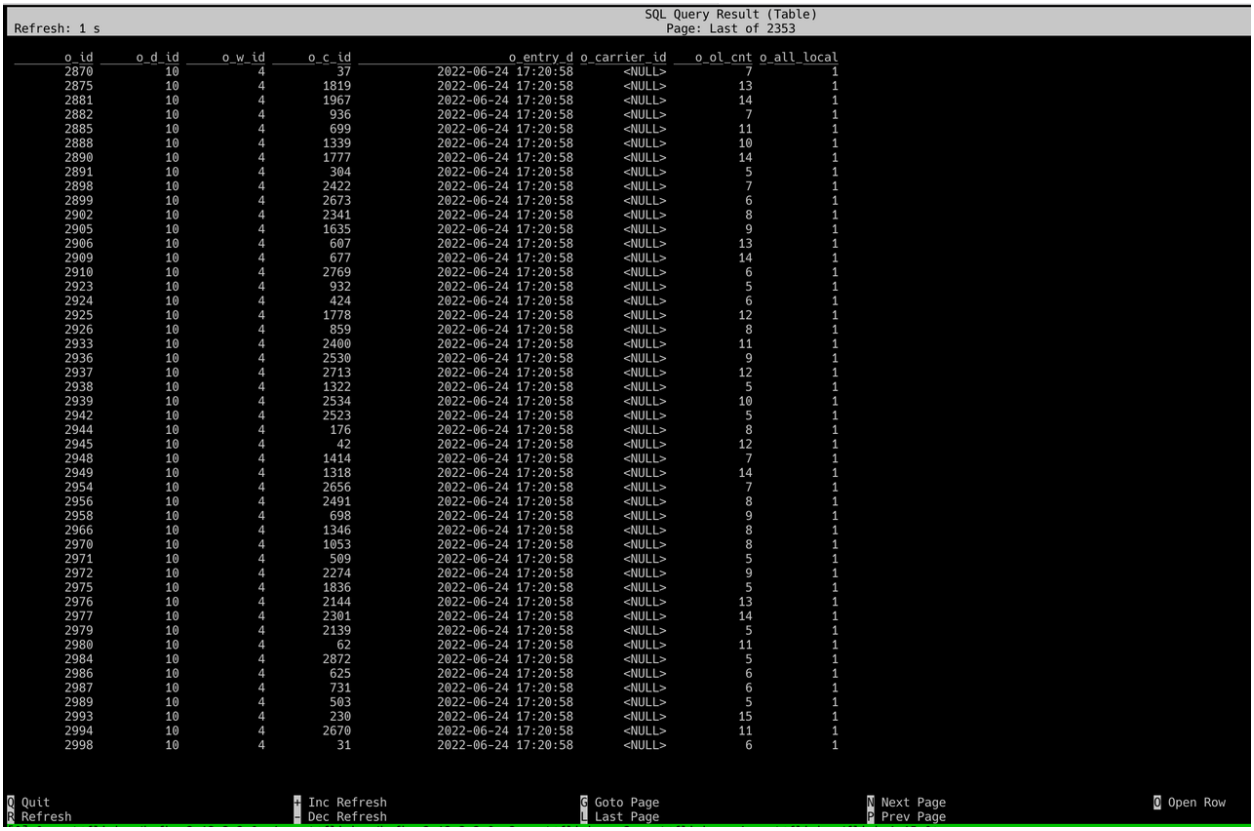


图 37: SQL query result

至此，就完成了 TiDB 与 Flink 的数据集成。

8 运维操作

8.1 升级 TiDB 版本

8.1.1 使用 TiUP 升级 TiDB

本文档适用于以下升级路径：

- 使用 TiUP 从 TiDB 4.0 版本升级至 TiDB 7.0。
- 使用 TiUP 从 TiDB 5.0-5.4 版本升级至 TiDB 7.0。
- 使用 TiUP 从 TiDB 6.0 版本升级至 TiDB 7.0。
- 使用 TiUP 从 TiDB 6.1 版本升级至 TiDB 7.0。
- 使用 TiUP 从 TiDB 6.2 版本升级至 TiDB 7.0。
- 使用 TiUP 从 TiDB 6.3 版本升级至 TiDB 7.0。
- 使用 TiUP 从 TiDB 6.4 版本升级至 TiDB 7.0。
- 使用 TiUP 从 TiDB 6.5 版本升级至 TiDB 7.0。
- 使用 TiUP 从 TiDB 6.6 版本升级至 TiDB 7.0。

警告：

- 不支持将 TiFlash 组件从 5.3 之前的老版本在线升级至 5.3 及之后的版本，只能采用停机升级。如果集群中其他组件（如 tidb，tikv）不能停机升级，参考不停机升级中的注意事项。
- 在升级 TiDB 集群的过程中，请勿执行 DDL 语句，否则可能会出现行为未定义的问题。
- 集群中有 DDL 语句正在被执行时（通常为 ADD INDEX 和列类型变更等耗时较久的 DDL 语句），请勿进行升级操作。在升级前，建议使用ADMIN SHOW DDL 命令查看集群中是否有正在进行的 DDL Job。如需升级，请等待 DDL 执行完成或使用ADMIN CANCEL DDL 命令取消该 DDL Job 后再进行升级。

注意：

- 如果原集群是 3.0 或 3.1 或更早的版本，不支持直接升级到 7.0.0 及后续修订版本。你需要先从早期版本升级到 4.0 后，再从 4.0 升级到 7.0.0 及后续修订版本。
- 配置参数server-version 的值会被 TiDB 节点用于验证当前 TiDB 的版本。因此在进行 TiDB 集群升级前，请将 server-version 的值设置为空或者当前 TiDB 真实的版本值，避免出现非预期行为。

8.1.1.1 1. 升级兼容性说明

- TiDB 目前暂不支持版本降级或升级后回退。
- 使用 TiDB Ansible 管理的 4.0 版本集群，需要先按照 4.0 版本文档的说明将集群导入到 TiUP (tiup cluster) 管理后，再按本文档说明升级到 7.0.0 版本。
- 若要将 3.0 之前的版本升级至 7.0.0 版本：
  1. 首先通过 TiDB Ansible 升级到 3.0 版本。
  2. 然后按照 4.0 版本文档的说明，使用 TiUP (tiup cluster) 将 TiDB Ansible 配置导入。
  3. 将集群升级至 4.0 版本。
  4. 按本文档说明将集群升级到 7.0.0 版本。
- 支持 TiDB Binlog，TiCDC，TiFlash 等组件版本的升级。
- 将 v6.3.0 之前的 TiFlash 升级至 v6.3.0 及之后的版本时，需要特别注意：在 Linux AMD64 架构的硬件平台部署 TiFlash 时，CPU 必须支持 AVX2 指令集。而在 Linux ARM64 架构的硬件平台部署 TiFlash 时，CPU 必须支持 ARMv8 架构。具体请参考6.3.0 版本 Release Notes 中的描述。
- 具体不同版本的兼容性说明，请查看各个版本的Release Note。请根据各个版本的 Release Note 的兼容性更改调整集群的配置。
- 升级 v5.3 之前版本的集群到 v5.3 及后续版本时，默认部署的 Prometheus 会从 v2.8.1 升级到 v2.27.1，v2.27.1 提供更多的功能并解决了安全风险。Prometheus v2.27.1 相对于 v2.8.1 存在 Alert 时间格式变化，详情见 Prometheus commit。

8.1.1.2 2. 升级前准备

本部分介绍实际开始升级前需要进行的更新 TiUP 和 TiUP Cluster 组件版本等准备工作。

8.1.1.2.1 2.1 查阅兼容性变更

查阅 TiDB v7.0.0 release notes 中的兼容性变更。如果有任何变更影响到了你的升级，请采取相应的措施。

8.1.1.2.2 2.2 升级 TiUP 或更新 TiUP 离线镜像

升级 TiUP 和 TiUP Cluster

> **注意：**
>
> 如果原集群中控机不能访问 `https://tiup-mirrors.pingcap.com` 地址，可跳过本步骤，然后更新 TiUP 离线镜像。

1. 先升级 TiUP 版本（建议 `tiup` 版本不低于 `1.11.3`）：

```
tiup update --self
tiup --version
```

2. 再升级 TiUP Cluster 版本（建议 `tiup cluster` 版本不低于 `1.11.3`）：

```
tiup update cluster
tiup cluster --version
```

更新 TiUP 离线镜像

> **注意：**
>
> 如果原集群不是通过离线部署方式部署的，可忽略此步骤。

可以参考使用 TiUP 部署 TiDB 集群的步骤下载部署新版本的 TiUP 离线镜像，上传到中控机。在执行 `local_install.sh` 后，TiUP 会完成覆盖升级。

```
tar xzvf tidb-community-server-${version}-linux-amd64.tar.gz
sh tidb-community-server-${version}-linux-amd64/local_install.sh
source /home/tidb/.bash_profile
```

建议：

关于 TiDB-community-server 软件包和 TiDB-community-toolkit 软件包的内容物，请查阅TiDB离线包。

覆盖升级完成后，需将 server 和 toolkit 两个离线镜像合并，执行以下命令合并离线组件到 server 目录下。

```
tar xf tidb-community-toolkit-${version}-linux-amd64.tar.gz
ls -ld tidb-community-server-${version}-linux-amd64 tidb-community-toolkit-${version}-linux-amd64
cd tidb-community-server-${version}-linux-amd64/
cp -rp keys ~/.tiup/
tiup mirror merge ../tidb-community-toolkit-${version}-linux-amd64
```

离线镜像合并后，执行下列命令升级 Cluster 组件：

```
tiup update cluster
```

此时离线镜像已经更新成功。如果覆盖后发现 TiUP 运行报错，可能是 manifest 未更新导致，可尝试 `rm -rf ~/.tiup/manifests/*` 后再使用。

8.1.1.2.3 2.3 编辑 TiUP Cluster 拓扑配置文件

注意：

以下情况可跳过此步骤：

- 原集群没有修改过配置参数，或通过 tiup cluster 修改过参数但不需要调整。
- 升级后对未修改过的配置项希望使用 7.0.0 默认参数。

1. 进入拓扑文件的 vi 编辑模式：

   ```
   tiup cluster edit-config <cluster-name>
   ```

2. 参考 topology 配置模板的格式，将希望修改的参数填到拓扑文件的 server_configs 下面。

修改完成后 :wq 保存并退出编辑模式，输入 Y 确认变更。

注意：

升级到 7.0.0 版本前，请确认已在 4.0 修改的参数在 7.0.0 版本中是兼容的，可参考TiKV 配置文件描述。

8.1.1.2.4 2.4 检查当前集群的健康状况

为避免升级过程中出现未定义行为或其他故障，建议在升级前对集群当前的 region 健康状态进行检查，此操作可通过 check 子命令完成。

```
tiup cluster check <cluster-name> --cluster
```

执行结束后，最后会输出 region status 检查结果。如果结果为 “All regions are healthy”，则说明当前集群中所有 region 均为健康状态，可以继续执行升级；如果结果为 “Regions are not fully healthy: m miss-peer, n pending-peer” 并提示 “Please fix unhealthy regions before other operations.”，则说明当前集群中有 region 处在异常状态，应先排除相应异常状态，并再次检查结果为 “All regions are healthy” 后再继续升级。

8.1.1.2.5 2.5 检查当前集群的 DDL 和 Backup 情况

为避免升级过程中出现未定义行为或其他故障，建议检查以下指标后再进行升级操作。

- 集群 DDL 情况：建议使用`ADMIN SHOW DDL` 命令查看集群中是否有正在进行的 DDL Job。如需升级，请等待 DDL 执行完成或使用`ADMIN CANCEL DDL` 命令取消该 DDL Job 后再进行升级。
- 集群 Backup 情况：建议使用`SHOW [BACKUPS|RESTORES]` 命令查看集群中是否有正在进行的 Backup 或者 Restore 任务。如需升级，请等待 Backup 执行完成后，得到一个有效的备份后再执行升级。

8.1.1.3 3. 升级 TiDB 集群

本部分介绍如何滚动升级 TiDB 集群以及如何进行升级后的验证。

8.1.1.3.1 3.1 将集群升级到指定版本

升级的方式有两种：不停机升级和停机升级。TiUP Cluster 默认的升级 TiDB 集群的方式是不停机升级，即升级过程中集群仍然可以对外提供服务。升级时会对各节点逐个迁移 leader 后再升级和重启，因此对于大规模集群需要较长时间才能完成整个升级操作。如果业务有维护窗口可供数据库停机维护，则可以使用停机升级的方式快速进行升级操作。

不停机升级

```
tiup cluster upgrade <cluster-name> <version>
```

以升级到 7.0.0 版本为例：

```
tiup cluster upgrade <cluster-name> v7.0.0
```

> 注意：
>
> - 滚动升级会逐个升级所有的组件。升级 TiKV 期间，会逐个将 TiKV 上的所有 leader 切走再停止该 TiKV 实例。默认超时时间为 5 分钟（300 秒），超时后会直接停止该实例。
> - 使用 `--force` 参数可以在不驱逐 leader 的前提下快速升级集群至新版本，但是该方式会忽略所有升级中的错误，在升级失败后得不到有效提示，请谨慎使用。

- 如果希望保持性能稳定，则需要保证 TiKV 上的所有 leader 驱逐完成后再停止该 TiKV 实例，可以指定 `--transfer-timeout` 为一个更大的值，如 `--transfer-timeout 3600`，单位为秒。
- 若想将 TiFlash 从 5.3 之前的版本升级到 5.3 及之后的版本，必须进行 TiFlash 的停机升级。参考如下步骤，可以在确保其他组件正常运行的情况下升级 TiFlash：

1. 关闭 TiFlash 实例：`tiup cluster stop <cluster-name> -R tiflash`
2. 使用 `--offline` 参数在不重启（只更新文件）的情况下升级集群：`tiup cluster upgrade <cluster-name> <version> --offline`，例如 `tiup cluster upgrade <cluster-name> v6.3.0 --offline`
3. reload 整个集群：`tiup cluster reload <cluster-name>`。此时，TiFlash 也会正常启动，无需额外操作。

- 在对使用 TiDB Binlog 的集群进行滚动升级过程中，请避免新创建聚簇索引表。

停机升级

在停机升级前，首先需要将整个集群关停。

```
tiup cluster stop <cluster-name>
```

之后通过 `upgrade` 命令添加 `--offline` 参数来进行停机升级，其中 `<cluster-name>` 为集群名，`<version>` 为升级的目标版本，例如 `v6.5.0`。

```
tiup cluster upgrade <cluster-name> <version> --offline
```

升级完成后集群不会自动启动，需要使用 `start` 命令来启动集群。

```
tiup cluster start <cluster-name>
```

8.1.1.3.2 3.2 升级后验证

执行 `display` 命令来查看最新的集群版本 `TiDB Version`：

```
tiup cluster display <cluster-name>
```

```
Cluster type:       tidb
Cluster name:       <cluster-name>
Cluster version:    v7.0.0
```

8.1.1.4 4. 升级 FAQ

本部分介绍使用 TiUP 升级 TiDB 集群遇到的常见问题。

8.1.1.4.1 4.1 升级时报错中断，处理完报错后，如何继续升级

重新执行 `tiup cluster upgrade` 命令进行升级，升级操作会重启之前已经升级完成的节点。如果不希望重启已经升级过的节点，可以使用 `replay` 子命令来重试操作，具体方法如下：

1. 使用 `tiup cluster audit` 命令查看操作记录：

```
tiup cluster audit
```

在其中找到失败的升级操作记录，并记下该操作记录的 ID，下一步中将使用 `<audit-id>` 表示操作记录 ID 的值。

2. 使用 `tiup cluster replay <audit-id>` 命令重试对应操作：

```
tiup cluster replay <audit-id>
```

8.1.1.4.2 4.2 升级过程中 evict leader 等待时间过长，如何跳过该步骤快速升级

可以指定 `--force`，升级时会跳过 `PD transfer leader` 和 `TiKV evict leader` 过程，直接重启并升级版本，对线上运行的集群性能影响较大。命令如下，其中 `<version>` 为升级的目标版本，例如 `v6.5.0`：

```
tiup cluster upgrade <cluster-name> <version> --force
```

8.1.1.4.3 4.3 升级完成后，如何更新 pd-ctl 等周边工具版本

可通过 TiUP 安装对应版本的 `ctl` 组件来更新相关工具版本：

```
tiup install ctl:v7.0.0
```

8.1.2 使用 TiDB Operator

8.1.3 TiFlash v6.2 升级帮助

本文介绍从 TiFlash 低版本升级至 v6.2 时功能模块的变化，以及推荐的应对方法。

如需了解标准升级流程，请参考如下文档：

- 使用 TiUP 升级 TiDB
- 使用 TiDB Operator 升级 TiDB

> 注意：
>
> - v6.2.0 新增了一项名为FastScan 的实验功能，该功能在 v7.0.0 GA。FastScan 在牺牲强一致性保证的前提下可以大幅提升扫表性能。
> - 不推荐跨主干版本升级包含 TiFlash 的 TiDB 集群，如从 v4.x 升级至 v6.x，请先升级至 v5.x，然后再升级至 v6.x。

- v4.x. 已接近产品周期尾声，请尽早升级到 v5.x 及以上版本。具体的版本周期请参考 TiDB 版本周期支持策略。
- v6.0 作为非 LTS 版本，不会推出后续的 bug 修复版，请尽量使用 v6.1 及之后的 LTS 版本。
- 若想将 TiFlash 从 v5.3.0 之前的版本升级到 v5.3.0 及之后的版本，必须进行 TiFlash 的停机升级。参考如下步骤，可以在确保其他组件正常运行的情况下升级 TiFlash：
  - 关闭 TiFlash 实例：`tiup cluster stop <cluster-name> -R tiflash`
  - 使用 `--offline` 参数在不重启（只更新文件）的情况下升级集群：`tiup cluster upgrade <cluster-name> <version> --offline`，例如 `tiup cluster upgrade <cluster-name> v5.3.0 --offline`
  - reload 整个集群：`tiup cluster reload <cluster-name>`。此时，TiFlash 也会正常启动，无需额外操作。

8.1.3.1 从 v5.x 或 v6.0 升级至 v6.1

从 v5.x 或 v6.0 升级至 v6.1 时，需要注意 TiFlash Proxy 和动态分区裁剪功能的变化。

8.1.3.1.1 TiFlash Proxy

TiFlash 在 v6.1.0 对 Proxy 做了升级（与 TiKV v6.0.0 对齐）。新的 Proxy 版本升级了 RocksDB 版本，在升级过程中会自动将数据格式转换为新版本。

正常升级时，不会有明显风险。如果特殊场景（如测试验证）需要降级，请注意，v6.1 降级到之前的任意版本时，会无法解析新版 RocksDB 配置，从而导致 TiFlash 重启失败。请做好升级验证工作，并尽可能准备应急方案。

测试环境及特殊回退需求下的对策

强制缩容 TiFlash 节点，并重新同步数据。操作步骤详见缩容 TiFlash 节点。

8.1.3.1.2 动态分区裁剪

如果你没有也不打算开启动态分区裁剪，可略过本部分。

- TiDB v6.1 全新安装：默认开启动态分区裁剪 (Dynamic Pruning)。
- TiDB v6.0 及之前版本：默认关闭动态分区裁剪。旧版本升级遵循已有设定，不会自动开启（相对的也不会关闭）此功能。

  升级完成之后，如果要启用动态分区裁剪特性，请确保 `tidb_partition_prune_mode` 的值为 `dynamic`，并手动更新分区表的全局统计信息。关于如何手动更新统计信息，参见动态裁剪模式。

8.1.3.2 从 v5.x 或 v6.0 升级至 v6.2

TiFlash 在 v6.2.0 将数据格式升级到 V3 版本，因此，从 v5.x 或 v6.0 升级至 v6.2 时，除了需要注意TiFlash Proxy 和动态分区裁剪的变化，还应注意 PageStorage 变更数据版本带来的影响，具体如下：

- 已有节点升级 v6.2 后，随着数据不断写入，旧版本的数据会逐步转换成新版本数据。

- 新旧版本的数据格式不能做到完全的转换，这会带来一定系统开销（通常不影响业务，但需要注意）。因此升级完成后，建议使用COMPACT 命令触发数据整理 (Compaction) 将相关表的数据转成新版本格式。操作步骤如下：

    1. 对每张有 TiFlash 副本（replica）的表执行如下命令：

    ```
    ALTER TABLE <table_name> COMPACT tiflash replica;
    ```

    2. 重启 TiFlash 节点。

你可以在 Grafana 监控查看是否还有表使用旧的数据版本：Tiflash summary > storage pool > Storage Pool Run Mode

- Only V2：使用 PageStorage V2 的表数量（包括分区数）
- Only V3：使用 PageStorage V3 的表数量（包括分区数）
- Mix Mode：从 V2 迁移到 V3 的表数量（包括分区数）

测试环境及特殊回退需求下的对策

强制缩容 TiFlash 节点，并重新同步数据。操作步骤详见缩容 TiFlash 节点。

8.1.3.3 从 v6.1 升级至 v6.2

从 v6.1 升级至 v6.2 时，需要注意 PageStorage 变更数据版本带来的影响。具体请参考从 v5.x 或 v6.0 升级至 v6.2 中关于 PageStorage 的描述。

8.2 扩缩容

8.2.1 使用 TiUP 扩容缩容 TiDB 集群

TiDB 集群可以在不中断线上服务的情况下进行扩容和缩容。

本文介绍如何使用 TiUP 扩容缩容集群中的 TiDB、TiKV、PD、TiCDC 或者 TiFlash 节点。如未安装 TiUP，可参考部署文档中的步骤。

你可以通过 `tiup cluster list` 查看当前的集群名称列表。

例如，集群原拓扑结构如下所示：

主机 IP	服务
10.0.1.3	TiDB + TiFlash
10.0.1.4	TiDB + PD
10.0.1.5	TiKV + Monitor
10.0.1.1	TiKV
10.0.1.2	TiKV

8.2.1.1 扩容 TiDB/PD/TiKV 节点

如果要添加一个 TiDB 节点，IP 地址为 10.0.1.5，可以按照如下步骤进行操作。

> **注意：**
>
> 添加 PD 节点和添加 TiDB 节点的步骤类似。添加 TiKV 节点前，建议预先根据集群的负载情况调整 PD 调度参数。

8.2.1.1.1 1. 编写扩容拓扑配置

> **注意：**
>
> - 默认情况下，可以不填写端口以及目录信息。但在单机多实例场景下，则需要分配不同的端口以及目录，如果有端口或目录冲突，会在部署或扩容时提醒。
> - 从 TiUP v1.0.0 开始，扩容配置会继承原集群配置的 global 部分。

在 scale-out.yml 文件添加扩容拓扑配置：

```
vi scale-out.yml
```

```
tidb_servers:
  - host: 10.0.1.5
    ssh_port: 22
    port: 4000
    status_port: 10080
    deploy_dir: /tidb-deploy/tidb-4000
    log_dir: /tidb-deploy/tidb-4000/log
```

TiKV 配置文件参考：

```
tikv_servers:
  - host: 10.0.1.5
    ssh_port: 22
    port: 20160
    status_port: 20180
    deploy_dir: /tidb-deploy/tikv-20160
    data_dir: /tidb-data/tikv-20160
    log_dir: /tidb-deploy/tikv-20160/log
```

PD 配置文件参考：

```
pd_servers:
```

```
- host: 10.0.1.5
  ssh_port: 22
  name: pd-1
  client_port: 2379
  peer_port: 2380
  deploy_dir: /tidb-deploy/pd-2379
  data_dir: /tidb-data/pd-2379
  log_dir: /tidb-deploy/pd-2379/log
```

可以使用 `tiup cluster edit-config <cluster-name>` 查看当前集群的配置信息，因为其中的 `global` 和 `server_configs` 参数配置默认会被 `scale-out.yml` 继承，因此也会在 `scale-out.yml` 中生效。

8.2.1.1.2 2. 执行扩容命令

执行 scale-out 命令前，先使用 check 及 `check --apply` 命令，检查和自动修复集群存在的潜在风险：

> **注意：**
>
> 针对 scale-out 命令的检查功能在 tiup cluster v1.9.3 及后续版本中支持，请操作前先升级 tiup cluster 版本。

（1）检查集群存在的潜在风险：

```
shell tiup cluster check <cluster-name> scale-out.yml --cluster --user root [-p] [-i /home/root/.
↪ ssh/gcp_rsa]
```

（2）自动修复集群存在的潜在风险：

```
shell tiup cluster check <cluster-name> scale-out.yml --cluster --apply --user root [-p] [-i /
↪ home/root/.ssh/gcp_rsa]
```

（3）执行 scale-out 命令扩容 TiDB 集群：

```
shell tiup cluster scale-out <cluster-name> scale-out.yml [-p] [-i /home/root/.ssh/gcp_rsa]
```

以上操作示例中：

- 扩容配置文件为 `scale-out.yml`。
- `--user root` 表示通过 root 用户登录到目标主机完成集群部署，该用户需要有 ssh 到目标机器的权限，并且在目标机器有 sudo 权限。也可以用其他有 ssh 和 sudo 权限的用户完成部署。
- [-i] 及 [-p] 为可选项，如果已经配置免密登录目标机，则不需填写。否则选择其一即可，[-i] 为可登录到目标机的 root 用户（或 –user 指定的其他用户）的私钥，也可使用 [-p] 交互式输入该用户的密码。

预期日志结尾输出 `Scaled cluster `<cluster-name>` out successfully` 信息，表示扩容操作成功。

8.2.1.1.3 3. 检查集群状态

```
tiup cluster display <cluster-name>
```

打开浏览器访问监控平台 http://10.0.1.5:3000，监控整个集群和新增节点的状态。

扩容后，集群拓扑结构如下所示：

主机 IP	服务
10.0.1.3	TiDB + TiFlash
10.0.1.4	TiDB + PD
10.0.1.5	TiDB + TiKV + Monitor
10.0.1.1	TiKV
10.0.1.2	TiKV

8.2.1.2 扩容 TiFlash 节点

如果要添加一个 TiFlash 节点，其 IP 地址为 10.0.1.4，可以按照如下步骤进行操作。

> 注意：
>
> 在原有 TiDB 集群上新增 TiFlash 组件需要注意：
>
> 1. 首先确认当前 TiDB 的版本支持 TiFlash，否则需要先升级 TiDB 集群至 v5.0 以上版本。
> 2. 执 行 `tiup ctl:v<CLUSTER_VERSION> pd -u http://<pd_ip>:<pd_port> config set enable-placement-rules true` 命令，以开启 PD 的 Placement Rules 功能。或通过pd-ctl 执行对应的命令。

8.2.1.2.1 1. 添加节点信息到 scale-out.yml 文件

编写 scale-out.yml 文件，添加该 TiFlash 节点信息（目前只支持 ip，不支持域名）：

```
tiflash_servers:
  - host: 10.0.1.4
```

8.2.1.2.2 2. 运行扩容命令

```
tiup cluster scale-out <cluster-name> scale-out.yml
```

> 注意：
>
> 此处假设当前执行命令的用户和新增的机器打通了互信，如果不满足已打通互信的条件，需要通过 -p 来输入新机器的密码，或通过 -i 指定私钥文件。

8.2.1.2.3 3. 查看集群状态

```
tiup cluster display <cluster-name>
```

打开浏览器访问监控平台 http://10.0.1.5:3000，监控整个集群和新增节点的状态。

扩容后，集群拓扑结构如下所示：

主机 IP	服务
10.0.1.3	TiDB + TiFlash
10.0.1.4	TiDB + PD + TiFlash
10.0.1.5	TiDB+ TiKV + Monitor
10.0.1.1	TiKV
10.0.1.2	TiKV

8.2.1.3 扩容 TiCDC 节点

如果要添加 TiCDC 节点，IP 地址为 10.0.1.3、10.0.1.4，可以按照如下步骤进行操作。

8.2.1.3.1 1. 添加节点信息到 scale-out.yml 文件

编写 scale-out.yml 文件：

```
cdc_servers:
  - host: 10.0.1.3
    gc-ttl: 86400
    data_dir: /tidb-data/cdc-8300
  - host: 10.0.1.4
    gc-ttl: 86400
    data_dir: /tidb-data/cdc-8300
```

8.2.1.3.2 2. 运行扩容命令

```
tiup cluster scale-out <cluster-name> scale-out.yml
```

> **注意：**
>
> 此处假设当前执行命令的用户和新增的机器打通了互信，如果不满足已打通互信的条件，需要通过 -p 来输入新机器的密码，或通过 -i 指定私钥文件。

8.2.1.3.3 3. 查看集群状态

```
tiup cluster display <cluster-name>
```

打开浏览器访问监控平台 http://10.0.1.5:3000，监控整个集群和新增节点的状态。

扩容后，集群拓扑结构如下所示：

主机 IP	服务
10.0.1.3	TiDB + TiFlash + TiCDC
10.0.1.4	TiDB + PD + TiFlash + TiCDC
10.0.1.5	TiDB+ TiKV + Monitor
10.0.1.1	TiKV
10.0.1.2	TiKV

8.2.1.4 缩容 TiDB/PD/TiKV 节点

如果要移除 IP 地址为 10.0.1.5 的一个 TiKV 节点，可以按照如下步骤进行操作。

> 注意：
>
> - 移除 TiDB、 PD 节点和移除 TiKV 节点的步骤类似。
> - 由于 TiKV、 TiFlash 和 TiDB Binlog 组件是异步下线的，且下线过程耗时较长，所以 TiUP 对 TiKV、 TiFlash 和 TiDB Binlog 组件做了特殊处理，详情参考下线特殊处理。

> 注意：
>
> TiKV 中的 PD Client 会缓存 PD 节点的列表。当前版本的 TiKV 有定期自动更新 PD 节点的机制，可以降低 TiKV 缓存的 PD 节点列表过旧这一问题出现的概率。但你应尽量避免在扩容新 PD 后直接一次性缩容所有扩容前就已经存在的 PD 节点。如果需要，请确保在下线所有之前存在的 PD 节点前将 PD 的 leader 切换至新扩容的 PD 节点。

8.2.1.4.1 1. 查看节点 ID 信息

```
tiup cluster display <cluster-name>
```

```
Starting /root/.tiup/components/cluster/v1.11.3/cluster display <cluster-name>

TiDB Cluster: <cluster-name>

TiDB Version: v7.0.0

ID       Role         Host    Ports                            Status  Data Dir        Deploy Dir
```

```
--        ----          ----       -----                              ------  --------
    ↪ ----------

10.0.1.3:8300  cdc           10.0.1.3    8300                             Up      data/cdc-8300
    ↪       deploy/cdc-8300

10.0.1.4:8300  cdc           10.0.1.4    8300                             Up      data/cdc-8300
    ↪       deploy/cdc-8300

10.0.1.4:2379  pd            10.0.1.4    2379/2380                         Healthy data/pd-2379
    ↪       deploy/pd-2379

10.0.1.1:20160 tikv          10.0.1.1    20160/20180                       Up      data/tikv-20160
    ↪      deploy/tikv-20160

10.0.1.2:20160 tikv          10.0.1.2    20160/20180                       Up      data/tikv-20160
    ↪      deploy/tikv-20160

10.0.1.5:20160 tikv         10.0.1.5    20160/20180                      Up      data/tikv-20160
    ↪      deploy/tikv-20160

10.0.1.3:4000  tidb         10.0.1.3    4000/10080                       Up      -
    ↪ deploy/tidb-4000

10.0.1.4:4000  tidb         10.0.1.4    4000/10080                       Up      -
    ↪ deploy/tidb-4000

10.0.1.5:4000  tidb          10.0.1.5    4000/10080                        Up      -
    ↪ deploy/tidb-4000

10.0.1.3:9000   tiflash       10.0.1.3    9000/8123/3930/20170/20292/8234  Up       data/tiflash
    ↪ -9000        deploy/tiflash-9000

10.0.1.4:9000   tiflash       10.0.1.4    9000/8123/3930/20170/20292/8234  Up       data/tiflash
    ↪ -9000        deploy/tiflash-9000

10.0.1.5:9090  prometheus    10.0.1.5    9090                             Up      data/prometheus
    ↪ -9090  deploy/prometheus-9090

10.0.1.5:3000  grafana       10.0.1.5    3000                             Up      -
    ↪ deploy/grafana-3000

10.0.1.5:9093  alertmanager 10.0.1.5    9093/9094                        Up      data/
    ↪ alertmanager-9093 deploy/alertmanager-9093
```

8.2.1.4.2 2. 执行缩容操作

```
tiup cluster scale-in <cluster-name> --node 10.0.1.5:20160
```

其中 --node 参数为需要下线节点的 ID。

预期输出 Scaled cluster <cluster-name> in successfully 信息，表示缩容操作成功。

8.2.1.4.3 3. 检查集群状态

下线需要一定时间，下线节点的状态变为 Tombstone 就说明下线成功。

执行如下命令检查节点是否下线成功：

```
tiup cluster display <cluster-name>
```

打开浏览器访问监控平台 http://10.0.1.5:3000，监控整个集群的状态。

调整后，拓扑结构如下：

Host IP	Service
10.0.1.3	TiDB + TiFlash + TiCDC
10.0.1.4	TiDB + PD + TiFlash + TiCDC
10.0.1.5	TiDB + Monitor（TiKV 已删除）
10.0.1.1	TiKV
10.0.1.2	TiKV

8.2.1.5 缩容 TiFlash 节点

如果要缩容 IP 地址为 10.0.1.4 的一个 TiFlash 节点，可以按照如下步骤进行操作。

8.2.1.5.1 1. 根据 TiFlash 剩余节点数调整数据表的副本数

1. 查询是否有数据表的 TiFlash 副本数大于缩容后的 TiFlash 节点数。tobe_left_nodes 表示缩容后的 TiFlash 节点数。如果查询结果为空，可以开始执行缩容。如果查询结果不为空，则需要修改相关表的 TiFlash 副本数。

   ```
   SELECT * FROM information_schema.tiflash_replica WHERE REPLICA_COUNT >  'tobe_left_nodes';
   ```

2. 对所有 TiFlash 副本数大于缩容后的 TiFlash 节点数的表执行以下语句，new_replica_num 必须小于等于 tobe_left_nodes：

   ```
   ALTER TABLE <db-name>.<table-name> SET tiflash replica 'new_replica_num';
   ```

3. 重新执行步骤 1，确保没有数据表的 TiFlash 副本数大于缩容后的 TiFlash 节点数。

8.2.1.5.2 2. 执行缩容操作

接下来，请任选下列方案其一进行缩容。

方案一：通过 TiUP 缩容 TiFlash 节点

1. 通过以下命令确定需要下线的节点名称：

```
tiup cluster display <cluster-name>
```

2. 执行 scale-in 命令来下线节点，假设步骤 1 中获得该节点名为 10.0.1.4:9000

```
tiup cluster scale-in <cluster-name> --node 10.0.1.4:9000
```

方案二：手动缩容 TiFlash 节点

在特殊情况下（比如需要强制下线节点），或者 TiUP 操作失败的情况下，可以使用以下方法手动下线 TiFlash 节点。

1. 使用 pd-ctl 的 store 命令在 PD 中查看该 TiFlash 节点对应的 store id。

   - 在pd-ctl（tidb-ansible 目录下的 resources/bin 包含对应的二进制文件）中输入 store 命令。
   - 若使用 TiUP 部署，可以调用以下命令代替 pd-ctl：

```
tiup ctl:v<CLUSTER_VERSION> pd -u http://<pd_ip>:<pd_port> store
```

> 注意：
> 如果集群中有多个 PD 实例，只需在以上命令中指定一个活跃 PD 实例的 IP:端口即可。

2. 在 pd-ctl 中下线该 TiFlash 节点。

   - 在 pd-ctl 中输入 store delete <store_id>，其中 <store_id> 为上一步查到的该 TiFlash 节点对应的 store id。
   - 若通过 TiUP 部署，可以调用以下命令代替 pd-ctl：

```
tiup ctl:v<CLUSTER_VERSION> pd -u http://<pd_ip>:<pd_port> store delete <store_id>
```

> 注意：
> 如果集群中有多个 PD 实例，只需在以上命令中指定一个活跃 PD 实例的 IP:端口即可。

3. 等待该 TiFlash 节点对应的 store 消失或者 state_name 变成 Tombstone 再关闭 TiFlash 进程。
4. 手动删除 TiFlash 的数据文件，具体位置可查看在集群拓扑配置文件中 TiFlash 配置部分下的 data_dir 目录。
5. 从 TiUP 拓扑信息中删除已经下线的 TiFlash 节点信息：

```
tiup cluster scale-in <cluster-name> --node <pd_ip>:<pd_port> --force
```

> **注意：**
>
> 如果在集群中所有的 TiFlash 节点停止运行之前，没有取消所有同步到 TiFlash 的表，则需要手动在 PD 中清除同步规则，否则无法成功完成 TiFlash 节点的下线。

手动在 PD 中清除同步规则的步骤如下：

1. 查询当前 PD 实例中所有与 TiFlash 相关的数据同步规则。

```
curl http://<pd_ip>:<pd_port>/pd/api/v1/config/rules/group/tiflash
```

```
[
  {
    "group_id": "tiflash",
    "id": "table-45-r",
    "override": true,
    "start_key": "7480000000000000FF2D5F720000000000FA",
    "end_key": "7480000000000000FF2E00000000000000F8",
    "role": "learner",
    "count": 1,
    "label_constraints": [
      {
        "key": "engine",
        "op": "in",
        "values": [
          "tiflash"
        ]
      }
    ]
  }
]
```

2. 删除所有与 TiFlash 相关的数据同步规则。以 `id` 为 `table-45-r` 的规则为例，通过以下命令可以删除该规则。

```
curl -v -X DELETE http://<pd_ip>:<pd_port>/pd/api/v1/config/rule/tiflash/table-45-r
```

8.2.1.5.3 3. 查看集群状态

```
tiup cluster display <cluster-name>
```

打开浏览器访问监控平台 http://10.0.1.5:3000，监控整个集群的状态。

调整后，拓扑结构如下：

Host IP	Service
10.0.1.3	TiDB + TiFlash + TiCDC
10.0.1.4	TiDB + PD + TiCDC （TiFlash 已删除）
10.0.1.5	TiDB + Monitor
10.0.1.1	TiKV
10.0.1.2	TiKV

8.2.1.6 缩容 TiCDC 节点

如果要缩容 IP 地址为 10.0.1.4 的一个 TiCDC 节点，可以按照如下步骤进行操作。

8.2.1.6.1 1. 下线该 TiCDC 节点

```
tiup cluster scale-in <cluster-name> --node 10.0.1.4:8300
```

8.2.1.6.2 2. 查看集群状态

```
tiup cluster display <cluster-name>
```

打开浏览器访问监控平台 http://10.0.1.5:3000，监控整个集群的状态。

调整后，拓扑结构如下：

Host IP	Service
10.0.1.3	TiDB + TiFlash + TiCDC
10.0.1.4	TiDB + PD + (TiCDC 已删除）
10.0.1.5	TiDB + Monitor
10.0.1.1	TiKV
10.0.1.2	TiKV

8.2.2 使用 TiDB Operator

8.3 备份与恢复

8.3.1 TiDB 备份与恢复概述

基于 Raft 协议和合理的部署拓扑规划，TiDB 实现了集群的高可用，当集群中少数节点挂掉时，集群依然能对外提供服务。在此基础上，为了更进一步保证用户数据的安全，TiDB 还提供了集群的备份与恢复 (Backup & Restore, BR) 功能，作为数据安全的最后一道防线，使得集群能够免于严重的自然灾害，提供业务误操作 “复原” 的能力。

TiDB 备份恢复功能可以用于满足以下业务的需求：

- 备份集群数据到灾备系统，并保证 Recovery Point Objective (RPO) 低至 5 分钟，减少灾难场景下数据的丢失。

- 处理业务数据写错的案例，提供业务操作的“复原”能力。
- 审计业务的历史数据，满足司法审查的需求。
- 复制 (Clone) 生产环境，方便问题诊断、性能调优验证、仿真测试等。

8.3.1.1 使用须知

本部分介绍使用 TiDB 备份恢复功能前的注意事项，包括使用限制和使用建议。

8.3.1.1.1 使用限制

- PITR 仅支持恢复到全新的空集群。
- PITR 仅支持集群粒度的恢复，不支持对单个 database 或 table 的恢复。
- PITR 不支持恢复系统表中用户表和权限表的数据。
- 不支持在一个集群上同时运行多个数据备份任务。
- PITR 数据恢复任务运行期间，不支持同时运行日志备份任务，也不支持通过 TiCDC 同步数据到下游集群。

8.3.1.1.2 使用建议

进行快照备份

- 推荐在业务低峰时执行集群快照数据备份，这样能最大程度地减少对业务的影响。
- 不推荐同时运行多个集群快照数据备份任务。不同的任务并行，不仅会导致备份的性能降低，影响在线业务，还会因为任务之间缺少协调机制造成任务失败，甚至对集群的状态产生影响。

进行快照恢复

- BR 恢复数据时会尽可能多地占用恢复集群的资源，因此推荐恢复数据到新集群或离线集群。应避免恢复数据到正在提供服务的生产集群，否则，恢复期间会对业务产生不可避免的影响。

备份存储和网络配置

- 推荐使用支持 Amazon S3、 GCS 或 Azure Blob Storage 协议的存储系统保存备份数据；
- 应确保 BR、 TiKV 节点和备份存储系统有足够的网络带宽，备份存储系统能提供足够的写入和读取性能，否则，它们有可能成为备份恢复时的性能瓶颈。

8.3.1.2 功能使用

根据 TiDB 部署方式的不同，使用备份恢复功能的方式也不同。本文主要介绍在本地部署方式下，如何使用 br 命令行工具进行 TiDB 的备份和恢复。

其它 TiDB 的部署方式的备份恢复功能使用，可以参考：

- 备份恢复部署在 TiDB Cloud 上的 TiDB。推荐在 TiDB Cloud 上创建 TiDB 集群，集群的运维管理将由 TiDB Cloud 团队托管完成，你可以聚焦于业务。
- 备份恢复部署在 Kubernetes 上的 TiDB。如果你使用 TiDB Operator 在 Kubernetes 中部署了 TiDB 集群，建议通过 Kubernetes CustomResourceDefinition (CRD) 来提交备份和恢复任务。

8.3.1.3 功能介绍

使用备份恢复功能，你可以进行以下两类操作：

- 对集群进行备份：你可以对集群某个时间点的全量数据进行备份（全量备份），也可以对业务写入在 TiDB 产生的数据变更记录进行备份（日志备份，日志指的是 TiKV 中的 kv 变更数据的记录）。
- 恢复备份数据：
  - 你可以恢复某个全量备份，或者全量备份中的部分库/表，将目标集群恢复到该全量备份对应的数据状态。
  - 基于备份（全量和日志）数据，你可以指定任意时间点，将目标集群恢复到该时间点所对应的备份集群数据状态 (Point-in-time recovery, PITR)。

8.3.1.3.1 备份集群数据

全量备份是对集群某个时间点的全量数据进行备份。TiDB 支持以下方式的全量备份：

- 快照数据备份：TiDB 集群快照数据包含某个物理时间点上集群满足事务一致性的所有数据。BR 支持备份集群快照数据，使用请参考快照备份。

全量备份一般会占用不小的存储空间，且只包含某个时间点的集群数据。如果你需要灵活的选择恢复的时间点，即实现 PITR，可以按以下说明同时使用两种备份方式：

- 开启日志备份任务后，任务会在所有 TiKV 节点上持续运行，以小批量的形式定期将 TiDB 变更数据备份到指定存储中。
- 定期执行快照备份，备份集群全量数据到备份存储，例如在每天零点进行集群快照备份。

备份的性能，以及对集群的影响

- 在集群 CPU 和 I/O 资源充裕时，集群快照数据备份对 TiDB 集群的影响可以保持在 20% 以下，通过合理的配置 TiDB 集群用于备份资源，影响可以降低到 10% 及更低；在集群 CPU 和 I/O 资源紧张时，可以通过调整 TiKV 配置项`backup.num-threads` 来调整备份任务使用的工作线程数量以降低备份任务对 TiDB 集群的影响；单 TiKV 存储节点的备份速度可以达到 50 MB/s ~ 100 MB/s，备份速度具有可扩展性；更详细说明请参考备份性能和影响。
- 单独运行日志备份时影响约在 5%。日志备份每隔 3 ~ 5 分钟将上次刷新后产生的变更数据记录刷新到备份存储中，可以实现低至五分钟 RPO 的集群容灾目标。

8.3.1.3.2 恢复备份数据

与备份功能相对应，你可以进行两种类型的恢复：全量恢复和 PITR。

- 恢复某个全量备份
  - 恢复集群快照数据备份：你可以在一个空集群或不存在数据冲突（相同 schema 或 table）的集群执行快照备份恢复，将该集群恢复到快照备份对应数据状态。使用请参考恢复快照备份。此外你可以只恢复备份数据中指定库/表的局部数据。该功能在恢复过程中过滤掉不需要的数据。使用请参考恢复备份数据中指定库表的数据。

- 恢复到集群的历史任意时间点 (PITR)
    - 通过 `br restore point` 功能。你可以指定要恢复的时间点，恢复时间点之前最近的快照数据备份，以及日志备份数据。BR 会自动判断和读取恢复需要的数据，然后将这些数据依次恢复到指定的集群。

恢复的性能

- 恢复集群快照数据备份，速度可以达到单 TiKV 存储节点 100 MiB/s，恢复速度具有可扩展性；BR 只支持恢复数据到新集群，会尽可能多的使用恢复集群的资源。更详细说明请参考恢复性能和影响。
- 恢复日志备份数据，速度可以达到 30 GiB/h。更详细说明请参考PITR 性能和影响。

8.3.1.4 备份存储

TiDB 支持将数据备份到 Amazon S3、Google Cloud Storage (GCS)、Azure Blob Storage、NFS，或者实现 S3 协议的其他文件存储服务。更多备份存储的详细信息，请参考如下内容：

- 使用 URI 格式指定备份存储
- 配置备份存储的访问权限

8.3.1.5 兼容性

在使用备份恢复功能之前，需要先了解 BR 工具与其他功能的兼容性以及使用限制。

8.3.1.5.1 与其他功能的兼容性

某些功能在开启或关闭状态下，会导致备份恢复功能使用出错。因此需要保证恢复集群的这些配置，与备份集群备份时的配置相同。

功能	相关 issue	解决方式
GBK charset		BR 在 v5.4.0 之前不支持恢复 `charset = GBK` 的表。并且，任何版本的 BR 都不支持恢复 `charset = GBK` 的表到 v5.4.0 之前的 TiDB 集群。

功能	相关 issue	解决方式
聚簇索引	#565	确保恢复时集群的 `tidb_enable_clustered_index` 全局变量和备份时一致，否则会导致数据不一致的问题，例如 `default not found` 和数据索引不一致。

功能	相关 issue	解决方式
New collation	#352	确保恢复时集群的 new_collations_enabled_on_first_bootstrap 变量值和备份时的一致，否则会导致数据索引不一致和 checksum 通不过。更多信息，请参考FAQ - BR 为什么会报 new_collations_enabled_on_first_bootstrap 不匹配？。
全局临时表		确保使用 BR v5.3.0 及以上版本进行备份和恢复，否则会导致全局临时表的表定义错误。

功能	相关 issue	解决方式
TiDB Lightning Physical Import		上游数据库使用 TiDB Lightning Physical 方式导入的数据，无法作为数据日志备份下来。推荐在数据导入后执行一次全量备份，细节参考上游数据库使用 TiDB Lightning Physical 方式导入数据的恢复。

8.3.1.5.2 版本间兼容性

> **注意：**
>
> 在执行备份和恢复操作时，建议使用和 TiDB 集群相同版本的 BR。

在执行备份和恢复操作前，BR 会比较自身版本与 TiDB 集群版本。如果版本之间不兼容，BR 会提示报错并退出。如要跳过版本检查，可以设置 `--check-requirements=false`。需要注意的是，跳过检查可能会遇到数据

不兼容的问题。

备份版本（纵向） 恢复版本（横向）	恢复到 TiDB v6.0	恢复到 TiDB v6.1	恢复到 TiDB v6.2	恢复到 TiDB v6.3、v6.4 或 v6.5	恢复到 TiDB v6.6
TiDB v6.0、v6.1、v6.2、v6.3、v6.4 或 v6.5 快照备份	兼容（已知问题，如果备份数据中包含空库可能导致报错，参考 #36379）	兼容	兼容	兼容	兼容（需使用 v6.6 及以上版本的 BR）
TiDB v6.3、v6.4、v6.5 或 v6.6 日志备份	不兼容	不兼容	不兼容	兼容	兼容

8.3.1.6 探索更多

- TiDB 快照备份与恢复使用指南
- TiDB 日志备份与 PITR 使用指南
- 备份存储

8.3.2 架构设计

8.3.2.1 TiDB 备份与恢复功能架构概述

正如TiDB 备份与恢复概述所介绍，TiDB 备份恢复功能包含了多种不同类型的集群数据对象的备份与恢复实现。这些功能都以 Backup & Restore (BR) 和 TiDB Operator 为使用入口，创建相应的任务从 TiKV 节点上备份数据，或者恢复数据到 TiKV 节点。

关于各种备份恢复功能的实现架构，请参考以下链接：

- 全量数据备份与恢复
  - 备份集群快照数据
  - 恢复快照备份数据
- 数据变更日志备份
  - 日志备份 - 备份 kv 数据变更
- Point-in-time recovery (PITR)
  - 恢复到指定时间点

8.3.2.2 TiDB 快照备份与恢复功能架构

本文以使用 BR 工具进行备份与恢复为例，介绍 TiDB 集群快照数据备份和恢复的架构设计与流程。

8.3.2.2.1 架构设计

快照数据备份和恢复的架构如下：

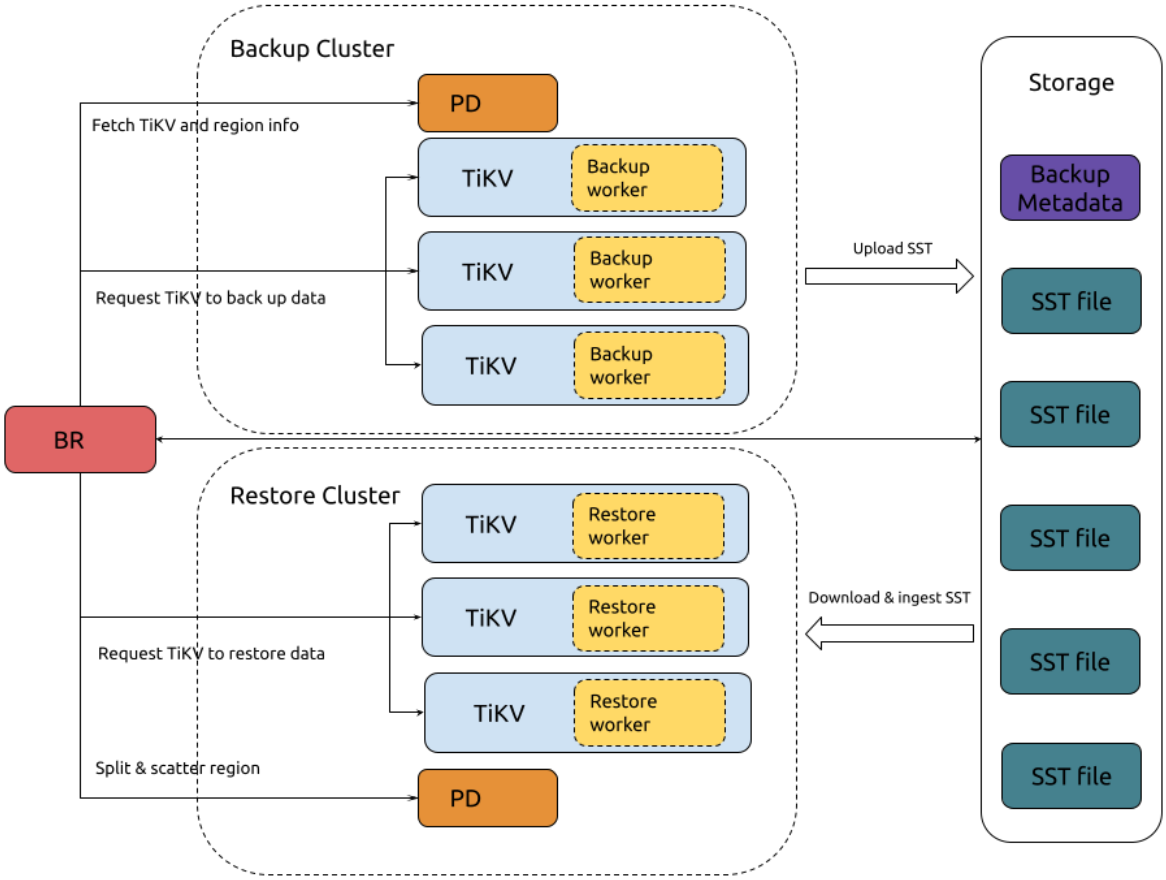


图 38: BR snapshot backup and restore architecture

8.3.2.2.2 备份流程

集群快照数据备份的流程如下：

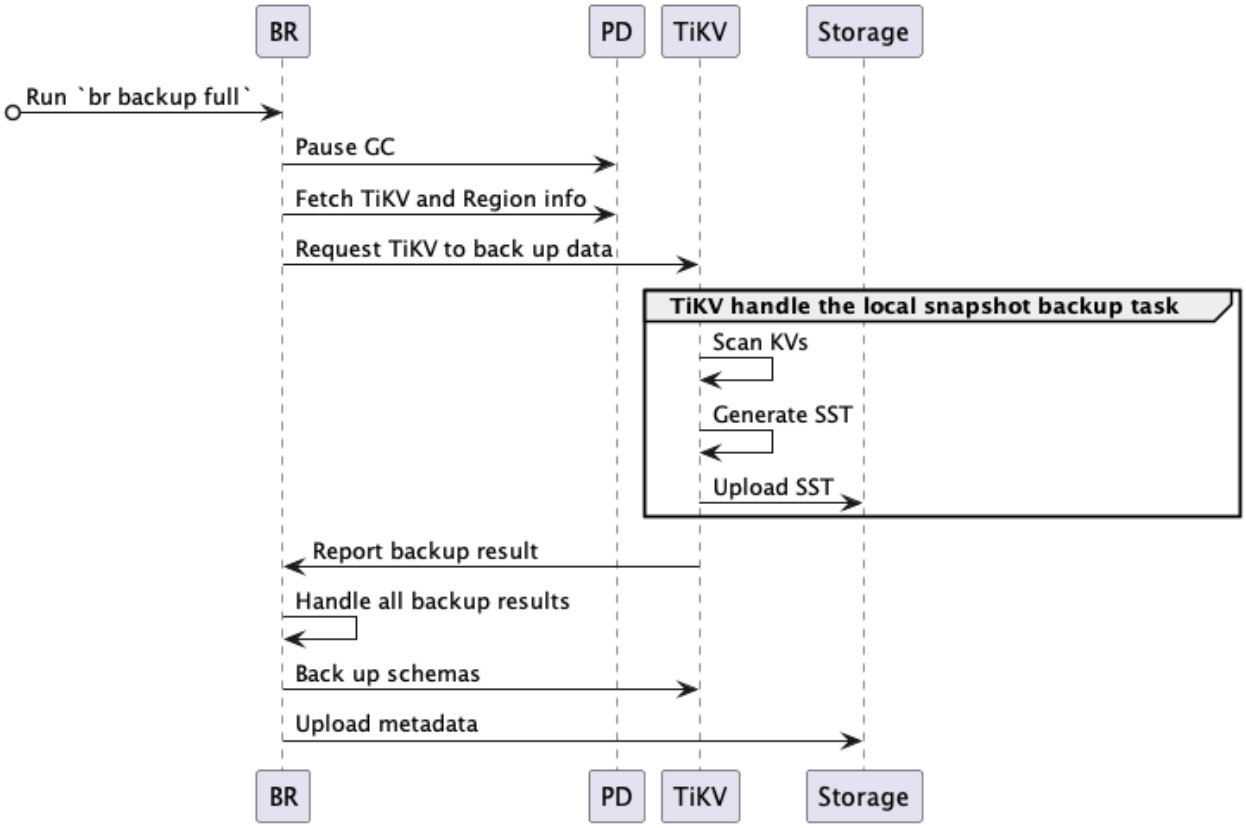


图 39: snapshot backup process design

完整的备份交互流程描述如下：

1. BR 接收备份命令 `br backup full`。
    - 获得备份快照点 (backup ts) 和备份存储地址。
2. BR 调度备份数据。
    - Pause GC：配置 TiDB GC，防止要备份的数据被TiDB GC 机制回收。
    - Fetch TiKV and Region info：访问 PD，获取所有 TiKV 节点访问地址以及数据的Region 分布信息。
    - Request TiKV to back up data：创建备份请求，发送给 TiKV 节点，备份请求包含 backup ts、需要备份的 region、备份存储地址。
3. TiKV 接受备份请求，初始化 backup worker。
4. TiKV 备份数据。
    - Scan KVs：backup worker 从 Region (only leader) 读取 backup ts 对应的数据。
    - Generate SST：backup worker 将读取到的数据保存到 SST 文件，存储在内存中。
    - Upload SST：backup worker 上传 SST 文件到备份存储中。
5. BR 从各个 TiKV 获取备份结果。

- 如果局部数据因为 Region 变动而备份失败，比如 TiKV 节点故障，BR 将重试这些数据的备份。
- 如果任意数据被判断为不可重试的备份失败，则备份任务失败。
- 全部数据备份成功后，则在最后完成元信息备份。

6. BR 备份元信息。

    - Back up schemas：备份 table schema，同时计算 table data checksum。
    - Upload metadata：生成 backup metadata，并上传到备份存储。backup metadata 包含 backup ts、表和对应的备份文件、data checksum 和 file checksum 等信息。

8.3.2.2.3 恢复流程

恢复集群快照备份数据的流程如下：

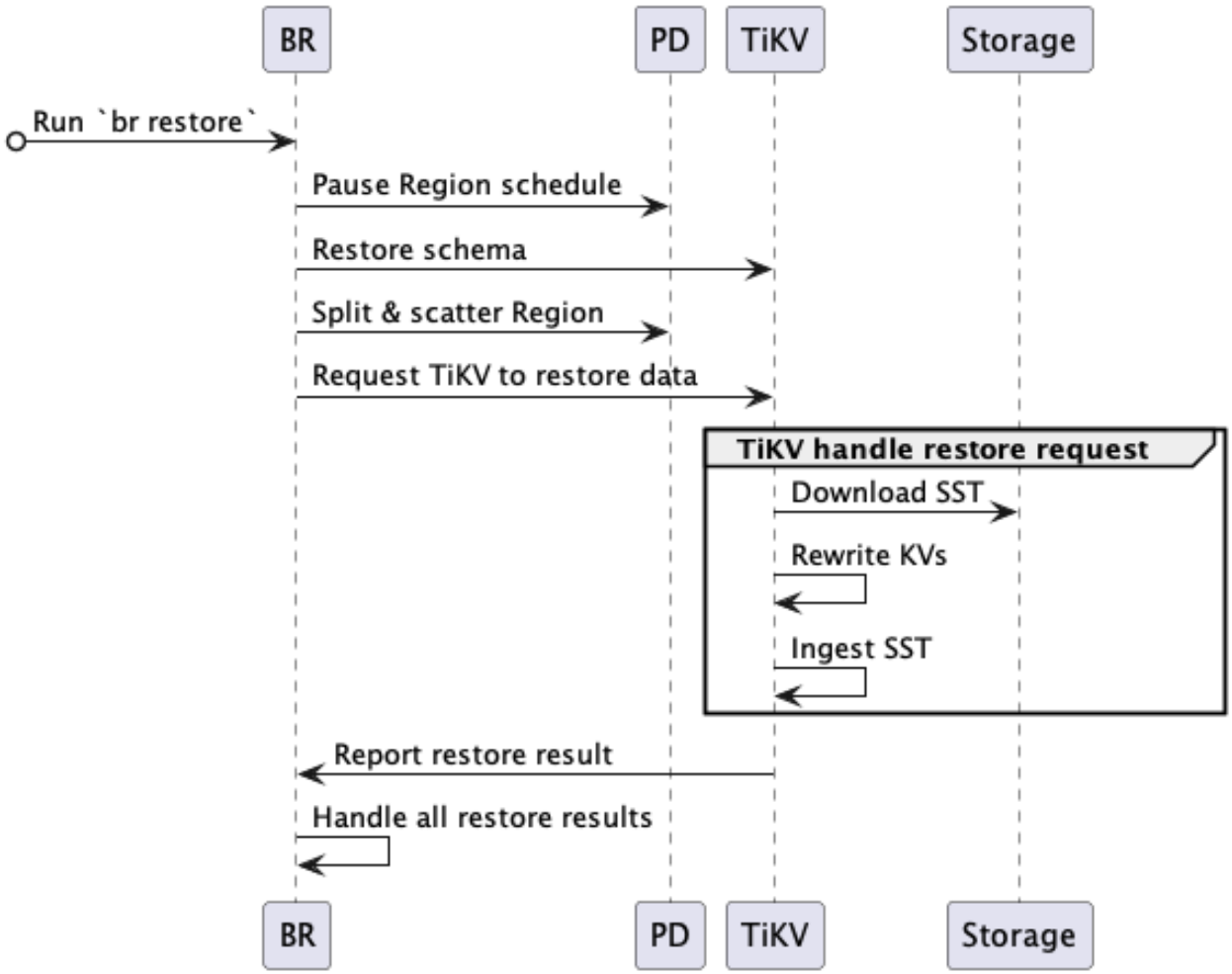


图 40: snapshot restore process design

完整的恢复交互流程描述如下：

1. BR 接收恢复命令 `br restore`。

- 获得快照备份数据存储地址、要恢复的 database 或 table。
- 检查要恢复的 table 是否存在及是否符合要求。

2. BR 调度恢复数据。

    - Pause Region schedule：请求 PD 在恢复期间关闭自动 Region schedule。
    - Restore schema：读取备份数据的 schema、恢复的 database 和 table（注意新建表的 table ID 与备份数据可能不一样）。
    - Split & scatter Region：BR 基于备份数据信息，请求 PD 分配 Region (split Region)，并调度 Region 均匀分布到存储节点上 (scatter Region)。每个 Region 都有明确的数据范围 [start key, end key)。
    - Request TiKV to restore data：根据 PD 分配的 Region 结果，发送恢复请求到对应的 TiKV 节点，恢复请求包含要恢复的备份数据及 rewrite 规则。

3. TiKV 接受恢复请求，初始化 restore worker。

    - restore worker 计算恢复数据需要读取的备份数据。

4. TiKV 恢复数据。

    - Download SST：restore worker 从备份存储中下载相应的备份数据到本地。
    - Rewrite KVs：restore worker 根据新建表 table ID，对备份数据 kv 进行重写，即将原有的kv 编码中的 table ID 替换为新创建的 table ID。对 index ID，restore worker 也进行相同处理。
    - Ingest SST：restore worker 将处理好的 SST 文件 ingest 到 RocksDB 中。
    - Report restore result：restore worker 返回恢复结果给 BR。

5. BR 从各个 TiKV 获取恢复结果。

    - 如果局部数据恢复因为 `RegionNotFound` 或 `EpochNotMatch` 等原因失败，比如 TiKV 节点故障，BR 重试恢复这些数据。
    - 如果存在备份数据不可重试的恢复失败，则恢复任务失败。
    - 全部备份都恢复成功后，则整个恢复任务成功。

详细的快照数据备份恢与恢复流程设计，可以参考备份恢复设计方案。

8.3.2.2.4 备份文件

文件类型

快照备份会产生如下类型文件：

- SST 文件：存储 TiKV 备份下来的数据信息。单个 SST 文件大小等于 TiKV Region 的大小。
- `backupmeta` 文件：存储本次备份的元信息，包括备份文件数、备份文件的 Key 区间、备份文件大小和备份文件 Hash (sha256) 值。
- `backup.lock` 文件：用于防止多次备份到同一目录。

SST 文件的命名格式

当备份数据到 Google Cloud Storage 或 Azure Blob Storage 时，SST 文件以 `storeID_regionID_regionEpoch_keyHash_timestamp_cf` 的格式命名。格式名的解释如下：

- `storeID`：TiKV 节点编号

- `regionID`：Region 编号
- `regionEpoch`：Region 版本号
- keyHash：Range startKey 的 Hash (sha256) 值，确保唯一性
- `timestamp`：TiKV 节点生成 SST 文件名时刻的 Unix 时间戳
- `cf`：RocksDB 的 ColumnFamily（只备份 cf 为 `default` 或 `write` 的数据）

当备份数据到 Amazon S3 或网络盘上时，SST 文件以 `regionID_regionEpoch_keyHash_timestamp_cf` 的格式命名。

- `regionID`：Region 编号
- `regionEpoch`：Region 版本号
- keyHash：Range startKey 的 Hash (sha256) 值，确保唯一性
- `timestamp`：TiKV 节点生成 SST 文件名时刻的 Unix 时间戳
- `cf`：RocksDB 的 ColumnFamily（只备份 cf 为 `default` 或 `write` 的数据）

SST 文件存储格式

- 关于 SST 文件存储格式，可以参考 RocksDB SST table 介绍。
- 关于 SST 文件中存储的备份数据编码格式，可以参考TiDB 表数据与 Key-Value 的映射关系。

备份文件目录结构

将数据备份到 Google Cloud Storage 或 Azure Blob Storage 上时，SST 文件、`backupmeta` 文件和 `backup.lock` 文件在同一目录下。目录结构如下：

```
.
└── 20220621
    ├── backupmeta
    ├── backup.lock
    ├── {storeID}-{regionID}-{regionEpoch}-{keyHash}-{timestamp}-{cf}.sst
    ├── {storeID}-{regionID}-{regionEpoch}-{keyHash}-{timestamp}-{cf}.sst
    └── {storeID}-{regionID}-{regionEpoch}-{keyHash}-{timestamp}-{cf}.sst
```

将数据备份到 Amazon S3 或网络盘上时，SST 文件会根据 `storeID` 划分子目录。目录结构如下：

```
.
└── 20220621
    ├── backupmeta
    ├── backup.lock
    ├── store1
    │   └── {regionID}-{regionEpoch}-{keyHash}-{timestamp}-{cf}.sst
    ├── store100
    │   └── {regionID}-{regionEpoch}-{keyHash}-{timestamp}-{cf}.sst
    ├── store2
    │   └── {regionID}-{regionEpoch}-{keyHash}-{timestamp}-{cf}.sst
    ├── store3
    ├── store4
    └── store5
```

8.3.2.2.5 探索更多

- TiDB 快照备份与恢复使用指南

8.3.2.3 TiDB 日志备份与 PITR 功能架构

本文以使用 BR 工具进行备份与恢复为例，介绍 TiDB 集群日志备份与 Point-in-time recovery (PITR) 的架构设计与流程。

8.3.2.3.1 架构设计

日志备份和 PITR 的架构如下：

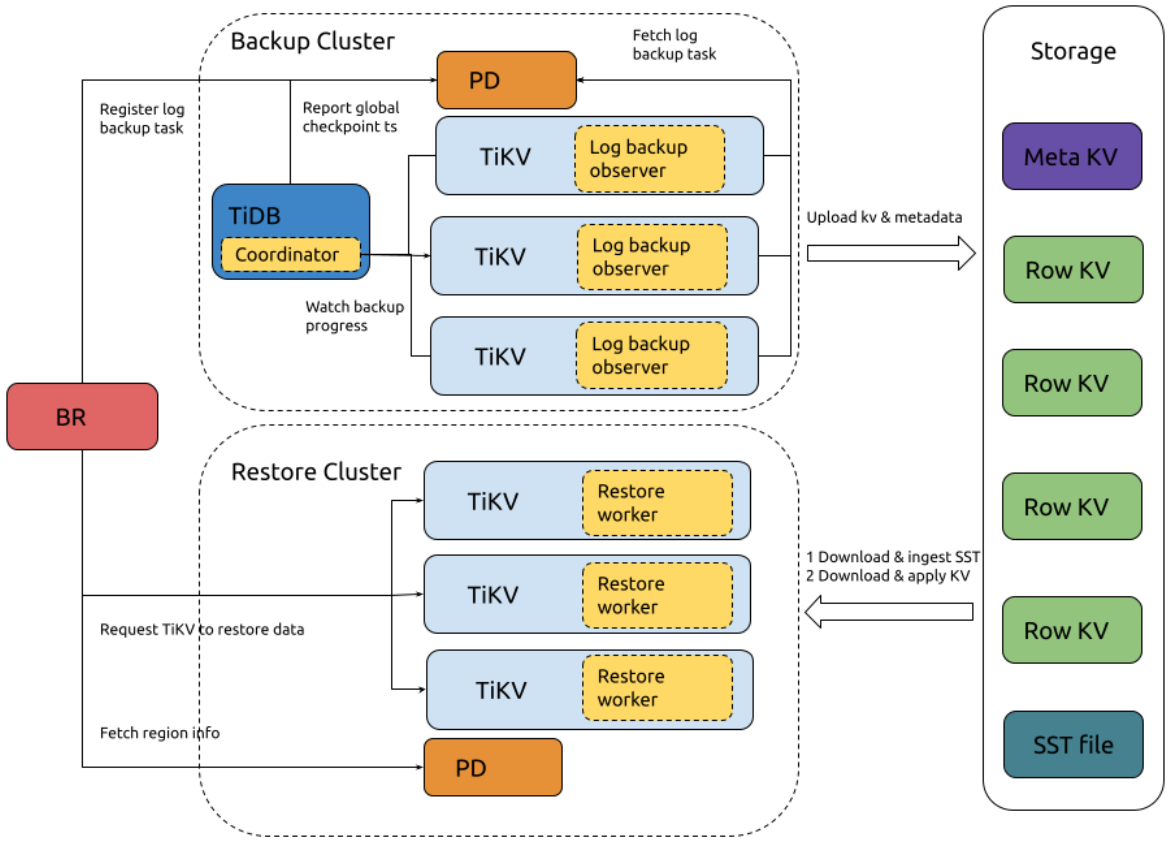


图 41: BR log backup and PITR architecture

8.3.2.3.2 日志备份

日志备份的流程如下：

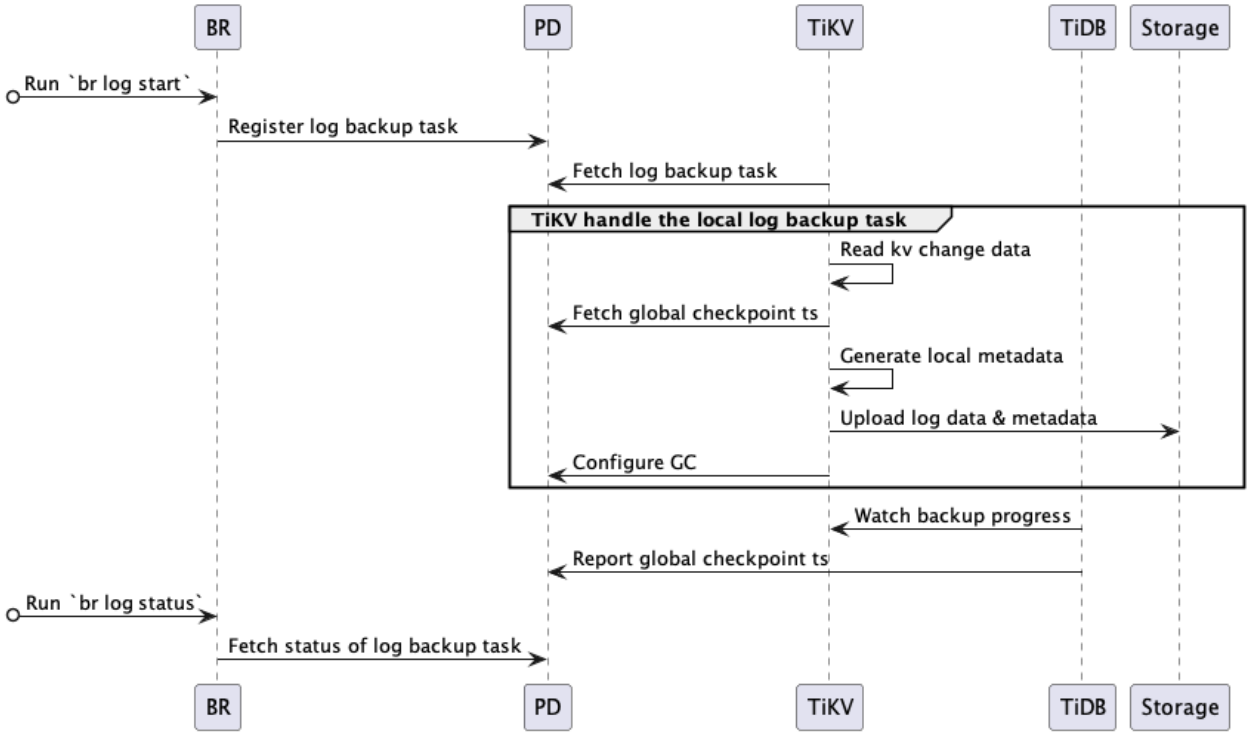


图 42: BR log backup process design

系统组件和关键概念：

- local metadata：表示单 TiKV 节点备份下来的数据元信息，主要包括：local checkpoint ts、 global checkpoint ts、备份文件信息。
- local checkpoint ts (in local metadata)：表示这个 TiKV 中所有小于 local checkpoint ts 的日志数据已经备份到目标存储。
- global checkpoint ts：表示所有 TiKV 中小于 global checkpoint ts 的日志数据已经备份到目标存储。它由运行在 TiDB 中的 Coordinator 模块收集所有 TiKV 的 local checkpoint ts 计算所得，然后上报给 PD。
- TiDB Coordinator 组件：TiDB 集群的某个节点会被选举为 Coordinator，负责收集和计算整个日志备份任务的进度 (global checkpoint ts)。 该组件设计上无状态，在其故障后可以从存活的 TiDB 节点中重新选出一个节点作为 Coordinator。
- TiKV log observer 组件：运行在 TiDB 集群的每个 TiKV 节点，负责从 TiKV 读取和备份日志数据。TiKV 节点故障的话，该节点负责备份数据范围，在 Region 重新选举后，会被其他 TiKV 节点负责，这些节点会从 global checkpoint ts 重新备份故障范围的数据。

完整的备份交互流程描述如下：

1. BR 接收备份命令 `br log start`。

- 解析获取日志备份任务的 checkpoint ts（日志备份起始位置）、备份存储地址。
- Register log backup task：在 PD 注册日志备份任务 (log backup task)。

2. TiKV 监控日志备份任务的创建与更新。

- Fetch log backup task：每个 TiKV 节点的 log backup observer 监听 PD 中日志备份任务的创建与更新，然后备份该节点上在备份范围内的数据。

3. TiKV log backup observer 持续地备份 KV 变更日志。

- Read kv change data：读取 kv 数据变更，然后保存到自定义格式的备份文件中。
- Fetch global checkpoint ts：定期从 PD 查询 global checkpoint ts。
- Generate local metadata：定期生成 local metadata（包含 local checkpoint ts、global checkpoint ts、备份文件信息）。
- Upload log data & metadata：定期将日志备份数据和 local metadata 上传到备份存储中。
- Configure GC：请求 PD 阻止未备份的数据（大于 local checkpoint ts）被TiDB GC 机制回收掉。

4. TiDB Coordinator 监控日志备份进度。

- Watch backup progress：轮询所有 TiKV 节点，获取各个 Region 的备份进度 (Region checkpoint ts)。
- Report global checkpoint ts：根据各个 Region checkpoint ts，计算整个日志备份任务的进度 (global checkpoint ts)，然后上报给 PD。

5. PD 持久化日志备份任务状态。可以通过 `br log status` 查询。

8.3.2.3.3 PITR

PITR 的流程如下：

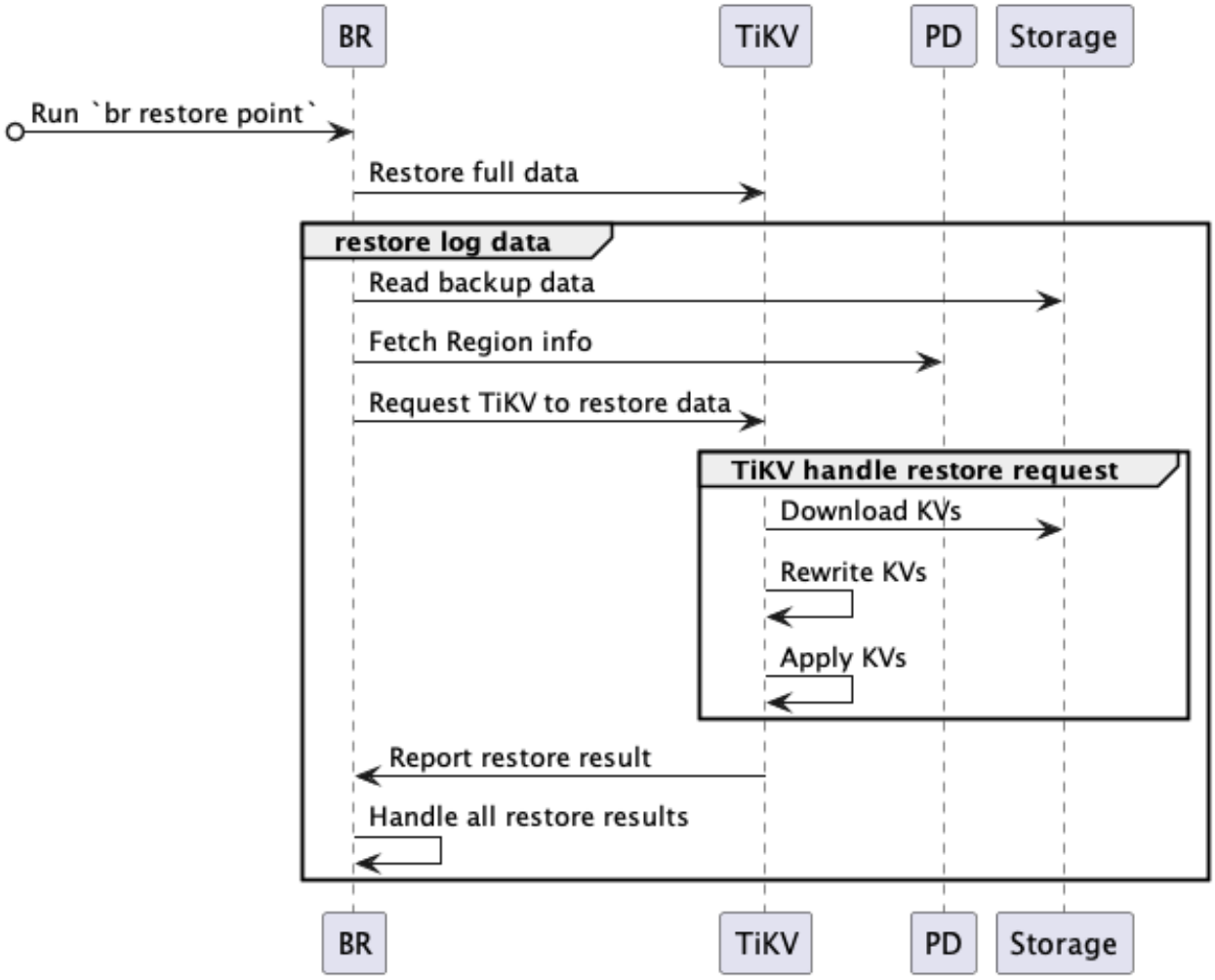


图 43: Point-in-time recovery process design

完整的 PITR 交互流程描述如下：

1. BR 接收恢复命令 `br restore point`。

- 解析获取全量备份数据地址、日志备份数据地址、恢复到的时间点。
- 查询备份数据中恢复数据对象（database 或 table），并检查要恢复的表是否存在并符合要求。

2. BR 恢复全量备份。

- 进行快照备份数据恢复，恢复流程参考恢复快照备份数据。

3. BR 恢复日志备份。

- Read backup data：读取日志备份数据，计算需要恢复的日志备份数据。

- Fetch Region info：访问 PD，获取所有 Region 和 KV range 的对应关系。
- Request TiKV to restore data：创建日志恢复请求，发送到对应的 TiKV，日志恢复请求包含要恢复的日志备份数据信息。

4. TiKV 接受 BR 的恢复请求，初始化 log restore worker。

- log restore worker 获取需要恢复的日志备份数据。

5. TiKV 恢复日志备份数据。

- Download KVs：log restore worker 根据日志恢复请求中要恢复的备份数据，从备份存储中下载相应的备份数据到本地。
- Rewrite KVs：log restore worker 根据恢复集群表的 table ID 对备份数据的 kv 进行重写 —— 将原有的kv 编码中的 table ID 替换为新创建的 table ID。对 index ID，log restore worker 也进行相同的处理。
- Apply KVs：log restore worker 将处理好的 kv 通过 raft 接口写入 store (RocksDB) 中。
- Report restore result：log restore worker 返回恢复结果给 BR。

6. BR 从各个 TiKV 获取恢复结果。

- 如果局部数据恢复因为 RegionNotFound 或 EpochNotMatch 等原因失败，比如 TiKV 节点故障，BR 重试恢复这些数据。
- 如果存在备份数据不可重试的恢复失败，则恢复任务失败。
- 全部备份数据都恢复成功后，则恢复任务成功。

8.3.2.3.4 日志备份文件

日志备份会产生如下类型文件：

- `{min_ts}-{uuid}.log` 文件：存储备份下来的 kv 数据变更记录。其中 `{min_ts}` 是该文件中所有 kv 数据变更记录数对应的最小 ts；`{uuid}` 是生成该文件的时候随机生成的。
- `{checkpoint_ts}-{uuid}.meta` 文件：每个 TiKV 节点每次上传日志备份数据时会生成一个该文件，保存本次上传的所有日志备份数据文件。其中 `{checkpoint_ts}` 是本节点的日志备份的 checkpoint，所有 TiKV 节点的最小的 checkpoint 就是日志备份任务最新的 checkpoint；`{uuid}` 是生成该文件的时候随机生成的。
- `{store_id}.ts` 文件：每个 TiKV 节点每次上传日志备份数据时会使用 global checkpoint ts 更新该文件。其中 `{store_id}` 是 TiKV 的 store ID。
- `v1_stream_truncate_safepoint.txt` 文件：保存最近一次通过 `br log truncate` 删除日志备份数据后，存储中最早的日志备份数据对应的 ts。

备份文件目录结构

```
.
├── v1
│   ├── backupmeta
│   │   ├── {min_restored_ts}-{uuid}.meta
│   │   ├── {checkpoint}-{uuid}.meta
```

```
|   ├── global_checkpoint
|   |   ├── {store_id}.ts
|   ├── {date}
|   |   ├── {hour}
|   |   |   ├── {store_id}
|   |   |   |   ├── {min_ts}-{uuid}.log
|   |   |   |   ├── {min_ts}-{uuid}.log
├── v1_stream_truncate_safepoint.txt
```

具体示例如下：

```
.
├── v1
|   ├── backupmeta
|   |   ├── ...
|   |   ├── 435213818858112001-e2569bda-a75a-4411-88de-f469b49d6256.meta
|   |   ├── 435214043785779202-1780f291-3b8a-455e-a31d-8a1302c43ead.meta
|   |   ├── 435214443785779202-224f1408-fff5-445f-8e41-ca4fcfbd2a67.meta
|   ├── global_checkpoint
|   |   ├── 1.ts
|   |   ├── 2.ts
|   |   ├── 3.ts
|   ├── 20220811
|   |   ├── 03
|   |   |   ├── 1
|   |   |   |   ├── ...
|   |   |   |   ├── 435213866703257604-60fcbdb6-8f55-4098-b3e7-2ce604dafe54.log
|   |   |   |   ├── 435214023989657606-72ce65ff-1fa8-4705-9fd9-cb4a1e803a56.log
|   |   |   ├── 2
|   |   |   |   ├── ...
|   |   |   |   ├── 435214102632857605-11deba64-beff-4414-bc9c-7a161b6fb22c.log
|   |   |   |   ├── 435214417205657604-e6980303-cbaa-4629-a863-1e745d7b8aed.log
|   |   |   ├── 3
|   |   |   |   ├── ...
|   |   |   |   ├── 435214495848857605-7bf65e92-8c43-427e-b81e-f0050bd40be0.log
|   |   |   |   ├── 435214574492057604-80d3b15e-3d9f-4b0c-b133-87ed3f6b2697.log
├── v1_stream_truncate_safepoint.txt
```

8.3.3 使用 BR 进行备份与恢复

8.3.3.1 TiDB 备份与恢复功能使用概述

本文介绍使用 TiDB 备份与恢复功能的最佳实践，包括如何选择备份方案、如何管理备份数据，以及如何安装和部署备份恢复工具。

8.3.3.1.1 使用概览

深入 TiDB 备份和恢复功能使用之前，建议先了解推荐的集群备份和恢复方案。

如何备份数据？

TiDB 支持两种类型的备份，应该使用哪种备份？全量备份包含集群某个时间点的全量数据，日志备份包含业务写入在 TiDB 产生的数据变更记录。推荐这两种备份方式一起使用：

- 开启日志备份：运行 `br log start` 命令来启动日志备份任务，任务会在每个 TiKV 节点上持续运行，以小批量的形式定期将 TiDB 变更数据备份到指定存储中。
- 定期执行快照（全量）备份：运行 `br backup full` 命令来备份集群快照到备份存储，例如在每天零点进行集群快照备份。

如何管理备份数据？

BR 只提供备份和恢复的基础功能，尚不支持备份管理的功能，因此你需要自行规划备份数据的管理事项，可能包含以下的问题：

- 选择哪种备份存储系统？
- 数据备份的时候，备份数据应该放在什么目录下？
- 全量备份和日志备份的数据目录如何组织？
- 如何处理存储系统中历史备份数据？

下面是处理这些问题的推荐方式：

选择备份存储

Amazon S3、Google Cloud Storage (GCS)、Azure Blob Storage 是推荐的存储系统选择，使用这些系统，你无需担心备份容量、备份带宽规划等。

如果 TiDB 集群部署在自建机房中，则推荐以下方式：

- 搭建 MinIO 作为备份存储系统，使用 S3 协议将数据备份到 MinIO 中。
- 挂载 NFS（如 NAS）盘到 br 工具和所有的 TiKV 实例，使用 POSIX file system 接口将备份数据写入对应的 NFS 目录中。

> 注意：
>
> 如果没有挂载 NFS 到 br 工具或 TiKV 节点，或者使用了支持 S3、GCS 或 Azure Blob Storage 协议的远端存储，那么 br 工具备份的数据会在各个 TiKV 节点生成。注意这不是推荐的 br 工具使用方式，因为备份数据会分散在各个节点的本地文件系统中，聚集这些备份数据可能会造成数据冗余和运维上的麻烦，而且在不聚集这些数据便直接恢复的时候会遇到 `SST file not found` 报错。

组织备份数据目录

- 全量备份和日志备份保存在相同的目录下，方便统一管理，例如 backup-${cluster-id}。
- 每个全量备份保存到命名带有备份日期的目录下，例如 backup-${cluster-id}/fullbackup-202209081330。
- 日志备份数据保存在一个固定目录下，例如 backup-${cluster-id}/logbackup。日志备份程序会在 logbackup 目录中每天切分出来一个新的子目录来区分每天的日志备份数据。

处理历史备份数据

假设你设置了备份保留期，即保存固定时间的备份数据，比如 7 天。请注意备份保留期的概念，后面使用教程中也会多次遇到。

- 进行 PITR 不仅需要恢复时间点之前的全量备份，还需要全量备份和恢复时间点之间的日志备份，因此，对于超过备份保留期的日志备份，应执行 br log truncate 命令删除指定时间点之前的备份。建议只清理全量快照之前的日志备份。
- 对于超过备份保留期的全量备份，建议直接删除或者归档全量备份的目录。

如何恢复数据？

- 如果你只有全量备份数据，或者想恢复某个确定的全量备份，那么可以使用 br restore 恢复指定的全量备份。
- 如果你按照以上推荐的的方式进行备份，那么你可以使用 br restore point 恢复到备份保留期内任意时间点。

8.3.3.1.2 部署和使用 BR

使用备份恢复功能的部署要求如下：

- BR、TiKV 节点和备份存储系统需要提供大于备份速度的的网络带宽。当集群特别大的时候，备份和恢复速度上限受限于备份网络的带宽。
- 备份存储系统还需要提供足够的写入/读取性能 (IOPS)，否则它有可能成为备份恢复时的性能瓶颈。
- TiKV 节点需要为备份准备至少额外的两个 CPU core 和高性能的磁盘，否则备份将对集群上运行的业务产生影响。
- 推荐 br 工具运行在 8 核 +/16 GB+ 的节点上。

目前支持以下几种方式来使用 BR。

通过命令行工具（推荐）

TiDB 支持使用 br 工具进行备份恢复。

- 安装方法可以使用 TiUP 在线安装：tiup install br。
- 了解如何使用 br 命令行工具进行备份与恢复，请参阅：
  - TiDB 快照备份与恢复功能使用
  - TiDB 日志备份与 PITR 功能使用
  - TiDB 集群备份与恢复实践示例

通过 SQL 语句

TiDB 支持使用 SQL 语句进行全量快照备份和恢复：

- BACKUP 进行全量快照数据备份。
- RESTORE 进行快照备份恢复。
- SHOW BACKUPS|RESTORES 查看备份恢复的进度。

在 Kubernetes 环境下通过 TiDB Operator

在 Kubernetes 环境下，支持通过 TiDB Operator 支持以 S3、GCS、Azure blob storage 作为备份存储，并从这些存储系统中恢复备份数据。使用文档请参阅使用 TiDB Operator 进行备份恢复。

8.3.3.1.3 探索更多

- TiDB 备份与恢复概述
- TiDB 备份与恢复功能架构

8.3.3.2 TiDB 快照备份与恢复使用指南

本文介绍如何使用 br 命令行工具进行 TiDB 快照备份和恢复。使用前，请先安装 br 命令行工具。

快照备份是集群全量备份的一种实现。它基于 TiDB 的多版本并发控制 (MVCC) 实现，将指定快照包含的所有数据备份到目标存储中。备份下来的数据大小约等于集群（压缩后的）单副本数据大小。备份完成之后，你可以在一个空集群或不存在数据冲突（相同 schema 或 table）的集群执行快照备份恢复，将集群恢复到快照备份时的数据状态，同时恢复功能会依据集群副本设置恢复出多副本。

除了基础的备份和恢复功能，快照备份和恢复还提供以下功能：

- 备份指定时间点的快照数据
- 恢复指定数据库或表的数据

8.3.3.2.1 对集群进行快照备份

> 注意：
>
> - 以下场景采用 Amazon S3 Access key 和 Secret key 授权方式来进行模拟。如果使用 IAM Role 授权，需要设置 `--send-credentials-to-tikv` 为 `false`。
> - 如果使用不同存储或者其他授权方式，请参考备份存储来进行参数调整。

使用 `br backup full` 可以进行一次快照备份。该命令的详细使用帮助可以通过执行 `br backup full --help` 查看。

```
tiup br backup full --pd "${PD_IP}:2379" \
    --backupts '2022-09-08 13:30:00' \
    --storage "s3://backup-101/snapshot-202209081330?access-key=${access-key}&secret-access-key=$
        ↪ {secret-access-key}" \
    --ratelimit 128 \
```

以上命令中：

- `--backupts`：快照对应的物理时间点，格式可以是TSO 或者时间戳，例如 400036290571534337 或者 `2018-05-11 01:42:23`。如果该快照的数据被垃圾回收 (GC) 了，那么 `br backup` 命令会报错并退出。如果你没有指定该参数，那么 br 会选取备份开始的时间点所对应的快照。
- `--storage`：数据备份到的存储地址。快照备份支持以 Amazon S3、 Google Cloud Storage、 Azure Blob Storage 为备份存储，以上命令以 Amazon S3 为示例。详细存储地址格式请参考备份存储 URI 配置。
- `--ratelimit`：每个 TiKV 备份数据的速度上限，单位为 MiB/s。

在快照备份过程中，终端会显示备份进度条。在备份完成后，会输出备份耗时、速度、备份数据大小等信息。

```
Full Backup <---------------------------------------------------------------------------------->
    ↪ 100.00%
Checksum <------------------------------------------------------------------------------------>
    ↪ 100.00%
*** ["Full Backup success summary"] *** [backup-checksum=3.597416ms] [backup-fast-checksum
    ↪ =2.36975ms] *** [total-take=4.715509333s] [BackupTS=435844546560000000] [total-kv=1131] [
    ↪ total-kv-size=250kB] [average-speed=53.02kB/s] [backup-data-size(after-compressed)=71.33
    ↪ kB] [Size=71330]
```

8.3.3.2.2 查询快照备份的时间点信息

出于管理备份数的需要，如果你需要查看某个快照备份对应的快照物理时间点，可以执行下面的命令：

```
tiup br validate decode --field="end-version" \
--storage "s3://backup-101/snapshot-202209081330?access-key=${access-key}&secret-access-key=${
    ↪ secret-access-key}" | tail -n1
```

结果输出如下，对应物理时间 `2022-09-08 13:30:00 +0800 CST`：

```
435844546560000000
```

8.3.3.2.3 恢复快照备份数据

如果你需要恢复备份的快照数据，则可以使用 `br restore full`。该命令的详细使用帮助可以通过执行 `br restore full --help` 查看。

将上文备份的快照数据恢复到目标集群：

```
tiup br restore full --pd "${PD_IP}:2379" \
--storage "s3://backup-101/snapshot-202209081330?access-key=${access-key}&secret-access-key=${
    ↪ secret-access-key}"
```

在恢复快照备份数据过程中，终端会显示恢复进度条。在完成恢复后，会输出恢复耗时、速度、恢复数据大小等信息。

```
Full Restore <---------------------------------------------------------------------------->
    ↪ 100.00%
*** ["Full Restore success summary"] *** [total-take=4.344617542s] [total-kv=5] [total-kv-size
    ↪ =327B] [average-speed=75.27B/s] [restore-data-size(after-compressed)=4.813kB] [Size=4813]
    ↪  [BackupTS=435844901803917314]
```

恢复备份数据中指定库表的数据

br 命令行工具支持只恢复备份数据中指定库、表的部分数据，该功能用于在恢复过程中过滤不需要的数据。

恢复单个数据库的数据

要将备份数据中的某个数据库恢复到集群中，可以使用 `br restore db` 命令。以下示例只恢复 `test` 库的数据：

```
tiup br restore db --pd "${PD_IP}:2379" \
--db "test" \
--storage "s3://backup-101/snapshot-202209081330?access-key=${access-key}&secret-access-key=${
    ↪ secret-access-key}"
```

以上命令中 `--db` 选项指定了需要恢复的数据库名。

恢复单张表的数据

要将备份数据中的某张数据表恢复到集群中，可以使用 `br restore table` 命令。以下示例只恢复 `test.usertable` 表的数据：

```
tiup br restore table --pd "${PD_IP}:2379" \
--db "test" \
--table "usertable" \
--storage "s3://backup-101/snapshot-202209081330?access-key=${access-key}&secret-access-key=${
    ↪ secret-access-key}"
```

以上命令中 `--db` 选项指定了需要恢复的数据库名，`--table` 选项指定了需要恢复的表名。

使用表库过滤功能恢复部分数据

要通过复杂的过滤条件恢复多个表，可以使用 `br restore full` 命令，并用 `--filter` 或 `-f` 指定表库过滤的条件。以下示例恢复符合 `db*.tbl*` 条件的表的数据：

```
tiup br restore full --pd "${PD_IP}:2379" \
--filter 'db*.tbl*' \
--storage "s3://backup-101/snapshot-202209081330?access-key=${access-key}&secret-access-key=${
    ↪ secret-access-key}"
```

恢复 mysql 数据库下的表

自 br v5.1.0 开始，快照备份会备份 mysql schema 下的系统表数据，而不会默认恢复这些数据。自 br v6.2.0 开始，在设置 --with-sys-table 下，恢复数据时将同时恢复部分系统表相关数据。

可恢复的部分系统表：

```
+----------------------------------+
| mysql.columns_priv               |
| mysql.db                         |
| mysql.default_roles              |
| mysql.global_grants              |
| mysql.global_priv                |
| mysql.role_edges                 |
| mysql.tables_priv                |
| mysql.user                       |
+----------------------------------+
```

不能恢复以下系统表：

- 统计信息表 (mysql.stat_*)
- 系统变量表 (mysql.tidb、mysql.global_variables)
- 其他系统表

当恢复系统权限相关数据的时候，请注意：

- BR 不会恢复 user 为 cloud_admin 并且 host 为 '%' 的用户数据，该用户是 TiDB Cloud 预留用户。请不要在你的环境中创建 cloud_admin 的用户或者角色，因为依赖 cloud_admin 的用户的权限将不能被完整恢复。
- 在恢复数据前 BR 会检查目标集群的系统表是否跟备份数据中的系统表兼容。这里的兼容是指满足以下所有条件：
  - 目标集群需要存在备份中的系统权限表。
  - 目标集群系统权限表列数需要与备份数据中一致，列的顺序可以有差异。
  - 目标集群系统权限表的列需要与备份数据兼容。如果为带长度类型（包括整型、字符串等类型），前者长度需大于或等于后者，如果为 ENUM 类型，则应该为后者超集。

8.3.3.2.4 性能与影响

快照备份的性能与影响

TiDB 备份功能对集群性能（事务延迟和 QPS）有一定的影响，但是可以通过调整备份的线程数backup.num-threads，以及增加集群配置，来降低备份对集群性能的影响。

为了更加具体说明备份对集群的影响，下面列举了多次快照备份测试结论来说明影响的范围：

- （使用 5.3 及之前版本）在默认配置下，单 TiKV 存储节点上备份线程数量是节点 CPU 总数量的 75% 时，QPS 会下降到备份之前的 35% 左右。
- （使用 5.4 及以后版本）单 TiKV 存储节点上备份的线程数量不大于 8、集群总 CPU 利用率不超过 80% 时，备份任务对集群（无论读写负载）影响最大在 20% 左右。

- （使用 5.4 及以后版本）单 TiKV 存储节点上备份的线程数量不大于 8、集群总 CPU 利用率不超过 75% 时，备份任务对集群（无论读写负载）影响最大在 10% 左右。
- （使用 5.4 及以后版本）单 TiKV 存储节点上备份的线程数量不大于 8、集群总 CPU 利用率不超过 60% 时，备份任务对集群（无论读写负载）几乎没有影响。

你可以通过如下方案手动控制备份对集群性能带来的影响。但是，这两种方案在减少备份对集群的影响的同时，也会降低备份任务的速度。

- 使用 `--ratelimit` 参数对备份任务进行限速。请注意，这个参数限制的是把备份文件存储到外部存储的速度。计算备份文件的大小时，请以备份日志中的 `backup data size(after compressed)` 为准。
- 调节 TiKV 配置项`backup.num-threads`，限制备份任务使用的工作线程数量。内部测试数据表明，当备份的线程数量不大于 8、集群总 CPU 利用率不超过 60% 时，备份任务对集群（无论读写负载）几乎没有影响。

通过限制备份的线程数量可以降低备份对集群性能的影响，但是这会影响到备份的性能，以上的多次备份测试结果显示，单 TiKV 存储节点上备份速度和备份线程数量呈正比。在线程数量较少的时候，备份速度约为 20 MiB/线程数。例如，单 TiKV 节点 5 个备份线程可达到 100 MiB/s 的备份速度。

快照恢复的性能与影响

- TiDB 恢复的时候会尽可能打满 TiKV CPU、磁盘 IO、网络带宽等资源，所以推荐在空的集群上执行备份数据的恢复，避免对正在运行的业务产生影响。
- 备份数据的恢复速度与集群配置、部署、运行的业务都有比较大的关系。在内部多场景仿真测试中，单 TiKV 存储节点上备份数据恢复速度能够达到 100 MiB/s。在不同用户场景下，快照恢复的性能和影响应以实际测试结论为准。

8.3.3.2.5 探索更多

- TiDB 集群备份与恢复实践示例
- br 命令行手册
- 快照备份与恢复架构设计

8.3.3.3 TiDB 日志备份与 PITR 使用指南

全量备份包含集群某个时间点的全量数据，但是不包含其他时间点的更新数据，而 TiDB 日志备份能够将业务写入 TiDB 的数据记录及时备份到指定存储中。如果你需要灵活的选择恢复的时间点，即实现 Point-in-time recovery (PITR)，可以开启日志备份，并定期执行快照（全量）备份。

使用 br 命令行工具备份与恢复数据前，请先安装 br 命令行工具。

8.3.3.3.1 对集群进行备份

开启日志备份

注意：

- 以下场景采用 Amazon S3 Access key 和 Secret key 授权方式来进行模拟。如果使用 IAM Role 授权，需要设置 `--send-credentials-to-tikv` 为 `false`。
- 如果使用不同存储或者其他授权方式，请参考备份存储来进行参数调整。

执行 `br log start` 命令启动日志备份任务，一个集群只能启动一个日志备份任务。

```
tiup br log start --task-name=pitr --pd "${PD_IP}:2379" \
--storage 's3://backup-101/logbackup?access-key=${access-key}&secret-access-key=${secret-access-
    ↪ key}"'
```

日志备份任务启动后，会在 TiDB 集群后台持续地运行，直到你手动将其暂停。在这过程中，TiDB 变更数据将以小批量的形式定期备份到指定存储中。如果你需要查询日志备份任务当前状态，执行如下命令：

```
tiup br log status --task-name=pitr --pd "${PD_IP}:2379"
```

日志备份任务状态输出如下：

```
● Total 1 Tasks.
> #1 <
    name: pitr
    status: ● NORMAL
    start: 2022-05-13 11:09:40.7 +0800
      end: 2035-01-01 00:00:00 +0800
    storage: s3://backup-101/log-backup
    speed(est.): 0.00 ops/s
checkpoint[global]: 2022-05-13 11:31:47.2 +0800; gap=4m53s
```

定期执行全量备份

快照备份功能可作为全量备份的方法，运行 `br backup full` 命令可以按照固定的周期（比如 2 天）进行全量备份。

```
tiup br backup full --pd "${PD_IP}:2379" \
--storage 's3://backup-101/snapshot-${date}?access-key=${access-key}&secret-access-key=${secret-
    ↪ access-key}"'
```

8.3.3.3.2 进行 PITR

如果你想恢复到备份保留期内的任意时间点，可以使用 `br restore point` 命令。执行该命令时，你需要指定要恢复的时间点、恢复时间点之前最近的快照备份以及日志备份数据。br 命令行工具会自动判断和读取恢复需要的数据，然后将这些数据依次恢复到指定的集群。

```
br restore point --pd "${PD_IP}:2379" \
--storage='s3://backup-101/logbackup?access-key=${access-key}&secret-access-key=${secret-access-
    ↪ key}"' \
--full-backup-storage='s3://backup-101/snapshot-${date}?access-key=${access-key}&secret-access-
    ↪ key=${secret-access-key}"' \
--restored-ts '2022-05-15 18:00:00+0800'
```

恢复期间，可通过终端中的进度条查看进度，如下。恢复分为两个阶段：全量恢复 (Full Restore) 和日志恢复 (Restore Meta Files 和 Restore KV Files)。每个阶段完成恢复后，br 命令行工具都会输出恢复耗时和恢复数据大小等信息。

```
Full Restore
    ↪ <-------------------------------------------------------------------------------------------------------------
    ↪  100.00%
*** ["Full Restore success summary"] ****** [total-take=xxx.xxxs] [restore-data-size(after-
    ↪ compressed)=xxx.xxx] [Size=xxxx] [BackupTS={TS}] [total-kv=xxx] [total-kv-size=xxx] [
    ↪ average-speed=xxx]
Restore Meta Files
    ↪ <-------------------------------------------------------------------------------------------------------------
    ↪  100.00%
Restore KV Files
    ↪ <-------------------------------------------------------------------------------------------------------------
    ↪  100.00%
*** ["restore log success summary"] [total-take=xxx.xx] [restore-from={TS}] [restore-to={TS}] [
    ↪ total-kv-count=xxx] [total-size=xxx]
```

8.3.3.3.3 清理过期的日志备份数据

如使用指南概览所述：

- 进行 PITR 不仅需要恢复时间点之前的全量备份，还需要全量备份和恢复时间点之间的日志备份，因此，对于超过备份保留期的日志备份，应执行 `br log truncate` 命令删除指定时间点之前的备份。建议只清理全量快照之前的日志备份。

你可以按照以下步骤清理超过备份保留期的备份数据：

1. 查找备份保留期之外的最近一次全量备份。
2. 使用 `validate` 指令获取该备份对应的时间点。假如需要清理 2022/09/01 之前的备份数据，则应查找该日期之前的最近一次全量备份，且保证它不会被清理。

```
FULL_BACKUP_TS=`tiup br validate decode --field="end-version" --storage "s3://backup-101/
    ↪ snapshot-${date}?access-key=${access-key}&secret-access-key=${secret-access-key}"|
    ↪ tail -n1`
```

3. 清理该快照备份 FULL_BACKUP_TS 之前的日志备份数据。

```
tiup br log truncate --until=${FULL_BACKUP_TS} --storage='s3://backup-101/logbackup?access-
    ↪ key=${access-key}&secret-access-key=${secret-access-key}"'
```

4. 清理该快照备份 FULL_BACKUP_TS 之前的快照备份数据。

```
rm -rf s3://backup-101/snapshot-${date}
```

8.3.3.3.4 PITR 的性能与影响

能力指标

- PITR 恢复速度，平均到单台 TiKV 节点：全量恢复为 280 GB/h，日志恢复为 30 GB/h
- 使用 `br log truncate` 清理过期的日志备份数据速度为 600 GB/h

> 注意：
>
> 以上功能指标是根据下述两个场景测试得出的结论，如有出入，建议以实际测试结果为准：
>
> - 全量恢复速度 = 全量恢复集群数据量 / (时间 * TiKV 数量)
> - 日志恢复速度 = 日志恢复总量 / (时间 * TiKV 数量)

测试场景 1 (TiDB Cloud 上部署)

- TiKV 节点 (8 core，16 GB 内存) 数量：21
- Region 数量：183,000
- 集群新增日志数据：10 GB/h
- 写入 (INSERT/UPDATE/DELETE) QPS：10,000

测试场景 2 (本地部署)

- TiKV 节点 (8 core，64 GB 内存) 数量：6
- Region 数量：50,000
- 集群新增日志数据：10 GB/h
- 写入 (INSERT/UPDATE/DELETE) QPS：10,000

8.3.3.3.5 探索更多

- TiDB 集群备份与恢复实践示例
- br 命令行手册
- 日志备份与 PITR 架构设计

8.3.3.4 TiDB 备份与恢复实践示例

TiDB 快照备份与恢复使用指南及TiDB 日志备份与 PITR 使用指南系统介绍了 TiDB 提供的备份与恢复方案，即快照（全量）备份与恢复、日志备份和恢复到指定时间点 (Point-in-time recovery, PITR)。本文档将基于具体的使用场景，介绍如何快速上手使用 TiDB 的备份与恢复方案。

介绍具体操作前，设想有如下使用场景，你在 AWS 部署了一套 TiDB 生产集群，业务团队提出如下需求：

- 及时备份用户数据变更，在数据库遭遇异常情况时，能够以最小的数据丢失代价（容忍异常前几分钟内的用户数据丢失）快速地恢复业务。
- 每个月不定期进行业务审计。接收到审计请求后，提供一个数据库来查询审计要求的一个月内某个时间点的数据。

通过 TiDB 提供的 PITR 功能，你可以满足业务团队的需求。

8.3.3.4.1 部署 TiDB 集群和 br 命令行工具

使用 PITR 功能，需要部署 v6.2.0 或以上版本的 TiDB 集群，并且更新 br 命令行工具到与 TiDB 集群相同的版本，本文假设使用的是 v7.0.0 版本。

下表介绍了在 TiDB 集群中使用日志备份功能的推荐配置。

组件	CPU	内存	硬盘类型	AWS 机型	实例数量
TiDB	8 核 +	16 GB+	SAS	c5.2xlarge	2
PD	8 核 +	16 GB+	SSD	c5.2xlarge	3
TiKV	8 核 +	32 GB+	SSD	m5.2xlarge	3
br cli	8 核 +	16 GB+	SAS	c5.2xlarge	1
监控	8 核 +	16 GB+	SAS	c5.2xlarge	1

> 注意：
>
> - br 命令行工具执行备份恢复功能需要访问 PD 和 TiKV，请确保 br 命令行工具与所有 PD 和 TiKV 连接正常。
> - br 命令行工具与 PD 所在服务器时区需要相同。

使用 TiUP 部署或升级 TiDB 集群：

- 如果没有部署 TiDB 集群，请部署 TiDB 集群。
- 如果已经部署的 TiDB 集群版本低于 v6.2.0，请升级 TiDB 集群。

使用 TiUP 安装或升级 br 命令行工具：

- 安装：

```
tiup install br:v7.0.0
```

- 升级：

```
tiup update br:v7.0.0
```

8.3.3.4.2 配置备份存储 (Amazon S3)

在开始备份任务之前需要准备好备份存储，包括：

1. 准备用于存放备份数据的 S3 bucket 和目录；
2. 配置访问 S3 中备份目录的权限；
3. 规划备份数据保存的目录结构。

配置备份存储的步骤如下：

1. 在 S3 创建用于保存备份数据的目录 `s3://tidb-pitr-bucket/backup-data`。

    1. 创建 bucket。你也可以选择已有的 S3 bucket 来保存备份数据。如果没有可用的 bucket，可以参照 AWS 官方文档创建一个 S3 Bucket。本文使用的 bucket 名为 `tidb-pitr-bucket`。
    2. 创建备份数据总目录。在上一步创建的 bucket（例如 `tidb-pitr-bucket`）下创建目录 `backup-data`，参考 AWS 官方文档。

2. 配置 br 命令行工具和 TiKV 访问 S3 中的备份目录的权限。本文推荐使用最安全的 IAM 访问方式，配置过程可以参考控制存储桶访问。权限要求如下：

    - 备份集群的 TiKV 和 br 命令行工具需要的 `s3://tidb-pitr-bucket/backup-data` 权限：`s3:ListBucket`、`s3:PutObject` 和 `s3:AbortMultipartUpload`。
    - 恢复集群的 TiKV 和 br 命令行工具需要 `s3://tidb-pitr-bucket/backup-data` 的最小权限：`s3:ListBucket` 和 `s3:GetObject`。

3. 规划备份数据保存的目录结构，以及快照（全量）备份和日志备份的目录。

    - 所有快照备份保存在 `s3://tidb-pitr-bucket/backup-data/snapshot-${date}` 目录下，`${date}` 为快照备份开始的时间点，如在 2022/05/12 00:01:30 开始的快照备份保存为 `s3://tidb-pitr-bucket/backup-data/snapshot-20220512000130`。
    - 日志备份保存在 `s3://tidb-pitr-bucket/backup-data/log-backup/` 目录下。

8.3.3.4.3 确定备份策略

为了满足业务最小数据丢失、快速恢复、一个月内任意时间点审计需求，你可以制定如下备份策略：

- 运行日志备份，持续不断备份数据库数据变更；
- 每隔两天在零点左右进行一次快照备份；
- 保存 30 天内的快照备份和日志备份数据，清理超过 30 天的备份数据。

8.3.3.4.4 执行日志备份

启动日志备份任务后，日志备份进程会在 TiKV 集群运行，持续不断将数据库变更数据备份到 S3 中。日志备份任务启动命令：

```
tiup br log start --task-name=pitr --pd="${PD_IP}:2379" \
--storage='s3://tidb-pitr-bucket/backup-data/log-backup'
```

启动日志备份任务后，可以查询日志备份任务状态：

```
tiup br log status --task-name=pitr --pd="${PD_IP}:2379"

● Total 1 Tasks.
> #1 <
    name: pitr
    status: ● NORMAL
    start: 2022-05-13 11:09:40.7 +0800
      end: 2035-01-01 00:00:00 +0800
    storage: s3://tidb-pitr-bucket/backup-data/log-backup
    speed(est.): 0.00 ops/s
checkpoint[global]: 2022-05-13 11:31:47.2 +0800; gap=4m53s
```

8.3.3.4.5 执行快照备份

通过自动化运维工具（如 crontab）设置定期的快照备份任务，例如：每隔两天在零点左右进行一次快照（全量）备份。下面是两次备份的示例：

- 在 2022/05/14 00:00:00 执行一次快照备份：

```
tiup br backup full --pd="${PD_IP}:2379" \
--storage='s3://tidb-pitr-bucket/backup-data/snapshot-20220514000000' \
--backupts='2022/05/14 00:00:00'
```

- 在 2022/05/16 00:00:00 执行一次快照备份：

```
tiup br backup full --pd="${PD_IP}:2379" \
--storage='s3://tidb-pitr-bucket/backup-data/snapshot-20220516000000' \
--backupts='2022/05/16 00:00:00'
```

8.3.3.4.6 执行 PITR

假设你接到需求，要准备一个集群查询 2022/05/15 18:00:00 时间点的用户数据。此时，你可以制定 PITR 方案，恢复 2022/05/14 的快照备份和该快照到 2022/05/15 18:00:00 之间的日志备份数据，从而收集到目标数据。执行命令如下：

```
tiup br restore point --pd="${PD_IP}:2379" \
--storage='s3://tidb-pitr-bucket/backup-data/log-backup' \
```

```
--full-backup-storage='s3://tidb-pitr-bucket/backup-data/snapshot-20220514000000' \
--restored-ts '2022-05-15 18:00:00+0800'

Full Restore
    ↪ <-----------------------------------------------------------------------------------------------------
    ↪  100.00%
[2022/05/29 18:15:39.132 +08:00] [INFO] [collector.go:69] ["Full Restore success summary"] [total
    ↪ -ranges=12] [ranges-succeed=xxx] [ranges-failed=0] [split-region=xxx.xxxµs] [restore-
    ↪ ranges=xxx] [total-take=xxx.xxxs] [restore-data-size(after-compressed)=xxx.xxx] [Size=
    ↪ xxxx] [BackupTS={TS}] [total-kv=xxx] [total-kv-size=xxx] [average-speed=xxx]
Restore Meta Files
    ↪ <-----------------------------------------------------------------------------------------------------
    ↪  100.00%
Restore KV Files
    ↪ <-----------------------------------------------------------------------------------------------------
    ↪  100.00%
[2022/05/29 18:15:39.325 +08:00] [INFO] [collector.go:69] ["restore log success summary"] [total-
    ↪ take=xxx.xx] [restore-from={TS}] [restore-to={TS}] [total-kv-count=xxx] [total-size=xxx]
```

8.3.3.4.7 清理过期备份数据

通过自动化运维工具（如 crontab) 每两天定期清理过期备份数据的任务。

下面是执行过期备份数据清理任务：

- 删除早于 2022/05/14 00:00:00 的快照备份

```
shell rm s3://tidb-pitr-bucket/backup-data/snapshot-20220514000000
```

- 删除早于 2022/05/14 00:00:00 的日志备份数据

```
shell tiup br log truncate --until='2022-05-14 00:00:00 +0800' --storage='s3://tidb-pitr-bucket/
↪ backup-data/log-backup'
```

8.3.3.4.8 探索更多

- 备份存储
- 快照备份与恢复命令手册
- 日志备份与 PITR 命令手册

8.3.3.5 备份存储

TiDB 支持 Amazon S3、 Google Cloud Storage (GCS)、 Azure Blob Storage 和 NFS 作为备份恢复的存储。具体来说，可以在 br 的 `--storage` 或 `-s` 选项中指定备份存储的 URI。本文介绍不同外部存储服务中URI 的定义格式、存储过程中的鉴权方案以及存储服务端加密。

8.3.3.5.1 BR 向 TiKV 发送凭证

命令行参数	描述	默认值
--send-credentials-to-tikv	是否将 BR 获取到的权限凭证发送给 TiKV。	true

在默认情况下，使用 Amazon S3、 Google Cloud Storage (GCS)、 Azure Blob Storage 存储时，BR 会将凭证发送到每个 TiKV 节点，以减少设置的复杂性。该操作由参数 --send-credentials-to-tikv（或简写为 -c）控制。

但是，这个操作不适合云端环境，如果采用了 IAM Role 方式授权，那么每个节点都有自己的角色和权限。在这种情况下，你需要设置 --send-credentials-to-tikv=false（或简写为 -c=0）来禁止发送凭证：

```
./br backup full -c=0 -u pd-service:2379 --storage 's3://bucket-name/prefix'
```

使用 SQL 进行备份恢复时，可加上 SEND_CREDENTIALS_TO_TIKV = FALSE 选项：

```
BACKUP DATABASE * TO 's3://bucket-name/prefix' SEND_CREDENTIALS_TO_TIKV = FALSE;
```

8.3.3.5.2 URI 格式

格式说明

本部分介绍存储服务的 URI 格式：

```
[scheme]://[host]/[path]?[parameters]
```

- scheme：s3
- host：bucket name
- parameters：
  - access-key：访问密钥
  - secret-access-key：秘密访问密钥
  - session-token：会话令牌
  - use-accelerate-endpoint：是否在 Amazon S3 上使用加速端点，默认为 false
  - endpoint：Amazon S3 兼容服务自定义端点的 URL，例如 <https://s3.example.com/>
  - force-path-style：使用路径类型 (path-style)，而不是虚拟托管类型 (virtual-hosted-style)，默认为 true
  - storage-class：上传对象的存储类别，例如 STANDARD、 STANDARD_IA
  - sse：加密上传的服务端加密算法，可以设置为空、 AES256 或 aws:kms
  - sse-kms-key-id：如果 sse 设置为 aws:kms，则使用该参数指定 KMS ID
  - acl：上传对象的标准 ACL (Canned ACL)，例如 private、 authenticated-read
  - role-arn：当需要使用特定的 IAM 角色来访问第三方 Amazon S3 的数据时，使用这个参数来指定 IAM 角色的对应 Amazon Resource Name (ARN)（例如 arn:aws:iam::888888888888:role/my-role）。关于使用 IAM 角色访问第三方 Amazon S3 数据的场景，请参考 AWS 相关文档介绍。
  - external-id：当需要使用特定的 IAM 角色来访问第三方 Amazon S3 的数据时，可能需要同时提供正确的外部 ID 来确保用户有权限代入该 IAM 角色。这个参数用来指定对应的外部 ID，确保成功代入 IAM 角色。外部 ID 可以是任意字符串，并且不是必须的，一般由控制 Amazon S3 数据访问的第三方来指定。如果第三方对于 IAM 角色没有要求指定外部 ID，则可以不需要提供该参数也能顺利代入对应的 IAM 角色，从而访问对应的 Amazon S3 数据。

- scheme：gcs 或 gs
- host：bucket name
- parameters：
  - credentials-file：迁移工具节点上凭证 JSON 文件的路径
  - storage-class：上传对象的存储类别，例如 STANDARD 或 COLDLINE
  - predefined-acl：上传对象的预定义 ACL，例如 private 或 project-private
- scheme：azure 或 azblob
- host：container name
- parameters：
  - account-name：存储账户名
  - account-key：访问密钥
  - access-tier：上传对象的存储类别，例如 Hot、Cool、Archive，默认为 Hot

URI 示例

本部分示例以 host（上表中 bucket name、container name）为 external 为例进行介绍。

备份快照数据到 Amazon S3

```
./br backup full -u "${PD_IP}:2379" \
--storage "s3://external/backup-20220915?access-key=${access-key}&secret-access-key=${secret-
    ↪ access-key}"
```

从 Amazon S3 恢复快照备份数据

```
./br restore full -u "${PD_IP}:2379" \
--storage "s3://external/backup-20220915?access-key=${access-key}&secret-access-key=${secret-
    ↪ access-key}"
```

备份快照数据到 GCS

```
./br backup full --pd "${PD_IP}:2379" \
--storage "gcs://external/backup-20220915?credentials-file=${credentials-file-path}"
```

从 GCS 恢复快照备份数据

```
./br restore full --pd "${PD_IP}:2379" \
--storage "gcs://external/backup-20220915?credentials-file=${credentials-file-path}"
```

备份快照数据到 Azure Blob Storage

```
./br backup full -u "${PD_IP}:2379" \
--storage "azure://external/backup-20220915?account-name=${account-name}&account-key=${account-
    ↪ key}"
```

从 Azure Blob Storage 恢复快照备份数据中 test 数据库

```
./br restore db --db test -u "${PD_IP}:2379" \
--storage "azure://external/backup-20220915account-name=${account-name}&account-key=${account-key
    ↪ }"
```

8.3.3.5.3 鉴权

将数据存储到云服务存储系统时，根据云服务供应商的不同，需要设置不同的鉴权参数。本部分介绍使用 Amazon S3、 GCS 及 Azure Blob Storage 时所用存储服务的鉴权方式以及如何配置访问相应存储服务的账户。

在备份之前，需要为 br 命令行工具访问 Amazon S3 中的备份目录设置相应的访问权限：

- 备份时 TiKV 和 br 命令行工具需要的访问备份数据目录的最小权限：s3:ListBucket、 s3:PutObject 和 s3:AbortMultipartUpload。
- 恢复时 TiKV 和 br 命令行工具需要的访问备份数据目录的最小权限：s3:ListBucket 和 s3:GetObject。

如果你还没有创建备份数据保存目录，可以参考创建存储桶在指定的区域中创建一个 S3 存储桶。如果需要使用文件夹，可以参考使用文件夹在 Amazon S3 控制台中组织对象在存储桶中创建一个文件夹。

配置访问 Amazon S3 的账户可以通过以下两种方式：

- 方式一：指定访问密钥

  如果指定访问密钥和秘密访问密钥，将按照指定的访问密钥和秘密访问密钥进行鉴权。除了在 URI 中指定密钥外，还支持以下方式：

  - br 命令行工具读取 $AWS_ACCESS_KEY_ID 和 $AWS_SECRET_ACCESS_KEY 环境变量
  - br 命令行工具读取 $AWS_ACCESS_KEY 和 $AWS_SECRET_KEY 环境变量
  - br 命令行工具读取共享凭证文件，路径由 $AWS_SHARED_CREDENTIALS_FILE 环境变量指定
  - br 命令行工具读取共享凭证文件，路径为 ~/.aws/credentials

- 方式二：基于 IAM Role 进行访问

  为运行 TiKV 和 br 命令行工具的 EC2 实例关联一个配置了访问 S3 访问权限的 IAM role。正确设置后，br 命令行工具可以直接访问对应的 S3 中的备份目录，而不需要额外的设置。

  ```
  br backup full --pd "${PD_IP}:2379" \
  --storage "s3://${host}/${path}"
  ```

配置访问 GCS 的账户可以通过指定访问密钥的方式。如果指定了 credentials-file 参数，将按照指定的 credentials-file 进行鉴权。除了在 URI 中指定密钥文件外，还支持以下方式：

- br 命令行工具读取位于 $GOOGLE_APPLICATION_CREDENTIALS 环境变量所指定路径的文件内容
- br 命令行工具读取位于 ~/.config/gcloud/application_default_credentials.json 的文件内容
- 在 GCE 或 GAE 中运行时，从元数据服务器中获取的凭证

- 方式一：指定访问密钥

  在 URI 配置 account-name 和 account-key，则使用该参数指定的密钥。除了在 URI 中指定密钥文件外，还支持 br 命令行工具读取 $AZURE_STORAGE_KEY 的方式。

- 方式二：使用 Azure AD 备份恢复

  在 br 命令行工具运行环境配置环境变量 $AZURE_CLIENT_ID、$AZURE_TENANT_ID 和 $AZURE_CLIENT_SECRET。

  - 当集群使用 TiUP 启动时，TiKV 会使用 systemd 服务。以下示例介绍如何为 TiKV 配置上述三个环境变量：

    > 注意：
    > 该流程在第 3 步中需要重启 TiKV。如果你的集群不适合重启，请使用指定访问密钥的方式进行备份恢复。

    1. 假设该节点上 TiKV 端口为 24000，即 systemd 服务名为 tikv-24000：

    ```
    systemctl edit tikv-24000
    ```

    2. 编辑三个环境变量的信息：

    ```
    [Service]
    Environment="AZURE_CLIENT_ID=aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa"
    Environment="AZURE_TENANT_ID=aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa"
    Environment="AZURE_CLIENT_SECRET=aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
    ```

    3. 重新加载配置并重启 TiKV：

    ```
    systemctl daemon-reload
    systemctl restart tikv-24000
    ```

  - 为命令行启动的 TiKV 和 br 命令行工具配置 Azure AD 的信息，只需要确定运行环境中存在 $AZURE_CLIENT_ID、$AZURE_TENANT_ID 和 $AZURE_CLIENT_SECRET。通过运行下列命令行，可以确认 br 命令行工具和 TiKV 运行环境中是否存在这三个环境变量：

  ```
  echo $AZURE_CLIENT_ID
  echo $AZURE_TENANT_ID
  echo $AZURE_CLIENT_SECRET
  ```

  - 使用 br 命令行工具将数据备份至 Azure Blob Storage：

  ```
  ./br backup full -u "${PD_IP}:2379" \
  --storage "azure://external/backup-20220915?account-name=${account-name}"
  ```

8.3.3.5.4 存储服务端加密

Amazon S3 存储服务端加密备份数据

TiDB 备份恢复功能支持对备份到 Amazon S3 的数据进行 S3 服务端加密 (SSE)。S3 服务端加密也支持使用用户自行创建的 AWS KMS 密钥，详细信息请参考BR S3 服务端加密。

8.3.3.5.5 存储服务其他功能支持

TiDB 备份恢复功能从 v6.3.0 支持 AWS S3 Object Lock 功能。你可以在 AWS 中开启 S3 Object Lock 功能来防止备份数据写入后被修改或者删除。

8.3.4 br cli 命令手册

8.3.4.1 br 命令行手册

本文介绍 br 命令的定义、组成、常用选项，以及快照备份与恢复、日志备份与 PITR (Point-in-time recovery) 功能使用的命令。

8.3.4.1.1 br 命令行描述

br 命令是由子命令、选项和参数组成的。子命令即不带 - 或者 -- 的字符。选项即以 - 或者 -- 开头的字符。参数即子命令或选项字符后紧跟的、并传递给命令和选项的字符。

以下是一条完整的 br 命令行：

```
br backup full --pd "${PD_IP}:2379" \
--storage "s3://backup-data/snapshot-202209081330/"
```

上面命令行中各部分的解释如下：

- backup：br 的子命令。
- full：br backup 的子命令。
- -s 或 --storage：备份数据的存储地址选项。"s3://backup-data/snapshot-202209081330/" 是 -s 的参数值。
- --pd：PD 访问地址选项。"${PD_IP}:2379" 是 --pd 的参数值。

命令和子命令

br 由多层命令组成。目前，br 包含的主要命令有：

- br backup：用于备份 TiDB 集群的全量数据。
- br log：用于启动和管理日志备份任务。
- br restore：用于恢复备份数据到 TiDB 集群。

br backup 和 br restore 还包含这些子命令：

- full：用于备份或恢复整个备份数据。
- db：用于备份或恢复集群中的指定数据库。
- table：用于备份或恢复集群指定数据库中的单张表。

常用选项

- --pd：PD 访问地址选项，例如 "${PD_IP}:2379"。

- `-s` 或 `--storage`：备份数据的存储地址选项。TiDB 备份恢复支持以 Amazon S3、 Google Cloud Storage (GCS)、Azure Blob Storage 及 NFS 为备份存储。详细参考备份存储 URI 配置。
- `--ca`：指定 PEM 格式的受信任 CA 的证书文件路径。
- `--cert`：指定 PEM 格式的 SSL 证书文件路径。
- `--key`：指定 PEM 格式的 SSL 证书密钥文件路径。
- `--status-addr`：向 Prometheus 提供统计数据的监听地址。

8.3.4.1.2 全量备份命令行

使用 `br backup` 命令来备份集群全量数据。可选择添加 `full` 或 `table` 子命令来指定备份的范围：全部集群数据 (`full`) 或单张表的数据 (`table`)。

- 备份集群快照数据
- 备份单个数据库的数据
- 备份单张表的数据
- 使用表库过滤功能备份多张表的数据
- 加密快照备份数据

8.3.4.1.3 日志备份命令行

使用 `br log` 命令来开启和管理日志备份任务。

- 启动日志备份
- 查询日志备份状态
- 暂停和恢复日志备份任务
- 停止和重启日志备份任务
- 清理日志备份数据
- 查看备份数据元信息

8.3.4.1.4 恢复备份数据命令行

使用 `br restore` 命令来恢复备份数据。可选择添加 `full`、`db` 或 `table` 子命令来指定恢复操作的范围：全部集群数据 (`full`)、某个数据库 (`db`) 或某张数据表 (`table`)。

- Point-in-time recovery
- 恢复快照备份数据
- 恢复单个数据库的快照备份数据
- 恢复单张表的快照备份数据
- 使用表库功能过滤恢复快照数据
- 恢复加密的快照备份数据

8.3.4.2 TiDB 快照备份与恢复命令行手册

本文按备份恢复的场景介绍快照备份和恢复的命令行，包括：

- 备份集群快照

- 备份单个数据库的数据
- 备份单张表的数据
- 使用表库过滤功能备份多张表的数据
- 备份数据加密
- 恢复快照备份数据
- 恢复单个数据库的数据
- 恢复单张表的数据
- 使用表库功能过滤恢复数据
- 恢复加密的快照备份数据

如果你想了解如何进行快照备份与恢复，可以参考以下教程：

- TiDB 快照备份与恢复使用指南
- TiDB 集群备份与恢复实践示例

8.3.4.2.1 备份集群快照

执行 `br backup full` 命令，可以备份 TiDB 最新的或者指定时间点的快照数据。执行 `br backup full --help` 可获取该命令的使用帮助。

```
br backup full \
    --pd "${PD_IP}:2379" \
    --backupts '2022-09-08 13:30:00' \
    --storage "s3://${backup_collection_addr}/snapshot-${date}?access-key=${access-key}&secret-
        ↪ access-key=${secret-access-key}" \
    --ratelimit 128 \
    --log-file backupfull.log
```

以上命令中：

- `--backupts`：快照对应的物理时间点。如果该快照的数据已经被 GC，那么 `br backup` 命令会报错退出；如果没有指定该参数，br 命令行工具会选取备份开始的时间点所对应的快照。
- `--ratelimit`：每个 TiKV 执行备份任务的速度上限（单位 MiB/s）。
- `--log-file`：备份日志写入的目标文件。

> **注意：**
>
> BR 工具已支持自适应 GC，会自动将 backupTS（默认是最新的 PD timestamp）注册到 PD 的 `safePoint`，保证 TiDB 的 GC Safe Point 在备份期间不会向前移动，即可避免手动设置 GC。

备份期间终端会显示进度条，效果如下。当进度条达到 100% 时，表示备份完成。

```
Full Backup <---------/.....................................................> 17.12%.
```

8.3.4.2.2 备份 TiDB 集群指定库表的数据

br 工具支持只备份集群快照和增量数据中指定库或表的局部数据。在快照备份和增量数据备份的基础上，该功能可过滤掉不需要的数据，只备份关键业务的数据。

备份单个数据库的数据

执行 `br backup db` 命令，可备份集群中指定单个数据库的数据。

下面是将数据库 `test` 备份到 Amazon S3 的示例：

```
br backup db \
    --pd "${PD_IP}:2379" \
    --db test \
    --storage "s3://${backup_collection_addr}/snapshot-${date}?access-key=${access-key}&secret-
        ↪ access-key=${secret-access-key}" \
    --ratelimit 128 \
    --log-file backuptable.log
```

db 子命令的选项为 `--db`，用来指定数据库名。其他选项的含义与备份 TiDB 集群快照相同。

备份单张表的数据

执行 `br backup table` 命令，可备份集群中指定单张表的数据。

下面是将表 `test.usertable` 备份到 Amazon S3 的示例：

```
br backup table \
    --pd "${PD_IP}:2379" \
    --db test \
    --table usertable \
    --storage "s3://${backup_collection_addr}/snapshot-${date}?access-key=${access-key}&secret-
        ↪ access-key=${secret-access-key}" \
    --ratelimit 128 \
    --log-file backuptable.log
```

`table` 子命令有 `--db` 和 `--table` 两个选项，分别用来指定数据库名和表名。其他选项的含义与备份 TiDB 集群快照相同。

使用表库过滤功能备份多张表的数据

如果你需要以更复杂的过滤条件来备份多个库或表，执行 `br backup full` 命令，并使用 `--filter` 或 `-f` 来指定表库过滤规则。

下面是将所有符合 `db*.tbl*` 条件的表的数据备份到 Amazon S3 的示例：

```
br backup full \
    --pd "${PD_IP}:2379" \
    --filter 'db*.tbl*' \
    --storage "s3://${backup_collection_addr}/snapshot-${date}?access-key=${access-key}&secret-
        ↪ access-key=${secret-access-key}" \
    --ratelimit 128 \
    --log-file backupfull.log
```

8.3.4.2.3 备份数据加密

> **警告：**
>
> 当前该功能为实验特性，不建议在生产环境中使用。

br 命令行工具支持在备份端，或备份到 Amazon S3 的时候在存储服务端进行备份数据加密，你可以根据自己情况选择其中一种使用。

自 TiDB v5.3.0 起，你可配置下列参数在备份过程中实现数据加密：

- `--crypter.method`：加密算法，支持 `aes128-ctr`、`aes192-ctr` 和 `aes256-ctr` 三种算法，缺省值为 `plaintext`，表示不加密
- `--crypter.key`：加密密钥，十六进制字符串格式，`aes128-ctr` 对应 128 位（16 字节）密钥长度，`aes192-ctr` 为 24 字节，`aes256-ctr` 为 32 字节
- `--crypter.key-file`：密钥文件，可直接将存放密钥的文件路径作为参数传入，此时 `crypter.key` 不需要配置

备份加密的示例如下：

```
br backup full\
    --pd ${PD_IP}:2379 \
    --storage "s3://${backup_collection_addr}/snapshot-${date}?access-key=${access-key}&secret-
        ↪ access-key=${secret-access-key}" \
    --crypter.method aes128-ctr \
    --crypter.key 0123456789abcdef0123456789abcdef
```

> **注意：**
>
> - 密钥丢失，备份的数据将无法恢复到集群中。
> - 加密功能需在 br 工具和 TiDB 集群都不低于 v5.3.0 的版本上使用，且加密备份得到的数据无法在低于 v5.3.0 版本的集群上恢复。

8.3.4.2.4 恢复快照备份数据

执行 `br restore full` 命令，可将集群恢复到快照备份对应的数据状态。

```
br restore full \
    --pd "${PD_IP}:2379" \
    --storage "s3://${backup_collection_addr}/snapshot-${date}?access-key=${access-key}&secret-
        ↪ access-key=${secret-access-key}" \
    --ratelimit 128 \
    --log-file restorefull.log
```

以上命令中，

- `--ratelimit`：每个 TiKV 执行恢复任务的速度上限（单位 MiB/s）
- `--log-file`：备份日志写入的目标文件

恢复期间终端会显示进度条，效果如下。当进度条达到 100% 时，表示恢复完成。在完成恢复后，br 工具为了确保数据安全性，还会校验恢复数据。

```
Full Restore <---------/.............................................> 17.12%.
```

8.3.4.2.5 恢复备份数据中指定库表的数据

br 命令行工具支持只恢复备份数据中指定库/表的局部数据。该功能在恢复过程中过滤掉不需要的数据，可以用于往 TiDB 集群上恢复指定库/表的数据。

恢复单个数据库的数据

执行 `br restore db` 命令，可将单个数据库恢复到对应的状态。

示例：恢复 S3 中的库 `test` 的数据：

```
br restore db \
    --pd "${PD_IP}:2379" \
    --db "test" \
    --ratelimit 128 \
    --storage "s3://${backup_collection_addr}/snapshot-${date}?access-key=${access-key}&secret-
        ↪ access-key=${secret-access-key}" \
    --log-file restore_db.log
```

以上命令中 `--db` 选项指定了需要恢复的数据库名字。其余选项的含义与恢复快照备份数据相同。

> **注意：**
>
> 由于备份数据的元文件 `backupmeta` 记录了数据库名 `--db`，因此只能将数据恢复到同名的数据库，否则无法恢复成功。推荐把备份文件恢复到另一个集群的同名数据库中。

恢复单张表的数据

执行 `br restore table` 命令，可将单张表的数据恢复到对应的状态。

下面是恢复 Amazon S3 中 `test.usertable` 表数据的示例：

```
br restore table \
    --pd "${PD_IP}:2379" \
    --db "test" \
    --table "usertable" \
    --ratelimit 128 \
```

```
    --storage "s3://${backup_collection_addr}/snapshot-${date}?access-key=${access-key}&secret-
        ↪ access-key=${secret-access-key}" \
    --log-file restore_table.log
```

以上命令中 `--table` 选项指定了需要恢复的表名。其余选项的含义与恢复单个数据库相同。

使用表库功能过滤恢复数据

如果你需要用复杂的过滤条件来恢复多个表，执行 `br restore full` 命令，并用 `--filter` 或 `-f` 指定使用表库过滤。

下面是恢复 Amazon S3 中符合 `db*.tbl*` 条件的表的数据的示例：

```
br restore full \
    --pd "${PD_IP}:2379" \
    --filter 'db*.tbl*' \
    --storage "s3://${backup_collection_addr}/snapshot-${date}?access-key=${access-key}&secret-
        ↪ access-key=${secret-access-key}" \
    --log-file restorefull.log
```

8.3.4.2.6 恢复加密的快照备份数据

> **警告：**
>
> 当前该功能为实验特性，不建议在生产环境中使用。

在对数据做加密备份后，恢复操作需传入相应的解密参数，解密算法或密钥不正确则无法恢复，解密参数和加密参数一致即可。解密恢复的示例如下：

```
br restore full\
    --pd "${PD_IP}:2379" \
    --storage "s3://${backup_collection_addr}/snapshot-${date}?access-key=${access-key}&secret-
        ↪ access-key=${secret-access-key}" \
    --crypter.method aes128-ctr \
    --crypter.key 0123456789abcdef0123456789abcdef
```

8.3.4.3 TiDB 日志备份与 PITR 命令行手册

本文介绍 TiDB 日志备份和 PITR (Point-in-time recovery) 命令行。

如果你想了解如何进行日志备份与 PITR，可以参考以下教程：

- TiDB 日志备份与 PITR 使用指南
- TiDB 集群备份与恢复实践示例

8.3.4.3.1 日志备份命令行介绍

你可以执行 `br log` 命令来开启和管理日志备份任务：

```
./br log --help

backup stream log from TiDB/TiKV cluster

Usage:
  br log [command]

Available Commands:
  metadata    get the metadata of log dir
  pause       pause a log backup task
  resume      resume a log backup task
  start       start a log backup task
  status      get status for the log backup task
  stop        stop a log backup task
  truncate    truncate the log data until sometime
```

各个子命令的作用如下：

- `br log start`：启动一个日志备份任务
- `br log status`：查询日志备份任务状态
- `br log pause`：暂停日志备份任务
- `br log resume`：重启暂停的备份任务
- `br log stop`：停止备份任务，并删除任务元信息
- `br log truncate`：从备份存储中清理日志备份数据
- `br log metadata`：查询备份存储中备份数据的元信息

启动日志备份

执行 `br log start` 命令，你可以在备份集群启动一个日志备份任务。该任务在 TiDB 集群持续地运行，及时地将 KV 变更日志保存到备份存储中。

执行 `br log start --help` 命令可获取该子命令使用介绍：

```
./br log start --help
start a log backup task

Usage:
  br log start [flags]

Flags:
  -h, --help                help for start
  --start-ts string         usually equals last full backupTS, used for backup log. Default value
      ↪ is current ts. support TSO or datetime, e.g. '400036290571534337' or '2018-05-11
      ↪ 01:42:23+0800'.
```

```
  --task-name string       The task name for the backup log task.

Global Flags:
 --ca string                  CA certificate path for TLS connection
 --cert string                Certificate path for TLS connection
 --key string                 Private key path for TLS connection
 -u, --pd strings             PD address (default [127.0.0.1:2379])
 -s, --storage string         specify the url where backup storage, eg, "s3://bucket/path/prefix"
```

以上命令行示例只展示了常用的参数，这些参数作用如下：

- `--start-ts`：指定开始备份日志的起始时间点。如果未指定，备份程序选取当前时间作为 `start-ts`。
- `task-name`：指定日志备份任务名。该名称也用于查询备份状态、暂停、重启和恢复备份任务等操作。
- `ca`、`cert`、`key`：指定使用 mTLS 加密方式与 TiKV 和 PD 进行通讯。
- `--pd`：指定备份集群的 PD 访问地址。br 命令行工具需要访问 PD，发起日志备份任务。
- `--storage`：指定备份存储地址。日志备份支持以 Amazon S3、 Google Cloud Storage (GCS)、 Azure Blob Storage 为备份存储，以上命令以 S3 为示例。详细参考备份存储 URI 格式。

使用示例：

```
./br log start --task-name=pitr --pd="${PD_IP}:2379" \
--storage='s3://backup-101/logbackup?access-key=${access-key}&secret-access-key=${secret-access-
    ↪ key}"'
```

查询日志备份任务

执行 `br log status` 命令，你可以查询日志备份任务状态。

执行 `br log status --help` 命令可获取该子命令使用介绍：

```
./br log status --help
get status for the log backup task

Usage:
  br log status [flags]

Flags:
  -h, --help           help for status
  --json               Print JSON as the output.
  --task-name string   The task name for backup stream log. If default, get status of all of
      ↪ tasks (default "*")

Global Flags:
 --ca string                  CA certificate path for TLS connection
 --cert string                Certificate path for TLS connection
 --key string                 Private key path for TLS connection
 -u, --pd strings             PD address (default [127.0.0.1:2379])
```

以上示例中，`--task-name` 为常用参数，它用来指定日志备份任务名。默认值为 `*`，即显示全部任务。

使用示例：

```
./br log status --task-name=pitr --pd="${PD_IP}:2379"
```

命令输出如下：

```
• Total 1 Tasks.
> #1 <
               name: pitr
             status: • NORMAL
              start: 2022-07-14 20:08:03.268 +0800
                end: 2090-11-18 22:07:45.624 +0800
            storage: s3://backup-101/logbackup
       speed(est.): 0.82 ops/s
checkpoint[global]: 2022-07-25 22:52:15.518 +0800; gap=2m52s
```

命令输出中的字段含义如下：

- `status`：任务状态，包括 NORMAL（正常）、ERROR（异常）和 PAUSE（暂停）三种状态。
- `start`：日志备份任务开始的时间，该值为备份任务启动时候指定的 `start-ts`。
- `storage`：备份存储。
- `speed`：日志备份任务的总 QPS（每秒备份的日志个数）。
- `checkpoint [global]`：集群中早于该 `checkpoint` 的数据都已经保存到备份存储，它也是备份数据可恢复的最近时间点。
- `error [store]`：存储节点上的日志备份组件运行遇到的异常。

暂停和恢复日志备份任务

执行 `br log pause` 命令，你可以暂停正在运行的日志备份任务。

执行 `br log pause -help` 可获取该子命令使用介绍：

```
./br log pause --help
pause a log backup task

Usage:
  br log pause [flags]

Flags:
  --gc-ttl int         the TTL (in seconds) that PD holds for BR's GC safepoint (default 86400)
  -h, --help           help for status
  --task-name string   The task name for backup stream log.

Global Flags:
 --ca string                  CA certificate path for TLS connection
 --cert string                Certificate path for TLS connection
```

```
 --key string                  Private key path for TLS connection
 -u, --pd strings              PD address (default [127.0.0.1:2379])
```

> **注意：**
>
> - 暂停日志备份任务后，备份程序为了防止生成变更日志的 MVCC 数据被删除，暂停任务程序会自动将当前备份点 checkpoint 设置为 service safepoint，允许最多保留最近 24 小时内的 MVCC 数据。如果暂停的日志备份任务超过 24 小时未恢复，对应的数据就会被 GC，不会备份。
> - 保留过多的 MVCC 数据会影响 TiDB 集群的存储容量和性能，任务暂停后请及时恢复任务。

使用示例：

```
./br log pause --task-name=pitr --pd="${PD_IP}:2379"
```

执行 `br log resume` 命令，你可以恢复被暂停的日志备份任务。

执行 `br log resume --help` 命令可获取该子命令使用介绍：

```
./br log resume --help
resume a log backup task

Usage:
  br log resume [flags]

Flags:
  -h, --help           help for status
  --task-name string   The task name for backup stream log.

Global Flags:
 --ca string                   CA certificate path for TLS connection
 --cert string                 Certificate path for TLS connection
 --key string                  Private key path for TLS connection
 -u, --pd strings              PD address (default [127.0.0.1:2379])
```

暂停日志备份任务超过了 24 小时后，执行 `br log resume` 会报错，提示备份数据丢失。处理方法请参考恢复日志备份任务失败。

使用示例：

```
./br log resume --task-name=pitr --pd="${PD_IP}:2379"
```

停止和重启日志备份任务

通过执行 `br log stop` 命令，你可以停止正在进行的日志备份任务。停止的任务，可以通过 `--storage` 路径重新启动。

停止日志备份任务

执行 `br log stop` 命令，可以停止日志备份任务，该命令会清理备份集群中的任务元信息。

执行 `br log stop --help` 命令可获取该子命令使用介绍：

```
./br log stop --help
stop a log backup task

Usage:
  br log stop [flags]

Flags:
  -h, --help           help for status
  --task-name string   The task name for the backup log task.

Global Flags:
 --ca string                  CA certificate path for TLS connection
 --cert string                Certificate path for TLS connection
 --key string                 Private key path for TLS connection
 -u, --pd strings             PD address (default [127.0.0.1:2379])
```

> **注意：**
>
> 请谨慎使用该命令，如果你只需暂时停止日志备份，请使用 `br log pause` 和 `br log resume` 命令。

使用示例：

```
./br log stop --task-name=pitr --pd="${PD_IP}:2379"
```

重新启动备份任务

当使用 `br log stop` 命令停止日志备份任务后，可在另一个 `--storage` 路径下重新创建一个新的日志备份任务，也可以在原来的 `--storage` 路径下执行 `br log start` 命令重新启动日志备份任务。如果是在原来的 `--storage` 路径重启任务，需要注意：

- 重启备份任务的 `--storage` 参数需要与停止任务之前的参数相同。
- 此时不需要填入 `--start-ts` 参数，程序将自动从上次的备份进度点开始备份数据。
- 如果停止任务后的时间过长，多版本的数据已经被 GC，则在重启备份任务时会报错 `BR:Backup:ErrBackupGCSafepointExceeded`，此时只能配置另外的日志路径来重新创建日志备份任务。

清理日志备份数据

执行 `br log truncate` 命令，你可以从备份存储中删除过期或不再需要的备份日志数据。

执行 `br log truncate --help` 命令可获取该子命令使用介绍：

```
./br log truncate --help
truncate the incremental log until sometime.

Usage:
  br log truncate [flags]

Flags:
  --dry-run        Run the command but don't really delete the files.
  -h, --help       help for truncate
  --until string   Remove all backup data until this TS.(support TSO or datetime, e.g.
      ↪ '400036290571534337' or '2018-05-11 01:42:23+0800'.)
  -y, --yes        Skip all prompts and always execute the command.


Global Flags:
  -s, --storage string         specify the url where backup storage, eg, "s3://bucket/path/prefix
      ↪ "
```

该命令只需要访问备份存储，不需要访问备份集群。此外，常用的参数如下：

- `--dry-run`：运行命令，但是不删除文件。
- `--until`：早于该参数指定时间点的日志备份数据会被删除。建议使用快照备份的时间点作为该参数值。
- `--storage`：指定备份存储地址。日志备份支持以 Amazon S3、 Google Cloud Storage (GCS)、 Azure Blob Storage 为备份存储。详细参考备份存储 URI 格式。

使用示例：

```
./br log truncate --until='2022-07-26 21:20:00+0800' \
--storage='s3://backup-101/logbackup?access-key=${access-key}&secret-access-key=${secret-access-
    ↪ key}"'
```

该子命令运行后输出以下信息：

```
Reading Metadata... DONE; take = 277.911599ms
We are going to remove 9 files, until 2022-07-26 21:20:00.0000.
Sure? (y/N) y
Clearing data files... DONE; take = 43.504161ms, kv-count = 53, kv-size = 4573(4.573kB)
Removing metadata... DONE; take = 24.038962ms
```

查看备份数据元信息

执行 `br log metadata` 命令，你可以查看备份存储中保存的日志备份的元信息，例如最早和最近的可恢复时间点。

执行 `br log metadata --help` 命令可获取该子命令使用介绍：

```
./br log metadata --help
get the metadata of log backup storage

Usage:
  br log metadata [flags]

Flags:
  -h, --help       help for metadata

Global Flags:
  -s, --storage string         specify the url where backup storage, eg, "s3://bucket/path/prefix
      ↪ "
```

该命令只需要访问备份存储，不需要访问备份集群。

以上示例中，`--storage` 为常用参数，它用来指定备份存储地址。日志备份支持以 Amazon S3、 Google Cloud Storage (GCS)、 Azure Blob Storage 为备份存储。详细参考备份存储 URI 格式。

使用示例：

```
./br log metadata --storage='s3://backup-101/logbackup?access-key=${access-key}&secret-access-key
    ↪ =${secret-access-key}"'
```

该子命令运行后输出以下信息：

```
[2022/07/25 23:02:57.236 +08:00] [INFO] [collector.go:69] ["log metadata"] [log-min-ts
    ↪ =434582449885806593] [log-min-date="2022-07-14 20:08:03.268 +0800"] [log-max-ts
    ↪ =434834300106964993] [log-max-date="2022-07-25 23:00:15.618 +0800"]
```

8.3.4.3.2 恢复到指定时间点 PITR

执行 `br restore point` 命令，你可以在新集群上进行 PITR，或者只恢复日志备份数据。

执行 `br restore point --help` 命令可获取该命令使用介绍：

```
./br restore point --help
restore data from log until specify commit timestamp

Usage:
  br restore point [flags]

Flags:
  --full-backup-storage string specify the backup full storage. fill it if want restore full
      ↪ backup before restore log.
  -h, --help                   help for point
  --restored-ts string         the point of restore, used for log restore. support TSO or
      ↪ datetime, e.g. '400036290571534337' or '2018-05-11 01:42:23+0800'
```

```
  --start-ts string            the start timestamp which log restore from. support TSO or
      ↪ datetime, e.g. '400036290571534337' or '2018-05-11 01:42:23+0800'


Global Flags:
 --ca string                  CA certificate path for TLS connection
 --cert string                Certificate path for TLS connection
 --key string                 Private key path for TLS connection
 -u, --pd strings             PD address (default [127.0.0.1:2379])
 -s, --storage string         specify the url where backup storage, eg, "s3://bucket/path/prefix"
```

以上示例只展示了常用的参数，这些参数作用如下：

- `--full-backup-storage`：指定快照（全量）备份的存储地址。如果你要使用 PITR，需要指定该参数，并选择恢复时间点之前最近的快照备份；如果只恢复日志备份数据，则不需要指定该参数。快照备份支持以 Amazon S3、 Google Cloud Storage (GCS)、 Azure Blob Storage 为备份存储。详细参考备份存储 URI 格式。
- `--restored-ts`：指定恢复到的时间点。如果没有指定该参数，则恢复到日志备份数据最后的可恢复时间点（备份数据的 checkpoint）。
- `--start-ts`：指定日志备份恢复的起始时间点。如果你只恢复日志备份数据，不恢复快照备份，需要指定这个参数。
- `ca`、`cert`、`key`：指定使用 mTLS 加密方式与 TiKV 和 PD 进行通讯。
- `--pd`：指定恢复集群的 PD 访问地址。
- `--storage`：指定备份存储地址。日志备份支持以 Amazon S3、 Google Cloud Storage (GCS)、 Azure Blob Storage 为备份存储。详细参考备份存储 URI 格式。

使用示例：

```
./br restore point --pd="${PD_IP}:2379"
--storage='s3://backup-101/logbackup?access-key=${access-key}&secret-access-key=${secret-access-
    ↪ key}"'
--full-backup-storage='s3://backup-101/snapshot-202205120000?access-key=${access-key}&secret-
    ↪ access-key=${secret-access-key}"'

Full Restore
    ↪ <----------------------------------------------------------------------------------------------------------
    ↪  100.00%
*** ***["Full Restore success summary"] ****** [total-take=3.112928252s] [restore-data-size(after
    ↪ -compressed)=5.056kB] [Size=5056] [BackupTS=434693927394607136] [total-kv=4] [total-kv-
    ↪ size=290B] [average-speed=93.16B/s]
Restore Meta Files
    ↪ <----------------------------------------------------------------------------------------------------------
    ↪  100.00%
Restore KV Files
    ↪ <----------------------------------------------------------------------------------------------------------
    ↪  100.00%
```

```
"restore log success summary"] [total-take=192.955533ms] [restore-from=434693681289625602] [
    ↪ restore-to=434693753549881345] [total-kv-count=33] [total-size=21551]
```

> 注意：
>
> - 不支持重复恢复某段时间区间的日志，如多次重复恢复 `[t1=10, t2=20)` 区间的日志数据，可能会造成恢复后的数据不正确。
> - 多次恢复不同时间区间的日志时，需保证恢复日志的连续性。如先后恢复 `[t1, t2)`、`[t2, t3)` 和 `[t3, t4)` 三个区间的日志可以保证正确性，而在恢复 `[t1, t2)` 后跳过 `[t2, t3)` 直接恢复 `[t3, t4)` 的区间可能导致恢复之后的数据不正确。

8.3.5 参考指南

8.3.5.1 BR 特性

8.3.5.1.1 备份自动调节从 v5.4 版本开始引入

在 TiDB v5.4.0 之前，默认情况下，使用 BR 进行备份任务时使用的线程数量占总逻辑 CPU 数量的 75%。在没有限速的前提下，备份会消耗大量的集群资源，这会对在线集群的性能造成相当大的影响。虽然你可以通过调节线程池的大小的方式来减少备份对集群性能的影响，但观察负载、手动调节线程池大小也是一件繁琐的事情。

为了减少备份任务对在线集群的影响，从 TiDB v5.4.0 起，引入了自动调节功能，此功能默认开启。在集群资源占用率较高的情况下，备份功能可以通过该功能自动限制备份使用的资源，从而减少对集群的影响。

使用场景

如果你希望减少备份对集群的影响，那么，你可以开启自动调节功能。开启该功能后，备份功能会在不过度影响集群的前提下，以最快的速度进行数据备份。

或者，你也可以使用 TiKV 配置项`backup.num-threads` 或参数 `--ratelimit` 进行备份限速。

使用方法

自动调节功能默认打开，无需额外配置。

> 注意：
>
> v5.3.x 版本的集群，在升级到 v5.4.0 及以上版本后，自动调节功能默认关闭，需手动开启。

如需开启备份自动调节功能，可以通过把 TiKV 配置项`backup.enable-auto-tune` 设置为 `true` 的方式来完成。

TiKV 支持动态配置自动调节功能，因此，在开启或关闭该功能时，无需重启集群。你可以运行以下命令动态启动或停止备份自动调节功能：

```
tikv-ctl modify-tikv-config -n backup.enable-auto-tune -v <true|false>
```

在离线备份场景中，你也可以使用 tikv-ctl 把 backup.num-threads 修改为更大的数字，从而提升备份速度。

使用限制

自动调节是一个粗粒度的限流方案，它的优势在无需手动调节。但是，由于调节的粒度不够精确，该功能有可能无法彻底移除备份对集群的影响。

该功能的已知问题及其解决方案如下：

- 问题 1：对于以写负载为主的集群，自动调节可能会让工作负载和备份进入一种“正反馈循环”：备份会占用较多资源，导致工作负载使用的资源变少。此时，自动调节会误以为资源使用率下降，从而让备份运行得更加激进。在这种情况下，自动调节实际上失效。
  - 解决方法：手动调节 backup.num-threads，限制处理备份的工作线程数量。具体原理如下：
    目前，备份过程会涉及大量的 SST 解码、编码、压缩、解压，而此过程会需要消耗大量的 CPU 资源。另外，以往的测试证明，备份过程中，用于备份的线程池的 CPU 利用率接近 100%。也就是说，备份任务会占用大量 CPU 资源。通过调整备份任务使用的线程数量，TiKV 可以控制备份任务使用的 CPU 核心数，从而减少其任务对集群性能的影响。
- 问题 2：对于存在热点的集群，产生热点的 TiKV 节点可能会被过度限流，从而拉慢备份的整体进度。
  - 解决方法：消除热点节点，或者在热点节点上关闭自动调节（关闭此功能可能会导致集群性能降低）。
- 问题 3：对于流量抖动非常大的场景，由于自动调节每隔auto-tune-refresh-interval（默认为一分钟）才会计算出新的限流，所以可能无法很好地应对流量抖动非常厉害的场景。
  - 解决方法：关闭自动调节。

实现原理

自动调节会通过调节备份时使用的工作线程池的大小，保证集群的 CPU 总体使用率不超过某个特定的值。

这个特性还有两个配置项未在 TiKV 文档中列出，仅在内部调试使用，正常备份时无需配置这两个参数。

- backup.auto-tune-remain-threads：
  - 通过控制备份任务占用的资源，自动调节会保证该节点中至少有该数量的核心会保持空闲的状态。
  - 默认值：round(0.2 * vCPU)
- backup.auto-tune-refresh-interval：
  - 每隔该值的时间段，自动调节会刷新统计信息并重新计算备份任务使用的 CPU 核心数的上限。
  - 默认值：1m

以下是一个使用自动调节功能的示例，其中 * 代表集群中被备份任务占用的 CPU，^ 代表其它任务占用的 CPU，- 代表空闲 CPU。

```
|--------| 系统总共有 8 颗逻辑 CPU。
|****----| 默认配置 `backup.num-threads` 为 `4`。请注意，在任何时候自动调节都不会让线程池大小大于
    ↪  `backup.num-threads`。
|^^****--| 默认配置 `auto-tune-remain-threads` = round(8 * 0.2) = 2。
    ↪ 自动调节会将备份任务的线程池大小调节至 `4`。
|^^^^**--| 由于集群的工作负载加重，自动调节将备份任务的线程池大小调节至 `2`。调节后，集群中仍有 2
    ↪  个 CPU 核心数保持空闲。
```

在监控面板的“Backup CPU Utilization”中，可以看到自动限流目前选择的线程池的大小：

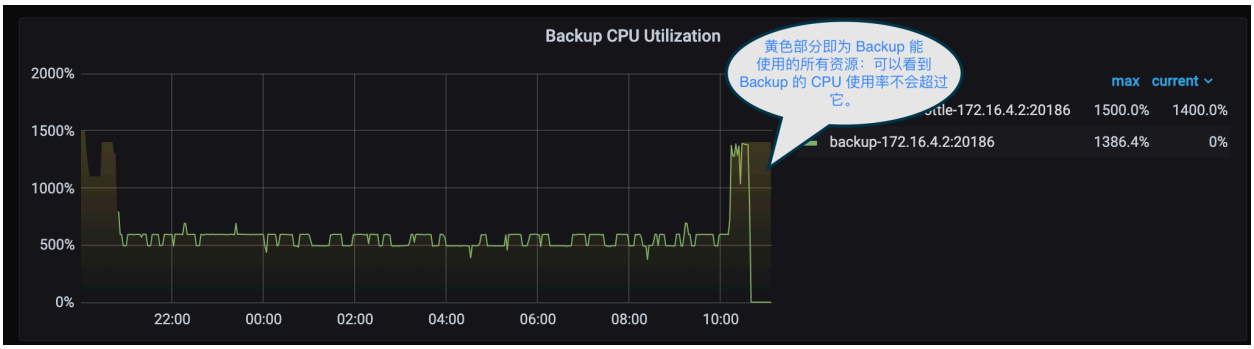


图 44: Grafana dashboard example of backup auto-tune metrics

图片中，黄色半透明的填充部分为开启自动调节后备份任务可用的线程，即备份任务能使用的所有资源。从中可以看到备份任务的 CPU 使用率不会超过黄色部分。

8.3.5.1.2 批量建表

使用 Backup & Restore (BR) 执行数据恢复任务时，BR 会先在目标 TiDB 集群上创建库和表，然后再把已备份的数据恢复到表中。TiDB v6.0.0 之前，在数据恢复阶段创建表时，BR 采用了串行执行的方案。然而，当需要恢复的数据中带有大量的表（约 50000 张）时，该方案会在创建表上消耗较多时间。

为了加快创建表的速度，以减少数据恢复的时间，TiDB 在 v6.0.0 中引入了批量建表功能，此功能默认开启。

> 注意：
>
> - 为使用批量建表功能，TiDB 和 br 命令行工具都必须为 v6.0.0 或以上版本。如果 TiDB 或 br 命令行工具的任意一方的版本低于 v6.0.0，br 命令行工具会采用原来的串行建表方案。
> - 如果你在使用 TiUP 等集群管理工具，而且你使用的 TiDB 和 br 命令行工具是 v6.0.0 或以上版本，或是从 v6.0.0 以下版本升级至该版本，此时，br 工具会默认开启批量建表功能，你不需要额外进行相关配置。

使用场景

当需要恢复的数据中带有大量的表（约 50000 张）时，你可以使用批量建表功能显著提升数据恢复的速度。

具体提速效果可参考测试结果。

使用方法

Backup & Restore (BR) 默认开启批量建表功能，在 v6.0.0 或以上版本中默认设置了 `--ddl-batch-size=128`（以 128 张表为一批，并发创建多批表），以加快恢复时的建表速度。因此，你不需要额外配置该参数。

如果需要关闭此功能，你可以参考以下命令将 `--ddl-batch-size` 的值设置为 `1`：

```
br restore full \
--storage local:///br_data/ --pd "${PD_IP}:2379" --log-file restore.log \
--ddl-batch-size=1
```

关闭批量建表功能后，BR 会采用原来的串行建表方案。

实现原理

- v6.0.0 前的串行建表方案：

  执行数据恢复任务时，BR 会先在目标 TiDB 集群上创建库和表，然后再把已备份的数据恢复到表中。建表时，BR 会调用 TiDB 内部 API 后开始创建表，其运作方式类似 BR 执行 SQL `Create Table` 语句。建表任务由 TiDB DDL owner 依次串行执行。DDL owner 每创建一张表会引起一次 DDL schema 版本的变更，而每次的 schema 版本的变更都需要同步到其他 TiDB DDL worker（含 BR）。因此，当需要创建的表的数量比较多时，串行建表方案会导致建表时间过长。

- v6.0.0 起的批量建表方案：

  在默认情况下，BR 会以 128 张表为一批，并发创建多批表。采用该方案后，BR 每建一批表时，TiDB schema 版本只会变更一次。此方法极大地提高了建表速度。

功能测试

以下是在 TiDB v6.0.0 集群中测试批量建表功能的内容。具体的测试环境如下：

- 集群配置：
  - 15 个 TiKV 实例，每个 TiKV 实例共有 16 个 CPU 核心、80 GB 内存、16 个处理 RPC 请求的线程（即`import.num-threads` = 16）
  - 3 个 TiDB 实例，每个 TiDB 实例共有 16 个 CPU 核心、32 GB 内存。
  - 3 个 PD 实例，每个 PD 实例共有 16 个 CPU 核心、32 GB 内存。
- 待恢复数据的规模：16.16 TB

测试结果如下：

```
'[2022/03/12 22:37:49.060 +08:00] [INFO] [collector.go:67] ["Full restore success summary"] [
  ↪ total-ranges=751760] [ranges-succeed=751760] [ranges-failed=0] [split-region=1h33m18
  ↪ .078448449s] [restore-ranges=542693] [total-take=1h41m35.471476438s] [restore-data-size(
  ↪ after-compressed)=8.337TB] [Size=8336694965072] [BackupTS=431773933856882690] [total-kv
  ↪ =148015861383] [total-kv-size=16.16TB] [average-speed=2.661GB/s]'
```

从结果可见，单个 TiKV 实例的平均恢复速度高达 181.65 MB/s（即 `average-speed/tikv_count`）。

8.3.5.1.3 断点备份

快照备份会因为一些可恢复性错误导致提前结束，例如硬盘空间占满、节点宕机等等一些突发情况。在 TiDB v6.5.0 之前，在错误被处理之后，之前备份的数据会作废，你需要重新进行备份。对大规模集群来说，会造成大量额外成本。

为了尽可能继续上一次的备份，从 TiDB v6.5.0 起，备份恢复特性引入了断点备份的功能，此功能默认开启。开启该功能后，备份在意外退出后可以保留上一次备份的大部分进度。

使用场景

如果你的 TiDB 集群规模很大，无法接受备份失败后需要重新备份的情况，那么，你可以开启断点备份功能。开启该功能后，br 工具会定期记录已备份的分片，使得在下一次重试备份时回到离上一次退出时较近的进度。

使用限制

断点备份功能依赖于 GC 机制，且无法恢复所有已备份的数据，具体描述如下。

确保在 GC 前重试

在备份过程中，br 命令行工具会定期向 PD 更新备份 snapshot 的 `gc-safepoint`，从而避免在备份过程中数据被 GC。当 br 工具退出后，PD 中备份 snapshot 的 `gc-safepoint` 将无法及时更新。因此，在下一次重试备份前，数据有可能已经被 GC。

为了避免数据被 GC，当 br 工具没有指定具体的 `gcttl` 时，br 工具在默认情况下会让 `gc-safepoint` 保留大约 1 小时。如果你需要延长这个时间，可以设置参数 `gcttl`。

例如，设置 `gcttl` 为 15 小时（54000 秒）来延长 `gc-safepoint` 的保留时间：

```
br backup full \
--storage local:///br_data/ --pd "${PD_IP}:2379" \
--gcttl 54000
```

> **注意：**
>
> 快照备份结束后会将备份前创建的 `gc-safepoint` 删除，不需要手动删除。

部分数据需要重新备份

在重试备份时，部分已备份的数据可能需要重新备份，包括正在备份的数据和还未被断点记录的数据。

- 由错误引起的退出，br 工具会在退出前将已备份的数据元信息持久化到外部存储中，因此只有正在备份的数据需要在下一次重试中重新备份。
- 如果 br 工具进程被系统中断，那么 br 将无法把已备份的数据元信息持久化到外部存储中。br 工具持久化已备份的数据元信息的周期为 30 秒，因此大约最近 30 秒内的已完成备份的数据无法持久化到外部存储，在下次重试时需要重新备份。

实现原理

在快照备份过程中，br 工具将表编码成对应的键空间，并生成备份的 RPC 请求发送给所有 TiKV 节点。接收到备份请求后，TiKV 节点会选择请求范围内的数据进行备份。每当 TiKV 节点备份完一个 `region` 级别的数据，就会返回给 br 工具这段范围相关的备份信息。

通过记录这些返回的备份信息，br 工具能够及时获知已备份完成的键范围。断点备份功能会定期将这些新增的相关备份信息上传至外部存储中，从而持久化存储该备份任务已完成的键范围。

在重试备份时，br 工具会从外部存储读取已经备份的键范围，再与所需备份的键范围做比较，获得一个差集，从而得出断点备份还需要备份的键范围。

8.3.5.2 使用 Dumpling 和 TiDB Lightning 备份与恢复

本文档介绍如何使用 Dumpling 和 TiDB Lightning 进行全量备份与恢复。

在备份与恢复场景中，如果需要全量备份少量数据（例如小于 50 GB），且不要求备份速度，你可以使用Dumpling 从 TiDB 数据库导出数据进行备份，再使用TiDB Lightning 将数据导入至 TiDB 数据库实现恢复。更多备份与恢复的相关信息，参见TiDB 备份与恢复概述。

8.3.5.2.1 前提条件

- 安装和运行 Dumpling：

```
tiup install dumpling && tiup dumpling
```

- 安装和运行 TiDB Lightning：

```
tiup install tidb lightning && tiup tidb lightning
```

- 获取 Dumpling 所需上游数据库权限
- 获取 TiDB Lightning 所需下游数据库权限

8.3.5.2.2 资源要求

操作系统：本文档示例使用的是若干新的、纯净版 CentOS 7 实例，你可以在本地虚拟化一台主机，或在供应商提供的平台上部署一台小型的云虚拟主机。TiDB Lightning 运行过程中，默认会占满 CPU，建议单独部署在一台主机上。如果条件不允许，你可以将 TiDB Lightning 和其他组件（比如 `tikv-server`）部署在同一台机器上，然后设置 `region-concurrency` 配置项的值为逻辑 CPU 数的 75%，以限制 TiDB Lightning 对 CPU 资源的使用。

内存和 CPU：因为 TiDB Lightning 对计算机资源消耗较高，建议分配 64 GB 以上的内存以及 32 核以上的 CPU，而且确保 CPU 核数和内存 (GB) 比为 1:2 以上，以获取最佳性能。

磁盘空间：

推荐使用 Amazon S3、Google Cloud Storage (GCS) 和 Azure Blob Storage 等外部存储，以便能够快速存储备份文件，且不受磁盘空间限制。

如果需要保存单次备份数据到本地磁盘，需要注意以下磁盘空间限制：

- Dumpling 需要能够储存整个数据源的存储空间，即可以容纳要导出的所有上游表的空间。计算方式参考目标数据库所需空间。
- TiDB Lightning 导入期间，需要临时空间来存储排序键值对，磁盘空间需要至少能存储数据源的最大单表。

说明：目前无法精确计算 Dumpling 从 TiDB 导出的数据大小，但你可以用下面 SQL 语句统计信息表的 `data_length` 字段估算数据量：

```
/* 统计所有 schema 大小，单位 MiB，注意修改 ${schema_name} */
SELECT table_schema,SUM(data_length)/1024/1024 AS data_length,SUM(index_length)/1024/1024 AS
    ↪ index_length,SUM(data_length+index_length)/1024/1024 AS SUM FROM information_schema.
    ↪ tables WHERE table_schema = "${schema_name}" GROUP BY table_schema;
```

```
/* 统计最大单表，单位 MiB，注意修改 ${schema_name} */
SELECT table_name,table_schema,SUM(data_length)/1024/1024 AS data_length,SUM(index_length)
    ↪ /1024/1024 AS index_length,SUM(data_length+index_length)/1024/1024 AS SUM from
    ↪ information_schema.tables WHERE table_schema = "${schema_name}" GROUP BY table_name,
    ↪ table_schema ORDER BY SUM DESC LIMIT 5;
```

目标 TiKV 集群的磁盘空间要求

目标 TiKV 集群必须有足够空间接收新导入的数据。除了标准硬件配置以外，目标 TiKV 集群的总存储空间必须大于数据源大小 × 副本数量 × 2。例如，集群默认使用 3 副本，那么总存储空间需为数据源大小的 6 倍以上。公式中的 2 倍可能难以理解，其依据是以下因素的估算空间占用：

- 索引会占据额外的空间。
- RocksDB 的空间放大效应。

8.3.5.2.3 使用 Dumpling 备份全量数据

1. 运行以下命令，从 TiDB 导出全量数据至 Amazon S3 存储路径 `s3://my-bucket/sql-backup`：

    ```
    tiup dumpling -h ${ip} -P 3306 -u root -t 16 -r 200000 -F 256MiB -B my_db1 -f 'my_db1.table
        ↪ [12]' -o 's3://my-bucket/sql-backup'
    ```

    Dumpling 默认导出数据格式为 SQL 文件，你也可以通过设置 `--filetype` 指定导出文件的类型。

    关于更多 Dumpling 的配置，请参考Dumpling 主要选项表。

2. 导出完成后，可以在数据存储目录 `s3://my-bucket/sql-backup` 查看导出的备份文件。

8.3.5.2.4 使用 TiDB Lightning 恢复全量数据

1. 编写配置文件 `tidb-lightning.toml`，将 Dumpling 备份的全量数据从 `s3://my-bucket/sql-backup` 恢复到目标 TiDB 集群：

    ```
    [lightning]
    # 日志
    level = "info"
    file = "tidb-lightning.log"

    [tikv-importer]
    # "local"：默认使用该模式，适用于 TB 级以上大数据量，但导入期间下游 TiDB 无法对外提供服务。
    # "tidb"：TB 级以下数据量也可以采用`tidb`后端模式，下游 TiDB 可正常提供服务。
        ↪ 关于后端模式更多信息请参阅：https://docs.pingcap.com/tidb/stable/tidb-lightning-
        ↪ backends
    backend = "local"
    # 设置排序的键值对的临时存放地址，目标路径必须是一个空目录，目录空间须大于待导入数据集的大小
        ↪ 。建议设为与 `data-source-dir` 不同的磁盘目录并使用闪存介质，独占 IO
        ↪ 会获得更好的导入性能
    ```

```
sorted-kv-dir = "${sorted-kv-dir}"

[mydumper]
# 源数据目录，即上一章节中 Dumpling 保存数据的路径。
data-source-dir = "${data-path}" # 本地或 S3 路径，例如：'s3://my-bucket/sql-backup'

[tidb]
# 目标集群的信息
host = ${host}                # 例如：172.16.32.1
port = ${port}                # 例如：4000
user = "${user_name}"         # 例如："root"
password = "${password}"      # 例如："rootroot"
status-port = ${status-port}  # 导入过程 Lightning 需要在从 TiDB 的“状态端口”
    ↪ 获取表结构信息，例如：10080
pd-addr = "${ip}:${port}"     # 集群 PD 的地址，Lightning 通过 PD 获取部分信息，例如
    ↪ 172.16.31.3:2379。当 backend = "local" 时 status-port 和 pd-addr 必须正确填写，
    ↪ 否则导入将出现异常。
```

关于更多 TiDB Lightning 的配置，请参考TiDB Lightning 配置参数。

2. 运行 tidb-lightning。如果直接在命令行中启动程序，可能会因为 SIGHUP 信号而退出，建议配合 nohup 或 screen 等工具，如：

   若从 Amazon S3 导入，则需将有权限访问该 S3 后端存储的账号的 SecretKey 和 AccessKey 作为环境变量传入 Lightning 节点。同时还支持从 ~/.aws/credentials 读取凭证文件。

```
export AWS_ACCESS_KEY_ID=${access_key}
export AWS_SECRET_ACCESS_KEY=${secret_key}
nohup tiup tidb-lightning -config tidb-lightning.toml > nohup.out 2>&1 &
```

3. 导入开始后，可以通过 grep 日志关键字 progress 查看进度，默认 5 分钟更新一次。

4. 导入完毕后，TiDB Lightning 会自动退出。查看 tidb-lightning.log 日志末尾是否有 the whole procedure ↪ completed，如果有，数据导入成功，恢复完成。如果没有，则表示导入遇到了问题，可根据日志中的 error 提示解决遇到的问题。

> **注意：**
>
> 无论导入成功与否，最后一行都会显示 tidb lightning exit。它只是表示 TiDB Lightning 正常退出，不代表恢复任务完成。

如果恢复过程中遇到问题，请参见TiDB Lightning 常见问题。

8.3.5.3 备份与恢复 RawKV 数据

TiKV 可以独立于 TiDB，与 PD 构成 KV 数据库，此时的产品形态为 RawKV。TiKV-BR 工具支持对使用 RawKV 的产品进行备份和恢复。

更多信息，请参考 TiKV-BR 用户文档。

8.3.5.4 TiDB 增量备份与恢复使用指南

TiDB 集群增量数据包含在某个时间段的起始和结束两个快照的变化差异的数据，以及之间的 DDL。增量备份的数据相对比全量备份数据而言数据量更小，适合配合快照备份一起使用，减少备份的数据量。进行增量备份的时候，需要保证备份时间范围内的多版本数据没有被TiDB GC 机制回收。例如，每个小时进行一次增量备份，则需要调整 TiDB 集群的 GC Lifetime 设置至少大于 1 小时。

> **警告：**
>
> 当前该功能已经停止开发迭代，推荐你选择日志备份与恢复功能代替。

8.3.5.4.1 对集群进行增量备份

使用 `br backup` 进行增量备份只需要指定上一次的备份时间戳 `--lastbackupts`，br 命令行工具会判定需要备份 `lastbackupts` 和当前时间之间增量数据。使用 `validate` 指令获取上一次备份的时间戳，示例如下：

```
LAST_BACKUP_TS=`tiup br validate decode --field="end-version" --storage "s3://backup-101/snapshot
    ↪ -202209081330?access-key=${access-key}&secret-access-key=${secret-access-key}"| tail -n1`
```

备份 `(LAST_BACKUP_TS, current timestamp]` 之间的增量数据，以及这段时间内的 DDL：

```
tiup br backup full --pd "${PD_IP}:2379" \
--storage "s3://backup-101/snapshot-202209081330/incr?access-key=${access-key}&secret-access-key=
    ↪ ${secret-access-key}" \
--lastbackupts ${LAST_BACKUP_TS} \
--ratelimit 128
```

以上命令会中：

- `--lastbackupts`：上一次的备份时间戳。
- `--ratelimit`：每个 TiKV 执行备份任务的速度上限（单位 MiB/s）。
- `storage`：数据备份到存储地址。增量备份数据需要与快照备份数据保存在不同的路径下，例如上例保存在全量备份数据下的 `incr` 目录中。详细参考备份存储 URI 配置。

8.3.5.4.2 恢复增量备份数据

恢复增量数据的时候，需要保证备份时指定的 `LAST_BACKUP_TS` 之前备份的数据已经全部恢复到目标集群。同时，因为增量恢复的时候会更新数据，因此你需要保证此时不会有其他写入，避免出现冲突。

恢复全量备份数据，备份数据存储在 `backup-101/snapshot-202209081330` 目录下：

```
tiup br restore full --pd "${PD_IP}:2379" \
--storage "s3://backup-101/snapshot-202209081330?access-key=${access-key}&secret-access-key=${
    ↪ secret-access-key}"
```

恢复全量备份后的增量备份数据，备份数据存储在 backup-101/snapshot-202209081330/incr 目录下：

```
tiup br restore full --pd "${PD_IP}:2379" \
--storage "s3://backup-101/snapshot-202209081330/incr?access-key=${access-key}&secret-access-key=
    ↪ ${secret-access-key}"
```

8.4 集群容灾

8.4.1 TiDB 容灾方案概述

本文将以如下结构系统介绍 TiDB 容灾解决方案：

- 介绍容灾解决方案涉及的基本概念。
- 介绍核心组件 TiDB、TiCDC 及 BR 的架构。
- 介绍各种容灾方案。
- 对比不同的容灾解决方案。

8.4.1.1 基本概念

- RTO (Recovery Time Objective)：是指灾难发生后，系统恢复服务所需的时间。
- RPO (Recovery Point Objective)：是指灾难发生后，确保对业务不产生损失的前提下，可以丢失的最大数据量。

下面的图形描述了这两个概念：

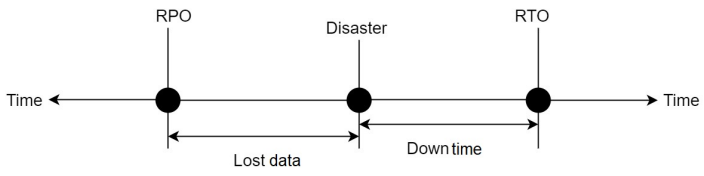


图 45: RTO and RPO

- 错误容忍目标：由于灾难可能影响的地域范围是不同的，在本文中，使用“错误容忍目标”来描述系统能够容忍的灾难的最大范围。
- 区域：本文主要讨论区域 (region) 级别的容灾方案，这里的区域通常是指一个物理世界中的地区或者城市。

8.4.1.2 组件架构

在介绍具体的容灾方案之前，本部分将从容灾角度介绍容灾系统中的组件架构，包括 TiDB、TiCDC 和 BR。

8.4.1.2.1 TiDB 架构

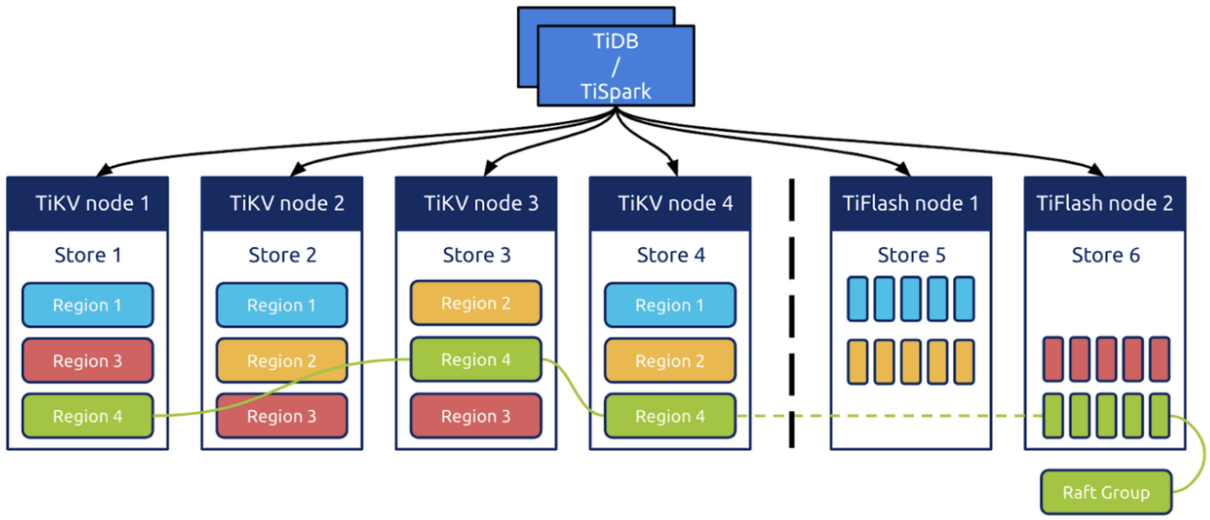


图 46: TiDB architecture

TiDB 的设计采用了计算、存储分离的架构：

- TiDB 为系统的计算层。
- TiKV 是系统的存储层，采用行存的方式保存数据库的数据记录，其中Region 是经过排序的若干行数据的集合，也是系统调度数据的单位。同一个 Region 的数据保存至少 3 份，通过 Raft 协议在日志层复制数据改变。
- TiFlash 副本是可选的，它是一款列式存储，用于加速分析类查询的速度。数据通过 Raft group 中的 learner 角色与 TiKV 中的数据进行复制。

由于 TiDB 保存了三份完整的数据副本，所以天然就具备了基于多副本数据复制的容灾能力。同时，由于 TiDB 采用了 Raft log 来进行事务日志同步，也在一定程度上具备了基于事务日志同步的容灾能力。

8.4.1.2.2 TiCDC 架构

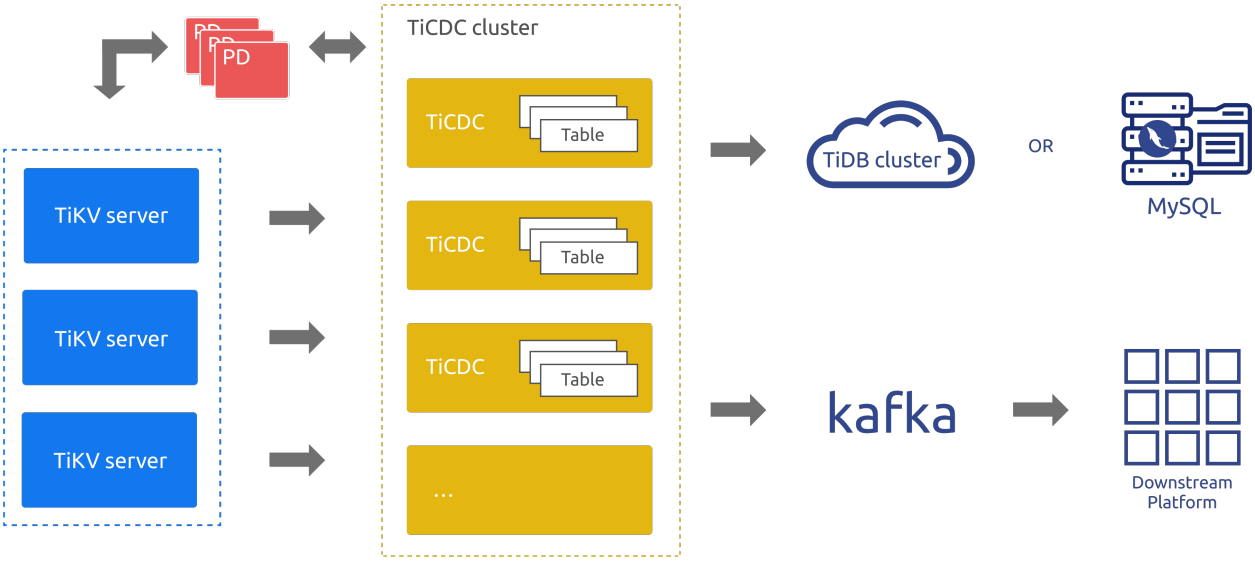


图 47: TiCDC architecture

TiCDC 作为 TiDB 的增量数据同步工具，通过 PD 内部的 etcd 实现高可用，通过多个 Capture 进程获取 TiKV 节点上的数据改变，在内部进行排序、合并等处理之后，通过多个同步任务，同时向多个下游系统进行数据同步。在上面的架构中：

- TiKV server：负责将对应节点上数据的改变推送到 TiCDC 节点。当然，如果 TiCDC 发现收到的数据改变不完整，也会主动联系 TiKV server 获取需要的数据改变。
- TiCDC：负责启动多个 Capture 进程，每个 Capture 负责拉取一部分的 KV change logs，并对获取到的数据改变进行排序，最后同步到不同的下游当中。

从上面的架构中可以看到，TiCDC 的架构和事务日志复制系统比较类似，但是扩展性更好，同时又兼顾了逻辑数据复制的很多特点。所以，TiCDC 也可以为 TiDB 在容灾场景提供很好的帮助和补充。

8.4.1.2.3 BR 架构

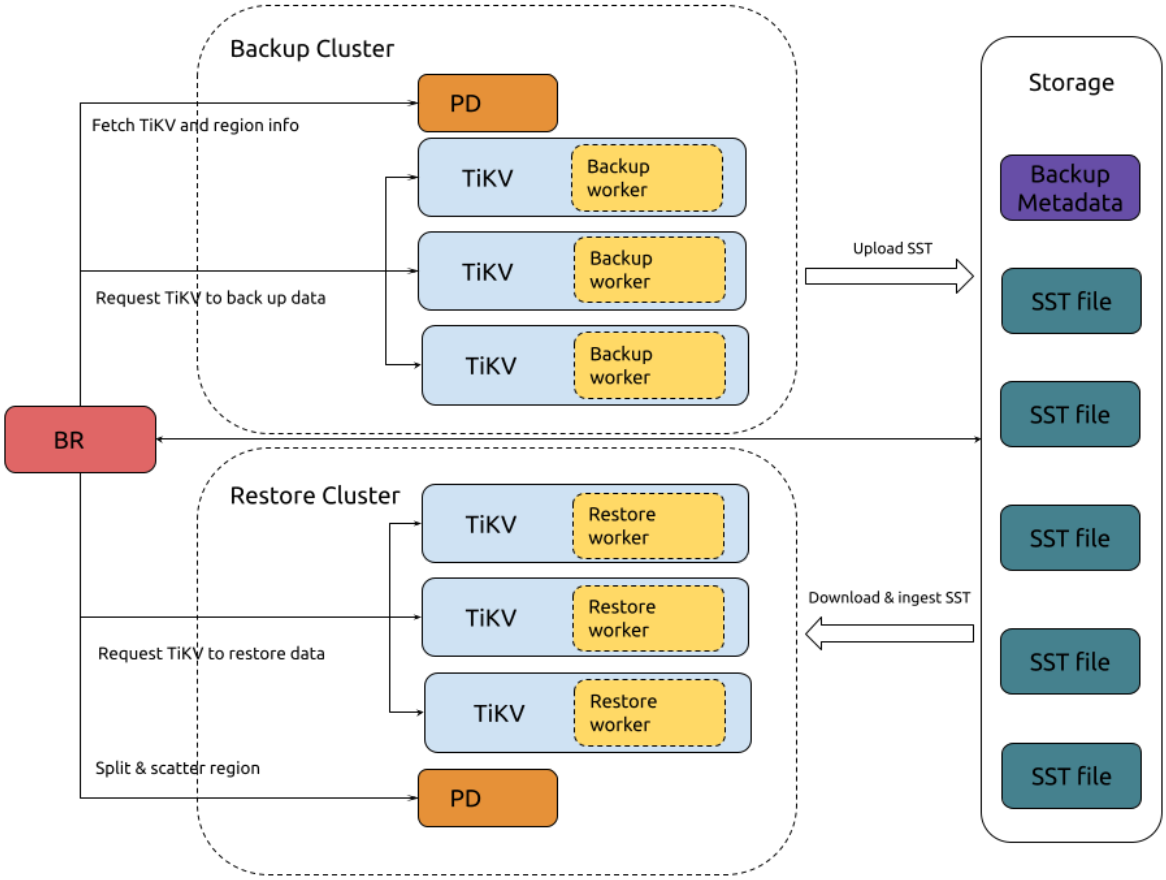


图 48: BR architecture

BR 作为 TiDB 的备份恢复工具，可以对 TiDB 集群进行基于时间点的全量快照备份和持续的日志备份，从而保护 TiDB 集群的数据。当 TiDB 集群完全不可用时，可以通过备份文件，在全新的集群中进行恢复。备份恢复通常是数据安全的最后一道防线。

8.4.1.3 方案介绍

8.4.1.3.1 基于 TiCDC 的主备集群容灾方案

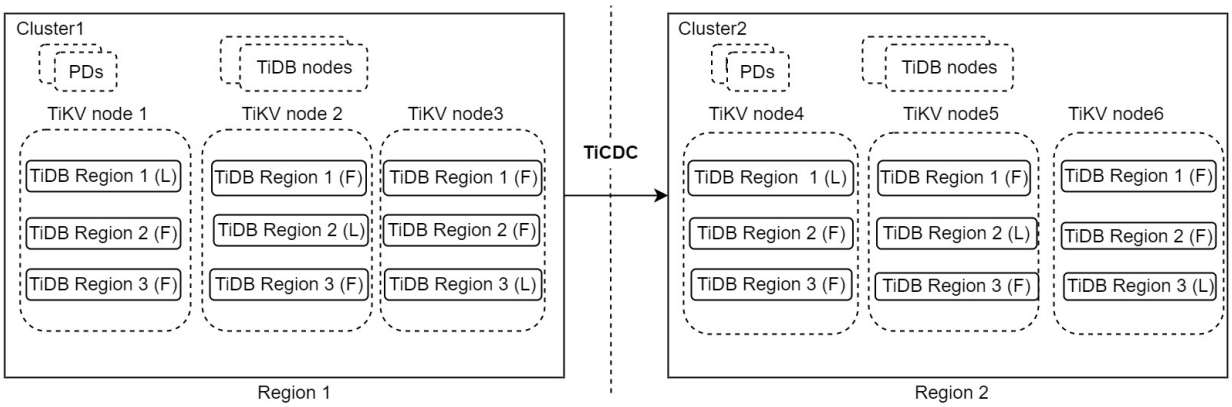


图 49: Primary-secondary cluster DR

在上面的架构中包含了两个 TiDB 集群，Cluster1 为主用集群，运行在区域 1 (Region 1)，包含 3 个副本，承担读写业务。Cluster2 作为灾备集群，运行在区域 2 (Region 2)。当 Cluster1 出现灾难时，Cluster2 继续对外提供服务。两个集群之间通过 TiCDC 进行数据改变的同步。这种架构，简称为 “1:1” 解决方案。

这种架构看起来非常简洁，可用性比较高，最大的错误容忍目标可以做到区域级别，写能力也能够得到扩展，RPO 在秒级别，RTO 在分钟级别，甚至更低。如果 RPO 为 0 并不是必须满足的要求，推荐在重要生产系统使用该容灾方案。对于该方案的详细信息，请参考基于主备集群的容灾方案。

8.4.1.3.2 基于多副本的单集群容灾方案

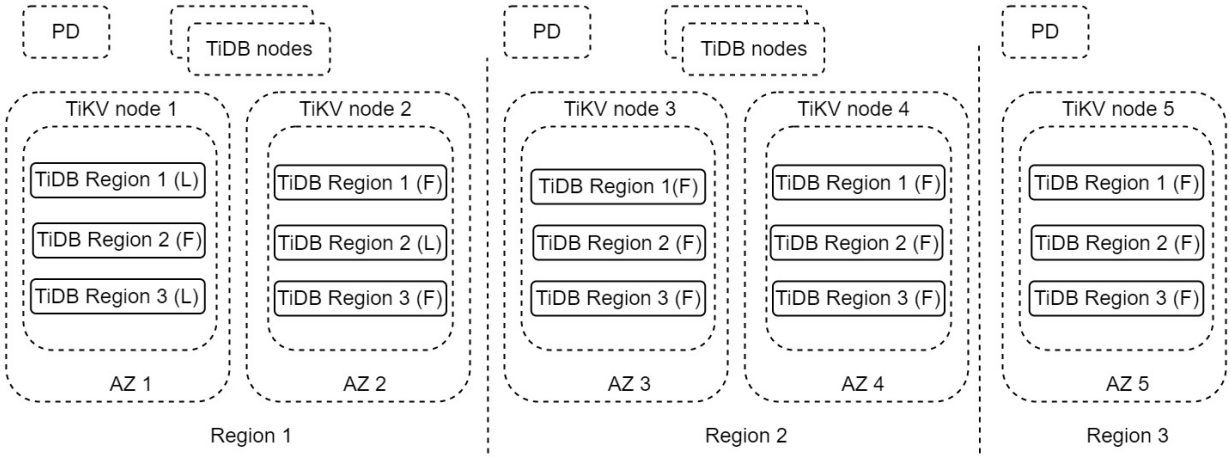


图 50: Multi-replica cluster DR

在上面的架构中，每个区域都包含两份完整的数据副本，它们位于不同的可用区 (Available Zone, AZ) 当中（通常情况下，两个可用区之间的网络速度和带宽条件较好，在同一个区域中的不同 AZ 中读写请求的延迟很低），整个集群横跨了三个区域。区域 1 通常是用来处理读写业务请求的主区域，当区域 1 出现灾难后完全不可用时，区域 2 可以作为灾难恢复的区域。而区域 3 (Region 3) 更多的是为了满足多数派协议而存在的一个副本。这

种架构，简称为“2-2-1”解决方案。

该方案最大的错误容忍目标可以达到区域级别，写能力也能够得到扩展，并且 RPO 为 0，RTO 也可以达到分钟级别，甚至更低。如果 RPO 为 0 是必须满足的要求，推荐在重要生产系统使用该容灾方案。对于该方案的详细信息，请参考基于多副本的单集群容灾方案。

8.4.1.3.3 多副本与 TiCDC 相结合的容灾解决方案

以上两种容灾解决方案都可以实现区域级别的容灾，但是都无法解决多个区域同时不可用的问题。如果你的系统非常重要，需要“错误容忍目标”达到多个区域，就需要将以上两种容灾解决方案进行结合。

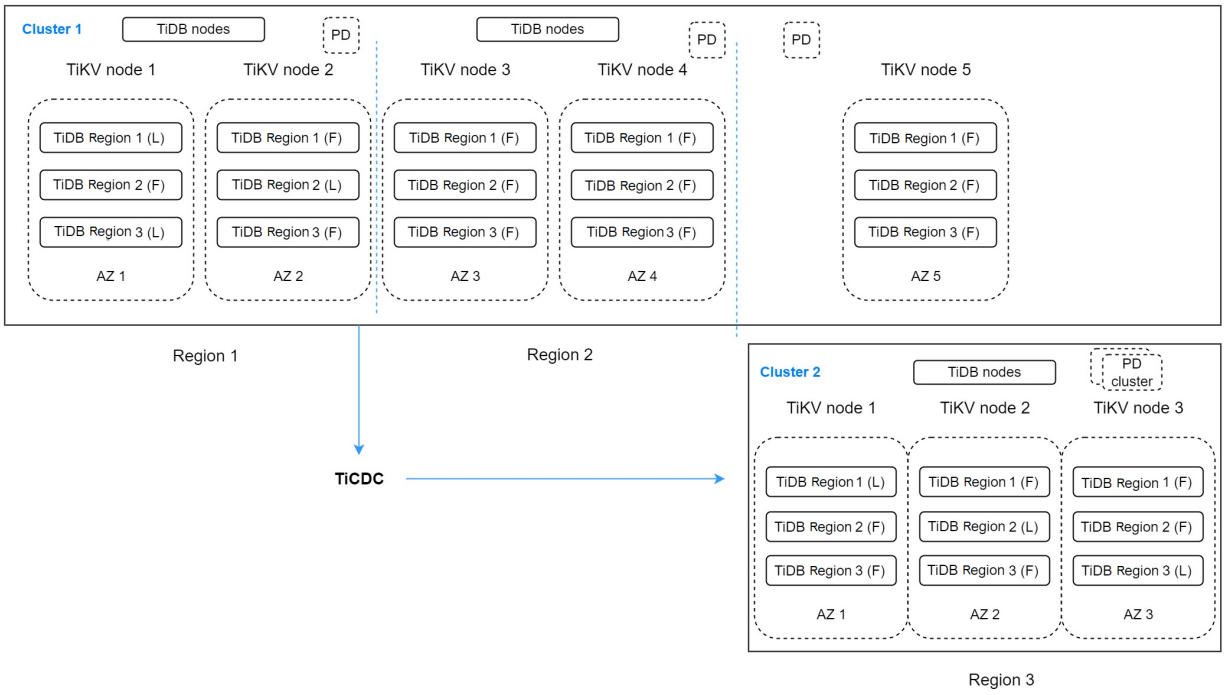


图 51: TiCDC-based multi-replica cluster DR

在上面的部署中存在两个 TiDB 集群。Cluster1 有 5 个副本，跨 3 个区域。区域 1 (Region 1) 包含两个副本作为主区域，用于服务写入。区域 2 (Region 2) 有两个副本作为区域 1 的容灾区域，可以提供一些延迟不敏感的读取服务。最后一个副本用于投票，位于区域 3 (Region 3) 中。

作为区域 1 和区域 2 的容灾集群，Cluster2 在区域 3 中运行，并包含 3 个副本。TiCDC 在两个集群之间同步数据更改。这种部署可能看起来比较复杂，但它可以将容错目标提高到多区域。如果多区域故障不要求 RPO 必须为 0，这种架构是一个很好的选择。这种架构，简称为“2-2-1:1”解决方案。

当然，如果“错误容忍目标”为多个区域，并且 RPO 为 0 是一个必须满足的要求，你也可以考虑创建一个包含至少 9 个副本，横跨 5 个区域的集群来实现该能力。这种架构，简称为“2-2-2-2-1”解决方案。

8.4.1.3.4 基于备份与恢复的容灾解决方案

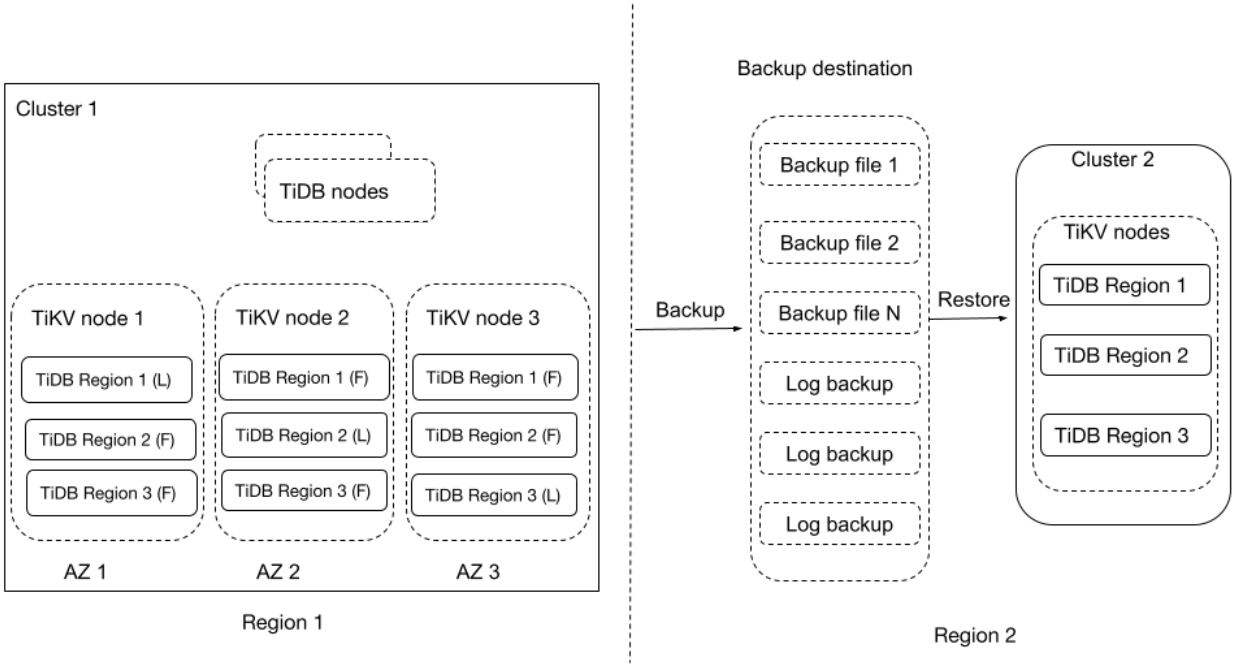


图 52: BR-based cluster DR

按照上面的部署，TiDB Cluster1 部署在区域 1 (Region 1)，BR 工具定期将集群的数据备份到区域 2 (Region 2)，并且持续将数据改变日志也备份到区域 2。当区域 1 出现灾难导致 Cluster1 无法恢复时，你可以使用备份的数据和数据改变在区域 2 恢复新的集群 Cluster2 对外提供服务。

基于备份恢复的容灾方案，目前，RPO 低于 5 分钟，而 RTO 则取决于需要恢复的集群数据大小，对于 v6.5.0 版本的 BR，其恢复速度可以参考快照恢复的性能与影响和PITR 的性能与影响。通常来说，大部分客户会把跨区域的备份作为数据安全的最后一道防线，是大多数系统都需要的。对于该方案的详细信息，请参考基于备份与恢复的容灾方案。

另外，从 v6.5.0 开始，BR 支持基于 AWS 上的 EBS 快照的快速恢复。如果你在 AWS 上运行 TiDB 集群，要求备份过程对集群没有任何影响，并且要求恢复的时间尽量短，可以考虑使用该特性来降低系统的 RTO。

8.4.1.3.5 其他容灾解决方案

除了以上容灾方案，针对同城双中心这种特定的场景，如果 RPO=0 是一个必须的条件，你也可以采用 DR-AUTO sync 解决方案。详细的信息请参考单区域双 AZ 部署 TiDB。

8.4.1.4 方案对比

最后，对本文提到的各种容灾解决方案进行对比，以方便你根据自己的业务需要选择合适的容灾方案。

容灾方案	TCO	错误容忍目标	RPO	RTO	网络延迟要求	使用的系统
基于多副本的单集群容灾方案 (2-2-1)	高	单个区域	0	分钟级	区域之间的网络延迟要求小于 30 ms。	对灾备和响应时间有明确要求 (RPO = 0) 的重要生产系统。

容灾方案	TCO	错误容忍目标	RPO	RTO	网络延迟要求	使用的系统
基于 TiCDC 的主备集群容灾方案 (1:1)	中等	单个区域	小于 10 秒	小于 5 分钟	区域之间的网络延迟要求小于 100 ms。	对灾备和响应时间有明确要求 (RPO > 0) 的重要生产系统。

容灾方案	TCO	错误容忍目标	RPO	RTO	网络延迟要求	使用的系统
多副本与 TiCDC 相结合的容灾解决方案 (2-2-1:1)	高	多个区域	小于 10 秒	小于 5 分钟	对于通过多副本进行容灾的区域，网络延迟建议小于 30 ms。对于第三区域与其他区域之间，建议延迟小于 100	对灾难恢复和响应时间有严格要求的关键生产系统。

容灾方案	TCO	错误容忍目标	RPO	RTO	网络延迟要求	使用的系统
基于备份恢复的容灾方案	低	单个区域	小于 5 分钟	小时级	无特殊要求	能够接受 RPO < 5 分钟，RTO 达到小时级别的系统。

8.4.2 基于主备集群的容灾方案

使用主、备数据库进行容灾是一种常用的容灾方式。在这种方案下，系统有一个主用集群和一个备用集群。主集群用于处理用户的请求，备集群负责备份主集群的数据。当主集群发生故障时，系统可以切换到备集群，使用备份的数据继续提供服务。这样，系统就可以在发生故障的情况下继续正常运行，避免因为故障而导致的服务中断。

主备集群容灾方案具有如下优势：

- 高可用性：主、备集群的架构可以有效提高系统的可用性，使得系统在遇到故障时能够快速恢复。
- 快速切换：在主集群发生故障的情况下，系统可以快速切换到备用集群，继续提供服务。
- 数据一致性：备用集群会近实时备份主集群的数据，因此，在故障发生后切换到备集群时，数据基本是最新的。

本文包含以下主要内容：

- 构建主备集群
- 从主集群复制数据至备集群
- 监控集群

- 容灾切换

同时，本文还介绍了如何在备用集群上进行业务查询，以及如何在主备集群间进行双向同步。

8.4.2.1 基于 TiCDC 构建 TiDB 主备集群

8.4.2.1.1 架构概览

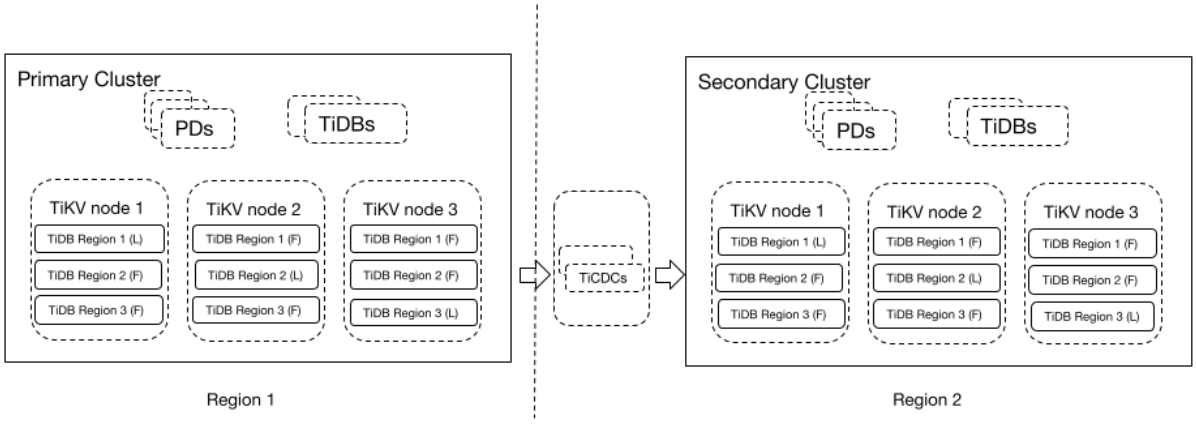


图 53: TiCDC secondary cluster architecture

上述架构包含两个 TiDB 集群：Primary Cluster 和 Secondary Cluster。

- Primary Cluster：主用集群，运行在区域 1 (Region 1)，三副本，用于处理读写业务。
- Secondary Cluster：备用集群，运行在区域 2 (Region 2)，通过 TiCDC 从 Primary Cluster 同步数据。

这种容灾架构简洁易用，可以容忍区域级别的故障，既可以保证主用集群的写入性能不会下降，还可以在备用集群处理一些延迟不敏感的只读业务。该方案的 Recovery Point Objective (RPO) 在秒级别，Recovery Time Objective (RTO) 可以达到分钟级别甚至更低。这个方案适用于重要的生产系统。

> 注意：
>
> 不要使用多个 TiCDC Changefeed 同步数据至备用集群，也不要在备用集群基础上运行另一个备用集群，否则，备用集群的数据事务完整性无法保证。

8.4.2.1.2 搭建主备集群

本文将 TiDB 主集群和备用集群分别部署在两个不同的区域（区域 1 和区域 2）。由于主备集群之间存在一定的网络延迟，TiCDC 与 TiDB 备用集群应部署在一起，以实现最好的数据同步性能。在本教程示例中，每台服务器部署一个组件节点，具体的部署拓扑如下：

区域	主机	集群	组件
区域 1	10.0.1.1/10.0.1.2/10.0.1.3	Primary	PD
区域 1	10.0.1.4/10.0.1.5	Primary	TiDB
区域 1	10.0.1.6/10.0.1.7/10.0.1.8	Primary	TiKV
区域 1	10.0.1.9	Primary	Monitor、 Grafana 或 AlterManager
区域 2	10.1.1.9/10.1.1.10	Primary	TiCDC
区域 2	10.1.1.1/10.1.1.2/10.1.1.3	Secondary	PD
区域 2	10.1.1.4/10.1.1.5	Secondary	TiDB
区域 2	10.1.1.6/10.1.1.7/10.1.1.8	Secondary	TiKV
区域 2	10.0.1.11	Secondary	Monitor、 Grafana 或 AlterManager

关于服务器配置信息，可以参考如下文档：

- TiDB 软件和硬件环境建议配置
- TiCDC 软件和硬件环境推荐配置

部署 TiDB 主集群和备用集群的详细过程，可以参考部署 TiDB 集群。

部署 TiCDC 组件需要注意的是，Secondary Cluster 和 TiCDC 需要在一起部署和管理，并且它们之间的网络需要能够连通。

- 如果需要在已有的 Primary Cluster 上部署 TiCDC，请参考部署 TiCDC 组件。
- 如果部署全新的 Primary Cluster 和 TiCDC 组件，则可以使用以下 TiUP 部署模版，并按照需要修改配置参数：

```
global:
user: "tidb"
ssh_port: 22
deploy_dir: "/tidb-deploy"
data_dir: "/tidb-data"
server_configs: {}
pd_servers:
- host: 10.0.1.1
- host: 10.0.1.2
- host: 10.0.1.3
tidb_servers:
- host: 10.0.1.4
- host: 10.0.1.5
tikv_servers:
- host: 10.0.1.6
```

```
- host: 10.0.1.7
- host: 10.0.1.8
monitoring_servers:
- host: 10.0.1.9
grafana_servers:
- host: 10.0.1.9
alertmanager_servers:
- host: 10.0.1.9
cdc_servers:
- host: 10.1.1.9
    gc-ttl: 86400
    data_dir: "/cdc-data"
    ticdc_cluster_id: "DR_TiCDC"
- host: 10.1.1.10
    gc-ttl: 86400
    data_dir: "/cdc-data"
    ticdc_cluster_id: "DR_TiCDC"
```

8.4.2.1.3 从主集群复制数据到备用集群

搭建好 TiDB 主集群和备用集群之后，需要先将主集群的数据迁移到备用集群，然后创建同步任务从主集群复制实时变更数据到备用集群。

选择外部存储

数据迁移和实时变更数据复制都需要使用外部存储。推荐使用 Amazon S3 作为存储系统。如果 TiDB 集群部署在自建机房中，则推荐以下方式：

- 搭建 MinIO 作为备份存储系统，使用 S3 协议将数据备份到 MinIO 中。
- 挂载 NFS 盘（如 NAS）到 br、TiKV 和 TiCDC 实例节点，使用 POSIX 文件系统接口将备份数据写入对应的 NFS 目录中。

下面以 MinIO 为示例，仅供参考。注意需要在区域 1 或者区域 2 中准备独立的服务器部署 MinIO。

```
wget https://dl.min.io/server/minio/release/linux-amd64/minio
chmod +x minio
### 配置访问 MinIO 的 access-key 和 access-secret-id
export HOST_IP='10.0.1.10' # 替换为实际部署 MinIO 的机器 IP 地址
export MINIO_ROOT_USER='minio'
export MINIO_ROOT_PASSWORD='miniostorage'
### 创建 TiCDC redo log 和 backup 数据保存的目录，其中 redo、backup 为 bucket 名字
mkdir -p data/redo
mkdir -p data/backup
### 启动 MinIO，暴露端口在 6060
nohup ./minio server ./data --address :6060 &
```

上述命令启动了一个单节点的 MinIO server 模拟 S3 服务，相关参数为：

- endpoint：http://10.0.1.10:6060/
- access-key：minio
- secret-access-key：miniostorage
- bucket：redo/backup

其访问链接为：

```
s3://backup?access-key=minio&secret-access-key=miniostorage&endpoint=http://10.0.1.10:6060&force-
    ↪ path-style=true
```

数据迁移

在 TiDB 主备集群之间使用TiDB 备份恢复功能 BR 进行数据迁移。

1. 关闭垃圾回收机制 (GC)。为了保证增量迁移过程中新写入的数据不丢失，在开始备份之前，需要关闭上游集群的 GC 机制，以确保系统不再清理历史数据。

   执行如下命令关闭 GC：

   ```
   SET GLOBAL tidb_gc_enable=FALSE;
   ```

   查询 tidb_gc_enable 的取值，以确认 GC 是否已关闭：

   ```
   SELECT @@global.tidb_gc_enable;
   ```

   输出结果为 0 表明 GC 已关闭：

   ```
   +--------------------------+
   | @@global.tidb_gc_enable |
   +--------------------------+
   |                        0 |
   +--------------------------+
   1 row in set (0.00 sec)
   ```

   > 注意：
   > 在生产集群中，关闭 GC 机制和备份操作会一定程度上降低集群的读性能。建议在业务低峰期进行备份，并设置合适的 RATE_LIMIT 限制备份操作对线上业务的影响。

2. 备份数据。在 TiDB 主集群中执行 BACKUP 语句备份数据：

   ```
   BACKUP DATABASE * TO ``s3://backup?access-key=minio&secret-access-key=miniostorage&endpoint=
       ↪ http://10.0.1.10:6060&force-path-style=true``;
   ```

   ```
   +----------------------+----------+--------------------+---------------------+--------------------
       ↪
   | Destination          | Size     | BackupTS           | Queue Time          | Execution
       ↪ Time       |
   ```

```
+----------------------+----------+--------------------+--------------------+----------------------+
    ↪
| s3://backup          | 10315858 | 431434047157698561 | 2022-02-25 19:57:59 | 2022-02-25
    ↪ 19:57:59 |
+----------------------+----------+--------------------+--------------------+----------------------+
    ↪
1 row in set (2.11 sec)
```

备份语句提交成功后，TiDB 会返回关于备份数据的元信息，这里需要重点关注 BackupTS，它意味着该时间点之前的数据会被备份。本文后续步骤中，使用 BackupTS 作为实时变更数据复制的起始时间点。

3. 恢复数据。在 TiDB 备用集群中执行 RESTORE 语句恢复数据：

```
RESTORE DATABASE * FROM ''s3://backup?access-key=minio&secret-access-key=miniostorage&
    ↪ endpoint=http://10.0.1.10:6060&force-path-style=true`';
```

```
+----------------------+----------+----------+--------------------+--------------------+
| Destination          | Size     | BackupTS | Queue Time         | Execution Time     |
+----------------------+----------+----------+--------------------+--------------------+
| s3://backup          | 10315858 | 0        | 2022-02-25 20:03:59 | 2022-02-25 20:03:59 |
+----------------------+----------+----------+--------------------+--------------------+
1 row in set (41.85 sec)
```

复制实时变更数据

完成以上数据迁移的操作步骤后，从备份的 BackupTS 时间点开始同步主集群增量变更数据到备用集群。

1. 创建 TiCDC 同步任务 Changefeed。

   创建 Changefeed 配置文件并保存为 changefeed.toml。

```
[consistent]
# eventual consistency：使用 redo log，提供上游灾难情况下的最终一致性。
level = "eventual"
# 单个 redo log 文件大小，单位 MiB，默认值 64，建议该值不超过 128。
max-log-size = 64
# 刷新或上传 redo log 至 S3 的间隔，单位毫秒，默认 1000，建议范围 500-2000。
flush-interval = 2000
# 存储 redo log 的地址
storage = "s3://redo?access-key=minio&secret-access-key=miniostorage&endpoint=http
    ↪ ://10.0.1.10:6060&force-path-style=true"
```

   在主用集群中，执行以下命令创建从主用集群到备用集群的同步链路：

```
tiup cdc cli changefeed create --server=http://10.1.1.9:8300 --sink-uri="mysql://{username
    ↪ }:{password}@10.1.1.4:4000" --changefeed-id="dr-primary-to-secondary" --start-ts
    ↪ ="431434047157698561"
```

更多关于 Changefeed 的配置，请参考TiCDC Changefeed 配置参数。

2. 查询 Changefeed 是否正常运行。使用 `changefeed query` 命令可以查询特定同步任务（对应某个同步任务的信息和状态），指定 `--simple` 或 `-s` 参数会简化输出，提供基本的同步状态和 checkpoint 信息。不指定该参数会输出详细的任务配置、同步状态和同步表信息。

```
tiup cdc cli changefeed query -s --server=http://10.1.1.9:8300 --changefeed-id="dr-primary-
    ↪ to-secondary"
```

```
{
"state": "normal",
"tso": 431434047157998561,  # changefeed 已经同步到的时间点
"checkpoint": "2020-08-27 10:12:19.579", # TSO 对应的物理时间点
"error": null
}
```

3. 重新开启 GC。

   TiCDC 可以保证未同步的历史数据不会被回收。因此，创建完从主集群到备用集群的 Changefeed 之后，就可以执行如下命令恢复集群的垃圾回收功能。

执行如下命令打开 GC：

````
```sql
SET GLOBAL tidb_gc_enable=TRUE;
```

查询 `tidb_gc_enable` 的取值，判断 GC 是否已开启：

```sql
SELECT @@global.tidb_gc_enable;
```

结果输出 `1` 表明 GC 已开启：

```
+--------------------------+
| @@global.tidb_gc_enable |
+--------------------------+
|                        1 |
+--------------------------+
1 row in set (0.00 sec)
```
````

8.4.2.1.4 主备集群状态监控

TiDB 目前还没有提供 DR Dashboard，你可以通过以下 Dashboard 了解 TiDB 主备集群的状态，从而决定是否需要进行容灾切换：

- TiDB 集群运行状态 Dashboard
- Changefeed 运行状态

8.4.2.1.5 容灾切换

本部分介绍容灾演练，遇到真正灾难时的主备切换，以及重建灾备集群的步骤。

计划中的主备切换

定期对非常重要的业务系统进行容灾演练，检验系统的可靠性是非常有必要的。下面是容灾演练推荐的操作步骤，因为没有考虑演练中业务写入是否为模拟、业务访问数据库是否使用 proxy 服务等，与实际的演练场景会有出入，请根据你的实际情况进行修改。

1. 停止主集群上的业务写入。
2. 业务写入完全停止后，查询 TiDB 集群当前最新的 TSO (Position)：

```
mysql> show master status;
+-------------+--------------------+--------------+------------------+-------------------+
| File        | Position           | Binlog_Do_DB | Binlog_Ignore_DB | Executed_Gtid_Set |
+-------------+--------------------+--------------+------------------+-------------------+
| tidb-binlog | 438223974697009153 |              |                  |                   |
+-------------+--------------------+--------------+------------------+-------------------+
1 row in set (0.33 sec)
```

3. 轮询 Changefeed `dr-primary-to-secondary` 的同步位置时间点 TSO 直到满足 `TSO >= Position`。

```
tiup cdc cli changefeed query -s --server=http://10.1.1.9:8300 --changefeed-id="dr-primary-
    ↪ to-secondary"

{
    "state": "normal",
    "tso": 438224029039198209,  # Changefeed 已经同步到的时间点
    "checkpoint": "2022-12-22 14:53:25.307", # TSO 对应的物理时间点
    "error": null
}
```

4. 停止 Changefeed `dr-primary-to-secondary`。通过删除 Changefeed 的方式，暂停 Changefeed `dr-primary-to-secondary`：

```
tiup cdc cli changefeed remove --server=http://10.1.1.9:8300 --changefeed-id="dr-primary-to-
    ↪ secondary"
```

5. 创建 Changefeed `dr-secondary-to-primary`。不需要指定 Changefeed `start-ts` 参数，Changefeed 从当前时间开始同步即可。
6. 修改业务应用的数据库访问配置，并重启业务应用，使得业务访问备用集群。
7. 检查业务状态是否正常。

容灾演练后，再重复一遍以上步骤，即可恢复原有的系统主备配置。

真正灾难中主备切换

当发生真正的灾难，比如主集群所在区域停电，主备集群的同步链路可能会突然中断，从而导致备用集群数据处于事务不一致的状态。

1. 恢复备用集群到事务一致的状态。在区域 2 的任意 TiCDC 节点执行以下命令，以向备用集群重放 redo log，使备用集群达到最终一致性状态：

```
tiup cdc redo apply --storage "s3://redo?access-key=minio&secret-access-key=miniostorage&
    ↪ endpoint=http://10.0.1.10:6060&force-path-style=true" --tmp-dir /tmp/redo --sink-uri
    ↪  "mysql://{username}:{password}@10.1.1.4:4000"
```

   命令中参数描述如下：

   - `--storage`：指定 redo log 所在的 S3 位置
   - `--tmp-dir`：为从 S3 下载 redo log 的缓存目录
   - `--sink-uri`：指定备份集群的地址

2. 修改业务应用的数据库访问配置，并重启业务应用，使得业务访问备用集群。
3. 检查业务状态是否正常。

灾难后重建主备集群

当 TiDB 主集群所遭遇的灾难解决后，或者主集群暂时不能恢复，此时 TiDB 集群是脆弱的，只有一个备用集群临时作为新的主集群提供服务。为了维持系统的可靠性，需要重建灾备集群保护系统的可靠性。

目前，重建 TiDB 主备集群，通用的方案是重新部署一个新的集群，组成新的容灾主备集群。操作请参考：

1. 搭建主备集群。
2. 从主集群复制数据到备用集群。
3. 完成以上操作步骤后，如果你希望新集群成为主集群，那么请参考主从切换。

> 注意：
>
> 如果在业务上能够修正灾难发生后主集群和备用集群的数据不一致的问题，那么也可以使用修正后的集群重建主备集群，而不需要重建新集群。

8.4.2.1.6 在备用集群上进行业务查询

在主备集群容灾场景中，将备用集群作为只读集群来运行一些延迟不敏感的查询是常见的需求，TiDB 主备集群容灾方案也提供了这种功能。

创建 Changefeed 时，你只需要在配置文件中开启 Syncpoint 功能，Changefeed 就会定期 (`sync-point-interval`) 在备用集群中通过执行 `SET GLOBAL tidb_external_ts = @@tidb_current_ts` 设置已复制完成的一致性快照点。

当业务需要从备用集群查询数据的时候，在业务应用中设置 `SET GLOBAL|SESSION tidb_enable_external_ts_read = ON;` 就可以在备用集群上获得事务状态完成的数据。

```
### 从 v6.4.0 开始支持，使用 Syncpoint 功能需要同步任务拥有下游集群的 SYSTEM_VARIABLES_ADMIN 或者
    ↪  SUPER 权限
enable-sync-point = true

### 记录主集群和备用集群一致快照点的时间间隔，它也代表能读到完整事务的最大延迟时间，
    ↪ 比如在备用集群读取到主集群两分钟之前的事务数据
### 配置格式为 h m s，例如 "1h30m30s"。默认值为 10m，最小值为 30s
sync-point-interval = "10m"

### Syncpoint 功能在下游表中保存的数据的时长，超过这个时间的数据会被清理
### 配置格式为 h m s，例如 "24h30m30s"。默认值为 24h
sync-point-retention = "1h"

[consistent]
### eventual consistency： 使用 redo log，提供上游灾难情况下的最终一致性。
level = "eventual"
### 单个 redo log 文件大小，单位 MiB，默认值 64，建议该值不超过 128。
max-log-size = 64
### 刷新或上传 redo log 至 S3 的间隔，单位毫秒，默认 1000，建议范围 500-2000。
flush-interval = 2000
### 存储 redo log
storage = "s3://redo?access-key=minio&secret-access-key=miniostorage&endpoint=http
    ↪ ://10.0.1.10:6060&force-path-style=true"
```

> **注意：**
>
> 在主备集群容灾架构中，每个备用集群只能被一个 Changefeed 同步数据，否则就无法保证备用集群的事务完整性。

8.4.2.1.7 在主备集群之间进行双向复制

在主备集群容灾场景中，部分用户希望让两个区域的 TiDB 集群互为灾备集群：用户的业务流量按其区域属性写入对应的 TiDB 集群，同时两套 TiDB 集群备份对方集群的数据。

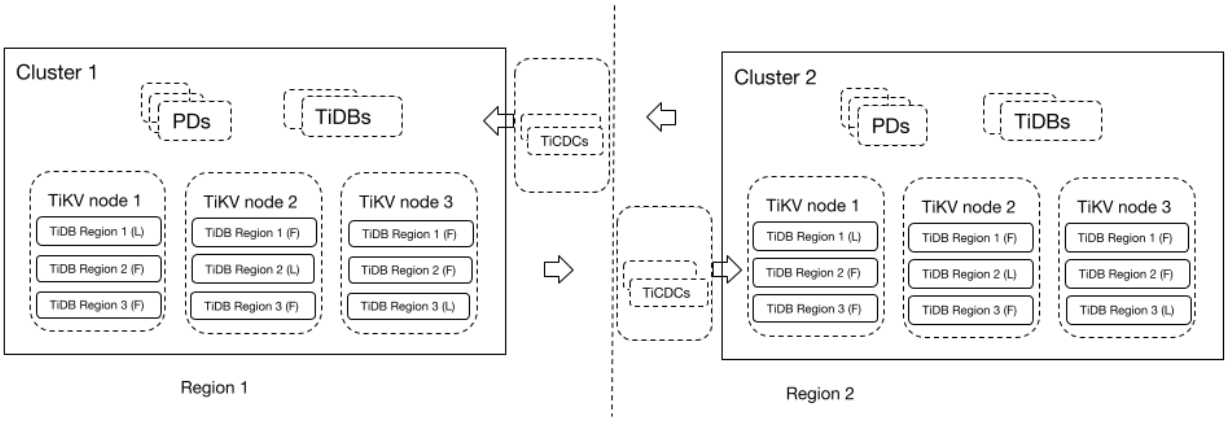


图 54: TiCDC bidirectional replication

在双向复制容灾集群方案中，两个区域的 TiDB 集群互相备份对方的数据，使得它们可以在故障发生时互为灾备集群。这种方案既能满足安全性和可靠性的需求，同时也能保证数据库的写入性能。在计划中的主备切换场景中，不需要停止正在运行的 Changefeed 和启动新的 Changefeed 等操作，在运维上也更加简单。

搭建双向容灾复制集群的步骤，请参考教程TiCDC 双向复制。

8.4.2.2 常见问题处理

以上任何步骤遇到问题，可以先通过TiDB FAQ 查找问题的处理方法。如果问题仍不能解决，请在 TiDB 项目中提 issue。

8.4.3 基于多副本的单集群容灾方案

本文介绍了基于多副本的单集群容灾方案，文档内容组织如下：

- 方案简介
- 搭建集群
- 配置副本
- 监控集群
- 容灾切换

8.4.3.1 简介

对于重要的生产系统，很多用户需要能够实现区域级别的容灾，并且做到 RPO = 0 和分钟级别的 RTO。TiDB 作为基于 Raft 协议的分布式数据库，其自带的多副本特性可以用于支持区域级别的容灾目标，并同时确保数据的一致性和高可用性。而同区域可用区 (Available Zone, AZ) 之间的网络延迟相对较小，可以把业务流量同时派发到同区域两个 AZ，并通过控制 Region Leader 和 PD Leader 分布实现同区域 AZ 共同负载业务流量。

8.4.3.2 搭建集群和配置副本

在这一部分当中，会以一个 5 副本的集群为例，演示如何使用 TiUP 创建一个跨 3 个区域的集群，以及如何控制数据和 PD 的分布位置，从而达到容灾的目的。

在下面的示例中，TiDB 集群的区域 1 作为 primary region，区域 2 作为 secondary region，而区域 3 则作为投票使用的第三个区域，一共包含 5 个副本。同理，PD 集群也包含了 5 个副本，其功能和 TiDB 集群的功能基本一致。

1. 创建类似于以下的集群拓扑文件：

```
global:
  user: "root"
  ssh_port: 22
  deploy_dir: "/data/tidb_cluster/tidb-deploy"
  data_dir: "/data/tidb_cluster/tidb-data"

server_configs:
  tikv:
    server.grpc-compression-type: gzip
  pd:
    replication.location-labels:  ["Region","AZ"] # PD 会根据 TiKV 节点的 Region 和 AZ
        ↪ 配置来进行副本的调度。

pd_servers:
  - host: tidb-dr-test1
    name: "pd-1"
  - host: tidb-dr-test2
    name: "pd-2"
  - host: tidb-dr-test3
    name: "pd-3"
  - host: tidb-dr-test4
    name: "pd-4"
  - host: tidb-dr-test5
    name: "pd-5"

tidb_servers:
  - host: tidb-dr-test1
  - host: tidb-dr-test3

tikv_servers:  # 在 TiKV 节点中通过 labels 选项来对每个 TiKV 节点所在的 Region 和 AZ
    ↪ 进行标记
  - host: tidb-dr-test1
    config:
      server.labels: { Region: "Region1", AZ: "AZ1" }
  - host: tidb-dr-test2
    config:
      server.labels: { Region: "Region1", AZ: "AZ2" }
```

```
  - host: tidb-dr-test3
    config:
      server.labels: { Region: "Region2", AZ: "AZ3" }
  - host: tidb-dr-test4
    config:
      server.labels: { Region: "Region2", AZ: "AZ4" }
  - host: tidb-dr-test5
    config:
      server.labels: { Region: "Region3", AZ: "AZ5" }

      raftstore.raft-min-election-timeout-ticks: 1000
      raftstore.raft-max-election-timeout-ticks: 1200

monitoring_servers:
  - host: tidb-dr-test2

grafana_servers:
  - host: tidb-dr-test2

alertmanager_servers:
  - host: tidb-dr-test2
```

在上面的配置中，使用了以下一系列配置来针对跨区域容灾场景进行优化：

- 使用 `server.grpc-compression-type`：gzip 启用 TiKV 之间的消息压缩，从而降低网络流量。
- 使 用 `raftstore.raft-min-election-timeout-ticks` 和 `raftstore.raft-max-election-timeout-ticks` 延长区域 3 参加选举的时间，从而避免该区域中的副本被选举为主节点。

2. 使用上面的配置文件创建集群：

```
tiup cluster deploy drtest v6.4.0 ./topo.yaml
tiup cluster start drtest --init
tiup cluster display drtest
```

对集群的副本数和 Leader 限制进行配置：

```
tiup ctl:v6.4.0 pd config set max-replicas 5
tiup ctl:v6.4.0 pd config set label-property reject-leader Region Region3

# 下面的步骤用于向集群中添加一些测试数据，可选
tiup bench tpcc  prepare -H 127.0.0.1 -P 4000 -D tpcc --warehouses 1
```

指定 PD leader 的优先级:

```
tiup ctl:v6.4.0 pd member leader_priority  pd-1 4
tiup ctl:v6.4.0 pd member leader_priority  pd-2 3
tiup ctl:v6.4.0 pd member leader_priority  pd-3 2
tiup ctl:v6.4.0 pd member leader_priority  pd-4 1
```

```
tiup ctl:v6.4.0 pd member leader_priority  pd-5 0
```

> **注意：**
>
> 优先级数值越大的节点成为 leader 的可能性越高。

3. 创建 placement rule，并将测试表的主副本固定在区域 1：

```
-- 创建两个 placement rules，第一个是区域 1 作为主区域，在系统正常时使用，第二个是区域 2
    ↪ 作为备区域。
-- 作为主区域，当区域 1 出现问题时，区域 2 会作为主区域。
MySQL [(none)]> CREATE PLACEMENT POLICY primary_rule_for_region1 PRIMARY_REGION="Region1"
    ↪ REGIONS="Region1, Region2,Region3";
MySQL [(none)]> CREATE PLACEMENT POLICY secondary_rule_for_region2 PRIMARY_REGION="Region2"
    ↪ REGIONS="Region1,Region2,Region3";

-- 将刚刚创建的规则 primary_rule_for_region1 应用到对应的用户表上。
ALTER TABLE tpcc.warehouse PLACEMENT POLICY=primary_rule_for_region1;
ALTER TABLE tpcc.district PLACEMENT POLICY=primary_rule_for_region1;

-- 说明：请根据需要修改上面的数据库名称、表名和 placement rule 的名称。

-- 使用类似下面的查询，用户可以查看每个区域包含的 leader 数量，以确认 leader 迁移是否完成。
SELECT STORE_ID, address, leader_count, label FROM TIKV_STORE_STATUS ORDER BY store_id;
```

下面的语句可以产生一个 SQL 脚本，把所有非系统 schema 中的表的 leader 都设置到特定的区域上：

```
SET @region_name=primary_rule_for_region1;
SELECT concat('ALTER TABLE ', table_schema, '.', table_name, ' PLACEMENT POLICY=',
    ↪ @region_name, ';') FROM information_schema.tables WHERE table_schema NOT IN ('
    ↪ METRICS_SCHEMA', 'PERFORMANCE_SCHEMA', 'INFORMATION_SCHEMA','mysql');
```

8.4.3.3 监控集群

对于部署的集群，你可以通过访问集群中的 Grafana 地址或者 TiDB Dashboard 组件来对集群中的各个 TiKV、TiDB 和 PD 组件的各种性能指标进行监控。根据组件的状态，确定是否进行容灾切换。详细信息，请参考如下文档：

- TiDB 重要监控指标详解
- TiKV 监控指标详解
- PD 重要监控指标详解
- TiDB Dashboard 监控页面

8.4.3.4 容灾切换

本部分介绍容灾切换，包括计划内切换和计划外切换。

8.4.3.4.1 计划内切换

指根据维护需要进行的主备区域切换，可用于验证容灾系统是否可以正常工作。本部分介绍如何在计划内切换主备区域。

1. 执行如下命令，将所有用户表和 PD Leader 都切换到区域 2：

```
-- 将之前创建的规则 secondary_rule_for_region2 应用到对应的用户表上。
ALTER TABLE tpcc.warehouse PLACEMENT POLICY=secondary_rule_for_region2;
ALTER TABLE tpcc.district PLACEMENT POLICY=secondary_rule_for_region2;
```

说明：请根据需要修改上面的数据库名称、表名和 placement rule 的名称。

执行如下命令，调低区域 1 的 PD 节点的优先级，并调高区域 2 的 PD 节点的优先级。

```
tiup ctl:v6.4.0 pd member leader_priority pd-1 2
tiup ctl:v6.4.0 pd member leader_priority pd-2 1
tiup ctl:v6.4.0 pd member leader_priority pd-3 4
tiup ctl:v6.4.0 pd member leader_priority pd-4 3
```

2. 观察 Grafana 中 PD 和 TiKV 部分中的内容，确保 PD 的 Leader 和用户表的 Leader 已经迁移到对应的区域。另外，切换回原有区域的步骤与上面的步骤基本相同，本文不做过多的描述。

8.4.3.4.2 计划外切换

计划外切换，指灾难发生时的主备区域切换，或者为了验证容灾系统的有效性，而模拟灾难发生时的主备区域切换。

1. 执行类似下面的命令终止区域 1 上所有的 TiKV、 TiDB 和 PD 节点:

```
tiup cluster stop drtest -N tidb-dr-test1:20160,tidb-dr-test2:20160,tidb-dr-test1:2379,tidb-
    ↪ dr-test2:2379
```

2. 运行类似于下面的命令切换用户表的 leader 到区域 2:

```
-- 将之前创建的规则 secondary_rule_for_region2 应用到对应的用户表上。
ALTER TABLE tpcc.warehouse PLACEMENT POLICY=secondary_rule_for_region2;
ALTER TABLE tpcc.district PLACEMENT POLICY=secondary_rule_for_region2;

---可以使用类似下面的查询查看每个区域包含的 leader 数量，以确认 leader 迁移是否完成。
SELECT STORE_ID, address, leader_count, label FROM TIKV_STORE_STATUS ORDER BY store_id;
```

当区域 1 恢复正常之后，可以使用类似于上面的命令将用户表的 leader 重新切换到区域 1。

8.4.4 基于备份与恢复的容灾方案

TiDB 集群自身的多副本特性可以让其容忍单个机房或地理区域的故障，并继续向外提供服务。在更大范围的自然灾害、软件漏洞、硬件故障、病毒攻击或常见的人为误操作的情况下，TiDB 的备份与恢复功能可以将数据备份到独立的灾备存储设备中，以保护用户数据不受损坏。与其他方案相比，备份与恢复功能具有灵活性、可靠性、可恢复性和成本效益等优势：

- 灵活性：可以在不同的时间点进行备份，并且可以根据需要调整备份频率。这使得备份与恢复功能更加灵活，能够更好地适应不同的业务场景。
- 可靠性：备份与恢复功能通常会将备份数据保存在独立的存储设备上，使得数据安全得到进一步升级。
- 可恢复性：任何意外情况导致的原始数据丢失或损坏，都可以通过恢复备份数据的方式来恢复。这使得备份恢复功能具有很高的可恢复性，能够保证数据库的正常使用。
- 成本效益：备份恢复功能通常具有较低的成本，可以在不增加太多费用的情况下实现对数据库的安全保护。

总的来说，备份恢复功能是数据安全的最后一道防线，可以在不增加太多成本的情况下，提高数据库的安全性和可靠性。它能够在各种意外情况发生时保护数据，使你能够安心地使用数据库，而不必担心数据丢失或损坏的风险。

8.4.4.1 进行备份恢复

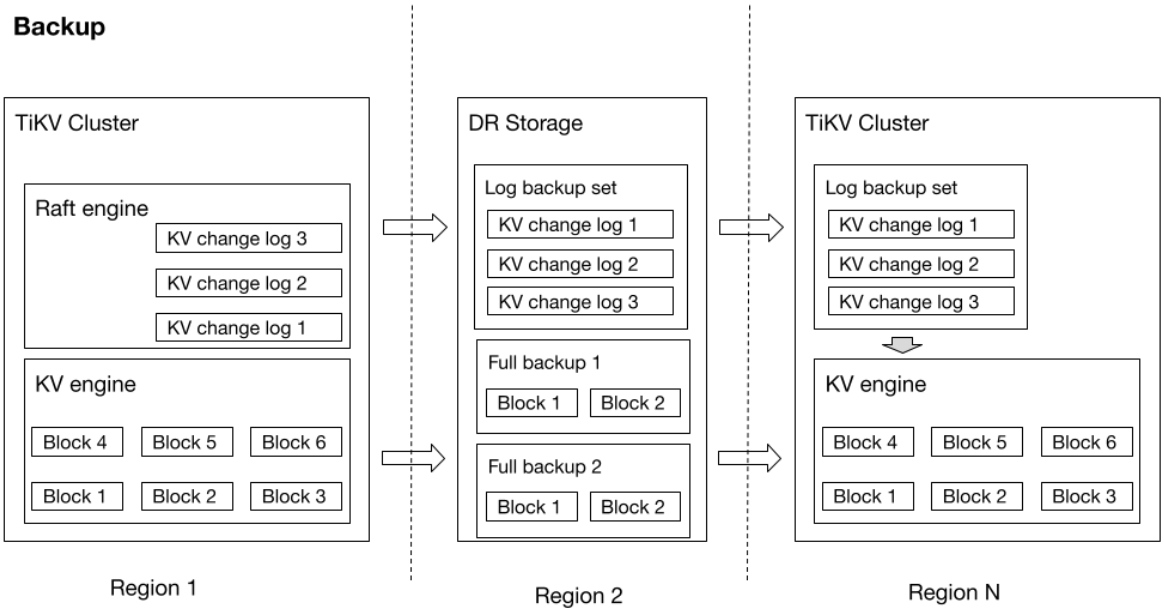


图 55: BR log backup and PITR architecture

按照上述架构，你可以将数据备份到其他区域的灾备存储设备中，并在需要时将数据恢复回来。这样，系统就能够容忍单个区域的故障，并且 Recovery Point Objective (RPO) 可以达到 5 分钟，Recovery Time Objective (RTO) 通常在几十分钟到数小时之间。但是，如果数据库尺寸较大，RTO 时间可能会更长。

此外，TiDB 还提供了基于块存储快照技术的备份和恢复特性，可以将集群的恢复时间缩短到小时级别甚至 1 小时以内。TiDB 也在不断完善和优化基于存储快照技术的备份和恢复能力，以提供更好的服务。

TiDB 提供了丰富的文档，帮助你了解如何在容灾场景下使用备份和恢复功能。其中，

- TiDB 备份和恢复功能使用概述提供了备份与恢复的使用指引，包含备份策略及备份数据存放组织等。
- TiDB 备份恢复常见问题列出了使用备份与恢复功能时可能遇到的问题，以及解决方案。
- TiDB 备份与恢复功能架构描述了 TiDB 备份与恢复功能的设计架构，包括备份和恢复的流程，以及备份文件设计。

8.5 使用资源管控 (Resource Control) 实现资源隔离

> **警告：**
>
> 资源管控是 TiDB 在 v7.0.0 是实验特性，其语法或者行为表现在 GA 前可能会发生变化。

使用资源管控特性，集群管理员可以定义资源组 (Resource Group)，通过资源组限定读写的配额。

TiDB 资源管控特性提供了两层资源管理能力，包括在 TiDB 层的流控能力和 TiKV 层的优先级调度的能力。两个能力可以单独或者同时开启，详情请参见参数组合效果表。将用户绑定到某个资源组后，TiDB 层会根据用户所绑定资源组设定的读写配额对用户的读写请求做流控，TiKV 层会根据读写配额映射的优先级来对请求做调度。通过流控和调度这两层控制，可以实现应用的资源隔离，满足服务质量 (QoS) 要求。

- TiDB 流控：TiDB 流控使用令牌桶算法 做流控。如果桶内令牌数不够，而且资源组没有指定 BURSTABLE 特性，属于该资源组的请求会等待令牌桶回填令牌并重试，重试可能会超时失败。
- TiKV 调度：你可以为资源组设置绝对优先级 (PRIORITY)，不同的资源按照 PRIORITY 的设置进行调度，PRIORITY 高的任务会被优先调度。如果没有设置绝对优先级 (PRIORITY)，TiKV 会将资源组的 RU_PER_SEC 取值映射成各自资源组读写请求的优先级，并基于各自的优先级在存储层使用优先级队列调度处理请求。

8.5.1 使用场景

资源管控特性的引入对 TiDB 具有里程碑的意义。它能够将一个分布式数据库集群划分成多个逻辑单元，即使个别单元对资源过度使用，也不会挤占其他单元所需的资源。利用该特性：

- 你可以将多个来自不同系统的中小型应用合入一个 TiDB 集群中，个别应用的负载升高，不会影响其他业务的正常运行。而在系统负载较低的时候，繁忙的应用即使超过设定的读写配额，也仍然可以被分配到所需的系统资源，达到资源的最大化利用。
- 你可以选择将所有测试环境合入一个集群，或者将消耗较大的批量任务编入一个单独的资源组，在保证重要应用获得必要资源的同时，提升硬件利用率，降低运行成本。

此外，合理利用资源管控特性可以减少集群数量，降低运维难度及管理成本。

8.5.2 使用限制

目前，资源管控特性具有以下限制:

- 只支持对前台客户发起的读写请求做限流和调度，不支持对 DDL 以及 Auto Analyze 等后台任务的限流和调度。
- 资源管控将带来额外的调度开销。因此，开启该特性后，性能可能会有轻微下降。

8.5.3 什么是 Request Unit (RU)

Request Unit (RU) 是 TiDB 对 CPU、IO 等系统资源的统一抽象的计量单位，用于表示对数据库的单个请求消耗的资源量。请求消耗的 RU 数量取决于多种因素，例如操作类型或正在检索或修改的数据量。目前，RU 包含以下资源的统计信息：

```
<thead>
    <tr>
        <th>资源类型</th>
        <th>RU 消耗</th>
    </tr>
</thead>
<tbody>
    <tr>
        <td rowspan="3">Read</td>
        <td>2 storage read batches 消耗 1 RU</td>
    </tr>
    <tr>
        <td>8 storage read requests 消耗 1 RU</td>
    </tr>
    <tr>
        <td>64 KiB read request payload 消耗 1 RU</td>
    </tr>
    <tr>
        <td rowspan="3">Write</td>
        <td>1 storage write batch 消耗 1 RU * 副本数</td>
    </tr>
    <tr>
        <td>1 storage write request 消耗 1 RU</td>
    </tr>
    <tr>
        <td>1 KiB write request payload 消耗 1 RU</td>
    </tr>
    <tr>
        <td>SQL CPU</td>
        <td> 3 ms 消耗 1 RU</td>
    </tr>
</tbody>
```

注意：

- 每个写操作最终都被会复制到所有副本（TiKV 默认 3 个数据副本），并且每次复制都被认为是一个不同的写操作。
- 除了用户执行的查询之外，RU 还可以被后台任务消耗，例如自动统计信息收集。
- 上表只列举了本地部署的 TiDB 计算 RU 时涉及的相关资源，其中不包括网络和存储部分。TiDB Serverless 的 RU 可参考 TiDB Serverless Pricing Details。

8.5.4 估算 SQL 所消耗的 RU

你可以通过`EXPLAIN ANALYZE` 语句获取到 SQL 执行时所消耗的 RU。注意 RU 的大小会受缓存的影响（比如下推计算结果缓存），多次执行同一条 SQL 所消耗的 RU 可能会有不同。因此这个 RU 值并不代表每次执行的精确值，但可以作为估算的参考。

8.5.5 相关参数

资源管控特性引入了两个新的全局开关变量：

- TiDB: 通过配置全局变量`tidb_enable_resource_control` 控制是否打开资源组流控。
- TiKV: 通过配置参数`resource-control.enabled` 控制是否使用基于资源组配额的请求调度。

这两个参数的组合效果见下表：

resource-control.enabled	tidb_enable_resource_control= ON	tidb_enable_resource_control= OFF
resource-control.enabled= true	流控和调度（推荐组合）	无效配置
resource-control.enabled= false	仅流控（不推荐）	特性被关闭

关于资源管控实现机制及相关参数的详细介绍，请参考 RFC: Global Resource Control in TiDB。

8.5.6 使用方法

8.5.6.1 管理资源组

创建、修改、删除资源组，需要拥有 SUPER 或者 RESOURCE_GROUP_ADMIN 权限。

你可以通过`CREATE RESOURCE GROUP` 在集群中创建资源组。

对于已有的资源组，可以通过`ALTER RESOURCE GROUP` 修改资源组的读写配额，对资源组的配额修改会立即生效。

可以使用DROP RESOURCE GROUP 删除资源组。

8.5.6.2 创建资源组

下面举例说明如何创建资源组。

1. 创建 rg1 资源组，RU 的回填速度是每秒 500 RU，并且允许这个资源组的应用超额占用资源。

```
CREATE RESOURCE GROUP IF NOT EXISTS rg1 RU_PER_SEC = 500 BURSTABLE;
```

2. 创建 rg2 资源组，RU 的回填速度是每秒 600 RU。在系统资源充足的时候，不允许这个资源组的应用超额占用资源。

```
CREATE RESOURCE GROUP IF NOT EXISTS rg2 RU_PER_SEC = 600;
```

3. 创建 rg3 资源组，设置绝对优先级为 HIGH。绝对优先级目前支持 LOW|MEDIUM|HIGH，资源组的默认绝对优先级为 MEDIUM。

```
CREATE RESOURCE GROUP IF NOT EXISTS rg3 RU_PER_SEC = 100 PRIORITY = HIGH;
```

8.5.6.3 绑定资源组

TiDB 支持如下三个级别的资源组设置：

- 用户级别。通过CREATE USER 或ALTER USER 语句将用户绑定到特定的资源组。绑定后，对应的用户新创建的会话会自动绑定对应的资源组。
- 会话级别。通过SET RESOURCE GROUP 设置当前会话使用的资源组。
- 语句级别。通过RESOURCE_GROUP() Optimizer Hint 设置当前语句使用的资源组。

8.5.6.3.1 将用户绑定到资源组

下面的示例创建一个用户 usr1 并将其绑定到资源组 rg1。其中 rg1 为创建资源组示例中创建的资源组。

```
CREATE USER 'usr1'@'%' IDENTIFIED BY '123' RESOURCE GROUP rg1;
```

下面示例使用 ALTER USER 将用户 usr2 绑定到资源组 rg2。其中 rg2 为创建资源组示例中创建的资源组。

```
ALTER USER usr2 RESOURCE GROUP rg2;
```

绑定用户后，用户新建立的会话对资源的占用会受到指定用量 (RU) 的限制。如果系统负载比较高，没有多余的容量，用户 usr2 的资源消耗速度会被严格控制不超过指定用量。由于 usr1 绑定的 rg1 配置了 BURSTABLE，所以 usr1 消耗速度允许超过指定用量。

如果资源组对应的请求太多导致资源组的资源不足，客户端的请求处理会发生等待。如果等待时间过长，请求会报错。

注意：

- 使用 CREATE USER 或者 ALTER USER 将用户绑定到资源组后，只会对该用户新建的会话生效，不会对该用户已有的会话生效。
- TiDB 集群在初始化时会自动创建 default 资源组，其 RU_PER_SEC 的默认值为 UNLIMITED (等同于 INT 类型最大值，即 2147483647)，且为 BURSTABLE 模式。对于没有绑定资源组的语句会自动绑定至此资源组。此资源组不支持删除，但允许修改其 RU 的配置。

8.5.6.3.2 将当前会话绑定到资源组

通过把当前会话绑定到资源组，会话对资源的占用会受到指定用量 (RU) 的限制。

下面的示例将当前的会话绑定至资源组 rg1。

```
SET RESOURCE GROUP rg1;
```

8.5.6.3.3 将语句绑定到资源组

通过在 SQL 语句中添加RESOURCE_GROUP(resource_group_name) Hint，可以将该语句绑定到指定的资源组。此 Hint 支持 SELECT、 INSERT、 UPDATE、 DELETE 四种语句。

示例：

```
SELECT /*+ RESOURCE_GROUP(rg1) */ * FROM t limit 10;
```

8.5.7 关闭资源管控特性

1. 执行以下命令关闭资源管控特性：

   ```
   SET GLOBAL tidb_enable_resource_control = 'OFF';
   ```

2. 将 TiKV 参数resource-control.enabled 设为 false，关闭按照资源组配额调度。

8.5.8 监控与图表

TiDB 会定时采集资源管控的运行时信息，并在 Grafana 的 Resource Control Dashboard 中提供了相关指标的可视化图表。指标详情参见Resource Control 监控指标详解 。

TiKV 中也记录了来自于不同资源组的请求 QPS，详见TiKV 监控指标详解

8.5.9 工具兼容性

资源管控目前为实验特性，不影响数据导入导出以及其他同步工具的正常使用，BR、 TiDB Lightning、 TiCDC 等工具不支持对资源管控相关 DDL 的处理，这些工具的资源消耗也不受资源管控的限制。

8.5.10 另请参阅

- CREATE RESOURCE GROUP
- ALTER RESOURCE GROUP
- DROP RESOURCE GROUP
- RESOURCE GROUP RFC

8.6 时区支持

TiDB 使用的时区由 time_zone 全局变量和 session 变量决定。time_zone 的默认值是 System，System 对应的实际时区在 TiDB 集群 bootstrap 初始化时设置。具体逻辑如下：

- 优先使用 TZ 环境变量
- 如果失败，则从 /etc/localtime 的实际软链地址提取。
- 如果上面两种都失败则使用 UTC 作为系统时区。

在运行过程中可以修改全局时区：

```
SET GLOBAL time_zone = timezone;
```

TiDB 还可以通过设置 session 变量 time_zone 为每个连接维护各自的时区。默认条件下，这个值取的是全局变量 time_zone 的值。修改 session 使用的时区：

```
SET time_zone = timezone;
```

使用以下 SQL 语句查看当前全局时区、客户端时区和系统时区的值：

```
SELECT @@global.time_zone, @@session.time_zone, @@global.system_time_zone;
```

设置 time_zone 的值的格式：

- ‘SYSTEM’ 表明使用系统时间
- 相对于 UTC 时间的偏移，比如 ‘+10:00’ 或者 ‘-6:00’
- 某个时区的名字，比如 ‘Europe/Helsinki’，‘US/Eastern’ 或 ‘MET’

NOW() 和 CURTIME() 的返回值都受到时区设置的影响。

> 注意：
>
> 只有 Timestamp 数据类型的值是受时区影响的。可以理解为，Timestamp 数据类型的实际表示使用的是（字面值 + 时区信息）。其它时间和日期类型，比如 Datetime/Date/Time 是不包含时区信息的，所以也不受到时区变化的影响。

```
create table t (ts timestamp, dt datetime);
```

```
Query OK, 0 rows affected (0.02 sec)
```

```
set @@time_zone = 'UTC';
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
insert into t values ('2017-09-30 11:11:11', '2017-09-30 11:11:11');
```

```
Query OK, 1 row affected (0.00 sec)
```

```
set @@time_zone = '+8:00';
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
select * from t;
```

```
+---------------------|---------------------+
| ts                  | dt                  |
+---------------------|---------------------+
| 2017-09-30 19:11:11 | 2017-09-30 11:11:11 |
+---------------------|---------------------+
1 row in set (0.00 sec)
```

上面的例子中，无论怎么调整时区的值，Datetime 类型字段的值是不受影响的，而 Timestamp 则随着时区改变，显示的值会发生变化。其实 Timestamp 持久化到存储的值始终没有变化过，只是根据时区的不同显示值不同。

Timestamp 类型和 Datetime 等类型的值，两者相互转换的过程中，会涉及到时区。这种情况一律基于 session 的当前 `time_zone` 时区处理。

另外，用户在导数据的过程中，也要需注意主库和从库之间的时区设定是否一致。

8.7 日常巡检

TiDB 作为分布式数据库，对比单机数据库机制更加复杂，其自带的监控项也很丰富。为了更便捷地运维 TiDB，本文介绍了运维 TiDB 集群需要常关注的关键性能指标。

8.7.1 TiDB Dashboard 关键指标

从 4.0 版本开始，TiDB 提供了一个新的TiDB Dashboard 运维管理工具，集成在 PD 组件上，默认地址为 `http://${pd-ip}:${pd_port}/dashboard`。

使用 TiDB Dashboard，简化了对 TiDB 数据库的运维，可在一个界面查看整个分布式数据库集群的运行状况。下面举例说明。



图 56: 实例面板

以上实例面板的各指标说明如下：

- 状态：用于查看实例状态是否正常，如果在线可忽略此项。
- 启动时间：是关键指标。如果有发现启动时间有变动，那么需要排查组件重启的原因。
- 版本、部署路径、Git 哈希值：通过查看这三个指标可以避免有部署路径和版本不一致或者错误的情况。

8.7.1.2 主机面板



图 57: 主机面板

通过主机面板可以查看 CPU、内存、磁盘使用率。当任何资源的使用率超过 80% 时，推荐扩容对应组件。

8.7.1.3 SQL 分析面板



图 58: SQL 分析面板

通过 SQL 分析面板可以分析对集群影响较大的慢 SQL，然后进行对应的 SQL 优化。

8.7.1.4 Region 信息面板

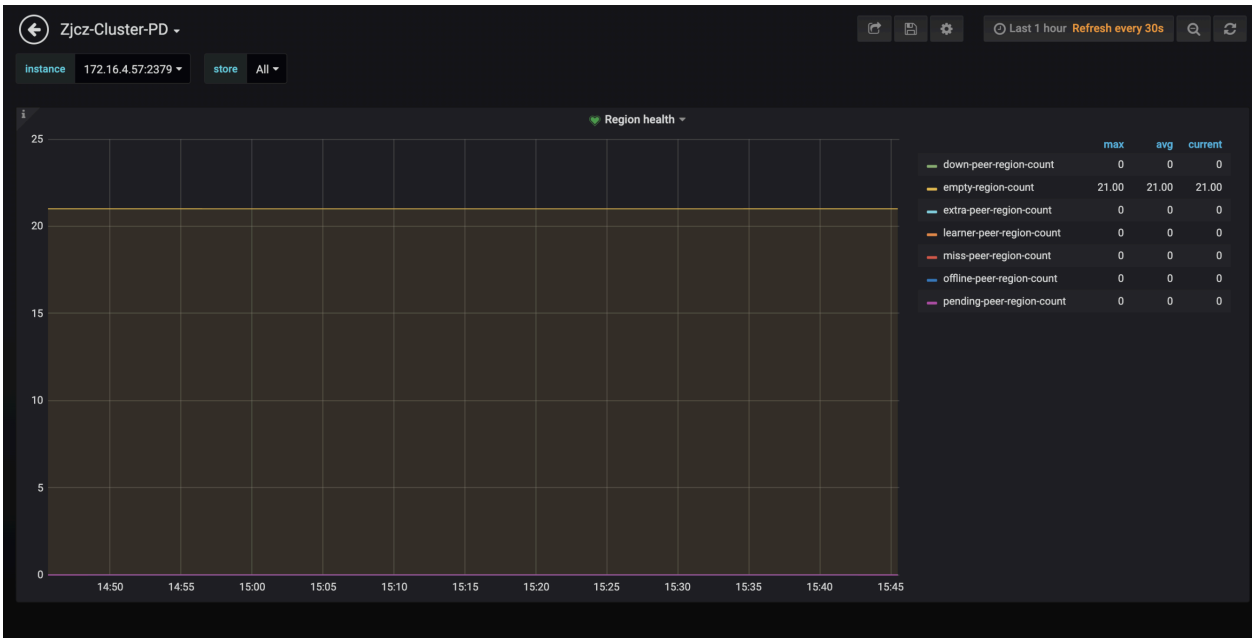


图 59: Region 信息面板

以上 Region 信息面板说明如下：

- `miss-peer-region-count`：缺副本的 Region 数量，不会一直大于 0。
- `extra-peer-region-count`：多副本的 Region 数量，调度过程中会产生。
- `empty-region-count`：空 Region 的数量，一般是 `TRUNCATE TABLE/DROP TABLE` 语句导致。如果数量较多，可以考虑开启跨表 Region merge。
- `pending-peer-region-count`：Raft log 落后的 Region 数量。由于调度产生少量的 pending peer 是正常的，但是如果 pending peer 的数量持续（超过 30 分钟）很高，可能存在问题。

- down-peer-region-count：Raft leader 上报有不响应 peer 的 Region 数量。
- offline-peer-region-count：peer 下线过程中的 Region 数量。

原则上来说，该监控面板偶尔有数据是符合预期的。但长期有数据，需要排查是否存在问题。

8.7.1.5 KV Request Duration

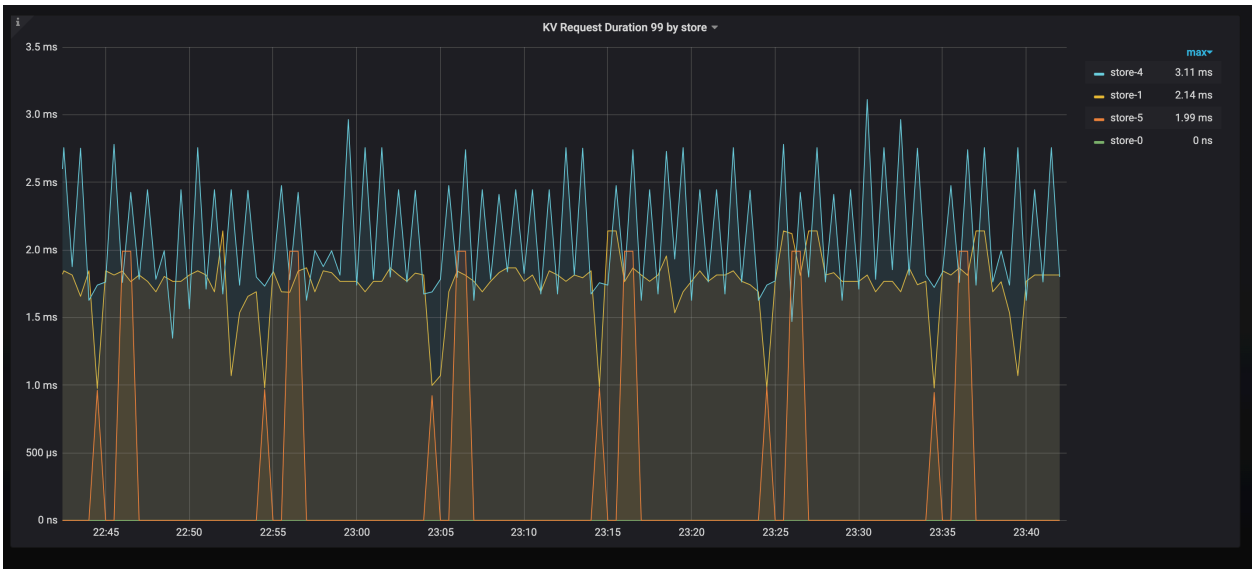


图 60: TiKV 相应时间

TiKV 当前.99（百分位）的响应时间。如果发现有明显高的节点，可以排查是否有热点，或者相关节点性能较差。

8.7.1.6 PD TSO Wait Duration

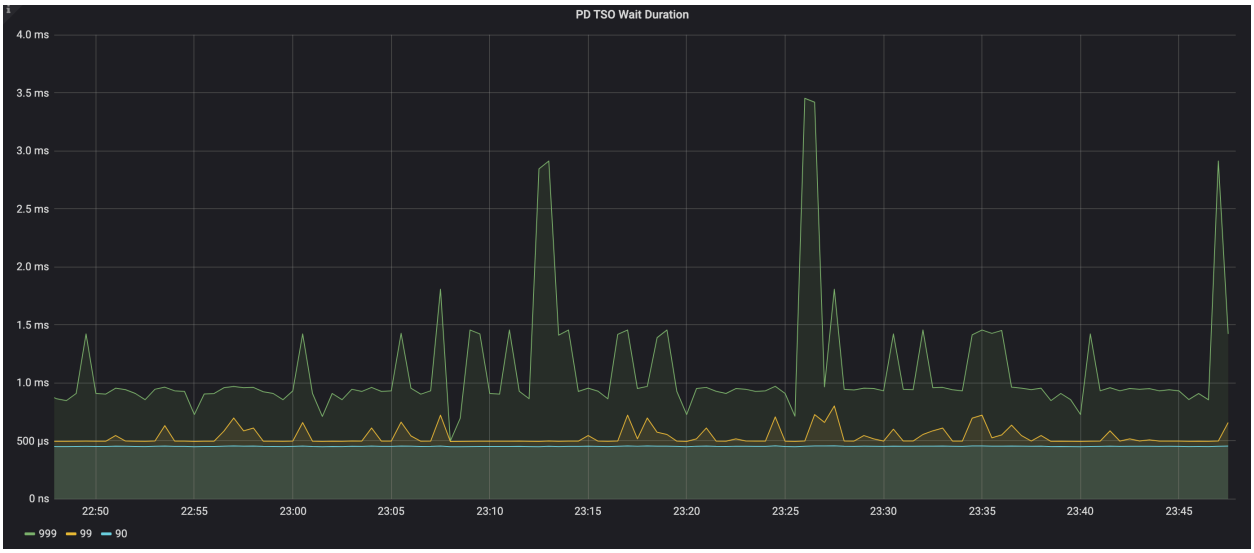


图 61: TiDB 从 PD 获取 TSO 的时间

TiDB 从 PD 获取 TSO 的时间。如果相关响应时间较高，一般常见原因如下：

- TiDB 到 PD 的网络延迟较高，可以手动 Ping 一下网络延迟。
- TiDB 压力较高，导致获取较慢。
- PD 服务器压力较高，导致获取较慢。

8.7.1.7 Overview 面板

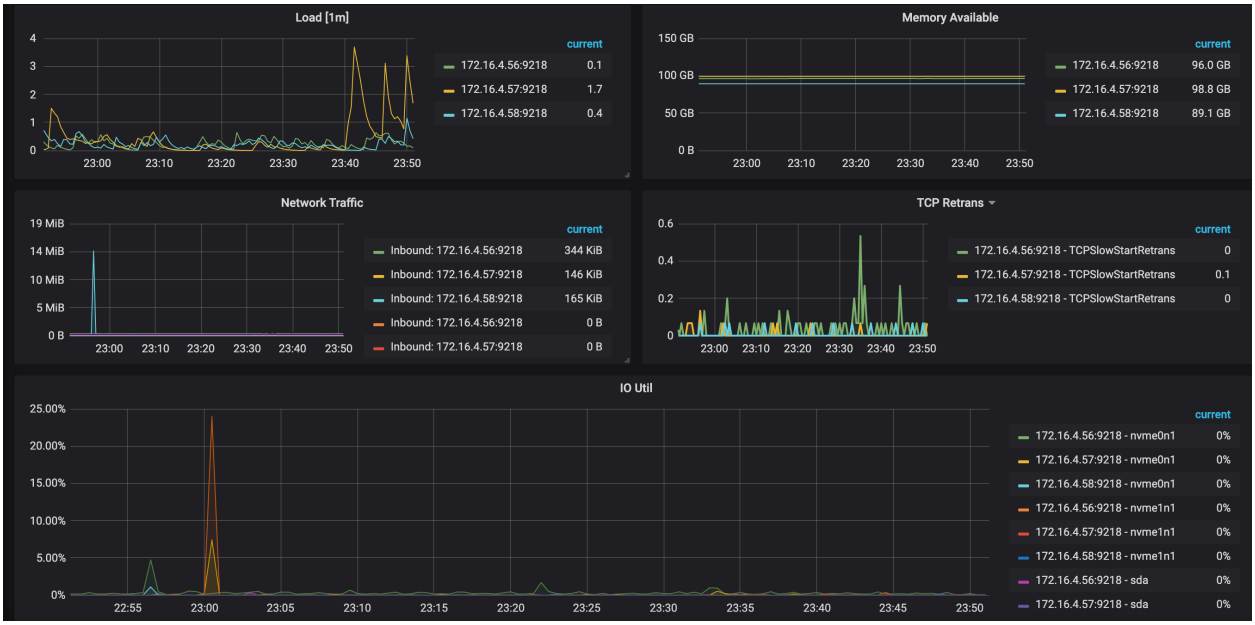


图 62: Overview 面板

以上面板展示常见的负载、内存、网络、IO 监控。发现有瓶颈时，推荐扩容或者优化集群拓扑，优化 SQL、集群参数等。

8.7.1.8 异常监控面板

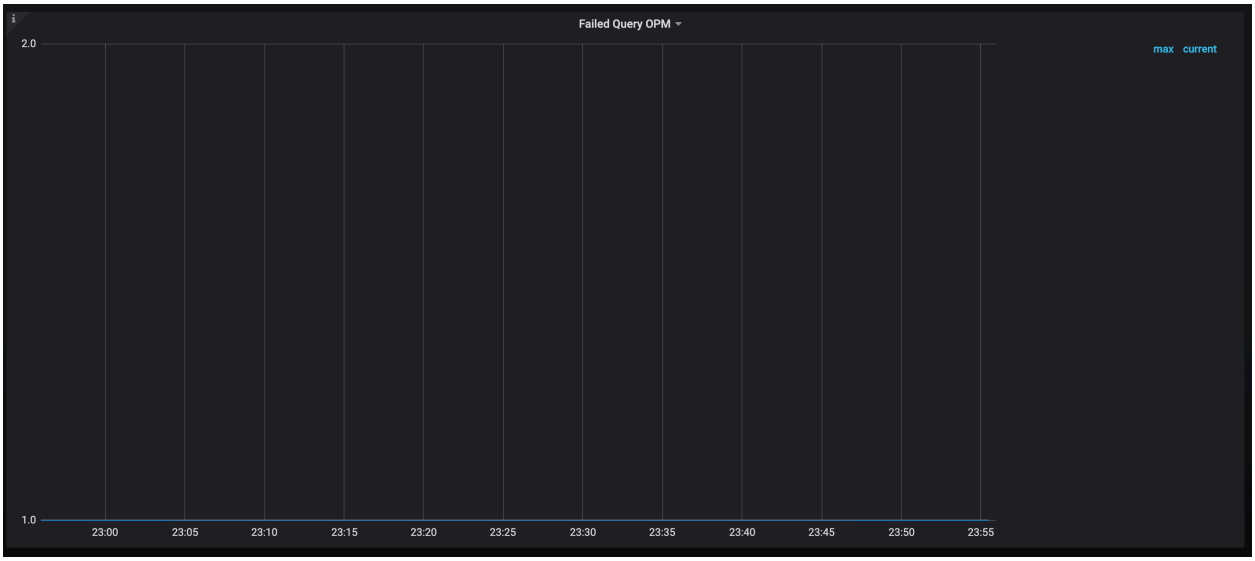


图 63: 异常监控面板

以上面板展示每个 TiDB 实例上，执行 SQL 语句发生的错误，并按照错误类型进行统计，例如语法错误、主键冲突等。

8.7.1.9 GC 状态面板

```
(root@127.0.0.1) [(none)]>select * from mysql.tidb;
+-----------------------+------------------------------------------------------------------------------------------------------+-----------------------------------------------------------------------------------------------------------
```

8.8 TiFlash 集群运维

本文介绍TiFlash 集群运维的一些常见操作，包括查看 TiFlash 版本、TiFlash 重要日志及系统表。

8.8.1 查看 TiFlash 版本

查看 TiFlash 版本有以下两种方法：

- 假设 TiFlash 的二进制文件名为 `tiflash`，则可以通过 `./tiflash version` 方式获取 TiFlash 版本。

  但是由于 TiFlash 的运行依赖于动态库 `libtiflash_proxy.so`，因此需要将包含动态库 `libtiflash_proxy.so` 的目录路径添加到环境变量 `LD_LIBRARY_PATH` 后，上述命令才能正常执行。

  例如，当 `tiflash` 和 `libtiflash_proxy.so` 在同一个目录下时，切换到该目录后，可以通过如下命令查看 TiFlash 版本：

```
LD_LIBRARY_PATH=./ ./tiflash version
```

- 在 TiFlash 日志（日志路径见配置文件 tiflash.toml [logger] 部分）中查看 TiFlash 版本，例如：

```
<information>: TiFlash version: TiFlash 0.2.0 master-375035282451103999f3863c691e2fc2
```

8.8.2 TiFlash 重要日志介绍

日志信息	日志含义
`[INFO] [<unknown>] ["KVStore: Start to persist [region 47, applied: term 6 index 10]"] [thread_id=23]`	在 TiFlash 中看到类似日志代表数据开始同步
`[DEBUG] [<unknown>] ["CoprocessorHandler: grpc::Status DB::CoprocessorHandler::execute(): Handling DAG request"] [thread_id=30]`	该日志代表 TiFlash 开始处理一个 Coprocessor 请求

日志信息	日志含义
[DEBUG] [<unknown >] [" CoprocessorHandler : grpc::Status DB:: CoprocessorHandler ::execute(): Handle DAG request done"] [thread_id=30]	该日志代表 TiFlash 完成 Coprocessor 请求的处理

你可以找到一个 Coprocessor 请求的开始或结束，然后通过日志前面打印的线程号找到该 Coprocessor 请求的其他相关日志。

8.8.3 TiFlash 系统表

`information_schema.tiflash_replica` 系统表的列名及含义如下：

列名	含义
TABLE_SCHEMA	数据库名
TABLE_NAME	表名
TABLE_ID	表 ID
REPLICA_COUNT	TiFlash 副本数
LOCATION_LABELS	给 PD 的 hint，让 Region 的多个副本尽可能按照 LOCATION_LABELS 里的设置分散
AVAILABLE	是否可用（0/1）
PROGRESS	同步进度 [0.0~1.0]

8.9 TiUP 常见运维操作

本文介绍了使用 TiUP 运维 TiDB 集群的常见操作，包括查看集群列表、启动集群、查看集群状态、修改配置参数、关闭集群、销毁集群等。

8.9.1 查看集群列表

TiUP cluster 组件可以用来管理多个 TiDB 集群，在每个 TiDB 集群部署完毕后，该集群会出现在 TiUP 的集群列表里，可以使用 list 命令来查看。

```
tiup cluster list
```

8.9.2 启动集群

启动集群操作会按 PD -> TiKV -> Pump -> TiDB -> TiFlash -> Drainer -> TiCDC -> Prometheus -> Grafana -> Alertmanager 的顺序启动整个 TiDB 集群所有组件：

```
tiup cluster start ${cluster-name}
```

> 注意：
>
> 你需要将 `${cluster-name}` 替换成实际的集群名字，若忘记集群名字，可通过 `tiup cluster list` 查看。

该命令支持通过 `-R` 和 `-N` 参数来只启动部分组件。

例如，下列命令只启动 PD 组件：

```
tiup cluster start ${cluster-name} -R pd
```

下列命令只启动 `1.2.3.4` 和 `1.2.3.5` 这两台机器上的 PD 组件：

```
tiup cluster start ${cluster-name} -N 1.2.3.4:2379,1.2.3.5:2379
```

> 注意：
>
> 若通过 `-R` 和 `-N` 启动指定组件，需要保证启动顺序正确（例如需要先启动 PD 才能启动 TiKV），否则可能导致启动失败。

8.9.3 查看集群状态

集群启动之后需要检查每个组件的运行状态，以确保每个组件工作正常。TiUP 提供了 display 命令，节省了登录到每台机器上去查看进程的时间。

```
tiup cluster display ${cluster-name}
```

8.9.4 修改配置参数

集群运行过程中，如果需要调整某个组件的参数，可以使用 `edit-config` 命令来编辑参数。具体的操作步骤如下：

1. 以编辑模式打开该集群的配置文件：

   ```
   tiup cluster edit-config ${cluster-name}
   ```

2. 设置参数：

   首先确定配置的生效范围，有以下两种生效范围：

   - 如果配置的生效范围为该组件全局，则配置到 server_configs。例如：

   ```
   server_configs:
     tidb:
       log.slow-threshold: 300
   ```

   - 如果配置的生效范围为某个节点，则配置到具体节点的 config 中。例如：

   ```
   tidb_servers:
   - host: 10.0.1.11
     port: 4000
     config:
         log.slow-threshold: 300
   ```

   参数的格式参考 TiUP 配置参数模版。

   配置项层次结构使用 . 表示。

   关于组件的更多配置参数说明，可参考 tidb config.toml.example、tikv config.toml.example 和 pd config.toml.example。

3. 执行 reload 命令滚动分发配置、重启相应组件：

   ```
   tiup cluster reload ${cluster-name} [-N <nodes>] [-R <roles>]
   ```

8.9.4.1 示例

如果要调整 tidb-server 中事务大小限制参数 txn-total-size-limit 为 1G，该参数位于 performance 模块下，调整后的配置如下：

```
server_configs:
  tidb:
    performance.txn-total-size-limit: 1073741824
```

然后执行 tiup cluster reload ${cluster-name} -R tidb 命令滚动重启。

8.9.5 Hotfix 版本替换

常规的升级集群请参考升级文档，但是在某些场景下（例如 Debug），可能需要用一个临时的包替换正在运行的组件，此时可以用 patch 命令：

```
tiup cluster patch --help
```

```
Replace the remote package with a specified package and restart the service

Usage:
  tiup cluster patch <cluster-name> <package-path> [flags]

Flags:
  -h, --help                    帮助信息
  -N, --node strings            指定被替换的节点
      --overwrite               在未来的 scale-out 操作中使用当前指定的临时包
  -R, --role strings            指定被替换的服务类型
      --transfer-timeout int    transfer leader 的超时时间

Global Flags:
      --native-ssh        使用系统默认的 SSH 客户端
      --wait-timeout int  等待操作超时的时间
      --ssh-timeout int   SSH 连接的超时时间
  -y, --yes               跳过所有的确认步骤
```

例如，有一个 TiDB 实例的 hotfix 包放在 `/tmp/tidb-hotfix.tar.gz` 目录下。如果此时想要替换集群上的所有 TiDB 实例，则可以执行以下命令：

```
tiup cluster patch test-cluster /tmp/tidb-hotfix.tar.gz -R tidb
```

或者只替换其中一个 TiDB 实例：

```
tiup cluster patch test-cluster /tmp/tidb-hotfix.tar.gz -N 172.16.4.5:4000
```

8.9.6 重命名集群

部署并启动集群后，可以通过 `tiup cluster rename` 命令来对集群重命名：

```
tiup cluster rename ${cluster-name} ${new-name}
```

> 注意：
>
> - 重命名集群会重启监控（Prometheus 和 Grafana）。
> - 重命名集群之后 Grafana 可能会残留一些旧集群名的面板，需要手动删除这些面板。

8.9.7 关闭集群

关闭集群操作会按 Alertmanager -> Grafana -> Prometheus -> TiCDC -> Drainer -> TiFlash -> TiDB -> Pump -> TiKV -> PD 的顺序关闭整个 TiDB 集群所有组件（同时也会关闭监控组件）：

```
tiup cluster stop ${cluster-name}
```

和 start 命令类似，stop 命令也支持通过 -R 和 -N 参数来只停止部分组件。

例如，下列命令只停止 TiDB 组件：

```
tiup cluster stop ${cluster-name} -R tidb
```

下列命令只停止 1.2.3.4 和 1.2.3.5 这两台机器上的 TiDB 组件：

```
tiup cluster stop ${cluster-name} -N 1.2.3.4:4000,1.2.3.5:4000
```

8.9.8 清除集群数据

此操作会关闭所有服务，并清空其数据目录或/和日志目录，并且无法恢复，需要谨慎操作。

清空集群所有服务的数据，但保留日志：

```
tiup cluster clean ${cluster-name} --data
```

清空集群所有服务的日志，但保留数据：

```
tiup cluster clean ${cluster-name} --log
```

清空集群所有服务的数据和日志：

```
tiup cluster clean ${cluster-name} --all
```

清空 Prometheus 以外的所有服务的日志和数据：

```
tiup cluster clean ${cluster-name} --all --ignore-role prometheus
```

清空节点 172.16.13.11:9000 以外的所有服务的日志和数据：

```
tiup cluster clean ${cluster-name} --all --ignore-node 172.16.13.11:9000
```

清空部署在 172.16.13.12 以外的所有服务的日志和数据：

```
tiup cluster clean ${cluster-name} --all --ignore-node 172.16.13.12
```

8.9.9 销毁集群

销毁集群操作会关闭服务，清空数据目录和部署目录，并且无法恢复，需要谨慎操作。

```
tiup cluster destroy ${cluster-name}
```

8.10 在线修改集群配置

在线配置变更主要是通过利用 SQL 对包括 TiDB、TiKV 以及 PD 在内的各组件的配置进行在线更新。用户可以通过在线配置变更对各组件进行性能调优而无需重启集群组件。但目前在线修改 TiDB 实例配置的方式和修改其他组件 (TiKV, PD) 的有所不同。

8.10.1 常用操作

8.10.1.1 查看实例配置

可以通过 SQL 语句 `show config` 来直接查看集群所有实例的配置信息，结果如下：

```
show config;
```

```
+--
    ↪ ----+-----------------+-------------------------------------------------------------+-------------------
    ↪
| Type | Instance        | Name                                                        | Value
    ↪
    ↪ |
+--
    ↪ ----+-----------------+-------------------------------------------------------------+-------------------
    ↪
| tidb | 127.0.0.1:4001  | advertise-address                                           | 127.0.0.1
    ↪
    ↪ |
| tidb | 127.0.0.1:4001  | binlog.binlog-socket                                        |
    ↪
    ↪ |
| tidb | 127.0.0.1:4001  | binlog.enable                                               | false
    ↪
    ↪ |
| tidb | 127.0.0.1:4001  | binlog.ignore-error                                         | false
    ↪
    ↪ |
| tidb | 127.0.0.1:4001  | binlog.strategy                                             | range
    ↪
    ↪ |
| tidb | 127.0.0.1:4001  | binlog.write-timeout                                        | 15s
    ↪
    ↪ |
| tidb | 127.0.0.1:4001  | check-mb4-value-in-utf8                                     | true
    ↪
    ↪ |

...
```

还可以根据对应的字段进行过滤，如：

```
show config where type='tidb'
show config where instance in (...)
show config where name like '%log%'
show config where type='tikv' and name='log.level'
```

8.10.1.2 在线修改 TiKV 配置

> 注意：
>
> 在线修改 TiKV 配置项后，同时会自动修改 TiKV 的配置文件。但还需要使用 `tiup edit-config` 命令来修改对应的配置项，否则 `upgrade` 和 `reload` 等运维操作会将在线修改配置后的结果覆盖。修改配置的操作请参考：使用 TiUP 修改配置。执行 `tiup edit-config` 后不需要执行 `tiup reload` 操作。

执行 SQL 语句 `set config`，可以结合实例地址或组件类型来修改单个实例配置或全部实例配置，如：

修改全部 TiKV 实例配置：

> 注意：
>
> 建议使用反引号包裹变量名称。

```
set config tikv `split.qps-threshold`=1000
```

修改单个 TiKV 实例配置：

```
set config "127.0.0.1:20180" `split.qps-threshold`=1000
```

设置成功会返回 `Query OK`：

```
Query OK, 0 rows affected (0.01 sec)
```

在批量修改时如果有错误发生，会以 warning 的形式返回：

```
set config tikv `log-level`='warn';
```

```
Query OK, 0 rows affected, 1 warning (0.04 sec)
```

```
show warnings;
```

```
+--
    ↪ -------+------+-----------------------------------------------------------------------------------------
    ↪
| Level   | Code | Message
    ↪
    ↪ |
+--
    ↪ -------+------+-----------------------------------------------------------------------------------------
    ↪
| Warning | 1105 | bad request to http://127.0.0.1:20180/config: fail to update, error: "config
    ↪ log-level can not be changed" |
+--
    ↪ -------+------+-----------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

批量修改配置不保证原子性，可能出现某些实例成功，而某些失败的情况。如使用 `set tikv key=val` 命令修改整个 TiKV 集群配置时，可能有部分实例失败，请执行 `show warnings` 进行查看。

如遇到部分修改失败的情况，需要重新执行对应的修改语句，或通过修改单个实例的方式完成修改。如果因网络或者机器故障等原因无法访问到的 TiKV，需要等到恢复后再次进行修改。

针对 TiKV 可在线修改的配置项，如果成功修改后，修改的结果会被持久化到配置文件中，后续以配置文件中的配置为准。某些配置项名称可能和 TiDB 预留关键字冲突，如 `limit`、`key` 等，对于此类配置项，需要用反引号 ` 包裹起来，如 `` `raftstore.raft-log-gc-size-limit` ``。

支持的配置项列表如下：

配置项	简介
log.level	日志等级

配置项	简介
raftstore.raft-max-inflight-msgs	待确认的日志个数，如果超过这个数量，Raft 状态机会减缓发送日志的速度
raftstore.raft-log-gc-tick-interval	删除 Raft 日志的轮询任务调度间隔时间
raftstore.raft-log-gc-threshold	允许残余的 Raft 日志个数，软限制
raftstore.raft-log-gc-count-limit	允许残余的 Raft 日志个数，硬限制

配置项	简介
raftstore.raft-log-gc-size-limit	允许残余的 Raft 日志大小，硬限制
raftstore.raft-max-size-per-msg	允许生成的单个消息包的大小，软限制
raftstore.raft-entry-max-size	单个 Raft 日志最大大小，硬限制
raftstore.raft-entry-cache-life-time	内存中日志 cache 允许的最长残留时间
raftstore.split-region-check-tick-interval	检查 Region 是否需要分裂的时间间隔

配置项	简介
raftstore.region-split-check-diff	允许 Region 数据超过指定大小的最大值检查
raftstore.region-compact-check-interval	检查是否需要人工触发 RocksDB compaction 的时间间隔
raftstore.region-compact-check-step	每轮校验人工 compaction 时，一次性检查的 Region 个数
raftstore.region-compact-min-tombstones	触发 RocksDB compaction 需要的 tombstone 个数

配置项	简介
raftstore.region-compact-tombstones-percent	触发 RocksDB compaction 需要的 tombstone 所占比例
raftstore.pd-heartbeat-tick-interval	触发 Region 对 PD 心跳的时间间隔
raftstore.pd-store-heartbeat-tick-interval	触发 store 对 PD 心跳的时间间隔
raftstore.snap-mgr-gc-tick-interval	触发回收过期 snapshot 文件的时间间隔
raftstore.snap-gc-timeout	snapshot 文件的最长保存时间

配置项	简介
raftstore.lock-cf-compact-interval	触发对 lock CF 进行 compact 检查的时间间隔
raftstore.lock-cf-compact-bytes-threshold	触发对 lock CF 进行 compact 的大小
raftstore.messages-per-tick	每轮处理的消息最大个数
raftstore.max-peer-down-duration	副本允许的最长未响应时间

配置项	简介
raftstore.max-leader-missing-duration	允许副本处于无主状态的最长时间，超过将会向 PD 校验自己是否已经被删除
raftstore.abnormal-leader-missing-duration	允许副本处于无主状态的时间，超过将视为异常，标记在 metrics 和日志中

配置项	简介
raftstore.peer-stale-state-check-interval	触发检验副本是否处于无主状态的时间间隔
raftstore.consistency-check-interval	触发一致性检查的时间间隔（不建议使用该配置项，因为与 TiDB GC 操作不兼容）
raftstore.raft-store-max-leader-lease	主可信任期的最长时间
raftstore.merge-check-tick-interval	触发 Merge 完成检查的时间间隔

配置项	简介
raftstore.backup-import-sst-interval	触发检查过期 SST 文件的时间间隔
raftstore.local-read-batch-size	一轮处理读请求的最大个数
raftstore.apply-yield-write-size	apply 线程每一轮处理单个状态机写入的最大数据量

配置项	简介
raftstore.hibernate-timeout	启动后进入静默状态前需要等待的最短时间，在该时间段内不会进入静默状态（未 re-lease）
raftstore.apply-pool-size	处理把数据落盘至磁盘的线程池中线程的数量，即 Apply 线程池大小

配置项	简介
raftstore.store-pool-size	处理 Raft 的线程池中线程的数量，即 Raft-store 线程池的大小
raftstore.apply-max-batch-size	Apply 状态机由 Batch-System 批量执行数据写入请求，该配置项指定每批可执行请求的最多 Raft 状态机个数。

配置项	简介
raftstore.store-max-batch-size	Raft 状态机由 Batch-System 批量执行把日志落盘至磁盘的请求，该配置项指定每批可执行请求的最多 Raft 状态机个数。

配置项	简介
raftstore.store-io-pool-size	处理 Raft I/O 任务的线程池中线程的数量，即 StoreWriter 线程池的大小（不支持将该配置项由非零值调整为 0，或者从 0 调整为非零值）

配置项	简介
readpool.unified.max-thread-count	统一处理读请求的线程池最多的线程数量，即 UnifyReadPool 线程池大小
readpool.unified.auto-adjust-pool-size	是否开启自适应调整 UnifyReadPool 的大小
coprocessor.split-region-on-table	开启按 table 分裂 Region 的开关
coprocessor.batch-split-limit	批量分裂 Region 的阈值

配置项	简介
coprocessor.region-max-size	Region 容量空间的最大值
coprocessor.region-split-size	分裂后新 Region 的大小
coprocessor.region-max-keys	最多允许的 key 的个数
coprocessor.region-split-keys	分裂后新 Region 的 key 的个数
pessimistic-txn.wait-for-lock-timeout	悲观事务遇到锁后的最长等待时间
pessimistic-txn.wake-up-delay-duration	悲观事务被重新唤醒的时间

配置项	简介
pessimistic-txn.pipelined	是否开启流水线式加悲观锁流程
pessimistic-txn.in-memory	是否开启内存悲观锁功能
quota.foreground-cpu-time	限制处理 TiKV 前台读写请求所使用的 CPU 资源使用量，软限制
quota.foreground-write-bandwidth	限制前台事务写入的带宽，软限制

配置项	简介
quota.foreground-read-bandwidth	限制前台事务读取数据和 Coprocessor 读取数据的带宽，软限制
quota.background-cpu-time	限制后台处理 TiKV 后台读写请求所使用的 CPU 资源使用量，软限制
quota.background-write-bandwidth	限制后台事务写入的带宽，软限制，暂未生效

配置项	简介
quota.background-read-bandwidth	限制后台事务读取数据和 Coprocessor 读取数据的带宽，软限制，暂未生效
quota.enable-auto-tune	是否支持 quota 动态调整。如果打开该配置项，TiKV 会根据 TiKV 实例的负载情况动态调整对后台请求的限制 quota

配置项	简介
quota.max-delay-duration	单次读写请求被强制等待的最大时间
gc.ratio-threshold	跳过 Region GC 的阈值（GC 版本个数/key 个数）
gc.batch-keys	一轮处理 key 的个数
gc.max-write-bytes-per-sec	一秒可写入 RocksDB 的最大字节数
gc.enable-compaction-filter	是否使用 compaction filter

配置项	简介
gc.compaction.filter-skip-version-check	是否跳过 compaction filter 的集群版本检查（未 release）
{db-name}.max-total-wal-size	WAL 总大小限制
{db-name}.max-background-jobs	RocksDB 后台线程个数
{db-name}.max-background-flushes	RocksDB flush 线程个数
{db-name}.max-open-files	RocksDB 可以打开的文件总数
{db-name}.compaction-readahead-size	Compaction 时候 readahead 的大小

配置项	简介
{db-name}.bytes-per-sync	异步同步的限速速率
{db-name}.wal-bytes-per-sync	WAL 同步的限速速率
{db-name}.writable-file-max-buffer-size	WritableFileWrite 所使用的最大的 buffer 大小
{db-name}.{cf-name}.block-cache-size	block cache 大小
{db-name}.{cf-name}.write-buffer-size	memtable 大小
{db-name}.{cf-name}.max-write-buffer-number	最大 memtable 个数
{db-name}.{cf-name}.max-bytes-for-level-base	base level (L-1) 最大字节数

配置项	简介
{db-name}.{cf-name}.target-file-size-base	base level 的目标文件大小
{db-name}.{cf-name}.level0-file-num-compaction-trigger	触发 compaction 的 L0 文件最大个数
{db-name}.{cf-name}.level0-slowdown-writes-trigger	触发写入减缓的 L0 文件最大个数
{db-name}.{cf-name}.level0-stop-writes-trigger	完全阻停写入的 L0 文件最大个数
{db-name}.{cf-name}.max-compaction-bytes	一次 compaction 最大写入字节数
{db-name}.{cf-name}.max-bytes-for-level-multiplier	每一层的默认放大倍数

配置项	简介
{db-name}.{cf-name}.disable-auto-compactions	自动 compaction 的开关
{db-name}.{cf-name}.soft-pending-compaction-bytes-limit	pending compaction bytes 的软限制
{db-name}.{cf-name}.hard-pending-compaction-bytes-limit	pending compaction bytes 的硬限制
{db-name}.{cf-name}.titan.blob-run-mode	处理 blob 文件的模式
server.grpc-memory-pool-quota	gRPC 可使用的内存大小限制
server.max-grpc-send-msg-len	gRPC 可发送的最大消息长度

配置项	简介
server.raft-msg-max-batch-size	单个 gRPC 消息可包含的最大 Raft 消息个数
server.simplify-metrics	精简监控采样数据的开关
server.snap-io-max-bytes-per-sec	处理 snapshot 时最大允许使用的磁盘带宽
server.concurrent-send-snap-limit	同时发送 snapshot 的最大个数
server.concurrent-recv-snap-limit	同时接受 snapshot 的最大个数

配置项	简介
storage.block-cache.capacity	共享 block cache 的大小（自 v4.0.3 起支持）
storage.scheduler-worker-pool-size	scheduler 线程池中线程的数量
backup.num-threads	backup 线程的数量（自 v4.0.3 起支持）

配置项	简介
split.qps-threshold	对 Region 执行 load-base-split 的阈值。如果连续 10s 内，某个 Region 的读请求的 QPS 超过 qps-threshold，则尝试切分该 Region

配置项	简介
split.byte-threshold	对 Region 执行 load-base-split 的阈值。如果连续 10s 内，某个 Region 的读请求的流量超过 byte-threshold，则尝试切分该 Region

配置项	简介
split.region-cpu-overload-threshold-ratio	对 Region 执行 load-base-split 的阈值。如果连续 10s 内，某个 Region 的 Unified Read Pool CPU 使用时间占比超过了 region-cpu-overload-threshold-ratio，则尝试切分该 Region（自 v6.2.0 起支持）

配置项	简介
split.split-balance-score	load-base-split 的控制参数，确保 Region 切分后左右访问尽量均匀，数值越小越均匀，但也可能导致无法切分
split.split-contained-score	load-base-split 的控制参数，数值越小，Region 切分后跨 Region 的访问越少

配置项	简介
cdc.min-ts-interval	定期推进 Resolved TS 的时间间隔
cdc.old-value-cache-memory-quota	缓存在内存中的 TiCDC Old Value 的条目占用内存的上限
cdc.sink-memory-quota	缓存在内存中的 TiCDC 数据变更事件占用内存的上限
cdc.incremental-scan-speed-limit	增量扫描历史数据的速度上限

配置项	简介
cdc.increment-scan-concurrency	增量扫描历史数据任务的最大并发执行个数

上述前缀为 {db-name} 或 {db-name}.{cf-name} 的是 RocksDB 相关的配置项。db-name 的取值可为 rocksdb 或 raftdb。

- 当 db-name 为 rocksdb 时，cf-name 的可取值有：defaultcf、writecf、lockcf、raftcf；
- 当 db-name 为 raftdb 时，cf-name 的可取值有：defaultcf。

具体配置项的意义可参考TiKV 配置文件描述

8.10.1.3 在线修改 PD 配置

PD 暂不支持单个实例拥有独立配置。所有实例共享一份配置，可以通过下列方式修改 PD 的配置项：

```
set config pd `log.level`='info'
```

设置成功会返回 Query OK：

```
Query OK, 0 rows affected (0.01 sec)
```

针对 PD 可在线修改的配置项，成功修改后则会持久化到 etcd 中，不会对配置文件进行持久化，后续以 etcd 中的配置为准。同上，若和 TiDB 预留关键字冲突，需要用反引号 ` 包裹此类配置项，例如 `schedule.leader-schedule-limit`。

支持配置项列表如下：

配置项	简介
log.level	日志级别
cluster-version	集群的版本
schedule.max-merge-region-size	控制 Region Merge 的 size 上限（单位是 MiB）
schedule.max-merge-region-keys	控制 Region Merge 的 key 数量上限
schedule.patrol-region-interval	控制 replicaChecker 检查 Region 健康状态的运行频率
schedule.split-merge-interval	控制对同一个 Region 做 split 和 merge 操作的间隔
schedule.max-snapshot-count	控制单个 store 最多同时接收或发送的 snapshot 数量

配置项	简介
schedule.max-pending-peer-count	控制单个 store 的 pending peer 上限
schedule.max-store-down-time	PD 认为失联 store 无法恢复的时间
schedule.leader-schedule-policy	用于控制 leader 调度的策略
schedule.leader-schedule-limit	可以控制同时进行 leader 调度的任务个数
schedule.region-schedule-limit	可以控制同时进行 Region 调度的任务个数
schedule.replica-schedule-limit	可以控制同时进行 replica 调度的任务个数
schedule.merge-schedule-limit	控制同时进行的 Region Merge 调度的任务
schedule.hot-region-schedule-limit	可以控制同时进行的热点调度的任务个数
schedule.hot-region-cache-hits-threshold	用于设置 Region 被视为热点的阈值
schedule.high-space-ratio	用于设置 store 空间充裕的阈值
schedule.low-space-ratio	用于设置 store 空间不足的阈值
schedule.tolerant-size-ratio	控制 balance 缓冲区大小
schedule.enable-remove-down-replica	用于开启自动删除 DownReplica 的特性
schedule.enable-replace-offline-replica	用于开启迁移 OfflineReplica 的特性
schedule.enable-make-up-replica	用于开启补充副本的特性
schedule.enable-remove-extra-replica	用于开启删除多余副本的特性
schedule.enable-location-replacement	用于开启隔离级别检查
schedule.enable-cross-table-merge	用于开启跨表 Merge
schedule.enable-one-way-merge	用于开启单向 Merge（只允许和下一个相邻的 Region Merge）
replication.max-replicas	用于设置副本的数量
replication.location-labels	用于设置 TiKV 集群的拓扑信息
replication.enable-placement-rules	开启 Placement Rules
replication.strictly-match-label	开启 label 检查
pd-server.use-region-storage	开启独立的 Region 存储
pd-server.max-gap-reset-ts	用于设置最大的重置 timestamp 的间隔（BR）
pd-server.key-type	用于设置集群 key 的类型
pd-server.metric-storage	用于设置集群 metrics 的存储地址
pd-server.dashboard-address	用于设置 dashboard 的地址
replication-mode.replication-mode	备份的模式

具体配置项意义可参考PD 配置文件描述。

8.10.1.4 在线修改 TiDB 配置

在线修改 TiDB 配置的方式和 TiKV/PD 有所不同，你可以通过修改系统变量来实现。

下面例子展示了如何通过变量 `tidb_slow_log_threshold` 在线修改配置项 `slow-threshold`。

`slow-threshold` 默认值是 300 毫秒，可以通过设置系统变量 `tidb_slow_log_threshold` 将其修改为 200 毫秒：

```
set tidb_slow_log_threshold = 200;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
select @@tidb_slow_log_threshold;
```

```
+---------------------------+
| @@tidb_slow_log_threshold |
+---------------------------+
| 200                       |
+---------------------------+
1 row in set (0.00 sec)
```

支持在线修改的配置项和相应的 TiDB 系统变量如下：

配置项	对应变量	简介
log.enable-slow-log	tidb_enable_slow_log	慢日志的开关
log.slow-threshold	tidb_slow_log_threshold	慢日志阈值
log.expensive-threshold	tidb_expensive_query_time_threshold	expensive 查询阈值

8.10.1.5 在线修改 TiFlash 配置

目前，你可以通过修改系统变量`tidb_max_tiflash_threads` 来在线修改 TiFlash 配置项 max_threads。`tidb_max_tiflash_threads` 表示 TiFlash 中 request 执行的最大并发度。

`tidb_max_tiflash_threads` 默认值是 `-1`，表示此系统变量无效，由 TiFlash 的配置文件决定 max_threads。你可以通过设置系统变量 `tidb_max_tiflash_threads` 将其修改为 10：

```
set tidb_max_tiflash_threads = 10;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
select @@tidb_max_tiflash_threads;
```

```
+----------------------------+
| @@tidb_max_tiflash_threads |
+----------------------------+
| 10                         |
+----------------------------+
1 row in set (0.00 sec)
```

8.11 Online Unsafe Recovery 使用文档

> 警告：
>
> - 此功能为有损恢复，无法保证数据索引一致性和事务完整性，若有问题需要额外的工具或者步骤进行相应修复。
> - 该功能自 v6.1.0 版本开始引入。在 TiDB v6.1 以下版本为实验特性，行为与本文描述有区别，不推荐使用。在其他版本使用该功能时，请参考相应版本文档。

当多数副本的永久性损坏造成部分数据不可读写时，可以使用 Online Unsafe Recovery 功能进行数据有损恢复，使 TiKV 正常提供服务。

8.11.1 功能说明

在 TiDB 中，根据用户定义的多种副本规则，一份数据可能会同时存储在多个节点中，从而保证在单个或少数节点暂时离线或损坏时，读写数据不受任何影响。但是，当一个 Region 的多数或全部副本在短时间内全部下线时，该 Region 会处于暂不可用的状态，无法进行读写操作。

如果一段数据的多数副本发生了永久性损坏（如磁盘损坏）等问题，从而导致节点无法上线时，此段数据会一直保持暂不可用的状态。这时，如果用户希望集群恢复正常使用，在用户能够容忍数据回退或数据丢失的前提下，用户理论上可以通过手动移除不可用副本的方式，使 Region 重新形成多数派，进而让上层业务可以写入和读取（可能是 stale 的，或者为空）这一段数据分片。

在这个情况下，当存有可容忍丢失的数据的部分节点受到永久性损坏时，用户可以通过使用 Online Unsafe Recovery，快速简单地进行有损恢复。使用 Online Unsafe Recovery 时，PD 会自动暂停调度（包括 split 和 merge），然后收集全部节点内的数据分片元信息，用 PD 的全局视角生成一份更实时、更完整的恢复计划后，将其计划下发给各个存活的节点，使各节点执行数据恢复任务。另外，下发恢复计划后，PD 还会定期查看恢复进度，并在必要时重新向各节点分发恢复计划。

8.11.2 适用场景

Online Unsafe Recovery 功能适用于以下场景：

- 部分节点受到永久性损坏，导致节点无法重启，造成业务端的部分数据不可读、不可写。
- 可以容忍数据丢失，希望受影响的数据恢复读写。

8.11.3 使用步骤

8.11.3.1 前提条件

在使用 Online Unsafe Recovery 功能进行数据有损恢复前，请确认以下事项：

- 离线节点导致部分数据确实不可用。
- 离线节点确实无法自动恢复或重启。

8.11.3.2 第 1 步：指定无法恢复的节点

使用 PD Control 执行`unsafe remove-failed-stores <store_id>[,<store_id>,...]`命令，指定已确定无法恢复的所有 TiKV 节点，并用逗号隔开，以触发自动恢复。

```
pd-ctl -u <pd_addr> unsafe remove-failed-stores <store_id1,store_id2,...>
```

命令输出 `Success` 表示向 PD 注册任务成功。但仅表示请求已被接受，并不代表恢复成功。恢复任务在后台进行，具体进度使用`show`查看。命令输出 `Failed` 表示注册任务失败，可能的错误有：

- `unsafe recovery is running`：已经有正在进行的恢复任务

- `invalid input store x doesn't exist`：指定的 store ID 不存在
- `invalid input store x is up and connected`：指定的 store ID 仍然是健康的状态，不应该进行恢复

可通过 `--timeout <seconds>` 指定可允许执行恢复的最长时间。若未指定，默认为 5 分钟。当超时后，恢复中断报错。

若 PD 进行过灾难性恢复`pd-recover` 操作，丢失了无法恢复的 TiKV 节点的 store 信息，因此无法确定要传的 store ID 时，可指定 `--auto-detect` 参数允许传入一个空的 store ID 列表。在该模式下，所有未在 PD store 列表中的 store ID 均被认为无法恢复，进行移除。

> 注意：
>
> - 由于此命令需要收集来自所有 Peer 的信息，可能会造成 PD 短时间内有明显的内存使用量上涨（10 万个 Peer 预计使用约 500 MiB 内存）。
> - 若执行过程中 PD 发生重启，则恢复中断，需重新触发命令。
> - 一旦执行，所指定的节点将被设为 Tombstone 状态，不再允许启动。
> - 执行过程中，所有调度以及 split/merge 都会被暂停，待恢复成功或失败后自动恢复。

8.11.3.3 第 2 步：查看进度等待结束

节点移除命令运行成功后，使用 PD Control 执行`unsafe remove-failed-stores show` 命令，查看移除进度。

```
pd-ctl -u <pd_addr> unsafe remove-failed-stores show
```

恢复过程有多个可能的阶段：

- `collect report`：初始阶段，第一次接收 TiKV 的报告获得的全局信息。
- `tombstone tiflash learner`：在不健康的 Region 中，删除比其他健康 Peer 要新的 TiFlash learner，防止极端情况造成 panic。
- `force leader for commit merge`：特殊阶段。在有未完成的 commit merge 时出现，优先对有 commit merge 的 Region 进行 `force leader`，防止极端情况。
- `force leader`：强制不健康的 Region 在剩余的健康 Peer 中指定一个成为 Raft leader。
- `demote failed voter`：将 Region 不健康的 Voter 降级为 Learner，之后 Region 就可以正常地选出 Raft leader。
- `create empty region`：创建一个空 Region 来补足 key range 的空洞，主要针对的是某些 Region 的所有副本所在的 Store 都损坏了。

每一阶段按照 JSON 格式输出，包括信息，时间，以及具体的恢复计划。例如：

```
[
    {
        "info": "Unsafe recovery enters collect report stage",
        "time": "......",
        "details" : [
            "failed stores 4, 5, 6",
```

```
        ]
    },
    {
        "info": "Unsafe recovery enters force leader stage",
        "time": "......",
        "actions": {
            "store 1": [
                "force leader on regions: 1001, 1002"
            ],
            "store 2": [
                "force leader on regions: 1003"
            ]
        }
    },
    {
        "info": "Unsafe recovery enters demote failed voter stage",
        "time": "......",
        "actions": {
            "store 1": [
                "region 1001 demotes peers { id:101 store_id:4 }, { id:102 store_id:5 }",
                "region 1002 demotes peers { id:103 store_id:5 }, { id:104 store_id:6 }",
            ],
            "store 2": [
                "region 1003 demotes peers { id:105 store_id:4 }, { id:106 store_id:6 }",
            ]
        }
    },
    {
        "info": "Collecting reports from alive stores(1/3)",
        "time": "......",
        "details": [
            "Stores that have not dispatched plan: ",
            "Stores that have reported to PD: 4",
            "Stores that have not reported to PD: 5, 6",
        ]
    }
]
```

PD 下发恢复计划后，会等待 TiKV 上报执行的结果。如上述输出中最后一阶段的 `Collecting reports from alive stores` 显示 PD 下发恢复计划和接受 TiKV 报告的具体状态。

整个恢复过程包括多个阶段，可能存在某一阶段的多次重试。一般情况下，预计时间为 3~10 个 store heartbeat 周期（一个 store heartbeat 默认为 10s）。当恢复完成后，命令执行结果最后一阶段显示 `"Unsafe recovery finished"`，以及受影响的 Region 所属的 table id（若无或使用 RawKV 则不显示）和受影响的 SQL 元数据 Region。如：

```
{
    "info": "Unsafe recovery finished",
    "time": "......",
    "details": [
        "Affected table ids: 64, 27",
        "Affected meta regions: 1001",
    ]
}
```

得到受影响的 table id 后，可以使用 `INFORMATION_SCHEMA.TABLES` 来查看受影响的表名。

```
SELECT TABLE_SCHEMA, TABLE_NAME, TIDB_TABLE_ID FROM INFORMATION_SCHEMA.TABLES WHERE TIDB_TABLE_ID
    ↪  IN (64, 27);
```

> 注意：
>
> - 恢复操作把一些 failed Voter 变成了 failed Learner，之后还需要 PD 调度经过一些时间将这些 failed Learner 移除。
> - 建议及时添加新的节点。

若执行过程中发生错误，最后一阶段会显示 `"Unsafe recovery failed"` 以及具体错误。如：

```
{
    "info": "Unsafe recovery failed: <error>",
    "time": "......"
}
```

8.11.3.4 第 3 步：检查数据索引一致性（RawKV 不需要）

> 注意：
>
> 数据可以读写并不代表没有数据丢失。

执行完成后，数据和索引可能会不一致。请使用`ADMIN CHECK` 对受影响的表进行数据索引的一致性检查。

```
ADMIN CHECK TABLE table_name;
```

若结果有不一致的索引，可以通过重命名旧索引、创建新索引，然后再删除旧索引的步骤来修复数据索引不一致的问题。

1. 重命名旧索引：

```
ALTER TABLE table_name RENAME INDEX index_name TO index_name_lame_duck;
```

2. 创建新索引：

```
ALTER TABLE table_name ADD INDEX index_name (column_name);
```

3. 删除旧索引：

```
ALTER TABLE table_name DROP INDEX index_name_lame_duck;
```

8.11.3.5 第 4 步：移除无法恢复的节点（可选）

1. 缩容无法恢复的节点：

```
tiup cluster scale-in <cluster-name> -N <host> --force
```

2. 清理 Tombstone 节点：

```
tiup cluster prune <cluster-name>
```

1. 删除该 PersistentVolumeClaim。

```
kubectl delete -n ${namespace} pvc ${pvc_name} --wait=false
```

2. 删除 TiKV Pod，并等待新创建的 TiKV Pod 加入集群。

```
kubectl delete -n ${namespace} pod ${pod_name}
```

8.12 搭建双集群主从复制

本文档介绍如何配置一个 TiDB 集群以及该集群的 TiDB 或 MySQL 从集群，并将增量数据实时从主集群同步到从集群，主要包含以下内容：

1. 配置一个 TiDB 集群以及该集群的 TiDB 或 MySQL 从集群。
2. 将增量数据实时从主集群同步到从集群。
3. 在主集群发生灾难利用 Redo log 恢复一致性数据。

如果你需要配置一个运行中的 TiDB 集群和其从集群，以进行实时增量数据同步，可使用Backup & Restore (BR)和TiCDC。

8.12.1 第 1 步：搭建环境

1. 部署集群。

   使用 tiup playground 快速部署 TiDB 上下游测试集群。生产环境可以参考tiup 官方文档根据业务需求来部署集群。

   为了方便展示和理解，我们简化部署结构，需要准备以下两台机器，分别来部署上游主集群和下游从集群。假设 IP 地址分别为:

   - NodeA: 172.16.6.123，部署上游 TiDB
   - NodeB: 172.16.6.124，部署下游 TiDB

```
# 在 NodeA 上创建上游集群
tiup --tag upstream playground --host 0.0.0.0 --db 1 --pd 1 --kv 1 --tiflash 0 --ticdc 1
# 在 NodeB 上创建下游集群
tiup --tag downstream playground --host 0.0.0.0 --db 1 --pd 1 --kv 1 --tiflash 0 --ticdc 0
# 查看集群状态
tiup status
```

2. 初始化数据。

   测试集群中默认创建了 test 数据库，因此可以使用 sysbench 工具生成测试数据，用以模拟真实集群中的历史数据。

```
sysbench oltp_write_only --config-file=./tidb-config --tables=10 --table-size=10000 prepare
```

   这里通过 sysbench 运行 oltp_write_only 脚本，其将在测试数据库中生成 10 张表，每张表包含 10000 行初始数据。tidb-config 的配置如下：

```
mysql-host=172.16.6.122 # 这里需要替换为实际上游集群 ip
mysql-port=4000
mysql-user=root
mysql-password=
db-driver=mysql          # 设置数据库驱动为 mysql
mysql-db=test            # 设置测试数据库为 test
report-interval=10       # 设置定期统计的时间间隔为 10 秒
threads=10               # 设置 worker 线程数量为 10
time=0                   # 设置脚本总执行时间，0 表示不限制
rate=100                 # 设置平均事务速率 tps = 100
```

3. 模拟业务负载。

   实际生产集群的数据迁移过程中，通常原集群还会写入新的业务数据，本文中可以通过 sysbench 工具模拟持续的写入负载，下面的命令会使用 10 个 worker 在数据库中的 sbtest1、sbtest2 和 sbtest3 三张表中持续写入数据，其总 tps 限制为 100。

```
sysbench oltp_write_only --config-file=./tidb-config --tables=3 run
```

4. 准备外部存储。

   在全量数据备份中，上下游集群均需访问备份文件，因此推荐使用外部存储存储备份文件，本文中通过 Minio 模拟兼容 S3 的存储服务：

```
wget https://dl.min.io/server/minio/release/linux-amd64/minio
chmod +x minio
# 配置访问 minio 的 access-key access-secret-id
export `HOST_IP`='172.16.6.123' # 替换为实际部署 minio 的机器 ip
export** **MINIO_ROOT_USER**='**minio'
export MINIO_ROOT_PASSWORD='miniostorage'
# 创建 redo 和 backup 数据目录， 其中 redo, backup 为 bucket 名字
mkdir -p data/redo
mkdir -p data/backup
# 启动 minio, 暴露端口在 6060
nohup ./minio server ./data --address :6060 &
```

   上述命令行启动了一个单节点的 minio server 模拟 S3 服务，其相关参数为：

   - Endpoint：`http://${HOST_IP}:6060/`
   - Access-key：`minio`
   - Secret-access-key: `miniostorage`
   - Bucket: `redo`

   其访问链接为如下:

```
s3://backup?access-key=minio&secret-access-key=miniostorage&endpoint=http://${HOST_IP}:6060&
    ↪ force-path-style=true
```

8.12.2 第 2 步：迁移全量数据

搭建好测试环境后，可以使用 BR 工具的备份和恢复功能迁移全量数据。BR 工具有多种使用方式，本文中使用 SQL 语句 BACKUP 和 RESTORE 进行备份恢复。

> 注意：
>
> - 在生产集群中，关闭 GC 机制和备份操作会一定程度上降低集群的读性能，建议在业务低峰期进行备份，并设置合适的 RATE_LIMIT 限制备份操作对线上业务的影响。
> - 上下游集群版本不一致时，应检查 BR 工具的兼容性。本文假设上下游集群版本相同。

1. 关闭 GC。

   为了保证增量迁移过程中新写入的数据不丢失，在开始备份之前，需要关闭上游集群的垃圾回收 (GC) 机制，以确保系统不再清理历史数据。

   执行如下命令关闭 GC：

```
MySQL [test]> SET GLOBAL tidb_gc_enable=FALSE;
```

```
Query OK, 0 rows affected (0.01 sec)
```

查询 tidb_gc_enable 的取值，判断 GC 是否已关闭：

```
MySQL [test]> SELECT @@global.tidb_gc_enable;
```

```
+--------------------------+
| @@global.tidb_gc_enable |
+--------------------------+
|                        0 |
+--------------------------+
1 row in set (0.00 sec)
```

2. 备份数据。

   在上游集群中执行 BACKUP 语句备份数据：

```
MySQL [(none)]> BACKUP DATABASE * TO ``s3://backup?access-key=minio&secret-access-key=
    ↪ miniostorage&endpoint=http://${HOST_IP}:6060&force-path-style=true`` RATE_LIMIT =
    ↪ 120 MB/SECOND;
```

```
+----------------------+----------+--------------------+---------------------+--------------------+
    ↪
| Destination          | Size     | BackupTS           | Queue Time          | Execution
    ↪ Time      |
+----------------------+----------+--------------------+---------------------+--------------------+
    ↪
| local:///tmp/backup/ | 10315858 | 431434047157698561 | 2022-02-25 19:57:59 | 2022-02-25
    ↪ 19:57:59 |
+----------------------+----------+--------------------+---------------------+--------------------+
    ↪
1 row in set (2.11 sec)
```

   备份语句提交成功后，TiDB 会返回关于备份数据的元信息，这里需要重点关注 BackupTS，它意味着该时间点之前数据会被备份，后边的教程中，本文将使用 BackupTS 作为数据校验截止时间和 TiCDC 增量扫描的开始时间。

3. 恢复数据。

   在下游集群中执行 RESTORE 语句恢复数据：

```
mysql> RESTORE DATABASE * FROM ``s3://backup?access-key=minio&secret-access-key=miniostorage
    ↪ &endpoint=http://${HOST_IP}:6060&force-path-style=true``;
```

```
+---------------------+----------+--------------------+--------------------+--------------------+
    ↪
| Destination         | Size     | BackupTS           | Queue Time         | Execution
    ↪ Time      |
+---------------------+----------+--------------------+--------------------+--------------------+
    ↪
| local:///tmp/backup/ | 10315858 | 431434141450371074 | 2022-02-25 20:03:59 | 2022-02-25
    ↪ 20:03:59 |
+---------------------+----------+--------------------+--------------------+--------------------+
    ↪
1 row in set (41.85 sec)
```

4. （可选）校验数据。

   通过sync-diff-inspector 工具，可以验证上下游数据在某个时间点的一致性。从上述备份和恢复命令的输出可以看到，上游集群备份的时间点为 431434047157698561，下游集群完成数据恢复的时间点为 431434141450371074。

```
sync_diff_inspector -C ./config.yaml
```

   关于 sync-diff-inspector 的配置方法，请参考配置文件说明。在本文中，相应的配置如下：

```
# Diff Configuration.
######################### Global config #########################
check-thread-count = 4
export-fix-sql = true
check-struct-only = false

######################### Datasource config #########################
[data-sources]
[data-sources.upstream]
    host = "172.16.6.123" # 替换为实际上游集群 ip
    port = 4000
    user = "root"
    password = ""
    snapshot = "431434047157698561" # 配置为实际的备份时间点
[data-sources.downstream]
    host = "172.16.6.124" # 替换为实际下游集群 ip
    port = 4000
    user = "root"
    password = ""
    snapshot = "431434141450371074" # 配置为实际的恢复时间点

######################### Task config #########################
[task]
    output-dir = "./output"
```

```
    source-instances = ["upstream"]
    target-instance = "downstream"
    target-check-tables = ["*.*"]
```

8.12.3 第 3 步：迁移增量数据

1. 部署 TiCDC。

   完成全量数据迁移后，就可以部署并配置 TiCDC 集群同步增量数据，实际生产集群中请参考TiCDC 部署。本文在创建测试集群时，已经启动了一个 TiCDC 节点，因此可以直接进行 changefeed 的配置。

2. 创建同步任务。

   创建 changefeed 配置文件并保存为 changefeed.toml。

```
[consistent]
# 一致性级别，配置成 eventual 表示开启一致性复制
level = "eventual"
# 使用 S3 来存储 redo log，其他可选为 local，nfs
storage = "s3://redo?access-key=minio&secret-access-key=miniostorage&endpoint=http
    ↪ ://172.16.6.125:6060&force-path-style=true"
```

   在上游集群中，执行以下命令创建从上游到下游集群的同步链路：

```
tiup cdc cli changefeed create --server=http://172.16.6.122:8300 --sink-uri="mysql://root:
    ↪ @172.16.6.125:4000" --changefeed-id="primary-to-secondary" --start-ts
    ↪ ="431434047157698561"
```

   以上命令中：

   - --server：TiCDC 集群任意一节点的地址
   - --sink-uri：同步任务下游的地址
   - --start-ts：TiCDC 同步的起点，需要设置为实际的备份时间点（也就是第 2 步：迁移全量数据提到的 BackupTS）

   更多关于 changefeed 的配置，请参考TiCDC Changefeed 配置参数。

3. 重新开启 GC。

   TiCDC 可以保证未同步的历史数据不会被回收。因此，创建完从上游到下游集群的 changefeed 之后，就可以执行如下命令恢复集群的垃圾回收功能。详情请参考TiCDC GC safepoint 的完整行为。

执行如下命令打开 GC：

```
```sql
MySQL [test]> SET GLOBAL tidb_gc_enable=TRUE;
```


```
Query OK, 0 rows affected (0.01 sec)
```
```

```
```

查询 `tidb_gc_enable` 的取值，判断 GC 是否已开启：

```sql
MySQL [test]> SELECT @@global.tidb_gc_enable;
```

```
+--------------------------+
| @@global.tidb_gc_enable |
+--------------------------+
|                        1 |
+--------------------------+
1 row in set (0.00 sec)
```

8.12.4 第 4 步：模拟主集群故障

模拟在业务过程中上游 TiDB 发生灾难性故障无法再启动起来，这里可以直接使用 Ctrl + C 终止 tiup playground 进程。

8.12.5 第 5 步：使用 redo log 确保数据一致性

在正常同步过程中，为了提高 TiCDC 的吞吐能力，TiCDC 会将事务并行写入下游。因此，当 TiCDC 同步链路意外中断时，下游可能不会恰好停在与上游一致的状态。我们这里需要使用 TiCDC 的命令行工具来向下游重放 redo log，使下游达到最终一致性状态。

```
tiup cdc redo apply --storage "s3://redo?access-key=minio&secret-access-key=miniostorage&endpoint
    ↪ =http://172.16.6.123:6060&force-path-style=true" --tmp-dir /tmp/redo --sink-uri "mysql://
    ↪ root:@172.16.6.124:4000"
```

- `--storage`：指定 redo log 所在的 S3 位置以及 credential
- `--tmp-dir`：为从 S3 下载 redo log 的缓存目录
- `--sink-uri`：指定下游集群的地址

8.12.6 第 6 步：恢复主集群及业务

现在从集群有了某一时刻全部的一致性数据，你需要重新搭建主从集群来保证数据可靠性。

1. 在 NodeA 重新搭建一个新的 TiDB 集群作为新的主集群。

    ```
    tiup --tag upstream playground v5.4.0 --host 0.0.0.0 --db 1 --pd 1 --kv 1 --tiflash 0 --
        ↪ ticdc 1
    ```

2. 使用 BR 将从集群数据全量备份恢复到主集群。

```
# 全量备份从集群的数据
tiup br --pd http://172.16.6.124:2379 backup full --storage ./backup
# 全量恢复从集群的数据
tiup br --pd http://172.16.6.123:2379 restore full --storage ./backup
```

3. 创建一个 TiCDC 同步任务，备份主集群数据到从集群。

```
# 创建 changefeed
tiup cdc cli changefeed create --server=http://172.16.6.122:8300 --sink-uri="mysql://root:
    ↪ @172.16.6.125:4000" --changefeed-id="primary-to-secondary"
```

9 监控与告警

9.1 TiDB 监控框架概述

TiDB 使用开源时序数据库 Prometheus 作为监控和性能指标信息存储方案，使用 Grafana 作为可视化组件进行展示。

9.1.1 Prometheus 在 TiDB 中的应用

Prometheus 是一个拥有多维度数据模型的、灵活的查询语句的时序数据库。Prometheus 作为热门的开源项目，拥有活跃的社区及众多的成功案例。

Prometheus 提供了多个组件供用户使用。目前，TiDB 使用了以下组件：

- Prometheus Server：用于收集和存储时间序列数据。
- Client 代码库：用于定制程序中需要的 Metric。
- Alertmanager：用于实现报警机制。

其结构如下图所示：

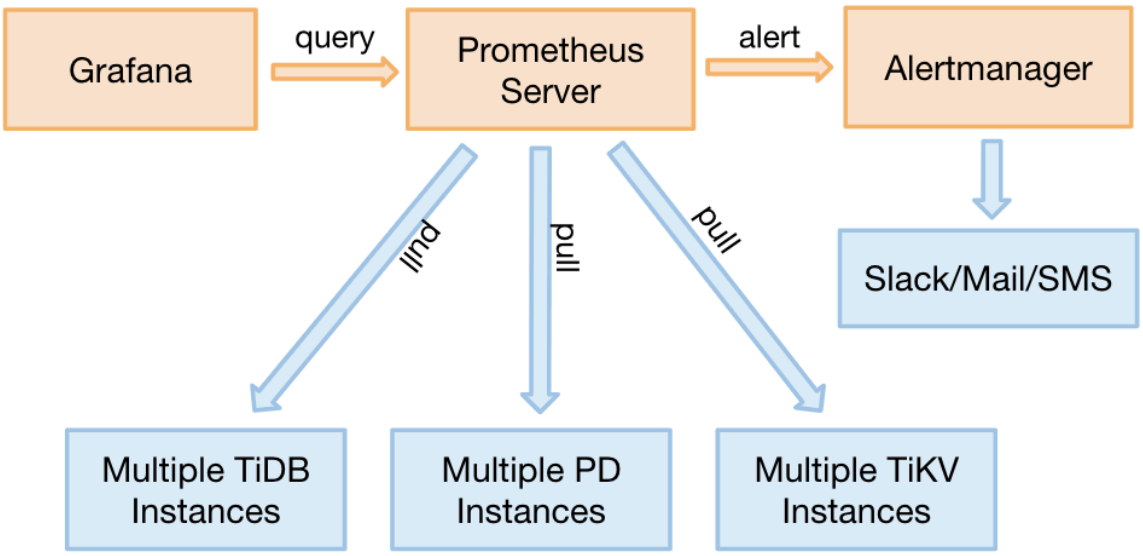


图 65: Prometheus in TiDB

9.1.2 Grafana 在 TiDB 中的应用

Grafana 是一个开源的 metric 分析及可视化系统。TiDB 使用 Grafana 来展示 TiDB 集群各组件的相关监控，监控项分组如下图所示：

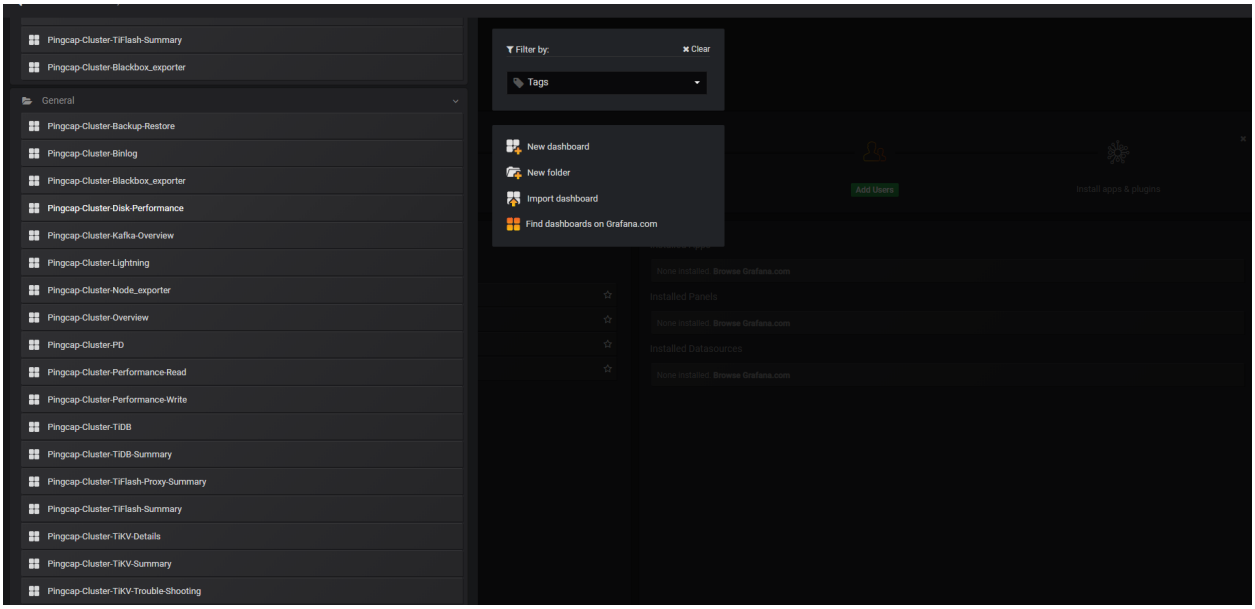


图 66: Grafana monitored_groups

- {TiDB_Cluster_name}-Backup-Restore：备份恢复相关的监控项。

- {TiDB_Cluster_name}-Binlog：TiDB Binlog 相关的监控项。
- {TiDB_Cluster_name}-Blackbox_exporter：网络探活相关监控项。
- {TiDB_Cluster_name}-Disk-Performance：磁盘性能相关监控项。
- {TiDB_Cluster_name}-Kafka-Overview：Kafka 相关监控项。
- {TiDB_Cluster_name}-Lightning：TiDB Lightning 组件相关监控项。
- {TiDB_Cluster_name}-Node_exporter：操作系统相关监控项。
- {TiDB_Cluster_name}-Overview：重要组件监控概览。
- {TiDB_Cluster_name}-PD：PD server 组件相关监控项。
- {TiDB_Cluster_name}-Performance-Read：读性能相关监控项。
- {TiDB_Cluster_name}-Performance-Write：写性能相关监控项。
- {TiDB_Cluster_name}-TiDB：TiDB server 组件详细监控项。
- {TiDB_Cluster_name}-TiDB-Summary：TiDB server 相关监控项概览。
- {TiDB_Cluster_name}-TiFlash-Proxy-Summary：数据同步到 TiFlash 的代理 server 监控项概览。
- {TiDB_Cluster_name}-TiFlash-Summary：TiFlash server 相关监控项概览。
- {TiDB_Cluster_name}-TiKV-Details：TiKV server 组件详细监控项。
- {TiDB_Cluster_name}-TiKV-Summary：TiKV server 监控项概览。
- {TiDB_Cluster_name}-TiKV-Trouble-Shooting：TiKV 错误诊断相关监控项。
- {TiDB_Cluster_name}-TiCDC：TiCDC 组件详细监控项。

每个分组包含多个监控项页签，页签中包含多个详细的监控项信息。以 Overview 监控组为例，其中包含 5 个页签，每个页签内有相应的监控指标看板，如下图所示：



图 67: Grafana Overview

要快速了解 TiDB 监控与报警系统的体系、该系统背后的数据流转方式、系统管理方法、系统使用方法和常用监控指标，建议观看下面的培训视频（时长 29 分钟）。注意本视频只作为学习参考，具体的监控指标与相关报警规则，请以文档内容为准。

9.2 TiDB 集群监控 API

TiDB 提供了以下几种接口来监控集群状态：

- 状态接口：通过 HTTP 接口对外汇报组件的信息。通过状态接口，你可获取当前 TiDB Server 的运行状态以及表的存储信息。
- Metrics 接口：使用 Prometheus 记录组件中各种操作的详细信息，使用 Grafana 进行可视化展示。

9.2.1 使用状态接口

状态接口用于监控组件的一些基本信息，并且可以作为 keepalive 的监测接口。另外，通过 PD 的状态接口可以看到整个 TiKV 集群的详细信息。

9.2.1.1 TiDB Server

- TiDB API 地址：`http://${host}:${port}`
- 默认端口：10080

9.2.1.1.1 运行状态

以下示例中，通过访问 `http://${host}:${port}/status` 获取当前 TiDB Server 的状态，并判断该 TiDB Server 是否存活。结果以 JSON 格式返回：

```
curl http://127.0.0.1:10080/status
```

```
{
    connections: 0,  # 当前 TiDB Server 上的客户端连接数
    version: "5.7.25-TiDB-v4.0.0-rc-141-g7267747ae",  # TiDB 版本号
    git_hash: "7267747ae0ec624dffc3fdedb00f1ed36e10284b"  # TiDB 当前代码的 Git Hash
}
```

9.2.1.1.2 存储信息

以下示例中，通过访问 `http://${host}:${port}/schema_storage/${db}/${table}` 获取指定数据表的存储信息。结果以 JSON 格式返回：

```
curl http://127.0.0.1:10080/schema_storage/mysql/stats_histograms
```

```
{
    "table_schema": "mysql",
    "table_name": "stats_histograms",
    "table_rows": 0,
    "avg_row_length": 0,
    "data_length": 0,
    "max_data_length": 0,
    "index_length": 0,
    "data_free": 0
}
```

```
curl http://127.0.0.1:10080/schema_storage/test
```

```
[
    {
        "table_schema": "test",
        "table_name": "test",
        "table_rows": 0,
        "avg_row_length": 0,
        "data_length": 0,
        "max_data_length": 0,
```

```
        "index_length": 0,
        "data_free": 0
    }
]
```

9.2.1.2 PD Server

- PD API 地址：`http://${host}:${port}/pd/api/v1/${api_name}`
- 默认端口：2379
- 各类 api_name 详细信息：参见 PD API Doc

通过该接口可以获取当前所有 TiKV 节点的状态以及负载均衡信息。下面以一个单节点的 TiKV 集群为例，说明用户需要了解的信息：

```
curl http://127.0.0.1:2379/pd/api/v1/stores
```

```
{
  "count": 1,  # TiKV 节点数量
  "stores": [  # TiKV 节点的列表
    # 集群中单个 TiKV 节点的信息
    {
      "store": {
        "id": 1,
        "address": "127.0.0.1:20160",
        "version": "4.0.0-rc.2",
        "status_address": "172.16.5.90:20382",
        "git_hash": "2fdb2804bf8ffaab4b18c4996970e19906296497",
        "start_timestamp": 1590029618,
        "deploy_path": "/data2/tidb_test/v4.0.rc.2/tikv-20372/bin",
        "last_heartbeat": 1590030038949235439,
        "state_name": "Up"
      },
      "status": {
        "capacity": "3.581TiB",  # 存储总容量
        "available": "3.552TiB",  # 存储剩余容量
        "used_size": "31.77MiB",
        "leader_count": 174,
        "leader_weight": 1,
        "leader_score": 174,
        "leader_size": 174,
        "region_count": 531,
        "region_weight": 1,
        "region_score": 531,
        "region_size": 531,
        "start_ts": "2020-05-21T10:53:38+08:00",
```

```
        "last_heartbeat_ts": "2020-05-21T11:00:38.949235439+08:00",
        "uptime": "7m0.949235439s"
      }
    }
  ]
```

9.2.2 使用 metrics 接口

Metrics 接口用于监控整个集群的状态和性能。

- 如果使用其他方式部署 TiDB 集群，在使用 metrics 接口前，需先部署 Prometheus 和 Grafana。

成功部署 Prometheus 和 Grafana 之后，配置 Grafana。

9.3 TiDB 集群监控部署

本文档适用于希望手动部署 TiDB 监控报警系统的用户。TiUP 部署方式，会同时自动部署监控报警系统，无需手动部署。

9.3.1 部署 Prometheus 和 Grafana

假设 TiDB 的拓扑结构如下：

节点	主机 IP	服务
Node1	192.168.199.113	PD1, TiDB, node_export, Prometheus, Grafana
Node2	192.168.199.114	PD2, node_export
Node3	192.168.199.115	PD3, node_export
Node4	192.168.199.116	TiKV1, node_export
Node5	192.168.199.117	TiKV2, node_export
Node6	192.168.199.118	TiKV3, node_export

9.3.1.1 第 1 步：下载二进制包

下载二进制包：

```
wget https://download.pingcap.org/prometheus-2.27.1.linux-amd64.tar.gz
wget https://download.pingcap.org/node_exporter-v1.3.1-linux-amd64.tar.gz
wget https://download.pingcap.org/grafana-7.5.11.linux-amd64.tar.gz
```

解压二进制包：

```
tar -xzf prometheus-2.27.1.linux-amd64.tar.gz
tar -xzf node_exporter-v1.3.1-linux-amd64.tar.gz
tar -xzf grafana-7.5.11.linux-amd64.tar.gz
```

9.3.1.2 第 2 步：在 Node1，Node2，Node3，Node4 上启动 node_exporter

```
cd node_exporter-v1.3.1-linux-amd64
```

启动 node_exporter 服务：

```
./node_exporter --web.listen-address=":9100" \
    --log.level="info" &
```

9.3.1.3 第 3 步：在 Node1 上启动 Prometheus

编辑 Prometheus 的配置文件：

```
cd prometheus-2.27.1.linux-amd64 &&
vi prometheus.yml
```

```
...

global:
  scrape_interval:     15s
  evaluation_interval: 15s
  # scrape_timeout 设置为全局默认值 (10s)
  external_labels:
    cluster: 'test-cluster'
    monitor: "prometheus"

scrape_configs:
  - job_name: 'overwritten-nodes'
    honor_labels: true  # 不要覆盖 job 和实例的 label
    static_configs:
    - targets:
      - '192.168.199.113:9100'
      - '192.168.199.114:9100'
      - '192.168.199.115:9100'
      - '192.168.199.116:9100'
      - '192.168.199.117:9100'
      - '192.168.199.118:9100'

  - job_name: 'tidb'
    honor_labels: true  # 不要覆盖 job 和实例的 label
    static_configs:
    - targets:
      - '192.168.199.113:10080'

  - job_name: 'pd'
    honor_labels: true  # 不要覆盖 job 和实例的 label
```

```
    static_configs:
    - targets:
      - '192.168.199.113:2379'
      - '192.168.199.114:2379'
      - '192.168.199.115:2379'

  - job_name: 'tikv'
    honor_labels: true  # 不要覆盖 job 和实例的 label
    static_configs:
    - targets:
      - '192.168.199.116:20180'
      - '192.168.199.117:20180'
      - '192.168.199.118:20180'

...
```

启动 Prometheus 服务：

```
./prometheus \
    --config.file="./prometheus.yml" \
    --web.listen-address=":9090" \
    --web.external-url="http://192.168.199.113:9090/" \
    --web.enable-admin-api \
    --log.level="info" \
    --storage.tsdb.path="./data.metrics" \
    --storage.tsdb.retention="15d" &
```

9.3.1.4 第 4 步：在 Node1 上启动 Grafana

编辑 Grafana 的配置文件：

```
cd grafana-7.5.11 &&
vi conf/grafana.ini
```

```
...

[paths]
data = ./data
logs = ./data/log
plugins = ./data/plugins
[server]
http_port = 3000
domain = 192.168.199.113
[database]
[session]
[analytics]
```

```
check_for_updates = true
[security]
admin_user = admin
admin_password = admin
[snapshots]
[users]
[auth.anonymous]
[auth.basic]
[auth.ldap]
[smtp]
[emails]
[log]
mode = file
[log.console]
[log.file]
level = info
format = text
[log.syslog]
[event_publisher]
[dashboards.json]
enabled = false
path = ./data/dashboards
[metrics]
[grafana_net]
url = https://grafana.net

...
```

启动 Grafana 服务：

```
./bin/grafana-server \
    --config="./conf/grafana.ini" &
```

9.3.2 配置 Grafana

本小节介绍如何配置 Grafana。

9.3.2.1 第 1 步：添加 Prometheus 数据源

1. 登录 Grafana 界面。

    - **默认地址：** `http://localhost:3000`
    - **默认账户：** admin
    - **默认密码：** admin

> **注意：**
>
> Change Password 步骤可以选择 Skip。

2. 点击 Grafana 侧边栏菜单 Configuration 中的 Data Source。
3. 点击 Add data source。
4. 指定数据源的相关信息：
    - 在 Name 处，为数据源指定一个名称。
    - 在 Type 处，选择 Prometheus。
    - 在 URL 处，指定 Prometheus 的 IP 地址。
    - 根据需求指定其它字段。
5. 点击 Add 保存新的数据源。

9.3.2.2 第 2 步：导入 Grafana 面板

执行以下步骤，为 PD Server、 TiKV Server 和 TiDB Server 分别导入 Grafana 面板：

1. 点击侧边栏的 Grafana 图标。
2. 在侧边栏菜单中，依次点击 Dashboards > Import 打开 Import Dashboard 窗口。
3. 点击 Upload .json File 上传对应的 JSON 文件（从 pingcap/tidb、 tikv/tikv 和 tikv/pd 下载 TiDB Grafana 配置文件）。

> **注意：**
>
> TiKV、 PD 和 TiDB 面板对应的 JSON 文件分别为 `tikv_summary.json`，`tikv_details.json`，`tikv_trouble_shooting.json`，`pd.json`，`tidb.json`，`tidb_summary.json`。

4. 点击 Load。
5. 选择一个 Prometheus 数据源。
6. 点击 Import，Prometheus 面板即导入成功。

9.3.3 查看组件 metrics

在顶部菜单中，点击 New dashboard，选择要查看的面板。

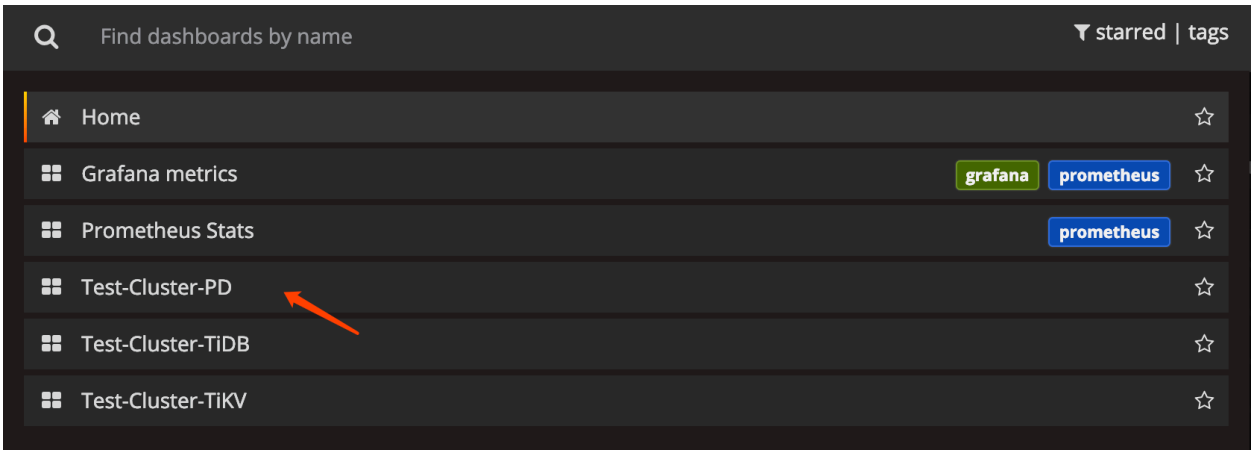


图 68: view dashboard

可查看以下集群组件信息：

- TiDB Server:
  - query 处理时间，可以看到延迟和吞吐
  - ddl 过程监控
  - TiKV client 相关的监控
  - PD client 相关的监控
- PD Server:
  - 命令执行的总次数
  - 某个命令执行失败的总次数
  - 某个命令执行成功的耗时统计
  - 某个命令执行失败的耗时统计
  - 某个命令执行完成并返回结果的耗时统计
- TiKV Server:
  - GC 监控
  - 执行 KV 命令的总次数
  - Scheduler 执行命令的耗时统计
  - Raft propose 命令的总次数
  - Raft 执行命令的耗时统计
  - Raft 执行命令失败的总次数
  - Raft 处理 ready 状态的总次数

> 警告：
>
> - 从 TiDB v6.0.0 起，PingCAP 不再维护 MetricsTool 工具。从 v6.1.0 起，不再维护 MetricsTool 的文档。

- 如需导出监控数据，建议使用PingCAP Clinic 诊断服务一键导出诊断 TiDB 集群时所需要的信息，包括监控数据、日志、集群拓扑、配置、参数等。

9.4 将 Grafana 监控数据导出成快照

注意：

目前该工具仅支持在 Grafana v6.x.x 上使用。

在故障诊断中，监控数据十分重要。当你请求远程协助时，技术支持人员有时需要查看 Grafana Dashboard 以确认问题所在。MetricsTool 用于将 Grafana Dashboard 的快照导出为本地文件，并将快照可视化。因此，你可以在不泄露 Grafana 服务器上其他敏感信息的前提下，将监控数据以快照形式分享给外部人员，同时也方便外部人员准确识读数据图表。

9.4.1 使用方法

可以通过访问 <https://metricstool.pingcap.net> 来使用 MetricsTool。它主要提供以下三种功能：

- 导出快照：提供一段在浏览器开发者工具上运行的用户脚本。你可以使用这个脚本在任意 Grafana v6.x.x 服务器上下载当前 Dashboard 中所有可见面板的快照。

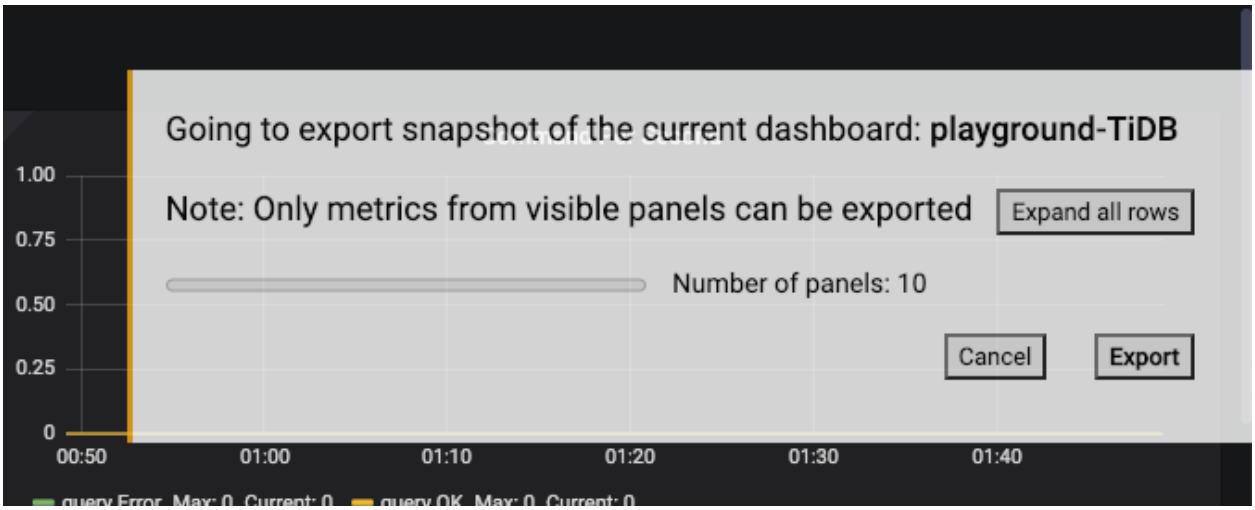


图 69: 运行用户脚本后的 MetricsTool Exporter 截图

- 快照可视化：通过网页端可视化工具 Visualizer 将快照导出文件可视化。快照经过可视化后，操作起来与实际的 Grafana Dashboard 无异。

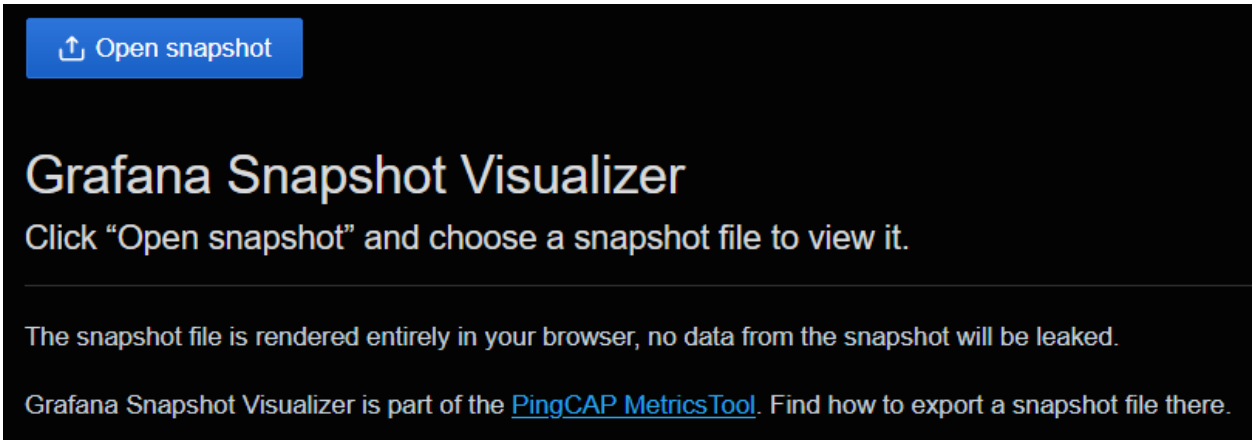


图 70: MetricsTool Visualizer 截图

- 导入快照：介绍如何将导出的快照重新导入到已有的 Grafana 实例中。

9.4.2 FAQs

9.4.2.1 与直接截图及导出 PDF 相比，MetricTool 有什么优势？

MetricsTool 导出的快照文件包含快照生成时的监控指标实际数值。你可以通过 Visualizer 与渲染的图表进行交互，比如切换序列、选择一个较小的时间范围以及检查特定时间点的监控数据值等，就像在操作一个实际的 Grafana Dashboard 一样，因此它比 PDF 文件和截图更强大。

9.4.2.2 快照文件里都包含什么？

快照文件包含所选时间范围内所有图表和面板数据的值，但不保存数据源的原始监控指标，所以无法在 Visualizer 中编辑查询表达式。

9.4.2.3 Visualizer 会将上传的快照文件保存到 PingCAP 的服务器上吗？

不会。快照文件解析全部在浏览器中完成，Visualizer 不会将任何信息发送给 PingCAP。你可以放心地使用 Visualizer 查看带有敏感信息的快照文件，不用担心信息会泄露给第三方。

9.4.2.4 可以在所有监控指标数据都加载完毕前就运行脚本吗？

可以。虽然脚本会弹出提示，让你等所有监控数据加载完毕后再运行，但可以手动跳过等待并导出快照，以免有些监控数据加载的时间过长。

9.4.2.5 快照文件可视化后，可以通过网页链接分享吗？

不能。但你可以分享快照文件，并说明如何使用 Visualizer 查看。如果确实需要通过网页链接分享，可以尝试使用 Grafana 内置的 `snapshot.raintank.io` 服务。但在这样做之前，要确保不会泄漏隐私信息。

9.5 TiDB 集群报警规则

本文介绍了 TiDB 集群中各组件的报警规则，包括 TiDB、 TiKV、 PD、 TiFlash、 TiDB Binlog、 TiCDC、 Node_exporter 和 Blackbox_exporter 的各报警项的规则描述及处理方法。

按照严重程度由高到低，报警项可分为紧急级别 > 严重级别 > 警告级别三类。该分级适用于以下各组件的报警项。

严重程度	说明
紧急级别	最高严重程度，服务不可用，通常由于服务停止或节点故障导致，此时需要马上进行人工干预
严重级别	服务可用性下降，需要用户密切关注异常指标
警告级别	对某一问题或错误的提醒

9.5.1 TiDB 报警规则

本节介绍了 TiDB 组件的报警项。

9.5.1.1 紧急级别报警项

9.5.1.1.1 TiDB_schema_error

- 报警规则：

  ```
  increase(tidb_session_schema_lease_error_total{type="outdated"}[15m])> 0
  ```

- 规则描述：

  TiDB 在一个 Lease 时间内没有重载到最新的 Schema 信息。如果 TiDB 无法继续对外提供服务，则报警。

- 处理方法：

  该问题通常由于 TiKV Region 不可用或超时导致，需要看 TiKV 的监控指标定位问题。

9.5.1.1.2 TiDB_tikvclient_region_err_total

- 报警规则：

  ```
  increase(tidb_tikvclient_region_err_total[10m])> 6000
  ```

- 规则描述：

  TiDB 访问 TiKV 时发生了 Region 错误。如果在 10 分钟之内该错误多于 6000 次，则报警。

- 处理方法：

  查看 TiKV 的监控状态。

9.5.1.1.3 TiDB_domain_load_schema_total

- 报警规则：

  ```
  increase(tidb_domain_load_schema_total{type="failed"}[10m])> 10
  ```

- 规则描述：

  TiDB 重载最新的 Schema 信息失败的总次数。如果在 10 分钟之内重载失败次数超过 10 次，则报警。

- 处理方法：

  参考TiDB_schema_error 的处理方法。

9.5.1.1.4 TiDB_monitor_keep_alive

- 报警规则：

  ```
  increase(tidb_monitor_keep_alive_total[10m])< 100
  ```

- 规则描述：

  表示 TiDB 的进程是否仍然存在。如果在 10 分钟之内 `tidb_monitor_keep_alive_total` 增加次数少于 100，则 TiDB 的进程可能已经退出，此时会报警。

- 处理方法：

  - 检查 TiDB 进程是否 OOM。
  - 检查机器是否发生了重启。

9.5.1.2 严重级别报警项

9.5.1.2.1 TiDB_server_panic_total

- 报警规则：

  ```
  increase(tidb_server_panic_total[10m])> 0
  ```

- 规则描述：

  发生崩溃的 TiDB 线程的数量。当出现崩溃的时候会报警。该线程通常会被恢复，否则 TiDB 会频繁重启。

- 处理方法：

  收集 panic 日志，定位原因。

9.5.1.3 警告级别报警项

9.5.1.3.1 TiDB_memory_abnormal

- 报警规则：

```
go_memstats_heap_inuse_bytes{job="tidb"} > 1e+10
```

- 规则描述：

  对 TiDB 内存使用量的监控。如果内存使用大于 10 G，则报警。

- 处理方法：

  通过 HTTP API 来排查 goroutine 泄露的问题。

9.5.1.3.2 TiDB_query_duration

- 报警规则：

```
histogram_quantile(0.99, sum(rate(tidb_server_handle_query_duration_seconds_bucket[1m]))BY (
↪ le, instance))> 1
```

- 规则描述：

  TiDB 处理请求的延时。如果延迟大于 1 秒的概率超过 99%，则报警。

- 处理方法：

  查看 TiDB 的日志，搜索 SLOW_QUERY 和 TIME_COP_PROCESS 关键字，查找慢 SQL。

9.5.1.3.3 TiDB_server_event_error

- 报警规则：

```
increase(tidb_server_event_total{type=~"server_start|server_hang"}[15m])> 0
```

- 规则描述：

  TiDB 服务中发生的事件数量。当出现以下事件的时候会报警：

  1. start：TiDB 服务启动。
  2. hang：当发生了 Critical 级别的事件时（目前只有 Binlog 写不进去一种情况），TiDB 进入 hang 模式，并等待人工 Kill。

- 处理方法：

  - 重启 TiDB 以恢复服务。
  - 检查 TiDB Binlog 服务是否正常。

9.5.1.3.4 TiDB_tikvclient_backoff_seconds_count

- 报警规则：

  ```
  increase(tidb_tikvclient_backoff_seconds_count[10m])> 10
  ```

- 规则描述：

  TiDB 访问 TiKV 发生错误时发起重试的次数。如果在 10 分钟之内重试次数多于 10 次，则报警。

- 处理方法：

  查看 TiKV 的监控状态。

9.5.1.3.5 TiDB_monitor_time_jump_back_error

- 报警规则：

  ```
  increase(tidb_monitor_time_jump_back_total[10m])> 0
  ```

- 规则描述：

  如果 TiDB 所在机器的时间发生了回退，则报警。

- 处理方法：

  排查 NTP 配置。

9.5.1.3.6 TiDB_ddl_waiting_jobs

- 报警规则：

  ```
  sum(tidb_ddl_waiting_jobs)> 5
  ```

- 规则描述：

  如果 TiDB 中等待执行的 DDL 任务的数量大于 5，则报警。

- 处理方法：

  通过 `admin show ddl` 语句检查是否有耗时的 add index 操作正在执行。

9.5.2 PD 报警规则

本节介绍了 PD 组件的报警项。

9.5.2.1 紧急级别报警项

9.5.2.1.1 PD_cluster_down_store_nums

- 报警规则：

```
(sum(pd_cluster_status{type="store_down_count"})by (instance)> 0)and (sum(etcd_server_is_leader
↪ )by (instance)> 0)
```

- 规则描述：

  PD 长时间（默认配置是 30 分钟）没有收到 TiKV/TiFlash 心跳。

- 处理方法：

  - 检查 TiKV/TiFlash 进程是否正常、网络是否隔离以及负载是否过高，并尽可能地恢复服务。
  - 如果确定 TiKV/TiFlash 无法恢复，可做下线处理。

9.5.2.2 严重级别报警项

9.5.2.2.1 PD_etcd_write_disk_latency

- 报警规则：

```
histogram_quantile(0.99, sum(rate(etcd_disk_wal_fsync_duration_seconds_bucket[1m]))by (
↪ instance, job, le))> 1
```

- 规则描述：

  fsync 操作延迟大于 1s，代表 etcd 写盘慢，这很容易引起 PD leader 超时或者 TSO 无法及时存盘等问题，从而导致整个集群停止服务。

- 处理方法：

  - 排查写入慢的原因。可能是由于其他服务导致系统负载过高。可以检查 PD 本身是否占用了大量 CPU 或 IO 资源。
  - 可尝试重启 PD 或手动 transfer leader 至其他的 PD 来恢复服务。
  - 如果由于环境原因无法恢复，可将有问题的 PD 下线替换。

9.5.2.2.2 PD_miss_peer_region_count

- 报警规则：

```
(sum(pd_regions_status{type="miss-peer-region-count"})by (instance)> 100)and (sum(etcd_server_is_leader
↪ )by (instance)> 0)
```

- 规则描述：

  Region 的副本数小于 `max-replicas` 配置的值。

- 处理方法：

  - 查看是否有 TiKV 宕机或在做下线操作，尝试定位问题产生的原因。
  - 观察 region health 面板，查看 `miss-peer-region-count` 是否在不断减少。

9.5.2.3 警告级别报警项

9.5.2.3.1 PD_cluster_lost_connect_store_nums

- 报警规则：

```
(sum(pd_cluster_status{type="store_disconnected_count"})by (instance)> 0)and (sum(etcd_server_is_leader
↪ )by (instance)> 0)
```

- 规则描述：

  PD 在 20 秒之内未收到 TiKV/TiFlash 上报心跳。正常情况下是每 10 秒收到 1 次心跳。

- 处理方法：

  - 排查是否在重启 TiKV/TiFlash。
  - 检查 TiKV/TiFlash 进程是否正常、网络是否隔离以及负载是否过高，并尽可能地恢复服务。
  - 如果确定 TiKV/TiFlash 无法恢复，可做下线处理。
  - 如果确定 TiKV/TiFlash 可以恢复，但在短时间内还无法恢复，可以考虑延长 `max-down-time` 配置，防止超时后 TiKV/TiFlash 被判定为无法恢复并开始搬移数据。

9.5.2.3.2 PD_cluster_unhealthy_tikv_nums

- 报警规则：

```
(sum(pd_cluster_status{type="store_unhealth_count"})by (instance)> 0)and (sum(etcd_server_is_leader
↪ )by (instance)> 0)
```

- 规则描述：

  表示存在异常状态的 Store。如果这种状态持续一段时间（取决于配置的`max-store-down-time`，默认值为 `30m`），Store 可能会变为 `Offline` 状态并触发`PD_cluster_down_store_nums` 报警。

- 处理方法：

  检查 TiKV Store 的状态。

9.5.2.3.3 PD_cluster_low_space

- 报警规则：

```
(sum(pd_cluster_status{type="store_low_space_count"})by (instance)> 0)and (sum(etcd_server_is_leader
↪ )by (instance)> 0)
```

- 规则描述：

  表示 TiKV/TiFlash 节点空间不足。

- 处理方法：

  - 检查集群中的空间是否普遍不足。如果是，则需要扩容。
  - 检查 Region balance 调度是否有问题。如果有问题，会导致数据分布不均衡。
  - 检查是否有文件占用了大量磁盘空间，比如日志、快照、core dump 等文件。
  - 降低该节点的 Region weight 来减少数据量。
  - 无法释放空间时，可以考虑主动下线该节点，防止由于磁盘空间不足而宕机。

9.5.2.3.4 PD_etcd_network_peer_latency

- 报警规则：

```
histogram_quantile(0.99, sum(rate(etcd_network_peer_round_trip_time_seconds_bucket[1m]))by (
↪ To, instance, job, le))> 1
```

- 规则描述：

  PD 节点之间网络延迟高，严重情况下会导致 leader 超时和 TSO 存盘超时，从而影响集群服务。

- 处理方法：

  - 检查网络状况和系统负载情况。
  - 如果由于环境原因无法恢复，可将有问题的 PD 下线替换。

9.5.2.3.5 PD_tidb_handle_requests_duration

- 报警规则：

```
histogram_quantile(0.99, sum(rate(pd_client_request_handle_requests_duration_seconds_bucket{
↪ type="tso"}[1m]))by (instance, job, le))> 0.1
```

- 规则描述：

  PD 处理 TSO 请求耗时过长，一般是由于负载过高。

- 处理方法：

  - 检查服务器负载状况。
  - 使用 pprof 抓取 PD 的 CPU profile 进行分析。
  - 手动切换 PD leader。
  - 如果是环境问题，则将有问题的 PD 下线替换。

9.5.2.3.6 PD_down_peer_region_nums

- 报警规则：

```
(sum(pd_regions_status{type="down-peer-region-count"})by (instance)> 0)and (sum(etcd_server_is_leader
↪ )by (instance)> 0)
```

- 规则描述：

  Raft leader 上报有不响应 peer 的 Region 数量。

- 处理方法：

  - 检查是否有 TiKV 宕机，或刚发生重启，或者繁忙。
  - 观察 region health 面板，检查 down_peer_region_count 是否在不断减少。
  - 检查是否有 TiKV 之间网络不通。

9.5.2.3.7 PD_pending_peer_region_count

- 报警规则：

```
(sum(pd_regions_status{type="pending-peer-region-count"})by (instance)> 100)and (sum(etcd_server_is_leader
↪ )by (instance)> 0)
```

- 规则描述：

  Raft log 落后的 Region 过多。由于调度产生少量的 pending peer 是正常的，但是如果持续很高，就可能有问题。

- 处理方法：

  - 观察 region health 面板，检查 pending_peer_region_count 是否在不断减少。
  - 检查 TiKV 之间的网络状况，特别是带宽是否足够。

9.5.2.3.8 PD_leader_change

- 报警规则：

```
count(changes(pd_tso_events{type="save"}[10m])> 0)>= 2
```

- 规则描述：

  近期发生了 PD leader 切换。

- 处理方法：

  - 排除人为因素，比如重启 PD、手动 transfer leader 或调整 leader 优先级等。
  - 检查网络状况和系统负载情况。
  - 如果由于环境原因无法恢复，可将有问题的 PD 下线替换。

9.5.2.3.9 TiKV_space_used_more_than_80%

- 报警规则：

```
sum(pd_cluster_status{type="storage_size"})/ sum(pd_cluster_status{type="storage_capacity"})
↪ * 100 > 80
```

- 规则描述：

  集群空间占用超过 80%。

- 处理方法：

  - 确认是否需要扩容。
  - 排查是否有文件占用了大量磁盘空间，比如日志、快照或 core dump 等文件。

9.5.2.3.10 PD_system_time_slow

- 报警规则：

  ```
  changes(pd_tso_events{type="system_time_slow"}[10m])>= 1
  ```

- 规则描述：

  系统时间可能发生回退。

- 处理方法：

  检查系统时间设置是否正确。

9.5.2.3.11 PD_no_store_for_making_replica

- 报警规则：

  ```
  increase(pd_checker_event_count{type="replica_checker", name="no_target_store"}[1m])> 0
  ```

- 规则描述：

  没有合适的 store 用来补副本。

- 处理方法：

  - 检查 store 是否空间不足。
  - 根据 label 配置（如果有这个配置的话）来检查是否有可以补副本的 store。

9.5.2.3.12 PD_cluster_slow_tikv_nums

- 报警规则：

  ```
  sum(pd_cluster_status{type="store_slow_count"})by (instance)> 0)and (sum(etcd_server_is_leader
  ↪ )by (instance)> 0
  ```

- 规则描述：

  某一个 TiKV 被检测为慢节点。慢节点的检测由 TiKV `raftstore.inspect-interval` 参数控制，参见TiKV 配置文件描述。

- 处理方法：

  - 观察TiKV-Details > PD 面板，查看 Store Slow Score 监控指标，找出指标数值超过 80 的节点，该节点即为被检测到的慢节点。
  - 观察TiKV-Details > Raft IO 面板，查看延迟是否升高。如果延迟很高，表明磁盘可能存在瓶颈。
  - 调大 TiKV `raftstore.inspect-interval` 参数，提高延迟检测的超时上限。
  - 如果需要进一步分析报警的 TiKV 节点的性能问题，找到优化方法，可以参考性能分析和优化方法。

9.5.3 TiKV 报警规则

本节介绍了 TiKV 组件的报警项。

9.5.3.1 紧急级别报警项

9.5.3.1.1 TiKV_memory_used_too_fast

- 报警规则：

```
process_resident_memory_bytes{job=~"tikv",instance=~".*"} - (process_resident_memory_bytes{
↪ job=~"tikv",instance=~".*"} offset 5m)> 5*1024*1024*1024
```

- 规则描述：

  目前没有和内存相关的 TiKV 的监控，你可以通过 Node_exporter 监控集群内机器的内存使用情况。如上规则表示，如果在 5 分钟之内内存使用超过 5GB（TiKV 内存占用的太快），则报警。

- 处理方法：

  调整 `rocksdb.defaultcf` 和 `rocksdb.writecf` 的 `block-cache-size` 的大小。

9.5.3.1.2 TiKV_GC_can_not_work

- 报警规则：

```
sum(increase(tikv_gcworker_gc_tasks_vec{task="gc"}[1d]))< 1 and (sum(increase(tikv_gc_compaction_filter_per
↪ [1d]))< 1 and sum(increase(tikv_engine_event_total{db="kv", cf="write", type="compaction
↪ "}[1d]))>= 1)
```

- 规则描述：

  在 24 小时内一个 TiKV 实例上没有成功执行 GC，说明 GC 不能正常工作了。短期内 GC 不运行不会造成太大的影响，但如果 GC 一直不运行，版本会越来越多，从而导致查询变慢。

- 处理方法：

  1. 执行 `SELECT VARIABLE_VALUE FROM mysql.tidb WHERE VARIABLE_NAME="tikv_gc_leader_desc"` 来找到 gc leader 对应的 `tidb-server`；
  2. 查看该 `tidb-server` 的日志，grep gc_worker tidb.log；
  3. 如果发现这段时间一直在 resolve locks（最后一条日志是 `start resolve locks`）或者 delete ranges（最后一条日志是 `start delete {number} ranges`），说明 GC 进程是正常的。否则请从 PingCAP 官方或 TiDB 社区获取支持。

9.5.3.2 严重级别报警项

9.5.3.2.1 TiKV_server_report_failure_msg_total

- 报警规则：

```
sum(rate(tikv_server_report_failure_msg_total{type="unreachable"}[10m]))BY (store_id)> 10
```

- 规则描述：

  表明无法连接远端的 TiKV。

- 处理方法：

  1. 检查网络是否通畅。
  2. 检查远端 TiKV 是否挂掉。
  3. 如果远端 TiKV 没有挂掉，检查压力是否太大，参考TiKV_channel_full_total 处理方法。

9.5.3.2.2 TiKV_channel_full_total

- 报警规则：

```
sum(rate(tikv_channel_full_total[10m]))BY (type, instance)> 0
```

- 规则描述：

  该错误通常是因为 Raftstore 线程卡死，TiKV 的压力已经非常大了。

- 处理方法：

  1. 观察TiKV-Details > Raft Propose 面板，查看这个报警的 TiKV 节点是否明显比其他 TiKV 高很多。如果是，表明这个 TiKV 上有热点，需要检查热点调度是否能正常工作。
  2. 观察TiKV-Details > Raft IO 面板，查看延迟是否升高。如果延迟很高，表明磁盘可能存在瓶颈。
  3. 观察TiKV-Details > Raft process 面板，关注 `tick duration` 是否很高。如果是，需要将 TiKV 配置项`raftstore.raft-base-tick-interval` 设置为 `"2s"`。

9.5.3.2.3 TiKV_write_stall

- 报警规则：

```
delta(tikv_engine_write_stall[10m])> 0
```

- 规则描述：

  RocksDB 写入压力太大，出现了 stall。

- 处理方法：

  1. 观察磁盘监控，排除磁盘问题。
  2. 看 TiKV 是否有写入热点。
  3. 在 `[rocksdb]` 和 `[raftdb]` 配置下调大 `max-sub-compactions` 的值。

9.5.3.2.4 TiKV_raft_log_lag

- 报警规则：

```
histogram_quantile(0.99, sum(rate(tikv_raftstore_log_lag_bucket[1m]))by (le, instance))>
↪ 5000
```

- 规则描述：

  这个值偏大，表明 Follower 已经远远落后于 Leader，Raft 没法正常同步了。可能的原因是 Follower 所在的 TiKV 卡住或者挂掉了。

9.5.3.2.5 TiKV_async_request_snapshot_duration_seconds

- 报警规则：

```
histogram_quantile(0.99, sum(rate(tikv_storage_engine_async_request_duration_seconds_bucket{
↪ type="snapshot"}[1m]))by (le, instance, type))> 1
```

- 规则描述：

  这个值偏大，表明 Raftstore 负载压力很大，可能已经卡住。

- 处理方法：

  参考TiKV_channel_full_total 的处理方法。

9.5.3.2.6 TiKV_async_request_write_duration_seconds

- 报警规则：

```
histogram_quantile(0.99, sum(rate(tikv_storage_engine_async_request_duration_seconds_bucket{
↪ type="write"}[1m]))by (le, instance, type))> 1
```

- 规则描述：

  这个值偏大，表明 Raft write 耗时很长。

- 处理方法：

  1. 观察TiKV-Details > Raft propose 面板，查看这个报警的 TiKV 节点的 99% Propose wait duration per server 是否明显比其他 TiKV 高很多。如果是，表明这个 TiKV 上有热点，需要检查热点调度是否能正常工作。
  2. 观察TiKV-Details > Raft IO 面板，查看延迟是否升高。如果延迟很高，表明磁盘可能存在瓶颈。
  3. 如果需要进一步分析报警的 TiKV 节点的性能问题，找到优化方法，可以参考性能分析和优化方法。

9.5.3.2.7 TiKV_coprocessor_request_wait_seconds

- 报警规则：

```
histogram_quantile(0.9999, sum(rate(tikv_coprocessor_request_wait_seconds_bucket[1m]))by (le
↪ , instance, req))> 10
```

- 规则描述：

  这个值偏大，表明 Coprocessor worker 压力很大。可能有比较慢的任务卡住了 Coprocessor 线程。

- 处理方法：

  1. 从 TiDB 日志中查看慢查询日志，看查询是否用到了索引或全表扫，或者看是否需要做 analyze。
  2. 排查是否有热点。
  3. 查看 Coprocessor 监控，看 `coporcessor table/index scan` 里 `total` 和 `process` 是否匹配。如果相差太大，表明做了太多的无效查询。看是否有 `over seek bound`，如果有，表明版本太多，GC 工作不及时，需要增大并行 GC 的线程数。

9.5.3.2.8 TiKV_raftstore_thread_cpu_seconds_total

- 报警规则：

```
sum(rate(tikv_thread_cpu_seconds_total{name=~"raftstore_.*"}[1m]))by (instance, name)> 1.6
```

- 规则描述：

Raftstore 线程压力太大。

- 处理方法：

参考TiKV_channel_full_total 的处理方法。

9.5.3.2.9 TiKV_raft_append_log_duration_secs

- 报警规则：

```
histogram_quantile(0.99, sum(rate(tikv_raftstore_append_log_duration_seconds_bucket[1m]))by
↪ (le, instance))> 1
```

- 规则描述：

表示 append Raft log 的耗时，如果高，通常是因为 IO 太忙了。

9.5.3.2.10 TiKV_raft_apply_log_duration_secs

- 报警规则：

```
histogram_quantile(0.99, sum(rate(tikv_raftstore_apply_log_duration_seconds_bucket[1m]))by (
↪ le, instance))> 1
```

- 规则描述：

表示 apply Raft log 耗时，如果高，通常是因为 IO 太忙了。

9.5.3.2.11 TiKV_scheduler_latch_wait_duration_seconds

- 报警规则：

```
histogram_quantile(0.99, sum(rate(tikv_scheduler_latch_wait_duration_seconds_bucket[1m]))by
↪ (le, instance, type))> 1
```

- 规则描述：

Scheduler 中写操作获取内存锁时的等待时间。如果这个值高，表明写操作冲突较多，也可能是某些引起冲突的操作耗时较长，阻塞了其它等待相同锁的操作。

- 处理方法：

1. 查看 Scheduler-All 监控中的 scheduler command duration，看哪一个命令耗时最大。
2. 查看 Scheduler-All 监控中的 scheduler scan details，看 `total` 和 `process` 是否匹配。如果相差太大，表明有很多无效的扫描，另外观察是否有 `over seek bound`，如果太多，表明 GC 不及时。
3. 查看 Storage 监控中的 storage async snapshot/write duration，看是否 Raft 操作不及时。

9.5.3.2.12 TiKV_thread_apply_worker_cpu_seconds

- 报警规则：

```
max(rate(tikv_thread_cpu_seconds_total{name=~"apply_.*"}[1m]))by (instance)> 0.9
```

- 规则描述：

Apply Raft log 线程压力太大，已经接近或超过 apply 线程的处理上限。通常是因为短期内写入的数据量太多造成的。

9.5.3.3 警告级别报警项

9.5.3.3.1 TiKV_leader_drops

- 报警规则：

```
delta(tikv_pd_heartbeat_tick_total{type="leader"}[30s])< -10
```

- 规则描述：

该问题通常是因为 Raftstore 线程卡住了。

- 处理方法：

1. 参考TiKV_channel_full_total 的处理方法。
2. 如果 TiKV 压力很小，考虑 PD 的调度是否太频繁。可以查看 PD 页面的 Operator Create 面板，排查 PD 产生调度的类型和数量。

9.5.3.3.2 TiKV_raft_process_ready_duration_secs

- 报警规则：

```
histogram_quantile(0.999, sum(rate(tikv_raftstore_raft_process_duration_secs_bucket{type='
↪ ready'}[1m]))by (le, instance, type))> 2
```

- 规则描述：

表示处理 Raft ready 的耗时。这个值大，通常是因为 append log 任务卡住了。

9.5.3.3.3 TiKV_raft_process_tick_duration_secs

- 报警规则：

```
histogram_quantile(0.999, sum(rate(tikv_raftstore_raft_process_duration_secs_bucket{type='
↪ tick' }[1m]))by (le, instance, type))> 2
```

- 规则描述：

表示处理 Raft tick 的耗时，这个值大，通常是因为 Region 太多导致的。

- 处理方法：

1. 考虑使用更高等级的日志，比如 warn 或者 error。
2. 在 [raftstore] 配置下添加 raft-base-tick-interval = “2s”。

9.5.3.3.4 TiKV_scheduler_context_total

- 报警规则：

  ```
  abs(delta( tikv_scheduler_context_total[5m]))> 1000
  ```

- 规则描述：

  Scheduler 正在执行的写命令数量。这个值高，表示任务完成得不及时。

- 处理方法：

  参考TiKV_scheduler_latch_wait_duration_seconds 的处理方法。

9.5.3.3.5 TiKV_scheduler_command_duration_seconds

- 报警规则：

  ```
  histogram_quantile(0.99, sum(rate(tikv_scheduler_command_duration_seconds_bucket[1m]))by (le
  ↪ , instance, type))> 1
  ```

- 规则描述：

  表明 Scheduler 执行命令的耗时。

- 处理方法：

  参考TiKV_scheduler_latch_wait_duration_seconds 的处理方法。

9.5.3.3.6 TiKV_coprocessor_outdated_request_wait_seconds

- 报警规则：

  ```
  delta(tikv_coprocessor_outdated_request_wait_seconds_count[10m])> 0
  ```

- 规则描述：

  Coprocessor 已经过期的请求等待的时间。这个值高，表示 Coprocessor 压力已经非常大了。

- 处理方法：

  参考TiKV_coprocessor_request_wait_seconds 的处理方法。

9.5.3.3.7 TiKV_coprocessor_pending_request

- 报警规则：

  ```
  delta(tikv_coprocessor_pending_request[10m])> 5000
  ```

- 规则描述：

  Coprocessor 排队的请求。

- 处理方法：

  参考TiKV_coprocessor_request_wait_seconds 的处理方法。

9.5.3.3.8 TiKV_batch_request_snapshot_nums

- 报警规则：

```
sum(rate(tikv_thread_cpu_seconds_total{name=~"cop_.*"}[1m]))by (instance)/ (count(tikv_thread_cpu_seconds_t
↪ {name=~"cop_.*"})* 0.9)/ count(count(tikv_thread_cpu_seconds_total)by (instance))> 0
```

- 规则描述：

  某个 TiKV 的 Coprocessor CPU 使用率超过了 90%。

9.5.3.3.9 TiKV_pending_task

- 报警规则：

```
sum(tikv_worker_pending_task_total)BY (instance,name)> 1000
```

- 规则描述：

  TiKV 等待的任务数量。

- 处理方法：

  观察TiKV-Details > Task 面板，查看是哪一类任务的 `Worker pending tasks` 值偏高。

9.5.3.3.10 TiKV_low_space

- 报警规则：

```
sum(tikv_store_size_bytes{type="available"})by (instance)/ sum(tikv_store_size_bytes{type="
↪ capacity"})by (instance)< 0.2
```

- 规则描述：

  TiKV 数据量超过节点配置容量或物理磁盘容量的 80%。

- 处理方法：

  确认节点空间均衡情况，做好扩容计划。

9.5.3.3.11 TiKV_approximate_region_size

- 报警规则：

```
histogram_quantile(0.99, sum(rate(tikv_raftstore_region_size_bucket[1m]))by (le))> 1073741824
↪
```

- 规则描述：

  TiKV split checker 扫描到的最大的 Region approximate size 在 1 分钟内持续大于 1 GB。

- 处理方法：

  Region 分裂的速度不及写入的速度。为缓解这种情况，建议更新到支持 batch-split 的版本 (>= 2.1.0-rc1)。如暂时无法更新，可以使用 `pd-ctl operator add split-region <region_id> --policy=approximate` 手动分裂 Region。

9.5.4 TiFlash 报警规则

关于 TiFlash 报警规则的详细描述，参见TiFlash 报警规则。

9.5.5 TiDB Binlog 报警规则

关于 TiDB Binlog 报警规则的详细描述，参见TiDB Binlog 集群监控报警文档。

9.5.6 TiCDC 报警规则

关于 TiCDC 报警规则的详细描述，参见TiCDC 集群监控报警。

9.5.7 Node_exporter 主机报警规则

本节介绍了 Node_exporter 主机的报警项。

9.5.7.1 紧急级别报警项

9.5.7.1.1 NODE_disk_used_more_than_80%

- 报警规则：

```
node_filesystem_avail_bytes{fstype=~"(ext.|xfs)", mountpoint!~"/boot"} / node_filesystem_size_bytes
↪ {fstype=~"(ext.|xfs)", mountpoint!~"/boot"} * 100 <= 20
```

- 规则描述：

  机器磁盘空间使用率超过 80%。

- 处理方法：

  登录机器，执行 `df -h` 命令，查看磁盘空间使用率，做好扩容计划。

9.5.7.1.2 NODE_disk_inode_more_than_80%

- 报警规则：

```
node_filesystem_files_free{fstype=~"(ext.|xfs)"} / node_filesystem_files{fstype=~"(ext.|xfs)
↪ "} * 100 < 20
```

- 规则描述：

  机器磁盘挂载目录文件系统 inode 使用率超过 80%。

- 处理方法：

  登录机器，执行 `df -i` 命令，查看磁盘挂载目录文件系统 inode 使用率，做好扩容计划。

9.5.7.1.3 NODE_disk_readonly

- 报警规则：

```
node_filesystem_readonly{fstype=~"(ext.|xfs)"} == 1
```

- 规则描述：

  磁盘挂载目录文件系统只读，无法写入数据，一般是因为磁盘故障或文件系统损坏。

- 处理方法：

  - 登录机器创建文件测试是否正常。
  - 检查该服务器硬盘指示灯是否正常，如异常，需更换磁盘并修复该机器文件系统。

9.5.7.2 严重级别报警项

9.5.7.2.1 NODE_memory_used_more_than_80%

- 报警规则：

```
(((node_memory_MemTotal_bytes-node_memory_MemFree_bytes-node_memory_Cached_bytes)/(node_memory_MemTotal_byt
↪ )*100))>= 80
```

- 规则描述：

  机器内存使用率超过 80%。

- 处理方法：

  - 在 Grafana Node Exporter 页面查看该主机的 Memory 面板，检查 Used 是否过高，Available 内存是否过低。
  - 登录机器，执行 `free -m` 命令查看内存使用情况，执行 top 看是否有异常进程的内存使用率过高。

9.5.7.3 警告级别报警项

9.5.7.3.1 NODE_node_overload

- 报警规则：

```
(node_load5 / count without (cpu, mode)(node_cpu_seconds_total{mode="system"}))> 1
```

- 规则描述：

  机器 CPU 负载较高。

- 处理方法：

  - 在 Grafana Node exporter 页面上查看该主机的 CPU Usage 及 Load Average，检查是否过高。
  - 登录机器，执行 top 查看 load average 及 CPU 使用率，看是否是异常进程的 CPU 使用率过高。

9.5.7.3.2 NODE_cpu_used_more_than_80%

- 报警规则：

  ```
  avg(irate(node_cpu_seconds_total{mode="idle"}[5m]))by(instance)* 100 <= 20
  ```

- 规则描述：

  机器 CPU 使用率超过 80%。

- 处理方法：

  - 在 Grafana Node exporter 页面上查看该主机的 CPU Usage 及 Load Average，检查是否过高。
  - 登录机器，执行 top 查看 load average 及 CPU 使用率，看是否是异常进程的 CPU 使用率过高。

9.5.7.3.3 NODE_tcp_estab_num_more_than_50000

- 报警规则：

  ```
  node_netstat_Tcp_CurrEstab > 50000
  ```

- 规则描述：

  机器 establish 状态的 TCP 链接超过 50,000。

- 处理方法：

  登录机器执行 ss -s 可查看当前系统 estab 状态的 TCP 链接数，执行 netstat 查看是否有异常链接。

9.5.7.3.4 NODE_disk_read_latency_more_than_32ms

- 报警规则：

  ```
  ((rate(node_disk_read_time_seconds_total{device=~".+"}[5m])/ rate(node_disk_reads_completed_total
  ↪ {device=~".+"}[5m]))or (irate(node_disk_read_time_seconds_total{device=~".+"}[5m])/ irate
  ↪ (node_disk_reads_completed_total{device=~".+"}[5m])))* 1000 > 32
  ```

- 规则描述：

  磁盘读延迟超过 32 毫秒。

- 处理方法：

  - 查看 Grafana Disk Performance Dashboard 观察磁盘使用情况。
  - 查看 Disk Latency 面板观察磁盘的读延迟。
  - 查看 Disk IO Utilization 面板观察 IO 使用率。

9.5.7.3.5 NODE_disk_write_latency_more_than_16ms

- 报警规则：

```
((rate(node_disk_write_time_seconds_total{device=~".+"}[5m])/ rate(node_disk_writes_completed_total
↪ {device=~".+"}[5m]))or (irate(node_disk_write_time_seconds_total{device=~".+"}[5m])/
↪ irate(node_disk_writes_completed_total{device=~".+"}[5m])))> 16
```

- 规则描述：

  机器磁盘写延迟超过 16 毫秒。

- 处理方法：

  - 查看 Grafana Disk Performance Dashboard 观察磁盘使用情况。
  - 查看 Disk Latency 面板可查看磁盘的写延迟。
  - 查看 Disk IO Utilization 面板可查看 IO 使用率。

9.5.8 Blackbox_exporter TCP、ICMP 和 HTTP 报警规则

本节介绍了 Blackbox_exporter TCP、ICMP 和 HTTP 的报警项。

9.5.8.1 紧急级别报警项

9.5.8.1.1 TiDB_server_is_down

- 报警规则：

```
probe_success{group="tidb"} == 0
```

- 规则描述：

  TiDB 服务端口探测失败。

- 处理方法：

  - 检查 TiDB 服务所在机器是否宕机。
  - 检查 TiDB 进程是否存在。
  - 检查监控机与 TiDB 服务所在机器之间网络是否正常。

9.5.8.1.2 TiFlash_server_is_down

- 报警规则：

```
probe_success{group="tiflash"} == 0
```

- 规则描述：

  TiFlash 服务端口探测失败。

- 处理方法：

  - 检查 TiFlash 服务所在机器是否宕机。
  - 检查 TiFlash 进程是否存在。
  - 检查监控机与 TiFlash 服务所在机器之间网络是否正常。

9.5.8.1.3 Pump_server_is_down

- 报警规则：

  ```
  probe_success{group="pump"} == 0
  ```

- 规则描述：

  Pump 服务端口探测失败。

- 处理方法：

  - 检查 Pump 服务所在机器是否宕机。
  - 检查 Pump 进程是否存在。
  - 检查监控机与 Pump 服务所在机器之间网络是否正常。

9.5.8.1.4 Drainer_server_is_down

- 报警规则：

  ```
  probe_success{group="drainer"} == 0
  ```

- 规则描述：

  Drainer 服务端口探测失败。

- 处理方法：

  - 检查 Drainer 服务所在机器是否宕机。
  - 检查 Drainer 进程是否存在。
  - 检查监控机与 Drainer 服务所在机器之间网络是否正常。

9.5.8.1.5 TiKV_server_is_down

- 报警规则：

  ```
  probe_success{group="tikv"} == 0
  ```

- 规则描述：

  TiKV 服务端口探测失败。

- 处理方法：

  - 检查 TiKV 服务所在机器是否宕机。
  - 检查 TiKV 进程是否存在。
  - 检查监控机与 TiKV 服务所在机器之间网络是否正常。

9.5.8.1.6 PD_server_is_down

- 报警规则：

  ```
  probe_success{group="pd"} == 0
  ```

- 规则描述：

  PD 服务端口探测失败。

- 处理方法：

  - 检查 PD 服务所在机器是否宕机。
  - 检查 PD 进程是否存在。
  - 检查监控机与 PD 服务所在机器之间网络是否正常。

9.5.8.1.7 Node_exporter_server_is_down

- 报警规则：

  ```
  probe_success{group="node_exporter"} == 0
  ```

- 规则描述：

  Node_exporter 服务端口探测失败。

- 处理方法：

  - 检查 Node_exporter 服务所在机器是否宕机。
  - 检查 Node_exporter 进程是否存在。
  - 检查监控机与 Node_exporter 服务所在机器之间网络是否正常。

9.5.8.1.8 Blackbox_exporter_server_is_down

- 报警规则：

  ```
  probe_success{group="blackbox_exporter"} == 0
  ```

- 规则描述：

  Blackbox_exporter 服务端口探测失败。

- 处理方法：

  - 检查 Blackbox_exporter 服务所在机器是否宕机。
  - 检查 Blackbox_exporter 进程是否存在。
  - 检查监控机与 Blackbox_exporter 服务所在机器之间网络是否正常。

9.5.8.1.9 Grafana_server_is_down

- 报警规则：

  ```
  probe_success{group="grafana"} == 0
  ```

- 规则描述：

  Grafana 服务端口探测失败。

- 处理方法：

  - 检查 Grafana 服务所在机器是否宕机。
  - 检查 Grafana 进程是否存在。
  - 检查监控机与 Grafana 服务所在机器之间网络是否正常。

9.5.8.1.10 Pushgateway_server_is_down

- 报警规则：

  ```
  probe_success{group="pushgateway"} == 0
  ```

- 规则描述：

  Pushgateway 服务端口探测失败。

- 处理方法：

  - 检查 Pushgateway 服务所在机器是否宕机。
  - 检查 Pushgateway 进程是否存在。
  - 检查监控机与 Pushgateway 服务所在机器之间网络是否正常。

9.5.8.1.11 Kafka_exporter_is_down

- 报警规则：

  ```
  probe_success{group="kafka_exporter"} == 0
  ```

- 规则描述：

  Kafka_exporter 服务端口探测失败。

- 处理方法：

  - 检查 Kafka_exporter 服务所在机器是否宕机。
  - 检查 Kafka_exporter 进程是否存在。
  - 检查监控机与 Kafka_exporter 服务所在机器之间网络是否正常。

9.5.8.1.12 Pushgateway_metrics_interface

- 报警规则：

  ```
  probe_success{job="blackbox_exporter_http"} == 0
  ```

- 规则描述：

  Pushgateway 服务 http 接口探测失败。

- 处理方法：

  - 检查 Pushgateway 服务所在机器是否宕机。
  - 检查 Pushgateway 进程是否存在。
  - 检查监控机与 Pushgateway 服务所在机器之间网络是否正常。

9.5.8.2 警告级别报警项

9.5.8.2.1 BLACKER_ping_latency_more_than_1s

- 报警规则：

  ```
  max_over_time(probe_duration_seconds{job=~"blackbox_exporter.*_icmp"}[1m])> 1
  ```

- 规则描述：

  Ping 延迟超过 1 秒。

- 处理方法：

  - 在 Grafana Blackbox Exporter 页面上检查两个节点间的 ping 延迟是否太高。
  - 在 Grafana Node Exporter 页面的 TCP 面板上检查是否有丢包。

9.6 TiFlash 报警规则

本文介绍了 TiFlash 集群的报警规则。

9.6.1 TiFlash_schema_error

- 报警规则：

  ```
  increase(tiflash_schema_apply_count{type="failed"}[15m])> 0
  ```

- 规则描述：

  出现 schema apply 错误时报警。

- 处理方法：

  可能是逻辑问题，联系 TiFlash 开发人员。

9.6.2 TiFlash_schema_apply_duration

- 报警规则：

```
histogram_quantile(0.99, sum(rate(tiflash_schema_apply_duration_seconds_bucket[1m]))BY (le,
↪ instance))> 20
```

- 规则描述：

apply 时间超过 20 秒的概率超过 99% 时报警。

- 处理方法：

可能是 TiFlash 存储引擎内部问题，联系 TiFlash 开发人员。

9.6.3 TiFlash_raft_read_index_duration

- 报警规则：

```
histogram_quantile(0.99, sum(rate(tiflash_raft_read_index_duration_seconds_bucket[1m]))BY (
↪ le, instance))> 3
```

- 规则描述：

read index 时间超过 3 秒的概率超过 99% 时报警。

> 注意：
>
> read index 请求是发送给 TiKV leader 的 kvproto 请求，TiKV region 的重试，或 Store 的繁忙/网络问题都可能导致 read index 请求时间过长。

- 处理方法：

可能 TiKV 集群分裂/迁移频繁，导致频繁重试，可以查看 TiKV 集群状态确认。

9.6.4 TiFlash_raft_wait_index_duration

- 报警规则：

```
histogram_quantile(0.99, sum(rate(tiflash_raft_wait_index_duration_seconds_bucket[1m]))BY (
↪ le, instance))> 2
```

- 规则描述：

TiFlash 等待 Region Raft Index 的时间超过 2 秒的概率超过 99% 时报警。

- 处理方法：

可能 TiKV 和 Proxy 的通信出现问题，联系 TiFlash 开发人员确认。

9.7 自定义监控组件的配置

使用 TiUP 部署 TiDB 集群时，TiUP 会同时自动部署 Prometheus、 Grafana 和 Alertmanager 等监控组件，并且在集群扩容中自动为新增节点添加监控配置。

需要注意的是，TiUP 会使用自己的配置参数覆盖监控组件的配置，如果你直接修改监控组件的配置文件，修改的配置文件可能在对集群进行 deploy/scale-out/scale-in/reload 等操作中被 TiUP 所覆盖，导致配置不生效。

如果需要自定义 Prometheus、 Grafana 和 Alertmanager 等监控组件的配置，请参考本文在 TiDB 集群的拓扑配置 topology.yaml 文件中添加对应的配置项。

> 注意：
>
> - 在自定义监控组件的配置时，请勿直接修改监控组件的配置文件。因为在对集群进行 deploy/scale-out/scale-in/reload 等操作时，TiUP 会使用自己的配置参数覆盖监控组件的配置。
> - 如果监控组件不是由 TiUP 部署和管理，可以直接修改监控组件的配置文件，无需参考本文档。
> - 本文所述功能在 TiUP v1.9.0 及后续版本支持，使用本功能前请检查 TiUP 版本号。

9.7.1 自定义 Prometheus 配置

目前，TiUP 支持自定义 Prometheus 的 rule 配置和 scrape 配置。

9.7.1.1 自定义 Prometheus rule

1. 将自定义的 rule 配置文件放到 TiUP 所在机器的某个目录下。
2. 在 TiDB 集群的拓扑配置 topology.yaml 文件中，将自定义规则文件目录 rule_dir 设置为实际 rule 配置文件的目录。

以下为 topology.yaml 文件中的 monitoring_servers 配置示例:

```
## # Server configs are used to specify the configuration of Prometheus Server.
monitoring_servers:
  # # The ip address of the Monitoring Server.
  - host: 127.0.0.1
    rule_dir: /home/tidb/prometheus_rule   # prometheus rule dir on TiUP machine
```

上述配置后，在集群进行 deploy/scale-out/scale-in/reload 操作时，TiUP 将读取本机 /home/tidb/prometheus_rule 路径下的自定义 rule，然后将该配置发送到 Prometheus Server，替换默认配置规则。

9.7.1.2 自定义 Prometheus scrape 配置

1. 打开 TiDB 集群的拓扑配置文件 topology.yaml。
2. 在 monitoring_servers 的配置部分添加 additional_scrape_conf 字段。

以下为 topology.yaml 文件中的 monitoring_servers 配置示例:

```
monitoring_servers:
- host: xxxxxxx
  ssh_port: 22
  port: 9090
  deploy_dir: /tidb-deploy/prometheus-9090
  data_dir: /tidb-data/prometheus-9090
  log_dir: /tidb-deploy/prometheus-9090/log
  external_alertmanagers: []
  arch: amd64
  os: linux
  additional_scrape_conf:
    metric_relabel_configs:
      - source_labels: [__name__]
        separator: ;
        regex: tikv_thread_nonvoluntary_context_switches|tikv_thread_voluntary_context_switches|
            ↪ tikv_threads_io_bytes_total
        action: drop
      - source_labels: [__name__,name]
        separator: ;
        regex: tikv_thread_cpu_seconds_total;(tokio|rocksdb).+
        action: drop
```

上述配置后，在集群进行 deploy/scale-out/scale-in/reload 操作时，TiUP 会将 additional_scrape_conf 字段的内容会添加到 Prometheus 配置文件的对应参数中。

9.7.2 自定义 Grafana 配置

目前，TiUP 支持自定义 Grafana Dashboard 和其他配置。

9.7.2.1 自定义 Grafana Dashboard

1. 将自定义的 Dashboard 配置文件放到 TiUP 所在机器的某个目录下。
2. 在 TiDB 集群的拓扑配置 topology.yaml 文件中，将自定义规则文件目录 dashboard_dir 设置为实际放置 Dashboard 配置文件的目录。

以下为 topology.yaml 文件中的 grafana_servers 配置示例：

```
## # Server configs are used to specify the configuration of Grafana Servers.
grafana_servers:
  # # The ip address of the Grafana Server.
  - host: 127.0.0.1
    dashboard_dir: /home/tidb/dashboards   # grafana dashboard dir on TiUP machine
```

上述配置后，在集群进行 deploy/scale-out/scale-in/reload 操作时，TiUP 将读取本机 /home/tidb/dashboards 路径下的自定义 Dashboard，然后将该配置发送到 Grafana Server，替换默认配置规则。

9.7.2.2 自定义 Grafana 其他配置

1. 打开集群配置文件 topology.yaml。
2. 在 grafana_servers 的配置部分添加其他配置。

以下为 topology.yaml 文件中的log.file level 字段以及 smtp 配置示例：

```
## # Server configs are used to specify the configuration of Grafana Servers.
grafana_servers:
  # # The ip address of the Grafana Server.
  - host: 127.0.0.1
    config:
      log.file.level: warning
      smtp.enabled: true
      smtp.host: {IP}:{port}
      smtp.user: example@pingcap.com
      smtp.password: {password}
      smtp.skip_verify: true
```

上述配置后，在集群进行 deploy/scale-out/scale-in/reload 操作时，TiUP 会将 config 字段的内容会添加到 grafana 的配置文件 grafana.ini 中。

9.7.3 自定义 Alertmanager 配置

目前，TiUP 支持自定义配置 Alertmanager 的监听地址。

TiUP 部署的 Alertmanager 默认监听 alertmanager_servers.host，如果你使用代理，则无法访问 Alertmanager。此时，你可以在集群配置文件 topology.yaml 中添加 listen_host 指定监听地址，使得 Alertmanager 可以通过代理访问。推荐配置为 0.0.0.0。

以下示例将 listen_host 字段设置为 0.0.0.0。

```
alertmanager_servers:
  # # The ip address of the Alertmanager Server.
  - host: 172.16.7.147
    listen_host: 0.0.0.0
```

```
# # SSH port of the server.
ssh_port: 22
```

上述配置后，在集群进行 deploy/scale-out/scale-in/reload 操作时，TiUP 会将 listen_host 字段的内容会添加到 Alertmanager 启动参数的 ‘–web.listen-address’ 中。

9.8 备份恢复监控告警

本文介绍备份恢复的监控和告警，包括如何部署监控、监控指标及常用告警项。

9.8.1 日志备份监控

日志备份支持功能使用 Prometheus 采集监控指标，目前所有的监控指标都内置在 TiKV 中。

9.8.1.1 部署监控

- 通过 TiUP 部署的集群，Prometheus 会自动采集相关的监控指标。
- 手动部署的集群，需要参考TiDB 集群监控部署，在 Prometheus 配置文件的 `scrape_configs` 中加入 TiKV 相关的 job。

9.8.1.2 配置 Grafana

- 通过 TiUP 部署的集群，Grafana 中内置了 Backup log 的面板。
- 手动部署的集群，需要参考导入 Grafana 面板，将 tikv_details.json 文件上传到 Grafana 中。之后在 TiKV-Details Dashboard 中找到 Backup Log 面板即可。

9.8.1.3 监控指标

指标	类型	说明
tikv_log_backup_interal_actor_acting_duration_sec	Histogram	处理内部各种消息事件的耗时。`message :: TaskType`
tikv_log_backup_initial_scan_reason	Counter	触发增量扫的原因统计。主要是 Leader 迁移或者 Region Version 变更。`reason :: {"leader-changed", "region-changed", "retry"}`
tikv_log_backup_event_handle_duration_sec	Histogram	处理 KV Event 的耗时。和 `tikv_log_backup_on_event_duration_seconds` 相比，这个指标还包含了一些内部转化消耗的时间。`stage :: {"to_stream_event", "save_to_temp_file"}`
tikv_log_backup_handle_kv_batch	Histogram	由 RaftStore 发送的 KV 对的 Batch 大小统计，统计数据为 Region 级别。
tikv_log_backup_initial_scan_disk_read	Counter	增量扫期间，从硬盘读取的数据量的大小。在 Linux 系统下，这个信息来自于 procfs，是实际从 block device 读取的数据量的大小；配置项 `initial-scan-rate-limit` 也是施加于这个数值上。

指标	类型	说明
tikv_log_backup_incremental_scan_bytes	Histogram	增量扫期间，实际产生的 KV 对的大小。因为压缩和读放大的缘故，这个数值和 `tikv_log_backup_initial_scan_disk_read` 不一定相同。
tikv_log_backup_skip_kv_count	Counter	日志备份期间，因为对备份没有帮助而被跳过的 Raft Event 数量。
tikv_log_backup_errors	Counter	日志备份期间，遇到的可以重试或可以忽略的错误。`type :: ErrorType`
tikv_log_backup_fatal_errors	Counter	日志备份期间，遇到的不可重试或不可忽略的错误。当该类错误出现的时候，日志备份任务会被暂停。`type :: ErrorType`
tikv_log_backup_heap_memory	Gauge	日志备份期间，增量扫发现的、尚未被消费的事件占用的内存。
tikv_log_backup_on_event_duration_seconds	Histogram	将 KV Event 保存到临时文件各个阶段的耗时。`stage :: {"write_to_tempfile", "syscall_write"}`
tikv_log_backup_store_checkpoint_ts	Gauge	Store 级别的 Checkpoint TS，已经弃用。其含义更加接近于 Store 当前注册的 GC Safepoint。`task :: string`
tidb_log_backup_last_checkpoint	Gauge	全局 Checkpoint TS，表示日志备份功能中已经备份的时间点。`task :: string`
tikv_log_backup_flush_duration_sec	Histogram	将本地临时文件移动到外部存储的耗时。`stage :: {"generate_metadata", "save_files", "clear_temp_files"}`
tikv_log_backup_flush_file_size	Histogram	备份产生的文件的大小统计。
tikv_log_backup_initial_scan_duration_sec	Histogram	增量扫的整体耗时统计。
tikv_log_backup_skip_retry_observe	Counter	在日志备份过程中，遇到的可忽略错误的统计，即放弃 retry 的原因。`reason :: {"region-absent", "not-leader", "stale-command"}`
tikv_log_backup_initial_scan_operations	Counter	增量扫过程中，RocksDB 相关的操作统计。`cf :: {"default", "write", "lock"}, op :: RocksDBOP`
tikv_log_backup_enabled	Counter	日志备份功能是否开启，若值大于 0，表示开启
tikv_log_backup_observed_region	Gauge	被监听的 Region 数量
tikv_log_backup_task_status	Gauge	日志备份任务状态，0-Running 1-Paused 2-Error `task :: string`
tikv_log_backup_pending_initial_scan	Gauge	尚未执行的增量扫的统计。`stage :: {"queuing", "executing"}`

9.8.1.4 日志备份告警

9.8.1.4.1 配置告警

目前 Point-in-time recovery (PITR) 还未内置告警项，本节介绍如何在 PITR 中配置告警项，以及推荐的告警项规则。

告警规则配置可以参考下面的步骤：

1. 在 Prometheus 所在节点上创建告警规则的配置文件（例如 `pitr.rules.yml`），参考 Prometheus 文档 和下列推荐的告警项及配置样例填写告警规则。
2. 在 Prometheus 配置文件中的 `rule_files` 字段填入告警规则文件的路径。
3. 通过向 Prometheus 进程发送 SIGHUP 信号（`kill -HUP pid`）或向 `http://prometheus-addr/-/reload` 发送 HTTP POST 请求（使用 HTTP 请求方式前需要在启动 Prometheus 时指定 `--web.enable-lifecycle` 参数）。

以下为推荐的告警项配置：

9.8.1.4.2 LogBackupRunningRPOMoreThan10m

- 表达式：`max(time()- tidb_log_backup_last_checkpoint / 262144000)by (task)/ 60 > 10 and max( ↪ tidb_log_backup_last_checkpoint)by (task)> 0 and max(tikv_log_backup_task_status)by (task ↪ )== 0`
- 告警级别：warning
- 说明：日志数据超过 10 分钟未持久化到存储中，该配置项主要用于提醒，大部分情况下，不会影响日志备份。

Prometheus 中的配置样例如下：

```
groups:
- name: PiTR
  rules:
  - alert: LogBackupRunningRPOMoreThan10m
    expr: max(time() - tidb_log_backup_last_checkpoint / 262144000) by (task) / 60 > 10 and max(
        ↪ tidb_log_backup_last_checkpoint) by (task) > 0 and max(tikv_log_backup_task_status)
        ↪ by (task) == 0
    labels:
      severity: warning
    annotations:
      summary: RPO of log backup is high
      message: RPO of the log backup task {{ $labels.task }} is more than 10m
```

9.8.1.4.3 LogBackupRunningRPOMoreThan30m

- 表达式：`max(time()- tidb_log_backup_last_checkpoint / 262144000)by (task)/ 60 > 30 and max( ↪ tidb_log_backup_last_checkpoint)by (task)> 0 and max(tikv_log_backup_task_status)by (task ↪ )== 0`
- 告警级别：critical
- 说明：日志数据超过 30 分钟未持久化到存储中，出现该告警表示极有可能出现异常，可以查看 TiKV 日志定位原因。

9.8.1.4.4 LogBackupPausingMoreThan2h

- 表达式：`max(time()- tidb_log_backup_last_checkpoint / 262144000)by (task)/ 3600 > 2 and max( ↪ tidb_log_backup_last_checkpoint)by (task)> 0 and max(tikv_log_backup_task_status)by (task ↪ )== 1`
- 告警级别：warning
- 说明：日志备份任务处于暂停状态超过 2 小时，该告警主要用于提醒，建议尽早执行 `br log resume` 恢复任务。

9.8.1.4.5 LogBackupPausingMoreThan12h

- 表 达 式：`max(time()- tidb_log_backup_last_checkpoint / 262144000)by (task)/ 3600 > 12 and max(tidb_log_backup_last_checkpoint)by (task)> 0 and max(tikv_log_backup_task_status)by (task)== 1`
- 告警级别：critical
- 说明：日志备份任务处于暂停状态超过 12 小时，应尽快执行 `br log resume` 恢复任务。任务处于暂停状态时间过长会有数据丢失的风险。

9.8.1.4.6 LogBackupFailed

- 表达式：`max(tikv_log_backup_task_status)by (task)== 2 and max(tidb_log_backup_last_checkpoint)by (task)> 0`
- 告警级别：critical
- 说明：日志备份任务进入失败状态，需要执行 `br log status` 查看失败原因，如有必要还需进一步查看 TiKV 日志。

9.8.1.4.7 LogBackupGCSafePointExceedsCheckpoint

- 表达式：`min(tidb_log_backup_last_checkpoint)by (instance)- max(tikv_gcworker_autogc_safe_point)by (instance)< 0`
- 告警级别：critical
- 说明：部分数据在备份前被 GC，此时已有部分数据丢失，极有可能对业务产生影响。

10 故障诊断

10.1 故障诊断问题汇总

10.1.1 TiDB 集群问题导图

本篇文档总结了使用 TiDB 及其组件时的常见错误。遇到相关错误时，可以通过本文档的问题导图来排查错误原因并进行处理。

10.1.1.1 1. 服务不可用

10.1.1.1.1 1.1 客户端报 `Region is Unavailable` 错误

- 1.1.1 `Region is Unavailable` 一般是由于 Region 在一段时间不可用（可能会遇到 `TiKV server is busy`；或者发送给 TiKV 的请求由于 `not leader` 或者 `epoch not match` 等原因被打回；又或者请求 TiKV 超时等），TiDB 内部会进行 `backoff` 重试。`backoff` 的时间超过一定阈值（默认 20s）后就会报错给客户端。如果 `backoff` 在阈值内，客户端对该错误无感知。
- 1.1.2 多台 TiKV 同时内存不足 (OOM)，导致 Region OOM 期间内没有 Leader，见案例 case-991。

- 1.1.3 TiKV 报 `TiKV server is busy` 错误，超过 backoff 时间，参考4.3 客户端报 `server is busy` 错误。`TiKV server is busy` 属于内部流控机制，后续可能不计入 backoff 时间。
- 1.1.4 多台 TiKV 启动不了，导致 Region 没有 Leader。单台物理主机部署多个 TiKV 实例，一个物理机挂掉，由于 label 配置错误导致 Region 没有 Leader，见案例 case-228。
- 1.1.5 follower apply 落后，成为 Leader 之后把收到的请求以 `epoch not match` 理由打回，见案例 case-958（TiKV 内部需要优化该机制）。

10.1.1.1.2 1.2 PD 异常导致服务不可用

查看本文档5. PD 问题。

10.1.1.2 2. 延迟明显升高

10.1.1.2.1 2.1 延迟短暂升高

- 2.1.1 TiDB 执行计划不对导致延迟升高，请参考3.3 执行计划不对。
- 2.1.2 PD 出现选举问题或者 OOM 问题，请参考5.2 PD 选举问题和5.3 PD OOM 问题。
- 2.1.3 某些 TiKV 大量掉 Leader，请参考4.4 某些 TiKV 大量掉 Leader。
- 2.1.4 其他原因，请参考读写延迟增加。

10.1.1.2.2 2.2 Latency 持续升高

- 2.2.1 TiKV 单线程瓶颈
  - 单个 TiKV Region 过多，导致单个 gRPC 线程成为瓶颈（查看监控：Grafana -> TiKV-details -> Thread CPU/gRPC CPU Per Thread），v3.x 以上版本可以开启 Hibernate Region 特性解决该问题，见案例 case-612。
  - v3.0 之前版本 Raftstore 单线程或者 apply 单线程到达瓶颈（查看监控：Grafana -> TiKV-details -> Thread CPU/raft store CPU 和 Async apply CPU 超过 80%）。可以选择扩容 TiKV（v2.x 版本）实例，或者升级到多线程模型的 v3.x 版本。
- 2.2.2 CPU load 升高。
- 2.2.3 TiKV 写入慢，请参考4.5 TiKV 写入慢。
- 2.2.4 TiDB 执行计划不对，请参考3.3 执行计划不对。
- 2.2.5 其他原因，请参考读写延迟增加。

10.1.1.3 3. TiDB 问题

10.1.1.3.1 3.1 DDL

- 3.1.1 修改 decimal 字段长度时报错 "ERROR 1105 (HY000): unsupported modify decimal column precision"。TiDB 暂时不支持修改 decimal 字段长度。
- 3.1.2 TiDB DDL job 卡住不动/执行很慢（通过 admin show ddl jobs 可以查看 DDL 进度）：
  - 原因 1：与外部组件 (PD/TiKV) 的网络问题。
  - 原因 2：早期版本（v3.0.8 之前）TiDB 内部自身负载很重（高并发下可能产生了很多协程）。
  - 原因 3：早期版本（v2.1.15 & v3.0.0-rc1 之前）PD 实例删除 TiDB key 无效的问题，会导致每次 DDL 变更都需要等 2 个 lease（很慢）。
  - 其他未知原因，请上报 bug。
  - 解决方法：原因 1 需要检查与外部组件的网络问题；原因 2 和 3 已经修复，需要升级到高版本；其他原因，可选择以下兜底方案进行 DDL owner 迁移。
  - DDL owner 迁移方案：
    * 如果与该 TiDB 集群可以网络互通，执行重新进行 owner 选举命令：curl -X POST http://{TiDBIP}:10080/ddl/owner/resign
    * 如果与该 TiDB 集群不可以网络互通，需旁路下线，通过 tidb-ctl 工具，从 PD 集群的 etcd 中直接删除 DDL owner，之后也会重新选举：tidb-ctl etcd delowner [LeaseID] [flags] + ownerKey
- 3.1.3 TiDB 日志中报 information schema is changed 的错误：
  - 报错的详细原因以及解决办法参见触发 Information schema is changed 错误的原因。
  - 背景知识：schema version 的增长数量与每个 DDL 变更操作的 schema state 个数一致，例如 create table 操作会有 1 个版本变更，add column 操作会有 4 个版本变更（详情可以参考 online schema change），所以太多的 column 变更操作会导致 schema version 增长得很快。
- 3.1.4 TiDB 日志中报 information schema is out of date 的错误：
  - 原因 1：执行 DML 的 TiDB 被 graceful kill 后准备退出，且此 DML 对应的事务执行时间超过一个 DDL lease，在事务提交的时候会报此错。
  - 原因 2：TiDB 在执行 DML 时，有一段时间连不上 PD 和 TiKV，导致 TiDB 在一个 DDL Lease（默认 45s）内没有加载新的 schema，或者 TiDB 断开与 PD 之间带 keep alive 设置的连接。
  - 原因 3：TiKV 压力大或网络超时，通过监控 Grafana -> TiDB 和 TiKV 节点的负载情况来确认是否是该原因。
  - 解决方法：第 1 种原因，在 TiDB 启动时手动重试该 DML 即可；第 2 种原因，需要检查 TiDB 实例和 PD 及 TiKV 的网络波动情况；第 3 种原因，需要检查 TiKV 为什么繁忙，参考4. TiKV 问题。

10.1.1.3.2 3.2 OOM 问题

- 3.2.1 现象
  - 客户端：客户端收到 TiDB server 报错 ERROR 2013 (HY000): Lost connection to MySQL server during query

  - 日志：
    * `dmesg -T | grep tidb-server` 结果中有事故发生附近时间点的 OOM-killer 的日志。
    * tidb.log 中可以 grep 到事故发生后附近时间的 `"Welcome to TiDB"` 的日志（即 TiDB server 发生重启）。
    * tidb_stderr.log 中能 grep 到 `fatal error: "runtime: out of memory"` 或 `"cannot allocate memory"`。
    * v2.1.8 及其之前的版本，tidb_stderr.log 中能 grep 到 `fatal error: stack overflow`。
  - 监控：TiDB server 实例所在机器可用内存迅速回升

- 3.2.2 定位造成 OOM 的 SQL（目前所有版本都无法完成精准定位，需要在发现 SQL 后再做进一步分析，确认 OOM 是否的确由该 SQL 造成）：
  - `>= v3.0.0` 的版本，可以在 tidb.log 中 `grep "expensive_query"`，该 log 会记录运行超时、或使用内存超过阈值的 SQL。
  - `< v3.0.0` 的版本，通过 `grep "memory exceeds quota"` 定位运行时内存超限的 SQL。

> 注意：
> 单条 SQL 内存阈值的默认值为 1GB，可通过系统变量`tidb_mem_quota_query` 进行设置。

- 3.2.3 缓解 OOM 问题
  - 通过开启 SWAP 的方式，可以缓解由于大查询使用内存过多而造成的 OOM 问题。但该方法由于存在 I/O 开销，会在内存空间不足时对大查询性能造成一定影响。性能回退程度受剩余内存量、读写盘速度影响。

- 3.2.4 OOM 常见原因
  - SQL 中包含 join，通过 `explain` 查看发现该 join 选用 `HashJoin` 算法且 `inner` 端的表很大。
  - 单条 UPDATE/DELETE 涉及的查询数据量太大，见案例 case-882。
  - SQL 中包含 `Union` 连接的多条子查询，见案例 case-1828。

更多 OOM 的排查方法，请参考TiDB OOM 故障排查。

10.1.1.3.3 3.3 执行计划不对

- 3.3.1 现象
  - SQL 相比于之前的执行时间有较大程度变慢，执行计划突然发生改变。如果慢日志中输出了执行计划，可以直接对比执行计划。
  - SQL 执行时间相比于其他数据库（例如 MySQL）有较大差距。可以对比其他数据库执行计划，例如 `Join Order` 是否不同。
  - 慢日志中 SQL 执行时间 `Scan Keys` 数目较大。

- 3.3.2 排查执行计划问题

- `explain analyze {SQL}` 在执行时间可以接受的情况下，对比 `explain analyze` 结果中 `count` 和 execution info 中 `rows` 的数目差距。如果在 `TableScan/IndexScan` 行上发现比较大的差距，很大可能是统计信息出问题；如果在其他行上发现较大差距，则也有可能是非统计信息问题。
- `select count(*)` 在执行计划中包含 `join` 等情况下，explain analyze 可能耗时过长；此时可以通过对 `TableScan/IndexScan` 上的条件进行 `select count(*)`，并对比 `explain` 结果中的 `row count` 信息，确定是不是统计信息的问题。

- 3.3.3 缓解问题
  - v3.0 及以上版本可以使用 `SQL Bind` 功能固定执行计划。
  - 更新统计信息。在大致确定问题是由统计信息导致的情况下，先dump 统计信息保留现场。如果是由于统计信息过期导致，例如 `show stats_meta` 中 modify count/row count 大于某个值（例如 0.3）或者表中存在时间列的索引情况下，可以先尝试 analyze table 恢复；如果配置了 auto analyze，可以查看系统变量 `tidb_auto_analyze_ratio` 是否过大（例如大于 0.3），以及当前时间是否在 `tidb_auto_analyze_start_time` 和 `tidb_auto_analyze_end_time` 范围内。
  - 其他情况，请上报 bug。

10.1.1.3.4 3.4 SQL 执行报错

- 3.4.1 客户端报 `ERROR 1265(01000)Data Truncated` 错误。原因是 TiDB 内在计算 `Decimal` 类型处理精度的时候，和 MySQL 不兼容。该错误已于 v3.0.10 中修复 (#14438)，具体原因如下：

  在 MySQL 内，如果两个大精度 `Decimal` 做除法运算，超出最大小数精度时 (30)，会只保留 30 位且不报错。TiDB 在计算结果上，也是这样实现的，但是在内部表示 `Decimal` 的结构体内，有一个表示小数精度的字段，还是保留的真实精度。

  比如 `(0.1^30)/ 10`，TiDB 和 MySQL 的结果都为 0，是正确的，因为精度最多 30；但是 TiDB 内表示精度的那个字段，还是 31；

  多次 `Decimal` 除法计算后，虽然结果正确，但是这个精度可能越来越大，最终超过 TiDB 内的另一个阈值 72，此时就会报 `Data Truncated` 的错误；`Decimal` 的乘法计算就不会有这个问题，因为绕过越界，会直接把精度设置为最大精度限制。

  解决方法：可以通过手动加 `Cast(xx as decimal(a, b))` 来绕过这个问题，a 和 b 就是目标的精度。

10.1.1.3.5 3.5 慢查询问题

要定位慢查询，参阅慢查询日志。要处理慢查询，参阅分析慢查询。

10.1.1.3.6 3.6 热点问题

TiDB 作为分布式数据库，内建负载均衡机制，尽可能将业务负载均匀地分布到不同计算或存储节点上，更好地利用上整体系统资源。然而，机制不是万能的，在一些场景下仍会有部分业务负载不能被很好地分散，影响性能，形成单点的过高负载，也称为热点。

TiDB 提供了完整的方案用于排查、解决或规避这类热点。通过均衡负载热点，可以提升整体性能，包括提高 QPS 和降低延迟等。详情参见TiDB 热点问题处理。

10.1.1.3.7 3.7 TiDB 磁盘 I/O 过高

当出现系统响应变慢的时候，如果已经排查了 CPU 的瓶颈、数据事务冲突的瓶颈后，问题仍存在，就需要从 I/O 来入手来辅助判断目前的系统瓶颈点。参考TiDB 磁盘 I/O 过高的处理办法了解如何定位和处理 TiDB 存储 I/O 过高的问题。

10.1.1.3.8 3.8 锁冲突问题

TiDB 支持完整的分布式事务，自 v3.0 版本起，提供乐观事务与悲观事务两种事务模式。要了解如何排查锁相关的问题，以及如何处理乐观和悲观锁冲突的问题，请参考TiDB 锁冲突问题处理。

10.1.1.3.9 3.9 数据索引一致性报错

当执行事务或执行 `ADMIN CHECK [TABLE|INDEX]` 命令时，TiDB 会对数据索引的一致性进行检查。如果检查发现 record key-value 和 index key-value 不一致，即存储行数据的键值对和存储其对应索引的键值对之间不一致（例如多索引或缺索引），TiDB 会报数据索引一致性错误，并在日志文件中打印相关错误日志。

要了解更多数据索引一致性报错信息以及如何绕过检查，请参考数据索引一致性报错。

10.1.1.4 4. TiKV 问题

10.1.1.4.1 4.1 TiKV panic 启动不了

- 4.1.1 `sync-log = false`,机器断电之后出现 `unexpected raft log index: last_index X < applied_index Y` 错误。符合预期，需通过 `tikv-ctl` 工具恢复 Region。
- 4.1.2 虚拟机部署 TiKV，`kill` 虚拟机或物理机断电，出现 `entries[X, Y] is unavailable from storage` 错误。符合预期，虚拟机的 fsync 不可靠，需通过 `tikv-ctl` 工具恢复 Region。
- 4.1.3 其他原因（非预期，需报 bug）。

10.1.1.4.2 4.2 TiKV OOM

- 4.2.1 `block-cache` 配置太大导致 OOM：
  - 在监控 Grafana -> TiKV-details 选中对应的 instance 后，查看 RocksDB 的 `block cache size` 监控来确认是否是该问题。
  - 同时，请检查 `[storage.block-cache] capacity = # "1GB"` 参数是否设置合理，默认情况下 TiKV 的 `block-cache` 设置为机器总内存的 45%；在 container 部署时，需要显式指定该参数，因为 TiKV 获取的是物理机的内存，可能会超出 container 的内存限制。
- 4.2.2 Coprocessor 收到大量大查询，返回的数据量太大，gRPC 的发送速度跟不上 Coprocessor 往外输出数据的速度，导致 OOM：
  - 可以通过检查监控：Grafana -> TiKV-details -> coprocessor overview 的 `response size` 是否超过 `network outbound` 流量来确认是否属于这种情况。
- 4.2.3 其他部分占用太多内存（非预期，需报 bug）。

10.1.1.4.3 4.3 客户端报 server is busy 错误

通过查看监控：Grafana -> TiKV -> errors 确认具体 busy 原因。server is busy 是 TiKV 自身的流控机制，TiKV 通过这种方式告知 tidb/ti-client 当前 TiKV 的压力过大，稍后再尝试。

- 4.3.1 TiKV RocksDB 出现 write stall。一个 TiKV 包含两个 RocksDB 实例，一个用于存储 Raft 日志，位于 data/raft。另一个用于存储真正的数据，位于 data/db。通过 grep "Stalling" RocksDB 日志查看 stall 的具体原因，RocksDB 日志是 LOG 开头的文件，LOG 为当前日志。
  - level0 sst 太多导致 stall，可以添加参数 [rocksdb] max-sub-compactions = 2（或者 3），加快 level0 sst 往下 compact 的速度。该参数的意思是将从 level0 到 level1 的 compaction 任务最多切成 max-sub-compactions 个子任务交给多线程并发执行，见案例 case-815。
  - pending compaction bytes 太多导致 stall，磁盘 I/O 能力在业务高峰跟不上写入，可以通过调大对应 Column Family (CF) 的 soft-pending-compaction-bytes-limit 和 hard-pending-compaction-bytes-limit 参数来缓解：
    * 如果 pending compaction bytes 达到该阈值，RocksDB 会放慢写入速度。默认值 64GB，[rocksdb.defaultcf] soft-pending-compaction-bytes-limit = "128GB"。
    * 如果 pending compaction bytes 达到该阈值，RocksDB 会 stop 写入，通常不太可能触发该情况，因为在达到 soft-pending-compaction-bytes-limit 的阈值之后会放慢写入速度。默认值 256GB，hard-pending-compaction-bytes-limit = "512GB"。
    * 如果磁盘 IO 能力持续跟不上写入，建议扩容。如果磁盘的吞吐达到了上限（例如 SATA SSD 的吞吐相对 NVME SSD 会低很多）导致 write stall，但是 CPU 资源又比较充足，可以尝试采用压缩率更高的压缩算法来缓解磁盘的压力，用 CPU 资源换磁盘资源。
    * 比如 default cf compaction 压力比较大，调整参数 [rocksdb.defaultcf] compression-per-level = ["no", "no", "lz4", "lz4", "lz4", "zstd", "zstd"] 改成 compression-per-level = ["no", "no", "zstd", "zstd", "zstd", "zstd", "zstd"]。
  - memtable 太多导致 stall。该问题一般发生在瞬间写入量比较大，并且 memtable flush 到磁盘的速度比较慢的情况下。如果磁盘写入速度不能改善，并且只有业务峰值会出现这种情况，可以通过调大对应 CF 的 max-write-buffer-number 来缓解：
    * 例如 [rocksdb.defaultcf] max-write-buffer-number = 8（默认值 5），同时请求注意在高峰期可能会占用更多的内存，因为可能存在于内存中的 memtable 会更多。
- 4.3.2 scheduler too busy
  - 写入冲突严重，latch wait duration 比较高，查看监控：Grafana -> TiKV-details -> scheduler prewrite 或者 scheduler commit 的 latch wait duration。scheduler 写入任务堆积，导致超过了 [storage] scheduler-pending-write-threshold = "100MB" 设置的阈值。可通过查看 MVCC_CONFLICT_COUNTER 对应的 metric 来确认是否属于该情况。
  - 写入慢导致写入堆积，该 TiKV 正在写入的数据超过了 [storage] scheduler-pending-write-threshold = "100MB" 设置的阈值。请参考4.5 TiKV 写入慢。
- 4.3.3 raftstore is busy，主要是消息的处理速度没有跟上接收消息的速度。短时间的 channel full 不会影响服务，长时间持续出现该错误可能会导致 Leader 切换走。
  - append log 遇到了 stall，参考4.3.1 客户端报 server is busy 错误。

  - append log duration 比较高，导致处理消息不及时，可以参考4.5 TiKV 写入慢分析为什么 append log duration 比较高。
  - 瞬间收到大量消息（查看 TiKV Raft messages 面板），Raftstore 没处理过来，通常情况下短时间的 channel full 不会影响服务。

- 4.3.4 TiKV Coprocessor 排队，任务堆积超过了 Coprocessor 线程数 * readpool.coprocessor.max-tasks-per-worker-[normal|low|high]。大量大查询导致 Coprocessor 出现了堆积情况，需要确认是否由于执行计划变化而导致了大量扫表操作，请参考3.3 执行计划不对。

10.1.1.4.4 4.4 某些 TiKV 大量掉 Leader

- 4.4.1 TiKV 重启，导致重新选举。
  - TiKV panic 之后又被 systemd 重新拉起正常运行，可以通过查看 TiKV 的日志来确认是否有 panic，这种情况属于非预期，需报 bug。
  - 被第三者 stop/kill，被 systemd 重新拉起。查看 dmesg 和 TiKV log 确认原因。
  - TiKV 发生 OOM 导致重启了，参考4.2 TiKV OOM 问题。
  - 动态调整 THP 导致 hung 住，见案例 case-500。
- 4.4.2 查看监控：Grafana -> TiKV-details -> errors 面板 server is busy，看到 TiKV RocksDB 出现 write stall 导致发生重新选举，请参考4.3.1。
- 4.4.3 网络隔离导致重新选举。

10.1.1.4.5 4.5 TiKV 写入慢

- 4.5.1 通过查看 TiKV gRPC 的 prewrite/commit/raw-put（仅限 raw kv 集群）duration 确认确实是 TiKV 写入慢了。通常情况下可以按照 performance-map 来定位到底哪个阶段慢了，下面列出几种常见的情况。
- 4.5.2 scheduler CPU 繁忙（仅限 transaction kv）。prewrite/commit 的 scheduler command duration 比 scheduler latch wait duration + storage async write duration 更长，并且 scheduler worker CPU 比较高，例如超过 scheduler-worker-pool-size * 100% 的 80%，并且或者整个机器的 CPU 资源比较紧张。如果写入量很大，确认下是否 [storage] scheduler-worker-pool-size 配置得太小。其他情况，需报 bug。
- 4.5.3 Append log 慢。TiKV Grafana 的 Raft IO/append log duration 比较高，通常情况下是由于写盘慢了，可以检查 RocksDB - Raft 的 WAL Sync Duration max 值来确认，否则可能需要报 bug。
- 4.5.4 Raftstore 线程繁忙。TiKV Grafana 的 Raft Propose/propose wait duration 明显高于 append log duration。请查看以下情况：
  - [raftstore] store-pool-size 配置是否过小（该值建议在 [1,5] 之间，不建议太大）。
  - 机器的 CPU 是不是不够。
- 4.5.5 apply 慢了。TiKV Grafana 的 Raft IO/apply log duration 比较高，通常会伴随着 Raft Propose/apply wait duration 比较高。可能是以下原因引起的：
  - [raftstore] apply-pool-size 配置过小（建议在 [1, 5] 之间，不建议太大），Thread CPU/apply cpu 比较高；

  - 机器的 CPU 资源不够了；
  - Region 写入热点问题，单个 apply 线程 CPU 使用率比较高（通过修改 Grafana 表达式，加上 by (instance, name) 来看各个线程的 CPU 使用情况），暂时对于单个 Region 的热点写入没有很好的方式，最近在优化该场景；
  - 写 RocksDB 比较慢，RocksDB kv/max write duration 比较高（单个 Raft log 可能包含很多个 kv，写 RocksDB 的时候会把 128 个 kv 放在一个 write batch 写入到 RocksDB，所以一次 apply log 可能涉及到多次 RocksDB 的 write）；
  - 其他情况，需报 bug。

- 4.5.6 Raft commit log 慢了。

  - TiKV Grafana 的 Raft IO/commit log duration 比较高（4.x 版本的 Grafana 才有该 metric）。每个 Region 对应一个独立的 Raft group，Raft 本身是有流控机制的，类似 TCP 的滑动窗口机制，通过参数 [raftstore] raft-max-inflight-msgs = 256 来控制滑动窗口的大小，如果有热点写入并且 commit log duration 比较高可以适度调大该参数，比如 1024。

- 4.5.7 其他情况，请参考 Performance Map 上的写入路径来分析。

10.1.1.5 5. PD 问题

10.1.1.5.1 5.1 PD 调度问题

- 5.1.1 merge 问题：

  - 跨表空 Region 无法 merge，需要修改 TiKV 的 [coprocessor] split-region-on-table = false 参数来解决，4.x 版本该参数默认为 false，见案例 case-896。
  - Region merge 慢，可检查监控 Grafana -> PD -> operator 面板是否有 merge 的 operator 产生，可以适当调大 merge-schedule-limit 参数来加速 merge。

- 5.1.2 补副本/上下线问题：

  - TiKV 磁盘使用 80% 容量，PD 不会进行补副本操作，miss peer 数量上升，需要扩容 TiKV，见案例 case-801。
  - 下线 TiKV，有 Region 长时间迁移不走。v3.0.4 版本已经修复改问题，见 #5526 和案例 case-870。

- 5.1.3 Balance 问题：

  - Leader/Region count 分布不均，见案例 case-394, case-759。主要原因是 balance 是依赖 Region/leader 的 size 去调度的，所以可能会造成 count 数量的不均衡，v4.0 新增了一个参数 [leader-schedule-policy]，可以调整 Leader 的调度策略，根据 “count” 或者是 “size” 进行调度。

10.1.1.5.2 5.2 PD 选举问题

- 5.2.1 PD 发生 Leader 切换：

  - 磁盘问题，PD 所在的节点 I/O 被打满，排查是否有其他 I/O 高的组件与 PD 混部以及盘的健康情况，可通过监控 Grafana -> disk performance -> latency 和 load 等指标进行验证，必要时可以使用 fio 工具对盘进行检测，见案例 case-292。

- 网络问题，PD 日志中有 `lost the TCP streaming connection`，排查 PD 之间网络是否有问题，可通过监控 Grafana -> PD -> etcd 的 `round trip` 来验证，见案例 case-177。
- 系统 load 高，日志中能看到 `server is likely overloaded`，见案例 case-214。

- 5.2.2 PD 选不出 Leader 或者选举慢：
  - 选不出 Leader，PD 日志中有 `lease is not expired`，见 #10355。v3.0.x 版本和 v2.1.19 版本已修复该问题，见案例 case-875。
  - 选举慢，Region 加载时间长。从 PD 日志中 `grep "regions cost"`（例如日志中可能是 “load 460927 regions cost 11.77099s”），如果出现秒级，则说明较慢，v3.0 版本可开启 Region storage（设置 `use-region-storage` 为 `true`），该特性能极大缩短加载 Region 的时间，见案例 case-429。

- 5.2.3 TiDB 执行 SQL 时报 PD timeout：
  - PD 没 Leader 或者有切换，参考5.2.1 PD 选举问题和5.2.2 PD 选举问题。
  - 网络问题，排查网络相关情况。通过监控 Grafana -> blackbox_exporter -> ping latency 确定 TiDB 到 PD Leader 的网络是否正常。
  - PD panic，需报 bug。
  - PD OOM，参考5.3 PD OOM 问题。
  - 其他原因，通过 `curl http://127.0.0.1:2379/debug/pprof/goroutine?debug=2` 抓 goroutine，报 bug。

- 5.2.4 其他问题
  - PD 报 FATAL 错误，日志中有 `range failed to find revision pair`，v3.0.8 已经修复该问题，见 #2040。详情参考案例 case-947。
  - 其他原因，需报 bug。

10.1.1.5.3 5.3 PD OOM

- 5.3.1 使用 `/api/v1/regions` 接口时 Region 数量过多，可能会导致 PD OOM，在 v3.0.8 版本中修复，见 #1986。
- 5.3.2 滚动升级的时候 PD OOM，gRPC 消息大小没限制，监控可看到 TCP InSegs 较大，在 v3.0.6 版本中修复，见 #1952。

10.1.1.5.4 5.4 Grafana 显示问题

- 5.4.1 监控 Grafana -> PD -> cluster -> role 显示 follower，Grafana 表达式问题，在 v3.0.8 版本修复。

10.1.1.6 6. 生态 Tools 问题

10.1.1.6.1 6.1 TiDB Binlog 问题

- 6.1.1 TiDB Binlog 是将 TiDB 的修改同步给下游 TiDB 或者 MySQL 的工具，见 TiDB Binlog on GitHub。
- 6.1.2 Pump/Drainer Status 中 Update Time 正常更新，日志中也没有异常，但下游没有数据写入。
  - TiDB 配置中没有开启 binlog，需要修改 TiDB 配置 `[binlog]`。
- 6.1.3 Drainer 中的 sarama 报 EOF 错误。
  - Drainer 使用的 Kafka 客户端版本和 Kafka 版本不匹配，需要修改配置 `[syncer.to] kafka-version` 来解决。
- 6.1.4 Drainer 写 Kafka 失败然后 panic，Kafka 报 `Message was too large` 错误。
  - binlog 数据太大，造成写 Kafka 的单条消息太大，需要修改 Kafka 的下列配置来解决：

    ```
    message.max.bytes=1073741824
    replica.fetch.max.bytes=1073741824
    fetch.message.max.bytes=1073741824
    ```

    见案例 case-789。
- 6.1.5 上下游数据不一致
  - 部分 TiDB 节点没有开启 binlog。v3.0.6 及之后的版本可以通过访问 http://127.0.0.1:10080/info/all 接口可以检查所有节点的 binlog 状态。之前的版本可以通过查看配置文件来确认是否开启了 binlog。
  - 部分 TiDB 节点进入 ignore binlog 状态。v3.0.6 及之后的版本可以通过访问 http://127.0.0.1:10080/info/all 接口可以检查所有节点的 binlog 状态。之前的版本需要看 TiDB 的日志中是否有 ignore binlog 关键字来确认是该问题。
  - 上下游 Timestamp 列的值不一致：
    * 时区问题，需要确保 Drainer 和上下游数据库时区一致，Drainer 通过 `/etc/localtime` 获取时区，不支持 TZ 环境变量，见案例 case-826。
    * TiDB 中 Timestamp 的默认值为 `null`，MySQL 5.7 中 Timestamp 默认值为当前时间（MySQL 8 无此问题），因此当上游写入 `null` 的 Timestamp 且下游是 MySQL 5.7 时，Timestamp 列数据不一致。在开启 binlog 前，在上游执行 `set @@global.explicit_defaults_for_timestamp=on;` 可解决此问题。
  - 其他情况需报 bug。
- 6.1.6 同步慢
  - 下游是 TiDB/MySQL，上游频繁进行 DDL 操作，见案例 case-1023。
  - 下游是 TiDB/MySQL，需要同步的表中存在没有主键且没有唯一索引的表，这种情况会导致 binlog 性能下降，建议加主键或唯一索引。
  - 下游输出到文件，检查目标磁盘/网络盘是否慢。
  - 其他情况需报 bug。
- 6.1.7 Pump 无法写 binlog，报 `no space left on device` 错误。

  - 本地磁盘空间不足，Pump 无法正常写 binlog 数据。需要清理磁盘空间，然后重启 Pump。

- 6.1.8 Pump 启动时报错 `fail to notify all living drainer`。

  - Pump 启动时需要通知所有 Online 状态的 Drainer，如果通知失败则会打印该错误日志。
  - 可以使用 binlogctl 工具查看所有 Drainer 的状态是否有异常，保证 Online 状态的 Drainer 都在正常工作。如果某个 Drainer 的状态和实际运行情况不一致，则使用 binlogctl 修改状态，然后再重启 Pump。见案例fail-to-notify-all-living-drainer。

- 6.1.9 Drainer 报错 `gen update sqls failed: table xxx: row data is corruption []`。

  - 触发条件：上游做 Drop Column DDL 的时候同时在做这张表的 DML。已经在 v3.0.6 修复，见 case-820。

- 6.1.10 Drainer 同步卡住，进程活跃但 checkpoint 不更新。

  - 已知 bug 在 v3.0.4 修复，见案例 case-741。

- 6.1.11 任何组件 `panic`。

  - 需报 bug。

10.1.1.6.2 6.2 DM 问题

- 6.2.1 TiDB Data Migration (DM) 是能将 MySQL/MariaDB 的数据迁移到 TiDB 的迁移工具，详情见 DM on GitHub。

- 6.2.2 执行 `query-status` 或查看日志时出现 `Access denied for user 'root'@'172.31.43.27' (using password: YES)`。

  - 在所有 DM 配置文件中，数据库相关的密码都必须使用经 dmctl 加密后的密文（若数据库密码为空，则无需加密）。在 v1.0.6 及以后的版本可使用明文密码。
  - 在 DM 运行过程中，上下游数据库的用户必须具备相应的读写权限。在启动同步任务过程中，DM 会自动进行相应权限的检查。
  - 同一套 DM 集群，混合部署不同版本的 DM-worker/DM-master/dmctl，见案例 AskTUG-1049。

- 6.2.3 DM 同步任务中断并包含 `driver: bad connection` 错误。

  - 发生 `driver: bad connection` 错误时，通常表示 DM 到下游 TiDB 的数据库连接出现了异常（如网络故障、TiDB 重启等）且当前请求的数据暂时未能发送到 TiDB。
    * 1.0.0 GA 之前的版本，DM 发生该类型错误时，需要先使用 `stop-task` 命令停止任务后再使用 `start-task` 命令重启任务。
    * 1.0.0 GA 版本，增加对此类错误的自动重试机制，见 #265。

- 6.2.4 同步任务中断并包含 `invalid connection` 错误。

  - 发生 `invalid connection` 错误时，通常表示 DM 到下游 TiDB 的数据库连接出现了异常（如网络故障、TiDB 重启、TiKV busy 等）且当前请求已有部分数据发送到了 TiDB。由于 DM 中存在同步任务并发向下游复制数据的特性，因此在任务中断时可能同时包含多个错误（可通过 `query-status` 或 `query-error` 查询当前错误）：

    * 如果错误中仅包含 `invalid connection` 类型的错误，且当前处于增量复制阶段，则 DM 会自动进行重试。
    * 如果 DM 由于版本问题（v1.0.0-rc.1 后引入自动重试）等未自动进行重试或自动重试未能成功，则可尝试先使用 `stop-task` 停止任务，然后再使用 `start-task` 重启任务。

- 6.2.5 Relay 处理单元报错 `event from * in * diff from passed-in event *` 或同步任务中断并包含 `get binlog error ERROR 1236 (HY000)`、`binlog checksum mismatch, data may be corrupted` 等 binlog 获取或解析失败错误。

    - 在 DM 进行 relay log 拉取与增量同步过程中，如果遇到了上游超过 4 GB 的 binlog 文件，就可能出现这两个错误。原因是 DM 在写 relay log 时需要依据 binlog position 及文件大小对 event 进行验证，且需要保存同步的 binlog position 信息作为 checkpoint。但是 MySQL binlog position 官方定义使用 uint32 存储，所以超过 4 GB 部分的 binlog position 的 offset 值会溢出，进而出现上面的错误。
        * 对于 relay 处理单元，可通过官网步骤进行手动处理。
        * 对于 binlog replication 处理单元，可通过官网步骤进行手动处理。

- 6.2.6 DM 同步中断，日志报错 `ERROR 1236 (HY000)The slave is connecting using CHANGE MASTER TO MASTER_AUTO_POSITION = 1, but the master has purged binary logs containing GTIDs that the slave requires.`。

    - 检查 master 的 binlog 是否被 purge。
    - 检查 relay.meta 中记录的位点信息。
        * relay.meta 中记录空的 GTID 信息，DM-worker 进程在退出时、以及定时 (30s) 会把内存中的 GTID 信息保存到 relay.meta 中，在没有获取到上游 GTID 信息的情况下，把空的 GTID 信息保存到了 relay.meta 中。见案例 case-772。
        * relay.meta 中记录的 binlog event 不完整触发 recover 流程后记录错误的 GTID 信息，该问题可能会在 1.0.2 之前的版本遇到，已在 1.0.2 版本修复。

- 6.2.7 DM 同步报错 `Error 1366: incorrect utf8 value eda0bdedb29d(\ufffd\ufffd\ufffd\ufffd\ufffd\ufffd)`。

    - 该值 MySQL 8.0 和 TiDB 都不能写入成功，但是 MySQL 5.7 可以写入成功。可以开启 TiDB 动态参数 `tidb_skip_utf8_check` 参数，跳过数据格式检查。

10.1.1.6.3 6.3 TiDB Lightning 问题

- 6.3.1 TiDB Lightning 是快速的全量数据导入工具，见 TiDB Lightning on GitHub。

- 6.3.2 导入速度太慢。

    - `region-concurrency` 设定太高，线程间争用资源反而减低了效率。排查方法如下：
        * 从日志的开头搜寻 `region-concurrency` 能知道 TiDB Lightning 读到的参数是多少；
        * 如果 TiDB Lightning 与其他服务（如 TiKV Importer）共用一台服务器，必需手动将 `region-concurrency` 设为该服务器 CPU 数量的 75%；
        * 如果 CPU 设有限额（例如从 Kubernetes 指定的上限），TiDB Lightning 可能无法自动判断出来，此时亦需要手动调整 `region-concurrency`。

- 表结构太复杂。每条索引都会额外增加 KV 对，如果有 N 条索引，实际导入的大小就差不多是 Mydumper 文件的 N+1 倍。如果索引不太重要，可以考虑先从 schema 去掉，待导入完成后再使用 CREATE INDEX 加回去。
- TiDB Lightning 版本太旧。尝试使用最新的版本，可能会有改善。

- 6.3.3 `checksum failed: checksum mismatched remote vs local`
  - 原因 1：这张表可能本身已有数据，影响最终结果。
  - 原因 2：如果目标数据库的校验和全是 0，表示没有发生任何导入，有可能是集群太忙无法接收任何数据。
  - 原因 3：如果数据源是由机器生成而不是从 Mydumper 备份的，需确保数据符合表的限制。例如：
    * 自增 (AUTO_INCREMENT) 的列需要为正数，不能为 0。
    * 单一键和主键 (UNIQUE and PRIMARY KEYs) 不能有重复的值。
  - 解决办法：参考官网步骤处理。

- 6.3.4 `Checkpoint for … has invalid status:(`错误码`)`
  - 原因：断点续传已启用。TiDB Lightning 或 TiKV Importer 之前发生了异常退出。为了防止数据意外损坏，TiDB Lightning 在错误解决以前不会启动。错误码是小于 25 的整数，可能的取值是 0、3、6、9、12、14、15、17、18、20、21。整数越大，表示异常退出所发生的步骤在导入流程中越晚。
  - 解决办法：参考官网步骤处理。

- 6.3.5 `ResourceTemporarilyUnavailable("Too many open engines …: 8")`
  - 原因：并行打开的引擎文件 (engine files) 超出 tikv-importer 里的限制。这可能由配置错误引起。即使配置没问题，如果 tidb-lightning 曾经异常退出，也有可能令引擎文件残留在打开的状态，占据可用的数量。
  - 解决办法：参考官网步骤处理。

- 6.3.6 `cannot guess encoding for input file, please convert to UTF-8 manually`
  - 原因：TiDB Lightning 只支持 UTF-8 和 GB-18030 编码的表架构。此错误代表数据源不是这里任一个编码。也有可能是文件中混合了不同的编码，例如在不同的环境运行过 ALTER TABLE，使表架构同时出现 UTF-8 和 GB-18030 的字符。
  - 解决办法：参考官网步骤处理。

- 6.3.7 `[sql2kv] sql encode error = [types:1292]invalid time format: '{1970 1 1 0 45 0 0}'`
  - 原因：一个 timestamp 类型的时间戳记录了不存在的时间值。时间值不存在是由于夏令时切换或超出支持的范围 (1970 年 1 月 1 日至 2038 年 1 月 19 日)。
  - 解决办法：参考官网步骤处理。

10.1.1.7 7. 常见日志分析

10.1.1.7.1 7.1 TiDB

- 7.1.1 `GC life time is shorter than transaction duration`。事务执行时间太长，超过了 GC lifetime（默认为 10 分钟），可以通过修改系统变量`tidb_gc_life_time`来延长 life time，通常情况下不建议修改，因为延长时限可能导致大量老版本数据的堆积（如果有大量 UPDATE 和 DELETE 语句）。
- 7.1.2 `txn takes too much time`。事务太长时间（超过 590s）没有提交，准备提交的时候报该错误。可以通过调大 `[tikv-client] max-txn-time-use = 590` 参数，以及调大 `GC life time` 来绕过该问题（如果确实有这个需求）。通常情况下，建议看看业务是否真的需要执行这么长时间的事务。
- 7.1.3 coprocessor.go 报 `request outdated`。发往 TiKV 的 Coprocessor 请求在 TiKV 端排队时间超过了 60s，直接返回该错误。需要排查 TiKV Coprocessor 为什么排队这么严重。
- 7.1.4 region_cache.go 大量报 `switch region peer to next due to send request fail` 且 error 信息是 `context deadline exceeded`。请求 TiKV 超时触发 region cache 切换请求到其他节点，可以对日志中的 addr 字段继续 `grep "<addr> cancelled"`，根据 grep 结果：
  - `send request is cancelled`。请求发送阶段超时，可以排查监控 Grafana -> TiDB -> Batch Client/Pending Request Count by TiKV 是否大于 128，确定是否因发送远超 KV 处理能力导致发送堆积。如果 Pending Request 不多，需要排查日志确认是否因为对应 KV 有运维变更，导致短暂报出；否则非预期，需报 bug。
  - `wait response is cancelled`。请求发送到 TiKV 后超时未收到 TiKV 响应。需要排查对应地址 TiKV 的响应时间和对应 Region 在当时的 PD 和 KV 日志，确定为什么 KV 未及时响应。
- 7.1.5 distsql.go 报 `inconsistent index`。数据索引疑似发生不一致，首先对报错的信息中 index 所在表执行 `admin check table <TableName>` 命令，如果检查失败，则先通过以下命令禁用 GC，然后报 bug。

```
SET GLOBAL tidb_gc_enable = 0;
```

10.1.1.7.2 7.2 TiKV

- 7.2.1 `key is locked` 读写冲突，读请求碰到还未提交的数据，需要等待其提交之后才能读。少量这个错误对业务无影响，大量出现这个错误说明业务读写冲突比较严重。
- 7.2.2 `write conflict` 乐观事务中的写写冲突，同时多个事务对相同的 key 进行修改，只有一个事务会成功，其他事务会自动重取 timestamp 然后进行重试，不影响业务。如果业务冲突很严重可能会导致重试多次之后事务失败，这种情况下建议使用悲观锁。报错以及解决方法详情，参考乐观事务模型下写写冲突问题排查。
- 7.2.3 TxnLockNotFound 事务提交太慢，过了 TTL (Time To Live) 时间之后被其他事务回滚了，该事务会自动重试，通常情况下对业务无感知。对于 0.25 MB 以内的小事务，TTL 默认时间为 3 秒。详情参见锁被清除 (LockNotFound) 错误。
- 7.2.4 `PessimisticLockNotFound` 类似 `TxnLockNotFound`，悲观事务提交太慢被其他事务回滚了。
- 7.2.5 `stale_epoch` 请求的 epoch 太旧了，TiDB 会更新路由之后再重新发送请求，业务无感知。epoch 在 Region 发生 split/merge 以及迁移副本的时候会变化。

- 7.2.6 `peer is not leader` 请求发到了非 Leader 的副本上，TiDB 会根据该错误更新本地路由（如果错误 response 里携带了最新 Leader 是哪个副本这一信息），并且重新发送请求到最新 Leader，一般情况下业务无感知。在 v3.0 后 TiDB 在原 Leader 请求失败时会尝试其他 peer，也会导致 TiKV 频繁出现 `not leader` 日志，可以通过查看 TiDB 对应 Region 的 `switch region peer to next due to send request fail` 日志，排查发送失败根本原因，参考7.1.4 TiDB。另外也可能是由于其他原因导致一些 Region 一直没有 Leader，请参考4.4 某些 TiKV 大量掉 Leader。

10.1.2 TiDB 集群故障诊断

当试用 TiDB 遇到问题时，请先参考本篇文档。如果问题未解决，请按文档要求收集必要的信息通过 Github 提供给 TiDB 开发者。

10.1.2.1 如何给 TiDB 开发者报告错误

当使用 TiDB 遇到问题并且通过后面所列信息无法解决时，请收集以下信息并创建新 Issue:

- 具体的出错信息以及正在执行的操作
- 当前所有组件的状态
- 出问题组件 log 中的 error/fatal/panic 信息
- 机器配置以及部署拓扑
- dmesg 中 TiDB 组件相关的问题

10.1.2.2 数据库连接不上

首先请确认集群的各项服务是否已经启动，包括 tidb-server、pd-server、tikv-server。请用 ps 命令查看所有进程是否在。如果某个组件的进程已经不在了，请参考对应的章节排查错误。

如果所有的进程都在，请查看 tidb-server 的日志，看是否有报错？常见的错误包括：

- InformationSchema is out of date

  无法连接 tikv-server，请检查 pd-server 以及 tikv-server 的状态和日志。

- panic

  程序有错误，请将具体的 panic log 提供给 TiDB 开发者。

  如果是清空数据并重新部署服务，请确认以下信息：

- pd-server、tikv-server 数据都已清空

  tikv-server 存储具体的数据，pd-server 存储 tikv-server 中数据的元信息。如果只清空 pd-server 或只清空 tikv-server 的数据，会导致两边数据不匹配。

- 清空 pd-server 和 tikv-server 的数据并重启后，也需要重启 tidb-server

  集群 ID 是由 pd-server 在集群初始化时随机分配，所以重新部署集群后，集群 ID 会发生变化。tidb-server 业务需要重启以获取新的集群 ID。

10.1.2.3 tidb-server 启动报错

tidb-server 无法启动的常见情况包括：

- 启动参数错误

  请参考TiDB 命令行参数。

- 端口被占用：`lsof -i:port`

  请确保 tidb-server 启动所需要的端口未被占用。

- 无法连接 pd-server

  首先检查 pd-server 的进程状态和日志，确保 pd-server 成功启动，对应端口已打开：`lsof -i:port`。

  若 pd-server 正常，则需要检查 tidb-server 机器和 pd-server 对应端口之间的连通性，确保网段连通且对应服务端口已添加到防火墙白名单中，可通过 nc 或 curl 工具检查。

  例如，假设 tidb 服务位于 `192.168.1.100`，无法连接的 pd 位于 `192.168.1.101`，且 2379 为其 client port，则可以在 tidb 机器上执行 `nc -v -z 192.168.1.101 2379`，测试是否可以访问端口。或使用 `curl -v 192.168.1.101:2379/pd/api/v1/leader` 直接检查 pd 是否正常服务。

10.1.2.4 tikv-server 启动报错

- 启动参数错误

  请参考TiKV 启动参数文档。

- 端口被占用：`lsof -i:port`

  请确保 tikv-server 启动所需要的端口未被占用：`lsof -i:port`。

- 无法连接 pd-server

  首先检查 pd-server 的进程状态和日志。确保 pd-server 成功启动，对应端口已打开：`lsof -i:port`。

  若 pd-server 正常，则需要检查 tikv-server 机器和 pd-server 对应端口之间的连通性，确保网段连通且对应服务端口已添加到防火墙白名单中，可通过 nc 或 curl 工具检查。具体命令参考上一节。

- 文件被占用

  不要在一个数据库文件目录上打开两个 tikv。

10.1.2.5 pd-server 启动报错

- 启动参数错误

  请参考PD 命令行参数文档。

- 端口被占用：`lsof -i:port`

  请确保 pd-server 启动所需要的端口未被占用：`lsof -i:port`。

10.1.2.6 TiDB/TiKV/PD 进程异常退出

- 进程是否是启动在前台

  当前终端退出给其所有子进程发送 HUP 信号，从而导致进程退出。

- 是否是在命令行用过 nohup+& 方式直接运行

  这样依然可能导致进程因终端连接突然中断，作为终端 SHELL 的子进程被杀掉。

  推荐将启动命令写在脚本中，通过脚本运行（相当于二次 fork 启动）。

10.1.2.7 TiKV 进程异常重启

- 检查 dmesg 或者 syslog 里面是否有 OOM 信息

  如果有 OOM 信息并且杀掉的进程为 TiKV，请减少 TiKV 的 RocksDB 的各个 CF 的 `block-cache-size` 值。

- 检查 TiKV 日志是否有 panic 的 log

  提交 Issue 并附上 panic 的 log。

10.1.2.8 TiDB panic

请提供 panic 的 log。

10.1.2.9 连接被拒绝

- 请确保操作系统的网络参数正确，包括但不限于
  - 连接字符串中的端口和 tidb-server 启动的端口需要一致
  - 请保证防火墙的配置正确

10.1.2.10 Too many open files

在启动进程之前，请确保 `ulimit -n` 的结果足够大，推荐设为 unlimited 或者是大于 1000000。

10.1.2.11 数据库访问超时，系统负载高

首先检查SLOW-QUERY 日志，判断是否是因为某条 SQL 语句导致。如果未能解决，请提供如下信息：

- 部署的拓扑结构
  - tidb-server/pd-server/tikv-server 部署了几个实例
  - 这些实例在机器上是如何分布的
- 机器的硬件配置
  - CPU 核数
  - 内存大小
  - 硬盘类型（SSD 还是机械硬盘）
  - 是实体机还是虚拟机

- 机器上除了 TiDB 集群之外是否还有其他服务
- pd-server 和 tikv-server 是否分开部署
- 目前正在进行什么操作
- 用 `top -H` 命令查看当前占用 CPU 的线程名
- 最近一段时间的网络/IO 监控数据是否有异常

10.1.3 TiFlash 常见问题

本文介绍了一些 TiFlash 常见问题、原因及解决办法。

10.1.3.1 TiFlash 未能正常启动

该问题可能由多个因素构成，可以通过以下步骤依次排查：

1. 检查系统环境是否是 CentOS8。

   CentOS8 中缺少 `libnsl.so` 系统库，可以通过手动安装的方式解决：

   ```
   dnf install libnsl
   ```

2. 检查系统的 `ulimit` 参数设置。

   ```
   ulimit -n 1000000
   ```

3. 使用 PD Control 工具检查在该节点（相同 IP 和 Port）是否有之前未成功下线的 TiFlash 实例，并将它们强制下线。（下线步骤参考手动缩容 TiFlash 节点）

如果遇到上述方法无法解决的问题，可以打包 TiFlash 的 log 文件夹，并在 AskTUG 社区中提问。

10.1.3.2 TiFlash 副本始终处于不可用状态

该问题一般由于配置错误或者环境问题导致 TiFlash 处于异常状态，可以先通过以下步骤定位问题组件：

1. 使用 pd-ctl 检查 PD 的Placement Rules 功能是否开启：

   ```
   echo 'config show replication' | /path/to/pd-ctl -u http://${pd-ip}:${pd-port}
   ```

   - 如果返回 `true`，进入下一步。
   - 如果返回 `false`，你需要先开启 Placement Rules 特性 后再进入下一步。

2. 通过 TiFlash-Summary 监控面板下的 UpTime 检查操作系统中 TiFlash 进程是否正常。

3. 通过 pd-ctl 查看 TiFlash proxy 状态是否正常：

   ```
   echo "store" | /path/to/pd-ctl -u http://${pd-ip}:${pd-port}
   ```

   store.labels 中含有 `{"key": "engine", "value": "tiflash"}` 信息的为 TiFlash proxy。

4. 查看 pd buddy 是否正常打印日志（日志路径的对应配置项 [flash.flash_cluster] log 设置的值，默认为 TiFlash 配置文件配置的 tmp 目录下）。
5. 检查配置的副本数是否小于等于集群 TiKV 节点数。若配置的副本数超过 TiKV 节点数，则 PD 不会向 TiFlash 同步数据；

```
echo 'config placement-rules show' | /path/to/pd-ctl -u http://${pd-ip}:${pd-port}
```

再确认 “default: count” 参数值。

> **注意：**
> 开启 Placement Rules 后，原先的 `max-replicas` 及 `location-labels` 配置项将不再生效。如果需要调整副本策略，应当使用 Placement Rules 相关接口。

6. 检查 TiFlash 节点对应 store 所在机器剩余的磁盘空间是否充足。默认情况下当磁盘剩余空间小于该 store 的 capacity 的 20%（通过 low-space-ratio 参数控制）时，PD 不会向 TiFlash 调度数据。

10.1.3.3 部分查询返回 Region Unavailable 的错误

如果在 TiFlash 上的负载压力过大，会导致 TiFlash 数据同步落后，部分查询可能会返回 `Region Unavailable` 的错误。

在这种情况下，可以增加 TiFlash 节点分担负载压力。

10.1.3.4 数据文件损坏

可依照如下步骤进行处理：

1. 参照下线 TiFlash 节点下线对应的 TiFlash 节点。
2. 清除该 TiFlash 节点的相关数据。
3. 重新在集群中部署 TiFlash 节点。

10.1.3.5 TiFlash 分析慢

如果语句中含有 MPP 模式不支持的算子或函数等，TiDB 不会选择 MPP 模式，可能导致分析慢。此时，可以执行 EXPLAIN 语句检查 SQL 中是否含有 TiFlash 不支持的函数或算子。

```
create table t(a datetime);
alter table t set tiflash replica 1;
insert into t values('2022-01-13');
set @@session.tidb_enforce_mpp=1;
explain select count(*) from t where subtime(a, '12:00:00') > '2022-01-01' group by a;
show warnings;
```

示例中，warning 消息显示，因为 TiDB 5.4 及更早的版本尚不支持 `subtime` 函数的下推，因此 TiDB 没有选择 MPP 模式。

```
+---------+------+-----------------------------------------------------------------------------------+
> | Level   | Code | Message
    ↪ |
+---------+------+-----------------------------------------------------------------------------------+
| Warning | 1105 | Scalar function 'subtime'(signature: SubDatetimeAndString, return type:
    ↪ datetime) is not supported to push down to tiflash now.       |
+---------+------+-----------------------------------------------------------------------------------+
```

10.1.3.6 TiFlash 数据不同步

在部署完 TiFlash 节点且进行了数据的同步操作（ALTER 操作）之后，如果实际没有数据同步到 TiFlash 节点，你可以通过以下步骤确认或解决问题：

1. 检查同步操作是否执行。

   执行 `ALTER table <tbl_name> set tiflash replica <num>` 操作，查看是否有正常返回:

   - 如果有正常返回，进入下一步。
   - 如果无正常返回，请执行 `SELECT * FROM information_schema.tiflash_replica` 检查是否已经创建 TiFlash replica。如果没有，请重新执行 `ALTER table ${tbl_name} set tiflash replica ${num}`，查看是否有其他执行语句（如 `add index`），或者检查 DDL 操作是否正常。

2. 检查 TiFlash Region 同步是否正常。

   查看 `progress` 是否有变化:

   - 如果有变化，说明 TiFlash 同步正常，进入下一步。
   - 如果没有变化，说明 TiFlash 同步异常，在 `tidb.log` 中，搜索 `Tiflash replica is not available` 相关日志。检查对应表的 `progress` 是否更新。如果无更新，请检查 `tiflash log` 来获取更多信息。例如，在 `tiflash log` 中搜索 `lag_region_info` 来判断同步落后的 Region。

3. 使用 pd-ctl 检查 PD 的Placement Rules 功能是否开启：

   ```
   echo 'config show replication' | /path/to/pd-ctl -u http://<pd-ip>:<pd-port>
   ```

   - 如果返回 `true`，进入下一步。
   - 如果返回 `false`，你需要先开启 Placement Rules 特性，然后进入下一步。

4. 检查集群副本数 `max-replicas` 配置是否合理。

   - 如果 `max-replicas` 取值未超过 TiKV 节点数，进入下一步。
   - 如果 `max-replicas` 超过 TiKV 节点数，PD 不会向 TiFlash 同步数据。此时，请将 `max-replicas` 修改为小于等于 TiKV 节点数的整数。

> **注意：**
>
> `max-replicas` 的默认值是 3。在生产环境中，TiKV 节点数一般大于该值；在测试环境中，可以修改为 1。

```
curl -X POST -d '{
    "group_id": "pd",
    "id": "default",
    "start_key": "",
    "end_key": "",
    "role": "voter",
    "count": 3,
    "location_labels": [
    "host"

    ]
}' <http://172.16.x.xxx:2379/pd/api/v1/config/rule>
```

5. 检查 TiDB 是否为表创建 Placement rule。

   搜索 TiDB DDL Owner 的日志，检查 TiDB 是否通知 PD 添加 Placement rule。对于非分区表搜索 `ConfigureTiFlashPDForTable`；对于分区表，搜索 `ConfigureTiFlashPDForPartitions`。

   - 有关键字，进入下一步。
   - 没有关键字，收集相关组件的日志进行排查。

6. 检查 PD 是否为表设置 Placement rule。

   可以通过 `curl http://<pd-ip>:<pd-port>/pd/api/v1/config/rules/group/tiflash` 查询比较当前 PD 上的所有 TiFlash 的 Placement rule。如果观察到有 id 为 `table-<table_id>-r` 的 Rule，则表示 PD rule 设置成功。

7. 检查 PD 是否正常发起调度。

   查看 `pd.log` 日志是否出现 `table-<table_id>-r` 关键字，且之后是否出现 `add operator` 之类的调度行为。

   - 是，PD 调度正常。
   - 否，PD 调度异常。

10.1.3.7 TiFlash 数据同步卡住

如果 TiFlash 数据一开始可以正常同步，过一段时间后全部或者部分数据无法继续同步，你可以通过以下步骤确认或解决问题：

1. 检查磁盘空间。

   检查磁盘使用空间比例是否高于 `low-space-ratio` 的值（默认值 0.8，即当节点的空间占用比例超过 80% 时，为避免磁盘空间被耗尽，PD 会尽可能避免往该节点迁移数据）。

   - 如果磁盘使用率大于等于 `low-space-ratio`，说明磁盘空间不足。此时，请删除不必要的文件，如 `${data}/flash/` 目录下的 `space_placeholder_file` 文件（必要时可在删除文件后将 `reserve-space` 设置为 0MB）。
   - 如果磁盘使用率小于 `low-space-ratio`，说明磁盘空间正常，进入下一步。

2. 检查是否有 `down peer`（`down peer` 没有清理干净可能会导致同步卡住）。

- 执行 pd-ctl region check-down-peer 命令检查是否有 down peer。
- 如果存在 down peer，执行 pd-ctl operator add remove-peer <region-id> <tiflash-store-id> 命令将其清除。

10.1.3.8 数据同步慢

同步慢可能由多种原因引起，你可以按以下步骤进行排查。

1. 调大调度参数store limit，加快同步速度。
2. 调整 TiFlash 侧负载。

   TiFlash 负载过大会引起同步慢，可通过 Grafana 中的 TiFlash-Summary 面板查看各个指标的负载情况：

   - Applying snapshots Count: TiFlash-summary > raft > Applying snapshots Count
   - Snapshot Predecode Duration: TiFlash-summary > raft > Snapshot Predecode Duration
   - Snapshot Flush Duration: TiFlash-summary > raft > Snapshot Flush Duration
   - Write Stall Duration: TiFlash-summary > Storage Write Stall > Write Stall Duration
   - generate snapshot CPU: TiFlash-Proxy-Details > Thread CPU > Region task worker pre-handle/ ↪ generate snapshot CPU

   根据业务优先级，调整负载情况。

10.2 故障场景

10.2.1 慢查询

10.2.1.1 慢查询日志

TiDB 会将执行时间超过tidb_slow_log_threshold（默认值为 300 毫秒）的语句输出到slow-query-file（默认值为“tidb-slow.log”）日志文件中，用于帮助用户定位慢查询语句，分析和解决 SQL 执行的性能问题。

TiDB 默认启用慢查询日志，可以修改系统变量tidb_enable_slow_log 来启用或禁用它。

10.2.1.1.1 日志示例

```
#### Time: 2019-08-14T09:26:59.487776265+08:00
#### Txn_start_ts: 410450924122144769
#### User@Host: root[root] @ localhost [127.0.0.1]
#### Conn_ID: 3086
#### Exec_retry_time: 5.1 Exec_retry_count: 3
#### Query_time: 1.527627037
#### Parse_time: 0.000054933
#### Compile_time: 0.000129729
#### Rewrite_time: 0.000000003 Preproc_subqueries: 2 Preproc_subqueries_time: 0.000000002
#### Optimize_time: 0.00000001
#### Wait_TS: 0.00001078
```

```
#### Process_time: 0.07 Request_count: 1 Total_keys: 131073 Process_keys: 131072 Prewrite_time:
    ↪ 0.335415029 Commit_time: 0.032175429 Get_commit_ts_time: 0.000177098
    ↪ Local_latch_wait_time: 0.106869448 Write_keys: 131072 Write_size: 3538944 Prewrite_region
    ↪ : 1
#### DB: test
#### Is_internal: false
#### Digest: 50a2e32d2abbd6c1764b1b7f2058d428ef2712b029282b776beb9506a365c0f1
#### Stats: t:pseudo
#### Num_cop_tasks: 1
#### Cop_proc_avg: 0.07 Cop_proc_p90: 0.07 Cop_proc_max: 0.07 Cop_proc_addr: 172.16.5.87:20171
#### Cop_wait_avg: 0 Cop_wait_p90: 0 Cop_wait_max: 0 Cop_wait_addr: 172.16.5.87:20171
#### Cop_backoff_regionMiss_total_times: 200 Cop_backoff_regionMiss_total_time: 0.2
    ↪ Cop_backoff_regionMiss_max_time: 0.2 Cop_backoff_regionMiss_max_addr: 127.0.0.1
    ↪ Cop_backoff_regionMiss_avg_time: 0.2 Cop_backoff_regionMiss_p90_time: 0.2
#### Cop_backoff_rpcPD_total_times: 200 Cop_backoff_rpcPD_total_time: 0.2
    ↪ Cop_backoff_rpcPD_max_time: 0.2 Cop_backoff_rpcPD_max_addr: 127.0.0.1
    ↪ Cop_backoff_rpcPD_avg_time: 0.2 Cop_backoff_rpcPD_p90_time: 0.2
#### Cop_backoff_rpcTiKV_total_times: 200 Cop_backoff_rpcTiKV_total_time: 0.2
    ↪ Cop_backoff_rpcTiKV_max_time: 0.2 Cop_backoff_rpcTiKV_max_addr: 127.0.0.1
    ↪ Cop_backoff_rpcTiKV_avg_time: 0.2 Cop_backoff_rpcTiKV_p90_time: 0.2
#### Mem_max: 525211
#### Disk_max: 65536
#### Prepared: false
#### Plan_from_cache: false
#### Succ: true
#### Plan: tidb_decode_plan('
    ↪ ZJAwCTMyXzcJMAkyMAlkYXRhOlRhYmxlU2Nhbl82CjEJMTBfNgkxAR0AdAEY1Dp0LCByYW5nZTpbLWluZiwraW5mXSwga2VlcCBvcmRlc
    ↪ ==')
use test;
insert into t select * from t;
```

10.2.1.1.2 字段含义说明

> **注意：**
>
> 慢查询日志中所有时间相关字段的单位都是“秒”

Slow Query 基础信息：

- `Time`：表示日志打印时间。
- `Query_time`：表示执行这个语句花费的时间。
- `Parse_time`：表示这个语句在语法解析阶段花费的时间。

- Compile_time：表示这个语句在查询优化阶段花费的时间。
- Optimize_time：表示这个语句在优化查询计划阶段花费的时间。
- Wait_TS：表示这个语句在等待获取事务 TS 阶段花费的时间。
- Query：表示 SQL 语句。慢日志里面不会打印 Query，但映射到内存表后，对应的字段叫 Query。
- Digest：表示 SQL 语句的指纹。
- Txn_start_ts：表示事务的开始时间戳，也是事务的唯一 ID，可以用这个值在 TiDB 日志中查找事务相关的其他日志。
- Is_internal：表示是否为 TiDB 内部的 SQL 语句。true 表示 TiDB 系统内部执行的 SQL 语句，false 表示用户执行的 SQL 语句。
- Index_names：表示这个语句执行用到的索引。
- Stats：表示这个语句涉及表的统计信息健康状态，pseudo 状态表示统计信息状态不健康。
- Succ：表示语句是否执行成功。
- Backoff_time：表示语句遇到需要重试的错误时在重试前等待的时间。常见的需要重试的错误有以下几种：遇到了 lock、Region 分裂、tikv server is busy。
- Plan：表示语句的执行计划，用 select tidb_decode_plan('xxx...') SQL 语句可以解析出具体的执行计划。
- Binary_plan：表示以二进制格式编码后的语句的执行计划，用 select tidb_decode_binary_plan('xxx ↪ ...') SQL 语句可以解析出具体的执行计划。传递的信息和 Plan 字段基本相同，但是解析出的执行计划的格式会和 Plan 字段不同。
- Prepared：表示这个语句是否是 Prepare 或 Execute 的请求。
- Plan_from_cache：表示这个语句是否命中了执行计划缓存。
- Plan_from_binding：表示这个语句是否用的绑定的执行计划。
- Has_more_results：表示这个语句的查询结果是否还有更多的数据待用户发起 fetch 命令获取。
- Rewrite_time：表示这个语句在查询改写阶段花费的时间。
- Preproc_subqueries：表示这个语句中被提前执行的子查询个数，如 where id in (select if from t) 这个子查询就可能被提前执行。
- Preproc_subqueries_time：表示这个语句中被提前执行的子查询耗时。
- Exec_retry_count：表示这个语句执行的重试次数。一般出现在悲观事务中，上锁失败时重试执行该语句。
- Exec_retry_time：表示这个语句的重试执行时间。例如某个查询一共执行了三次（前两次失败），则 Exec_retry_time 表示前两次的执行时间之和，Query_time 减去 Exec_retry_time 则为最后一次执行时间。
- KV_total：表示这个语句在 TiKV/TiFlash 上所有 RPC 请求花费的时间。
- PD_total：表示这个语句在 PD 上所有 RPC 请求花费的时间。
- Backoff_total：表示这个语句在执行过程中所有 backoff 花费的时间。
- Write_sql_response_total：表示这个语句把结果发送回客户端花费的时间。
- Result_rows：表示这个语句查询结果的行数。
- Warnings：表示这个语句执行过程中产生的警告，采用 JSON 格式。通常和SHOW WARNINGS 语句的输出结果一致，但是可能会包含SHOW WARNINGS 中没有的警告，因而可以提供更多诊断信息。这类警告将被标记为 IsExtra: true。
- IsExplicitTxn：表示这个语句是否在一个明确声明的事务中。如果是 false，表示这个语句的事务是 autocommit=1，即语句执行完成后就自动提交的事务。

和事务执行相关的字段：

- Prewrite_time：表示事务两阶段提交中第一阶段（prewrite 阶段）的耗时。

- `Commit_time`：表示事务两阶段提交中第二阶段（commit 阶段）的耗时。
- `Get_commit_ts_time`：表示事务两阶段提交中第二阶段（commit 阶段）获取 commit 时间戳的耗时。
- `Local_latch_wait_time`：表示事务两阶段提交中第二阶段（commit 阶段）发起前在 TiDB 侧等锁的耗时。
- `Write_keys`：表示该事务向 TiKV 的 Write CF 写入 Key 的数量。
- `Write_size`：表示事务提交时写 key 或 value 的总大小。
- `Prewrite_region`：表示事务两阶段提交中第一阶段（prewrite 阶段）涉及的 TiKV Region 数量。每个 Region 会触发一次远程过程调用。

和内存使用相关的字段：

- `Mem_max`：表示执行期间 TiDB 使用的最大内存空间，单位为 byte。

和硬盘使用相关的字段：

- `Disk_max`：表示执行期间 TiDB 使用的最大硬盘空间，单位为 byte。

和 SQL 执行的用户相关的字段：

- `User`：表示执行语句的用户名。
- `Host`：表示执行语句的用户地址。
- `Conn_ID`：表示用户的链接 ID，可以用类似 `con:3` 的关键字在 TiDB 日志中查找该链接相关的其他日志。
- `DB`：表示执行语句时使用的 database。

和 TiKV Coprocessor Task 相关的字段：

- `Request_count`：表示这个语句发送的 Coprocessor 请求的数量。
- `Total_keys`：表示 Coprocessor 扫过的 key 的数量。
- `Process_time`：执行 SQL 在 TiKV 的处理时间之和，因为数据会并行的发到 TiKV 执行，这个值可能会超过 `Query_time`。
- `Wait_time`：表示这个语句在 TiKV 的等待时间之和，因为 TiKV 的 Coprocessor 线程数是有限的，当所有的 Coprocessor 线程都在工作的时候，请求会排队；当队列中有某些请求耗时很长的时候，后面的请求的等待时间都会增加。
- `Process_keys`：表示 Coprocessor 处理的 key 的数量。相比 total_keys，processed_keys 不包含 MVCC 的旧版本。如果 processed_keys 和 total_keys 相差很大，说明旧版本比较多。
- `Num_cop_tasks`：表示这个语句发送的 Coprocessor 请求的数量。
- `Cop_proc_avg`：cop-task 的平均执行时间，包括一些无法统计的等待时间，如 RocksDB 内的 mutex。
- `Cop_proc_p90`：cop-task 的 P90 分位执行时间。
- `Cop_proc_max`：cop-task 的最大执行时间。
- `Cop_proc_addr`：执行时间最长的 cop-task 所在地址。
- `Cop_wait_avg`：cop-task 的平均等待时间，包括请求排队和获取 snapshot 时间。
- `Cop_wait_p90`：cop-task 的 P90 分位等待时间。
- `Cop_wait_max`：cop-task 的最大等待时间。
- `Cop_wait_addr`：等待时间最长的 cop-task 所在地址。
- `Cop_backoff_{backoff-type}_total_times`：因某种错误造成的 backoff 总次数。
- `Cop_backoff_{backoff-type}_total_time`：因某种错误造成的 backoff 总时间。

- Cop_backoff_{backoff-type}_max_time：因某种错误造成的最大 backoff 时间。
- Cop_backoff_{backoff-type}_max_addr：因某种错误造成的最大 backoff 时间的 cop-task 地址。
- Cop_backoff_{backoff-type}_avg_time：因某种错误造成的平均 backoff 时间。
- Cop_backoff_{backoff-type}_p90_time：因某种错误造成的 P90 分位 backoff 时间。

10.2.1.1.3 相关系统变量

- tidb_slow_log_threshold：设置慢日志的阈值，执行时间超过阈值的 SQL 语句将被记录到慢日志中。默认值是 300 ms。
- tidb_query_log_max_len：设置慢日志记录 SQL 语句的最大长度。默认值是 4096 byte。
- tidb_redact_log：设置慢日志记录 SQL 时是否将用户数据脱敏用 ? 代替。默认值是 0，即关闭该功能。
- tidb_enable_collect_execution_info：设置是否记录执行计划中各个算子的物理执行信息，默认值是 1。该功能对性能的影响约为 3%。开启该项后查看 Plan 的示例如下：

```
> select tidb_decode_plan('jAOIMAk1XzE3CTAJMQlmdW5jczpjb3VudChDb2x1bW4jNyktPkMJC/
    ↪ BMNQkxCXRpbWU6MTAuOTMxNTA1bXMsIGxvb3BzOjIJMzcyIEJ5dGVzCU4vQQoxCTMyXzE4CTAJMQlpbmRleDpTdHJlYW1BZ2dfOQkxCXQ
    ↪ ');
+--
    ↪ ------------------------------------------------------------------------------------------------------
    ↪
| tidb_decode_plan('jAOIMAk1XzE3CTAJMQlmdW5jczpjb3VudChDb2x1bW4jNyktPkMJC/
    ↪ BMNQkxCXRpbWU6MTAuOTMxNTA1bXMsIGxvb3BzOjIJMzcyIEJ5dGVzCU4vQQoxCTMyXzE4CTAJMQlpbmRleDpTdHJlYW1BZ2dfOQkxCXQ
    ↪  |
+------------------------------------------------------------------------------------------------------
    ↪
|     id                    task    estRows               operator info
    ↪                                                   actRows    execution info
    ↪                                                                 memory       disk
    ↪                              |
|     StreamAgg_17          root    1                     funcs:count(Column#7)->Column#5
    ↪                                1          time:10.931505ms, loops:2
    ↪                                                      372 Bytes    N/A
    ↪                              |
|     └─IndexReader_18      root    1                     index:StreamAgg_9
    ↪                                              1          time:10.927685ms, loops:2, rpc
    ↪ num: 1, rpc time:10.884355ms, proc keys:25007    206 Bytes    N/A
    ↪                              |
|       └─StreamAgg_9       cop     1                     funcs:count(1)->Column#7
    ↪                                        1          time:11ms, loops:25
    ↪                                                             N/A          N/A
    ↪                              |
|         └─IndexScan_16    cop     31281.857819905217    table:t, index:idx(a), range:[-inf
    ↪ ,50000), keep order:false    25007      time:11ms, loops:25
    ↪                                                             N/A          N/A
    ↪                              |
```

```
+-----------------------
```

```
+----------+----------------------------+----------------------------+
| count(*) | min(time)                  | max(time)                  |
+----------+----------------------------+----------------------------+
| 2618049  | 2020-03-10 00:00:00.427138 | 2020-03-10 23:00:22.716728 |
+----------+----------------------------+----------------------------+
```

> **注意：**
>
> 如果指定时间范围内的慢日志文件被删除，或者并没有慢查询，则查询结果会返回空。

TiDB 4.0 中新增了CLUSTER_SLOW_QUERY 系统表，用来查询所有 TiDB 节点的慢查询信息，表结构在 SLOW_QUERY 的基础上多增加了 INSTANCE 列，表示该行慢查询信息来自的 TiDB 节点地址。使用方式和SLOW_QUERY 系统表一样。

关于查询 CLUSTER_SLOW_QUERY 表，TiDB 会把相关的计算和判断下推到其他节点执行，而不是把其他节点的慢查询数据都取回来在一台 TiDB 上执行。

10.2.1.1.5 查询 SLOW_QUERY/CLUSTER_SLOW_QUERY 示例

搜索 Top N 的慢查询

查询 Top 2 的用户慢查询。is_internal=false 表示排除 TiDB 内部的慢查询，只看用户的慢查询：

```
select query_time, query
from information_schema.slow_query
where is_internal = false  -- 排除 TiDB 内部的慢查询 SQL
order by query_time desc
limit 2;
```

输出样例：

```
+--------------+-----------------------------------------------------------------+
| query_time   | query                                                           |
+--------------+-----------------------------------------------------------------+
| 12.77583857  | select * from t_slim, t_wide where t_slim.c0=t_wide.c0;         |
|  0.734982725 | select t0.c0, t1.c1 from t_slim t0, t_wide t1 where t0.c0=t1.c0; |
+--------------+-----------------------------------------------------------------+
```

搜索某个用户的 Top N 慢查询

下面例子中搜索 test 用户执行的慢查询 SQL，且按执行消耗时间逆序排序显式前 2 条：

```
select query_time, query, user
from information_schema.slow_query
where is_internal = false  -- 排除 TiDB 内部的慢查询 SQL
```

```
  and user = "test"        -- 查找的用户名
order by query_time desc
limit 2;
```

输出样例：

```
+-------------+-----------------------------------------------------------------------+----------------+
    ↪
| Query_time  | query                                                                 | user
    ↪           |
+-------------+-----------------------------------------------------------------------+----------------+
    ↪
| 0.676408014 | select t0.c0, t1.c1 from t_slim t0, t_wide t1 where t0.c0=t1.c1; | test
    ↪           |
+-------------+-----------------------------------------------------------------------+----------------+
    ↪
```

根据 SQL 指纹搜索同类慢查询

在得到 Top N 的慢查询 SQL 后，可通过 SQL 指纹继续搜索同类慢查询 SQL。

先获取 Top N 的慢查询和对应的 SQL 指纹：

```
select query_time, query, digest
from information_schema.slow_query
where is_internal = false
order by query_time desc
limit 1;
```

输出样例：

```
+-------------+----------------------------+-------------------------------------------------------------------+
    ↪
| query_time  | query                      | digest
    ↪                                                                 |
+-------------+----------------------------+-------------------------------------------------------------------+
    ↪
| 0.302558006 | select * from t1 where a=1; | 4751
    ↪ cb6008fda383e22dacb601fde85425dc8f8cf669338d55d944bafb46a6fa |
+-------------+----------------------------+-------------------------------------------------------------------+
    ↪
```

再根据 SQL 指纹搜索同类慢查询：

```
select query, query_time
from information_schema.slow_query
where digest = "4751cb6008fda383e22dacb601fde85425dc8f8cf669338d55d944bafb46a6fa";
```

输出样例：

```
+-----------------------------+-------------+
| query                       | query_time  |
+-----------------------------+-------------+
| select * from t1 where a=1; | 0.302558006 |
| select * from t1 where a=2; | 0.401313532 |
+-----------------------------+-------------+
```

搜索统计信息为 pseudo 的慢查询 SQL 语句

```
select query, query_time, stats
from information_schema.slow_query
where is_internal = false
  and stats like '%pseudo%';
```

输出样例：

```
+-----------------------------+-------------+---------------------------------+
| query                       | query_time  | stats                           |
+-----------------------------+-------------+---------------------------------+
| select * from t1 where a=1; | 0.302558006 | t1:pseudo                       |
| select * from t1 where a=2; | 0.401313532 | t1:pseudo                       |
| select * from t1 where a>2; | 0.602011247 | t1:pseudo                       |
| select * from t1 where a>3; | 0.50077719  | t1:pseudo                       |
| select * from t1 join t2;   | 0.931260518 | t1:407872303825682445,t2:pseudo |
+-----------------------------+-------------+---------------------------------+
```

查询执行计划发生变化的慢查询

由于统计信息过时，或者统计信息因为误差无法精确反映数据的真实分布情况时，可能导致同类型 SQL 的执行计划发生改变导致执行变慢，可以用以下 SQL 查询哪些 SQL 具有不同的执行计划：

```
select count(distinct plan_digest) as count,
       digest,
       min(query)
from cluster_slow_query
group by digest
having count > 1
limit 3\G
```

输出样例：

```
***************************[ 1. row ]***************************
count      | 2
digest     | 17b4518fde82e32021877878bec2bb309619d384fca944106fcaf9c93b536e94
min(query) | SELECT DISTINCT c FROM sbtest25 WHERE id BETWEEN ? AND ? ORDER BY c [arguments:
    ↪ (291638, 291737)];
***************************[ 2. row ]***************************
```

```
count      | 2
digest     | 9337865f3e2ee71c1c2e740e773b6dd85f23ad00f8fa1f11a795e62e15fc9b23
min(query) | SELECT DISTINCT c FROM sbtest22 WHERE id BETWEEN ? AND ? ORDER BY c [arguments:
    ↪ (215420, 215519)];
***************************[ 3. row ]***************************
count      | 2
digest     | db705c89ca2dfc1d39d10e0f30f285cbbadec7e24da4f15af461b148d8ffb020
min(query) | SELECT DISTINCT c FROM sbtest11 WHERE id BETWEEN ? AND ? ORDER BY c [arguments:
    ↪ (303359, 303458)];
```

然后可以用查询结果中的 SQL 指纹进一步查询不同的 plan

```
select min(plan),
       plan_digest
from cluster_slow_query
where digest='17b4518fde82e32021877878bec2bb309619d384fca944106fcaf9c93b536e94'
group by plan_digest\G
```

输出样例：

```
*************************** 1. row ***************************
  min(plan):    Sort_6                  root    100.00131380758702      sbtest.sbtest25.c:asc
        └─HashAgg_10            root    100.00131380758702      group by:sbtest.sbtest25.c,
            ↪ funcs:firstrow(sbtest.sbtest25.c)->sbtest.sbtest25.c
          └─TableReader_15      root    100.00131380758702      data:TableRangeScan_14
            └─TableScan_14      cop     100.00131380758702      table:sbtest25, range
                ↪ :[502791,502890], keep order:false
plan_digest: 6afbbd21f60ca6c6fdf3d3cd94f7c7a49dd93c00fcf8774646da492e50e204ee
*************************** 2. row ***************************
  min(plan):    Sort_6                  root    1                       sbtest.sbtest25.c:asc
        └─HashAgg_12            root    1                       group by:sbtest.sbtest25.c,
            ↪ funcs:firstrow(sbtest.sbtest25.c)->sbtest.sbtest25.c
          └─TableReader_13      root    1                       data:HashAgg_8
            └─HashAgg_8         cop     1                       group by:sbtest.sbtest25.c,
              └─TableScan_11    cop     1.2440069558121831      table:sbtest25, range
                  ↪ :[472745,472844], keep order:false
```

查询集群各个 TIDB 节点的慢查询数量

```
select instance, count(*) from information_schema.cluster_slow_query where time >= "2020-03-06
    ↪ 00:00:00" and time < now() group by instance;
```

输出样例：

```
+---------------+----------+
| instance      | count(*) |
+---------------+----------+
```

```
| 0.0.0.0:10081 | 124      |
| 0.0.0.0:10080 | 119771   |
+---------------+----------+
```

查询仅出现在异常时间段的慢日志

假如发现 2020-03-10 13:24:00 ~ 2020-03-10 13:27:00 的 QPS 降低或者延迟上升等问题，可能是由于突然出现大查询导致的，可以用下面 SQL 查询仅出现在异常时间段的慢日志，其中 2020-03-10 13:20:00 ~ 2020-03-10 13:23:00 为正常时间段。

```
SELECT * FROM
    (SELECT /*+ AGG_TO_COP(), HASH_AGG() */ count(*),
         min(time),
         sum(query_time) AS sum_query_time,
         sum(Process_time) AS sum_process_time,
         sum(Wait_time) AS sum_wait_time,
         sum(Commit_time),
         sum(Request_count),
         sum(process_keys),
         sum(Write_keys),
         max(Cop_proc_max),
         min(query),min(prev_stmt),
         digest
    FROM information_schema.CLUSTER_SLOW_QUERY
    WHERE time >= '2020-03-10 13:24:00'
            AND time < '2020-03-10 13:27:00'
            AND Is_internal = false
    GROUP BY  digest) AS t1
WHERE t1.digest NOT IN
    (SELECT /*+ AGG_TO_COP(), HASH_AGG() */ digest
    FROM information_schema.CLUSTER_SLOW_QUERY
    WHERE time >= '2020-03-10 13:20:00'
            AND time < '2020-03-10 13:23:00'
    GROUP BY  digest)
ORDER BY  t1.sum_query_time DESC limit 10\G
```

输出样例：

```
***************************[ 1. row ]***************************
count(*)           | 200
min(time)          | 2020-03-10 13:24:27.216186
sum_query_time     | 50.114126194
sum_process_time   | 268.351
sum_wait_time      | 8.476
sum(Commit_time)   | 1.044304306
sum(Request_count) | 6077
sum(process_keys)  | 202871950
```

```
sum(Write_keys)     | 319500
max(Cop_proc_max)   | 0.263
min(query)          | delete from test.tcs2 limit 5000;
min(prev_stmt)      |
digest              | 24bd6d8a9b238086c9b8c3d240ad4ef32f79ce94cf5a468c0b8fe1eb5f8d03df
```

10.2.1.1.6 解析其他的 TiDB 慢日志文件

TiDB 通过 session 变量 `tidb_slow_query_file` 控制查询 `INFORMATION_SCHEMA.SLOW_QUERY` 时要读取和解析的文件，可通过修改改 session 变量的值来查询其他慢查询日志文件的内容：

```
set tidb_slow_query_file = "/path-to-log/tidb-slow.log"
```

10.2.1.1.7 用 pt-query-digest 工具分析 TiDB 慢日志

可以用 `pt-query-digest` 工具分析 TiDB 慢日志。

> **注意：**
>
> 建议使用 pt-query-digest 3.0.13 及以上版本。

示例如下：

```
pt-query-digest --report tidb-slow.log
```

输出样例：

```
#### 320ms user time, 20ms system time, 27.00M rss, 221.32M vsz
#### Current date: Mon Mar 18 13:18:51 2019
#### Hostname: localhost.localdomain
#### Files: tidb-slow.log
#### Overall: 1.02k total, 21 unique, 0 QPS, 0x concurrency _____________________
#### Time range: 2019-03-18-12:22:16 to 2019-03-18-13:08:52
#### Attribute          total     min     max     avg     95%  stddev  median
#### ============     ======= ======= ======= ======= ======= ======= =======
#### Exec time           218s    10ms     13s   213ms    30ms       1s    19ms
#### Query size       175.37k       9   2.01k  175.89  158.58  122.36  158.58
#### Commit time         46ms     2ms     7ms     3ms     7ms     1ms     3ms
#### Conn ID               71       1      16    8.88   15.25    4.06    9.83
#### Process keys     581.87k       2 103.15k  596.43  400.73   3.91k  400.73
#### Process time         31s     1ms     10s    32ms    19ms   334ms    16ms
#### Request coun       1.97k       1      10    2.02    1.96    0.33    1.96
#### Total keys       636.43k       2 103.16k  652.35  793.42   3.97k  400.73
#### Txn start ts     374.38E       0  16.00E 375.48P   1.25P  89.05T   1.25P
```

```
#### Wait time          943ms     1ms     19ms     1ms     2ms     1ms    972us
.
.
.
```

定位问题语句的方法

并不是所有 SLOW_QUERY 的语句都是有问题的。会造成集群整体压力增大的，是那些 process_time 很大的语句。wait_time 很大，但 process_time 很小的语句通常不是问题语句，是因为被问题语句阻塞，在执行队列等待造成的响应时间过长。

10.2.1.1.8 ADMIN SHOW SLOW 命令

除了获取 TiDB 日志，还有一种定位慢查询的方式是通过 ADMIN SHOW SLOW SQL 命令：

```
ADMIN SHOW SLOW recent N;
```

```
ADMIN SHOW SLOW TOP [internal | all] N;
```

recent N 会显示最近的 N 条慢查询记录，例如：

```
ADMIN SHOW SLOW recent 10;
```

top N 则显示最近一段时间（大约几天）内，最慢的查询记录。如果指定 internal 选项，则返回查询系统内部 SQL 的慢查询记录；如果指定 all 选项，返回系统内部和用户 SQL 汇总以后的慢查询记录；默认只返回用户 SQL 中的慢查询记录。

```
ADMIN SHOW SLOW TOP 3;
ADMIN SHOW SLOW TOP internal 3;
ADMIN SHOW SLOW TOP all 5;
```

由于内存限制，保留的慢查询记录的条数是有限的。当命令查询的 N 大于记录条数时，返回的结果记录条数会小于 N。

输出内容详细说明，如下：

列名	描述
start	SQL 语句执行开始时间
duration	SQL 语句执行持续时间
details	执行语句的详细信息
succ	SQL 语句执行是否成功，1：成功，0：失败
conn_id	session 连接 ID
transaction_ts	事务提交的 commit ts
user	执行该语句的用户名
db	执行该 SQL 涉及到 database
table_ids	执行该 SQL 涉及到表的 ID
index_ids	执行该 SQL 涉及到索引 ID
internal	表示为 TiDB 内部的 SQL 语句

列名	描述
digest	表示 SQL 语句的指纹
sql	执行的 SQL 语句

10.2.1.2 分析慢查询

处理慢查询分为两步：

1. 从大量查询中定位出哪一类查询比较慢
2. 分析这类慢查询的原因

第一步可以通过慢日志、statement-summary 方便地定位，推荐直接使用TiDB Dashboard，它整合了这两个功能，且能方便直观地在浏览器中展示出来。本文聚焦第二步。

首先将慢查询归因成两大类：

- 优化器问题：如选错索引，选错 Join 类型或顺序。
- 系统性问题：将非优化器问题都归结于此类。如：某个 TiKV 实例忙导致处理请求慢，Region 信息过期导致查询变慢。

实际中，优化器问题可能造成系统性问题。例如对于某类查询，优化器应使用索引，但却使用了全表扫。这可能导致这类 SQL 消耗大量资源，造成某些 KV 实例 CPU 飚高等问题。表现上看就是一个系统性问题，但本质是优化器问题。

分析优化器问题需要有判断执行计划是否合理的能力，而系统性问题的定位相对简单，因此面对慢查询推荐的分析过程如下：

1. 定位查询瓶颈：即查询过程中耗时多的部分
2. 分析系统性问题：根据瓶颈点，结合当时的监控/日志等信息，分析可能的原因
3. 分析优化器问题：分析是否有更好的执行计划

接下来会分别介绍上面几点。

10.2.1.2.1 定位查询瓶颈

定位查询瓶颈需要对查询过程有一个大致理解，TiDB 处理查询过程的关键阶段都在 performance-map 图中了。

查询的耗时信息可以从下面几种方式获得：

- 慢日志（推荐直接在TiDB Dashboard 中查看）
- `explain analyze` 语句

他们的侧重点不同：

- 慢日志记录了 SQL 从解析到返回，几乎所有阶段的耗时，较为全面（在 TiDB Dashboard 中可以直观地查询和分析慢日志）;

- explain analyze 可以拿到 SQL 实际执行中每个执行算子的耗时，对执行耗时有更细分的统计；

总的来说，利用慢日志和 explain analyze 可以比较准确地定位查询的瓶颈点，帮助你判断这条 SQL 慢在哪个模块 (TiDB/TiKV)，慢在哪个阶段，下面会有一些例子。

另外在 4.0.3 之后，慢日志中的 Plan 字段也会包含 SQL 的执行信息，也就是 explain analyze 的结果，这样一来 SQL 的所有耗时信息都可以在慢日志中找到。

10.2.1.2.2 分析系统性问题

对于系统性问题，我们根据执行阶段，分成三个大类：

1. TiKV 处理慢：如 coprocessor 处理数据慢
2. TiDB 执行慢：主要指执行阶段，如某个 Join 算子处理数据慢
3. 其他关键阶段慢：如取时间戳慢

拿到一个慢查询，我们应该先根据已有信息判断大致是哪个大类，再具体分析。

TiKV 处理慢

如果是 TiKV 处理慢，可以很明显的通过 explain analyze 中看出来。例如下面这个例子，可以看到 StreamAgg_8 和 TableFullScan_15 这两个 tikv-task（在 task 列可以看出这两个任务类型是 cop[tikv]）花费了 170ms，而 TiDB 部分的算子耗时，减去这 170ms 后，耗时占比非常小，说明瓶颈在 TiKV。

```
+--
    ↪ -------------------------+---------+---------+-----------+---------------+--------------------------------
    ↪
| id                        | estRows | actRows | task      | access object | execution info
    ↪                                                                  | operator info
    ↪                    | memory    | disk |
+--
    ↪ -------------------------+---------+---------+-----------+---------------+--------------------------------
    ↪
| StreamAgg_16              | 1.00    | 1       | root      |               | time:170.08572ms,
    ↪ loops:2                                                        | funcs:count(Column#5)->
    ↪ Column#3 | 372 Bytes | N/A  |
| └─TableReader_17          | 1.00    | 1       | root      |               | time:170.080369ms
    ↪ , loops:2, rpc num: 1, rpc time:17.023347ms, proc keys:28672 | data:StreamAgg_8
    ↪                 | 202 Bytes | N/A  |
|   └─StreamAgg_8           | 1.00    | 1       | cop[tikv] |               | time:170ms, loops
    ↪ :29                                                            | funcs:count(1)->Column#5
    ↪        | N/A       | N/A  |
|     └─TableFullScan_15    | 7.00    | 28672   | cop[tikv] | table:t       | time:170ms, loops
    ↪ :29                                                            | keep order:false, stats:
    ↪ pseudo  | N/A       | N/A  |
+--
    ↪ -------------------------+---------+---------+-----------+---------------+--------------------------------
    ↪
```

另外在慢日志中，Cop_process 和 Cop_wait 字段也可以帮助判断，如下面这个例子，查询整个耗时是 180.85ms 左右，而最大的那个 coptask 就消耗了 171ms，可以说明对这个查询而言，瓶颈在 TiKV 侧。

慢日志中的各个字段的说明可以参考慢查询日志中的字段含义说明

```
#### Query_time: 0.18085
...
#### Num_cop_tasks: 1
#### Cop_process: Avg_time: 170ms P90_time: 170ms Max_time: 170ms Max_addr: 10.6.131.78
#### Cop_wait: Avg_time: 1ms P90_time: 1ms Max_time: 1ms Max_Addr: 10.6.131.78
```

根据上述方式判断是 TiKV 慢后，可以依次排查 TiKV 慢的原因。

TiKV 实例忙

一条 SQL 可能会去从多个 TiKV 上拿数据，如果某个 TiKV 响应慢，可能拖慢整个 SQL 的处理速度。

慢日志中的 Cop_wait 可以帮忙判断这个问题：

```
#### Cop_wait: Avg_time: 1ms P90_time: 2ms Max_time: 110ms Max_Addr: 10.6.131.78
```

如上图，发给 10.6.131.78 的一个 cop-task 等待了 110ms 才被执行，可以判断是当时该实例忙，此时可以打开当时的 CPU 监控辅助判断。

过期 key 多

如果 TiKV 上过期的数据比较多，在扫描的时候则需要处理这些不必要的数据，影响处理速度。

这可以通过 Total_keys 和 Processed_keys 判断，如果两者相差较大，则说明旧版本的 key 太多：

```
...
#### Total_keys: 2215187529 Processed_keys: 1108056368
...
```

其他关键阶段慢

取 TS 慢

可以对比慢日志中的 Wait_TS 和 Query_time，因为 TS 有预取操作，通常来说 Wait_TS 应该很低。

```
#### Query_time: 0.0300000
...
#### Wait_TS: 0.02500000
```

Region 信息过期

TiDB 侧 Region 信息可能过期，此时 TiKV 可能返回 regionMiss 的错误，然后 TiDB 会从 PD 去重新获取 Region 信息，这些信息会被反应在 Cop_backoff 信息内，失败的次数和总耗时都会被记录下来。

```
#### Cop_backoff_regionMiss_total_times: 200 Cop_backoff_regionMiss_total_time: 0.2
    ↪ Cop_backoff_regionMiss_max_time: 0.2 Cop_backoff_regionMiss_max_addr: 127.0.0.1
    ↪ Cop_backoff_regionMiss_avg_time: 0.2 Cop_backoff_regionMiss_p90_time: 0.2
#### Cop_backoff_rpcPD_total_times: 200 Cop_backoff_rpcPD_total_time: 0.2
    ↪ Cop_backoff_rpcPD_max_time: 0.2 Cop_backoff_rpcPD_max_addr: 127.0.0.1
    ↪ Cop_backoff_rpcPD_avg_time: 0.2 Cop_backoff_rpcPD_p90_time: 0.2
```

子查询被提前执行

对于带有非关联子查询的语句，子查询部分可能被提前执行，如：`select * from t1 where a = (select max(a)from t2)`，`select max(a)from t2` 部分可能在优化阶段被提前执行。这种查询用 `explain analyze` 看不到对应的耗时，如下：

```
mysql> explain analyze select count(*) from t where a=(select max(t1.a) from t t1, t t2 where t1.a=t2.a);
+----------------------------+----------+---------+-----------+---------------+--------------------------------+-----------------------+------+
| id                         | estRows  | actRows | task      | access object | execution info                 | operator info         | memory    | disk |
+----------------------------+----------+---------+-----------+---------------+--------------------------------+-----------------------+-----------+------+
| StreamAgg_59               | 1.00     | 1       | root      |               | time:4.69267ms, loops:2        | funcs:count(Column#10)->Column#8 | 372 Bytes | N/A  |
| └─TableReader_60           | 1.00     | 1       | root      |               | time:4.690428ms, loops:2       | data:StreamAgg_48     | 141 Bytes | N/A  |
|   └─StreamAgg_48           | 1.00     |         | cop[tikv] |               | time:0ns, loops:0              | funcs:count(1)->Column#10 | N/A       | N/A  |
|     └─Selection_58         | 16384.00 |         | cop[tikv] |               | time:0ns, loops:0              | eq(test.t.a, 1)       | N/A       | N/A  |
|       └─TableFullScan_57   | 16384.00 | -1      | cop[tikv] | table:t       | time:0s, loops:0               | keep order:false      | N/A       | N/A  |
+----------------------------+----------+---------+-----------+---------------+--------------------------------+-----------------------+-----------+------+
5 rows in set (7.77 sec)
```

不过可以从慢日志中排查这种情况：

```
#### Query_time: 7.770634843
...
#### Rewrite_time: 7.765673663 Preproc_subqueries: 1 Preproc_subqueries_time: 7.765231874
```

可以看到有 1 个子查询被提前执行，花费了 7.76s。

TiDB 执行慢

这里我们假设 TiDB 的执行计划正确（不正确的情况在分析优化器问题这一节中说明），但是执行上很慢；

解决这类问题主要靠调整参数或利用 hint，并结合 `explain analyze` 对 SQL 进行调整。

并发太低

如果发现瓶颈在有并发的算子上，可以通过调整并发度来尝试提速，如下面的执行计划中：

```
mysql> explain analyze select sum(t1.a) from t t1, t t2 where t1.a=t2.a;
```

```
+--
    ↪ -------------------------------+--------------+-----------+-----------+---------------+-----------------
    ↪
| id                              | estRows      | actRows   | task      | access object |
    ↪ execution info                                                                  |
    ↪ operator info                                  | memory           | disk    |
+--
    ↪ -------------------------------+--------------+-----------+-----------+---------------+-----------------
    ↪
| HashAgg_11                      | 1.00         | 1         | root      |               | time
    ↪ :9.666832189s, loops:2, PartialConcurrency:4, FinalConcurrency:4                | funcs:
    ↪ sum(Column#6)->Column#5                  | 322.125 KB       | N/A     |
| └─Projection_24                 | 268435456.00 | 268435456 | root      |               | time
    ↪ :9.098644711s, loops:262145, Concurrency:4                                      | cast(
    ↪ test.t.a, decimal(65,0) BINARY)->Column#6 | 199 KB           | N/A     |
|   └─HashJoin_14                 | 268435456.00 | 268435456 | root      |               | time
    ↪ :6.616773501s, loops:262145, Concurrency:5, probe collision:0, build:881.404µs  | inner
    ↪ join, equal:[eq(test.t.a, test.t.a)]     | 131.75 KB        | 0 Bytes |
|     ├─TableReader_21(Build)     | 16384.00     | 16384     | root      |               | time
    ↪ :6.553717ms, loops:17                                                           | data:
    ↪ Selection_20                             | 33.6318359375 KB | N/A     |
|     │ └─Selection_20            | 16384.00     |           | cop[tikv] |               | time
    ↪ :0ns, loops:0                                                                   | not(
    ↪ isnull(test.t.a))                         | N/A              | N/A     |
|     │   └─TableFullScan_19      | 16384.00     | -1        | cop[tikv] | table:t2      | time
    ↪ :0s, loops:0                                                                    | keep
    ↪ order:false                              | N/A              | N/A     |
|     └─TableReader_18(Probe)     | 16384.00     | 16384     | root      |               | time
    ↪ :6.880923ms, loops:17                                                           | data:
    ↪ Selection_17                             | 33.6318359375 KB | N/A     |
|       └─Selection_17            | 16384.00     |           | cop[tikv] |               | time
    ↪ :0ns, loops:0                                                                   | not(
    ↪ isnull(test.t.a))                         | N/A              | N/A     |
|         └─TableFullScan_16      | 16384.00     | -1        | cop[tikv] | table:t1      | time
    ↪ :0s, loops:0                                                                    | keep
    ↪ order:false                              | N/A              | N/A     |
+--
    ↪ -------------------------------+--------------+-----------+-----------+---------------+-----------------
    ↪
9 rows in set (9.67 sec)
```

发现耗时主要在 HashJoin_14 和 Projection_24，可以酌情通过 SQL 变量来提高他们的并发度进行提速。

system-variables 中有所有的系统变量，如想提高 HashJoin_14 的并发度，则可以修改变量 tidb_hash_join_concurrency ↪ 。

产生了落盘

执行慢的另一个原因是执行过程中，因为到达内存限制，产生了落盘，这点在执行计划和慢日志中都能看到：

```
+--
    ↪ ----------------------+----------+---------+----------+--------------+------------------------------+
    ↪
| id                    | estRows   | actRows | task      | access object | execution info
    ↪               | operator info        | memory                | disk           |
+--
    ↪ ----------------------+----------+---------+----------+--------------+------------------------------+
    ↪
| Sort_4                | 462144.00 | 462144  | root      |               | time:2.02848898s,
    ↪ loops:453  | test.t.a             | 149.68795776367188 MB | 219.3203125 MB |
| └─TableReader_8         | 462144.00 | 462144  | root      |               | time:616.211272ms,
    ↪  loops:453 | data:TableFullScan_7 | 197.49601364135742 MB | N/A            |
|   └─TableFullScan_7     | 462144.00 | -1      | cop[tikv] | table:t       | time:0s, loops:0
    ↪             | keep order:false     | N/A                   | N/A            |
+--
    ↪ ----------------------+----------+---------+----------+--------------+------------------------------+
    ↪
```

```
...
#### Disk_max: 229974016
...
```

做了笛卡尔积的 Join

做笛卡尔积的 Join 会产生 左边孩子行数 * 右边孩子行数 这么多数据，效率较低，应该尽量避免；

目前对于产生笛卡尔积的 Join 会在执行计划中显示的标明 CARTESIAN，如下：

```
mysql> explain select * from t t1, t t2 where t1.a>t2.a;
+--
    ↪ ---------------------------+-------------+-----------+--------------+--------------------------------
    ↪
| id                           | estRows     | task      | access object | operator info
    ↪                                    |
+--
    ↪ ---------------------------+-------------+-----------+--------------+--------------------------------
    ↪
| HashJoin_8                   | 99800100.00 | root      |               | CARTESIAN inner join,
    ↪ other cond:gt(test.t.a, test.t.a) |
| ├─TableReader_15(Build)      | 9990.00     | root      |               | data:Selection_14
    ↪                                   |
| │ └─Selection_14             | 9990.00     | cop[tikv] |               | not(isnull(test.t.a))
    ↪                                  |
```

```
| |   └─TableFullScan_13       | 10000.00    | cop[tikv] | table:t2      | keep order:false,
    ↪ stats:pseudo                          |
| └─TableReader_12(Probe)      | 9990.00     | root      |               | data:Selection_11
    ↪                                       |
|   └─Selection_11             | 9990.00     | cop[tikv] |               | not(isnull(test.t.a))
    ↪                                  |
|     └─TableFullScan_10       | 10000.00    | cop[tikv] | table:t1      | keep order:false,
    ↪ stats:pseudo                          |
+--
    ↪ ---------------------------+-------------+-----------+---------------+--------------------------------
    ↪
```

10.2.1.2.3 分析优化器问题

分析优化问题需要有判断执行计划是否合理的能力，这需要对优化过程和各算子有一定了解。

下面是一组例子，假设表结构为 `create table t (id int, a int, b int, c int, primary key(id), key(a)
↪ , key(b, c))`：

1. `select * from t`：没有过滤条件，会扫全表，所以会用 `TableFullScan` 算子读取数据；
2. `select a from t where a=2`：有过滤条件且只读索引列，所以会用 `IndexReader` 算子读取数据；
3. `select * from t where a=2`：在 `a` 有过滤条件，但索引 `a` 不能完全覆盖需要读取的内容，因此会采用 `IndexLookup`；
4. `select b from t where c=3`：多列索引没有前缀条件就用不上，所以会用 `IndexFullScan`；
5. …

上面举例了数据读入相关的算子，在理解 TiDB 执行计划中描述了更多算子的情况。

另外阅读SQL 性能调优整个小节能增加你对 TiDB 优化器的了解，帮助判断执行计划是否合理。

由于大多数优化器问题在SQL 性能调优已经有解释，这里就直接列举出来跳转过去：

1. 索引选择错误
2. Join 顺序错误
3. 表达式未下推

10.2.2 TiDB OOM 故障排查

本文总结了 TiDB Out of Memory (OOM) 常见问题的解决思路、故障现象、故障原因、解决方法，以及需要收集的诊断信息。在遇到 OOM 问题时，你可以参考本文档来排查错误原因并进行处理。

10.2.2.1 常见故障现象

OOM 常见的故障现象包括（但不限于）：

- 客户端报错：`SQL error, errno = 2013, state = 'HY000': Lost connection to MySQL server during
  ↪ query`

- 查看 Grafana 监控，发现以下现象：
  - TiDB > Server > Memory Usage 显示 process/heapInUse 持续升高，达到阈值后掉零
  - TiDB > Server > Uptime 显示为掉零
  - TiDB-Runtime > Memory Usage 显示 estimate-inuse 持续升高
- 查看 `tidb.log`，可发现如下日志条目：
  - OOM 相关的 Alarm：`[WARN] [memory_usage_alarm.go:139] ["tidb-server has the risk of OOM. Running SQLs and heap profile will be recorded in record path"]`。关于该日志的详细说明，请参考memory-usage-alarm-ratio。
  - 重启相关的日志条目：`[INFO] [printer.go:33] ["Welcome to TiDB."]`。

10.2.2.2 整体排查思路

在排查 OOM 问题时，整体遵循以下排查思路：

1. 确认是否属于 OOM 问题。

   执行以下命令查看操作系统日志。如果故障发生的时间点附近存在 `oom-killer` 的日志，则可以确定是 OOM 问题。

   ```
   dmesg -T | grep tidb-server
   ```

   下面是包含 `oom-killer` 的日志输出示例：

   ```
   ......
   Mar 14 16:55:03 localhost kernel: tidb-server invoked oom-killer: gfp_mask=0x201da, order=0,
       ↪  oom_score_adj=0
   Mar 14 16:55:03 localhost kernel: tidb-server cpuset=/ mems_allowed=0
   Mar 14 16:55:03 localhost kernel: CPU: 14 PID: 21966 Comm: tidb-server Kdump: loaded Not
       ↪ tainted 3.10.0-1160.el7.x86_64 #1
   Mar 14 16:55:03 localhost kernel: Hardware name: QEMU Standard PC (i440FX + PIIX, 1996),
       ↪ BIOS rel-1.14.0-0-g155821a1990b-prebuilt.qemu.org 04/01/2014
   ......
   Mar 14 16:55:03 localhost kernel: Out of memory: Kill process 21945 (tidb-server) score 956
       ↪ or sacrifice child
   Mar 14 16:55:03 localhost kernel: Killed process 21945 (tidb-server), UID 1000, total-vm
       ↪ :33027492kB, anon-rss:31303276kB, file-rss:0kB, shmem-rss:0kB
   Mar 14 16:55:07 localhost systemd: tidb-4000.service: main process exited, code=killed,
       ↪ status=9/KILL
   ......
   ```

2. 确认是 OOM 问题之后，可以进一步排查触发 OOM 的原因是部署问题还是数据库问题。
   - 如果是部署问题触发 OOM，需要排查资源配置、混合部署的影响。
   - 如果是数据库问题触发 OOM，常见原因有：
     - TiDB 处理较大的数据流量，如：大查询、大写入、数据导入等。

  - TiDB 的高并发场景，多条 SQL 并发消耗资源，或者算子并发高。
  - TiDB 内存泄露，资源没有释放。

具体排查方法请参考下面的章节。

10.2.2.3 常见故障原因和解决方法

根据 OOM 出现的原因，一般可以分为以下几种情况：

- 部署问题
- 数据库问题
- 客户端问题

10.2.2.3.1 部署问题

如果是由于部署不当导致的 OOM 问题，常见的原因有：

- 操作系统内存容量规划偏小，导致内存不足。
- TiUP `resource_control` 配置不合理。
- 在混合部署的情况下（指 TiDB 和其他应用程序部署在同一台服务器上），其他应用程序抢占资源导致 TiDB 被 `oom-killer` 关闭。

10.2.2.3.2 数据库问题

本节介绍由数据库问题导致的 OOM 问题和解决办法。

> 注意：
>
> 如果 SQL 返回 `ERROR 1105 (HY000): Out Of Memory Quota![conn_id=54]`，是由于配置了`tidb_mem_quota_query` 导致，数据库的内存使用控制行为会触发该报错。此报错为正常行为。

执行 SQL 语句时消耗太多内存

可以根据以下不同的触发 OOM 的原因，采取相应的措施减少 SQL 的内存使用：

- 如果 SQL 的执行计划不优，比如由于缺少合适的索引、统计信息过期、优化器 bug 等原因，会导致选错 SQL 的执行计划，进而出现巨大的中间结果集累积在内存中。这种情况下可以考虑采取以下措施：
  - 添加合适的索引
  - 使用算子的数据落盘功能
  - 调整表之间的 JOIN 顺序
  - 使用 hint 进行调优
- 一些算子和函数不支持下推到存储层，导致出现巨大的中间结果集累积。此时可能需要改写业务 SQL，或使用 hint 进行调优，来使用可下推的函数或算子。

- 执行计划中存在算子 HashAgg。HashAgg 是多线程并发执行，虽然执行速度较快，但会消耗较多内存。可以尝试使用 STREAM_AGG() 替代。
- 调小同时读取的 Region 的数量，或降低算子并发度，以避免因高并发导致的内存问题。对应的系统变量包括：
  - tidb_distsql_scan_concurrency
  - tidb_index_serial_scan_concurrency
  - tidb_executor_concurrency
- 问题发生时间附近，session 的并发度过高，此时可能需要添加节点进行扩容。

大事务或大写入消耗太多内存

需要提前进行内存的容量规划，这是因为执行事务时 TiDB 进程的内存消耗相对于事务大小会存在一定程度的放大，最大可能达到提交事务大小的 2 到 3 倍以上。

针对单个大事务，可以通过拆分的方式调小事务大小。

收集和加载统计信息的过程中消耗太多内存

TiDB 节点启动后需要加载统计信息到内存中。统计信息的收集过程会消耗内存，可以通过以下方式控制内存使用量：

- 使用指定采样率、指定只收集特定列的统计信息、减少 ANALYZE 并发度等手段减少内存使用。
- TiDB v6.1.0 开始引入了系统变量tidb_stats_cache_mem_quota，可以对统计信息的内存使用进行限制。
- TiDB v6.1.0 开始引入了系统变量tidb_mem_quota_analyze，用于控制 TiDB 更新统计信息时的最大总内存占用。

更多信息请参见统计信息简介。

预处理语句 (Prepared Statement) 使用过量

客户端不断创建预处理语句但未执行deallocate prepare stmt 会导致内存持续上涨，最终触发 TiDB OOM。原因是预处理语句占用的内存要在 session 关闭后才会释放。这一点在长连接下尤需注意。

要解决该问题，可以考虑采取以下措施：

- 调整 session 的生命周期。
- 调整连接池的 wait_timeout 和 max_execution_time 时长。
- 使用系统变量max_prepared_stmt_count 进行限制。

系统变量配置不当

系统变量tidb_enable_rate_limit_action 在单条查询仅涉及读数据的情况下，对内存控制效果较好。若还存在额外的计算操作（如连接、聚合等），启动该变量可能会导致内存不受tidb_mem_quota_query 控制，加剧 OOM 风险。

建议关闭该变量。从 TiDB v6.3.0 开始，该变量默认关闭。

10.2.2.3.3 客户端问题

若客户端发生 OOM，则需要排查以下方面：

- 观察 Grafana TiDB Details > Server > Client Data Traffic 的趋势和速度，查看是否存在网络阻塞。
- 检查是否存在错误的 JDBC 配置参数导致的应用 OOM。例如流式读取的相关参数 `defaultFetchSize` 配置有误，会造成数据在客户端大量缓存。

10.2.2.4 处理 OOM 问题需要收集的诊断信息

为定位 OOM 故障，通常需要收集以下信息：

- 操作系统的内存相关配置：
  - TiUP 上的配置：`resource_control.memory_limit`
  - 操作系统的配置：
    * 内存信息：`cat /proc/meminfo`
    * 相关内核参数：`vm.overcommit_memory`
  - NUMA 相关信息：
    * `numactl --hardware`
    * `numactl --show`
- 数据库的版本和内存相关配置：
  - TiDB 版本
  - `tidb_mem_quota_query`
  - `memory-usage-alarm-ratio`
  - `mem-quota-query`
  - `oom-action`
  - `tidb_enable_rate_limit_action`
  - `tidb_server_memory_limit`
  - `oom-use-tmp-storage`
  - `tmp-storage-path`
  - `tmp-storage-quota`
  - `tidb_analyze_version`
- 在 Grafana 查看 TiDB 内存的日常使用情况：TiDB > Server > Memory Usage
- 查看内存消耗较多的 SQL 语句：
  - 可以从 TiDB Dashboard 中查看 SQL 语句分析、慢查询，查看内存使用量
  - 查看 `INFORMATION_SCHEMA` 中的 `SLOW_QUERY` 和 `CLUSTER_SLOW_QUERY`
  - 各个 TiDB 节点的 `tidb_slow_query.log`
  - 执行 `grep "expensive_query" tidb.log` 查看对应的日志条目
  - 执行 `EXPLAIN ANALYZE` 查看算子的内存消耗
  - 执行 `SELECT * FROM information_schema.processlist;` 查看 SQL 对应的 `MEM` 列的值
- 执行以下命令收集内存使用率高的时候 TiDB 的 Profile 信息：

```
curl -G http://{TiDBIP}:10080/debug/zip?seconds=10" > profile.zip
```

- 执行 `grep "tidb-server has the risk of OOM" tidb.log` 查看 TiDB Server 收集的告警文件路径，例如：

```
["tidb-server has the risk of OOM. Running SQLs and heap profile will be recorded in record
    ↪ path"] ["is tidb_server_memory_limit set"=false] ["system memory total"=14388137984]
    ↪  ["system memory usage"=11897434112] ["tidb-server memory usage"=11223572312] [
    ↪ memory-usage-alarm-ratio=0.8] ["record path"="/tmp/0_tidb/
    ↪ MC4wLjAuMDo0MDAwLzAuMC4wLjA6MTAwODA=/tmp-storage/record"]
```

10.2.2.5 探索更多

- TiDB 内存调优
- TiKV 内存调优

10.2.3 TiDB 热点问题处理

本文介绍如何定位和解决读写热点问题。

TiDB 作为分布式数据库，内建负载均衡机制，尽可能将业务负载均匀地分布到不同计算或存储节点上，更好地利用上整体系统资源。然而，机制不是万能的，在一些场景下仍会有部分业务负载不能被很好地分散，影响性能，形成单点的过高负载，也称为热点。

TiDB 提供了完整的方案用于排查、解决或规避这类热点。通过均衡负载热点，可以提升整体性能，包括提高 QPS 和降低延迟等。

10.2.3.1 常见热点场景

10.2.3.1.1 TiDB 编码规则回顾

TiDB 对每个表分配一个 TableID，每一个索引都会分配一个 IndexID，每一行分配一个 RowID（默认情况下，如果表使用整数型的 Primary Key，那么会用 Primary Key 的值当做 RowID）。其中 TableID 在整个集群内唯一，IndexID/RowID 在表内唯一，这些 ID 都是 int64 类型。

每行数据按照如下规则进行编码成 Key-Value pair：

```
Key: tablePrefix{tableID}_recordPrefixSep{rowID}
Value: [col1, col2, col3, col4]
```

其中 Key 的 `tablePrefix` 和 `recordPrefixSep` 都是特定的字符串常量，用于在 KV 空间内区分其他数据。

对于 Index 数据，会按照如下规则编码成 Key-Value pair：

```
Key: tablePrefix{tableID}_indexPrefixSep{indexID}_indexedColumnsValue
Value: rowID
```

Index 数据还需要考虑 Unique Index 和非 Unique Index 两种情况，对于 Unique Index，可以按照上述编码规则。但是对于非 Unique Index，通过这种编码并不能构造出唯一的 Key，因为同一个 Index 的 `tablePrefix{tableID} ↪ _indexPrefixSep{indexID}` 都一样，可能有多行数据的 ColumnsValue 是一样的，所以对于非 Unique Index 的编码做了一点调整：

```
Key: tablePrefix{tableID}_indexPrefixSep{indexID}_indexedColumnsValue_rowID
Value: null
```

10.2.3.1.2 表热点

从 TiDB 编码规则可知，同一个表的数据会在以表 ID 开头为前缀的一个 range 中，数据的顺序按照 RowID 的值顺序排列。在表 insert 的过程中如果 RowID 的值是递增的，则插入的行只能在末端追加。当 Region 达到一定的大小之后会进行分裂，分裂之后还是只能在 range 范围的末端追加，永远只能在一个 Region 上进行 insert 操作，形成热点。

常见的 increment 类型自增主键就是顺序递增的，默认情况下，在主键为整数型时，会用主键值当做 RowID，此时 RowID 为顺序递增，在大量 insert 时形成表的写入热点。

同时，TiDB 中 RowID 默认也按照自增的方式顺序递增，主键不为整数类型时，同样会遇到写入热点的问题。

此外，当写入或读取数据存在热点时，即出现新建表或分区的写入热点问题和只读场景下周期性读热点问题时，你可以使用表属性控制 Region 合并。具体的热点场景描述和解决方法可以查看使用表属性控制 Region 合并的使用场景。

10.2.3.1.3 索引热点

索引热点与表热点类似，常见的热点场景出现在时间顺序单调递增的字段，或者插入大量重复值的场景。

10.2.3.2 确定存在热点问题

性能问题不一定是热点造成的，也可能存在多个因素共同影响，在排查前需要先确认是否与热点相关。

- 判断写热点依据：打开监控面板 TiKV-Trouble-Shooting 中 Hot Write 面板，观察 Raftstore CPU 监控是否存在个别 TiKV 节点的指标明显高于其他节点的现象。
- 判断读热点依据：打开监控面板 TIKV-Details 中 Thread_CPU，查看 coprocessor cpu 有没有明显的某个 TiKV 特别高。

10.2.3.3 使用 TiDB Dashboard 定位热点表

TiDB Dashboard 中的流量可视化功能可帮助用户缩小热点排查范围到表级别。以下是流量可视化功能展示的一个热力图样例，该图横坐标是时间，纵坐标是各个表和索引，颜色越亮代表其流量越大。可在工具栏中切换显示读或写流量。

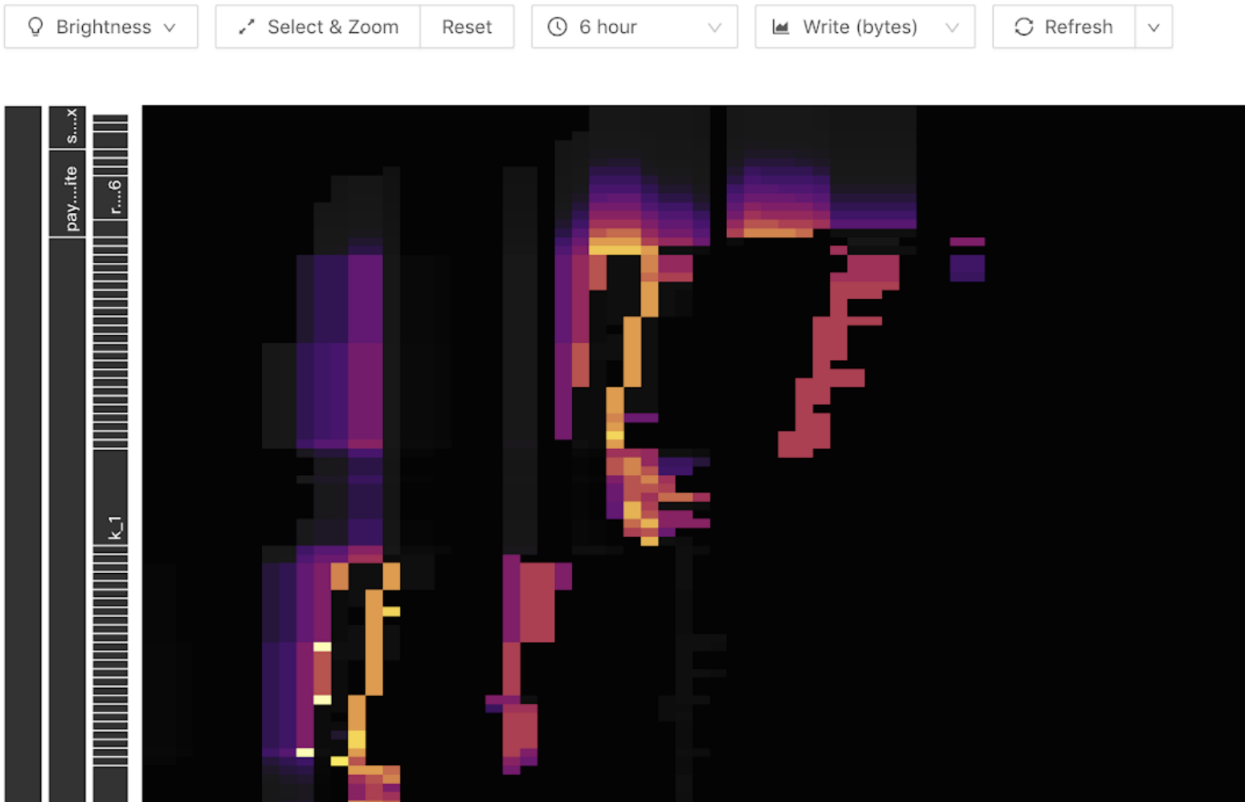


图 71: Dashboard 示例 1

当图中写入流量图出现以下明亮斜线（斜向上或斜向下）时，由于写入只出现在末端，随着表 Region 数量变多，呈现出阶梯状。此时说明该表构成了写入热点：

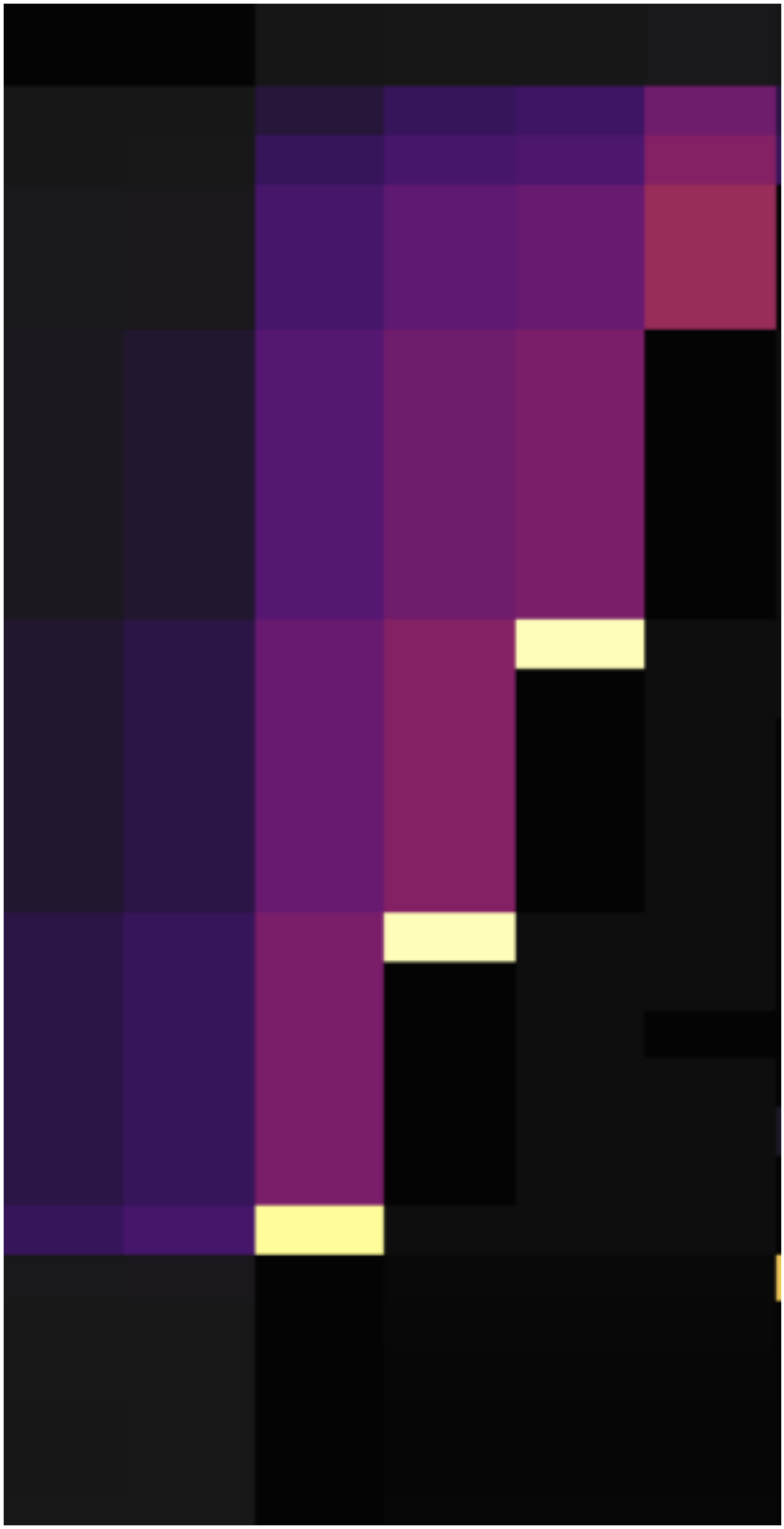


图 72: Dashboard 示例 2

对于读热点，在热力图中一般表现为一条明亮的横线，通常是有大量访问的小表，如下图所示：

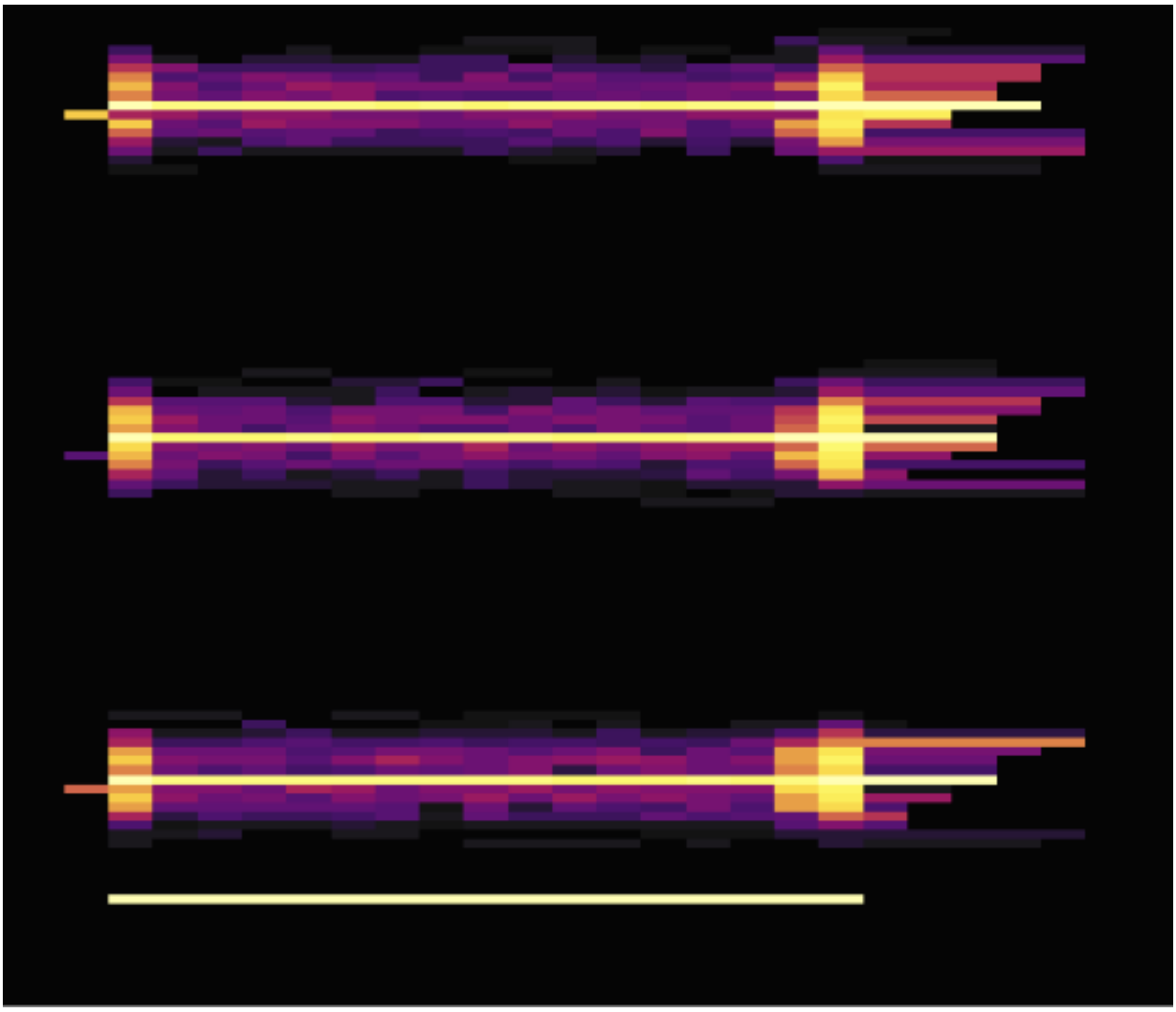


图 73: Dashboard 示例 3

将鼠标移到亮色块上，即可看到是什么表或索引具有大流量，如下图所示：

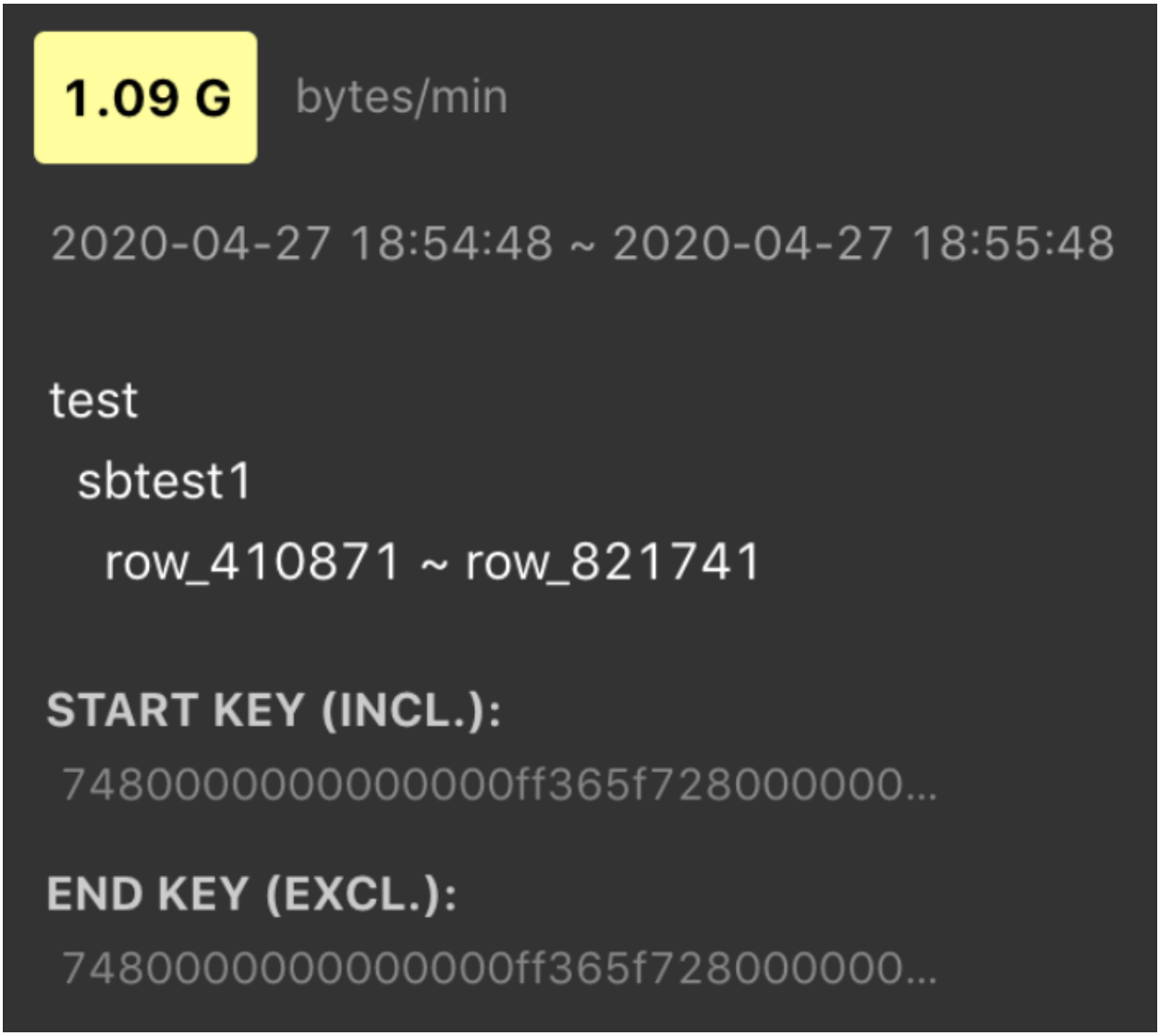


图 74: Dashboard 示例 4

10.2.3.4 使用 SHARD_ROW_ID_BITS 处理热点表

对于非聚簇索引主键或没有主键的表，TiDB 会使用一个隐式的自增 RowID，大量 INSERT 时会把数据集中写入单个 Region，造成写入热点。

通过设置SHARD_ROW_ID_BITS，可以把 RowID 打散写入多个不同的 Region，缓解写入热点问题。

```
SHARD_ROW_ID_BITS = 4 表示 16 个分片
SHARD_ROW_ID_BITS = 6 表示 64 个分片
SHARD_ROW_ID_BITS = 0 表示默认值 1 个分片
```

语句示例：

```
CREATE TABLE: CREATE TABLE t (c int) SHARD_ROW_ID_BITS = 4;
```

```
ALTER TABLE: ALTER TABLE t SHARD_ROW_ID_BITS = 4;
```

SHARD_ROW_ID_BITS 的值可以动态修改，每次修改之后，只对新写入的数据生效。

对于含有 CLUSTERED 主键的表，TiDB 会使用表的主键作为 RowID，因为 SHARD_ROW_ID_BITS 会改变 RowID 生成规则，所以此时无法使用 SHARD_ROW_ID_BITS 选项。而对于使用 NONCLUSTERED 主键的表，TiDB 会使用自动分配的 64 位整数作为 RowID，此时也可以使用 SHARD_ROW_ID_BITS 特性。要了解关于 CLUSTERED 主键的详细信息，请参考聚簇索引。

以下是两张无主键情况下使用 SHARD_ROW_ID_BITS 打散热点后的流量图，第一张展示了打散前的情况，第二张展示了打散后的情况。

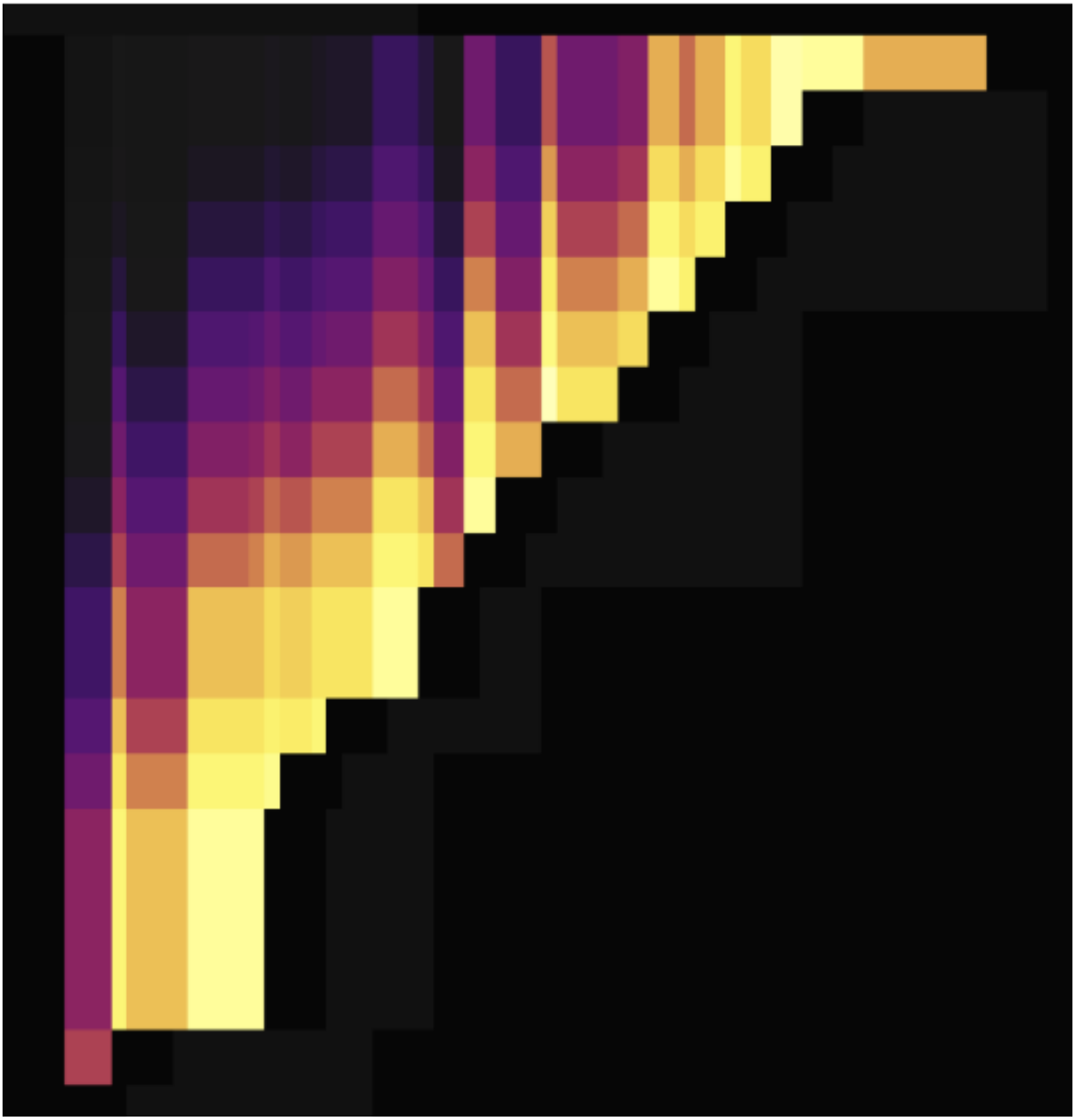


图 75: Dashboard 示例 5

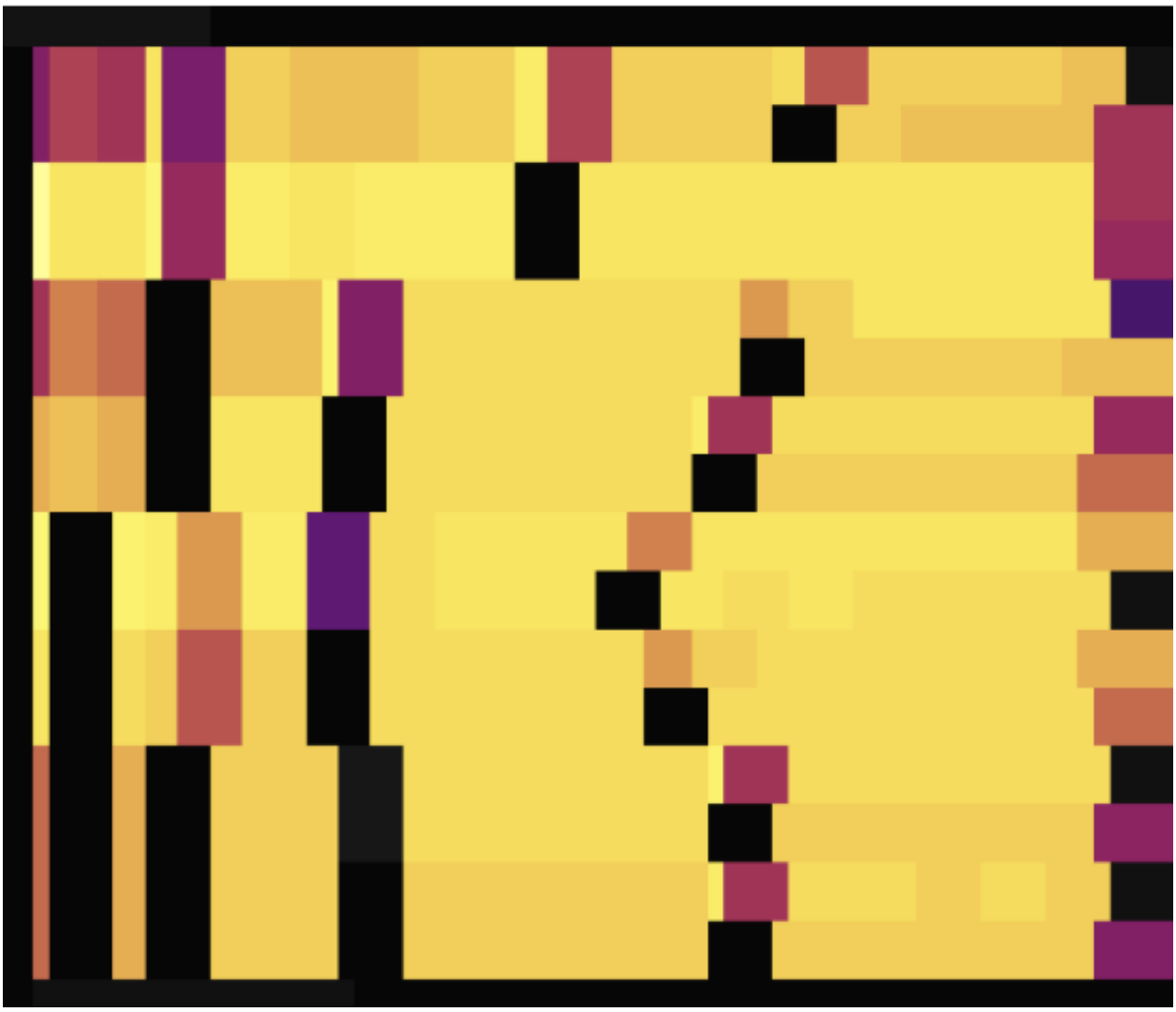


图 76: Dashboard 示例 6

从流量图可见，设置 SHARD_ROW_ID_BITS 后，流量热点由之前的只在一个 Region 上变得很分散。

10.2.3.5 使用 AUTO_RANDOM 处理自增主键热点表

使用 AUTO_RANDOM 处理自增主键热点表，适用于代替自增主键，解决自增主键带来的写入热点。

使用该功能后，将由 TiDB 生成随机分布且空间耗尽前不重复的主键，达到离散写入、打散写入热点的目的。

注意 TiDB 生成的主键不再是自增的主键，可使用 LAST_INSERT_ID() 获取上次分配的主键值。

将建表语句中的 AUTO_INCREMENT 改为 AUTO_RANDOM 即可使用该功能，适用于主键只需要保证唯一，不包含业务意义的场景。示例如下：

```
CREATE TABLE t (a BIGINT PRIMARY KEY AUTO_RANDOM, b varchar(255));
INSERT INTO t (b) VALUES ("foo");
SELECT * FROM t;
```

```
+------------+---+
| a          | b |
+------------+---+
| 1073741825 | b |
+------------+---+
```

```
SELECT LAST_INSERT_ID();
```

```
+------------------+
| LAST_INSERT_ID() |
+------------------+
| 1073741825       |
+------------------+
```

以下是将 AUTO_INCREMENT 表改为 AUTO_RANDOM 打散热点后的流量图，第一张是 AUTO_INCREMENT，第二张是 AUTO_RANDOM。

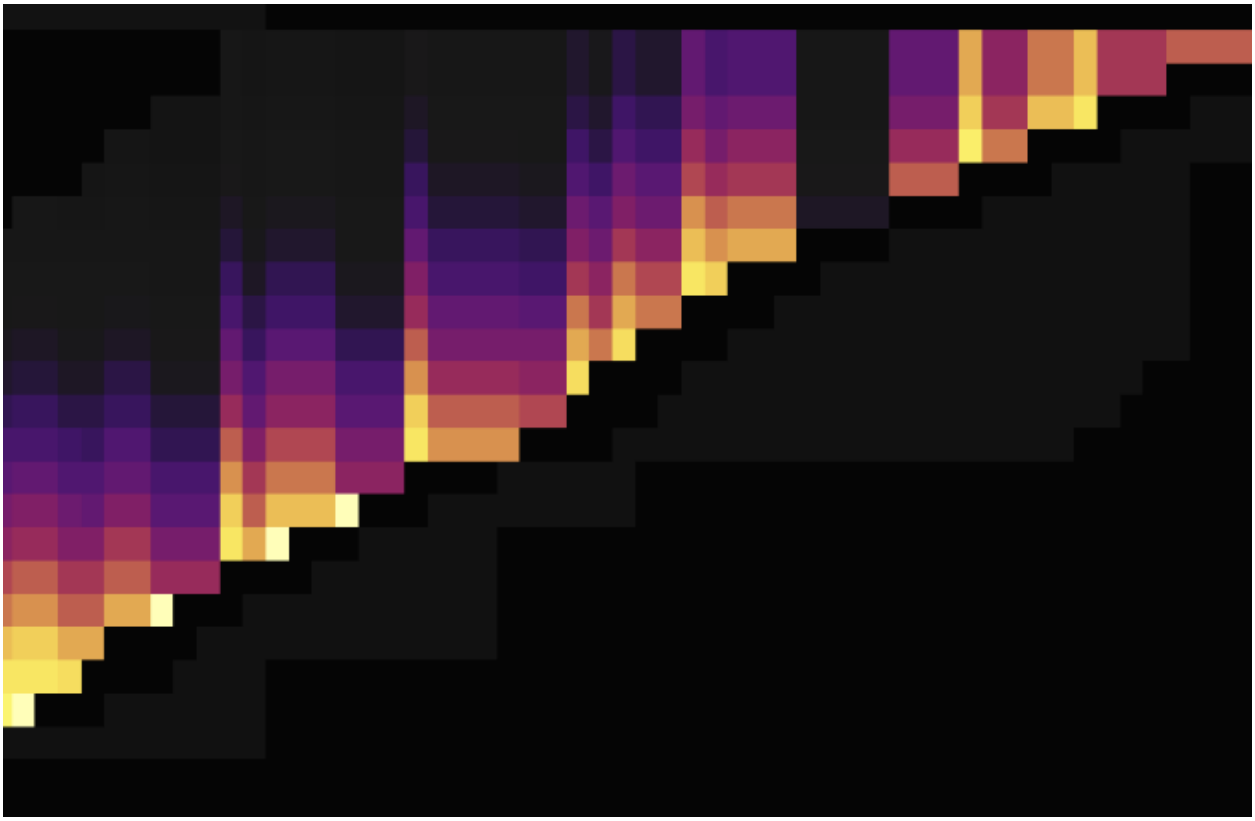


图 77: Dashboard 示例 7

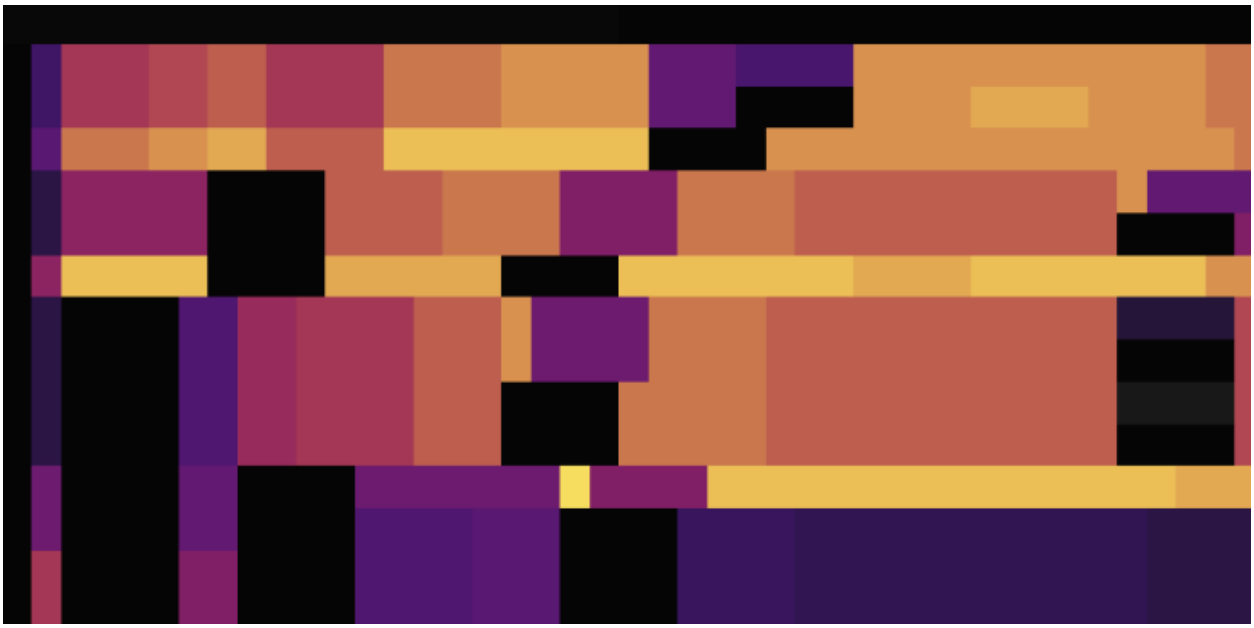


图 78: Dashboard 示例 8

由流量图可见，使用 AUTO_RANDOM 代替 AUTO_INCREMENT 能很好地打散热点。

更详细的说明参见AUTO_RANDOM 文档。

10.2.3.6 小表热点的优化

TiDB 的 Coprocessor Cache 功能支持下推计算结果缓存。开启该功能后，将在 TiDB 实例侧缓存下推给 TiKV 计算的结果，对于小表读热点能起到比较好的效果。

更详细的说明参见下推计算结果缓存文档。

其他相关资料：

- TiDB 高并发写入场景最佳实践
- Split Region 使用文档

10.2.4 读写延迟增加

本文档介绍读写延迟增加、抖动时的排查思路，可能的原因和解决方法。

10.2.4.1 常见原因

10.2.4.1.1 TiDB 执行计划不对导致延迟增高

查询语句的执行计划不稳定，偶尔执行计划选择错误的索引，导致查询延迟增加。

现象：

- 如果慢日志中输出了执行计划，可以直接查看执行计划。用 `select tidb_decode_plan('xxx...')` 语句可以解析出具体的执行计划。
- 监控中的 key 扫描异常升高；慢日志中 SQL 执行时间 `Scan Keys` 数目较大。
- SQL 执行时间相比于其他数据库（例如 MySQL）有较大差距。可以对比其他数据库执行计划，例如 `Join Order` 是否不同。

可能的原因：

- 统计信息不准确

解决方案：

- 更新统计信息
  - 手动 `analyze table`，配合 crontab 定期 `analyze`，维持统计信息准确度。
  - 自动 `auto analyze`，调低 `analyze ratio` 阈值，提高收集频次，并设置运行时间窗口。示例如下：
    * `set global tidb_auto_analyze_ratio=0.2;`
    * `set global tidb_auto_analyze_start_time='00:00 +0800';`
    * `set global tidb_auto_analyze_end_time='06:00 +0800';`
- 绑定执行计划
  - 修改业务 SQL，使用 `use index` 固定使用列上的索引。
  - 3.0 版本下，业务可以不用修改 SQL，使用 `create global binding` 创建 `force index` 的绑定 SQL。
  - 4.0 版本支持 SQL Plan Management，可以避免因执行计划不稳定导致的性能下降。

10.2.4.1.2 PD 异常

现象：

监控中 PD TSO 的 wait duration 异常升高。wait duration 代表从开始等待 PD 返回，到等待结束的时间。

可能的原因：

- 磁盘问题。PD 所在的节点 I/O 被占满，排查是否有其他 I/O 高的组件与 PD 混合部署以及磁盘的健康情况，可通过监控 Grafana -> disk performance -> latency 和 load 等指标进行验证，必要时可以使用 fio 工具对盘进行检测，见案例 case-292。
- PD 之间的网络问题。PD 日志中有 `"lost the TCP streaming connection"`，排查 PD 之间网络是否有问题，可通过监控 Grafana -> PD -> etcd 的 round trip 来验证，见案例 case-177。
- 系统负载高，日志中能看到 `"server is likely overloaded"`，见案例 case-214。
- 选举不出 leader。PD 日志中有 `"lease is not expired"`，见 issues https://github.com/etcd-io/etcd/issues/10355。v3.0.x 版本和 v2.1.19 版本已解决该问题，见案例 case-875。
- 选举慢。Region 加载时间长，从 PD 日志中 `grep "regions cost"`（例如日志中可能是 `load 460927 regions cost 11.77099s`），如果出现秒级，则说明较慢，v3.0 版本可开启 Region Storage（设置 `use-region-storage` 为 `true`），该特性能极大缩短加载 Region 的时间，见案例 case-429。

- TiDB 与 PD 之间的网络问题，应排查网络相关情况。通过监控 Grafana -> blackbox_exporter -> ping latency 确定 TiDB 到 PD leader 的网络是否正常。
- PD 报 FATAL 错误，日志中有 `"range failed to find revision pair"`。v3.0.8 中已经解决问题，见 PR https://github.com/pingcap/pd/pull/2040。详情请参考案例 case-947。
- 使用 `/api/v1/regions` 接口时 Region 数量过多可能会导致 PD OOM。已于 v3.0.8 版本修复，见 https://github.com/pingcap/pd/pull/1986。
- 滚动升级的时候 PD OOM，gRPC 消息大小没限制，监控可看到 TCP InSegs 较大，已于 v3.0.6 版本修复，见 https://github.com/pingcap/pd/pull/1952。
- PD panic，请提交 bug。
- 其他原因，通过 `curl http://127.0.0.1:2379/debug/pprof/goroutine?debug=2` 抓取 goroutine，并提交 bug。

10.2.4.1.3 TiKV 异常

现象：

监控中 KV Cmd Duration 异常升高。KV Cmd Duration 是 TiDB 发送请求给 TiKV 到收到回复的延迟。

可能的原因：

- 查看 gRPC duration。gRPC duration 是请求在 TiKV 端的总耗时。通过对比 TiKV 的 gRPC duration 以及 TiDB 中的 KV duration 可以发现潜在的网络问题。比如 gRPC duration 很短但是 TiDB 的 KV duration 显示很长，说明 TiDB 和 TiKV 之间网络延迟可能很高，或者 TiDB 和 TiKV 之间的网卡带宽被占满。
- TiKV 重启了导致重新选举
  - TiKV panic 之后又被 `systemd` 重新拉起正常运行，可以通过查看 TiKV 的日志来确认是否有 `panic`，这种情况属于非预期，需要报 bug。
  - 被第三者 `stop/kill`，被 `systemd` 重新拉起。查看 `dmesg` 和 `TiKV log` 确认原因。
  - TiKV 发生 OOM 导致重启了。
  - 动态调整 THP 导致 hung 住，见案例 case-500。
- 查看监控：Grafana -> TiKV-details -> errors 面板 `server is busy` 看到 TiKV RocksDB 出现 write stall 导致发生重新选举。
- TiKV 发生网络隔离导致重新选举。
- `block-cache` 配置太大导致 OOM，在监控 Grafana -> TiKV-details 选中对应的 instance 之后查看 RocksDB 的 `block cache size` 监控来确认是否是该问题。同时请检查 `[storage.block-cache] capacity = # "1GB"` 参数是否设置合理，默认情况下 TiKV 的 `block-cache` 设置为机器总内存的 45%。在容器化部署时需要显式指定该参数，因为 TiKV 获取的是物理机的内存，可能会超出单个 container 的内存限制。
- Coprocessor 收到大量大查询，返回的数据量太大，gRPC 发送速度跟不上 Coprocessor 向客户端输出数据的速度导致 OOM。可以通过检查监控：Grafana -> TiKV-details -> coprocessor overview 的 `response size` 是否超过 `network outbound` 流量来确认是否属于这种情况。

10.2.4.1.4 TiKV 单线程瓶颈

TiKV 中存在一些单线程线程，可能会成为瓶颈。

- 单个 TiKV Region 过多导致单个 gRPC 线程成为瓶颈（查看 Grafana -> TiKV-details -> Thread CPU/gRPC CPU Per Thread 监控），v3.x 以上版本可以开启 Hibernate Region 特性来解决，见案例 case-612。
- v3.0 之前版本 Raftstore 单线程或者 Apply 单线程到达瓶颈（Grafana -> TiKV-details -> Thread CPU/raft store CPU 和 Async apply CPU 超过 80%），可以选择扩容 TiKV（v2.x 版本）实例或者升级到多线程模型的 v3.x 版本。

10.2.4.1.5 CPU Load 升高

现象：

CPU 资源使用到达瓶颈

可能的原因：

- 热点问题。
- 整体负载高，排查 TiDB 的 slow query 和 expensive query。对运行的 query 进行优化，如果缺索引就加索引，如果可以批量执行就批量执行。另一个方案是对集群进行扩容。

10.2.4.2 其它原因

10.2.4.2.1 集群维护

通常大多数的线上集群有 3 或 5 个 PD 节点，如果维护的主机上有 PD 组件，需要具体考虑节点是 leader 还是 follower，关闭 follower 对集群运行没有任何影响，关闭 leader 需要先切换，并在切换时有 3 秒左右的性能抖动。

10.2.4.2.2 少数派副本离线

TiDB 集群默认配置为 3 副本，每一个 Region 都会在集群中保存 3 份，它们之间通过 Raft 协议来选举 Leader 并同步数据。Raft 协议可以保证在数量小于副本数（注意：不是节点数）一半的节点挂掉或者隔离的情况下，仍然能够提供服务，并且不丢失任何数据。对于 3 副本集群，挂掉一个节点可能会导致性能抖动，可用性和正确性理论上不会受影响。

10.2.4.2.3 新增索引

由于创建索引在扫表回填索引的时候会消耗大量资源，甚至与一些频繁更新的字段会发生冲突导致正常业务受到影响。大表创建索引的过程往往会持续很长时间，所以要尽可能地平衡执行时间和集群性能之间的关系，比如选择非高频更新时间段。

参数调整：

目前主要使用 `tidb_ddl_reorg_worker_cnt` 和 `tidb_ddl_reorg_batch_size` 这两个参数来动态调整索引创建速度，通常来说它们的值越小对系统影响越小，但是执行时间越长。

一般情况下，先将值保持为默认的 4 和 256，观察集群资源使用情况和响应速度，再逐渐调大 `tidb_ddl_reorg_worker_cnt` 参数来增加并发，观察监控如果系统没有发生明显的抖动，再逐渐调大 `tidb_ddl_reorg_batch_size` 参数，但如果索引涉及的列更新很频繁的话就会造成大量冲突造成失败重试。

另外还可以通过调整参数 tidb_ddl_reorg_priority 为 PRIORITY_HIGH 来让创建索引的任务保持高优先级来提升速度，但在通用 OLTP 系统上，一般建议保持默认。

10.2.4.2.4 GC 压力大

TiDB 的事务的实现采用了 MVCC（多版本并发控制）机制，当新写入的数据覆盖旧的数据时，旧的数据不会被替换掉，而是与新写入的数据同时保留，并以时间戳来区分版本。GC 的任务便是清理不再需要的旧数据。

- Resolve Locks 阶段在 TiKV 一侧会产生大量的 scan_lock 请求，可以在 gRPC 相关的 metrics 中观察到。scan_lock 请求会对全部的 Region 调用。
- Delete Ranges 阶段会往 TiKV 发送少量的 unsafe_destroy_range 请求，也可能没有。可以在 gRPC 相关的 metrics 中和 GC 分类下的 GC tasks 中观察到。
- Do GC 阶段，默认每台 TiKV 会自动扫描本机上的 leader Region 并对每一个 leader 进行 GC，这一活动可以在 GC 分类下的 GC tasks 中观察到。

10.2.5 乐观事务模型下写写冲突问题排查

本文介绍 TiDB 中乐观锁下写写冲突出现的原因以及解决方案。

在 v3.0.8 版本之前，TiDB 默认采用乐观事务模型，在事务执行过程中并不会做冲突检测，而是在事务最终 COMMIT 提交时触发两阶段提交，并检测是否存在写写冲突。当出现写写冲突，并且开启了事务重试机制，则 TiDB 会在限定次数内进行重试，最终重试成功或者达到重试次数上限后，会给客户端返回结果。因此，如果 TiDB 集群中存在大量的写写冲突情况，容易导致集群的 Duration 比较高。

10.2.5.1 出现写写冲突的原因

TiDB 中使用 Percolator 事务模型来实现 TiDB 中的事务。Percolator 总体上就是一个二阶段提交的实现。具体的二阶段提交过程可参考乐观事务文档。

当客户端发起 COMMIT 请求的时候，TiDB 开始两阶段提交：

1. TiDB 从所有要写入的 Key 中选择一个作为当前事务的 Primary Key
2. TiDB 向所有的本次提交涉及到的 TiKV 发起 prewrite 请求，TiKV 判断是否所有 Key 都可以 prewrite 成功
3. TiDB 收到所有 Key 都 prewrite 成功的消息
4. TiDB 向 PD 请求 commit_ts
5. TiDB 向 Primary Key 发起第二阶段提交。Primary Key 所在的 TiKV 收到 commit 操作后，检查数据合法性，清理 prewrite 阶段留下的锁
6. TiDB 收到两阶段提交成功的信息

写写冲突发生在 prewrite 阶段，当发现有其他的事务在写当前 Key (data.commit_ts > txn.start_ts)，则会发生写写冲突。

TiDB 会根据 tidb_disable_txn_auto_retry 和 tidb_retry_limit 参数设置的情况决定是否进行重试，如果设置了不重试，或者重试次数达到上限后还是没有 prewrite 成功，则向 TiDB 返回 Write Conflict 错误。

10.2.5.2 如何判断当前集群存在写写冲突

可以通过 Grafana 监控查看集群写写冲突的情况：

- 通过 TiDB 监控面板中 KV Errors 监控栏中 KV Backoff OPS 监控指标项，查看 TiKV 中返回错误信息的数量

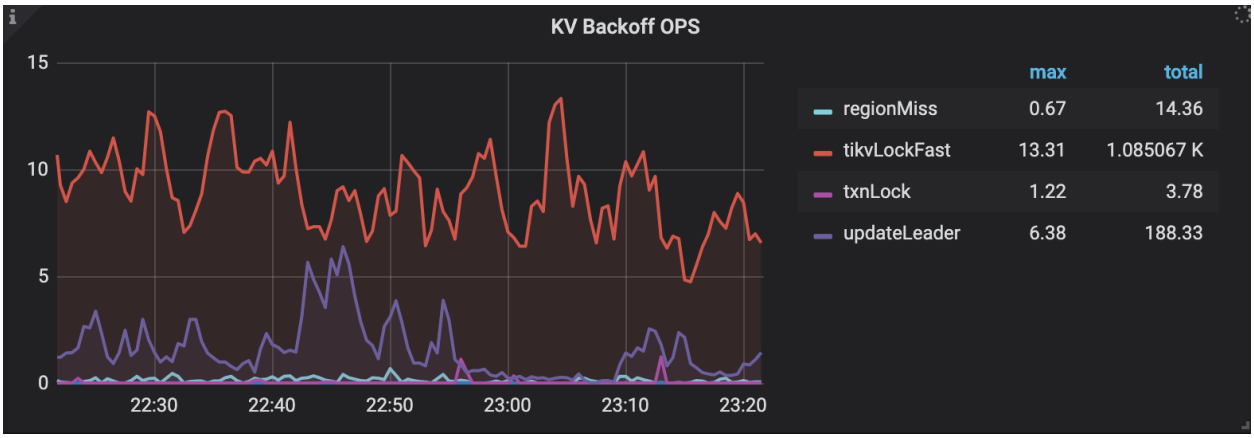


图 79: kv-backoff-ops

txnlock 表示集群中存在写写冲突，txnLockFast 表示集群中存在读写冲突。

- 通过 TiDB 监控面板中 KV Errors 监控栏中 Lock Resolve OPS 监控指标项，查看事务冲突相关的数量

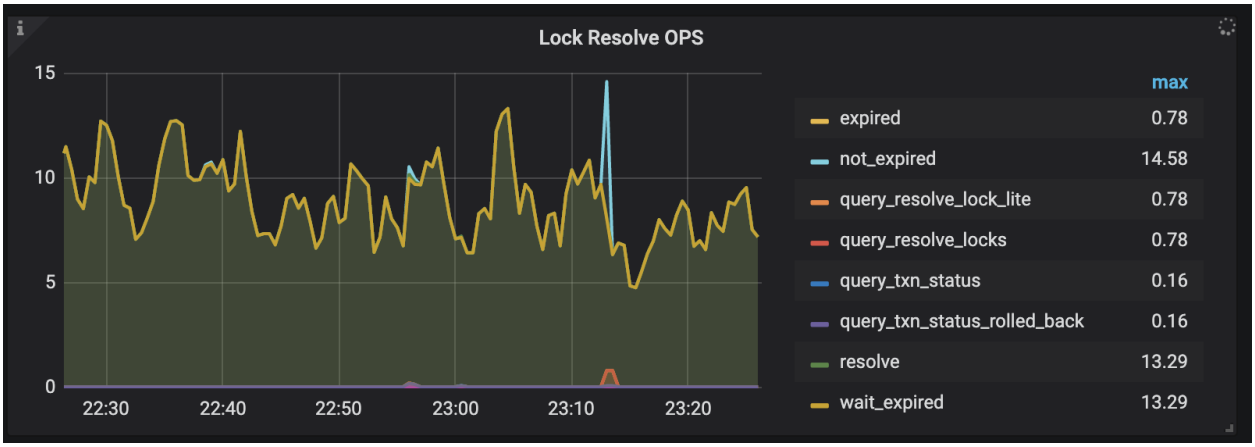


图 80: lock-resolve-ops

expired、 not_expired、 wait_expired 表示对应的 lock 状态

- 通过 TiDB 监控面板中 KV Errors 监控栏中 KV Retry Duration 监控指标项，查看 KV 重试请求的时间

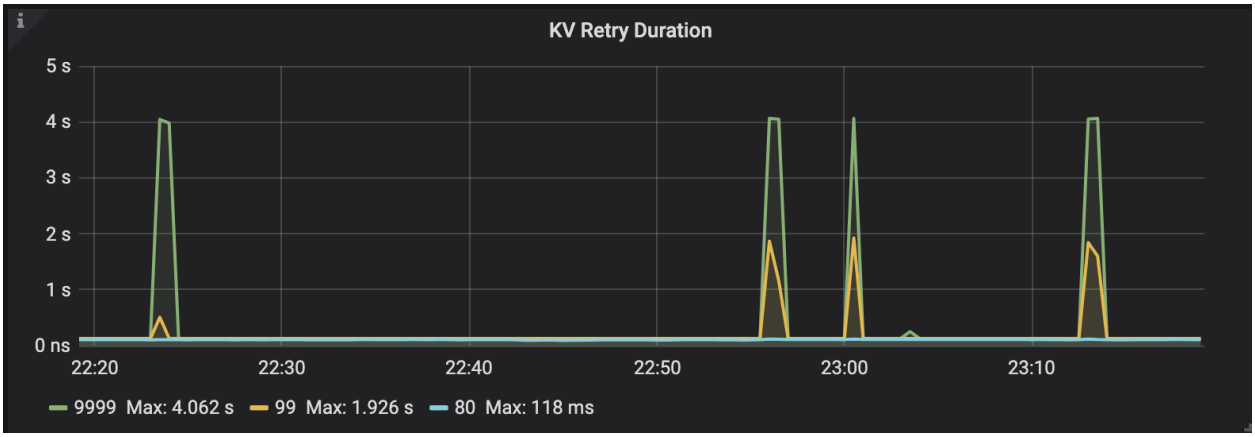


图 81: kv-retry-duration

也可以通过 TiDB 日志查看是否有 `[kv:9007]Write conflict` 关键字，如果搜索到对应关键字，则可以表明集群中存在写写冲突。

10.2.5.3 如何解决写写冲突问题

如果通过以上方式判断出集群中存在大量的写写冲突，建议找到冲突的数据，以及写写冲突的原因，看是否能从应用程序修改逻辑，加上重试的逻辑。当出现写写冲突的时候，可以在 TiDB 日志中看到类似的日志：

```
[2020/05/12 15:17:01.568 +08:00] [WARN] [session.go:446] ["commit failed"] [conn=3] ["finished
    ↪ txn"="Txn{state=invalid}"] [error="[kv:9007]Write conflict, txnStartTS
    ↪ =416617006551793665, conflictStartTS=416617018650001409, conflictCommitTS
    ↪ =416617023093080065, key={tableID=47, indexID=1, indexValues={string, }} primary={tableID
    ↪ =47, indexID=1, indexValues={string, }} [try again later]"]
```

关于日志的解释如下：

- `[kv:9007]Write conflict`：表示出现了写写冲突
- `txnStartTS=416617006551793665`：表示当前事务的 start_ts 时间戳，可以通过 pd-ctl 工具将时间戳转换为具体时间
- `conflictStartTS=416617018650001409`：表示冲突事务的 start_ts 时间戳，可以通过 pd-ctl 工具将时间戳转换为具体时间
- `conflictCommitTS=416617023093080065`：表示冲突事务的 commit_ts 时间戳，可以通过 pd-ctl 工具将时间戳转换为具体时间
- `key={tableID=47, indexID=1, indexValues={string, }}`：表示当前事务中冲突的数据，tableID 表示发生冲突的表的 ID，indexID 表示是索引数据发生了冲突。如果是数据发生了冲突，会打印 `handle=x` 表示对应哪行数据发生了冲突，indexValues 表示发生冲突的索引数据
- `primary={tableID=47, indexID=1, indexValues={string, }}`：表示当前事务中的 Primary Key 信息

通过 pd-ctl 将时间戳转换为可读时间：

```
tiup ctl:v<CLUSTER_VERSION> pd -u https://127.0.0.1:2379 tso {TIMESTAMP}
```

通过 tableID 查找具体的表名：

```
curl http://{TiDBIP}:10080/db-table/{tableID}
```

通过 indexID 查找具体的索引名：

```
SELECT * FROM INFORMATION_SCHEMA.TIDB_INDEXES WHERE TABLE_SCHEMA='{db_name}' AND TABLE_NAME='{
    ↪ table_name}' AND INDEX_ID={indexID};
```

另外在 v3.0.8 及之后版本默认使用悲观事务模式，从而避免在事务提交的时候因为冲突而导致失败，无需修改应用程序。悲观事务模式下会在每个 DML 语句执行的时候，加上悲观锁，用于防止其他事务修改相同 Key，从而保证在最后提交的 prewrite 阶段不会出现写写冲突的情况。

10.2.6 TiDB 磁盘 I/O 过高的处理办法

本文主要介绍如何定位和处理 TiDB 存储 I/O 过高的问题。

10.2.6.1 确认当前 I/O 指标

当出现系统响应变慢的时候，如果已经排查了 CPU 的瓶颈、数据事务冲突的瓶颈后，就需要从 I/O 来入手来辅助判断目前的系统瓶颈点。

10.2.6.1.1 从监控定位 I/O 问题

最快速的定位手段是从监控来查看整体的 I/O 情况，可以从集群部署工具 (TiUP) 默认会部署的监控组件 Grafana 来查看对应的 I/O 监控，跟 I/O 相关的 Dashboard 有 `Overview`，`Node_exporter`，`Disk-Performance`。

第一类面板

在 `Overview` > `System Info` > `IO Util` 中，可以看到集群中每个机器的 I/O 情况，该指标和 Linux iostat 监控中的 util 类似，百分比越高代表磁盘 I/O 占用越高：

- 如果监控中只有一台机器的 I/O 高，那么可以辅助判断当前有读写热点。
- 如果监控中大部分机器的 I/O 都高，那么集群现在有高 I/O 负载存在。

如果发现某台机器的 I/O 比较高，可以从监控 `Disk-Performance Dashboard` 进一步观察 I/O 的使用情况，结合 `Disk Latency`，`Disk Load` 等 metric 判断是否存在异常，必要时可以使用 fio 工具对磁盘进行检测。

第二类面板

TiDB 集群主要的持久化组件是 TiKV 集群，一个 TiKV 包含两个 RocksDB 实例：一个用于存储 Raft 日志，位于 data/raft，一个用于存储真正的数据，位于 data/db。

在 `TiKV-Details` > `Raft IO` 中，可以看到这两个实例磁盘写入的相关 metric：

- `Append log duration`：该监控表明了存储 Raft 日志的 RocksDB 写入的响应时间，99% 的响应应该在 50ms 以内。
- `Apply log duration`：该监控表明了存储真正数据的 RocksDB 写入的响应时间，99% 的响应应该在 100ms 以内。

这两个监控还有 .. per server 的监控面板来提供辅助查看热点写入的情况。

第三类面板

在 TiKV-Details > Storage 中，有关于 storage 相关情况的监控：

- Storage command total：收到的不同命令的个数。
- Storage async write duration：包括了磁盘 sync duration 等监控项，可能和 Raft IO 有关。如遇到异常情况，需要通过 log 来检查相关组件的工作状态是否正常。

其他面板

此外，可能还需要一些其它内容来辅助确认瓶颈是否为 I/O，并可以尝试调整一些参数。通过查看 TiKV gRPC 的 prewrite/commit/raw-put（仅限 raw kv 集群）duration，确认确实是 TiKV 写入慢了。常见的几种情况如下：

- append log 慢。TiKV Grafana 的 Raft I/O 和 append log duration 比较高，通常情况下是由于写盘慢了，可以检查 RocksDB - raft 的 WAL Sync Duration max 值来确认，否则可能需要报 bug。
- raftstore 线程繁忙。TiKV grafana 的 Raft Propose/propose wait duration 明显高于 append log duration。请查看以下两点：
  - [raftstore] 的 store-pool-size 配置是否过小（该值建议在 [1,5] 之间，不建议太大）。
  - 机器的 CPU 是不是不够了。
- apply log 慢。TiKV Grafana 的 Raft I/O 和 apply log duration 比较高，通常会伴随着 Raft Propose/apply wait duration 比较高。可能的情况如下：
  - [raftstore] 的 apply-pool-size 配置过小（建议在 [1, 5] 之间，不建议太大），Thread CPU/apply cpu 比较高；
  - 机器的 CPU 资源不够了。
  - Region 写入热点问题，单个 apply 线程 CPU 使用率比较高（通过修改 Grafana 表达式，加上 by (instance, name) 来看各个线程的 cpu 使用情况），暂时对于单个 Region 的热点写入没有很好的方式，最近在优化该场景。
  - 写 RocksDB 比较慢，RocksDB kv/max write duration 比较高（单个 Raft log 可能包含很多个 kv，写 RocksDB 的时候会把 128 个 kv 放在一个 write batch 写入到 RocksDB，所以一次 apply log 可能涉及到多次 RocksDB 的 write）。
  - 其他情况，需要报 bug。
- raft commit log 慢。TiKV Grafana 的 Raft I/O 和 commit log duration 比较高（4.x 版本的 Grafana 才有该 metric）。每个 Region 对应一个独立的 Raft group，Raft 本身是有流控机制的，类似 TCP 的滑动窗口机制，通过参数raftstore raft-max-inflight-msgs = 256 来控制滑动窗口的大小，如果有热点写入并且 commit log duration 比较高可以适度调大改参数，比如 1024。

10.2.6.1.2 从 log 定位 I/O 问题

- 如果客户端报 server is busy 错误，特别是 raftstore is busy 的错误信息，会和 I/O 有相关性。

  可以通过查看监控：grafana -> TiKV -> errors 监控确认具体 busy 原因。其中，server is busy 是 TiKV 自身的流控机制，TiKV 通过这种方式告知 tidb/ti-client 当前 TiKV 的压力过大，过一会儿再尝试。

- TiKV RocksDB 日志出现 `write stall`。

  可能是 `level0 sst` 太多导致 stall。可以添加参数 `[rocksdb] max-sub-compactions = 2`（或者 `3`）加快 level0 sst 往下 compact 的速度，该参数的意思是将从 level0 到 level1 的 compaction 任务最多切成 `max-sub-compactions` 个子任务交给多线程并发执行。

  如果磁盘 I/O 能力持续跟不上写入，建议扩容。如果磁盘的吞吐达到了上限（例如 SATA SSD 的吞吐相对 NVME SSD 会低很多）导致 write stall，但是 CPU 资源又比较充足，可以尝试采用压缩率更高的压缩算法来缓解磁盘的压力，用 CPU 资源换磁盘资源。

  比如 `default cf compaction` 压力比较大时，可以调整参数 `[rocksdb.defaultcf] compression-per-level = ["no", "no", "lz4", "lz4", "lz4", "zstd", "zstd"]` 改成 `compression-per-level = ["no", "no", "zstd", "zstd", "zstd", "zstd", "zstd"]`。

10.2.6.1.3 从告警发现 I/O 问题

集群部署工具 (TiUP) 默认部署的告警组件，官方已经预置了相关的告警项目和阈值，I/O 相关项包括：

- TiKV_write_stall
- TiKV_raft_log_lag
- TiKV_async_request_snapshot_duration_seconds
- TiKV_async_request_write_duration_seconds
- TiKV_raft_append_log_duration_secs
- TiKV_raft_apply_log_duration_secs

10.2.6.2 I/O 问题处理方案

1. 当确认为热点 I/O 问题的时候，需要参考TiDB 热点问题处理来消除相关的热点 I/O 情况。
2. 当确认整体 I/O 已经到达瓶颈的时候，且从业务侧能够判断 I/O 的能力会持续的跟不上，那么就可以利用分布式数据库的 scale 的能力，采用扩容 TiKV 节点数量的方案来获取更大的整体 I/O 吞吐量。
3. 调整上述说明中的一些参数，使用计算/内存资源来换取磁盘的存储资源。

10.2.7 TiDB 锁冲突问题处理

TiDB 支持完整的分布式事务，自 v3.0 版本起，提供乐观事务与悲观事务两种事务模式。本文介绍如何使用 Lock View 排查锁相关的问题，以及如何处理使用乐观事务或者悲观事务的过程中常见的锁冲突问题。

10.2.7.1 使用 Lock View 排查锁相关的问题

自 v5.1 版本起，TiDB 支持 Lock View 功能。该功能在 `information_schema` 中内置了若干系统表，用于提供更多关于锁冲突和锁等待的信息。

> **注意：**
>
> Lock View 功能目前仅提供悲观锁的冲突和等待信息。

关于这些系统表的详细说明，请参考以下文档：

- TIDB_TRX 与 CLUSTER_TIDB_TRX：提供当前 TiDB 节点上或整个集群上所有运行中的事务的信息，包括事务是否处于等锁状态、等锁时间和事务曾经执行过的语句的 Digest 等信息。
- DATA_LOCK_WAITS：提供关于 TiKV 内的悲观锁等锁信息，包括阻塞和被阻塞的事务的 start_ts、被阻塞的 SQL 语句的 Digest 和发生等待的 key。
- DEADLOCKS 与 CLUSTER_DEADLOCKS：提供当前 TiDB 节点上或整个集群上最近发生过的若干次死锁的相关信息，包括死锁环中事务之间的等待关系、事务当前正在执行的语句的 Digest 和发生等待的 key。

> 注意：
>
> Lock View 所属的系统表中展示的 SQL 语句为归一化的 SQL 语句（即去除了格式和参数的 SQL 语句），通过内部查询从 SQL Digest 获得，因而无法获取包括格式和参数在内的完整语句。有关 SQL Digest 和归一化 SQL 语句的详细介绍，请参阅Statement Summary Tables。

以下为排查部分问题的示例。

10.2.7.1.1 死锁错误

要获取最近发生的死锁错误的信息，可查询 DEADLOCKS 或 CLUSTER_DEADLOCKS 表。

以查询 DEADLOCKS 表为例，请执行以下 SQL 语句：

```
select * from information_schema.deadlocks;
```

示例输出：

```
+--
    ↪ -----------+----------------------------+-----------+--------------------+-----------------------------
    ↪
| DEADLOCK_ID | OCCUR_TIME                 | RETRYABLE | TRY_LOCK_TRX_ID    | CURRENT_SQL_DIGEST
    ↪                                                 | CURRENT_SQL_DIGEST_TEXT                 |
    ↪  KEY                                    | KEY_INFO
    ↪
    ↪ | TRX_HOLDING_LOCK   |
+--
    ↪ -----------+----------------------------+-----------+--------------------+-----------------------------
    ↪
|           1 | 2021-08-05 11:09:03.230341 |         0 | 426812829645406216 | 22230766411
    ↪ edb40f27a68dadefc63c6c6970d5827f1e5e22fc97be2c4d8350d | update `t` set `v` = ? where `id`
    ↪  = ? ; | 74800000000000000355F728000000000000002 | {"db_id":1,"db_name":"test","table_id"
    ↪ :53,"table_name":"t","handle_type":"int","handle_value":"2"} | 426812829645406217 |
|           1 | 2021-08-05 11:09:03.230341 |         0 | 426812829645406217 | 22230766411
    ↪ edb40f27a68dadefc63c6c6970d5827f1e5e22fc97be2c4d8350d | update `t` set `v` = ? where `id`
    ↪  = ? ; | 74800000000000000355F728000000000000001 | {"db_id":1,"db_name":"test","table_id"
    ↪ :53,"table_name":"t","handle_type":"int","handle_value":"1"} | 426812829645406216 |
```

```
+--
    ↪ -----------+---------------------------+-----------+--------------------+-------------------------------
    ↪
```

查询结果会显示死锁错误中多个事务之间的等待关系和各个事务当前正在执行的 SQL 语句的归一化形式（即去掉参数和格式的形式），以及发生冲突的 key 及其从 key 中解读出的一些信息。

例如在上述例子中，第一行意味着 ID 为 426812829645406216 的事务当前正在执行形如 `update `t` set `v` = ? where `id` = ?` 的语句，被另一个 ID 为 426812829645406217 的事务阻塞；而 426812829645406217 同样也在执行一条形如 `update `t` set `v` = ? where `id` = ?` 的语句，并被 ID 为 426812829645406216 的事务阻塞，两个事务因而构成死锁。

10.2.7.1.2 少数热点 key 造成锁排队

DATA_LOCK_WAITS 系统表提供 TiKV 节点上的等锁情况。查询该表时，TiDB 将自动从所有 TiKV 节点上获取当前时刻的等锁信息。当少数热点 key 频繁被上锁并阻塞较多事务时，你可以查询 DATA_LOCK_WAITS 表并按 key 对结果进行聚合，以尝试找出经常发生问题的 key：

```
select `key`, count(*) as `count` from information_schema.data_lock_waits group by `key` order by
    ↪  `count` desc;
```

示例输出：

```
+----------------------------------------+-------+
| key                                    | count |
+----------------------------------------+-------+
| 748000000000000415F728000000000000001 |     2 |
| 748000000000000415F728000000000000002 |     1 |
+----------------------------------------+-------+
```

为避免偶然性，你可考虑进行多次查询。

如果已知频繁出问题的 key，可尝试从 TIDB_TRX 或 CLUSTER_TIDB_TRX 表中获取试图上锁该 key 的事务的信息。

需要注意 TIDB_TRX 和 CLUSTER_TIDB_TRX 表所展示的信息也是对其进行查询的时刻正在运行的事务的信息，并不展示已经结束的事务。如果并发的事务数量很大，该查询的结果集也可能很大，可以考虑添加 limit 子句，或用 where 子句筛选出等锁时间较长的事务。需要注意，对 Lock View 中的多张表进行 join 时，不同表之间的数据并不保证在同一时刻获取，因而不同表中的信息可能并不同步。

以用 where 子句筛选出等锁时间较长的事务为例，请执行以下 SQL 语句：

```
select trx.* from information_schema.data_lock_waits as l left join information_schema.tidb_trx
    ↪ as trx on l.trx_id = trx.id where l.key = "748000000000000415F728000000000000001"\G
```

示例输出：

```
*************************** 1. row ***************************
                    ID: 426831815660273668
            START_TIME: 2021-08-06 07:16:00.081000
    CURRENT_SQL_DIGEST: 06da614b93e62713bd282d4685fc5b88d688337f36e88fe55871726ce0eb80d7
```

```
CURRENT_SQL_DIGEST_TEXT: update `t` set `v` = `v` + ? where `id` = ? ;
                  STATE: LockWaiting
     WAITING_START_TIME: 2021-08-06 07:16:00.087720
        MEM_BUFFER_KEYS: 0
       MEM_BUFFER_BYTES: 0
             SESSION_ID: 77
                   USER: root
                     DB: test
        ALL_SQL_DIGESTS: ["0fdc781f19da1c6078c9de7eadef8a307889c001e05f107847bee4cfc8f3cdf3","06
            ↪ da614b93e62713bd282d4685fc5b88d688337f36e88fe55871726ce0eb80d7"]
*************************** 2. row ***************************
                     ID: 426831818019569665
             START_TIME: 2021-08-06 07:16:09.081000
     CURRENT_SQL_DIGEST: 06da614b93e62713bd282d4685fc5b88d688337f36e88fe55871726ce0eb80d7
CURRENT_SQL_DIGEST_TEXT: update `t` set `v` = `v` + ? where `id` = ? ;
                  STATE: LockWaiting
     WAITING_START_TIME: 2021-08-06 07:16:09.290271
        MEM_BUFFER_KEYS: 0
       MEM_BUFFER_BYTES: 0
             SESSION_ID: 75
                   USER: root
                     DB: test
        ALL_SQL_DIGESTS: ["0fdc781f19da1c6078c9de7eadef8a307889c001e05f107847bee4cfc8f3cdf3","06
            ↪ da614b93e62713bd282d4685fc5b88d688337f36e88fe55871726ce0eb80d7"]
2 rows in set (0.00 sec)
```

10.2.7.1.3 事务被长时间阻塞

如果已知一个事务被另一事务（或多个事务）阻塞，且已知当前事务的 start_ts（即事务 ID），则可使用如下方式获取导致该事务阻塞的事务的信息。注意对 Lock View 中的多张表进行 join 时，不同表之间的数据并不保证在同一时刻获取，因而可能不同表中的信息可能并不同步。

```
select l.key, trx.*, tidb_decode_sql_digests(trx.all_sql_digests) as sqls from information_schema
    ↪ .data_lock_waits as l join information_schema.cluster_tidb_trx as trx on l.
    ↪ current_holding_trx_id = trx.id where l.trx_id = 426831965449355272\G
```

示例输出：

```
*************************** 1. row ***************************
                     key: 7480000000000004D5F728000000000000001
                INSTANCE: 127.0.0.1:10080
                      ID: 426832040186609668
              START_TIME: 2021-08-06 07:30:16.581000
     CURRENT_SQL_DIGEST: 06da614b93e62713bd282d4685fc5b88d688337f36e88fe55871726ce0eb80d7
CURRENT_SQL_DIGEST_TEXT: update `t` set `v` = `v` + ? where `id` = ? ;
                  STATE: LockWaiting
```

```
    WAITING_START_TIME: 2021-08-06 07:30:16.592763
       MEM_BUFFER_KEYS: 1
      MEM_BUFFER_BYTES: 19
            SESSION_ID: 113
                  USER: root
                    DB: test
       ALL_SQL_DIGESTS: ["0fdc781f19da1c6078c9de7eadef8a307889c001e05f107847bee4cfc8f3cdf3","
           ↪ a4e28cc182bdd18288e2a34180499b9404cd0ba07e3cc34b6b3be7b7c2de7fe9","06
           ↪ da614b93e62713bd282d4685fc5b88d688337f36e88fe55871726ce0eb80d7"]
                  sqls: ["begin ;","select * from `t` where `id` = ? for update ;","update `t`
                      ↪ set `v` = `v` + ? where `id` = ? ;"]
1 row in set (0.01 sec)
```

上述查询中，对 CLUSTER_TIDB_TRX 表的 ALL_SQL_DIGESTS 列使用了TIDB_DECODE_SQL_DIGESTS 函数，目的是将该列（内容为一组 SQL Digest）转换为其对应的归一化 SQL 语句，便于阅读。

如果当前事务的 start_ts 未知，可以尝试从 TIDB_TRX / CLUSTER_TIDB_TRX 表或者PROCESSLIST / CLUSTER_PROCESSLIST 表中的信息进行判断。

10.2.7.2 处理乐观锁冲突问题

以下介绍乐观事务模式下常见的锁冲突问题的处理方式。

10.2.7.2.1 读写冲突

在 TiDB 中，读取数据时，会获取一个包含当前物理时间且全局唯一递增的时间戳作为当前事务的 start_ts。事务在读取时，需要读到目标 key 的 commit_ts 小于这个事务的 start_ts 的最新的数据版本。当读取时发现目标 key 上存在 lock 时，因为无法知道上锁的那个事务是在 Commit 阶段还是 Prewrite 阶段，所以就会出现读写冲突的情况，如下图：

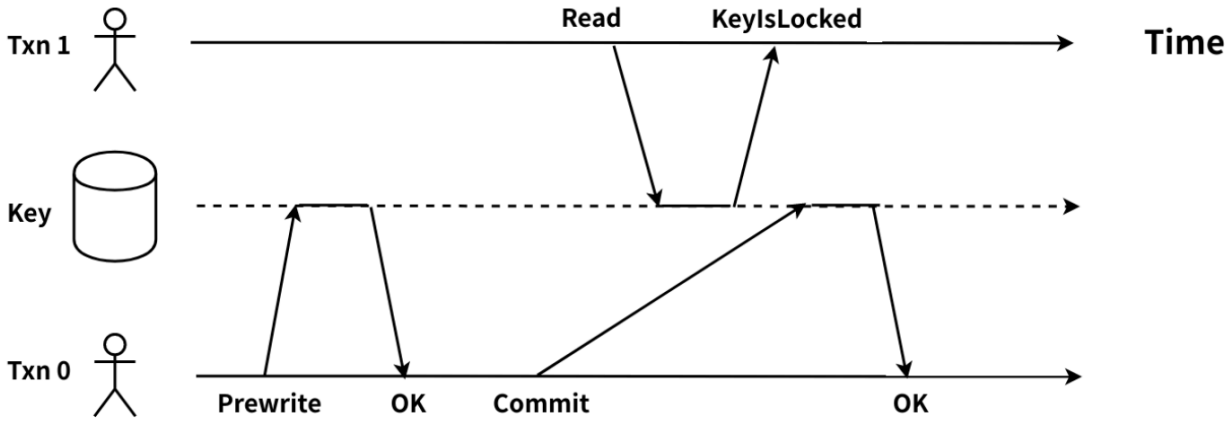


图 82: 读写冲突

分析：

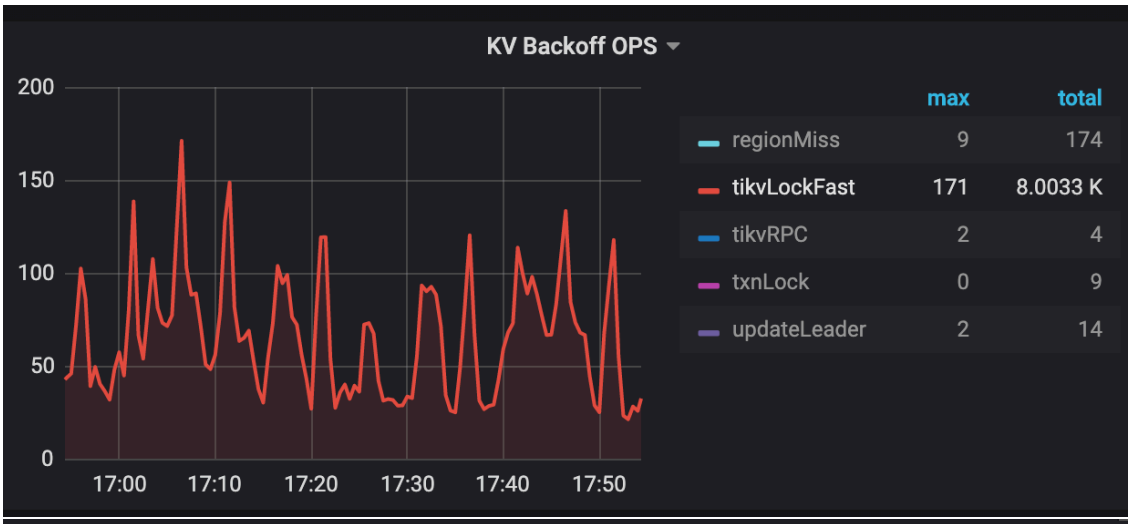
Txn0 完成了 Prewrite，在 Commit 的过程中 Txn1 对该 key 发起了读请求，Txn1 需要读取 start_ts > commit_ts 最近的 key 的版本。此时，Txn1 的 `start_ts` > Txn0 的 lock_ts，需要读取的 key 上的锁信息仍未清理，故无法判断 Txn0 是否提交成功，因此 Txn1 与 Txn0 出现读写冲突。

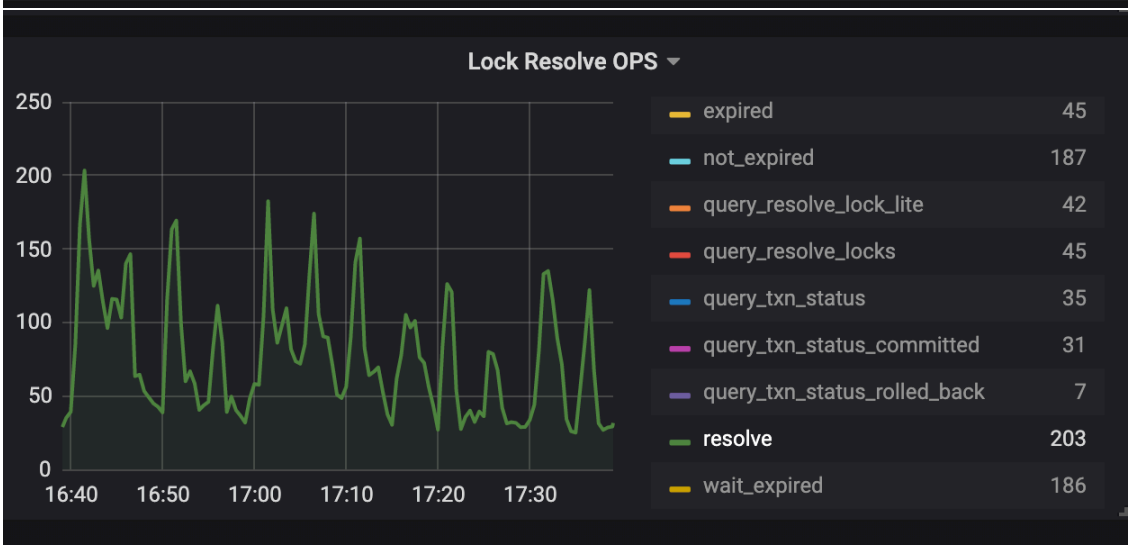
你可以通过如下两种途径来检测当前环境中是否存在读写冲突：

1. TiDB 监控及日志

    - 通过 TiDB Grafana 监控分析：

      观察 KV Errors 下 Lock Resolve OPS 面板中的 not_expired/resolve 监控项以及 KV Backoff OPS 面板中的 tikvLockFast 监控项，如果有较为明显的上升趋势，那么可能是当前的环境中出现了大量的读写冲突。其中，not_expired 是指对应的锁还没有超时，resolve 是指尝试清锁的操作，tikvLockFast 代表出现了读写冲突。





    - 通过 TiDB 日志分析：

      在 TiDB 的日志中可以看到下列信息：

```
[INFO] [coprocessor.go:743] ["[TIME_COP_PROCESS] resp_time:406.038899ms txnStartTS
    ↪ :416643508703592451 region_id:8297 store_addr:10.8.1.208:20160 backoff_ms:255
    ↪ backoff_types:[txnLockFast,txnLockFast] kv_process_ms:333 scan_total_write:0
```

```
    ↪ scan_processed_write:0 scan_total_data:0 scan_processed_data:0 scan_total_lock:0
    ↪  scan_processed_lock:0"]
```

- txnStartTS：发起读请求的事务的 start_ts，如上面示例中的 416643508703592451
- backoff_types：读写发生了冲突，并且读请求进行了 backoff 重试，重试的类型为 txnLockFast
- backoff_ms：读请求 backoff 重试的耗时，单位为 ms，如上面示例中的 255
- region_id：读请求访问的目标 region 的 id

2. 通过 TiKV 日志分析：

   在 TiKV 的日志可以看到下列信息：

```
[ERROR] [endpoint.rs:454] [error-response] [err=""locked primary_lock:7480000000000004
    ↪ D35F698000000000000010380000000004C788E0380000000004C0748 lock_version:
    ↪ 411402933858205712 key: 7480000000000004D35F7280000000004C0748 lock_ttl: 3008
    ↪ txn_size: 1""]
```

   这段报错信息表示出现了读写冲突，当读数据时发现 key 有锁阻碍读，这个锁包括未提交的乐观锁和未提交的 prewrite 后的悲观锁。

   - primary_lock：锁对应事务的 primary lock。
   - lock_version：锁对应事务的 start_ts。
   - key：表示被锁的 key。
   - lock_ttl: 锁的 TTL。
   - txn_size：锁所在事务在其 Region 的 key 数量，指导清锁方式。

处理建议：

- 在遇到读写冲突时会有 backoff 自动重试机制，如上述示例中 Txn1 会进行 backoff 重试，单次初始 10 ms，单次最大 3000 ms，总共最大 20000 ms
- 可以使用 TiDB Control 的子命令decoder 来查看指定 key 对应的行的 table id 以及 rowid：

```
./tidb-ctl decoder "t\x00\x00\x00\x00\x00\x00\x00\x1c_r\x00\x00\x00\x00\x00\x00\x00\xfa"
format: table_row
table_id: -9223372036854775780
row_id: -9223372036854775558
```

10.2.7.2.2 KeyIsLocked 错误

事务在 Prewrite 阶段的第一步就会检查是否有写写冲突，第二步会检查目标 key 是否已经被另一个事务上锁。当检测到该 key 被 lock 后，会在 TiKV 端报出 KeyIsLocked。目前该报错信息没有打印到 TiDB 以及 TiKV 的日志中。与读写冲突一样，在出现 KeyIsLocked 时，后台会自动进行 backoff 重试。

你可以通过 TiDB Grafana 监控检测 KeyIsLocked 错误：

观察 KV Errors 下 Lock Resolve OPS 面板中的 resolve 监控项以及 KV Backoff OPS 面板中的 txnLock 监控项，会有比较明显的上升趋势，其中 resolve 是指尝试清锁的操作，txnLock 代表出现了写冲突。



处理建议：

- 监控中出现少量 txnLock，无需过多关注。后台会自动进行 backoff 重试，单次初始 100 ms，单次最大 3000 ms。
- 如果出现大量的 txnLock，需要从业务的角度评估下冲突的原因。
- 使用悲观锁模式。

10.2.7.2.3 锁被清除 (LockNotFound) 错误

TxnLockNotFound 错误是由于事务提交的慢了，超过了 TTL 的时间。当要提交时，发现被其他事务给 Rollback 掉了。在开启 TiDB 自动重试事务的情况下，会自动在后台进行事务重试（注意显示和隐式事务的差别）。

你可以通过如下两种途径来查看 LockNotFound 报错信息：

1. 查看 TiDB 日志

如果出现了 TxnLockNotFound 的报错，会在 TiDB 的日志中看到下面的信息：

```
[WARN] [session.go:446] ["commit failed"] [conn=149370] ["finished txn"="Txn{state=invalid
    ↪ }"] [error="[kv:6]Error: KV error safe to retry tikv restarts txn: Txn(Mvcc(
    ↪ TxnLockNotFound{ start_ts: 412720515987275779, commit_ts: 412720519984971777, key:
    ↪ [116, 128, 0, 0, 0, 0, 1, 111, 16, 95, 114, 128, 0, 0, 0, 0, 0, 2] })) [try again
    ↪  later]"]
```

- start_ts：出现 TxnLockNotFound 报错的事务的 start_ts，如上例中的 412720515987275779
- commit_ts：出现 TxnLockNotFound 报错的事务的 commit_ts，如上例中的 412720519984971777

2. 查看 TiKV 日志

   如果出现了 TxnLockNotFound 的报错，在 TiKV 的日志中同样可以看到相应的报错信息：

```
Error: KV error safe to retry restarts txn: Txn(Mvcc(TxnLockNotFound)) [ERROR [Kv.rs:708] ["
    ↪ KvService::batch_raft send response fail"] [err=RemoteStoped]
```

处理建议：

- 通过检查 start_ts 和 commit_ts 之间的提交间隔，可以确认是否超过了默认的 TTL 的时间。

  查看提交间隔：

```
tiup ctl:v<CLUSTER_VERSION> pd tso [start_ts]
tiup ctl:v<CLUSTER_VERSION> pd tso [commit_ts]
```

- 建议检查下是否是因为写入性能的缓慢导致事务提交的效率差，进而出现了锁被清除的情况。
- 在关闭 TiDB 事务重试的情况下，需要在应用端捕获异常，并进行重试。

10.2.7.3 处理悲观锁冲突问题

以下介绍悲观事务模式下常见的锁冲突问题的处理方式。

> 注意：
>
> 即使设置了悲观事务模式，autocommit 事务仍然会优先尝试使用乐观事务模式进行提交，并在发生冲突后、自动重试时切换为悲观事务模式。

10.2.7.3.1 读写冲突

报错信息以及处理建议同乐观锁模式。

10.2.7.3.2 pessimistic lock retry limit reached

在冲突非常严重的场景下，或者当发生 write conflict 时，乐观事务会直接终止，而悲观事务会尝试用最新数据重试该语句直到没有 write conflict。因为 TiDB 的加锁操作是一个写入操作，且操作过程是先读后写，需要 2 次 RPC。如果在这中间发生了 write conflict，那么会重试。每次重试都会打印日志，不用特别关注。重试次数由pessimistic-txn.max-retry-count 定义。

可通过查看 TiDB 日志查看报错信息：

悲观事务模式下，如果发生 write conflict，并且重试的次数达到了上限，那么在 TiDB 的日志中会出现含有下述关键字的报错信息。如下：

```
err="pessimistic lock retry limit reached"
```

处理建议：

- 如果上述报错出现的比较频繁，建议从业务的角度进行调整。
- 如果业务中包含对同一行（同一个 key）的高并发上锁而频繁冲突，可以尝试启用系统变量tidb_pessimistic_txn_fair_locking。需要注意启用该选项可能对存在锁冲突的事务带来一定程度的吞吐下降（平均延迟上升）的代价。对于新部署的集群，该选项默认启用 (ON)。

10.2.7.3.3 Lock wait timeout exceeded

在悲观锁模式下，事务之间出现会等锁的情况。等锁的超时时间由 TiDB 的innodb_lock_wait_timeout 参数来定义，这个是 SQL 语句层面的最大允许等锁时间，即一个 SQL 语句期望加锁，但锁一直获取不到，超过这个时间，TiDB 不会再尝试加锁，会向客户端返回相应的报错信息。

可通过查看 TiDB 日志查看报错信息：

当出现等锁超时的情况时，会向客户端返回下述报错信息：

```
ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction
```

处理建议：

- 如果出现的次数非常频繁，建议从业务逻辑的角度来进行调整。

10.2.7.3.4 TTL manager has timed out

除了有不能超出 GC 时间的限制外，悲观锁的 TTL 有上限，默认为 1 小时，所以执行时间超过 1 小时的悲观事务有可能提交失败。这个超时时间由 TiDB 参数 performance.max-txn-ttl 指定。

可通过查看 TiDB 日志查看报错信息：

当悲观锁的事务执行时间超过 TTL 时，会出现下述报错：

```
TTL manager has timed out, pessimistic locks may expire, please commit or rollback this
    ↪ transaction
```

处理建议：

- 当遇到该报错时，建议确认下业务逻辑是否可以进行优化，如将大事务拆分为小事务。在未使用大事务的前提下，大事务可能会触发 TiDB 的事务限制。
- 可适当调整相关参数，使其符合事务要求。

10.2.7.3.5 Deadlock found when trying to get lock

死锁是指两个或两个以上的事务在执行过程中，由于竞争资源而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去，将永远在互相等待。此时，需要终止其中一个事务使其能够继续推进下去。

TiDB 在使用悲观锁的情况下，多个事务之间出现了死锁，必定有一个事务 abort 来解开死锁。在客户端层面行为和 MySQL 一致，在客户端返回表示死锁的 Error 1213。如下：

```
[err="[executor:1213]Deadlock found when trying to get lock; try restarting transaction"]
```

处理建议：

- 如果难以确认产生死锁的原因，对于 v5.1 及以后的版本，建议尝试查询 `INFORMATION_SCHEMA.DEADLOCKS` 或 `INFORMATION_SCHEMA.CLUSTER_DEADLOCKS` 系统表来获取死锁的等待链信息。详情请参考死锁错误小节和DEADLOCKS 表文档。
- 如果死锁出现非常频繁，需要调整业务代码来降低发生概率。

10.2.8 数据索引一致性报错

当执行事务或执行`ADMIN CHECK [TABLE|INDEX]` 命令时，TiDB 会对数据索引的一致性进行检查。如果检查发现 record key-value 和 index key-value 不一致，即存储行数据的键值对和存储其对应索引的键值对之间不一致（例如多索引或缺索引），TiDB 会报数据索引一致性错误，并在日志文件中打印相关错误日志。

本文对数据索引一致性的报错信息进行了说明，并提供了一些绕过检查的方法。遇到报错时，你可以前往 AskTUG 论坛，与社区用户交流；如果是订阅用户，请联系 PingCAP 服务与支持。

10.2.8.1 错误样例解读

当数据索引不一致时，你可以通过查看 TiDB 的报错信息了解行数据和索引数据在哪一项不一致，或者查看相关错误日志进行判断。

10.2.8.1.1 执行事务中的报错

本节列出了 TiDB 在执行事务过程中可能出现的数据索引不一致性的报错，并通过举例对这些信息的含义进行了解释。

Error 8133

```
ERROR 8133 (HY000): data inconsistency in table: t, index: k2, index-count:1 != record-count:0
```

上述错误表明，对于表 `t` 中的 `k2` 索引，表中索引数量为 `1`，行记录的数量为 `0`，数量不一致。

Error 8138

```
ERROR 8138 (HY000): writing inconsistent data in table: t, expected-values:{KindString green} !=
↪ record-values:{KindString GREEN}
```

上述错误表明，事务试图写入的行值有误。即将写入的数据中，编码后的行数据与编码前的原始数据不符。

Error 8139

```
ERROR 8139 (HY000): writing inconsistent data in table: t, index: i1, index-handle:4 != record-
↪ handle:3, index: tables.mutation{key:kv.Key{0x74, 0x80, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x49, 0
↪ x5f, 0x69, 0x80, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x1, 0x1, 0x68, 0x65, 0x6c, 0x6c, 0x6f, 0x0, 0
↪ x0, 0x0, 0xfc, 0x1, 0x68, 0x65, 0x6c, 0x6c, 0x6f, 0x0, 0x0, 0x0, 0xfc, 0x3, 0x80, 0x0, 0x0, 0
↪ x0, 0x0, 0x0, 0x0, 0x4}, flags:0x0, value:[]uint8{0x30}, indexID:1}, record: tables.mutation{
↪ key:kv.Key{0x74, 0x80, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x49, 0x5f, 0x72, 0x80, 0x0, 0x0, 0x0, 0
↪ x0, 0x0, 0x0, 0x3}, flags:0xd, value:[]uint8{0x80, 0x0, 0x2, 0x0, 0x0, 0x0, 0x1, 0x2, 0x5, 0x0
↪ , 0xa, 0x0, 0x68, 0x65, 0x6c, 0x6c, 0x6f, 0x68, 0x65, 0x6c, 0x6c, 0x6f}, indexID:0}
```

上述错误表明，即将写入的数据中，handle（即行数据的 key）值不一致。对于表 t 中的 i1 索引，该事务即将写入的某行在索引键值对中的 handle 值是 4，在行记录键值对中的 handle 值是 3。这行数据将不会被写入。

Error 8140

```
ERROR 8140 (HY000): writing inconsistent data in table: t, index: i2, col: c1, indexed-value:{
↪ KindString hellp} != record-value:{KindString hello}
```

上述错误表明，事务试图写入的行和索引的值不一致。对于表 t 中的 i2 索引，该事务即将写入的某行在索引键值对中的数据是 hellp，在行记录键值对中的数据是hello。这行数据将不会被写入。

Error 8141

```
ERROR 8141 (HY000): assertion failed: key: 748000000000000405f7201330000000000000000f8, assertion
↪ : NotExist, start_ts: 430590532931813377, existing start ts: 430590532931551233, existing
↪ commit ts: 430590532931551234
```

上述错误表明，事务提交时断言失败。根据数据索引一致的假设，TiDB 断言 key 7480000000000000405
↪ f7201330000000000000f8 不存在，提交事务时发现该 key 存在，是由 start ts 为 430590532931551233 的事务写入的。TiDB 会将该 key 的 MVCC (Multi-Version Concurrency Control) 历史输出到日志。

10.2.8.1.2 Admin check 中的报错

本节列出了执行ADMIN CHECK [TABLE|INDEX] 系列语句时 TiDB 可能出现的数据索引不一致报错，并通过举例对这些信息的含义进行了解释。

Error 8003

```
ERROR 8003 (HY000): table count 3 != index(idx)count 2
```

上述错误表明，在ADMIN CHECK 语句所执行的表上有 3 个行键值对，但只有 2 个索引键值对。

Error 8134

```
ERROR 8134 (HY000): data inconsistency in table: t, index: c2, col: c2, handle: "2", index-values
↪ :"KindInt64 13" != record-values:"KindInt64 12", compare err:<nil>
```

上述错误表明，对于表 t 中的 c2 索引，handle 为 2 的行对应的索引键值对中列 c2 的值是 13，行记录键值对中列 c2 的值是 12。

Error 8223

```
ERROR 8223 (HY000): data inconsistency in table: t2, index: i1, handle: {hello, hello}, index-
↪ values:"" != record-values:"handle: {hello, hello}, values: [KindString hello KindString hello
↪ ]"
```

上述错误表明，`index-values` 为空，`record-values` 不为空，说明不存在对应的索引，但存在对应的行。

10.2.8.2 报错处理

发生报错时，不要自行处理，请从 PingCAP 官方或 TiDB 社区获取支持。如果业务急需跳过此类报错，可以使用以下方法绕过检查。

10.2.8.2.1 改写 SQL

如果只有某一条 SQL 语句报错，可以尝试将其改写为其它等价的 SQL 形式，以使用不同的执行算子来尝试绕过。

10.2.8.2.2 关闭错误检查

对于事务执行中出现的一些报错，可以使用以下方法绕过检查：

- 对于错误代码为 8138、8139 和 8140 的错误，可以通过设置 `set @@tidb_enable_mutation_checker=0` 跳过检查。
- 对于错误代码为 8141 的错误，可以通过设置 `set @@tidb_txn_assertion_level=OFF` 跳过检查。

> **注意：**
>
> 关闭 `tidb_enable_mutation_checker` 和 `tidb_txn_assertion_level` 开关会关闭对所有 SQL 语句的对应检查。

对于其它错误代码，包括执行`ADMIN CHECK [TABLE|INDEX]` 系列语句或执行事务中的报错，由于数据中已经存在不一致，无法跳过对应的检查。

10.3 故障诊断方法

10.3.1 SQL 诊断

SQL 诊断功能是在 TiDB 4.0 版本中引入的特性，用于提升 TiDB 问题定位的效率。TiDB 4.0 版本以前，用户需要使用不同的工具以异构的方式获取不同信息。新的 SQL 诊断系统对这些离散的信息进行了整体设计，它整合系统各个维度的信息，通过系统表的方式向上层提供一致的接口，提供监控汇总与自动诊断，方便用户查询集群信息。

SQL 诊断共分三大块：

- 集群信息表：TiDB 4.0 诊断系统添加了集群信息表，为原先离散的各实例信息提供了统一的获取方式。它将整个集群的集群拓扑、硬件信息、软件信息、内核参数、监控、系统信息、慢查询、语句、日志完全整合在表中，让用户能够统一使用 SQL 进行查询。

- 集群监控表：TiDB 4.0 诊断系统添加了集群监控系统表，所有表都在 `metrics_schema` 中，可以通过 SQL 语句来查询监控信息。比起原先的可视化监控，SQL 查询监控允许用户对整个集群的所有监控进行关联查询，并对比不同时间段的结果，迅速找出性能瓶颈。由于 TiDB 集群的监控指标数量较大，SQL 诊断还提供了监控汇总表，让用户能够更便捷地从众多监控中找出异常的监控项。
- 自动诊断：尽管用户可以手动执行 SQL 来查询集群信息表、集群监控表与汇总表来定位问题，但自动诊断可以快速对常见异常进行定位。SQL 诊断基于已有的集群信息表和监控表，提供了与之相关的诊断结果表与诊断汇总表来执行自动诊断。

10.3.1.1 集群信息表

集群信息表将一个集群中的所有实例的信息都汇聚在一起，让用户仅通过一条 SQL 就能查询整个集群相关信息。集群信息表列表如下：

- 集群拓扑表`information_schema.cluster_info` 用于获取集群当前的拓扑信息，以及各个实例的版本、版本对应的 Git Hash、各实例的启动时间、各实例的运行时间。
- 集群配置表`information_schema.cluster_config` 用于获取集群当前所有实例的配置。对于 TiDB 4.0 之前的版本，用户必须逐个访问各个实例的 HTTP API 才能获取这些配置信息。
- 集群硬件表`information_schema.cluster_hardware` 用于快速查询集群硬件信息。
- 集群负载表`information_schema.cluster_load` 用于查询集群不同实例以及不同硬件类型的负载信息。
- 内核参数表`information_schema.cluster_systeminfo` 用于查询集群不同实例的内核配置信息。目前支持查询 sysctl 的信息。
- 集群日志表`information_schema.cluster_log` 用于集群日志查询，通过将查询条件下推到各个实例，降低日志查询对集群的影响，性能影响小于等 grep 命令。

TiDB 4.0 之前的系统表，只能查看当前实例信息，TiDB 4.0 实现了对应的集群表，可以在单个 TiDB 实例上拥有整个集群的全局视图。这些表目前都位于 `information_schema` 中，查询方式与其他 `information_schema` 系统表一致。

10.3.1.2 集群监控表

为了能够动态地观察并对比不同时间段的集群情况，TiDB 4.0 诊断系统添加了集群监控系统表。所有监控表都在 `metrics_schema` 中，可以通过 SQL 的方式查询监控信息。SQL 查询监控允许用户对整个集群的所有监控进行关联查询，并对比不同时间段的结果，迅速找出性能瓶颈。

- `information_schema.metrics_tables`：由于目前添加的系统表数量较多，因此用户可以通过该表查询这些监控表的相关元信息。

由于 TiDB 集群的监控指标数量较大，因此 TiDB 4.0 提供以下监控汇总表：

- 监控汇总表`information_schema.metrics_summary` 用于汇总所有监控数据，以提升用户排查各监控指标的效率。
- 监控汇总并按 label 聚合表`information_schema.metrics_summary_by_label` 同样用于汇总所有监控数据，但该表会对各项监控的不同的 label 进行聚合统计。

10.3.1.3 自动诊断

以上集群信息表和集群监控表均需要用户手动执行 SQL 语句来排查集群问题。TiDB 4.0 中添加了自动诊断功能，根据已有的基础信息表，提供诊断相关的系统表，使诊断自动执行。自动诊断相关的系统表如下：

- 诊断结果表`information_schema.inspection_result` 用于展示对系统的诊断结果。诊断是惰性触发，使用 `select * from inspection_result` 会触发所有诊断规则对系统进行诊断，并在结果中展示系统中的故障或风险。
- 诊断汇总表`information_schema.inspection_summary` 用于对特定链路或模块的监控进行汇总，用户可以根据整个模块或链路的上下文来排查定位问题。

10.3.2 Statement Summary Tables

针对 SQL 性能相关的问题，MySQL 在 `performance_schema` 提供了 statement summary tables，用来监控和统计 SQL。例如其中的一张表 `events_statements_summary_by_digest`，提供了丰富的字段，包括延迟、执行次数、扫描行数、全表扫描次数等，有助于用户定位 SQL 问题。

为此，从 4.0.0-rc.1 版本开始，TiDB 在 `information_schema`（而不是 `performance_schema`）中提供与 `events_statements_summary_by_digest` 功能相似的系统表：

- `statements_summary`
- `statements_summary_history`
- `cluster_statements_summary`
- `cluster_statements_summary_history`

本文将详细介绍这些表，以及如何利用它们来排查 SQL 性能问题。

10.3.2.1 `statements_summary`

`statements_summary` 是 `information_schema` 里的一张系统表，它把 SQL 按 SQL digest 和 plan digest 分组，统计每一组的 SQL 信息。

此处的 SQL digest 与 slow log 里的 SQL digest 一样，是把 SQL 规一化后算出的唯一标识符。SQL 的规一化会忽略常量、空白符、大小写的差别。即语法一致的 SQL 语句，其 digest 也相同。

例如：

```
SELECT * FROM employee WHERE id IN (1, 2, 3) AND salary BETWEEN 1000 AND 2000;
select * from EMPLOYEE where ID in (4, 5) and SALARY between 3000 and 4000;
```

归一化后都是：

```
select * from employee where id in (...) and salary between ? and ?;
```

此处的 plan digest 是把执行计划规一化后算出的唯一标识符。执行计划的规一化会忽略常量的差别。由于相同的 SQL 可能产生不同的执行计划，所以可能分到多个组，同一个组内的执行计划是相同的。

`statements_summary` 用于保存 SQL 监控指标聚合后的结果。一般来说，每一项监控指标都包含平均值和最大值。例如执行延时对应 `AVG_LATENCY` 和 `MAX_LATENCY` 两个字段，分别是平均延时和最大延时。

为了监控指标的即时性，statements_summary 里的数据定期被清空，只展现最近一段时间内的聚合结果。清空周期由系统变量 tidb_stmt_summary_refresh_interval 设置。如果刚好在清空之后进行查询，显示的数据可能很少。

以下为查询 statements_summary 的部分结果：

```
  SUMMARY_BEGIN_TIME: 2020-01-02 11:00:00
    SUMMARY_END_TIME: 2020-01-02 11:30:00
           STMT_TYPE: Select
         SCHEMA_NAME: test
              DIGEST: 0611cc2fe792f8c146cc97d39b31d9562014cf15f8d41f23a4938ca341f54182
         DIGEST_TEXT: select * from employee where id = ?
         TABLE_NAMES: test.employee
         INDEX_NAMES: NULL
         SAMPLE_USER: root
          EXEC_COUNT: 3
         SUM_LATENCY: 1035161
         MAX_LATENCY: 399594
         MIN_LATENCY: 301353
         AVG_LATENCY: 345053
   AVG_PARSE_LATENCY: 57000
   MAX_PARSE_LATENCY: 57000
 AVG_COMPILE_LATENCY: 175458
 MAX_COMPILE_LATENCY: 175458
 ...........
             AVG_MEM: 103
             MAX_MEM: 103
            AVG_DISK: 65535
            MAX_DISK: 65535
   AVG_AFFECTED_ROWS: 0
          FIRST_SEEN: 2020-01-02 11:12:54
           LAST_SEEN: 2020-01-02 11:25:24
   QUERY_SAMPLE_TEXT: select * from employee where id=3100
    PREV_SAMPLE_TEXT:
         PLAN_DIGEST: f415b8d52640b535b9b12a9c148a8630d2c6d59e419aad29397842e32e8e5de3
                PLAN:  Point_Get_1     root    1       table:employee, handle:3100
```

> **注意：**
>
> 在 TiDB 中，statement summary tables 中字段的时间单位是纳秒 (ns)，而 MySQL 中的时间单位是皮秒 (ps)。

10.3.2.2 statements_summary_history

statements_summary_history 的表结构与 statements_summary 完全相同，用于保存历史时间段的数据。通过历史数据，可以排查过去出现的异常，也可以对比不同时间的监控指标。

字段 SUMMARY_BEGIN_TIME 和 SUMMARY_END_TIME 代表历史时间段的开始时间和结束时间。

10.3.2.3 statements_summary_evicted

statements_summary 表的容量受 tidb_stmt_summary_max_stmt_count 配置控制，内部使用 LRU 算法，一旦接收到的 SQL 种类超过了 tidb_stmt_summary_max_stmt_count，表中最久未被命中的记录就会被驱逐出表。TiDB 引入了 statements_summary_evicted 表，该表记录了各个时段被驱逐 SQL 语句的具体数量。

只有当 SQL 语句被 statement summary 表驱逐的时候，statements_summary_evicted 表的内容才会更新。statements_summary_evicted 表记录发生驱逐的时间段和被驱逐 SQL 的数量。

10.3.2.4 statement summary 的 cluster 表

statements_summary、statements_summary_history 和 statements_summary_evicted 仅显示单台 TiDB server 的 statement summary 数据。若要查询整个集群的数据，需要查询 cluster_statements_summary、cluster_statements_summary_history 或 cluster_statements_summary_evicted 表。

cluster_statements_summary 显示各台 TiDB server 的 statements_summary 数据，cluster_statements_summary_history 显示各台 TiDB server 的 statements_summary_history 数据，而 cluster_statements_summary_evicted 则显示各台 TiDB server 的 statements_summary_evicted 数据。这三张表用字段 INSTANCE 表示 TiDB server 的地址，其他字段与 statements_summary、statements_summary_history 和 statements_summary_evicted 表相同。

10.3.2.5 参数配置

以下系统变量用于控制 statement summary：

- tidb_enable_stmt_summary：是否打开 statement summary 功能。1 代表打开，0 代表关闭，默认打开。statement summary 关闭后，系统表里的数据会被清空，下次打开后重新统计。经测试，打开后对性能几乎没有影响。
- tidb_stmt_summary_refresh_interval：statements_summary 的清空周期，单位是秒 (s)，默认值是 1800。
- tidb_stmt_summary_history_size：statements_summary_history 保存每种 SQL 的历史的数量，也是 statements_summary_evicted 的表容量，默认值是 24。
- tidb_stmt_summary_max_stmt_count：statement summary tables 保存的 SQL 种类数量，默认 3000 条。当 SQL 种类超过该值时，会移除最近没有使用的 SQL。这些 SQL 将会被 DIGEST 为 NULL 的行和 statements_summary_evicted 统计记录。DIGEST 为 NULL 的行数据在TiDB Dashboard SQL 语句分析列表页面 中显示为 Others。
- tidb_stmt_summary_max_sql_length：字段 DIGEST_TEXT 和 QUERY_SAMPLE_TEXT 的最大显示长度，默认值是 4096。
- tidb_stmt_summary_internal_query：是否统计 TiDB 的内部 SQL。1 代表统计，0 代表不统计，默认不统计。

> 注意：

> 当一种 SQL 因为达到 `tidb_stmt_summary_max_stmt_count` 限制要被移除时，TiDB 会移除该 SQL 语句种类在所有时间段的数据。因此，即使一个时间段内的 SQL 种类数量没有达到上限，显示的 SQL 语句数量也会比实际的少。如遇到该情况，对性能也有一些影响，建议调大 `tidb_stmt_summary_max_stmt_count` 的值。

statement summary 配置示例如下：

```
set global tidb_stmt_summary_max_stmt_count = 3000;
set global tidb_enable_stmt_summary = true;
set global tidb_stmt_summary_refresh_interval = 1800;
set global tidb_stmt_summary_history_size = 24;
```

以上配置生效后，`statements_summary` 每 30 分钟清空一次，`statements_summary_history` 最多保存 3000 种 SQL 种类的数据，每种类型的 SQL 保存最近出现过的 24 个时间段的数据。`statements_summary_evicted` 保存最近 24 个发生了 evict 的时间段记录；`statements_summary_evicted` 则以 30 分钟为一个记录周期，表容量为 24 个时间段。

> 注意：
>
> - 假设某种 SQL 每分钟都出现，那 `statements_summary_history` 中会保存这种 SQL 最近 12 个小时的数据。但如果某种 SQL 只在每天 00:00 ~ 00:30 出现，则 `statements_summary_history` 中会保存这种 SQL 24 个时间段的数据，每个时间段的间隔都是 1 天，所以会有这种 SQL 最近 24 天的数据。
> - `tidb_stmt_summary_history_size`、`tidb_stmt_summary_max_stmt_count`、`tidb_stmt_summary_max_sql_length` 这些配置都影响内存占用，建议根据实际情况调整（取决于 SQL 大小、SQL 数量、机器配置）不宜设置得过大。内存大小可通过 `tidb_stmt_summary_history_size * tidb_stmt_summary_max_stmt_count * tidb_stmt_summary_max_sql_length * 3` 来进行估算。

10.3.2.5.1 为 statement summary 设定合适的大小

在系统运行一段时间后（视系统负载而定），可以查看 `statements_summary` 表检查是否发生了 evict，例如：

```
select @@global.tidb_stmt_summary_max_stmt_count;
select count(*) from information_schema.statements_summary;
```

```
+-----------------------------------------+
| @@global.tidb_stmt_summary_max_stmt_count |
+-----------------------------------------+
| 3000                                    |
+-----------------------------------------+
1 row in set (0.001 sec)
```

```
+----------+
| count(*) |
+----------+
|     3001 |
+----------+
1 row in set (0.001 sec)
```

可以发现 statements_summary 表已经满了。再查看 statements_summary_evicted 表检查 evict 的数据。

```
select * from information_schema.statements_summary_evicted;
```

```
+---------------------+---------------------+---------------+
| BEGIN_TIME          | END_TIME            | EVICTED_COUNT |
+---------------------+---------------------+---------------+
| 2020-01-02 16:30:00 | 2020-01-02 17:00:00 |            59 |
+---------------------+---------------------+---------------+
| 2020-01-02 16:00:00 | 2020-01-02 16:30:00 |            45 |
+---------------------+---------------------+---------------+
2 row in set (0.001 sec)
```

由上可知，对最多 59 种 SQL 发生了 evict，也就是说最少应将 statement summary 的容量增大至 59 条记录。

10.3.2.6 目前的限制

由于 statement summary tables 默认都存储在内存中，TiDB server 重启后，statement summary 会全部丢失。

为解决该问题，TiDB v6.6.0 实验性地引入了statement summary 持久化功能，该功能默认为关闭。开启该功能后，历史数据不再存储在内存内，而是直接写入磁盘。TiDB server 重启后，历史数据也依然可用。

10.3.2.7 持久化 statements summary

> **警告：**
>
> statements summary 持久化目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现 bug，请在 GitHub 上提 issue 反馈。

如目前的限制一节所描述，默认情况下 statements summary 只在内存中维护，一旦 TiDB server 发生重启，所有 statements summary 数据都会丢失。自 v6.6.0 起，TiDB 实验性地提供了配置项tidb_stmt_summary_enable_persistent 来允许用户控制是否开启 statements summary 持久化。

如果要开启 statements summary 持久化，可以在 TiDB 配置文件中添加如下配置：

```
[instance]
tidb_stmt_summary_enable_persistent = true
```

```
### 以下配置为默认值，可根据需求调整。
### tidb_stmt_summary_filename = "tidb-statements.log"
### tidb_stmt_summary_file_max_days = 3
### tidb_stmt_summary_file_max_size = 64 # MiB
### tidb_stmt_summary_file_max_backups = 0
```

开启 statements summary 持久化后，内存中只维护当前的实时数据，不再维护历史数据。历史数据生成后直接被写入磁盘文件，写入周期参考参数配置一节所描述的 tidb_stmt_summary_refresh_interval。后续针对 statements_summary_history 或 cluster_statements_summary_history 表的查询将结合内存和磁盘两处数据返回结果。

> 注意：
>
> - 当开启持久化后，由于不再于内存中维护历史数据，因此参数配置一节所描述的 tidb_stmt_summary_history_size 将不再生效，而是由 tidb_stmt_summary_file_max_days、tidb_stmt_summary_file_max_size 和 tidb_stmt_summary_file_max_backups 这三项配置来决定历史数据在磁盘上的保留数量和时间。
> - tidb_stmt_summary_refresh_interval 取值越小，数据写入到磁盘就越实时，但写入磁盘的冗余数据也会随之增多。

10.3.2.8 排查示例

下面用两个示例问题演示如何利用 statement summary 来排查。

10.3.2.8.1 SQL 延迟比较大，是不是服务端的问题？

例如客户端显示 employee 表的点查比较慢，那么可以按 SQL 文本来模糊查询：

```
SELECT avg_latency, exec_count, query_sample_text
    FROM information_schema.statements_summary
    WHERE digest_text LIKE 'select * from employee%';
```

结果如下，avg_latency 是 1 ms 和 0.3 ms，在正常范围，所以可以判定不是服务端的问题，继而排查客户端或网络问题。

```
+-------------+------------+----------------------------------------+
| avg_latency | exec_count | query_sample_text                      |
+-------------+------------+----------------------------------------+
|     1042040 |          2 | select * from employee where name='eric' |
|      345053 |          3 | select * from employee where id=3100     |
+-------------+------------+----------------------------------------+
2 rows in set (0.00 sec)
```

10.3.2.8.2 哪类 SQL 的总耗时最高？

假如上午 10:00 到 10:30 的 QPS 明显下降，可以从历史表中找出当时耗时最高的三类 SQL：

```
SELECT sum_latency, avg_latency, exec_count, query_sample_text
    FROM information_schema.statements_summary_history
    WHERE summary_begin_time='2020-01-02 10:00:00'
    ORDER BY sum_latency DESC LIMIT 3;
```

结果显示以下三类 SQL 的总延迟最高，所以这些 SQL 需要重点优化。

```
+-------------+-------------+-----------+-----------------------------------------------------------------------------
    ↪
| sum_latency | avg_latency | exec_count | query_sample_text
    ↪                                                          |
+-------------+-------------+-----------+-----------------------------------------------------------------------------
    ↪
|     7855660 |     1122237 |          7 | select avg(salary) from employee where company_id=2013
    ↪                  |
|     7241960 |     1448392 |          5 | select * from employee join company on employee.
    ↪ company_id=company.id |
|     2084081 |     1042040 |          2 | select * from employee where name='eric'
    ↪                                |
+-------------+-------------+-----------+-----------------------------------------------------------------------------
    ↪
3 rows in set (0.00 sec)
```

10.3.2.9 表的字段介绍

10.3.2.9.1 `statements_summary` 字段介绍

下面介绍 `statements_summary` 表中各个字段的含义。

SQL 的基础信息：

- `STMT_TYPE`：SQL 语句的类型
- `SCHEMA_NAME`：执行这类 SQL 的当前 schema
- `DIGEST`：这类 SQL 的 digest
- `DIGEST_TEXT`：规一化后的 SQL
- `QUERY_SAMPLE_TEXT`：这类 SQL 的原 SQL 语句，多条语句只取其中一条
- `TABLE_NAMES`：SQL 中涉及的所有表，多张表用，分隔
- `INDEX_NAMES`：SQL 中使用的索引名，多个索引用，分隔
- `SAMPLE_USER`：执行这类 SQL 的用户名，多个用户名只取其中一个
- `PLAN_DIGEST`：执行计划的 digest
- `PLAN`：原执行计划，多条语句只取其中一条的执行计划
- `BINARY_PLAN`：以二进制格式编码后的原执行计划，存在多条语句时，只取其中一条语句的执行计划。用 `select tidb_decode_binary_plan('xxx...')` SQL 语句可以解析出具体的执行计划。

- PLAN_CACHE_HITS：这类 SQL 语句命中 plan cache 的总次数
- PLAN_IN_CACHE：这类 SQL 语句的上次执行是否命中了 plan cache

执行时间相关的信息：

- SUMMARY_BEGIN_TIME：当前统计的时间段的开始时间
- SUMMARY_END_TIME：当前统计的时间段的结束时间
- FIRST_SEEN：这类 SQL 的首次出现时间
- LAST_SEEN：这类 SQL 的最后一次出现时间

在 TiDB server 上的执行数据：

- EXEC_COUNT：这类 SQL 的总执行次数
- SUM_ERRORS：执行过程中遇到的 error 的总数
- SUM_WARNINGS：执行过程中遇到的 warning 的总数
- SUM_LATENCY：这类 SQL 的总延时
- MAX_LATENCY：这类 SQL 的最大延时
- MIN_LATENCY：这类 SQL 的最小延时
- AVG_LATENCY：这类 SQL 的平均延时
- AVG_PARSE_LATENCY：解析器的平均延时
- MAX_PARSE_LATENCY：解析器的最大延时
- AVG_COMPILE_LATENCY：优化器的平均延时
- MAX_COMPILE_LATENCY：优化器的最大延时
- AVG_MEM：使用的平均内存，单位 byte
- MAX_MEM：使用的最大内存，单位 byte
- AVG_DISK：使用的平均硬盘空间，单位 byte
- MAX_DISK：使用的最大硬盘空间，单位 byte

和 TiKV Coprocessor Task 相关的字段：

- SUM_COP_TASK_NUM：发送 Coprocessor 请求的总数
- MAX_COP_PROCESS_TIME：cop-task 的最大处理时间
- MAX_COP_PROCESS_ADDRESS：执行时间最长的 cop-task 所在地址
- MAX_COP_WAIT_TIME：cop-task 的最大等待时间
- MAX_COP_WAIT_ADDRESS：等待时间最长的 cop-task 所在地址
- AVG_PROCESS_TIME：SQL 在 TiKV 的平均处理时间
- MAX_PROCESS_TIME：SQL 在 TiKV 的最大处理时间
- AVG_WAIT_TIME：SQL 在 TiKV 的平均等待时间
- MAX_WAIT_TIME：SQL 在 TiKV 的最大等待时间
- AVG_BACKOFF_TIME：SQL 遇到需要重试的错误时在重试前的平均等待时间
- MAX_BACKOFF_TIME：SQL 遇到需要重试的错误时在重试前的最大等待时间
- AVG_TOTAL_KEYS：Coprocessor 扫过的 key 的平均数量
- MAX_TOTAL_KEYS：Coprocessor 扫过的 key 的最大数量
- AVG_PROCESSED_KEYS：Coprocessor 处理的 key 的平均数量。相比 avg_total_keys，avg_processed_keys 不包含 MVCC 的旧版本。如果 avg_total_keys 和 avg_processed_keys 相差很大，说明旧版本比较多

- MAX_PROCESSED_KEYS：Coprocessor 处理的 key 的最大数量

和事务相关的字段：

- AVG_PREWRITE_TIME：prewrite 阶段消耗的平均时间
- MAX_PREWRITE_TIME prewrite 阶段消耗的最大时间
- AVG_COMMIT_TIME：commit 阶段消耗的平均时间
- MAX_COMMIT_TIME：commit 阶段消耗的最大时间
- AVG_GET_COMMIT_TS_TIME：获取 commit_ts 的平均时间
- MAX_GET_COMMIT_TS_TIME：获取 commit_ts 的最大时间
- AVG_COMMIT_BACKOFF_TIME：commit 时遇到需要重试的错误时在重试前的平均等待时间
- MAX_COMMIT_BACKOFF_TIME：commit 时遇到需要重试的错误时在重试前的最大等待时间
- AVG_RESOLVE_LOCK_TIME：解决事务的锁冲突的平均时间
- MAX_RESOLVE_LOCK_TIME：解决事务的锁冲突的最大时间
- AVG_LOCAL_LATCH_WAIT_TIME：本地事务等待的平均时间
- MAX_LOCAL_LATCH_WAIT_TIME：本地事务等待的最大时间
- AVG_WRITE_KEYS：写入 key 的平均数量
- MAX_WRITE_KEYS：写入 key 的最大数量
- AVG_WRITE_SIZE：写入的平均数据量，单位 byte
- MAX_WRITE_SIZE：写入的最大数据量，单位 byte
- AVG_PREWRITE_REGIONS：prewrite 涉及的平均 Region 数量
- MAX_PREWRITE_REGIONS：prewrite 涉及的最大 Region 数量
- AVG_TXN_RETRY：事务平均重试次数
- MAX_TXN_RETRY：事务最大重试次数
- SUM_BACKOFF_TIMES：这类 SQL 遇到需要重试的错误后的总重试次数
- BACKOFF_TYPES：遇到需要重试的错误时的所有错误类型及每种类型重试的次数，格式为 类型:次数。如有多种错误则用 , 分隔，例如 txnLock:2,pdRPC:1
- AVG_AFFECTED_ROWS：平均影响行数
- PREV_SAMPLE_TEXT：当 SQL 是 COMMIT 时，该字段为 COMMIT 的前一条语句；否则该字段为空字符串。当 SQL 是 COMMIT 时，按 digest 和 prev_sample_text 一起分组，即不同 prev_sample_text 的 COMMIT 也会分到不同的行

10.3.2.9.2 statements_summary_evicted 字段介绍

- BEGIN_TIME: 记录的开始时间；
- END_TIME: 记录的结束时间；
- EVICTED_COUNT：在记录的时间段内 evict 了多少种 SQL。

10.3.3 TiDB Dashboard Top SQL 页面

TiDB Dashboard 的 Top SQL 功能允许你可视化地监控和探索数据库中各个 SQL 语句在执行过程中的 CPU 开销情况，从而对数据库性能问题进行优化和处理。Top SQL 持续收集各个 TiDB 及 TiKV 实例每秒的实时 CPU 负载等数据（按 SQL 类型分别统计），并存储至多 30 天。你可以通过 Top SQL 展示的图表及表格快速分析某个 TiDB 或 TiKV 实例在某段时间中高 CPU 负载是来自于哪些 SQL 语句。

Top SQL 具有以下功能：

- 通过图表及表格，可视化地展示 CPU 开销最多的 5 类 SQL 语句。
- 展示每秒请求数、平均延迟、查询计划等详细执行信息。
- 支持统计所有正在执行、尚未执行完毕的 SQL 语句。
- 支持查看集群中指定 TiDB 及 TiKV 实例的情况。

10.3.3.1 推荐适用场景

Top SQL 适用于分析性能问题。以下列举了一些典型的 Top SQL 适用场景：

- 通过监控图发现集群中有个别 TiKV 实例的 CPU 非常高，期望了解 CPU 热点来自于哪些 SQL 语句，以便对其进行优化、更好地利用上分布式资源。
- 集群整体 CPU 占用率非常高、数据库查询缓慢，期望快速知悉目前哪些 SQL 语句开销了最多的 CPU 资源，以便对它们进行优化。
- 集群整体 CPU 占用率突然发生了显著变化，期望了解变化前后主要的 SQL 资源开销区别。
- 分析集群当前最消耗资源的 SQL 语句情况，希望对它们进行优化以便降低硬件开支。

Top SQL 不能用于解答与性能无关的问题，例如数据正确性或异常崩溃问题。

当前 Top SQL 仍然处于早期阶段，功能正在持续加强。以下列举了一些目前暂不支持的场景，供参考：

- 暂时不支持分析 Top 5 以外 SQL 语句的开销情况（如多业务混合时）。
- 暂时不支持按 User、Database 等不同维度分析 Top N SQL 语句的开销情况。
- 暂时不支持分析并非由于 CPU 负载高导致的数据库性能问题，例如锁冲突。

10.3.3.2 访问页面

你可以通过以下任一方式访问 Top SQL 页面：

- 登录 TiDB Dashboard 后，在左侧导航栏中点击 Top SQL

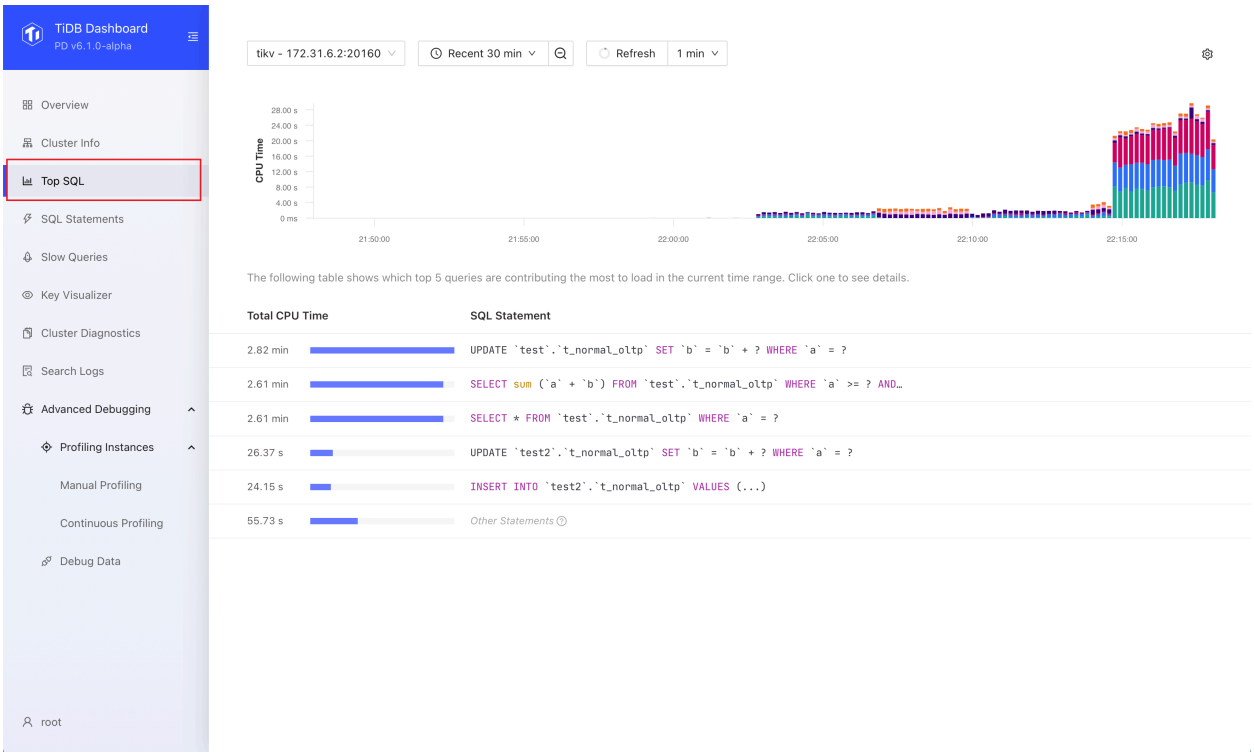


图 83: Top SQL

- 在浏览器中访问 http://127.0.0.1:2379/dashboard/#/topsql（将 127.0.0.1:2379 替换为实际 PD 实例地址和端口）。

10.3.3.3 启用 Top SQL

> **注意：**
>
> 要使用 Top SQL，你需要使用 TiUP（v1.9.0 及以上版本）或 TiDB Operator（v1.3.0 及以上版本）部署或升级集群。如果你已经使用旧版本 TiUP 或 TiDB Operator 进行了集群升级，请参见FAQ 进行处理。

Top SQL 开启后会对集群性能产生轻微的影响（平均 3% 以内），因此该功能默认关闭。你可以通过以下方法启用 Top SQL：

1. 访问Top SQL 页面。
2. 点击打开设置 (Open Settings)。在右侧设置 (Settings) 页面，将启用特性 (Enable Feature) 下方的开关打开。
3. 点击保存 (Save)。

你仅能看到开启功能之后的 CPU 负载细节情况，在开启功能之前的 CPU 负载细节无法在界面上呈现。另外，数据有至多 1 分钟左右的延迟，因此你可能需要等待片刻才能看到数据。

除了通过图形化界面以外，你也可以配置 TiDB 系统变量tidb_enable_top_sql 来启用 Top SQL 功能：

```
SET GLOBAL tidb_enable_top_sql = 1;
```

10.3.3.4 使用 Top SQL

以下是 Top SQL 的常用步骤：

1. 访问Top SQL 页面。
2. 选择一个你想要观察负载的具体 TiDB 或 TiKV 实例。

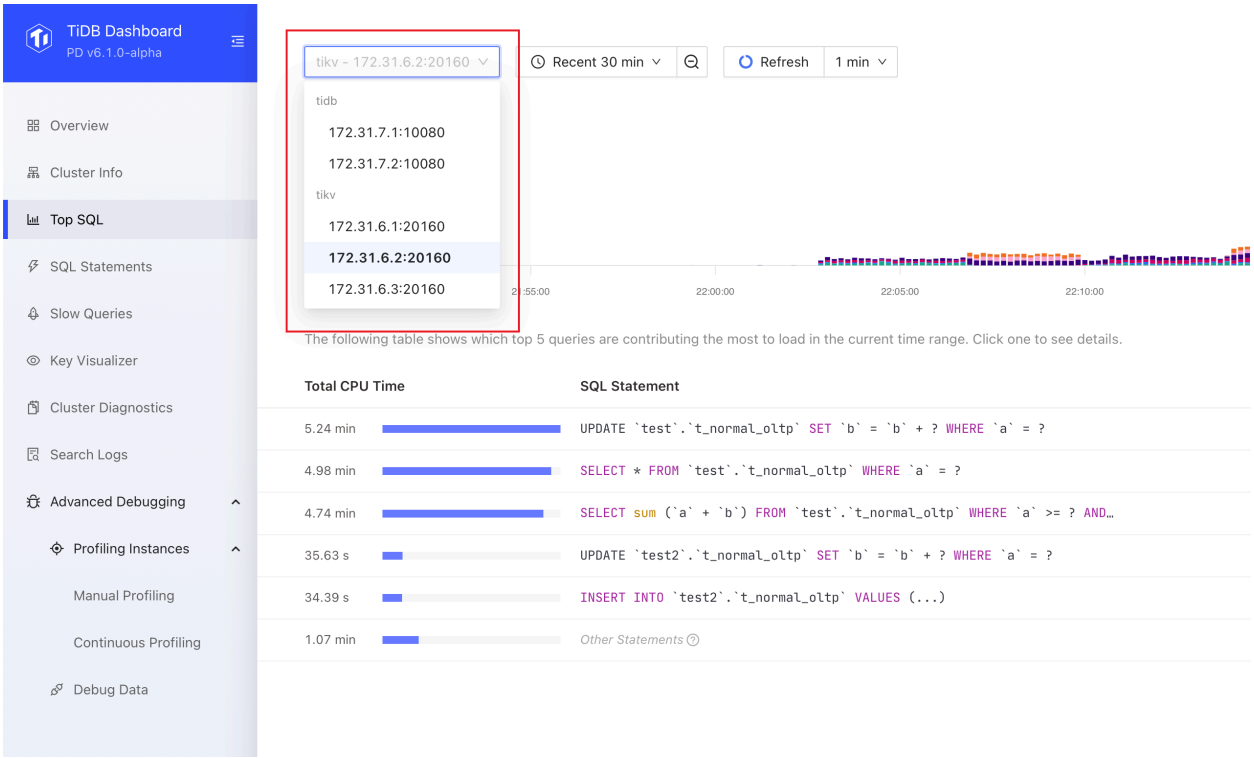


图 84: 选择实例

如果你不知道要观察哪一个 TiDB 或 TiKV 实例，可以选择任意一个实例。在集群 CPU 开销非常不均衡的情况下，你可以首先通过 Grafana 中的 CPU 监控来确定具体期望观察的实例。

3. 观察 Top SQL 呈现的 Top 5 图表及表格。

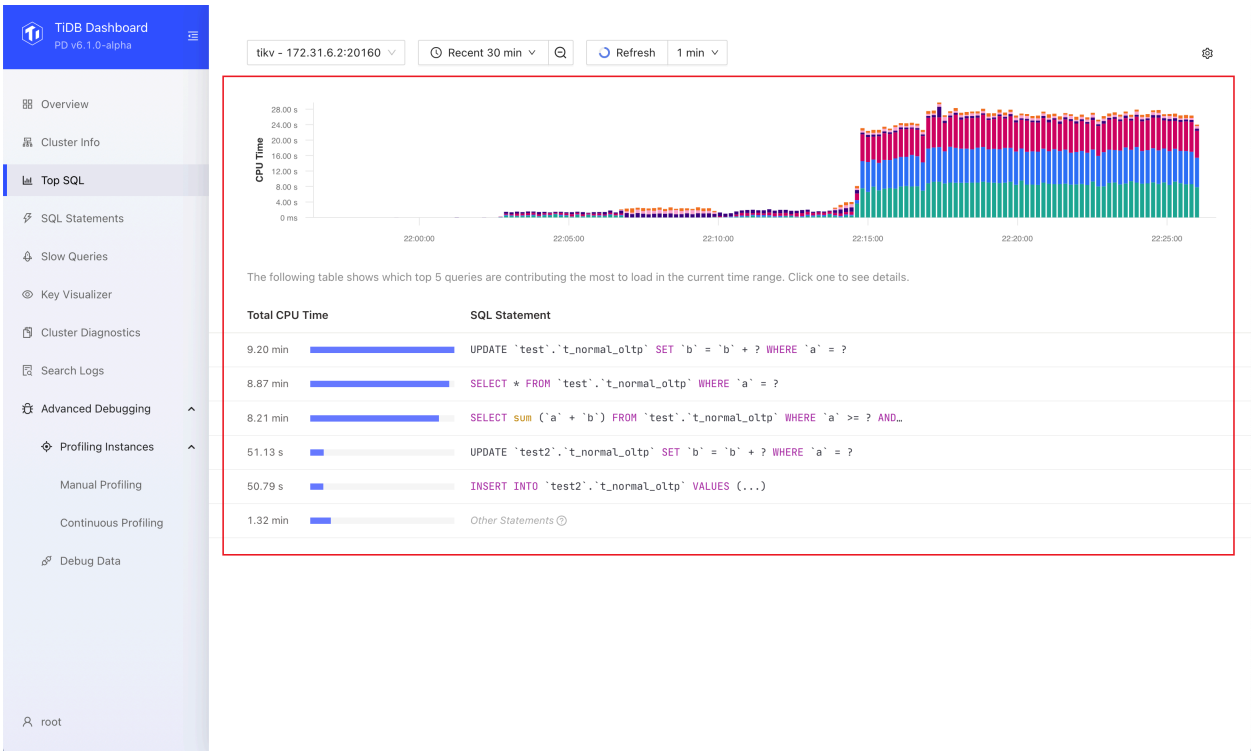


图 85: 图表表格

柱状图中色块的大小代表了 SQL 语句在该时刻消耗的 CPU 资源的多少，不同颜色区分了不同类型的 SQL 语句。大多数情况下，你都应该关注图表中相应时间范围内 CPU 资源开销较大的 SQL 语句。

4. 点击表格中的某一个 SQL 语句后，可以展开查看该语句不同执行计划的执行情况，例如 Call/sec（平均每秒请求数）、Scan Indexes/sec（平均每秒扫描索引数）等。

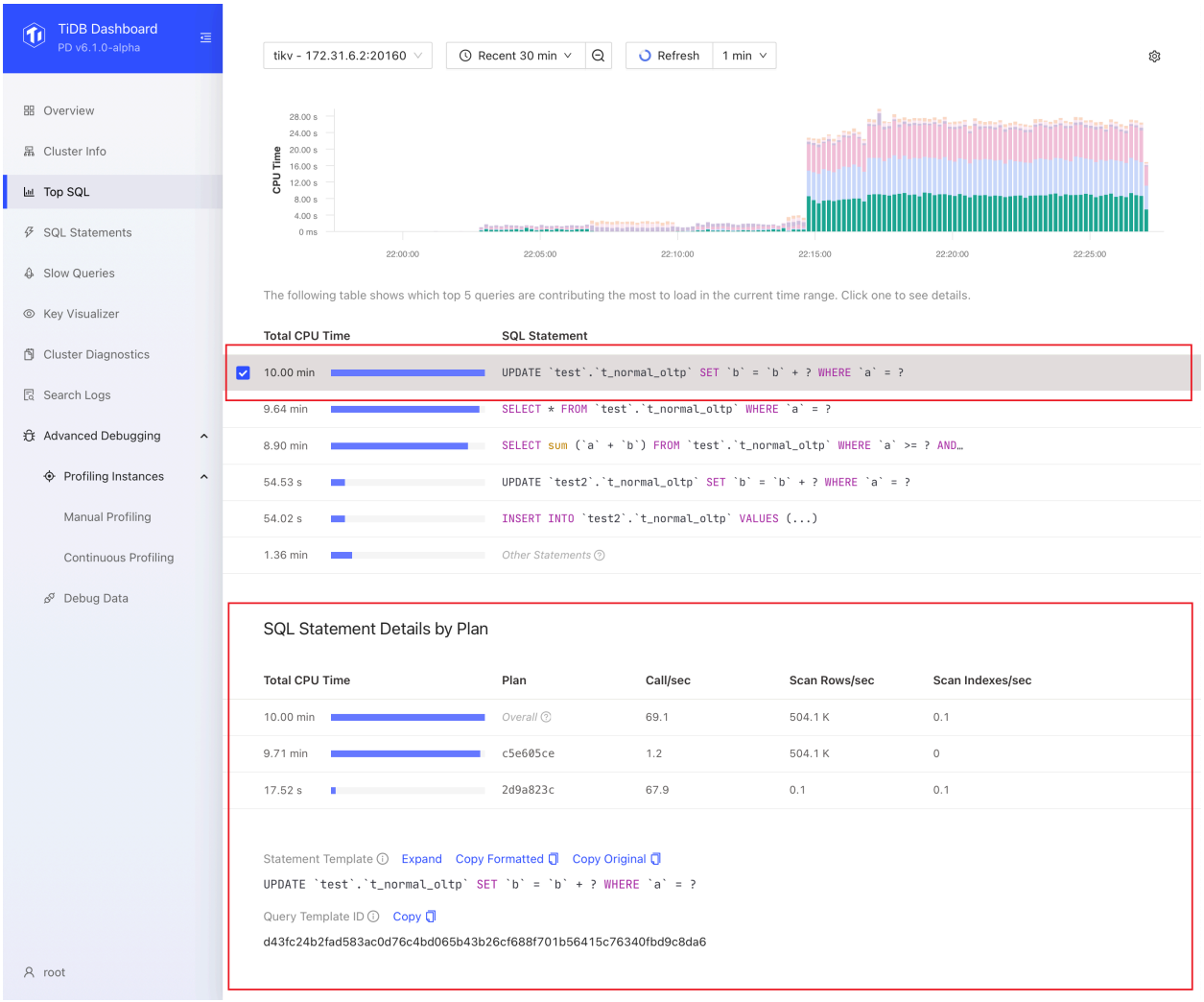


图 86: 详情

5. 基于这些初步线索，进一步在SQL 语句分析或慢查询界面中了解该 SQL 语句开销大量 CPU 资源、或扫大量数据的详细原因。

除此以外：

- 你可以在时间选择器中调整时间范围，或在图表中框选一个时间范围，来更精确、细致地观察问题。更小的时间范围将能提供细节更丰富的数据，数据最高精度可达 1 秒。



图 87: 修改时间范围

- 如果图表中显示的数据已过时，你可以点击刷新 (Refresh) 按钮，或在刷新 (Refresh) 下拉列表中选择自动刷新。

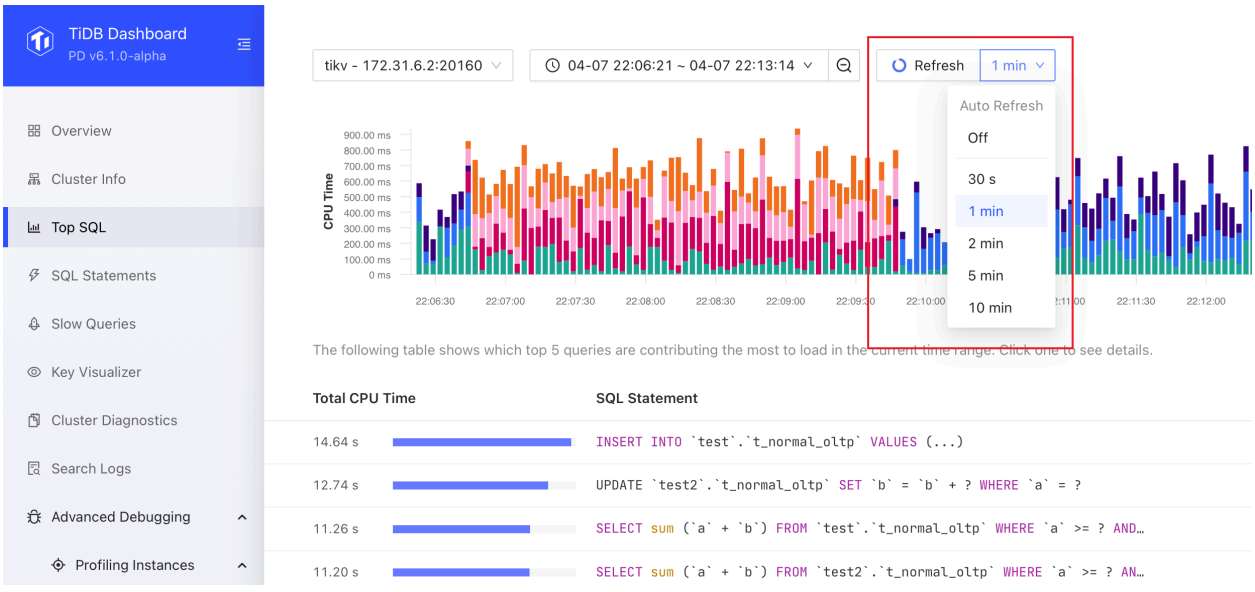


图 88: 刷新

10.3.3.5 停用 Top SQL

你可以通过以下步骤停用该功能：

1. 访问Top SQL 页面。
2. 点击右上角齿轮按钮打开设置界面，将启用特性 (Enable Feature) 下方的开关关闭。
3. 点击保存 (Save)。
4. 在弹出的关闭 Top SQL 功能 (Disable Top SQL Feature) 对话框中，点击确认 (Disable)。

除了通过图形化界面以外，你也可以配置 TiDB 系统变量tidb_enable_top_sql 来停用 Top SQL 功能：

```
SET GLOBAL tidb_enable_top_sql = 0;
```

10.3.3.6 常见问题

1. 界面上提示 “集群中未启动必要组件 NgMonitoring” 无法启用功能

请参见TiDB Dashboard FAQ。

2. 该功能开启后对集群是否有性能影响？

该功能对集群性能有轻微影响。根据我们的测算，该功能对集群的平均性能影响小于 3%。

3. 该功能目前是什么状态？

该功能是正式特性，在生产环境中可用。

4. 界面中显示的其他语句（Other Statements）是什么意思？

其他所有非 Top 5 语句产生的 CPU 开销或执行情况都会被统计在该项中。你可以基于这一项了解 Top 5 的 SQL 语句开销在整体所有 SQL 语句的 CPU 开销中的比例。

5. Top SQL 展示的 CPU 开销总和与进程的实际 CPU 开销是什么关系？

它们之间有很强的相关性，但不完全一致。以 TiKV 为例，TiKV 的 CPU 开销还可能来自于其他副本的数据同步写入，这些开销不会被计入 Top SQL。但总的来说，Top SQL 中开销比例比较大的 SQL 语句实际的 CPU 资源开销也确实会更大。

6. Top SQL 图表的纵坐标是什么意思？

代表消耗 CPU 资源的多少。消耗资源越多的 SQL 语句，该值越大。在绝大多数情况下，你都不需要关心纵坐标具体数值的含义。

7. 还没有执行完毕的 SQL 语句会被统计到吗？

会。Top SQL 图表上所展示的每一时刻 CPU 开销比例即为这一时刻所有正在运行的 SQL 语句的 CPU 开销情况。

10.3.4 定位消耗系统资源多的查询

TiDB 会将执行时间超过tidb_expensive_query_time_threshold 限制（默认值为 60s），或使用内存超过tidb_mem_quota_query 限制（默认值为 1 GB）的语句输出到tidb-server 日志文件（默认文件为 “tidb.log”）中，用于在语句执行结束前定位消耗系统资源多的查询语句（以下简称为 expensive query），帮助用户分析和解决语句执行的性能问题。

注意，expensive query 日志和慢查询日志的区别是，慢查询日志是在语句执行完后才打印，expensive query 日志可以将正在执行的语句的相关信息打印出来。当一条语句在执行过程中达到资源使用阈值时（执行时间/使用内存量），TiDB 会即时将这条语句的相关信息写入日志。

10.3.4.1 Expensive query 日志示例

```
[2020/02/05 15:32:25.096 +08:00] [WARN] [expensivequery.go:167] [expensive_query] [cost_time
    ↪ =60.008338935s] [wait_time=0s] [request_count=1] [total_keys=70] [process_keys=65] [
    ↪ num_cop_tasks=1] [process_avg_time=0s] [process_p90_time=0s] [process_max_time=0s] [
    ↪ process_max_addr=10.0.1.9:20160] [wait_avg_time=0.002s] [wait_p90_time=0.002s] [
    ↪ wait_max_time=0.002s] [wait_max_addr=10.0.1.9:20160] [stats=t:pseudo] [conn_id=60026] [
    ↪ user=root] [database=test] [table_ids="[122]"] [txn_start_ts=414420273735139329] [mem_max
    ↪ ="1035 Bytes (1.0107421875 KB)"] [sql="insert into t select sleep(1) from t"]
```

10.3.4.2 字段含义说明

基本字段：

- `cost_time`：日志打印时语句已经花费的执行时间。
- `stats`：语句涉及到的表或索引使用的统计信息版本。值为 pseudo 时表示无可用统计信息，需要对表或索引进行 analyze。
- `table_ids`：语句涉及到的表的 ID。
- `txn_start_ts`：事务的开始时间戳，也是事务的唯一 ID，可以用这个值在 TiDB 日志中查找事务相关的其他日志。
- `sql`：SQL 语句。

和内存使用相关的字段：

- `mem_max`：日志打印时语句已经使用的内存空间。该项使用两种单位标识内存使用量，分别为 Bytes 以及易于阅读的自适应单位（比如 MB、GB 等）。

和 SQL 执行的用户相关的字段：

- `user`：执行语句的用户名。
- `conn_id`：用户的连接 ID，可以用类似 `con:60026` 的关键字在 TiDB 日志中查找该连接相关的其他日志。
- `database`：执行语句时使用的 database。

和 TiKV Coprocessor Task 相关的字段：

- `wait_time`：该语句在 TiKV 的等待时间之和，因为 TiKV 的 Coprocessor 线程数是有限的，当所有的 Coprocessor 线程都在工作的时候，请求会排队；当队列中有某些请求耗时很长的时候，后面的请求的等待时间都会增加。
- `request_count`：该语句发送的 Coprocessor 请求的数量。
- `total_keys`：Coprocessor 扫过的 key 的数量。
- `processed_keys`：Coprocessor 处理的 key 的数量。与 total_keys 相比，processed_keys 不包含 MVCC 的旧版本。如果 processed_keys 和 total_keys 相差很大，说明旧版本比较多。
- `num_cop_tasks`：该语句发送的 Coprocessor 请求的数量。
- `process_avg_time`：Coprocessor 执行 task 的平均执行时间。
- `process_p90_time`：Coprocessor 执行 task 的 P90 分位执行时间。

- `process_max_time`：Coprocessor 执行 task 的最长执行时间。
- `process_max_addr`：task 执行时间最长的 Coprocessor 所在地址。
- `wait_avg_time`：Coprocessor 上 task 的等待时间。
- `wait_p90_time`：Coprocessor 上 task 的 P90 分位等待时间。
- `wait_max_time`：Coprocessor 上 task 的最长等待时间。
- `wait_max_addr`：task 等待时间最长的 Coprocessor 所在地址。

10.3.5 使用 PLAN REPLAYER 保存和恢复集群现场信息

用户在定位排查 TiDB 集群问题时，经常需要提供系统和查询计划相关的信息。为了帮助用户更方便地获取相关信息，更高效地排查集群问题，TiDB 在 v5.3.0 中引入了 `PLAN REPLAYER` 命令，用于“一键”保存和恢复现场问题的相关信息，提升查询计划问题诊断的效率，同时方便将问题归档管理。

`PLAN REPLAYER` 主要功能如下：

- 导出排查现场 TiDB 集群的相关信息，导出为 ZIP 格式的文件用于保存。
- 在任意 TiDB 集群上导入另一 TiDB 集群现场信息的 ZIP 文件。

10.3.5.1 使用 PLAN REPLAYER 导出集群信息

你可以使用 `PLAN REPLAYER` 来保存 TiDB 集群的现场信息。导出接口如下：

```
PLAN REPLAYER DUMP EXPLAIN [ANALYZE] sql-statement;
```

TiDB 根据 `sql-statement` 整理出以下集群现场信息：

- TiDB 版本信息
- TiDB 配置信息
- TiDB Session 系统变量
- TiDB 执行计划绑定信息（SQL Binding）
- `sql-statement` 中所包含的表结构
- `sql-statement` 中所包含表的统计信息
- `EXPLAIN [ANALYZE] sql-statement` 的结果

> **注意：**
>
> `PLAN REPLAYER` 不会导出表中数据

10.3.5.1.1 PLAN REPLAYER 导出示例

```
use test;
create table t(a int, b int);
insert into t values(1,1), (2, 2), (3, 3);
analyze table t;
```

```
plan replayer dump explain select * from t;
```

PLAN REPLAYER DUMP 会将以上信息打包整理成 ZIP 文件，并返回文件标识作为执行结果。该文件为一次性文件，被下载后 TiDB 会将其删除。

> **注意：**
>
> ZIP 文件最多会在 TiDB 集群中保存一个小时，超时后 TiDB 会将其删除。

```
MySQL [test]> plan replayer dump explain select * from t;
```

```
+------------------------------------------------------------------+
| Dump_link                                                        |
+------------------------------------------------------------------+
| replayer_JOGvpu4t7dssySqJfTtS4A==_1635750890568691080.zip |
+------------------------------------------------------------------+
1 row in set (0.015 sec)
```

你同样可以通过tidb_last_plan_replayer_token 这个会话变量来获取上一次 PLAN REPLAYER dump 执行的结果。

```
SELECT @@tidb_last_plan_replayer_token;
```

```
| @@tidb_last_plan_replayer_token                          |
+----------------------------------------------------------+
| replayer_Fdamsm3C7ZiPJ-LQqgVjkA==_1663304195885090000.zip |
+----------------------------------------------------------+
1 row in set (0.00 sec)
```

对于多条 SQL 的情况，你可以通过文件的方式来获取 plan replayer dump 的结果，多条 SQL 语句在文件中以；进行分隔。

```
plan replayer dump explain 'sqls.txt';
```

```
SELECT @@tidb_last_plan_replayer_token;
```

```
+----------------------------------------------------------+
| @@tidb_last_plan_replayer_token                          |
+----------------------------------------------------------+
| replayer_LEDKg8sb-K0u24QesiH8ig==_1663226556509182000.zip |
+----------------------------------------------------------+
1 row in set (0.00 sec)
```

因为 MySQL Client 无法下载文件，所以需要通过 TiDB HTTP 接口和文件标识下载文件：

```
http://${tidb-server-ip}:${tidb-server-status-port}/plan_replayer/dump/${file_token}
```

其中，`${tidb-server-ip}:${tidb-server-status-port}` 是集群中任意 TiDB server 的地址。示例如下：

```
curl http://127.0.0.1:10080/plan_replayer/dump/replayer_JOGvpu4t7dssySqJfTtS4A==
    ↪ _1635750890568691080.zip > plan_replayer.zip
```

10.3.5.2 使用 PLAN REPLAYER 导入集群信息

> **警告：**
>
> PLAN REPLAYER 在一个 TiDB 集群上导入另一集群的现场信息，会修改导入集群的 TiDB Session 系统变量、执行计划绑定信息、表结构和统计信息。

有 PLAN REPLAYER 导出的 ZIP 文件后，用户便可以通过 PLAN REPLAYER 导入接口在任意 TiDB 集群上恢复另一集群地现场信息。语法如下：

```
PLAN REPLAYER LOAD 'file_name';
```

以上语句中，`file_name` 为要导入的 ZIP 文件名。

示例如下：

```
PLAN REPLAYER LOAD 'plan_replayer.zip';
```

导入完毕后，该 TiDB 集群就载入了所需要的表结构、统计信息等其他影响构造 Plan 所需要的信息。你可以通过以下方式查看执行计划以及验证统计信息:

```
mysql> desc t;
+-------+---------+------+------+---------+-------+
| Field | Type    | Null | Key  | Default | Extra |
+-------+---------+------+------+---------+-------+
| a     | int(11) | YES  |      | NULL    |       |
| b     | int(11) | YES  |      | NULL    |       |
+-------+---------+------+------+---------+-------+
2 rows in set (0.01 sec)

mysql> explain select * from t where a = 1 or b =1;
+--
    ↪ ----------------------+---------+-----------+---------------+----------------------------------+
    ↪
| id                    | estRows | task      | access object | operator info
    ↪                       |
```

```
+--
    ↪ ----------------------+---------+-----------+---------------+--------------------------------+
    ↪
| TableReader_7           | 0.01    | root      |               | data:Selection_6
    ↪                      |
| └─Selection_6           | 0.01    | cop[tikv] |               | or(eq(test.t.a, 1), eq(test.t.
    ↪ b, 1)) |
|   └─TableFullScan_5     | 6.00    | cop[tikv] | table:t       | keep order:false, stats:pseudo
    ↪        |
+--
    ↪ ----------------------+---------+-----------+---------------+--------------------------------+
    ↪
3 rows in set (0.00 sec)

mysql> show stats_meta;
+---------+------------+----------------+---------------------+--------------+-----------+
| Db_name | Table_name | Partition_name | Update_time         | Modify_count | Row_count |
+---------+------------+----------------+---------------------+--------------+-----------+
| test    | t          |                | 2022-08-26 15:52:07 |            3 |         6 |
+---------+------------+----------------+---------------------+--------------+-----------+
1 row in set (0.04 sec)
```

加载并还原所需现场后，即可在该现场诊断和改进执行计划。

10.3.5.3 使用 PLAN REPLAYER CAPTURE 抓取目标计划

在用户定位 TiDB 执行计划的部分场景中，目标 SQL 语句与目标计划可能仅在查询中偶尔出现，无法使用 PLAN REPLAYER 直接抓取。此时你可以使用 PLAN REPLAYER CAPTURE 来帮助定向抓取目标 SQL 语句与目标计划的优化器信息。

PLAN REPLAYER CAPTURE 主要功能如下：

- 在 TiDB 集群内部提前注册目标 SQL 语句与执行计划的 Digest，并开始匹配目标查询。
- 当目标查询匹配成功时，直接抓取其优化器相关信息，导出为 ZIP 格式的文件用于保存。
- 针对匹配到的每组 SQL 和执行计划，信息只抓取一次。
- 通过系统表显示正在进行的匹配任务，以及生成的文件。
- 定时清理历史文件。

10.3.5.3.1 开启 PLAN REPLAYER CAPTURE

PLAN REPLAYER CAPTURE 功能通过系统变量tidb_enable_plan_replayer_capture 控制。要开启 PLAN REPLAYER
↪ CAPTURE，将变量值设为 ON。

10.3.5.3.2 使用 PLAN REPLAYER CAPTURE 功能

你可以通过以下方式向 TiDB 集群注册目标 SQL 语句和计划的 Digest:

```
PLAN REPLAYER CAPTURE 'sql_digest' 'plan_digest';
```

当你的目标 SQL 语句对应多种执行计划，且你想抓取所有执行计划时，你可以通过以下 SQL 语句一键注册:

```
PLAN REPLAYER CAPTURE 'sql_digest' '*';
```

10.3.5.3.3 查看 PLAN REPLAYER CAPTURE 抓取任务

你可以通过以下方式查看集群中目前正在工作的 PLAN REPLAYER CAPTURE 的抓取任务:

```
mysql> PLAN PLAYER CAPTURE 'example_sql' 'example_plan';
Query OK, 1 row affected (0.01 sec)

mysql> SELECT * FROM mysql.plan_replayer_task;
+-------------+--------------+---------------------+
| sql_digest  | plan_digest  | update_time         |
+-------------+--------------+---------------------+
| example_sql | example_plan | 2023-01-28 11:58:22 |
+-------------+--------------+---------------------+
1 row in set (0.01 sec)
```

10.3.5.3.4 查看 PLAN REPLAYER CAPTURE 抓取结果

当 PLAN REPLAYER CAPTURE 成功抓取到结果后，可以通过以下 SQL 语句查看用于下载的文件标识:

```
mysql> SELECT * FROM mysql.plan_replayer_status;
+--
    ↪ ----------------------------------------------------------------+-----------------------------------
    ↪
| sql_digest                                                         | plan_digest
    ↪                                                                | origin_sql | token
    ↪                                                                | update_time         | fail_reason |
    ↪  instance        |
+--
    ↪ ----------------------------------------------------------------+-----------------------------------
    ↪
| 086e3fbd2732f7671c17f299d4320689deeeb87ba031240e1e598a0ca14f808c | 042
    ↪ de2a6652a6d20afc629ff90b8507b7587a1c7e1eb122c3e0b808b1d80cc02 |            |
    ↪ replayer_Utah4nkz2sIEzkks7tIRog==_1668746293523179156.zip | 2022-11-18 12:38:13 | NULL
    ↪        | 172.16.4.4:4022 |
| b5b38322b7be560edb04f33f15b15a885e7c6209a22b56b0804622e397199b54 | 1770
    ↪ efeb3f91936e095f0344b629562bf1b204f6e46439b7d8f842319297c3b5 |            |
    ↪ replayer_Z2mUXNHDjU_WBmGdWQqifw==_1668746293560115314.zip | 2022-11-18 12:38:13 | NULL
    ↪        | 172.16.4.4:4022 |
```

```
| 96d00c0b3f08795fe94e2d712fa1078ab7809faf4e81d198f276c0dede818cf9 | 8892
    ↪ f74ac2a42c2c6b6152352bc491b5c07c73ac3ed66487b2c990909bae83e8 |             |
    ↪ replayer_RZcRHJB7BaCccxFfOIAhWg==_1668746293578282450.zip | 2022-11-18 12:38:13 | NULL
    ↪         | 172.16.4.4:4022 |
+--
    ↪ ----------------------------------------------------------------+-------------------------------------
    ↪
3 rows in set (0.00 sec)
```

下载 PLAN REPLAYER CAPTURE 的文件方法与 PLAN REPLAYER 相同，请参考PLAN REPLAYER 导出示例。

> 注意：
>
> PLAN REPLAYER CAPTURE 的结果文件最多会在 TiDB 集群中保存一个小时，超时后 TiDB 会将其删除。

10.3.5.4 使用 PLAN REPLAYER CONTINUOUS CAPTURE

开启 PLAN REPLAYER CONTINUOUS CAPTURE 功能后，TiDB 将以 SQL DIGEST 和 PLAN DIGEST 为维度异步地将业务 SQL 语句以 PLAN REPLAYER 的方式进行记录，对于相同 DIGEST 的 SQL 语句与执行计划，PLAN REPLAYER CONTINUOUS ↪ CAPTURE 不会重复记录。

10.3.5.4.1 开启 PLAN REPLAYER CONTINUOUS CAPTURE

PLAN REPLAYER CONTINUOUS CAPTURE 功能通过系统变量tidb_enable_plan_replayer_continuous_capture 控制。要开启 PLAN REPLAYER CONTINUOUS CAPTURE，将变量值设为 ON。

10.3.5.4.2 查看 PLAN REPLAYER CONTINUOUS CAPTURE 抓取结果

查看 PLAN REPLAYER CONTINUOUS CAPTURE 抓取结果的方法同查看 PLAN REPLAYER CAPTURE 抓取结果。

10.4 支持资源

如果你在使用 TiDB 的过程中遇到了问题，你可以通过以下方式向 PingCAP 或 TiDB 社区寻求帮助：

- 从 PingCAP 获取支持（需要订阅 TiDB 企业版）：
  - 提交工单
- 从 TiDB 社区寻求帮助：
  - AskTUG 论坛
  - Stack Overflow（在 #tidb 标签下提问）
- 了解 TiDB 的实现和设计
  - TiDB Internals 论坛

11 性能调优

11.1 优化手册

11.1.1 TiDB 性能优化概述

本文介绍性能优化的基本概念，比如用户响应时间、吞吐和数据库时间，以及性能优化的通用流程。

11.1.1.1 用户响应时间和数据库时间

11.1.1.1.1 用户响应时间

用户响应时间是指应用系统为用户返回请求结果所消耗的时间。一个典型的用户请求的处理时序图如下，包含了用户和应用系统的网络延迟、应用的处理时间、应用和数据库的交互时的网络延迟和数据库的服务时间等。用户响应时间受到请求链路上各个子系统的影响，比如网络延迟和带宽、系统并发用户数和请求类型、服务器 CPU 和 IO 资源使用率等。要对整个系统进行有效的优化，你需要先定位用户响应时间的瓶颈。

你可以通过以下公式计算指定时间范围 (ΔT) 内总的用户响应时间：

ΔT 时间内总的用户响应时间 = 平均 TPS (Transactions Per Second) x 用户平均响应时间 x ΔT。

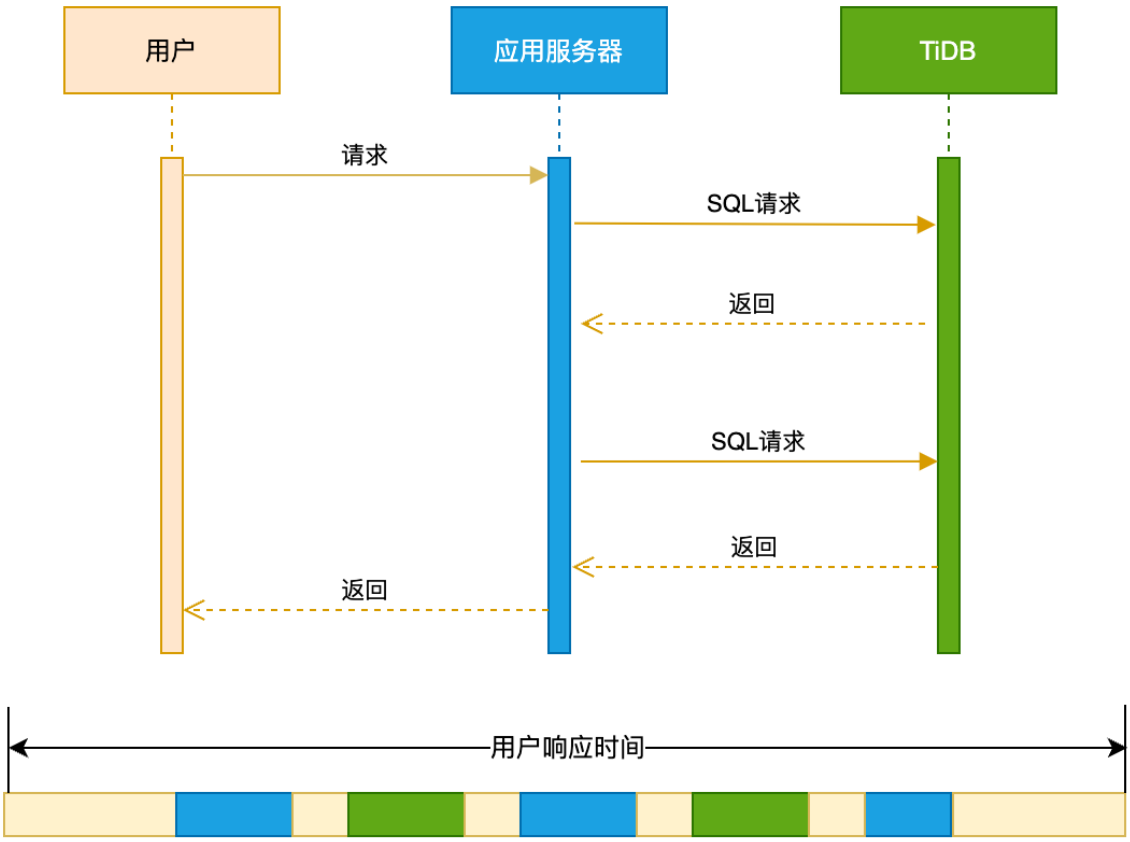


图 89: 用户响应时间

11.1.1.1.2 数据库时间

数据库时间是指数据库系统提供服务的时间，ΔT 时间内的数据库时间为数据库并发处理所有应用请求的时间总和。

你可以通过以下任一方式计算数据库时间：

- 方式一：通过 QPS 乘以平均 query 延迟乘以 ΔT，即 `DB Time in ΔT = QPS × avg latency × ΔT`
- 方式二：通过平均活跃会话数乘以 ΔT，即 `DB Time in ΔT = avg active connections × ΔT`
- 方式三：通过 TiDB 内部的 Prometheus 指标 tidb_server_tokens 计算，即 `ΔT DB Time = rate(tidb_server_tokens )× ΔT`

11.1.1.2 用户响应时间和系统吞吐的关系

用户响应时间包含完成用户请求的服务时间、排队时间和并发等待时间，即：

```
User Response time = Service time + Queuing delay + Coherency delay
```

- Service Time（完成用户请求的服务时间）：系统处理请求时需要消耗某种资源的时间，比如数据库完成一次 SQL 请求需要消耗的 CPU 时间。
- Queuing delay（排队延迟时间）：系统处理请求时为了等待某种资源的服务，在队列中等待调度的时间。
- Coherency delay（并发等待延迟）：系统处理请求时为了访问共享资源，需要和其他并发的任务进行通信和协作的时间。

系统吞吐指系统每秒完成的请求数量。用户响应时间和吞吐通常是反比倒数的关系。随着吞吐的上升，系统资源利用率上升，请求服务的排队延迟会随之上升，当资源利用率超过某个拐点，排队延迟会急剧上升。

例如，对于运行 OLTP 负载的数据库系统，当 CPU 利用率超过 65% 之后，CPU 的排队调度延迟会明显上升。因为系统的并发请求不是完全独立的，请求之间存在共享资源的协同和争用，比如不同的数据库请求可能对同样的数据有互斥的加锁操作。当资源利用率上升时，排队和调度延迟上升，这将导致持有的共享资源无法及时释放，反过来延长了其他任务对共享资源的等待时间。

11.1.1.3 性能优化流程

性能优化流程包含以下 6 个步骤：

1. 定义优化目标
2. 建立性能基线
3. 定位用户响应时间的瓶颈
4. 提出优化方案，预估每种方案的收益、风险和成本
5. 实施优化
6. 评估优化结果

一个性能优化项目，经常需要对步骤 2 到 6 进行多次循环，才能达到优化的目标。

11.1.1.3.1 第 1 步：定义优化目标

不同类型系统优化目标不同。例如，对于一个金融核心的 OLTP 系统，优化目标可能是降低交易的长尾延迟；对于一个财务结算系统，优化目标可能是更充分利用硬件资源，缩短批量结算任务时间。

一个好的优化目标应该是容易量化的，比如：

- 好的优化目标：” 业务高峰期上午 9 点到 10 点，转账交易的 p99 延迟需要小于 200 毫秒 “
- 差的优化目标：” 系统太慢了没有响应，需要优化 “

定义一个清晰的优化目标有助于指导后续的性能优化工作。

11.1.1.3.2 第 2 步：建立性能基线

为了高效地进行性能优化，你需要采集当前的性能数据以建立性能基线。需要采集的性能数据通常包含以下内容：

- 用户响应时间的平均值和长尾值、应用系统的吞吐

- 数据库时间、Query 延迟和 QPS 等数据库性能数据。

  TiDB 针对不同维度的性能数据进行了完善的测量和存储，例如慢日志、Top SQL、持续性能分析功能和流量可视化等。此外，你还可以对存储在 Prometheus 中的时序指标数据进行历史回溯和对比。

- 资源使用率，包含 CPU、IO 和网络等资源
- 配置信息，比如应用系统、数据库和操作系统的配置

11.1.1.3.3 第 3 步：定位用户响应时间的瓶颈

基于性能基线的数据，定位或者推测用户响应时间的瓶颈。

现实中的应用程序往往没有对用户请求的链路进行完整的测量和记录，因此你无法通过应用程序对用户响应时间进行自上而下有效的分解。

与之相反的是，数据库内部对于 query 延迟和吞吐等性能指标记录非常完善。基于数据库时间，你可以判断用户响应时间的瓶颈是否在数据库中。

- 如果瓶颈不在数据库中，需要借助数据库外部搜集的资源利用率，或者对应用程序进行 Profile，以确定数据库外部的瓶颈。常见场景包括应用程序或者代理服务器资源不足，应用程序存在串行点无法充分利用硬件资源等。
- 如果瓶颈存在数据库中，你可以通过数据库完善的调优工具进行数据库内部性能分析和诊断。常见场景包括存在慢 SQL、应用程序使用数据库的方式不合理、数据库存在读写热点等。

具体的分析诊断方法和工具，请参考性能优化方法。

11.1.1.3.4 第 4 步：提出优化方案，评估每种方案的收益、风险和成本

通过性能分析确定系统瓶颈点之后，根据实际情况提出低成本、低风险、并能获得最大的收益的优化方案。

根据阿姆达尔定律，性能优化的最大收益，取决于优化的部分在整个系统的占比。因此，你需要根据性能数据，确认系统瓶颈和相应的占比，预估瓶颈解决或者优化之后的收益。

需要注意的是，即使某个方案针对最大瓶颈点的优化潜在收益最大，也需要同时评估该方案的风险和成本。例如：

- 对于资源过载的系统，最直接的优化方案是扩容，但是实际中可能因为扩容方案成本太高而无法被采纳。
- 当某个业务模块里的一个慢 SQL 导致整个模块的响应时间很慢时，升级到数据库新版本的方案可以解决这个慢 SQL 问题，但是同时可能影响原来没有问题的模块，因此该方案可能存在潜在的高风险。一个低风险的方案是不升级数据库版本，直接改写现有慢 SQL，在当前数据库版本中解决该问题。

11.1.1.3.5 第 5 步：实施优化

综合考量收益、风险和成本，选定一种或者多种优化方案进行实施，并对生产系统的变更进行周全的准备和详细的记录。

为了降低风险和验证优化方案的收益，建议在测试环境和准生产环境对变更的内容进行验证和完整的回归。例如，针对一个查询业务的慢 SQL，如果选定的优化方案是新建索引优化查询的访问路径，你需要确保新的索引不会在现有的数据插入业务中引入明显的写入热点，导致其他业务变慢。

11.1.1.3.6 第 6 步：评估优化结果

实施优化之后，需要评估优化结果。

- 如果达到优化目标，整个优化项目顺利完成。
- 如果未达到优化目标，你需要重复步骤 2 到 6，直到达到优化目标。

达到优化目标之后，为了应对业务的增长，你可能还需要进一步做好系统的容量规划。

11.1.2 TiDB 性能分析和优化

本文介绍了基于数据库时间的系统优化方法，以及如何利用 TiDB Performance Overview 面板进行性能分析和优化。

通过本文中介绍的方法，你可以从全局、自顶向下的角度分析用户响应时间和数据库时间，确认用户响应时间的瓶颈是否在数据库中。如果瓶颈在数据库中，你可以通过数据库时间概览和 SQL 延迟的分解，定位数据库内部的瓶颈点，并进行针对性的优化。

11.1.2.1 基于数据库时间的性能优化方法

TiDB 对 SQL 的处理路径和数据库时间进行了完善的测量和记录，方便定位数据库的性能瓶颈。即使在用户响应时间的性能数据缺失的情况下，基于 TiDB 数据库时间的相关性能指标，你也可以达到以下两个性能分析目标：

1. 通过对比 SQL 处理平均延迟和事务中 TiDB 连接的空闲时间，确定整个系统的瓶颈是否在 TiDB 中。
2. 如果瓶颈在 TiDB 内部，根据数据库时间概览、颜色优化法、关键指标和资源利用率、自上而下的延迟分解，定位到性能瓶颈具体在整个分布式系统的哪个模块。

11.1.2.1.1 确定整个系统的瓶颈是否在 TiDB 中

- 如果事务中 TiDB 连接的平均空闲时间比 SQL 平均处理延迟高，说明应用的事务处理中，主要的延迟不在数据库中，数据库时间占用户响应时间比例小，可以确认瓶颈不在数据库中。

  在这种情况下，需要关注数据库外部的组件，比如应用服务器硬件资源是否存在瓶颈，应用到数据库的网络延迟是否过高等。

- 如果 SQL 平均处理延迟比事务中 TiDB 连接的平均空闲时间高，说明事务中主要的瓶颈在 TiDB 内部，数据库时间占用户响应时间比例大。

11.1.2.1.2 如果瓶颈在 TiDB 内部，如何定位

一个典型的 SQL 的处理流程如下所示，TiDB 的性能指标覆盖了绝大部分的处理路径，对数据库时间进行不同维度的分解和上色，用户可以快速的了解负载特性和数据库内部的瓶颈。

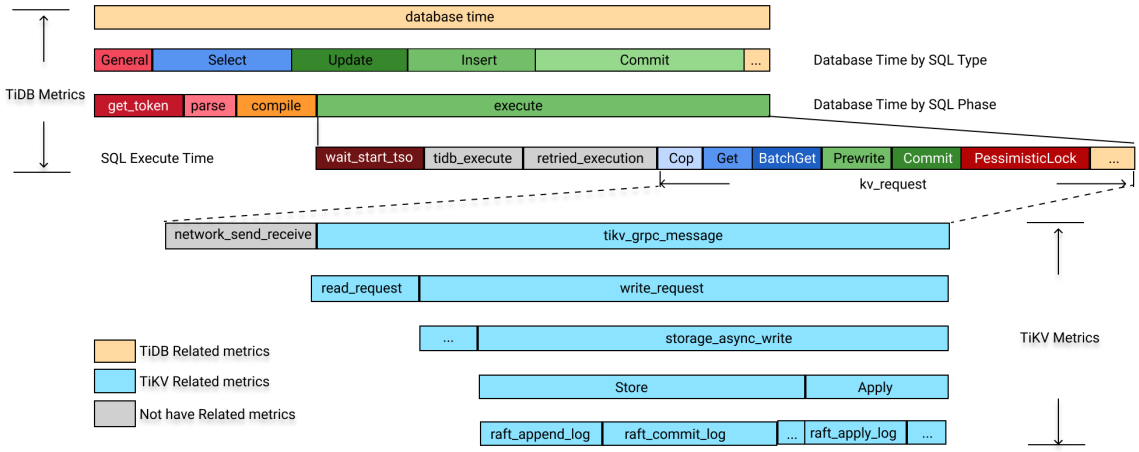


图 90: 数据库时间分解图

数据库时间是所有 SQL 处理时间的总和。通过以下三个维度对数据库时间进行分解，可以帮助你快速定位 TiDB 内部瓶颈：

- 按 SQL 处理类型分解，判断哪种类型的 SQL 语句消耗数据库时间最多。对应的分解公式为：

  ```
  DB Time = Select Time + Insert Time + Update Time + Delete Time + Commit Time + ...
  ```

- 按 SQL 处理的 4 个步骤（即 get_token/parse/compile/execute）分解，判断哪个步骤消耗的时间最多。对应的分解公式为：

  ```
  DB Time = Get Token Time + Parse Time + Comiple Time + Execute Time
  ```

- 对于 execute 耗时，按照 TiDB 执行器本身的时间、TSO 等待时间、KV 请求时间和重试的执行时间，判断执行阶段的瓶颈。对应的分解公式为：

  ```
  Execute Time ~= TiDB Executor Time + KV Request Time + PD TSO Wait Time + Retried execution
  ↪ time
  ```

11.1.2.2 利用 Performance Overview 面板进行性能分析和优化

本章介绍如何利用 Grafana 中的 Performance Overview 面板进行基于数据库时间的性能分析和优化。

Performance Overview 面板按总分结构对 TiDB、TiKV、PD 的性能指标进行编排组织，包括以下三个部分：

- 数据库时间和 SQL 执行时间概览：通过颜色标记不同 SQL 类型，SQL 不同执行阶段、不同请求的数据库时间，帮助你快速识别数据库负载特征和性能瓶颈。
- 关键指标和资源利用率：包含数据库 QPS、应用和数据库的连接信息和请求命令类型、数据库内部 TSO 和 KV 请求 OPS、TiDB 和 TiKV 的资源使用概况。
- 自上而下的延迟分解：包括 Query 延迟和连接空闲时间对比、Query 延迟分解、execute 阶段 TSO 请求和 KV 请求的延迟、TiKV 内部写延迟的分解等。

11.1.2.2.1 数据库时间和 SQL 执行时间概览

Database Time 指标为 TiDB 每秒处理 SQL 的延迟总和，即 TiDB 集群每秒并发处理应用 SQL 请求的总时间 (等于活跃连接数)。

Performance Overview 面板提供了以下三个面积堆叠图，帮助你了解数据库负载的类型，快速定位数据库时间的瓶颈主要是处理什么语句，集中在哪个执行阶段，SQL 执行阶段主要等待 TiKV 或者 PD 哪种请求类型。

- Database Time By SQL Type
- Database Time By SQL Phase
- SQL Execute Time Overview

颜色优化法

通过观察数据库时间分解图和执行时间概览图，你可以直观地区分正常或者异常的时间消耗，快速定位集群的异常瓶颈点，高效了解集群的负载特征。对于正常的时间消耗和请求类型，图中显示颜色为绿色系或蓝色系。如果非绿色或蓝色系的颜色在这两张图中占据了明显的比例，意味着数据库时间的分布不合理。

- Database Time By SQL Type：蓝色标识代表 Select 语句，绿色标识代表 Update、Insert、Commit 等 DML 语句。红色标识代表 General 类型，包含 StmtPrepare、StmtReset、StmtFetch、StmtClose 等命令。
- Database Time By SQL Phase：execute 执行阶段为绿色，其他三个阶段偏红色系，如果非绿色的颜色占比明显，意味着在执行阶段之外数据库消耗了过多时间，需要进一步分析根源。一个常见的场景是因为无法使用执行计划缓存，导致 compile 阶段的橙色占比明显。
- SQL Execute Time Overview：绿色系标识代表常规的写 KV 请求（例如 Prewrite 和 Commit），蓝色系标识代表常规的读 KV 请求（例如 Cop 和 Get），紫色系标识代表 TiFlash MPP 请求，其他色系标识需要注意的问题。例如，悲观锁加锁请求为红色，TSO 等待为深褐色。如果非蓝色系或者非绿色系占比明显，意味着执行阶段存在异常的瓶颈。例如，当发生严重锁冲突时，红色的悲观锁时间会占比明显；当负载中 TSO 等待的消耗时间过长时，深褐色会占比明显。

示例 1：TPC-C 负载

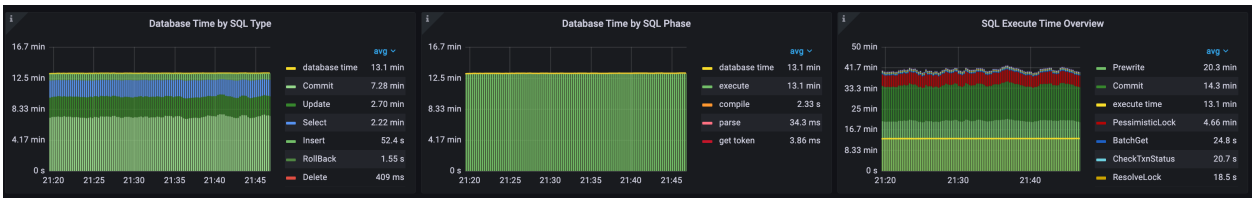


图 91: TPC-C

- Database Time by SQL Type：主要消耗时间的语句为 commit、update、select 和 insert 语句。
- Database Time by SQL Phase：主要消耗时间的阶段为绿色的 execute 阶段。
- SQL Execute Time Overview：执行阶段主要消耗时间的 KV 请求为绿色的 Prewrite 和 Commit。

> 注意：

- KV 请求的总时间大于 execute time 为正常现象，因为 TiDB 执行器可能并发向多个 TiKV 发送 KV 请求，导致总的 KV 请求等待时间大于 execute time。TPC-C 负载中，事务提交时，TiDB 会向多个 TiKV 并行发送 Prewrite 和 Commit 请求，所以这个例子中 Prewrite、Commit 和 PessimisticLock 请求的总时间明显大于 execute time。
- execute time 也可能明显大于 KV 请求的总时间加上 tso_wait 的时间，这意味着 SQL 执行阶段主要时间花在 TiDB 执行器内部。两种常见的例子：
  - 例 1：TiDB 执行器从 TiKV 读取大量数据之后，需要在 TiDB 内部进行复杂的关联和聚合，消耗大量时间。
  - 例 2：应用的写语句锁冲突严重，频繁锁重试导致 `Retried execution time` 过长。

示例 2：OLTP 读密集负载

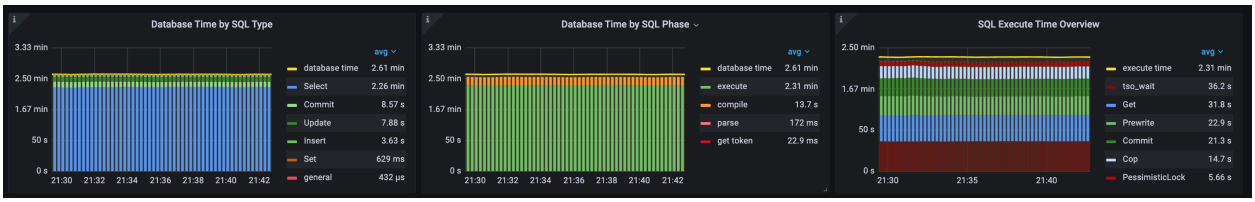


图 92: OLTP

- Database Time by SQL Type：主要消耗时间的语句为 select、commit、update 和 insert 语句。其中，select 占据绝大部分的数据库时间。
- Database Time by SQL Phase：主要消耗时间的阶段为绿色的 execute 阶段。
- SQL Execute Time Overview：执行阶段主要消耗时间为深褐色的 pd tso_wait、蓝色的 KV Get 和绿色的 Prewrite 和 Commit。

示例 3：只读 OLTP 负载

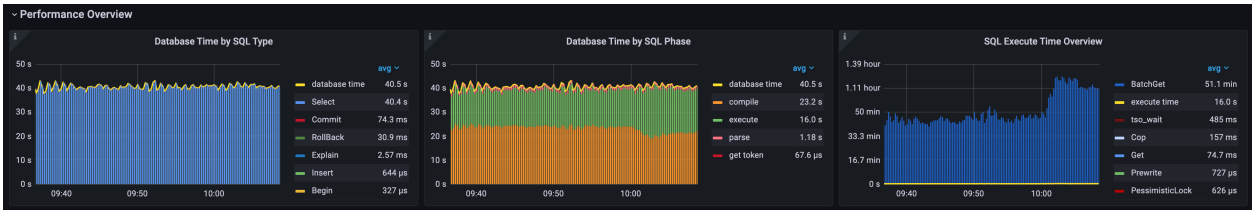


图 93: OLTP

- Database Time by SQL Type：几乎所有语句为 select。
- Database Time by SQL Phase：主要消耗时间的阶段为橙色的 compile 和绿色的 execute 阶段。compile 阶段延迟最高，代表着 TiDB 生成执行计划的过程耗时过长，需要根据后续的性能数据进一步确定根源。
- SQL Execute Time Overview：执行阶段主要消耗时间的 KV 请求为蓝色 BatchGet。

> **注意：**
>
> 示例 3 select 语句需要从多个 TiKV 并行读取几千行数据，BatchGet 请求的总时间远大于执行时间。

示例 4：锁争用负载

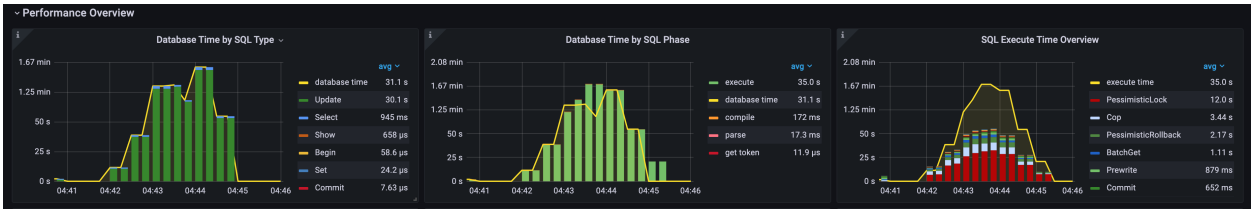


图 94: OLTP

- Database Time by SQL Type：主要为 Update 语句。
- Database Time by SQL Phase：主要消耗时间的阶段为绿色的 execute 阶段。
- SQL Execute Time Overview：执行阶段主要消耗时间的 KV 请求为红色的悲观锁 PessimisticLock，execute time 明显大于 KV 请求的总时间，这是因为应用的写语句锁冲突严重，频繁锁重试导致 `Retried execution time` 过长。目前 `Retried execution time` 消耗的时间，TiDB 尚未进行测量。

示例 5：HTAP CH-Benchmark 负载

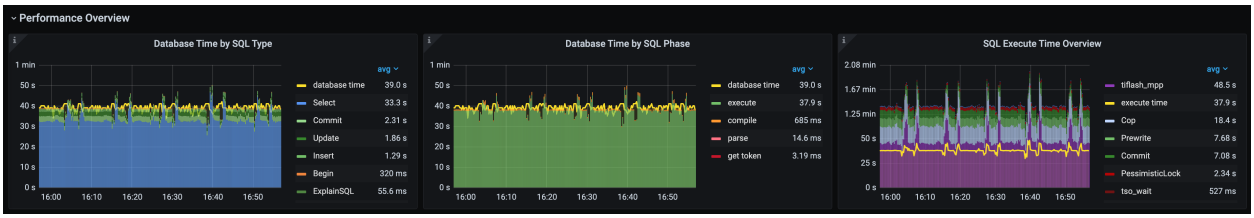


图 95: HTAP

- Database Time by SQL Type：主要为 Select 语句。
- Database Time by SQL Phase：主要消耗时间的阶段为绿色的 execute 阶段。
- SQL Execute Time Overview：执行阶段主要消耗时间为紫色的 `tiflash_mpp` 请求，其次是 KV 请求，包括蓝色的 Cop，以及绿色的 `Prewrite` 和 `Commit`。

11.1.2.2.2 TiDB 关键指标和集群资源利用率

Query Per Second、 Command Per Second 和 Prepared-Plan-Cache

通过观察 Performance Overview 里的以下三个面板，可以了解应用的负载类型，与 TiDB 的交互方式，以及是否能有效地利用 TiDB 的执行计划缓存。

- QPS：表示 Query Per Second，包含应用的 SQL 语句类型执行次数分布。
- CPS By Type：CPS 表示 Command Per Second，Command 代表 MySQL 协议的命令类型。同样一个查询语句可以通过 query 或者 prepared statement 的命令类型发送到 TiDB。
- Queries Using Plan Cache OPS：TiDB 集群每秒执行计划缓存的命中次数（即 `avg-hit`）和未命中次数（即 `avg-miss`）。

  StmtExecute 每秒执行次数等于 `avg-hit + avg-miss`。执行计划缓存只支持 prepared statement 命令。当 TiDB 开启执行计划缓存时，存在三种使用情况：

  - 完全无法命中执行计划缓存：每秒命中次数 `avg-hit` 为 0，`avg-miss` 等于 StmtExecute 命令每秒执行次数。可能的原因包括：
    * 应用使用了 query 命令。
    * 每次 StmtExecute 执行之后，应用调用了 StmtClose 命令，导致缓存的执行计划被清理。
    * StmtExecute 执行的所有语句都不符合缓存的条件，导致无法命中执行计划缓存。
  - 完全命中执行计划缓存：每秒命中次数 `avg-hit` 等于 StmtExecute 命令每秒执行次数，每秒未命中次数 `avg-miss` 等于 0。
  - 部分命中执行计划缓存：每秒命中次数 `avg-hit` 小于 StmtExecute 命令每秒执行次数。执行计划缓存目前存在一些限制，比如不支持子查询，该类型的 SQL 执行计划无法被缓存。

示例 1：TPC-C 负载

TPC-C 负载类型主要以 Update、Select 和 Insert 语句为主。总的 QPS 等于每秒 StmtExecute 的次数，并且 StmtExecute 每秒的数据基本等于 Queries Using Plan Cache OPS 面板的 `avg-hits`。这是 OLTP 负载理想的情况，客户端执行使用 prepared statement，并且在客户端缓存了 prepared statement 对象，执行每条 SQL 语句时直接调用 statement 执行。执行时都命中执行计划缓存，不需要重新 compile 生成执行计划。

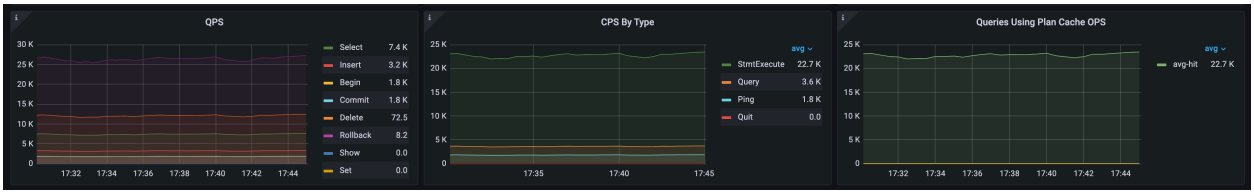


图 96: TPC-C

示例 2：只读 OLTP 负载，使用 query 命令无法使用执行计划缓存

这个负载中，Commit QPS = Rollback QPS = Select QPS。应用开启了 auto-commit 并发，每次从连接池获取连接都会执行 rollback，因此这三种语句的执行次数是相同的。

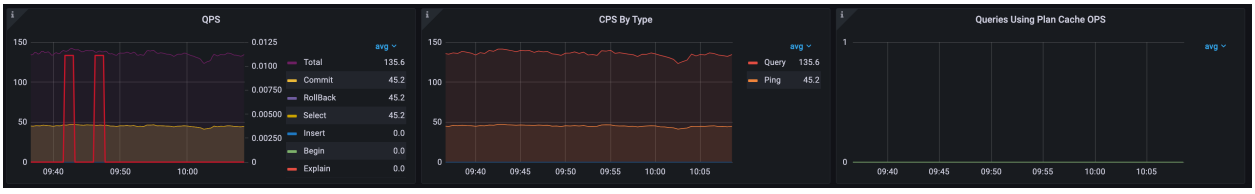


图 97: OLTP-Query

- QPS 面板中出现的红色加粗线为 Failed Query，坐标的值为右边的 Y 轴。非 0 代表此负载中存在错误语句。
- 总的 QPS 等于 CPS By Type 面板中的 Query，说明应用中使用了 query 命令。
- Queries Using Plan Cache OPS 面板没有数据，因为不使用 prepared statement 接口，无法使用 TiDB 的执行计划缓存，意味着应用的每一条 query，TiDB 都需要重新解析，重新生成执行计划。通常会导致 compile 时间变长以及 TiDB CPU 消耗的增加。

示例 3：OLTP 负载，使用 prepared statement 接口无法使用执行计划缓存

StmtPreare 次数 = StmtExecute 次数 = StmtClose 次数 ~= StmtFetch 次数，应用使用了 prepare > execute > fetch > close 的 loop，很多框架都会在 execute 之后调用 close，确保资源不会泄露。这会带来两个问题：

- 执行每条 SQL 语句需要 4 个命令，以及 4 次网络往返。
- Queries Using Plan Cache OPS 为 0，无法命中执行计划缓存。StmtClose 命令默认会清理缓存的执行计划，导致下一次 StmtPreare 命令需要重新生成执行计划。

> 注意：
>
> 从 TiDB v6.0.0 起，你可以通过全局变量 (`set global tidb_ignore_prepared_cache_close_stmt =on;`) 控制 StmtClose 命令不清理已被缓存的执行计划，使得下一次的 SQL 的执行不需要重新生成执行计划。

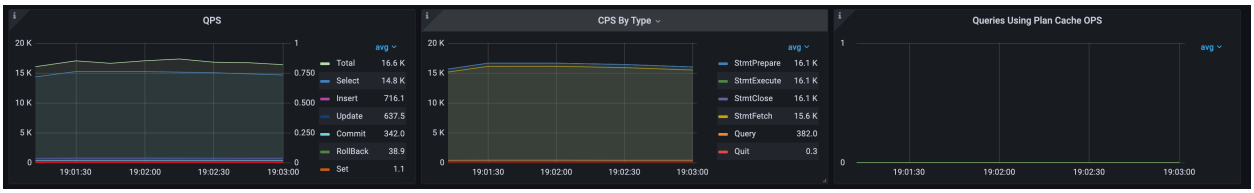


图 98: OLTP-Prepared

示例 4：Prepared Statement 存在资源泄漏

StmtPrepare 每秒执行次数远大于 StmtClose，说明应用程序存在 prepared statement 对象泄漏。

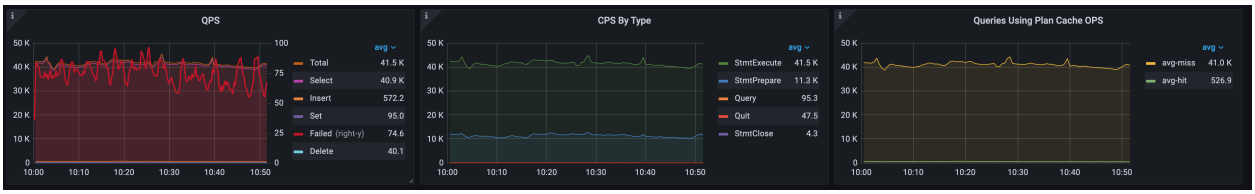


图 99: OLTP-Query

- QPS 面板中出现的红色加粗线为 Failed Query，坐标的值为右边的 Y 轴。每秒错误语句为 74.6 条。

- CPS By Type 面板中的 StmtPrepare 每秒执行次数远大于 StmtClose，说明应用程序存在 prepared statement 对象泄漏。
- Queries Using Plan Cache OPS 面板中的 `avg-miss` 几乎等于 CPS By Type 面板中的 StmtExecute，说明几乎所有的 SQL 执行都未命中执行计划缓存。

KV/TSO Request OPS 和 KV Request Time By Source

- 在 KV/TSO Request OPS 面板中，你可以查看 KV 和 TSO 每秒请求的数据统计。其中，`kv request total` 代表 TiDB 到 TiKV 所有请求的总和。通过观察 TiDB 到 PD 和 TiKV 的请求类型，可以了解集群内部的负载特征。
- 在 KV Request Time By Source 面板中，你可以查看每种 KV 请求和请求来源的时间占比。
  - kv request total time 是每秒总的 KV 和 TiFlash 请求处理时间
  - 每种 KV 请求和请求来源组成柱状堆叠图，`external` 标识正常业务的请求，`internal` 标识内部活动的请求（比如 DDL、auto analyze 等请求）。

示例 1：繁忙的负载



图 100: TPC-C

在此 TPC-C 负载中：

- 每秒总的 KV 请求的数量为 79.7 K。按请求数量排序，最高的请求类型为 `Prewrite`、`Commit`、`PessimisticsLock` 和 `BatchGet` 等。
- KV 处理时间来源主要为 `Commit-external_Commit`、`Prewrite-external_Commit`，说明消耗时间最高的 KV 请求为 `Commit` 和 `Prewrite`，并且来源于外部的 Commit 语句。

示例 2：Analyze 负载



图 101: OLTP

集群中只有 analyze 语句运行：

- 每秒总的 KV 请求数据是 35.5，Cop 请求次数是每秒 9.3。

- KV 处理时间主要来源为 `Cop-internal_stats`，说明 Cop 请求来源于内部的 analyze 操作。

TiDB CPU，以及 TiKV CPU 和 IO 使用情况

在 TiDB CPU 和 TiKV CPU/IO MBps 这两个面板中，你可以观察到 TiDB 和 TiKV 的逻辑 CPU 使用率和 IO 吞吐，包含平均、最大和 delta（最大 CPU 使用率减去最小 CPU 使用率），从而用来判定 TiDB 和 TiKV 总体的 CPU 使用率。

- 通过 `delta` 值，你可以判断 TiDB 是否存在 CPU 使用负载不均衡（通常伴随着应用连接不均衡），TiKV 是否存在热点。
- 通过 TiDB 和 TiKV 的资源使用概览，你可以快速判断集群是否存在资源瓶颈，最需要扩容的组件是 TiDB 还是 TiKV。

示例 1：TiDB 资源使用率高

下图负载中，每个 TiDB 和 TiKV 配置 8 CPU。



图 102: TPC-C

- TiDB 平均 CPU 为 575%。最大 CPU 为 643%，delta CPU 为 136%。
- TiKV 平均 CPU 为 146%，最大 CPU 215%。delta CPU 为 118%。TiKV 的平均 IO 吞吐为 9.06 MB/s，最大 IO 吞吐为 19.7 MB/s，delta IO 吞吐为 17.1 MB/s。

由此可以判断，TiDB 的 CPU 消耗明显更高，并接近于 8 CPU 的瓶颈，可以考虑扩容 TiDB。

示例 2：TiKV 资源使用率高

下图 TPC-C 负载中，每个 TiDB 和 TiKV 配置 16 CPU。

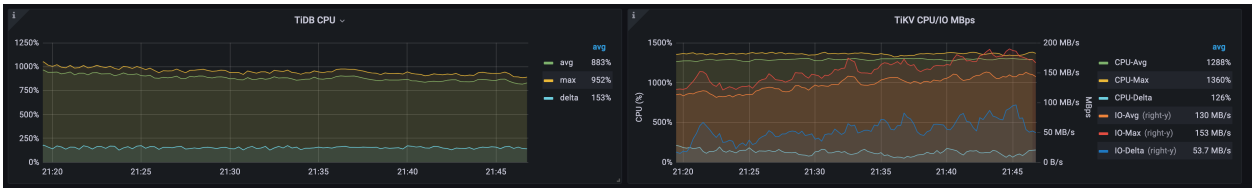


图 103: TPC-C

- TiDB 平均 CPU 为 883%。最大 CPU 为 962%，delta CPU 为 153%。
- TiKV 平均 CPU 为 1288%，最大 CPU 1360%。delta CPU 为 126%。TiKV 的平均 IO 吞吐为 130 MB/s，最大 IO 吞吐为 153 MB/s，delta IO 吞吐为 53.7 MB/s。

由此可以判断，TiKV 的 CPU 消耗更高，因为 TPC-C 是一个写密集场景，这是正常现象，可以考虑扩容 TiKV 节点提升性能。

11.1.2.2.3 Query 延迟分解和关键的延迟指标

延迟面板通常包含平均值和 99 分位数，平均值用来定位总体的瓶颈，99 分位数用来判断是否存在延迟严重抖动的情况。判断性能抖动范围时，可能还需要需要借助 999 分位数。

Duration、 Connection Idle Duration 和 Connection Count

Duration 面板包含了所有语句的 99 延迟和每种 SQL 类型的平均延迟。Connection Idle Duration 面板包含连接空闲的平均和 99 延迟，连接空闲时包含两种状态：

- in-txn：代表事务中连接的空闲时间，即当连接处于事务中时，处理完上一条 SQL 之后，收到下一条 SQL 语句的间隔时间。
- not-in-txn：当连接没有处于事务中，处理完上一条 SQL 之后，收到下一条 SQL 语句的间隔时间。

应用进行数据库事务时，通常使用同一个数据库连接。对比 query 的平均延迟和 connection idle duration 的延迟，可以判断整个系统性能瓶颈或者用户响应时间的抖动是否是由 TiDB 导致的。

- 如果应用负载不是只读的，包含事务，对比 query 平均延迟和 `avg-in-txn` 可以判断应用处理事务时，主要的时间是花在数据库内部还是在数据库外面，借此定位用户响应时间的瓶颈。
- 如果是只读负载，不存在 `avg-in-txn` 指标，可以对比 query 平均延迟和 `avg-not-in-txn` 指标。

现实的客户负载中，瓶颈在数据库外面的情况并不少见，例如：

- 客户端服务器配置过低，CPU 资源不够。
- 使用 HAProxy 作为 TiDB 集群代理，但是 HAProxy CPU 资源不够。
- 使用 HAProxy 作为 TiDB 集群代理，但是高负载下 HAProxy 服务器的网络带宽被打满。
- 应用服务器到数据库延迟过高，比如公有云环境应用和 TiDB 集群不在同一个地区，比如数据库的 DNS 均衡器和 TiDB 集群不在同一个地区。
- 客户端程序存在瓶颈，无法充分利用服务器的多 CPU 核或者多 Numa 资源，比如应用只使用一个 JVM 向 TiDB 建立上千个 JDBC 连接。

在 Connection Count （连接信息）面板中，你可以查看总的连接数和每个 TiDB 节点的连接数，并由此判断连接总数是否正常，各 TiDB 节点的连接数是否不均衡。`active connections` 记录着活跃连接数，等于每秒的数据库时间，右侧 Y 轴为 `disconnection/s`，代表集群每秒断开连接的数量，用来判断应用是否使用了短连接。

示例 1：disconnection/s 过高

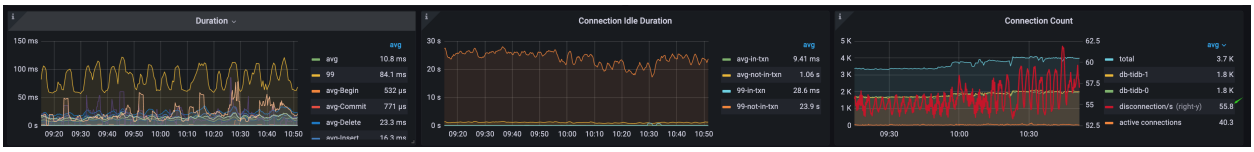


图 104: high disconnection/s

在此负载中：

- 所有 SQL 语句的平均延迟 10.8 ms，P99 延迟 84.1 ms。

- 事务中连接空闲时间 avg-in-txn 为 9.4 ms。
- 集群总的连接数为 3.7K，每个 TiDB 节点的连接数为 1.8 K。平均活跃连接数为 40.3，大部分连接处于空闲状态。disonnnection/s 平均为 55.8，说明应用在频繁的新建和断开连接。短连接的行为会对 TiDB 的资源和响应时间造成一定的影响。

示例 2：用户响应时间的瓶颈在 TiDB 中

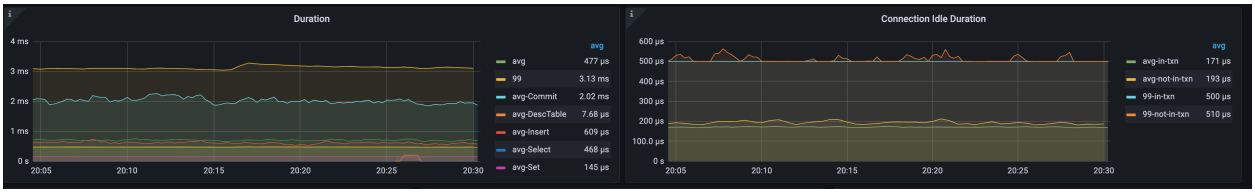


图 105: TiDB is the Bottleneck

在此 TPC-C 负载中：

- 所有 SQL 语句的平均延迟 477 us，99 延迟 3.13 ms。平均 commit 语句 2.02 ms，平均 insert 语句 609 us，平均查询语句 468 us。
- 事务中连接空闲时间 avg-in-txn 171 us。

由此可以判断，平均的 query 延迟明显大于 avg-in-txn，说明事务处理中，主要的瓶颈在数据库内部。

示例 3：用户响应时间的瓶颈不在 TiDB 中

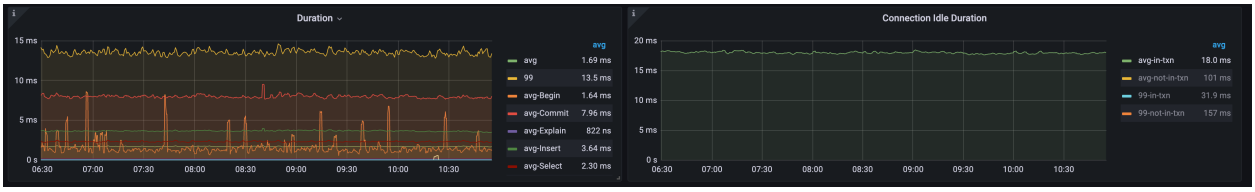


图 106: TiDB is not the Bottleneck

在此负载中，平均 query 延迟为 1.69 ms，事务中连接空闲时间 avg-in-txn 为 18 ms。说明事务中，TiDB 平均花了 1.69 ms 处理完一个 SQL 语句之后，需要等待 18 ms 才能收到下一条语句。

由此可以判断，用户响应时间的瓶颈不在 TiDB 中。这个例子是在一个公有云环境下，因为应用和数据库不在同一个地区，应用和数据库之间的网络延迟高导致了超高的连接空闲时间。

Parse、 Compile 和 Execute Duration

在 TiDB 中，从输入查询文本到返回结果的典型处理流程。

SQL 在 TiDB 内部的处理分为四个阶段，get token、 parse、 compile 和 execute：

- get token 阶段：通常只有几微秒的时间，可以忽略。除非 TiDB 单个实例的连接数达到的token-limit 的限制，创建连接的时候被限流。

- parse 阶段：query 语句解析为抽象语法树 abstract syntax tree (AST)。
- compile 阶段：根据 parse 阶段输出的 AST 和统计信息，编译出执行计划。整个过程主要步骤为逻辑优化与物理优化，前者通过一些规则对查询计划进行优化，例如基于关系代数的列裁剪等，后者通过统计信息和基于成本的优化器，对执行计划的成本进行估算，并选择整体成本最小的物理执行计划。
- execute 阶段：时间消耗视情况，先等待全局唯一的时间戳 TSO，之后执行器根据执行计划中算子涉及的 Key 范围，构建出 TiKV 的 API 请求，分发到 TiKV。execute 时间包含 TSO 等待时间、KV 请求的时间和 TiDB 执行器本身处理数据的时间。

如果应用统一使用 query 或者 StmtExecute MySQL 命令接口，可以使用以下公式来定位平均延迟的瓶颈。

```
avg Query Duration = avg Get Token + avg Parse Duration + avg Compile Duration + avg Execute
    ↪ Duration
```

通常 execute 阶段会占 query 延迟的主要部分，在以下情况下，parse 和 compile 阶段也会占比明显。

- parse 阶段延迟占比明显：比如 query 语句很长，文本解析消耗大量的 CPU。
- compile 阶段延迟占比明显：如果应用没有使用执行计划缓存，每个语句都需要生成执行计划。compile 阶段的延迟可能达到几毫秒或者几十毫秒。如果无法命中执行计划缓存，compile 阶段需要进行逻辑优化和物理优化，这将消耗大量的 CPU 和内存，并给 Go Runtime 带来压力（因为 TiDB 是 Go 编写的），进一步影响 TiDB 其他组件的性能。这说明，OLTP 负载在 TiDB 中是否能高效运行，执行计划缓存扮演了重要的角色。

示例 1：数据库瓶颈在 compile 阶段

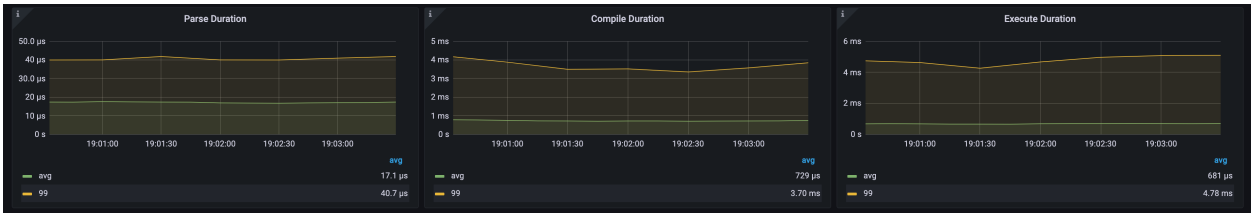


图 107: Compile

此图中 parse、compile 和 execute 阶段的平均时间分别为 17.1 us、729 us 和 681 us。因为应用使用 query 命令接口，无法使用执行计划缓存，所以 compile 阶段延迟高。

示例 2：数据库瓶颈在 execute 阶段

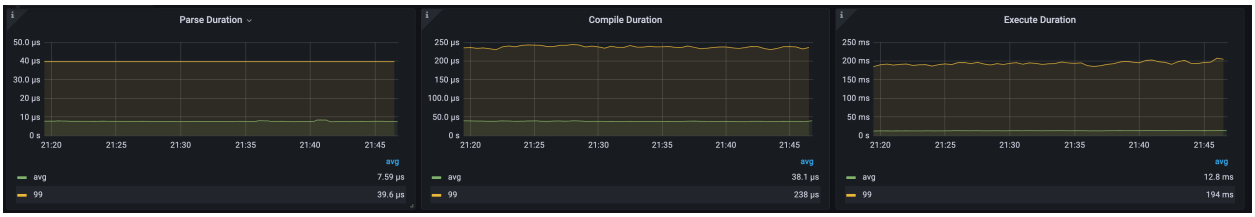


图 108: Execute

在此 TPC-C 负载中，parse、compile 和 execute 阶段的平均时间分别为 7.39us、38.1us 和 12.8ms。query 延迟的瓶颈在于 execute 阶段。

KV 和 TSO Request Duration

在 execute 阶段，TiDB 会跟 PD 和 TiKV 进行交互。如下图所示，当 TiDB 处理 SQL 语句请求时，在进行 parse 和 compile 之前，如果需要获取 TSO，会先请求生成 TSO。PD Client 不会阻塞调用者，而是直接返回一个 TSFuture，并在后台异步处理 TSO 请求的收发，一旦完成立即返回给 TSFuture，TSFuture 的持有者则需要调用 Wait 方法来获得最终的 TSO 结果。当 TiDB 完成 parse 和 compile 之后，进入 execute 阶段，此时存在两个情况：

- 如果 TSO 请求已经完成，Wait 方法会立刻返回一个可用的 TSO 或 error
- 如果 TSO 请求还未完成，Wait 方法会 block 住等待一个可用的 TSO 或 error（说明 gRPC 请求已发送但尚未收到返回结果，网络延迟较高）

TSO 等待的时间记录为 TSO WAIT，TSO 请求的网络时间记录为 TSO RPC。TiDB TSO 等待完成之后，执行过程中通常需要和 TiKV 进行读写交互：

- 读的 KV 请求常见类型：Get、BatchGet 和 Cop
- 写的 KV 请求常见类型：PessimisticLock，二阶段提交的 Prewrite 和 Commit

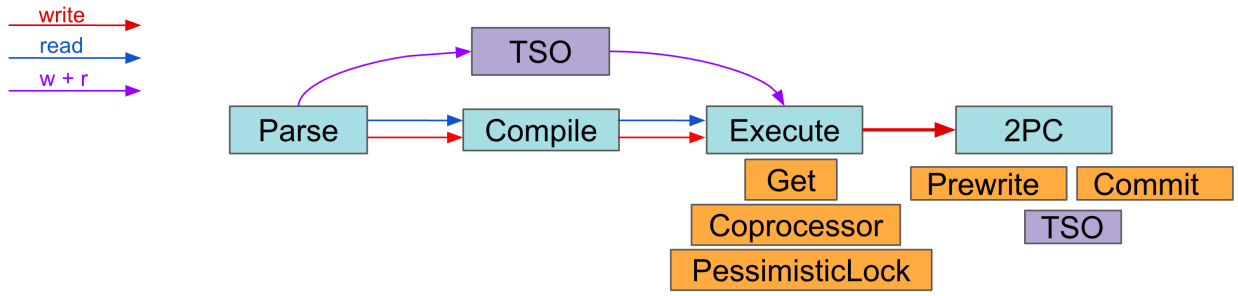


图 109: Execute

这一部分的指标对应以下三个面板：

- Avg TiDB KV Request Duration：TiDB 测量的 KV 请求的平均延迟
- Avg TiKV GRPC Duration：TiKV 内部 GRPC 消息处理的平均延迟
- PD TSO Wait/RPC Duration：TiDB 执行器等待 TSO 延迟 (wait) 和 TSO 请求的网络延迟 (rpc)。

其中，Avg TiDB KV Request Duration 和 Avg TiKV GRPC Duration 的关系如下

```
Avg TiDB KV Request Duration = Avg TiKV GRPC Duration + TiDB 与 TiKV 之间的网络延迟 + TiKV GRPC
    ↪ 处理时间 + TiDB GRPC 处理时间和调度延迟。
```

Avg TiDB KV Request Duration 和 Avg TiKV GRPC Duration 的差值跟网络流量和延迟，TiDB 和 TiKV 的资源使用情况密切相关。

- 同一个机房内，Avg TiDB KV Request Duration 和 Avg TiKV GRPC Duration 的差值通常应该小于 2 毫秒。

- 同一地区的不同可用区，Avg TiDB KV Request Duration 和 Avg TiKV GRPC Duration 的差值通常应该小于 5 毫秒。

示例 1：同机器低负载的集群

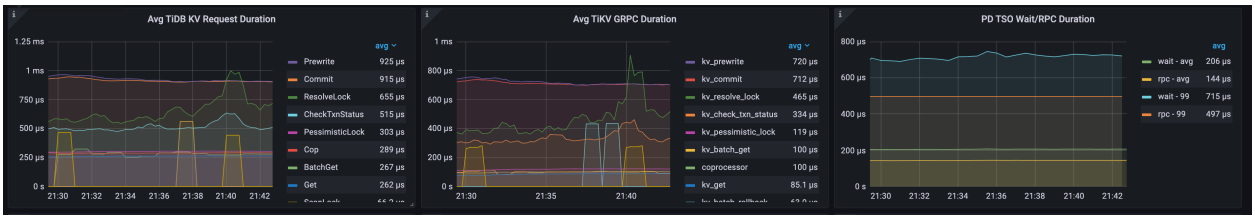


图 110: Same Data Center

在此负载中，TiDB 侧平均 Prewrite 请求延迟为 925 us，TiKV 内部 kv_prewrite 平均处理延迟为 720 us，相差 200 us 左右，是同机房内正常的延迟。TSO wait 平均延迟 206 us，rpc 时间为 144 us。

示例 2：公有云集群，负载正常

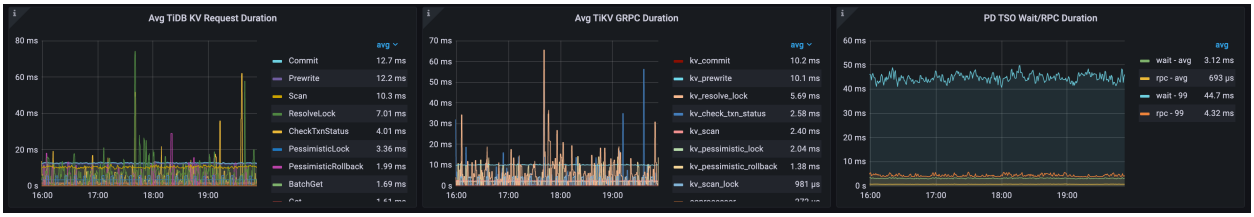


图 111: Cloud Env

在此示例中，TiDB 集群部署在同一个地区的不同机房。TiDB 侧平均 Commit 请求延迟为 12.7 ms，TiKV 内部 kv_commit 平均处理延迟为 10.2 ms，相差 2.5 ms 左右。TSO wait 平均延迟为 3.12 ms，rpc 时间为 693 us。

示例 3：公有云集群，资源严重过载

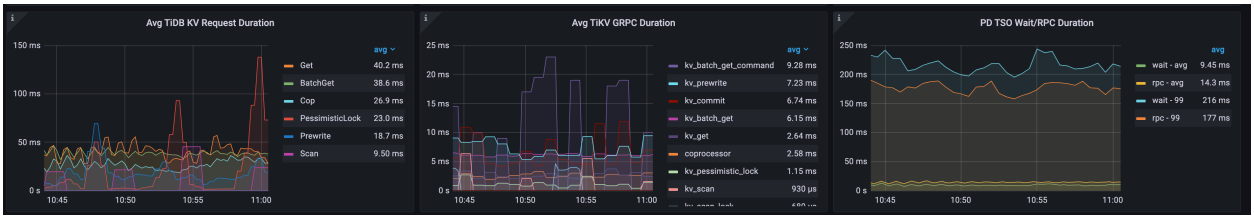


图 112: Cloud Env, TiDB Overloaded

在此示例中，TiDB 集群部署在同一个地区的不同机房，TiDB 网络和 CPU 资源严重过载。TiDB 侧平均 BatchGet 请求延迟为 38.6 ms，TiKV 内部 kv_batch_get 平均处理延迟为 6.15 ms，相差超过 32 ms，远高于正常值。TSO wait 平均延迟为 9.45 ms，rpc 时间为 14.3 ms。

Storage Async Write Duration、 Store Duration 和 Apply Duration

TiKV 对于写请求的处理流程如下图

- `scheduler worker` 会先处理写请求，进行事务一致性检查，并把写请求转化成键值对，发送到 `raftstore` 模块。
- `raftstore` 为 TiKV 的共识模块，使用 Raft 共识算法，使多个 TiKV 组成的存储层可以容错。

  Raftstore 分为 store 线程和 apply 线程。：

  - store 线程负载处理 Raft 消息和新的 `proposals`。当收到新的 `proposals` 时，leader 节点的 store 线程会写入本地 Raft DB，并将消息复制到多个 follower 节点。当这个 `proposals` 在多数实例持久化成功之后，`proposals` 成功被提交。
  - apply 线程会负载将提交的内容写入到 KV DB 中。当写操作的内容被成功的写入 KV 数据库中，apply 线程会通知外层请求写请求已经完成。

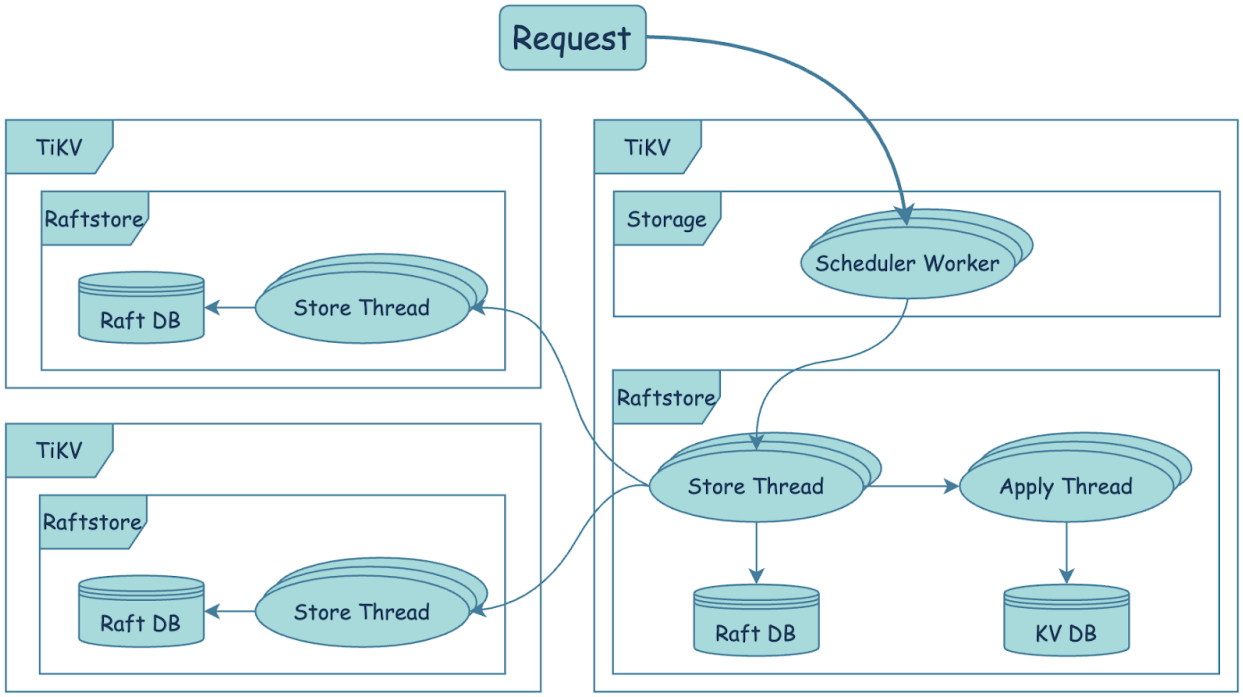


图 113: TiKV Write

Storage Async Write Duration 指标记录写请求进入 raftstore 之后的延迟，采集的粒度具体到每个请求的级别。

Storage Async Write Duration 分为 Store Duration 和 Apply Duration。你可以通过以下公式定位写请求的瓶颈主要是在 Store 还是 Apply 步骤。

```
avg Storage Async Write Duration  = avg Store Duration + avg Apply Duration
```

> 注意：
>
> Store Duration 和 Apply Duration 从 v5.3.0 版本开始支持。

示例 1：同一个 OLTP 负载在 v5.3.0 和 v5.4.0 版本的对比

v5.4.0 版本，一个写密集的 OLTP 负载 QPS 比 v5.3.0 提升了 14%。应用以上公式

- v5.3.0：24.4 ms ~= 17.7 ms + 6.59 ms
- v5.4.0：21.4 ms ~= 14.0 ms + 7.33 ms

因为 v5.4.0 版本中，TiKV 对 gRPC 模块进行了优化，优化了 Raft 日志复制速度，相比 v5.3.0 降低了 Store Duration。

v5.3.0：

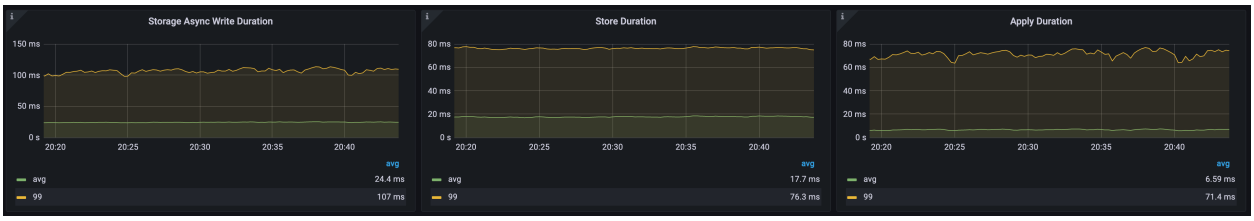


图 114: v5.3.0

v5.4.0：



图 115: v5.4.0

示例 2：Store Duration 瓶颈明显

应用以上公式：10.1 ms ~= 9.81 ms + 0.304 ms，说明写请求的延迟瓶颈在 Store Duration。

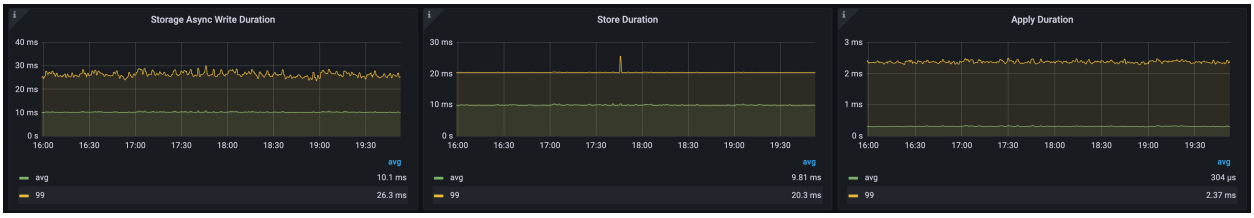


图 116: Store

Commit Log Duration、Append Log Duration 和 Apply Log Duration

Commit Log Duration、Append Log Duration 和 Apply Log Duration 这三个延迟是 raftstore 内部关键操作的延迟记录。这些记录采集的粒度是 batch 操作级别的，每个操作会把多个写请求合并在一起，因此不能直接对应上文的 Store Duration 和 Apply Duration。

- Commit Log Duration 和 Append Log Duration 均为 store 线程的操作。Commit Log Duration 包含复制 Raft 日志到其他 TiKV 节点，保证 raft-log 的持久化。一般包含两次 Append Log Duration，一次 leader，一次 follower 的。Commit Log Duration 延迟通常会明显高于 Append Log Duration，因为包含了通过网络复制 Raft 日志到其他 TiKV 的时间。
- Apply Log Duration 记录了 apply 线程 apply Raft 日志的延迟。

Commit Log Duration 慢的常见场景：

- TiKV CPU 资源存在瓶颈，调度延迟高
- `raftstore.store-pool-size` 设置过小或者过大（过大也可能导致性能下降）
- IO 延迟高，导致 Append Log Duration 延迟高
- TiKV 之间的网络延迟比较高
- TiKV 的 gRPC 线程数设置过小或者多个 gRPC CPU 资源使用不均衡

Apply Log Duration 慢的常见场景：

- TiKV CPU 资源存在瓶颈，调度延迟高
- `raftstore.apply-pool-size` 设置过小或者过大（过大也可能导致性能下降）
- IO 延迟比较高

示例 1：同一个 OLTP 负载在 v5.3.0 和 v5.4.0 版本的对比

v5.4.0 版本，一个写密集的 OLTP 负载 QPS 比 v5.3.0 提升了 14%。对比这三个关键延迟：

Avg Duration	v5.3.0(ms)	v5.4.0(ms)
Append Log Duration	0.27	0.303
Commit Log Duration	13	8.68
Apply Log Duration	0.457	0.514

因为 v5.4.0 版本中，TiKV 对 gRPC 模块进行了优化，优化了 Raft 日志复制速度，相比 v5.3.0 降低了 Commit Log Duration 和 Store Duration。

v5.3.0：

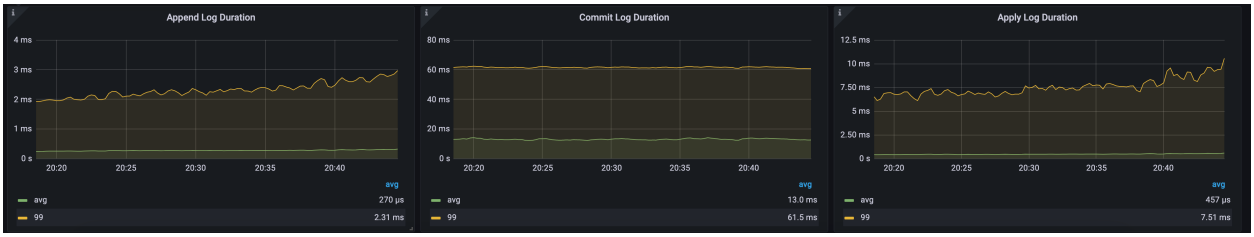


图 117: v5.3.0

v5.4.0：

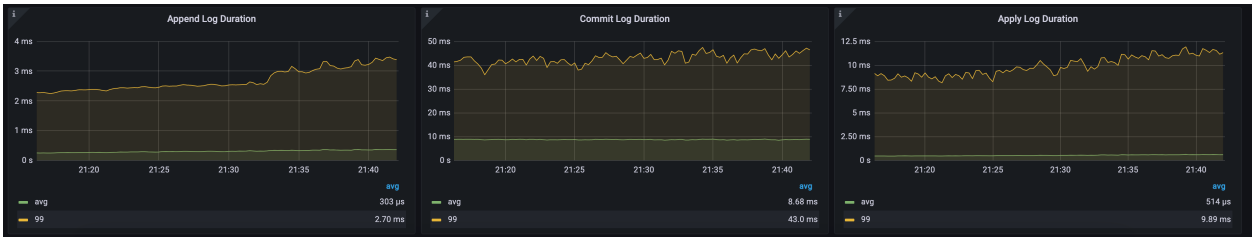


图 118: v5.4.0

示例 2：Commit Log Duration 瓶颈明显的例子

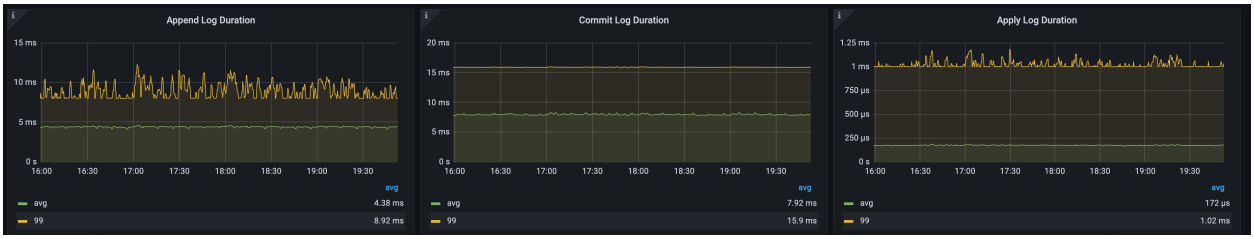


图 119: Store

- 平均 Append Log Duration = 4.38 ms
- 平均 Commit Log Duration = 7.92 ms
- 平均 Apply Log Duration = 172 us

Store 线程的 Commit Log Duration 明显比 Apply Log Duration 高，并且 Append Log Duration 比 Apply Log Duration 明显的高，说明 Store 线程在 CPU 和 IO 都可能都存在瓶颈。可能降低 Commit Log Duration 和 Append Log Duration 的方式如下：

- 如果 TiKV CPU 资源充足，考虑增加 Store 线程，即 `raftstore.store-pool-size`。
- 如果 TiDB 为 v5.4.0 及之后的版本，考虑启用`Raft Engine`，Raft Engine 具有更轻量的执行路径，在一些场景下显著减少 IO 写入量和写入请求的长尾延迟，启用方式为设置：`raft-engine.enable: true`
- 如果 TiKV CPU 资源充足，且 TiDB 为 v5.3.0 及之后的版本，考虑启用`StoreWriter`。启用方式：`raftstore.store-io-pool-size: 1`。

11.1.2.3 低于 v6.1.0 的 TiDB 版本如何使用 Performance overview 面板

从 v6.1.0 起，TiDB Grafana 组件默认内置了 Performance Overview 面板。Performance overview 面板兼容 TiDB v4.x 和 v5.x 版本。如果你的 TiDB 版本低于 v6.1.0，需要手动导入 `performance_overview.json`。

导入方法如图所示：

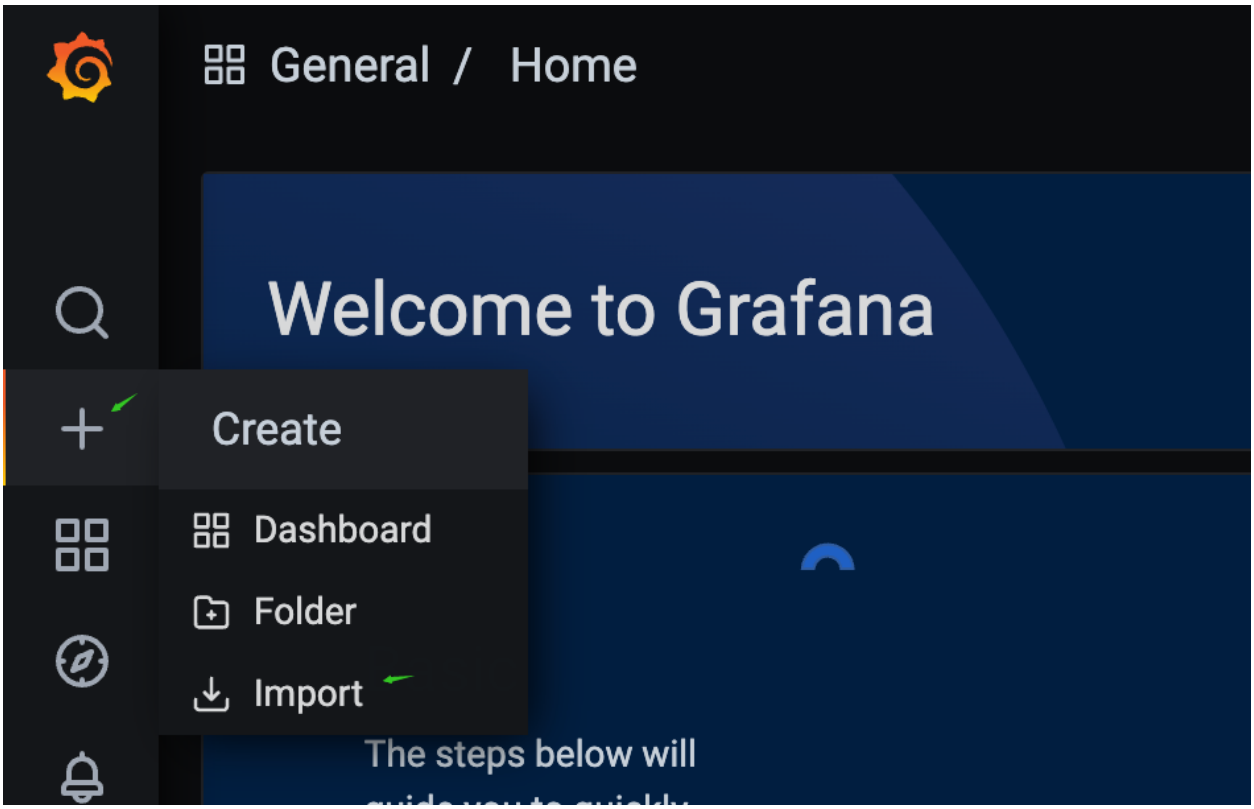


图 120: Store

11.1.3 OLTP 负载性能优化实践

TiDB 提供了完善的性能诊断和分析功能，例如 TiDB Dashboard 的Top SQL 和Continuous Profiling 功能，以及 TiDB Performance Overview 面板。

本文介绍如何综合利用这些功能，对同一个 OLTP 负载在七种不同运行场景下的性能表现进行分析和对比，并演示了具体的 OLTP 负载的优化过程，帮助你更快地对 TiDB 的性能进行分析和优化。

> 注意：
>
> Top SQL 和Continuous Profiling 功能默认关闭，需要提前开启。

在这些场景中，通过使用不同的 JDBC 配置运行同一个应用程序，你可以观察应用和数据库之间不同的交互方式将如何影响系统整体的性能，从而更好地掌握开发 Java 应用使用 TiDB 的最佳实践。

11.1.3.1 负载环境

本文使用一个银行交易系统 OLTP 仿真模拟负载进行演示。以下为该负载的仿真环境配置：

- 负载应用程序的开发语言：JAVA

- 涉及业务的 SQL 语句：共 200 条，其中 90% 都是 SELECT 语句，属于典型的读密集 OLTP 场景。
- 涉及交易的表：共 60 张，存在修改操作类的表为 12 张，其余 48 张表只读。
- 应用程序使用的隔离级别：`read committed`。
- TiDB 集群配置：3 个 TiDB 节点和 3 个 TiKV 节点，各节点分配 16 CPU。
- 客户端服务器配置：36 CPU。

11.1.3.2 场景 1：使用 Query 接口

11.1.3.2.1 应用配置

应用程序使用以下 JDBC 配置，通过 Query 接口连接数据库。

```
useServerPrepStmts=false
```

11.1.3.2.2 性能分析

TiDB Dashboard

从以下 Dashboard 的 Top SQL 页面可以观察到，非业务 SQL 类型 `SELECT @@session.tx_isolation` 消耗的资源最多。虽然 TiDB 处理这类 SQL 语句的速度快，但由于执行次数最多导致总体 CPU 耗时最多。

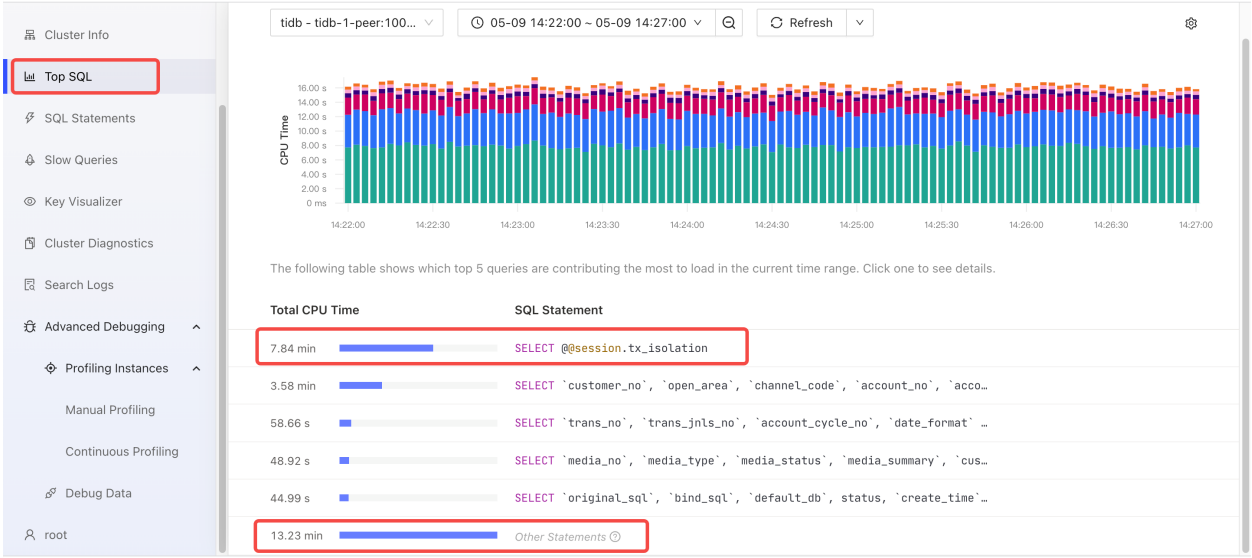


图 121: dashboard-for-query-interface

观察以下 TiDB 的火焰图，可以发现，在 SQL 的执行过程中，Compile 和 Optimize 等函数的 CPU 消耗占比明显。因为应用使用了 Query 接口，TiDB 无法使用执行计划缓存，导致每个 SQL 都需要编译生成执行计划。

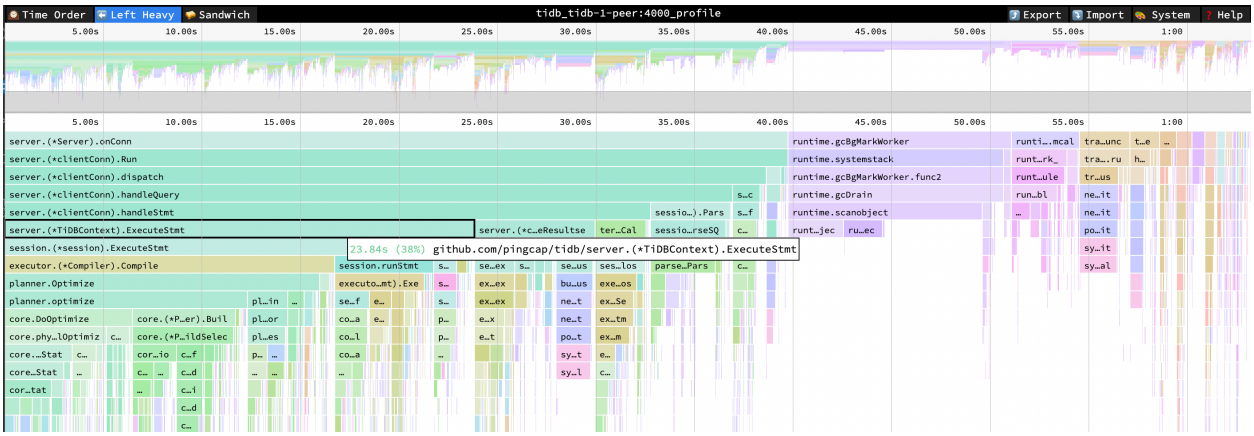


图 122: flame-graph-for-query-interface

- ExecuteStmt cpu = 38% cpu time = 23.84s
- Compile cpu = 27% cpu time = 17.17s
- Optimize cpu = 26% cpu time = 16.41s

Performance Overview 面板

观察以下 Performance Overview 面板中数据库时间概览和 QPS 的数据：

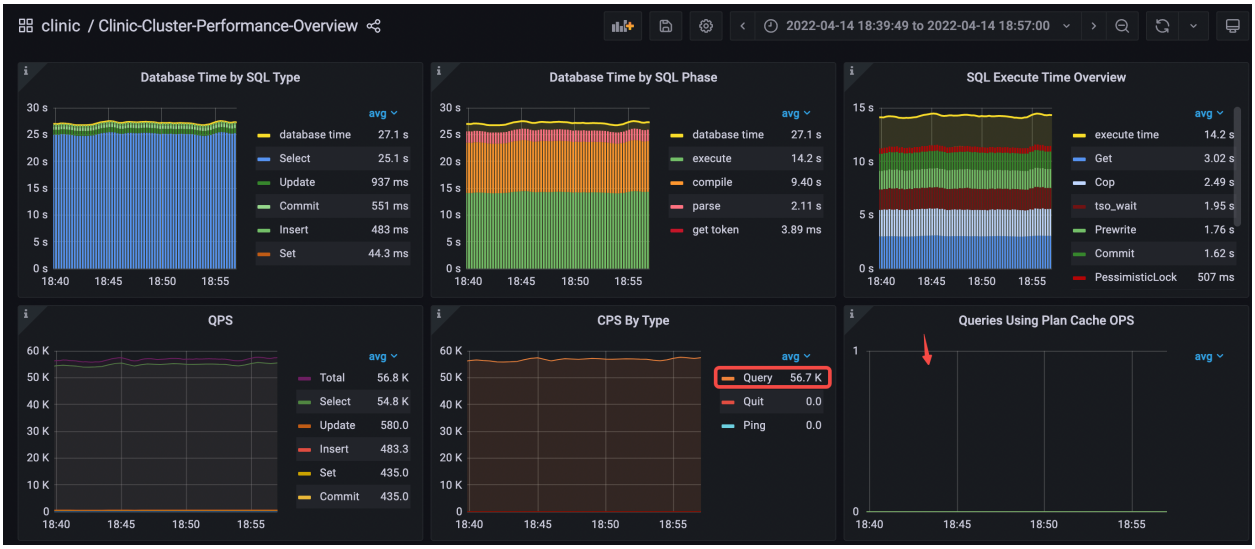


图 123: performance-overview-1-for-query-interface

- Database Time by SQL Type 中 Select 语句耗时最多
- Database Time by SQL Phase 中 execute 和 compile 占比最多
- SQL Execute Time Overview 中占比最多的分别是 Get、 Cop 和 tso wait
- CPS By Type 只有 query 这一种 command
- Queries Using Plan Cache OPS 没有数据，说明无法命中执行计划缓存

- execute 和 compile 的延迟在 query duration 中占比最高
- avg QPS = 56.8k

观察集群的资源消耗，TiDB CPU 的平均利用率为 925%，TiKV CPU 的平均利用率为 201%，TiKV IO 平均吞吐为 18.7 MB/s。TiDB 的资源消耗明显更高。

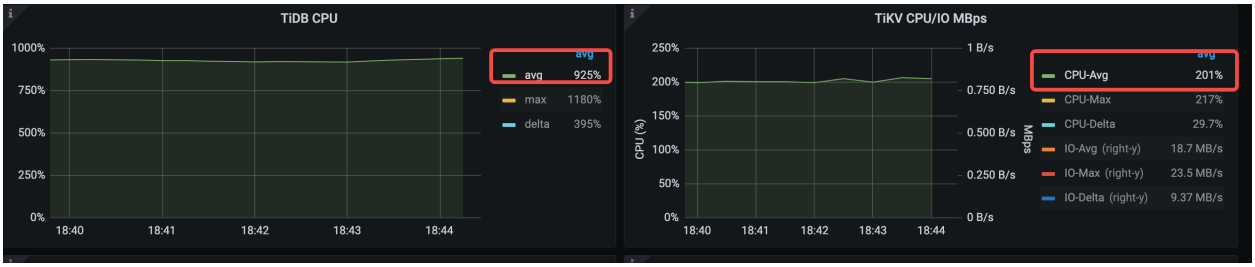


图 124: performance-overview-2-for-query-interface

11.1.3.2.3 分析结论

需要屏蔽这些大量无用的非业务 SQL 语句。

11.1.3.3 场景 2：使用 maxPerformance 配置

11.1.3.3.1 应用配置

在场景 1 中的 JDBC 连接串的基础上，新增一个参数 `useConfigs=maxPerformance`。这个参数可以用来屏蔽 JDBC 向数据库发送的一些查询设置类的 SQL 语句（例如 `select @@session.transaction_read_only`），完整配置如下：

```
useServerPrepStmts=false&useConfigs=maxPerformance
```

11.1.3.3.2 性能分析

TiDB Dashboard

在 Dashboard 的 Top SQL 页面，可以看到原本占比最多的 `SELECT @@session.tx_isolation` 已消失。

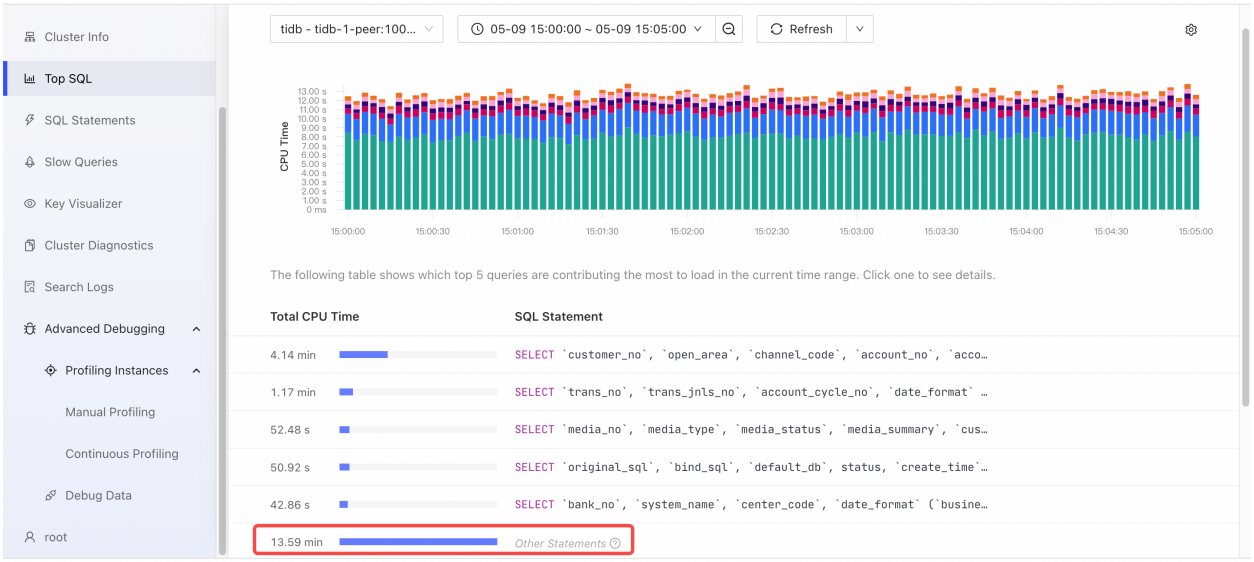


图 125: dashboard-for-maxPerformance

观察以下 TiDB 的火焰图，可以发现 SQL 语句执行中 Compile 和 Optimize 等函数 CPU 消耗占比高：

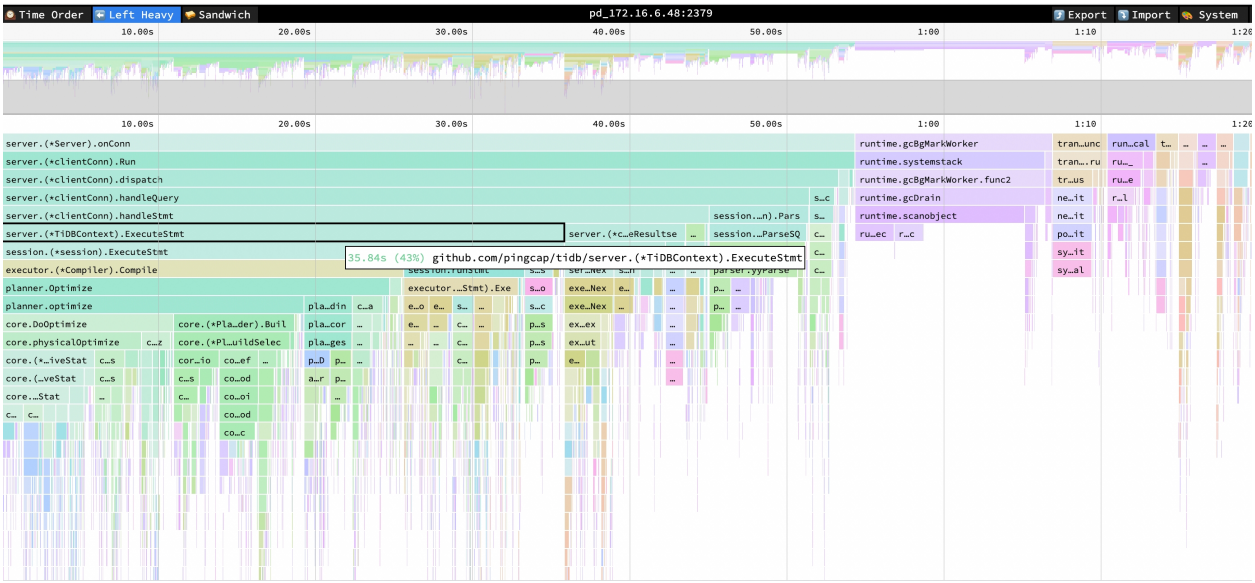


图 126: flame-graph-for-maxPerformance

- ExecuteStmt cpu = 43% cpu time =35.84s
- Compile cpu = 31% cpu time =25.61s
- Optimize cpu = 30% cpu time = 24.74s

Performance Overview 面板

数据库时间概览和 QPS 的数据如下：

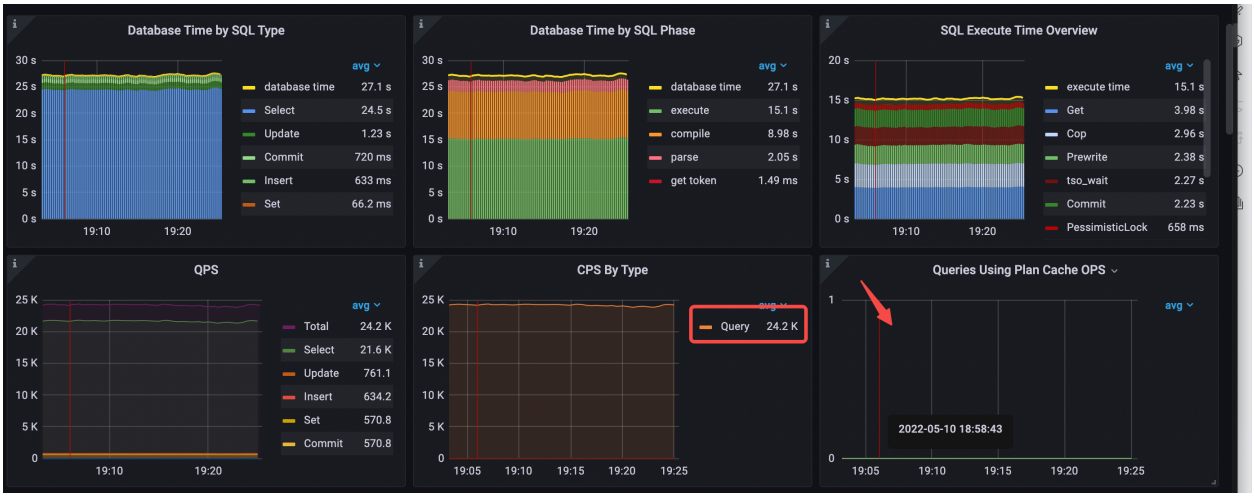


图 127: performance-overview-1-for-maxPerformance

- Database Time by SQL Type 中 select 语句耗时最多
- Database Time by SQL Phase 中 execute 和 compile 占比最多
- SQL Execute Time Overview 中占比最多的分别是 Get、 Cop、 Prewrite 和 tso_wait
- execute 和 compile 的延迟在 db time 中占比最高
- CPS By Type 只有 query 这一种 command
- avg QPS = 24.2k (56.3k->24.2k)
- 无法命中 plan cache

从场景 1 到场景 2，TiDB CPU 平均利用率从 925% 下降到 874%，TiKV CPU 平均利用率从 201% 上升到 250% 左右。

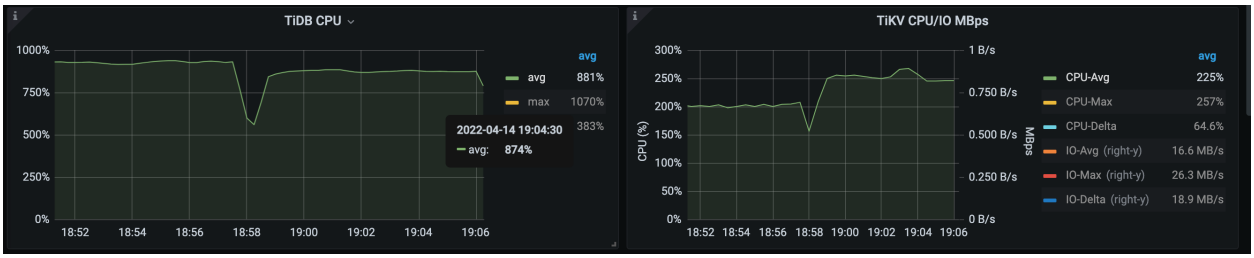


图 128: performance-overview-2-for-maxPerformance

关键延迟指标变化如下：



图 129: performance-overview-3-for-maxPerformance

- avg query duration = 1.12ms (479µs->1.12ms)
- avg parse duration = 84.7µs (37.2µs->84.7µs)
- avg compile duration = 370µs (166µs->370µs)
- avg execution duration = 626µs (251µs->626µs)

11.1.3.3.3 分析结论

和场景 1 相比，场景 2 的 QPS 有明显的下降，平均 query duration 和 parse、compile 以及 execute duration 有明显的上升。这是因为场景 1 中类似 `select @@session.transaction_read_only` 这样执行次数多且处理时间快的 SQL 语句拉低了性能平均值，场景 2 屏蔽这类语句后只剩下纯业务 SQL，从而带来了 duration 平均值的上升。

当应用使用 query 接口时，TiDB 无法使用执行计划缓存，编译执行计划消耗高。此时，建议使用 Prepared Statement 预编译接口，利用 TiDB 的执行计划缓存来降低 compile 带来的 TiDB CPU 消耗，降低延迟。

11.1.3.4 场景 3：使用 Prepared Statement 接口，未开启执行计划缓存

11.1.3.4.1 应用配置

应用程序使用以下连接配置，和场景 2 对比，JDBC 的 `useServerPrepStmts` 参数值修改为 `true`，表示启用了预编译语句的接口。

```
useServerPrepStmts=true&useConfigs=maxPerformance"
```

11.1.3.4.2 性能分析

TiDB Dashboard

观察以下 TiDB 的火焰图，可以发现启用 Prepared Statement 接口之后，CompileExecutePreparedStmt 和 Optimize 的 CPU 占比依然明显。

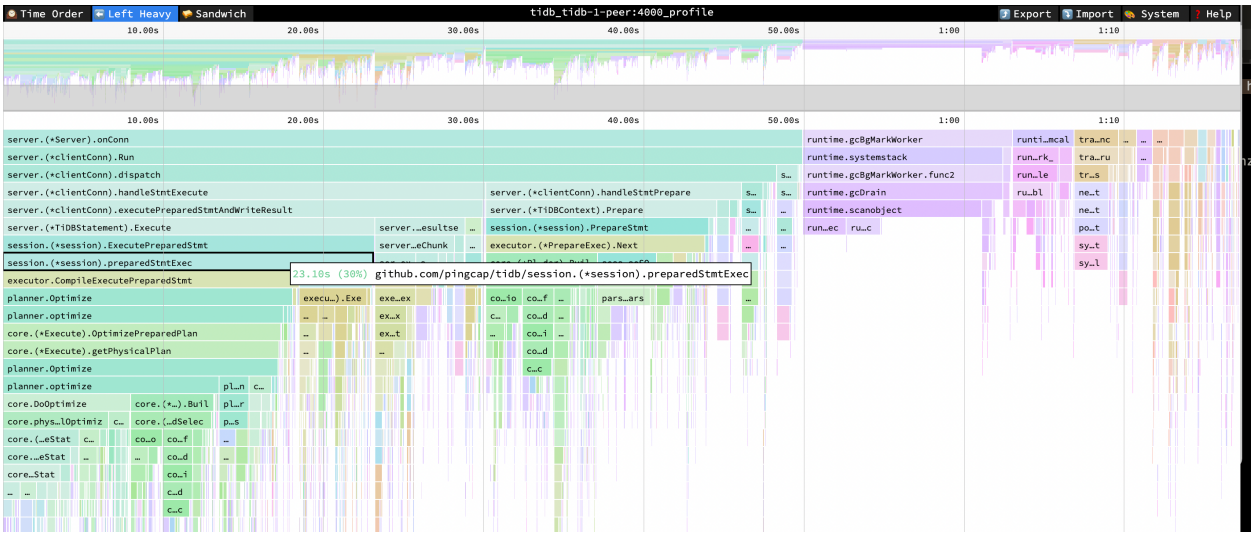


图 130: flame-graph-for-PrepStmts

- ExecutePreparedStmt cpu = 31% cpu time = 23.10s
- preparedStmtExec cpu = 30% cpu time = 22.92s
- CompileExecutePreparedStmt cpu = 24% cpu time = 17.83s
- Optimize cpu = 23% cpu time = 17.29s

Performance Overview 面板

使用 Prepared Statement 接口之后，数据库时间概览和 QPS 的数据如下：

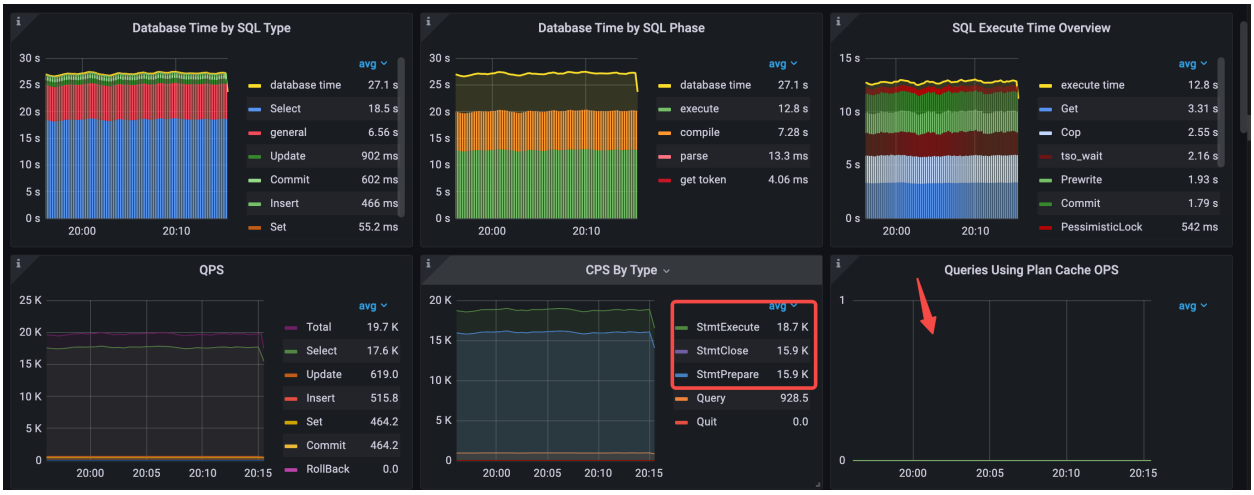


图 131: performance-overview-1-for-PrepStmts

QPS 从 24.4k 下降到 19.7k，从 CPS By Type 面板可以看到应用程序使用了三种 Prepared 命令。Database Time Overview 出现了 general 的语句类型（包含了 StmtPrepare 和 StmtClose 等命令的执行耗时），占比排名第二。这说明，即使使用了 Prepared Statement 接口，执行计划缓存也没有命中，原因在于 TiDB 内部处理 StmtClose 命令时，会清理修改语句的执行计划缓存。

- Database Time by SQL Type 中 select 语句耗时最多，其次是 general 语句
- Database Time by SQL Phase 中 execute 和 compile 占比最多
- SQL Execute Time Overview 中占比最多的分别是 Get、 Cop、 Prewrite 和 tso_wait
- CPS By Type 变成 3 种 command：StmtPrepare、 StmtExecute、 StmtClose
- avg QPS = 19.7k (24.4k->19.7k)
- 无法命中 plan cache

TiDB CPU 平均利用率从 874% 上升到 936%



图 132: performance-overview-1-for-PrepStmts

主要延迟数据如下：

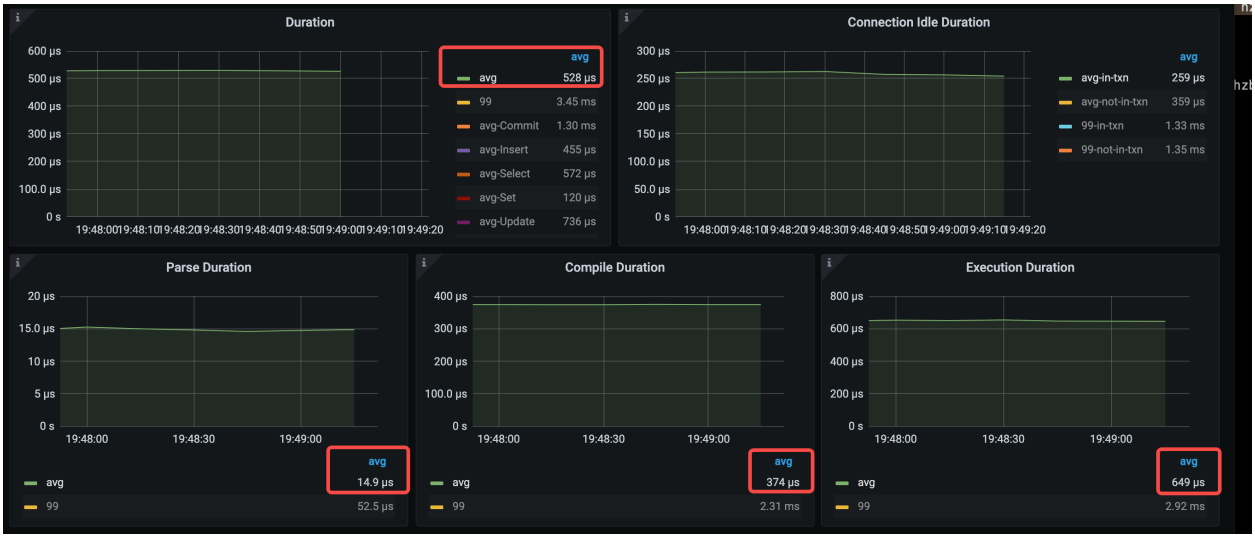


图 133: performance-overview-2-for-PrepStmts

- avg query duration = 528µs (1.12ms->528µs)
- avg parse duration = 14.9µs (84.7µs->14.9µs)
- avg compile duration = 374µs (370µs->374µs)
- avg execution duration = 649µs (626µs->649µs)

11.1.3.4.3 分析结论

和场景 2 不同的是，场景 3 启用了 prepare 预编译接口但是仍然无法命中缓存。此外，场景 2 的 CPS By Type 只有 query 这一种 command 类型，场景 3 多了 3 种 command 类型（StmtPrepare、StmtExecute、StmtClose）。与场景 2 相比，相当于多了两次网络往返的延迟。

- QPS 的降低原因分析：从 CPS By Type 面板可以看到，场景 2 只有 query 这一种 command 类型，但场景 3 新增了 3 种 command 类型，即 StmtPrepare、StmtExecute 和 StmtClose。其中，StmtExecute 和 query 为常规类型 command，会被 QPS 统计，而 StmtPrepare 和 StmtClose 为非常规类型 command，不会被 QPS 统计，所以 QPS 降低了。非常规类型 command 的 StmtPrepare 和 StmtClose 被统计在 general SQL 类型中，因此可以看到 database overview 中多了 general 的时间，且占比在 database time 的四分之一以上。
- 平均 query duration 明显降低原因分析：场景 3 新增了 StmtPrepare 和 StmtClose 这两种 command 类型，TiDB 内部处理时，query duration 也会单独计算，这两类命令处理速度很快，所以平均 query duration 明显被拉低。

虽然场景 3 使用了 Prepare 预编译接口但是因为出现了 StmtClose 导致缓存失效，很多应用框架也会在 execute 后调用 close 方法来防止内存泄漏。从 v6.0.0 版本开始，你可以设置全局变量 `tidb_ignore_prepared_cache_close_stmt=on;`。设置后，即使应用调用了 StmtClose 方法，TiDB 也不会清除缓存的执行计划，使得下一次的 SQL 执行能重用现有的执行计划，避免重复编译执行计划。

11.1.3.5 场景 4：使用 Prepared Statement 接口，开启执行计划缓存

11.1.3.5.1 应用配置

应用配置保持不变。设置以下参数，解决即使应用触发 StmtClose 导致无法命中缓存的问题。

- 设置 TiDB 全局变量 `set global tidb_ignore_prepared_cache_close_stmt=on;`（TiDB v6.0.0 起正式使用，默认关闭）
- 设置 TiDB 配置项 `prepared-plan-cache: {enabled: true}` 开启 plan cache 功能

11.1.3.5.2 性能分析

TiDB Dashboard

观察 TiDB 的 CPU 火焰图，可以看到 CompileExecutePreparedStmt 和 Optimize 没有明显的 CPU 消耗。Prepare 命令的 CPU 占比 25%，包含了 PlanBuilder 和 parseSQL 等 Prepare 解析相关的函数。

PreparseStmt cpu = 25% cpu time = 12.75s



图 134: flame-graph-for-3-commands

Performance Overview 面板

在 Performance Overview 面板种，最显著的变化来自于 Compile 阶段的占比，从场景 3 每秒消耗 8.95 秒降低为 1.18 秒。执行计划缓存的命中次数大致等于 StmtExecute 次数。在 QPS 上升的前提下，每秒 Select 语句消耗的数据库时间降低了，general 类型的语句消耗时间变长。

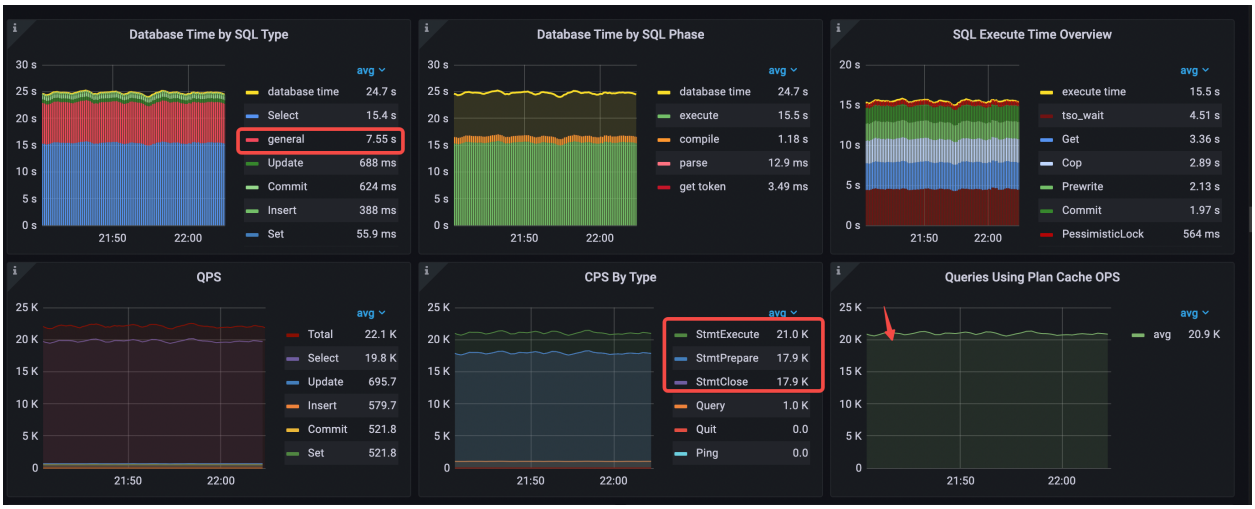


图 135: performance-overview-1-for-3-commands

- Database Time by SQL Type 中 select 语句耗时最多
- Database Time by SQL Phase 中 execute 占比最多
- SQL Execute Time Overview 中占比最多的分别是 tso wait、 Get 和 Cop
- 命中 plan cache，Queries Using Plan Cache OPS 大致等于 StmtExecute 每秒的次数
- CPS By Type 仍然是 3 种 command
- general time 相比场景 3 变长，因为 QPS 上升了
- avg QPS = 22.1k (19.7k->22.1k)

TiDB CPU 平均利用率从 936% 下降到 827%。

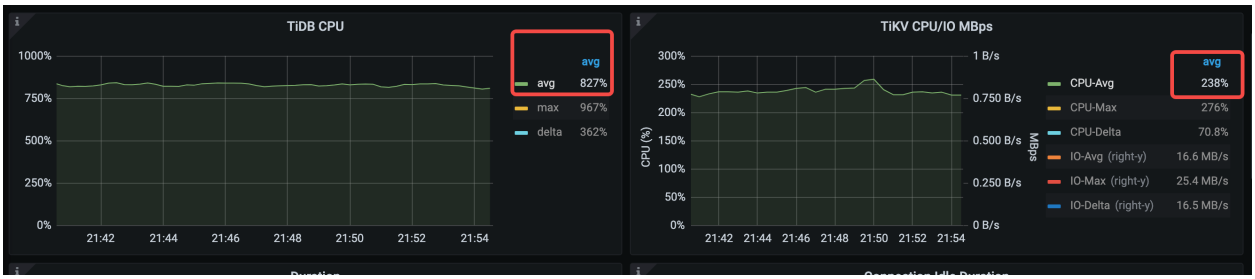


图 136: performance-overview-2-for-3-commands

Compile 平均时间显著下降，从 374 us 下降到 53.3 us，因为 QPS 的上升，平均 execute 时间有所上升。



图 137: performance-overview-3-for-3-commands

- avg query duration = 426µs (528µs->426µs)
- avg parse duration = 12.3µs (14.8µs->12.3µs)
- avg compile duration = 53.3µs (374µs->53.3µs)
- avg execution duration = 699µs (649µs->699us)

11.1.3.5.3 分析结论

和场景 3 相比，场景 4 同样存在 3 种 command 类型，不同的是场景 4 可以命中执行计划缓存，所以大大降低了 compile duration，同时降低了 query duration，并且提升了 QPS。

因为 StmtPrepare 和 StmtClose 两种命令消耗的数据库时间明显，并且增加了应用程序每执行一条 SQL 语句需要跟 TiDB 交互的次数。下一个场景将通过 JDBC 配置优化掉这两种命令。

11.1.3.6 场景 5：客户端缓存 prepared 对象

11.1.3.6.1 应用配置

和场景 4 相比，新增 3 个 JDBC 参数配置 `cachePrepStmts=true&prepStmtCacheSize=1000&prepStmtCacheSqlLimit=20480`，解释如下：

- `cachePrepStmts = true`：在客户端缓存 prepared statement 对象，消除 StmtPrepare 和 StmtClose 调用。
- `prepStmtCacheSize`：需要配置为大于 0 的值
- `prepStmtCacheSqlLimit`：需要设置为大于 SQL 文本的长度

完整的 JDBC 参数配置如下：

```
useServerPrepStmts=true&cachePrepStmts=true&prepStmtCacheSize=1000&prepStmtCacheSqlLimit=20480&
    ↪ useConfigs=maxPerformance
```

11.1.3.6.2 性能分析

TiDB Dashboard

观察以下 TiDB 的火焰图，可以发现 Prepare 命令的高 CPU 消耗未再出现。

- ExecutePreparedStmt cpu = 22% cpu time = 8.4s



图 138: flame-graph-for-1-command

Performance Overview 面板

在 Performance Overview 面板中，最显著的变化是，CPS By Type 面板中三种 Stmt command 类型变成了一种，Database Time by SQL Type 面板中的 general 语句类型消失了，QPS 面板中 QPS 上升到了 30.9k。

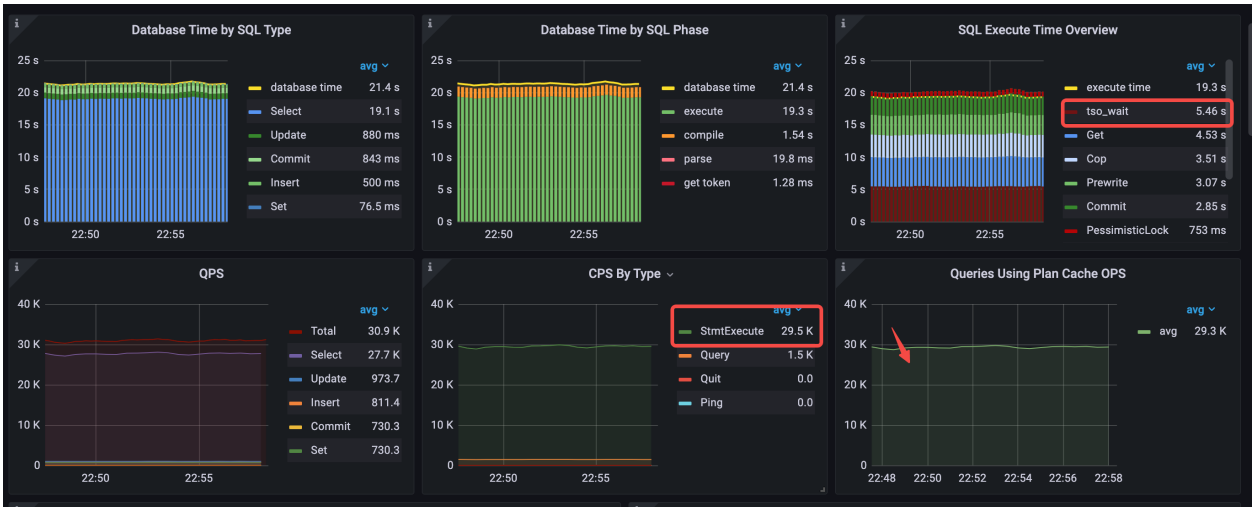


图 139: performance-overview-for-1-command

- Database Time by SQL Type 中 select 语句耗时最多，general 语句类型消失了
- Database Time by SQL Phase 中主要为 execute
- SQL Execute Time Overview 中占比最多的分别是 tso wait、 Get 和 Cop
- 命中 plan cache，Queries Using Plan Cache OPS 大致等于 StmtExecute 每秒的次数
- CPS By Type 只有一种 command，即 StmtExecute
- avg QPS = 30.9k (22.1k->30.9k)

TiDB CPU 平均利用率从 827% 下降到 577%，随着 QPS 的上升，TiKV CPU 平均利用率上升为 313%。



图 140: performance-overview-for-2-command

关键的延迟指标如下：

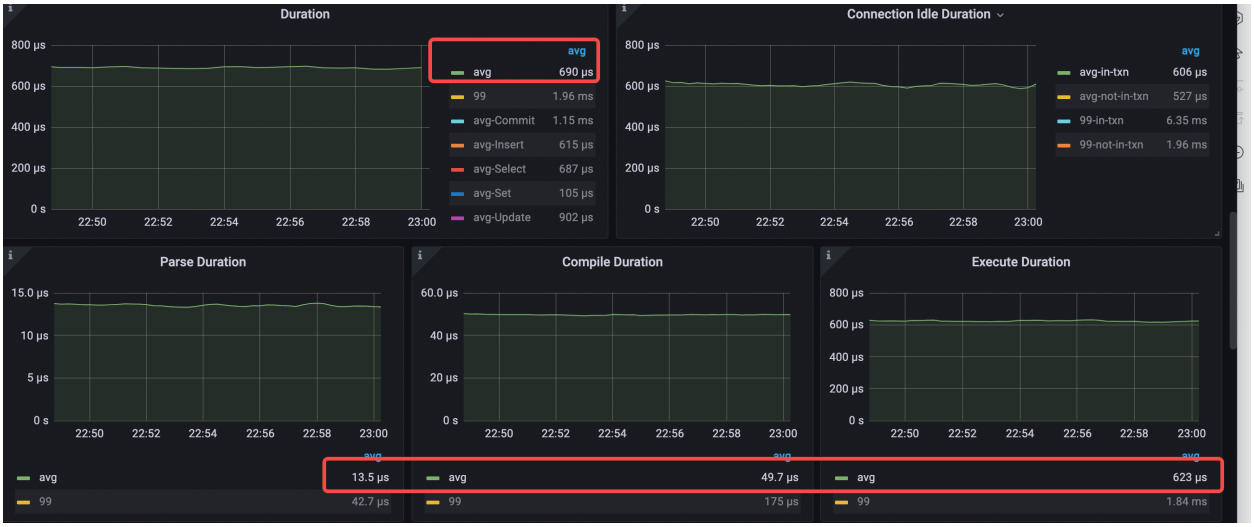


图 141: performance-overview-for-3-command

- avg query duration = 690µs (426->690µs)
- avg parse duration = 13.5µs (12.3µs->13.5µs)
- avg compile duration = 49.7µs (53.3µs->49.7µs)
- avg execution duration = 623µs (699us->623µs)
- avg pd tso wait duration = 196µs (224µs->196µs)
- connection idle duration avg-in-txn = 608µs (250µs->608µs)

11.1.3.6.3 分析结论

- 和场景 4 相比，场景 5 的 CPS By Type 只有 StmtExecute 这一种 command，减少了两次的网络往返，系统总体 QPS 上升。
- 在 QPS 上升的情况下，从 parse duration、compile duration、execution duration 来看延迟降低了，但 query duration 反而上升了。这是因为，StmtPrepare 和 StmtClose 处理的速度非常快，消除这两种 command 类型之后，平均的 query duration 就会上升。
- Database Time by SQL Phase 中 execute 占比非常高接近于 database time，同时 SQL Execute Time Overview 中占比最多的是 tso wait，超过四分之一的 execute 时间是在等待 tso。
- 每秒的 tso wait 总时间为 5.46s。平均 tso wait 时间为 196 us，每秒的 tso cmd 次数为 28k，非常接近于 QPS 的 30.9k。因为在 TiDB 对于 `read committed` 隔离级别的实现中，事务中的每个 SQL 语句都需要都到 PD 请求 tso。

TiDB v6.0 提供了 `rc read`，针对 `read committed` 隔离级别进行了减少 tso cmd 的优化。该功能由全局变量 `set global tidb_rc_read_check_ts=on;`控制。启用此变量后，TiDB 默认行为和 `repeatable-read` 隔离级别一致，只需要从 PD 获取 start-ts 和 commit-ts。事务中的语句先使用 start-ts 从 TiKV 读取数据。如果读到的数据小于 start-ts，则直接返回数据；如果读到大于 start-ts 的数据，则需要丢弃数据，并向 PD 请求 TSO 再进行重试。后续语句的 for update ts 使用最新的 PD TSO。

11.1.3.7 场景 6：开启 tidb_rc_read_check_ts 变量降低 TSO 请求

11.1.3.7.1 应用配置

应用配置不变，和场景 5 不同的是，设置 `set global tidb_rc_read_check_ts=on;`，减少 TSO 请求。

11.1.3.7.2 性能分析

TiDB Dashboard

TiDB 的 CPU 火焰图没有明显变化。

- ExecutePreparedStmt cpu = 22% cpu time = 8.4s

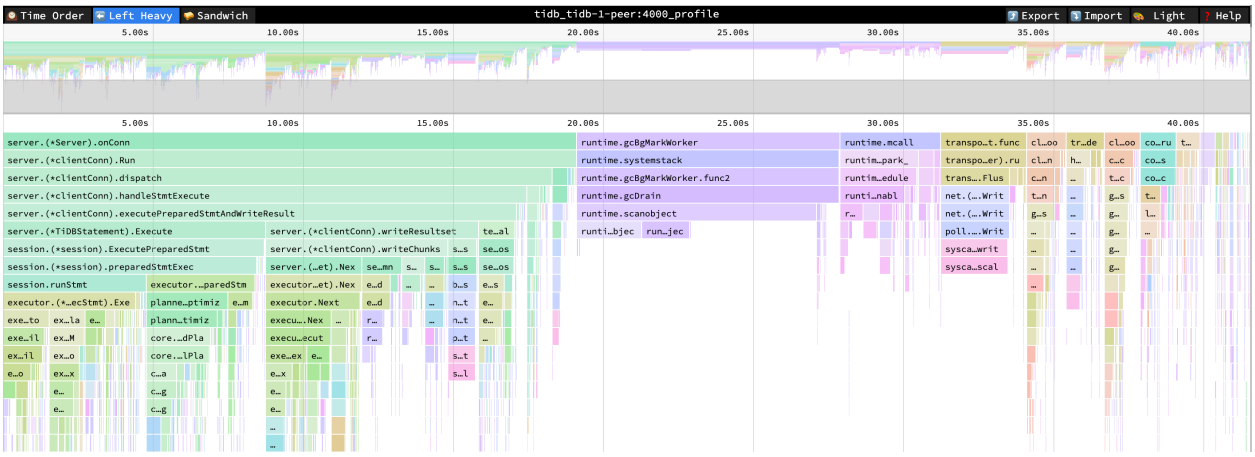


图 142: flame-graph-for-rc-read

Performance Overview 面板

使用 RC read 之后，QPS 从 30.9k 上升到 34.9k，每秒消耗的 tso wait 时间从 5.46 s 下降到 456 ms。

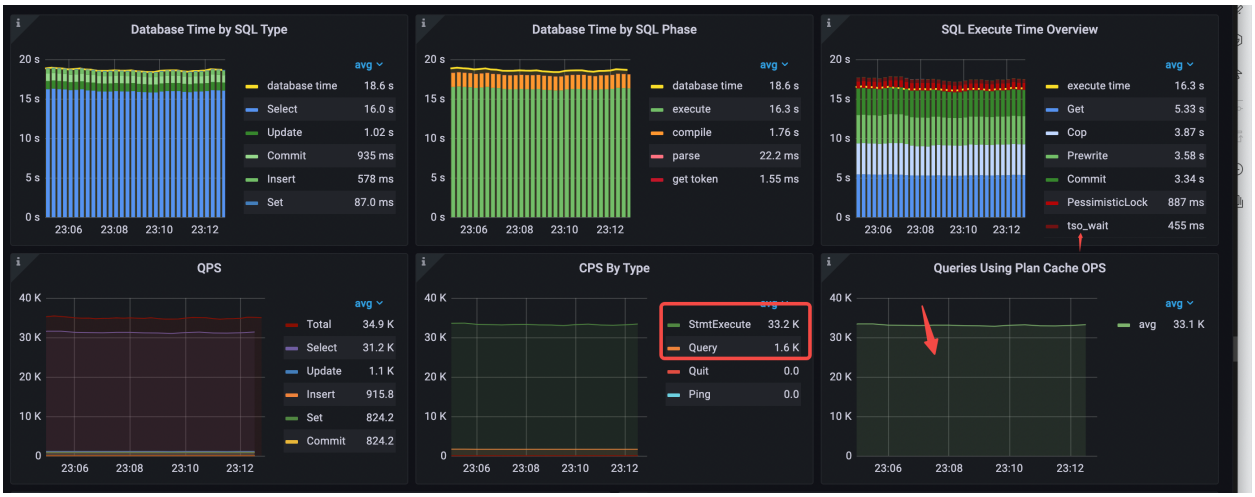


图 143: performance-overview-1-for-rc-read

- Database Time by SQL Type 中 select 语句耗时最多
- Database Time by SQL Phase 中 execute 占比最高
- SQL Execute Time Overview 中占比最多的分别是 Get、 Cop 和 Prewrite
- 命中 plan cache，Queries Using Plan Cache OPS 大致等于 StmtExecute 每秒的次数
- CPS By Type 只有一种 command
- avg QPS = 34.9k (30.9k->34.9k)

每秒 tso cmd 从 28.3k 下降到 2.7k。

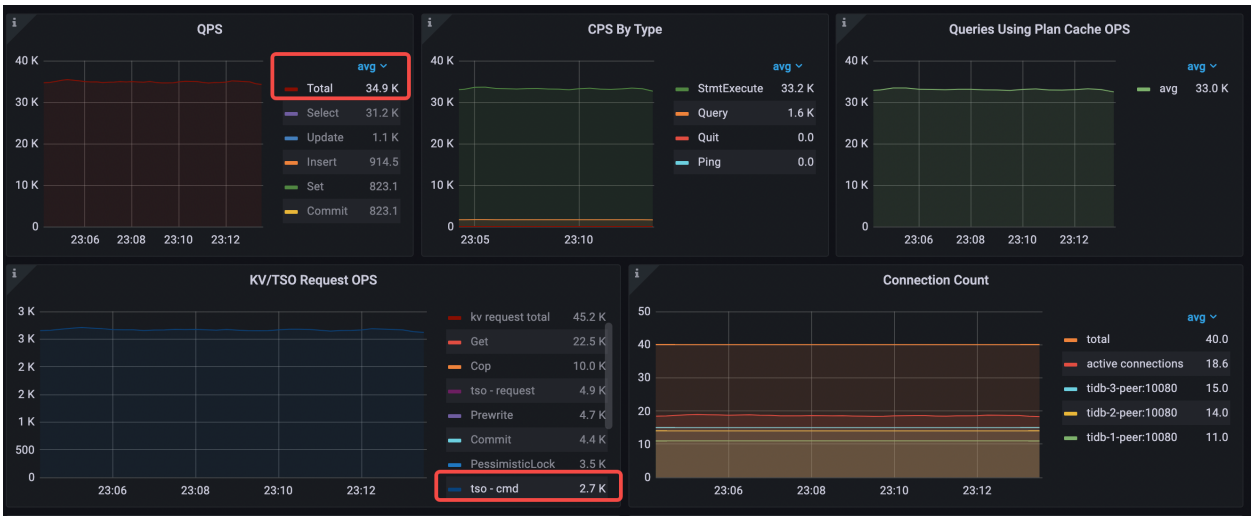


图 144: performance-overview-2-for-rc-read

平均 TiDB CPU 上升为 603% (577%->603%)。



图 145: performance-overview-3-for-rc-read

关键延迟指标如下：

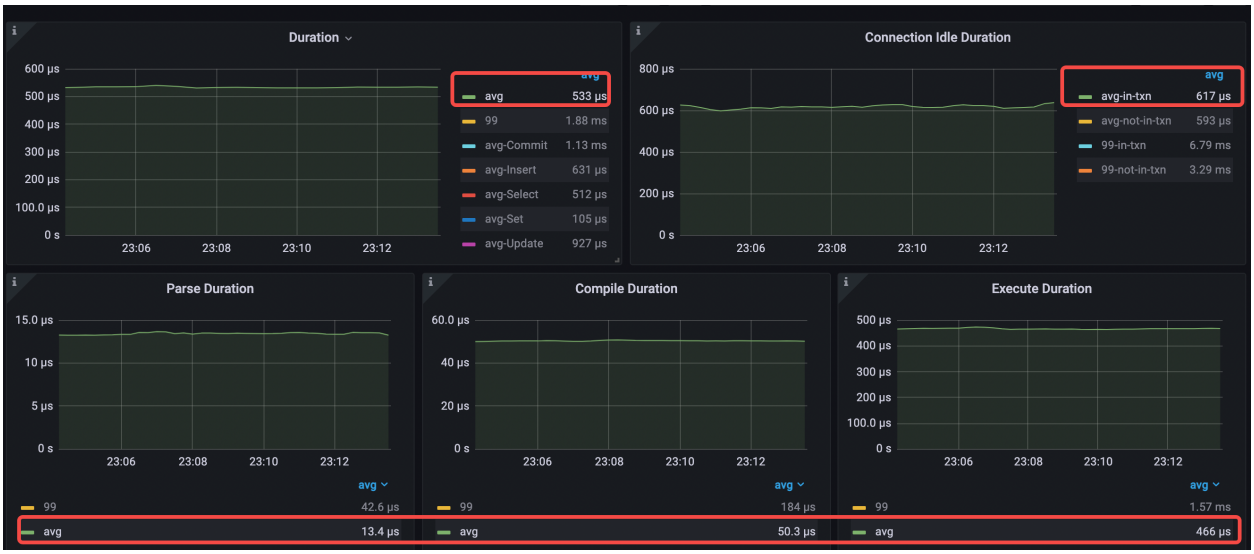


图 146: performance-overview-4-for-rc-read

- avg query duration = 533µs (690µs->533µs)
- avg parse duration = 13.4µs (13.5µs->13.4µs)
- avg compile duration = 50.3µs (49.7µs->50.3µs)
- avg execution duration = 466µs (623µs->466µs)
- avg pd tso wait duration = 171µs (196µs->171µs)

11.1.3.7.3 分析结论

通过 `set global tidb_rc_read_check_ts=on;` 启用 RC Read 之后，RC Read 明显降低了 tso cmd 次数从而降低了 tso wait 以及平均 query duration，并且提升了 QPS。

当前数据库时间和延迟瓶颈都在 execute 阶段，而 execute 阶段占比最高的为 Get 和 Cop 读请求。这个负载中，大部分表是只读或者很少修改，可以使用 v6.0.0 的小表缓存功能，使用 TiDB 缓存这些小表的数据，降低 KV 读请求的等待时间和资源消耗。

11.1.3.8 场景 7：使用小表缓存

11.1.3.8.1 应用配置

应用配置不变，在场景 6 的基础上设置了对业务的只读表进行缓存 `alter table t1 cache;`。

11.1.3.8.2 性能分析

TiDB Dashboard

TiDB CPU 火焰图没有明显变化。

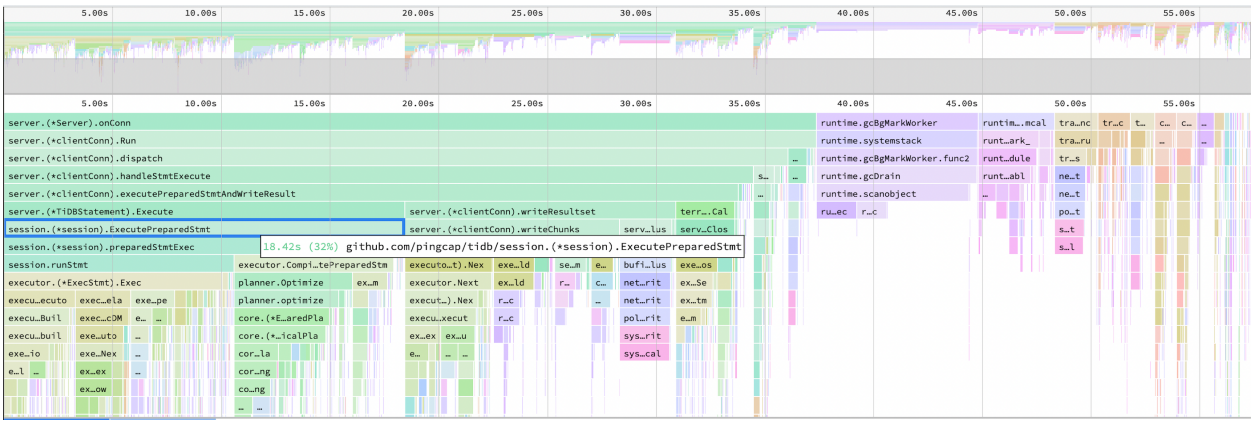


图 147: flame-graph-for-table-cache

Performance Overview 面板

QPS 从 34.9k 上升到 40.9k，execute 时间中占比最高的 KV 请求类型变成了 Prewrite 和 Commit。Get 每秒的时间从 5.33 秒下降到 1.75 秒，Cop 每秒的时间从 3.87 下降到 1.09 秒。

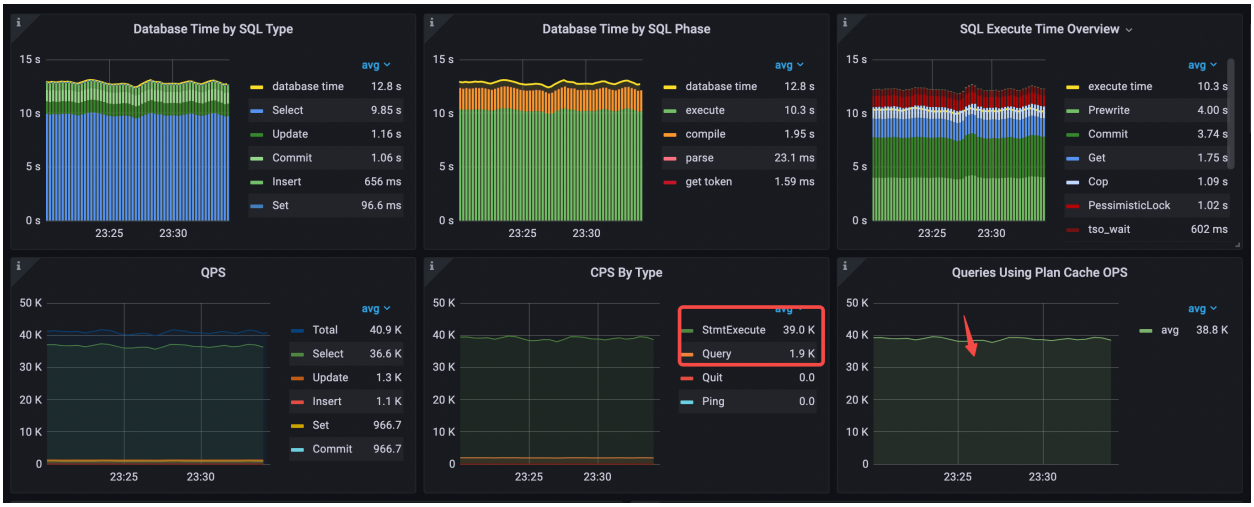


图 148: performance-overview-1-for-table-cache

- Database Time by SQL Type 中 select 语句耗时最多
- Database Time by SQL Phase 中 execute 和 compile 占比最多
- SQL Execute Time Overview 中占比最多的分别是 Prewrite、Commit 和 Get
- 命中 plan cache，Queries Using Plan Cache OPS 大致等于 StmtExecute 每秒的次数
- CPS By Type 只有 1 种 command
- avg QPS = 40.9k (34.9k->40.9k)

TiDB CPU 平均利用率从 603% 下降到 478%，TiKV CPU 平均利用率从 346% 下降到 256%。



图 149: performance-overview-2-for-table-cache

Query 平均延迟从 533 us 下降到 313 us。 execute 平均延迟从 466 us 下降到 250 us。

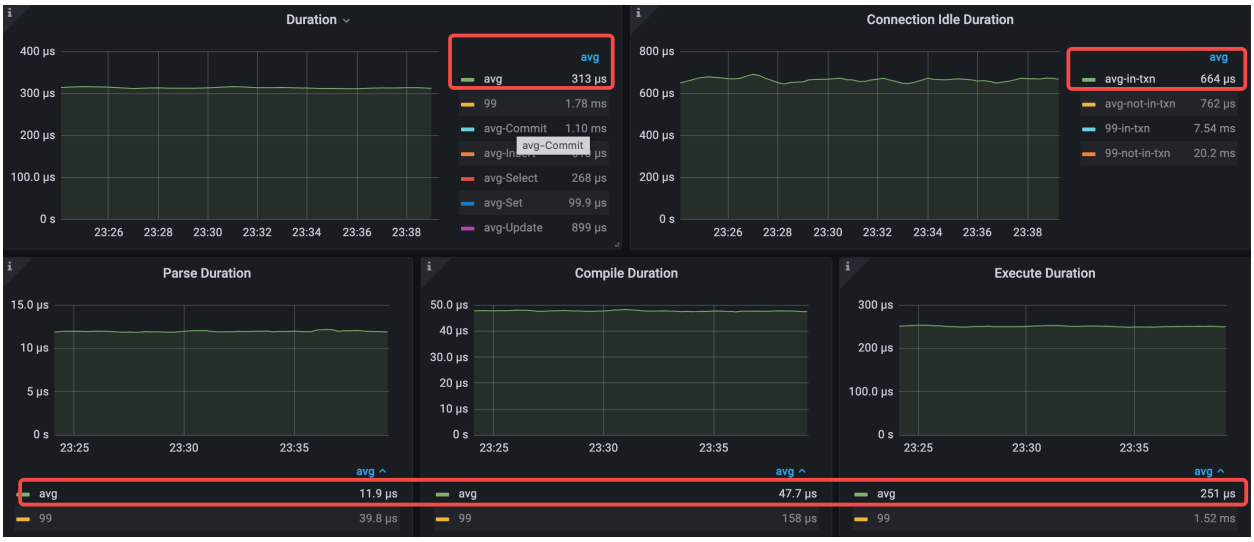


图 150: performance-overview-3-for-table-cache

- avg query duration = 313µs (533µs->313µs)
- avg parse duration = 11.9µs (13.4µs->11.9µs)
- avg compile duration = 47.7µs (50.3µs->47.7µs)
- avg execution duration = 251µs (466µs->251µs)

11.1.3.8.3 分析结论

将所有只读表进行缓存后，可以看到，execute duration 下降非常明显，原因是所有只读表都缓存在 TiDB 中，不再需要到 TiKV 中查询数据，所以 query duration 下降，QPS 上升。

这个是比较乐观的结果，实际的业务中可能只读表的数据量较大无法全部在 TiDB 中进行缓存。另一个限制是，当前小表缓存的功能虽然支持写操作，但是写操作默认需要等 3 秒，确保所有 TiDB 节点的缓存失效，对于对延迟要求比较高的应用，可能暂时不太友好。

11.1.3.9 总结

以下表格展示了七个不同场景的性能表现：

指标	场景 1	场景 2	场景 3	场景 4	场景 5	场景 6	场景 7	对比场景 5 和场景 2 (%)	对比场景 7 和场景 3(%)
query duration	479µs	1120µs	528µs	426µs	690µs	533µs	313µs	-38%	-51%
QPS	56.3k	24.2k	19.7k	22.1k	30.9k	34.9k	40.9k	+28%	+108%

其中，场景 2 是应用程序使用 Query 接口的常见场景，场景 5 是应用程序使用 Prepared Statement 接口的理想场景。

- 对比场景 2 和场景 5，可以发现通过使用 Java 应用开发的最佳实践以及客户端缓存 Prepared Statement 对象，每条 SQL 只需要一次命令和数据库交互，就能命中执行计划缓存，从而使 Query 延迟下降了 38%，QPS 上升 28%，同时，TiDB CPU 平均利用率从 936% 下降到 577%。
- 对比场景 2 和场景 7，可以看到在场景 5 的基础上使用了 RC Read、小表缓存等 TiDB 最新的优化功能，延迟降低了 51%，QPS 提升了 108%，同时，TiDB CPU 平均利用率从 936% 下降到 478%。

通过对比各场景的性能表现，可以得出以下结论：

- TiDB 的执行计划缓存对于 OLTP 发挥着至关重要的作用。而从 v6.0.0 开始引入的 RC Read 和小表缓存功能，在这个负载的深度优化中，也发挥了重要的作用。
- TiDB 兼容 MySQL 协议的不同命令，最佳的性能表现来自于应用程序使用 Prepared Statement 接口，并设置以下 JDBC 连接参数：

```
useServerPrepStmts=true&cachePrepStmts=true&prepStmtCacheSize=1000&prepStmtCacheSqlLimit
    ↪ =20480&useConfigs=maxPerformance
```

- 在性能分析和优化过程中，推荐使用 TiDB Dashboard （例如 Top SQL 功能和持续性能分析功能）和 Performance Overview 面板。
  - Top SQL 功能允许你可视化地监控和探索数据库中各个 SQL 语句在执行过程中的 CPU 开销情况，从而对数据库性能问题进行优化和处理。
  - 持续性能分析功能 可以持续地收集 TiDB、TiKV、PD 各个实例的性能数据。应用程序使用不同接口跟 TiDB 交互时，TiDB 的 CPU 消耗有着巨大的差距。
  - Performance Overview 面板 提供了数据库时间的概览和 SQL 执行时间分解信息。借助这个面板，你可以进行基于数据库时间的性能分析和诊断，确定整个系统的性能瓶颈是否处于 TiDB 中。如果瓶颈在 TiDB 中，你可以通过数据库时间和延迟的分解，以及集群关键指标和资源使用情况，确认 TiDB 内部的性能瓶颈，并进行针对性的优化。

综合使用以上几个功能，你可以针对现实中的应用进行高效的性能分析和优化。

11.1.4 TiFlash 性能分析和优化方法

本文介绍 TiFlash 资源使用率和关键的性能指标。你可以通过 Performance Overview 面板中的TiFlash 面板，来监控和评估 TiFlash 集群的性能。

11.1.4.1 TiFlash 集群资源利用率

通过以下三个指标，你可以快速判断 TiFlash 集群的资源使用率：

- CPU：每个 TiFlash 实例的 CPU 使用率
- Memory：每个 TiFlash 实例内存的使用情况
- IO utilization：每个 TiFlash 实例的 IO 使用率

示例：CH-benCHmark 负载资源使用率

该 TiFlash 集群包含两个节点，每个节点配置均为 16 核、48G 内存。当 CH-benCHmark 负载运行时，CPU 利用率最高可达到 1500%，内存占用最大可达 20 GB，IO 利用率达到 91%。这表明 TiFlash 节点资源接近饱和状态。

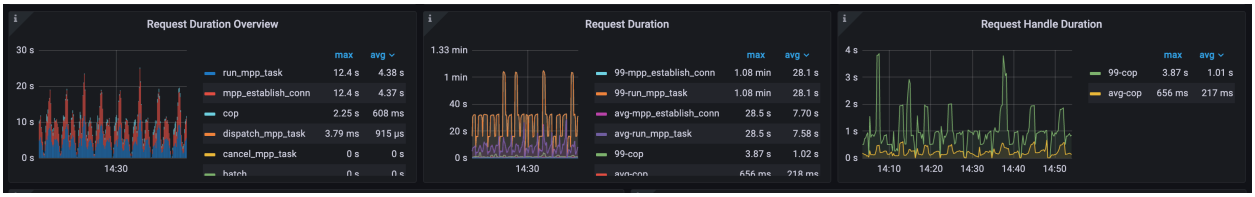


图 151: CH-TiFlash-MPP

11.1.4.2 TiFlash 关键性能指标

11.1.4.2.1 吞吐指标

通过以下指标，你可以了解 TiFlash 的吞吐情况：

- MPP Query count：每个 TiFlash 实例 MPP 查询数量的瞬时值，表示当前 TiFlash 实例需要处理的 MPP 查询数量（包括正在处理的以及还没被调度到的）。
- Request QPS：所有 TiFlash 实例收到的 coprocessor 请求数量。
  - `run_mpp_task`、`dispatch_mpp_task` 和 `mpp_establish_conn` 为 MPP 请求。
  - `batch`：batch 请求数量。
  - `cop`：直接通过 coprocessor 接口发送的 coprocessor 请求数量。
  - `cop_execution`：正在执行的 coprocessor 请求数量。
  - `remote_read`、`remote_read_constructed` 和 `remote_read_sent` 为 remote read 相关指标，remote read 增多一般意味着系统出现了问题。
- Executor QPS：所有 TiFlash 实例收到的请求中，每种 dag 算子的数量，其中 `table_scan` 是扫表算子，`selection` 是过滤算子，`aggregation` 是聚合算子，`top_n` 是 TopN 算子，`limit` 是 limit 算子，`join` 为关联算子，`exchange_sender` 和 `exchange_receiver` 为数据发送和接收算子。

11.1.4.2.2 延迟指标

通过以下指标，你可以了解 TiFlash 的延迟处理情况：

- Request Duration Overview：每秒所有 TiFlash 实例处理所有请求类型的总时长堆叠图。

  - 如果请求类型为 run_mpp_task、dispatch_mpp_task 或 mpp_establish_conn，说明 SQL 语句的执行已经部分或者完全下推到 TiFlash 上进行，通常包含 join 和数据分发的操作，这是 TiFlash 最常见的服务类型。
  - 如果请求类型为 cop，说明整个语句并没有完全下推到 TiFlash，通常 TiDB 会将全表扫描算子下推到 TiFlash 上进行数据访问和过滤。在堆叠图中，如果 cop 占据主导地位，需要仔细权衡是否合理。
    * 如果 SQL 访问的数据量很大，优化器可能根据成本模型估算 TiFlash 全表扫描的成本更低。
    * 如果表结构缺少合适的索引，即使访问的数据量很少，优化器也只能将查询下推到 TiFlash 进行全表扫描。在这种情况下，创建合适的索引，通过 TiKV 访问数据更加高效。

- Request Duration：所有 TiFlash 实例每种 MPP 和 coprocessor 请求类型的总处理时间，包含平均和 P99 处理延迟。
- Request Handle Duration：指 cop 和 batch cop 从开始执行到执行结束的时间，不包括等待时间，只包含 cop 和 batch cop 两种类型，包含平均和 P99 延迟。

示例 1：TiFlash MPP 请求处理时间概览

如下图所示，在此负载中，run_mpp_task 和 mpp_establish_conn 请求的处理时间占比最高，表明大部分请求都是完全下推到 TiFlash 上执行的 MPP 任务。

而 cop 请求处理时间占比较小，说明存在一部分请求是通过 coprocessor 下推到 TiFlash 上进行数据访问和过滤的。

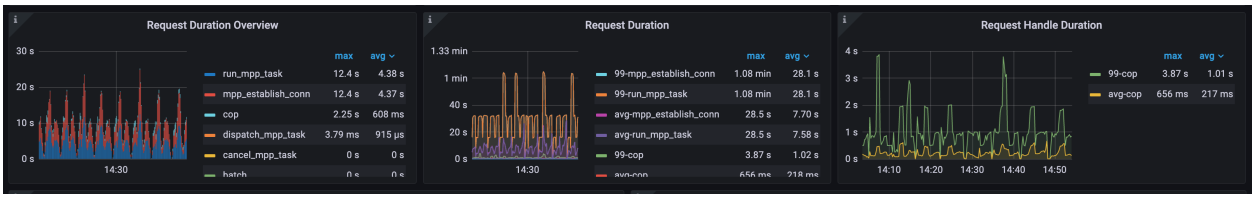


图 152: CH-TiFlash-MPP

示例 2：TiFlash cop 请求处理时间占比高

如下图所示，在此负载中，cop 请求的处理时间占比最高，可以通过查看 SQL 执行计划来确认 cop 请求产生的原因。

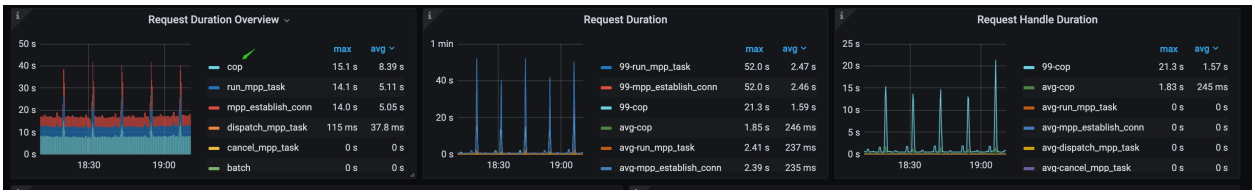


图 153: Cop

11.1.4.2.3 Raft 相关指标

通过以下指标，你可以了解 TiFlash 的 Raft 同步情况：

- Raft Wait Index Duration：所有 TiFlash 实例等待本地 Region index >= read_index 所花费的时间，即进行 wait_index 操作的延迟。如果 Wait Index 延迟过高，说明 TiKV 和 TiFlash 之间数据同步存在明显的延迟，可能的原因包括：
  - TiKV 资源过载
  - TiFlash 资源过载，特别是 IO 资源
  - TiKV 和 TiFlash 之间存在网络瓶颈
- Raft Batch Read Index Duration：所有 TiFlash 实例 Read Index 的延迟。如果该指标过高，说明 TiFlash 和 TiKV 之间的交互速度较慢，可能的原因包括：
  - TiFlash 资源过载
  - TiKV 资源过载
  - TiFlash 和 TiKV 之间存在网络瓶颈

11.1.4.2.4 IO 流量指标

通过以下指标，你可以了解 TiFlash 的 IO 流量情况：

- Write Throughput By Instance：每个 TiFlash 实例写入数据的吞吐量，包括 apply Raft 数据日志以及 Raft 快照的写入吞吐量。
- Write flow：所有 TiFlash 实例磁盘写操作的流量。
  - File Descriptor：TiFlash 所使用的 DeltaTree 存储引擎的稳定层。
  - Page：指 Pagestore，TiFlash 所使用的 DeltaTree 存储引擎的 Delta 变更层。
- Read flow：所有 TiFlash 实例磁盘读操作的流量。
  - File Descriptor：TiFlash 所使用的 DeltaTree 存储引擎的稳定层。
  - Page：指 Pagestore，TiFlash 所使用的 DeltaTree 存储引擎的 Delta 变更层。

你可以通过 `(Read flow + Write flow)÷ 总的 Write Throughput By Instance` 计算出整个 TiFlash 集群的写放大倍数。

示例 1：CH-benCHmark 负载本地部署环境 Raft 和 IO 指标

如下图所示，该 TiFlash 集群的 Raft Wait Index 和 Raft Batch Read Index 99 分位数较高，分别为 3.24 秒和 753 毫秒。这是因为该集群的 TiFlash 负载较高，数据同步存在延迟。

该集群包含两个 TiFlash 节点，每秒 TiKV 同步到 TiFlash 的增量数据约为 28 MB。稳定层 (File Descriptor) 的文件描述符最大写流量为 939 MB/s，最大读流量为 1.1 GiB/s，而 Delta 层 (Page) 最大写流量为 74 MB/s，最大读流量为 111 MB/s。该环境中的 TiFlash 使用独立的 NVME 盘，具有较强的 IO 吞吐能力。

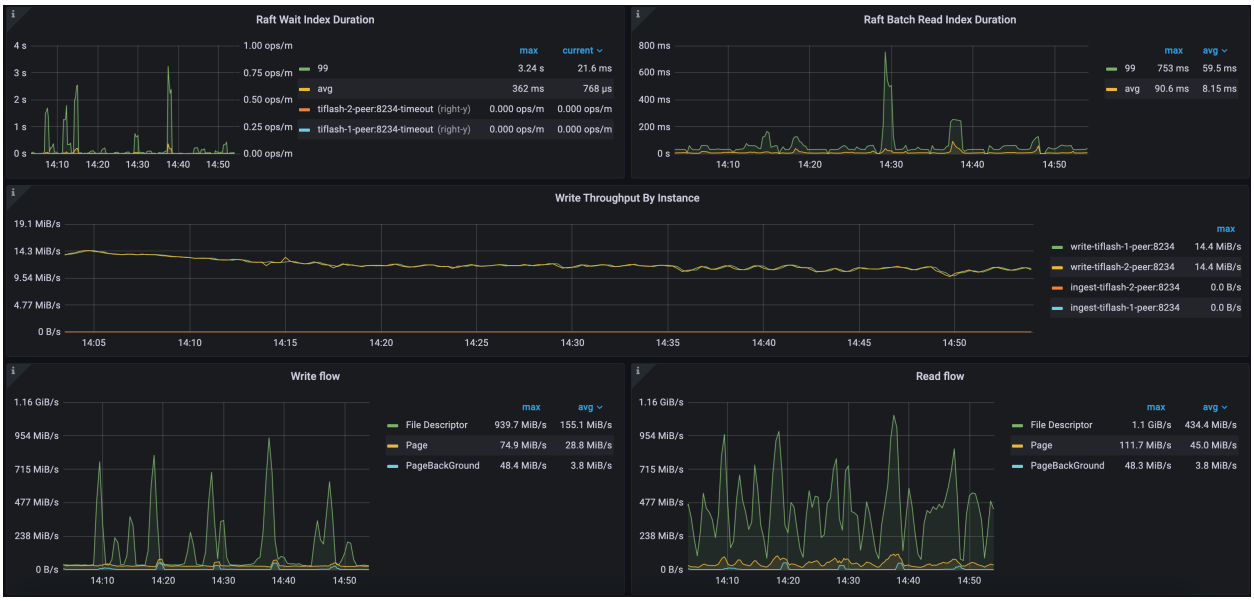


图 154: CH-2TiFlash-OP

示例 2： CH-benCHmark 负载 公有云环境 Raft 和 IO 指标

如下图所示，Raft Wait Index 等待时间 99 分位数最高为 438 毫秒，Raft Batch Read Index 等待时间 99 分位数最高为 125 毫秒。该集群只有一个 TiFlash 节点，每秒 TiKV 同步到 TiFlash 的增量数据约为 5 MB。稳定层 (File Descriptor) 的最大写入流量为 78 MB/s，最大读取流量为 221 MB/s，Delta 层 (Page) 最大写入流量为 8 MB/s，最大读取流量为 18 MB/s。这个环境中的 TiFlash 使用的是 AWS EBS 云盘，其 IO 吞吐能力相对较弱。



图 155: CH-TiFlash-MPP

11.1.5 TiCDC 性能分析和优化方法

本文介绍 TiCDC 资源使用率和关键的性能指标。你可以通过 Performance Overview 面板中的CDC 面板来监控和评估 TiCDC 同步数据的性能。

11.1.5.1 TiCDC 集群的资源利用率

通过以下三个指标，你可以快速判断 TiCDC 集群的资源使用率：

- CPU usage：TiCDC 节点的 CPU 使用情况
- Memory usage：TiCDC 节点的内存使用情况
- Goroutine count：TiCDC 节点 Goroutine 的个数

11.1.5.2 TiCDC 数据同步关键指标

11.1.5.2.1 TiCDC 整体指标

通过以下指标，你可以了解 TiCDC 数据同步的整体情况：

- Changefeed checkpoint lag：同步任务上下游数据的进度差，以时间单位秒计算。

  如果 TiCDC 消费数据的速度和写入下游的速度能跟上上游的数据变更，该指标将保持在较小的延迟范围内，通常是 10 秒以内。如果 TiCDC 消费数据的速度和写入下游的速度跟不上上游的数据变更，则该指标将持续增长。

  该指标增长（即 TiCDC checkpoint lag 增长）的常见原因如下：

  - 系统资源不足：如果 TiCDC 系统中的 CPU、内存或磁盘空间不足，可能会导致数据处理速度过慢，从而导致 TiCDC Changefeed checkpoint 过长。
  - 网络问题：如果 TiCDC 系统中存在网络中断、延迟或带宽不足的问题，可能会影响数据的传输速度，从而导致 TiCDC Changefeed checkpoint 过长。
  - 上游 QPS 过高：如果 TiCDC 系统需要处理的数据量过大，可能会导致数据处理超时，从而导致 TiCDC Changefeed checkpoint 增长，通常一个 TiCDC 节点处理的 QPS 上限为 60K 左右。
  - 数据库问题：
    * 上游 TiKV 集群 `min resolved ts` 和最新的 PD TSO 差距过大，通常是因为上游写入负载过大，TiKV 无法及时推进 resolved ts 造成。
    * 下游数据库写入延迟高导致 TiCDC 无法及时将数据同步到下游。

- Changefeed resolved ts lag：TiCDC 节点内部同步状态与上游的进度差，以时间单位秒计算。如果 TiCDC Changefeed resolved ts lag 值很高，可能意味着 TiCDC 系统的 Puller 或者 Sorter 模块数据处理能力不足，或者可能存在网络延迟或磁盘读写速度慢的问题。在这种情况下，需要采取适当的措施，例如增加 TiCDC 实例数量或优化网络配置，以确保 TiCDC 系统的高效和稳定运行。

- The status of changefeeds：Changefeed 各状态的解释，请参考Changefeed 状态流转。

示例 1：单个 TiCDC 节点上游 QPS 过高导致 checkpoint lag 过高

如下图所示，因为上游 QPS 过高，该集群中只有单个 TiCDC 节点，TiCDC 节点处于过载状态，CPU 使用率较高，Changefeed checkpoint lag 和 Changefeed resolved ts lag 持续增长。Changefeeds 的状态间歇性地从 0 变为 1，意味着 changefeed 不断出错。你可尝试通过增加资源解决该问题：

- 添加 TiCDC 节点：将 TiCDC 集群扩展到多个节点，以增加处理能力。
- 优化 TiCDC 节点的资源：提高 TiCDC 节点的 CPU 和内存配置，以改善性能。



图 156: TiCDC overview

11.1.5.2.2 数据流吞吐指标和下游延迟信息

通过以下指标，你可以了解数据流的吞吐和下游延迟信息：

- Puller output events/s：TiCDC 节点中 Puller 模块每秒输出到 Sorter 模块的数据变更行数
- Sorter output events/s：TiCDC 节点中 Sorter 模块每秒输出到 Mounter 模块的行数
- Mounter output events/s：TiCDC 节点中 Mounter 模块每秒输出到 Sink 模块的行数
- Table sink output events/s：TiCDC 节点中 Table Sorter 模块每秒输出到 Sink 模块的行数
- SinkV2 - Sink flush rows/s：TiCDC 节点中 Sink 模块每秒输出到下游的行数
- Transaction Sink Full Flush Duration：TiCDC 节点中 MySQL Sink 写下游事务的平均延迟和 p999 延迟
- MQ Worker Send Message Duration Percentile：下游为 Kafka 时 MQ worker 发送消息的延迟
- Kafka Outgoing Bytes：MQ Workload 写下游事务的流量

示例 2：下游数据库写入速度对 TiCDC 数据同步性能的影响

如下图所示，该环境上下游都为 TiDB 集群。通过 `TiCDC Puller output events/s` 可以确认上游数据库的 QPS 值。通过 `Transaction Sink Full Flush Duration` 可以确认，第一段负载的下游数据库平均写入延迟高，第二段负载的下游平均写入延迟低。

- 在第一段负载期间，由于下游 TiDB 集群写入数据缓慢，导致 TiCDC 消费数据的速度跟不上上游的 QPS，引起 Changefeed checkpoint lag 不断增长。然而，Changefeed resolved ts lag 仍然在 300 毫秒以内，说明同步延迟和吞吐瓶颈不在 puller 和 sorter 模块中，而在下游的 sink 模块。
- 在第二段负载期间，因为下游 TiDB 集群的写入速度快，TiCDC 同步数据的速度完全追上了上游的速度，因此 Changefeed checkpoint lag 和 Changefeed resolved ts lag 保持在 500 毫秒以内，此时 TiCDC 的同步速度较为理想。



图 157: TiCDC overview



图 158: data flow and txn latency

11.1.6 延迟的拆解分析

本文将 TiDB 中 SQL 语句的延迟拆解成各项监控指标，并从用户角度对指标进行分析，包括：

- 通用 SQL 层
- 读请求
- 写请求
- 批量请求
- TiKV 快照
- 异步写入

这些分析可以让你深入了解 TiDB 在执行 SQL 查询时的耗时情况，有助于诊断 TiDB 关键运行路径的问题。除了延迟的指标拆解之外，诊断场景小节介绍了如何在真实场景中分析延迟。

建议在阅读本文前，先阅读TiDB 性能分析和优化方法。需要注意的是，在将延迟拆解成监控指标时，延迟的耗时数值采用的是均值，而非某几个特定慢查询对应的数值。许多指标都会以直方图的形式展示，以更好地展现耗时或者延迟的分布情况。你需要按如下公式使用总和 (sum) 及数量 (count) 来计算均值 (avg)。

```
avg = ${metric_name}_sum / ${metric_name}_count
```

本文介绍的监控指标可以从 TiDB 的 Prometheus 管理界面中查询到。

11.1.6.1 通用 SQL 层

通用 SQL 层带来的延迟存在于 TiDB 的最顶部，所有 SQL 查询都具有这一部分的延迟。下面是通用 SQL 层操作的时间消耗图：

```
Diagram(
    NonTerminal("Token wait duration"),
    Choice(
        0,
        Comment("Prepared statement"),
        NonTerminal("Parse duration"),
    ),
    OneOrMore(
        Sequence(
        Choice(
            0,
            NonTerminal("Optimize prepared plan duration"),
            Sequence(
            Comment("Plan cache miss"),
            NonTerminal("Compile duration"),
            ),
        ),
        NonTerminal("TSO wait duration"),
        NonTerminal("Execution duration"),
        ),
        Comment("Retry"),
    ),
)
```

通用 SQL 层的延迟可以使用 `e2e duration` 指标观察。它的计算方式是：

```
e2e duration =
    tidb_server_get_token_duration_seconds +
    tidb_session_parse_duration_seconds +
    tidb_session_compile_duration_seconds +
    tidb_session_execute_duration_seconds{type="general"}
```

- `tidb_server_get_token_duration_seconds` 代表令牌 (Token) 等待耗时，它通常小于 1 微秒，因而足以被忽略。

- `tidb_session_parse_duration_seconds` 代表把 SQL 查询解析成抽象语法树 (AST, Abstract Syntax Tree) 的耗时。要跳过这部分的耗时，可以使用PREPARE/EXECUTE 语句。
- `tidb_session_compile_duration_seconds` 代表把抽象语法树编译成执行计划的耗时。要跳过这部分的耗时，可以使用执行计划缓存。
- `tidb_session_execute_duration_seconds{type="general"}` 代表执行各种不同用户查询的耗时。这部分的耗时需要进行细粒度的拆解，以用来分析性能问题或瓶颈。

通常来说，OLTP (Online Transactional Processing) 工作负载可以分为读请求和写请求两类。下面的小节会分别对读请求和写请求进行介绍。这两类请求共享了一些关键代码，但执行方式是不一样的。

11.1.6.2 读请求

读请求只有一种处理形式。

11.1.6.2.1 点查 (Point Get)

下面是点查操作的时间消耗图：

```
Diagram(
    Choice(
        0,
        NonTerminal("Resolve TSO"),
        Comment("Read by clustered PK in auto-commit-txn mode or snapshot read"),
    ),
    Choice(
        0,
        NonTerminal("Read handle by index key"),
        Comment("Read by clustered PK, encode handle by key"),
    ),
    NonTerminal("Read value by handle"),
)
```

在点查过程中，`tidb_session_execute_duration_seconds{type="general"}` 使用如下方式计算：

```
tidb_session_execute_duration_seconds{type="general"} =
    pd_client_cmd_handle_cmds_duration_seconds{type="wait"} +
    read handle duration +
    read value duration
```

`pd_client_cmd_handle_cmds_duration_seconds{type="wait"}` 代表从 PD 中读取TSO 的耗时。在 auto-commit 事务模式下从聚簇索引主键或者快照中读取时，该数值为 0。

`read handle duration` 和 `read value duration` 使用如下方式计算：

```
read handle duration = read value duration =
    tidb_tikvclient_txn_cmd_duration_seconds{type="get"} =
    send request duration =
    tidb_tikvclient_request_seconds{type="Get"} =
```

```
    tidb_tikvclient_batch_wait_duration +
    tidb_tikvclient_batch_send_latency +
    tikv_grpc_msg_duration_seconds{type="kv_get"} +
    tidb_tikvclient_rpc_net_latency_seconds{store="?"}
```

`tidb_tikvclient_request_seconds{type="Get"}` 代表通过 gRPC 发往 TiKV 的批量 get 请求耗时。关于 `tidb_tikvclient_batch_wait_duration`、`tidb_tikvclient_batch_send_latency` 和 `tidb_tikvclient_rpc_net_latency_seconds{store="?"}` 等批量请求客户端的耗时计算方式，请参考批量请求小节。

`tikv_grpc_msg_duration_seconds{type="kv_get"}` 使用如下方式计算：

```
tikv_grpc_msg_duration_seconds{type="kv_get"} =
    tikv_storage_engine_async_request_duration_seconds{type="snapshot"} +
    tikv_engine_seek_micro_seconds{type="seek_average"} +
    read value duration +
    read value duration(non-short value)
```

此时，请求已经到达 TiKV。TiKV 在处理 get 请求时，会进行一次 seek 和一到两次 read 操作。其中，短数据的键和值被编码在一个 write CF 中，因而只需要进行一次 read 操作。TiKV 在处理 get 请求前会先获取一个快照。关于 TiKV 快照耗时的计算方式，请参考TiKV 快照小节。

`read value duration(from disk)` 使用如下方式计算：

```
read value duration(from disk) =
    sum(rate(tikv_storage_rocksdb_perf{metric="block_read_time",req="get/batch_get_command"})) /
        sum(rate(tikv_storage_rocksdb_perf{metric="block_read_count",req="get/
        batch_get_command"}))
```

TiKV 采用 RocksDB 作为存储引擎。如果 block cache 中找不到要求的值，TiKV 需要从磁盘中读取。对于 `tikv_storage_rocksdb_perf`，get 请求可以是 `get` 或者 `batch_get_command`。

11.1.6.2.2 批量点查 (Batch Point Get)

下面是批量点查操作的时间消耗图：

```
Diagram(
  NonTerminal("Resolve TSO"),
  Choice(
    0,
    NonTerminal("Read all handles by index keys"),
    Comment("Read by clustered PK, encode handle by keys"),
  ),
  NonTerminal("Read values by handles"),
)
```

在进行批量点查时，`tidb_session_execute_duration_seconds{type="general"}` 使用如下方式计算：

```
tidb_session_execute_duration_seconds{type="general"} =
    pd_client_cmd_handle_cmds_duration_seconds{type="wait"} +
```

```
    read handles duration +
    read values duration
```

批量点查的过程几乎与点查一致。不同的是，批量点查会同时得到多个值。

`read handles duration` 和 `read values duration` 使用如下方式计算：

```
read handles duration = read values duration =
    tidb_tikvclient_txn_cmd_duration_seconds{type="batch_get"} =
    send request duration =
    tidb_tikvclient_request_seconds{type="BatchGet"} =
    tidb_tikvclient_batch_wait_duration(transaction) +
    tidb_tikvclient_batch_send_latency(transaction) +
    tikv_grpc_msg_duration_seconds{type="kv_batch_get"} +
    tidb_tikvclient_rpc_net_latency_seconds{store="?"}(transaction)
```

关于 `tidb_tikvclient_batch_wait_duration`、`tidb_tikvclient_batch_send_latency` 和 `tidb_tikvclient_rpc_net_latency_seco` `{store="?"}` 等批量请求客户端耗时的计算方式，请参考批量请求小节。

耗时 `tikv_grpc_msg_duration_seconds{type="kv_batch_get"}` 使用如下方式计算：

```
tikv_grpc_msg_duration_seconds{type="kv_batch_get"} =
    tikv_storage_engine_async_request_duration_seconds{type="snapshot"} +
    n * (
        tikv_engine_seek_micro_seconds{type="seek_max"} +
        read value duration +
        read value duration(non-short value)
    )

read value duration(from disk) =
    sum(rate(tikv_storage_rocksdb_perf{metric="block_read_time",req="batch_get"})) / sum(rate(
        tikv_storage_rocksdb_perf{metric="block_read_count",req="batch_get"}))
```

TiKV 会首先得到一个快照，然后从同一个快照中读取多个值。read 操作的耗时和点查中的一致。当从磁盘中读取数据时，其平均耗时可以通过带有 `req="batch_get"` 属性的 `tikv_storage_rocksdb_perf` 来计算。

11.1.6.2.3 表扫描和索引扫描 (Table Scan 和 Index Scan)

下面是表扫描和索引扫描的时间消耗图：

```
Diagram(
    Stack(
        NonTerminal("Resolve TSO"),
        NonTerminal("Load region cache for related table/index ranges"),
        OneOrMore(
            NonTerminal("Wait for result"),
            Comment("Next loop: drain the result"),
        ),
```

```
    ),
)
```

在进行表扫描和索引扫描时，耗时 `tidb_session_execute_duration_seconds{type="general"}` 使用如下方式计算：

```
tidb_session_execute_duration_seconds{type="general"} =
    pd_client_cmd_handle_cmds_duration_seconds{type="wait"} +
    req_per_copr * (
        tidb_distsql_handle_query_duration_seconds{sql_type="general"}
    )
    tidb_distsql_handle_query_duration_seconds{sql_type="general"} <= send request duration
```

表扫描和索引扫描采用相同的方式处理。`req_per_copr` 是被分配的任务数量。由于执行协处理器和返回数据在不同的线程中运行，`tidb_distsql_handle_query_duration_seconds{sql_type="general"}` 是等待时间，并且小于 `send request duration`。

`send request duration` 和 `req_per_copr` 使用如下方式计算：

```
send request duration =
    tidb_tikvclient_batch_wait_duration +
    tidb_tikvclient_batch_send_latency +
    tikv_grpc_msg_duration_seconds{type="coprocessor"} +
    tidb_tikvclient_rpc_net_latency_seconds{store="?"}

tikv_grpc_msg_duration_seconds{type="coprocessor"} =
    tikv_coprocessor_request_wait_seconds{type="snapshot"} +
    tikv_coprocessor_request_wait_seconds{type="schedule"} +
    tikv_coprocessor_request_handler_build_seconds{type="index/select"} +
    tikv_coprocessor_request_handle_seconds{type="index/select"}

req_per_copr = rate(tidb_distsql_handle_query_duration_seconds_count) / rate(
    ↪ tidb_distsql_scan_keys_partial_num_count)
```

在 TiKV 中，表扫描的类型是 `select`，而索引扫描的类型是 `index`。`select` 和 `index` 类型的内部耗时是一致的。

11.1.6.2.4 索引回表 (Index Look Up)

下面是通过索引回表操作的时间消耗图：

```
Diagram(
    Stack(
        NonTerminal("Resolve TSO"),
        NonTerminal("Load region cache for related index ranges"),
        OneOrMore(
            Sequence(
                NonTerminal("Wait for index scan result"),
                NonTerminal("Wait for table scan result"),
```

```
            ),
        Comment("Next loop: drain the result"),
        ),
    ),
)
```

在通过索引回表的过程中，耗时 `tidb_session_execute_duration_seconds{type="general"}` 使用如下方式计算:

```
tidb_session_execute_duration_seconds{type="general"} =
    pd_client_cmd_handle_cmds_duration_seconds{type="wait"} +
    req_per_copr * (
        tidb_distsql_handle_query_duration_seconds{sql_type="general"}
    ) +
    req_per_copr * (
        tidb_distsql_handle_query_duration_seconds{sql_type="general"}
    )

req_per_copr = rate(tidb_distsql_handle_query_duration_seconds_count) / rate(
    ↪ tidb_distsql_scan_keys_partial_num_count)
```

一次通过索引回表的过程结合了索引查找和表查找，其中索引查找和表查找按流水线方式处理。

11.1.6.3 写请求

写请求有多个变种，因而比读请求复杂得多。下面是写请求的时间消耗图：

```
Diagram(
    NonTerminal("Execute write query"),
    Choice(
        0,
        NonTerminal("Pessimistic lock keys"),
        Comment("bypass in optimistic transaction"),
    ),
    Choice(
        0,
        NonTerminal("Auto Commit Transaction"),
        Comment("bypass in non-auto-commit or explicit transaction"),
    ),
)
```

	悲观事务	乐观事务
Auto-commit	执行 + 加锁 + 提交	执行 + 提交
非 auto-commit	执行 + 加锁	执行

一次写请求可以被分解成以下三个阶段：

- 执行阶段：执行并把更改写入 TiDB 的内存中
- 加锁阶段：获取执行结果的悲观锁
- 提交阶段：通过两阶段提交协议 (2PC) 来提交事务

在执行阶段，TiDB 在内存中修改数据，其延迟主要源自读取所需的数据。对于更新和删除查询，TiDB 先从 TiKV 读取数据，再更新或删除内存中的数据。

带有点查和批量点查的加锁时读取操作 (SELECT FOR UPDATE) 是一个例外。该操作会在单个 RPC (Remote Procedure Call) 请求中完成读取和加锁操作。

11.1.6.3.1 加锁时点查

下面是加锁时点查的时间消耗图：

```
Diagram(
    Choice(
        0,
        Sequence(
            NonTerminal("Read handle key by index key"),
            NonTerminal("Lock index key"),
        ),
        Comment("Clustered index"),
    ),
    NonTerminal("Lock handle key"),
    NonTerminal("Read value from pessimistic lock cache"),
)
```

在加锁时点查过程中，耗时 `execution(clustered PK)` 和 `execution(non-clustered PK or UK)` 使用如下方式计算：

```
execution(clustered PK) =
    tidb_tikvclient_txn_cmd_duration_seconds{type="lock_keys"}
execution(non-clustered PK or UK) =
    2 * tidb_tikvclient_txn_cmd_duration_seconds{type="lock_keys"}
```

加锁时点查会锁定键并获取其对应的值。相比先执行再获取锁的方式，该操作可以节省一次来回通信。加锁时点查的耗时可以看作与加锁耗时是一样的。

11.1.6.3.2 加锁时批量点查

下面是加锁时批量点查的时间消耗图：

```
Diagram(
    Choice(
        0,
        NonTerminal("Read handle keys by index keys"),
```

```
        Comment("Clustered index"),
    ),
    NonTerminal("Lock index and handle keys"),
    NonTerminal("Read values from pessimistic lock cache"),
)
```

在加锁时批量点查过程中，耗时 `execution(clustered PK)` 和 `execution(non-clustered PK or UK)` 使用如下方式计算：

```
execution(clustered PK) =
    tidb_tikvclient_txn_cmd_duration_seconds{type="lock_keys"}
execution(non-clustered PK or UK) =
    tidb_tikvclient_txn_cmd_duration_seconds{type="batch_get"} +
    tidb_tikvclient_txn_cmd_duration_seconds{type="lock_keys"}
```

加锁时批量点查的执行过程与加锁时点查相似。不同的点在于，加锁时批量点查会在单个 RPC 请求中读取多个值。关于 `tidb_tikvclient_txn_cmd_duration_seconds{type="batch_get"}` 耗时的计算方式，请参考批量点查小节。

11.1.6.3.3 加锁阶段

本小节介绍加锁阶段的耗时。

```
round = ceil(
    sum(rate(tidb_tikvclient_txn_regions_num_sum{type="2pc_pessimistic_lock"})) /
    sum(rate(tidb_tikvclient_txn_regions_num_count{type="2pc_pessimistic_lock"})) /
    committer-concurrency
)

lock = tidb_tikvclient_txn_cmd_duration_seconds{type="lock_keys"} =
    round * tidb_tikvclient_request_seconds{type="PessimisticLock"}
```

锁是按照两阶段锁的结构来获取的，带有流量控制机制。流量控制会按照 `committer-concurrency`（默认值为 128）来限制并发在线请求的数量。为了简单说明，流量控制可以看作是请求延迟 (round) 的倍增。

`tidb_tikvclient_request_seconds{type="PessimisticLock"}` 使用如下方式计算：

```
tidb_tikvclient_request_seconds{type="PessimisticLock"} =
    tidb_tikvclient_batch_wait_duration +
    tidb_tikvclient_batch_send_latency +
    tikv_grpc_msg_duration_seconds{type="kv_pessimistic_lock"} +
    tidb_tikvclient_rpc_net_latency_seconds{store="?"}
```

关于 `tidb_tikvclient_batch_wait_duration`、`tidb_tikvclient_batch_send_latency` 和 `tidb_tikvclient_rpc_net_latency_seco`
↪ `{store="?"}` 等批量请求客户端耗时的计算方式，请参考批量请求小节。

耗时 `tikv_grpc_msg_duration_seconds{type="kv_pessimistic_lock"}` 使用如下方式计算：

```
tikv_grpc_msg_duration_seconds{type="kv_pessimistic_lock"} =
    tikv_scheduler_latch_wait_duration_seconds{type="acquire_pessimistic_lock"} +
    tikv_storage_engine_async_request_duration_seconds{type="snapshot"} +
    (lock in-mem key count + lock on-disk key count) * lock read duration +
    lock on-disk key count / (lock in-mem key count + lock on-disk key count) *
    lock write duration
```

- 自 TiDB v6.0 起，TiKV 默认使用内存悲观锁。内存悲观锁会跳过异步写入的过程。
- `tikv_storage_engine_async_request_duration_seconds{type="snapshot"}`是快照类型耗时，详情请参考TiKV 快照小节.
- `lock in-mem key count` 和 `lock on-disk key count` 使用如下方式计算：

```
lock in-mem key count =
    sum(rate(tikv_in_memory_pessimistic_locking{result="success"})) /
    sum(rate(tikv_grpc_msg_duration_seconds_count{type="kv_pessimistic_lock"}))

lock on-disk key count =
    sum(rate(tikv_in_memory_pessimistic_locking{result="full"})) /
    sum(rate(tikv_grpc_msg_duration_seconds_count{type="kv_pessimistic_lock"}))
```

内存和磁盘中被加锁的键数量可以从内存锁计数中计算得出。TiKV 在得到锁之前会读取键对应的值，其读取耗时可以从 RocksDB performance context 中计算得出：

```
lock read duration(from disk) =
    sum(rate(tikv_storage_rocksdb_perf{metric="block_read_time",req="
        ↪ acquire_pessimistic_lock"})) / sum(rate(tikv_storage_rocksdb_perf{metric="
        ↪ block_read_count",req="acquire_pessimistic_lock"}))
```

- `lock write duration` 是写入磁盘锁的耗时，具体计算方式请参考异步写入小节。

11.1.6.3.4 提交阶段

本小节介绍提交阶段的耗时。下面是提交操作的时间消耗图：

```
Diagram(
    Stack(
        Sequence(
            Choice(
                0,
                Comment("use 2pc or causal consistency"),
                NonTerminal("Get min-commit-ts"),
            ),
            Optional("Async prewrite binlog"),
            NonTerminal("Prewrite mutations"),
```

```
            Optional("Wait prewrite binlog result"),
        ),
        Sequence(
            Choice(
                1,
                Comment("1pc"),
                Sequence(
                    Comment("2pc"),
                    NonTerminal("Get commit-ts"),
                    NonTerminal("Check schema"),
                    NonTerminal("Commit PK mutation"),
                ),
                Sequence(
                    Comment("async-commit"),
                    NonTerminal("Commit mutations asynchronously"),
                ),
            ),
            Choice(
                0,
                Comment("committed"),
                NonTerminal("Async cleanup"),
            ),
            Optional("Commit binlog"),
        ),
    ),
)
```

提交的耗时使用如下方式计算：

```
commit =
    Get_latest_ts_time +
    Prewrite_time +
    Get_commit_ts_time +
    Commit_time

Get_latest_ts_time = Get_commit_ts_time =
    pd_client_cmd_handle_cmds_duration_seconds{type="wait"}

prewrite_round = ceil(
    sum(rate(tidb_tikvclient_txn_regions_num_sum{type="2pc_prewrite"})) /
    sum(rate(tidb_tikvclient_txn_regions_num_count{type="2pc_prewrite"})) /
    committer-concurrency
)

commit_round = ceil(
    sum(rate(tidb_tikvclient_txn_regions_num_sum{type="2pc_commit"})) /
```

```
    sum(rate(tidb_tikvclient_txn_regions_num_count{type="2pc_commit"})) /
    committer-concurrency
)

Prewrite_time =
    prewrite_round * tidb_tikvclient_request_seconds{type="Prewrite"}

Commit_time =
    commit_round * tidb_tikvclient_request_seconds{type="Commit"}
```

提交的耗时可以拆解为以下四个指标：

- `Get_latest_ts_time` 代表异步提交或单阶段 (1PC) 提交事务中获取最新 TSO 的耗时。
- `Prewrite_time` 代表预先写阶段的耗时。
- `Get_commit_ts_time` 代表普通两阶段 (2PC) 事务的耗时。
- `Commit_time` 代表提交阶段的耗时。需要注意的是，异步提交和单阶段 (1PC) 事务没有此阶段。

与悲观锁一样，流量控制充当延迟的放大，即上述公式中的 `prewrite_round` 和 `commit_round`。

`tidb_tikvclient_request_seconds{type="Prewrite"}` 和 `tidb_tikvclient_request_seconds{type="Commit"}` 的耗时使用如下方式计算：

```
tidb_tikvclient_request_seconds{type="Prewrite"} =
    tidb_tikvclient_batch_wait_duration +
    tidb_tikvclient_batch_send_latency +
    tikv_grpc_msg_duration_seconds{type="kv_prewrite"} +
    tidb_tikvclient_rpc_net_latency_seconds{store="?"}

tidb_tikvclient_request_seconds{type="Commit"} =
    tidb_tikvclient_batch_wait_duration +
    tidb_tikvclient_batch_send_latency +
    tikv_grpc_msg_duration_seconds{type="kv_commit"} +
    tidb_tikvclient_rpc_net_latency_seconds{store="?"}
```

关于 `tidb_tikvclient_batch_wait_duration`、`tidb_tikvclient_batch_send_latency` 和 `tidb_tikvclient_rpc_net_latency_seco`
↪ `{store="?"}` 等批量请求客户端耗时的计算方式，请参考批量请求小节。

`tikv_grpc_msg_duration_seconds{type="kv_prewrite"}` 使用如下方式计算：

```
tikv_grpc_msg_duration_seconds{type="kv_prewrite"} =
    prewrite key count * prewrite read duration +
    prewrite write duration

prewrite key count =
    sum(rate(tikv_scheduler_kv_command_key_write_sum{type="prewrite"})) /
    sum(rate(tikv_scheduler_kv_command_key_write_count{type="prewrite"}))
```

```
prewrite read duration(from disk) =
    sum(rate(tikv_storage_rocksdb_perf{metric="block_read_time",req="prewrite"})) / sum(rate(
        ↪ tikv_storage_rocksdb_perf{metric="block_read_count",req="prewrite"}))
```

与 TiKV 中的锁一样，预先写在读取和写入阶段均进行了处理。读取阶段的耗时可以从 RocksDB performance context 计算。有关写入阶段耗时的计算方式，请参考异步写入部分。

`tikv_grpc_msg_duration_seconds{type="kv_commit"}`使用如下方式计算：

```
tikv_grpc_msg_duration_seconds{type="kv_commit"} =
    commit key count * commit read duration +
    commit write duration

commit key count =
    sum(rate(tikv_scheduler_kv_command_key_write_sum{type="commit"})) /
    sum(rate(tikv_scheduler_kv_command_key_write_count{type="commit"}))

commit read duration(from disk) =
    sum(rate(tikv_storage_rocksdb_perf{metric="block_read_time",req="commit"})) / sum(rate(
        ↪ tikv_storage_rocksdb_perf{metric="block_read_count",req="commit"})) (storage)
```

`kv_commit` 的耗时与 `kv_prewrite` 几乎一致。关于写入阶段耗时的计算方式，请参考异步写入小节。

11.1.6.4 批量请求

下面是批量请求客户端的时间消耗图：

```
Diagram(
    NonTerminal("Get conn pool to the target store"),
    Choice(
        0,
        Sequence(
            Comment("Batch enabled"),
                NonTerminal("Push request to channel"),
                NonTerminal("Wait response"),
            ),
            Sequence(
            NonTerminal("Get conn from pool"),
            NonTerminal("Call RPC"),
            Choice(
                0,
                Comment("Unary call"),
                NonTerminal("Recv first"),
            ),
        ),
    ),
)
```

- 总体的发送请求耗时看作 tidb_tikvclient_request_seconds。
- RPC 客户端为每个存储维护各自的连接池（称为 ConnArray），每个连接池都包含带有一个发送批量请求 channel 的 BatchConn
- 绝大多数情况下，当存储是 TiKV 并且 batch 大小为正数时，批量请求开启。
- 批量请求 channel 的大小是tikv-client.max-batch-size 的值（默认值为 128）。请求入队的耗时看作 tidb_tikvclient_batch_wait_duration。
- 一共有 CmdBatchCop、CmdCopStream 和 CmdMPPConn 三种流式请求。流式请求会引入一个额外的 recv() 调用来获取流中的第一个响应。

tidb_tikvclient_request_seconds 大致使用如下方式计算（部分延迟不包含在内）：

```
tidb_tikvclient_request_seconds{type="?"} =
    tidb_tikvclient_batch_wait_duration +
    tidb_tikvclient_batch_send_latency +
    tikv_grpc_msg_duration_seconds{type="kv_?"} +
    tidb_tikvclient_rpc_net_latency_seconds{store="?"}
```

- tidb_tikvclient_batch_wait_duration 记录批量请求系统的等待耗时。
- tidb_tikvclient_batch_send_latency 记录批量请求系统的编码耗时。
- tikv_grpc_msg_duration_seconds{type="kv_?"} 是 TiKV 的处理耗时。
- tidb_tikvclient_rpc_net_latency_seconds 记录网络延迟。

11.1.6.5 TiKV 快照

下面是 TiKV 快照操作的时间消耗图：

```
Diagram(
    Choice(
        0,
        Comment("Local Read"),
        Sequence(
            NonTerminal("Propose Wait"),
            NonTerminal("Read index Read Wait"),
        ),
    ),
    NonTerminal("Fetch A Snapshot From KV Engine"),
)
```

一个 TiKV 快照的总体耗时可以从 tikv_storage_engine_async_request_duration_seconds{type="snapshot"} 指标查看，它的计算方式如下：

```
tikv_storage_engine_async_request_duration_seconds{type="snapshot"} =
    tikv_coprocessor_request_wait_seconds{type="snapshot"} =
    tikv_raftstore_request_wait_time_duration_secs +
    tikv_raftstore_commit_log_duration_seconds +
    get snapshot from rocksdb duration
```

当 leader lease 过期时，TiKV 会在从 RocksDB 获取快照之前提出读索引命令。`tikv_raftstore_request_wait_time_duration_secs` 和 `tikv_raftstore_commit_log_duration_seconds` 是提交读索引命令的耗时。

从 RocksDB 获取快照通常是一个快速操作，因此 `get snapshot from rocksdb duration` 的耗时可以被忽略。

11.1.6.6 异步写入

异步写入是 TiKV 通过回调将数据异步写入基于 Raft 的复制状态机 (Replicated State Machine) 的过程。

- 下面是异步 IO 未开启时，异步写入过程的时间消耗图：

```
Diagram(
    NonTerminal("Propose Wait"),
    NonTerminal("Process Command"),
    Choice(
        0,
        Sequence(
            NonTerminal("Wait Current Batch"),
            NonTerminal("Write to Log Engine"),
        ),
        Sequence(
            NonTerminal("RaftMsg Send Wait"),
            NonTerminal("Commit Log Wait"),
        ),
    ),
    NonTerminal("Apply Wait"),
    NonTerminal("Apply Log"),
)
```

- 下面是异步 IO 开启时，异步写入过程的时间消耗图：

```
Diagram(
    NonTerminal("Propose Wait"),
    NonTerminal("Process Command"),
    Choice(
        0,
        NonTerminal("Wait Until Persisted by Write Worker"),
        Sequence(
            NonTerminal("RaftMsg Send Wait"),
            NonTerminal("Commit Log Wait"),
        ),
    ),
    NonTerminal("Apply Wait"),
    NonTerminal("Apply Log"),
)
```

异步写入耗时的计算方式如下：

```
async write duration(async io disabled) =
    propose +
    async io disabled commit +
    tikv_raftstore_apply_wait_time_duration_secs +
    tikv_raftstore_apply_log_duration_seconds

async write duration(async io enabled) =
    propose +
    async io enabled commit +
    tikv_raftstore_apply_wait_time_duration_secs +
    tikv_raftstore_apply_log_duration_seconds
```

异步写入可以拆解为以下三个阶段：

- 提案阶段 (Propose)
- 提交阶段 (Commit)
- 应用阶段 (Apply)：对应上面公式中的 `tikv_raftstore_apply_wait_time_duration_secs + tikv_raftstore_apply_log_dura`
  ↪

提案阶段耗时的计算方式如下：

```
propose =
    propose wait duration +
    propose duration

propose wait duration =
    tikv_raftstore_store_wf_batch_wait_duration_seconds

propose duration =
    tikv_raftstore_store_wf_send_to_queue_duration_seconds -
    tikv_raftstore_store_wf_batch_wait_duration_seconds
```

Raft 过程以瀑布方式记录，因此，提案阶段的耗时是根据 `tikv_raftstore_store_wf_send_to_queue_duration_seconds`
↪ 和 `tikv_raftstore_store_wf_batch_wait_duration_seconds` 两个指标之间的差值计算的。

提交阶段耗时的计算方式如下：

```
async io disabled commit = max(
    persist log locally duration,
    replicate log duration
)

async io enabled commit = max(
    wait by write worker duration,
    replicate log duration
)
```

从 TiDB v5.3.0 开始，TiKV 支持通过 StoreWriter 线程池写入 Raft 日志，即异步 IO。异步 IO 会改变提交过程，仅在store-io-pool-size 数值大于 0 时启用。耗时 persist log locally duration 和 wait by write worker duration 的计算方式如下：

```
persist log locally duration =
    batch wait duration +
    write to raft db duration

batch wait duration =
    tikv_raftstore_store_wf_before_write_duration_seconds -
    tikv_raftstore_store_wf_send_to_queue_duration_seconds

write to raft db duration =
    tikv_raftstore_store_wf_write_end_duration_seconds -
    tikv_raftstore_store_wf_before_write_duration_seconds

wait by write worker duration =
    tikv_raftstore_store_wf_persist_duration_seconds -
    tikv_raftstore_store_wf_send_to_queue_duration_seconds
```

是否开启异步 IO 的区别在于本地持久化日志的耗时。使用异步 IO 可以直接从瀑布指标中计算本地持久化日志的耗时，忽略批处理等待耗时。

replicate log duration 代表 quorum 副本中日志持久化的耗时，其中包含 RPC 耗时和大多数日志持久化的耗时。replicate log duration 耗时的计算方式如下：

```
replicate log duration =
    raftmsg send wait duration +
    commit log wait duration

raftmsg send wait duration =
    tikv_raftstore_store_wf_send_proposal_duration_seconds -
    tikv_raftstore_store_wf_send_to_queue_duration_seconds

commit log wait duration =
    tikv_raftstore_store_wf_commit_log_duration -
    tikv_raftstore_store_wf_send_proposal_duration_seconds
```

11.1.6.6.1 Raft DB

下面是 Raft DB 的时间消耗图：

```
Diagram(
    NonTerminal("Wait for Writer Leader"),
    NonTerminal("Write and Sync Log"),
    NonTerminal("Apply Log to Memtable"),
)
```

```
write to raft db duration = raft db write duration
commit log wait duration >= raft db write duration

raft db write duration(raft engine enabled) =
    raft_engine_write_preprocess_duration_seconds +
    raft_engine_write_leader_duration_seconds +
    raft_engine_write_apply_duration_seconds

raft db write duration(raft engine disabled) =
    tikv_raftstore_store_perf_context_time_duration_secs{type="write_thread_wait"} +
    tikv_raftstore_store_perf_context_time_duration_secs{type="
        ↪ write_scheduling_flushes_compactions_time"} +
    tikv_raftstore_store_perf_context_time_duration_secs{type="write_wal_time"} +
    tikv_raftstore_store_perf_context_time_duration_secs{type="write_memtable_time"}
```

`commit log wait duration` 是 quorum 副本中最长的耗时，可能大于 `raft db write duration`。

从 TiDB v6.1.0 开始，TiKV 默认使用Raft Engine 作为日志存储引擎，这将改变写入日志的过程。

11.1.6.6.2 KV DB

下面是 KV DB 的时间消耗图：

```
Diagram(
    NonTerminal("Wait for Writer Leader"),
    NonTerminal("Preprocess"),
    Choice(
        0,
        Comment("No Need to Switch"),
        NonTerminal("Switch WAL or Memtable"),
    ),
    NonTerminal("Write and Sync WAL"),
    NonTerminal("Apply to Memtable"),
)
```

```
tikv_raftstore_apply_log_duration_seconds =
    tikv_raftstore_apply_perf_context_time_duration_secs{type="write_thread_wait"} +
    tikv_raftstore_apply_perf_context_time_duration_secs{type="
        ↪ write_scheduling_flushes_compactions_time"} +
    tikv_raftstore_apply_perf_context_time_duration_secs{type="write_wal_time"} +
    tikv_raftstore_apply_perf_context_time_duration_secs{type="write_memtable_time"}
```

在异步写入过程中，提交的日志需要应用到 KV DB 中，应用耗时可以根据 RocksDB performance context 进行计算。

11.1.6.7 诊断场景

前面的部分详细介绍了 SQL 查询过程中执行时间的细粒度指标。本小节主要介绍遇到慢读取或慢写入查询时常见的指标分析过程。所有指标均可在Performance Overview 面板的 Database Time 中查看。

11.1.6.7.1 慢读取查询

如果 SELECT 语句占 Database Time 的很大一部分，你可以认为 TiDB 在读查询时速度很慢。

慢查询的执行计划可以在 TiDB Dashboard 中的Top SQL 语句 区域查看。要分析慢读取查询的耗时，你可以根据前面的描述分析点查、批量点查和表扫描和索引扫描的耗时情况。

11.1.6.7.2 慢写入查询

在分析慢写入查询之前，你需要查看 `tikv_scheduler_latch_wait_duration_seconds_sum{type="acquire_pessimistic_lock"} by (instance)` 指标来确认冲突的原因：

- 如果这个指标在某些特定的 TiKV 实例中很高，则在热点区域可能会存在冲突。
- 如果这个指标在所有实例中都很高，则业务中可能存在冲突。

如果是业务中存在冲突，那么你可以分析加锁和提交阶段的耗时。

11.2 配置调优

11.2.0.1 操作系统性能参数调优

本文档仅用于描述如何优化 CentOS 7 的各个子系统。

> 注意：
>
> - CentOS 7 操作系统的默认配置适用于中等负载下运行的大多数服务。调整特定子系统的性能可能会对其他子系统产生负面影响。因此在调整系统之前，请备份所有用户数据和配置信息；
> - 请在测试环境下对所有修改做好充分测试后，再应用到生产环境中。

11.2.0.1.1 性能分析工具

系统调优需要根据系统性能分析的结果做指导，因此本文先列出常用的性能分析方法。

60 秒分析法

60 秒分析法由《性能之巅》的作者 Brendan Gregg 及其所在的 Netflix 性能工程团队公布。所用到的工具均可从发行版的官方源获取，通过分析以下清单中的输出，可定位大部分常见的性能问题。

- `uptime`
- `dmesg | tail`
- `vmstat 1`

- `mpstat -P ALL 1`
- `pidstat 1`
- `iostat -xz 1`
- `free -m`
- `sar -n DEV 1`
- `sar -n TCP,ETCP 1`
- `top`

具体用法可查询相应 man 手册。

perf

perf 是 Linux 内核提供的一个重要的性能分析工具，它涵盖硬件级别（CPU/PMU 和性能监视单元）功能和软件功能（软件计数器和跟踪点）。详细用法请参考 perf Examples。

BCC/bpftrace

CentOS 从 7.6 版本起，内核已实现对 BPF (Berkeley Packet Filter) 的支持，因此可根据上述清单的结果，选取适当的工具进行深入分析。相比 perf/ftrace，BPF 提供了可编程能力和更小的性能开销。相比 kprobe，BPF 提供了更高的安全性，更适合在生产环境上使用。关于 BCC 工具集的使用请参考 BPF Compiler Collection (BCC)。

11.2.0.1.2 性能调优

性能调优将根据内核子系统进行分类描述。

处理器 ——动态节能技术

cpufreq 是一个动态调整 CPU 频率的模块，可支持五种模式。为保证服务性能应选用 performance 模式，将 CPU 频率固定工作在其支持的最高运行频率上，不进行动态调节，操作命令为 `cpupower frequency-set --governor performance`。

处理器 ——中断亲和性

- 自动平衡：可通过 `irqbalance` 服务实现。
- 手动平衡：
  - 确定需要平衡中断的设备，从 CentOS 7.5 开始，系统会自动为某些设备及其驱动程序配置最佳的中断关联性。不能再手动配置其亲和性。目前已知的有使用 `be2iscsi` 驱动的设备，以及 NVMe 设置；
  - 对于其他设备，可查询其芯片手册，是否支持分发中断。
    * 若不支持，则该设备的所有中断会路由到同一个 CPU 上，无法对其进行修改。
    * 若支持，则计算 `smp_affinity` 掩码并设置对应的配置文件，具体请参考内核 IRQ-affinity 文档。

NUMA 绑核

为尽可能的避免跨 NUMA (Non-Uniform Memory Access) 访问内存，可以通过设置线程的 CPU 亲和性来实现 NUMA 绑核。对于普通程序，可使用 `numactl` 命令来绑定，具体用法请查询 man 手册。对于网卡中断，请参考下文网络章节。

内存 ——透明大页

对于数据库应用，不推荐使用 THP，因为数据库往往具有稀疏而不是连续的内存访问模式，且当高阶内存碎片化比较严重时，分配 THP 页面会出现较大的延迟。若开启针对 THP 的直接内存规整功能，也会出现系统 CPU 使用率激增的现象，因此建议关闭 THP。

```
echo never > /sys/kernel/mm/transparent_hugepage/enabled
echo never > /sys/kernel/mm/transparent_hugepage/defrag
```

内存 ——虚拟内存参数

- `dirty_ratio` 百分比值。当脏的 page cache 总量达到系统内存总量的这一百分比后，系统将开始使用 `pdflush` 操作将脏的 page cache 写入磁盘。默认值为 20%，通常不需调整。对于高性能 SSD，比如 NVMe 设备来说，降低其值有利于提高内存回收时的效率。
- `dirty_background_ratio` 百分比值。当脏的 page cache 总量达到系统内存总量的这一百分比后，系统开始在后台将脏的 page cache 写入磁盘。默认值为 10%，通常不需调整。对于高性能 SSD，比如 NVMe 设备来说，设置较低的值有利于提高内存回收时的效率。

存储及文件系统

内核 I/O 栈链路较长，包含了文件系统层、块设备层和驱动层。

I/O 调度器

I/O 调度程序确定 I/O 操作何时在存储设备上运行以及持续多长时间。也称为 I/O 升降机。对于 SSD 设备，宜设置为 noop。

```
echo noop > /sys/block/${SSD_DEV_NAME}/queue/scheduler
```

格式化参数 ——块大小

块 (block) 是文件系统的工作单元。块大小决定了单个块中可以存储多少数据，因此决定了一次写入或读取的最小数据量。

默认块大小适用于大多数使用情况。但是，如果块大小（或多个块的大小）与通常一次读取或写入的数据量相同或稍大，则文件系统将性能更好，数据存储效率更高。小文件仍将使用整个块。文件可以分布在多个块中，但这会增加运行时开销。

使用 `mkfs` 命令格式化设备时，将块大小指定为文件系统选项的一部分。指定块大小的参数随文件系统的不同而不同。有关详细信息，请查询对应文件系统的 `mkfs` 手册页，比如 `man mkfs.ext4`。

挂载参数

`noatime` 读取文件时，将禁用对元数据的更新。它还启用了 `nodiratime` 行为，该行为会在读取目录时禁用对元数据的更新。

网络

网络子系统由具有敏感连接的许多不同部分组成。因此，CentOS 7 网络子系统旨在为大多数工作负载提供最佳性能，并自动优化其性能。因此，通常无需手动调整网络性能。

网络问题通常由硬件或相关设施出现问题导致的，因此在调优协议栈前，请先排除硬件问题。

尽管网络堆栈在很大程度上是自我优化的。但是在网络数据包处理过程中，以下方面可能会成为瓶颈并降低性能：

- 网卡硬件缓存：正确观察硬件层面的丢包方法是使用 `ethtool -S ${NIC_DEV_NAME}` 命令观察 `drops` 字段。当出现丢包现象时，主要考虑是硬/软中断的处理速度跟不上网卡接收速度。若接收缓存小于最大

限制时，也可尝试增加 RX 缓存来防止丢包。查询命令为：`ethtool -g ${NIC_DEV_NAME}`，修改命令为 `ethtool -G ${NIC_DEV_NAME}`。

- 硬中断：若网卡支持 Receive-Side Scaling（RSS 也称为多网卡接收）功能，则观察 `/proc/interrupts` 网卡中断，如果出现了中断不均衡的情况，请参考处理器动态节能技术，处理器调优，NUMA 绑核。若不支持 RSS 或 RSS 数量远小于物理 CPU 核数，则可配置 Receive Packet Steering（RPS，可以看作 RSS 的软件实现），及 RPS 的扩展 Receive Flow Steering (RFS)。具体设置请参考内核文档。
- 软中断：观察 `/proc/net/softnet_stat` 监控。如果除第三列的其他列的数值在增长，则应适度调大 `net.core.netdev_budget` 或 `net.core.dev_weight` 值，使 `softirq` 可以获得更多的 CPU 时间。除此之外，也需要检查 CPU 使用情况，确定哪些任务在频繁占用 CPU，能否优化。
- 应用的套接字接收队列：监控 `ss -nmp` 的 `Recv-q` 列，若队列已满，则应考虑增大应用程序套接字的缓存大小或使用自动调整缓存的方式。除此之外，也要考虑能否优化应用层的架构，降低读取套接字的间隔。
- 以太网流控：若网卡和交换机支持流控功能，可通过使能此功能，给内核一些时间来处理网卡队列中的数据，来规避网卡缓存溢出的问题。对于网卡测，可通过 `ethtool -a ${NIC_DEV_NAME}` 命令检查是否支持/使能，并通过 `ethtool -A ${NIC_DEV_NAME}` 命令开启。对于交换机，请查询其手册。
- 中断合并：过于频繁的硬件中断会降低系统性能，而过晚的硬件中断会导致丢包。对于较新的网卡支持中断合并功能，并允许驱动自动调节硬件中断数。可通过 `ethtool -c ${NIC_DEV_NAME}` 命令检查，`ethtool -C ${NIC_DEV_NAME}` 命令开启。自适应模式使网卡可以自动调节中断合并。在自适应模式下，驱动程序将检查流量模式和内核接收模式，并实时评估合并设置，以防止数据包丢失。不同品牌的网卡具有不同的功能和默认配置，具体请参考网卡手册。
- 适配器队列：在协议栈处理之前，内核利用此队列缓存网卡接收的数据，每个 CPU 都有各自的 backlog 队列。此队列可缓存的最大 packets 数量为 `netdev_max_backlog`。观察 `/proc/net/softnet_stat` 第二列，当某行的第二列持续增加，则意味着 CPU [row-1] 队列已满，数据包被丢失，可通过持续加倍 `net.core.netdev_max_backlog` 值来解决。
- 发送队列：发送队列长度值确定在发送之前可以排队的数据包数量。默认值是 1000，对于 10 Gbps 足够。但若从 `ip -s link` 的输出中观察到 `TX errors` 值时，可尝试加倍该数据包数量：`ip link set dev ${NIC_DEV_NAME} txqueuelen 2000`。
- 驱动：网卡驱动通常也会提供调优参数，请查询设备硬件手册及其驱动文档。

11.2.0.2 TiDB 内存控制文档

目前 TiDB 已经能够做到追踪单条 SQL 查询过程中的内存使用情况，当内存使用超过一定阈值后也能采取一些操作来预防 OOM 或者排查 OOM 原因。你可以使用系统变量`tidb_mem_oom_action` 来控制查询超过内存限制后所采取的操作：

- 如果变量值为 LOG，那么当一条 SQL 的内存使用超过一定阈值（由 session 变量 `tidb_mem_quota_query` 控制）后，这条 SQL 会继续执行，但 TiDB 会在 log 文件中打印一条 LOG。
- 如果变量值为 CANCEL，那么当一条 SQL 的内存使用超过一定阈值后，TiDB 会立即中断这条 SQL 的执行，并给客户端返回一个错误，错误信息中会详细写明在这条 SQL 执行过程中占用内存的各个物理执行算子的内存使用情况。

11.2.0.2.1 如何配置一条 SQL 执行过程中的内存使用阈值

使用系统变量`tidb_mem_quota_query` 来配置一条 SQL 执行过程中的内存使用阈值，单位为字节。例如：

配置整条 SQL 的内存使用阈值为 8GB：

```
SET tidb_mem_quota_query = 8 << 30;
```

配置整条 SQL 的内存使用阈值为 8MB：

```
SET tidb_mem_quota_query = 8 << 20;
```

配置整条 SQL 的内存使用阈值为 8KB：

```
SET tidb_mem_quota_query = 8 << 10;
```

11.2.0.2.2 如何配置 tidb-server 实例使用内存的阈值

自 v6.5.0 版本起，可以通过系统变量tidb_server_memory_limit 设置 tidb-server 实例的内存使用阈值。

例如，配置 tidb-server 实例的内存使用总量，将其设置成为 32 GB：

```
SET GLOBAL tidb_server_memory_limit = "32GB";
```

设置该变量后，当 tidb-server 实例的内存用量达到 32 GB 时，TiDB 会依次终止正在执行的 SQL 操作中内存用量最大的 SQL 操作，直至 tidb-server 实例内存使用下降到 32 GB 以下。被强制终止的 SQL 操作会向客户端返回报错信息 `Out Of Memory Quota!`。

当前 `tidb_server_memory_limit` 所设的内存限制不终止以下 SQL 操作：

- DDL 操作
- 包含窗口函数和公共表表达式的 SQL 操作

> 警告：
>
> - TiDB 在启动过程中不保证tidb_server_memory_limit 限制生效。如果操作系统的空闲内存不足，TiDB 仍有可能出现 OOM。你需要确保 TiDB 实例有足够的可用内存。
> - 在内存控制过程中，TiDB 的整体内存使用量可能会略微超过 `tidb_server_memory_limit` 的限制。
> - `server-memory-quota` 配置项自 v6.5.0 起被废弃。为了保证兼容性，在升级到 v6.5.0 或更高版本的集群后，`tidb_server_memory_limit` 会继承配置项 `server-memory-quota` 的值。如果集群在升级至 v6.5.0 或更高版本前没有配置 `server-memory-quota`，`tidb_server_memory_limit` 会使用默认值，即 80%。

在 tidb-server 实例内存用量到达总内存的一定比例时（比例由系统变量tidb_server_memory_limit_gc_trigger 控制），tidb-server 会尝试主动触发一次 Golang GC 以缓解内存压力。为了避免实例内存在阈值上下范围不断波动导致频繁 GC 进而带来的性能问题，该 GC 方式 1 分钟最多只会触发 1 次。

11.2.0.2.3 使用 INFORMATION_SCHEMA 系统表查看当前 tidb-server 的内存用量

要查看当前实例或集群的内存使用情况，你可以查询系统表INFORMATION_SCHEMA.(CLUSTER_)MEMORY_USAGE。

要查看本实例或集群中内存相关的操作和执行依据，可以查询系统表INFORMATION_SCHEMA.(CLUSTER_)MEMORY_USAGE_OPS_HISTORY。对于每个实例，该表保留最近 50 条记录。

11.2.0.2.4 tidb-server 内存占用过高时的报警

当 tidb-server 实例的内存使用量超过内存阈值（默认为总内存量的 70%）且满足以下任一条件时，TiDB 将记录相关状态文件，并打印报警日志。

- 第一次内存使用量超过内存阈值。
- 内存使用量超过内存阈值，且距离上一次报警超过 60 秒。
- 内存使用量超过内存阈值，且（本次内存使用量 - 上次报警时内存使用量）/ 总内存量 > 10%。

你可以通过系统变量tidb_memory_usage_alarm_ratio 修改触发该报警的内存使用比率，从而控制内存报警的阈值。

当触发 tidb-server 内存占用过高的报警时，TiDB 的报警行为如下：

- TiDB 将以下信息记录到 TiDB 日志文件filename 所在目录中。
  - 当前正在执行的所有 SQL 语句中内存使用最高的 10 条语句和运行时间最长的 10 条语句的相关信息
  - goroutine 栈信息
  - 堆内存使用状态
- TiDB 将输出一条包含关键字 `tidb-server has the risk of OOM` 以及以下内存相关系统变量的日志。
  - tidb_mem_oom_action
  - tidb_mem_quota_query
  - tidb_server_memory_limit
  - tidb_analyze_version
  - tidb_enable_rate_limit_action

为避免报警时产生的状态文件累积过多，目前 TiDB 默认只保留最近 5 次报警时所生成的状态文件。你可以通过配置系统变量tidb_memory_usage_alarm_keep_record_num 调整该次数。

下例通过构造一个占用大量内存的 SQL 语句触发报警，对该报警功能进行演示：

1. 配置报警比例为 0.85：

```
SET GLOBAL tidb_memory_usage_alarm_ratio = 0.85;
```

2. 创建单表 `CREATE TABLE t(a int);` 并插入 1000 行数据。
3. 执行 `select * from t t1 join t t2 join t t3 order by t1.a`。该 SQL 语句会输出 1000000000 条记录，占用巨大的内存，进而触发报警。

4. 检查 tidb.log 文件，其中会记录系统总内存、系统当前内存使用量、tidb-server 实例的内存使用量以及状态文件所在目录。

```
[2022/10/11 16:39:02.281 +08:00] [WARN] [memoryusagealarm.go:212] ["tidb-server has the risk
    ↪  of OOM because of memory usage exceeds alarm ratio. Running SQLs and heap profile
    ↪ will be recorded in record path"] ["is tidb_server_memory_limit set"=false] ["system
    ↪  memory total"=33682427904] ["system memory usage"=22120655360] ["tidb-server memory
    ↪  usage"=21468556992] [memory-usage-alarm-ratio=0.85] ["record path"=/tiup/deploy/
    ↪ tidb-4000/log/oom_record]
```

以上 Log 字段的含义如下：

- is tidb_server_memory_limit set：表示系统变量tidb_server_memory_limit 是否被设置
- system memory total：表示当前系统的总内存
- system memory usage：表示当前系统的内存使用量
- tidb-server memory usage：表示 tidb-server 实例的内存使用量
- memory-usage-alarm-ratio：表示系统变量tidb_memory_usage_alarm_ratio 的值
- record path：表示状态文件存放的目录

5. 通过访问状态文件所在目录（该示例中的目录为 /tiup/deploy/tidb-4000/log/oom_record），可以看到标记了记录时间的 record 目录（例：record2022-10-09T17:18:38+08:00），其中包括 goroutinue、heap、running_sql 3 个文件，文件以记录状态文件的时间为后缀。这 3 个文件分别用来记录报警时的 goroutine 栈信息，堆内存使用状态，及正在运行的 SQL 信息。其中 running_sql 文件内容请参考expensive-queries。

11.2.0.2.5 tidb-server 其它内存控制策略

流量控制

- TiDB 支持对读数据算子的动态内存控制功能。读数据的算子默认启用tidb_distsql_scan_concurrency 所允许的最大线程数来读取数据。当单条 SQL 语句的内存使用每超过tidb_mem_quota_query 一次，读数据的算子就会停止一个线程。
- 流控行为由参数tidb_enable_rate_limit_action 控制。
- 当流控被触发时，会在日志中打印一条包含关键字 memory exceeds quota, destroy one token now 的日志。

数据落盘

TiDB 支持对执行算子的数据落盘功能。当 SQL 的内存使用超过 Memory Quota 时，tidb-server 可以通过落盘执行算子的中间数据，缓解内存压力。支持落盘的算子有：Sort、 MergeJoin、 HashJoin、 HashAgg。

- 落盘行为由参数tidb_mem_quota_query、 tidb_enable_tmp_storage_on_oom、 tmp-storage-path、 tmp-
  ↪ storage-quota 共同控制。
- 当落盘被触发时，TiDB 会在日志中打印一条包含关键字 memory exceeds quota, spill to disk now 或 memory exceeds quota, set aggregate mode to spill-mode 的日志。
- Sort、 MergeJoin、 HashJoin 落盘是从 v4.0.0 版本开始引入的，HashAgg 落盘是从 v5.2.0 版本开始引入的。
- 当包含 Sort、MergeJoin 或 HashJoin 的 SQL 语句引起内存 OOM 时，TiDB 默认会触发落盘。当包含 HashAgg 算子的 SQL 语句引起内存 OOM 时，TiDB 默认不触发落盘，请设置系统变量 tidb_executor_concurrency = 1 来触发 HashAgg 落盘功能。

注意：

- HashAgg 落盘功能目前不支持 distinct 聚合函数。使用 distinct 函数且内存占用过大时，无法进行落盘。

本示例通过构造一个占用大量内存的 SQL 语句，对 HashAgg 落盘功能进行演示：

1. 将 SQL 语句的 Memory Quota 配置为 1GB（默认 1GB）：

```
SET tidb_mem_quota_query = 1 << 30;
```

2. 创建单表 `CREATE TABLE t(a int);` 并插入 256 行不同的数据。

3. 尝试执行以下 SQL 语句：

```
[tidb]> explain analyze select /*+ HASH_AGG() */ count(*) from t t1 join t t2 join t t3
    ↪ group by t1.a, t2.a, t3.a;
```

该 SQL 语句占用大量内存，返回 Out of Memory Quota 错误。

```
ERROR 1105 (HY000): Out Of Memory Quota![conn_id=3]
```

4. 设置系统变量 `tidb_executor_concurrency` 将执行器的并发度调整为 1。在此配置下，内存不足时 HashAgg 会自动尝试触发落盘。

```
SET tidb_executor_concurrency = 1;
```

5. 执行相同的 SQL 语句，不再返回错误，可以执行成功。从详细的执行计划可以看出，HashAgg 使用了 600MB 的硬盘空间。

```
[tidb]> explain analyze select /*+ HASH_AGG() */ count(*) from t t1 join t t2 join t t3
    ↪ group by t1.a, t2.a, t3.a;
```

```
+--
    ↪ -------------------------------+-------------+----------+-----------+---------------+---------------
    ↪
| id                              | estRows     | actRows  | task      | access object |
    ↪ execution info
    ↪
    ↪ | operator info                                          | memory    | disk
    ↪      |
+--
    ↪ -------------------------------+-------------+----------+-----------+---------------+---------------
    ↪
```

```
| HashAgg_11                      | 204.80      | 16777216 | root      |               |
    ↪ time:1m37.4s, loops:16385
    ↪
    ↪ | group by:test.t.a, test.t.a, test.t.a, funcs:count(1)->Column#7 | 1.13 GB   |
    ↪ 600.0 MB |
| └─HashJoin_12                   | 16777216.00 | 16777216 | root      |               |
    ↪ time:21.5s, loops:16385, build_hash_table:{total:267.2µs, fetch:228.9µs, build:38.2
    ↪ µs}, probe:{concurrency:1, total:35s, max:35s, probe:35s, fetch:962.2µs}         |
    ↪ CARTESIAN inner join                                          | 8.23 KB   | 4 KB
    ↪     |
|   ├─TableReader_21(Build)       | 256.00      | 256      | root      |               |
    ↪ time:87.2µs, loops:2, cop_task: {num: 1, max: 150µs, proc_keys: 0, rpc_num: 1,
    ↪ rpc_time: 145.1µs, copr_cache_hit_ratio: 0.00}
    ↪                                       | data:TableFullScan_20
    ↪                                           | 885 Bytes | N/A      |
|   │ └─TableFullScan_20          | 256.00      | 256      | cop[tikv] | table:t3      |
    ↪ tikv_task:{time:23.2µs, loops:256}
    ↪
    ↪ | keep order:false, stats:pseudo                                  | N/A       | N/A
    ↪     |
|   └─HashJoin_14(Probe)          | 65536.00    | 65536    | root      |               |
    ↪ time:728.1µs, loops:65, build_hash_table:{total:307.5µs, fetch:277.6µs, build:29.9µs
    ↪ }, probe:{concurrency:1, total:34.3s, max:34.3s, probe:34.3s, fetch:278µs}      |
    ↪ CARTESIAN inner join                                          | 8.23 KB   | 4 KB
    ↪     |
|     ├─TableReader_19(Build)     | 256.00      | 256      | root      |               |
    ↪ time:126.2µs, loops:2, cop_task: {num: 1, max: 308.4µs, proc_keys: 0, rpc_num: 1,
    ↪ rpc_time: 295.3µs, copr_cache_hit_ratio: 0.00}                                  |
    ↪ data:TableFullScan_18                                         | 885 Bytes | N/A
    ↪     |
|     │ └─TableFullScan_18        | 256.00      | 256      | cop[tikv] | table:t2      |
    ↪ tikv_task:{time:79.2µs, loops:256}
    ↪
    ↪ | keep order:false, stats:pseudo                                  | N/A       | N/A
    ↪     |
|     └─TableReader_17(Probe)     | 256.00      | 256      | root      |               |
    ↪ time:211.1µs, loops:2, cop_task: {num: 1, max: 295.5µs, proc_keys: 0, rpc_num: 1,
    ↪ rpc_time: 279.7µs, copr_cache_hit_ratio: 0.00}                                  |
    ↪ data:TableFullScan_16                                         | 885 Bytes | N/A
    ↪     |
|       └─TableFullScan_16        | 256.00      | 256      | cop[tikv] | table:t1      |
    ↪ tikv_task:{time:71.4µs, loops:256}
    ↪
    ↪ | keep order:false, stats:pseudo                                  | N/A       | N/A
    ↪     |
```

```
+--
    ↪ ------------------------------+-------------+----------+-----------+---------------+--------------
    ↪
9 rows in set (1 min 37.428 sec)
```

11.2.0.2.6 其它

设置环境变量 GOMEMLIMIT 缓解 OOM 问题

Golang 自 Go 1.19 版本开始引入 GOMEMLIMIT 环境变量，该变量用来设置触发 Go GC 的内存上限。

对于 v6.1.3 <= TiDB < v6.5.0 的版本，你可以通过手动设置 Go GOMEMLIMIT 环境变量的方式来缓解一类 OOM 问题。该类 OOM 问题具有一个典型特征：观察 Grafana 监控，OOM 前的时刻，TiDB-Runtime > Memory Usage 面板中 estimate-inuse 立柱部分在整个立柱中仅仅占一半。如下图所示：

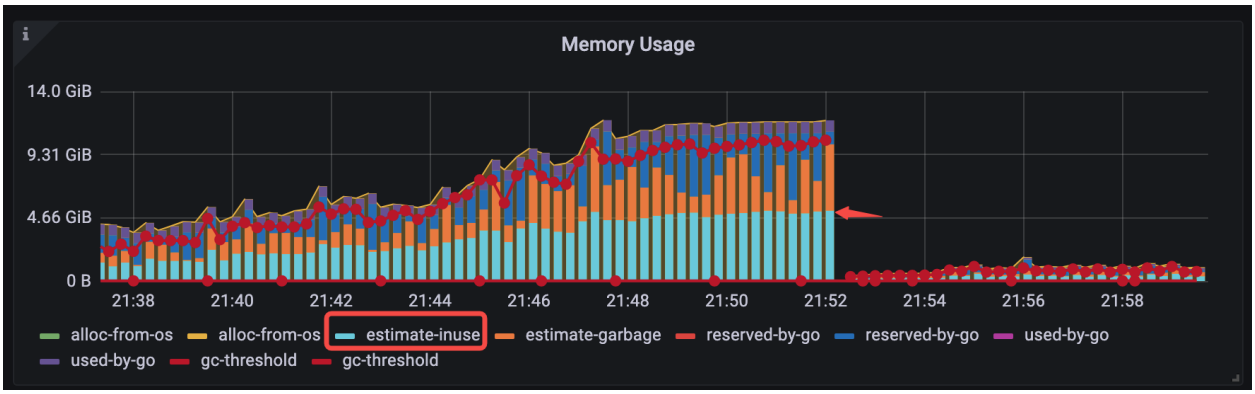


图 159: normal OOM case example

为了验证 GOMEMLIMIT 在该类场景下的效果，以下通过一个对比实验进行说明：

- 在 TiDB v6.1.2 下，模拟负载在持续运行几分钟后，TiDB server 会发生 OOM（系统内存约 48 GiB）：



图 160: v6.1.2 workload oom

- 在 TiDB v6.1.3 下，设置 GOMEMLIMIT 为 40000 MiB，模拟负载长期稳定运行、TiDB server 未发生 OOM 且进程最高内存用量稳定在 40.8 GiB 左右：

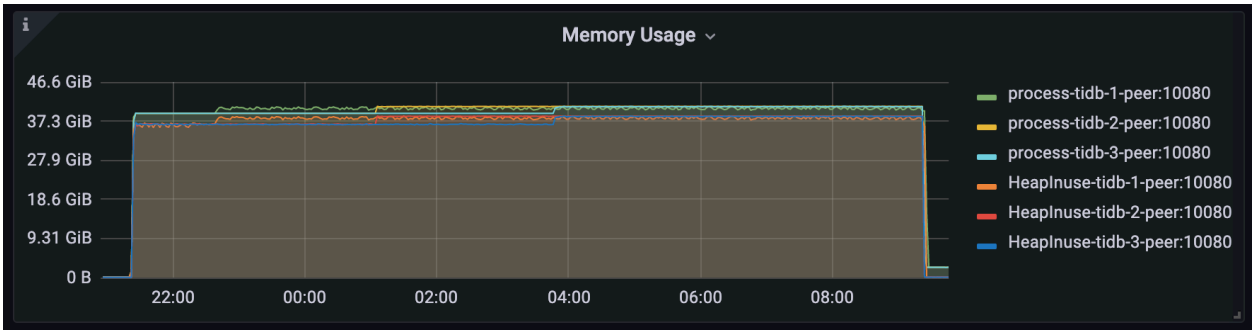


图 161: v6.1.3 workload no oom with GOMEMLIMIT

11.2.0.3 TiKV 线程池性能调优

本文主要介绍 TiKV 线程池性能调优的主要手段，以及 TiKV 内部线程池的主要用途。

11.2.0.3.1 线程池介绍

在 TiKV 中，线程池主要由 gRPC、Scheduler、UnifyReadPool、Raftstore、StoreWriter、Apply、RocksDB 以及其它一些占用 CPU 不多的定时任务与检测组件组成，这里主要介绍几个占用 CPU 比较多且会对用户读写请求的性能产生影响的线程池。

- gRPC 线程池：负责处理所有网络请求，它会把不同任务类型的请求转发给不同的线程池。
- Scheduler 线程池：负责检测写事务冲突，把事务的两阶段提交、悲观锁上锁、事务回滚等请求转化为 key-value 对数组，然后交给 Raftstore 线程进行 Raft 日志复制。
- Raftstore 线程池：
  - 处理所有的 Raft 消息以及添加新日志的提议 (Propose)。
  - 处理 Raft 日志。如果`store-io-pool-size`配置项的值为 0，Raftstore 线程将日志写入到磁盘；如果该值不为 0，Raftstore 线程将日志发送给 StoreWriter 线程处理。
  - 当日志在多数副本中达成一致后，Raftstore 线程把该日志发送给 Apply 线程处理。
- StoreWriter 线程池：负责将所有 Raft 日志写入到磁盘，再把结果返回到 Raftstore 线程。
- Apply 线程池：当收到从 Raftstore 线程池发来的已提交日志后，负责将其解析为 key-value 请求，然后写入 RocksDB 并且调用回调函数通知 gRPC 线程池中的写请求完成，返回结果给客户端。
- RocksDB 线程池：RocksDB 进行 Compact 和 Flush 任务的线程池，关于 RocksDB 的架构与 Compact 操作请参考 RocksDB: A Persistent Key-Value Store for Flash and RAM Storage。
- UnifyReadPool 线程池：由 Coprocessor 线程池与 Storage Read Pool 合并而来，所有的读取请求包括 kv get、kv batch get、raw kv get、coprocessor 等都会在这个线程池中执行。

11.2.0.3.2 TiKV 的只读请求

TiKV 的读取请求分为两类：

- 一类是指定查询某一行或者某几行的简单查询，这类查询会运行在 Storage Read Pool 中。
- 另一类是复杂的聚合计算、范围查询，这类请求会运行在 Coprocessor Read Pool 中。

从 TiKV 5.0 版本起，默认所有的读取请求都通过统一的线程池进行查询。如果是从 TiKV 4.0 升级上来的 TiKV 集群且升级前未打开 `readpool.storage` 的 `use-unified-pool` 配置，则升级后所有的读取请求仍然继续使用独立的线程池进行查询，可以将 `readpool.storage.use-unified-pool` 设置为 `true` 使所有的读取请求通过统一的线程池进行查询。

11.2.0.3.3 TiKV 线程池调优

- gRPC 线程池的大小默认配置 (`server.grpc-concurrency`) 是 5。由于 gRPC 线程池几乎不会有多少计算开销，它主要负责网络 IO、反序列化请求，因此该配置通常不需要调整。
  - 如果部署的机器 CPU 核数特别少（小于等于 8），可以考虑将该配置 (`server.grpc-concurrency`) 设置为 2。
  - 如果机器配置很高，并且 TiKV 承担了非常大量的读写请求，观察到 Grafana 上的监控 Thread CPU 的 gRPC poll CPU 的数值超过了 server.grpc-concurrency 大小的 80%，那么可以考虑适当调大 `server.grpc-concurrency` 以控制该线程池使用率在 80% 以下（即 Grafana 上的指标低于 80% * `server.grpc-concurrency` 的值）。
- Scheduler 线程池的大小配置 (`storage.scheduler-worker-pool-size`) 在 TiKV 检测到机器 CPU 核数大于等于 16 时默认为 8，小于 16 时默认为 4。它主要用于将复杂的事务请求转化为简单的 key-value 读写。但是 scheduler 线程池本身不进行任何写操作。
  - 如果检测到有事务冲突，那么它会提前返回冲突结果给客户端。
  - 如果未检测到事务冲突，那么它会把需要写入的 key-value 合并成一条 Raft 日志交给 Raftstore 线程进行 Raft 日志复制。

  通常来说为了避免过多的线程切换，最好确保 scheduler 线程池的利用率保持在 50% ~ 75% 之间。（如果线程池大小为 8 的话，那么 Grafana 上的 TiKV-Details.Thread CPU.scheduler worker CPU 应当在 400% ~ 600% 之间较为合理）
- Raftstore 线程池是 TiKV 中最复杂的一个线程池，默认大小 (由 `raftstore.store-pool-size` 控制) 为 2。StoreWriter 线程池的默认大小 (由 `raftstore.store-io-pool-size` 控制) 为 0。
  - 当 StoreWriter 线程池大小为 0 时，所有的写请求都会由 Raftstore 线程以 fsync 的方式写入 RocksDB。此时建议采取如下调优操作：
    * 将 Raftstore 线程的整体 CPU 使用率控制在 60% 以下。当把 Raftstore 线程数设为默认值 2 时，将 Grafana 监控上 TiKV-Details、Thread CPU、Raft store CPU 面版上的数值控制在 120% 以内。由于存在 I/O 请求，理论上 Raftstore 线程的 CPU 使用率总是低于 100%。
    * 不建议为了提升写性能而盲目增大 Raftstore 线程池大小，这样可能会适得其反，增加磁盘负担，导致性能变差。
  - 当 StoreWriter 线程池大小不为 0 时，所有写请求都由 StoreWriter 线程以 fsync 的方式写入 RocksDB。此时建议采取如下调优操作：

    * 仅在整体 CPU 资源比较充裕的情况下启用 StoreWriter 线程池，并将 StoreWriter 线程和 Raftstore 线程的 CPU 使用率控制在 80% 以下。
      与写请求在 Raftstore 线程完成的情况相比，理论上 StoreWriter 线程处理写请求能够显著地降低写延迟和读的尾延迟。然而，写入速度变得更快意味着 Raft 日志也变得更多，从而导致 Raftstore 线程、Apply 线程和 gRPC 线程的 CPU 开销增多。在这种情况下，CPU 资源不足可能会抵消优化效果，反而还可能比原来的写速度更慢，因此若是 CPU 资源不充裕则不建议开启 StoreWriter 线程。由于 Raftstore 线程把绝大部分的 I/O 请求交给 StoreWriter，因此 Raftstore 线程的 CPU 使用率控制在 80% 以下即可。
    * 大多数情况下将 StoreWriter 线程池的大小设为 1 或 2 即可。这是因为 StoreWriter 线程池的大小会影响 Raft 日志数量，所以该值不宜过大。如果 CPU 使用率高于 80%，可以考虑再增加其大小。
    * 注意 Raft 日志增多对其他线程池 CPU 开销的影响，必要的时候需要相应地增加 Raftstore 线程、Apply 线程和 gRPC 线程的数量。

- UnifyReadPool 负责处理所有的读取请求。默认配置 (`readpool.unified.max-thread-count`) 大小为机器 CPU 数的 80%（如机器为 16 核，则默认线程池大小为 12）。

  通常建议根据业务负载特性调整其 CPU 使用率在线程池大小的 60% ~ 90% 之间（如果用户 Grafana 上 TiKV-Details.Thread CPU.Unified read pool CPU 的峰值不超过 800%，那么建议用户将 `readpool.unified.max-thread-count` 设置为 10，过多的线程数会造成更频繁的线程切换，并且抢占其他线程池的资源）。

  TiKV 从 v6.3.0 开始支持根据统一读线程池 (UnifyReadPool) 的 CPU 利用率自动动态调整线程池的线程数量，可以通过配置`readpool.unified.auto-adjust-pool-size` 开启此功能。对于重读并且峰值 CPU 利用率超过 80% 的集群，建议开启此配置。

- RocksDB 线程池是 RocksDB 进行 Compact 和 Flush 任务的线程池，通常不需要配置。

  - 如果机器 CPU 核数较少，可将 `rocksdb.max-background-jobs` 与 `raftdb.max-background-jobs` 同时设置为 4。
  - 如果遇到了 Write Stall，可查看 Grafana 监控上 RocksDB-kv 中的 Write Stall Reason 有哪些指标不为 0。
    * 如果是由 pending compaction bytes 相关原因引起的，可将 `rocksdb.max-sub-compactions` 设置为 2 或者 3（该配置表示单次 compaction job 允许使用的子线程数量，TiKV 4.0 版本默认值为 3，3.0 版本默认值为 1）。
    * 如果原因是 memtable count 相关，建议调大所有列的 `max-write-buffer-number`（默认为 5）。
    * 如果原因是 level0 file limit 相关，建议调大如下参数为 64 或者更高：

      ```
      rocksdb.defaultcf.level0-slowdown-writes-trigger
      rocksdb.writecf.level0-slowdown-writes-trigger
      rocksdb.lockcf.level0-slowdown-writes-trigger
      rocksdb.defaultcf.level0-stop-writes-trigger
      rocksdb.writecf.level0-stop-writes-trigger
      rocksdb.lockcf.level0-stop-writes-trigger
      ```

11.2.0.4 TiKV 内存参数性能调优

本文档用于描述如何根据机器配置情况来调整 TiKV 的参数，使 TiKV 的性能达到最优。你可以在 etc/config-template.toml 找到配置文件模版，参考使用 TiUP 修改配置参数进行操作，部分配置项可以通过在线修改 TiKV 配置方式在线更新。具体配置项的含义可参考TiKV 配置文件描述。

TiKV 最底层使用的是 RocksDB 做为持久化存储，所以 TiKV 的很多性能相关的参数都是与 RocksDB 相关的。TiKV 使用了两个 RocksDB 实例，默认 RocksDB 实例存储 KV 数据，Raft RocksDB 实例（简称 RaftDB）存储 Raft 数据。

TiKV 使用了 RocksDB 的 Column Families (CF) 特性。

- 默认 RocksDB 实例将 KV 数据存储在内部的 default、write 和 lock 3 个 CF 内。
  - default CF 存储的是真正的数据，与其对应的参数位于 [rocksdb.defaultcf] 项中；
  - write CF 存储的是数据的版本信息 (MVCC) 以及索引相关的数据，相关的参数位于 [rocksdb.writecf] 项中；
  - lock CF 存储的是锁信息，系统使用默认参数。
- Raft RocksDB 实例存储 Raft log。
  - default CF 主要存储的是 Raft log，与其对应的参数位于 [raftdb.defaultcf] 项中。

所有的 CF 默认共同使用一个 block cache 实例。通过在 [storage.block-cache] 下设置 capacity 参数，你可以配置该 block cache 的大小。block cache 越大，能够缓存的热点数据越多，读取数据越容易，同时占用的系统内存也越多。

> **注意：**
>
> 在 TiKV 3.0 之前的版本中，不支持使用 shared block cache，需要为每个 CF 单独配置 block cache。

每个 CF 有各自的 write-buffer，大小通过 write-buffer-size 控制。

11.2.0.4.1 参数说明

```
#### 日志级别，可选值为：trace，debug，warn，error，info，off
log-level = "info"

[server]
#### 监听地址
#### addr = "127.0.0.1:20160"

#### gRPC 线程池大小
#### grpc-concurrency = 4
#### TiKV 每个实例之间的 gRPC 连接数
#### grpc-raft-conn-num = 10

#### TiDB 过来的大部分读请求都会发送到 TiKV 的 Coprocessor 进行处理，该参数用于设置
#### coprocessor 线程的个数，如果业务是读请求比较多，增加 coprocessor 的线程数，但应比系统的
#### CPU 核数小。例如：TiKV 所在的机器有 32 core，在重读的场景下甚至可以将该参数设置为 30。在没有
#### 设置该参数的情况下，TiKV 会自动将该值设置为 CPU 总核数乘以 0.8。
```

```
#### end-point-concurrency = 8

#### 可以给 TiKV 实例打标签，用于副本的调度
#### labels = {zone = "cn-east-1", host = "118", disk = "ssd"}

[storage]
#### 数据目录
#### data-dir = "/tmp/tikv/store"

#### 通常情况下使用默认值就可以了。在导数据的情况下建议将该参数设置为 1024000。
#### scheduler-concurrency = 102400
#### 该参数控制写入线程的个数，当写入操作比较频繁的时候，需要把该参数调大。使用 top -H -p tikv-
    ↪ pid
#### 发现名称为 sched-worker-pool 的线程都特别忙，这个时候就需要将 scheduler-worker-pool-size
#### 参数调大，增加写线程的个数。
#### scheduler-worker-pool-size = 4

[storage.block-cache]
##### 是否为 RocksDB 的所有 CF 都创建一个 `shared block cache`。
##### ## RocksDB 使用 block cache 来缓存未压缩的数据块。较大的 block cache 可以加快读取速度。
##### 推荐开启 `shared block cache` 参数。这样只需要设置全部缓存大小，使配置过程更加方便。
##### 在大多数情况下，可以通过 LRU 算法在各 CF 间自动平衡缓存用量。
##### ## `storage.block-cache` 会话中的其余配置仅在开启 `shared block cache` 时起作用。
##### 从 v6.6.0 开始，该选项永远开启且无法关闭。
#### shared = true
##### `shared block cache` 的大小。正常情况下应设置为系统全部内存的 30%-50%。
##### 如果未设置该参数，则由以下字段或其默认值的总和决定。
##### ##   * rocksdb.defaultcf.block-cache-size 或系统全部内存的 25%
#####   * rocksdb.writecf.block-cache-size 或系统全部内存的 15%
#####   * rocksdb.lockcf.block-cache-size 或系统全部内存的 2%
#####   * raftdb.defaultcf.block-cache-size 或系统全部内存的 2%
##### ## 要在单个物理机上部署多个 TiKV 节点，需要显式配置该参数。
##### 否则，TiKV 中可能会出现 OOM 错误。
#### capacity = "1GB"

[pd]
#### pd 的地址
#### endpoints = ["127.0.0.1:2379","127.0.0.2:2379","127.0.0.3:2379"]

[metric]
#### 将 metrics 推送给 Prometheus pushgateway 的时间间隔
interval = "15s"
#### Prometheus pushgateway 的地址
address = ""
job = "tikv"
```

```
[raftstore]
#### Raft RocksDB 目录。默认值是 [storage.data-dir] 的 raft 子目录。
#### 如果机器上有多块磁盘，可以将 Raft RocksDB 的数据放在不同的盘上，提高 TiKV 的性能。
#### raftdb-path = "/tmp/tikv/store/raft"

#### 当 Region 写入的数据量超过该阈值的时候，TiKV 会检查该 Region 是否需要分裂。为了减少检查过程
#### 中扫描数据的成本，导入数据过程中可以将该值设置为 32 MB，正常运行状态下使用默认值即可。
region-split-check-diff = "32MB"

[coprocessor]

##### 当区间为 [a,e) 的 Region 的大小超过 `region_max_size`，TiKV 会尝试分裂该 Region，例如分裂成
    ↪  [a,b)、[b,c)、[c,d)、[d,e) 等区间的 Region 后
##### 这些 Region [a,b), [b,c), [c,d) 的大小为 `region_split_size`（或者稍大于 `region_split_size
    ↪ `）
#### region-max-size = "144MB"
#### region-split-size = "96MB"

[rocksdb]
#### RocksDB 进行后台任务的最大线程数，后台任务包括 compaction 和 flush。具体 RocksDB
    ↪ 为什么需要进行 compaction，
#### 请参考 RocksDB 的相关资料。在写流量比较大的时候（例如导数据），建议开启更多的线程，
#### 但应小于 CPU 的核数。例如在导数据的时候，32 核 CPU 的机器，可以设置成 28。
#### max-background-jobs = 8

#### RocksDB 能够打开的最大文件句柄数。
#### max-open-files = 40960

#### RocksDB MANIFEST 文件的大小限制.# 更详细的信息请参考：https://github.com/facebook/rocksdb/
    ↪ wiki/MANIFEST
max-manifest-file-size = "20MB"

#### RocksDB write-ahead logs 目录。如果机器上有两块盘，可以将 RocksDB 的数据和 WAL 日志放在
#### 不同的盘上，提高 TiKV 的性能。
#### wal-dir = "/tmp/tikv/store"

#### 下面两个参数用于怎样处理 RocksDB 归档 WAL。
#### 更多详细信息请参考：https://github.com/facebook/rocksdb/wiki/How-to-persist-in-memory-
    ↪ RocksDB-database%3F
#### wal-ttl-seconds = 0
#### wal-size-limit = 0

#### RocksDB WAL 日志的最大总大小，通常情况下使用默认值就可以了。
#### max-total-wal-size = "4GB"
```

```
#### 开启 RocksDB compaction 过程中的预读功能，如果使用的是机械磁盘，建议该值至少为2MB。
#### compaction-readahead-size = "2MB"

[rocksdb.defaultcf]
#### 数据块大小。RocksDB 是按照 block 为单元对数据进行压缩的，同时 block 也是缓存在 block-cache
#### 中的最小单元（类似其他数据库的 page 概念）。
block-size = "64KB"

#### RocksDB 每一层数据的压缩方式，可选的值为：no,snappy,zlib,bzip2,lz4,lz4hc,zstd。
#### no:no:lz4:lz4:lz4:zstd:zstd 表示 level0 和 level1 不压缩，level2 到 level4 采用 lz4 压缩算法
    ↪ ，
#### level5 和 level6 采用 zstd 压缩算法,。
#### no 表示没有压缩，lz4 是速度和压缩比较为中庸的压缩算法，zlib 的压缩比很高，对存储空间比较友
#### 好，但是压缩速度比较慢，压缩的时候需要占用较多的 CPU 资源。不同的机器需要根据 CPU 以及 I/O
    ↪ 资
#### 源情况来配置怎样的压缩方式。例如：如果采用的压缩方式为"no:no:lz4:lz4:lz4:zstd:zstd"，在大量
#### 写入数据的情况下（导数据），发现系统的 I/O 压力很大（使用 iostat 发现 %util 持续 100% 或者使
#### 用 top 命令发现 iowait 特别多），而 CPU 的资源还比较充裕，这个时候可以考虑将 level0 和
#### level1 开启压缩，用 CPU 资源换取 I/O 资源。如果采用的压缩方式
#### 为"no:no:lz4:lz4:lz4:zstd:zstd"，在大量写入数据的情况下，发现系统的 I/O 压力不大，但是 CPU
#### 资源已经吃光了，top -H 发现有大量的 bg 开头的线程（RocksDB 的 compaction 线程）在运行，这
#### 个时候可以考虑用 I/O 资源换取 CPU 资源，将压缩方式改成"no:no:no:lz4:lz4:zstd:zstd"。总之，目
#### 的是为了最大限度地利用系统的现有资源，使 TiKV 的性能在现有的资源情况下充分发挥。
compression-per-level = ["no", "no", "lz4", "lz4", "lz4", "zstd", "zstd"]

#### RocksDB memtable 的大小。
write-buffer-size = "128MB"

#### 最多允许几个 memtable 存在。写入到 RocksDB 的数据首先会记录到 WAL 日志里面，然后会插入到
#### memtable 里面，当 memtable 的大小到达了 write-buffer-size 限定的大小的时候，当前的
#### memtable 会变成只读的，然后生成一个新的 memtable 接收新的写入。只读的 memtable 会被
#### RocksDB 的 flush 线程（max-background-flushes 参数能够控制 flush 线程的最大个数）
#### flush 到磁盘，成为 level0 的一个 sst 文件。当 flush 线程忙不过来，导致等待 flush 到磁盘的
#### memtable 的数量到达 max-write-buffer-number 限定的个数的时候，RocksDB 会将新的写入
#### stall 住，stall 是 RocksDB 的一种流控机制。在导数据的时候可以将 max-write-buffer-number
#### 的值设置的更大一点，例如 10。
max-write-buffer-number = 5

#### 当 level0 的 sst 文件个数到达 level0-slowdown-writes-trigger 指定的限度的时候，
#### RocksDB 会尝试减慢写入的速度。因为 level0 的 sst 太多会导致 RocksDB 的读放大上升。
#### level0-slowdown-writes-trigger 和 level0-stop-writes-trigger 是 RocksDB 进行流控的
#### 另一个表现。当 level0 的 sst 的文件个数到达 4（默认值），level0 的 sst 文件会和 level1 中
#### 有 overlap 的 sst 文件进行 compaction，缓解读放大的问题。
level0-slowdown-writes-trigger = 20
```

```
#### 当 level0 的 sst 文件个数到达 level0-stop-writes-trigger 指定的限度的时候，RocksDB 会
#### stall 住新的写入。
level0-stop-writes-trigger = 36

#### 当 level1 的数据量大小达到 max-bytes-for-level-base 限定的值的时候，会触发 level1 的
#### sst 和 level2 种有 overlap 的 sst 进行 compaction。
#### 黄金定律：max-bytes-for-level-base 的设置的第一参考原则就是保证和 level0 的数据量大致相
#### 等，这样能够减少不必要的 compaction。例如压缩方式为"no:no:lz4:lz4:lz4:lz4:lz4"，那么
#### max-bytes-for-level-base 的值应该是 write-buffer-size 的大小乘以 4，因为 level0 和
#### level1 都没有压缩，而且 level0 触发 compaction 的条件是 sst 的个数到达 4（默认值）。在
#### level0 和 level1 都采取了压缩的情况下，就需要分析下 RocksDB 的日志，看一个 memtable 的压
#### 缩成一个 sst 文件的大小大概是多少，例如 32MB，那么 max-bytes-for-level-base 的建议值就应
#### 该是 32MB * 4 = 128MB。
max-bytes-for-level-base = "512MB"

#### sst 文件的大小。level0 的 sst 文件的大小受 write-buffer-size 和 level0 采用的压缩算法的
#### 影响，target-file-size-base 参数用于控制 level1-level6 单个 sst 文件的大小。
target-file-size-base = "32MB"

[rocksdb.writecf]
#### 保持和 rocksdb.defaultcf.compression-per-level 一致。
compression-per-level = ["no", "no", "lz4", "lz4", "lz4", "zstd", "zstd"]

#### 保持和 rocksdb.defaultcf.write-buffer-size 一致。
write-buffer-size = "128MB"
max-write-buffer-number = 5
min-write-buffer-number-to-merge = 1

#### 保持和 rocksdb.defaultcf.max-bytes-for-level-base 一致。
max-bytes-for-level-base = "512MB"
target-file-size-base = "32MB"

[raftdb]
#### RaftDB 能够打开的最大文件句柄数。
#### max-open-files = 40960

#### 开启 RaftDB compaction 过程中的预读功能，如果使用的是机械磁盘，建议该值至少为2MB。
#### compaction-readahead-size = "2MB"

[raftdb.defaultcf]
#### 保持和 rocksdb.defaultcf.compression-per-level 一致。
compression-per-level = ["no", "no", "lz4", "lz4", "lz4", "zstd", "zstd"]

#### 保持和 rocksdb.defaultcf.write-buffer-size 一致。
```

```
write-buffer-size = "128MB"
max-write-buffer-number = 5
min-write-buffer-number-to-merge = 1

#### 保持和 rocksdb.defaultcf.max-bytes-for-level-base 一致。
max-bytes-for-level-base = "512MB"
target-file-size-base = "32MB"
```

11.2.0.4.2 TiKV 内存使用情况

除了以上列出的 block-cache 以及 write-buffer 会占用系统内存外：

1. 需预留一些内存作为系统的 page cache
2. TiKV 在处理大的查询的时候（例如 select * from ...）会读取数据然后在内存中生成对应的数据结构返回给 TiDB，这个过程中 TiKV 会占用一部分内存

11.2.0.4.3 TiKV 机器配置推荐

1. 生产环境中，不建议将 TiKV 部署在 CPU 核数小于 8 或内存低于 32GB 的机器上
2. 如果对写入吞吐要求比较高，建议使用吞吐能力比较好的磁盘
3. 如果对读写的延迟要求非常高，建议使用 IOPS 比较高的 SSD 盘

11.2.0.5 Follower Read

当系统中存在读取热点 Region 导致 leader 资源紧张成为整个系统读取瓶颈时，启用 Follower Read 功能可明显降低 leader 的负担，并且通过在多个 follower 之间均衡负载，显著地提升整体系统的吞吐能力。本文主要介绍 Follower Read 的使用方法与实现机制。

11.2.0.5.1 概述

Follower Read 功能是指在强一致性读的前提下使用 Region 的 follower 副本来承载数据读取的任务，从而提升 TiDB 集群的吞吐能力并降低 leader 负载。Follower Read 包含一系列将 TiKV 读取负载从 Region 的 leader 副本上 offload 到 follower 副本的负载均衡机制。TiKV 的 Follower Read 可以保证数据读取的一致性，可以为用户提供强一致的数据读取能力。

> 注意：
>
> 为了获得强一致读取的能力，在当前的实现中，follower 节点需要向 leader 节点询问当前的执行进度（即 ReadIndex），这会产生一次额外的网络请求开销，因此目前 Follower Read 的主要优势是将集群中的读请求与写请求隔离开，并提升整体的读吞吐量。

11.2.0.5.2 使用方式

要开启 TiDB 的 Follower Read 功能，将变量 tidb_replica_read 的值设置为对应的目标值即可：

```
set [session | global] tidb_replica_read = '<目标值>';
```

作用域：SESSION | GLOBAL

默认值：leader

该变量用于设置期待的数据读取方式。

- 当设置为默认值 leader 或者空字符串时，TiDB 会维持原有行为方式，将所有的读取操作都发送给 leader 副本处理。
- 当设置为 follower 时，TiDB 会选择 Region 的 follower 副本完成所有的数据读取操作。
- 当设置为 leader-and-follower 时，TiDB 可以选择任意副本来执行读取操作，此时读请求会在 leader 和 follower 之间负载均衡。
- 当设置为 prefer-leader 时，TiDB 会优先选择 leader 副本执行读取操作。当 leader 副本的处理速度明显变慢时，例如由于磁盘或网络性能抖动，TiDB 将选择其他可用的 follower 副本来执行读取操作。
- 当设置为 closest-replicas 时，TiDB 会优先选择分布在同一可用区的副本执行读取操作，对应的副本可以是 leader 或 follower。如果同一可用区内没有副本分布，则会从 leader 执行读取。
- 当设置为 closest-adaptive 时：
  - 当一个读请求的预估返回结果大于或等于变量tidb_adaptive_closest_read_threshold 的值时，TiDB 会优先选择分布在同一可用区的副本执行读取操作。此时，为了避免读流量在各个可用区分布不均衡，TiDB 会动态检测当前在线的所有 TiDB 和 TiKV 的可用区数量分布，在每个可用区中 closest-adaptive 配置实际生效的 TiDB 节点数总是与包含 TiDB 节点最少的可用区中的 TiDB 节点数相同，并将其他多出的 TiDB 节点自动切换为读取 leader 副本。例如，如果 TiDB 分布在 3 个可用区，其中 A 和 B 两个可用区各包含 3 个 TiDB 节点，C 可用区只包含 2 个 TiDB 节点，那么每个可用区中 closest-adaptive 实际生效的 TiDB 节点数为 2，A 和 B 可用区中各有 1 个节点自动被切换为读取 leader 副本。
  - 当一个读请求的预估返回结果小于变量tidb_adaptive_closest_read_threshold 的值时，TiDB 会选择 leader 副本执行读取操作。
- 当设置为 learner 时，TiDB 会选择 learner 副本执行读取操作。在读取时，如果当前 Region 没有 learner 副本，TiDB 会报错。

> 注意：
>
> 当设置为 closest-replicas 或 closest-adaptive 时，你需要配置集群以确保副本按照指定的设置分布在各个可用区。请参考通过拓扑 label 进行副本调度为 PD 配置 location-labels 并为 TiDB 和 TiKV 设置正确的 labels。TiDB 依赖 zone 标签匹配位于同一可用区的 TiKV，因此请务必在 PD 的 location-labels 配置中包含 zone 并确保每个 TiDB 和 TiKV 节点的 labels 配置中包含 zone。如果是使用 TiDB Operator 部署的集群，请参考数据的高可用进行配置。

11.2.0.5.3 实现机制

在 Follower Read 功能出现之前，TiDB 采用 strong leader 策略将所有的读写操作全部提交到 Region 的 leader 节点上完成。虽然 TiKV 能够很均匀地将 Region 分散到多个物理节点上，但是对于每一个 Region 来说，只有 leader 副本能够对外提供服务，另外的 follower 除了时刻同步数据准备着 failover 时投票切换成为 leader 外，没有办法对 TiDB 的请求提供任何帮助。

为了允许在 TiKV 的 follower 节点进行数据读取，同时又不破坏线性一致性和 Snapshot Isolation 的事务隔离，Region 的 follower 节点需要使用 Raft `ReadIndex` 协议确保当前读请求可以读到当前 leader 上已经 commit 的最新数据。在 TiDB 层面，Follower Read 只需根据负载均衡策略将某个 Region 的读取请求发送到 follower 节点。

Follower 强一致读

TiKV follower 节点处理读取请求时，首先使用 Raft `ReadIndex` 协议与 Region 当前的 leader 进行一次交互，来获取当前 Raft group 最新的 commit index。本地 apply 到所获取的 leader 最新 commit index 后，便可以开始正常的读取请求处理流程。

Follower 副本选择策略

由于 TiKV 的 Follower Read 不会破坏 TiDB 的 Snapshot Isolation 事务隔离级别，因此 TiDB 选择 follower 的策略可以采用 round robin 的方式。目前，对于 Coprocessor 请求，Follower Read 负载均衡策略粒度是连接级别的，对于一个 TiDB 的客户端连接在某个具体的 Region 上会固定使用同一个 follower，只有在选中的 follower 发生故障或者因调度策略发生调整的情况下才会进行切换。而对于非 Coprocessor 请求（点查等），Follower Read 负载均衡策略粒度是事务级别的，对于一个 TiDB 的事务在某个具体的 Region 上会固定使用同一个 follower，同样在 follower 发生故障或者因调度策略发生调整的情况下才会进行切换。

11.2.0.6 Region 性能调优

本文介绍了如何通过调整 Region 大小等方法对 Region 进行性能调优以及如何在大 Region 下使用 bucket 进行并发查询优化。

11.2.0.6.1 概述

TiKV 自动将底层数据进行分片，所有数据按照 key 的范围划分为若干个 Region。当某个 Region 的大小超过一定限制后，TiKV 会将它分裂为多个 Region。

在大量数据的场景下，可能会出现 Region 数量过多，从而带来更多的资源开销和导致性能回退的问题。在固定数据量下，Region 越大，则 Region 个数越少。从 v6.1.0 开始，TiDB 支持设置自定义的 Region 大小。Region 默认的大小约为 96 MiB，将其调大可以减少 Region 个数。

开启Hibernate Region 或`Region Merge` 也可以减少过多 Region 带来的性能开销。

11.2.0.6.2 使用 `region-split-size` 调整 Region 大小

> **警告：**
>
> 自定义 Region 大小是在 TiDB v6.1.0 引入的实验特性，不建议在生产环境中配置。使用此特性的风险包括：

- 更容易发生性能抖动。
- 查询性能回退，尤其是大范围数据查询的性能会有回退。
- 调度变慢。

Region 的大小可以通过coprocessor.region-split-size 进行设置。推荐的 Region 大小为 96 MiB、 128 MiB、 256 MiB。region-split-size 越大，性能会越容易发生抖动。不推荐将 Region 大小设置超过 1 GiB，强烈建议不超过 10 GiB。如果你使用了 TiFlash，则 Region 大小不能超过 256 MiB。如果使用 Dumpling 工具，则 Region 大小不能超过 1 GiB。Region 调大以后，使用 Dumpling 工具时，需要降低并发，否则 TiDB 会有 OOM 的风险。

11.2.0.6.3 使用 bucket 增加并发

> **警告：**
>
> 当前该功能为实验特性，不建议在生产环境中使用。

Region 调大以后，为了增加查询并发，应当设置coprocessor.enable-region-bucket 为 true。这个配置会将每个 Region 划分为更小的区间 bucket，并且以这个更小的区间作为并发查询单位，以提高扫描数据的并发度。bucket 的大小通过coprocessor.region-bucket-size 来控制，默认值为 96MiB。

11.2.0.7 TiFlash 性能调优

本文介绍了 TiFlash 性能调优的几种方式，包括机器资源规划和 TiDB 参数调优，通过这些方式，TiFlash 性能可以达到最优状态。

11.2.0.7.1 资源规划

对于希望节省机器资源，并且完全没有隔离要求的场景，可以使用 TiKV 和 TiFlash 联合部署。建议为 TiKV 与 TiFlash 分别留够资源，同时避免共享磁盘。

11.2.0.7.2 TiDB 相关参数调优

本部分介绍如何通过调整 TiDB 相关参数来提升 TiFlash 性能，具体包括如下几个方面：

- 强制开启 MPP 模式
- 聚合下推 Join 或 Union
- 开启 Distinct 优化
- 使用 ALTER TABLE...COMPACT 整理数据
- 使用 Broadcast Hash Join 代替 Shuffled Hash Join
- 设置更大的执行并发度
- 设置细粒度 Shuffle 参数

强制开启 MPP 模式

MPP 执行计划可以充分利用分布式计算资源，从而显著提高批量数据查询的效率。当查询没有生成 MPP 执行计划的时候，你可以强制开启 MPP：

tidb_enforce_mpp 变量用于控制是否忽略优化器代价估算，强制使用 TiFlash 的 MPP 模式执行查询。要开启 MPP 模式查询，执行如下命令：

```
set @@tidb_enforce_mpp = ON;
```

在以下示例中，tidb_enforce_mpp 开启前，TiDB 需要从 TiKV 读取数据，并在 TiDB 侧执行 Join 和 Aggregation。tidb_enforce_mpp 开启后，Join 和 Aggregation 被下推到了 TiFlash。需要注意的是，有时候优化器生成的执行计划并不稳定，可能生成 MPP，也可能不生成，这时候可以开启 tidb_enforce_mpp，强制优化器生成 MPP 计划。

MPP 模式开启前：

```
mysql> explain analyze select o_orderpriority, count(*) as order_count from orders where
    ↪ o_orderdate >= '1995-01-01' and o_orderdate < date_add('1995-01-01', interval '3' month)
    ↪ and exists (select * from lineitem where l_orderkey = o_orderkey and l_commitdate <
    ↪ l_receiptdate) group by o_orderpriority;
+--
    ↪ ----------------------------------+-------------+----------+----------+-----------------------------
    ↪
| id                                | estRows      | actRows   | task      | access object
    ↪                                          | execution info
    ↪
    ↪ | operator info
    ↪
    ↪ | memory     | disk |
+--
    ↪ ----------------------------------+-------------+----------+----------+-----------------------------
    ↪
| Projection_9                      | 1.00         | 5         | root      |
    ↪                                                      | time:22.8s, loops:5,
    ↪ Concurrency:OFF
    ↪
    ↪ | tpch_100.orders.o_orderpriority, Column#27
    ↪
    ↪ | 1016 Bytes | N/A  |
| └─HashAgg_10                       | 1.00         | 5         | root      |
    ↪                                                     | time:22.8s, loops:5,
    ↪ partial_worker:{wall_time:22.817403422s, concurrency:5, task_num:5027, tot_wait:1m52
    ↪ .560509138s, tot_exec:1.523754659s, tot_time:1m54.086196522s, max:22.817363394s, p95
    ↪ :22.817363394s}, final_worker:{wall_time:22.817416046s, concurrency:5, task_num:20,
    ↪ tot_wait:1m54.086784741s, tot_exec:155.94µs, tot_time:1m54.086948982s, max:22.817396284s,
    ↪  p95:22.817396284s}
    ↪
    ↪ | group by:tpch_100.orders.o_orderpriority, funcs:count(1)->Column#27, funcs:firstrow(
```

```
    ↪ tpch_100.orders.o_orderpriority)->tpch_100.orders.o_orderpriority
    ↪                                  | 511.7 KB    | N/A   |
|   └─IndexHashJoin_18                 | 4429262.13    | 5145138    | root      |
    ↪                                                              | time:22.8s, loops:5028, inner:{
    ↪ total:1m52.6s, concurrency:5, task:227, construct:5.17s, fetch:1m40s, build:1.22s, join
    ↪ :7.49s}
    ↪
    ↪ | semi join, inner:IndexLookUp_15, outer key:tpch_100.orders.o_orderkey, inner key:
    ↪ tpch_100.lineitem.l_orderkey, equal cond:eq(tpch_100.orders.o_orderkey, tpch_100.lineitem
    ↪ .l_orderkey) | 46.8 MB     | N/A   |
|     ├─TableReader_37(Build)          | 5536577.67    | 5610951    | root      |
    ↪                                                              | time:4.89s, loops:5467,
    ↪ cop_task: {num: 2127, max: 408.7ms, min: 277.5µs, avg: 81.6ms, p95: 249.7ms,
    ↪ max_proc_keys: 226272, p95_proc_keys: 220128, tot_proc: 2m50.2s, tot_wait: 40ms, rpc_num:
    ↪  2127, rpc_time: 2m53.6s, copr_cache_hit_ratio: 0.02, distsql_concurrency: 15}
    ↪
    ↪ | data:Selection_36
    ↪
    ↪ | 3.42 MB     | N/A   |
|     │ └─Selection_36                 | 5536577.67    | 5610951    | cop[tikv] |
    ↪                                                              | tikv_task:{proc max:402ms, min
    ↪ :0s, avg: 79.7ms, p80:151ms, p95:245ms, iters:155122, tasks:2127}, scan_detail: {
    ↪ total_process_keys: 149478336, total_process_keys_size: 22705157593, total_keys:
    ↪ 149480413, get_snapshot_time: 187.9ms, rocksdb: {key_skipped_count: 149478336, block: {
    ↪ cache_hit_count: 393690}}}
    ↪
    ↪ | ge(tpch_100.orders.o_orderdate, 1995-01-01 00:00:00.000000), lt(tpch_100.orders.
    ↪ o_orderdate, 1995-04-01 00:00:00.000000)
    ↪                                                                  | N/A         | N/A   |
|     │   └─TableFullScan_35           | 150000000.00 | 150000000 | cop[tikv] | table:orders
    ↪                                              | tikv_task:{proc max:382ms, min:0s, avg: 74.6
    ↪ ms, p80:141ms, p95:229ms, iters:155122, tasks:2127}
    ↪
    ↪ | keep order:false
    ↪
    ↪ | N/A         | N/A   |
|     └─IndexLookUp_15(Probe)          | 22018496.89   | 14185840   | root      |
    ↪                                                              | time:1m33.4s, loops:14204,
    ↪ index_task: {total_time: 1m10.8s, fetch_handle: 1m10.7s, build: 3.9ms, wait: 13.3ms},
    ↪ table_task: {total_time: 1m44.4s, num: 1989, concurrency: 5}
    ↪
    ↪ |
    ↪
    ↪ | 17.9 KB     | N/A   |
|       ├─IndexRangeScan_12(Build)     | 27523121.11   | 22441356   | cop[tikv] | table:lineitem,
```

```
    ↪ index:PRIMARY(L_ORDERKEY, L_LINENUMBER) | time:1m8.8s, loops:23635, cop_task: {num:
    ↪ 12537, max: 179.3ms, min: 331.4µs, avg: 22.7ms, p95: 66.9ms, max_proc_keys: 13755,
    ↪ p95_proc_keys: 5088, tot_proc: 3m44s, tot_wait: 129ms, rpc_num: 12537, rpc_time: 4m44s,
    ↪ copr_cache_hit_ratio: 0.06, distsql_concurrency: 15}, tikv_task:{proc max:170ms, min:0s,
    ↪ avg: 18.8ms, p80:29ms, p95:62ms, iters:68823, tasks:12537}, scan_detail: {
    ↪ total_process_keys: 21985403, total_process_keys_size: 1165226359, total_keys: 27492546,
    ↪ get_snapshot_time: 563.1ms, rocksdb: {key_skipped_count: 21985403, block: {
    ↪ cache_hit_count: 44004354}}}    | range: decided by [eq(tpch_100.lineitem.l_orderkey,
    ↪ tpch_100.orders.o_orderkey)], keep order:false
    ↪                                                                                | N/A
    ↪         | N/A  |
|       └─Selection_14(Probe)          | 22018496.89  | 14185840  | cop[tikv] |
    ↪                                                          | time:1m34.8s, loops:16882,
    ↪ cop_task: {num: 12057, max: 240.4ms, min: 328.5µs, avg: 20.4ms, p95: 73.1ms,
    ↪ max_proc_keys: 15005, p95_proc_keys: 6118, tot_proc: 3m39.8s, tot_wait: 300ms, rpc_num:
    ↪ 12057, rpc_time: 4m5.7s, copr_cache_hit_ratio: 0.28, distsql_concurrency: 15}, tikv_task
    ↪ :{proc max:230ms, min:0s, avg: 27.5ms, p80:45ms, p95:82ms, iters:74165, tasks:12057},
    ↪ scan_detail: {total_process_keys: 19916937, total_process_keys_size: 3957799619,
    ↪ total_keys: 24053143, get_snapshot_time: 1.07s, rocksdb: {key_skipped_count: 19254567,
    ↪ block: {cache_hit_count: 37816908}}}  | lt(tpch_100.lineitem.l_commitdate, tpch_100.
    ↪ lineitem.l_receiptdate)
    ↪
    ↪ | N/A        | N/A  |
|         └─TableRowIDScan_13          | 27523121.11  | 22441356  | cop[tikv] | table:lineitem
    ↪                                          | tikv_task:{proc max:230ms, min:0s, avg: 27.2ms
    ↪ , p80:45ms, p95:81ms, iters:74165, tasks:12057}
    ↪
    ↪ | keep order:false
    ↪
    ↪ | N/A        | N/A  |
+--
    ↪ ------------------------------------+--------------+-----------+-----------+-----------------------------
    ↪
10 rows in set (22.82 sec)
```

开启 MPP 模式：

```
mysql> set @@tidb_enforce_mpp = ON;
Query OK, 0 rows affected (0.00 sec)
```

MPP 模式开启后：

```
mysql> explain analyze select o_orderpriority, count(*) as order_count from orders where
    ↪ o_orderdate >= '1995-01-01' and o_orderdate < date_add('1995-01-01', interval '3' month)
    ↪ and exists (select * from lineitem where l_orderkey = o_orderkey and l_commitdate <
    ↪ l_receiptdate) group by o_orderpriority;
```

```
+--
    ↪ ----------------------------------------------+--------------+-----------+--------------+--------------
    ↪
| id                                              | estRows      | actRows   | task         |
    ↪ access object  | execution info
    ↪
    ↪ | operator info
    ↪
    ↪ | memory | disk |
+--
    ↪ ----------------------------------------------+--------------+-----------+--------------+--------------
    ↪
| TableReader_59                                  | 1.00         | 5         | root         |
    ↪                | time:5.97s, loops:2, cop_task: {num: 8, max: 0s, min: 0s, avg: 0s, p95:
    ↪ 0s, copr_cache_hit_ratio: 0.00}
    ↪
    ↪ | data:ExchangeSender_58
    ↪
    ↪ | N/A    | N/A  |
| └─ExchangeSender_58                             | 1.00         | 5         | mpp[tiflash] |
    ↪                | tiflash_task:{proc max:5.97s, min:5.97s, avg: 5.97s, p80:5.97s, p95:5.97
    ↪ s, iters:5, tasks:3, threads:60}
    ↪
    ↪ | ExchangeType: PassThrough
    ↪
    ↪ | N/A    | N/A  |
|   └─Projection_10                               | 1.00         | 5         | mpp[tiflash] |
    ↪                | tiflash_task:{proc max:5.97s, min:5.96s, avg: 5.97s, p80:5.97s, p95:5.97
    ↪ s, iters:5, tasks:3, threads:60}
    ↪
    ↪ | tpch_100.orders.o_orderpriority, Column#27
    ↪
    ↪ | N/A    | N/A  |
|     └─Projection_54                             | 1.00         | 5         | mpp[tiflash] |
    ↪                | tiflash_task:{proc max:5.97s, min:5.96s, avg: 5.97s, p80:5.97s, p95:5.97
    ↪ s, iters:5, tasks:3, threads:60}
    ↪
    ↪ | Column#27, tpch_100.orders.o_orderpriority
    ↪
    ↪ | N/A    | N/A  |
|       └─HashAgg_55                              | 1.00         | 5         | mpp[tiflash] |
    ↪                | tiflash_task:{proc max:5.97s, min:5.96s, avg: 5.97s, p80:5.97s, p95:5.97
    ↪ s, iters:5, tasks:3, threads:60}
    ↪
    ↪ | group by:tpch_100.orders.o_orderpriority, funcs:sum(Column#32)->Column#27, funcs:
```

```
    ↪ firstrow(tpch_100.orders.o_orderpriority)->tpch_100.orders.o_orderpriority, stream_count:
    ↪  20 | N/A    | N/A  |
|         └─ExchangeReceiver_57                    | 1.00         | 15        | mpp[tiflash] |
    ↪                | tiflash_task:{proc max:5.97s, min:5.96s, avg: 5.97s, p80:5.97s, p95:5.97
    ↪ s, iters:5, tasks:3, threads:60}
    ↪
    ↪ | stream_count: 20
    ↪
    ↪ | N/A    | N/A  |
|           └─ExchangeSender_56                    | 1.00         | 15        | mpp[tiflash] |
    ↪                | tiflash_task:{proc max:5.96s, min:0s, avg: 1.99s, p80:5.96s, p95:5.96s,
    ↪ iters:768, tasks:3, threads:60}
    ↪
    ↪ | ExchangeType: HashPartition, Hash Cols: [name: tpch_100.orders.o_orderpriority, collate
    ↪ : utf8mb4_bin], stream_count: 20                                                          |
    ↪ N/A    | N/A  |
|             └─HashAgg_52                         | 1.00         | 15        | mpp[tiflash] |
    ↪                | tiflash_task:{proc max:5.96s, min:0s, avg: 1.99s, p80:5.96s, p95:5.96s,
    ↪ iters:768, tasks:3, threads:60}
    ↪
    ↪ | group by:tpch_100.orders.o_orderpriority, funcs:count(1)->Column#32
    ↪
    ↪ | N/A    | N/A  |
|               └─HashJoin_41                      | 4429262.13   | 5145138   | mpp[tiflash] |
    ↪                | tiflash_task:{proc max:5.95s, min:0s, avg: 1.98s, p80:5.95s, p95:5.95s,
    ↪ iters:118, tasks:3, threads:60}
    ↪
    ↪ | semi join, equal:[eq(tpch_100.orders.o_orderkey, tpch_100.lineitem.l_orderkey)],
    ↪ stream_count: 20
    ↪                                                                                | N/A    | N/
    ↪ A  |
|                 ├─ExchangeReceiver_26(Build)     | 480030321.60 | 379356474 | mpp[tiflash] |
    ↪                | tiflash_task:{proc max:3.26s, min:0s, avg: 1.09s, p80:3.26s, p95:3.26s,
    ↪ iters:42469, tasks:3, threads:60}
    ↪
    ↪ | stream_count: 20
    ↪
    ↪ | N/A    | N/A  |
|                 │ └─ExchangeSender_25            | 480030321.60 | 379356474 | mpp[tiflash] |
    ↪                | tiflash_task:{proc max:3.37s, min:0s, avg: 1.12s, p80:3.37s, p95:3.37s,
    ↪ iters:9292, tasks:3, threads:60}
    ↪
    ↪ | ExchangeType: HashPartition, Hash Cols: [name: tpch_100.lineitem.l_orderkey, collate:
    ↪ binary], stream_count: 20                                                                |
    ↪  N/A    | N/A  |
```

```
| │   └─Projection_60              | 480030321.60 | 379356474 | mpp[tiflash] |
    ↪                | tiflash_task:{proc max:819.1ms, min:0s, avg: 273ms, p80:819.1ms, p95
    ↪ :819.1ms, iters:9292, tasks:3, threads:60}
    ↪
    ↪ | tpch_100.lineitem.l_orderkey
    ↪
    ↪ | N/A    | N/A  |
| │     └─Selection_24             | 480030321.60 | 379356474 | mpp[tiflash] |
    ↪                | tiflash_task:{proc max:817.1ms, min:0s, avg: 272.4ms, p80:817.1ms, p95
    ↪ :817.1ms, iters:9292, tasks:3, threads:60}
    ↪
    ↪ | lt(tpch_100.lineitem.l_commitdate, tpch_100.lineitem.l_receiptdate)
    ↪
    ↪ | N/A    | N/A  |
| │       └─TableFullScan_23       | 600037902.00 | 600037902 | mpp[tiflash] |
    ↪ table:lineitem | tiflash_task:{proc max:328.1ms, min:0s, avg: 109.4ms, p80:328.1ms, p95
    ↪ :328.1ms, iters:9292, tasks:3, threads:60}, tiflash_scan:{dtfile:{total_scanned_packs
    ↪ :73475, total_skipped_packs:13452, total_scanned_rows:600230532, total_skipped_rows
    ↪ :109887612, total_rs_index_load_time: 35ms, total_read_time: 34065ms},
    ↪ total_create_snapshot_time: 0ms} | keep order:false
    ↪
    ↪ | N/A    | N/A  |
| └─ExchangeReceiver_22(Probe)     | 5536577.67   | 5610951   | mpp[tiflash] |
    ↪                | tiflash_task:{proc max:2.5s, min:0s, avg: 833.7ms, p80:2.5s, p95:2.5s,
    ↪ iters:669, tasks:3, threads:60}
    ↪
    ↪ |
    ↪
    ↪ | N/A    | N/A  |
|   └─ExchangeSender_21            | 5536577.67   | 5610951   | mpp[tiflash] |
    ↪                  | tiflash_task:{proc max:5.93s, min:0s, avg: 1.98s, p80:5.93s, p95:5.93s,
    ↪ iters:2336, tasks:3, threads:60}
    ↪
    ↪ | ExchangeType: HashPartition, Hash Cols: [name: tpch_100.orders.o_orderkey, collate:
    ↪ binary]
    ↪                                                                                | N/A
    ↪    | N/A  |
|     └─Selection_20               | 5536577.67   | 5610951   | mpp[tiflash] |
    ↪                | tiflash_task:{proc max:325.5ms, min:0s, avg: 108.5ms, p80:325.5ms, p95
    ↪ :325.5ms, iters:2336, tasks:3, threads:60}
    ↪
    ↪ | ge(tpch_100.orders.o_orderdate, 1995-01-01 00:00:00.000000), lt(tpch_100.orders.
    ↪ o_orderdate, 1995-04-01 00:00:00.000000)
    ↪                                                        | N/A    | N/A  |
|       └─TableFullScan_19         | 150000000.00 | 150000000 | mpp[tiflash] |
```

```
    ↪ table:orders   | tiflash_task:{proc max:238.5ms, min:0s, avg: 79.5ms, p80:238.5ms, p95
    ↪ :238.5ms, iters:2336, tasks:3, threads:60}, tiflash_scan:{dtfile:{total_scanned_packs
    ↪ :18541, total_skipped_packs:186, total_scanned_rows:150000611, total_skipped_rows
    ↪ :1513328, total_rs_index_load_time: 75ms, total_read_time: 11454ms},
    ↪ total_create_snapshot_time: 0ms}      | keep order:false
    ↪
    ↪ | N/A    | N/A  |
+--
    ↪ ------------------------------------------------+-------------+----------+-------------+-------------
    ↪
18 rows in set (6.00 sec)
```

聚合下推 Join 或 Union

将聚合操作下推到 Join 或 Union 之前执行，有可能能显著减少 Join 或 Union 需要处理的数据量，从而提升性能。

tidb_opt_agg_push_down 变量用来设置优化器是否执行聚合函数下推到 Join 之前的优化操作。当查询中聚合操作执行很慢时，可以尝试设置该变量为 ON。

```
set @@tidb_opt_agg_push_down = ON;
```

在以下示例中，tidb_opt_agg_push_down 开启前，HashAgg_58 在 HashJoin_41 之后执行。tidb_opt_agg_push_down 开启后，新生成的 HashAgg_21 和 HashAgg_32 在 HashJoin_76 之前执行，显著减少了 Join 需要处理的数据量。

tidb_opt_agg_push_down 开启前:

```
mysql> explain analyze select count(*) from t1 join t2 where t1.a = t2.b group by t1.a;
+--
    ↪ ---------------------------------------+-------------+----------+-------------+----------------+------
    ↪
| id                                    | estRows      | actRows   | task         | access
    ↪ object | execution info
    ↪
    ↪ | operator info
    ↪                                                                                 |
    ↪ memory | disk |
+--
    ↪ ---------------------------------------+-------------+----------+-------------+----------------+------
    ↪
| TableReader_66                        | 100450000.00 | 20        | root         |
    ↪               | time:2.13s, loops:2, cop_task: {num: 21, max: 0s, min: 0s, avg: 0s, p95:
    ↪ 0s, copr_cache_hit_ratio: 0.00}
    ↪
    ↪ | data:ExchangeSender_65
    ↪                                                                          | N/A    | N/
    ↪ A  |
```

```
| └─ExchangeSender_65                     | 100450000.00 | 20        | mpp[tiflash] |
    ↪               | tiflash_task:{proc max:2.13s, min:2.08s, avg: 2.11s, p80:2.13s, p95:2.13s
    ↪ , iters:768, tasks:3, threads:60}
    ↪
    ↪ | ExchangeType: PassThrough
    ↪                                                                          | N/A    | N/A
    ↪ |
|   └─Projection_60                       | 100450000.00 | 20        | mpp[tiflash] |
    ↪               | tiflash_task:{proc max:2.13s, min:2.08s, avg: 2.11s, p80:2.13s, p95:2.13s
    ↪ , iters:768, tasks:3, threads:60}
    ↪
    ↪ | Column#43
    ↪
    ↪ | N/A    | N/A  |
|     └─HashAgg_58                        | 100450000.00 | 20        | mpp[tiflash] |
    ↪               | tiflash_task:{proc max:2.13s, min:2.08s, avg: 2.11s, p80:2.13s, p95:2.13s
    ↪ , iters:768, tasks:3, threads:60}
    ↪
    ↪ | group by:test.t1.a, funcs:count(1)->Column#43
    ↪                                                          | N/A    | N/A  |
|       └─HashJoin_41                     | 600000000.00 | 600000000 | mpp[tiflash] |
    ↪               | tiflash_task:{proc max:1.77s, min:1.67s, avg: 1.73s, p80:1.77s, p95:1.77s
    ↪ , iters:10955, tasks:3, threads:60}
    ↪
    ↪ | inner join, equal:[eq(test.t1.a, test.t2.b)], stream_count: 20
    ↪                                    | N/A    | N/A  |
|         ├─ExchangeReceiver_22(Build)    | 100450000.00 | 99980000  | mpp[tiflash] |
    ↪               | tiflash_task:{proc max:171ms, min:127ms, avg: 154.5ms, p80:171ms, p95:171
    ↪ ms, iters:10440, tasks:3, threads:60}
    ↪
    ↪ | stream_count: 20
    ↪                                                                                | N/A
    ↪    | N/A  |
|         │ └─ExchangeSender_21           | 100450000.00 | 99980000  | mpp[tiflash] |
    ↪               | tiflash_task:{proc max:498.7ms, min:0s, avg: 166.2ms, p80:498.7ms, p95
    ↪ :498.7ms, iters:1653, tasks:3, threads:60}
    ↪
    ↪ | ExchangeType: HashPartition, Hash Cols: [name: test.t1.a, collate: binary],
    ↪ stream_count: 20       | N/A    | N/A  |
|         │   └─Selection_20              | 100450000.00 | 99980000  | mpp[tiflash] |
    ↪               | tiflash_task:{proc max:83.7ms, min:0s, avg: 27.9ms, p80:83.7ms, p95:83.7
    ↪ ms, iters:1653, tasks:3, threads:60}
    ↪
    ↪ | not(isnull(test.t1.a))
    ↪                                                                        | N/A    | N/
```

```
    ↪ A  |
|         |     └─TableFullScan_19         | 100450000.00 | 99980000  | mpp[tiflash] | table:t1
    ↪      | tiflash_task:{proc max:41.7ms, min:0s, avg: 13.9ms, p80:41.7ms, p95:41.7ms, iters
    ↪ :1653, tasks:3, threads:60}, tiflash_scan:{dtfile:{total_scanned_packs:12336,
    ↪ total_skipped_packs:5759, total_scanned_rows:100083566, total_skipped_rows:46786721,
    ↪ total_rs_index_load_time: 5ms, total_read_time: 1105ms}, total_create_snapshot_time: 0ms}
    ↪  | keep order:false
    ↪                                                                                | N/A
    ↪    | N/A  |
|         └─ExchangeReceiver_26(Probe)     | 600000000.00 | 600000000 | mpp[tiflash] |
    ↪               | tiflash_task:{proc max:373ms, min:71ms, avg: 253.2ms, p80:373ms, p95:373
    ↪ ms, iters:28135, tasks:3, threads:60}
    ↪
    ↪ |
    ↪
    ↪ | N/A    | N/A  |
|           └─ExchangeSender_25            | 600000000.00 | 600000000 | mpp[tiflash] |
    ↪               | tiflash_task:{proc max:2.07s, min:0s, avg: 689.6ms, p80:2.07s, p95:2.07s,
    ↪  iters:9388, tasks:3, threads:60}
    ↪
    ↪ | ExchangeType: HashPartition, Hash Cols: [name: test.t2.b, collate: binary]
    ↪                          | N/A    | N/A  |
|             └─Selection_24               | 600000000.00 | 600000000 | mpp[tiflash] |
    ↪               | tiflash_task:{proc max:146.9ms, min:0s, avg: 49ms, p80:146.9ms, p95:146.9
    ↪ ms, iters:9388, tasks:3, threads:60}
    ↪
    ↪ | not(isnull(test.t2.b))
    ↪                                                                      | N/A    | N/
    ↪ A  |
|               └─TableFullScan_23         | 600000000.00 | 600000000 | mpp[tiflash] | table:t2
    ↪      | tiflash_task:{proc max:63.8ms, min:0s, avg: 21.3ms, p80:63.8ms, p95:63.8ms, iters
    ↪ :9388, tasks:3, threads:60}, tiflash_scan:{dtfile:{total_scanned_packs:73834,
    ↪ total_skipped_packs:439, total_scanned_rows:600007458, total_skipped_rows:3553131,
    ↪ total_rs_index_load_time: 12ms, total_read_time: 270ms}, total_create_snapshot_time: 0ms}
    ↪    | keep order:false
    ↪                                                                                | N/A
    ↪    | N/A  |
+--
    ↪ ----------------------------------------+--------------+-----------+--------------+----------------+------
    ↪
13 rows in set (2.15 sec)
```

开启 tidb_opt_agg_push_down：

```
mysql> set @@tidb_opt_agg_push_down = ON;
Query OK, 0 rows affected (0.00 sec)
```

开启 tidb_opt_agg_push_down 后：

```
mysql> explain analyze select count(*) from t1 join t2 where t1.a = t2.b group by t1.a;
+--
    ↪ ------------------------------------------------+--------------+-----------+--------------+----------
    ↪
| id                                                | estRows      | actRows   | task
    ↪ | access object | execution info
    ↪
    ↪ | operator info
    ↪
    ↪ | memory | disk |
+--
    ↪ ------------------------------------------------+--------------+-----------+--------------+----------
    ↪
| TableReader_85                                    | 20.00        | 20        | root
    ↪ |               | time:432.4ms, loops:2, cop_task: {num: 19, max: 0s, min: 0s, avg: 0s,
    ↪ p95: 0s, copr_cache_hit_ratio: 0.00}
    ↪
    ↪ | data:ExchangeSender_84
    ↪
    ↪ | N/A    | N/A  |
| └─ExchangeSender_84                               | 20.00        | 20        | mpp[tiflash]
    ↪  |               | tiflash_task:{proc max:431.1ms, min:429.3ms, avg: 430.4ms, p80:431.1ms
    ↪ , p95:431.1ms, iters:16, tasks:3, threads:60}
    ↪
    ↪ | ExchangeType: PassThrough
    ↪
    ↪ | N/A    | N/A  |
|   └─Projection_83                                 | 20.00        | 20        | mpp[tiflash]
    ↪  |               | tiflash_task:{proc max:430.1ms, min:429.3ms, avg: 429.7ms, p80:430.1ms
    ↪ , p95:430.1ms, iters:16, tasks:3, threads:60}
    ↪
    ↪ | Column#43
    ↪
    ↪ | N/A    | N/A  |
|     └─HashAgg_82                                 | 20.00        | 20        | mpp[tiflash]
    ↪  |               | tiflash_task:{proc max:430.1ms, min:429.3ms, avg: 429.7ms, p80:430.1ms
    ↪ , p95:430.1ms, iters:16, tasks:3, threads:60}
    ↪
    ↪ | group by:test.t1.a, funcs:count(Column#44)->Column#43, stream_count: 20
    ↪                                                    | N/A    | N/A  |
|       └─ExchangeReceiver_78                       | 20.00        | 20        | mpp[tiflash]
    ↪  |               | tiflash_task:{proc max:430.1ms, min:429.3ms, avg: 429.7ms, p80:430.1ms
```

```
    ↪ , p95:430.1ms, iters:16, tasks:3, threads:60}
    ↪
    ↪ | stream_count: 20
    ↪
    ↪ | N/A    | N/A  |
|         └─ExchangeSender_77                          | 20.00        | 20        | mpp[tiflash]
    ↪  |               | tiflash_task:{proc max:427ms, min:0s, avg: 142.3ms, p80:427ms, p95:427
    ↪ ms, iters:60, tasks:3, threads:60}
    ↪
    ↪ | ExchangeType: HashPartition, Hash Cols: [name: test.t1.a, collate: binary],
    ↪ stream_count: 20                                  | N/A    | N/A  |
|           └─HashJoin_76                              | 20.00        | 20        | mpp[tiflash]
    ↪  |               | tiflash_task:{proc max:426ms, min:0s, avg: 142ms, p80:426ms, p95:426ms
    ↪ , iters:60, tasks:3, threads:60}
    ↪
    ↪ | inner join, equal:[eq(test.t1.a, test.t2.b)]
    ↪                                                                                  | N/A
    ↪    | N/A  |
|             ├─ExchangeReceiver_36(Build)             | 20.00        | 60        | mpp[tiflash]
    ↪  |               | tiflash_task:{proc max:374ms, min:0s, avg: 124.7ms, p80:374ms, p95:374
    ↪ ms, iters:46, tasks:3, threads:60}
    ↪
    ↪ |
    ↪
    ↪ | N/A    | N/A  |
|             │ └─ExchangeSender_35                    | 20.00        | 20        | mpp[tiflash]
    ↪  |               | tiflash_task:{proc max:373.7ms, min:0s, avg: 124.6ms, p80:373.7ms, p95
    ↪ :373.7ms, iters:16, tasks:3, threads:60}
    ↪
    ↪ | ExchangeType: Broadcast
    ↪
    ↪ | N/A    | N/A  |
|             │   └─Projection_31                      | 20.00        | 20        | mpp[tiflash]
    ↪  |               | tiflash_task:{proc max:373.7ms, min:0s, avg: 124.6ms, p80:373.7ms, p95
    ↪ :373.7ms, iters:16, tasks:3, threads:60}
    ↪
    ↪ | Column#44, test.t2.b
    ↪
    ↪ | N/A    | N/A  |
|             │     └─HashAgg_32                       | 20.00        | 20        | mpp[tiflash]
    ↪  |               | tiflash_task:{proc max:373.7ms, min:0s, avg: 124.6ms, p80:373.7ms, p95
    ↪ :373.7ms, iters:16, tasks:3, threads:60}
    ↪
    ↪ | group by:test.t2.b, funcs:sum(Column#45)->Column#44, funcs:firstrow(test.t2.b)->test.t2
    ↪ .b, stream_count: 20                   | N/A    | N/A  |
```

```
|             |       └─ExchangeReceiver_34            | 20.00        | 60        | mpp[tiflash]
    ↪  |               | tiflash_task:{proc max:373.7ms, min:0s, avg: 124.6ms, p80:373.7ms, p95
    ↪ :373.7ms, iters:16, tasks:3, threads:60}
    ↪
    ↪ | stream_count: 20
    ↪
    ↪ | N/A    | N/A  |
|             |         └─ExchangeSender_33            | 20.00        | 60        | mpp[tiflash]
    ↪  |               | tiflash_task:{proc max:368.3ms, min:0s, avg: 122.8ms, p80:368.3ms, p95
    ↪ :368.3ms, iters:3, tasks:3, threads:60}
    ↪
    ↪ | ExchangeType: HashPartition, Hash Cols: [name: test.t2.b, collate: binary],
    ↪ stream_count: 20                                  | N/A    | N/A  |
|             |           └─HashAgg_21                 | 20.00        | 60        | mpp[tiflash]
    ↪  |               | tiflash_task:{proc max:367.3ms, min:0s, avg: 122.4ms, p80:367.3ms, p95
    ↪ :367.3ms, iters:3, tasks:3, threads:60}
    ↪
    ↪ | group by:test.t2.b, funcs:count(1)->Column#45
    ↪                                                                                  | N/A
    ↪ | N/A  |
|             |             └─Selection_30             | 600000000.00 | 600000000 | mpp[tiflash]
    ↪  |               | tiflash_task:{proc max:77.2ms, min:0s, avg: 25.7ms, p80:77.2ms, p95
    ↪ :77.2ms, iters:9389, tasks:3, threads:60}
    ↪
    ↪ | not(isnull(test.t2.b))
    ↪
    ↪ | N/A    | N/A  |
|             |               └─TableFullScan_29       | 600000000.00 | 600000000 | mpp[tiflash]
    ↪  | table:t2      | tiflash_task:{proc max:37.2ms, min:0s, avg: 12.4ms, p80:37.2ms, p95
    ↪ :37.2ms, iters:9389, tasks:3, threads:60}, tiflash_scan:{dtfile:{total_scanned_packs
    ↪ :73834, total_skipped_packs:412, total_scanned_rows:600002896, total_skipped_rows
    ↪ :3342145, total_rs_index_load_time: 0ms, total_read_time: 45ms},
    ↪ total_create_snapshot_time: 0ms} | keep order:false
    ↪
    ↪ | N/A    | N/A  |
|             └─Selection_19(Probe)                  | 100450000.00 | 99980000  | mpp[tiflash]
    ↪  |               | tiflash_task:{proc max:30ms, min:0s, avg: 10ms, p80:30ms, p95:30ms,
    ↪ iters:1645, tasks:3, threads:60}
    ↪
    ↪ | not(isnull(test.t1.a))
    ↪
    ↪ | N/A    | N/A  |
|               └─TableFullScan_18                   | 100450000.00 | 99980000  | mpp[tiflash]
    ↪  | table:t1      | tiflash_task:{proc max:22ms, min:0s, avg: 7.34ms, p80:22ms, p95:22ms,
    ↪ iters:1645, tasks:3, threads:60}, tiflash_scan:{dtfile:{total_scanned_packs:12326,
```

```
    ↪ total_skipped_packs:4762, total_scanned_rows:100045535, total_skipped_rows:38651216,
    ↪ total_rs_index_load_time: 4ms, total_read_time: 355ms}, total_create_snapshot_time: 0ms}
    ↪    | keep order:false
    ↪
    ↪ | N/A    | N/A  |
+--
    ↪ ------------------------------------------------+--------------+-----------+--------------+----------
    ↪
18 rows in set (0.46 sec)
```

开启 Distinct 优化

TiFlash 暂时还不支持部分可接受 Distinct 列的聚合函数，比如 Sum。默认情况下，整个聚合函数运算都会在 TiDB 端执行。通过开启 Distinct 优化，部分操作可以下推到 TiFlash，从而提升查询性能：

tidb_opt_distinct_agg_push_down 变量用来设置优化器是否执行带有 Distinct 的聚合函数（比如 select ↪ sum(distinct a)from t）下推到 Coprocessor 的优化操作。当查询中带有 Distinct 的聚合操作执行很慢时，可以尝试设置该变量为 ON。

```
set @@tidb_opt_distinct_agg_push_down = ON;
```

在以下示例中，tidb_opt_distinct_agg_push_down 开启前，TiDB 需要从 TiFlash 读取所有数据，并在 TiDB 侧执行 distinct。tidb_opt_distinct_agg_push_down 开启后，distinct a 被下推到了 TiFlash，在 HashAgg_6 里新增里一个 group by 列 test.t.a。这边查询结果里面的两个 warnings 是警告聚合函数不能完全下推。

tidb_opt_distinct_agg_push_down 开启前：

```
mysql> explain analyze select count(distinct a) from test.t;
+--
    ↪ -------------------------+--------------+-----------+--------------+---------------+-----------------
    ↪
| id                        | estRows      | actRows   | task         | access object |
    ↪ execution info
    ↪
    ↪ | operator info                                        | memory    | disk     |
+--
    ↪ -------------------------+--------------+-----------+--------------+---------------+-----------------
    ↪
| HashAgg_6                 | 1.00         | 1         | root         |               | time:2
    ↪ m23.2s, loops:2
    ↪
    ↪ | funcs:sum(distinct Column#23)->Column#22               | 41.3 KB   | 0 Bytes |
| └─Projection_16           | 600000000.00 | 600000000 | root         |               | time
    ↪ :2.51s, loops:586548, Concurrency:5
    ↪
    ↪ | cast(test.t.a, decimal(10,0) BINARY)->Column#23        | 243.2 KB | N/A      |
|   └─TableReader_11        | 600000000.00 | 600000000 | root         |               | time
    ↪ :1.6s, loops:586548, cop_task: {num: 1173, max: 256.2ms, min: 25.1ms, avg: 46.9ms, p95:
```

```
    ↪ 63.5ms, rpc_num: 1173, rpc_time: 55s, copr_cache_hit_ratio: 0.00, distsql_concurrency:
    ↪ 15}
    ↪
    ↪ | data:TableFullScan_10                                  | 70.2 MB   | N/A      |
|     └─TableFullScan_10     | 600000000.00 | 600000000 | cop[tiflash] | table:t       |
    ↪ tiflash_task:{proc max:9ms, min:531µs, avg: 4.49ms, p80:5.55ms, p95:6.74ms, iters:9390,
    ↪ tasks:1173, threads:1173}, tiflash_scan:{dtfile:{total_scanned_packs:73834,
    ↪ total_skipped_packs:1231, total_scanned_rows:600010914, total_skipped_rows:9988978,
    ↪ total_rs_index_load_time: 0ms, total_read_time: 16ms}, total_create_snapshot_time: 0ms}
    ↪ | keep order:false                                      | N/A       | N/A      |
+--
    ↪ -------------------------+-------------+-----------+-------------+---------------+-----------------
    ↪
4 rows in set, 2 warnings (2 min 23.21 sec)
```

开启 tidb_opt_distinct_agg_push_down：

```
mysql> set @@tidb_opt_distinct_agg_push_down = ON;
Query OK, 0 rows affected (0.00 sec)
```

tidb_opt_distinct_agg_push_down 开启后：

```
mysql> explain analyze select count(distinct a) from test.t;
+--
    ↪ -------------------------+-------------+-----------+------------------+---------------+-------------
    ↪
| id                          | estRows     | actRows   | task             | access object |
    ↪ execution info
    ↪
    ↪ | operator info                                        | memory    | disk     |
+--
    ↪ -------------------------+-------------+-----------+------------------+---------------+-------------
    ↪
| HashAgg_10                  | 1.00        | 1         | root             |               |
    ↪ time:233.8ms, loops:2
    ↪
    ↪ | funcs:sum(distinct Column#23)->Column#22               | 2.42 KB   | 0 Bytes |
|  └─Projection_12            | 1.00        | 3         | root             |               |
    ↪ time:233.7ms, loops:2, Concurrency:OFF
    ↪
    ↪ | cast(test.c.a, decimal(10,0) BINARY)->Column#23        | 380 Bytes | N/A      |
|    └─TableReader_11         | 1.00        | 3         | root             |               |
    ↪ time:233.7ms, loops:2, cop_task: {num: 6, max: 0s, min: 0s, avg: 0s, p95: 0s,
    ↪ copr_cache_hit_ratio: 0.00, distsql_concurrency: 15}
    ↪
    ↪ | data:HashAgg_6                                       | 100 Bytes | N/A      |
```

```
|     └─HashAgg_6              | 1.00         | 3         | batchCop[tiflash] |               |
    ↪ tiflash_task:{proc max:225.8ms, min:210.7ms, avg: 216.9ms, p80:225.8ms, p95:225.8ms,
    ↪ iters:3, tasks:3, threads:60}
    ↪
    ↪ | group by:test.t.a,                                    | N/A       | N/A     |
|       └─TableFullScan_9      | 600000000.00 | 600000000 | batchCop[tiflash] | table:t       |
    ↪ tiflash_task:{proc max:50.3ms, min:33.7ms, avg: 44.6ms, p80:50.3ms, p95:50.3ms, iters
    ↪ :9387, tasks:3, threads:60}, tiflash_scan:{dtfile:{total_scanned_packs:73833,
    ↪ total_skipped_packs:475, total_scanned_rows:600000000, total_skipped_rows:3852098,
    ↪ total_rs_index_load_time: 0ms, total_read_time: 84ms}, total_create_snapshot_time: 0ms} |
    ↪  keep order:false                                      | N/A       | N/A     |
+--
    ↪ --------------------------+--------------+-----------+-------------------+---------------+--------------
    ↪
5 rows in set, 2 warnings (0.24 sec)
```

使用 ALTER TABLE...COMPACT 整理数据

ALTER TABLE ... COMPACT 可以触发 TiFlash 节点对某个表或者某个分区进行数据整理。数据整理时，表中的物理数据会被重写，如清理已删除的数据、合并多版本数据等，从而可以获得更高的访问性能，并减少磁盘空间占用。

```
ALTER TABLE employees COMPACT TIFLASH REPLICA;
```

```
ALTER TABLE employees COMPACT PARTITION pNorth, pEast TIFLASH REPLICA;
```

使用 Broadcast Hash Join 代替 Shuffled Hash Join

对于有小表的 Join 算子，Broadcast Hash Join 可以避免大表的网络传输，从而提升计算性能。

- tidb_broadcast_join_threshold_size，单位为 bytes。如果表大小（字节数）小于该值，则选择 Broadcast Hash Join 算法。否则选择 Shuffled Hash Join 算法。

  ```
  set @@tidb_broadcast_join_threshold_size = 2000000;
  ```

- tidb_broadcast_join_threshold_count，单位为行数。如果 join 的对象为子查询，优化器无法估计子查询结果集大小，在这种情况下通过结果集行数判断。如果子查询的行数估计值小于该变量，则选择 Broadcast Hash Join 算法。否则选择 Shuffled Hash Join 算法。

  ```
  set @@tidb_broadcast_join_threshold_count = 100000;
  ```

在以下示例中，tidb_broadcast_join_threshold_size 设置前，ExchangeSender_29 的 ExchangeType 是 HashPartition。tidb_broadcast_join_threshold_size 设置后，ExchangeSender_29 的 ExchangeType 改变为 Broadcast。

tidb_broadcast_join_threshold_size 设置前：

```
mysql> explain analyze select max(l_shipdate), max(l_commitdate), max(l_receiptdate) from
    ↪ supplier,lineitem where s_suppkey = l_suppkey;
+--
    ↪ ----------------------------------------+-------------+-----------+-------------+----------------+-----
    ↪
| id                                      | estRows      | actRows   | task         | access
    ↪ object  | execution info
    ↪
    ↪ | operator info
    ↪
    ↪ | memory  | disk |
+--
    ↪ ----------------------------------------+-------------+-----------+-------------+----------------+-----
    ↪
| HashAgg_45                              | 1.00         | 1         | root         |
    ↪                | time:3.8s, loops:2, partial_worker:{wall_time:3.798284809s, concurrency
    ↪ :5, task_num:1, tot_wait:18.99079929s, tot_exec:8.193µs, tot_time:18.990819019s, max
    ↪ :3.798181723s, p95:3.798181723s}, final_worker:{wall_time:0s, concurrency:5, task_num:1,
    ↪ tot_wait:18.991291379s, tot_exec:197.45µs, tot_time:18.991494363s, max:3.798334972s, p95
    ↪ :3.798334972s}   | funcs:max(Column#28)->Column#25, funcs:max(Column#29)->Column#26,
    ↪ funcs:max(Column#30)->Column#27
    ↪                                                                          | 17.3 KB | N/A  |
| └─TableReader_47                         | 1.00         | 3         | root         |
    ↪                | time:3.8s, loops:2, cop_task: {num: 6, max: 0s, min: 0s, avg: 0s, p95: 0
    ↪ s, copr_cache_hit_ratio: 0.00}
    ↪
    ↪ | data:ExchangeSender_46
    ↪
    ↪ | N/A     | N/A  |
|   └─ExchangeSender_46                    | 1.00         | 3         | mpp[tiflash] |
    ↪                | tiflash_task:{proc max:3.79s, min:3.78s, avg: 3.78s, p80:3.79s, p95:3.79
    ↪ s, iters:3, tasks:3, threads:60}
    ↪
    ↪ | ExchangeType: PassThrough
    ↪
    ↪ | N/A     | N/A  |
|     └─HashAgg_13                         | 1.00         | 3         | mpp[tiflash] |
    ↪                | tiflash_task:{proc max:3.79s, min:3.78s, avg: 3.78s, p80:3.79s, p95:3.79
    ↪ s, iters:3, tasks:3, threads:60}
    ↪
    ↪ | funcs:max(tpch_100.lineitem.l_shipdate)->Column#28, funcs:max(tpch_100.lineitem.
    ↪ l_commitdate)->Column#29, funcs:max(tpch_100.lineitem.l_receiptdate)->Column#30 | N/A
    ↪     | N/A  |
|       └─HashJoin_44                      | 600845438.27 | 600037902 | mpp[tiflash] |
```

```
    ↪                | tiflash_task:{proc max:3.17s, min:3.12s, avg: 3.14s, p80:3.17s, p95:3.17
    ↪ s, iters:11143, tasks:3, threads:60}
    ↪
    ↪ | inner join, equal:[eq(tpch_100.supplier.s_suppkey, tpch_100.lineitem.l_suppkey)],
    ↪ stream_count: 20                                                                  | N/A     |
    ↪  N/A  |
|         ├─ExchangeReceiver_30(Build)     | 1000000.00   | 1000000   | mpp[tiflash] |
    ↪                | tiflash_task:{proc max:23.3ms, min:21ms, avg: 22.5ms, p80:23.3ms, p95
    ↪ :23.3ms, iters:120, tasks:3, threads:60}
    ↪
    ↪ | stream_count: 20
    ↪
    ↪ | N/A     | N/A  |
|         │ └─ExchangeSender_29           | 1000000.00   | 1000000   | mpp[tiflash] |
    ↪                | tiflash_task:{proc max:22.7ms, min:0s, avg: 7.57ms, p80:22.7ms, p95:22.7
    ↪ ms, iters:16, tasks:3, threads:2}
    ↪
    ↪ | ExchangeType: HashPartition, Hash Cols: [name: tpch_100.supplier.s_suppkey, collate:
    ↪ binary], stream_count: 20                                                         | N/A     | N/
    ↪ A  |
|         │   └─TableFullScan_28          | 1000000.00   | 1000000   | mpp[tiflash] | table:
    ↪ supplier | tiflash_task:{proc max:9.71ms, min:0s, avg: 3.24ms, p80:9.71ms, p95:9.71ms,
    ↪ iters:16, tasks:3, threads:2}, tiflash_scan:{dtfile:{total_scanned_packs:123,
    ↪ total_skipped_packs:0, total_scanned_rows:1000000, total_skipped_rows:0,
    ↪ total_rs_index_load_time: 0ms, total_read_time: 2ms}, total_create_snapshot_time: 0ms}
    ↪                                    | keep order:false
    ↪
    ↪ | N/A     | N/A  |
|         └─ExchangeReceiver_33(Probe)     | 600037902.00 | 600037902 | mpp[tiflash] |
    ↪                | tiflash_task:{proc max:564.3ms, min:340ms, avg: 438.5ms, p80:564.3ms,
    ↪ p95:564.3ms, iters:27583, tasks:3, threads:60}
    ↪
    ↪ |
    ↪
    ↪ | N/A     | N/A  |
|           └─ExchangeSender_32           | 600037902.00 | 600037902 | mpp[tiflash] |
    ↪                | tiflash_task:{proc max:3.69s, min:0s, avg: 1.23s, p80:3.69s, p95:3.69s,
    ↪ iters:9298, tasks:3, threads:60}
    ↪
    ↪ | ExchangeType: HashPartition, Hash Cols: [name: tpch_100.lineitem.l_suppkey, collate:
    ↪ binary]                                                                           | N/A     | N/
    ↪ A  |
|             └─TableFullScan_31          | 600037902.00 | 600037902 | mpp[tiflash] | table:
    ↪ lineitem | tiflash_task:{proc max:62.8ms, min:0s, avg: 20.9ms, p80:62.8ms, p95:62.8ms,
    ↪ iters:9298, tasks:3, threads:60}, tiflash_scan:{dtfile:{total_scanned_packs:73465,
```

```
    ↪ total_skipped_packs:13060, total_scanned_rows:600168663, total_skipped_rows:106699256,
    ↪ total_rs_index_load_time: 16ms, total_read_time: 27979ms}, total_create_snapshot_time: 0
    ↪ ms}            | keep order:false
    ↪
    ↪ | N/A     | N/A  |
+--
    ↪ ----------------------------------------+-------------+-----------+-------------+----------------+-----
    ↪
11 rows in set (3.83 sec)
```

设置 tidb_broadcast_join_threshold_size：

```
mysql> set @@tidb_broadcast_join_threshold_size = 10000000;
Query OK, 0 rows affected (0.00 sec)
```

tidb_broadcast_join_threshold_size 设置后：

```
mysql> explain analyze select max(l_shipdate), max(l_commitdate), max(l_receiptdate) from
    ↪ supplier,lineitem where s_suppkey = l_suppkey;
+--
    ↪ ----------------------------------------+-------------+-----------+-------------+----------------+-----
    ↪
| id                                    | estRows      | actRows   | task         | access
    ↪ object  | execution info
    ↪
    ↪ | operator info
    ↪
    ↪ | memory  | disk |
+--
    ↪ ----------------------------------------+-------------+-----------+-------------+----------------+-----
    ↪
| HashAgg_43                            | 1.00         | 1         | root         |
    ↪                | time:2.75s, loops:2, partial_worker:{wall_time:2.748445779s, concurrency
    ↪ :5, task_num:1, tot_wait:13.74202679s, tot_exec:8.012µs, tot_time:13.742045721s, max
    ↪ :2.748414915s, p95:2.748414915s}, final_worker:{wall_time:2.74848039s, concurrency:5,
    ↪ task_num:1, tot_wait:13.742157526s, tot_exec:53.191µs, tot_time:13.742214417s, max
    ↪ :2.748462393s, p95:2.748462393s}   | funcs:max(Column#28)->Column#25, funcs:max(Column
    ↪ #29)->Column#26, funcs:max(Column#30)->Column#27
    ↪                                                                    | 17.3 KB | N/A  |
| └─TableReader_45                       | 1.00         | 3         | root         |
    ↪                | time:2.75s, loops:2, cop_task: {num: 6, max: 0s, min: 0s, avg: 0s, p95:
    ↪ 0s, copr_cache_hit_ratio: 0.00}
    ↪
    ↪ | data:ExchangeSender_44
    ↪
    ↪ | N/A     | N/A  |
```

```
|   └─ExchangeSender_44                    | 1.00          | 3          | mpp[tiflash] |
    ↪                | tiflash_task:{proc max:2.74s, min:2.65s, avg: 2.7s, p80:2.74s, p95:2.74s
    ↪ , iters:3, tasks:3, threads:60}
    ↪
    ↪ | ExchangeType: PassThrough
    ↪
    ↪ | N/A     | N/A  |
|     └─HashAgg_13                        | 1.00          | 3          | mpp[tiflash] |
    ↪                | tiflash_task:{proc max:2.74s, min:2.65s, avg: 2.7s, p80:2.74s, p95:2.74s
    ↪ , iters:3, tasks:3, threads:60}
    ↪
    ↪ | funcs:max(tpch_100.lineitem.l_shipdate)->Column#28, funcs:max(tpch_100.lineitem.
    ↪ l_commitdate)->Column#29, funcs:max(tpch_100.lineitem.l_receiptdate)->Column#30 | N/A
    ↪     | N/A  |
|       └─HashJoin_42                     | 600845438.27  | 600037902  | mpp[tiflash] |
    ↪                | tiflash_task:{proc max:2.15s, min:2.06s, avg: 2.11s, p80:2.15s, p95:2.15
    ↪ s, iters:9286, tasks:3, threads:60}
    ↪
    ↪ | inner join, equal:[eq(tpch_100.supplier.s_suppkey, tpch_100.lineitem.l_suppkey)]
    ↪                                                                                | N/A
    ↪ | N/A  |
|         ├─ExchangeReceiver_30(Build)    | 1000000.00    | 3000000    | mpp[tiflash] |
    ↪                | tiflash_task:{proc max:28.1ms, min:24.9ms, avg: 27ms, p80:28.1ms, p95
    ↪ :28.1ms, iters:48, tasks:3, threads:60}
    ↪
    ↪ |
    ↪
    ↪ | N/A     | N/A  |
|         │ └─ExchangeSender_29           | 1000000.00    | 1000000    | mpp[tiflash] |
    ↪                | tiflash_task:{proc max:14ms, min:0s, avg: 4.67ms, p80:14ms, p95:14ms,
    ↪ iters:16, tasks:3, threads:2}
    ↪
    ↪ | ExchangeType: Broadcast
    ↪
    ↪ | N/A     | N/A  |
|         │   └─TableFullScan_28          | 1000000.00    | 1000000    | mpp[tiflash] | table:
    ↪ supplier | tiflash_task:{proc max:9ms, min:0s, avg: 3ms, p80:9ms, p95:9ms, iters:16,
    ↪ tasks:3, threads:2}, tiflash_scan:{dtfile:{total_scanned_packs:123, total_skipped_packs
    ↪ :0, total_scanned_rows:1000000, total_skipped_rows:0, total_rs_index_load_time: 0ms,
    ↪ total_read_time: 2ms}, total_create_snapshot_time: 0ms}
    ↪                                                          | keep order:false
    ↪
    ↪ | N/A     | N/A  |
|         └─TableFullScan_31(Probe)       | 600037902.00  | 600037902  | mpp[tiflash] | table:
    ↪ lineitem | tiflash_task:{proc max:57.9ms, min:42.9ms, avg: 51.3ms, p80:57.9ms, p95:57.9ms |
```

```
    ↪ , iters:9297, tasks:3, threads:60}, tiflash_scan:{dtfile:{total_scanned_packs:73464,
    ↪ total_skipped_packs:12985, total_scanned_rows:600169085, total_skipped_rows:106014866,
    ↪ total_rs_index_load_time: 23ms, total_read_time: 21667ms}, total_create_snapshot_time: 0
    ↪ ms}                  | keep order:false
    ↪
    ↪ | N/A     | N/A  |
+--
    ↪ ----------------------------------------+-------------+----------+-------------+----------------+-----
    ↪
9 rows in set (2.76 sec)
```

设置更大的执行并发度

设置更大的执行并发度，可以让 TiFlash 占用更多系统 CPU 资源，从而提升查询性能。

tidb_max_tiflash_threads，单位为线程。用来设置 TiFlash 中 request 执行的最大并发度。

```
set @@tidb_max_tiflash_threads = 20;
```

在以下示例中，tidb_max_tiflash_threads 设置前，所有 TiFlash 的算子的执行线程数为 24。tidb_max_tiflash_threads
↪ 设置后，所有 TiFlash 的算子的执行线程数为 60。

tidb_max_tiflash_threads 设置前：

```
mysql> explain analyze select a, count(*) from t group by a;
+--
    ↪ ----------------------------------+-------------+----------+-------------+---------------+----------
    ↪
| id                                  | estRows      | actRows   | task         | access object
    ↪ | execution info
    ↪
    ↪ | operator info
    ↪
    ↪ | memory | disk |
+--
    ↪ ----------------------------------+-------------+----------+-------------+---------------+----------
    ↪
| TableReader_44                      | 20.00        | 20        | root         |
    ↪ | time:655.9ms, loops:2, cop_task: {num: 14, max: 0s, min: 0s, avg: 0s, p95: 0s,
    ↪ copr_cache_hit_ratio: 0.00}
    ↪
    ↪ | data:ExchangeSender_43
    ↪
    ↪ | N/A    | N/A  |
| └─ExchangeSender_43                  | 20.00        | 20        | mpp[tiflash] |
    ↪                | tiflash_task:{proc max:654.6ms, min:652.4ms, avg: 653.8ms, p80:654.6ms,
    ↪ p95:654.6ms, iters:11, tasks:3, threads:24}
    ↪
```

```
    ↪ | ExchangeType: PassThrough
    ↪
    ↪ | N/A    | N/A  |
|   └─Projection_5                    | 20.00        | 20        | mpp[tiflash] |
    ↪               | tiflash_task:{proc max:654.3ms, min:651.4ms, avg: 653.1ms, p80:654.3ms,
    ↪ p95:654.3ms, iters:11, tasks:3, threads:24}
    ↪
    ↪ | test.t.a, Column#22
    ↪
    ↪ | N/A    | N/A  |
|     └─Projection_39                 | 20.00        | 20        | mpp[tiflash] |
    ↪               | tiflash_task:{proc max:654.3ms, min:651.4ms, avg: 653.1ms, p80:654.3ms,
    ↪ p95:654.3ms, iters:11, tasks:3, threads:24}
    ↪
    ↪ | Column#22, test.t.a
    ↪
    ↪ | N/A    | N/A  |
|       └─HashAgg_40                  | 20.00        | 20        | mpp[tiflash] |
    ↪               | tiflash_task:{proc max:654.3ms, min:651.4ms, avg: 653.1ms, p80:654.3ms,
    ↪ p95:654.3ms, iters:11, tasks:3, threads:24}
    ↪
    ↪ | group by:test.t.a, funcs:sum(Column#29)->Column#22, funcs:firstrow(test.t.a)->test.t.a,
    ↪  stream_count: 8                      | N/A    | N/A  |
|         └─ExchangeReceiver_42       | 20.00        | 60        | mpp[tiflash] |
    ↪               | tiflash_task:{proc max:654.3ms, min:651.4ms, avg: 653.1ms, p80:654.3ms,
    ↪ p95:654.3ms, iters:11, tasks:3, threads:24}
    ↪
    ↪ | stream_count: 8
    ↪
    ↪ | N/A    | N/A  |
|           └─ExchangeSender_41       | 20.00        | 60        | mpp[tiflash] |
    ↪               | tiflash_task:{proc max:649ms, min:0s, avg: 216.3ms, p80:649ms, p95:649ms,
    ↪  iters:3, tasks:3, threads:24}
    ↪
    ↪ | ExchangeType: HashPartition, Hash Cols: [name: test.t.a, collate: binary], stream_count
    ↪ : 8                                   | N/A    | N/A  |
|             └─HashAgg_37            | 20.00        | 60        | mpp[tiflash] |
    ↪               | tiflash_task:{proc max:649ms, min:0s, avg: 216.3ms, p80:649ms, p95:649ms,
    ↪  iters:3, tasks:3, threads:24}
    ↪
    ↪ | group by:test.t.a, funcs:count(1)->Column#29
    ↪                                                                                | N/A
    ↪ | N/A  |
|               └─TableFullScan_26    | 600000000.00 | 600000000 | mpp[tiflash] | table:t
    ↪       | tiflash_task:{proc max:40ms, min:0s, avg: 13.3ms, p80:40ms, p95:40ms, iters:9386,
```

```
    ↪  tasks:3, threads:24}, tiflash_scan:{dtfile:{total_scanned_packs:73833,
    ↪ total_skipped_packs:174, total_scanned_rows:600000000, total_skipped_rows:1402537,
    ↪ total_rs_index_load_time: 0ms, total_read_time: 4ms}, total_create_snapshot_time: 0ms} |
    ↪ keep order:false
    ↪
    ↪ | N/A    | N/A  |
+--
    ↪ ----------------------------------+-------------+----------+-------------+---------------+---------
    ↪
9 rows in set (0.67 sec)
```

设置 tidb_max_tiflash_threads：

```
mysql> set @@tidb_max_tiflash_threads = 20;
Query OK, 0 rows affected (0.00 sec)
```

tidb_max_tiflash_threads 设置后：

```
mysql> explain analyze select a, count(*) from t group by a;
+--
    ↪ ----------------------------------+-------------+----------+-------------+---------------+---------
    ↪
| id                                | estRows      | actRows   | task         | access object
    ↪ | execution info
    ↪
    ↪ | operator info
    ↪
    ↪ | memory | disk |
+--
    ↪ ----------------------------------+-------------+----------+-------------+---------------+---------
    ↪
| TableReader_44                      | 20.00        | 20        | root         |
    ↪ | time:357.7ms, loops:2, cop_task: {num: 19, max: 0s, min: 0s, avg: 0s, p95: 0s,
    ↪ copr_cache_hit_ratio: 0.00}
    ↪
    ↪ | data:ExchangeSender_43
    ↪
    ↪ | N/A    | N/A  |
| └─ExchangeSender_43                 | 20.00        | 20        | mpp[tiflash] |
    ↪               | tiflash_task:{proc max:355.3ms, min:354.6ms, avg: 355ms, p80:355.3ms, p95
    ↪ :355.3ms, iters:16, tasks:3, threads:60}
    ↪
    ↪ | ExchangeType: PassThrough
    ↪
    ↪ | N/A    | N/A  |
|   └─Projection_5                    | 20.00        | 20        | mpp[tiflash] |
    ↪               | tiflash_task:{proc max:354.3ms, min:353.6ms, avg: 354ms, p80:354.3ms, p95
```

```
    ↪ :354.3ms, iters:16, tasks:3, threads:60}
    ↪
    ↪ | test.t.a, Column#22
    ↪
    ↪ | N/A    | N/A  |
|   └─Projection_39                  | 20.00        | 20        | mpp[tiflash] |
    ↪              | tiflash_task:{proc max:354.3ms, min:353.6ms, avg: 354ms, p80:354.3ms, p95
    ↪ :354.3ms, iters:16, tasks:3, threads:60}
    ↪
    ↪ | Column#22, test.t.a
    ↪
    ↪ | N/A    | N/A  |
|     └─HashAgg_40                   | 20.00        | 20        | mpp[tiflash] |
    ↪              | tiflash_task:{proc max:354.3ms, min:353.6ms, avg: 354ms, p80:354.3ms, p95
    ↪ :354.3ms, iters:16, tasks:3, threads:60}
    ↪
    ↪ | group by:test.t.a, funcs:sum(Column#29)->Column#22, funcs:firstrow(test.t.a)->test.t.a,
    ↪  stream_count: 20                     | N/A    | N/A  |
|       └─ExchangeReceiver_42        | 20.00        | 60        | mpp[tiflash] |
    ↪              | tiflash_task:{proc max:354.3ms, min:353.6ms, avg: 354ms, p80:354.3ms, p95
    ↪ :354.3ms, iters:16, tasks:3, threads:60}
    ↪
    ↪ | stream_count: 20
    ↪
    ↪ | N/A    | N/A  |
|         └─ExchangeSender_41        | 20.00        | 60        | mpp[tiflash] |
    ↪              | tiflash_task:{proc max:349.6ms, min:0s, avg: 116.5ms, p80:349.6ms, p95
    ↪ :349.6ms, iters:3, tasks:3, threads:60}
    ↪
    ↪ | ExchangeType: HashPartition, Hash Cols: [name: test.t.a, collate: binary], stream_count
    ↪ : 20                                  | N/A    | N/A  |
|           └─HashAgg_37             | 20.00        | 60        | mpp[tiflash] |
    ↪              | tiflash_task:{proc max:347.6ms, min:0s, avg: 115.9ms, p80:347.6ms, p95
    ↪ :347.6ms, iters:3, tasks:3, threads:60}
    ↪
    ↪ | group by:test.t.a, funcs:count(1)->Column#29
    ↪                                                                                | N/A
    ↪    | N/A  |
|             └─TableFullScan_26     | 600000000.00 | 600000000 | mpp[tiflash] | table:t
    ↪       | tiflash_task:{proc max:36.6ms, min:0s, avg: 12.2ms, p80:36.6ms, p95:36.6ms, iters
    ↪ :9389, tasks:3, threads:60}, tiflash_scan:{dtfile:{total_scanned_packs:73833,
    ↪ total_skipped_packs:418, total_scanned_rows:600001386, total_skipped_rows:3381854,
    ↪ total_rs_index_load_time: 10ms, total_read_time: 61ms}, total_create_snapshot_time: 0ms}
    ↪ | keep order:false
    ↪
```

```
    ↪ | N/A    | N/A  |
+--
    ↪ ----------------------------------+-------------+----------+-------------+---------------+----------
    ↪
9 rows in set (0.37 sec)
```

设置细粒度 Shuffle 参数

细粒度 Shuffle 可以通过参数增加窗口函数执行的并发度，让函数执行占用更多系统资源，从而提升查询性能。

tiflash_fine_grained_shuffle_stream_count，单位为线程数。当窗口函数下推到 TiFlash 执行时，可以通过该变量控制窗口函数执行的并行度。

```
set @@tiflash_fine_grained_shuffle_stream_count = 20;
```

在以下示例中，tiflash_fine_grained_shuffle_stream_count 设置前，[ExchangeSender_11, ExchangeReceiver_12 ↪ , Sort_13, Window_22] 的 stream_count 是 8。tiflash_fine_grained_shuffle_stream_count 设置后，[ExchangeSender_11, ExchangeReceiver_12, Sort_13, Window_22] 的 stream_count 是 20。

tiflash_fine_grained_shuffle_stream_count 设置前：

```
mysql> explain analyze select *, row_number() over (partition by a) from t;
+--
    ↪ --------------------------------+-------------+----------+-------------+---------------+----------
    ↪
| id                              | estRows      | actRows   | task         | access object |
    ↪ execution info
    ↪
    ↪ | operator info
    ↪
    ↪ | memory | disk |
+--
    ↪ --------------------------------+-------------+----------+-------------+---------------+----------
    ↪
| TableReader_24                  | 600000000.00 | 600000000 | root         |               |
    ↪ time:4m30.5s, loops:585941, cop_task: {num: 9163, max: 0s, min: 0s, avg: 0s, p95: 0s,
    ↪ copr_cache_hit_ratio: 0.00}
    ↪
    ↪ | data:ExchangeSender_23
    ↪
    ↪ | N/A    | N/A  |
| └─ExchangeSender_23             | 600000000.00 | 600000000 | mpp[tiflash] |               |
    ↪ tiflash_task:{proc max:4m30.5s, min:3m4.8s, avg: 3m36.1s, p80:4m30.5s, p95:4m30.5s, iters
    ↪ :9160, tasks:3, threads:24}
    ↪
    ↪ | ExchangeType: PassThrough
    ↪
    ↪ | N/A    | N/A  |
```

```
|   └─Window_22                    | 600000000.00 | 600000000 | mpp[tiflash] |                |
    ↪ tiflash_task:{proc max:3m31.6s, min:2m26.2s, avg: 2m50.7s, p80:3m31.6s, p95:3m31.6s,
    ↪ iters:9160, tasks:3, threads:24}
    ↪
    ↪ | row_number()->Column#23 over(partition by test.t.a rows between current row and current
    ↪  row), stream_count: 8        | N/A    | N/A  |
|     └─Sort_13                    | 600000000.00 | 600000000 | mpp[tiflash] |                |
    ↪ tiflash_task:{proc max:3m28.6s, min:2m24.2s, avg: 2m48.4s, p80:3m28.6s, p95:3m28.6s,
    ↪ iters:9160, tasks:3, threads:24}
    ↪
    ↪ | test.t.a, stream_count: 8
    ↪
    ↪ | N/A    | N/A  |
|       └─ExchangeReceiver_12      | 600000000.00 | 600000000 | mpp[tiflash] |                |
    ↪ tiflash_task:{proc max:32.4s, min:32s, avg: 32.1s, p80:32.4s, p95:32.4s, iters:49307,
    ↪ tasks:3, threads:24}
    ↪
    ↪ | stream_count: 8
    ↪
    ↪ | N/A    | N/A  |
|         └─ExchangeSender_11      | 600000000.00 | 600000000 | mpp[tiflash] |                |
    ↪ tiflash_task:{proc max:32s, min:0s, avg: 10.7s, p80:32s, p95:32s, iters:9386, tasks:3,
    ↪ threads:60}
    ↪
    ↪ | ExchangeType: HashPartition, Hash Cols: [name: test.t.a, collate: binary], stream_count
    ↪ : 8                           | N/A    | N/A  |
|           └─TableFullScan_10     | 600000000.00 | 600000000 | mpp[tiflash] | table:t        |
    ↪ tiflash_task:{proc max:113.9ms, min:0s, avg: 38ms, p80:113.9ms, p95:113.9ms, iters:9386,
    ↪ tasks:3, threads:60}, tiflash_scan:{dtfile:{total_scanned_packs:73834,
    ↪ total_skipped_packs:190, total_scanned_rows:600000000, total_skipped_rows:1536382,
    ↪ total_rs_index_load_time: 16ms, total_read_time: 166324ms}, total_create_snapshot_time: 0
    ↪ ms} | keep order:false
    ↪
    ↪ | N/A    | N/A  |
+--
    ↪ ---------------------------------+--------------+-----------+--------------+---------------+--------------
    ↪
7 rows in set (4 min 30.59 sec)
```

设置 tiflash_fine_grained_shuffle_stream_count：

```
mysql> set @@tiflash_fine_grained_shuffle_stream_count = 20;
Query OK, 0 rows affected (0.00 sec)
```

tiflash_fine_grained_shuffle_stream_count 设置后：

```
mysql> explain analyze select *, row_number() over (partition by a) from t;
+--
    ↪ -------------------------------+-------------+----------+-------------+---------------+-------------
    ↪
| id                              | estRows      | actRows   | task         | access object |
    ↪ execution info
    ↪
    ↪ | operator info
    ↪
    ↪ | memory | disk |
+--
    ↪ -------------------------------+-------------+----------+-------------+---------------+-------------
    ↪
| TableReader_24                  | 600000000.00 | 600000000 | root         |               |
    ↪ time:2m55s, loops:585941, cop_task: {num: 9163, max: 0s, min: 0s, avg: 0s, p95: 0s,
    ↪ copr_cache_hit_ratio: 0.00}
    ↪
    ↪ | data:ExchangeSender_23
    ↪
    ↪ | N/A    | N/A  |
| └─ExchangeSender_23             | 600000000.00 | 600000000 | mpp[tiflash] |               |
    ↪ tiflash_task:{proc max:2m55s, min:1m37s, avg: 2m28.7s, p80:2m55s, p95:2m55s, iters:9160,
    ↪ tasks:3, threads:60}
    ↪
    ↪ | ExchangeType: PassThrough
    ↪
    ↪ | N/A    | N/A  |
|   └─Window_22                   | 600000000.00 | 600000000 | mpp[tiflash] |               |
    ↪ tiflash_task:{proc max:2m12.9s, min:1m17s, avg: 1m54.2s, p80:2m12.9s, p95:2m12.9s, iters
    ↪ :9160, tasks:3, threads:60}
    ↪
    ↪ | row_number()->Column#23 over(partition by test.t.a rows between current row and current
    ↪  row), stream_count: 20        | N/A    | N/A  |
|     └─Sort_13                   | 600000000.00 | 600000000 | mpp[tiflash] |               |
    ↪ tiflash_task:{proc max:2m10.9s, min:1m16s, avg: 1m52.5s, p80:2m10.9s, p95:2m10.9s, iters
    ↪ :9160, tasks:3, threads:60}
    ↪
    ↪ | test.t.a, stream_count: 20
    ↪
    ↪ | N/A    | N/A  |
|       └─ExchangeReceiver_12     | 600000000.00 | 600000000 | mpp[tiflash] |               |
    ↪ tiflash_task:{proc max:27.2s, min:25.5s, avg: 26.6s, p80:27.2s, p95:27.2s, iters:49602,
    ↪ tasks:3, threads:60}
    ↪
    ↪ | stream_count: 20
```

```
    ↪
    ↪ | N/A    | N/A  |
|         └─ExchangeSender_11      | 600000000.00 | 600000000 | mpp[tiflash] |               |
    ↪ tiflash_task:{proc max:25.5s, min:0s, avg: 8.51s, p80:25.5s, p95:25.5s, iters:9388, tasks
    ↪ :3, threads:60}
    ↪
    ↪ | ExchangeType: HashPartition, Hash Cols: [name: test.t.a, collate: binary], stream_count
    ↪ : 20                           | N/A    | N/A  |
|           └─TableFullScan_10     | 600000000.00 | 600000000 | mpp[tiflash] | table:t       |
    ↪ tiflash_task:{proc max:167.3ms, min:0s, avg: 55.8ms, p80:167.3ms, p95:167.3ms, iters
    ↪ :9388, tasks:3, threads:60}, tiflash_scan:{dtfile:{total_scanned_packs:73834,
    ↪ total_skipped_packs:408, total_scanned_rows:600002896, total_skipped_rows:3307316,
    ↪ total_rs_index_load_time: 20ms, total_read_time: 179431ms}, total_create_snapshot_time: 0
    ↪ ms} | keep order:false
    ↪
    ↪ | N/A    | N/A  |
+--
    ↪ ---------------------------------+--------------+-----------+-------------+---------------+--------------
    ↪
7 rows in set (2 min 55.09 sec)
```

11.2.0.8 下推计算结果缓存

TiDB 从 4.0 起支持下推计算结果缓存（即 Coprocessor Cache 功能）。开启该功能后，将在 TiDB 实例侧缓存下推给 TiKV 计算的结果，在部分场景下起到加速效果。

11.2.0.8.1 配置

Coprocessor Cache 的配置均位于 TiDB 的 `tikv-client.copr-cache` 配置项中。Coprocessor 的具体开启和配置方法，见TiDB 配置文件描述。

11.2.0.8.2 特性说明

- 所有 SQL 在单个 TiDB 实例上的首次执行都不会被缓存。
- 缓存仅存储在 TiDB 内存中，TiDB 重启后缓存会失效。
- 不同 TiDB 实例之间不共享缓存。
- 缓存的是下推计算结果，即使缓存命中，后续仍有 TiDB 计算。
- 缓存以 Region 为单位。对 Region 的写入会导致涉及该 Region 的缓存失效。基于此原因，该功能主要会对很少变更的数据有效果。
- 下推计算请求相同时，缓存会被命中。通常在以下场景下，下推计算的请求是相同或部分相同的：
  - SQL 语句完全一致，例如重复执行相同的 SQL 语句。
    该场景下所有下推计算的请求都是一致的，所有请求都能利用上下推计算缓存。
  - SQL 语句包含一个变化的条件，其他部分一致，变化的条件是表主键或分区主键。
    该场景下一部分下推计算的请求会与之前出现过的一致，部分请求能利用上下推计算结果缓存。

- SQL 语句包含多个变化的条件，其他部分一致，变化的条件完全匹配一个复合索引列。
  该场景下一部分下推计算的请求会与之前出现过的一致，部分请求能利用上下推计算结果缓存。

- 该功能对用户透明，开启和关闭都不影响计算结果，仅影响 SQL 执行时间。

11.2.0.8.3 检验缓存效果

可以通过执行 EXPLAIN ANALYZE 或查看 Grafana 监控面板来检查 Coprocessor 的缓存效果。

使用 EXPLAIN ANALYZE

用户可以通过EXPLAIN ANALYZE 语句查看读表算子中的缓存命中率，示例如下：

```
EXPLAIN ANALYZE SELECT * FROM t USE INDEX(a);
+--
    ↪ ----------------------------+-----------+---------+-----------+-----------------------+---------------
    ↪
| id                            | estRows   | actRows | task      | access object         |
    ↪ execution info
    ↪
    ↪ | operator info                  | memory               | disk |
+--
    ↪ ----------------------------+-----------+---------+-----------+-----------------------+---------------
    ↪
| IndexLookUp_6                 | 262400.00 | 262400  | root      |                       | time
    ↪ :620.513742ms, loops:258, cop_task: {num: 4, max: 5.530817ms, min: 1.51829ms, avg:
    ↪ 2.70883ms, p95: 5.530817ms, max_proc_keys: 2480, p95_proc_keys: 2480, tot_proc: 1ms,
    ↪ tot_wait: 1ms, rpc_num: 4, rpc_time: 10.816328ms, copr_cache_hit_rate: 0.75} |
    ↪                                  | 6.685169219970703 MB  | N/A  |
| ├─IndexFullScan_4(Build)      | 262400.00 | 262400  | cop[tikv] | table:t, index:a(a, c) |
    ↪ proc max:93ms, min:1ms, p80:93ms, p95:93ms, iters:275, tasks:4
    ↪
    ↪ | keep order:false, stats:pseudo | 1.7549400329589844 MB | N/A  |
| └─TableRowIDScan_5(Probe)     | 262400.00 | 0       | cop[tikv] | table:t               |
    ↪ time:0ns, loops:0
    ↪
    ↪ | keep order:false, stats:pseudo | N/A                  | N/A  |
+--
    ↪ ----------------------------+-----------+---------+-----------+-----------------------+---------------
    ↪
3 rows in set (0.62 sec)
```

执行结果的 execution info 列有 copr_cache_hit_ratio 信息，表示下推计算结果缓存的命中率。以上示例的 0.75 表示命中率大约是 75%。

查看 Grafana 监控面板

在 Grafana 监控中，tidb 命名空间下 distsql 子系统中可见 copr-cache 面板，该面板监控整个集群中下推计算结果缓存的命中次数、未命中次数和缓存丢弃次数。

11.2.0.9 垃圾回收 (GC)

11.2.0.9.1 GC 机制简介

TiDB 的事务的实现采用了 MVCC（多版本并发控制）机制，当新写入的数据覆盖旧的数据时，旧的数据不会被替换掉，而是与新写入的数据同时保留，并以时间戳来区分版本。Garbage Collection (GC) 的任务便是清理不再需要的旧数据。

整体流程

一个 TiDB 集群中会有一个 TiDB 实例被选举为 GC leader，GC 的运行由 GC leader 来控制。

GC 会被定期触发。每次 GC 时，首先，TiDB 会计算一个称为 safe point 的时间戳，接下来 TiDB 会在保证 safe point 之后的快照全部拥有正确数据的前提下，删除更早的过期数据。每一轮 GC 分为以下三个步骤：

1. “Resolve Locks” 阶段会对所有 Region 扫描 safe point 之前的锁，并清理这些锁。
2. “Delete Ranges” 阶段快速地删除由于 DROP TABLE/DROP INDEX 等操作产生的整区间的废弃数据。
3. “Do GC” 阶段每个 TiKV 节点将会各自扫描该节点上的数据，并对每一个 key 删除其不再需要的旧版本。

默认配置下，GC 每 10 分钟触发一次，每次 GC 会保留最近 10 分钟内的数据（即默认 GC life time 为 10 分钟，safe point 的计算方式为当前时间减去 GC life time）。如果一轮 GC 运行时间太久，那么在一轮 GC 完成之前，即使到了下一次触发 GC 的时间也不会开始下一轮 GC。另外，为了使持续时间较长的事务能在超过 GC life time 之后仍然可以正常运行，safe point 不会超过正在执行中的事务的开始时间 (start_ts)。

实现细节

Resolve Locks（清理锁）

TiDB 的事务是基于 Google Percolator 模型实现的，事务的提交是一个两阶段提交的过程。第一阶段完成时，所有涉及的 key 都会上锁，其中一个锁会被选为 Primary，其余的锁 (Secondary) 则会存储一个指向 Primary 的指针；第二阶段会将 Primary 锁所在的 key 加上一个 Write 记录，并去除锁。这里的 Write 记录就是历史上对该 key 进行写入或删除，或者该 key 上发生事务回滚的记录。Primary 锁被替换为何种 Write 记录标志着该事务提交成功与否。接下来，所有 Secondary 锁也会被依次替换。如果因为某些原因（如发生故障等），这些 Secondary 锁没有完成替换、残留了下来，那么也可以根据锁中的信息取找到 Primary，并根据 Primary 是否提交来判断整个事务是否提交。但是，如果 Primary 的信息在 GC 中被删除了，而该事务又存在未成功提交的 Secondary 锁，那么就永远无法得知该锁是否可以提交。这样，数据的正确性就无法保证。

Resolve Locks 这一步的任务即对 safe point 之前的锁进行清理。即如果一个锁对应的 Primary 已经提交，那么该锁也应该被提交；反之，则应该回滚。而如果 Primary 仍然是上锁的状态（没有提交也没有回滚），则应当将该事务视为超时失败而回滚。

Resolve Locks 有两种执行模式：

- LEGACY（默认模式）：由 GC leader 对所有的 Region 发送请求扫描过期的锁，并对扫到的锁查询 Primary 的状态，再发送请求对其进行提交或回滚。
- PHYSICAL：TiDB 绕过 Raft 层直接扫描每个 TiKV 节点上的数据。

> 警告：
>
> PHYSICAL模式（即启用 Green GC）目前是实验性功能，不建议在生产环境中使用。

你可以通过修改系统变量tidb_gc_scan_lock_mode 的值切换 Resolve Locks 的执行模式。

Delete Ranges（删除区间）

在执行 DROP TABLE/INDEX 等操作时，会有大量连续的数据被删除。如果对每个 key 都进行删除操作、再对每个 key 进行 GC 的话，那么执行效率和空间回收速度都可能非常的低下。事实上，这种时候 TiDB 并不会对每个 key 进行删除操作，而是将这些待删除的区间及删除操作的时间戳记录下来。Delete Ranges 会将这些时间戳在 safe point 之前的区间进行快速的物理删除。

Do GC（进行 GC 清理）

这一步即删除所有 key 的过期版本。为了保证 safe point 之后的任何时间戳都具有一致的快照，这一步删除 safe point 之前提交的数据，但是会对每个 key 保留 safe point 前的最后一次写入（除非最后一次写入是删除）。

在进行这一步时，TiDB 只需要将 safe point 发送给 PD，即可结束整轮 GC。TiKV 会自行检测到 safe point 发生了更新，会对当前节点上所有作为 Region leader 进行 GC。与此同时，GC leader 可以继续触发下一轮 GC。

> 注意：
>
> 从 TiDB 5.0 版本起，CENTRAL GC 模式（需要 TiDB 服务器发送 GC 请求到各个 Region）已经废弃，Do GC 这一步将只以 DISTRIBUTED GC 模式（从 TiDB 3.0 版起的默认模式）运行。

11.2.0.9.2 GC 配置

你可以通过以下系统变量进行 GC 配置：

- tidb_gc_enable
- tidb_gc_run_interval
- tidb_gc_life_time
- tidb_gc_concurrency
- tidb_gc_scan_lock_mode
- tidb_gc_max_wait_time

流控

TiDB 支持 GC 流控，可通过配置 gc.max-write-bytes-per-sec 限制 GC worker 每秒数据写入量，降低对正常请求的影响，0 为关闭该功能。该配置可通过 tikv-ctl 动态修改：

```
tikv-ctl --host=ip:port modify-tikv-config -n gc.max-write-bytes-per-sec -v 10MB
```

TiDB 5.0 引入的变化

在 TiDB 5.0 之前的版本中，GC 是通过系统表 mysql.tidb 进行配置的。从 TiDB 5.0 版本起，GC 仍然可以通过系统表 mysql.tidb 进行配置，但建议你使用系统变量进行配置，这样可以确保对配置的任何更改都能得到验证，防止造成异常行为 (#20655)。

TiDB 5.0 及之后的版本不再需要向各个 TiKV Region 都发送触发 GC 的请求，因此不再提供 CENTRAL GC 模式的支持，取而代之的是效率更高的 DISTRIBUTED GC 模式（自 TiDB 3.0 起的默认 GC 模式）。

如果要了解 TiDB 历史版本中 GC 配置的变化信息，请使用左侧导航栏中的“TIDB 版本选择器”切换到本文档的历史版本。

TiDB 6.1.0 引入的变化

在 TiDB 6.1.0 之前的版本中，TiDB 内部事务不会影响 GC safe point 推进。从 TiDB 6.1.0 版本起，计算 safe point 时会考虑内部事务的 startTS，从而解决内部事务因访问的数据被清理掉而导致失败的问题。带来的负面影响是如果内部事务运行时间过长，会导致 safe point 长时间不推进，进而会影响业务性能。

TiDB v6.1.0 引入了系统变量`tidb_gc_max_wait_time`控制活跃事务阻塞 GC safe point 推进的最长时间，超过该值后 GC safe point 会强制向后推进。

GC in Compaction Filter 机制

GC in Compaction Filter 机制是在分布式 GC 模式 (DISTRIBUTED GC mode) 的基础上，由 RocksDB 的 Compaction 过程来进行 GC，而不再使用一个单独的 GC worker 线程。这样做的好处是避免了 GC 引起的额外磁盘读取，以及避免清理掉的旧版本残留大量删除标记影响顺序扫描性能。可以由 TiKV 配置文件中的以下开关控制：

```
[gc]
enable-compaction-filter = true
```

该 GC 机制可通过在线配置变更开启：

```
show config where type = 'tikv' and name like '%enable-compaction-filter%';
```

```
+------+------------------+-----------------------------+-------+
| Type | Instance         | Name                        | Value |
+------+------------------+-----------------------------+-------+
| tikv | 172.16.5.37:20163 | gc.enable-compaction-filter | false |
| tikv | 172.16.5.36:20163 | gc.enable-compaction-filter | false |
| tikv | 172.16.5.35:20163 | gc.enable-compaction-filter | false |
+------+------------------+-----------------------------+-------+
```

```
set config tikv gc.enable-compaction-filter = true;
show config where type = 'tikv' and name like '%enable-compaction-filter%';
```

```
+------+------------------+-----------------------------+-------+
| Type | Instance         | Name                        | Value |
+------+------------------+-----------------------------+-------+
| tikv | 172.16.5.37:20163 | gc.enable-compaction-filter | true  |
| tikv | 172.16.5.36:20163 | gc.enable-compaction-filter | true  |
| tikv | 172.16.5.35:20163 | gc.enable-compaction-filter | true  |
+------+------------------+-----------------------------+-------+
```

11.3 SQL 性能调优

11.3.1 SQL 性能调优

SQL 是一种声明性语言。一条 SQL 语句描述的是最终结果应该如何，而非按顺序执行的步骤。TiDB 会优化 SQL 语句的执行，语义上允许以任何顺序执行查询的各部分，前提是能正确返回语句所描述的最终结果。

SQL 性能优化的过程，可以理解为 GPS 导航的过程。你提供地址后，GPS 软件利用各种统计信息（例如以前的行程、速度限制等元数据，以及实时交通信息）规划出一条最省时的路线。这与 TiDB 中的 SQL 性能优化过程相对应。

本章节包括以下文档，可帮助你更好地理解查询执行计划：

- 理解 TiDB 执行计划介绍如何使用 EXPLAIN 语句来理解 TiDB 是如何执行某个查询的。
- SQL 优化流程概览介绍 TiDB 可以使用的几种优化，以提高查询性能。
- 控制执行计划介绍如何控制执行计划的生成。TiDB 的执行计划非最优时，建议控制执行计划。

11.3.2 理解 TiDB 执行计划

11.3.2.1 TiDB 执行计划概览

> **注意：**
>
> 使用 MySQL 客户端连接到 TiDB 时，为避免输出结果在终端中换行，可先执行 `pager less -S` 命令。执行命令后，新的 EXPLAIN 的输出结果不再换行，可按右箭头 → 键水平滚动阅读输出结果。

使用 EXPLAIN 可查看 TiDB 执行某条语句时选用的执行计划。也就是说，TiDB 在考虑上数百或数千种可能的执行计划后，最终认定该执行计划消耗的资源最少、执行的速度最快。

EXPLAIN 示例如下：

```
CREATE TABLE t (id INT NOT NULL PRIMARY KEY auto_increment, a INT NOT NULL, pad1 VARCHAR(255),
    ↪ INDEX(a));
INSERT INTO t VALUES (1, 1, 'aaa'),(2,2, 'bbb');
EXPLAIN SELECT * FROM t WHERE a = 1;
```

返回的结果如下：

```
Query OK, 0 rows affected (0.96 sec)

Query OK, 2 rows affected (0.02 sec)
Records: 2  Duplicates: 0  Warnings: 0

+--
    ↪ ------------------------------+---------+-----------+---------------------+-------------------------------
    ↪
```

```
| id                            | estRows | task      | access object       | operator info
    ↪                                 |
+--
    ↪ ------------------------------+---------+-----------+---------------------+------------------------------
    ↪
| IndexLookUp_10                | 10.00   | root      |                     |
    ↪                                                 |
| ├─IndexRangeScan_8(Build)     | 10.00   | cop[tikv] | table:t, index:a(a) | range:[1,1], keep
    ↪ order:false, stats:pseudo |
| └─TableRowIDScan_9(Probe)     | 10.00   | cop[tikv] | table:t             | keep order:false,
    ↪ stats:pseudo              |
+--
    ↪ ------------------------------+---------+-----------+---------------------+------------------------------
    ↪
3 rows in set (0.00 sec)
```

EXPLAIN 实际不会执行查询。EXPLAIN ANALYZE 可用于实际执行查询并显示执行计划。如果 TiDB 所选的执行计划非最优，可用 EXPLAIN 或 EXPLAIN ANALYZE 来进行诊断。有关 EXPLAIN 用法的详细内容，参阅以下文档：

- MPP 模式查询的执行计划
- 索引查询的执行计划
- Join 查询的执行计划
- 子查询的执行计划
- 聚合查询的执行计划
- 视图查询的执行计划
- 分区查询的执行计划
- 索引合并查询的执行计划

11.3.2.1.1 解读 EXPLAIN 的返回结果

EXPLAIN 的返回结果包含以下字段：

- id 为算子名，或执行 SQL 语句需要执行的子任务。详见算子简介。
- estRows 为显示 TiDB 预计会处理的行数。该预估数可能基于字典信息（例如访问方法基于主键或唯一键），或基于 CMSketch 或直方图等统计信息估算而来。
- task 显示算子在执行语句时的所在位置。详见Task 简介。
- access-object 显示被访问的表、分区和索引。显示的索引为部分索引。以上示例中 TiDB 使用了 a 列的索引。尤其是在有组合索引的情况下，该字段显示的信息很有参考意义。
- operator info 显示访问表、分区和索引的其他信息。详见operator info 结果。

> **注意：**
>
> 在执行计划返回结果中，自 v6.4.0 版本起，特定算子（即 IndexJoin 和 Apply 算子的 Probe 端所有子节点）的 estRows 字段意义与 v6.4.0 版本之前的有所不同。

在 v6.4.0 之前，estRows 表示对于 Build 端子节点的每一行，Probe 端预计会处理的行数。自 v6.4.0 起，estRows 表示 Probe 端预计会处理的总行数。由于 EXPLAIN ANALYZE 中展示的实际行数（actRows 列）表示的是总行数，v6.4.0 起这些算子 estRows 的含义与 actRows 列的含义保持一致。

例如：

```
CREATE TABLE t1(a INT, b INT);
CREATE TABLE t2(a INT, b INT, INDEX ia(a));
EXPLAIN SELECT /*+ INL_JOIN(t2) */ * FROM t1 JOIN t2 ON t1.a = t2.a;
EXPLAIN SELECT (SELECT a FROM t2 WHERE t2.a = t1.b LIMIT 1) FROM t1;
```

```
-- v6.4.0 之前：
+--
    ↪ -------------------------------+----------+-----------+----------------------+-----------------
    ↪
| id                              | estRows  | task      | access object         |
    ↪ operator info
    ↪
    ↪ |
+--
    ↪ -------------------------------+----------+-----------+----------------------+-----------------
    ↪
| IndexJoin_12                    | 12487.50 | root      |                       |
    ↪ inner join, inner:IndexLookUp_11, outer key:test.t1.a, inner key:test.t2.a,
    ↪  equal cond:eq(test.t1.a, test.t2.a) |
| ├─TableReader_24(Build)         | 9990.00  | root      |                       |
    ↪  data:Selection_23
    ↪
    ↪ |
| │ └─Selection_23                | 9990.00  | cop[tikv] |                       |
    ↪  not(isnull(test.t1.a))
    ↪
    ↪ |
| │   └─TableFullScan_22          | 10000.00 | cop[tikv] | table:t1              |
    ↪  keep order:false, stats:pseudo
    ↪
    ↪ |
| └─IndexLookUp_11(Probe)         | 1.25     | root      |                       |
    ↪
    ↪
    ↪ |
|   ├─Selection_10(Build)         | 1.25     | cop[tikv] |                       |
    ↪  not(isnull(test.t2.a))
    ↪
```

```
|    ↪ |
|   | └─IndexRangeScan_8          | 1.25     | cop[tikv] | table:t2, index:ia(a) |
|    ↪  range: decided by [eq(test.t2.a, test.t1.a)], keep order:false, stats:
|    ↪ pseudo                                        |
|   └─TableRowIDScan_9(Probe)     | 1.25     | cop[tikv] | table:t2              |
|    ↪  keep order:false, stats:pseudo
|    ↪
|    ↪ |
+--
|    ↪ ------------------------------+----------+-----------+----------------------+------------------
|    ↪
+--
|    ↪ ------------------------------+----------+-----------+----------------------+------------------
|    ↪
| id                            | estRows  | task      | access object         |
|    ↪ operator info
|    ↪                                                                  |
+--
|    ↪ ------------------------------+----------+-----------+----------------------+------------------
|    ↪
| Projection_12                 | 10000.00 | root      |                       |
|    ↪ test.t2.a
|    ↪                                                                        |
| └─Apply_14                     | 10000.00 | root      |                       |
|    ↪  CARTESIAN left outer join
|    ↪                                                      |
|   ├─TableReader_16(Build)      | 10000.00 | root      |                       |
|    ↪  data:TableFullScan_15
|    ↪                                                           |
|   | └─TableFullScan_15         | 10000.00 | cop[tikv] | table:t1              |
|    ↪  keep order:false, stats:pseudo
|    ↪                                                  |
|   └─Limit_17(Probe)            | 1.00     | root      |                       |
|    ↪  offset:0, count:1
|    ↪                                                               |
|     └─IndexReader_21           | 1.00     | root      |                       |
|    ↪  index:Limit_20
|    ↪                                                                  |
|       └─Limit_20               | 1.00     | cop[tikv] |                       |
|    ↪  offset:0, count:1
|    ↪                                                               |
|         └─IndexRangeScan_19    | 1.00     | cop[tikv] | table:t2, index:ia(a) |
|    ↪  range: decided by [eq(test.t2.a, test.t1.b)], keep order:false, stats:
```

```
    ↪ pseudo |
+--
    ↪ ------------------------------+----------+-----------+----------------------+------------------
    ↪
-- 自 v6.4.0 起:

-- 可以发现 `IndexLookUp_11`、`Selection_10`、`IndexRangeScan_8` 和 `
    ↪ TableRowIDScan_9` 在 `estRows` 列显示的行数与 v6.4.0 以前的不同
+--
    ↪ ------------------------------+----------+-----------+----------------------+------------------
    ↪
| id                            | estRows  | task      | access object         |
    ↪ operator info
    ↪
    ↪ |
+--
    ↪ ------------------------------+----------+-----------+----------------------+------------------
    ↪
| IndexJoin_12                  | 12487.50 | root      |                       |
    ↪ inner join, inner:IndexLookUp_11, outer key:test.t1.a, inner key:test.t2.a,
    ↪  equal cond:eq(test.t1.a, test.t2.a) |
| ├─TableReader_24(Build)       | 9990.00  | root      |                       |
    ↪  data:Selection_23
    ↪
    ↪ |
| │ └─Selection_23              | 9990.00  | cop[tikv] |                       |
    ↪  not(isnull(test.t1.a))
    ↪
    ↪ |
| │   └─TableFullScan_22        | 10000.00 | cop[tikv] | table:t1              |
    ↪  keep order:false, stats:pseudo
    ↪
    ↪ |
| └─IndexLookUp_11(Probe)       | 12487.50 | root      |                       |
    ↪
    ↪
    ↪ |
|   ├─Selection_10(Build)       | 12487.50 | cop[tikv] |                       |
    ↪  not(isnull(test.t2.a))
    ↪
    ↪ |
|   │ └─IndexRangeScan_8        | 12500.00 | cop[tikv] | table:t2, index:ia(a) |
```

```
    ↪  range: decided by [eq(test.t2.a, test.t1.a)], keep order:false, stats:
    ↪ pseudo                                      |
|   └─TableRowIDScan_9(Probe)     | 12487.50 | cop[tikv] | table:t2              |
    ↪  keep order:false, stats:pseudo
    ↪
    ↪ |
+--
    ↪ ------------------------------+----------+-----------+----------------------+------------------
    ↪

-- 可以发现 `Limit_17`、`IndexReader_21`、`Limit_20` 和 `IndexRangeScan_19` 在 `
    ↪ estRows` 列显示的行数与 v6.4.0 以前的不同
+--
    ↪ ------------------------------+----------+-----------+----------------------+------------------
    ↪
| id                              | estRows  | task      | access object         |
    ↪ operator info
    ↪                                                                  |
+--
    ↪ ------------------------------+----------+-----------+----------------------+------------------
    ↪
| Projection_12                   | 10000.00 | root      |                       |
    ↪ test.t2.a
    ↪                                                                      |
| └─Apply_14                       | 10000.00 | root      |                       |
    ↪  CARTESIAN left outer join
    ↪                                                    |
|   ├─TableReader_16(Build)       | 10000.00 | root      |                       |
    ↪  data:TableFullScan_15
    ↪                                                        |
|   │ └─TableFullScan_15          | 10000.00 | cop[tikv] | table:t1              |
    ↪  keep order:false, stats:pseudo
    ↪                                                  |
|   └─Limit_17(Probe)             | 10000.00 | root      |                       |
    ↪  offset:0, count:1
    ↪                                                            |
|     └─IndexReader_21            | 10000.00 | root      |                       |
    ↪  index:Limit_20
    ↪                                                               |
|       └─Limit_20                | 10000.00 | cop[tikv] |                       |
    ↪  offset:0, count:1
    ↪                                                            |
|         └─IndexRangeScan_19     | 10000.00 | cop[tikv] | table:t2, index:ia(a) |
```

```
    ↪  range: decided by [eq(test.t2.a, test.t1.b)], keep order:false, stats:
    ↪ pseudo |
+--
    ↪ ------------------------------+---------+----------+----------------------+------------------
    ↪
```

算子简介

算子是为返回查询结果而执行的特定步骤。真正执行扫表（读盘或者读 TiKV Block Cache）操作的算子有如下几类：

- TableFullScan：全表扫描。
- TableRangeScan：带有范围的表数据扫描。
- TableRowIDScan：根据上层传递下来的 RowID 扫描表数据。时常在索引读操作后检索符合条件的行。
- IndexFullScan：另一种“全表扫描”，扫的是索引数据，不是表数据。
- IndexRangeScan：带有范围的索引数据扫描操作。

TiDB 会汇聚 TiKV/TiFlash 上扫描的数据或者计算结果，这种“数据汇聚”算子目前有如下几类：

- TableReader：将 TiKV 上底层扫表算子 TableFullScan 或 TableRangeScan 得到的数据进行汇总。
- IndexReader：将 TiKV 上底层扫表算子 IndexFullScan 或 IndexRangeScan 得到的数据进行汇总。
- IndexLookUp：先汇总 Build 端 TiKV 扫描上来的 RowID，再去 Probe 端上根据这些 RowID 精确地读取 TiKV 上的数据。Build 端是 IndexFullScan 或 IndexRangeScan 类型的算子，Probe 端是 TableRowIDScan 类型的算子。
- IndexMerge：和 IndexLookupReader 类似，可以看做是它的扩展，可以同时读取多个索引的数据，有多个 Build 端，一个 Probe 端。执行过程也很类似，先汇总所有 Build 端 TiKV 扫描上来的 RowID，再去 Probe 端上根据这些 RowID 精确地读取 TiKV 上的数据。Build 端是 IndexFullScan 或 IndexRangeScan 类型的算子，Probe 端是 TableRowIDScan 类型的算子。

算子的执行顺序

算子的结构是树状的，但在查询执行过程中，并不严格要求子节点任务在父节点之前完成。TiDB 支持同一查询内的并行处理，即子节点“流入”父节点。父节点、子节点和同级节点可能并行执行查询的一部分。

在以上示例中，├─IndexRangeScan_8(Build) 算子为 a(a) 索引所匹配的行查找内部 RowID。└─TableRowIDScan_9 ↪ (Probe) 算子随后从表中检索这些行。

Build 总是先于 Probe 执行，并且 Build 总是出现在 Probe 前面。即如果一个算子有多个子节点，子节点 ID 后面有 Build 关键字的算子总是先于有 Probe 关键字的算子执行。TiDB 在展现执行计划的时候，Build 端总是第一个出现，接着才是 Probe 端。

范围查询

在 WHERE/HAVING/ON 条件中，TiDB 优化器会分析主键或索引键的查询返回。如数字、日期类型的比较符，如大于、小于、等于以及大于等于、小于等于，字符类型的 LIKE 符号等。

若要使用索引，条件必须是“Sargable”(Search ARGument ABLE) 的。例如条件 YEAR(date_column)< 1992 不能使用索引，但 date_column < '1992-01-01 就可以使用索引。

推荐使用同一类型的数据以及同一类型的字符串和排序规则进行比较，以避免引入额外的 cast 操作而导致不能利用索引。

可以在范围查询条件中使用 AND（求交集）和 OR（求并集）进行组合。对于多维组合索引，可以对多个列使用条件。例如对组合索引 (a, b, c)：

- 当 a 为等值查询时，可以继续求 b 的查询范围。
- 当 b 也为等值查询时，可以继续求 c 的查询范围。
- 反之，如果 a 为非等值查询，则只能求 a 的范围。

Task 简介

目前 TiDB 的计算任务分为两种不同的 task：cop task 和 root task。Cop task 是指使用 TiKV 中的 Coprocessor 执行的计算任务，root task 是指在 TiDB 中执行的计算任务。

SQL 优化的目标之一是将计算尽可能地下推到 TiKV 中执行。TiKV 中的 Coprocessor 能支持大部分 SQL 内建函数（包括聚合函数和标量函数）、SQL LIMIT 操作、索引扫描和表扫描。

operator info 结果

EXPLAIN 返回结果中 operator info 列可显示诸如条件下推等信息。本文以上示例中，operator info 结果各字段解释如下：

- range: [1,1] 表示查询的 WHERE 字句 (a = 1) 被下推到了 TiKV，对应的 task 为 cop[tikv]。
- keep order:false 表示该查询的语义不需要 TiKV 按顺序返回结果。如果查询指定了排序（例如 SELECT * FROM t WHERE a = 1 ORDER BY id），该字段的返回结果为 keep order:true。
- stats:pseudo 表示 estRows 显示的预估数可能不准确。TiDB 定期在后台更新统计信息。也可以通过执行 ANALYZE TABLE t 来手动更新统计信息。

EXPLAIN 执行后，不同算子返回不同的信息。你可以使用 Optimizer Hints 来控制优化器的行为，以此控制物理算子的选择。例如 /*+ HASH_JOIN(t1, t2)*/ 表示优化器将使用 Hash Join 算法。详细内容见Optimizer Hints。

11.3.2.1.2 算子相关的系统变量

TiDB 在 MySQL 的基础上，定义了一些专用的系统变量和语法用来优化性能。其中一些系统变量和具体的算子相关，比如算子的并发度，算子的内存使用上限，是否允许使用分区表等。这些都可以通过系统变量进行控制，从而影响各个算子执行的效率。

如果读者想要详细了解所有的系统变量及其使用规则，可以参见系统变量和语法。

11.3.2.1.3 另请参阅

- EXPLAIN
- EXPLAIN ANALYZE
- ANALYZE TABLE
- TRACE
- TiDB in Action
- 系统变量

11.3.2.2 使用 EXPLAIN 解读执行计划

SQL 是一种声明性语言，因此用户无法根据 SQL 语句直接判断一条查询的执行是否有效率。用户首先要使用EXPLAIN 语句查看当前的执行计划。

以 bikeshare 数据库示例（英文）中的一个 SQL 语句为例，该语句统计了 2017 年 7 月 1 日的行程次数：

```
EXPLAIN SELECT count(*) FROM trips WHERE start_date BETWEEN '2017-07-01 00:00:00' AND '2017-07-01
    ↪  23:59:59';
```

```
+--
    ↪ ---------------------------+----------+-----------+---------------+--------------------------------
    ↪
| id                           | estRows  | task      | access object | operator info
    ↪
    ↪ |
+--
    ↪ ---------------------------+----------+-----------+---------------+--------------------------------
    ↪
| StreamAgg_20                 | 1.00     | root      |               | funcs:count(Column#13)->
    ↪ Column#11
    ↪                                                                                                  |
| └─TableReader_21             | 1.00     | root      |               | data:StreamAgg_9
    ↪
    ↪ |
|   └─StreamAgg_9              | 1.00     | cop[tikv] |               | funcs:count(1)->Column
    ↪ #13
    ↪
    ↪ |
|     └─Selection_19           | 250.00   | cop[tikv] |               | ge(bikeshare.trips.
    ↪ start_date, 2017-07-01 00:00:00.000000), le(bikeshare.trips.start_date, 2017-07-01
    ↪ 23:59:59.000000) |
|       └─TableFullScan_18     | 10000.00 | cop[tikv] | table:trips   | keep order:false, stats:
    ↪ pseudo
    ↪                                                                                                  |
+--
    ↪ ---------------------------+----------+-----------+---------------+--------------------------------
    ↪
5 rows in set (0.00 sec)
```

以上是该查询的执行计划结果。从 └─TableFullScan_18 算子开始向上看，查询的执行过程如下（非最佳执行计划）：

1. Coprocessor (TiKV) 读取整张 trips 表的数据，作为一次 TableFullScan 操作，再将读取到的数据传递给 Selection_19 算子。Selection_19 算子仍在 TiKV 内。
2. Selection_19 算子根据谓词 WHERE start_date BETWEEN .. 进行数据过滤。预计大约有 250 行数据满足该过滤条件（基于统计信息以及算子的执行逻辑估算而来）。└─TableFullScan_18 算子显示 stats:pseudo，

表示该表没有实际统计信息，执行 ANALYZE TABLE trips 收集统计信息后，预计的估算的数字会更加准确。

3. COUNT 函数随后应用于满足过滤条件的行，这一过程也是在 TiKV (cop[tikv]) 中的 StreamAgg_9 算子内完成的。TiKV coprocessor 能执行一些 MySQL 内置函数，COUNT 是其中之一。

4. StreamAgg_9 算子执行的结果会被传递给 TableReader_21 算子（位于 TiDB 进程中，即 root 任务）。执行计划中，TableReader_21 算子的 estRows 为 1，表示该算子将从每个访问的 TiKV Region 接收一行数据。这一请求过程的详情，可参阅EXPLAIN ANALYZE。

5. StreamAgg_20 算子随后对 └─TableReader_21 算子传来的每行数据计算 COUNT 函数的结果。StreamAgg_20 是根算子，会将结果返回给客户端。

> **注意：**
>
> 要查看 TiDB 中某张表的 Region 信息，可执行SHOW TABLE REGIONS 语句。

11.3.2.2.1 评估当前的性能

EXPLAIN 语句只返回查询的执行计划，并不执行该查询。若要获取实际的执行时间，可执行该查询，或使用 EXPLAIN ANALYZE 语句：

```
EXPLAIN ANALYZE SELECT count(*) FROM trips WHERE start_date BETWEEN '2017-07-01 00:00:00' AND '
    ↪ 2017-07-01 23:59:59';
```

```
+--
    ↪ ----------------------------+----------+----------+-----------+---------------+-------------------------
    ↪
| id                           | estRows  | actRows  | task      | access object | execution info
    ↪
    ↪
    ↪ | operator info
    ↪
    ↪ | memory    | disk |
+--
    ↪ ----------------------------+----------+----------+-----------+---------------+-------------------------
    ↪
| StreamAgg_20                 | 1.00     | 1        | root      |               | time
    ↪ :1.031417203s, loops:2
    ↪
    ↪ | funcs:count(Column#13)->Column#11
    ↪                                                                                 |
    ↪ 632 Bytes | N/A  |
| └─TableReader_21             | 1.00     | 56       | root      |               | time
    ↪ :1.031408123s, loops:2, cop_task: {num: 56, max: 782.147269ms, min: 5.759953ms, avg:
```

```
    ↪ 252.005927ms, p95: 609.294603ms, max_proc_keys: 910371, p95_proc_keys: 704775, tot_proc:
    ↪ 11.524s, tot_wait: 580ms, rpc_num: 56, rpc_time: 14.111932641s} | data:StreamAgg_9
    ↪
    ↪ | 328 Bytes | N/A  |
|   └─StreamAgg_9              | 1.00     | 56       | cop[tikv] |               | proc max:640
    ↪ ms, min:8ms, p80:276ms, p95:480ms, iters:18695, tasks:56
    ↪
    ↪ | funcs:count(1)->Column#13
    ↪
    ↪ | N/A       | N/A  |
|     └─Selection_19           | 250.00   | 11409    | cop[tikv] |               | proc max:640
    ↪ ms, min:8ms, p80:276ms, p95:476ms, iters:18695, tasks:56
    ↪
    ↪ | ge(bikeshare.trips.start_date, 2017-07-01 00:00:00.000000), le(bikeshare.trips.
    ↪ start_date, 2017-07-01 23:59:59.000000) | N/A       | N/A  |
|       └─TableFullScan_18     | 10000.00 | 19117643 | cop[tikv] | table:trips   | proc max:612
    ↪ ms, min:8ms, p80:248ms, p95:460ms, iters:18695, tasks:56
    ↪
    ↪ | keep order:false, stats:pseudo
    ↪                                                                                |
    ↪  N/A       | N/A  |
+--
    ↪ ---------------------------+----------+----------+-----------+---------------+--------------------------
    ↪
5 rows in set (1.03 sec)
```

执行以上示例查询耗时 1.03 秒，说明执行性能较为理想。

以上 EXPLAIN ANALYZE 的结果中，actRows 表明一些 estRows 预估数不准确（预估返回 10000 行数据但实际返回 19117643 行）。└─TableFullScan_18 算子的 operator info 列 (stats:pseudo) 信息也表明该算子的预估数不准确。

如果先执行ANALYZE TABLE 再执行 EXPLAIN ANALYZE，预估数与实际数会更接近：

```
ANALYZE TABLE trips;
EXPLAIN ANALYZE SELECT count(*) FROM trips WHERE start_date BETWEEN '2017-07-01 00:00:00' AND '
    ↪ 2017-07-01 23:59:59';
```

```
Query OK, 0 rows affected (10.22 sec)

+--
    ↪ ---------------------------+-------------+----------+-----------+---------------+--------------------------
    ↪
| id                           | estRows     | actRows  | task      | access object | execution
    ↪ info
    ↪
    ↪ | operator info
```

```
    ↪
    ↪ | memory    | disk |
+--
    ↪ ---------------------------+-------------+----------+-----------+---------------+-----------------------
    ↪
| StreamAgg_20                | 1.00        | 1        | root      |               | time
    ↪ :926.393612ms, loops:2
    ↪
    ↪ | funcs:count(Column#13)->Column#11
    ↪                                                                                 |
    ↪ 632 Bytes | N/A  |
| └─TableReader_21            | 1.00        | 56       | root      |               | time
    ↪ :926.384792ms, loops:2, cop_task: {num: 56, max: 850.94424ms, min: 6.042079ms, avg:
    ↪ 234.987725ms, p95: 495.474806ms, max_proc_keys: 910371, p95_proc_keys: 704775, tot_proc:
    ↪ 10.656s, tot_wait: 904ms, rpc_num: 56, rpc_time: 13.158911952s} | data:StreamAgg_9
    ↪
    ↪ | 328 Bytes | N/A  |
|   └─StreamAgg_9             | 1.00        | 56       | cop[tikv] |               | proc max
    ↪ :592ms, min:4ms, p80:244ms, p95:480ms, iters:18695, tasks:56
    ↪
    ↪ | funcs:count(1)->Column#13
    ↪
    ↪ | N/A       | N/A  |
|     └─Selection_19          | 432.89      | 11409    | cop[tikv] |               | proc max
    ↪ :592ms, min:4ms, p80:244ms, p95:480ms, iters:18695, tasks:56
    ↪
    ↪ | ge(bikeshare.trips.start_date, 2017-07-01 00:00:00.000000), le(bikeshare.trips.
    ↪ start_date, 2017-07-01 23:59:59.000000) | N/A       | N/A  |
|       └─TableFullScan_18    | 19117643.00 | 19117643 | cop[tikv] | table:trips   | proc max
    ↪ :564ms, min:4ms, p80:228ms, p95:456ms, iters:18695, tasks:56
    ↪
    ↪ | keep order:false
    ↪
    ↪ | N/A       | N/A  |
+--
    ↪ ---------------------------+-------------+----------+-----------+---------------+-----------------------
    ↪
5 rows in set (0.93 sec)
```

执行 ANALYZE TABLE 后，可以看到 └─TableFullScan_18 算子的预估行数是准确的，└─Selection_19 算子的预估行数也更接近实际行数。以上两个示例中的执行计划（即 TiDB 执行查询所使用的一组算子）未改变，但过时的统计信息常常导致 TiDB 选择到非最优的执行计划。

除 ANALYZE TABLE 外，达到tidb_auto_analyze_ratio 阈值后，TiDB 会自动在后台重新生成统计数据。若要查看 TiDB 有多接近该阈值（即 TiDB 判断统计数据有多健康），可执行SHOW STATS_HEALTHY 语句。

```
SHOW STATS_HEALTHY;
```

```
+-----------+------------+----------------+---------+
| Db_name   | Table_name | Partition_name | Healthy |
+-----------+------------+----------------+---------+
| bikeshare | trips      |                |     100 |
+-----------+------------+----------------+---------+
1 row in set (0.00 sec)
```

11.3.2.2.2 确定优化方案

当前执行计划是有效率的：

- 大部分任务是在 TiKV 内处理的，需要通过网络传输给 TiDB 处理的仅有 56 行数据，每行都满足过滤条件，而且都很短。
- 在 TiDB (StreamAgg_20) 中和在 TiKV (└─StreamAgg_9) 中汇总行数都使用了 Stream Aggregate，该算法在内存使用方面很有效率。

当前执行计划存在的最大问题在于谓词 `start_date BETWEEN '2017-07-01 00:00:00' AND '2017-07-01 23:59:59'` 并未立即生效，先是 TableFullScan 算子读取所有行数据，然后才进行过滤选择。可以在 `SHOW CREATE TABLE trips` 的返回结果中找出问题原因：

```
SHOW CREATE TABLE trips\G
```

```
*************************** 1. row ***************************
       Table: trips
Create Table: CREATE TABLE `trips` (
  `trip_id` bigint(20) NOT NULL AUTO_INCREMENT,
  `duration` int(11) NOT NULL,
  `start_date` datetime DEFAULT NULL,
  `end_date` datetime DEFAULT NULL,
  `start_station_number` int(11) DEFAULT NULL,
  `start_station` varchar(255) DEFAULT NULL,
  `end_station_number` int(11) DEFAULT NULL,
  `end_station` varchar(255) DEFAULT NULL,
  `bike_number` varchar(255) DEFAULT NULL,
  `member_type` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`trip_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin AUTO_INCREMENT=20477318
1 row in set (0.00 sec)
```

以上返回结果显示，`start_date` 列没有索引。要将该谓词下推到 index reader 算子，还需要一个索引。添加索引如下：

```
ALTER TABLE trips ADD INDEX (start_date);
```

```
Query OK, 0 rows affected (2 min 10.23 sec)
```

> **注意：**
>
> 你可通过执行ADMIN SHOW DDL JOBS 语句来查看 DDL 任务的进度。TiDB 中的默认值的设置较为保守，因此添加索引不会对生产环境下的负载造成太大影响。测试环境下，可以考虑调大tidb_ddl_reorg_batch_size 和tidb_ddl_reorg_worker_cnt 的值。在参照系统上，将批处理大小设为 10240，将 worker count 并发度设置为 32，该系统可获得 10 倍的性能提升（较之使用默认值）。

添加索引后，可以使用 EXPLAIN 重复该查询。在以下返回结果中，可见 TiDB 选择了新的执行计划，而且不再使用 TableFullScan 和 Selection 算子。

```
EXPLAIN SELECT count(*) FROM trips WHERE start_date BETWEEN '2017-07-01 00:00:00' AND '2017-07-01
    ↪  23:59:59';
```

```
+--
    ↪ --------------------------+---------+-----------+-----------------------------------------+-----------
    ↪
| id                          | estRows | task      | access object                           |
    ↪  operator info                                                    |
+--
    ↪ --------------------------+---------+-----------+-----------------------------------------+-----------
    ↪
| StreamAgg_17                | 1.00    | root      |                                         |
    ↪  funcs:count(Column#13)->Column#11                                |
| └─IndexReader_18            | 1.00    | root      |
    ↪ | index:StreamAgg_9                                               |
|   └─StreamAgg_9             | 1.00    | cop[tikv] |
    ↪ | funcs:count(1)->Column#13                                       |
|     └─IndexRangeScan_16     | 8471.88 | cop[tikv] | table:trips, index:start_date(start_date)
    ↪ | range:[2017-07-01 00:00:00,2017-07-01 23:59:59], keep order:false |
+--
    ↪ --------------------------+---------+-----------+-----------------------------------------+-----------
    ↪
4 rows in set (0.00 sec)
```

若要比较实际的执行时间，可再次使用 EXPLAIN ANALYZE 语句：

```
EXPLAIN ANALYZE SELECT count(*) FROM trips WHERE start_date BETWEEN '2017-07-01 00:00:00' AND '
    ↪ 2017-07-01 23:59:59';
```

```
+--
    ↪ -------------------------+---------+---------+-----------+-----------------------------------+-
    ↪
| id                          | estRows | actRows | task      | access object
    ↪                             | execution info
    ↪
    ↪ | operator info                                                | memory    | disk |
+--
    ↪ -------------------------+---------+---------+-----------+-----------------------------------+-
    ↪
| StreamAgg_17                | 1.00    | 1       | root      |
    ↪                                          | time:4.516728ms, loops:2
    ↪                                                                                              |
    ↪  funcs:count(Column#13)->Column#11                                  | 372 Bytes | N/A  |
| └─IndexReader_18            | 1.00    | 1       | root      |
    ↪                                          | time:4.514278ms, loops:2, cop_task: {num: 1,
    ↪ max:4.462288ms, proc_keys: 11409, rpc_num: 1, rpc_time: 4.457148ms} | index:StreamAgg_9
    ↪                                               | 238 Bytes | N/A  |
|   └─StreamAgg_9             | 1.00    | 1       | cop[tikv] |
    ↪                                          | time:4ms, loops:12
    ↪
    ↪ | funcs:count(1)->Column#13                                        | N/A       | N/A  |
|     └─IndexRangeScan_16     | 8471.88 | 11409   | cop[tikv] | table:trips, index:start_date(
    ↪ start_date) | time:4ms, loops:12
    ↪
    ↪ | range:[2017-07-01 00:00:00,2017-07-01 23:59:59], keep order:false | N/A       | N/A  |
+--
    ↪ -------------------------+---------+---------+-----------+-----------------------------------+-
    ↪
4 rows in set (0.00 sec)
```

从以上结果可看出，查询时间已从 1.03 秒减少到 0.0 秒。

> **注意：**
>
> 以上示例另一个可用的优化方案是coprocessor cache。如果你无法添加索引，可考虑开启 coprocessor cache 功能。开启后，只要算子上次执行以来 Region 未作更改，TiKV 将从缓存中返回值。这也有助于减少 TableFullScan 和 Selection 算子的大部分运算成本。

11.3.2.3 用 EXPLAIN 查看 MPP 模式查询的执行计划

TiDB 支持使用MPP 模式来执行查询。在 MPP 执行模式下，SQL 优化器会生成 MPP 的执行计划。注意 MPP 模式仅对有TiFlash 副本的表生效。

本文档使用的示例数据如下：

```
CREATE TABLE t1 (id int, value int);
INSERT INTO t1 values(1,2),(2,3),(1,3);
ALTER TABLE t1 set tiflash replica 1;
ANALYZE TABLE t1;
SET tidb_allow_mpp = 1;
```

11.3.2.3.1 MPP 查询片段和 MPP 任务

在 MPP 模式下，一个查询在逻辑上会被切分为多个 MPP 查询片段 (query fragment)。示例如下：

```
EXPLAIN SELECT COUNT(*) FROM t1 GROUP BY id;
```

这个查询在 MPP 模式下会包含两个查询片段，一个为一阶段聚合，一个为二阶段聚合（最终聚合）。在查询执行的时候每个查询片段都会被实例化为一个或者多个 MPP 任务。

11.3.2.3.2 Exchange 算子

MPP 查询的执行计划中有两个 MPP 特有的 Exchange 算子，分别为 ExchangeReceiver 和 ExchangeSender。ExchangeReceiver 表示从下游查询片段读取数据，ExchangeSender 表示下游查询片段向上游查询片段发送数据。在 MPP 执行模式下，每个 MPP 查询片段的根算子均为 ExchangeSender 算子，即每个查询片段以 ExchangeSender 为界进行划分。一个简单的 MPP 计划如下：

```
EXPLAIN SELECT COUNT(*) FROM t1 GROUP BY id;
```

```
+--
    ↪ ---------------------------------+---------+-------------------+---------------+-------------------------
    ↪
| id                                | estRows | task              | access object | operator
    ↪ info                                    |
+--
    ↪ ---------------------------------+---------+-------------------+---------------+-------------------------
    ↪
| TableReader_31                    | 2.00    | root              |               | data:
    ↪ ExchangeSender_30                              |
| └─ExchangeSender_30               | 2.00    | batchCop[tiflash] |               |
    ↪ ExchangeType: PassThrough                      |
|   └─Projection_26                 | 2.00    | batchCop[tiflash] |               | Column#4
    ↪                                                |
|     └─HashAgg_27                  | 2.00    | batchCop[tiflash] |               | group by:
    ↪ test.t1.id, funcs:sum(Column#7)->Column#4 |
|       └─ExchangeReceiver_29       | 2.00    | batchCop[tiflash] |               |
    ↪                                                |
|         └─ExchangeSender_28       | 2.00    | batchCop[tiflash] |               |
    ↪ ExchangeType: HashPartition, Hash Cols: test.t1.id |
```

```
|           └─HashAgg_9              | 2.00    | batchCop[tiflash] |              | group by:
    ↪ test.t1.id, funcs:count(1)->Column#7      |
|             └─TableFullScan_25     | 3.00    | batchCop[tiflash] | table:t1     | keep order:
    ↪ false                                     |
+--
    ↪ ---------------------------------+---------+-------------------+--------------+---------------------
    ↪
```

以上执行计划中有两个查询片段：

- `[TableFullScan_25, HashAgg_9, ExchangeSender_28]` 为第一个查询片段，其主要完成一阶段聚合的计算。
- `[ExchangeReceiver_29, HashAgg_27, Projection_26, ExchangeSender_30]` 为第二个查询片段，其主要完成二阶段聚合的计算。

ExchangeSender 算子的 `operator info` 列输出了 ExchangeType 信息。目前有以下三种 ExchangeType：

- HashPartition：ExchangeSender 把数据按 Hash 值进行分区之后分发给上游的 MPP 任务的 ExchangeReceiver 算子，通常在 Hash Aggregation 以及 Shuffle Hash Join 算法中使用。
- Broadcast：ExchangeSender 通过广播的方式把数据分发给上游的 MPP 任务，通常在 Broadcast Join 中使用。
- PassThrough：ExchangeSender 把数据分发给上游的 MPP Task，与 Broadcast 的区别是此时上游有且仅有一个 MPP 任务，通常用于向 TiDB 返回数据。

上述例子中 ExchangeSender 的 ExchangeType 为 HashPartition 以及 PassThrough，分别对应于 Hash Aggregation 运算以及向 TiDB 返回数据。

另外一个典型的 MPP 应用为 join 运算。TiDB MPP 支持两种类型的 join，分别为：

- Shuffle Hash Join：join 的 input 通过 HashPartition 的方式 shuffle 数据，上游的 MPP 任务进行分区内的 join。
- Broadcast Join：join 中的小表以 Broadcast 的方式把数据广播到各个节点，各个节点各自进行 join。

典型的 Shuffle Hash Join 执行计划如下：

```
SET tidb_broadcast_join_threshold_count=0; SET tidb_broadcast_join_threshold_size=0; EXPLAIN
    ↪ SELECT COUNT(*) FROM t1 a JOIN t1 b ON a.id = b.id;
```

```
+--
    ↪ ---------------------------------------+---------+-------------+---------------+-----------------------
    ↪
| id                                    | estRows | task        | access object | operator info
    ↪                                       |
+--
    ↪ ---------------------------------------+---------+-------------+---------------+-----------------------
    ↪
| StreamAgg_14                          | 1.00    | root        |               | funcs:count
    ↪ (1)->Column#7                         |
```

```
| └─TableReader_48                        | 9.00    | root         |              | data:
    ↪ ExchangeSender_47                                 |
|   └─ExchangeSender_47                   | 9.00    | cop[tiflash] |              | ExchangeType
    ↪ : PassThrough                           |
|     └─HashJoin_44                       | 9.00    | cop[tiflash] |              | inner join,
    ↪ equal:[eq(test.t1.id, test.t1.id)]      |
|       ├─ExchangeReceiver_19(Build)      | 6.00    | cop[tiflash] |              |
    ↪                                                   |
|       │ └─ExchangeSender_18             | 6.00    | cop[tiflash] |              | ExchangeType
    ↪ : HashPartition, Hash Cols: test.t1.id |
|       │   └─Selection_17                | 6.00    | cop[tiflash] |              | not(isnull(
    ↪ test.t1.id))                            |
|       │     └─TableFullScan_16          | 6.00    | cop[tiflash] | table:a      | keep order:
    ↪ false                                   |
|       └─ExchangeReceiver_23(Probe)      | 6.00    | cop[tiflash] |              |
    ↪                                                   |
|         └─ExchangeSender_22             | 6.00    | cop[tiflash] |              | ExchangeType
    ↪ : HashPartition, Hash Cols: test.t1.id |
|           └─Selection_21                | 6.00    | cop[tiflash] |              | not(isnull(
    ↪ test.t1.id))                            |
|             └─TableFullScan_20          | 6.00    | cop[tiflash] | table:b      | keep order:
    ↪ false                                   |
+--
    ↪ ----------------------------------------+---------+--------------+--------------+-------------------------
    ↪
12 rows in set (0.00 sec)
```

以上执行计划中，

- [TableFullScan_20, Selection_21, ExchangeSender_22] 完成表 b 的数据读取并通过 HashPartition 的方式把数据 shuffle 给上游 MPP 任务。
- [TableFullScan_16, Selection_17, ExchangeSender_18] 完成表 a 的数据读取并通过 HashPartition 的方式把数据 shuffle 给上游 MPP 任务。
- [ExchangeReceiver_19, ExchangeReceiver_23, HashJoin_44, ExchangeSender_47] 完成 join 并把数据返回给 TiDB。

典型的 Broadcast Join 执行计划如下：

```
EXPLAIN SELECT COUNT(*) FROM t1 a JOIN t1 b ON a.id = b.id;
```

```
+--
    ↪ ----------------------------------------+---------+--------------+--------------+-------------------------
    ↪
| id                                    | estRows | task         | access object | operator info
    ↪                                     |
```

```
+--
    ↪ ------------------------------------+---------+-------------+---------------+-----------------------------
    ↪
| StreamAgg_15                          | 1.00    | root        |               | funcs:count
    ↪ (1)->Column#7                       |
| └─TableReader_47                      | 9.00    | root        |               | data:
    ↪ ExchangeSender_46                           |
|   └─ExchangeSender_46                 | 9.00    | cop[tiflash] |              | ExchangeType
    ↪ : PassThrough                       |
|     └─HashJoin_43                     | 9.00    | cop[tiflash] |              | inner join,
    ↪ equal:[eq(test.t1.id, test.t1.id)] |
|       ├─ExchangeReceiver_20(Build)    | 6.00    | cop[tiflash] |              |
    ↪                                             |
|       | └─ExchangeSender_19           | 6.00    | cop[tiflash] |              | ExchangeType
    ↪ : Broadcast                         |
|       |   └─Selection_18              | 6.00    | cop[tiflash] |              | not(isnull(
    ↪ test.t1.id))                        |
|       |     └─TableFullScan_17        | 6.00    | cop[tiflash] | table:a      | keep order:
    ↪ false                               |
|       └─Selection_22(Probe)           | 6.00    | cop[tiflash] |              | not(isnull(
    ↪ test.t1.id))                        |
|         └─TableFullScan_21            | 6.00    | cop[tiflash] | table:b      | keep order:
    ↪ false                               |
+--
    ↪ ------------------------------------+---------+-------------+---------------+-----------------------------
    ↪
```

以上执行计划中，

- `[TableFullScan_17, Selection_18, ExchangeSender_19]` 从小表（表 a）读数据并广播给大表（表 b）数据所在的各个节点。
- `[TableFullScan_21, Selection_22, ExchangeReceiver_20, HashJoin_43, ExchangeSender_46]` 完成 join 并将数据返回给 TiDB。

11.3.2.3.3 对 MPP 模式的查询使用 EXPLAIN ANALYZE

`EXPLAIN ANALYZE` 语句与 `EXPLAIN` 类似，但还会输出一些运行时的信息。一个简单的 `EXPLAIN ANALYZE` 输出信息如下：

```
EXPLAIN ANALYZE SELECT COUNT(*) FROM t1 GROUP BY id;
```

```
+--
    ↪ ---------------------------------+---------+---------+------------------+---------------+-------------
    ↪
| id                                  | estRows | actRows | task             | access object |
    ↪ execution info
```

```
    ↪                                                                                                                    |
    ↪ operator info                                                                                  | memory | disk |
+--
    ↪ ---------------------------------+---------+---------+------------------+---------------+-------------------------
    ↪
| TableReader_31                      | 4.00    | 2       | root             |               |
    ↪ time:44.5ms, loops:2, cop_task: {num: 1, max: 0s, proc_keys: 0, copr_cache_hit_ratio:
    ↪ 0.00}       | data:ExchangeSender_30                                                              | N/A    | N
    ↪ /A  |
| └─ExchangeSender_30                 | 4.00    | 2       | batchCop[tiflash] |               |
    ↪ tiflash_task:{time:16.5ms, loops:1, threads:1}
    ↪                                                                                                  | ExchangeType: PassThrough, tasks:
    ↪ [2, 3, 4]                     | N/A    | N/A  |
|   └─Projection_26                   | 4.00    | 2       | batchCop[tiflash] |               |
    ↪ tiflash_task:{time:16.5ms, loops:1, threads:1}
    ↪                                                                                                  | Column#4
    ↪                                                                                                     | N/A    | N/A  |
|     └─HashAgg_27                    | 4.00    | 2       | batchCop[tiflash] |               |
    ↪ tiflash_task:{time:16.5ms, loops:1, threads:1}
    ↪                                                                                                  | group by:test.t1.id, funcs:sum(
    ↪ Column#7)->Column#4             | N/A    | N/A  |
|       └─ExchangeReceiver_29         | 4.00    | 2       | batchCop[tiflash] |               |
    ↪ tiflash_task:{time:14.5ms, loops:1, threads:20}
    ↪                                                                                                |
    ↪                                                                                                              | N/A    | N/A  |
|         └─ExchangeSender_28         | 4.00    | 0       | batchCop[tiflash] |               |
    ↪ tiflash_task:{time:9.49ms, loops:0, threads:0}
    ↪                                                                                                  | ExchangeType: HashPartition, Hash
    ↪ Cols: test.t1.id, tasks: [1] | N/A    | N/A  |
|           └─HashAgg_9               | 4.00    | 0       | batchCop[tiflash] |               |
    ↪ tiflash_task:{time:9.49ms, loops:0, threads:0}
    ↪                                                                                                  | group by:test.t1.id, funcs:count(1)
    ↪ ->Column#7                    | N/A    | N/A  |
|             └─TableFullScan_25      | 6.00    | 0       | batchCop[tiflash] | table:t1      |
    ↪ tiflash_task:{time:9.49ms, loops:0, threads:0}, tiflash_scan:{dtfile:{total_scanned_packs
    ↪ :1,...}} | keep order:false                                                                                  | N/A    | N/A
    ↪  |
+--
    ↪ ---------------------------------+---------+---------+------------------+---------------+-------------------------
    ↪
```

与 EXPLAIN 相比，ExchangeSender 的 operator info 中多了 task id 的输出，其记录了该查询片段实例化成的 MPP 任务的任务 ID。此外 MPP 算子中都会有 threads 这一列，这列记录了 MPP 在执行该算子时使用的并发数（如果集群由多个节点组成，该并发数是所有节点并发数相加的结果）。

11.3.2.3.4 MPP Version 和 Exchange 数据压缩

从 v6.6.0 开始，MPP 执行计划新增字段 MppVersion 和 Compression。

- MppVersion：MPP 执行计划的版本号，可通过系统变量mpp_version 设置。
- Compression：Exchange 算子的数据压缩模式，可通过系统变量mpp_exchange_compression_mode 设置。如果未启用数据压缩，则执行计划中不显示该字段。

```
mysql > EXPLAIN SELECT COUNT(*) AS count_order FROM lineitem GROUP BY l_returnflag, l_linestatus
    ↪ ORDER BY l_returnflag, l_linestatus;

+--
    ↪ ------------------------------------+-------------+-------------+---------------+------------------
    ↪
| id                                  | estRows      | task         | access object  |
    ↪ operator info
    ↪
    ↪ |
+--
    ↪ ------------------------------------+-------------+-------------+---------------+------------------
    ↪
| Projection_6                        | 3.00         | root         |                | Column
    ↪ #18
    ↪
    ↪ |
| └─Sort_8                             | 3.00         | root         |                |
    ↪ tpch100.lineitem.l_returnflag, tpch100.lineitem.l_linestatus
    ↪
    ↪ |
|   └─TableReader_36                   | 3.00         | root         |                |
    ↪ MppVersion: 1, data:ExchangeSender_35
    ↪
    ↪ |
|     └─ExchangeSender_35              | 3.00         | mpp[tiflash] |                |
    ↪ ExchangeType: PassThrough
    ↪
    ↪ |
|       └─Projection_31                | 3.00         | mpp[tiflash] |                | Column
    ↪ #18, tpch100.lineitem.l_returnflag, tpch100.lineitem.l_linestatus
    ↪
    ↪ |
|         └─HashAgg_32                 | 3.00         | mpp[tiflash] |                | group
    ↪ by:tpch100.lineitem.l_linestatus, tpch100.lineitem.l_returnflag, funcs:sum(Column#23)->
    ↪ Column#18, funcs:firstrow(tpch100.lineitem.l_returnflag)->tpch100.lineitem.l_returnflag,
    ↪ funcs:firstrow(tpch100.lineitem.l_linestatus)->tpch100.lineitem.l_linestatus,
    ↪ stream_count: 20 |
```

```
|           └─ExchangeReceiver_34        | 3.00         | mpp[tiflash] |                |
    ↪ stream_count: 20
    ↪
    ↪ |
|             └─ExchangeSender_33        | 3.00         | mpp[tiflash] |                |
    ↪ ExchangeType: HashPartition, Compression: FAST, Hash Cols: [name: tpch100.lineitem.
    ↪ l_returnflag, collate: utf8mb4_bin], [name: tpch100.lineitem.l_linestatus, collate:
    ↪ utf8mb4_bin], stream_count: 20
    ↪                                                                                 |
|               └─HashAgg_14             | 3.00         | mpp[tiflash] |                | group
    ↪ by:tpch100.lineitem.l_linestatus, tpch100.lineitem.l_returnflag, funcs:count(1)->Column
    ↪ #23
    ↪
    ↪ |
|                 └─TableFullScan_30     | 600037902.00 | mpp[tiflash] | table:lineitem | keep
    ↪ order:false
    ↪
    ↪ |
+--
    ↪ ---------------------------------------+--------------+--------------+----------------+-----------------
    ↪
```

例如上面的执行计划结果中，TiDB 使用了版本为 1 的 MPP 执行计划来构建 TableReader。其中类型为 HashPartition 的 ExchangeSender 算子使用 FAST 数据压缩模式，类型为 PassThrough 的 ExchangeSender 算子未启用数据压缩。

11.3.2.3.5 其他类型查询的执行计划

- 索引查询的执行计划
- Join 查询的执行计划
- 子查询的执行计划
- 聚合查询的执行计划
- 视图查询的执行计划
- 分区查询的执行计划
- 索引合并查询的执行计划

11.3.2.4 用 EXPLAIN 查看索引查询的执行计划

SQL 查询可能会使用索引，可以通过 EXPLAIN 语句来查看索引查询的执行计划。本文提供多个示例，以帮助用户理解索引查询是如何执行的。

TiDB 支持以下使用索引的算子来提升查询速度：

- IndexLookup
- IndexReader

- Point_Get 和 Batch_Point_Get
- IndexFullScan

本文档中的示例都基于以下数据：

```
CREATE TABLE t1 (
 id INT NOT NULL PRIMARY KEY auto_increment,
 intkey INT NOT NULL,
 pad1 VARBINARY(1024),
 INDEX (intkey)
);

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1024), RANDOM_BYTES(1024) FROM dual;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1024), RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 10000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1024), RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 10000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1024), RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 10000;
```

11.3.2.4.1 IndexLookup

TiDB 从二级索引检索数据时会使用 IndexLookup 算子。例如，以下所有查询均会在 intkey 列的索引上使用 IndexLookup 算子：

```
EXPLAIN SELECT * FROM t1 WHERE intkey = 123;
EXPLAIN SELECT * FROM t1 WHERE intkey < 10;
EXPLAIN SELECT * FROM t1 WHERE intkey BETWEEN 300 AND 310;
EXPLAIN SELECT * FROM t1 WHERE intkey IN (123,29,98);
EXPLAIN SELECT * FROM t1 WHERE intkey >= 99 AND intkey <= 103;
```

```
+--
    ↪ ----------------------------+---------+-----------+--------------------------------+-------------------
    ↪
| id                            | estRows | task      | access object                  | operator
    ↪  info                      |
+--
    ↪ ----------------------------+---------+-----------+--------------------------------+-------------------
    ↪
| IndexLookUp_10                | 1.00    | root      |                                |
    ↪                                      |
| ├─IndexRangeScan_8(Build)     | 1.00    | cop[tikv] | table:t1, index:intkey(intkey) | range
    ↪ :[123,123], keep order:false |
| └─TableRowIDScan_9(Probe)     | 1.00    | cop[tikv] | table:t1                       | keep
    ↪ order:false                  |
```

```
+--
    ↪ ----------------------------+---------+-----------+--------------------------------+-----------------
    ↪
3 rows in set (0.00 sec)

+--
    ↪ ----------------------------+---------+-----------+--------------------------------+-----------------
    ↪
| id                           | estRows | task      | access object                  | operator
    ↪  info                      |
+--
    ↪ ----------------------------+---------+-----------+--------------------------------+-----------------
    ↪
| IndexLookUp_10               | 3.60    | root      |                                |
    ↪                                     |
| ├─IndexRangeScan_8(Build)    | 3.60    | cop[tikv] | table:t1, index:intkey(intkey) | range
    ↪ :[-inf,10), keep order:false |
| └─TableRowIDScan_9(Probe)    | 3.60    | cop[tikv] | table:t1                       | keep
    ↪ order:false                  |
+--
    ↪ ----------------------------+---------+-----------+--------------------------------+-----------------
    ↪
3 rows in set (0.00 sec)

+--
    ↪ ----------------------------+---------+-----------+--------------------------------+-----------------
    ↪
| id                           | estRows | task      | access object                  | operator
    ↪  info                      |
+--
    ↪ ----------------------------+---------+-----------+--------------------------------+-----------------
    ↪
| IndexLookUp_10               | 5.67    | root      |                                |
    ↪                                     |
| ├─IndexRangeScan_8(Build)    | 5.67    | cop[tikv] | table:t1, index:intkey(intkey) | range
    ↪ :[300,310], keep order:false |
| └─TableRowIDScan_9(Probe)    | 5.67    | cop[tikv] | table:t1                       | keep
    ↪ order:false                  |
+--
    ↪ ----------------------------+---------+-----------+--------------------------------+-----------------
    ↪
3 rows in set (0.00 sec)

+--
    ↪ ----------------------------+---------+-----------+--------------------------------+-----------------
```

```
    ↪
| id                            | estRows | task      | access object                    | operator
    ↪  info                                      |
+--
    ↪ ------------------------------+---------+-----------+----------------------------------+------------------
    ↪
| IndexLookUp_10                | 4.00    | root      |                                  |
    ↪                                                  |
| ├─IndexRangeScan_8(Build)     | 4.00    | cop[tikv] | table:t1, index:intkey(intkey) | range
    ↪ :[29,29], [98,98], [123,123], keep order:false |
| └─TableRowIDScan_9(Probe)     | 4.00    | cop[tikv] | table:t1                         | keep
    ↪ order:false                                    |
+--
    ↪ ------------------------------+---------+-----------+----------------------------------+------------------
    ↪
3 rows in set (0.00 sec)

+--
    ↪ ------------------------------+---------+-----------+----------------------------------+------------------
    ↪
| id                            | estRows | task      | access object                    | operator
    ↪  info                     |
+--
    ↪ ------------------------------+---------+-----------+----------------------------------+------------------
    ↪
| IndexLookUp_10                | 6.00    | root      |                                  |
    ↪                                       |
| ├─IndexRangeScan_8(Build)     | 6.00    | cop[tikv] | table:t1, index:intkey(intkey) | range
    ↪ :[99,103], keep order:false |
| └─TableRowIDScan_9(Probe)     | 6.00    | cop[tikv] | table:t1                         | keep
    ↪ order:false                   |
+--
    ↪ ------------------------------+---------+-----------+----------------------------------+------------------
    ↪
3 rows in set (0.00 sec)
```

IndexLookup 算子有以下两个子节点：

- ├─IndexRangeScan_8(Build) 算子节点对 intkey 列的索引执行范围扫描，并检索内部的 RowID 值（对此表而言，即为主键）。
- └─TableRowIDScan_9(Probe) 算子节点随后从表数据中检索整行。

IndexLookup 任务分以上两步执行。如果满足条件的行较多，SQL 优化器可能会根据统计信息选择使用 TableFullScan 算子。在以下示例中，很多行都满足 intkey > 100 这一条件，因此优化器选择了 TableFullScan：

```
EXPLAIN SELECT * FROM t1 WHERE intkey > 100;
```

```
+-------------------------+---------+-----------+---------------+-------------------------+
| id                      | estRows | task      | access object | operator info           |
+-------------------------+---------+-----------+---------------+-------------------------+
| TableReader_7           | 898.50  | root      |               | data:Selection_6        |
| └─Selection_6           | 898.50  | cop[tikv] |               | gt(test.t1.intkey, 100) |
|   └─TableFullScan_5     | 1010.00 | cop[tikv] | table:t1      | keep order:false        |
+-------------------------+---------+-----------+---------------+-------------------------+
3 rows in set (0.00 sec)
```

IndexLookup 算子能在带索引的列上有效优化 LIMIT：

```
EXPLAIN SELECT * FROM t1 ORDER BY intkey DESC LIMIT 10;
```

```
+--
    ↪ ----------------------------+---------+-----------+--------------------------------+------------------
    ↪
| id                            | estRows | task      | access object                  |
    ↪ operator info                       |
+--
    ↪ ----------------------------+---------+-----------+--------------------------------+------------------
    ↪
| IndexLookUp_21                | 10.00   | root      |                                | limit
    ↪ embedded(offset:0, count:10) |
| ├─Limit_20(Build)             | 10.00   | cop[tikv] |                                | offset
    ↪ :0, count:10                |
| │ └─IndexFullScan_18          | 10.00   | cop[tikv] | table:t1, index:intkey(intkey) | keep
    ↪ order:true, desc            |
| └─TableRowIDScan_19(Probe)    | 10.00   | cop[tikv] | table:t1                       | keep
    ↪ order:false, stats:pseudo   |
+--
    ↪ ----------------------------+---------+-----------+--------------------------------+------------------
    ↪
4 rows in set (0.00 sec)
```

以上示例中，TiDB 从 intkey 索引读取最后 10 行，然后从表数据中检索这些行的 RowID 值。

11.3.2.4.2 IndexReader

TiDB 支持覆盖索引优化 (covering index optimization)。如果 TiDB 能从索引中检索出所有行，就会跳过 IndexLookup 任务中通常所需的第二步（即从表数据中检索整行）。示例如下：

```
EXPLAIN SELECT * FROM t1 WHERE intkey = 123;
EXPLAIN SELECT id FROM t1 WHERE intkey = 123;
```

```
+--
    ↪ ----------------------------+---------+-----------+--------------------------------+-------------------
    ↪
| id                            | estRows | task      | access object                  | operator
    ↪  info                      |
+--
    ↪ ----------------------------+---------+-----------+--------------------------------+-------------------
    ↪
| IndexLookUp_10                | 1.00    | root      |                                |
    ↪                                      |
| ├─IndexRangeScan_8(Build)     | 1.00    | cop[tikv] | table:t1, index:intkey(intkey) | range
    ↪ :[123,123], keep order:false |
| └─TableRowIDScan_9(Probe)     | 1.00    | cop[tikv] | table:t1                       | keep
    ↪ order:false                  |
+--
    ↪ ----------------------------+---------+-----------+--------------------------------+-------------------
    ↪
3 rows in set (0.00 sec)

+--
    ↪ ------------------------+---------+-----------+--------------------------------+-----------------------
    ↪
| id                        | estRows | task      | access object                  | operator info
    ↪                       |
+--
    ↪ ------------------------+---------+-----------+--------------------------------+-----------------------
    ↪
| Projection_4              | 1.00    | root      |                                | test.t1.id
    ↪                          |
| └─IndexReader_6           | 1.00    | root      |                                | index:
    ↪ IndexRangeScan_5             |
|   └─IndexRangeScan_5      | 1.00    | cop[tikv] | table:t1, index:intkey(intkey) | range
    ↪ :[123,123], keep order:false |
+--
    ↪ ------------------------+---------+-----------+--------------------------------+-----------------------
    ↪
3 rows in set (0.00 sec)
```

以上结果中，id 也是内部的 RowID 值，因此 id 也存储在 intkey 索引中。部分 └─IndexRangeScan_5 任务使用 intkey 索引后，可直接返回 RowID 值。

11.3.2.4.3 Point_Get 和 Batch_Point_Get

TiDB 直接从主键或唯一键检索数据时会使用 Point_Get 或 Batch_Point_Get 算子。这两个算子比 IndexLookup 更有效率。示例如下：

```
EXPLAIN SELECT * FROM t1 WHERE id = 1234;
EXPLAIN SELECT * FROM t1 WHERE id IN (1234,123);

ALTER TABLE t1 ADD unique_key INT;
UPDATE t1 SET unique_key = id;
ALTER TABLE t1 ADD UNIQUE KEY (unique_key);

EXPLAIN SELECT * FROM t1 WHERE unique_key = 1234;
EXPLAIN SELECT * FROM t1 WHERE unique_key IN (1234, 123);
```

```
+-------------+---------+------+---------------+---------------+
| id          | estRows | task | access object | operator info |
+-------------+---------+------+---------------+---------------+
| Point_Get_1 | 1.00    | root | table:t1      | handle:1234   |
+-------------+---------+------+---------------+---------------+
1 row in set (0.00 sec)

+--
    ↪ -----------------+---------+------+---------------+-----------------------------------------------+
    ↪
| id                | estRows | task | access object | operator info
    ↪                                    |
+--
    ↪ -----------------+---------+------+---------------+-----------------------------------------------+
    ↪
| Batch_Point_Get_1 | 2.00    | root | table:t1      | handle:[1234 123], keep order:false, desc:
    ↪ false |
+--
    ↪ -----------------+---------+------+---------------+-----------------------------------------------+
    ↪
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.27 sec)

Query OK, 1010 rows affected (0.06 sec)
Rows matched: 1010  Changed: 1010  Warnings: 0

Query OK, 0 rows affected (0.37 sec)

+-------------+---------+------+-----------------------------------------+---------------+
| id          | estRows | task | access object                           | operator info |
+-------------+---------+------+-----------------------------------------+---------------+
| Point_Get_1 | 1.00    | root | table:t1, index:unique_key(unique_key) |               |
+-------------+---------+------+-----------------------------------------+---------------+
```

```
1 row in set (0.00 sec)

+--
    ↪ -----------------+---------+------+----------------------------------------+-----------------------------
    ↪
| id                | estRows | task | access object                          | operator info
    ↪                 |
+--
    ↪ -----------------+---------+------+----------------------------------------+-----------------------------
    ↪
| Batch_Point_Get_1 | 2.00    | root | table:t1, index:unique_key(unique_key) | keep order:false,
    ↪  desc:false |
+--
    ↪ -----------------+---------+------+----------------------------------------+-----------------------------
    ↪
1 row in set (0.00 sec)
```

11.3.2.4.4 IndexFullScan

索引是有序的，所以优化器可以使用 `IndexFullScan` 算子来优化常见的查询，例如在索引值上使用 `MIN` 或 `Max` 函数：

```
EXPLAIN SELECT MIN(intkey) FROM t1;
EXPLAIN SELECT MAX(intkey) FROM t1;
```

```
+--
    ↪ ---------------------------+---------+-----------+--------------------------------+----------------------
    ↪
| id                           | estRows | task      | access object                  | operator
    ↪ info                       |
+--
    ↪ ---------------------------+---------+-----------+--------------------------------+----------------------
    ↪
| StreamAgg_12                 | 1.00    | root      |                                | funcs:min
    ↪ (test.t1.intkey)->Column#4 |
| └─Limit_16                   | 1.00    | root      |                                 | offset
    ↪ :0, count:1                    |
|   └─IndexReader_29           | 1.00    | root      |                                 | index:
    ↪ Limit_28                       |
|     └─Limit_28               | 1.00    | cop[tikv] |                                 | offset
    ↪ :0, count:1                    |
|       └─IndexFullScan_27     | 1.00    | cop[tikv] | table:t1, index:intkey(intkey) | keep
    ↪ order:true                      |
+--
    ↪ ---------------------------+---------+-----------+--------------------------------+----------------------
    ↪
```

```
5 rows in set (0.00 sec)

+--
    ↪ ---------------------------+---------+-----------+--------------------------------+-------------------
    ↪
| id                           | estRows | task      | access object                  | operator
    ↪ info                         |
+--
    ↪ ---------------------------+---------+-----------+--------------------------------+-------------------
    ↪
| StreamAgg_12                 | 1.00    | root      |                                | funcs:max
    ↪ (test.t1.intkey)->Column#4 |
| └─Limit_16                   | 1.00    | root      |                                | offset
    ↪ :0, count:1                  |
|   └─IndexReader_29           | 1.00    | root      |                                | index:
    ↪ Limit_28                     |
|     └─Limit_28               | 1.00    | cop[tikv] |                                | offset
    ↪ :0, count:1                  |
|       └─IndexFullScan_27     | 1.00    | cop[tikv] | table:t1, index:intkey(intkey) | keep
    ↪ order:true, desc             |
+--
    ↪ ---------------------------+---------+-----------+--------------------------------+-------------------
    ↪
5 rows in set (0.00 sec)
```

以上语句的执行过程中，TiDB 在每一个 TiKV Region 上执行 IndexFullScan 操作。虽然算子名为 FullScan 即全扫描，TiDB 只读取第一行 (└─Limit_28)。每个 TiKV Region 返回各自的 MIN 或 MAX 值给 TiDB，TiDB 再执行流聚合运算来过滤出一行数据。即使表为空，带 MAX 或 MIN 函数的流聚合运算也能保证返回 NULL 值。

相反，在没有索引的值上执行 MIN 函数会在每一个 TiKV Region 上执行 TableFullScan 操作。该查询会要求在 TiKV 中扫描所有行，但 TopN 计算可保证每个 TiKV Region 只返回一行数据给 TiDB。尽管 TopN 能减少 TiDB 和 TiKV 之间的多余数据传输，但该查询的效率仍远不及以上示例 (MIN 能够使用索引)。

```
EXPLAIN SELECT MIN(pad1) FROM t1;
```

```
+--
    ↪ ----------------------------+---------+-----------+--------------+--------------------------------+
    ↪
| id                           | estRows | task      | access object | operator info
    ↪                      |
+--
    ↪ ----------------------------+---------+-----------+--------------+--------------------------------+
    ↪
| StreamAgg_13                 | 1.00    | root      |               | funcs:min(test.t1.pad1)
    ↪ ->Column#4 |
| └─TopN_14                    | 1.00    | root      |               | test.t1.pad1, offset:0,
    ↪  count:1   |
```

```
|   └─TableReader_23              | 1.00    | root      |               | data:TopN_22
    ↪                       |
|     └─TopN_22                   | 1.00    | cop[tikv] |               | test.t1.pad1, offset:0,
    ↪  count:1   |
|       └─Selection_21            | 1008.99 | cop[tikv] |               | not(isnull(test.t1.pad1
    ↪ ))         |
|         └─TableFullScan_20      | 1010.00 | cop[tikv] | table:t1      | keep order:false
    ↪                   |
+--
    ↪ ------------------------------+---------+-----------+---------------+--------------------------------+
    ↪
6 rows in set (0.00 sec)
```

执行以下语句时，TiDB 将使用 IndexFullScan 算子扫描索引中的每一行：

```
EXPLAIN SELECT SUM(intkey) FROM t1;
EXPLAIN SELECT AVG(intkey) FROM t1;
```

```
+--
    ↪ -------------------------+---------+-----------+-------------------------------+-------------------
    ↪
| id                          | estRows | task      | access object                 | operator
    ↪ info                        |
+--
    ↪ -------------------------+---------+-----------+-------------------------------+-------------------
    ↪
| StreamAgg_20                | 1.00    | root      |                               | funcs:sum(
    ↪ Column#6)->Column#4       |
| └─IndexReader_21            | 1.00    | root      |                               | index:
    ↪ StreamAgg_8                   |
|   └─StreamAgg_8             | 1.00    | cop[tikv] |                               | funcs:sum(
    ↪ test.t1.intkey)->Column#6 |
|     └─IndexFullScan_19      | 1010.00 | cop[tikv] | table:t1, index:intkey(intkey) | keep order
    ↪ :false                    |
+--
    ↪ -------------------------+---------+-----------+-------------------------------+-------------------
    ↪
4 rows in set (0.00 sec)

+--
    ↪ -------------------------+---------+-----------+-------------------------------+-------------------
    ↪
| id                          | estRows | task      | access object                 | operator
    ↪ info                                                          |
+--
    ↪ -------------------------+---------+-----------+-------------------------------+-------------------
```

```
    ↪
| StreamAgg_20               | 1.00    | root      |                                | funcs:avg(
    ↪ Column#7, Column#8)->Column#4                                        |
| └─IndexReader_21           | 1.00    | root      |                                 | index:
    ↪ StreamAgg_8                                                          |
|   └─StreamAgg_8            | 1.00    | cop[tikv] |                                 | funcs:
    ↪ count(test.t1.intkey)->Column#7, funcs:sum(test.t1.intkey)->Column#8 |
|     └─IndexFullScan_19     | 1010.00 | cop[tikv] | table:t1, index:intkey(intkey) | keep order
    ↪ :false                                                               |
+--
    ↪ -------------------------+---------+-----------+--------------------------------+-----------------------
    ↪
4 rows in set (0.00 sec)
```

以上示例中，IndexFullScan 比 TableFullScan 更有效率，因为 (intkey + RowID) 索引中值的长度小于整行的长度。

以下语句不支持使用 IndexFullScan 算子，因为涉及该表中的其他列：

```
EXPLAIN SELECT AVG(intkey), ANY_VALUE(pad1) FROM t1;
```

```
+--
    ↪ ---------------------------+---------+-----------+--------------+--------------------------------
    ↪
| id                           | estRows | task      | access object | operator info
    ↪
    ↪ |
+--
    ↪ ---------------------------+---------+-----------+--------------+--------------------------------
    ↪
| Projection_4                 | 1.00    | root      |               | Column#4, any_value(test.
    ↪ t1.pad1)->Column#5
    ↪                                                                                                 |
| └─StreamAgg_16               | 1.00    | root      |               | funcs:avg(Column#10,
    ↪ Column#11)->Column#4, funcs:firstrow(Column#12)->test.t1.pad1
    ↪                                      |
|   └─TableReader_17           | 1.00    | root      |               | data:StreamAgg_8
    ↪
    ↪ |
|     └─StreamAgg_8            | 1.00    | cop[tikv] |               | funcs:count(test.t1.
    ↪ intkey)->Column#10, funcs:sum(test.t1.intkey)->Column#11, funcs:firstrow(test.t1.pad1)->
    ↪ Column#12 |
|       └─TableFullScan_15     | 1010.00 | cop[tikv] | table:t1      | keep order:false
    ↪
    ↪ |
+--
    ↪ ---------------------------+---------+-----------+--------------+--------------------------------
```

```
    ↪
5 rows in set (0.00 sec)
```

11.3.2.4.5 其他类型查询的执行计划

- MPP 模式查询的执行计划
- Join 查询的执行计划
- 子查询的执行计划
- 聚合查询的执行计划
- 视图查询的执行计划
- 分区查询的执行计划
- 索引合并查询的执行计划

11.3.2.5 用 EXPLAIN 查看 JOIN 查询的执行计划

SQL 查询中可能会使用 JOIN 进行表连接，可以通过 EXPLAIN 语句来查看 JOIN 查询的执行计划。本文提供多个示例，以帮助用户理解表连接查询是如何执行的。

在 TiDB 中，SQL 优化器需要确定数据表的连接顺序，且要判断对于某条特定的 SQL 语句，哪一种 Join 算法最为高效。

本文档使用的示例数据如下：

```
CREATE TABLE t1 (id BIGINT NOT NULL PRIMARY KEY auto_increment, pad1 BLOB, pad2 BLOB, pad3 BLOB,
    ↪ int_col INT NOT NULL DEFAULT 0);
CREATE TABLE t2 (id BIGINT NOT NULL PRIMARY KEY auto_increment, t1_id BIGINT NOT NULL, pad1 BLOB,
    ↪  pad2 BLOB, pad3 BLOB, INDEX(t1_id));
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM
    ↪ dual;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
```

```
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
UPDATE t1 SET int_col = 1 WHERE pad1 = (SELECT pad1 FROM t1 ORDER BY RAND() LIMIT 1);
SELECT SLEEP(1);
ANALYZE TABLE t1, t2;
```

11.3.2.5.1 Index Join

如果预计需要连接的行数较少（一般小于 1 万行），推荐使用 Index Join 算法。这个算法与 MySQL 主要使用的 Join 算法类似。在下表的示例中，`TableReader_28(Build)` 算子首先读取表 `t1`，然后根据在 `t1` 中匹配到的每行数据，依次探查表 `t2` 中的数据：

> **注意：**
>
> 在执行计划返回结果中，自 v6.4.0 版本起，特定算子（即 IndexJoin 和 Apply 算子的 Probe 端所有子节点）的 estRows 字段意义与 v6.4.0 版本之前的有所不同。细节请参考TiDB 执行计划概览。

```
EXPLAIN SELECT /*+ INL_JOIN(t1, t2) */ * FROM t1 INNER JOIN t2 ON t1.id = t2.t1_id;
```

```
+--
    ↪ -------------------------------+----------+-----------+-------------------------------+------------------
    ↪
```

```
| id                              | estRows  | task      | access object                 |
    ↪ operator info
    ↪
    ↪ |
+--
    ↪ -------------------------------+----------+-----------+-------------------------------+------------------
    ↪
| IndexJoin_11                    | 90000.00 | root      |                               | inner
    ↪ join, inner:IndexLookUp_10, outer key:test.t1.id, inner key:test.t2.t1_id, equal cond:eq(
    ↪ test.t1.id, test.t2.t1_id) |
| ├─TableReader_29(Build)         | 71010.00 | root      |                               | data:
    ↪ TableFullScan_28
    ↪
    ↪ |
| │ └─TableFullScan_28            | 71010.00 | cop[tikv] | table:t1                      | keep
    ↪ order:false
    ↪
    ↪ |
| └─IndexLookUp_10(Probe)         | 90000.00 | root      |                               |
    ↪
    ↪ |
|   ├─IndexRangeScan_8(Build)     | 90000.00 | cop[tikv] | table:t2, index:t1_id(t1_id) | range:
    ↪  decided by [eq(test.t2.t1_id, test.t1.id)], keep order:false
    ↪                                                        |
|   └─TableRowIDScan_9(Probe)     | 90000.00 | cop[tikv] | table:t2                      | keep
    ↪ order:false
    ↪
    ↪ |
+--
    ↪ -------------------------------+----------+-----------+-------------------------------+------------------
    ↪
```

Index Join 算法对内存消耗较小，但如果需要执行大量探查操作，运行速度可能会慢于其他 Join 算法。以下面这条查询语句为例：

```
SELECT * FROM t1 INNER JOIN t2 ON t1.id=t2.t1_id WHERE t1.pad1 = 'value' and t2.pad1='value';
```

在 Inner Join 操作中，TiDB 会先执行 Join Reorder 算法，所以不能确定会先读取 t1 还是 t2。假设 TiDB 先读取了 t1 来构建 Build 端，那么 TiDB 会在探查 t2 前先根据谓词 t1.col = 'value' 筛选数据，但接下来每次探查 t2 时都要应用谓词 t2.col='value'。所以对于这条语句，Index Join 算法可能不如其他 Join 算法高效。

但如果 Build 端的数据量比 Probe 端小，且 Probe 端的数据已预先建立了索引，那么这种情况下 Index Join 算法效率更高。在下面这段查询语句中，因为 Index Join 比 Hash Join 效率低，所以 SQL 优化器选择了 Hash Join 算法：

```
-- 删除已有索引
ALTER TABLE t2 DROP INDEX t1_id;
```

```
EXPLAIN ANALYZE SELECT /*+ INL_JOIN(t1, t2) */  * FROM t1 INNER JOIN t2 ON t1.id = t2.t1_id WHERE
    ↪  t1.int_col = 1;
EXPLAIN ANALYZE SELECT /*+ HASH_JOIN(t1, t2) */  * FROM t1 INNER JOIN t2 ON t1.id = t2.t1_id
    ↪ WHERE t1.int_col = 1;
EXPLAIN ANALYZE SELECT * FROM t1 INNER JOIN t2 ON t1.id = t2.t1_id WHERE t1.int_col = 1;
```

```
+--
    ↪ --------------------------+----------+---------+-----------+--------------+-----------------------------
    ↪
| id                          | estRows  | actRows | task      | access object | execution info
    ↪
    ↪ | operator info
    ↪
    ↪ | memory  | disk |
+--
    ↪ --------------------------+----------+---------+-----------+--------------+-----------------------------
    ↪
| IndexJoin_14                | 90000.00 | 0       | root      |               | time:330.2ms,
    ↪ loops:1, inner:{total:72.2ms, concurrency:5, task:12, construct:58.6ms, fetch:13.5ms,
    ↪ build:2.12µs}, probe:26.1ms
    ↪
    ↪ | inner join, inner:TableReader_10, outer key:test.t2.t1_id, inner key:test.t1.id, equal
    ↪ cond:eq(test.t2.t1_id, test.t1.id) | 88.5 MB | N/A  |
| ├─TableReader_20(Build)     | 90000.00 | 90000   | root      |               | time:307.2ms,
    ↪ loops:96, cop_task: {num: 24, max: 130.6ms, min: 170.9µs, avg: 33.5ms, p95: 105ms,
    ↪ max_proc_keys: 10687, p95_proc_keys: 9184, tot_proc: 472ms, rpc_num: 24, rpc_time: 802.4
    ↪ ms, copr_cache_hit_ratio: 0.62, distsql_concurrency: 15}
    ↪                                                                                | data:
    ↪ TableFullScan_19
    ↪
    ↪ | 58.6 MB | N/A  |
| │ └─TableFullScan_19        | 90000.00 | 90000   | cop[tikv] | table:t2      | tikv_task:{proc
    ↪  max:34ms, min:0s, avg: 15.3ms, p80:24ms, p95:30ms, iters:181, tasks:24}, scan_detail: {
    ↪ total_process_keys: 69744, total_process_keys_size: 217533936, total_keys: 69753,
    ↪ get_snapshot_time: 701.6µs, rocksdb: {delete_skipped_count: 97368, key_skipped_count:
    ↪ 236847, block: {cache_hit_count: 3509}}}  | keep order:false
    ↪
    ↪ | N/A     | N/A  |
| └─TableReader_10(Probe)     | 12617.92 | 0       | root      |               | time:11.9ms,
    ↪ loops:12, cop_task: {num: 42, max: 848.8µs, min: 199µs, avg: 451.8µs, p95: 846.2µs,
    ↪ max_proc_keys: 7, p95_proc_keys: 5, rpc_num: 42, rpc_time: 18.3ms, copr_cache_hit_ratio:
    ↪ 0.00, distsql_concurrency: 15}
    ↪
    ↪ | data:Selection_9
    ↪
```

```
    ↪ | N/A     | N/A  |
|   └─Selection_9             | 12617.92 | 0       | cop[tikv] |               | tikv_task:{proc
    ↪  max:0s, min:0s, avg: 0s, p80:0s, p95:0s, iters:42, tasks:42}, scan_detail: {
    ↪ total_process_keys: 56, total_process_keys_size: 174608, total_keys: 77,
    ↪ get_snapshot_time: 727.7µs, rocksdb: {block: {cache_hit_count: 154}}}
    ↪                                                                              | eq(test.
    ↪ t1.int_col, 1)
    ↪
    ↪ | N/A     | N/A  |
|     └─TableRangeScan_8      | 90000.00 | 56      | cop[tikv] | table:t1      | tikv_task:{proc
    ↪  max:0s, min:0s, avg: 0s, p80:0s, p95:0s, iters:42, tasks:42}
    ↪
    ↪ | range: decided by [test.t2.t1_id], keep order:false
    ↪                                                                    | N/A     | N/A  |
+--
    ↪ --------------------------+----------+---------+-----------+---------------+-------------------------------
    ↪


+--
    ↪ --------------------------+----------+---------+-----------+---------------+-------------------------------
    ↪
| id                          | estRows  | actRows | task      | access object | execution info
    ↪
    ↪ | operator info                                  | memory  | disk    |
+--
    ↪ --------------------------+----------+---------+-----------+---------------+-------------------------------
    ↪
| HashJoin_20                 | 90000.00 | 0       | root      |               | time:313.6ms,
    ↪ loops:1, build_hash_table:{total:24.6ms, fetch:21.2ms, build:3.32ms}, probe:{concurrency
    ↪ :5, total:1.57s, max:313.5ms, probe:18.9ms, fetch:1.55s}
    ↪
    ↪ | inner join, equal:[eq(test.t1.id, test.t2.t1_id)] | 32.0 MB | 0 Bytes |
| ├─TableReader_23(Build)     | 9955.54  | 10000   | root      |               | time:23.6ms,
    ↪ loops:12, cop_task: {num: 11, max: 504.6µs, min: 203.7µs, avg: 377.4µs, p95: 504.6µs,
    ↪ rpc_num: 11, rpc_time: 3.92ms, copr_cache_hit_ratio: 1.00, distsql_concurrency: 15}
    ↪
    ↪ | data:Selection_22                              | 14.9 MB | N/A     |
| │ └─Selection_22             | 9955.54  | 10000   | cop[tikv] |               | tikv_task:{
    ↪ proc max:104ms, min:3ms, avg: 24.4ms, p80:33ms, p95:104ms, iters:113, tasks:11},
    ↪ scan_detail: {get_snapshot_time: 241.4µs, rocksdb: {block: {}}}
    ↪
    ↪ | eq(test.t1.int_col, 1)                         | N/A     | N/A     |
| │   └─TableFullScan_21       | 71010.00 | 71010   | cop[tikv] | table:t1      | tikv_task:{
    ↪ proc max:101ms, min:3ms, avg: 23.8ms, p80:33ms, p95:101ms, iters:113, tasks:11}
    ↪
```

```
    ↪ | keep order:false                                  | N/A     | N/A     |
| └─TableReader_25(Probe)      | 90000.00 | 90000   | root      |               | time:293.7ms,
    ↪ loops:91, cop_task: {num: 24, max: 105.7ms, min: 210.9µs, avg: 31.4ms, p95: 103.8ms,
    ↪ max_proc_keys: 10687, p95_proc_keys: 9184, tot_proc: 407ms, rpc_num: 24, rpc_time: 752.2
    ↪ ms, copr_cache_hit_ratio: 0.62, distsql_concurrency: 15}
    ↪                                                                    | data:
    ↪ TableFullScan_24                             | 58.6 MB | N/A     |
|   └─TableFullScan_24         | 90000.00 | 90000   | cop[tikv] | table:t2      | tikv_task:{
    ↪ proc max:31ms, min:0s, avg: 13ms, p80:19ms, p95:26ms, iters:181, tasks:24}, scan_detail:
    ↪ {total_process_keys: 69744, total_process_keys_size: 217533936, total_keys: 69753,
    ↪ get_snapshot_time: 637.2µs, rocksdb: {delete_skipped_count: 97368, key_skipped_count:
    ↪ 236847, block: {cache_hit_count: 3509}}}  | keep order:false
    ↪                                    | N/A     | N/A     |
+--
    ↪ ---------------------------+----------+---------+-----------+---------------+----------------------------
    ↪

+--
    ↪ ---------------------------+----------+---------+-----------+---------------+----------------------------
    ↪
| id                           | estRows  | actRows | task      | access object | execution info
    ↪
    ↪ | operator info                                  | memory  | disk    |
+--
    ↪ ---------------------------+----------+---------+-----------+---------------+----------------------------
    ↪
| HashJoin_21                  | 90000.00 | 0       | root      |               | time:331.7ms,
    ↪ loops:1, build_hash_table:{total:32.7ms, fetch:26ms, build:6.73ms}, probe:{concurrency:5,
    ↪  total:1.66s, max:331.3ms, probe:16ms, fetch:1.64s}
    ↪
    ↪ | inner join, equal:[eq(test.t1.id, test.t2.t1_id)] | 32.3 MB | 0 Bytes |
| ├─TableReader_26(Build)      | 9955.54  | 10000   | root      |               | time:30.4ms,
    ↪ loops:13, cop_task: {num: 11, max: 1.87ms, min: 844.7µs, avg: 1.29ms, p95: 1.87ms,
    ↪ rpc_num: 11, rpc_time: 13.5ms, copr_cache_hit_ratio: 1.00, distsql_concurrency: 15}
    ↪
    ↪ | data:Selection_25                              | 12.2 MB | N/A     |
| │ └─Selection_25              | 9955.54  | 10000   | cop[tikv] |               | tikv_task:{
    ↪ proc max:104ms, min:3ms, avg: 24.4ms, p80:33ms, p95:104ms, iters:113, tasks:11},
    ↪ scan_detail: {get_snapshot_time: 521µs, rocksdb: {block: {}}}
    ↪
    ↪ | eq(test.t1.int_col, 1)                         | N/A     | N/A     |
| │   └─TableFullScan_24        | 71010.00 | 71010   | cop[tikv] | table:t1      | tikv_task:{
    ↪ proc max:101ms, min:3ms, avg: 23.8ms, p80:33ms, p95:101ms, iters:113, tasks:11}
    ↪
    ↪ | keep order:false                               | N/A     | N/A     |
```

```
| └─TableReader_23(Probe)      | 90000.00 | 90000   | root      |               | time:308.6ms,
    ↪ loops:91, cop_task: {num: 24, max: 123.3ms, min: 518.9µs, avg: 32.4ms, p95: 113.4ms,
    ↪ max_proc_keys: 10687, p95_proc_keys: 9184, tot_proc: 499ms, rpc_num: 24, rpc_time: 776ms,
    ↪  copr_cache_hit_ratio: 0.62, distsql_concurrency: 15}
    ↪                                                                              | data:
    ↪ TableFullScan_22                              | 58.6 MB | N/A     |
|   └─TableFullScan_22         | 90000.00 | 90000   | cop[tikv] | table:t2      | tikv_task:{
    ↪ proc max:44ms, min:0s, avg: 16.8ms, p80:27ms, p95:40ms, iters:181, tasks:24}, scan_detail
    ↪ : {total_process_keys: 69744, total_process_keys_size: 217533936, total_keys: 69753,
    ↪ get_snapshot_time: 955.4µs, rocksdb: {delete_skipped_count: 97368, key_skipped_count:
    ↪ 236847, block: {cache_hit_count: 3509}}}  | keep order:false
    ↪                                    | N/A     | N/A     |
+--
    ↪ ---------------------------+----------+---------+-----------+--------------+-------------------------
    ↪
```

在上面所示的 Index Join 操作中，`t1.int_col` 一列的索引被删除了。如果加上这个索引，操作执行速度可以从 0.3 秒 提高到 0.06 秒，如下表所示：

```
-- 重新添加索引
ALTER TABLE t2 ADD INDEX (t1_id);

EXPLAIN ANALYZE SELECT /*+ INL_JOIN(t1, t2) */  * FROM t1 INNER JOIN t2 ON t1.id = t2.t1_id WHERE
    ↪  t1.int_col = 1;
EXPLAIN ANALYZE SELECT /*+ HASH_JOIN(t1, t2) */  * FROM t1 INNER JOIN t2 ON t1.id = t2.t1_id
    ↪ WHERE t1.int_col = 1;
EXPLAIN ANALYZE SELECT * FROM t1 INNER JOIN t2 ON t1.id = t2.t1_id WHERE t1.int_col = 1;
```

```
+--
    ↪ -------------------------------+----------+---------+-----------+---------------------------+--------
    ↪
| id                               | estRows  | actRows | task      | access object
    ↪                | execution info
    ↪
    ↪ | operator info
    ↪
    ↪ | memory    | disk |
+--
    ↪ -------------------------------+----------+---------+-----------+---------------------------+--------
    ↪
| IndexJoin_12                     | 90000.00 | 0       | root      |
    ↪                                | time:65.6ms, loops:1, inner:{total:129.7ms, concurrency:5,
    ↪  task:7, construct:7.13ms, fetch:122.5ms, build:16.4µs}, probe:2.54ms
    ↪
    ↪ | inner join, inner:IndexLookUp_11, outer key:test.t1.id, inner key:test.t2.t1_id, equal
    ↪ cond:eq(test.t1.id, test.t2.t1_id) | 28.7 MB   | N/A  |
```

```
| ├─TableReader_33(Build)          | 9955.54  | 10000   | root      |
    ↪                               | time:15.4ms, loops:16, cop_task: {num: 11, max: 1.52ms,
    ↪ min: 211.5µs, avg: 416.8µs, p95: 1.52ms, rpc_num: 11, rpc_time: 4.36ms,
    ↪ copr_cache_hit_ratio: 1.00, distsql_concurrency: 15}
    ↪
    ↪ | data:Selection_32
    ↪
    ↪ | 13.9 MB   | N/A  |
| | └─Selection_32                 | 9955.54  | 10000   | cop[tikv] |
    ↪                               | tikv_task:{proc max:104ms, min:3ms, avg: 24.4ms, p80:33ms,
    ↪  p95:104ms, iters:113, tasks:11}, scan_detail: {get_snapshot_time: 185µs, rocksdb: {block
    ↪ : {}}}
    ↪
    ↪ | eq(test.t1.int_col, 1)
    ↪
    ↪ | N/A       | N/A  |
| |   └─TableFullScan_31           | 71010.00 | 71010   | cop[tikv] | table:t1
    ↪                       | tikv_task:{proc max:101ms, min:3ms, avg: 23.8ms, p80:33ms, p95:101
    ↪ ms, iters:113, tasks:11}
    ↪
    ↪ | keep order:false
    ↪
    ↪ | N/A       | N/A  |
| └─IndexLookUp_11(Probe)          | 90000.00 | 0       | root      |
    ↪                               | time:115.6ms, loops:7
    ↪
    ↪ |
    ↪
    ↪ | 555 Bytes | N/A  |
|   ├─IndexRangeScan_9(Build)      | 90000.00 | 0       | cop[tikv] | table:t2, index:t1_id(
    ↪ t1_id) | time:114.3ms, loops:7, cop_task: {num: 7, max: 42ms, min: 1.3ms, avg: 16.2ms,
    ↪ p95: 42ms, tot_proc: 71ms, rpc_num: 7, rpc_time: 113.2ms, copr_cache_hit_ratio: 0.29,
    ↪ distsql_concurrency: 15}, tikv_task:{proc max:37ms, min:0s, avg: 11.3ms, p80:20ms, p95:37
    ↪ ms, iters:7, tasks:7}, scan_detail: {total_keys: 9296, get_snapshot_time: 141.9µs,
    ↪ rocksdb: {block: {cache_hit_count: 18592}}}  | range: decided by [eq(test.t2.t1_id, test.
    ↪ t1.id)], keep order:false                                                    | N/A
    ↪       | N/A  |
|   └─TableRowIDScan_10(Probe)     | 90000.00 | 0       | cop[tikv] | table:t2
    ↪                       |
    ↪
    ↪ | keep order:false
    ↪
    ↪ | N/A       | N/A  |
+--
    ↪ -------------------------------+----------+---------+-----------+---------------------------+--------
```

```
    ↪
+--
    ↪ ---------------------------+----------+---------+-----------+--------------+-------------------------
    ↪
| id                           | estRows  | actRows | task      | access object | execution info
    ↪
    ↪ | operator info                                    | memory  | disk    |
+--
    ↪ ---------------------------+----------+---------+-----------+--------------+-------------------------
    ↪
| HashJoin_32                  | 90000.00 | 0       | root      |               | time:320.2ms,
    ↪ loops:1, build_hash_table:{total:19.3ms, fetch:16.8ms, build:2.52ms}, probe:{concurrency
    ↪ :5, total:1.6s, max:320.1ms, probe:16.1ms, fetch:1.58s}
    ↪
    ↪ | inner join, equal:[eq(test.t1.id, test.t2.t1_id)] | 32.0 MB | 0 Bytes |
| ├─TableReader_35(Build)      | 9955.54  | 10000   | root      |               | time:18.6ms,
    ↪ loops:12, cop_task: {num: 11, max: 713.8µs, min: 197.3µs, avg: 368.5µs, p95: 713.8µs,
    ↪ rpc_num: 11, rpc_time: 3.83ms, copr_cache_hit_ratio: 1.00, distsql_concurrency: 15}
    ↪
    ↪ | data:Selection_34                                | 14.9 MB | N/A     |
| │ └─Selection_34             | 9955.54  | 10000   | cop[tikv] |               | tikv_task:{
    ↪ proc max:104ms, min:3ms, avg: 24.4ms, p80:33ms, p95:104ms, iters:113, tasks:11},
    ↪ scan_detail: {get_snapshot_time: 178.9µs, rocksdb: {block: {}}}
    ↪
    ↪ | eq(test.t1.int_col, 1)                           | N/A     | N/A     |
| │   └─TableFullScan_33       | 71010.00 | 71010   | cop[tikv] | table:t1      | tikv_task:{
    ↪ proc max:101ms, min:3ms, avg: 23.8ms, p80:33ms, p95:101ms, iters:113, tasks:11}
    ↪
    ↪ | keep order:false                                 | N/A     | N/A     |
| └─TableReader_37(Probe)      | 90000.00 | 90000   | root      |               | time:304.4ms,
    ↪ loops:91, cop_task: {num: 24, max: 114ms, min: 251.1µs, avg: 33.1ms, p95: 110.4ms,
    ↪ max_proc_keys: 10687, p95_proc_keys: 9184, tot_proc: 492ms, rpc_num: 24, rpc_time: 793ms,
    ↪  copr_cache_hit_ratio: 0.62, distsql_concurrency: 15}
    ↪                                                                                | data:
    ↪ TableFullScan_36                             | 58.6 MB | N/A     |
|   └─TableFullScan_36         | 90000.00 | 90000   | cop[tikv] | table:t2      | tikv_task:{
    ↪ proc max:38ms, min:3ms, avg: 14.1ms, p80:23ms, p95:35ms, iters:181, tasks:24},
    ↪ scan_detail: {total_process_keys: 69744, total_process_keys_size: 217533936, total_keys:
    ↪ 139497, get_snapshot_time: 577.2µs, rocksdb: {delete_skipped_count: 44208,
    ↪ key_skipped_count: 253431, block: {cache_hit_count: 3527}}}  | keep order:false
    ↪                                  | N/A     | N/A     |
+--
    ↪ ---------------------------+----------+---------+-----------+--------------+-------------------------
    ↪
```

```
+--
    ↪ ---------------------------+----------+---------+-----------+--------------+--------------------------------
    ↪
| id                           | estRows  | actRows | task      | access object | execution info
    ↪
    ↪ | operator info                                    | memory  | disk    |
+--
    ↪ ---------------------------+----------+---------+-----------+--------------+--------------------------------
    ↪
| HashJoin_33                  | 90000.00 | 0       | root      |               | time:306.3ms,
    ↪ loops:1, build_hash_table:{total:20.5ms, fetch:17.1ms, build:3.45ms}, probe:{concurrency
    ↪ :5, total:1.53s, max:305.9ms, probe:17.1ms, fetch:1.51s}
    ↪
    ↪ | inner join, equal:[eq(test.t1.id, test.t2.t1_id)] | 32.0 MB | 0 Bytes |
| ├─TableReader_42(Build)      | 9955.54  | 10000   | root      |               | time:19.6ms,
    ↪ loops:12, cop_task: {num: 11, max: 1.07ms, min: 246.1µs, avg: 600µs, p95: 1.07ms, rpc_num
    ↪ : 11, rpc_time: 6.17ms, copr_cache_hit_ratio: 1.00, distsql_concurrency: 15}
    ↪
    ↪ | data:Selection_41                                | 19.7 MB | N/A     |
| │ └─Selection_41             | 9955.54  | 10000   | cop[tikv] |               | tikv_task:{
    ↪ proc max:104ms, min:3ms, avg: 24.4ms, p80:33ms, p95:104ms, iters:113, tasks:11},
    ↪ scan_detail: {get_snapshot_time: 282.9µs, rocksdb: {block: {}}}
    ↪
    ↪ | eq(test.t1.int_col, 1)                           | N/A     | N/A     |
| │   └─TableFullScan_40       | 71010.00 | 71010   | cop[tikv] | table:t1      | tikv_task:{
    ↪ proc max:101ms, min:3ms, avg: 23.8ms, p80:33ms, p95:101ms, iters:113, tasks:11}
    ↪
    ↪ | keep order:false                                 | N/A     | N/A     |
| └─TableReader_44(Probe)      | 90000.00 | 90000   | root      |               | time:289.2ms,
    ↪ loops:91, cop_task: {num: 24, max: 108.2ms, min: 252.8µs, avg: 31.3ms, p95: 106.1ms,
    ↪ max_proc_keys: 10687, p95_proc_keys: 9184, tot_proc: 445ms, rpc_num: 24, rpc_time: 750.4
    ↪ ms, copr_cache_hit_ratio: 0.62, distsql_concurrency: 15}
    ↪                                                                          | data:
    ↪ TableFullScan_43                             | 58.6 MB | N/A     |
|   └─TableFullScan_43         | 90000.00 | 90000   | cop[tikv] | table:t2      | tikv_task:{
    ↪ proc max:31ms, min:3ms, avg: 13.3ms, p80:24ms, p95:30ms, iters:181, tasks:24},
    ↪ scan_detail: {total_process_keys: 69744, total_process_keys_size: 217533936, total_keys:
    ↪ 139497, get_snapshot_time: 730.2µs, rocksdb: {delete_skipped_count: 44208,
    ↪ key_skipped_count: 253431, block: {cache_hit_count: 3527}}}  | keep order:false
    ↪                                  | N/A     | N/A     |
+--
    ↪ ---------------------------+----------+---------+-----------+--------------+--------------------------------
    ↪
```

> **注意：**
>
> 在上方示例中，SQL 优化器之所以选择了性能较差的 Hash Join 算法，而不是 Index Join 算法，原因在于查询优化是一个 NP 完全问题，可能会选择不太理想的计划。如果需要频繁调用这个查询，建议通过执行计划管理的方式将 Hint 与 SQL 语句绑定，这样要比在发送给 TiDB 的 SQL 语句中插入 Hint 更容易管理。

Index Join 相关算法

如果使用 Hint `INL_JOIN` 进行 Index Join 操作，TiDB 会在连接外表之前创建一个中间结果的 Hash Table。TiDB 同样也支持使用 Hint `INL_HASH_JOIN` 在外表上建 Hash Table。以上所述的 Index Join 相关算法都由 SQL 优化器自动选择。

配置

Index Join 算法的性能受以下系统变量影响：

- `tidb_index_join_batch_size`（默认值：`25000`）- `index lookup join` 操作的 batch 大小。
- `tidb_index_lookup_join_concurrency`（默认值：`4`）- 可以并发执行的 index lookup 任务数。

11.3.2.5.2 Hash Join

在 Hash Join 操作中，TiDB 首先读取 Build 端的数据并将其缓存在 Hash Table 中，然后再读取 Probe 端的数据，使用 Probe 端的数据来探查 Hash Table 以获得所需行。与 Index Join 算法相比，Hash Join 要消耗更多内存，但如果需要连接的行数很多，运行速度会比 Index Join 快。TiDB 中的 Hash Join 算子是多线程的，并且可以并发执行。

下面是一个 Hash Join 示例：

```
EXPLAIN SELECT /*+ HASH_JOIN(t1, t2) */ * FROM t1, t2 WHERE t1.id = t2.id;
```

```
+--
    ↪ --------------------------+-----------+-----------+---------------+--------------------------------
    ↪
| id                          | estRows   | task      | access object | operator info
    ↪                                  |
+--
    ↪ --------------------------+-----------+-----------+---------------+--------------------------------
    ↪
| HashJoin_27                 | 142020.00 | root      |               | inner join, equal:[eq(
    ↪ test.t1.id, test.t2.id)] |
| ├─TableReader_29(Build)     | 142020.00 | root      |               | data:TableFullScan_28
    ↪                           |
| | └─TableFullScan_28        | 142020.00 | cop[tikv] | table:t1      | keep order:false
    ↪                                 |
| └─TableReader_31(Probe)     | 180000.00 | root      |               | data:TableFullScan_30
    ↪                           |
```

```
|   └─TableFullScan_30        | 180000.00 | cop[tikv] | table:t2      | keep order:false
    ↪                                |
+--
    ↪ --------------------------+-----------+-----------+---------------+--------------------------------
    ↪
5 rows in set (0.00 sec)
```

TiDB 会按照以下顺序执行 HashJoin_27 算子：

1. 将 Build 端数据缓存在内存中。
2. 根据缓存数据在 Build 端构造一个 Hash Table。
3. 读取 Probe 端的数据。
4. 使用 Probe 端的数据来探查 Hash Table。
5. 将符合条件的结果返回给用户。

EXPLAIN 返回结果中的 operator info 一列记录了 HashJoin_27 的其他信息，包括该查询是 Inner Join 还是 Outer Join 以及 Join 的条件是什么等。在上面给出的示例中，该查询为 Inner Join，Join 条件是 equal:[eq(test.t1.id, ↪ test.t2.id)]，与查询语句中的 WHERE t1.id = t2.id 部分对应。下面例子中其他几个 Join 算子的 operator info 和此处类似。

运行数据

如果在执行操作时，内存使用超过了tidb_mem_quota_query规定的值（默认为 1 GB），且tidb_enable_tmp_storage_on_oom ↪ 的值为 ON（默认为 ON），那么 TiDB 会尝试使用临时存储，在磁盘上创建 Hash Join 的 Build 端。EXPLAIN ↪ ANALYZE 返回结果中的 execution info 一栏记录了有关内存使用情况等运行数据。下面的例子展示了 tidb_mem_quota_query 的值分别设为 1 GB（默认）及 500 MB 时，EXPLAIN ANALYZE 的返回结果（当内存配额设为 500 MB 时，磁盘用作临时存储区）：

```
EXPLAIN ANALYZE SELECT /*+ HASH_JOIN(t1, t2) */ * FROM t1, t2 WHERE t1.id = t2.id;
SET tidb_mem_quota_query=500 * 1024 * 1024;
EXPLAIN ANALYZE SELECT /*+ HASH_JOIN(t1, t2) */ * FROM t1, t2 WHERE t1.id = t2.id;
```

```
+--
    ↪ --------------------------+-----------+---------+-----------+---------------+--------------------------
    ↪
| id                          | estRows   | actRows | task      | access object | execution info
    ↪
    ↪ | operator info                                  | memory                | disk    |
+--
    ↪ --------------------------+-----------+---------+-----------+---------------+--------------------------
    ↪
| HashJoin_27                 | 142020.00 | 71010   | root      |               | time:647.508572
    ↪ ms, loops:72, build_hash_table:{total:579.254415ms, fetch:566.91012ms, build:12.344295ms
    ↪ }, probe:{concurrency:5, total:3.23315006s, max:647.520113ms, probe:330.884716ms, fetch
    ↪ :2.902265344s}                                          | inner join, equal:[eq(test.
    ↪ t1.id, test.t2.id)] | 209.61642456054688 MB | 0 Bytes |
```

```
| ├─TableReader_29(Build)     | 142020.00 | 71010   | root      |               | time
    ↪ :567.088247ms, loops:72, cop_task: {num: 2, max: 569.809411ms, min: 369.67451ms, avg:
    ↪ 469.74196ms, p95: 569.809411ms, max_proc_keys: 39245, p95_proc_keys: 39245, tot_proc: 400
    ↪ ms, rpc_num: 2, rpc_time: 939.447231ms, copr_cache_hit_ratio: 0.00}   | data:
    ↪ TableFullScan_28                          | 210.2100534439087 MB  | N/A     |
| | └─TableFullScan_28        | 142020.00 | 71010   | cop[tikv] | table:t1      | proc max:64ms,
    ↪  min:48ms, p80:64ms, p95:64ms, iters:79, tasks:2
    ↪
    ↪ | keep order:false                                | N/A                   | N/A     |
| └─TableReader_31(Probe)     | 180000.00 | 90000   | root      |               | time
    ↪ :337.233636ms, loops:91, cop_task: {num: 3, max: 569.790741ms, min: 332.758911ms, avg:
    ↪ 421.543165ms, p95: 569.790741ms, max_proc_keys: 31719, p95_proc_keys: 31719, tot_proc:
    ↪ 500ms, rpc_num: 3, rpc_time: 1.264570696s, copr_cache_hit_ratio: 0.00} | data:
    ↪ TableFullScan_30                          | 267.1126985549927 MB  | N/A     |
|   └─TableFullScan_30        | 180000.00 | 90000   | cop[tikv] | table:t2      | proc max:84ms,
    ↪  min:72ms, p80:84ms, p95:84ms, iters:102, tasks:3
    ↪
    ↪ | keep order:false                                | N/A                   | N/A     |
+--
    ↪ ---------------------------+-----------+---------+-----------+---------------+-----------------------------
    ↪
5 rows in set (0.65 sec)

Query OK, 0 rows affected (0.00 sec)

+--
    ↪ ---------------------------+-----------+---------+-----------+---------------+-----------------------------
    ↪
| id                          | estRows   | actRows | task      | access object | execution info
    ↪
    ↪ | operator info                                   | memory                | disk
    ↪                 |
+--
    ↪ ---------------------------+-----------+---------+-----------+---------------+-----------------------------
    ↪
| HashJoin_27                 | 142020.00 | 71010   | root      |               | time:963.983353
    ↪ ms, loops:72, build_hash_table:{total:775.961447ms, fetch:503.789677ms, build:272.17177ms
    ↪ }, probe:{concurrency:5, total:4.805454793s, max:963.973133ms, probe:922.156835ms, fetch
    ↪ :3.883297958s}                                          | inner join, equal:[eq(test.t1.
    ↪ id, test.t2.id)] | 93.53974533081055 MB  | 210.7459259033203 MB |
| ├─TableReader_29(Build)     | 142020.00 | 71010   | root      |               | time
    ↪ :504.062018ms, loops:72, cop_task: {num: 2, max: 509.276857ms, min: 402.66386ms, avg:
    ↪ 455.970358ms, p95: 509.276857ms, max_proc_keys: 39245, p95_proc_keys: 39245, tot_proc:
    ↪ 384ms, rpc_num: 2, rpc_time: 911.893237ms, copr_cache_hit_ratio: 0.00}  | data:
    ↪ TableFullScan_28                          | 210.20934200286865 MB | N/A
```

```
    ↪ |
| | └─TableFullScan_28        | 142020.00 | 71010    | cop[tikv] | table:t1      | proc max:88ms,
    ↪  min:72ms, p80:88ms, p95:88ms, iters:79, tasks:2
    ↪
    ↪ | keep order:false                              | N/A                  | N/A
    ↪                  |
| └─TableReader_31(Probe)     | 180000.00 | 90000    | root      |               | time
    ↪ :363.058382ms, loops:91, cop_task: {num: 3, max: 412.659191ms, min: 358.489688ms, avg:
    ↪ 391.463008ms, p95: 412.659191ms, max_proc_keys: 31719, p95_proc_keys: 31719, tot_proc:
    ↪ 484ms, rpc_num: 3, rpc_time: 1.174326746s, copr_cache_hit_ratio: 0.00} | data:
    ↪ TableFullScan_30                          | 267.11340618133545 MB | N/A
    ↪ |
|   └─TableFullScan_30        | 180000.00 | 90000    | cop[tikv] | table:t2      | proc max:92ms,
    ↪  min:64ms, p80:92ms, p95:92ms, iters:102, tasks:3
    ↪
    ↪ | keep order:false                              | N/A                  | N/A
    ↪                  |
+--
    ↪ --------------------------+----------+---------+----------+--------------+-------------------------
    ↪
5 rows in set (0.98 sec)
```

配置

Hash Join 算法的性能受以下系统变量影响：

- tidb_mem_quota_query（默认值：1GB）- 如果某条查询的内存消耗超出了配额，TiDB 会尝试将 Hash Join 的 Build 端移到磁盘上以节省内存。
- tidb_hash_join_concurrency（默认值：5）- 可以并发执行的 Hash Join 任务数量。

11.3.2.5.3 Merge Join

Merge Join 是一种特殊的 Join 算法。当两个关联表要 Join 的字段需要按排好的顺序读取时，适用 Merge Join 算法。由于 Build 端和 Probe 端的数据都会读取，这种算法的 Join 操作是流式的，类似“拉链式合并”的高效版。Merge Join 占用的内存要远低于 Hash Join，但 Merge Join 不能并发执行。

下面是一个使用 Merge Join 的例子：

```
EXPLAIN SELECT /*+ MERGE_JOIN(t1, t2) */ * FROM t1, t2 WHERE t1.id = t2.id;
```

```
+--
    ↪ --------------------------+----------+----------+--------------+--------------------------------
    ↪
| id                          | estRows    | task      | access object | operator info
    ↪                                      |
+--
    ↪ --------------------------+----------+----------+--------------+--------------------------------
    ↪
```

```
| MergeJoin_7                 | 142020.00 | root      |               | inner join, left key:test
    ↪ .t1.id, right key:test.t2.id |
| ├─TableReader_12(Build)     | 180000.00 | root      |               | data:TableFullScan_11
    ↪                                 |
| │ └─TableFullScan_11        | 180000.00 | cop[tikv] | table:t2      | keep order:true
    ↪                                   |
| └─TableReader_10(Probe)     | 142020.00 | root      |               | data:TableFullScan_9
    ↪                                  |
|   └─TableFullScan_9         | 142020.00 | cop[tikv] | table:t1      | keep order:true
    ↪                                   |
+--
    ↪ --------------------------+-----------+-----------+---------------+-----------------------------------
    ↪
5 rows in set (0.00 sec)
```

TiDB 会按照以下顺序执行 Merge Join 算子：

1. 从 Build 端把一个 Join Group 的数据全部读取到内存中。
2. 读取 Probe 端的数据。
3. 将 Probe 端的每行数据与 Build 端的一个完整 Join Group 比较，依次查看是否匹配（除了满足等值条件以外，还有其他非等值条件，这里的“匹配”主要是指查看是否满足非等值条件）。Join Group 指的是所有 Join Key 上值相同的数据。

11.3.2.5.4 其他类型查询的执行计划

- MPP 模式查询的执行计划
- 索引查询的执行计划
- 子查询的执行计划
- 聚合查询的执行计划
- 视图查询的执行计划
- 分区查询的执行计划
- 索引合并查询的执行计划

11.3.2.6 用 EXPLAIN 查看子查询的执行计划

TiDB 会执行多种子查询相关的优化，以提升子查询的执行性能。本文档介绍一些常见子查询的优化方式，以及如何解读 EXPLAIN 语句返回的执行计划信息。

本文档所使用的示例表数据如下：

```
CREATE TABLE t1 (id BIGINT NOT NULL PRIMARY KEY auto_increment, pad1 BLOB, pad2 BLOB, pad3 BLOB,
    ↪ int_col INT NOT NULL DEFAULT 0);
CREATE TABLE t2 (id BIGINT NOT NULL PRIMARY KEY auto_increment, t1_id BIGINT NOT NULL, pad1 BLOB,
    ↪  pad2 BLOB, pad3 BLOB, INDEX(t1_id));
CREATE TABLE t3 (
 id INT NOT NULL PRIMARY KEY auto_increment,
```

```
 t1_id INT NOT NULL,
 UNIQUE (t1_id)
);

INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM
    ↪ dual;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
UPDATE t1 SET int_col = 1 WHERE pad1 = (SELECT pad1 FROM t1 ORDER BY RAND() LIMIT 1);
```

```
INSERT INTO t3 SELECT NULL, id FROM t1 WHERE id < 1000;

SELECT SLEEP(1);
ANALYZE TABLE t1, t2, t3;
```

11.3.2.6.1 Inner join（无 UNIQUE 约束的子查询）

以下示例中，IN 子查询会从表 t2 中搜索一列 ID。为保证语义正确性，TiDB 需要保证 t1_id 列的值具有唯一性。使用 EXPLAIN 可查看到该查询的执行计划去掉重复项并执行 Inner Join 内连接操作：

```
EXPLAIN SELECT * FROM t1 WHERE id IN (SELECT t1_id FROM t2);
```

```
+--
    ↪ ------------------------------+----------+-----------+------------------------------+-------------------
    ↪
| id                            | estRows  | task      | access object                | operator
    ↪  info
    ↪
    ↪ |
+--
    ↪ ------------------------------+----------+-----------+------------------------------+-------------------
    ↪
| IndexJoin_15                  | 21.11    | root      |                              | inner
    ↪ join, inner:TableReader_12, outer key:test.t2.t1_id, inner key:test.t1.id, equal cond:eq(
    ↪ test.t2.t1_id, test.t1.id) |
| ├─StreamAgg_44(Build)         | 21.11    | root      |                              | group
    ↪ by:test.t2.t1_id, funcs:firstrow(test.t2.t1_id)->test.t2.t1_id
    ↪                                                  |
| │ └─IndexReader_45            | 21.11    | root      |                              | index:
    ↪ StreamAgg_34
    ↪
    ↪ |
| │   └─StreamAgg_34            | 21.11    | cop[tikv] |                              | group
    ↪ by:test.t2.t1_id,
    ↪
    ↪ |
| │     └─IndexFullScan_26      | 90000.00 | cop[tikv] | table:t2, index:t1_id(t1_id) | keep
    ↪ order:true
    ↪
    ↪ |
| └─TableReader_12(Probe)       | 21.11    | root      |                              | data:
    ↪ TableRangeScan_11
    ↪
    ↪ |
```

```
|   └─TableRangeScan_11          | 21.11    | cop[tikv] | table:t1                    | range:
    ↪ decided by [test.t2.t1_id], keep order:false
    ↪                                                                                  |
+--
    ↪ ----------------------------+----------+-----------+-----------------------------+-----------------
    ↪
```

由上述查询结果可知，TiDB 通过索引连接操作 | IndexJoin_14 将子查询做了连接转化。该执行计划首先在 TiKV 侧通过索引扫描算子 └─IndexFullScan_31 读取 t2.t1_id 列的值，然后由 └─StreamAgg_39 算子的部分任务在 TiKV 中对 t1_id 值进行去重，然后采用 ├─StreamAgg_49(Build) 算子的部分任务在 TiDB 中对 t1_id 值再次进行去重，去重操作由聚合函数 firstrow(test.t2.t1_id) 执行；之后将操作结果与 t1 表的主键相连接，连接条件是 eq(test.t1.id, test.t2.t1_id)。

11.3.2.6.2 Inner join（有 UNIQUE 约束的子查询）

在上述示例中，为了确保 t1_id 值在与表 t1 连接前具有唯一性，需要执行聚合运算。在以下示例中，由于 UNIQUE 约束已能确保 t3.t1_id 列值的唯一：

```
EXPLAIN SELECT * FROM t1 WHERE id IN (SELECT t1_id FROM t3);
```

```
+--
    ↪ ----------------------------+---------+-----------+-----------------------------+-----------------
    ↪
| id                            | estRows | task      | access object               | operator
    ↪ info
    ↪
    ↪ |
+--
    ↪ ----------------------------+---------+-----------+-----------------------------+-----------------
    ↪
| IndexJoin_18                  | 999.00  | root      |                             | inner join,
    ↪ inner:TableReader_15, outer key:test.t3.t1_id, inner key:test.t1.id, equal cond:eq(test.
    ↪ t3.t1_id, test.t1.id) |
| ├─IndexReader_41(Build)       | 999.00  | root      |                             | index:
    ↪ IndexFullScan_40
    ↪
    ↪ |
| │ └─IndexFullScan_40          | 999.00  | cop[tikv] | table:t3, index:t1_id(t1_id) | keep order:
    ↪ false
    ↪
    ↪ |
| └─TableReader_15(Probe)       | 999.00  | root      |                             | data:
    ↪ TableRangeScan_14
    ↪
    ↪ |
```

```
|   └─TableRangeScan_14       | 999.00   | cop[tikv] | table:t1                    | range:
    ↪ decided by [test.t3.t1_id], keep order:false
    ↪                                                                               |
+--
    ↪ -------------------------+---------+-----------+-----------------------------+-----------------------
    ↪
```

从语义上看，因为约束保证了 t3.t1_id 列值的唯一性，TiDB 可以直接执行 INNER JOIN 查询。

11.3.2.6.3 Semi Join（关联查询）

在前两个示例中，通过 StreamAgg 聚合操作或通过 UNIQUE 约束保证子查询数据的唯一性之后，TiDB 才能够执行 Inner Join 操作。这两种连接均使用了 Index Join。

下面的例子中，TiDB 优化器则选择了一种不同的执行计划：

```
EXPLAIN SELECT * FROM t1 WHERE id IN (SELECT t1_id FROM t2 WHERE t1_id != t1.int_col);
```

```
+--
    ↪ -------------------------+----------+-----------+-----------------------------+-----------------------
    ↪
| id                          | estRows  | task      | access object               | operator
    ↪ info
    ↪
    ↪ |
+--
    ↪ -------------------------+----------+-----------+-----------------------------+-----------------------
    ↪
| MergeJoin_9                 | 45446.40 | root      |                             | semi join,
    ↪ left key:test.t1.id, right key:test.t2.t1_id, other cond:ne(test.t2.t1_id, test.t1.
    ↪ int_col) |
| ├─IndexReader_24(Build)     | 90000.00 | root      |                             | index:
    ↪ IndexFullScan_23
    ↪                                                                                 |
| │ └─IndexFullScan_23        | 90000.00 | cop[tikv] | table:t2, index:t1_id(t1_id) | keep order
    ↪ :true
    ↪                                                                                       |
| └─TableReader_22(Probe)     | 56808.00 | root      |                             | data:
    ↪ Selection_21
    ↪                                                                                      |
|   └─Selection_21            | 56808.00 | cop[tikv] |                             | ne(test.t1
    ↪ .id, test.t1.int_col)
    ↪                                                                           |
|     └─TableFullScan_20      | 71010.00 | cop[tikv] | table:t1                    | keep order
    ↪ :true
    ↪                                                                                 |
```

```
+--
    ↪ --------------------------+----------+-----------+------------------------------+-----------------------
    ↪
```

由上述查询结果可知，TiDB 执行了 Semi Join。不同于 Inner Join，Semi Join 仅允许右键 (t2.t1_id) 上的第一个值，也就是该操作将去除 Join 算子任务中的重复数据。Join 算法也包含 Merge Join，会按照排序顺序同时从左侧和右侧读取数据，这是一种高效的 Zipper Merge。

可以将原语句视为关联子查询，因为它引入了子查询外的 t1.int_col 列。然而，EXPLAIN 语句的返回结果显示的是关联子查询去关联后的执行计划。条件 t1_id != t1.int_col 会被重写为 t1.id != t1.int_col。TiDB 可以从表 t1 中读取数据并且在 └─Selection_21 中执行此操作，因此这种去关联和重写操作会极大提高执行效率。

11.3.2.6.4 Anti Semi Join（NOT IN 子查询）

在以下示例中，除非子查询中存在 t3.t1_id，否则该查询将（从语义上）返回表 t3 中的所有行：

```
EXPLAIN SELECT * FROM t3 WHERE t1_id NOT IN (SELECT id FROM t1 WHERE int_col < 100);
```

```
+--
    ↪ --------------------------+---------+-----------+--------------+-------------------------------------
    ↪
| id                          | estRows | task      | access object | operator info
    ↪
    ↪ |
+--
    ↪ --------------------------+---------+-----------+--------------+-------------------------------------
    ↪
| IndexJoin_16                | 799.20  | root      |               | anti semi join, inner:
    ↪ TableReader_12, outer key:test.t3.t1_id, inner key:test.t1.id, equal cond:eq(test.t3.
    ↪ t1_id, test.t1.id) |
| ├─TableReader_28(Build)     | 999.00  | root      |               | data:TableFullScan_27
    ↪
    ↪ |
| │ └─TableFullScan_27        | 999.00  | cop[tikv] | table:t3      | keep order:false
    ↪
    ↪ |
| └─TableReader_12(Probe)     | 999.00  | root      |               | data:Selection_11
    ↪
    ↪ |
|   └─Selection_11            | 999.00  | cop[tikv] |               | lt(test.t1.int_col, 100)
    ↪
    ↪ |
|     └─TableRangeScan_10     | 999.00  | cop[tikv] | table:t1      | range: decided by [test.t3
    ↪ .t1_id], keep order:false
    ↪                                                                                          |
```

```
+--
    ↪ -------------------------+---------+-----------+---------------+-------------------------------------
    ↪
```

上述查询首先读取了表 t3，然后根据主键开始探测 (probe) 表 t1。连接类型是 anti semi join，即反半连接：之所以使用 anti，是因为上述示例有不存在匹配值（即 NOT IN）的情况；使用 Semi Join 则是因为仅需要匹配第一行后就可以停止查询。

11.3.2.6.5 Null-Aware Semi Join（IN 和 = ANY 子查询）

IN 和 = ANY 的集合运算符号具有特殊的三值属性（true、false 和 NULL）。这意味着在该运算符所转化得到的 Join 类型中需要对 Join key 两侧的 NULL 进行特殊的感知和处理。

IN 和 = ANY 算子引导的子查询会分别转为 Semi Join 和 Left Outer Semi Join。在上述Semi Join 小节中，示例中 Join key 两侧的列 test.t1.id 和 test.t2.t1_id 都为 not NULL 属性，所以 Semi Join 本身不需要 Null-Aware 的性质来辅助运算，即不需要特殊处理 NULL。当前 TiDB 对于 Null-Aware Semi Join 没有特定的优化，其实现本质都是基于笛卡尔积加过滤 (filter) 的模式。以下为 Null-Aware Semi Join 的例子：

```
CREATE TABLE t(a INT, b INT);
CREATE TABLE s(a INT, b INT);
EXPLAIN SELECT (a,b) IN (SELECT * FROM s) FROM t;
EXPLAIN SELECT * FROM t WHERE (a,b) IN (SELECT * FROM s);
```

```
tidb> EXPLAIN SELECT (a,b) IN (SELECT * FROM s) FROM t;
+--
    ↪ -------------------------+---------+-----------+---------------+-------------------------------------
    ↪
| id                          | estRows | task      | access object | operator info
    ↪                                                                                 |
+--
    ↪ -------------------------+---------+-----------+---------------+-------------------------------------
    ↪
| HashJoin_8                  | 1.00    | root      |               | CARTESIAN left outer semi
    ↪ join, other cond:eq(test.t.a, test.s.a), eq(test.t.b, test.s.b) |
| ├─TableReader_12(Build)     | 1.00    | root      |               | data:TableFullScan_11
    ↪                                                                       |
| │ └─TableFullScan_11        | 1.00    | cop[tikv] | table:s       | keep order:false, stats:
    ↪ pseudo                                                                |
| └─TableReader_10(Probe)     | 1.00    | root      |               | data:TableFullScan_9
    ↪                                                                       |
|   └─TableFullScan_9         | 1.00    | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo                                                                |
+--
    ↪ -------------------------+---------+-----------+---------------+-------------------------------------
    ↪
5 rows in set (0.00 sec)
```

```
tidb> EXPLAIN SELECT * FROM t WHERE (a,b) IN (SELECT * FROM s);
+--
    ↪ ---------------------------+---------+-----------+--------------+-------------------------------
    ↪
| id                           | estRows | task      | access object | operator info
    ↪                                                                                              |
+--
    ↪ ---------------------------+---------+-----------+--------------+-------------------------------
    ↪
| HashJoin_11                  | 1.00    | root      |               | inner join, equal:[eq(test
    ↪ .t.a, test.s.a) eq(test.t.b, test.s.b)]                                      |
| ├─TableReader_14(Build)      | 1.00    | root      |               | data:Selection_13
    ↪                                                                                             |
| │ └─Selection_13             | 1.00    | cop[tikv] |               | not(isnull(test.t.a)),
    ↪ not(isnull(test.t.b))                                                                        |
| │   └─TableFullScan_12       | 1.00    | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo                                                                                      |
| └─HashAgg_17(Probe)          | 1.00    | root      |               | group by:test.s.a, test.s
    ↪ .b, funcs:firstrow(test.s.a)->test.s.a, funcs:firstrow(test.s.b)->test.s.b |
|   └─TableReader_24           | 1.00    | root      |               | data:Selection_23
    ↪                                                                                             |
|     └─Selection_23           | 1.00    | cop[tikv] |               | not(isnull(test.s.a)),
    ↪ not(isnull(test.s.b))                                                                        |
|       └─TableFullScan_22     | 1.00    | cop[tikv] | table:s       | keep order:false, stats:
    ↪ pseudo                                                                                      |
+--
    ↪ ---------------------------+---------+-----------+--------------+-------------------------------
    ↪
8 rows in set (0.01 sec)
```

第一个查询 `EXPLAIN SELECT (a,b)IN (SELECT * FROM s)FROM t;` 中，由于 t 表和 s 表的 a、b 列都是 NULLABLE 的，所以 IN 子查询所转化的 Left Outer Semi Join 是具有 Null-Aware 性质的。具体实现是先进行笛卡尔积，然后将 IN 或 = ANY 所连接的列作为普通等值条件放到 other condition 进行过滤（filter）。

第二个查询 `EXPLAIN SELECT * FROM t WHERE (a,b)IN (SELECT * FROM s);` 中，由于 t 表和 s 表的 a、b 列都是 NULLABLE 的，IN 子查询本应该转为具有 Null-Aware 性质的 Semi Join，但当前 TiDB 进行了优化，直接将 Semi Join 转为了 Inner Join + Aggregate 的方式来实现。这是因为在非 scalar 输出的 IN 子查询中，NULL 和 `false` 是等效的。下推过滤的 NULL 行导致了 WHERE 子句的否定语义，因此可以事先忽略这些行。

> **注意：**
>
> Exists 操作符也会被转成 Semi Join，但是 Exists 操作符号本身不具有集合运算 Null-Aware 的性质。

11.3.2.6.6 Null-Aware Anti Semi Join（NOT IN 和 != ALL 子查询）

NOT IN 和 != ALL 的集合运算运算具有特殊的三值属性（true、false 和 NULL）。这意味着在其所转化得到的 Join 类型中需要对 Join key 两侧的 NULL 进行特殊的感知和处理。

NOT IN 和 != ALL 算子引导的子查询会对应地转为 Anti Semi Join 和 Anti Left Outer Semi Join。在上述的Anti Semi Join 小节中，由于示例中 Join key 两侧的列 test.t3.t1_id 和 test.t1.id 都是 not NULL 属性的，所以 Anti Semi Join 本身不需要 Null-Aware 的性质来辅助计算，即不需要特殊处理 NULL。

在 TiDB v6.3.0 版本，TiDB 引入了针对 Null-Aware Anti Join (NAAJ) 的如下特殊优化：

- 利用 Null-Aware 的等值条件 (NA-EQ) 构建哈希连接

  由于集合操作符引入的等值需要对等值两侧操作符数的 NULL 值做特殊处理，这里称需要 Null-Aware 的等值条件为 NA-EQ 条件。与 v6.3.0 之前版本不同的是，TiDB 不会再将 NA-EQ 条件处理成普通 EQ 条件，而是专门放置于 Join 后置的 other condition 中，匹配笛卡尔积后再判断结果集的合法性。

  在 TiDB v6.3.0 版本中，NA-EQ 这种弱化的等值条件依然会被用来构建哈希值 (Hash Join)，大大减少了匹配时所需遍历的数据量，加速匹配过程。在 build 表 DISTINCT 值比例趋近 1 的时候，加速效果更为显著。

- 利用两侧数据源 NULL 值的特殊性质加速匹配过程的返回

  由于 Anti Semi Join 自身具有 CNF (Conjunctive normal form) 表达式的属性，其任何一侧出现的 NULL 值都会导致确定的结果。利用这个性质可以来加速整个匹配过程。

以下为 Null-Aware Anti Semi Join 的例子：

```
CREATE TABLE t(a INT, b INT);
CREATE TABLE s(a INT, b INT);
EXPLAIN SELECT (a, b) NOT IN (SELECT * FROM s) FROM t;
EXPLAIN SELECT * FROM t WHERE (a, b) NOT IN (SELECT * FROM s);
```

```
tidb> EXPLAIN SELECT (a, b) NOT IN (SELECT * FROM s) FROM t;
+--
    ↪ --------------------------+----------+-----------+---------------+--------------------------------
    ↪
| id                          | estRows  | task      | access object | operator info
    ↪                                                                                                  |
+--
    ↪ --------------------------+----------+-----------+---------------+--------------------------------
    ↪
| HashJoin_8                  | 10000.00 | root      |               | Null-aware anti left outer
    ↪  semi join, equal:[eq(test.t.b, test.s.b) eq(test.t.a, test.s.a)] |
| ├─TableReader_12(Build)     | 10000.00 | root      |               | data:TableFullScan_11
    ↪                                                                                                  |
| │ └─TableFullScan_11        | 10000.00 | cop[tikv] | table:s       | keep order:false, stats:
    ↪ pseudo                                                                                           |
| └─TableReader_10(Probe)     | 10000.00 | root      |               | data:TableFullScan_9
    ↪                                                                                                  |
```

```
|   └─TableFullScan_9         | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo                                                                  |
+--
    ↪ --------------------------+----------+-----------+---------------+-------------------------------------
    ↪
5 rows in set (0.00 sec)

tidb> EXPLAIN SELECT * FROM t WHERE (a, b) NOT IN (SELECT * FROM s);
+--
    ↪ --------------------------+----------+-----------+---------------+-------------------------------------
    ↪
| id                          | estRows  | task      | access object | operator info
    ↪                                                                      |
+--
    ↪ --------------------------+----------+-----------+---------------+-------------------------------------
    ↪
| HashJoin_8                  | 8000.00  | root      |               | Null-aware anti semi join,
    ↪  equal:[eq(test.t.b, test.s.b) eq(test.t.a, test.s.a)] |
| ├─TableReader_12(Build)     | 10000.00 | root      |               | data:TableFullScan_11
    ↪                                                                 |
| │ └─TableFullScan_11        | 10000.00 | cop[tikv] | table:s       | keep order:false, stats:
    ↪ pseudo                                                        |
| └─TableReader_10(Probe)     | 10000.00 | root      |               | data:TableFullScan_9
    ↪                                                                 |
|   └─TableFullScan_9         | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo                                                        |
+--
    ↪ --------------------------+----------+-----------+---------------+-------------------------------------
    ↪
5 rows in set (0.00 sec)
```

第一个查询 `EXPLAIN SELECT (a, b)NOT IN (SELECT * FROM s)FROM t;` 中，由于 `t` 表和 `s` 表的 `a`、`b` 列都是 NULLABLE 的，所以 `NOT IN` 子查询所转化的 Left Outer Semi Join 是具有 Null-Aware 性质的。不同的是，NAAJ 优化将 NA-EQ 条件也作为了 Hash Join 的连接条件，大大加速了 Join 的计算。

第二个查询 `EXPLAIN SELECT * FROM t WHERE (a, b)NOT IN (SELECT * FROM s);` 中，由于 `t` 表和 `s` 表的 `a`、`b` 列都是 NULLABLE 的，所以 `NOT IN` 子查询所转化的 Anti Semi Join 是具有 Null-Aware 性质的。不同的是，NAAJ 优化将 NA-EQ 条件也作为了 Hash Join 的连接条件，大大加速了 Join 的计算。

当前 TiDB 仅针对 Anti Semi Join 和 Anti Left Outer Semi Join 实现了 `NULL` 感知。目前仅支持 Hash Join 类型且其 build 表只能固定为右侧表。

> **注意：**
>
> `Not Exists` 操作符也会被转成 Anti Semi Join，但是 `Not Exists` 符号本身不具有集合运算 Null-Aware 的性质。

11.3.2.6.7 其他类型查询的执行计划

- MPP 模式查询的执行计划
- 索引查询的执行计划
- Join 查询的执行计划
- 聚合查询的执行计划
- 视图查询的执行计划
- 分区查询的执行计划
- 索引合并查询的执行计划

11.3.2.7 用 EXPLAIN 查看聚合查询执行计划

SQL 查询中可能会使用聚合计算，可以通过 EXPLAIN 语句来查看聚合查询的执行计划。本文提供多个示例，以帮助用户理解聚合查询是如何执行的。

SQL 优化器会选择以下任一算子实现数据聚合：

- Hash Aggregation
- Stream Aggregation

为了提高查询效率，数据聚合在 Coprocessor 层和 TiDB 层均会执行。现有示例如下：

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY auto_increment, pad1 BLOB, pad2 BLOB, pad3 BLOB);
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM dual;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    ↪ JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    ↪ JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    ↪ JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    ↪ JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    ↪ JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    ↪ JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    ↪ JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    ↪ JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    ↪ JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    ↪ JOIN t1 b JOIN t1 c LIMIT 10000;
```

```
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    ↪ JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    ↪ JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    ↪ JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    ↪ JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    ↪ JOIN t1 b JOIN t1 c LIMIT 10000;
SELECT SLEEP(1);
ANALYZE TABLE t1;
```

以上示例创建表格 `t1` 并插入数据后，再执行`SHOW TABLE REGIONS` 语句。从以下 `SHOW TABLE REGIONS` 的执行结果可知，表 `t1` 被切分为多个 Region：

```
SHOW TABLE t1 REGIONS;
```

```
+--
    ↪ ---------+--------------+--------------+-----------+-----------------+-------+------------+--------------
    ↪
| REGION_ID | START_KEY     | END_KEY       | LEADER_ID | LEADER_STORE_ID | PEERS | SCATTERING |
    ↪ WRITTEN_BYTES | READ_BYTES | APPROXIMATE_SIZE(MB) | APPROXIMATE_KEYS |
+--
    ↪ ---------+--------------+--------------+-----------+-----------------+-------+------------+--------------
    ↪
|        64 | t_64_         | t_64_r_31766 |        65 |               1 | 65    |          0 |
    ↪           1325 |  102033520 |                   98 |            52797 |
|        66 | t_64_r_31766 | t_64_r_63531 |        67 |               1 | 67    |          0 |
    ↪           1325 |   72522521 |                  104 |            78495 |
|        68 | t_64_r_63531 | t_64_r_95296 |        69 |               1 | 69    |          0 |
    ↪           1325 |          0 |                  104 |            95433 |
|         2 | t_64_r_95296 |              |         3 |               1 | 3     |          0 |
    ↪           1501 |          0 |                   81 |            63211 |
+--
    ↪ ---------+--------------+--------------+-----------+-----------------+-------+------------+--------------
    ↪
4 rows in set (0.00 sec)
```

使用 `EXPLAIN` 查看以下聚合语句的执行计划。可以看到 `└─StreamAgg_8` 算子先执行在 TiKV 内每个 Region 上，然后 TiKV 的每个 Region 会返回一行数据给 TiDB，TiDB 在 `StreamAgg_16` 算子上对每个 Region 返回的数据进行聚合：

```
EXPLAIN SELECT COUNT(*) FROM t1;
```

```
+--
    ↪ -------------------------+-----------+-----------+---------------+--------------------------------+
    ↪
```

```
| id                          | estRows    | task      | access object | operator info
    ↪                    |
+--
    ↪ -------------------------+-----------+-----------+---------------+-------------------------------+
    ↪
| StreamAgg_16                | 1.00       | root      |               | funcs:count(Column#7)->
    ↪ Column#5 |
| └─TableReader_17            | 1.00       | root      |               | data:StreamAgg_8
    ↪                  |
|   └─StreamAgg_8             | 1.00       | cop[tikv] |               | funcs:count(1)->Column#7
    ↪         |
|     └─TableFullScan_15      | 242020.00 | cop[tikv] | table:t1      | keep order:false
    ↪                  |
+--
    ↪ -------------------------+-----------+-----------+---------------+-------------------------------+
    ↪
4 rows in set (0.00 sec)
```

同样，通过执行 EXPLAIN ANALYZE 语句可知，actRows 与 SHOW TABLE REGIONS 返回结果中的 Region 数匹配，这是因为执行使用了 TableFullScan 全表扫并且没有二级索引：

```
EXPLAIN ANALYZE SELECT COUNT(*) FROM t1;
```

```
+--
    ↪ -------------------------+-----------+---------+-----------+---------------+--------------------------------
    ↪
| id                          | estRows    | actRows | task      | access object | execution info
    ↪
    ↪ | operator info                   | memory    | disk |
+--
    ↪ -------------------------+-----------+---------+-----------+---------------+--------------------------------
    ↪
| StreamAgg_16                | 1.00       | 1       | root      |               | time:12.609575ms
    ↪ , loops:2
    ↪
    ↪ | funcs:count(Column#7)->Column#5 | 372 Bytes | N/A  |
| └─TableReader_17            | 1.00       | 4       | root      |               | time:12.605155
    ↪ ms, loops:2, cop_task: {num: 4, max: 12.538245ms, min: 9.256838ms, avg: 10.895114ms, p95:
    ↪  12.538245ms, max_proc_keys: 31765, p95_proc_keys: 31765, tot_proc: 48ms, rpc_num: 4,
    ↪ rpc_time: 43.530707ms, copr_cache_hit_ratio: 0.00} | data:StreamAgg_8               |
    ↪ 293 Bytes | N/A  |
|   └─StreamAgg_8             | 1.00       | 4       | cop[tikv] |               | proc max:12ms,
    ↪ min:12ms, p80:12ms, p95:12ms, iters:122, tasks:4
    ↪
    ↪ | funcs:count(1)->Column#7        | N/A       | N/A  |
```

```
|     └─TableFullScan_15     | 242020.00 | 121010  | cop[tikv] | table:t1      | proc max:12ms,
    ↪ min:12ms, p80:12ms, p95:12ms, iters:122, tasks:4
    ↪
    ↪ | keep order:false               | N/A       | N/A  |
+--
    ↪ -------------------------+----------+---------+----------+--------------+-------------------------------
    ↪
4 rows in set (0.01 sec)
```

11.3.2.7.1 Hash Aggregation

Hash Aggregation 算法在执行聚合时使用 Hash 表存储中间结果。此算法采用多线程并发优化，执行速度快，但与 Stream Aggregation 算法相比会消耗较多内存。

下面是一个使用 Hash Aggregation（即 HashAgg 算子）的例子：

```
EXPLAIN SELECT /*+ HASH_AGG() */ count(*) FROM t1;
```

```
+--
    ↪ -------------------------+----------+----------+--------------+---------------------------+
    ↪
| id                        | estRows   | task      | access object | operator info
    ↪                   |
+--
    ↪ -------------------------+----------+----------+--------------+---------------------------+
    ↪
| HashAgg_9                 | 1.00      | root      |               | funcs:count(Column#6)->
    ↪ Column#5 |
| └─TableReader_10          | 1.00      | root      |               | data:HashAgg_5
    ↪                   |
|   └─HashAgg_5             | 1.00      | cop[tikv] |               | funcs:count(1)->Column#6
    ↪         |
|     └─TableFullScan_8     | 242020.00 | cop[tikv] | table:t1      | keep order:false
    ↪                 |
+--
    ↪ -------------------------+----------+----------+--------------+---------------------------+
    ↪
4 rows in set (0.00 sec)
```

`operator info` 列显示，用于聚合数据的 Hash 函数为 `funcs:count(1)->Column#6`。

11.3.2.7.2 Stream Aggregation

Stream Aggregation 算法通常会比 Hash Aggregation 算法占用更少的内存。但是此算法要求数据按顺序发送，以便对依次到达的值实现流式数据聚合。

下面是一个使用 Stream Aggregation 的例子：

```
CREATE TABLE t2 (id INT NOT NULL PRIMARY KEY, col1 INT NOT NULL);
INSERT INTO t2 VALUES (1, 9),(2, 3),(3,1),(4,8),(6,3);
EXPLAIN SELECT /*+ STREAM_AGG() */ col1, count(*) FROM t2 GROUP BY col1;
```

```
Query OK, 0 rows affected (0.11 sec)

Query OK, 5 rows affected (0.01 sec)
Records: 5  Duplicates: 0  Warnings: 0

+--
    ↪ ---------------------------+----------+-----------+---------------+--------------------------------
    ↪
| id                           | estRows  | task      | access object | operator info
    ↪                                                                                                    |
+--
    ↪ ---------------------------+----------+-----------+---------------+--------------------------------
    ↪
| Projection_4                 | 8000.00  | root      |               | test.t2.col1, Column#3
    ↪                                                                                                    |
| └─StreamAgg_8                | 8000.00  | root      |               | group by:test.t2.col1,
    ↪ funcs:count(1)->Column#3, funcs:firstrow(test.t2.col1)->test.t2.col1 |
|   └─Sort_13                  | 10000.00 | root      |               | test.t2.col1
    ↪                                                                                                    |
|     └─TableReader_12         | 10000.00 | root      |               | data:TableFullScan_11
    ↪                                                                                                    |
|       └─TableFullScan_11     | 10000.00 | cop[tikv] | table:t2      | keep order:false, stats:
    ↪ pseudo                                                                                             |
+--
    ↪ ---------------------------+----------+-----------+---------------+--------------------------------
    ↪
5 rows in set (0.00 sec)
```

以上示例中，可以在 col1 上添加索引来消除 └─Sort_13 算子。添加索引后，TiDB 就可以按顺序读取数据并消除 └─Sort_13 算子。

```
ALTER TABLE t2 ADD INDEX (col1);
EXPLAIN SELECT /*+ STREAM_AGG() */ col1, count(*) FROM t2 GROUP BY col1;
```

```
Query OK, 0 rows affected (0.28 sec)

+--
    ↪ ---------------------------+---------+-----------+---------------------------+----------------------
    ↪
| id                           | estRows | task      | access object             | operator info
    ↪                                                                                                  |
```

```
+--
    ↪ ---------------------------+---------+-----------+---------------------------+-----------------------
    ↪
| Projection_4                 | 4.00    | root      |                           | test.t2.col1,
    ↪  Column#3                                                                                              |
| └─StreamAgg_14               | 4.00    | root      |                           | group by:
    ↪ test.t2.col1, funcs:count(Column#4)->Column#3, funcs:firstrow(test.t2.col1)->test.t2.col1
    ↪  |
|   └─IndexReader_15           | 4.00    | root      |                           | index:
    ↪ StreamAgg_8
    ↪                                                                                                        |
|     └─StreamAgg_8            | 4.00    | cop[tikv] |                           | group by:
    ↪ test.t2.col1, funcs:count(1)->Column#4
    ↪                                                       |
|       └─IndexFullScan_13     | 5.00    | cop[tikv] | table:t2, index:col1(col1) | keep order:
    ↪ true, stats:pseudo                                                                                      |
+--
    ↪ ---------------------------+---------+-----------+---------------------------+-----------------------
    ↪
5 rows in set (0.00 sec)
```

11.3.2.7.3 其他类型查询的执行计划

- MPP 模式查询的执行计划
- 索引查询的执行计划
- Join 查询的执行计划
- 子查询的执行计划
- 视图查询的执行计划
- 分区查询的执行计划
- 索引合并查询的执行计划

11.3.2.8 用 EXPLAIN 查看带视图的 SQL 执行计划

EXPLAIN 语句返回的结果会显示视图引用的表和索引，而不是视图本身的名称。这是因为视图是一张虚拟表，本身并不存储任何数据。视图的定义会和查询语句的其余部分在 SQL 优化过程中进行合并。

参考 bikeshare 数据库示例（英文），以下两个示例查询的执行方式类似：

```
ALTER TABLE trips ADD INDEX (duration);
CREATE OR REPLACE VIEW long_trips AS SELECT * FROM trips WHERE duration > 3600;
EXPLAIN SELECT * FROM long_trips;
EXPLAIN SELECT * FROM trips WHERE duration > 3600;
```

```
Query OK, 0 rows affected (2 min 10.11 sec)
```

```
Query OK, 0 rows affected (0.13 sec)

+--
    ↪ ----------------------------+------------+-----------+-----------------------------------+---------
    ↪
| id                            | estRows    | task      | access object
    ↪                           | operator info                         |
+--
    ↪ ----------------------------+------------+-----------+-----------------------------------+---------
    ↪
| IndexLookUp_12                | 6372547.67 | root      |
    ↪                                        |                                   |
| ├─IndexRangeScan_10(Build)    | 6372547.67 | cop[tikv] | table:trips, index:duration(duration
    ↪ ) | range:(3600,+inf], keep order:false |
| └─TableRowIDScan_11(Probe)    | 6372547.67 | cop[tikv] | table:trips
    ↪                           | keep order:false                      |
+--
    ↪ ----------------------------+------------+-----------+-----------------------------------+---------
    ↪
3 rows in set (0.00 sec)

+--
    ↪ ----------------------------+-----------+-----------+-----------------------------------+----------
    ↪
| id                            | estRows   | task      | access object                         |
    ↪  operator info                        |
+--
    ↪ ----------------------------+-----------+-----------+-----------------------------------+----------
    ↪
| IndexLookUp_10                | 833219.37 | root      |                                   |
    ↪                                       |
| ├─IndexRangeScan_8(Build)     | 833219.37 | cop[tikv] | table:trips, index:duration(duration)
    ↪ | range:(3600,+inf], keep order:false |
| └─TableRowIDScan_9(Probe)     | 833219.37 | cop[tikv] | table:trips
    ↪ | keep order:false                    |
+--
    ↪ ----------------------------+-----------+-----------+-----------------------------------+----------
    ↪
3 rows in set (0.00 sec)
```

同样，该视图中的谓词被下推至基表：

```
EXPLAIN SELECT * FROM long_trips WHERE bike_number = 'W00950';
EXPLAIN SELECT * FROM trips WHERE bike_number = 'W00950';
```

```
+--
    ↪ ------------------------------+---------+-----------+---------------------------------------+------------
    ↪
| id                            | estRows | task      | access object                         |
    ↪ operator info                                 |
+--
    ↪ ------------------------------+---------+-----------+---------------------------------------+------------
    ↪
| IndexLookUp_14                | 3.33    | root      |                                       |
    ↪                                                 |
| ├─IndexRangeScan_11(Build)    | 3333.33 | cop[tikv] | table:trips, index:duration(duration) |
    ↪  range:(3600,+inf], keep order:false, stats:pseudo |
| └─Selection_13(Probe)         | 3.33    | cop[tikv] |                                       |
    ↪  eq(bikeshare.trips.bike_number, "W00950")         |
|   └─TableRowIDScan_12         | 3333.33 | cop[tikv] | table:trips                           |
    ↪  keep order:false, stats:pseudo                    |
+--
    ↪ ------------------------------+---------+-----------+---------------------------------------+------------
    ↪
4 rows in set (0.00 sec)

+--
    ↪ ----------------------+-------------+-----------+--------------+---------------------------------
    ↪
| id                    | estRows     | task      | access object | operator info
    ↪                           |
+--
    ↪ ----------------------+-------------+-----------+--------------+---------------------------------
    ↪
| TableReader_7         | 43.00       | root      |               | data:Selection_6
    ↪                          |
| └─Selection_6         | 43.00       | cop[tikv] |               | eq(bikeshare.trips.
    ↪ bike_number, "W00950") |
|   └─TableFullScan_5   | 19117643.00 | cop[tikv] | table:trips   | keep order:false
    ↪                          |
+--
    ↪ ----------------------+-------------+-----------+--------------+---------------------------------
    ↪
3 rows in set (0.00 sec)
```

执行以上第一条语句时使用了索引，满足视图定义，接着在 TiDB 读取行时应用了 `bike_number = 'W00950'` 条件。执行以上第二条语句时，不存在满足该语句的索引，因此使用了 `TableFullScan`。

TiDB 使用的索引可以同时满足视图定义和语句本身，如以下组合索引所示：

```
ALTER TABLE trips ADD INDEX (bike_number, duration);
```

```
EXPLAIN SELECT * FROM long_trips WHERE bike_number = 'W00950';
EXPLAIN SELECT * FROM trips WHERE bike_number = 'W00950';
```

```
Query OK, 0 rows affected (2 min 31.20 sec)

+--
    ↪ ----------------------------+----------+-----------+-------------------------------------------------
    ↪
| id                            | estRows  | task      | access object
    ↪                                         | operator info
    ↪                                         |
+--
    ↪ ----------------------------+----------+-----------+-------------------------------------------------
    ↪
| IndexLookUp_13                | 63725.48 | root      |
    ↪                                                     |
    ↪                                                     |
| ├─IndexRangeScan_11(Build)    | 63725.48 | cop[tikv] | table:trips, index:bike_number(
    ↪ bike_number, duration) | range:("W00950" 3600,"W00950" +inf], keep order:false |
| └─TableRowIDScan_12(Probe)    | 63725.48 | cop[tikv] | table:trips
    ↪                                           | keep order:false
    ↪                                       |
+--
    ↪ ----------------------------+----------+-----------+-------------------------------------------------
    ↪
3 rows in set (0.00 sec)

+--
    ↪ ----------------------------+----------+-----------+-------------------------------------------------
    ↪
| id                            | estRows  | task      | access object
    ↪                                         | operator info                                 |
+--
    ↪ ----------------------------+----------+-----------+-------------------------------------------------
    ↪
| IndexLookUp_10                | 19117.64 | root      |
    ↪                                                     |
    ↪                                                 |
| ├─IndexRangeScan_8(Build)     | 19117.64 | cop[tikv] | table:trips, index:bike_number(
    ↪ bike_number, duration) | range:["W00950","W00950"], keep order:false |
| └─TableRowIDScan_9(Probe)     | 19117.64 | cop[tikv] | table:trips
    ↪                                           | keep order:false                              |
+--
    ↪ ----------------------------+----------+-----------+-------------------------------------------------
    ↪
```

```
3 rows in set (0.00 sec)
```

在第一条语句中，TiDB 能够使用组合索引的两个部分 (bike_number, duration)。在第二条语句，TiDB 仅使用了索引 (bike_number, duration) 的第一部分 bike_number。

11.3.2.8.1 其他类型查询的执行计划

- MPP 模式查询的执行计划
- 索引查询的执行计划
- Join 查询的执行计划
- 子查询的执行计划
- 聚合查询的执行计划
- 分区查询的执行计划
- 索引合并查询的执行计划

11.3.2.9 用 EXPLAIN 查看分区查询的执行计划

使用 EXPLAIN 语句可以查看 TiDB 在执行查询时需要访问的分区。由于存在分区裁剪，所显示的分区通常只是所有分区的一个子集。本文档介绍了常见分区表的一些优化方式，以及如何解读 EXPLAIN 语句返回的执行计划信息。

本文档所使用的示例数据如下：

```
CREATE TABLE t1 (
 id BIGINT NOT NULL auto_increment,
 d date NOT NULL,
 pad1 BLOB,
 pad2 BLOB,
 pad3 BLOB,
 PRIMARY KEY (id,d)
) PARTITION BY RANGE (YEAR(d)) (
 PARTITION p2016 VALUES LESS THAN (2017),
 PARTITION p2017 VALUES LESS THAN (2018),
 PARTITION p2018 VALUES LESS THAN (2019),
 PARTITION p2019 VALUES LESS THAN (2020),
 PARTITION pmax VALUES LESS THAN MAXVALUE
);

INSERT INTO t1 (d, pad1, pad2, pad3) VALUES
 ('2016-01-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2016-06-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2016-09-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2017-01-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2017-06-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2017-09-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2018-01-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
```

```
 ('2018-06-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2018-09-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2019-01-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2019-06-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2019-09-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2020-01-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2020-06-01', RANDOM_BYTES(102), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2020-09-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024));

INSERT INTO t1 SELECT NULL, a.d, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪ t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, a.d, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪ t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, a.d, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪ t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, a.d, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪ t1 a JOIN t1 b JOIN t1 c LIMIT 10000;

SELECT SLEEP(1);
ANALYZE TABLE t1;
```

以下示例解释了基于新建分区表 `t1` 的一条语句：

```
EXPLAIN SELECT COUNT(*) FROM t1 WHERE d = '2017-06-01';
```

```
+--
    ↪ ----------------------------+---------+-----------+---------------------------+----------------------------
    ↪
| id                            | estRows | task      | access object             | operator info
    ↪                               |
+--
    ↪ ----------------------------+---------+-----------+---------------------------+----------------------------
    ↪
| StreamAgg_21                  | 1.00    | root      |                           | funcs:count(
    ↪ Column#8)->Column#6           |
| └─TableReader_22              | 1.00    | root      |                           | data:
    ↪ StreamAgg_10                    |
|   └─StreamAgg_10              | 1.00    | cop[tikv] |                           | funcs:count
    ↪ (1)->Column#8                 |
|     └─Selection_20            | 8.87    | cop[tikv] |                           | eq(test.t1.d,
    ↪  2017-06-01 00:00:00.000000) |
|       └─TableFullScan_19      | 8870.00 | cop[tikv] | table:t1, partition:p2017 | keep order:
    ↪ false                         |
+--
    ↪ ----------------------------+---------+-----------+---------------------------+----------------------------
    ↪
```

```
5 rows in set (0.01 sec)
```

由上述 EXPLAIN 结果可知，从最末尾的 └─TableFullScan_19 算子开始，再返回到根部的 StreamAgg_21 算子的执行过程如下：

- TiDB 成功地识别出只需要访问一个分区 (p2017)，并将该信息在 access object 列中注明。
- └─TableFullScan_19 算子先对整个分区进行扫描，然后执行 └─Selection_20 算子筛选起始日期为 2017-06-01 00:00:00.000000 的行。
- 之后，└─Selection_20 算子匹配的行在 Coprocessor 中进行流式聚合，Coprocessor 本身就可以理解聚合函数 count。
- 每个 Coprocessor 请求会发送一行数据给 TiDB 的 └─TableReader_22 算子，然后将数据在 StreamAgg_21 算子下进行流式聚合，再将一行数据返回给客户端。

以下示例中，分区裁剪不会消除任何分区：

```
EXPLAIN SELECT COUNT(*) FROM t1 WHERE YEAR(d) = 2017;
```

```
+--
    ↪ ---------------------------------+----------+-----------+--------------------------+-----------------
    ↪
| id                                | estRows  | task      | access object            |
    ↪ operator info                     |
+--
    ↪ ---------------------------------+----------+-----------+--------------------------+-----------------
    ↪
| HashAgg_20                          | 1.00     | root      |                          | funcs:
    ↪ count(Column#7)->Column#6  |
| └─PartitionUnion_21                 | 5.00     | root      |                          |
    ↪                                   |
|   ├─StreamAgg_36                    | 1.00     | root      |                          | funcs:
    ↪ count(Column#9)->Column#7  |
|   │ └─TableReader_37                | 1.00     | root      |                          | data:
    ↪ StreamAgg_25                |
|   │   └─StreamAgg_25                | 1.00     | cop[tikv] |                          | funcs:
    ↪ count(1)->Column#9         |
|   │     └─Selection_35              | 6000.00  | cop[tikv] |                          | eq(
    ↪ year(test.t1.d), 2017)        |
|   │       └─TableFullScan_34        | 7500.00  | cop[tikv] | table:t1, partition:p2016 | keep
    ↪ order:false                 |
|   ├─StreamAgg_55                    | 1.00     | root      |                          | funcs:
    ↪ count(Column#11)->Column#7 |
|   │ └─TableReader_56                | 1.00     | root      |                          | data:
    ↪ StreamAgg_44                |
|   │   └─StreamAgg_44                | 1.00     | cop[tikv] |                          | funcs:
    ↪ count(1)->Column#11        |
```

```
|   |   └─Selection_54             | 14192.00 | cop[tikv] |                           | eq(
    ↪ year(test.t1.d), 2017)       |
|   |     └─TableFullScan_53       | 17740.00 | cop[tikv] | table:t1, partition:p2017 | keep
    ↪ order:false                  |
|   ├─StreamAgg_74                 | 1.00     | root      |                           | funcs:
    ↪ count(Column#13)->Column#7 |
|   | └─TableReader_75             | 1.00     | root      |                           | data:
    ↪ StreamAgg_63                 |
|   |   └─StreamAgg_63             | 1.00     | cop[tikv] |                           | funcs:
    ↪ count(1)->Column#13          |
|   |     └─Selection_73           | 3977.60  | cop[tikv] |                           | eq(
    ↪ year(test.t1.d), 2017)       |
|   |       └─TableFullScan_72     | 4972.00  | cop[tikv] | table:t1, partition:p2018 | keep
    ↪ order:false                  |
|   ├─StreamAgg_93                 | 1.00     | root      |                           | funcs:
    ↪ count(Column#15)->Column#7 |
|   | └─TableReader_94             | 1.00     | root      |                           | data:
    ↪ StreamAgg_82                 |
|   |   └─StreamAgg_82             | 1.00     | cop[tikv] |                           | funcs:
    ↪ count(1)->Column#15          |
|   |     └─Selection_92           | 20361.60 | cop[tikv] |                           | eq(
    ↪ year(test.t1.d), 2017)       |
|   |       └─TableFullScan_91     | 25452.00 | cop[tikv] | table:t1, partition:p2019 | keep
    ↪ order:false                  |
|   └─StreamAgg_112                | 1.00     | root      |                           | funcs:
    ↪ count(Column#17)->Column#7 |
|     └─TableReader_113            | 1.00     | root      |                           | data:
    ↪ StreamAgg_101                |
|       └─StreamAgg_101            | 1.00     | cop[tikv] |                           | funcs:
    ↪ count(1)->Column#17          |
|         └─Selection_111          | 8892.80  | cop[tikv] |                           | eq(
    ↪ year(test.t1.d), 2017)       |
|           └─TableFullScan_110    | 11116.00 | cop[tikv] | table:t1, partition:pmax  | keep
    ↪ order:false                  |
+--
    ↪ ---------------------------------+----------+-----------+---------------------------+-----------------
    ↪
27 rows in set (0.00 sec)
```

由上述 EXPLAIN 结果可知：

- TiDB 认为需要访问所有分区 (p2016..pMax)。这是因为 TiDB 将谓词 YEAR (d) = 2017 视为 non-sargable。这个问题并非是 TiDB 特有的。
- 在扫描每个分区时，Selection 算子将筛选出年份不为 2017 的行。
- 在每个分区上会执行流式聚合，以计算匹配的行数。

- └─PartitionUnion_21 算子会合并访问每个分区后的结果。

11.3.2.9.1 其他类型查询的执行计划

- MPP 模式查询的执行计划
- 索引查询的执行计划
- Join 查询的执行计划
- 子查询的执行计划
- 聚合查询的执行计划
- 视图查询的执行计划
- 索引合并查询的执行计划

11.3.2.10 用 EXPLAIN 查看索引合并的 SQL 执行计划

索引合并是从 TiDB v4.0 起引入的一种新的表访问方式。在这种访问方式下，TiDB 优化器可以选择对一张表使用多个索引，并将每个索引的返回结果进行合并。在某些场景下，这种访问方式能够减少大量不必要的数据扫描，提升查询的执行效率。

TiDB 中的索引合并分为交集型和并集型两种类型，分别适用于由 AND 连接的表达式和由 OR 连接的表达式。其中，并集型索引合并在 TiDB v4.0 作为实验功能引入，在 v5.4.0 成为正式功能 (GA)。交集型索引合并从 TiDB v6.5.0 起引入，且必须使用USE_INDEX_MERGE Hint 指定才能使用。

11.3.2.10.1 开启索引合并

在v5.4.0及以上版本的新建集群中，索引合并默认开启。在其他情况下如果未开启，可将 tidb_enable_index_merge ↪ 的值设为 ON 来开启索引合并功能。

```
SET session tidb_enable_index_merge = ON;
```

11.3.2.10.2 示例

```
CREATE TABLE t(a int, b int, c int, d int, INDEX idx_a(a), INDEX idx_b(b), INDEX idx_c(c), INDEX
    ↪ idx_d(d));
```

```
EXPLAIN SELECT /*+ NO_INDEX_MERGE() */ * FROM t WHERE a = 1 OR b = 1;
+--
    ↪ ----------------------+----------+-----------+---------------+----------------------------------+
    ↪
| id                      | estRows  | task      | access object | operator info
    ↪                         |
+--
    ↪ ----------------------+----------+-----------+---------------+----------------------------------+
    ↪
| TableReader_7           | 19.99    | root      |               | data:Selection_6
    ↪                          |
```

```
| └─Selection_6           | 19.99    | cop[tikv] |               | or(eq(test.t.a, 1), eq(test.t
    ↪ .b, 1)) |
|   └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo       |
+--
    ↪ ----------------------+----------+-----------+---------------+--------------------------------+
    ↪
EXPLAIN SELECT /*+ USE_INDEX_MERGE(t) */ * FROM t WHERE a > 1 OR b > 1;
+--
    ↪ ----------------------------+---------+-----------+-------------------------+-----------------------------
    ↪
| id                          | estRows | task      | access object           | operator info
    ↪                              |
+--
    ↪ ----------------------------+---------+-----------+-------------------------+-----------------------------
    ↪
| IndexMerge_8                | 5555.56 | root      |                         | type: union
    ↪                                 |
| ├─IndexRangeScan_5(Build)     | 3333.33 | cop[tikv] | table:t, index:idx_a(a) | range:(1,+inf
    ↪ ], keep order:false, stats:pseudo |
| ├─IndexRangeScan_6(Build)     | 3333.33 | cop[tikv] | table:t, index:idx_b(b) | range:(1,+inf
    ↪ ], keep order:false, stats:pseudo |
| └─TableRowIDScan_7(Probe)     | 5555.56 | cop[tikv] | table:t                 | keep order:
    ↪ false, stats:pseudo                |
+--
    ↪ ----------------------------+---------+-----------+-------------------------+-----------------------------
    ↪
```

例如，在上述示例中，过滤条件是使用 `OR` 连接的 `WHERE` 子句。在启用索引合并前，每个表只能使用一个索引，不能将 `a = 1` 下推到索引 `a`，也不能将 `b = 1` 下推到索引 `b`。当 `t` 中存在大量数据时，全表扫描的效率会很低。

对于以上查询语句，优化器选择了并集型索引合并的方式访问表。在这种访问方式下，优化器可以选择对一张表使用多个索引，并将每个索引的返回结果进行合并，生成以上两个示例中的后一个执行计划。此时的 `IndexMerge_8` 算子的 `operator info` 中的 `type: union` 表示该算子是一个并集型索引合并算子。它有三个子节点，其中 `IndexRangeScan_5` 和 `IndexRangeScan_6` 根据范围扫描得到符合条件的所有 `RowID`，再由 `TableRowIDScan_7` 算子根据这些 `RowID` 精确地读取所有满足条件的数据。

其中对于 `IndexRangeScan/TableRangeScan` 一类按范围进行的扫表操作，`EXPLAIN` 表中 `operator info` 列相比于其他扫表操作，多了被扫描数据的范围这一信息。比如上面的例子中，`IndexRangeScan_13` 算子中的 `range:(1,+inf]` 这一信息表示该算子扫描了从 `1` 到正无穷这个范围的数据。

```
EXPLAIN SELECT /*+ NO_INDEX_MERGE() */ * FROM t WHERE a > 1 AND b > 1 AND c = 1;  --
    ↪ 不使用索引合并


+--
    ↪ ----------------------------+---------+-----------+-------------------------+-----------------------------
    ↪
```

```
| id                            | estRows | task      | access object           | operator info
    ↪                               |
+--
    ↪ ------------------------------+---------+-----------+-------------------------+-----------------------
    ↪
| IndexLookUp_19                | 1.11    | root      |                         |
    ↪                                           |
| ├─IndexRangeScan_16(Build)    | 10.00   | cop[tikv] | table:t, index:idx_c(c) | range:[1,1],
    ↪ keep order:false, stats:pseudo |
| └─Selection_18(Probe)         | 1.11    | cop[tikv] |                         | gt(test.t.a,
    ↪ 1), gt(test.t.b, 1)           |
|   └─TableRowIDScan_17         | 10.00   | cop[tikv] | table:t                 | keep order:
    ↪ false, stats:pseudo             |
+--
    ↪ ------------------------------+---------+-----------+-------------------------+-----------------------
    ↪

EXPLAIN SELECT /*+ USE_INDEX_MERGE(t, idx_a, idx_b, idx_c) */ * FROM t WHERE a > 1 AND b > 1 AND
    ↪ c = 1;  -- 使用索引合并

+--
    ↪ ------------------------------+---------+-----------+-------------------------+-----------------------
    ↪
| id                            | estRows | task      | access object           | operator info
    ↪                                   |
+--
    ↪ ------------------------------+---------+-----------+-------------------------+-----------------------
    ↪
| IndexMerge_9                  | 1.11    | root      |                         | type:
    ↪ intersection                            |
| ├─IndexRangeScan_5(Build)     | 3333.33 | cop[tikv] | table:t, index:idx_a(a) | range:(1,+inf
    ↪ ], keep order:false, stats:pseudo |
| ├─IndexRangeScan_6(Build)     | 3333.33 | cop[tikv] | table:t, index:idx_b(b) | range:(1,+inf
    ↪ ], keep order:false, stats:pseudo |
| ├─IndexRangeScan_7(Build)     | 10.00   | cop[tikv] | table:t, index:idx_c(c) | range:[1,1],
    ↪ keep order:false, stats:pseudo    |
| └─TableRowIDScan_8(Probe)     | 1.11    | cop[tikv] | table:t                 | keep order:
    ↪ false, stats:pseudo                |
+--
    ↪ ------------------------------+---------+-----------+-------------------------+-----------------------
    ↪
```

在如上示例中，过滤条件是使用 AND 连接的 WHERE 子句。在启用索引合并前，只能选择使用 idx_a、idx_b 或 idx_c 三个索引中的一个。

如果三个过滤条件中的其中一个的过滤性非常好，直接选择对应的索引即可达到理想的执行效率。但如果数据分布同时满足以下三种情形，可以考虑使用交集型索引合并：

- 全表的数据量相当大，导致直接读全表的执行效率非常低下
- 每个过滤条件单独的过滤性都不够好，导致 IndexLookUp 使用单个索引的执行效率也不够理想
- 三个过滤条件整体的过滤性非常好

在交集型索引合并访问方式下，优化器可以选择对一张表使用多个索引，并将每个索引的返回结果取交集，生成以上两个示例中的后一个执行计划。此时的 IndexMerge_9 算子的 operator info 中的 type: intersection 表示该算子是一个交集型索引合并算子。该执行计划的其它部分和上述并集型索引合并示例类似。

> 注意：
>
> - TiDB 的索引合并特性在 v5.4.0 及之后的版本默认开启，即tidb_enable_index_merge 为 ON。
> - 如果查询中使用了 SQL 优化器 Hint USE_INDEX_MERGE，无论 tidb_enable_index_merge 开关是否开启，都会强制使用索引合并特性。当过滤条件中有无法下推的表达式时，必须使用 Hint USE_INDEX_MERGE 才能开启索引合并。
> - 如果查询有除了全表扫描以外的单索引扫描方式可以选择，优化器不会自动选择索引合并，只能通过 Hint 指定使用索引合并。
> - 索引合并目前无法在临时表上使用。
> - 交集型索引合并目前不会被优化器自动选择，必须使用USE_INDEX_MERGE Hint 指定表名和索引名时才会被选择。

11.3.2.10.3 其他类型查询的执行计划

- MPP 模式查询的执行计划
- 索引查询的执行计划
- Join 查询的执行计划
- 子查询的执行计划
- 聚合查询的执行计划
- 视图查询的执行计划
- 分区查询的执行计划

11.3.3 SQL 优化流程

11.3.3.1 SQL 优化流程简介

在 TiDB 中，从输入的查询文本到最终的执行计划执行结果的过程可以见下图。

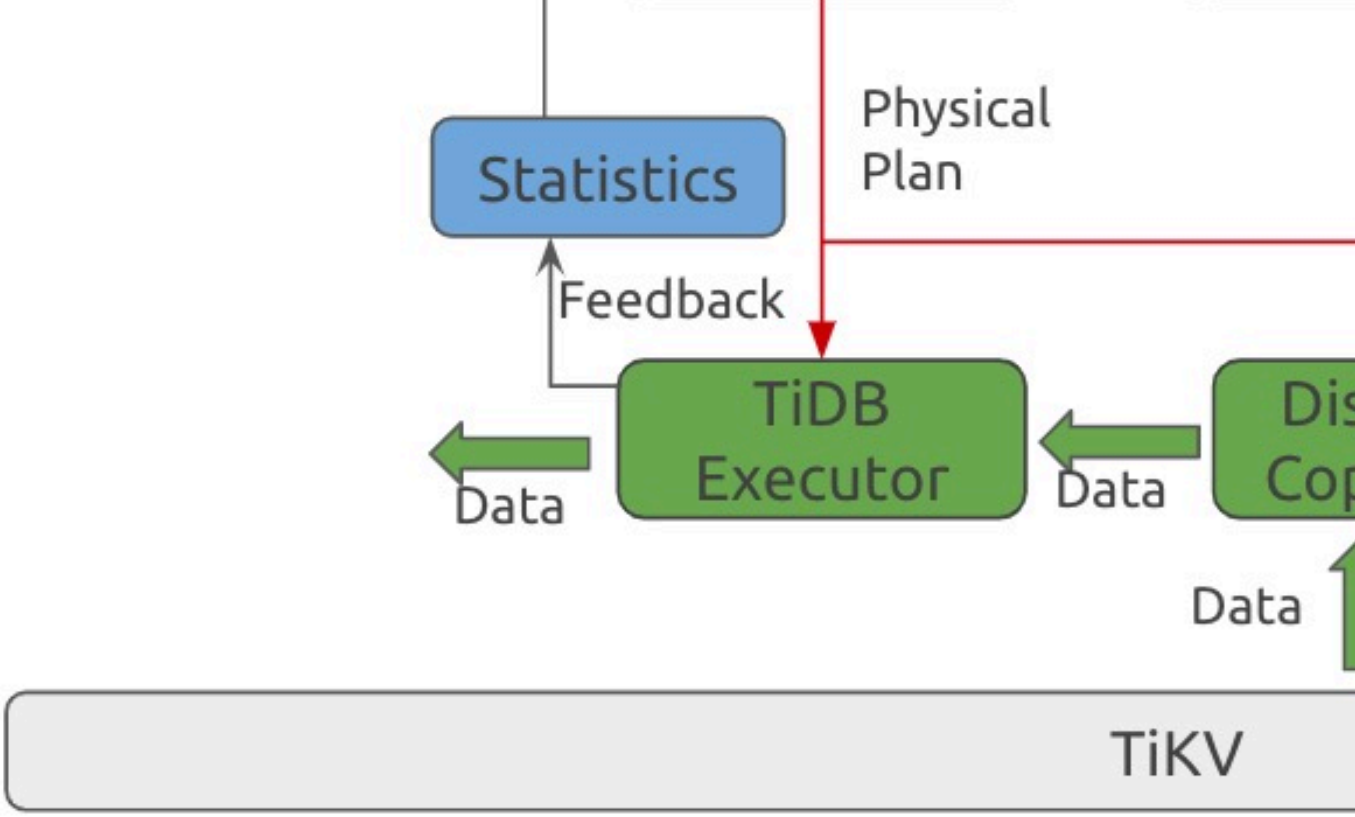


图 162: SQL Optimization

在经过了 parser 对原始查询文本的解析以及一些简单的合法性验证后，TiDB 首先会对查询做一些逻辑上的等价变化，详细的变化可以查询逻辑优化章节。

通过这些等价变化，使得这个查询在逻辑执行计划上可以变得更易于处理。在等价变化结束之后，TiDB 会得到一个与原始查询等价的查询计划结构，之后根据数据分布、以及一个算子具体的执行开销，来获得一个最终的执行计划，这部分内容可以查询物理优化章节。

同时，TiDB 在执行PREPARE 语句时，可以选择开启缓存来降低 TiDB 生成执行计划的开销，这部分内容会在执行计划缓存一节中介绍。

11.3.3.2 逻辑优化

11.3.3.2.1 逻辑优化

本章节将对一些比较关键的逻辑改写进行说明，帮助大家理解 TiDB 如何生成最终的查询计划。比如在 TiDB 输入 `select * from t where t.a in (select t1.a from t1 where t1.b=t.b)` 这个查询时，在最终的执行计划中将看不到这个 `t.a in (select t1.a from t1 where t1.b=t.b)` 这个 IN 子查询的存在，这便是因为 TiDB 对这里进行了一些改写。

本章节会介绍如下几个关键改写：

- 子查询相关的优化
- 列裁剪
- 关联子查询去关联
- Max/Min 消除

- 谓词下推
- 分区裁剪
- TopN 和 Limit 下推
- Join Reorder

11.3.3.2.2 子查询相关的优化

本文主要介绍子查询相关的优化。

通常会遇到如下情况的子查询：

- NOT IN (SELECT ... FROM ...)
- NOT EXISTS (SELECT ... FROM ...)
- IN (SELECT ... FROM ..)
- EXISTS (SELECT ... FROM ...)
- ... >/>=/</<=/=/!= (SELECT ... FROM ...)

有时，子查询中包含了非子查询中的列，如 `select * from t where t.a in (select * from t2 where t.b =t2.b)` 中，子查询中的 `t.b` 不是子查询中的列，而是从子查询外面引入的列。这种子查询通常会被称为关联子查询，外部引入的列会被称为关联列，关联子查询相关的优化参见关联子查询去关联。本文主要关注不涉及关联列的子查询。

子查询默认会以理解 TiDB 执行计划中提到的 `semi join` 作为默认的执行方式，同时对于一些特殊的子查询，TiDB 会做一些逻辑上的替换使得查询可以获得更好的执行性能。

`... < ALL (SELECT ... FROM ...)` 或者 `... > ANY (SELECT ... FROM ...)`

对于这种情况，可以将 `ALL` 或者 `ANY` 用 `MAX` 以及 `MIN` 来代替。不过由于在表为空时，`MAX(EXPR)` 以及 `MIN(EXPR)` 的结果会为 `NULL`，其表现形式和 `EXPR` 是有 `NULL` 值的结果一样。以及外部表达式结果为 `NULL` 时也会影响表达式的最终结果，因此这里完整的改写会是如下的形式：

- `t.id < all(select s.id from s)` 会被改写为 `t.id < min(s.id)and if(sum(s.id is null)!= 0, null, true)`。
- `t.id < any (select s.id from s)` 会被改写为 `t.id < max(s.id)or if(sum(s.id is null)!= 0, null, false)`。

`... != ANY (SELECT ... FROM ...)`

对于这种情况，当子查询中不同值的个数只有一种的话，那只要和这个值对比就即可。如果子查询中不同值的个数多于 1 个，那么必然会有不相等的情况出现。因此这样的子查询可以采取如下的改写手段：

- `select * from t where t.id != any (select s.id from s)` 会被改写为 `select t.* from t, (select s.id, count(distinct s.id)as cnt_distinct from s)where (t.id != s.id or cnt_distinct > 1)`

`... = ALL (SELECT ... FROM ...)`

对于这种情况，当子查询中不同值的个数多于一种的话，那么这个表达式的结果必然为假。因此这样的子查询在 TiDB 中会改写为如下的形式：

- select * from t where t.id = all (select s.id from s) 会被改写为 select t.* from t, (select s
  ↪ .id, count(distinct s.id)as cnt_distinct from s)where (t.id = s.id and cnt_distinct <= 1)

... IN (SELECT ... FROM ...)

对于这种情况，会将其 IN 的子查询改写为 SELECT ... FROM ... GROUP ... 的形式，然后将 IN 改写为普通的 JOIN 的形式。如 select * from t1 where t1.a in (select t2.a from t2) 会被改写为 select t1.* from t1
↪ , (select distinct(a)a from t2)t2 where t1.a = t2.a 的形式。同时这里的 DISTINCT 可以在 t2.a 具有 UNIQUE 属性时被自动消去。

```
explain select * from t1 where t1.a in (select t2.a from t2);
```

```
+--
    ↪ ---------------------------+---------+-----------+-----------------------+-----------------------------
    ↪
| id                           | estRows | task      | access object         | operator info
    ↪                                                                  |
+--
    ↪ ---------------------------+---------+-----------+-----------------------+-----------------------------
    ↪
| IndexJoin_12                 | 9990.00 | root      |                       | inner join, inner
    ↪ :TableReader_11, outer key:test.t2.a, inner key:test.t1.a |
| ├─HashAgg_21(Build)          | 7992.00 | root      |                       | group by:test.t2
    ↪ .a, funcs:firstrow(test.t2.a)->test.t2.a                  |
| │ └─IndexReader_28           | 9990.00 | root      |                       | index:
    ↪ IndexFullScan_27                                                 |
| │   └─IndexFullScan_27       | 9990.00 | cop[tikv] | table:t2, index:idx(a) | keep order:false
    ↪ , stats:pseudo                                            |
| └─TableReader_11(Probe)      | 7992.00 | root      |                       | data:
    ↪ TableRangeScan_10                                                |
|   └─TableRangeScan_10        | 7992.00 | cop[tikv] | table:t1              | range: decided
    ↪ by [test.t2.a], keep order:false, stats:pseudo             |
+--
    ↪ ---------------------------+---------+-----------+-----------------------+-----------------------------
    ↪
```

这个改写会在 IN 子查询相对较小，而外部查询相对较大时产生更好的执行性能。因为不经过改写的情况下，我们无法使用以 t2 为驱动表的 index join。同时这里的弊端便是，当改写删成的聚合无法被自动消去且 t2 表比较大时，反而会影响查询的性能。目前 TiDB 中使用tidb_opt_insubq_to_join_and_agg 变量来控制这个优化的打开与否。当遇到不合适这个优化的情况可以手动关闭。

EXISTS 子查询以及 ... >/>=/</<=/=/!= (SELECT ... FROM ...)

当前对于这种场景的子查询，当它不是关联子查询时，TiDB 会在优化阶段提前展开它，将其直接替换为一个结果集直接判断结果。如下图中，EXISTS 会提前在优化阶段被执行为 TRUE，从而不会在最终的执行结果中看到它。

```
create table t1(a int);
```

```
create table t2(a int);
insert into t2 values(1);
explain select * from t1 where exists (select * from t2);
```

```
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
| id                    | estRows  | task      | access object | operator info
    ↪ |
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
| TableReader_12          | 10000.00 | root      |               | data:TableFullScan_11
    ↪ |
| └─TableFullScan_11      | 10000.00 | cop[tikv] | table:t1      | keep order:false, stats:pseudo
    ↪  |
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
```

在上述优化中，优化器会自动优化语句执行。除以上情况外，你也可以在语句中添加SEMI_JOIN_REWRITE hint进一步改写语句。

如果不使用 SEMI_JOIN_REWRITE 进行改写，Semi Join 在选择 Hash Join 的执行方式时，只能够使用子查询构建哈希表，因此在子查询比外查询结果集大时，执行速度可能会不及预期。Semi Join 在选择 Index Join 的执行方式时，只能够使用外查询作为驱动表，因此在子查询比外查询结果集小时，执行速度可能会不及预期。

使用 SEMI_JOIN_REWRITE 改写后，优化器便可以扩大选择范围，选择更好的执行方式。

11.3.3.2.3 列裁剪

列裁剪的基本思想在于：对于算子中实际用不上的列，优化器在优化的过程中没有必要保留它们。对这些列的删除会减少 I/O 资源占用，并为后续的优化带来便利。下面给出一个列重复的例子：

假设表 t 里面有 a b c d 四列，执行如下语句：

```
select a from t where b > 5
```

在该查询的过程中，t 表实际上只有 a, b 两列会被用到，而 c, d 的数据则显得多余。对应到该语句的查询计划，Selection 算子会用到 b 列，下面接着的 DataSource 算子会用到 a, b 两列，而剩下 c, d 两列则都可以裁剪掉，DataSource 算子在读数据时不需要将它们读进来。

出于上述考量，TiDB 会在逻辑优化阶段进行自上而下的扫描，裁剪不需要的列，减少资源浪费。该扫描过程称作“列裁剪”，对应逻辑优化规则中的 columnPruner。如果要关闭这个规则，可以在参照优化规则及表达式下推的黑名单中的关闭方法。

11.3.3.2.4 关联子查询去关联

子查询相关的优化中介绍了当没有关联列时，TiDB 是如何处理子查询的。由于为关联子查询解除关联依赖比较复杂，本文档中会介绍一些简单的场景以及这个优化规则的适用范围。

简介

以 `select * from t1 where t1.a < (select sum(t2.a)from t2 where t2.b = t1.b)` 为例，这里子查询 `t1.a < (select sum(t2.a)from t2 where t2.b = t1.b)` 中涉及了关联列上的条件 `t2.b=t1.b`，不过恰好由于这是一个等值条件，因此可以将其等价的改写为 `select t1.* from t1, (select b, sum(a)sum_a from t2 group by b)t2 where t1.b = t2.b and t1.a < t2.sum_a;`。这样，一个关联子查询就被重新改写为 JOIN 的形式。

TiDB 之所以要进行这样的改写，是因为关联子查询每次子查询执行时都是要和它的外部查询结果绑定的。在上面的例子中，如果 `t1.a` 有一千万个值，那这个子查询就要被重复执行一千万次，因为 `t2.b=t1.b` 这个条件会随着 `t1.a` 值的不同而发生变化。当通过一些手段将关联依赖解除后，这个子查询就只需要被执行一次了。

限制

这种改写的弊端在于，在关联没有被解除时，优化器是可以使用关联列上的索引的。也就是说，虽然这个子查询可能被重复执行多次，但是每次都可以使用索引过滤数据。而解除关联的变换上，通常是会导致关联列的位置发生改变而导致虽然子查询只被执行了一次，但是单次执行的时间会比没有解除关联时的单次执行时间长。

因此，在外部的值比较少的情况下，不解除关联依赖反而可能对执行性能更有帮助。这时可以通过使用 Optimizer Hint NO_DECORRELATE 或优化规则及表达式下推的黑名单中关闭 “子查询去关联” 优化规则的方式来关闭这个优化。在一般情况下，推荐使用 Optimizer Hint 并在需要时配合执行计划管理功能来禁止解除关联。

样例

```
create table t1(a int, b int);
create table t2(a int, b int, index idx(b));
explain select * from t1 where t1.a < (select sum(t2.a) from t2 where t2.b = t1.b);
```

```
+--
    ↪ --------------------------------+----------+-----------+---------------+--------------------------------
    ↪
| id                                | estRows  | task      | access object | operator info
    ↪                                                                                 |
+--
    ↪ --------------------------------+----------+-----------+---------------+--------------------------------
    ↪
| HashJoin_11                       | 9990.00  | root      |               | inner join, equal:[eq
    ↪ (test.t1.b, test.t2.b)], other cond:lt(cast(test.t1.a), Column#7)  |
| ├─HashAgg_23(Build)               | 7992.00  | root      |               | group by:test.t2.b,
    ↪ funcs:sum(Column#8)->Column#7, funcs:firstrow(test.t2.b)->test.t2.b |
| │ └─TableReader_24                | 7992.00  | root      |               | data:HashAgg_16
    ↪                                                                        |
| │   └─HashAgg_16                  | 7992.00  | cop[tikv] |               | group by:test.t2.b,
    ↪ funcs:sum(test.t2.a)->Column#8                                         |
| │     └─Selection_22              | 9990.00  | cop[tikv] |               | not(isnull(test.t2.b
    ↪ ))                                                                     |
```

```
| |       └─TableFullScan_21       | 10000.00 | cop[tikv] | table:t2      | keep order:false,
    ↪ stats:pseudo                                                          |
| └─TableReader_15(Probe)          | 9990.00  | root      |               | data:Selection_14
    ↪                                                                       |
|   └─Selection_14                 | 9990.00  | cop[tikv] |               | not(isnull(test.t1.b
    ↪ ))                                                                    |
|     └─TableFullScan_13           | 10000.00 | cop[tikv] | table:t1      | keep order:false,
    ↪ stats:pseudo                                                          |
+--
    ↪ --------------------------------+----------+-----------+---------------+--------------------------------
    ↪
```

上面是优化生效的情况，可以看到 HashJoin_11 是一个普通的 inner join。

接下来，通过 Optimizer Hint NO_DECORRELATE 提示优化器不对该子查询解除关联：

```
explain select * from t1 where t1.a < (select /*+ NO_DECORRELATE() */ sum(t2.a) from t2 where t2.
    ↪ b = t1.b);
```

```
+--
    ↪ ---------------------------------------+-----------+-----------+-----------------------+--------------
    ↪
| id                                    | estRows   | task      | access object         |
    ↪ operator info                                                                     |
+--
    ↪ ---------------------------------------+-----------+-----------+-----------------------+--------------
    ↪
| Projection_10                         | 10000.00  | root      |                       |
    ↪ test.t1.a, test.t1.b                                                              |
| └─Apply_12                             | 10000.00  | root      |                       |
    ↪ CARTESIAN inner join, other cond:lt(cast(test.t1.a, decimal(10,0) BINARY), Column#7) |
|   ├─TableReader_14(Build)              | 10000.00  | root      |                       |
    ↪ data:TableFullScan_13                                                             |
|   │ └─TableFullScan_13                 | 10000.00  | cop[tikv] | table:t1              |
    ↪ keep order:false, stats:pseudo                                                    |
|   └─MaxOneRow_15(Probe)                | 10000.00  | root      |                       |
    ↪                                                                                   |
|     └─StreamAgg_20                     | 10000.00  | root      |                       |
    ↪ funcs:sum(Column#14)->Column#7                                                    |
|       └─Projection_45                  | 100000.00 | root      |                       |
    ↪ cast(test.t2.a, decimal(10,0) BINARY)->Column#14                                  |
|         └─IndexLookUp_44               | 100000.00 | root      |                       |
    ↪                                                                                   |
|           ├─IndexRangeScan_42(Build)   | 100000.00 | cop[tikv] | table:t2, index:idx(b) |
    ↪ range: decided by [eq(test.t2.b, test.t1.b)], keep order:false, stats:pseudo      |
|           └─TableRowIDScan_43(Probe)   | 100000.00 | cop[tikv] | table:t2              |
    ↪ keep order:false, stats:pseudo                                                    |
```

```
+--
    ↪ ----------------------------------------+-----------+-----------+-------------------------+---------------
    ↪
```

也可以通过全局关闭关联规则达到同样的效果：

```
insert into mysql.opt_rule_blacklist values("decorrelate");
admin reload opt_rule_blacklist;
explain select * from t1 where t1.a < (select sum(t2.a) from t2 where t2.b = t1.b);
```

```
+----------------------------------------+-----------+-----------+-------------------------+-----------------
    ↪
| id                                     | estRows   | task      | access object           |
    ↪ operator info                                                                       |
+----------------------------------------+-----------+-----------+-------------------------+-----------------
    ↪
| Projection_10                          | 10000.00  | root      |                         |
    ↪ test.t1.a, test.t1.b                                                                |
| └─Apply_12                             | 10000.00  | root      |                         |
    ↪ CARTESIAN inner join, other cond:lt(cast(test.t1.a, decimal(10,0) BINARY), Column#7) |
|   ├─TableReader_14(Build)              | 10000.00  | root      |                         |
    ↪ data:TableFullScan_13                                                               |
|   │ └─TableFullScan_13                 | 10000.00  | cop[tikv] | table:t1                |
    ↪ keep order:false, stats:pseudo                                                      |
|   └─MaxOneRow_15(Probe)                | 10000.00  | root      |                         |
    ↪                                                                                     |
|     └─StreamAgg_20                     | 10000.00  | root      |                         |
    ↪ funcs:sum(Column#14)->Column#7                                                      |
|       └─Projection_45                  | 100000.00 | root      |                         |
    ↪ cast(test.t2.a, decimal(10,0) BINARY)->Column#14                                    |
|         └─IndexLookUp_44               | 100000.00 | root      |                         |
    ↪                                                                                     |
|           ├─IndexRangeScan_42(Build)   | 100000.00 | cop[tikv] | table:t2, index:idx(b)  |
    ↪ range: decided by [eq(test.t2.b, test.t1.b)], keep order:false, stats:pseudo       |
|           └─TableRowIDScan_43(Probe)   | 100000.00 | cop[tikv] | table:t2                |
    ↪ keep order:false, stats:pseudo                                                      |
+----------------------------------------+-----------+-----------+-------------------------+-----------------
    ↪
```

在执行了关闭关联规则的语句后，可以在 `IndexRangeScan_25(Build)` 的 `operator info` 中看到 `range: decided by [eq(test.t2.b, test.t1.b)]`。这部分信息就是关联依赖未被解除时，TiDB 使用关联条件进行索引范围查询的显示结果。

11.3.3.2.5 Max/Min 函数消除规则

在 SQL 中包含了 max/min 函数时，查询优化器会尝试使用 max/min 消除优化规则来将 max/min 聚合函数转换为 TopN 算子，从而能够有效地利用索引进行查询。

根据 select 语句中 max/min 函数的个数，这一优化规则有以下两种表现形式：

- 只有一个 max/min 函数时的优化规则
- 存在多个 max/min 函数时的优化规则

只有一个 max/min 函数时的优化规则

当一个 SQL 满足以下条件时，就会应用这个规则：

- 只有一个聚合函数，且为 max 或者 min 函数。
- 聚合函数没有相应的 group by 语句。

例如：

```
select max(a) from t
```

这时 max/min 消除优化规则会将其重写为：

```
select max(a) from (select a from t where a is not null order by a desc limit 1) t
```

这个新的 SQL 语句在 a 列存在索引（或 a 列是某个联合索引的前缀）时，能够利用索引只扫描一行数据来得到最大或者最小值，从而避免对整个表的扫描。

上述例子最终得到的执行计划如下：

```
mysql> explain select max(a) from t;
+------------------------------+---------+-----------+-------------------------+-------------------------------
    ↪
| id                           | estRows | task      | access object           | operator info
    ↪                        |
+------------------------------+---------+-----------+-------------------------+-------------------------------
    ↪
| StreamAgg_13                 | 1.00    | root      |                         | funcs:max(test.t
    ↪ .a)->Column#4       |
| └─Limit_17                   | 1.00    | root      |                         | offset:0, count
    ↪ :1                  |
|   └─IndexReader_27           | 1.00    | root      |                         | index:Limit_26
    ↪                       |
|     └─Limit_26               | 1.00    | cop[tikv] |                         | offset:0, count
    ↪ :1                  |
|       └─IndexFullScan_25     | 1.00    | cop[tikv] | table:t, index:idx_a(a) | keep order:true
    ↪ , desc, stats:pseudo |
+------------------------------+---------+-----------+-------------------------+-------------------------------
    ↪
5 rows in set (0.00 sec)
```

存在多个 max/min 函数时的优化规则

当一个 SQL 满足以下条件时，就会应用这个规则：

- 有多个聚合函数，且所有的聚合函数都是 max/min
- 聚合函数没有相应的 `group by` 语句。
- 每个 `max/min` 聚合函数参数中的列都有索引能够保序。

下面是一个简单的例子：

```
select max(a) - min(a) from t
```

优化规则会先检查 a 列是否存在索引能够为其保序，如果存在，这个 SQL 会先被重写为两个子查询的笛卡尔积：

```
select max_a - min_a
from
    (select max(a) as max_a from t) t1,
    (select min(a) as min_a from t) t2
```

这样，两个子句中的 `max/min` 函数就可以使用上述“只有一个 `max/min` 函数时的优化规则”分别进行优化，最终重写为：

```
select max_a - min_a
from
    (select max(a) as max_a from (select a from t where a is not null order by a desc limit 1) t)
        ↪  t1,
    (select min(a) as min_a from (select a from t where a is not null order by a asc limit 1) t)
        ↪ t2
```

同样的，如果 a 列能够使用索引保序，那这个优化只会扫描两行数据，避免了对整个表的扫描。但如果 a 列没有可以保序的索引，这个变换会使原本只需一次的全表扫描变成两次，因此这个规则就不会被应用。

最后得到的执行计划：

```
mysql> explain select max(a)-min(a) from t;
+-----------------------------------+---------+-----------+-------------------------+-----------------------------
    ↪
| id                                | estRows | task      | access object           | operator
    ↪ info                        |
+-----------------------------------+---------+-----------+-------------------------+-----------------------------
    ↪
| Projection_17                     | 1.00    | root      |                         | minus(
    ↪ Column#4, Column#5)->Column#6 |
| └─HashJoin_18                     | 1.00    | root      |                         | CARTESIAN
    ↪  inner join                 |
|   ├─StreamAgg_45(Build)           | 1.00    | root      |                         | funcs:min
    ↪ (test.t.a)->Column#5        |
```

```
|   |  └─Limit_49                    | 1.00    | root      |                          | offset:0,
    ↪  count:1                   |
|   |    └─IndexReader_59            | 1.00    | root      |                          | index:
    ↪ Limit_58                     |
|   |      └─Limit_58                | 1.00    | cop[tikv] |                          | offset:0,
    ↪  count:1                   |
|   |        └─IndexFullScan_57      | 1.00    | cop[tikv] | table:t, index:idx_a(a) | keep
    ↪ order:true, stats:pseudo      |
|   └─StreamAgg_24(Probe)            | 1.00    | root      |                          | funcs:max
    ↪ (test.t.a)->Column#4      |
|     └─Limit_28                     | 1.00    | root      |                          | offset:0,
    ↪  count:1                   |
|       └─IndexReader_38             | 1.00    | root      |                          | index:
    ↪ Limit_37                     |
|         └─Limit_37                 | 1.00    | cop[tikv] |                          | offset:0,
    ↪  count:1                   |
|           └─IndexFullScan_36       | 1.00    | cop[tikv] | table:t, index:idx_a(a) | keep
    ↪ order:true, desc, stats:pseudo |
+-----------------------------------+---------+-----------+-------------------------+-------------------------
    ↪
12 rows in set (0.01 sec)
```

11.3.3.2.6 谓词下推

本文档介绍 TiDB 逻辑优化规则中的谓词下推规则，旨在让读者对谓词下推形成理解，并了解常见的谓词下推适用及不适用的场景。

谓词下推将查询语句中的过滤表达式计算尽可能下推到距离数据源最近的地方，以尽早完成数据的过滤，进而显著地减少数据传输或计算的开销。

示例

以下通过一些例子对谓词下推优化进行说明，其中示例 1、2、3 为谓词下推适用的案例，示例 4、5、6 为谓词下推不适用的案例。

示例 1: 谓词下推到存储层

```
create table t(id int primary key, a int);
explain select * from t where a < 1;
+--
    ↪ ----------------------+----------+-----------+---------------+--------------------------------+
    ↪
| id                    | estRows  | task      | access object | operator info
    ↪                  |
+--
    ↪ ----------------------+----------+-----------+---------------+--------------------------------+
    ↪
```

```
| TableReader_7           | 3323.33  | root      |               | data:Selection_6
    ↪               |
| └─Selection_6           | 3323.33  | cop[tikv] |               | lt(test.t.a, 1)
    ↪                |
|   └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo |
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
3 rows in set (0.00 sec)
```

在该查询中，将谓词 `a < 1` 下推到 TiKV 上对数据进行过滤，可以减少由于网络传输带来的开销。

示例 2: 谓词下推到存储层

```
create table t(id int primary key, a int not null);
explain select * from t where a < substring('123', 1, 1);
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
| id                      | estRows  | task      | access object | operator info
    ↪                   |
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
| TableReader_7           | 3323.33  | root      |               | data:Selection_6
    ↪               |
| └─Selection_6           | 3323.33  | cop[tikv] |               | lt(test.t.a, 1)
    ↪                |
|   └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo |
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
```

该查询与示例 1 中的查询生成了完成一样的执行计划，这是因为谓词 `a < substring('123', 1, 1)` 的 substring 的入参均为常量，因此可以提前计算，进而简化得到等价的谓词 `a < 1`。进一步的，可以将 `a < 1` 下推至 TiKV 上。

示例 3: 谓词下推到 join 下方

```
create table t(id int primary key, a int not null);
create table s(id int primary key, a int not null);
explain select * from t join s on t.a = s.a where t.a < 1;
+--
    ↪ ----------------------------+----------+-----------+---------------+--------------------------------
    ↪
```

```
| id                          | estRows  | task      | access object | operator info
    ↪                              |
+--
    ↪ ---------------------------+----------+-----------+---------------+-------------------------------
    ↪
| HashJoin_8                  | 4154.17  | root      |               | inner join, equal:[eq(
    ↪ test.t.a, test.s.a)] |
| ├─TableReader_15(Build)     | 3323.33  | root      |               | data:Selection_14
    ↪                            |
| │ └─Selection_14            | 3323.33  | cop[tikv] |               | lt(test.s.a, 1)
    ↪                              |
| │   └─TableFullScan_13      | 10000.00 | cop[tikv] | table:s       | keep order:false, stats:
    ↪ pseudo             |
| └─TableReader_12(Probe)     | 3323.33  | root      |               | data:Selection_11
    ↪                            |
|   └─Selection_11            | 3323.33  | cop[tikv] |               | lt(test.t.a, 1)
    ↪                              |
|     └─TableFullScan_10      | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo             |
+--
    ↪ ---------------------------+----------+-----------+---------------+-------------------------------
    ↪
7 rows in set (0.00 sec)
```

在该查询中，将谓词 `t.a < 1` 下推到 join 前进行过滤，可以减少 join 时的计算开销。

此外，这条 SQL 执行的是内连接，且 ON 条件是 `t.a = s.a`，可以由 `t.a < 1` 推导出谓词 `s.a < 1`，并将其下推至 join 运算前对 s 表进行过滤，可以进一步减少 join 时的计算开销。

示例 4: 存储层不支持的谓词无法下推

```
create table t(id int primary key, a varchar(10) not null);
desc select * from t where truncate(a, " ") = '1';
+--
    ↪ ----------------------+----------+-----------+---------------+--------------------------------
    ↪
| id                    | estRows  | task      | access object | operator info
    ↪                                  |
+--
    ↪ ----------------------+----------+-----------+---------------+--------------------------------
    ↪
| Selection_5           | 8000.00  | root      |               | eq(truncate(cast(test.t.a,
    ↪ double BINARY), 0), 1) |
| └─TableReader_7       | 10000.00 | root      |               | data:TableFullScan_6
    ↪                               |
|   └─TableFullScan_6   | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo                    |
```

```
+--
    ↪ ----------------------+----------+-----------+---------------+--------------------------------
    ↪
```

在该查询中，存在谓词 `truncate(a, " ")= '1'`。

从 explain 结果中可以看到，该谓词没有被下推到 TiKV 上进行计算，这是因为 TiKV coprocessor 中没有对 `truncate` 内置函数进行支持，因此无法将其下推到 TiKV 上。

示例 5: 外连接中内表上的谓词不能下推

```
create table t(id int primary key, a int not null);
create table s(id int primary key, a int not null);
explain select * from t left join s on t.a = s.a where s.a is null;
+--
    ↪ ----------------------------+----------+-----------+--------------+--------------------------------
    ↪
| id                            | estRows  | task      | access object | operator info
    ↪                                  |
+--
    ↪ ----------------------------+----------+-----------+--------------+--------------------------------
    ↪
| Selection_7                   | 10000.00 | root      |               | isnull(test.s.a)
    ↪                                    |
| └─HashJoin_8                  | 12500.00 | root      |               | left outer join, equal
    ↪ :[eq(test.t.a, test.s.a)] |
|   ├─TableReader_13(Build)     | 10000.00 | root      |               | data:TableFullScan_12
    ↪                           |
|   │ └─TableFullScan_12        | 10000.00 | cop[tikv] | table:s       | keep order:false, stats
    ↪ :pseudo                   |
|   └─TableReader_11(Probe)     | 10000.00 | root      |               | data:TableFullScan_10
    ↪                           |
|     └─TableFullScan_10        | 10000.00 | cop[tikv] | table:t       | keep order:false, stats
    ↪ :pseudo                   |
+--
    ↪ ----------------------------+----------+-----------+--------------+--------------------------------
    ↪
6 rows in set (0.00 sec)
```

在该查询中，内表 s 上存在谓词 `s.a is null`。

从 explain 中可以看到，该谓词没有被下推到 join 前进行计算，这是因为外连接在不满足 on 条件时会对内表填充 NULL，而在该查询中 `s.a is null` 用来对 join 后的结果进行过滤，如果将其下推到 join 前在内表上进行过滤，则下推前后不等价，因此不可进行下推。

示例 6: 谓词中包含用户变量时不能下推

```
create table t(id int primary key, a char);
set @a = 1;
```

```
explain select * from t where a < @a;
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
| id                      | estRows  | task      | access object | operator info
    ↪                   |
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
| Selection_5             | 8000.00  | root      |               | lt(test.t.a, getvar("a"))
    ↪       |
| └─TableReader_7         | 10000.00 | root      |               | data:TableFullScan_6
    ↪            |
|   └─TableFullScan_6     | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo |
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
3 rows in set (0.00 sec)
```

在该查询中，表 t 上存在谓词 a < @a，其中 @a 为值为 1 的用户变量。

从 explain 中可以看到，该谓词没有像示例 2 中一样，将谓词简化为 a < 1 并下推到 TiKV 上进行计算。这是因为，用户变量 @a 的值可能会某些场景下在查询过程中发生改变，且 TiKV 对于用户变量 @a 的值不可知，因此 TiDB 不会将 @a 替换为 1，且不会下推至 TiKV 上进行计算。

一个帮助理解的例子如下：

```
create table t(id int primary key, a int);
insert into t values(1, 1), (2,2);
set @a = 1;
select id, a, @a:=@a+1 from t where a = @a;
+----+------+----------+
| id | a    | @a:=@a+1 |
+----+------+----------+
|  1 |    1 | 2        |
|  2 |    2 | 3        |
+----+------+----------+
2 rows in set (0.00 sec)
```

可以从在该查询中看到，@a 的值会在查询过程中发生改变，因此如果将 a = @a 替换为 a = 1 并下推至 TiKV，则优化前后不等价。

11.3.3.2.7 分区裁剪

分区裁剪是只有当目标表为分区表时，才可以进行的一种优化方式。分区裁剪通过分析查询语句中的过滤条件，只选择可能满足条件的分区，不扫描匹配不上的分区，进而显著地减少计算的数据量。

例如：

```
CREATE TABLE t1 (
 id INT NOT NULL PRIMARY KEY,
 pad VARCHAR(100)
)
PARTITION BY RANGE COLUMNS(id) (
 PARTITION p0 VALUES LESS THAN (100),
 PARTITION p1 VALUES LESS THAN (200),
 PARTITION p2 VALUES LESS THAN (MAXVALUE)
);
INSERT INTO t1 VALUES (1, 'test1'),(101, 'test2'), (201, 'test3');
EXPLAIN SELECT * FROM t1 WHERE id BETWEEN 80 AND 120;
```

```
+--
    ↪ -------------------------+---------+-----------+----------------------+-----------------------------
    ↪
| id                          | estRows | task      | access object          | operator info
    ↪                                  |
+--
    ↪ -------------------------+---------+-----------+----------------------+-----------------------------
    ↪
| PartitionUnion_8            | 80.00   | root      |                        |
    ↪                                                |
| ├─TableReader_10            | 40.00   | root      |                        | data:
    ↪ TableRangeScan_9                             |
| │ └─TableRangeScan_9        | 40.00   | cop[tikv] | table:t1, partition:p0 | range:[80,120],
    ↪ keep order:false, stats:pseudo |
| └─TableReader_12            | 40.00   | root      |                        | data:
    ↪ TableRangeScan_11                            |
|   └─TableRangeScan_11       | 40.00   | cop[tikv] | table:t1, partition:p1 | range:[80,120],
    ↪ keep order:false, stats:pseudo |
+--
    ↪ -------------------------+---------+-----------+----------------------+-----------------------------
    ↪
5 rows in set (0.00 sec)
```

分区裁剪的使用场景

分区表有 Range 分区和 hash 分区两种形式，分区裁剪对两种分区表也有不同的使用场景。

分区裁剪在 Hash 分区表上的应用

Hash 分区表上可以使用分区裁剪的场景

只有等值比较的查询条件能够支持 Hash 分区表的裁剪。

```
create table t (x int) partition by hash(x) partitions 4;
explain select * from t where x = 1;
```

```
+--
    ↪ ----------------------+----------+-----------+----------------------+------------------------------+
    ↪
| id                      | estRows  | task      | access object        | operator info
    ↪                  |
+--
    ↪ ----------------------+----------+-----------+----------------------+------------------------------+
    ↪
| TableReader_8           | 10.00    | root      |                      | data:Selection_7
    ↪               |
| └─Selection_7           | 10.00    | cop[tikv] |                      | eq(test.t.x, 1)
    ↪                |
|   └─TableFullScan_6     | 10000.00 | cop[tikv] | table:t, partition:p1 | keep order:false,
    ↪ stats:pseudo |
+--
    ↪ ----------------------+----------+-----------+----------------------+------------------------------+
    ↪
```

在这条 SQL 中，由条件 `x = 1` 可以知道所有结果均在一个分区上。数值 1 在经过 Hash 后，可以确定其在分区 p1 中。因此只需要扫描分区 p1，而无需访问一定不会出现相关结果的 p2、p3、p4 分区。从执行计划来看，其中只出现了一个 TableFullScan 算子，且在 access object 中指定了 p1 分区，确认 partition pruning 生效了。

Hash 分区表上不能使用分区裁剪的场景

场景一

不能确定查询结果只在一个分区上的条件：如 in, between, `>` `<` `>=` `<=` 等查询条件，不能使用分区裁剪的优化。

```
create table t (x int) partition by hash(x) partitions 4;
explain select * from t where x > 2;
```

```
+--
    ↪ --------------------------+----------+-----------+----------------------+------------------------------
    ↪
| id                          | estRows  | task      | access object        | operator info
    ↪                  |
+--
    ↪ --------------------------+----------+-----------+----------------------+------------------------------
    ↪
| Union_10                    | 13333.33 | root      |                      |
    ↪                              |
| ├─TableReader_13            | 3333.33  | root      |                      | data:
    ↪ Selection_12              |
| │ └─Selection_12            | 3333.33  | cop[tikv] |                      | gt(test.t.x, 2)
    ↪                |
```

```
| │   └─TableFullScan_11       | 10000.00 | cop[tikv] | table:t, partition:p0 | keep order:false
    ↪ , stats:pseudo |
| ├─TableReader_16             | 3333.33  | root      |                       | data:
    ↪ Selection_15             |
| │ └─Selection_15             | 3333.33  | cop[tikv] |                       | gt(test.t.x, 2)
    ↪                 |
| │   └─TableFullScan_14       | 10000.00 | cop[tikv] | table:t, partition:p1 | keep order:false
    ↪ , stats:pseudo |
| ├─TableReader_19             | 3333.33  | root      |                       | data:
    ↪ Selection_18             |
| │ └─Selection_18             | 3333.33  | cop[tikv] |                       | gt(test.t.x, 2)
    ↪                 |
| │   └─TableFullScan_17       | 10000.00 | cop[tikv] | table:t, partition:p2 | keep order:false
    ↪ , stats:pseudo |
| └─TableReader_22             | 3333.33  | root      |                       | data:
    ↪ Selection_21             |
|   └─Selection_21             | 3333.33  | cop[tikv] |                       | gt(test.t.x, 2)
    ↪                 |
|     └─TableFullScan_20       | 10000.00 | cop[tikv] | table:t, partition:p3 | keep order:false
    ↪ , stats:pseudo |
+--
    ↪ ---------------------------+----------+-----------+-----------------------+-----------------------------
    ↪
```

在这条 SQL 中，`x > 2` 条件无法确定对应的 Hash Partition，所以不能使用分区裁剪。

场景二

由于分区裁剪的规则优化是在查询计划的生成阶段，对于执行阶段才能获取到过滤条件的场景，无法利用分区裁剪的优化。

```
create table t (x int) partition by hash(x) partitions 4;
explain select * from t2 where x = (select * from t1 where t2.x = t1.x and t2.x < 2);
```

```
+--
    ↪ ----------------------------------+----------+-----------+-----------------------+-------------------
    ↪
| id                                  | estRows  | task      | access object         | operator
    ↪  info                               |
+--
    ↪ ----------------------------------+----------+-----------+-----------------------+-------------------
    ↪
| Projection_13                       | 9990.00  | root      |                       | test.t2.
    ↪ x                                   |
| └─Apply_15                           | 9990.00  | root      |                       | inner
    ↪ join, equal:[eq(test.t2.x, test.t1.x)] |
```

```
|   ├─TableReader_18(Build)            | 9990.00  | root      |                         | data:
    ↪ Selection_17                              |
|   │ └─Selection_17                   | 9990.00  | cop[tikv] |                         | not(
    ↪ isnull(test.t2.x))                         |
|   │   └─TableFullScan_16             | 10000.00 | cop[tikv] | table:t2                | keep
    ↪ order:false, stats:pseudo                 |
|   └─Selection_19(Probe)              | 0.80     | root      |                         | not(
    ↪ isnull(test.t1.x))                         |
|     └─MaxOneRow_20                   | 1.00     | root      |                         |
    ↪                                           |
|       └─Union_21                     | 2.00     | root      |                         |
    ↪                                           |
|         ├─TableReader_24             | 2.00     | root      |                         | data:
    ↪ Selection_23                              |
|         │ └─Selection_23             | 2.00     | cop[tikv] |                         | eq(test
    ↪ .t2.x, test.t1.x), lt(test.t2.x, 2)    |
|         │   └─TableFullScan_22       | 2500.00  | cop[tikv] | table:t1, partition:p0  | keep
    ↪ order:false, stats:pseudo                 |
|         └─TableReader_27             | 2.00     | root      |                         | data:
    ↪ Selection_26                              |
|           └─Selection_26             | 2.00     | cop[tikv] |                         | eq(test
    ↪ .t2.x, test.t1.x), lt(test.t2.x, 2)    |
|             └─TableFullScan_25       | 2500.00  | cop[tikv] | table:t1, partition:p1  | keep
    ↪ order:false, stats:pseudo                 |
+--
    ↪ ------------------------------------+----------+-----------+-------------------------+-------------------
    ↪
```

这个查询每从 `t2` 读取一行，都会去分区表 `t1` 上进行查询，理论上这时会满足 `t1.x = val` 的过滤条件，但实际上由于分区裁剪只作用于查询计划生成阶段，而不是执行阶段，因而不会做裁剪。

分区裁剪在 Range 分区表上的应用

Range 分区表上可以使用分区裁剪的场景

场景一

等值比较的查询条件可以使用分区裁剪。

```
create table t (x int) partition by range (x) (
    partition p0 values less than (5),
    partition p1 values less than (10),
    partition p2 values less than (15)
    );
explain select * from t where x = 3;
```

```
+--
    ↪ ----------------------+----------+-----------+----------------------+------------------------------+
    ↪
```

```
| id                      | estRows  | task      | access object         | operator info
    ↪                  |
+--
    ↪ ----------------------+----------+-----------+----------------------+------------------------------+
    ↪
| TableReader_8           | 10.00    | root      |                       | data:Selection_7
    ↪               |
| └─Selection_7           | 10.00    | cop[tikv] |                       | eq(test.t.x, 3)
    ↪                |
|   └─TableFullScan_6     | 10000.00 | cop[tikv] | table:t, partition:p0 | keep order:false,
    ↪ stats:pseudo |
+--
    ↪ ----------------------+----------+-----------+----------------------+------------------------------+
    ↪
```

使用 in 条件的等值比较查询条件也可以使用分区裁剪。

```
create table t (x int) partition by range (x) (
    partition p0 values less than (5),
    partition p1 values less than (10),
    partition p2 values less than (15)
    );
explain select * from t where x in(1,13);
```

```
+--
    ↪ --------------------------+----------+-----------+----------------------+------------------------------
    ↪
| id                          | estRows  | task      | access object         | operator info
    ↪                  |
+--
    ↪ --------------------------+----------+-----------+----------------------+------------------------------
    ↪
| Union_8                     | 40.00    | root      |                       |
    ↪                                |
| ├─TableReader_11            | 20.00    | root      |                       | data:Selection_10
    ↪               |
| │ └─Selection_10            | 20.00    | cop[tikv] |                       | in(test.t.x, 1,
    ↪ 13)            |
| │   └─TableFullScan_9       | 10000.00 | cop[tikv] | table:t, partition:p0 | keep order:false,
    ↪  stats:pseudo |
| └─TableReader_14            | 20.00    | root      |                       | data:Selection_13
    ↪               |
|   └─Selection_13            | 20.00    | cop[tikv] |                       | in(test.t.x, 1,
    ↪ 13)            |
|     └─TableFullScan_12      | 10000.00 | cop[tikv] | table:t, partition:p2 | keep order:false,
    ↪  stats:pseudo |
```

```
+--
    ↪ -------------------------+----------+-----------+----------------------+-----------------------------
    ↪
```

在这条 SQL 中，由条件 `x in(1,13)` 可以知道所有结果只会分布在几个分区上。经过分析，发现所有 `x = 1` 的记录都在分区 `p0` 上，所有 `x = 13` 的记录都在分区 `p2` 上，因此只需要访问 `p0`、`p2` 这两个分区，

场景二

区间比较的查询条件如 `between, > < = >= <=` 可以使用分区裁剪。

```
create table t (x int) partition by range (x) (
    partition p0 values less than (5),
    partition p1 values less than (10),
    partition p2 values less than (15)
    );
explain select * from t where x between 7 and 14;
```

```
+--
    ↪ -------------------------+----------+-----------+----------------------+-----------------------------
    ↪
| id                          | estRows  | task      | access object        | operator info
    ↪                     |
+--
    ↪ -------------------------+----------+-----------+----------------------+-----------------------------
    ↪
| Union_8                     | 500.00   | root      |                      |
    ↪                                 |
| ├─TableReader_11            | 250.00   | root      |                      | data:Selection_10
    ↪                   |
| │ └─Selection_10            | 250.00   | cop[tikv] |                      | ge(test.t.x, 7),
    ↪ le(test.t.x, 14) |
| │   └─TableFullScan_9       | 10000.00 | cop[tikv] | table:t, partition:p1 | keep order:false,
    ↪  stats:pseudo    |
| └─TableReader_14            | 250.00   | root      |                      | data:Selection_13
    ↪                   |
|   └─Selection_13            | 250.00   | cop[tikv] |                      | ge(test.t.x, 7),
    ↪ le(test.t.x, 14) |
|     └─TableFullScan_12      | 10000.00 | cop[tikv] | table:t, partition:p2 | keep order:false,
    ↪  stats:pseudo    |
+--
    ↪ -------------------------+----------+-----------+----------------------+-----------------------------
    ↪
```

场景三

分区表达式为 `fn(col)` 的简单形式，查询条件是 `> < = >= <=` 之一，且 `fn` 是单调函数，可以使用分区裁剪。

关于 fn 函数，对于任意 x y，如果 x > y，则 fn(x)> fn(y)，那么这种是严格递增的单调函数。非严格递增的单调函数也可以符合分区裁剪要求，只要函数 fn 满足：对于任意 x y，如果 x > y，则 fn(x)>= fn(y)。理论上，所有满足单调条件（严格或者非严格）的函数都支持分区裁剪。目前，TiDB 支持的单调函数如下：

- UNIX_TIMESTAMP()
- TO_DAYS()

例如，分区表达式是 fn(col) 形式，fn 为我们支持的单调函数 to_days，就可以使用分区裁剪：

```
create table t (id datetime) partition by range (to_days(id)) (
    partition p0 values less than (to_days('2020-04-01')),
    partition p1 values less than (to_days('2020-05-01')));
explain select * from t where id > '2020-04-18';
```

```
+--
    ↪ ----------------------+----------+-----------+----------------------+-------------------------------
    ↪
| id                      | estRows  | task      | access object        | operator info
    ↪                              |
+--
    ↪ ----------------------+----------+-----------+----------------------+-------------------------------
    ↪
| TableReader_8           | 3333.33  | root      |                      | data:Selection_7
    ↪                            |
| └─Selection_7           | 3333.33  | cop[tikv] |                      | gt(test.t.id,
    ↪ 2020-04-18 00:00:00.000000) |
|   └─TableFullScan_6     | 10000.00 | cop[tikv] | table:t, partition:p1 | keep order:false,
    ↪ stats:pseudo             |
+--
    ↪ ----------------------+----------+-----------+----------------------+-------------------------------
    ↪
```

Range 分区表上不能使用分区裁剪的场景

由于分区裁剪的规则优化是在查询计划的生成阶段，对于执行阶段才能获取到过滤条件的场景，无法利用分区裁剪的优化。

```
create table t1 (x int) partition by range (x) (
    partition p0 values less than (5),
    partition p1 values less than (10));
create table t2 (x int);
explain select * from t2 where x < (select * from t1 where t2.x < t1.x and t2.x < 2);
```

```
+--
    ↪ ----------------------------------+----------+-----------+----------------------+-----------------
    ↪
```

```
| id                                | estRows  | task      | access object          | operator
    ↪  info                                          |
+--
    ↪ ----------------------------------+----------+-----------+------------------------+------------------
    ↪
| Projection_13                       | 9990.00  | root      |                        | test.t2.
    ↪ x                                              |
| └─Apply_15                          | 9990.00  | root      |                        |
    ↪ CARTESIAN inner join, other cond:lt(test.t2.x, test.t1.x) |
|   ├─TableReader_18(Build)           | 9990.00  | root      |                        | data:
    ↪ Selection_17                                   |
|   │ └─Selection_17                  | 9990.00  | cop[tikv] |                        | not(
    ↪ isnull(test.t2.x))                              |
|   │   └─TableFullScan_16            | 10000.00 | cop[tikv] | table:t2               | keep
    ↪ order:false, stats:pseudo                       |
|   └─Selection_19(Probe)             | 0.80     | root      |                        | not(
    ↪ isnull(test.t1.x))                              |
|     └─MaxOneRow_20                  | 1.00     | root      |                        |
    ↪                                                 |
|       └─Union_21                    | 2.00     | root      |                        |
    ↪                                                 |
|         ├─TableReader_24            | 2.00     | root      |                        | data:
    ↪ Selection_23                                   |
|         │ └─Selection_23            | 2.00     | cop[tikv] |                        | lt(test
    ↪ .t2.x, 2), lt(test.t2.x, test.t1.x)             |
|         │   └─TableFullScan_22      | 2.50     | cop[tikv] | table:t1, partition:p0 | keep
    ↪ order:false, stats:pseudo                       |
|         └─TableReader_27            | 2.00     | root      |                        | data:
    ↪ Selection_26                                   |
|           └─Selection_26            | 2.00     | cop[tikv] |                        | lt(test
    ↪ .t2.x, 2), lt(test.t2.x, test.t1.x)             |
|             └─TableFullScan_25      | 2.50     | cop[tikv] | table:t1, partition:p1 | keep
    ↪ order:false, stats:pseudo                       |
+--
    ↪ ----------------------------------+----------+-----------+------------------------+------------------
    ↪
14 rows in set (0.00 sec)
```

这个查询每从 t2 读取一行，都会去分区表 t1 上进行查询，理论上这时会满足 t1.x > val 的过滤条件，但实际上由于分区裁剪只作用于查询计划生成阶段，而不是执行阶段，因而不会做裁剪。

11.3.3.2.8 TopN 和 Limit 下推

SQL 中的 LIMIT 子句在 TiDB 查询计划树中对应 Limit 算子节点，ORDER BY 子句在查询计划树中对应 Sort 算子节点，此外，我们会将相邻的 Limit 和 Sort 算子组合成 TopN 算子节点，表示按某个排序规则提取记录的前 N 项。

从另一方面来说，Limit 节点等价于一个排序规则为空的 TopN 节点。

和谓词下推类似，TopN（及 Limit，下同）下推将查询计划树中的 TopN 计算尽可能下推到距离数据源最近的地方，以尽早完成数据的过滤，进而显著地减少数据传输或计算的开销。

如果要关闭这个规则，可参照优化规则及表达式下推的黑名单中的关闭方法。

示例

以下通过一些例子对 TopN 下推进行说明。

示例 1：下推到存储层 Coprocessor

```
create table t(id int primary key, a int not null);
explain select * from t order by a limit 10;
```

```
+---------------------------+----------+-----------+---------------+--------------------------------+
    ↪
| id                        | estRows  | task      | access object | operator info
    ↪                  |
+---------------------------+----------+-----------+---------------+--------------------------------+
    ↪
| TopN_7                    | 10.00    | root      |               | test.t.a, offset:0, count
    ↪ :10   |
| └─TableReader_15           | 10.00    | root      |               | data:TopN_14
    ↪                   |
|   └─TopN_14                | 10.00    | cop[tikv] |               | test.t.a, offset:0, count
    ↪ :10   |
|     └─TableFullScan_13     | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo |
+---------------------------+----------+-----------+---------------+--------------------------------+
    ↪
4 rows in set (0.00 sec)
```

在该查询中，将 TopN 算子节点下推到 TiKV 上对数据进行过滤，每个 Coprocessor 只向 TiDB 传输 10 条记录。在 TiDB 将数据整合后，再进行最终的过滤。

示例 2：TopN 下推过 Join 的情况（排序规则仅依赖于外表中的列）

```
create table t(id int primary key, a int not null);
create table s(id int primary key, a int not null);
explain select * from t left join s on t.a = s.a order by t.a limit 10;
```

```
+---------------------------------+----------+-----------+---------------+--------------------------------
    ↪
| id                              | estRows  | task      | access object | operator info
    ↪                                    |
+---------------------------------+----------+-----------+---------------+--------------------------------
    ↪
```

```
| TopN_12                          | 10.00    | root      |               | test.t.a, offset:0,
    ↪ count:10                     |
| └─HashJoin_17                    | 12.50    | root      |               | left outer join,
    ↪ equal:[eq(test.t.a, test.s.a)] |
|   ├─TopN_18(Build)               | 10.00    | root      |               | test.t.a, offset:0,
    ↪ count:10                     |
|   │ └─TableReader_26             | 10.00    | root      |               | data:TopN_25
    ↪                                    |
|   │   └─TopN_25                  | 10.00    | cop[tikv] |               | test.t.a, offset:0,
    ↪ count:10                     |
|   │     └─TableFullScan_24       | 10000.00 | cop[tikv] | table:t       | keep order:false,
    ↪ stats:pseudo                   |
|   └─TableReader_30(Probe)        | 10000.00 | root      |               | data:
    ↪ TableFullScan_29                           |
|     └─TableFullScan_29           | 10000.00 | cop[tikv] | table:s       | keep order:false,
    ↪ stats:pseudo                   |
+----------------------------------+----------+-----------+---------------+--------------------------------
    ↪
8 rows in set (0.01 sec)
```

在该查询中，TopN 算子的排序规则仅依赖于外表 t 中的列，可以将 TopN 下推到 Join 之前进行一次计算，以减少 Join 时的计算开销。除此之外，TiDB 同样将 TopN 下推到了存储层中。

示例 3：TopN 不能下推过 Join 的情况

```
create table t(id int primary key, a int not null);
create table s(id int primary key, a int not null);
explain select * from t join s on t.a = s.a order by t.id limit 10;
```

```
+--------------------------------+----------+-----------+---------------+--------------------------------
    ↪
| id                             | estRows  | task      | access object | operator info
    ↪                                |
+--------------------------------+----------+-----------+---------------+--------------------------------
    ↪
| TopN_12                        | 10.00    | root      |               | test.t.id, offset:0,
    ↪ count:10                |
| └─HashJoin_16                  | 12500.00 | root      |               | inner join, equal:[eq(
    ↪ test.t.a, test.s.a)] |
|   ├─TableReader_21(Build)      | 10000.00 | root      |               | data:TableFullScan_20
    ↪                         |
|   │ └─TableFullScan_20         | 10000.00 | cop[tikv] | table:s       | keep order:false, stats
    ↪ :pseudo              |
|   └─TableReader_19(Probe)      | 10000.00 | root      |               | data:TableFullScan_18
    ↪                         |
|     └─TableFullScan_18         | 10000.00 | cop[tikv] | table:t       | keep order:false, stats
    ↪ :pseudo              |
```

```
+-------------------------------+----------+-----------+---------------+-------------------------------------------
    ↪
6 rows in set (0.00 sec)
```

TopN 无法下推过 Inner Join。以上面的查询为例，如果先 Join 得到 100 条记录，再做 TopN 可以剩余 10 条记录。而如果在 TopN 之前就过滤到剩余 10 条记录，做完 Join 之后可能就剩下 5 条了，导致了结果的差异。

同理，TopN 无法下推到 Outer Join 的内表上。在 TopN 的排序规则涉及多张表上的列时，也无法下推，如 `t.a+s.a`。只有当 TopN 的排序规则仅依赖于外表上的列时，才可以下推。

示例 4：TopN 转换成 Limit 的情况

```
create table t(id int primary key, a int not null);
create table s(id int primary key, a int not null);
explain select * from t left join s on t.a = s.a order by t.id limit 10;
```

```
+----------------------------------+----------+-----------+---------------+-------------------------------------------
    ↪
| id                               | estRows  | task      | access object | operator info
    ↪                                    |
+----------------------------------+----------+-----------+---------------+-------------------------------------------
    ↪
| TopN_12                          | 10.00    | root      |               | test.t.id, offset:0,
    ↪ count:10                    |
| └─HashJoin_17                    | 12.50    | root      |               | left outer join,
    ↪ equal:[eq(test.t.a, test.s.a)] |
|   ├─Limit_21(Build)              | 10.00    | root      |               | offset:0, count:10
    ↪                              |
|   │ └─TableReader_31             | 10.00    | root      |               | data:Limit_30
    ↪                                    |
|   │   └─Limit_30                 | 10.00    | cop[tikv] |               | offset:0, count:10
    ↪                              |
|   │     └─TableFullScan_29       | 10.00    | cop[tikv] | table:t       | keep order:true,
    ↪ stats:pseudo                   |
|   └─TableReader_35(Probe)        | 10000.00 | root      |               | data:
    ↪ TableFullScan_34                          |
|     └─TableFullScan_34           | 10000.00 | cop[tikv] | table:s       | keep order:false,
    ↪ stats:pseudo                  |
+----------------------------------+----------+-----------+---------------+-------------------------------------------
    ↪
8 rows in set (0.00 sec)
```

在上面的查询中，TopN 首先推到了外表 t 上。然后因为它要对 `t.id` 进行排序，而 `t.id` 是表 t 的主键，可以直接按顺序读出 (`keep order:true`)，从而省略了 TopN 中的排序，将其简化为 Limit。

11.3.3.2.9 Join Reorder 算法简介

在实际的业务场景中，多个表的 Join 语句是很常见的，而 Join 的执行效率和各个表参与 Join 的顺序有关系。如 `select * from t1, t2, t3 where t1.a=t2.a and t3.a=t2.a`，这个 SQL 中可能的执行顺序有 “t1 和 t2 先做 Join，然后再和 t3 做 Join” 以及 “t2 和 t3 先做 Join，然后再和 t1 做 Join” 两种情况。根据 `t1` 和 `t3` 的数据量及数据分布，这两种执行顺序会有不同的性能表现。

因此优化器需要实现一种决定 Join 顺序的算法。目前 TiDB 中存在两种 Join Reorder 算法，贪心算法和动态规划算法。

- Join Reorder 贪心算法：在所有参与 Join 的节点中，选择行数最小的表与其他各表分别做一次 Join 的结果估算，然后选择其中结果最小的一对进行 Join，再继续这个过程进入下一轮的选择和 Join，直到所有的节点都完成 Join。
- Join Reorder 动态规划算法：在所有参与 Join 的节点中，枚举所有可能的 Join 顺序，然后选择最优的 Join 顺序。

Join Reorder 贪心算法实例

以三个表 t1、t2、t3 的 Join 为例。首先获取所有参与 Join 的节点，将所有节点按照行数多少，从少到多进行排序。

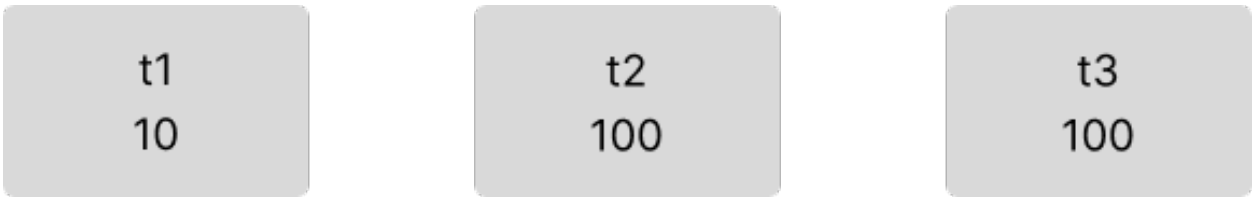


图 163: join-reorder-1

之后选定其中最小的表，将其与其他两个表分别做一次 Join，观察输出的结果集大小，选择其中结果更小的一对。

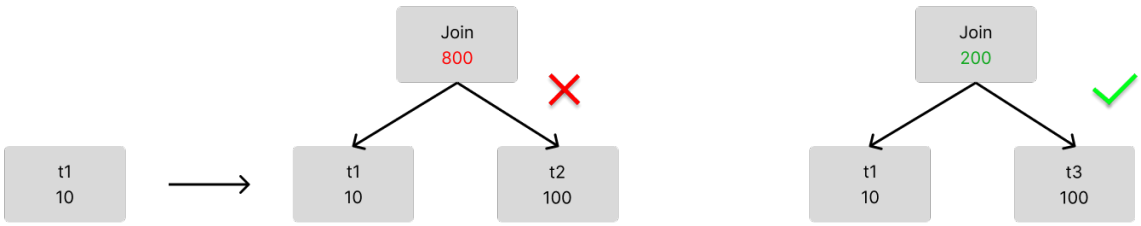


图 164: join-reorder-2

然后进入下一轮的选择，如果这时是四个表，那么就继续比较输出结果集的大小，进行选择。这里只有三个表，因此就直接得到了最终的 Join 结果。

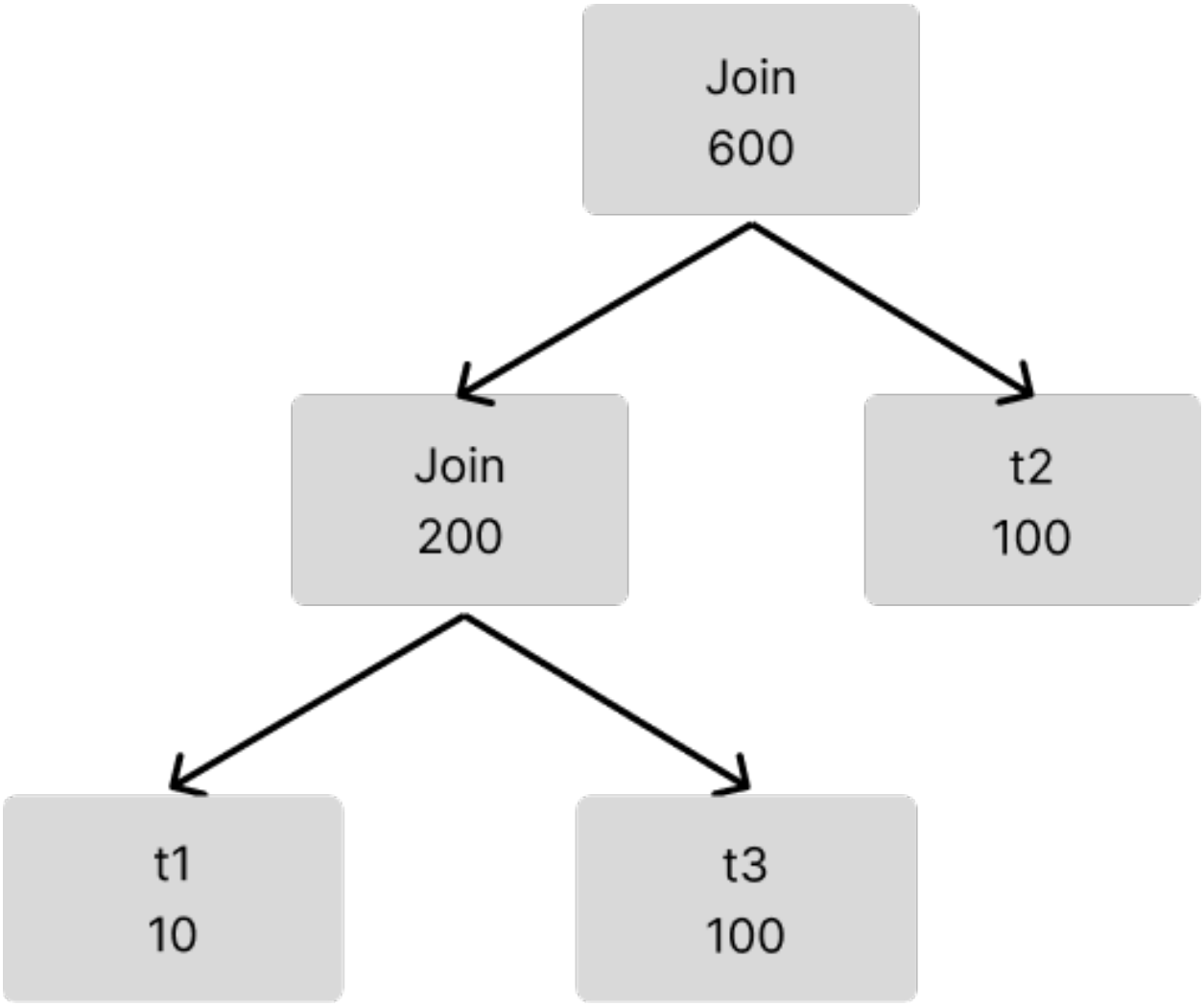


图 165: join-reorder-3

Join Reorder 动态规划算法实例

仍然以上述例子为例。动态规划算法会枚举所有的可能性，因此相对贪心算法必须从 t1 表开始枚举，动态规划算法可以枚举如下的 Join 顺序。

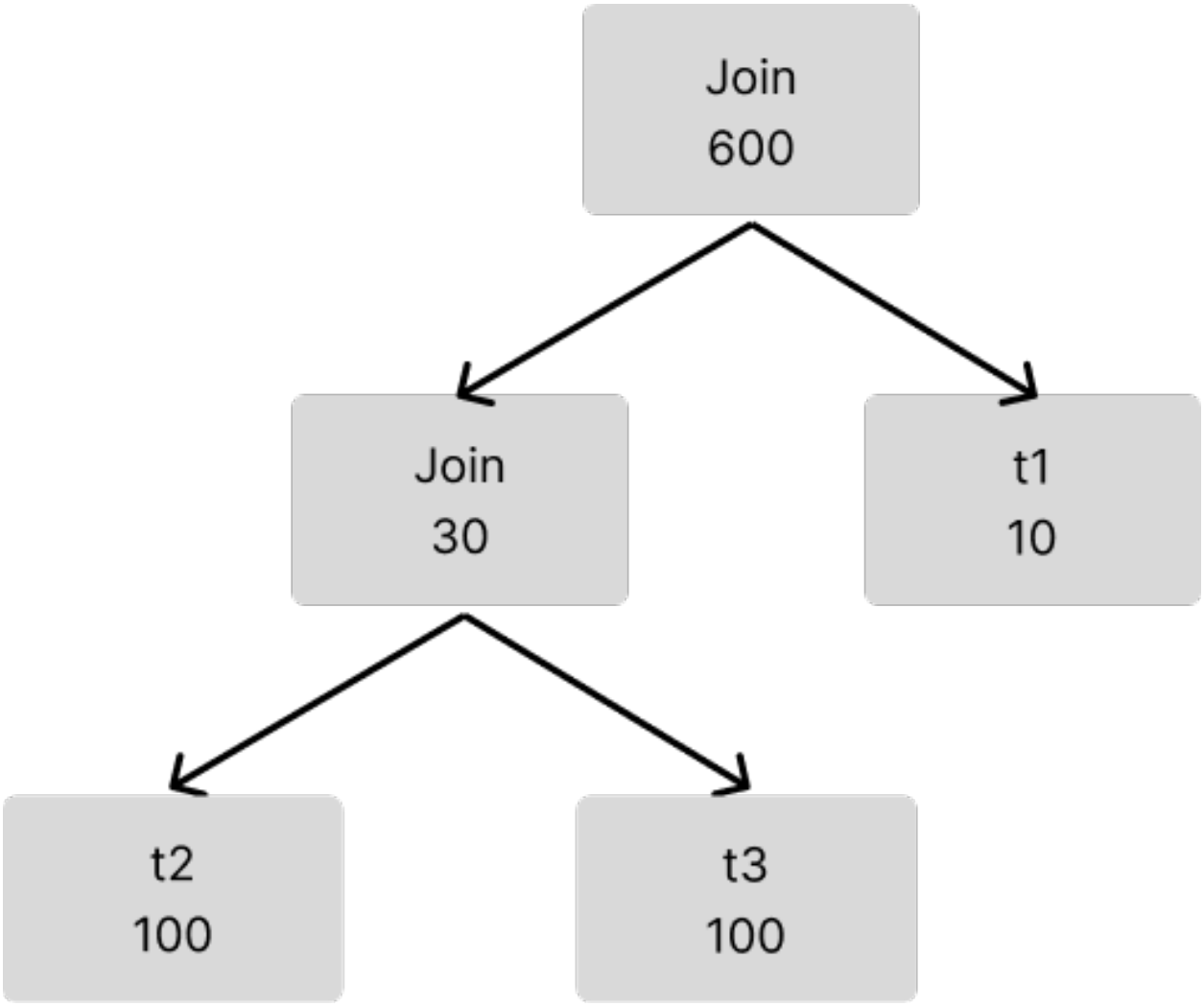


图 166: join-reorder-4

当该选择比贪心算法更优时，动态规划算法便可以选择到更优的 Join 顺序。

相应地，因为会枚举所有的可能性，动态规划算法会消耗更多的时间，也会更容易受统计信息影响。

Join Reorder 算法的控制

目前 Join Reorder 算法由变量`tidb_opt_join_reorder_threshold`控制，当参与 Join Reorder 的节点个数大于该阈值时选择贪心算法，反之选择动态规划算法。

Join Reorder 算法限制

当前的 Join Reorder 算法存在如下限制：

- 受结果集的计算算法所限并不会保证一定会选到合适的 Join order
- 是否启用 Outer Join 的 Join Reorder 功能由系统变量`tidb_enable_outer_join_reorder`控制。
- 目前动态规划算法无法进行 Outer Join 的 Join Reorder。

目前 TiDB 中支持使用 `STRAIGHT_JOIN` 语法来强制指定一种 Join 顺序，参见语法元素说明。

11.3.3.2.10 从窗口函数中推导 TopN 或 Limit

窗口函数是一种常见的 SQL 函数。对于 ROW_NUMBER() 或者 RANK() 等编号相关的窗口函数，一种常见的用法是在进行窗口函数求值之后，对求值的结果进行过滤，例如：

```
SELECT * FROM (SELECT ROW_NUMBER() OVER (ORDER BY a) AS rownumber FROM t) dt WHERE rownumber <= 3
```

按照正常的 SQL 执行流程，TiDB 需要先对 t 表的所有数据进行排序，然后为每一行都求得相应的 ROW_NUMBER() 结果，最后再进行 rownumber <= 3 的过滤。

从 v7.0.0 开始，TiDB 支持从窗口函数中推导 TopN 或 Limit 算子。通过该优化规则，TiDB 可以将原始 SQL 等价改写成以下形式：

```
WITH t_topN AS (SELECT a FROM t1 ORDER BY a LIMIT 3) SELECT * FROM (SELECT ROW_NUMBER() OVER (
    ↪ ORDER BY a) AS rownumber FROM t_topN) dt WHERE rownumber <= 3
```

可以看出，改写后，TiDB 可以从窗口函数与后续的过滤条件中推导出一个 TopN 算子，相比于原始 SQL 中的 Sort 算子（对应 ORDER BY），TopN 算子的运行效率远高于 Sort 算子，而且 TiKV 和 TiFlash 均支持 TopN 算子的下推，因此能进一步提升改写后的 SQL 的性能。

从窗口函数中推导 TopN 或 Limit 默认关闭。你可以通过将 session 变量tidb_opt_derive_topn 设置为 ON 开启该功能。

开启后，如需关闭，可以进行以下操作之一：

- 设置 session 变量tidb_opt_derive_topn 为 false。
- 可参照优化规则及表达式下推的黑名单中的关闭方法。

限制

- 目前仅 ROW_NUMBER() 窗口函数支持 SQL 语句改写。
- 只有当 SQL 语句的过滤条件是针对 ROW_NUMBER() 结果而且过滤条件为 < 或者 <= 时，TiDB 才支持改写 SQL 语句。

示例

以下通过一些例子对该优化规则进行说明。

不包含 PARTITION BY 的窗口函数

示例 1：不包含 ORDER BY 的窗口函数

```
CREATE TABLE t(id int, value int);
SET tidb_opt_derive_topn=on;
EXPLAIN SELECT * FROM (SELECT ROW_NUMBER() OVER () AS rownumber FROM t) dt WHERE rownumber <= 3;
```

```
+---------------------------------+---------+-----------+---------------+--------------------------------
    ↪
| id                              | estRows | task      | access object | operator info
    ↪                                                                   |
+---------------------------------+---------+-----------+---------------+--------------------------------
    ↪
```

```
| Projection_9                     | 2.40    | root      |               | Column#5
    ↪                                                                  |
| └─Selection_10                   | 2.40    | root      |               | le(Column#5, 3)
    ↪                                                                  |
|   └─Window_11                    | 3.00    | root      |               | row_number()->Column
    ↪ #5 over(rows between current row and current row) |
|     └─Limit_15                   | 3.00    | root      |               | offset:0, count:3
    ↪                                                                  |
|       └─TableReader_26           | 3.00    | root      |               | data:Limit_25
    ↪                                                                  |
|         └─Limit_25               | 3.00    | cop[tikv] |               | offset:0, count:3
    ↪                                                                  |
|           └─TableFullScan_24     | 3.00    | cop[tikv] | table:t       | keep order:false,
    ↪ stats:pseudo                                                     |
+----------------------------------+---------+-----------+---------------+--------------------------------
    ↪
```

在该查询中，优化器从窗口函数中推导出来了 Limit 并将它下推给了 TiKV。

示例 2：包含 ORDER BY 的窗口函数

```
CREATE TABLE t(id int, value int);
SET tidb_opt_derive_topn=on;
EXPLAIN SELECT * FROM (SELECT ROW_NUMBER() OVER (ORDER BY value) AS rownumber FROM t) dt WHERE
    ↪ rownumber <= 3;
```

```
+----------------------------------+----------+-----------+---------------+--------------------------------
    ↪
| id                               | estRows  | task      | access object | operator info
    ↪                                                                                  |
+----------------------------------+----------+-----------+---------------+--------------------------------
    ↪
| Projection_10                    | 2.40     | root      |               | Column#5
    ↪                                                                                  |
| └─Selection_11                   | 2.40     | root      |               | le(Column#5, 3)
    ↪                                                                                  |
|   └─Window_12                    | 3.00     | root      |               | row_number()->Column
    ↪ #5 over(order by test.t.value rows between current row and current row) |
|     └─TopN_13                    | 3.00     | root      |               | test.t.value, offset
    ↪ :0, count:3                                                                      |
|       └─TableReader_25           | 3.00     | root      |               | data:TopN_24
    ↪                                                                                  |
|         └─TopN_24                | 3.00     | cop[tikv] |               | test.t.value, offset
    ↪ :0, count:3                                                                      |
|           └─TableFullScan_23     | 10000.00 | cop[tikv] | table:t       | keep order:false,
    ↪ stats:pseudo                                                                     |
```

```
+-----------------------------------+----------+-----------+---------------+-------------------------------------------
    ↪
```

在该查询中，优化器从窗口函数中推导出来了 TopN 并将它下推给了 TiKV。

包含 PARTITION BY 的窗口函数

> **注意：**
>
> 当窗口函数包含 PARTITION BY 时，该优化规则仅在 partition 列是主键的前缀且主键是聚簇索引的时候才能生效。

示例 3：不包含 ORDER BY 的窗口函数

```
CREATE TABLE t(id1 int, id2 int, value1 int, value2 int, primary key(id1,id2) clustered);
SET tidb_opt_derive_topn=on;
EXPLAIN SELECT * FROM (SELECT ROW_NUMBER() OVER (PARTITION BY id1) AS rownumber FROM t) dt WHERE
    ↪ rownumber <= 3;
```

```
+-----------------------------------+---------+-----------+---------------+-------------------------------------------
    ↪
| id                                | estRows | task      | access object | operator info
    ↪                                                                                  |
+-----------------------------------+---------+-----------+---------------+-------------------------------------------
    ↪
| Projection_10                     | 2.40    | root      |               | Column#6
    ↪                                                                                  |
| └─Selection_11                    | 2.40    | root      |               | le(Column#6, 3)
    ↪                                                                                  |
|   └─Shuffle_26                    | 3.00    | root      |               | execution info:
    ↪ concurrency:2, data sources:[TableReader_24]                                     |
|     └─Window_12                   | 3.00    | root      |               | row_number()->
    ↪ Column#6 over(partition by test.t.id1 rows between current row and current row) |
|       └─Sort_25                   | 3.00    | root      |               | test.t.id1
    ↪                                                                                  |
|         └─TableReader_24          | 3.00    | root      |               | data:Limit_23
    ↪                                                                                  |
|           └─Limit_23              | 3.00    | cop[tikv] |               | partition by test.t
    ↪ .id1, offset:0, count:3                                                          |
|             └─TableFullScan_22    | 3.00    | cop[tikv] | table:t       | keep order:false,
    ↪ stats:pseudo                                                                     |
+-----------------------------------+---------+-----------+---------------+-------------------------------------------
    ↪
```

在该查询中，优化器从窗口函数中推导出来了 Limit 并将它下推给了 TiKV。值得一提的是这个 Limit 其实是 partition Limit，也就是说对于每个相同 id1 值组成的一组数据上都会进行一次 Limit。

示例 4：包含 ORDER BY 的窗口函数

```
CREATE TABLE t(id1 int, id2 int, value1 int, value2 int, primary key(id1,id2) clustered);
SET tidb_opt_derive_topn=on;
EXPLAIN SELECT * FROM (SELECT ROW_NUMBER() OVER (PARTITION BY id1 ORDER BY value1) AS rownumber
    ↪ FROM t) dt WHERE rownumber <= 3;
```

```
+-----------------------------------+----------+-----------+---------------+--------------------------------
    ↪
| id                                | estRows  | task      | access object | operator info
    ↪
    ↪ |
+-----------------------------------+----------+-----------+---------------+--------------------------------
    ↪
| Projection_10                     | 2.40     | root      |               | Column#6
    ↪
    ↪ |
| └─Selection_11                    | 2.40     | root      |               | le(Column#6, 3)
    ↪
    ↪ |
|   └─Shuffle_23                    | 3.00     | root      |               | execution info:
    ↪ concurrency:3, data sources:[TableReader_21]
    ↪                                                                    |
|     └─Window_12                   | 3.00     | root      |               | row_number()->
    ↪ Column#6 over(partition by test.t.id1 order by test.t.value1 rows between current row and
    ↪  current row) |
|       └─Sort_22                   | 3.00     | root      |               | test.t.id1, test.t
    ↪ .value1
    ↪
    ↪ |
|         └─TableReader_21          | 3.00     | root      |               | data:TopN_19
    ↪
    ↪ |
|           └─TopN_19               | 3.00     | cop[tikv] |               | partition by test.
    ↪ t.id1 order by test.t.value1, offset:0, count:3
    ↪                                                       |
|             └─TableFullScan_18    | 10000.00 | cop[tikv] | table:t       | keep order:false,
    ↪ stats:pseudo
    ↪                                                                                  |
+-----------------------------------+----------+-----------+---------------+--------------------------------
    ↪
```

在该查询中，优化器从窗口函数中推导出来了 TopN 并将它下推给了 TiKV。需要注意的是，这个 TopN 其实是 partition 的 TopN, 也就是说对于每个相同 id1 值组成的一组数据上都会进行一次 TopN。

示例 5：PARTITION BY 列不是主键的前缀

```
CREATE TABLE t(id1 int, id2 int, value1 int, value2 int, primary key(id1,id2) clustered);
SET tidb_opt_derive_topn=on;
EXPLAIN SELECT * FROM (SELECT ROW_NUMBER() OVER (PARTITION BY value1) AS rownumber FROM t) dt
    ↪ WHERE rownumber <= 3;
```

```
+---------------------------------+----------+-----------+--------------+--------------------------------
    ↪
| id                              | estRows  | task      | access object | operator info
    ↪                                                                                  |
+---------------------------------+----------+-----------+--------------+--------------------------------
    ↪
| Projection_9                    | 8000.00  | root      |               | Column#6
    ↪                                                                                                 |
| └─Selection_10                   | 8000.00  | root      |               | le(Column#6, 3)
    ↪                                                                                               |
|   └─Shuffle_15                   | 10000.00 | root      |               | execution info:
    ↪ concurrency:5, data sources:[TableReader_13]                                                   |
|     └─Window_11                  | 10000.00 | root      |               | row_number()->Column
    ↪ #6 over(partition by test.t.value1 rows between current row and current row) |
|       └─Sort_14                  | 10000.00 | root      |               | test.t.value1
    ↪                                                                                                |
|         └─TableReader_13         | 10000.00 | root      |               | data:
    ↪ TableFullScan_12
    ↪                                                                                       |
|           └─TableFullScan_12     | 10000.00 | cop[tikv] | table:t       | keep order:false,
    ↪ stats:pseudo                                                                             |
+---------------------------------+----------+-----------+--------------+--------------------------------
    ↪
```

在该查询中，因为 partition 的列不是主键的前缀，所以 SQL 没有被改写。

示例 6：PARTITION BY 列是主键的前缀，但主键不是聚簇索引

```
CREATE TABLE t(id1 int, id2 int, value1 int, value2 int, primary key(id1,id2) nonclustered);
SET tidb_opt_derive_topn=on;
EXPLAIN SELECT * FROM (SELECT ROW_NUMBER() OVER (PARTITION BY id1) AS rownumber FROM t use index
    ↪ ()) dt WHERE rownumber <= 3;
```

```
+---------------------------------+----------+-----------+--------------+--------------------------------
    ↪
| id                              | estRows  | task      | access object | operator info
    ↪                                                                                  |
+---------------------------------+----------+-----------+--------------+--------------------------------
    ↪
```

```
| Projection_9                    | 8000.00  | root      |               | Column#7
    ↪                                                                                  |
| └─Selection_10                  | 8000.00  | root      |               | le(Column#7, 3)
    ↪                                                                                  |
|   └─Shuffle_15                  | 10000.00 | root      |               | execution info:
    ↪ concurrency:5, data sources:[TableReader_13]                                     |
|     └─Window_11                 | 10000.00 | root      |               | row_number()->Column
    ↪ #7 over(partition by test.t.id1 rows between current row and current row) |
|       └─Sort_14                 | 10000.00 | root      |               | test.t.id1
    ↪                                                                                  |
|         └─TableReader_13        | 10000.00 | root      |               | data:
    ↪ TableFullScan_12
    ↪ |
|           └─TableFullScan_12    | 10000.00 | cop[tikv] | table:t       | keep order:false,
    ↪ stats:pseudo                                                                     |
+---------------------------------+----------+-----------+---------------+-------------------------------------
    ↪
```

在该查询中，即使 PARTITION 的列是主键的前缀，但是因为主键不是聚簇索引，所以 SQL 没被改写。

11.3.3.3 物理优化

11.3.3.3.1 物理优化

物理优化是基于代价的优化，为上一阶段产生的逻辑执行计划制定物理执行计划。这一阶段中，优化器会为逻辑执行计划中的每个算子选择具体的物理实现。逻辑算子的不同物理实现有着不同的时间复杂度、资源消耗和物理属性等。在这个过程中，优化器会根据数据的统计信息来确定不同物理实现的代价，并选择整体代价最小的物理执行计划。

理解 TiDB 执行计划文档已对每个物理算子进行了一些介绍。在本章我们会重点介绍以下方面：

- 在索引的选择中会介绍 TiDB 在一张表有多个索引时，如何选择最优的索引进行表的访问。
- 在统计信息简介中会介绍 TiDB 收集了哪些统计信息来获得表的数据分布情况
- 在错误索引的解决方案中会介绍当发现 TiDB 索引选错时，你应该使用那些手段来让它使用正确的索引
- 在Distinct 优化中会介绍在物理优化中会做的一个有关 DISTINCT 关键字的优化，在这一小节中会介绍它的优缺点以及如何使用它。
- 在代价模型中会介绍在物理优化时，TiDB 怎么通过代价模型来选择一个最优的执行计划。

11.3.3.3.2 索引的选择

从存储层读取数据是 SQL 计算过程中最为耗时的部分之一，TiDB 目前支持从不同的存储和不同的索引中读取数据，索引选择得是否合理将很大程度上决定一个查询的运行速度。

本章节将介绍 TiDB 如何选择索引去读入数据，以及相关的一些控制索引选择的方式。

读表

在介绍索引的选择之前，首先要了解 TiDB 有哪些读表的方式，这些方式的触发条件是什么，不同方式有什么区别，各有什么优劣。

读表算子

读表算子	触发条件	适用场景	说明
PointGet/BatchPoint 算子	读表的范围是一个或多个单点范围	任何场景	如果能被触发，通常被认为是最快的算子，因为其直接调用 kvget 的接口进行计算，不走 coprocessor

读表算子	触发条件	适用场景	说明
TableReader	无	任何场景	该 TableReader 算子用于 TiKV。从 TiKV 端直接扫描表数据，一般被认为是效率最低的算子，除非在 _tidb_rowid ↪ 这一列上存在范围查询，或者无其他可以选择的读表算子时，才会选择这个算子

读表算子	触发条件	适用场景	说明
TableReader	表在 TiFlash 节点上存在副本	需要读取的列比较少，但是需要计算的行很多	该 TableReader 算子用于 TiFlash。TiFlash 是列式存储，如果需要对少量的列和大量的行进行计算，一般会选择这个算子
IndexReader	表有一个或多个索引，且计算所需的列被包含在索引里	存在较小的索引上的范围查询，或者对索引列有顺序需求的时候	当存在多个索引的时候，会根据估算代价选择合理的索引
IndexLookupReader	表有一个或多个索引，且计算所需的列不完全被包含在索引里	同 IndexReader	因为计算列不完全被包含在索引里，所以读完索引后需要回表，这里会比 IndexReader 多一些开销

读表算子	触发条件	适用场景	说明
IndexMerge	表有多个索引或多值索引	使用多值索引或同时使用多个索引的时候	可以通过optimizer hints 指定使用该算子，或让优化器根据代价估算自动选择该算子，参见用 EXPLAIN 查看索引合并的 SQL 执行计划

> **注意：**
>
> TableReader 是基于 `_tidb_rowid` 的索引，TiFlash 是列存索引，所以索引的选择即是读表算子的选择。

索引的选择

TiDB 基于规则或基于代价来选择索引。基于的规则包括前置规则和 Skyline-Pruning。在选择索引时，TiDB 会先尝试前置规则。如果存在索引满足某一条前置规则，则直接选择该索引。否则，TiDB 会采用 Skyline-Pruning 来排除不合适的索引，然后基于每个读表算子的代价估算，选择代价最小的索引。

基于规则选择

前置规则

TiDB 采用如下的启发式前置规则来选择索引：

- 规则 1：如果存在索引满足“唯一性索引全匹配 + 不需要回表（即该索引生成的计划是 IndexReader）”时，直接选择该索引。
- 规则 2：如果存在索引满足“唯一性索引全匹配 + 需要回表（即该索引生成的计划是 IndexLookupReader）”时，选择满足该条件且回表行数最小的索引作为候选索引。
- 规则 3：如果存在索引满足“普通索引不需要回表 + 读取行数小于一定阈值”时，选择满足该条件且读取行数最小的索引作为候选索引。

- 规则 4：如果规则 2 和 3 之中仅选出一条候选索引，则选择该候选索引。如果规则 2 和 3 均选出候选索引，则选择读取行数（读索引行数 + 回表行数）较小的索引。

上述规则中的“索引全匹配”指每个索引列上均存在等值条件。在执行 `EXPLAIN FORMAT = 'verbose' ...` 语句时，如果前置规则匹配了某一索引，TiDB 会输出一条 NOTE 级别的 warning 提示该索引匹配了前置规则。

在以下示例中，因为索引 `idx_b` 满足规则 2 中“唯一性索引全匹配 + 需要回表”的条件，TiDB 选择索引 `idx_b` 作为访问路径，`SHOW WARNING` 返回了索引 `idx_b` 命中前置规则的提示。

```
mysql> CREATE TABLE t(a INT PRIMARY KEY, b INT, c INT, UNIQUE INDEX idx_b(b));
Query OK, 0 rows affected (0.01 sec)

mysql> EXPLAIN FORMAT = 'verbose' SELECT b, c FROM t WHERE b = 3 OR b = 6;
+--
    ↪ ----------------+---------+---------+------+------------------------+-----------------------------+
    ↪
| id                | estRows | estCost | task | access object          | operator info
    ↪                 |
+--
    ↪ ----------------+---------+---------+------+------------------------+-----------------------------+
    ↪
| Batch_Point_Get_5 | 2.00    | 8.80    | root | table:t, index:idx_b(b) | keep order:false, desc
    ↪ :false |
+--
    ↪ ----------------+---------+---------+------+------------------------+-----------------------------+
    ↪
1 row in set, 1 warning (0.00 sec)

mysql> SHOW WARNINGS;
+--
    ↪ -----+------+-----------------------------------------------------------------------------------------+
    ↪
| Level | Code | Message
    ↪                                                                                                  |
+--
    ↪ -----+------+-----------------------------------------------------------------------------------------+
    ↪
| Note  | 1105 | unique index idx_b of t is selected since the path only has point ranges with
    ↪ double scan |
+--
    ↪ -----+------+-----------------------------------------------------------------------------------------+
    ↪
1 row in set (0.00 sec)
```

Skyline-Pruning

Skyline-Pruning 是一个针对索引的启发式过滤规则，能降低错误估算导致选错索引的概率。Skyline-Pruning 从以

下三个维度衡量一个索引的优劣：

- 索引的列涵盖了多少访问条件。“访问条件”指的是可以转化为某列范围的 where 条件，如果某个索引的列集合涵盖的访问条件越多，那么它在这个维度上更优。
- 选择该索引读表时，是否需要回表（即该索引生成的计划是 IndexReader 还是 IndexLookupReader）。不用回表的索引在这个维度上优于需要回表的索引。如果均需要回表，则比较索引的列涵盖了多少过滤条件。过滤条件指的是可以根据索引判断的 where 条件。如果某个索引的列集合涵盖的访问条件越多，则回表数量越少，那么它在这个维度上越优。
- 选择该索引是否能满足一定的顺序。因为索引的读取可以保证某些列集合的顺序，所以满足查询要求顺序的索引在这个维度上优于不满足的索引。

对于以上三种维度，如果索引 idx_a 在这三个维度上都不比 idx_b 差，且有一个维度比 idx_b 好，那么 TiDB 会优先选择 idx_a。在执行 EXPLAIN FORMAT = 'verbose' ... 语句时，如果 Skyline-Pruning 排除了某些索引，TiDB 会输出一条 NOTE 级别的 warning 提示哪些索引在 Skyline-Pruning 排除之后保留下来。

在以下示例中，索引 idx_b 和 idx_e 均劣于 idx_b_c，因而被 Skyline-Pruning 排除，SHOW WARNING 的返回结果显示了经过 Skyline-Pruning 后剩余的索引。

```
mysql> CREATE TABLE t(a INT PRIMARY KEY, b INT, c INT, d INT, e INT, INDEX idx_b(b), INDEX
    ↪ idx_b_c(b, c), INDEX idx_e(e));
Query OK, 0 rows affected (0.01 sec)

mysql> EXPLAIN FORMAT = 'verbose' SELECT * FROM t WHERE b = 2 AND c > 4;
+--
    ↪ ----------------------------+---------+---------+-----------+-----------------------------+-----------
    ↪
| id                            | estRows | estCost | task      | access object               |
    ↪ operator info                                       |
+--
    ↪ ----------------------------+---------+---------+-----------+-----------------------------+-----------
    ↪
| IndexLookUp_10                | 33.33   | 738.29  | root      |                             |
    ↪                                                     |
| ├─IndexRangeScan_8(Build)     | 33.33   | 2370.00 | cop[tikv] | table:t, index:idx_b_c(b, c) |
    ↪  range:(2 4,2 +inf], keep order:false, stats:pseudo |
| └─TableRowIDScan_9(Probe)     | 33.33   | 2370.00 | cop[tikv] | table:t                      |
    ↪  keep order:false, stats:pseudo                     |
+--
    ↪ ----------------------------+---------+---------+-----------+-----------------------------+-----------
    ↪
3 rows in set, 1 warning (0.00 sec)

mysql> SHOW WARNINGS;
+--
    ↪ -----+------+-----------------------------------------------------------------------------------+
    ↪
```

```
| Level | Code | Message
    ↪                                                
```

使用多值索引

> 警告：
>
> 当前该功能为实验特性，不建议在生产环境中使用。

多值索引和普通索引有所不同，TiDB 目前只会使用IndexMerge 来访问多值索引。因此要想使用多值索引进行数据访问，请确保tidb_enable_index_merge 被设置为 ON。

多值索引的使用限制请参考CREATE INDEX。

目前 TiDB 支持将 json_member_of、json_contains 和 json_overlaps 条件自动转换成 IndexMerge 来访问多值索引。既可以依赖优化器根据代价自动选择，也可通过use_index_merge optimizer hint 或use_index 指定选择多值索引，见下面例子：

```
mysql> CREATE TABLE t1 (j JSON, INDEX idx((CAST(j->'$.path' AS SIGNED ARRAY)))); -- 使用 '$.path'
    ↪  作为路径创建多值索引
Query OK, 0 rows affected (0.04 sec)

mysql> EXPLAIN SELECT /*+ use_index_merge(t1, idx) */ * FROM t1 WHERE (1 MEMBER OF (j->'$.path'))
    ↪ ;
+--
    ↪ -------------------------------+---------+-----------+-----------------------------------------------------
    ↪
| id                              | estRows | task      | access object
    ↪                                                                 | operator info
    ↪                                                             |
+--
    ↪ -------------------------------+---------+-----------+-----------------------------------------------------
    ↪
| Selection_5                     | 8000.00 | root      |
    ↪                                                                                 |
    ↪ json_memberof(cast(1, json BINARY), json_extract(test.t1.j, "$.path")) |
| └─IndexMerge_8                  | 10.00   | root      |
    ↪                                                                                 | type: union
    ↪                                                                 |
|   ├─IndexRangeScan_6(Build)     | 10.00   | cop[tikv] | table:t1, index:idx(cast(json_extract
    ↪ (`j`, _utf8'$.path') as signed array)) | range:[1,1], keep order:false, stats:pseudo
    ↪                             |
|   └─TableRowIDScan_7(Probe)     | 10.00   | cop[tikv] | table:t1
    ↪                                                                     | keep order:false,
    ↪ stats:pseudo                                          |
+--
    ↪ -------------------------------+---------+-----------+-----------------------------------------------------
    ↪
```

```
4 rows in set, 1 warning (0.00 sec)

mysql> EXPLAIN SELECT /*+ use_index_merge(t1, idx) */ * FROM t1 WHERE JSON_CONTAINS((j->'$.path')
    ↪ , '[1, 2, 3]');
+--
    ↪ ----------------------------+---------+-----------+-----------------------------------------------------------
    ↪
| id                            | estRows | task      | access object
    ↪                                                                | operator info
    ↪                                |
+--
    ↪ ----------------------------+---------+-----------+-----------------------------------------------------------
    ↪
| IndexMerge_9                  | 10.00   | root      |
    ↪                                                                                  | type:
    ↪ intersection                           |
| ├─IndexRangeScan_5(Build)     | 10.00   | cop[tikv] | table:t1, index:idx(cast(json_extract(`j
    ↪ `, _utf8'$.path') as signed array)) | range:[1,1], keep order:false, stats:pseudo |
| ├─IndexRangeScan_6(Build)     | 10.00   | cop[tikv] | table:t1, index:idx(cast(json_extract(`j
    ↪ `, _utf8'$.path') as signed array)) | range:[2,2], keep order:false, stats:pseudo |
| ├─IndexRangeScan_7(Build)     | 10.00   | cop[tikv] | table:t1, index:idx(cast(json_extract(`j
    ↪ `, _utf8'$.path') as signed array)) | range:[3,3], keep order:false, stats:pseudo |
| └─TableRowIDScan_8(Probe)     | 10.00   | cop[tikv] | table:t1
    ↪                                                                          | keep order:false,
    ↪ stats:pseudo              |
+--
    ↪ ----------------------------+---------+-----------+-----------------------------------------------------------
    ↪
5 rows in set (0.00 sec)

mysql> EXPLAIN SELECT /*+ use_index_merge(t1, idx) */ * FROM t1 WHERE JSON_OVERLAPS((j->'$.path')
    ↪ , '[1, 2, 3]');
+--
    ↪ ------------------------------+---------+-----------+-----------------------------------------------------------
    ↪
| id                              | estRows | task      | access object
    ↪                                                                | operator info
    ↪                                                                     |
+--
    ↪ ------------------------------+---------+-----------+-----------------------------------------------------------
    ↪
| Selection_5                     | 8000.00 | root      |
    ↪                                                                                  |
    ↪ json_overlaps(json_extract(test.t1.j, "$.path"), cast("[1, 2, 3]", json BINARY)) |
| └─IndexMerge_10                  | 10.00   | root      |
```

```
    ↪                                                                                  | type: union
    ↪                                                                                  |
|   ├─IndexRangeScan_6(Build)     | 10.00   | cop[tikv] | table:t1, index:idx(cast(json_extract
    ↪ (`j`, _utf8'$.path') as signed array)) | range:[1,1], keep order:false, stats:pseudo
    ↪                                        |
|   ├─IndexRangeScan_7(Build)     | 10.00   | cop[tikv] | table:t1, index:idx(cast(json_extract
    ↪ (`j`, _utf8'$.path') as signed array)) | range:[2,2], keep order:false, stats:pseudo
    ↪                                        |
|   ├─IndexRangeScan_8(Build)     | 10.00   | cop[tikv] | table:t1, index:idx(cast(json_extract
    ↪ (`j`, _utf8'$.path') as signed array)) | range:[3,3], keep order:false, stats:pseudo
    ↪                                        |
|   └─TableRowIDScan_9(Probe)     | 10.00   | cop[tikv] | table:t1
    ↪                                                                    | keep order:false,
    ↪ stats:pseudo                                                    |
+--
    ↪ -------------------------------+---------+-----------+-------------------------------------------------------
    ↪
6 rows in set, 1 warning (0.00 sec)
```

复合的多值索引，也一样可以使用 IndexMerge 进行访问：

```
mysql> CREATE TABLE t2 (a INT, j JSON, b INT, INDEX idx(a, (CAST(j->'$.path' AS SIGNED ARRAY)), b
    ↪ ));
Query OK, 0 rows affected (0.04 sec)

mysql> EXPLAIN SELECT /*+ use_index_merge(t2, idx) */ * FROM t2 WHERE a=1 AND (1 MEMBER OF (j->'$
    ↪ .path')) AND b=2;
+--
    ↪ -------------------------------+---------+-----------+-------------------------------------------------------
    ↪
| id                              | estRows | task      | access object
    ↪                                                                    | operator info
    ↪                                                          |
+--
    ↪ -------------------------------+---------+-----------+-------------------------------------------------------
    ↪
| Selection_5                     | 0.01    | root      |
    ↪                                                                                  |
    ↪ json_memberof(cast(1, json BINARY), json_extract(test.t2.j, "$.path")) |
| └─IndexMerge_8                  | 0.00    | root      |
    ↪                                                                                  | type:
    ↪  union                                                                  |
|   ├─IndexRangeScan_6(Build)     | 0.00    | cop[tikv] | table:t2, index:idx(a, cast(
    ↪ json_extract(`j`, _utf8'$.path') as signed array), b) | range:[1 1 2,1 1 2], keep order:
    ↪ false, stats:pseudo                    |
|   └─TableRowIDScan_7(Probe)     | 0.00    | cop[tikv] | table:t2
```

```
    ↪                                                                                  | keep order:
    ↪ false, stats:pseudo                                                  |
+--
    ↪ -------------------------------+---------+-----------+-------------------------------------------------------
    ↪
4 rows in set, 1 warning (0.00 sec)

mysql> EXPLAIN SELECT /*+ use_index_merge(t2, idx) */ * FROM t2 WHERE a=1 AND JSON_CONTAINS((j->'
    ↪ $.path'), '[1, 2, 3]');
+--
    ↪ -------------------------------+---------+-----------+-------------------------------------------------------
    ↪
| id                             | estRows | task      | access object
    ↪                                                                          | operator info
    ↪                                      |
+--
    ↪ -------------------------------+---------+-----------+-------------------------------------------------------
    ↪
| IndexMerge_9                   | 0.10    | root      |
    ↪                                                                                          | type:
    ↪  intersection                                  |
| ├─IndexRangeScan_5(Build)      | 0.10    | cop[tikv] | table:t2, index:idx(a, cast(json_extract
    ↪ (`j`, _utf8'$.path') as signed array), b) | range:[1 1,1 1], keep order:false, stats:
    ↪ pseudo |
| ├─IndexRangeScan_6(Build)      | 0.10    | cop[tikv] | table:t2, index:idx(a, cast(json_extract
    ↪ (`j`, _utf8'$.path') as signed array), b) | range:[1 2,1 2], keep order:false, stats:
    ↪ pseudo |
| ├─IndexRangeScan_7(Build)      | 0.10    | cop[tikv] | table:t2, index:idx(a, cast(json_extract
    ↪ (`j`, _utf8'$.path') as signed array), b) | range:[1 3,1 3], keep order:false, stats:
    ↪ pseudo |
| └─TableRowIDScan_8(Probe)      | 0.10    | cop[tikv] | table:t2
    ↪                                                                                  | keep order:
    ↪ false, stats:pseudo                   |
+--
    ↪ -------------------------------+---------+-----------+-------------------------------------------------------
    ↪
5 rows in set (0.00 sec)

mysql> EXPLAIN SELECT /*+ use_index_merge(t2, idx) */ * FROM t2 WHERE a=1 AND JSON_OVERLAPS((j->'
    ↪ $.path'), '[1, 2, 3]');
+--
    ↪ -------------------------------+---------+-----------+-------------------------------------------------------
    ↪
| id                              | estRows | task      | access object
    ↪                                                                          | operator info
```

```
    ↪                                                                  |
+--
    ↪ -------------------------------+---------+-----------+-------------------------------------------------------
    ↪
| Selection_5                     | 8.00    | root      |
    ↪                                                                                  |
    ↪ json_overlaps(json_extract(test.t2.j, "$.path"), cast("[1, 2, 3]", json BINARY)) |
| └─IndexMerge_10                 | 0.10    | root      |
    ↪                                                                                  | type:
    ↪  union                                                                  |
|   ├─IndexRangeScan_6(Build)     | 0.10    | cop[tikv] | table:t2, index:idx(a, cast(
    ↪ json_extract(`j`, _utf8'$.path') as signed array), b) | range:[1 1,1 1], keep order:false
    ↪ , stats:pseudo                                  |
|   ├─IndexRangeScan_7(Build)     | 0.10    | cop[tikv] | table:t2, index:idx(a, cast(
    ↪ json_extract(`j`, _utf8'$.path') as signed array), b) | range:[1 2,1 2], keep order:false
    ↪ , stats:pseudo                                  |
|   ├─IndexRangeScan_8(Build)     | 0.10    | cop[tikv] | table:t2, index:idx(a, cast(
    ↪ json_extract(`j`, _utf8'$.path') as signed array), b) | range:[1 3,1 3], keep order:false
    ↪ , stats:pseudo                                  |
|   └─TableRowIDScan_9(Probe)     | 0.10    | cop[tikv] | table:t2
    ↪                                                                      | keep order:
    ↪ false, stats:pseudo                                                |
+--
    ↪ -------------------------------+---------+-----------+-------------------------------------------------------
    ↪
6 rows in set, 1 warning (0.00 sec)
```

对于由多个 member of 组成的 OR 条件，也可以使用 IndexMerge 进行访问：

```
mysql> CREATE TABLE t3 (a INT, j JSON, INDEX idx(a, (CAST(j AS SIGNED ARRAY))));
Query OK, 0 rows affected (0.04 sec)

mysql> EXPLAIN SELECT /*+ use_index_merge(t3, idx) */ * FROM t3 WHERE ((a=1 AND (1 member of (j))
    ↪ )) OR ((a=2 AND (2 member of (j))));
+--
    ↪ -------------------------------+---------+-----------+-------------------------------------------------------
    ↪
| id                              | estRows | task      | access object
    ↪                                       | operator info
    ↪
    ↪ |
+--
    ↪ -------------------------------+---------+-----------+-------------------------------------------------------
    ↪
| Selection_5                     | 0.08    | root      |
    ↪                                                   | or(and(eq(test.t3.a, 1),
```

```
    ↪ json_memberof(cast(1, json BINARY), test.t3.j)), and(eq(test.t3.a, 2), json_memberof(cast
    ↪ (2, json BINARY), test.t3.j))) |
| └─IndexMerge_9                  | 0.10    | root      |
    ↪                                                    | type: union
    ↪
    ↪ |
|   ├─IndexRangeScan_6(Build)     | 0.10    | cop[tikv] | table:t3, index:idx(a, cast(`j` as
    ↪ signed array)) | range:[1 1,1 1], keep order:false, stats:pseudo
    ↪
    ↪ |
|   ├─IndexRangeScan_7(Build)     | 0.10    | cop[tikv] | table:t3, index:idx(a, cast(`j` as
    ↪ signed array)) | range:[2 2,2 2], keep order:false, stats:pseudo
    ↪
    ↪ |
|   └─TableRowIDScan_8(Probe)     | 0.10    | cop[tikv] | table:t3
    ↪                                                       | keep order:false, stats:pseudo
    ↪
    ↪ |
+--
    ↪ ------------------------------+---------+-----------+-----------------------------------------------
    ↪
```

以下场景暂时无法使用 IndexMerge 来访问多值索引：

```
mysql> CREATE TABLE t4 (j JSON, INDEX idx((CAST(j AS SIGNED ARRAY))));
Query OK, 0 rows affected (0.04 sec)

-- 如果查询包含多个 json_contains 通过 OR 组成的条件，则无法通过 IndexMerge 访问索引
mysql> EXPLAIN SELECT /*+ use_index_merge(t3, idx) */ * FROM t3 WHERE (json_contains(j, '[1, 2]')
    ↪ ) OR (json_contains(j, '[3, 4]'));
+--
    ↪ ----------------------+----------+-----------+---------------+-------------------------------------
    ↪
| id                      | estRows  | task      | access object | operator info
    ↪
    ↪ |
+--
    ↪ ----------------------+----------+-----------+---------------+-------------------------------------
    ↪
| TableReader_7           | 9600.00  | root      |               | data:Selection_6
    ↪
    ↪ |
| └─Selection_6           | 9600.00  | cop[tikv] |               | or(json_contains(test.t3.j,
    ↪ cast("[1, 2]", json BINARY)), json_contains(test.t3.j, cast("[3, 4]", json BINARY))) |
|   └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t3      | keep order:false, stats:
    ↪ pseudo
```

```
    ↪ |
+--
    ↪ ----------------------+----------+-----------+--------------+--------------------------------
    ↪
3 rows in set, 1 warning (0.00 sec)

mysql> SHOW WARNINGS;
+---------+------+----------------------------+
| Level   | Code | Message                    |
+---------+------+----------------------------+
| Warning | 1105 | IndexMerge is inapplicable |
+---------+------+----------------------------+
1 row in set (0.00 sec)

mysql> EXPLAIN SELECT /*+ use_index_merge(t3, idx) */ * FROM t3 WHERE (json_contains(j, '[1, 2]')
    ↪ ) OR (json_contains(j, '[3, 4]'));
+--
    ↪ ----------------------+----------+-----------+--------------+--------------------------------
    ↪
| id                    | estRows  | task      | access object | operator info
    ↪
    ↪ |
+--
    ↪ ----------------------+----------+-----------+--------------+--------------------------------
    ↪
| TableReader_7           | 9600.00  | root      |               | data:Selection_6
    ↪
    ↪ |
| └─Selection_6           | 9600.00  | cop[tikv] |               | or(json_contains(test.t3.j,
    ↪ cast("[1, 2]", json BINARY)), json_contains(test.t3.j, cast("[3, 4]", json BINARY))) |
|   └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t3      | keep order:false, stats:
    ↪ pseudo
    ↪ |
+--
    ↪ ----------------------+----------+-----------+--------------+--------------------------------
    ↪
3 rows in set, 1 warning (0.01 sec)

-- 如果查询中包含较为复杂的多层 OR / AND 嵌套形成的表达式，则无法通过 IndexMerge 访问索引
mysql> EXPLAIN SELECT /*+ use_index_merge(t3, idx) */ * FROM t3 WHERE ((1 member of (j)) AND (2
    ↪ member of (j))) OR ((3 member of (j)) AND (4 member of (j)));
+--
    ↪ ----------------------+----------+-----------+--------------+--------------------------------
    ↪
| id                    | estRows  | task      | access object | operator info
```

```
    ↪
    ↪ |
+--
    ↪ ----------------------+----------+-----------+---------------+--------------------------------
    ↪
| Selection_5             | 8000.00  | root      |               | or(and(json_memberof(cast(1,
    ↪ json BINARY), test.t3.j), json_memberof(cast(2, json BINARY), test.t3.j)), and(
    ↪ json_memberof(cast(3, json BINARY), test.t3.j), json_memberof(cast(4, json BINARY), test.
    ↪ t3.j))) |
| └─TableReader_7         | 10000.00 | root      |               | data:TableFullScan_6
    ↪
    ↪ |
|   └─TableFullScan_6     | 10000.00 | cop[tikv] | table:t3      | keep order:false, stats:
    ↪ pseudo
    ↪
    ↪ |
+--
    ↪ ----------------------+----------+-----------+---------------+--------------------------------
    ↪
3 rows in set, 2 warnings (0.00 sec)
```

受限于多值索引的特性，当多值索引无法生效时，使用use_index 可能会返回 Can't find a proper physical
↪ plan for this query 的错误，而使用use_index_merge 不会，因此建议使用 use_index_merge：

```
mysql> EXPLAIN SELECT /*+ use_index(t3, idx) */ * FROM t3 WHERE ((1 member of (j)) AND (2 member
    ↪ of (j))) OR ((3 member of (j)) AND (4 member of (j)));
ERROR 1815 (HY000): Internal : Cant find a proper physical plan for this query

mysql> EXPLAIN SELECT /*+ use_index_merge(t3, idx) */ * FROM t3 WHERE ((1 member of (j)) AND (2
    ↪ member of (j))) OR ((3 member of (j)) AND (4 member of (j)));
+--
    ↪ ----------------------+----------+-----------+---------------+--------------------------------
    ↪
| id                      | estRows  | task      | access object | operator info
    ↪
    ↪ |
+--
    ↪ ----------------------+----------+-----------+---------------+--------------------------------
    ↪
| Selection_5             | 8000.00  | root      |               | or(and(json_memberof(cast(1,
    ↪ json BINARY), test.t3.j), json_memberof(cast(2, json BINARY), test.t3.j)), and(
    ↪ json_memberof(cast(3, json BINARY), test.t3.j), json_memberof(cast(4, json BINARY), test.
    ↪ t3.j))) |
| └─TableReader_7         | 10000.00 | root      |               | data:TableFullScan_6
    ↪
    ↪ |
```

```
|   └─TableFullScan_6     | 10000.00 | cop[tikv] | table:t3      | keep order:false, stats:
    ↪ pseudo
    ↪
    ↪ |
+--
    ↪ ----------------------+----------+-----------+---------------+-------------------------------
    ↪
3 rows in set, 2 warnings (0.00 sec)
```

11.3.3.3.3 统计信息简介

TiDB 使用统计信息来决定索引的选择。

统计信息版本

变量 tidb_analyze_version 用于控制所收集到的统计信息。目前 TiDB 中支持两种统计信息：tidb_analyze_version ↪ = 1 以及 tidb_analyze_version = 2。在 v5.3.0 及之后的版本中，该变量的默认值为 2。如果从 v5.3.0 之前版本的集群升级至 v5.3.0 及之后的版本，tidb_analyze_version 的默认值不发生变化。

Version 2 的统计信息避免了 Version 1 中因为哈希冲突导致的在较大的数据量中可能产生的较大误差，并保持了大多数场景中的估算精度。

两种版本中，TiDB 维护的统计信息如下：

信息	Version 1	Version 2
表的总行数	√	√
列的 Count-Min Sketch	√	×
索引的 Count-Min Sketch	√	×
列的 Top-N	√	√（改善了维护方式和精度）
索引的 Top-N	√（维护精度不足，会产生较大误差）	√（改善了维护方式和精度）
列的直方图	√	√（直方图中不包含 Top-N 中出现的值）
索引的直方图	√	√（直方图的桶中记录了各自的不同值的个数，且直方图不包
列的 NULL 值个数	√	√
索引的 NULL 值个数	√	√
列的平均长度	√	√
索引的平均长度	√	√

当 tidb_analyze_version = 2 时，如果执行 ANALYZE 语句后发生 OOM，需要设置全局变量 tidb_analyze_version ↪ = 1，回退到 Version 1，然后根据情况进行以下操作：

- 如果 ANALYZE 语句是手动执行的，你需要手动 ANALYZE 每张需要的表：

```
SELECT DISTINCT(CONCAT('ANALYZE TABLE ', table_schema, '.', table_name, ';')) FROM
    ↪ information_schema.tables, mysql.stats_histograms WHERE stats_ver = 2 AND table_id =
    ↪  tidb_table_id;
```

- 如果 ANALYZE 语句是开启了自动 ANALYZE 后 TiDB 自动执行的，使用以下 SQL 语句生成 DROP STATS 的语句并执行：

```
sql  SELECT DISTINCT(CONCAT('DROP STATS ', table_schema, '.', table_name, ';'))FROM information_schema
↪ .tables, mysql.stats_histograms WHERE stats_ver = 2 AND table_id = tidb_table_id;
```

- 如果上一条语句返回结果太长，不方便复制粘贴，可以将结果导出到临时文件后，再执行：

```
sql  SELECT DISTINCT... INTO outfile '/tmp/sql.txt'; mysql -h ${TiDB_IP} -u user -P ${TIDB_PORT}
↪ ... < '/tmp/sql.txt';
```

本文接下来将简单介绍其中出现的直方图和 Count-Min Sketch 以及 Top-N 这些数据结构，以及详细介绍统计信息的收集和维护。

直方图简介

直方图是一种对数据分布情况进行描述的工具，它会按照数据的值大小进行分桶，并用一些简单的数据来描述每个桶，比如落在桶里的值的个数。在 TiDB 中，会对每个表具体的列构建一个等深直方图，区间查询的估算便是借助该直方图来进行。

等深直方图，就是让落入每个桶里的值数量尽量相等。举个例子，比方说对于给定的集合 {1.6, 1.9, 1.9, 2.0, 2.4, 2.6, 2.7, 2.7, 2.8, 2.9, 3.4, 3.5}，并且生成 4 个桶，那么最终的等深直方图就会如下图所示，包含四个桶 [1.6, 1.9]，[2.0, 2.6]，[2.7, 2.8]，[2.9, 3.5]，其桶深均为 3。

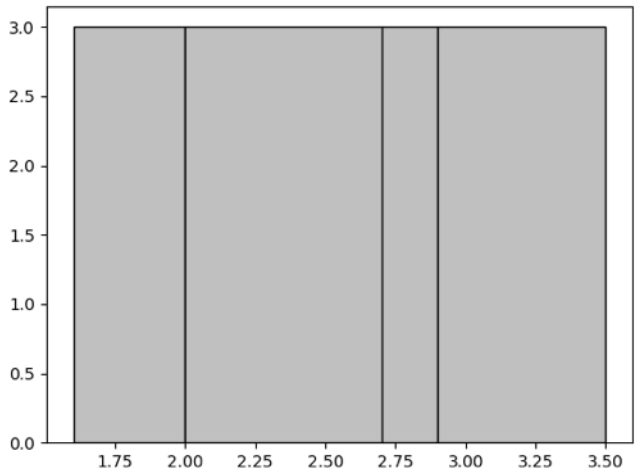


图 167: 等深直方图示例

在手动收集统计信息一节中有控制直方图桶数量上限的参数。当桶数量越多，直方图的估算精度就越高，不过也会同时增大统计信息的内存使用，可以视具体情况来做调整。

Count-Min Sketch

Count-Min Sketch 是一种哈希结构，当查询中出现诸如 `a = 1` 或者 IN 查询（如 `a in (1, 2, 3)`）这样的等值查询时，TiDB 便会使用这个数据结构来进行估算。

由于 Count-Min Sketch 是一个哈希结构，就有出现哈希碰撞的可能。当在 EXPLAIN 语句中发现等值查询的估算偏离实际值较大时，就可以认为是一个比较大的值和一个比较小的值被哈希到了一起。这时有以下两种手段来避免这个情况：

- 修改手动收集统计信息中提到的 WITH NUM TOPN 参数。TiDB 会将出现频率前 x 大的数据单独储存，之后的数据再储存到 Count-Min Sketch 中。因此可以调大这个值来避免一个比较大的值和一个比较小的值被哈希到一起。在 TiDB 中，这个参数的默认值是 20，最大可以设置为 1024。
- 修改统计信息的收集 -手动收集中提到的 WITH NUM CMSKETCH DEPTH 和 WITH NUM CMSKETCH WIDTH 两个参数，这两个参数会影响哈希的桶数和碰撞概率，可是适当调大来减少冲突概率，同时它会影响统计信息的内存使用，可以视具体情况来调整。在 TiDB 中，DEPTH 的默认值是 5，WIDTH 的默认值是 2048。

Top-N values

Top-N 即是这个列或者这个索引中，出现次数前 n 的值。TiDB 会记录前 n 个值的具体的值以及出现次数。

统计信息的收集

手动收集

可以通过执行 ANALYZE 语句来收集统计信息。

> 注意：
>
> 在 TiDB 中执行 ANALYZE TABLE 语句比在 MySQL 或 InnoDB 中耗时更长。InnoDB 采样的只是少量页面，但 TiDB 会完全重构一系列统计信息。适用于 MySQL 的脚本会误以为执行 ANALYZE TABLE 耗时较短。
>
> 如需更快的分析速度，可将 tidb_enable_fast_analyze 设置为 1 来打开快速分析功能。该参数的默认值为 0。
>
> 快速分析功能开启后，TiDB 会随机采样约 10000 行的数据来构建统计信息。因此在数据分布不均匀或者数据量比较少的情况下，统计信息的准确度会比较差。可能导致执行计划不优，比如选错索引。如果可以接受普通 ANALYZE 语句的执行时间，则推荐关闭快速分析功能。
>
> tidb_enable_fast_analyze 为实验性功能，目前与 tidb_analyze_version=2 的统计信息不完全匹配。因此开启 tidb_enable_fast_analyze 时需要将 tidb_analyze_version 的值设置为 1。

全量收集

可以通过以下几种语法进行全量收集。

收集 TableNameList 中所有表的统计信息：

```
ANALYZE TABLE TableNameList [WITH NUM BUCKETS|TOPN|CMSKETCH DEPTH|CMSKETCH WIDTH]|[WITH NUM
    ↪ SAMPLES|WITH FLOATNUM SAMPLERATE];
```

- WITH NUM BUCKETS 用于指定生成直方图的桶数量上限。
- WITH NUM TOPN 用于指定生成 TOPN 数目的上限。
- WITH NUM CMSKETCH DEPTH 用于指定 CM Sketch 的长。

- WITH NUM CMSKETCH WIDTH 用于指定 CM Sketch 的宽。
- WITH NUM SAMPLES 用于指定采样的数目。
- WITH FLOAT_NUM SAMPLERATE 用于指定采样率。

WITH NUM SAMPLES 与 WITH FLOAT_NUM SAMPLERATE 这两种设置对应了两种不同的收集采样的算法。

- WITH NUM SAMPLES 指定了采样集的大小，在 TiDB 中是以蓄水池采样的方式实现。当表较大时，不推荐使用这种方式收集统计信息。因为蓄水池采样中间结果集会产生一定的冗余结果，会对内存等资源造成额外的压力。
- WITH FLOAT_NUM SAMPLERATE 是在 v5.3.0 中引入的采样方式，指定的采样率的大小，是取值范围 (0, 1] 的参数。在 TiDB 中是以伯努利采样的方式实现，更适合对较大的表进行采样，在收集效率和资源使用上更有优势。

在 v5.3.0 之前 TiDB 采用蓄水池采样的方式收集统计信息。自 v5.3.0 版本起，TiDB Version 2 的统计信息默认会选取伯努利采样的方式收集统计信息。若要重新使用蓄水池采样的方式采样，可以使用 WITH NUM SAMPLES 语句。

> 注意：
>
> 目前采样率基于自适应算法进行计算。当你通过SHOW STATS_META 可以观察到一个表的行数时，可通过这个行数去计算采集 10 万行所对应的采样率。如果你观察不到这个值，可通过TABLE_STORAGE_STATS 表的 TABLE_KEYS 列作为另一个参考来计算采样率。
>
> 通常情况下，STATS_META 相对 TABLE_KEYS 更可信，但是通过TiDB Lightning 等方式导入数据结束后，STATS_META 结果是 0。为了处理这个情况，你可以在 STATS_META 的结果远小于 TABLE_KEYS 的结果时，使用 TABLE_KEYS 计算采样率。

收集部分列的统计信息

执行 SQL 语句时，优化器在大多数情况下只会用到部分列（例如，WHERE、JOIN、ORDER BY、GROUP BY 子句中出现的列）的统计信息，这些被用到的列称为 PREDICATE COLUMNS。

如果一个表有很多列，收集所有列的统计信息会有较大的开销。为了降低开销，你可以只收集指定列或者 PREDICATE COLUMNS 的统计信息供优化器使用。

> 注意：
>
> 收集部分列的统计信息的功能仅适用于 tidb_analyze_version = 2 的情况。

- 如果要收集指定列的统计信息，请使用以下语法：

```
ANALYZE TABLE TableName COLUMNS ColumnNameList [WITH NUM BUCKETS|TOPN|CMSKETCH DEPTH|
    ↪ CMSKETCH WIDTH]|[WITH NUM SAMPLES|WITH FLOATNUM SAMPLERATE];
```

其中，ColumnNameList 表示指定列的名称列表。如果需要指定多列，请使用用逗号 , 分隔列名。例如，ANALYZE table t columns a, b。该语法除了收集指定表中指定列的统计信息，将同时收集该表中索引列的统计信息以及所有索引的统计信息。

> 注意：
> 该语法为全量收集。例如，在使用该语法收集了 a 列和 b 列的统计信息之后，如果还想要增加收集 c 列的统计信息，需要在语法中同时指定这三列 ANALYZE TABLE t columns a, b, c，而不是只指定新增的那一列 ANALYZE TABLE t COLUMNS c。

- 如果要收集 PREDICATE COLUMNS 的统计信息，请进行以下操作：

> 警告：
> 收集 PREDICATE COLUMNS 的统计信息目前为实验特性，不建议在生产环境中使用。

1. 设置系统变量tidb_enable_column_tracking 的值为 ON 开启 TiDB 对 PREDICATE COLUMNS 的收集。开启后，TiDB 将每隔 100 * stats-lease 时间将 PREDICATE COLUMNS 信息写入系统表 mysql.column_stats_usage。
2. 在业务的查询模式稳定以后，使用以下语法收集 PREDICATE COLUMNS 的统计信息。

```
ANALYZE TABLE TableName PREDICATE COLUMNS [WITH NUM BUCKETS|TOPN|CMSKETCH DEPTH|
    CMSKETCH WIDTH]|[WITH NUM SAMPLES|WITH FLOATNUM SAMPLERATE];
```

   该语法除了收集指定表中 PREDICATE COLUMNS 的统计信息，将同时收集该表中索引列的统计信息以及所有索引的统计信息。

> 注意：
> - 如果系统表 mysql.column_stats_usage 中没有关于该表的PREDICATE COLUMNS 记录，执行以上语句会收集该表中所有列的统计信息以及所有索引的统计信息。
> - 使用该语法收集统计信息后，当执行一种新的类型的 SQL 查询时，优化器可能会暂时使用旧的或者 pseudo 的列统计信息，然后在下一次收集统计信息的时候收集该列的统计信息。

- 如果要收集所有列的统计信息以及所有索引的统计信息，可以使用以下语法：

```
ANALYZE TABLE TableName ALL COLUMNS [WITH NUM BUCKETS|TOPN|CMSKETCH DEPTH|CMSKETCH WIDTH]|[
    WITH NUM SAMPLES|WITH FLOATNUM SAMPLERATE];
```

如果要持久化 ANALYZE 语句中列的配置（包括 COLUMNS ColumnNameList、PREDICATE COLUMNS、ALL COLUMNS），请设置系统变量 tidb_persist_analyze_options 的值设置为 ON 以开启ANALYZE 配置持久化特性。开启 ANALYZE 配置持久化特性后：

- 当 TiDB 自动收集统计信息或者你手动执行 ANALYZE 语句收集统计信息但未指定列的配置时，TiDB 会继续沿用之前持久化的配置。

- 当多次手动执行 ANALYZE 语句并指定列的配置时，TiDB 会使用最新一次 ANALYZE 指定的配置项覆盖上一次记录的持久化配置。

如果你想查看一个表中哪些列是 PREDICATE COLUMNS，哪些列的统计信息已经被收集，可以使用以下语法：

```
SHOW COLUMN_STATS_USAGE [ShowLikeOrWhere];
```

SHOW COLUMN_STATS_USAGE 会输出 6 列，具体如下：

列名	说明
Db_name	数据库名
Table_name	表名
Partition_name	分区名
Column_name	列名
Last_used_at	该列统计信息在最近一次查询优化中被用到的时间
Last_analyzed_at	该列统计信息最近一次被收集的时间

在以下示例中，执行 ANALYZE TABLE t PREDICATE COLUMNS; 后，TiDB 将收集 b，c，d 列的统计信息，其中 b 列是 PREDICATE COLUMN，c 列和 d 列是索引列。

```
SET GLOBAL tidb_enable_column_tracking = ON;
Query OK, 0 rows affected (0.00 sec)

CREATE TABLE t (a INT, b INT, c INT, d INT, INDEX idx_c_d(c, d));
Query OK, 0 rows affected (0.00 sec)

-- 在此查询中优化器用到了 b 列的统计信息。
SELECT * FROM t WHERE b > 1;
Empty set (0.00 sec)

-- 等待一段时间（100 * stats-lease）后，TiDB 将收集的 `PREDICATE COLUMNS` 写入 mysql.
    ↪ column_stats_usage。
-- 指定 `last_used_at IS NOT NULL` 表示显示 TiDB 收集到的 `PREDICATE COLUMNS`。
SHOW COLUMN_STATS_USAGE WHERE db_name = 'test' AND table_name = 't' AND last_used_at IS NOT NULL;
+---------+------------+----------------+-------------+---------------------+------------------+
| Db_name | Table_name | Partition_name | Column_name | Last_used_at        | Last_analyzed_at |
+---------+------------+----------------+-------------+---------------------+------------------+
| test    | t          |                | b           | 2022-01-05 17:21:33 | NULL             |
+---------+------------+----------------+-------------+---------------------+------------------+
1 row in set (0.00 sec)

ANALYZE TABLE t PREDICATE COLUMNS;
Query OK, 0 rows affected, 1 warning (0.03 sec)

-- 指定 `last_analyzed_at IS NOT NULL` 表示显示收集过统计信息的列。
```

```
SHOW COLUMN_STATS_USAGE WHERE db_name = 'test' AND table_name = 't' AND last_analyzed_at IS NOT
    ↪ NULL;
+--
    ↪ -------+------------+----------------+-------------+---------------------+-------------------+
    ↪
| Db_name | Table_name | Partition_name | Column_name | Last_used_at        | Last_analyzed_at
    ↪    |
+--
    ↪ -------+------------+----------------+-------------+---------------------+-------------------+
    ↪
| test    | t          |                | b           | 2022-01-05 17:21:33 | 2022-01-05 17:23:06
    ↪  |
| test    | t          |                | c           | NULL                | 2022-01-05 17:23:06
    ↪  |
| test    | t          |                | d           | NULL                | 2022-01-05 17:23:06
    ↪  |
+--
    ↪ -------+------------+----------------+-------------+---------------------+-------------------+
    ↪
3 rows in set (0.00 sec)
```

收集索引的统计信息

如果要收集 TableName 中 IndexNameList 里所有索引的统计信息，请使用以下语法：

```
ANALYZE TABLE TableName INDEX [IndexNameList] [WITH NUM BUCKETS|TOPN|CMSKETCH DEPTH|CMSKETCH
    ↪ WIDTH]|[WITH NUM SAMPLES|WITH FLOATNUM SAMPLERATE];
```

当 IndexNameList 为空时，该语法将收集 TableName 中所有索引的统计信息。

> 注意：
>
> 为了保证前后统计信息的一致性，当设置 `tidb_analyze_version=2` 时，该语句也会收集整个表的统计信息（包括所有列和所有索引的统计信息）而不限于索引的统计信息。

收集分区的统计信息

- 如果要收集 TableName 中所有的 PartitionNameList 中分区的统计信息，请使用以下语法：

```
ANALYZE TABLE TableName PARTITION PartitionNameList [WITH NUM BUCKETS|TOPN|CMSKETCH DEPTH|
    ↪ CMSKETCH WIDTH]|[WITH NUM SAMPLES|WITH FLOATNUM SAMPLERATE];
```

- 如果要收集 TableName 中所有的 PartitionNameList 中分区的索引统计信息，请使用以下语法：

```
ANALYZE TABLE TableName PARTITION PartitionNameList INDEX [IndexNameList] [WITH NUM BUCKETS|
    ↪ TOPN|CMSKETCH DEPTH|CMSKETCH WIDTH]|[WITH NUM SAMPLES|WITH FLOATNUM SAMPLERATE];
```

- 当收集分区的统计信息时，如果只收集部分列的统计信息，请使用以下语法：

> 警告：
>
> 收集 PREDICATE COLUMNS 的统计信息目前为实验特性，不建议在生产环境中使用。

```
ANALYZE TABLE TableName PARTITION PartitionNameList [COLUMNS ColumnNameList|PREDICATE
    ↪ COLUMNS|ALL COLUMNS] [WITH NUM BUCKETS|TOPN|CMSKETCH DEPTH|CMSKETCH WIDTH]|[WITH NUM
    ↪  SAMPLES|WITH FLOATNUM SAMPLERATE];
```

动态裁剪模式下的分区表统计信息

分区表在开启动态裁剪模式的情况下，TiDB 将收集表级别的汇总统计信息，以下称 GlobalStats。目前 GlobalStats 由分区统计信息合并汇总得到。在动态裁剪模式开启的情况下，任一分区上的统计信息更新都会触发 GlobalStats 的更新。

> 注意：
>
> - 当触发 GlobalStats 更新时：
>   - 若某些分区上缺少统计信息（比如新增的未 analyze 过的分区），会停止生成 GlobalStats，并通过 warning 信息提示用户缺少分区的统计信息。
>   - 若某些列的统计信息合并过程中，缺少某些分区在该列上的统计信息（在不同分区上 analyze 时指定了不同的列），会停止生成 GlobalStats，并通过 warning 信息提示用户缺少列在分区上的统计信息。
> - 在动态裁剪模式开启的情况下，分区和表的 ANALYZE 配置需要保持一致，因此 ANALYZE TABLE TableName PARTITION PartitionNameList 命令后指定的 COLUMNS 配置和 WITH 后指定的 OPTIONS 配置将被忽略，并会通过 warning 信息提示用户。

增量收集

对于类似时间列这样的单调不减列，在进行全量收集后，可以使用增量收集来单独分析新增的部分，以提高分析的速度。

> 注意：
>
> 1. 目前只有索引提供了增量收集的功能
> 2. 使用增量收集时，必须保证表上只有插入操作，且应用方需要保证索引列上新插入的值是单调不减的，否则会导致统计信息不准，影响 TiDB 优化器选择合适的执行计划

可以通过以下几种语法进行增量收集。

增量收集 TableName 中所有的 IndexNameList 中的索引列的统计信息：

```
ANALYZE INCREMENTAL TABLE TableName INDEX [IndexNameList] [WITH NUM BUCKETS|TOPN|CMSKETCH DEPTH|
    ↪ CMSKETCH WIDTH]|[WITH NUM SAMPLES|WITH FLOATNUM SAMPLERATE];
```

增量收集 TableName 中所有的 PartitionNameList 中分区的索引列统计信息：

```
ANALYZE INCREMENTAL TABLE TableName PARTITION PartitionNameList INDEX [IndexNameList] [WITH NUM
    ↪ BUCKETS|TOPN|CMSKETCH DEPTH|CMSKETCH WIDTH]|[WITH NUM SAMPLES|WITH FLOATNUM SAMPLERATE];
```

自动更新

在发生增加，删除以及修改语句时，TiDB 会自动更新表的总行数以及修改的行数。这些信息会定期持久化下来，更新的周期是 20 * stats-lease，stats-lease 的默认值是 3s，如果将其指定为 0，那么将不会自动更新。

和统计信息自动更新相关的三个系统变量如下：

系统变量名	默认值	功能
tidb_auto_analyze_ratio	0.5	自动更新阈值
tidb_auto_analyze_start_time	00:00 +0000	一天中能够进行自动更新的开始时间
tidb_auto_analyze_end_time	23:59 +0000	一天中能够进行自动更新的结束时间

当某个表 tbl 的修改行数与总行数的比值大于 tidb_auto_analyze_ratio，并且当前时间在 tidb_auto_analyze_start_time 和 tidb_auto_analyze_end_time 之间时，TiDB 会在后台执行 ANALYZE TABLE tbl 语句自动更新这个表的统计信息。

为了避免小表因为少量数据修改而频繁触发自动更新，当表的行数小于 1000 时，TiDB 不会触发对此表的自动更新。你可以通过 SHOW STATS_META 来查看表的行数情况。

从 TiDB v6.0 起，TiDB 支持通过 KILL 语句终止正在后台运行的 ANALYZE 任务。如果发现正在后台运行的 ANALYZE 任务消耗大量资源影响业务，你可以通过以下步骤终止该 ANALYZE 任务：

1. 执行以下 SQL 语句：

   ```
   SHOW ANALYZE STATUS
   ```

   查看 instance 列和 process_id 列获得正在执行后台 ANALYZE 任务的 TiDB 实例地址和任务 ID。

2. 终止正在后台运行的 ANALYZE 任务。

- 如果enable-global-kill 的值为 true (默认为 true)，你可以直接执行 KILL TIDB ${id}; 语句。其中，${id} 为上一步中查询得到的后台 ANALYZE 任务的 ID。
- 如果 enable-global-kill 的值为 false，你需要先使用客户端连接到执行后台 ANALYZE 任务的 TiDB 实例，然后再执行 KILL TIDB ${id}; 语句。如果使用客户端连接到其他 TiDB 实例，或者客户端和 TiDB 中间有代理，KILL 语句不能终止后台的 ANALYZE 任务。

关于 KILL 语句的更多信息，请参考KILL [TIDB]。

控制 ANALYZE 并发度

执行 ANALYZE 语句的时候，你可以通过一些参数来调整并发度，以控制对系统的影响。

tidb_build_stats_concurrency

目前 ANALYZE 执行的时候会被切分成一个个小的任务，每个任务只负责某一个列或者索引。tidb_build_stats_concurrency 可以控制同时执行的任务的数量，其默认值是 4。

tidb_distsql_scan_concurrency

在执行分析普通列任务的时候，tidb_distsql_scan_concurrency 可以用于控制一次读取的 Region 数量，其默认值是 15。

tidb_index_serial_scan_concurrency

在执行分析索引列任务的时候，tidb_index_serial_scan_concurrency 可以用于控制一次读取的 Region 数量，其默认值是 1。

ANALYZE 配置持久化

从 v5.4.0 起，TiDB 支持 ANALYZE 配置持久化功能，方便后续收集统计信息时沿用已有配置项。

TiDB 支持持久化的配置项包括：

配置项	对应的 ANALYZE 语法
直方图桶数	WITH NUM BUCKETS
TopN 个数	WITH NUM TOPN
采样数	WITH NUM SAMPLES
采样率	WITH FLOATNUM SAMPLERATE

配置项	对应的 ANALYZE 语法
ANALYZE 的列的类型	AnalyzeColumnOption ::= (‘ALL COLUMNS’ | ‘PREDICATE COLUMNS’ | ‘COLUMNS’ ColumnNameList)
ANALYZE 的列	ColumnNameList ::= Identifier (‘,’ Identifier)*

开启 ANALYZE 配置持久化功能

ANALYZE 配置持久化功能默认开启（系统变量 tidb_analyze_version 为默认值 2，tidb_persist_analyze_options 为默认值 ON），用于记录手动执行 ANALYZE 语句时指定的持久化配置项。记录后，当 TiDB 下一次自动更新统计信息或者你手动收集统计信息但未指定配置项时，TiDB 会按照记录的配置项收集统计信息。

多次手动执行 ANALYZE 语句并指定持久化配置项时，TiDB 会使用最新一次 ANALYZE 指定的配置项覆盖上一次记录的持久化配置。

关闭 ANALYZE 配置持久化功能

要关闭 ANALYZE 配置持久化功能，请设置系统变量 tidb_persist_analyze_options 为 OFF。由于 ANALYZE 配置持久化功能在 tidb_analyze_version = 1 的情况下不适用，因此设置 tidb_analyze_version = 1 同样会达到关闭配置持久化的效果。

关闭 ANALYZE 配置持久化功能后，已持久化的配置记录不会被清除。因此，当再次开启该功能时，TiDB 会继续使用之前记录的持久化配置收集统计信息。

> **注意：**
>
> 当再次开启 ANALYZE 配置持久化功能时，如果之前记录的持久化配置项已经不适用当前的数据，请手动执行 ANALYZE 语句并指定新的持久化配置项。

分区表的 ANALYZE 配置持久化功能

在静态裁剪模式下 ANALYZE 分区表时，配置持久化遵守：

- ANALYZE TABLE 时会持久化表级别的配置和实际被 ANALYZE 的所有分区的配置
- 分区的统计信息会继承使用表级别的持久化配置
- ANALYZE TABLE ··· PARTITION ··· WITH ··· 所指定的分区配置只持久化到分区级别，不会影响表级别的持久化配置
- 当 ANALYZE 语句指定了配置，且同时存在持久化配置时，按照语句 > 分区 > 表的优先级继承和重写配置信息

在动态裁剪模式下 ANALYZE 分区表时，配置持久化遵守：

- ANALYZE TABLE 时只持久化表级别的配置
- 分区的统计信息会继承使用表级别的持久化配置
- GlobalStats 会使用表级别的持久化配置
- ANALYZE TABLE ··· PARTITION ··· WITH ··· 所指定的分区配置会被忽略，且不会被持久化

统计信息收集的内存限制

> **警告：**
>
> 目前限制 ANALYZE 的内存使用量为实验特性，在生产环境中使用时可能存在内存统计有误差的情况。

TiDB 从 v6.1.0 开始引入了统计信息收集的内存限制，你可以通过`tidb_mem_quota_analyze` 变量来控制 TiDB 更新统计信息时的最大总内存占用。

要合理地配置 `tidb_mem_quota_analyze` 值，你需要考虑集群的数据。在使用默认采样率的情况下，主要考虑列的数量、列上的值的大小，以及 TiDB 的内存配置。你可参考以下建议来配置变量的最大值和最小值：

> **注意：**
>
> 以下配置建议仅供参考，实际配置需要在真实场景中测试确定。

- 最小值：需要大于 TiDB 从集群上列最多的表收集统计信息时使用的最大内存。一个粗略的参考信息是，在测试集上，20 列的表在默认配置下，统计信息收集的最大内存使用约为 800 MiB；160 列的表在默认配置下，统计信息收集的最大内存使用约为 5 GiB。
- 最大值：需要小于集群在不进行统计信息收集时的内存空余量。

查看 ANALYZE 状态

在执行 ANALYZE 时，可以通过 SQL 语句来查看当前 ANALYZE 的状态。

语法如下：

```
SHOW ANALYZE STATUS [ShowLikeOrWhere];
```

该语句会输出 ANALYZE 的状态，可以通过使用 ShowLikeOrWhere 来筛选需要的信息。

目前 SHOW ANALYZE STATUS 会输出 11 列，具体如下：

列名	说明
table_schema	数据库名
table_name	表名
partition_name	分区名
job_info	任务具体信息。如果分析索引，该信息会包含索引名。当 tidb_analyze_version =2 时，该信息会包含采样率
processed_rows	已经分析的行数
start_time	任务开始执行的时间
end_time	任务结束执行的时间
state	任务状态，包括 pending（等待）、running（正在执行）、finished（执行成功）和 failed（执行失败）
fail_reason	任务失败的原因。如果执行成功则为 NULL。
instance	执行任务的 TiDB 实例
process_id	执行任务的 process ID

从 TiDB v6.1.0 起，执行 SHOW ANALYZE STATUS 语句将显示集群级别的任务，且 TiDB 重启后仍能看到重启之前的任务记录。在 TiDB v6.1.0 之前，执行 SHOW ANALYZE STATUS 语句仅显示实例级别的任务，且 TiDB 重启后任务记录会被清空。

SHOW ANALYZE STATUS 仅显示最近的任务记录。从 TiDB v6.1 起，你可以通过系统表 mysql.analyze_jobs 查看过去 7 天内的历史记录。

当设置了tidb_mem_quota_analyze 且 TiDB 后台的统计信息自动更新任务的内存占用超过了这个阈值时，自动更新任务会重试。失败的任务和重试的任务都可以在 SHOW ANALYZE STATUS 的结果中查看。

当tidb_max_auto_analyze_time 大于 0 时，如果后台统计信息自动更新任务的执行时间超过这个阈值，该任务会被终止。

```
mysql> SHOW ANALYZE STATUS [ShowLikeOrWhere];
+--
    ↪ ------------+------------+----------------+-------------------------------------------------------------
    ↪
| Table_schema | Table_name | Partition_name | Job_info
    ↪                                                                                                  |
```

```
    ↪ Processed_rows | Start_time          | End_time            | State    | Fail_reason
    ↪                                                                 |
+--
    ↪ ------------+------------+---------------+-------------------------------------------------------------------
    ↪
| test         | sbtest1    |               | retry auto analyze table all columns with 100 topn
    ↪ , 0.055 samplerate                     |        2000000 | 2022-05-07 16:41:09 |
    ↪ 2022-05-07 16:41:20 | finished | NULL
    ↪                                                                                 |
| test         | sbtest1    |               | auto analyze table all columns with 100 topn, 0.5
    ↪ samplerate                             |              0 | 2022-05-07 16:40:50 |
    ↪ 2022-05-07 16:41:09 | failed   | analyze panic due to memory quota exceeds, please try
    ↪ with smaller samplerate |
```

统计信息的查看

你可以通过一些语句来查看统计信息的状态。

表的元信息

你可以通过 SHOW STATS_META 来查看表的总行数以及修改的行数等信息。

语法如下：

```
SHOW STATS_META [ShowLikeOrWhere];
```

其中，ShowLikeOrWhereOpt 部分的语法图为：

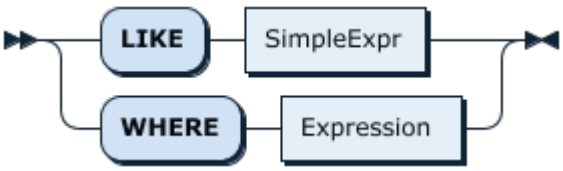


图 168: ShowLikeOrWhereOpt

目前 SHOW STATS_META 会输出 6 列，具体如下：

列名	说明
db_name	数据库名
table_name	表名
partition_name	分区名
update_time	更新时间
modify_count	修改的行数
row_count	总行数

> **注意：**
>
> 在 TiDB 根据 DML 语句自动更新总行数以及修改的行数时，`update_time` 也会被更新，因此并不能认为 `update_time` 是最近一次发生 Analyze 的时间。

表的健康度信息

通过 `SHOW STATS_HEALTHY` 可以查看表的统计信息健康度，并粗略估计表上统计信息的准确度。当 `modify_count >= row_count` 时，健康度为 0；当 `modify_count < row_count` 时，健康度为 `(1 - modify_count/row_count) * 100`。

语法如下：

```
SHOW STATS_HEALTHY [ShowLikeOrWhere];
```

`SHOW STATS_HEALTHY` 的语法图为：



图 169: ShowStatsHealthy

目前，`SHOW STATS_HEALTHY` 会输出 4 列，具体如下：

列名	说明
db_name	数据库名
table_name	表名
partition_name	分区名
healthy	健康度

列的元信息

你可以通过 `SHOW STATS_HISTOGRAMS` 来查看列的不同值数量以及 NULL 数量等信息。

语法如下：

```
SHOW STATS_HISTOGRAMS [ShowLikeOrWhere];
```

该语句会输出所有列的不同值数量以及 NULL 数量等信息，你可以通过 ShowLikeOrWhere 来筛选需要的信息。

目前 `SHOW STATS_HISTOGRAMS` 会输出 10 列，具体如下：

列名	说明
db_name	数据库名
table_name	表名
partition_name	分区名
column_name	根据 is_index 来变化：is_index 为 0 时是列名，为 1 时是索引名

列名	说明
is_index	是否是索引列
update_time	更新时间
distinct_count	不同值数量
null_count	NULL 的数量
avg_col_size	列平均长度
correlation	该列与整型主键的皮尔逊系数，表示两列之间的关联程度

直方图桶的信息

你可以通过 `SHOW STATS_BUCKETS` 来查看直方图每个桶的信息。

语法如下：

```
SHOW STATS_BUCKETS [ShowLikeOrWhere];
```

语法图：

SHOW STATS_BUCKETS:



图 170: SHOW STATS_BUCKETS

该语句会输出所有桶的信息，你可以通过 ShowLikeOrWhere 来筛选需要的信息。

目前 `SHOW STATS_BUCKETS` 会输出 11 列，具体如下：

列名	说明
db_name	数据库名
table_name	表名
partition_name	分区名
column_name	根据 is_index 来变化：is_index 为 0 时是列名，为 1 时是索引名
is_index	是否是索引列
bucket_id	桶的编号
count	所有落在这个桶及之前桶中值的数量
repeats	最大值出现的次数
lower_bound	最小值
upper_bound	最大值
ndv	当前桶内不同值的个数。当 `tidb_analyze_version` = 1 时，该值恒为 0，没有实际意义。

Top-N 信息

你可以通过 `SHOW STATS_TOPN` 来查看当前 TiDB 中收集的 Top-N 值的信息。

语法如下：

```
SHOW STATS_TOPN [ShowLikeOrWhere];
```

目前 SHOW STATS_TOPN 会输出 7 列，具体如下：

列名	说明
db_name	数据库名
table_name	表名
partition_name	分区名
column_name	根据 is_index 来变化：is_index 为 0 时是列名，为 1 时是索引名
is_index	是否是索引列
value	该列的值
count	该值出现的次数

删除统计信息

可以通过执行 DROP STATS 语句来删除统计信息。

```
DROP STATS TableName;
```

该语句会删除 TableName 中所有的统计信息，如果是分区表，包括所有分区的统计信息和分区动态裁剪模式下生成的 GlobalStats。

```
DROP STATS TableName PARTITION PartitionNameList;
```

该语句只删除 PartitionNameList 中对应的分区的统计信息。

```
DROP STATS TableName GLOBAL;
```

该语句只删除该表在分区动态裁剪模式下生成的 GlobalStats。

统计信息的加载

默认情况下，列的统计信息占用空间大小不同，TiDB 对统计信息的加载方式也会不同。

- 对于 count、distinctCount、nullCount 等占用空间较小的统计信息，只要有数据更新，TiDB 就会自动将对应的统计信息加载进内存供 SQL 优化阶段使用。
- 对于直方图、TopN、CMSketch 等占用空间较大的统计信息，为了确保 SQL 执行的性能，TiDB 会按需进行异步加载。例如，对于直方图，只有当某条 SQL 语句的优化阶段使用到了某列的直方图统计信息时，TiDB 才会将该列的直方图信息加载到内存。按需异步加载的优势是统计信息加载不会影响到 SQL 执行的性能，但在 SQL 优化时有可能使用不完整的统计信息。

从 v5.4.0 开始，TiDB 引入了统计信息同步加载的特性，支持执行当前 SQL 语句时将直方图、TopN、CMSketch 等占用空间较大的统计信息同步加载到内存，提高该 SQL 语句优化时统计信息的完整性。

要开启该特性，请将系统变量tidb_stats_load_sync_wait 的值设置为 SQL 优化等待加载列的完整统计信息的超时时间（单位为毫秒）。该变量默认值为 100，代表开启统计信息同步加载。

开启该特性后，你可以进一步配置该特性：

- 通过修改系统变量tidb_stats_load_pseudo_timeout 的值控制 SQL 优化等待超时后的行为。该变量默认值为 ON，代表超时后 SQL 优化过程不会使用任何列上的直方图、TopN 或 CMSketch。当设置该变量为 OFF 时，代表超时后 SQL 执行失败。
- 通过修改 TiDB 配置项stats-load-concurrency 的值控制统计信息同步加载可以并发处理的最大列数。该配置项的默认值为 5。
- 通过修改 TiDB 配置项stats-load-queue-size 的值设置统计信息同步加载最多可以缓存多少列的请求。该配置项的默认值为 1000。

统计信息的导入导出

导出统计信息

统计信息的导出接口如下。

- 通过以下接口可以获取数据库 ${db_name} 中的表 ${table_name} 的 JSON 格式的统计信息：

```
http://${tidb-server-ip}:${tidb-server-status-port}/stats/dump/${db_name}/${table_name}
```

  示例如下：

```
curl -s http://127.0.0.1:10080/stats/dump/test/t1 -o /tmp/t1.json
```

- 通过以下接口可以获取数据库 ${db_name} 中的表 ${table_name} 在指定时间上的 JSON 格式统计信息。指定的时间应在 GC SafePoint 之后。

```
http://${tidb-server-ip}:${tidb-server-status-port}/stats/dump/${db_name}/${table_name}/${
    ↪ yyyyMMddHHmmss}
```

导入统计信息

> 注意：
>
> 启动 MySQL 客户端时，请使用 --local-infile=1 参数。

导入的统计信息一般是通过统计信息导出接口得到的 json 文件。

语法如下：

```
LOAD STATS 'file_name';
```

file_name 为要导入的统计信息的文件名。

锁定统计信息

> 警告：
>
> 锁定统计信息目前为实验特性，不建议在生产环境中使用。

从 v6.5.0 开始，TiDB 引入了锁定统计信息的特性。当一张表的统计信息被锁定以后，该表的统计信息将无法被修改，也无法对该表进行 ANALYZE 操作。用例如下：

创建表 t，插入一些数据，在未锁定表 t 的统计信息的情况下成功执行 ANALYZE 语句。

```
mysql> create table t(a int, b int);
Query OK, 0 rows affected (0.03 sec)

mysql> insert into t values (1,2), (3,4), (5,6), (7,8);
Query OK, 4 rows affected (0.00 sec)
Records: 4  Duplicates: 0  Warnings: 0

mysql> analyze table t;
Query OK, 0 rows affected, 1 warning (0.02 sec)

mysql> show warnings;
+-------+------+----------------------------------------------------------------+
| Level | Code | Message                                                        |
+-------+------+----------------------------------------------------------------+
| Note  | 1105 | Analyze use auto adjusted sample rate 1.000000 for table test.t |
+-------+------+----------------------------------------------------------------+
1 row in set (0.00 sec)
```

锁定表 t 的统计信息，执行 ANALYZE 语句，warning 提示跳过对表 t 的 ANALYZE。

```
mysql> lock stats t;
Query OK, 0 rows affected (0.00 sec)

mysql> show stats_locked;
+---------+------------+----------------+--------+
| Db_name | Table_name | Partition_name | Status |
+---------+------------+----------------+--------+
| test    | t          |                | locked |
+---------+------------+----------------+--------+
1 row in set (0.01 sec)

mysql> analyze table t;
Query OK, 0 rows affected, 2 warnings (0.00 sec)

mysql> show warnings;
+---------+------+----------------------------------------------------------------+
| Level   | Code | Message                                                        |
+---------+------+----------------------------------------------------------------+
| Note    | 1105 | Analyze use auto adjusted sample rate 1.000000 for table test.t |
| Warning | 1105 | skip analyze locked table: t                                   |
+---------+------+----------------------------------------------------------------+
2 rows in set (0.00 sec)
```

解锁表 t 的统计信息，成功执行 ANALYZE 语句。

```
mysql> unlock stats t;
Query OK, 0 rows affected (0.01 sec)

mysql> analyze table t;
Query OK, 0 rows affected, 1 warning (0.03 sec)

mysql> show warnings;
+-------+------+-----------------------------------------------------------------+
| Level | Code | Message                                                         |
+-------+------+-----------------------------------------------------------------+
| Note  | 1105 | Analyze use auto adjusted sample rate 1.000000 for table test.t |
+-------+------+-----------------------------------------------------------------+
1 row in set (0.00 sec)
```

另请参阅

- LOAD STATS
- DROP STATS
- LOCK STATS
- UNLOCK STATS
- SHOW STATS_LOCKED

11.3.3.3.4 错误索引的解决方案

在观察到某个查询的执行速度达不到预期时，可能是它的索引使用有误，这时就需要通过一些手段来解决。通常可以先使用表的健康度信息来查看统计信息的健康度。根据健康度可以分为以下两种情况处理。

健康度较低

这意味着距离 TiDB 上次 ANALYZE 很久了。这时可以先使用 ANALYZE 命令对统计信息进行更新。更新之后如果索引的使用上还是错误的，可以查看下一小节。

健康度接近 100%

这时意味着刚刚结束 ANALYZE 命令或者结束后不久。这时可能和 TiDB 对行数的估算逻辑有关。

对于等值查询，错误索引可能是由Count-Min Sketch 引起的。这时可以先检查是不是这种特殊情况，然后进行对应的处理。

如果经过检查发现不是上面的可能情况，可以使用Optimizer Hints 中提到的 USE_INDEX 或者 use index 来强制选择索引。同时也可以使用执行计划管理中提到的方式来非侵入地更改查询的行为。

其他情况

除去上述情况外，也存在因为数据的更新导致现有所有索引都不再适合的情况。这时就需要对条件和数据分布进行分析，查看是否有新的索引可以加快查询速度，然后使用ADD INDEX 命令增加新的索引。

11.3.3.3.5 Distinct 优化

本文档介绍可用于 DISTINCT 的优化，包括简单 DISTINCT 和聚合函数 DISTINCT 的优化。

简单 DISTINCT

通常简单的 DISTINCT 会被优化成 GROUP BY 来执行。例如：

```
mysql> explain select DISTINCT a from t;
+--
    ↪ ----------------------+---------+-----------+---------------+--------------------------------
    ↪
| id                      | estRows | task      | access object | operator info
    ↪                                        |
+--
    ↪ ----------------------+---------+-----------+---------------+--------------------------------
    ↪
| HashAgg_6               | 2.40    | root      |               | group by:test.t.a, funcs:
    ↪ firstrow(test.t.a)->test.t.a |
| └─TableReader_11        | 3.00    | root      |               | data:TableFullScan_10
    ↪                                  |
|   └─TableFullScan_10    | 3.00    | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo                         |
+--
    ↪ ----------------------+---------+-----------+---------------+--------------------------------
    ↪
3 rows in set (0.00 sec)
```

聚合函数 DISTINCT

通常来说，带有 DISTINCT 的聚合函数会单线程的在 TiDB 侧执行。使用系统变量 tidb_opt_distinct_agg_push_down
↪ 或者 TiDB 的配置项 distinct-agg-push-down 控制优化器是否执行带有 DISTINCT 的聚合函数（比如 select
↪ count(distinct a)from t）下推到 Coprocessor 的优化操作。

在以下示例中，tidb_opt_distinct_agg_push_down 开启前，TiDB 需要从 TiKV 读取所有数据，并在 TiDB 侧执行 disctinct。tidb_opt_distinct_agg_push_down 开启后，distinct a 被下推到了 Coprocessor，在 HashAgg_5 里新增了一个 group by 列 test.t.a。

```
mysql> desc select count(distinct a) from test.t;
+--
    ↪ ----------------------+----------+-----------+---------------+--------------------------------
    ↪
| id                      | estRows  | task      | access object | operator info
    ↪                            |
+--
    ↪ ----------------------+----------+-----------+---------------+--------------------------------
    ↪
| StreamAgg_6             | 1.00     | root      |               | funcs:count(distinct test.t.a)
    ↪ ->Column#4 |
```

```
| └─TableReader_10        | 10000.00 | root      |               | data:TableFullScan_9
    ↪                     |
|   └─TableFullScan_9     | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo            |
+--
    ↪ ----------------------+----------+-----------+---------------+--------------------------------
    ↪
3 rows in set (0.01 sec)

mysql> set session tidb_opt_distinct_agg_push_down = 1;
Query OK, 0 rows affected (0.00 sec)

mysql> desc select count(distinct a) from test.t;
+--
    ↪ ------------------------+----------+-----------+---------------+--------------------------------
    ↪
| id                        | estRows  | task      | access object | operator info
    ↪                           |
+--
    ↪ ------------------------+----------+-----------+---------------+--------------------------------
    ↪
| HashAgg_8                 | 1.00     | root      |               | funcs:count(distinct test.t.
    ↪ a)->Column#3 |
| └─TableReader_9           | 1.00     | root      |               | data:HashAgg_5
    ↪                           |
|   └─HashAgg_5             | 1.00     | cop[tikv] |               | group by:test.t.a,
    ↪                         |
|     └─TableFullScan_7     | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo            |
+--
    ↪ ------------------------+----------+-----------+---------------+--------------------------------
    ↪
4 rows in set (0.00 sec)
```

11.3.3.3.6 代价模型

TiDB 在进行物理优化时会使用代价模型来进行索引选择和算子选择，如下图所示：

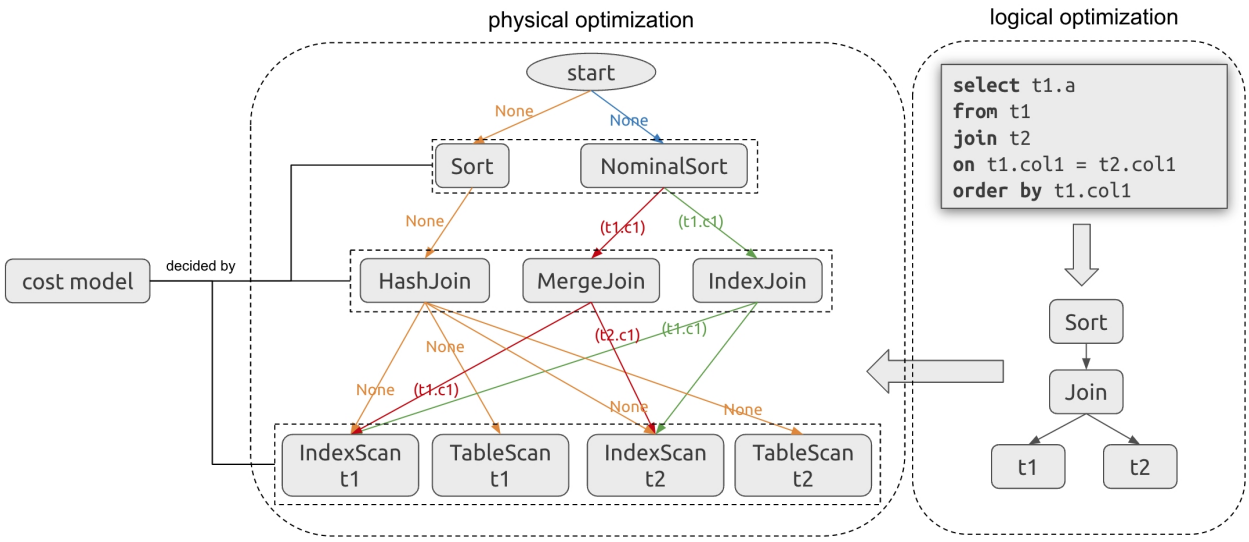


图 171: CostModel

TiDB 会计算每个索引的访问代价和计划中每个物理算子的执行代价（如 HashJoin、IndexJoin 等），选择代价最低的计划。

下面是一个简化的例子，用来解释代价模型的原理，比如有这样一张表：

```
mysql> SHOW CREATE TABLE t;
+--
    ↪ -----+----------------------------------------------------------------------------------------------------------------
    ↪
| Table | Create Table
    ↪
    ↪ |
+--
    ↪ -----+----------------------------------------------------------------------------------------------------------------
    ↪
| t     | CREATE TABLE `t` (
  `a` int(11) DEFAULT NULL,
  `b` int(11) DEFAULT NULL,
  `c` int(11) DEFAULT NULL,
  KEY `b` (`b`),
  KEY `c` (`c`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin |
+--
    ↪ -----+----------------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

在处理查询 `SELECT * FROM t WHERE b < 100 and c < 100` 时，假设 TiDB 对 `b < 100` 和 `c < 100` 的行数估计分别为 20 和 500，INT 类型索引行宽为 8，则 TiDB 会分别计算两个索引的代价：

- 索引 b 的扫描代价 = b < 100 的行数 * 索引 b 的行宽 = 20 * 8 = 160
- 索引 c 的扫描代价 = c < 100 的行数 * 索引 c 的行宽 = 500 * 8 = 4000

由于扫描 b 的代价更低，因此 TiDB 会选择 b 索引。

上述是一个简化后的例子，只是用于做原理解释，实际 TiDB 的代价模型会更加复杂。

Cost Model Version 2

TiDB v6.2.0 引入了新的代价模型 Cost Model Version 2。

Cost Model Version 2 对代价公式进行了更精确的回归校准，调整了部分代价公式，比此前版本的代价公式更加准确。

你可以通过设置变量`tidb_cost_model_version`来控制代价模型的版本。

> 注意：
> 切换代价模型版本可能引起执行计划的变动。

11.3.3.4 执行计划缓存

TiDB 支持对 Prepare / Execute 请求的执行计划缓存。其中包括以下两种形式的预处理语句：

- 使用 COM_STMT_PREPARE 和 COM_STMT_EXECUTE 的协议功能；
- 执行 Prepare / Execute SQL 语句查询；

TiDB 优化器对这两类查询的处理是一样的：Prepare 时将参数化的 SQL 查询解析成 AST（抽象语法树），每次 Execute 时根据保存的 AST 和具体的参数值生成执行计划。

当开启执行计划缓存后，每条 Prepare 语句的第一次 Execute 会检查当前查询是否可以使用执行计划缓存，如果可以则将生成的执行计划放进一个由 LRU 链表构成的缓存中；在后续的 Execute 中，会先从缓存中获取执行计划，并检查是否可用，如果获取和检查成功则跳过生成执行计划这一步，否则重新生成执行计划并放入缓存中。

在当前版本中，当 Prepare 语句符合以下条件任何一条，查询或者计划不会被缓存：

- SELECT、UPDATE、INSERT、DELETE、Union、Intersect、Except 以外的 SQL 语句；
- 访问分区表、临时表或访问表中包含生成列的查询；
- 查询中包含非关联子查询，例如 `SELECT * FROM t1 WHERE t1.a > (SELECT 1 FROM t2 WHERE t2.b < 1)`；
- 执行计划中带有 PhysicalApply 算子的关联子查询，例如 `SELECT * FROM t1 WHERE t1.a > (SELECT a FROM t2 WHERE t1.b > t2.b)`；
- 包含 ignore_plan_cache 这一 Hint 的查询，例如 `select /*+ ignore_plan_cache()*/ * from t`；
- 包含除 ? 外其他变量（即系统变量或用户自定义变量）的查询，例如 `select * from t where a>? and b>@x`；
- 查询包含无法被缓存函数。目前不能被缓存的函数有：database()、current_user、current_role、user、connection_id、last_insert_id、row_count、version、like；

- LIMIT 后面带有变量 (例如 LIMIT ? 或 LIMIT 10, ?) 且变量值大于 10000 的执行计划不缓存；
- ? 直接在 Order By 后的查询，如 Order By ?，此时 ? 表示根据 Order By 后第几列排序，排序列不同的查询使用同一个计划可能导致错误结果，故不缓存；如果是普通表达式，如 Order By a+? 则会缓存；
- ? 紧跟在 Group by 后的查询，如 Group By ?，此时 ? 表示根据 Group By 后第几列聚合，聚合列不同的查询使用同一个计划可能导致错误结果，故不缓存；如果是普通表达式，如 Group By a+? 则会缓存；
- ? 出现在窗口函数 Window Frame 定义中的查询，如 (partition by year order by sale rows ? preceding)；如果 ? 出现在窗口函数的其他位置，则会缓存；
- 用参数进行 int 和 string 比较的查询，如 c_int >= ? 或者 c_int in (?, ?)等，其中 ? 为字符串类型，如 set @x='123'；此时为了保证结果和 MySQL 兼容性，需要每次对参数进行调整，故不会缓存；
- 会访问 TiFlash 的计划不会被缓存；
- 大部分情况下计划中含有 TableDual 的计划将将不会被缓存，除非当前执行的 Prepare 语句不含参数，则对应的 TableDual 计划可以被缓存。

LRU 链表是设计成 session 级别的缓存，因为 Prepare / Execute 不能跨 session 执行。LRU 链表的每个元素是一个 key-value 对，value 是执行计划，key 由如下几部分组成：

- 执行 Execute 时所在数据库的名字；
- Prepare 语句的标识符，即紧跟在 PREPARE 关键字后的名字；
- 当前的 schema 版本，每条执行成功的 DDL 语句会修改 schema 版本；
- 执行 Execute 时的 SQL Mode；
- 当前设置的时区，即系统变量 time_zone 的值；
- 系统变量 sql_select_limit 的值；

key 中任何一项变动（如切换数据库，重命名 Prepare 语句，执行 DDL，或修改 SQL Mode / time_zone 的值），或 LRU 淘汰机制触发都会导致 Execute 时无法命中执行计划缓存。

成功从缓存中获取到执行计划后，TiDB 会先检查执行计划是否依然合法，如果当前 Execute 在显式事务里执行，并且引用的表在事务前序语句中被修改，而缓存的执行计划对该表访问不包含 UnionScan 算子，则它不能被执行。

在通过合法性检测后，会根据当前最新参数值，对执行计划的扫描范围做相应调整，再用它执行获取数据。

关于执行计划缓存和查询性能有几点值得注意：

- 不管计划是否已经被缓存，都会受到 SQL Binding 的影响。对于没有被缓存的计划，即在第一次执行 Execute 时，会受到已有 SQL Binding 的影响；而对于已经缓存的计划，如果有新的 SQL Binding 被创建产生，则原有已经被缓存的计划会失效。
- 已经被缓存的计划不会受到统计信息更新、优化规则和表达式下推黑名单更新的影响，仍然会使用已经保存在缓存中的计划。
- 重启 TiDB 实例时（如不停机滚动升级 TiDB 集群），Prepare 信息会丢失，此时执行 execute stmt ... 可能会遇到 Prepared Statement not found 的错误，此时需要再执行一次 prepare stmt ...。
- 考虑到不同 Execute 的参数会不同，执行计划缓存为了保证适配性会禁止一些和具体参数值密切相关的激进查询优化手段，导致对特定的一些参数值，查询计划可能不是最优。比如查询的过滤条件为 where a > ? and a < ?，第一次 Execute 时参数分别为 2 和 1，考虑到这两个参数下次执行时可能会是 1 和 2，优化器不会生成对当前参数最优的 TableDual 执行计划。
- 如果不考虑缓存失效和淘汰，一份执行计划缓存会对应各种不同的参数取值，理论上也会导致某些取值下执行计划非最优。比如查询过滤条件为 where a < ?，假如第一次执行 Execute 时用的参数值为

1，此时优化器生成最优的 IndexScan 执行计划放入缓存，在后续执行 Exeucte 时参数变为 10000，此时 TableScan 可能才是更优执行计划，但由于执行计划缓存，执行时还是会使用先前生成的 IndexScan。因此执行计划缓存更适用于查询较为简单（查询编译耗时占比较高）且执行计划较为固定的业务场景。

自 v6.1.0 起，执行计划缓存功能默认打开，可以通过变量tidb_enable_prepared_plan_cache 启用或关闭这项功能。

> **注意：**
>
> 执行计划缓存功能仅针对 Prepare / Execute 请求，对普通查询无效。

在开启了执行计划缓存功能后，可以通过 session 级别的系统变量 last_plan_from_cache 查看上一条 Execute 语句是否使用了缓存的执行计划，例如：

```
MySQL [test]> create table t(a int);
Query OK, 0 rows affected (0.00 sec)

MySQL [test]> prepare stmt from 'select * from t where a = ?';
Query OK, 0 rows affected (0.00 sec)

MySQL [test]> set @a = 1;
Query OK, 0 rows affected (0.00 sec)

-- 第一次 execute 生成执行计划放入缓存
MySQL [test]> execute stmt using @a;
Empty set (0.00 sec)

MySQL [test]> select @@last_plan_from_cache;
+------------------------+
| @@last_plan_from_cache |
+------------------------+
| 0                      |
+------------------------+
1 row in set (0.00 sec)

-- 第二次 execute 命中缓存
MySQL [test]> execute stmt using @a;
Empty set (0.00 sec)

MySQL [test]> select @@last_plan_from_cache;
+------------------------+
| @@last_plan_from_cache |
+------------------------+
| 1                      |
```

```
+------------------------+
1 row in set (0.00 sec)
```

如果发现某一组 Prepare / Execute 由于执行计划缓存导致了非预期行为，可以通过 SQL Hint ignore_plan_cache() 让该组语句不使用缓存。还是用上述的 stmt 为例：

```
MySQL [test]> prepare stmt from 'select /*+ ignore_plan_cache() */ * from t where a = ?';
Query OK, 0 rows affected (0.00 sec)

MySQL [test]> set @a = 1;
Query OK, 0 rows affected (0.00 sec)

MySQL [test]> execute stmt using @a;
Empty set (0.00 sec)

MySQL [test]> select @@last_plan_from_cache;
+------------------------+
| @@last_plan_from_cache |
+------------------------+
| 0                      |
+------------------------+
1 row in set (0.00 sec)

MySQL [test]> execute stmt using @a;
Empty set (0.00 sec)

MySQL [test]> select @@last_plan_from_cache;
+------------------------+
| @@last_plan_from_cache |
+------------------------+
| 0                      |
+------------------------+
1 row in set (0.00 sec)
```

11.3.3.4.1 诊断 Prepared Plan Cache

对于无法进行缓存的查询或计划，可通过 SHOW WARNINGS 语句查看查询或计划是否被缓存。如果未被缓存，则可在结果中查看无法被缓存的原因。示例如下：

```
mysql> PREPARE st FROM 'SELECT * FROM t WHERE a > (SELECT MAX(a) FROM t)';  -- 该查询包含子查询，因此无法被缓存
Query OK, 0 rows affected, 1 warning (0.01 sec)

mysql> SHOW WARNINGS;  -- 查看查询计划无法被缓存的原因
+---------+------+----------------------------------------------+
| Level   | Code | Message                                      |
```

```
+---------+------+--------------------------------------------------+
| Warning | 1105 | skip plan-cache: sub-queries are un-cacheable |
+---------+------+--------------------------------------------------+
1 row in set (0.00 sec)

mysql> PREPARE st FROM 'SELECT * FROM t WHERE a<?';
Query OK, 0 rows affected (0.00 sec)

mysql> SET @a='1';
Query OK, 0 rows affected (0.00 sec)

mysql> EXECUTE st USING @a;  -- 该优化中进行了非 INT 类型到 INT 类型的转换，
    ↪ 产生的执行计划可能随着参数变化而存在风险，因此 TiDB 不缓存该计划
Empty set, 1 warning (0.01 sec)

mysql> SHOW WARNINGS;
+---------+------+--------------------------------------------------+
| Level   | Code | Message                                          |
+---------+------+--------------------------------------------------+
| Warning | 1105 | skip plan-cache: '1' may be converted to INT |
+---------+------+--------------------------------------------------+
1 row in set (0.00 sec)
```

11.3.3.4.2 Prepared Plan Cache 的内存管理

使用 Prepared Plan Cache 会有一定的内存开销，可以通过 Grafana 中的Plan Cache Memory Usage 监控查看每台 TiDB 实例上所有 SESSION 所缓存的计划占用的总内存。

> **注意：**
>
> 考虑到 Golang 的内存回收机制以及部分未统计的内存结构，Grafana 中显示的内存与实际的堆内存使用量并不相等。经过实验验证存在约 ±20% 的误差。

对于每台 TiDB 实例上所缓存的执行计划总数量，可以通过 Grafana 中的Plan Cache Plan Num 监控查看。

Grafana 中 Plan Cache Memory Usage 和 Plan Cache Plan Num 监控如下图所示：

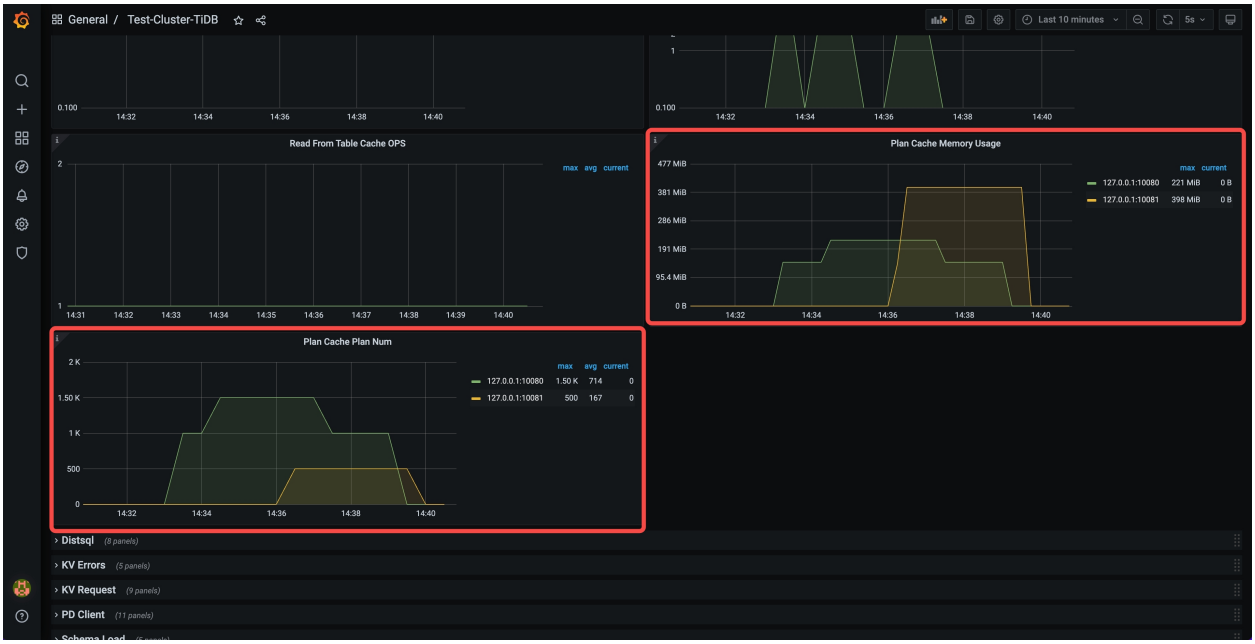


图 172: grafana_panels

目前可以通过变量 `tidb_prepared_plan_cache_size` 来设置每个 SESSION 最多缓存的计划数量，针对不同的环境，推荐的设置如下，你可以结合监控进行调整：

- TiDB Server 实例内存阈值 <= 64 GiB 时，`tidb_prepared_plan_cache_size = 50`
- TiDB Server 实例内存阈值 > 64 GiB 时，`tidb_prepared_plan_cache_size = 100`

当 TiDB Server 的内存余量小于一定阈值时，会触发 Plan Cache 的内存保护机制，此时会对一些缓存的计划进行逐出。

目前该阈值由变量 `tidb_prepared_plan_cache_memory_guard_ratio` 控制，默认为 0.1，即 10%，也就是当剩余内存不足 10%（使用内存超过 90%）时，会触发此机制。

由于内存限制，Plan Cache 可能出现 Cache Miss 的情况，可以通过 Grafana 中的Plan Cache Miss OPS 监控查看。

11.3.3.4.3 手动清空计划缓存

通过执行 `ADMIN FLUSH [SESSION | INSTANCE] PLAN_CACHE` 语句，你可以手动清空计划缓存。

该语句中的作用域 `[SESSION | INSTANCE]` 用于指定需要清空的缓存级别，可以为 SESSION 或 INSTANCE。如果不指定作用域，该语句默认清空 SESSION 级别的缓存。

下面是一个清空计划缓存的例子：

```
MySQL [test]> create table t (a int);
Query OK, 0 rows affected (0.00 sec)

MySQL [test]> prepare stmt from 'select * from t';
Query OK, 0 rows affected (0.00 sec)
```

```
MySQL [test]> execute stmt;
Empty set (0.00 sec)

MySQL [test]> execute stmt;
Empty set (0.00 sec)

MySQL [test]> select @@last_plan_from_cache; -- 选择计划缓存
+------------------------+
| @@last_plan_from_cache |
+------------------------+
|                      1 |
+------------------------+
1 row in set (0.00 sec)

MySQL [test]> admin flush session plan_cache; -- 清空当前 session 的计划缓存
Query OK, 0 rows affected (0.00 sec)

MySQL [test]> execute stmt;
Empty set (0.00 sec)

MySQL [test]> select @@last_plan_from_cache; -- 由于缓存被清空，此时无法再次选中
+------------------------+
| @@last_plan_from_cache |
+------------------------+
|                      0 |
+------------------------+
1 row in set (0.00 sec)
```

TiDB 暂不支持清空 GLOBAL 级别的计划缓存，即不支持一次性清空整个集群的计划缓存，使用时会报错：

```
MySQL [test]> admin flush global plan_cache;
ERROR 1105 (HY000): Do not support the 'admin flush global scope.'
```

11.3.3.4.4 忽略 COM_STMT_CLOSE 指令和 DEALLOCATE PREPARE 语句

为了减少每次执行 SQL 语句的语法分析，Prepared Statement 推荐的使用方式是，prepare 一次，然后 execute 多次，最后 deallocate prepare。例如：

```
MySQL [test]> prepare stmt from '...'; -- prepare 一次
MySQL [test]> execute stmt using ...;  -- execute 一次
MySQL [test]> ...
MySQL [test]> execute stmt using ...;  -- execute 多次
MySQL [test]> deallocate prepare stmt; -- 使用完成后释放
```

如果你习惯于在每次 execute 后都立即执行 deallocate prepare，如：

```
MySQL [test]> prepare stmt from '...'; -- 第一次 prepare
MySQL [test]> execute stmt using ...;
MySQL [test]> deallocate prepare stmt; -- 一次使用后立即释放
MySQL [test]> prepare stmt from '...'; -- 第二次 prepare
MySQL [test]> execute stmt using ...;
MySQL [test]> deallocate prepare stmt; -- 再次释放
```

这样的使用方式会让第一次执行得到的计划被立即清理，不能在第二次被复用。

为了兼容这样的使用方式，从 v6.0 起，TiDB 支持tidb_ignore_prepared_cache_close_stmt 变量。打开该变量后，TiDB 会忽略关闭 Prepare Statement 的信号，解决上述问题，如：

```
mysql> set @@tidb_ignore_prepared_cache_close_stmt=1;  -- 打开开关
Query OK, 0 rows affected (0.00 sec)

mysql> prepare stmt from 'select * from t'; -- 第一次 prepare
Query OK, 0 rows affected (0.00 sec)

mysql> execute stmt;                        -- 第一次 execute
Empty set (0.00 sec)

mysql> deallocate prepare stmt;             -- 第一次 execute 后立即释放
Query OK, 0 rows affected (0.00 sec)

mysql> prepare stmt from 'select * from t'; -- 第二次 prepare
Query OK, 0 rows affected (0.00 sec)

mysql> execute stmt;                        -- 第二次 execute
Empty set (0.00 sec)

mysql> select @@last_plan_from_cache;       -- 因为开关打开，第二次依旧能复用上一次的计划
+------------------------+
| @@last_plan_from_cache |
+------------------------+
|                      1 |
+------------------------+
1 row in set (0.00 sec)
```

监控

在Grafana 面板的 TiDB 页面，Executor 部分包含“Queries Using Plan Cache OPS”和“Plan Cache Miss OPS”两个图表，用以检查 TiDB 和应用是否正确配置，以便 SQL 执行计划缓存能正常工作。TiDB 页面的 Server 部分还提供了“Prepared Statement Count”图表，如果应用使用了预处理语句，这个图表会显示非零值。通过数值变化，可以判断 SQL 执行计划缓存是否正常工作。



图 173: sql_plan_cache

11.3.3.5 非 Prepare 语句执行计划缓存

> **警告：**
>
> 非 Prepare 语句执行计划缓存 (Non-Prepared Plan Cache) 目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现 bug，请在 GitHub 上提 issue 反馈。

对于某些非 PREPARE 语句，TiDB 可以像Prepare/Execute 语句一样支持执行计划缓存。这可以让这些语句跳过优化器阶段，以提升性能。

11.3.3.5.1 原理

Non-Prepared Plan Cache 为会话级别，并且与Prepared Plan Cache 相互独立，缓存的计划互不影响。Non-Prepared Plan Cache 功能的基本原理如下：

1. 开启 Non-Prepared Plan Cache 后，TiDB 首先根据 AST（抽象语法树）对查询进行参数化。例如，将 SELECT * FROM t WHERE b < 10 AND a = 1 参数化为 SELECT * FROM t WHERE b < ? and a = ?。
2. 然后，使用参数化后的查询在 Non-Prepared Plan Cache 中查找。
3. 如果能找到可以直接复用的计划，则直接使用，并跳过整个优化过程。
4. 否则，继续进行查询优化，并在最后将生成的计划放回到缓存中，以便下次复用。

11.3.3.5.2 使用方法

目前，你可以通过tidb_enable_non_prepared_plan_cache 开启或关闭 Non-Prepared Plan Cache。同时，你还可以通过tidb_non_prepared_plan_cache_size 来控制 Non-Prepared Plan Cache 的大小。当缓存的计划数超过 tidb_non_prepared_plan_cache_size 时，TiDB 会使用 LRU (Least Recently Used) 策略进行逐出。

11.3.3.5.3 示例

下面是一个使用示例：

1. 创建用于测试的表 t：

```
CREATE TABLE t (a INT, b INT, KEY(b));
```

2. 开启 Non-Prepared Plan Cache：

```
SET tidb_enable_non_prepared_plan_cache = true;
```

3. 依次执行以下查询：

```
SELECT * FROM t WHERE b < 10 AND a = 1;
SELECT * FROM t WHERE b < 5 AND a = 2;
```

4. 查看第二个查询语句是否命中缓存：

```
SELECT @@last_plan_from_cache;
```

输出结果中 last_plan_from_cache 的值为 1，表示第二次执行的查询计划来自于缓存：

```
+------------------------+
| @@last_plan_from_cache |
+------------------------+
|                      1 |
+------------------------+
1 row in set (0.00 sec)
```

11.3.3.5.4 限制

TiDB 对参数化后形式相同的查询，只能缓存一个计划。例如，对于 SELECT * FROM t WHERE a < 1 和 SELECT * FROM t WHERE a < 100000 这两个查询语句，由于参数化后的形式相同，均为 SELECT * FROM t WHERE a < ?，因此它们会共用一个计划。

如果由此产生性能问题，可以使用 ignore_plan_cache() Hint 忽略计划缓存中的计划，让优化器每次重新为 SQL 生成执行计划。如果无法修改 SQL，可以通过创建 binding 来解决，例如 CREATE BINDING FOR SELECT ... USING SELECT /*+ ignore_plan_cache()*/ ...。

由于上述风险以及执行计划缓存只在简单查询上有明显收益（如果查询较为复杂，查询本身执行时间较长，使用执行计划缓存收益不大），TiDB 目前对 Non-Prepared Plan Cache 的生效范围有严格的限制。具体限制如下：

- Prepared Plan Cache 不支持的查询或者计划，Non-Prepared Plan Cache 也不支持。
- 目前仅支持包含 Scan、Selection 或 Projection 算子的单表的点查或范围查询，例如 SELECT * FROM t WHERE a < 10 AND b in (1, 2)。
- 不支持包含 Agg、Limit、Window 或 Sort 等复杂算子的查询。
- 不支持包含非范围查询条件，例如：

  - 不支持 LIKE，例如 c LIKE 'c%'
  - 不支持 + 操作，例如 a+1 < 2

- 不支持过滤条件中包含 JSON、ENUM、SET 或 BIT 类型的列的查询，例如 SELECT * FROM t WHERE json_col = '{}'。
- 不支持过滤条件中出现 NULL 值的查询，例如 SELECT * FROM t WHERE a is NULL。
- 不支持参数化后参数个数超过 50 个的查询，例如 SELECT * FROM t WHERE a in (1, 2, 3, ... 51)。
- 不支持访问分区表、虚拟列、临时表、视图、或内存表的查询，例如 SELECT * FROM INFORMATION_SCHEMA.COLUMNS，其中 COLUMNS 为 TiDB 内存表。
- 不支持带有 Hint、子查询、Lock 的查询。
- 不支持 DML 语句。

11.3.3.5.5 诊断

开启 Non-Prepared Plan Cache 后，可以使用 EXPLAIN FORMAT='plan_cache' SELECT ... 语句验证查询是否能够命中缓存。对于无法命中缓存的查询，系统会通过 warning 的方式返回无法命中的原因。

需要注意的是，如果不加 FORMAT='plan_cache'，则 EXPLAIN 语句永远不会命中缓存。

执行下面 EXPLAIN FORMAT='plan_cache' 语句，查看查询是否能够命中：

```
EXPLAIN FORMAT='plan_cache' SELECT * FROM t WHERE a+2 < 10;
```

输出结果示例如下：

```
3 rows in set, 1 warning (0.00 sec)
```

通过 SHOW warnings; 查看无法命中缓存的查询信息：

```
SHOW warnings;
```

输出结果示例如下：

```
+---------+------+-----------------------------------------------------------------------------+
| Level   | Code | Message                                                                     |
+---------+------+-----------------------------------------------------------------------------+
| Warning | 1105 | skip non-prep plan cache: query has some unsupported binary operation |
+---------+------+-----------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

可以看到在上述例子中，由于 Non-Prepared Plan Cache 不支持 + 操作，所以无法命中缓存。

11.3.3.5.6 监控

开启 Non-Prepared Plan Cache 后，可以在以下几个面板中查看缓存的内存使用情况、缓存中计划的个数、缓存命中的情况等信息。



图 174: non-prepared-plan-cache

`statements_summary` 表和慢查询日志也会体现缓存的命中情况。下面是查看 `statements_summary` 表中缓存命中情况的例子：

1. 创建表 `t`：

```
CREATE TABLE t (a int);
```

2. 打开 Non-Prepared Plan Cache 开关：

```
SET @@tidb_enable_non_prepared_plan_cache=1;
```

3. 依次执行以下三个查询：

```
SELECT * FROM t WHERE a<1;
SELECT * FROM t WHERE a<2;
SELECT * FROM t WHERE a<3;
```

4. 查询 `statements_summary` 表查看查询命中缓存的情况：

```
SELECT digest_text, query_sample_text, exec_count, plan_in_cache, plan_cache_hits FROM
    ↪ INFORMATION_SCHEMA.STATEMENTS_SUMMARY WHERE digest_text LIKE '%SELECT * FROM %';
```

输出结果如下：

```
+--
    ↪ ------------------------------+-------------------------------------------+------------+------------
    ↪
| digest_text                       | query_sample_text                         | exec_count |
    ↪ plan_in_cache | plan_cache_hits |
+--
    ↪ ------------------------------+-------------------------------------------+------------+------------
    ↪
```

```
| SELECT * FROM `t` WHERE `a` < ? | SELECT * FROM t WHERE a<1 [arguments: 1] |          3 |
    ↪             1 |               2 |
+--
    ↪ ------------------------------+-----------------------------------------+------------+------------
    ↪
1 row in set (0.01 sec)
```

可以看到，查询执行了三次且命中缓存两次。

11.3.4 控制执行计划

11.3.4.1 控制执行计划

SQL 性能调优的前两个章节介绍了如何理解 TiDB 的执行计划以及 TiDB 如何生成一个执行计划。本章节将介绍当你确定了执行计划所存在的问题时，可以使用哪些手段来控制执行计划的生成。本章节主要包括以下三方面内容：

- Optimizer Hints中，我们会介绍如何使用 Hint 来指导 TiDB 生成执行计划。
- 但是使用 Hint 会侵入性地更改 SQL，在一些场景下并不能简单的插入 Hint。在执行计划管理中，我们会介绍 TiDB 如何使用另一种语法来非侵入地控制执行计划的生成，同时还会介绍后台自动对执行计划进行演进的手段。该手段可用来减轻诸如版本升级等原因造成的执行计划不稳定和集群性能下降的问题。
- 最后在优化规则及表达式下推的黑名单中，我们会介绍黑名单的使用。

11.3.4.2 Optimizer Hints

TiDB 支持 Optimizer Hints 语法，它基于 MySQL 5.7 中介绍的类似 comment 的语法，例如 `/*+ HINT_NAME(t1, t2)*/`。当 TiDB 优化器选择的不是最优查询计划时，建议使用 Optimizer Hints。

> **注意：**
>
> MySQL 命令行客户端在 5.7.7 版本之前默认清除了 Optimizer Hints。如果需要在这些早期版本的客户端中使用 `Hint` 语法，需要在启动客户端时加上 `--comments` 选项，例如 `mysql -h 127.0.0.1 -P 4000 -uroot --comments`。

11.3.4.2.1 语法

> **注意：**
>
> 如果需要提示优化器使用的表不在 `USE DATABASE` 所指定的数据库内，需要显式指定数据库名。例如：

```
tidb> SELECT /*+ HASH_JOIN(t2, t) */ * FROM t, test2.t2;
Empty set, 1 warning (0.00 sec)

tidb> SHOW WARNINGS;
+--
    ↪ -------+------+-----------------------------------------------------------------------------------------
    ↪
| Level   | Code | Message
    ↪
    ↪ |
+--
    ↪ -------+------+-----------------------------------------------------------------------------------------
    ↪
| Warning | 1815 | There are no matching table names for (t2) in optimizer hint /*+
    ↪  HASH_JOIN(t2, t) */ or /*+ TIDB_HJ(t2, t) */. Maybe you can use the table
    ↪ alias name |
+--
    ↪ -------+------+-----------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)

tidb> SELECT /*+ HASH_JOIN(test2.t2, t) */ * FROM t, test2.t2;
Empty set (0.00 sec)

tidb> SELECT /*+ READ_FROM_STORAGE(TIFLASH[test1.t1,test2.t2]) */ t1.a FROM test1.t
    ↪  t1, test2.t t2 WHERE t1.a = t2.a;
Empty set (0.00 sec)
```

本文档中后续示例演示部分，皆是同一个数据库范围内的表。如果你使用的表不在同一个数据库内，请参照指示显式指定数据库名。

Optimizer Hints 不区分大小写，通过 `/*+ ... */` 注释的形式跟在 SELECT、UPDATE 或 DELETE 关键字的后面。INSERT 关键字后不支持 Optimizer Hints。

多个不同的 Hint 之间需用逗号隔开，例如：

```
SELECT /*+ USE_INDEX(t1, idx1), HASH_AGG(), HASH_JOIN(t1) */ count(*) FROM t t1, t t2 WHERE t1.a
    ↪ = t2.b;
```

可以通过Explain/Explain Analyze 语句的输出，来查看 Optimizer Hints 对查询执行计划的影响。

如果 Optimizer Hints 包含语法错误或不完整，查询语句不会报错，而是按照没有 Optimizer Hints 的情况执行。如果 Hint 不适用于当前语句，TiDB 会返回 Warning，用户可以在查询结束后通过 Show Warnings 命令查看具体信息。

> 注意：
>
> 如果注释不是跟在指定的关键字后，会被当作是普通的 MySQL comment，注释不会生效，且不会上报 warning。

TiDB 目前支持的 Optimizer Hints 根据生效范围的不同可以划分为两类：第一类是在查询块范围生效的 Hint，例如`/*+ HASH_AGG()*/`；第二类是在整个查询范围生效的 Hint，例如`/*+ MEMORY_QUOTA(1024 MB)*/`。

每条语句中每一个查询和子查询都对应着一个不同的查询块，每个查询块有自己对应的名字。以下面这条语句为例：

```
SELECT * FROM (SELECT * FROM t) t1, (SELECT * FROM t) t2;
```

该查询语句有 3 个查询块，最外面一层 SELECT 所在的查询块的名字为 sel_1，两个 SELECT 子查询的名字依次为 sel_2 和 sel_3。其中数字序号根据 SELECT 出现的位置从左到右计数。如果分别用 DELETE 和 UPDATE 查询替代第一个 SELECT 查询，则对应的查询块名字分别为 del_1 和 upd_1。

11.3.4.2.2 查询块范围生效的 Hint

这类 Hint 可以跟在查询语句中任意 SELECT、UPDATE 或 DELETE 关键字的后面。通过在 Hint 中使用查询块名字可以控制 Hint 的生效范围，以及准确标识查询中的每一个表（有可能表的名字或者别名相同），方便明确 Hint 的参数指向。若不显式地在 Hint 中指定查询块，Hint 默认作用于当前查询块。以如下查询为例：

```
SELECT /*+ HASH_JOIN(@sel_1 t1@sel_1, t3) */ * FROM (SELECT t1.a, t1.b FROM t t1, t t2 WHERE t1.a
    ↪  = t2.a) t1, t t3 WHERE t1.b = t3.b;
```

该 Hint 在 sel_1 这个查询块中生效，参数分别为 sel_1 中的 t1 表（sel_2 中也有一个 t1 表）和 t3 表。

如上例所述，在 Hint 中使用查询块名字的方式有两种：第一种是作为 Hint 的第一个参数，与其他参数用空格隔开。除 QB_NAME 外，本节所列的所有 Hint 除自身明确列出的参数外都有一个隐藏的可选参数 @QB_NAME，通过使用这个参数可以指定该 Hint 的生效范围；第二种在 Hint 中使用查询块名字的方式是在参数中的某一个表名后面加 @QB_NAME，用以明确指出该参数是哪个查询块中的表。

> 注意：
>
> Hint 声明的位置必须在指定生效的查询块之中或之前，不能是在之后的查询块中，否则无法生效。

QB_NAME

当查询语句是包含多层嵌套子查询的复杂语句时，识别某个查询块的序号和名字很可能会出错，Hint QB_NAME 可以方便我们使用查询块。QB_NAME 是 Query Block Name 的缩写，用于为某个查询块指定新的名字，同时查询块原本默认的名字依然有效。例如：

```
SELECT /*+ QB_NAME(QB1) */ * FROM (SELECT * FROM t) t1, (SELECT * FROM t) t2;
```

这条 Hint 将最外层 SELECT 查询块的命名为 QB1，此时 QB1 和默认名称 sel_1 对于这个查询块来说都是有效的。

> 注意：
>
> 上述例子中，如果指定的 QB_NAME 为 sel_2，并且不给原本 sel_2 对应的第二个查询块指定新的 QB_NAME，则第二个查询块的默认名字 sel_2 会失效。

MERGE_JOIN(t1_name [, tl_name …])

MERGE_JOIN(t1_name [, tl_name ...]) 提示优化器对指定表使用 Sort Merge Join 算法。这个算法通常会占用更少的内存，但执行时间会更久。当数据量太大，或系统内存不足时，建议尝试使用。例如：

```
SELECT /*+ MERGE_JOIN(t1, t2) */ * FROM t1, t2 WHERE t1.id = t2.id;
```

> 注意：
>
> MERGE_JOIN 的别名是 TIDB_SMJ，在 3.0.x 及之前版本仅支持使用该别名；之后的版本同时支持使用这两种名称，但推荐使用 MERGE_JOIN。

INL_JOIN(t1_name [, tl_name …])

INL_JOIN(t1_name [, tl_name ...]) 提示优化器对指定表使用 Index Nested Loop Join 算法。这个算法可能会在某些场景更快，消耗更少系统资源，有的场景会更慢，消耗更多系统资源。对于外表经过 WHERE 条件过滤后结果集较小（小于 1 万行）的场景，可以尝试使用。例如：

```
SELECT /*+ INL_JOIN(t1, t2) */ * FROM t1, t2 WHERE t1.id = t2.id;
```

INL_JOIN() 中的参数是建立查询计划时内表的候选表，比如 INL_JOIN(t1) 只会考虑使用 t1 作为内表构建查询计划。表如果指定了别名，就只能使用表的别名作为 INL_JOIN() 的参数；如果没有指定别名，则用表的本名作为其参数。比如在 SELECT /*+ INL_JOIN(t1)*/ * FROM t t1, t t2 WHERE t1.a = t2.b; 中，INL_JOIN() 的参数只能使用 t 的别名 t1 或 t2，不能用 t。

> 注意：
>
> INL_JOIN 的别名是 TIDB_INLJ，在 3.0.x 及之前版本仅支持使用该别名；之后的版本同时支持使用这两种名称，但推荐使用 INL_JOIN。

INL_HASH_JOIN

INL_HASH_JOIN(t1_name [, tl_name]) 提示优化器使用 Index Nested Loop Hash Join 算法。该算法与 Index Nested Loop Join 使用条件完全一样，两者的区别是 INL_JOIN 会在连接的内表上建哈希表，而 INL_HASH_JOIN 会在连接的外表上建哈希表，后者对于内存的使用是有固定上限的，而前者使用的内存使用取决于内表匹配到的行数。

HASH_JOIN(t1_name [, tl_name ⋯])

HASH_JOIN(t1_name [, tl_name ...]) 提示优化器对指定表使用 Hash Join 算法。这个算法多线程并发执行，执行速度较快，但会消耗较多内存。例如：

```
SELECT /*+ HASH_JOIN(t1, t2) */ * FROM t1, t2 WHERE t1.id = t2.id;
```

> 注意：
>
> HASH_JOIN 的别名是 TIDB_HJ，在 3.0.x 及之前版本仅支持使用该别名；之后的版本同时支持使用这两种名称，推荐使用 HASH_JOIN。

HASH_JOIN_BUILD(t1_name [, tl_name ⋯])

HASH_JOIN_BUILD(t1_name [, tl_name ...]) 提示优化器对指定表使用 Hash Join 算法，同时将指定表作为 Hash Join 算法的 Build 端，即用指定表来构建哈希表。例如：

```
SELECT /*+ HASH_JOIN_BUILD(t1) */ * FROM t1, t2 WHERE t1.id = t2.id;
```

HASH_JOIN_PROBE(t1_name [, tl_name ⋯])

HASH_JOIN_PROBE(t1_name [, tl_name ...]) 提示优化器对指定表使用 Hash Join 算法，同时将指定表作为 Hash Join 算法的探测（Probe）端，即用指定表作为探测端来执行 Hash Join 算法。例如：

```
SELECT /*+ HASH_JOIN_PROBE(t2) */ * FROM t1, t2 WHERE t1.id = t2.id;
```

SEMI_JOIN_REWRITE()

SEMI_JOIN_REWRITE() 提示优化器将查询语句中的半连接 (Semi Join) 改写为普通的内连接。目前该 Hint 只作用于 EXISTS 子查询。

如果不使用该 Hint 进行改写，Semi Join 在选择 Hash Join 的执行方式时，只能够使用子查询构建哈希表，因此在子查询比外查询结果集大时，执行速度可能会不及预期。Semi Join 在选择 Index Join 的执行方式时，只能够使用外查询作为驱动表，因此在子查询比外查询结果集小时，执行速度可能会不及预期。

在使用了 SEMI_JOIN_REWRITE() 进行改写后，优化器便可以扩大选择范围，选择更好的执行方式。

```
-- 不使用 SEMI_JOIN_REWRITE() 进行改写
EXPLAIN SELECT * FROM t WHERE EXISTS (SELECT 1 FROM t1 WHERE t1.a = t.a);
```

```
+--
    ↪ ---------------------------+---------+-----------+-----------------------+-----------------------------
    ↪
```

```
| id                          | estRows | task      | access object          | operator info
    ↪                                    |
+--
    ↪ --------------------------+---------+-----------+------------------------+-----------------------------
    ↪
| MergeJoin_9                 | 7992.00 | root      |                        | semi join, left
    ↪ key:test.t.a, right key:test.t1.a |
| ├─IndexReader_25(Build)     | 9990.00 | root      |                         | index:
    ↪ IndexFullScan_24                               |
| │ └─IndexFullScan_24        | 9990.00 | cop[tikv] | table:t1, index:idx(a) | keep order:true,
    ↪ stats:pseudo                       |
| └─IndexReader_23(Probe)     | 9990.00 | root      |                        | index:
    ↪ IndexFullScan_22                               |
|   └─IndexFullScan_22        | 9990.00 | cop[tikv] | table:t, index:idx(a)  | keep order:true,
    ↪ stats:pseudo                       |
+--
    ↪ --------------------------+---------+-----------+------------------------+-----------------------------
    ↪
```

```
-- 使用 SEMI_JOIN_REWRITE() 进行改写
EXPLAIN SELECT * FROM t WHERE EXISTS (SELECT /*+ SEMI_JOIN_REWRITE() */ 1 FROM t1 WHERE t1.a = t.
    ↪ a);
```

```
+--
    ↪ --------------------------+---------+-----------+------------------------+-----------------------------
    ↪
| id                          | estRows | task      | access object          | operator info
    ↪
    ↪ |
+--
    ↪ --------------------------+---------+-----------+------------------------+-----------------------------
    ↪
| IndexJoin_16                | 1.25    | root      |                        | inner join, inner
    ↪ :IndexReader_15, outer key:test.t1.a, inner key:test.t.a, equal cond:eq(test.t1.a, test.t
    ↪ .a) |
| ├─StreamAgg_39(Build)       | 1.00    | root      |                         | group by:test.t1
    ↪ .a, funcs:firstrow(test.t1.a)->test.t1.a
    ↪                                                    |
| │ └─IndexReader_34          | 1.00    | root      |                         | index:
    ↪ IndexFullScan_33
    ↪                                                                              |
| │   └─IndexFullScan_33      | 1.00    | cop[tikv] | table:t1, index:idx(a) | keep order:true
    ↪
    ↪ |
| └─IndexReader_15(Probe)     | 1.25    | root      |                         | index:
```

```
    ↪ Selection_14
    ↪
    ↪ |
|   └─Selection_14            | 1.25    | cop[tikv] |                        | not(isnull(test.
    ↪ t.a))
    ↪                                                                                  |
|     └─IndexRangeScan_13     | 1.25    | cop[tikv] | table:t, index:idx(a)  | range: decided
    ↪ by [eq(test.t.a, test.t1.a)], keep order:false, stats:pseudo
    ↪                                  |
+--
    ↪ ---------------------------+---------+-----------+-----------------------+---------------------------
    ↪
```

在上述例子中可以看到，在使用了 Hint 之后，TiDB 可以选择由表 t1 作为驱动表的 IndexJoin 的执行方式。

SHUFFLE_JOIN(t1_name [, tl_name …])

SHUFFLE_JOIN(t1_name [, tl_name ...]) 提示优化器对指定表使用 Shuffle Join 算法，该 Hint 只在 MPP 模式下生效。例如：

```
SELECT /*+ SHUFFLE_JOIN(t1, t2) */ * FROM t1, t2 WHERE t1.id = t2.id;
```

注意：

- 使用该 Hint 前，需要保证当前 TiDB 集群能够支持在查询中使用 TiFlash MPP 模式，具体细节见文档使用 TiFlash MPP 模式。
- 该 Hint 能与HASH_JOIN_BUILD Hint 和HASH_JOIN_PROBE Hint 组合使用，达到控制 Shuffle Join 算法的 Build 端和 Probe 端的作用。

BROADCAST_JOIN(t1_name [, tl_name …])

BROADCAST_JOIN(t1_name [, tl_name ...]) 提示优化器对指定表使用 Broadcast Join 算法，该 Hint 只在 MPP 模式下生效。例如：

```
SELECT /*+ BROADCAST_JOIN(t1, t2) */ * FROM t1, t2 WHERE t1.id = t2.id;
```

注意：

- 使用该 Hint 前，需要保证当前 TiDB 集群能够支持在查询中使用 TiFlash MPP 模式，具体细节见文档使用 TiFlash MPP 模式。
- 该 Hint 能与HASH_JOIN_BUILD Hint 和HASH_JOIN_PROBE Hint 组合使用，达到控制 Broadcast Join 算法的 Build 端和 Probe 端的作用。

NO_DECORRELATE()

NO_DECORRELATE() 提示优化器不要尝试解除指定查询块中对应子查询的关联。该 Hint 适用于包含关联列的 EXISTS、IN、ANY、ALL、SOME 和标量子查询，即关联子查询。

将该 Hint 写在一个查询块中后，对于该子查询和其外部查询块之间的关联列，优化器将不再尝试解除关联，而是始终使用 Apply 算子来执行查询。

默认情况下，TiDB 会尝试对关联子查询解除关联，以达到更高的执行效率。但是在一部分场景下，解除关联反而会降低执行效率。这种情况下，可以使用该 Hint 来人工提示优化器不要进行解除关联操作。例如：

```
create table t1(a int, b int);
create table t2(a int, b int, index idx(b));
```

```
-- 不使用 NO_DECORRELATE()
explain select * from t1 where t1.a < (select sum(t2.a) from t2 where t2.b = t1.b);
```

```
+--
    ↪ --------------------------------+----------+-----------+--------------+--------------------------------
    ↪
| id                                | estRows  | task      | access object | operator info
    ↪
    ↪ |
+--
    ↪ --------------------------------+----------+-----------+--------------+--------------------------------
    ↪
| HashJoin_11                       | 9990.00  | root      |               | inner join, equal:[eq
    ↪ (test.t1.b, test.t2.b)], other cond:lt(cast(test.t1.a, decimal(10,0) BINARY), Column#7) |
| ├─HashAgg_23(Build)               | 7992.00  | root      |               | group by:test.t2.b,
    ↪ funcs:sum(Column#8)->Column#7, funcs:firstrow(test.t2.b)->test.t2.b
    ↪ |
| │ └─TableReader_24                | 7992.00  | root      |               | data:HashAgg_16
    ↪
    ↪ |
| │   └─HashAgg_16                  | 7992.00  | cop[tikv] |               | group by:test.t2.b,
    ↪ funcs:sum(test.t2.a)->Column#8
    ↪ |
| │     └─Selection_22              | 9990.00  | cop[tikv] |               | not(isnull(test.t2.b
    ↪ ))
    ↪ |
| │       └─TableFullScan_21        | 10000.00 | cop[tikv] | table:t2      | keep order:false,
    ↪ stats:pseudo
    ↪                                                                                  |
| └─TableReader_15(Probe)           | 9990.00  | root      |               | data:Selection_14
    ↪
    ↪ |
|   └─Selection_14                  | 9990.00  | cop[tikv] |               | not(isnull(test.t1.b
    ↪ ))
```

```
    ↪ |
|     └─TableFullScan_13          | 10000.00 | cop[tikv] | table:t1      | keep order:false,
    ↪ stats:pseudo
    ↪                                                                                  |
+--
    ↪ --------------------------------+----------+-----------+---------------+-------------------------------
    ↪
```

从以上执行计划中可以发现，优化器自动解除了关联。解除关联之后的执行计划不包含 Apply 算子，取而代之的是子查询和外部查询块之间的 Join 运算，而原本的带有关联列的过滤条件 `t2.b = t1.b` 也变成了一个普通的 join 条件。

```
-- 使用 NO_DECORRELATE()
explain select * from t1 where t1.a < (select /*+ NO_DECORRELATE() */ sum(t2.a) from t2 where t2.
    ↪ b = t1.b);
```

```
+--
    ↪ ----------------------------------------+-----------+-----------+-----------------------+---------------
    ↪
| id                                        | estRows   | task      | access object         |
    ↪ operator info                                                                          |
+--
    ↪ ----------------------------------------+-----------+-----------+-----------------------+---------------
    ↪
| Projection_10                             | 10000.00  | root      |                       |
    ↪ test.t1.a, test.t1.b                                                                   |
| └─Apply_12                                | 10000.00  | root      |                       |
    ↪ CARTESIAN inner join, other cond:lt(cast(test.t1.a, decimal(10,0) BINARY), Column#7) |
|   ├─TableReader_14(Build)                 | 10000.00  | root      |                       |
    ↪ data:TableFullScan_13                                                                  |
|   │ └─TableFullScan_13                    | 10000.00  | cop[tikv] | table:t1              |
    ↪ keep order:false, stats:pseudo                                                         |
|   └─MaxOneRow_15(Probe)                   | 10000.00  | root      |                       |
    ↪                                                                                        |
|     └─StreamAgg_20                        | 10000.00  | root      |                       |
    ↪ funcs:sum(Column#14)->Column#7                                                         |
|       └─Projection_45                     | 100000.00 | root      |                       |
    ↪ cast(test.t2.a, decimal(10,0) BINARY)->Column#14                                       |
|         └─IndexLookUp_44                  | 100000.00 | root      |                       |
    ↪                                                                                        |
|           ├─IndexRangeScan_42(Build)      | 100000.00 | cop[tikv] | table:t2, index:idx(b) |
    ↪ range: decided by [eq(test.t2.b, test.t1.b)], keep order:false, stats:pseudo          |
|           └─TableRowIDScan_43(Probe)      | 100000.00 | cop[tikv] | table:t2              |
    ↪ keep order:false, stats:pseudo                                                         |
+--
    ↪ ----------------------------------------+-----------+-----------+-----------------------+---------------
```

```
    ↪
```

从以上执行计划中可以发现，优化器没有解除关联。执行计划中包含 Apply 算子，而带有关联列的条件 `t2.b = t1.b` 仍然是访问 `t2` 表时的过滤条件。

HASH_AGG()

`HASH_AGG()` 提示优化器对指定查询块中所有聚合函数使用 Hash Aggregation 算法。这个算法多线程并发执行，执行速度较快，但会消耗较多内存。例如：

```
SELECT /*+ HASH_AGG() */ count(*) FROM t1, t2 WHERE t1.a > 10 GROUP BY t1.id;
```

STREAM_AGG()

`STREAM_AGG()` 提示优化器对指定查询块中所有聚合函数使用 Stream Aggregation 算法。这个算法通常会占用更少的内存，但执行时间会更久。数据量太大，或系统内存不足时，建议尝试使用。例如：

```
SELECT /*+ STREAM_AGG() */ count(*) FROM t1, t2 WHERE t1.a > 10 GROUP BY t1.id;
```

MPP_1PHASE_AGG()

`MPP_1PHASE_AGG()` 提示优化器对指定查询块中所有聚合函数使用一阶段聚合算法，该 Hint 只在 MPP 模式下生效。例如：

```
SELECT /*+ MPP_1PHASE_AGG() */ COUNT(*) FROM t1, t2 WHERE t1.a > 10 GROUP BY t1.id;
```

> 注意：
>
> 使用该 Hint 前，需要保证当前 TiDB 集群能够支持在查询中使用 TiFlash MPP 模式，具体细节见文档使用 TiFlash MPP 模式。

MPP_2PHASE_AGG()

`MPP_2PHASE_AGG()` 提示优化器对指定查询块中所有聚合函数使用二阶段聚合算法，该 Hint 只在 MPP 模式下生效。例如：

```
SELECT /*+ MPP_2PHASE_AGG() */ COUNT(*) FROM t1, t2 WHERE t1.a > 10 GROUP BY t1.id;
```

> 注意：
>
> 使用该 Hint 前，需要保证当前 TiDB 集群能够支持在查询中使用 TiFlash MPP 模式，具体细节见文档使用 TiFlash MPP 模式。

USE_INDEX(t1_name, idx1_name [, idx2_name …])

`USE_INDEX(t1_name, idx1_name [, idx2_name ...])` 提示优化器对指定表仅使用给出的索引。

下面例子的效果等价于 `SELECT * FROM t t1 use index(idx1, idx2);`：

```
SELECT /*+ USE_INDEX(t1, idx1, idx2) */ * FROM t1;
```

> **注意：**
>
> 当该 Hint 中只指定表名，不指定索引名时，表示不考虑使用任何索引，而是选择全表扫。

FORCE_INDEX(t1_name, idx1_name [, idx2_name …])

`FORCE_INDEX(t1_name, idx1_name [, idx2_name ...])` 提示优化器对指定表仅使用给出的索引。

`FORCE_INDEX(t1_name, idx1_name [, idx2_name ...])` 的使用方法、作用和 `USE_INDEX(t1_name, idx1_name [, idx2_name ...])` 相同。

以下四个查询语句的效果相同：

```
SELECT /*+ USE_INDEX(t, idx1) */ * FROM t;
SELECT /*+ FORCE_INDEX(t, idx1) */ * FROM t;
SELECT * FROM t use index(idx1);
SELECT * FROM t force index(idx1);
```

IGNORE_INDEX(t1_name, idx1_name [, idx2_name …])

`IGNORE_INDEX(t1_name, idx1_name [, idx2_name ...])` 提示优化器对指定表忽略给出的索引。

下面例子的效果等价于 `SELECT * FROM t t1 ignore index(idx1, idx2);`：

```
SELECT /*+ IGNORE_INDEX(t1, idx1, idx2) */ * FROM t t1;
```

ORDER_INDEX(t1_name, idx1_name [, idx2_name …])

`ORDER_INDEX(t1_name, idx1_name [, idx2_name ...])` 提示优化器对指定表仅使用给出的索引，并且按顺序读取指定的索引。

> **警告：**
>
> 这个 hint 有可能会导致 SQL 语句报错，建议先进行测试。如果测试时发生报错，请移除该 Hint。如果测试时运行正常，则可以继续使用。

此 hint 通常应用在下面这种场景中：

```
CREATE TABLE t(a INT, b INT, key(a), key(b));
EXPLAIN SELECT /*+ ORDER_INDEX(t, a) */ a FROM t ORDER BY a LIMIT 10;
```

```
+--
    ↪ -------------------------+---------+-----------+--------------------+-----------------------------+
    ↪
| id                         | estRows | task      | access object      | operator info
    ↪                  |
+--
    ↪ -------------------------+---------+-----------+--------------------+-----------------------------+
    ↪
| Limit_10                   | 10.00   | root      |                    | offset:0, count:10
    ↪            |
| └─IndexReader_14           | 10.00   | root      |                    | index:Limit_13
    ↪                |
|   └─Limit_13               | 10.00   | cop[tikv] |                    | offset:0, count:10
    ↪            |
|     └─IndexFullScan_12     | 10.00   | cop[tikv] | table:t, index:a(a) | keep order:true,
    ↪ stats:pseudo |
+--
    ↪ -------------------------+---------+-----------+--------------------+-----------------------------+
    ↪
```

优化器对该查询会生成两类计划：`Limit + IndexScan(keep order: true)` 和 `TopN + IndexScan(keep order: false)`，当使用了 `ORDER_INDEX` Hint，优化器会选择前一种按照顺序读取索引的计划。

> 注意：
>
> - 如果查询本身并不需要按顺序读取索引，即在不使用 Hint 的前提下，优化器在任何情况下都不会生成按顺序读取索引的计划。此时，如果指定了 `ORDER_INDEX` Hint，会出现报错 `Can't find a proper physical plan for this query`，此时应考虑移除对应的 `ORDER_INDEX` Hint。
> - 分区表上的索引无法支持按顺序读取，所以不应该对分区表及其相关的索引使用 `ORDER_INDEX` Hint。

NO_ORDER_INDEX(t1_name, idx1_name [, idx2_name …])

`NO_ORDER_INDEX(t1_name, idx1_name [, idx2_name ...])` 提示优化器对指定表仅使用给出的索引，并且不按顺序读取指定的索引。通常应用在下面这种场景中:

以下示例中查询语句的效果等价于 `SELECT * FROM t t1 use index(idx1, idx2);`：

```
CREATE TABLE t(a INT, b INT, key(a), key(b));
EXPLAIN SELECT /*+ NO_ORDER_INDEX(t, a) */ a FROM t ORDER BY a LIMIT 10;
```

```
+--
    ↪ -------------------------+----------+-----------+--------------------+-------------------------------+
    ↪
| id                         | estRows  | task      | access object      | operator info
    ↪                  |
+--
    ↪ -------------------------+----------+-----------+--------------------+-------------------------------+
    ↪
| TopN_7                     | 10.00    | root      |                    | test.t.a, offset:0,
    ↪ count:10   |
| └─IndexReader_14           | 10.00    | root      |                    | index:TopN_13
    ↪                  |
|   └─TopN_13                | 10.00    | cop[tikv] |                    | test.t.a, offset:0,
    ↪ count:10   |
|     └─IndexFullScan_12     | 10000.00 | cop[tikv] | table:t, index:a(a) | keep order:false,
    ↪ stats:pseudo |
+--
    ↪ -------------------------+----------+-----------+--------------------+-------------------------------+
    ↪
```

和 ORDER_INDEX Hint 的示例相同，优化器对该查询会生成两类计划：`Limit + IndexScan(keep order: true)` 和 `TopN + IndexScan(keep order: false)`，当使用了 NO_ORDER_INDEX Hint，优化器会选择后一种不按照顺序读取索引的计划。

AGG_TO_COP()

`AGG_TO_COP()` 提示优化器将指定查询块中的聚合函数下推到 coprocessor。如果优化器没有下推某些适合下推的聚合函数，建议尝试使用。例如：

```
SELECT /*+ AGG_TO_COP() */ sum(t1.a) FROM t t1;
```

LIMIT_TO_COP()

`LIMIT_TO_COP()` 提示优化器将指定查询块中的 `Limit` 和 `TopN` 算子下推到 coprocessor。优化器没有下推 `Limit` 或者 `TopN` 算子时建议尝试使用该提示。例如：

```
SELECT /*+ LIMIT_TO_COP() */ * FROM t WHERE a = 1 AND b > 10 ORDER BY c LIMIT 1;
```

READ_FROM_STORAGE(TIFLASH[t1_name [, tl_name …]], TIKV[t2_name [, tl_name …]])

`READ_FROM_STORAGE(TIFLASH[t1_name [, tl_name ...]], TIKV[t2_name [, tl_name ...]])` 提示优化器从指定的存储引擎来读取指定的表，目前支持的存储引擎参数有 `TIKV` 和 `TIFLASH`。如果为表指定了别名，就只能使用表的别名作为 `READ_FROM_STORAGE()` 的参数；如果没有指定别名，则用表的本名作为其参数。例如：

```
SELECT /*+ READ_FROM_STORAGE(TIFLASH[t1], TIKV[t2]) */ t1.a FROM t t1, t t2 WHERE t1.a = t2.a;
```

USE_INDEX_MERGE(t1_name, idx1_name [, idx2_name …])

`USE_INDEX_MERGE(t1_name, idx1_name [, idx2_name ...])` 提示优化器通过索引合并的方式来访问指定的表。索引合并分为并集型和交集型两种类型，详情参见用 EXPLAIN 查看索引合并的 SQL 执行计划。

若显式地指定索引列表，优化器会尝试在索引列表中选取索引来构建索引合并。若不指定索引列表，优化器会尝试在所有可用的索引中选取索引来构建索引合并。

对于交集型索引合并，索引列表是必选参数。对于并集型索引合并，Hint 中的索引列表为可选参数。示例如下。

```
SELECT /*+ USE_INDEX_MERGE(t1, idx_a, idx_b, idx_c) */ * FROM t1 WHERE t1.a > 10 OR t1.b > 10;
```

当对同一张表有多个 USE_INDEX_MERGE Hint 时，优化器会从这些 Hint 指定的索引列表的并集中尝试选取索引。

> **注意：**
>
> USE_INDEX_MERGE 的参数是索引名，而不是列名。对于主键索引，索引名为 primary。

LEADING(t1_name [, tl_name …])

LEADING(t1_name [, tl_name ...]) 提示优化器在生成多表连接的执行计划时，按照 hint 中表名出现的顺序来确定多表连接的顺序。例如：

```
SELECT /*+ LEADING(t1, t2) */ * FROM t1, t2, t3 WHERE t1.id = t2.id and t2.id = t3.id;
```

在以上多表连接查询语句中，LEADING() 中表出现的顺序决定了优化器将会先对表 t1 和 t2 进行连接，再将结果和表 t3 进行连接。该 hint 比STRAIGHT_JOIN 更为通用。

LEADING hint 在以下情况下会失效：

- 指定了多个 LEADING hint
- LEADING hint 中指定的表名不存在
- LEADING hint 中指定了重复的表名
- 优化器无法按照 LEADING hint 指定的顺序进行表连接
- 已经存在 straight_join() hint
- 查询语句中包含 outer join 且同时指定了包含笛卡尔积的情况
- 和选择 join 算法的 hint（即 MERGE_JOIN、 INL_JOIN、 INL_HASH_JOIN、 HASH_JOIN）同时使用时

当出现了上述失效的情况，会输出 warning 警告。

```
-- 指定了多个 LEADING hint

SELECT /*+ LEADING(t1, t2) LEADING(t3) */ * FROM t1, t2, t3 WHERE t1.id = t2.id and t2.id = t3.id
    ↪ ;

-- 通过执行 `show warnings` 了解具体产生冲突的原因

SHOW WARNINGS;
```

```
+--
    ↪ -------+------+-----------------------------------------------------------------------------------------
    ↪
| Level   | Code | Message
    ↪
    ↪ |
+--
    ↪ -------+------+-----------------------------------------------------------------------------------------
    ↪
| Warning | 1815 | We can only use one leading hint at most, when multiple leading hints are used
    ↪ , all leading hints will be invalid |
+--
    ↪ -------+------+-----------------------------------------------------------------------------------------
    ↪
```

> **注意：**
>
> 如果查询语句中包含了 outer join，你只能在 hint 中指定可以用于交换连接顺序的表。如果 hint 中存在不能用于交换的表，则该 hint 会失效。例如在 `SELECT * FROM t1 LEFT JOIN (t2 JOIN t3 JOIN t4)ON t1.a = t2.a;` 中，如果想要控制 `t2`、`t3`、`t4` 表的连接顺序，那么在使用 `LEADING` hint 时，hint 中不能出现 `t1` 表。

MERGE()

在含有公共表表达式的查询中使用 `MERGE()` hint，可关闭对当前子查询的物化过程，并将内部查询的内联展开到外部查询。该 hint 适用于非递归的公共表表达式查询，在某些场景下，使用该 hint 会比默认分配一块临时空间的语句执行效率更高。例如将外部查询的条件下推或在嵌套的 CTE 查询中：

```
-- 使用 hint 将外部查询条件的谓词下推
WITH CTE AS (SELECT /*+ MERGE() */ * FROM tc WHERE tc.a < 60) SELECT * FROM CTE WHERE CTE.a <18;

-- 在嵌套 CTE 查询中使用该 hint 来指定将某个 CTE 内联展开到外部查询
WITH CTE1 AS (SELECT * FROM t1), CTE2 AS (WITH CTE3 AS (SELECT /*+ MERGE() */ * FROM t2), CTE4 AS
    ↪  (SELECT * FROM t3) SELECT * FROM CTE3, CTE4) SELECT * FROM CTE1, CTE2;
```

> **注意：**
>
> `MERGE()` 只适用于简单的 CTE 查询，在以下情况下无法使用该 hint：
>
> - 递归的 CTE 查询
> - 子查询中有无法进行内联展开的部分，例如聚合算子、窗口函数以及 `DINSTINCT` 等
>
> 当 CTE 引用次数过多时，查询性能可能低于默认的物化方式。

11.3.4.2.3 全局生效的 Hint

全局生效的 Hint 和视图有关，可以使查询中定义的 Hint 能够在视图内部生效。添加这类 Hint 需要两步：先用 QB_NAME Hint 为视图内的查询块命名，再以“视图名 @ 查询块名”的方式加入实际需要的 Hint。

第 1 步：使用 QB_NAME Hint 重命名视图内的查询块

首先使用QB_NAME Hint 重命名视图内部的查询块。其中针对视图的 QB_NAME Hint 的概念与查询块范围生效的 QB_NAME Hint相同，只是在语法上进行了相应的拓展。从 QB_NAME(QB) 拓展为 QB_NAME(QB, 视图名@查询块名 [.视图名@查询块名 .视图名@查询块名 ...])。

> 注意：
>
> @查询块名 与后面紧跟的 .视图名@查询块名 部分之间需要有一个空格，否则 .视图名@查询块名 会被视作前面 @查询块名 的一部分。例如，QB_NAME(v2_1, v2@SEL_1 .@SEL_1) 不能写为 QB_NAME(v2_1, v2@SEL_1.@SEL_1)。

- 对于单个视图、不包含子查询的简单语句，下面以重命名视图 v 的第一个查询块为例：

```
SELECT /* 注释：当前查询块的名字为默认的 @SEL_1 */ * FROM v;
```

对于视图 v 来说，从查询语句开始的首个视图是 v@SEL_1。视图 v 的第一个查询块可以声明为 QB_NAME(v_1, v@SEL_1 .@SEL_1)，也可以省略 @SEL_1 简写成 QB_NAME(v_1, v)：

```
CREATE VIEW v AS SELECT /* 注释：当前查询块的名字为默认的 @SEL_1 */ * FROM t;

-- 使用全局生效的 Hint
SELECT /*+ QB_NAME(v_1, v) USE_INDEX(t@v_1, idx) */ * FROM v;
```

- 对于嵌套视图和包含子查询的复杂语句，下面以重命名视图 v1、v2 的两个查询块为例：

```
SELECT /* 注释：当前查询块的名字为默认的 @SEL_1 */ * FROM v2 JOIN (
    SELECT /* 注释：当前查询块的名字为默认的 @SEL_2 */ * FROM v2) vv;
```

对于第一个视图 v2 来说，从上面的语句开始的首个视图是 v2@SEL_1。对于第二个视图 v2 来说，首个视图表为 v2@SEL_2。下面的查询部分仅考虑第一个视图 v2。

视图 v2 的第一个查询块可以声明为 QB_NAME(v2_1, v2@SEL_1 .@SEL_1)，视图 v2 的第二个查询块可以声明为 QB_NAME(v2_2, v2@SEL_1 .@SEL_2)：

```
CREATE VIEW v2 AS
    SELECT * FROM t JOIN /* 注释：对于视图 v2 来说，当前查询块的名字为默认的 @SEL_1，
        ↪ 因此当前查询块的视图列表是 v2@SEL_1 .@SEL_1 */
    (
        SELECT COUNT(*) FROM t1 JOIN v1 /* 注释：对于视图 v2 来说，当前查询块的名字为默认的
            ↪ @SEL_2，因此当前查询块的视图列表是 v2@SEL_1 .@SEL_2 */
    ) tt;
```

对于视图 v1 来说，从上面的语句开始的首个视图是 v2@SEL_1 .v1@SEL_2。视图 v1 的第一个查询块可以声明为 QB_NAME(v1_1, v2@SEL_1 .v1@SEL_2 .@SEL_1)，视图 v1 的第二个查询块可以声明为 QB_NAME(v1_2, v2@SEL_1 .v1@SEL_2 .@SEL_2)：

```
CREATE VIEW v1 AS SELECT * FROM t JOIN /* 注释：对于视图 v1 来说，当前查询块的名字为默认的@SEL_1，因此当前查询块的视图列表是 v2@SEL_1 .v1@SEL_2 .@SEL_1 */
    (
        SELECT COUNT(*) FROM t1 JOIN t2 /* 注释：对于视图 v1 来说，当前查询块的名字为默认的@SEL_2，因此当前查询块的视图列表是 v2@SEL_1 .v1@SEL_2 .@SEL_2 */
    ) tt;
```

注意：

- 与视图相关的全局生效的 Hint 必须先定义了对应的 QB_NAME Hint 才能使用。
- 使用一个 Hint 来指定视图内的多个表名时，需要保证在同一个 Hint 中出现的表名处于同一个视图的同一个查询块中。
- 对于最外层的查询来说，在定义和视图相关的 QB_NAME Hint 时：
  - 对于 QB_NAME Hint 中视图列表序列的第一项，在不显式声明 @SEL_ 时，默认和定义 QB_NAME Hint 的查询块位置保持一致，即省略 @SEL_ 的查询 SELECT /*+ QB_NAME(qb1, v2)*/ * FROM v2 JOIN (SELECT /*+ QB_NAME(qb2, v2)*/ * FROM v2)vv; 相当于 SELECT /*+ QB_NAME(qb1, v2@SEL_1)*/ * FROM v2 JOIN (SELECT /*+ QB_NAME(qb2, v2@SEL_2)*/ * FROM v2)vv;。
  - 对于 QB_NAME Hint 中视图列表序列第一项之外的其他部分，只有 @SEL_1 可省略。即，如果声明处于当前部分的第一个查询块中，则 @SEL_1 可以省略，否则，不能省略 @SEL_。对于上面的例子：
    - 视图 v2 的第一个查询块可以声明为 QB_NAME(v2_1, v2)
    - 视图 v2 的第二个查询块可以声明为 QB_NAME(v2_2, v2.@SEL_2)
    - 视图 v1 的第一个查询块可以声明为 QB_NAME(v1_1, v2.v1@SEL_2)
    - 视图 v1 的第二个查询块可以声明为 QB_NAME(v1_2, v2.v1@SEL_2 .@SEL_2)

第 2 步：添加实际需要的 Hint

在定义好视图查询块部分的 QB_NAME Hint 后，你可以通过查询块的名字使用查询块范围生效的 Hint，以“视图名 @ 查询块名”的方式加入实际需要的 Hint，使其在视图内部生效。例如：

- 指定视图 v2 中第一个查询块的 MERGE_JOIN() Hint：

```
SELECT /*+ QB_NAME(v2_1, v2) merge_join(t@v2_1) */ * FROM v2;
```

- 指定视图 v2 中第二个查询块的 MERGE_JOIN() 和 STREAM_AGG() Hint：

```
SELECT /*+ QB_NAME(v2_2, v2.@SEL_2) merge_join(t1@v2_2) stream_agg(@v2_2) */ * FROM v2;
```

- 指定视图 v1 中第一个查询块的 HASH_JOIN() Hint：

```
SELECT /*+ QB_NAME(v1_1, v2.v1@SEL_2) hash_join(t@v1_1) */ * FROM v2;
```

- 指定视图 v1 中第二个查询块的 HASH_JOIN() 和 HASH_AGG() Hint：

```
SELECT /*+ QB_NAME(v1_2, v2.v1@SEL_2 .@SEL_2) hash_join(t1@v1_2) hash_agg(@v1_2) */ * FROM
    ↪ v2;
```

11.3.4.2.4 查询范围生效的 Hint

这类 Hint 只能跟在语句中第一个 SELECT、UPDATE 或 DELETE 关键字的后面，等同于在当前这条查询运行时对指定的系统变量进行修改，其优先级高于现有系统变量的值。

> 注意：
>
> 这类 Hint 虽然也有隐藏的可选变量 @QB_NAME，但就算指定了该值，Hint 还是会在整个查询范围生效。

NO_INDEX_MERGE()

NO_INDEX_MERGE() 会关闭优化器的 index merge 功能。

下面的例子不会使用 index merge：

```
SELECT /*+ NO_INDEX_MERGE() */ * FROM t WHERE t.a > 0 or t.b > 0;
```

除了 Hint 外，系统变量 tidb_enable_index_merge 也能决定是否开启该功能。

> 注意：
>
> - NO_INDEX_MERGE 优先级高于 USE_INDEX_MERGE，当这两类 Hint 同时存在时，USE_INDEX_MERGE 不会生效。
> - 当存在子查询时，NO_INDEX_MERGE 放在最外层才能生效。

USE_TOJA(boolean_value)

参数 boolean_value 可以是 TRUE 或者 FALSE。USE_TOJA(TRUE) 会开启优化器尝试将 in (subquery) 条件转换为 join 和 aggregation 的功能。相对地，USE_TOJA(FALSE) 会关闭该功能。

下面的例子会将 in (SELECT t2.a FROM t2)subq 转换为等价的 join 和 aggregation：

```
SELECT /*+ USE_TOJA(TRUE) */ t1.a, t1.b FROM t1 WHERE t1.a in (SELECT t2.a FROM t2) subq;
```

除了 Hint 外，系统变量 tidb_opt_insubq_to_join_and_agg 也能决定是否开启该功能。

MAX_EXECUTION_TIME(N)

MAX_EXECUTION_TIME(N) 把语句的执行时间限制在 N 毫秒以内，超时后服务器会终止这条语句的执行。

下面的 Hint 设置了 1000 毫秒（即 1 秒）超时：

```
SELECT /*+ MAX_EXECUTION_TIME(1000) */ * FROM t1 inner join t2 WHERE t1.id = t2.id;
```

除了 Hint 之外，系统变量 global.max_execution_time 也能对语句执行时间进行限制。

MEMORY_QUOTA(N)

MEMORY_QUOTA(N) 用于限制语句执行时的内存使用。该 Hint 支持 MB 和 GB 两种单位。内存使用超过该限制时会根据当前设置的内存超限行为来打出一条 log 或者终止语句的执行。

下面的 Hint 设置了 1024 MB 的内存限制：

```
SELECT /*+ MEMORY_QUOTA(1024 MB) */ * FROM t;
```

除了 Hint 外，系统变量tidb_mem_quota_query 也能限制语句执行的内存使用。

READ_CONSISTENT_REPLICA()

READ_CONSISTENT_REPLICA() 会开启从数据一致的 TiKV follower 节点读取数据的特性。

下面的例子会从 follower 节点读取数据：

```
SELECT /*+ READ_CONSISTENT_REPLICA() */ * FROM t;
```

除了 Hint 外，环境变量 tidb_replica_read 设为 'follower' 或者 'leader' 也能决定是否开启该特性。

IGNORE_PLAN_CACHE()

IGNORE_PLAN_CACHE() 提示优化器在处理当前 prepare 语句时不使用 plan cache。

该 Hint 用于在Prepared Plan Cache 开启的场景下临时对某类查询禁用 plan cache。

以下示例强制该 prepare 语句不使用 plan cache：

```
prepare stmt FROM 'SELECT  /*+ IGNORE_PLAN_CACHE() */ * FROM t WHERE t.id = ?';
```

STRAIGHT_JOIN()

STRAIGHT_JOIN() 提示优化器在生成表连接顺序时按照表名在 FROM 子句中出现的顺序进行连接。

```
SELECT /*+ STRAIGHT_JOIN() */ * FROM t t1, t t2 WHERE t1.a = t2.a;
```

注意：

- STRAIGHT_JOIN 优先级高于 LEADING，当这两类 Hint 同时存在时，LEADING 不会生效。
- 建议使用 LEADING Hint，它比 STRAIGHT_JOIN Hint 更通用。

NTH_PLAN(N)

NTH_PLAN(N) 提示优化器选用在物理优化阶段搜索到的第 N 个物理计划。N 必须是正整数。

如果指定的 N 超出了物理优化阶段的搜索范围，TiDB 会返回 warning，并根据不存在该 Hint 时一样的策略选择最优物理计划。

该 Hint 在启用 cascades planner 的情况下不会生效。

以下示例会强制优化器在物理阶段选择搜索到的第 3 个物理计划：

```
SELECT /*+ NTH_PLAN(3) */ count(*) from t where a > 5;
```

> 注意：
>
> NTH_PLAN(N) 主要用于测试用途，并且在未来不保证其兼容性，请谨慎使用。

RESOURCE_GROUP(resource_group_name)

RESOURCE_GROUP(resource_group_name) 用于使用资源管控 (Resource Control) 实现资源隔离。此 Hint 将临时使用指定的资源组执行当前的语句。如果指定的资源组不存在，则该 Hint 将被忽略。

示例：

```
SELECT /*+ RESOURCE_GROUP(rg1) */ * FROM t limit 10;
```

11.3.4.3 执行计划管理 (SPM)

执行计划管理，又称 SPM (SQL Plan Management)，是通过执行计划绑定，对执行计划进行人为干预的一系列功能，包括执行计划绑定、自动捕获绑定、自动演进绑定等。

11.3.4.3.1 执行计划绑定 (SQL Binding)

执行计划绑定是 SPM 的基础。在优化器 Hints 中介绍了可以通过 Hint 的方式选择指定的执行计划，但有时需要在不修改 SQL 语句的情况下干预执行计划的选择。执行计划绑定功能使得可以在不修改 SQL 语句的情况下选择指定的执行计划。

> 注意：
>
> 要使用执行计划绑定，你需要拥有 SUPER 权限。如果在使用过程中系统提示权限不足，可参考权限管理补充所需权限。

创建绑定

你可以根据 SQL 或者历史执行计划为指定的 SQL 语句创建绑定。

根据 SQL 创建绑定

```
CREATE [GLOBAL | SESSION] BINDING FOR BindableStmt USING BindableStmt;
```

该语句可以在 GLOBAL 或者 SESSION 作用域内为 SQL 绑定执行计划。目前，如下 SQL 类型 (BindableStmt) 可创建执行计划绑定：SELECT、DELETE、UPDATE 和带有 SELECT 子查询的 INSERT/REPLACE。

> **注意：**
>
> 绑定的优先级高于手工添加的 Hint，即在有绑定的时候执行带有 Hint 的语句时，该语句中控制优化器行为的 Hint 不会生效，但是其他类别的 Hint 仍然能够生效。

其中，有两类特定的语法由于语法冲突不能创建执行计划绑定，例如：

```
-- 类型一：使用 `JOIN` 关键字但不通过 `USING` 关键字指定关联列的笛卡尔积
CREATE GLOBAL BINDING for
    SELECT * FROM t t1 JOIN t t2
USING
    SELECT * FROM t t1 JOIN t t2;

-- 类型二：包含了 `USING` 关键字的 `delete` 语句
CREATE GLOBAL BINDING for
    delete FROM t1 USING t1 JOIN t2 ON t1.a = t2.a
USING
    delete FROM t1 USING t1 JOIN t2 ON t1.a = t2.a;
```

可以通过等价的 SQL 改写绕过这个语法冲突的问题。例如，上述两个例子可以改写为：

```
-- 类型一的第一种改写：为 `JOIN` 关键字添加 `USING` 子句
CREATE GLOBAL BINDING for
    SELECT * FROM t t1 JOIN t t2 USING (a)
USING
    SELECT * FROM t t1 JOIN t t2 USING (a);

-- 类型一的第二种改写：去掉 `JOIN` 关键字
CREATE GLOBAL BINDING for
    SELECT * FROM t t1, t t2
USING
    SELECT * FROM t t1, t t2;

-- 类型二的改写：去掉 `delete` 语句中的 `USING` 关键字
CREATE GLOBAL BINDING for
    delete t1 FROM t1 JOIN t2 ON t1.a = t2.a
USING
    delete t1 FROM t1 JOIN t2 ON t1.a = t2.a;
```

> **注意：**
>
> 在对带 SELECT 子查询的 INSERT/REPLACE 语句创建执行计划绑定时，需要将想要绑定的优化器 Hints 指定在 SELECT 子查询中，而不是 INSERT/REPLACE 关键字后，不然优化器 Hints 不会生效。

例如：

```
-- Hint 能生效的用法
CREATE GLOBAL BINDING for
    INSERT INTO t1 SELECT * FROM t2 WHERE a > 1 AND b = 1
USING
    INSERT INTO t1 SELECT /*+ use_index(@sel_1 t2, a) */ * FROM t2 WHERE a > 1 AND b = 1;

-- Hint 不能生效的用法
CREATE GLOBAL BINDING for
    INSERT INTO t1 SELECT * FROM t2 WHERE a > 1 AND b = 1
USING
    INSERT /*+ use_index(@sel_1 t2, a) */ INTO t1 SELECT * FROM t2 WHERE a > 1 AND b = 1;
```

如果在创建执行计划绑定时不指定作用域，隐式作用域 SESSION 会被使用。TiDB 优化器会将被绑定的 SQL 进行“标准化”处理，然后存储到系统表中。在处理 SQL 查询时，只要“标准化”后的 SQL 和系统表中某个被绑定的 SQL 语句一致，并且系统变量tidb_use_plan_baselines 的值为 on（其默认值为 on），即可使用相应的优化器 Hint。如果存在多个可匹配的执行计划，优化器会从中选择代价最小的一个进行绑定。

标准化：把 SQL 中的常量变成变量参数，对空格和换行符等做标准化处理，并对查询引用到的表显式指定数据库。例如：

```
SELECT * FROM t WHERE a >    1
-- 以上语句标准化后如下：
SELECT * FROM test . t WHERE a > ?
```

> **注意：**
>
> 在进行标准化的时候，被逗号，连接起来的多个常量会被标准化为 ... 而不是 ?。
>
> 例如：
>
> ```
> SELECT * FROM t limit 10
> SELECT * FROM t limit 10, 20
> SELECT * FROM t WHERE a IN (1)
> SELECT * FROM t WHERE a IN (1,2,3)
> -- 以上语句标准化后如下：
> SELECT * FROM test . t limit ?
> SELECT * FROM test . t limit ...
> ```

```
SELECT * FROM test . t WHERE a IN ( ? )
SELECT * FROM test . t WHERE a IN ( ... )
```

因此包含单个常量的 SQL 语句和包含被逗号连接起来多个常量的 SQL 语句，在被绑定时会被 TiDB 视作不同的 SQL 语句，需要分别创建绑定。

值得注意的是，如果一条 SQL 语句在 GLOBAL 和 SESSION 作用域内都有与之绑定的执行计划，因为优化器在遇到 SESSION 绑定时会忽略 GLOBAL 绑定的执行计划，该语句在 SESSION 作用域内绑定的执行计划会屏蔽掉语句在 GLOBAL 作用域内绑定的执行计划。

例如：

```
-- 创建一个 global binding，指定其使用 sort merge join
CREATE GLOBAL BINDING for
    SELECT * FROM t1, t2 WHERE t1.id = t2.id
USING
    SELECT /*+ merge_join(t1, t2) */ * FROM t1, t2 WHERE t1.id = t2.id;

-- 从该 SQL 的执行计划中可以看到其使用了 global binding 中指定的 sort merge join
explain SELECT * FROM t1, t2 WHERE t1.id = t2.id;

-- 创建另一个 session binding，指定其使用 hash join
CREATE BINDING for
    SELECT * FROM t1, t2 WHERE t1.id = t2.id
USING
    SELECT /*+ hash_join(t1, t2) */ * FROM t1, t2 WHERE t1.id = t2.id;

-- 从该 SQL 的执行计划中可以看到其使用了 session binding 中指定的 hash join，而不是 global
    ↪ binding 中指定的 sort merge join
explain SELECT * FROM t1, t2 WHERE t1.id = t2.id;
```

第一个 SELECT 语句在执行时优化器会通过 GLOBAL 作用域内的绑定为其加上 `sm_join(t1, t2)` hint，`explain` 出的执行计划中最上层的节点为 MergeJoin。而第二个 SELECT 语句在执行时优化器则会忽视 GLOBAL 作用域内的绑定而使用 SESSION 作用域内的绑定为该语句加上 `hash_join(t1, t2)` hint，`explain` 出的执行计划中最上层的节点为 HashJoin。

每个标准化的 SQL 只能同时有一个通过 `CREATE BINDING` 创建的绑定。对相同的标准化 SQL 创建多个绑定时，会保留最后一个创建的绑定，之前的所有绑定（创建的和演进出来的）都会被删除。但 session 绑定和 global 绑定仍然允许共存，不受这个逻辑影响。

另外，创建绑定时，TiDB 要求 session 处于某个数据库上下文中，也就是执行过 `use ${database}` 或者客户端连接时指定了数据库。

需要注意的是原始 SQL 和绑定 SQL 在参数化以及去掉 Hint 后文本必须相同，否则创建会失败，例如：

```
CREATE BINDING FOR SELECT * FROM t WHERE a > 1 USING SELECT * FROM t use index(idx) WHERE a > 2;
```

可以创建成功，因为原始 SQL 和绑定 SQL 在参数化以及去掉 Hint 后文本都是 `SELECT * FROM test . t WHERE a > ?`，而

```
CREATE BINDING FOR SELECT * FROM t WHERE a > 1 USING SELECT * FROM t use index(idx) WHERE b > 2;
```

则不可以创建成功，因为原始 SQL 在经过处理后是 `SELECT * FROM test . t WHERE a > ?`，而绑定 SQL 在经过处理后是 `SELECT * FROM test . t WHERE b > ?`。

> **注意：**
>
> 对于 PREPARE/EXECUTE 语句组，或者用二进制协议执行的查询，创建执行计划绑定的对象应当是查询语句本身，而不是 PREPARE/EXECUTE 语句。

根据历史执行计划创建绑定

如需将 SQL 语句的执行计划固定为之前使用过的执行计划，可以使用 `plan_digest` 为该 SQL 语句绑定一个历史的执行计划。相比于使用 SQL 创建绑定的方式，此方式更加简便。

目前，根据历史执行计划创建绑定有一些限制：

- 该功能是根据历史的执行计划生成 hint 而实现的绑定，历史的执行计划来源是Statement Summary Tables，因此在使用此功能之前需开启系统变量`tidb_enable_stmt_summary`。
- 对于带有子查询的查询、访问 TiFlash 的查询、3 张表或更多表进行 Join 的查询，目前还不支持通过历史执行计划进行绑定。

使用方式:

```
CREATE [GLOBAL | SESSION] BINDING FROM HISTORY USING PLAN DIGEST 'plan_digest';
```

该语句使用 `plan_digest` 为 SQL 语句绑定执行计划，在不指定作用域时默认作用域为 SESSION。所创建绑定的适用 SQL、优先级、作用域、生效条件等与根据 SQL 创建绑定相同。

使用此绑定方式时，你需要先从 `statements_summary` 中找到需要绑定的执行计划对应的 `plan_digest`，再通过 `plan_digest` 创建绑定。具体步骤如下：

1. 从 `Statement Summary Tables` 的记录中查找执行计划对应的 `plan_digest`。

   例如:

   ```
   CREATE TABLE t(id INT PRIMARY KEY , a INT, KEY(a));
   SELECT /*+ IGNORE_INDEX(t, a) */ * FROM t WHERE a = 1;
   SELECT * FROM INFORMATION_SCHEMA.STATEMENTS_SUMMARY WHERE QUERY_SAMPLE_TEXT = 'SELECT /*+
       ↪ IGNORE_INDEX(t, a) */ * FROM t WHERE a = 1'\G;
   ```

   以下为 `statements_summary` 部分查询结果：

```
SUMMARY_BEGIN_TIME: 2022-12-01 19:00:00
..........
      DIGEST_TEXT: select * from `t` where `a` = ?
..........
      PLAN_DIGEST: 4e3159169cc63c14b139a4e7d72eae1759875c9a9581f94bb2079aae961189cb
            PLAN:  id                  task        estRows operator info
                ↪                           actRows execution info
                ↪
                ↪ memory      disk
                   TableReader_7       root        10      data:Selection_6
                       ↪                        0       time:4.05ms, loops:1, cop_task: {
                       ↪ num: 1, max: 598.6µs, proc_keys: 0, rpc_num: 2, rpc_time: 609.8
                       ↪ µs, copr_cache_hit_ratio: 0.00, distsql_concurrency: 15}   176
                       ↪ Bytes   N/A
                    └─Selection_6       cop[tikv]   10      eq(test.t.a, 1)
                       ↪                          0       tikv_task:{time:560.8µs, loops
                       ↪ :0}
                       ↪
                       ↪ N/A         N/A
                      └─TableFullScan_5 cop[tikv]   10000   table:t, keep order:false,
                          ↪ stats:pseudo 0       tikv_task:{time:560.8µs, loops:0}
                          ↪
                          ↪ N/A         N/A
      BINARY_PLAN: 6
          ↪ QOYCuQDCg1UYWJsZVJlYWRlcl83Ev8BCgtTZWxlY3Rpb25fNhKOAQoPBSJQRnVsbFNjYW5fNSEBAAAAOA0
          ↪ /
          ↪ QSkAAQHwW4jDQDgCQAJKCwoJCgR0ZXN0EgF0Uh5rZWVwIG9yZGVyOmZhbHNlLCBzdGF0czpwc2V1ZG9qInRpa3ZfdGFzaz
          ↪ ////CQMEAXgJCBD///8
          ↪ BIQFzCDhVQw19BAAkBX0QUg9lcSgBfCAudC5hLCAxKWrmYQAYHOi0gc6hBB1hJAFAAVIQZGF0YTo9GgRaFAW4HDQuMDVtc
          ↪ ////////////8BGAE=
```

可以看到执行计划对应的 plan_digest 为 4e3159169cc63c14b139a4e7d72eae1759875c9a9581f94bb2079aae961189cb ↪ 。

2. 使用 plan_digest 创建绑定。

```
CREATE BINDING FROM HISTORY USING PLAN DIGEST '4
    ↪ e3159169cc63c14b139a4e7d72eae1759875c9a9581f94bb2079aae961189cb';
```

创建完毕后可以查看绑定，验证绑定是否生效。

```
SHOW BINDINGS\G;
```

```
*************************** 1. row ***************************
Original_sql: select * from `test` . `t` where `a` = ?
```

```
    Bind_sql: SELECT /*+ use_index(@`sel_1` `test`.`t` ) ignore_index(`t` `a`)*/ * FROM `test`.`t
        ↪ ` WHERE `a` = 1
       ...........
  Sql_digest: 6909a1bbce5f64ade0a532d7058dd77b6ad5d5068aee22a531304280de48349f
 Plan_digest:
1 row in set (0.01 sec)

ERROR:
No query specified
```

```
SELECT * FROM t WHERE a = 1;
SELECT @@LAST_PLAN_FROM_BINDING;
```

```
+--------------------------+
| @@LAST_PLAN_FROM_BINDING |
+--------------------------+
|                        1 |
+--------------------------+
1 row in set (0.00 sec)
```

删除绑定

你可以根据 SQL 语句或者 sql_digest 删除绑定。

根据 SQL 语句删除绑定

```
DROP [GLOBAL | SESSION] BINDING FOR BindableStmt;
```

该语句可以在 GLOBAL 或者 SESSION 作用域内删除指定的执行计划绑定，在不指定作用域时默认作用域为 SESSION。

一般来说，SESSION 作用域的绑定主要用于测试或在某些特殊情况下使用。若需要集群中所有的 TiDB 进程都生效，则需要使用 GLOBAL 作用域的绑定。SESSION 作用域对 GLOBAL 作用域绑定的屏蔽效果会持续到该 SESSION 结束。

承接上面关于 SESSION 绑定屏蔽 GLOBAL 绑定的例子，继续执行：

```
-- 删除 session 中创建的 binding
DROP session binding for SELECT * FROM t1, t2 WHERE t1.id = t2.id;

-- 重新查看该 SQL 的执行计划
explain SELECT * FROM t1,t2 WHERE t1.id = t2.id;
```

在这里 SESSION 作用域内被删除掉的绑定会屏蔽 GLOBAL 作用域内相应的绑定，优化器不会为 SELECT 语句添加 sm_join(t1, t2) hint，explain 给出的执行计划中最上层节点并不被 hint 固定为 MergeJoin，而是由优化器经过代价估算后自主进行选择。

根据 sql_digest 删除绑定

除了可以根据 SQL 语句删除对应的绑定以外，也可以根据 sql_digest 删除绑定：

```
DROP [GLOBAL | SESSION] BINDING FOR SQL DIGEST 'sql_digest';
```

该语句用于在 GLOBAL 或者 SESSION 作用域内删除 `sql_digest` 对应的的执行计划绑定，在不指定作用域时默认作用域为 SESSION。你可以通过查看绑定语句获取 `sql_digest`。

> **注意：**
>
> 执行 `DROP GLOBAL BINDING` 会删除当前 tidb-server 实例缓存中的绑定，并将系统表中对应行的状态修改为 'deleted'。该语句不会直接删除系统表中的记录，因为其他 tidb-server 实例需要读取系统表中的 'deleted' 状态来删除其缓存中对应的绑定。对于这些系统表中状态为 'deleted' 的记录，后台线程每隔 100 个 `bind-info-lease`（默认值为 `3s`，合计 `300s`）会触发一次对 `update_time` 在 10 个 `bind-info-lease` 以前的绑定（确保所有 tidb-server 实例已经读取过这个 'deleted' 状态并更新完缓存）的回收清除操作。

变更绑定状态

根据 SQL 语句变更绑定状态

```
SET BINDING [ENABLED | DISABLED] FOR BindableStmt;
```

该语句可以在 GLOBAL 作用域内变更指定执行计划的绑定状态，默认作用域为 GLOBAL，该作用域不可更改。

使用时，只能将 `Disabled` 的绑定改为 `Enabled` 状态，或将 `Enabled` 的绑定改为 `Disabled` 状态。如果没有可以改变状态的绑定，则会输出一条内容为 `There are no bindings can be set the status. Please check the SQL text` 的警告。需要注意的是，当绑定被设置成 `Disabled` 状态时，查询语句不会使用该绑定。

根据 `sql_digest` 变更绑定状态

除了可以根据 SQL 语句变更对应的绑定状态以外，也可以根据 `sql_digest` 变更绑定状态：

```
SET BINDING [ENABLED | DISABLED] FOR SQL DIGEST 'sql_digest';
```

使用 `sql_digest` 所能变更的绑定状态和生效情况与根据 SQL 语句变更绑定状态相同。如果没有可以改变状态的绑定，则会输出一条内容为 `can't find any binding for 'sql_digest'` 的警告。

查看绑定

```
SHOW [GLOBAL | SESSION] BINDINGS [ShowLikeOrWhere];
```

该语句会按照绑定更新时间由新到旧的顺序输出 GLOBAL 或者 SESSION 作用域内的执行计划绑定，在不指定作用域时默认作用域为 SESSION。目前 `SHOW BINDINGS` 会输出 11 列，具体如下：

列名	说明
original_sql	参数化后的原始 SQL
bind_sql	带 Hint 的绑定 SQL

列名	说明
default_db	默认数据库名
status	状态，包括 enabled（可用，从 v6.0 开始取代之前版本的 using 状态）、disabled（不可用）、deleted（已删除）、invalid（无效）、rejected（演进时被拒绝）和 pending verify（等待演进验证）
create_time	创建时间
update_time	更新时间
charset	字符集
collation	排序规则
source	创建方式，包括 manual（根据 SQL 创建绑定生成）、history（根据历史执行计划创建绑定生成）、capture（由 TiDB 自动创建生成）和 evolve（由 TiDB 自动演进生成）
sql_digest	归一化后的 SQL 的 digest
plan_digest	执行计划的 digest

排查绑定

绑定的排查通常有两种方式：

- 使用系统变量`last_plan_from_binding`显示上一条执行语句是否采用 binding 的执行计划。

```
-- 创建一个 global binding

CREATE GLOBAL BINDING for
    SELECT * FROM t
USING
    SELECT /*+ USE_INDEX(t, idx_a) */ * FROM t;
```

```
SELECT * FROM t;
SELECT @@[SESSION.]last_plan_from_binding;
```

```
+--------------------------+
| @@last_plan_from_binding |
+--------------------------+
|                        1 |
+--------------------------+
1 row in set (0.00 sec)
```

- 使用 `explain format = 'verbose'` 语句查看 SQL 语句的查询计划。如果 SQL 语句使用了 binding，可以接着执行 `show warnings` 了解该 SQL 语句使用了哪一条 binding。

```
-- 创建一个 global binding

CREATE GLOBAL BINDING for
    SELECT * FROM t
USING
    SELECT /*+ USE_INDEX(t, idx_a) */ * FROM t;

-- 使用 explain format = 'verbose' 语句查看 SQL 的执行计划

explain format = 'verbose' SELECT * FROM t;

-- 通过执行 `show warnings` 了解该 SQL 语句使用了哪一条 binding

show warnings;
```

```
+-------+------+-----------------------------------------------------------------------------+
| Level | Code | Message                                                                     |
+-------+------+-----------------------------------------------------------------------------+
| Note  | 1105 | Using the bindSQL: SELECT /*+ USE_INDEX(`t` `idx_a`)*/ * FROM `test`.`t` |
+-------+------+-----------------------------------------------------------------------------+
1 row in set (0.01 sec)
```

对绑定进行缓存

每个 TiDB 实例都有一个 LRU (Least Recently Used) Cache 对绑定进行缓存，缓存的容量由系统变量 tidb_mem_quota_binding_cache 进行控制。缓存会影响绑定的使用和查看，因此你只能使用和查看存在于缓存中的绑定。

如需查看绑定的使用情况，可以执行 `SHOW binding_cache status` 语句。该语句无法指定作用域，默认作用域为 GLOBAL。该语句可查看缓存中可用绑定的数量、系统中所有可用绑定的数量、缓存中所有绑定的内存使用量及缓存的内存容量。

```
SHOW binding_cache status;
```

```
+-------------------+-------------------+--------------+--------------+
| bindings_in_cache | bindings_in_table | memory_usage | memory_quota |
+-------------------+-------------------+--------------+--------------+
|                 1 |                 1 | 159 Bytes    | 64 MB        |
+-------------------+-------------------+--------------+--------------+
1 row in set (0.00 sec)
```

11.3.4.3.2 自动捕获绑定 (Baseline Capturing)

自动绑定会对符合捕获条件的查询进行捕获，为符合条件的查询生成相应的绑定。通常用于升级时的计划回退防护。

使用方式

通过将 `tidb_capture_plan_baselines` 的值设置为 on（其默认值为 `off`）可以打开自动捕获绑定功能。

> 注意：
>
> 自动绑定功能依赖于Statement Summary，因此在使用自动绑定之前需打开 Statement Summary 开关。

开启自动绑定功能后，每隔 `bind-info-lease`（默认值为 `3s`）会遍历一次 Statement Summary 中的历史 SQL 语句，并为至少出现两次的 SQL 语句自动捕获绑定。绑定的执行计划为 Statement Summary 中记录执行这条语句时使用的执行计划。

对于以下几种 SQL 语句，TiDB 不会自动捕获绑定：

- EXPLAIN 和 EXPLAIN ANALYZE 语句；
- TiDB 内部执行的 SQL 语句，比如统计信息自动加载使用的 SELECT 查询；
- 存在 `Enabled` 或 `Disabled` 状态绑定的语句；
- 满足捕获绑定黑名单过滤条件的语句。

> 注意：
>
> 当前，绑定通过生成一组 Hints 来固定查询语句生成的执行计划，从而确保执行计划不发生变化。对于大多数 OLTP 查询，TiDB 能够保证计划前后一致，如使用相同的索引、相同的 Join 方式（如 HashJoin、IndexJoin）等。但是，受限于当前 Hints 的完善程度，对于一些较为复杂的查询，如两个表以上的 Join 和复杂的 OLAP、MPP 类查询，TiDB 无法保证计划在绑定前后完全一致。

对于 PREPARE/EXECUTE 语句组，或通过二进制协议执行的查询，TiDB 会为真正的查询（而不是 PREPARE/EXECUTE 语句）自动捕获绑定。

> **注意：**
>
> 由于 TiDB 存在一些内嵌 SQL 保证一些功能的正确性，所以自动捕获绑定时会默认屏蔽内嵌 SQL。

过滤捕获绑定

使用本功能，你可以设置黑名单，将满足黑名单规则的查询排除在捕获范围之外。黑名单支持的过滤维度包括表名、频率和用户名。

使用方式

将过滤规则插入到系统表 `mysql.capture_plan_baselines_blacklist` 中，该过滤规则即刻起会在整个集群范围内生效。

```
-- 按照表名进行过滤
INSERT INTO mysql.capture_plan_baselines_blacklist(filter_type, filter_value) VALUES('table', '
    ↪ test.t');

-- 通过通配符来实现按照数据库名和表名进行过滤
INSERT INTO mysql.capture_plan_baselines_blacklist(filter_type, filter_value) VALUES('table', '
    ↪ test.table_*');
INSERT INTO mysql.capture_plan_baselines_blacklist(filter_type, filter_value) VALUES('table', '
    ↪ db_*.table_*');

-- 按照执行频率进行过滤
INSERT INTO mysql.capture_plan_baselines_blacklist(filter_type, filter_value) VALUES('frequency',
    ↪  '2');

-- 按照用户名进行过滤
INSERT INTO mysql.capture_plan_baselines_blacklist(filter_type, filter_value) VALUES('user', '
    ↪ user1');
```

维度名称	说明	注意事项
table	按照表名进行过滤，每个过滤规则均采用 `db.table` 形式，支持通配符。详细规则可以参考直接使用表名和使用通配符。	字母大小写不敏感，如果包含非法内容，日志会输出 `[sql-bind] failed to load mysql ↪ .capture_plan_baselines_blacklist` 警告。
frequency	按照频率进行过滤，默认捕获执行超过一次的语句。可以设置较大值来捕获执行频繁的语句。	插入的值小于 1 会被认为是非法值，同时，日志会输出 `[sql-bind] frequency ↪ threshold is less than 1, ignore it` 警告。如果插入了多条频率过滤规则，频率最大的值会被用作过滤条件。

维度名称	说明	注意事项
user	按照用户名进行过滤，黑名单用户名执行的语句不会被捕获。	如果多个用户执行同一条语句，只有当他们的用户名都在黑名单的时候，该语句才不会被捕获。

> 注意：
>
> - 修改黑名单需要数据库的 super privilege 权限。
> - 如果黑名单包含了非法的过滤内容时，TiDB 会在日志中输出 `[sql-bind] unknown capture filter type, ignore it` 进行提示。

升级时的计划回退防护

当需要升级 TiDB 集群时，你可以利用自动捕获绑定对潜在的计划回退风险进行一定程度的防护，具体流程为：

1. 升级前打开自动捕获。

> 注意：
> 经测试，长期打开自动捕获对集群负载的性能影响很小。尽量长期打开自动捕获，以确保重要的查询（出现过两次及以上）都能被捕获到。

2. 进行 TiDB 集群的升级。在升级完成后，这些通过捕获的绑定会发挥作用，确保在升级后，查询的计划不会改变。
3. 升级完成后，根据情况手动删除绑定。
    - 通过`SHOW GLOBAL BINDINGS`语句检查绑定来源：
      根据输出中的 `Source` 字段对绑定的来源进行区分，确认是通过捕获 (`capture`) 生成还是通过手动创建 (`manual`) 生成。
    - 确定 `capture` 的绑定是否需要保留：

```
-- 查看绑定生效时的计划
SET @@SESSION.TIDB_USE_PLAN_BASELINES = true;
EXPLAIN FORMAT='VERBOSE' SELECT * FROM t1 WHERE ...;

-- 查看绑定不生效时的计划
SET @@SESSION.TIDB_USE_PLAN_BASELINES = false;
EXPLAIN FORMAT='VERBOSE' SELECT * FROM t1 WHERE ...;
```

        - 如果屏蔽绑定前后，查询得到的计划一致，则可以安全删除此绑定。
        - 如果计划不一样，则可能需要对此计划变化的原因进行排查，如检查统计信息等操作。在这种情况下需要保留此绑定，确保计划不发生变化。

11.3.4.3.3 自动演进绑定 (Baseline Evolution)

自动演进绑定，在 TiDB 4.0 版本引入，是执行计划管理的重要功能之一。

由于某些数据变更后，原先绑定的执行计划可能是一个不优的计划。为了解决该问题，引入自动演进绑定功能来自动优化已经绑定的执行计划。

另外自动演进绑定还可以一定程度上避免统计信息改动后，对执行计划带来的抖动。

使用方式

通过以下语句可以开启自动演进绑定功能：

```
SET GLOBAL tidb_evolve_plan_baselines = ON;
```

`tidb_evolve_plan_baselines` 的默认值为 `off`。

> 警告：
>
> - 自动演进功能目前为实验特性，存在未知风险，不建议在生产环境中使用。
> - 此变量开关已强制关闭，直到自动演进成为正式功能 GA (Generally Available)。如果你尝试打开开关，会产生报错。如果你已经在生产环境中使用了此功能，请尽快将它禁用。如发现 binding 状态不如预期，请从 PingCAP 官方或 TiDB 社区获取支持。

在打开自动演进功能后，如果优化器选出的最优执行计划不在之前绑定的执行计划之中，会将其记录为待验证的执行计划。每隔 `bind-info-lease`（默认值为 `3s`），会选出一个待验证的执行计划，将其和已经绑定的执行计划中代价最小的比较实际运行时间。如果待验证的运行时间更优的话（目前判断标准是运行时间小于等于已绑定执行计划运行时间的 2/3），会将其标记为可使用的绑定。以下示例描述上述过程。

假如有表 `t` 定义如下：

```
CREATE TABLE t(a INT, b INT, KEY(a), KEY(b));
```

在表 `t` 上进行如下查询：

```
SELECT * FROM t WHERE a < 100 AND b < 100;
```

表上满足条件 `a < 100` 的行很少。但由于某些原因，优化器没能选中使用索引 `a` 这个最优执行计划，而是误选了速度慢的全表扫，那么用户首先可以通过如下语句创建一个绑定：

```
CREATE GLOBAL BINDING for SELECT * FROM t WHERE a < 100 AND b < 100 USING SELECT * FROM t use
    ↪ index(a) WHERE a < 100 AND b < 100;
```

当以上查询语句再次执行时，优化器会在刚创建绑定的干预下选择使用索引 `a`，进而降低查询时间。

假如随着在表中进行插入和修改，表中满足条件 `a < 100` 的行变得越来越多，而满足条件 `b < 100` 的行变得越来越少，这时再在绑定的干预下使用索引 `a` 可能就不是最优了。

绑定的演进可以解决这类问题。当优化器感知到表数据变化后，会对这条查询生成使用索引 `b` 的执行计划。但由于绑定的存在，这个执行计划不会被采纳和执行，不过它会被存在后台的演进列表里。在演进过程中，

如果它被验证为执行时间明显低于使用索引 a 的执行时间（即当前绑定的执行计划），那么索引 b 会被加入到可用的绑定列表中。在此之后，当这条查询再次被执行时，优化器首先生成使用索引 b 的执行计划，并确认它在绑定列表中，所以会采纳它并执行，进而可以在数据变化后降低这条查询的执行时间。

为了减少自动演进对集群的影响，可以通过设置 `tidb_evolve_plan_task_max_time` 来限制每个执行计划运行的最长时间，其默认值为 600s。实际在验证执行计划时，计划的最长运行时间还会被限制为不超过已验证执行计划的运行时间的两倍；通过 `tidb_evolve_plan_task_start_time` 和 `tidb_evolve_plan_task_end_time` 可以限制运行演进任务的时间窗口，默认值分别为 `00:00 +0000` 和 `23:59 +0000`。

注意事项

由于自动演进绑定会自动地创建新的绑定，当查询的环境发生变动时，自动创建的绑定可能会有多种行为的选择。这里列出一些注意事项：

- 自动演进只会对存在至少一个 global 绑定的标准化 SQL 进行演进。
- 由于创建新的绑定会删除之前所有绑定（对于一条标准化 SQL），自动演进的绑定也会在手动重新创建绑定后被删除。
- 所有和计算过程相关的 hint，在演进时都会被保留。计算过程相关的 hint 有如下几种：

Hint	说明
memory_quota	查询过程最多可以使用多少内存
use_toja	优化器是否考虑把子查询转化为 join
use_cascades	是否使用 cascades 优化器
no_index_merge	优化器是否考虑将 index merge 作为一个读表选项
read_consistent_replica	是否强制读表时使用 follower read
max_execution_time	查询过程最多消耗多少时间

- `read_from_storage` 是一个非常特别的 hint，因为它指定了读表时选择从 TiKV 读还是从 TiFlash 读。由于 TiDB 提供隔离读的功能，当隔离条件变化时，这个 hint 对演进出来的执行计划影响很大，所以当最初创建的绑定中存在这个 hint，TiDB 会无视其所有演进的绑定。

11.3.4.3.4 升级检查 (Upgrade Checklist)

执行计划管理功能 (SPM) 在版本升级过程中可能会出现一些兼容性问题导致升级失败，你需要在版本升级前做一些检查，确保版本顺利升级。

- 当你尝试从 v5.2 以前的版本（即 v4.0、v5.0、v5.1）升级到当前版本，需要注意在升级前检查自动演进的开关 `tidb_evolve_plan_baselines` 是否已经关闭。如果尚未关闭，则需要将其关闭后再进行升级。具体操作如下所示：

```
-- 在待升级的版本上检查自动演进的开关 `tidb_evolve_plan_baselines` 是否关闭。

SELECT @@global.tidb_evolve_plan_baselines;

-- 如果演进的开关 `tidb_evolve_plan_baselines` 尚未关闭，则需要将其关闭。
```

```
set global tidb_evolve_plan_baselines = off;
```

- 当你尝试从 v4.0 版本升级到当前版本，需要注意在升级前检查所有可用绑定对应的查询语句在新版本中是否存在语法错误。如果存在语法错误，则需要删除对应的绑定。

  具体操作如下所示：

```
-- 在待升级的版本上检查现有可用绑定对应的查询语句。

SELECT bind_sql FROM mysql.bind_info WHERE status = 'using';

-- 将上一条查询得到的结果，在新版本的测试环境中进行验证。

bind_sql_0;
bind_sql_1;
...

-- 如果报错信息是语法错误（ERROR 1064 (42000): You have an error in your SQL syntax），
    ↪ 则需要删除对应的绑定。
-- 如果是其他错误，如未找到表，则表示语法兼容，不需要进行额外的处理。
```

11.3.4.4 优化规则与表达式下推的黑名单

本文主要介绍优化规则的黑名单与表达式下推的黑名单。

11.3.4.4.1 优化规则黑名单

优化规则黑名单是针对优化规则的调优手段之一，主要用于手动禁用一些优化规则。

重要的优化规则

优化规则	规则名称	简介
列裁剪	column_prune	对于上层算子不需要的列，不在下层算子输出该列，减少计算
子查询去关联	decorrelate	尝试对相关子查询进行改写，将其转换为普通 join 或 aggregation 计算
聚合消除	aggregation_eliminate	尝试消除执行计划中的某些不必要的聚合算子
投影消除	projection_eliminate	消除执行计划中不必要的投影算子
最大最小消除	max_min_eliminate	改写聚合中的 max/min 计算，转化为 `order by + limit 1`
谓词下推	predicate_push_down	尝试将执行计划中过滤条件下推到离数据源更近的算子上
外连接消除	outer_join_eliminate	尝试消除执行计划中不必要的 left join 或者 right join
分区裁剪	partition_processor	将分区表查询改成为用 union all，并裁剪掉不满足过滤条件的分区
聚合下推	aggregation_push_down	尝试将执行计划中的聚合算子下推到更底层的计算节点
TopN 下推	topn_push_down	尝试将执行计划中的 TopN 算子下推到离数据源更近的算子上
Join 重排序	join_reorder	对多表 join 确定连接顺序
从窗口函数中推导 TopN 或 Limit	derive_topn_from_window	从窗口函数中推导出 TopN 或者 Limit

禁用优化规则

当某些优化规则在一些特殊查询中的优化结果不理想时，可以使用优化规则黑名单禁用一些优化规则。

使用方法

> 注意：
>
> 以下操作都需要数据库的 super privilege 权限。每个优化规则都有各自的名字，比如列裁剪的名字是 “column_prune”。所有优化规则的名字都可以在重要的优化规则表格中第二列查到。

- 如果你想禁用某些规则，可以在 `mysql.opt_rule_blacklist` 表中写入规则的名字，例如：

```
INSERT INTO mysql.opt_rule_blacklist VALUES("join_reorder"), ("topn_push_down");
```

执行以下 SQL 语句可让禁用规则立即生效，包括相应 TiDB Server 的所有旧链接：

```
ADMIN reload opt_rule_blacklist;
```

> 注意：
>
> `admin reload opt_rule_blacklist` 只对执行该 SQL 语句的 TiDB server 生效。若需要集群中所有 TiDB server 生效，需要在每台 TiDB server 上执行该 SQL 语句。

- 需要解除一条规则的禁用时，需要删除表中禁用该条规则的相应数据，再执行 `admin reload`：

```
DELETE FROM mysql.opt_rule_blacklist WHERE name IN ("join_reorder", "topn_push_down");
admin reload opt_rule_blacklist;
```

11.3.4.4.2 表达式下推黑名单

表达式下推黑名单是针对表达式下推的调优手段之一，主要用于对于某些存储类型手动禁用一些表达式。

已支持下推的表达式

目前已经支持下推的表达式信息，请参考表达式列表。

禁止特定表达式下推

当函数的计算过程由于下推而出现异常时，可通过黑名单功能禁止其下推来快速恢复业务。具体而言，你可以将上述支持的函数或运算符名加入黑名单 `mysql.expr_pushdown_blacklist` 中，以禁止特定表达式下推。

`mysql.expr_pushdown_blacklist` 的 schema 如下：

```
DESC mysql.expr_pushdown_blacklist;
```

```
+------------+--------------+------+------+-------------------+-------+
| Field      | Type         | Null | Key  | Default           | Extra |
+------------+--------------+------+------+-------------------+-------+
```

```
| name       | char(100)    | NO   |      | NULL              |       |
| store_type | char(100)    | NO   |      | tikv,tiflash,tidb |       |
| reason     | varchar(200) | YES  |      | NULL              |       |
+------------+--------------+------+------+-------------------+-------+
3 rows in set (0.00 sec)
```

以上结果字段解释如下：

- `name`：禁止下推的函数名。
- `store_type`：用于指明希望禁止该函数下推到哪些组件进行计算。组件可选 `tidb`、`tikv` 和 `tiflash`。`store_type` 不区分大小写，如果需要禁止向多个存储引擎下推，各个存储之间需用逗号隔开。
  - `store_type` 为 `tidb` 时表示在读取 TiDB 内存表时，是否允许该函数在其他 TiDB Server 上执行。
  - `store_type` 为 `tikv` 时表示是否允许该函数在 TiKV Server 的 Coprocessor 模块中执行。
  - `store_type` 为 `tiflash` 时表示是否允许该函数在 TiFlash Server 的 Coprocessor 模块中执行。
- `reason`：用于记录该函数被加入黑名单的原因。

使用方法

加入黑名单

如果要将一个或多个函数或运算符加入黑名单，执行以下步骤：

1. 向 `mysql.expr_pushdown_blacklist` 插入对应的函数名或运算符名以及希望禁止下推的存储引擎集合。
2. 执行 `admin reload expr_pushdown_blacklist;`。

移出黑名单

如果要将一个或多个函数及运算符移出黑名单，执行以下步骤：

1. 从 `mysql.expr_pushdown_blacklist` 表中删除对应的函数名或运算符名。
2. 执行 `admin reload expr_pushdown_blacklist;`。

> **注意：**
>
> `admin reload expr_pushdown_blacklist` 只对执行该 SQL 语句的 TiDB server 生效。若需要集群中所有 TiDB server 生效，需要在每台 TiDB server 上执行该 SQL 语句。

表达式黑名单用法示例

以下示例首先将运算符 < 及 > 加入黑名单，然后将运算符 > 从黑名单中移出。

黑名单是否生效可以从 `explain` 结果中进行观察（参见TiDB 执行计划概览）。

1. 对于以下 SQL 语句，`where` 条件中的 `a < 2` 和 `a > 2` 可以下推到 TiKV 进行计算。

```
EXPLAIN SELECT * FROM t WHERE a < 2 AND a > 2;
```

```
+--
    ↪ ----------------------+----------+-----------+--------------+--------------------------------+
    ↪
| id                      | estRows  | task      | access object | operator info
    ↪                       |
+--
    ↪ ----------------------+----------+-----------+--------------+--------------------------------+
    ↪
| TableReader_7           | 0.00     | root      |               | data:Selection_6
    ↪                     |
| └─Selection_6           | 0.00     | cop[tikv] |               | gt(ssb_1.t.a, 2), lt(
    ↪ ssb_1.t.a, 2) |
|   └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo      |
+--
    ↪ ----------------------+----------+-----------+--------------+--------------------------------+
    ↪
3 rows in set (0.00 sec)
```

2. 往 `mysql.expr_pushdown_blacklist` 表中插入禁用表达式，并且执行 `admin reload expr_pushdown_blacklist` ↪ 。

```
INSERT INTO mysql.expr_pushdown_blacklist VALUES('<','tikv',''), ('>','tikv','');
```

```
Query OK, 2 rows affected (0.01 sec)
Records: 2  Duplicates: 0  Warnings: 0
```

```
ADMIN reload expr_pushdown_blacklist;
```

```
Query OK, 0 rows affected (0.00 sec)
```

3. 重新观察执行计划，发现表达式下推黑名单生效，`where` 条件中的 `<` 和 `>` 没有被下推到 TiKV Coprocessor 上。

```
EXPLAIN SELECT * FROM t WHERE a < 2 and a > 2;
```

```
+--
    ↪ ----------------------+----------+-----------+--------------+--------------------------------+
    ↪
| id                      | estRows  | task      | access object | operator info
    ↪                       |
+--
    ↪ ----------------------+----------+-----------+--------------+--------------------------------+
    ↪
```

```
| Selection_7             | 10000.00 | root      |               | gt(ssb_1.t.a, 2), lt(
    ↪ ssb_1.t.a, 2) |
| └─TableReader_6         | 10000.00 | root      |               | data:TableFullScan_5
    ↪                |
|   └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo      |
+--
    ↪ ----------------------+----------+-----------+--------------+--------------------------------+
    ↪
3 rows in set (0.00 sec)
```

4. 将某一表达式（>大于）禁用规则从黑名单表中删除，并且执行 `admin reload expr_pushdown_blacklist` ↪ 。

```
DELETE FROM mysql.expr_pushdown_blacklist WHERE name = '>';
```

```
Query OK, 1 row affected (0.01 sec)
```

```
ADMIN reload expr_pushdown_blacklist;
```

```
Query OK, 0 rows affected (0.00 sec)
```

5. 重新观察执行计划，可以看到只有 > 表达式被重新下推到 TiKV Coprocessor，< 表达式仍然被禁用下推。

```
EXPLAIN SELECT * FROM t WHERE a < 2 AND a > 2;
```

```
+--
    ↪ -------------------------+----------+-----------+---------------+--------------------------------+
    ↪
| id                        | estRows  | task      | access object | operator info
    ↪                  |
+--
    ↪ -------------------------+----------+-----------+---------------+--------------------------------+
    ↪
| Selection_8               | 0.00     | root      |               | lt(ssb_1.t.a, 2)
    ↪               |
| └─TableReader_7           | 0.00     | root      |               | data:Selection_6
    ↪               |
|   └─Selection_6           | 0.00     | cop[tikv] |               | gt(ssb_1.t.a, 2)
    ↪               |
|     └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t       | keep order:false,
    ↪ stats:pseudo |
+--
    ↪ -------------------------+----------+-----------+---------------+--------------------------------+
    ↪
4 rows in set (0.00 sec)
```

12 教程

12.1 单区域多 AZ 部署 TiDB

作为一栈式实时 HTAP 数据库，TiDB 兼顾了传统关系型数据库的优秀特性、NoSQL 数据库可扩展性以及跨可用区 (Availability Zone, AZ) 场景下的高可用。本文档旨在介绍同区域多 AZ 部署 TiDB 的方案。

本文中的区域指的是地理隔离的不同位置，AZ 指的是区域内部划分的相互独立的资源集合。本文描述的方案同样适用于一个城市内多个数据中心（同城多中心）的场景。

12.1.1 了解 Raft 协议

Raft 是一种分布式一致性算法，在 TiDB 集群的多种组件中，PD 和 TiKV 都通过 Raft 实现了数据的容灾。Raft 的灾难恢复能力通过如下机制实现：

- Raft 成员的本质是日志复制和状态机。Raft 成员之间通过复制日志来实现数据同步；Raft 成员在不同条件下切换自己的成员状态，其目标是选出 leader 以提供对外服务。
- Raft 是一个表决系统，它遵循多数派协议，在一个 Raft Group 中，某成员获得大多数投票，它的成员状态就会转变为 leader。也就是说，当一个 Raft Group 还保有大多数节点 (majority) 时，它就能够选出 leader 以提供对外服务。

遵循 Raft 可靠性的特点，放到现实场景中：

- 想克服任意 1 台服务器 (host) 的故障，应至少提供 3 台服务器。
- 想克服任意 1 个机柜 (rack) 的故障，应至少提供 3 个机柜。
- 想克服任意 1 个可用区（AZ，也可以是同城的多个机房）的故障，应至少提供 3 个 AZ。
- 想应对任意 1 个区域的灾难场景，应至少规划 3 个区域用于部署集群。

可见，原生 Raft 协议对于偶数副本的支持并不是很友好，考虑跨区域网络延迟影响，同区域三 AZ 可能是最适合部署 Raft 的高可用及容灾方案。

12.1.2 同区域三 AZ 方案

同区域三 AZ 方案，即同区域有三个机房部署 TiDB 集群，AZ 间的数据在集群内部（通过 Raft 协议）进行同步。同区域三 AZ 可同时对外进行读写服务，任意中心发生故障不影响数据一致性。

12.1.2.1 简易架构图

集群 TiDB、TiKV 和 PD 组件分别部署在 3 个不同的 AZ，这是最常规且高可用性最高的方案。

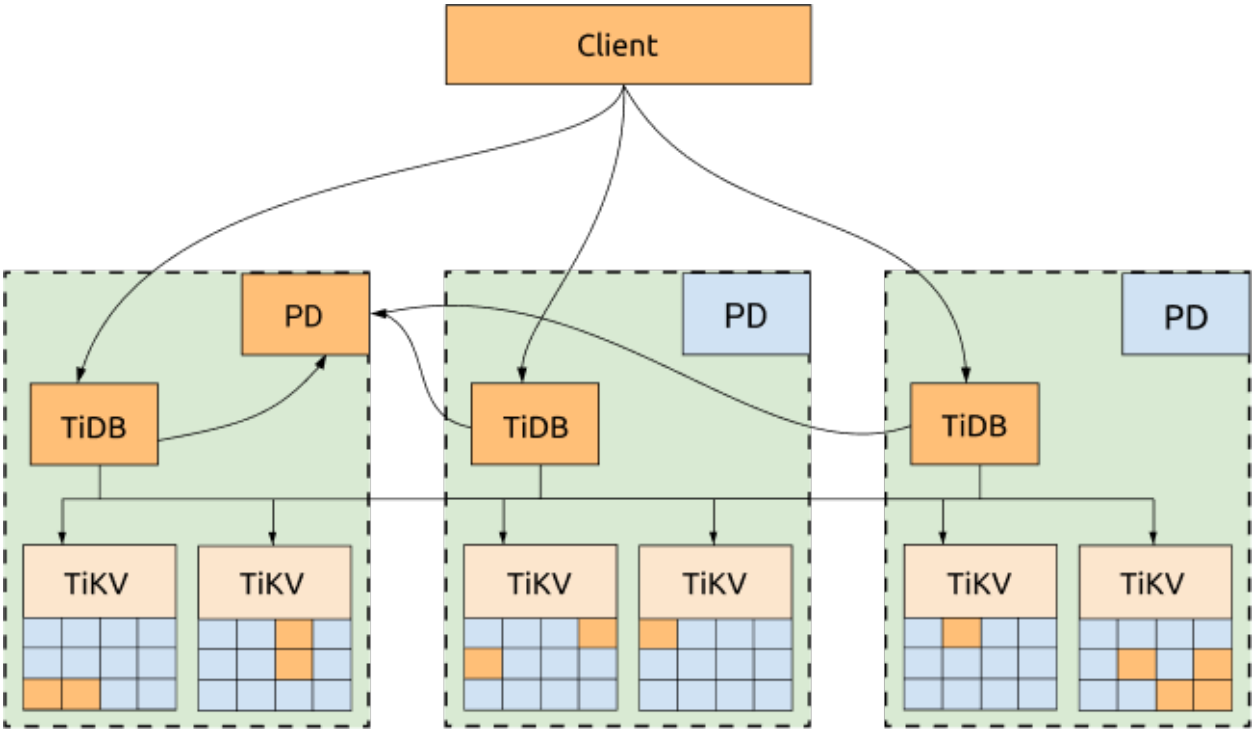


图 175: 三 AZ 部署

优点：

- 所有数据的副本分布在三个 AZ，具备高可用和容灾能力
- 任何一个 AZ 失效后，不会产生任何数据丢失 (RPO = 0)
- 任何一个 AZ 失效后，其他两个 AZ 会自动发起 leader election，并在一定时间内（通常 20s 以内）自动恢复服务

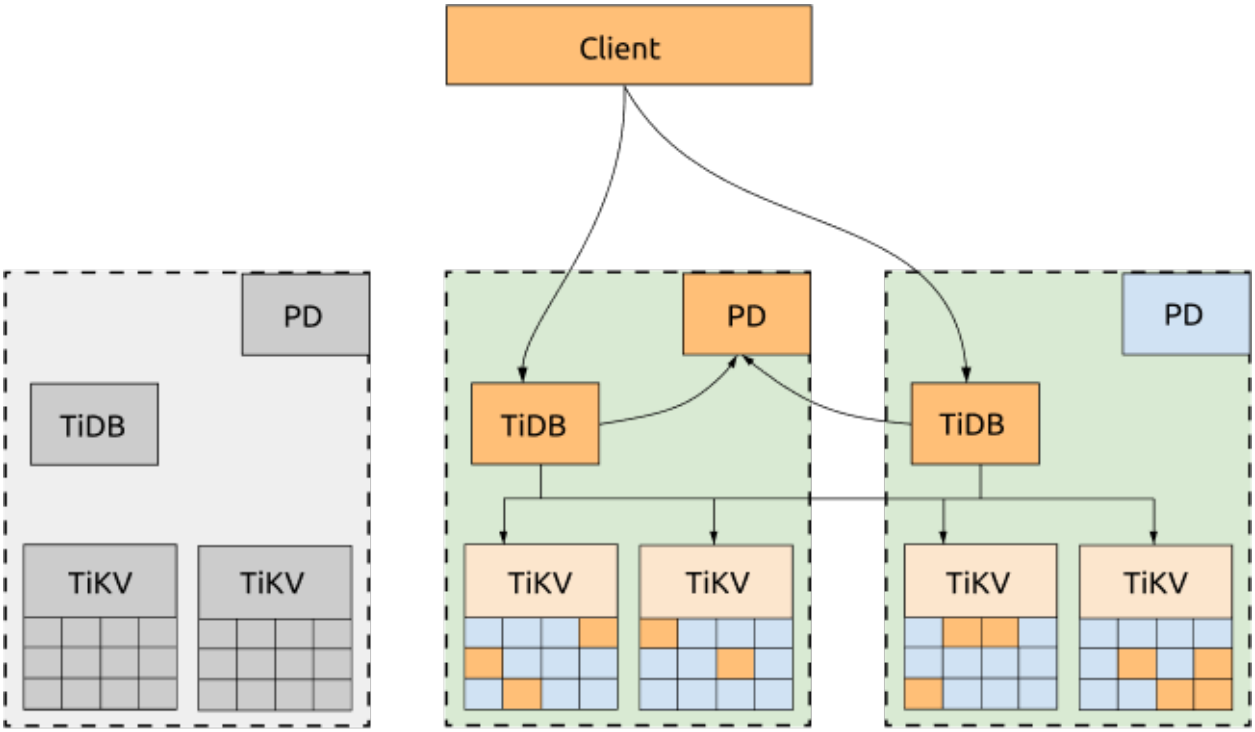


图 176: 三 AZ 部署容灾

缺点：

性能受网络延迟影响。具体影响如下：

- 对于写入的场景，所有写入的数据需要同步复制到至少两个 AZ，由于 TiDB 写入过程使用两阶段提交，故写入延迟至少需要两倍 AZ 间的延迟。
- 对于读请求来说，如果数据 leader 与发起读取的 TiDB 节点不在同一个 AZ，也会受网络延迟影响。
- TiDB 中的每个事务都需要向 PD leader 获取 TSO，当 TiDB 与 PD leader 不在同一个 AZ 时，TiDB 上运行的事务也会因此受网络延迟影响，每个有写入的事务会获取两次 TSO。

12.1.2.2 架构优化图

如果不需要每个 AZ 同时对外提供服务，可以将业务流量全部派发到一个 AZ，并通过调度策略把 Region leader 和 PD leader 都迁移到同一个 AZ。这样，不管是从 PD 获取 TSO，还是读取 Region，都不会受 AZ 间网络的影响。当该 AZ 失效时，PD leader 和 Region leader 会自动在其它 AZ 选出，只需要把业务流量转移至其他存活的 AZ 即可。

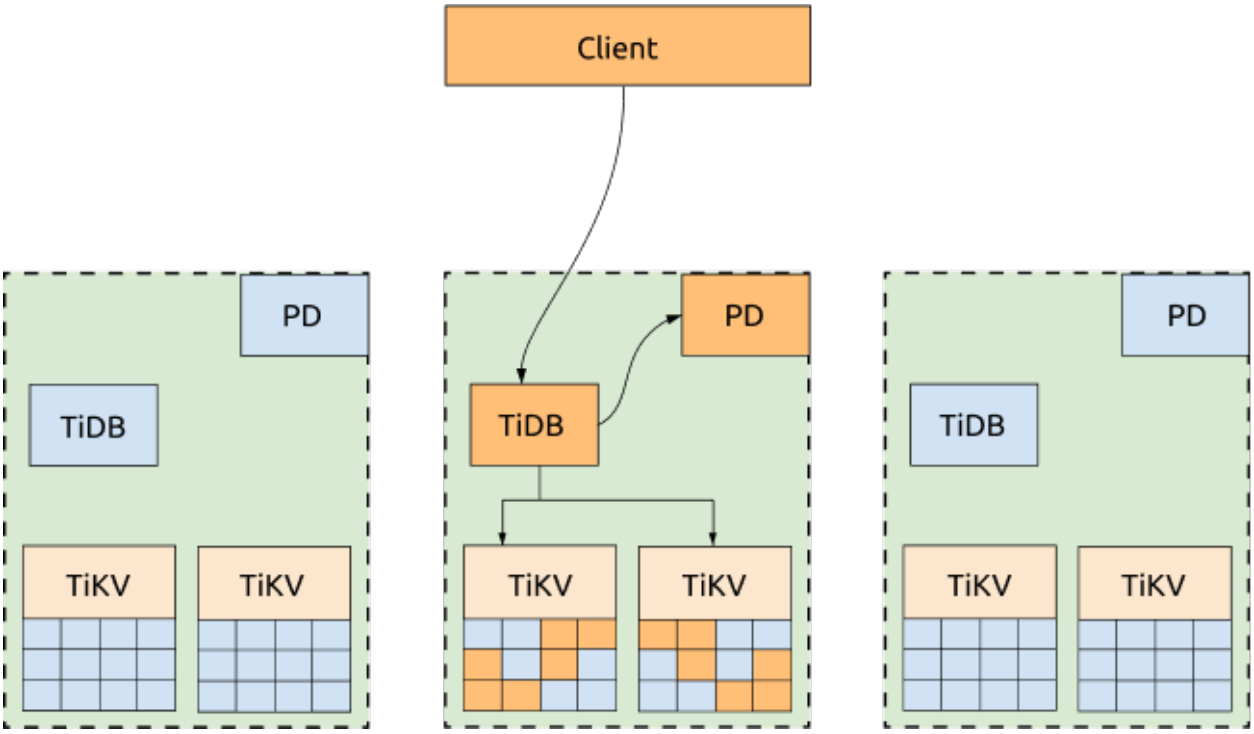


图 177: 三 AZ 部署读性能优化

优点：

集群 TSO 获取能力以及读取性能有所提升。具体调度策略设置模板参照如下：

```
-- 其他 AZ 将 leader 驱逐至承载业务流量的 AZ

config set label-property reject-leader LabelName labelValue

-- 迁移 PD leader 并设置优先级
member leader transfer pdName1
member leader_priority pdName1 5
member leader_priority pdName2 4
member leader_priority pdName3 3
```

> 注意：
>
> TiDB 5.2 及以上版本默认不支持 `label-property` 配置。若要设置副本策略，请使用Placement Rules。

缺点：

- 写入场景仍受 AZ 网络延迟影响，这是因为遵循 Raft 多数派协议，所有写入的数据需要同步复制到至少两个 AZ
- TiDB Server 是 AZ 级别单点
- 业务流量纯走单 AZ，性能受限于单 AZ 网络带宽压力
- TSO 获取能力以及读取性能受限于业务流量 AZ 集群 PD、TiKV 组件是否正常，否则仍受跨 AZ 网络交互影响

12.1.2.3 样例部署图

12.1.2.3.1 样例拓扑架构

假设某区域有三个 AZ，AZ1、AZ2 和 AZ3。每个 AZ 中有两套机架，每个机架有三台服务器，不考虑混合布署以及单台机器多实例部署，同区域三 AZ 架构集群（3 副本）部署参考如下：

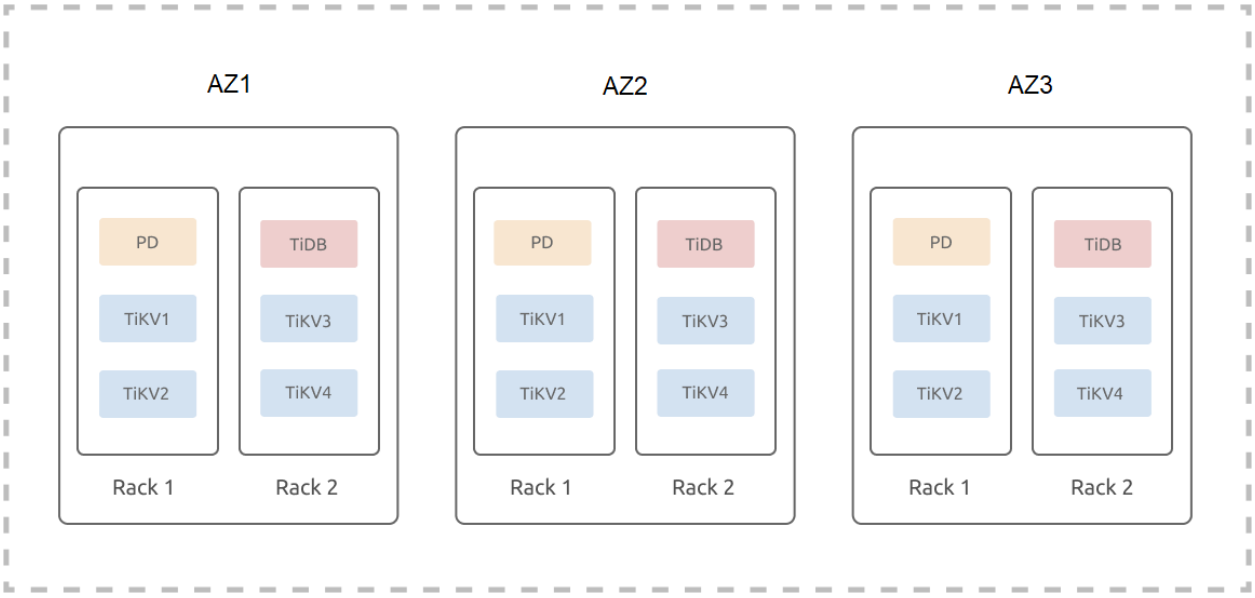


图 178: 同区域三 AZ 集群部署

12.1.2.3.2 TiKV Labels 简介

TiKV 是一个 Multi-Raft 系统，其数据按 Region（默认 96M）切分，每个 Region 的 3 个副本构成了一个 Raft Group。假设一个 3 副本 TiDB 集群，由于 Region 的副本数与 TiKV 实例数量无关，则一个 Region 的 3 个副本只会被调度到其中 3 个 TiKV 实例上，也就是说即使集群扩容 N 个 TiKV 实例，其本质仍是一个 3 副本集群。

由于 3 副本的 Raft Group 只能容忍 1 副本故障，当集群被扩容到 N 个 TiKV 实例时，这个集群依然只能容忍一个 TiKV 实例的故障。2 个 TiKV 实例的故障可能会导致某些 Region 丢失多个副本，整个集群的数据也不再完整，访问到这些 Region 上的数据的 SQL 请求将会失败。而 N 个 TiKV 实例中同时有两个发生故障的概率是远远高于 3 个 TiKV 中同时有两个发生故障的概率的，也就是说 Multi-Raft 系统集群扩容 TiKV 实例越多，其可用性是逐渐降低的。

正因为 Multi-Raft TiKV 系统局限性，Labels 标签应运而出，其主要用于描述 TiKV 的位置信息。Label 信息随着部署或滚动更新操作刷新到 TiKV 的启动配置文件中，启动后的 TiKV 会将自己最新的 Label 信息上报给 PD，PD 根据用户登记的 Label 名称（也就是 Label 元信息），结合 TiKV 的拓扑进行 Region 副本的最优调度，从而提高系统可用性。

12.1.2.3.3 TiKV Labels 样例规划

针对 TiKV Labels 标签，你需要根据已有的物理资源、容灾能力容忍度等方面进行设计与规划，进而提升系统的可用性和容灾能力。并根据已规划的拓扑架构，在集群初始化配置文件中进行配置（此处省略其他非重点项）：

```
server_configs:
  pd:
    replication.location-labels: ["zone","az","rack","host"]

tikv_servers:
  - host: 10.63.10.30
    config:
      server.labels: { zone: "z1", az: "az1", rack: "r1", host: "30" }
  - host: 10.63.10.31
    config:
      server.labels: { zone: "z1", az: "az1", rack: "r1", host: "31" }
  - host: 10.63.10.32
    config:
      server.labels: { zone: "z1", az: "az1", rack: "r2", host: "32" }
  - host: 10.63.10.33
    config:
      server.labels: { zone: "z1", az: "az1", rack: "r2", host: "33" }

  - host: 10.63.10.34
    config:
      server.labels: { zone: "z2", az: "az2", rack: "r1", host: "34" }
  - host: 10.63.10.35
    config:
      server.labels: { zone: "z2", az: "az2", rack: "r1", host: "35" }
  - host: 10.63.10.36
    config:
      server.labels: { zone: "z2", az: "az2", rack: "r2", host: "36" }
  - host: 10.63.10.37
    config:
      server.labels: { zone: "z2", az: "az2", rack: "r2", host: "37" }

  - host: 10.63.10.38
    config:
      server.labels: { zone: "z3", az: "az3", rack: "r1", host: "38" }
  - host: 10.63.10.39
```

```
    config:
      server.labels: { zone: "z3", az: "az3", rack: "r1", host: "39" }
  - host: 10.63.10.40
    config:
      server.labels: { zone: "z3", az: "az3", rack: "r2", host: "40" }
  - host: 10.63.10.41
    config:
      server.labels: { zone: "z3", az: "az3", rack: "r2", host: "41" }
```

本例中，zone 表示逻辑可用区层级，用于控制副本的隔离（当前集群 3 副本）。

不直接采用 az、rack 和 host 三层 Label 结构，是因为考虑到将来可能会扩容 AZ，假设新扩容的 AZ 编号是 AZ2、AZ3 和 AZ4，则只需在对应可用区下扩容 AZ，rack 也只需在对应 AZ 下扩容。

如果直接采用 AZ、rack 和 host 三层 Label 结构，那么扩容 AZ 操作可能需重新添加 Label，TiKV 数据整体需要 Rebalance。

12.1.2.4 高可用和容灾分析

采用区域多 AZ 方案，当任意一个 AZ 故障时，集群能自动恢复服务，不需要人工介入，并能保证数据一致性。注意，各种调度策略主要用于优化性能，当发生故障时，调度机制总是优先考虑可用性而不是性能。

12.2 双区域多 AZ 部署 TiDB

本文档简要介绍双区域多可用区 (Availability Zone, AZ) 部署的架构模型及配置。

本文中的区域指的是地理隔离的不同位置，AZ 指的是区域内部划分的相互独立的资源集合，本文描述的方案同样适用于在两个城市部署三个数据中心（两地三中心）的场景。

12.2.1 简介

双区域多 AZ 架构，即生产数据 AZ、同区域灾备 AZ、跨区域灾备 AZ 的高可用容灾方案。在这种模式下，两个区域的三个 AZ 互联互通，如果一个 AZ 发生故障或灾难，其他 AZ 可以正常运行并对关键业务或全部业务实现接管。相比同区域多 AZ 方案，双区域三 AZ 具有跨区域级高可用能力，可以应对区域级自然灾害。

TiDB 分布式数据库采用 Raft 算法，可以原生支持双区域三 AZ 架构，并保证集群数据的一致性和高可用。而且，同区域 AZ 网络延迟相对较小，可以把业务流量同时派发到同区域两个 AZ，并通过控制 Region Leader 和 PD Leader 分布实现同区域 AZ 共同负载业务流量。

12.2.2 架构

本文以北京和西安部署集群为例，阐述 TiDB 分布式数据库双区域三 AZ 架构的部署模型。

假设北京有两个 AZ，AZ1 和 AZ2，西安有一个 AZ，AZ3。北京同区域两 AZ 之间网络延迟低于 3 ms，北京与西安之间的网络使用 ISP 专线，延迟约为 20 ms。

下图为集群部署架构图，具体如下：

- 集群采用双区域三 AZ 部署方式，分别为北京 AZ1，北京 AZ2，西安 AZ3。
- 集群采用 5 副本模式，其中 AZ1 和 AZ2 分别放 2 份副本，AZ3 放 1 份副本；TiKV 按机柜设置 Label，即每个机柜上有 1 份副本。
- 副本间通过 Raft 协议保证数据的一致性和高可用，对用户完全透明。

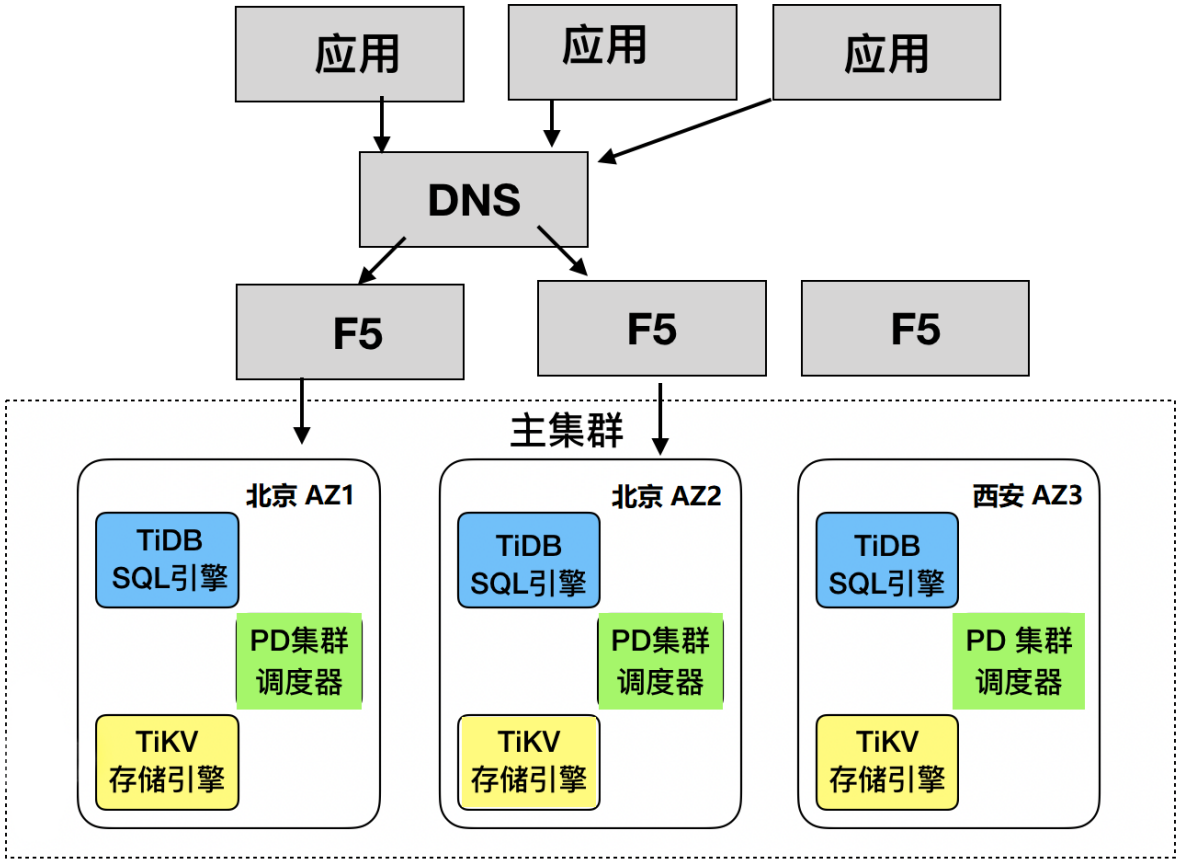


图 179: 双区域三 AZ 集群架构图

该架构具备高可用能力，同时通过 PD 调度保证 Region Leader 只出现在同区域的两个 AZ。相比于三 AZ，即 Region Leader 分布不受限制的方案，双区域三 AZ 方案有以下优缺点：

- 优点
  - Region Leader 都在同区域 AZ，延迟低，数据写入速度更优。
  - 双 AZ 可同时对外提供服务，资源利用率更高。
  - 任一 AZ 失效后，另一 AZ 接管服务，业务可用并且不发生数据丢失。
- 缺点

- 因为数据一致性是基于 Raft 算法实现，当同区域两个 AZ 同时失效时，因为远程灾备 AZ 只剩下一份副本，不满足 Raft 算法大多数副本存活的要求。最终将导致集群暂时不可用，需要从一副本恢复集群，丢失少部分还没同步的热数据。这种情况出现概率较小。
- 由于使用了网络专线，该架构下网络设施成本较高。
- 双区域三 AZ 需设置 5 副本，数据冗余度增加，空间成本攀升。

12.2.2.1 详细示例

北京、西安双区域三 AZ 配置详解：

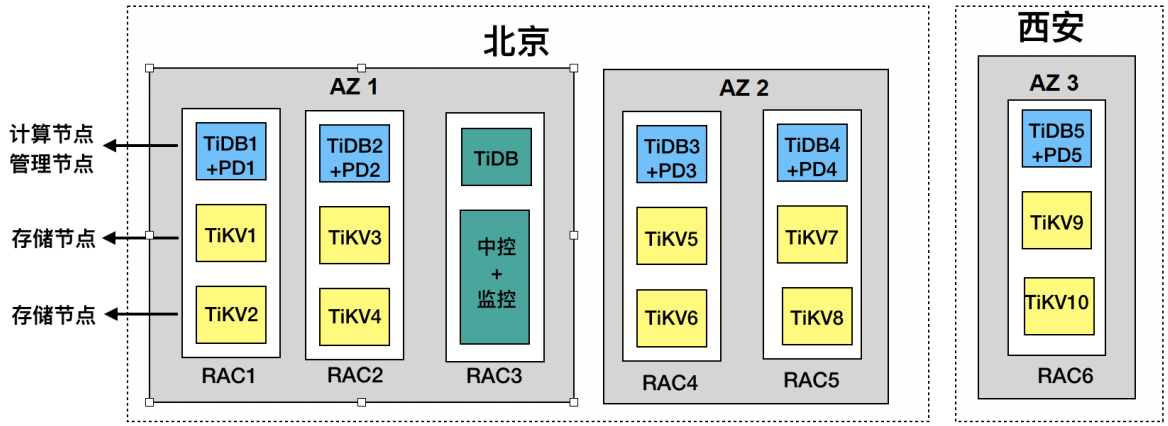


图 180: 双区域三 AZ 配置详图

如上图所示，北京有两个可用区 AZ1 和 AZ2，可用区 AZ1 有三套机架 rac1、rac2 和 rac3，可用区 AZ2 有两套机架 rac4 和 rac5；西安可用区 AZ3 有一套机架 rac6。

AZ1 的 rac1 机架中，一台服务器部署了 TiDB 和 PD 服务，另外两台服务器部署了 TiKV 服务，其中，每台 TiKV 服务器部署了两个 TiKV 实例 (tikv-server)，rac2、rac4、rac5 和 rac6 类似。

机架 rac3 上部署了 TiDB Server、中控及监控服务器。TiDB Server 用于日常管理维护和备份。中控和监控服务器上部署了 Prometheus、Grafana 以及恢复工具。

另可增加备份服务器，其上部署 Drainer，Drainer 以输出 file 文件的方式将 binlog 数据保存到指定位置，实现增量备份的目的。

12.2.3 配置

12.2.3.1 示例

以下为一个 `tiup topology.yaml` 文件示例：

```
## # Global variables are applied to all deployments and used as the default value of
## # the deployments if a specific deployment value is missing.
global:
  user: "tidb"
```

```
  ssh_port: 22
  deploy_dir: "/data/tidb_cluster/tidb-deploy"
  data_dir: "/data/tidb_cluster/tidb-data"

server_configs:
  tikv:
    server.grpc-compression-type: gzip
  pd:
    replication.location-labels:  ["dc","zone","rack","host"]

pd_servers:
  - host: 10.63.10.10
    name: "pd-10"
  - host: 10.63.10.11
    name: "pd-11"
  - host: 10.63.10.12
    name: "pd-12"
  - host: 10.63.10.13
    name: "pd-13"
  - host: 10.63.10.14
    name: "pd-14"

tidb_servers:
  - host: 10.63.10.10
  - host: 10.63.10.11
  - host: 10.63.10.12
  - host: 10.63.10.13
  - host: 10.63.10.14

tikv_servers:
  - host: 10.63.10.30
    config:
      server.labels: { az: "1", replication zone: "1", rack: "1", host: "30" }
  - host: 10.63.10.31
    config:
      server.labels: { az: "1", replication zone: "2", rack: "2", host: "31" }
  - host: 10.63.10.32
    config:
      server.labels: { az: "2", replication zone: "3", rack: "3", host: "32" }
  - host: 10.63.10.33
    config:
      server.labels: { az: "2", replication zone: "4", rack: "4", host: "33" }
  - host: 10.63.10.34
    config:
      server.labels: { az: "3", replication zone: "5", rack: "5", host: "34" }
```

```
      raftstore.raft-min-election-timeout-ticks: 1000
      raftstore.raft-max-election-timeout-ticks: 1200

monitoring_servers:
  - host: 10.63.10.60

grafana_servers:
  - host: 10.63.10.60

alertmanager_servers:
  - host: 10.63.10.60
```

12.2.3.2 Labels 设计

在双区域三 AZ 部署方式下，对于 Labels 的设计需要充分考虑到系统的可用性和容灾能力，建议根据部署的物理结构来定义 AZ、 replication zone、 rack 和 host 四个等级。

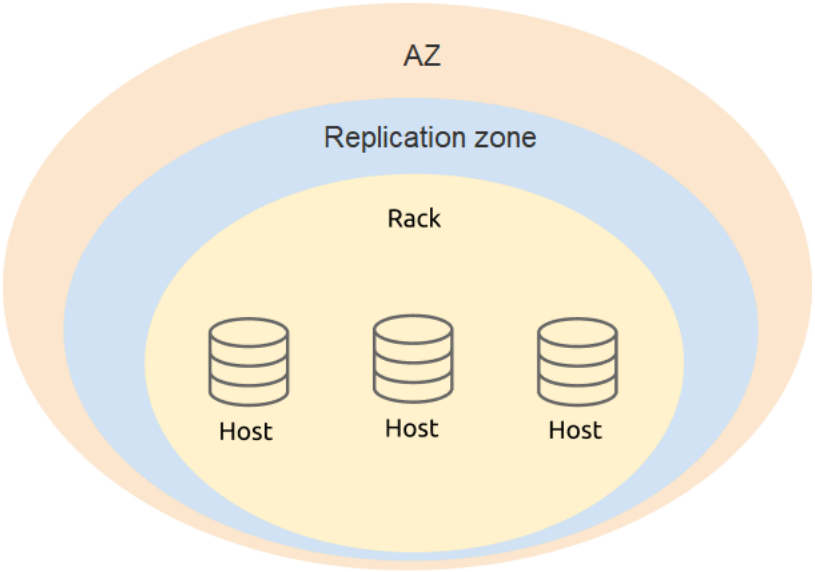


图 181: Label 逻辑定义图

PD 设置中添加 TiKV label 的等级配置。

```
server_configs:
  pd:
    replication.location-labels:  ["az","replication zone","rack","host"]
```

tikv_servers 设置基于 TiKV 真实物理部署位置的 Label 信息，方便 PD 进行全局管理和调度。

```
tikv_servers:
  - host: 10.63.10.30
    config:
```

```
    server.labels: { az: "1", replication zone: "1", rack: "1", host: "30" }
- host: 10.63.10.31
  config:
    server.labels: { az: "1", replication zone: "2", rack: "2", host: "31" }
- host: 10.63.10.32
  config:
    server.labels: { az: "2", replication zone: "3", rack: "3", host: "32" }
- host: 10.63.10.33
  config:
    server.labels: { az: "2", replication zone: "4", rack: "4", host: "33" }
- host: 10.63.10.34
  config:
    server.labels: { az: "3", replication zone: "5", rack: "5", host: "34" }
```

12.2.3.3 参数配置优化

在双区域三 AZ 的架构部署中，从性能优化的角度，除了常规参数配置外，还需要对集群中相关组件参数进行调整。

- 启用 TiKV gRPC 消息压缩。由于需要在网络中传输集群数据，可开启 gRPC 消息压缩，降低网络流量。

```
server.grpc-compression-type: gzip
```

- 优化跨区域 AZ3 的 TiKV 节点网络，修改 TiKV 的如下参数，拉长跨区域副本参与选举的时间，避免跨区域 TiKV 中的副本参与 Raft 选举。

```
raftstore.raft-min-election-timeout-ticks: 1000
raftstore.raft-max-election-timeout-ticks: 1200
```

- 调度设置。在集群启动后，通过 `tiup ctl:v<CLUSTER_VERSION> pd` 工具进行调度策略修改。修改 TiKV Raft 副本数按照安装时规划好的副本数进行设置，在本例中为 5 副本。

```
config set max-replicas 5
```

- 禁止向跨区域 AZ 调度 Raft Leader，当 Raft Leader 在跨区域 AZ 时，会造成不必要的本区域 AZ 与远程 AZ 间的网络消耗，同时，网络带宽和延迟也会对 TiDB 的集群性能产生影响。

```
config set label-property reject-leader dc 3
```

> **注意：**
>
> TiDB 5.2 及以上版本默认不支持 `label-property` 配置。若要设置副本策略，请使用Placement Rules。

- 设置 PD 的优先级，为了避免出现跨区域 AZ 的 PD 成为 Leader，可以将本区域 AZ 的 PD 优先级调高（数字越大，优先级越高），将跨区域的 PD 优先级调低。

```
member leader_priority PD-10 5
member leader_priority PD-11 5
member leader_priority PD-12 5
member leader_priority PD-13 5
member leader_priority PD-14 1
```

12.3 单区域双 AZ 部署 TiDB

本文介绍单区域双可用区 (Availability Zone, AZ) 的部署模式，包括方案架构、示例配置、副本方法及启用该模式的方法。

本文中的区域指的是地理隔离的不同位置，AZ 指的是区域内部划分的相互独立的资源集合，本文所描述的方案同样适用于同一城市两个数据中心（同城双中心）的场景。

12.3.1 简介

TiDB 通常采用多 AZ 部署方案保证集群高可用和容灾能力。多 AZ 部署方案包括单区域多 AZ 部署模式、双区域多 AZ 部署模式等多种部署模式。本文介绍单区域双 AZ 部署方案，即在同一区域部署两个 AZ，成本更低，同样能满足高可用和容灾要求。该部署方案采用自适应同步模式，即 Data Replication Auto Synchronous，简称 DR Auto-Sync。

单区域双 AZ 部署方案下，两个 AZ 通常位于同一个城市或两个相邻城市（例如北京和廊坊），相距 50 km 以内，AZ 间的网络连接延迟小于 1.5 ms，带宽大于 10 Gbps。

12.3.2 部署架构

本文以某市为例，市里有两个 AZ，AZ1 和 AZ2，AZ1 为主用区域，AZ2 为从属区域，分别位于城东和城西。

下图为集群部署架构图，具体如下：

- 集群采用推荐的 6 副本模式，其中 AZ1 中放 3 个 Voter，AZ2 中放 2 个 Follower 副本和 1 个 Learner 副本。TiKV 按机房的实际情况打上合适的 Label。
- 副本间通过 Raft 协议保证数据的一致性和高可用，对用户完全透明。

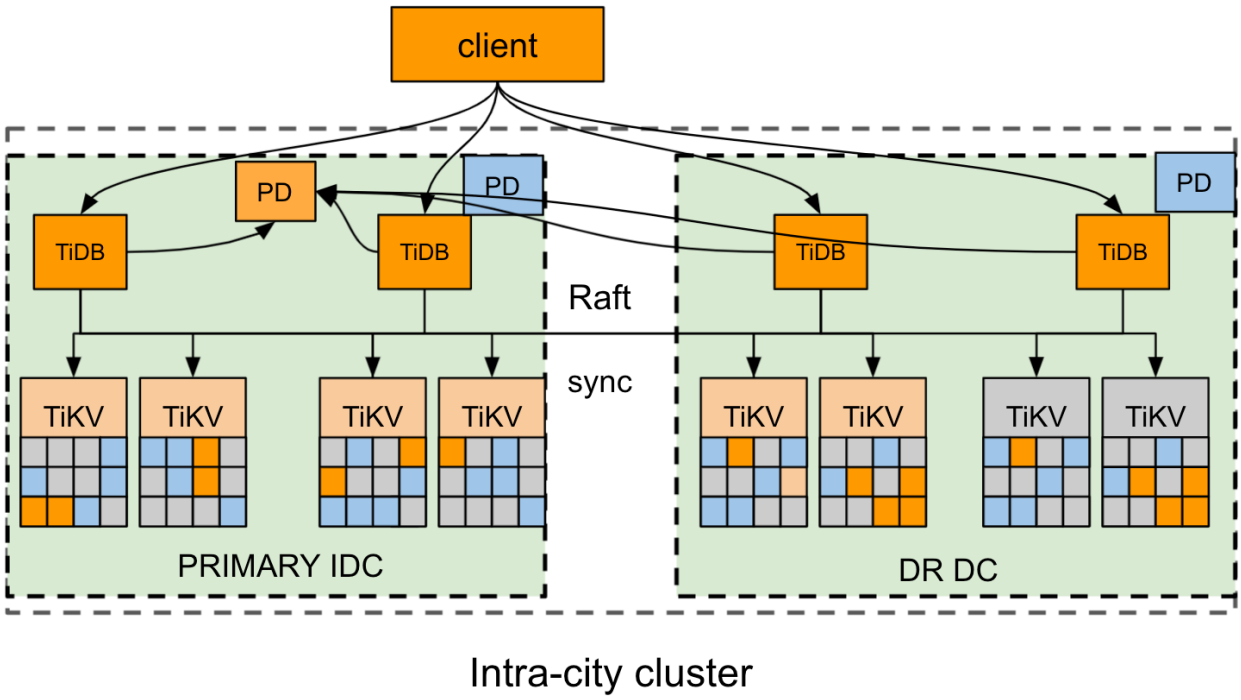


图 182: 单区域双 AZ 集群架构图

该部署方案定义了三种状态来控制和标示集群的同步状态，该状态约束了 TiKV 的同步方式。集群的复制模式可以自动在三种状态之间自适应切换。要了解切换过程，请参考状态转换。

- sync：同步复制模式，此时从 AZ (DR) 至少有一个副本与主 AZ (PRIMARY) 进行同步，Raft 算法保证每条日志按 Label 同步复制到 DR。
- async：异步复制模式，此时不保证 DR 与 PRIMARY 完全同步，Raft 算法使用经典的 majority 方式复制日志。
- sync-recover：恢复同步，此时不保证 DR 与 PRIMARY 完全同步，Raft 逐步切换成 Label 复制，切换成功后汇报给 PD。

12.3.3 配置

12.3.3.1 示例

以下 `tiup topology.yaml` 示例拓扑文件为单区域双 AZ 典型的拓扑配置：

```
## # Global variables are applied to all deployments and used as the default value of
## # the deployments if a specific deployment value is missing.
global:
  user: "tidb"
  ssh_port: 22
  deploy_dir: "/data/tidb_cluster/tidb-deploy"
  data_dir: "/data/tidb_cluster/tidb-data"
server_configs:
  pd:
```

```
    replication.location-labels:  ["az","rack","host"]
pd_servers:
  - host: 10.63.10.10
    name: "pd-10"
  - host: 10.63.10.11
    name: "pd-11"
  - host: 10.63.10.12
    name: "pd-12"
tidb_servers:
  - host: 10.63.10.10
  - host: 10.63.10.11
  - host: 10.63.10.12
tikv_servers:
  - host: 10.63.10.30
    config:
      server.labels: { az: "east", rack: "east-1", host: "30" }
  - host: 10.63.10.31
    config:
      server.labels: { az: "east", rack: "east-2", host: "31" }
  - host: 10.63.10.32
    config:
      server.labels: { az: "east", rack: "east-3", host: "32" }
  - host: 10.63.10.33
    config:
      server.labels: { az: "west", rack: "west-1", host: "33" }
  - host: 10.63.10.34
    config:
      server.labels: { az: "west", rack: "west-2", host: "34" }
  - host: 10.63.10.35
    config:
      server.labels: { az: "west", rack: "west-3", host: "35" }
monitoring_servers:
  - host: 10.63.10.60
grafana_servers:
  - host: 10.63.10.60
alertmanager_servers:
  - host: 10.63.10.60
```

12.3.3.2 Placement Rules 规划

为了按照规划的集群拓扑进行部署，你需要使用Placement Rules 来规划集群副本的放置位置。以 6 副本（3 个 Voter 副本在主 AZ，2 个 Follower 副本和 1 个 Learner 副本在从 AZ）的部署方式为例，可使用 Placement Rules 进行如下副本配置：

```
cat rule.json
```

```
[
  {
    "group_id": "pd",
    "group_index": 0,
    "group_override": false,
    "rules": [
      {
        "group_id": "pd",
        "id": "az-east",
        "start_key": "",
        "end_key": "",
        "role": "voter",
        "count": 3,
        "label_constraints": [
          {
            "key": "az",
            "op": "in",
            "values": [
              "east"
            ]
          }
        ],
        "location_labels": [
          "az",
          "rack",
          "host"
        ]
      },
      {
        "group_id": "pd",
        "id": "az-west",
        "start_key": "",
        "end_key": "",
        "role": "follower",
        "count": 2,
        "label_constraints": [
          {
            "key": "az",
            "op": "in",
            "values": [
              "west"
            ]
          }
        ],
        "location_labels": [
```

```
          "az",
          "rack",
          "host"
        ]
      },
      {
        "group_id": "pd",
        "id": "az-west",
        "start_key": "",
        "end_key": "",
        "role": "learner",
        "count": 1,
        "label_constraints": [
          {
            "key": "az",
            "op": "in",
            "values": [
              "west"
            ]
          }
        ],
        "location_labels": [
          "az",
          "rack",
          "host"
        ]
      }
    ]
  }
]
```

如果需要使用 `rule.json` 中的配置，你可以使用以下命令把原有的配置备份到 `default.json` 文件，再使用 `rule.json` 中的配置覆盖原有配置：

```
pd-ctl config placement-rules rule-bundle load --out="default.json"
pd-ctl config placement-rules rule-bundle save --in="rule.json"
```

如果需要回退配置，你可以还原备份的 `default.json` 文件或者手动编写如下的 json 文件并将其覆盖到现有的配置文件中：

```
cat default.json
[
  {
    "group_id": "pd",
    "group_index": 0,
    "group_override": false,
```

```
    "rules": [
      {
        "group_id": "pd",
        "id": "default",
        "start_key": "",
        "end_key": "",
        "role": "voter",
        "count": 5
      }
    ]
  }
]
```

12.3.3.3 启用自适应同步模式

副本的复制模式由 PD 节点控制。如果要使用 DR Auto-sync 自适应同步模式，需要按照以下任一方法修改 PD 的配置。

- 方法一：先配置 PD 的配置文件，然后部署集群。

```
[replication-mode]
replication-mode = "dr-auto-sync"
[replication-mode.dr-auto-sync]
label-key = "az"
primary = "east"
dr = "west"
primary-replicas = 3
dr-replicas = 2
wait-store-timeout = "1m"
```

- 方法二：如果已经部署了集群，则使用 pd-ctl 命令修改 PD 的配置。

```
config set replication-mode dr-auto-sync
config set replication-mode dr-auto-sync label-key az
config set replication-mode dr-auto-sync primary east
config set replication-mode dr-auto-sync dr west
config set replication-mode dr-auto-sync primary-replicas 3
config set replication-mode dr-auto-sync dr-replicas 2
```

配置项说明：

- `replication-mode` 为待启用的复制模式，以上示例中设置为 `dr-auto-sync`。默认使用 majority 算法。
- `label-key` 用于区分不同的 AZ，需要和 Placement Rules 相匹配。其中主 AZ 为“east”，从 AZ 为“west”。
- `primary-replicas` 是主 AZ 上 Voter 副本的数量。
- `dr-replicas` 是从 AZ 上 Voter 副本的数量。

- `wait-store-timeout` 是当出现网络隔离或者故障时，切换到异步复制模式的等待时间。如果超过这个时间还没恢复，则自动切换到异步复制模式。默认时间为 60 秒。

如果需要检查当前集群的复制状态，可以通过以下 API 获取：

```
curl http://pd_ip:pd_port/pd/api/v1/replication_mode/status
```

```
{
  "mode": "dr-auto-sync",
  "dr-auto-sync": {
    "label-key": "az",
    "state": "sync"
  }
}
```

12.3.3.3.1 状态转换

简单来讲，集群的复制模式可以自动在三种状态之间自适应的切换：

- 当集群一切正常时，会进入同步复制模式来最大化地保障灾备 AZ 的数据完整性。
- 当 AZ 网络断连或灾备 AZ 发生整体故障时，在经过一段提前设置好的保护窗口之后，集群会进入异步复制状态，来保障业务的可用性。
- 当 AZ 网络重连或灾备 AZ 整体恢复之后，灾备 AZ 的 TiKV 节点会重新加入到集群，逐步同步数据并最终转为同步复制模式。

状态转换的细节过程如下：

1. 初始化：集群在初次启动时处于 sync（同步复制）模式，PD 会下发信息给 TiKV，所有 TiKV 节点会严格按照 sync 模式的要求进行工作。
2. 同步切异步：PD 通过定时检查 TiKV 的心跳信息来判断 TiKV 是否宕机或断连。如果宕机数超过 Primary 和 DR 各自副本的数量 `primary-replicas` 和 `dr-replicas`，意味着无法完成同步复制，需要切换状态。当宕机时间超过了 `wait-store-timeout` 设定的时间，PD 将集群状态切换成 async（异步复制）模式。然后 PD 再将 async 状态下发到所有 TiKV 节点，TiKV 的复制模式由双 AZ 同步方式转为原生的 Raft 大多数落实方式 (majority)。
3. 异步切同步：PD 通过定时检查 TiKV 的心跳信息来判断 TiKV 是否恢复连接，如果宕机数小于 Primary 和 DR 各自副本的数量，意味着可以切回同步。PD 会将集群复制状态先切换至 sync-recover，再将该状态下发给所有 TiKV 节点。TiKV 的所有 Region 逐步切换成双 AZ 同步复制模式，切换成功后通过心跳将状态同步信息给 PD。PD 记录 TiKV 上 Region 的状态并统计恢复进度。当 TiKV 的所有 Region 都完成了同步复制模式的切换，PD 将集群复制状态切换为 sync。

12.3.3.4 灾难恢复

本节介绍单区域双 AZ 部署提供的容灾恢复方案。

当处于同步复制状态的集群发生了灾难，可进行 RPO = 0 的数据恢复：

- 如果主 AZ 发生故障，丢失了大多数 Voter 副本，但是从 AZ 有完整的数据，可在从 AZ 恢复数据。此时需要人工介入，通过专业工具恢复。如需获取支持，请联系 PingCAP 服务与支持。
- 如果从 AZ 发生故障，丢失了少数 Voter 副本，能自动切换成 async 异步复制模式。

当不处于同步复制状态的集群发生了灾难，不能保证满足 RPO = 0 进行数据恢复：

- 如果丢失了大多数 Voter 副本，需要人工介入，通过专业工具恢复（恢复方式请联系 TiDB 团队）。

12.4 读取历史数据

12.4.1 使用 Stale Read 功能读取历史数据（推荐）

12.4.1.1 Stale Read 功能的使用场景

本文档介绍 Stale Read 的使用场景。Stale Read 是一种读取历史数据版本的机制，读取 TiDB 中存储的历史数据版本。通过 Stale Read 功能，你能从指定时间点或时间范围内读取对应的历史数据，从而避免数据同步带来延迟。当使用 Stale Read 时，TiDB 默认会随机选择一个副本来读取数据，因此能利用所有副本。如果你的应用程序不能容忍读到非实时的数据，请勿使用 Stale Read，否则读到的数据可能不是最新成功写入的数据。

12.4.1.1.1 场景描述

- 场景一：如果一个事务仅涉及只读操作，并且一定程度上可容忍牺牲实时性，你可以使用 Stale Read 功能来读取历史数据。由于牺牲了一定的实时性，使用 Stale Read 后，TiDB 可以让请求分发到任一个副本上，使得查询的执行获得更大的吞吐量。特别是在一些小表的查询场景中，如果使用了强一致性读，Leader 可能集中在某一个存储节点上，导致查询压力集中在该节点，成为整个查询的瓶颈。通过 Stale Read，可以提升了查询整体的吞吐能力，从而显著提升查询性能。
- 场景二：在部分跨数据中心部署的场景中，如果使用了强一致性的 Follower 读，为了读到的数据与 Leader 上的数据一致，会产生跨数据中心获取 `Readindex` 来校验的请求，导致整体查询的访问延迟增加。通过使用 Stale Read 功能，可以牺牲一定的实时性，就近访问对应数据所在当前中心的副本，避免跨数据中心的网络延迟，降低整体查询的访问延迟。详情参考在三数据中心下就近读取数据。

12.4.1.1.2 使用方法

TiDB 提供语句级别、会话级别以及全局级别的 Stale Read 使用方式，具体使用方法如下：

- 语句级别：
  - 指定一个精确的时间点（推荐）：如需 TiDB 读取一个时间点上保证全局事务记录一致性的数据并且不破坏隔离级别，你可以指定这个时间点对应的时间戳。要使用该方式，请参阅`AS OF TIMESTAMP` 语法文档。
  - 指定时间范围：如需 TiDB 读取在一个时间范围内尽可能新的数据并且不破坏隔离级别，你可以指定一个时间范围。在指定时间范围内，TiDB 会选择一个合适的时间戳，该时间戳能保证所访问的副本上不存在开始于这个时间戳之前且还没有提交的相关事务，即能保证在所访问的可用副本上可执行读取操作而且不会被阻塞。要使用该方式，请参阅`AS OF TIMESTAMP` 语法文档和该文档中`TIDB_BOUNDED_STALENESS` 函数部分的介绍。

- 会话级别：
  - 指定时间范围：在会话级别中，如需 TiDB 在后续的查询中读取一个时间范围内尽可能新的数据并且不破坏隔离级别，你可以通过设置一个 session 变量 `tidb_read_staleness` 来指定一个时间范围。要使用该方式，请参阅通过系统变量 `tidb_read_staleness` 读取历史数据。

除此以外，你也可以通过设置系统变量`tidb_external_ts` 来在某一会话或全局范围读取某一时间点前的历史数据。要使用该方式，请参阅通过系统变量 `tidb_external_ts` 读取历史数据。

减少 Stale Read 延时

Stale Read 功能会定期推进 TiDB 集群的 Resolved TS 时间戳，该时间戳能保证 TiDB 读到满足事务一致性的数据。当 Stale Read 使用的时间戳（比如 `AS OF TIMESTAMP '2016-10-08 16:45:26'`）大于 Resolved TS 时，Stale Read 会先触发 TiDB 推进 Resolved TS，等待推进完成后再读取数据，从而导致延时上升。

通过调整下面 TiKV 的配置项，你可以使 TiDB 加快 Resolved TS 推进，以减少 Stale Read 延时：

```
[resolved-ts]
advance-ts-interval = "20s" # 默认为 20 秒，可适当调小该值以加快 Resolved TS 推进，比如调整为 1
    ↪ 秒。
```

> 注意：
>
> 调小该参数会增加 TiKV CPU 使用率和各节点之间的流量。

12.4.1.1.3 限制

当对表的 Stale Read 查询下推到 TiFlash 时，如果该表在 Stale Read 所指定的读取时间戳之后执行过 DDL 操作，此查询将会报错。原因是 TiFlash 只支持从最新的表结构读取数据。

例如：

```
create table t1(id int);
alter table t1 set tiflash replica 1;
```

一分钟后进行 DDL 操作：

```
alter table t1 add column c1 int not null;
```

然后使用 Stale Read 读取一分钟前的数据：

```
set @@session.tidb_enforce_mpp=1;
select * from t1 as of timestamp NOW() - INTERVAL 1 minute;
```

此时 TiFlash 会报错：

```
ERROR 1105 (HY000): other error for mpp stream: From MPP<query:<query_ts:1673950975508472943,
    ↪ local_query_id:18, server_id:111947, start_ts:438816196526080000>,task_id:1>: Code: 0, e.
    ↪ displayText() = DB::TiFlashException: Table 323 schema version 104 newer than query
    ↪ schema version 100, e.what() = DB::TiFlashException,
```

把 Stale Read 指定的读取时间戳改成 DDL 操作完成之后的时间，即可避免该错误。

12.4.1.2 使用 AS OF TIMESTAMP 语法读取历史数据

本文档介绍如何通过 `AS OF TIMESTAMP` 语句使用Stale Read 功能来读取 TiDB 历史版本数据，包括具体的操作示例以及历史数据的保存策略。

TiDB 支持通过标准 SQL 接口，即通过 `AS OF TIMESTAMP` SQL 语法的形式读取历史数据，无需特殊的服务器或者驱动器。当数据被更新或删除后，你可以通过 SQL 接口将更新或删除前的数据读取出来。

> **注意：**
>
> 读取历史数据时，即使当前数据的表结构相较于历史数据的表结构已经发生改变，历史数据也会以当时的历史表结构来返回。

12.4.1.2.1 语法方式

你可以通过以下三种方式使用 `AS OF TIMESTAMP` 语法：

- `SELECT ... FROM ... AS OF TIMESTAMP`
- `START TRANSACTION READ ONLY AS OF TIMESTAMP`
- `SET TRANSACTION READ ONLY AS OF TIMESTAMP`

如果你想要指定一个精确的时间点，可在 `AS OF TIMESTAMP` 中使用日期时间和时间函数，日期时间的格式为：“2016-10-08 16:45:26.999”，最小时间精度范围为毫秒，通常可只写到秒，例如 “2016-10-08 16:45:26”。你也可以通过 `NOW(3)` 函数获得精确到毫秒的当前时间。如果想读取几秒前的数据，推荐使用例如 `NOW()- INTERVAL 10 SECOND` 的表达式。(推荐)

如果你想要指定一个时间范围，需要使用 `TIDB_BOUNDED_STALENESS()` 函数。使用该函数，TiDB 会在指定的时间范围内选择一个合适的时间戳，该时间戳能保证所访问的副本上不存在开始于这个时间戳之前且还没有提交的相关事务，即能保证所访问的可用副本上执行读取操作而且不会被阻塞。用法为 `TIDB_BOUNDED_STALENESS (t1, t2)`，其中 `t1` 和 `t2` 为时间范围的两端，支持使用日期时间和时间函数。

示例如下：

- `AS OF TIMESTAMP '2016-10-08 16:45:26'` 表示读取在 2016 年 10 月 8 日 16 点 45 分 26 秒时最新的数据。
- `AS OF TIMESTAMP NOW()- INTERVAL 10 SECOND` 表示读取 10 秒前最新的数据。
- `AS OF TIMESTAMP TIDB_BOUNDED_STALENESS('2016-10-08 16:45:26', '2016-10-08 16:45:29')` 表示读取在 2016 年 10 月 8 日 16 点 45 分 26 秒到 29 秒的时间范围内尽可能新的数据。

- `AS OF TIMESTAMP TIDB_BOUNDED_STALENESS(NOW()- INTERVAL 20 SECOND, NOW())` 表示读取 20 秒前到现在的时间范围内尽可能新的数据。

> 注意：
>
> 除了指定时间戳，`AS OF TIMESTAMP` 语法最常用使用的方式是读几秒前的数据。如果采用这种方式，推荐读 5 秒以上的历史数据。
>
> 使用 Stale Read 时需要为 TiDB 和 PD 节点部署 NTP 服务，防止 TiDB 指定的时间戳超过当前最新的 TSO 分配进度（如几秒后的时间戳），或者落后于 GC safe point 的时间戳。当指定的时间戳超过服务范围，TiDB 会返回错误。
>
> 你可以通过调整 TiKV 的 `advance-ts-interval` 配置项提高 Stale Read 数据的时效性（即减少延时）。详情参见减少 Stale Read 延时。

12.4.1.2.2 示例

本节通过多个示例介绍 `AS OF TIMESTAMP` 语法的不同使用方法。在本节中，先介绍如何准备用于恢复的数据，再分别展示如何通过 `SELECT`、`START TRANSACTION READ ONLY AS OF TIMESTAMP`、`SET TRANSACTION READ ONLY AS OF TIMESTAMP` 使用 `AS OF TIMESTAMP`。

准备数据

在准备数据阶段，创建一张表，并插入若干行数据：

```
create table t (c int);
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
insert into t values (1), (2), (3);
```

```
Query OK, 3 rows affected (0.00 sec)
```

查看表中的数据：

```
select * from t;
```

```
+------+
| c    |
+------+
|    1 |
|    2 |
|    3 |
+------+
3 rows in set (0.00 sec)
```

查看当前时间：

```
select now();
```

```
+---------------------+
| now()               |
+---------------------+
| 2021-05-26 16:45:26 |
+---------------------+
1 row in set (0.00 sec)
```

更新某一行数据：

```
update t set c=22 where c=2;
```

```
Query OK, 1 row affected (0.00 sec)
```

确认数据已经被更新：

```
select * from t;
```

```
+------+
| c    |
+------+
|    1 |
|   22 |
|    3 |
+------+
3 rows in set (0.00 sec)
```

通过 SELECT 读取历史数据

通过SELECT ... FROM ... AS OF TIMESTAMP 语句读取一个基于历史时间的数据。

```
select * from t as of timestamp '2021-05-26 16:45:26';
```

```
+------+
| c    |
+------+
|    1 |
|    2 |
|    3 |
+------+
3 rows in set (0.00 sec)
```

> **注意：**
>
> 通过 SELECT 语句读取多个表时要保证 TIMESTAMP EXPRESSION 是一致的。比如：`select * from t as of timestamp NOW()- INTERVAL 2 SECOND, c as of timestamp NOW()- INTERVAL 2 SECOND;`。此外，在 SELECT 语句中，你必须要指定相关数据表的 as of 信息，若不指定，SELECT 语句会默认读最新的数据。

通过 START TRANSACTION READ ONLY AS OF TIMESTAMP 读取历史数据

通过START TRANSACTION READ ONLY AS OF TIMESTAMP 语句，你可以开启一个基于历史时间的只读事务，该事务基于所提供的历史时间来读取历史数据。

```
start transaction read only as of timestamp '2021-05-26 16:45:26';
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
select * from t;
```

```
+------+
| c    |
+------+
|    1 |
|    2 |
|    3 |
+------+
3 rows in set (0.00 sec)
```

```
commit;
```

```
Query OK, 0 rows affected (0.00 sec)
```

当事务结束后，即可读取最新数据。

```
select * from t;
```

```
+------+
| c    |
+------+
|    1 |
|   22 |
|    3 |
+------+
3 rows in set (0.00 sec)
```

> **注意：**
>
> 通过 START TRANSACTION READ ONLY AS OF TIMESTAMP 开启的事务为只读事务。假如在该事务中执行写入操作，操作将会被该事务拒绝。

通过 SET TRANSACTION READ ONLY AS OF TIMESTAMP 读取历史数据

通过SET TRANSACTION READ ONLY AS OF TIMESTAMP 语句，你可以将下一个事务设置为基于指定历史时间的只读事务。该事务将会基于所提供的历史时间来读取历史数据。

```
set transaction read only as of timestamp '2021-05-26 16:45:26';
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
begin;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
select * from t;
```

```
+------+
| c    |
+------+
|    1 |
|    2 |
|    3 |
+------+
3 rows in set (0.00 sec)
```

```
commit;
```

```
Query OK, 0 rows affected (0.00 sec)
```

当事务结束后，即可读取最新数据。

```
select * from t;
```

```
+------+
| c    |
+------+
|    1 |
|   22 |
|    3 |
+------+
3 rows in set (0.00 sec)
```

> **注意：**
>
> 通过 SET TRANSACTION READ ONLY AS OF TIMESTAMP 开启的事务为只读事务。假如在该事务中执行写入操作，操作将会被该事务拒绝。

12.4.1.3 通过系统变量 tidb_read_staleness 读取历史数据

为支持读取历史版本数据，TiDB 从 5.4 版本起引入了一个新的系统变量 tidb_read_staleness。本文档介绍如何通过该系统变量读取历史数据，其中包括具体的操作流程。

12.4.1.3.1 功能介绍

系统变量 tidb_read_staleness 用于设置当前会话允许读取的历史数据范围，其数据类型为 int，作用域为 SESSION。设置该变量后，TiDB 会从参数允许的范围内选出一个尽可能新的时间戳，并影响后继的所有读操作。比如，如果该变量的值设置为 -5，TiDB 会在 5 秒时间范围内，保证 TiKV 拥有对应历史版本数据的情况下，选择尽可能新的一个时间戳。

开启 tidb_read_staleness 后，你仍可以进行以下操作：

- 在当前会话中插入、修改、删除数据或进行 DML 操作。这些语句不会受到 tidb_read_staleness 的影响。
- 在当前会话开启交互式事务。在该事务内的查询依旧是读取最新版本的数据。

完成对历史版本数据的读取后，你可以通过以下两种方式来读取最新版本的数据。

- 结束当前会话。
- 使用 SET 语句，把 tidb_read_staleness 变量的值设为 ""。

> **注意：**
>
> 你可以通过调整 TiKV 的 advance-ts-interval 配置项提高 Stale Read 数据的时效性（即减少延时），详情参见减少 Stale Read 延时。

12.4.1.3.2 示例

本节通过具体操作示例介绍系统变量 tidb_read_staleness的使用方法。

1. 初始化阶段。创建一个表后，在表中插入几行数据：

```
create table t (c int);
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
insert into t values (1), (2), (3);
```

```
Query OK, 3 rows affected (0.00 sec)
```

2. 查看表中的数据：

```
select * from t;
```

```
+------+
| c    |
+------+
|    1 |
|    2 |
|    3 |
+------+
3 rows in set (0.00 sec)
```

3. 更新某一行数据：

```
update t set c=22 where c=2;
```

```
Query OK, 1 row affected (0.00 sec)
```

4. 确认数据已经被更新：

```
select * from t;
```

```
+------+
| c    |
+------+
|    1 |
|   22 |
|    3 |
+------+
3 rows in set (0.00 sec)
```

5. 设置一个特殊的环境变量 `tidb_read_staleness`。

   该变量的作用域为 SESSION，设置变量值后，TiDB 会读取变量值时间之前的最新一个版本的数据。

   以下设置表示 TiDB 会从 5 秒前至现在的时间范围内选择一个尽可能新的时间戳，将其用作为历史数据读取的时间戳：

```
set @@tidb_read_staleness="-5";
```

```
Query OK, 0 rows affected (0.00 sec)
```

> **注意：**
>
> 必须在 tidb_read_staleness 前使用 @@，而非 @。因为 @@ 表示系统变量，@ 则表示用户变量。你需要根据第 3 步到第 4 步所花费的时间，来设定你要读取的历史时间范围，即 tidb_read_staleness 的值。否则，查询结果中显示的会是最新数据，而非历史数据。因此，请根据自己的实际操作情况调整该时间范围。比如，在本示例中，由于设定的时间范围是 5 秒，你需要在 5 秒内完成第 3 步和第 4 步。

这里读取到的内容即为更新前的数据，也就是历史版本的数据：

```
select * from t;
```

```
+------+
| c    |
+------+
|    1 |
|    2 |
|    3 |
+------+
3 rows in set (0.00 sec)
```

6. 清空这个变量后，即可读取最新版本数据：

```
set @@tidb_read_staleness="";
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
select * from t;
```

```
+------+
| c    |
+------+
|    1 |
|   22 |
|    3 |
+------+
3 rows in set (0.00 sec)
```

12.4.1.4 通过系统变量 tidb_external_ts 读取历史数据

为了支持读取历史版本数据，TiDB 从 v6.4.0 起引入了一个新的系统变量tidb_external_ts。本文档介绍如何通过该系统变量读取历史数据，其中包括具体的操作流程。

12.4.1.4.1 场景介绍

通过配置让 TiDB 能够读取某一固定时间点的历史数据对于 TiCDC 等数据同步工具非常有用。在数据同步工具完成了某一时间点前的数据同步之后，可以通过设置下游 TiDB 的 `tidb_external_ts` 系统变量，使得下游 TiDB 的请求能够读取到该时间点前的数据。这将避免在同步过程中，下游 TiDB 读取到尚未完全同步而不一致的数据。

12.4.1.4.2 功能介绍

系统变量`tidb_external_ts` 用于指定启用 `tidb_enable_external_ts_read` 时，读取历史数据使用的时间戳。

系统变量`tidb_enable_external_ts_read` 控制着是否在当前会话或全局启用读取历史数据的功能。默认值为 `OFF`，这意味着该功能关闭，并且设置 `tidb_external_ts` 没有作用。当该变量被全局地设置为 `ON` 时，所有的请求都将读取到 `tidb_external_ts` 指定时间之前的历史数据。如果 `tidb_enable_external_ts_read` 仅在某一会话被设置为 `ON`，则只有该会话中的请求会读取到历史数据。

当 `tidb_enable_external_ts_read` 被设置为 `ON` 时，TiDB 会进入只读模式，任何写请求都会失败并且返回错误 `ERROR 1836 (HY000): Running in read-only mode`。

12.4.1.4.3 示例

以下是一个使用该功能的示例：

1. 创建一个表后，在表中插入几行数据：

```
CREATE TABLE t (c INT);
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
INSERT INTO t VALUES (1), (2), (3);
```

```
Query OK, 3 rows affected (0.00 sec)
```

2. 查看表中的数据：

```
SELECT * FROM t;
```

```
+------+
| c    |
+------+
|    1 |
|    2 |
|    3 |
+------+
3 rows in set (0.00 sec)
```

3. 将 `tidb_external_ts` 设置为 `@@tidb_current_ts`：

```
START TRANSACTION;
SET GLOBAL tidb_external_ts = @@tidb_current_ts;
COMMIT;
```

4. 插入新的一行并确认新的一行已经被插入：

```
INSERT INTO t VALUES (4);
```

```
Query OK, 1 row affected (0.001 sec)
```

```
SELECT * FROM t;
```

```
+------+
| id   |
+------+
|    1 |
|    2 |
|    3 |
|    4 |
+------+
4 rows in set (0.00 sec)
```

5. 将 tidb_enable_external_ts_read 设置为 ON 后，再次查询表中的数据：

```
SET tidb_enable_external_ts_read = ON;
SELECT * FROM t;
```

```
+------+
| c    |
+------+
|    1 |
|    2 |
|    3 |
+------+
3 rows in set (0.00 sec)
```

因为 tidb_external_ts 被设置为插入这一行之前的时间，在启动 tidb_enable_external_ts_read 后，将读取不到新插入的行。

12.4.2 通过系统变量 tidb_snapshot 读取历史数据

本文档介绍如何通过系统变量 tidb_snapshot 读取历史数据，包括具体的操作流程以及历史数据的保存策略。

> 注意：
>
> 你还可以使用Stale Read 功能读取历史数据。更推荐使用 Stale Read 读取历史数据。

12.4.2.1 功能说明

TiDB 实现了通过标准 SQL 接口读取历史数据功能，无需特殊的 client 或者 driver。当数据被更新、删除后，依然可以通过 SQL 接口将更新/删除前的数据读取出来。

> 注意：
>
> 读取历史数据时，即使当前数据的表结构相较于历史数据的表结构已经发生改变，历史数据也会使用当时的表结构来返回数据。

12.4.2.2 操作流程

为支持读取历史版本数据，TiDB 引入了一个新的系统变量`tidb_snapshot`：

- 这个变量的作用域为 SESSION。
- 你可以通过标准的 SET 语句修改这个变量的值。
- 这个变量的数据类型为文本类型，能够存储 TSO 和日期时间。TSO 是从 PD 端获取的全局授时的时间戳，日期时间的格式为：“2016-10-08 16:45:26.999”，一般来说可以只写到秒，比如”2016-10-08 16:45:26”。
- 当这个变量被设置时，TiDB 会按照设置的时间戳建立 Snapshot（没有开销，只是创建数据结构），随后所有的 SELECT 操作都会从这个 Snapshot 上读取数据。

> 注意：
>
> TiDB 的事务是通过 PD 进行全局授时，所以存储的数据版本也是以 PD 所授时间戳作为版本号。在生成 Snapshot 时，是以 tidb_snapshot 变量的值作为版本号，如果 TiDB Server 所在机器和 PD Server 所在机器的本地时间相差较大，需要以 PD 的时间为准。

当读取历史版本操作结束后，可以结束当前 Session 或者是通过 SET 语句将 tidb_snapshot 变量的值设为 “”，即可读取最新版本的数据。

12.4.2.3 历史数据保留策略

TiDB 使用 MVCC 管理版本，当更新/删除数据时，不会做真正的数据删除，只会添加一个新版本数据，所以可以保留历史数据。历史数据不会全部保留，超过一定时间的历史数据会被彻底删除，以减小空间占用以及避免历史版本过多引入的性能开销。

TiDB 使用周期性运行的 GC（Garbage Collection，垃圾回收）来进行清理，关于 GC 的详细介绍参见TiDB 垃圾回收 (GC)。

这里需要重点关注的是：

- 使用系统变量tidb_gc_life_time 可以配置历史版本的保留时间（默认值是 10m0s）。
- 使用 SQL 语句 SELECT * FROM mysql.tidb WHERE variable_name = 'tikv_gc_safe_point' 可以查询当前的 safePoint，即当前可以读的最旧的快照。在每次 GC 开始运行时，safePoint 将自动更新。

12.4.2.4 示例

1. 初始化阶段，创建一个表，并插入几行数据：

```
create table t (c int);
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
insert into t values (1), (2), (3);
```

```
Query OK, 3 rows affected (0.00 sec)
```

2. 查看表中的数据：

```
select * from t;
```

```
+------+
| c    |
+------+
|    1 |
|    2 |
|    3 |
+------+
3 rows in set (0.00 sec)
```

3. 查看当前时间：

```
select now();
```

```
+---------------------+
| now()               |
+---------------------+
| 2016-10-08 16:45:26 |
+---------------------+
1 row in set (0.00 sec)
```

4. 更新某一行数据：

```
update t set c=22 where c=2;
```

```
Query OK, 1 row affected (0.00 sec)
```

5. 确认数据已经被更新：

```
select * from t;
```

```
+------+
| c    |
+------+
|    1 |
|   22 |
|    3 |
+------+
3 rows in set (0.00 sec)
```

6. 设置一个特殊的环境变量，这个是一个 session scope 的变量，其意义为读取这个时间之前的最新的一个版本。

```
set @@tidb_snapshot="2016-10-08 16:45:26";
```

```
Query OK, 0 rows affected (0.00 sec)
```

> 注意：
>
> - 这里的时间设置的是 update 语句之前的那个时间。
> - 在 `tidb_snapshot` 前须使用 `@@` 而非 `@`，因为 `@@` 表示系统变量，`@` 表示用户变量。

这里读取到的内容即为 update 之前的内容，也就是历史版本：

```
select * from t;
```

```
+------+
| c    |
+------+
|    1 |
|    2 |
|    3 |
+------+
3 rows in set (0.00 sec)
```

7. 清空这个变量后，即可读取最新版本数据：

```
set @@tidb_snapshot="";
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
select * from t;
```

```
+------+
| c    |
+------+
|    1 |
|   22 |
|    3 |
+------+
3 rows in set (0.00 sec)
```

> **注意：**
>
> 在 tidb_snapshot 前须使用 @@ 而非 @，因为 @@ 表示系统变量，@ 表示用户变量。

12.4.2.5 历史数据恢复策略

在恢复历史版本的数据之前，需要确保在对数据进行操作时，垃圾回收机制 (GC) 不会清除历史数据。如下所示，可以通过设置 tidb_gc_life_time 变量来调整 GC 清理的周期。不要忘记在恢复历史数据后将该变量设置回之前的值。

```
SET GLOBAL tidb_gc_life_time="60m";
```

> **注意：**
>
> 将 GC life time 从默认的 10 分钟增加到半小时及以上，会导致同一行保留有多个版本并占用更多的磁盘空间，也可能会影响某些操作的性能，例如扫描。进行扫描操作时，TiDB 读取数据需要跳过这些有多个版本的同一行，从而影响到扫描性能。

如果想要恢复历史版本的数据，可以使用以下任意一种方法进行设置：

- 对于简单场景，在设置 tidb_snapshot 变量后使用SELECT 语句并复制粘贴输出结果，或者使用 SELECT ... INTO OUTFILE 语句并使用LOAD DATA 语句来导入数据。
- 使用Dumpling 导出 TiDB 的历史数据快照。Dumpling 在导出较大的数据集时有较好的性能。

12.5 最佳实践

12.5.1 TiDB 最佳实践

本文档总结使用 TiDB 时的一些最佳实践，主要涉及 SQL 使用和 OLAP/OLTP 优化技巧，特别是一些 TiDB 专有的优化开关。

建议先阅读讲解 TiDB 原理的三篇文章（讲存储，说计算，谈调度），再来看这篇文章。

12.5.1.1 前言

数据库是一个通用的基础组件，在开发过程中会考虑到多种目标场景，在具体的业务场景中，需要根据业务的实际情况对数据的参数或者使用方式进行调整。

TiDB 是一个兼容 MySQL 协议和语法的分布式数据库，但是由于其内部实现，特别是支持分布式存储以及分布式事务，使得一些使用方法和 MySQL 有所区别。

12.5.1.2 基本概念

TiDB 的最佳实践与其实现原理密切相关，建议读者先了解一些基本的实现机制，包括 Raft、分布式事务、数据分片、负载均衡、SQL 到 KV 的映射方案、二级索引的实现方法、分布式执行引擎。下面会做一点简单的介绍，更详细的信息可以参考 PingCAP 公众号以及知乎专栏的一些文章。

12.5.1.2.1 Raft

Raft 是一种一致性协议，能提供强一致的数据复制保证，TiDB 最底层用 Raft 来同步数据。每次写入都要写入多数副本，才能对外返回成功，这样即使丢掉少数副本，也能保证系统中还有最新的数据。比如最大 3 副本的话，每次写入 2 副本才算成功，任何时候，只丢失一个副本的情况下，存活的两个副本中至少有一个具有最新的数据。

相比 Master-Slave 方式的同步，同样是保存三副本，Raft 的方式更为高效，写入的延迟取决于最快的两个副本，而不是最慢的那个副本。所以使用 Raft 同步的情况下，异地多活成为可能。在典型的两地三中心场景下，每次写入只需要本数据中心以及离得近的一个数据中心写入成功就能保证数据的一致性，而并不需要三个数据中心都写成功。但是这并不意味着在任何场景都能构建跨机房部署的业务，当写入量比较大时候，机房之间的带宽和延迟成为关键因素，如果写入速度超过机房之间的带宽，或者是机房之间延迟过大，整个 Raft 同步机制依然无法很好的运转。

12.5.1.2.2 分布式事务

TiDB 提供完整的分布式事务，事务模型是在 Google Percolator 的基础上做了一些优化。具体的实现可以参考《Percolator 和 TiDB 事务算法》这篇文章。本文档只讨论以下几点：

- 乐观锁

  TiDB 的乐观事务模型，只有在真正提交的时候，才会做冲突检测。如果有冲突，则需要重试。这种模型在冲突严重的场景下，会比较低效，因为重试之前的操作都是无效的，需要重复做。举一个比较极端的例子，就是把数据库当做计数器用，如果访问的并发度比较高，那么一定会有严重的冲突，导致大量的重试甚至是超时。但是如果访问冲突并不十分严重，那么乐观锁模型具备较高的效率。在冲突严重的场景下，推荐使用悲观锁，或在系统架构层面解决问题，比如将计数器放在 Redis 中。

- 悲观锁

  TiDB 的悲观事务模式，悲观事务的行为和 MySQL 基本一致，在执行阶段就会上锁，先到先得，避免冲突情况下的重试，可以保证有较多冲突的事务的成功率。悲观锁同时解决了希望通过 `select for update` 对数据提前锁定的场景。但如果业务场景本身冲突较少，乐观锁的性能会更有优势。

- 事务大小限制

  由于分布式事务要做两阶段提交，并且底层还需要做 Raft 复制，如果一个事务非常大，会使得提交过程非常慢，并且会卡住下面的 Raft 复制流程。为了避免系统出现被卡住的情况，我们对事务的大小做了限制：

  - 单个事务包含的 SQL 语句不超过 5000 条（默认）
    * 单条 KV entry 不超过 6MB（默认）
    * KV entry 的总大小不超过 10G

  在 Google 的 Cloud Spanner 上面，也有类似的限制。

12.5.1.2.3 数据分片

TiKV 自动将底层数据按照 Key 的 Range 进行分片。每个 Region 是一个 Key 的范围，从 `StartKey` 到 EndKey 的左闭右开区间。Region 中的 Key-Value 总量超过一定值，就会自动分裂。这部分用户不需要担心。

12.5.1.2.4 负载均衡

PD 会根据整个 TiKV 集群的状态，对集群的负载进行调度。调度是以 Region 为单位，以 PD 配置的策略为调度逻辑，自动完成。

12.5.1.2.5 SQL on KV

TiDB 自动将 SQL 结构映射为 KV 结构。具体的可以参考《三篇文章了解 TiDB 技术内幕 - 说计算》这篇文档。简单来说，TiDB 执行了以下操作：

- 一行数据映射为一个 KV，Key 以 `TableID` 构造前缀，以行 ID 为后缀
- 一条索引映射为一个 KV，Key 以 `TableID+IndexID` 构造前缀，以索引值构造后缀

可以看到，对于一个表中的数据或者索引，会具有相同的前缀，这样在 TiKV 的 Key 空间内，这些 Key-Value 会在相邻的位置。那么当写入量很大，并且集中在一个表上面时，就会造成写入的热点，特别是连续写入的数据中某些索引值也是连续的（比如 update time 这种按时间递增的字段），会在很少的几个 Region 上形成写入热点，成为整个系统的瓶颈。同样，如果所有的数据读取操作也都集中在很小的一个范围内（比如在连续的几万或者十几万行数据上），那么可能造成数据的访问热点。

12.5.1.2.6 二级索引

TiDB 支持完整的二级索引，并且是全局索引，很多查询可以通过索引来优化。如果利用好二级索引，对业务非常重要，很多 MySQL 上的经验在 TiDB 这里依然适用，不过 TiDB 还有一些自己的特点，需要注意，这一节主要讨论在 TiDB 上使用二级索引的一些注意事项。

- 二级索引是否越多越好

  二级索引能加速查询，但是要注意新增一个索引是有副作用的。上一节介绍了索引的存储模型，那么每增加一个索引，在插入一条数据的时候，就要新增一个 Key-Value，所以索引越多，写入越慢，并且空间占用越大。另外过多的索引也会影响优化器运行时间，并且不合适的索引会误导优化器。所以索引并不是越多越好。

- 对哪些列建索引比较合适

  上文提到，索引很重要但不是越多越好，因此需要根据具体的业务特点创建合适的索引。原则上需要对查询中需要用到的列创建索引，目的是提高性能。下面几种情况适合创建索引：

  - 区分度比较大的列，通过索引能显著地减少过滤后的行数
  - 有多个查询条件时，可以选择组合索引，注意需要把等值条件的列放在组合索引的前面

  这里举一个例子，假设常用的查询是 `select * from t where c1 = 10 and c2 = 100 and c3 > 10`，那么可以考虑建立组合索引 `Index cidx (c1, c2, c3)`，这样可以用查询条件构造出一个索引前缀进行 Scan。

- 通过索引查询和直接扫描 Table 的区别

  TiDB 实现了全局索引，所以索引和 Table 中的数据并不一定在一个数据分片上。通过索引查询的时候，需要先扫描索引，得到对应的行 ID，然后通过行 ID 去取数据，所以可能会涉及到两次网络请求，会有一定的性能开销。

  如果查询涉及到大量的行，那么扫描索引是并发进行，只要第一批结果已经返回，就可以开始去取 Table 的数据，所以这里是一个并行 + Pipeline 的模式，虽然有两次访问的开销，但是延迟并不会很大。

  以下情况不会涉及到两次访问的问题：

  - 索引中的列已经满足了查询需求。比如 Table `t` 上面的列 `c` 有索引，查询是 `select c from t where c > 10;`，这个时候，只需要访问索引，就可以拿到所需要的全部数据。这种情况称之为覆盖索引 (Covering Index)。所以如果很关注查询性能，可以将部分不需要过滤但是需要在查询结果中返回的列放入索引中，构造成组合索引，比如这个例子：`select c1, c2 from t where c1 > 10;`，要优化这个查询可以创建组合索引 `Index c12 (c1, c2)`。
  - 表的 Primary Key 是整数类型。在这种情况下，TiDB 会将 Primary Key 的值当做行 ID，所以如果查询条件是在 PK 上面，那么可以直接构造出行 ID 的范围，直接扫描 Table 数据，获取结果。

- 查询并发度

  数据分散在很多 Region 上，所以 TiDB 在做查询的时候会并发进行，默认的并发度比较保守，因为过高的并发度会消耗大量的系统资源，且对于 OLTP 类型的查询，往往不会涉及到大量的数据，较低的并发度已经可以满足需求。对于 OLAP 类型的 Query，往往需要较高的并发度。所以 TiDB 支持通过 System Variable 来调整查询并发度。

  - tidb_distsql_scan_concurrency

  在进行扫描数据的时候的并发度，这里包括扫描 Table 以及索引数据。

  - tidb_index_lookup_size

  如果是需要访问索引获取行 ID 之后再访问 Table 数据，那么每次会把一批行 ID 作为一次请求去访问 Table 数据，这个参数可以设置 Batch 的大小，较大的 Batch 会使得延迟增加，较小的 Batch 可能会造成更多的查询次数。这个参数的合适大小与查询涉及的数据量有关。一般不需要调整。

- tidb_index_lookup_concurrency

  如果是需要访问索引获取行 ID 之后再访问 Table 数据，每次通过行 ID 获取数据时候的并发度通过这个参数调节。

- 通过索引保证结果顺序

  索引除了可以用来过滤数据之外，还能用来对数据排序，首先按照索引的顺序获取行 ID，然后再按照行 ID 的返回顺序返回行的内容，这样可以保证返回结果按照索引列有序。前面提到了扫索引和获取 Row 之间是并行 + Pipeline 模式，如果要求按照索引的顺序返回 Row，那么这两次查询之间的并发度设置的太高并不会降低延迟，所以默认的并发度比较保守。可以通过tidb_index_serial_scan_concurrency 变量进行并发度调整。

- 逆序索引

  目前 TiDB 支持对索引进行逆序 Scan，目前速度比顺序 Scan 慢一些，通常情况下慢 20%，在数据频繁修改造成版本较多的情况下，会慢的更多。如果可能，建议避免对索引的逆序 Scan。

12.5.1.3 场景与实践

上一节我们讨论了一些 TiDB 基本的实现机制及其对使用带来的影响，本节我们从具体的使用场景出发，谈一些更为具体的操作实践。我们以从部署到支撑业务这条链路为序，进行讨论。

12.5.1.3.1 部署

在部署之前请务必阅读TiDB 部署建议以及对硬件的需求。

推荐通过TiUP 部署 TiDB 集群，这个工具可以部署、停止、销毁、升级整个集群，非常方便易用。非常不推荐手动部署，后期的维护和升级会很麻烦。

12.5.1.3.2 导入数据

为了提高导入数据期间的写入性能，可以对 TiKV 的参数进行调优，具体的文档查看TiKV 性能参数调优。

12.5.1.3.3 写入

上面提到了 TiDB 对单个事务的大小有限制，这层限制是在 KV 层面，反映在 SQL 层面的话，简单来说一行数据会映射为一个 KV entry，每多一个索引，也会增加一个 KV entry。

> **注意：**
>
> 对事务的大小限制，要考虑 TiDB 做编码以及事务额外 Key 的开销，在使用的时候，建议每个事务的行数不超过 200 行，且单行数据小于 100k，否则可能性能不佳。

建议无论是 Insert，Update 还是 Delete 语句，都通过分 Batch 或者是加 Limit 的方式限制。

在删除大量数据的时候，建议使用 `Delete from t where xx limit 5000;` 这样的方案，通过循环来删除，用 `Affected Rows == 0` 作为循环结束条件。

如果一次删除的数据量非常大，这种循环的方式会越来越慢，因为每次删除都是从前向后遍历，前面的删除之后，短时间内会残留不少删除标记（后续会被 GC 清理掉），影响后面的 Delete 语句。如果有可能，建议把 Where 条件细化。举个例子，假设要删除 2017-05-26 当天的所有数据，那么可以这样做：

```
for i from 0 to 23:
    while affected_rows > 0:
        delete from t where insert_time >= i:00:00 and insert_time < (i+1):00:00 limit 5000;
        affected_rows = select affected_rows()
```

上面是一段伪代码，意思就是要把大块的数据拆成小块删除，以避免删除过程中前面的 Delete 语句影响后面的 Delete 语句。

12.5.1.3.4 查询

看业务的查询需求以及具体的语句，可以参考TiDB 专用系统变量和语法这篇文档。可以通过 SET 语句控制 SQL 执行的并发度，另外通过 Hint 控制 Join 物理算子选择。

另外 MySQL 标准的索引选择 Hint 语法，也可以用，通过 `Use Index/Ignore Index hint` 控制优化器选择索引。

如果是个 OLTP 和 OLAP 混合类型的业务，可以把 TP 请求和 AP 请求发送到不同的 tidb-server 上，这样能够减小 AP 业务对于 TP 业务的影响。承载 AP 业务的 tidb-server 推荐使用高配的机器，比如 CPU 核数比较多，内存比较大。

但彻底的隔离 OLTP 和 OLAP，推荐将 OLAP 的业务跑在 TiFlash 上。TiFlash 是列存引擎，在 OLAP 的分析查询场景上，性能极具亮点，TiFlash 可以在存储层上做到物理隔离，并可做到一致性读取。

12.5.1.3.5 监控和日志

Metrics 系统是了解系统状态的最佳方法，建议所有的用户都部署监控系统。

TiDB 使用 Grafana + Prometheus 监控系统状态。如果使用 TiUP 部署集群，那么会自动部署和配置监控系统。

监控系统中的监控项很多，大部分是给 TiDB 开发者查看的内容，如果没有对源代码比较深入的了解，并没有必要了解这些监控项。我们会精简出一些和业务相关或者是系统关键组件状态相关的监控项，放在一个独立的 `overview` 面板中，供用户使用。

除了监控之外，查看日志也是了解系统状态的常用方法。TiDB 的三个组件 tidb-server/tikv-server/pd-server 都有一个 `--log-file` 的参数。如果启动的时候设置了这个参数，那么日志会保存着参数所设置的文件的位置，另外会自动的按天对 Log 文件做归档。如果没有设置 `--log-file` 参数，日志会输出在 `stderr` 中。

从 4.0 版本开始，从解决易用性的角度出发，提供了TiDB Dashboard UI 系统，通过浏览器访问 `http://PD_IP:PD_PORT/dashboard` 即可打开 TiDB Dashboard。TiDB Dashboard 可以提供集群状态、性能分析、流量可视化、SQL 诊断、日志搜索等功能。

12.5.1.3.6 文档

了解一个系统或者解决使用中的问题最好的方法是阅读文档，明白实现原理。TiDB 有大量的官方文档，希望大家在遇到问题的时候能先尝试通过文档或者搜索 Issue list 寻找解决方案。官方文档查看 docs-cn。如果希望阅读英文文档，可以查看 docs。

其中的FAQ 和故障诊断章节建议大家仔细阅读。另外 TiDB 还有一些不错的工具，也有配套的文档，具体的见各项工具的 GitHub 页面。

除了文档之外，还有很多不错的文章介绍 TiDB 的各项技术细节内幕，大家可以关注下面这些文章发布渠道：

- 公众号：微信搜索 PingCAP
- 知乎专栏：TiDB 的后花园
- 官方博客

12.5.1.4 TiDB 的最佳适用场景

简单来说，TiDB 适合具备下面这些特点的场景：

- 数据量大，单机保存不下
- 不希望做 Sharding 或者懒得做 Sharding
- 访问模式上没有明显的热点
- 需要事务、需要强一致、需要灾备
- 希望 Real-Time HTAP，减少存储链路

12.5.2 开发 Java 应用使用 TiDB 的最佳实践

本文主要介绍如何开发 Java 应用程序以更好地使用 TiDB，包括开发中的常见问题与最佳实践。

12.5.2.1 Java 应用中的数据库相关组件

通常 Java 应用中和数据库相关的常用组件有：

- 网络协议：客户端通过标准 MySQL 协议和 TiDB 进行网络交互。
- JDBC API 及实现：Java 应用通常使用 JDBC (Java Database Connectivity) 来访问数据库。JDBC 定义了访问数据库 API，而 JDBC 实现完成标准 API 到 MySQL 协议的转换，常见的 JDBC 实现是 MySQL Connector/J，此外有些用户可能使用 MariaDB Connector/J。
- 数据库连接池：为了避免每次创建连接，通常应用会选择使用数据库连接池来复用连接，JDBC DataSource 定义了连接池 API，开发者可根据实际需求选择使用某种开源连接池实现。
- 数据访问框架：应用通常选择通过数据访问框架 (MyBatis, Hibernate) 的封装来进一步简化和管理数据库访问操作。
- 业务实现：业务逻辑控制着何时发送和发送什么指令到数据库，其中有些业务会使用 Spring Transaction 切面来控制管理事务的开始和提交逻辑。

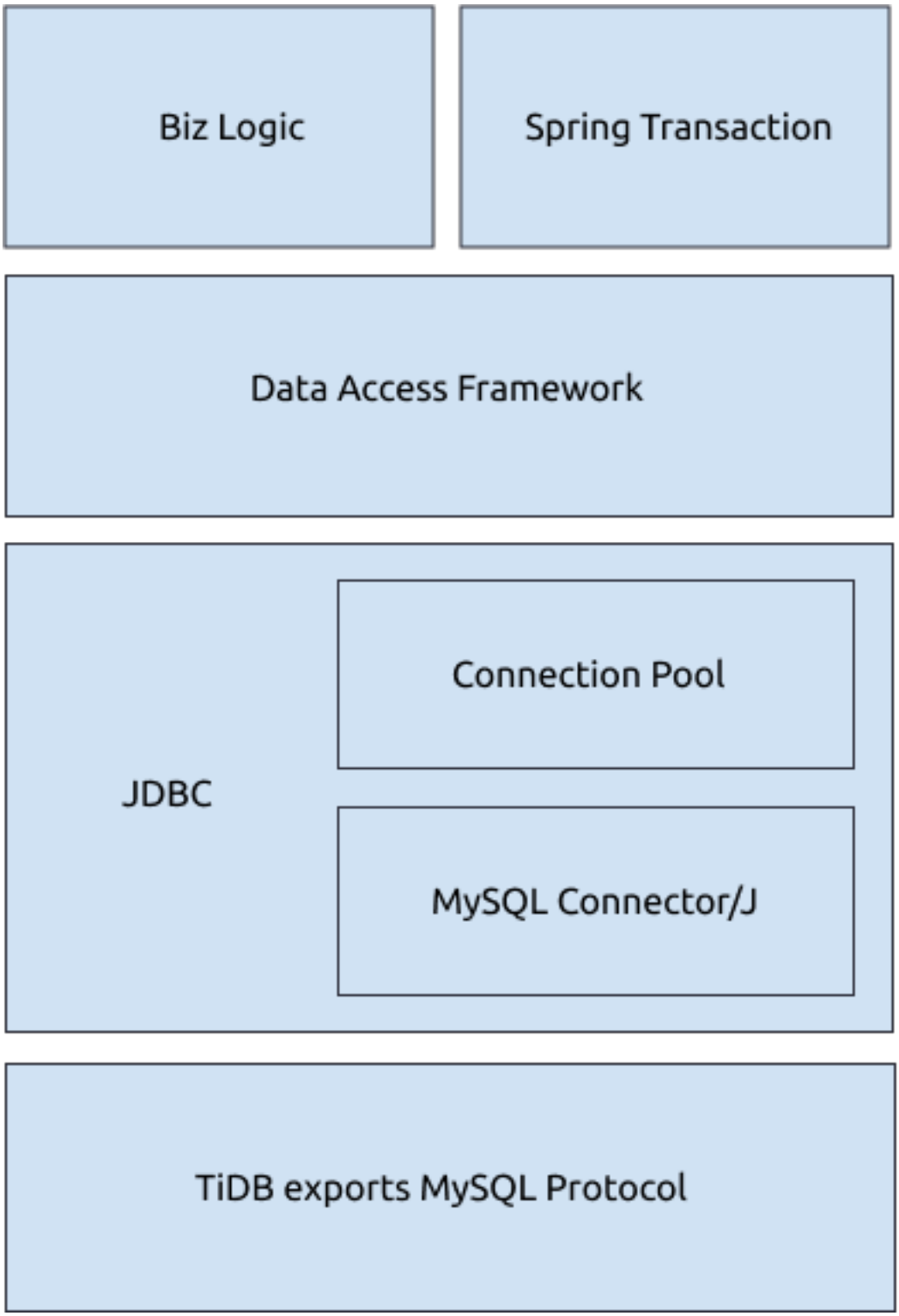


图 183: Java Component

如上图所示，应用可能使用 Spring Transaction 来管理控制事务非手工启停，通过类似 MyBatis 的数据访问框架管理生成和执行 SQL，通过连接池获取已池化的长连接，最后通过 JDBC 接口调用实现通过 MySQL 协议和 TiDB

完成交互。

接下来将分别介绍使用各个组件时可能需要关注的问题。

12.5.2.2 JDBC

Java 应用尽管可以选择在不同的框架中封装，但在最底层一般会通过调用 JDBC 来与数据库服务器进行交互。对于 JDBC，需要关注的主要有：API 的使用选择和 API Implementer 的参数配置。

12.5.2.2.1 JDBC API

对于基本的 JDBC API 使用可以参考 JDBC 官方教程，本文主要强调几个比较重要的 API 选择。

使用 Prepare API

对于 OLTP 场景，程序发送给数据库的 SQL 语句在去除参数变化后都是可穷举的某几类，因此建议使用预处理语句 (Prepared Statements) 代替普通的文本执行，并复用预处理语句来直接执行，从而避免 TiDB 重复解析和生成 SQL 执行计划的开销。

目前多数上层框架都会调用 Prepare API 进行 SQL 执行，如果直接使用 JDBC API 进行开发，注意选择使用 Prepare API。

另外需要注意 MySQL Connector/J 实现中默认只会做客户端的语句预处理，会将 ? 在客户端替换后以文本形式发送到服务端，所以除了要使用 Prepare API，还需要在 JDBC 连接参数中配置 `useServerPrepStmts = true`，才能在 TiDB 服务器端进行语句预处理（下面参数配置章节有详细介绍）。

使用 Batch 批量插入更新

对于批量插入更新，如果插入记录较多，可以选择使用 addBatch/executeBatch API。通过 addBatch 的方式将多条 SQL 的插入更新记录先缓存在客户端，然后在 executeBatch 时一起发送到数据库服务器。

> 注意：
>
> 对于 MySQL Connector/J 实现，默认 Batch 只是将多次 addBatch 的 SQL 发送时机延迟到调用 executeBatch 的时候，但实际网络发送还是会一条条的发送，通常不会降低与数据库服务器的网络交互次数。
>
> 如果希望 Batch 网络发送，需要在 JDBC 连接参数中配置 `rewriteBatchedStatements = true`（下面参数配置章节有详细介绍）。

使用 StreamingResult 流式获取执行结果

一般情况下，为提升执行效率，JDBC 会默认提前获取查询结果并将其保存在客户端内存中。但在查询返回超大结果集的场景中，客户端会希望数据库服务器减少向客户端一次返回的记录数，等客户端在有限内存处理完一部分后再去向服务器要下一批。

在 JDBC 中通常有以下两种处理方式：

- 设置 `FetchSize` 为 `Integer.MIN_VALUE` 让客户端不缓存，客户端通过 StreamingResult 的方式从网络连接上流式读取执行结果。

使用流式读取数据时，需要将 resultset 读取完成或 close 后，才能继续使用该语句进行下次查询，否则会报错 No statements may be issued when any streaming result sets are open and in use on a ↪ given connection. Ensure that you have called .close()on any active streaming result sets ↪ before attempting more queries.。

如果需要在 resultset 读取完成或 close 前进行查询避免上述报错，可在 URL 中添加配置参数 clobberStreamingResults=true，这样会自动 close resultset，但之前流式查询未被读取的结果集会丢失。

- 使用 Cursor Fetch，首先需设置 FetchSize 为正整数，且在 JDBC URL 中配置 useCursorFetch = true。

TiDB 中同时支持两种方式，但更推荐使用第一种将 FetchSize 设置为 Integer.MIN_VALUE 的方式，比第二种功能实现更简单且执行效率更高。

12.5.2.2.2 MySQL JDBC 参数

JDBC 实现通常通过 JDBC URL 参数的形式来提供实现相关的配置。这里以 MySQL 官方的 Connector/J 来介绍参数配置（如果使用的是 MariaDB，可以参考 MariaDB 的类似配置）。因为配置项较多，这里主要关注几个可能影响到性能的参数。

Prepare 相关参数

useServerPrepStmts

默认情况下，useServerPrepStmts 的值为 false，即尽管使用了 Prepare API，也只会在客户端做“prepare”。因此为了避免服务器重复解析的开销，如果同一条 SQL 语句需要多次使用 Prepare API，则建议设置该选项为 true。

在 TiDB 监控中可以通过 Query Summary > CPS By Instance 查看请求命令类型，如果请求中 COM_QUERY 被 COM_STMT_EXECUTE 或 COM_STMT_PREPARE 代替即生效。

cachePrepStmts

虽然 useServerPrepStmts = true 能让服务端执行预处理语句，但默认情况下客户端每次执行完后会 close 预处理语句，并不会复用，这样预处理的效率甚至不如文本执行。所以建议开启 useServerPrepStmts = true 后同时配置 cachePrepStmts = true，这会让客户端缓存预处理语句。

在 TiDB 监控中可以通过 Query Summary > CPS By Instance 查看请求命令类型，如果请求中 COM_STMT_EXECUTE 数目远远多于 COM_STMT_PREPARE 即生效。

另外，通过 useConfigs = maxPerformance 配置会同时配置多个参数，其中也包括 cachePrepStmts = true。

prepStmtCacheSqlLimit

在配置 cachePrepStmts 后还需要注意 prepStmtCacheSqlLimit 配置（默认为 256），该配置控制客户端缓存预处理语句的最大长度，超过该长度将不会被缓存。

在一些场景 SQL 的长度可能超过该配置，导致预处理 SQL 不能复用，建议根据应用 SQL 长度情况决定是否需要调大该值。

在 TiDB 监控中通过 Query Summary > CPS By Instance 查看请求命令类型，如果已经配置了 cachePrepStmts = true，但 COM_STMT_PREPARE 还是和 COM_STMT_EXECUTE 基本相等且有 COM_STMT_CLOSE，需要检查这个配置项是否设置得太小。

prepStmtCacheSize

prepStmtCacheSize 控制缓存的预处理语句数目（默认为 25），如果应用需要预处理的 SQL 种类很多且希望复用预处理语句，可以调大该值。

和上一条类似，在监控中通过 Query Summary > CPS By Instance 查看请求中 COM_STMT_EXECUTE 数目是否远远多于 COM_STMT_PREPARE 来确认是否正常。

Batch 相关参数

在进行 batch 写入处理时推荐配置 rewriteBatchedStatements = true，在已经使用 addBatch 或 executeBatch 后默认 JDBC 还是会一条条 SQL 发送，例如：

```
pstmt = prepare(“insert into t (a) values(?)”);
pstmt.setInt(1, 10);
pstmt.addBatch();
pstmt.setInt(1, 11);
pstmt.addBatch();
pstmt.setInt(1, 12);
pstmt.executeBatch();
```

虽然使用了 batch 但发送到 TiDB 语句还是单独的多条 insert：

```
insert into t(a) values(10);
insert into t(a) values(11);
insert into t(a) values(12);
```

如果设置 rewriteBatchedStatements = true，发送到 TiDB 的 SQL 将是：

```
insert into t(a) values(10),(11),(12);
```

需要注意的是，insert 语句的改写，只能将多个 values 后的值拼接成一整条 SQL，insert 语句如果有其他差异将无法被改写。例如：

```
insert into t (a) values (10) on duplicate key update a = 10;
insert into t (a) values (11) on duplicate key update a = 11;
insert into t (a) values (12) on duplicate key update a = 12;
```

上述 insert 语句将无法被改写成一条语句。该例子中，如果将 SQL 改写成如下形式：

```
insert into t (a) values (10) on duplicate key update a = values(a);
insert into t (a) values (11) on duplicate key update a = values(a);
insert into t (a) values (12) on duplicate key update a = values(a);
```

即可满足改写条件，最终被改写成：

```
insert into t (a) values (10), (11), (12) on duplicate key update a = values(a);
```

批量更新时如果有 3 处或 3 处以上更新，则 SQL 语句会改写为 multiple-queries 的形式并发送，这样可以有效减少客户端到服务器的请求开销，但副作用是会产生较大的 SQL 语句，例如这样：

```
update t set a = 10 where id = 1; update t set a = 11 where id = 2; update t set a = 12 where id
    ↪ = 3;
```

另外，因为一个客户端 bug，批量更新时如果要配置 `rewriteBatchedStatements = true` 和 `useServerPrepStmts ↪ = true`，推荐同时配置 `allowMultiQueries = true` 参数来避免这个 bug。

集成参数

通过监控可能会发现，虽然业务只向集群进行 insert 操作，却看到有很多多余的 select 语句。通常这是因为 JDBC 发送了一些查询设置类的 SQL 语句（例如 `select @@session.transaction_read_only`）。这些 SQL 对 TiDB 无用，推荐配置 `useConfigs = maxPerformance` 来避免额外开销。

`useConfigs = maxPerformance` 会包含一组配置，可查看 MySQL Connector/J 8.0 版本 或 5.1 版本 来确认当前 MySQL Connector/J 中 `maxPerformance` 包含的具体配置。

配置后查看监控，可以看到多余语句减少。

超时参数

TiDB 提供两个与 MySQL 兼容的超时控制参数，`wait_timeout` 和 `max_execution_time`。这两个参数分别控制与 Java 应用连接的空闲超时时间和连接中 SQL 执行的超时时间，即控制 TiDB 与 Java 应用的连接最长闲多久和最长忙多久。这两个参数的默认值都是 0，即默认允许连接无限闲置以及无限忙碌（一个 SQL 语句执行无限的长的时间）。

但在实际生产环境中，空闲连接和一直无限执行的 SQL 对数据库和应用都有不好的影响。你可以通过在应用的连接字符串中配置这两个参数来避免空闲连接和执行时间过长的 SQL 语句。例如，设置 `sessionVariables ↪ =wait_timeout=3600`（1 小时）和 `sessionVariables=max_execution_time=300000`（5 分钟）。

12.5.2.3 连接池

TiDB (MySQL) 连接建立是比较昂贵的操作（至少对于 OLTP），除了建立 TCP 连接外还需要进行连接鉴权操作，所以客户端通常会把 TiDB (MySQL) 连接保存到连接池中进行复用。

Java 的连接池实现很多 (HikariCP, tomcat-jdbc, druid, c3p0, dbcp)，TiDB 不会限定使用的连接池，应用可以根据业务特点自行选择连接池实现。

12.5.2.3.1 连接数配置

比较常见的是应用需要根据自身情况配置合适的连接池大小，以 HikariCP 为例：

- `maximumPoolSize`：连接池最大连接数，配置过大会导致 TiDB 消耗资源维护无用连接，配置过小则会导致应用获取连接变慢，所以需根据应用自身特点配置合适的值，可参考这篇文章。
- `minimumIdle`：连接池最小空闲连接数，主要用于在应用空闲时存留一些连接以应对突发请求，同样是需要根据业务情况进行配置。

应用在使用连接池时，需要注意连接使用完成后归还连接，推荐应用使用对应的连接池相关监控（如 `metricRegistry`），通过监控能及时定位连接池问题。

12.5.2.3.2 探活配置

连接池维护到 TiDB 的长连接，TiDB 默认不会主动关闭客户端连接（除非报错），但一般客户端到 TiDB 之间还会有 LVS 或 HAProxy 之类的网络代理，它们通常会在连接空闲一定时间（由代理的 idle 配置决定）后主动清理连接。除了注意代理的 idle 配置外，连接池还需要进行保活或探测连接。

如果常在 Java 应用中看到以下错误：

```
The last packet sent successfully to the server was 3600000 milliseconds ago. The driver has not
    ↪ received any packets from the server. com.mysql.jdbc.exceptions.jdbc4.
    ↪ CommunicationsException: Communications link failure
```

如果 `n milliseconds ago` 中的 n 如果是 0 或很小的值，则通常是执行的 SQL 导致 TiDB 异常退出引起的报错，推荐查看 TiDB stderr 日志；如果 n 是一个非常大的值（比如这里的 3600000），很可能是因为这个连接空闲太久然后被中间 proxy 关闭了，通常解决方式除了调大 proxy 的 idle 配置，还可以让连接池执行以下操作：

- 每次使用连接前检查连接是否可用。
- 使用单独线程定期检查连接是否可用。
- 定期发送 test query 保活连接。

不同的连接池实现可能会支持其中一种或多种方式，可以查看所使用的连接池文档来寻找对应配置。

12.5.2.4 数据访问框架

业务应用通常会使用某种数据访问框架来简化数据库的访问。

12.5.2.4.1 MyBatis

MyBatis 是目前比较流行的 Java 数据访问框架，主要用于管理 SQL 并完成结果集和 Java 对象的来回映射工作。MyBatis 和 TiDB 兼容性很好，从历史 issue 可以看出 MyBatis 很少出现问题。这里主要关注如下几个配置。

Mapper 参数

MyBatis 的 Mapper 中支持两种参数：

- `select 1 from t where id = #{param1}` 会作为预处理语句，被转换为 `select 1 from t where id = ?` 进行预处理，并使用实际参数来复用执行，通过配合前面的 Prepare 连接参数能获得最佳性能。
- `select 1 from t where id = ${param2}` 会做文本替换为 `select 1 from t where id = 1` 执行，如果这条语句被预处理为不同参数，可能会导致 TiDB 缓存大量的预处理语句，并且以这种方式执行 SQL 有注入安全风险。

动态 SQL Batch

动态 SQL - foreach

要支持将多条 insert 语句自动重写为 `insert ... values(...), (...), ...` 的形式，除了前面所说的在 JDBC 配置 `rewriteBatchedStatements = true` 外，MyBatis 还可以使用动态 SQL 来半自动生成 batch insert。比如下面的 mapper:

```
<insert id="insertTestBatch" parameterType="java.util.List" fetchSize="1">
  insert into test
   (id, v1, v2)
  values
  <foreach item="item" index="index" collection="list" separator=",">
  (
   #{item.id}, #{item.v1}, #{item.v2}
  )
  </foreach>
  on duplicate key update v2 = v1 + values(v1)
</insert>
```

会生成一个 `insert on duplicate key update` 语句，values 后面的 `(?, ?, ?)` 数目是根据传入的 list 个数决定，最终效果和使用 `rewriteBatchStatements = true` 类似，可以有效减少客户端和 TiDB 的网络交互次数，同样需要注意预处理后超过 `prepStmtCacheSqlLimit` 限制导致不缓存预处理语句的问题。

Streaming 结果

前面介绍了在 JDBC 中如何使用流式读取结果，除了 JDBC 相应的配置外，在 MyBatis 中如果希望读取超大结果集合也需要注意：

- 可以通过在 mapper 配置中对单独一条 SQL 设置 `fetchSize`（见上一段代码段），效果等同于调用 JDBC `setFetchSize`
- 可以使用带 `ResultHandler` 的查询接口来避免一次获取整个结果集
- 可以使用 `Cursor` 类来进行流式读取

对于使用 xml 配置映射，可以通过在映射 `<select>` 部分配置 `fetchSize="-2147483648"(Integer.MIN_VALUE)` 来流式读取结果。

```
<select id="getAll" resultMap="postResultMap" fetchSize="-2147483648">
  select * from post;
</select>
```

而使用代码配置映射，则可以使用 `@Options(fetchSize = Integer.MIN_VALUE)` 并返回 `Cursor` 从而让 SQL 结果能被流式读取。

```
@Select("select * from post")
@Options(fetchSize = Integer.MIN_VALUE)
Cursor<Post> queryAllPost();
```

12.5.2.4.2 ExecutorType

在 `openSession` 的时候可以选择 `ExecutorType`，MyBatis 支持三种 executor：

- Simple：每次执行都会向 JDBC 进行预处理语句的调用（如果 JDBC 配置有开启 `cachePrepStmts`，重复的预处理语句会复用）。

- Reuse：在 executor 中缓存预处理语句，这样不用 JDBC 的 cachePrepStmts 也能减少重复预处理语句的调用。
- Batch：每次更新只有在 addBatch 到 query 或 commit 时才会调用 executeBatch 执行，如果 JDBC 层开启了 rewriteBatchStatements，则会尝试改写，没有开启则会一条条发送。

通常默认值是 Simple，需要在调用 openSession 时改变 ExecutorType。如果是 Batch 执行，会遇到事务中前面的 update 或 insert 都非常快，而在读数据或 commit 事务时比较慢的情况，这实际上是正常的，在排查慢 SQL 时需要注意。

12.5.2.5 Spring Transaction

在应用代码中业务可能会通过使用 Spring Transaction 和 AOP 切面的方式来启停事务。

通过在方法定义上添加 @Transactional 注解标记方法，AOP 将会在方法前开启事务，方法返回结果前提交事务。如果遇到类似业务，可以通过查找代码 @Transactional 来确定事务的开启和关闭时机。需要特别注意有内嵌的情况，如果发生内嵌，Spring 会根据 Propagation 配置使用不同的行为。

12.5.2.6 其他

12.5.2.6.1 排查工具

在 Java 应用发生问题并且不知道业务逻辑情况下，使用 JVM 强大的排查工具会比较有用。这里简单介绍几个常用工具：

jstack

jstack 对应于 Go 中的 pprof/goroutine，可以比较方便地排查进程卡死的问题。

通过执行 `jstack pid`，即可输出目标进程中所有线程的线程 id 和堆栈信息。输出中默认只有 Java 堆栈，如果希望同时输出 JVM 中的 C++ 堆栈，需要加 `-m` 选项。

通过多次 jstack 可以方便地发现卡死问题（比如：都通过 Mybatis BatchExecutor flush 调用 update）或死锁问题（比如：测试程序都在抢占应用中某把锁导致没发送 SQL）

另外，`top -p $PID -H` 或者 Java swiss knife 都是常用的查看线程 ID 的方法。通过 `printf "%x\n" pid` 把线程 ID 转换成 16 进制，然后去 jstack 输出结果中找对应线程的栈信息，可以定位”某个线程占用 CPU 比较高，不知道它在执行什么”的问题。

jmap & mat

和 Go 中的 pprof/heap 不同，jmap 会将整个进程的内存快照 dump 下来（go 是分配器的采样），然后可以通过另一个工具 mat 做分析。

通过 mat 可以看到进程中所有对象的关联信息和属性，还可以观察线程运行的状态。比如：我们可以通过 mat 找到当前应用中有多少 MySQL 连接对象，每个连接对象的地址和状态信息是什么。

需要注意 mat 默认只会处理 reachable objects，如果要排查 young gc 问题可以在 mat 配置中设置查看 unreachable objects。另外对于调查 young gc 问题（或者大量生命周期较短的对象）的内存分配，用 Java Flight Recorder 比较方便。

trace

线上应用通常无法修改代码，又希望在 Java 中做动态插桩来定位问题，推荐使用 btrace 或 arthas trace。它们可以在不重启进程的情况下动态插入 trace 代码。

火焰图

Java 应用中获取火焰图较繁琐，可参阅 Java Flame Graphs Introduction: Fire For Everyone! 来手动获取。

12.5.2.7 总结

本文从常用的和数据库交互的 Java 组件的角度，阐述了开发 Java 应用程序使用 TiDB 的常见问题与解决办法。TiDB 是高度兼容 MySQL 协议的数据库，基于 MySQL 开发的 Java 应用的最佳实践也多适用于 TiDB。

欢迎大家在 ASK TUG 踊跃发言，和我们一起分享讨论 Java 应用使用 TiDB 的实践技巧或遇到的问题。

12.5.3 HAProxy 在 TiDB 中的最佳实践

本文介绍 HAProxy 在 TiDB 中的最佳配置和使用方法。HAProxy 提供 TCP 协议下的负载均衡能力，TiDB 客户端通过连接 HAProxy 提供的浮动 IP 即可对数据进行操作，实现 TiDB Server 层的负载均衡。

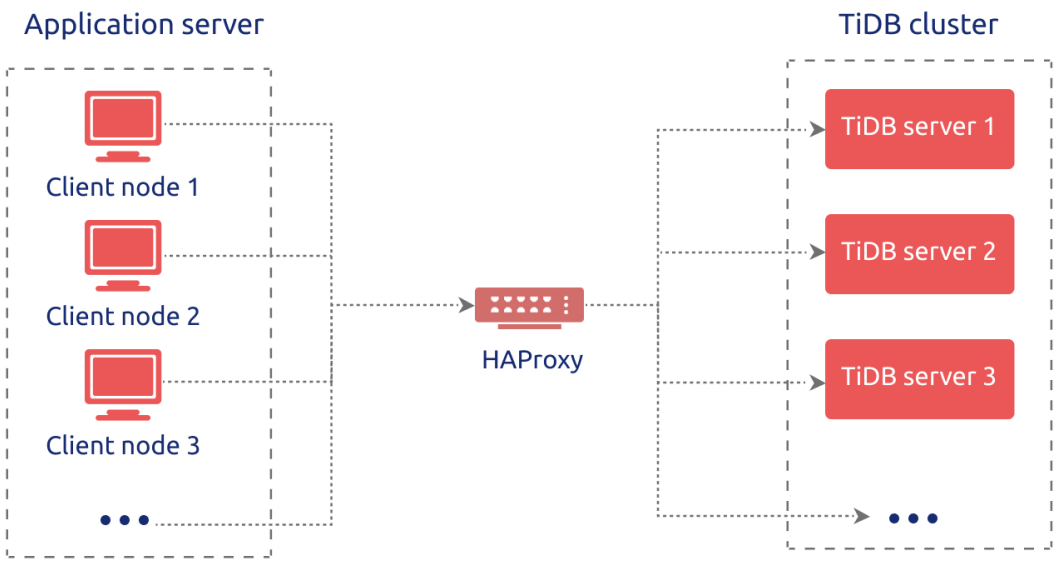


图 184: HAProxy 在 TiDB 中的最佳实践

> **注意：**
>
> TiDB 支持的最小 HAProxy 版本为 v1.5。使用 v1.5 到 v2.1 之间的 HAProxy 时，需要在 `mysql-check` 中配置 `post-41`。建议使用 HAProxy v2.2 或更高版本。

12.5.3.1 HAProxy 简介

HAProxy 是由 C 语言编写的自由开放源码的软件，为基于 TCP 和 HTTP 协议的应用程序提供高可用性、负载均衡和代理服务。因为 HAProxy 能够快速、高效使用 CPU 和内存，所以目前使用非常广泛，许多知名网站诸如 GitHub、Bitbucket、Stack Overflow、Reddit、Tumblr、Twitter 和 Tuenti 以及亚马逊网络服务系统都在使用 HAProxy。

HAProxy 由 Linux 内核的核心贡献者 Willy Tarreau 于 2000 年编写，他现在仍然负责该项目的维护，并在开源社区免费提供版本迭代。本文示例使用 HAProxy 2.6。推荐使用最新稳定版的 HAProxy，详情见已发布的 HAProxy 版本。

12.5.3.2 HAProxy 部分核心功能介绍

- 高可用性：HAProxy 提供优雅关闭服务和无缝切换的高可用功能；
- 负载均衡：L4 (TCP) 和 L7 (HTTP) 两种负载均衡模式，至少 9 类均衡算法，比如 roundrobin，leastconn，random 等；
- 健康检查：对 HAProxy 配置的 HTTP 或者 TCP 模式状态进行检查；
- 会话保持：在应用程序没有提供会话保持功能的情况下，HAProxy 可以提供该项功能；
- SSL：支持 HTTPS 通信和解析；
- 监控与统计：通过 web 页面可以实时监控服务状态以及具体的流量信息。

12.5.3.3 准备环境

在部署 HAProxy 之前，需准备好以下环境。

12.5.3.3.1 硬件要求

根据官方文档，对 HAProxy 的服务器硬件配置有以下建议，也可以根据负载均衡环境进行推算，在此基础上提高服务器配置。

硬件资源	最低配置
CPU	2 核，3.5 GHz
内存	16 GB
存储容量	50 GB（SATA 盘）
网卡	万兆网卡

12.5.3.3.2 依赖软件

根据 HAProxy 官方文档，对操作系统和依赖包有以下建议，如果通过 yum 源部署安装 HAProxy 软件，依赖包无需单独安装。

操作系统

Linux 操作系统	版本
Red Hat Enterprise Linux	7 或者 8
CentOS	7 或者 8
Oracle Enterprise Linux	7 或者 8
Ubuntu LTS	18.04 或者以上版本

> 注意：
>
> - 其他操作系统支持情况，详见 HAProxy 文档。

依赖包

- epel-release
- gcc
- systemd-devel

执行如下命令安装依赖包：

```
yum -y install epel-release gcc systemd-devel
```

12.5.3.4 部署 HAProxy

HAProxy 配置 Database 负载均衡场景操作简单，以下部署操作具有普遍性，不具有特殊性，建议根据实际场景，个性化配置相关的配置文件。

12.5.3.4.1 安装 HAProxy

1. 下载 HAProxy 2.6.2 的源码包：

   ```
   wget https://www.haproxy.org/download/2.6/src/haproxy-2.6.2.tar.gz
   ```

2. 解压源码包：

   ```
   tar zxf haproxy-2.6.2.tar.gz
   ```

3. 从源码编译 HAProxy 应用：

   ```
   cd haproxy-2.6.2
   make clean
   make -j 8 TARGET=linux-glibc USE_THREAD=1
   make PREFIX=${/app/haproxy} SBINDIR=${/app/haproxy/bin} install  # 将 `${/app/haproxy}` 和 `
       ↪ ${/app/haproxy/bin}` 替换为自定义的实际路径。
   ```

4. 重新配置 `profile` 文件：

   ```
   echo 'export PATH=/app/haproxy/bin:$PATH' >> /etc/profile
   . /etc/profile
   ```

5. 检查 HAProxy 是否安装成功：

```
which haproxy
```

HAProxy 命令介绍

执行如下命令查看命令行参数及基本用法：

```
haproxy --help
```

参数	说明
-v	显示简略的版本信息。
-vv	显示详细的版本信息。
-d	开启 debug 模式。
-db	禁用后台模式和多进程模式。
-dM [<byte >]	执行分配内存。
-V	启动过程显示配置和轮询信息。
-D	开启守护进程模式。
-C <dir>	在加载配置文件之前更改目录位置至 <dir>。
-W	主从模式。
-q	静默模式，不输出信息。
-c	只检查配置文件并在尝试绑定之前退出。
-n <limit>	设置每个进程的最大总连接数为 <limit>。
-m <limit>	设置所有进程的最大可用内存为 <limit> (单位：MB)。

参数	说明
-N <limit>	设置单点最大连接数为 <limit>，默认为 2000。
-L <name>	将本地实例对等名称改为 <name>，默认为本地主机名。
-p <file>	将 HAProxy 所有子进程的 PID 信息写入 <file>。
-de	禁止使用 epoll(7)，epoll(7) 仅在 Linux 2.6 和某些定制的 Linux 2.4 系统上可用。
-dp	禁止使用 epoll(2)，可改用 select(2)。
-dS	禁止使用 splice(2)，splice(2) 在一些旧版 Linux 内核上不可用。
-dR	禁止使用 SO_REUSEPORT。
-dr	忽略服务器地址解析失败。
-dV	禁止在服务器端使用 SSL。

参数	说明
`-sf <pidlist>`	启动后，向 pidlist 中的 PID 发送 `finish` 信号，收到此信号的进程在退出之前将等待所有会话完成，即优雅停止服务。此选项必须最后指定，后跟任意数量的 PID。从技术上讲，SIGTTOU 和 SIGUSR1 都被发送。
`-st <pidlist>`	启动后，向 pidlist 中的 PID 发送 `terminate` 信号，收到此信号的进程将立即终止，关闭所有活动会话。此选项必须最后指定，后跟任意数量的 PID。从技术上讲，SIGTTOU 和 SIGTERM 都被发送。
`-x <unix_socket>`	连接指定的 socket 并从旧进程中获取所有 listening socket，然后，使用这些 socket 而不是绑定新的。

参数	说明
-S <bind >[,< bind_option >...]	主从模式下，创建绑定到主进程的 socket，此 socket 可访问每个子进程的 socket。

更多有关 HAProxy 命令参数的信息，可参阅 Management Guide of HAProxy 和 General Commands Manual of HAProxy。

12.5.3.4.2 配置 HAProxy

yum 安装过程中会生成配置模版，你也可以根据实际场景自定义配置如下配置项。

```
global                                  # 全局配置。
   log         127.0.0.1 local2         # 定义全局的 syslog 服务器，最多可以定义两个。
   chroot      /var/lib/haproxy         # 更改当前目录并为启动进程设置超级用户权限，
       ↪ 从而提高安全性。
   pidfile     /var/run/haproxy.pid     # 将 HAProxy 进程的 PID 写入 pidfile。
   maxconn     4096                     # 单个 HAProxy 进程可接受的最大并发连接数，
       ↪ 等价于命令行参数 "-n"。
   nbthread    48                       # 最大线程数。线程数的上限与 CPU 数量相同。
   user        haproxy                  # 同 UID 参数。
   group       haproxy                  # 同 GID 参数，建议使用专用用户组。
   daemon                               # 让 HAProxy 以守护进程的方式工作于后台，
       ↪ 等同于命令行参数“-D”的功能。当然，也可以在命令行中用“-db”参数将其禁用。
   stats socket /var/lib/haproxy/stats  # 统计信息保存位置。

defaults                                # 默认配置。
   log global                           # 日志继承全局配置段的设置。
   retries 2                            # 向上游服务器尝试连接的最大次数，
       ↪ 超过此值便认为后端服务器不可用。
   timeout connect  2s                  # HAProxy 与后端服务器连接超时时间。
       ↪ 如果在同一个局域网内，可设置成较短的时间。
   timeout client 30000s                # 客户端与 HAProxy 连接后，数据传输完毕，
       ↪ 即非活动连接的超时时间。
   timeout server 30000s                # 服务器端非活动连接的超时时间。

listen admin_stats                      # frontend 和 backend 的组合体，
    ↪ 此监控组的名称可按需进行自定义。
   bind 0.0.0.0:8080                    # 监听端口。
   mode http                            # 监控运行的模式，此处为 `http` 模式。
   option httplog                       # 开始启用记录 HTTP 请求的日志功能。
   maxconn 10                           # 最大并发连接数。
   stats refresh 30s                    # 每隔 30 秒自动刷新监控页面。
```

```
   stats uri /haproxy                      # 监控页面的 URL。
   stats realm HAProxy                     # 监控页面的提示信息。
   stats auth admin:pingcap123             # 监控页面的用户和密码，可设置多个用户名。
   stats hide-version                      # 隐藏监控页面上的 HAProxy 版本信息。
   stats  admin if TRUE                    # 手工启用或禁用后端服务器（HAProxy 1.4.9
       ↪ 及之后版本开始支持）。

listen tidb-cluster                        # 配置 database 负载均衡。
   bind 0.0.0.0:3390                       # 浮动 IP 和 监听端口。
   mode tcp                                # HAProxy 要使用第 4 层的传输层。
   balance leastconn                       # 连接数最少的服务器优先接收连接。`leastconn`
       ↪ 建议用于长会话服务，例如 LDAP、SQL、TSE 等，而不是短会话协议，如 HTTP。该算法是动态的
       ↪ ，对于启动慢的服务器，服务器权重会在运行中作调整。
   server tidb-1 10.9.18.229:4000 check inter 2000 rise 2 fall 3       # 检测 4000 端口，
       ↪ 检测频率为每 2000 毫秒一次。如果 2 次检测为成功，则认为服务器可用；如果 3 次检测为失败
       ↪ ，则认为服务器不可用。
   server tidb-2 10.9.39.208:4000 check inter 2000 rise 2 fall 3
   server tidb-3 10.9.64.166:4000 check inter 2000 rise 2 fall 3
```

如要通过 SHOW PROCESSLIST 查看连接来源 IP，需要配置使用 PROXY 协议连接 TiDB。

```
   server tidb-1 10.9.18.229:4000 send-proxy check inter 2000 rise 2 fall 3
   server tidb-2 10.9.39.208:4000 send-proxy check inter 2000 rise 2 fall 3
   server tidb-3 10.9.64.166:4000 send-proxy check inter 2000 rise 2 fall 3
```

> **注意：**
>
> 使用 PROXY 协议时，你需要在 tidb-server 的配置文件中设置proxy-protocol.networks。

12.5.3.4.3 启动 HAProxy

要启动 HAProxy，执行 haproxy 命令。默认读取 /etc/haproxy/haproxy.cfg（推荐）。

```
haproxy -f /etc/haproxy/haproxy.cfg
```

12.5.3.4.4 停止 HAProxy

要停止 HAProxy，使用 kill -9 命令。

1. 执行如下命令：

   ```
   ps -ef | grep haproxy
   ```

2. 终止 HAProxy 相关的 PID 进程：

```
kill -9 ${haproxy.pid}
```

12.5.4 TiDB 高并发写入场景最佳实践

在 TiDB 的使用过程中，一个典型场景是高并发批量写入数据到 TiDB。本文阐述了该场景中的常见问题，旨在给出一个业务的最佳实践，帮助读者避免因使用 TiDB 不当而影响业务开发。

12.5.4.1 目标读者

本文假设你已对 TiDB 有一定的了解，推荐先阅读 TiDB 原理相关的三篇文章（讲存储，说计算，谈调度），以及 TiDB Best Practice。

12.5.4.2 高并发批量插入场景

高并发批量插入的场景通常出现在业务系统的批量任务中，例如清算以及结算等业务。此类场景存在以下特点：

- 数据量大
- 需要短时间内将历史数据入库
- 需要短时间内读取大量数据

这就对 TiDB 提出了以下挑战：

- 写入/读取能力是否可以线性水平扩展
- 随着数据持续大并发写入，数据库性能是否稳定不衰减

对于分布式数据库来说，除了本身的基础性能外，最重要的就是充分利用所有节点能力，避免让单个节点成为瓶颈。

12.5.4.3 TiDB 数据分布原理

如果要解决以上挑战，需要从 TiDB 数据切分以及调度的原理开始讲起。这里只作简单说明，详情可参阅谈调度。

TiDB 以 Region 为单位对数据进行切分，每个 Region 有大小限制（默认 96M）。Region 的切分方式是范围切分。每个 Region 会有多副本，每一组副本，称为一个 Raft Group。每个 Raft Group 中由 Leader 负责执行这块数据的读 & 写（TiDB 支持Follower-Read）。Leader 会自动地被 PD 组件均匀调度在不同的物理节点上，用以均分读写压力。

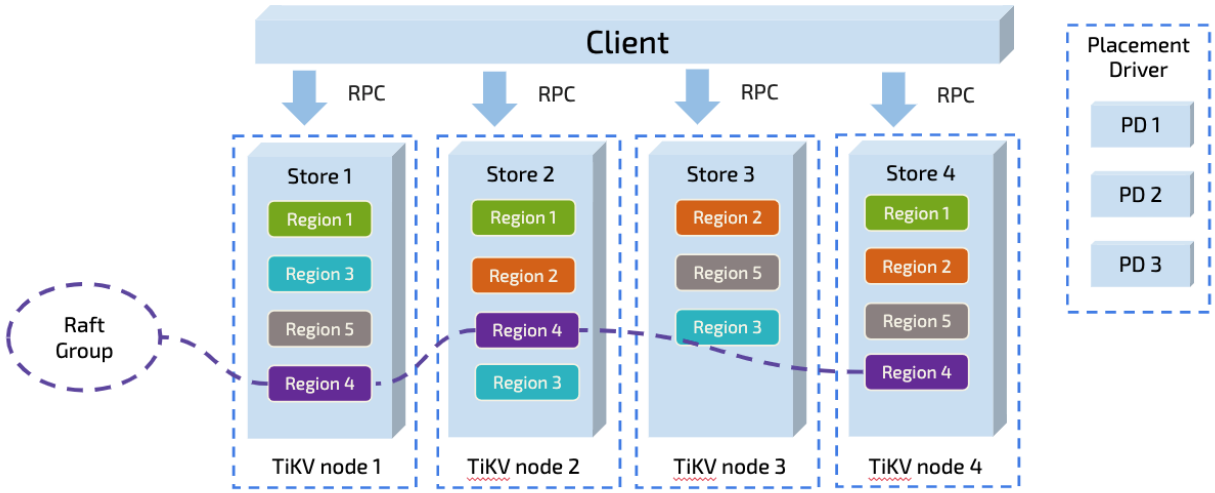


图 185: TiDB 数据概览

从原理上来说，只要没有业务上的写入热点（即业务写入没有 AUTO_INCREMENT 的主键和单调递增的索引，更多细节可参阅 TiDB 正确使用方式），依靠这个架构，TiDB 不仅具备线性扩展的读写能力，也能够充分利用分布式资源。从这一点看，TiDB 尤其适合高并发批量写入场景的业务。

但理论场景和实际情况往往存在不同。以下实例说明了热点是如何产生的。

12.5.4.4 热点产生的实例

以下为一张示例表：

```
CREATE TABLE IF NOT EXISTS TEST_HOTSPOT(
      id         BIGINT PRIMARY KEY,
      age        INT,
      user_name  VARCHAR(32),
      email      VARCHAR(128)
)
```

这个表的结构非常简单，除了 id 为主键以外，没有额外的二级索引。将数据写入该表的语句如下，id 通过随机数离散生成：

```
SET SESSION cte_max_recursion_depth = 1000000;
INSERT INTO TEST_HOTSPOT
SELECT
  n,                                        -- ID
  RAND()*80,                                -- 0 到 80 之间的随机数
  CONCAT('user-',n),
  CONCAT(
    CHAR(65 + (RAND() * 25) USING ascii),   -- 65 到 65+25 之间的随机数，转换为一个 A-Z 字符
    '-user-',
```

```
    n,
    '@example.com'
  )
FROM
  (WITH RECURSIVE nr(n) AS
    (SELECT 1                              -- 从 1 开始 CTE
      UNION ALL SELECT n + 1               -- 每次循环 n 增加 1
      FROM nr WHERE n < 1000000            -- 当 n 为 1_000_000 时停止循环
    ) SELECT n FROM nr
  ) a;
```

负载是短时间内密集地执行以上写入语句。

以上操作看似符合理论场景中的 TiDB 最佳实践，业务上没有热点产生。只要有足够的机器，就可以充分利用 TiDB 的分布式能力。要验证是否真的符合最佳实践，可以在实验环境中进行测试。

部署拓扑 2 个 TiDB 节点，3 个 PD 节点，6 个 TiKV 节点。请忽略 QPS，因为测试只是为了阐述原理，并非 benchmark。

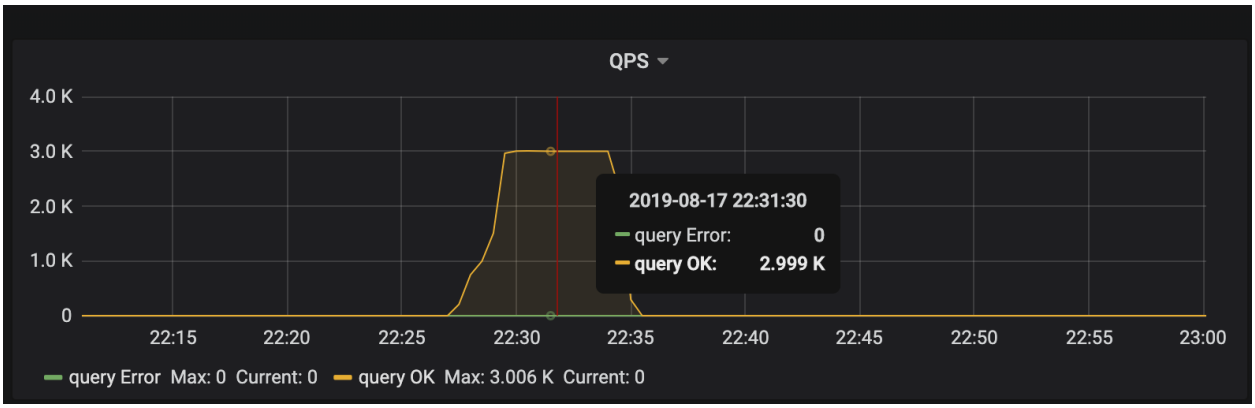


图 186: QPS1

客户端在短时间内发起了“密集”的写入，TiDB 收到的请求是 3K QPS。理论上，压力应该均摊给 6 个 TiKV 节点。但是从 TiKV 节点的 CPU 使用情况上看，存在明显的写入倾斜（tikv - 3 节点是写入热点）：

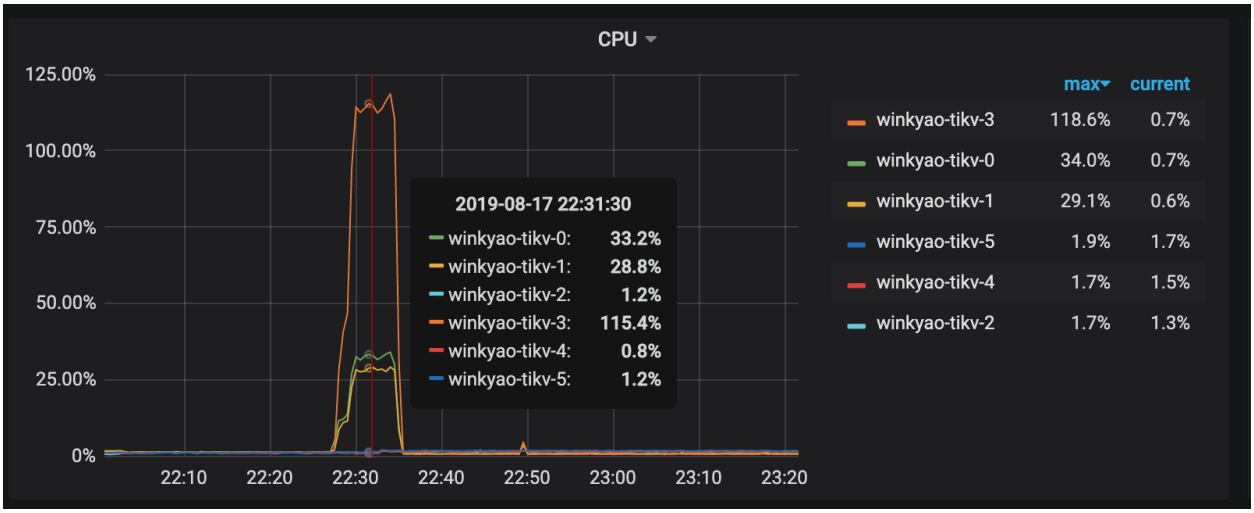


图 187: QPS2

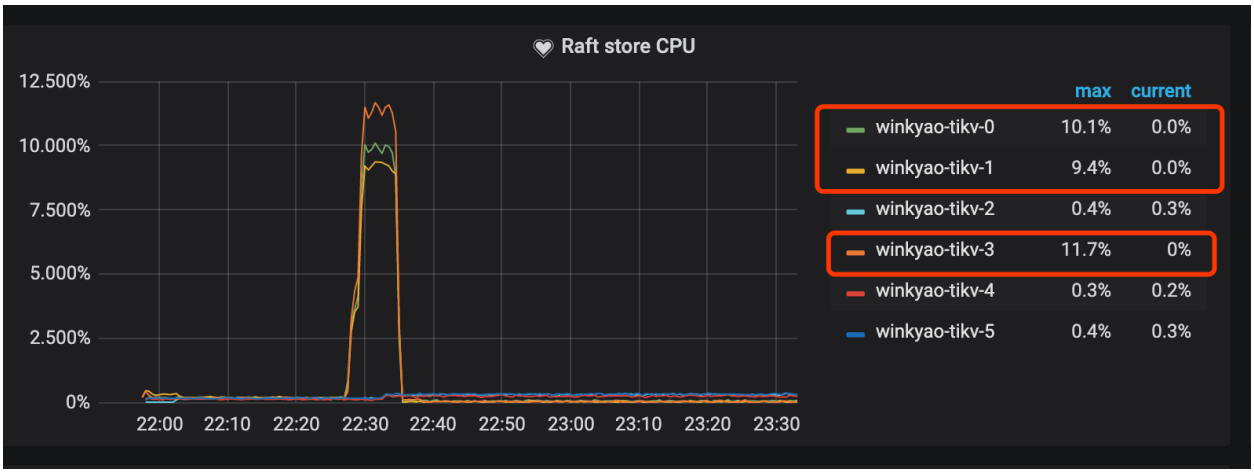


图 188: QPS3

Raft store CPU 为 `raftstore` 线程的 CPU 使用率，通常代表写入的负载。在这个场景下 tikv-3 为 Raft Leader，tikv-0 和 tikv-1 是 Raft 的 Follower，其他的 TiKV 节点的负载几乎为空。

从 PD 的监控中也可以证明热点的产生：

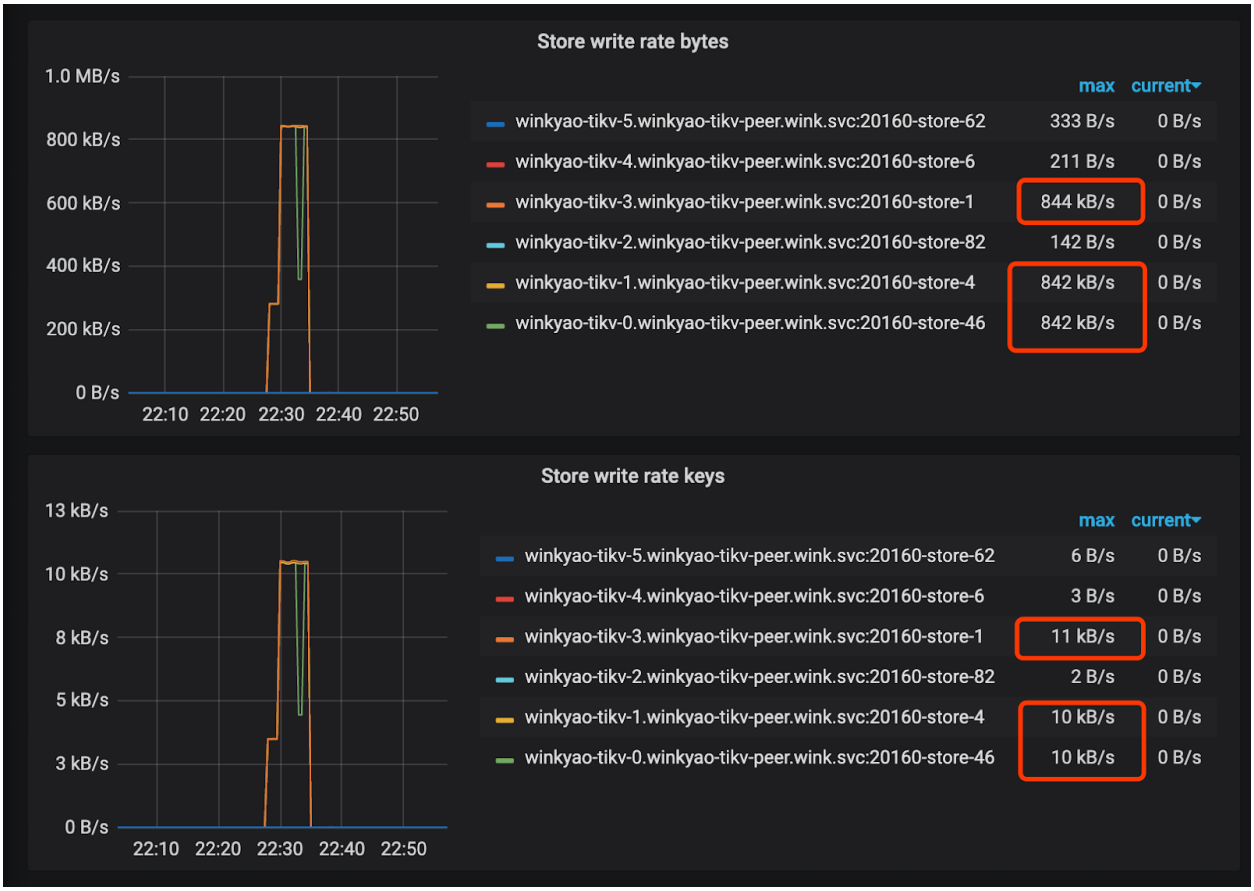


图 189: QPS4

12.5.4.5 热点问题产生的原因

以上测试并未达到理论场景中最佳实践，因为刚创建表的时候，这个表在 TiKV 中只会对应为一个 Region，范围是：

```
[CommonPrefix + TableID, CommonPrefix + TableID + 1)
```

短时间内大量数据会持续写入到同一个 Region 上。

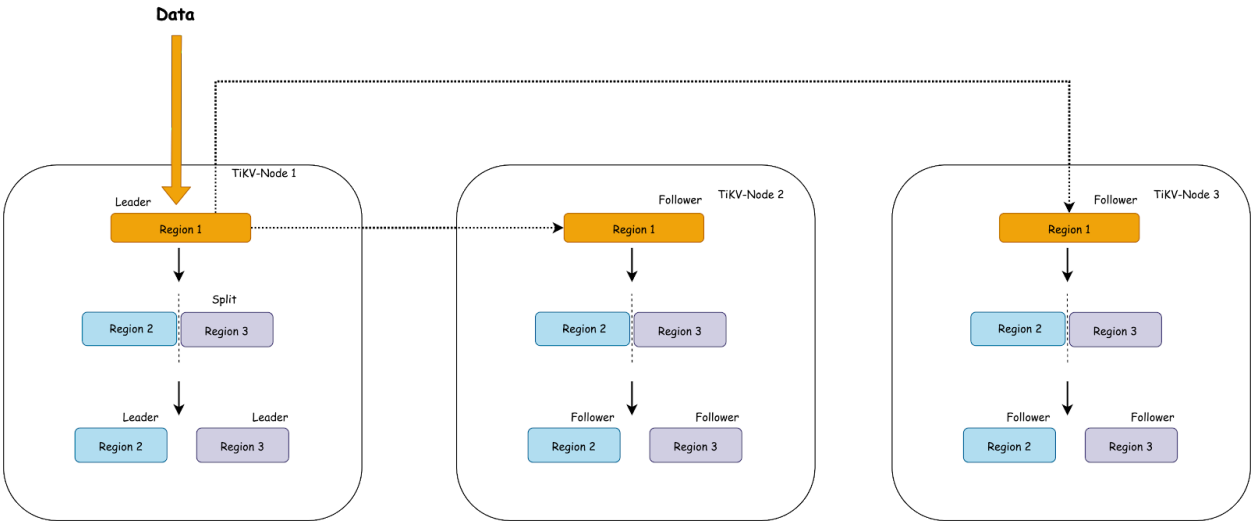


图 190: TiKV Region 分裂流程

上图简单描述了这个过程，随着数据持续写入，TiKV 会将一个 Region 切分为多个。但因为首先发起选举的是原 Leader 所在的 Store，所以新切分好的两个 Region 的 Leader 很可能还会在原 Store 上。新切分好的 Region 2，3 上，也会重复之前发生在 Region 1 上的过程。也就是压力会密集地集中在 TiKV-Node 1 上。

在持续写入的过程中，PD 发现 Node 1 中产生了热点，会将 Leader 均分到其他的 Node 上。如果 TiKV 的节点数多于副本数的话，TiKV 会尽可能将 Region 迁移到空闲的节点上。这两个操作在数据插入的过程中，也能在 PD 监控中得到印证：

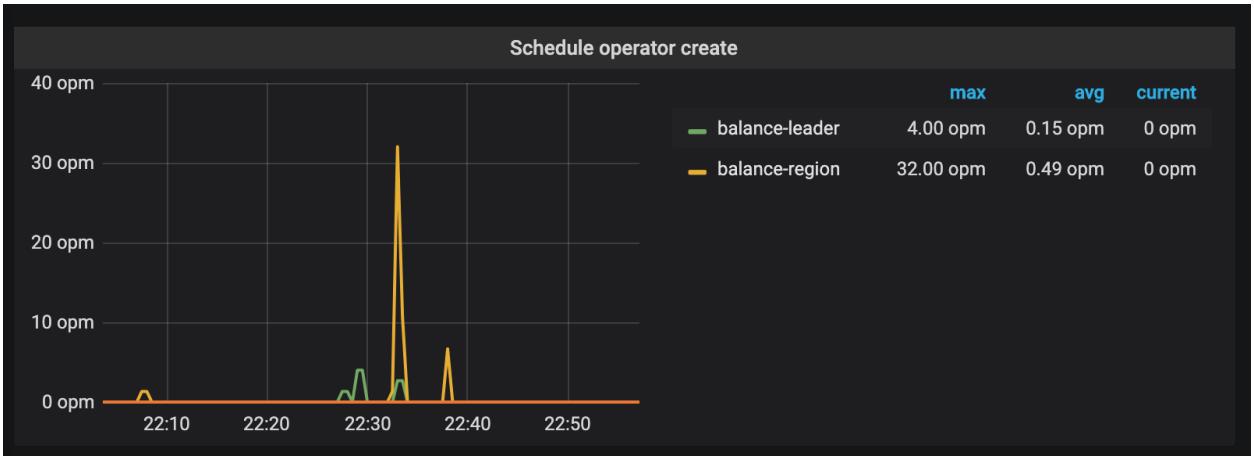


图 191: QPS5

在持续写入一段时间后，整个集群会被 PD 自动地调度成一个压力均匀的状态，到那个时候整个集群的能力才会真正被利用起来。在大多数情况下，以上热点产生的过程是没有问题的，这个阶段属于表 Region 的预热阶段。

但是对于高并发批量密集写入场景来说，应该避免这个阶段。

12.5.4.6 热点问题的规避方法

为了达到场景理论中的最佳性能，可跳过这个预热阶段，直接将 Region 切分为预期的数量，提前调度到集群的各个节点中。

TiDB 在 v3.0.x 以及 v2.1.13 后支持一个叫Split Region 的新特性。这个特性提供了新的语法：

```
SPLIT TABLE table_name [INDEX index_name] BETWEEN (lower_value) AND (upper_value) REGIONS
    ↪ region_num
```

```
SPLIT TABLE table_name [INDEX index_name] BY (value_list) [, (value_list)]
```

但是 TiDB 并不会自动提前完成这个切分操作。原因如下：

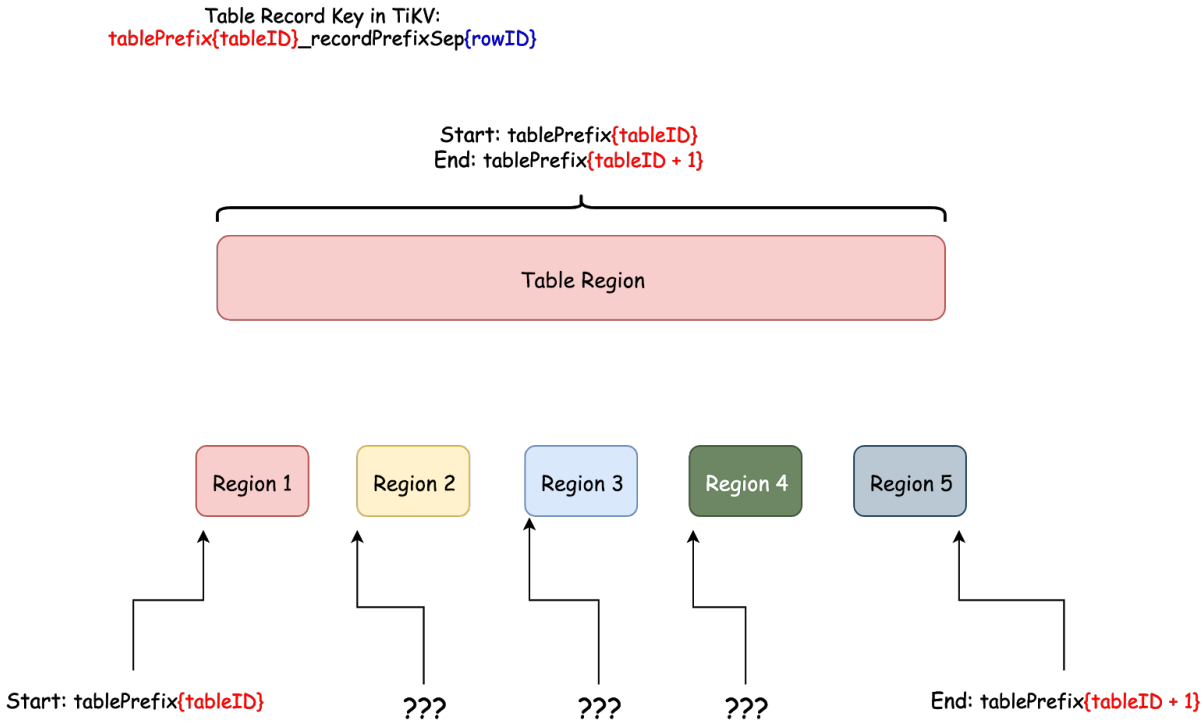


图 192: Table Region Range

从上图可知，根据行数据 key 的编码规则，行 ID (rowID) 是行数据中唯一可变的部分。在 TiDB 中，rowID 是一个 Int64 整型。但是用户不一定能将 Int64 整型范围均匀切分成需要的份数，然后均匀分布在不同的节点上，还需要结合实际情况。

如果行 ID 的写入是完全离散的，那么上述方式是可行的。如果行 ID 或者索引有固定的范围或者前缀（例如，只在 [2000w，5000w) 的范围内离散插入数据），这种写入依然在业务上不产生热点，但是如果按上面的方式进行切分，那么有可能一开始数据仍只写入到某个 Region 上。

作为一款通用数据库，TiDB 并不对数据的分布作假设，所以开始只用一个 Region 来对应一个表。等到真实数据插入进来以后，TiDB 自动根据数据的分布来作切分。这种方式是较通用的。

所以 TiDB 提供了 Split Region 语法，专门针对短时批量写入场景作优化。基于以上案例，下面尝试用 Split Region 语法提前切散 Region，再观察负载情况。

由于测试的写入数据在正数范围内完全离散，所以用以下语句，在 Int64 空间内提前将表切分为 128 个 Region：

```
SPLIT TABLE TEST_HOTSPOT BETWEEN (0) AND (9223372036854775807) REGIONS 128;
```

切分完成以后，可以通过 SHOW TABLE test_hotspot REGIONS; 语句查看打散的情况。如果 SCATTERING 列值全部为 0，代表调度成功。

也可以通过以下 SQL 语句查看 Region 的分布。你需要将 table_name 替换为实际的表名。

```
SELECT
    p.STORE_ID,
    COUNT(s.REGION_ID) PEER_COUNT
FROM
    INFORMATION_SCHEMA.TIKV_REGION_STATUS s
    JOIN INFORMATION_SCHEMA.TIKV_REGION_PEERS p ON s.REGION_ID = p.REGION_ID
WHERE
    TABLE_NAME = 'table_name'
    AND p.is_leader = 1
GROUP BY
    p.STORE_ID
ORDER BY
    PEER_COUNT DESC;
```

再重新运行写入负载：

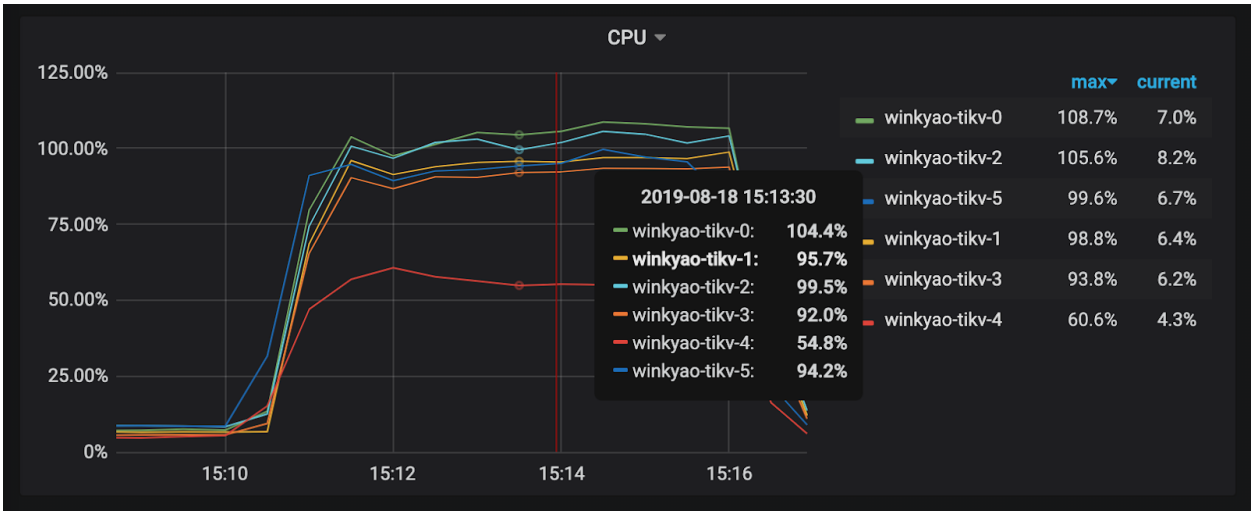


图 193: QPS6

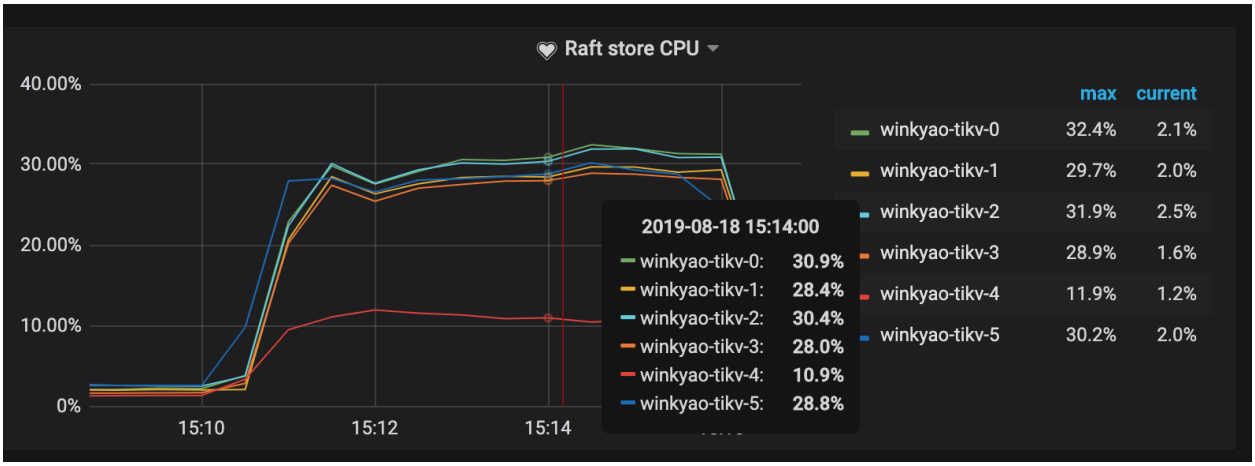


图 194: QPS7

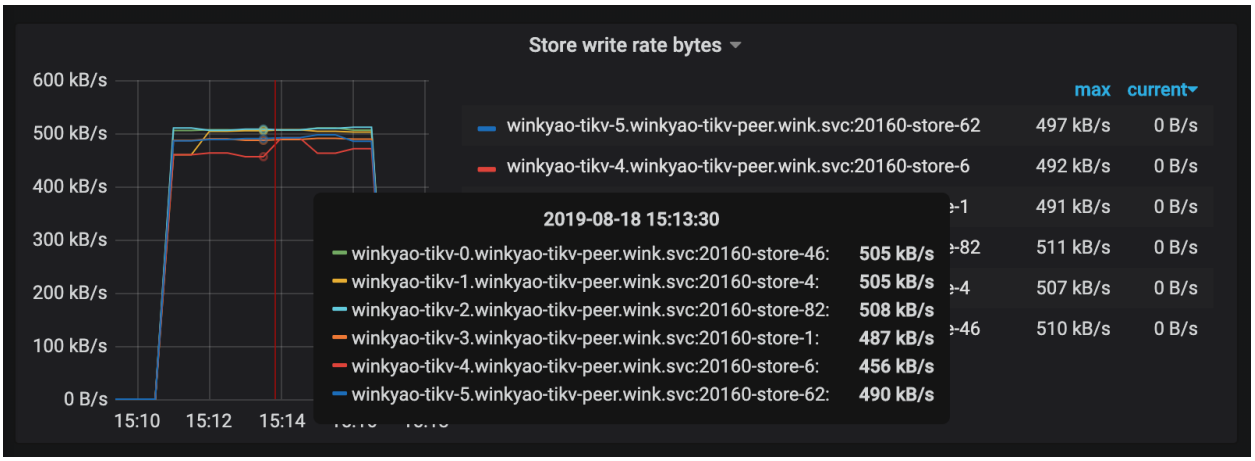


图 195: QPS8

可以看到已经消除了明显的热点问题了。

本示例仅为一个简单的表，还有索引热点的问题需要考虑。读者可参阅Split Region 文档来了解如何预先切散索引相关的 Region。

12.5.4.6.1 更复杂的热点问题

问题一：

如果表没有主键或者主键不是整数类型，而且用户也不想自己生成一个随机分布的主键 ID 的话，TiDB 内部有一个隐式的 `_tidb_rowid` 列作为行 ID。在不使用 `SHARD_ROW_ID_BITS` 的情况下，`_tidb_rowid` 列的值基本也为单调递增，此时也会有写热点存在（参阅`SHARD_ROW_ID_BITS` 的详细说明）。

要避免由 `_tidb_rowid` 带来的写入热点问题，可以在建表时，使用 `SHARD_ROW_ID_BITS` 和 `PRE_SPLIT_REGIONS` 这两个建表选项（参阅`PRE_SPLIT_REGIONS` 的详细说明）。

SHARD_ROW_ID_BITS 用于将 _tidb_rowid 列生成的行 ID 随机打散。PRE_SPLIT_REGIONS 用于在建完表后预先进行 Split region。

> **注意：**
>
> PRE_SPLIT_REGIONS 的值必须小于或等于 SHARD_ROW_ID_BITS。

以下全局变量会影响 PRE_SPLIT_REGIONS 的行为，需要特别注意：

- tidb_scatter_region：该变量用于控制建表完成后是否等待预切分和打散 Region 完成后再返回结果。如果建表后有大批量写入，需要设置该变量值为 1，表示等待所有 Region 都切分和打散完成后再返回结果给客户端。否则未打散完成就进行写入会对写入性能影响有较大的影响。

示例：

```
create table t (a int, b int) SHARD_ROW_ID_BITS = 4 PRE_SPLIT_REGIONS=3;
```

- SHARD_ROW_ID_BITS = 4 表示 tidb_rowid 的值会随机分布成 16 (16=2^4) 个范围区间。
- PRE_SPLIT_REGIONS=3 表示建完表后提前切分出 8 (2^3) 个 Region。

开始写数据进表 t 后，数据会被写入提前切分好的 8 个 Region 中，这样也避免了刚开始建表完后因为只有一个 Region 而存在的写热点问题。

问题二：

如果表的主键为整数类型，并且该表使用了 AUTO_INCREMENT 来保证主键唯一性（不需要连续或递增）的表而言，由于 TiDB 直接使用主键行值作为 _tidb_rowid，此时无法使用 SHARD_ROW_ID_BITS 来打散热点。

要解决上述热点问题，可以利用 AUTO_RANDOM 列属性（参阅AUTO_RANDOM 的详细说明），将 AUTO_INCREMENT 改为 AUTO_RANDOM，插入数据时让 TiDB 自动为整型主键列分配一个值，消除行 ID 的连续性，从而达到打散热点的目的。

12.5.4.7 参数配置

TiDB 2.1 版本中在 SQL 层引入了 latch 机制，用于在写入冲突比较频繁的场景中提前发现事务冲突，减少 TiDB 和 TiKV 事务提交时写写冲突导致的重试。通常，跑批场景使用的是存量数据，所以并不存在事务的写入冲突。可以把 TiDB 的 latch 功能关闭，以减少为细小对象分配内存：

```
[txn-local-latches]
enabled = false
```

12.5.5 使用 Grafana 监控 TiDB 的最佳实践

使用 TiUP 部署 TiDB 集群时，如果在拓扑配置中添加了 Grafana 和 Prometheus，会部署一套Grafana + Prometheus 的监控平台，用于收集和展示 TiDB 集群各个组件和机器的 metric 信息。本文主要介绍使用 TiDB 监控的最佳实践，旨在帮助 TiDB 用户高效利用丰富的 metric 信息来分析 TiDB 的集群状态或进行故障诊断。

12.5.5.1 监控架构

Prometheus 是一个拥有多维度数据模型和灵活查询语句的时序数据库。Grafana 是一个开源的 metric 分析及可视化系统。

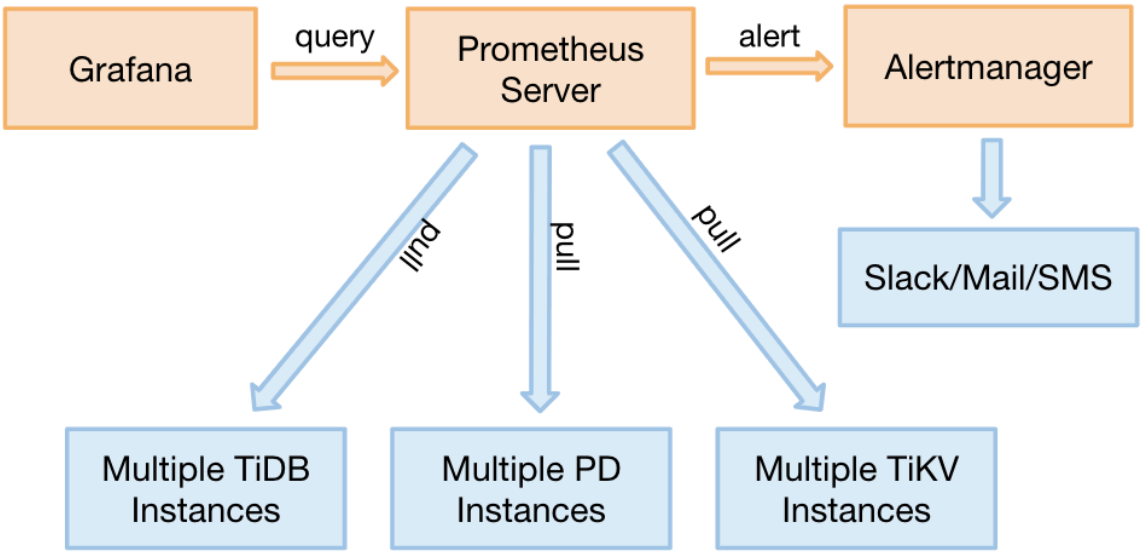


图 196: TiDB 监控整体架构

从 TiDB 2.1.3 版本开始，监控可以支持 pull，这是一个非常好的调整，它有以下几个优点：

- 如果 Prometheus 需要迁移，无需重启整个 TiDB 集群。调整前，因为组件要调整 push 的目标地址，迁移 Prometheus 需要重启整个集群。
- 支持部署 2 套独立的 Grafana + Prometheus 的监控平台（非 HA），防止监控的单点。
- 去掉了 Pushgateway 这个单点组件。

12.5.5.2 监控数据的来源与展示

TiDB 的 3 个核心组件（TiDB server、TiKV server 和 PD server）可以通过 HTTP 接口来获取 metric 数据。这些 metric 均是从程序代码中上传的，默认端口如下：

组件	端口
TiDB server	10080
TiKV server	20181
PD server	2379

下面以 TiDB server 为例，展示如何通过 HTTP 接口查看一个语句的 QPS 数据：

```
curl http://__tidb_ip__:10080/metrics |grep tidb_executor_statement_total
```

```
### 可以看到实时 QPS 数据，并根据不同 type 对 SQL 语句进行了区分，value 是 counter 类型的累计值（
    ↪ 科学计数法）。
tidb_executor_statement_total{type="Delete"} 520197
tidb_executor_statement_total{type="Explain"} 1
tidb_executor_statement_total{type="Insert"} 7.20799402e+08
tidb_executor_statement_total{type="Select"} 2.64983586e+08
tidb_executor_statement_total{type="Set"} 2.399075e+06
tidb_executor_statement_total{type="Show"} 500531
tidb_executor_statement_total{type="Use"} 466016
```

这些数据会存储在 Prometheus 中，然后在 Grafana 上进行展示。在面板上点击鼠标右键会出现 Edit 按钮（或直接按 E 键），如下图所示：

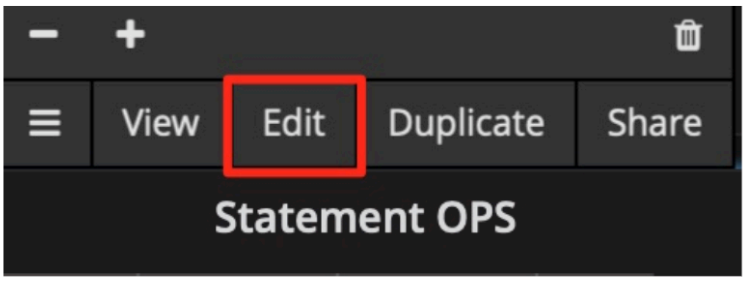


图 197: Metrics 面板的编辑入口

点击 Edit 按钮之后，在 Metrics 面板上可以看到利用该 metric 的 query 表达式。面板上一些细节的含义如下：

- `rate[1m]`：表示 1 分钟的增长速率，只能用于 counter 类型的数据。
- `sum`：表示 value 求和。
- `by type`：表示将求和后的数据按 metric 原始值中的 type 进行分组。
- `Legend format`：表示指标名称的格式。
- `Resolution`：默认打点步长是 15s，Resolution 表示是否将多个样本数据合并成一个点。

Metrics 面板中的表达式如下：



图 198: Metric 面板中的表达式

Prometheus 支持很多表达式与函数，更多表达式请参考 Prometheus 官网页面。

12.5.5.3 Grafana 使用技巧

本小节介绍高效利用 Grafana 监控分析 TiDB 指标的七个技巧。

12.5.5.3.1 技巧 1：查看所有维度并编辑表达式

在监控数据的来源与展示一节的示例中，数据是按照 type 进行分组的。如果你想知道是否还能按其它维度分组，并快速查看还有哪些维度，可采用以下技巧：在 query 的表达式上只保留指标名称，不做任何计算，Legend format 也留空。这样就能显示出原始的 metric 数据。比如，下图能看到有 3 个维度（instance、job 和 type）：

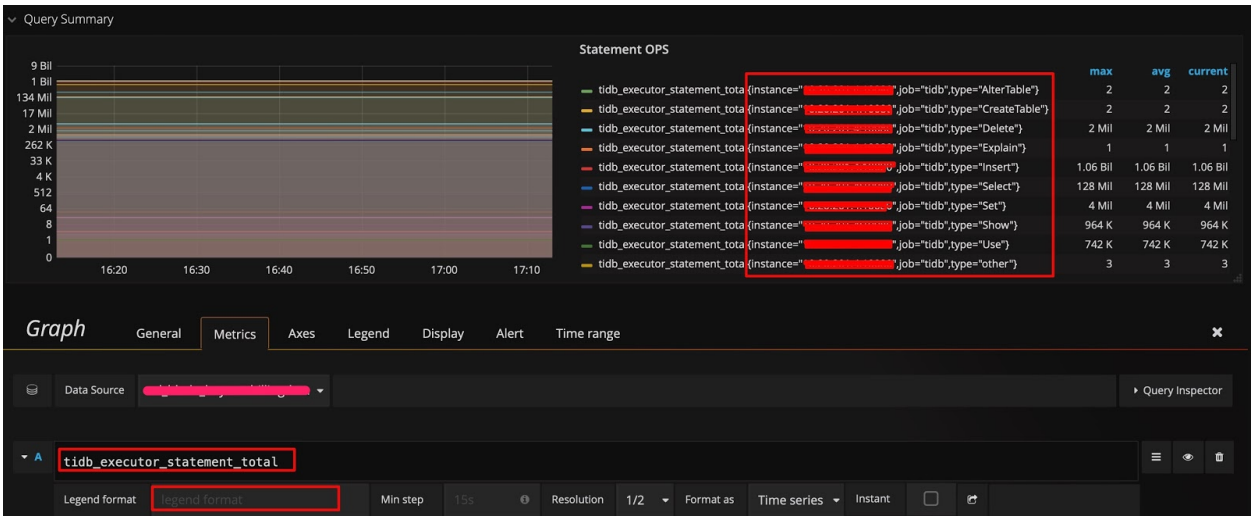


图 199: 编辑表达式并查看所有维度

然后调整表达式，在原有的 type 后面加上 instance 这个维度，在 Legend format 处增加 {{instance}}，就可以看到每个 TiDB server 上执行的不同类型 SQL 语句的 QPS 了。如下图所示：

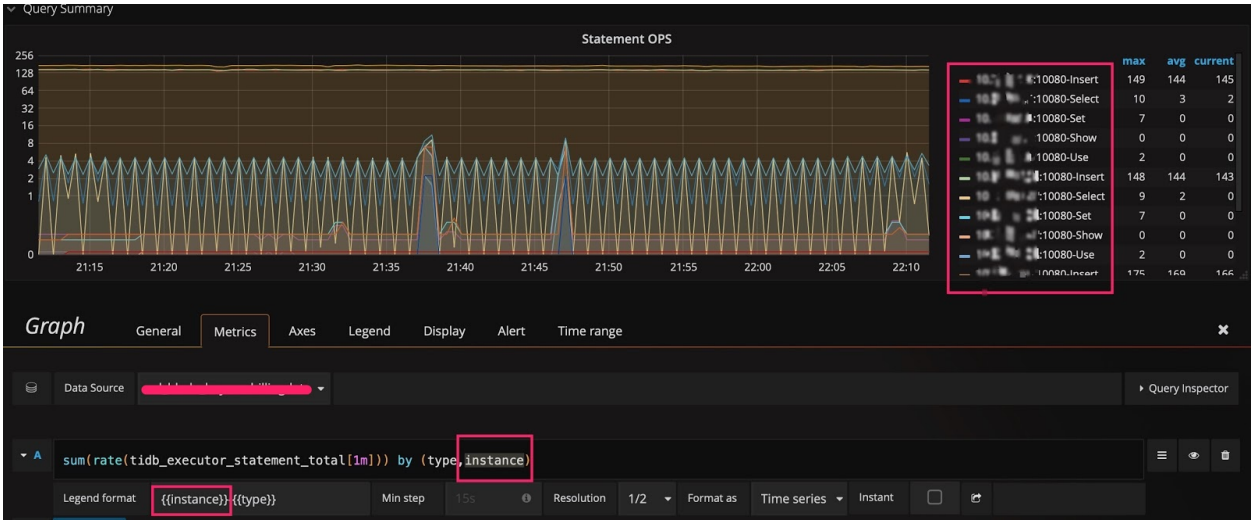


图 200: 给表达式增加一个 instance 维度

12.5.5.3.2 技巧 2：调整 Y 轴标尺的计算方式

以 Query Duration 指标为例，默认的比例尺采用 2 的对数计算，显示上会将差距缩小。为了观察到明显的变化，可以将比例尺改为线性，从下面两张图中可以看到显示上的区别，明显发现那个时刻有个 SQL 语句运行较慢。

当然也不是所有场景都适合用线性，比如观察 1 个月的性能趋势，用线性可能就会有很多噪点，不好观察。

标尺默认的比例尺为 2 的对数：



图 201: 标尺默认的比例尺为 2 的对数

将标尺的比例尺调整为线性：

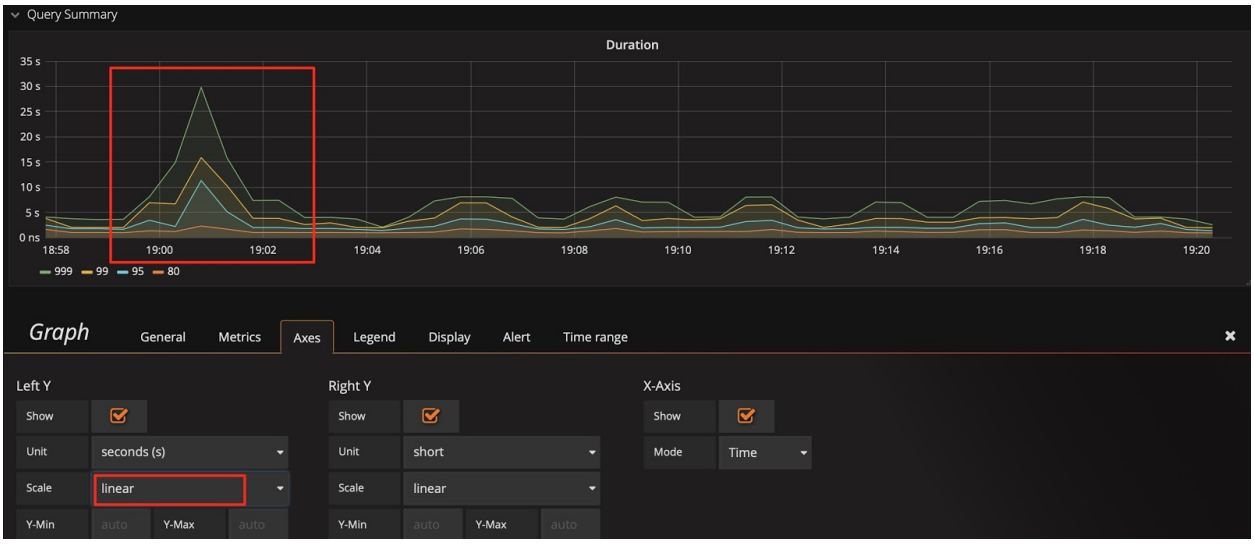


图 202: 调整标尺的比例尺为线性

> **建议：**
>
> 结合技巧 1，会发现这里还有一个 `sql_type` 的维度，可以立刻分析出是 `SELECT` 慢还是 `UPDATE` 慢；并且可以分析出是哪个 instance 上的语句慢。

12.5.5.3.3 技巧 3：调整 Y 轴基线，放大变化

有时已经用了线性比例尺，却还是看不出变化趋势。比如下图中，在扩容后想观察 `Store size` 的实时变化效果，但由于基数较大，观察不到微弱的变化。这时可以将 Y 轴最小值从 0 改为 `auto`，将上部放大。观察下面两张图的区别，可以看出数据已开始迁移了。

基线默认为 0：

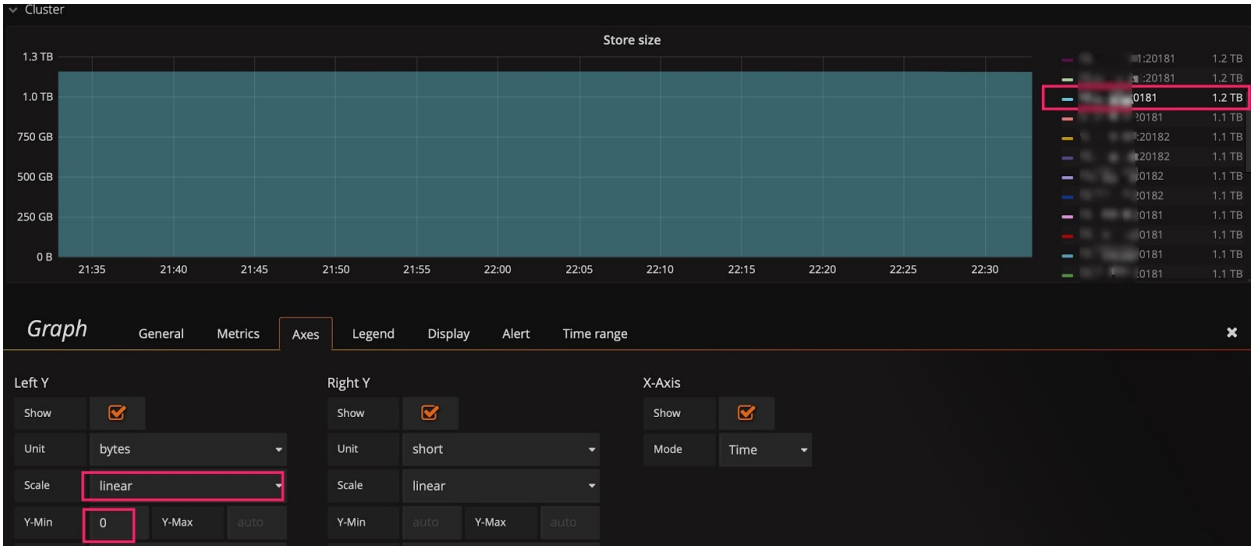


图 203: 基线默认为 0

将基线调整为 auto：



图 204: 调整基线为 auto

12.5.5.3.4 技巧 4：标尺联动

在 Settings 面板中，有一个 Graph Tooltip 设置项，默认使用 Default。

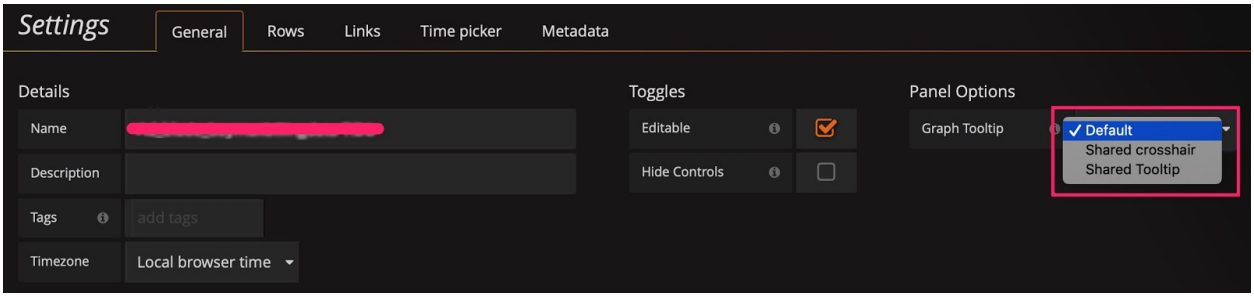


图 205: 图形展示工具

下面将图形展示工具分别调整为 Shared crosshair 和 Shared Tooltip 看看效果。可以看到标尺能联动展示了，方便排查问题时确认 2 个指标的关联性。

将图形展示工具调整为 Shared crosshair：

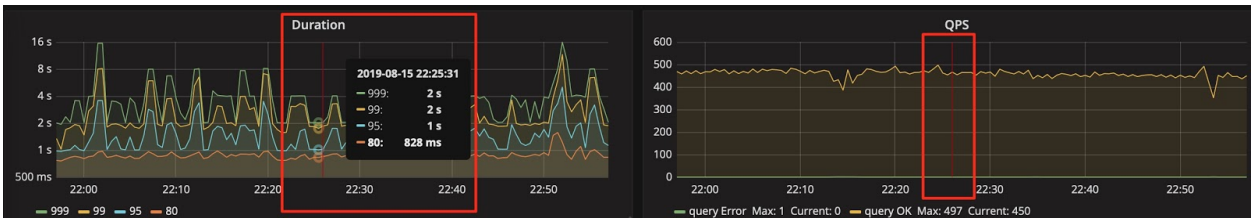


图 206: 调整图形展示工具为 Shared crosshair

将图形展示工具调整为 Shared Tooltip：

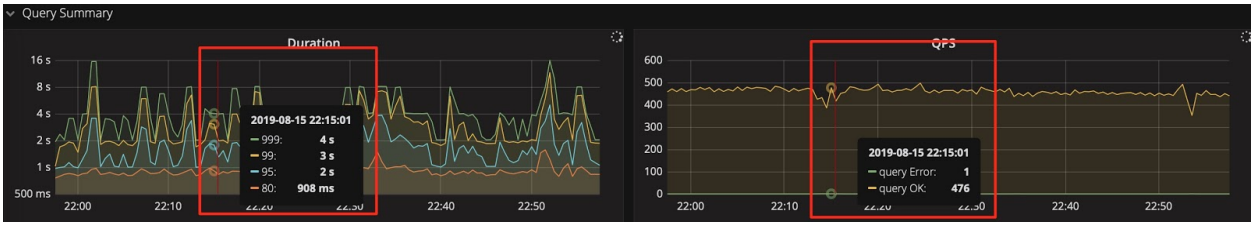


图 207: 调整图形展示工具为 Shared Tooltip

12.5.5.3.5 技巧 5：手动输入 ip: 端口号查看历史信息

PD 的 dashboard 只展示当前 leader 的 metric 信息，而有时想看历史上 PD leader 当时的状况，但是 instance 下拉列表中已不存在这个成员了。此时，可以手动输入 ip:2379 来查看当时的数据。

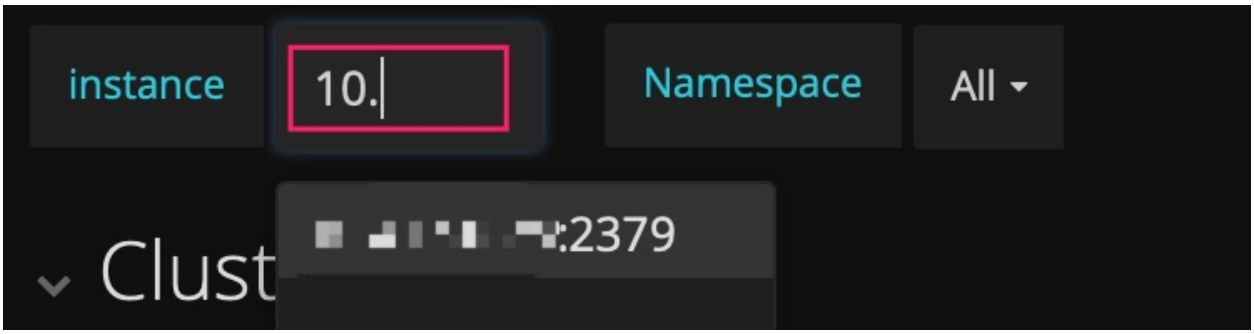


图 208: 查看历史 metric 信息

12.5.5.3.6 技巧 6：巧用 Avg 函数

通常默认图例中只有 Max 和 Current 函数。当指标波动较大时，可以增加 Avg 等其它汇总函数的图例，来看一段时间的整体趋势。

增加 Avg 等汇总函数：

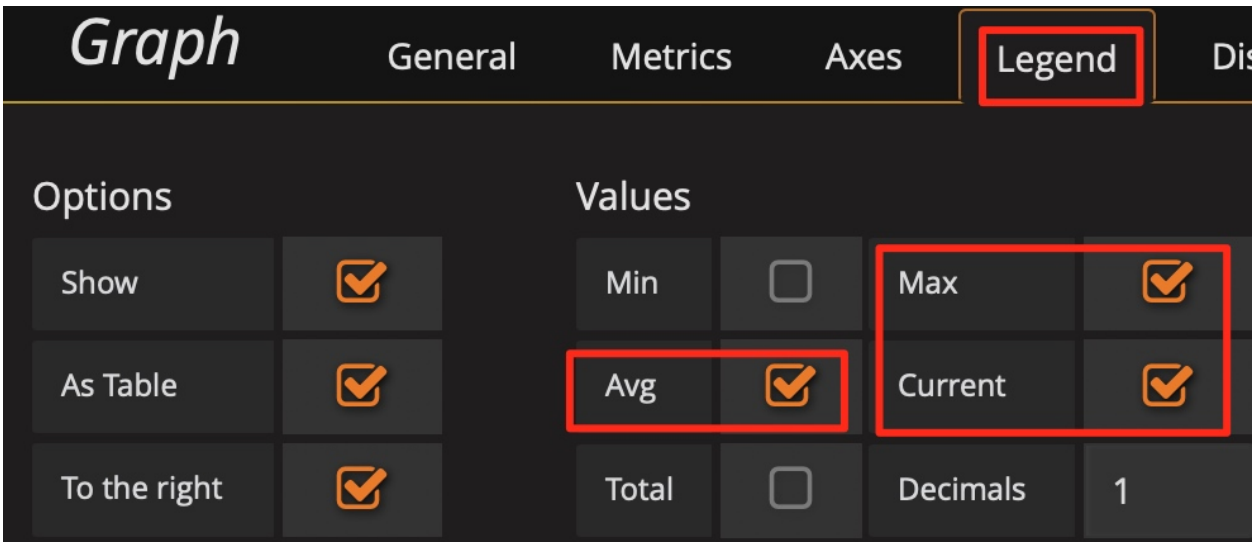


图 209: 增加 Avg 等汇总函数

然后查看整体趋势：

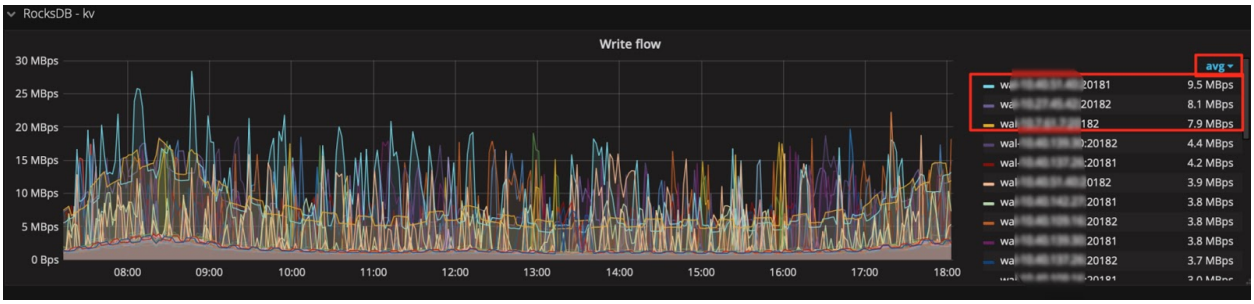


图 210: 增加 Avg 函数查看整体趋势

12.5.5.3.7 技巧 7：使用 Prometheus 的 API 接口获得表达式的结果

Grafana 通过 Prometheus 的接口获取数据，你也可以用该接口来获取数据，这个用法还可以衍生出许多功能：

- 自动获取集群规模、状态等信息。
- 对表达式稍加改动给报表提供数据，如统计每天的 QPS 总量、每天的 QPS 峰值和每天的响应时间。
- 将重要的指标进行定期健康巡检。

Prometheus 的 API 接口如下：

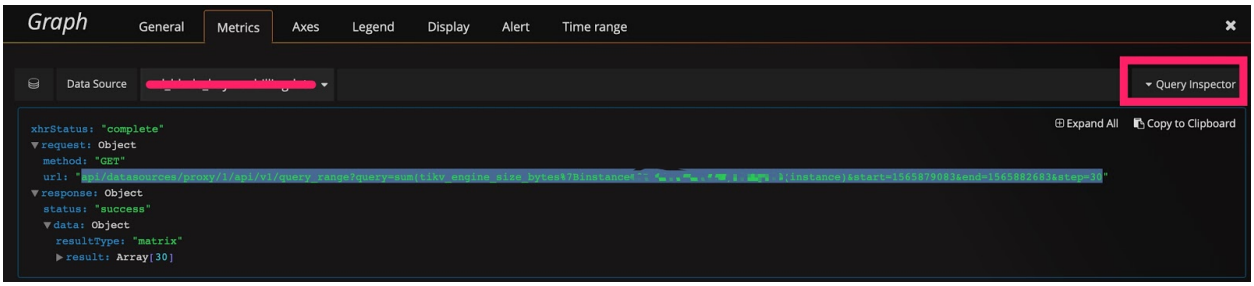


图 211: Prometheus 的 API 接口

```
curl -u user:pass 'http://__grafana_ip__:3000/api/datasources/proxy/1/api/v1/query_range?query=
    ↪ sum(tikv_engine_size_bytes%7Binstancexxxxxxxxx20181%22%7D)%20by%20(instance)&start
    ↪ =1565879269&end=1565882869&step=30' |python -m json.tool
```

```
{
    "data": {
        "result": [
            {
                "metric": {
                    "instance": "xxxxxxxxxx:20181"
                },
                "values": [
                    [
```

```
                    1565879269,
                    "1006046235280"
                ],
                [
                    1565879299,
                    "1006057877794"
                ],
                [
                    1565879329,
                    "1006021550039"
                ],
                [
                    1565879359,
                    "1006021550039"
                ],
                [
                    1565882869,
                    "1006132630123"
                ]
            ]
        }
    ],
    "resultType": "matrix"
},
"status": "success"
}
```


12.5.5.4 总结

Grafana + Prometheus 监控平台是一套非常强大的组合工具，用好这套工具可以为分析节省很多时间，提高效率，更重要的是，我们可以更容易发现问题。在运维 TiDB 集群，尤其是数据量大的情况下，这套工具能派上大用场。

12.5.6 PD 调度策略最佳实践

本文将详细介绍 PD 调度系统的原理，并通过几个典型场景的分析和处理方式，分享调度策略的最佳实践和调优方式，帮助大家在使用过程中快速定位问题。本文假定你对 TiDB，TiKV 以及 PD 已经有一定的了解，相关核心概念如下：

- Leader/Follower/Learner
- Operator
- Operator Step
- Pending/Down
- Region/Peer/Raft Group

- Region Split
- Scheduler
- Store

> 注意：
>
> 本文内容基于 TiDB 3.0 版本，更早的版本 (2.x) 缺少部分功能的支持，但是基本原理类似，也可以以本文作为参考。

12.5.6.1 PD 调度原理

该部分介绍调度系统涉及到的相关原理和流程。

12.5.6.1.1 调度流程

宏观上来看，调度流程大体可划分为 3 个部分：

1. 信息收集

   TiKV 节点周期性地向 PD 上报 `StoreHeartbeat` 和 `RegionHeartbeat` 两种心跳消息：

   - `StoreHeartbeat` 包含 Store 的基本信息、容量、剩余空间和读写流量等数据。
   - `RegionHeartbeat` 包含 Region 的范围、副本分布、副本状态、数据量和读写流量等数据。

   PD 梳理并转存这些信息供调度进行决策。

2. 生成调度

   不同的调度器从自身的逻辑和需求出发，考虑各种限制和约束后生成待执行的 Operator。这里所说的限制和约束包括但不限于：

   - 不往处于异常状态中（如断连、下线、繁忙、空间不足或在大量收发 snapshot 等）的 Store 添加副本
   - Balance 时不选择状态异常的 Region
   - 不尝试把 Leader 转移给 Pending Peer
   - 不尝试直接移除 Leader
   - 不破坏 Region 各种副本的物理隔离
   - 不破坏 Label property 等约束

3. 执行调度

   调度执行的步骤为：

   a. Operator 先进入一个由 `OperatorController` 管理的等待队列。

   b. `OperatorController` 会根据配置以一定的并发量从等待队列中取出 Operator 并执行。执行的过程就是依次把每个 Operator Step 下发给对应 Region 的 Leader。

   c. 标记 Operator 为 “finish” 或 “timeout” 状态，然后从执行列表中移除。

12.5.6.1.2 负载均衡

Region 负载均衡调度主要依赖 balance-leader 和 balance-region 两个调度器。二者的调度目标是将 Region 均匀地分散在集群中的所有 Store 上，但它们各有侧重：balance-leader 关注 Region 的 Leader，目的是分散处理客户端请求的压力；balance-region 关注 Region 的各个 Peer，目的是分散存储的压力，同时避免出现爆盘等状况。

balance-leader 与 balance-region 有着相似的调度流程：

1. 根据不同 Store 的对应资源量的情况分别打分。
2. 不断从得分较高的 Store 选择 Leader 或 Peer 迁移到得分较低的 Store 上。

两者的分数计算上也有一定差异：balance-leader 比较简单，使用 Store 上所有 Leader 所对应的 Region Size 加和作为得分。因为不同节点存储容量可能不一致，计算 balance-region 得分会分以下三种情况：

- 当空间富余时使用数据量计算得分（使不同节点数据量基本上均衡）
- 当空间不足时由使用剩余空间计算得分（使不同节点剩余空间基本均衡）
- 处于中间态时则同时考虑两个因素做加权和当作得分

此外，为了应对不同节点可能在性能等方面存在差异的问题，还可为 Store 设置负载均衡的权重。leader-weight 和 region-weight 分别用于控制 Leader 权重以及 Region 权重（默认值都为 “1”）。假如把某个 Store 的 leader-weight 设为 “2”，调度稳定后，则该节点的 Leader 数量约为普通节点的 2 倍；假如把某个 Store 的 region-weight 设为 “0.5”，那么调度稳定后该节点的 Region 数量约为其他节点的一半。

12.5.6.1.3 热点调度

热点调度对应的调度器是 hot-region-scheduler。从 TiDB v3.0 版本开始，统计热点 Region 的方式为：

1. 根据 Store 上报的信息，统计出持续一段时间读或写流量超过一定阈值的 Region。
2. 用与负载均衡类似的方式把这些 Region 分散开来。

对于写热点，热点调度会同时尝试打散热点 Region 的 Peer 和 Leader；对于读热点，由于只有 Leader 承载读压力，热点调度会尝试将热点 Region 的 Leader 打散。

12.5.6.1.4 集群拓扑感知

让 PD 感知不同节点分布的拓扑是为了通过调度使不同 Region 的各个副本尽可能分散，保证高可用和容灾。PD 会在后台不断扫描所有 Region，当发现 Region 的分布不是当前的最优化状态时，会生成调度以替换 Peer，将 Region 调整至最佳状态。

负责这个检查的组件叫 replicaChecker（跟 Scheduler 类似，但是不可关闭）。它依赖于 location-labels 配置项来进行调度。比如配置 [zone,rack,host] 定义了三层的拓扑结构：集群分为多个 zone（可用区），每个 zone 下有多个 rack（机架），每个 rack 下有多个 host（主机）。PD 在调度时首先会尝试将 Region 的 Peer 放置在不同的 zone，假如无法满足（比如配置 3 副本但总共只有 2 个 zone）则保证放置在不同的 rack；假如 rack 的数量也不足以保证隔离，那么再尝试 host 级别的隔离，以此类推。

12.5.6.1.5 缩容及故障恢复

缩容是指预备将某个 Store 下线，通过命令将该 Store 标记为 “Offline” 状态，此时 PD 通过调度将待下线节点上的 Region 迁移至其他节点。

故障恢复是指当有 Store 发生故障且无法恢复时，有 Peer 分布在对应 Store 上的 Region 会产生缺少副本的状况，此时 PD 需要在其他节点上为这些 Region 补副本。

这两种情况的处理过程基本上是一样的。`replicaChecker` 检查到 Region 存在异常状态的 Peer 后，生成调度在健康的 Store 上创建新副本替换异常的副本。

12.5.6.1.6 Region merge

Region merge 指的是为了避免删除数据后大量小甚至空的 Region 消耗系统资源，通过调度把相邻的小 Region 合并的过程。Region merge 由 `mergeChecker` 负责，其过程与 `replicaChecker` 类似：PD 在后台遍历，发现连续的小 Region 后发起调度。

具体来说，当某个新分裂出来的 Region 存在的时间超过配置项`split-merge-interval` 的值（默认 1h）后，如果同时满足以下情况，该 Region 会触发 Region merge 调度：

- 该 Region 大小小于配置项`max-merge-region-size` 的值（默认 20 MiB）
- 该 Region 中 key 的数量小于配置项`max-merge-region-keys` 的值（默认 200000）

12.5.6.2 查询调度状态

你可以通过观察 PD 相关的 Metrics 或使用 pd-ctl 工具等方式查看调度系统状态。更具体的信息可以参考PD 监控和PD Control。

12.5.6.2.1 Operator 状态

Grafana PD/Operator 页面展示了 Operator 的相关统计信息。其中比较重要的有：

- Schedule Operator Create：Operator 的创建情况
- Operator finish duration：Operator 执行耗时的情况
- Operator Step duration：不同 Operator Step 执行耗时的情况

查询 Operator 的 pd-ctl 命令有：

- `operator show`：查询当前调度生成的所有 Operator
- `operator show [admin | leader | region]`：按照类型查询 Operator

12.5.6.2.2 Balance 状态

Grafana PD/Statistics - Balance 页面展示了负载均衡的相关统计信息，其中比较重要的有：

- Store Leader/Region score：每个 Store 的得分
- Store Leader/Region count：每个 Store 的 Leader/Region 数量
- Store available：每个 Store 的剩余空间

使用 pd-ctl 的 `store` 命令可以查询 Store 的得分、数量、剩余空间和 weight 等信息。

12.5.6.2.3 热点调度状态

Grafana PD/Statistics - hotspot 页面展示了热点 Region 的相关统计信息，其中比较重要的有：

- Hot write Region’s leader/peer distribution：写热点 Region 的 Leader/Peer 分布情况
- Hot read Region’s leader distribution：读热点 Region 的 Leader 分布情况

使用 pd-ctl 同样可以查询上述信息，可以使用的命令有：

- `hot read`：查询读热点 Region 信息
- `hot write`：查询写热点 Region 信息
- `hot store`：按 Store 统计热点分布情况
- `region topread [limit]`：查询当前读流量最大的 Region
- `region topwrite [limit]`：查询当前写流量最大的 Region

12.5.6.2.4 Region 健康度

Grafana PD/Cluster/Region health 面板展示了异常 Region 的相关统计信息，包括 Pending Peer、Down Peer、Offline Peer，以及副本数过多或过少的 Region。

通过 pd-ctl 的 `region check` 命令可以查看具体异常的 Region 列表：

- `region check miss-peer`：缺副本的 Region
- `region check extra-peer`：多副本的 Region
- `region check down-peer`：有副本状态为 Down 的 Region
- `region check pending-peer`：有副本状态为 Pending 的 Region

12.5.6.3 调度策略控制

使用 pd-ctl 可以从以下三个方面来调整 PD 的调度策略。更具体的信息可以参考PD Control。

12.5.6.3.1 启停调度器

pd-ctl 支持动态创建和删除 Scheduler，你可以通过这些操作来控制 PD 的调度行为，如：

- `scheduler show`：显示当前系统中的 Scheduler
- `scheduler remove balance-leader-scheduler`：删除（停用）balance region 调度器
- `scheduler add evict-leader-scheduler 1`：添加移除 Store 1 的所有 Leader 的调度器

12.5.6.3.2 手动添加 Operator

PD 支持直接通过 pd-ctl 来创建或删除 Operator，如：

- `operator add add-peer 2 5`：在 Store 5 上为 Region 2 添加 Peer
- `operator add transfer-leader 2 5`：将 Region 2 的 Leader 迁移至 Store 5
- `operator add split-region 2`：将 Region 2 拆分为 2 个大小相当的 Region
- `operator remove 2`：取消 Region 2 当前待执行的 Operator

12.5.6.3.3 调度参数调整

使用 pd-ctl 执行 `config show` 命令可以查看所有的调度参数，执行 `config set {key} {value}` 可以调整对应参数的值。常见调整如下：

- `leader-schedule-limit`：控制 Transfer Leader 调度的并发数
- `region-schedule-limit`：控制增删 Peer 调度的并发数
- `enable-replace-offline-replica`：开启节点下线的调度
- `enable-location-replacement`：开启调整 Region 隔离级别相关的调度
- `max-snapshot-count`：每个 Store 允许的最大收发 Snapshot 的并发数

12.5.6.4 典型场景分析与处理

该部分通过几个典型场景及其应对方式说明 PD 调度策略的最佳实践。

12.5.6.4.1 Leader/Region 分布不均衡

PD 的打分机制决定了一般情况下，不同 Store 的 Leader Count 和 Region Count 不能完全说明负载均衡状态，所以需要从 TiKV 的实际负载或者存储空间占用来判断是否有负载不均衡的状况。

确认 Leader/Region 分布不均衡后，首先观察不同 Store 的打分情况。

如果不同 Store 的打分是接近的，说明 PD 认为此时已经是均衡状态了，可能的原因有：

- 存在热点导致负载不均衡。可以参考热点分布不均匀中的解决办法进行分析处理。
- 存在大量空 Region 或小 Region，因此不同 Store 的 Leader 数量差别特别大，导致 Raftstore 负担过重。此时需要开启Region Merge 并尽可能加速合并。
- 不同 Store 的软硬件环境存在差异。可以参考负载均衡一节视实际情况调整 `leader-weight` 和 `region-weight` 来控制 Leader/Region 的分布。
- 其他不明原因。仍可以通过调整 `leader-weight` 和 `region-weight` 来控制 Leader/Region 的分布。

如果不同 Store 的分数差异较大，需要进一步检查 Operator 的相关 Metrics，特别关注 Operator 的生成和执行情况，这时大体上又分两种情况：

- 生成的调度是正常的，但是调度的速度很慢。可能的原因有：
  - 调度速度受限于 limit 配置。PD 默认配置的 limit 比较保守，在不对正常业务造成显著影响的前提下，可以酌情将 `leader-schedule-limit` 或 `region-schedule-limit` 调大一些。此外，`max-pending-peer-count` 以及 `max-snapshot-count` 限制也可以放宽。
  - 系统中同时运行有其他的调度任务产生竞争，导致 balance 速度上不去。这种情况下如果 balance 调度的优先级更高，可以先停掉其他的调度或者限制其他调度的速度。例如 Region 没均衡的情况下做下线节点操作，下线的调度与 Region Balance 会抢占 `region-schedule-limit` 配额，此时你可以调小 `replica-schedule-limit` 以限制下线调度的速度，或者设置 `enable-replace-offline-replica = false` 来暂时关闭下线流程。
  - 调度执行得太慢。可以通过 Operator step duration 进行判断。通常不涉及到收发 Snapshot 的 Step（比如 `TransferLeader`，`RemovePeer`，`PromoteLearner` 等）的完成时间应该在毫秒级，涉及到 Snapshot 的 Step（如 `AddLearner`，`AddPeer` 等）的完成时间为数十秒。如果耗时明显过高，可能是 TiKV 压力过大或者网络等方面的瓶颈导致的，需要具体情况具体分析。

- 没能生成对应的 balance 调度。可能的原因有：
  - 调度器未被启用。比如对应的 Scheduler 被删除了，或者 limit 被设置为 “0”。
  - 由于其他约束无法进行调度。比如系统中有 `evict-leader-scheduler`，此时无法把 Leader 迁移至对应的 Store。再比如设置了 Label property，也会导致部分 Store 不接受 Leader。
  - 集群拓扑的限制导致无法均衡。比如 3 副本 3 数据中心的集群，由于副本隔离的限制，每个 Region 的 3 个副本都分别分布在不同的数据中心，假如这 3 个数据中心的 Store 数不一样，最后调度就会收敛在每个数据中心均衡，但是全局不均衡的状态。

12.5.6.4.2 节点下线速度慢

这个场景需要从 Operator 相关的 Metrics 入手，分析 Operator 的生成执行情况。

如果调度在正常生成，只是速度很慢，可能的原因有：

- 调度速度受限于 limit 配置。可以适当调大 `replica-schedule-limit`，`max-pending-peer-count` 以及 `max-snapshot-count` 限制也可以放宽。
- 系统中同时运行有其他的调度任务产生竞争。处理方法参考Leader/Region 分布不均衡。
- 下线单个节点时，由于待操作的 Region 有很大一部分（3 副本配置下约 1/3）的 Leader 都集中在下线的节点上，下线速度会受限于这个单点生成 Snapshot 的速度。你可以通过手动给该节点添加一个 `evict-leader-scheduler` 调度器迁走 Leader 来加速。

如果没有对应的 Operator 调度生成，可能的原因有：

- 下线调度被关闭，或者 `replica-schedule-limit` 被设为 “0”。
- 找不到节点来转移 Region。例如相同 Label 的替代节点可用容量都不足 20%，PD 为了避免爆盘的风险会停止调度。这种情况需要添加更多节点，或者删除一些数据释放空间。

12.5.6.4.3 节点上线速度慢

目前 PD 没有对节点上线特殊处理。节点上线实际上是依靠 balance region 机制来调度的，所以参考Leader/Region 分布不均衡中的排查步骤即可。

12.5.6.4.4 热点分布不均匀

热点调度的问题大体上可以分为以下几种情况：

- 从 PD 的 Metrics 能看出来有不少 hot Region，但是调度速度跟不上，不能及时地把热点 Region 分散开来。

解决方法：调大 `hot-region-schedule-limit` 并减少其他调度器的 limit 配额，从而加快热点调度的速度。还可调小 `hot-region-cache-hits-threshold` 使 PD 对更快响应流量的变化。

- 单一 Region 形成热点，比如大量请求频繁 scan 一个小表，这个可以从业务角度或者 Metrics 统计的热点信息看出来。由于单 Region 热点现阶段无法使用打散的手段来消除，需要确认热点 Region 后手动添加 `split-region` 调度将这样的 Region 拆开。

- 从 PD 的统计来看没有热点，但是从 TiKV 的相关 Metrics 可以看出部分节点负载明显高于其他节点，成为整个系统的瓶颈。这是因为目前 PD 统计热点 Region 的维度比较单一，仅针对流量进行分析，在某些场景下无法准确定位热点。例如部分 Region 有大量的点查请求，从流量上来看并不显著，但是过高的 QPS 导致关键模块达到瓶颈。

  解决方法：首先从业务层面确定形成热点的 table，然后添加 `scatter-range-scheduler` 调度器使这个 table 的所有 Region 均匀分布。TiDB 也在其 HTTP API 中提供了相关接口来简化这个操作，具体可以参考 TiDB HTTP API 文档。

12.5.6.4.5 Region Merge 速度慢

Region Merge 速度慢也很有可能是受到 limit 配置的限制（`merge-schedule-limit` 及 `region-schedule-limit`），或者是与其他调度器产生了竞争。具体来说，可有如下处理方式：

- 假如已经从相关 Metrics 得知系统中有大量的空 Region，这时可以通过把 `max-merge-region-size` 和 `max-merge-region-keys` 调整为较小值来加快 Merge 速度。这是因为 Merge 的过程涉及到副本迁移，所以 Merge 的 Region 越小，速度就越快。如果生成 Merge Operator 的速度很快，想进一步加快 Region Merge 过程，还可以把 `patrol-region-interval` 调整为“10ms”(从 v5.3.0 起，此配置项默认值为“10ms”)，这个能加快巡检 Region 的速度，但是会消耗更多的 CPU 资源。
- 创建过大量表后（包括执行 `Truncate Table` 操作）又清空了。此时如果开启了 split table 特性，这些空 Region 是无法合并的，此时需要调整以下参数关闭这个特性：
  - TiKV: 将 `split-region-on-table` 设为 `false`，该参数不支持动态修改。
  - PD: 使用 PD Control，根据集群情况选择性地设置以下参数。
    * 如果集群中不存在 TiDB 实例，将`key-type` 的值设置为 `raw` 或 `txn`。此时，无论 `enable-cross-table-merge` 设置如何，PD 均可以跨表合并 Region。该参数支持动态修改。

```
config set key-type txn
```

    * 如果集群中存在 TiDB 实例，将 `key-type` 的值设置为 `table`。此时将 `enable-cross-table-merge` 设置为 `true`，可以使 PD 跨表合并 Region。该参数支持动态修改。

```
config set enable-cross-table-merge true
```

  如果修改未生效，请参阅FAQ - 修改 TiKV/PD 的 toml 配置文件后没有生效。

> 注意：
> 在开启 `placement-rules` 后，请合理切换 `key-type`，避免无法正常解码 key。

- 对于 3.0.4 和 2.1.16 以前的版本，Region 中 Key 的个数 (`approximate_keys`) 在特定情况下（大部分发生在删表之后）统计不准确，造成 keys 的统计值很大，无法满足 `max-merge-region-keys` 的约束。你可以通过调大 `max-merge-region-keys` 来避免这个问题。

12.5.6.4.6 TiKV 节点故障处理策略

没有人工介入时，PD 处理 TiKV 节点故障的默认行为是，等待半小时之后（可通过 `max-store-down-time` 配置调整），将此节点设置为 Down 状态，并开始为涉及到的 Region 补充副本。

实践中，如果能确定这个节点的故障是不可恢复的，可以立即做下线处理，这样 PD 能尽快补齐副本，降低数据丢失的风险。与之相对，如果确定这个节点是能恢复的，但可能半小时之内来不及，则可以把 `max-store-down-time` 临时调整为比较大的值，这样能避免超时之后产生不必要的副本补充，造成资源浪费。

自 v5.2.0 起，TiKV 引入了慢节点检测机制。通过对 TiKV 中的请求进行采样，计算出一个范围在 1~100 的分数。当分数大于等于 80 时，该 TiKV 节点会被设置为 slow 状态。可以通过添加`evict-slow-store-scheduler` 来针对慢节点进行对应的检测和调度。当检测到有且只有一个 TiKV 节点为慢节点，并且该 TiKV 的 slow score 到达限定值（默认 80）时，将节点上的 leader 驱逐（其作用类似于 `evict-leader-scheduler`）。

12.5.7 海量 Region 集群调优最佳实践

在 TiDB 的架构中，所有数据以一定 key range 被切分成若干 Region 分布在多个 TiKV 实例上。随着数据的写入，一个集群中会产生上百万个甚至千万个 Region。单个 TiKV 实例上产生过多的 Region 会给集群带来较大的负担，影响整个集群的性能表现。

本文将介绍 TiKV 核心模块 Raftstore 的工作流程，海量 Region 导致性能问题的原因，以及优化性能的方法。

12.5.7.1 Raftstore 的工作流程

一个 TiKV 实例上有多个 Region。Region 消息是通过 Raftstore 模块驱动 Raft 状态机来处理的。这些消息包括 Region 上读写请求的处理、Raft log 的持久化和复制、Raft 的心跳处理等。但是，Region 数量增多会影响整个集群的性能。为了解释这一点，需要先了解 TiKV 的核心模块 Raftstore 的工作流程。

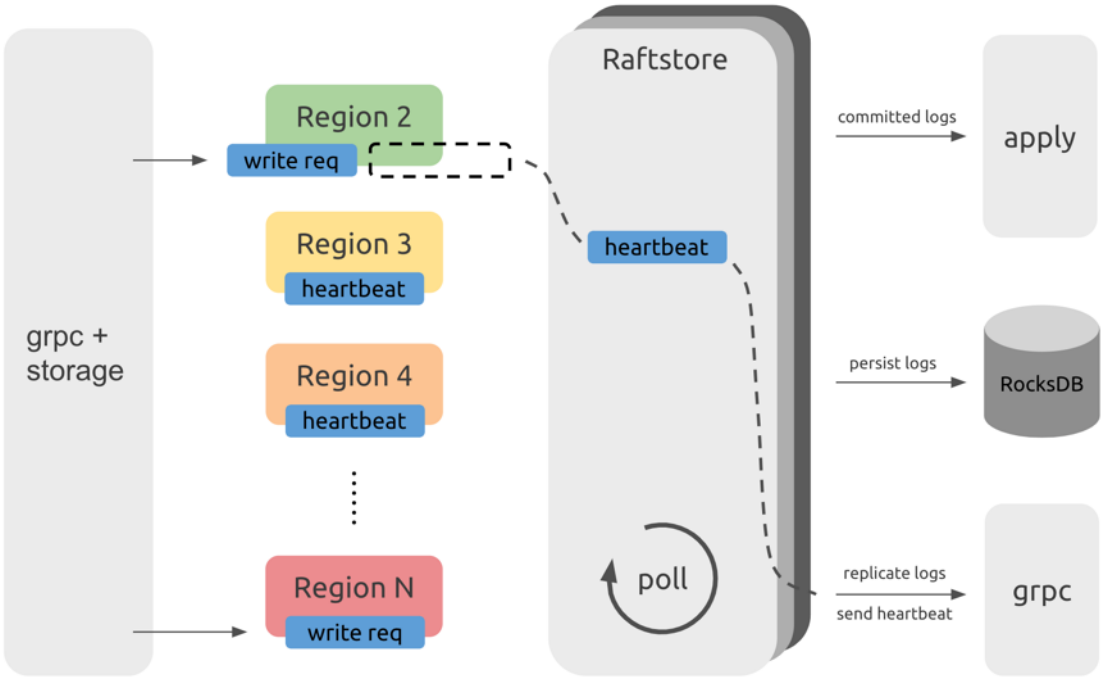


图 212: 图 1 Raftstore 处理流程示意图

> 注意：
>
> 该图仅为示意，不代表代码层面的实际结构。

上图是 Raftstore 处理流程的示意图。如图所示，从 TiDB 发来的请求会通过 gRPC 和 storage 模块变成最终的 KV 读写消息，并被发往相应的 Region，而这些消息并不会被立即处理而是被暂存下来。Raftstore 会轮询检查每个 Region 是否有需要处理的消息。如果 Region 有需要处理的消息，那么 Raftstore 会驱动 Raft 状态机去处理这些消息，并根据这些消息所产生的状态变更去进行后续操作。例如，在有写请求时，Raft 状态机需要将日志落盘并且将日志发送给其他 Region 副本；在达到心跳间隔时，Raft 状态机需要将心跳信息发送给其他 Region 副本。

12.5.7.2 性能问题

从 Raftstore 处理流程示意图可以看出，需要依次处理各个 Region 的消息。那么在 Region 数量较多的情况下，Raftstore 需要花费一些时间去处理大量 Region 的心跳，从而带来一些延迟，导致某些读写请求得不到及时处理。如果读写压力较大，Raftstore 线程的 CPU 使用率容易达到瓶颈，导致延迟进一步增加，进而影响性能表现。

通常在有负载的情况下，如果 Raftstore 的 CPU 使用率达到了 85% 以上，即可视为达到繁忙状态且成为了瓶颈，同时 `propose wait duration` 可能会高达百毫秒级别。

> 注意：
>
> - Raftstore 的 CPU 使用率是指单线程的情况。如果是多线程 Raftstore，可等比例放大使用率。
> - 由于 Raftstore 线程中有 I/O 操作，所以 CPU 使用率不可能达到 100%。

12.5.7.2.1 性能监控

可在 Grafana 的 TiKV 面板下查看相关的监控 metrics：

- Thread-CPU 下的 `Raft store CPU`

  参考值：低于 `raftstore.store-pool-size * 85%`。

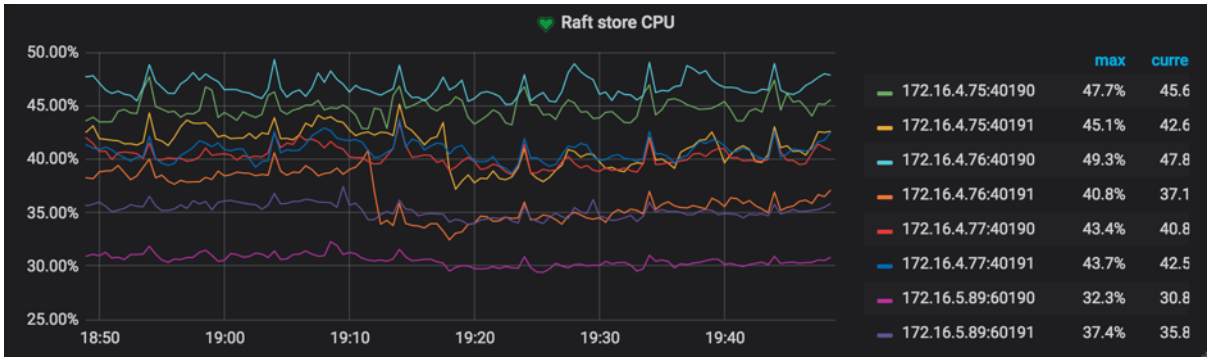


图 213: 图 2 查看 Raftstore CPU

- Raft Propose 下的 `Propose wait duration`

  `Propose wait duration` 是从发送请求给 Raftstore，到 Raftstore 真正开始处理请求之间的延迟时间。如果该延迟时间较长，说明 Raftstore 比较繁忙或者处理 append log 比较耗时导致 Raftstore 不能及时处理请求。

  参考值：低于 50-100ms。

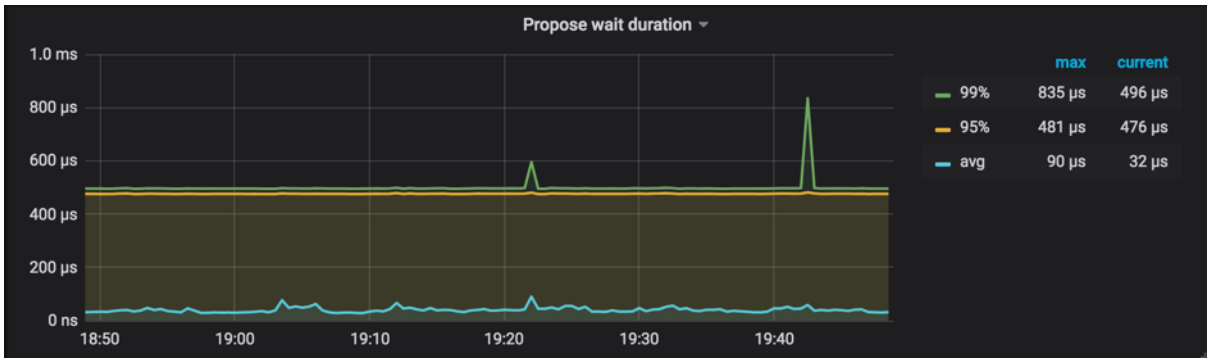


图 214: 图 3 查看 Propose wait duration

12.5.7.3 性能优化方法

找到性能问题的根源后，可从以下两个方向来解决性能问题：

- 减少单个 TiKV 实例的 Region 数
- 减少单个 Region 的消息数

12.5.7.3.1 方法一：增加 TiKV 实例

如果 I/O 资源和 CPU 资源都比较充足，可在单台机器上部署多个 TiKV 实例，以减少单个 TiKV 实例上的 Region 个数；或者增加 TiKV 集群的机器数。

12.5.7.3.2 方法二：调整 `raft-base-tick-interval`

除了减少 Region 个数外，还可以通过减少 Region 单位时间内的消息数量来减小 Raftstore 的压力。例如，在 TiKV 配置中适当调大 `raft-base-tick-interval`：

```
[raftstore]
raft-base-tick-interval = "2s"
```

`raft-base-tick-interval` 是 Raftstore 驱动每个 Region 的 Raft 状态机的时间间隔，也就是每隔该时长就需要向 Raft 状态机发送一个 tick 消息。增加该时间间隔，可以有效减少 Raftstore 的消息数量。

需要注意的是，该 tick 消息的间隔也决定了 `election timeout` 和 `heartbeat` 的间隔。示例如下：

```
raft-election-timeout = raft-base-tick-interval * raft-election-timeout-ticks
raft-heartbeat-interval = raft-base-tick-interval * raft-heartbeat-ticks
```

如果 Region Follower 在 `raft-election-timeout` 间隔内未收到来自 Leader 的心跳，就会判断 Leader 出现故障而发起新的选举。`raft-heartbeat-interval` 是 Leader 向 Follower 发送心跳的间隔，因此调大 `raft-base-tick-interval` 可以减少单位时间内 Raft 发送的网络消息，但也会让 Raft 检测到 Leader 故障的时间更长。

12.5.7.3.3 方法三：提高 Raftstore 并发数

从 v3.0 版本起，Raftstore 已经扩展为多线程，极大降低了 Raftstore 线程成为瓶颈的可能性。

TiKV 默认将 `raftstore.store-pool-size` 配置为 2。如果 Raftstore 出现瓶颈，可以根据实际情况适当调高该参数值，但不建议设置过高以免引入不必要的线程切换开销。

12.5.7.3.4 方法四：开启 Hibernate Region 功能

在实际情况中，读写请求并不会均匀分布到每个 Region 上，而是集中在少数的 Region 上。那么可以尽量减少暂时空闲的 Region 的消息数量，这也就是 Hibernate Region 的功能。无必要时可不进行 `raft-base-tick`，即不驱动空闲 Region 的 Raft 状态机，那么就不会触发这些 Region 的 Raft 产生心跳信息，极大地减小了 Raftstore 的工作负担。

Hibernate Region 在 TiKV master 分支上默认开启。可根据实际情况和需求来配置此功能的开启和关闭，请参阅配置 Hibernate Region。

12.5.7.3.5 方法五：开启 Region Merge

> **注意：**
>
> 从 TiDB v3.0 开始，`Region Merge` 默认开启。

开启 `Region Merge` 也能减少 Region 的个数。与 `Region Split` 相反，`Region Merge` 是通过调度把相邻的小 Region 合并的过程。在集群中删除数据或者执行 `Drop Table/Truncate Table` 语句后，可以将小 Region 甚至空 Region 进行合并以减少资源的消耗。

通过 pd-ctl 设置以下参数即可开启 `Region Merge`：

```
config set max-merge-region-size 20
config set max-merge-region-keys 200000
config set merge-schedule-limit 8
```

详情请参考如何配置 Region Merge（英文）和PD 配置文件描述。

同时，默认配置的 `Region Merge` 的参数设置较为保守，可以根据需求参考PD 调度策略最佳实践中提供的方法加快 `Region Merge` 过程的速度。

12.5.7.3.6 方法六：调整 Region 大小

Region 默认的大小约为 96 MiB，将其调大也可以减少 Region 个数，具体介绍可参考使用更大的 Region。

> **警告：**
>
> 自定义 Region 大小是在 TiDB v6.1.0 引入的实验特性，不建议在生产环境中配置。使用此特性的风险包括：

- 更容易发生性能抖动。
- 查询性能回退，尤其是大范围数据查询的性能会有回退。
- 调度变慢。

12.5.7.4 其他问题和解决方案

12.5.7.4.1 切换 PD Leader 的速度慢

PD 需要将 Region Meta 信息持久化在 etcd 上，以保证切换 PD Leader 节点后 PD 能快速继续提供 Region 路由服务。随着 Region 数量的增加，etcd 出现性能问题，使得 PD 在切换 Leader 时从 etcd 获取 Region Meta 信息的速度较慢。在百万 Region 量级时，从 etcd 获取信息的时间可能需要十几秒甚至几十秒。

因此从 v3.0 版本起，PD 默认开启配置项 `use-region-storage`，将 Region Meta 信息存在本地的 LevelDB 中，并通过其他机制同步 PD 节点间的信息。

12.5.7.4.2 PD 路由信息更新不及时

在 TiKV 中，pd-worker 模块将 Region Meta 信息定期上报给 PD，在 TiKV 重启或者切换 Region Leader 时需要通过统计信息重新计算 Region 的 `approximate size/keys`。因此在 Region 数量较多的情况下，pd-worker 单线程可能成为瓶颈，造成任务得不到及时处理而堆积起来。因此 PD 不能及时获取某些 Region Meta 信息以致路由信息更新不及时。该问题不会影响实际的读写，但可能导致 PD 调度不准确以及 TiDB 更新 Region cache 时需要多几次 round-trip。

可在 TiKV Grafana 面板中查看 Task 下的 Worker pending tasks 来确定 pd-worker 是否有任务堆积。通常来说，pending tasks 应该维持在一个比较低的值。

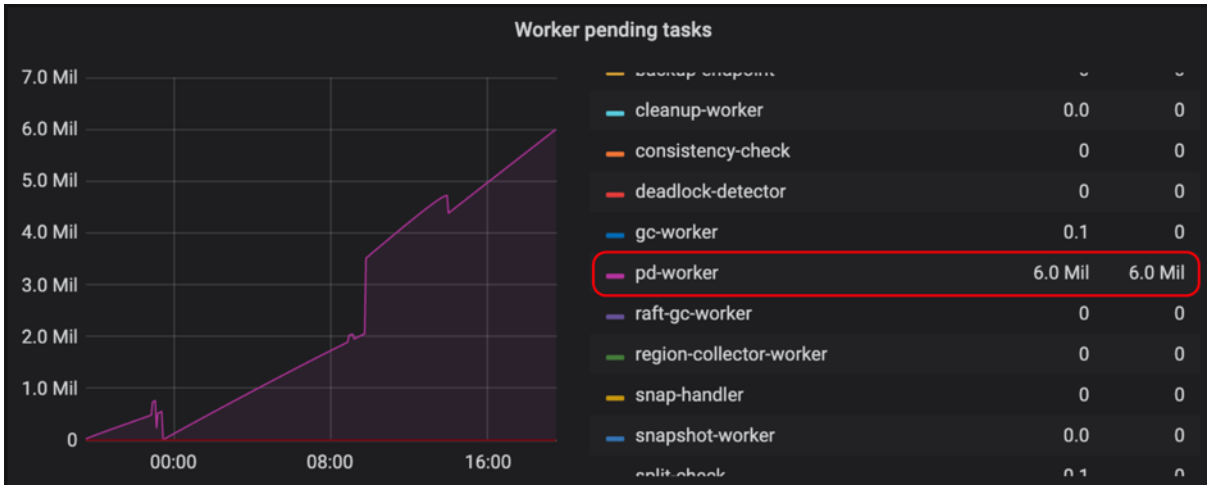


图 215: 图 4 查看 pd-worker

老版本 TiDB (< v3.0.5) pd-worker 的效率有一些缺陷，如果碰到类似问题，建议升级至最新版本。

12.5.7.4.3 Prometheus 查询 metrics 的速度慢

在大规模集群中，随着 TiKV 实例数的增加，Prometheus 查询 metrics 时的计算压力较大，导致 Grafana 查看 metrics 的速度较慢。从 v3.0 版本起设置了一些 metrics 的预计算，让这个问题有所缓解。

12.5.8 三节点混合部署的最佳实践

在对性能要求不高且需要控制成本的场景下，将 TiDB、TiKV、PD 混合部署在三台机器上是一个可行的方案。

本文以 TPC-C 作为工作负载，提供一些在三节点混合部署场景下部署和参数调整的建议。

12.5.8.1 部署环境和测试方法

三台物理机，每台机器有 16 个 CPU 核心且内存为 32 GB。每节点（台机器）上混合部署 1 TiDB（实例）+ 1 TiKV（实例）+ 1 PD（实例）。

由于 PD 和 TiKV 都会存储信息到磁盘，磁盘的写入读取延迟会直接影响到 PD 和 TiKV 的服务延迟，为了防止 PD 和 TiKV 对磁盘资源的争抢导致相互影响，推荐 PD 和 TiKV 采用不同的磁盘。

在 TiUP bench 中使用 TPC-C 5000 Warehouse 数据，每次以 `terminals` 参数为 128 的并发度测试 12 小时。主要观察集群性能的稳定性指标。

下图是默认参数配置下，12 小时内集群的 QPS 监控，可以看倒有比较明显的抖动。



图 216: QPS with default config

调整参数后，稳定性得到了改善。



图 217: QPS with modified config

12.5.8.2 参数调整

上图中出现抖动的主要原因是，默认的线程池和后台任务资源分配针对的是资源比较充足的机器。在混合部署的场景下，因为可用资源需要被多个组件共享，所以需要通过配置参数进行限制。

本次测试最终采用的配置如下：

```
tikv:
    readpool.unified.max-thread-count: 6
    server.grpc-concurrency: 2
    storage.scheduler-worker-pool-size: 2
    gc.max-write-bytes-per-sec: 300K
    rocksdb.max-background-jobs: 3
    rocksdb.max-sub-compactions: 1
    rocksdb.rate-bytes-per-sec: “200M”

  tidb:
    performance.max-procs: 8
```

接下来会分别介绍这几个参数的意义和调整方法。

12.5.8.2.1 TiKV 线程池大小配置

本节的配置项会调整一些与前台业务有关的线程池的资源配比。缩减这些线程池会损失一些性能，但在混合部署场景下可用资源有限，本身也难以达到很高的性能，所以选择牺牲一小部分性能去换取整体的稳定性。

可以先以默认配置为基础运行一次实际负载的测试，观察各线程池的实际使用量，再修改这些配置项，缩减利用率不高的线程池大小。

`readpool.unified.max-thread-count`

该参数默认取值为机器线程数的 80%，因为是混合部署场景，你需要手动计算并指定该值。可以先设置成期望 TiKV 使用的 CPU 线程数的 80%。

`server.grpc-concurrency`

该参数默认为 4。因为现有部署方案的 CPU 资源有限，实际请求数也不会很多。可以把该参数值调低，后续观察监控面板保持其使用率在 80% 以下即可。

本次测试最后选择设置该参数值为 2，通过 gRPC poll CPU 面板观察，利用率正好在 80% 左右。

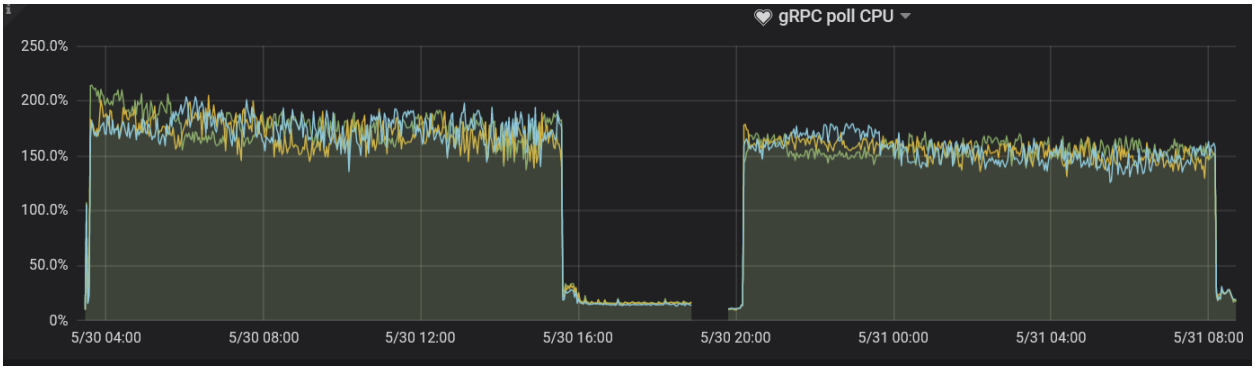


图 218: gRPC Pool CPU

`storage.scheduler-worker-pool-size`

在 TiKV 检测到机器 CPU 数大于或等于 16 时，该参数值默认为 8。CPU 数小于 16 时，该参数值默认为 4。它主要用于将复杂的事务请求转化为简单的 key-value 读写。但是 scheduler 线程池本身不进行任何写操作。

一般来说该线程池的利用率保持在 50% - 75% 之间是比较好的。和 gRPC 线程池情况类似，混合部署时该参数默认取值偏大，资源利用不充分。本次测试最后选择取值为 2，通过 Scheduler worker CPU 面板观察，利用率比较符合最佳实践。



图 219: Scheduler Worker CPU

12.5.8.2.2 TiKV 后台任务资源配置

除前台任务之外，TiKV 上还会持续地有后台任务进行数据整理和过期数据的清除。默认配置为这些后台任务分配了比较多的资源以应对大流量的写入。

混合部署场景下，默认配置就不是很合适了，需要通过以下参数对这些后台任务地资源使用量进行限制。

`rocksdb.max-background-jobs` 和 `rocksdb.max-sub-compactions`

RocksDB 线程池是进行 Compact 和 Flush 任务的线程池，默认大小为 8。这明显超出了实际可以使用的资源，需要限制。`rocksdb.max-sub-compactions` 是单个 compaction 任务的子任务并发数，默认值为 3，在写入流量不大的情况下可以进行限制。

这次测试最终将 `rocksdb.max-background-jobs` 设置为 3，将 `rocksdb.max-sub-compactions` 设置为 1。在 12 小时的 TPC-C 负载下没有发生 Write Stall，根据实际负载进行这两项参数的优化时，可以逐步调低这两个配置，并通过监控观察。

- 如果遇到了 Write Stall，可以先调大 `rocksdb.max-background-jobs` 的取值。
- 如果还是存在问题，可将 `rocksdb.max-sub-compactions` 设置为 2 或者 3。

`rocksdb.rate-bytes-per-sec`

该参数用于限制后台 compaction 任务的磁盘流量。默认配置下没有进行任何限制。为了避免 compaction 挤占前台服务资源，可以根据硬盘的顺序读写速度进行调整，为正常的服务保留足够多的磁盘带宽。

类似 compaction 线程池的调整方法，调整该参数值后，也根据是否存在 Write Stall 来判断取值是否合理。

`gc.max_write_bytes_per_sec`

因为 TiDB 使用的是 MVCC 的模型，TiKV 还需要周期性地在后台清除旧版本的数据。当可用资源有限的时候，这个操作会引起周期性的性能抖动。可以用 `gc.max_write_bytes_per_sec` 来限制这一操作的资源使用。



图 220: GC Impact

除了在配置文件中设置该参数之外，还可以通过 tikv-ctl 动态调节，为调整该参数提供便利：

```
tiup ctl:v<CLUSTER_VERSION> tikv --host=${ip:port} modify-tikv-config -n gc.
    ↪ max_write_bytes_per_sec -v ${limit}
```

> 注意：
>
> 对于更新频繁的业务场景，限制 GC 流量可能会导致 MVCC 版本堆积，进而影响读取性能。所以这个参数的取值需要进行多次尝试，在性能抖动和性能衰退之间找一个比较平衡的取值。

12.5.8.2.3 TiDB 参数调整

一般通过系统变量调整 TiDB 执行算子的优化参数，例如 `tidb_hash_join_concurrency`、`tidb_index_lookup_join_concurrency` ↪ 等。

本测试中没有对这些参数进行调整。如果实际的业务负载测试中，出现执行算子消耗过多 CPU 资源的情况，可以针对业务场景对特定的算子资源使用量进行限制。这部分内容可以参考TiDB 系统变量文档。

`performance.max-procs`

该参数控制着整个 Go 进程能使用的 CPU 核心数量，默认情况下为当前机器或者 cgroups 的 CPU 数量。

Go 运行时会定期使用一定比例的线程进行 GC 等后台工作。在混合部署模式下，如果不对这一参数进行限制，GC 等后台操作会过多占用 CPU 资源。

12.5.9 在三数据中心下就近读取数据

在三数据中心模式下，Region 的三个副本都会隔离在各个数据中心里。然而在强一致读的要求下，TiDB 的每一个查询都需要访问对应数据的 Leader 副本，而查询的来源可能和 Leader 所在的数据中心不一致，这就会引起跨数据中心的数据访问，从而造成访问的延迟上升。本文主要介绍使用Stale Read 功能，以牺牲数据实时性的方式，避免跨数据中心的访问，从而降低访问的延迟。

12.5.9.1 部署三数据中心的 TiDB 集群

部署三数据中心的方法，参考同城多数据中心部署 TiDB。

请注意，如果 TiKV 和 TiDB 都有 `labels` 配置项，在给 TiKV 和 TiDB 配置标签时，同一个数据中心下的 TiKV 和 TiDB 应该配置相同的 zone 标签。假设 TiKV 和 TiDB 都在 `dc-1` 数据中心下，那么两者都需要配置如下标签：

```
[labels]
zone=dc-1
```

12.5.9.2 使用 Stale Read 就近读取数据

Stale Read 为用户提供了一种读取历史数据的一种机制。使用 Stale Read 功能，你能从指定时间点或时间范围内读取对应的历史数据，从而避免数据同步带来延迟。在部分跨数据中心部署的场景中使用 Stale Read 功能，通过牺牲一定的实时性，TiDB 可就近访问对应数据所在当前中心的副本，避免跨数据中心的网络延迟，降低整体查询的访问延迟。

当 TiDB 收到 Stale Read 查询时，假如对应的 TiDB 配置了 zone 标签，而且`tidb_replica_read` 为 `closest-replicas`，就会将请求发送到对应数据副本所在 TiKV 拥有相同的 zone 标签的节点上。

如何使用 Stale Read 查询，参考使用 AS OF TIMESTAMP 语法读取历史数据。

12.5.10 UUID 最佳实践

12.5.10.1 UUID 概述

通用唯一标识符（UUID）用作主键而不是 AUTO_INCREMENT 整数值时，可以提供以下好处：

- UUID 可以在多个系统生成，而不会产生冲突。某些情况下可以减少到 TiDB 的网络往返次数，从而提高性能。
- 绝大多数编程语言和数据库系统都支持 UUID。
- 用在 URL 中时，UUID 不容易被枚举攻击。相比之下，使用 `auto_increment` 数字，则很容易让发票 ID 或用户 ID 被猜出。

12.5.10.2 最佳实践

12.5.10.2.1 二进制存储

UUID 文本是一个包含 36 字符的字符串，如 `ab06f63e-8fe7-11ec-a514-5405db7aad56`。使用 `UUID_TO_BIN()` 可将 UUID 文本格式转换为 16 字节的二进制格式。这样，你可以将文本存储在 `BINARY(16)` 列中。检索 UUID 时，可以使用 `BIN_TO_UUID()` 函数再将其转换回文本格式。

12.5.10.2.2 UUID 格式二进制顺序和聚簇主键

UUID_TO_BIN() 函数可以接收一个参数 (UUID) 或两个参数（第一个为 UUID，第二个为 swap_flag）。建议不要在 TiDB 中设置 swap_flag，以避免出现热点问题。

同时，你也可以在 UUID 主键上显式设置CLUSTERED 选项来避免热点问题。

为了演示 swap_flag 的效果，本文以表结构相同的两张表为例。区别在于，uuid_demo_1 表中插入的数据使用 UUID_TO_BIN(?, 0)，而 uuid_demo_2 表中使用 UUID_TO_BIN(?, 1)。

在如下的流量可视化页面，你可以看到写入操作集中在 uuid_demo_2 表的单个 Region 中，而这个表中的二进制格式字段顺序被调换过。

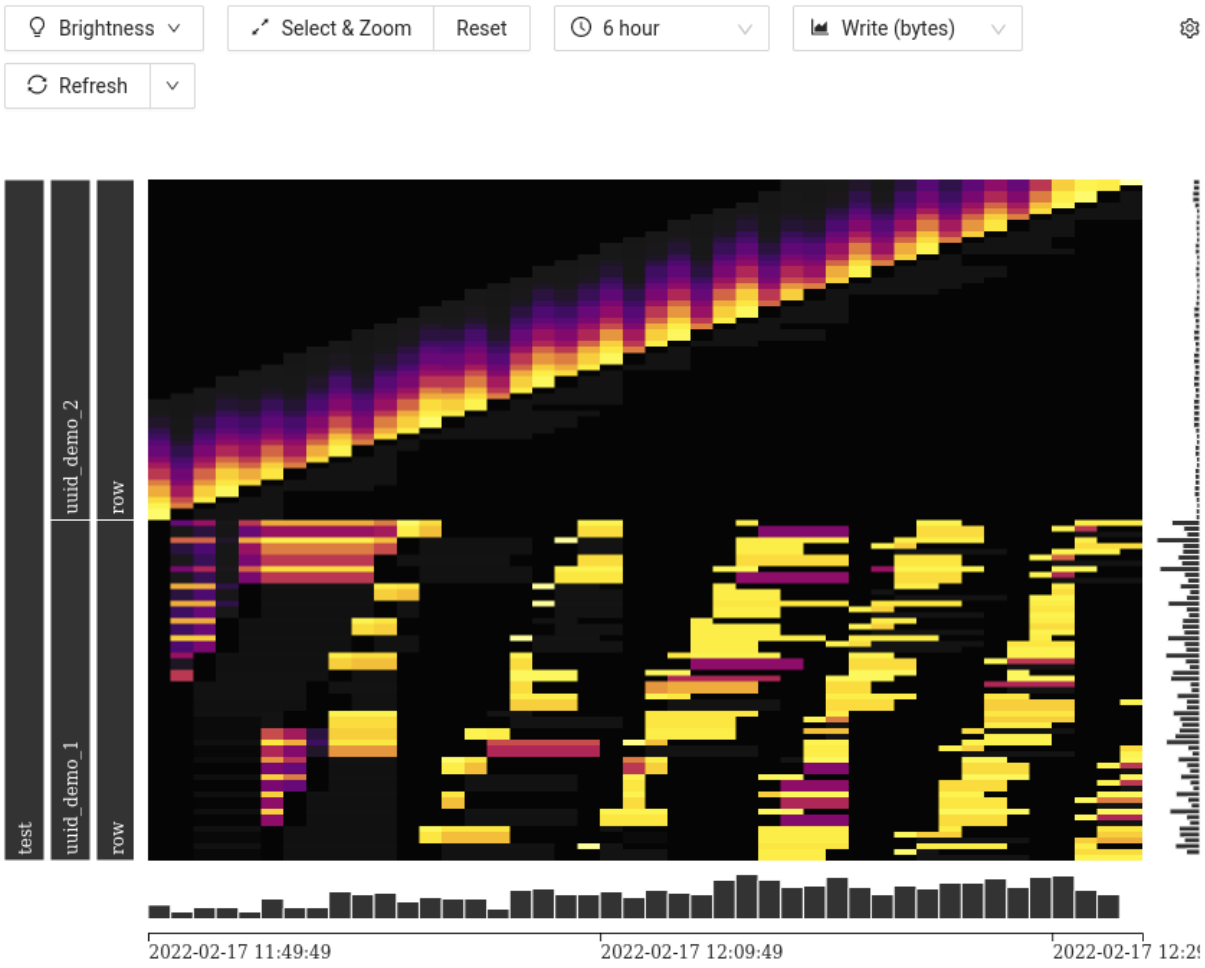


图 221: Key Visualizer

```
CREATE TABLE `uuid_demo_1` (
  `uuid` varbinary(16) NOT NULL,
  `c1` varchar(255) NOT NULL,
  PRIMARY KEY (`uuid`) CLUSTERED
```

```
)
```

```
CREATE TABLE `uuid_demo_2` (
  `uuid` varbinary(16) NOT NULL,
  `c1` varchar(255) NOT NULL,
  PRIMARY KEY (`uuid`) CLUSTERED
)
```

12.5.10.3 与 MySQL 兼容性

UUID 也可以在 MySQL 中使用。MySQL 8.0 引入了 `BIN_TO_UUID()` 和 `UUID_TO_BIN()` 函数。`UUID()` 函数在较早的 MySQL 版本中也可以使用。

12.5.11 只读存储节点最佳实践

本文档介绍如何配置只读存储节点，以及如何将备份、分析、测试等流量导向这些节点，使这些对延迟要求较低的负载与线上重要服务在物理上达到隔离的效果。

12.5.11.1 操作步骤

12.5.11.1.1 1. 将部分 TiKV 节点指定为只读节点

通过给 TiKV 节点标记特殊 label（使用 `$` 作为 label key 的前缀）的方式，可以把部分节点指定为特殊只读节点。除非通过设置 Placement Rules 的方式显式指定这些节点存储某些数据，否则 PD 不会调度任何数据到这些节点上。

只读节点可通过执行 `tiup cluster edit-config` 命令进行配置：

```
tikv_servers:
  - host: ...
    ...
    labels:
      $mode: readonly
```

12.5.11.1.2 2. 将数据以 learner 形式存储在只读节点

1. 使用如下 `pd-ctl config placement-rules` 命令导出默认 Placement Rules：

   ```
   pd-ctl config placement-rules rule-bundle load --out="rules.json"
   ```

   如果之前没有配置过 Placement Rules，那么会导出如下内容：

   ```
   [
     {
       "group_id": "pd",
   ```

```
    "group_index": 0,
    "group_override": false,
    "rules": [
      {
        "group_id": "pd",
        "id": "default",
        "start_key": "",
        "end_key": "",
        "role": "voter",
        "count": 3
      }
    ]
  }
]
```

2. 将所有数据在只读节点以 learner 方式存储一份。如下示例基于默认配置：

```
[
  {
    "group_id": "pd",
    "group_index": 0,
    "group_override": false,
    "rules": [
      {
        "group_id": "pd",
        "id": "default",
        "start_key": "",
        "end_key": "",
        "role": "voter",
        "count": 3
      },
      {
        "group_id": "pd",
        "id": "readonly",
        "start_key": "",
        "end_key": "",
        "role": "learner",
        "count": 1,
        "label_constraints": [
          {
            "key": "$mode",
            "op": "in",
            "values": [
              "readonly"
            ]
          }
```

```
        ],
        "version": 1
      }
    ]
  }
]
```

3. 执行 pd-ctl config placement-rules 命令将上面的配置写入 PD：

```
pd-ctl config placement-rules rule-bundle save --in="rules.json"
```

注意：

- 当对已存在大量数据的集群进行如上操作时，整个集群可能需要一段时间才能将数据完全复制到只读节点上。在这期间，只读节点可能无法进行服务。
- 因为备份的特殊下推实现机制，每个 label 所对应的 learner 数量不能超过 1，否则会导致在备份时产生重复数据。

12.5.11.1.3 3. 使用 Follower Read 功能读取只读节点

3.1 在 TiDB 中使用 Follower Read

你可以将系统变量tidb_replica_read 设置为 learner 来读取只读节点上的数据：

```
set tidb_replica_read=learner;
```

3.2 在 TiSpark 中使用 Follower Read

你可以在 Spark 配置文件中设置 spark.tispark.replica_read = learner 来读取只读节点上的数据：

```
spark.tispark.replica_read learner
```

3.3 在备份集群数据时只备份 Follower 节点

你可以在 br 命令行中添加 --replica-read-label 参数，来读取只读节点上的数据。注意，在 shell 中运行如下命令时需使用单引号包裹 label，以防止 $ 被 shell 解析。

```
br backup full ... --replica-read-label '$mode:readonly'
```

12.6 Placement Rules 使用文档

> **注意：**
>
> 本文介绍如何手动在 Placement Driver (PD) 中设置 Placement Rules。推荐使用Placement Rules in SQL，让你更方便地设置表和分区的放置。

Placement Rules 是 PD 在 4.0 版本引入的一套副本规则系统，用于指导 PD 针对不同类型的数据生成对应的调度。通过组合不同的调度规则，用户可以精细地控制任何一段连续数据的副本数量、存放位置、主机类型、是否参与 Raft 投票、是否可以担任 Raft leader 等属性。

Placement Rules 特性在 TiDB v5.0 及以上的版本中默认开启。如需关闭 Placement Rules 特性，请参考关闭 Placement Rules。

12.6.1 规则系统介绍

整个规则系统的配置由多条规则即 Rule 组成。每条 Rule 可以指定不同的副本数量、Raft 角色、放置位置等属性，以及这条规则生效的 key range。PD 在进行调度时，会先根据 Region 的 key range 在规则系统中查到该 Region 对应的规则，然后再生成对应的调度，来使得 Region 副本的分布情况符合 Rule。

多条规则的 key range 可以有重叠部分的，即一个 Region 能匹配到多条规则。这种情况下 PD 根据 Rule 的属性来决定规则是相互覆盖还是同时生效。如果有多条规则同时生效，PD 会按照规则的堆叠次序依次去生成调度进行规则匹配。

此外，为了满足不同来源的规则相互隔离的需求，支持更灵活的方式来组织规则，还引入了分组 (Group) 的概念。通常情况下，用户可根据规则的不同来源把规则放置在不同的 Group。

Placement Rules 示意图如下所示：

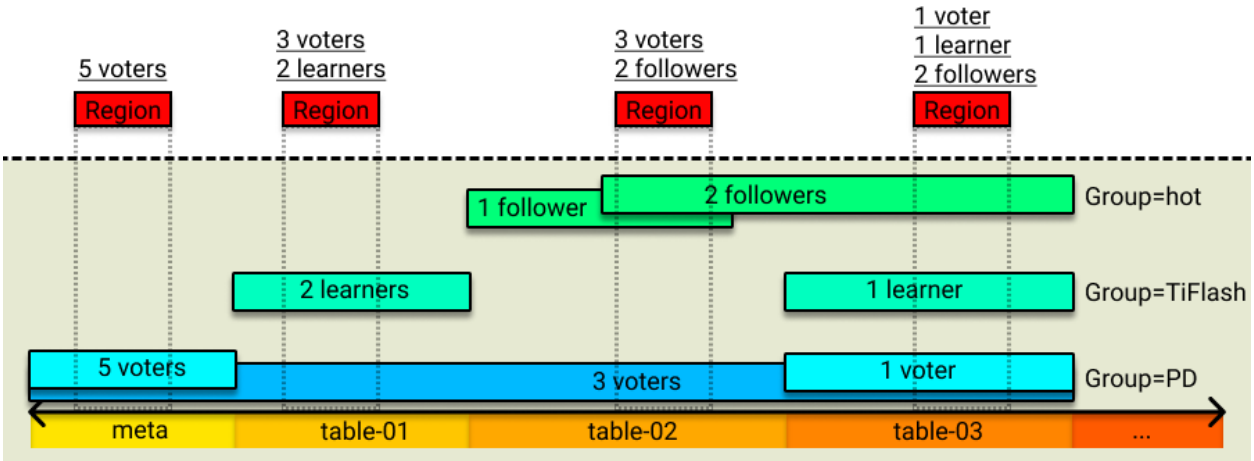


图 222: Placement rules overview

12.6.1.1 规则字段

以下是每条规则中各个字段的具体含义：

字段名	类型及约束	说明
GroupID	string	分组 ID，标识规则的来源
ID	string	分组内唯一 ID
Index	int	分组内堆叠次序
Override	true/false	是否覆盖 index 的更小 Rule（限分组内）
StartKey	string，十六进制编码	适用 Range 起始 key
EndKey	string，十六进制编码	适用 Range 终止 key
Role	string	副本角色，包括 voter/leader/follower/learner
Count	int，正整数	副本数量
LabelConstraint	[]Constraint	用于按 label 筛选节点
LocationLabels	[]string	用于物理隔离
IsolationLevel	string	用于设置最小强制物理隔离级别

LabelConstraint 与 Kubernetes 中的功能类似，支持通过 in、notIn、exists 和 notExists 四种原语来筛选 label。这四种原语的意义如下：

- in：给定 key 的 label value 包含在给定列表中。
- notIn：给定 key 的 label value 不包含在给定列表中。
- exists：包含给定的 label key。
- notExists：不包含给定的 label key。

LocationLabels 的意义和作用与 PD v4.0 之前的版本相同。比如配置 [zone,rack,host] 定义了三层的拓扑结构：集群分为多个 zone（可用区），每个 zone 下有多个 rack（机架），每个 rack 下有多个 host（主机）。PD 在调度时首先会尝试将 Region 的 Peer 放置在不同的 zone，假如无法满足（比如配置 3 副本但总共只有 2 个 zone）则保证放置在不同的 rack；假如 rack 的数量也不足以保证隔离，那么再尝试 host 级别的隔离，以此类推。

IsolationLevel 的意义和作用详细请参考配置集群拓扑。例如已配置 LocationLabels 为 [zone,rack,host] 的前提下，设置 IsolationLevel 为 zone，则 PD 在调度时会保证每个 Region 的所有 Peer 均被放置在不同的 zone。假如无法满足 IsolationLevel 的最小强制隔离级别限制（比如配置 3 副本但总共只有 2 个 zone），PD 也不会尝试补足，以满足该限制。IsolationLevel 默认值为空字符串，即禁用状态。

12.6.1.2 规则分组字段

以下是规则分组字段的含义：

字段名	类型及约束	说明
ID	string	分组 ID，用于标识规则来源
Index	int	不同分组的堆叠次序
Override	true/false	是否覆盖 index 更小的分组

如果不单独设置规则分组，默认 Override=false，对应的行为是不同分组之间相互不影响，不同分组内的规则是同时生效的。

12.6.2 配置规则操作步骤

本节的操作步骤以使用pd-ctl 工具为例，涉及到的命令也支持通过 HTTP API 进行调用。

12.6.2.1 开启 Placement Rules 特性

Placement Rules 特性在 TiDB v5.0 及以上的版本中默认开启。如需关闭 Placement Rules 特性，请参考关闭 Placement Rules。如需在关闭后重新开启该特性，可以集群初始化以前设置 PD 配置文件：

```
[replication]
enable-placement-rules = true
```

这样，PD 在初始化成功后会开启这个特性，并根据 `max-replicas` 及 `location-labels` 配置生成对应的规则：

```
{
  "group_id": "pd",
  "id": "default",
  "start_key": "",
  "end_key": "",
  "role": "voter",
  "count": 3,
  "location_labels": ["zone", "rack", "host"],
  "isolation_level": ""
}
```

如果是已经初始化过的集群，也可以通过 pd-ctl 进行在线开启：

```
pd-ctl config placement-rules enable
```

PD 同样将根据系统的 `max-replicas` 及 `location-labels` 生成默认的规则。

> **注意：**
>
> 开启 Placement Rules 后，原先的 `max-replicas` 及 `location-labels` 配置项将不再生效。如果需要调整副本策略，应当使用 Placement Rules 相关接口。

12.6.2.2 关闭 Placement Rules 特性

使用 pd-ctl 可以关闭 Placement Rules 特性，切换为之前的调度策略。

```
pd-ctl config placement-rules disable
```

> **注意：**
>
> 关闭 Placement Rules 后，PD 将使用原先的 `max-replicas` 及 `location-labels` 配置。在 Placement Rules 开启期间对 Rule 的修改不会导致这两项配置的同步更新。此外，设置好的所有 Rule 都会保留在系统中，会在下次开启 Placement Rules 时被使用。

12.6.2.3 使用 pd-ctl 设置规则

> **注意：**
>
> 规则的变更将实时地影响 PD 调度，不恰当的规则设置可能导致副本数较少，影响系统的高可用。

pd-ctl 支持使用多种方式查看系统中的 Rule，输出是 json 格式的 Rule 或 Rule 列表：

- 查看所有规则列表

```
pd-ctl config placement-rules show
```

- 查看 PD Group 的所有规则列表

```
pd-ctl config placement-rules show --group=pd
```

- 查看对应 Group 和 ID 的某条规则

```
pd-ctl config placement-rules show --group=pd --id=default
```

- 查看 Region 所匹配的规则列表

```
pd-ctl config placement-rules show --region=2
```

  上面的例子中 2 为 Region ID。

新增和编辑规则是类似的，需要把对应的规则写进文件，然后使用 save 命令保存至 PD：

```
cat > rules.json <<EOF
[
    {
        "group_id": "pd",
        "id": "rule1",
        "role": "voter",
        "count": 3,
        "location_labels": ["zone", "rack", "host"]
```

```
    },
    {
        "group_id": "pd",
        "id": "rule2",
        "role": "voter",
        "count": 2,
        "location_labels": ["zone", "rack", "host"]
    }
]
EOF
pd-ctl config placement save --in=rules.json
```

以上操作会将 rule1、rule2 两条规则写入 PD，如果系统中已经存在 GroupID+ID 相同的规则，则会覆盖该规则。

如果需要删除某条规则，只需要将规则的 count 置为 0 即可，对应 GroupID+ID 相同的规则会被删除。以下命令将删除 pd/rule2 这条规则：

```
cat > rules.json <<EOF
[
    {
        "group_id": "pd",
        "id": "rule2"
    }
]
EOF
pd-ctl config placement save --in=rules.json
```

12.6.2.4 使用 pd-ctl 设置规则分组

- 查看所有的规则分组列表

```
pd-ctl config placement-rules rule-group show
```

- 查看指定 ID 的规则分组

```
pd-ctl config placement-rules rule-group show pd
```

- 设置规则分组的 index 和 override 属性

```
pd-ctl config placement-rules rule-group set pd 100 true
```

- 删除规则分组配置（如组内还有规则，则使用默认分组配置）

```
pd-ctl config placement-rules rule-group delete pd
```

12.6.2.5 使用 pd-ctl 批量更新分组及组内规则

使用 `rule-bundle` 子命令，可以方便地同时查看和修改规则分组及组内的所有规则。

该子命令中 `get {group_id}` 用来查询一个分组，输出结果为嵌套形式的规则分组和组内规则：

```
pd-ctl config placement-rules rule-bundle get pd
```

输出示例：

```
{
  "group_id": "pd",
  "group_index": 0,
  "group_override": false,
  "rules": [
    {
      "group_id": "pd",
      "id": "default",
      "start_key": "",
      "end_key": "",
      "role": "voter",
      "count": 3
    }
  ]
}
```

`rule-bundle get` 子命令中可以添加 `--out` 参数来将输出写入文件，方便后续修改保存。

```
pd-ctl config placement-rules rule-bundle get pd --out="group.json"
```

修改完成后，使用 `rule-bundle set` 子命令将文件中的配置保存至 PD 服务器。与前面介绍的 `save` 不同，此命令会替换服务器端该分组内的所有规则。

```
pd-ctl config placement-rules rule-bundle set pd --in="group.json"
```

12.6.2.6 使用 pd-ctl 查看和修改所有配置

用户还可以使用 pd-ctl 查看和修改所有配置，即把全部配置保存至文件，修改后再覆盖保存。该操作同样使用 `rule-bundle` 子命令。

下面的命令将所有配置保存至 `rules.json` 文件：

```
pd-ctl config placement-rules rule-bundle load --out="rules.json"
```

编辑完文件后，使用下面的命令将配置保存至 PD 服务器：

```
pd-ctl config placement-rules rule-bundle save --in="rules.json"
```

12.6.2.7 使用 tidb-ctl 查询表相关的 key range

若需要针对元数据或某个特定的表进行特殊配置，可以通过 tidb-ctl 的 keyrange 命令来查询相关的 key。注意要添加 --encode 返回 PD 中的表示形式。

```
tidb-ctl keyrange --database test --table ttt --encode
```

```
global ranges:
  meta: (6d00000000000000f8, 6e00000000000000f8)
  table: (7400000000000000f8, 7500000000000000f8)
table ttt ranges: (NOTE: key range might be changed after DDL)
  table: (7480000000000000ff2d00000000000000f8, 7480000000000000ff2e00000000000000f8)
  table indexes: (7480000000000000ff2d5f690000000000fa, 7480000000000000ff2d5f720000000000fa)
    index c2: (7480000000000000ff2d5f698000000000ff0000010000000000fa, 7480000000000000
        ↪ ff2d5f698000000000ff0000020000000000fa)
    index c3: (7480000000000000ff2d5f698000000000ff0000020000000000fa, 7480000000000000
        ↪ ff2d5f698000000000ff0000030000000000fa)
    index c4: (7480000000000000ff2d5f698000000000ff0000030000000000fa, 7480000000000000
        ↪ ff2d5f698000000000ff0000040000000000fa)
  table rows: (7480000000000000ff2d5f720000000000fa, 7480000000000000ff2e00000000000000f8)
```

> **注意：**
>
> DDL 等操作会导致 table ID 发生变化，需要同步更新对应的规则。

12.6.3 典型场景示例

本部分介绍 Placement Rules 的使用场景示例。

12.6.3.1 场景一：普通的表使用 3 副本，元数据使用 5 副本提升集群容灾能力

只需要增加一条规则，将 key range 限定在 meta 数据的范围，并把 count 值设为 5。添加规则示例如下：

```
{
  "group_id": "pd",
  "id": "meta",
  "index": 1,
  "override": true,
  "start_key": "6d00000000000000f8",
  "end_key": "6e00000000000000f8",
  "role": "voter",
  "count": 5,
  "location_labels": ["zone", "rack", "host"]
}
```

12.6.3.2 场景二：5 副本按 2-2-1 的比例放置在 3 个数据中心，且第 3 个中心不产生 Leader

创建三条规则，分别设置副本数为 2、2、1，并且在每个规则内通过 `label_constraints` 将副本限定在对应的数据中心内。另外，不需要 leader 的数据中心将 `role` 改为 `follower`。

```
[
    {
        "group_id": "pd",
        "id": "zone1",
        "start_key": "",
        "end_key": "",
        "role": "voter",
        "count": 2,
        "label_constraints": [
            {"key": "zone", "op": "in", "values": ["zone1"]}
        ],
        "location_labels": ["rack", "host"]
    },
    {
        "group_id": "pd",
        "id": "zone2",
        "start_key": "",
        "end_key": "",
        "role": "voter",
        "count": 2,
        "label_constraints": [
            {"key": "zone", "op": "in", "values": ["zone2"]}
        ],
        "location_labels": ["rack", "host"]
    },
    {
        "group_id": "pd",
        "id": "zone3",
        "start_key": "",
        "end_key": "",
        "role": "follower",
        "count": 1,
        "label_constraints": [
            {"key": "zone", "op": "in", "values": ["zone3"]}
        ],
        "location_labels": ["rack", "host"]
    }
]
```

12.6.3.3 场景三：为某张表添加 2 个 TiFlash Learner 副本

为表的 row key 单独添加一条规则，限定数量为 2，并且通过 label_constraints 保证副本产生在 engine=tiflash 的节点。注意这里使用了单独的 group_id，保证这条规则不会与系统中其他来源的规则互相覆盖或产生冲突。

```
{
  "group_id": "tiflash",
  "id": "learner-replica-table-ttt",
  "start_key": "7480000000000000ff2d5f720000000000fa",
  "end_key": "7480000000000000ff2e00000000000000f8",
  "role": "learner",
  "count": 2,
  "label_constraints": [
    {"key": "engine", "op": "in", "values": ["tiflash"]}
  ],
  "location_labels": ["host"]
}
```

12.6.3.4 场景四：为某张表在有高性能磁盘的北京节点添加 2 个 Follower 副本

这个例子展示了比较复杂的 label_constraints 配置，下面的例子限定了副本放置在 bj1 或 bj2 机房，且磁盘类型为 nvme。

```
{
  "group_id": "follower-read",
  "id": "follower-read-table-ttt",
  "start_key": "7480000000000000ff2d00000000000000f8",
  "end_key": "7480000000000000ff2e00000000000000f8",
  "role": "follower",
  "count": 2,
  "label_constraints": [
    {"key": "zone", "op": "in", "values": ["bj1", "bj2"]},
    {"key": "disk", "op": "in", "values": ["nvme"]}
  ],
  "location_labels": ["host"]
}
```

12.6.3.5 场景五：将某张表迁移至 SSD 节点

与场景三不同，这个场景不是要在原有配置的基础上增加新副本，而是要强制覆盖一段数据的其它配置，因此需要通过配置规则分组来指定一个足够大的 index 以及设置 override 来覆盖原有规则。

规则：

```
{
  "group_id": "ssd-override",
  "id": "ssd-table-45",
  "start_key": "7480000000000000ff2d5f720000000000fa",
```

```
  "end_key": "7480000000000000ff2e00000000000000f8",
  "role": "voter",
  "count": 3,
  "label_constraints": [
    {"key": "disk", "op": "in", "values": ["ssd"]}
  ],
  "location_labels": ["rack", "host"]
}
```

规则分组：

```
{
  "id": "ssd-override",
  "index": 1024,
  "override": true,
}
```

12.7 Load Base Split

Load Base Split 是 TiKV 在 4.0 版本引入的特性，旨在解决 Region 访问分布不均匀造成的热点问题，比如小表的全表扫描。

12.7.1 场景描述

在 TiDB 中，当流量集中在某些节点时很容易形成热点。PD 会尝试通过调度 Hot Region，尽可能让这些 Hot Region 均匀分布在各个节点上，以求获得更好的性能。

但是 PD 的调度的最小粒度是 Region。如果集群的热点数目少于节点数目，或者说存在某几个热点流量远高于其他 Region，对 PD 的热点调度来说，能做到的也只是让热点从一个节点转移到另一个节点，而无法让整个集群承担负载。

这种场景在读请求居多的 workload 中尤为常见。例如对小表的全表扫描和索引查找，或者是对一些字段的频繁访问。

在此之前解决此类问题的办法是手动输入命令去拆分一个或几个热点 Region，但是这样的操作存在以下两个问题：

- 均匀拆分 Region 并不一定是最好的选择，请求可能集中在某几个 Key 上，即使均匀拆分后热点可能仍然集中在其中一个 Region 上，可能需要经过多次均匀拆分才能达到目标。
- 人工介入不够及时和易用。

12.7.2 实现原理

Load Base Split 会基于统计信息自动拆分 Region。通过统计信息识别出读流量或 CPU 使用率在 10s 内持续超过阈值的 Region，并在合适的位置将这些 Region 拆分。在选择拆分的位置时，会尽可能平衡拆分后两个 Region 的访问量，并尽量避免跨 Region 的访问。

Load Base Split 后的 Region 不会被迅速 Merge。一方面，PD 的 MergeChecker 会跳过 hot Region，另一方面 PD 也会针对心跳信息中的 QPS去进行判断，避免 Merge 两个 QPS 很高的 Region。

12.7.3 使用方法

目前的 Load Base Split 的控制参数如下：

- split.qps-threshold：表明一个 Region 被识别为热点的 QPS 阈值，默认为每秒 3000 QPS。
- split.byte-threshold：自 v5.0 引入，表明一个 Region 被识别为热点的流量阈值，单位为 Byte，默认为每秒 30 MiB 流量。
- split.region-cpu-overload-threshold-ratio：自 v6.2.0 引入，表明一个 Region 被识别为热点的 CPU 使用率（占读线程池 CPU 时间的百分比）阈值，默认为 0.25。

如果连续 10s 内，某个 Region 每秒的各类读请求之和超过了 split.qps-threshold、流量超过了 split.byte-threshold，或 CPU 使用率在 Unified Read Pool 内的占比超过了 split.region-cpu-overload-threshold-ratio，那么就会尝试对此 Region 进行拆分。

目前默认开启 Load Base Split，但配置相对保守。如果想要关闭这个功能，将 QPS 和 Byte 阈值全部调到足够高并将 CPU 占比阈值调为 0 即可。

目前有两种办法修改配置：

- 通过 SQL 语句修改，例如：

```
# 设置 QPS 阈值为 1500
SET config tikv split.qps-threshold=1500;
# 设置 Byte 阈值为 15 MiB (15 * 1024 * 1024)
SET config tikv split.byte-threshold=15728640;
# 设置 CPU 使用率阈值为 50%
SET config tikv split.region-cpu-overload-threshold-ratio=0.5;
```

- 通过 TiKV 修改，例如：

```
curl -X POST "http://ip:status_port/config" -H "accept: application/json" -d '{"split.qps-
    ↪ threshold":"1500"}'
curl -X POST "http://ip:status_port/config" -H "accept: application/json" -d '{"split.byte-
    ↪ threshold":"15728640"}'
curl -X POST "http://ip:status_port/config" -H "accept: application/json" -d '{"split.region
    ↪ -cpu-overload-threshold-ratio":"0.5"}'
```

同理，目前也有两种办法查看配置：

- 通过 SQL 查看，例如：

```
show config where type='tikv' and name like '%split.qps-threshold%'
```

- 通过 TiKV 查看，例如：

```
curl "http://ip:status_port/config"
```

> 注意：
>
> 从 v4.0.0-rc.2 起可以使用 SQL 语句来修改和查看配置。

12.8 Store Limit

Store Limit 是 PD 在 3.0 版本引入的特性，旨在能够更加细粒度地控制调度的速度，针对不同调度场景进行调优。

12.8.1 实现原理

PD 的调度是以 operator 为单位执行的。一个 operator 可能包含多个调度操作。示例如下：

```
"replace-down-replica {mv peer: store [2] to [3]} (kind:region,replica, region:10(4,5), createAt
    ↪ :2020-05-18 06:40:25.775636418 +0000 UTC m=+2168762.679540369, startAt:2020-05-18
    ↪ 06:40:25.775684648 +0000 UTC m=+2168762.679588599, currentStep:0, steps:[add learner peer
    ↪  20 on store 3, promote learner peer 20 on store 3 to voter, remove peer on store 2])"
```

以上示例中，`replace-down-replica` 这个 operator 具体包含以下操作：

1. 在 `store 3` 上添加一个 learner peer，ID 为 20。
2. 将 `store 3` 上 ID 为 20 的 learner peer 提升为 voter。
3. 删除 `store 2` 上的 peer。

Store Limit 是通过在内存中维护了一个 store ID 到令牌桶的映射，来实现 store 级别的限速。这里不同的操作对应不同的令牌桶，目前仅支持限制添加 learner/peer 和删除 peer 两种操作的速度，即对应于每个 store 存在两种类型的令牌桶。

每次 operator 产生后会检查所包含的操作对应的令牌桶中是否有足够的 token。如果 token 充足才会将该 operator 加入到调度的队列中，同时从令牌桶中拿走对应的 token，否则该 operator 被丢弃。令牌桶会按照固定的速率补充 token，从而实现限速的目的。

Store Limit 与 PD 其他 limit 相关的参数（如 `region-schedule-limit`，`leader-schedule-limit` 等）不同的是，Store Limit 限制的主要是 operator 的消费速度，而其他的 limit 主要是限制 operator 的产生速度。引入 Store Limit 特性之前，调度的限速主要是全局的，所以即使限制了全局的速度，但还是有可能存在调度都集中在部分 store 上面，因而影响集群的性能。而 Store Limit 通过将限速的粒度进一步细化，可以更好的控制调度的行为。

12.8.2 使用方法

Store Limit 相关的参数可以通过 `pd-ctl` 进行设置。

12.8.2.1 查看当前 store 的 limit 设置

查看当前 store 的 limit 示例如下：

```
store limit                          // 显示所有 store 添加和删除 peer 的速度上限。
store limit add-peer                 // 显示所有 store 添加 peer 的速度上限。
store limit remove-peer              // 显示所有 store 删除 peer 的速度上限。
```

12.8.2.2 设置全部 store 的 limit

设置全部 store 的 limit 示例如下：

```
store limit all 5                    // 设置所有 store 添加和删除 peer 的速度上限为每分钟 5 个。
store limit all 5 add-peer           // 设置所有 store 添加 peer 的速度上限为每分钟 5 个。
store limit all 5 remove-peer        // 设置所有 store 删除 peer 的速度上限为每分钟 5 个。
```

12.8.2.3 设置单个 store 的 limit

设置单个 store 的 limit 示例如下：

```
store limit 1 5                      // 设置 store 1 添加和删除 peer 的速度上限为每分钟 5 个。
store limit 1 5 add-peer             // 设置 store 1 添加 peer 的速度上限为每分钟 5 个。
store limit 1 5 remove-peer          // 设置 store 1 删除 peer 的速度上限为每分钟 5 个。
```

12.9 DDL 语句的执行原理及最佳实践

本文介绍了 TiDB 中 DDL 语句的执行原理（包括 DDL Owner 模块和在线变更 DDL 的流程）和最佳实践。

12.9.1 DDL 执行原理

TiDB 采用在线异步变更的方式执行 DDL 语句，从而实现 DDL 语句的执行不会阻塞其他会话中的 DML 语句。因此，在业务执行过程中，你可以通过在线异步变更 DDL 对数据库对象定义进行变更。

12.9.1.1 DDL 语句类型简介

按照执行期间是否阻塞用户业务，DDL 语句可以划分为：

- 离线 DDL 语句：即数据库接收到用户的 DDL 语句后，会先对要修改的数据库对象进行加锁，再执行元数据变更，在 DDL 执行过程中将阻塞用户业务对数据的修改。
- 在线 DDL 语句：即数据库在执行 DDL 语句时，通过一定的方法，使得 DDL 执行不阻塞用户业务，且能够保证用户业务可在 DDL 执行期间提交修改，在执行过程中保证对应对象的数据正确性与一致性。

按照是否需要操作 DDL 目标对象所包括的数据来划分，DDL 语句可以划分为：

- 逻辑 DDL 语句：通常只修改数据库对象的元数据，不对变更对象存储的数据进行处理，例如变更表名或变更列名。

  在 TiDB 中，逻辑 DDL 语句又被称为 General DDL。General DDL 的执行时间通常较短，只需要几十毫秒或者几秒。执行这类 DDL 语句几乎不消耗系统资源，因此不会影响业务负载。

- 物理 DDL 语句：不但会修改变更对象的元数据，同时也修改变更对象所存储的用户数据。例如，为表创建索引，不仅需要变更表的定义，同时也需要做一次全表扫描以构建新增加的索引。

  在 TiDB 中，物理 DDL 被称为 Reorg DDL（Reorg 即 Reorganization）。目前物理 DDL 只包含 `ADD INDEX` 以及有损列类型变更（例如从 `INT` 转成 `CHAR` 类型）这两种类型。物理 DDL 的特点是执行时间较长，且执行时间与表的数据量、机器配置以及业务负载有关。

  执行物理 DDL 会影响业务负载，具体有两个方面。一方面需要从 TiKV 中读取数据并写入新数据，因此会消耗 TiKV 的 CPU 及 I/O 资源。另一方面，DDL Owner 所在的 TiDB 节点需要进行相应的计算，因此会消耗更多的 CPU 资源。由于目前 TiDB 还不支持分布式执行 DDL 语句，因此其他 TiDB 节点不会占用更多的系统资源。

> 注意：
>
> DDL 语句对用户业务的影响通常都是由于执行物理 DDL 任务造成的。因此要优化 DDL 语句对于用户业务的影响，重点在于对物理 DDL 任务执行期间的设计，降低对于用户业务的影响。

12.9.1.2 TiDB DDL 模块

TiDB DDL 模块引入 DDL Owner（简称 Owner）角色来代理执行所有进入到 TiDB 集群中的 DDL 语句。对于当前 TiDB DDL 模块的实现，在同一时间，整个 TiDB 集群中只有一个 TiDB 节点能当选为 Owner。当选 Owner 后，TiDB 节点中启动的 worker 才能处理集群中的 DDL 任务。

TiDB 通过 etcd 的选举功能从多个 TiDB 节点中选举出一个节点来担任 Owner 的宿主节点。默认情况下，每个 TiDB 节点都可能当选 Owner（你可以通过配置 `run-ddl` 控制某个 TiDB 节点是否竞选 Owner）。Owner 节点是有任期的，并会主动维护自己的任期，即续约。当 Owner 节点宕机后，其他节点可以通过 etcd 感知到并重新选举出新的 Owner，在集群中继续担任 DDL 任务执行者的角色。

DDL Owner 的简单示意图如下：

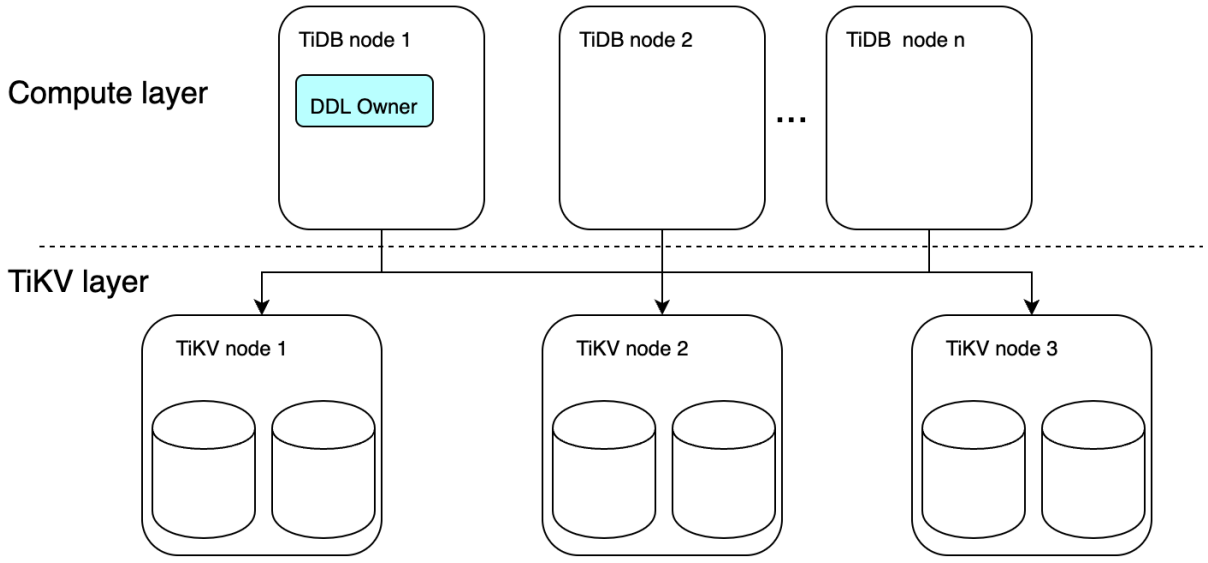


图 223: DDL Owner

你可以通过 ADMIN SHOW DDL 语句查看当前 DDL owner：

```
ADMIN SHOW DDL;
```

```
+--
    ↪ ----------+--------------------------------------+--------------+--------------+-------------------------
    ↪
| SCHEMA_VER | OWNER_ID                             | OWNER_ADDRESS | RUNNING_JOBS | SELF_ID
    ↪                              | QUERY |
+--
    ↪ ----------+--------------------------------------+--------------+--------------+-------------------------
    ↪
|         26 | 2d1982af-fa63-43ad-a3d5-73710683cc63 | 0.0.0.0:4000  |              | 2d1982af-
    ↪ fa63-43ad-a3d5-73710683cc63 |       |
+--
    ↪ ----------+--------------------------------------+--------------+--------------+-------------------------
    ↪
1 row in set (0.00 sec)
```

12.9.1.3 TiDB 在线 DDL 异步变更的原理

TiDB DDL 模块从设计之初就选择了在线异步变更的模式，为 TiDB 的用户提供了不停机变更业务的服务能力。

DDL 变更即两个状态之间的切换（变更前 -> 变更后）。在线变更 DDL，是在两个状态之间引入多个相互兼容的小版本状态。同时，在同一个 DDL 语句在执行期间，对于同一集群中不同 TiDB 节点，允许不同节点的变更对象小版本存在不同（集群中各 TiDB 节点变更对象的小版本差距不超过两个版本），因为相邻两个小版本之间可以相互兼容。

通过多个小版本演进的方式，确保多个 TiDB 节点能够正确同步元数据，并保证期间执行用户事务更改数据的正确性与一致性。

以 ADD INDEX 为例，整个变更状态流程如下：

```
absent -> delete only -> write only -> write reorg -> public
```

对于用户来说，新建的索引在 public 状态前都不可用。

在 v6.2.0 之前，TiDB SQL 层中处理异步 Schema 变更的基本流程如下：

1. MySQL Client 发送给 TiDB server 一个 DDL 操作请求。
2. 某个 TiDB server 收到请求（即 TiDB server 的 MySQL Protocol 层对请求进行解析优化），然后发送到 TiDB SQL 层进行执行。

   TiDB SQL 层接到 DDL 请求后，会启动 start job 模块根据请求将请求封装成特定的 DDL Job（即 DDL 任务），然后将此 Job 按语句类型分类，分别存储到 KV 层的对应 DDL Job 队列，并通知自身对应的 worker 有 Job 需要处理。
3. 接收到处理 Job 通知的 worker，会判断自身是否处于 DDL Owner 的角色。如果是 Owner 角色则直接处理此 Job。如果没有处于 Owner 角色则退出不做任何处理。

   假设某台 TiDB server 不是 Owner 角色，那么其他某个节点一定有一个是 Owner。处于 Owner 角色的节点的 worker 通过定期检测机制来检查是否有 Job 可以被执行。如果发现有 Job，那么 worker 就会处理该 Job。
4. Worker 处理完 Job 后，会将此 Job 从 KV 层对应的 Job queue 中移除，并放入 job history queue。之前封装 Job 的 start job 模块会定期在 job history queue 中查看是否有已经处理完成的 Job 的 ID。如果有，则这个 Job 对应的整个 DDL 操作结束执行。
5. TiDB server 将 DDL 处理结果返回至 MySQL Client。

在 TiDB v6.2.0 前，该 DDL 执行框架存在以下限制：

- TiKV 集群中只有 general job queue 和 add index job queue 两个队列，分别处理逻辑 DDL 和物理 DDL。
- DDL Owner 总是以先入先出的方式处理 DDL Job。
- DDL Owner 每次只能执行一个同种类型（逻辑或物理）的 DDL 任务，这个约束较为严格。

这些限制可能会导致一些“非预期”的 DDL 阻塞行为。具体可以参考SQL FAQ - DDL 执行。

在 TiDB v6.2 之前，由于 Owner 每次只能执行一个同种类型（逻辑或物理）的 DDL 任务，这个约束较为严格，同时影响用户体验。

当 DDL 任务之间不存在相关依赖时，并行执行并不会影响数据正确性和一致性。例如：用户 A 在 T1 表上增加一个索引，同时用户 B 从 T2 表删除一列。这两条 DDL 语句可以并行执行。

为了提升 DDL 执行的用户体验，从 v6.2.0 起，TiDB 对原有的 DDL Owner 角色进行了升级，使得 Owner 能对 DDL 任务做相关性判断，判断逻辑如下：

- 涉及同一张表的 DDL 相互阻塞。
- DROP DATABASE 和数据库内所有对象的 DDL 互相阻塞。
- 涉及不同表的加索引和列类型变更可以并发执行。

- 逻辑 DDL 需要等待之前正在执行的逻辑 DDL 执行完才能执行。
- 其他情况下 DDL 可以根据 Concurrent DDL 并行度可用情况确定是否可以执行。

具体来说，TiDB 在 v6.2.0 中对 DDL 执行框架进行了如下升级：

- DDL Owner 能够根据以上判断逻辑并行执行 DDL 任务。
- 改善了 DDL Job 队列先入先出的问题。DDL Owner 不再选择当前队列最前面的 DDL Job，而是选择当前可以执行的 DDL Job。
- 扩充了处理物理 DDL 的 worker 数量，使得能够并行地添加多个物理 DDL。

  因为 TiDB 中所有支持的 DDL 任务都是以在线变更的方式来实现的，TiDB 通过 Owner 即可对新的 DDL Job 进行相关性判断，并根据相关性结果进行 DDL 任务的调度，从而使分布式数据库实现了和传统数据库中 DDL 并发相同的效果。

并发 DDL 框架的实现进一步加强了 TiDB 中 DDL 语句的执行能力，并更符合商用数据库的使用习惯。

12.9.2 最佳实践

12.9.2.1 通过系统变量来平衡物理 DDL 的执行速度与对业务负载的影响

执行物理 DDL（包括添加索引或列类型变更）时，适当调整以下系统变量可以平衡 DDL 执行速度与对业务负载的影响：

- `tidb_ddl_reorg_worker_cnt`：这个变量用来设置 DDL 操作 reorg worker 的数量，控制回填的并发度。
- `tidb_ddl_reorg_batch_size`：这个变量用来设置 DDL 操作 `re-organize` 阶段的 batch size，以控制回填的数据量。

  推荐值：

  - 在无其他负载情况下，如需让 `ADD INDEX` 尽快完成，可以将 `tidb_ddl_reorg_worker_cnt` 和 `tidb_ddl_reorg_batch_size` 的值适当调大，例如将两个变量值分别调为 20 和 2048。
  - 在有其他负载情况下，如需让 `ADD INDEX` 尽量不影响其他业务，可以将 `tidb_ddl_reorg_worker_cnt` 和 `tidb_ddl_reorg_batch_size` 适当调小，例如将两个变量值分别调为 4 和 256。

建议：

- 以上两个变量均可以在 DDL 任务执行过程中动态调整，并且在下一个事务批次中生效。
- 根据 DDL 操作的类型，并结合业务负载压力，选择合适的时间点执行，例如建议在业务负载比较低的情况运行 `ADD INDEX` 操作。
- 由于添加索引的时间跨度较长，发送相关的指令后，TiDB 会在后台执行任务，TiDB server 宕机不会影响继续执行。

12.9.2.2 并发发送 DDL 请求实现快速建大量表

一个建表的操作耗时大约 50 毫秒。受框架的限制，建表耗时可能更长。

为了更快地建表，推荐通过并发发送多个 DDL 请求以达到最快建表速度。如果串行地发送 DDL 请求，并且没有发给 Owner 节点，则建表速度会很慢。

12.9.2.3 在一条 ALTER 语句中进行多次变更

自 v6.2.0 起，TiDB 支持在一条 ALTER 语句中修改一张表的多个模式对象（如列、索引），同时保证整个语句的原子性。因此推荐在一条 ALTER 语句中进行多次变更。

12.9.2.4 检查读写性能

在添加索引时，回填数据阶段会对集群造成一定的读写压力。在 ADD INDEX 的命令发送成功后，并且在 write reorg 阶段，建议检查 Grafana 面板上 TiDB 和 TiKV 读写相关的性能指标，以及业务响应时间，来确定 ADD INDEX 操作对集群是否造成影响。

12.9.3 DDL 相关的命令介绍

- ADMIN SHOW DDL：用于查看 TiDB DDL 的状态，包括当前 schema 版本号、DDL Owner 的 DDL ID 和地址、正在执行的 DDL 任务和 SQL、当前 TiDB 实例的 DDL ID。详情参阅ADMIN SHOW DDL。
- ADMIN SHOW DDL JOBS：查看集群环境中的 DDL 任务运行中详细的状态。详情参阅ADMIN SHOW DDL JOBS。
- ADMIN SHOW DDL JOB QUERIES job_id [, job_id]：用于查看 job_id 对应的 DDL 任务的原始 SQL 语句。详情参阅ADMIN SHOW DDL JOB QUERIES。
- ADMIN CANCEL DDL JOBS job_id [, job_id]：用于取消已经提交但未执行完成的 DDL 任务。取消完成后，执行 DDL 任务的 SQL 语句会返回 ERROR 8214 (HY000): Cancelled DDL job 错误。

  取消一个已经执行完成的 DDL 任务会在 RESULT 列看到 DDL Job:90 not found 的错误，表示该任务已从 DDL 等待队列中被移除。

12.9.4 常见问题

DDL 语句执行相关的常见问题，参考SQL FAQ - DDL 执行。

13 TiDB 工具

13.1 TiDB 工具功能概览

TiDB 提供了丰富的工具，可以帮助你进行部署运维、数据管理（例如，数据迁移、备份恢复、数据校验）、在 TiKV 上运行 Spark SQL。请根据需要选择适用的工具。

13.1.1 部署运维工具

TiDB 提供了 TiUP、TiDB Operator 和 TiUniManager 三种部署运维工具，满足你在不同系统环境下的部署运维需求。

13.1.1.1 在物理机或虚拟机上部署运维 TiDB

13.1.1.1.1 TiUP

TiUP 是在物理机或虚拟机上的 TiDB 包管理器，管理着 TiDB 的众多的组件，如 TiDB、PD、TiKV 等。当你想要运行 TiDB 生态中任何组件时，只需要执行一行 TiUP 命令即可。

TiUP cluster 是 TiUP 提供的使用 Golang 编写的集群管理组件，通过 TiUP cluster 组件就可以进行日常的运维工作，包括部署、启动、关闭、销毁、弹性扩缩容、升级 TiDB 集群，以及管理 TiDB 集群参数。

基本信息：

- 术语及核心概念
- 使用 TiUP 部署 TiDB 集群
- TiUP 组件管理
- 适用 TiDB 版本：v4.0 及以上

13.1.1.1.2 TiUniManager

TiUniManager 是一款以 TiDB 数据库为核心的数据库管理平台，帮助用户在本地部署环境或公有云环境中管理 TiDB 集群。

TiUniManager 不仅提供对 TiDB 集群的全生命周期的可视化管理，也同时一站式提供 TiDB 数据库参数管理、数据库版本升级、克隆集群、主备集群切换、数据导入导出、数据同步、数据备份恢复服务，能有效提高 TiDB 集群运维效率，降低企业运维成本。

基本信息：

- TiUniManager 使用场景
- TiUniManager 安装和运维指南
- TiUniManager 与 TiUP 的关系

13.1.1.2 在 Kubernetes 上部署运维 TiDB - TiDB Operator

TiDB Operator 是 Kubernetes 上的 TiDB 集群自动运维系统，提供包括部署、升级、扩缩容、备份恢复、配置变更的 TiDB 全生命周期管理。借助 TiDB Operator，TiDB 可以无缝运行在公有云或自托管的 Kubernetes 集群上。

基本信息：

- TiDB Operator 架构
- 在 Kubernetes 上部署运维 TiDB 快速上手
- 适用 TiDB 版本：v2.1 及以上

13.1.2 数据管理工具

TiDB 提供了丰富的数据管理工具，例如数据迁移、导入导出、备份恢复、增量同步、数据校验等。

13.1.2.1 数据迁入 - TiDB Data Migration (DM)

TiDB Data Migration (DM) 是将 MySQL/MariaDB 数据迁移到 TiDB 的工具，支持全量数据的迁移和增量数据的复制。

基本信息：

- TiDB DM 的输入：MySQL/MariaDB
- TiDB DM 的输出：TiDB 集群
- 适用 TiDB 版本：所有版本
- Kubernetes 支持：使用 TiDB Operator 在 Kubernetes 上部署 TiDB DM。

如果数据量在 TB 级别以下，推荐直接使用 TiDB DM 迁移 MySQL/MariaDB 数据到 TiDB（迁移的过程包括全量数据的导出导入和增量数据的复制）。

如果数据量在 TB 级别，推荐的迁移步骤如下：

1. 使用Dumpling 导出 MySQL/MariaDB 全量数据。
2. 使用TiDB Lightning 将全量导出数据导入 TiDB 集群。
3. 使用 TiDB DM 复制 MySQL/MariaDB 增量数据到 TiDB。

> 注意：
>
> - 原 Syncer 工具已停止维护，不再推荐使用，相关场景请使用 TiDB DM 的增量复制模式进行替代。

13.1.2.2 全量导出 - Dumpling

Dumpling 是一个用于从 MySQL/TiDB 进行全量逻辑导出的工具。

基本信息：

- Dumpling 的输入：MySQL/TiDB 集群
- Dumpling 的输出：SQL/CSV 文件
- 适用 TiDB 版本：所有版本
- Kubernetes 支持：尚未支持

> 注意：
>
> PingCAP 之前维护的 Mydumper 工具 fork 自 Mydumper project，针对 TiDB 的特性进行了优化。Mydumper 已经被Dumpling 工具取代，并使用 Go 语言编写，支持更多针对 TiDB 特性的优化。建议切换到 Dumpling。

13.1.2.3 全量导入 - TiDB Lightning

TiDB Lightning 是一个用于将全量数据导入到 TiDB 集群的工具。

使用 TiDB Lightning 导入数据到 TiDB 时，有以下模式：

- `Physical Import Mode` 模式：TiDB Lightning 将数据解析为有序的键值对，并直接将其导入 TiKV。这种模式一般用于导入大量的数据（TB 级别）到新集群，但在数据导入过程中集群无法提供正常的服务。
- `Logical Import Mode` 模式：以 TiDB/MySQL 作为后端，这种模式相比 `Physical Import Mode`，导入速度较慢，但是可以在线导入，同时也支持将数据导入到 MySQL。

基本信息：

- TiDB Lightning 的输入：
  - Dumping 输出文件
  - 其他格式兼容的 CSV 文件
  - 从 Aurora 或者 Hive 导出的 Parquet 文件
- 适用 TiDB 版本：v2.1 及以上
- Kubernetes 支持：使用 TiDB Lightning 快速恢复 Kubernetes 上的 TiDB 集群数据

> 注意：
>
> 原 Loader 工具已停止维护，不再推荐使用。相关场景请使用 TiDB Lightning 的 `tidb` 模式进行替代。

13.1.2.4 备份和恢复 - Backup & Restore

Backup & Restore (BR) 是一个对 TiDB 进行分布式备份和恢复的工具，可以高效地对大数据量的 TiDB 集群进行数据备份和恢复。

基本信息：

- 备份输出和恢复输入的文件类型
- 适用 TiDB 版本：v4.0 及以上
- Kubernetes 支持：使用 BR 工具备份 TiDB 集群数据到兼容 S3 的存储，使用 BR 工具恢复 S3 兼容存储上的备份数据

13.1.2.5 TiDB 增量数据同步 - TiCDC

TiCDC 是一款通过拉取 TiKV 变更日志实现的 TiDB 增量数据同步工具，具有将数据还原到与上游任意 TSO 一致状态的能力，同时提供开放数据协议 (TiCDC Open Protocol)，支持其他系统订阅数据变更。

基本信息：

- TiCDC 的输入：TiDB 集群
- TiCDC 的输出：TiDB 集群、 MySQL、 Kafka、 Confluent
- 适用 TiDB 版本：v4.0.6 及以上

13.1.2.6 TiDB 增量日志同步 - TiDB Binlog

TiDB Binlog 是收集 TiDB 的增量 binlog 数据，并提供准实时同步和备份的工具。该工具可用于 TiDB 集群间的增量数据同步，如将其中一个 TiDB 集群作为另一个 TiDB 集群的从集群。

基本信息：

- TiDB Binlog 的输入：TiDB 集群
- TiDB Binlog 的输出：TiDB 集群、MySQL、Kafka 或者增量备份文件
- 适用 TiDB 版本：v2.1 及以上
- Kubernetes 支持：TiDB Binlog 运维文档，Kubernetes 上的 TiDB Binlog Drainer 配置

13.1.2.7 数据校验 - sync-diff-inspector

sync-diff-inspector 是一个用于校验 MySQL/TiDB 中两份数据是否一致的工具。该工具还提供了修复数据的功能，可用于修复少量不一致的数据。

基本信息：

- sync-diff-inspector 的输入：TiDB、MySQL
- sync-diff-inspector 的输出：TiDB、MySQL
- 适用 TiDB 版本：所有版本

13.1.3 OLAP 分析工具 - TiSpark

TiSpark 是 PingCAP 为解决用户复杂 OLAP 需求而推出的产品。它借助 Spark 平台，同时融合 TiKV 分布式集群的优势，和 TiDB 一起为用户一站式解决 HTAP (Hybrid Transactional/Analytical Processing) 的需求。

13.2 TiDB 工具的使用场景

本文档从数据迁移工具的使用场景出发，介绍部分常见场景下的迁移工具的选择。

13.2.1 在物理机或虚拟机上部署运维 TiDB

当需要在物理机或虚拟机上部署运维 TiDB 时，你可以先安装TiUP，再通过 TiUP 管理 TiDB 的众多组件，如 TiDB、PD、TiKV 等。

13.2.2 在 Kubernetes 上部署运维 TiDB

当需要在 Kubernetes 上部署运维 TiDB 时，你可以先创建 Kubernetes 集群，部署TiDB Operator，然后使用 TiDB Operator 部署运维 TiDB 集群。

13.2.3 从 CSV 导入数据到 TiDB

当需要将其他工具导出的格式兼容的 CSV files 导入到 TiDB 时，可使用TiDB Lightning。

13.2.4 从 MySQL/Aurora 导入全量数据

当需要从 MySQL/Aurora 导入全量数据时，可先使用Dumpling 将数据导出为 SQL dump files，然后再使用TiDB Lightning 将数据导入到 TiDB 集群。

13.2.5 从 MySQL/Aurora 迁移数据

当既需要从 MySQL/Aurora 导入全量数据，又需要迁移增量数据时，可使用TiDB Data Migration (DM) 完成从 Amazon Aurora 迁移数据到 TiDB。

如果全量数据量较大（TB 级别），则可先使用Dumpling 与TiDB Lightning 完成全量数据的迁移，再使用 DM 完成增量数据的迁移。

13.2.6 TiDB 集群备份与恢复

当需要对 TiDB 集群进行备份或在之后对 TiDB 集群进行恢复时，可使用BR。

13.2.7 迁出数据到 TiDB

当需要将 TiDB 集群的数据迁出到其他 TiDB 集群时，可使用Dumpling 从 TiDB 将全量数据导出为 SQL dump files，然后再使用TiDB Lightning 将数据导入到 TiDB。

如果还需要执行增量数据的迁移，则可使用TiCDC。

13.2.8 TiDB 增量数据订阅

当需要订阅 TiDB 增量数据的变更时，可使用TiCDC。

13.3 TiDB 工具下载

本文介绍如何下载 TiDB 工具包以及TiUniManager。关于 TiDB 工具包的内容，请查看TiDB 离线包。

13.3.1 TiDB 工具包下载

TiDB 工具包中包含了一些常用的 TiDB 工具，例如数据导出工具 Dumpling、数据导入工具 TiDB Lightning、备份恢复工具 BR。

> 建议：
>
> 如果你的部署环境能访问互联网，无需单独下载 TiDB 工具包，可以直接通过使用TiUP 命令一键部署所需的 TiDB 工具。

13.3.1.1 环境要求

- 操作系统：Linux
- 架构：amd64 或 arm64

13.3.1.2 下载步骤

1. 访问 TiDB 社区版页面。
2. 找到 TiDB-community-toolkit 软件包，点击立即下载。

> 注意：
>
> - 点击立即下载后，默认下载当前 TiDB 的最新发布版本。如需下载其它版本，请在 TiDB 社区版页面底部查看其它版本下载信息。
> - 如需在 Kubernetes 上部署运维 TiDB，无需下载 TiDB-community-toolkit 软件包，请参考离线安装 TiDB Operator。
> - 如需使用PD Control 工具 `pd-ctl`，请下载 TiDB-community-server 软件包。

13.3.1.3 TiDB 工具包说明

在 TiDB 工具包中，你可以依据要使用的工具，选择安装对应的离线包。

工具	离线包名称

TiUP	`tiup-linux-{arch}.tar.gz`
	`tiup-{tiup-version}-linux-{arch}.tar.gz`
	`dm-{tiup-version}-linux-{arch}.tar.gz`
	`server-{version}-linux-{arch}.tar.gz`

工具	离线包名称
Dumpling	dumpling-{version}-linux-{arch}.tar.gz
TiDB Lightning	tidb-lightning-ctl tidb-lightning-{version}-linux-{arch}.tar.gz
TiDB DM (Data Migration)	dm-worker-{version}-linux-{arch}.tar.gz dm-master-{version}-linux-{arch}.tar.gz dmctl-{version}-linux-{arch}.tar.gz
TiCDC	cdc-{version}-linux-{arch}.tar.gz

工具	离线包名称
TiDB Binlog	pump-{version}-linux-{arch}.tar.gz drainer-{version}-linux-{arch}.tar.gz binlogctl reparo
Backup & Restore (BR)	br-{version}-linux-{arch}.tar.gz
sync-diff-inspector	sync_diff_inspector
TiSpark	tispark-{tispark-version}-any-any.tar.gz spark-{spark-version}-any-any.tar.gz
PD Recover	pd-recover-{version}-linux-{arch}.tar.gz

注意：

以上离线包名称中，`{version}` 取决于离线包中工具的版本号，`{arch}` 取决于离线包对应的架构（amd64 或 arm64）。

13.3.2 TiUniManager 下载

TiUniManager 是为 TiDB 打造的管控平台软件和数据库运维管理平台。使用下表中的链接下载 TiUniManager：

安装包	操作系统	架构	SHA256 校验和
https ↪ ://download ↪ .pingcap ↪ .org ↪ /em ↪ -enterprise ↪ -server ↪ -{version} ↪ }-linux ↪ -amd64 ↪ .tar ↪ .gz	Linux	amd64	https ↪ ://download ↪ .pingcap ↪ .org ↪ /em ↪ -enterprise ↪ -server ↪ -{version} ↪ }-linux ↪ -amd64 ↪ .tar ↪ .gz ↪ .sha256 ↪

注意：

- 下载链接中的 {version} 为 TiUniManager 的版本号。例如，v1.0.2 版本的下载链接为 https://download.pingcap.org/em-enterprise-server-v1.0.2-linux-amd64.tar.gz。
- TiUniManager 从 v1.0.2 起开放源代码，因此下载链接中 {version} 支持的最低版本为 v1.0.2。你不能将 {version} 替换为 v1.0.0 或 v1.0.1。

13.4 TiUP

13.4.1 TiUP 文档地图

13.4.1.1 使用文档

- TiUP 概览：对 TiUP 进行整体介绍，如何安装和基本的用法以及相关术语
- TiUP 术语：解释使用 TiUP 过程中可能用到的术语，了解 TiUP 的核心概念
- TiUP 组件管理：详细介绍 TiUP 所有命令，如何使用 TiUP 下载、更新、删除组件
- TiUP FAQ：TiUP 使用过程中的常见问题，包含 TiUP 第三方组件的 FAQ
- TiUP 故障排查：如果在使用 TiUP 过程中遇到问题，可以参考故障排查文档的解决方案
- TiUP 参考手册：TiUP 详细参考手册，包含各类命令、组件、镜像。

13.4.1.2 资源

- AskTUG TiUP 主题
- TiUP Issues：TiUP Github Issues 列表

13.4.2 TiUP 简介

在各种系统软件和应用软件的安装管理中，包管理器均有着广泛的应用，包管理工具的出现大大简化了软件的安装和升级维护工作。例如，几乎所有使用 RPM 的 Linux 都会使用 yum 来进行包管理，而 Anaconda 则可以非常方便地管理 Python 的环境和相关软件包。

在早期的 TiDB 生态中，没有专门的包管理工具，使用者只能通过相应的配置文件和文件夹命名来手动管理，如 Prometheus 等第三方监控报表工具甚至需要额外的特殊管理，这样大大提升了运维管理难度。

从 TiDB 4.0 版本开始，TiUP 作为新的工具，承担着包管理器的角色，管理着 TiDB 生态下众多的组件，如 TiDB、PD、TiKV 等。用户想要运行 TiDB 生态中任何组件时，只需要执行 TiUP 一行命令即可，相比以前，极大地降低了管理难度。

13.4.2.1 安装 TiUP

TiUP 安装过程十分简洁，无论是 Darwin 还是 Linux 操作系统，执行一行命令即可安装成功：

```
curl --proto '=https' --tlsv1.2 -sSf https://tiup-mirrors.pingcap.com/install.sh | sh
```

该命令将 TiUP 安装在 `$HOME/.tiup` 文件夹下，之后安装的组件以及组件运行产生的数据也会放在该文件夹下。同时，它还会自动将 `$HOME/.tiup/bin` 加入到 Shell Profile 文件的 PATH 环境变量中，这样你就可以直接使用 TiUP 了。

例如，你可以查看 TiUP 的版本：

```
tiup --version
```

> **注意：**
>
> 对于 v1.11.3 及以上版本的 TiUP，默认不会收集使用情况信息分享给 PingCAP。若要了解所收集的信息详情及如何关闭分享行为，请参见遥测。

13.4.2.2 TiUP 生态介绍

TiUP 的直接功能是作为 TiDB 生态中的包管理器，但这并不是它的最终使命。TiUP 的愿景是将 TiDB 生态中所有工具的使用门槛降到极致，这个仅仅靠包管理功能是做不到的，还需要引入一些额外的包来丰富这个系统，它们一起加入到 TiUP 生态中，让 TiDB 的世界变得更简单。

TiUP 系列文档的主要内容就是介绍 TiUP 及这些包的功能和使用方式。

在 TiUP 生态中，你可以通过在任何命令后加上 `--help` 的方式来获得帮助信息，比如通过以下命令获取 TiUP 本身的帮助信息:

```
tiup --help
```

```
TiUP is a command-line component management tool that can help to download and install
TiDB platform components to the local system. You can run a specific version of a component via
"tiup <component>[:version]". If no version number is specified, the latest version installed
locally will be used. If the specified component does not have any version installed locally,
the latest stable version will be downloaded from the repository.

Usage:
  tiup [flags] <command> [args...]
  tiup [flags] <component> [args...]

Available Commands:
  install     Install a specific version of a component
  list        List the available TiDB components or versions
  uninstall   Uninstall components or versions of a component
  update      Update tiup components to the latest version
  status      List the status of instantiated components
  clean       Clean the data of instantiated components
  mirror      Manage a repository mirror for TiUP components
  help        Help about any command or component

Components Manifest:
  use "tiup list" to fetch the latest components manifest

Flags:
      --binary <component>[:version]   Print binary path of a specific version of a component <
          ↪ component>[:version]
```

```
                                    and the latest version installed will be selected if no
                                        ↪ version specified
      --binpath string              Specify the binary path of component instance
  -h, --help                        help for tiup
  -T, --tag string                  Specify a tag for component instance
  -v, --version                     version for tiup

Component instances with the same "tag" will share a data directory ($TIUP_HOME/data/$tag):
  $ tiup --tag mycluster playground

Examples:
  $ tiup playground                 # Quick start
  $ tiup playground nightly         # Start a playground with the latest nightly version
  $ tiup install <component>[:version] # Install a component of specific version
  $ tiup update --all               # Update all installed components to the latest version
  $ tiup update --nightly           # Update all installed components to the nightly version
  $ tiup update --self              # Update the "tiup" to the latest version
  $ tiup list                       # Fetch the latest supported components list
  $ tiup status                     # Display all running/terminated instances
  $ tiup clean <name>               # Clean the data of running/terminated instance (Kill
      ↪ process if it's running)
  $ tiup clean --all                # Clean the data of all running/terminated instances

Use "tiup [command] --help" for more information about a command.
```

输出的帮助信息较长，你可以只关注两部分：

- 可用的命令
  - install：用于安装组件
  - list：查看可用组件列表
  - uninstall：卸载组件
  - update：更新组件版本
  - status：查看组件运行记录
  - clean：清除组件运行记录
  - mirror：从官方镜像克隆一个私有镜像
  - help：输出帮助信息
- 可用的组件
  - playground：在本机启动集群
  - client：连接本机的集群
  - cluster：部署用于生产环境的集群
  - bench：对数据库进行压力测试

注意：

- 可用的组件会持续增加，以 `tiup list` 输出结果为准。
- 组件的可用版本列表也会持续增加，以 `tiup list <component>` 输出结果为准。

命令和组件的区别在于，命令是 TiUP 自带的，用于进行包管理的操作。而组件是 TiUP 通过包管理操作安装的独立组件包。比如执行 `tiup list` 命令，TiUP 会直接运行自己内部的代码，而执行 `tiup playground` 命令则会先检查本地有没有叫做 playground 的组件包，若没有则先从镜像上下载过来，然后运行这个组件包。

13.4.3 TiUP 术语及核心概念

本文主要说明 TiUP 的重要术语和核心概念。

13.4.3.1 TiUP 组件

TiUP 程序只包含少数几个命令，用来下载、更新、卸载组件。TiUP 通过各种组件来扩展其功能。组件是一个可以运行的程序或脚本，通过 `tiup <component>` 运行组件时，TiUP 会添加一组环境变量，并为该程序创建好对应的数据目录，然后运行该程序。

通过运行 `tiup <component>` 命令，你可以运行支持的 TiUP 组件，其中运行的逻辑为：

1. 如果用户通过 `tiup <component>[:version]` 运行指定某个组件的特定版本：
    - 组件在本地未安装任何版本，则从镜像服务器下载最新稳定版本
    - 组件在本地安装有其他版本，但是没有用户指定的版本，则从镜像服务器下载用户指定版本
    - 如果本地已经安装指定版本，则设置环境变量来运行已经安装的版本
2. 如果用户通过 `tiup <component>` 运行某个组件，且未指定任何版本：
    - 组件在本地未安装任何版本，则从镜像服务器下载最新稳定版本
    - 如果本地已经安装部分版本，则设置环境变量来运行已经安装的版本中的最新版本

13.4.3.2 TiUP 镜像仓库

TiUP 的所有组件都从镜像仓库 (mirrors) 下载，镜像仓库包含各个组件的 TAR 包以及对应的元信息（版本、入口启动文件、校验和）。TiUP 默认使用 PingCAP 官方的镜像仓库。用户可以通过 `TIUP_MIRRORS` 环境变量自定义镜像仓库。

镜像仓库可以是本地文件目录或在线 HTTP 服务器：

1. `TIUP_MIRRORS=/path/to/local tiup list`
2. `TIUP_MIRRORS=https://private-mirrors.example.com tiup list`

13.4.4 使用 TiUP 命令管理组件

TiUP 主要通过以下一些命令来管理组件：

- list：查询组件列表，用于了解可以安装哪些组件，以及这些组件可选哪些版本
- install：安装某个组件的特定版本
- update：升级某个组件到最新的版本
- uninstall：卸载组件
- status：查看组件运行状态
- clean：清理组件实例
- help：打印帮助信息，后面跟其他 TiUP 命令则是打印该命令的使用方法

本文介绍常用的组件管理操作及相应命令。

13.4.4.1 查询组件列表

你可以使用 `tiup list` 命令来查询组件列表。该命令用法如下：

- `tiup list`：查看当前有哪些组件可以安装
- `tiup list ${component}`：查看某个组件有哪些版本可以安装

你也可以在命令中组合使用以下参数 (flag)：

- `--installed`：查看本地已经安装了哪些组件，或者已经安装了某个组件的哪些版本
- `--all`：显式隐藏的组件
- `--verbose`：显式所有列（安装的版本、支持的平台）

示例一：查看当前已经安装的所有组件

```
tiup list --installed
```

示例二：从服务器获取 TiKV 所有可安装版本组件列表

```
tiup list tikv
```

13.4.4.2 安装组件

你可以使用 `tiup install` 命令来安装组件。该命令的用法如下：

- `tiup install <component>`：安装指定组件的最新稳定版
- `tiup install <component>:[version]`：安装指定组件的指定版本

示例一：使用 TiUP 安装最新稳定版的 TiDB

```
tiup install tidb
```

示例二：使用 TiUP 安装 nightly 版本的 TiDB

```
tiup install tidb:nightly
```

示例三：使用 TiUP 安装 v7.0.0 版本的 TiKV

```
tiup install tikv:v7.0.0
```

13.4.4.3 升级组件

在官方组件提供了新版之后，你可以使用 `tiup update` 命令来升级组件。除了以下几个参数，该命令的用法基本和 `tiup install` 相同：

- `--all`：升级所有组件
- `--nightly`：升级至 nightly 版本
- `--self`：升级 TiUP 自己至最新版本
- `--force`：强制升级至最新版本

示例一：升级所有组件至最新版本

```
tiup update --all
```

示例二：升级所有组件至 nightly 版本

```
tiup update --all --nightly
```

示例三：升级 TiUP 至最新版本

```
tiup update --self
```

13.4.4.4 运行组件

安装完成之后，你可以使用 `tiup <component>` 命令来启动相应的组件：

```
tiup [flags] <component>[:version] [args...]

Flags:
  -T, --tag string                    为组件实例指定 tag
```

该命令需要提供一个组件的名字以及可选的版本，若不提供版本，则使用该组件已安装的最新稳定版。

在组件启动之前，TiUP 会先为它创建一个目录，然后将组件放到该目录中运行。组件会将所有数据生成在该目录中，目录的名字就是该组件运行时指定的 tag 名称。如果不指定 tag，则会随机生成一个 tag 名称，并且在实例终止时自动删除工作目录。

如果想要多次启动同一个组件并复用之前的工作目录，就可以在启动时用 `--tag` 指定相同的名字。指定 tag 后，在实例终止时就不会自动删除工作目录，方便下次启动时复用。

示例一：运行 v7.0.0 版本的 TiDB

```
tiup tidb:v7.0.0
```

示例二：指定 tag 运行 TiKV

```
tiup --tag=experiment tikv
```

13.4.4.4.1 查询组件运行状态

你可以使用 `tiup status` 命令来查看组件的运行状态：

```
tiup status
```

运行该命令会得到一个实例列表，每行一个实例。列表中包含这些列：

- Name：实例的 tag 名称
- Component：实例的组件名称
- PID：实例运行的进程 ID
- Status：实例状态，`RUNNING` 表示正在运行，`TERM` 表示已经终止
- Created Time：实例的启动时间
- Directory：实例的工作目录，可以通过 `--tag` 指定
- Binary：实例的可执行程序，可以通过 `--binpath` 指定
- Args：实例的运行参数

13.4.4.4.2 清理组件实例

你可以使用 `tiup clean` 命令来清理组件实例，并删除工作目录。如果在清理之前实例还在运行，会先 kill 相关进程。该命令用法如下：

```
tiup clean [tag] [flags]
```

支持以下参数：

- `--all`：清除所有的实例信息

其中 tag 表示要清理的实例 tag，如果使用了 `--all` 则不传递 tag。

示例一：清理 tag 名称为 `experiment` 的组件实例

```
tiup clean experiment
```

示例二：清理所有组件实例

```
tiup clean --all
```

13.4.4.4.3 卸载组件

TiUP 安装的组件会占用本地磁盘空间，如果不想保留过多老版本的组件，可以先查看当前安装了哪些版本的组件，然后再卸载某个组件。

你可以使用 `tiup uninstall` 命令来卸载某个组件的所有版本或者特定版本，也支持卸载所有组件。该命令用法如下：

```
tiup uninstall [component][:version] [flags]
```

支持的参数：

- `--all`：卸载所有的组件或版本
- `--self`：卸载 TiUP 自身

component 为要卸载的组件名称，version 为要卸载的版本，这两个都可以省略，省略任何一个都需要加上 `--all` 参数：

- 若省略版本，加 `--all` 表示卸载该组件所有版本
- 若版本和组件都省略，则加 `--all` 表示卸载所有组件及其所有版本

示例一：卸载 v7.0.0 版本的 TiDB

```
tiup uninstall tidb:v7.0.0
```

示例二：卸载所有版本的 TiKV

```
tiup uninstall tikv --all
```

示例三：卸载所有已经安装的组件

```
tiup uninstall --all
```

13.4.5 TiUP FAQ

13.4.5.1 TiUP 是否可以不使用官方镜像源？

TiUP 支持通过环境变量 `TIUP_MIRRORS` 指定镜像源，镜像源的地址可以是一个本地目录或 HTTP 服务器地址。如果用户的环境不能访问网络，可以建立自己的离线镜像源使用 TiUP。

如果在使用非官方镜像之后想要切回官方镜像可以采取以下任一措施：

- 将 `TIUP_MIRRORS` 变量设置成官方镜像的地址：`https://tiup-mirrors.pingcap.com`。
- 先确保 `TIUP_MIRRORS` 变量没有设置，再使用执行 `tiup mirror set https://tiup-mirrors.pingcap.com` 命令。

13.4.5.2 如何将自己编写的组件放入 TiUP 镜像仓库？

TiUP 暂时不支持外部开发的组件，但是 TiUP Team 已经制定了 TiUP 组件开发规范，同时正在开发 tiup-publish 组件，完成 tiup-publish 组件后，开发者可以通过 `tiup publish <comp> <version>` 将自己开发的组件发布到 TiUP 的官方镜像仓库。

13.4.5.3 tiup-playground 和 tiup-cluster 有什么区别？

TiUP Playground 组件主要定位是快速上手和搭建单机的开发环境，支持 Linux/macOS，要运行一个指定版本的 TiUP 集群更加简单。TiUP Cluster 组件主要是部署生产环境集群，通常是一个大规模的集群，还包含运维相关操作。

13.4.5.4 怎么样编写 tiup-cluster 组件的拓扑文件？

可以参考拓扑文件的样例，样例中包含了：

1. 两地三中心
2. 最小部署拓扑
3. 完整拓扑文件

可以根据自己的需求选择不同的模板，进行编辑。

13.4.5.5 同一个主机是否可以部署多个实例？

同一个主机可以使用 TiUP Cluster 部署多个实例，但是需要配置不同的端口和目录信息，否则可能导致目录以及端口冲突。

13.4.5.6 是否可以检测同一个集群内的端口和目录冲突？

同一个集群的端口和目录冲突会在部署和扩容的时候进行检测，如果有目录和端口冲突，本次部署或扩容会中断。

13.4.5.7 是否可以检测不同集群的端口和目录冲突？

如果不同集群是由同一个 TiUP 中控机部署的，会在部署和扩容时进行检测，如果属于不同的 TiUP 中控机，目前不支持检测。

13.4.5.8 集群部署期间，TiUP 收到报错 `ssh: handshake failed: read tcp 10.10.10.34:38980 -> 10.10.10.34:3600: read: connection reset by peer`

该报错可能是因为 TiUP 默认并发超过 ssh 默认最大连接数导致，可尝试加大 ssh 默认连接数，然后重启 sshd 服务解决：

```
vi /etc/ssh/sshd_config
MaxSessions 1000
MaxStartups 1000
```

13.4.6 TiUP 故障排查

本文介绍 TiUP 使用过程中一些常见的故障及排查方式，如果本文不包含你目前遇到的问题，可以通过以下方式求助：

1. Github Issues 新建一个 Issue。
2. 在 AskTUG 提交你的问题。

13.4.6.1 1. TiUP 命令故障排查

13.4.6.1.1 1.1 使用 `tiup list` 看不到最新的组件列表

TiUP 并不会每次都从镜像服务器更新最新的组件列表，可以通过 `tiup list` 来强制刷新组件列表。

13.4.6.1.2 1.2 使用 `tiup list <component>` 看不到一个组件的最新版本信息

同 1.1 一样，组件的版本信息只会在本地无缓存的情况下从镜像服务器获取，可以通过 `tiup list <component>` 刷新组件列表。

13.4.6.1.3 1.3 下载组件的过程中中断

如果下载组件的过程中网络中断，可能是由于网络不稳定导致的，可以尝试重新下载，如果多次不能成功下载，请反馈到 Github Issues，可能是由于 CDN 服务器导致的。

13.4.6.1.4 1.4 下载组件过程中出现 checksum 错误

由于 CDN 会有短暂的缓存时间，导致新的 checksum 文件和组件包不匹配，建议过 5 分钟后重试，如果依然不匹配，请反馈到 Github Issues。

13.4.6.2 2. TiUP Cluster 组件故障排查

13.4.6.2.1 2.1 部署过程中提示 `unable to authenticate, attempted methods [none publickey]`

由于部署时会向远程主机上传组件包，以及进行初始化，这个过程需要连接到远程主机，该错误是由于找不到连接到远程主机的 SSH 私钥导致的。请确认你是否通过 `tiup cluster deploy -i identity_file` 指定该私钥。

1. 如果没有指定 `-i` 参数，可能是由于 TiUP 没有自动找到私钥路径，建议通过 `-i` 显式指定私钥路径。
2. 如果指定了 `-i` 参数，可能是由于指定的私钥不能登录，可以通过手动执行 `ssh -i identity_file user@remote` 命令来验证。
3. 如果是通过密码登录远程主机，请确保指定了 `-p` 参数，同时输入了正确的登录密码。

13.4.6.2.2 2.2 使用 TiUP Cluster 升级中断

为了避免用户误用，TiUP Cluster 不支持指定部分节点升级，所以升级失败之后，需要重新进行升级操作，包括升级过程中的幂等操作。

升级操作会分为以下几步：

1. 首先备份所有节点的老版本组件
2. 分发新的组件到远程
3. 滚动重启所有组件

如果升级操作在滚动重启时中断，可以不用重复进行 `tiup cluster upgrade` 操作，而是通过 `tiup cluster restart -N <node1> -N <node2>` 来重启未完成重启的节点。如果同一组件的未重启节点数量比较多，也可以通过 `tiup cluster restart -R <component>` 来重启某一个类型的组件。

13.4.6.2.3 2.3 升级发现 node_exporter-9100.service/blackbox_exporter-9115.service 不存在

这种情况可能是由于之前的集群是由 TiDB Ansible 迁移过来的，且之前 TiDB Ansible 未部署 exporter 导致的。要解决这种情况，可以暂时通过手动从其他节点复制缺少的文件到新的节点。后续我们会在迁移过程中补全缺失的组件。

13.4.7 TiUP 命令参考手册

13.4.7.1 TiUP 命令概览

TiUP 在 TiDB 生态中承担包管理器的功能，管理着 TiDB 生态下众多的组件，如 TiDB、PD、TiKV 等。

13.4.7.1.1 语法

```
tiup [flags] <command> [args...]        # 执行命令
#### or
tiup [flags] <component> [args...]      # 运行组件
```

使用 --help 命令可以获取特定命令的信息，每个命令的摘要都显示了其参数及其用法。必须参数显示在尖括号中，可选参数显示在方括号中。

<command> 代表命令名字，支持的命令列表请参考下方命令清单，<component> 代表组件名，支持的组件列表请参考下方组件清单。

13.4.7.1.2 选项

–binary

打印指定组件的二进制文件路径：

- 执行 tiup --binary <component> 将打印已安装的 <component> 组件的最新稳定版路径，若 <component> 组件未安装，则报错
- 执行 tiup --binary <component>:<version> 将打印已经安装的 <component> 组件的 <version> 版本所在的路径，若该版本未安装，则报错
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 false。在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。

> 注意：
>
> 该选项只能用于 tiup [flags] <component> [args...] 格式的命令。

–binpath (string)

指定要执行的组件的路径：执行一个组件时，如果不想使用 TiUP 镜像中的二进制文件，可以使用该参数使用自定义路径的二进制文件替换之。

> **注意：**
>
> 该选项只能用于 `tiup [flags] <component> [args...]` 格式的命令。

-T, –tag (string)

对启动的组件指定一个 `tag`：有的组件在执行过程中需要使用磁盘存储，TiUP 会分配一个临时目录作为该组件本次执行的存储目录，如果希望分配固定目录，可以用 `-T/--tag` 来指定目录名字，这样多次执行使用同样的 `tag` 就能读写到同一批文件。

-v, –version

打印 TiUP 的版本

–help

打印帮助信息

13.4.7.1.3 命令清单

TiUP 包含众多的命令，这些命令又包含了许多子命令，具体命令及其子命令的说明请参考对应的链接：

- install：安装组件
- list：查看组件列表
- uninstall：卸载组件
- update：升级已安装的组件
- status：查看组件运行状态
- clean：清理组件数据目录
- mirror：镜像管理
- telemetry：遥测开关
- completion：TiUP 命令补全
- env：查看 TiUP 相关环境变量
- help：查看特定命令或组件的帮助文档

13.4.7.1.4 组件清单

- cluster：生产环境 TiDB 集群管理
- dm：生产环境 DM 集群管理

13.4.7.2 TiUP 命令

13.4.7.2.1 tiup clean

命令 `tiup clean` 用于清除组件运行过程中产生的数据。

语法

```
tiup clean [name] [flags]
```

[name] 取值为status 命令输出的 Name 字段。若省略 [name]，则必须配合 --all 使用。

选项

–all

- 清除所有运行记录。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 false。在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。

输出

```
Clean instance of `%s`, directory: %s
```

<< 返回上一页 - TiUP 命令清单

13.4.7.2.2 tiup completion

为了降低使用成本，TiUP 提供了 tiup completion 命令用于生成命令行自动补全的配置文件。目前支持 bash 和 zsh 两种 shell 的命令补全。

如果是 bash，需要提前安装好 bash-completion：

- 在 macOS 上的安装方式为：如果 bash 版本小于 4.1，执行 brew install bash-completion；如果 bash 版本大于等于 4.1，则执行 brew install bash-completion@2。
- 在 Linux 上的安装方式为：使用包管理器安装 bash-completion 包，例如执行 yum install bash-completion 或者 apt install bash-completion。

语法

```
tiup completion <shell>
```

<shell> 为 shell 类型，目前支持 bash 和 zsh。

使用方式

bash

将自动补全代码写入一个文件，并且在 .bash_profile 中对其执行 source 命令：

```
tiup completion bash > ~/.tiup.completion.bash

printf "
##### tiup shell completion
source '$HOME/.tiup.completion.bash'
" >> $HOME/.bash_profile

source $HOME/.bash_profile
```

zsh

```
tiup completion zsh > "${fpath[1]}/_tiup"
```

<< 返回上一页 - TiUP 命令清单

13.4.7.2.3 tiup env

TiUP 为用户提供了灵活的定制化接口，其中一部分是使用环境变量来实现的，命令 `tiup env` 用于查询 TiUP 支持用户自定义的环境变量以及它们此时的值。

语法

```
tiup env [name1...N]
```

`[name1...N]` 用于查看指定的环境变量，若不指定，则默认查看所有支持的环境变量。

选项

无

输出

- 若未指定 `[name1...N]`，则输出 “{key}” = “{value}” 列表
- 若指定了 `[name1...N]`，则按顺序输出 “{value}” 列表

以上输出中若 `value` 为空则代表未设置环境变量的值，此时 TiUP 会使用默认值。

<< 返回上一页 - TiUP 命令清单

13.4.7.2.4 tiup help

TiUP 命令行界面为用户提供了丰富的帮助信息，用户可以通过 `help` 命令或者 `--help` 参数查看。`tiup help <command>` 命令等价于 `tiup <command> --help`。

语法

```
tiup help [command]
```

`[command]` 用于指定要查看哪个命令的帮助信息，若不指定，则查看 TiUP 自身的帮助信息。

选项

无

输出

`[command]` 或 TiUP 的帮助信息。

<< 返回上一页 - TiUP 命令清单

13.4.7.2.5 tiup install

命令 `tiup install` 用于组件安装，它会从镜像仓库中下载指定版本的组件包，并在本地的 TiUP 数据目录中解压，以便后续使用。另外，当 TiUP 需要运行一个镜像仓库中不存在的组件时，会尝试先下载该组件，再自动运行，若仓库中不存在会报错。

语法

```
tiup install <component1>[:version] [component2...N] [flags]
```

`<component1>` 和 `<component2>` 代表组件名字，`[version]` 代表一个可选的版本号，若不加 `version`，则安装指定组件的最新稳定版本。`[component2...N]` 表示可同时指定多个组件或同一个组件的多个版本。

选项

无

输出

- 正常情况下输出组件的下载信息
- 若组件不存在则报错 `The component "%s" not found`
- 若版本不存在则报错 `version %s not supported by component %s`

<< 返回上一页 - TiUP 命令清单

13.4.7.2.6 tiup list

命令 `tiup list` 用于查询镜像中可用的组件列表。

语法

```
tiup list [component] [flags]
```

`[component]` 是可选的组件名称。若指定，则列出该组件的所有版本；若不指定，则列出所有组件列表。

选项

–all

- 显示所有组件。默认只显示非隐藏组件。
- 数据类型：`BOOLEAN`
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

–installed

- 只显示已经安装的组件或版本。
- 数据类型：`BOOLEAN`
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

–verbose

- 在组件列表中显示已安装的版本列表。默认组件列表不显示当前已安装的版本。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 false。在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。

输出

- 若未指定 [component]
  - 若指定 –verbose：输出 组件名 (Name)、已安装版本 (Installed)、组件管理员 (Owner)、组件描述 (Description) 构成的组件信息列表
  - 若不指定 –verbose：输出 组件名 (Name)、组件管理员 (Owner)、组件描述 (Description) 构成的组件信息列表
- 若指定 [component]
  - 若 [component] 存在：输出 版本 (Version)、是否已安装 (Installed)、发布时间 (Release)、支持的平台 (Platforms) 构成的版本信息列表
  - 若 [component] 不存在：报错 failed to fetch component: unknown component

<< 返回上一页 - TiUP 命令清单

13.4.7.2.7 tiup mirror

tiup mirror

在 TiUP 中，镜像是一个非常重要的概念，目前 TiUP 支持两种形式的镜像：

- 本地镜像：即 TiUP 客户端和镜像在同一台机器上，客户端通过文件系统访问镜像
- 远程镜像：即 TiUP 客户端和镜像不在同一台机器上，客户端通过网络访问镜像

命令 tiup mirror 用于管理镜像，提供了创建镜像，组件发布，密钥管理等多种功能。

语法

```
tiup mirror <command> [flags]
```

<command> 代表子命令，支持的子命令列表请参考下方命令清单。

选项

无

命令清单

- genkey: 生成私钥文件
- sign: 使用私钥文件对特定文件进行签名
- init: 创建一个空的镜像
- set: 设置当前镜像
- grant: 为当前镜像引入新的组件管理员
- publish: 向当前镜像推送新的组件

- modify: 修改当前镜像中的组件属性
- rotate: 更新当前镜像中的根证书
- clone: 从已有镜像克隆一个新的镜像
- merge: 合并镜像

<< 返回上一页 - TiUP 命令清单

tiup mirror clone

命令 `tiup mirror clone` 用于克隆一个已经存在的镜像或克隆部分组件生成一个新的镜像。新旧镜像的组件相同，但使用的签名密钥不同。

语法

```
tiup mirror clone <target-dir> [global version] [flags]
```

- `<target-dir>` 是本地存放克隆下来的镜像的路径，如果不存在则会自动创建。
- 若指定了 `[global version]` 参数，TiUP 会尝试克隆指定版本的所有组件。若某些组件没有指定的版本，则克隆其最新版本。

选项

-f, –full

- 是否克隆整个镜像。指定该选项后会从目标镜像完整克隆所有组件的所有版本，此时其他指定的选项将失效。
- 数据类型：`BOOLEAN`
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

-a, –arch

- 仅克隆能在指定平台上运行的组件。
- 数据类型：`STRINGS`
- 该参数接受以逗号分隔的多个平台名称，例如 `amd64,arm64`。如果未指定该选项，默认克隆 AMD64 和 ARM64 平台的组件，即 `amd64,arm64`。

-o, –os

- 仅克隆能在指定操作系统上运行的组件。
- 数据类型：`STRINGS`
- 该参数接受以逗号分隔的多个操作系统名称，例如 `linux,darwin`。如果未指定该选项，默认克隆 Linux 和 Darwin 系统的组件，即 `linux,darwin`。

–prefix

- 匹配版本时是否前缀匹配。默认情况下必须严格匹配指定的版本才会下载，指定该选项之后，仅前缀匹配指定的版本也会被下载。

- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 false。在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。

–{component}（strings，默认为空）

指定要克隆的 {component} 组件的版本列表。{component} 为组件名，可选的组件名可执行tiup list --all查看。

<< 返回上一页 - TiUP Mirror 命令清单

tiup mirror genkey

在 TiUP 镜像的定义中，有三类角色：

- 镜像管理员：拥有 root.json、 index.json、 snapshot.json 以及 timestamp.json 的修改权限
- 组件管理员：拥有相关组件的修改权限
- 普通用户：可以下载并使用组件

由于修改文件需要相关的管理员进行签名，因此管理员必须拥有自己的私钥。命令 tiup mirror genkey 就是用于生成私钥的。

> 警告：
>
> 请勿通过网络传输私钥。

语法

```
tiup mirror genkey [flags]
```

选项

-n, –name

- 密钥的名字，该名字决定最终生成的文件名。生成的私钥文件路径为：${TIUP_HOME}/keys/{name}.json，其中 TIUP_HOME 为 TiUP 的 Home 目录，默认路径为 $HOME/.tiup，name 为 -n/--name 指定的密钥名字。
- 数据类型：STRING
- 如果不指定该选项，密钥名默认为 private。

-p, –public

- 显示当前私钥对应的公钥，当前私钥名字由 -n/--name 选项指定。
- 当指定了 -p/--public 时，不会创建新的私钥。若 -n/--name 指定的私钥不存在，则报错。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 false。在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。

–save

- 将公钥信息储存为文件放置于当前目录，文件名称为 `{hash-prefix}-public.json`，其中 `hash-prefix` 为该密钥 ID 的前 16 位。
- 数据类型：`BOOLEAN`
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

- 若未指定 `-p/--public`：
  - 若指定的密钥已存在：`Key already exists, skipped`
  - 若指定的密钥不存在：`private key have been write to ${TIUP_HOME}/keys/{name}.json`
- 若指定 `-p/--public`：
  - 若指定的密钥不存在：`Error: open ${TIUP_HOME}/keys/{name}.json: no such file or directory`
  - 若指定的密钥存在：输出该密钥对应的公钥内容

<< 返回上一页 - TiUP Mirror 命令清单

tiup mirror grant

命令 `tiup mirror grant` 用来向当前镜像中引入一个组件管理员。

组件管理员可以使用其密钥发布新的组件，也可以对其之前发布的组件作出修改。添加组件管理员时，待添加的组件管理员需要先将其公钥发送给镜像管理员。

> **注意：**
>
> 该命令仅支持在当前镜像为本地镜像时使用。

语法

```
tiup mirror grant <id> [flags]
```

`<id>` 为该组件管理员的 ID，该 ID 需要在整个镜像中唯一，建议使用符合正则 `^[a-z\d](?:[a-z\d]|-(?=[a-z\d])){0,38}$` 的字符串。

选项

-k, –key（string，默认 ${TIUP_HOME}/keys/private.json）

指定引入的组件管理员的密钥。该密钥可以是公钥也可以是私钥。如果传入私钥，会被转换成对应的公钥储存在镜像中。

一个密钥只能被一个组件管理员使用。

-n, –name（string，默认 `<id>`）

指定组件管理员的名字，该名字会展示在组件列表的 `Owner` 字段上。若未指定 `-n/--name` 则使用 `<id>` 作为组件管理员名字。

输出

- 若执行成功：无输出
- 若管理员 ID 重复：`Error: owner %s exists`
- 若密钥已被其他管理员使用：`Error: key %s exists`

<< 返回上一页 - TiUP Mirror 命令清单

tiup mirror init

命令 `tiup mirror init` 用于初始化一个空的镜像。初始化的镜像不包含任何组件和组件管理员，仅生成以下文件：

```
+ <mirror-dir>                              # 镜像根目录
|-- root.json                               # 镜像根证书
|-- 1.index.json                            # 组件/用户索引
|-- snapshot.json                           # 镜像最新快照
|-- timestamp.json                          # 镜像最新时间戳
|--+ keys                                   # 镜像私钥（可移动到其他位置）
   |-- {hash1..hashN}-root.json             # 根证书私钥
   |-- {hash}-index.json                    # 索引私钥
   |-- {hash}-snapshot.json                 # 快照私钥
   |-- {hash}-timestamp.json                # 时间戳私钥
```

以上文件的具体作用及内容格式请参考镜像说明。

语法

```
tiup mirror init <path> [flags]
```

<path> 为本地目录路径，可以为相对路径。TiUP 会以此路径为镜像文件存放路径，在其中生成文件。若该目录已存在，则必须保证为空，若该目录不存在，则 TiUP 会自动创建。

选项

-k, –key-dir（string，默认 {path}/keys）

指定生成私钥文件的目录。若指定的文件目录不存在，则会自动创建。

输出

- 若成功：无输出
- 若 <path> 不为空：`Error: the target path '%s' is not an empty directory`
- 若 <path> 不是目录：`Error: fdopendir: not a directory`

<< 返回上一页 - TiUP Mirror 命令清单

tiup mirror merge

命令 `tiup mirror merge` 用于合并一个或多个镜像到当前镜像。

执行此命令需要满足几个条件：

- 目标镜像的所有组件的管理员 ID 必须在当前镜像中存在。

- 执行该命令用户的 ${TIUP_HOME}/keys 目录中有上述管理员 ID 在当前镜像中对应的所有私钥（可以使用命令tiup mirror set 将当前镜像切换成目前有权限修改的镜像）。

语法

```
tiup mirror merge <mirror-dir-1> [mirror-dir-N] [flags]
```

- <mirror-dir-1>：要合并到当前镜像的第一个镜像
- [mirror-dir-N]：要合并到当前镜像的第 N 个镜像

选项

无

输出

- 成功：无输出
- 当前镜像缺失目标镜像某个组件的管理员，或 ${TIUP_HOME}/keys 缺失该管理员的私钥：Error: missing owner keys for owner %s on component %s

<< 返回上一页 - TiUP Mirror 命令清单

tiup mirror modify

命令 tiup mirror modify 用于修改已发布的组件。只有合法的组件管理员才可以修改组件，且只能修改其自己发布的组件。组件发布方式参考publish 命令。

语法

```
tiup mirror modify <component>[:version] [flags]
```

各个参数解释如下：

- <component>：组件名称
- [version]：想要修改的版本，若不指定，则表示修改整个组件

选项

-k, –key（string，默认 ${TIUP_HOME}/keys/private.json）

组件管理员的私钥，客户端需要使用该私钥对组件信息 ({component}.json) 进行签名。

–yank

- 将指定组件或指定版本标记为不可用：
  - 标记组件不可用之后 tiup list 将看不到该组件，也无法安装该组件的新版本。
  - 标记版本不可用之后 tiup list <component> 将看不到该版本，也无法安装该版本。
- 数据类型：BOOLEAN

- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

–hide

- 将该组件设置为隐藏，隐藏之后该组件将不在 `tiup list` 的列表中显示，但是可通过执行 `tiup list --all` 查看。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

> **注意：**
>
> 该选项只能应用于组件上，无法应用于组件的版本上。

输出

- 若成功：无输出
- 若该组件管理员无权修改目标组件：
  - 若使用远程镜像：`Error: The server refused, make sure you have access to this component`
  - 若使用本地镜像：`Error: the signature is not correct`

<< 返回上一页 - TiUP Mirror 命令清单

tiup mirror publish

命令 `tiup mirror publish` 用于发布新组件，或已有组件的新版本。只有有权限的组件管理员才可以发布组件。引入组件管理员的方式可参考grant 命令。

语法

```
tiup mirror publish <comp-name> <version> <tarball> <entry> [flags]
```

各个参数解释如下：

- `<comp-name>`：组件名，如 `tidb`，建议使用符合正则 `^[a-z\d](?:[a-z\d]|-(?=[a-z\d])){0,38}$` 的字符串
- `<version>`：当前正在发布的版本，版本号需要符合 Semantic Versioning
- `<tarball>`：`.tar.gz` 包的本地路径，需要将组件的可执行文件及依赖放在该包中，由 TiUP 上传到镜像
- `<entry>`：组件的可执行文件在 `<tarball>` 中的位置

选项

-k, –key（string，默认 ${TIUP_HOME}/keys/private.json）

组件管理员的私钥，客户端需要使用该私钥对组件信息 (`{component}.json`) 进行签名。

–arch（string，默认 ${GOARCH}）

该 `<tarlball>` 中的二进制文件运行的平台，一个 `<tarball>` 只能选以下三个平台之一：

- amd64：表示在 amd64 架构的机器上运行
- arm64：表示在 arm64 架构的机器上运行
- any：表示可以在以上两种架构的机器上运行（比如脚本）

> 注意：
>
> 若 --arch 指定为 any，则 --os 也必须指定为 any。

–os（string，默认 ${GOOS}）

该 <tarball> 中的二进制文件运行的操作系统，一个 <tarball> 只能选以下三个操作系统之一：

- linux：表示在 Linux 操作系统上运行
- darwin：表示在 Darwin 操作系统上运行
- any：表示可以在以上两种操作系统上运行（比如脚本）

> 注意：
>
> 若 --os 指定为 any，则 --arch 也必须指定为 any。

–desc（string，默认为空）

该组件的描述信息。

–hide

- 是否为隐藏组件。若为隐藏组件，则不在 tiup list 的列表中显示，但在 tiup list --all 的列表中会显示。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 false。在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。

输出

- 若成功：无输出
- 若该组件管理员无权修改目标组件：
  - 若使用远程镜像：Error: The server refused, make sure you have access to this component
  - 若使用本地镜像：Error: the signature is not correct

<< 返回上一页 - TiUP Mirror 命令清单

tiup mirror rotate

TiUP 的镜像中有一个非常重要的文件：root.json，里面记录了整个系统需要使用的公钥，是 TiUP 信任链的基础，它的内容主要包含几个部分：

- N 个管理员的签名，对于官方镜像，N 为 5，默认初始化的镜像 N 为 3
- 用于验证以下文件的公钥：
  - root.json
  - index.json
  - snapshot.json
  - timestamp.json
- 过期时间，对于官方镜像，为 root.json 创建时间后延一年

关于镜像的详细介绍可以参考镜像说明。

在某些情况下，用户需要更新 root.json:

- 更换镜像的密钥
- 更新证书过期时间

更新 root.json 内容之后，必须由管理员对其进行重新签名，否则客户端会拒绝，更新流程如下：

1. 更新 root.json 的内容
2. N 个管理员对新的 root.json 进行签名
3. 更新 snapshot.json，记录新的 root.json 的 version
4. 对新的 snapshot.json 进行签名
5. 更新 timestamp.json，记录新的 snapshot.json 的 hash
6. 对新的 timestamp.json 进行签名

TiUP 使用命令 `tiup mirror rotate` 来自动化以上流程。

> 注意：
>
> - 经测试，小于 TiUP v1.5.0 的版本无法正确获得新的 root.json #983。
> - 使用此功能前请确保所有的 TiUP 客户端升级到了 v1.5.0 或以上版本。

语法

```
tiup mirror rotate [flags]
```

该命令会启动一个编辑器，修改其内容为目标值（比如将 `expires` 字段的值向后推移），然后需要将 `version` 字段加一并保存。保存之后会启动一个临时的 http 服务器，等待 N 个不同的镜像管理员签名。

镜像管理员签名的方式参考sign 命令。

选项

–addr（string，默认 0.0.0.0:8080）

临时服务器的监听地址，需要确保该地址可以被其他镜像管理员访问，这样管理员才能使用sign 命令签名。

输出

- 各个镜像管理员当前的签名状态

<< 返回上一页 - TiUP Mirror 命令清单

tiup mirror set

命令 `tiup mirror set` 用于切换当前镜像，支持本地文件系统和远程网络两种镜像。

官方镜像为 `https://tiup-mirrors.pingcap.com`。

语法

```
tiup mirror set <mirror-addr> [flags]
```

`<mirror-addr>` 为镜像地址，可以有两种形式：

- 网络地址：以 http 或者 https 开头，如 `http://172.16.5.5:8080`，`https://tiup-mirrors.pingcap.com` 等
- 本地文件路径：镜像目录的绝对路径，比如 `/path/to/local-tiup-mirror`

选项

-r, –root（string，默认 `{mirror-dir}/root.json`）

指定根证书。

每个镜像的根证书不相同，而根证书是镜像安全性最关键的一环，在使用网络镜像时，可能遭受中间人攻击，为了避免此类攻击，推荐手动将根网络镜像的根证书下载到本地：

```
wget <mirror-addr>/root.json -O /path/to/local/root.json
```

然后进行人工查验，认定无误之后，再通过手工指定根证书的方式切换镜像：

```
tiup mirror set <mirror-addr> -r /path/to/local/root.json
```

在这种操作方式下，如果中间人在 `wget` 之前攻击了镜像，用户可发现根证书不正确。如果在 `wget` 之后攻击了镜像，TiUP 会发现镜像和根证书不符。

输出

无

<< 返回上一页 - TiUP Mirror 命令清单

tiup mirror sign

命令 `tiup mirror sign` 用于对镜像中定义的元信息文件（*.json）进行签名，这些文件可能储存在本地文件系统，也可以放在远端使用 http 协议提供签名入口。

语法

```
tiup mirror sign <manifest-file> [flags]
```

`<manifest-file>` 为被签名的文件地址，可以有两种地址：

- 网络地址：http 或者 https 开头，如 `http://172.16.5.5:8080/rotate/root.json`

- 本地文件路径：相对路径或绝对路径均可

如果是网络地址，该地址必须提供以下功能：

- 支持以 `http get` 访问，此时应当返回被签名文件的完整内容（包含 signatures 字段）
- 支持以 `http post` 访问，客户端会在 `http get` 返回的内容的 signatures 字段中加上本次的签名 POST 到该地址

选项

-k, –key (string，默认 ${TIUP_HOME}/keys/private.json)

指定用于签名的私钥位置。

–timeout (int，默认 10)

通过网络签名时网络的访问超时时间，单位为秒。

> 注意：
>
> 只有当 `<manifest-file>` 为网络地址时该选项有效。

输出

- 成功：无输出
- 文件已被指定的 key 签名过：`Error: this manifest file has already been signed by specified key`
- 文件不是合法的 manifest：`Error: unmarshal manifest: %s`

<< 返回上一页 - TiUP Mirror 命令清单

13.4.7.2.8 tiup status

使用命令 `tiup status` 可查看组件的运行信息：通过 `tiup [flags] <component> [args...]` 运行组件之后，可以通过该命令查看组件的运行信息。

> 注意：
>
> 只能查询到以下两种组件的信息：
>
> - 尚在运行的组件
> - 通过 `tiup -T/--tag` 指定 tag 运行的组件

语法

```
tiup status [flags]
```

选项

无

输出

由以下字段构成的表格：

- Name: 通过 `-T/--tag` 指定的 Tag 名字，若未指定，则为随机字符串
- Component: 运行的组件
- PID: 对应的进程 ID
- Status: 组件运行状态
- Created Time: 启动时间
- Directory: 数据目录
- Binary: 二进制文件路径
- Args: 启动参数

组件运行状态 (Status)

组件可能处于如下任一运行状态：

- 在线 (Up)：组件正常运行。
- 离线 (Down) 或无法访问 (Unreachable)：组件未启动或对应主机存在网络问题。
- 已缩容下线 (Tombstone)：组件上的数据已被完整迁出并缩容完毕。仅 TiKV 或 TiFlash 组件存在该状态。
- 下线中 (Pending Offline)：组件上的数据正在被迁出并缩容。仅 TiKV 或 TiFlash 组件存在该状态。
- 未知 (Unknown)：未知的组件运行状态。

> 注意：
>
> TiUP 显示的 `Pending Offline`、PD API 返回的 `Offline` 以及 TiDB Dashboard 显示的 `Leaving` 这三个状态的含义相同。

组件运行状态来自于 PD 的调度信息。更详细的描述请参考TiDB 数据库的调度 – 信息收集。

<< 返回上一页 - TiUP 命令清单

13.4.7.2.9 tiup telemetry

在 v1.11.3 及以上的 TiUP 版本中，遥测功能默认关闭，即默认不收集使用情况信息分享给 PingCAP。在 v1.11.3 以下的 TiUP 版本中，遥测功能默认开启，即默认收集使用情况信息分享给 PingCAP 用于改善产品。

当 TiUP 遥测功能开启时，执行 TiUP 命令时会将使用情况信息分享给 PingCAP，包括（但不限于）：

- 随机生成的遥测标示符

- TiUP 命令的执行情况，如命令执行是否成功、命令执行耗时等
- 使用 TiUP 进行部署的情况，如部署的目标机器硬件信息、组件版本号、修改过的部署配置名称等

以下信息不会被分享：

- 集群的准确名字
- 集群的拓扑结构
- 集群的配置文件

TiUP 使用命令 `tiup telemetry` 来控制遥测。

语法

```
tiup telemetry <command>
```

`<command>` 代表子命令，支持的子命令列表请参考下方命令一节。

命令

status

命令 `tiup telemetry status` 查看当前的遥测设置，输出以下信息：

- `status`: 当前是否开启遥测 `(enable|disable)`
- `uuid`: 随机生成的遥测标示符

reset

命令 `tiup telemetry reset` 重置当前的遥测标示符，以一个新的随机标识符代替之。

enable

命令 `tiup telemetry enable` 启用遥测。

disable

命令 `tiup telemetry disable` 停用遥测。

<< 返回上一页 - TiUP 命令清单

13.4.7.2.10 tiup uninstall

命令 `tiup uninstall` 用于卸载已安装的组件。

语法

```
tiup uninstall <component1>:<version> [component2...N] [flags]
```

- `<component1>` 表示要卸载的组件名字
- `<version>` 表示要卸载的版本，如果省略，则表示卸载该组件的全部已安装版本，因为安全原因，省略 `<version>` 时必须加上选项 `--all` 明确表示需要卸载该组件的所有版本
- `[component2...N]` 表示可指定卸载多个组件或版本

选项

–all

- 卸载指定组件的全部已安装版本，省略 <version> 时使用。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 false。在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。

–self

- 卸载 TiUP 自身：删除所有从镜像上下载过来的数据，但会保留 TiUP 及其组件产生的数据，数据存放在 TIUP_HOME 环境变量指定的目录中，若未设置过 TIUP_HOME，则默认值为 ~/.tiup/。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 false。在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。

输出

- 正常退出：Uninstalled component "%s" successfully!
- 若未指定 <version> 也未指定 --all：报错 Use "tiup uninstall tidbx --all" if you want to remove all versions.

<< 返回上一页 - TiUP 命令清单

13.4.7.2.11 tiup update

命令 tiup update 用于升级已安装的组件或者自身。

语法

```
tiup update [component1][:version] [component2..N] [flags]
```

- [component1] 表示要升级的组件名字
- [version] 表示要升级的版本，如果省略，则表示升级到该组件的最新稳定版本
- [component2...N] 表示可指定升级多个组件或版本。如果一个组件也不指定：即 [component1][:version] [component2..N] 为空，则需要配合使用 --all 选项或 --self 选项。

升级操作不会删除旧的版本，仍然可以在执行时指定旧版本使用。

选项

–all

- 若未指定任何组件，则必须指定该选项。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 false。在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。

–force

- 若指定的组件版本已经安装，则默认跳过升级操作，指定该参数可强制升级已安装版本。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

–nightly

- 将指定组件升级到 nightly 版本。使用该参数的命令等价于 `tiup update <component>:nightly`。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

–self

- 升级 TiUP 自身。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

- 升级成功：`Updated successfully!`
- 目标版本不存在：`Error: version %s not supported by component %s`

<< 返回上一页 - TiUP 命令清单

13.4.7.3 TiUP Cluster 命令

13.4.7.3.1 TiUP Cluster

TiUP Cluster 是 TiUP 提供的使用 Golang 编写的集群管理组件，通过 TiUP Cluster 组件就可以进行日常的运维工作，包括部署、启动、关闭、销毁、弹性扩缩容、升级 TiDB 集群、管理 TiDB 集群参数。

语法

```
tiup cluster [command] [flags]
```

`[command]` 代表命令名字，支持的命令列表请参考下方命令清单。

选项

–ssh (string，默认 builtin)

指定 SSH 客户端连接远端（部署 TiDB 服务的机器）执行命令，支持以下值：

- builtin：使用 tiup-cluster 内置的 easyssh 客户端
- system：使用当前操作系统默认的 SSH 客户端
- none：不使用 ssh 客户端，这种方式只支持部署到当前机器

–ssh-timeout（uint，默认 5）

设置 SSH 连接超时时间，单位为秒。

–wait-timeout（uint，默认 120）

运维过程中涉及到很多操作：指定 systemctl 启动/停止服务，等待端口上线/下线等，每个操作可能会消耗数秒。`--wait-timeout` 用于设置每个步骤的最长等待时间（单位为秒），超时后报错退出。

-y, –yes

- 跳过所有风险操作的二次确认，除非是使用脚本调用 TiUP，否则不推荐使用。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

-v, –version

- 输出 TiUP Cluster 当前版本信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

-h, –help

- 输出相关命令的帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

命令清单

- import：导入 ansible 部署的集群
- template：输出拓扑模版
- check：部署前后的集群检查
- deploy：根据指定拓扑部署集群
- list：查询已部署的集群列表
- display：展示指定集群状态
- start：启动指定集群
- stop：停止指定集群
- restart：重启指定集群
- scale-in：缩容指定集群
- scale-out：扩容指定集群
- upgrade：升级指定集群
- prune：销毁指定集群中状态为 Tombstone 的实例
- edit-config：修改指定集群配置
- reload：重载指定集群配置
- patch：替换已部署集群的某个服务
- rename：重命名集群
- clean：删除指定集群数据

- destroy：销毁指定集群
- audit：查询集群操作审计日志
- replay：重试指定命令
- enable：开启指定集群或服务开机自启动
- disable：关闭指定集群或服务开机自启动
- meta backup：备份指定集群运维操作所需的 TiUP meta 文件
- meta restore：恢复指定集群的 TiUP meta 文件
- help：输出帮助信息

<< 返回上一页 - TiUP 组件清单

13.4.7.3.2 tiup cluster audit

命令 `tiup cluster audit` 可以用于查看历史上对所有集群执行了什么命令，以及每个命令的执行日志。

语法

```
tiup cluster audit [audit-id] [flags]
```

- 若不填写 `[audit-id]` 则按时间倒序输出操作记录的表格，第一列为 audit-id
- 若填写 `[audit-id]` 则查看指定的 audit-id 的执行日志

选项

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

- 若指定 `[audit-id]`，则输出对应的执行日志
- 若不指定 `[audit-id]` 则输出含有以下字段的表格：
  - ID：该条记录对应的 audit-id
  - Time：该条记录对应的命令执行时间
  - Command：该条记录对应的命令

<< 返回上一页 - TiUP Cluster 命令清单

13.4.7.3.3 tiup cluster audit cleanup

命令 `tiup cluster audit cleanup` 可以用于清理 `tiup cluster` 产生的执行日志。

语法

```
tiup cluster audit cleanup [flags]
```

选项

–retain-days

- 执行日志保留天数
- 数据类型：INT
- 默认值：60，单位为“天”
- 默认保留 60 天的执行日志，即删除 60 天之前的执行日志。

-h, –help

- 输出帮助信息
- 数据类型：BOOLEAN
- 默认值：false
- 在命令中添加该选项，并传入 true 或不传值，均可开启此功能。

输出

```
clean audit log successfully
```

<< 返回上一页 - TiUP Cluster 命令清单

13.4.7.3.4 tiup cluster check

对于严肃的生产环境，在正式上线之前需要进行一系列检查，来确保集群拥有最好的表现。为了简化人工检查的步骤，TiUP Cluster 提供了 check 子命令，用于检查指定集群的机器硬件和软件环境是否满足正常运行条件。

检查项列表

操作系统版本

检查部署机操作系统发行版和版本：目前仅支持部署在 CentOS 7 的操作系统上，之后随兼容性改进可能支持更多系统版本。

CPU EPOLLEXCLUSIVE

检查部署机 CPU 是否支持 EPOLLEXCLUSIVE。

numactl

检查部署机是否安装 numactl，若用户配置绑核，则必须安装 numactl。

系统时间

检查部署机系统时间是否同步：将部署机系统时间与中控机对比，偏差超出某一阈值（500ms）后报错。

系统时区

检查部署机系统时区是否同步：将部署机系统的时区配置进行对比，如果时区不一致则报错。

时间同步服务

检查部署机是否配置了时间同步服务：即 ntpd 是否在运行

Swap 分区

检查部署机是否启用 Swap 分区：建议禁用 Swap 分区

内核参数

检查各项内核参数的值：

- net.ipv4.tcp_tw_recycle: 0
- net.ipv4.tcp_syncookies: 0
- net.core.somaxconn: 32768
- vm.swappiness: 0
- vm.overcommit_memory: 0 或 1
- fs.file-max: 1000000

THP（透明大页）

检查部署机是否启用透明大页：建议禁用透明大页。

系统限制

检查 /etc/security/limits.conf 中各项 limit 值：

```
<deploy-user>    soft    nofile    1000000
<deploy-user>    hard    nofile    1000000
<deploy-user>    soft    stack     10240
```

其中 <deploy-user> 为部署、运行 TiDB 集群的用户，最后一列的数值为要求达到的最小值。

SELinux

检查 SELinux 是否启用：建议用户禁用 SELinux。

防火墙

检查 FirewallD 服务是否启用：建议用户禁用 FirewallD 或为 TiDB 集群各服务添加允许规则。

irqbalance

检查 irqbalance 服务是否启用：建议用户启用 irqbalance 服务。

磁盘挂载参数

检查 ext4 分区的挂载参数：确保挂载参数包含 nodelalloc,noatime 选项。

端口占用

检查部署机上是否已有进程占用了端口：检查拓扑中定义的端口（包括自动补全的默认端口）在部署机上是否已被占用。

> 注意：
>
> 端口占用检查假设集群尚未启动，如果检查的是已经部署并启动的集群，那么端口占用检查一定会失败，因为端口确实被占用了。

CPU 核心数

检查部署机 CPU 信息：建议生产集群 CPU 逻辑核心数 >= 16

> 注意：
> 默认不检查 CPU 核心数，需要通过选项 `--enable-cpu` 启用。

内存大小

检查部署机的内存大小：建议生产集群总内存容量 >= 32Gb。

> 注意：
> 默认不检查内存大小，需要通过选项 `--enable-mem` 启用。

fio 磁盘性能测试

使用 fio 测试 data_dir 所在磁盘的性能，包括三个测试项目：

- fio_randread_write_latency
- fio_randread_write
- fio_randread

> 注意：
> 默认不进行 fio 磁盘性能测试，需要通过选项 `--enable-disk` 启用。

语法

```
tiup cluster check <topology.yml | cluster-name> [flags]
```

- 若集群尚未部署，需要传递将用于部署集群的topology.yml 文件，tiup-cluster 会根据该文件的内容连接到对应机器去检查。
- 若集群已经部署，则可以使用集群的名字 `<cluster-name>` 作为检查对象。
- 如果需要检查已部署集群的扩容拓扑文件，可以将 `<scale-out.yml>` 和 `<cluster-name>` 作为检查对象。

> 注意：
> 若传递的是集群名字，则需要配合 `--cluster` 选项使用。

选项

–apply

- 尝试自动修复失败的检查项，目前仅会尝试修复以下项目：
  - SELinux
  - 防火墙
  - irqbalance
  - 内核参数
  - 系统 Limits
  - THP（透明大页）
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

> 注意：
>
> `tiup cluster check` 也支持修复已部署集群的扩容拓扑文件，命令格式：
>
> ```
> tiup cluster check <cluster-name> scale-out.yml --cluster --apply --user root [-p]
>     ↪ [-i /home/root/.ssh/gcp_rsa]
> ```

–cluster

- 对已部署的集群进行检查。
- 数据类型：BOOLEAN
- 默认值：`false`
- 在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。
- 命令格式：

```
tiup cluster check <topology.yml | cluster-name> --cluster [flags]
```

> 注意：
>
> - 若选择的格式为 `tiup cluster check <cluster-name>`,则必须加上该选项：`tiup cluster check <cluster-name> --cluster`。
> - `tiup cluster check` 也支持检查已部署集群的扩容拓扑文件，命令格式：
>
> ```
> shell  tiup cluster check <cluster-name> scale-out.yml --cluster --user root [-p]
> ↪ [-i /home/root/.ssh/gcp_rsa]
> ```

-N, –node

- 指定要检查的节点。该选项的值为以逗号分割的节点 ID 列表，节点 ID 为`tiup-component-cluster-display` 命令返回的集群状态表格的第一列。
- 数据类型：STRINGS
- 如果不指定该选项，默认检查所有节点，即 []。

> 注意：
>
> 若同时指定了 -R, --role，那么将检查它们的交集中的服务。

-R, –role

- 指定要检查的角色。该选项的值为以逗号分割的节点角色列表，角色为`tiup-component-cluster-display` 命令返回的集群状态表格的第二列。
- 数据类型：STRINGS
- 如果不指定该选项，默认检查所有角色。

> 注意：
>
> 若同时指定了 -N, --node，那么将检查它们的交集中的服务。

–enable-cpu

- 默认情况下 tiup-cluster 不检查 CPU 核心数，该选项用于启用 CPU 核心数检查。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 false。在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。

–enable-disk

- 默认情况下 tiup-cluster 不进行 fio 磁盘性能测试，该选项用于启用 fio 磁盘性能测试。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 false。在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。

–enable-mem

默认情况下 tiup-cluster 不检查内存大小，该选项用于启用内存大小检查。

-u, –user（string，默认为当前执行命令的用户）

指定连接目标机器的用户名，该用户在目标机器上需要有免密 sudo root 的权限。

> **注意：**
>
> 仅当 `--cluster` 选项为 false 时该选项有效，否则该值固定为部署集群时拓扑文件中指定的用户名。

-i, –identity_file（string，默认 ~/.ssh/id_rsa）

指定连接目标机器的密钥文件。

> **注意：**
>
> 仅当 `--cluster` 选项为 false 时该选项有效，否则该值固定为 `${TIUP_HOME}/storage/cluster/clusters/<cluster-name>/ssh/id_rsa`

-p, –password

- 在连接目标机器时使用密码登录：
  - 对于指定了 `--cluster` 的集群，密码为部署集群时拓扑文件中指定的用户的密码
  - 对于未指定 `--cluster` 的集群，密码为 `-u/--user` 参数指定的用户的密码
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

输出含有以下字段的表格：

- Node：目标节点
- Check：检查项
- Result：检查结果（Pass/Warn/Fail）
- Message：结果描述

<< 返回上一页 - TiUP Cluster 命令清单

13.4.7.3.5 tiup cluster clean

在测试环境中，有时候需要将集群重置回刚部署的状态，即删除所有数据，命令 `tiup cluster clean` 可以很方便的做到这一点：它会停止集群，然后删除集群上的数据。手工重启集群之后，就能得到一个全新的集群了。

> **警告：**
>
> 该命令一定会先停止集群（即使选择只清理日志也是），生产环境请勿使用。

语法

```
tiup cluster clean <cluster-name> [flags]
```

`<cluster-name>` 为要清理的集群。

选项

–all

- 同时清理数据和日志，等价于同时指定 `--data` 和 `--log`，若不指定该选项，则必须至少指定以下选项之一：
  - –data：清理数据
  - –log：清理日志
- 数据类型：`BOOLEAN`
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

–data

该选项开启数据清理，若不指定该选项，也不指定 `--all`，则不清理数据。

–log

- 该选项开启日志清理，若不指定该选项，也不指定 `--all`，则不清理日志。
- 数据类型：`BOOLEAN`
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

–ignore-node（StringArray，默认为空）

指定不需要清理的节点，如需指定多个，重复使用多次该选项：`--ignore-node <node-A> --ignore-node <node-B>`。

–ignore-role（StringArray，默认为空）

指定不需要清理的角色，如需指定多个，重复使用多次该选项：`--ignore-role <role-A> --ignore-role <role-B>`。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

tiup-cluster 的执行日志。

<< 返回上一页 - TiUP Cluster 命令清单

13.4.7.3.6 tiup cluster deploy

命令 `tiup cluster deploy` 用于部署一个全新的集群。

语法

```
tiup cluster deploy <cluster-name> <version> <topology.yaml> [flags]
```

- `<cluster-name>` 表示新集群的名字，不能和现有集群同名
- `<version>` 为要部署的 TiDB 集群版本号，如 `v7.0.0`
- `<topology.yaml>` 为事先编写好的拓扑文件

选项

-u, –user（string，默认为当前执行命令的用户）

指定连接目标机器的用户名，该用户在目标机器上需要有免密 sudo root 的权限。

-i, –identity_file（string，默认 ~/.ssh/id_rsa）

指定连接目标机器的密钥文件。

-p, –password

- 在连接目标机器时使用密码登录，不可和 `-i/--identity_file` 同时使用。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

–ignore-config-check

- 在组件二进制文件部署之后，TiUP 会对 TiDB，TiKV 和 PD 组件执行配置检查，检查方式为 `<binary> --config-check <config-file>`，其中 `<binary>` 为部署的二进制文件的路径，`<config-file>` 为根据用户配置生成的配置文件。如果想要跳过该项检查，可以使用该选项。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

–no-labels

- 当两个或多个 TiKV 部署到同一台机器时，会存在一个风险：由于 PD 无法感知集群的拓扑结构，可能将一个 Region 的多个副本调度到一台物理机上的不同 TiKV，这样这台物理机就成为了单点。为了避免这种情况，用户可以通过 label 来告诉 PD 不要将相同的 Region 调度到同一台机器上（配置方式参考通过拓扑 label 进行副本调度）。
- 但是对于测试环境，可能并不在意是否将一个 Region 的副本调度到了同一台机器上，这个时候可以使用 `--no-labels` 来绕过检查。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

–skip-create-user

- 在部署集群时，tiup-cluster 会先检查拓扑文件中指定的用户名是否存在，如果不存在就会创建一个。指定 `--skip-create-user` 选项后不再检查用户是否存在，直接跳过创建步骤。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

部署日志。

<< 返回上一页 - TiUP Cluster 命令清单

13.4.7.3.7 tiup cluster destroy

当业务下线之后，如果想将集群占有的机器释放出来让给其他业务使用，需要清理掉集群上的数据以及部署的二进制文件。`tiup cluster destroy` 命令会执行以下操作销毁集群：

- 停止集群
- 对于每个服务，删除其日志目录，部署目录，数据目录
- 如果各个服务的数据目录/部署目录的父目录是由 tiup-cluster 创建的，也一并删除

语法

```
tiup cluster destroy <cluster-name> [flags]
```

`<cluster-name>` 为要销毁的集群名字。

选项

–force

- 在某些情况下，有可能集群中的某些节点已经宕机，导致无法通过 SSH 连接到节点进行操作，这个时候可以通过 `--force` 选项忽略这些错误。

- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

–retain-node-data（StringArray，默认为空）

指定需要保留数据的节点，如需指定多个，重复使用多次该选项：`--retain-node-data <node-A> --retain-node-data <node-B>`。

–retain-role-data（StringArray，默认为空）

指定需要保留数据的角色，如需指定多个，重复使用多次该选项：`--retain-role-data <role-A> --retain-role-data <role-B>`。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

tiup-cluster 的执行日志。

<< 返回上一页 - TiUP Cluster 命令清单

13.4.7.3.8 tiup cluster disable

命令 `tiup cluster disable` 用于关闭集群服务所在机器重启之后的自启动，该命令会到指定的节点上去执行 `systemctl disable <service>` 来关闭服务的自启动。

语法

```
tiup cluster disable <cluster-name> [flags]
```

`<cluster-name>` 为要关闭自启的集群。

选项

-N, –node

- 指定要关闭自启的节点，该选项的值为以逗号分割的节点 ID 列表，节点 ID 为`tiup-component-cluster-display` 命令返回的集群状态表格的第一列。
- 数据类型：STRINGS
- 如果不指定该选项，默认关闭所有节点的自启。

> **注意：**
>
> 若同时指定了 `-R, --role`，那么将关闭它们的交集中的服务自启。

-R, –role

- 指定要关闭自启的角色，该选项的值为以逗号分割的节点角色列表，角色为tiup-component-cluster-display 命令返回的集群状态表格的第二列。
- 数据类型：STRINGS
- 如果不指定该选项，默认关闭所有角色的自启。

> 注意：
>
> 若同时指定了 -N, --node，那么将关闭它们的交集中的服务自启。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 false。在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。

输出

tiup-cluster 的执行日志。

<< 返回上一页 - TiUP Cluster 命令清单

13.4.7.3.9 tiup cluster display

如果想查看集群中每个组件的运行状态，逐一登录到各个机器上查看显然很低效。因此，tiup-cluster 提供了 tiup cluster display 命令来高效完成这件工作。

语法

```
tiup cluster display <cluster-name> [flags]
```

<cluster-name> 为要操作的集群名字，如果忘记集群名字可通过集群列表查看。

选项

–dashboard

- 默认情况会展示整个集群的所有节点信息，加上该选项后仅展示 dashboard 的信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 false。在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。

-N, –node（strings，默认为 []，表示所有节点）

指定要查询的节点，不指定则表示所有节点。该选项的值为以逗号分割的节点 ID 列表，如果不确定要查询节点的 ID，不指定此选项，输出会显示所有节点的 ID 和状态信息。

> 注意：
>
> 若同时指定了 `-R, --role`，那么将查询它们的交集中的服务状态。

-R, –role（ strings，默认为 []，表示所有角色 ）

指定要查询的角色，不指定则表示所有角色。该选项的值为以逗号分割的节点角色列表，如果不确定要查询节点的角色，不指定此选项，输出会显示所有节点的角色和状态信息。

> 注意：
>
> 若同时指定了 `-N, --node`，那么将查询它们的交集中的服务状态。

–process

- 加上该选项后会增加展示节点的 CPU 和内存的使用信息，默认情况下不展示。
- 数据类型：BOOLEAN
- 默认值：false
- 在命令中添加该选项，并传入 true 或不传值，均可开启此功能。

–uptime

- 加上该选项后会增加展示节点的 uptime 信息，默认情况下不展示。
- 数据类型：BOOLEAN
- 默认值：false
- 在命令中添加该选项，并传入 true 或不传值，均可开启此功能。

–status-timeout

- 获取节点状态信息的超时时间。
- 数据类型：INT
- 默认值：10，单位为 s。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 false。在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。

输出

- 集群名称
- 集群版本
- SSH 客户端类型
- Dashboard 地址
- 含有以下字段的表格：
  - ID：节点 ID，由 IP:PORT 构成
  - Role：该节点部署的服务角色（如 TiDB、TiKV 等）
  - Host：该节点对应的机器 IP
  - Ports：服务占用的端口号
  - OS/Arch：该节点的操作系统和机器架构
  - Status：该节点服务当前的状态
  - Data Dir：服务的数据目录，- 表示没有数据目录
  - Deploy Dir：服务的部署目录

节点服务的状态 (Status)

节点服务可能处于如下任一状态：

- 在线 (Up)：节点服务正常运行。
- 离线 (Down) 或无法访问 (Unreachable)：节点服务未启动或对应主机存在网络问题。
- 已缩容下线 (Tombstone)：节点服务上的数据已被完整迁出并缩容完毕。仅 TiKV 或 TiFlash 存在该状态。
- 下线中 (Pending Offline)：节点服务上的数据正在被迁出并缩容。仅 TiKV 或 TiFlash 存在该状态。
- 未知 (Unknown)：未知的节点服务运行状态。

> 注意：
>
> TiUP 显示的 `Pending Offline`、PD API 返回的 `Offline` 以及 TiDB Dashboard 显示的 `Leaving` 这三个状态的含义相同。

节点服务状态来自于 PD 的调度信息。更详细的描述请参考TiDB 数据库的调度 - 信息收集。

<< 返回上一页 - TiUP Cluster 命令清单

13.4.7.3.10 tiup cluster edit-config

在部署集群之后，如果需要再调整集群服务的配置，则可以使用命令 `tiup cluster edit-config`，它会启动一个编辑器（默认为 $EDITOR 环境变量指定的值，当 EDITOR 环境变量不存在时，使用 vi 打开）允许用户修改指定集群的拓扑文件。

> 注意：
>
> - 修改配置时不能增删机器，增删机器属于集群扩容和集群缩容的功能。
> - 执行完该命令后配置只是在中控机上修改了，要应用配置需要执行 `tiup cluster reload` 命令来重新加载。

语法

```
tiup cluster edit-config <cluster-name> [flags]
```

<cluster-name> 代表要操作的集群名。

选项

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 false。在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。

输出

- 正常情况无输出
- 若修改了不能修改的字段，则保存文件时报错并提示用户重新编辑，不能修改的字段参考拓扑文件中的相关描述

<< 返回上一页 - TiUP Cluster 命令清单

13.4.7.3.11 tiup cluster enable

命令 tiup cluster enable 用于设置集群服务在机器重启后的自启动，该命令会到指定的节点上去执行 systemctl enable <service> 来开启服务的自启。

> 注意：
>
> 集群全部关机重启时，服务启动的顺序由节点的操作系统启动顺序决定。重启顺序不正确时，在某些情况下，重启后的集群仍然不能提供服务（比如先启动了 TiKV 但未启动 PD，systemd 重启 TiKV 多次仍未找到 PD 则会放弃）。

语法

```
tiup cluster enable <cluster-name> [flags]
```

<cluster-name> 为要启用自启的集群。

选项

-N, –node

- 指定要开启自启的节点，该选项的值为以逗号分割的节点 ID 列表，节点 ID 为tiup-component-cluster-display 命令返回的集群状态表格的第一列。
- 数据类型：STRINGS

- 如果不指定该选项，默认开启所有节点的自启。

> 注意：
>
> 若同时指定了 `-R, --role`，那么将开启它们的交集中的服务自启。

-R, –role

- 指定要开启自启的角色，该选项的值为以逗号分割的节点角色列表，角色为`tiup-component-cluster-display` 命令返回的集群状态表格的第二列。
- 数据类型：`STRINGS`
- 如果不指定该选项，默认开启所有角色的自启。

> 注意：
>
> 若同时指定了 `-N, --node`，那么将开启它们的交集中的服务自启。

-h, –help

- 输出帮助信息。
- 数据类型：`BOOLEAN`
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

tiup-cluster 的执行日志。

<< 返回上一页 - TiUP Cluster 命令清单

13.4.7.3.12 tiup cluster help

tiup-cluster 在命令行界面为用户提供了丰富的帮助信息，这些帮助信息可以通过 `help` 命令或者 `--help` 参数获得。`tiup cluster help <command>` 基本等价于 `tiup cluster <command> --help`。

语法

```
tiup cluster help [command] [flags]
```

`[command]` 用于指定要查看哪个命令的帮助信息，若不指定，则查看 tiup-cluster 自身的帮助信息。

-h, –help

- 输出帮助信息。
- 数据类型：`BOOLEAN`

- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

`[command]` 或 tiup-cluster 的帮助信息。

<< 返回上一页 - TiUP Cluster 命令清单

13.4.7.3.13 tiup cluster import

在 TiDB 4.0 以前的版本，集群多是通过 TiDB Ansible 部署的。TiUP Cluster 提供了 `import` 命令用于让这些集群过渡到使用 tiup-cluster 组件管理。

> 注意：
>
> - TiDB Ansible 配置导入到 TiUP 中管理后，不能再通过 TiDB Ansible 对集群进行操作，否则可能因元信息不一致造成冲突
> - 如果使用 TiDB Ansible 部署的集群存在以下情况之一，暂不支持导入：
>   - 启用了 TLS 加密功能的集群
>   - 纯 KV 集群（没有 TiDB 实例的集群）
>   - 启用了 Kafka 的集群
>   - 启用了 Spark 的集群
>   - 启用了 TiDB Lightning/ Importer 的集群
>   - 仍使用老版本 push 的方式收集监控指标（从 v3.0 起默认为 `pull` 模式，如果没有特意调整过则可以支持）
>   - 在 `inventory.ini` 配置文件中单独为机器的 `node_exporter` / `blackbox_exporter` 通过 `node_exporter_port`/`blackbox_exporter_port` 设置了非默认端口（在 `group_vars` 目录中统一配置的可以兼容）
> - 如果使用 TiDB Ansible 部署的集群中有部分节点未部署监控，应当先使用 TiDB Ansible 在 `inventory.ini` 文件的 `monitored_servers` 分组中补充对应节点的信息，并通过 `deploy.yaml` playbook 将补充的监控组件部署完整。否则在集群数据导入 TiUP 后进行其他运维操作时，可能会因监控组件缺失而出错。

语法

```
tiup cluster import [flags]
```

选项

-d, –dir string（string，默认当前目录）

指定 TiDB Ansible 所在目录。

–ansible-config（string，默认 “./ansible.cfg”）

指定 Ansible 的配置文件路径。

–inventory string（string，默认 “inventory.ini” ）

指定 Ansible inventory 文件的名字。

–no-backup

- 默认情况下，导入成功之后会将 `--dir` 指定的目录里所有内容备份到 `${TIUP_HOME}/.tiup/storage/cluster/clusters/{cluster-name}/ansible-backup` 下。该选项用于禁用默认的备份步骤，如果该目录下有多个 inventory 文件（部署了多个集群），推荐禁用默认备份。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

–rename（string，默认为空）

重命名导入的集群。默认集群名为 inventory 中指定的 cluster_name。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

导入过程的日志信息。

<< 返回上一页 - TiUP Cluster 命令清单

13.4.7.3.14 tiup cluster list

tiup-cluster 支持使用同一个中控机部署多套集群，而命令 `tiup cluster list` 可以查看当前登录的用户使用该中控机部署了哪些集群。

> 注意：
>
> 部署的集群数据默认放在 `~/.tiup/storage/cluster/clusters/` 目录下，因此在同一台中控机上，当前登录用户无法查看其他用户部署的集群。

语法

```
tiup cluster list [flags]
```

选项

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN

- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

输出含有以下字段的表格：

- Name：集群名字
- User：部署用户
- Version：集群版本
- Path：集群部署数据在中控机上的路径
- PrivateKey：连接集群的私钥所在路径

<< 返回上一页 - TiUP Cluster 命令清单

13.4.7.3.15 tiup cluster meta backup

如果运维所需的 TiUP meta 文件丢失，会导致无法继续使用 TiUP 管理集群。你可以通过 `tiup cluster meta backup` 命令定期备份 TiUP meta 文件。

语法

```
tiup cluster meta backup <cluster-name> [flags]
```

`<cluster-name>` 为要操作的集群名字，如果忘记集群名字可通过集群列表查看。

选项

–file（string，默认为当前目录）

指定 TiUP meta 备份文件存储的目标目录。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

tiup-cluster 的执行日志。

<< 返回上一页 - TiUP Cluster 命令清单

13.4.7.3.16 tiup cluster meta restore

当需要恢复 TiUP meta 文件时，可以通过 `tiup cluster meta restore` 命令从备份文件中恢复。

语法

```
tiup cluster meta restore <cluster-name> <backup-file> [flags]
```

- <cluster-name> 代表需要执行操作的集群名。
- <backup-file> 代表 TiUP meta 备份文件所在的文件路径。

> 注意：
>
> 恢复操作会覆盖当前的 meta 文件，建议仅在 meta 文件丢失的情况下进行恢复。

选项

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 false。在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。

输出

tiup-cluster 的执行日志。

<< 返回上一页 - TiUP Cluster 命令清单

13.4.7.3.17 tiup cluster patch

在集群运行过程中，如果需要动态替换某个服务的二进制文件（即替换过程中保持集群可用），那么可以使用 tiup cluster patch 命令，它会完成以下几件事情：

- 将用于替换的二进制包上传到目标机器
- 如果目标服务是 TiKV、TiFlash 或者 TiDB Binlog 之类的存储服务，则先通过 API 下线节点
- 停止目标服务
- 解压二进制包，替换服务
- 启动目标服务

语法

```
tiup cluster patch <cluster-name> <package-path> [flags]
```

- <cluster-name> 代表要操作的集群名
- <package-path> 为用于替换的二进制包路径

准备二进制包

在运行 tiup cluster patch 命令之前，你需要打包所需的二进制文件。请按照以下步骤操作：

1. 确定以下变量的值：
    - ${component}：需要替换的组件名（例如 tidb、tikv、pd）。

- `${version}`：组件的版本（例如 `v7.0.0`、`v6.5.1`）。
- `${os}`：操作系统 (`linux`)。
- `${arch}`：组件运行的平台 (`amd64`、`arm64`)。

2. 下载当前的组件包：

```
wget https://tiup-mirrors.pingcap.com/${component}-${version}-${os}-${arch}.tar.gz -O /tmp/$
    ↪ {component}-${version}-${os}-${arch}.tar.gz
```

3. 创建临时打包目录：

```
mkdir -p /tmp/package && cd /tmp/package
```

4. 解压原来的二进制包：

```
tar xf /tmp/${component}-${version}-${os}-${arch}.tar.gz
```

5. 查看临时打包目录中的文件结构：

```
find .
```

6. 将要替换的二进制文件或配置文件复制到临时目录的对应位置。

7. 将临时目录中的所有文件打包：

```
tar czf /tmp/${component}-hotfix-${os}-${arch}.tar.gz *
```

完成上述步骤后，你可以在 `tiup cluster patch` 命令中使用 `/tmp/${component}-hotfix-${os}-${arch}.tar.gz` 作为 `<package-path>`。

选项

–overwrite

- 对某个组件（比如 TiDB，TiKV）进行 patch 后，如果要在该集群扩容该组件，tiup-cluster 会默认使用 patch 前的版本。如果希望后续扩容的时候也使用 patch 之后的版本，需要指定 `--overwrite` 选项。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

–transfer-timeout（uint，默认 600）

在重启 PD 或 TiKV 时，会先将被重启节点的 leader 迁移到其他节点，迁移过程会需要一定时间，可以通过设置 `--transfer-timeout` 设置最长等待时间（单位为秒），超时之后会跳过等待直接重启服务。

> 注意：
>
> 若出现跳过等待直接重启的情况，服务性能可能会出现抖动。

-N, –node（strings，默认为 []，未选中任何节点）

指定要替换的节点，该选项的值为以逗号分割的节点 ID 列表，节点 ID 为集群状态表格的第一列。

> 注意：
> 若同时指定了 `-R, --role`，那么将替换它们的交集中的服务。

-R, –role（strings，默认为 []，未选中任何角色）

指定要替换的角色，该选项的值为以逗号分割的节点角色列表，角色为集群状态表格的第二列。

> 注意：
> 若同时指定了 `-N, --node`，那么将替换它们的交集中的服务。

–offline

- 声明当前集群处于停止状态。指定该选项时，TiUP Cluster 仅原地替换集群组件的二进制文件，不执行迁移 Leader 以及重启服务等操作。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

tiup-cluster 的执行日志。

<< 返回上一页 - TiUP Cluster 命令清单

13.4.7.3.18 tiup cluster prune

在缩容集群时，对于某些组件，并不会立即停止服务并删除数据，而是需要等数据调度完成之后，用户手动执行 `tiup cluster prune` 命令清理。

语法

```
tiup cluster prune <cluster-name> [flags]
```

选项

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

清理过程的日志。

<< 返回上一页 - TiUP Cluster 命令清单

13.4.7.3.19 tiup cluster reload

在修改集群配置之后，需要通过 `tiup cluster reload` 命令让集群重新加载配置才会生效，该命令会将中控机的配置发布到服务运行的远端机器，并按照升级的流程按顺序重启服务，重启过程中集群仍然可用。

语法

```
tiup cluster reload <cluster-name> [flags]
```

`<cluster-name>` 代表要操作的集群名。

选项

–force

- 忽略重新加载过程中的错误，强制 reload。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

–transfer-timeout（uint，默认 600）

在重启 PD 或 TiKV 时，会先将被重启节点的 leader 迁移到其他节点，迁移过程会需要一定时间，可以通过设置 `--transfer-timeout` 设置最长等待时间（单位为秒），超时之后会跳过等待直接重启服务。

> 注意：
>
> 若出现跳过等待直接重启的情况，服务性能可能会出现抖动。

–ignore-config-check

- 在组件二进制文件部署之后，TiUP 会对 TiDB，TiKV 和 PD 组件执行配置检查，检查方式为 `<binary> --config-check <config-file>`，其中 `<binary>` 为部署的二进制文件的路径，`<config-file>` 为根据用户配置生成的配置文件。如果想要跳过该项检查，可以使用该选项。
- 数据类型：BOOLEAN

- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

-N, –node（strings，默认为 []，表示所有节点）

指定要重启的节点，不指定则表示所有节点。该选项的值为以逗号分割的节点 ID 列表，节点 ID 为集群状态表格的第一列。

> 注意：
>
> - 若同时指定了 `-R, --role`，那么将重启它们的交集中的服务
> - 若指定了选项 `--skip-restart`，则该选项无效

-R, –role（strings，默认为 []，表示所有角色）

指定要重启的角色，不指定则表示所有角色。该选项的值为以逗号分割的节点角色列表，角色为集群状态表格的第二列。

> 注意：
>
> 1. 若同时指定了 `-N, --node`，那么将重启它们的交集中的服务
> 2. 若指定了选项 `--skip-restart`，则该选项无效

–skip-restart

- 命令 `tiup cluster reload` 会执行两个操作：
  - 刷新所有节点配置
  - 重启指定节点
- 该选项指定后仅刷新配置，不重启任何节点，这样刷新的配置也不会应用，需要等对应服务下次重启才会生效。
- 数据类型：`BOOLEAN`
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

-h, –help

- 输出帮助信息。
- 数据类型：`BOOLEAN`
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

tiup-cluster 的执行日志。

<< 返回上一页 - TiUP Cluster 命令清单

13.4.7.3.20 tiup cluster rename

集群名字是部署集群时指定的，在集群部署之后，如果想更改集群名字，可以通过 `tiup cluster rename` 命令来实现。

> 注意：
>
> 如果配置了 `grafana_servers` 的 dashboard_dir 字段，在执行 tiup cluster rename 命令进行集群重命名后，需要额外做以下操作：
>
> - 对于本地的 dashboards 目录中的 `*.json` 文件，将 datasource 字段的值更新为新的集群名（这是因为 datasource 是以集群名命名的）
> - 执行 `tiup cluster reload -R grafana` 命令

语法

```
tiup cluster rename <old-cluster-name> <new-cluster-name> [flags]
```

- `<old-cluster-name>` 老的集群名
- `<new-cluster-name>` 新的集群名

选项

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

tiup-cluster 的执行日志。

<< 返回上一页 - TiUP Cluster 命令清单

13.4.7.3.21 tiup cluster replay

对集群进行升级或重启等操作时，操作有可能因为环境的原因而偶然失败。这时如果重新进行操作，需要从头开始执行所有步骤。如果集群规模较大，会耗费较长时间。此时可以使用 `tiup cluster replay` 命令重试刚才失败的命令，并且跳过已经成功的步骤。

语法

```
tiup cluster replay <audit-id> [flags]
```

- `<audit-id>` 代表要重试的命令对应的 `audit-id`。使用`tiup cluster audit` 可查看历史命令及其 `audit-id`。

选项

-h, –help

输出帮助信息。

输出

<audit-id> 对应的命令的输出。

<< 返回上一页 - TiUP Cluster 命令清单

13.4.7.3.22 tiup cluster restart

命令 tiup cluster restart 用于重启指定集群的所有或部分服务。

> 注意：
> 重启过程中会有一段时间服务不可用。

语法

```
tiup cluster restart <cluster-name> [flags]
```

<cluster-name> 为要操作的集群名字，如果忘记集群名字可通过集群列表查看。

选项

-N, –node（strings，默认为 []，表示所有节点）

指定要重启的节点，不指定则表示所有节点。该选项的值为以逗号分割的节点 ID 列表，节点 ID 为集群状态表格的第一列。

> 注意：
> 若同时指定了 -R, --role，那么将重启它们的交集中的服务。

-R, –role（strings，默认为 []，表示所有角色）

指定要重启的角色，不指定则表示所有角色。该选项的值为以逗号分割的节点角色列表，角色为集群状态表格的第二列。

> 注意：
> 若同时指定了 -N, --node，那么将重启它们的交集中的服务。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

重启服务的日志。

<< 返回上一页 - TiUP Cluster 命令清单

13.4.7.3.23 tiup cluster scale-in

`tiup cluster scale-in` 命令用于集群缩容，缩容即下线服务，最终会将指定的节点从集群中移除，并删除遗留的相关文件。

下线特殊处理

由于 TiKV，TiFlash 和 TiDB Binlog 组件的下线是异步的（需要先通过 API 执行移除操作）并且下线过程耗时较长（需要持续观察节点是否已经下线成功），所以对 TiKV，TiFlash 和 TiDB Binlog 组件做了特殊处理：

- 对 TiKV，TiFlash 及 TiDB Binlog 组件的操作:
  - tiup-cluster 通过 API 将其下线后直接退出而不等待下线完成
  - 执行 `tiup cluster display` 查看下线节点的状态，等待其状态变为 Tombstone
  - 执行 `tiup cluster prune` 命令清理 Tombstone 节点，该命令会执行以下操作：
    * 停止已经下线掉的节点的服务
    * 清理已经下线掉的节点的相关数据文件
    * 更新集群的拓扑，移除已经下线掉的节点
- 对其他组件的操作
  - 下线 PD 组件时，会通过 API 将指定节点从集群中删除掉（这个过程很快），然后停掉指定 PD 的服务并且清除该节点的相关数据文件
  - 下线其他组件时，直接停止并且清除节点的相关数据文件

语法

```
tiup cluster scale-in <cluster-name> [flags]
```

`<cluster-name>` 为要操作的集群名字，如果忘记集群名字可通过集群列表查看。

选项

-N, –node（strings，无默认值，必须非空）

选择要缩容的节点，若缩容多个节点，以逗号分割。

–force

- 在某些情况下，有可能被缩容的节点宿主机已经宕机，导致无法通过 SSH 连接到节点进行操作，这个时候可以通过 `--force` 选项强制将其从集群中移除。

- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

> **警告：**
>
> 使用该选项强制移除正在服务和下线中的 TiKV / TiFlash 节点时，这些节点会被直接删除，不等待数据调度完成，因此这个场景下，数据丢失风险非常大。不建议对未宕机的节点使用该选项。如果元数据所在的 Region 发生数据丢失，整个集群将不可用且无法恢复。

–transfer-timeout（uint，默认 600）

在缩容 PD 或 TiKV 时，会先将被缩容节点的 leader 迁移到其他节点，迁移过程会需要一定时间，可以通过设置 `--transfer-timeout` 设置最长等待时间（单位为秒），超时之后会跳过等待直接缩容服务。

> **注意：**
>
> 若出现跳过等待直接缩容的情况，服务性能可能会出现抖动。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

缩容日志

<< 返回上一页 - TiUP Cluster 命令清单

13.4.7.3.24 tiup cluster scale-out

`tiup cluster scale-out` 命令用于集群扩容，扩容的内部逻辑与部署类似，tiup-cluster 组件会先建立新节点的 SSH 连接，在目标节点上创建必要的目录，然后执行部署并且启动服务。其中 PD 节点的扩容会通过 join 方式加入到集群中，并且会更新与 PD 有关联的服务的配置；其他服务直接启动加入到集群中。

语法

```
tiup cluster scale-out <cluster-name> <topology.yaml> [flags]
```

- `<cluster-name>` 为要操作的集群名字，如果忘记集群名字可通过集群列表查看
- `<topology.yaml>` 为事先编写好的扩容拓扑文件，该文件应当仅包含扩容部分的拓扑

选项

-u, –user（string，默认为当前执行命令的用户）

指定连接目标机器的用户名，该用户在目标机器上需要有免密 sudo root 的权限。

-i, –identity_file（string，默认 ~/.ssh/id_rsa）

指定连接目标机器的密钥文件。

-p, –password

- 在连接目标机器时使用密码登录，不可和 `-i/--identity_file` 同时使用。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

–no-labels

- 当两个或多个 TiKV 部署到同一台机器时，会存在一个风险：由于 PD 无法感知集群的拓扑结构，可能将一个 Region 的多个副本调度到一台物理机上的不同 TiKV，这样这台物理机就成为了单点。为了避免这种情况，用户可以通过 label 来指定 PD 不要将相同的 Region 调度到同一台机器上（配置方式参考通过拓扑 label 进行副本调度）。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

但是对于测试环境，可能并不在意是否将一个 Region 的副本调度到了同一台机器上，这个时候可以使用 `--no-labels` 来绕过检查。

–skip-create-user

在扩容集群时，tiup-cluster 会先检查拓扑文件中指定的用户名是否存在，如果不存在就会创建一个。指定 `--skip-create-user` 选项后不再检查用户是否存在，直接跳过创建步骤。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

扩容日志

<< 返回上一页 - TiUP Cluster 命令清单

13.4.7.3.25 tiup cluster start

命令 `tiup cluster start` 用于启动指定集群的所有或部分服务。

语法

```
tiup cluster start <cluster-name> [flags]
```

<cluster-name> 为要操作的集群名字，如果忘记集群名字可通过集群列表查看。

选项

–init

以安全方式启动集群。推荐在集群第一次启动时使用，该方式会在启动时自动生成 TiDB root 用户的密码，并在命令行界面返回密码。

> 注意：
>
> - 使用安全启动方式后，不能通过无密码的 root 用户登录数据库，你需要记录命令行返回的密码进行后续操作。
> - 该自动生成的密码只会返回一次，如果没有记录或者忘记该密码，请参照忘记 root 密码修改密码。

-N, –node（strings，默认为 []，表示所有节点）

指定要启动的节点，不指定则表示所有节点。该选项的值为以逗号分割的节点 ID 列表，节点 ID 为集群状态表格的第一列。

> 注意：
>
> 若同时指定了 -R, --role，那么将启动它们的交集中的服务。

-R, –role（strings，默认为 []，表示所有角色）

指定要启动的角色，不指定则表示所有角色。该选项的值为以逗号分割的节点角色列表，角色为集群状态表格的第二列。

> 注意：
>
> 若同时指定了 -N, --node，那么将启动它们的交集中的服务。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 false。在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。

输出

启动日志。

<< 返回上一页 - TiUP Cluster 命令清单

13.4.7.3.26 tiup cluster stop

命令 `tiup cluster stop` 用于停止指定集群的所有服务或部分服务。

> **注意：**
>
> 核心服务停止后集群将无法提供服务。

语法

```
tiup cluster stop <cluster-name> [flags]
```

`<cluster-name>` 为要操作的集群名字，如果忘记集群名字可通过集群列表查看。

选项

-N, –node（strings，默认为 []，表示所有节点）

指定要停止的节点，不指定则表示所有节点。该选项的值为以逗号分割的节点 ID 列表，节点 ID 为集群状态表格的第一列。

> **注意：**
>
> 若同时指定了 `-R, --role`，那么将停止它们的交集中的服务。

-R, –role（strings，默认为 []，表示所有角色）

指定要停止的角色，不指定则表示所有角色。该选项的值为以逗号分割的节点角色列表，角色为集群状态表格的第二列。

> **注意：**
>
> 若同时指定了 `-N, --node`，那么将停止它们的交集中的服务。

-h, –help

- 输出帮助信息。
- 数据类型：`BOOLEAN`
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

停止服务的日志。

<< 返回上一页 - TiUP Cluster 命令清单

13.4.7.3.27 tiup cluster template

部署集群之前，需要准备一份集群的拓扑文件。TiUP 内置了拓扑文件的模版，用户可以通过修改该模版来生成最终的拓扑文件。使用 `tiup cluster template` 命令可以输出 TiUP 内置的模版内容。

语法

```
tiup cluster template [flags]
```

如果不指定该选项，输出的默认模版包含以下实例：

- 3 个 PD 实例
- 3 个 TiKV 实例
- 1 个 TiDB 实例
- 1 个 Prometheus 实例
- 1 个 Grafana 实例
- 1 个 Alertmanager 实例

选项

–full

- 输出详细的拓扑模版，该模版会以注释的形式带上可配置的参数。在命令中添加该选项，可开启该选项。
- 如果不指定该选项，默认输出最简单的拓扑模版。

–multi-dc

- 输出多数据中心的拓扑模版。在命令中添加该选项，可开启该选项。
- 如果不指定该选项，默认输出单地单机房的拓扑模版。

-h, –help

输出帮助信息。

输出

根据指定选项输出拓扑模版，可重定向到拓扑文件中用于部署。

<< 返回上一页 - TiUP Cluster 命令清单

13.4.7.3.28 tiup cluster upgrade

命令 `tiup cluster upgrade` 用于将指定集群升级到特定版本。

语法

```
tiup cluster upgrade <cluster-name> <version> [flags]
```

- `<cluster-name>` 为要操作的集群名字，如果忘记集群名字可通过集群列表查看。

- <version> 为要升级到的目标版本，例如 v6.5.0。目前仅允许升级到比当前集群更高的版本，不允许升级到比当前集群更低的版本，即不允许降级。同时也不允许升级成 nightly 版本

选项

–force

- 升级集群需要保证集群目前是启动的，在某些情况下，可能希望在集群未启动的状态下升级，这时候可以使用 --force 忽略升级过程的错误，强制替换二进制文件并启动集群。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 false。在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。

> 注意：
>
> 对正在提供服务的集群强制升级可能导致集群服务不可用。对于未启动的集群，升级成功后会自动启动集群。

–transfer-timeout（uint，默认 600）

在升级 PD 或 TiKV 时，会先将被升级节点的 leader 迁移到其他节点，迁移过程会需要一定时间，可以通过设置 --transfer-timeout 设置最长等待时间（单位为秒），超时之后会跳过等待直接升级服务。

> 注意：
>
> 若出现跳过等待直接升级的情况，服务性能可能会出现抖动。

–ignore-config-check

- 在二进制文件更新之后，TiUP 会对 TiDB，TiKV 和 PD 组件执行配置检查，检查方式为 <binary> --config-check <config-file>，其中 <binary> 为新部署的二进制文件的路径，<config-file> 为根据用户配置生成的配置文件。如果想要跳过该项检查，可以使用该选项。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 false。在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。

–offline

- 声明当前集群处于停止状态。指定该选项时，TiUP Cluster 仅原地替换集群组件的二进制文件，不执行迁移 Leader 以及重启服务等操作。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 false。在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

升级服务的日志。

<< 返回上一页 - TiUP Cluster 命令清单

13.4.7.4 TiUP DM 命令

13.4.7.4.1 TiUP DM

类似于TiUP Cluster 对 TiDB 集群的管理，TiUP DM 是用于对 DM 集群的日常运维工作，包括部署、启动、关闭、销毁、弹性扩缩容、升级 DM 集群、管理 DM 集群参数。

语法

```
tiup dm [command] [flags]
```

`[command]` 代表命令名字，支持的命令列表请参考下方命令清单。

选项

–ssh (string，默认 builtin)

指定 SSH 客户端连接远端（部署 TiDB 服务的机器）执行命令，支持以下几个值：

- builtin：使用 tiup-cluster 内置的 easyssh 客户端
- system：使用当前操作系统默认的 ssh 客户端
- none：不使用 ssh 客户端，这种方式只支持部署到当前机器

–ssh-timeout（uint，默认 5）

设置 ssh 连接超时时间，单位为秒。

–wait-timeout（uint，默认 120）

运维过程中涉及到很多操作：指定 systemctl 启动/停止服务，等待端口上线/下线等，每个操作可能会消耗数秒，`--wait-timeout` 用于设置每个步骤的最长等待时间（单位为秒），超时后报错退出。

-y, –yes

- 跳过所有风险操作的二次确认，除非是使用脚本调用 TiUP，否则不推荐使用。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

-v, –version

- 输出 tiup-dm 当前版本信息。

- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

-h, –help

- 输出相关命令的帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

命令清单

- import：导入使用 DM-Ansible 部署的 DM v1.0 集群
- template：输出拓扑模版
- deploy：根据指定拓扑部署集群
- list：查询已部署的集群列表
- display：展示指定集群状态
- start：启动指定集群
- stop：停止指定集群
- restart：重启指定集群
- scale-in：缩容指定集群
- scale-out：扩容指定集群
- upgrade：升级指定集群
- prune：销毁指定集群中状态为 Tombstone 的实例
- edit-config：修改指定集群配置
- reload：重载指定集群配置
- patch：替换已部署集群的某个服务
- destroy：销毁指定集群
- audit：查询集群操作审计日志
- replay：重试指定命令
- enable：开启指定集群或服务开机自启动
- disable：关闭指定集群或服务开机自启动
- help：输出帮助信息

<< 返回上一页 - TiUP 组件清单

13.4.7.4.2 tiup dm audit

命令 `tiup dm audit` 可以用于查看执行在所有集群上的历史命令，以及每个命令的执行日志。

语法

```
tiup dm audit [audit-id] [flags]
```

- 若不填写 `[audit-id]` 则按时间倒序输出操作记录的表格，第一列为 audit-id
- 若填写 `[audit-id]` 则查看指定的 audit-id 的执行日志

选项

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

- 若指定 `[audit-id]`，则输出对应的执行日志
- 若不指定 `[audit-id]` 则输出含有以下字段的表格：
  - ID：该条记录对应的 audit-id
  - Time：该条记录对应的命令执行时间
  - Command：该条记录对应的命令

<< 返回上一页 - TiUP DM 命令清单

13.4.7.4.3 tiup dm deploy

命令 `tiup dm deploy` 用于部署一个全新的集群。

语法

```
tiup dm deploy <cluster-name> <version> <topology.yaml> [flags]
```

- `<cluster-name>` 表示新集群的名字，不能和现有集群同名
- `<version>` 为要部署的 DM 集群版本号，如 `v2.0.0`
- `<topology.yaml>` 为事先编写好的拓扑文件

选项

-u, –user（string，默认为当前执行命令的用户）

指定连接目标机器的用户名，该用户在目标机器上需要有免密 sudo root 的权限。

-i, –identity_file（string，默认 ~/.ssh/id_rsa）

指定连接目标机器的密钥文件。

-p, –password

- 在连接目标机器时使用密码登录，不可和 `-i/--identity_file` 同时使用。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

-h, –help

- 输出帮助信息。

- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

部署日志。

<< 返回上一页 - TiUP DM 命令清单

13.4.7.4.4 tiup dm destroy

当业务下线之后，如果想将集群占有的机器释放出来让给其他业务使用，需要清理掉集群上的数据以及部署的二进制文件。`tiup dm destroy` 命令会执行以下操作销毁集群：

- 停止集群
- 对于每个服务，删除其日志目录，部署目录，数据目录
- 如果各个服务的数据目录/部署目录的父目录是由 tiup-dm 创建的，也一并删除

语法

```
tiup dm destroy <cluster-name> [flags]
```

`<cluster-name>` 为要销毁的集群名字。

选项

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

tiup-dm 的执行日志。

<< 返回上一页 - TiUP DM 命令清单

13.4.7.4.5 tiup dm disable

集群服务所在的机器重启之后会自启动，命令 `tiup dm disable` 用于关闭该自启动。该命令会在指定的节点上执行 `systemctl disable <service>` 来关闭服务的自启动。

语法

```
tiup dm disable <cluster-name> [flags]
```

`<cluster-name>` 为要关闭自启的集群。

选项

-N, –node

- 指定要关闭自启的节点，该选项的值为以逗号分割的节点 ID 列表，节点 ID 为集群状态表格的第一列。
- 数据类型：STRINGS
- 如果不指定该选项，默认关闭所有节点的自启。

> 注意：
>
> 若同时指定了 -R, --role，那么将关闭它们的交集中的服务自启。

-R, –role

- 指定要关闭自启的角色，该选项的值为以逗号分割的节点角色列表，角色为集群状态表格的第二列。
- 数据类型：STRINGS
- 如果不指定该选项，默认关闭所有角色的自启。

> 注意：
>
> 若同时指定了 -N, --node，那么将关闭它们的交集中的服务自启。

-h, –help

输出帮助信息。

输出

tiup-dm 的执行日志。

<< 返回上一页 - TiUP DM 命令清单

13.4.7.4.6 tiup dm display

如果想查看集群中每个组件的运行状态，逐一登录到各个机器上查看显然很低效。因此，tiup-dm 提供了 tiup dm display 命令来高效完成这件工作。

语法

```
tiup dm display <cluster-name> [flags]
```

<cluster-name> 为要操作的集群名字，如果忘记集群名字可查看集群列表。

选项

-N, –node（strings，默认为 []，表示所有节点）

指定要查询的节点，不指定则表示所有节点。该选项的值为以逗号分割的节点 ID 列表，如果不确定要查询节点的 ID，不指定此选项，输出会显示所有节点的 ID 和状态信息。

> **注意：**
>
> 若同时指定了 `-R, --role`，那么将查询它们的交集中的服务状态。

-R, –role（strings，默认为 []，表示所有角色）

指定要查询的角色，不指定则表示所有角色。该选项的值为以逗号分割的节点角色列表，如果不确定要查询节点的角色，不指定此选项，输出会显示所有节点的角色和状态信息。

> **注意：**
>
> 若同时指定了 `-N, --node`，那么将查询它们的交集中的服务状态。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

- 集群名称
- 集群版本
- SSH 客户端类型
- 含有以下字段的表格：
  - ID：节点 ID，由 IP:PORT 构成
  - Role：该节点部署的服务角色（如 TiDB 和 TiKV 等）
  - Host：该节点对应的机器 IP
  - Ports：服务占用的端口号
  - OS/Arch：该节点的操作系统和机器架构
  - Status：该节点服务当前的状态
  - Data Dir：服务的数据目录，- 表示没有数据目录
  - Deploy Dir：服务的部署目录

<< 返回上一页 - TiUP DM 命令清单

13.4.7.4.7 tiup dm edit-config

在部署集群之后，如果需要再调整集群服务的配置，则可以使用命令 `tiup dm edit-config`，它会启动一个编辑器（默认为 `$EDITOR` 环境变量指定的值，当 EDITOR 环境变量不存在时，使用 vi 打开）允许用户修改指定集群的拓扑文件。

注意：

- 修改配置时不能增删机器，增删机器属于集群扩容和集群缩容的功能。
- 执行完该命令后配置只是在中控机上修改了，要应用配置需要执行 `tiup dm reload` 命令来重新加载。

语法

```
tiup dm edit-config <cluster-name> [flags]
```

`<cluster-name>` 代表要操作的集群名。

选项

-h, –help

- 输出帮助信息。
- 数据类型：`BOOLEAN`
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

- 正常情况无输出
- 若修改了不能修改的字段，则保存文件时报错并提示用户重新编辑，不能修改的字段参考拓扑文件中的相关描述

<< 返回上一页 - TiUP DM 命令清单

13.4.7.4.8 tiup dm enable

命令 `tiup dm enable` 用于设置集群服务在机器重启后的自启动，该命令会到指定的节点上去执行 `systemctl enable <service>` 来开启服务的自启。

语法

```
tiup dm enable <cluster-name> [flags]
```

`<cluster-name>` 为要启用自启的集群。

选项

-N, –node

- 指定要开启自启的节点，该选项的值为以逗号分割的节点 ID 列表，节点 ID 为集群状态表格的第一列。
- 数据类型：`STRINGS`
- 如果不指定该选项，默认开启所有节点的自启。

> **注意：**
>
> 若同时指定了 `-R, --role`，那么将开启它们的交集中的服务自启。

-R, –role

- 指定要开启自启的角色，该选项的值为以逗号分割的节点角色列表，角色为集群状态表格的第二列。
- 数据类型：STRINGS
- 如果不指定该选项，默认开启所有角色的自启。

> **注意：**
>
> 若同时指定了 `-N, --node`，那么将开启它们的交集中的服务自启。

-h, –help

输出帮助信息。

输出

tiup-dm 的执行日志。

<< 返回上一页 - TiUP DM 命令清单

13.4.7.4.9 tiup dm help

tiup-dm 在命令行界面为用户提供了丰富的帮助信息，这些帮助信息可以通过 `help` 命令或者 `--help` 参数获得。基本上，`tiup dm help <command>` 等价于 `tiup dm <command> --help`。

语法

```
tiup dm help [command] [flags]
```

`[command]` 用于指定要查看哪个命令的帮助信息，若不指定，则查看 tiup-dm 自身的帮助信息。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

`[command]` 或 tiup-dm 的帮助信息。

<< 返回上一页 - TiUP DM 命令清单

13.4.7.4.10 tiup dm import 仅适用于升级 DM v1.0

该命令仅适用于将 DM 集群从 v1.0 升级到 v2.0 或更高版本。

在 DM 1.0 版本，集群基本是通过 TiDB Ansible 部署的，TiUP DM 提供了 `import` 命令导入 v1.0 的集群并重新部署 v2.0 的集群。

> 注意：
>
> - 不支持导入 v1.0 集群中的 DM Portal 组件
> - 导入前请先停止原集群
> - 对于需要升级到 v2.0 的数据迁移任务，请不要执行 `stop-task`
> - 仅支持导入到 v2.0.0-rc.2 或更高版本
> - `import` 命令用于将 DM v1.0 集群导入到全新的 v2.0 集群。如果需要将数据迁移任务导入到已有的 v2.0 集群，请参考TiDB Data Migration 1.0.x 到 2.0.x+ 手动升级
> - 部分组件生成的部署目录会跟原集群不一样，具体可以使用 `display` 命令查看
> - 导入前运行 `tiup update --self && tiup update dm` 确认升级 TiUP DM 组件到最新版本
> - 导入后集群中仅会有一个 DM-master 节点，可参考扩容节点对 DM-master 进行扩容

语法

```
tiup dm import [flags]
```

选项

-v, –cluster-version（string，required）

重新部署的版本号，必须指定 v2.0.0-rc.2 或更高版本。

-d, –dir string（string，默认当前目录）

指定 TiDB Ansible 所在目录。

–inventory string（string，默认 “inventory.ini”）

指定 Ansible inventory 文件的名字。

–rename（string，默认为空）

重命名导入的集群。默认集群名为 inventory 中指定的 `cluster_name`。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

导入过程的日志信息。

<< 返回上一页 - TiUP DM 命令清单

13.4.7.4.11 tiup dm list

tiup-dm 支持使用同一个中控机部署多套集群，而命令 `tiup dm list` 可以查看当前登录的用户使用该中控机部署了哪些集群。

> **注意：**
>
> 部署的集群数据默认放在 `~/.tiup/storage/dm/clusters/` 目录下，因此在同一台中控机上，当前登录用户无法查看其他用户部署的集群。

语法

```
tiup dm list [flags]
```

选项

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

输出含有以下字段的表格：

- Name：集群名字
- User：部署用户
- Version：集群版本
- Path：集群部署数据在中控机上的路径
- PrivateKey：连接集群的私钥所在路径

<< 返回上一页 - TiUP DM 命令清单

13.4.7.4.12 在线应用 Hotfix 到 DM 集群

在集群运行过程中，如果需要动态替换某个服务的二进制文件（即替换过程中保持集群可用），那么可以使用 `tiup dm patch` 命令，它会完成以下几件事情：

- 将用于替换的二进制包上传到目标机器
- 通过 API 下线节点
- 停止目标服务
- 解压二进制包，替换服务
- 启动目标服务

语法

```
tiup dm patch <cluster-name> <package-path> [flags]
```

- <cluster-name> 代表要操作的集群名。
- <package-path> 为用于替换的二进制包。

准备条件

执行 tiup dm patch 命令前，需要进行以下操作准备用于替换的二进制包：

- 确定当前要替换的组件名称 ${component} (dm-master，dm-worker 等) 以及其版本 ${version} (v2.0.0，v2.0.1 等)，以及其运行的平台 ${os} (linux) 和 ${arch} (amd64, arm64)
- 下载当前的组件包：wget https://tiup-mirrors.pingcap.com/${component}-${version}-${os}-${arch}.tar.gz -O /tmp/${component}-${version}-${os}-${arch}.tar.gz
- 建立临时打包目录：mkdir -p /tmp/package && cd /tmp/package
- 解压原来的二进制包：tar xf /tmp/${component}-${version}-${os}-${arch}.tar.gz
- 查看临时打包目录中的文件结构：find .
- 将要替换的二进制文件或配置文件复制到临时目录的对应位置
- 重新打包 tar czf /tmp/${component}-hotfix-${os}-${arch}.tar.gz *

完成以上操作后，/tmp/${component}-hotfix-${os}-${arch}.tar.gz 就可以作为 <package-path> 用于 patch 命令中。

选项

–overwrite

- 对某个组件（比如 TiDB，TiKV）进行 patch 之后，该集群扩容该组件时，tiup-dm 默认会用原来的版本。如果希望后续扩容的时候也使用 patch 之后的版本的话，就需要指定 --overwrite 选项。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 false。在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。

-N, –node（strings，默认为 []，未选中任何节点）

指定要替换的节点，该选项的值为以逗号分割的节点 ID 列表，节点 ID 为集群状态表格的第一列。

> 注意：
>
> 若同时指定了 -R, --role，那么将替换它们的交集中的服务。

-R, –role（strings，默认为 []，未选中任何角色）

指定要替换的角色，该选项的值为以逗号分割的节点角色列表，角色为集群状态表格的第二列。

> **注意：**
>
> 若同时指定了 `-N, --node`，那么将替换它们的交集中的服务。

–offline

声明当前集群处于离线状态。指定该选项时，TiUP DM 仅原地替换集群组件的二进制文件，不重启服务。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

在线应用 Hotfix 示例

以下将在使用 TiUP 部署的 DM 环境中演示如何应用 `v5.3.0-hotfix` 到 `v5.3.0`集群，其他部署方式可能需要调整部分操作。

> **注意：**
>
> Hotfix 仅用于紧急修复，其日常维护较为复杂，建议在正式版本发布后及时升级。

准备工作

在开始应用 Hotfix 之前，请准备好 Hotfix 补丁包 `dm-linux-amd64.tar.gz`，并确认当前环境 DM 软件版本：

```
/home/tidb/dm/deploy/dm-master-8261/bin/dm-master/dm-master -V
```

输出示例：

```
Release Version: v5.3.0
Git Commit Hash: 20626babf21fc381d4364646c40dd84598533d66
Git Branch: heads/refs/tags/v5.3.0
UTC Build Time: 2021-11-29 08:29:49
Go Version: go version go1.16.4 linux/amd64
```

制作 Patch 补丁包并应用到 DM 集群

1. 准备当前环境版本的 DM 软件包：

```
mkdir -p /tmp/package
tar -zxvf /root/.tiup/storage/dm/packages/dm-master-v5.3.0-linux-amd64.tar.gz -C /tmp/
    ↪ package/
tar -zxvf /root/.tiup/storage/dm/packages/dm-worker-v5.3.0-linux-amd64.tar.gz -C /tmp/
    ↪ package/
```

2. 替换新的二进制文件：

```
# 解压 Hotfix 压缩包并替换
cd /root; tar -zxvf dm-linux-amd64.tar.gz
cp /root/dm-linux-amd64/bin/dm-master /tmp/package/dm-master/dm-master
cp /root/dm-linux-amd64/bin/dm-worker /tmp/package/dm-worker/dm-worker

# 重新打包
# 注意，其他部署方式可能有所不同
cd /tmp/package/ && tar -czvf dm-master-hotfix-linux-amd64.tar.gz dm-master/
cd /tmp/package/ && tar -czvf dm-worker-hotfix-linux-amd64.tar.gz dm-worker/
```

3. 应用补丁。

   查询当前集群状态，以名称为 dm-test 的集群为例：

```
tiup dm display dm-test
```

   输出示例：

```
Cluster type:       dm
Cluster name:       dm-test
Cluster version:    v5.3.0
Deploy user:        tidb
SSH type:           builtin
ID                  Role                Host           Ports      OS/Arch       Status
    ↪ Data Dir                              Deploy Dir
--                  ----                ----           -----      -------       ------
    ↪ --------                              ----------
172.16.100.21:9093  alertmanager        172.16.100.21  9093/9094  linux/x86_64  Up
    ↪ /home/tidb/dm/data/alertmanager-9093  /home/tidb/dm/deploy/alertmanager-9093
172.16.100.21:8261  dm-master           172.16.100.21  8261/8291  linux/x86_64  Healthy|L
    ↪ /home/tidb/dm/data/dm-master-8261     /home/tidb/dm/deploy/dm-master-8261
172.16.100.21:8262  dm-worker           172.16.100.21  8262       linux/x86_64  Free
    ↪ /home/tidb/dm/data/dm-worker-8262     /home/tidb/dm/deploy/dm-worker-8262
172.16.100.21:3000  grafana             172.16.100.21  3000       linux/x86_64  Up
    ↪ -                                     /home/tidb/dm/deploy/grafana-3000
172.16.100.21:9090  prometheus          172.16.100.21  9090       linux/x86_64  Up
    ↪ /home/tidb/dm/data/prometheus-9090    /home/tidb/dm/deploy/prometheus-9090
Total nodes: 5
```

   将补丁应用到指定节点或指定角色，若同时使用 -R 和 -N，将会取其交集。

```
# 为指定节点应用补丁
tiup dm patch dm-test dm-master-hotfix-linux-amd64.tar.gz -N 172.16.100.21:8261
tiup dm patch dm-test dm-worker-hotfix-linux-amd64.tar.gz -N 172.16.100.21:8262

# 为指定角色应用补丁
```

```
tiup dm patch dm-test dm-master-hotfix-linux-amd64.tar.gz -R dm-master
tiup dm patch dm-test dm-worker-hotfix-linux-amd64.tar.gz -R dm-worker
```

4. 查看补丁应用结果：

```
/home/tidb/dm/deploy/dm-master-8261/bin/dm-master/dm-master -V
```

输出示例：

```
Release Version: v5.3.0-20211230
Git Commit Hash: ca7070c45013c24d34bd9c1e936071253451d707
Git Branch: heads/refs/tags/v5.3.0-20211230
UTC Build Time: 2022-01-05 14:19:02
Go Version: go version go1.16.4 linux/amd64
```

集群信息也会有所变化：

```
tiup dm display dm-test
```

输出示例：

```
Starting component `dm`: /root/.tiup/components/dm/v1.8.1/tiup-dm display dm-test
Cluster type:       dm
Cluster name:       dm-test
Cluster version:    v5.3.0
Deploy user:        tidb
SSH type:           builtin
ID                  Role                 Host           Ports      OS/Arch       Status
    ↪ Data Dir                              Deploy Dir
--                  ----                 ----           -----      -------       ------
    ↪ --------                              ----------
172.16.100.21:9093  alertmanager         172.16.100.21  9093/9094  linux/x86_64  Up
    ↪ /home/tidb/dm/data/alertmanager-9093  /home/tidb/dm/deploy/alertmanager-9093
172.16.100.21:8261  dm-master (patched)  172.16.100.21  8261/8291  linux/x86_64  Healthy|L
    ↪ /home/tidb/dm/data/dm-master-8261     /home/tidb/dm/deploy/dm-master-8261
172.16.100.21:8262  dm-worker (patched)  172.16.100.21  8262       linux/x86_64  Free
    ↪ /home/tidb/dm/data/dm-worker-8262     /home/tidb/dm/deploy/dm-worker-8262
172.16.100.21:3000  grafana              172.16.100.21  3000       linux/x86_64  Up
    ↪ -                                     /home/tidb/dm/deploy/grafana-3000
172.16.100.21:9090  prometheus           172.16.100.21  9090       linux/x86_64  Up
    ↪ /home/tidb/dm/data/prometheus-9090    /home/tidb/dm/deploy/prometheus-9090
Total nodes: 5
```

<< 返回上一页 - TiUP DM 命令清单

13.4.7.4.13 tiup dm prune

在缩容集群后，etcd 中仍然会有少量元信息不会被清理，通常不会有问题，如果确实需要清理，可以手动执行 `tiup dm prune` 命令清理。

语法

```
tiup dm prune <cluster-name> [flags]
```

选项

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

清理过程的日志。

<< 返回上一页 - TiUP DM 命令清单

13.4.7.4.14 tiup dm reload

在修改集群配置之后，需要通过 `tiup dm reload` 命令让集群重新加载配置才会生效，该命令会将中控机的配置发布到服务运行的远端机器，并按照升级的流程按顺序重启服务，重启过程中集群仍然可用。

语法

```
tiup dm reload <cluster-name> [flags]
```

`<cluster-name>` 代表要操作的集群名。

选项

-N, –node（strings，默认为 []，表示所有节点）

指定要重启的节点，不指定则表示所有节点。该选项的值为以逗号分割的节点 ID 列表，节点 ID 为集群状态表格的第一列。

> 注意：
>
> - 若同时指定了 `-R, --role`，那么将重启它们的交集中的服务
> - 若指定了选项 `--skip-restart`，则该选项无效

-R, –role（strings，默认为 []，表示所有角色）

指定要重启的角色，不指定则表示所有角色。该选项的值为以逗号分割的节点角色列表，角色为集群状态表格的第二列。

注意：

- 若同时指定了 `-N, --node`，那么将重启它们的交集中的服务
- 若指定了选项 `--skip-restart`，则该选项无效

–skip-restart

- 命令 `tiup dm reload` 会执行两个操作：
  - 刷新所有节点配置
  - 重启指定节点
- 该选项指定后仅刷新配置，不重启任何节点，这样刷新的配置也不会应用，需要等对应服务下次重启才会生效。
- 数据类型：`BOOLEAN`
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

-h, –help

- 输出帮助信息。
- 数据类型：`BOOLEAN`
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

tiup-dm 的执行日志。

<< 返回上一页 - TiUP DM 命令清单

13.4.7.4.15 tiup dm replay

对集群进行升级或重启等操作时，操作有可能因为环境的原因而偶然失败。这时如果重新进行操作，需要从头开始执行所有步骤。如果集群规模较大，会耗费较长时间。此时可以使用 `tiup dm replay` 命令，命令重试刚才失败的命令，并且跳过已经成功的步骤。

语法

```
tiup dm replay <audit-id> [flags]
```

- `<audit-id>` 代表要重试的命令对应的 `audit-id`。使用`tiup dm audit` 可查看历史命令及其 `audit-id`。

选项

-h, –help

输出帮助信息。

输出

<audit-id> 对应的命令的输出。

<< 返回上一页 - TiUP DM 命令清单

13.4.7.4.16 tiup dm restart

命令 tiup dm restart 用于重启指定集群的所有或部分服务。

> 注意：
>
> 重启过程中会有一段时间服务不可用。

语法

```
tiup dm restart <cluster-name> [flags]
```

<cluster-name> 为要操作的集群名字，如果忘记集群名字可查看集群列表。

选项

-N, –node（strings，默认为 []，表示所有节点）

指定要重启的节点，不指定则表示所有节点。该选项的值为以逗号分割的节点 ID 列表，节点 ID 为集群状态表格的第一列。

> 注意：
>
> 若同时指定了 -R, --role，那么将重启它们的交集中的服务。

-R, –role（strings，默认为 []，表示所有角色）

指定要重启的角色，不指定则表示所有角色。该选项的值为以逗号分割的节点角色列表，角色为集群状态表格的第二列。

> 注意：
>
> 若同时指定了 -N, --node，那么将重启它们的交集中的服务。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

重启服务的日志。

<< 返回上一页 - TiUP DM 命令清单

13.4.7.4.17 tiup dm scale-in

`tiup dm scale-in` 命令用于集群缩容，缩容即下线服务，最终会将指定的节点从集群中移除，并删除遗留的相关文件。

语法

```
tiup dm scale-in <cluster-name> [flags]
```

`<cluster-name>` 为要操作的集群名字，如果忘记集群名字可查看集群列表。

选项

-N, –node（strings，无默认值，必须非空）

选择要缩容的节点，若缩容多个节点，以逗号分割。

–force

- 在某些情况下，有可能被缩容的节点宿主机已经宕机，导致无法通过 SSH 连接到节点进行操作，这个时候可以通过 `--force` 选项强制将其从集群中移除。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

缩容日志

<< 返回上一页 - TiUP DM 命令清单

13.4.7.4.18 tiup dm scale-out

`tiup dm scale-out` 命令用于集群扩容，扩容的内部逻辑与部署类似，tiup-dm 组件会先建立新节点的 SSH 连接，在目标节点上创建必要的目录，然后执行部署并且启动服务。

语法

```
tiup dm scale-out <cluster-name> <topology.yaml> [flags]
```

- `<cluster-name>` 为要操作的集群名字，如果忘记集群名字可查看集群列表
- `<topology.yaml>` 为事先编写好的扩容拓扑文件，该文件应当仅包含扩容部分的拓扑

选项

-u, –user（string，默认为当前执行命令的用户）

指定连接目标机器的用户名，该用户在目标机器上需要有免密 sudo root 的权限。

-i, –identity_file（string，默认 ~/.ssh/id_rsa）

指定连接目标机器的密钥文件。

-p, –password

- 在连接目标机器时使用密码登录，不可和 `-i/--identity_file` 同时使用。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

扩容日志

<< 返回上一页 - TiUP DM 命令清单

13.4.7.4.19 tiup dm start

命令 `tiup dm start` 用于启动指定集群的所有或部分服务。

语法

```
tiup dm start <cluster-name> [flags]
```

`<cluster-name>` 为要操作的集群名字，如果忘记集群名字可查看集群列表。

选项

-N, –node（strings，默认为 []，表示所有节点）

指定要启动的节点，不指定则表示所有节点。该选项的值为以逗号分割的节点 ID 列表，节点 ID 为集群状态表格的第一列。

> 注意：
>
> 若同时指定了 `-R, --role`，那么将启动它们的交集中的服务。

-R, –role（strings，默认为 []，表示所有角色）

指定要启动的角色，不指定则表示所有角色。该选项的值为以逗号分割的节点角色列表，角色为集群状态表格的第二列。

> 注意：
>
> 若同时指定了 `-N, --node`，那么将启动它们的交集中的服务。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

启动日志。

<< 返回上一页 - TiUP DM 命令清单

13.4.7.4.20 tiup dm stop

命令 `tiup dm stop` 用于停止指定集群的所有服务或部分服务。

> 注意：
>
> 核心服务停止后集群将无法提供服务。

语法

```
tiup dm stop <cluster-name> [flags]
```

<cluster-name> 为要操作的集群名字，如果忘记集群名字可查看集群列表。

选项

-N, –node（strings，默认为 []，表示所有节点）

指定要停止的节点，不指定则表示所有节点。该选项的值为以逗号分割的节点 ID 列表，节点 ID 为集群状态表格的第一列。

> 注意：
>
> 若同时指定了 -R, --role，那么将停止它们的交集中的服务。

-R, –role（strings，默认为 []，表示所有角色）

指定要停止的角色，不指定则表示所有角色。该选项的值为以逗号分割的节点角色列表，角色为集群状态表格的第二列。

> 注意：
>
> 若同时指定了 -N, --node，那么将停止它们的交集中的服务。

-h, –help

- 输出帮助信息。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 false。在命令中添加该选项，并传入 true 值或不传值，均可开启此功能。

输出

停止服务的日志。

<< 返回上一页 - TiUP DM 命令清单

13.4.7.4.21 tiup dm template

部署集群之前，需要准备一份集群的拓扑文件。TiUP 内置了拓扑文件的模版，用户可以通过修改模版来生成最终的拓扑文件。使用 tiup dm template 命令可以输出 TiUP 内置的模版内容。

语法

```
tiup dm template [flags]
```

如果不指定该选项，输出的默认模版包含以下实例：

- 3 个 DM-master 实例

- 3 个 DM-worker 实例
- 1 个 Prometheus 实例
- 1 个 Grafana 实例
- 1 个 Alertmanager 实例

选项

–full

- 输出详细的拓扑模版，该模版会以注释的形式带上可配置的参数。在命令中添加该选项，可开启该选项。
- 如果不指定该选项，默认输出最简单的拓扑模版。

-h, –help

输出帮助信息。

输出

根据指定选项输出拓扑模版到标准输出，可重定向到拓扑文件中用于部署。

<< 返回上一页 - TiUP DM 命令清单

13.4.7.4.22 tiup dm upgrade

命令 `tiup dm upgrade` 用于将指定集群升级到特定版本。

语法

```
tiup dm upgrade <cluster-name> <version> [flags]
```

- `<cluster-name>` 为要操作的集群名字，如果忘记集群名字可查看集群列表。
- `<version>` 为要升级到的目标版本，例如 `v6.5.0`。目前仅允许升级到比当前集群更高的版本，不允许升级到比当前集群更低的版本，即不允许降级。同时也不允许升级成 nightly 版本

选项

–offline

声明当前集群处于离线状态。指定该选项时，TiUP DM 仅原地替换集群组件的二进制文件，不重启服务。

-h, –help

- 输出帮助信息。
- 数据类型：`BOOLEAN`
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

输出

升级服务的日志。

<< 返回上一页 - TiUP DM 命令清单

13.4.8 通过 TiUP 部署 TiDB 集群的拓扑文件配置

通过 TiUP 部署或扩容 TiDB 集群时，需要提供一份拓扑文件（示例）来描述集群拓扑。

同样，修改集群配置也是通过编辑拓扑文件来实现的，区别在于修改配置时仅允许修改部分字段。本文档介绍拓扑文件的各个区块以各区块中的各字段。

13.4.8.1 文件结构

一个通过 TiUP 部署的 TiDB 集群拓扑文件可能包含以下区块：

- global：集群全局配置，其中一些是集群的默认值，可以在实例里面单独配置
- monitored：监控服务配置，即 blackbox exporter 和 node exporter，每台机器上都会部署一个 node exporter 和一个 blackbox exporter
- server_configs：组件全局配置，可单独针对每个组件配置，若在实例中存在同名配置项，那么以实例中配置的为准
- pd_servers：PD 实例的配置，用来指定 PD 组件部署到哪些机器上
- tidb_servers：TiDB 实例的配置，用来指定 TiDB 组件部署到哪些机器上
- tikv_servers：TiKV 实例的配置，用来指定 TiKV 组件部署到哪些机器上
- tiflash_servers：TiFlash 实例的配置，用来指定 TiFlash 组件部署到哪些机器上
- pump_servers：Pump 实例的配置，用来指定 Pump 组件部署到哪些机器上
- drainer_servers：Drainer 实例的配置，用来指定 Drainer 组件部署到哪些机器上
- cdc_servers：CDC 实例的配置，用来指定 CDC 组件部署到哪些机器上
- tispark_masters：TiSpark Master 实例的配置，用来指定 TiSpark Master 组件部署到哪台机器上，仅允许部署一个 TiSpark Master 节点
- tispark_workers：TiSpark Worker 实例的配置，用来指定 TiSpark Worker 组件部署到哪些机器上
- monitoring_servers：用来指定 Prometheus 和 NGMonitoring 部署在哪些机器上，TiUP 支持部署多台 Prometheus 实例，但真实投入使用的只有第一个
- grafana_servers：Grafana 实例的配置，用来指定 Grafana 部署在哪台机器上
- alertmanager_servers：Alertemanager 实例的配置，用来指定 Alertmanager 部署在哪些机器上

13.4.8.1.1 global

`global` 区块为集群的全局配置，包含以下字段：

- `user`：以什么用户来启动部署的集群，默认值：“tidb”，如果 `<user>` 字段指定的用户在目标机器上不存在，会自动尝试创建
- `group`：自动创建用户时指定用户所属的用户组，默认和 `<user>` 字段值相同，若指定的组不存在，则自动创建
- `ssh_port`：指定连接目标机器进行操作的时候使用的 SSH 端口，默认值：22
- `enable_tls`：是否对集群启用 TLS。启用之后，组件之间、客户端与组件之间都必须使用生成的 TLS 证书进行连接，默认值：false
- `deploy_dir`：每个组件的部署目录，默认值：“deploy”。其应用规则如下：
  - 如果在实例级别配置了绝对路径的 `deploy_dir`，那么实际部署目录为该实例设定的 `deploy_dir`
  - 对于每个实例，如果用户未配置 `deploy_dir`，其默认值为相对路径 `<component-name>-<component-port>`

  - 如果 global.deploy_dir 为绝对路径，那么组件会部署到 <global.deploy_dir>/<instance.deploy_dir> 目录
  - 如果 global.deploy_dir 为相对路径，那么组件会部署到 /home/<global.user>/<global.deploy_dir>/<instance.deploy_dir> 目录

- data_dir：数据目录，默认值："data"。其应用规则如下：
  - 如果在实例级别配置了绝对路径的 data_dir，那么实际数据目录为该实例设定的 data_dir
  - 对于每个实例，如果用户未配置 data_dir，其默认值为 <global.data_dir>
  - 如果 data_dir 为相对路径，那么组件数据将放到 <deploy_dir>/<data_dir> 中，其中 <deploy_dir> 的计算规则请参考 deploy_dir 字段的应用规则
- log_dir：日志目录，默认值："log"。其应用规则如下：
  - 如果在实例级别配置了绝对路径的 log_dir，那么实际日志目录为该实例设定的 log_dir
  - 对于每个实例，如果用户未配置 log_dir，其默认值为 <global.log_dir>
  - 如果 log_dir 为相对路径，那么组件日志将放到 <deploy_dir>/<log_dir> 中，其中 <deploy_dir> 的计算规则请参考 deploy_dir 字段的应用规则
- os：目标机器的操作系统，该字段决定了向目标机器推送适配哪个操作系统的组件，默认值：linux
- arch：目标机器的 CPU 架构，该字段决定了向目标机器推送哪个平台的二进制包，支持 amd64 和 arm64，默认值：amd64
- resource_control：运行时资源控制，该字段下所有配置都将写入 systemd 的 service 文件中，默认无限制。支持控制的资源如下：
  - memory_limit: 限制运行时最大内存，例如 "2G" 表示最多使用 2GB 内存
  - cpu_quota：限制运行时最大 CPU 占用率，例如 "200%"
  - io_read_bandwidth_max：读磁盘 I/O 的最大带宽，例如："/dev/disk/by-path/pci-0000:00:1f.2-scsi-0:0:0:0 100M"
  - io_write_bandwidth_max：写磁盘 I/O 的最大带宽，例如："/dev/disk/by-path/pci-0000:00:1f.2-scsi-0:0:0:0 100M"
  - limit_core：控制 core dump 的大小

global 配置示例：

```
global:
  user: "tidb"
  resource_control:
    memory_limit: "2G"
```

上述配置指定使用 tidb 用户启动集群，同时限制每个组件运行时最多只能使用 2GB 内存。

13.4.8.1.2 monitored

monitored 用于配置目标机上的监控服务：node_exporter 和 blackbox_exporter。包含以下字段：

- node_exporter_port：node_exporter 的服务端口，默认值：9100
- blackbox_exporter_port：blackbox_exporter 的服务端口，默认值：9115
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成

- data_dir：指定数据目录，若不指定，或指定为相对目录，则按照 global 中配置的 data_dir 生成
- log_dir：指定日志目录，若不指定，或指定为相对目录，则按照 global 中配置的 log_dir 生成

monitored 配置示例：

```
monitored:
  node_exporter_port: 9100
  blackbox_exporter_port: 9115
```

上述配置指定了 node_exporter 使用 9100 端口，blackbox_exporter 使用 9115 端口。

13.4.8.1.3 server_configs

server_configs 用于配置服务，生成各组件的配置文件，类似 global 区块，该区块内的配置可以在具体的实例中被覆盖。主要包含以下字段：

- tidb：TiDB 服务的相关配置，支持的完整配置请参考TiDB 配置文件描述
- tikv：TiKV 服务的相关配置，支持的完整配置请参考TiKV 配置文件描述
- pd：PD 服务的相关配置，支持的完整配置请参考PD 配置文件描述
- tiflash：TiFlash 服务的相关配置，支持的完整配置请参考TiFlash 配置参数
- tiflash_learner：每个 TiFlash 中内置了一个特殊的 TiKV，该配置项用于配置这个特殊的 TiKV，一般不建议修改这个配置项下的内容
- pump：Pump 服务的相关配置，支持的完整配置请参考TiDB Binlog 配置说明
- drainer：Drainer 服务的相关配置，支持的完整配置请参考TiDB Binlog 配置说明
- cdc：CDC 服务的相关配置，支持的完整配置请参考TiCDC 安装部署

server_configs 配置示例：

```
server_configs:
  tidb:
    lease: "45s"
    split-table: true
    token-limit: 1000
    instance.tidb_enable_ddl: true
  tikv:
    log-level: "info"
    readpool.unified.min-thread-count: 1
```

上述配置指定了 TiDB 和 TiKV 的全局配置。

13.4.8.1.4 pd_servers

pd_servers 指定了将 PD 的服务部署到哪些机器上，同时可以指定每台机器上的服务配置。pd_servers 是一个数组，每个数组的元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略

- listen_host：当机器上有多个 IP 时，可以指定服务的监听 IP，默认为 0.0.0.0
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- name：指定该 PD 实例的名字，不同实例的名字必须唯一，否则无法部署
- client_port：指定 PD 的客户端连接端口，默认值：2379
- peer_port：指定 PD 之间互相通信的端口，默认值：2380
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- data_dir：指定数据目录，若不指定，或指定为相对目录，则按照 global 中配置的 data_dir 生成
- log_dir：指定日志目录，若不指定，或指定为相对目录，则按照 global 中配置的 log_dir 生成
- numa_node：为该实例分配 NUMA 策略，如果指定了该参数，需要确保目标机装了 numactl。在指定该参数的情况下会通过 numactl 分配 cpubind 和 membind 策略。该字段参数为 string 类型，字段值填 NUMA 节点的 ID，例如 “0,1”
- config：该字段配置规则和 server_configs 里的 pd 配置规则相同，若配置了该字段，会将该字段内容和 server_configs 里的 pd 内容合并（若字段重叠，以本字段内容为准），然后生成配置文件并下发到 host 指定的机器
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- resource_control：针对该服务的资源控制。如果配置了该字段，会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global 中的 resource_control

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- listen_host
- name
- client_port
- peer_port
- deploy_dir
- data_dir
- log_dir
- arch
- os

pd_servers 配置示例：

```
pd_servers:
  - host: 10.0.1.11
    config:
      schedule.max-merge-region-size: 20
      schedule.max-merge-region-keys: 200000
  - host: 10.0.1.12
```

上述配置指定了将 PD 部署到 10.0.1.11 和 10.0.1.12，并针对 10.0.1.11 的 PD 进行一些特殊配置。

13.4.8.1.5 tidb_servers

tidb_servers 指定了将 TiDB 服务部署到哪些机器上，同时可以指定每台机器上的服务配置。tidb_servers 是一个数组，每个数组的元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- listen_host：当机器上有多个 IP 时，可以指定服务的监听 IP，默认为 0.0.0.0
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- port：TiDB 服务的监听端口，用于提供给 MySQL 客户端连接，默认值：4000
- status_port：TiDB 状态服务的监听端口，用于外部通过 http 请求查看 TiDB 服务的状态，默认值：10080
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- log_dir：指定日志目录，若不指定，或指定为相对目录，则按照 global 中配置的 log_dir 生成
- numa_node：为该实例分配 NUMA 策略，如果指定了该参数，需要确保目标机装了 numactl，在指定该参数的情况下会通过 numactl 分配 cpubind 和 membind 策略。该字段参数为 string 类型，字段值填 NUMA 节点的 ID，例如 “0,1”
- config：该字段配置规则和 server_configs 里的 tidb 配置规则相同，若配置了该字段，会将该字段内容和 server_configs 里的 tidb 内容合并（若字段重叠，以本字段内容为准），然后生成配置文件并下发到 host 指定的机器
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- resource_control：针对该服务的资源控制，如果配置了该字段，会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global 中的 resource_control

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- listen_host
- port
- status_port
- deploy_dir
- log_dir
- arch
- os

tidb_servers 配置示例：

```
tidb_servers:
  - host: 10.0.1.14
    config:
      log.level: warn
      log.slow-query-file: tidb-slow-overwrited.log
  - host: 10.0.1.15
```

13.4.8.1.6 tikv_servers

tikv_servers 约定了将 TiKV 服务部署到哪些机器上，同时可以指定每台机器上的服务配置。tikv_servers 是一个数组，每个数组元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- listen_host：当机器上有多个 IP 时，可以指定服务的监听 IP，默认为 0.0.0.0
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- port：TiKV 服务的监听端口，默认值：20160
- status_port：TiKV 状态服务的监听端口，默认值：20180
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- data_dir：指定数据目录，若不指定，或指定为相对目录，则按照 global 中配置的 data_dir 生成
- log_dir：指定日志目录，若不指定，或指定为相对目录，则按照 global 中配置的 log_dir 生成
- numa_node：为该实例分配 NUMA 策略，如果指定了该参数，需要确保目标机装了 numactl，在指定该参数的情况下会通过 numactl 分配 cpubind 和 membind 策略。该字段参数为 string 类型，字段值填 NUMA 节点的 ID，例如 “0,1”
- config：该字段配置规则和 server_configs 里的 tikv 配置规则相同，若配置了该字段，会将该字段内容和 server_configs 里的 tikv 内容合并（若字段重叠，以本字段内容为准），然后生成配置文件并下发到 host 指定的机器
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- resource_control：针对该服务的资源控制，如果配置了该字段，会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global 中的 resource_control

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- listen_host
- port
- status_port
- deploy_dir
- data_dir
- log_dir
- arch
- os

tikv_servers 配置示例：

```
tikv_servers:
  - host: 10.0.1.14
    config:
      server.labels: { zone: "zone1", host: "host1" }
  - host: 10.0.1.15
    config:
      server.labels: { zone: "zone1", host: "host2" }
```

13.4.8.1.7 tiflash_servers

tiflash_servers 约定了将 TiFlash 服务部署到哪些机器上，同时可以指定每台机器上的服务配置。该区块是一个数组，每个数组元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- tcp_port：TiFlash TCP 服务的端口，默认 9000
- http_port：TiFlash HTTP 服务的端口，默认 8123
- flash_service_port：TiFlash 提供服务的端口，TiDB 通过该端口从 TiFlash 读数据，默认 3930
- metrics_port：TiFlash 的状态端口，用于输出 metric 数据，默认 8234
- flash_proxy_port：内置 TiKV 的端口，默认 20170
- flash_proxy_status_port：内置 TiKV 的状态端口，默认为 20292
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- data_dir：指定数据目录，若不指定，或指定为相对目录，则按照 global 中配置的 data_dir 生成，TiFlash 的数据目录支持多个，采用逗号分割
- log_dir：指定日志目录，若不指定，或指定为相对目录，则按照 global 中配置的 log_dir 生成
- tmp_path: TiFlash 临时文件的存放路径，默认使用 [path 或者 storage.latest.dir 的第一个目录] + “/tmp”
- numa_node：为该实例分配 NUMA 策略，如果指定了该参数，需要确保目标机装了 numactl，在指定该参数的情况下会通过 numactl 分配 cpubind 和 membind 策略。该字段参数为 string 类型，字段值填 NUMA 节点的 ID，例如 “0,1”
- config：该字段配置规则和 server_configs 里的 tiflash 配置规则相同，若配置了该字段，会将该字段内容和 server_configs 里的 tiflash 内容合并（若字段重叠，以本字段内容为准），然后生成配置文件并下发到 host 指定的机器
- learner_config：每个 TiFlash 中内置了一个特殊的 TiKV 模块，该配置项用于配置这个特殊的 TiKV 模块，一般不建议修改这个配置项下的内容
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- resource_control：针对该服务的资源控制，如果配置了该字段，会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global 中的 resource_control

以上字段中，data_dir 在部署完成之后只能新增目录，而下列字段部署完成之后不能再修改：

- host
- tcp_port
- http_port
- flash_service_port
- flash_proxy_port
- flash_proxy_status_port
- metrics_port
- deploy_dir
- log_dir
- tmp_path
- arch

- os

tiflash_servers 配置示例：

```
tiflash_servers:
  - host: 10.0.1.21
  - host: 10.0.1.22
```

13.4.8.1.8 pump_servers

pump_servers 约定了将 TiDB Binlog 组件的 Pump 服务部署到哪些机器上，同时可以指定每台机器上的服务配置。pump_servers 是一个数组，每个数组元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- port：Pump 服务的监听端口，默认 8250
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- data_dir：指定数据目录，若不指定，或指定为相对目录，则按照 global 中配置的 data_dir 生成
- log_dir：指定日志目录，若不指定，或指定为相对目录，则按照 global 中配置的 log_dir 生成
- numa_node：为该实例分配 NUMA 策略，如果指定了该参数，需要确保目标机装了 numactl，在指定该参数的情况下会通过 numactl 分配 cpubind 和 membind 策略。该字段参数为 string 类型，字段值填 NUMA 节点的 ID，例如 “0,1”
- config：该字段配置规则和 server_configs 里的 pump 配置规则相同，若配置了该字段，会将该字段内容和 server_configs 里的 pump 内容合并（若字段重叠，以本字段内容为准），然后生成配置文件并下发到 host 指定的机器
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- resource_control：针对该服务的资源控制，如果配置了该字段，会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global 中的 resource_control

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- port
- deploy_dir
- data_dir
- log_dir
- arch
- os

pump_servers 配置示例：

```
pump_servers:
  - host: 10.0.1.21
    config:
      gc: 7
  - host: 10.0.1.22
```

13.4.8.1.9 drainer_servers

drainer_servers 约定了将 TiDB Binlog 组件的 Drainer 服务部署到哪些机器上，同时可以指定每台机器上的服务配置，drainer_servers 是一个数组，每个数组元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- port：Drainer 服务的监听端口，默认 8249
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- data_dir：指定数据目录，若不指定，或指定为相对目录，则按照 global 中配置的 data_dir 生成
- log_dir：指定日志目录，若不指定，或指定为相对目录，则按照 global 中配置的 log_dir 生成
- commit_ts：[已废弃]Drainer 启动的时候会去读取 checkpoint，如果读取不到，就会使用该字段做为初次启动开始的同步时间点，该字段默认为 -1（从 PD 总获取最新时间戳作为 commit_ts）
- numa_node：为该实例分配 NUMA 策略，如果指定了该参数，需要确保目标机装了 numactl，在指定该参数的情况下会通过 numactl 分配 cpubind 和 membind 策略。该字段参数为 string 类型，字段值填 NUMA 节点的 ID，例如 “0,1”
- config：该字段配置规则和 server_configs 里的 drainer 配置规则相同，若配置了该字段，会将该字段内容和 server_configs 里的 drainer 内容合并（若字段重叠，以本字段内容为准），然后生成配置文件并下发到 host 指定的机器
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- resource_control：针对该服务的资源控制，如果配置了该字段，会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global 中的 resource_control

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- port
- deploy_dir
- data_dir
- log_dir
- arch
- os

其中 commit_ts 字段自 tiup cluster v1.9.2 开始已经废弃，不再被记录到 drainer 的启动脚本中。如果你仍需要配置该参数，请参照下面的示例在 config 中配置 initial-commit-ts 字段。

drainer_servers 配置示例：

```
drainer_servers:
  - host: 10.0.1.21
    config:
      initial-commit-ts: -1
      syncer.db-type: "mysql"
      syncer.to.host: "127.0.0.1"
      syncer.to.user: "root"
      syncer.to.password: ""
      syncer.to.port: 3306
      syncer.ignore-table:
        - db-name: test
          tbl-name: log
        - db-name: test
          tbl-name: audit
```

13.4.8.1.10 cdc_servers

cdc_servers 约定了将 TiCDC 服务部署到哪些机器上，同时可以指定每台机器上的服务配置，cdc_servers 是一个数组，每个数组元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- port：TiCDC 服务的监听端口，默认 8300
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- data_dir：指定数据目录。若不指定该字段或指定为相对目录，数据目录则按照 global 中配置的 data_dir 生成。
- log_dir：指定日志目录，若不指定，或指定为相对目录，则按照 global 中配置的 log_dir 生成
- gc-ttl：TiCDC 在 PD 设置的服务级别 GC safepoint 的 TTL (Time To Live) 时长，单位为秒，默认值为 86400，即 24 小时
- tz：TiCDC 服务使用的时区。TiCDC 在内部转换 timestamp 等时间数据类型和向下游同步数据时使用该时区，默认为进程运行本地时区。
- numa_node：为该实例分配 NUMA 策略，如果指定了该参数，需要确保目标机装了 numactl，在指定该参数的情况下会通过 numactl 分配 cpubind 和 membind 策略。该字段参数为 string 类型，字段值填 NUMA 节点的 ID，例如 “0,1”
- config：该字段配置规则和 server_configs 里的 cdc 内容合并（若字段重叠，以本字段内容为准），然后生成配置文件并下发到 host 指定的机器
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- resource_control：针对该服务的资源控制，如果配置了该字段，会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global 中的 resource_control
- ticdc_cluster_id：指定该服务器对应的 TiCDC 集群 ID。若不指定该字段，则自动加入默认 TiCDC 集群。该配置只在 v6.3.0 及以上 TiDB 版本中才生效。

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- port
- deploy_dir
- data_dir
- log_dir
- arch
- os
- ticdc_cluster_id

cdc_servers 配置示例：

```
cdc_servers:
  - host: 10.0.1.20
    gc-ttl: 86400
    data_dir: "/cdc-data"
  - host: 10.0.1.21
    gc-ttl: 86400
    data_dir: "/cdc-data"
```

13.4.8.1.11 tispark_masters

tispark_masters 约定了将 TiSpark 的 master 节点部署到哪些机器上，同时可以指定每台机器上的服务配置，tispark_masters 是一个数组，每个数组元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- listen_host：当机器上有多个 IP 时，可以指定服务的监听 IP，默认为 0.0.0.0
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- port：Spark 的监听端口，节点之前通讯用，默认 7077
- web_port：Spark 的网页端口，提供网页服务，可查看任务情况，默认 8080
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- java_home：指定要使用的 JRE 环境所在的路径。该参数对应系统环境变量 JAVA_HOME
- spark_config：用于配置 TiSpark 服务，生成配置文件并下发到 host 指定的机器
- spark_env：配置 Spark 启动时的环境变量
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- listen_host
- port
- web_port

- deploy_dir
- arch
- os

tispark_masters 配置示例：

```
tispark_masters:
  - host: 10.0.1.21
    spark_config:
      spark.driver.memory: "2g"
      spark.eventLog.enabled: "False"
      spark.tispark.grpc.framesize: 2147483647
      spark.tispark.grpc.timeout_in_sec: 100
      spark.tispark.meta.reload_period_in_sec: 60
      spark.tispark.request.command.priority: "Low"
      spark.tispark.table.scan_concurrency: 256
    spark_env:
      SPARK_EXECUTOR_CORES: 5
      SPARK_EXECUTOR_MEMORY: "10g"
      SPARK_WORKER_CORES: 5
      SPARK_WORKER_MEMORY: "10g"
  - host: 10.0.1.22
```

13.4.8.1.12 tispark_workers

tispark_workers 约定了将 TiSpark 的 worker 节点部署到哪些机器上，同时可以指定每台机器上的服务配置，tispark_workers 是一个数组，每个数组元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- listen_host：当机器上有多个 IP 时，可以指定服务的监听 IP，默认为 0.0.0.0
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- port：Spark 的监听端口，节点之前通讯用，默认 7077
- web_port：Spark 的网页端口，提供网页服务，可查看任务情况，默认 8080
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- java_home：指定要使用的 JRE 环境所在的路径。该参数对应系统环境变量 JAVA_HOME
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- listen_host
- port
- web_port

- deploy_dir
- arch
- os

tispark_workers 配置示例：

```
tispark_workers:
  - host: 10.0.1.22
  - host: 10.0.1.23
```

13.4.8.1.13 monitoring_servers

monitoring_servers 约定了将 Prometheus 服务部署到哪台机器上，同时可以指定这台机器上的服务配置，monitoring_servers 是一个数组，每个数组元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，必填项
- ng_port：指定 NgMonitoring 组件监听的端口，在 TiUP v1.7.0 引入，用于支持 TiDB Dashboard 中持续性能分析和Top SQL 功能。默认值：12020
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- port：指定 Prometheus 提供服务的端口，默认值：9090
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- data_dir：指定数据目录，若不指定，或指定为相对目录，则按照 global 中配置的 data_dir 生成
- log_dir：指定日志目录，若不指定，或指定为相对目录，则按照 global 中配置的 log_dir 生成
- numa_node：为该实例分配 NUMA 策略，如果指定了该参数，需要确保目标机装了 numactl，在指定该参数的情况下会通过 numactl 分配 cpubind 和 membind 策略。该字段参数为 string 类型，字段值填 NUMA 节点的 ID，例如 “0,1”
- storage_retention：Prometheus 监控数据保留时间，默认 “30d”
- rule_dir：该字段指定一个本地目录，该目录中应当含有完整的 *.rules.yml 文件，这些文件会在集群配置初始化阶段被传输到目标机器上，作为 Prometheus 的规则
- remote_config：用于支持将 Prometheus 数据写到远端，或从远端读取数据，该字段下有两个配置：
  - remote_write：参考 Prometheus <remote_write> 文档
  - remote_read：参考 Prometheus <remote_read> 文档
- external_alertmanagers：若配置了 external_alertmanagers，Prometheus 会将配置行为报警通知到集群外的 Alertmanager。该字段为一个数组，数组的元素为每个外部的 Alertmanager，由 host 和 web_port 字段构成
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- resource_control：针对该服务的资源控制，如果配置了该字段，会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global 中的 resource_control

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host

- port
- deploy_dir
- data_dir
- log_dir
- arch
- os

monitoring_servers 配置示例：

```
monitoring_servers:
  - host: 10.0.1.11
    rule_dir: /local/rule/dir
    remote_config:
      remote_write:
      - queue_config:
          batch_send_deadline: 5m
          capacity: 100000
          max_samples_per_send: 10000
          max_shards: 300
        url: http://127.0.0.1:8003/write
      remote_read:
      - url: http://127.0.0.1:8003/read
    external_alertmanagers:
    - host: 10.1.1.1
      web_port: 9093
    - host: 10.1.1.2
      web_port: 9094
```

13.4.8.1.14 grafana_servers

grafana_servers 约定了将 Grafana 服务部署到哪台机器上，同时可以指定这台机器上的服务配置，grafana_servers 是一个数组，每个数组元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- port：指定 Grafana 提供服务的端口，默认值：3000
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- username：Grafana 登录界面的用户名
- password：Grafana 对应的密码
- dashboard_dir：该字段指定一个本地目录，该目录中应当含有完整的 dashboard(*.json) 文件，这些文件会在集群配置初始化阶段被传输到目标机器上，作为 Grafana 的 dashboards

- resource_control：针对该服务的资源控制，如果配置了该字段，会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global 中的 resource_control

> 注意：
>
> 如果配置了 grafana_servers 的 dashboard_dir 字段，在执行 tiup cluster rename 命令进行集群重命名后，需要完成以下操作：
>
> 1. 对于本地的 dashboards 目录中的 *.json 文件，将 datasource 字段的值更新为新的集群名（这是因为 datasource 是以集群名命名的）
> 2. 执行 tiup cluster reload -R grafana 命令

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- port
- deploy_dir
- arch
- os

grafana_servers 配置示例：

```
grafana_servers:
  - host: 10.0.1.11
    dashboard_dir: /local/dashboard/dir
```

13.4.8.1.15 alertmanager_servers

alertmanager_servers 约定了将 Alertmanager 服务部署到哪些机器上，同时可以指定这台机器上的服务配置，alertmanager_servers 是一个数组，每个数组元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- web_port：指定 Alertmanager 提供网页服务的端口，默认值：9093
- cluster_port：指定 Alertmanger 和其他 Alertmanager 通讯的端口，默认值：9094
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- data_dir：指定数据目录，若不指定，或指定为相对目录，则按照 global 中配置的 data_dir 生成
- log_dir：指定日志目录，若不指定，或指定为相对目录，则按照 global 中配置的 log_dir 生成
- numa_node：为该实例分配 NUMA 策略，如果指定了该参数，需要确保目标机装了 numactl，在指定该参数的情况下会通过 numactl 分配 cpubind 和 membind 策略。该字段参数为 string 类型，字段值填 NUMA 节点的 ID，例如 “0,1”

- config_file：该字段指定一个本地文件，该文件会在集群配置初始化阶段被传输到目标机器上，作为 Alertmanager 的配置
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- resource_control：针对该服务的资源控制，如果配置了该字段，会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global 中的 resource_control

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- web_port
- cluster_port
- deploy_dir
- data_dir
- log_dir
- arch
- os

alertmanager_servers 配置示例：

```
alertmanager_servers:
  - host: 10.0.1.11
    config_file: /local/config/file
  - host: 10.0.1.12
    config_file: /local/config/file
```

13.4.9 通过 TiUP 部署 DM 集群的拓扑文件配置

在部署或扩容 TiDB Data Migration (DM) 集群时，需要提供一份拓扑文件来描述集群拓扑，同样，修改配置也是通过编辑拓扑文件来实现的，区别在于修改配置时仅允许修改部分字段。

拓扑文件示例参考。

13.4.9.1 文件结构

一个 DM 集群的拓扑文件可能包含以下区块：

- global：集群全局配置，其中一些是集群的默认值，可以在实例里面单独配置
- server_configs：组件全局配置，可单独针对每个组件配置，若在实例中存在同名配置项，那么以实例中配置的为准
- master_servers：DM master 实例的配置，用来指定 DM 组件的 master 服务部署到哪些机器上
- worker_servers：DM worker 实例的配置，用来指定 DM 组件的 worker 服务部署到哪些机器上
- monitoring_servers：用来指定 Prometheus 部署在哪机器上，TiUP 支持部署多台 Prometheus 实例，但真实投入使用的只有第一个
- grafana_servers：Grafana 实例的配置，用来指定 Grafana 部署在哪台机器上
- alertmanager_servers：Alertemanager 实例的配置，用来指定 Alertmanager 部署在哪些机器上

13.4.9.1.1 global

global 区块为集群的全局配置，包含以下字段：

- user：以什么用户来启动部署的集群，默认值：“tidb”，如果 <user> 字段指定的用户在目标机器上不存在，会自动尝试创建
- group：自动创建用户时指定用户所属的用户组，默认和 <user> 字段值相同，若指定的组不存在，则自动创建
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，默认值：22
- deploy_dir：每个组件的部署目录，默认值：“deploy”，其应用规则如下：
  - 如果在实例级别配置了绝对路径的 deploy_dir，那么实际部署目录为该实例设定的 deploy_dir
  - 对于每个实例，如果用户未配置 deploy_dir，其默认值为相对路径 <component-name>-<component-port>
  - 如果 global.deploy_dir 为绝对路径，那么组件会部署到 <global.deploy_dir>/<instance.deploy_dir> 目录
  - 如果 global.deploy_dir 为相对路径，那么组件会部署到 /home/<global.user>/<global.deploy_dir>/<instance.deploy_dir> 目录
- data_dir：数据目录，默认值：“data”，其应用规则如下：
  - 如果在实例级别配置了绝对路径的 data_dir，那么实际数据目录为该实例设定的 data_dir
  - 对于每个实例，如果用户未配置 data_dir，其默认值为 <global.data_dir>
  - 如果 data_dir 为相对路径，那么组件数据将放到 <deploy_dir>/<data_dir> 中，其中 <deploy_dir> 的计算规则请参考 deploy_dir 字段的应用规则
- log_dir：数据目录，默认值：“log”，其应用规则如下：
  - 如果在实例级别配置了绝对路径的 log_dir，那么实际日志目录为该实例设定的 log_dir
  - 对于每个实例，如果用户未配置 log_dir，其默认值为 <global.log_dir>
  - 如果 log_dir 为相对路径，那么组件日志将放到 <deploy_dir>/<log_dir> 中，其中 <deploy_dir> 的计算规则请参考 deploy_dir 字段的应用规则
- os：目标机器的操作系统，该字段决定了向目标机器推送适配哪个操作系统的组件，默认值：linux
- arch：目标机器的 CPU 架构，该字段决定了向目标机器推送哪个平台的二进制包，支持 amd64 和 arm64，默认值：amd64
- resource_control：运行时资源控制，该字段下所有配置都将写入 systemd 的 service 文件中，默认不限制，支持控制的资源：
  - memory_limit: 限制运行时最大内存，例如“2G”表示最多使用 2GB 内存
  - cpu_quota：限制运行时最大 CPU 占用率，例如“200%”
  - io_read_bandwidth_max：读磁盘 IO 的最大带宽，例如：“/dev/disk/by-path/pci-0000:00:1f.2-scsi-0:0:0:0 100M”
  - io_write_bandwidth_max：写磁盘 IO 的最大带宽，例如：“/dev/disk/by-path/pci-0000:00:1f.2-scsi-0:0:0:0 100M”
  - limit_core：控制 core dump 大小

global 配置示例：

```
global:
  user: "tidb"
```

```
  resource_control:
    memory_limit: "2G"
```

上述配置指定使用 tidb 用户启动集群，同时限制每个组件运行时最多只能使用 2GB 内存。

13.4.9.1.2 server_configs

server_configs 用于配置服务，生成各组件的配置文件，类似 global 区块，该区块内的配置可以在具体的实例中被覆盖。主要包含以下字段：

- master：DM master 服务的相关配置，支持的完整配置请参考DM-master 配置文件介绍
- worker：DM worker 服务的相关配置，支持的完整配置请参考DM-worker 配置文件介绍

server_configs 配置示例：

```
server_configs:
  master:
    log-level: info
    rpc-timeout: "30s"
    rpc-rate-limit: 10.0
    rpc-rate-burst: 40
  worker:
    log-level: info
```

13.4.9.2 master_servers

master_servers 约定了将 DM 组件的 master 节点部署到哪些机器上，同时可以指定每台机器上的服务配置，master_servers 是一个数组，每个数组的元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- name：指定该 DM master 实例的名字，不同实例的名字必须唯一，否则无法部署
- port：指定 DM master 提供给服务的端口，默认 8261
- peer_port：指定 DM master 之间互相通信的端口，默认值：8291
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- data_dir：指定数据目录，若不指定，或指定为相对目录，则按照 global 中配置的 data_dir 生成
- log_dir：指定日志目录，若不指定，或指定为相对目录，则按照 global 中配置的 log_dir 生成
- numa_node：为该实例分配 NUMA 策略，如果指定了该参数，需要确保目标机装了 numactl，在指定该参数的情况下会通过 numactl 分配 cpubind 和 membind 策略。该字段参数为 string 类型，字段值填 NUMA 节点 ID，比如 “0,1”
- config：该字段配置规则和 server_configs 里的 master 配置规则相同，若配置了该字段，会将该字段内容和 server_configs 里的 master 内容合并（若字段重叠，以本字段内容为准），然后生成配置文件并下发到 host 指定的机器
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os

- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- resource_control：针对该服务的资源控制，如果配置了该字段，会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global 中的 resource_control
- v1_source_path：从 v1.0.x 升级时，可指定该目录，该目录中应当存放有 V1 的源的配置文件

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- name
- port
- peer_port
- deploy_dir
- data_dir
- log_dir
- arch
- os
- v1_source_path

master_servers 配置示例：

```
master_servers:
  - host: 10.0.1.11
    name: master1
    ssh_port: 22
    port: 8261
    peer_port: 8291
    deploy_dir: "/dm-deploy/dm-master-8261"
    data_dir: "/dm-data/dm-master-8261"
    log_dir: "/dm-deploy/dm-master-8261/log"
    numa_node: "0,1"
    # The following configs are used to overwrite the `server_configs.master` values.
    config:
      log-level: info
      rpc-timeout: "30s"
      rpc-rate-limit: 10.0
      rpc-rate-burst: 40
  - host: 10.0.1.18
    name: master2
  - host: 10.0.1.19
    name: master3
```

13.4.9.3 worker_servers

worker_servers 约定了将 DM 组件的 worker 节点部署到哪些机器上，同时可以指定每台机器上的服务配置，worker_servers 是一个数组，每个数组的元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- name：指定该 DM worker 实例的名字，不同实例的名字必须唯一，否则无法部署
- port：指定 DM worker 提供给服务的端口，默认 8262
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- data_dir：指定数据目录，若不指定，或指定为相对目录，则按照 global 中配置的 data_dir 生成
- log_dir：指定日志目录，若不指定，或指定为相对目录，则按照 global 中配置的 log_dir 生成
- numa_node：为该实例分配 NUMA 策略，如果指定了该参数，需要确保目标机装了 numactl，在指定该参数的情况下会通过 numactl 分配 cpubind 和 membind 策略。该字段参数为 string 类型，字段值填 NUMA 节点 ID，比如 “0,1”
- config：该字段配置规则和 server_configs 里的 worker 配置规则相同，若配置了该字段，会将该字段内容和 server_configs 里的 worker 内容合并（若字段重叠，以本字段内容为准），然后生成配置文件并下发到 host 指定的机器
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- resource_control：针对该服务的资源控制，如果配置了该字段，会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global 中的 resource_control

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- name
- port
- deploy_dir
- data_dir
- log_dir
- arch
- os

worker_servers 配置示例：

```
worker_servers:
  - host: 10.0.1.12
    ssh_port: 22
    port: 8262
    deploy_dir: "/dm-deploy/dm-worker-8262"
    log_dir: "/dm-deploy/dm-worker-8262/log"
    numa_node: "0,1"
    # config is used to overwrite the `server_configs.worker` values
    config:
      log-level: info
  - host: 10.0.1.19
```

13.4.9.3.1 monitoring_servers

monitoring_servers 约定了将 Prometheus 服务部署到哪台机器上，同时可以指定这台机器上的服务配置，monitoring_servers 是一个数组，每个数组元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- port：指定 Prometheus 提供服务的端口，默认值：9090
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- data_dir：指定数据目录，若不指定，或指定为相对目录，则按照 global 中配置的 data_dir 生成
- log_dir：指定日志目录，若不指定，或指定为相对目录，则按照 global 中配置的 log_dir 生成
- numa_node：为该实例分配 NUMA 策略，如果指定了该参数，需要确保目标机装了 numactl，在指定该参数的情况下会通过 numactl 分配 cpubind 和 membind 策略。该字段参数为 string 类型，字段值填 NUMA 节点 ID，比如 “0,1”
- storage_retention：Prometheus 监控数据保留时间，默认 “15d”
- rule_dir：该字段指定一个本地目录，该目录中应当含有完整的 *.rules.yml 文件，这些文件会在集群配置初始化阶段被传输到目标机器上，作为 Prometheus 的规则
- remote_config：用于支持将 Prometheus 数据写到远端，或从远端读取数据，该字段下有两个配置：
  - remote_write：参考 Prometheus <remote_write> 文档
  - remote_read：参考 Prometheus <remote_read> 文档
- external_alertmanagers：若配置了 external_alertmanagers，Prometheus 会将配置行为报警通知到集群外的 Alertmanager。该字段为一个数组，数组的元素为每个外部的 Alertmanager，由 host 和 web_port 字段构成
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- resource_control：针对该服务的资源控制，如果配置了该字段，会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global 中的 resource_control

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- port
- deploy_dir
- data_dir
- log_dir
- arch
- os

monitoring_servers 配置示例：

```
monitoring_servers:
  - host: 10.0.1.11
    rule_dir: /local/rule/dir
```

```
  remote_config:
    remote_write:
    - queue_config:
        batch_send_deadline: 5m
        capacity: 100000
        max_samples_per_send: 10000
        max_shards: 300
      url: http://127.0.0.1:8003/write
    remote_read:
    - url: http://127.0.0.1:8003/read
  external_alertmanagers:
  - host: 10.1.1.1
    web_port: 9093
  - host: 10.1.1.2
    web_port: 9094
```

13.4.9.3.2 grafana_servers

grafana_servers 约定了将 Grafana 服务部署到哪台机器上，同时可以指定这台机器上的服务配置，grafana_servers 是一个数组，每个数组元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- port：指定 Grafana 提供服务的端口，默认值：3000
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- username：Grafana 登录界面的用户名
- password：Grafana 对应的密码
- dashboard_dir：该字段指定一个本地目录，该目录中应当含有完整的 dashboard(*.json) 文件，这些文件会在集群配置初始化阶段被传输到目标机器上，作为 Grafana 的 dashboards
- resource_control：针对该服务的资源控制，如果配置了该字段，会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global中的 resource_control

> 注意：
>
> 如果配置了 grafana_servers 的 dashboard_dir 字段，在执行 tiup cluster rename 命令进行集群重命名后，需要完成以下操作：
>
> 1. 在本地的 dashboards 目录中，将 datasource 字段的值更新为新的集群名（datasource 是以集群名命名的）
> 2. 执行 tiup cluster reload -R grafana 命令

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- port
- deploy_dir
- arch
- os

grafana_servers 配置示例：

```
grafana_servers:
  - host: 10.0.1.11
    dashboard_dir: /local/dashboard/dir
```

13.4.9.3.3 alertmanager_servers

alertmanager_servers 约定了将 Alertmanager 服务部署到哪些机器上，同时可以指定这台机器上的服务配置，alertmanager_servers 是一个数组，每个数组元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- web_port：指定 Alertmanager 提供网页服务的端口，默认值：9093
- cluster_port：指定 Alertmanager 和其他 Alertmanager 通讯的端口，默认值：9094
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- data_dir：指定数据目录，若不指定，或指定为相对目录，则按照 global 中配置的 data_dir 生成
- log_dir：指定日志目录，若不指定，或指定为相对目录，则按照 global 中配置的 log_dir 生成
- numa_node：为该实例分配 NUMA 策略，如果指定了该参数，需要确保目标机装了 numactl，在指定该参数的情况下会通过 numactl 分配 cpubind 和 membind 策略。该字段参数为 string 类型，字段值填 NUMA 节点 ID，比如 “0,1”
- config_file：该字段指定一个本地文件，该文件会在集群配置初始化阶段被传输到目标机器上，作为 Alertmanager 的配置
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- resource_control：针对该服务的资源控制，如果配置了该字段，会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global 中的 resource_control

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- web_port
- cluster_port
- deploy_dir
- data_dir

- log_dir
- arch
- os

alertmanager_servers 配置示例：

```
alertmanager_servers:
  - host: 10.0.1.11
    config_file: /local/config/file
  - host: 10.0.1.12
    config_file: /local/config/file
```

13.4.10 TiUP 镜像参考指南

TiUP 镜像是 TiUP 的组件仓库，存放了一系列的组件和这些组件的元信息。镜像有两种存在形式：

- 本地磁盘上的目录：用于服务本地的 TiUP 客户端，文档中将称之为本地镜像
- 基于远程的磁盘目录启动的 HTTP 镜像：服务远程的 TiUP 客户端，文档中将称之为远程镜像

13.4.10.1 镜像的创建与更新

镜像可以通过以下两种方式创建：

- 通过命令 tiup mirror init 从零生成
- 通过命令 tiup mirror clone 从已有镜像克隆

在创建镜像之后，可以通过 tiup mirror 相关命令来给镜像添加组件或删除组件，无论是通过何种方式更新镜像，TiUP 都不会从镜像中删除任何文件，而是通过增加文件并分配新版本号的方式更新。

13.4.10.2 镜像结构

一个典型的镜像目录结构如下：

```
+ <mirror-dir>                                  # 镜像根目录
|-- root.json                                   # 镜像根证书
|-- {2..N}.root.json                            # 镜像根证书
|-- {1..N}.index.json                           # 组件/用户索引
|-- {1..N}.{component}.json                     # 组件元信息
|-- {component}-{version}-{os}-{arch}.tar.gz    # 组件二进制包
|-- snapshot.json                               # 镜像最新快照
|-- timestamp.json                              # 镜像最新时间戳
|--+ commits                                    # 镜像更新日志（可删除）
   |--+ commit-{ts1..tsN}
      |-- {N}.root.json
      |-- {N}.{component}.json
      |-- {N}.index.json
```

```
      |-- {component}-{version}-{os}-{arch}.tar.gz
      |-- snapshot.json
      |-- timestamp.json
|--+ keys                                    # 镜像私钥（可移动到其他位置）
   |-- {hash1..hashN}-root.json              # 根证书私钥
   |-- {hash}-index.json                     # 索引私钥
   |-- {hash}-snapshot.json                  # 快照私钥
   |-- {hash}-timestamp.json                 # 时间戳私钥
```

注意：

- commits 目录是在更新镜像过程中产生的日志，用于回滚镜像，磁盘空间不足时可以定期删除旧的文件夹
- keys 文件夹中存放的私钥较敏感，建议单独妥善保管

13.4.10.2.1 根证书

在 TiUP 镜像中，根证书用于存放其他元信息文件的公钥，每次获取到任何元信息文件（*.json）都需要根据其文件类型（root，index，snapshot，timestamp）在当前已安装的 root.json 中找到对应的公钥，然后用公钥验证其签名是否合法。

根证书文件格式如下：

```
{
    "signatures": [                                     # 每个元信息文件有一系列的签名，
        ↪ 签名由该文件对应的几个私钥签出
        {
            "keyid": "{id-of-root-key-1}",              # 第一个参与签名私钥的 ID，该 ID
                ↪ 由私钥对应的公钥内容哈希得到
            "sig": "{signature-by-root-key-1}"          # 该私钥对此文件 signed
                ↪ 部分签名的结果
        },
        ...
        {
            "keyid": "{id-of-root-key-N}",              # 第 N 个参与签名私钥的 ID
            "sig": "{signature-by-root-key-N}"          # 该私钥对此文件 signed
                ↪ 部分签名的结果
        }
    ],
    "signed": {                                         # 被签名的部分
        "_type": "root",                                # 该字段说明本文件的类型，root.
            ↪ json 的类型就是 root
        "expires": "{expiration-date-of-this-file}",    # 该文件的过期时间，
            ↪ 过期后客户端会拒绝此文件
```

```
        "roles": {                                          # root.json 中的 roles
            ↪ 用来记录对各个元文件签名的密钥
            "{role:index,root,snapshot,timestamp}": {       # 涉及的元文件类型包括 index,
                ↪ root, snapshot, timestamp
                "keys": {                                   # 只有 keys
                    ↪ 中记录的密钥签名才是合法的
                    "{id-of-the-key-1}": {                  # 用于签名 {role} 的第 1 个密钥
                        ↪ ID
                        "keytype": "rsa",                   # 密钥类型，目前固定为 rsa
                        "keyval": {                         # 密钥的 payload
                            "public": "{public-key-content}"    # 表示公钥内容
                        },
                        "scheme": "rsassa-pss-sha256"       # 目前固定为 rsassa-pss-sha256
                    },
                    "{id-of-the-key-N}": {                  # 用于签名 {role} 的第 N 个密钥
                        ↪ ID
                        "keytype": "rsa",
                        "keyval": {
                            "public": "{public-key-content}"
                        },
                        "scheme": "rsassa-pss-sha256"
                    }
                },
                "threshold": {N},                           # threshold 指示该元文件需要至少
                    ↪ N 个密钥签名
                "url": "/{role}.json"                       # url 是指该文件的获取地址，对于
                    ↪ index 文件，需要在前面加上版本号，即 /{N}.index.json
            }
        },
        "spec_version": "0.1.0",                            # 本文件遵循的规范版本，
            ↪ 未来变更文件结构需要升级版本号，目前为 0.1.0
        "version": {N}                                      # 本文件的版本号，
            ↪ 每次更新文件需要创建一个新的 {N+1}.root.json，并将其 version 设置为 N + 1
    }
}
```

13.4.10.2.2 索引

索引文件记录了镜像中所有的组件以及组件的所有者信息。

其格式如下：

```
{
    "signatures": [                                         # 该文件的签名
        {
            "keyid": "{id-of-index-key-1}",                 # 第一个参与签名的 key 的 ID
```

```
            "sig": "{signature-by-index-key-1}",                # 该私钥对此文件 signed
                ↪ 部分签名的结果
        },
        ...
        {
            "keyid": "{id-of-root-key-N}",                      # 第 N 个参与签名私钥的 ID
            "sig": "{signature-by-root-key-N}"                  # 该私钥对此文件 signed
                ↪ 部分签名的结果
        }
    ],
    "signed": {
        "_type": "index",                                       # 指示该文件类型
        "components": {                                         # 组件列表
            "{component1}": {                                   # 第一个组件的名称
                "hidden": {bool},                               # 是否是隐藏组件
                "owner": "{owner-id}",                          # 组件管理员 ID
                "standalone": {bool},                           # 该组件是否可独立运行
                "url": "/{component}.json",                     # 获取组件的地址，需要加上版本号
                    ↪ : /{N}.{component}.json
                "yanked": {bool}                                # 该组件是否已被标记为删除
            },
            ...
            "{componentN}": {                                   # 第 N 个组件的名称
                ...
            },
        },
        "default_components": ["{component1}.."{componentN}"], # 镜像必须包含的默认组件，
            ↪ 该字段目前固定为空（未启用）
        "expires": "{expiration-date-of-this-file}",            # 该文件的过期时间，
            ↪ 过期后客户端会拒绝此文件
        "owners": {
            "{owner1}": {                                       # 第一个属主的 ID
                "keys": {                                       # 只有 keys
                    ↪ 中记录的密钥签名才是合法的
                    "{id-of-the-key-1}": {                      # 该属主的第一个密钥
                        "keytype": "rsa",                       # 密钥类型，目前固定为 rsa
                        "keyval": {                             # 密钥的 payload
                            "public": "{public-key-content}"    # 表示公钥内容
                        },
                        "scheme": "rsassa-pss-sha256"           # 目前固定为 rsassa-pss-sha256
                    },
                    ...
                    "{id-of-the-key-N}": {                      # 该属主的第 N 个密钥
                        ...
                    }
```

```
                },
                "name": "{owner-name}",                          # 该属主的名字
                "threshod": {N}                                  #
                    ↪ 指示该属主拥有的组件必须含有至少 N 个合法签名
            },
            ...
            "{ownerN}": {                                        # 第 N 个属主的 ID
                ...
            }
        }
        "spec_version": "0.1.0",                                 # 本文件遵循的规范版本,
            ↪ 未来变更文件结构需要升级版本号，目前为 0.1.0
        "version": {N}                                           # 本文件的版本号,
            ↪ 每次更新文件需要创建一个新的 {N+1}.index.json，并将其 version 设置为 N + 1
    }
}
```

13.4.10.2.3 组件

组件元信息文件记录了特定组件的平台以及版本信息。

其格式如下：

```
{
    "signatures": [                                              # 该文件的签名
        {
            "keyid": "{id-of-index-key-1}",                      # 第一个参与签名的 key 的 ID
            "sig": "{signature-by-index-key-1}",                 # 该私钥对此文件 signed
                ↪ 部分签名的结果
        },
        ...
        {
            "keyid": "{id-of-root-key-N}",                       # 第 N 个参与签名私钥的 ID
            "sig": "{signature-by-root-key-N}"                   # 该私钥对此文件 signed
                ↪ 部分签名的结果
        }
    ],
    "signed": {
        "_type": "component",                                    # 指示该文件类型
        "description": "{description-of-the-component}",         # 该组件的描述信息
        "expires": "{expiration-date-of-this-file}",             # 该文件的过期时间,
            ↪ 过期后客户端会拒绝此文件
        "id": "{component-id}",                                  # 该组件的 ID，具有全局唯一性
        "nightly": "{nightly-cursor}",                           # nightly 游标，值为最新的
            ↪ nightly 的版本号（如 v5.0.0-nightly-20201209）
```

```
        "platforms": {                                          # 该组件支持的平台（如 darwin/
            ↪ amd64，linux/arm64 等）
            "{platform-pair-1}": {
                "{version-1}": {                                # Semantic Version 版本号（如 v1
                    ↪ .0.0 等）
                    "dependencies": null,                       # 用于约定组件之间的依赖关系，
                        ↪ 该字段尚未使用，固定为 null
                    "entry": "{entry}",                         # 入口二进制文件位于 tar
                        ↪ 包的相对路径
                    "hashs": {                                  # tar 包的 checksum，我们使用
                        ↪ sha256 和 sha512
                        "sha256": "{sum-of-sha256}",
                        "sha512": "{sum-of-sha512}",
                    },
                    "length": {length-of-tar},                  # tar 包的长度
                    "released": "{release-time}",               # 该版本的 release 时间
                    "url": "{url-of-tar}",                      # tar 包的下载地址
                    "yanked": {bool}                            # 该版本是否已被禁用
                }
            },
            ...
            "{platform-pair-N}": {
                ...
            }
        },
        "spec_version": "0.1.0",                                # 本文件遵循的规范版本，
            ↪ 未来变更文件结构需要升级版本号，目前为 0.1.0
        "version": {N}                                          # 本文件的版本号，
            ↪ 每次更新文件需要创建一个新的 {N+1}.{component}.json，并将其 version 设置为 N + 1
}
```

13.4.10.2.4 快照

快照文件记录了各个元文件当前的版本号。

其格式如下：

```
{
    "signatures": [                                             # 该文件的签名
        {
            "keyid": "{id-of-index-key-1}",                     # 第一个参与签名的 key 的 ID
            "sig": "{signature-by-index-key-1}",                # 该私钥对此文件 signed
                ↪ 部分签名的结果
        },
        ...
        {
```

```
            "keyid": "{id-of-root-key-N}",                          # 第 N 个参与签名私钥的 ID
            "sig": "{signature-by-root-key-N}"                      # 该私钥对此文件 signed
                ↪ 部分签名的结果
        }
    ],
    "signed": {
        "_type": "snapshot",                                        # 指示该文件类型
        "expires": "{expiration-date-of-this-file}",                # 该文件的过期时间，
            ↪ 过期后客户端会拒绝此文件
        "meta": {                                                   # 其他元文件的信息
            "/root.json": {
                "length": {length-of-json-file},                    # root.json 的长度
                "version": {version-of-json-file}                   # root.json 的 version
            },
            "/index.json": {
                "length": {length-of-json-file},
                "version": {version-of-json-file}
            },
            "/{component-1}.json": {
                "length": {length-of-json-file},
                "version": {version-of-json-file}
            },
            ...
            "/{component-N}.json": {
                ...
            }
        },
        "spec_version": "0.1.0",                                    # 本文件遵循的规范版本，
            ↪ 未来变更文件结构需要升级版本号，目前为 0.1.0
        "version": 0                                                # 本文件的版本号，固定为 0
    }
```

13.4.10.2.5 时间戳

时间戳文件记录了当前快照 checksum。

其文件格式如下：

```
{
    "signatures": [                                                 # 该文件的签名
        {
            "keyid": "{id-of-index-key-1}",                         # 第一个参与签名的 key 的 ID
            "sig": "{signature-by-index-key-1}",                    # 该私钥对此文件 signed
                ↪ 部分签名的结果
        },
        ...
```

```
        {
            "keyid": "{id-of-root-key-N}",                          # 第 N 个参与签名私钥的 ID
            "sig": "{signature-by-root-key-N}"                      # 该私钥对此文件 signed
                ↪ 部分签名的结果
        }
    ],
    "signed": {
        "_type": "timestamp",                                       # 指示该文件类型
        "expires": "{expiration-date-of-this-file}",                # 该文件的过期时间，
            ↪ 过期后客户端会拒绝此文件
        "meta": {                                                   # snapshot.json 的信息
            "/snapshot.json": {
                "hashes": {
                    "sha256": "{sum-of-sha256}"                     # snapshot.json 的 sha256
                },
                "length": {length-of-json-file}                     # snapshot.json 的长度
            }
        },
        "spec_version": "0.1.0",                                    # 本文件遵循的规范版本，
            ↪ 未来变更文件结构需要升级版本号，目前为 0.1.0
        "version": {N}                                              # 本文件的版本号，
            ↪ 每次更新文件需要覆盖 timestamp.json，并将其 version 设置为 N + 1
```

13.4.10.3 客户端工作流程

客户端通过以下逻辑保证从镜像下载到的文件是安全的：

- 客户端安装时随 binary 附带了一个 root.json
- 客户端运行时以已有的 root.json 为基础，做如下操作：
  1. 获取 root.json 中的 version，记为 N
  2. 向镜像请求 {N+1}.root.json，若成功，使用 root.json 中记录的公钥验证该文件是否合法
- 向镜像请求 timestamp.json，并使用 root.json 中记录的公钥验证该文件是否合法
- 检查 timestamp.json 中记录的 snapshot.json 的 checksum 和本地的 snapshot.json 的 checksum 是否吻合
  - 若不吻合，则向镜像请求最新的 snapshot.json 并使用 root.json 中记录的公钥验证该文件是否合法
- 对于 index.json 文件，从 snapshot.json 中获取其版本号 N，并请求 {N}.index.json，然后使用 root.json 中记录的公钥验证该文件是否合法
- 对于组件（如 tidb.json，tikv.json），从 snapshot.json 中获取其版本号 N，并请求 {N}.{component}.json，然后使用 index.json 中记录的公钥验证该文件是否合法
- 对于组件 tar 文件，从 {component}.json 中获取其 url 及 checksum，请求 url 得到 tar 包，并验证 checksum 是否正确

13.4.11 TiUP 组件文档

13.4.11.1 本地快速部署 TiDB 集群

TiDB 集群是由多个组件构成的分布式系统，一个典型的 TiDB 集群至少由 3 个 PD 节点、3 个 TiKV 节点和 2 个 TiDB 节点构成。对于想要快速体验 TiDB 的用户来说，手工部署这么多组件是非常耗时且麻烦的事情。本文介绍 TiUP 中的 playground 组件，以及如何通过 playground 组件快速搭建一套本地的 TiDB 测试环境。

13.4.11.1.1 playground 组件介绍

playground 组件的基本用法：

```
tiup playground ${version} [flags]
```

如果直接执行 `tiup playground` 命令，则 TiUP playground 会使用本地安装的 TiDB/TiKV/PD 组件或者安装这些组件的稳定版本，来启动一个由 1 个 TiKV、1 个 TiDB、1 个 PD 和 1 个 TiFlash 实例构成的集群。该命令实际做了以下事情：

- 因为该命令没有指定 playground 的版本，TiUP 会先查找已安装的 playground 的最新版本，假设已安装的 playground 最新版为 v1.11.3，则该命令相当于 tiup playground:v1.11.3
- 如果 playground 从未安装过任何版本的 TiDB/TiKV/PD 组件，TiUP 会先安装这些组件的最新稳定版，然后再启动运行这些组件的实例
- 因为该命令没有指定 TiDB/PD/TiKV 各组件的版本，默认情况下，它会使用各组件的最新发布版本，假设当前为 v7.0.0，则该命令相当于 tiup playground:1.11.3 v7.0.0
- 因为该命令也没有指定各组件的个数，默认情况下，它会启动由 1 个 TiDB、1 个 TiKV、1 个 PD 和 1 个 TiFlash 实例构成的最小化集群
- 在依次启动完各个 TiDB 组件后，playground 会提醒集群启动成功，并告诉你一些有用的信息，譬如如何通过 MySQL 客户端连接集群、如何访问TiDB Dashboard 等

playground 的命令行参数说明：

```
Flags:
      --db int                    设置集群中 TiDB 节点的数量（默认为1）
      --db.host host              指定 TiDB 的监听地址
      --db.port int               指定 TiDB 的端口号
      --db.binpath string         指定 TiDB 二进制文件的位置（开发调试用，可忽略）
      --db.config string          指定 TiDB 的配置文件（开发调试用，可忽略）
      --db.timeout int            指定 TiDB 最长等待超时时间，单位为秒。若配置为 0，则永不超时。
      --drainer int               设置集群中 Drainer 数据
      --drainer.binpath string    指定 Drainer 二进制文件的位置（开发调试用，可忽略）
      --drainer.config string     指定 Drainer 的配置文件
  -h, --help                      打印帮助信息
      --host string               设置每个组件的监听地址（默认为 127.0.0.1），
          ↪ 如果要提供给别的电脑访问，可设置为 0.0.0.0
      --kv int                    设置集群中 TiKV 节点的数量（默认为1）
      --kv.binpath string         指定 TiKV 二进制文件的位置（开发调试用，可忽略）
      --kv.config string          指定 TiKV 的配置文件（开发调试用，可忽略）
      --mode string               指定 playground 的运行模式，取值选项为 'tidb'（默认）和 'tikv-
          ↪ slim'
      --pd int                    设置集群中 PD 节点的数量（默认为1）
```

```
      --pd.host host              指定 PD 的监听地址
      --pd.binpath string        指定 PD 二进制文件的位置（开发调试用，可忽略）
      --pd.config string         指定 PD 的配置文件（开发调试用，可忽略）
      --pump int                 设置集群中 Pump 节点的数量（非 0 的时候 TiDB 会开启 TiDB Binlog
          ↪ ）
      --pump.binpath string      指定 Pump 二进制文件的位置（开发调试用，可忽略）
      --pump.config string       指定 Pump 的配置文件（开发调试用，可忽略）
  -T, --tag string               设置 playground 的 tag 信息
      --ticdc int                设置集群中 TiCDC 节点的数量（默认为 0）
      --ticdc.binpath string     指定 TiCDC 二进制文件的位置（开发调试用，可忽略）
      --ticdc.config string      指定 TiCDC 的配置文件（开发调试用，可忽略）
      --tiflash int              设置集群中 TiFlash 节点的数量（默认为 1）
      --tiflash.binpath string   指定 TiFlash 的二进制文件位置（开发调试用，可忽略）
      --tiflash.config string    指定 TiFlash 的配置文件（开发调试用，可忽略）
      --tiflash.timeout int      指定 TiFlash 最长等待超时时间，单位为秒，若配置为 0，则永不超时
          ↪ 。
  -v, --version                  显示 playground 的版本号
      --without-monitor          设置不使用 Prometheus 和 Grafana 的监控功能。若不添加此参数，
          ↪ 则默认开启监控功能。
```

13.4.11.1.2 使用示例

查看可用的 TiDB 版本

```
tiup list tidb
```

启动一个指定版本的 TiDB 集群

```
tiup playground ${version}
```

将 `${version}` 替换为所需的版本号。

启动一个每日构建版的 TiDB 集群

```
tiup playground nightly
```

nightly 就是这个集群的版本号，这里指定为每日构建版本。

覆盖 PD 的默认配置

首先，你需要复制 PD 的配置模版。假设你将复制的配置文件放置在 `~/config/pd.toml`，按需修改一些内容后，执行以下命令可以覆盖 PD 的默认配置：

```
tiup playground --pd.config ~/config/pd.toml
```

替换默认的二进制文件

默认启动 playground 时，各个组件都是使用官方镜像组件包中的二进制文件启动的，如果本地编译了一个临时的二进制文件想要放入集群中测试，可以使用 `--{comp}.binpath` 这个参数替换，例如执行以下命令替换 TiDB 的二进制文件：

```
tiup playground --db.binpath /xx/tidb-server
```

启动多个组件实例

默认情况下各启动一个 TiDB、TiKV 和 PD 实例，如果希望启动多个，可以加上如下参数：

```
tiup playground --db 3 --pd 3 --kv 3
```

启动集群时指定 tag

Playground 集群在命令行退出时，会默认清空所有的集群数据。如果想要启动一个数据不被自动删除的 Playground 集群，需要在启动时指定集群 tag，指定后可以在 `~/.tiup/data` 路径下找到该集群的数据。在集群启动时指定 tag 的方法如下：

```
tiup playground --tag <tagname>
```

以这种方式启动的集群，在集群关闭以后，数据文件会保留。下一次可以继续使用该 tag 启动集群，从而使用从上一次集群关闭时的数据。

13.4.11.1.3 快速连接到由 playground 启动的 TiDB 集群

TiUP 提供了 `client` 组件，用于自动寻找并连接 playground 在本地启动的 TiDB 集群，使用方式为：

```
tiup client
```

该命令会在控制台上提供当前机器上由 playground 启动的 TiDB 集群列表，选中需要连接的 TiDB 集群，点击回车后，可以打开一个自带的 MySQL 客户端以连接 TiDB。

13.4.11.1.4 查看已启动集群的信息

```
tiup playground display
```

可以看到如下信息：

```
Pid    Role     Uptime
---    ----     ------
84518  pd       35m22.929404512s
84519  tikv     35m22.927757153s
84520  pump     35m22.92618275s
86189  tidb     exited
86526  tidb     34m28.293148663s
86190  drainer  35m19.91349249s
```

13.4.11.1.5 扩容集群

扩容集群的命令行参数与启动集群的相似。以下命令可以扩容两个 TiDB：

```
tiup playground scale-out --db 2
```

13.4.11.1.6 缩容集群

可在 `tiup playground scale-in` 命令中指定 `pid`，以缩容对应的实例。可以通过 `tiup playground display` 命令查看 `pid`。

```
tiup playground scale-in --pid 86526
```

13.4.11.2 使用 TiUP 部署运维 TiDB 线上集群

本文重在介绍如何使用 TiUP 的 cluster 组件，如果需要线上部署的完整步骤，可参考使用 TiUP 部署 TiDB 集群。

与 playground 组件用于部署本地测试集群类似，cluster 组件用于快速部署生产集群。对比 playground，cluster 组件提供了更强大的生产集群管理功能，包括对集群的升级、缩容、扩容甚至操作、审计等。

cluster 组件的帮助文档如下：

```
tiup cluster
```

```
Starting component `cluster`: /home/tidb/.tiup/components/cluster/v1.11.3/cluster
Deploy a TiDB cluster for production

Usage:
  tiup cluster [command]

Available Commands:
  check       对集群进行预检
  deploy      部署集群
  start       启动已部署的集群
  stop        停止集群
  restart     重启集群
  scale-in    集群缩容
  scale-out   集群扩容
  clean       清理数据
  destroy     销毁集群
  upgrade     升级集群
  display     获取集群信息
  list        获取集群列表
  audit       查看集群操作日志
  import      导入一个由 TiDB Ansible 部署的集群
  edit-config 编辑 TiDB 集群的配置
  reload      用于必要时重载集群配置
  patch       使用临时的组件包替换集群上已部署的组件
  help        打印 Help 信息

Flags:
  -c, --concurrency int     最大并行任务数（默认值为 5）
      --format string       （实验特性）输出的格式，支持 [default, json]（默认值为 "default"）
  -h, --help                帮助信息
```

```
      --ssh string           （实验特性）SSH 执行类型，可选值为 'builtin'、'system'、'none'。
      --ssh-timeout uint     SSH 连接超时时间
  -v, --version              版本信息
      --wait-timeout uint    等待操作超时的时间
  -y, --yes                  跳过所有的确认步骤
```

13.4.11.2.1 部署集群

部署集群的命令为 `tiup cluster deploy`，一般用法为：

```
tiup cluster deploy <cluster-name> <version> <topology.yaml> [flags]
```

该命令需要提供集群的名字、集群使用的 TiDB 版本（例如 `v6.5.0`），以及一个集群的拓扑文件。

拓扑文件的编写可参考示例。以一个最简单的拓扑为例，将下列文件保存为 `/tmp/topology.yaml`：

> **注意：**
>
> TiUP Cluster 组件的部署和扩容拓扑是使用 yaml 语法编写，所以需要注意缩进。

```
---

pd_servers:
  - host: 172.16.5.134
    name: pd-134
  - host: 172.16.5.139
    name: pd-139
  - host: 172.16.5.140
    name: pd-140

tidb_servers:
  - host: 172.16.5.134
  - host: 172.16.5.139
  - host: 172.16.5.140

tikv_servers:
  - host: 172.16.5.134
  - host: 172.16.5.139
  - host: 172.16.5.140

tiflash_servers:
  - host: 172.16.5.141
  - host: 172.16.5.142
  - host: 172.16.5.143
```

```
grafana_servers:
  - host: 172.16.5.134

monitoring_servers:
  - host: 172.16.5.134
```

TiUP 默认部署在 amd64 架构上运行的 binary，若目标机器为 arm64 架构，可以在拓扑文件中进行配置：

```
global:
  arch: "arm64"           # 让所有机器默认使用 arm64 的 binary

tidb_servers:
  - host: 172.16.5.134
    arch: "amd64"         # 这台机器会使用 amd64 的 binary
  - host: 172.16.5.139
    arch: "arm64"         # 这台机器会使用 arm64 的 binary
  - host: 172.16.5.140    # 没有配置 arch 字段的机器，会使用 global 中的默认值，这个例子中是
      ↪ arm64

...
```

假如我们想要使用 TiDB 的 v7.0.0 版本，集群名字为 prod-cluster，则执行以下命令：

```
tiup cluster deploy -p prod-cluster v7.0.0 /tmp/topology.yaml
```

执行过程中会再次确认拓扑结构并提示输入目标机器上的 root 密码（-p 表示使用密码）：

```
Please confirm your topology:
TiDB Cluster: prod-cluster
TiDB Version: v7.0.0
Type        Host          Ports                          OS/Arch       Directories
----        ----          -----                          -------       -----------
pd          172.16.5.134  2379/2380                      linux/x86_64  deploy/pd-2379,data/pd
    ↪ -2379
pd          172.16.5.139  2379/2380                      linux/x86_64  deploy/pd-2379,data/pd
    ↪ -2379
pd          172.16.5.140  2379/2380                      linux/x86_64  deploy/pd-2379,data/pd
    ↪ -2379
tikv        172.16.5.134  20160/20180                    linux/x86_64  deploy/tikv-20160,data/
    ↪ tikv-20160
tikv        172.16.5.139  20160/20180                    linux/x86_64  deploy/tikv-20160,data/
    ↪ tikv-20160
tikv        172.16.5.140  20160/20180                    linux/x86_64  deploy/tikv-20160,data/
    ↪ tikv-20160
tidb        172.16.5.134  4000/10080                     linux/x86_64  deploy/tidb-4000
tidb        172.16.5.139  4000/10080                     linux/x86_64  deploy/tidb-4000
```

```
tidb         172.16.5.140  4000/10080                       linux/x86_64  deploy/tidb-4000
tiflash      172.16.5.141  9000/8123/3930/20170/20292/8234  linux/x86_64  deploy/tiflash-9000,data
    ↪ /tiflash-9000
tiflash      172.16.5.142  9000/8123/3930/20170/20292/8234  linux/x86_64  deploy/tiflash-9000,data
    ↪ /tiflash-9000
tiflash      172.16.5.143  9000/8123/3930/20170/20292/8234  linux/x86_64  deploy/tiflash-9000,data
    ↪ /tiflash-9000
prometheus   172.16.5.134  9090         deploy/prometheus-9090,data/prometheus-9090
grafana      172.16.5.134  3000         deploy/grafana-3000
Attention:
    1. If the topology is not what you expected, check your yaml file.
    1. Please confirm there is no port/directory conflicts in same host.
Do you want to continue? [y/N]:
```

输入密码后 tiup-cluster 便会下载需要的组件并部署到对应的机器上，当看到以下提示时说明部署成功：

```
Deployed cluster `prod-cluster` successfully
```

13.4.11.2.2 查看集群列表

集群部署成功后，可以通过 `tiup cluster list` 命令在集群列表中查看该集群：

```
tiup cluster list
```

```
Starting /root/.tiup/components/cluster/v1.11.3/cluster list
Name          User  Version    Path                                            PrivateKey
----          ----  -------    ----                                            ----------
prod-cluster  tidb  v7.0.0    /root/.tiup/storage/cluster/clusters/prod-cluster  /root/.tiup/
    ↪ storage/cluster/clusters/prod-cluster/ssh/id_rsa
```

13.4.11.2.3 启动集群

集群部署成功后，可以执行以下命令启动该集群。如果忘记了部署的集群的名字，可以使用 `tiup cluster list` 命令查看。

```
tiup cluster start prod-cluster
```

TiUP 使用 Systemd 启动守护进程。如果进程意外退出，会在 15s 间隔后被重新拉起。

13.4.11.2.4 检查集群状态

如果想查看集群中每个组件的运行状态，逐一登录到各个机器上查看显然很低效。因此，TiUP 提供了 `tiup cluster display` 命令，用法如下：

```
tiup cluster display prod-cluster
```

```
Starting /root/.tiup/components/cluster/v1.11.3/cluster display prod-cluster
TiDB Cluster: prod-cluster
TiDB Version: v7.0.0
ID                  Role        Host          Ports                            OS/Arch
    ↪ Status  Data Dir              Deploy Dir
--                  ----        ----          -----                            -------
    ↪ ------  --------              ----------
172.16.5.134:3000   grafana     172.16.5.134  3000                             linux/x86_64  Up
    ↪       -                     deploy/grafana-3000
172.16.5.134:2379   pd          172.16.5.134  2379/2380                        linux/x86_64  Up|L
    ↪     data/pd-2379          deploy/pd-2379
172.16.5.139:2379   pd          172.16.5.139  2379/2380                        linux/x86_64  Up|
    ↪ UI    data/pd-2379          deploy/pd-2379
172.16.5.140:2379   pd          172.16.5.140  2379/2380                        linux/x86_64  Up
    ↪       data/pd-2379          deploy/pd-2379
172.16.5.134:9090   prometheus  172.16.5.134  9090                             linux/x86_64  Up
    ↪       data/prometheus-9090  deploy/prometheus-9090
172.16.5.134:4000   tidb        172.16.5.134  4000/10080                       linux/x86_64  Up
    ↪       -                     deploy/tidb-4000
172.16.5.139:4000   tidb        172.16.5.139  4000/10080                       linux/x86_64  Up
    ↪       -                     deploy/tidb-4000
172.16.5.140:4000   tidb        172.16.5.140  4000/10080                       linux/x86_64  Up
    ↪       -                     deploy/tidb-4000
172.16.5.141:9000   tiflash     172.16.5.141  9000/8123/3930/20170/20292/8234  linux/x86_64  Up
    ↪       data/tiflash-9000     deploy/tiflash-9000
172.16.5.142:9000   tiflash     172.16.5.142  9000/8123/3930/20170/20292/8234  linux/x86_64  Up
    ↪       data/tiflash-9000     deploy/tiflash-9000
172.16.5.143:9000   tiflash     172.16.5.143  9000/8123/3930/20170/20292/8234  linux/x86_64  Up
    ↪       data/tiflash-9000     deploy/tiflash-9000
172.16.5.134:20160  tikv        172.16.5.134  20160/20180                      linux/x86_64  Up
    ↪       data/tikv-20160       deploy/tikv-20160
172.16.5.139:20160  tikv        172.16.5.139  20160/20180                      linux/x86_64  Up
    ↪       data/tikv-20160       deploy/tikv-20160
172.16.5.140:20160  tikv        172.16.5.140  20160/20180                      linux/x86_64  Up
    ↪       data/tikv-20160       deploy/tikv-20160
```

Status 列用 Up 或者 Down 表示该服务是否正常。对于 PD 组件，同时可能会带有 |L 表示该 PD 是 Leader，|UI 表示该 PD 运行着TiDB Dashboard。

13.4.11.2.5 缩容节点

> **注意：**

本节只展示缩容命令的语法示例，线上扩缩容具体步骤可参考使用 TiUP 扩容缩容 TiDB 集群。

缩容即下线服务，最终会将指定的节点从集群中移除，并删除遗留的相关文件。

由于 TiKV、TiFlash 和 TiDB Binlog 组件的下线是异步的（需要先通过 API 执行移除操作）并且下线过程耗时较长（需要持续观察节点是否已经下线成功），所以对 TiKV、TiFlash 和 TiDB Binlog 组件做了特殊处理：

- 对 TiKV、TiFlash 及 Binlog 组件的操作
  - TiUP cluster 通过 API 将其下线后直接退出而不等待下线完成
  - 等之后再执行集群操作相关的命令时，会检查是否存在已经下线完成的 TiKV、TiFlash 或者 Binlog 节点。如果不存在，则继续执行指定的操作；如果存在，则执行如下操作：
    1. 停止已经下线掉的节点的服务
    2. 清理已经下线掉的节点的相关数据文件
    3. 更新集群的拓扑，移除已经下线掉的节点
- 对其他组件的操作
  - 下线 PD 组件时，会通过 API 将指定节点从集群中删除掉（这个过程很快），然后停掉指定 PD 的服务并且清除该节点的相关数据文件
  - 下线其他组件时，直接停止并且清除节点的相关数据文件

缩容命令的基本用法：

```
tiup cluster scale-in <cluster-name> -N <node-id>
```

它需要指定至少两个参数，一个是集群名字，另一个是节点 ID。节点 ID 可以参考上一节使用 `tiup cluster display` 命令获取。

比如想缩容 172.16.5.140 上的 TiKV 节点，可以执行：

```
tiup cluster scale-in prod-cluster -N 172.16.5.140:20160
```

通过 `tiup cluster display` 可以看到该 TiKV 已经被标记为 `Offline`：

```
tiup cluster display prod-cluster
```

```
Starting /root/.tiup/components/cluster/v1.11.3/cluster display prod-cluster
TiDB Cluster: prod-cluster
TiDB Version: v7.0.0
ID                  Role        Host          Ports                             OS/Arch
    ↪ Status   Data Dir              Deploy Dir
--                  ----        ----          -----                             -------
    ↪ ------   --------              ----------
172.16.5.134:3000   grafana     172.16.5.134  3000                              linux/x86_64  Up
    ↪       -                       deploy/grafana-3000
172.16.5.134:2379   pd          172.16.5.134  2379/2380                         linux/x86_64  Up|L
    ↪      data/pd-2379          deploy/pd-2379
```

```
172.16.5.139:2379   pd          172.16.5.139  2379/2380                          linux/x86_64  Up|
    ↪ UI    data/pd-2379          deploy/pd-2379
172.16.5.140:2379   pd          172.16.5.140  2379/2380                          linux/x86_64  Up
    ↪       data/pd-2379          deploy/pd-2379
172.16.5.134:9090   prometheus  172.16.5.134  9090                               linux/x86_64  Up
    ↪       data/prometheus-9090  deploy/prometheus-9090
172.16.5.134:4000   tidb        172.16.5.134  4000/10080                         linux/x86_64  Up
    ↪       -                     deploy/tidb-4000
172.16.5.139:4000   tidb        172.16.5.139  4000/10080                         linux/x86_64  Up
    ↪       -                     deploy/tidb-4000
172.16.5.140:4000   tidb        172.16.5.140  4000/10080                         linux/x86_64  Up
    ↪       -                     deploy/tidb-4000
172.16.5.141:9000   tiflash     172.16.5.141  9000/8123/3930/20170/20292/8234    linux/x86_64  Up
    ↪       data/tiflash-9000     deploy/tiflash-9000
172.16.5.142:9000   tiflash     172.16.5.142  9000/8123/3930/20170/20292/8234    linux/x86_64  Up
    ↪       data/tiflash-9000     deploy/tiflash-9000
172.16.5.143:9000   tiflash     172.16.5.143  9000/8123/3930/20170/20292/8234    linux/x86_64  Up
    ↪       data/tiflash-9000     deploy/tiflash-9000
172.16.5.134:20160  tikv        172.16.5.134  20160/20180                        linux/x86_64  Up
    ↪       data/tikv-20160       deploy/tikv-20160
172.16.5.139:20160  tikv        172.16.5.139  20160/20180                        linux/x86_64  Up
    ↪       data/tikv-20160       deploy/tikv-20160
172.16.5.140:20160  tikv        172.16.5.140  20160/20180                        linux/x86_64
    ↪ Offline  data/tikv-20160       deploy/tikv-20160
```

待 PD 将其数据调度到其他 TiKV 后，该节点会被自动删除。

13.4.11.2.6 扩容节点

> **注意：**
>
> 本节只用于展示扩容命令的语法示例，线上扩缩容可参考使用 TiUP 扩容缩容 TiDB 集群。

扩容的内部逻辑与部署类似，TiUP cluster 组件会先保证节点的 SSH 连接，在目标节点上创建必要的目录，然后执行部署并且启动服务。其中 PD 节点的扩容会通过 join 方式加入到集群中，并且会更新与 PD 有关联的服务的配置；其他服务直接启动加入到集群中。所有服务在扩容时都会做正确性验证，最终返回是否扩容成功。

例如，在集群 `tidb-test` 中扩容一个 TiKV 节点和一个 PD 节点：

1. 新建 scale.yaml 文件，添加新增的 TiKV 和 PD 节点 IP：

> **注意：**
>
> 需要新建一个拓扑文件，文件中只写入扩容节点的描述信息，不要包含已存在的节点。

```
---

pd_servers:
  - host: 172.16.5.140

tikv_servers:
  - host: 172.16.5.140
```

2. 执行扩容操作。TiUP cluster 根据 scale.yaml 文件中声明的端口、目录等信息在集群中添加相应的节点：

```
tiup cluster scale-out tidb-test scale.yaml
```

执行完成之后可以通过 `tiup cluster display tidb-test` 命令检查扩容后的集群状态。

13.4.11.2.7 滚动升级

> **注意：**
>
> 本节只用于展示命令的语法示例，线上升级请参考使用 TiUP 升级 TiDB。

滚动升级功能借助 TiDB 的分布式能力，升级过程中尽量保证对前端业务透明、无感知。升级时会先检查各个组件的配置文件是否合理，如果配置有问题，则报错退出；如果配置没有问题，则工具会逐个节点进行升级。其中对不同节点有不同的操作。

不同节点的操作

- 升级 PD 节点
  - 优先升级非 Leader 节点
  - 所有非 Leader 节点升级完成后再升级 Leader 节点
    * 工具会向 PD 发送一条命令将 Leader 迁移到升级完成的节点上
    * 当 Leader 已经切换到其他节点之后，再对旧的 Leader 节点做升级操作
  - 同时升级过程中，若发现有不健康的节点，工具会中止本次升级并退出，此时需要由人工判断、修复后再执行升级。
- 升级 TiKV 节点
  - 先在 PD 中添加一个迁移对应 TiKV 上 Region leader 的调度，通过迁移 Leader 确保升级过程中不影响前端业务
  - 等待迁移 Leader 完成之后，再对该 TiKV 节点进行升级更新
  - 等更新后的 TiKV 正常启动之后再移除迁移 Leader 的调度
- 升级其他服务
  - 正常停止服务再更新

升级操作

升级命令参数如下：

```
Usage:
  tiup cluster upgrade <cluster-name> <version> [flags]

Flags:
      --force                 在不 transfer leader 的情况下强制升级（危险操作）
  -h, --help                  帮助手册
      --transfer-timeout int  transfer leader 的超时时间

Global Flags:
      --ssh string         （实验特性）SSH 执行类型，可选值为 'builtin'、'system'、'none'。
      --wait-timeout int   等待操作超时的时间
      --ssh-timeout int    SSH 连接的超时时间
  -y, --yes                跳过所有的确认步骤
```

例如，把集群升级到 v7.0.0 的命令为：

```
tiup cluster upgrade tidb-test v7.0.0
```

13.4.11.2.8 更新配置

如果想要动态更新组件的配置，TiUP cluster 组件为每个集群保存了一份当前的配置，如果想要编辑这份配置，则执行 `tiup cluster edit-config <cluster-name>` 命令。例如：

```
tiup cluster edit-config prod-cluster
```

然后 TiUP cluster 组件会使用 vi 打开配置文件供编辑（如果你想要使用其他编辑器，请使用 `EDITOR` 环境变量自定义编辑器，例如 `export EDITOR=nano`），编辑完之后保存即可。此时的配置并没有应用到集群，如果想要让它生效，还需要执行：

```
tiup cluster reload prod-cluster
```

该操作会将配置发送到目标机器，重启集群，使配置生效。

> 注意：
>
> 对于监控组件，可以通过执行 `tiup cluster edit-config` 命令在对应实例上添加自定义配置路径来进行配置自定义，例如：

```
---

grafana_servers:
```

```
  - host: 172.16.5.134
    dashboard_dir: /path/to/local/dashboards/dir

monitoring_servers:
  - host: 172.16.5.134
    rule_dir: /path/to/local/rules/dir

alertmanager_servers:
  - host: 172.16.5.134
    config_file: /path/to/local/alertmanager.yml
```

路径内容格式如下：

- grafana_servers 的 dashboard_dir 字段指定的文件夹中应当含有完整的 *.json 文件。
- monitoring_servers 的 rule_dir 字段定义的文件夹中应当含有完整的 *.rules.yml 文件。
- alertmanager_servers 的 config_file 格式请参考 Alertmanager 配置模板。

在执行 tiup reload 时，TiUP 会将中控机上对应的配置上传到目标机器对应的配置目录中，上传之前会删除目标机器中已有的旧配置文件。如果想要修改某一个配置文件，请确保将所有的（包含未修改的）配置文件都放在同一个目录中。例如，要修改 Grafana 的 tidb.json 文件，可以先将 Grafana 的 dashboards 目录中所有的 *.json 文件拷贝到本地目录中，再修改 tidb.json 文件。否则最终的目标机器上将缺失其他的 JSON 文件。

> 注意：
>
> 如果配置了 grafana_servers 的 dashboard_dir 字段，在执行 tiup cluster rename 命令进行集群重命名后，需要完成以下操作：
>
> 1. 在本地的 dashboards 目录中，将集群名修改为新的集群名。
> 2. 在本地的 dashboards 目录中，将 datasource 更新为新的集群名（datasource 是以集群名命名的）。
> 3. 执行 tiup cluster reload -R grafana 命令。

13.4.11.2.9 更新组件

常规的升级集群可以使用 upgrade 命令，但是在某些场景下（例如 Debug），可能需要用一个临时的包替换正在运行的组件，此时可以用 patch 命令：

```
tiup cluster patch --help
```

```
Replace the remote package with a specified package and restart the service

Usage:
  tiup cluster patch <cluster-name> <package-path> [flags]
```

```
Flags:
  -h, --help                    帮助信息
  -N, --node strings            指定被替换的节点
      --overwrite               在未来的 scale-out 操作中使用当前指定的临时包
  -R, --role strings            指定被替换的服务类型
      --transfer-timeout int    transfer leader 的超时时间

Global Flags:
      --ssh string          （实验特性）SSH 执行类型，可选值为 'builtin'、'system'、'none'。
      --wait-timeout int    等待操作超时的时间
      --ssh-timeout int     SSH 连接的超时时间
  -y, --yes                 跳过所有的确认步骤
```

例如，有一个 TiDB 的 hotfix 包放在 `/tmp/tidb-hotfix.tar.gz`，如果此时想要替换集群上的所有 TiDB，则可以执行：

```
tiup cluster patch test-cluster /tmp/tidb-hotfix.tar.gz -R tidb
```

或者只替换其中一个 TiDB：

```
tiup cluster patch test-cluster /tmp/tidb-hotfix.tar.gz -N 172.16.4.5:4000
```

13.4.11.2.10 导入 TiDB Ansible 集群

> **注意：**
>
> TiUP cluster 组件对 TiSpark 的支持目前为实验性特性，暂不支持导入启用了 TiSpark 组件的集群。

在 TiUP 之前，一般使用 TiDB Ansible 部署 TiDB 集群，import 命令用于将这部分集群过渡给 TiUP 接管。import 命令用法如下：

```
tiup cluster import --help
```

```
Import an exist TiDB cluster from TiDB-Ansible

Usage:
  tiup cluster import [flags]

Flags:
  -d, --dir string         TiDB-Ansible 的目录，默认为当前目录
  -h, --help               import 的帮助信息
      --inventory string   inventory 文件的名字（默认为 "inventory.ini"）
      --no-backup          不备份 Ansible 目录，用于存在多个 inventory 文件的 Ansible 目录
  -r, --rename NAME        重命名被导入的集群
```

```
Global Flags:
      --ssh string         （实验特性）SSH 执行类型，可选值为 'builtin'、'system'、'none'。
      --wait-timeout int   等待操作超时的时间
      --ssh-timeout int    SSH 连接的超时时间
  -y, --yes                跳过所有的确认步骤
```

例如，导入一个 TiDB Ansible 集群：

```
cd tidb-ansible
tiup cluster import
```

或者

```
tiup cluster import --dir=/path/to/tidb-ansible
```

13.4.11.2.11 查看操作日志

操作日志的查看可以借助 audit 命令，其用法如下：

```
Usage:
  tiup cluster audit [audit-id] [flags]

Flags:
  -h, --help   help for audit
```

在不使用 [audit-id] 参数时，该命令会显示执行的命令列表，如下：

```
tiup cluster audit
```

```
Starting component `cluster`: /home/tidb/.tiup/components/cluster/v1.11.3/cluster audit
ID      Time                       Command
--      ----                       -------
4BLhr0  2022-03-29T23:55:09+08:00  /home/tidb/.tiup/components/cluster/v1.11.3/cluster deploy
    ↪ test v7.0.0 /tmp/topology.yaml
4BKWjF  2022-03-029T23:36:57+08:00  /home/tidb/.tiup/components/cluster/v1.11.3/cluster deploy
    ↪ test v7.0.0 /tmp/topology.yaml
4BKVwH  2022-03-29T23:02:08+08:00  /home/tidb/.tiup/components/cluster/v1.11.3/cluster deploy
    ↪ test v7.0.0 /tmp/topology.yaml
4BKKH1  2022-03-29T16:39:04+08:00  /home/tidb/.tiup/components/cluster/v1.11.3/cluster destroy
    ↪ test
4BKKDx  2022-03-29T16:36:57+08:00  /home/tidb/.tiup/components/cluster/v1.11.3/cluster deploy
    ↪ test v7.0.0 /tmp/topology.yaml
```

第一列为 audit-id，如果想看某个命令的执行日志，则传入这个 audit-id：

```
tiup cluster audit 4BLhr0
```

13.4.11.2.12 在集群节点机器上执行命令

exec 命令可以很方便地到集群的机器上执行命令，其使用方式如下：

```
Usage:
  tiup cluster exec <cluster-name> [flags]

Flags:
      --command string    需要执行的命令（默认是 "ls"）
  -h, --help              帮助信息
  -N, --node strings      指定要执行的节点 ID （节点 id 通过 display 命令获取）
  -R, --role strings      指定要执行的 role
      --sudo              是否使用 root（默认为 false）
```

例如，如果要到所有的 TiDB 节点上执行 `ls /tmp`：

```
tiup cluster exec test-cluster --command='ls /tmp'
```

13.4.11.2.13 集群控制工具 (controllers)

在 TiUP 之前，我们用 tidb-ctl、tikv-ctl、pd-ctl 等工具操控集群，为了方便下载和使用，TiUP 将它们集成到了统一的组件 ctl 中：

```
Usage:
  tiup ctl:v<CLUSTER_VERSION> {tidb/pd/tikv/binlog/etcd} [flags]

Flags:
  -h, --help   help for tiup
```

这个命令和之前的命令对应关系为：

```
tidb-ctl [args] = tiup ctl tidb [args]
pd-ctl [args] = tiup ctl pd [args]
tikv-ctl [args] = tiup ctl tikv [args]
binlogctl [args] = tiup ctl bindlog [args]
etcdctl [args] = tiup ctl etcd [args]
```

例如，以前查看 store 的命令为 `pd-ctl -u http://127.0.0.1:2379 store`，集成到 TiUP 中的命令为：

```
tiup ctl:v<CLUSTER_VERSION> pd -u http://127.0.0.1:2379 store
```

13.4.11.2.14 部署机环境检查

使用 check 子命令可以对部署机的环境进行一系列检查，并输出检查结果。通过执行 check 子命令，可以发现常见的不合理配置或不支持情况。命令参数列表如下：

```
Usage:
  tiup cluster check <topology.yml | cluster-name> [flags]
```

```
Flags:
      --apply                  Try to fix failed checks
      --cluster                Check existing cluster, the input is a cluster name.
      --enable-cpu             Enable CPU thread count check
      --enable-disk            Enable disk IO (fio) check
      --enable-mem             Enable memory size check
  -h, --help                   help for check
  -i, --identity_file string   The path of the SSH identity file. If specified, public key
      ↪ authentication will be used.
  -p, --password               Use password of target hosts. If specified, password
      ↪ authentication will be used.
      --user string            The user name to login via SSH. The user must has root (or sudo)
          ↪ privilege.
```

默认情况下，此功能用于在部署前进行环境检查，通过指定 `--cluster` 参数切换模式，也可以用于对已部署集群的部署机进行检查，例如：

```
#### check deploy servers before deploy
tiup cluster check topology.yml --user tidb -p

#### check deploy servers of an existing cluster
tiup cluster check <cluster-name> --cluster
```

其中，CPU 线程数检查、内存大小检查和磁盘性能检查三项默认关闭，对于生产环境，建议将此三项检测开启并确保通过，以获得最佳性能。

- CPU：线程数大于等于 16 为通过检查
- 内存：物理内存总大小大于等于 32 GB 为通过检查
- 磁盘：对 `data_dir` 所在分区执行 `fio` 测试并记录结果

在运行检测时，若指定了 `--apply` 参数，程序将尝试对其中未通过的项目自动修复。自动修复仅限于部分可通过修改配置或系统参数调整的项目，其它未修复的项目需要根据实际情况手工处理。

环境检查不是部署集群的必需流程。对于生产环境建议在部署前执行环境检查并通过所有检测项。如果未通过全部检查项，也可能正常部署和运行集群，但可能无法获得最佳性能表现。

13.4.11.2.15 使用中控机系统自带的 SSH 客户端连接集群

在以上所有操作中，涉及到对集群机器的操作都是通过 TiUP 内置的 SSH 客户端连接集群执行命令，但是在某些场景下，需要使用系统自带的 SSH 客户端来对集群执行操作，比如：

- 使用 SSH 插件来做认证
- 使用定制的 SSH 客户端

此时可以通过命令行参数 `--ssh=system` 启用系统自带命令行：

- 部署集群：`tiup cluster deploy <cluster-name> <version> <topo> --ssh=system`，其中 `<cluster-name>` 为集群名称，`<version>` 为 TiDB 集群版本（例如 `v6.5.0`），`<topo>` 为拓扑文件路径
- 启动集群：`tiup cluster start <cluster-name> --ssh=system`
- 升级集群：`tiup cluster upgrade ... --ssh=system`

所有涉及集群操作的步骤都可以加上 `--ssh=system` 来使用系统自带的客户端。

也可以使用环境变量 `TIUP_NATIVE_SSH` 来指定是否使用本地 SSH 客户端，避免每个命令都需要添加 `--ssh=system` 参数：

```
export TIUP_NATIVE_SSH=true
#### 或者
export TIUP_NATIVE_SSh=1
#### 或者
export TIUP_NATIVE_SSH=enable
```

若环境变量和 `--ssh` 同时指定，则以 `--ssh` 为准。

> 注意：
>
> 在部署集群的步骤中，若需要使用密码的方式连接 (`-p`)，或者密钥文件设置了 passphrase，则需要保证中控机上安装了 sshpass，否则连接时会报错。

13.4.11.2.16 迁移中控机与备份

TiUP 相关的数据都存储在用户 home 目录的 `.tiup` 目录下，若要迁移中控机只需要拷贝 `.tiup` 目录到对应目标机器即可。

1. 在原机器 home 目录下执行 `tar czvf tiup.tar.gz .tiup`。
2. 把 `tiup.tar.gz` 拷贝到目标机器 home 目录。
3. 在目标机器 home 目录下执行 `tar xzvf tiup.tar.gz`。
4. 添加 `.tiup` 目录到 PATH 环境变量。

   如使用 bash 并且是 tidb 用户，在 `~/.bashrc` 中添加 `export PATH=/home/tidb/.tiup/bin:$PATH` 后执行 `source ~/.bashrc`，根据使用的 shell 与用户做相应调整。

> 注意：
>
> 为了避免中控机磁盘损坏等异常情况导致 TiUP 数据丢失，建议定时备份 `.tiup` 目录。

13.4.11.2.17 备份与恢复集群部署和运维所需的 meta 文件

如果运维所需的 meta 文件丢失，会导致无法继续使用 TiUP 管理集群，建议通过以下方式定期备份 meta 文件：

```
tiup cluster meta backup ${cluster_name}
```

如果 meta 文件丢失，可以使用以下方法恢复 meta 文件：

```
tiup cluster meta restore ${cluster_name} ${backup_file}
```

> **注意：**
>
> 恢复操作会覆盖当前的 meta 文件，建议仅在 meta 文件丢失的情况下进行恢复。

13.4.11.3 搭建私有镜像

在构建私有云时，通常会使用隔离的网络环境，此时无法访问 TiUP 的官方镜像。因此，TiUP 提供了构建私有镜像的方案，它主要由 mirror 指令来实现，该方案也可用于离线部署。使用私有镜像，你可以使用自己构建和打包的组件。

13.4.11.3.1 mirror 指令介绍

`mirror` 指令的帮助文档如下：

```
tiup mirror --help
```

```
The `mirror` command is used to manage a component repository for TiUP, you can use
it to create a private repository, or to add new component to an existing repository.
The repository can be used either online or offline.
It also provides some useful utilities to help manage keys, users, and versions
of components or the repository itself.

Usage:
  tiup mirror <command> [flags]

Available Commands:
  init        Initialize an empty repository
  sign        Add signatures to a manifest file
  genkey      Generate a new key pair
  clone       Clone a local mirror from remote mirror and download all selected components
  merge       Merge two or more offline mirrors
  publish     Publish a component
  show        Show the mirror address
  set         Set mirror address
  modify      Modify published component
```

```
  renew       Renew the manifest of a published component.
  grant       grant a new owner
  rotate      Rotate root.json

Flags:
  -h, --help          help for mirror
      --repo string   Path to the repository

Global Flags:
      --help Help for this command

Use "tiup mirror [command] --help" for more information about a command.
```

13.4.11.3.2 克隆镜像

执行 `tiup mirror clone` 命令，可构建本地地镜像：

```
tiup mirror clone <target-dir> [global-version] [flags]
```

- `target-dir`：指需要把克隆下来的数据放到哪个目录里。
- `global-version`：用于为所有组件快速设置一个共同的版本。

`tiup mirror clone` 命令提供了很多可选参数，日后可能会提供更多。但这些参数其实可以分为四类：

1. 克隆时是否使用前缀匹配方式匹配版本

   如果指定了 `--prefix` 参数，则会才用前缀匹配方式匹配克隆的版本号。例：指定 `--prefix` 时，填写版本 “v5.0.0” 将会匹配 “v5.0.0-rc”，“v5.0.0”

2. 是否全量克隆

   如果指定了 `--full` 参数，则会完整地克隆官方镜像。

   > 注意：
   > 如果既不指定 `--full` 参数，又不指定 `global-version` 或克隆的 component 版本，那么 TiUP 就只会克隆一些元信息。

3. 限定只克隆特定平台的包

   如果只想克隆某个平台的包，那么可以使用 `--os` 和 `--arch` 来限定：

   - 只想克隆 linux 平台的，则执行 `tiup mirror clone <target-dir> [global-version] --os=linux`
   - 只想克隆 amd64 架构的，则执行 `tiup mirror clone <target-dir> [global-version] --arch=amd64`
   - 只想克隆 linux/amd64 的，则执行 `tiup mirror clone <target-dir> [global-version] --os=linux --arch=amd64`

4. 限定只克隆组件的特定版本

   如果只想克隆某个组件的某一个版本而不是所有版本，则使用 `--<component>=<version>` 来限定，例如：

   - 只想克隆 TiDB 的 v7.0.0 版本，则执行 `tiup mirror clone <target-dir> --tidb v7.0.0`
   - 只想克隆 TiDB 的 v7.0.0 版本，以及 TiKV 的所有版本，则执行 `tiup mirror clone <target-dir> --tidb v7.0.0 --tikv all`
   - 克隆一个集群的所有组件的 v7.0.0 版本，则执行 `tiup mirror clone <target-dir> v7.0.0`

克隆完成后，签名密钥会自动设置。

管理私有仓库

你可以通过 SCP 和 NFS 文件共享方式，将 `tiup mirror clone` 克隆下来的仓库共享给其他主机，也可以通过 HTTP 或 HTTPS 协议来共享。可以使用 `tiup mirror set <location>` 指定仓库的位置。

```
tiup mirror set /shared_data/tiup
```

```
tiup mirror set https://tiup-mirror.example.com/
```

> 注意：
>
> 如果在执行了 `tiup mirror clone` 的机器上执行 `tiup mirror set`，下次执行 `tiup mirror clone` 时，机器会从本地镜像而非远程镜像进行克隆。因此，更新私有镜像前，需要执行 `tiup mirror set --reset` 来重置镜像。

还可以通过 `TIUP_MIRRORS` 环境变量来使用镜像。下面是一个使用私有仓库运行 `tiup list` 的例子。

```
export TIUP_MIRRORS=/shared_data/tiup
tiup list
```

设置 `TIUP_MIRRORS` 会永久改变镜像配置，例如 `tiup mirror set`。详情请参考 tiup issue #651。

更新私有仓库

如果使用同样的 `target-dir` 目录再次运行 `tiup mirror clone` 命令，机器会创建新的 manifest，并下载可用的最新版本的组件。

> 注意：
>
> 重新创建 manifest 之前，请确保所有组件和版本（包括之前下载的早期版本）都包含在内。

13.4.11.3.3 自定义仓库

你可以创建一个自定义仓库，以使用自己构建的 TiDB 组件，例如 TiDB、TiKV 或 PD。你也可以创建自己的 TiUP 组件。

要创建自己的组件，请执行 `tiup package` 命令，并按照组件打包的说明进行操作。

创建自定义仓库

以下命令在 `/data/mirror` 目录下创建一个空仓库：

```
tiup mirror init /data/mirror
```

创建仓库时，密钥会被写入 `/data/mirror/keys`。

以下命令在 `~/.tiup/keys/private.json` 中创建一个私钥：

```
tiup mirror genkey
```

以下命令为 `jdoe` 授予 `/data/mirror` 路径下私钥 `~/.tiup/keys/private.json` 的所有权：

```
tiup mirror set /data/mirror
tiup mirror grant jdoe
```

使用自定义组件

1. 创建一个名为 `hello` 的自定义组件：

   ```
   $ cat > hello.c << END
   > #include <stdio.h>
   int main() {
     printf("hello\n");
     return (0);
   }
   END
   $ gcc hello.c -o hello
   $ tiup package hello --entry hello --name hello --release v0.0.1
   ```

   `package/hello-v0.0.1-linux-amd64.tar.gz` 创建成功。

2. 创建一个仓库和一个私钥，并为仓库授予所有权：

   ```
   $ tiup mirror init /tmp/m
   $ tiup mirror genkey
   $ tiup mirror set /tmp/m
   $ tiup mirror grant $USER
   ```

   ```
   tiup mirror publish hello v0.0.1 package/hello-v0.0.1-linux-amd64.tar.gz hello
   ```

3. 运行组件。如果组件还没有安装，会先下载安装：

```
$ tiup hello
```

```
The component `hello` version  is not installed; downloading from repository.
Starting component `hello`: /home/dvaneeden/.tiup/components/hello/v0.0.1/hello
hello
```

执行 `tiup mirror merge` 命令，可以将自定义组件的仓库合并到另一个仓库中。这一操作假设 `/data/my_custom_components` 中的所有组件都使用 `$USER` 签名：

```
$ tiup mirror set /data/my_mirror
$ tiup mirror grant $USER
$ tiup mirror merge /data/my_custom_components
```

13.4.11.4 使用 TiUP bench 组件压测 TiDB

在测试数据库性能时，经常需要对数据库进行压测，为了满足这一需求，TiUP 集成了 bench 组件。TiUP bench 组件提供多种压测的 workloads，命令分别如下：

```
tiup bench tpcc   # 以 TPC-C 作为 workload 压测
tiup bench tpch   # 以 TPC-H 作为 workload 压测
tiup bench ch     # 以 CH-benCHmark 作为 workload 压测
tiup bench ycsb   # 以 YCSB 作为 workload 压测
tiup bench rawsql # 以自定义 SQL 文件作为 workload 压测
```

其中 `tpcc`, `tpch`, `ch`, `rawsql` 支持如下命令行参数。`ycsb` 使用方法较为不同，它主要通过 properties 文件进行配置，详见 go-ycsb 使用说明。

```
  -t, --acThreads int         OLAP 并发线程数，仅适用于 CH-benCHmark (默认 1)
      --conn-params string    数据库连接参数，例如:
                              `--conn-params tidb_isolation_read_engines='tiflash'` 设置 TiDB
                                  ↪ 通过 TiFlash 进行查询
                              `--conn-params sslmode=disable` 设置连接 PostgreSQL 不启用加密
      --count int             总执行次数，0 表示无限次
  -D, --db string             被压测的数据库名 (默认为 "test")
  -d, --driver string         数据库驱动: mysql, postgres (默认 "mysql")
      --dropdata              在 prepare 数据之前清除历史数据
  -h, --help                  输出 bench 命令的帮助信息
  -H, --host strings          数据库的主机地址 (默认 ["127.0.0.1"])
      --ignore-error          忽略压测时数据库报出的错误
      --interval duration     两次报告输出时间的间隔 (默认 10s)
      --isolation int         隔离级别 0: Default, 1: ReadUncommitted,
                              2: ReadCommitted, 3: WriteCommitted, 4: RepeatableRead,
                              5: Snapshot, 6: Serializable, 7: Linerizable
      --max-procs int         Go Runtime 能够使用的最大系统线程数
      --output string         输出格式 plain, table, json (默认为 "plain")
  -p, --password string       数据库密码
```

```
  -P, --port ints              数据库端口（默认 [4000]）
      --pprof string           pprof 地址
      --silence                压测过程中不打印错误信息
  -S, --statusPort int         TiDB 状态端口（默认 10080）
  -T, --threads int            压测并发线程数（默认 16）
      --time duration          总执行时长（默认 2562047h47m16.854775807s）
  -U, --user string            压测时使用的数据库用户（默认 "root"）
```

- `--host` 和 `--port` 支持以逗号分隔传入多个值，以启用客户端负载均衡。例如，当指定 `--host 172.16.4.1,172.16.4.2 --port 4000,4001` 时，负载程序将以轮询调度的方式连接到 172.16.4.1:4000, 172.16.4.1:4001, 172.16.4.2:4000, 172.16.4.2:4001 这 4 个实例上。
- `--conn-params` 需要符合 query string 格式，不同数据库支持不同参数，如：
  - `--conn-params tidb_isolation_read_engines='tiflash'` 设置 TiDB 通过 TiFlash 进行查询。
  - `--conn-params sslmode=disable` 设置连接 PostgreSQL 不启用加密。
- 当运行 CH-benCHmark 时，可以通过 `--ap-host`, `--ap-port`, `--ap-conn-params` 来指定独立的 TiDB 实例用于 OLAP 查询。

下文分别介绍如何使用 TiUP 运行 TPC-C, TPC-H 以及 YCSB 测试。

13.4.11.4.1 使用 TiUP 运行 TPC-C 测试

TiUP bench 组件支持如下运行 TPC-C 测试的命令和参数：

```
Available Commands:
  check       检查数据一致性
  cleanup     清除数据
  prepare     准备数据
  run         开始压测

Flags:
      --check-all            运行所有的一致性检测
  -h, --help                 输出 TPC-C 的帮助信息
      --partition-type int   分区类型（默认为 1）
                             1 代表 HASH 分区类型
                             2 代表 RANGE 分区类型
                             3 代表 LIST 分区类型并按 HASH 方式划分
                             4 代表 LIST 分区类型并按 RANGE 方式划分
      --parts int            分区仓库的数量（默认为 1）
      --warehouses int       仓库的数量（默认为 10）
```

TPC-C 测试步骤

以下为简化后的关键步骤。完整的测试流程可以参考如何对 TiDB 进行 TPC-C 测试

1. 通过 HASH 使用 4 个分区创建 4 个仓库：

```
tiup bench tpcc --warehouses 4 --parts 4 prepare
```

2. 运行 TPC-C 测试：

```
tiup bench tpcc --warehouses 4 --time 10m run
```

3. 检查一致性：

```
tiup bench tpcc --warehouses 4 check
```

4. 清理数据：

```
tiup bench tpcc --warehouses 4 cleanup
```

当需要测试大数据集时，直接写入数据通常较慢，此时可以使用如下命令生成 CSV 数据集，然后通过TiDB Lightning 导入数据。

- 生成 CSV 文件：

```
shell tiup bench tpcc --warehouses 4 prepare --output-dir data --output-type=csv
```

- 为指定的表生成 CSV 文件：

```
shell tiup bench tpcc --warehouses 4 prepare --output-dir data --output-type=csv --tables history
↪ ,orders
```

13.4.11.4.2 使用 TiUP 运行 TPC-H 测试

TiUP bench 组件支持如下运行 TPC-H 测试的命令和参数：

```
Available Commands:
  cleanup     清除数据
  prepare     准备数据
  run         开始压测

Flags:
      --check            检查输出数据，只有 scale 因子为 1 时有效
  -h, --help             tpch 的帮助信息
      --queries string   所有的查询语句（默认 "q1,q2,q3,q4,q5,q6,q7,q8,q9,q10,q11,q12,q13,q14,q15
          ↪ ,q16,q17,q18,q19,q20,q21,q22")
      --sf int           scale 因子
```

13.4.11.4.3 TPC-H 测试步骤

1. 准备数据：

```
tiup bench tpch --sf=1 prepare
```

2. 运行 TPC-H 测试，根据是否检查结果执行相应命令：

   - 检查结果：

   ```
   tiup bench tpch --count=22 --sf=1 --check=true run
   ```

   - 不检查结果：

   ```
   tiup bench tpch --count=22 --sf=1 run
   ```

3. 清理数据：

```
tiup bench tpch cleanup
```

13.4.11.4.4 使用 TiUP 运行 YCSB 测试

你可以使用 TiUP 对 TiDB 和 TiKV 节点分别进行 YCSB 测试。

测试 TiDB

1. 准备数据：

```
tiup bench ycsb load tidb -p tidb.instances="127.0.0.1:4000" -p recordcount=10000
```

2. 运行 YCSB 测试：

```
# 默认读写比例为 95:5
tiup bench ycsb run tidb -p tidb.instances="127.0.0.1:4000" -p operationcount=10000
```

测试 TiKV

1. 准备数据：

```
tiup bench ycsb load tikv -p tikv.pd="127.0.0.1:2379" -p recordcount=10000
```

2. 运行 YCSB 测试：

```
# 默认读写比例为 95:5
tiup bench ycsb run tikv -p tikv.pd="127.0.0.1:2379" -p operationcount=10000
```

13.4.11.4.5 使用 TiUP 运行 RawSQL 测试

你可以将 OLAP 查询写到 SQL 文件中，通过 `tiup bench rawsql` 执行测试，步骤如下：

1. 准备数据和需要执行的查询：

```
-- 准备数据
CREATE TABLE t (a int);
INSERT INTO t VALUES (1), (2), (3);

-- 构造查询，保存为 demo.sql
SELECT a, sleep(rand()) FROM t WHERE a < 4*rand();
```

2. 运行 RawSQL 测试：

```
shell  tiup bench rawsql run --count 60 --query-files demo.sql
```

13.5 TiDB Operator

TiDB Operator 是 Kubernetes 上的 TiDB 集群自动运维系统，提供包括部署、升级、扩缩容、备份恢复、配置变更的 TiDB 全生命周期管理。借助 TiDB Operator，TiDB 可以无缝运行在公有云或自托管的 Kubernetes 集群上。

TiDB Operator 的文档目前独立于 TiDB 文档，文档名称为 TiDB on Kubernetes 用户文档。要访问 TiDB Operator 的文档，请点击以下链接：

- TiDB on Kubernetes 用户文档

13.6 TiDB Data Migration

13.6.1 TiDB Data Migration 简介

TiDB Data Migration (DM) 是一款便捷的数据迁移工具，支持从与 MySQL 协议兼容的数据库（MySQL、MariaDB、Aurora MySQL）到 TiDB 的全量数据迁移和增量数据同步。使用 DM 工具有利于简化数据迁移过程，降低数据迁移运维成本。

13.6.1.1 产品特性

- 与 TiDB 同样保持 MySQL 兼容性。高度兼容 MySQL 5.7 协议、MySQL 5.7 常用的功能及语法。
- 支持 DML & DDL 事件同步。支持解析和同步 binlog 中的 DML 和 DDL 事件。
- 支持合库合表同步模式。可以方便的将上游各个分片 MySQL 实例的各个分表数据，合并同步到下游 TiDB 的一张表。支持自定义编写同步规则以方便各种可能的同步需求，且具备自动识别和处理上游分片 MySQL 的 DDL 变更，大幅简化运维成本。
- 内置多种过滤器以灵活适应不同场景。支持以预定义事件类型、正则表达式、SQL 表达式等多种方式在数据同步过程中对 MySQL binlog 事件进行过滤。
- 集中管理。DM 支持上千个节点的集群规模，可同时运行并集中管理大量数据迁移同步任务。

- 对第三方 Online Schema Change 工具变更过程的同步优化。在 MySQL 生态中，gh-ost 与 pt-osc 等工具被广泛使用，DM 对其变更过程进行了特殊的优化，以避免对不必要的中间数据进行迁移。详细信息可参考online-ddl。
- 高可用。支持迁移任务在不同节点自由调度，少量工作节点宕机并不会影响进行中的任务。

13.6.1.2 快速安装

```
curl --proto '=https' --tlsv1.2 -sSf https://tiup-mirrors.pingcap.com/install.sh | sh
tiup install dm dmctl
```

13.6.1.3 使用限制

在使用 DM 工具之前，需了解以下限制：

- 数据库版本要求
  - MySQL 版本 5.5 ~ 5.7
  - MySQL 版本 = 8.0 （实验特性）
  - MariaDB 版本 >= 10.1.2 （实验特性）

> **注意：**
>
> 如果上游 MySQL/MariaDB servers 间构成主从复制结构，则需要 MySQL 版本高于 5.7.1 或者 MariaDB 版本等于或高于 10.1.3。

- DDL 语法兼容性限制
  - 目前，TiDB 部分兼容 MySQL 支持的 DDL 语句。因为 DM 使用 TiDB parser 来解析处理 DDL 语句，所以目前仅支持 TiDB parser 支持的 DDL 语法。详见TiDB DDL 语法支持。
  - DM 遇到不兼容的 DDL 语句时会报错。要解决此报错，需要使用 dmctl 手动处理，要么跳过该 DDL 语句，要么用指定的 DDL 语句来替换它。详见如何处理不兼容的 DDL 语句。
  - DM 不会将视图的 DDL 语句同步到下游的 TiDB 集群，也不会将针对视图的 DML 语句同步到下游。在该场景下，建议用户在下游 TiDB 集群中自行创建视图。
- GBK 字符集兼容性限制
  - DM 在 v5.4.0 之前不支持将 `charset=GBK` 的表迁移到 TiDB。

13.6.1.4 Contributing

欢迎参与 DM 开源项目并万分感谢您的贡献，可以查看 CONTRIBUTING.md 了解更多信息。

13.6.1.5 社区技术支持

您可以通过在线文档了解和使用 DM，如果您遇到无法解决的问题，可以选择以下途径之一联系我们。

- GitHub
- AskTUG

13.6.1.6 License

DM 遵循 Apache 2.0 协议，在 LICENSE 了解更多信息。

13.6.1.7 版本变更说明

在 v5.4 之前，DM 工具的文档独立于 TiDB 文档。要访问这些早期版本的 DM 文档，请点击以下链接：

- DM v5.3 文档
- DM v2.0 文档
- DM v1.0 文档

> 注意：
>
> - DM 的 GitHub 代码仓库已于 2021 年 10 月迁移至 pingcap/tiflow。如有任何关于 DM 的问题，请在 `pingcap/tiflow` 仓库提交，以获得后续支持。
> - 在较早版本中（v1.0 和 v2.0），DM 采用独立于 TiDB 的版本号。从 DM v5.3 起，DM 采用与 TiDB 相同的版本号。DM v2.0 的下一个版本为 DM v5.3。DM v2.0 到 v5.3 无兼容性变更，升级过程与正常升级无差异。

要快速了解 DM 的原理架构、适用场景，建议先观看下面的培训视频。注意本视频只作为学习参考，具体操作步骤和最新功能，请以文档内容为准。

13.6.2 Data Migration 架构

DM 主要包括三个组件：DM-master，DM-worker 和 dmctl。

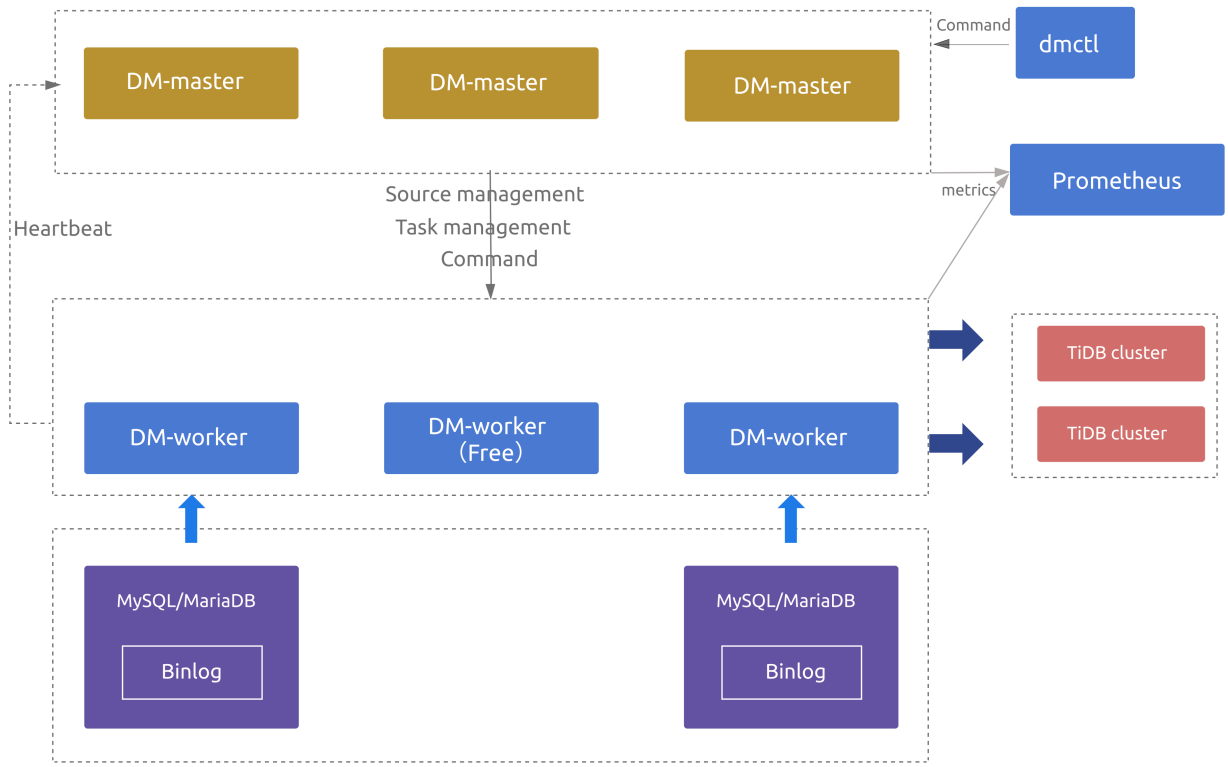


图 224: Data Migration architecture

13.6.2.1 架构组件

13.6.2.1.1 DM-master

DM-master 负责管理和调度数据迁移任务的各项操作。

- 保存 DM 集群的拓扑信息
- 监控 DM-worker 进程的运行状态
- 监控数据迁移任务的运行状态
- 提供数据迁移任务管理的统一入口
- 协调分库分表场景下各个实例分表的 DDL 迁移

13.6.2.1.2 DM-worker

DM-worker 负责执行具体的数据迁移任务。

- 将 binlog 数据持久化保存在本地
- 保存数据迁移子任务的配置信息
- 编排数据迁移子任务的运行
- 监控数据迁移子任务的运行状态

有关于 DM-worker 的更多介绍，详见DM-worker 简介。

13.6.2.1.3 dmctl

dmctl 是用来控制 DM 集群的命令行工具。

- 创建、更新或删除数据迁移任务
- 查看数据迁移任务状态
- 处理数据迁移任务错误
- 校验数据迁移任务配置的正确性

有关于 dmctl 的使用介绍，详见dmctl 使用。

13.6.2.2 架构特性

13.6.2.2.1 高可用

当部署多个 DM-master 节点时，所有 DM-master 节点将使用内部嵌入的 etcd 组成集群。该 DM-master 集群用于存储集群节点信息、任务配置等元数据，同时通过 etcd 选举出 leader 节点。该 leader 节点用于提供集群管理、数据迁移任务管理相关的各类服务。因此，若可用的 DM-master 节点数超过部署节点的半数，即可正常提供服务。

当部署的 DM-worker 节点数超过上游 MySQL/MariaDB 节点数时，超出上游节点数的相关 DM-worker 节点默认将处于空闲状态。若某个 DM-worker 节点下线或与 DM-master leader 发生网络隔离，DM-master 能自动将与原 DM-worker 节点相关的数据迁移任务调度到其他空闲的 DM-worker 节点上（若原 DM-worker 节点为网络隔离状态，则其会自动停止相关的数据迁移任务）；若无空闲的 DM-worker 节点可供调度，则原 DM-worker 相关的数据迁移任务将暂时挂起，直到有空闲 DM-worker 节点后自动恢复。

> 注意：
>
> 当数据迁移任务处于全量导出或导入阶段时，该迁移任务暂不支持高可用，主要原因为：
>
> - 对于全量导出，MySQL 暂不支持指定从特定快照点导出，也就是说数据迁移任务被重新调度或重启后，无法继续从前一次中断时刻继续导出。
> - 对于全量导入，DM-worker 暂不支持跨节点读取全量导出数据，也就是说数据迁移任务被调度到的新 DM-worker 节点无法读取调度发生前原 DM-worker 节点上的全量导出数据。

13.6.3 TiDB Data Migration 快速上手指南

本文介绍如何快速体验使用数据迁移工具 TiDB Data Migration (DM) 从 MySQL 迁移数据到 TiDB。此文档用于快速体验 DM 产品功能特性，并不建议适合在生产环境中使用。

13.6.3.1 第 1 步：部署 DM 集群

1. 安装 TiUP 工具并通过 TiUP 快速部署dmctl。

```
curl --proto '=https' --tlsv1.2 -sSf https://tiup-mirrors.pingcap.com/install.sh | sh
tiup install dm dmctl
```

2. 生成 DM 集群最小拓扑文件。

```
tiup dm template
```

3. 复制输出的配置信息，修改 IP 地址后保存为 `topology.yaml` 文件，使用 TiUP 部署 DM 集群。

```
tiup dm deploy dm-test 6.0.0 topology.yaml -p
```

13.6.3.2 第 2 步：准备数据源

你可以使用一个或多个 MySQL 实例作为上游数据源。

1. 为每一个数据源编写如下配置文件：

```
source-id: "mysql-01"

from:
  host: "127.0.0.1"
  user: "root"
  password: "fCxfQ9XKCezSzuCD0Wf5dUD+LsKegSg="  # 使用 tiup dmctl --encrypt "123456" 加密。
  port: 3306
```

2. 使用如下命令将数据源增加至 DM 集群。其中，`mysql-01.yaml` 是上一步编写的配置文件。

```
tiup dmctl --master-addr=127.0.0.1:8261 operate-source create mysql-01.yaml # --master-addr
    ↪ 填写 master_servers 其中之一。
```

若环境中并不存在可供测试的 MySQL 实例，可以使用以下方式通过 Docker 快速创建一个测试实例。步骤如下：

1. 编写 MySQL 配置文件：

```
mkdir -p /tmp/mysqltest && cd /tmp/mysqltest

cat > my.cnf <<EOF
[mysqld]
bind-address     = 0.0.0.0
character-set-server=utf8
collation-server=utf8_bin
default-storage-engine=INNODB
transaction-isolation=READ-COMMITTED
server-id        = 100
binlog_format    = row
```

```
log_bin          = /var/lib/mysql/mysql-bin.log
show_compatibility_56 = ON
EOF
```

2. 使用 Docker 启动 MySQL 实例：

```
docker run --name mysql-01 -v /tmp/mysqltest:/etc/mysql/conf.d -e MYSQL_ROOT_PASSWORD=my-
    ↪ secret-pw -d -p 3306:3306 mysql:5.7
```

3. 待 MySQL 启动后，即可连接该实例。

> **注意：**
> 该命令仅适用于体验数据迁移过程，不能用于生产环境和压力测试。

```
mysql -uroot -p -h 127.0.0.1 -P 3306
```

13.6.3.3 第 3 步：准备下游数据库

可以选择已存在的 TiDB 集群作为数据同步目标。

如果没有可以用于测试的 TiDB 集群，则使用以下命令快速构建演示环境。

```
tiup playground
```

13.6.3.4 第 4 步：准备测试数据

在一个或多个数据源中创建测试表和数据。如果你使用已存在的 MySQL 数据库，且数据库中已有可用数据，可跳过这一步。

```
drop database if exists `testdm`;
create database `testdm`;
use `testdm`;
create table t1 (id bigint, uid int, name varchar(80), info varchar(100), primary key (`id`),
    ↪ unique key(`uid`)) DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin;
create table t2 (id bigint, uid int, name varchar(80), info varchar(100), primary key (`id`),
    ↪ unique key(`uid`)) DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin;
insert into t1 (id, uid, name) values (1, 10001, 'Gabriel García Márquez'), (2, 10002, 'Cien años
    ↪  de soledad');
insert into t2 (id, uid, name) values (3, 20001, 'José Arcadio Buendía'), (4, 20002, 'Úrsula
    ↪ Iguarán'), (5, 20003, 'José Arcadio');
```

13.6.3.5 第 5 步：编写数据同步任务

1. 创建任务的配置文件 `testdm-task.yaml`：

```
name: testdm
task-mode: all

target-database:
  host: "127.0.0.1"
  port: 4000
  user: "root"
  password: "" # 如果密码不为空，则推荐使用经过 dmctl 加密的密文

# 填写一个或多个所需同步的数据源信息
mysql-instances:
  - source-id: "mysql-01"
    block-allow-list:  "ba-rule1"

block-allow-list:
  ba-rule1:
    do-dbs: ["testdm"]
```

2. 使用 dmctl 创建任务：

```
tiup dmctl --master-addr 127.0.0.1:8261 start-task testdm-task.yaml
```

这样就成功创建了一个将 `mysql-01` 数据源迁移到 TiDB 的任务。

13.6.3.6 第 6 步：查看迁移任务状态

在创建迁移任务之后，可以用 `dmctl query-status` 来查看任务的状态。

```
tiup dmctl --master-addr 127.0.0.1:8261 query-status testdm
```

13.6.4 DM 数据迁移最佳实践

TiDB Data Migration (DM) 是由 PingCAP 开发维护的数据迁移同步工具，主要支持的源数据库类型为各类遵循 MySQL 协议标准的关系型数据库，如 MySQL、 Percona MySQL、 MariaDB、 Amazon RDS for MySQL、 Amazon Aurora 等。

DM 的使用场景主要有：

- 从兼容 MySQL 的单一实例中全量和增量迁移数据到 TiDB
- 将小数据量（小于 1 TB）分库分表 MySQL 合并迁移数据到 TiDB
- 在业务数据中台、业务数据实时汇聚等数据中枢场景中，作为数据同步中间件来使用

本文档介绍了如何优雅高效地使用 DM，以及如何规避使用 DM 的常见误区。

13.6.4.1 性能边界定位

DM 的性能参数如下表所示。

参数	限制
最大同步节点（Work Nodes）	1000
最大同步任务数量	600
最大同步 QPS	30k QPS/worker
最大 Binlog 吞吐量	20 MB/s/worker
每个 Task 处理的表数量	无限制

- DM 支持同时管理 1000 个同步节点（Work Node），最大同步任务数量为 600 个。为了保证同步节点的高可用，应预留一部分同步节点作为备用节点。建议预留的节点数量为已开启同步任务的同步节点数量的 20% ~ 50%。
- 单机部署 Work Node 数量。在服务器配置较好情况下，要保证每个 Work Node 至少有 2 核 CPU 加 4G 内存的可用工作资源，并且应为主机预留 10% ~ 20% 的系统资源。
- 单个同步节点（Work Node），理论最大同步 QPS 为 30K QPS/worker（不同 Schema 和 workload 会有所差异），处理上游 Binlog 的能力最高为 20 MB/s/worker。
- 如果将 DM 作为需要长期使用的数据同步中间件，需要注意 DM 组件的部署架构。请参见DM-master 与 DM-woker 部署实践。

13.6.4.2 数据迁移前

在所有数据迁移之前，整体方案的设计是至关重要的。下面我们分别从业务侧要点及实施侧要点两个方面讲一下相关的实践经验和适用场景。

13.6.4.2.1 业务侧要点

为了让压力可以平均分配到多个节点上，在 Schema 设计上，分布式数据库与传统数据库差别很大，既要保证较低的业务迁移成本，又要保证迁移后应用逻辑的正确性。下面就从几个方面来看业务迁移前的最佳实践。

Schema 的设计中 AUTO_INCREMENT 对业务的影响

TiDB 的 AUTO_INCREMENT 与 MySQL 的 AUTO_INCREMENT 整体上看是相互兼容的。但因为 TiDB 作为分布式数据库，一般会有多个计算节点（client 端入口），应用数据写入时会将负载均分开，这就导致在有 AUTO_INCREMENT 列的表上，可能出现不连续的自增 ID。详细原理参考AUTO_INCREMENT。

如果业务对自增 ID 有强依赖，可以考虑使用SEQUENCE 函数。

是否使用聚簇索引

TiDB 在建表时可以声明为主键创建聚簇索引或非聚簇索引。下面介绍各方案的优势和劣势。

- 聚簇索引

  聚簇索引使用主键作为数据存储的 handle ID（行 ID），在使用主键查询时可以减少一次回表的操作，有效提升查询效能。但如果表有大量数据写入且主键使用AUTO_INCREMENT，非常容易造成数据存储的写入热点问题，导致集群整体效能不能充分利用，出现单存储节点的性能瓶颈问题。

- 非聚簇索引 + `shard row id bit`

  使用非聚簇索引，再配合表提示 `shard row id bit`，可以在继续使用 AUTO_INCREMENT 的情况下有效避免数据写入热点的产生。但是由于多了一次回表操作，此时使用主键查询数据，查询性能将有所影响。

- 聚簇索引 + 外部分布式发号器

  如果想使用聚簇索引，但还希望 ID 是单调递增的，那么可以考虑使用外部分布式发号器，如雪花算法 (Snowflake)、Leaf 等，来解决问题。由应用程序产生序列 ID，可以一定程度上保证 ID 的单调递增性，同时也保留了使用聚簇索引带来的收益。但相关应用程序需要进行改造。

- 聚簇索引 + AUTO_RANDOM

  此方案是目前分布式数据库既能避免出现写入热点问题，又能保留聚簇索引带来的查询收益的方案。整体改造也相对轻量，可以在业务切换使用 TiDB 作为写库时，修改 Schema 属性来达到目的。如果在后续查询时一定要利用 ID 列进行排序，可以使用AUTO_RANDOM ID 列左移 5 位来保证查询数据的顺序性。示例：

```
CREATE TABLE t (a bigint PRIMARY KEY AUTO_RANDOM, b varchar(255));
Select  a, a<<5 ,b from t order by a <<5 desc
```

下表汇总了不同使用场景的推荐方案和优劣势。

场景	推荐方案	优势	劣势
TiDB 未来作为主库使用，并会有大量数据写入。业务逻辑强依赖主键 ID 的连续性。	将表建立为非聚簇索引，并设置 SHARD_ROW_ID_BIT。使用 SEQUENCE 作为主键列。	可以有效避免数据写入热点，保证业务数据的连续性和单调递增。	数据写入的吞吐能力会下降 (为保证数据写入连续性); 主键查询性能有所下降。

场景	推荐方案	优势	劣势
TiDB 未来作为主库使用，并会有大量数据写入。业务逻辑强依赖主键 ID 的递增特性。	将表建立为非聚簇索引，并设置 SHARD_ROW_ID_BIT；使用应用程序发号器来定义主键 ID。	可以有效避免数据写入热点；可以保证数据写入性能；可以有效保证业务数据是趋势性递增，但不能保证数据连续性。	对原有代码有一定的改造成本；外部发号器对时钟准确性有强依赖，引入新的故障风险点。

场景	推荐方案	优势	劣势
TiDB 未来作为主库使用，并会有大量数据写入。业务逻辑不依赖主键 ID 的连续性。	将表建立为聚簇索引表；主键列设置为 AUTO_RANDOM 属性。	可以有效避免数据写入热点；有非常有限的写入吞吐能力；主键查询性能优异；可以平滑将 AUTO_INCREMENT 属性切换为 AUTO_RANDOM 属性。	主键 ID 是完全随机的，业务数据排序建议使用插入时间列来完成。如果一定要使用主键 ID 排序，可以用 ID 左移 5 的方式查询，此方式查询的数据可以保证趋势递增的特性。

场景	推荐方案	优势	劣势
TiDB 未来作为只读的数据中台使用。	将表建立为非聚簇索引，并设置 SHARD_ROW_ID_BIT；使主键列维持与源数据库类型一致即可。	可以有效避免数据写入热点；改造成本低。	主键查询性能有所下降。

13.6.4.2.2 分库分表要点

分与合

DM 支持将上游分库分表的数据合并到下游 TiDB 中的同一个表，这也是 TiDB 推荐的一种方式。

除了数据合并场景外，另一个典型场景为 数据归档场景。在此场景中，数据不断写入，随着时间流逝，大量的数据从热数据逐渐转变为温冷数据。在 TiDB 中，你可以通过Placement Rules 放置规则来按照一定规则对数据设置不同的放置规则，而最小粒度即为分区表 (Partition)。

所以建议在遇到有大规模数据写入的场景，一开始就规划好未来是否需要归档或者有冷热数据分别存储在不同介质的需要。如果有，那么在迁移前请设置好分区表规则（目前 TiDB 还不支持 Table Rebuild 操作）。避免因为初期考虑不周，导致后期需要重新建表及重新导入数据。

悲观 DDL 锁与乐观 DDL 锁

DM 默认会使用悲观 DDL 锁模式。在分库分表迁移与同步场景中，上游相关分表发生 Schema 变更后，会阻断后续的 DML 向下游 TiDB 写入。此时需要等待上游各分表的 Schema 都变更完毕并自动确认所有分表的结构一致后，从同步阻断记录点继续数据同步。

- 如果上游 Schema 变更时间较长，可能导致上游 Binlog 被清理，此问题可以通过开启 DM 的 Relay log 功能来避免。
- 如果不希望因为上游 Schema 变更阻塞数据写入，可以考虑使用乐观 DDL 锁模式。此时 DM 在发现上游分表 Schema 变更时也不会阻断数据同步，而是会持续同步数据。但如果同步期间 DM 发现上下游数据格式不兼容，将停止同步任务。此时需要人工介入处理。

下表汇总了乐观 DDL 锁和悲观 DDL 锁的优劣势。

场景	优势	劣势
悲观 DDL 锁（默认）	最大程度保证数据同步任务的可靠性	如果分表较多，将长时间阻断数据同步任务。并有可能因为上游的 Binlog 已被清理而导致同步中断。可以通过开启 DM-worker 的 Relay log 来避免问题。详情请参考Relay log 的使用 .

场景	优势	劣势
乐观 DDL 锁	数据同步任务基本不会出现相关阻塞延迟	此模式下需要保证 Schema 变更的兼容性（增加列是否具有默认值）。如果考虑不周，可能出现未被发现的上下游数据不一致问题。更多限制，请参考乐观模式下分库分表合并迁移。

13.6.4.2.3 其他限制与影响

上下游的数据类型

这里主要需要考虑上下游的数据类型问题。TiDB 目前支持绝大部分 MySQL 的数据类型。但一些特殊类型尚不支持（如空间类型）。关于数据类型的兼容性，请参考数据类型概述。

字符集与排序规则

自 TiDB v6.0.0 以后，默认使用新排序规则。如果需要 TiDB 支持 utf8_general_ci、 utf8mb4_general_ci、 utf8_unicode_ci、utf8mb4_unicode_ci、gbk_chinese_ci 和 gbk_bin 这几种排序规则，需要在集群创建时声明，将 new_collations_enabled_on_first_bootst
↪ 的值设为 true。更详细信息请参考字符集和排序规则。

TiDB 默认使用的字符集为 utf8mb4。建议同步上下游及应用统一使用 utf8mb4。如果上游有显式指定的字符集或者排序规则，需要确认 TiDB 是否支持。

从 v6.0.0 起，TiDB 支持 GBK 字符集。有关字符集的限制详见：

- 字符集和排序规则
- GBK 兼容情况

13.6.4.2.4 实施侧要点

DM-master 与 DM-woker 部署实践

DM 整体架构分为 DM-master 与 DM-worker。

- DM-master 主要负责同步任务的元数据管理，以及 DM-worker 的中心调度，是整个 DM 平台的核心。所以 DM-master 可以部署为集群模式，以保证 DM 同步平台的可用性。
- DM-worker 负责执行上下游同步任务，是无状态节点。最多可以部署 1000 个节点。在需要将 DM 作为数据同步平台的场景，可以预留一部分空闲的 DM-worker，以保证同步任务的高可用。

同步任务规划

分库分表场景。在分库分表迁移场景，根据上游分库分表的种类进行同步任务的拆分，如 usertable_1~50 和 Logtable_1~50 是两类分表，那么就应该分别建 2 个 Task 任务进行同步。这样做的好处是可有效简化同步任务模板的复杂度，并有效控制数据同步中断的影响范围。

大规模数据迁移同步场景。可以参考以下思路进行 Task 任务拆分：

- 如果上游需要同步多个数据库，可以按照不同数据库拆分 Task。
- 根据上游写入压力拆分任务。即把上游 DML 操作频繁的表，拆分到单独的 Task 任务中，将其他没有频繁 DML 操作的表使用另一个 Task 任务进行同步。此方式可在一定程度上加速同步任务的推进能力。尤其是在上游有大量 Log 写入某张表，但业务关注的是其他表时，此方法可以有效解决此类问题。

请注意，拆分同步任务后只能保证数据同步的最终一致性，实时一致性因各种原因可能出现较大偏差。

下表给出了在不同的数据迁移与同步场景下部署 DM-master 与 DM-worker 的推荐方案。

场景	DM-master 部署	DM-worker 部署
小规模数据（1 TB 以下），一次性数据迁移场景	部署 1 个 DM-master 节点	根据上游数据源数量，部署 1 ~ N 个 DM-worker 节点。一般情况下 1 个 DM-worker 节点。
大规模数据（1 TB 以上）及分库分表，一次性数据迁移场景	推荐部署 3 个 DM-master 节点，来保证在长时间数据迁移时 DM 集群的可用性	根据数据源数量或同步任务数量部署 DM-Worker 节点。推荐多部署 1~3 个空闲 DM-Worker 节点。

场景	DM-master 部署	DM-worker 部署
长期数据同步迁移场景	务必部署 3 个 DM-master 节点。如在云上部署，尽量将 DM-master 部署在不同的可用区（AZ）	根据数据源数量或同步任务数量部署 DM-worker 节点。务必部署实际需要 DM-worker 节点数量的 1.5 ~ 2 倍的 DM-worker 节点数量。

上游数据源选择与设置

DM 支持存量数据迁移，但在做全量迁移时会对整库进行全量数据备份。DM 采用的备份方式为并行逻辑备份，备份 MySQL 期间会加上全局只读锁 `FLUSH TABLES WITH READ LOCK`。此时会短暂地阻塞上游数据库的 DML 和 DDL 操作。所以强烈建议使用上游的备库来进行全量备份，并同时在数据源开启 GTID 的功能 (`enable-gtid: true`)。这样既可避免存量迁移时对上游业务的影响，也可以在增量同步期间再切换到上游主库节点降低数据同步的延迟。切换上游 MySQL 数据源的方法，请参考切换 DM-worker 与上游 MySQL 实例的连接。

下面是一些特殊场景下的注意事项：

- 只能在上游主库进行全量备份

  在该场景中，可以在同步任务配置中设置一致性参数为 `none`，`mydumpers.global.extra-args: "--consistency none"` 避免给主库加全局只读锁，但有可能破坏全量备份的数据一致性，导致最终上下游数据不一致。

- 利用备份快照解决存量迁移（只适用 AWS 上 MySQL RDS 和 Aurora RDS 的迁移）

  如果要迁移的数据库正好为 AWS MySQL RDS 或者 Aurora RDS，可以利用 RDS Snapshot 备份将 Amazon S3 中的备份数据直接迁移到 TiDB，以此保证存量数据迁移的一致性。整个操作流程以及后续增量同步方法，请参考从 Amazon Aurora 迁移数据到 TiDB。

13.6.4.2.5 配置细节详解

大小写

TiDB 默认情况下对 Schema name 大小写不敏感，即 `lower_case_table_names:2`。但上游 MySQL 大多为 Linux 系统，默认对大小写敏感。此时需要注意，在 DM 数据同步任务设置时将 `case-sensitive` 设置为 `true`，保证可以正确同步上游的 Schema。

特殊情况下，比如上游一个数据库中，既有大写表如 `Table`，又有小写表如 `table`，那么 Schema 创建时将报错:

```
ERROR 1050 (42S01): Table '{tablename}' already exists
```

过滤规则

在配置数据源时即可配置过滤规则。配置方法请参考数据迁移任务配置向导。

配置过滤规则的好处有:

- 可以减少下游处理 Binglog Event 的数量，提升同步效能
- 可以减少不必要的 Relay log 的落盘，节约磁盘空间

> 注意:
>
> 在分库分表场景中，如果你在数据源配置了过滤规则，请确保数据源与同步任务中设置的过滤规则相匹配。如果不匹配，将会导致同步任务长期接收不到增量数据。

Relay Log 的使用

MySQL 的主从复制在 Secondary 端会保存一份 Relay log，以此保证异步复制的可靠性与效能。DM 也支持在 DM-worker 侧保存一份 Relay log，并可设置存储位置、过期清理时间等信息。此功能适用于以下场景:

- 在进行全量 + 增量数据迁移时，因为全量迁移数据量较大，整个过程耗费时间超过了上游 Binlog 归档的时间，导致增量同步任务不能正常启动，如果开启 Relay Log 在全量同步启动同时，DM-worker 即会开始接收 Relay log，避免增量任务启动失败。
- 在使用 DM 进行长期数据同步的场景中，有时因为各种原因导致同步任务长时间阻塞，此时开启了 Relay log 功能，可以有效应对同步任务阻塞而导致的上游 Binglog 被回收问题。

在使用 Relay Log 功能时也会有一定的限制。DM 支持高可用，当某个 DM-worker 出现故障，会尝试将空闲的 DM-worker 实例提升为工作实例，如果此时上游 Binlog 没有包含必要的同步日志，将可能出现同步中断情况。此时需要人工干预，尽快将 Relay log 复制到新的 DM-worker 节点上来，并修改相应的 Relay meta 文件。具体方法请参考故障处理。

上游使用在线变更工具 PT-osc/GH-ost

在日常运维 MySQL 时，想要在线变更表结构，一般会使用 PT-osc/GH-ost 这类工具，以此保证 DDL 变更对线上业务的影响最小。但整个过程会被如实地记录到 MySQL Binlog 中，如果全部同步到下游 TiDB，将产生大量的写放大，既不高效也不经济。DM 在设置 Task 任务时候可以设置支持三方数据同步工具 PT-osc 或 GH-ost，配置后将不再同步大量冗余数据，并且能保证数据同步的一致性。具体设置方式请参考迁移使用 GH-ost/PT-osc 的源数据库。

13.6.4.3 数据迁移中

在这个环节基本上遇到的问题都是 Troubleshooting 类的问题，在这里给大家列举一些场景的问题及处理方式。

13.6.4.3.1 上游与下游 Schema 不一致

常见报错信息：

- `messages: Column count doesn't match value count: 3 (columns)vs 2 (values)`
- `Schema/Column doesn't match`

此类问题主要原因是下游 TiDB 中增加或修改了索引，或者下游比上游更多列。当出现此类的同步报错信息时，请检查是否是上下游 Schema 不一致导致的同步中断。

要解决此类问题，只需将 DM 中缓存的 Schema 信息更新成与下游 TiDB Schema 一致即可。具体方法参考管理迁移表的表结构。

如果是下游比上游多列的场景，请参考下游存在更多列的迁移场景。

13.6.4.3.2 处理因为 DDL 中断的数据同步任务

DM 支持跳过或者替换导致同步任务中断的 DDL 语句。并且针对是否为分库分表合并场景有对应不同的操作，具体请参考处理出错的 DDL 语句。

13.6.4.4 数据迁移后的数据校验

在完成数据迁移后，建议对新旧数据进行数据一致性校验。TiDB 提供了相应的同步工具sync-diff-inspector 来帮助你完成数据校验工作。

通过管理 DM 中的同步任务，sync-diff-inspector 可以自动管理需要进行数据一致性检查的 Table 列表，相较之前的手动配置更加的高效。具体参考基于 DM 同步场景下的数据校验。

自 DM v6.2.0 版本开始，DM 支持在增量同步的同时进行数据校验。具体参考DM 增量数据校验。

13.6.4.5 数据长期同步

如果将 DM 作为持续的数据同步平台，建议一定要做好必要的元信息备份。一方面是保证同步集群故障重建的能力，另一方面可以实现同步任务的版本控制。具体实现方式参考导出和导入集群的数据源和任务配置。

13.6.5 部署 DM 集群

13.6.5.1 TiDB Data Migration 集群软硬件环境需求

TiDB Data Migration (DM) 支持主流的 Linux 操作系统，具体版本要求见下表：

Linux 操作系统	版本
Red Hat Enterprise Linux	7.3 及以上
CentOS	7.3 及以上
Oracle Enterprise Linux	7.3 及以上
Ubuntu LTS	16.04 及以上

DM 可以在 Intel 架构服务器环境及主流虚拟化环境中部署和运行。

13.6.5.1.1 服务器建议配置

DM 支持部署和运行在 Intel x86-64 架构的 64 位通用硬件服务器平台。对于开发，测试，及生产环境的服务器硬件配置（不包含操作系统本身的占用）有以下要求和建议：

开发及测试环境

组件	CPU	内存	本地存储	网络	实例数量（最低要求）
DM-master	4 核 +	8 GB+	SAS，200 GB+	千兆网卡	1
DM-worker	8 核 +	16 GB+	SAS，200 GB+（大于迁移数据的大小）	千兆网卡	上游 MySQL 实例的数量

注意：

- 在功能验证的测试环境中的 DM-master 和 DM-worker 可以部署在同一台服务器上。
- 如进行性能相关的测试，避免采用低性能存储和网络硬件配置，防止对测试结果的正确性产生干扰。
- 如果仅验证功能，可以单机部署一个 DM-master，DM-worker 部署的数量至少是上游 MySQL 实例的数量。为了保证高可用性，建议部署更多的 DM-worker。
- DM-worker 在 dump 和 load 阶段需要存储全量数据，因此 DM-worker 的磁盘空间需要大于需要迁移数据的总量；如果迁移任务开启了 relay log，DM-worker 也需要一定的磁盘空间来存储上游的 binlog 数据。

生产环境

组件	CPU	内存	硬盘类型	网络	实例数量（最低要求）
DM-master	4 核 +	8 GB+	SAS，200 GB+	千兆网卡	3
DM-worker	16 核 +	32 GB+	SSD，200 GB+（大于迁移数据的大小）	万兆网卡	大于上游 MySQL 实例的数量
监控	8 核 +	16 GB+	SAS，200 GB+	千兆网卡	1

注意：

- 在生产环境中，不建议将 DM-master 和 DM-worker 部署和运行在同一个服务器上，以防 DM-worker 对磁盘的写入干扰 DM-master 高可用组件使用磁盘。
- 在遇到性能问题时可参照配置调优尝试修改任务配置。调优效果不明显时，可以尝试升级服务器配置。

13.6.5.1.2 下游数据库所需空间

目标 TiKV 集群必须有足够空间接收新导入的数据。除了标准硬件配置以外，目标 TiKV 集群的总存储空间必须大于 数据源大小 × 副本数量 × 2。例如集群默认使用 3 副本，那么总存储空间需为数据源大小的 6 倍以上。公式中的 2 倍可能难以理解，其依据是以下因素的估算空间占用：

- 索引会占据额外的空间
- RocksDB 的空间放大效应

可以用下面 SQL 语句统计信息表的 data_length 字段估算数据量：

统计所有 schema 大小，单位 MiB，注意修改 ${schema_name}

```
select table_schema,sum(data_length)/1024/1024 as data_length,sum(index_length)/1024/1024 as
    ↪ index_length,sum(data_length+index_length)/1024/1024 as sum from information_schema.
    ↪ tables where table_schema = "${schema_name}" group by table_schema;
```

统计最大单表，单位 MiB，注意修改 ${schema_name}

```
select table_name,table_schema,sum(data_length)/1024/1024 as data_length,sum(index_length)
    ↪ /1024/1024 as index_length,sum(data_length+index_length)/1024/1024 as sum from
    ↪ information_schema.tables where table_schema = "${schema_name}" group by table_name,
    ↪ table_schema order by sum  desc limit 5;
```

13.6.5.2 使用 TiUP 部署 DM 集群

TiUP 是 TiDB 4.0 版本引入的集群运维工具，TiUP DM 是 TiUP 提供的使用 Golang 编写的集群管理组件，通过 TiUP DM 组件就可以进行日常的运维工作，包括部署、启动、关闭、销毁、扩缩容、升级 DM 集群以及管理 DM 集群参数。

目前 TiUP 可以支持部署 v2.0 及以上版本的 DM。本文将介绍不同集群拓扑的具体部署步骤。

注意：

如果部署机器的操作系统支持 SELinux，请确保 SELinux 处于关闭状态。

13.6.5.2.1 前提条件

当 DM 执行全量数据复制任务时，每个 DM-worker 只绑定一个上游数据库。DM-worker 首先在上游导出全部数据，然后将数据导入下游数据库。因此，DM-worker 的主机空间需要容纳所有要导出的上游表，具体存储路径在后续创建迁移任务时指定。

另外，部署 DM 集群需参照DM 集群软硬件环境需求，满足相应要求。

13.6.5.2.2 第 1 步：在中控机上安装 TiUP 组件

使用普通用户登录中控机，以 tidb 用户为例，后续安装 TiUP 及集群管理操作均通过该用户完成：

1. 执行如下命令安装 TiUP 工具：

```
curl --proto '=https' --tlsv1.2 -sSf https://tiup-mirrors.pingcap.com/install.sh | sh
```

安装完成后，~/.bashrc 已将 TiUP 加入到路径中，你需要新开一个终端或重新声明全局变量 source ~/.bashrc 来使用 TiUP。

2. 安装 TiUP DM 组件：

```
tiup install dm dmctl
```

13.6.5.2.3 第 2 步：编辑初始化配置文件

请根据不同的集群拓扑，编辑 TiUP 所需的集群初始化配置文件。

请根据配置文件模板，新建一个配置文件 topology.yaml。如果有其他组合场景的需求，请根据多个模板自行调整。

可以使用 tiup dm template > topology.yaml 命令快速生成配置文件模板。

部署 3 个 DM-master、3 个 DM-worker 与 1 个监控组件的配置如下：

```
#全局变量适用于配置中的其他组件。如果组件实例中缺少一个特定值，则相应的全局变量将用作默认值。
global:
  user: "tidb"
  ssh_port: 22
  deploy_dir: "/dm-deploy"
  data_dir: "/dm-data"

server_configs:
  master:
    log-level: info
    # rpc-timeout: "30s"
    # rpc-rate-limit: 10.0
    # rpc-rate-burst: 40
  worker:
    log-level: info
```

```
master_servers:
  - host: 10.0.1.11
    name: master1
    ssh_port: 22
    port: 8261
    # peer_port: 8291
    # deploy_dir: "/dm-deploy/dm-master-8261"
    # data_dir: "/dm-data/dm-master-8261"
    # log_dir: "/dm-deploy/dm-master-8261/log"
    # numa_node: "0,1"
    # 下列配置项用于覆盖 `server_configs.master` 的值。
    config:
      log-level: info
      # rpc-timeout: "30s"
      # rpc-rate-limit: 10.0
      # rpc-rate-burst: 40
  - host: 10.0.1.18
    name: master2
    ssh_port: 22
    port: 8261
  - host: 10.0.1.19
    name: master3
    ssh_port: 22
    port: 8261
#### 如果不需要确保 DM 集群高可用，则可只部署 1 个 DM-master 节点，且部署的 DM-worker
    ↪ 节点数量不少于上游待迁移的 MySQL/MariaDB 实例数。
#### 如果需要确保 DM 集群高可用，则推荐部署 3 个 DM-master 节点，且部署的 DM-worker
    ↪ 节点数量大于上游待迁移的 MySQL/MariaDB 实例数（如 DM-worker 节点数量比上游实例数多 2 个
    ↪ ）。
worker_servers:
  - host: 10.0.1.12
    ssh_port: 22
    port: 8262
    # deploy_dir: "/dm-deploy/dm-worker-8262"
    # log_dir: "/dm-deploy/dm-worker-8262/log"
    # numa_node: "0,1"
    # 下列配置项用于覆盖 `server_configs.worker` 的值。
    config:
      log-level: info
  - host: 10.0.1.19
    ssh_port: 22
    port: 8262

monitoring_servers:
```

```
  - host: 10.0.1.13
    ssh_port: 22
    port: 9090
    # deploy_dir: "/tidb-deploy/prometheus-8249"
    # data_dir: "/tidb-data/prometheus-8249"
    # log_dir: "/tidb-deploy/prometheus-8249/log"

grafana_servers:
  - host: 10.0.1.14
    port: 3000
    # deploy_dir: /tidb-deploy/grafana-3000

alertmanager_servers:
  - host: 10.0.1.15
    ssh_port: 22
    web_port: 9093
    # cluster_port: 9094
    # deploy_dir: "/tidb-deploy/alertmanager-9093"
    # data_dir: "/tidb-data/alertmanager-9093"
    # log_dir: "/tidb-deploy/alertmanager-9093/log"
```

注意：

- 不建议在一台主机上运行太多 DM-worker。每个 DM-worker 至少应有 2 核 CPU 和 4 GiB 内存。
- 需要确保以下组件间端口可正常连通：
  - 各 DM-master 节点间的 peer_port（默认为 8291）可互相连通。
  - 各 DM-master 节点可连通所有 DM-worker 节点的 port（默认为 8262）。
  - 各 DM-worker 节点可连通所有 DM-master 节点的 port（默认为 8261）。
  - TiUP 节点可连通所有 DM-master 节点的 port（默认为 8261）。
  - TiUP 节点可连通所有 DM-worker 节点的 port（默认为 8262）。

更多 master_servers.host.config 参数说明，请参考 master parameter；更多 worker_servers.host.config 参数说明，请参考 worker parameter。

13.6.5.2.4 第 3 步：执行部署命令

注意：

通过 TiUP 进行集群部署可以使用密钥或者交互密码方式来进行安全认证：

- 如果是密钥方式，可以通过 -i 或者 --identity_file 来指定密钥的路径；
- 如果是密码方式，可以通过 -p 进入密码交互窗口；
- 如果已经配置免密登录目标机，则不需填写认证。

```
tiup dm deploy dm-test ${version} ./topology.yaml --user root [-p] [-i /home/root/.ssh/gcp_rsa]
```

以上部署命令中：

- 通过 TiUP DM 部署的集群名称为 dm-test。
- 部署版本为 ${version}，可以通过执行 tiup list dm-master 来查看 TiUP 支持的最新版本。
- 初始化配置文件为 topology.yaml。
- --user root：通过 root 用户登录到目标主机完成集群部署，该用户需要有 ssh 到目标机器的权限，并且在目标机器有 sudo 权限。也可以用其他有 ssh 和 sudo 权限的用户完成部署。
- -i 及 -p：非必选项，如果已经配置免密登录目标机，则不需填写，否则选择其一即可。-i 为可登录到目标机的 root 用户（或 --user 指定的其他用户）的私钥，也可使用 -p 交互式输入该用户的密码。
- TiUP DM 使用内置的 SSH 客户端，如需使用系统自带的 SSH 客户端，请参考 TiUP DM 文档中使用中控机系统自带的 SSH 客户端连接集群章节进行设置。

预期日志结尾输出会有 Deployed cluster `dm-test` successfully 关键词，表示部署成功。

13.6.5.2.5 第 4 步：查看 TiUP 管理的集群情况

```
tiup dm list
```

TiUP 支持管理多个 DM 集群，该命令会输出当前通过 TiUP DM 管理的所有集群信息，包括集群名称、部署用户、版本、密钥信息等：

```
Name  User  Version  Path                                       PrivateKey
----  ----  -------  ----                                       ----------
dm-test  tidb  ${version}  /root/.tiup/storage/dm/clusters/dm-test  /root/.tiup/storage/dm/
    ↪ clusters/dm-test/ssh/id_rsa
```

13.6.5.2.6 第 5 步：检查部署的 DM 集群情况

例如，执行如下命令检查 dm-test 集群情况：

```
tiup dm display dm-test
```

预期输出包括 dm-test 集群中实例 ID、角色、主机、监听端口和状态（由于还未启动，所以状态为 Down/inactive）、目录信息。

13.6.5.2.7 第 6 步：启动集群

```
tiup dm start dm-test
```

预期结果输出 Started cluster `dm-test` successfully 表示启动成功。

13.6.5.2.8 第 7 步：验证集群运行状态

通过以下 TiUP 命令检查集群状态：

```
tiup dm display dm-test
```

在输出结果中，如果 Status 状态信息为 Up，说明集群状态正常。

13.6.5.2.9 第 8 步：使用 dmctl 管理迁移任务

dmctl 是用来控制集群运行命令的工具，推荐通过 TiUP 获取该工具。

dmctl 支持命令模式与交互模式，具体请见使用 dmctl 运维集群。

13.6.5.3 使用 TiUP 离线镜像部署 DM 集群

本文介绍如何使用 TiUP 离线部署 DM 集群，具体的操作步骤如下。

13.6.5.3.1 第 1 步：准备 TiUP 离线组件包

- 在线环境中安装 TiUP 包管理器工具。

  1. 执行如下命令安装 TiUP 工具：

     ```
     curl --proto '=https' --tlsv1.2 -sSf https://tiup-mirrors.pingcap.com/install.sh | sh
     ```

  2. 重新声明全局环境变量：

     ```
     source .bash_profile
     ```

  3. 确认 TiUP 工具是否安装：

     ```
     which tiup
     ```

- 使用 TiUP 制作离线镜像。

  1. 在一台和外网相通的机器上拉取需要的组件：

     ```
     # 将 ${version} 修改成实际需要的版本
     tiup mirror clone tidb-dm-${version}-linux-amd64 --os=linux --arch=amd64 \
         --dm-master=${version} --dm-worker=${version} --dmctl=${version} \
         --alertmanager=v0.17.0 --grafana=v4.0.3 --prometheus=v4.0.3 \
         --tiup=v$(tiup --version|grep 'tiup'|awk -F ' ' '{print $1}') --dm=v$(tiup --
             ↪ version|grep 'tiup'|awk -F ' ' '{print $1}')
     ```

     该命令会在当前目录下创建一个名叫 `tidb-dm-${version}-linux-amd64` 的目录，里面包含 TiUP 管理的组件包。

  2. 通过 tar 命令将该组件包打包然后发送到隔离环境的中控机：

     ```
     tar czvf tidb-dm-${version}-linux-amd64.tar.gz tidb-dm-${version}-linux-amd64
     ```

     此时，`tidb-dm-${version}-linux-amd64.tar.gz` 就是一个独立的离线环境包。

13.6.5.3.2 第 2 步: 部署离线环境 TiUP 组件

将离线包发送到目标集群的中控机后，执行以下命令安装 TiUP 组件：

```
#### 将 ${version} 修改成实际需要的版本
tar xzvf tidb-dm-${version}-linux-amd64.tar.gz
sh tidb-dm-${version}-linux-amd64/local_install.sh
source /home/tidb/.bash_profile
```

local_install.sh 脚本会自动执行 `tiup mirror set tidb-dm-${version}-linux-amd64` 命令将当前镜像地址设置为 tidb-dm-${version}-linux-amd64。

若需将镜像切换到其他目录，可以通过手动执行 `tiup mirror set <mirror-dir>` 进行切换，切换后如果再想切换成官方镜像，可执行 `tiup mirror set https://tiup-mirrors.pingcap.com`。

13.6.5.3.3 第 3 步：编辑初始化配置文件

请根据不同的集群拓扑，编辑 TiUP 所需的集群初始化配置文件。

请根据配置文件模板，新建一个配置文件 topology.yaml。如果有其他组合场景的需求，请根据多个模板自行调整。

部署 3 个 DM-master、3 个 DM-worker 与 1 个监控组件的配置如下：

```
---
global:
  user: "tidb"
  ssh_port: 22
  deploy_dir: "/home/tidb/dm/deploy"
  data_dir: "/home/tidb/dm/data"
  # arch: "amd64"

master_servers:
  - host: 172.19.0.101
  - host: 172.19.0.102
  - host: 172.19.0.103

worker_servers:
  - host: 172.19.0.101
  - host: 172.19.0.102
  - host: 172.19.0.103

monitoring_servers:
  - host: 172.19.0.101

grafana_servers:
  - host: 172.19.0.101

alertmanager_servers:
```

```
  - host: 172.19.0.101
```

注意：

- 如果不需要确保 DM 集群高可用，则可只部署 1 个 DM-master 节点，且部署的 DM-worker 节点数量不少于上游待迁移的 MySQL/MariaDB 实例数。
- 如果需要确保 DM 集群高可用，则推荐部署 3 个 DM-master 节点，且部署的 DM-worker 节点数量大于上游待迁移的 MySQL/MariaDB 实例数（如 DM-worker 节点数量比上游实例数多 2 个）。
- 对于需要全局生效的参数，请在配置文件中 `server_configs` 的对应组件下配置。
- 对于需要某个节点生效的参数，请在具体节点的 `config` 中配置。
- 配置的层次结构使用 `.` 表示。如：`log.slow-threshold`。更多格式说明，请参考 TiUP 配置参数模版。
- 更多参数说明，请参考 master `config.toml.example`、worker `config.toml.example`。
- 需要确保以下组件间端口可正常连通：
  - 各 DM-master 节点间的 `peer_port`（默认为 8291）可互相连通。
  - 各 DM-master 节点可连通所有 DM-worker 节点的 `port`（默认为 8262）。
  - 各 DM-worker 节点可连通所有 DM-master 节点的 `port`（默认为 8261）。
  - TiUP 节点可连通所有 DM-master 节点的 `port`（默认为 8261）。
  - TiUP 节点可连通所有 DM-worker 节点的 `port`（默认为 8262）。

13.6.5.3.4 第 4 步：执行部署命令

注意：

通过 TiUP 进行集群部署可以使用密钥或者交互密码方式来进行安全认证：

- 如果是密钥方式，可以通过 `-i` 或者 `--identity_file` 来指定密钥的路径；
- 如果是密码方式，可以通过 `-p` 进入密码交互窗口；
- 如果已经配置免密登录目标机，则不需填写认证。

```
tiup dm deploy dm-test ${version} ./topology.yaml --user root [-p] [-i /home/root/.ssh/gcp_rsa]
```

以上部署命令中：

- 通过 TiUP DM 部署的集群名称为 `dm-test`。
- 部署版本为 `${version}`，可以通过执行 `tiup list dm-master` 来查看 TiUP 支持的最新版本。

- 初始化配置文件为 topology.yaml。
- --user root：通过 root 用户登录到目标主机完成集群部署，该用户需要有 ssh 到目标机器的权限，并且在目标机器有 sudo 权限。也可以用其他有 ssh 和 sudo 权限的用户完成部署。
- -i 及 -p：非必选项，如果已经配置免密登录目标机，则不需填写，否则选择其一即可。-i 为可登录到目标机的 root 用户（或 --user 指定的其他用户）的私钥，也可使用 -p 交互式输入该用户的密码。
- TiUP DM 使用内置的 SSH 客户端，如需使用系统自带的 SSH 客户端，请参考 TiUP DM 文档中使用中控机系统自带的 SSH 客户端连接集群章节进行设置。

预期日志结尾输出会有 Deployed cluster `dm-test` successfully 关键词，表示部署成功。

13.6.5.3.5 第 5 步：查看 TiUP 管理的集群情况

```
tiup dm list
```

TiUP 支持管理多个 DM 集群，该命令会输出当前通过 TiUP DM 管理的所有集群信息，包括集群名称、部署用户、版本、密钥信息等：

```
Name  User  Version  Path                                       PrivateKey
----  ----  -------  ----                                       ----------
dm-test  tidb  ${version}  /root/.tiup/storage/dm/clusters/dm-test  /root/.tiup/storage/dm/
    ↪ clusters/dm-test/ssh/id_rsa
```

13.6.5.3.6 第 6 步：检查部署的 DM 集群情况

例如，执行如下命令检查 dm-test 集群情况：

```
tiup dm display dm-test
```

预期输出包括 dm-test 集群中实例 ID、角色、主机、监听端口和状态（由于还未启动，所以状态为 Down/inactive）、目录信息。

13.6.5.3.7 第 7 步：启动集群

```
tiup dm start dm-test
```

预期结果输出 Started cluster `dm-test` successfully 表示启动成功。

13.6.5.3.8 第 8 步：验证集群运行状态

通过以下 TiUP 命令检查集群状态：

```
tiup dm display dm-test
```

在输出结果中，如果 Status 状态信息为 Up，说明集群状态正常。

13.6.5.4 使用 DM binary 部署 DM 集群

本文将介绍如何使用 DM binary 快速部署 DM 集群。

> 注意：
> 对于生产环境，推荐使用 TiUP 部署 DM 集群及相关监控组件。

13.6.5.4.1 下载 DM 安装包

DM 安装包位于 TiDB 离线工具包中。下载方式，请参考TiDB 工具下载。

13.6.5.4.2 使用样例

假设在五台服务器上部署两个 DM-worker 实例和三个 DM-master 实例。各个节点的信息如下：

实例	服务器地址	端口
DM-master1	192.168.0.4	8261
DM-master2	192.168.0.5	8261
DM-master3	192.168.0.6	8261
DM-worker1	192.168.0.7	8262
DM-worker2	192.168.0.8	8262

下面以此为例，说明如何部署 DM。

> 注意：
>
> - 在单机部署多个 DM-master 或 DM-worker 时，需要确保每个实例的端口以及运行命令的当前目录各不相同。
> - 如果不需要确保 DM 集群高可用，则可只部署 1 个 DM-master 节点，且部署的 DM-worker 节点数量不少于上游待迁移的 MySQL/MariaDB 实例数。
> - 如果需要确保 DM 集群高可用，则推荐部署 3 个 DM-master 节点，且部署的 DM-worker 节点数量大于上游待迁移的 MySQL/MariaDB 实例数（如 DM-worker 节点数量比上游实例数多 2 个）。
> - 需要确保以下组件间端口可正常连通：
>   - 各 DM-master 节点间的 8291 端口可互相连通。
>   - 各 DM-master 节点可连通所有 DM-worker 节点的 8262 端口。
>   - 各 DM-worker 节点可连通所有 DM-master 节点的 8261 端口。

部署 DM-master

DM-master 提供命令行参数和配置文件两种配置方式。

使用命令行参数部署 DM-master

DM-master 的命令行参数说明：

```
./dm-master --help
```

```
Usage of dm-master:
  -L string
        log level: debug, info, warn, error, fatal (default "info")
  -V    prints version and exit
  -advertise-addr string
        advertise address for client traffic (default "${master-addr}")
  -advertise-peer-urls string
        advertise URLs for peer traffic (default "${peer-urls}")
  -config string
        path to config file
  -data-dir string
        path to the data directory (default "default.${name}")
  -initial-cluster string
        initial cluster configuration for bootstrapping, e.g. dm-master=http://127.0.0.1:8291
  -join string
        join to an existing cluster (usage: cluster's "${master-addr}" list, e.g.
            ↪ "127.0.0.1:8261,127.0.0.1:18261"
  -log-file string
        log file path
  -master-addr string
        master API server and status addr
  -name string
        human-readable name for this DM-master member
  -peer-urls string
        URLs for peer traffic (default "http://127.0.0.1:8291")
  -print-sample-config
        print sample config file of dm-worker
```

> **注意：**
>
> 某些情况下，无法使用命令行参数来配置 DM-master，因为有的配置并未暴露给命令行。

使用配置文件部署 DM-master

推荐使用配置文件，把以下配置文件内容写入到 `conf/dm-master1.toml` 中。

DM-master 的配置文件：

```
#### Master Configuration.

name = "master1"

#### 日志配置
log-level = "info"
log-file = "dm-master.log"

#### DM-master 监听地址
master-addr = "192.168.0.4:8261"

#### DM-master 节点的对等 URL
peer-urls = "192.168.0.4:8291"

#### 初始集群中所有 DM-master 的 advertise-peer-urls 的值
initial-cluster = "master1=http://192.168.0.4:8291,master2=http://192.168.0.5:8291,master3=http
    ↪ ://192.168.0.6:8291"
```

在终端中使用下面的命令运行 DM-master：

> 注意：
>
> 执行该命令后控制台不会输出日志，可以通过 `tail -f dm-master.log` 查看运行日志。

```
./dm-master -config conf/dm-master1.toml
```

对于 DM-master2 和 DM-master3，修改配置文件中的 `name` 为 `master2` 和 `master3`，并将 `peer-urls` 的值改为 `192.168.0.5:8291` 和 `192.168.0.6:8291` 即可。

部署 DM-worker

DM-worker 提供命令行参数和配置文件两种配置方式。

使用命令行参数部署 DM-worker

查看 DM-worker 的命令行参数说明：

```
./dm-worker --help
```

```
Usage of worker:
  -L string
        log level: debug, info, warn, error, fatal (default "info")
  -V    prints version and exit
  -advertise-addr string
        advertise address for client traffic (default "${worker-addr}")
```

```
  -config string
        path to config file
  -join string
        join to an existing cluster (usage: dm-master cluster's "${master-addr}")
  -keepalive-ttl int
        dm-worker's TTL for keepalive with etcd (in seconds) (default 10)
  -log-file string
        log file path
  -name string
        human-readable name for DM-worker member
  -print-sample-config
        print sample config file of dm-worker
  -worker-addr string
        listen address for client traffic
```

> **注意：**
>
> 某些情况下，无法使用命令行参数的方法来配置 DM-worker，因为有的配置并未暴露给命令行。

使用配置文件部署 DM-worker

推荐使用配置文件来配置 DM-worker，把以下配置文件内容写入到 `conf/dm-worker1.toml` 中。

DM-worker 的配置文件：

```
#### Worker Configuration.

name = "worker1"

#### 日志配置
log-level = "info"
log-file = "dm-worker.log"

#### DM-worker 的地址
worker-addr = ":8262"

#### 对应集群中 DM-master 配置中的 master-addr
join = "192.168.0.4:8261,192.168.0.5:8261,192.168.0.6:8261"
```

在终端中使用下面的命令运行 DM-worker：

```
./dm-worker -config conf/dm-worker1.toml
```

对于 DM-worker2，修改配置文件中的 `name` 为 `worker2` 即可。

这样，DM 集群就部署成功了。

13.6.5.5 在 Kubernetes 环境中部署

13.6.6 入门指南

13.6.6.1 创建 TiDB Data Migration 数据源

> **注意：**
>
> 在创建数据源之前，你需要先使用 TiUP 部署 DM 集群。

本文档介绍如何为 TiDB Data Migration (DM) 的数据迁移任务创建数据源。

数据源包含了访问迁移任务上游所需的信息。数据迁移任务需要引用对应的数据源来获取访问配置信息。因此，在创建数据迁移任务之前，需要先创建任务的数据源。详细的数据源管理命令请参考管理上游数据源。

13.6.6.1.1 第一步：配置数据源

1. （可选）加密数据源密码

   在 DM 的配置文件中，推荐使用经 dmctl 加密后的密文密码。按照下面的示例可以获得数据源的密文密码，用于下一步编写数据源配置文件。

```
tiup dmctl encrypt 'abc!@#123'
```

```
MKxn0Qo3m3XOyjCnhEMtsUCm83EhGQDZ/T4=
```

2. 编写数据源配置文件

   每个数据源需要一个单独的配置文件来创建数据源。按照下面示例创建 ID 为 “mysql-01” 的数据源，创建数据源配置文件 `./source-mysql-01.yaml`：

```
source-id: "mysql-01"    # 数据源 ID，在数据迁移任务配置和 dmctl 命令行中引用该 source-id
    ↪ 可以关联到对应的数据源

from:
  host: "127.0.0.1"
  port: 3306
  user: "root"
  password: "MKxn0Qo3m3XOyjCnhEMtsUCm83EhGQDZ/T4=" # 推荐使用 dmctl
      ↪ 对上游数据源的用户密码加密之后的密码
  security:                                        # 上游数据源 TLS 相关配置。
      ↪ 如果没有需要则可以删除
    ssl-ca: "/path/to/ca.pem"
    ssl-cert: "/path/to/cert.pem"
    ssl-key: "/path/to/key.pem"
```

13.6.6.1.2 第二步：创建数据源

使用如下命令创建数据源：

```
tiup dmctl --master-addr <master-addr> operate-source create ./source-mysql-01.yaml
```

数据源配置文件的其他配置参考数据源配置文件介绍。

命令返回结果如下：

```
{
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "",
            "source": "mysql-01",
            "worker": "dm-worker-1"
        }
    ]
}
```

13.6.6.1.3 第三步：查询创建的数据源

创建数据源后，可以使用如下命令查看创建的数据源：

- 如果知道数据源的 source-id，可以通过 dmctl config source <source-id> 命令直接查看数据源配置：

```
tiup dmctl --master-addr <master-addr> config source mysql-01
```

```
{
  "result": true,
  "msg": "",
  "cfg": "enable-gtid: false
    flavor: mysql
    source-id: mysql-01
    from:
      host: 127.0.0.1
      port: 3306
      user: root
      password: '******'
}
```

- 如果不知道数据源的 source-id，可以先通过 dmctl operate-source show 命令查看源数据库列表，从中可以找到对应的数据源。

```
tiup dmctl --master-addr <master-addr> operate-source show
```

```
{
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "source is added but there is no free worker to bound",
            "source": "mysql-02",
            "worker": ""
        },
        {
            "result": true,
            "msg": "",
            "source": "mysql-01",
            "worker": "dm-worker-1"
        }
    ]
}
```

13.6.6.2 管理 TiDB Data Migration 上游数据源

dmctl 是运维 TiDB Data Migration (DM) 集群的命令行工具。本文介绍了如何使用 dmctl 组件来管理数据源配置，包括如何加密数据库密码，数据源操作，查看数据源配置，改变数据源与 DM-worker 的绑定关系。

13.6.6.2.1 加密数据库密码

在 DM 相关配置文件中，推荐使用经 dmctl 加密后的密码。对于同一个原始密码，每次加密后密码不同。

```
./dmctl -encrypt 'abc!@#123'
```

```
MKxnOQo3m3XOyjCnhEMtsUCm83EhGQDZ/T4=
```

13.6.6.2.2 数据源操作

operate-source 命令向 DM 集群加载、列出、移除数据源。

```
help operate-source
```

```
`create`/`stop`/`show` upstream MySQL/MariaDB source.

Usage:
  dmctl operate-source <operate-type> [config-file ...] [--print-sample-config] [flags]

Flags:
  -h, --help                  help for operate-source
```

```
  -p, --print-sample-config   print sample config file of source

Global Flags:
  -s, --source strings   MySQL Source ID
```

参数解释

- `create`：创建一个或多个上游的数据库源。创建多个数据源失败时，会尝试回滚到执行命令之前的状态
- `stop`：停止一个或多个上游的数据库源。停止多个数据源失败时，可能有部分数据源已成功停止
- `show`：显示已添加的数据源以及对应的 DM-worker
- `config-file`：
  - 指定 `source.yaml` 的文件路径
  - 可传递多个文件路径
- `--print-sample-config`：打印示例配置文件。该参数会忽视其余参数

命令用法示例

使用 `operate-source` 命令创建数据源配置：

```
operate-source create ./source.yaml
```

其中 `source.yaml` 的配置参考上游数据库配置文件介绍。

结果如下：

```
{
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-01",
            "worker": "dm-worker-1"
        }
    ]
}
```

13.6.6.2.3 查看数据源配置

> **注意：**
>
> `config` 命令仅在 DM v6.0 及其以后版本支持，之前版本可使用 `get-config` 命令。

如果知道 source-id，可以通过 `dmctl --master-addr <master-addr> config source <source-id>` 命令直接查看数据源配置。

```
config source mysql-replica-01
```

```
{
  "result": true,
  "msg": "",
  "cfg": "enable-gtid: false
    flavor: mysql
    source-id: mysql-replica-01
    from:
      host: 127.0.0.1
      port: 8407
      user: root
      password: '******'
}
```

如果不知道 source-id，可以先通过 `dmctl --master-addr <master-addr> operate-source show` 查看源数据库列表。

```
operate-source show
```

```
{
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "source is added but there is no free worker to bound",
            "source": "mysql-replica-02",
            "worker": ""
        },
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-01",
            "worker": "dm-worker-1"
        }
    ]
}
```

13.6.6.2.4 改变数据源与 DM-worker 的绑定关系

`transfer-source` 用于改变数据源与 DM-worker 的绑定关系。

```
help transfer-source
```

```
Transfers a upstream MySQL/MariaDB source to a free worker.

Usage:
  dmctl transfer-source <source-id> <worker-id> [flags]

Flags:
  -h, --help   help for transfer-source

Global Flags:
  -s, --source strings   MySQL Source ID.
```

在改变绑定关系前，DM 会检查待解绑的 worker 是否正在运行同步任务，如果正在运行则需要先暂停任务，并在改变绑定关系后恢复任务。

命令用法示例

如果不清楚 DM-worker 的绑定关系，可以通过 `dmctl --master-addr <master-addr> list-member --worker` 查看。

```
list-member --worker
```

```
{
    "result": true,
    "msg": "",
    "members": [
        {
            "worker": {
                "msg": "",
                "workers": [
                    {
                        "name": "dm-worker-1",
                        "addr": "127.0.0.1:8262",
                        "stage": "bound",
                        "source": "mysql-replica-01"
                    },
                    {
                        "name": "dm-worker-2",
                        "addr": "127.0.0.1:8263",
                        "stage": "free",
                        "source": ""
                    }
                ]
            }
        }
```

```
    ]
}
```

在本示例中 mysql-replica-01 绑定到了 dm-worker-1 上。使用如下命令可以将该数据源绑定到 dm-worker-2 上

```
transfer-source mysql-replica-01 dm-worker-2
```

```
{
    "result": true,
    "msg": ""
}
```

再次通过 dmctl --master-addr <master-addr> list-member --worker 查看，检查命令已生效。

```
list-member --worker
```

```
{
    "result": true,
    "msg": "",
    "members": [
        {
            "worker": {
                "msg": "",
                "workers": [
                    {
                        "name": "dm-worker-1",
                        "addr": "127.0.0.1:8262",
                        "stage": "free",
                        "source": ""
                    },
                    {
                        "name": "dm-worker-2",
                        "addr": "127.0.0.1:8263",
                        "stage": "bound",
                        "source": "mysql-replica-01"
                    }
                ]
            }
        }
    ]
}
```

13.6.6.3 TiDB Data Migration 数据迁移任务配置向导

本文档介绍如何配置 TiDB Data Migration (DM) 的数据迁移任务。

13.6.6.3.1 配置需要迁移的数据源

配置需要迁移的数据源之前，首先应该确认已经在 DM 创建相应数据源：

- 查看数据源可以参考查看数据源配置
- 创建数据源可以参考在 DM 创建数据源
- 数据源配置可以参考数据源配置文件介绍

仿照下面的 `mysql-instances`: 示例定义数据迁移任务需要同步的单个或者多个数据源。

```
---

##### ********* 任务信息配置 *********
name: test              # 任务名称，需要全局唯一

##### ******** 数据源配置 **********
mysql-instances:
  - source-id: "mysql-replica-01"  # 从 source-id = mysql-replica-01 的数据源迁移数据
  - source-id: "mysql-replica-02"  # 从 source-id = mysql-replica-02 的数据源迁移数据
```

13.6.6.3.2 配置迁移的目标 TiDB 集群

仿照下面的 `target-database`: 示例定义迁移的目标 TiDB 集群。

```
---

##### ********* 任务信息配置 *********
name: test              # 任务名称，需要全局唯一

##### ******** 数据源配置 **********
mysql-instances:
  - source-id: "mysql-replica-01"  # 从 source-id = mysql-replica-01 的数据源迁移数据
  - source-id: "mysql-replica-02"  # 从 source-id = mysql-replica-02 的数据源迁移数据

##### ******** 目标 TiDB 配置 **********
target-database:       # 目标 TiDB 配置
  host: "127.0.0.1"
  port: 4000
  user: "root"
  password: ""         # 如果密码不为空，则推荐使用经过 dmctl 加密的密文
```

13.6.6.3.3 配置需要迁移的表

如果不需要过滤或迁移特定表，可以跳过该项配置。

配置从数据源迁移表的黑白名单，则需要添加两个定义，详细配置规则参考Block & Allow Lists：

1. 定义全局的黑白名单规则

```
block-allow-list:
  bw-rule-1:                            # 规则名称
    do-dbs: ["test.*", "user"]          # 迁移哪些库，支持通配符 "*" 和 "?"，do-dbs 和 ignore
        ↪ -dbs 只需要配置一个，如果两者同时配置只有 do-dbs 会生效
    # ignore-dbs: ["mysql", "account"] # 忽略哪些库，支持通配符 "*" 和 "?"
    do-tables:                          # 迁移哪些表，do-tables 和 ignore-tables
        ↪ 只需要配置一个，如果两者同时配置只有 do-tables 会生效
    - db-name: "test.*"
      tbl-name: "t.*"
    - db-name: "user"
      tbl-name: "information"
  bw-rule-2:                           # 规则名称
    ignore-tables:                     # 忽略哪些表
    - db-name: "user"
      tbl-name: "log"
```

2. 在数据源配置中引用黑白名单规则，过滤该数据源需要迁移的表

```
mysql-instances:
    - source-id: "mysql-replica-01"  # 从 source-id = mysql-replica-01 的数据源迁移数据
      block-allow-list:  "bw-rule-1" # 黑白名单配置名称，如果 DM 版本早于 v2.0.0-beta.2
          ↪ 则使用 black-white-list
    - source-id: "mysql-replica-02"  # 从 source-id = mysql-replica-02 的数据源迁移数据
      block-allow-list:  "bw-rule-2" # 黑白名单配置名称，如果 DM 版本早于 v2.0.0-beta.2
          ↪ 则使用 black-white-list
```

13.6.6.3.4 配置需要过滤的操作

如果不需要过滤特定库或者特定表的特定操作，可以跳过该项配置。

配置过滤特定操作，则需要添加两个定义，详细配置规则参考Binlog Event Filter：

1. 定义全局的数据源操作过滤规则

```
filters:                                    # 定义过滤数据源特定操作的规则，
    ↪ 可以定义多个规则
  filter-rule-1:                            # 规则名称
    schema-pattern: "test_*"                # 匹配数据源的库名，支持通配符 "*" 和 "?"
    table-pattern: "t_*"                    # 匹配数据源的表名，支持通配符 "*" 和 "?"
    events: ["truncate table", "drop table"]    # 匹配上 schema-pattern 和 table-pattern
        ↪ 的库或者表的操作类型
    action: Ignore                          # 迁移 ( Do ) 还是忽略(Ignore)
  filter-rule-2:
    schema-pattern: "test"
    events: ["all dml"]
```

```
    action: Do
```

2. 在数据源配置中引用数据源操作过滤规则，过滤该数据源的指定库或表的指定操作

```
mysql-instances:
  - source-id: "mysql-replica-01"    # 从 source-id = mysql-replica-01 的数据源迁移数据
    block-allow-list:  "bw-rule-1"   # 黑白名单配置名称，如果 DM 版本早于 v2.0.0-beta.2
        ↪ 则使用 black-white-list
    filter-rules: ["filter-rule-1"]  # 过滤数据源特定操作的规则，可以配置多个过滤规则
  - source-id: "mysql-replica-02"    # 从 source-id = mysql-replica-02 的数据源迁移数据
    block-allow-list:  "bw-rule-2"   # 黑白名单配置名称，如果 DM 版本早于 v2.0.0-beta.2
        ↪ 则使用 black-white-list
    filter-rules: ["filter-rule-2"]  # 过滤数据源特定操作的规则，可以配置多个过滤规则
```

13.6.6.3.5 配置需要数据源表到目标 TiDB 表的映射

如果不需要将数据源表路由到不同名的目标 TiDB 表，可以跳过该项配置。分库分表合并迁移的场景必须配置该规则。

配置数据源表迁移到目标 TiDB 表的路由规则，则需要添加两个定义，详细配置规则参考Table Routing：

1. 定义全局的路由规则

```
routes:                          # 定义数据源表迁移到目标 TiDB 表的路由规则，
    ↪ 可以定义多个规则
  route-rule-1:                  # 规则名称
    schema-pattern: "test_*"     # 匹配数据源的库名，支持通配符 "*" 和 "?"
    table-pattern: "t_*"         # 匹配数据源的表名，支持通配符 "*" 和 "?"
    target-schema: "test"        # 目标 TiDB 库名
    target-table: "t"            # 目标 TiDB 表名
  route-rule-2:
    schema-pattern: "test_*"
    target-schema: "test"
```

2. 在数据源配置中引用路由规则，过滤该数据源需要迁移的表

```
mysql-instances:
  - source-id: "mysql-replica-01"                    # 从 source-id = mysql-replica-01
      ↪ 的数据源迁移数据
    block-allow-list:  "bw-rule-1"                   # 黑白名单配置名称，如果 DM 版本早于
        ↪ v2.0.0-beta.2 则使用 black-white-list
    filter-rules: ["filter-rule-1"]                  # 过滤数据源特定操作的规则，
        ↪ 可以配置多个过滤规则
    route-rules: ["route-rule-1", "route-rule-2"]    # 数据源表迁移到目标 TiDB 表的路由规则
        ↪ ，可以定义多个规则
  - source-id: "mysql-replica-02"                    # 从 source-id = mysql-replica-02
      ↪ 的数据源迁移数据
```

```
    block-allow-list:  "bw-rule-2"                     # 黑白名单配置名称，如果 DM 版本早于
        ↪ v2.0.0-beta.2 则使用 black-white-list
    filter-rules: ["filter-rule-2"]                    # 过滤数据源特定操作的规则，
        ↪ 可以配置多个过滤规则
```

13.6.6.3.6 配置是否进行分库分表合并

如果是分库分表合并的数据迁移场景，并且需要同步分库分表的 DDL，则必须显式配置 shard-mode，否则不要配置该选项。

分库分表 DDL 同步问题特别多，请确认了解 DM 同步分库分表 DDL 的原理和限制后，谨慎使用。

```
---

##### ********* 任务信息配置 *********
name: test                      # 任务名称，需要全局唯一
shard-mode: "pessimistic"       # 默认值为 "" 即无需协调。如果为分库分表合并任务，
    ↪ 请设置为悲观协调模式 "pessimistic"。在深入了解乐观协调模式的原理和使用限制后，
    ↪ 也可以设置为乐观协调模式 "optimistic"
```

13.6.6.3.7 其他配置

下面是本数据迁移任务配置向导的完整示例。完整的任务配置参见DM 任务完整配置文件介绍。

```
---

##### ********* 任务信息配置 *********
name: test                      # 任务名称，需要全局唯一
shard-mode: "pessimistic"       # 默认值为 "" 即无需协调。如果为分库分表合并任务，
    ↪ 请设置为悲观协调模式 "pessimistic"。在深入了解乐观协调模式的原理和使用限制后，
    ↪ 也可以设置为乐观协调模式 "optimistic"
task-mode: all                  # 任务模式，可设为 "full" - "只进行全量数据迁移"、"incremental" -
    ↪  "Binlog 实时同步"、"all" - "全量 + Binlog 迁移"
#### timezone: "UTC"                # 指定数据迁移任务时 SQL Session 使用的时区。DM
    ↪ 默认使用目标库的全局时区配置进行数据迁移，并且自动确保同步数据的正确性。
    ↪ 使用自定义时区依然可以确保整个流程的正确性，但一般不需要手动指定。

##### ******** 数据源配置 **********
mysql-instances:
  - source-id: "mysql-replica-01"                  # 从 source-id = mysql-replica-01
      ↪ 的数据源迁移数据
    block-allow-list:  "bw-rule-1"                 # 黑白名单配置名称，如果 DM 版本早于 v2.0.0-
        ↪ beta.2 则使用 black-white-list
    filter-rules: ["filter-rule-1"]                # 过滤数据源特定操作的规则，
        ↪ 可以配置多个过滤规则
```

```
    route-rules: ["route-rule-1", "route-rule-2"]   # 数据源表迁移到目标 TiDB 表的路由规则，
        ↪ 可以定义多个规则
  - source-id: "mysql-replica-02"                   # 从 source-id = mysql-replica-02
      ↪ 的数据源迁移数据
    block-allow-list:  "bw-rule-2"                  # 黑白名单配置名称，如果 DM 版本早于 v2.0.0-
        ↪ beta.2 则使用 black-white-list
    filter-rules: ["filter-rule-2"]                 # 过滤数据源特定操作的规则，
        ↪ 可以配置多个过滤规则
    route-rules: ["route-rule-2"]                   # 数据源表迁移到目标 TiDB 表的路由规则，
        ↪ 可以定义多个规则

##### ******** 目标 TiDB 配置 **********
target-database:       # 目标 TiDB 配置
  host: "127.0.0.1"
  port: 4000
  user: "root"
  password: ""         # 如果密码不为空，则推荐使用经过 dmctl 加密的密文

##### ******** 功能配置 **********
block-allow-list:                       # 定义数据源迁移表的过滤规则，可以定义多个规则。如果 DM
    ↪ 版本早于 v2.0.0-beta.2 则使用 black-white-list
  bw-rule-1:                            # 规则名称
    do-dbs: ["test.*", "user"]          # 迁移哪些库，支持通配符 "*" 和 "?"，do-dbs 和 ignore-dbs
        ↪ 只需要配置一个，如果两者同时配置只有 do-dbs 会生效
    # ignore-dbs: ["mysql", "account"] # 忽略哪些库，支持通配符 "*" 和 "?"
    do-tables:                          # 迁移哪些表，do-tables 和 ignore-tables 只需要配置一个，
        ↪ 如果两者同时配置只有 do-tables 会生效
    - db-name: "test.*"
      tbl-name: "t.*"
    - db-name: "user"
      tbl-name: "information"
  bw-rule-2:                           # 规则名称
    ignore-tables:                     # 忽略哪些表
    - db-name: "user"
      tbl-name: "log"

filters:                                        # 定义过滤数据源特定操作的规则，可以定义多个规则
  filter-rule-1:                                # 规则名称
    schema-pattern: "test_*"                    # 匹配数据源的库名，支持通配符 "*" 和 "?"
    table-pattern: "t_*"                        # 匹配数据源的表名，支持通配符 "*" 和 "?"
    events: ["truncate table", "drop table"]    # 匹配上 schema-pattern 和 table-pattern
        ↪ 的库或者表的操作类型
    action: Ignore                              # 迁移（Do）还是忽略(Ignore)
  filter-rule-2:
    schema-pattern: "test"
```

```
    events: ["all dml"]
    action: Do

routes:                             # 定义数据源表迁移到目标 TiDB 表的路由规则，可以定义多个规则
  route-rule-1:                     # 规则名称
    schema-pattern: "test_*"        # 匹配数据源的库名，支持通配符 "*" 和 "?"
    table-pattern: "t_*"            # 匹配数据源的表名，支持通配符 "*" 和 "?"
    target-schema: "test"           # 目标 TiDB 库名
    target-table: "t"               # 目标 TiDB 表名
  route-rule-2:
    schema-pattern: "test_*"
    target-schema: "test"
```

13.6.6.4 TiDB Data Migration 分库分表合并

TiDB Data Migration (DM) 支持将上游 MySQL/MariaDB 各分库分表中的 DML、DDL 数据合并后迁移到下游 TiDB 的库表中。

如果你需要从小数据量分库分表 MySQL 合并迁移数据到 TiDB，请参考这篇文档

13.6.6.4.1 使用限制

目前分库分表合并功能仅支持有限的场景，使用该功能前，请仔细阅读悲观模式分库分表合并迁移使用限制和乐观模式分库分表合并迁移使用限制。

13.6.6.4.2 参数配置

在任务配置文件中设置：

```
shard-mode: "pessimistic" # 默认值为 "" 即无需协调。如果为分库分表合并任务，请设置为悲观协调模式
    ↪ "pessimistic"。在深入了解乐观协调模式的原理和使用限制后，也可以设置为乐观协调模式 "
    ↪ optimistic"
```

13.6.6.4.3 手动处理 Sharding DDL Lock

如果分库分表合并迁移过程中发生了异常，对于部分场景，可尝试参考手动处理 Sharding DDL Lock 进行处理。

13.6.6.5 TiDB Data Migration 表路由

使用 TiDB Data Migration (DM) 迁移数据时，你可以配置表路由 (Table Routing) 规则，指定将上游 MySQL/MariaDB 实例的特定表迁移到下游的特定表。

> **注意：**

- 不支持对同一个表设置多个不同的路由规则。
- Schema 的匹配规则需要单独设置，用来迁移 `CREATE/DROP SCHEMA ...`，例如下面配置表路由中的 rule-2。

13.6.6.5.1 配置表路由

在迁移任务配置文件中，添加如下配置：

```
routes:
  rule-1:
    schema-pattern: "test_*"
    table-pattern: "t_*"
    target-schema: "test"
    target-table: "t"
    # extract-table、extract-schema 和 extract-source 为可选配置，仅在需要提取分表、
        ↪ 分库和数据源信息时填写
    extract-table:
      table-regexp: "t_(.*)"
      target-column: "c_table"
    extract-schema:
      schema-regexp: "test_(.*)"
      target-column: "c_schema"
    extract-source:
      source-regexp: "(.*)"
      target-column: "c_source"
  rule-2:
    schema-pattern: "test_*"
    target-schema: "test"
```

在简单任务场景下，推荐使用通配符匹配库表名，但需注意以下版本差异：

- 对于 v1.0.5 版及后续版本，表路由支持通配符匹配。但注意所有版本中通配符匹配中的 * 符号 只能有一个，且必须在末尾。
- 对于 v1.0.5 以前的版本，表路由支持通配符，但不支持 `[...]` 与 `[!...]` 表达式。

13.6.6.5.2 参数解释

- 对于匹配上schema-pattern/table-pattern 规则的上游 MySQL/MariaDB 实例的表，DM 将它们迁移到下游的 `target-schema/target-table`。
- 对于匹配上 `schema-pattern/table-pattern` 规则的分表，DM 通过 `extract-table.table-regexp` 提取分表信息，通过 `extract-schema.schema-regexp` 提取分库信息，通过 `extract-source.source-regexp` 提取数据来源信息，然后写入到下游合表对应的 `target-column` 中。

13.6.6.5.3 使用示例

下面展示了四个不同场景下的配置示例。如果你需要从小数据量分库分表 MySQL 合并迁移数据到 TiDB，请参考这篇文档

分库分表合并

假设存在分库分表场景，需要将上游两个 MySQL 实例的表 test_{1,2,3...}.t_{1,2,3...} 迁移到下游 TiDB 的一张表 test.t。

为了迁移到下游实例的表 test.t，需要创建以下表路由规则：

- rule-1 用来迁移匹配上 schema-pattern: "test_*" 和 table-pattern: "t_*" 的表的 DML/DDL 语句到下游的 test.t。
- rule-2 用来迁移匹配上 schema-pattern: "test_*" 的库的 DDL 语句，例如 CREATE/DROP SCHEMA xx。

> 注意：
>
> - 如果下游 TiDB schema: test 已经存在，并且不会被删除，则可以省略 rule-2。
> - 如果下游 TiDB schema: test 不存在，只设置了 rule_1，则迁移会报错 schema test doesn't exist。

```
rule-1:
  schema-pattern: "test_*"
  table-pattern: "t_*"
  target-schema: "test"
  target-table: "t"
rule-2:
  schema-pattern: "test_*"
  target-schema: "test"
```

提取分库分表数据源信息写入合表

假设存在分库分表场景，需要将上游两个 MySQL 实例的表 test_{11,12,13...}.t_{1,2,3...} 迁移到下游 TiDB 的一张表 test.t，同时需要提取分库分表的源信息写入下游合表中，用于标识合表中各行数据的来源。

为了迁移到下游实例的表 test.t，需要创建和分库分表合并场景类似的表路由规则，并在其中增加 extract-table、extract-schema、extract-source 配置用于提取分库分表源数据信息：

- extract-table：对于匹配上 schema-pattern 和 table-pattern 的分表，DM 根据 table-regexp 提取分表，并将去除 t_ 后的后缀信息写入合表的 target-column，即 c_table 列中。
- extract-schema：对于匹配上 schema-pattern和 table-pattern 的分库，DM 根据 schema-regexp 提取分库，并将去除 test_ 后的后缀信息写入合表的 target-column，即 c_schema 列中。
- extract-source：对于匹配上 schema-pattern 和 table-pattern 的分表，DM 将其数据源信息写入合表的 target-column，即 c_source 列中。

```
  rule-1:
    schema-pattern: "test_*"
    table-pattern: "t_*"
    target-schema: "test"
    target-table: "t"
    extract-table:
      table-regexp: "t_(.*)"
      target-column: "c_table"
    extract-schema:
      schema-regexp: "test_(.*)"
      target-column: "c_schema"
    extract-source:
      source-regexp: "(.*)"
      target-column: "c_source"
  rule-2:
    schema-pattern: "test_*"
    target-schema: "test"
```

为了提取上游分表来源信息数据以写入到下游合表，必须在启动迁移任务前手动在下游创建好对应合表，合表需要包含用于存放分表源数据信息的三个扩展列 `target-column`（表名列、库名列、数据源列），扩展列必须为表末尾列且必须为字符串类型。

```
CREATE TABLE `test`.`t` (
    a int(11) PRIMARY KEY,
    c_table varchar(10) DEFAULT NULL,
    c_schema varchar(10) DEFAULT NULL,
    c_source varchar(10) DEFAULT NULL
);
```

例如，上游源数据为：

数据源 `mysql-01`:

```
mysql> select * from test_11.t_1;
+---+
| a |
+---+
| 1 |
+---+
mysql> select * from test_11.t_2;
+---+
| a |
+---+
| 2 |
+---+
mysql> select * from test_12.t_1;
```

```
+---+
| a |
+---+
| 3 |
+---+
```

数据源 `mysql-02`:

```
mysql> select * from test_13.t_3;
+---+
| a |
+---+
| 4 |
+---+
```

则 DM 同步后合表中的数据将为：

```
mysql> select * from test.t;
+---+---------+----------+----------+
| a | c_table | c_schema | c_source |
+---+---------+----------+----------+
| 1 | 1       | 11       | mysql-01 |
| 2 | 2       | 11       | mysql-01 |
| 3 | 1       | 12       | mysql-01 |
| 4 | 3       | 13       | mysql-02 |
+---+---------+----------+----------+
```

错误的合表建表示例：

> 注意：
> 以下错误都可能导致分库分表数据源信息写入合表失败。

- `c-table` 列不在末尾

```
CREATE TABLE `test`.`t` (
    c_table varchar(10) DEFAULT NULL,
    a int(11) PRIMARY KEY,
    c_schema varchar(10) DEFAULT NULL,
    c_source varchar(10) DEFAULT NULL
);
```

- `c-source` 列缺失

```
CREATE TABLE `test`.`t` (
    a int(11) PRIMARY KEY,
    c_table varchar(10) DEFAULT NULL,
    c_schema varchar(10) DEFAULT NULL,
);
```

- c_schema 列为非 string 类型

```
CREATE TABLE `test`.`t` (
    a int(11) PRIMARY KEY,
    c_table varchar(10) DEFAULT NULL,
    c_schema int(11) DEFAULT NULL,
    c_source varchar(10) DEFAULT NULL,
);
```

分库合并

假设存在分库场景，将上游两个 MySQL 实例 `test_{1,2,3...}.t_{1,2,3...}` 迁移到下游 TiDB 的 `test.t_{1,2,3...}`，创建一条路由规则即可：

```
  rule-1:
    schema-pattern: "test_*"
    target-schema: "test"
```

错误的表路由

假设存在下面两个路由规则，`test_1_bak.t_1_bak` 可以匹配上 `rule-1` 和 `rule-2`，违反 table 路由的限制而报错。

```
  rule-0:
    schema-pattern: "test_*"
    target-schema: "test"
  rule-1:
    schema-pattern: "test_*"
    table-pattern: "t_*"
    target-schema: "test"
    target-table: "t"
  rule-2:
    schema-pattern: "test_1_bak"
    table-pattern: "t_1_bak"
    target-schema: "test"
    target-table: "t_bak"
```

13.6.6.6 TiDB Data Migration 黑白名单过滤

使用 TiDB Data Migration (DM) 迁移数据时，你可以配置上游数据库实例表的黑白名单过滤 (Block & Allow List) 规则，用来过滤或者只迁移某些 `database/table` 的所有操作。

13.6.6.6.1 配置黑白名单

在迁移任务配置文件中，添加如下配置：

```
block-allow-list:             # 如果 DM 版本早于 v2.0.0-beta.2 则使用 black-white-list。
  rule-1:
    do-dbs: ["test*"]         # 非 ~ 字符开头，表示规则是通配符；v1.0.5 及后续版本支持通配符规则
        ↪ 。
    do-tables:
    - db-name: "test[123]"    # 匹配 test1、test2、test3。
      tbl-name: "t[1-5]"      # 匹配 t1、t2、t3、t4、t5。
    - db-name: "test"
      tbl-name: "t"
  rule-2:
    do-dbs: ["~^test.*"]      # 以 ~ 字符开头，表示规则是正则表达式。
    ignore-dbs: ["mysql"]
    do-tables:
    - db-name: "~^test.*"
      tbl-name: "~^t.*"
    - db-name: "test"
      tbl-name: "t"
    ignore-tables:
    - db-name: "test"
      tbl-name: "log"
```

在简单任务场景下，推荐使用通配符匹配库表名，但需注意以下版本差异：

- 对于 v1.0.5 版及后续版本，黑白名单支持通配符匹配。但注意所有版本中通配符匹配中的 * 符号 只能有一个，且必须在末尾。
- 对于 v1.0.5 以前的版本，黑白名单仅支持正则表达式。

13.6.6.6.2 参数解释

- do-dbs：要迁移的库的白名单，类似于 MySQL 中的 replicate-do-db。
- ignore-dbs：要迁移的库的黑名单，类似于 MySQL 中的 replicate-ignore-db。
- do-tables：要迁移的表的白名单，类似于 MySQL 中的 replicate-do-table。必须同时指定 db-name 与 tbl-name。
- ignore-tables：要迁移的表的黑名单，类似于 MySQL 中的 replicate-ignore-table。必须同时指定 db-name 与 tbl-name。

以上参数值以 ~ 开头时均支持使用正则表达式来匹配库名、表名。

13.6.6.6.3 过滤规则

- do-dbs 与 ignore-dbs 对应的过滤规则与 MySQL 中的 Evaluation of Database-Level Replication and Binary Logging Options 类似。

- do-tables 与 ignore-tables 对应的过滤规则与 MySQL 中的 Evaluation of Table-Level Replication Options 类似。

> 注意：
>
> DM 中黑白名单过滤规则与 MySQL 中相应规则存在以下区别：
>
> - MySQL 中存在 replicate-wild-do-table 与 replicate-wild-ignore-table 用于支持通配符，DM 中各配置参数直接支持以 ~ 字符开头的正则表达式。
> - DM 当前只支持 ROW 格式的 binlog，不支持 STATEMENT/MIXED 格式的 binlog，因此应与 MySQL 中 ROW 格式下的规则对应。
> - 对于 DDL，MySQL 仅依据默认的 database 名称（USE 语句显式指定的 database）进行判断，而 DM 优先依据 DDL 中的 database 名称部分进行判断，并当 DDL 中不包含 database 名称时再依据 USE 部分进行判断。假设需要判断的 SQL 为 `USE test_db_2; CREATE TABLE test_db_1.test_table (c1 INT PRIMARY KEY)`，且 MySQL 配置了 replicate-do-db=test_db_1、DM 配置了 do-dbs: ["test_db_1"]，则对于 MySQL 该规则不会生效，而对于 DM 该规则会生效。

判断 table test.t 是否应该被过滤的流程如下：

1. 首先进行 schema 过滤判断
    - 如果 do-dbs 不为空，判断 do-dbs 中是否存在一个匹配的 schema。
        - 如果存在，则进入 table 过滤判断。
        - 如果不存在，则过滤 test.t。
    - 如果 do-dbs 为空并且 ignore-dbs 不为空，判断 ignore-dbs 中是否存在一个匹配的 schema。
        - 如果存在，则过滤 test.t。
        - 如果不存在，则进入 table 过滤判断。
    - 如果 do-dbs 和 ignore-dbs 都为空，则进入 table 过滤判断。
2. 进行 table 过滤判断
    1. 如果 do-tables 不为空，判断 do-tables 中是否存在一个匹配的 table。
        - 如果存在，则迁移 test.t。
        - 如果不存在，则过滤 test.t。
    2. 如果 ignore-tables 不为空，判断 ignore-tables 中是否存在一个匹配的 table。
        - 如果存在，则过滤 test.t.
        - 如果不存在，则迁移 test.t。
    3. 如果 do-tables 和 ignore-tables 都为空，则迁移 test.t。

> 注意：
>
> 如果是判断 schema test 是否应该被过滤，则只进行 schema 过滤判断。

13.6.6.6.4 使用示例

假设上游 MySQL 实例包含以下表：

```
`logs`.`messages_2016`
`logs`.`messages_2017`
`logs`.`messages_2018`
`forum`.`users`
`forum`.`messages`
`forum_backup_2016`.`messages`
`forum_backup_2017`.`messages`
`forum_backup_2018`.`messages`
```

配置如下：

```
block-allow-list:  # 如果 DM 版本早于 v2.0.0-beta.2 则使用 black-white-list。
  bw-rule:
    do-dbs: ["forum_backup_2018", "forum"]
    ignore-dbs: ["~^forum_backup_"]
    do-tables:
    - db-name: "logs"
      tbl-name: "~_2018$"
    - db-name: "~^forum.*"
       tbl-name: "messages"
    ignore-tables:
    - db-name: "~.*"
       tbl-name: "^messages.*"
```

应用 bw-rule 规则后：

table	是否过滤	过滤的原因
logs ↪ .messages_2016 ↪	是	schema logs 没有匹配到 do-dbs 任意一项
logs ↪ .messages_2017 ↪	是	schema logs 没有匹配到 do-dbs 任意一项
logs ↪ .messages_2018 ↪	是	schema logs 没有匹配到 do-dbs 任意一项
forum_backup_2016 ↪ .messages ↪	是	schema forum_backup_2016 没有匹配到 do-dbs 任意一项

table	是否过滤	过滤的原因
forum_backup_2017 ↪ .messages ↪	是	schema forum_backup_2017 没有匹配到 do-dbs 任意一项
forum ↪ .users ↪	是	1. schema forum 匹配到 do-dbs，进入 table 过滤判断 2. schema 和 table 没有匹配到 do-tables 和 ignore-tables 中任意一项，并且 do-tables 不为空，因此过滤
forum ↪ .messages ↪	否	1. schema forum 匹配到 do-dbs，进入 table 过滤判断 2. schema 和 table 匹配到 do-tables 的 db-name: "~^forum ↪ .*",tbl-name: ↪ "messages"
forum_backup_2018 ↪ .messages ↪	否	1. schema forum_backup_2018 匹配到 do-dbs，进入 table 过滤判断 2. schema 和 table 匹配到 do-tables 的 db-name: "~^forum ↪ .*",tbl-name: ↪ "messages"

13.6.6.7 TiDB Data Migration Binlog 事件过滤

TiDB Data Migration (DM) 的 Binlog 事件过滤 (Binlog event filter) 是比迁移表黑白名单更加细粒度的过滤规则，可以指定只迁移或者过滤掉某些 schema / table 的指定类型 binlog，比如 INSERT、TRUNCATE TABLE。

13.6.6.8 配置 Binlog 事件过滤

在迁移任务配置文件中，添加如下配置：

```
filters:
  rule-1:
    schema-pattern: "test_*"
```

```
  table-pattern: "t_*"
  events: ["truncate table", "drop table"]
 sql-pattern: ["^DROP\\s+PROCEDURE", "^CREATE\\s+PROCEDURE"]
  action: Ignore
```

从 DM v2.0.2 开始，你也可以在上游数据库配置文件中配置 Binlog 事件过滤。见上游数据库配置文件介绍。

在简单任务场景下，推荐使用通配符匹配库表名，但需注意以下版本差异：

- 对于 v1.0.5 版及后续版本，Binlog 事件过滤支持通配符匹配。但注意所有版本中通配符匹配中的 * 符号只能有一个，且必须在末尾。
- 对于 v1.0.5 以前的版本，Binlog Event Filter 支持通配符，但不支持 `[...]` 与 `[!...]` 表达式。

13.6.6.9 参数解释

- `schema-pattern/table-pattern`：对匹配上的上游 MySQL/MariaDB 实例的表的 binlog events 或者 DDL SQL 语句通过以下规则进行过滤。
- `events`：binlog events 数组，仅支持从以下 `Event` 中选择一项或多项。

Event	分类	解释
all		代表包含下面所有的 events
all dml		代表包含下面所有 DML events
all ddl		代表包含下面所有 DDL events
none		代表不包含下面所有 events
none ddl		代表不包含下面所有 DDL events
none dml		代表不包含下面所有 DML events
insert	DML	insert DML event
update	DML	update DML event
delete	DML	delete DML event
create database	DDL	create database event
drop database	DDL	drop database event
create table	DDL	create table event
create index	DDL	create index event
drop table	DDL	drop table event
truncate table	DDL	truncate table event
rename table	DDL	rename table event
drop index	DDL	drop index event
alter table	DDL	alter table event

- `sql-pattern`：用于过滤指定的 DDL SQL 语句，支持正则表达式匹配，例如上面示例中的 `"^DROP\\s+PROCEDURE"`。
- `action`：string (`Do` / `Ignore`)；进行下面规则判断，满足其中之一则过滤，否则不过滤。
  - `Do`：白名单。binlog event 如果满足下面两个条件之一就会被过滤掉：

    * 不在该 rule 的 events 中。
    * 如果规则的 sql-pattern 不为空的话，对应的 SQL 没有匹配上 sql-pattern 中任意一项。
  - Ignore：黑名单。如果满足下面两个条件之一就会被过滤掉：
    * 在该 rule 的 events 中。
    * 如果规则的 sql-pattern 不为空的话，对应的 SQL 可以匹配上 sql-pattern 中任意一项。
  - 同一个表匹配上多个规则时，将会顺序应用这些规则，并且黑名单的优先级高于白名单，即如果同时存在规则 Ignore 和 Do 应用在某个表上，那么 Ignore 生效。

13.6.6.10 使用示例

13.6.6.10.1 过滤分库分表的所有删除操作

需要设置下面两个 Binlog event filter rule 来过滤掉所有的删除操作：

- filter-table-rule 过滤掉所有匹配到 pattern test_*.t_* 的 table 的 turncate table、drop table、delete statement 操作。
- filter-schema-rule 过滤掉所有匹配到 pattern test_* 的 schema 的 drop database 操作。

```
filters:
  filter-table-rule:
    schema-pattern: "test_*"
    table-pattern: "t_*"
    events: ["truncate table", "drop table", "delete"]
    action: Ignore
  filter-schema-rule:
    schema-pattern: "test_*"
    events: ["drop database"]
    action: Ignore
```

13.6.6.10.2 只迁移分库分表的 DML 操作

需要设置下面两个 Binlog event filter rule 只迁移 DML 操作：

- do-table-rule 只迁移所有匹配到 pattern test_*.t_* 的 table 的 create table、insert、update、delete 操作。
- do-schema-rule 只迁移所有匹配到 pattern test_* 的 schema 的 create database 操作。

> **注意：**
>
> 迁移 create database/table 的原因是创建库和表后才能迁移 DML。

```
filters:
  do-table-rule:
    schema-pattern: "test_*"
    table-pattern: "t_*"
    events: ["create table", "all dml"]
    action: Do
  do-schema-rule:
    schema-pattern: "test_*"
    events: ["create database"]
    action: Do
```

13.6.6.10.3 过滤 TiDB 不支持的 SQL 语句

可设置如下规则过滤 TiDB 不支持的 PROCEDURE 语句：

```
filters:
  filter-procedure-rule:
    schema-pattern: "test_*"
    table-pattern: "t_*"
    sql-pattern: ["^DROP\\s+PROCEDURE", "^CREATE\\s+PROCEDURE"]
    action: Ignore
```

13.6.6.10.4 过滤 TiDB parser 不支持的 SQL 语句

对于 TiDB parser 不支持的 SQL 语句，DM 无法解析获得 schema/table 信息，因此需要使用全局过滤规则：`schema-pattern: "*"`。

> **注意：**
>
> 全局过滤规则的设置必须尽可能严格，以避免过滤掉需要迁移的数据。

可设置如下规则过滤某些版本的 TiDB parser 不支持的 PARTITION 语句：

```
filters:
  filter-partition-rule:
    schema-pattern: "*"
    sql-pattern: ["ALTER\\s+TABLE[\\s\\S]*ADD\\s+PARTITION", "ALTER\\s+TABLE[\\s\\S]*DROP\\s+
        ↪ PARTITION"]
    action: Ignore
```

13.6.6.11 通过 SQL 表达式过滤 DML

在进行增量数据迁移时，可以通过如何过滤 binlog 事件功能过滤某些类型的 binlog event，例如不向下游迁移 DELETE 事件以达到归档、审计等目的。但是 binlog event filter 无法以更细粒度判断某一行的 DELETE 事件是否要被过滤。

为了解决上述问题，从 v2.0.5 起，DM 支持在增量数据同步阶段使用`binlog value filter`过滤迁移数据。DM 支持的 ROW 格式的 binlog 中，binlog event 带有所有列的值。你可以基于这些值配置 SQL 表达式。如果该表达式对于某条行变更的计算结果是 TRUE，DM 就不会向下游迁移该条行变更。

具体操作步骤和实现细节，请参考如何通过 SQL 表达式过滤 DML。

13.6.6.12 TiDB Data Migration 对 online DDL 工具的支持

在 MySQL 生态中，gh-ost 与 pt-osc 等工具较广泛地被使用，TiDB Data Migration (DM) 对其提供了特殊的支持以避免对不必要的中间数据进行迁移。本文介绍了在 DM 中使用常见 online DDL 工具的方法和注意事项。

有关 DM 对 online DDL 工具支持的原理、处理流程等，可参考online-ddl。

13.6.6.12.1 使用限制

- DM 仅针对 gh-ost 与 pt-osc 做了特殊支持。
- 在开启 `online-ddl` 时，增量复制对应的 checkpoint 应不处于 online DDL 执行过程中。如上游某次 online DDL 操作开始于 binlog `position-A`、结束于 `position-B`，则增量复制的起始点应早于 `position-A` 或晚于 `position-B`，否则可能出现迁移出错，具体可参考FAQ。

13.6.6.12.2 参数配置

在 v2.0.5 及之后的版本，请在 `task` 配置文件中使用 `online-ddl` 配置项。

如上游 MySQL/MariaDB （同时）使用 gh-ost 或 pt-osc 工具，则在 task 的配置文件中设置：

```
online-ddl: true
```

> **注意：**
>
> 自 v2.0.5 起，`online-ddl-scheme` 已被弃用，请使用 `online-ddl` 代替 `online-ddl-scheme`。如设置 `online-ddl: true` 会覆盖掉 `online-ddl-scheme`。如设置 `online-ddl-scheme: "pt"` 或 `online-ddl-scheme: "gh-ost"` 会被转换为 `online-ddl: true`。

在 v2.0.5 之前的版本（不含 v2.0.5），请在 `task` 配置文件中使用 `online-ddl-scheme` 配置项。

如上游 MySQL/MariaDB 使用的是 gh-ost 工具，则在 task 的配置文件中设置：

```
online-ddl-scheme: "gh-ost"
```

如上游 MySQL/MariaDB 使用的是 pt-osc 工具，则在 task 的配置文件中设置：

```
online-ddl-scheme: "pt"
```

13.6.6.13 迁移任务操作

13.6.6.13.1 TiDB Data Migration 任务前置检查

本文介绍了 TiDB Data Migration (DM) 的任务前置检查功能。此功能用于提前检测出上游 MySQL 实例配置中可能存在的一些错误。

使用场景

为了使数据迁移任务顺利进行，DM 在启动迁移任务时会自动触发任务前置检查，并返回检查结果。只有当前置检查通过后，DM 才开始执行迁移任务。

如果你想要手动触发前置检查，运行 `check-task` 命令即可。

例如：

```
tiup dmctl check-task ./task.yaml
```

检查项说明

当任务前置检查被触发时，DM 会按照配置的迁移模式对相应的检查项进行检查。

本小节列出了前置检查中的所有检查项。

> 注意：
>
> 针对前置检查中必须通过的检查项，本文在检查项名称前标注了（必须）。
>
> - 对于标注了（必须）的检查项，如果检查没有通过，DM 会在检查结束后返回错误并拒绝执行该迁移任务。此时，请依据错误信息修改配置，在满足检查项要求后重试。
> - 对于未标注（必须）的检查项，如果检查没有通过，DM 会在检查结束后返回警告。如果检查结果中只有警告没有错误，DM 将自动开始执行该迁移任务。

通用检查项

对于任何一种迁移模式，前置检查都会包含以下通用检查项：

- 数据库版本
  - MySQL 版本 > 5.5
  - MariaDB 版本 >= 10.1.2

> 警告：
>
> - 使用 DM 从 MySQL v8.0 迁移数据到 TiDB 目前为实验特性（从 DM v2.0 引入），不建议在生产环境下使用。
> - 使用 DM 从 MariaDB 迁移数据到 TiDB 目前为实验特性，不建议在生产环境下使用。

- 上游 MySQL 表结构的兼容性
  - 检查上游表是否设置了外键。TiDB 不支持外键，如果上游表设置了外键，则返回警告。
  - 检查上游字符集是否与 TiDB 兼容，详见TiDB 支持的字符集。
  - 检查上游表中是否存在主键或唯一键约束（从 v1.0.7 版本引入）。

> 警告：
>
> - 上游使用不兼容的字符集时，下游可以使用 utf8mb4 字符集建表兼容同步，但不建议这样做。建议调整上游的字符集，使用下游支持的字符集。
> - 上游表不存在主键或唯一键约束时，可能出现单行数据在下游被重复同步多次的情况，同步性能也会降低，不建议在生产环境下使用。

全量数据迁移检查项

对于全量数据迁移模式（`task-mode: full`），除了通用检查项，前置检查还会包含以下检查项：

- （必须）上游数据库的 dump 权限
  - 检查是否有 INFORMATION_SCHEMA 和 dump 表的 SELECT 权限。
  - 如果 consistency=flush，将检查是否有 RELOAD 权限。
  - 如果 consistency=flush/lock，将检查是否有 dump 表的 LOCK TABLES 权限。
- （必须）上游 MySQL 多实例分库分表的一致性
  - 悲观协调模式下，检查所有分表的表结构是否一致，检查内容包括：
    * Column 数量
    * Column 名称
    * Column 顺序
    * Column 类型
    * 主键
    * 唯一索引
  - 乐观协调模式下，检查所有分表结构是否满足乐观协调兼容。
  - 如果曾经通过 `start-task` 命令成功启动任务，那么将不会对一致性进行检查。
- 分表中自增主键检查
  - 分表存在自增主键时返回警告。如果存在自增主键冲突，请参照自增主键冲突处理解决。

Physical Import 检查项

在任务配置中使用 `import-mode: "physical"` 后，会增加如下的前置检查项以保证Physical Import 正常运行。如果参照提示后仍然难以完成这些前置检查，你可以尝试使用Logical Import 进行导入。

- 下游数据库中的空 Region

  - 如果空 Region 的数量大于 `max(1000, 表的数量 * 3)`，即大于 “1000” 和 “3 倍表数量” 二者中的较大者，前置检查会向用户返回警告。可以调整 PD 相关参数加快空 Region 的合并速度，并等待空 Region 数量下降以解除警告。参见PD 调度策略最佳实践 - Region Merge 速度慢

- 下游数据库中的 Region 分布

  - 统计不同的 TiKV 上的 Region 数目。假设 Region 数最少的 TiKV 节点上拥有 a 个 Region，Region 数最多的 TiKV 节点上拥有 b 个 Region，如果 a/b 小于 0.75，则前置检查会向用户返回警告。可以调整 PD 相关参数加快 Region 调度速度，并等待 Region 数目变化以解除警告。参见PD 调度策略最佳实践 - Leader/Region 分布不均衡

- 下游数据库 TiDB、 PD、 TiKV 组件的版本

  - Physical Import 需要调用 TiDB、 PD、 TiKV 接口，如果版本不符合要求，会返回错误。

- 下游数据库的剩余空间

  - 估算上游数据库所有白名单表的大小之和 (`source_size`)，如果下游数据库剩余空间小于 `source_size`，前置检查会返回错误；如果下游数据库剩余空间小于 TiKV 副本数 * `source_size` * 2，前置检查会返回警告。

- 下游数据库是否在运行与 Physical Import 不兼容的任务

  - 目前 Physical Import 不兼容TiCDC、 PITR 任务，如果发现下游数据库正在运行这些任务，前置检查会返回错误。

增量数据迁移检查项

对于增量数据迁移模式（`task-mode: incremental`），除了通用检查项，前置检查还会包含以下检查项：

- （必须）上游数据库的 REPLICATION 权限

  - 检查是否有 REPLICATION CLIENT 权限。
  - 检查是否有 REPLICATION SLAVE 权限。

- 数据库主从配置

  - 建议上游数据库设置数据库 ID `server_id`（非 AWS Aurora 环境建议开启 GTID），防止主从复制切换出错。

- （必须）MySQL binlog 配置

  - 检查 binlog 是否开启（DM 要求 binlog 必须开启）。
  - 检查是否有 `binlog_format=ROW`（DM 只支持 ROW 格式的 binlog 迁移）。
  - 检查是否有 `binlog_row_image=FULL`（DM 只支持 `binlog_row_image=FULL`）。
  - 如果配置了 `binlog_do_db` 或者 `binlog_ignore_db`，那么检查需要迁移的库表，是否满足 `binlog_do_db` 和 `binlog_ignore_db` 的条件。

- （必须）检查上游是否处于Online-DDL 过程中，即创建了 `ghost` 表，但还未执行 `rename` 的阶段。如果处于 online-DDL 中，则检查报错，请等待 DDL 结束后重试。

全量加增量数据迁移检查项

对于全量加增量数据迁移模式（`task-mode: all`），除了通用检查项，前置检查还会包含全量数据迁移检查项，以及增量数据迁移检查项。

可忽略检查项

一般情况下前置检查项用于提前发现环境中可能存在的风险，不建议忽略。如果你的数据迁移任务面临特殊场景，你可以通过`ignore-checking-items`配置项来跳过部分检查项。

值	含义
dump_privilege	检查上游 MySQL 实例用户的 dump 相关权限
replication_privilege	检查上游 MySQL 实例用户的 replication 相关权限
version	检查上游数据库版本
server_id	检查上游数据库是否设置 server_id
binlog_enable	检查上游数据库是否已启用 binlog
table_schema	检查上游 MySQL 表结构的兼容性
schema_of_shard_tables	检查上游 MySQL 多实例分库分表的表结构一致性
auto_increment_ID	检查上游 MySQL 多实例分库分表的自增主键冲突
online_ddl	检查上游是否处于Online-DDL 过程中
empty_region	Physical Import 检查空 Region 的数目
region_distribution	Physical Import 检查 Region 的分布
downstream_version	检查下游数据库 TiDB、 PD、 TiKV 的版本
free_space	检查下游数据库的剩余空间
downstream_mutex_features	检查下游数据库是否存在与 Physical Import 不兼容的任务

> **注意：**
>
> 6.0 之前的版本支持忽略更多的检查项，但诸如 `binlog_row_image` 等参数，若配置错误可能导致同步时丢失数据，因此在 6.0 版本中移除了部分与数据安全相关的检查项。

配置检查参数

任务前置检查支持多线程并行。即使分表数目达到万级别，检查也可以在分钟级完成。

你可以通过数据迁移任务配置文件里 `mydumpers` 字段中的 `threads` 参数指定线程的数量。

```
mydumpers:                          # dump 处理单元的运行配置参数
  global:                           # 配置名称
    threads: 4                      # dump
        ↪ 处理单元从上游数据库实例导出数据和执行前置检查时访问上游的线程数量，默认值为 4
    chunk-filesize: 64              # dump 处理单元生成的数据文件大小，默认值为 64，单位为 MB
    extra-args: "--consistency none" # dump 处理单元的其他参数，不需要在 extra-args 中配置 table-
        ↪ list，DM 会自动生成
```

> **注意：**
>
> `threads` 参数的值决定了上游和 DM 之间的物理连接数。过大的 `threads` 可能会加大上游的负载，请注意控制 `threads` 大小。

13.6.6.13.2 创建 TiDB Data Migration 数据迁移任务

`start-task` 命令用于创建数据迁移任务。当数据迁移任务启动时，TiDB Data Migration (DM) 将自动对相应权限和配置进行前置检查。

```
help start-task
```

```
Starts a task as defined in the configuration file

Usage:
  dmctl start-task [-s source ...] [--remove-meta] <config-file> [flags]

Flags:
  -h, --help                help for start-task
      --remove-meta         whether to remove task's meta data
      --start-time string   specify the start time of binlog replication, e.g. '2021-10-21
          ↪ 00:01:00' or 2021-10-21T00:01:00

Global Flags:
      --config string        Path to config file.
      --master-addr string   Master API server address, this parameter is required when
          ↪ interacting with the dm-master
      --rpc-timeout string   RPC timeout, default is 10m. (default "10m")
  -s, --source strings       MySQL Source ID.
      --ssl-ca string        Path of file that contains list of trusted SSL CAs for connection.
      --ssl-cert string      Path of file that contains X509 certificate in PEM format for
          ↪ connection.
      --ssl-key string       Path of file that contains X509 key in PEM format for connection.
  -V, --version              Prints version and exit.
```

命令用法示例

```
start-task [ -s "mysql-replica-01"] ./task.yaml
```

参数解释

- `-s`：
  - 可选

  - 指定在特定的一个 MySQL 源上执行 `task.yaml`
  - 如果设置，则只启动指定任务在该 MySQL 源上的子任务

- `config-file`：
  - 必选
  - 指定 `task.yaml` 的文件路径

- `remove-meta`:
  - 可选
  - 如果设置，则在启动指定任务时会移除该任务之前存在的 metadata

- `start-time`:
  - 可选，格式为 `'2021-10-21 00:01:00'` 或 `2021-10-21T00:01:00`
  - 对于增量任务，可以通过该参数大致指定任务起始位点，该参数比任务配置文件中的 binlog 位置优先级更高，也比下游 checkpoint 中的 binlog 位置优先级更高
  - 当该任务存在 checkpoint 时，如果通过这种方式启动任务，DM 会自动开启 safe mode 直到同步过 checkpoint，以避免重置任务到更早位置时遇到数据重复的报错。向前重置起始位点时，如果起始位点的表结构与下游当前表结构不一致可能会在同步时报错；向后重置起始位点时，需要注意跳过的 binlog 可能在下游残留脏数据
  - 指定了过早的时间时，会从最早的 binlog 开始同步
  - 指定了过晚的时间时，会报错 `start-time {input-time} is too late, no binlog location matches it`

返回结果示例

```
start-task task.yaml
```

```
{
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-01",
            "worker": "worker1"
        }
    ]
}
```

13.6.6.13.3 TiDB Data Migration 查询任务状态

本文介绍 TiDB Data Migration (DM) `query-status` 命令的查询结果、任务状态与子任务状态。

查询结果

```
» query-status
```

```
{
    "result": true,     # 查询是否成功
    "msg": "",          # 查询失败原因描述
    "tasks": [          # 迁移 task 列表
        {
            "taskName": "test",         # 任务名称
            "taskStatus": "Running",    # 任务运行状态
            "sources": [                # 该任务的上游 MySQL 列表
                "mysql-replica-01",
                "mysql-replica-02"
            ]
        },
        {
            "taskName": "test2",
            "taskStatus": "Paused",
            "sources": [
                "mysql-replica-01",
                "mysql-replica-02"
            ]
        }
    ]
}
```

关于 tasks 下的 taskStatus 状态的详细定义，请参阅任务状态。

推荐的 query-status 使用方法是：

1. 首先使用 query-status 查看各个 task 的运行状态是否正常。
2. 如果发现其中某一 task 状态有问题，通过 query-status <出错任务的 taskName> 来得到更详细的错误信息。

任务状态

DM 的迁移任务状态取决于其分配到 DM-worker 上的子任务状态，定义见下表：

任务对应的所有子任务的状态	任务状态
任一子任务处于“Paused”状态且返回结果有错误信息	Error - Some error occurred in subtask
任一处于 Sync 阶段的子任务处于“Running”状态但其 Relay 处理单元未运行 (处于 Error/-Paused/Stopped 状态)	Error - Relay status is Error/-Paused/Stopped

任务对应的所有子任务的状态	任务状态
任一子任务处于“Paused”状态且返回结果没有错误信息	Paused
所有子任务处于“New”状态	New
所有子任务处于“Finished”状态	Finished
所有子任务处于“Stopped”状态	Stopped
其他情况	Running

详情查询结果

```
» query-status test
```

```
{
    "result": true,     # 查询是否成功
    "msg": "",          # 查询失败原因描述
    "sources": [        # 上游 MySQL 列表
        {
            "result": true,
            "msg": "",
            "sourceStatus": {                   # 上游 MySQL 的信息
                "source": "mysql-replica-01",
                "worker": "worker1",
                "result": null,
                "relayStatus": null
            },
            "subTaskStatus": [              # 上游 MySQL 所有子任务的信息
                {
                    "name": "test",         # 子任务名称
                    "stage": "Running",     # 子任务运行状态，包括 “New”，“Running”，“Paused
                        ↪ ”，“Stopped” 以及 “Finished”
                    "unit": "Sync",         # DM 的处理单元，包括 “Check”，“Dump“，“Load”
                        ↪ 以及 “Sync”
                    "result": null,         # 子任务失败时显示错误信息
                    "unresolvedDDLLockID": "test-`test`.`t_target`",    # sharding DDL lock ID,
                        ↪ 可用于异常情况下手动处理 sharding DDL lock
                    "sync": {                   # 当前 `Sync` 处理单元的迁移信息
                        "masterBinlog": "(bin.000001, 3234)",                               #
                            ↪ 上游数据库当前的 binlog position
                        "masterBinlogGtid": "c0149e17-dff1-11e8-b6a8-0242ac110004:1-14",    #
                            ↪ 上游数据库当前的 GTID 信息
                        "syncerBinlog": "(bin.000001, 2525)",                               #
                            ↪ 已被 `Sync` 处理单元迁移的 binlog position
                        "syncerBinlogGtid": "",                                             #
                            ↪ 使用 GTID 迁移的 binlog position
                        "blockingDDLs": [       # 当前被阻塞的 DDL 列表。该项仅在当前 DM-worker
                            ↪ 所有上游表都处于 “synced“ 状态时才有数值，
                            ↪ 此时该列表包含的是待执行或待跳过的 sharding DDL 语句
                            "USE `test`; ALTER TABLE `test`.`t_target` DROP COLUMN `age`;"
                        ],
                        "unresolvedGroups": [   # 没有被解决的 sharding group 信息
                            {
                                "target": "`test`.`t_target`",                  # 待迁移的下游表
                                "DDLs": [
                                    "USE `test`; ALTER TABLE `test`.`t_target` DROP COLUMN `age
                                        ↪ `;"
                                ],
```

```
                        "firstPos": "(bin|000001.000001, 3130)",        # sharding DDL
                            ↪ 语句起始 binlog position
                        "synced": [                                     # `Sync`
                            ↪ 处理单元已经读到该 sharding DDL 的上游分表
                            "`test`.`t2`"
                            "`test`.`t3`"
                            "`test`.`t1`"
                        ],
                        "unsynced": [                                   # `Sync`
                            ↪ 处理单元未读到该 sharding DDL 的上游分表。
                            ↪ 如有上游分表未完成同步，`blockingDDLs` 为空
                        ]
                    }
                ],
                "synced": false,        # 增量复制是否已追上上游。由于后台 `Sync`
                    ↪ 单元并不会实时刷新保存点，当前值为 `false`
                    ↪ 并不一定代表发生了迁移延迟
                "totalRows": "12",      # 该子任务中迁移的行数
                "totalRps": "1",        # 该子任务中每秒迁移的行数
                "recentRps": "1"        # 该子任务中最后一秒迁移的行数
            }
        }
    ]
},
{
    "result": true,
    "msg": "",
    "sourceStatus": {
        "source": "mysql-replica-02",
        "worker": "worker2",
        "result": null,
        "relayStatus": null
    },
    "subTaskStatus": [
        {
            "name": "test",
            "stage": "Running",
            "unit": "Load",
            "result": null,
            "unresolvedDDLLockID": "",
            "load": {                                   # `Load` 处理单元的迁移信息
                "finishedBytes": "115",                 # 已全量导入的字节数
                "totalBytes": "452",                    # 总计需要导入的字节数
                "progress": "25.44 %",                  # 全量导入进度
                "bps": "2734"                           # 全量导入速度
```

```
            }
        }
    ]
},
{
    "result": true,
    "sourceStatus": {
        "source": "mysql-replica-03",
        "worker": "worker3",
        "result": null,
        "relayStatus": null
    },
    "subTaskStatus": [
        {
            "name": "test",
            "stage": "Paused",
            "unit": "Load",
            "result": {                 # 错误示例
                "isCanceled": false,
                "errors": [
                    {
                        "Type": "ExecSQL",
                        "msg": "Error 1062: Duplicate entry '1155173304420532225' for key
                            ↪  'PRIMARY'\n/home/jenkins/workspace/build_dm/go/src/
                            ↪ github.com/pingcap/tidb-enterprise-tools/loader/db.go
                            ↪ :160: \n/home/jenkins/workspace/build_dm/go/src/github.
                            ↪ com/pingcap/tidb-enterprise-tools/loader/db.go:105: \n/
                            ↪ home/jenkins/workspace/build_dm/go/src/github.com/pingcap
                            ↪ /tidb-enterprise-tools/loader/loader.go:138: file test.t1
                            ↪ .sql"
                    }
                ],
                "detail": null
            },
            "unresolvedDDLLockID": "",
            "load": {
                "finishedBytes": "0",
                "totalBytes": "156",
                "progress": "0.00 %",
                "bps": "0"
            }
        }
    ]
},
{
```

```
            "result": true,
            "msg": "",
            "sourceStatus": {
                "source": "mysql-replica-04",
                "worker": "worker4",
                "result": null,
                "relayStatus": null
            },
            "subTaskStatus": [
                {
                    "name": "test",
                    "stage": "Running",
                    "unit": "Dump",
                    "result": null,
                    "unresolvedDDLLockID": "",
                    "dump": {                       # `Dump` 处理单元的迁移信息
                        "totalTables": "10",        # 需要 dump 的表数量
                        "completedTables": "3",     # 已 dump 的表数量
                        "finishedBytes": "2542",    # 已 dump 的字节数
                        "finishedRows": "32",       # 已 dump 的行数
                        "estimateTotalRows": "563", # 预估需要 dump 的行数
                        "progress": "30.52 %",      # dump 的进度
                        "bps": "445"                # dump 的速度
                    }
                }
            ]
        },
    ]
}
```

关于 sources 下 subTaskStatus 中 stage 状态和状态转换关系的详细信息，请参阅子任务状态。

关于 sources 下 subTaskStatus 中 unresolvedDDLLockID的操作细节，请参阅手动处理 Sharding DDL Lock。

子任务状态

状态描述

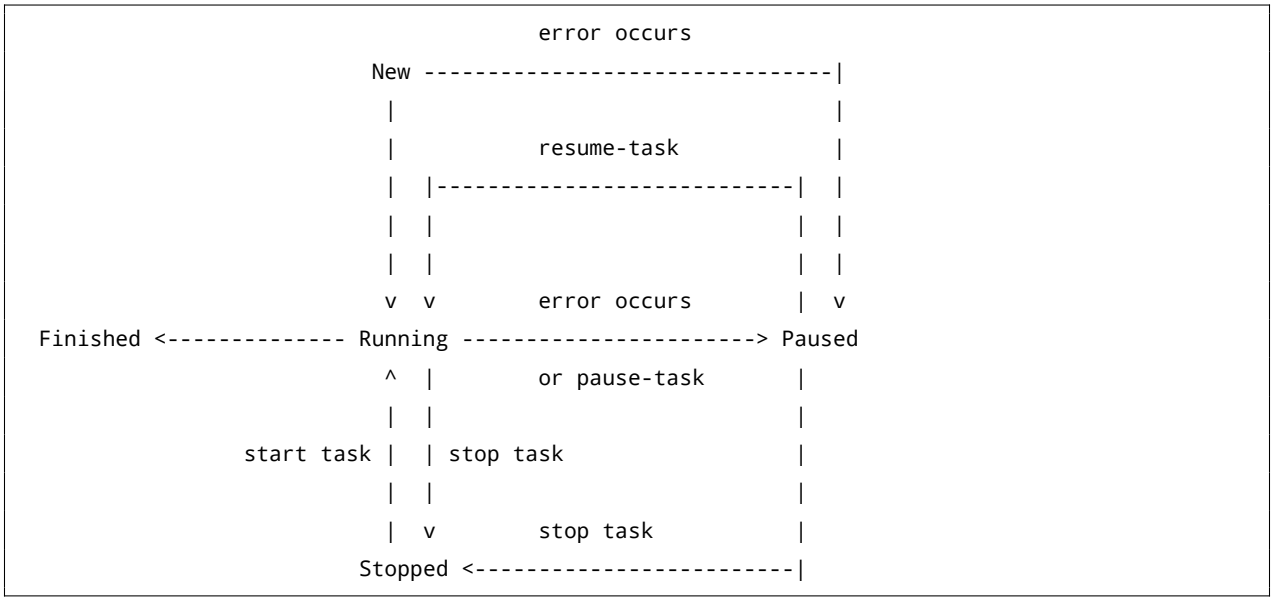
- New：
  - 初始状态。
  - 如果子任务没有发生错误，状态切换为 Running，其他情况则切换为 Paused。
- Running：正常运行状态。
- Paused：
  - 暂停状态。

- 子任务发生错误，状态切换为 Paused。
- 如在子任务为 Running 状态下执行 pause-task 命令，任务状态会切换为 Paused。
- 如子任务处于该状态，可以使用 resume-task 命令恢复任务。

- Stopped：
  - 停止状态。
  - 如在子任务为 Running 或 Paused 状态下执行 stop-task 命令，任务状态会切换为 Stopped。
  - 如子任务处于该状态，不可使用 resume-task 命令恢复任务。

- Finished：
  - 任务完成状态。
  - 只有 task-mode 为 full 的任务正常完成后，任务才会切换为该状态。

状态转换图



13.6.6.13.4 暂停 TiDB Data Migration 数据迁移任务

pause-task 命令用于暂停数据迁移任务。

> 注意：
>
> 有关 pause-task 与 stop-task 的区别如下：
>
> - 使用 pause-task 仅暂停迁移任务的执行，但仍然会在内存中保留任务的状态信息等，且可通过 query-status 进行查询；使用 stop-task 会停止迁移任务的执行，并移除内存中与该任务相关的信息，且不可再通过 query-status 进行查询，但不会移除已经写入到下游数据库中的数据以及其中的 checkpoint 等 dm_meta 信息。

- 使用 pause-task 暂停迁移任务期间，由于任务本身仍然存在，因此不能再启动同名的新任务，且会阻止对该任务所需 relay log 的清理；使用 stop-task 停止任务后，由于任务不再存在，因此可以再启动同名的新任务，且不会阻止对 relay log 的清理。
- pause-task 一般用于临时暂停迁移任务以排查问题等；stop-task 一般用于永久删除迁移任务或通过与 start-task 配合以更新配置信息。

```
help pause-task
```

```
pause a specified running task

Usage:
 dmctl pause-task [-s source ...] <task-name | task-file> [flags]

Flags:
 -h, --help   help for pause-task

Global Flags:
 -s, --source strings   MySQL Source ID
```

命令用法示例

```
pause-task [-s "mysql-replica-01"] task-name
```

参数解释

- -s：
  - 可选
  - 指定在特定的一个 MySQL 源上暂停数据迁移任务的子任务
  - 如果设置，则只暂停该任务在指定 MySQL 源上的子任务
- task-name| task-file：
  - 必选
  - 指定任务名称或任务文件路径

返回结果示例

```
pause-task test
```

```
{
    "op": "Pause",
    "result": true,
    "msg": "",
    "sources": [
```

```
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-01",
            "worker": "worker1"
        }
    ]
}
```

13.6.6.13.5 恢复 TiDB Data Migration 数据迁移任务

`resume-task` 命令用于恢复处于 Paused 状态的数据迁移任务，通常用于在人为处理完造成迁移任务暂停的故障后手动恢复迁移任务。

```
help resume-task
```

```
resume a specified paused task

Usage:
 dmctl resume-task [-s source ...] <task-name | task-file> [flags]

Flags:
 -h, --help   help for resume-task

Global Flags:
 -s, --source strings   MySQL Source ID
```

命令用法示例

```
resume-task [-s "mysql-replica-01"] task-name
```

参数解释

- `-s`:
  - 可选
  - 指定在特定的一个 MySQL 源上恢复数据迁移任务的子任务
  - 如果设置，则只恢复该任务在指定 MySQL 源上的子任务
- `task-name | task-file`:
  - 必选
  - 指定任务名称或任务文件路径

返回结果示例

```
resume-task test
```

```
{
    "op": "Resume",
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-01",
            "worker": "worker1"
        }
    ]
}
```

13.6.6.13.6 停止 TiDB Data Migration 数据迁移任务

stop-task 命令用于停止数据迁移任务。有关 stop-task 与 pause-task 的区别，请参考暂停数据迁移任务中的相关说明。

```
help stop-task
```

```
stop a specified task

Usage:
 dmctl stop-task [-s source ...] <task-name | task-file> [flags]

Flags:
 -h, --help   help for stop-task

Global Flags:
 -s, --source strings   MySQL Source ID
```

命令用法示例

```
stop-task [-s "mysql-replica-01"]  task-name
```

参数解释

- -s：
  - 可选
  - 指定在特定的一个 MySQL 源上停止数据迁移任务的子任务
  - 如果设置，则只停止该任务在指定 MySQL 源上的子任务
- task-name | task-file：
  - 必选

- 指定任务名称或任务文件路径

返回结果示例

```
stop-task test
```

```
{
    "op": "Stop",
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-01",
            "worker": "worker1"
        }
    ]
}
```

13.6.7 进阶教程

13.6.7.1 分库分表合并迁移

13.6.7.1.1 分库分表合并迁移

本文介绍了 DM 提供的分库分表的合并迁移功能，此功能可用于将上游 MySQL/MariaDB 实例中结构相同/不同的表迁移到下游 TiDB 的同一个表中。DM 不仅支持迁移上游的 DML 数据，也支持协调迁移多个上游分表的 DDL 表结构变更。

简介

DM 支持对上游多个分表的数据合并迁移到 TiDB 的一个表中，在迁移过程中需要协调各个分表的 DDL，以及该 DDL 前后的 DML。针对用户的使用场景，DM 支持悲观协调和乐观协调两种不同的模式。

> 注意：
>
> - 要执行分库分表合并迁移任务，必须在任务配置文件中设置 `shard-mode`。
> - DM 对分库分表的合并默认使用悲观协调模式，在文档中如果没特殊说明则默认为悲观协调模式。
> - 在没有深入了解乐观模式的原理和使用限制的情况下不建议使用该模式，否则可能造成迁移中断甚至数据不一致的严重后果。

悲观协调模式

当上游一个分表执行某一 DDL 后，这个分表的迁移会暂停，并等待其他所有分表都执行了同样的 DDL 才在下游执行该 DDL 并继续数据迁移。“悲观协调”模式的优点是可以保证迁移到下游的数据不会出错，详细的介绍请参考悲观模式下分库分表合并迁移。

乐观协调模式

在一个分表上执行的 DDL，DM 会自动修改成兼容其他分表的语句，并立即迁移到下游，不会阻挡任何分表 DML 的迁移。“乐观协调”模式的优点是处理 DDL 时不会阻塞数据的迁移，但是使用不当会有迁移中断甚至数据不一致的风险，详细的介绍请参考乐观模式下分库分表合并迁移。

悲观协调模式与乐观协调模式的对比

悲观协调模式	乐观协调模式
发起 DDL 的分表会暂停 DML 迁移	发起 DDL 的分表会继续 DML 迁移
每个分表的 DDL 执行次序和语句必须相同	每个分表只需保持表结构互相兼容即可
DDL 在整个分表群达成一致后才迁移到下游	每个分表的 DDL 会即时影响下游
错误的 DDL 操作在侦测到后可以被拦截	错误的 DDL 操作也会被迁移到下游，可能在侦测到之前已使部分上下游数据不一

13.6.7.1.2 悲观模式下分库分表合并迁移

本文介绍了 DM 提供的悲观模式（默认模式）下分库分表合并迁移功能。此功能用于将上游 MySQL/MariaDB 实例中结构相同的表迁移到下游 TiDB 的同一个表中。

使用限制

DM 在悲观模式下进行分表 DDL 的迁移有以下几点使用限制：

- 对于一个逻辑 sharding group（需要合并迁移到下游同一个表的所有分表组成的 group），只能使用一个仅包含这些分表所在数据源的任务进行迁移。
- 在一个逻辑 sharding group 内，所有上游分表必须以相同的顺序执行相同的 DDL 语句（库名和表名可以不同），并且只有在所有分表执行完当前一条 DDL 语句后，下一条 DDL 语句才能执行。
    - 比如，如果在 table_1 表中先增加列 a 后再增加列 b，则在 table_2 表中就不能先增加列 b 后再增加列 a，因为 DM 不支持以不同的顺序来执行相同的 DDL 语句。
- 在一个逻辑 sharding group 内，所有上游分表都应该执行对应的 DDL 语句。
    - 比如，若 DM-worker-2 对应的一个或多个上游分表未执行 DDL 语句，则其他已执行 DDL 语句的 DM-worker 都会暂停迁移任务，直到等到 DM-worker-2 收到上游对应的 DDL 语句。
- sharding group 数据迁移任务不支持 `DROP DATABASE/TABLE` 语句。
    - DM-worker 中的 binlog 复制单元（sync）会自动忽略掉上游分表的 `DROP DATABASE` 和 `DROP TABLE` 语句。
- sharding group 数据迁移任务不支持 `TRUNCATE TABLE` 语句。
    - DM-worker 中的 binlog 复制单元（sync）会自动忽略掉上游分表的 `TRUNCATE TABLE` 语句。

- sharding group 数据迁移任务支持 RENAME TABLE 语句，但有如下限制（online DDL 中的 RENAME 有特殊方案进行支持）：
  - 只支持 RENAME TABLE 到一个不存在的表。
  - 一条 RENAME TABLE 语句只能包含一个 RENAME 操作。
- sharding group 数据迁移任务要求一个 DDL 语句仅包含对一张表的操作。
- 增量复制任务需要确认开始迁移的 binlog position 上各分表的表结构必须一致，才能确保来自不同分表的 DML 语句能够迁移到表结构确定的下游，并且后续各分表的 DDL 语句能够正确匹配与迁移。
- 如果需要变更table routing 规则，必须先等所有 sharding DDL 语句迁移完成。
  - 在 sharding DDL 语句迁移过程中，使用 dmctl 尝试变更 router-rules 会报错。
- 如果需要创建新表加入到一个正在执行 DDL 语句的 sharding group 中，则必须保持新表结构和最新更改的表结构一致。
  - 比如，原 table_1, table_2 表初始时有 (a, b) 两列，sharding DDL 语句执行后有 (a, b, c) 三列，则迁移完成后新创建的表也应当有 (a, b, c) 三列。
- 由于已经收到 DDL 语句的 DM-worker 会暂停任务以等待其他 DM-worker 收到对应的 DDL 语句，因此数据迁移延迟会增加。

背景

目前，DM 使用 ROW 格式的 binlog 进行数据迁移，且 binlog 中不包含表结构信息。在 ROW 格式的 binlog 迁移过程中，如果不需要将多个上游表合并迁移到下游的同一个表，则只存在一个上游表的 DDL 语句会更新对应下游表结构。ROW 格式的 binlog 可以认为是具有 self-description 属性。

分库分表合并迁移过程中，可以根据 column values 及下游的表结构构造出相应的 DML 语句，但此时若上游的分表执行 DDL 语句进行了表结构变更，则必须对该 DDL 语句进行额外迁移处理，以避免因为表结构和 binlog 数据不一致而造成迁移出错的问题。

以下是一个简化后的例子：

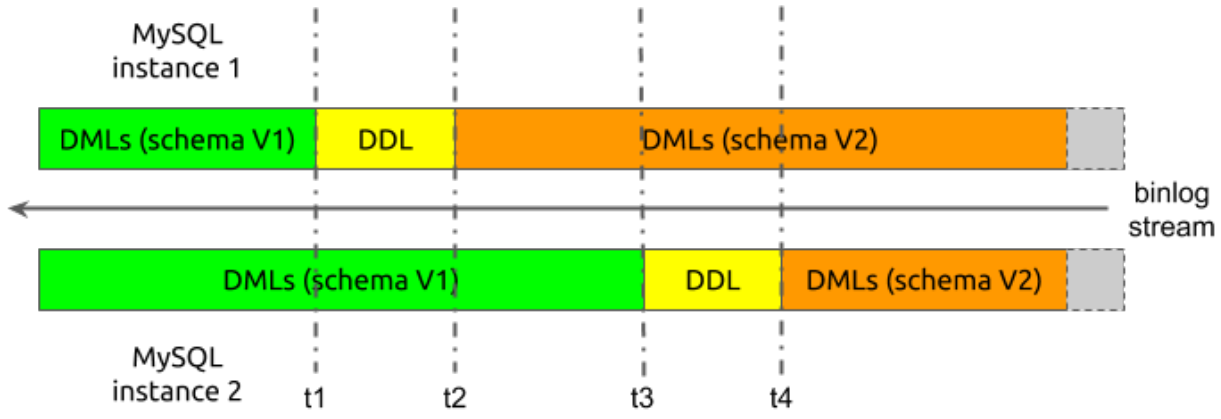


图 225: shard-ddl-example-1

在上图的例子中，分表的合库合表过程简化成了上游只有两个 MySQL 实例，每个实例内只有一个表。假设在数据迁移开始时，将两个分表的表结构版本记为 schema V1，将 DDL 语句执行完后的表结构版本记为 schema V2。

现在，假设数据迁移过程中，DM-worker 内的 binlog 复制单元（sync）从两个上游分表收到的 binlog 数据有如下时序：

1. 开始迁移时，sync 从两个分表收到的都是 schema V1 版本的 DML 语句。
2. 在 t1 时刻，sync 收到实例 1 上分表的 DDL 语句。
3. 从 t2 时刻开始，sync 从实例 1 收到的是 schema V2 版本的 DML 语句；但从实例 2 收到的仍是 schema V1 版本的 DML 语句。
4. 在 t3 时刻，sync 收到实例 2 上分表的 DDL 语句。
5. 从 t4 时刻开始，sync 从实例 2 收到的也是 schema V2 版本的 DML 语句。

假设在数据迁移过程中，不对分表的 DDL 语句进行额外处理。当实例 1 的 DDL 语句迁移到下游后，下游的表结构会变更成为 schema V2 版本。但在 t2 到 t3 这段时间内，sync 从实例 2 上收到的仍是 schema V1 版本的 DML 语句。当尝试把这些 schema V1 版本的 DML 语句迁移到下游时，就会由于 DML 语句与表结构的不一致而发生错误，从而无法正确迁移数据。

实现原理

基于上述例子，本部分介绍了 DM 在默认的悲观模式下合库合表过程中进行 DDL 迁移的实现原理。

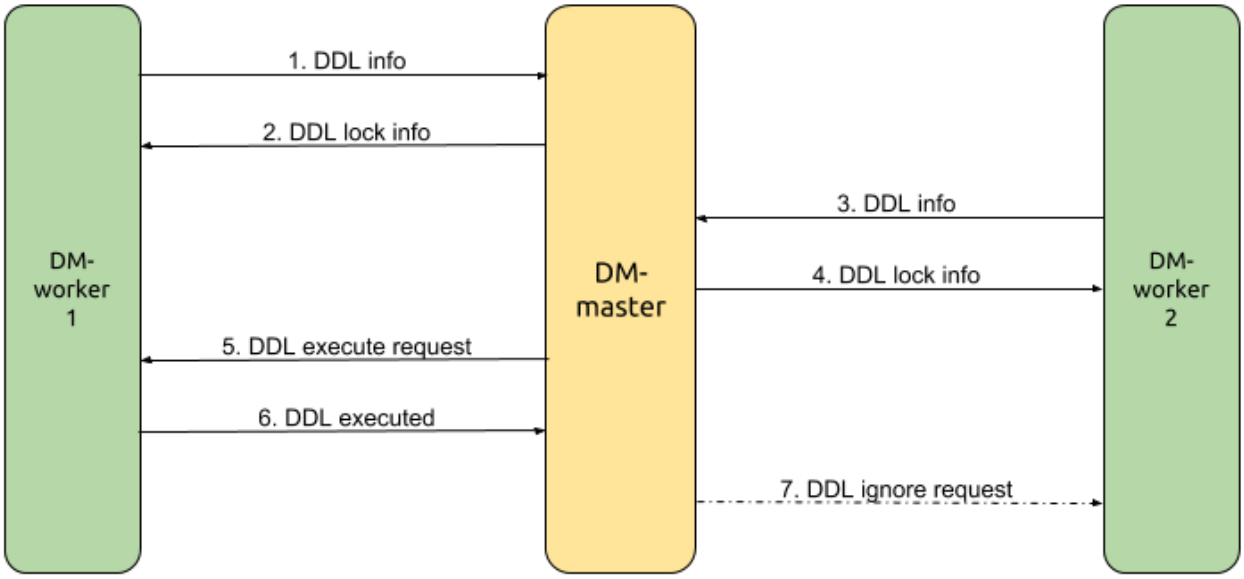


图 226: shard-ddl-flow

在这个例子中，DM-worker-1 负责迁移来自 MySQL 实例 1 的数据，DM-worker-2 负责迁移来自 MySQL 实例 2 的数据，DM-master 负责协调多个 DM-worker 间的 DDL 迁移。

从 DM-worker-1 收到 DDL 语句开始，简化后的 DDL 迁移流程为：

1. 在 t1 时刻，DM-worker-1 收到来自 MySQL 实例 1 的 DDL 语句，自身暂停该 DDL 语句对应任务的 DDL 及 DML 数据迁移，并将 DDL 相关信息发送给 DM-master。
2. DM-master 根据收到的 DDL 信息判断得知需要协调该 DDL 语句的迁移，于是为该 DDL 语句创建一个锁，并将 DDL 锁信息发回给 DM-worker-1，同时将 DM-worker-1 标记为这个锁的 owner。
3. DM-worker-2 继续进行 DML 语句的迁移，直到在 t3 时刻收到来自 MySQL 实例 2 的 DDL 语句，自身暂停该 DDL 语句对应任务的数据迁移，并将 DDL 相关信息发送给 DM-master。
4. DM-master 根据收到的 DDL 信息判断得知该 DDL 语句对应的锁信息已经存在，于是直接将对应锁信息发回给 DM-worker-2。
5. 根据任务启动时的配置信息、上游 MySQL 实例分表信息、部署拓扑信息等，DM-master 判断得知自身已经收到了来自待合表的所有上游分表的 DDL 语句，于是请求 DDL 锁的 owner（DM-worker-1）向下游迁移执行该 DDL。
6. DM-worker-1 根据第二步收到的 DDL 锁信息验证 DDL 语句执行请求；向下游执行 DDL，并将执行结果反馈给 DM-master；若 DDL 语句执行成功，则自身开始继续迁移后续的（从 t2 时刻对应的 binlog 开始的）DML 语句。
7. DM-master 收到来自 owner 执行 DDL 语句成功的响应，于是请求在等待该 DDL 锁的所有其他 DM-worker（DM-worker-2）忽略该 DDL 语句，直接继续迁移后续的（从 t4 时刻对应的 binlog 开始的）DML 语句。

根据上面的流程，可以归纳出 DM 协调多个 DM-worker 间 sharding DDL 迁移的特点：

- 根据任务配置与 DM 集群部署拓扑信息，DM-master 内部也会建立一个逻辑 sharding group 来协调 DDL 迁移，group 中的成员为负责处理该迁移任务拆解后的各子任务的 DM-worker。
- 各 DM-worker 从 binlog event 中收到 DDL 语句后，会将 DDL 信息发送给 DM-master。
- DM-master 根据来自 DM-worker 的 DDL 信息及 sharding group 信息创建或更新 DDL 锁。
- 如果 sharding group 的所有成员都收到了某一条相同的 DDL 语句，则表明上游分表在该 DDL 执行前的 DML 语句都已经迁移完成，此时可以执行该 DDL 语句，并继续后续的 DML 迁移。
- 上游所有分表的 DDL 在经过table router 转换后需要保持一致，因此仅需 DDL 锁的 owner 执行一次该 DDL 语句即可，其他 DM-worker 可直接忽略对应的 DDL 语句。

在上面的示例中，每个 DM-worker 对应的上游 MySQL 实例中只有一个待合并的分表。但在实际场景下，一个 MySQL 实例可能有多个分库内的多个分表需要进行合并，这种情况下，sharding DDL 的协调迁移过程将更加复杂。

假设同一个 MySQL 实例中有 table_1 和 table_2 两个分表需要进行合并：

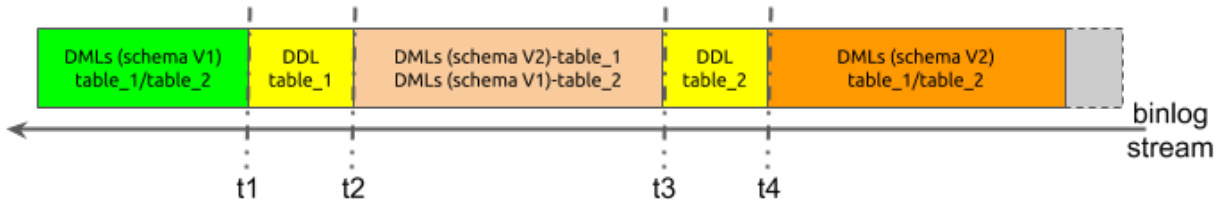


图 227: shard-ddl-example-2

在这个例子中，由于数据来自同一个 MySQL 实例，因此所有数据都是从同一个 binlog 流中获得，时序如下：

1. 开始迁移时，DM-worker 内的 sync 从两个分表收到的数据都是 schema V1 版本的 DML 语句。
2. 在 t1 时刻，sync 收到 table_1 分表的 DDL 语句。
3. 从 t2 到 t3 时刻，sync 收到的数据同时包含 table_1 的 DML 语句（schema V2 版本）及 table_2 的 DML 语句（schema V1 版本）。
4. 在 t3 时刻，sync 收到 table_2 分表的 DDL 语句。
5. 从 t4 时刻开始，sync 从两个分表收到的数据都是 schema V2 版本的 DML 语句。

假设在数据迁移过程中，不对分表的 DDL 语句进行额外处理。当 table_1 的 DDL 语句迁移到下游从而变更下游表结构后，table_2 的 DML 语句（schema V1 版本）将无法正常迁移。因此，在单个 DM-worker 内部，我们也构造了与 DM-master 内类似的逻辑 sharding group，但 group 的成员是同一个上游 MySQL 实例的不同分表。

DM-worker 内协调处理 sharding group 的迁移与 DM-master 处理 DM-worker 之间的迁移不完全一致，主要原因包括：

- 当 DM-worker 收到 table_1 分表的 DDL 语句时，迁移不能暂停，需要继续解析 binlog 才能获得后续 table_2 分表的 DDL 语句，即需要从 t2 时刻继续解析直到 t3 时刻。
- 在继续解析 t2 到 t3 时刻的 binlog 的过程中，table_1 分表的 DML 语句（schema V2 版本）不能向下游迁移；但当 sharding DDL 迁移并执行成功后，这些 DML 语句则需要迁移到下游。

DM-worker 内部 sharding DDL 迁移的简化流程为：

1. 在 t1 时刻，DM-worker 收到 table_1 的 DDL 语句，并记录 DDL 信息及此时的 binlog 位置点信息。
2. DM-worker 继续向前解析 t2 到 t3 时刻的 binlog。
3. 对于 table_1 的 DML 语句（schema V2 版本），忽略；对于 table_2 的 DML 语句（schema V1 版本），正常迁移到下游。
4. 在 t3 时刻，DM-worker 收到 table_2 的 DDL 语句，并记录 DDL 信息及此时的 binlog 位置点信息。
5. 根据迁移任务配置信息、上游库表信息等，DM-worker 判断得知该 MySQL 实例上所有分表的 DDL 语句都已收到；于是将该 DDL 语句迁移到下游执行并变更下游表结构。
6. DM-worker 设置 binlog 流的新解析起始位置点为第一步时保存的位置点。
7. DM-worker 重新开始解析从 t2 到 t3 时刻的 binlog。
8. 对于 table_1 的 DML 语句（schema V2 版本），正常迁移到下游；对于 table_2 的 DML 语句（schema V1 版本），忽略。
9. 解析到达第四步时保存的 binlog 位置点，可得知在第三步时被忽略的所有 DML 语句都已经重新迁移到下游。
10. DM-worker 继续从 t4 时刻对应的 binlog 位置点开始正常迁移。

综上可知，DM 在处理 sharding DDL 迁移时，主要通过两级 sharding group 来进行协调控制，简化的流程为：

1. 各 DM-worker 独立地协调对应上游 MySQL 实例内多个分表组成的 sharding group 的 DDL 迁移。
2. 当 DM-worker 收到所有分表的 DDL 语句时，向 DM-master 发送 DDL 相关信息。
3. DM-master 根据 DM-worker 发来的 DDL 信息，协调由各 DM-worker 组成的 sharing group 的 DDL 迁移。
4. 当 DM-master 收到所有 DM-worker 的 DDL 信息时，请求 DDL 锁的 owner（某个 DM-worker）执行该 DDL 语句。
5. DDL 锁的 owner 执行 DDL 语句，并将结果反馈给 DM-master；自身开始重新迁移在内部协调 DDL 迁移过程中被忽略的 DML 语句。
6. 当 DM-master 收到 owner 执行 DDL 成功的消息后，请求其他所有 DM-worker 继续开始迁移。
7. 其他所有 DM-worker 各自开始重新迁移在内部协调 DDL 迁移过程中被忽略的 DML 语句。
8. 在完成被忽略的 DML 语句的重新迁移后，所有 DM-worker 继续正常迁移。

13.6.7.1.3 乐观模式下分库分表合并迁移

本文介绍了 DM 提供的乐观模式下分库分表的合并迁移功能，此功能可用于将上游 MySQL/MariaDB 实例中结构相同/不同的表迁移到下游 TiDB 的同一个表中。

> **注意：**
>
> 在没有深入了解乐观模式的原理和使用限制的情况下不建议使用该模式，否则可能造成迁移中断甚至数据不一致的严重后果。

背景

DM 支持在线上执行分库分表的 DDL 语句（通称 Sharding DDL），默认使用“悲观模式”，即当上游一个分表执行某一 DDL 后，这个分表的迁移会暂停，等待其他所有分表都执行了同样的 DDL 才在下游执行该 DDL 并继续数据迁移。这种“悲观协调”模式的优点是可以保证迁移到下游的数据不会出错，缺点是会暂停数据迁移而不利于对上游进行灰度变更。有些用户可能会花较长时间在单一分表执行 DDL，验证一定时间后才会更改其他分表的结构。在悲观模式迁移的设定下，这些 DDL 会阻塞迁移，binlog 事件会大量积压。

因此，需要提供一种新的“乐观协调”模式，在一个分表上执行的 DDL，自动修改成兼容其他分表的语句后，立即迁移到下游，不会阻挡任何分表执行的 DML 的迁移。

乐观协调模式的配置

在任务的配置文件中指定 `shard-mode` 为 `optimistic` 则使用“乐观协调”模式，示例配置文件可以参考DM 任务完整配置文件介绍。

使用限制

使用“乐观协调”模式有一定的风险，需要严格遵照以下方针：

- 执行每个批次的 DDL 前和后，要确保每个分表的结构达成一致。

- 进行灰度 DDL 时，只集中在一个分表上测试。
- 灰度完成后，在其他分表上尽量以最简单直接的 DDL 迁移到最终的 schema，而不要重新执行灰度测试中对或错的每一步。
  - 例如：在分表执行过 `ADD COLUMN A INT; DROP COLUMN A; ADD COLUMN A FLOAT;`，在其他分表直接执行 `ADD COLUMN A FLOAT` 即可，不需要三条 DDL 都执行一遍。
- 执行 DDL 时要注意观察 DM 迁移状态。当迁移报错时，需要判断这个批次的 DDL 是否会造成数据不一致。

“乐观协调”模式下，上游执行的大部分 DDL 无需特别关注，将自动同步，本文使用“一类 DDL”指代。同时，还有一部分改变列名、列属性或列默认值的 DDL，称为“二类 DDL”，上游执行时要注意必须保证该 DDL 在各分表中按相同的顺序执行。

“二类 DDL”举例如下：

- 修改列的类型：`ALTER TABLE table_name MODIFY COLUMN column_name VARCHAR(20)`。
- 重命名列：`ALTER TABLE table_name RENAME COLUMN column_1 TO column_2;`。
- 增加没有默认值且非空的列：`ALTER TABLE table_name ADD COLUMN column_1 NOT NULL;`。
- 重命名索引：`ALTER TABLE table_name RENAME INDEX index_1 TO index_2;`。

各分表在执行以上 DDL 时，若顺序不同将导致同步中断，例如下述场景：

- 分表 1 先重命名列，再修改列类型
  1. 重命名列：`ALTER TABLE table_name RENAME COLUMN column_1 TO column_2;`。
  2. 修改列类型：`ALTER TABLE table_name MODIFY COLUMN column_3 VARCHAR(20);`。
- 分表 2 先修改列类型，再重命名列
  1. 修改列类型：`ALTER TABLE table_name MODIFY COLUMN column_3 VARCHAR(20)`。
  2. 重命名列：`ALTER TABLE table_name RENAME COLUMN column_1 TO column_2;`。

此外，不论是使用“乐观协调”或“悲观协调”，DM 仍是有以下限制：

- 不支持 `DROP TABLE/DROP DATABASE`。
- 不支持 `TRUNCATE TABLE`。
- 单条 DDL 语句要求仅包含对一张表的操作。
- TiDB 不支持的 DDL 语句在 DM 也不支持。
- 新增列的默认值不能包含 `current_timestamp`、`rand()`、`uuid()` 等，否则会造成上下游数据不一致。

风险

使用乐观模式迁移时，由于 DDL 会即时迁移到下游，若使用不当，可能导致上下游数据不一致。

使数据不一致的操作

- 各分表的表结构不兼容，例：
  - 两个分表各自添加相同名称的列，但其类型不同。
  - 两个分表各自添加相同名称的列，但其默认值不同。

  - 两个分表各自添加相同名称的生成列，但其生成表达式不同。
  - 两个分表各自添加相同名称的索引，但其键组合不同。
  - 其他同名异构的情况。

- 在分表上执行对数据具有破坏性的 DDL，然后尝试回滚，例：
  - 删除一列 X，之后又把 X 加回来。

例子

例如以下三个分表合并迁移到 TiDB：

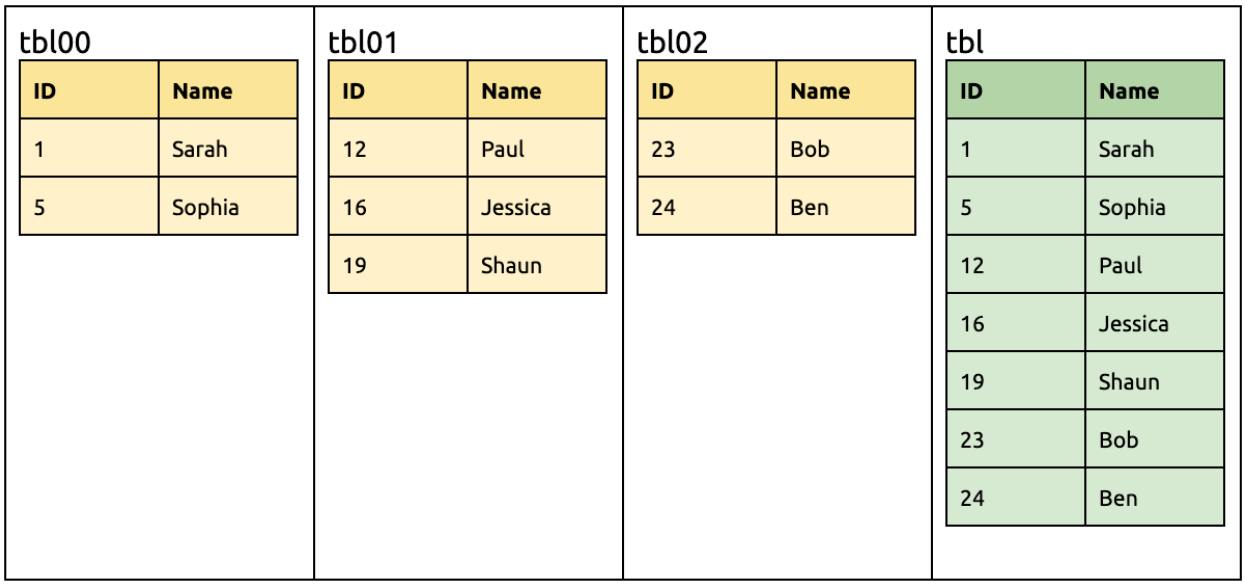


图 228: optimistic-ddl-fail-example-1

在 tbl01 新增一列 Age，默认值定为 0：

```
ALTER TABLE `tbl01` ADD COLUMN `Age` INT DEFAULT 0;
```

tbl00

ID	Name
1	Sarah
5	Sophia

tbl01

ID	Name	Age
12	Paul	0
16	Jessica	0
19	Shaun	0

tbl02

ID	Name
23	Bob
24	Ben

tbl

ID	Name	Age
1	Sarah	0
5	Sophia	0
12	Paul	0
16	Jessica	0
19	Shaun	0
23	Bob	0
24	Ben	0

图 229: optimistic-ddl-fail-example-2

在 tbl00 新增一列 Age，但默认值定为 -1：

```
ALTER TABLE `tbl00` ADD COLUMN `Age` INT DEFAULT -1;
```

tbl00

ID	Name	Age
1	Sarah	-1
5	Sophia	-1

tbl01

ID	Name	Age
12	Paul	0
16	Jessica	0
19	Shaun	0

tbl02

ID	Name
23	Bob
24	Ben

tbl

ID	Name	Age
1	Sarah	0
5	Sophia	0
12	Paul	0
16	Jessica	0
19	Shaun	0
23	Bob	0
24	Ben	0

图 230: optimistic-ddl-fail-example-3

此时所有来自 tbl00 的 Age 都不一致了。这是由于 DEFAULT 0 和 DEFAULT -1 互不兼容。虽然 DM 遇到这种情况会报错，但上下游不一致的问题就需要手动去解决。

原理

在“乐观协调”模式下，DM-worker 接收到来自上游的 DDL 后，会把更新后的表结构转送给 DM-master。DM-worker 会追踪各分表当前的表结构，DM-master 合并成可兼容来自每个分表 DML 的合成结构，然后把与此对应的 DDL 迁移到下游；对于 DML 会直接迁移到下游。

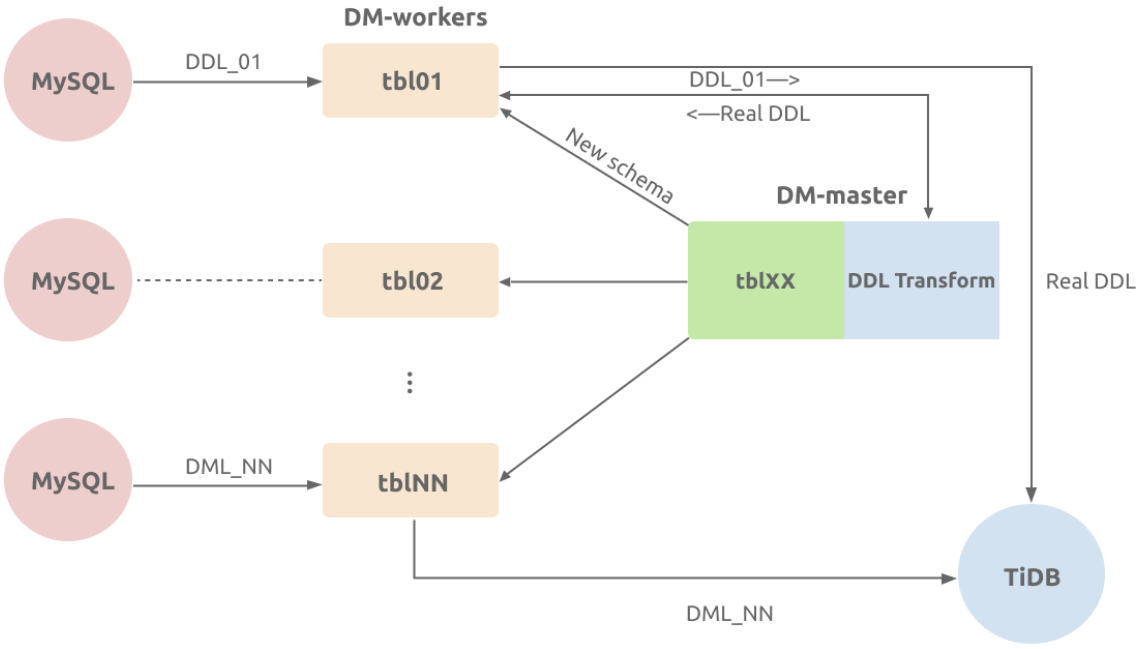


图 231: optimistic-ddl-flow

例子

例如上游 MySQL 有三个分表（`tbl00`, `tbl01` 以及 `tbl02`），使用 DM 迁移到下游 TiDB 的 `tbl` 表中，如下图所示：

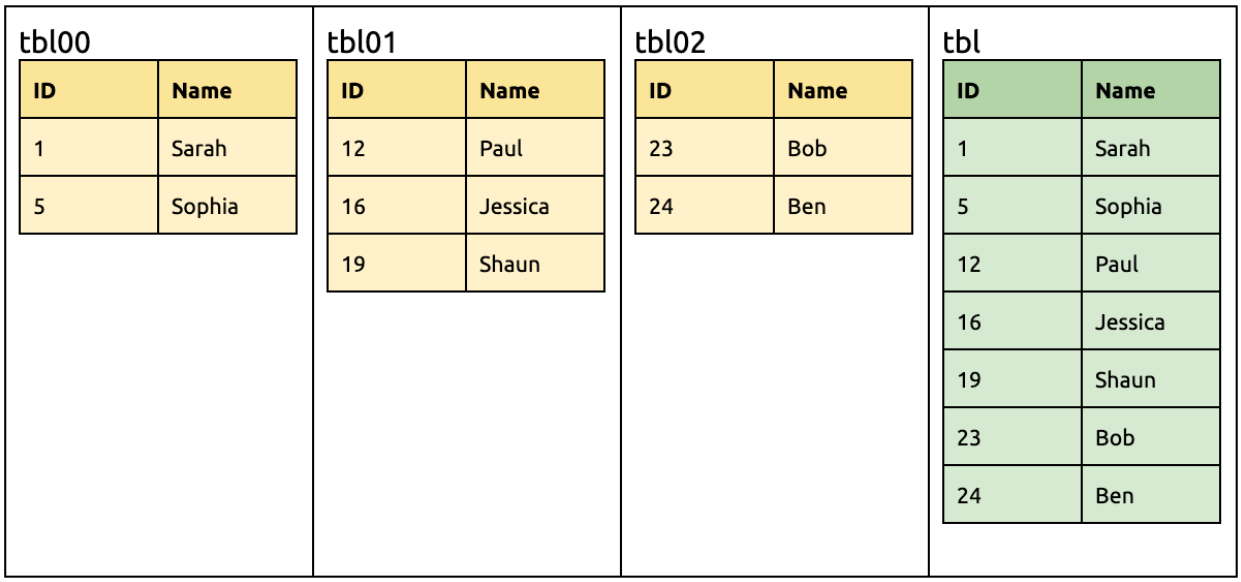


图 232: optimistic-ddl-example-1

在上游增加一列 `Level`：

```
ALTER TABLE `tbl00` ADD COLUMN `Level` INT;
```

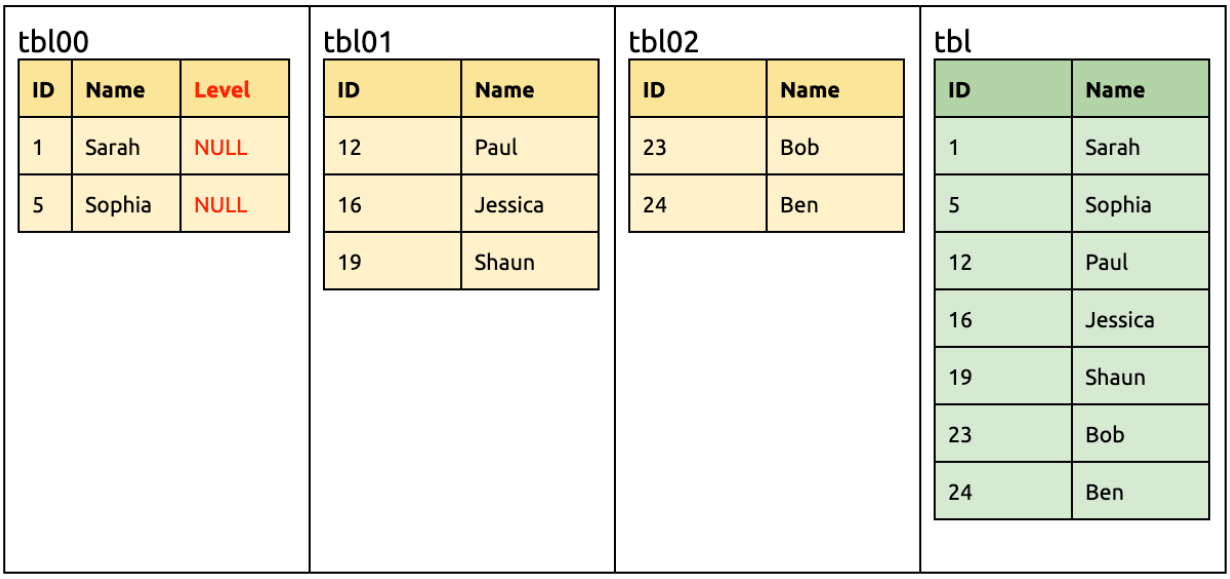


图 233: optimistic-ddl-example-2

此时下游 TiDB 要准备接受来自 `tbl00` 有 `Level` 的 DML、以及来自 `tbl01` 和 `tbl02` 没有 `Level` 的 DML。

tbl00

ID	Name	Level
1	Sarah	NULL
5	Sophia	NULL

tbl01

ID	Name
12	Paul
16	Jessica
19	Shaun

tbl02

ID	Name
23	Bob
24	Ben

tbl

ID	Name	Level
1	Sarah	NULL
5	Sophia	NULL
12	Paul	NULL
16	Jessica	NULL
19	Shaun	NULL
23	Bob	NULL
24	Ben	NULL

图 234: optimistic-ddl-example-3

这时候如下的 DML 无需修改就可以迁移到下游：

```
UPDATE `tbl00` SET `Level` = 9 WHERE `ID` = 1;
INSERT INTO `tbl02` (`ID`, `Name`) VALUES (27, 'Tony');
```

tbl00

ID	Name	Level
1	Sarah	9
5	Sophia	NULL

tbl01

ID	Name
12	Paul
16	Jessica
19	Shaun

tbl02

ID	Name
23	Bob
24	Ben
27	Tony

tbl

ID	Name	Level
1	Sarah	9
5	Sophia	NULL
12	Paul	NULL
16	Jessica	NULL
19	Shaun	NULL
23	Bob	NULL
24	Ben	NULL
27	Tony	NULL

图 235: optimistic-ddl-example-4

在 tbl01 同样增加一列 Level：

```
ALTER TABLE `tbl01` ADD COLUMN `Level` INT;
```

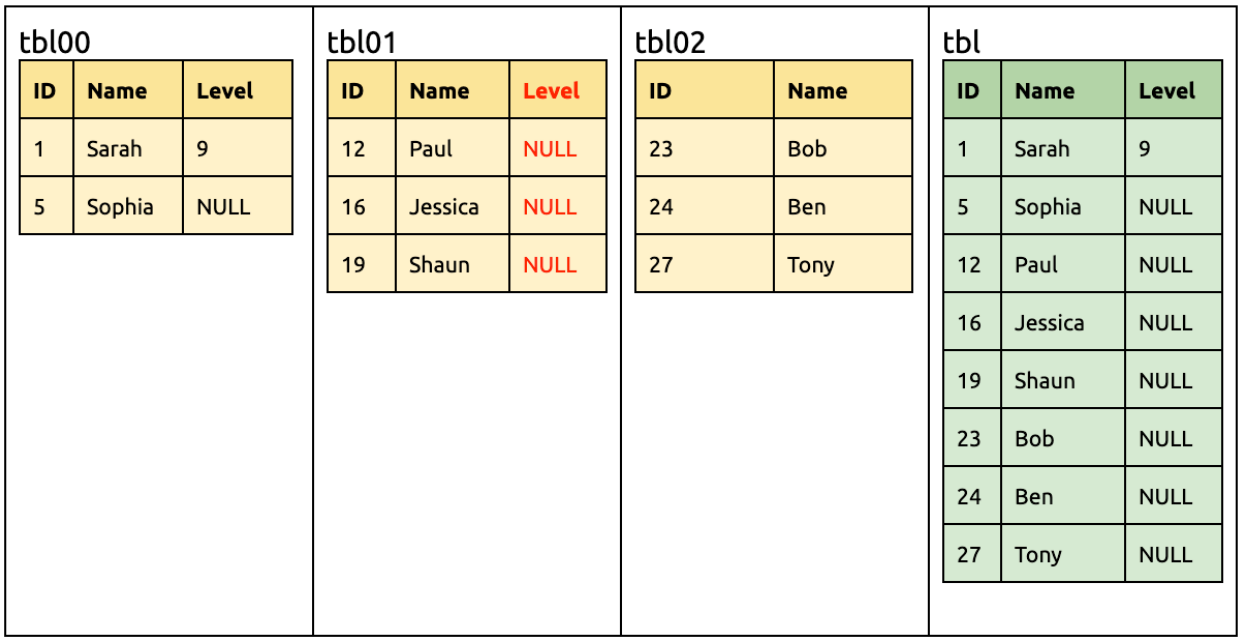


图 236: optimistic-ddl-example-5

此时下游已经有相同的 Level 列了，所以 DM-master 比较表结构之后不做任何操作。

在 tbl01 删除一列 Name：

```
ALTER TABLE `tbl01` DROP COLUMN `Name`;
```

tbl00

ID	Name	Level
1	Sarah	9
5	Sophia	NULL

tbl01

ID	Level
12	NULL
16	NULL
19	NULL

tbl02

ID	Name
23	Bob
24	Ben
27	Tony

tbl

ID	Name	Level
1	Sarah	9
5	Sophia	NULL
12	Paul	NULL
16	Jessica	NULL
19	Shaun	NULL
23	Bob	NULL
24	Ben	NULL
27	Tony	NULL

图 237: optimistic-ddl-example-6

此时下游仍需要接收来自 `tbl00` 和 `tbl02` 含 `Name` 的 DML 语句，因此不会立刻删除该列。

同样，各种 DML 仍可直接迁移到下游：

```
INSERT INTO `tbl01` (`ID`, `Level`) VALUES (15, 7);
UPDATE `tbl00` SET `Level` = 5 WHERE `ID` = 5;
```

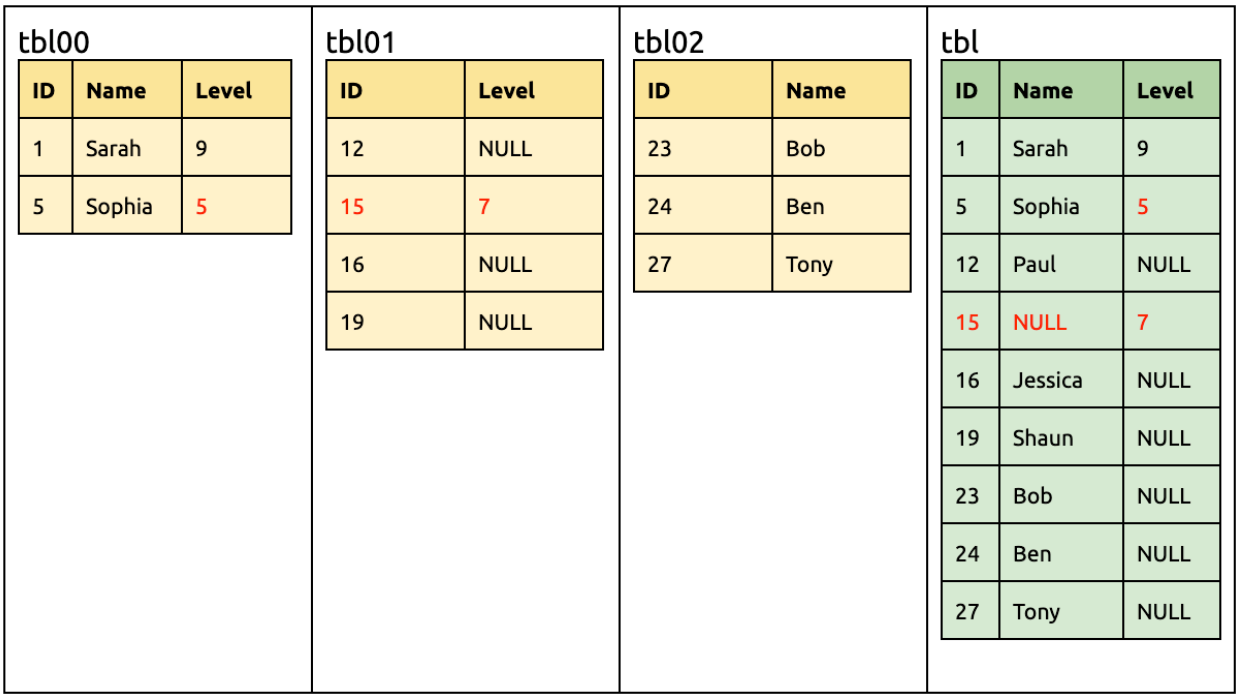


图 238: optimistic-ddl-example-7

在 `tbl02` 增加一列 `Level`：

```
ALTER TABLE `tbl02` ADD COLUMN `Level` INT;
```

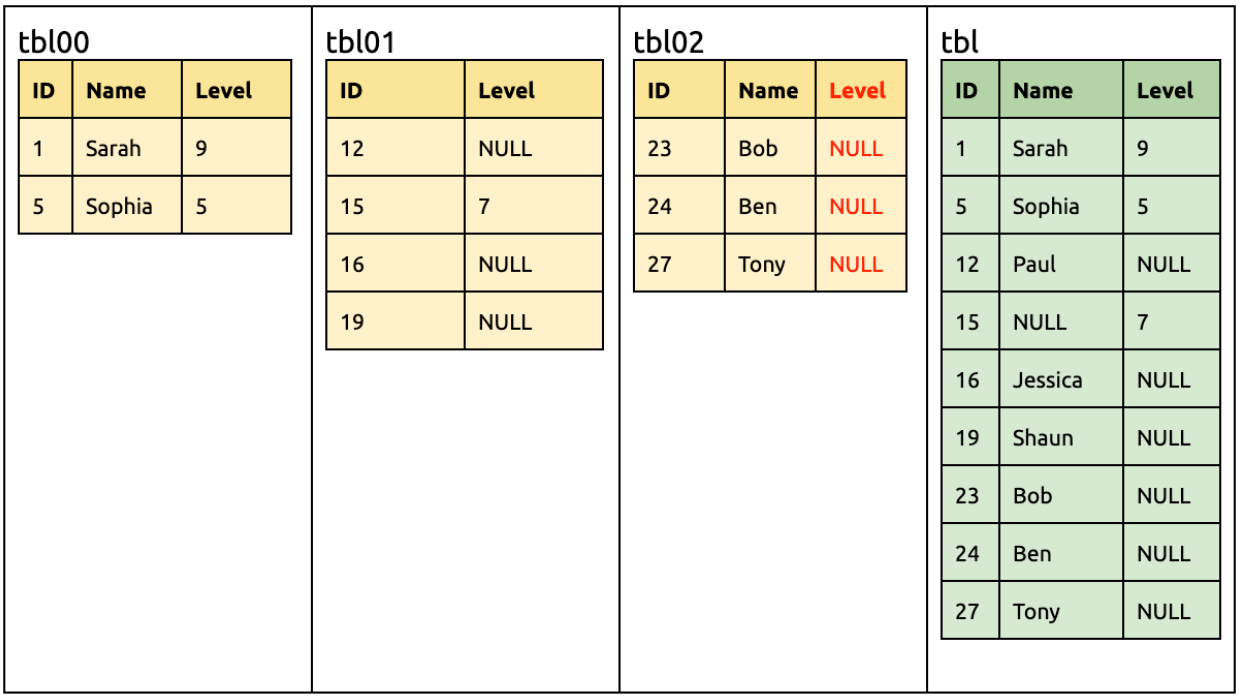


图 239: optimistic-ddl-example-8

此时所有分表都已有 `Level` 列。

在 `tbl00` 和 `tbl02` 各删除一列 `Name`：

```
ALTER TABLE `tbl00` DROP COLUMN `Name`;
ALTER TABLE `tbl02` DROP COLUMN `Name`;
```

tbl00

ID	Level
1	9
5	5

tbl01

ID	Level
12	NULL
15	7
16	NULL
19	NULL

tbl02

ID	Level
23	NULL
24	NULL
27	NULL

tbl

ID	Name	Level
1	Sarah	9
5	Sophia	5
12	Paul	NULL
15	NULL	7
16	Jessica	NULL
19	Shaun	NULL
23	Bob	NULL
24	Ben	NULL
27	Tony	NULL

图 240: optimistic-ddl-example-9

到此步 Name 列也从所有分表消失了，所以可以安全从下游移除：

```
ALTER TABLE `tbl` DROP COLUMN `Name`;
```

tbl00

ID	Level
1	9
5	5

tbl01

ID	Level
12	NULL
15	7
16	NULL
19	NULL

tbl02

ID	Level
23	NULL
24	NULL
27	NULL

tbl

ID	Level
1	9
5	5
12	NULL
15	7
16	NULL
19	NULL
23	NULL
24	NULL
27	NULL

图 241: optimistic-ddl-example-10

13.6.7.1.4 手动处理 Sharding DDL Lock

DM (Data Migration) 使用 sharding DDL lock 来确保分库分表的 DDL 操作可以正确执行。绝大多数情况下，该锁定机制可自动完成；但在部分异常情况发生时，需要使用 `shard-ddl-lock` 手动处理异常的 DDL lock。

> 注意：
>
> - 本文档只适用于悲观协调模式下 sharding DDL lock 的处理。
> - 本文档的命令在交互模式中进行，因此在以下命令示例中未添加转义字符。在命令行模式中，你需要添加转义字符，防止报错。
> - 不要轻易使用 `shard-ddl-lock unlock` 命令，除非完全明确当前场景下使用这些命令可能会造成的影响，并能接受这些影响。
> - 在手动处理异常的 DDL lock 前，请确保已经了解 DM 的分库分表合并迁移原理。

命令介绍

`shard-ddl-lock`

该命令用于查看 DDL lock 和主动请求 DM-master 解除指定的 DDL lock。命令仅在 DM v6.0 及其以后版本支持，之前版本可使用 `show-ddl-locks` 和 `unlock-ddl-lock` 命令。

```
shard-ddl-lock -h
```

```
maintain or show shard-ddl locks information

Usage:
  dmctl shard-ddl-lock [task] [flags]
  dmctl shard-ddl-lock [command]

Available Commands:
  unlock      Unlock un-resolved DDL locks forcely

Flags:
  -h, --help   help for shard-ddl-lock

Global Flags:
  -s, --source strings   MySQL Source ID.

Use "dmctl shard-ddl-lock [command] --help" for more information about a command.
```

参数解释

- `shard-ddl-lock [task] [flags]`:
  - 用于查询当前 DM-master 上存在的 DDL lock 信息
- `shard-ddl-lock [command]`
  - 用于主动请求 DM-master 解除指定的 DDL lock, command 只支持 `unlock`

命令示例

`shard-ddl-lock [task] [flags]`

使用 `shard-ddl-lock [task] [flags]` 命令，查询当前 DM-master 上存在的 DDL lock 信息。

例如：

```
shard-ddl-lock test
```

期望输出

```
{
    "result": true,                                     # 查询 lock 操作本身是否成功
    "msg": "",                                          # 查询 lock
        ↪ 操作失败时的原因或其它描述信息（如不存在任务 lock）
    "locks": [                                          # 当前存在的 lock 信息列表
        {
            "ID": "test-`shard_db`.`shard_table`",      # lock 的 ID 标识，当前由任务名与 DDL
                ↪ 对应的 schema/table 信息组成
            "task": "test",                             # lock 所属的任务名
```

```
            "mode": "pessimistic"                           # shard DDL 协调模式，可为悲观模式 "
                ↪ pessimistic" 或乐观模式 "optimistic"
            "owner": "mysql-replica-01",                    # lock 的 owner (
                ↪ 在悲观模式时为第一个遇到该 DDL 的 source ID)，在乐观模式时总为空
            "DDLs": [                                       # 在悲观模式时为 lock 对应的 DDL 列表
                ↪ ，在乐观模式时总为空
                "USE `shard_db`; ALTER TABLE `shard_db`.`shard_table` DROP COLUMN `c2`;"
            ],
            "synced": [                                     # 已经收到对应 MySQL 实例内所有分表
                ↪ DDL 的 source 列表
                "mysql-replica-01"
            ],
            "unsynced": [                                   # 尚未收到对应 MySQL 实例内所有分表
                ↪ DDL 的 source 列表
                "mysql-replica-02"
            ]
        }
    ]
}
```

shard-ddl-lock unlock

用于主动请求 DM-master 解除指定的 DDL lock，包括的操作：请求 owner 执行 DDL 操作，请求其他非 owner 的 DM-worker 跳过 DDL 操作，移除 DM-master 上的 lock 信息。

> **注意：**
>
> shard-ddl-lock unlock 当前仅对悲观协调模式 (pessimistic) 下产生的 lock 有效。

```
shard-ddl-lock unlock -h
```

```
Unlock un-resolved DDL locks forcely

Usage:
  dmctl shard-ddl-lock unlock <lock-id> [flags]

Flags:
  -a, --action string     accept skip/exec values which means whether to skip or execute ddls (
      ↪ default "skip")
  -d, --database string   database name of the table
  -f, --force-remove      force to remove DDL lock
  -h, --help              help for unlock
  -o, --owner string      source to replace the default owner
  -t, --table string      table name
```

```
Global Flags:
  -s, --source strings   MySQL Source ID.
```

shard-ddl-lock unlock 命令支持以下参数：

- -o, --owner：
  - flag 参数，string，可选
  - 不指定时，请求默认的 owner（shard-ddl-lock 返回结果中的 owner）执行 DDL 操作；指定时，请求该 MySQL source（替代默认的 owner）执行 DDL 操作
  - 除非原 owner 已经从集群中移除，否则不应该指定新的 owner
- -f, --force-remove：
  - flag 参数，boolean，可选
  - 不指定时，仅在 owner 执行 DDL 成功时移除 lock 信息；指定时，即使 owner 执行 DDL 失败也强制移除 lock 信息（此后将无法再次查询或操作该 lock）
- lock-id：
  - 非 flag 参数，string，必选
  - 指定需要执行 unlock 操作的 DDL lock ID（即 shard-ddl-lock 返回结果中的 ID）

以下是一个使用 shard-ddl-lock unlock 命令的示例：

```
shard-ddl-lock unlock test-`shard_db`.`shard_table`
```

```
{
    "result": true,                                     # unlock lock 操作是否成功
    "msg": "",                                          # unlock lock 操作失败时的原因
}
```

支持场景

目前，使用 shard-ddl-lock unlock 命令仅支持处理以下两种 sharding DDL lock 异常情况。

场景一：部分 MySQL source 被移除

Lock 异常原因

在 DM-master 尝试自动 unlock sharding DDL lock 之前，需要等待所有 MySQL source 的 sharding DDL events 全部到达（详见分库分表合并迁移原理）。如果 sharding DDL 已经在迁移过程中，同时有部分 MySQL source 被移除，且不再计划重新加载它们（按业务需求移除了这部分 MySQL source），则会由于永远无法等齐所有的 DDL 而造成 lock 无法自动 unlock。

手动处理示例

假设上游有 MySQL-1（mysql-replica-01）和 MySQL-2（mysql-replica-02）两个实例，其中 MySQL-1 中有 shard_db_1.shard_table_1 和 shard_db_1.shard_table_2 两个表，MySQL-2 中有 shard_db_2.shard_table_1 和 shard_db_2.shard_table_2 两个表。现在需要将这 4 个表合并后迁移到下游 TiDB 的 shard_db.shard_table 表中。

初始表结构如下：

```
SHOW CREATE TABLE shard_db_1.shard_table_1;
```

```
+---------------+----------------------------------------+
| Table         | Create Table                           |
+---------------+----------------------------------------+
| shard_table_1 | CREATE TABLE `shard_table_1` (
  `c1` int(11) NOT NULL,
  PRIMARY KEY (`c1`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 |
+---------------+----------------------------------------+
```

上游分表将执行以下 DDL 语句变更表结构：

```
ALTER TABLE shard_db_*.shard_table_* ADD COLUMN c2 INT;
```

MySQL 及 DM 操作与处理流程如下：

1. mysql-replica-01 对应的两个分表执行了对应的 DDL 操作进行表结构变更。

   ```
   ALTER TABLE shard_db_1.shard_table_1 ADD COLUMN c2 INT;
   ```

   ```
   ALTER TABLE shard_db_1.shard_table_2 ADD COLUMN c2 INT;
   ```

2. DM-worker 接受到 mysql-replica-01 两个分表的 DDL 之后，将对应的 DDL 信息发送给 DM-master，DM-master 创建相应的 DDL lock。

3. 使用 shard-ddl-lock 查看当前的 DDL lock 信息。

   ```
   shard-ddl-lock test
   ```

   ```
   {
       "result": true,
       "msg": "",
       "locks": [
           {
               "ID": "test-`shard_db`.`shard_table`",
               "task": "test",
               "mode": "pessimistic"
               "owner": "mysql-replica-01",
               "DDLs": [
                   "USE `shard_db`; ALTER TABLE `shard_db`.`shard_table` ADD COLUMN `c2` int
                       ↪ (11);"
               ],
               "synced": [
                   "mysql-replica-01"
               ],
   ```

```
            "unsynced": [
                "mysql-replica-02"
            ]
        }
    ]
}
```

4. 由于业务需要，`mysql-replica-02` 对应的数据不再需要迁移到下游 TiDB，对 `mysql-replica-02` 执行了移除操作。

5. DM-master 上 ID 为 `test-`shard_db`.`shard_table`` 的 lock 无法等到 `mysql-replica-02` 的 DDL 操作信息。`shard-ddl-lock` 返回的 `unsynced` 中一直包含 `mysql-replica-02` 的信息。

6. 使用 `shard-ddl-lock unlock` 来请求 DM-master 主动 unlock 该 DDL lock。

    - 如果 DDL lock 的 owner 也已经被移除，可以使用 `--owner` 参数指定其他 MySQL source 作为新 owner 来执行 DDL。
    - 当存在任意 MySQL source 报错时，`result` 将为 `false`，此时请仔细检查各 MySQL source 的错误是否是预期可接受的。

    ```
    shard-ddl-lock unlock test-`shard_db`.`shard_table`
    ```

    ```
    {
        "result": true,
        "msg": ""
    }
    ```

7. 使用 `shard-ddl-lock` 确认 DDL lock 是否被成功 unlock。

```
shard-ddl-lock test
```

```
{
    "result": true,
    "msg": "no DDL lock exists",
    "locks": [
    ]
}
```

8. 查看下游 TiDB 中的表结构是否变更成功。

```
SHOW CREATE TABLE shard_db.shard_table;
```

```
+-------------+-----------------------------------------------------+
| Table       | Create Table                                        |
+-------------+-----------------------------------------------------+
| shard_table | CREATE TABLE `shard_table` (
  `c1` int(11) NOT NULL,
```

```
  `c2` int(11) DEFAULT NULL,
  PRIMARY KEY (`c1`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 COLLATE=latin1_bin |
+-------------+-----------------------------------------------------+
```

9. 使用 query-status 确认迁移任务是否正常。

手动处理后的影响

使用 shard-ddl-lock unlock 手动执行 unlock 操作后，由于该任务的配置信息中仍然包含了已下线的 MySQL source，如果不进行处理，则当下次 sharding DDL 到达时，仍会出现 lock 无法自动完成迁移的情况。

因此，在手动解锁 DDL lock 后，需要再执行以下操作：

1. 使用 stop-task 停止运行中的任务。
2. 更新任务配置文件，将已下线 MySQL source 对应的信息从配置文件中移除。
3. 使用 start-task 及新任务配置文件重新启动任务。

> 注意：
>
> 在 shard-ddl-lock unlock 之后，如果已下线的 MySQL source 重新加载并尝试对其中的分表进行数据迁移，则会由于数据与下游的表结构不匹配而发生错误。

场景二：unlock 过程中部分 DM-worker 异常停止或网络中断

Lock 异常原因

在 DM-master 收到所有 DM-worker 的 DDL 信息后，执行自动 unlock DDL lock 的操作主要包括以下步骤：

1. 请求 lock owner 执行 DDL 操作，并更新对应分表的 checkpoint。
2. 在 owner 执行 DDL 操作成功后，移除 DM-master 上保存的 DDL lock 信息。
3. 在 owner 执行 DDL 操作成功后，请求其他所有非 owner 跳过 DDL 操作并更新对应分表的 checkpoint。
4. DM-master 在所有 owner/非 owner 操作成功后，移除对应的 DDL lock 信息。

上述 unlock DDL lock 的操作不是原子的。如果非 owner 跳过 DDL 操作成功后，所在的 DM-worker 异常停止或与下游 TiDB 发生网络异常，造成无法成功更新 checkpoint。

当非 owner 对应的 MySQL source 恢复数据迁移时，会尝试请求 DM-master 重新协调异常发生前已经协调过的 DDL、且永远无法等到其他 MySQL source 的对应 DDL，造成该 DDL 操作对应 lock 的自动 unlock。

手动处理示例

仍然假设是部分 MySQL source 被移除示例中的上下游表结构及合表迁移需求。

当在 DM-master 自动执行 unlock 操作的过程中，owner (mysql-replica-01) 成功执行了 DDL 操作且开始继续进行后续迁移，但在请求非 owner (mysql-replica-02) 跳过 DDL 操作的过程中，由于对应的 DM-worker 发生了重启在跳过 DDL 后未能更新 checkpoint。

mysql-replica-02 对应的数据迁移子任务恢复后，将在 DM-master 上创建一个新的 lock，但其他 MySQL source 此时已经执行或跳过 DDL 操作并在进行后续迁移。

处理流程如下：

1. 使用 shard-ddl-lock 确认 DM-master 上存在该 DDL 操作对应的 lock。

   应该仅有 mysql-replica-02 处于 synced 状态：

```
shard-ddl-lock
```

```
{
    "result": true,
    "msg": "",
    "locks": [
        {
            "ID": "test-`shard_db`.`shard_table`",
            "task": "test",
            "mode": "pessimistic"
            "owner": "mysql-replica-02",
            "DDLs": [
                "USE `shard_db`; ALTER TABLE `shard_db`.`shard_table` ADD COLUMN `c2` int
                    ↪ (11);"
            ],
            "synced": [
                "mysql-replica-02"
            ],
            "unsynced": [
                "mysql-replica-01"
            ]
        }
    ]
}
```

2. 使用 shard-ddl-lock unlock 请求 DM-master unlock 该 lock。

   - Lock 过程中会尝试再次向下游执行该 DDL 操作（重启前的原 owner 已向下游执行过该 DDL 操作），需要确保该 DDL 操作可被多次执行。

```
shard-ddl-lock unlock test-`shard_db`.`shard_table`
```

```
{
    "result": true,
    "msg": "",
}
```

3. 使用 shard-ddl-lock 确认 DDL lock 是否被成功 unlock。

4. 使用 query-status 确认迁移任务是否正常。

手动处理后的影响

手动 unlock sharding DDL lock 后，后续的 sharding DDL 将可以自动正常迁移。

13.6.7.2 迁移使用 GH-ost/PT-osc 的源数据库

在生产业务中执行 DDL 时，产生的锁表操作会一定程度阻塞数据库的读取或者写入。为了把对读写的影响降到最低，用户往往会选择 online DDL 工具执行 DDL。常见的 Online DDL 工具有 gh-ost 和 pt-osc。

在使用 DM 完成 MySQL 到 TiDB 的数据迁移时，可以开启online-ddl配置，实现 DM 工具与 gh-ost 或 pt-osc 的协同。关于如何开启 online-ddl配置及开启该配置后的工作流程，请参考上游使用 pt-osc/gh-ost 工具的持续同步场景。本文仅介绍 DM 与 online DDL 工具协作的细节。

13.6.7.2.1 DM 与 online DDL 工具协作细节

DM 与 online DDL 工具 gh-ost 和 pt-osc 在实现 online-schema-change 过程中的协作细节如下。

online-schema-change: gh-ost

gh-ost 在实现 online-schema-change 的过程会产生 3 种 table：

- gho：用于应用 DDL，待 gho 表中数据迁移到与 origin table 一致后，通过 rename 的方式替换 origin table。
- ghc：用于存放 online-schema-change 相关的信息。
- del：对 origin table 执行 rename 操作而生成。

DM 在迁移过程中会把上述 table 分成 3 类：

- ghostTable : _*_gho
- trashTable : _*_ghc、_*_del
- realTable : 执行 online-ddl 的 origin table

gh-ost 涉及的主要 SQL 以及 DM 的处理：

1. 创建 _ghc 表：

```
Create /* gh-ost */ table `test`.`_test4_ghc` (
                        id bigint auto_increment,
                        last_update timestamp not null DEFAULT CURRENT_TIMESTAMP ON UPDATE
                            ↪ CURRENT_TIMESTAMP,
                        hint varchar(64) charset ascii not null,
                        value varchar(4096) charset ascii not null,
                        primary key(id),
                        unique key hint_uidx(hint)
                ) auto_increment=256 ;
```

DM：不执行 _test4_ghc 的创建操作。

2. 创建 _gho 表：

```
Create /* gh-ost */ table `test`.`_test4_gho` like `test`.`test4` ;
```

DM：不执行 _test4_gho 的创建操作，根据 ghost_schema、ghost_table 以及 dm_worker 的 server_id，删除下游 dm_meta.{task_name}_onlineddl 的记录，清理内存中的相关信息。

```
DELETE FROM dm_meta.{task_name}_onlineddl WHERE id = {server_id} and ghost_schema = {
    ↪ ghost_schema} and ghost_table = {ghost_table};
```

3. 在 _gho 表应用需要执行的 DDL：

```
Alter /* gh-ost */ table `test`.`_test4_gho` add column cl1 varchar(20) not null ;
```

DM：不执行 _test4_gho 的 DDL 操作，而是把该 DDL 记录到 dm_meta.{task_name}_onlineddl 以及内存中。

```
REPLACE INTO dm_meta.{task_name}_onlineddl (id, ghost_schema , ghost_table , ddls) VALUES
    ↪ (......);
```

4. 往 _ghc 表写入数据，以及往 _gho 表同步 origin table 的数据：

```
INSERT /* gh-ost */ INTO `test`.`_test4_ghc` VALUES (......);

INSERT /* gh-ost `test`.`test4` */ ignore INTO `test`.`_test4_gho` (`id`, `date`, `
    ↪ account_id`, `conversion_price`, `ocpc_matched_conversions`, `ad_cost`, `cl2`)
  (SELECT `id`, `date`, `account_id`, `conversion_price`, `ocpc_matched_conversions`, `
      ↪ ad_cost`, `cl2` FROM `test`.`test4` FORCE INDEX (`PRIMARY`)
    WHERE (((`id` > _binary'1') OR ((`id` = _binary'1'))) AND ((`id` < _binary'2') OR ((`id`
        ↪  = _binary'2')))) lock IN share mode
  )   ;
```

DM：只要不是 realtable 的 DML 全部不执行。

5. 数据同步完成后 origin table 与 _gho 一起改名，完成 online DDL 操作：

```
Rename /* gh-ost */ table `test`.`test4` to `test`.`_test4_del`, `test`.`_test4_gho` to `
    ↪ test`.`test4`;
```

DM 执行以下两个操作:

- 把 rename 语句拆分成两个 SQL：

```
rename test.test4 to test._test4_del;
rename test._test4_gho to test.test4;
```

- 不执行 rename to _test4_del。当要执行 rename ghost_table to origin table 的时候，并不执行 rename 语句，而是把步骤 3 记录在内存中的 DDL 读取出来，然后把 ghost_table、ghost_schema 替换为 origin_table 以及对应的 schema，再执行替换后的 DDL。

```
alter table test._test4_gho add column cl1 varchar(20) not null;
--替换为
alter table test.test4 add column cl1 varchar(20) not null;
```

> **注意：**
>
> 具体 gh-ost 的 SQL 会根据工具执行时所带的参数而变化。本文只列出主要的 SQL，具体可以参考 gh-ost 官方文档。

online-schema-change: pt

pt-osc 在实现 online-schema-change 的过程会产生 2 种 table：

- new：用于应用 DDL，待表中数据同步到与 origin table 一致后，再通过 rename 的方式替换 origin table。
- old：对 origin table 执行 rename 操作后生成。
- 3 种 trigger：pt_osc_*_ins、pt_osc_*_upd、pt_osc_*_del，用于在 pt_osc 过程中，同步 origin table 新产生的数据到 new。

DM 在迁移过程中会把上述 table 分成 3 类：

- ghostTable : _*_new
- trashTable : _*_old
- realTable : 执行的 online-ddl 的 origin table

pt-osc 主要涉及的 SQL 以及 DM 的处理：

1. 创建 _new 表：

```
CREATE TABLE `test`.`_test4_new` (id int(11) NOT NULL AUTO_INCREMENT,
date date DEFAULT NULL, account_id bigint(20) DEFAULT NULL, conversion_price decimal(20,3)
    ↪ DEFAULT NULL,  ocpc_matched_conversions bigint(20) DEFAULT NULL, ad_cost decimal
    ↪ (20,3) DEFAULT NULL,cl2 varchar(20) COLLATE utf8mb4_bin NOT NULL,cl1 varchar(20)
    ↪ COLLATE utf8mb4_bin NOT NULL,PRIMARY KEY (id) ) ENGINE=InnoDB AUTO_INCREMENT=3
    ↪ DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin ;
```

   DM: 不执行 _test4_new 的创建操作。根据 ghost_schema、ghost_table 以及 dm_worker 的 server_id，删除下游 dm_meta.{task_name}_onlineddl 的记录，清理内存中的相关信息。

```
DELETE FROM dm_meta.{task_name}_onlineddl WHERE id = {server_id} and ghost_schema = {
    ↪ ghost_schema} and ghost_table = {ghost_table};
```

2. 在 _new 表上执行 DDL：

```
ALTER TABLE `test`.`_test4_new` add column c3 int;
```

DM: 不执行 _test4_new 的 DDL 操作，而是把该 DDL 记录到 dm_meta.{task_name}_onlineddl 以及内存中。

```
REPLACE INTO dm_meta.{task_name}_onlineddl (id, ghost_schema , ghost_table , ddls) VALUES
    ↪ (......);
```

3. 创建用于同步数据的 3 个 Trigger：

```
CREATE TRIGGER `pt_osc_test_test4_del` AFTER DELETE ON `test`.`test4` ...... ;
CREATE TRIGGER `pt_osc_test_test4_upd` AFTER UPDATE ON `test`.`test4` ...... ;
CREATE TRIGGER `pt_osc_test_test4_ins` AFTER INSERT ON `test`.`test4` ...... ;
```

DM: 不执行 TiDB 不支持的相关 Trigger 操作。

4. 往 _new 表同步 origin table 的数据：

```
INSERT LOW_PRIORITY IGNORE INTO `test`.`_test4_new` (`id`, `date`, `account_id`, `
    ↪ conversion_price`, `ocpc_matched_conversions`, `ad_cost`, `cl2`, `cl1`) SELECT `id`,
    ↪  `date`, `account_id`, `conversion_price`, `ocpc_matched_conversions`, `ad_cost`, `
    ↪ cl2`, `cl1` FROM `test`.`test4` LOCK IN SHARE MODE /*pt-online-schema-change 3227
    ↪ copy table*/
```

DM: 只要不是 realTable 的 DML 全部不执行。

5. 数据同步完成后 origin table 与 _new 一起改名，完成 online DDL 操作：

```
RENAME TABLE `test`.`test4` TO `test`.`_test4_old`, `test`.`_test4_new` TO `test`.`test4`
```

DM 执行以下两个操作:

- 把 rename 语句拆分成两个 SQL。

  ```
  sql rename test.test4 to test._test4_old; rename test._test4_new to test.test4;
  ```

- 不执行 rename to _test4_old。当要执行 rename ghost_table to origin table 的时候，并不执行 rename，而是把步骤 2 记录在内存中的 DDL 读取出来，然后把 ghost_table、ghost_schema 替换为 origin_table 以及对应的 schema，再执行替换后的 DDL。

  ```
  sql  ALTER TABLE `test`.`_test4_new` add column c3 int; --替换为 ALTER TABLE `test`.`
  ↪ test4` add column c3 int;
  ```

6. 删除 _old 表以及 online DDL 的 3 个 Trigger：

```
DROP TABLE IF EXISTS `test`.`_test4_old`;
DROP TRIGGER IF EXISTS `pt_osc_test_test4_del` AFTER DELETE ON `test`.`test4` ...... ;
DROP TRIGGER IF EXISTS `pt_osc_test_test4_upd` AFTER UPDATE ON `test`.`test4` ...... ;
DROP TRIGGER IF EXISTS `pt_osc_test_test4_ins` AFTER INSERT ON `test`.`test4` ...... ;
```

DM: 不执行 _test4_old 以及 Trigger 的删除操作。

> **注意：**
>
> 具体 pt-osc 的 SQL 会根据工具执行时所带的参数而变化。本文只列出主要的 SQL，具体可以参考 pt-osc 官方文档。

13.6.7.2.2 其他 Online Schema Change 工具

在某些场景下，你可能需要变更 online schema change 工具的默认行为，自定义 `ghost table` 和 `trash table` 的名称；或者期望使用 gh-ost 和 pt-osc 之外的工具（原理和变更流程仍然保持一致）。此时则需要自行编写正则表达式以匹配`ghost table` 和 `trash table`。

自 v2.0.7 起，DM 实验性支持修改过的 online schema change 工具。在 DM 任务配置中设置 `online-ddl=true` 后，配合 `shadow-table-rules` 和 `trash-table-rules` 即可支持通过正则表达式来匹配修改过的临时表。

假设自定义 pt-osc 的 `ghost table` 规则为 `_{origin_table}_pcnew`，`trash table` 规则为 `_{origin_table}_pcold`，那么自定义规则需配置如下：

```
online-ddl: true
shadow-table-rules: ["^_(.+)_(?:pcnew)$"]
trash-table-rules: ["^_(.+)_(?:pcold)$"]
```

13.6.7.3 下游存在更多列的迁移场景

本文档介绍数据同步时，下游存在更多列的迁移场景需要的注意事项。具体迁移操作可参考已有数据迁移场景：

- 从小数据量 MySQL 迁移数据到 TiDB
- 从大数据量 MySQL 迁移数据到 TiDB
- 从小数据量分库分表 MySQL 合并迁移数据到 TiDB
- 从大数据量分库分表 MySQL 合并迁移数据到 TiDB

13.6.7.3.1 使用 DM 迁移至存在更多列的下游

DM 同步上游的 binlog 时，会尝试使用下游当前的表结构来解析 binlog 并生成相应的 DML 语句。如果上游的 binlog 里数据表的列数与下游表结构的列数不一致，则会产生如下错误：

```
"errors": [
    {
        "ErrCode": 36027,
        "ErrClass": "sync-unit",
        "ErrScope": "internal",
        "ErrLevel": "high",
        "Message": "startLocation: [position: (mysql-bin.000001, 2022), gtid-set:09bec856-ba95-11
            ↪ ea-850a-58f2b4af5188:1-9 ], endLocation: [position: (mysql-bin.000001, 2022),
            ↪ gtid-set: 09bec856-ba95-11ea-850a-58f2b4af5188:1-9]: gen insert sqls failed,
```

```
            ↪ schema: log, table: messages: Column count doesn't match value count: 3 (columns)
            ↪  vs 2 (values)",
        "RawCause": "",
        "Workaround": ""
    }
]
```

例如上游表结构为：

```
#### 上游表结构
CREATE TABLE `messages` (
  `id` int(11) NOT NULL,
  PRIMARY KEY (`id`)
)
```

例如下游表结构为：

```
#### 下游表结构
CREATE TABLE `messages` (
  `id` int(11) NOT NULL,
  `message` varchar(255) DEFAULT NULL, # 下游比上游多出的列。
  PRIMARY KEY (`id`)
)
```

当 DM 尝试使用下游表结构解析上游产生的 binlog event 时，DM 会报出上述 `Column count doesn't match` 错误。

此时，你可以使用 `binlog-schema` 命令来为数据源中需要迁移的表指定表结构，表结构需要对应 DM 将要开始同步的 binlog event 的数据。如果你在进行分表合并的数据迁移，那么需要为每个分表按照如下步骤在 DM 中设置用于解析 binlog event 的表结构。具体操作为：

1. 在 DM 中，新建一个 `.sql` 文件，并将上游表结构对应的 `CREATE TABLE` 语句添加到该文件。例如，将以下表结构保存到 `log.messages.sql` 中。如果是 6.0 及以上版本，可以直接通过 `--from-source/--from-target` 更新，无需创建 SQL 文件。可参考管理迁移表的表结构。

    ```
    # 上游表结构
    CREATE TABLE `messages` (
    `id` int(11) NOT NULL,
    PRIMARY KEY (`id`)
    )
    ```

2. 使用 `binlog-schema` 命令为数据源中需要迁移的表设置表结构（此时数据迁移任务应该由于上述 `Column count doesn't match` 错误而处于 Paused 状态）。

    ```
    tiup dmctl --master-addr ${advertise-addr} binlog-schema update -s ${source-id} ${task-name}
        ↪  ${database-name} ${table-name} ${schema-file}
    ```

    该命令中的参数描述如下：

参数	描述
--master-addr	指定 dmctl 要连接的集群的任意 DM-master 节点的 ${advertise-addr}。${advertise-addr} 表示 DM-master 向外界宣告的地址。
binlog	手动更新 schema 信息
-s	指定 source。${source-id} 表示 MySQL 数据源 ID。

参数	描述
${task-name}	指定 task。表示数据同步任务配置文件 task.yaml 中定义的同步任务名称。
${database}	指定 database。表示上游数据库名。
${table}	指定 table。表示上游数据表名。
${schema-file}	指定表的 schema 文件。表示将被设置的表结构文件。

例如：

```
tiup dmctl --master-addr 172.16.10.71:8261 binlog-schema update -s mysql-01 task-test -d log
    ↪  -t message log.message.sql
```

3. 使用 resume-task 命令恢复处于 Paused 状态的同步任务。

```
tiup dmctl --master-addr ${advertise-addr} resume-task ${task-name}
```

4. 使用 query-status 命令确认数据迁移任务是否运行正常。

```
tiup dmctl --master-addr ${advertise-addr} query-status resume-task ${task-name}
```

13.6.7.4 DM 增量数据校验

本文介绍了如何使用 DM 增量数据校验功能、DM 增量数据校验的原理以及相关的使用限制。

13.6.7.4.1 使用场景

在将增量数据从上游迁移到下游数据库的过程中，数据的流转有小概率导致错误或者丢失的情况。对于需要依赖于强数据一致的场景，如信贷、证券等业务，你可以在数据迁移完成之后对数据进行全量校验，确保数据的一致性。然而，在某些增量复制的业务场景下，上游和下游的写入是持续的、不会中断的，因为上下游的数据在不断变化，导致用户难以对表里面的全部数据进行一致性校验（例如使用sync-diff-inspector）。

在增量数据复制的场景下，你可以使用 DM 的增量校验功能，在数据持续写入下游的增量复制过程中确保迁移数据的完整性、一致性。

13.6.7.4.2 开启增量数据校验

你可以使用下面任意方法开启增量数据校验：

- 在任务配置中开启
- 通过 dmctl 开启

方法 1：在任务配置中开启

你可以在任务配置文件中加入以下内容来开启增量数据校验：

```
#### 给需要开启增量校验功能的上游数据库添加增量校验配置
mysql-instances:
  - source-id: "mysql1"
    block-allow-list: "bw-rule-1"
    validator-config-name: "global"
validators:
  global:
    mode: full # 也可以是 fast，默认是 none，即不开启校验
    worker-count: 4 # 后台校验的 validation worker 数量，默认是 4 个
    row-error-delay: 30m # 某一行多久没有验证通过会被标记为 error row，默认是 30m，即 30 分钟
```

示例中各配置项的含义如下：

- mode：校验模式，可以是 none、full、fast。

  - `none`：默认值，即不开启校验。
  - `full`：将变动行和下游数据库中获取的行数据进行每列对比。
  - `fast`：只判断这一行在下游数据库是否存在。

- `worker-count`：增量校验功能使用的 worker 数量（每个 worker 都是一个 goroutine）。
- `row-error-delay`：某一行多久没有验证通过会被标记为 error row，默认是 30 分钟。

完整配置请查阅DM 任务完整配置文件。

方法 2：通过 dmctl 开启

你可以使用 `dmctl validation start` 命令来开启增量数据校验：

```
Usage:
  dmctl validation start [--all-task] [task-name] [flags]

Flags:
      --all-task            whether applied to all tasks
  -h, --help                help for start
      --mode string         specify the mode of validation: full (default), fast; this flag will
          ↪ be ignored if the validation task has been ever enabled but currently paused (
          ↪ default "full")
      --start-time string   specify the start time of binlog for validation, e.g. '2021-10-21
          ↪ 00:01:00' or 2021-10-21T00:01:00
```

- `--mode`：指定开启的模式，可以是 fast 或者 full。
- `--start-time`：指定 validator 开启校验的位置，格式是：2021-10-21 00:01:00 或者 2021-10-21T00:01:00。
- `task-name`：需要开启增量数据校验的任务名，你也可以用 `--all-task` 来为当前所有任务开启增量数据校验。

示例：

```
dmctl --master-addr=127.0.0.1:8261 validation start --start-time 2021-10-21T00:01:00 --mode full
    ↪ my_dm_task
```

13.6.7.4.3 使用增量数据校验

在使用增量数据校验时，通过 dmctl 工具，你可以查询到增量校验当前的校验状态，也可以对校验出的错误行（error row）进行及时处理。所谓的错误行，就是在增量校验过程中，被检查出上下游数据不一致的行。

查看增量校验的状态

你可以使用两种方式查看增量校验的状态。

方式 1：用 `dmctl query-status <task-name>` 命令查看任务状态，如果开启了增量校验，校验结果会显示在每个 subtask 的 validation 字段里面。示例输出：

```
"subTaskStatus": [
    {
```

```
        "name": "test",
        "stage": "Running",
        "unit": "Sync",
        "result": null,
        "unresolvedDDLLockID": "",
        "sync": {
            ...
        },
        "validation": {
            "task": "test", // 任务名
            "source": "mysql-01", // source id
            "mode": "full", // 校验模式
            "stage": "Running", // 当前状态，Running 或者 Stopped
            "validatorBinlog": "(mysql-bin.000001, 5989)", // 校验到的 binlog 位置
            "validatorBinlogGtid": "1642618e-cf65-11ec-9e3d-0242ac110002:1-30", // 同上，用 GTID
                ↪ 表示
            "result": null, // 当增量校验异常时，显示异常信息
            "processedRowsStatus": "insert/update/delete: 0/0/0", // 已经处理的 binlog
                ↪ 数据行的统计信息
            "pendingRowsStatus": "insert/update/delete: 0/0/0", // 还未校验或者校验失败，
                ↪ 但还没标记为`错误行`的数据行统计信息
            "errorRowsStatus": "new/ignored/resolved: 0/0/0" // `错误行`统计信息，
                ↪ 三种状态的错误会在下文讲解
        }
    }
]
```

方式 2：使用 `dmctl validation status <taskname>` 来查询增量校验的状态：

```
dmctl validation status [--table-stage stage] <task-name> [flags]
Flags:
  -h, --help                help for status
      --table-stage string   filter validation tables by stage: running/stopped
```

在上述命令中，你可以设置 `--table-stage` 来过滤正在校验或者已经停止校验的表。示例输出：

```
{
    "result": true,
    "msg": "",
    "validators": [
        {
            "task": "test",
            "source": "mysql-01",
            "mode": "full",
            "stage": "Running",
            "validatorBinlog": "(mysql-bin.000001, 6571)",
```

```
            "validatorBinlogGtid": "",
            "result": null,
            "processedRowsStatus": "insert/update/delete: 2/0/0",
            "pendingRowsStatus": "insert/update/delete: 0/0/0",
            "errorRowsStatus": "new/ignored/resolved: 0/0/0"
        }
    ],
    "tableStatuses": [
        {
            "source": "mysql-01", // source id
            "srcTable": "`db`.`test1`", // 源表名
            "dstTable": "`db`.`test1`", // 目标表名
            "stage": "Running", // 校验状态
            "message": "" // 具体错误信息显示
        }
    ]
}
```

如果你想要查询错误行的详细信息，比如错误原因、错误时间等，可以使用 `dmctl validation show-error` 命令：

```
Usage:
  dmctl validation show-error [--error error-state] <task-name> [flags]

Flags:
      --error string   filtering type of error: all, ignored, or unprocessed (default "
          ↪ unprocessed")
  -h, --help           help for show-error
```

示例输出：

```
{
    "result": true,
    "msg": "",
    "error": [
        {
            "id": "1", // 错误行标识符，在后续的处理错误行中用到
            "source": "mysql-replica-01", // source id
            "srcTable": "`validator_basic`.`test`", // 错误行源表
            "srcData": "[0, 0]", // 错误行具体数据
            "dstTable": "`validator_basic`.`test`", // 错误行目标表
            "dstData": "[]", // 错误行在下游的数据
            "errorType": "Expected rows not exist", // 错误原因
            "status": "NewErr", // 错误状态
            "time": "2022-07-04 13:33:02", // 错误行发现时间
            "message": "" // 额外信息
```

```
        }
    ]
}
```

处理增量校验错误行

当增量数据校验发现错误行后，你需要手动处理这些错误行。

在增量校验出现错误行时，增量校验不会停下，而是会把这些错误行记录下来，让用户自己去发现处理。错误行没有被处理时，默认状态是 unprocessed。如果你在下游手动矫正了该错误行的错误，增量校验也不会去自动获取矫正后的信息，仍会将该错误行记录在 error 中。

如果你不想在 validation status 中再看到这个错误行、或者你需要给已经解决的错误行打上标记，你可以使用 `validation show-error` 找到错误行的 id，然后使用错误处理命令来对这些错误行进行处理或者标记。

dmctl 提供了三种错误处理命令：

- `clear-error`：清理掉错误行，`show-error`命令将不再展示该`error row`。

```
Usage:
  dmctl validation clear-error <task-name> <error-id|--all> [flags]

Flags:
      --all    all errors
  -h, --help   help for clear-error
```

- `ignore-error`：忽略该错误行，将这个错误行标记为 ignored。

```
Usage:
  dmctl validation ignore-error <task-name> <error-id|--all> [flags]

Flags:
      --all    all errors
  -h, --help   help for ignore-error
```

- `resolve-error`：已手动解决该错误行，将这个错误行标记为 resolved。

```
Usage:
  dmctl validation resolve-error <task-name> <error-id|--all> [flags]

Flags:
      --all    all errors
  -h, --help   help for resolve-error
```

13.6.7.4.4 停止增量数据校验

如果你需要停止增量数据校验，可以使用 `validation stop` 命令：

```
Usage:
  dmctl validation stop [--all-task] [task-name] [flags]

Flags:
      --all-task   whether applied to all tasks
  -h, --help       help for stop
```

用法可参考`dmctl validation start`命令。

13.6.7.4.5 原理

DM 增量校验（validator）的简要架构如下所示：

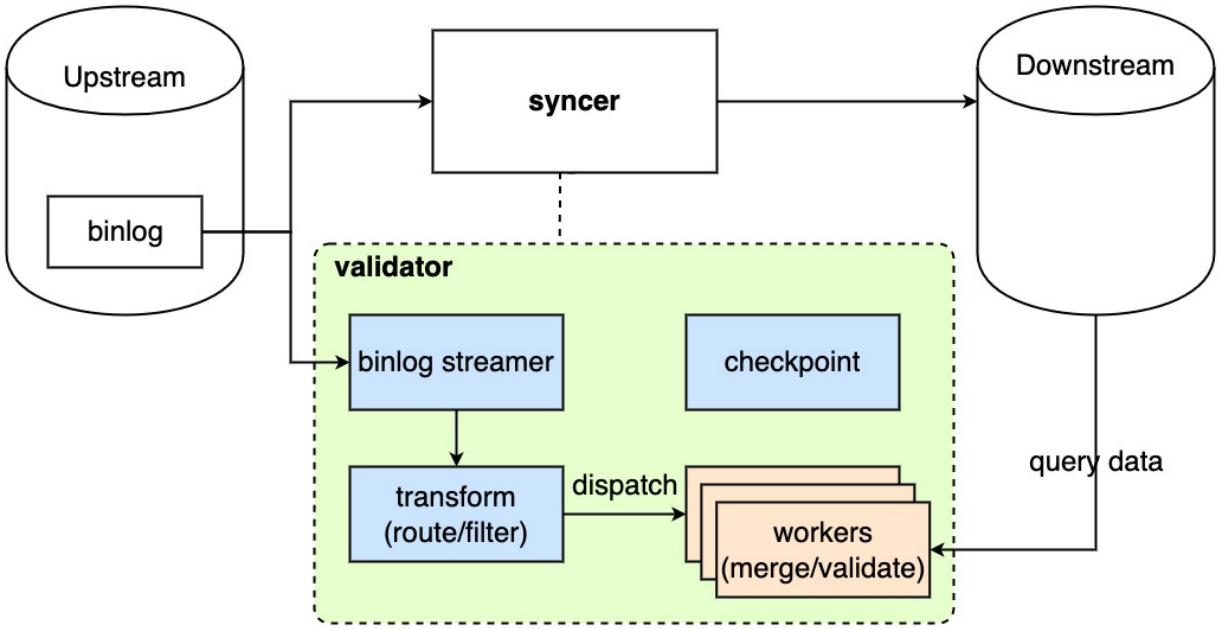


图 242: validator summary

增量数据校验的工作生命周期如下所示：

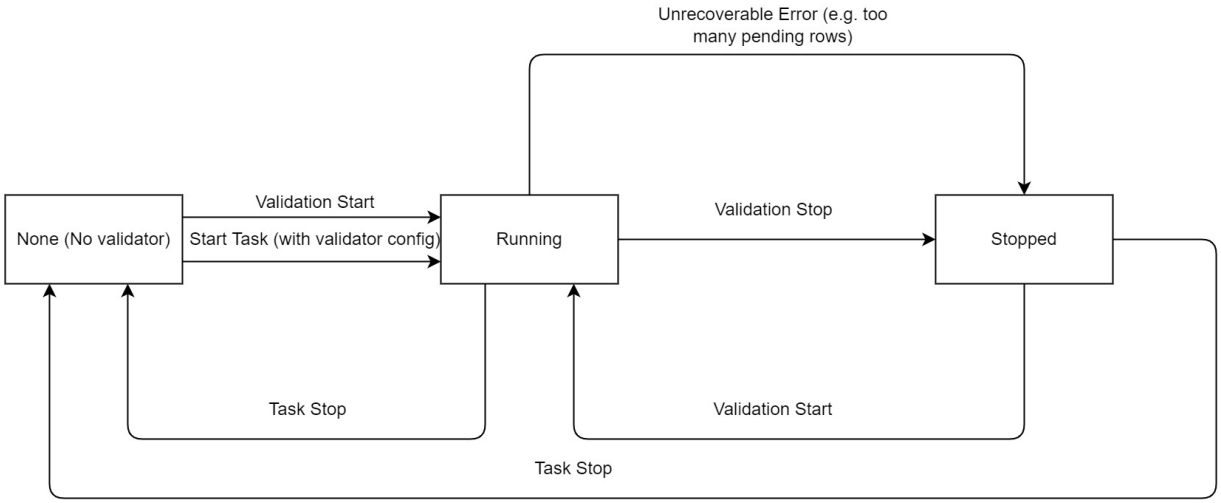


图 243: validator lifecycle

增量数据校验的具体处理流程如下：

1. validator 从上游拉取 binlog 事件，获取到发生变更的数据行：
    - validator 只会校验增量复制 (syncer) 完成的事件，如果该事件还没有被 syncer 处理，则 validator 会暂停，等待 syncer 处理完成。
    - 如果该事件已被 syncer 处理完成，则进行下面的步骤。
2. validator 将 binlog 解析，并通过黑白名单、过滤器的筛选，表路由的重定向（和 syncer 保持一致）之后，将这些行变动交给在后台运行的 validation worker。
3. validation worker 合并相同表、相同主键的行变动，避免进行“过期”的校验，并将这些行先缓存到内存中。
4. 当 validation worker 积攒了一定数量的行或者到了某个时间间隔之后，validation worker 根据这些行的主键信息在下游数据库中查询当下的数据，并和变动行期望的数据进行对比。
5. validation worker 进行数据校验。如果当前配置是 full mode，会将变动行和下游数据库中获取的行数据进行每列对比；如果当前配置是 fast mode，则只会判断这一行是否还存在。
    - 如果校验成功，则将该行从内存中删除。
    - 如果校验失败，不会马上报错，而是在一定间隔之后继续校验。
    - 对于某些已经在很长时间（由用户定义）内都没有校验成功的行，则将这些行定义为错误行（error row），写入到下游的 meta 库中。你可以通过查询迁移任务信息来获取错误行的数量等信息，详见查看增量校验的状态以及处理增量校验错误行。

13.6.7.4.6 使用限制

- 校验目标表必须有主键或者 not null 的唯一键。
- 上游迁移 DDL 时有以下限制：
    - DDL 不能变更主键，不能调整列顺序，不能删除已有列。

  - 该表不能被 DROP。

- 不支持按照 expression 过滤事件的任务。
- 由于 TiDB 和 MySQL 的浮点数精度有差异，精度范围内误差也会判断为相等（即绝对误差小于 10^-6）。
- 不支持校验的数据类型：

  - JSON
  - 二进制数据

13.6.8 运维管理

13.6.8.1 集群版本升级

13.6.8.1.1 使用 TiUP 运维 DM 集群

本文介绍如何使用 TiUP 的 DM 组件运维 DM 集群。

如果你还未部署 DM 集群，可参考使用 TiUP 部署 DM 集群。

> 注意：
>
> - 需要确保以下组件间端口可正常连通：
>   - 各 DM-master 节点间的 peer_port（默认为 8291）可互相连通。
>   - 各 DM-master 节点可连通所有 DM-worker 节点的 port（默认为 8262）。
>   - 各 DM-worker 节点可连通所有 DM-master 节点的 port（默认为 8261）。
>   - TiUP 节点可连通所有 DM-master 节点的 port（默认为 8261）。
>   - TiUP 节点可连通所有 DM-worker 节点的 port（默认为 8262）。

TiUP DM 组件的帮助信息如下：

```
tiup dm --help
```

```
Deploy a DM cluster for production

Usage:
  tiup dm [flags]
  tiup dm [command]

Available Commands:
  deploy      Deploy a DM cluster for production
  start       Start a DM cluster
  stop        Stop a DM cluster
  restart     Restart a DM cluster
  list        List all clusters
```

```
  destroy     Destroy a specified DM cluster
  audit       Show audit log of cluster operation
  exec        Run shell command on host in the dm cluster
  edit-config Edit DM cluster config
  display     Display information of a DM cluster
  reload      Reload a DM cluster's config and restart if needed
  upgrade     Upgrade a specified DM cluster
  patch       Replace the remote package with a specified package and restart the service
  scale-out   Scale out a DM cluster
  scale-in    Scale in a DM cluster
  import      Import an exist DM 1.0 cluster from dm-ansible and re-deploy 2.0 version
  help        Help about any command

Flags:
  -h, --help               help for tiup-dm
      --native-ssh         Use the native SSH client installed on local system instead of the
          ↪ build-in one.
      --ssh-timeout int    Timeout in seconds to connect host via SSH, ignored for operations
          ↪ that don't need an SSH connection. (default 5)
  -v, --version            version for tiup-dm
      --wait-timeout int   Timeout in seconds to wait for an operation to complete, ignored for
          ↪ operations that don't fit. (default 60)
  -y, --yes                Skip all confirmations and assumes 'yes'
```

查看集群列表

集群部署成功后，可以通过 `tiup dm list` 命令在集群列表中查看该集群：

```
tiup dm list
```

```
Name  User  Version  Path                                       PrivateKey
----  ----  -------  ----                                       ----------
prod-cluster  tidb  ${version}  /root/.tiup/storage/dm/clusters/test  /root/.tiup/storage/dm/
    ↪ clusters/test/ssh/id_rsa
```

启动集群

集群部署成功后，可以执行以下命令启动该集群。如果忘记了部署的集群名字，可以使用 `tiup dm list` 命令查看。

```
tiup dm start prod-cluster
```

检查集群状态

如果想查看集群中每个组件的运行状态，逐一登录到各个机器上查看显然很低效。因此，TiUP 提供了 `tiup dm display` 命令，用法如下：

```
tiup dm display prod-cluster
```

```
dm Cluster: prod-cluster
dm Version: ${version}
ID                  Role          Host          Ports      OS/Arch       Status     Data Dir
    ↪                             Deploy Dir
--                  ----          ----          -----      -------       ------     --------
    ↪                             ----------
172.19.0.101:9093   alertmanager  172.19.0.101  9093/9094  linux/x86_64  Up         /home/tidb/
    ↪ data/alertmanager-9093  /home/tidb/deploy/alertmanager-9093
172.19.0.101:8261   dm-master     172.19.0.101  8261/8291  linux/x86_64  Healthy|L  /home/tidb/
    ↪ data/dm-master-8261     /home/tidb/deploy/dm-master-8261
172.19.0.102:8261   dm-master     172.19.0.102  8261/8291  linux/x86_64  Healthy    /home/tidb/
    ↪ data/dm-master-8261     /home/tidb/deploy/dm-master-8261
172.19.0.103:8261   dm-master     172.19.0.103  8261/8291  linux/x86_64  Healthy    /home/tidb/
    ↪ data/dm-master-8261     /home/tidb/deploy/dm-master-8261
172.19.0.101:8262   dm-worker     172.19.0.101  8262       linux/x86_64  Free       /home/tidb/
    ↪ data/dm-worker-8262     /home/tidb/deploy/dm-worker-8262
172.19.0.102:8262   dm-worker     172.19.0.102  8262       linux/x86_64  Free       /home/tidb/
    ↪ data/dm-worker-8262     /home/tidb/deploy/dm-worker-8262
172.19.0.103:8262   dm-worker     172.19.0.103  8262       linux/x86_64  Free       /home/tidb/
    ↪ data/dm-worker-8262     /home/tidb/deploy/dm-worker-8262
172.19.0.101:3000   grafana       172.19.0.101  3000       linux/x86_64  Up         -
    ↪                                   /home/tidb/deploy/grafana-3000
172.19.0.101:9090   prometheus    172.19.0.101  9090       linux/x86_64  Up         /home/tidb/
    ↪ data/prometheus-9090    /home/tidb/deploy/prometheus-9090
```

Status 列用 Up 或者 Down 表示该服务是否正常。对于 DM-master 组件，同时可能会带有 |L 表示该 DM-master 是 Leader，对于 DM-worker 组件，Free 表示当前 DM-worker 没有与上游绑定。

缩容节点

缩容即下线服务，最终会将指定的节点从集群中移除，并删除遗留的相关数据文件。

缩容操作进行时，内部对 DM-master、 DM-worker 组件的操作流程为：

1. 停止组件进程
2. 调用 DM-master 删除 member 的 API
3. 清除节点的相关数据文件

缩容命令的基本用法：

```
tiup dm scale-in <cluster-name> -N <node-id>
```

它需要指定至少两个参数，一个是集群名字，另一个是节点 ID。节点 ID 可以参考上一节使用 `tiup dm display` 命令获取。

比如想缩容 172.16.5.140 上的 DM-worker 节点（DM-master 的缩容类似），可以执行：

```
tiup dm scale-in prod-cluster -N 172.16.5.140:8262
```

扩容节点

扩容的内部逻辑与部署类似，TiUP DM 组件会先保证节点的 SSH 连接，在目标节点上创建必要的目录，然后执行部署并且启动服务。

例如，在集群 `prod-cluster` 中扩容一个 DM-worker 节点（DM-master 的扩容类似）：

1. 新建 scale.yaml 文件，添加新增的 woker 节点信息：

> 注意：
> 需要新建一个拓扑文件，文件中只写入扩容节点的描述信息，不要包含已存在的节点。
> 其他更多配置项（如：部署目录等）请参考 TiUP 配置参数模版。

```
---


worker_servers:
  - host: 172.16.5.140
```

2. 执行扩容操作。TiUP DM 根据 scale.yaml 文件中声明的端口、目录等信息在集群中添加相应的节点：

```
tiup dm scale-out prod-cluster scale.yaml
```

执行完成之后可以通过 `tiup dm display prod-cluster` 命令检查扩容后的集群状态。

滚动升级

> 注意：
> 从 v2.0.5 版本开始，dmctl 支持导出和导入集群的数据源和任务配置。
> 升级前，可使用 `config export` 命令导出集群的配置文件，升级后如需降级回退到旧版本，可重建旧集群后，使用 `config import` 导入之前的配置。
> 对于 v2.0.5 之前版本的集群，可使用 v2.0.5 及之后版本的 dmctl 导出和导入集群配置。
> 对于 v2.0.2 之后的版本，导入集群配置时暂不支持自动恢复 relay worker 相关配置，可手动执行 `start-relay` 命令开启 relay log。

滚动升级过程中尽量保证对前端业务透明、无感知，其中对不同节点有不同的操作。

升级操作

可使用 `tiup dm upgrade` 命令来升级集群。例如，以下示例将集群升级到 `${version}`，执行命令前，将 `${version}` 替换为实际需要的版本：

```
tiup dm upgrade prod-cluster `${version}`
```

更新配置

如果想要动态更新组件的配置，TiUP DM 组件为每个集群保存了一份当前的配置，如果想要编辑这份配置，则执行 `tiup dm edit-config <cluster-name>` 命令。例如：

```
tiup dm edit-config prod-cluster
```

然后 TiUP DM 组件会使用 vi 打开配置文件供编辑（如果你想要使用其他编辑器，请使用 EDITOR 环境变量自定义编辑器，例如 `export EDITOR=nano`），编辑完之后保存即可。此时的配置并没有应用到集群，如果想要让它生效，还需要执行：

```
tiup dm reload prod-cluster
```

该操作会将配置发送到目标机器，滚动重启集群，使配置生效。

更新组件

常规的升级集群可以使用 upgrade 命令，但是在某些场景下（例如 Debug），可能需要用一个临时的包替换正在运行的组件，此时可以用 patch 命令：

```
tiup dm patch --help
```

```
Replace the remote package with a specified package and restart the service

Usage:
  tiup dm patch <cluster-name> <package-path> [flags]

Flags:
  -h, --help                    help for patch
  -N, --node strings            Specify the nodes
      --overwrite               Use this package in the future scale-out operations
  -R, --role strings            Specify the role
      --transfer-timeout int    Timeout in seconds when transferring dm-master leaders (default
          ↪ 600)

Global Flags:
      --native-ssh         Use the native SSH client installed on local system instead of the
          ↪ build-in one.
      --ssh-timeout int    Timeout in seconds to connect host via SSH, ignored for operations
          ↪ that don't need an SSH connection. (default 5)
      --wait-timeout int   Timeout in seconds to wait for an operation to complete, ignored for
          ↪ operations that don't fit. (default 60)
  -y, --yes                Skip all confirmations and assumes 'yes'
```

例如，有一个 DM-master 的 hotfix 包放在 `/tmp/dm-master-hotfix.tar.gz`，如果此时想要替换集群上的所有 DM-master，则可以执行：

```
tiup dm patch prod-cluster /tmp/dm-master-hotfix.tar.gz -R dm-master
```

或者只替换其中一个 DM-master：

```
tiup dm patch prod-cluster /tmp/dm--hotfix.tar.gz -N 172.16.4.5:8261
```

导入和升级由 DM-Ansible 部署的 DM 1.0 集群

> 注意：
>
> - TiUP 不支持在 DM 1.0 集群中导入 DM Portal 组件。
> - 导入 DM 集群前，终止运行待导入的集群。
> - 如果需要将迁移任务升级至 2.0，不可对该任务执行 `stop-run` 命令。
> - TiUP 仅支持导入数据到 v2.0.0-rc.2 或以上版本的 DM 集群。
> - `import` 命令用于将数据从 DM 1.0 集群导入到一个新的 DM 2.0 集群。如果你需要将 DM 迁移任务导入一个现有的 DM 2.0 集群，可以参考TiDB Data Migration 1.0.x 到 2.0+ 手动升级。
> - 导入集群某些组件的部署目录与原始集群的部署目录不同，可以执行 `display` 命令来查看相关信息。
> - 导入前，执行 `tiup update --self && tiup update dm`，以确保 DM 组件是最新版本。
> - 导入后，集群中只存在一个 DM-master 节点。请参考扩容节点 来扩展 DM-master。

引入 TiUP 前，DM-Ansible 用于部署 DM 集群。要使 TiUP 接管由 DM-Ansible 部署的 DM 1.0 集群，需要执行 `import` 命令：

例如，以下命令可导入使用 DM-Ansible 部署的集群：

```
tiup dm import --dir=/path/to/dm-ansible --cluster-version ${version}
```

可以执行 `tiup list dm-master` 来查看 TiUP 支持的最新集群版本。

`import` 命令的执行过程如下：

1. 根据之前使用 DM-Ansible 部署的 DM 集群，TiUP 生成一个拓扑文件 `topology.yml`。
2. 确认拓扑文件生成后，你可以用这个文件来部署 v2.0 或更高版本的 DM 集群。

部署完成后，执行 `tiup dm start` 命令来启动集群和开始 DM 内核升级流程。

查看操作日志

操作日志的查看可以借助 `audit` 命令，其用法如下：

```
Usage:
  tiup dm audit [audit-id] [flags]

Flags:
  -h, --help   help for audit
```

在不使用 `[audit-id]` 参数时，该命令会显示执行的命令列表，如下：

```
tiup dm audit
```

```
ID      Time                  Command
--      ----                  -------
4D5kQY  2020-08-13T05:38:19Z  tiup dm display test
4D5kNv  2020-08-13T05:36:13Z  tiup dm list
4D5kNr  2020-08-13T05:36:10Z  tiup dm deploy -p prod-cluster ${version} ./examples/dm/minimal.
    ↪ yaml
```

第一列为 audit-id，如果想看某个命令的执行日志，则传入这个 audit-id：

```
tiup dm audit 4D5kQY
```

在集群节点机器上执行命令

exec 命令可以很方便地到集群的机器上执行命令，使用方式如下：

```
Usage:
  tiup dm exec <cluster-name> [flags]

Flags:
      --command string   the command run on cluster host (default "ls")
  -h, --help             help for exec
  -N, --node strings     Only exec on host with specified nodes
  -R, --role strings     Only exec on host with specified roles
      --sudo             use root permissions (default false)
```

例如，如果要到所有的 DM 节点上执行 `ls /tmp`，则可以执行：

```
tiup dm exec prod-cluster --command='ls /tmp'
```

集群控制工具 (dmctl)

TiUP 集成了 DM 的控制工具 `dmctl`：

运行命令如下：

```
tiup dmctl [args]
```

指定 dmctl 版本。执行如下命令前，将 ${version} 修改为实际需要的版本。

```
tiup dmctl: ${version} [args]
```

例如，以前添加 source 命令为 `dmctl --master-addr master1:8261 operate-source create /tmp/source1.yml`，集成到 TiUP 中的命令为：

```
tiup dmctl --master-addr master1:8261 operate-source create /tmp/source1.yml
```

使用中控机系统自带的 SSH 客户端连接集群

在以上所有操作中，涉及到对集群机器的操作都是通过 TiUP 内置的 SSH 客户端连接集群执行命令，但是在某些场景下，需要使用系统自带的 SSH 客户端来对集群执行操作，比如：

- 使用 SSH 插件来做认证
- 使用定制的 SSH 客户端

此时可以通过命令行参数 `--native-ssh` 启用系统自带命令行：

- 部署集群：`tiup dm deploy <cluster-name> <version> <topo> --native-ssh`，其中 `<cluster-name>` 为集群名称，`<version>` 为 DM 集群版本（例如 `v6.5.0`），`<topo>` 为拓扑文件路径
- 启动集群：`tiup dm start <cluster-name> --native-ssh`
- 升级集群：`tiup dm upgrade ... --native-ssh`

所有涉及集群操作的步骤都可以加上 `--native-ssh` 来使用系统自带的客户端。

也可以使用环境变量 `TIUP_NATIVE_SSH` 来指定是否使用本地 SSH 客户端，避免每个命令都需要添加 `--native-ssh` 参数：

```
export TIUP_NATIVE_SSH=true
##### 或者
export TIUP_NATIVE_SSH=1
##### 或者
export TIUP_NATIVE_SSH=enable
```

若环境变量和 `--native-ssh` 同时指定，则以 `--native-ssh` 为准。

> 注意：
>
> 在部署集群的步骤中，若需要使用密码的方式连接 (-p)，或者密钥文件设置了 passphrase，则需要保证中控机上安装了 sshpass，否则连接时会报错。

13.6.8.1.2 TiDB Data Migration 1.0.x 到 2.0+ 手动升级

本文档主要介绍如何手动从 DM v1.0.x 升级到 v2.0+，主要思路为利用 v1.0.x 时的全局 checkpoint 信息在 v2.0+ 集群中启动一个新的增量数据复制任务。

> 注意：
>
> - DM 当前不支持在数据迁移任务处于全量导出或全量导入过程中从 v1.0.x 升级到 v2.0+。
> - 由于 DM 各组件间用于交互的 gRPC 协议进行了较大变更，因此需确保升级前后 DM 集群各组件（包括 dmctl）使用相同的版本。

- 由于 DM 集群的元数据存储（如 checkpoint、shard DDL lock 状态及 online DDL 元信息等）发生了较大变更，升级到 v2.0+ 后无法自动复用 v1.0.x 的元数据，因此在执行升级操作前需要确保：
  - 所有数据迁移任务不处于 shard DDL 协调过程中。
  - 所有数据迁移任务不处于 online DDL 协调过程中。

下面是手动升级的具体步骤。

第 1 步：准备 v2.0+ 的配置文件

准备的 v2.0+ 的配置文件包括上游数据库的配置文件以及数据迁移任务的配置文件。

上游数据库配置文件

在 v2.0+ 中将上游数据库 source 相关的配置从 DM-worker 的进程配置中独立了出来，因此需要根据v1.0.x 的 DM-worker 配置拆分得到 source 配置。

> 注意：
>
> 当前从 v1.0.x 升级到 v2.0+ 时，如在 source 配置中启用了 `enable-gtid`，则后续需要通过解析 binlog 或 relay log 文件获取 binlog position 对应的 GTID sets。

从 DM-Ansible 部署的 v1.0.x 升级

如果 v1.0.x 是使用 DM-Ansible 部署的，且假设在 `inventory.ini` 中有如下 `dm_worker_servers` 配置：

```
[dm_master_servers]
dm_worker1 ansible_host=172.16.10.72 server_id=101 source_id="mysql-replica-01" mysql_host
    ↪ =172.16.10.81 mysql_user=root mysql_password='VjX8cEeTX+qcvZ3bPaO4h0C80pe/1aU='
    ↪ mysql_port=3306
dm_worker2 ansible_host=172.16.10.73 server_id=102 source_id="mysql-replica-02" mysql_host
    ↪ =172.16.10.82 mysql_user=root mysql_password='VjX8cEeTX+qcvZ3bPaO4h0C80pe/1aU='
    ↪ mysql_port=3306
```

则可以转换得到如下两个 source 配置文件：

```
##### 原 dm_worker1 对应的 source 配置，如命名为 source1.yaml
server-id: 101                                  # 对应原 `server_id`
source-id: "mysql-replica-01"                   # 对应原 `source_id`
from:
  host: "172.16.10.81"                          # 对应原 `mysql_host`
  port: 3306                                    # 对应原 `mysql_port`
  user: "root"                                  # 对应原 `mysql_user`
  password: "VjX8cEeTX+qcvZ3bPaO4h0C80pe/1aU="  # 对应原 `mysql_password`
```

```
##### 原 dm_worker2 对应的 source 配置，如命名为 source2.yaml
server-id: 102                                  # 对应原 `server_id`
source-id: "mysql-replica-02"                   # 对应原 `source_id`
from:
  host: "172.16.10.82"                          # 对应原 `mysql_host`
  port: 3306                                    # 对应原 `mysql_port`
  user: "root"                                  # 对应原 `mysql_user`
  password: "VjX8cEeTX+qcvZ3bPaO4h0C80pe/1aU="   # 对应原 `mysql_password`
```

从 Binary 部署的 v1.0.x 升级

如果 v1.0.x 是使用 Binary 部署的，且对应的 DM-worker 配置如下：

```
log-level = "info"
log-file = "dm-worker.log"
worker-addr = ":8262"

server-id = 101
source-id = "mysql-replica-01"
flavor = "mysql"

[from]
host = "172.16.10.81"
user = "root"
password = "VjX8cEeTX+qcvZ3bPaO4h0C80pe/1aU="
port = 3306
```

则可转换得到如下的一个 source 配置文件：

```
server-id: 101                                  # 对应原 `server-id`
source-id: "mysql-replica-01"                   # 对应原 `source-id`
flavor: "mysql"                                 # 对应原 `flavor`
from:
  host: "172.16.10.81"                          # 对应原 `from.host`
  port: 3306                                    # 对应原 `from.port`
  user: "root"                                  # 对应原 `from.user`
  password: "VjX8cEeTX+qcvZ3bPaO4h0C80pe/1aU="   # 对应原 `from.password`
```

数据迁移任务配置文件

对于数据迁移任务配置向导，v2.0+ 基本与 v1.0.x 保持兼容，可直接复制 v1.0.x 的配置。

第 2 步：部署 v2.0+ 集群

> **注意：**
>
> 如果已有其他可用的 v2.0+ 集群，可跳过此步。

使用 TiUP 按所需要节点数部署新的 v2.0+ 集群。

第 3 步：下线 v1.0.x 集群

如果原 v1.0.x 集群是使用 DM-Ansible 部署的，则使用 DM-Ansible 下线 v1.0.x 集群。

如果原 v1.0.x 集群是使用 Binary 部署，则直接停止 DM-worker 与 DM-master 进程。

第 4 步：升级数据迁移任务

1. 使用operate-source 命令将准备 v2.0+ 的配置文件中得到的上游数据库 source 配置加载到 v2.0+ 集群中。
2. 在下游 TiDB 中，从 v1.0.x 的数据复制任务对应的增量 checkpoint 表中获取对应的全局 checkpoint 信息。
   - 假设 v1.0.x 的数据迁移配置中未额外指定 `meta-schema`（或指定其值为默认的`dm_meta`），且对应的任务名为 `task_v1`，则对应的 checkpoint 信息在下游 TiDB 的 `` `dm_meta`.`task_v1_syncer_checkpoint` `` 表中。
   - 使用以下 SQL 语句分别获取该数据迁移任务对应的所有上游数据库 source 的全局 checkpoint 信息。

```
> SELECT `id`, `binlog_name`, `binlog_pos` FROM `dm_meta`.`task_v1_syncer_checkpoint`
    ↪ WHERE `is_global`=1;
+------------------+--------------------------+------------+
| id               | binlog_name              | binlog_pos |
+------------------+--------------------------+------------+
| mysql-replica-01 | mysql-bin|000001.000123 | 15847      |
| mysql-replica-02 | mysql-bin|000001.000456 | 10485      |
+------------------+--------------------------+------------+
```

3. 更新 v1.0.x 的数据迁移任务配置文件以启动新的 v2.0+ 数据迁移任务。
   - 如 v1.0.x 的数据迁移任务配置文件为 `task_v1.yaml`，则将其复制一份为 `task_v2.yaml`。
   - 对 `task_v2.yaml` 进行以下修改：
     - 将 name 修改为一个新的、不存在的名称，如 `task_v2`
     - 将 `task-mode` 修改为 `incremental`
     - 根据 step.2 中获取的全局 checkpoint 信息，为各 source 设置增量复制的起始点，如：

```
mysql-instances:
  - source-id: "mysql-replica-01"        # 对应 checkpoint 信息所属的 `id`
    meta:
      binlog-name: "mysql-bin.000123"    # 对应 checkpoint 信息中的 `binlog_name`，
          ↪ 但不包含 `|000001` 部分
      binlog-pos: 15847                  # 对应 checkpoint 信息中的 `binlog_pos`

  - source-id: "mysql-replica-02"
    meta:
      binlog-name: "mysql-bin.000456"
      binlog-pos: 10485
```

> **注意：**
> 如在 source 配置中启动了 `enable-gtid`，当前需要通过解析 binlog 或 relay log 文件获取 binlog position 对应的 GTID sets 并在 `meta` 中设置为 `binlog-gtid`。

4. 使用`start-task` 命令以 v2.0+ 的数据迁移任务配置文件启动升级后的数据迁移任务。
5. 使用`query-status` 命令确认数据迁移任务是否运行正常。

如果数据迁移任务运行正常，则表明 DM 升级到 v2.0+ 的操作成功。

13.6.8.2 集群运维工具

13.6.8.2.1 使用 WebUI 管理 DM 迁移任务

DM WebUI 是一个 TiDB Data Migration (DM) 迁移任务管理界面，方便用户以直观的方式管理大量迁移任务，无需使用 dmctl 命令行，简化任务管理的操作步骤。

本文档介绍 DM WebUI 的访问方式、使用前提、各界面的使用场景以及注意事项。

> **警告：**
>
> - DM WebUI 当前为实验特性，不建议在生产环境中使用。
> - DM WebUI 中 `task` 的生命周期有所改变，不建议与 dmctl 同时使用。

DM WebUI 主要包含以下界面：

- 数据迁移
  - 任务列表：提供创建迁移任务的界面入口，并展示各迁移任务的详细信息。帮助用户监控、创建、删除、配置迁移任务。
  - 上游配置：用户在此页面配置同步任务的上游数据源信息。管理数据迁移环境中的上游配置，包含了，新增、删除上游配置、监控上游配置对应的同步任务状态、修改上游配置等相关的管理功能。
  - 同步详情：展示更加详细的任务状态信息。根据用户指定的筛选条件查看同步任务的具体配置和状态信息，包括上下游配置信息，上下游数据库名称，源表和目标表的关系等。
- 集群管理
  - 成员列表：展示 DM 集群中所有的 master 和 worker 节点，以及 worker 节点与 source 的绑定关系。查看当前 DM 集群的配置信息和各个 worker 的状态信息，并提供基本的管理功能。

界面示例如下：

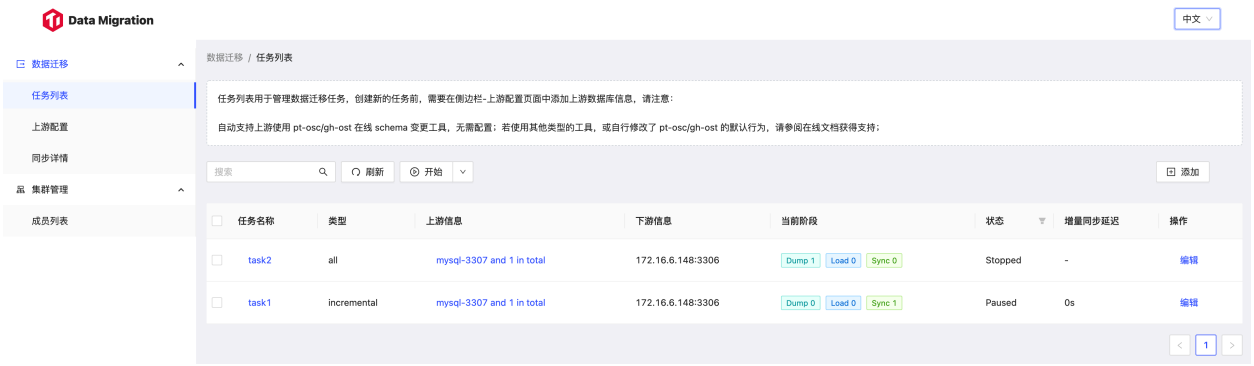


图 244: webui

访问方式

在开启OpenAPI 后，你可以从 DM 集群的任意 master 节点访问 DM WebUI，访问端口与 DM OpenAPI 保持一致，默认为 8261。访问地址示例：`http://{master_ip}:{master_port}/dashboard/`。

数据迁移

数据迁移包含上游配置、任务列表、同步详情三个界面。

上游配置

创建迁移任务之前，你需要先创建同步任务的上游数据源信息。你可在上游配置页面创建上游任务的配置。创建时，请注意以下事项：

- 如果存在主从切换，请务必在上游 MySQL 开启 GTID，并在创建上游配置时将 GTID 设为 `True`，否则数据迁移任务将在主从切换时中断（AWS Aurora 除外）。
- 若某个上游数据库需要临时下线，可将其“停用”，但停用期间其他正在同步的 MySQL 实例不可执行 DDL 操作，否则停用的实例被启用后将无法正常同步。
- 当多个迁移任务使用同一个上游时，可能对其造成额外压力。开启 relay log 可降低对上游的影响，建议开启 relay log。

任务列表

你可通过任务列表界面查看迁移任务详情，并创建迁移任务。

查看迁移任务详情

在任务列表中，点击任务名称，详情页面会从右侧滑出。详情页面展示了更加详细的任务状态信息。在信息详情页面，你可以查看每一个子任务的运行情况，以及此迁移任务当前完整的配置项信息。

在 DM 中，迁移任务中的每一个子任务可能处于不同的阶段，即全量导出 (dump) -> 全量导入 (load) -> 增量同步 (sync)。因此任务的当前阶段以子任务所处阶段的统计信息来展示，可以更加清楚的了解任务运行情况。

创建迁移任务

要在该界面创建任务，点击右上角的添加按钮即可。创建迁移任务时，你可以使用以下任一方式：

- 通过向导方式。通过 WebUI 根据指引一步步填写所需信息进行任务创建，此种方式比较适合入门级用户及日常使用。

- 通过配置文件。通过直接粘贴或编写 JSON 格式的任务配置文件进行创建，支持更多的参数调整，适合熟练的用户使用。

同步详情

你可以通过同步详情页面查看迁移任务中所配置迁移规则的运行情况。同步详情页面支持根据任务、数据源、表库名称进行查询。

查询结果中包含上游表至下游表的对应信息，因此请慎重使用 .* 等，以防止查询结果过多导致页面反应迟缓。

集群管理

成员列表

成员列表页面展示 DM 集群中所有的 master 和 worker 节点，以及 worker 节点与 source 的绑定关系。

13.6.8.2.2 使用 dmctl 运维 TiDB Data Migration 集群

> 注意：
> 对于用 TiUP 部署的 DM 集群，推荐直接使用`tiup dmctl`命令。

dmctl 是用来运维 DM 集群的命令行工具，支持交互模式和命令模式。

dmctl 交互模式

进入交互模式，与 DM-master 进行交互：

> 注意：
> 交互模式下不具有 bash 的特性，比如不需要通过引号传递字符串参数而应当直接传递。

```
./dmctl --master-addr 172.16.30.14:8261
```

```
Welcome to dmctl
Release Version: ${version}
Git Commit Hash: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Git Branch: release-x.x
UTC Build Time: yyyy-mm-dd hh:mm:ss
Go Version: go version gox.xx linux/amd64

» help
DM control
```

```
Usage:
  dmctl [command]

Available Commands:
  binlog          manage or show binlog operations
  binlog-schema   manage or show table schema in schema tracker
  check-task      Checks the configuration file of the task
  config          manage config operations
  decrypt         Decrypts cipher text to plain text
  encrypt         Encrypts plain text to cipher text
  help            Gets help about any command
  list-member     Lists member information
  offline-member  Offlines member which has been closed
  operate-leader  `evict`/`cancel-evict` the leader
  operate-source  `create`/`stop`/`show` upstream MySQL/MariaDB source
  pause-relay     Pauses DM-worker's relay unit
  pause-task      Pauses a specified running task or all (sub)tasks bound to a source
  purge-relay     Purges relay log files of the DM-worker according to the specified filename
  query-status    Queries task status
  resume-relay    Resumes DM-worker's relay unit
  resume-task     Resumes a specified paused task or all (sub)tasks bound to a source
  shard-ddl-lock  maintain or show shard-ddl locks information
  start-relay     Starts workers pulling relay log for a source
  start-task      Starts a task as defined in the configuration file
  stop-relay      Stops workers pulling relay log for a source
  stop-task       Stops a specified task or all (sub)tasks bound to a source
  transfer-source Transfers a upstream MySQL/MariaDB source to a free worker

Flags:
  -h, --help             help for dmctl
  -s, --source strings   MySQL Source ID.

Use "dmctl [command] --help" for more information about a command.
```

dmctl 命令模式

命令模式跟交互模式的区别是，执行命令时只需要在 dmctl 命令后紧接着执行任务操作，任务操作同交互模式的参数一致。

> **注意：**
>
> - 一条 dmctl 命令只能跟一个任务操作
> - 从 v2.0.4 版本开始，支持从环境变量 (DM_MASTER_ADDR) 里读取 `-master-addr` 参数

```
./dmctl --master-addr 172.16.30.14:8261 start-task task.yaml
./dmctl --master-addr 172.16.30.14:8261 stop-task task
./dmctl --master-addr 172.16.30.14:8261 query-status

export DM_MASTER_ADDR="172.16.30.14:8261"
./dmctl query-status
```

```
Available Commands:
  binlog          manage or show binlog operations
  binlog-schema   manage or show table schema in schema tracker
  check-task      Checks the configuration file of the task
  config          manage config operations
  decrypt         Decrypts cipher text to plain text
  encrypt         Encrypts plain text to cipher text
  help            Gets help about any command
  list-member     Lists member information
  offline-member  Offlines member which has been closed
  operate-leader  `evict`/`cancel-evict` the leader
  operate-source  `create`/`stop`/`show` upstream MySQL/MariaDB source
  pause-relay     Pauses DM-worker's relay unit
  pause-task      Pauses a specified running task or all (sub)tasks bound to a source
  purge-relay     Purges relay log files of the DM-worker according to the specified filename
  query-status    Queries task status
  resume-relay    Resumes DM-worker's relay unit
  resume-task     Resumes a specified paused task or all (sub)tasks bound to a source
  shard-ddl-lock  maintain or show shard-ddl locks information
  start-relay     Starts workers pulling relay log for a source
  start-task      Starts a task as defined in the configuration file
  stop-relay      Stops workers pulling relay log for a source
  stop-task       Stops a specified task or all (sub)tasks bound to a source
  transfer-source Transfers a upstream MySQL/MariaDB source to a free worker

Flags:
      --config string        Path to config file.
  -h, --help                 help for dmctl
      --master-addr string   Master API server address, this parameter is required when
          ↪ interacting with the dm-master
      --rpc-timeout string   RPC timeout, default is 10m. (default "10m")
  -s, --source strings       MySQL Source ID.
      --ssl-ca string        Path of file that contains list of trusted SSL CAs for connection.
      --ssl-cert string      Path of file that contains X509 certificate in PEM format for
          ↪ connection.
      --ssl-key string       Path of file that contains X509 key in PEM format for connection.
  -V, --version              Prints version and exit.
```

```
Use "dmctl [command] --help" for more information about a command.
```

13.6.8.3 性能调优

13.6.8.3.1 DM 5.4.0 性能测试报告

TiDB Data Migration (DM) 5.4.0 的性能测试数据与 DM 5.3.0 的测试数据持平，因此不单独提供测试报告。若要了解 DM 5.3.0 版本的性能测试目的、环境、场景和结果，请参阅 DM 5.3.0 性能测试报告。

13.6.8.3.2 DM 配置优化

本文档介绍如何对迁移任务的配置进行优化，从而提高 DM 的数据迁移性能。

全量导出

全量导出相关的配置项为 `mydumpers`，下面介绍和性能相关的参数如何配置。

`rows`

设置 `rows` 选项可以开启单表多线程并发导出，值为导出的每个 chunk 包含的最大行数。开启后，DM 会在 MySQL 的单表并发导出时，优先选出一列做拆分基准，选择的优先级为主键 > 唯一索引 > 普通索引，选出目标列后需保证该列为整数类型（如 `INT`、`MEDIUMINT`、`BIGINT` 等）。

`rows` 的值可以设置为 10000，具体设置的值可以根据表中包含数据的总行数以及数据库的性能做调整。另外也需要设置 `threads` 来控制并发线程数量，默认值为 4，可以适当做些调整。

`chunk-filesize`

DM 全量备份时会根据 `chunk-filesize` 参数的值把每个表的数据划分成多个 chunk，每个 chunk 保存到一个文件中，大小约为 `chunk-filesize`。根据这个参数把数据切分到多个文件中，这样就可以利用 DM Load 处理单元的并行处理逻辑提高导入速度。该参数默认值为 64（单位为 MB），正常情况下不需要设置，也可以根据全量数据的大小做适当的调整。

> 注意：
>
> - `mydumpers` 的参数值不支持在迁移任务创建后更新，所以需要在创建任务前确定好各个参数的值。如果需要更新，则需要使用 dmctl stop 任务后更新配置文件，然后再重新创建任务。
> - `mydumpers.threads` 可以使用配置项 `mydumper-thread` 替代来简化配置。
> - 如果设置了 `rows`，DM 会忽略 `chunk-filesize` 的值。

全量导入

全量导入相关的配置项为 `loaders`，下面介绍和性能相关的参数如何配置。

`pool-size`

pool-size 为 DM Load 阶段线程数量的设置，默认值为 16，正常情况下不需要设置，也可以根据全量数据的大小以及数据库的性能做适当的调整。

> 注意：
>
> - loaders 的参数值不支持在迁移任务创建后更新，所以需要在创建任务前确定好各个参数的值。如果需要更新，则需要使用 dmctl stop 任务后更新配置文件，然后再重新创建任务。
> - loaders.pool-size 可以使用配置项 loader-thread 替代来简化配置。

增量复制

增量复制相关的配置为 syncers，下面介绍和性能相关的参数如何配置。

worker-count

worker-count 为 DM Sync 阶段并发迁移 DML 的线程数量设置，默认值为 16，如果对迁移速度有较高的要求，可以适当调高改参数的值。

batch

batch 为 DM Sync 阶段迁移数据到下游数据库时，每个事务包含的 DML 的数量，默认值为 100，正常情况下不需要调整。

> 注意：
>
> - syncers 的参数值不支持在迁移任务创建后更新，所以需要在创建任务前确定好各个参数的值。如果需要更新，则需要使用 dmctl stop 任务后更新配置文件，然后再重新创建任务。
> - syncers.worker-count 可以使用配置项 syncer-thread 替代来简化配置。
> - worker-count 和 batch 的设置需要根据实际的场景进行调整，例如：DM 到下游数据库的网络延迟较高，可以适当调高 worker-count，调低 batch。

13.6.8.3.3 DM 集群性能测试

本文档介绍如何构建测试场景对 DM 集群进行性能测试，包括数据迁移速度、延迟等。

迁移数据流

可以使用简单的迁移数据流来测试 DM 集群的数据迁移性能，即单个 MySQL 实例到 TiDB 的数据迁移：MySQL -> DM -> TiDB。

部署测试环境

- 使用 TiUP 部署 TiDB 测试集群，所有配置使用 TiUP 提供的默认配置。

- 部署 MySQL 服务，开启 ROW 模式 binlog，其他配置项使用默认配置。
- 部署 DM 集群，部署一个 DM-worker 和一个 DM-master 即可。

性能测试

迁移数据表结构

使用如下结构的表进行性能测试：

```
CREATE TABLE `sbtest` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `k` int(11) NOT NULL DEFAULT '0',
  `c` char(120) CHARSET utf8mb4 COLLATE utf8mb4_bin NOT NULL DEFAULT '',
  `pad` char(60) CHARSET utf8mb4 COLLATE utf8mb4_bin NOT NULL DEFAULT '',
  PRIMARY KEY (`id`),
  KEY `k_1` (`k`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin
```

全量导入性能测试用例

生成测试数据

使用 sysbench 在上游创建测试表，并生成全量导入的测试数据。sysbench 生成数据的命令如下所示：

```
sysbench --test=oltp_insert --tables=4 --mysql-host=172.16.4.40 --mysql-port=3306 --mysql-user=
    ↪ root --mysql-db=dm_benchmark --db-driver=mysql --table-size=50000000 prepare
```

创建数据迁移任务

1. 创建上游 MySQL 的 source，将 source-id 配置为 source-1。详细操作方法参考：加载数据源配置。
2. 创建 full 模式的 DM 迁移任务，示例任务配置文件如下：

“ ‘yaml

---

name: test-full

task-mode: full

使用实际测试环境中 TiDB 的信息配置 target-database: host: “192.168.0.1” port: 4000 user: “root” password: “”

mysql-instances: - source-id: “source-1” block-allow-list: “instance” # 如果 DM 版本早于 v2.0.0-beta.2 则使用 black-white-list mydumper-config-name: “global” loader-thread: 16

配置 sysbench 生成数据所在的库的名称 block-allow-list: # 如果 DM 版本早于 v2.0.0-beta.2 则使用 black-white-list instance: do-dbs: [“dm_benchmark”]

mydumpers: global: rows: 32000 threads: 32 “ ‘

创建数据迁移任务的详细操作参考创建数据迁移任务。

注意：

- 在 mydumpers 配置项中使用 rows 选项，可以开启单表多线程并发导出，加快数据导出速度。
- mysql-instances 配置中的 loader-thread 以及 mydumpers 配置项中的 rows 和 threads 可以做适当调整，测试在不同配置下对性能的影响。

获取测试结果

观察 DM-worker 日志，当出现 all data files have been finished 时，表示全量数据导入完成，此时可以看到消耗时间。示例日志如下：

```
[INFO] [loader.go:604] ["all data files have been finished"] [task=test] [unit=load] ["cost time
    ↪ "=52.439796ms]
```

根据测试数据的数据量和导入消耗时间，可以算出全量数据的迁移速度。

增量复制性能测试用例

初始化表

使用 sysbench 在上游创建测试表。

创建数据迁移任务

1. 创建上游 MySQL 的 source, source-id 配置为 source-1（如果在全量迁移性能测试中已经创建，则不需要再次创建）。详细操作方法参考：加载数据源配置。
2. 创建 all 模式的 DM 迁移任务，示例任务配置文件如下：

“‘yaml

name: test-all

task-mode: all

使用实际测试环境中 TiDB 的信息配置 target-database: host: “192.168.0.1” port: 4000 user: “root” password: “”

mysql-instances: - source-id: “source-1” block-allow-list: “instance” # 如果 DM 版本早于 v2.0.0-beta.2 则使用 black-white-list syncer-config-name: “global”

配置 sysbench 生成数据所在的库的名称 block-allow-list: # 如果 DM 版本早于 v2.0.0-beta.2 则使用 black-white-list instance: do-dbs: [“dm_benchmark”]

syncers: global: worker-count: 16 batch: 100 “‘

创建数据迁移任务的详细操作参考创建数据迁移任务。

> **注意：**
>
> syncers 配置项中的 worker-count 和 batch 可以做适当调整，测试在不同配置下性能的差异。

生成增量数据

执行 sysbench 命令在上游持续生成增量数据：

```
sysbench --test=oltp_insert --tables=4 --num-threads=32 --mysql-host=172.17.4.40 --mysql-port
    ↪ =3306 --mysql-user=root --mysql-db=dm_benchmark --db-driver=mysql --report-interval=10 --
    ↪ time=1800 run
```

> **注意：**
>
> 可以通过调整 sysbench 的语句类型，测试在不同业务场景下 DM 的数据迁移性能。

获取测试结果

通过 query-status 命令观测 DM 的迁移状态，通过 Grafana 观测 DM 的监控指标。主要包括单位时间内完成的 job 数量 finished sqls jobs 等，详细的监控指标说明参考Binlog replication 监控指标。

13.6.8.3.4 TiDB Data Migration 性能问题及处理方法

本文档介绍 TiDB Data Migration (DM) 中可能存在的、常见的性能问题及其处理方法。

在诊断与处理性能问题时，请确保已经正确配置并安装 DM 的监控组件，并能在 Grafana 监控面板查看DM 的监控指标。

在诊断性能问题时，请先确保对应组件正在正常运行，否则可能出现监控指标异常的情况，对性能问题的诊断造成干扰。

在诊断问题前，也可以先了解 DM 的性能测试报告。

当数据迁移过程存在较大延迟时，若需快速定位瓶颈是在 DM 组件内部还是在 TiDB 集群，可先排查写入 SQL 到下游 部分的 DML queue remain length。

relay log 模块的性能问题及处理方法

在relay log 的监控部分，可以主要通过 binlog file gap between master and relay 监控项确认是否存在性能问题。如果该指标长时间大于 1，通常表明存在性能问题；如果该指标基本为 0，一般表明没有性能问题。

如果 binlog file gap between master and relay 基本为 0，但仍怀疑存在性能问题，则可以继续查看 binlog pos，如果该指标中 master 远大于 relay，则表明可能存在性能问题。

如果存在性能问题，则继续根据 relay log 模块的主要处理流程分别进行诊断与处理。

读取 binlog 数据

与 relay log 模块从上游读取 binlog 数据相关的主要性能指标是 `read binlog event duration`，该指标表示从上游 MySQL/MariaDB 读取到单个 binlog event 所需要的时间，理想情况下应接近于 DM-worker 与 MySQL/MariaDB 实例间的网络延迟。

对于同机房的数据迁移，这部分一般不会成为性能瓶颈；如果该值过大，请排查 DM-worker 与 MySQL/MariaDB 间的网络连通情况。

对于跨机房的数据迁移，可尝试将 DM-worker 与 MySQL/MariaDB 部署在同一机房，而仍将 TiDB 集群部署在目标机房。

从上游读取 binlog 数据这一流程细分后包括以下三个子流程：

- 上游 MySQL/MariaDB 从本地读取 binlog 数据并通过网络进行发送。上游 MySQL/MariaDB 负载无异常时，该子流程通常不会成为瓶颈。
- binlog 数据通过网络从 MySQL/MariaDB 所在机器传输到 DM-worker 所在机器。该子流程主要由 DM-worker 与上游 MySQL/MariaDB 的网络连通情况决定。
- DM-worker 从网络数据流中读取 binlog 数据，并构造成 binlog event。当 DM-worker 负载无异常时，该子流程通常不会成为瓶颈。

> 注意：
>
> 如果 `read binlog event duration` 的值较大，另一个可能的原因是上游 MySQL/MariaDB 负载较低，一段时间内暂时没有需要发送给 DM 的 binlog event，relay log 模块处于等待状态，导致该值包含了额外的等待时间。

binlog 数据解码与验证

将 binlog event 读取到 DM 内存后，会进行必要的数据解码与验证，这部分通常不会存在性能瓶颈，因此监控面板上默认无对应性能指标。如果需要查看相应指标，可手动为 Prometheus 中的 `dm_relay_read_transform_duration` 添加相应的监控。

写入 relay log 文件

在将 binlog event 写入 relay log 文件时，相关的主要性能指标是 `write relay log duration`，该指标在 `binlog event size` 不是特别大时，值应在微秒级别。如果该值过大，需排查磁盘写入性能，如尽量优先为 DM-worker 使用本地 SSD 等。

Load 模块的性能问题及处理方法

Load 模块主要操作为从本地读取 SQL 文件数据并写入到下游，对应的主要性能指标是 `transaction execution latency`，如果该值过大，则通常需要根据下游数据库的监控对下游性能进行排查。

另外，也可以查看是否 DM 到下游数据库间的网络存在较大的延迟。

Binlog replication 模块的性能问题及处理方法

在Binlog replication 的监控部分，可以主要通过 `binlog file gap between master and syncer` 监控项确认是否存在性能问题，如果该指标长时间大于 1，则通常表明存在性能问题；如果该指标基本为 0，则一般表明没有性能问题。

如果 binlog file gap between master and syncer 长时间大于 1，则可以再通过 binlog file gap between relay and syncer 判断延迟主要存在于哪个模块，如果该值基本为 0，则延迟可能存在于 relay log 模块，请先参考relay log 模块的性能问题及处理方法 进行处理；否则继续对 Binlog replication 进行排查。

读取 binlog 数据

Binlog replication 模块会根据配置选择从上游 MySQL/MariaDB 或 relay log 文件中读取 binlog event，对应的主要性能指标是 read binlog event duration，该值的范围一般是几微秒至几十微秒。

- 如果是从上游 MySQL/MariaDB 读取 binlog event，则可参考 relay log 模块下的读取 binlog 数据进行排查与处理。
- 如果是从 relay log 文件中读取，则在 binlog event size 不是特别大时，read binlog event duration 的值应在微秒级别。如果 read binlog event duration 过大，则需排查磁盘读取性能，如尽量优先为 DM-worker 使用本地 SSD 等。

binlog event 转换

Binlog replication 模块从 binlog event 数据中尝试构造 DML、解析 DDL 以及进行table router 转换等，主要的性能指标是 transform binlog event duration。

这部分的耗时受上游写入的业务特点影响较大，如对于 INSERT INTO 语句，转换单个 VALUES 的时间和转换大量 VALUES 的时间差距很多，其波动范围可能从几十微秒至上百微秒，但一般不会成为系统的瓶颈。

写入 SQL 到下游

Binlog replication 模块将转换后的 SQL 写入到下游时，涉及到的性能指标主要包括 DML queue remain length 与 transaction execution latency。

DM 在从 binlog event 构造出 SQL 后，会使用 worker-count 个队列尝试并发写入到下游。但为了避免监控条目过多，会将并发队列编号按 8 取模，即所有并发队列在监控上会对应到 q_0 到 q_7 的某一项。

DML queue remain length 用于表示并发处理队列中尚未取出并开始用于向下游写入的 DML 语句数，理想情况下，各 q_* 对应的曲线应该基本一致，如果极不一致则表明并发的负载极不均衡。

如果负载不均衡，请确认需要迁移的所有表结构中都有主键或唯一键，如没有主键或唯一键则请尝试为其添加主键或唯一键；如果存在主键或唯一键时仍存在该问题，可尝试升级 DM 到 v1.0.5 及以上的版本。

当整个数据迁移链路无明显延迟时，DML queue remain length 对应曲线应基本为 0，且最大通常应不超过任务配置文件中的 batch 值。

如果确认数据迁移链路存在明显延迟，且 DML queue remain length 中各 q_* 对应的曲线基本一致且基本为 0，则表明 DM 未能及时地从上游读取数据、进行转换或进行并行分发（如瓶颈存在于 relay log 模块等），请参考本文档前述各节进行排查。

如果 DML queue remain length 对应曲线不为 0（最大一般不超过 1024），则通常表明向下游写入 SQL 时存在瓶颈，可通过 transaction execution latency 查看向下游执行单个事务的耗时情况。

transaction execution latency 一般应在几十毫秒。如果该值过高，则通常需要根据下游数据库的监控对下游性能进行排查，另外也可以关注是否 DM 到下游数据库间的网络存在较大的延迟。

此外，也可通过 statement execution latency 查看向下游写入 BEGIN、INSERT/UPDATE/DELETE、COMMIT 等单条语句的耗时情况。

13.6.8.4 数据源管理

13.6.8.4.1 切换 DM-worker 与上游 MySQL 实例的连接

当需要对 DM-worker 所连接的上游 MySQL 实例进行停机维护或该实例意外宕机时，需要将 DM-worker 的连接切换到同一个主从复制集群内的另一个 MySQL 实例上。本文介绍如何将 DM-worker 的连接从一个 MySQL 实例切换到另一个 MySQL 实例上。

> 注意：
>
> - 仅支持在同一个主从复制集内的 MySQL 实例间进行切换。
> - 将要切换到的 MySQL 实例必须拥有 DM-worker 所需的 binlog。
> - DM-worker 必须以 GTID sets 模式运行，即对应 source 配置文件中指定 `enable-gtid: true`。
> - DM 仅支持以下两种切换场景，且必须严格按照各场景的步骤执行操作，否则可能需要根据切换后的 MySQL 实例重新搭建 DM 集群并完整重做数据迁移任务。

有关 GTID sets 的概念解释，请参考 MySQL 文档。

虚拟 IP 环境下切换 DM-worker 与 MySQL 实例的连接

如果 DM-worker 通过虚拟 IP (VIP) 连接上游的 MySQL 实例，更改 VIP 所指向的 MySQL 实例，即是在 DM-worker 对应上游连接地址不变的情况下切换 DM-worker 实际所连接的 MySQL 实例。

> 注意：
>
> 如果不对 DM 执行必要变更，当切换 VIP 所指向的 MySQL 实例时，DM 内部不同的 connection 可能会同时连接到切换前后不同的 MySQL 实例，造成 DM 拉取的 binlog 与从上游获取到的其他状态不一致，从而导致难以预期的异常行为甚至数据损坏。

如果 DM-worker 连接的 VIP 需要指向新的 MySQL 实例，需要按以下步骤进行操作：

1. 使用 `query-status` 命令获取当前 binlog replication 处理单元已复制到下游的 binlog 对应的 GTID sets (`syncerBinlogGtid`)，记为 `gtid-S`。
2. 在将要切换到的 MySQL 实例上使用 `SELECT @@GLOBAL.gtid_purged;` 获取已经被 purged 的 binlog 对应的 GTID sets，记为 `gtid-P`。
3. 在将要切换到的 MySQL 实例上使用 `SELECT @@GLOBAL.gtid_executed;` 获取所有已经执行成功的事务对应的 GTID sets，记为 `gtid-E`。
4. 确保满足以下关系，否则不支持将 DM-worker 连接切换到相应的 MySQL 实例：
    - `gtid-S` 包含 `gtid-P`（`gtid-P` 可以为空）
    - `gtid-E` 包含 `gtid-S`
5. 使用 `pause-task` 命令暂停所有运行中的数据迁移任务。

6. 变更 VIP 以指向新的 MySQL 实例。
7. 使用 `resume-task` 命令恢复之前的数据迁移任务。

变更 DM-worker 连接的上游 MySQL 实例地址

若要变更 DM-worker 的配置信息来使 DM-worker 连接到新的上游 MySQL 实例，需要按以下步骤进行操作：

1. 使用 `query-status` 命令获取当前 binlog replication 处理单元已复制到下游的 binlog 对应的 GTID sets (`syncerBinlogGtid`)，记为 `gtid-S`。
2. 在将要切换到的 MySQL 实例上使用 `SELECT @@GLOBAL.gtid_purged;` 获取已经被 purged 的 binlog 对应的 GTID sets，记为 `gtid-P`。
3. 在将要切换到的 MySQL 实例上使用 `SELECT @@GLOBAL.gtid_executed;` 获取所有已经执行成功的事务对应的 GTID sets，记为 `gtid-E`。
4. 确保满足以下关系，否则不支持将 DM-worker 连接切换到相应的 MySQL 实例：
    - `gtid-S` 包含 `gtid-P`（`gtid-P` 可以为空）
    - `gtid-E` 包含 `gtid-S`
5. 使用 `stop-task` 命令停止所有运行中的数据迁移任务。
6. 使用 `operate-source stop` 命令从 DM 集群中移除原 MySQL 实例地址对应的 source 配置。
7. 更新 source 配置文件中的 MySQL 实例地址并使用 `operate-source create` 命令将新的 source 配置重新加载到 DM 集群中。
8. 使用 `start-task` 命令重新启动数据迁移任务。

13.6.8.5 任务管理

13.6.8.5.1 使用 TiDB Data Migration 处理出错的 DDL 语句

本文介绍了如何使用 TiDB Data Migration (DM) 来处理出错的 DDL 语句。

目前，TiDB 并不完全兼容所有的 MySQL 语法（详见DDL 的限制）。当使用 DM 从 MySQL 迁移数据到 TiDB 时，如果 TiDB 不支持对应的 DDL 语句，可能会造成错误并中断迁移任务。在这种情况下，DM 提供 `binlog` 命令来恢复迁移。

使用限制

如果业务不能接受下游 TiDB 跳过异常的 DDL 语句，也不接受使用其他 DDL 语句作为替代，不接受插入其他 DDL 语句，则不适合使用此方式进行处理。

比如：`DROP PRIMARY KEY`，这种情况下，只能在下游重建一个（DDL 执行完后的）新表结构对应的表，并将原表的全部数据重新导入该新表。

支持场景

迁移过程中，上游执行了 TiDB 不支持的 DDL 语句并迁移到了 DM，造成迁移任务中断。

- 如果业务能接受下游 TiDB 不执行该 DDL 语句，则使用 `binlog skip <task-name>` 跳过对该 DDL 语句的迁移以恢复迁移任务。
- 如果业务能接受下游 TiDB 执行其他 DDL 语句来作为替代，则使用 `binlog replace <task-name>` 替代该 DDL 的迁移以恢复迁移任务。

- 如果业务能接受下游 TiDB 插入执行其他 DDL 语句，则使用 `binlog inject <task-name>` 插入其他 DDL 以恢复迁移任务。

命令介绍

使用 dmctl 手动处理出错的 DDL 语句时，主要使用的命令包括 `query-status`、`binlog`。

query-status

`query-status` 命令用于查询当前 MySQL 实例内子任务及 relay 单元等的状态和错误信息，详见查询状态。

binlog

`binlog` 命令管理和查看 binlog 操作。命令仅在 DM v6.0 及其以后版本支持，之前版本可使用 `handle-error` 命令。

`binlog` 命令用法如下：

```
binlog -h
```

```
manage or show binlog operations

Usage:
  dmctl binlog [command]

Available Commands:
  inject      inject the current error event or a specific binlog position (binlog-pos) with some
      ↪  ddls
  list        list error handle command at binlog position (binlog-pos) or after binlog position
      ↪ (binlog-pos)
  replace     replace the current error event or a specific binlog position (binlog-pos) with
      ↪ some ddls
  revert      revert the current binlog operation or a specific binlog position (binlog-pos)
      ↪ operation
  skip        skip the current error event or a specific binlog position (binlog-pos) event

Flags:
  -b, --binlog-pos string   position used to match binlog event if matched the binlog operation
      ↪ will be applied. The format like "mysql-bin|000001.000003:3270"
  -h, --help                help for binlog

Global Flags:
  -s, --source strings   MySQL Source ID.

Use "dmctl binlog [command] --help" for more information about a command.
```

`binlog` 支持如下子命令：

- `inject`：在 DDL binlog 位置插入 DDL 语句，binlog 位置指定方式参考 `-b, --binlog-pos`。

- `list`：查看 binlog 位置以及此位置之后的有效 inject/skip/replace 操作，binlog 位置指定方式参考 `-b, --binlog-pos`。
- `replace`：替代 DDL binlog 位置的 DDL 语句，binlog 位置指定方式参考 `-b, --binlog-pos`。
- `revert`：重置 binlog 位置的 inject/skip/replace 操作，仅在先前的操作没有最终生效前执行，binlog 位置指定方式参考 `-b, --binlog-pos`
- `skip`：跳过 binlog 位置的 DDL 语句, binlog 位置指定方式参考 `-b, --binlog-pos`。

`binlog` 支持如下参数：

- `-b, --binlog-pos`：
  - 类型：string。
  - 指定 binlog 位置，表示操作将在 `binlog-pos` 与 binlog event 的 position 匹配时生效。若不指定，DM 会默认置为当前出错的 DDL 语句的 binlog 位置。
  - 格式：`binlog-filename:binlog-pos`，例如 `mysql-bin|000001.000003:3270`。
  - 在迁移执行出错后，binlog position 可直接从 `query-status` 返回的 `startLocation` 中的 `position` 获得；在迁移执行出错前，binlog position 可在上游 MySQL 中使用 `SHOW BINLOG EVENTS` 获得。
- `-s, --source strings`：
  - 类型：string。
  - `source` 指定预设操作将生效的 MySQL 实例。
- 其他参数参考 `-h` 提示。

使用示例

迁移中断执行跳过操作

如需在迁移中断时执行跳过操作，可使用 `binlog skip` 命令：

```
binlog skip -h
```

```
skip the current error event or a specific binlog position (binlog-pos) event

Usage:
  dmctl binlog skip <task-name> [flags]

Flags:
  -h, --help   help for skip

Global Flags:
  -b, --binlog-pos string   position used to match binlog event if matched the binlog operation
      ↪ will be applied. The format like "mysql-bin|000001.000003:3270"
  -s, --source strings      MySQL Source ID.
```

非合库合表场景

假设现在需要将上游的 `db1.tbl1` 表迁移到下游 TiDB，初始时表结构为：

```
SHOW CREATE TABLE db1.tbl1;
```

```
+-------+-----------------------------------------------+
| Table | Create Table                                  |
+-------+-----------------------------------------------+
| tbl1  | CREATE TABLE `tbl1` (
  `c1` int(11) NOT NULL,
  `c2` decimal(11,3) DEFAULT NULL,
  PRIMARY KEY (`c1`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 |
+-------+-----------------------------------------------+
```

此时，上游执行以下 DDL 操作修改表结构（将列的 DECIMAL(11, 3) 修改为 DECIMAL(10, 3)）：

```
ALTER TABLE db1.tbl1 CHANGE c2 c2 DECIMAL (10, 3);
```

则会由于 TiDB 不支持该 DDL 语句而导致 DM 迁移任务中断，使用 `query-status <task-name>` 命令可看到如下错误：

```
ERROR 8200 (HY000): Unsupported modify column: can't change decimal column precision
```

假设业务上可以接受下游 TiDB 不执行此 DDL 语句（即继续保持原有的表结构），则可以通过使用 `binlog skip <task-name>` 命令跳过该 DDL 语句以恢复迁移任务。操作步骤如下：

1. 使用 `binlog skip <task-name>` 跳过当前错误的 DDL 语句

    ```
    » binlog skip test
    ```

    ```
    {
        "result": true,
        "msg": "",
        "sources": [
            {
                "result": true,
                "msg": "",
                "source": "mysql-replica-01",
                "worker": "worker1"
            }
        ]
    }
    ```

2. 使用 `query-status <task-name>` 查看任务状态

    ```
    » query-status test
    ```

    执行结果

```
{
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "",
            "sourceStatus": {
                "source": "mysql-replica-01",
                "worker": "worker1",
                "result": null,
                "relayStatus": null
            },
            "subTaskStatus": [
                {
                    "name": "test",
                    "stage": "Running",
                    "unit": "Sync",
                    "result": null,
                    "unresolvedDDLLockID": "",
                    "sync": {
                        "masterBinlog": "(DESKTOP-T561TSO-bin.000001, 2388)",
                        "masterBinlogGtid": "143bdef3-dd4a-11ea-8b00-00155de45f57:1-10",
                        "syncerBinlog": "(DESKTOP-T561TSO-bin.000001, 2388)",
                        "syncerBinlogGtid": "143bdef3-dd4a-11ea-8b00-00155de45f57:1-4",
                        "blockingDDLs": [
                        ],
                        "unresolvedGroups": [
                        ],
                        "synced": true,
                        "binlogType": "remote",
                        "totalRows": "4",
                        "totalRps": "0",
                        "recentRps": "0"
                    }
                }
            ]
        }
    ]
}
```


可以看到任务运行正常，错误的 DDL 被跳过。

合库合表场景

假设现在存在如下四个上游表需要合并迁移到下游的同一个表 `shard_db`.`shard_table`，任务模式为悲观协调模式：

- MySQL 实例 1 内有 shard_db_1 库，包括 shard_table_1 和 shard_table_2 两个表。
- MySQL 实例 2 内有 shard_db_2 库，包括 shard_table_1 和 shard_table_2 两个表。

初始时表结构为：

```
SHOW CREATE TABLE shard_db.shard_table;
```

```
+-------+-----------------------------------------------------------------------------------------------------------------
    ↪
| Table | Create Table
    ↪
    ↪ |
+-------+-----------------------------------------------------------------------------------------------------------------
    ↪
| tb    | CREATE TABLE `shard_table` (
  `id` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 COLLATE=latin1_bin |
+-------+-----------------------------------------------------------------------------------------------------------------
    ↪
```

此时，在上游所有分表上都执行以下 DDL 操作修改表字符集

```
ALTER TABLE `shard_db_*`.`shard_table_*` CHARACTER SET LATIN1 COLLATE LATIN1_DANISH_CI;
```

则会由于 TiDB 不支持该 DDL 语句而导致 DM 迁移任务中断，使用 query-status 命令可以看到 MySQL 实例 1 的 shard_db_1.shard_table_1 表和 MySQL 实例 2 的 shard_db_2.shard_table_1 表报错：

```
{
    "Message": "cannot track DDL: ALTER TABLE `shard_db_1`.`shard_table_1` CHARACTER SET UTF8
        ↪ COLLATE UTF8_UNICODE_CI",
    "RawCause": "[ddl:8200]Unsupported modify charset from latin1 to utf8"
}
```

```
{
    "Message": "cannot track DDL: ALTER TABLE `shard_db_2`.`shard_table_1` CHARACTER SET UTF8
        ↪ COLLATE UTF8_UNICODE_CI",
    "RawCause": "[ddl:8200]Unsupported modify charset from latin1 to utf8"
}
```

假设业务上可以接受下游 TiDB 不执行此 DDL 语句（即继续保持原有的表结构），则可以通过使用 binlog skip
↪ <task-name> 命令跳过该 DDL 语句以恢复迁移任务。操作步骤如下：

1. 使用 binlog skip <task-name> 跳过 MySQL 实例 1 和实例 2 当前错误的 DDL 语句

```
» binlog skip test
```


```
{
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-01",
            "worker": "worker1"
        },
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-02",
            "worker": "worker2"
        }
    ]
}
```


2. 使用 `query-status <task-name>` 查看任务状态，可以看到 MySQL 实例 1 的 `shard_db_1.shard_table_2` 表和 MySQL 实例 2 的 `shard_db_2.shard_table_2` 表报错：


```
{
    "Message": "cannot track DDL: ALTER TABLE `shard_db_1`.`shard_table_2` CHARACTER SET
        ↪ UTF8 COLLATE UTF8_UNICODE_CI",
    "RawCause": "[ddl:8200]Unsupported modify charset from latin1 to utf8"
}
```

```
{
    "Message": "cannot track DDL: ALTER TABLE `shard_db_2`.`shard_table_2` CHARACTER SET
        ↪ UTF8 COLLATE UTF8_UNICODE_CI",
    "RawCause": "[ddl:8200]Unsupported modify charset from latin1 to utf8"
}
```


3. 继续使用 `binlog skip <task-name>` 跳过 MySQL 实例 1 和实例 2 当前错误的 DDL 语句

```
» binlog skip test
```


```
{
    "result": true,
    "msg": "",
    "sources": [
        {
```

```
            "result": true,
            "msg": "",
            "source": "mysql-replica-01",
            "worker": "worker1"
        },
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-02",
            "worker": "worker2"
        }
    ]
}
```

4. 使用 `query-status <task-name>` 查看任务状态

```
» query-status test
```

执行结果

```
{
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "",
            "sourceStatus": {
                "source": "mysql-replica-01",
                "worker": "worker1",
                "result": null,
                "relayStatus": null
            },
            "subTaskStatus": [
                {
                    "name": "test",
                    "stage": "Running",
                    "unit": "Sync",
                    "result": null,
                    "unresolvedDDLLockID": "",
                    "sync": {
                        "masterBinlog": "(DESKTOP-T561TSO-bin.000001, 2388)",
                        "masterBinlogGtid": "143bdef3-dd4a-11ea-8b00-00155de45f57:1-10",
                        "syncerBinlog": "(DESKTOP-T561TSO-bin.000001, 2388)",
                        "syncerBinlogGtid": "143bdef3-dd4a-11ea-8b00-00155de45f57:1-4",
                        "blockingDDLs": [
```

```
                    ],
                    "unresolvedGroups": [
                    ],
                    "synced": true,
                    "binlogType": "remote",
                    "totalRows": "4",
                    "totalRps": "0",
                    "recentRps": "0"
                }
            }
        ]
    },
    {
        "result": true,
        "msg": "",
        "sourceStatus": {
            "source": "mysql-replica-02",
            "worker": "worker2",
            "result": null,
            "relayStatus": null
        },
        "subTaskStatus": [
            {
                "name": "test",
                "stage": "Running",
                "unit": "Sync",
                "result": null,
                "unresolvedDDLLockID": "",
                "sync": {
                    "masterBinlog": "(DESKTOP-T561TSO-bin.000001, 2388)",
                    "masterBinlogGtid": "143bdef3-dd4a-11ea-8b00-00155de45f57:1-10",
                    "syncerBinlog": "(DESKTOP-T561TSO-bin.000001, 2388)",
                    "syncerBinlogGtid": "143bdef3-dd4a-11ea-8b00-00155de45f57:1-4",
                    "blockingDDLs": [
                    ],
                    "unresolvedGroups": [
                    ],
                    "synced": true,
                    "binlogType": "remote",
                    "totalRows": "4",
                    "totalRps": "0",
                    "recentRps": "0"
                }
            }
        ]
```

```
        }
    ]
}
```

可以看到任务运行正常，无错误信息。四条 DDL 全部被跳过。

迁移中断执行替代操作

如需在迁移中断时执行替代操作，可使用 `binlog replace` 命令：

```
binlog replace -h
```

```
replace the current error event or a specific binlog position (binlog-pos) with some ddls

Usage:
  dmctl binlog replace <task-name> <replace-sql1> <replace-sql2>... [flags]

Flags:
  -h, --help   help for replace

Global Flags:
  -b, --binlog-pos string   position used to match binlog event if matched the binlog operation
      ↪ will be applied. The format like "mysql-bin|000001.000003:3270"
  -s, --source strings      MySQL Source ID.
```

非合库合表场景

假设现在需要将上游的 `db1.tbl1` 表迁移到下游 TiDB，初始时表结构为：

```
SHOW CREATE TABLE db1.tbl1;
```

```
+-------+-------------------------------------------------------------------------------------------------------------
    ↪
| Table | Create Table
    ↪
    ↪ |
+-------+-------------------------------------------------------------------------------------------------------------
    ↪
| tb    | CREATE TABLE `tbl1` (
  `id` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 COLLATE=latin1_bin |
+-------+-------------------------------------------------------------------------------------------------------------
    ↪
```

此时，上游执行以下 DDL 操作增加新列，并添加 UNIQUE 约束

```
ALTER TABLE `db1`.`tbl1` ADD COLUMN new_col INT UNIQUE;
```

则会由于 TiDB 不支持该 DDL 语句而导致 DM 迁移任务中断，使用 query-status 命令可看到如下错误：

```
{
    "Message": "cannot track DDL: ALTER TABLE `db1`.`tbl1` ADD COLUMN `new_col` INT UNIQUE KEY",
    "RawCause": "[ddl:8200]unsupported add column 'new_col' constraint UNIQUE KEY when altering '
        ↪ db1.tbl1'",
}
```

我们将该 DDL 替换成两条等价的 DDL。操作步骤如下：

1. 使用如下命令替换错误的 DDL 语句

```
» binlog replace test "ALTER TABLE `db1`.`tbl1` ADD COLUMN `new_col` INT;ALTER TABLE `db1`.`
    ↪ tbl1` ADD UNIQUE(`new_col`)";
```

```
{
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-01",
            "worker": "worker1"
        }
    ]
}
```

2. 使用 query-status <task-name> 查看任务状态

```
» query-status test
```

执行结果

```
{
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "",
            "sourceStatus": {
                "source": "mysql-replica-01",
                "worker": "worker1",
                "result": null,
                "relayStatus": null
            },
```

```
            "subTaskStatus": [
                {
                    "name": "test",
                    "stage": "Running",
                    "unit": "Sync",
                    "result": null,
                    "unresolvedDDLLockID": "",
                    "sync": {
                        "masterBinlog": "(DESKTOP-T561TSO-bin.000001, 2388)",
                        "masterBinlogGtid": "143bdef3-dd4a-11ea-8b00-00155de45f57:1-10",
                        "syncerBinlog": "(DESKTOP-T561TSO-bin.000001, 2388)",
                        "syncerBinlogGtid": "143bdef3-dd4a-11ea-8b00-00155de45f57:1-4",
                        "blockingDDLs": [
                        ],
                        "unresolvedGroups": [
                        ],
                        "synced": true,
                        "binlogType": "remote",
                        "totalRows": "4",
                        "totalRps": "0",
                        "recentRps": "0"
                    }
                }
            ]
        }
    ]
}
```


可以看到任务运行正常，错误的 DDL 已被替换且执行成功。

合库合表场景

假设现在存在如下四个上游表需要合并迁移到下游的同一个表 `shard_db`.`shard_table`，任务模式为悲观协调模式：

- MySQL 实例 1 内有 shard_db_1 库，包括 shard_table_1 和 shard_table_2 两个表。
- MySQL 实例 2 内有 shard_db_2 库，包括 shard_table_1 和 shard_table_2 两个表。

初始时表结构为：

```
SHOW CREATE TABLE shard_db.shard_table;
```

```
+-------+-----------------------------------------------------------------------------------------------------------------
    ↪
| Table | Create Table
    ↪
    ↪ |
```

```
+-------+-----------------------
```

```
            "worker": "worker1"
        }
    ]
}
```

```
» binlog replace test -s mysql-replica-02 "ALTER TABLE `shard_db_2`.`shard_table_1` ADD
    ↪ COLUMN `new_col` INT;ALTER TABLE `shard_db_2`.`shard_table_1` ADD UNIQUE(`new_col`)"
    ↪ ;
```

```
{
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-02",
            "worker": "worker2"
        }
    ]
}
```

2. 使用 `query-status <task-name>` 查看任务状态，可以看到 MySQL 实例 1 的 `shard_db_1.shard_table_2` 表和 MySQL 实例 2 的 `shard_db_2.shard_table_2` 表报错：

```
{
    "Message": "detect inconsistent DDL sequence from source ... ddls: [ALTER TABLE `
        ↪ shard_db`.`tb` ADD COLUMN `new_col` INT UNIQUE KEY] source: `shard_db_1`.`
        ↪ shard_table_2`], right DDL sequence should be ..."
}
```

```
{
    "Message": "detect inconsistent DDL sequence from source ... ddls: [ALTER TABLE `
        ↪ shard_db`.`tb` ADD COLUMN `new_col` INT UNIQUE KEY] source: `shard_db_2`.`
        ↪ shard_table_2`], right DDL sequence should be ..."
}
```

3. 使用如下命令继续分别替换 MySQL 实例 1 和实例 2 中错误的 DDL 语句

```
» binlog replace test -s mysql-replica-01 "ALTER TABLE `shard_db_1`.`shard_table_2` ADD
    ↪ COLUMN `new_col` INT;ALTER TABLE `shard_db_1`.`shard_table_2` ADD UNIQUE(`new_col`)"
    ↪ ;
```

```
{
    "result": true,
    "msg": "",
```

```
    "sources": [
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-01",
            "worker": "worker1"
        }
    ]
}
```

```
» binlog replace test -s mysql-replica-02 "ALTER TABLE `shard_db_2`.`shard_table_2` ADD
    ↪ COLUMN `new_col` INT;ALTER TABLE `shard_db_2`.`shard_table_2` ADD UNIQUE(`new_col`)"
    ↪ ;
```

```
{
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-02",
            "worker": "worker2"
        }
    ]
}
```

4. 使用 `query-status <task-name>` 查看任务状态

```
» query-status test
```

执行结果

```
{
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "",
            "sourceStatus": {
                "source": "mysql-replica-01",
                "worker": "worker1",
                "result": null,
                "relayStatus": null
            },
```

```
        "subTaskStatus": [
            {
                "name": "test",
                "stage": "Running",
                "unit": "Sync",
                "result": null,
                "unresolvedDDLLockID": "",
                "sync": {
                    "masterBinlog": "(DESKTOP-T561TSO-bin.000001, 2388)",
                    "masterBinlogGtid": "143bdef3-dd4a-11ea-8b00-00155de45f57:1-10",
                    "syncerBinlog": "(DESKTOP-T561TSO-bin.000001, 2388)",
                    "syncerBinlogGtid": "143bdef3-dd4a-11ea-8b00-00155de45f57:1-4",
                    "blockingDDLs": [
                    ],
                    "unresolvedGroups": [
                    ],
                    "unresolvedGroups": [
                    ],
                    "synced": true,
                    "binlogType": "remote",
                    "totalRows": "4",
                    "totalRps": "0",
                    "recentRps": "0"
                }
            }
        ]
    },
    {
        "result": true,
        "msg": "",
        "sourceStatus": {
            "source": "mysql-replica-02",
            "worker": "worker2",
            "result": null,
            "relayStatus": null
        },
        "subTaskStatus": [
            {
                "name": "test",
                "stage": "Running",
                "unit": "Sync",
                "result": null,
                "unresolvedDDLLockID": "",
                "sync": {
                    "masterBinlog": "(DESKTOP-T561TSO-bin.000001, 2388)",
```

```
                    "masterBinlogGtid": "143bdef3-dd4a-11ea-8b00-00155de45f57:1-10",
                    "syncerBinlog": "(DESKTOP-T561TSO-bin.000001, 2388)",
                    "syncerBinlogGtid": "143bdef3-dd4a-11ea-8b00-00155de45f57:1-4",
                    "blockingDDLs": [
                    ],
                    "unresolvedGroups": [
                    ],
                    "unresolvedGroups": [
                    ],
                    "synced": try,
                    "binlogType": "remote",
                    "totalRows": "4",
                    "totalRps": "0",
                    "recentRps": "0"
                }
            }
        ]
    }
]
}
```

可以看到任务运行正常，无错误信息。四条 DDL 全部被替换。

其他命令

binlog 其他命令的使用，请参考上述 `binlog skip`、`binlog replace` 命令的使用方式。

13.6.8.5.2 管理 TiDB Data Migration 迁移表的表结构

dmctl 是运维 TiDB Data Migration (DM) 集群的命令行工具，本文介绍如何使用 dmctl 组件来管理通过 DM 迁移的表在 DM 内部的表结构。

DM 执行增量迁移时，首先读取上游的 binlog，然后生成 SQL 语句执行到下游。但是，上游的 binlog 中并不记录表的完整结构信息，为了生成 SQL 语句，DM 内部维护了待迁移的表的 schema 信息，即表结构信息。

为了一些特殊场景及处理其他可能的由于 schema 不匹配导致的迁移中断等问题，DM 提供了 `binlog-schema` 命令来获取、修改、删除 DM 内部维护的表结构。

原理介绍

DM 中的 schema 信息来源包括以下几部分：

1. 执行全量数据迁移 (`task-mode=all`) 时，任务将经过 dump/load/sync（全量导出/全量导入/增量同步）三个阶段。dump 阶段将表结构信息随着数据一并导出，并自动在下游创建相关表。sync 阶段时以该表结构作为起始的表结构信息。
2. sync 阶段中，处理 DDL 语句（如 `ALTER TABLE`）时，DM 执行该语句的同时更新内部维护的表结构信息。
3. 如果任务是增量迁移 (`task-mode=incremental`)，下游已经将待迁移的表创建完成，DM 会从下游数据库获取该表的结构信息（此行为随版本变化而不同）。

增量迁移过程的 schema 信息维护较为复杂，在整个数据链路中包含以下几类可能相同或不同的表结构。

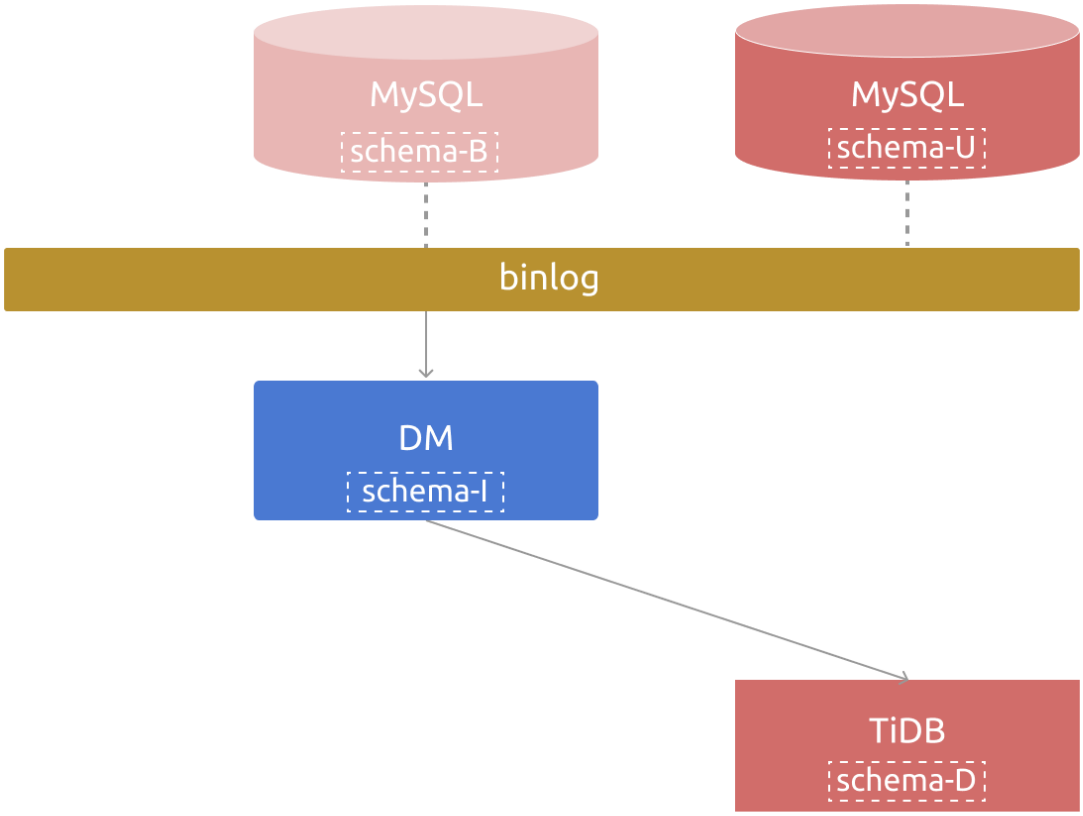


图 245: 表结构

- 上游当前时刻的表结构（记为 schema-U）。
- 当前 DM 正在消费的 binlog event 的表结构（记为 schema-B，其对应于上游某个历史时刻的表结构）。
- DM 内部（schema tracker 组件）当前维护的表结构（记为 schema-I）。
- 下游 TiDB 集群中的表结构（记为 schema-D）。

在大多数情况下，以上 4 类表结构一致。当上游执行 DDL 变更表结构后，schema-U 即会发生变更；DM 通过将该 DDL 应用于内部的 schema tracker 组件及下游 TiDB，会先后更新 schema-I、schema-D 以与 schema-U 保持一致，因而随后能正常消费 binlog 中在 DDL 之后对应表结构为 schema-B 的 binlog event。即当 DDL 被复制成功后，仍能保持 schema-U、schema-B、schema-I 及 schema-D 的一致。

需要注意以下不一致的情况：

- 在开启乐观 shard DDL 支持的数据迁移过程中，下游合并表的 schema-D 可能与部分分表对应的 schema-B 及 schema-I 并不一致，但 DM 仍保持 schema-I 与 schema-B 的一致，以确保能正常解析 DML 对应的 binlog event。

- 当下游比上游多部分列时，schema-D 也可能会与 schema-B 及 schema-I 并不一致。全量数据迁移 (task-mode=all) 时 DM 会自动处理不一致的情况；而增量迁移 (task-mode=incremental) 时，由于任务首次启动，内部尚无 Schema 信息，DM 将自动读取下游表结构，即 schema-D，更新自身的 schema-I (此行为随版本变化而不同)。在此之后，如果 DM 使用 schema-I 解析 schema-B 的 binlog，将会导致 Column count doesn't match value count 错误，详情可参考：下游存在更多列的迁移场景

binlog-schema 命令可以获取、修改、删除 DM 内部维护的表结构 schema-I。

> **注意：**
>
> binlog-schema 命令仅在 DM v6.0 及其以后版本支持，之前版本可使用 operate-schema 命令。

命令介绍

```
help binlog-schema
```

```
manage or show table schema in schema tracker

Usage:
  dmctl binlog-schema [command]

Available Commands:
  delete      delete table schema structure
  list        show table schema structure
  update      update tables schema structure

Flags:
  -h, --help   help for binlog-schema

Global Flags:
  -s, --source strings   MySQL Source ID.

Use "dmctl binlog-schema [command] --help" for more information about a command.
```

> **注意：**
>
> - 由于表结构在数据迁移过程中可能会发生变更，为获取确定性的表结构，当前 binlog-schema 命令仅能在数据迁移任务处于 Paused 状态时可用。
> - 强烈建议在修改表结构前，首先获取并备份表结构，以免误操作导致数据丢失。

参数解释

- `delete`：删除表结构
- `list`：获取查看表结构
- `update`：更新设置表结构
- `-s` 或 `--source`:
  - 必选
  - 指定操作将应用到的 MySQL 源

使用示例

获取表结构

如需获取表结构，可使用 `binlog-schema list` 命令：

```
help binlog-schema list
```

```
show table schema structure

Usage:
  dmctl binlog-schema list <task-name> <database> <table> [flags]

Flags:
  -h, --help   help for list

Global Flags:
  -s, --source strings   MySQL Source ID.
```

假设要获取 `db_single` 任务对应于 `mysql-replica-01` MySQL 源的 `` `db_single`.`t1` `` 表的表结构，则执行如下命令：

```
binlog-schema list -s mysql-replica-01 task_single db_single t1
```

```
{
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "CREATE TABLE `t1` ( `c1` int(11) NOT NULL, `c2` int(11) DEFAULT NULL, PRIMARY
                ↪  KEY (`c1`)) ENGINE=InnoDB DEFAULT CHARSET=latin1 COLLATE=latin1_bin",
            "source": "mysql-replica-01",
            "worker": "127.0.0.1:8262"
        }
    ]
}
```

更新设置表结构

如需更新设置表结构，可使用 `binlog-schema update` 命令：

```
help binlog-schema update
```

```
update tables schema structure

Usage:
  dmctl binlog-schema update <task-name> <database> <table> [schema-file] [flags]

Flags:
      --flush         flush the table info and checkpoint immediately (default true)
      --from-source   use the schema from upstream database as the schema of the specified tables
      --from-target   use the schema from downstream database as the schema of the specified
          ↪ tables
  -h, --help          help for update
      --sync          sync the table info to master to resolve shard ddl lock, only for
          ↪ optimistic mode now (default true)

Global Flags:
  -s, --source strings   MySQL Source ID.
```

假设要设置 db_single 任务对应于 mysql-replica-01 MySQL 源的 `db_single`.`t1` 表的表结构为如下所示：

```
CREATE TABLE `t1` (
    `c1` int(11) NOT NULL,
    `c2` bigint(11) DEFAULT NULL,
    PRIMARY KEY (`c1`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1 COLLATE=latin1_bin
```

则将上述 CREATE TABLE 语句保存为文件（如 db_single.t1-schema.sql）后，执行如下命令：

```
binlog-schema update -s mysql-replica-01 task_single db_single t1 db_single.t1-schema.sql
```

```
{
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-01",
            "worker": "127.0.0.1:8262"
        }
    ]
}
```

删除表结构

如需删除表结构，可使用 binlog-schema delete 命令：

```
help binlog-schema delete
```

```
delete table schema structure

Usage:
  dmctl binlog-schema delete <task-name> <database> <table> [flags]

Flags:
  -h, --help   help for delete

Global Flags:
  -s, --source strings   MySQL Source ID.
```

> **注意：**
>
> 删除 DM 内部维护的表结构后，如果后续有该表的 DDL/DML 需要复制到下游，则 DM 会依次尝试从 checkpoint 表里 `table_info` 字段、乐观 shard DDL 协调中的元信息以及下游 TiDB 中对应的该表获取表结构。

假设要删除 `db_single` 任务对应于 `mysql-replica-01` MySQL 源的 `` `db_single`.`t1` `` 表的表结构，则执行如下命令：

```
binlog-schema delete -s mysql-replica-01 task_single db_single t1
```

```
{
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-01",
            "worker": "127.0.0.1:8262"
        }
    ]
}
```

13.6.8.6 TiDB Data Migration 导出和导入集群的数据源和任务配置

`config` 命令用于导出和导入集群的数据源和任务配置。

> **注意：**
>
> 对于 v2.0.5 版本之前的集群，可使用 v2.0.5 版本及之后的 dmctl 导出和导入集群的数据源和任务配置文件。

```
» help config
Commands to import/export config

Usage:
  dmctl config [command]

Available Commands:
  export      Export the configurations of sources and tasks.
  import      Import the configurations of sources and tasks.

Flags:
  -h, --help   help for config

Global Flags:
  -s, --source strings   MySQL Source ID.

Use "dmctl config [command] --help" for more information about a command.
```

13.6.8.6.1 导出集群的数据源和任务配置

使用 `export` 子命令导出集群的数据源和任务配置到指定文件夹中。

```
config export [--dir directory]
```

参数解释

- `dir`：
  - 可选
  - 指定导出文件夹路径
  - 默认值为 `./configs`

返回结果示例

```
config export -d /tmp/configs
```

```
export configs to directory `/tmp/configs` succeed
```

13.6.8.6.2 导入集群的数据源和任务配置

使用 `import` 子命令从指定文件夹中导入集群的数据源和任务配置。

```
config import [--dir directory]
```

> **注意：**
>
> 对于 v2.0.2 版本之后的集群，暂不支持自动导入 relay worker 的相关配置，可以手动使用 `start-relay` 命令开启 relay log。

参数解释

- `dir`：
  - 可选
  - 指定导入文件夹路径
  - 默认值为 `./configs`

返回结果示例

```
config import -d /tmp/configs
```

```
start creating sources
start creating tasks
import configs from directory `/tmp/configs` succeed
```

13.6.8.7 TiDB Data Migration 处理告警

本文档介绍如何处理 TiDB Data Migration (DM) 中的告警。

13.6.8.7.1 高可用告警

`DM_master_all_down`

当全部 DM-master 离线时触发该告警。发生该错误时，需要检查集群环境，并通过各节点日志排查错误。

`DM_worker_offline`

存在离线的 DM-worker 超过一小时会触发该告警。在高可用架构下，该告警可能不会直接中断任务，但是会提升任务中断的风险。处理告警可以查看对应 DM-worker 节点的工作状态，检查是否连通，并通过日志排查错误。

`DM_DDL_error`

处理 shard DDL 时出现错误，此时需要参考DM 故障诊断进行处理。

`DM_pending_DDL`

存在未完成的 shard DDL 并超过一小时会触发该告警。在某些应用场景下，存在未完成的 shard DDL 可能是用户所期望的。在用户预期以外的场景下，可以通过手动处理 Sharding DDL Lock解决。

13.6.8.7.2 任务状态告警

`DM_task_state`

当 DM-worker 内有子任务处于 Paused 状态超过 20 分钟时会触发该告警，此时需要参考DM 故障诊断进行处理。

13.6.8.7.3 relay log 告警

`DM_relay_process_exits_with_error`

当 relay log 处理单元遇到无法自动恢复的错误时（如找不到 binlog 文件），或者短时间内多次遇到（如 2 分钟内遇到 3 次以上）可自动恢复的错误时（如网络问题），会触发该告警，此时需要参考DM 故障诊断进行处理。

`DM_remain_storage_of_relay_log`

当 relay log 所在磁盘的剩余可用容量小于 10G 时会触发该告警，对应的处理方法包括：

- 手动清理该磁盘上其他无用数据以增加可用容量。
- 尝试调整 relay log 的自动清理策略或执行手动清理。
- 使用 `pause-relay` 命令暂停 relay log 的拉取，并在磁盘空间合适之后使用 `resume-relay` 命令恢复。需要注意上游数据源不要清理尚未拉取的 binlog。

`DM_relay_log_data_corruption`

当 relay log 处理单元在校验从上游读取到的 binlog event 且发现 checksum 信息异常时会转为 Paused 状态并立即触发告警，此时需要参考DM 故障诊断进行处理。

`DM_fail_to_read_binlog_from_master`

当 relay log 处理单元在尝试从上游读取 binlog event 发生错误时，会转为 Paused 状态并立即触发该告警，此时需要参考DM 故障诊断进行处理。

`DM_fail_to_write_relay_log`

当 relay log 处理单元在尝试将 binlog event 写入 relay log 文件发生错误时，会转为 Paused 状态并立即触发该告警，此时需要参考DM 故障诊断进行处理。

`DM_binlog_file_gap_between_master_relay`

当 relay log 处理单元已拉取到的最新的 binlog 文件个数落后于当前上游 MySQL/MariaDB 超过 1 个（不含 1 个）且持续 10 分钟时会触发该告警，此时需要参考性能问题及处理方法对 relay log 处理单元相关的性能问题进行排查与处理。

13.6.8.7.4 Dump/Load 告警

`DM_dump_process_exists_with_error`

当 Dump 处理单元遇到无法自动恢复的错误时（如找不到 binlog 文件），或者短时间内多次遇到（如 2 分钟内遇到 3 次以上）可自动恢复的错误时（如网络问题），会触发该告警，此时需要参考DM 故障诊断进行处理。

`DM_load_process_exists_with_error`

当 Load 处理单元遇到无法自动恢复的错误时（如找不到 binlog 文件），或者短时间内多次遇到（如 2 分钟内遇到 3 次以上）可自动恢复的错误时（如网络问题），会触发该告警，此时需要参考DM 故障诊断进行处理。

13.6.8.7.5 Binlog replication 告警

`DM_sync_process_exists_with_error`

当 Binlog replication 处理单元遇到无法自动恢复的错误时（如找不到 binlog 文件），或者短时间内多次遇到（如 2 分钟内遇到 3 次以上）可自动恢复的错误时（如网络问题），会触发该告警，此时需要参考DM 故障诊断进行处理。

`DM_binlog_file_gap_between_master_syncer`

当 Binlog replication 处理单元已处理到的最新的 binlog 文件个数落后于当前上游 MySQL/MariaDB 超过 1 个（不含 1 个）且持续 10 分钟时 DM 会触发该告警，此时需要参考性能问题及处理方法对 Binlog replication 处理单元相关的性能问题进行排查与处理。

`DM_binlog_file_gap_between_relay_syncer`

当 Binlog replication 处理单元已处理到的最新的 binlog 文件个数落后于当前 relay log 处理单元超过 1 个（不含 1 个）且持续 10 分钟时 DM 会触发该告警，此时需要参考性能问题及处理方法对 Binlog replication 处理单元相关的性能问题进行排查与处理。

13.6.8.8 TiDB Data Migration 日常巡检

本文总结了 TiDB Data Migration (DM) 工具日常巡检的方法：

- 方法一：执行 `query-status` 命令查看任务运行状态以及相关错误输出。详见查询状态。
- 方法二：如果使用 TiUP 部署 DM 集群时正确部署了 Prometheus 与 Grafana，如 Grafana 的地址为 `172.16.10.71` ↪ ，可在浏览器中打开 http://172.16.10.71:3000 进入 Grafana，选择 DM 的 Dashboard 即可查看 DM 相关监控项。具体监控指标参照监控与告警设置。
- 方法三：通过日志文件查看 DM 运行状态和相关错误。
  - DM-master 日志目录：通过 DM-master 进程参数 `--log-file` 设置。如果使用 TiUP 部署 DM，则日志目录位于 `{log_dir}`。
  - DM-worker 日志目录：通过 DM-worker 进程参数 `--log-file` 设置。如果使用 TiUP 部署 DM，则日志目录位于 `{log_dir}`。

13.6.9 参考手册

13.6.9.1 架构组件

13.6.9.1.1 DM-worker 简介

DM-worker 是 DM (Data Migration) 的一个组件，负责执行具体的数据迁移任务。

其主要功能如下：

- 注册为一台 MySQL 或 MariaDB 服务器的 slave。
- 读取 MySQL 或 MariaDB 的 binlog event，并将这些 event 持久化保存在本地 (relay log)。
- 单个 DM-worker 支持迁移一个 MySQL 或 MariaDB 实例的数据到下游的多个 TiDB 实例。
- 多个 DM-Worker 支持迁移多个 MySQL 或 MariaDB 实例的数据到下游的一个 TiDB 实例。

DM-worker 处理单元

DM-worker 任务包含如下多个逻辑处理单元。

Relay log

Relay log 持久化保存从上游 MySQL 或 MariaDB 读取的 binlog，并对 binlog replication 处理单元提供读取 binlog event 的功能。

其原理和功能与 MySQL relay log 类似，详见 MySQL Relay Log。

dump 处理单元

dump 处理单元从上游 MySQL 或 MariaDB 导出全量数据到本地磁盘。

load 处理单元

load 处理单元读取 dump 处理单元导出的数据文件，然后加载到下游 TiDB。

Binlog replication/sync 处理单元

Binlog replication/sync 处理单元读取上游 MySQL/MariaDB 的 binlog event 或 relay log 处理单元的 binlog event，将这些 event 转化为 SQL 语句，再将这些 SQL 语句应用到下游 TiDB。

DM-worker 所需权限

本小节主要介绍使用 DM-worker 时所需的上下游数据库用户权限以及各处理单元所需的用户权限。

上游数据库用户权限

上游数据库 (MySQL/MariaDB) 用户必须拥有以下权限：

权限	作用域
SELECT	Tables
RELOAD	Global
REPLICATION SLAVE	Global
REPLICATION CLIENT	Global

如果要迁移 db1 的数据到 TiDB，可执行如下的 GRANT 语句：

```
GRANT RELOAD,REPLICATION SLAVE, REPLICATION CLIENT ON *.* TO 'your_user'@'your_wildcard_of_host'
GRANT SELECT ON db1.* TO 'your_user'@'your_wildcard_of_host';
```

如果还要迁移其他数据库的数据到 TiDB，请确保已赋予这些库跟 db1 一样的权限。

下游数据库用户权限

下游数据库 (TiDB) 用户必须拥有以下权限：

权限	作用域
SELECT	Tables
INSERT	Tables
UPDATE	Tables
DELETE	Tables

权限	作用域
CREATE	Databases，tables
DROP	Databases，tables
ALTER	Tables
INDEX	Tables

对要执行迁移操作的数据库或表执行下面的 GRANT 语句：

```
GRANT SELECT,INSERT,UPDATE,DELETE,CREATE,DROP,ALTER,INDEX  ON db.table TO 'your_user'@'
    ↪ your_wildcard_of_host';
GRANT ALL ON dm_meta.* TO 'your_user'@'your_wildcard_of_host';
```

处理单元所需的最小权限

处理单元	最小上游 (MySQL/MariaDB) 权限	最小下游 (TiDB) 权限	最小系统权限
Relay log	REPLICATION SLAVE (读取 binlog) REPLICATION CLIENT (show master status, show slave status)	无	本地读/写磁盘
Dump	SELECTRELOAD (获取读锁将表数据刷到磁盘，进行一些操作后，再释放读锁对表进行解锁)	无	本地写磁盘
Load	无	SELECT (查询 checkpoint 历史) CREATE (创建数据库或表) DELETE (删除 checkpoint) INSERT (插入 dump 数据)	读/写本地文件
Binlog replica-tion	REPLICATION SLAVE (读 binlog) REPLICATION CLIENT (show master status, show slave status)	SELECT (显示索引和列) INSERT (DML)UPDATE (DML)DELETE (DML)CREATE (创建数据库或表) DROP (删除数据库或表) ALTER (修改表) INDEX (创建或删除索引)	本地读/写磁盘

> 注意：
>
> 这些权限并非一成不变。随着需求改变，这些权限也可能会改变。

13.6.9.1.2 DM 安全模式

安全模式 (safe mode) 是 DM 在进行增量同步时候的一种运行模式，在安全模式中，DM 增量同步组件在同步 binlog event 时，将把所有 INSERT 和 UPDATE 操作强制进行改写后再在下游执行。

安全模式的目的是在增量同步过程中，同一条 binlog event 能够在下游被重复同步且保证幂等性，从而确保增量同步能够“安全”进行。

DM 从 checkpoint 恢复数据同步任务后，可能重复执行某些 binlog 事件而导致下述问题：

1. 在进行增量同步过程中，执行 DML 的操作和写 checkpoint 的操作并不是同步的；写 checkpoint 的操作和写下游数据的操作也并不能保证原子性。因此，当 DM 异常退出时，checkpoint 可能只记录到退出时刻之前的一个恢复点。
2. 当 DM 重启同步任务，并从 checkpoint 重新开始增量数据同步时，checkpoint 之后的部分数据可能已经在异常退出前被处理过了，从而导致部分 SQL 语句重复执行。
3. 如果重复执行 INSERT 操作，会导致主键或唯一索引冲突，引发同步中断；如果重复执行 UPDATE 操作，会导致不能根据筛选条件找到之前对应的更新记录。

在安全模式下，通过改写 SQL 语句，DM 可以解决上述问题。

安全模式原理

安全模式通过 SQL 语句改写来保证 binlog event 的幂等性。具体来说，在安全模式下：

- INSERT 语句会被改写成 REPLACE 语句。
- UPDATE 语句会被分析，得到该语句涉及的行的主键或唯一索引的值，然后改写成 DELETE + REPLACE 语句：先根据主键或唯一索引的定位删除对应的行，然后使用 REPLACE 语句插入一条最新值的行记录。

REPLACE 操作是 MySQL 特有的数据插入语法。使用 REPLACE 语法插入数据时，如果新插入的数据和现有数据存在主键或唯一约束冲突，MySQL 会删除所有冲突的记录，然后再执行插入记录操作，相当于“强制插入”的操作。具体请参考 MySQL 官方文档的 REPLACE 语句相关介绍。

比如，假设一张表 dummydb.dummytbl 的主键是 id，在这张表中重复执行下面的 SQL 语句：

```
INSERT INTO dummydb.dummytbl (id, int_value, str_value) VALUES (123, 999, 'abc');
UPDATE dummydb.dummytbl SET int_value = 888999 WHERE int_value = 999;    # 假设没有其他 int_value
    ↪  = 999的数据
UPDATE dummydb.dummytbl SET id = 999 WHERE id = 888;     # 更新主键操作
```

启用安全模式后，上述 SQL 语句再次在下游执行时，会被改写为下面的 SQL 语句：

```
REPLACE INTO dummydb.dummytbl (id, int_value, str_value) VALUES (123, 999, 'abc');
DELETE FROM dummydb.dummytbl WHERE id = 123;
REPLACE INTO dummydb.dummytbl (id, int_value, str_value) VALUES (123, 888999, 'abc');
DELETE FROM dummydb.dummytbl WHERE id = 888;
REPLACE INTO dummydb.dummytbl (id, int_value, str_value) VALUES (999, 888888, 'abc888');
```

上述语句中，UPDATE 被改写为 DELETE + REPLACE，而不是 DELETE + INSERT。如果这里使用 INSERT，那么 id = 999 的记录在已经存在的情况下重复插入时，数据库会报错主键冲突。因此这里使用了 REPLACE，新的记录会替换之前已经插入的记录。

通过这样的语句改写，在进行重复的插入或更新操作时，DM 都会使用执行该操作后的行数据来覆盖之前已经存在的行数据。这样保证了插入和更新操作的可重复执行。

开启安全模式

自动开启

当 DM 从 checkpoint 恢复增量同步任务（例如 worker 重启，网络中断重连等）时，会自动开启一段时间的安全模式。开启安全模式的逻辑，与 checkpoint 中的 safemode_exit_point 信息相关。当一个增量同步任务异常暂停时，DM 会先尝试将内存中所有的 DML 全部同步到下游，然后记录当前内存中从上游拉取到的最新的 binlog 位置点，记作 safemode_exit_point，把 safemode_exit_point 保存在异常暂停之前最后一个 checkpoint 当中。

当 DM 从 checkpoint 恢复增量同步任务时，会根据下面的判断逻辑来开启安全模式：

- 如果 checkpoint 信息里面额外记录了 safemode_exit_point，说明该增量同步任务属于异常暂停。如果 DM 在恢复该增量同步任务时，发现待恢复的 checkpoint 的 binlog 位置点在 safemode_exit_point 之前，那么说明从这个 checkpoint 到 safemode_exit_point 之间的 binlog 可能已经在下游被执行；恢复该任务后，部分 binlog 会被重复执行。因此，DM 判断在这段 binlog 范围内需要开启安全模式。在当前的 binlog 位置点超过了 safemode_exit_point 后，未手动开启 safe-mode 的情况下，DM 将关闭安全模式。
- 如果 checkpoint 信息里没有 safemode_exit_point 信息，则有两种情况：

    1. 这是一个全新的任务，或该任务是正常暂停。
    2. 该任务异常暂停时，记录 safemode_exit_point 失败；或是 DM 进程异常退出。

    第二种情况下，DM 并不知道哪些 checkpoint 之后的 binlog 已经被执行过。为了保险起见，只要没有在 checkpoint 找到 safemode_exit_point 信息，DM 就会在前 2 个 checkpoint 间隔中都开启安全模式，确保这段时间里的 binlog 被重复执行不会引发问题。默认的 checkpoint 间隔是 30 秒，也就是说，一个正常的增量同步任务开始时，前 2 * 30 = 60 秒会自动强制开启安全模式。

如果你需要调整增量同步任务开始时安全模式的持续时间，可以设置 syncers 的配置项safe-mode-duration。

手动开启

你可以通过调整增量任务配置文件中 syncer 的配置项 safe-mode，控制是否需要全程开启安全模式。safe-mode 是布尔类型参数，默认为 false。如果设置为 true，则表示在增量同步的全过程中都会开启安全模式。例如，下面是一个开启了安全模式的任务配置示例：

```
syncers:                            # sync 处理单元的运行配置参数
  global:                           # 配置名称
```

```
    # 其他配置省略
    safe-mode: true                  # 增量同步任务全程开启"安全模式"
    # 其他配置省略
##### ----------- 实例配置 -----------
mysql-instances:
  -
    source-id: "mysql-replica-01"
    # 其他配置省略
    syncer-config-name: "global"            # syncers 配置的名称
```

注意事项

如果你出于安全考虑，希望全程开启安全模式，你需要注意以下事项：

- 安全模式对于增量同步有额外开销。频繁的 DELETE + REPLACE 操作会带来主键或唯一索引的频繁变化，带来了比单纯 UPDATE 语句更大的性能开销。
- 安全模式会强制替换主键相同的记录，可能导致下游的记录数据丢失。如果由于上游分片表合并导入下游的配置不当，可能导致在上游合并导入到下游数据库的时候，出现大量的主键或唯一索引冲突。如果此时全程开启安全模式，会导致下游大量的数据丢失，并且可能无法从任务中看到有任何的异常，导致大量数据不一致。
- 安全模式依赖于通过主键或唯一索引来判断冲突。如果下游数据库对应的表没有主键或唯一索引，会导致 REPLACE 语句无法实现替换插入的目的。这种情况下，即使开启了安全模式，DM 将 INSERT 语句改写成 REPLACE 并执行后，依旧会向下游插入重复记录。

因此，如果上游存在主键重复的数据，业务可以接受丢失重复数据和性能损失，你可以开启安全模式来忽略数据重复错误。

13.6.9.1.3 DM Relay Log

DM (Data Migration) 工具的 relay log 由若干组有编号的文件和一个索引文件组成。这些有编号的文件包含了描述数据库更改的事件。索引文件包含所有使用过的 relay log 的文件名。

在启用 relay log 功能后，DM-worker 会自动将上游 binlog 迁移到本地配置目录（若使用 TiUP 部署 DM，则迁移目录默认为 <deploy_dir> / <relay_log> ）。本地配置目录 <relay_log> 的默认值是 relay-dir，可在上游数据库配置文件中进行修改。自 v5.4.0 版本起，你可以在DM-worker 配置文件中通过 relay-dir 配置本地配置目录，其优先级高于上游数据库的配置文件。

适用场景

MySQL 的存储空间是有限制的，通常会设置 binlog 的最长保存时间，当上游把 binlog 清除掉之后，如果 DM 还需要对应位置的 binlog 就会拉取失败，导致同步任务出错。每增加一个同步任务，DM 都会在上游建立一条连接用于拉取 binlog，连接数过多会对上游造成比较大的负载。开启 relay log 后，同一个上游的多个同步任务可以复用已经拉到本地的 relay log，减少了对上游的压力。

在全量加增量数据迁移任务 (task-mode=all) 中，DM 需要先进行全量数据迁移，再根据 binlog 进行增量同步。若全量阶段持续时间较长，上游 binlog 可能会被清除，导致增量同步无法进行。若提前开启了 relay log，DM 会自动在本地保留足够的日志，保证增量任务正常进行。

一般情况下建议开启 relay log，但需知晓 relay log 可能导致的负面作用：

由于 relay log 需要写入到磁盘中，这一过程会产生外部 IO 和一些 CPU 消耗，可能导致整个同步链路变长，从而增加数据同步的时延。对时延要求十分敏感的同步任务，暂时不推荐使用 relay log。

> **注意：**
>
> DM v2.0.7 及之后的版本中，对 relay log 写入进行了优化，增加的时延和 CPU 消耗已相对较小。

使用 relay log

开启/关闭 relay log

在 v5.4.0 及之后的版本中，你可以通过将 `enable-relay` 设为 `true` 开启 relay log。自 v5.4.0 起，DM-worker 在绑定上游数据源时，会检查上游数据源配置中的 `enable-relay` 项。如果 `enable-relay` 为 `true`，则为该数据源启用 relay log 功能。

具体配置方式参见上游数据源配置文件介绍

除此以外，你也可以通过 `start-relay` 或 `stop-relay` 命令动态调整数据源的 `enable-relay` 并即时开启或关闭 relay log。

```
start-relay -s mysql-replica-01
```

```
{
    "result": true,
    "msg": ""
}
```

> **注意：**
>
> 在 v2.0.2 及之后的 v2.0 版本，以及在 v5.3.0 版本中，上游数据源配置中的 `enable-relay` 项失效，你只能通过`start-relay` 和 `stop-relay`命令开启和关闭 relay log。加载数据源配置时，如果 DM 发现配置中的 `enable-relay` 项为 `true`，会给出如下信息提示：
>
> ```
> Please use `start-relay` to specify which workers should pull relay log of relay-
>     ↪ enabled sources.
> ```

> **警告：**
>
> 该启动方式在 v6.1 版本中标记为弃用，在未来版本可能会被移除。相关命令的输出中您会看到如下提示：`start-relay/stop-relay with worker name will be deprecated soon. You ↪ can try stopping relay first and use start-relay without worker name instead`。

start-relay 命令可以配置一个或多个 DM-worker 为指定数据源迁移 relay log，但只能指定空闲或者已绑定了该上游数据源的 DM-worker。使用示例如下：

```
start-relay -s mysql-replica-01 worker1 worker2
```

```
{
    "result": true,
    "msg": ""
}
```

```
stop-relay -s mysql-replica-01 worker1 worker2
```

```
{
    "result": true,
    "msg": ""
}
```

在 v2.0.2 之前的版本（不含 v2.0.2），DM-worker 在绑定上游数据源时，会检查上游数据源配置中的 enable-relay 项。如果 enable-relay 为 true，则为该数据源启用 relay log 功能。

具体配置方式参见上游数据源配置文件介绍

查询 relay log

使用 query-status -s 命令，可以查询 relay log 的状态：

```
query-status -s mysql-replica-01
```

期望输出

```
{
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "no sub task started",
            "sourceStatus": {
                "source": "mysql-replica-01",
                "worker": "worker2",
                "result": null,
                "relayStatus": {
                    "masterBinlog": "(mysql-bin.000005, 916)",
                    "masterBinlogGtid": "09bec856-ba95-11ea-850a-58f2b4af5188:1-28",
                    "relaySubDir": "09bec856-ba95-11ea-850a-58f2b4af5188.000001",
                    "relayBinlog": "(mysql-bin.000005, 4)",
                    "relayBinlogGtid": "09bec856-ba95-11ea-850a-58f2b4af5188:1-28",
                    "relayCatchUpMaster": false,
```

```
                    "stage": "Running",
                    "result": null
                }
            },
            "subTaskStatus": [
            ]
        },
        {
            "result": true,
            "msg": "no sub task started",
            "sourceStatus": {
                "source": "mysql-replica-01",
                "worker": "worker1",
                "result": null,
                "relayStatus": {
                    "masterBinlog": "(mysql-bin.000005, 916)",
                    "masterBinlogGtid": "09bec856-ba95-11ea-850a-58f2b4af5188:1-28",
                    "relaySubDir": "09bec856-ba95-11ea-850a-58f2b4af5188.000001",
                    "relayBinlog": "(mysql-bin.000005, 916)",
                    "relayBinlogGtid": "",
                    "relayCatchUpMaster": true,
                    "stage": "Running",
                    "result": null
                }
            },
            "subTaskStatus": [
            ]
        }
    ]
}
```

暂停/恢复 relay log

pause-relay 与 resume-relay 命令可以分别暂停及恢复拉取 relay log。这两个命令执行时都需要指定上游数据源的 source-id，例如：

```
pause-relay -s mysql-replica-01 -s mysql-replica-02
```

期望输出

```
{
    "op": "PauseRelay",
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
```

```
            "msg": "",
            "source": "mysql-replica-01",
            "worker": "worker1"
        },
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-02",
            "worker": "worker2"
        }
    ]
}
```

```
resume-relay -s mysql-replica-01
```

期望输出

```
{
    "op": "ResumeRelay",
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-01",
            "worker": "worker1"
        }
    ]
}
```

清理 relay log

DM 提供两种清理 relay log 的方式，手动清理和自动清理。两种清理方法都不会清理活跃的 relay log。

> 注意：
>
> - 活跃的 relay log：该 relay log 正在被同步任务使用。活跃的 relay log 当前只在 Syncer Unit 被更新和写入，如果一个为 All 模式的同步任务在全量导出/导入阶段花费了超过数据源 purge 里配置的过期时间，该 relay log 依旧会被清除。
> - 过期的 relay log：该 relay log 文件最后被改动的时间与当前时间的差值大于配置文件中的 expires 字段。

自动数据清理

启用自动数据清理需在 source 配置文件中进行以下配置：

```
##### relay log purge strategy
purge:
    interval: 3600
    expires: 24
    remain-space: 15
```

- purge.interval
  - 后台自动清理的时间间隔，以秒为单位。
  - 默认为 “3600”，表示每 3600 秒执行一次后台清理任务。
- purge.expires
  - 当前 relay 处理单元没有写入、或已有数据迁移任务当前或未来不需要读取的 relay log 在被后台清理前可保留的小时数。
  - 默认为 “0”，表示不按 relay log 的更新时间执行数据清理。
- purge.remain-space
  - 剩余磁盘空间，单位为 GB。若剩余磁盘空间小于该配置，则指定的 DM-worker 机器会在后台尝试自动清理可被安全清理的 relay-log。若这一数字被设为 “0”，则表示不按剩余磁盘空间来清理数据。
  - 默认为 “15”，表示可用磁盘空间小于 15GB 时，DM-master 会尝试安全地清理 relay log。

手动数据清理

手动数据清理是指使用 dmctl 提供的 purge-relay 命令，通过指定 subdir 和 binlog 文件名，来清理掉指定 binlog 之前的所有 relay log。若在命令中不填 -subdir 选项，则默认清理最新 relay log 子目录之前的所有 relay log。

假设当前 relay log 的目录结构如下：

```
tree .
```

```
.
|-- deb76a2b-09cc-11e9-9129-5242cf3bb246.000001
|   |-- mysql-bin.000001
|   |-- mysql-bin.000002
|   |-- mysql-bin.000003
|   `-- relay.meta
|-- deb76a2b-09cc-11e9-9129-5242cf3bb246.000003
|   |-- mysql-bin.000001
|   `-- relay.meta
|-- e4e0e8ab-09cc-11e9-9220-82cc35207219.000002
|   |-- mysql-bin.000001
|   `-- relay.meta
`-- server-uuid.index
```

```
cat server-uuid.index
```

```
deb76a2b-09cc-11e9-9129-5242cf3bb246.000001
e4e0e8ab-09cc-11e9-9220-82cc35207219.000002
deb76a2b-09cc-11e9-9129-5242cf3bb246.000003
```

在 dmctl 中使用 purge-relay 命令的示例如下：

- 以下命令指定的 relay log 子目录为 e4e0e8ab-09cc-11e9-9220-82cc35207219.000002，该子目录之前的 relay log 子目录为 deb76a2b-09cc-11e9-9129-5242cf3bb246.000001。所以该命令实际清空了 deb76a2b-09
  ↪ cc-11e9-9129-5242cf3bb246.000001 子目录，保留 e4e0e8ab-09cc-11e9-9220-82cc35207219.000002 和 deb76a2b-09cc-11e9-9129-5242cf3bb246.000003 子目录。

```
purge-relay -s mysql-replica-01 --filename mysql-bin.000001 --sub-dir e4e0e8ab-09cc-11e9
    ↪ -9220-82cc35207219.000002
```

- 以下命令默认 --sub-dir 为最新的 deb76a2b-09cc-11e9-9129-5242cf3bb246.000003 子目录。该目录之前的 relay log 子目录为 deb76a2b-09cc-11e9-9129-5242cf3bb246.000001 和 e4e0e8ab-09cc-11e9-9220-82
  ↪ cc35207219.000002，所以该命令实际清空了这两个子目录，保留了 deb76a2b-09cc-11e9-9129-5242
  ↪ cf3bb246.000003。

```
purge-relay -s mysql-replica-01 --filename mysql-bin.000001
```

Relay log 内部机制

目录结构

Relay log 本地存储的目录结构示例如下：

```
<deploy_dir>/<relay_log>/
|-- 7e427cc0-091c-11e9-9e45-72b7c59d52d7.000001
|   |-- mysql-bin.000001
|   |-- mysql-bin.000002
|   |-- mysql-bin.000003
|   |-- mysql-bin.000004
|   `-- relay.meta
|-- 842965eb-091c-11e9-9e45-9a3bff03fa39.000002
|   |-- mysql-bin.000001
|   `-- relay.meta
`-- server-uuid.index
```

- subdir：
  - DM-worker 把从上游数据库迁移到的 binlog 存储在同一目录下，每个目录都为一个 subdir。
  - subdir 的命名格式为 <上游数据库 UUID>.<本地 subdir 序列号>。
  - 在上游进行 primary 和 secondary 实例切换后，DM-worker 会生成一个序号递增的新 subdir 目录。
  - 在以上示例中，对于 7e427cc0-091c-11e9-9e45-72b7c59d52d7.000001 这一目录，7e427cc0-091c-11
    ↪ e9-9e45-72b7c59d52d7 是上游数据库的 UUID，000001 是本地 subdir 的序列号。

- `server-uuid.index`：记录当前可用的 subdir 目录。
- `relay.meta`：存储每个 subdir 中已迁移的 binlog 信息。例如，

```
cat c0149e17-dff1-11e8-b6a8-0242ac110004.000001/relay.meta
```

```
binlog-name = "mysql-bin.000010"    # 当前迁移的 binlog 名
binlog-pos = 63083620               # 当前迁移的 binlog 位置
binlog-gtid = "c0149e17-dff1-11e8-b6a8-0242ac110004:1-3328" # 当前迁移的 binlog GTID
```

也可能包含多个 GTID：

```
cat 92acbd8a-c844-11e7-94a1-1866daf8accc.000001/relay.meta
```

```
binlog-name = "mysql-bin.018393"
binlog-pos = 277987307
binlog-gtid = "3ccc475b-2343-11e7-be21-6c0b84d59f30:1-14,406a3f61-690d-11e7-87c5-6
    ↪ c92bf46f384:1-94321383,53bfca22-690d-11e7-8a62-18ded7a37b78:1-495,686e1ab6-c47e-11e7
    ↪ -a42c-6c92bf46f384:1-34981190,03fc0263-28c7-11e7-a653-6c0b84d59f30:1-7041423,05474
    ↪ d3c-28c7-11e7-8352-203db246dd3d:1-170,10b039fc-c843-11e7-8f6a-1866daf8d810
    ↪ :1-308290454"
```

DM 接收 binlog 的位置

- 从保存的 checkpoint 中（默认位于下游 dm_meta 库），获取各同步任务需要该数据源的最早位置。如果该位置晚于下述任何一个位置，则从此位置开始迁移。
- 若本地 relay log 有效（有效是指 relay log 具有有效的 `server-uuid.index`，`subdir` 和 `relay.meta` 文件），DM-worker 从 `relay.meta` 记录的位置恢复迁移。
- 若不存在有效的本地 relay log，但上游数据源配置文件中指定了 `relay-binlog-name` 或 `relay-binlog-gtid`：
  - 在非 GTID 模式下，若指定了 `relay-binlog-name`，则 DM-worker 从指定的 binlog 文件开始迁移。
  - 在 GTID 模式下，若指定了 `relay-binlog-gtid`，则 DM-worker 从指定的 GTID 开始迁移。
- 若不存在有效的本地 relay log，而且 DM 配置文件中未指定 `relay-binlog-name` 或 `relay-binlog-gtid`：
  - 在非 GTID 模式下，DM-worker 从自身所有 subtask 正在同步的最早 binlog 开始迁移，直至最新。
  - 在 GTID 模式下，DM-worker 从自身所有 subtask 正在同步的最早 GTID 开始迁移，直至最新。

> 注意：
> 若上游的 relay log 被清理掉，则会发生错误。在这种情况下，需设置 `relay-binlog-gtid` 来指定迁移的起始位置。

探索更多

- DM 源码阅读系列文章（六）relay log 的实现 | TiDB 工具

13.6.9.1.4 Data Migration DDL 特殊处理说明

DM 同步过程中，根据 DDL 语句以及所处场景的不同，将采用不同的处理方式。

忽略的 DDL 语句

以下语句 DM 并未支持，因此解析之后直接跳过。

```
<tr>
    <th>描述</th>
    <th>SQL</th>
</tr>
<tr>
    <td>transaction</td>
    <td><code>^SAVEPOINT</code></td>
</tr>
<tr>
    <td>skip all flush sqls</td>
    <td><code>^FLUSH</code></td>
</tr>
<tr>
    <td rowspan="3">table maintenance</td>
    <td><code>^OPTIMIZE\\s+TABLE</code></td>
</tr>
<tr>
    <td><code>^ANALYZE\\s+TABLE</code></td>
</tr>
<tr>
    <td><code>^REPAIR\\s+TABLE</code></td>
</tr>
<tr>
    <td>temporary table</td>
    <td><code>^DROP\\s+(\\/\\*\\!40005\\s+)?TEMPORARY\\s+(\\*\\/\\s+)?TABLE</code></td>
</tr>
<tr>
    <td rowspan="2">trigger</td>
    <td><code>^CREATE\\s+(DEFINER\\s?=.+?)?TRIGGER</code></td>
</tr>
<tr>
    <td><code>^DROP\\s+TRIGGER</code></td>
</tr>
<tr>
    <td rowspan="3">procedure</td>
    <td><code>^DROP\\s+PROCEDURE</code></td>
</tr>
<tr>
    <td><code>^CREATE\\s+(DEFINER\\s?=.+?)?PROCEDURE</code></td>
</tr>
```

```
<tr>
    <td><code>^ALTER\\s+PROCEDURE</code></td>
</tr>
<tr>
    <td rowspan="3">view</td>
    <td><code>^CREATE\\s*(OR REPLACE)?\\s+(ALGORITHM\\s?=.+?)?(DEFINER\\s?=.+?)?\\s+(SQL SECURITY
        ↪  DEFINER)?VIEW</code></td>
</tr>
<tr>
    <td><code>^DROP\\s+VIEW</code></td>
</tr>
<tr>
    <td><code>^ALTER\\s+(ALGORITHM\\s?=.+?)?(DEFINER\\s?=.+?)?(SQL SECURITY DEFINER)?VIEW</code
        ↪ ></td>
</tr>
<tr>
    <td rowspan="4">function</td>
    <td><code>^CREATE\\s+(AGGREGATE)?\\s*?FUNCTION</code></td>
</tr>
<tr>
    <td><code>^CREATE\\s+(DEFINER\\s?=.+?)?FUNCTION</code></td>
</tr>
<tr>
    <td><code>^ALTER\\s+FUNCTION</code></td>
</tr>
<tr>
    <td><code>^DROP\\s+FUNCTION</code></td>
</tr>
<tr>
    <td rowspan="3">tableSpace</td>
    <td><code>^CREATE\\s+TABLESPACE</code></td>
</tr>
<tr>
    <td><code>^ALTER\\s+TABLESPACE</code></td>
</tr>
<tr>
    <td><code>^DROP\\s+TABLESPACE</code></td>
</tr>
<tr>
    <td rowspan="3">event</td>
    <td><code>^CREATE\\s+(DEFINER\\s?=.+?)?EVENT</code></td>
</tr>
<tr>
    <td><code>^ALTER\\s+(DEFINER\\s?=.+?)?EVENT</code></td>
</tr>
```

```
<tr>
    <td><code>^DROP\\s+EVENT</code></td>
</tr>
<tr>
    <td rowspan="7">account management</td>
    <td><code>^GRANT</code></td>
</tr>
<tr>
    <td><code>^REVOKE</code></td>
</tr>
<tr>
    <td><code>^CREATE\\s+USER</code></td>
</tr>
<tr>
    <td><code>^ALTER\\s+USER</code></td>
</tr>
<tr>
    <td><code>^RENAME\\s+USER</code></td>
</tr>
<tr>
    <td><code>^DROP\\s+USER</code></td>
</tr>
<tr>
    <td><code>^DROP\\s+USER</code></td>
</tr>
```

改写的 DDL 语句

以下语句在同步到下游前会进行改写。

原始语句	实际执行语句
^CREATE DATABASE...	^CREATE DATABASE...IF NOT EXISTS
^CREATE TABLE...	^CREATE TABLE..IF NOT EXISTS
^DROP DATABASE...	^DROP DATABASE...IF EXISTS
^DROP TABLE...	^DROP TABLE...IF EXISTS
^DROP INDEX...	^DROP INDEX...IF EXISTS

合库合表迁移任务

当使用悲观协调模式和乐观协调模式进行分库分表合并迁移时，DDL 同步的行为存在变更，具体请参考悲观模式和乐观模式。

Online DDL

Online DDL 特性也会对 DDL 事件进行特殊处理，详情可参考迁移使用 GH-ost/PT-osc 的源数据库。

13.6.9.2 运行机制

13.6.9.2.1 Data Migration 中的 DML 同步机制

本文介绍了 DM 核心处理单元 Sync 如何同步从数据源或 relay log 中读取到的 DML。本文将介绍 DML 事件在 DM 内部的完整处理流程，包括 binlog 读取、过滤、路由、转换、优化以及执行等逻辑，并详细解读 DML 优化逻辑和 DML 执行逻辑。

DML 处理流程

Sync 单元对 DML 的处理步骤如下：

1. 从 MySQL、 MariaDB 或 relay log 中，读取 binlog event。
2. 转换读取到的 binlog event：
    1. Binlog filter：根据 binlog 表达式过滤 binlog event，通过 `filters` 配置。
    2. Table routing：根据 “库/表” 路由规则对 “库/表” 名进行转换，通过 `routes` 配置。
    3. Expression filter：根据 SQL 表达式过滤 binlog event，通过 `expression-filter` 配置。
3. 优化 DML 执行：
    1. Compactor：将对同一条记录（主键相同）的多个操作合并成一个操作，通过 `syncer.compact` 开启。
    2. Causality：将不同记录（主键不同）进行冲突检测，提升同步并发度。
    3. Merger：将多条 binlog 合并成一条 DML，通过 `syncer.multiple-rows` 开启。
4. 将 DML 执行到下游。
5. 定期保存 binlog position 或 GTID 到 checkpoint 中。

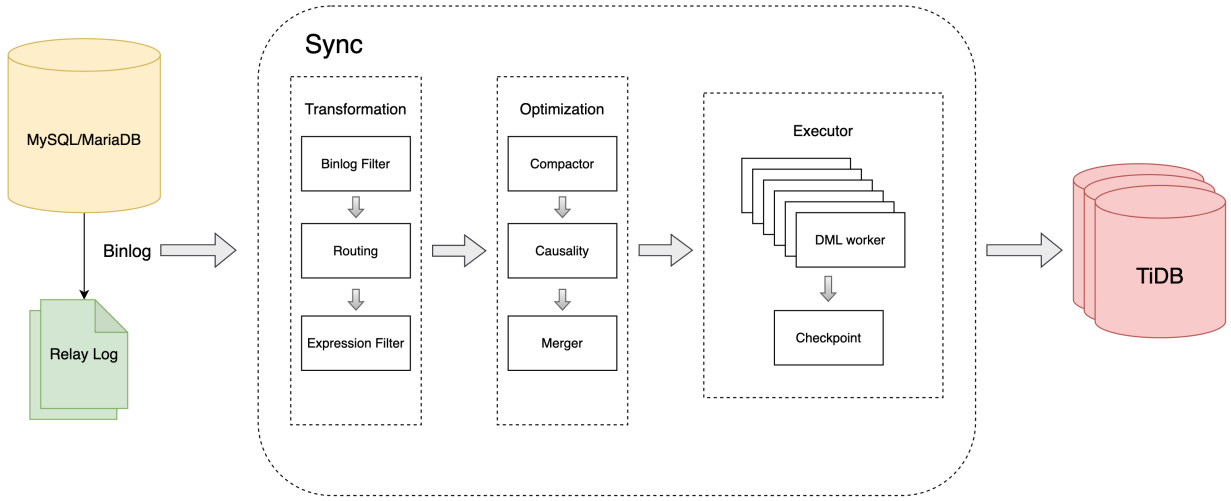


图 246: DML 处理逻辑

DML 优化逻辑

Sync 单元通过 Compactor、 Causality、 Merger 三个步骤，实现对 DML 的优化逻辑。

Compactor

DM 根据上游 binlog 记录，捕获记录的变更并同步到下游。当上游对同一条记录短时间内做了多次变更时（INSERT/UPDATE/DELETE），DM 可以通过 Compactor 将多次变更压缩成一次变更，减少下游压力，提升吞吐。例如：

```
INSERT + UPDATE => INSERT
INSERT + DELETE => DELETE
UPDATE + UPDATE => UPDATE
UPDATE + DELETE => DELETE
DELETE + INSERT => UPDATE
```

Compactor 特性默认关闭，如需启用可在迁移任务的 sync 配置模块开启，例如：

```
syncers:                            # sync 处理单元的运行配置参数
  global:                           # 配置名称
    ...                             # 省略其他配置
    compact: true
```

Causality

MySQL binlog 顺序同步模型要求按照 binlog 顺序依次同步 binlog event，这样的同步模型无法满足高 QPS 低同步延迟的需求。此外，由于不是所有的 binlog 涉及到的操作都存在冲突，顺序同步也是非必要的。

DM 通过冲突检测机制，识别出需要顺序执行的 binlog，确保这些 binlog 顺序执行的同时，最大程度地保持其他 binlog 并发执行，以此提高 binlog 同步的性能。

Causality 采用一种类似并查集的算法，对每一个 DML 进行分类，将相互关联的 DML 分为一组。具体算法可参考并行执行 DML。

Merger

根据 MySQL binlog 协议，每条 binlog 对应一行数据的变更操作。通过 Merger，DM 可以将多条 binlog 合并成一条 DML，再执行到下游，减少网络的交互。例如：

```
  INSERT tb(a,b) VALUES(1,1);
+ INSERT tb(a,b) VALUES(2,2);
= INSERT tb(a,b) VALUES(1,1),(2,2);

  UPDATE tb SET a=1, b=1 WHERE a=1;
+ UPDATE tb SET a=2, b=2 WHERE a=2;
= INSERT tb(a,b) VALUES(1,1),(2,2) ON DUPLICATE UPDATE a=VALUES(a), b=VALUES(b)

  DELETE tb WHERE a=1
+ DELETE tb WHERE a=2
= DELETE tb WHERE (a) IN (1),(2);
```

Merger 特性默认关闭，如需启用可在迁移任务的 sync 配置模块开启，例如：

```
syncers:                            # sync 处理单元的运行配置参数
  global:                           # 配置名称
```

```
...                                    # 省略其他配置
multiple-rows: false
```

DML 执行逻辑

Sync 单元对 DML 进行优化后，再进行执行逻辑。

DML 生成

DM 内嵌一个 schema tracker，用于记录上下游的 schema 信息：

- 当 DM 收到 DDL 时，DM 更新内部 schema tracker 的表结构。
- 当收到 DML 时，DM 根据 schema tracker 的表结构生成对应的 DML。

生成 DML 的具体逻辑如下：

1. Sync 记录上游的初始表结构：
    - 当启动全量与增量任务时，Sync 使用上游全量同步时导出的表结构作为上游的初始表结构。
    - 当启动增量任务时，由于 MySQL binlog 没有记录表结构信息，Sync 使用下游对应的表的表结构作为上游的初始表结构。
2. 由于用户上下游表结构可能不一致，例如下游比上游多了额外的列，或者上下游主键不一致，为了保证数据同步的正确性，DM 记录下游对应表的主键和唯一键信息。
3. DM 生成 DML：
    - 使用 schema tracker 中记录的上游表结构生成 DML 语句的列名。
    - 使用 binlog 中记录的列值生成 DML 语句的列值。
    - 使用 schema tracker 中记录的下游主键或唯一键生成 DML 语句中的 WHERE 条件。当表结构无唯一键时，DM 会使用 binlog 中记录的所有列值作为 WHERE 条件。

Worker count

Causality 可以通过冲突检测算法将 binlog 分成多个 group 并发地执行到下游。DM 通过设置 worker-count，控制并发的数量。当下游 TiDB 的 CPU 占用不高时，增大并发的数量可以有效地提高数据同步的吞吐量。

你可以通过`syncer.worker-count` 配置项，修改并发迁移 DML 的线程数量。

Batch

DM 将多条 DML 积累到一个事务中执行到下游。当 DML Worker 收到 DML 时，将 DML 加入到缓存中。当缓存中 DML 数量达到预定阈值时，或者 DML worker 较长时间没有收到 DML 时，DML worker 将缓存中的 DML 执行到下游。

你可以通过`syncer.batch` 配置项，修改每个事务包含的 DML 的数量。

checkpoint

DML 执行和 checkpoint 更新的操作不是原子的。

在 DM 中，checkpoint 默认每 30 秒更新一次。同时，由于存在多个 DML worker 进程，checkpoint 进程会计算所有 DML worker 中同步进度最早的 binlog 位点，将该位点作为当前同步的 checkpoint。所有早于此位点的 binlog，都已保证被成功地执行到下游。

注意事项

事务一致性

DM 是按照“行级别”进行数据同步的，并不保证事务原样同步。在 DM 中，上游的一个事务会被拆成多行，分发到不同的 DML Worker 中并发执行。因此，当 DM 同步任务报错暂停，或者用户手动暂停任务时，下游可能停留在一个中间状态；即上游一个事务中的 DML 语句，可能一部分同步到下游，一部分没有，导致下游处于不一致的状态。

为了尽可能使任务暂停时下游处于一致状态，DM v5.3.0 起，任务暂停时，会等待 10 秒，使正在处理的上游事务全部同步到下游再真正暂停任务。如果上游一个事务在 10 秒内还未全部同步到下游，那么下游仍然可能处于不一致的状态。

安全模式

DML 执行和 checkpoint 写操作不是同步的，并且写 checkpoint 操作和写下游数据也并不能保证原子性。当 DM 因为某些原因异常退出时，checkpoint 可能只记录到退出时刻之前的一个恢复点。因此，当同步任务重启时，DM 可能会重复写入部分数据，也就是说，DM 实际上提供的是“至少一次处理”的逻辑（At-least-once processing），相同的数据可能会被处理一次以上。

为了保证数据是可重入的，DM 在异常重启时会进入安全模式。具体逻辑参阅DM 安全模式。

开启安全模式期间，为了保证数据可重入，DM 会进行如下转换：

- 将上游 INSERT 语句转换成 REPLACE 语句
- 将上游 UPDATE 语句转换成 DELETE + REPLACE 语句。

精确一次处理 (Exactly-Once Processing)

目前 DM 仅保证最终一致性，尚未支持“精确一次处理”及“保持事务原有顺序同步”。

13.6.9.3 命令行

13.6.9.3.1 TiDB Data Migration 命令行参数

本文档介绍 TiDB Data Migration (DM) 中各组件的主要命令行参数。

DM-master

`--advertise-addr`

- DM-master 用于接收客户端请求的外部地址
- 默认值为 `"{master-addr}"`
- 可选参数，可以为 `"域名:port"` 的形式

`--advertise-peer-urls`

- DM-master 节点间通信的外部连接地址
- 默认值为 `"{peer-urls}"`
- 可选参数，可以为 `"http(s)://域名:port"` 的形式

--config

- DM-master 配置文件路径
- 默认值为 ""
- 可选参数

--data-dir

- DM-master 用于存储自身数据的目录
- 默认值为 "default.{name}"
- 可选参数

--initial-cluster

- 用于 bootstrap DM-master 集群的 "{节点名}={外部地址}" 列表
- 默认值为 "{name}={advertise-peer-urls}"
- 在未指定 join 参数时需要指定该参数。一个 3 节点集群的配置示例为 "dm-master-1=http://172.16.15.11:8291,dm-master-2=http://172.16.15.12:8291,dm-master-3=http://172.16.15.13:8291"

--join

- DM-master 节点加入到已有集群时，已有集群的 advertise-addr 地址列表
- 默认值为 ""
- 未指定 initial-cluster 参数时需要指定该参数。一个新节点加入到一个已有 2 个节点的集群的示例为 "172.16.15.11:8261,172.16.15.12:8261"

--log-file

- log 输出文件名
- 默认值为 ""
- 可选参数

-L

- log 级别
- 默认值为 "info"
- 可选参数

--master-addr

- DM-master 监听客户端请求的地址
- 默认值为 ""
- 必选参数

`--name`

- DM-master 节点名称
- 默认值为 `"dm-master-{hostname}"`
- 必选参数

`--peer-urls`

- DM-master 节点间通信的监听地址
- 默认值为 `"http://127.0.0.1:8291"`
- 必选参数

DM-worker

`--advertise-addr`

- DM-worker 用于接受客户端请求的外部地址
- 默认值为 `"{worker-addr}"`
- 可选参数，可以为 `"域名:port"` 的形式

`--config`

- DM-worker 配置文件路径
- 默认值为 `""`
- 可选参数

`--join`

- DM-worker 注册到集群时，相应集群的 DM-master 节点的 `{advertise-addr}` 列表
- 默认值为 `""`
- 必选参数，一个 3 DM-master 节点的集群配置示例为 `"172.16.15.11:8261,172.16.15.12:8261,172.16.15.13:8261"`
  ↪

`--log-file`

- log 输出文件名
- 默认值为 `""`
- 可选参数

`-L`

- log 级别
- 默认值为 `"info"`
- 可选参数

`--name`

- DM-worker 节点名称
- 默认值为 `"{advertise-addr}"`
- 必选参数

`--worker-addr`

- DM-worker 监听客户端请求的地址
- 默认值为 `""`
- 必选参数

dmctl

`--config`

- dmctl 配置文件路径
- 默认值为 `""`
- 可选参数

`--master-addr`

- dmctl 要连接的集群的任意 DM-master 节点的 `{advertise-addr}`
- 默认值为 `""`
- 需要与 DM-master 交互时为必选参数

`--encrypt`

- 将明文数据库密码加密成密文
- 默认值为 `""`
- 指定该参数时，仅用于加密明文而不会与 DM-master 交互

`--decrypt`

- 将使用 dmctl 加密过的密文解密为明文
- 默认值为 `""`
- 指定该参数时，仅用于解密密文而不会与 DM-master 交互

13.6.9.4 配置文件

13.6.9.4.1 DM 配置简介

本文档简要介绍 DM (Data Migration) 的配置文件和数据迁移任务的配置。

配置文件

- `dm-master.toml`：DM-master 进程的配置文件，包括 DM-master 的拓扑信息、日志等各项配置。配置说明详见DM-master 配置文件介绍。
- `dm-worker.toml`：DM-worker 进程的配置文件，包括 DM-worker 的拓扑信息、日志等各项配置。配置说明详见DM-worker 配置文件介绍。
- `source.yaml`：上游数据库 MySQL/MariaDB 相关配置。配置说明详见上游数据库配置文件介绍。

迁移任务配置

创建数据迁移任务

具体步骤如下：

1. 使用 dmctl 将数据源配置加载到 DM 集群；
2. 参考数据任务配置向导来创建 `your_task.yaml`；
3. 使用 dmctl 创建数据迁移任务。

关键概念

DM 配置的关键概念如下：

概念	解释	配置文件
source-id	唯一确定一个 MySQL 或 MariaDB 实例，或者一个具有主从结构的复制组，字符串长度不大于 32	`source.yaml` 的 `source-id`；`task.yaml` 的 `source-id`
DM-master ID	唯一确定一个 DM-master（取值于 `dm-master.toml` 的 `master-addr` 参数）	`dm-master.toml` 的 `master-addr`
DM-worker ID	唯一确定一个 DM-worker（取值于 `dm-worker.toml` 的 `worker-addr` 参数）	`dm-worker.toml` 的 `worker-addr`

13.6.9.4.2 TiDB Data Migration 上游数据库配置文件介绍

本文介绍 TiDB Data Migration (DM) 上游数据库的配置文件，包括配置文件示例与配置项说明。

配置文件示例

上游数据库的示例配置文件如下所示：

```
source-id: "mysql-replica-01"

##### 是否开启 GTID
enable-gtid: false

##### 是否开启 relay log
enable-relay: false       # 该配置项从 DM v2.0.2 版本起弃用，使用 `start-relay` 命令开启 relay
    ↪ log
relay-binlog-name: ""     # 拉取上游 binlog 的起始文件名
relay-binlog-gtid: ""     # 拉取上游 binlog 的起始 GTID
##### relay-dir: "relay-dir"  # 存储 relay log 的目录，默认值为 "relay-dir"。从 v6.1
    ↪ 版本起该配置标记为弃用，被 worker 配置中的同名参数取代

from:
  host: "127.0.0.1"
  port: 3306
  user: "root"
  password: "ZqMLjZ2j5khNelDEfDoUhkD5aV5fIJOe0fiog9w=" # 推荐使用 dmctl
      ↪ 对上游数据库的用户密码加密之后的密码
  security:                       # 上游数据库 TLS 相关配置
    ssl-ca: "/path/to/ca.pem"
    ssl-cert: "/path/to/cert.pem"
    ssl-key: "/path/to/key.pem"

##### purge:
#####   interval: 3600
#####   expires: 0
#####   remain-space: 15

##### checker:
#####   check-enable: true
#####   backoff-rollback: 5m0s
#####   backoff-max: 5m0s       # backoff 的最大值，不能小于 1s

##### 从 DM v2.0.2 开始，Binlog event filter 也可以在上游数据库配置文件中进行配置
##### case-sensitive: false
##### filters:
##### - schema-pattern: dmctl
#####   table-pattern: t_1
#####   events: []
#####   sql-pattern:
```

```
#####   - alter table .* add column `aaa` int
#####   action: Ignore
```

> **注意：**
>
> 在 DM v2.0.1 版本中，请勿同时配置 `enable-gtid` 与 `enable-relay` 为 `true`，否则可能引发增量数据丢失问题。

配置项说明

Global 配置

配置项	说明
`source-id`	标识一个 MySQL 实例。
`enable-gtid`	是否使用 GTID 方式从上游拉取 binlog，默认值为 false。一般情况下不需要手动配置，如果上游数据库启用了 GTID 支持，且需要做主从切换，则将该配置项设置为 true。
`enable-relay`	是否开启 relay log，默认值为 false。自 DM v2.0.2 版本起，该配置项弃用，需使用 `start-relay` 命令开启 relay log。
`relay-binlog-name`	拉取上游 binlog 的起始文件名，例如“mysql-bin.000002”，该配置在 `enable-gtid` 为 false 的情况下生效。如果不配置该项，DM-worker 将从正在同步的最早的 binlog 文件开始拉取，一般情况下不需要手动配置。
`relay-binlog-gtid`	拉取上游 binlog 的起始 GTID，例如“e9a1fc22-ec08-11e9-b2ac-0242ac110003:1-7849”，该配置在 `enable-gtid` 为 true 的情况下生效。如果不配置该项，DM-worker 将从正在同步的最早 binlog GTID 开始拉取 binlog，一般情况下不需要手动配置。
`relay-dir`	存储 relay log 的目录，默认值为“./relay_log”。
`host`	上游数据库的 host。
`port`	上游数据库的端口。
`user`	上游数据库使用的用户名。
`password`	上游数据库的用户密码。注意：推荐使用 dmctl 加密后的密码。
`security`	上游数据库 TLS 相关配置。配置的证书文件路径需能被所有节点访问。若配置为本地路径，则集群所有节点需要将证书文件拷贝一份放在各节点机器相同的路径位置上。

relay log 清理策略配置（purge 配置项）

一般情况下不需要手动配置，如果 relay log 数据量较大，磁盘空间不足，则可以通过设置该配置项来避免 relay log 写满磁盘。

配置项	说明
`interval`	定期检查 relay log 是否过期的间隔时间，默认值：3600，单位：秒。
`expires`	relay log 的过期时间，默认值为 0，单位：小时。未由 relay 处理单元进行写入、或已有数据迁移任务当前或未来不需要读取的 relay log 在超过过期时间后会被 DM 删除。如果不设置则 DM 不会自动清理过期的 relay log。
`remain-space`	设置最小的可用磁盘空间。当磁盘可用空间小于这个值时，DM-worker 会尝试删除 relay log，默认值：15，单位：GB。

> 注意：
>
> 仅在 `interval` 不为 0 且 `expires` 和 `remain-space` 两个配置项中至少有一个不为 0 的情况下 DM 的自动清理策略才会生效。

任务状态检查配置（checker 配置项）

DM 会定期检查当前任务状态以及错误信息，判断恢复任务能否消除错误，并自动尝试恢复任务进行重试。DM 会使用指数回退策略调整检查间隔。这些行为可以通过如下配置进行调整：

配置项	说明
`check-enable`	启用自动重试功能。
`backoff-rollback`	如果指数回退策略的间隔大于该值，且任务处于正常状态，尝试减小间隔。
`backoff-max`	指数回退策略的间隔的最大值，该值必须大于 1 秒。

Binlog event filter

从 DM v2.0.2 开始，Binlog event filter 也可以在上游数据库配置文件中进行配置。

配置项	说明
`case-sensitive`	Binlog event filter 标识符是否大小写敏感。默认值：false。
`filters`	配置 Binlog event filter，含义见Binlog event filter 参数解释。

13.6.9.4.3 DM 任务完整配置文件介绍

本文档主要介绍 Data Migration (DM) 的任务完整的配置文件，包含全局配置 和实例配置 两部分。

关键概念

关于包括 source-id 和 DM-worker ID 在内的关键概念的介绍，请参阅关键概念。

完整配置文件示例

下面是一个完整的配置文件示例，通过该示例可以完成复杂的数据迁移功能。

```
---

##### ----------- 全局配置 -----------
###### ********* 基本信息配置 *********
name: test                        # 任务名称，需要全局唯一
task-mode: all                    # 任务模式，可设为 "full" - "只进行全量数据迁移"、"incremental" -
    ↪  "Binlog 实时同步"、"all" - "全量 + Binlog 实时同步"
shard-mode: "pessimistic"         # 任务协调模式，可选的模式有 ""、"pessimistic、"optimistic"。
    ↪ 默认值为 "" 即无需协调。如果是分库分表合并任务，请设置为悲观协调模式 "pessimistic"。
                                  # 在 v2.0.6 版本后乐观模式逐渐成熟，
                                      ↪ 深入了解乐观协调模式的原理和使用限制后，
                                      ↪ 也可以设置为乐观协调模式 "optimistic"
meta-schema: "dm_meta"            # 下游储存 `meta` 信息的数据库
##### timezone: "Asia/Shanghai"      # 指定数据迁移任务时 SQL Session 使用的时区。DM
    ↪ 默认使用目标库的全局时区配置进行数据迁移，并且自动确保同步数据的正确性。
    ↪ 使用自定义时区依然可以确保整个流程的正确性，但一般不需要手动指定。

case-sensitive: false             # schema/table 是否大小写敏感
online-ddl: true                  # 支持上游 "gh-ost" 、"pt" 的自动处理
online-ddl-scheme: "gh-ost"       # `online-ddl-scheme` 已被弃用，建议使用 `online-ddl`。
clean-dump-file: true             # 是否清理 dump 阶段产生的文件，包括 metadata 文件、建库建表 SQL
    ↪ 文件以及数据导入 SQL 文件
collation_compatible: "loose"     # 同步 CREATE 语句中缺省 Collation 的方式，可选 "loose" 和 "
    ↪ strict"，默认为 "loose"。"loose" 模式不会显式补充上游缺省的 Collation，"strict"
    ↪ 会显式补充上游缺省的 Collation。当使用 "strict" 模式，但下游不支持上游缺省的 Collation 时
    ↪ ，下游可能会报错。
ignore-checking-items: []         # 忽略检查项。可用值请参考 precheck 说明：https://docs.pingcap.
    ↪ com/zh/tidb/stable/dm-precheck。

target-database:                  # 下游数据库实例配置
  host: "192.168.0.1"
  port: 4000
  user: "root"
  password: "/Q7B9DizNLLTTfiZHv9WoEAKamfpIUs="  # 推荐使用经 `dmctl encrypt` 加密后的密码
  max-allowed-packet: 67108864                 # 设置 DM 内部连接 TiDB 服务器时，TiDB 客户端的 "
      ↪ max_allowed_packet" 限制（即接受的最大数据包限制），单位为字节，默认 67108864 (64 MB)
                                               # 该配置项从 DM v2.0.0 起弃用，DM 会自动获取连接
                                                   ↪ TiDB 的 "max_allowed_packet"
```

```
  session:                                  # 设置 TiDB 的 session 变量，在 v1.0.6 版本引入。
      ↪ 更多变量及解释参见 `https://docs.pingcap.com/zh/tidb/stable/system-variables`
    sql_mode: "ANSI_QUOTES,NO_ZERO_IN_DATE,NO_ZERO_DATE" # 从 DM v2.0.0 起，
        ↪ 如果配置文件中没有出现该项，DM 会自动从下游 TiDB 中获得适合用于 "sql_mode" 的值。
        ↪ 手动配置该项具有更高优先级
    tidb_skip_utf8_check: 1                 # 从 DM v2.0.0 起，
        ↪ 如果配置文件中没有出现该项，DM 会自动从下游 TiDB 中获得适合用于 "tidb_skip_utf8_check
        ↪ " 的值。手动配置该项具有更高优先级
    tidb_constraint_check_in_place: 0
  security:                         # 下游 TiDB TLS 相关配置
    ssl-ca: "/path/to/ca.pem"
    ssl-cert: "/path/to/cert.pem"
    ssl-key: "/path/to/key.pem"

###### ******** 功能配置集 **********

routes:                             # 上游和下游表之间的路由 table routing 规则集
  route-rule-1:                     # 配置名称
    schema-pattern: "test_*"        # 库名匹配规则，支持通配符 "*" 和 "?"
    table-pattern: "t_*"            # 表名匹配规则，支持通配符 "*" 和 "?"
    target-schema: "test"           # 目标库名称
    target-table: "t"               # 目标表名称
    # 可选配置：提取各分库分表的源信息，并写入下游用户自建的列，用于标识合表中各行数据的来源。
        ↪ 如果配置该项，需要提前在下游手动创建合表，具体可参考 “table routing 文档” <https://
        ↪ docs.pingcap.com/zh/tidb/v7.0/dm-key-features#table-routing>。
    # extract-table:                                    # 提取分表去除 t_ 的后缀信息，
        ↪ 并写入下游合表 c_table 列，例如，t_01 分表的数据会提取 01 写入下游 c_table 列
    #   table-regexp: "t_(.*)"
    #   target-column: "c_table"
    # extract-schema:                                   # 提取分库去除 test_ 的后缀信息，
        ↪ 并写入下游合表 c_schema 列，例如，test_02 分库的数据会提取 02 写入下游 c_schema 列
    #   schema-regexp: "test_(.*)"
    #   target-column: "c_schema"
    # extract-source:                                   # 提取数据库源实例信息写入 c_source
        ↪ 列，例如，mysql-replica-01 数据源实例的数据会提取 mysql-replica-01 写入下游 c_source
        ↪ 列
    #   source-regexp: "(.*)"
    #   target-column: "c_source"
  route-rule-2:
    schema-pattern: "test_*"
    target-schema: "test"

filters:                                # 上游数据库实例匹配的表的 binlog event filter
    ↪ 规则集
  filter-rule-1:                        # 配置名称
```

```
    schema-pattern: "test_*"                    # 库名匹配规则，支持通配符 "*" 和 "?"
    table-pattern: "t_*"                        # 表名匹配规则，支持通配符 "*" 和 "?"
    events: ["truncate table", "drop table"]    # 匹配哪些 event 类型
    action: Ignore                              # 对与符合匹配规则的 binlog 迁移（Do）还是忽略(
        ↪ Ignore)
  filter-rule-2:
    schema-pattern: "test_*"
    events: ["all dml"]
    action: Do

expression-filter:                  # 定义数据源迁移行变更的过滤规则，可以定义多个规则
  # 过滤 `expr_filter`.`tbl` 的 c 为偶数的插入
  even_c:                           # 规则名称
    schema: "expr_filter"           # 要匹配的上游数据库库名，不支持通配符匹配或正则匹配
    table: "tbl"                    # 要匹配的上游表名，不支持通配符匹配或正则匹配
    insert-value-expr: "c % 2 = 0"

block-allow-list:                   # 定义数据源迁移表的过滤规则，可以定义多个规则。如果 DM
    ↪ 版本早于 v2.0.0-beta.2 则使用 black-white-list
  bw-rule-1:                        # 规则名称
    do-dbs: ["~^test.*", "user"]    # 迁移哪些库
    ignore-dbs: ["mysql", "account"] # 忽略哪些库
    do-tables:                      # 迁移哪些表
    - db-name: "~^test.*"
      tbl-name: "~^t.*"
    - db-name: "user"
      tbl-name: "information"
  bw-rule-2:                        # 规则名称
    ignore-tables:                  # 忽略哪些表
    - db-name: "user"
      tbl-name: "log"

mydumpers:                          # dump 处理单元的运行配置参数
  global:                           # 配置名称
    threads: 4                      # dump 处理单元从上游数据库实例导出数据和 check-task
        ↪ 访问上游的线程数量，默认值为 4
    chunk-filesize: 64              # dump 处理单元生成的数据文件大小，默认值为 64，单位为 MB
    extra-args: "--consistency none" # dump 处理单元的其他参数，不需要在 extra-args 中配置 table-
        ↪ list，DM 会自动生成

loaders:                            # load 处理单元的运行配置参数
  global:                           # 配置名称
    pool-size: 16                   # load 处理单元并发执行 dump 处理单元的 SQL 文件的线程数量，
        ↪ 默认值为 16，当有多个实例同时向 TiDB 迁移数据时可根据负载情况适当调小该值
```

```
# 保存上游全量导出数据的目录。该配置项的默认值为 "./dumped_data"。
# 支持配置为本地文件系统路径，也支持配置为 Amazon S3 路径，如：s3://dm_bucket/dumped_data?
    ↪ endpoint=s3-website.us-east-2.amazonaws.com&access_key=s3accesskey&secret_access_key=
    ↪ s3secretkey&force_path_style=true
dir: "./dumped_data"

# 全量阶段数据导入的模式。可以设置为如下几种模式：
# - "logical"(默认)。使用 TiDB Lightning logical import 进行导入。文档：https://docs.pingcap.
    ↪ com/zh/tidb/stable/tidb-lightning-logical-import-mode
# - "physical"。使用 TiDB Lightning physical import 进行导入。文档：https://docs.pingcap.com/
    ↪ zh/tidb/stable/tidb-lightning-physical-import-mode
#   当前 "physical" 为实验特性，不建议在生产环境中使用。
import-mode: "logical"
# logical import 针对冲突数据的解决方式：
# - "replace"（默认值）。表示用最新数据替代已有数据。
# - "ignore"。保留已有数据，忽略新数据。
# - "error"。插入重复数据时报错并停止同步任务。
on-duplicate-logical: "replace"
# physical import 针对冲突数据的解决方式：
# - "none"（默认）。对应 TiDB Lightning physical import 冲突数据检测的 "none" 选项
# (https://docs.pingcap.com/zh/tidb/stable/tidb-lightning-physical-import-mode-usage#
    ↪ 冲突数据检测)，
# 表示遇到冲突数据时不进行处理。该模式性能最佳，但下游数据库会遇到数据索引不一致的问题。
# - "manual"。对应 TiDB Lightning physical import 冲突数据检测的 "remove" 选项
# (https://docs.pingcap.com/zh/tidb/stable/tidb-lightning-physical-import-mode-usage#
    ↪ 冲突数据检测)。
# 在遇到冲突数据时将所有相互冲突的数据删除，并记录在 ${meta-schema}_${name}.conflict_error_v1
    ↪  表中。
# 在本配置文件中，会记录在 dm_meta_test.conflict_error_v1 表中。全量导入阶段结束后，任务
# 会暂停并提示用户查询这张表并按照文档进行手动处理。使用 resume-task 命令让任务恢复运行并
# 进入到增量同步阶段。
on-duplicate-physical: "none"
# physical import 用作本地排序的目录位置，该选项的默认值与 dir 配置项一致。具体说明可以参见
    ↪ TiDB Lightning 对存储空间的需求：https://docs.pingcap.com/zh/tidb/stable/tidb-
    ↪ lightning-physical-import-mode#运行环境需求
sorting-dir-physical: "./dumped_data"
# 磁盘空间限制，对应 TiDB Lightning disk-quota 配置。具体说明参见文档：https://docs.pingcap.
    ↪ com/zh/tidb/stable/tidb-lightning-physical-import-mode-usage#磁盘资源配额-从-v620-
    ↪ 版本开始引入
disk-quota-physical: "0"
# physical import 在导入完成一张表后，对每一个表执行 `ADMIN CHECKSUM TABLE <table>`
    ↪ 进行数据校验的配置：
# - "required"（默认值）。表示导入完成后进行数据校验，如果校验失败会让任务暂停，
    ↪ 需要用户手动处理。
# - "optional"。表示导入完成后进行数据校验，如果校验失败会打印 warn 日志，任务不会暂停。
```

```
    # - "off"。表示导入完成后不进行数据校验。
    # Checksum 对比失败通常表示导入异常（数据丢失或数据不一致），因此建议总是开启 Checksum。
    checksum-physical: "required"
    # 配置在 CHECKSUM 结束后是否对所有表执行 `ANALYZE TABLE <table>` 操作。
    # - "required"（默认值）。表示导入完成后进行 ANALYZE 操作，ANALYZE 操作失败时任务暂停，
        ↪ 需要用户手动处理。
    # - "optional"。表示导入完成后进行 ANALYZE 操作，ANALYZE 操作失败时输出警告日志，任务不会暂停
        ↪ 。
    # - "off"。表示导入完成后不进行 ANALYZE 操作。
    # ANALYZE 只影响统计数据，在大部分场景下建议不开启 ANALYZE。
    analyze: "off"
    # Physical Import Mode 向 TiKV 写入 KV 数据的并发度。当 dm-worker 和 TiKV
        ↪ 网络传输速度超过万兆时，可适当增加这个值。
    # range-concurrency: 16
    # Physical Import Mode 向 TiKV 发送 KV 数据时是否启用压缩。目前仅支持 Gzip 压缩算法，可填写 "
        ↪ gzip" 或 "gz"。默认不启用压缩。
    # compress-kv-pairs: ""
    # PD server 的地址，填一个即可。该值为空时，默认使用 TiDB 查询到的 PD 地址信息。
    # pd-addr: "192.168.0.1:2379"

syncers:                            # sync 处理单元的运行配置参数
  global:                           # 配置名称
    worker-count: 16                # 应用已传输到本地的 binlog 的并发线程数量，默认值为 16。
        ↪ 调整此参数不会影响上游拉取日志的并发，但会对下游产生显著压力。
    batch: 100                      # sync 迁移到下游数据库的一个事务批次 SQL 语句数，默认值为
        ↪ 100，建议一般不超过 500。
    enable-ansi-quotes: true        # 若 `session` 中设置 `sql-mode: "ANSI_QUOTES"`，
        ↪ 则需开启此项

    # 设置为 true，则将来自上游的 `INSERT` 改写为 `REPLACE`，将 `UPDATE` 改写为 `DELETE` 与 `
        ↪ REPLACE`，保证在表结构中存在主键或唯一索引的条件下迁移数据时可以重复导入 DML。
    safe-mode: false
    # 自动安全模式的持续时间
    # 如不设置或者设置为 ""，则默认为 `checkpoint-flush-interval`（默认为 30s）的两倍，即 60s。
    # 如设置为 "0s"，则在 DM 自动进入安全模式的时候报错。
    # 如设置为正常值，例如 "1m30s"，则在该任务异常暂停、记录 `safemode_exit_point` 失败、或是 DM
        ↪ 进程异常退出时，把安全模式持续时间调整为 1 分 30 秒。详情可见[自动开启安全模式](https
        ↪ ://docs.pingcap.com/zh/tidb/stable/dm-safe-mode#自动开启)。
    safe-mode-duration: "60s"
    # 设置为 true，DM 会在不增加延迟的情况下，尽可能地将上游对同一条数据的多次操作压缩成一次操作
        ↪ 。
    # 如 INSERT INTO tb(a,b) VALUES(1,1); UPDATE tb SET b=11 WHERE a=1; 会被压缩成 INSERT INTO tb
        ↪ (a,b) VALUES(1,11); 其中 a 为主键
    # 如 UPDATE tb SET b=1 WHERE a=1; UPDATE tb(a,b) SET b=2 WHERE a=1; 会被压缩成 UPDATE tb(a,b)
        ↪  SET b=2 WHERE a=1; 其中 a 为主键
```

```
    # 如 DELETE FROM tb WHERE a=1; INSERT INTO tb(a,b) VALUES(1,1); 会被压缩成 REPLACE INTO tb(a,
        ↪ b) VALUES(1,1); 其中 a 为主键
    compact: false
    # 设置为 true，DM 会尽可能地将多条同类型的语句合并到一条语句中，生成一条带多行数据的 SQL 语句
        ↪ 。
    # 如 INSERT INTO tb(a,b) VALUES(1,1); INSERT INTO tb(a,b) VALUES(2,2); 会变成 INSERT INTO tb(
        ↪ a,b) VALUES(1,1),(2,2);
    # 如 UPDATE tb SET b=11 WHERE a=1; UPDATE tb(a,b) set b=22 WHERE a=2; 会变成 INSERT INTO tb(a
        ↪ ,b) VALUES(1,11),(2,22) ON DUPLICATE KEY UPDATE a=VALUES(a), b=VALUES(b); 其中 a
        ↪ 为主键
    # 如 DELETE FROM tb WHERE a=1; DELETE FROM tb WHERE a=2 会变成 DELETE FROM tb WHERE (a) IN
        ↪ (1),(2)；其中 a 为主键
    multiple-rows: false

validators:              # 增量数据校验的运行配置参数
  global:                # 配置名称
    # full：校验每一行中每一列数据是否正确
    # fast：仅校验这一行是否有成功迁移到下游
    # none：不校验
    mode: full           # 可选填 full，fast 和 none，默认是 none，即不开启校验。
    worker-count: 4      # 后台校验的 validation worker 数量，默认是 4 个
    row-error-delay: 30m # 某一行多久没有校验通过会被标记为 error row，默认是 30 分钟

##### ----------- 实例配置 -----------
mysql-instances:
  -
    source-id: "mysql-replica-01"           # 对应 source.toml 中的 `source-id`
    meta:                                   # `task-mode` 为 `incremental` 且下游数据库的 `
        ↪ checkpoint` 不存在时 binlog 迁移开始的位置；如果 checkpoint 存在，则以 `checkpoint`
        ↪ 为准。如果 `meta` 项和下游数据库的 `checkpoint` 都不存在，则从上游当前最新的 binlog
        ↪ 位置开始迁移
      binlog-name: binlog.000001
      binlog-pos: 4
      binlog-gtid: "03fc0263-28c7-11e7-a653-6c0b84d59f30:1-7041423,05474d3c-28c7-11e7-8352-203
          ↪ db246dd3d:1-170"  # 对于 source 中指定了 `enable-gtid: true` 的增量任务，
          ↪ 需要指定该值

    route-rules: ["route-rule-1", "route-rule-2"]    # 该上游数据库实例匹配的表到下游数据库的
        ↪ table routing 规则名称
    filter-rules: ["filter-rule-1", "filter-rule-2"] # 该上游数据库实例匹配的表的 binlog event
        ↪ filter 规则名称
    block-allow-list:  "bw-rule-1"                   # 该上游数据库实例匹配的表的 block-allow-
        ↪ list 过滤规则名称，如果 DM 版本早于 v2.0.0-beta.2 则使用 black-white-list
    expression-filters: ["even_c"]                   # 使用名为 even_c 的表达式过滤规则
```

```
  mydumper-config-name: "global"          # mydumpers 配置的名称
  loader-config-name: "global"            # loaders 配置的名称
  syncer-config-name: "global"            # syncers 配置的名称
  validator-config-name: "global"         # validators 配置的名称
-
  source-id: "mysql-replica-02"  # 对应 source.toml 中的 `source-id`
  mydumper-thread: 4             # dump 处理单元用于导出数据的线程数量，等同于 mydumpers
      ↪ 配置中的 `threads`，当同时指定它们时 `mydumper-thread` 优先级更高
  loader-thread: 16              # load 处理单元用于导入数据的线程数量，等同于 loaders 配置中的
      ↪  `pool-size`，当同时指定它们时 `loader-thread` 优先级更高。当有多个实例同时向 TiDB
      ↪ 迁移数据时可根据负载情况适当调小该值
  syncer-thread: 16              # sync 处理单元用于复制增量数据的线程数量，等同于 syncers
      ↪ 配置中的 `worker-count`，当同时指定它们时 `syncer-thread` 优先级更高。
      ↪ 当有多个实例同时向 TiDB 迁移数据时可根据负载情况适当调小该值
```

配置顺序

通过上面的配置文件示例，可以看出配置文件总共分为两个部分：全局配置和实例配置，其中全局配置又分为基本信息配置和实例配置，配置顺序如下：

1. 编辑全局配置。
2. 根据全局配置编辑实例配置。

全局配置

任务基本信息配置

配置任务的基本信息，配置项的说明参见以上示例配置文件中的注释。其中 task-mode 需要特殊说明：

task-mode

- 描述：任务模式，可以通过任务模式来指定需要执行的数据迁移工作。
- 值为字符串（full，incremental 或 all）。
  - full：只全量备份上游数据库，然后将数据全量导入到下游数据库。
  - incremental：只通过 binlog 把上游数据库的增量修改复制到下游数据库，可以设置实例配置的 meta 配置项来指定增量复制开始的位置。
  - all：full + incremental。先全量备份上游数据库，将数据全量导入到下游数据库，然后从全量数据备份时导出的位置信息 (binlog position) 开始通过 binlog 增量复制数据到下游数据库。

功能配置集

全局配置主要包含下列功能配置集：

配置项	说明
routes	上游和下游表之间的路由 table routing 规则集。如果上游与下游的库名、表名一致，则不需要配置该项。使用场景及示例配置参见Table Routing

配置项	说明
filters	上游数据库实例匹配的表的 binlog event filter 规则集。如果不需要对 binlog 进行过滤，则不需要配置该项。使用场景及示例配置参见Binlog Event Filter
block-allow-list	该上游数据库实例匹配的表的 block & allow lists 过滤规则集。建议通过该项指定需要迁移的库和表，否则会迁移所有的库和表。使用场景及示例配置参见Block & Allow Lists
mydumpers	dump 处理单元的运行配置参数。如果默认配置可以满足需求，则不需要配置该项，也可以只使用 mydumper-thread 对 thread 配置项单独进行配置。
loaders	load 处理单元的运行配置参数。如果默认配置可以满足需求，则不需要配置该项，也可以只使用 loader-thread 对 pool-size 配置项单独进行配置。
syncers	sync 处理单元的运行配置参数。如果默认配置可以满足需求，则不需要配置该项，也可以只使用 syncer-thread 对 worker-count 配置项单独进行配置。

各个功能配置集的参数及解释参见完整配置文件示例中的注释说明。

实例配置

本小节定义具体的数据迁移子任务，DM 支持从单个或者多个上游 MySQL 实例迁移数据到同一个下游数据库实例。

在该项配置中设置数据迁移子任务中各个功能对应的配置集中的配置名称，关于这些配置项的更多配置细节，参见功能配置集的相关配置项，对应关系如下：

配置项	相关配置项
route-rules	routes
filter-rules	filters
block-allow-list	block-allow-list
mydumper-config-name	mydumpers
loader-config-name	loaders
syncer-config-name	syncers

13.6.9.4.4 DM-master 配置文件介绍

本文介绍 DM-master 的配置文件，包括配置文件示例与配置项说明。

配置文件示例

DM-master 的示例配置文件如下所示：

```
name = "dm-master"
```

```
##### log configuration
log-level = "info"
log-file = "dm-master.log"

##### DM-master listening address
master-addr = ":8261"
advertise-addr = "127.0.0.1:8261"

##### URLs for peer traffic
peer-urls = "http://127.0.0.1:8291"
advertise-peer-urls = "http://127.0.0.1:8291"

##### cluster configuration
initial-cluster = "master1=http://127.0.0.1:8291,master2=http://127.0.0.1:8292,master3=http
    ↪ ://127.0.0.1:8293"
join = ""

ssl-ca = "/path/to/ca.pem"
ssl-cert = "/path/to/cert.pem"
ssl-key = "/path/to/key.pem"
cert-allowed-cn = ["dm"]
```

配置项说明

Global 配置

配置项	说明
name	标识一个 DM-master。
log-level	日志级别：debug、info、warn、error、fatal。默认为 info。
log-file	日志文件，如果不配置，日志会输出到标准输出中。
master-addr	DM-master 服务的地址，可以省略 IP 信息，例如：“:8261”。
advertise-addr	DM-master 向外界宣告的地址。
peer-urls	DM-master 节点的对等 URL。
advertise-peer-urls	DM-master 向外界宣告的对等 URL。默认为 peer-urls 的值。
initial-cluster	初始集群中所有 DM-master 的 advertise-peer-urls 的值。
join	集群里已有的 DM-master 的 advertise-peer-urls 的值。如果是新加入的 DM-master 节点，使用 join 替代 initial-cluster。
ssl-ca	DM-master 组件用于与其它组件连接的 SSL CA 证书所在的路径
ssl-cert	DM-master 组件用于与其它组件连接的 PEM 格式的 X509 证书所在的路径

配置项	说明
ssl-key	DM-master 组件用于与其它组件连接的 PEM 格式的 X509 密钥所在的路径
cert-allowed-cn	证书检查 Common Name 列表

13.6.9.4.5 DM-worker 配置文件介绍

本文介绍 DM-worker 的配置文件，包括配置文件示例与配置项说明。

配置文件示例

```
##### Worker Configuration.

name = "worker1"

##### Log configuration.
log-level = "info"
log-file = "dm-worker.log"

##### DM-worker listen address.
worker-addr = ":8262"
advertise-addr = "127.0.0.1:8262"
join = "http://127.0.0.1:8261,http://127.0.0.1:8361,http://127.0.0.1:8461"

keepalive-ttl = 60
relay-keepalive-ttl = 1800 # 版本 2.0.2 新增
##### relay-dir = "relay_log" # 版本 5.4.0 新增。使用相对路径时注意结合部署、启动方式确认路径位置
    ↪ 。

ssl-ca = "/path/to/ca.pem"
ssl-cert = "/path/to/cert.pem"
ssl-key = "/path/to/key.pem"
cert-allowed-cn = ["dm"]
```

配置项说明

Global 配置

配置项	说明
name	标识一个 DM-worker。
log-level	日志级别：debug、info、warn、error、fatal。默认为 info。
log-file	日志文件，如果不配置日志会输出到标准输出中。
worker-addr	DM-worker 服务的地址，可以省略 IP 信息，例如：“:8262”。
advertise-addr	DM-worker 向外界宣告的地址。

配置项	说明
join	对应一个或多个 DM-master 配置中的master-addr。
keepalive-ttl	当绑定的上游数据源没有启用 relay log 时，DM-worker 向 DM-master 保持存活的周期，单位为秒。默认是 60 秒。
relay-keepalive ↪ -ttl	当绑定的上游数据源启用 relay log 时，DM-worker 向 DM-master 保持存活的周期，单位为秒。默认是 1800 秒。在版本 2.0.2 新增。
relay-dir	当绑定的上游数据源启用 relay log 时，DM-worker 将 relay log 保存在该路径下。该配置优先级比上游数据源配置更高。在版本 5.4.0 新增。
ssl-ca	DM-worker 组件用于与其它组件连接的 SSL CA 证书所在的路径
ssl-cert	DM-worker 组件用于与其它组件连接的 PEM 格式的 X509 证书所在的路径
ssl-key	DM-worker 组件用于与其它组件连接的 PEM 格式的 X509 密钥所在的路径
cert-allowed-cn	证书检查 Common Name 列表

13.6.9.4.6 DM Table Selector

Table Selector 提供了一种基于通配符 来匹配指定 schema/table 的功能。

通配符

table selector 在 schema-pattern/table-pattern 中可以使用以下两个通配符：

- 星号 (*)
  - 匹配零个或者多个字符。例如，doc* 匹配 doc 和 document，但是不匹配 dodo。
  - * 只能放在 pattern 的最后一位，例如，支持 doc*，但是不支持 do*c。
- 问号 (?)
  - 匹配任意一个空字符除外的字符。

匹配规则

- schema-pattern 限制不能为空；
- table-pattern 可以设置为空。设置为空时，将只根据 schema-pattern 对 schema 进行匹配，然后返回匹配结果；
- table-pattern 不为空时，分别根据 schema-pattern 和 table-pattern 进行匹配，两个都匹配则结果为匹配。

使用示例

- 匹配所有库名以 schema_ 开头的 schema 和 table

```
schema-pattern:  "schema_*"
table-pattern:  ""
```

- 匹配所有库名以 schema_ 为前缀，并且表名以 table_ 前缀的表

```
schema-pattern = "schema_*"
table-pattern = "table_*"
```

13.6.9.5 使用 OpenAPI 运维 TiDB Data Migration 集群

TiDB Data Migration (DM) 提供 OpenAPI 功能，你可以通过 OpenAPI 方便地对 DM 集群进行查询和运维操作。OpenAPI 的功能范围和dmctl 工具相当。

如需开启 OpenAPI，可通过以下方法：

- 如果你的 DM 集群是通过二进制直接部署的，则在 DM-master 的配置文件中添加如下配置：

```
openapi = true
```

- 如果你的 DM 集群是通过 TiUP 部署的，则在拓扑文件中添加如下配置：

```
server_configs:
  master:
    openapi: true
```

> 注意：
>
> - DM 提供符合 OpenAPI 3.0.0 标准的 Spec 文档，其中包含了所有 API 的请求参数和返回体，你可自行复制到如 Swagger Editor 等工具中在线预览文档。
> - 部署 DM-master 后，你可访问 http://{master-addr}/api/v1/docs 在线预览文档。

你可以通过 OpenAPI 完成 DM 集群的如下运维操作：

13.6.9.5.1 集群相关 API

- 获取 DM-master 节点信息
- 下线 DM-master 节点
- 获取 DM-worker 节点信息
- 下线 DM-worker 节点

13.6.9.5.2 数据源相关 API

- 创建数据源
- 获取数据源
- 删除数据源
- 更新数据源
- 启用数据源
- 停用数据源
- 获取数据源状态
- 获取数据源列表
- 对数据源开启 relay-log 功能
- 对数据源停止 relay-log 功能
- 清除数据源不需要的 relay-log 文件
- 更改数据源和 DM-worker 的绑定关系
- 获取数据源的数据库名列表
- 获取数据源的指定数据库的表名列表

13.6.9.5.3 同步任务相关 API

- 创建同步任务
- 获取同步任务
- 删除同步任务
- 更新同步任务
- 开始同步任务
- 停止同步任务
- 获取同步任务状态
- 获取同步任务列表
- 获取同步任务的同步规则列表
- 获取同步任务关联数据源的数据库名列表
- 获取同步任务关联数据源的数据表名列表
- 获取同步任务关联数据源的数据表的创建语句
- 更新同步任务关联数据源的数据表的创建语句
- 删除同步任务关联数据源的数据表

本文档以下部分描述当前提供的 API 的具体使用方法。

13.6.9.5.4 API 统一错误格式

对 API 发起的请求后，如发生错误，返回错误信息的格式如下所示：

```
{
    "error_msg": "",
    "error_code": ""
}
```

如上所示，`error_msg` 描述错误信息，`error_code` 则是对应的错误码。

13.6.9.5.5 获取 DM-master 节点信息

该接口是一个同步接口，请求成功会返回对应节点的状态信息。

请求 URI

```
GET /api/v1/cluster/masters
```

使用样例

```
curl -X 'GET' \
  'http://127.0.0.1:8261/api/v1/cluster/masters' \
  -H 'accept: application/json'
```

```
{
  "total": 1,
  "data": [
    {
      "name": "master1",
      "alive": true,
      "leader": true,
      "addr": "127.0.0.1:8261"
    }
  ]
}
```

13.6.9.5.6 下线 DM-master 节点

该接口是一个同步接口，请求成功后返回体的 Status Code 是 204。

请求 URI

```
DELETE /api/v1/cluster/masters/{master-name}
```

使用样例

```
curl -X 'DELETE' \
  'http://127.0.0.1:8261/api/v1/cluster/masters/master1' \
  -H 'accept: */*'
```

13.6.9.5.7 获取 DM-worker 节点信息

该接口是一个同步接口，请求成功会返回对应节点的状态信息。

请求 URI

```
GET /api/v1/cluster/workers
```

使用样例

```
curl -X 'GET' \
  'http://127.0.0.1:8261/api/v1/cluster/workers' \
  -H 'accept: application/json'
```

```
{
  "total": 1,
  "data": [
    {
      "name": "worker1",
      "addr": "127.0.0.1:8261",
      "bound_stage": "bound",
      "bound_source_name": "mysql-01"
    }
  ]
}
```

13.6.9.5.8 下线 DM-worker 节点

该接口是一个同步接口，请求成功后返回体的 Status Code 是 204。

请求 URI

```
DELETE /api/v1/cluster/workers/{worker-name}
```

使用样例

```
curl -X 'DELETE' \
  'http://127.0.0.1:8261/api/v1/cluster/workers/worker1' \
  -H 'accept: */*'
```

13.6.9.5.9 创建数据源

该接口是一个同步接口，请求成功会返回对应数据源信息。

请求 URI

```
POST /api/v1/sources
```

使用样例

```
curl -X 'POST' \
  'http://127.0.0.1:8261/api/v1/sources' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "source_name": "mysql-01",
  "host": "127.0.0.1",
  "port": 3306,
```

```
  "user": "root",
  "password": "123456",
  "enable": true,
  "enable_gtid": false,
  "security": {
    "ssl_ca_content": "",
    "ssl_cert_content": "",
    "ssl_key_content": "",
    "cert_allowed_cn": [
      "string"
    ]
  },
  "purge": {
    "interval": 3600,
    "expires": 0,
    "remain_space": 15
  }
}'
```

```
{
  "source_name": "mysql-01",
  "host": "127.0.0.1",
  "port": 3306,
  "user": "root",
  "password": "123456",
  "enable": true,
  "enable_gtid": false,
  "security": {
    "ssl_ca_content": "",
    "ssl_cert_content": "",
    "ssl_key_content": "",
    "cert_allowed_cn": [
      "string"
    ]
  },
  "purge": {
    "interval": 3600,
    "expires": 0,
    "remain_space": 15
  },
  "status_list": [
    {
      "source_name": "mysql-replica-01",
      "worker_name": "worker-1",
      "relay_status": {
```

```
        "master_binlog": "(mysql-bin.000001, 1979)",
        "master_binlog_gtid": "e9a1fc22-ec08-11e9-b2ac-0242ac110003:1-7849",
        "relay_dir": "./sub_dir",
        "relay_binlog_gtid": "e9a1fc22-ec08-11e9-b2ac-0242ac110003:1-7849",
        "relay_catch_up_master": true,
        "stage": "Running"
      },
      "error_msg": "string"
    }
  ]
}
```

13.6.9.5.10 获取数据源

该接口是一个同步接口，请求成功会返回数据源列表信息。

请求 URI

```
GET /api/v1/sources/{source-name}
```

使用样例

```
curl -X 'GET' \
  'http://127.0.0.1:8261/api/v1/sources/source-1?with_status=true' \
  -H 'accept: application/json'
```

```
{
  "source_name": "mysql-01",
  "host": "127.0.0.1",
  "port": 3306,
  "user": "root",
  "password": "123456",
  "enable_gtid": false,
  "enable": false,
  "flavor": "mysql",
  "task_name_list": [
    "task1"
  ],
  "security": {
    "ssl_ca_content": "",
    "ssl_cert_content": "",
    "ssl_key_content": "",
    "cert_allowed_cn": [
      "string"
    ]
  },
  "purge": {
```

```
    "interval": 3600,
    "expires": 0,
    "remain_space": 15
  },
  "status_list": [
    {
      "source_name": "mysql-replica-01",
      "worker_name": "worker-1",
      "relay_status": {
        "master_binlog": "(mysql-bin.000001, 1979)",
        "master_binlog_gtid": "e9a1fc22-ec08-11e9-b2ac-0242ac110003:1-7849",
        "relay_dir": "./sub_dir",
        "relay_binlog_gtid": "e9a1fc22-ec08-11e9-b2ac-0242ac110003:1-7849",
        "relay_catch_up_master": true,
        "stage": "Running"
      },
      "error_msg": "string"
    }
  ],
  "relay_config": {
    "enable_relay": true,
    "relay_binlog_name": "mysql-bin.000002",
    "relay_binlog_gtid": "e9a1fc22-ec08-11e9-b2ac-0242ac110003:1-7849",
    "relay_dir": "./relay_log"
  }
}
```

13.6.9.5.11 删除数据源

该接口是一个同步接口，请求成功后返回的 Status Code 是 204。

请求 URI

```
DELETE /api/v1/sources/{source-name}
```

使用样例

```
curl -X 'DELETE' \
  'http://127.0.0.1:8261/api/v1/sources/mysql-01?force=true' \
  -H 'accept: application/json'
```

13.6.9.5.12 更新数据源

该接口是一个同步接口，请求成功会返回对应的数据源信息。

> **注意：**
>
> 更新数据源配置时，须确保当前数据源下没有任何正在运行的同步任务。

请求 URI

```
PUT /api/v1/sources/{source-name}
```

使用样例

```
curl -X 'PUT' \
  'http://127.0.0.1:8261/api/v1/sources/mysql-01' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "source": {
    "source_name": "mysql-01",
    "host": "127.0.0.1",
    "port": 3306,
    "user": "root",
    "password": "123456",
    "enable_gtid": false,
    "enable": false,
    "flavor": "mysql",
    "task_name_list": [
      "task1"
    ],
    "security": {
      "ssl_ca_content": "",
      "ssl_cert_content": "",
      "ssl_key_content": "",
      "cert_allowed_cn": [
        "string"
      ]
    },
    "purge": {
      "interval": 3600,
      "expires": 0,
      "remain_space": 15
    },
    "relay_config": {
      "enable_relay": true,
      "relay_binlog_name": "mysql-bin.000002",
      "relay_binlog_gtid": "e9a1fc22-ec08-11e9-b2ac-0242ac110003:1-7849",
      "relay_dir": "./relay_log"
```

```
    }
  }
}'
```

```
{
  "source_name": "mysql-01",
  "host": "127.0.0.1",
  "port": 3306,
  "user": "root",
  "password": "123456",
  "enable": true,
  "enable_gtid": false,
  "security": {
    "ssl_ca_content": "",
    "ssl_cert_content": "",
    "ssl_key_content": "",
    "cert_allowed_cn": [
      "string"
    ]
  },
  "purge": {
    "interval": 3600,
    "expires": 0,
    "remain_space": 15
  }
}
```

13.6.9.5.13 启用数据源

这是一个同步接口，请求成功后会启用此数据源，并批量开始数据迁移任务中依赖该数据源的所有子任务。

请求 URI

`POST /api/v1/sources/{source-name}/enable`

使用样例

```
curl -X 'POST' \
  'http://127.0.0.1:8261/api/v1/sources/mysql-01/enable' \
  -H 'accept: */*' \
  -H 'Content-Type: application/json'
```

13.6.9.5.14 停用数据源

这是一个同步接口，请求成功后会停用此数据源，并批量停止数据迁移任务中依赖该数据源的所有子任务。

请求 URI

```
POST /api/v1/sources/{source-name}/disable
```

使用样例

```
curl -X 'POST' \
  'http://127.0.0.1:8261/api/v1/sources/mysql-01/disable' \
  -H 'accept: */*' \
  -H 'Content-Type: application/json'
```

13.6.9.5.15 获取数据源列表

该接口是一个同步接口，请求成功会返回数据源列表信息。

请求 URI

```
GET /api/v1/sources
```

使用样例

```
curl -X 'GET' \
  'http://127.0.0.1:8261/api/v1/sources?with_status=true' \
  -H 'accept: application/json'
```

```
{
  "data": [
    {
      "enable_gtid": false,
      "host": "127.0.0.1",
      "password": "******",
      "port": 3306,
      "purge": {
        "expires": 0,
        "interval": 3600,
        "remain_space": 15
      },
      "security": null,
      "source_name": "mysql-01",
      "user": "root"
    },
    {
      "enable_gtid": false,
      "host": "127.0.0.1",
      "password": "******",
      "port": 3307,
      "purge": {
        "expires": 0,
        "interval": 3600,
        "remain_space": 15
```

```
      },
      "security": null,
      "source_name": "mysql-02",
      "user": "root"
    }
  ],
  "total": 2
}
```

13.6.9.5.16 获取数据源状态

该接口是一个同步接口，请求成功会返回对应节点的状态信息。

请求 URI

```
GET /api/v1/sources/{source-name}/status
```

使用样例

```
curl -X 'GET' \
  'http://127.0.0.1:8261/api/v1/sources/mysql-replica-01/status' \
  -H 'accept: application/json'
```

```
{
  "total": 1,
  "data": [
    {
      "source_name": "mysql-replica-01",
      "worker_name": "worker-1",
      "relay_status": {
        "master_binlog": "(mysql-bin.000001, 1979)",
        "master_binlog_gtid": "e9a1fc22-ec08-11e9-b2ac-0242ac110003:1-7849",
        "relay_dir": "./sub_dir",
        "relay_binlog_gtid": "e9a1fc22-ec08-11e9-b2ac-0242ac110003:1-7849",
        "relay_catch_up_master": true,
        "stage": "Running"
      },
      "error_msg": "string"
    }
  ]
}
```

13.6.9.5.17 对数据源开启 relay-log 功能

这是一个异步接口，请求成功的 Status Code 是 200，可通过获取数据源状态接口获取最新的状态。

请求 URI

```
POST /api/v1/sources/{source-name}/relay/enable
```

使用样例

```
curl -X 'POST' \
  'http://127.0.0.1:8261/api/v1/sources/mysql-01/relay/enable' \
  -H 'accept: */*' \
  -H 'Content-Type: application/json' \
  -d '{
  "worker_name_list": [
    "worker-1"
  ],
  "relay_binlog_name": "mysql-bin.000002",
  "relay_binlog_gtid": "e9a1fc22-ec08-11e9-b2ac-0242ac110003:1-7849",
  "relay_dir": "./relay_log"
}'
```

13.6.9.5.18 对数据源停止 relay-log 功能

这是一个异步接口，请求成功的 Status Code 是 200，可通过获取数据源状态接口获取最新的状态。

请求 URI

```
POST /api/v1/sources/{source-name}/relay/disable
```

使用样例

```
curl -X 'POST' \
  'http://127.0.0.1:8261/api/v1/sources/mysql-01/relay/disable' \
  -H 'accept: */*' \
  -H 'Content-Type: application/json' \
  -d '{
  "worker_name_list": [
    "worker-1"
  ]
}'
```

13.6.9.5.19 清除数据源不需要的 relay-log 文件

这是一个异步接口，请求成功的 Status Code 是 200，可通过获取数据源状态接口获取最新的状态。

请求 URI

```
POST /api/v1/sources/{source-name}/relay/purge
```

使用样例

```
curl -X 'POST' \
  'http://127.0.0.1:8261/api/v1/sources/mysql-01/relay/purge' \
  -H 'accept: */*' \
```

```
  -H 'Content-Type: application/json' \
  -d '{
  "relay_binlog_name": "mysql-bin.000002",
  "relay_dir": "string"
}'
```

13.6.9.5.20 更改数据源和 DM-worker 的绑定关系

这是一个异步接口，请求成功的 Status Code 是 200，可通过获取 DM-worker 节点信息接口获取最新的状态。

请求 URI

```
POST /api/v1/sources/{source-name}/transfer
```

使用样例

```
curl -X 'POST' \
  'http://127.0.0.1:8261/api/v1/sources/mysql-01/transfer' \
  -H 'accept: */*' \
  -H 'Content-Type: application/json' \
  -d '{
  "worker_name": "worker-1"
}'
```

13.6.9.5.21 获取数据源的数据库名列表

该接口是一个同步接口，请求成功会返回对应的列表。

请求 URI

```
GET /api/v1/sources/{source-name}/schemas
```

使用样例

```
curl -X 'GET' \
  'http://127.0.0.1:8261/api/v1/sources/source-1/schemas' \
  -H 'accept: application/json'
```

```
[
  "db1"
]
```

13.6.9.5.22 获取数据源的指定数据库的表名列表

该接口是一个同步接口，请求成功会返回对应的列表。

请求 URI

```
GET /api/v1/sources/{source-name}/schemas/{schema-name}
```

使用样例

```
curl -X 'GET' \
  'http://127.0.0.1:8261/api/v1/sources/source-1/schemas/db1' \
  -H 'accept: application/json'
```

```
[
  "table1"
]
```

13.6.9.5.23 创建同步任务

这是一个同步接口，请求成功的 Status Code 是 200，请求成功会返回对应的同步任务信息。

请求 URI

```
POST /api/v1/tasks
```

使用样例

```
curl -X 'POST' \
  'http://127.0.0.1:8261/api/v1/tasks' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "task": {
    "name": "task-1",
    "task_mode": "all",
    "shard_mode": "pessimistic",
    "meta_schema": "dm-meta",
    "enhance_online_schema_change": true,
    "on_duplicate": "overwrite",
    "target_config": {
      "host": "127.0.0.1",
      "port": 3306,
      "user": "root",
      "password": "123456",
      "security": {
        "ssl_ca_content": "",
        "ssl_cert_content": "",
        "ssl_key_content": "",
        "cert_allowed_cn": [
          "string"
        ]
      }
    },
    "binlog_filter_rule": {
      "rule-1": {
```

```
      "ignore_event": [
        "all dml"
      ],
      "ignore_sql": [
        "^Drop"
      ]
    },
    "rule-2": {
      "ignore_event": [
        "all dml"
      ],
      "ignore_sql": [
        "^Drop"
      ]
    },
    "rule-3": {
      "ignore_event": [
        "all dml"
      ],
      "ignore_sql": [
        "^Drop"
      ]
    }
  },
  "table_migrate_rule": [
    {
      "source": {
        "source_name": "source-name",
        "schema": "db-*",
        "table": "tb-*"
      },
      "target": {
        "schema": "db1",
        "table": "tb1"
      },
      "binlog_filter_rule": [
        "rule-1",
        "rule-2",
        "rule-3",
      ]
    }
  ],
  "source_config": {
    "full_migrate_conf": {
      "export_threads": 4,
```

```
        "import_threads": 16,
        "data_dir": "./exported_data",
        "consistency": "auto"
      },
      "incr_migrate_conf": {
        "repl_threads": 16,
        "repl_batch": 100
      },
      "source_conf": [
        {
          "source_name": "mysql-replica-01",
          "binlog_name": "binlog.000001",
          "binlog_pos": 4,
          "binlog_gtid": "03fc0263-28c7-11e7-a653-6c0b84d59f30:1-7041423,05474d3c-28c7-11e7
              ↪ -8352-203db246dd3d:1-170"
        }
      ]
    }
  }
}'
```

```
{
  "name": "task-1",
  "task_mode": "all",
  "shard_mode": "pessimistic",
  "meta_schema": "dm-meta",
  "enhance_online_schema_change": true,
  "on_duplicate": "overwrite",
  "target_config": {
    "host": "127.0.0.1",
    "port": 3306,
    "user": "root",
    "password": "123456",
    "security": {
      "ssl_ca_content": "",
      "ssl_cert_content": "",
      "ssl_key_content": "",
      "cert_allowed_cn": [
        "string"
      ]
    }
  },
  "binlog_filter_rule": {
    "rule-1": {
      "ignore_event": [
```

```
        "all dml"
      ],
      "ignore_sql": [
        "^Drop"
      ]
    },
    "rule-2": {
      "ignore_event": [
        "all dml"
      ],
      "ignore_sql": [
        "^Drop"
      ]
    },
    "rule-3": {
      "ignore_event": [
        "all dml"
      ],
      "ignore_sql": [
        "^Drop"
      ]
    }
  },
  "table_migrate_rule": [
    {
      "source": {
        "source_name": "source-name",
        "schema": "db-*",
        "table": "tb-*"
      },
      "target": {
        "schema": "db1",
        "table": "tb1"
      },
      "binlog_filter_rule": [
        "rule-1",
        "rule-2",
        "rule-3",
      ]
    }
  ],
  "source_config": {
    "full_migrate_conf": {
      "export_threads": 4,
      "import_threads": 16,
```

```
      "data_dir": "./exported_data",
      "consistency": "auto"
    },
    "incr_migrate_conf": {
      "repl_threads": 16,
      "repl_batch": 100
    },
    "source_conf": [
      {
        "source_name": "mysql-replica-01",
        "binlog_name": "binlog.000001",
        "binlog_pos": 4,
        "binlog_gtid": "03fc0263-28c7-11e7-a653-6c0b84d59f30:1-7041423,05474d3c-28c7-11e7
            ↪ -8352-203db246dd3d:1-170"
      }
    ]
  }
}
```

13.6.9.5.24 获取同步任务

这是一个同步接口，请求成功的 Status Code 是 200。

请求 URI

```
GET /api/v1/tasks/{task-name}?with_status=true
```

使用样例

```
curl -X 'GET' \
  'http://127.0.0.1:8261/api/v1/tasks/task-1?with_status=true' \
  -H 'accept: application/json'
```

```
{
  "name": "task-1",
  "task_mode": "all",
  "shard_mode": "pessimistic",
  "meta_schema": "dm-meta",
  "enhance_online_schema_change": true,
  "on_duplicate": "overwrite",
  "target_config": {
    "host": "127.0.0.1",
    "port": 3306,
    "user": "root",
    "password": "123456",
    "security": {
      "ssl_ca_content": "",
```

```
    "ssl_cert_content": "",
    "ssl_key_content": "",
    "cert_allowed_cn": [
      "string"
    ]
  }
},
"binlog_filter_rule": {
  "rule-1": {
    "ignore_event": [
      "all dml"
    ],
    "ignore_sql": [
      "^Drop"
    ]
  },
  "rule-2": {
    "ignore_event": [
      "all dml"
    ],
    "ignore_sql": [
      "^Drop"
    ]
  },
  "rule-3": {
    "ignore_event": [
      "all dml"
    ],
    "ignore_sql": [
      "^Drop"
    ]
  }
},
"table_migrate_rule": [
  {
    "source": {
      "source_name": "source-name",
      "schema": "db-*",
      "table": "tb-*"
    },
    "target": {
      "schema": "db1",
      "table": "tb1"
    },
    "binlog_filter_rule": [
```

```
        "rule-1",
        "rule-2",
        "rule-3",
      ]
    }
  ],
  "source_config": {
    "full_migrate_conf": {
      "export_threads": 4,
      "import_threads": 16,
      "data_dir": "./exported_data",
      "consistency": "auto"
    },
    "incr_migrate_conf": {
      "repl_threads": 16,
      "repl_batch": 100
    },
    "source_conf": [
      {
        "source_name": "mysql-replica-01",
        "binlog_name": "binlog.000001",
        "binlog_pos": 4,
        "binlog_gtid": "03fc0263-28c7-11e7-a653-6c0b84d59f30:1-7041423,05474d3c-28c7-11e7
            ↪ -8352-203db246dd3d:1-170"
      }
    ]
  }
}
```

13.6.9.5.25 删除同步任务

该接口是一个同步接口，请求成功后返回的 Status Code 是 204。

请求 URI

```
DELETE /api/v1/tasks/{task-name}
```

使用样例

```
curl -X 'DELETE' \
  'http://127.0.0.1:8261/api/v1/tasks/task-1' \
  -H 'accept: application/json'
```

13.6.9.5.26 更新同步任务

该接口是一个同步接口，请求成功会返回对应同步任务的信息。

> **注意：**
>
> 更新同步任务配置时，须确保该任务处于暂停状态，并已经运行到增量同步的阶段，且仅有部分字段可以更新。

请求 URI

```
PUT /api/v1/tasks/{task-name}
```

使用样例

```
curl -X 'PUT' \
  'http://127.0.0.1:8261/api/v1/tasks/task-1' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "task": {
    "name": "task-1",
    "task_mode": "all",
    "shard_mode": "pessimistic",
    "meta_schema": "dm-meta",
    "enhance_online_schema_change": true,
    "on_duplicate": "overwrite",
    "target_config": {
      "host": "127.0.0.1",
      "port": 3306,
      "user": "root",
      "password": "123456",
      "security": {
        "ssl_ca_content": "",
        "ssl_cert_content": "",
        "ssl_key_content": "",
        "cert_allowed_cn": [
          "string"
        ]
      }
    },
    "binlog_filter_rule": {
      "rule-1": {
        "ignore_event": [
          "all dml"
        ],
        "ignore_sql": [
          "^Drop"
        ]
```

```
    },
    "rule-2": {
      "ignore_event": [
        "all dml"
      ],
      "ignore_sql": [
        "^Drop"
      ]
    },
    "rule-3": {
      "ignore_event": [
        "all dml"
      ],
      "ignore_sql": [
        "^Drop"
      ]
    }
  },
  "table_migrate_rule": [
    {
      "source": {
        "source_name": "source-name",
        "schema": "db-*",
        "table": "tb-*"
      },
      "target": {
        "schema": "db1",
        "table": "tb1"
      },
      "binlog_filter_rule": [
        "rule-1",
        "rule-2",
        "rule-3",
      ]
    }
  ],
  "source_config": {
    "full_migrate_conf": {
      "export_threads": 4,
      "import_threads": 16,
      "data_dir": "./exported_data",
      "consistency": "auto"
    },
    "incr_migrate_conf": {
      "repl_threads": 16,
```

```
        "repl_batch": 100
      },
      "source_conf": [
        {
          "source_name": "mysql-replica-01",
          "binlog_name": "binlog.000001",
          "binlog_pos": 4,
          "binlog_gtid": "03fc0263-28c7-11e7-a653-6c0b84d59f30:1-7041423,05474d3c-28c7-11e7
              ↪ -8352-203db246dd3d:1-170"
        }
      ]
    }
  }
}'
```

```
{
  "name": "task-1",
  "task_mode": "all",
  "shard_mode": "pessimistic",
  "meta_schema": "dm-meta",
  "enhance_online_schema_change": true,
  "on_duplicate": "overwrite",
  "target_config": {
    "host": "127.0.0.1",
    "port": 3306,
    "user": "root",
    "password": "123456",
    "security": {
      "ssl_ca_content": "",
      "ssl_cert_content": "",
      "ssl_key_content": "",
      "cert_allowed_cn": [
        "string"
      ]
    }
  },
  "binlog_filter_rule": {
    "rule-1": {
      "ignore_event": [
        "all dml"
      ],
      "ignore_sql": [
        "^Drop"
      ]
    },
```

```
  "rule-2": {
    "ignore_event": [
      "all dml"
    ],
    "ignore_sql": [
      "^Drop"
    ]
  },
  "rule-3": {
    "ignore_event": [
      "all dml"
    ],
    "ignore_sql": [
      "^Drop"
    ]
  }
},
"table_migrate_rule": [
  {
    "source": {
      "source_name": "source-name",
      "schema": "db-*",
      "table": "tb-*"
    },
    "target": {
      "schema": "db1",
      "table": "tb1"
    },
    "binlog_filter_rule": [
      "rule-1",
      "rule-2",
      "rule-3",
    ]
  }
],
"source_config": {
  "full_migrate_conf": {
    "export_threads": 4,
    "import_threads": 16,
    "data_dir": "./exported_data",
    "consistency": "auto"
  },
  "incr_migrate_conf": {
    "repl_threads": 16,
    "repl_batch": 100
```

```
    },
    "source_conf": [
      {
        "source_name": "mysql-replica-01",
        "binlog_name": "binlog.000001",
        "binlog_pos": 4,
        "binlog_gtid": "03fc0263-28c7-11e7-a653-6c0b84d59f30:1-7041423,05474d3c-28c7-11e7
            ↪ -8352-203db246dd3d:1-170"
      }
    ]
  }
}
```

13.6.9.5.27 开始同步任务

这是一个异步接口，请求成功的 Status Code 是 200。可通过获取同步任务状态接口获取最新的任务状态。

请求 URI

```
POST /api/v1/tasks/{task-name}/start
```

使用样例

```
curl -X 'POST' \
  'http://127.0.0.1:8261/api/v1/tasks/task-1/start' \
  -H 'accept: */*'
```

13.6.9.5.28 停止同步任务

这是一个异步接口，请求成功的 Status Code 是 200，可通过获取同步任务状态接口获取最新的任务状态。

请求 URI

```
POST /api/v1/tasks/{task-name}/stop
```

使用样例

```
curl -X 'POST' \
  'http://127.0.0.1:8261/api/v1/tasks/task-1/stop' \
  -H 'accept: */*'
```

13.6.9.5.29 获取同步任务状态

该接口是一个同步接口，请求成功会返回对应节点的状态信息。

请求 URI

```
GET /api/v1/tasks/task-1/status
```

使用样例

```
curl -X 'GET' \
  'http://127.0.0.1:8261/api/v1/tasks/task-1/status?stage=running' \
  -H 'accept: application/json'
```


```
{
  "total": 1,
  "data": [
    {
      "name": "string",
      "source_name": "string",
      "worker_name": "string",
      "stage": "runing",
      "unit": "sync",
      "unresolved_ddl_lock_id": "string",
      "load_status": {
        "finished_bytes": 0,
        "total_bytes": 0,
        "progress": "string",
        "meta_binlog": "string",
        "meta_binlog_gtid": "string"
      },
      "sync_status": {
        "total_events": 0,
        "total_tps": 0,
        "recent_tps": 0,
        "master_binlog": "string",
        "master_binlog_gtid": "string",
        "syncer_binlog": "string",
        "syncer_binlog_gtid": "string",
        "blocking_ddls": [
          "string"
        ],
        "unresolved_groups": [
          {
            "target": "string",
            "ddl_list": [
              "string"
            ],
            "first_location": "string",
            "synced": [
              "string"
            ],
            "unsynced": [
              "string"
```

```
            ]
          }
        ],
        "synced": true,
        "binlog_type": "string",
        "seconds_behind_master": 0
      }
    }
  ]
}
```

13.6.9.5.30 获取同步任务列表

该接口是一个同步接口，请求成功会返回对应的同步任务列表。

请求 URI

```
GET /api/v1/tasks
```

使用样例

```
curl -X 'GET' \
  'http://127.0.0.1:8261/api/v1/tasks' \
  -H 'accept: application/json'
```

```
{
  "total": 2,
  "data": [
    {
      "name": "task-1",
      "task_mode": "all",
      "shard_mode": "pessimistic",
      "meta_schema": "dm-meta",
      "enhance_online_schema_change": true,
      "on_duplicate": "overwrite",
      "target_config": {
        "host": "127.0.0.1",
        "port": 3306,
        "user": "root",
        "password": "123456",
        "security": {
          "ssl_ca_content": "",
          "ssl_cert_content": "",
          "ssl_key_content": "",
          "cert_allowed_cn": [
            "string"
          ]
```

```
    }
  },
  "binlog_filter_rule": {
    "rule-1": {
      "ignore_event": [
        "all dml"
      ],
      "ignore_sql": [
        "^Drop"
      ]
    },
    "rule-2": {
      "ignore_event": [
        "all dml"
      ],
      "ignore_sql": [
        "^Drop"
      ]
    },
    "rule-3": {
      "ignore_event": [
        "all dml"
      ],
      "ignore_sql": [
        "^Drop"
      ]
    }
  },
  "table_migrate_rule": [
    {
      "source": {
        "source_name": "source-name",
        "schema": "db-*",
        "table": "tb-*"
      },
      "target": {
        "schema": "db1",
        "table": "tb1"
      },
      "binlog_filter_rule": [
        "rule-1",
        "rule-2",
        "rule-3",
      ]
    }
```

```
      ],
      "source_config": {
        "full_migrate_conf": {
          "export_threads": 4,
          "import_threads": 16,
          "data_dir": "./exported_data",
          "consistency": "auto"
        },
        "incr_migrate_conf": {
          "repl_threads": 16,
          "repl_batch": 100
        },
        "source_conf": [
          {
            "source_name": "mysql-replica-01",
            "binlog_name": "binlog.000001",
            "binlog_pos": 4,
            "binlog_gtid": "03fc0263-28c7-11e7-a653-6c0b84d59f30:1-7041423,05474d3c-28c7-11e7
                ↪ -8352-203db246dd3d:1-170"
          }
        ]
      }
    }
  ]
}
```

13.6.9.5.31 获取同步任务的同步规则列表

该接口是一个同步接口，请求成功会返回对应同步任务的同步规则列表。

请求 URI

```
GET /api/v1/tasks/{task-name}/sources/{source-name}/migrate_targets
```

使用样例

```
curl -X 'GET' \
  'http://127.0.0.1:8261/api/v1/tasks/task-1/sources/source-1/migrate_targets' \
  -H 'accept: application/json'
```

```
{
  "total": 0,
  "data": [
    {
      "source_schema": "db1",
      "source_table": "tb1",
      "target_schema": "db1",
```

```
      "target_table": "tb1"
    }
  ]
}
```

13.6.9.5.32 获取同步任务关联数据源的数据库名列表

该接口是一个同步接口，请求成功会返回对应的列表。

请求 URI

```
GET /api/v1/tasks/{task-name}/sources/{source-name}/schemas
```

使用样例

```
curl -X 'GET' \
  'http://127.0.0.1:8261/api/v1/tasks/task-1/sources/source-1/schemas' \
  -H 'accept: application/json'
```

```
[
  "db1"
]
```

13.6.9.5.33 获取同步任务关联数据源的数据表名列表

该接口是一个同步接口，请求成功会返回对应的列表。

请求 URI

```
GET /api/v1/tasks/{task-name}/sources/{source-name}/schemas/{schema-name}
```

使用样例

```
curl -X 'GET' \
  'http://127.0.0.1:8261/api/v1/tasks/task-1/sources/source-1/schemas/db1' \
  -H 'accept: application/json'
```

```
[
  "table1"
]
```

13.6.9.5.34 获取同步任务关联数据源的数据表的创建语句

该接口是一个同步接口，请求成功会返回对应的创建语句。

请求 URI

```
GET /api/v1/tasks/{task-name}/sources/{source-name}/schemas/{schema-name}/{table-name}
```

使用样例

```
curl -X 'GET' \
  'http://127.0.0.1:8261/api/v1/tasks/task-1/sources/source-1/schemas/db1/table1' \
  -H 'accept: application/json'
```

```
{
  "schema_name": "db1",
  "table_name": "table1",
  "schema_create_sql": "CREATE TABLE `t1` (`id` int(11) NOT NULL AUTO_INCREMENT,PRIMARY KEY (`id
      ↪ `) /*T![clustered_index] CLUSTERED */) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=
      ↪ utf8mb4_bin"
}
```

13.6.9.5.35 更新同步任务关联数据源的数据表的创建语句

该接口是一个同步接口，返回体的 Status Code 是 200。

请求 URI

```
POST /api/v1/tasks/{task-name}/sources/{source-name}/schemas/{schema-name}/{table-name}
```

使用样例

```
curl -X 'PUT' \
  'http://127.0.0.1:8261/api/v1/tasks/task-1/sources/task-1/schemas/db1/table1' \
  -H 'accept: */*' \
  -H 'Content-Type: application/json' \
  -d '{
  "sql_content": "CREATE TABLE `t1` ( `c1` int(11) DEFAULT NULL, `c2` int(11) DEFAULT NULL, `c3`
      ↪ int(11) DEFAULT NULL) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin;",
  "flush": true,
  "sync": true
}'
```

13.6.9.5.36 删除同步任务关联数据源的数据表

该接口是一个同步接口，返回体的 Status Code 是 200。

请求 URI

```
DELETE /api/v1/tasks/{task-name}/sources/{source-name}/schemas/{schema-name}/{table-name}
```

使用样例

```
curl -X 'DELETE' \
  'http://127.0.0.1:8261/api/v1/tasks/task-1/sources/source-1/schemas/db1/table1' \
  -H 'accept: */*'
```

13.6.9.6 TiDB Data Migration 兼容性目录

DM 数据同步软件支持从不同类型的数据源迁移到 TiDB 集群。针对各种数据源类型，产品支持程度可以分为四个级别：

- 正式支持：该场景经过验证，并且通过完整的测试流程。
- 实验支持：虽然通过部分验证，但测试尚未覆盖所有预设场景或用户较少，存在少量场景下可能出错的风险。
- 未测试：DM 在迭代过程中尽量保证 MySQL 协议的兼容性，但由于资源限制，无法测试所有 MySQL 衍生版本。因此虽然技术原理上兼容，但是并未经完整测试，需要使用前自行验证。
- 不兼容：已发现明确不兼容的情况，不建议在生产环境中使用。

13.6.9.6.1 数据源

数据源	级别	备注
MySQL ≤ 5.5	未测试	
MySQL 5.6	正式支持	
MySQL 5.7	正式支持	
MySQL 8.0	实验支持	
MariaDB < 10.1.2	不兼容	时间类型的 binlog 不兼容
MariaDB 10.1.2 ~ 10.5.10	实验支持	
MariaDB > 10.5.10	不兼容	检查环节存在权限报错

13.6.9.6.2 目标数据库

> **警告：**
>
> 不建议使用 DM 5.3.0，因为当使用 GTID 同步且未开启 Relay log 的情况下，低概率会出现数据不同步。

目标数据库	级别	DM 版本
TiDB 6.0	正式支持	最低 5.3.1
TiDB 5.4	正式支持	最低 5.3.1
TiDB 5.3	正式支持	最低 5.3.1
TiDB 5.2	正式支持	最低 2.0.7，建议 5.4 版本
TiDB 5.1	正式支持	最低 2.0.4，建议 5.4 版本
TiDB 5.0	正式支持	最低 2.0.4，建议 5.4 版本
TiDB 4.x	正式支持	最低 2.0.1，建议 2.0.7 版本
TiDB 3.x	正式支持	最低 2.0.1，建议 2.0.7 版本
MySQL	实验支持	
MariaDB	实验支持	

13.6.9.7 安全

13.6.9.7.1 为 DM 的连接开启加密传输

本文介绍如何为 DM 的连接开启加密传输，包括 DM-master，DM-worker，dmctl 组件之间的连接以及 DM 组件与上下游数据库之间的连接。

为 DM-master，DM-worker，dmctl 组件之间的连接开启加密传输

本节介绍如何为 DM-master，DM-worker，dmctl 组件之间的连接开启加密传输。

配置开启加密传输

1. 准备证书。

   推荐为 DM-master、DM-worker 分别准备一个 Server 证书，并保证可以相互验证，而 dmctl 工具则可选择共用 Client 证书。

   有多种工具可以生成自签名证书，如 `openssl`，`cfssl` 及 `easy-rsa` 等基于 `openssl` 的工具。

   这里提供一个使用 `openssl` 生成证书的示例：生成自签名证书。

2. 配置证书。

> 注意：
> DM-master、DM-worker 与 dmctl 三个组件可使用同一套证书。

- DM-master

  在 DM-master 配置文件或命令行参数中设置：

```
ssl-ca = "/path/to/ca.pem"
ssl-cert = "/path/to/master-cert.pem"
ssl-key = "/path/to/master-key.pem"
```

- DM-worker

  在 DM-worker 配置文件或命令行参数中设置：

```
ssl-ca = "/path/to/ca.pem"
ssl-cert = "/path/to/worker-cert.pem"
ssl-key = "/path/to/worker-key.pem"
```

- dmctl

  若 DM 集群各个组件间开启加密传输后，在使用 dmctl 工具连接集群时，需要指定 Client 证书，示例如下：

```
./dmctl --master-addr=127.0.0.1:8261 --ssl-ca /path/to/ca.pem --ssl-cert /path/to/
    ↪ client-cert.pem --ssl-key /path/to/client-key.pem
```

认证组件调用者身份

通常被调用者除了校验调用者提供的密钥、证书和 CA 有效性外，还需要校验调用者身份以防止拥有有效证书的非法访问者进行访问（例如：DM-worker 只能被 DM-master 访问，需阻止拥有合法证书但非 DM-master 的其他访问者访问 DM-worker）。

如希望进行组件调用者身份认证，需要在生成证书时通过 Common Name (CN) 标识证书使用者身份，并在被调用者配置检查证书 Common Name 列表时检查调用者身份。

- DM-master

  在 config 文件或命令行参数中设置：

```
cert-allowed-cn = ["dm"]
```

- DM-worker

  在 config 文件或命令行参数中设置：

```
cert-allowed-cn = ["dm"]
```

证书重加载

DM-master、DM-worker 和 dmctl 都会在每次新建相互通讯的连接时重新读取当前的证书和密钥文件内容，实现证书和密钥的重加载。

当 ssl-ca、ssl-cert 或 ssl-key 的文件内容更新后，可通过重启 DM 组件使其重新加载证书与密钥内容并重新建立连接。

DM 组件与上下游数据库之间的连接开启加密传输

本节介绍如何为 DM 组件与上下游数据库之间的连接开启加密传输。

为上游数据库连接开启加密传输

1. 配置上游数据库，启用加密连接支持并设置 Server 证书，具体可参考 Using encrypted connections
2. 在 source 配置文件中设置 MySQL Client 证书：

> **注意：**
> 请确保所有 DM-master 与 DM-worker 组件能通过指定路径读取到证书与密钥文件的内容。

```
from:
    security:
        ssl-ca: "/path/to/mysql-ca.pem"
        ssl-cert: "/path/to/mysql-client-cert.pem"
        ssl-key: "/path/to/mysql-client-key.pem"
```

为下游 TiDB 连接开启加密传输

1. 配置下游 TiDB 启用加密连接支持，具体可参考配置 TiDB 启用加密连接支持
2. 在 task 配置文件中设置 TiDB Client 证书：

> 注意：
> 请确保所有 DM-master 与 DM-worker 组件能通过指定路径读取到证书与密钥文件的内容。

```
target-database:
    security:
        ssl-ca: "/path/to/tidb-ca.pem"
        ssl-cert: "/path/to/tidb-client-cert.pem"
        ssl-key: "/path/to/tidb-client-key.pem"
```

13.6.9.7.2 TiDB Data Migration 生成自签名证书

本文档提供使用 openssl 为 TiDB Data Migration (DM) 生成自签名证书的一个示例，你也可以根据自己的需求生成符合要求的证书和密钥。

假设实例集群拓扑如下：

Name	Host IP	Services
node1	172.16.10.11	DM-master1
node2	172.16.10.12	DM-master2
node3	172.16.10.13	DM-master3
node4	172.16.10.14	DM-worker1
node5	172.16.10.15	DM-worker2
node6	172.16.10.16	DM-worker3

安装 OpenSSL

对于 Debian 或 Ubuntu 操作系统：

```
apt install openssl
```

对于 RedHat 或 CentOS 操作系统：

```
yum install openssl
```

也可以参考 OpenSSL 官方的下载文档进行安装。

生成 CA 证书

CA 的作用是签发证书。实际情况中，请联系你的管理员签发证书或者使用信任的 CA 机构。CA 会管理多个证书对，这里只需生成原始的一对证书，步骤如下：

1. 生成 CA 密钥：

```
openssl genrsa -out ca-key.pem 4096
```

2. 生成 CA 证书：

```
openssl req -new -x509 -days 1000 -key ca-key.pem -out ca.pem
```

3. 验证 CA 证书：

```
openssl x509 -text -in ca.pem -noout
```

签发各个组件的证书

集群中可能使用到的证书

- master certificate 由 DM-master 使用，为其他组件验证 DM-master 身份。
- worker certificate 由 DM-worker 使用，为其他组件验证 DM-worker 身份。
- client certificate 由 dmctl 使用，用于 DM-master、 DM-worker 验证客户端。

为 DM-master 签发证书

给 DM-master 实例签发证书的步骤如下：

1. 生成该证书对应的私钥：

```
openssl genrsa -out master-key.pem 2048
```

2. 拷贝一份 OpenSSL 的配置模板文件。

   模板文件可能存在多个位置，请以实际位置为准：

```
cp /usr/lib/ssl/openssl.cnf .
```

   如果不知道实际位置，请在根目录下查找：

```
find / -name openssl.cnf
```

3. 编辑 `openssl.cnf`，在 `[ req ]` 字段下加入 `req_extensions = v3_req`，然后在 `[ v3_req ]` 字段下加入 `subjectAltName = @alt_names`。最后新建一个字段，根据前述的集群拓扑编辑 `Subject Alternative Name (SAN)` 的信息：

```
[ alt_names ]
IP.1 = 127.0.0.1
IP.2 = 172.16.10.11
IP.3 = 172.16.10.12
IP.4 = 172.16.10.13
```

   目前支持以下 SAN 检查项：

   - IP

- DNS
- URI

> **注意：**
>
> 如果要使用 0.0.0.0 等特殊 IP 用于连接通讯，也需要将其加入到 alt_names 中。

4. 保存 openssl.cnf 文件后，生成证书请求文件（在这一步中提供 Common Name (e.g. server FQDN or YOUR name)[]: 输入时，可以为该证书指定 Common Name (CN)，如 dm。其作用是让服务端验证接入的客户端的身份，各个组件默认不会开启验证，需要在配置文件中启用该功能才生效）：

```
openssl req -new -key master-key.pem -out master-cert.pem -config openssl.cnf
```

5. 签发生成证书：

```
openssl x509 -req -days 365 -CA ca.pem -CAkey ca-key.pem -CAcreateserial -in master-cert.pem
    -out master-cert.pem -extensions v3_req -extfile openssl.cnf
```

6. 验证证书携带 SAN 字段信息（可选）：

```
openssl x509 -text -in master-cert.pem -noout
```

7. 确认在当前目录下得到如下文件：

```
ca.pem
master-cert.pem
master-key.pem
```

> **注意：**
>
> 为 DM-worker 组件签发证书的过程类似，此文档不再赘述。

为 dmctl 签发证书

为客户端签发证书的步骤如下。

1. 生成该证书对应的私钥：

```
openssl genrsa -out client-key.pem 2048
```

2. 生成证书请求文件（在这一步也可以为该证书指定 Common Name，其作用是让服务端验证接入的客户端的身份，默认不会开启对各个组件的验证，需要在配置文件中启用该功能才生效）

```
openssl req -new -key client-key.pem -out client-cert.pem
```

3. 签发生成证书：

```
openssl x509 -req -days 365 -CA ca.pem -CAkey ca-key.pem -CAcreateserial -in client-cert.pem
    ↪  -out client-cert.pem
```

13.6.9.8 监控告警

13.6.9.8.1 DM 监控指标

使用 TiUP 部署 DM 集群的时候，会默认部署一套监控系统。

Task

在 Grafana dashboard 中，DM 默认名称为 DM-task。

Overview

overview 下包含运行当前选定 task 的所有 DM-worker/master instance/source 的部分监控指标。当前默认告警规则只针对于单个 DM-worker/master instance/source。

metric 名称	说明	告警说明	告警级别
task state	迁移子任务的状态	N/A	N/A
storage capacity	relay log 占有的磁盘的总容量	N/A	N/A
storage remain	relay log 占有的磁盘的剩余可用容量	N/A	N/A
binlog file gap between master and relay	relay 与上游 master 相比落后的 binlog file 个数	N/A	N/A
load progress	load unit 导入过程的进度百分比，值变化范围为：0% - 100%	N/A	N/A
binlog file gap between master and syncer	与上游 master 相比 binlog replication unit 落后的 binlog file 个数	N/A	N/A

metric 名称	说明	告警说明	告警级别
shard lock resolving	当前子任务是否正在等待 shard DDL 迁移，大于 0 表示正在等待迁移	N/A	N/A

Operate error

metric 名称	说明	告警说明	告警级别
before any operate error	在进行操作之前出错的次数	N/A	N/A
source bound error	数据源绑定操作出错次数	N/A	N/A
start error	子任务启动的出错次数	N/A	N/A
pause error	子任务暂停的出错次数	N/A	N/A
resume error	子任务恢复的出错次数	N/A	N/A
auto-resume error	子任务自动恢复的出错次数	N/A	N/A
update error	子任务更新的出错次数	N/A	N/A
stop error	子任务停止的出错次数	N/A	N/A

HA 高可用

metric 名称	说明	告警说明	告警级别
number of dm-masters start leader components per minute	每分钟内 DM-master 尝试启用 leader 相关组件次数	N/A	N/A
number of workers in different state	不同状态下有多少个 DM-worker	存在离线的 DM-worker 超过一小时	critical
workers' state	DM-worker 的状态	N/A	N/A

metric 名称	说明	告警说明	告警级别
number of worker event error	不同类型的 DM-worker 错误出现次数	N/A	N/A
shard ddl error per minute	每分钟内不同类型的 shard DDL 错误次数	发生 shard DDL 错误	critical
number of pending shard ddl	未完成的 shard DDL 数目	存在未完成的 shard DDL 数目超过一小时	critical

Task 状态

metric 名称	说明	告警说明	告警级别
task state	迁移子任务的状态	当子任务状态处于 Paused 超过 20 分钟时	critical

Dump/Load unit

下面 metrics 仅在 `task-mode` 为 `full` 或者 `all` 模式下会有值。

metric 名称	说明	告警说明	告警
dump progress	dump unit 导出过程的进度百分比，值变化范围为：0% - 100%	N/A	N/A
load progress	load unit 导入过程的进度百分比，值变化范围为：0% - 100%	N/A	N/A
checksum progress	load unit 导入完成后，数据校验过程的进度百分比，值变化范围为：0% - 100%	N/A	N/A
total bytes for load unit	load unit 导入过程中源数据解析、生成数据 KV、生成索引 KV 阶段处理的字节数	N/A	N/A
chunk process duration	load unit 处理数据源文件 chunk 的耗时，单位：秒	N/A	N/A
dump process exits with error	dump unit 在 DM-worker 内部遇到错误并且退出了	立即告警	critica
load process exits with error	load unit 在 DM-worker 内部遇到错误并且退出了	立即告警	critica

Binlog replication

下面 metrics 仅在 `task-mode` 为 `incremental` 或者 `all` 模式下会有值。

metric 名称	说明	告警说明	告警级别
remaining time to sync	预计 Syncer 还需要多少分钟可以和上游 master 完全同步，单位：分钟	N/A	N/A
replicate lag gauge	上游 master 到下游的 binlog 复制延迟时间，单位：秒	N/A	N/A
replicate lag histogram	上游 master 到下游的 binlog 复制延迟分布，单位：秒。注意由于统计机制不同，数据会有误差	N/A	N/A
process exist with error	binlog replication unit 在 DM-worker 内部遇到错误并且退出了	立即告警	critical
binlog file gap between master and syncer	与上游 master 相比落后的 binlog file 个数	落后 binlog file 个数超过 1 个（不含 1 个）且持续 10 分钟时	critical
binlog file gap between relay and syncer	与 relay 相比落后的 binlog file 个数	落后 binlog file 个数超过 1 个（不含 1 个）且持续 10 分钟时	critical
binlog event QPS	单位时间内接收到的 binlog event 数量 (不包含需要跳过的 event)	N/A	N/A

metric 名称	说明	告警说明	告警级别
skipped binlog event QPS	单位时间内接收到的需要跳过的 binlog event 数量	N/A	N/A
read binlog event duration	binlog replication unit 从 relay log 或上游 MySQL 读取 binlog 的耗时，单位：秒	N/A	N/A
transform binlog event duration	binlog replication unit 解析 binlog 并将 binlog 转换成 SQL 语句的耗时，单位：秒	N/A	N/A
dispatch binlog event duration	binlog replication unit 调度一条 binlog event 的耗时，单位：秒	N/A	N/A
transaction execution latency	binlog replication unit 执行事务到下游的耗时，单位：秒	N/A	N/A
binlog event size	binlog replication unit 从 relay log 或上游 MySQL 读取的单条 binlog event 的大小	N/A	N/A
DML queue remain length	剩余 DML job 队列的长度	N/A	N/A
total sqls jobs	单位时间内新增的 job 数量	N/A	N/A
finished sqls jobs	单位时间内完成的 job 数量	N/A	N/A
statement execution latency	binlog replication unit 执行语句到下游的耗时，单位：秒	N/A	N/A

metric 名称	说明	告警说明	告警级别
add job duration	binlog replication unit 增加一条 job 到队列的耗时，单位：秒	N/A	N/A
DML conflict detect duration	binlog replication unit 检测 DML 间冲突的耗时，单位：秒	N/A	N/A
skipped event duration	binlog replication unit 跳过 binlog event 的耗时，单位：秒	N/A	N/A
unsynced tables	当前子任务内还未收到 shard DDL 的分表数量	N/A	N/A
shard lock resolving	当前子任务是否正在等待 shard DDL 迁移，大于 0 表示正在等待迁移	N/A	N/A
ideal QPS	在 DM 运行耗时为 0 时可以达到的最高 QPS	N/A	N/A
binlog event row	一个 binlog 事件中的行数	N/A	N/A
finished transaction total	执行完毕的事务数量	N/A	N/A
replication transaction batch	执行到下游的事务里中 sql 行数	N/A	N/A
flush checkpoints time interval	检查点刷新时间间隔，单位：秒	N/A	N/A

Relay log

metric 名称	说明	告警说明	告警级别
storage capacity	relay log 占有的磁盘的总容量	N/A	N/A
storage remain	relay log 占有的磁盘的剩余可用容量	小于 10G 的时候需要告警	critical
process exits with error	relay log 在 DM-worker 内部遇到错误并且退出了	立即告警	critical
relay log data corruption	relay log 文件损坏的个数	立即告警	emergency
fail to read binlog from master	relay 从上游的 MySQL 读取 binlog 时遇到的错误数	立即告警	critical
fail to write relay log	relay 写 binlog 到磁盘时遇到的错误数	立即告警	critical
binlog file index	relay log 最大的文件序列号。如 value = 1 表示 relay-log.000001	N/A	N/A
binlog file gap between master and relay	relay 与上游 master 相比落后的 binlog file 个数	落后 binlog file 个数超过 1 个 (不含 1 个) 且持续 10 分钟时	critical
binlog pos	relay log 最新文件的写入 offset	N/A	N/A
read binlog event duration	relay log 从上游的 MySQL 读取 binlog 的时延，单位：秒	N/A	N/A

metric 名称	说明	告警说明	告警级别
write relay log duration	relay log 每次写 binlog 到磁盘的时延，单位：秒	N/A	N/A
binlog event size	relay log 写到磁盘的单条 binlog 的大小	N/A	N/A

Instance

在 Grafana dashboard 中，instance 的默认名称为 `DM-instance`。

Relay log

metric 名称	说明	告警说明	告警级别
storage capacity	relay log 占有的磁盘的总容量	N/A	N/A
storage remain	relay log 占有的磁盘的剩余可用容量	小于 10G 的时候需要告警	critical
process exits with error	relay log 在 DM-worker 内部遇到错误并且退出了	立即告警	critical
relay log data corruption	relay log 文件损坏的个数	立即告警	emergency
fail to read binlog from master	relay 从上游的 MySQL 读取 binlog 时遇到的错误数	立即告警	critical
fail to write relay log	relay 写 binlog 到磁盘时遇到的错误数	立即告警	critical
binlog file index	relay log 最大的文件序列号。如 value = 1 表示 relay-log.000001	N/A	N/A

metric 名称	说明	告警说明	告警级别
binlog file gap between master and relay	relay 与上游 master 相比落后的 binlog file 个数	落后 binlog file 个数超过 1 个（不含 1 个）且持续 10 分钟时	critical
binlog pos	relay log 最新文件的写入 offset	N/A	N/A
read binlog duration	relay log 从上游的 MySQL 读取 binlog 的时延，单位：秒	N/A	N/A
write relay log duration	relay log 每次写 binlog 到磁盘的时延，单位：秒	N/A	N/A
binlog size	relay log 写到磁盘的单条 binlog 的大小	N/A	N/A

task

metric 名称	说明	告警说明	告警级别
task state	迁移子任务的状态	当子任务状态处于 paused 超过 10 分钟时	critical
load progress	load unit 导入过程的进度百分比，值变化范围为：0% - 100%	N/A	N/A
binlog file gap between master and syncer	与上游 master 相比 binlog replication unit 落后的 binlog file 个数	N/A	N/A

metric 名称	说明	告警说明	告警级别
shard lock resolving	当前子任务是否正在等待 shard DDL 迁移，大于 0 表示正在等待迁移	N/A	N/A

13.6.9.8.2 DM 告警信息

使用 TiUP 部署 DM 集群的时候，会默认部署一套告警系统。

DM 的告警规则及其对应的处理方法可参考告警处理。

DM 的告警信息与监控指标均基于 Prometheus，告警规则与监控指标的对应关系可参考DM 监控指标。

13.6.9.9 TiDB Data Migration 故障及处理方法

本文档介绍 TiDB Data Migration (DM) 的错误系统及常见故障的处理方法。

13.6.9.9.1 DM 错误系统

在 DM 的错误系统中，对于一条特定的错误，通常主要包含以下信息：

- code：错误码。

  同一种错误都使用相同的错误码。错误码不随 DM 版本改变。

  在 DM 迭代过程中，部分错误可能会被移除，但错误码不会。新增的错误会使用新的错误码，不会复用已有的错误码。

- class：发生错误的类别。

  用于标记出现错误的系统子模块。

  下表展示所有的错误类别、错误对应的系统子模块、错误样例：

  |

  错误类别

```
| 错误对应的系统子模块                | 错误样例
    ↪                                                        |
```

```
:————— | :————————————— | :————————————————————————— |
database | 执行数据库操作出现错误 | [code=10003:class=database:scope=downstream:level=medium]
↪  database driver: invalid connection |
functional | 系统底层的基础函数错误 | [code=11005:class=functional:scope=internal:level=high]
↪  not allowed operation: alter multiple tables in one statement |
config | 配置错误 | [code=20005:class=config:scope=internal:level=medium] empty source-id not
↪ valid |
```

```
binlog-op | binlog 操作出现错误 | [code=22001:class=binlog-op:scope=internal:level=high] empty
↪ UUIDs not valid |
checkpoint | checkpoint 相关操作出现错误 | [code=24002:class=checkpoint:scope=internal:level=
↪ high] save point bin.1234 is older than current pos bin.1371 |
task-check | 进行任务检查时出现错误 | [code=26003:class=task-check:scope=internal:level=medium
↪ ] new table router error |
relay-event-lib| 执行 relay 模块基础功能时出现错误 | [code=28001:class=relay-event-lib:scope=
↪ internal:level=high] parse server-uuid.index |
relay-unit | relay 处理单元内出现错误 | [code=30015:class=relay-unit:scope=upstream:level=high]
↪  TCPReader get event: ERROR 1236 (HY000): Could not open log file |
dump-unit | dump 处理单元内出现错误 | [code=32001:class=dump-unit:scope=internal:level=high]
↪ mydumper runs with error: CRITICAL **: 15:12:17.559: Error connecting to database: Access
↪  denied for user 'root'@'172.17.0.1' (using password: NO) |
load-unit | load 处理单元内出现错误 | [code=34002:class=load-unit:scope=internal:level=high]
↪ corresponding ending of sql: ')' not found |
sync-unit | sync 处理单元内出现错误 | [code=36027:class=sync-unit:scope=internal:level=high]
↪ Column count doesn't match value count: 9 (columns)vs 10 (values) |
dm-master | DM-master 服务内部出现错误 | [code=38008:class=dm-master:scope=internal:level
↪ =high] grpc request error: rpc error: code = Unavailable desc = all SubConns are in
↪ TransientFailure, latest connection error: connection error: desc = "transport: Error
↪ while dialing dial tcp 172.17.0.2:8262: connect: connection refused" |
dm-worker | DM-worker 服务内部出现错误 | [code=40066:class=dm-worker:scope=internal:level=high]
↪  ExecuteDDL timeout, try use query-status to query whether the DDL is still blocking |
dm-tracer | DM-tracer 服务内部出现错误 | [code=42004:class=dm-tracer:scope=internal:level=medium
↪ ] trace event test.1 not found |
schema-tracker | 增量复制时记录 schema 变更出现错误 | [code=44006:class=schema-tracker:scope=
↪ internal:level=high],"cannot track DDL: ALTER TABLE test DROP COLUMN col1" |
scheduler | 数据迁移任务调度相关操作出错操作 | [code=46001:class=scheduler:scope=internal:
↪ level=high],"the scheduler has not started" |
dmctl | dmctl 内部或与其他组件交互出现错误 | [code=48001:class=dmctl:scope=internal:level=high
↪ ],"can not create grpc connection" |
```

- `scope`：错误作用域。

  用于标识错误发生时 DM 作用对象的范围和来源，包括未设置 (`not-set`)、上游数据库 (`upstream`)、下游数据库 (`downstream`)、内部 (`internal`) 四种类型。

  如果错误发生的逻辑直接涉及到上下游数据库请求，作用域会设置为 `upstream` 或 `downstream`，其他出错场景目前都设置为 `internal`。

- `level`：错误级别。

  错误的严重级别，包括低级别 (`low`)、中级别 (`medium`)、高级别 (`high`) 三种。

  低级别通常是用户操作、输入错误，不影响正常迁移任务；中级别通常是用户配置等错误，会影响部分新启动服务，不影响已有系统迁移状态；高级别通常是用户需要关注的一些错误，可能存在迁移任务中断等风险，需要用户进行处理。

- `message`：错误描述。

错误的详细描述信息。对于错误调用链上每一层额外增加的错误 message，采用 errors.Wrap 的模式来叠加和保存错误 message。wrap 最外层的 message 是 DM 内部对该错误的描述，wrap 最内层的 message 是该错误最底层出错位置的错误描述。

- workaround: 错误处理方法（可选）。

  对该错误的处理方法。对于部分明确的错误（如：配置信息错误等），DM 会在 workaround 中给出相应的人为处理方法。

- 错误堆栈信息（可选）。

  DM 根据错误的严重程度和必要性来选择是否输出错误堆栈。错误堆栈记录了错误发生时完整的堆栈调用信息。如果用户通过错误基本信息和错误 message 描述不能完全诊断出错误发生的原因，可以通过错误堆栈进一步跟进出错时代码的运行路径。

可在 DM 代码仓库已发布的错误码 中查询完整的错误码列表。

13.6.9.9.2 DM 故障诊断

如果在运行 DM 工具时出现了错误，请尝试以下解决方案：

1. 执行 query-status 命令查看任务运行状态以及相关错误输出。
2. 查看与该错误相关的日志文件。日志文件位于 DM-master、DM-worker 部署节点上，通过DM 错误系统 获取错误的关键信息，然后查看常见故障处理方法以寻找相应的解决方案。
3. 如果该错误还没有相应的解决方案，并且你无法通过查询日志或监控指标自行解决此问题，请从 PingCAP 官方或 TiDB 社区获取支持。
4. 一般情况下，错误处理完成后，只需使用 dmctl 重启任务即可。

```
resume-task ${task name}
```

但在某些情况下，你还需要重置数据迁移任务。有关何时需要重置以及如何重置，详见重置数据迁移任务。

13.6.9.9.3 常见故障处理方法

|

错误码

错误说明	解决方法
:——–	:————————————————————	:———————————————————–
code=10001	数据库操作异常	进一步分析错误信息和错误堆栈
code=10002	数据库底层的 bad connection 错误，通常表示 DM 到下游 TiDB 的数据库连接出现了异常（如网络故障、TiDB 重启等）且当前请求的数据暂时未能发送到 TiDB。	DM 提供针对此类错误的自动恢复。如果长时间未恢复，需要用户检查网络或 TiDB 状态。
code=10003	数据库底层 invalid connection 错误，通常表示 DM 到下游 TiDB 的数据库连接出现了异常（如	

网络故障、TiDB 重启、TiKV busy 等）且当前请求已有部分数据发送到了 TiDB。| DM 提供针对此类错误的自动恢复。如果未能正常恢复，需要用户进一步检查错误信息并根据具体场景进行分析。 |
code=10005 | 数据库查询类语句出错 | |
code=10006 | 数据库 EXECUTE 类型语句出错，包括 DDL 和 INSERT/UPDATE/DELETE 类型的 DML。更详细的错误信息可通过错误 message 获取。错误 message 中通常包含操作数据库所返回的错误码和错误信息。 | |
code=11006 | DM 内置的 parser 解析不兼容的 DDL 时出错 | 可参考Data Migration 故障诊断 -处理不兼容的 DDL 语句 提供的解决方案 |
code=20010 | 处理任务配置时，解密数据库的密码出错 | 检查任务配置中提供的下游数据库密码是否有使用 dmctl 正确加密 |
code=26002 | 任务检查创建数据库连接失败。更详细的错误信息可通过错误 message 获取。错误 message 中包含操作数据库所返回的错误码和错误信息。 | 检查 DM-master 所在的机器是否有权限访问上游 |
code=32001 | dump 处理单元异常 | 如果报错 `msg` 包含 `mydumper: argument list too long.`，则需要用户根据 block-allow-list，在 `task.yaml` 的 dump 处理单元的 `extra-args` 参数中手动加上 `--regex` 正则表达式设置要导出的库表。例如，如果要导出所有库中表名字为 `hello` 的表，可加上 `--regex '.*\\.hello$'`，如果要导出所有表，可加上 `--regex '.*'`。 |
code=38008 | DM 组件间的 gRPC 通信出错 | 检查 `class`，定位错误发生在哪些组件的交互环节，根据错误 message 判断是哪类通信错误。如果是 gRPC 建立连接出错，可检查通信服务端是否运行正常。 |

迁移任务中断并包含 `invalid connection` 错误

原因

发生 `invalid connection` 错误时，通常表示 DM 到下游 TiDB 的数据库连接出现了异常（如网络故障、TiDB 重启、TiKV busy 等）且当前请求已有部分数据发送到了 TiDB。

解决方案

由于 DM 中存在迁移任务并发向下游复制数据的特性，因此在任务中断时可能同时包含多个错误（可通过 `query-status` 查询当前错误）。

- 如果错误中仅包含 `invalid connection` 类型的错误且当前处于增量复制阶段，则 DM 会自动进行重试。
- 如果 DM 由于版本问题等未自动进行重试或自动重试未能成功，则可尝试先使用 `stop-task` 停止任务，然后再使用 `start-task` 重启任务。

迁移任务中断并包含 `driver: bad connection` 错误

原因

发生 `driver: bad connection` 错误时，通常表示 DM 到下游 TiDB 的数据库连接出现了异常（如网络故障、TiDB 重启等）且当前请求的数据暂时未能发送到 TiDB。

解决方案

当前版本 DM 会自动进行重试，如果由于版本问题等未自动重试，可先使用 `stop-task` 停止任务后再使用 `start-task` 重启任务。

relay 处理单元报错 `event from * in * diff from passed-in event *` 或迁移任务中断并包含 `get binlog error ERROR 1236 (HY000)`、`binlog checksum mismatch, data may be corrupted` 等 binlog 获取或解析失败错误

原因

在 DM 进行 relay log 拉取与增量复制过程中，如果遇到了上游超过 4GB 的 binlog 文件，就可能出现这两个错误。

原因是 DM 在写 relay log 时需要依据 binlog position 及文件大小对 event 进行验证，且需要保存迁移的 binlog position 信息作为 checkpoint。但是 MySQL binlog position 官方定义使用 uint32 存储，所以超过 4G 部分的 binlog position 的 offset 值会溢出，进而出现上面的错误。

解决方案

对于 relay 处理单元，可通过以下步骤手动恢复：

1. 在上游确认出错时对应的 binlog 文件的大小超出了 4GB。
2. 停止 DM-worker。
3. 将上游对应的 binlog 文件复制到 relay log 目录作为 relay log 文件。
4. 更新 relay log 目录内对应的 `relay.meta` 文件以从下一个 binlog 开始拉取。如果 DM worker 已开启 `enable_gtid`，那么在修改 `relay.meta` 文件时，同样需要修改下一个 binlog 对应的 GTID。如果未开启 `enable_gtid` 则无需修改 GTID。

   例如：报错时有 `binlog-name = "mysql-bin.004451"` 与 `binlog-pos = 2453`，则将其分别更新为 `binlog-name = "mysql-bin.004452"` 和 `binlog-pos = 4`，同时更新 `binlog-gtid = "f0e914ef-54cf-11e7-813d-6c92bf2fa791:1-138218058"`。
5. 重启 DM-worker。

对于 binlog replication 处理单元，可通过以下步骤手动恢复：

1. 在上游确认出错时对应的 binlog 文件的大小超出了 4GB。
2. 通过 `stop-task` 停止迁移任务。
3. 将下游 `dm_meta` 数据库中 global checkpoint 与每个 table 的 checkpoint 中的 `binlog_name` 更新为出错的 binlog 文件，将 `binlog_pos` 更新为已迁移过的一个合法的 position 值，比如 4。

   例如：出错任务名为 `dm_test`，对应的 `source-id` 为 `replica-1`，出错时对应的 binlog 文件为 `mysql-bin|000001.004451`，则执行 `UPDATE dm_test_syncer_checkpoint SET binlog_name='mysql-bin|000001.004451', binlog_pos = 4 WHERE id='replica-1';`。
4. 在迁移任务配置中为 `syncers` 部分设置 `safe-mode: true` 以保证可重入执行。
5. 通过 `start-task` 启动迁移任务。
6. 通过 `query-status` 观察迁移任务状态，当原造成出错的 relay log 文件迁移完成后，即可还原 `safe-mode` 为原始值并重启迁移任务。

执行 `query-status` 或查看日志时出现 `Access denied for user 'root'@'172.31.43.27' (using password: YES)`

在所有 DM 配置文件中，数据库相关的密码都推荐使用经 dmctl 加密后的密文（若数据库密码为空，则无需加密）。有关如何使用 dmctl 加密明文密码，参见使用 dmctl 加密数据库密码。

此外，在 DM 运行过程中，上下游数据库的用户必须具备相应的读写权限。在启动迁移任务过程中，DM 会自动进行相应权限的前置检查，详见上游 MySQL 实例配置前置检查。

load 处理单元报错 `packet for query is too large. Try adjusting the 'max_allowed_packet' variable`

原因

- MySQL client 和 MySQL/TiDB Server 都有 `max_allowed_packet` 配额的限制，如果在使用过程中违反其中任何一个 `max_allowed_packet` 配额，客户端程序就会收到对应的报错。目前最新版本的 DM 和 TiDB Server 的默认 `max_allowed_packet` 配额都为 64M。
- DM 的全量数据导入处理模块不支持对 dump 处理模块导出的 SQL 文件进行切分。

解决方案

- 推荐在 DM 的 dump 处理单元提供的配置 `extra-args` 中设置 `statement-size`:

  依据默认的 `--statement-size` 设置，DM 的 dump 处理单元默认生成的 `Insert Statement` 大小一般会在 1M 左右，使用默认值就可以确保绝大部分情况 load 处理单元不会报错 `packet for query is too large. Try adjusting the 'max_allowed_packet' variable`。

  有时候在 dump 过程中会出现下面的 WARN log。这个 WARN log 不影响 dump 的过程，只是说明 dump 的表可能是宽表。

  ```
  Row bigger than statement_size for xxx
  ```

- 如果宽表的单行超过了 64M，需要修改以下两项配置，并且确保其生效。
  - 在 TiDB Server 执行 `set @@global.max_allowed_packet=134217728`（`134217728 = 128M`）
  - 根据实际情况为 DM 的任务配置文件中的 `target-database` 增加配置 `max-allowed-packet: 134217728`(128M)，执行 `stop-task` 后再重新 `start-task`。

13.6.9.10 TiDB Data Migration 术语表

本文档介绍 TiDB Data Migration (TiDB DM) 相关术语。

13.6.9.10.1 B

Binlog

在 TiDB DM 中，Binlog 通常指 MySQL/MariaDB 生成的 binary log 文件，具体请参考 MySQL Binary Log 与 MariaDB Binary Log。

Binlog event

MySQL/MariaDB 生成的 Binlog 文件中的数据变更信息，具体请参考 MySQL Binlog Event 与 MariaDB Binlog Event。

Binlog event filter

比 Block & allow table list 更加细粒度的过滤功能，具体可参考Binlog Event Filter。

Binlog position

特定 Binlog event 在 Binlog 文件中的位置偏移信息，具体请参考 MySQL `SHOW BINLOG EVENTS` 与 MariaDB `SHOW BINLOG EVENTS`。

Binlog replication 处理单元/ sync 处理单元

DM-worker 内部用于读取上游 Binlog 或本地 Relay log 并迁移到下游的处理单元，每个 Subtask 对应一个 Binlog replication 处理单元。在当前文档中，有时也称作 Sync 处理单元。

Block & allow table list

针对上游数据库实例表的黑白名单过滤功能，具体可参考Block & Allow Table Lists。该功能与 MySQL Replication Filtering 及 MariaDB Replication Filters 类似。

13.6.9.10.2 C

Checkpoint

TiDB DM 在全量导入与增量复制过程中的断点信息，用于在重新启动或恢复任务时从之前已经处理过的位置继续执行。

- 对于全量导入，Checkpoint 信息对应于每个数据文件已经被成功导入的数据对应的文件内偏移量等信息，其在每个导入数据的事务中迁移更新；
- 对于增量复制，Checkpoint 信息对应于已经成功解析并导入到下游的Binlog event 对应的Binlog position 等信息，其在 DDL 导入成功后或距上次更新时间超过 30 秒等条件下更新。

另外，Relay 处理单元 对应的 `relay.meta` 内记录的信息也相当于 Checkpoint，其对应于 Relay 处理单元已经成功从上游拉取并写入到Relay log 的Binlog event 对应的Binlog position 或GTID 信息。

13.6.9.10.3 D

Dump 处理单元

DM-worker 内部用于从上游导出全量数据的处理单元，每个 Subtask 对应一个 Dump 处理单元。

13.6.9.10.4 F

复制/增量复制

使用 TiDB Data Migration 工具将上游数据库的增量数据复制到下游数据库的过程。

本用户手册中，在明确提到是“增量”的情况下，将使用“复制”或“增量复制”进行文档描述。

13.6.9.10.5 G

GTID

MySQL/MariaDB 的全局事务 ID，当启用该功能后会在 Binlog 文件中记录 GTID 相关信息，多个 GTID 即组成为 GTID Set，具体请参考 MySQL GTID Format and Storage 与 MariaDB Global Transaction ID。

13.6.9.10.6 L

Load 处理单元

DM-worker 内部用于将全量导出数据导入到下游的处理单元，每个 Subtask 对应一个 Load 处理单元。在当前文档中，有时也称作 Import 处理单元。

13.6.9.10.7 Q

迁移/全量迁移

使用 TiDB Data Migration 工具将上游数据库的全量数据迁移到下游数据库的过程。

本用户手册中，在明确提到是“全量”的情况下，将使用“迁移”或“全量迁移”进行文档描述；在明确提到是“全量 + 增量”的情况下，也将统一使用“迁移”进行文档描述。

13.6.9.10.8 R

Relay log

DM-worker 从上游 MySQL/MariaDB 拉取 Binlog 后存储在本地的文件，当前其格式为标准的 Binlog 格式，可使用版本兼容的 mysqlbinlog 等工具进行解析。其作用与 MySQL Relay Log 及 MariaDB Relay Log 相近。

有关 TiDB DM 内 Relay log 的目录结构、初始迁移规则、数据清理等内容，可参考TiDB DM Relay Log。

Relay 处理单元

DM-worker 内部用于从上游拉取 Binlog 并写入数据到 Relay log 的处理单元，每个 DM-worker 实例内部仅存在一个该处理单元。

13.6.9.10.9 S

Safe mode

指增量复制过程中，用于支持在表结构中存在主键或唯一索引的条件下可重复导入 DML 的模式。该模式的主要特点是：将来自上游的 INSERT 改写为 REPLACE，将 UPDATE 改写为 DELETE 与 REPLACE 后再向下游执行。

该模式会在满足如下任一条件时启用：

- 任务配置文件中设置 `safe-mode: true` 时会始终启用
- 合库合表模式下，DDL 尚未在所有分表完成同步时保持启用
- 在全量迁移任务中的 dump 处理单元配置 `--consistency none` 后，不能确定导出开始时的 binlog 变动是否影响了导出数据。Safe mode 会在增量复制这部分 binlog 时保持启用
- 任务出错停止并恢复后，对有些数据的操作可能会被执行两次时保持启用

Shard DDL

指合库合表迁移过程中，在上游各分表 (shard) 上执行的需要 TiDB DM 进行协调迁移的 DDL。在当前文档中，有时也称作 Sharding DDL。

Shard DDL lock

用于协调 Shard DDL 迁移的锁机制，具体原理可查看悲观模式下分库分表合并迁移实现原理。在当前文档中，有时也称作 Sharding DDL lock。

Shard group

指合库合表迁移过程中，需要合并迁移到下游同一张表的所有上游分表 (shard)，TiDB DM 内部具体实现时使用了两级抽象的 Shard group，具体可查看悲观模式下分库分表合并迁移实现原理。在当前文档中，有时也称作 Sharding group。

Subtask

数据迁移子任务，即数据迁移任务运行在单个 DM-worker 实例上的部分。根据任务配置的不同，单个数据迁移任务可能只有一个子任务，也可能有多个子任务。

Subtask status

数据迁移子任务所处的状态，目前包括 New、Running、Paused、Stopped 及 Finished 5 种状态。有关数据迁移任务、子任务状态的更多信息可参考任务状态。

13.6.9.10.10 T

Table routing

用于支持将上游 MySQL/MariaDB 实例的某些表迁移到下游指定表的路由功能，可以用于分库分表的合并迁移，具体可参考Table routing。

Task

数据迁移任务，执行 `start-task` 命令成功后即启动一个数据迁移任务。根据任务配置的不同，单个数据迁移任务既可能只在单个 DM-worker 实例上运行，也可能同时在多个 DM-worker 实例上运行。

Task status

数据迁移子任务所处的状态，由Subtask status 整合而来，具体信息可查看任务状态。

13.6.9.11 使用示例

13.6.9.11.1 使用 DM 迁移数据

本文介绍如何使用 DM 工具迁移数据。

第 1 步：部署 DM 集群

推荐使用 TiUP 部署 DM 集群；也可以使用 binary 部署 DM 集群用于体验或测试。

> 注意：
>
> - 在 DM 所有的配置文件中，对于数据库密码推荐使用 dmctl 加密后的密文。如果数据库密码为空，则不需要加密。关于如何使用 dmctl 加密明文密码，参考使用 dmctl 加密数据库密码。
> - 上下游数据库用户必须拥有相应的读写权限。

第 2 步：检查集群信息

使用 TiUP 部署 DM 集群后，相关配置信息如下：

- DM 集群相关组件配置信息

组件	主机	端口
dm_worker1	172.16.10.72	8262
dm_worker2	172.16.10.73	8262
dm_master	172.16.10.71	8261

- 上下游数据库实例相关信息

数据库实例	主机	端口	用户名	加密密码
上游 MySQL-1	172.16.10.81	3306	root	VjX8cEeTX+qcvZ3bPaO4h0C80pe/1aU=
上游 MySQL-2	172.16.10.82	3306	root	VjX8cEeTX+qcvZ3bPaO4h0C80pe/1aU=
下游 TiDB	172.16.10.83	4000	root	

上游 MySQL 数据库实例用户所需权限参见上游 MySQL 实例配置前置检查介绍。

第 3 步：创建数据源

1. 将 MySQL-1 的相关信息写入到 `conf/source1.yaml` 中：

```
# MySQL1 Configuration.

source-id: "mysql-replica-01"

# DM-worker 是否使用全局事务标识符（GTID）拉取 binlog。使用前提是在上游 MySQL 已开启 GTID
    ↪ 模式。
enable-gtid: false

from:
  host: "172.16.10.81"
  user: "root"
  password: "VjX8cEeTX+qcvZ3bPaO4h0C80pe/1aU="
  port: 3306
```

2. 在终端中执行下面的命令，使用 `tiup dmctl` 将 MySQL-1 的数据源配置加载到 DM 集群中：

```
tiup dmctl --master-addr 172.16.10.71:8261 operate-source create conf/source1.yaml
```

3. 对于 MySQL-2，修改配置文件中的相关信息，并执行相同的 dmctl 命令。

第 4 步：配置任务

假设需要将 MySQL-1 和 MySQL-2 实例的 `test_db` 库的 `test_table` 表以全量 + 增量的模式迁移到下游 TiDB 的 `test_db` 库的 `test_table` 表。

编辑任务配置文件 `task.yaml`：

```
##### 任务名，多个同时运行的任务不能重名。
name: "test"
##### 全量+增量 (all) 迁移模式。
task-mode: "all"
##### 下游 TiDB 配置信息。
target-database:
  host: "172.16.10.83"
  port: 4000
  user: "root"
  password: ""

##### 当前数据迁移任务需要的全部上游 MySQL 实例配置。
mysql-instances:
-
  # 上游实例或者复制组 ID，参考 `inventory.ini` 的 `source_id` 或者 `dm-master.toml` 的 `source-
      ↪ id 配置`。
  source-id: "mysql-replica-01"
  # 需要迁移的库名或表名的黑白名单的配置项名称，用于引用全局的黑白名单配置，全局配置见下面的 `
      ↪ block-allow-list` 的配置。
  block-allow-list: "global"          # 如果 DM 版本早于 v2.0.0-beta.2 则使用 black-white-list。
  # dump 处理单元的配置项名称，用于引用全局的 dump 处理单元配置。
  mydumper-config-name: "global"

-
  source-id: "mysql-replica-02"
  block-allow-list: "global"          # 如果 DM 版本早于 v2.0.0-beta.2 则使用 black-white-list。
  mydumper-config-name: "global"

##### 黑白名单全局配置，各实例通过配置项名引用。
block-allow-list:                     # 如果 DM 版本早于 v2.0.0-beta.2 则使用 black-white-list。
  global:
    do-tables:                        # 需要迁移的上游表的白名单。
    - db-name: "test_db"              # 需要迁移的表的库名。
      tbl-name: "test_table"          # 需要迁移的表的名称。

##### dump 处理单元全局配置，各实例通过配置项名引用。
mydumpers:
  global:
    extra-args: ""
```

第 5 步：启动任务

为了提前发现数据迁移任务的一些配置错误，DM 中增加了前置检查功能：

- 启动数据迁移任务时，DM 自动检查相应的权限和配置。

- 也可使用 `check-task` 命令手动前置检查上游的 MySQL 实例配置是否符合 DM 的配置要求。

> **注意：**
>
> 第一次启动数据迁移任务时，必须确保上游数据库已配置。否则，启动任务时会报错。

使用 `tiup dmctl` 执行以下命令启动数据迁移任务。其中，`task.yaml` 是之前编辑的配置文件。

```
tiup dmctl --master-addr 172.16.10.71:8261 start-task ./task.yaml
```

- 如果执行该命令后返回的结果如下，则表明任务已成功启动。

```
{
    "result": true,
    "msg": "",
    "workers": [
        {
            "result": true,
            "worker": "172.16.10.72:8262",
            "msg": ""
        },
        {
            "result": true,
            "worker": "172.16.10.73:8262",
            "msg": ""
        }
    ]
}
```

- 如果任务启动失败，可根据返回结果的提示进行配置变更后执行 `start-task task.yaml` 命令重新启动任务。

第 6 步：查询任务

如需了解 DM 集群中是否存在正在运行的迁移任务及任务状态等信息，可使用 `tiup dmctl` 执行以下命令进行查询：

```
tiup dmctl --master-addr 172.16.10.71:8261 query-status
```

第 7 步：停止任务

如果不再需要进行数据迁移，可以使用 `tiup dmctl` 执行以下命令停止迁移任务：

```
tiup dmctl --master-addr 172.16.10.71:8261 stop-task test
```

其中的 test 是 task.yaml 配置文件中 name 配置项设置的任务名。

第 8 步：监控任务与查看日志

如果使用 TiUP 部署 DM 集群时，正确部署了 Prometheus、Alertmanager 与 Grafana，且其地址均为 172.16.10.71。可在浏览器中打开 http://172.16.10.71:9093 进入 Alertmanager 查看 DM 告警信息；可在浏览器中打开 http://172.16.10.71:3000 进入 Grafana，选择 DM 的 dashboard 查看 DM 相关监控项。

DM 在运行过程中，DM-worker、 DM-master 及 dmctl 都会通过日志输出相关信息。各组件的日志目录如下：

- DM-master 日志目录：通过 DM-master 进程参数 --log-file 设置。如果使用 TiUP 部署 DM，则日志目录位于 {log_dir}。
- DM-worker 日志目录：通过 DM-worker 进程参数 --log-file 设置。如果使用 TiUP 部署 DM，则日志目录位于 {log_dir}。

13.6.9.11.2 创建 TiDB Data Migration 数据迁移任务

本文档介绍在 TiDB Data Migration (DM) 集群部署成功后，如何快速创建简单的数据迁移任务。

使用样例

在本地部署两个开启 binlog 的 MySQL 实例和一个 TiDB 实例；使用 DM 集群的一个 DM-master 来管理集群和数据迁移任务。各个节点的信息如下：

实例	服务器地址	端口
MySQL1	127.0.0.1	3306
MySQL2	127.0.0.1	3307
TiDB	127.0.0.1	4000
DM-master	127.0.0.1	8261

下面以此为例，说明如何创建数据迁移任务。

运行上游 MySQL

准备 2 个可运行的 MySQL 实例，也可以使用 Docker 快速启动 MySQL，示例命令如下：

```
docker run --rm --name mysql-3306 -p 3306:3306 -e MYSQL_ALLOW_EMPTY_PASSWORD=true mysql:5.7.22 --
    ↪ log-bin=mysql-bin --port=3306 --bind-address=0.0.0.0 --binlog-format=ROW --server-id=1 --
    ↪ gtid_mode=ON --enforce-gtid-consistency=true > mysql.3306.log 2>&1 &
docker run --rm --name mysql-3307 -p 3307:3307 -e MYSQL_ALLOW_EMPTY_PASSWORD=true mysql:5.7.22 --
    ↪ log-bin=mysql-bin --port=3307 --bind-address=0.0.0.0 --binlog-format=ROW --server-id=1 --
    ↪ gtid_mode=ON --enforce-gtid-consistency=true > mysql.3307.log 2>&1 &
```

准备数据

- 向 mysql-3306 写入示例数据。

```
drop database if exists `sharding1`;
create database `sharding1`;
use `sharding1`;
```

```
create table t1 (id bigint, uid int, name varchar(80), info varchar(100), primary key (`id`)
    ↪ , unique key(`uid`)) DEFAULT CHARSET=utf8mb4;
create table t2 (id bigint, uid int, name varchar(80), info varchar(100), primary key (`id`)
    ↪ , unique key(`uid`)) DEFAULT CHARSET=utf8mb4;
insert into t1 (id, uid, name) values (1, 10001, 'Gabriel García Márquez'), (2 ,10002, 'Cien
    ↪  años de soledad');
insert into t2 (id, uid, name) values (3,20001, 'José Arcadio Buendía'), (4,20002, 'Úrsula
    ↪ Iguarán'), (5,20003, 'José Arcadio');
```

- 向 mysql-3307 写入示例数据。

```
drop database if exists `sharding2`;
create database `sharding2`;
use `sharding2`;
create table t2 (id bigint, uid int, name varchar(80), info varchar(100), primary key (`id`)
    ↪ , unique key(`uid`)) DEFAULT CHARSET=utf8mb4;
create table t3 (id bigint, uid int, name varchar(80), info varchar(100), primary key (`id`)
    ↪ , unique key(`uid`)) DEFAULT CHARSET=utf8mb4;
insert into t2 (id, uid, name, info) values (6, 40000, 'Remedios Moscote', '{}');
insert into t3 (id, uid, name, info) values (7, 30001, 'Aureliano José', '{}'), (8, 30002, '
    ↪ Santa Sofía de la Piedad', '{}'), (9, 30003, '17 Aurelianos', NULL);
```

运行下游 TiDB

使用以下命令运行一个 TiDB server：

```
wget https://download.pingcap.org/tidb-community-server-v7.0.0-linux-amd64.tar.gz
tar -xzvf tidb-latest-linux-amd64.tar.gz
mv tidb-latest-linux-amd64/bin/tidb-server ./
./tidb-server -P 4000 --store mocktikv --log-file "./tidb.log" &
```

> **警告：**
>
> 本文档中 TiDB 的部署方法并不适用于生产或开发环境。

配置 MySQL 数据源

运行数据迁移任务前，需要对 source 进行配置，也就是 MySQL 的相关设置。

对密码进行加密

> **注意：**

- 如果数据库没有设置密码，则可以跳过该步骤。
- DM v1.0.6 及其以后版本可以使用明文密码配置 source 信息。

为了安全，可配置及使用加密后的密码。使用 dmctl 对 MySQL/TiDB 的密码进行加密，以密码为 “123456” 为例：

```
./dmctl encrypt "123456"
```

```
fCxfQ9XKCezSzuCD0Wf5dUD+LsKegSg=
```

记录该加密后的密码，用于下面新建 MySQL 数据源。

编写 source 配置文件

把以下配置文件内容写入到 conf/source1.yaml 中。

MySQL1 的配置文件：

```
##### MySQL1 Configuration.

source-id: "mysql-replica-01"

##### 是否开启 GTID
enable-gtid: true

from:
  host: "127.0.0.1"
  user: "root"
  password: "fCxfQ9XKCezSzuCD0Wf5dUD+LsKegSg="
  port: 3306
```

对于 MySQL2 数据源，将以上内容复制到文件 conf/source2.yaml 中，将 conf/source2.yaml 配置文件中的 name 修改为 mysql-replica-02，并将 password 和 port 改为相应的值。

创建 source

在终端中执行下面的命令，使用 dmctl 将 MySQL1 的数据源配置加载到 DM 集群中：

```
./dmctl --master-addr=127.0.0.1:8261 operate-source create conf/source1.yaml
```

对于 MySQL2，将上面命令中的配置文件替换成 MySQL2 对应的配置文件。

创建数据迁移任务

在导入准备数据后，MySQL1 和 MySQL2 实例中有若干个分表，这些分表的结构相同，所在库的名称都以 “sharding” 开头，表名称都以 “t” 开头，并且主键或唯一键不存在冲突（即每张分表的主键或唯一键各不相同）。现在需要把这些分表迁移到 TiDB 中的 db_target.t_target 表中。

首先创建任务的配置文件：

```
---
name: test
task-mode: all
shard-mode: "pessimistic"

target-database:
  host: "127.0.0.1"
  port: 4000
  user: "root"
  password: "" # 如果密码不为空，则推荐使用经过 dmctl 加密的密文

mysql-instances:
  - source-id: "mysql-replica-01"
    block-allow-list:  "instance"   # 如果 DM 版本早于 v2.0.0-beta.2 则使用 black-white-list
    route-rules: ["sharding-route-rules-table", "sharding-route-rules-schema"]
    mydumper-thread: 4
    loader-thread: 16
    syncer-thread: 16

  - source-id: "mysql-replica-02"
    block-allow-list:  "instance"  # 如果 DM 版本早于 v2.0.0-beta.2 则使用 black-white-list
    route-rules: ["sharding-route-rules-table", "sharding-route-rules-schema"]
    mydumper-thread: 4
    loader-thread: 16
    syncer-thread: 16

block-allow-list:                  # 如果 DM 版本早于 v2.0.0-beta.2 则使用 black-white-list
  instance:
    do-dbs: ["~^sharding[\\d]+"]
    do-tables:
    -  db-name: "~^sharding[\\d]+"
       tbl-name: "~^t[\\d]+"

routes:
  sharding-route-rules-table:
    schema-pattern: sharding*
    table-pattern: t*
    target-schema: db_target
    target-table: t_target

  sharding-route-rules-schema:
    schema-pattern: sharding*
    target-schema: db_target
```

将以上配置内容写入到 conf/task.yaml 文件中，使用 dmctl 创建任务：

```
./dmctl --master-addr 127.0.0.1:8261 start-task conf/task.yaml
```

结果如下：

```
{
    "result": true,
    "msg": "",
    "sources": [
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-01",
            "worker": "worker1"
        },
        {
            "result": true,
            "msg": "",
            "source": "mysql-replica-02",
            "worker": "worker2"
        }
    ]
}
```

这样就成功创建了一个将 MySQL1 和 MySQL2 实例中的分表数据迁移到 TiDB 的任务。

数据校验

修改上游 MySQL 分表中的数据，然后使用sync-diff-inspector 校验上下游数据是否一致，如果一致则说明迁移任务运行正常。

13.6.9.11.3 分表合并数据迁移最佳实践

本文阐述了使用 TiDB Data Migration（以下简称 DM）对分库分表进行合并迁移的场景中，DM 相关功能的支持和限制，旨在给出一个业务的最佳实践（使用默认的“悲观协调”模式）。

独立的数据迁移任务

在分库分表合并迁移的实现原理部分，我们介绍了 sharding group 的概念，简单来说可以理解为需要合并到下游同一个表的所有上游表即组成一个 sharding group。

当前的 sharding DDL 算法为了能协调在不同分表执行 DDL 对 schema 变更的影响，加入了一些使用限制。而当这些使用限制由于某些异常原因被打破时，我们需要手动处理 Sharding DDL Lock 甚至是完整重做整个数据迁移任务。

因此，为了减小异常发生时对数据迁移的影响，我们推荐将每一个 sharding group 拆分成一个独立的数据迁移任务。这样当异常发生时，可能只有少部分迁移任务需要进行手动处理，其他数据迁移任务可以不受影响。

手动处理 sharding DDL lock

从分库分表合并迁移的实现原理部分我们可以知道，DM 中的 sharding DDL lock 是用于协调不同上游分表向下游执行 DDL 的一种机制，本身并不是异常。

因此，当通过 `shard-ddl-lock` 查看到 DM-master 上存在 sharding DDL lock 时，或通过 `query-status` 查看到某些 DM-worker 有 `unresolvedGroups` 或 `blockingDDLs` 时，并不要急于使用 `shard-ddl-lock unlock` 尝试去手动解除 sharding DDL lock。

只有在确认当前未能自动解除 sharding DDL lock 是文档中所列的支持场景之一时，才能参照对应支持场景中的手动处理示例进行处理。对于其他未被支持的场景，我们建议完整重做整个数据迁移任务，即清空下游数据库中的数据以及该数据迁移任务相关的 `dm_meta` 信息后，重新执行全量数据及增量数据的迁移。

跨分表数据在主键或唯一索引冲突处理

来自多张分表的数据可能会引发主键或者唯一索引的数据冲突，这需要结合分表逻辑对每个主键或唯一索引进行检查。我们在此列举主键或唯一索引的三种情况：

- 分片键：通常来讲，相同的分片键始终会划分到同一张分表之中，因此分片键不会产生数据冲突。
- 自增主键：每个分表的自增主键会单独计数，因此会出现范围重叠的情况，这需要参照下一节自增主键冲突处理来解决。
- 其他主键或唯一索引：需要根据业务逻辑判断。如果出现数据冲突，也可参照下一节自增主键冲突处理预先在下游创建表结构。

自增主键冲突处理

推荐使用以下两种处理方式。

去掉自增主键的主键属性

假设上游分表的表结构如下：

```
CREATE TABLE `tbl_no_pk` (
  `auto_pk_c1` bigint(20) NOT NULL,
  `uk_c2` bigint(20) NOT NULL,
  `content_c3` text,
  PRIMARY KEY (`auto_pk_c1`),
  UNIQUE KEY `uk_c2` (`uk_c2`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

如果满足下列条件：

- `auto_pk_c1` 列对业务无意义，且不依赖该列的 `PRIMARY KEY` 属性。
- `uk_c2` 列有 `UNIQUE KEY` 属性，且能保证在所有上游分表间全局唯一。

则可以用以下步骤处理合表时可能由 `auto_pk_c1` 导致的 `ERROR 1062 (23000): Duplicate entry '***' for key 'PRIMARY'` 问题：

1. 在开始执行全量数据迁移前，在下游数据库创建用于合表迁移的表，但将 `auto_pk_c1` 的 `PRIMARY KEY` 属性修改为普通索引。

```
CREATE TABLE `tbl_no_pk_2` (
  `auto_pk_c1` bigint(20) NOT NULL,
  `uk_c2` bigint(20) NOT NULL,
  `content_c3` text,
  INDEX (`auto_pk_c1`),
  UNIQUE KEY `uk_c2` (`uk_c2`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

2. 在 task.yaml 文件中增加如下配置跳过自增主键冲突检查：

```
ignore-checking-items: ["auto_increment_ID"]
```

3. 启动数据迁移任务，执行全量与增量数据迁移。
4. 通过 query-status 验证数据迁移任务是否正常，在下游数据库中验证合表中是否已经存在了来自上游的数据。

使用联合主键

假设上游分表的表结构如下：

```
CREATE TABLE `tbl_multi_pk` (
  `auto_pk_c1` bigint(20) NOT NULL,
  `uuid_c2` bigint(20) NOT NULL,
  `content_c3` text,
  PRIMARY KEY (`auto_pk_c1`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

如果满足下列条件：

- 业务不依赖 auto_pk_c1 的 PRIMARY KEY 属性。
- auto_pk_c1 与 uuid_c2 的组合能确保全局唯一。
- 业务能接受将 auto_pk_c1 与 uuid_c2 组成联合 PRIMARY KEY。

则可以用以下步骤处理合表时可能由 auto_pk_c1 导致的 ERROR 1062 (23000): Duplicate entry '***' for key 'PRIMARY' 问题：

1. 在开始执行全量数据迁移前，在下游数据库创建用于合表迁移的表，但不为 auto_pk_c1 指定 PRIMARY KEY 属性，而是将 auto_pk_c1 与 uuid_c2 一起组成 PRIMARY KEY。

```
CREATE TABLE `tbl_multi_pk_c2` (
  `auto_pk_c1` bigint(20) NOT NULL,
  `uuid_c2` bigint(20) NOT NULL,
  `content_c3` text,
  PRIMARY KEY (`auto_pk_c1`,`uuid_c2`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1
```

2. 启动数据迁移任务，执行全量与增量数据迁移。
3. 通过 query-status 验证数据迁移任务是否正常，在下游数据库中验证合表中是否已经存在了来自上游的数据。

上游 RDS 封装分库分表的处理

上游数据源为 RDS 且使用了其分库分表功能的情况下，MySQL binlog 中的表名在 SQL client 连接时可能并不可见。例如在 UCloud 分布式数据库 UDDB 中，其 binlog 表名可能会多出 _0001 的后缀。这需要根据 binlog 中的表名规律，而不是 SQL client 所见的表名，来配置table routing 规则。

合表迁移过程中在上游增/删表

从分库分表合并迁移的实现原理部分我们可以知道，sharding DDL lock 的协调依赖于是否已收到上游已有各分表对应的 DDL，且当前 DM 在合表迁移过程中暂时不支持在上游动态增加或删除分表。如果需要在合表迁移过程中，对上游执行增、删分表操作，推荐按以下方式进行处理。

在上游增加分表

如果需要在上游增加新的分表，推荐按以下顺序执行操作：

1. 等待在上游分表上执行过的所有 sharding DDL 协调同步完成。
2. 通过 stop-task 停止任务。
3. 在上游添加新的分表。
4. 确保 task.yaml 配置能使新添加的分表与其他已有的分表能合并到同一个下游表。
5. 通过 start-task 启动任务。
6. 通过 query-status 验证数据迁移任务是否正常，在下游数据库中验证合表中是否已经存在了来自上游的数据。

在上游删除分表

如果需要在上游删除原有的分表，推荐按以下顺序执行操作：

1. 在上游删除原有的分表，并通过 SHOW BINLOG EVENTS 获取该 DROP TABLE 语句在 binlog 中对应的 End_log_pos，记为 Pos-M。
2. 通过 query-status 获取当前 DM 已经处理完成的 binlog event 对应的 position（syncerBinlog），记为 Pos-S。
3. 当 Pos-S 比 Pos-M 更大后，则说明 DM 已经处理完 DROP TABLE 语句，且该表在被删除前的数据都已经迁移到下游，可以进行后续操作；否则，继续等待 DM 进行数据迁移。
4. 通过 stop-task 停止任务。
5. 确保 task.yaml 配置能忽略上游已删除的分表。
6. 通过 start-task 启动任务。
7. 通过 query-status 验证数据迁移任务是否正常。

数据迁移限速/流控

当将多个上游 MySQL/MariaDB 实例的数据合并迁移到下游同一个 TiDB 集群时，由于每个与上游 MySQL/MariaDB 对应的 DM-worker 都会并发地进行全量与增量数据迁移，默认的并发度（全量阶段的 pool-size 与增量阶段的 worker-count）通过多个 DM-worker 累加后，可能会给下游造成过大的压力，此时应根据 TiDB 监控及 DM 监控进行初步的性能分析后，适当地调整各并发度参数的大小。后续 DM 也将考虑支持部分自动的流控策略。

13.6.9.12 异常解决

13.6.9.12.1 Data Migration 常见问题

DM 是否支持迁移阿里 RDS 以及其他云数据库的数据？

DM 仅支持解析标准版本的 MySQL/MariaDB 的 binlog，对于阿里云 RDS 以及其他云数据库没有进行过测试，如果确认其 binlog 为标准格式，则可以支持。

已知问题的兼容情况：

- 阿里云 RDS
  - 即使上游表没有主键，阿里云 RDS 的 binlog 中也会包含隐藏的主键列，与上游表结构不一致。
- 华为云 RDS
  - 不支持，详见：华为云数据库 RDS 是否支持直接读取 Binlog 备份文件。

task 配置中的黑白名单的正则表达式是否支持非获取匹配（?!）？

目前不支持，DM 仅支持 golang 标准库的正则，可以通过 re2-syntax 了解 golang 支持的正则表达式。

如果在上游执行的一个 statement 包含多个 DDL 操作，DM 是否支持迁移？

DM 会尝试将包含多个 DDL 变更操作的单条语句拆分成只包含一个 DDL 操作的多条语句，但是可能没有覆盖所有的场景。建议在上游执行的一条 statement 中只包含一个 DDL 操作，或者在测试环境中验证一下，如果不支持，可以给 DM 提 issue。

如何处理不兼容的 DDL 语句？

你需要使用 dmctl 手动处理 TiDB 不兼容的 DDL 语句（包括手动跳过该 DDL 语句或使用用户指定的 DDL 语句替换原 DDL 语句，详见处理出错的 DDL 语句）。

> **注意：**
>
> TiDB 目前并不兼容 MySQL 支持的所有 DDL 语句。

DM 是否会将视图的 DDL 语句和对视图的 DML 语句同步到下游的 TiDB 中？

目前 DM 不会将视图的 DDL 语句同步到下游的 TiDB 集群，也不会将针对视图的 DML 语句同步到下游。

如何重置数据迁移任务？

当数据迁移过程中发生异常且无法恢复时，需要重置数据迁移任务，对数据重新进行迁移：

1. 使用 `stop-task` 停止异常的数据迁移任务。
2. 清理下游已迁移的数据。
3. 从下面两种方式中选择其中一种重启数据迁移任务：

- 修改任务配置文件以指定新的任务名，然后使用 `start-task {task-config-file}` 重启迁移任务。
- 使用 `start-task --remove-meta {task-config-file}` 重启数据迁移任务。

设置了 `online-ddl-scheme: "gh-ost"`，gh-ost 表相关的 DDL 报错该如何处理？

```
[unit=Sync] ["error information"="{\"msg\":\"[code=36046:class=sync-unit:scope=internal:level=
    ↪ high] online ddls on ghost table `xxx`.`_xxxx_gho`\\ngithub.com/pingcap/dm/pkg/terror.(*
    ↪ Error).Generate ......
```

出现上述错误可能有以下原因：

DM 在最后 `rename ghost_table to origin table` 的步骤会把内存的 DDL 信息读出，并且还原为 origin table 的 DDL。而内存中的 DDL 信息是在 `alter ghost_table` 的时候进行处理，记录 ghost_table DDL 的信息；或者是在重启 dm-worker 启动 task 的时候，从 `dm_meta.{task_name}_onlineddl` 中读取出来。

因此，如果在增量复制过程中，指定的 Pos 跳过了 `alter ghost_table` 的 DDL，但是该 Pos 仍在 gh-ost 的 online-ddl 的过程中，就会因为 ghost_table 没有正确写入到内存以及 `dm_meta.{task_name}_onlineddl`，而导致该问题。

可以通过以下方式绕过这个问题：

1. 取消 task 的 `online-ddl-schema` 的配置。
2. 把 `_{table_name}_gho`、`_{table_name}_ghc`、`_{table_name}_del` 配置到 `block-allow-list.ignore-tables` 中。
3. 手工在下游的 TiDB 执行上游的 DDL。
4. 待 Pos 复制到 gh-ost 整体流程后的位置，再重新启用 `online-ddl-schema` 以及注释掉 `block-allow-list.ignore-tables`。

如何为已有迁移任务增加需要迁移的表？

假如已有数据迁移任务正在运行，但又有其他的表需要添加到该迁移任务中，可根据当前数据迁移任务所处的阶段按下列方式分别进行处理。

> 注意：
> 向已有数据迁移任务中增加需要迁移的表操作较复杂，请仅在确有强烈需求时进行。

迁移任务当前处于 Dump 阶段

由于 MySQL 不支持指定 snapshot 来进行导出，因此在导出过程中不支持更新迁移任务并重启以通过断点继续导出，故无法支持在该阶段动态增加需要迁移的表。

如果确实需要增加其他的表用于迁移，建议直接使用新的配置文件重新启动迁移任务。

迁移任务当前处于 Load 阶段

多个不同的数据迁移任务在导出时，通常对应于不同的 binlog position，如将它们在 Load 阶段合并导入，则无法就 binlog position 达成一致，因此不建议在 Load 阶段向数据迁移任务中增加需要迁移的表。

迁移任务当前处于 Sync 阶段

当数据迁移任务已经处于 Sync 阶段时，在配置文件中增加额外的表并重启任务，DM 并不会为新增的表重新执行全量导出与导入，而是会继续从之前的断点进行增量复制。

因此，如果需要新增的表对应的全量数据尚未导入到下游，则需要先使用单独的数据迁移任务将其全量数据导出并导入到下游。

将已有迁移任务对应的全局 checkpoint （ `is_global=1` ）中的 position 信息记为 `checkpoint-T`，如 `(mysql-bin.000100, 1234)`。将需要增加到迁移任务的表在全量导出的 `metedata`（或另一个处于 Sync 阶段的数据迁移任务的 checkpoint ）的 position 信息记为 `checkpoint-S`，如 `(mysql-bin.000099, 5678)`。则可通过以下步骤将表增加到迁移任务中：

1. 使用 `stop-task` 停止已有迁移任务。如果需要增加的表属于另一个运行中的迁移任务，则也将其停止。
2. 使用 MySQL 客户连接到下游 TiDB 数据库，手动更新已有迁移任务对应的 checkpoint 表中的信息为 `checkpoint-T` 与 `checkpoint-S` 中的较小值（在本例中，为 `(mysql-bin.000099, 5678)`）。
    - 需要更新的 checkpoint 表为 `{dm_meta}` 库中的 `{task-name}_syncer_checkpoint`。
    - 需要更新的 checkpoint 行为 `id={source-id}` 且 `is_global=1`。
    - 需要更新的 checkpoint 列为 `binlog_name` 与 `binlog_pos`。
3. 在迁移任务配置中为 `syncers` 部分设置 `safe-mode: true` 以保证可重入执行。
4. 通过 `start-task` 启动迁移任务。
5. 通过 `query-status` 观察迁移任务状态，当 `syncerBinlog` 超过 `checkpoint-T` 与 `checkpoint-S` 中的较大值后（在本例中，为 `(mysql-bin.000100, 1234)`），即可还原 `safe-mode` 为原始值并重启迁移任务。

全量导入过程中遇到报错 `packet for query is too large. Try adjusting the 'max_allowed_packet' variable`

尝试将

- TiDB Server 的全局变量 `max_allowed_packet`
- 任务配置文件中的配置项 `target-database.max-allowed-packet`（详情参见DM 任务完整配置文件介绍）

设置为比默认 67108864 (64M) 更大的值。

2.0+ 集群运行 1.0 已有数据迁移任务时报错 `Error 1054: Unknown column 'binlog_gtid' in 'field list'`

在 DM 2.0 之后，为 checkpoint 等元信息表引入了更多的字段。如果通过 `start-task` 直接使用 1.0 集群的任务配置文件从增量复制阶段继续运行，则会出现 `Error 1054: Unknown column 'binlog_gtid' in 'field list'` 错误。

对于此错误，可手动将 DM 1.0 的数据迁移任务导入到 2.0+ 集群。

TiUP 无法部署 DM 的某个版本（如 v2.0.0-hotfix）

你可以通过 `tiup list dm-master` 命令查看 TiUP 支持部署的 DM 版本。该命令未展示的版本不能由 TiUP 管理。

DM 同步报错 `parse mydumper metadata error: EOF`

该错误需要查看报错信息以及日志进一步分析。报错原因可能是 dump 单元由于缺少权限没有产生正确的 metadata 文件。

DM 分库分表同步中没有明显报错，但是下游数据丢失

需要检查配置项 `block-allow-list` 和 `table-route`：

- `block-allow-list` 填写的是上游数据库表，可以在 `do-tables` 前通过加 “~” 来进行正则匹配。
- `table-route` 不支持正则，采用的是通配符模式，所以 `table_parttern_[0-63]` 只会匹配 table_parttern_0 到 table_pattern_6 这 7 张表。

DM 上游无写入，replicate lag 监控无数据

在 DM v1.0 中，需要开启 `enable-heartbeat` 才会产生该监控数据。v2.0 及以后版本中，尚未启用该功能，replicate lag 监控无数据是预期行为。

DM v2.0.0 启动任务时出现 `fail to initial unit Sync of subtask`，报错信息的 RawCause 显示 `context deadline exceeded`

该问题是 DM v2.0.0 的已知问题，在同步任务的表数目较多时触发，将在 v2.0.1 修复。使用 TiUP 部署的用户可以升级到开发版 nightly 解决该问题，或者访问 GitHub 上 DM 仓库的 release 页面下载 v2.0.0-hotfix 版本手动替换可执行文件。

DM 同步中报错 `duplicate entry`

用户需要首先确认任务中没有配置 `disable-detect`（v2.0.7 及之前版本），没有其他同步程序或手动插入数据，表中没有配置相关的 DML 过滤器。

为了便于排查问题，用户收集到下游 TiDB 相关 general log 后可以在 AskTUG 社区 联系专家进行排查。收集 general log 的方式如下：

```
##### 开启 general log
curl -X POST -d "tidb_general_log=1" http://{TiDBIP}:10080/settings
##### 关闭 general log
curl -X POST -d "tidb_general_log=0" http://{TiDBIP}:10080/settings
```

在发生 `duplicate entry` 报错时，确认日志中包含冲突数据的记录。

监控中部分面板显示 `No data point`

请参照DM 监控指标查看各面板含义，部分面板没有数据是正常现象。例如没有发生错误、不存在 DDL lock、没有启用 relay 功能等情况，均可能使得对应面板没有数据。

DM v1.0 在任务出错时使用 `sql-skip` 命令无法跳过某些语句

首先需要检查执行 `sql-skip` 之后 binlog 位置是否在推进，如果是的话表示 `sql-skip` 已经生效。重复出错的原因是上游发送了多个不支持的 DDL，可以通过 `sql-skip -s <sql-pattern>` 进行模式匹配。

对于类似下面这种报错（报错中包含 `parse statement`）：

```
if the DDL is not needed, you can use a filter rule with \"*\" schema-pattern to ignore it.\n\t :
    ↪  parse statement: line 1 column 11 near \"EVENT `event_del_big_table` \r\nDISABLE\" %!(
    ↪ MISSING)(EXTRA string=ALTER EVENT `event_del_big_table` \r\nDISABLE
```

出现报错的原因是 TiDB parser 无法解析上游的 DDL，例如 `ALTER EVENT`，所以 `sql-skip` 不会按预期生效。可以在任务配置文件中添加Binlog 过滤规则进行过滤，并设置 `schema-pattern: "*"`。从 DM 2.0.1 版本开始，已预设过滤了 EVENT 相关语句。

在 DM v6.0 版本之后 `sql-skip`、`handle-error` 均已经被 `binlog` 替代，使用 `binlog` 命令可以跳过该类错误。

DM 同步时下游长时间出现 REPLACE 语句

请检查是否符合safe mode 触发条件。如果任务发生错误并自动恢复，或者发生高可用调度，会满足“启动或恢复任务的前 1 分钟”这一条件，因此启用 safe mode。

可以检查 DM-worker 日志，在其中搜索包含 `change count` 的行，该行的 `new count` 非零时会启用 safe mode。检查 safe mode 启用时间以及启用前是否有报错，以定位启用原因。

使用 DM v2.0 同步数据时重启 DM 进程，出现全量数据导入失败错误

在 DM v2.0.1 及更早版本中，如果全量导入操作未完成时发生重启，重启后的上游数据源与 DM worker 的绑定关系可能会发生变化。例如，可能出现 dump 单元的中间数据在 DM worker A 机器上，但却由 DM worker B 进行 load 单元的情况，进而导致操作失败。

该情况有两种解决方案：

- 如果数据量较小（TB 级以下）或任务有合库合表：清空下游数据库的已导入数据，同时清空导出数据目录，使用 dmctl 删除并 `start-task --remove-meta` 重建任务。后续尽量保证全量导出导入阶段 DM 没有冗余 worker 以及避免在该时段内重启或升级 DM 集群。
- 如果数据量较大（数 TB 或更多）：清空下游数据库的已导入数据，将 lightning 部署到数据所在的 DM worker 节点，使用lightning local backend 模式 导入 DM dump 单元导出的数据。全量导入完成后，修改任务的 `task-mode` 为 `incremental`，修改 `mysql-instance.meta.pos` 为 dump 单元导出数据 `metadata` 中记录的位置，启动一个增量任务。

使用 DM 同步数据时重启 DM 进程，增量任务出现 `ERROR 1236 (HY000): The slave is connecting using CHANGE MASTER TO MASTER_AUTO_POSITION = 1, but the master has purged binary logs containing GTIDs that the slave requires.` 错误

该错误表明全量迁移期间，dump 单元记录 metadata 中的 binlog 位置已经被上游清理。

解决方案：出现该问题时只能清空下游数据库已同步数据，并在停止任务后加上 `--remove-meta` 参数重建任务。

如要提前避免该问题，需要进行以下配置：

1. 在 DM 全量迁移未完成时调大上游 MySQL 的 `expire_logs_days` 变量，保证全量进行结束时 metadata 中的 binlog 位置到当前时间的 binlog 都还没有被清理掉。如果数据量较大，应该使用 dumpling + lightning 的方式加快全量迁移速度。
2. DM 任务开启 relay log 选项，保证 binlog 被清理后 DM 仍有 relay log 可读取。

使用 TiUP v1.3.0, v1.3.1 部署 DM 集群，DM 集群的 grafana 监控报错显示 `failed to fetch dashboard`

该问题为 TiUP 已知 bug，在 TiUP v1.3.2 中已进行修复。可采取以下任一方法解决：

- 方法一：使用 `tiup update --self && tiup update dm` 升级 TiUP 到更新版本，随后先缩容再扩容集群中的 grafana 节点，重建 grafana 服务。

- 方法二：

    1. 备份 deploy/grafana-$port/bin/public 文件夹。
    2. 下载 TiUP DM 离线镜像包，并进行解压，将其中的 grafana-v4.0.3-**.tar.gz 文件解压后，用解压出的 public/ 文件夹替换前面所描述的文件夹，运行 tiup dm restart $cluster_name -R grafana 重启 grafana 服务监控。

在 DM v2.0 中，同时开启 relay 与 gtid 同步 MySQL 时，运行 query-status 发现 syncer checkpoint 中 GTID 不连续

该问题为 DM 已知 bug，在完全满足以下两个条件时将会触发，DM 将在 v2.0.2 修复该问题：

1. DM 配置的 source 同时设置了 enable-relay 与 enable-gtid 为 true
2. DM 同步上游为 MySQL 从库，并且该从库通过 show binlog events in '<newest-binlog>' limit 2 查询出的 previous_gtids 区间不连续，例如：

```
mysql> show binlog events in 'mysql-bin.000005' limit 2;
+------------------+------+---------------+-----------+-------------+-----------------------------------------
    ↪
| Log_name         | Pos  | Event_type    | Server_id | End_log_pos | Info
    ↪                                                                |
+------------------+------+---------------+-----------+-------------+-----------------------------------------
    ↪
| mysql-bin.000005 |    4 | Format_desc   |    123452 |         123 | Server ver: 5.7.32-35-log,
    ↪  Binlog ver: 4                          |
| mysql-bin.000005 |  123 | Previous_gtids |    123452 |         194 | d3618e68-6052-11eb-a68b
    ↪ -0242ac110002:6-7                          |
+------------------+------+---------------+-----------+-------------+-----------------------------------------
    ↪
```

使用 dmctl 的 query-status <task> 指令查询任务信息，如果已经出现 subTaskStatus.sync.syncerBinlogGtid 不连续但 subTaskStatus.sync.masterBinlogGtid 连续的情况，例如下述例子：

```
query-status test
{
    ...
    "sources": [
        {
            ...
            "sourceStatus": {
                "source": "mysql1",
                ...
                "relayStatus": {
                    "masterBinlog": "(mysql-bin.000006, 744)",
                    "masterBinlogGtid": "f8004e25-6067-11eb-9fa3-0242ac110003:1-50",
                    ...
                }
            },
```

```
            "subTaskStatus": [
                {
                    ...
                    "sync": {
                        ...
                        "masterBinlog": "(mysql-bin.000006, 744)",
                        "masterBinlogGtid": "f8004e25-6067-11eb-9fa3-0242ac110003:1-50",
                        "syncerBinlog": "(mysql-bin|000001.000006, 738)",
                        "syncerBinlogGtid": "f8004e25-6067-11eb-9fa3-0242ac110003:1-20:40-49",
                        ...
                        "synced": false,
                        "binlogType": "local"
                    }
                }
            ]
        },
        {
            ...
            "sourceStatus": {
                "source": "mysql2",
                ...
                "relayStatus": {
                    "masterBinlog": "(mysql-bin.000007, 1979)",
                    "masterBinlogGtid": "ddb8974e-6064-11eb-8357-0242ac110002:1-25",
                    ...
                }
            },
            "subTaskStatus": [
                {
                    ...
                    "sync": {
                        "masterBinlog": "(mysql-bin.000007, 1979)",
                        "masterBinlogGtid": "ddb8974e-6064-11eb-8357-0242ac110002:1-25",
                        "syncerBinlog": "(mysql-bin|000001.000008, 1979)",
                        "syncerBinlogGtid": "ddb8974e-6064-11eb-8357-0242ac110002:1-25",
                        ...
                        "synced": true,
                        "binlogType": "local"
                    }
                }
            ]
        }
    ]
}
```

其中 mysql1 的 syncerBinlogGtid 不连续，已有数据丢失需要按下述方案之一处理：

- 如果全量导出任务 metadata 中的 position 到当前时间的上游数据库的 binlog 仍未被清理：
    1. 停止当前任务并删除所有 GTID 不连续的 source
    2. 设置所有 source 的 enable-relay 为 false
    3. 针对 GTID 不连续的 source（上例 mysql1），在对应的任务配置文件 task.yaml 中，把 task-mode 修改为 incremental 并配置增量任务起始点 mysql-instances.meta 为各个全量导出任务 metadata 的 binlog name，position 和 gtid 信息
    4. 配置 task.yaml 中的 syncers.safe-mode 为 true 并重启任务
    5. 待增量同步追上后，停止任务并在任务配置文件中设置 safe-mode 为 false
    6. 再次重启任务
- 如果上游数据库 binlog 已被清理但是本地 relay log 仍未被清理：
    1. 停止当前任务
    2. 针对 GTID 不连续的 source（上例 mysql1），在对应的任务配置文件 task.yaml 中，把 task-mode 修改为 incremental 并配置增量任务起始点 mysql-instances.meta 为各个全量导出任务 metadata 的 binlog name，position 和 gtid 信息
    3. 修改其中的 GTID 信息的 1-y 为 previous_gtids 的前段值，例如，上述例子需要改为 6-y
    4. 配置 task.yaml 中的 syncers.safe-mode 为 true 并重启任务
    5. 待增量同步追上后，停止任务并在任务配置文件中设置 safe-mode 为 false
    6. 再次重启任务
    7. 重启 source 并关闭 gtid 或 relay
- 如果上述条件均不满足或任务同步数据量较小：
    1. 清空下游数据库中数据
    2. 重启 source 并关闭 gtid 或 relay
    3. 重建任务并通过 start-task task.yaml --remove-meta 重新同步

上述处理方案中，针对正常同步的 source（如上例 mysql2），重设增量任务时起始点需设置 mysql-instances.
↪ meta 为 subTaskStatus.sync 的 syncerBinlog 与 syncerBinlogGtid。

在 DM 2.0 中开启 heartbeat，虚拟 IP 环境下切换 DM-worker 与 MySQL 实例的连接，遇到 “heartbeat config is different from previous used: serverID not equal” 错误

heartbeat 功能在 DM v2.0 及之后版本已经默认关闭，如果用户在同步任务配置文件中开启会干扰高可用特性，在配置文件中关闭该项（通过设置 enable-heartbeat: false，然后更新任务配置）即可解决。DM 将会在后续版本强制关闭该功能。

DM-master 在重启后无法加入集群，报错信息为 “fail to start embed etcd, RawCause: member xxx has already been bootstrapped”

DM-master 会在启动时将 etcd 信息记录在当前目录。如果重启后当前目录发生变化，会导致 DM 缺失 etcd 信息，从而启动失败。

推荐使用 TiUP 运维 DM 避免这一问题。在需要使用二进制部署的场合，需要在 DM-master 配置文件中使用绝对路径配置 data-dir 项，或者注意运行命令的当前目录。

使用 dmctl 执行命令时无法连接 DM-master

在使用 dmctl 执行相关命令时，发现连接 DM-master 失败（即使已在命令中指定 `--master-addr` 的参数值），报错内容类似 `RawCause: context deadline exceeded, Workaround: please check your network connection.`，但使用 `telnet <master-addr>` 之类的命令检查网络却没有发现异常。

这种情况可以检查下环境变量 `https_proxy`（注意，这里是 https ）。如果配置了该环境变量，dmctl 会自动去连接 `https_proxy` 指定的主机及端口，而如果该主机没有相应的 `proxy` 转发服务，则会导致连接失败。

解决方案：确认 `https_proxy` 是否必须要配置，如果不是必须的，取消该设置即可。如果环境必须，那么在原命令前加环境变量设置 `https_proxy="" ./dmctl --master-addr "x.x.x.x:8261"` 即可。

> **注意：**
>
> 关于 `proxy` 的环境变量有 `http_proxy`，`https_proxy`，`no_proxy` 等。如果依据上述解决方案处理后仍无法连接，可以考虑检查 `http_proxy` 和 `no_proxy` 的参数配置是否有影响。

v2.0.2 - v2.0.6 版本执行 start-relay 命令报错该如何处理？

```
flush local meta, Rawcause: open relay-dir/xxx.000001/relay.metayyyy: no such file or directory
```

上述报错在以下情况下有可能会被触发：

- DM 从 v2.0.1 及之前的版本升级到 v2.0.2 - v2.0.6 版本，且升级之前曾开启过 relay log，升级完后重新开启。
- 使用 stop-relay 命令暂停 relay log 后重新开启 relay log。

可以通过以下方式解决该问题：

- 重启 relay log:

  ```
  » stop-relay -s sourceID workerName
  » start-relay -s sourceID workerName
  ```

- 升级 DM 至 v2.0.7 或之后版本。

Load 单元报错 `Unknown character set`

由于 TiDB 只支持部分 MySQL 字符集，因此，在全量导入中，如果创建表结构时使用了 TiDB 不支持的字符集，DM 会报这个错误。你可以结合数据内容选择TiDB 支持的字符集，预先在下游创建表结构以绕过这个错误。

13.6.9.12.2 TiDB Data Migration 故障及处理方法

本文档介绍 TiDB Data Migration (DM) 的错误系统及常见故障的处理方法。

DM 错误系统

在 DM 的错误系统中，对于一条特定的错误，通常主要包含以下信息：

- code：错误码。

  同一种错误都使用相同的错误码。错误码不随 DM 版本改变。

  在 DM 迭代过程中，部分错误可能会被移除，但错误码不会。新增的错误会使用新的错误码，不会复用已有的错误码。

- class：发生错误的类别。

  用于标记出现错误的系统子模块。

  下表展示所有的错误类别、错误对应的系统子模块、错误样例：

  |

  错误类别

错误对应的系统子模块	错误样例

:————- | :———————— | :———————————————— |
database | 执行数据库操作出现错误 | [code=10003:class=database:scope=downstream:level=medium] database driver: invalid connection |
functional | 系统底层的基础函数错误 | [code=11005:class=functional:scope=internal:level=high] not allowed operation: alter multiple tables in one statement |
config | 配置错误 | [code=20005:class=config:scope=internal:level=medium] empty source-id not valid |
binlog-op | binlog 操作出现错误 | [code=22001:class=binlog-op:scope=internal:level=high] empty UUIDs not valid |
checkpoint | checkpoint 相关操作出现错误 | [code=24002:class=checkpoint:scope=internal:level=high] save point bin.1234 is older than current pos bin.1371 |
task-check | 进行任务检查时出现错误 | [code=26003:class=task-check:scope=internal:level=medium] new table router error |
relay-event-lib| 执行 relay 模块基础功能时出现错误 | [code=28001:class=relay-event-lib:scope=internal:level=high] parse server-uuid.index |
relay-unit | relay 处理单元内出现错误 | [code=30015:class=relay-unit:scope=upstream:level=high] TCPReader get event: ERROR 1236 (HY000): Could not open log file |
dump-unit | dump 处理单元内出现错误 | [code=32001:class=dump-unit:scope=internal:level=high] mydumper runs with error: CRITICAL **: 15:12:17.559: Error connecting to database: Access denied for user 'root'@'172.17.0.1' (using password: NO) |
load-unit | load 处理单元内出现错误 | [code=34002:class=load-unit:scope=internal:level=high] corresponding ending of sql: ')' not found |
sync-unit | sync 处理单元内出现错误 | [code=36027:class=sync-unit:scope=internal:level=high] Column count doesn't match value count: 9 (columns)vs 10 (values) |
dm-master | DM-master 服务内部出现错误 | [code=38008:class=dm-master:scope=internal:level=high] grpc request error: rpc error: code = Unavailable desc = all SubConns are in TransientFailure, latest connection error: connection error: desc = "transport: Error while dialing dial tcp 172.17.0.2:8262: connect: connection refused" |
dm-worker | DM-worker 服务内部出现错误 | [code=40066:class=dm-worker:scope=internal:level=high] ExecuteDDL timeout, try use query-status to query whether the DDL is still blocking |

```
dm-tracer | DM-tracer 服务内部出现错误 | [code=42004:class=dm-tracer:scope=internal:level=medium
↪ ] trace event test.1 not found |
schema-tracer | 增量复制时记录 schema 变更出现错误 | [code=44006:class=schema-tracker:scope=
↪ internal:level=high],"cannot track DDL: ALTER TABLE test DROP COLUMN col1" |
scheduler | 数据迁移任务调度相关操作出错操作 | [code=46001:class=scheduler:scope=internal:
↪ level=high],"the scheduler has not started" |
dmctl | dmctl 内部或与其他组件交互出现错误 | [code=48001:class=dmctl:scope=internal:level=high
↪ ],"can not create grpc connection" |
```

- `scope`：错误作用域。

  用于标识错误发生时 DM 作用对象的范围和来源，包括未设置 (`not-set`)、上游数据库 (`upstream`)、下游数据库 (`downstream`)、内部 (`internal`) 四种类型。

  如果错误发生的逻辑直接涉及到上下游数据库请求，作用域会设置为 `upstream` 或 `downstream`，其他出错场景目前都设置为 `internal`。

- `level`：错误级别。

  错误的严重级别，包括低级别 (`low`)、中级别 (`medium`)、高级别 (`high`) 三种。

  低级别通常是用户操作、输入错误，不影响正常迁移任务；中级别通常是用户配置等错误，会影响部分新启动服务，不影响已有系统迁移状态；高级别通常是用户需要关注的一些错误，可能存在迁移任务中断等风险，需要用户进行处理。

- `message`：错误描述。

  错误的详细描述信息。对于错误调用链上每一层额外增加的错误 message，采用 errors.Wrap 的模式来叠加和保存错误 message。wrap 最外层的 message 是 DM 内部对该错误的描述，wrap 最内层的 message 是该错误最底层出错位置的错误描述。

- workaround: 错误处理方法（可选）。

  对该错误的处理方法。对于部分明确的错误（如：配置信息错误等），DM 会在 `workaround` 中给出相应的人为处理方法。

- 错误堆栈信息（可选）。

  DM 根据错误的严重程度和必要性来选择是否输出错误堆栈。错误堆栈记录了错误发生时完整的堆栈调用信息。如果用户通过错误基本信息和错误 message 描述不能完全诊断出错误发生的原因，可以通过错误堆栈进一步跟进出错时代码的运行路径。

可在 DM 代码仓库已发布的错误码 中查询完整的错误码列表。

DM 故障诊断

如果在运行 DM 工具时出现了错误，请尝试以下解决方案：

1. 执行 `query-status` 命令查看任务运行状态以及相关错误输出。
2. 查看与该错误相关的日志文件。日志文件位于 DM-master、DM-worker 部署节点上，通过DM 错误系统 获取错误的关键信息，然后查看常见故障处理方法以寻找相应的解决方案。
3. 如果该错误还没有相应的解决方案，并且你无法通过查询日志或监控指标自行解决此问题，请从 PingCAP 官方或 TiDB 社区获取支持。

4. 一般情况下，错误处理完成后，只需使用 dmctl 重启任务即可。

```
resume-task ${task name}
```

但在某些情况下，你还需要重置数据迁移任务。有关何时需要重置以及如何重置，详见重置数据迁移任务。

常见故障处理方法

|

错误码

错误说明	解决方法

:———- | :———————————————— | :———————————————————- |
code=10001 | 数据库操作异常 | 进一步分析错误信息和错误堆栈 |
code=10002 | 数据库底层的 bad connection 错误，通常表示 DM 到下游 TiDB 的数据库连接出现了异常（如网络故障、TiDB 重启等）且当前请求的数据暂时未能发送到 TiDB。 | DM 提供针对此类错误的自动恢复。如果长时间未恢复，需要用户检查网络或 TiDB 状态。 |
code=10003 | 数据库底层 invalid connection 错误，通常表示 DM 到下游 TiDB 的数据库连接出现了异常（如网络故障、TiDB 重启、TiKV busy 等）且当前请求已有部分数据发送到了 TiDB。 | DM 提供针对此类错误的自动恢复。如果未能正常恢复，需要用户进一步检查错误信息并根据具体场景进行分析。 |
code=10005 | 数据库查询类语句出错 | |
code=10006 | 数据库 EXECUTE 类型语句出错，包括 DDL 和 INSERT/UPDATE/DELETE 类型的 DML。更详细的错误信息可通过错误 message 获取。错误 message 中通常包含操作数据库所返回的错误码和错误信息。 | |
code=11006 | DM 内置的 parser 解析不兼容的 DDL 时出错 | 可参考Data Migration 故障诊断 -处理不兼容的 DDL 语句 提供的解决方案 |
code=20010 | 处理任务配置时，解密数据库的密码出错 | 检查任务配置中提供的下游数据库密码是否有使用 dmctl 正确加密 |
code=26002 | 任务检查创建数据库连接失败。更详细的错误信息可通过错误 message 获取。错误 message 中包含操作数据库所返回的错误码和错误信息。 | 检查 DM-master 所在的机器是否有权限访问上游 |
code=32001 | dump 处理单元异常 | 如果报错 msg 包含 mydumper: argument list too long.，则需要用户根据 block-allow-list，在 task.yaml 的 dump 处理单元的 extra-args 参数中手动加上 --regex 正则表达式设置要导出的库表。例如，如果要导出所有库中表名字为 hello 的表，可加上 --regex '.*\\.hello$'，如果要导出所有表，可加上 --regex '.*'。 |
code=38008 | DM 组件间的 gRPC 通信出错 | 检查 class，定位错误发生在哪些组件的交互环节，根据错误 message 判断是哪类通信错误。如果是 gRPC 建立连接出错，可检查通信服务端是否运行正常。 |

迁移任务中断并包含 invalid connection 错误

原因

发生 invalid connection 错误时，通常表示 DM 到下游 TiDB 的数据库连接出现了异常（如网络故障、TiDB 重启、TiKV busy 等）且当前请求已有部分数据发送到了 TiDB。

解决方案

由于 DM 中存在迁移任务并发向下游复制数据的特性，因此在任务中断时可能同时包含多个错误（可通过 query-status 查询当前错误）。

- 如果错误中仅包含 `invalid connection` 类型的错误且当前处于增量复制阶段，则 DM 会自动进行重试。
- 如果 DM 由于版本问题等未自动进行重试或自动重试未能成功，则可尝试先使用 `stop-task` 停止任务，然后再使用 `start-task` 重启任务。

迁移任务中断并包含 `driver: bad connection` 错误

原因

发生 `driver: bad connection` 错误时，通常表示 DM 到下游 TiDB 的数据库连接出现了异常（如网络故障、TiDB 重启等）且当前请求的数据暂时未能发送到 TiDB。

解决方案

当前版本 DM 会自动进行重试，如果由于版本问题等未自动重试，可先使用 `stop-task` 停止任务后再使用 `start-task` 重启任务。

relay 处理单元报错 `event from * in * diff from passed-in event *` 或迁移任务中断并包含 `get binlog error ERROR 1236 (HY000)`、`binlog checksum mismatch, data may be corrupted` 等 binlog 获取或解析失败错误

原因

在 DM 进行 relay log 拉取与增量复制过程中，如果遇到了上游超过 4GB 的 binlog 文件，就可能出现这两个错误。

原因是 DM 在写 relay log 时需要依据 binlog position 及文件大小对 event 进行验证，且需要保存迁移的 binlog position 信息作为 checkpoint。但是 MySQL binlog position 官方定义使用 uint32 存储，所以超过 4G 部分的 binlog position 的 offset 值会溢出，进而出现上面的错误。

解决方案

对于 relay 处理单元，可通过以下步骤手动恢复：

1. 在上游确认出错时对应的 binlog 文件的大小超出了 4GB。
2. 停止 DM-worker。
3. 将上游对应的 binlog 文件复制到 relay log 目录作为 relay log 文件。
4. 更新 relay log 目录内对应的 `relay.meta` 文件以从下一个 binlog 开始拉取。如果 DM worker 已开启 `enable_gtid`，那么在修改 `relay.meta` 文件时，同样需要修改下一个 binlog 对应的 GTID。如果未开启 `enable_gtid` 则无需修改 GTID。

   例如：报错时有 `binlog-name = "mysql-bin.004451"` 与 `binlog-pos = 2453`，则将其分别更新为 `binlog-name = "mysql-bin.004452"` 和 `binlog-pos = 4`，同时更新 `binlog-gtid = "f0e914ef-54cf-11e7-813d-6c92bf2fa791:1-138218058"`。
5. 重启 DM-worker。

对于 binlog replication 处理单元，可通过以下步骤手动恢复：

1. 在上游确认出错时对应的 binlog 文件的大小超出了 4GB。
2. 通过 `stop-task` 停止迁移任务。

3. 将下游 dm_meta 数据库中 global checkpoint 与每个 table 的 checkpoint 中的 binlog_name 更新为出错的 binlog 文件，将 binlog_pos 更新为已迁移过的一个合法的 position 值，比如 4。

   例如：出错任务名为 dm_test，对应的 source-id 为 replica-1，出错时对应的 binlog 文件为 mysql-bin|000001.004451，则执行 UPDATE dm_test_syncer_checkpoint SET binlog_name='mysql-bin|000001.004451', binlog_pos = 4 WHERE id='replica-1';。

4. 在迁移任务配置中为 syncers 部分设置 safe-mode: true 以保证可重入执行。
5. 通过 start-task 启动迁移任务。
6. 通过 query-status 观察迁移任务状态，当原造成出错的 relay log 文件迁移完成后，即可还原 safe-mode 为原始值并重启迁移任务。

执行 query-status 或查看日志时出现 Access denied for user 'root'@'172.31.43.27' (using password: YES)

在所有 DM 配置文件中，数据库相关的密码都推荐使用经 dmctl 加密后的密文（若数据库密码为空，则无需加密）。有关如何使用 dmctl 加密明文密码，参见使用 dmctl 加密数据库密码。

此外，在 DM 运行过程中，上下游数据库的用户必须具备相应的读写权限。在启动迁移任务过程中，DM 会自动进行相应权限的前置检查，详见上游 MySQL 实例配置前置检查。

load 处理单元报错 packet for query is too large. Try adjusting the 'max_allowed_packet' variable

原因

- MySQL client 和 MySQL/TiDB Server 都有 max_allowed_packet 配额的限制，如果在使用过程中违反其中任何一个 max_allowed_packet 配额，客户端程序就会收到对应的报错。目前最新版本的 DM 和 TiDB Server 的默认 max_allowed_packet 配额都为 64M。
- DM 的全量数据导入处理模块不支持对 dump 处理模块导出的 SQL 文件进行切分。

解决方案

- 推荐在 DM 的 dump 处理单元提供的配置 extra-args 中设置 statement-size:

  依据默认的 --statement-size 设置，DM 的 dump 处理单元默认生成的 Insert Statement 大小一般会在 1M 左右，使用默认值就可以确保绝大部分情况 load 处理单元不会报错 packet for query is too large. Try adjusting the 'max_allowed_packet' variable。

  有时候在 dump 过程中会出现下面的 WARN log。这个 WARN log 不影响 dump 的过程，只是说明 dump 的表可能是宽表。

  ```
  Row bigger than statement_size for xxx
  ```

- 如果宽表的单行超过了 64M，需要修改以下两项配置，并且确保其生效。
  - 在 TiDB Server 执行 set @@global.max_allowed_packet=134217728（134217728 = 128M）
  - 根据实际情况为 DM 的任务配置文件中的 target-database 增加配置 max-allowed-packet: 134217728 (128M)，执行 stop-task 后再重新 start-task。

13.6.9.13 TiDB Data Migration 版本发布历史

从 DM v5.4.0 起，TiDB Data Migration 的 Release Notes 合并入相同版本号的 TiDB Release Notes。

- 如需阅读 v5.4.0 及之后版本的 DM Release Notes，请查看对应版本的 TiDB Release Notes 中 DM 相关的内容。
- 如需阅读 v5.3.0 及更早版本的 DM Release Notes，请参考以下链接：

13.6.9.13.1 5.3

- 5.3.0

13.6.9.13.2 2.0

- 2.0.7
- 2.0.6
- 2.0.5
- 2.0.4
- 2.0.3
- 2.0.2
- 2.0.1
- 2.0 GA
- 2.0.0-rc.2
- 2.0.0-rc

13.6.9.13.3 1.0

- 1.0.7
- 1.0.6
- 1.0.5
- 1.0.4
- 1.0.3
- 1.0.2

13.7 TiDB Lightning

13.7.1 TiDB Lightning 简介

TiDB Lightning 是用于从静态文件导入 TB 级数据到 TiDB 集群的工具，常用于 TiDB 集群的初始化数据导入。

要快速了解 Lightning 的基本原理和使用方法，建议先观看下面的培训视频（时长 32 分钟）。注意本视频只为学习参考，具体操作步骤和最新功能，请以文档内容为准。

TiDB Lightning 支持以下文件类型：

- Dumpling 生成的文件

- CSV 文件
- Amazon Aurora 生成的 Apache Parquet 文件

TiDB Lightning 支持从以下位置读取：

- 本地
- Amazon S3
- Google GCS

13.7.1.1 TiDB Lightning 整体架构

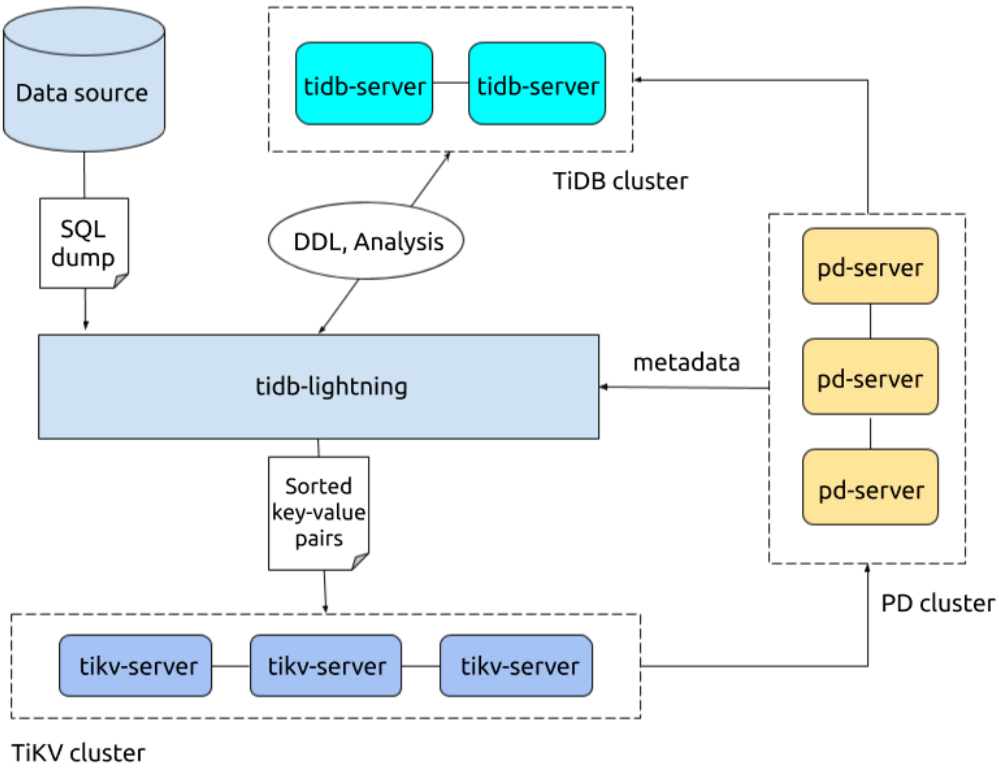


图 247: TiDB Lightning 整体架构

TiDB Lightning 目前支持两种导入方式，通过 backend 配置区分。不同的模式决定 TiDB Lightning 如何将数据导入到目标 TiDB 集群。

- Physical Import Mode：TiDB Lightning 首先将数据编码成键值对并排序存储在本地临时目录，然后将这些键值对上传到各个 TiKV 节点，最后调用 TiKV Ingest 接口将数据插入到 TiKV 的 RocksDB 中。如果用于初始化导

入，请优先考虑使用 Physical Import Mode，其拥有较高的导入速度。Physical Import Mode 对应的后端模式为 `local`。

- Logical Import Mode：TiDB Lightning 先将数据编码成 SQL，然后直接运行这些 SQL 语句进行数据导入。如果需要导入的集群为生产环境线上集群，或需要导入的目标表中已包含有数据，则应使用 Logical Import Mode。Logical Import Mode 对应的后端模式为 `tidb`。

导入模式	Physical Import Mode	Logical Import Mode
后端	`local`	`tidb`
速度	快 (100 ~ 500 GiB/小时)	慢 (10 ~ 50 GiB/小时)
资源使用率	高	低
占用网络带宽	高	低
导入时是否满足 ACID	否	是
目标表	必须为空	可以不为空
支持 TiDB 集群版本	>= v4.0.0	全部
导入期间是否允许 TiDB 对外提供服务	受限制	是

> **注意：**
>
> 以上性能数据用于对比两种模式的导入性能差异，实际导入速度受硬件配置、表结构、索引数量等多方面因素影响。

13.7.2 TiDB Lightning 快速上手

本教程假设使用的是若干新的、纯净版 CentOS 7 实例，你可以（使用 VMware、VirtualBox 及其他工具）在本地虚拟化或在供应商提供的平台上部署一台小型的云虚拟主机。因为 TiDB Lightning 对计算机资源消耗较高，建议分配 16 GB 以上的内存以及 32 核以上的 CPU 以获取最佳性能。

> **警告：**
>
> 本教程中的部署方法只适用于测试及功能体验，并不适用于生产或开发环境。

13.7.2.1 准备全量备份数据

我们使用dumpling 从 MySQL 导出数据，如下：

```
tiup dumpling -h 127.0.0.1 -P 3306 -u root -t 16 -F 256MB -B test -f 'test.t[12]' -o /data/
    ↪ my_database/
```

其中：

- `-B test`：从 `test` 数据库导出。
- `-f test.t[12]`：只导出 `test.t1` 和 `test.t2` 这两个表。
- `-t 16`：使用 16 个线程导出数据。
- `-F 256MB`：将每张表切分成多个文件，每个文件大小约为 256 MB。

这样全量备份数据就导出到了 `/data/my_database` 目录中。

13.7.2.2 部署 TiDB Lightning

13.7.2.2.1 第 1 步：部署 TiDB 集群

在开始数据导入之前，需先部署一套要进行导入的 TiDB 集群。本教程以 TiDB v5.4.0 版本为例，具体部署方法可参考使用 TiUP 部署 TiDB 集群。

13.7.2.2.2 第 2 步：下载 TiDB Lightning 安装包

TiDB Lightning 的安装包位于 TiDB 离线工具包中。下载方式，请参考TiDB 工具下载。

13.7.2.2.3 第 3 步：启动 `tidb-lightning`

1. 将安装包里的 `bin/tidb-lightning` 及 `bin/tidb-lightning-ctl` 上传至部署 TiDB Lightning 的服务器。
2. 将数据源也上传到同样的服务器。
3. 配置 `tidb-lightning.toml`。

```
[lightning]
# 日志
level = "info"
file = "tidb-lightning.log"

[tikv-importer]
# 选择使用的导入模式
backend = "local"
# 设置排序的键值对的临时存放地址，目标路径需要是一个空目录
sorted-kv-dir = "/mnt/ssd/sorted-kv-dir"

[mydumper]
# 源数据目录。
data-source-dir = "/data/my_datasource/"

# 配置通配符规则，默认规则会过滤 mysql、sys、INFORMATION_SCHEMA、PERFORMANCE_SCHEMA、
    ↪ METRICS_SCHEMA、INSPECTION_SCHEMA 系统数据库下的所有表
# 若不配置该项，导入系统表时会出现“找不到 schema”的异常
filter = ['*.*', '!mysql.*', '!sys.*', '!INFORMATION_SCHEMA.*', '!PERFORMANCE_SCHEMA.*', '!
    ↪ METRICS_SCHEMA.*', '!INSPECTION_SCHEMA.*']
```

```
[tidb]
# 目标集群的信息
host = "172.16.31.2"
port = 4000
user = "root"
password = "rootroot"
# 表架构信息在从 TiDB 的“状态端口”获取。
status-port = 10080
# 集群 pd 的地址
pd-addr = "172.16.31.3:2379"
```

4. 配置合适的参数运行 tidb-lightning。如果直接在命令行中用 nohup 启动程序，可能会因为 SIGHUP 信号而退出，建议把 nohup 放到脚本里面，如：

```
#!/bin/bash
nohup tiup tidb-lightning -config tidb-lightning.toml > nohup.out &
```

13.7.2.2.4 第 4 步：检查数据

导入完毕后，TiDB Lightning 会自动退出。若导入成功，日志的最后一行会显示 tidb lightning exit。

如果出错，请参见TiDB Lightning 常见问题。

13.7.2.3 总结

本教程对 TiDB Lightning 进行了简单的介绍，并快速部署了一套简单的 TiDB Lightning 集群，将全量备份数据导入到 TiDB 集群中。

关于 TiDB Lightning 的详细功能和使用，参见TiDB Lightning 简介。

13.7.3 部署 TiDB Lightning

本文主要介绍 TiDB Lightning 进行数据导入的硬件需求，以及手动部署 TiDB Lightning 的方式。 Lightning 不同的导入模式，其硬件要求有所不同，请先阅读：

- Physical Import Mode 必要条件及限制
- Logical Import Mode 必要条件及限制

13.7.3.1 使用 TiUP 联网部署（推荐）

1. 执行如下命令安装 TiUP 工具：

```
curl --proto '=https' --tlsv1.2 -sSf https://tiup-mirrors.pingcap.com/install.sh | sh
```

安装完成后，~/.bashrc 已将 TiUP 加入到路径中，你需要新开一个终端或重新声明全局变量 source ~/.bashrc 来使用 TiUP。(也可能是~/.profile，以 TiUP 输出为准。)

2. 安装 TiUP Lightning 组件：

```
tiup install tidb-lightning
```

13.7.3.2 手动部署

13.7.3.2.1 下载 TiDB Lightning 安装包

参考工具下载文档下载 TiDB Lightning 安装包（TiDB Lightning 完全兼容较低版本的 TiDB 集群，建议选择最新稳定版本）。

解压 Lightning 压缩包即可获得 `tidb-lightning` 可执行文件。

```
tar -zxvf tidb-lightning-${version}-linux-amd64.tar.gz
chmod +x tidb-lightning
```

13.7.3.2.2 升级 TiDB Lightning

你可以通过替换二进制文件升级 TiDB Lightning，无需其他配置。重启 TiDB Lightning 的具体操作参见FAQ。

如果当前有运行的导入任务，推荐任务完成后再升级 TiDB Lightning。否则，你可能需要从头重新导入，因为无法保证断点可以跨版本工作。

13.7.4 TiDB Lightning 目标数据库要求

使用 TiDB Lightning 导入数据前，先检查环境是否满足要求，这有助于减少导入过程的错误，避免导入失败的情况。

13.7.4.1 目标数据库权限要求

TiDB Lightning 导入数据时，根据导入方式和启用特性等，需要下游数据库用户具备不同的权限，可参考下表：

```
<td></td>
<td>特性</td>
<td>作用域</td>
<td>所需权限</td>
<td>备注</td>
```

```
<td rowspan="2">必需</td>
<td rowspan="2">基本功能</td>
<td>目标 table</td>
<td>CREATE,SELECT,INSERT,UPDATE,DELETE,DROP,ALTER</td>
<td>DROP 仅 tidb-lightning-ctl 在执行 checkpoint-destroy-all 时需要</td>
```

```
<td>目标 database</td>
<td>CREATE</td>
<td></td>
```

```
<td rowspan="4">必需</td>
<td>Logical Import Mode</td>
<td>information_schema.columns</td>
<td>SELECT</td>
<td></td>
```

```
<td  rowspan="3">Physical Import Mode</td>
<td>mysql.tidb</td>
<td>SELECT</td>
<td></td>
```

```
<td>-</td>
<td>SUPER</td>
<td></td>
```

```
<td>-</td>
<td>RESTRICTED_VARIABLES_ADMIN,RESTRICTED_TABLES_ADMIN</td>
<td>当目标 TiDB 开启 SEM</td>
```

```
<td>推荐</td>
<td>冲突检测，max-error</td>
<td>lightning.task-info-schema-name 配置的 schema</td>
<td>SELECT,INSERT,UPDATE,DELETE,CREATE,DROP</td>
<td>如不需要，该值必须设为""</td>
```

```
<td>可选</td>
<td>并行导入</td>
<td>lightning.meta-schema-name 配置的 schema</td>
<td>SELECT,INSERT,UPDATE,DELETE,CREATE,DROP</td>
<td>如不需要，该值必须设为""</td>
```

```
<td>可选</td>
<td>checkpoint.driver = "mysql"</td>
<td>checkpoint.schema 设置</td>
<td>SELECT,INSERT,UPDATE,DELETE,CREATE,DROP</td>
<td>使用数据库而非文件形式存放 checkpoint 信息时需要</td>
```

13.7.4.2 目标数据库所需空间

目标 TiKV 集群必须有足够空间接收新导入的数据。除了标准硬件配置以外，目标 TiKV 集群的总存储空间必须大于 数据源大小 × 副本数量 × 2。例如集群默认使用 3 副本，那么总存储空间需为数据源大小的 6 倍以上。公式中的 2 倍可能难以理解，其依据是以下因素的估算空间占用：

- 索引会占据额外的空间

- RocksDB 的空间放大效应

目前无法精确计算 Dumpling 从 MySQL 导出的数据大小，但你可以用下面 SQL 语句统计信息表的 data_length 字段估算数据量：

统计所有 schema 大小，单位 MiB，注意修改 ${schema_name}

```
SELECT table_schema, SUM(data_length)/1024/1024 AS data_length, SUM(index_length)/1024/1024 AS
    ↪ index_length, SUM(data_length+index_length)/1024/1024 AS sum FROM information_schema.
    ↪ tables WHERE table_schema = "${schema_name}" GROUP BY table_schema;
```

统计最大单表，单位 MiB，注意修改 ${schema_name}

```
SELECT table_name, table_schema, SUM(data_length)/1024/1024 AS data_length, SUM(index_length)
    ↪ /1024/1024 AS index_length, SUM(data_length+index_length)/1024/1024 AS sum FROM
    ↪ information_schema.tables WHERE table_schema = "${schema_name}" GROUP BY table_name,
    ↪ table_schema ORDER BY sum  DESC LIMIT 5;
```

13.7.5 数据源

13.7.5.1 TiDB Lightning 数据源

TiDB Lightning 支持从多种类型的文件导入数据到 TiDB 集群。通过以下配置为 TiDB Lightning 指定数据文件所在位置。

```
[mydumper]
#### 本地源数据目录或 S3 等外部存储 URI。关于外部存储 URI 详情可参考 https://docs.pingcap.com/zh/
    ↪ tidb/v6.6/backup-and-restore-storages#uri-%E6%A0%BC%E5%BC%8F。
data-source-dir = "/data/my_database"
```

TiDB Lightning 运行时将查找 data-source-dir 中所有符合命令规则的文件。

文件类型	分类	命名规则
Schema 文件	包含 DDL 语句 CREATE TABLE 的文件	${ ↪ db_name ↪ }. ↪ ${ ↪ table_name ↪ }-schema ↪ . ↪ sql ↪

文件类型	分类	命名规则
Schema 文件	包含 CREATE DATABASE DDL 语句的文件	${db_name}-schema.sql
数据文件	包含整张表的数据文件，该文件会被导入	${db_name}.${table_name}.${csv|sql|parquet}
数据文件	如果一个表分布于多个数据文件，这些文件命名需加上文件编号的后缀	${db_name}.${table_name}.001.${csv|sql|parquet}

文件类型	分类	命名规则
压缩文件	上述所有类型文件如带压缩文件名后缀，如 gzip、snappy 或 zstd，TiDB Lightning 会流式解压后进行导入	${db_name}.${table_name}.${csv|sql|parquet}.{compress}

TiDB Lightning 尽量并行处理数据，由于文件必须顺序读取，所以数据处理协程是文件级别的并发（通过 region-concurrency 配置控制）。因此导入大文件时性能比较差。通常建议单个文件尺寸为 256MiB，以获得最好的性能。

13.7.5.1.1 CSV

表结构

CSV 文件是没有表结构的。要导入 TiDB，就必须为其提供表结构。可以通过以下任一方法实现：

- 创建包含 DDL 语句的 ${db_name}.${table_name}-schema.sql 和 ${db_name}-schema-create.sql。
- 在 TiDB 中手动创建。

配置

CSV 格式可在 tidb-lightning.toml 文件中 [mydumper.csv] 下配置。大部分设置项在 MySQL [LOAD DATA] 语句中都有对应的项目。

```
[mydumper.csv]
```

```
#### 字段分隔符，支持一个或多个字符，默认值为 ','。如果数据中可能有逗号，
    ↪ 建议源文件导出时分隔符使用非常见组合字符例如'|+|'。
separator = ','
#### 引用定界符，设置为空表示字符串未加引号。
delimiter = '"'
#### 行尾定界字符，支持一个或多个字符。设置为空（默认值）表示 "\n"（换行）和 "\r\n" （回车+换行
    ↪ ），均表示行尾。
terminator = ""
#### CSV 文件是否包含表头。
#### 如果为 true，首行将会被跳过，且基于首行映射目标表的列。
header = true
#### CSV 是否包含 NULL。
#### 如果为 true，CSV 文件的任何列都不能解析为 NULL。
not-null = false
#### 如果 `not-null` 为 false（即 CSV 可以包含 NULL），
#### 为以下值的字段将会被解析为 NULL。
null = '\N'
#### 是否解析字段内的反斜线转义符。
backslash-escape = true
#### 是否移除以分隔符结束的行。
trim-last-separator = false
```

对于诸如 `separator`，`delimiter` 和 `terminator` 等取值为字符串的配置项，如果需要设置的字符串中包含特殊字符，可以通过使用反斜杠 `\` 转义的方式进行输入，输入的转义序列必须被包含在一对双引号 `"` 之间。例如，设置 `separator = "\u001f"` 表示使用 ASCII 字符 0X1F 作为字符串定界符。

你也可以使用单引号字符串 `'...'` 禁止对字符进行转义。

另外，设置 `separator = '\n'` 表示使用两个字符 `\` + n 作为字符串定界符，而不是转义后的换行符 `\n`。

更多详细的内容请参考 TOML v1.0.0 标准。

`separator`

- 指定字段分隔符。
- 可以为一个或多个字符，不能为空。
- 常用值：
  - CSV 用 `','`
  - TSV 用 `"\t"`
  - “ð0001” 表示使用 ASCII 字符 0x01
- 对应 LOAD DATA 语句中的 `FIELDS TERMINATED BY` 项。

`delimiter`

- 指定引用定界符。

- 如果 `delimiter` 为空，所有字段都会被取消引用。
- 常用值：
  - `'"'` 使用双引号引用字段，和 [RFC 4180] 一致。
  - `''` 不引用
- 对应 LOAD DATA 语句中的 `FIELDS ENCLOSED BY` 项。

参考 RFC 4180。

`terminator`

- 指定行尾定界符。
- 如果 `terminator` 为空，则 “\n” （换行）和 “\r\n” （回车 + 换行）均表示行尾。
- 对应 LOAD DATA 语句中的 `LINES TERMINATED BY` 项。

`header`

- 是否所有 CSV 文件都包含表头行。
- 如为 true，第一行会被用作列名。如为 false，第一行并无特殊性，按普通的数据行处理。

`not-null` 和 `null`

- `not-null` 决定是否所有字段不能为空。
- 如果 `not-null` 为 false，设定了 `null` 的字符串会被转换为 SQL NULL 而非具体数值。
- 引用不影响字段是否为空。

  例如有如下 CSV 文件：

```
A,B,C
\N,"\N",
```

  在默认设置（`not-null = false; null = '\N'`）下，列 A and B 导入 TiDB 后都将会转换为 NULL。列 C 是空字符串 `''`，但并不会解析为 NULL。

`backslash-escape`

- 是否解析字段内的反斜线转义符。
- 如果 `backslash-escape` 为 true，下列转义符会被识别并转换。

转义符	转换为
\0	空字符 (U+0000)
\b	退格 (U+0008)
\n	换行 (U+000A)
\r	回车 (U+000D)
\t	制表符 (U+0009)
\Z	Windows EOF (U+001A)

其他情况下（如 `\"`）反斜线会被移除，仅在字段中保留其后面的字符（`"`），这种情况下，保留的字符仅作为普通字符，特殊功能（如界定符）都会失效。

- 引用不会影响反斜线转义符的解析与否。
- 对应 LOAD DATA 语句中的 `FIELDS ESCAPED BY '\'` 项。

`trim-last-separator`

- 将 `separator` 字段当作终止符，并移除尾部所有分隔符。

  例如有如下 CSV 文件：

  ```
  A,,B,,
  ```

- 当 `trim-last-separator = false`，该文件会被解析为包含 5 个字段的行 `('A', '', 'B', '', '')`。
- 当 `trim-last-separator = true`，该文件会被解析为包含 3 个字段的行 `('A', '', 'B')`。
- 此配置项已被弃用，建议使用兼容性更好的 `terminator`。

  如果有如下旧的配置：

  ```
  separator = ','
  trim-last-separator = true
  ```

  建议修改为：

  ```
  separator = ','
  terminator = ",\n" # 请根据文件实际使用的换行符指定为 ",\n" 或 ",\r\n"
  ```

不可配置项

TiDB Lightning 并不完全支持 `LOAD DATA` 语句中的所有配置项。例如：

- 不可使用行前缀 (`LINES STARTING BY`)。
- 不可跳过表头 (`IGNORE n LINES`)。如有表头，必须是有效的列名。

启用严格格式

导入文件的大小统一约为 256 MB 时，TiDB Lightning 可达到最佳工作状态。如果导入单个 CSV 大文件，TiDB Lightning 只能使用一个线程来处理，这会降低导入速度。

要解决此问题，可先将 CSV 文件分割为多个文件。对于通用格式的 CSV 文件，在没有读取整个文件的情况下无法快速确定行的开始和结束位置。因此，默认情况下 TiDB Lightning 不会自动分割 CSV 文件。但如果你确定待导入的 CSV 文件符合特定的限制要求，则可以启用 `strict-format` 设置。启用后，TiDB Lightning 会将单个 CSV 大文件分割为单个大小为 256 MB 的多个文件块进行并行处理。

```
[mydumper]
strict-format = true
```

严格格式的 CSV 文件中，每个字段仅占一行，即必须满足以下条件之一：

- delimiter 为空；
- 每个字段不包含 `terminator` 对应的字符串。在默认配置下，对应每个字段不包含 CR（`\r`）或 LF（`\n`）。

如果 CSV 文件不是严格格式但 `strict-format` 被误设为 `true`，跨多行的单个完整字段会被分割成两部分，导致解析失败，甚至不报错地导入已损坏的数据。

常见配置示例

CSV

默认设置已按照 RFC 4180 调整。

```
[mydumper.csv]
separator = ',' # 如果数据中可能有逗号，建议源文件导出时分隔符使用非常见组合字符例如'|+|'
delimiter = '"'
header = true
not-null = false
null = '\N'
backslash-escape = true
```

示例内容：

```
ID,Region,Count
1,"East",32
2,"South",\N
3,"West",10
4,"North",39
```

TSV

```
[mydumper.csv]
separator = "\t"
delimiter = ''
header = true
not-null = false
null = 'NULL'
backslash-escape = false
```

示例内容：

```
ID    Region    Count
1     East      32
2     South     NULL
3     West      10
4     North     39
```

TPC-H DBGEN

```
[mydumper.csv]
separator = '|'
delimiter = ''
terminator = "|\n"
header = false
not-null = true
backslash-escape = false
```

示例内容：

```
1|East|32|
2|South|0|
3|West|10|
4|North|39|
```

13.7.5.1.2 SQL

TiDB Lightning 在处理 SQL 文件时，由于无法对单个文件进行快速分割，因此无法通过增加并发提高单个文件的导入速度。鉴于此，导出数据为 SQL 文件时应尽量避免单个 SQL 文件过大，通常单文件在 256MiB 左右可以达到最佳性能。

13.7.5.1.3 Parquet

TiDB Lightning 目前仅支持由 Amazon Aurora 或者 Hive 导出快照生成的 Parquet 文件。要识别其在 S3 的文件组织形式，需要使用如下配置匹配到所有的数据文件：

```
[[mydumper.files]]
#### 解析 AWS Aurora parquet 文件所需的表达式
pattern = '(?i)^(?:[^/]*/)*([a-z0-9_]+)\.([a-z0-9_]+)/(?:[^/]*/)*(?:[a-z0-9\-_.]+\.(parquet))$'
schema = '$1'
table = '$2'
type = '$3'
```

注意，此处仅说明 Aurora snapshot 导出的 parquet 文件如何匹配。Schema 文件需要单独导出及处理。

关于 `mydumper.files`，请参考自定义文件匹配。

13.7.5.1.4 压缩文件

TiDB Lightning 目前支持由 Dumpling 导出的压缩文件或满足符合上文命名规则的压缩文件，目前支持 `gzip`、`snappy`、`zstd` 压缩算法的压缩文件。在文件名符合命名规则时，TiDB Lightning 会自动识别压缩算法在流式解压后导入，无需额外配置。

> **注意**

- 由于 TiDB Lightning 无法对单个大压缩文件进行并发解压，因此压缩文件的大小会直接影响导入速度。建议压缩数据文件解压后的源文件大小不超过 256 MiB。
- TiDB Lightning 仅支持导入各自独立压缩的数据文件，不支持导入多个数据文件组成的单个压缩文件集合包。
- TiDB Lightning 不支持二次压缩的 `parquet` 文件，例如 `db.table.parquet.snappy`。如需压缩 `parquet` 文件，你可以配置 `parquet` 文件数据存储的压缩格式。
- TiDB v6.4.0 及之后版本的 TiDB Lightning 支持后缀为压缩算法 `gzip`、`snappy`、`zstd`，以及后缀名 `.bak` 的数据文件。其他后缀名会报错。你需要提前修改文件名，或将该类文件移出导入数据目录来避免此类错误。

13.7.5.1.5 自定义文件匹配

TiDB Lightning 仅识别符合命名要求的数据文件，但在某些情况下已提供的数据文件并不符合要求，因此可能出现 TiDB Lightning 在极短的时间结束，处理文件数量为 0 的情况。

为了解决此类问题，TiDB Lightning 提供了 `[[mydumper.files]]` 配置用于通过自定义表达式匹配数据文件。

以 AWS Aurora 导出至 S3 的快照文件为例，Parquet 文件的完整路径为：`S3://some-bucket/some-subdir/some-database/some-database.some-table/part-00000-c5a881bb-58ff-4ee6-1111-b41ecff340a3-c000.gz.parquet`。

通常 `data-source-dir` 会被配置为`S3://some-bucket/some-subdir/some-database/` 以导入 `some-database` 库。

根据上述 Parquet 文件的路径，你可以编写正则表达式 `(?i)^(?:[^/]*/)*([a-z0-9_]+)\.([a-z0-9_]+)/(?:[^/]*/)*(?:[a-z0-9\-_.]+\.(parquet))$`，得到的 match group 中 index=1 的内容为 `some-database`，index=2 的内容为 `some-table`，index=3 的内容为 `parquet`。

根据上述正则表达式及相应的 index 编写配置文件，TiDB Lightning 即可识别非默认命名规则的文件，最终实际配置如下：

```
[[mydumper.files]]
#### 解析 AWS Aurora parquet 文件所需的表达式
pattern = '(?i)^(?:[^/]*/)*([a-z0-9_]+)\.([a-z0-9_]+)/(?:[^/]*/)*(?:[a-z0-9\-_.]+\.(parquet))$'
schema = '$1'
table = '$2'
type = '$3'
```

- schema：目标库名称，值可以为：
  - 正则表达式匹配到的 group 序号，例如 “$1”。
  - 直接填写期望导入的库名，例如 “db1”。所有匹配到的文件均会导入 “db1”。
- table：目标表名称，值可以为：
  - 正则表达式匹配到的 group 序号，例如 “$2”。
  - 直接填写期望导入的库名，例如 “table1”。所有匹配到的文件均会导入 “table1”。
- type：文件类型，支持`sql`，`parquet`，`csv`，值可以为：

  - 正则表达式匹配到的 group 序号，例如 “$3”。

- key：文件的序号，即前文所述${db_name}.${table_name}.001.csv中的001。
  - 正则表达式匹配到的 group 序号，例如 “$4”。

13.7.5.1.6 从 Amazon S3 导入数据

如下为从 Amazon S3 导入数据的示例，更多配置参数描述，可参考外部存储 URI 格式。

- 使用本地已设置的权限访问 S3：

```
./tidb-lightning --tidb-port=4000 --pd-urls=127.0.0.1:2379 --backend=local --sorted-kv-dir=/
    ↪ tmp/sorted-kvs \
    -d 's3://my-bucket/sql-backup'
```

- 使用路径类型的请求模式：

```
./tidb-lightning --tidb-port=4000 --pd-urls=127.0.0.1:2379 --backend=local --sorted-kv-dir=/
    ↪ tmp/sorted-kvs \
    -d 's3://my-bucket/sql-backup?force-path-style=true&endpoint=http://10.154.10.132:8088'
```

- 使用 AWS IAM 角色的 ARN 来访问 S3 数据：

```
./tidb-lightning --tidb-port=4000 --pd-urls=127.0.0.1:2379 --backend=local --sorted-kv-dir=/
    ↪ tmp/sorted-kvs \
    -d 's3://my-bucket/test-data?role-arn=arn:aws:iam::888888888888:role/my-role'
```

- 使用 AWS IAM 用户密钥来访问 S3 数据：

```
./tidb-lightning --tidb-port=4000 --pd-urls=127.0.0.1:2379 --backend=local --sorted-kv-dir=/
    ↪ tmp/sorted-kvs \
    -d 's3://my-bucket/test-data?access_key={my_access_key}&secret_access_key={
        ↪ my_secret_access_key}'
```

- 使用 AWS IAM 角色的密钥以及会话令牌来访问 S3 数据：

```
./tidb-lightning --tidb-port=4000 --pd-urls=127.0.0.1:2379 --backend=local --sorted-kv-dir=/
    ↪ tmp/sorted-kvs \
    -d 's3://my-bucket/test-data?access_key={my_access_key}&secret_access_key={
        ↪ my_secret_access_key}&session-token={my_session_token}'
```

13.7.5.1.7 更多

- 使用 Dumpling 导出到 CSV 文件
- LOAD DATA

13.7.6 Physical Import Mode

13.7.6.1 Physical Import Mode 简介

Physical Import Mode 是 TiDB Lightning 支持的一种数据导入方式。Physical Import Mode 不经过 SQL 接口，而是直接将数据以键值对的形式插入 TiKV 节点，是一种高效、快速的导入模式。使用 Physical Import Mode 时，单个 Lightning 实例可导入的数据量为 10 TiB，理论上导入的数据量可以随着 Lightning 实例数量的增加而增加，目前已经有多个用户验证基于并行导入功能可以导入的数据量达 20 TiB。

使用前请务必自行阅读必要条件及限制。

Physical Import Mode 对应的后端模式为 `local`。

13.7.6.1.1 原理说明

1. 在导入数据之前，`tidb-lightning` 会自动将 TiKV 节点切换为“导入模式” (import mode)，优化写入效率并停止自动压缩。`tidb-lightning` 会根据 TiDB 集群的版本决定是否停止全局调度。

    - 当 TiDB 集群版本 >= v6.1.0 且 TiDB Lightning 版本 >= v6.2.0 时，`tidb-lightning` 在向 TiKV 导入数据时，只会暂停目标表数据范围所在 region 的调度，并在目标表导入完成后恢复调度。
    - 当 TiDB 集群版本 < v6.1.0 或 TiDB Lightning 版本 < v6.2.0 时，`tidb-lightning` 会暂停全局调度。

2. `tidb-lightning` 在目标数据库建立表结构，并获取其元数据。

    如果将 `add-index-by-sql` 设置为 `true`，`tidb-lightning` 会使用 SQL 接口添加索引，并且会在导入数据前移除目标表的所有次级索引。

3. 每张表都会被分割为多个连续的区块，这样来自大表 (200 GB+) 的数据就可以多个并发导入。

4. `tidb-lightning` 会为每一个区块准备一个“引擎文件 (engine file)”来处理键值对。`tidb-lightning` 会并发读取 SQL dump，将数据源转换成与 TiDB 相同编码的键值对，然后将这些键值对排序写入本地临时存储文件中。

5. 当一个引擎文件数据写入完毕时，`tidb-lightning` 便开始对目标 TiKV 集群数据进行分裂和调度，然后导入数据到 TiKV 集群。

    引擎文件包含两种：数据引擎与索引引擎，各自又对应两种键值对：行数据和次级索引。通常行数据在数据源里是完全有序的，而次级索引是无序的。因此，数据引擎文件在对应区块写入完成后会被立即上传，而所有的索引引擎文件只有在整张表所有区块编码完成后才会执行导入。

    注意当 `tidb-lightning` 使用 SQL 接口添加索引时 (即 `add-index-by-sql` 设置为 `true`)，索引引擎将不会写入数据，因为此时目标表的次级索引已经在第 2 步中被移除。

6. 整张表相关联的所有引擎文件完成导入后，`tidb-lightning` 会对比本地数据源及下游集群的校验和 (checksum)，确保导入的数据无损，然后添加在第 2 步中被移除的次级索引或让 TiDB 分析 (ANALYZE) 这些新增的数据，以优化日后的操作。同时，`tidb-lightning` 调整 AUTO_INCREMENT 值防止之后新增数据时发生冲突。

    表的自增 ID 是通过行数的上界估计值得到的，与表的数据文件总大小成正比。因此，最后的自增 ID 通常比实际行数大得多。这属于正常现象，因为在 TiDB 中自增 ID 不一定是连续分配的。

7. 在所有步骤完毕后，`tidb-lightning` 自动将 TiKV 切换回“普通模式” (normal mode)，并恢复可能被暂停的全局调度，此后 TiDB 集群可以正常对外提供服务。

13.7.6.1.2 必要条件及限制

运行环境需求

操作系统：建议使用新的、纯净版 CentOS 7 实例，你可以在本地虚拟化一台主机，或在供应商提供的平台上部署一台小型的云虚拟主机。TiDB Lightning 运行过程中，默认会占满 CPU，建议单独部署在一台主机上。如果条件不允许，你可以将 TiDB Lightning 和其他组件（比如 `tikv-server`）部署在同一台机器上，然后设置 `region-concurrency` 配置项的值为逻辑 CPU 数的 75%，以限制 TiDB Lightning 对 CPU 资源的使用。

内存和 CPU：

建议使用 32 核以上的 CPU 和 64 GiB 以上内存以获得更好的性能。

> 注意：
>
> 导入大量数据时，一个并发对内存的占用在 2 GiB 左右，也就是说总内存占用最大可达到 region-concurrency * 2 GiB。`region-concurrency` 默认与逻辑 CPU 的数量相同。如果内存的大小（GiB）小于逻辑 CPU 数量的两倍或运行时出现 OOM，需要手动调低 `region-concurrency` 参数以避免 TiDB Lightning OOM。

存储空间：配置项 `sorted-kv-dir` 设置排序的键值对的临时存放地址，目标路径必须是一个空目录，目录空间须大于待导入数据集的大小。建议与 `data-source-dir` 使用不同的存储设备，独占 IO 会获得更好的导入性能，且建议优先考虑配置闪存等高性能存储介质。

网络：建议使用 10 Gbps 以太网卡。

版本要求

- TiDB Lightning 版本 ≥ 4.0.3。
- TiDB 集群版本 ≥ v4.0.0。
- 如果目标 TiDB 集群是 v3.x 或以下的版本，需要使用 Importer-backend 来完成数据的导入。在这个模式下，`tidb-lightning` 需要将解析的键值对通过 gRPC 发送给 `tikv-importer` 并由 `tikv-importer` 完成数据的导入。

使用限制

- 请勿直接使用 Physical Import Mode 向已经投入生产的 TiDB 集群导入数据，这将对在线业务产生严重影响。如需向生产集群导入数据，请参考导入时限制调度范围从集群降低到表级别。
- 如果你的 TiDB 生产集群上有延迟敏感型业务，并且并发较小，不建议使用 TiDB Lightning 物理导入模式导入数据到该集群，因为可能会影响在线业务。
- 默认情况下，不应同时启动多个 TiDB Lightning 实例向同一 TiDB 集群导入数据，而应考虑使用并行导入特性。
- 使用多个 TiDB Lightning 向同一目标导入时，请勿混用不同的 backend，即不可同时使用 Physical Import Mode 和 Logical Import Mode 导入同一 TiDB 集群。

- 在导入数据的过程中，请勿在目标表进行 DDL 和 DML 操作，否则会导致导入失败或数据不一致。导入期间也不建议进行读操作，因为读取的数据可能不一致。请在导入操作完成后再进行读写操作。
- 单个 TiDB Lightning 进程导入单表不应超过 10 TB。使用并行导入时，TiDB Lightning 实例不应超过 10 个。

与其他组件一同使用的注意事项

- TiDB Lightning 与 TiFlash 一起使用时需要注意：
  - 无论是否已为一张表创建 TiFlash 副本，你都可以使用 TiDB Lightning 导入数据至该表。但该场景下，TiDB Lightning 导入数据耗费的时间更长，具体取决于 TiDB Lightning 部署机器的网卡带宽、TiFlash 节点的 CPU 及磁盘负载及 TiFlash 副本数等因素。
- TiDB Lightning 字符集相关的注意事项：
  - TiDB Lightning 在 v5.4.0 之前不支持导入 `charset=GBK` 的表。
- TiDB Lightning 与 TiCDC 一起使用时需要注意：
  - TiCDC 无法捕获 Physical Import Mode 插入的数据。

13.7.6.2 使用 Physical Import Mode

本文档介绍如何编写Physical Import Mode 的配置文件，如何进行性能调优、使用磁盘资源配额等内容。

使用物理导入模式有一些限制，使用前请务必阅读必要条件及限制。

13.7.6.2.1 配置及使用

可以通过以下配置文件使用 Physical Import Mode 执行数据导入：

```
[lightning]
#### 日志
level = "info"
file = "tidb-lightning.log"
max-size = 128 # MB
max-days = 28
max-backups = 14

#### 启动之前检查集群是否满足最低需求。
check-requirements = true

[mydumper]
#### 本地源数据目录或外部存储 URI。关于外部存储 URI 详情可参考 https://docs.pingcap.com/zh/tidb/
    ↪ v6.1/backup-and-restore-storages#uri-%E6%A0%BC%E5%BC%8F。
data-source-dir = "/data/my_database"

[tikv-importer]
#### 导入模式配置，设为 local 即使用 Physical Import Mode
```

```
backend = "local"

#### 冲突数据处理方式
duplicate-resolution = 'remove'

#### 本地进行 KV 排序的路径。
sorted-kv-dir = "./some-dir"

#### 限制 TiDB Lightning 向每个 TiKV 节点写入的带宽大小，默认为 0，表示不限制。
#### store-write-bwlimit = "128MiB"

#### Physical Import Mode 是否通过 SQL 方式添加索引。默认为 `false`，表示 TiDB Lightning
    ↪ 会将行数据以及索引数据都编码成 KV pairs 后一同导入 TiKV，实现机制和历史版本保持一致。
    ↪ 如果设置为 `true`，即 TiDB Lightning 会导完数据后，再使用 add index 的 SQL 来添加索引。
#### 通过 SQL 方式添加索引的优点是将导入数据与导入索引分开，可以快速导入数据，即使导入数据后，
    ↪ 索引添加失败，也不会影响数据的一致性。
#### add-index-by-sql = false

[tidb]
#### 目标集群的信息。tidb-server 的地址，填一个即可。
host = "172.16.31.1"
port = 4000
user = "root"
#### 设置连接 TiDB 的密码，可为明文或 Base64 编码。
password = ""
#### 必须配置。表结构信息从 TiDB 的“status-port”获取。
status-port = 10080
#### 必须配置。pd-server 的地址，填一个即可。
pd-addr = "172.16.31.4:2379"
#### tidb-lightning 引用了 TiDB 库，并生成产生一些日志。
#### 设置 TiDB 库的日志等级。
log-level = "error"

[post-restore]
#### 配置是否在导入完成后对每一个表执行 `ADMIN CHECKSUM TABLE <table>` 操作来验证数据的完整性。
#### 可选的配置项：
#### - "required"（默认）。在导入完成后执行 CHECKSUM 检查，如果 CHECKSUM 检查失败，则会报错退出。
#### - "optional"。在导入完成后执行 CHECKSUM 检查，如果报错，会输出一条 WARN 日志并忽略错误。
#### - "off"。导入结束后不执行 CHECKSUM 检查。
#### 默认值为 "required"。从 v4.0.8 开始，checksum 的默认值由此前的 "true" 改为 "required"。
#### # 注意：
#### 1. Checksum 对比失败通常表示导入异常（数据丢失或数据不一致），因此建议总是开启 Checksum。
#### 2. 考虑到与旧版本的兼容性，依然可以在本配置项设置 `true` 和 `false` 两个布尔值，其效果与 `
    ↪ required` 和 `off` 相同。
checksum = "required"
```

```
#### 配置是否在 CHECKSUM 结束后对所有表逐个执行 `ANALYZE TABLE <table>` 操作。
#### 此配置的可选配置项与 `checksum` 相同，但默认值为 "optional"。
analyze = "optional"
```

Lightning 的完整配置文件可参考完整配置及命令行参数。

13.7.6.2.2 冲突数据检测

冲突数据，即两条或两条以上的记录存在 PK/UK 列数据重复的情况。当数据源中的记录存在冲突数据，将导致该表真实总行数和使用唯一索引查询的总行数不一致的情况。冲突数据检测支持三种策略：

- remove：推荐方式。记录并删除所有的冲突记录，以确保目的 TiDB 中的数据状态保持一致。
- none：关闭冲突数据检测。该模式是两种模式中性能最佳的，但是可能会导致目的 TiDB 中出现数据不一致的情况。

在 v5.3 之前，TiDB Lightning 不具备冲突数据检测特性，若存在冲突数据将导致导入过程最后的 Checksum 环节失败。开启冲突检测特性的情况下，只要检测到冲突数据，TiDB Lightning 都会跳过最后的 Checksum 环节（因为必定失败）。

假设一张表 order_line 的表结构如下：

```
CREATE TABLE IF NOT EXISTS `order_line` (
  `ol_o_id` int(11) NOT NULL,
  `ol_d_id` int(11) NOT NULL,
  `ol_w_id` int(11) NOT NULL,
  `ol_number` int(11) NOT NULL,
  `ol_i_id` int(11) NOT NULL,
  `ol_supply_w_id` int(11) DEFAULT NULL,
  `ol_delivery_d` datetime DEFAULT NULL,
  `ol_quantity` int(11) DEFAULT NULL,
  `ol_amount` decimal(6,2) DEFAULT NULL,
  `ol_dist_info` char(24) DEFAULT NULL,
  PRIMARY KEY (`ol_w_id`,`ol_d_id`,`ol_o_id`,`ol_number`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin;
```

若在导入过程中检测到冲突数据，则可以查询 lightning_task_info.conflict_error_v1 表得到以下内容：

```
mysql> select table_name,index_name,key_data,row_data from conflict_error_v1 limit 10;
+--
    ↪ ------------------+------------+----------+-----------------------------------------------------------------
    ↪
|  table_name         | index_name | key_data | row_data
    ↪                                                                                 |
+--
    ↪ ------------------+------------+----------+-----------------------------------------------------------------
    ↪
```

```
| `tpcc`.`order_line` | PRIMARY    | 21829216 | (2677, 10, 10, 11, 75656, 10, NULL, 5, 5831.97, "
    ↪ HT5DN3EVb6kWTd4L37bsbogj") |
| `tpcc`.`order_line` | PRIMARY    | 49931672 | (2677, 10, 10, 11, 75656, 10, NULL, 5, 5831.97, "
    ↪ HT5DN3EVb6kWTd4L37bsbogj") |
| `tpcc`.`order_line` | PRIMARY    | 21829217 | (2677, 10, 10, 12, 76007, 10, NULL, 5, 9644.36, "
    ↪ bHuVoRfidQOq2rJ6ZC9Hd12E") |
| `tpcc`.`order_line` | PRIMARY    | 49931673 | (2677, 10, 10, 12, 76007, 10, NULL, 5, 9644.36, "
    ↪ bHuVoRfidQOq2rJ6ZC9Hd12E") |
| `tpcc`.`order_line` | PRIMARY    | 21829218 | (2677, 10, 10, 13, 85618, 10, NULL, 5, 7427.98, "
    ↪ t3rsesgi9rVAKi9tf6an5Rpv") |
| `tpcc`.`order_line` | PRIMARY    | 49931674 | (2677, 10, 10, 13, 85618, 10, NULL, 5, 7427.98, "
    ↪ t3rsesgi9rVAKi9tf6an5Rpv") |
| `tpcc`.`order_line` | PRIMARY    | 21829219 | (2677, 10, 10, 14, 15873, 10, NULL, 5, 133.21, "
    ↪ z1vH0e31tQydJGhfNYNa4ScD")  |
| `tpcc`.`order_line` | PRIMARY    | 49931675 | (2677, 10, 10, 14, 15873, 10, NULL, 5, 133.21, "
    ↪ z1vH0e31tQydJGhfNYNa4ScD")  |
| `tpcc`.`order_line` | PRIMARY    | 21829220 | (2678, 10, 10, 1, 44644, 10, NULL, 5, 8463.76, "
    ↪ TWKJBt5iJA4eF7FIVxnugNmz")  |
| `tpcc`.`order_line` | PRIMARY    | 49931676 | (2678, 10, 10, 1, 44644, 10, NULL, 5, 8463.76, "
    ↪ TWKJBt5iJA4eF7FIVxnugNmz")  |
+--
    ↪ ------------------+------------+-------------------------------------------------------------------------
    ↪
10 rows in set (0.14 sec)
```

根据上述信息人工甄别需要保留的重复数据，手动插回原表即可。

13.7.6.2.3 导入时限制调度范围从集群降低到表级别

自 TiDB Lightning v6.2.0 版本起，TiDB Lightning 提供机制控制导入数据过程对在线业务的影响。TiDB Lightning 不会暂停全局的调度，而是只暂停目标表数据范围所在 region 的调度，降低了对在线业务的影响。

> **注意：**
>
> TiDB Lightning 不支持导入数据到已有业务写入的数据表。
>
> TiDB 集群版本需大于等于 v6.1.0，更低的版本 TiDB Lightning 会保持原有行为，暂停全局调度，数据导入期间会给在线业务带来严重影响。

TiDB Lightning 默认情况下会在最小范围内暂停集群调度，无需额外配置。但默认配置下，TiDB 集群仍然会因为数据导入太快，使在线业务的性能受到影响，所以你需要额外配置几个选项来控制导入速度和其他可能影响集群性能的因素：

```
[tikv-importer]
#### 限制 TiDB Lightning 向每个 TiKV 节点写入的带宽大小。
```

```
store-write-bwlimit = "128MiB"

[tidb]
#### 使用更小的并发以降低计算 checksum 和执行 analyze 对事务延迟的影响。
distsql-scan-concurrency = 3
```

13.7.6.2.4 性能调优

提高 Lightning Physical Import Mode 导入性能最直接有效的方法：

- 升级 Lightning 所在节点的硬件，尤其重要的是 CPU 和 sorted-key-dir 所在存储设备的性能。
- 使用并行导入特性实现水平扩展。

当然，Lightning 也提供了部分并发相关配置以影响 Physical Import Mode 的导入性能。但是从长期实践的经验总结来看，以下四个配置项一般保持默认值即可，调整其数值并不会带来显著的性能提升，可作为了解内容阅读。

```
[lightning]
#### 引擎文件的最大并行数。
#### 每张表被切分成一个用于存储索引的“索引引擎”和若干存储行数据的“数据引擎”。
#### 这两项设置控制两种引擎文件的最大并发数。
index-concurrency = 2
table-concurrency = 6

#### 数据的并发数。默认与逻辑 CPU 的数量相同。
#### region-concurrency =

#### I/O 最大并发数。I/O 并发量太高时，会因硬盘内部缓存频繁被刷新
#### 而增加 I/O 等待时间，导致缓存未命中和读取速度降低。
#### 对于不同的存储介质，此参数可能需要调整以达到最佳效率。
io-concurrency = 5
```

导入时，每张表被切分成一个用于存储索引的“索引引擎”和若干存储行数据的“数据引擎”，index-concurrency用于调整“索引引擎”的并发度。

在调整 index-concurrency 时，需要注意 index-concurrency * 每个表对应的源文件数量 > region-concurrency 以确保 cpu 被充分利用，一般比例大概在 1.5 ~ 2 左右为优。index-concurrency 不应该设置的过大，但不低于 2 (默认)，过大会导致太多导入的 pipeline 变差，大量 index-engine 的 import 阶段堆积。

table-concurrency 同理，需要确保table-concurrency * 每个表对应的源文件数量 > region-concurrency 以确保 cpu 被充分利用。推荐值为region-concurrency * 4 / 每个表对应的源文件数量 左右，最少设置为 4.

如果表非常大，Lightning 会按照 100 GiB 的大小将表分割成多个批次处理，并发度由 table-concurrency 控制。

上述两个参数对导入速度影响不大，使用默认值即可。

io-concurrency 用于控制文件读取并发度，默认值为 5。可以认为在某个时刻只有 5 个句柄在执行读操作。由于文件读取速度一般不会是瓶颈，所以使用默认值即可。

读取文件数据后，lightning 还需要做后续处理，例如将数据在本地进行编码和排序。此类操作的并发度由 `region-concurrency` 配置控制。`region-concurrency` 的默认值为 CPU 核数，通常无需调整，建议不要将 Lightning 与其它组件部署在同一主机，如果客观条件限制必须混合部署，则需要根据实际负载调低 `region-concurrency`。

此外，TiKV 的num-threads 配置也可能影响性能，新集群建议设置为 CPU 核数。

13.7.6.2.5 磁盘资源配额从 v6.2.0 版本开始引入

TiDB Lightning 在使用物理模式导入数据时，会在本地磁盘创建大量的临时文件，用来对原始数据进行编码、排序、分割。当用户本地磁盘空间不足时，TiDB Lightning 会由于写入文件失败而报错退出。

为了减少这种情况的发生，你可以为 TiDB Lightning 配置磁盘配额 (disk quota)。当磁盘配额不足时，TiDB Lightning 会暂停读取源数据以及写入临时文件的过程，优先将已经完成排序的 key-value 写入到 TiKV，TiDB Lightning 删除本地临时文件后，再继续导入过程。

为 TiDB Lightning 开启磁盘配额，你需要在配置文件中加入配置项：

```
[tikv-importer]
#### 默认为 MaxInt64，即 9223372036854775807 字节
disk-quota = "10GB"
backend = "local"

[cron]
#### 检查磁盘配额的时间间隔，默认为 60 秒。
check-disk-quota = "30s"
```

`disk-quota` 配置项可以设置磁盘配额，默认为 MaxInt64，即 9223372036854775807 字节，这个值远远大于实际情况中可能需要的磁盘空间，相当于没开启。

`check-disk-quota` 配置项是检查磁盘配额的时间间隔，默认为 60 秒。由于检查临时文件使用空间的过程需要加锁，会使所有的导入线程都暂停，如果在每次写入之前都检查一次磁盘空间的使用情况，则会大大降低写入文件的效率（相当于单线程写入）。为了维持高效的写入，磁盘配额不会在每次写入之前检查，而是每隔一段时间暂停所有线程的写入并检查当前磁盘空间的使用情况。也就是说，当 `check-disk-quota` 配置项设置为一个非常大的值时，磁盘的使用空间有可能会大大超出磁盘配额，这样的情况下，磁盘配额功能可以说是不生效的。因此，`check-disk-quota` 的值建议不要设置太大，而具体设置多少则需要由 TiDB Lightning 具体运行的环境决定，因为不同的环境下，TiDB Lightning 写入临时文件的速度是不一样的。理论上来说，写入临时文件的速度越快，`check-disk-quota` 需要设置得越小。

13.7.7 Logical Import Mode

13.7.7.1 Logical Import Mode 简介

Logical Import Mode 是 TiDB Lightning 支持的一种数据导入方式。在 Logical Import Mode 下，TiDB Lightning 先将数据编码成 SQL，然后直接运行这些 SQL 语句进行数据导入。对于已有数据、对外提供服务的 TiDB 集群，推荐使用 Logical Import Mode 导入数据。Logical Import Mode 的行为与正常执行 SQL 并无差异，可保证 ACID。

Logical Import Mode 对应的后端模式为 `tidb`。

13.7.7.1.1 必要条件

操作系统：

建议使用新的、纯净版 CentOS 7 实例，你可以在本地虚拟化一台主机，或在供应商提供的平台上部署一台小型的云虚拟主机。TiDB Lightning 运行过程中，默认会占满 CPU，建议单独部署在一台主机上。如果条件不允许，你可以将 TiDB Lightning 和其他组件（比如 `tikv-server`）部署在同一台机器上，然后设置 `region-concurrency` 配置项的值为逻辑 CPU 数的 75%，以限制 TiDB Lightning 对 CPU 资源的使用。

内存和 CPU：

建议使用 4 核以上的 CPU 和 8 GiB 以上内存以获得更好的性能。根据长期的实践经验，Lightning 的 Logical Import Mode 没有显著（5 GiB 以上）的内存占用，但上调 `region-concurrency` 默认值将导致内存量增加。

网络：建议使用 1 Gbps 或 10 Gbps 以太网卡。

13.7.7.1.2 使用限制

使用多个 TiDB Lightning 向同一目标导入时，请勿混用不同的 backend，即不可同时使用 Physical Import Mode 和 Logical Import Mode 导入同一 TiDB 集群。

13.7.7.2 使用 Logical Import Mode

本文档介绍如何编写Logical Import Mode 的配置文件，如何进行性能调优等内容。

13.7.7.2.1 配置及使用

可以通过以下配置文件使用 Logical Import Mode 执行数据导入：

```
[lightning]
#### 日志
level = "info"
file = "tidb-lightning.log"
max-size = 128 # MB
max-days = 28
max-backups = 14

#### 启动之前检查集群是否满足最低需求。
check-requirements = true

[mydumper]
#### 本地源数据目录或外部存储 URI。关于外部存储 URI 详情可参考 https://docs.pingcap.com/zh/tidb/
    ↪ v6.6/backup-and-restore-storages#uri-%E6%A0%BC%E5%BC%8F。
data-source-dir = "/data/my_database"

[tikv-importer]
#### 导入模式配置，设为 tidb 即使用 Logical Import Mode
backend = "tidb"
```

```
#### Logical Import Mode 插入重复数据时执行的操作。
#### - replace：新数据替代已有数据
#### - ignore：保留已有数据，忽略新数据
#### - error：中止导入并报错
on-duplicate = "replace"

[tidb]
#### 目标集群的信息。tidb-server 的地址，填一个即可。
host = "172.16.31.1"
port = 4000
user = "root"
#### 设置连接 TiDB 的密码，可为明文或 Base64 编码。
password = ""
#### tidb-lightning 引用了 TiDB 库，并生成产生一些日志。
#### 设置 TiDB 库的日志等级。
log-level = "error"
```

TiDB Lightning 的完整配置文件可参考完整配置及命令行参数。

13.7.7.2.2 冲突数据检测

冲突数据，即两条或两条以上的记录存在主键或唯一键列数据重复的情况。当数据源中的记录存在冲突数据，将导致该表真实总行数和使用唯一索引查询的总行数不一致的情况。TiDB Lightning 的 Logical Import Mode 通过 `on-duplicate` 配置冲突数据检测的策略，TiDB Lightning 根据策略使用不同的 SQL 语句进行插入。

策略	冲突时默认行为	对应 SQL 语句
`replace`	新数据替代旧数据	`REPLACE INTO ...`
`ignore`	保留旧数据，忽略新数据	`INSERT IGNORE INTO ...`
`error`	中止导入	`INSERT INTO ...`

13.7.7.2.3 性能调优

- TiDB Lightning 的 Logical Import Mode 性能很大程度上取决于目标 TiDB 集群的写入性能，当遇到性能瓶颈时可参考 TiDB 相关性能优化文档。
- 如果发现目标 TiDB 集群的的写入尚未达到瓶颈，可以考虑增加 Lightning 配置中 `region-concurrency` 的值。`region-concurrency` 默认值为 CPU 核数，其含义在 Physical Import Mode 和 Logical Import Mode 下有所不同，Logical Import Mode 的 `region-concurrency` 表示写入并发数。配置示例：

```
[lightning]
region-concurrency = 32
```

- 调整目标 TiDB 集群的`raftstore.apply-pool-size` 和`raftstore.store-pool-size` 参数也可能提升导入速度。

13.7.8 TiDB Lightning 前置检查

从 TiDB 5.3.0 开始，TiDB Lightning 增加了前置检查功能，可以在执行迁移前检查配置。默认开启。该功能会自动进行一些磁盘空间和执行配置的常规检查，主要目的是确保后续的整个导入过程顺利。

下表介绍了各检查项和详细解释。

检查项	支持版本	解释
集群版本/状态是否正常	>= 5.3.0	检查配置中集群是否可以连接，Physical Import Mode 还会检查 TiKV/PD/TiFlash 版本是否支持。

检查项	支持版本	解释
是否有权限读取数据	>= 5.3.0	检查当从云存储（Amazon S3）读取数据的时候，是否有对应的权限，确保不会因权限缺失导致导入中断。

检查项	支持版本	解释
导入空间是否足够	>= 5.3.0	检查 TiKV 集群是否有足够空间导入数据。检查时会对数据源进行采样，通过采样结果预估索引大小占比。由于估算中考虑了索引，因此可能会出现尽管数据源大小低于本地盘可用空间，但依然无法通过检测的情况。
	1807	Physical Import Mode

检查项	支持版本	解释
Region 分布状态	>= 5.3.0	检查 TiKV 集群的 Region 分布是否均匀，以及是否存在大量空 region，如果空 Region 的数量大于 max(1000, 表的数量 * 3)，即大于“1000”和“3 倍表数量”二者中的较大者，TiDB Lightning 无法执行导入。

检查项	支持版本	解释
数据文件是否有大 CSV 文件	>= 5.3.0	当备份文件中出现大于 10 GiB 的 CSV 文件且无法进行自动切分 (Strict-For-mat=false) 的时候，会导致导入性能下降。该检查的目的是提醒用户确保数据格式的情况下，开启自动切分 CSV 功能。

检查项	支持版本	解释
是否可以从断点恢复	>= 5.3.0	该检查是确保断点恢复过程中，不会出现对源文件和数据库中 schema 进行修改，导致导入错误数据的情况。

检查项	支持版本	解释
是否可以导入数据到已存在的数据表中	>= 5.3.0	当导入到已创建好的数据表中时，该检查尽可能的检查此次导入的源文件是否和已存在的数据表匹配。检查列数是否匹配，如果源文件存在列名，则检查列名是否匹配。当源文件存在缺省列，则检查缺省列在数据表中是否存在
1811		

检查项	支持版本	解释
导入的目标表是否为空	>= 5.3.1	如果导入的目标表不为空，则 TiDB Lightning 会自动报错退出；如果开启了并行导入模式 (incremental-import = true)，则会跳过此检查项。

13.7.9 表库过滤

TiDB 数据迁移工具默认情况下作用于所有数据库，但实际使用中，往往只需要作用于其中的部分子集。例如，用户只想处理 `foo*` 和 `bar*` 形式的表，而无需对其他表进行操作。

从 TiDB 4.0 起，所有 TiDB 数据迁移工具都使用一个通用的过滤语法来定义子集。本文档介绍如何使用表库过滤功能。

13.7.9.1 使用表库过滤

13.7.9.1.1 命令行

在命令行中使用多个 `-f` 或 `--filter` 参数，即可在 TiDB 数据迁移工具中应用表库过滤规则。每个过滤规则均采用 `db.table` 形式，支持通配符（详情见下一节）。以下为各个工具中的使用示例：

- BR：

```
./br backup full -f 'foo*.*' -f 'bar*.*' -s 'local:///tmp/backup'
```

```
./br restore full -f 'foo*.*' -f 'bar*.*' -s 'local:///tmp/backup'
```

- Dumpling：

```
./dumpling -f 'foo*.*' -f 'bar*.*' -P 3306 -o /tmp/data/
```

- TiDB Lightning：

```
./tidb-lightning -f 'foo*.*' -f 'bar*.*' -d /tmp/data/ --backend tidb
```

13.7.9.1.2 TOML 配置文件

在 TOML 文件中，表库过滤规则以字符串数组的形式指定。以下为各个工具中的使用示例：

- TiDB Lightning：

```
[mydumper]
filter = ['foo*.*', 'bar*.*']
```

- TiCDC：

```
[filter]
rules = ['foo*.*', 'bar*.*']

[[sink.dispatchers]]
matcher = ['db1.*', 'db2.*', 'db3.*']
dispatcher = 'ts'
```

13.7.9.2 表库过滤语法

13.7.9.2.1 直接使用表名

每条表库过滤规则由“库”和“表”组成，两部分之间以英文句号 (.) 分隔。只有表名与规则完全相符的表才会被接受。

```
db1.tbl1
db2.tbl2
db3.tbl3
```

表名只由有效的标识符组成，例如：

- 数字（0 到 9）
- 字母（a 到 z，A 到 Z）

- $
- _
- 非 ASCII 字符（U+0080 到 U+10FFFF）

其他 ASCII 字符均为保留字。部分标点符号有特殊含义，详情见下一节。

13.7.9.2.2 使用通配符

表名的两个部分均支持使用通配符（详情见 fnmatch(3)）。

- `*`：匹配零个或多个字符。
- `?`：匹配一个字符。
- `[a-z]`：匹配“a”和“z”之间的一个字符。
- `[!a-z]`：匹配不在“a”和“z”之间的一个字符。

```
db[0-9].tbl[0-9a-f][0-9a-f]
data.*
*.backup_*
```

此处，“字符”指的是一个 Unicode 码位，例如：

- U+00E9 “é” 是 1 个字符。
- U+0065，U+0301 “é” 是 2 个字符。
- U+1F926 U+1F3FF U+200D U+2640 U+FE0F “🤦🏿‍♀️” 是 5 个字符。

13.7.9.2.3 使用文件导入

如需导入一个文件作为过滤规则，请在规则的开头加上一个“@”来指定文件名。库表过滤解析器将导入文件中的每一行都解析为一条额外的过滤规则。

例如，`config/filter.txt` 文件有以下内容：

```
employees.*
*.WorkOrder
```

以下两条表库过滤命令是等价的：

```
./dumpling -f '@config/filter.txt'
./dumpling -f 'employees.*' -f '*.WorkOrder'
```

导入的文件里不能使用过滤规则导入另一个文件。

13.7.9.2.4 注释与空行

导入的过滤规则文件中，每一行开头和结尾的空格都会被去除。此外，空行（空字符串）也将被忽略。

行首的 # 表示该行是注释，会被忽略。而不在行首的 # 则会被认为是语法错误。

```
### 这是一行注释
db.table   # 这一部分不是注释，且可能引起错误
```

13.7.9.2.5 排除规则

在一条过滤规则的开头加上 `!`，则表示符合这条规则的表不会被 TiDB 数据迁移工具处理。通过应用排除规则，库表过滤可以作为屏蔽名单来使用。

```
*.*
#^ 注意：必须先添加 *.* 规则来包括所有表
!*.Password
!employees.salaries
```

13.7.9.2.6 转义字符

如果需要将特殊字符转化为标识符，可以在特殊字符前加上反斜杠 `\`。

```
db\.with\.dots.*
```

为了简化语法并向上兼容，不支持下列字符序列：

- 在行尾去除空格后使用 `\`（使用 `[ ]` 来匹配行尾的空格）。
- 在 `\` 后使用数字或字母（`[0-9a-zA-Z]`）。特别是类似 C 的转义序列，如 `\0`、`\r`、`\n`、`\t` 等序列，目前在表库过滤规则中无意义。

13.7.9.2.7 引号包裹的标识符

除了 `\` 之外，还可以用 `"` 和 `` ` `` 来控制特殊字符。

```
"db.with.dots"."tbl\1"
`db.with.dots`.`tbl\2`
```

也可以通过输入两次引号，将引号包含在标识符内。

```
"foo""bar".`foo``bar`
### 等价于：
foo\"bar.foo\`bar
```

用引号包裹的标识符不可以跨越多行。

用引号只包裹标识符的一部分是无效的，例如：

```
"this is "invalid*.*
```

13.7.9.2.8 正则表达式

如果你需要使用较复杂的过滤规则，可以将每个匹配模型写为正则表达式，以 `/` 为分隔符：

```
/^db\d{2,}$/./^tbl\d{2,}$/
```

这类正则表示使用 Go dialect。只要标识符中有一个子字符串与正则表达式匹配，则视为匹配该模型。例如，`/b/` 匹配 db01。

> 注意：
>
> 正则表达式中的每一个 `/` 都需要转义为 `\/`，包括在 `[...]` 里面的 `/`。不允许在 `\Q...\E` 之间放置一个未转义的 `/`。

13.7.9.3 使用多个过滤规则

当表的名称与过滤列表中所有规则均不匹配时，默认情况下这些表被忽略。

要建立一个屏蔽名单，必须使用显式的 `*.*` 作为第一条过滤规则，否则所有表均被排除。

```
### 所有表均被过滤掉
./dumpling -f '!*.Password'

### 只有 “Password” 表被过滤掉，其余表仍保留
./dumpling -f '*.*' -f '!*.Password'
```

如果一个表的名称与过滤列表中的多个规则匹配，则以最后匹配的规则为准。例如：

```
### rule 1
employees.*
### rule 2
!*.dep*
### rule 3
*.departments
```

过滤结果如下：

表名	规则 1	规则 2	规则 3	结果
irrelevant.table				默认（拒绝）
employees.employees	√			规则 1（接受）
employees.dept_emp	√	√		规则 2（拒绝）
employees.departments	√	√	√	规则 3（接受）
else.departments		√	√	规则 3（接受）

> 注意：
>
> 在 TiDB 数据迁移工具的默认配置中，系统库总是被排除。系统库有以下六个：
>
> - INFORMATION_SCHEMA

- PERFORMANCE_SCHEMA
- METRICS_SCHEMA
- INSPECTION_SCHEMA
- mysql
- sys

13.7.10 TiDB Lightning 断点续传

大量的数据导入一般耗时数小时至数天，长时间运行的进程会有一定机率发生非正常中断。如果每次重启都从头开始，就会浪费掉之前已成功导入的数据。为此，TiDB Lightning 提供了“断点续传”的功能，即使 `tidb-lightning` 崩溃，在重启时仍然接着之前的进度继续工作。

本文主要介绍 TiDB Lightning 断点续传的启用与配置、断点的存储，以及断点续传的控制。

13.7.10.1 断点续传的启用与配置

```
[checkpoint]
### 启用断点续传。
### 导入时，TiDB Lightning 会记录当前进度。
### 若 TiDB Lightning 或其他组件异常退出，在重启时可以避免重复再导入已完成的数据。
enable = true

### 存储断点的方式
###  - file：存放在本地文件系统（要求 v2.1.1 或以上）
###  - mysql：存放在兼容 MySQL 的数据库服务器
driver = "file"

### 存储断点的架构名称（数据库名称）
### 仅在 driver = "mysql" 时生效
### schema = "tidb_lightning_checkpoint"

### 断点的存放位置
### # 若 driver = "file"，此参数为断点信息存放的文件路径。
### 如果不设置该参数则默认为 `/tmp/CHECKPOINT_SCHEMA.pb`
### # 若 driver = "mysql"，此参数为数据库连接参数（DSN），格式为“用户:密码@tcp(地址:端口)/”。
### 默认会重用 [tidb] 设置目标数据库来存储断点。
### 为避免加重目标集群的压力，建议另外使用一个兼容 MySQL 的数据库服务器。
### dsn = "/tmp/tidb_lightning_checkpoint.pb"

### 导入成功后是否保留断点。默认为删除。
### 保留断点可用于调试，但有可能泄漏数据源的元数据。
### keep-after-success = false
```

13.7.10.2 断点的存储

TiDB Lightning 支持两种存储方式：本地文件或 MySQL 数据库。

- 若 `driver = "file"`，断点会存放在一个本地文件，其路径由 `dsn` 参数指定。由于断点会频繁更新，建议将这个文件放到写入次数不受限制的盘上，例如 RAM disk。
- 若 `driver = "mysql"`，断点可以存放在任何兼容 MySQL 5.7 或以上的数据库中，包括 MariaDB 和 TiDB。在没有选择的情况下，默认会存在目标数据库里。

目标数据库在导入期间会有大量的操作，若使用目标数据库来存储断点会加重其负担，甚至有可能造成通信超时丢失数据。因此，强烈建议另外部署一台兼容 MySQL 的临时数据库服务器。此数据库也可以安装在 `tidb-lightning` 的主机上。导入完毕后可以删除。

13.7.10.3 断点续传的控制

若 `tidb-lightning` 因不可恢复的错误而退出（例如数据出错），重启时不会使用断点，而是直接报错离开。为保证已导入的数据安全，这些错误必须先解决掉才能继续。使用 `tidb-lightning-ctl` 工具可以标示已经恢复。

13.7.10.3.1 `--checkpoint-error-destroy`

```
tidb-lightning-ctl --checkpoint-error-destroy='`schema`.`table`'
```

该命令会让失败的表从头开始整个导入过程。选项中的架构和表名必须以反引号 (`) 包裹，而且区分大小写。

- 如果导入 `` `schema`.`table` `` 这个表曾经出错，这条命令会：
    1. 从目标数据库移除 (DROP) 这个表，清除已导入的数据。
    2. 将断点重设到“未开始”的状态。
- 如果 `` `schema`.`table` `` 没有出错，则无操作。

传入“all”会对所有表进行上述操作。这是最方便、安全但保守的断点错误解决方法：

```
tidb-lightning-ctl --checkpoint-error-destroy=all
```

13.7.10.3.2 `--checkpoint-error-ignore`

```
tidb-lightning-ctl --checkpoint-error-ignore='`schema`.`table`' &&
tidb-lightning-ctl --checkpoint-error-ignore=all
```

如果导入 `` `schema`.`table` `` 这个表曾经出错，这条命令会清除出错状态，如同没事发生过一样。传入“all”会对所有表进行上述操作。

> 注意：
>
> 除非确定错误可以忽略，否则不要使用这个选项。如果错误是真实的话，可能会导致数据不完全。启用校验和 (CHECKSUM) 可以防止数据出错被忽略。

13.7.10.3.3 --checkpoint-remove

```
tidb-lightning-ctl --checkpoint-remove='`schema`.`table`' &&
tidb-lightning-ctl --checkpoint-remove=all
```

无论是否有出错，把表的断点清除。

13.7.10.3.4 --checkpoint-dump

```
tidb-lightning-ctl --checkpoint-dump=output/directory
```

将所有断点备份到传入的文件夹，主要用于技术支持。此选项仅于 `driver = "mysql"` 时有效。

13.7.11 TiDB Lightning 并行导入

TiDB Lightning 的Physical Import Mode 从 v5.3.0 版本开始支持单表或多表数据的并行导入。通过支持同步启动多个实例，并行导入不同的单表或多表的不同数据，使 TiDB Lightning 具备水平扩展的能力，可大大降低导入大量数据所需的时间。

在技术实现上，TiDB Lightning 通过在目标 TiDB 中记录各个实例以及每个导入表导入数据的元信息，协调不同实例的 Row ID 分配范围、全局 Checksum 的记录和 TiKV 及 PD 的配置变更与恢复。

TiDB Lightning 并行导入可以用于以下场景：

- 并行导入分库分表的数据。在该场景中，来自上游多个数据库实例中的多个表，分别由不同的 TiDB Lightning 实例并行导入到下游 TiDB 数据库中。
- 并行导入单表的数据。在该场景中，存放在某个目录中或云存储（如 Amazon S3）中的多个单表文件，分别由不同的 TiDB Lightning 实例并行导入到下游 TiDB 数据库中。该功能为 v5.3.0 版本引入的新功能。

> 注意：
>
> - 并行导入只支持初始化 TiDB 的空表，不支持导入数据到已有业务写入的数据表，否则可能会导致数据不一致的情况。
> - 并行导入一般用于 Physical Import 模式，需要设置 `incremental-import = true`
> - 并行导入一般用于 Physical Import 模式；虽然也能用于 Logical Import 模式，但是一般不会带来明显的性能提升。

13.7.11.1 使用说明

使用 TiDB Lightning 并行导入需要设置 `incremental-import = true`。TiDB Lightning 在启动时，会在下游 TiDB 中注册元信息，并自动检测是否有其他实例向目标集群导入数据。如果有，则自动进入并行导入模式。

但是在并行导入时，需要注意以下情况：

- 解决主键或者唯一索引的冲突
- 导入性能优化

13.7.11.1.1 解决主键或者唯一索引的冲突

在使用Physical Import Mode 并行导入时，需要确保多个 TiDB Lightning 的数据源之间，以及它们和 TiDB 的目标表中的数据没有主键或者唯一索引的冲突，并且导入的目标表不能有其他应用进行数据写入。否则，TiDB Lightning 将无法保证导入结果的正确性，并且导入完成后相关的数据表将处于数据索引不一致的状态。

13.7.11.1.2 导入性能优化

由于 TiDB Lightning 需要将生成的 Key-Value 数据上传到对应 Region 的每一个副本所在的 TiKV 节点，其导入速度受目标集群规模的限制。在通常情况下，建议确保目标 TiDB 集群中的 TiKV 实例数量与 TiDB Lightning 的实例数量大于 n:1 (n 为 Region 的副本数量)。同时，在使用 TiDB Lightning 并行导入模式时，为达到最优性能，建议进行如下限制：

- 每个 TiDB Lightning 部署在单独的机器上面。TiDB Lightning 默认会消耗所有的 CPU 资源，在单台机器上面部署多个实例并不能提升性能。
- 每个 TiDB Lightning 实例导入的源文件总大小不超过 5 TiB
- TiDB Lightning 实例的总数量不超过 10 个

在使用 TiDB Lightning 并行导入分库分表数据的时候，请根据数据量大小选择使用的 TiDB Lightning 实例数量。

- 如果 MySQL 数据量小于 2 TiB，可以使用 1 个 TiDB Lightning 实例进行并行导入
- 如果 MySQL 数据量超过 2 TiB，并且 MySQL 实例总数小于 10 个，建议每个 MySQL 实例对应 1 个 TiDB Lightning 实例，而且并行 TiDB Lightning 实例数量不要超过 10 个
- 如果 MySQL 数据量超过 2 TiB，并且 MySQL 实例总数超过 10 个，建议这些 MySQL 实例导出的数据平均分配 5 到 10 个 TiDB Lightning 实例进行导入

接下来，本文档将以两个并行导入的示例，详细介绍了不同场景下并行导入的操作步骤：

- 示例 1：使用 Dumpling + TiDB Lightning 并行导入分库分表数据至 TiDB
- 示例 2：使用 TiDB Lightning 并行导入单表数据

13.7.11.1.3 使用限制

TiDB Lightning 在运行时，需要独占部分资源，因此如果需要在单台机器上面部署多个 TiDB Lightning 实例 (不建议生产环境使用) 或多台机器共享磁盘存储时，需要注意如下使用限制：

- 每个 TiDB Lightning 实例的 tikv-importer.sorted-kv-dir 必须设置为不同的路径。多个实例共享相同的路径会导致非预期的行为，可能导致导入失败或数据出错。
- 每个 TiDB Lightning 的 checkpoint 需要分开存储。checkpoint 的详细配置见TiDB Lightning 断点续传。
  - 如果设置 checkpoint.driver = “file”（默认值），需要确保每个实例设置的 checkpoint 的路径不同。
  - 如果设置 checkpoint.driver = “mysql”，需要为每个实例设置不同的 schema。
- 每个 TiDB Lightning 的 log 文件应该设置为不同的路径。共享同一个 log 文件将不利于日志的查询和排查问题。
- 如果开启Web 界面 或 Debug API，需要为每个实例的 `lightning.status-addr` 设置不同地址，否则，TiDB Lightning 进程会由于端口冲突无法启动。

13.7.11.2 示例 1：使用 Dumpling + TiDB Lightning 并行导入分库分表数据至 TiDB

在本示例中，假设上游为包含 10 个分表的 MySQL 集群，总共大小为 10 TiB。使用 5 个 TiDB Lightning 实例并行导入，每个实例导入 2 TiB 数据，预计可以将总导入时间（不包含 Dumpling 导出的耗时）由约 40 小时降至约 10 小时。

假设上游的库名为 `my_db`，每个分表的名字为 `my_table_01` ~ `my_table_10`，需要合并导入至下游的 `my_db.my_table` 表中。下面介绍具体的操作步骤。

13.7.11.2.1 第 1 步：使用 Dumpling 导出数据

在部署 TiDB Lightning 的 5 个节点上面分别导出两个分表的数据：

- 如果两个分表位于同一个 MySQL 实例中，可以直接使用 Dumpling 的 `--filter` 参数一次性导出。此时在使用 TiDB Lightning 导入时，指定 `data-source-dir` 为 Dumpling 数据导出的目录即可；
- 如果两个分表的数据分布在不同的 MySQL 节点上，则需要使用 Dumpling 分别导出，两次导出数据需要放置在同一父目录下不同子目录里，然后在使用 TiDB Lightning 导入时，`data-source-dir` 指定为此父级目录。

使用 Dumpling 导出数据的步骤，请参考Dumpling。

13.7.11.2.2 第 2 步：配置 TiDB Lightning 的数据源

创建 `tidb-lightning.toml` 配置文件，并加入如下内容：

```
[lightning]
status-addr = ":8289"

[mydumper]
### 设置为 Dumpling 导出数据的路径，如果 Dumpling 执行了多次并分属不同的目录，
    ↪ 请将多次导出的数据置放在相同的父目录下并指定此父目录即可。
data-source-dir = "/path/to/source-dir"

[tikv-importer]
### 是否允许向已存在数据的表导入数据。默认值为 false。
### 当使用并行导入模式时，由于多个 TiDB Lightning 实例同时导入一张表，因此此开关必须设置为 true。
```

```
incremental-import = true
### "local": Physical Import Mode，默认使用。适用于 TB 级以上大数据量，但导入期间下游 TiDB
    ↪ 无法对外提供服务。
### "tidb": Logical Import Mode。TB 级以下数据量也可以采用，下游 TiDB 可正常提供服务。
backend = "local"

### 设置本地排序数据的路径
sorted-kv-dir = "/path/to/sorted-dir"
```

如果数据源存放在 Amazon S3 或 GCS 等外部存储中，需要额外的连接配置，你可以为这类配置指定参数。如下例子假设数据源存放在 Amazon S3 中：

```
./tidb-lightning --tidb-port=4000 --pd-urls=127.0.0.1:2379 --backend=local --sorted-kv-dir=/tmp/
    ↪ sorted-kvs \
    -d 's3://my-bucket/sql-backup'
```

更多参数设置，请参考外部存储 URI 格式。

13.7.11.2.3 第 3 步：开启 TiDB Lightning 进行数据导入

在使用 TiDB Lightning 并行导入时，对每个 TiDB Lightning 节点机器配置的需求与非并行导入模式相同，每个 TiDB Lightning 节点需要消耗相同的资源，建议部署在不同的机器上。详细的部署步骤，请参考TiDB Lightning 部署与执行

依次在每台机器上面运行 TiDB Lightning。如果直接在命令行中用 nohup 启动程序，可能会因为 SIGHUP 信号而退出，建议把 nohup 放到脚本里面，如：

```
### !/bin/bash
nohup tiup tidb-lightning -config tidb-lightning.toml > nohup.out &
```

在并行导入的场景下，TiDB Lightning 在启动任务之后，会自动进行下列检查：

- 检查本地盘空间（即 `sort-kv-dir` 配置）以及 TiKV 集群是否有足够空间导入数据，空间大小的详细说明参考TiDB Lightning 下游数据库所需空间和TiDB Lightning 运行时资源要求。检查时会对数据源进行采样，通过采样结果预估索引大小占比。由于估算中考虑了索引，因此可能会出现尽管数据源大小低于本地盘可用空间，但依然无法通过检测的情况。
- 检查 TiKV 集群的 region 分布是否均匀，以及是否存在大量空 region，如果空 region 的数量大于 max(1000, 表的数量 * 3)，即大于“1000”和“3 倍表数量”二者中的最大者，则无法执行导入。
- 检查数据源导入数据是否有序，并且根据检查结果自动调整 `mydumper.batch-size` 的大小。因此 `mydumper.batch-size` 配置不再对用户开放。

你也可以通过 `lightning.check-requirements` 配置来关闭检查，执行强制导入。更多详细检查内容，可以查看Lightning 执行前检查项。

13.7.11.2.4 第 4 步：查看进度

开始导入后，可以通过以下任意方式查看进度：

- 通过 grep 日志关键字 progress 查看进度，默认 5 分钟更新一次。
- 通过监控面板查看进度。详情请参见TiDB Lightning 监控。

等待所有的 TiDB Lightning 运行结束，则整个导入完成。

13.7.11.3 示例 2：使用 TiDB Lightning 并行导入单表数据

TiDB Lightning 也支持并行导入单表的数据。例如，将存放在 Amazon S3 中的多个单表文件，分别由不同的 TiDB Lightning 实例并行导入到下游 TiDB 数据库中。该方法可以加快整体导入速度。TiDB Lightning 使用云端存储时的配置参数与 BR 大致相同，关于详细的参数配置，可以参考外部存储 URI 格式。

> **注意：**
>
> 在本地环境下，可以使用 Dumpling 的 `--filesize` 或 `--where` 参数，预先将单表的数据划分成不同的部分导出至多台机器的本地磁盘，此时依然可以使用并行导入功能，其配置与示例 1 相同。

假设通过 Dumpling 导出的源文件存放在 Amazon S3 云存储中，数据文件为 `my_db.my_table.00001.sql` ~ `my_db.my_table.10000.sql` 共计 10000 个 SQL 文件。如果希望使用 2 个 TiDB Lightning 实例加速导入，则需要在配置文件中增加如下设置：

```
[[mydumper.files]]
### db schema 文件
pattern = '(?i)^(?:[^/]*/)*my_db-schema-create\.sql'
schema = "my_db"
type = "schema-schema"

[[mydumper.files]]
### table schema 文件
pattern = '(?i)^(?:[^/]*/)*my_db\.my_table-schema\.sql'
schema = "my_db"
table = "my_table"
type = "table-schema"

[[mydumper.files]]
### 只导入 00001~05000 这些数据文件并忽略其他文件
pattern = '(?i)^(?:[^/]*/)*my_db\.my_table\.(0[0-4][0-9][0-9][0-9]|05000)\.sql'
schema = "my_db"
table = "my_table"
type = "sql"

[tikv-importer]
### 是否允许向已存在数据的表导入数据。默认值为 false。
### 当使用并行导入模式时，由于多个 TiDB Lightning 实例同时导入一张表，因此此开关必须设置为 true。
```

```
incremental-import = true
```

另外一个实例的配置修改为只导入 05001 ~ 10000 数据文件即可。

其他步骤请参考示例 1 中的相关步骤。

13.7.11.4 错误处理

13.7.11.4.1 部分 TiDB Lightning 节点异常终止

在并行导入过程中，如果一个或多个 TiDB Lightning 节点异常终止，需要首先根据日志中的报错明确异常退出的原因，然后根据错误类型做不同处理：

- 如果是正常退出，例如手动 Kill 或内存溢出被操作系统终止等，可以在适当调整配置后直接重启 TiDB Lightning，无须任何其他操作。
- 如果是不影响数据正确性的报错，例如网络超时，请按以下步骤解决：
    1. 在每一个失败的节点上，执行checkpoint-error-ignore 命令，值设置为 `all`，以清除断点续传源数据中记录的错误。
    2. 重启这些异常的节点，从断点位置继续导入。
- 如果在日志中看到影响数据正确性的报错，如 checksum mismatched，表示源文件中有非法的数据，请按以下步骤解决：
    1. 在每一个 Lightning 节点（包括成功导入数据的节点）上执行checkpoint-error-destroy 命令，以清除失败的表中已导入的数据，并将这些表的 checkpoint 状态重置为 “not yet started”。
    2. 使用`filter` 参数在所有 TiDB Lightning 节点（包括任务正常结束的节点）上重新配置和导入失败表的数据。重新配置任务时，不要将 checkpoint-error-destroy 命令放在每一个 Lightning 节点的启动脚本中，否则会删除多个并行导入任务使用的共享元数据，可能会导致数据导入过程出现问题。例如，如果启动了第二个 Lightning 导入任务，它将删除第一个数据导入任务写入的元数据，导致数据导入异常。

13.7.11.4.2 导入过程中报错 “Target table is calculating checksum. Please wait until the checksum is finished and try again”

在部分并行导入场景，如果表比较多或者一些表的数据量比较小，可能会出现当一个或多个任务开始处理某个表的时候，此表对应的其他任务已经完成，并开始数据一致性校验。此时，由于数据一致性校验不支持写入其他数据，对应的任务会返回 “Target table is calculating checksum. Please wait until the checksum is finished and try again” 错误。等校验任务完成，重启这些失败的任务，报错会消失，数据的正确性也不会受影响。

13.7.12 TiDB Lightning 错误处理功能

从 TiDB 5.4.0 开始，你可以配置 TiDB Lightning 以跳过诸如无效类型转换、唯一键冲突等错误，让导入任务持续进行，就如同出现错误的行数据不存在一样。你可以依据生成的报告，手动修复这些错误。该功能适用于以下场景：

- 要导入的数据有少许错误
- 手动定位错误比较困难
- 如果遇到错误就重启 TiDB Lightning，代价太大

本文介绍了类型错误处理功能 (`lightning.max-error`) 和重复问题处理功能 (`tikv-importer.duplicate-resolution`) 的使用方法，以及保存这些错误的数据库 (`lightning.task-info-schema-name`)，并提供了一个示例。

13.7.12.1 类型错误 (Type error)

你可以通过修改配置项 `lightning.max-error` 来增加数据类型相关的容错数量。如果设置为 N，那么 TiDB Lightning 允许数据源中出现 N 个错误，而且会跳过这些错误，一旦超过这个错误数就会退出。默认值为 0，表示不允许出现错误。

这些错误会被记录到数据库中。在导入完成后，你可以查看数据库中的数据，手动进行处理。请参见错误报告。

```
[lightning]
max-error = 0
```

该配置对下列错误有效：

- 无效值。例如：在 INT 列设置了 `'Text'`
- 数字溢出。例如：在 TINYINT 列设置了 500
- 字符串溢出。例如: 在 VARCHAR(5) 列中设置了'非常长的文字'
- 零日期时间，如 `'0000-00-00'` 和 `'2021-12-00'`
- 在 NOT NULL 列中设置了 NULL
- 生成的列表达式求值失败
- 列计数不匹配。行中数值的数量和列的数量不一致
- `on-duplicate = "error"` 时，TiDB 后端的唯一键/主键冲突
- 其他 SQL 错误

下列错误是致命错误，不能通过配置 `max-error` 跳过：

- 原始 CSV、SQL 或者 Parquet 文件中的语法错误，例如未闭合的引号
- I/O、网络、或系统权限错误

在 Physical Import Mode 下，唯一键/主键的冲突是单独处理的。相关内容将在接下来的章节进行介绍。

13.7.12.2 错误报告

如果 TiDB Lightning 在运行过程中收集到报错的记录，则在退出时会同时在终端和日志中输出各个类型报错数量的统计信息。

- 输出在终端的报错统计如下表所示：

#	ERROR TYPE	ERROR COUNT	ERROR DATA TABLE
1	Data Type	1000	lightning_task_info.type_error_v1

- 输出在 TiDB Lightning 的 log 文件的结尾如下：

```
[2022/03/13 05:33:57.736 +08:00] [WARN] [errormanager.go:459] ["Detect 1000 data type errors
    ↪  in total, please refer to table `lightning_task_info`.`type_error_v1` for more
    ↪ details"]
```

所有错误都会写入下游 TiDB 集群 `lightning_task_info` 数据库中的表中。在导入完成后，如果收集到报错的数据，你可以根据数据库中记录的内容，手动进行处理。

你可以使用 `lightning.task-info-schema-name` 配置更改数据库名称。

```
[lightning]
task-info-schema-name = 'lightning_task_info'
```

在此数据库中，TiDB Lightning 创建了 3 个表：

```
CREATE TABLE syntax_error_v1 (
    task_id     bigint NOT NULL,
    create_time datetime(6) NOT NULL DEFAULT now(6),
    table_name  varchar(261) NOT NULL,
    path        varchar(2048) NOT NULL,
    offset      bigint NOT NULL,
    error       text NOT NULL,
    context     text
);
CREATE TABLE type_error_v1 (
    task_id     bigint NOT NULL,
    create_time datetime(6) NOT NULL DEFAULT now(6),
    table_name  varchar(261) NOT NULL,
    path        varchar(2048) NOT NULL,
    offset      bigint NOT NULL,
    error       text NOT NULL,
    row_data    text NOT NULL
);
CREATE TABLE conflict_error_v1 (
    task_id     bigint NOT NULL,
    create_time datetime(6) NOT NULL DEFAULT now(6),
    table_name  varchar(261) NOT NULL,
    index_name  varchar(128) NOT NULL,
    key_data    text NOT NULL,
    row_data    text NOT NULL,
    raw_key     mediumblob NOT NULL,
    raw_value   mediumblob NOT NULL,
```

```
    raw_handle  mediumblob NOT NULL,
    raw_row     mediumblob NOT NULL,
    KEY (task_id, table_name)
);
```

type_error_v1 记录由 `max-error` 配置项管理的所有类型错误 (Type error)。每个错误一行。

conflict_error_v1 记录所有后端中的唯一键/主键冲突，每对冲突有两行。

列名	语法	类型	冲突	说明
task_id	√	√	√	生成此错误的 TiDB Lightning 任务 ID
create_time	√	√	√	记录错误的时间
table_name	√	√	√	包含错误的表的名称，格式为 ``db`.`tbl``
path	√	√		包含错误文件的路径
offset	√	√		文件中发现错误的字节位置
error	√	√		错误信息
context	√			围绕错误的文本
index_name			√	冲突中唯一键的名称。主键冲突为 'PRIMARY'
key_data			√	导致错误的行的格式化键句柄。该内容仅供人参考，机器不可读
row_data		√	√	导致错误的格式化行数据。该内容仅供人参考，机器不可读
raw_key			√	冲突的 KV 对的键
raw_value			√	冲突的 KV 对的值
raw_handle			√	冲突行的行句柄
raw_row			√	冲突行的编码值

> 注意：
>
> 错误报告记录的是文件偏移量，不是行号或列号，因为行号或列号的获取效率很低。你可以使用下列命令在字节位实现快速跳转（以 183 为例）：
>
> - shell：输出前面几行
>
> ```
> head -c 183 file.csv | tail
> ```
>
> - shell，输出后面几行
>
> ```
> tail -c +183 file.csv | head
> ```
>
> - vim：`:goto 183` 或 `183go`

13.7.12.3 示例

在该示例中，我们准备了一个包含一些已知错误的数据源。以下是处理这些错误的具体步骤：

1. 准备数据库和表结构：

```
mkdir example && cd example
echo 'CREATE SCHEMA example;' > example-schema-create.sql
echo 'CREATE TABLE t(a TINYINT PRIMARY KEY, b VARCHAR(12) NOT NULL UNIQUE);' > example.t-
    ↪ schema.sql
```

2. 准备数据：

```
cat <<EOF > example.t.1.sql
    INSERT INTO t (a, b) VALUES
    (0, NULL),              -- 列不为空
    (1, 'one'),
    (2, 'two'),
    (40, 'forty'),          -- 与下面的 `40` 冲突
    (54, 'fifty-four'),     -- 与下面的 ``'fifty-four'`` 冲突
    (77, 'seventy-seven'),  -- 字符串长度超过 12 个字符
    (600, 'six hundred'),   -- 数字超出了 TINYINT 数据类型支持的范围
    (40, 'forty'),         -- 与上面的 `40` 冲突
    (42, 'fifty-four');     -- 与上面的 ``'fifty-four'`` 冲突
EOF
```

3. 配置 TiDB Lightning，启用严格 SQL 模式，使用 Local 后端模式进行导入，通过删除解决重复项，并最多跳过 10 个错误：

```
cat <<EOF > config.toml
    [lightning]
    max-error = 10
    [tikv-importer]
    backend = 'local'
    sorted-kv-dir = '/tmp/lightning-tmp/'
    duplicate-resolution = 'remove'
    [mydumper]
    data-source-dir = '.'
    [tidb]
    host = '127.0.0.1'
    port = 4000
    user = 'root'
    password = ''
    sql-mode = 'STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE'
EOF
```

4. 运行 TiDB Lightning。因为已跳过所有错误，该命令执行完会成功退出：

```
tiup tidb-lightning -c config.toml
```

5. 验证导入的表仅包含两个正常行：

```
$ mysql -u root -h 127.0.0.1 -P 4000 -e 'select * from example.t'
+---+-----+
| a | b   |
+---+-----+
| 1 | one |
| 2 | two |
+---+-----+
```

6. 检查 `type_error_v1` 表是否捕获了涉及类型转换的三行：

```
$ mysql -u root -h 127.0.0.1 -P 4000 -e 'select * from lightning_task_info.type_error_v1;' -
    ↪ E
*************************** 1. row ***************************
    task_id: 1635888701843303564
create_time: 2021-11-02 21:31:42.620090
 table_name: `example`.`t`
       path: example.t.1.sql
     offset: 46
      error: failed to cast value as varchar(12) CHARACTER SET utf8mb4 COLLATE utf8mb4_bin
          ↪ for column `b` (#2): [table:1048]Column 'b' cannot be null
   row_data: (0,NULL)
*************************** 2. row ***************************
    task_id: 1635888701843303564
create_time: 2021-11-02 21:31:42.627496
 table_name: `example`.`t`
       path: example.t.1.sql
     offset: 183
      error: failed to cast value as varchar(12) CHARACTER SET utf8mb4 COLLATE utf8mb4_bin
          ↪ for column `b` (#2): [types:1406]Data Too Long, field len 12, data len 13
   row_data: (77,'seventy-seven')
*************************** 3. row ***************************
    task_id: 1635888701843303564
create_time: 2021-11-02 21:31:42.629929
 table_name: `example`.`t`
       path: example.t.1.sql
     offset: 253
      error: failed to cast value as tinyint(4) for column `a` (#1): [types:1690]constant
          ↪ 600 overflows tinyint
   row_data: (600,'six hundred')
```

7. 检查 `conflict_error_v1` 表是否捕获了具有唯一键/主键冲突的四行：

```
$ mysql -u root -h 127.0.0.1 -P 4000 -e 'select * from lightning_task_info.conflict_error_v1
    ↪ ;' --binary-as-hex -E
*************************** 1. row ***************************
```

```
    task_id: 1635888701843303564
create_time: 2021-11-02 21:31:42.669601
 table_name: `example`.`t`
 index_name: PRIMARY
   key_data: 40
   row_data: (40, "forty")
    raw_key: 0x74800000000000000C15F728000000000000028
  raw_value: 0x800001000000020500666F727479
 raw_handle: 0x74800000000000000C15F728000000000000028
    raw_row: 0x800001000000020500666F727479
*************************** 2. row ***************************
    task_id: 1635888701843303564
create_time: 2021-11-02 21:31:42.674798
 table_name: `example`.`t`
 index_name: PRIMARY
   key_data: 40
   row_data: (40, "fourty")
    raw_key: 0x74800000000000000C15F728000000000000028
  raw_value: 0x800001000000020600666F75727479
 raw_handle: 0x74800000000000000C15F728000000000000028
    raw_row: 0x800001000000020600666F75727479
*************************** 3. row ***************************
    task_id: 1635888701843303564
create_time: 2021-11-02 21:31:42.680332
 table_name: `example`.`t`
 index_name: b
   key_data: 54
   row_data: (54, "fifty-four")
    raw_key: 0x74800000000000000C15F6980000000000000001016669667479 2D666FFF7572000000000000F9
  raw_value: 0x0000000000000036
 raw_handle: 0x74800000000000000C15F728000000000000036
    raw_row: 0x800001000000020A0066696674792D666F7572
*************************** 4. row ***************************
    task_id: 1635888701843303564
create_time: 2021-11-02 21:31:42.681073
 table_name: `example`.`t`
 index_name: b
   key_data: 42
   row_data: (42, "fifty-four")
    raw_key: 0x74800000000000000C15F6980000000000000001016669667479 2D666FFF7572000000000000F9
  raw_value: 0x000000000000002A
 raw_handle: 0x74800000000000000C15F728000000000000002A
    raw_row: 0x800001000000020A0066696674792D666F7572
```

13.7.13 TiDB Lightning 故障处理

本文档总结了使用 TiDB Lightning 过程中常见的运行故障及解决方案。

13.7.13.1 TiDB Lightning 导入速度太慢

TiDB Lightning 的正常速度为每条线程每 2 分钟导入一个 256 MB 的数据文件，如果速度远慢于这个数值就是有问题。导入的速度可以检查日志提及 `restore chunk ... takes` 的记录，或者观察 Grafana 的监控信息。

导入速度太慢一般有几个原因：

原因 1：`region-concurrency` 设定太高，线程间争用资源反而减低了效率。

1. 从日志的开头搜寻 `region-concurrency` 能知道 TiDB Lightning 读到的参数是多少。
2. 如果 TiDB Lightning 与其他服务（如 TiKV Importer）共用一台服务器，必需手动将 `region-concurrency` 设为该服务器 CPU 数量的 75%。
3. 如果 CPU 设有限额（例如从 Kubernetes 指定的上限），TiDB Lightning 可能无法自动判断出来，此时亦需要手动调整 `region-concurrency`。

原因 2：表结构太复杂。

每条索引都会额外增加键值对。如果有 N 条索引，实际导入的大小就差不多是 Dumpling 文件的 N+1 倍。如果索引不太重要，可以考虑先从 schema 去掉，待导入完成后再使用 `CREATE INDEX` 加回去。

原因 3：单个文件过大。

把源数据分割为单个大小约为 256 MB 的多个文件时，TiDB Lightning 会并行处理数据，达到最佳效果。如果导入的单个文件过大，TiDB Lightning 可能无响应。

如果源数据是 CSV 格式文件，并且所有的 CSV 文件内都不存在包含字符换行符的字段 (U+000A 及 U+000D)，则可以启用 `strict-format`，TiDB Lightning 会自动分割大文件。

```
[mydumper]
strict-format = true
```

原因 4：TiDB Lightning 版本太旧。

建议试试最新的版本，可能会有改善。

13.7.13.2 `tidb-lightning` 进程意外退出

这种情况可能是启动方式不正确，导致收到 SIGHUP 信号而退出。此时 `tidb-lightning.log` 通常有如下日志：

```
[2018/08/10 07:29:08.310 +08:00] [INFO] [main.go:41] ["got signal to exit"] [signal=hangup]
```

不推荐在命令行中直接使用 nohup 启动进程，推荐使用脚本启动 `tidb-lightning`。

另外，如果从 TiDB Lightning 的 log 的最后一条日志显示遇到的错误是“Context canceled”，需要在日志中搜索第一条“ERROR”级别的日志。在这条日志之前，通常也会紧跟有一条“got signal to exit”，表示 Lightning 是收到中断信号然后退出的。

13.7.13.3 使用 TiDB Lightning 后，TiDB 集群变慢，CPU 占用高

如果 `tidb-lightning` 异常退出，集群可能仍处于“导入模式”(import mode)，该模式不适用于生产环境。此时可执行以下命令查看当前使用的模式：

```
tidb-lightning-ctl --config tidb-lightning.toml --fetch-mode
```

可执行以下命令强制切换回“普通模式”(normal mode)：

```
tidb-lightning-ctl --config tidb-lightning.toml --switch-mode=normal
```

13.7.13.4 TiDB Lightning 报错

13.7.13.4.1 `could not find first pair, this shouldn't happen`

报错原因是遍历本地排序的文件时出现异常，可能在 TiDB Lightning 打开的文件数量超过系统的上限时发生报错。在 Linux 系统中，可以使用 `ulimit -n` 命令确认此值是否过小。建议在导入期间将此设置调整为 1000000（即 `ulimit -n 1000000`）。

13.7.13.4.2 `checksum failed: checksum mismatched remote vs local`

原因：本地数据源跟目标数据库某个表的校验和不一致。这通常有更深层的原因，可以通过检查日志中包含 `checksum mismatched` 的行进一步定位。

包含 `checksum mismatched` 的行中有 `total_kvs: x vs y` 的信息，x 表示导入集群在完成导入后计算出的键值对（KV pairs）数目，y 表示本地数据源产生的键值对数目。

- x 大，即导入集群键值对更多：
  - 可能这张表在导入前已有数据，因此影响了数据校验。这也包括 TiDB Lightning 之前失败停机过，但没有正确重启。
- y 大，即本地数据源键值对更多：
  - 如果目标数据库的校验和全是 0，表示没有发生任何导入，有可能是集群太忙无法接收任何数据。
  - 可能导出数据中包含重复数据，例如唯一键和主键 (UNIQUE and PRIMARY KEYs) 有重复的值、下游表结构为大小写不敏感而数据为大小写敏感。
- 其他情况
  - 如果数据源是由机器生成而不是从 Dumpling 备份的，需确保数据符合表的限制，例如自增 (AUTO_INCREMENT) 的列需要为正数，不能为 0。

解决办法：

1. 使用 `tidb-lightning-ctl` 把出错的表删除，检查表结构与数据，重启 TiDB Lightning 重新导入此前出错的表。

   ```
   tidb-lightning-ctl --config conf/tidb-lightning.toml --checkpoint-error-destroy=all
   ```

2. 把断点存放在外部数据库（修改 `[checkpoint] dsn`），减轻目标集群压力。
3. 参考如何正确重启 TiDB Lightning中的解决办法。

13.7.13.4.3 Checkpoint for … has invalid status: (错误码)

原因：断点续传已启用。TiDB Lightning 或 TiKV Importer 之前发生了异常退出。为了防止数据意外损坏，TiDB Lightning 在错误解决以前不会启动。

错误码是小于 25 的整数，可能的取值是 0、3、6、9、12、14、15、17、18、20、21。整数越大，表示异常退出所发生的步骤在导入流程中越晚。

解决办法：

如果错误原因是非法数据源，使用 `tidb-lightning-ctl` 删除已导入数据，并重启 TiDB Lightning。

```
tidb-lightning-ctl --config conf/tidb-lightning.toml --checkpoint-error-destroy=all
```

其他解决方法请参考断点续传的控制。

13.7.13.4.4 ResourceTemporarilyUnavailable("Too many open engines …: …")

原因：并行打开的引擎文件 (engine files) 超出 `tikv-importer` 里的限制。这可能由配置错误引起。即使配置没问题，如果 `tidb-lightning` 曾经异常退出，也有可能令引擎文件残留在打开的状态，占据可用的数量。

解决办法：

1. 提高 `tikv-importer.toml` 内 `max-open-engines` 的值。这个设置主要由内存决定，计算公式为：

   最大内存使用量 ≈ `max-open-engines` × `write-buffer-size` × `max-write-buffer-number`

2. 降低 `table-concurrency` + `index-concurrency`，使之低于 `max-open-engines`。
3. 重启 `tikv-importer` 来强制移除所有引擎文件 (默认值为 `./data.import/`)。这样也会丢弃导入了一半的表，所以启动 TiDB Lightning 前必须清除过期的断点记录：

   ```
   tidb-lightning-ctl --config conf/tidb-lightning.toml --checkpoint-error-destroy=all
   ```

13.7.13.4.5 cannot guess encoding for input file, please convert to UTF-8 manually

原因：TiDB Lightning 只支持 UTF-8 和 GB-18030 编码的表架构。此错误代表数据源不是这里任一个编码。也有可能是文件中混合了不同的编码，例如，因为在不同的环境运行过 `ALTER TABLE`，使表架构同时出现 UTF-8 和 GB-18030 的字符。

解决办法：

1. 编辑数据源，保存为纯 UTF-8 或 GB-18030 的文件。
2. 手动在目标数据库创建所有的表。
3. 设置 `[mydumper] character-set = "binary"` 跳过这个检查。但是这样可能使数据库出现乱码。

13.7.13.4.6 [sql2kv] sql encode error = [types:1292]invalid time format: '{1970 1 1 …}'

原因: 一个 `timestamp` 类型的时间戳记录了不存在的时间值。时间值不存在是由于夏时制切换或超出支持的范围 (1970 年 1 月 1 日至 2038 年 1 月 19 日)。

解决办法:

1. 确保 TiDB Lightning 与数据源时区一致。

    - 手动部署的话，通过设定 `$TZ` 环境变量强制时区设定。

      强制使用 Asia/Shanghai 时区：

      ```
      TZ='Asia/Shanghai' bin/tidb-lightning -config tidb-lightning.toml
      ```

2. 导出数据时，必须加上 `--skip-tz-utc` 选项。

3. 确保整个集群使用的是同一最新版本的 `tzdata` (2018i 或更高版本)。

    如果你使用的是 CentOS 机器，你可以运行 `yum info tzdata` 命令查看 `tzdata` 的版本及是否有更新。然后运行 `yum upgrade tzdata` 命令升级 `tzdata`。

13.7.13.4.7 `[Error 8025: entry too large, the max entry size is 6291456]`

原因： TiDB Lightning 生成的单行 KV 超过了 TiDB 的限制。

解决办法:

目前无法绕过 TiDB 的限制，只能忽略这张表，确保其它表顺利导入。

13.7.13.4.8 switch-mode 时遇到 `rpc error: code = Unimplemented ...`

原因： 集群中有不支持 switch-mode 的节点。目前已知的组件中，4.0.0-rc.2 之前的 TiFlash 不支持 switch-mode 操作。

解决办法：

- 如果集群中有 TiFlash 节点，可以将集群更新到 4.0.0-rc.2 或更新版本。
- 如果不方便升级，可以临时禁用 TiFlash。

13.7.13.4.9 `tidb lightning encountered error: TiDB version too old, expected '>=4.0.0', found '3.0.18'`

TiDB Lightning Local-backend 只支持导入到 v4.0.0 及以上版本的 TiDB 集群。如果尝试使用 Local-backend 导入到 v2.x 或 v3.x 的集群，就会报以上错误。此时可以修改配置使用 Importer-backend 或 TiDB-backend 进行导入。

部分 `nightly` 版本的 TiDB 集群的版本可能类似 4.0.0-beta.2。这种版本的 TiDB Lightning 实际支持 Local-backend，如果使用 `nightly` 版本遇到该报错，可以通过设置配置 `check-requirements = false` 跳过版本检查。在设置此参数之前，请确保 TiDB Lightning 的配置支持对应的版本，否则无法保证导入成功。

13.7.13.4.10 `restore table test.district failed: unknown columns in header [...]`

出现该错误通常是因为 CSV 格式的数据文件不包含 header（第一行也是数据），因此需要在 TiDB Lightning 的配置文件中增加如下配置项：

```
[mydumper.csv]
header = false
```

13.7.13.4.11 Unknown character set

由于 TiDB 只支持部分 MySQL 字符集，因此，在导入流程中，如果创建表结构时使用了 TiDB 不支持的字符集，TiDB Lightning 会报这个错误。你可以结合数据内容选择TiDB 支持的字符集，预先在下游创建表结构以绕过这个错误。

13.7.13.4.12 invalid compression type ...

TiDB v6.4.0 及之后版本的 TiDB Lightning 不支持带有非 .bak 后缀的数据文件并报错。你需要提前修改文件名，或将该类文件移出导入数据目录来避免此类错误。更多详情请参考压缩导出的数据文件。

13.7.13.5 TiDB Lightning 配置参数

你可以使用配置文件或命令行配置 TiDB Lightning。本文主要介绍 TiDB Lightning 的全局配置、任务配置，以及如何使用命令行进行参数配置。

13.7.13.5.1 配置文件

TiDB Lightning 的配置文件分为“全局”和“任务”两种类别，二者在结构上兼容。只有当服务器模式开启时，全局配置和任务配置才会有区别；默认情况下，服务器模式为禁用状态，此时 TiDB Lightning 只会执行一个任务，且全局和任务配置使用同一配置文件。

TiDB Lightning 全局配置

```
###### tidb-lightning 全局配置

[lightning]
#### 用于进度展示 web 界面、拉取 Prometheus 监控项、暴露调试数据和提交导入任务（服务器模式下）的
    ↪ HTTP 端口。设置为 0 时为禁用状态。
status-addr = ':8289'

#### 服务器模式，默认值为 false，命令启动后会开始导入任务。如果改为 true，命令启动后会等待用户在
    ↪ web 界面上提交任务。
#### 详情参见“TiDB Lightning web 界面”文档
server-mode = false

#### 日志
level = "info"
file = "tidb-lightning.log"
max-size = 128 # MB
max-days = 28
max-backups = 14
```

TiDB Lightning 任务配置

```
###### tidb-lightning 任务配置

[lightning]
```

```
#### 启动之前检查集群是否满足最低需求，以及运行过程中检查 TiKV 的可用存储空间是否大于 10%。
#### check-requirements = true

#### 引擎文件的最大并行数。
#### 每张表被切分成一个用于存储索引的“索引引擎”和若干存储行数据的“数据引擎”。
#### 这两项设置控制两种引擎文件的最大并发数。
#### 这两项设置的值会影响 tikv-importer 的内存和磁盘用量。
#### 两项数值之和不能超过 tikv-importer 的 max-open-engines 的设定。
index-concurrency = 2
table-concurrency = 6

#### 数据的并发数。默认与逻辑 CPU 的数量相同。
#### 混合部署的情况下可以将其大小配置为逻辑 CPU 数的 75%，以限制 CPU 的使用。
#### region-concurrency =

#### I/O 最大并发数。I/O 并发量太高时，会因硬盘内部缓存频繁被刷新
#### 而增加 I/O 等待时间，导致缓存未命中和读取速度降低。
#### 对于不同的存储介质，此参数可能需要调整以达到最佳效率。
io-concurrency = 5

#### TiDB Lightning 停止迁移任务之前能容忍的最大非严重 (non-fatal errors) 错误数。
#### 在忽略非严重错误所在的行数据之后，迁移任务可以继续执行。
#### 将该值设置为 N，表示 TiDB Lightning 会在遇到第 (N+1) 个错误时停止迁移任务。
#### 被忽略的行会被记录到位于目标集群的 "task info" 数据库中。最大错误数可以通过下面参数配置。
#### 默认值为 MaxInt64 字节，即 9223372036854775807 字节。
max-error = 0
#### 参数 task-info-schema-name 指定用于存储 TiDB Lightning 执行结果的数据库。
#### 要关闭该功能，需要将该值设置为空字符串。
#### task-info-schema-name = 'lightning_task_info'

#### 在并行导入模式下，在目标集群保存各个 TiDB Lightning 实例元信息的 schema 名字，默认为 "
    ↪ lightning_metadata"。
#### 如果未开启并行导入模式，无须设置此配置项。
#### **注意：**
#### - 对于参与同一批并行导入的每个 TiDB Lightning 实例，该参数设置的值必须相同，
    ↪ 否则将无法确保导入数据的正确性。
#### - 如果开启并行导入模式，需要确保导入使用的用户（对于 tidb.user 配置项）
    ↪ 有权限创建和访问此配置对应的库。
#### - TiDB Lightning 在导入完成后会删除此 schema，因此不要使用已存在的库名配置该参数。
meta-schema-name = "lightning_metadata"

[security]
#### 指定集群中用于 TLS 连接的证书和密钥。
#### CA 的公钥证书。如果留空，则禁用 TLS。
#### ca-path = "/path/to/ca.pem"
```

```
#### 此服务的公钥证书。
#### cert-path = "/path/to/lightning.pem"
#### 该服务的密钥。
#### key-path = "/path/to/lightning.key"

[checkpoint]
#### 是否启用断点续传。
#### 导入数据时，TiDB Lightning 会记录当前表导入的进度。
#### 所以即使 TiDB Lightning 或其他组件异常退出，在重启时也可以避免重复再导入已完成的数据。
enable = true
#### 存储断点的数据库名称。
schema = "tidb_lightning_checkpoint"
#### 存储断点的方式。
####  - file：存放在本地文件系统。
####  - mysql：存放在兼容 MySQL 的数据库服务器。
driver = "file"

#### dsn 是数据源名称 (data source name)，表示断点的存放位置。
#### 若 driver = "file"，则 dsn 为断点信息存放的文件路径。
#若不设置该路径，则默认存储路径为“/tmp/CHECKPOINT_SCHEMA.pb”。
#### 若 driver = "mysql"，则 dsn 为“用户:密码@tcp(地址:端口)/”格式的 URL。
#### 若不设置该 URL，则默认会使用 [tidb] 部分指定的 TiDB 服务器来存储断点。
#### 为减少目标 TiDB 集群的压力，建议指定另一台兼容 MySQL 的数据库服务器来存储断点。
#### dsn = "/tmp/tidb_lightning_checkpoint.pb"

#### 所有数据导入成功后是否保留断点。设置为 false 时为删除断点。
#### 保留断点有利于进行调试，但会泄漏关于数据源的元数据。
#### keep-after-success = false

[tikv-importer]
#### "local"：Physical Import Mode，默认使用。适用于 TB 级以上大数据量，但导入期间下游 TiDB
    ↪ 无法对外提供服务。
#### "tidb"：Logical Import Mode。TB 级以下数据量可以采用，下游 TiDB 可正常提供服务。
#### backend = "local"
#### 是否允许启动多个 TiDB Lightning 实例（**physical import mode**）
    ↪ 并行导入数据到一个或多个目标表。默认取值为 false。注意，这个参数**不是用于增量导入数据
    ↪ **，仅限目标表为空的场景使用。
#### 多个 TiDB Lightning 实例（physical import mode）同时导入一张表时，此开关必须设置为 true。
    ↪ 但前提是目标表不能存在数据，即所有的数据都只能是由 TiDB Lightning 导入。
#### incremental-import = false
#### 当后端是 “importer” 时，tikv-importer 的监听地址（需改为实际地址）。
addr = "172.16.31.10:8287"
#### Logical Import Mode 插入冲突数据时执行的操作。关于冲突检测详细信息请查阅：https://docs.
    ↪ pingcap.com/zh/tidb/v7.0/tidb-lightning-logical-import-mode-usage#冲突数据检测
#### - replace：新数据替代已有数据
```

```
#### - ignore：保留已有数据，忽略新数据
#### - error：中止导入并报错
#### on-duplicate = "replace"

#### Physical Import Mode 设置是否检测和解决重复的记录（唯一键冲突）。
#### 目前支持三种解决方法：
####  - record：数据写入目标表后，将目标表中重复记录添加到目标 TiDB 中的 `lightning_task_info.
    ↪ conflict_error_v1` 表中。注意，该方法要求目标 TiKV 的版本为 v5.2.0 或更新版本。
    ↪ 如果版本过低，则会启用下面的 'none' 模式。
####  - none：不检测重复记录。该模式是三种模式中性能最佳的，但是如果数据源存在重复记录，会导致
    ↪ TiDB 中出现数据不一致的情况。
####  - remove：记录所有目标表中的重复记录，和 'record' 模式相似。但是会删除目标表所有的重复记录
    ↪ ，以确保目标 TiDB 中的数据状态保持一致。
#### duplicate-resolution = 'none'
#### Physical Import Mode 一次请求中发送的 KV 数量。
#### send-kv-pairs = 32768
#### Physical Import Mode 向 TiKV 发送 KV 时是否启用压缩。目前只支持 Gzip 压缩算法，可填写 "gzip"
    ↪  或者 "gz"。默认不启用压缩。
#### compress-kv-pairs = ""
#### Physical Import Mode 本地进行 KV 排序的路径。如果磁盘性能较低（如使用机械盘），建议设置成与
    ↪ `data-source-dir` 不同的磁盘，这样可有效提升导入性能。
#### sorted-kv-dir = ""
#### Physical Import Mode TiKV 写入 KV 数据的并发度。当 TiDB Lightning 和 TiKV
    ↪ 直接网络传输速度超过万兆的时候，可以适当增加这个值。
#### range-concurrency = 16
#### Physical Import Mode 限制 TiDB Lightning 向每个 TiKV 节点写入的带宽大小，默认为 0，
    ↪ 表示不限制。
#### store-write-bwlimit = "128MiB"

#### 使用 Physical Import Mode 时，配置 TiDB Lightning 本地临时文件使用的磁盘配额 (disk quota)。
    ↪ 当磁盘配额不足时，TiDB Lightning 会暂停读取源数据以及写入临时文件的过程，
    ↪ 优先将已经完成排序的 key-value 写入到 TiKV，TiDB Lightning 删除本地临时文件后，
    ↪ 再继续导入过程。
#### 需要同时配合把 `backend` 设置为 `local` 模式才能生效。
#### 默认值为 MaxInt64 字节，即 9223372036854775807 字节。
#### disk-quota = "10GB"

#### Physical Import Mode 是否通过 SQL 方式添加索引。默认为 `false`，表示 TiDB Lightning
    ↪ 会将行数据以及索引数据都编码成 KV pairs 后一同导入 TiKV，实现机制和历史版本保持一致。
    ↪ 如果设置为 `true`，即 TiDB Lightning 会在导入数据完成后，使用 add index 的 SQL 来添加索引
    ↪ 。
#### 通过 SQL 方式添加索引的优点是将导入数据与导入索引分开，可以快速导入数据，即使导入数据后，
    ↪ 索引添加失败，也不会影响数据的一致性。
#### add-index-by-sql = false
```

```
#### 在使用 TiDB Lightning 导入多租户的 TiDB cluster 的场景下，指定对应的 key space 名称。
    ↪ 默认取值为空字符串，表示 TiDB Lightning 会自动获取导入对应租户的 key space 名称；
    ↪ 如果指定了值，则使用指定的 key space 名称来导入。
#### keyspace-name = ""
[mydumper]
#### 设置文件读取的区块大小，确保该值比数据源的最长字符串长。
read-block-size = "64KiB" # 默认值

#### 引擎文件需按顺序导入。由于并行处理，多个数据引擎几乎在同时被导入，
#### 这样形成的处理队列会造成资源浪费。因此，为了合理分配资源，TiDB Lightning
#### 稍微增大了前几个区块的大小。该参数也决定了比例系数，即在完全并发下
#### “导入”和“写入”过程的持续时间比。这个值可以通过计算 1 GiB 大小的
#### 单张表的（导入时长/写入时长）得到。在日志文件中可以看到精确的时间。
#### 如果“导入”更快，区块大小的差异就会更小；比值为 0 时则说明区块大小一致。
#### 取值范围为（0 <= batch-import-ratio < 1）。
batch-import-ratio = 0.75

#### 本地源数据目录或外部存储 URI。关于外部存储 URI 详情可参考 https://docs.pingcap.com/zh/tidb/
    ↪ v6.6/backup-and-restore-storages#uri-%E6%A0%BC%E5%BC%8F。
data-source-dir = "/data/my_database"

#### 指定包含 `CREATE TABLE` 语句的表结构文件的字符集。只支持下列选项：
####  - utf8mb4：表结构文件必须使用 UTF-8 编码，否则会报错。
####  - gb18030：表结构文件必须使用 GB-18030 编码，否则会报错。
####  - auto：自动判断文件编码是 UTF-8 还是 GB-18030，两者皆非则会报错（默认）。
####  - binary：不尝试转换编码。
character-set = "auto"

#### 指定源数据文件的字符集，Lightning 会在导入过程中将源文件从指定的字符集转换为 UTF-8 编码。
#### 该配置项目前仅用于指定 CSV 文件的字符集。只支持下列选项：
####  - utf8mb4：源数据文件使用 UTF-8 编码。
####  - GB18030：源数据文件使用 GB-18030 编码。
####  - GBK：源数据文件使用 GBK 编码（GBK 编码是对 GB-2312 字符集的拓展，也被称为 Code Page
    ↪ 936）。
####  - binary：不尝试转换编码（默认）。
#### 留空此配置将默认使用 "binary"，即不尝试转换编码。
#### 需要注意的是，Lightning 不会对源数据文件的字符集做假定，仅会根据此配置对数据进行转码并导入。
#### 如果字符集设置与源数据文件的实际编码不符，可能会导致导入失败、导入缺失或导入数据乱码。
data-character-set = "binary"
#### 指定在源数据文件的字符集转换过程中，出现不兼容字符时的替换字符。
#### 此项不可与字段分隔符、引用界定符和换行符号重复。
#### 默认值为 "\uFFFD"，即 UTF-8 编码中的 "error" Rune 或 Unicode replacement character。
#### 改变默认值可能会导致潜在的源数据文件解析性能下降。
data-invalid-char-replace = "\uFFFD"
```

```
#### “严格”格式的导入数据可加快处理速度。
#### strict-format = true 要求：
#### 在 CSV 文件的所有记录中，每条数据记录的值不可包含字符换行符（U+000A 和 U+000D，即 \r 和 \n）
#### 甚至被引号包裹的字符换行符都不可包含，即换行符只可用来分隔行。
#### 导入数据源为严格格式时，TiDB Lightning 会快速定位大文件的分割位置进行并行处理。
#### 但是如果输入数据为非严格格式，可能会将一条完整的数据分割成两部分，导致结果出错。
#### 为保证数据安全而非追求处理速度，默认值为 false。
strict-format = false

#### 如果 strict-format = true，TiDB Lightning 会将 CSV 大文件分割为多个文件块进行并行处理。max-
    ↪ region-size 是分割后每个文件块的最大大小。
#### max-region-size = "256MiB" # 默认值

#### 只导入与该通配符规则相匹配的表。详情见相应章节。
filter = ['*.*', '!mysql.*', '!sys.*', '!INFORMATION_SCHEMA.*', '!PERFORMANCE_SCHEMA.*', '!
    ↪ METRICS_SCHEMA.*', '!INSPECTION_SCHEMA.*']

#### 配置 CSV 文件的解析方式。
[mydumper.csv]
#### 字段分隔符，支持一个或多个字符，默认值为 ','。
separator = ','
#### 引用定界符，设置为空表示字符串未加引号。
delimiter = '"'
#### 行尾定界字符，支持一个或多个字符。设置为空（默认值）表示 "\n"（换行）和 "\r\n" （回车+换行
    ↪ ），均表示行尾。
terminator = ""
#### CSV 文件是否包含表头。
#### 如果 header = true，将把首行的内容作为表头处理，不作为数据导入。如果设置为 false，首行也作为
    ↪  CSV 数据导入，此时请确保 CSV 文件的列顺序与目标表的列顺序一致，否则可能会导致数据差异。
header = true
#### CSV 表头是否匹配目标表的表结构。
#### 默认为 true，表示在导入数据时，会根据 CSV 表头的字段名去匹配目标表对应的列名，这样即使 CSV
    ↪ 文件和目标表列的顺序不一致也能按照对应的列名进行导入。
#### 如果 CSV 表头中的字段名和目标表的列名不匹配（例如，CSV
    ↪ 表头中的某些字段名在目标表中可能找不到对应的同名列）但列的顺序是一致的，请将该配置设置为
    ↪ false。
#### 这时，在导入的时候，会直接忽略 CSV 表头的内容，以避免导入错误。在这种情况下，直接把 CSV
    ↪ 数据按照目标表列的顺序导入。
#### 因此，如果列的顺序不一致，请手动调整一致后再导入，否则可能会导致数据差异。
#### 注意：只有在 header = true 时，该参数才会生效。如果 header = false ，表示 CSV 文件没有表头，
    ↪ 此时不需要考虑相关列名匹配的问题。
header-schema-match = true
#### CSV 文件是否包含 NULL。
#### 如果 not-null = true，CSV 所有列都不能解析为 NULL。
not-null = false
```

```
#### 如果 not-null = false（即 CSV 可以包含 NULL），
#### 为以下值的字段将会被解析为 NULL。
null = '\N'
#### 是否对字段内“\“进行转义
backslash-escape = true
#### 如果有行以分隔符结尾，删除尾部分隔符。
trim-last-separator = false

#### [[mydumper.files]]
#### 解析 AWS Aurora parquet 文件所需的表达式
#### pattern = '(?i)^(?:[^/]*/)*([a-z0-9_]+)\.([a-z0-9_]+)/(?:[^/]*/)*(?:[a-z0-9\-_.]+\.(parquet)
    ↪ )$'
#### schema = '$1'
#### table = '$2'
#### type = '$3'

[tidb]
#### 目标集群的信息。tidb-server 的地址，填一个即可。
host = "172.16.31.1"
port = 4000
user = "root"
#### 设置连接 TiDB 的密码，可为明文或 Base64 编码。
password = ""
#### 表结构信息从 TiDB 的“status-port”获取。
status-port = 10080
#### pd-server 的地址，填一个即可。
pd-addr = "172.16.31.4:2379"
#### tidb-lightning 引用了 TiDB 库，并生成产生一些日志。
#### 设置 TiDB 库的日志等级。
log-level = "error"

#### 设置 TiDB 会话变量，提升 Checksum 和 Analyze 的速度。
#### 各参数定义可参阅”控制 Analyze 并发度“文档
build-stats-concurrency = 20
distsql-scan-concurrency = 15
index-serial-scan-concurrency = 20
checksum-table-concurrency = 2

#### 解析和执行 SQL 语句的默认 SQL 模式。
sql-mode = "ONLY_FULL_GROUP_BY,NO_ENGINE_SUBSTITUTION"
#### `max-allowed-packet` 设置数据库连接允许的最大数据包大小，
#### 对应于系统参数中的 `max_allowed_packet`。 如果设置为 0，
#### 会使用下游数据库 global 级别的 `max_allowed_packet`。
max-allowed-packet = 67_108_864
```

```
#### SQL 连接是否使用 TLS。可选值为：
####  * ""            - 如果填充了 [tidb.security] 部分，则强制使用 TLS（与 "cluster" 情况相同
    ↪ ），否则与 "false" 情况相同
####  * "false"       - 禁用 TLS
####  * "cluster"     - 强制使用 TLS 并使用 [tidb.security] 部分中指定的 CA 验证服务器的证书
####  * "skip-verify" - 强制使用 TLS，但不验证服务器的证书（不安全！）
####  * "preferred"   - 与 "skip-verify" 相同，但是如果服务器不支持 TLS，则会退回到未加密的连接
#### tls = ""
#### 指定证书和密钥用于 TLS 连接 MySQL。
#### 默认为 [security] 部分的副本。
#### [tidb.security]
#### CA 的公钥证书。设置为空字符串可禁用 SQL 的 TLS。
#### ca-path = "/path/to/ca.pem"
#### 该服务的公钥证书。默认为 `security.cert-path` 的副本
#### cert-path = "/path/to/lightning.pem"
#### 此服务的私钥。默认为 `security.key-path` 的副本
#### key-path = "/path/to/lightning.key"

#### 对于 Physical Import Mode，数据导入完成后，TiDB Lightning 可以自动执行 Checksum 和 Analyze
    ↪ 操作。
#### 在生产环境中，建议总是开启 Checksum 和 Analyze。
#### 执行的顺序为：Checksum -> Analyze。
#### 注意：对于 Logical Import Mode，无须执行这两个阶段，因此在实际运行时总是会直接跳过。
[post-restore]
#### 配置是否在导入完成后对每一个表执行 `ADMIN CHECKSUM TABLE <table>` 操作来验证数据的完整性。
#### 可选的配置项：
#### - "required"（默认）。在导入完成后执行 CHECKSUM 检查，如果 CHECKSUM 检查失败，则会报错退出。
#### - "optional"。在导入完成后执行 CHECKSUM 检查，如果报错，会输出一条 WARN 日志并忽略错误。
#### - "off"。导入结束后不执行 CHECKSUM 检查。
#### 默认值为 "required"。从 v4.0.8 开始，checksum 的默认值由此前的 "true" 改为 "required"。
#### # 注意：
#### 1. Checksum 对比失败通常表示导入异常（数据丢失或数据不一致），因此建议总是开启 Checksum。
#### 2. 考虑到与旧版本的兼容性，依然可以在本配置项设置 `true` 和 `false` 两个布尔值，其效果与 `
    ↪ required` 和 `off` 相同。
checksum = "required"
#### 配置是否在 CHECKSUM 结束后对所有表逐个执行 `ANALYZE TABLE <table>` 操作。
#### 此配置的可选配置项与 `checksum` 相同，但默认值为 "optional"。
analyze = "optional"


#### 设置周期性后台操作。
#### 支持的单位：h（时）、m（分）、s（秒）。
[cron]
#### TiDB Lightning 自动刷新导入模式状态的持续时间，该值应小于 TiKV 对应的设定值。
switch-mode = "5m"
```

```
#### 在日志中打印导入进度的持续时间。
log-progress = "5m"

#### 使用 Physical Import Mode 时，检查本地磁盘配额的时间间隔，默认为 60 秒。
#### check-disk-quota = "60s"
```

13.7.13.5.2 命令行参数

`tidb-lightning`

使用 `tidb-lightning` 可以对下列参数进行配置：

参数	描述	对应配置项
–config file	从 file 读取全局设置。如果没有指定则使用默认设置。	
-V	输出程序的版本	
-d directory	读取数据的本地目录或外部存储 URI	`mydumper ↪ . ↪ data ↪ - ↪ source ↪ - ↪ dir`
-L level	日志的等级：debug、info、warn、error 或 fatal (默认为 info)	`lightning ↪ . ↪ log ↪ - ↪ level ↪`

参数	描述	对应配置项
-f rule	表库过滤的规则(可多次指定)	mydumper.filter
--backend backend	选择导入的模式：local为Physical Import Mode，tidb为Logical Import Mode	local
--log-file file	日志文件路径(默认值为/tmp/lightning.log.{timestamp}，设置为‘-’表示日志输出到终端)	lightning.log-file
--status-addr ip:port	TiDB Lightning服务器的监听地址	lightning.status-port

参数	描述	对应配置项
–importer host:port	TiKV Importer 的地址	tikv-importer.addr
–pd-urls host:port	PD endpoint 的地址	tidb.pd-addr
–tidb-host host	TiDB Server 的 host	tidb.host
–tidb-port port	TiDB Server 的端口（默认为 4000）	tidb.port
–tidb-status port	TiDB Server 的状态端口的（默认为 10080）	tidb.status-port
–tidb-user user	连接到 TiDB 的用户名	tidb.user
–tidb-password pass-word	连接到 TiDB 的密码，可为明文或 Base64 编码	tidb.password
–enable-checkpoint bool	是否启用断点 (默认值为 true)	checkpoint.enable

参数	描述	对应配置项
–analyze level	导入后分析表信息，可选值为 required、optional（默认值）、off	post-restore.analyze
–checksum level	导入后比较校验和，可选值为 required（默认值）、optional、off	post-restore.checksum
–check-requirements bool	开始任务之前检查集群版本兼容性，以及运行过程中检查 TiKV 的可用存储空间是否大于 10%（默认值为 true）	lightning.check-requirements
–ca file	TLS 连接的 CA 证书路径	security.ca-path

参数	描述	对应配置项
–cert file	TLS 连接的证书路径	`security.cert-path`
–key file	TLS 连接的私钥路径	`security.key-path`
–server-mode	在服务器模式下启动 TiDB Lightning	`lightning.server-mode`

如果同时对命令行参数和配置文件中的对应参数进行更改，命令行参数将优先生效。例如，在 `cfg.toml` 文件中，不管对日志等级做出什么修改，运行 `./tidb-lightning -L debug --config cfg.toml` 命令总是将日志级别设置为 “debug”。

`tidb-lightning-ctl`

使用 `tidb-lightning-ctl` 可以对下列参数进行配置：

参数	描述
–compact	执行 full compact
–switch-mode mode	将每个 TiKV Store 切换到指定模式（normal 或 import）
–fetch-mode	打印每个 TiKV Store 的当前模式
–import-engine uuid	将 TiKV Importer 上关闭的引擎文件导入到 TiKV 集群
–cleanup-engine uuid	删除 TiKV Importer 上的引擎文件
–checkpoint-dump folder	将当前的断点以 CSV 格式存储到文件夹中
–checkpoint-error-destroy tablename	删除断点，如果报错则删除该表
–checkpoint-error-ignore tablename	忽略指定表中断点的报错
–checkpoint-remove tablename	无条件删除表的断点

tablename 必须是`db`.`tbl` 中的限定表名（包括反引号），或关键词 `all`。

此外，上表中所有 `tidb-lightning` 的参数也适用于 `tidb-lightning-ctl`。

13.7.13.6 TiDB Lightning 命令行参数

你可以使用配置文件或命令行配置 TiDB Lightning。本文主要介绍 TiDB Lightning 的命令行参数。

13.7.13.6.1 命令行参数

`tidb-lightning`

使用 `tidb-lightning` 可以对下列参数进行配置：

参数	描述	对应配置项
–config file	从 file 读取全局设置。如果没有指定则使用默认设置。	
-V	输出程序的版本	
-d directory	读取数据的本地目录或外部存储 URI	`mydumper ↪ . ↪ data ↪ - ↪ source ↪ - ↪ dir`
-L level	日志的等级：debug、info、warn、error 或 fatal (默认为 info)	`lightning ↪ . ↪ log ↪ - ↪ level ↪`
-f rule	表库过滤的规则 (可多次指定)	`mydumper ↪ . ↪ filter ↪`

参数	描述	对应配置项
–backend backend	选择导入的模式：local 为 Physical Import Mode，tidb 为 Logical Import Mode	tikv-importer.backend
–log-file file	日志文件路径（默认值为 /tmp/lightning.log.{timestamp}，设置为 '-' 表示日志输出到终端）	lightning.log-file
–status-addr ip:port	TiDB Lightning 服务器的监听地址	lightning.status-port
–importer host:port	TiKV Importer 的地址	tikv-importer.addr
–pd-urls host:port	PD endpoint 的地址	tidb.pd-addr

参数	描述	对应配置项
–tidb-host host	TiDB Server 的 host	tidb.host
–tidb-port port	TiDB Server 的端口（默认为 4000）	tidb.port
–tidb-status port	TiDB Server 的状态端口的（默认为 10080）	tidb.status-port
–tidb-user user	连接到 TiDB 的用户名	tidb.user
–tidb-password pass-word	连接到 TiDB 的密码，可为明文或 Base64 编码	tidb.password
–enable-checkpoint bool	是否启用断点 (默认值为 true)	checkpoint.enable
–analyze level	导入后分析表信息，可选值为 re-quired、optional（默认值）、off	post-restore.analyze

参数	描述	对应配置项
–checksum level	导入后比较校验和，可选值为 required（默认值）、optional、off	post-restore.checksum
–check-requirements bool	任务开始之前检查集群版本兼容性，以及运行过程中检查 TiKV 的可用存储空间是否大于 10%（默认值为 true）	lightning.check-requirements
–ca file	TLS 连接的 CA 证书路径	security.ca-path
–cert file	TLS 连接的证书路径	security.cert-path

参数	描述	对应配置项
–key file	TLS 连接的私钥路径	security ↪ . ↪ key ↪ - ↪ path ↪
–server-mode	在服务器模式下启动 TiDB Lightning	lightning ↪ . ↪ server ↪ - ↪ mode ↪

如果同时对命令行参数和配置文件中的对应参数进行更改，命令行参数将优先生效。例如，在 `cfg.toml` 文件中，不管对日志等级做出什么修改，运行 `./tidb-lightning -L debug --config cfg.toml` 命令总是将日志级别设置为 “debug”。

`tidb-lightning-ctl`

所有 `tidb-lightning` 的参数也适用于 `tidb-lightning-ctl`。此外，使用 `tidb-lightning-ctl` 还可以对下列参数进行配置：

参数	描述
–compact	执行 full compact
–switch-mode mode	将每个 TiKV Store 切换到指定模式（normal 或 import）
–fetch-mode	打印每个 TiKV Store 的当前模式
–import-engine uuid	将 TiKV Importer 上关闭的引擎文件导入到 TiKV 集群
–cleanup-engine uuid	删除 TiKV Importer 上的引擎文件
–checkpoint-dump folder	将当前的断点以 CSV 格式存储到文件夹中
–checkpoint-error-destroy tablename	删除断点，如果报错则删除该表
–checkpoint-error-ignore tablename	忽略指定表中断点的报错
–checkpoint-remove tablename	无条件删除表的断点

tablename 必须是`db`.`tbl` 中的限定表名（包括反引号），或关键词 `all`。

13.7.13.7 TiDB Lightning 监控告警

`tidb-lightning` 支持使用 Prometheus 采集监控指标 (metrics)。本文主要介绍 TiDB Lightning 的监控配置与监控指标。

13.7.13.7.1 监控配置

如果是手动部署 TiDB Lightning，则参照以下步骤进行配置。

只要 Prometheus 能发现 tidb-lightning 和 tikv-importer 的监控地址，就能收集对应的监控指标。

监控的端口可在 tidb-lightning.toml 中配置：

```
[lightning]
#### 用于调试和 Prometheus 监控的 HTTP 端口。输入 0 关闭。
pprof-port = 8289

...
```

监控的端口也可在 tikv-importer.toml 配置：

```
#### 状态服务器的监听地址
status-server-address = '0.0.0.0:8286'
```

配置 Prometheus 后，tidb-lightning 才能发现服务器。配置方法如下，将服务器地址直接添加至 scrape_configs 部分：

```
...
scrape_configs:
  - job_name: 'lightning'
    static_configs:
      - targets: ['192.168.20.10:8289']
  - job_name: 'tikv-importer'
    static_configs:
      - targets: ['192.168.20.9:8286']
```

13.7.13.7.2 Grafana 面板

Grafana 的可视化面板可以让你在网页上监控 Prometheus 指标。

第一行：速度面板

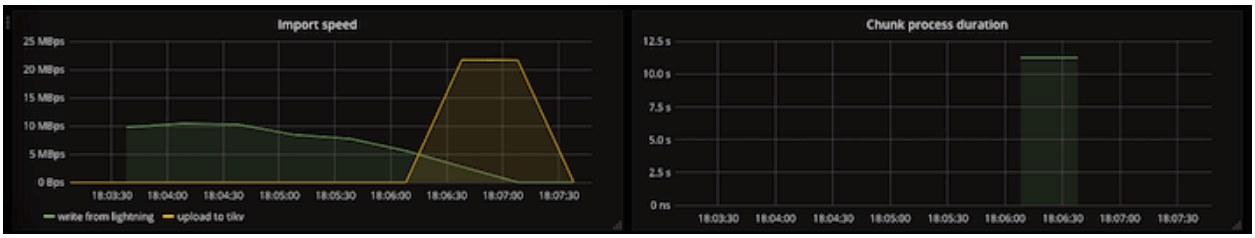


图 248: 第一行速度面板

面板名称	序列	描述
Import speed	write from lightning	从 TiDB Lightning 向 TiKV Importer 发送键值对的速度，取决于每个表的复杂性
Import speed	upload to tikv	从 TiKV Importer 上传 SST 文件到所有 TiKV 副本的总体速度
Chunk process duration		完全编码单个数据文件所需的平均时间

有时导入速度会降到 0，这是为了平衡其他部分的速度，属于正常现象。

第二行：进度面板

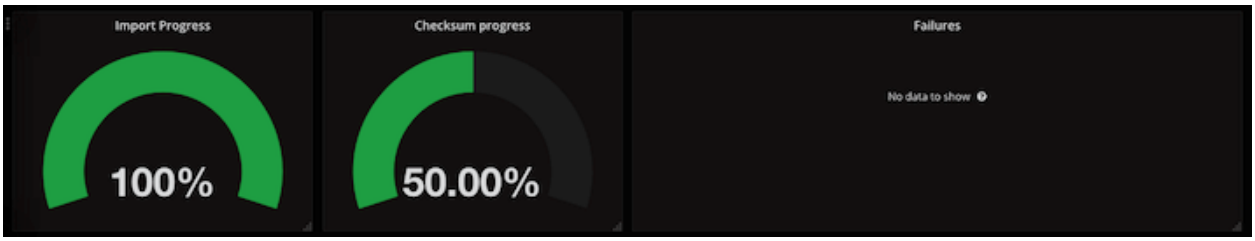


图 249: 第二行进度面板

面板名称	描述
Import progress	已编码的文件所占百分比

面板名称	描述
Checksum progress	已导入的表所占百分比
Failures	导入失败的表的数量以及故障点，通常为空

第三行：资源使用面板

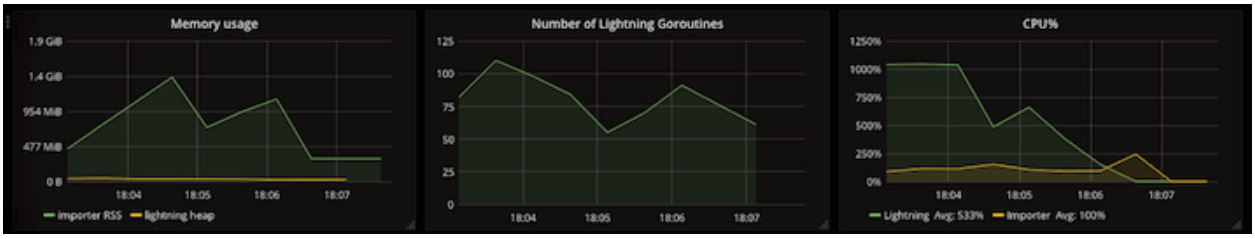


图 250: 第三行资源使用面板

面板名称	描述
Memory usage	每个服务占用的内存
Number of Lightning Goroutines	TiDB Lightning 使用的运行中的 goroutines 数量
CPU%	每个服务使用的逻辑 CPU 数量

第四行：配额使用面板

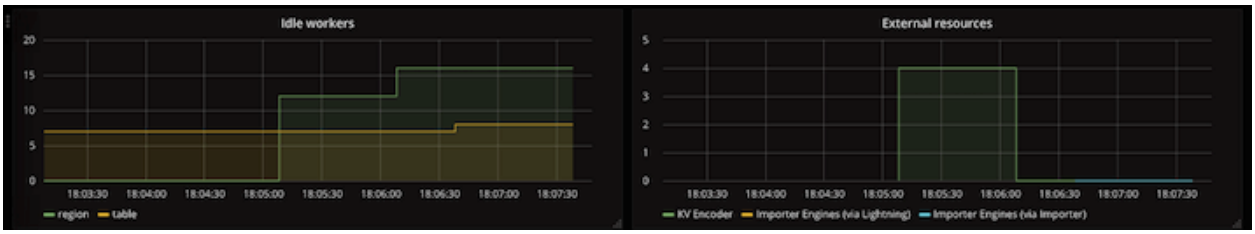


图 251: 第四行配额使用面板

面板名称	序列	描述
Idle workers	io	未使用的 `io-concurrency` 的数量，通常接近配置值（默认为 5），接近 0 时表示磁盘运行太慢
Idle workers	closed-engine	已关闭但未清理的引擎数量，通常接近 `index-concurrency` 与 `table-concurrency` 的和（默认为 8），接近 0 时表示 TiDB Lightning 比 TiKV Importer 快，导致 TiDB Lightning 延迟

面板名称	序列	描述
Idle workers	table	未使用的 table-concurrency 的数量，通常为 0，直到进程结束
Idle workers	index	未使用的 index-concurrency 的数量，通常为 0，直到进程结束
Idle workers	region	未使用的 region-concurrency 的数量，通常为 0，直到进程结束
External resources	KV Encoder	已激活的 KV encoder 的数量，通常与 region-concurrency 的数量相同，直到进程结束

面板名称	序列	描述
External resources	Importer Engines	打开的引擎文件数量，不应超过 `max-open-engines` 的设置

第五行：读取速度面板

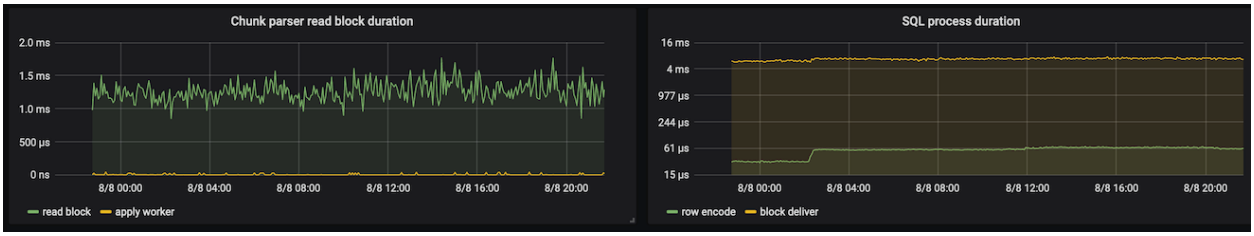


图 252: 第五行读取速度面板

面板名称	序列	描述
Chunk parser read block duration	read block	读取一个字节块来准备解析时所消耗的时间
Chunk parser read block duration	apply worker	等待 `io-concurrency` 空闲所消耗的时间
SQL process duration	row encode	解析和编码单行所消耗的时间
SQL process duration	block deliver	将一组键值对发送到 TiKV Importer 所消耗的时间

如果上述项的持续时间过长，则表示 TiDB Lightning 使用的磁盘运行太慢或 I/O 太忙。

第六行：存储空间面板

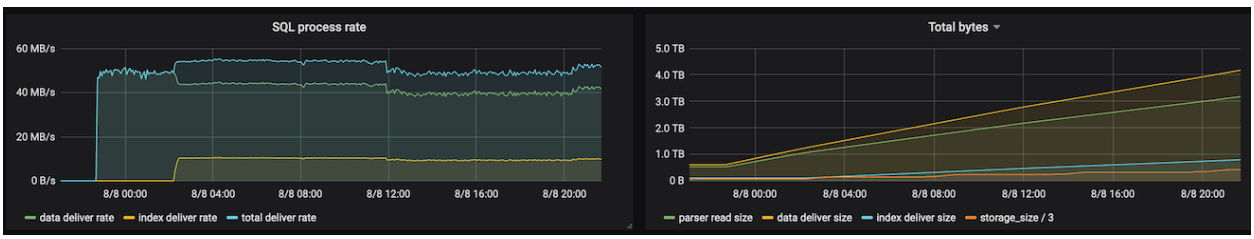


图 253: 第六行存储空间面板

面板名称	序列	描述
SQL process rate	data deliver rate	向 TiKV Importer 发送数据键值对的速度
SQL process rate	index deliver rate	向 TiKV Importer 发送索引键值对的速度
SQL process rate	total deliver rate	发送数据键值对及索引键值对的速度之和
Total bytes	parser read size	TiDB Lightning 正在读取的字节数
Total bytes	data deliver size	已发送到 TiKV Importer 的数据键值对的字节数
Total bytes	index deliver size	已发送到 TiKV Importer 的索引键值对的字节数
Total bytes	storage_size/3	TiKV 集群占用的存储空间大小的 1/3（3 为默认的副本数量）

第七行：导入速度面板



图 254: 第七行导入速度面板

面板名称	序列	描述
Delivery duration	Range delivery	将一个 range 的键值对上传到 TiKV 集群所消耗的时间
Delivery duration	SST delivery	将单个 SST 文件上传到 TiKV 集群所消耗的时间
SST process duration	Split SST	将键值对流切分成若干 SST 文件所消耗的时间
SST process duration	SST upload	上传单个 SST 文件所消耗的时间
SST process duration	SST ingest	ingest 单个 SST 文件所消耗的时间
SST process duration	SST size	单个 SST 文件的大小

13.7.13.7.3 监控指标

本节将详细描述 `tikv-importer` 和 `tidb-lightning` 的监控指标。

`tikv-importer`

`tikv-importer` 的监控指标皆以 `tikv_import_*` 为前缀。

- `tikv_import_rpc_duration`（直方图）

  完成一次 RPC 用时直方图。标签：

  - request：所执行 RPC 请求的类型
    - `switch_mode` — 将一个 TiKV 节点切换为 import/normal 模式
    - `open_engine` — 打开引擎文件
    - `write_engine` — 接收数据并写入引擎文件
    - `close_engine` — 关闭一个引擎文件

    * import_engine — 导入一个引擎文件到 TiKV 集群中
    * cleanup_engine — 删除一个引擎文件
    * compact_cluster — 显式压缩 TiKV 集群
    * upload — 上传一个 SST 文件
    * ingest — Ingest 一个 SST 文件
    * compact — 显式压缩一个 TiKV 节点
  - result：RPC 请求的执行结果
    * ok
    * error

- tikv_import_write_chunk_bytes（直方图）

  从 TiDB Lightning 接收的键值对区块大小（未压缩）的直方图。

- tikv_import_write_chunk_duration（直方图）

  从 tidb-lightning 接收每个键值对区块所需时间的直方图。

- tikv_import_upload_chunk_bytes（直方图）

  上传到 TiKV 的每个 SST 文件区块大小（压缩）的直方图。

- tikv_import_range_delivery_duration（直方图）

  将一个 range 的键值对发送至 dispatch-job 任务所需时间的直方图。

- tikv_import_split_sst_duration（直方图）

  将 range 从引擎文件中分离到单个 SST 文件中所需时间的直方图。

- tikv_import_sst_delivery_duration（直方图）

  将 SST 文件从 dispatch-job 任务发送到 ImportSSTJob 任务所需时间的直方图

- tikv_import_sst_recv_duration（直方图）

  ImportSSTJob 任务接收从 dispatch-job 任务发送过来的 SST 文件所需时间的直方图。

- tikv_import_sst_upload_duration（直方图）

  从 ImportSSTJob 任务上传 SST 文件到 TiKV 节点所需时间的直方图。

- tikv_import_sst_chunk_bytes（直方图）

  上传到 TiKV 节点的 SST 文件（压缩）大小的直方图。

- tikv_import_sst_ingest_duration（直方图）

  将 SST 文件传入至 TiKV 所需时间的直方图。

- tikv_import_each_phase（测量仪）

  表示运行阶段。值为 1 时表示在阶段内运行，值为 0 时表示在阶段内运行。标签：

  - phase：prepare / import

- tikv_import_wait_store_available_count（计数器）

  计算出现 TiKV 节点没有充足空间上传 SST 文件现象的次数。标签：

  - store_id：TiKV 存储 ID。

- tikv_import_upload_chunk_duration（直方图）

  上传到 TiKV 的每个区块所需时间的直方图。

tidb-lightning

tidb-lightning 的监控指标皆以 lightning_* 为前缀。

- lightning_importer_engine（计数器）

  计算已开启及关闭的引擎文件数量。标签：

  - type:
    * open
    * closed

- lightning_idle_workers（计量表盘）

  计算闲置的 worker。标签：

  - name：
    * table — 未使用的 table-concurrency 的数量，通常为 0，直到进程结束
    * index — 未使用的 index-concurrency 的数量，通常为 0，直到进程结束
    * region — 未使用的 region-concurrency 的数量，通常为 0，直到进程结束
    * io — 未使用的 io-concurrency 的数量，通常接近配置值（默认为 5），接近 0 时表示磁盘运行太慢
    * closed-engine — 已关闭但未清理的引擎数量，通常接近 index-concurrency 与 table-concurrency 的和（默认为 8），接近 0 时表示 TiDB Lightning 比 TiKV Importer 快，导致 TiDB Lightning 延迟

- lightning_kv_encoder（计数器）

  计算已开启及关闭的 KV 编码器。KV 编码器是运行于内存的 TiDB 实例，用于将 SQL 的 INSERT 语句转换成键值对。此度量的净值（开启减掉关闭）在正常情况下不应持续增长。标签：

  - type:
    * open
    * closed

- lightning_tables（计数器）

  计算处理过的表及其状态。标签：

  - state：表的状态，表明当前应执行的操作
    * pending — 等待处理
    * written — 所有数据已编码和传输
    * closed — 所有对应的引擎文件已关闭
    * imported — 所有引擎文件已上传到目标集群
    * altered_auto_inc — 自增 ID 已改

        * `checksum` — 已计算校验和
        * `analyzed` — 已进行统计信息分析
        * `completed` — 表格已完全导入并通过验证
    - result：当前操作的执行结果
        * `success` — 成功
        * `failure` — 失败（未完成）

- `lightning_engines`（计数器）

  计算处理后引擎文件的数量以及其状态。标签：

    - state：引擎文件的状态，表明当前应执行的操作
        * `pending` — 等待处理
        * `written` — 所有数据已编码和传输
        * `closed` — 引擎文件已关闭
        * `imported` — 当前引擎文件已上传到目标集群
        * `completed` — 当前引擎文件已完全导入
    - result：当前操作的执行结果
        * `success` — 成功
        * `failure` — 失败（未完成）

- `lightning_chunks`（计数器）

  计算处理过的 Chunks 及其状态。标签：

    - state: 单个 Chunk 的状态，表明该 Chunk 当前所处的阶段
        * `estimated` —（非状态）当前任务中 Chunk 的数量
        * `pending` — 已载入但未执行
        * `running` — 正在编码和发送数据
        * `finished` — 该 Chunk 已处理完毕
        * `failed` — 处理过程中发生错误

- `lightning_import_seconds`（直方图）

  导入每个表所需时间的直方图。

- `lightning_row_read_bytes`（直方图）

  单行 SQL 数据大小的直方图。

- `lightning_row_encode_seconds`（直方图）

  解码单行 SQL 数据到键值对所需时间的直方图。

- `lightning_row_kv_deliver_seconds`（直方图）

  发送一组与单行 SQL 数据对应的键值对所需时间的直方图。

- `lightning_block_deliver_seconds`（直方图）

  每个键值对中的区块传送到 `tikv-importer` 所需时间的直方图。

- `lightning_block_deliver_bytes`（直方图）

  发送到 Importer 的键值对中区块（未压缩）的大小的直方图。

- `lightning_chunk_parser_read_block_seconds`（直方图）

  数据文件解析每个 SQL 区块所需时间的直方图。

- `lightning_checksum_seconds`（直方图）

  计算表中 Checksum 所需时间的直方图。

- `lightning_apply_worker_seconds`（直方图）

  获取闲置 worker 等待时间的直方图 (参见 `lightning_idle_workers` 计量表盘)。标签：

  - name：
    * table
    * index
    * region
    * io
    * closed-engine

13.7.13.8 TiDB Lightning Web 界面

TiDB Lightning 支持在网页上查看导入进度或执行一些简单任务管理，这就是 TiDB Lightning 的服务器模式。本文将介绍服务器模式下的 Web 界面和一些常见操作。

启用服务器模式的方式有如下几种：

1. 在启动 `tidb-lightning` 时加上命令行参数 `--server-mode`。

```
tiup tidb-lightning --server-mode --status-addr :8289
```

2. 在配置文件中设置 `lightning.server-mode`。

```
[lightning]
server-mode = true
status-addr = ':8289'
```

TiDB Lightning 启动后，可以访问 `http://127.0.0.1:8289` 来管理程序（实际的 URL 取决于你的 `status-addr` 设置）。

服务器模式下，TiDB Lightning 不会立即开始运行，而是通过用户在 web 页面提交（多个）任务来导入数据。

13.7.13.8.1 TiDB Lightning Web 首页

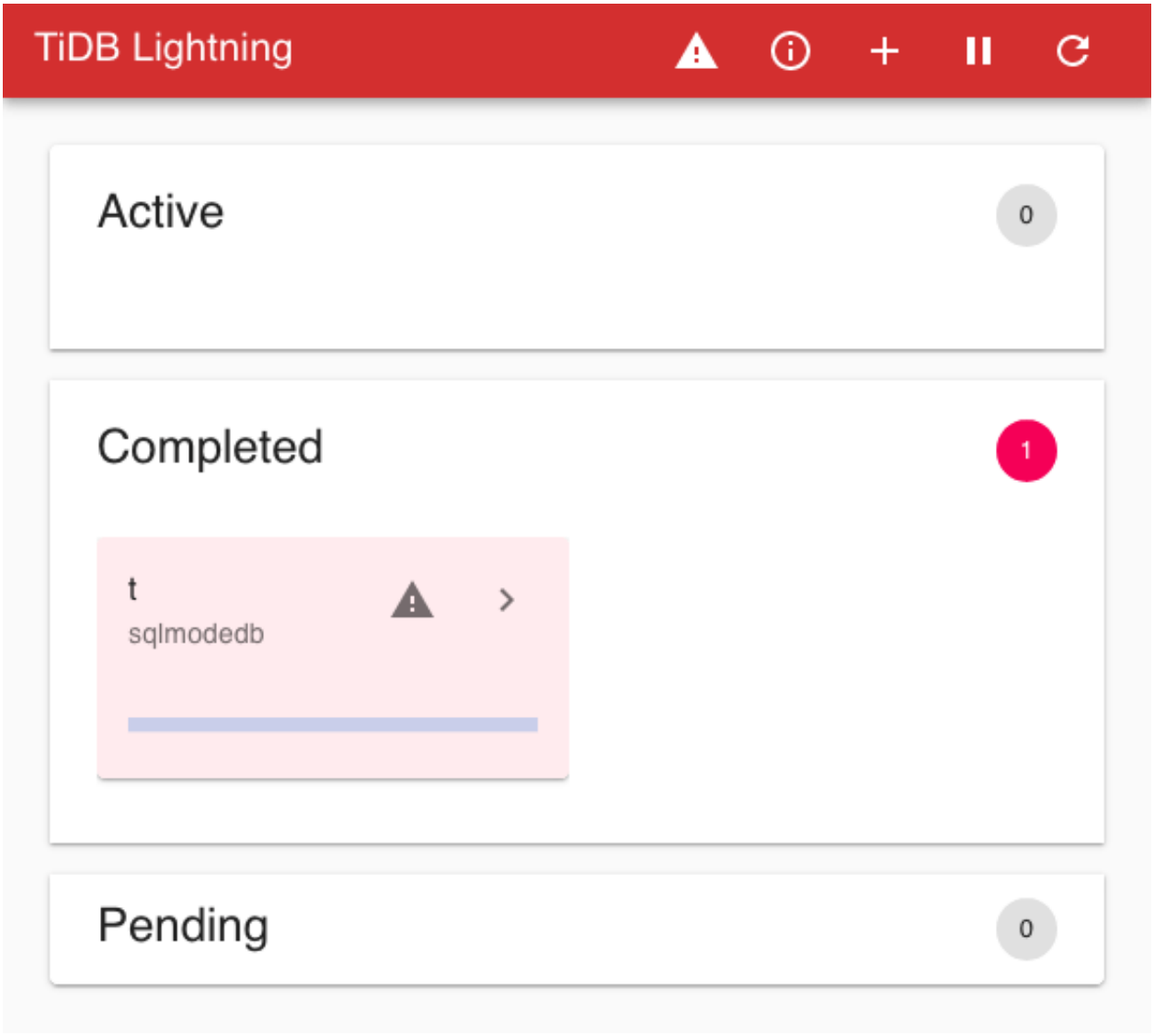


图 255: TiDB Lightning Web 首页

标题栏上图标所对应的功能，从左到右依次为：

图标	功能
“TiDB Lightning”	点击即返回首页
□	显示前一个任务的所有错误信息
ⓘ	列出当前及队列中的任务，可能会出现一个标记提示队列中任务的数量
+	提交单个任务
□/▸	暂停/继续当前操作
□	设置网页自动刷新

标题栏下方的三个面板显示了不同状态下的所有表：

- Active：当前正在导入这些表
- Completed：这些表导入成功或失败
- Pending：这些表还没有被处理

每个面板都包含用于描述表状态的卡片。

13.7.13.8.2 提交任务

点击标题栏的 + 图标提交任务。

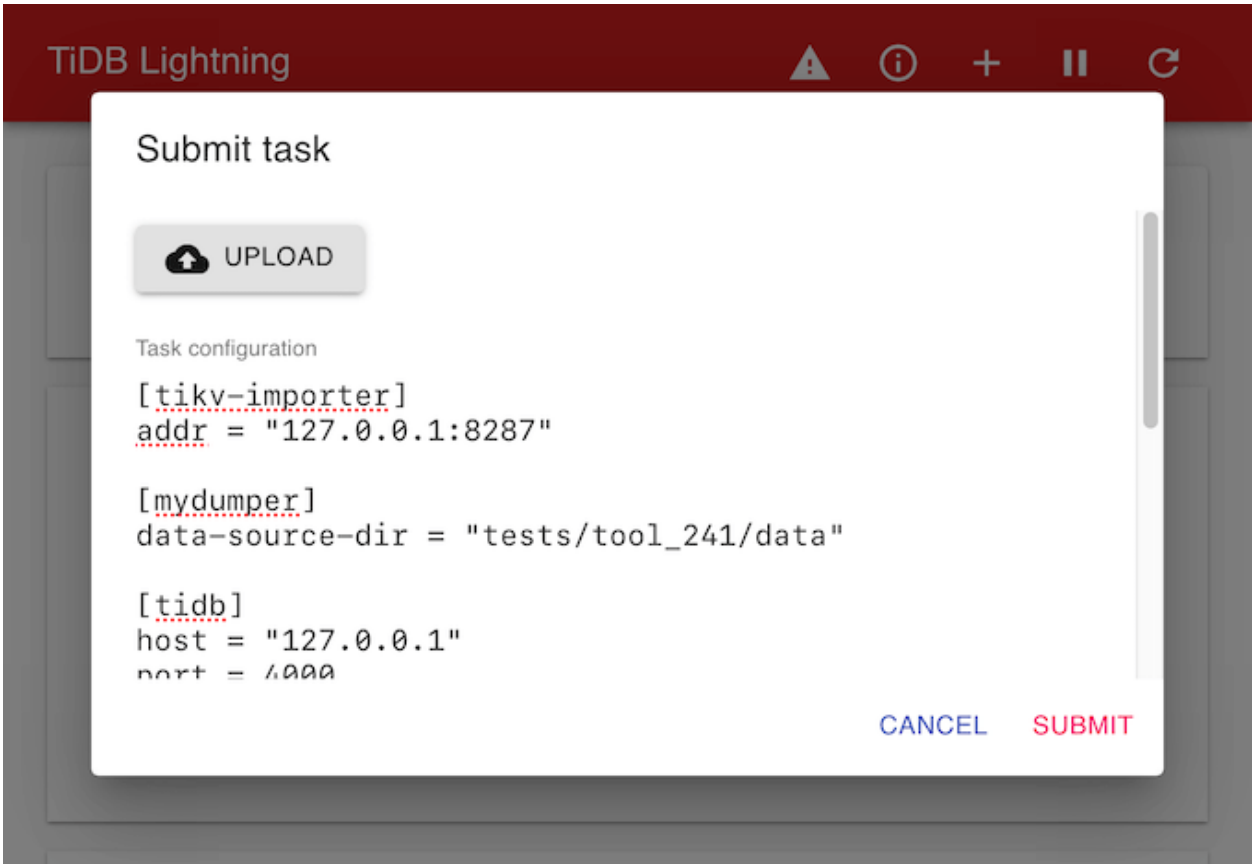


图 256: 提交任务对话框

任务 (task) 为 TOML 格式的文件，具体参考TiDB Lightning 任务配置。你也可以点击 UPLOAD 上传一个本地的 TOML 文件。

点击 SUBMIT 运行任务。如果当前有任务正在运行，新增任务会加入队列并在当前任务结束后执行。

13.7.13.8.3 查看导入进度

点击首页表格卡片上的 > 图标，查看表格导入的详细进度。



图 257: 表格导入进度

该页显示每张表的引擎文件的导入过程。

点击标题栏上的 TiDB Lightning 返回首页。

13.7.13.8.4 管理任务

单击标题栏上的 ⓘ 图标来管理当前及队列中的任务。



图 258: 任务管理页面

每个任务都是依据提交时间来标记。点击该任务将显示 JSON 格式的配置文件。

点击任务上的 ▯ 可以对该任务进行管理。你可以立即停止任务，或重新排序队列中的任务。

13.7.13.9 TiDB Lightning 常见问题

本文列出了一些使用 TiDB Lightning 时可能会遇到的问题与答案。

13.7.13.9.1 TiDB Lightning 对 TiDB/TiKV/PD 的最低版本要求是多少？

TiDB Lightning 的版本应与集群相同。如果使用 Local-backend 模式，最低版本要求为 4.0.0。如果使用 Importer-backend 或 TiDB-backend 模式最低版本要求是 2.0.9，但建议使用最新的稳定版本 3.0。

13.7.13.9.2 TiDB Lightning 支持导入多个库吗？

支持。

13.7.13.9.3 TiDB Lightning 对下游数据库的账号权限要求是怎样的？

详细权限描述参考TiDB Lightning 使用前提。

13.7.13.9.4 TiDB Lightning 在导数据过程中某个表报错了，会影响其他表吗？进程会马上退出吗？

如果只是个别表报错，不会影响整体。报错的那个表会停止处理，继续处理其他的表。

13.7.13.9.5 如何正确重启 TiDB Lightning？

如果使用 Importer-backend，根据 `tikv-importer` 的状态，重启 TiDB Lightning 的基本顺序如下：

如果 `tikv-importer` 仍在运行：

1. 结束 `tidb-lightning` 进程。
2. 执行修改操作（如修复数据源、更改设置、更换硬件等）。
3. 如果上面的修改操作更改了任何表，你还需要清除对应的断点。
4. 重启 `tidb-lightning`。

如果 `tikv-importer` 需要重启：

1. 结束 `tidb-lightning` 进程。
2. 结束 `tikv-importer` 进程。
3. 执行修改操作（如修复数据源、更改设置、更换硬件等）。
4. 重启 `tikv-importer`。
5. 重启 `tidb-lightning` 并等待，直到程序因校验和错误（如果有的话）而失败。
    - 重启 `tikv-importer` 将清除所有仍在写入的引擎文件，但是 `tidb-lightning` 并不会感知到该操作。从 v3.0 开始，最简单的方法是让 `tidb-lightning` 继续，然后再重试。
6. 清除失败的表及断点。
7. 再次重启 `tidb-lightning`。

如果使用 Local-backend 和 TiDB-backend，操作和 Importer-backend 的 `tikv-importer` 仍在运行时相同。

13.7.13.9.6 如何校验导入的数据的正确性？

TiDB Lightning 默认会对导入数据计算校验和 (checksum)，如果校验和不一致就会停止导入该表。可以在日志看到相关的信息。

TiDB 也支持从 MySQL 命令行运行`ADMIN CHECKSUM TABLE` 指令来计算校验和。

```
ADMIN CHECKSUM TABLE `schema`.`table`;
```

```
+---------+------------+--------------------+-----------+-------------+
| Db_name | Table_name | Checksum_crc64_xor | Total_kvs | Total_bytes |
+---------+------------+--------------------+-----------+-------------+
| schema  | table      | 5505282386844578743 |         3 |          96 |
+---------+------------+--------------------+-----------+-------------+
1 row in set (0.01 sec)
```

13.7.13.9.7 TiDB Lightning 支持哪些格式的数据源？

目前，TiDB Lightning 支持：

- 导入Dumpling、CSV 或Amazon Aurora Parquet 输出格式的数据源。
- 从本地盘或 Amazon S3 云盘读取数据。

13.7.13.9.8 我已经在下游创建好库和表了，TiDB Lightning 可以忽略建库建表操作吗？

自 v5.1 起，TiDB Lightning 可以自动识别下游的库和表。如果你使用低于 v5.1 的 TiDB Lightning，需在配置文档中的 `[mydumper]` 部分将 `no-schema` 设置为 `true` 即可。`no-schema=true` 会默认下游已经创建好所需的数据库和表，如果没有创建，会报错。

13.7.13.9.9 有些不合法的数据，能否通过关掉严格 SQL 模式 (Strict SQL Mode) 来导入？

可以。TiDB Lightning 默认的 `sql_mode` 为 `"STRICT_TRANS_TABLES,NO_ENGINE_SUBSTITUTION"`。

这个设置不允许一些非法的数值，例如 `1970-00-00` 这样的日期。可以修改配置文件 `[tidb]` 下的 `sql-mode` 值。

```
...
[tidb]
sql-mode = ""
...
```

13.7.13.9.10 可以启用一个 `tikv-importer`，同时有多个 `tidb-lightning` 进程导入数据吗？

只要每个 TiDB Lightning 操作的表互不相同就可以。

13.7.13.9.11 如何正确结束 `tikv-importer` 进程？

手动部署：如果 `tikv-importer` 正在前台运行，可直接按 Ctrl+C 退出。否则，可通过 `ps aux | grep tikv-importer` 获取进程 ID，然后通过 `kill «pid»` 结束进程。

13.7.13.9.12 如何正确结束 `tidb-lightning` 进程？

根据部署方式，选择相应操作结束进程。

手动部署：如果 `tidb-lightning` 正在前台运行，可直接按 Ctrl+C 退出。否则，可通过 `ps aux | grep tidb-lightning` 获取进程 ID，然后通过 `kill -2 «pid»` 结束进程。

13.7.13.9.13 TiDB Lightning 可以使用千兆网卡吗？

使用 TiDB Lightning 的 SST Mode 建议配置万兆网卡。

千兆网卡的总带宽只有 120 MB/s，而且需要与整个 TiKV 集群共享。在使用 TiDB Lightning Physical Import Mode 导入时，极易用尽所有带宽，继而因 PD 无法联络集群使集群断连。

13.7.13.9.14 为什么 TiDB Lightning 需要在 TiKV 集群预留这么多空间？

当使用默认的 3 副本设置时，TiDB Lightning 需要 TiKV 集群预留数据源大小 6 倍的空间。多出来的 2 倍是算上下列没储存在数据源的因素的保守估计：

- 索引会占据额外的空间
- RocksDB 的空间放大效应

13.7.13.9.15 TiDB Lightning 使用过程中是否可以重启 TiKV Importer？

不能，TiKV Importer 会在内存中存储一些引擎文件，重启后，`tidb-lightning` 会因连接失败而停止。此时，你需要清除失败的断点，因为这些 TiKV Importer 特有的信息丢失了。你可以在之后重启 TiDB Lightning。

13.7.13.9.16 如何清除所有与 TiDB Lightning 相关的中间数据？

1. 删除断点文件。

   ```
   tidb-lightning-ctl --config conf/tidb-lightning.toml --checkpoint-remove=all
   ```

   如果出于某些原因而无法运行该命令，你可以尝试手动删除 `/tmp/tidb_lightning_checkpoint.pb` 文件。

2. 如果使用 Local-backend，删除配置中 `sorted-kv-dir` 对应的目录；如果使用 Importer-backend，删除 `tikv-importer` 所在机器上的整个 `import` 文件目录。

3. 如果需要的话，删除 TiDB 集群上创建的所有表和库。

4. 清理残留的元信息。如果存在以下任意一种情况，需要手动清理元信息库：

   - 对于 v5.1.x 和 v5.2.x 版本的 TiDB Lightning, tidb-lightning-ctl 命令没有同时清理存储在目标集群的 metadata 库，需要手动清理。
   - 如果手动删除过断点文件，则需要手动清理下游的元信息库，否则可能影响后续导入的正确性。

   使用下面命令清理元信息：

   ```
   DROP DATABASE IF EXISTS `lightning_metadata`;
   ```

13.7.13.9.17 如何获取 TiDB Lightning 运行时的 goroutine 信息

1. 如果 TiDB Lightning 的配置文件中已经指定了`status-port`，可以跳过此步骤。否则，需要向 TiDB Lightning 发送 USR1 信号以开启 `status-port`。

   首先通过 `ps` 等命令获取 TiDB Lightning 的进程 PID，然后运行如下命令：

   ```
   kill -USR1 <lightning-pid>
   ```

   查看 TiDB Lightning 的日志，其中 `starting HTTP server` / `start HTTP server` / `started HTTP server` 的日志会显示新开启的 `status-port`。

2. 访问 `http://<lightning-ip>:<status-port>/debug/pprof/goroutine?debug=2` 可获取 goroutine 信息。

13.7.13.9.18 为什么 TiDB Lightning 不兼容 Placement Rules in SQL？

TiDB Lightning 不兼容Placement Rules in SQL。当 TiDB Lightning 导入的数据中包含放置策略 (placement policy) 时，TiDB Lightning 会报错。

不兼容的原因如下：

使用 Placement Rules in SQL，你可以从表或分区级别控制某些 TiKV 节点的数据存储位置。TiDB Lightning 从文本文件中读取数据，并导入到目标 TiDB 集群中。如果导出的数据文件中包含了放置规则 (placement rules) 的定义，在导入过程中，TiDB Lightning 必须根据该定义在目标集群中创建相应的放置规则策略。然而，当源集群和目标集群的拓扑结构不同时，这可能会导致问题。

假设源集群有如下拓扑结构：

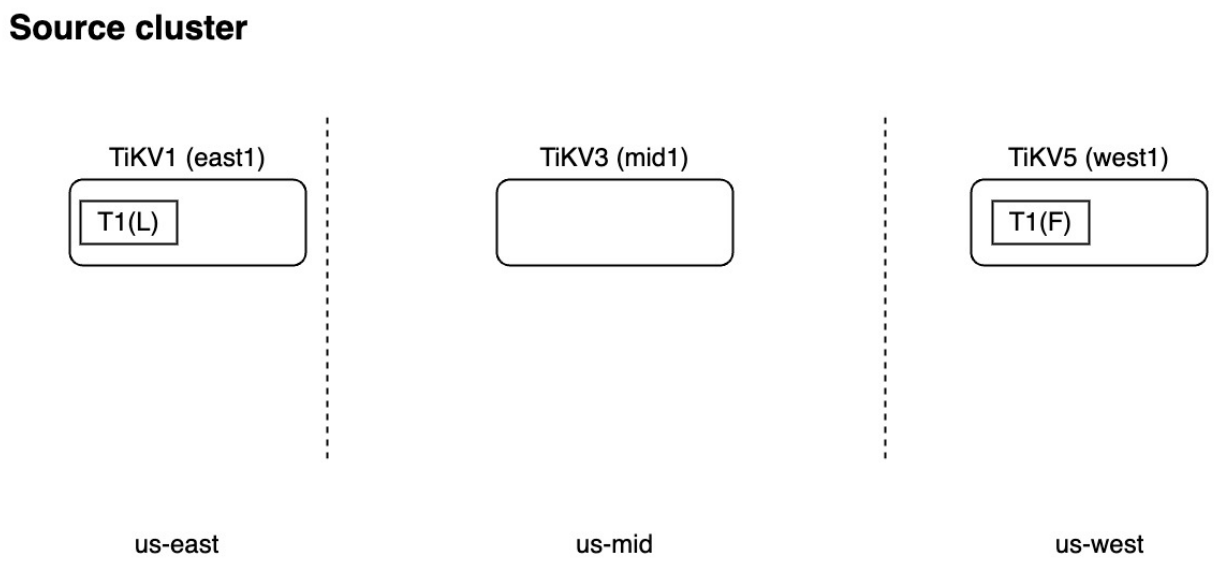


图 259: TiDB Lightning FAQ - 源集群拓扑结构

源集群中设置了这样的放置策略：

```
CREATE PLACEMENT POLICY p1 PRIMARY_REGION="us-east" REGIONS="us-east,us-west";
```

场景 1：目标集群中有 3 个副本，且拓扑结构与源集群不同。在这种情况下，当 TiDB Lightning 在目标集群中创建放置策略时，TiDB Lightning 不会报错，但目标集群中的语义是错误的。

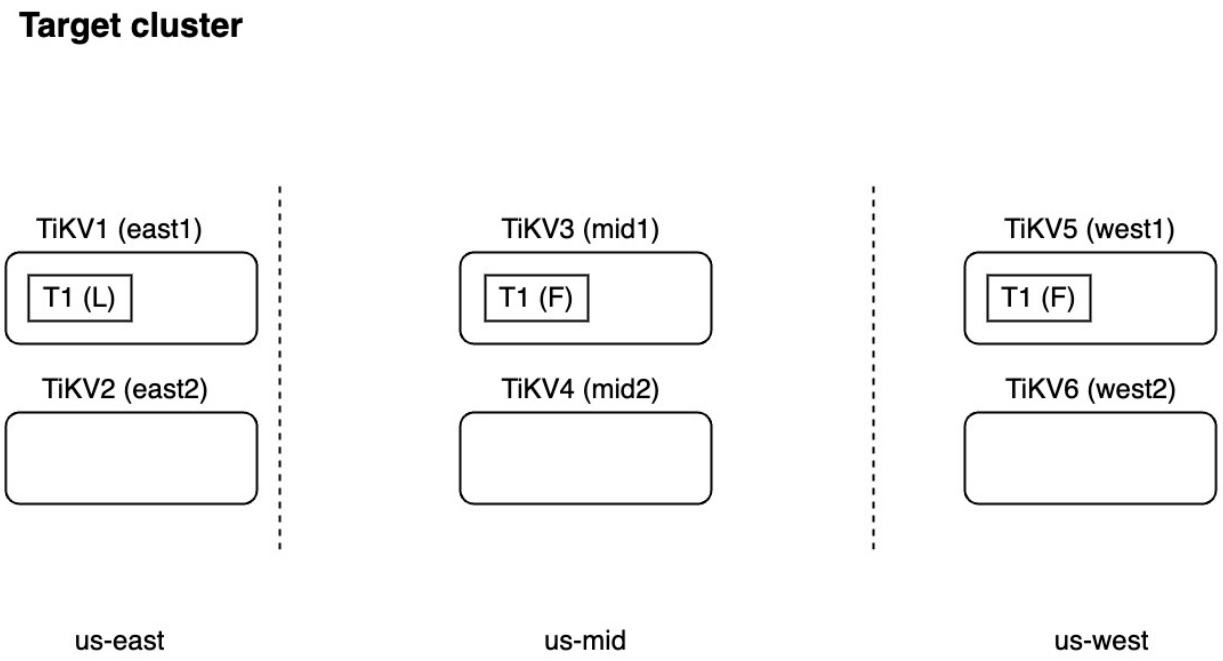


图 260: TiDB Lightning FAQ - 场景 1

场景 2：目标集群将 Follower 副本放置在区域“us-mid”的另一个 TiKV 节点上，且在拓扑结构中没有节点位于区域“us-west”中。在这种情况下，当 TiDB Lightning 在目标集群中创建放置策略时，TiDB Lightning 将报错。

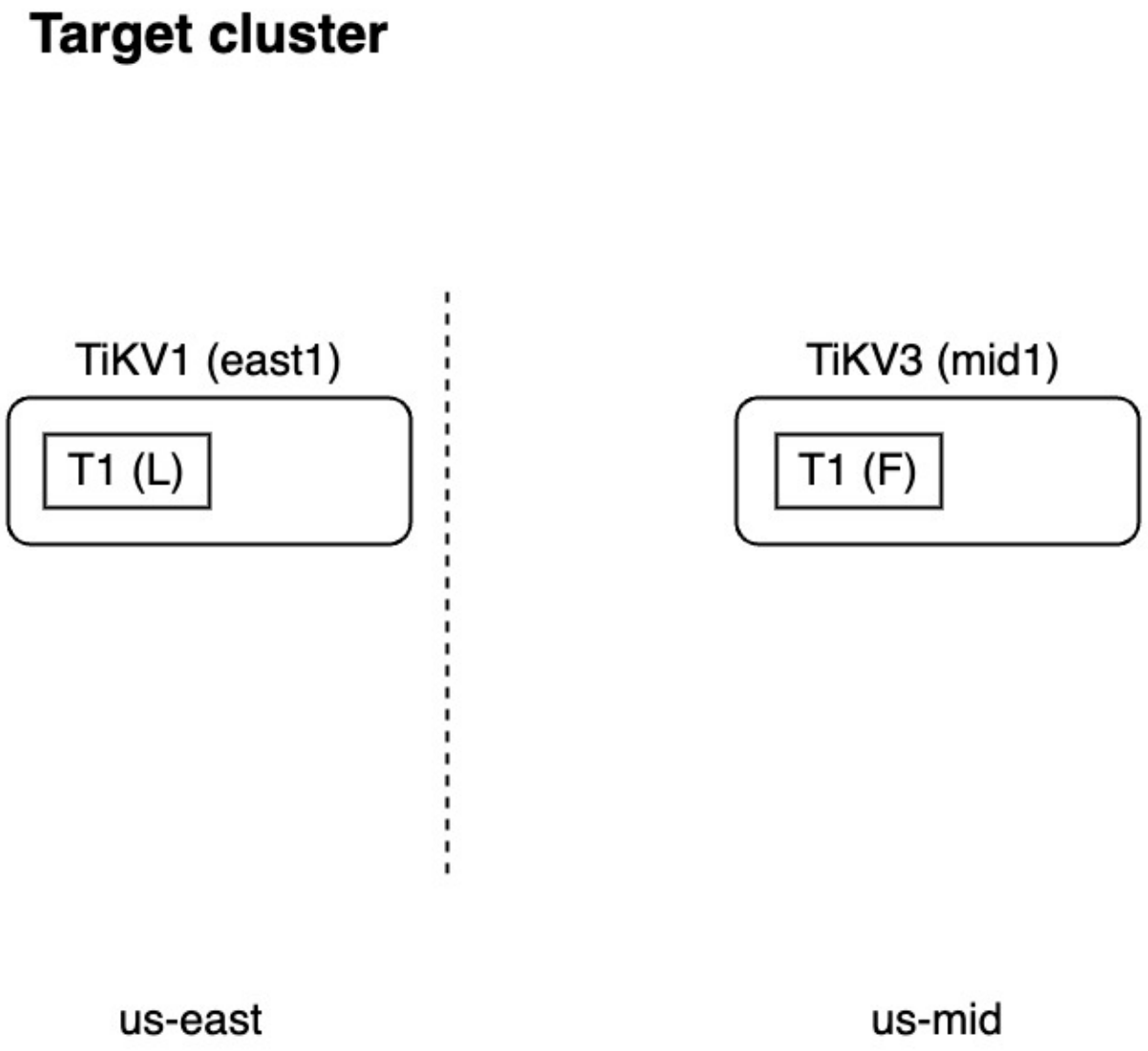


图 261: TiDB Lightning FAQ - 场景 2

解决方法：

如果要在使用 TiDB Lightning 同时使用 Placement Rules in SQL，你需要在导入数据到目标表之前，确保已经在目标 TiDB 集群中创建了相关的 label 和对象。因为 Placement Rules in SQL 作用于 PD 和 TiKV 层，TiDB Lightning 可以根据获取到的信息，决定应该使用哪个 TiKV 来存储导入的数据。使用这种方法后，Placement Rules in SQL 对 TiDB Lightning 是透明无感的。

具体操作步骤如下：

1. 规划数据分布的拓扑结构。
2. 为 TiKV 和 PD 配置必要的 label。
3. 创建放置规则策略，并将策略应用到目标表上。

4. 使用 TiDB Lightning 导入数据到目标表。

13.7.13.10 TiDB Lightning 术语表

本术语表提供了 TiDB Lightning 相关的术语和定义，这些术语会出现在 TiDB Lightning 的日志、监控指标、配置和文档中。

13.7.13.10.1 A

Analyze

统计信息分析。指重建 TiDB 表中的统计信息，即运行ANALYZE TABLE 语句。

因为 TiDB Lightning local backend 直接导入数据到 TiKV，统计信息不会自动更新，所以 TiDB Lightning 在导入后显式地分析每个表。如果不需要该操作，可以将 `post-restore.analyze` 设置为 `false`。

AUTO_INCREMENT_ID

用于为自增列分配默认值的自增 ID 计数器。每张表都有一个相关联的 AUTO_INCREMENT_ID 计数器。在 TiDB 中，该计数器还用于分配行 ID。

因为 TiDB Lightning local backend 直接导入数据到 TiKV，AUTO_INCREMENT_ID 计数器不会自动更新，所以 TiDB Lightning 显式地将 AUTO_INCREMENT_ID 改为一个有效值。即使表中没有自增列，这步仍是会执行。

13.7.13.10.2 B

Backend

也称作 Back end（后端），用于接受 TiDB Lightning 解析结果。

详情参阅TiDB Lightning Backends。

13.7.13.10.3 C

Checkpoint

断点。用于保证 TiDB Lightning 在导入数据时不断地将进度保存到本地文件或远程数据库中。这样即使进程崩溃，TiDB Lightning 也能从中间状态恢复。

详情参见TiDB Lightning 断点续传。

Checksum

校验和。一种用于验证导入数据正确性的方法。

在 TiDB Lightning 中，表的校验和是由 3 个数字组成的集合，由该表中每个键值对的内容计算得出。这些数字分别是：

- 键值对的数量
- 所有键值对的总长度
- 每个键值对 CRC-64-ECMA 按位异或的结果

TiDB Lightning 通过比较每个表的本地校验和和远程校验和来验证导入数据的正确性。如果有任一对校验和不匹配，导入进程就会停止。如果你需要跳过校验和检查，可以将 `post-restore.checksum` 设置为 `false`。

遇到校验和不匹配的问题时，参考常见问题进行处理。

Chunk

一段连续的源数据，通常相当于数据源中的单个文件。

如果单个文件太大，TiDB Lightning 可以将单个文件拆分成多个文件块。

Compaction

压缩。指将多个小 SST 文件合并为一个大 SST 文件并清理已删除的条目。TiDB Lightning 导入数据时，TiKV 在后台会自动压缩数据。

> **注意：**
>
> 出于遗留原因，你仍然可以将 TiDB Lightning 配置为在每次导入表时进行显式压缩，但是官方不推荐采用该操作，且该操作的相关设置默认是禁用的。
>
> 技术细节参阅 RocksDB 关于压缩的说明。

13.7.13.10.4 D

Data engine

数据引擎。用于对实际的行数据进行排序的引擎。

当一个表数据很多的时候，表的数据会被放置在多个数据引擎中以改善任务流水线并节省 TiKV Importer 的空间。默认条件下，每 100 GB 的 SQL 数据会打开一个新的数据引擎（可通过 `mydumper.batch-size` 配置项进行更改）。

TiDB Lightning 同时处理多个数据引擎（可通过 `lightning.table-concurrency` 配置项进行更改）。

13.7.13.10.5 E

Engine

引擎。在 TiKV Importer 中，一个引擎就是一个用于排序键值对的 RocksDB 实例。

TiDB Lightning 通过引擎将数据传送到 TiKV Importer 中。TiDB Lightning 先打开一个引擎，向其发送未排序的键值对，然后关闭引擎。随后，引擎会对收到的键值对进行排序操作。这些关闭的引擎可以进一步上传至 TiKV store 中为Ingest 做准备。

引擎使用 TiKV Importer 的 `import-dir` 作为临时存储，有时也会被称为引擎文件 (engine files)。

另见数据引擎和索引引擎。

13.7.13.10.6 F

Filter

配置列表，用于指定需要导入或不允许导入的表。

详情见表库过滤。

13.7.13.10.7 I

Import mode

导入模式。指通过降低读取速度和减少空间使用，来优化 TiKV 写入的配置模式。

导入过程中，TiDB Lightning 自动在导入模式和普通模式中来回切换。如果 TiKV 卡在导入模式，你可以使用 `tidb-lightning-ctl` 强制切换回普通模式。

Index engine

索引引擎。用于对索引进行排序的引擎。

不管表中有多少索引，每张表都只对应一个索引引擎。

TiDB Lightning 可同时处理多个索引引擎（可通过 `lightning.index-concurrency` 配置项进行更改）。由于每张表正好对应一个索引引擎，`lightning.index-concurrency` 配置项也限定了可同时处理的表的最大数量。

Ingest

指将SST 文件的全部内容插入到 RocksDB（TiKV）store 中的操作。

与逐个插入键值对相比，Ingest 的效率非常高。因此，该操作直接决定了 TiDB Lightning 的性能。

技术细节参阅 RocksDB 关于创建、Ingest SST 文件的 wiki 页面。

13.7.13.10.8 K

KV pair

即 key-value pair（键值对）。

KV encoder

用于将 SQL 或 CSV 行解析为键值对的例程。多个 KV encoder 会并行运行以加快处理速度。

13.7.13.10.9 L

Local checksum

本地校验和。在将键值对发送到 TiKV Importer 前，由 TiDB Lightning 计算的表的校验和。

13.7.13.10.10 N

Normal mode

普通模式。未启用导入模式时的模式。

13.7.13.10.11 P

Post-processing

指整个数据源被解析发送到 TiKV Importer 之后的一段时间。此时 TiDB Lightning 正在等待 TiKV Importer 上传、Ingest SST 文件。

13.7.13.10.12 R

Remote checksum

远程校验和。指导入 TiDB 后所计算的表的校验和。

13.7.13.10.13 S

Scattering

指随机再分配Region 中 leader 和 peer 的操作。Scattering 确保导入的数据在 TiKV store 中均匀分布，这样可以降低 PD 调度的压力。

Splitting

指 TiKV Importer 在上传之前会将单个引擎文件拆分为若干小SST 文件的操作。这是因为引擎文件通常很大（约为 100 GB），在 TiKV 中不适合视为单一的Region。拆分的文件大小可通过 `import.region-split-size` 配置项更改。

SST file

Sorted string table file（排序字符串表文件）。SST 文件是一种在 RocksDB 中（因而也是 TiKV 中）键值对集合在本地的存储形式。

TiKV Importer 从关闭的引擎中生成 SST 文件。这些 SST 文件接着被上传、 ingest 到 TiKV store 中。

13.8 使用 Dumpling 导出数据

使用数据导出工具 Dumpling，你可以把存储在 TiDB 或 MySQL 中的数据导出为 SQL 或 CSV 格式，用于逻辑全量备份。 Dumpling 也支持将数据导出到 Amazon S3 中。

要快速了解 Dumpling 的基本功能，建议先观看下面的培训视频（时长 28 分钟）。注意本视频只作为功能介绍、学习参考，具体操作步骤和最新功能，请以文档内容为准。

你可以通过下列任意方式获取 Dumpling：

- TiUP 执行 `tiup install dumpling` 命令。获取后，使用 `tiup dumpling ...` 命令运行 Dumpling。
- 下载包含 Dumpling 的tidb-toolkit 安装包。

更多详情，可以使用 –help 选项查看，或参考Dumpling 主要选项表。

使用 Dumpling 时，需要在已经启动的集群上执行导出命令。

TiDB 还提供了其他工具，你可以根据需要选择使用：

- 如果需要直接备份 SST 文件（键值对），或者对延迟不敏感的增量备份，请使用备份工具BR。

- 如果需要实时的增量备份，请使用TiCDC。
- 所有的导出数据都可以用TiDB Lightning 导回到 TiDB。

> 注意：
>
> PingCAP 之前维护的 Mydumper 工具 fork 自 mydumper project，针对 TiDB 的特性进行了优化。关于 Mydumper 的更多信息，请参考 v4.0 版 Mydumper 使用文档。Mydumper 目前已经不再开发新功能，其绝大部分功能已经被Dumpling 取代，请切换到 Dumpling。

相比 Mydumper，Dumpling 做了如下改进：

- 支持导出多种数据形式，包括 SQL/CSV。
- 支持全新的 table-filter，筛选数据更加方便。
- 支持导出到 Amazon S3 云盘。
- 针对 TiDB 进行了更多优化：
  - 支持配置 TiDB 单条 SQL 内存限制。
  - 针对 TiDB v4.0.0 及更新版本，如果 Dumpling 能够直接连接到 PD，则支持自动调整 TiDB GC 时间。
  - 使用 TiDB 的隐藏列 `_tidb_rowid` 优化了单表内数据的并发导出性能。
  - 对于 TiDB 可以设置tidb_snapshot 的值指定备份数据的时间点，从而保证备份的一致性，而不是通过 `FLUSH TABLES WITH READ LOCK` 来保证备份一致性。

> 注意：
>
> 在以下情况下，Dumpling 无法连接到 PD：
>
> - TiDB 集群正在 Kubernetes 上运行（Dumpling 本身在 Kubernetes 环境中运行时除外）。
> - TiDB 集群正在 TiDB Cloud 上运行。
>
> 在这种情况下，你需要手动调整 TiDB GC 时间，以避免导出失败。

13.8.1 从 TiDB/MySQL 导出数据

13.8.1.1 需要的权限

- SELECT：导出目标表时需要。
- RELOAD：使用 consistency flush 时需要。注意，只有 TiDB 支持该权限，当上游为 RDS 或采用托管服务时，可忽略该权限。
- LOCK TABLES：使用 consistency lock 时需要，需要导出的库表都有该权限。
- REPLICATION CLIENT：导出 metadata 记录数据快照点时需要，可选，如果不需要导出 metadata，可忽略该权限。

13.8.1.2 导出为 SQL 文件

本文假设在 127.0.0.1:4000 有一个 TiDB 实例，并且这个 TiDB 实例中有无密码的 root 用户。

Dumpling 默认导出数据格式为 SQL 文件。也可以通过设置 `--filetype sql` 导出数据到 SQL 文件：

```
dumpling -u root -P 4000 -h 127.0.0.1 --filetype sql -t 8 -o /tmp/test -r 200000 -F256MiB
```

以上命令中：

- -h、-P、-u 分别代表地址、端口、用户。如果需要密码验证，可以使用 -p $YOUR_SECRET_PASSWORD 将密码传给 Dumpling。
- -o（或 --output）用于选择存储导出文件的目录，支持本地文件路径或外部存储 URI 格式。
- -t 用于指定导出的线程数。增加线程数会增加 Dumpling 并发度提高导出速度，但也会加大数据库内存消耗，因此不宜设置过大。一般不超过 64。
- -r 用于开启表内并发加速导出。默认值是 0，表示不开启。取值大于 0 表示开启，取值是 INT 类型。当数据源为 TiDB 时，设置 -r 参数大于 0 表示使用 TiDB region 信息划分区间，同时减少内存使用。具体取值不影响划分算法。对数据源为 MySQL 且表的主键是 INT 的场景，该参数也有表内并发效果。
- -F 选项用于指定单个文件的最大大小，单位为 MiB，可接受类似 5GiB 或 8KB 的输入。如果你想使用 TiDB Lightning 将该文件加载到 TiDB 实例中，建议将 -F 选项的值保持在 256 MiB 或以下。

> 注意：
>
> 如果导出的单表大小超过 10 GB，强烈建议使用 -r 和 -F 参数。

13.8.1.3 导出为 CSV 文件

你可以通过使用 `--filetype csv` 导出数据到 CSV 文件。

当你导出 CSV 文件时，你可以使用 `--sql <SQL>` 导出指定 SQL 选择出来的记录。例如，导出 test.sbtest1 中所有 id < 100 的记录：

```
./dumpling -u root -P 4000 -h 127.0.0.1 -o /tmp/test --filetype csv --sql 'select * from `test`.`
    ↪ sbtest1` where id < 100' -F 100MiB --output-filename-template 'test.sbtest1.{{.Index}}'
```

以上命令中：

- --sql 选项仅仅可用于导出 CSV 文件的场景。上述命令将在要导出的所有表上执行 SELECT * FROM <table-name> WHERE id < 100 语句。如果部分表没有指定的字段，那么导出会失败。
- 使用 --sql 配置导出时，Dumpling 无法获知导出的表库信息，此时可以使用 --output-filename-template 选项来指定 CSV 文件的文件名格式，以方便后续使用TiDB Lightning 导入数据文件。例如 --output-filename-template='test.sbtest1.{{.Index}}' 指定导出的 CSV 文件为 test.sbtest1.000000000、test.sbtest1.000000001 等。
- 你可以使用 --csv-separator、--csv-delimiter 等选项，配置 CSV 文件的格式。具体信息可查阅Dumpling 主要选项表。

> **注意：**
>
> Dumpling 导出不区分字符串与关键字。如果导入的数据是 Boolean 类型的 `true` 和 `false`，导出时会被转换为 1 和 0。

13.8.1.4 压缩导出的数据文件

你可以使用 `--compress <format>` 压缩导出的 CSV、SQL 数据与表结构文件。该参数支持 `gzip`、`snappy`、`zstd` 压缩算法。默认不压缩。

- 该选项只能压缩单个数据与表结构文件，无法直接压缩整个文件夹生成单个压缩集合包。
- 该选项可以节省磁盘空间，但也会导致导出速度变慢，并增加 CPU 消耗。对导出速度要求较高的场景需慎用。
- TiDB Lightning v6.5.0 及以上版本支持直接使用 Dumpling 压缩文件作为数据源导入，无需额外配置。

13.8.1.5 输出文件格式

- `metadata`：此文件包含导出的起始时间，以及 master binary log 的位置。

```
cat metadata
```

```
Started dump at: 2020-11-10 10:40:19
SHOW MASTER STATUS:
        Log: tidb-binlog
        Pos: 420747102018863124

Finished dump at: 2020-11-10 10:40:20
```

- `{schema}-schema-create.sql`：创建 schema 的 SQL 文件。

```
cat test-schema-create.sql
```

```
CREATE DATABASE `test` /*!40100 DEFAULT CHARACTER SET utf8mb4 */;
```

- `{schema}.{table}-schema.sql`：创建 table 的 SQL 文件

```
cat test.t1-schema.sql
```

```
CREATE TABLE `t1` (
  `id` int(11) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin;
```

- `{schema}.{table}.{0001}.{sql|csv}`：数据源文件

```
cat test.t1.0.sql
```

```
/*!40101 SET NAMES binary*/;
INSERT INTO `t1` VALUES
(1);
```

- *-schema-view.sql、*-schema-trigger.sql、*-schema-post.sql：其他导出文件

13.8.1.6 导出到 Amazon S3 云盘

Dumpling 在 v4.0.8 及更新版本支持导出到 Amazon S3 云盘。如果需要将数据备份到 Amazon S3 后端存储，那么需要在 -o 参数中指定 Amazon S3 的存储路径。

可以参照 AWS 官方文档 - 如何创建 S3 存储桶在指定的 Region 区域中创建一个 S3 桶 Bucket。如有需要，还可以参照 AWS 官方文档 - 创建文件夹在 Bucket 中创建一个文件夹 Folder。

将有权限访问该 Amazon S3 后端存储的账号的 SecretKey 和 AccessKey 作为环境变量传入 Dumpling 节点。

```
export AWS_ACCESS_KEY_ID=${AccessKey}
export AWS_SECRET_ACCESS_KEY=${SecretKey}
```

Dumpling 同时还支持从 ~/.aws/credentials 读取凭证文件。Dumpling 导出到 Amazon S3 的配置参数与 BR 大致相同，更多参数描述，请参考外部存储 URI 格式。

```
./dumpling -u root -P 4000 -h 127.0.0.1 -r 200000 -o "s3://${Bucket}/${Folder}"
```

13.8.1.7 筛选导出的数据

13.8.1.7.1 使用 --where 选项筛选数据

默认情况下，Dumpling 会导出排除系统数据库（包括 mysql、sys、INFORMATION_SCHEMA、PERFORMANCE_SCHEMA、METRICS_SCHEMA 和 INSPECTION_SCHEMA）外所有其他数据库。你可以使用 --where <SQL where expression> 来指定要导出的记录。

```
./dumpling -u root -P 4000 -h 127.0.0.1 -o /tmp/test --where "id < 100"
```

上述命令将会导出各个表的 id < 100 的数据。注意 --where 参数无法与 --sql 一起使用。

13.8.1.7.2 使用 --filter 选项筛选数据

Dumpling 可以通过 --filter 指定 table-filter 来筛选特定的库表。table-filter 的语法与 .gitignore 相似，详细语法参考表库过滤。

```
./dumpling -u root -P 4000 -h 127.0.0.1 -o /tmp/test -r 200000 --filter "employees.*" --filter
    ↪ "*.WorkOrder"
```

上述命令将会导出 employees 数据库的所有表，以及所有数据库中的 WorkOrder 表。

13.8.1.7.3 使用 -B 或 -T 选项筛选数据

Dumpling 也可以通过 -B 或 -T 选项导出特定的数据库/数据表。

> 注意：
>
> - --filter 选项与 -T 选项不可同时使用。
> - -T 选项只能接受完整的 库名.表名 形式，不支持只指定表名。例：Dumpling 无法识别 -T WorkOrder。

例如通过指定：

- -B employees 导出 employees 数据库
- -T employees.WorkOrder 导出 employees.WorkOrder 数据表

13.8.1.8 通过并发提高 Dumpling 的导出效率

默认情况下，导出的文件会存储到 ./export-<current local time> 目录下。常用选项如下：

- -t 用于指定导出的线程数。增加线程数会增加 Dumpling 并发度提高导出速度，但也会加大数据库内存消耗，因此不宜设置过大。
- -r 选项用于指定单个文件的最大记录数，或者说，数据库中的行数。开启后 Dumpling 会开启表内并发，提高导出大表的速度。当上游为 TiDB 且版本为 v3.0 或更新版本时，设置 -r 参数大于 0 表示使用 TiDB region 信息划分表内并发，具体取值不影响划分算法。对上游为 MySQL 且表的主键是 int 的场景，该参数也有表内并发效果。
- --compress <format> 选项可以用于压缩导出的数据，支持 gzip、snappy、zstd 压缩算法。压缩可以显著降低导出数据的大小，同时如果存储的写入 I/O 带宽不足，可以使用该选项来加速导出。但该选项也有副作用，由于该选项会对每个文件单独压缩，因此会增加 CPU 消耗。

利用以上选项可以提高 Dumpling 的导出速度。

13.8.1.9 调整 Dumpling 的数据一致性选项

> 注意：
>
> 数据一致性选项的默认值为 auto。在大多数场景下，你不需要调整该选项。

Dumpling 通过 --consistency <consistency level> 标志控制导出数据“一致性保证”的方式。在使用 snapshot 来保证一致性的时候，可以使用 --snapshot 选项指定要备份的时间戳。还可以使用以下的一致性级别：

- flush：使用 FLUSH TABLES WITH READ LOCK 短暂地中断备份库的 DML 和 DDL 操作、保证备份连接的全局一致性和记录 POS 信息。所有的备份连接启动事务后释放该锁。推荐在业务低峰或者 MySQL 备份库上进行全量备份。
- snapshot：获取指定时间戳的一致性快照并导出。
- lock：为待导出的所有表上读锁。
- none：不做任何一致性保证。
- auto：对 MySQL 使用 flush，对 TiDB 使用 snapshot。

操作完成之后，你可以在 /tmp/test 查看导出的文件：

```
$ ls -lh /tmp/test | awk '{print $5 "\t" $9}'

140B  metadata
66B   test-schema-create.sql
300B  test.sbtest1-schema.sql
190K  test.sbtest1.0.sql
300B  test.sbtest2-schema.sql
190K  test.sbtest2.0.sql
300B  test.sbtest3-schema.sql
190K  test.sbtest3.0.sql
```

13.8.1.10 导出 TiDB 的历史数据快照

Dumping 可以通过 --snapshot 指定导出某个tidb_snapshot 的数据。

--snapshot 选项可设为 TSO（SHOW MASTER STATUS 输出的 Position 字段）或有效的 datetime 时间（YYYY-MM-DD hh:mm:ss 形式），例如：

```
./dumpling --snapshot 417773951312461825
./dumpling --snapshot "2020-07-02 17:12:45"
```

即可导出 TSO 为 417773951312461825 或 2020-07-02 17:12:45 时的 TiDB 历史数据快照。

13.8.1.11 控制导出 TiDB 大表（超过 1 TB）时的内存使用

Dumping 导出 TiDB 较大单表（超过 1 TB）时，可能会因为导出数据过大导致 TiDB 内存溢出 (OOM)，从而使连接中断导出失败。可以通过以下参数减少 TiDB 的内存使用。

- 设置 -r 参数，可以划分导出数据区块减少 TiDB 扫描数据的内存开销，同时也可开启表内并发提高导出效率。当上游为 TiDB 且版本为 v3.0 或更新版本时，设置 -r 参数大于 0 表示使用 TiDB region 信息划分表内并发，具体取值不影响划分算法。
- 调小 --tidb-mem-quota-query 参数到 8589934592 (8GB) 或更小。可控制 TiDB 单条查询语句的内存使用。
- 调整 --params "tidb_distsql_scan_concurrency=5" 参数，即设置导出时的 session 变量tidb_distsql_scan_concurrency 从而减少 TiDB scan 操作的并发度。

13.8.1.12 手动设置 TiDB GC 时间

如果导出的 TiDB 版本为 v4.0.0 或更新版本，并且 Dumpling 可以访问 TiDB 集群的 PD 地址，Dumpling 会自动配置延长 GC 时间且不会对原集群造成影响。

但是，在以下场景中，Dumpling 无法自动调整 GC 时间：

- 数据量非常大（超过 1 TB）。
- Dumpling 无法直接连接到 PD，例如 TiDB 集群运行在 TiDB Cloud 上，或者 TiDB 集群运行在 Kubernetes 上且与 Dumpling 分离。

在这些场景中，你必须提前手动调长 GC 时间，以避免因为导出过程中发生 GC 导致导出失败。

使用以下 SQL 语句手动调整 GC 时间：

```
SET GLOBAL tidb_gc_life_time = '720h';
```

在 Dumpling 退出后，无论导出是否成功，都必须将 GC 时间恢复为其原始值（默认值为 `10m`）。

```
SET GLOBAL tidb_gc_life_time = '10m';
```

13.8.2 Dumpling 主要选项表

主要选项	用途	默认值
-V 或 –version	输出 Dumpling 版本并直接退出	
-B 或 –database	导出指定数据库	
-T 或 –tables-list	导出指定数据表	

主要选项	用途	默认值
-f 或 –filter	导出能匹配模式的表，语法可参考table-filter	[*.*,!/^(↪ mysql ↪ | ↪ sys ↪ | ↪ INFORMATION_SCHEMA ↪ | ↪ PERFORMANCE_SCHEMA ↪ | ↪ METRICS_SCHEMA ↪ | ↪ INSPECTION_SCHEMA ↪) ↪ $ ↪ /.*] ↪ (导出除系统库外的所有库表)
–case-sensitive	table-filter 是否大小写敏感	false，大小写不敏感
-h 或 –host	连接的数据库主机的地址	“127.0.0.1”
-t 或 –threads	备份并发线程数	4

主要选项	用途	默认值
-r 或 –rows	用于开启表内并发加速导出。默认值是 0，表示不开启。取值大于 0 表示开启，取值是 INT 类型。当数据源为 TiDB 时，设置 -r 参数大于 0 表示使用 TiDB region 信息划分区间，同时减少内存使用。具体取值不影响划分算法。对数据源	

主要选项	用途	默认值
-L 或 –logfile	日志输出地址，为空时会输出到控制台	“”
–loglevel	日志级别 {debug,info,warn,error,dpanic,panic,fatal}	“info”
–logfmt	日志输出格式 {text,json}	“text”
-d 或 –no-data	不导出数据，适用于只导出 schema 场景	
–no-header	导出 csv 格式的 table 数据，不生成 header	
-W 或 –no-views	不导出 view	true
-m 或 –no-schemas	不导出 schema，只导出数据	

主要选项	用途	默认值
-s 或 –statement-size	控制 INSERT SQL 语句的大小，单位 bytes	
-F 或 –filesize	将 table 数据划分出来的文件大小，需指明单位 (如 128B, 64 KiB , 32 MiB , 1.5 GiB)	
–filetype	导出文件类型 (csv/sql)	“sql”

主要选项	用途	默认值
-o 或 –output	导出本地文件路径或外部存储 URI 格式	“./export-${time}”
-S 或 –sql	根据指定的 sql 导出数据，该选项不支持并发导出	

主要选项	用途	默认值
–consistency	flush: dump 前用 FTWRL snap-shot: 通过 TSO 来指定 dump 某个快照时间点的 TiDB 数据 lock: 对需要 dump 的所有表执行 `lock tables read` 命令 none: 不加锁 dump，无法保证一致性 auto: 对 MySQL 使用 –consistency flush；对 TiDB	“auto”

主要选项	用途	默认值
–snapshot	snapshot tso，只在 con- sis- tency=snapshot 下生效	
–where	对备份的数据表通过 where 条件指定范围	
-p 或 –password	连接的数据库主机的密码	
-P 或 –port	连接的数据库主机的端口	4000
-u 或 –user	连接的数据库主机的用户名	“root”
–dump-empty-database	导出空数据库的建库语句	true

主要选项	用途	默认值
–ca	用于 TLS 连接的 certifi-cate au-thor-ity 文件的地址	
–cert	用于 TLS 连接的 client certifi-cate 文件的地址	
–key	用于 TLS 连接的 client pri-vate key 文件的地址	
–csv-delimiter	csv 文件中字符类型变量的定界符	’ “’

主要选项	用途	默认值
–csv-separator	csv 文件中各值的分隔符，如果数据中可能有逗号，建议源文件导出时分隔符使用非常见组合字符	‘,’
–csv-null-value	csv 文件空值的表示	“\N”
–escape-backslash	使用反斜杠 (\) 来转义导出文件中的特殊字符	true

主要选项	用途	默认值
–output-filename-template	以 golang tem-plate 格式表示的数据文件名格式支持 {{. ↪ DB ↪ }} ↪ 、 {{. ↪ Table ↪ }} ↪ 、 {{. ↪ Index ↪ }} ↪ 三个参数分别表示数据文件的库名、表名、分块 ID	‘{{.DB}}.{{.Table}}.{{.Index}}’

主要选项	用途	默认值
–status-addr	Dumpling 的服务地址，包含了 Prometheus 拉取 metrics 信息及 pprof 调试的地址	“:8281”

主要选项	用途	默认值
–tidb-mem-quota-query	单条 dumpling 命令导出 SQL 语句的内存限制，单位为 byte。对于 v4.0.10 或以上版本，若不设置该参数，默认使用 TiDB 中的 mem-quota-query 配置项值作为内存限制值。对于 v4.0.10 以下版本，该参数值默认为 32	34359738368

主要选项	用途	默认值
–params	为需导出的数据库连接指定 ses-sion 变量，可接受的格式: “char-ac-ter_set_client=latin1,character_set_connection=latin1”	
-c 或 –compress	压缩 Dumpling 导出的 CSV、SQL 数据与表结构文件为指定格式，支持 “gzip”、“snappy” 和 “zstd” 压缩算法	“”

13.9 TiCDC

13.9.1 TiCDC 简介

TiCDC 是一款 TiDB 增量数据同步工具，通过拉取上游 TiKV 的数据变更日志，TiCDC 可以将数据解析为有序的行级变更数据输出到下游。

13.9.1.1 TiCDC 适用场景

- 提供多 TiDB 集群，跨区域数据高可用和容灾方案，保证在灾难发生时保证主备集群数据的最终一致性。
- 提供同步实时变更数据到异构系统的服务，为监控、缓存、全文索引、数据分析、异构数据库使用等场景提供数据源。

13.9.1.2 TiCDC 主要特性

13.9.1.2.1 核心能力

- 提供 TiDB -> TiDB 之间数据容灾复制的能力，实现秒级别 RPO 和分钟级别 RTO
- 提供 TiDB 之间双向复制的能力，支持通过 TiCDC 构建多写多活的 TiDB 集群
- 提供 TiDB -> MySQL（或其他兼容 MySQL 协议的数据库）的低延迟的增量数据同步能力
- 提供 TiDB -> Kafka 增量数据同步能力，推荐的数据格式包含Canal-JSON，Avro 等
- 提供表级别数据同步能力，支持同步过程中过滤数据库、表、DML、DDL 的能力
- 高可用架构，无单点故障；支持动态添加、删除 TiCDC 节点
- 支持通过Open API 进行集群管理，包括查询任务状态；动态修改任务配置；动态创建、删除任务等

13.9.1.2.2 数据同步顺序性

- TiCDC 对于所有的 DDL/DML 都能对外输出至少一次。
- TiCDC 在 TiKV/TiCDC 集群故障期间可能会重复发相同的 DDL/DML。对于重复的 DDL/DML：
  - MySQL sink 可以重复执行 DDL，对于在下游可重入的 DDL（譬如 truncate table）直接执行成功；对于在下游不可重入的 DDL（譬如 create table），执行失败，TiCDC 会忽略错误继续同步。
  - Kafka sink
    * Kafka sink 提供不同的数据分发策略，可以按照表、主键或 ts 等策略分发数据到不同 Kafka partition。使用表、主键分发策略，可以保证某一行的更新数据被顺序的发送到相同 partition。
    * 对所有的分发策略，我们都会定期发送 Resolved TS 消息到所有的 topic/partition，表示早于该 Resolved TS 的消息都已经发送到 topic/partition，消费程序可以利用 Resolved TS 对多个 topic/partition 的消息进行排序。
    * Kafka sink 会发送重复的消息，但重复消息不会破坏 Resolved TS 的约束，比如在 changefeed 暂停重启后，可能会按顺序发送 msg1、msg2、msg3、msg2、msg3。你可以在 Kafka 消费端进行过滤。

13.9.1.2.3 数据同步一致性

- MySQL sink
  - TiCDC 开启 redo log 后保证数据复制的最终一致性
  - TiCDC 保证单行的更新与上游更新顺序一致。
  - TiCDC 不保证下游事务的执行顺序和上游完全一致。

> **注意：**
>
> 从 v6.2 版本起，你可以通过配置 sink uri 参数`transaction-atomicity`来控制 TiCDC 是否拆分单表事务。拆分事务可以大幅降低 MySQL sink 同步大事务的延时和内存消耗。

13.9.1.3 TiCDC 架构

TiCDC 作为 TiDB 的增量数据同步工具，通过 PD 内部的 etcd 实现高可用，通过多个 TiCDC 进程获取 TiKV 节点上的数据改变，在内部进行排序、合并等处理之后，通过多个同步任务 (Changefeed)，同时向多个下游系统进行数据同步。

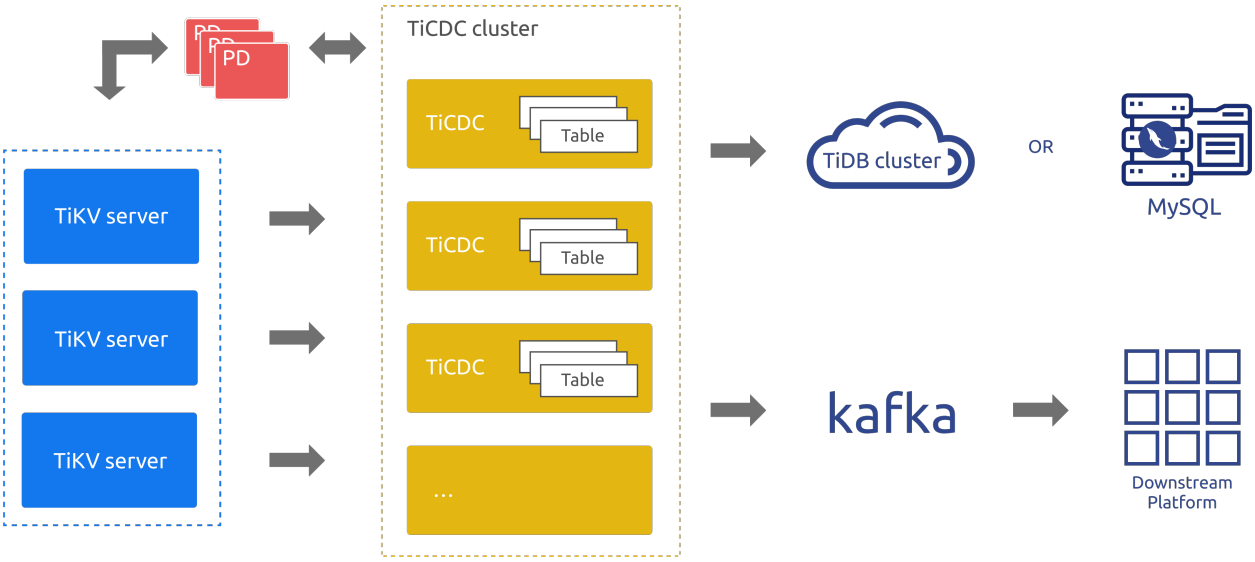


图 262: TiCDC architecture

在以上架构图中：

- TiKV Server：代表 TiDB 集群中的 TiKV 节点，当数据发生改变时 TiKV 节点会主动将发生的数据改变以变更日志（KV change logs，简称 change logs）的方式发送给 TiCDC 节点。当然，当 TiCDC 节点发现收到的 change logs 并不是连续的，也会主动发起请求，获得需要的 change logs。
- TiCDC：代表运行了运行 TiCDC 进程的各个节点。每个节点都运行一个 TiCDC 进程，每个进程会从 TiKV 节点中拉取一个或者多个表中的数据改变，并通过 Sink 模块同步到下游系统。
- PD：代表 TiDB 集群中的调度模块，负责集群数据的事实调度，这个模块通常是由 3 个 PD 节点构成的，内部通过 etcd 集群来实现选举等高可用相关的能力。TiCDC 集群使用了 PD 集群内置的 etcd 集群来保存自己的元数据信息，例如：节点的状态信息，changefeed 配置信息等。

另外，从上面的架构图中也可以看到，目前 TiCDC 支持将数据同步到 TiDB，MySQL 数据库，以及 Kafka 等。

13.9.1.4 最佳实践

- 使用 TiCDC 在两个 TiDB 集群间同步数据时，当上下游的延迟超过 100 ms 时，推荐将 TiCDC 部署在下游 TiDB 集群所在的区域 (IDC, region)。
- TiCDC 同步的表需要至少存在一个有效索引的表，有效索引的定义如下：
  - 主键 (`PRIMARY KEY`) 为有效索引。
  - 唯一索引 (`UNIQUE INDEX`) 中每一列在表结构中明确定义非空 (`NOT NULL`) 且不存在虚拟生成列 (`VIRTUAL GENERATED COLUMNS`)。
- 容灾场景下使用 TiCDC 需要配置redo log 实现最终一致性。
- 使用 TiCDC 同步单行较大 (> 1k) 的宽表时，推荐设置 TiCDC 参数per-table-memory-quota，使得 `per-table-memory-quota = ticdcTotalMemory / (tableCount * 2)`。`ticdcTotalMemory` 是一个 TiCDC 节点的内存，`tableCount` 是一个 TiCDC 节点同步的目标表的数量。

> 注意：
>
> 从 TiCDC v4.0.8 版本开始，可通过修改任务配置来同步没有有效索引的表，但在数据一致性的保证上有所减弱。具体使用方法和注意事项参考同步没有有效索引的表。

13.9.1.4.1 暂不支持的场景

目前 TiCDC 暂不支持的场景如下：

- 暂不支持单独使用 RawKV 的 TiKV 集群。
- 暂不支持在 TiDB 中创建 SEQUENCE 的 DDL 操作和SEQUENCE 函数。在上游 TiDB 使用 SEQUENCE 时，TiCDC 将会忽略掉上游执行的 SEQUENCE DDL 操作/函数，但是使用 SEQUENCE 函数的 DML 操作可以正确地同步。

13.9.2 TiCDC 安装部署与集群运维

本文档介绍部署 TiCDC 集群的软硬件环境要求，如何安装部署 TiCDC 集群，以及如何对 TiCDC 集群进行运维操作。你可以选择在安装 TiDB 集群的同时部署 TiCDC，也可以对原有 TiDB 集群新增 TiCDC 组件。

13.9.2.1 软件和硬件环境推荐配置

在生产环境中，TiCDC 的软件和硬件配置推荐如下：

Linux 操作系统	版本
Red Hat Enterprise Linux	7.3 及以上
CentOS	7.3 及以上

CPU	内存	硬盘类型	网络	TiCDC 集群实例数量（生产环境最低要求）
16 核 +	64 GB+	SSD	万兆网卡（2 块最佳）	2

更多信息参见TiDB 软件和硬件环境建议配置。

13.9.2.2 使用 TiUP 部署包含 TiCDC 组件的全新 TiDB 集群

在使用 TiUP 部署全新 TiDB 集群时，支持同时部署 TiCDC 组件。你需要在 TiUP 启动 TiDB 集群时的配置文件中加入 TiCDC 相关的部分，以下是一个示例：

```
cdc_servers:
  - host: 10.0.1.20
    gc-ttl: 86400
    data_dir: "/cdc-data"
  - host: 10.0.1.21
    gc-ttl: 86400
    data_dir: "/cdc-data"
```

更多参考：

- 详细配置参数，请参考编辑初始化配置文件。
- 具体可配置字段，请参考通过 TiUP 配置 cdc_servers。
- 部署集群的具体步骤，请参考使用 TiUP 部署 TiDB 集群。

> **注意：**
>
> 在安装之前，请确认 TiUP 中控机与 TiCDC 目标主机的SSH 互信及 sudo 免密已经完成配置。

13.9.2.3 使用 TiUP 在原有 TiDB 集群上新增或扩容 TiCDC 组件

扩容的方式与部署 TiCDC 集群的方式类似，推荐使用 TiUP 工具完成。

1. 编写一个名为 scale-out.yml 的配置文件，包含需要扩容的节点的配置信息。下面是一个示例：

```
cdc_servers:
  - host: 10.1.1.1
    gc-ttl: 86400
    data_dir: /tidb-data/cdc-8300
  - host: 10.1.1.2
    gc-ttl: 86400
    data_dir: /tidb-data/cdc-8300
  - host: 10.0.1.4
    gc-ttl: 86400
    data_dir: /tidb-data/cdc-8300
```

2. 在 TiUP 中控机上执行类似下面的命令进行扩容：

```
tiup cluster scale-out <cluster-name> scale-out.yml
```

更多用例说明，请参考扩容 TiCDC 节点。

13.9.2.4 使用 TiUP 在原有 TiDB 集群上移除或缩容 TiCDC 组件

推荐使用 TiUP 完成对 TiCDC 集群节点的缩容。使用类似下面的命令完成缩容:

```
tiup cluster scale-in <cluster-name> --node 10.0.1.4:8300
```

更多用例说明，请参考缩容 TiCDC 节点。

13.9.2.5 使用 TiUP 升级 TiCDC 集群

TiUP 支持升级 TiDB 集群，包括 TiCDC 组件。执行升级指令时，TiUP 会自动升级 TiCDC 组件，无需额外操作。操作示例如下：

```
tiup update --self && \
tiup update --all && \
tiup cluster upgrade <cluster-name> <version> --transfer-timeout 600
```

> 注意：
>
> 命令中的 `<cluster-name>` 需要替换为集群名字，`<version>` 需要替换为目标版本号，例如 v7.0.0。

13.9.2.5.1 升级的注意事项

升级 TiCDC 集群时，需要注意以下事项：

- TiCDC v4.0.2 对 changefeed 的配置做了调整，请参阅配置文件兼容注意事项。
- 升级期间遇到的问题及其解决办法，请参阅使用 TiUP 升级 TiDB。
- TiCDC 自 v6.3.0 起支持滚动升级，使用 TiUP 升级 TiCDC 节点期间，能够保证同步延迟稳定，不发生剧烈波动。满足以下条件将自动启用滚动升级：
  - TiCDC 版本大于等于 v6.3.0。
  - TiUP 版本大于等于 v1.11.3。
  - 集群中至少有两个正在运行的 TiCDC 实例。

13.9.2.6 使用 TiUP 变更 TiCDC 集群配置

本节介绍如何使用 TiUP 的tiup cluster edit-config 命令来修改 TiCDC 的配置。在以下例子中，假设需要把 TiCDC 的 gc-ttl 从默认值 86400 修改为 172800，即 48 小时。

1. 执行 tiup cluster edit-config 命令，注意将 <cluster-name> 替换成实际的集群名：

```
tiup cluster edit-config <cluster-name>
```

2. 在 vi 编辑器页面，修改server-configs 下的 cdc 配置：

```
server_configs:
  tidb: {}
  tikv: {}
  pd: {}
  tiflash: {}
  tiflash-learner: {}
  pump: {}
  drainer: {}
  cdc:
    gc-ttl: 172800
```

   以上把 TiCDC 的 gc-ttl 的值设置为 48 小时。

3. 执行 tiup cluster reload -R cdc 命令重新加载配置。

13.9.2.7 使用 TiUP 终止和启动 TiCDC 节点

使用 TiUP 可以方便地终止和启动 TiCDC 节点，命令如下：

- 终止 TiCDC 节点：tiup cluster stop -R cdc
- 启动 TiCDC 节点：tiup cluster start -R cdc
- 重启 TiCDC 节点：tiup cluster restart -R cdc

13.9.2.8 使用加密传输 (TLS) 功能

请参阅为 TiDB 组件间通信开启加密传输。

13.9.2.9 使用 TiCDC 命令行工具来查看集群状态

执行以下命令来查看 TiCDC 集群运行状态，注意需要将 v<CLUSTER_VERSION> 替换为 TiCDC 集群版本，例如 v6.5.0：

```
tiup ctl:v<CLUSTER_VERSION> cdc capture list --server=http://10.0.10.25:8300
```

```
[
  {
    "id": "806e3a1b-0e31-477f-9dd6-f3f2c570abdd",
```

```
    "is-owner": true,
    "address": "127.0.0.1:8300",
    "cluster-id": "default"
  },
  {
    "id": "ea2a4203-56fe-43a6-b442-7b295f458ebc",
    "is-owner": false,
    "address": "127.0.0.1:8301",
    "cluster-id": "default"
  }
]
```

- `id`：表示服务进程的 ID。
- `is-owner`：表示该服务进程是否为 owner 节点。
- `address`：该服务进程对外提供接口的地址。
- `cluster-id`：该 TiCDC 的集群 ID，默认值为 `default`。

13.9.3 Changefeed

13.9.3.1 Changefeed 概述

Changefeed 是 TiCDC 中的单个同步任务。Changefeed 将一个 TiDB 集群中数张表的变更数据输出到一个指定的下游。TiCDC 集群可以运行和管理多个 Changefeed。

13.9.3.1.1 Changefeed 状态流转

同步任务状态标识了同步任务的运行情况。在 TiCDC 运行过程中，同步任务可能会运行出错、手动暂停、恢复，或达到指定的 `TargetTs`，这些行为都可以导致同步任务状态发生变化。本节描述 TiCDC 同步任务的各状态以及状态之间的流转关系。

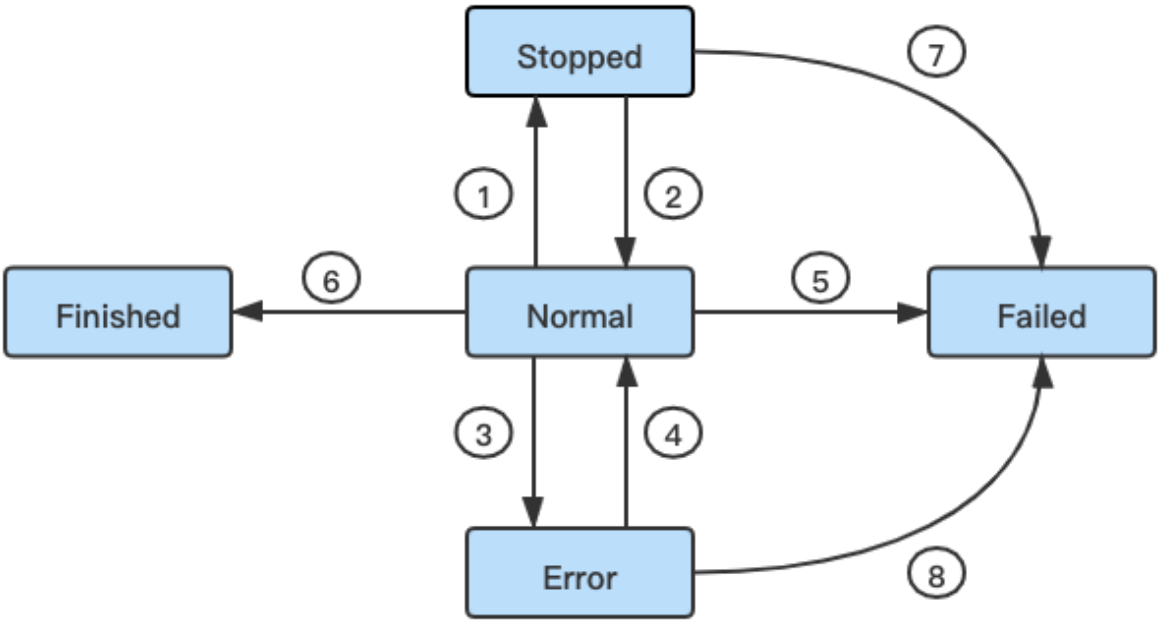


图 263: TiCDC state transfer

以上状态流转图中的状态说明如下：

- Normal：同步任务正常进行，checkpoint-ts 正常推进。
- Stopped：同步任务停止，由于用户手动暂停 (pause) changefeed。处于这个状态的 changefeed 会阻挡 GC 推进。
- Error：同步任务报错，由于某些可恢复的错误导致同步无法继续进行，处于这个状态的 changefeed 会不断尝试继续推进，直到状态转为 Normal。处于这个状态的 changefeed 会阻挡 GC 推进。
- Finished：同步任务完成，同步任务进度已经达到预设的 TargetTs。处于这个状态的 changefeed 不会阻挡 GC 推进。
- Failed：同步任务失败。由于发生了某些不可恢复的错误，导致同步无法继续进行，并且无法恢复。TiCDC 将为发生故障的 changefeed 保留数据 24 小时，以防止其在 GC 过程中被清除。

以上状态流转图中的编号说明如下：

- ① 执行 `changefeed pause` 命令。
- ② 执行 `changefeed resume` 恢复同步任务。
- ③ changefeed 运行过程中发生可恢复的错误，自动进行恢复。
- ④ 执行 `changefeed resume` 恢复同步任务。
- ⑤ changefeed 运行过程中发生不可恢复的错误。
- ⑥ changefeed 已经进行到预设的 TargetTs，同步自动停止。
- ⑦ changefeed 停滞时间超过 `gc-ttl` 所指定的时长，不可被恢复。
- ⑧ changefeed 尝试自动恢复过程中发生不可恢复的错误。

13.9.3.1.2 操作 Changefeed

通过 TiCDC 提供的命令行工具 `cdc cli`，你可以管理 TiCDC 集群和同步任务，具体可参考管理 TiCDC Changefeed。你也可以通过 HTTP 接口，即 TiCDC OpenAPI 来管理 TiCDC 集群和同步任务，详见TiCDC OpenAPI。

如果你使用的 TiCDC 是用 TiUP 部署的，可以通过 `tiup ctl:v<CLUSTER_VERSION> cdc` 来使用 TiCDC 命令行工具，注意需要将 `<version>` 替换为 TiCDC 集群版本。你也可以通过直接执行 `cdc cli` 直接使用命令行工具。

13.9.3.2 创建 Changefeed

13.9.3.2.1 同步数据到 MySQL 兼容数据库

本文介绍如何使用 TiCDC 创建一个将增量数据复制到下游 TiDB 数据库，或其他兼容 MySQL 协议数据库的 Changefeed。同时介绍了如何使用 TiCDC 灾难场景的最终一致性复制功能。

创建同步任务，复制增量数据到 MySQL 兼容数据库

使用以下命令来创建同步任务：

```
cdc cli changefeed create \
    --server=http://10.0.10.25:8300 \
    --sink-uri="mysql://root:123456@127.0.0.1:3306/" \
    --changefeed-id="simple-replication-task"
```

```
Create changefeed successfully!
ID: simple-replication-task
Info: {"sink-uri":"mysql://root:123456@127.0.0.1:3306/","opts":{},"create-time":"2020-03-12T22
    ↪ :04:08.103600025+08:00","start-ts":415241823337054209,"target-ts":0,"admin-job-type":0,"
    ↪ sort-engine":"unified","sort-dir":".","config":{"case-sensitive":true,"filter":{"rules
    ↪ ":["*.*"],"ignore-txn-start-ts":null,"ddl-allow-list":null},"mounter":{"worker-num":16},"
    ↪ sink":{"dispatchers":null},"scheduler":{"type":"table-number","polling-time":-1}},"state
    ↪ ":"normal","history":null,"error":null}
```

- `--server`：TiCDC 集群中任意一个 TiCDC 服务器的地址。
- `--changefeed-id`：同步任务的 ID，格式需要符合正则表达式 `^[a-zA-Z0-9]+(\-[a-zA-Z0-9]+)*$`。如果不指定该 ID，TiCDC 会自动生成一个 UUID（version 4 格式）作为 ID。
- `--sink-uri`：同步任务下游的地址，详见Sink URI 配置 `mysql/tidb`。
- `--start-ts`：指定 changefeed 的开始 TSO。TiCDC 集群将从这个 TSO 开始拉取数据。默认为当前时间。
- `--target-ts`：指定 changefeed 的目标 TSO。TiCDC 集群拉取数据直到这个 TSO 停止。默认为空，即 TiCDC 不会自动停止。
- `--config`：指定 changefeed 配置文件，详见：TiCDC Changefeed 配置参数。

Sink URI 配置 `mysql/tidb`

Sink URI 用于指定 TiCDC 目标系统的连接信息，遵循以下格式：

```
[scheme]://[userinfo@][host]:[port][/path]?[query_parameters]
```

> **注意：**
>
> `/path` 不适用于 MySQL sink。

一个通用的配置样例如下所示：

```
--sink-uri="mysql://root:123456@127.0.0.1:3306"
```

URI 中可配置的参数如下：

参数	描述
`root`	下游数据库的用户名。
`123456`	下游数据库密码。（可采用 Base64 进行编码）
`127.0.0.1`	下游数据库的 IP。
`3306`	下游数据的连接端口。
`worker-count`	向下游执行 SQL 的并发度（可选，默认值为 16）。
`cache-prep-stmts`	向下游执行 SQL 时是否使用 prepared statement 并且开启客户端的 prepared statement 缓存（可选，默认值为 `true`）。
`max-txn-row`	向下游执行 SQL 的 batch 大小（可选，默认值为 256）。
`ssl-ca`	连接下游 MySQL 实例所需的 CA 证书文件路径（可选）。
`ssl-cert`	连接下游 MySQL 实例所需的证书文件路径（可选）。
`ssl-key`	连接下游 MySQL 实例所需的证书密钥文件路径（可选）。
`time-zone`	连接下游 MySQL 实例时使用的时区名称，从 v4.0.8 开始生效。（可选。如果不指定该参数，使用 TiCDC 服务进程的时区；如果指定该参数但使用空值，则表示连接 MySQL 时不指定时区，使用下游默认时区）。
`transaction-atomicity`	指定事务的原子性级别（可选，默认值为 none）。当该值为 `table` 时 TiCDC 保证单表事务的原子性，当该值为 none 时 TiCDC 会拆分单表事务。

若需要对 Sink URI 中的数据库密码使用 Base64 进行编码，可以参考如下命令：

```
echo -n '123456' | base64   # 假设待编码的密码为 123456
```

编码后的密码如下：

```
MTIzNDU2
```

> **注意：**
>
> 当 Sink URI 中包含特殊字符时，如 `! * ' ( ) ; : @ & = + $ , / ? % # [ ]`，需要对 URI 特殊字符进行转义处理。你可以使用 URI Encoder 工具对 URI 进行转义。

灾难场景的最终一致性复制

从 v6.1.1 版本开始，容灾场景下的最终一致性复制功能 GA。从 v5.3.0 开始，TiCDC 支持将上游 TiDB 的增量数据备份到下游集群的对象存储或 NFS 文件系统。当上游集群出现了灾难，完全无法使用时，TiCDC 可以将下游集群恢复到最近的一致状态，即提供灾备场景的最终一致性复制能力，确保应用可以快速切换到下游集群，避免数据库长时间不可用，提高业务连续性。

目前，TiCDC 支持将 TiDB 集群的增量数据复制到 TiDB 或兼容 MySQL 的数据库系统（包括 Aurora、MySQL 和 MariaDB）。如果 TiCDC 在上游发生灾难前正常运行，且上游 TiDB 集群没有出现数据复制延迟大幅度增加的情况，灾难发生后，下游集群可以在 5 分钟之内恢复集群，并且最多丢失出现问题前 10 秒钟的数据，即 RTO <= 5 min，P95 RPO <= 10s。

当上游 TiDB 集群出现以下情况时，会导致 TiCDC 延迟上升，进而影响 RPO：

- TPS 短时间内大幅度上升
- 上游出现大事务或者长事务
- Reload 或 Upgrade 上游 TiKV 集群或 TiCDC 集群
- 执行耗时很长的 DDL 语句，例如：add index
- 使用过于激进的 PD 调度策略，导致频繁 region leader 迁移或 region merge/split

> 注意：
>
> TiCDC 最终一致性功能从 v6.1.1 开始支持兼容 Amazon S3 协议的对象存储，从 v6.1.4 开始支持兼容 GCS 和 Azure 协议的对象存储。

使用前提

- 准备高可用的对象存储或 NFS 系统，用于存储 TiCDC 的实时增量数据备份文件，在上游发生灾难情况下，该文件存储可以访问。
- TiCDC 对需要具备灾难场景最终一致性的 changefeed 开启该功能，开启方式是在 changefeed 配置文件中增加以下配置：

```
[consistent]
##### 一致性级别，选项有：
##### - none： 默认值，非灾难场景，只有在任务指定 finished-ts 情况下保证最终一致性。
##### - eventual： 使用 redo log，提供上游灾难情况下的最终一致性。
level = "eventual"

##### 单个 redo log 文件大小，单位 MiB，默认值 64，建议该值不超过 128。
max-log-size = 64

##### 刷新或上传 redo log 至 S3 的间隔，单位毫秒，建议该参数 >= 2000。
flush-interval = 2000
```

```
##### redo log 备份文件的地址，支持的 scheme 包括 nfs（NFS 目录）和 Amazon S3、GCS 和 Azure（
    ↪ 上传至对象存储）。
storage = "$SCHEME://logbucket/test-changefeed?endpoint=http://$ENDPOINT/"
```

灾难恢复

当上游发生灾难后，需要通过 cdc redo 命令在下游手动恢复。恢复流程如下：

1. 确保 TiCDC 进程已经退出，防止在数据恢复过程中上游恢复服务，TiCDC 重新开始同步数据。
2. 使用 cdc binary 进行数据恢复，具体命令如下：

```
cdc redo apply --tmp-dir="/tmp/cdc/redo/apply" \
    --storage="s3://logbucket/test-changefeed?endpoint=http://10.0.10.25:24927/" \
    --sink-uri="mysql://normal:123456@10.0.10.55:3306/"
```

以上命令中：

- tmp-dir：指定用于下载 TiCDC 增量数据备份文件的临时目录。
- storage：指定存储 TiCDC 增量数据备份文件的地址，为对象存储 URI 或者 NFS 目录。
- sink-uri：数据恢复的目标地址。scheme 仅支持 mysql。

13.9.3.2.2 同步数据到 Kafka

本文介绍如何使用 TiCDC 创建一个将增量数据复制到 Kafka 的 Changefeed。

创建同步任务，复制增量数据 Kafka

使用以下命令来创建同步任务：

```
cdc cli changefeed create \
    --server=http://10.0.10.25:8300 \
    --sink-uri="kafka://127.0.0.1:9092/topic-name?protocol=canal-json&kafka-version=2.4.0&
        ↪ partition-num=6&max-message-bytes=67108864&replication-factor=1" \
    --changefeed-id="simple-replication-task"
```

```
Create changefeed successfully!
ID: simple-replication-task
Info: {"sink-uri":"kafka://127.0.0.1:9092/topic-name?protocol=canal-json&kafka-version=2.4.0&
    ↪ partition-num=6&max-message-bytes=67108864&replication-factor=1","opts":{},"create-time
    ↪ ":"2020-03-12T22:04:08.103600025+08:00","start-ts":415241823337054209,"target-ts":0,"
    ↪ admin-job-type":0,"sort-engine":"unified","sort-dir":".","config":{"case-sensitive":true
    ↪ ,"filter":{"rules":["*.*"],"ignore-txn-start-ts":null,"ddl-allow-list":null},"mounter":{"
    ↪ worker-num":16},"sink":{"dispatchers":null},"scheduler":{"type":"table-number","polling-
    ↪ time":-1}},"state":"normal","history":null,"error":null}
```

- --server：TiCDC 集群中任意一个 TiCDC 服务器的地址。

- `--changefeed-id`：同步任务的 ID，格式需要符合正则表达式 `^[a-zA-Z0-9]+(\-[a-zA-Z0-9]+)*$`。如果不指定该 ID，TiCDC 会自动生成一个 UUID（version 4 格式）作为 ID。
- `--sink-uri`：同步任务下游的地址，详见：Sink URI 配置 Kafka。
- `--start-ts`：指定 changefeed 的开始 TSO。TiCDC 集群将从这个 TSO 开始拉取数据。默认为当前时间。
- `--target-ts`：指定 changefeed 的目标 TSO。TiCDC 集群拉取数据直到这个 TSO 停止。默认为空，即 TiCDC 不会自动停止。
- `--config`：指定 changefeed 配置文件，详见：TiCDC Changefeed 配置参数。

Sink URI 配置 kafka

Sink URI 用于指定 TiCDC 目标系统的连接信息，遵循以下格式：

```
[scheme]://[userinfo@][host]:[port][/path]?[query_parameters]
```

一个通用的配置样例如下所示：

```
--sink-uri="kafka://127.0.0.1:9092/topic-name?protocol=canal-json&kafka-version=2.4.0&partition-
    ↪ num=6&max-message-bytes=67108864&replication-factor=1"
```

URI 中可配置的的参数如下：

参数	描述
`127.0.0.1`	下游 Kafka 对外提供服务的 IP。
`9092`	下游 Kafka 的连接端口。
`topic-name`	变量，使用的 Kafka topic 名字。
`kafka-version`	下游 Kafka 版本号（可选，默认值 `2.4.0`，目前支持的最低版本为 `0.11.0.2`，最高版本为 `3.2.0`。该值需要与下游 Kafka 的实际版本保持一致）。
`kafka-client-id`	指定同步任务的 Kafka 客户端的 ID（可选，默认值为 `TiCDC_sarama_producer_`同步任务的 `ID`）。
`partition-num`	下游 Kafka partition 数量（可选，不能大于实际 partition 数量，否则创建同步任务会失败，默认值 `3`）。
`max-message-bytes`	每次向 Kafka broker 发送消息的最大数据量（可选，默认值 `10MB`）。从 v5.0.6 和 v4.0.6 开始，默认值分别从 64MB 和 256MB 调整至 10 MB。
`replication-factor`	Kafka 消息保存副本数（可选，默认值 `1`）。
`required-acks`	在 `Produce` 请求中使用的配置项，用于告知 broker 需要收到多少副本确认后才进行响应。可选值有：`0`（`NoResponse`：不发送任何响应，只有 TCP ACK），`1`（`WaitForLocal`：仅等待本地提交成功后再响应）和 `-1`（`WaitForAll`：等待所有同步副本提交后再响应。最小同步副本数量可通过 broker 的 `min.insync.replicas` 配置项进行配置）。（可选，默认值为 `-1`）。
`compression`	设置发送消息时使用的压缩算法（可选值为 `none`、`lz4`、`gzip`、`snappy` 和 `zstd`，默认值为 `none`）。
`protocol`	输出到 Kafka 的消息协议，可选值有 `canal-json`、`open-protocol`、`canal`、`avro`、`maxwell`。
`auto-create-topic`	当传入的 `topic-name` 在 Kafka 集群不存在时，TiCDC 是否要自动创建该 topic（可选，默认值 `true`）。

参数	描述
enable-tidb-extension	可选，默认值是 false。当输出协议为 canal-json 时，如果该值为 true，TiCDC 会发送WATERMARK 事件，并在 Kafka 消息中添加 TiDB 扩展字段。从 6.1.0 开始，该参数也可以和输出协议 avro 一起使用。如果该值为 true，TiCDC 会在 Kafka 消息中添加三个 TiDB 扩展字段。
max-batch-size	从 v4.0.9 开始引入。当消息协议支持把多条变更记录输出至一条 Kafka 消息时，该参数用于指定这一条 Kafka 消息中变更记录的最多数量。目前，仅当 Kafka 消息的 protocol 为 open-protocol 时有效（可选，默认值 16）。
enable-tls	连接下游 Kafka 实例是否使用 TLS（可选，默认值 false）。
ca	连接下游 Kafka 实例所需的 CA 证书文件路径（可选）。
cert	连接下游 Kafka 实例所需的证书文件路径（可选）。
key	连接下游 Kafka 实例所需的证书密钥文件路径（可选）。
sasl-user	连接下游 Kafka 实例所需的 SASL/PLAIN 或 SASL/SCRAM 认证的用户名（authcid）（可选）。
sasl-password	连接下游 Kafka 实例所需的 SASL/PLAIN 或 SASL/SCRAM 认证的密码（可选）。如有特殊字符，需要用 URL encode 转义。
sasl-mechanism	连接下游 Kafka 实例所需的 SASL 认证方式的名称，可选值有 plain、scram-sha-256、scram-sha-512 和 gssapi。
sasl-gssapi-auth-type	gssapi 认证类型，可选值有 user 和 keytab（可选）。
sasl-gssapi-keytab-↪ path	gssapi keytab 路径（可选）。
sasl-gssapi-kerberos-↪ config-path	gssapi kerberos 配置路径（可选）。
sasl-gssapi-service-↪ name	gssapi 服务名称（可选）。
sasl-gssapi-user	gssapi 认证使用的用户名（可选）。
sasl-gssapi-password	gssapi 认证使用的密码（可选）。如有特殊字符，需要用 URL encode 转义。
sasl-gssapi-realm	gssapi realm 名称（可选）。
sasl-gssapi-disable-↪ pafxfast	gssapi 是否禁用 PA-FX-FAST（可选）。
dial-timeout	和下游 Kafka 建立连接的超时时长，默认值为 10s。
read-timeout	读取下游 Kafka 返回的 response 的超时时长，默认值为 10s。
write-timeout	向下游 Kafka 发送 request 的超时时长，默认值为 10s。
avro-decimal-handling-↪ mode	仅在输出协议是 avro 时有效。该参数决定了如何处理 DECIMAL 类型的字段，值可以是 string 或 precise，表明映射成字符串还是浮点数。
avro-bigint-unsigned-↪ handling-mode	仅在输出协议是 avro 时有效。该参数决定了如何处理 BIGINT UNSIGNED 类型的字段，值可以是 string 或 long，表明映射成字符串还是 64 位整形。

最佳实践

- TiCDC 推荐用户自行创建 Kafka Topic，你至少需要设置该 Topic 每次向 Kafka broker 发送消息的最大数据量和下游 Kafka partition 的数量。在创建 changefeed 的时候，这两项设置分别对应 max-message-bytes 和 partition-num 参数。
- 如果你在创建 changefeed 时，使用了尚未存在的 Topic，那么 TiCDC 会尝试使用 partition-num 和

replication-factor 参数自行创建 Topic。建议明确指定这两个参数。

- 在大多数情况下，建议使用 canal-json 协议。

> **注意：**
>
> 当 protocol 为 open-protocol 时，TiCDC 会尽量避免产生长度超过 max-message-bytes 的消息。但如果单条数据变更记录需要超过 max-message-bytes 个字节来表示，为了避免静默失败，TiCDC 会试图输出这条消息并在日志中输出 Warning。

TiCDC 使用 Kafka 的认证与授权

使用 Kafka 的 SASL 认证时配置样例如下所示：

- SASL/PLAIN

```
shell --sink-uri="kafka://127.0.0.1:9092/topic-name?kafka-version=2.4.0&sasl-user=alice-user&sasl
↪ -password=alice-secret&sasl-mechanism=plain"
```

- SASL/SCRAM

SCRAM-SHA-256、SCRAM-SHA-512 与 PLAIN 方式类似，只需要将 sasl-mechanism 指定为对应的认证方式即可。

- SASL/GSSAPI

SASL/GSSAPI user 类型认证：

```
shell --sink-uri="kafka://127.0.0.1:9092/topic-name?kafka-version=2.4.0&sasl-mechanism=gssapi&
↪ sasl-gssapi-auth-type=user&sasl-gssapi-kerberos-config-path=/etc/krb5.conf&sasl-gssapi-service
↪ -name=kafka&sasl-gssapi-user=alice/for-kafka&sasl-gssapi-password=alice-secret&sasl-gssapi-
↪ realm=example.com"
```

sasl-gssapi-user 和 sasl-gssapi-realm 的值与 kerberos 中指定的 principle 有关。例如，principle 为 alice/for-
↪ kafka@example.com，则 sasl-gssapi-user 和 sasl-gssapi-realm 的值应该分别指定为 alice/for-kafka 和 example.com。

SASL/GSSAPI keytab 类型认证：

```
shell --sink-uri="kafka://127.0.0.1:9092/topic-name?kafka-version=2.4.0&sasl-mechanism=gssapi
↪ &sasl-gssapi-auth-type=keytab&sasl-gssapi-kerberos-config-path=/etc/krb5.conf&sasl-gssapi-
↪ service-name=kafka&sasl-gssapi-user=alice/for-kafka&sasl-gssapi-keytab-path=/var/lib/secret/
↪ alice.key&sasl-gssapi-realm=example.com"
```

SASL/GSSAPI 认证方式详见 Configuring GSSAPI。

- TLS/SSL 加密

如果 Kafka broker 启用了 TLS/SSL 加密，则需要在 `--sink-uri` 中增加 `enable-tls=true` 参数值。如果需要使用自签名证书，则还需要在 `--sink-uri` 中指定 `ca`、`cert` 跟 `key` 几个参数。

- ACL 授权

TiCDC 能够正常工作所需的最小权限集合如下：

```
- 对 Topic [资源类型](https://docs.confluent.io/platform/current/kafka/authorization.html#
    ↪ resources)的 `Create` 和 `Write` 权限。
- 对 Cluster 资源类型的 `DescribeConfigs` 权限。
```

TiCDC 集成 Kafka Connect (Confluent Platform)

如要使用 Confluent 提供的 data connectors 向关系型或非关系型数据库传输数据，请选择 `avro` 协议，并在 `schema-registry` 中提供 Confluent Schema Registry 的 URL。

配置样例如下所示：

```
--sink-uri="kafka://127.0.0.1:9092/topic-name?&protocol=avro&replication-factor=3" --schema-
    ↪ registry="http://127.0.0.1:8081" --config changefeed_config.toml
```

```
[sink]
dispatchers = [
 {matcher = ['*.*'], topic = "tidb_{schema}_{table}"},
]
```

集成具体步骤详见与 Confluent Cloud 进行数据集成。

自定义 Kafka Sink 的 Topic 和 Partition 的分发规则

Matcher 匹配规则

以上一节示例配置文件中的 dispatchers 配置项为例：

- 对于匹配了 matcher 规则的表，按照对应的 topic 表达式指定的策略进行分发。例如表 test3.aa，按照 topic 表达式 2 分发；表 test5.aa，按照 topic 表达式 3 分发。
- 对于匹配了多个 matcher 规则的表，以靠前的 matcher 对应的 topic 表达式为准。例如表 test1.aa，按照 topic 表达式 1 分发。
- 对于没有匹配任何 matcher 的表，将对应的数据变更事件发送到 –sink-uri 中指定的默认 topic 中。例如表 test10.aa 发送到默认 topic。
- 对于匹配了 matcher 规则但是没有指定 topic 分发器的表，将对应的数据变更发送到 –sink-uri 中指定的默认 topic 中。例如表 test6.aa 发送到默认 topic。

Topic 分发器

Topic 分发器用 topic = “xxx” 来指定，并使用 topic 表达式来实现灵活的 topic 分发策略。topic 的总数建议小于 1000。

Topic 表达式的基本规则为 `[prefix]{schema}[middle][{table}][suffix]`，详细解释如下：

- `prefix`：可选项，代表 Topic Name 的前缀。
- `{schema}`：必选项，用于匹配库名。
- `middle`：可选项，代表库表名之间的分隔符。
- `{table}`：可选项，用于匹配表名。
- `suffix`：可选项，代表 Topic Name 的后缀。

其中 `prefix`、`middle` 以及 `suffix` 仅允许出现大小写字母（`a-z`、`A-Z`）、数字（`0-9`）、点号（`.`）、下划线（`_`）和中划线（`-`）；`{schema}`、`{table}` 均为小写，诸如 `{Schema}` 以及 `{TABLE}` 这样的占位符是无效的。

一些示例如下：

- `matcher = ['test1.table1', 'test2.table2'], topic = "hello_{schema}_{table}"`
  - 对于表 `test1.table1` 对应的数据变更事件，发送到名为 `hello_test1_table1` 的 topic 中
  - 对于表 `test2.table2` 对应的数据变更事件，发送到名为 `hello_test2_table2` 的 topic 中
- `matcher = ['test3.*', 'test4.*'], topic = "hello_{schema}_world"`
  - 对于 `test3` 下的所有表对应的数据变更事件，发送到名为 `hello_test3_world` 的 topic 中
  - 对于 `test4` 下的所有表对应的数据变更事件，发送到名为 `hello_test4_ world` 的 topic 中
- `matcher = ['*.*'], topic = "{schema}_{table}"`
  - 对于 TiCDC 监听的所有表，按照“库名 _ 表名”的规则分别分发到独立的 topic 中；例如对于 `test.account` 表，TiCDC 会将其数据变更日志分发到名为 `test_account` 的 Topic 中。

DDL 事件的分发

库级别 DDL

诸如 `create database`、`drop database` 这类和某一张具体的表无关的 DDL，称之为库级别 DDL。对于库级别 DDL 对应的事件，被发送到 `--sink-uri` 中指定的默认 topic 中。

表级别 DDL

诸如 `alter table`、`create table` 这类和某一张具体的表相关的 DDL，称之为表级别 DDL。对于表级别 DDL 对应的事件，按照 dispatchers 的配置，被发送到相应的 topic 中。

例如，对于 `matcher = ['test.*'], topic = {schema}_{table}` 这样的 dispatchers 配置，DDL 事件分发情况如下：

- 若 DDL 事件中涉及单张表，则将 DDL 事件原样发送到相应的 topic 中。
  - 对于 DDL 事件 `drop table test.table1`，该事件会被发送到名为 `test_table1` 的 topic 中。
- 若 DDL 事件中涉及多张表（`rename table` / `drop table` / `drop view` 都可能涉及多张表），则将单个 DDL 事件拆分为多个发送到相应的 topic 中。
  - 对于 DDL 事件 `rename table test.table1 to test.table10, test.table2 to test.table20`，则将 `rename table test.table1 to test.table10` 的 DDL 事件发送到名为 `test_table1` 的 topic 中，将 `rename table test.table2 to test.table20` 的 DDL 事件发送到名为 `test.table2` 的 topic 中。

Partition 分发器

partition 分发器用 partition = “xxx” 来指定，支持 default、 ts、 index-value、 table 四种 partition 分发器，分发规则如下：

- default：有多个唯一索引（包括主键）时按照 table 模式分发；只有一个唯一索引（或主键）按照 index-value 模式分发；如果开启了 old value 特性，按照 table 分发
- ts：以行变更的 commitTs 做 Hash 计算并进行 event 分发
- index-value：以表的主键或者唯一索引的值做 Hash 计算并进行 event 分发
- table：以表的 schema 名和 table 名做 Hash 计算并进行 event 分发

> 注意：
>
> 从 v6.1 开始，为了明确配置项的含义，用来指定 partition 分发器的配置项由原来的 `dispatcher` 改为 `partition`，`partition` 为 `dispatcher` 的别名。例如，以下两条规则完全等价：
>
> ```
> [sink]
> dispatchers = [
>    {matcher = ['*.*'], dispatcher = "ts"},
>    {matcher = ['*.*'], partition = "ts"},
> ]
> ```
>
> 但是 `dispatcher` 与 `partition` 不能出现在同一条规则中。例如，以下规则非法：
>
> ```
> {matcher = ['*.*'], dispatcher = "ts", partition = "table"},
> ```

横向扩展大单表的负载到多个 TiCDC 节点

该功能可以按照大单表的数据量和每分钟的修改行数将表的同步范围切分为多个，并使各个范围之间所同步的数据量和修改行数基本相同。该功能将这些范围分布到多个 TiCDC 节点进行同步，使得多个 TiCDC 节点可以同时同步大单表。该功能可以解决以下两个问题：

- 单个 TiCDC 节点不能及时同步大单表。
- TiCDC 节点之间资源（CPU、内存等）消耗不均匀。

> 警告：
>
> TiCDC v7.0.0 仅支持在 Kafka 同步任务上开启大单表的横向扩展功能。

配置样例如下所示：

```
[scheduler]
##### 默认值为 "false"，设置为 "true" 以打开该功能。
```

```
enable-table-across-nodes = true
##### 打开该功能后，该功能只对 Region 个数大于 `region-threshold` 值的表生效。
region-threshold = 100000
##### 打开该功能后，该功能会对每分钟修改行数大于 `write-key-threshold` 值的表生效。
##### 注意：
##### * 该参数默认值为 0，代表该功能默认不会按表的修改行数来切分表的同步范围。
##### * 你可以根据集群负载来配置该参数，如 30000，代表当表每分钟的更新行数超过 30000 时，
    ↪ 该功能将会切分表的同步范围。
##### * 当 `region-threshold` 和 `write-key-threshold` 同时配置时，
#####   TiCDC 将优先检查修改行数是否大于 `write-key-threshold`，
#####   如果不超过，则再检查 Region 个数是否大于 `region-threshold`。
write-key-threshold = 30000
```

一个表包含的 Region 个数可用如下 SQL 查询：

```
SELECT COUNT(*) FROM INFORMATION_SCHEMA.TIKV_REGION_STATUS WHERE DB_NAME="database1" AND
    ↪ TABLE_NAME="table1" AND IS_INDEX=0;
```

13.9.3.2.3 同步数据到存储服务

从 TiDB v6.5.0 开始，TiCDC 支持将行变更事件保存至存储服务，如 Amazon S3、GCS、Azure Blob Storage 和 NFS。本文介绍如何使用 TiCDC 创建同步任务 (Changefeed) 将增量数据同步到这类存储服务，并介绍数据的存储方式。具体如下：

- 如何将变更数据同步至存储服务。
- 变更数据如何在存储服务中保存。

同步变更数据至存储服务

使用以下命令来创建同步任务：

```
cdc cli changefeed create \
    --server=http://10.0.10.25:8300 \
    --sink-uri="s3://logbucket/storage_test?protocol=canal-json" \
    --changefeed-id="simple-replication-task"
```

输出结果如下：

```
Info: {"upstream_id":7171388873935111376,"namespace":"default","id":"simple-replication-task","
    ↪ sink_uri":"s3://logbucket/storage_test?protocol=canal-json","create_time":"2022-11-29T18
    ↪ :52:05.566016967+08:00","start_ts":437706850431664129,"engine":"unified","config":{"
    ↪ case_sensitive":true,"enable_old_value":true,"force_replicate":false,"
    ↪ ignore_ineligible_table":false,"check_gc_safe_point":true,"enable_sync_point":false,"
    ↪ sync_point_interval":600000000000,"sync_point_retention":86400000000000,"filter":{"rules
    ↪ ":["*.*"],"event_filters":null},"mounter":{"worker_num":16},"sink":{"protocol":"canal-
    ↪ json","schema_registry":"","csv":{"delimiter":",","quote":"\"","null":"\\N","
    ↪ include_commit_ts":false},"column_selectors":null,"transaction_atomicity":"none","
```

```
    ↪ encoder_concurrency":16,"terminator":"\r\n","date_separator":"none","
    ↪ enable_partition_separator":false},"consistent":{"level":"none","max_log_size":64,"
    ↪ flush_interval":2000,"storage":""}},"state":"normal","creator_version":"v6.5.0-master-
    ↪ dirty"}
```

- `--server`：TiCDC 集群中任意一个 TiCDC 服务器的地址。
- `--changefeed-id`：同步任务的 ID。格式需要符合正则表达式 `^[a-zA-Z0-9]+(\-[a-zA-Z0-9]+)*$`。如果不指定该 ID，TiCDC 会自动生成一个 UUID（version 4 格式）作为 ID。
- `--sink-uri`：同步任务下游的地址。具体可参考配置 Sink URI。
- `--start-ts`：指定 changefeed 的开始 TSO。TiCDC 集群将从这个 TSO 开始拉取数据。默认为当前时间。
- `--target-ts`：指定 changefeed 的目标 TSO。TiCDC 集群拉取数据直到这个 TSO 停止。默认为空，即 TiCDC 不会自动停止。
- `--config`：指定 changefeed 配置文件，详见TiCDC Changefeed 配置参数。

配置 Sink URI

本章节介绍如何在 Sink URI 中配置存储服务 Amazon S3、GCS、Azure Blob Storage 以及 NFS。Sink URI 用于指定 TiCDC 下游系统的连接信息，遵循以下格式：

```
[scheme]://[host]/[path]?[query_parameters]
```

URI 的 `[query_parameters]` 中可配置的参数如下：

参数	描述	默认值	取值范围
`worker-count`	向下游存储服务保存数据变更记录的并发度	`16`	`[1, 512]`
`flush-interval`	向下游存储服务保存数据变更记录的间隔	`5s`	`[2s, 10m]`
`file-size`	单个数据变更文件的字节数超过 `file-size` 时将其保存至存储服务中	`67108864`	`[1048576, 536870912]`
`protocol`	输出到存储服务的消息协议	N/A	`canal-json` 和 `csv`
`enable-tidb-extension`	`protocol` 参数为 `canal-json` 时，如果该值为 `true`，TiCDC 会发送WATERMARK 事件，并在 canal-json 消息中添加TiDB 扩展字段。	`false`	`false` 和 `true`

> **注意：**
>
> `flush-interval` 与 `file-size` 二者只要满足其一就会向下游写入数据变更文件。
>
> `protocol` 是必选配置，如果 TiCDC 在创建 changefeed 时未解析到该配置，将会返回 `CDC:ErrSinkUnknownProtocol` 错误。

配置外部存储

Amazon S3 配置样例如下：

```
--sink-uri="s3://bucket/prefix?protocol=canal-json"
```

GCS 配置样例如下：

```
--sink-uri="gcs://bucket/prefix?protocol=canal-json"
```

Azure Blob Storage 配置样例如下：

```
--sink-uri="azure://bucket/prefix?protocol=canal-json"
```

> **建议：**
>
> Amazon S3、GCS 以及 Azure Blob Storage 的 URI 参数与 BR 中这三种外部存储的 URI 参数相同。详细参数说明请参考BR 备份存储服务的 URI 格式。

配置 NFS

NFS 配置样例如下：

```
--sink-uri="file:///my-directory/prefix?protocol=canal-json"
```

存储路径组织结构

本章节详细介绍数据变更记录、元数据与 DDL 事件的存储路径组织结构。

数据变更记录

数据变更记录将会存储到以下路径：

```
{scheme}://{prefix}/{schema}/{table}/{table-version-separator}/{partition-separator}/{date-
    ↪ separator}/CDC{num}.{extension}
```

- `scheme`：存储服务类型。例如：s3、gcs、azure、file。
- `prefix`：用户指定的父目录。例如：s3://bucket/bbb/ccc。
- `schema`：表所属的库名。例如：s3://bucket/bbb/ccc/test。
- `table`：表名。例如：s3://bucket/bbb/ccc/test/table1。
- `table-version-separator`：将文件路径按照表的版本进行分隔。例如：s3://bucket/bbb/ccc/test/table1/9999。
- `partition-separator`：将文件路径按照表的分区号进行分隔。例如：s3://bucket/bbb/ccc/test/table1/9999/20。
- `date-separator`：将文件路径按照事务提交的日期进行分隔，可选值如下：
  - `none`：不以 `date-separator` 分隔文件路径。例如：`test.table1` 版本号为 9999 的所有文件都存到 `s3://bucket/bbb/ccc/test/table1/9999` 路径下。
  - `year`：以事务提交的年份分隔文件路径。例如：s3://bucket/bbb/ccc/test/table1/9999/2022。
  - `month`：以事务提交的年份和月份分隔文件路径。例如：s3://bucket/bbb/ccc/test/table1/9999/2022-01。
  - `day`：以事务提交的年月日来分隔文件路径。例如：s3://bucket/bbb/ccc/test/table1/9999/2022-01-02。

- num：存储数据变更记录的目录下文件的序号。例如：s3://bucket/bbb/ccc/test/table1/9999/2022-01-02/CDC000005.csv。
- extension：文件的扩展名。v6.5.0 支持 CSV 和 Canal-JSON 格式。

> 注意：
>
> 表的版本可能在以下三种情况下发生变化：
>
> - 发生 DDL 操作后，表的版本为该 DDL 在上游 TiDB 执行结束的 TSO。但是，表版本的变化并不意味着表结构的变化。例如，在表中的某一列添加注释，不会导致 schema.json 文件内容发生变化。
> - 进程重启，表的版本为进程重启时 Changefeed 的 checkpoint TSO。在有很多表的情况下，重启时需要遍历所有目录并找到上一次重启时每张表写入的位置，这样的操作耗时较长。因此，TiCDC 选择在一个以 checkpoint TSO 为版本的新目录下写入数据，而不是在旧版本的目录下继续写入数据。
> - 发生表调度后，表的版本为该表调度到当前节点时 Changefeed 的 checkpoint TSO。

Index 文件

Index 文件用于防止已写入的数据被错误覆盖，与数据变更记录存储在相同路径：

```
{scheme}://{prefix}/{schema}/{table}/{table-version-separator}/{partition-separator}/{date-
    ↪ separator}/CDC.index
```

Index 文件记录了当前目录下所使用到的最大文件名，比如：

```
CDC000005.csv
```

上述内容表明该目录下 CDC000001.csv 到 CDC000004.csv 文件已被占用，当 TiCDC 集群中发生表调度或者节点重启时，新的节点会读取 Index 文件，并判断 CDC000005.csv 是否被占用。如果未被占用，则新节点会从 CDC000005.csv 开始写文件。如果已被占用，则从 CDC000006.csv 开始写文件，这样可防止覆盖其他节点写入的数据。

元数据

元数据信息将会存储到以下路径：

```
{scheme}://{prefix}/metadata
```

元数据信息以 JSON 格式存储到如下的文件中：

```
{
    "checkpoint-ts":433305438660591626
}
```

- checkpoint-ts：commit-ts 小于等于此 checkpoint-ts 的事务都被写入下游存储当中。

DDL 事件

当 DDL 事件引起表的版本变更时，TiCDC 将会切换到新的路径下写入数据变更记录，例如 test.table1 的版本从 9999 变更为 10000 时将会在 s3://bucket/bbb/ccc/test/table1/10000/2022-01-02/CDC000001.csv 路径中写入数据。并且，当 DDL 事件发生时，TiCDC 将生成一个 schema.json 文件存储表结构信息。

表结构信息将会存储到以下路径：

```
{scheme}://{prefix}/{schema}/{table}/{table-version-separator}/schema.json
```

一个示例 schema.json 文件如下：

```
{
    "Table":"table1",
    "Schema":"test",
    "Version":1,
    "TableVersion":10000,
    "Query": "ALTER TABLE test.table1 ADD OfficeLocation blob(20)",
    "TableColumns":[
        {
            "ColumnName":"Id",
            "ColumnType":"INT",
            "ColumnNullable":"false",
            "ColumnIsPk":"true"
        },
        {
            "ColumnName":"LastName",
            "ColumnType":"CHAR",
            "ColumnLength":"20"
        },
        {
            "ColumnName":"FirstName",
            "ColumnType":"VARCHAR",
            "ColumnLength":"30"
        },
        {
            "ColumnName":"HireDate",
            "ColumnType":"DATETIME"
        },
        {
            "ColumnName":"OfficeLocation",
            "ColumnType":"BLOB",
            "ColumnLength":"20"
        }
    ],
    "TableColumnsTotal":"5"
}
```

- `Table`：表名。
- `Schema`：表所属的库名。
- `Version`：Storage sink 协议版本号。
- `TableVersion`：表的版本号。
- `Query`：DDL 语句。
- `TableColumns`：该数组表示表中每一列的详细信息。
  - `ColumnName`：列名。
  - `ColumnType`：该列的类型。详见数据类型。
  - `ColumnLength`：该列的长度。详见数据类型。
  - `ColumnPrecision`：该列的精度。详见数据类型。
  - `ColumnScale`：该列小数位的长度。详见数据类型。
  - `ColumnNullable`：值为 `true` 时表示该列可以含 NULL 值。
  - `ColumnIsPk`：值为 `true` 时表示该列是主键的一部分。
- `TableColumnsTotal`：`TableColumns` 数组的大小。

数据类型

本章节主要介绍 `schema.json` 文件中使用的各种数据类型。数据类型定义为 `T(M[, D])`，详见数据类型概述。

整数类型

TiDB 中整数类型可被定义为 `IT[(M)] [UNSIGNED]`，其中：

- `IT` 为整数类型，包括 `TINYINT`、`SMALLINT`、`MEDIUMINT`、`INT`、`BIGINT` 和 `BIT`。
- `M` 为该类型的显示宽度。

`schema.json` 文件中对整数类型定义如下：

```
{
    "ColumnName":"COL1",
    "ColumnType":"{IT} [UNSIGNED]",
    "ColumnPrecision":"{M}"
}
```

小数类型

TiDB 中的小数类型可被定义为 `DT[(M,D)][UNSIGNED]`，其中：

- `DT` 为小数类型，包括 `FLOAT`、`DOUBLE`、`DECIMAL` 和 `NUMERIC`。
- `M` 为该类型数据的精度，即整数位加上小数位的总长度。
- `D` 为小数位的长度。

`schema.json` 文件中对小数类型的定义如下：

```
{
    "ColumnName":"COL1",
    "ColumnType":"{DT} [UNSIGNED]",
```

```
    "ColumnPrecision":"{M}",
    "ColumnScale":"{D}"
}
```

时间和日期类型

TiDB 中的日期类型可被定义为 DT，其中：

- DT 为日期类型，包括 DATE 和 YEAR。

schema.json 文件中对日期类型的定义如下：

```
{
    "ColumnName":"COL1",
    "ColumnType":"{DT}"
}
```

TiDB 中的时间类型可被定义为 TT[(M)]，其中：

- TT 为时间类型，包括 TIME、DATETIME 和 TIMESTAMP。
- M 为秒的精度，取值范围为 0~6。

schema.json 文件中对时间类型的定义如下：

```
{
    "ColumnName":"COL1",
    "ColumnType":"{TT}",
    "ColumnScale":"{M}"
}
```

字符串类型

TiDB 中的字符串类型可被定义为 ST[(M)]，其中：

- ST 为字符串类型，包括 CHAR、VARCHAR、TEXT、BINARY、BLOB、JSON 等。
- M 表示字符串的最大长度。

schema.json 文件中对字符串类型的定义如下：

```
{
    "ColumnName":"COL1",
    "ColumnType":"{ST}",
    "ColumnLength":"{M}"
}
```

Enum/Set 类型

schema.json 文件中对 Enum/Set 类型的定义如下：

```
{
    "ColumnName":"COL1",
    "ColumnType":"{ENUM/SET}",
}
```

13.9.3.3 管理 Changefeed

本文介绍 Changefeed 相关的各种管理方法，其中大部分功能是通过 TiCDC 的命令行工具来完成的。如果你需要，也可以通过 TiCDC 暴露的 HTTP 接口实现类似功能，详细信息参考TiCDC OpenAPI。

13.9.3.3.1 创建同步任务

使用以下命令来创建同步任务：

```
cdc cli changefeed create --server=http://10.0.10.25:8300 --sink-uri="mysql://root:123456@127
    ↪ .0.0.1:3306/" --changefeed-id="simple-replication-task"
```

```
Create changefeed successfully!
ID: simple-replication-task
Info: {"upstream_id":7178706266519722477,"namespace":"default","id":"simple-replication-task","
    ↪ sink_uri":"mysql://root:xxxxx@127.0.0.1:4000/?time-zone=","create_time":"2022-12-19T15
    ↪ :05:46.679218+08:00","start_ts":438156275634929669,"engine":"unified","config":{"
    ↪ case_sensitive":true,"enable_old_value":true,"force_replicate":false,"
    ↪ ignore_ineligible_table":false,"check_gc_safe_point":true,"enable_sync_point":true,"
    ↪ bdr_mode":false,"sync_point_interval":30000000000,"sync_point_retention":3600000000000,"
    ↪ filter":{"rules":["test.*"],"event_filters":null},"mounter":{"worker_num":16},"sink":{"
    ↪ protocol":"","schema_registry":"","csv":{"delimiter":",","quote":"\"","null":"\\N","
    ↪ include_commit_ts":false},"column_selectors":null,"transaction_atomicity":"none","
    ↪ encoder_concurrency":16,"terminator":"\r\n","date_separator":"none","
    ↪ enable_partition_separator":false},"consistent":{"level":"none","max_log_size":64,"
    ↪ flush_interval":2000,"storage":""}},"state":"normal","creator_version":"v6.5.0"}
```

13.9.3.3.2 查询同步任务列表

使用以下命令来查询同步任务列表：

```
cdc cli changefeed list --server=http://10.0.10.25:8300
```

```
[{
    "id": "simple-replication-task",
    "summary": {
      "state": "normal",
      "tso": 417886179132964865,
      "checkpoint": "2020-07-07 16:07:44.881",
      "error": null
```

```
    }
}]
```

- checkpoint 即为 TiCDC 已经将该时间点前的数据同步到了下游。
- state 为该同步任务的状态：
  - normal：正常同步
  - stopped：停止同步（手动暂停）
  - error：停止同步（出错）
  - removed：已删除任务（只在指定 --all 选项时才会显示该状态的任务。未指定时，可通过 query 查询该状态的任务）
  - finished：任务已经同步到指定 target-ts，处于已完成状态（只在指定 --all 选项时才会显示该状态的任务。未指定时，可通过 query 查询该状态的任务）。

13.9.3.3.3 查询特定同步任务

使用 changefeed query 命令可以查询特定同步任务（对应某个同步任务的信息和状态），指定 --simple 或 -s 参数会简化输出，提供最基本的同步状态和 checkpoint 信息。不指定该参数会输出详细的任务配置、同步状态和同步表信息。

```
cdc cli changefeed query -s --server=http://10.0.10.25:8300 --changefeed-id=simple-replication-
    ↪ task
```

```
{
 "state": "normal",
 "tso": 419035700154597378,
 "checkpoint": "2020-08-27 10:12:19.579",
 "error": null
}
```

以上命令中：

- state 代表当前 changefeed 的同步状态，各个状态必须和 changefeed list 中的状态相同。
- tso 代表当前 changefeed 中已经成功写入下游的最大事务 TSO。
- checkpoint 代表当前 changefeed 中已经成功写入下游的最大事务 TSO 对应的时间。
- error 记录当前 changefeed 是否有错误发生。

```
cdc cli changefeed query --server=http://10.0.10.25:8300 --changefeed-id=simple-replication-task
```

```
{
  "info": {
    "sink-uri": "mysql://127.0.0.1:3306/?max-txn-row=20\u0026worker-number=4",
    "opts": {},
    "create-time": "2020-08-27T10:33:41.687983832+08:00",
    "start-ts": 419036036249681921,
```

```
  "target-ts": 0,
  "admin-job-type": 0,
  "sort-engine": "unified",
  "sort-dir": ".",
  "config": {
    "case-sensitive": true,
    "enable-old-value": false,
    "filter": {
      "rules": [
        "*.*"
      ],
      "ignore-txn-start-ts": null,
      "ddl-allow-list": null
    },
    "mounter": {
      "worker-num": 16
    },
    "sink": {
      "dispatchers": null,
    },
    "scheduler": {
      "type": "table-number",
      "polling-time": -1
    }
  },
  "state": "normal",
  "history": null,
  "error": null
},
"status": {
  "resolved-ts": 419036036249681921,
  "checkpoint-ts": 419036036249681921,
  "admin-job-type": 0
},
"count": 0,
"task-status": [
  {
    "capture-id": "97173367-75dc-490c-ae2d-4e990f90da0f",
    "status": {
      "tables": {
        "47": {
          "start-ts": 419036036249681921
        }
      },
      "operation": null,
```

```
        "admin-job-type": 0
      }
    }
  ]
}
```

以上命令中：

- `info` 代表查询 changefeed 的同步配置。
- `status` 代表查询 changefeed 的同步状态信息。
  - `resolved-ts` 代表当前 changefeed 中已经成功从 TiKV 发送到 TiCDC 的最大事务 TS。
  - `checkpoint-ts` 代表当前 changefeed 中已经成功写入下游的最大事务 TS。
  - `admin-job-type` 代表一个 changefeed 的状态：
    * `0`：状态正常。
    * `1`：任务暂停，停止任务后所有同步 `processor` 会结束退出，同步任务的配置和同步状态都会保留，可以从 `checkpoint-ts` 恢复任务。
    * `2`：任务恢复，同步任务从 `checkpoint-ts` 继续同步。
    * `3`：任务已删除，接口请求后会结束所有同步 `processor`，并清理同步任务配置信息。同步状态保留，只提供查询，没有其他实际功能。
- `task-status` 代表查询 changefeed 所分配的各个同步子任务的状态信息。

13.9.3.3.4 停止同步任务

使用以下命令来停止同步任务：

```
cdc cli changefeed pause --server=http://10.0.10.25:8300 --changefeed-id simple-replication-task
```

以上命令中：

- `--changefeed-id=uuid` 为需要操作的 changefeed ID。

13.9.3.3.5 恢复同步任务

使用以下命令恢复同步任务：

```
cdc cli changefeed resume --server=http://10.0.10.25:8300 --changefeed-id simple-replication-task
```

- `--changefeed-id=uuid` 为需要操作的 changefeed ID。
- `--overwrite-checkpoint-ts`：从 v6.2 开始支持指定 changefeed 恢复的起始 TSO。TiCDC 集群将从这个 TSO 开始拉取数据。该项支持 now 或一个具体的 TSO（如 434873584621453313），指定的 TSO 应在 (GC safe point, CurrentTSO] 范围内。如未指定该参数，默认从当前的 `checkpoint-ts` 同步数据。
- `--no-confirm`：恢复同步任务时无需用户确认相关信息。默认为 false。

注意：

- 若 `--overwrite-checkpoint-ts` 指定的 TSO `t2` 大于 changefeed 的当前 checkpoint TSO `t1`（可通过 `cdc cli changefeed query` 命令获取），则会导致 `t1` 与 `t2` 之间的数据不会同步到下游，造成数据丢失。
- 若 `--overwrite-checkpoint-ts` 指定的 TSO `t2` 小于 changefeed 的当前 checkpoint TSO `t1`，则会导致 TiCDC 集群从一个旧的时间点 `t2` 重新拉取数据，可能会造成数据重复（例如 TiCDC 下游为 MQ sink）。

13.9.3.3.6 删除同步任务

使用以下命令删除同步任务：

```
cdc cli changefeed remove --server=http://10.0.10.25:8300 --changefeed-id simple-replication-task
```

- `--changefeed-id=uuid` 为需要操作的 changefeed ID。

13.9.3.3.7 更新同步任务配置

TiCDC 从 4.0.4 开始支持非动态修改同步任务配置，修改 changefeed 配置需要按照 暂停任务 -> 修改配置 -> 恢复任务 的流程。

```
cdc cli changefeed pause -c test-cf --server=http://10.0.10.25:8300
cdc cli changefeed update -c test-cf --server=http://10.0.10.25:8300 --sink-uri="mysql
    ↪ ://127.0.0.1:3306/?max-txn-row=20&worker-number=8" --config=changefeed.toml
cdc cli changefeed resume -c test-cf --server=http://10.0.10.25:8300
```

当前支持修改的配置包括：

- changefeed 的 `sink-uri`
- changefeed 配置文件及文件内所有配置
- changefeed 是否使用文件排序和排序目录
- changefeed 的 `target-ts`

13.9.3.3.8 管理同步子任务处理单元 (processor)

- 查询 `processor` 列表：

```
cdc cli processor list --server=http://10.0.10.25:8300
```

```
[
        {
                "id": "9f84ff74-abf9-407f-a6e2-56aa35b33888",
```

```
                "capture-id": "b293999a-4168-4988-a4f4-35d9589b226b",
                "changefeed-id": "simple-replication-task"
        }
]
```

- 查询特定 processor，对应于某个节点处理的同步子任务信息和状态：

```
cdc cli processor query --server=http://10.0.10.25:8300 --changefeed-id=simple-replication-
    ↪ task --capture-id=b293999a-4168-4988-a4f4-35d9589b226b
```

```
{
  "status": {
    "tables": {
      "56": {    # 56 表示同步表 id，对应 TiDB 中表的 tidb_table_id
        "start-ts": 417474117955485702
      }
    },
    "operation": null,
    "admin-job-type": 0
  },
  "position": {
    "checkpoint-ts": 417474143881789441,
    "resolved-ts": 417474143881789441,
    "count": 0
  }
}
```

以上命令中：

- status.tables 中每一个作为 key 的数字代表同步表的 id，对应 TiDB 中表的 tidb_table_id。
- resolved-ts 代表当前 Processor 中已经排序数据的最大 TSO。
- checkpoint-ts 代表当前 Processor 已经成功写入下游的事务的最大 TSO。

13.9.3.3.9 输出行变更的历史值从 v4.0.5 版本开始引入

默认配置下，同步任务输出的 TiCDC Open Protocol 行变更数据只包含变更后的值，不包含变更前行的值，因此该输出数据不满足 TiCDC Open Protocol 的消费端使用行变更历史值的需求。

从 v4.0.5 开始，TiCDC 支持输出行变更数据的历史值。若要开启该特性，需要在 changefeed 的配置文件的根级别指定以下配置：

```
enable-old-value = true
```

从 v5.0 开始默认启用该特性，开启该特性后 TiCDC Open Protocol 的输出格式参考TiCDC 开放数据协议 - Row Changed Event。

13.9.3.3.10 同步启用了 TiDB 新的 Collation 框架的表

从 v4.0.15、v5.0.4、v5.1.1 和 v5.2.0 开始，TiCDC 支持同步启用了 TiDB 新的 Collation 框架的表。

13.9.3.3.11 同步没有有效索引的表

从 v4.0.8 开始，TiCDC 支持通过修改任务配置来同步没有有效索引的表。若要开启该特性，需要在 changefeed 配置文件的根级别进行如下指定：

```
enable-old-value = true
force-replicate = true
```

> **警告：**
>
> 对于没有有效索引的表，INSERT 和 REPLACE 等操作不具备可重入性，因此会有数据冗余的风险。TiCDC 在同步过程中只保证数据至少分发一次，因此开启该特性同步没有有效索引的表，一定会导致数据冗余出现。如果不能接受数据冗余，建议增加有效索引，譬如增加具有 AUTO RANDOM 属性的主键列。

13.9.3.3.12 Unified Sorter 功能

> **注意：**
>
> 从 v6.0.0 开始，TiCDC 内部默认使用 DB Sorter 引擎来对数据进行排序，不再使用 Unified Sorter。建议用户不再主动配置 Sorter 项。

Unified Sorter 是 TiCDC 中的排序引擎功能，用于缓解以下场景造成的内存溢出问题：

- 如果 TiCDC 数据订阅任务的暂停中断时间长，其间积累了大量的增量更新数据需要同步。
- 从较早的时间点启动数据订阅任务，业务写入量大，积累了大量的更新数据需要同步。

对 v4.0.13 版本之后的 `cdc cli` 创建的 changefeed，默认开启 Unified Sorter。对 v4.0.13 版本前已经存在的 changefeed，则使用之前的配置。

要确定一个 changefeed 上是否开启了 Unified Sorter 功能，可执行以下示例命令查看（假设 PD 实例的 IP 地址为 `http://10.0.10.25:2379`）：

```
cdc cli --server="http://10.0.10.25:8300" changefeed query --changefeed-id=simple-replication-
    ↪ task | grep 'sort-engine'
```

以上命令的返回结果中，如果 `sort-engine` 的值为 “unified”，则说明 Unified Sorter 已在该 changefeed 上开启。

注意：

- 如果服务器使用机械硬盘或其他有延迟或吞吐有瓶颈的存储设备，Unified Sorter 性能会受到较大影响。
- Unified Sorter 默认使用 `data_dir` 储存临时文件。建议保证硬盘的空闲容量大于等于 500 GiB。对于生产环境，建议保证每个节点上的磁盘可用空间大于（业务允许的最大）`checkpoint-ts` 延迟 * 业务高峰上游写入流量。此外，如果在 `changefeed` 创建后预期需要同步大量历史数据，请确保每个节点的空闲容量大于等于要追赶的同步数据。

13.9.3.4 Changefeed 日志过滤器

TiCDC 支持基于表和事件两个维度的过滤功能，本文分别介绍两种过滤器的使用方法。

13.9.3.4.1 Table Filter 表过滤器

TiCDC 支持基于库表名的过滤功能，你可以通过下列配置来选定或过滤掉指定的表：

```
[filter]
#### 过滤器规则
rules = ['*.*', '!test.*']
```

常见的过滤器规则示例：

- `rules = ['*.*']`
  - 同步所有的表（不包含系统表）
- `rules = ['test1.*']`
  - 同步库 `test1` 下的所有表
- `rules = ['*.*', '!scm1.tbl2']`
  - 同步所有的表但排除表 `scm1.tbl2`
- `rules = ['scm1.tbl2', 'scm1.tbl3']`
  - 只同步表 `scm1.tbl2` 和 `scm1.tbl3`
- `rules = ['scm1.tidb_*']`
  - 同步库 `scm1` 下所有表名前缀为 `tidb_` 的表

更多用法说明参见：库表过滤语法

13.9.3.4.2 Event Filter 事件过滤器从 v6.2.0 版本开始引入

TiCDC 在 v6.2.0 中新增了事件过滤器功能，你可以通过配置该规则来过滤符合指定条件的 DML 和 DDL 事件。

以下是事件过滤器的配置规则示例：

```
[filter]
#### 事件过滤器的规则应该写在 filter 配置项之下，可以同时配置多个事件过滤器。

[[filter.event-filters]]
matcher = ["test.worker"] # 该过滤规则只应用于 test 库中的 worker 表
ignore-event = ["insert"] # 过滤掉 insert 事件
ignore-sql = ["^drop", "add column"] # 过滤掉以 "drop" 开头或者包含 "add column" 的 DDL
ignore-delete-value-expr = "name = 'john'" # 过滤掉包含 name = 'john' 条件的 delete DML
ignore-insert-value-expr = "id >= 100" # 过滤掉包含 id >= 100 条件的 insert DML
ignore-update-old-value-expr = "age < 18 or name = 'lili'" # 过滤掉旧值 age < 18 或 name = 'lili'
    ↪ 的 update DML
ignore-update-new-value-expr = "gender = 'male' and age > 18" # 过滤掉新值 gender = 'male' 且 age
    ↪ > 18 的 update DML
```

配置参数说明：

- `matcher`：该事件过滤器所要匹配的数据库名和表名，其匹配规则和表库过滤规则相一致。
- `ignore-event`：要过滤的事件类型，它是一个字符串数组，可以配置多个事件类型。目前支持的类型如下表所示:

Event	分类	别名	说明
all dml			匹配所有 DML events
all ddl			匹配所有 DDL events
insert	DML		匹配 insert DML event
update	DML		匹配 update DML event
delete	DML		匹配 delete DML event
create schema	DDL	create database	匹配 create database event
drop schema	DDL	drop database	匹配 drop database event
create table	DDL		匹配 create table event
drop table	DDL		匹配 drop table event
rename table	DDL		匹配 rename table event
truncate table	DDL		匹配 truncate table event
alter table	DDL		匹配 alter table event (包含 alter table 的所有子句和 create/drop index)
add table partition	DDL		匹配 add table partition event
drop table partition	DDL		匹配 drop table partition event
truncate table partition	DDL		匹配 truncate table partition event
create view	DDL		匹配 create view event
drop view	DDL		匹配 drop view event

- `ignore-sql`：要过滤的 DDL 语句的正则表达式。该参数接受一个字符串数组，数组中可以配置多条正则表达式。该配置仅对 DDL 事件生效。
- `ignore-delete-value-expr`：配置一个 SQL 表达式，对带有指定值的 DELETE 类型的 DML 事件生效。
- `ignore-insert-value-expr`：配置一个 SQL 表达式，对带有指定值的 INSERT 类型的 DML 事件生效。
- `ignore-update-old-value-expr`：配置一个 SQL 表达式，对带有指定旧值的 UPDATE 类型的 DML 事件生效。
- `ignore-update-new-value-expr`：配置一个 SQL 表达式，对带有指定新值的 UPDATE 类型的 DML 事件生效。

> 注意：
>
> - TiDB 在更新聚簇索引的列值时，会将一个 UPDATE 事件拆分成为 DELETE 和 INSERT 事件，TiCDC 无法将该类事件识别为 UPDATE 事件，因此无法正确地进行过滤。
> - 在配置 SQL 表达式时，请确保符合 matcher 规则的所有表均包含了对应 SQL 表达式中指明的所有列，否则同步任务将无法创建成功。此外，若在同步的过程中表的结构发生变化，不再包含 SQL 表达式中的列，那么同步任务将会进入无法自动恢复的错误状态，你需要手动修改配置并进行恢复操作。

13.9.3.4.3 DDL 白名单

目前 TiCDC 在同步 DDL 时使用白名单策略，只有在白名单内部的 DDL 会同步到下游，不在白名单内的 DDL 不会被 TiCDC 同步到下游。

以下为 TiCDC 支持同步的 DDL 的列表。

- create database
- drop database
- create table
- drop table
- add column
- drop column
- create index / add index
- drop index
- truncate table
- modify column
- rename table
- alter column default value
- alter table comment
- rename index
- add partition
- drop partition
- truncate partition
- create view
- drop view
- alter table character set

- alter database character set
- recover table
- add primary key
- drop primary key
- rebase auto id
- alter table index visibility
- exchange partition
- reorganize partition
- alter table ttl
- alter table remove ttl

13.9.3.5 TiCDC 双向复制

从 v6.5.0 版本开始，TiCDC 支持在两个 TiDB 集群之间进行双向复制。基于该功能，你可以使用 TiCDC 来构建 TiDB 集群的多写多活解决方案。

本文档以在两个 TiDB 集群之间进行双向复制为例，介绍双向复制的使用方法。

13.9.3.5.1 部署双向复制

TiCDC 复制功能只会将指定时间点之后的增量变更复制到下游集群。开始双向复制之前，需要采取以下步骤：

1. （可选）根据实际需要，使用数据导出工具Dumpling 和导入工具TiDB Lightning 将两个集群的数据导入到对方集群。
2. 在这两个 TiDB 集群之间部署两套 TiCDC 集群，集群的拓扑如下图所示，图中箭头所指的方向即为该 TiCDC 集群同步数据的流向。

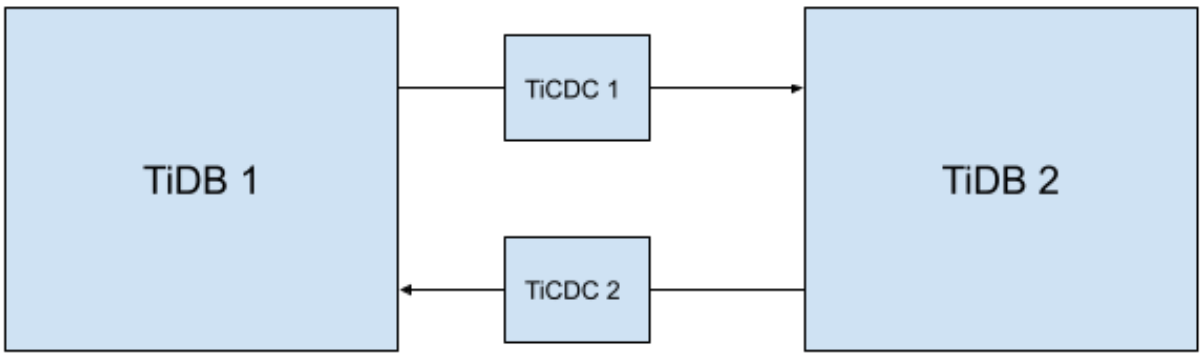


图 264: TiCDC bidirectional replication

3. 指定上下游集群的数据同步的开始时间点。
    1. 确认上下游集群的时间点。如果进行双集群容灾，那么建议确保两个集群在某个对应的时刻的数据是一致的，例如 TiDB A 在 `ts=1` 时刻与 TiDB B 在 `ts=2` 的时刻数据是一致的。

2. 在创建同步任务时，分别把对应集群的同步任务的 `--start-ts` 参数指定为对应的 `tso`，即上游为 TiDB A 的同步任务需设置参数 `--start-ts=1`，上游为 TiDB B 的同步任务需设置参数 `--start-ts=2`。

4. 在创建同步任务的 `--config` 参数所指定的配置文件中，添加如下配置:

```
# 是否启用 bdr 模式
bdr-mode = true
```

5. （可选）如果需要追踪数据写入源，可以使用 TiDB 系统变量`tidb_source_id` 为集群设置不同的数据源 ID。

这样，以上搭建好的集群即可对数据进行双向复制。

13.9.3.5.2 执行 DDL

开启双向复制功能后，TiCDC 不会同步任何 DDL。用户需要自行在上下游集群中分别执行 DDL。

需要注意的是，某些 DDL 会造成表结构变更或者数据更改时序问题，从而导致数据同步后出现不一致的情况。因此，在开启双向同步功能后，只有下表中的 DDL 可以在业务不停止数据写入的情况下执行。

事件	是否会引起 change-feed 错误	说明
create database	是	用户手动在上下游都执行了 DDL 之后，错误可以自动恢复
drop database	是	需要手动重启 changefeed，指定 `--overwrite-checkpoint-ts` 为该条 DDL 的 commitTs 来恢复
create table	是	用户手动在上下游都执行了 DDL 之后，错误可以自动恢复
drop table	是	需要手动重启 changefeed，指定 `--overwrite-checkpoint-ts` 为该条 ddl 的 commitTs 来恢复
alter table comment	否	
rename index	否	
alter table index visibility	否	
add partition	是	用户手动在上下游都执行了 DDL 之后，错误可以自动恢复
drop partition	否	
create view	否	
drop view	否	
alter column default value	否	
reorganize partition	是	用户手动在上下游都执行了 DDL 之后，错误可以自动恢复
alter table ttl	否	

事件	是否会引起 changefeed 错误	说明
alter table remove ttl	否	
add not unique index	否	
drop not unique index	否	

如果需要执行以上列表中不存在的 DDL，需要采取以下步骤：

1. 暂停所有集群中需要执行 DDL 的对应的表的写入操作。
2. 等待所有集群中对应表的所有写入已经同步到其他集群后，手动在每一个 TiDB 集群上单独执行所有的 DDL。
3. 等待 DDL 完成之后，重新恢复写入。

13.9.3.5.3 停止双向复制

在业务数据停止写入之后，你可以在两个集群中都插入一行特殊的值，通过检查这两行特殊的值来确保数据达到了一致的状态。

检查完毕之后，停止同步任务即可停止双向复制。

13.9.3.5.4 使用限制

- DDL 的限制见执行 DDL 小节。
- 双向复制的集群不具备检测写冲突的功能，写冲突将会导致未定义问题。你需要在业务层面保证不出现写冲突。
- TiCDC 双向复制功能支持超过 2 个集群的双向同步，但是不支持多个集群级联模式的同步，即 TiDB A -> TiDB B -> TiDB C -> TiDB A 的环形复制方式。在这种部署方式下，如果其中一个链路出现问题则会影响整个数据同步链路。因此，如果需要部署多个集群之间的双向复制，每个集群都需要与其他集群两两相连，即 `TiDB A <-> TiDB B`，`TiDB B <-> TiDB C`，`TiDB C <-> TiDB A`。

13.9.4 监控告警

13.9.4.1 TiCDC 基本监控指标

从 v7.0.0 版本开始，使用 TiUP 一键部署 Grafana 时，会自动在 Grafana 监控页面新增 TiCDC Summary Dashboard。通过该监控面板，你可以快速地了解 TiCDC 服务器运行状态和同步任务的基本情况。

下图显示了 TiCDC Dashboard 的监控栏：

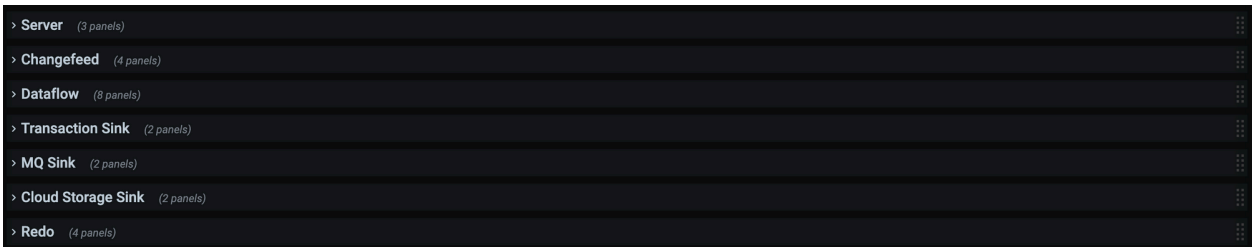


图 265: TiCDC Summary Dashboard - Overview

各监控栏说明如下：

- Server：集群中 TiCDC 节点的概要信息
- Changefeed：TiCDC 同步任务延迟和状态信息
- Dataflow：TiCDC 内部各个模块处理数据改变的各种统计信息
- Transaction Sink：下游为 MySQL 或者 TiDB 时的写延迟信息
- MQ Sink: 下游为 MQ 系统时的写延迟信息
- Cloud Storage Sink：下游为 Cloud Storage 时的写速率信息
- Redo：开启 Redo 功能时的写延迟信息

13.9.4.1.1 Server 监控栏

Server 监控栏示例如下：

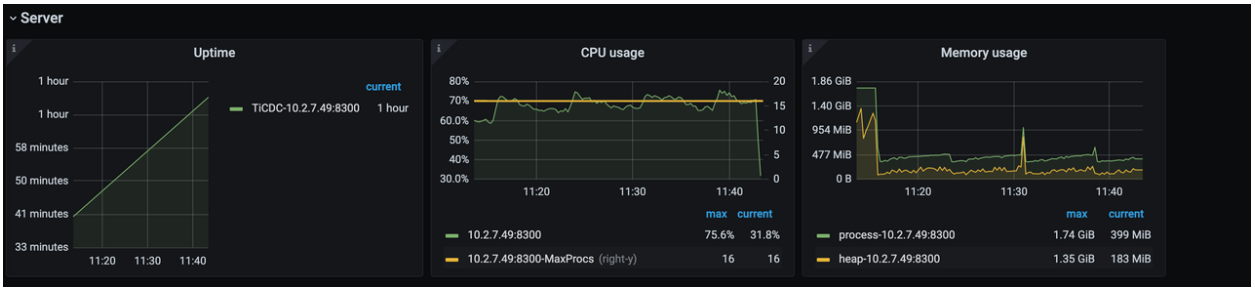


图 266: TiCDC Summary Dashboard - Server metrics

- Uptime：TiCDC 节点已经运行的时间。
- CPU usage：TiCDC 节点的 CPU 使用量。
- Memory usage：TiCDC 节点的内存使用量。

13.9.4.1.2 Changefeed 监控栏

Changefeed 监控栏示例如下：



图 267: TiCDC Summary Dashboard - Changefeed metrics

- Changefeed checkpoint lag：这个指标代表上游 TiDB 集群和下游系统之间的数据复制延迟，延迟以时间为单位。该指标反映了 Changefeed 整体的数据同步状况是否健康，通常情况下，lag 越小，说明同步任务状态越好。而当 lag 上升时，通常说明 Changefeed 的同步能力或者下游系统的消费能力无法匹配上游的写入速度。
- Changefeed resolved ts lag：这个指标代表了上游 TiDB 集群与 TiCDC 节点之间的数据延迟，延迟以时间为单位。该指标能够反映 Changefeed 拉取上游数据变更的能力，当 lag 上升时，说明 Changefeed 无法及时地拉取上游产生的数据变更。

13.9.4.1.3 Dataflow 监控栏



图 268: TiCDC Summary Dashboard - Puller metrics

- Puller output events/s：TiCDC 节点中 Puller 模块每秒输出到 Sorter 模块的数据变更个数。该指标能够反映 TiCDC 拉取上游变更事件的速度。
- Puller output events：TiCDC 节点中 Puller 模块输出到 Sorter 模块的数据变更总数。

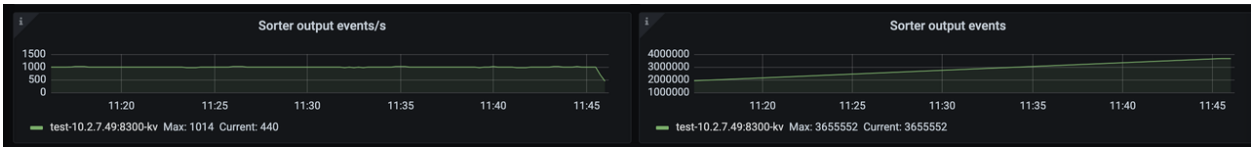


图 269: TiCDC Summary Dashboard - Sorter metrics

- Sorter output events/s：TiCDC 节点中 Sorter 模块每秒输出到 Sink 模块的数据变更个数。值得注意的是，Sorter 的数据输出速率会受到 Sink 模块的影响，因此在发现 Sorter 模块输出速率比 Puller 模块低时，不一定是因为 Sorter 模块排序速度过慢；而应该先观察 Sink 模块的相关指标，确认是否是因为 Sink 模块 Flush 数据的耗时较长，导致 Sorter 模块输出降低。
- Sorter output event：TiCDC 节点中 Sorter 模块输出到 Sink 模块的数据变更总个数。



图 270: TiCDC Summary Dashboard - Mounter metrics

- Mounter output events/s：TiCDC 节点中 Mounter 模块每秒解码的数据变更的个数。当上游发生的数据变更包含较多字段时，Mounter 的解码速度可能会受到影响。
- Mounter output event：TiCDC 节点中 Mounter 模块解码的数据变更的总个数。



图 271: TiCDC Summary Dashboard - Sink metrics

- Sink flush rows/s：TiCDC 节点中 Sink 模块每秒往下游输出的数据变更个数。该指标反映的是同步任务向下游进行同步的速率，当 Sink flush rows/s 小于 Puller output events/s 时，同步延迟可能会上升。
- Sink flush rows：TiCDC 节点中 Sink 模块输出的数据变更的总个数。

13.9.4.1.4 Transaction Sink 监控栏

Transaction Sink 监控栏示意如下，该监控栏只有下游为 MySQL 或者 TiDB 时才有数据。

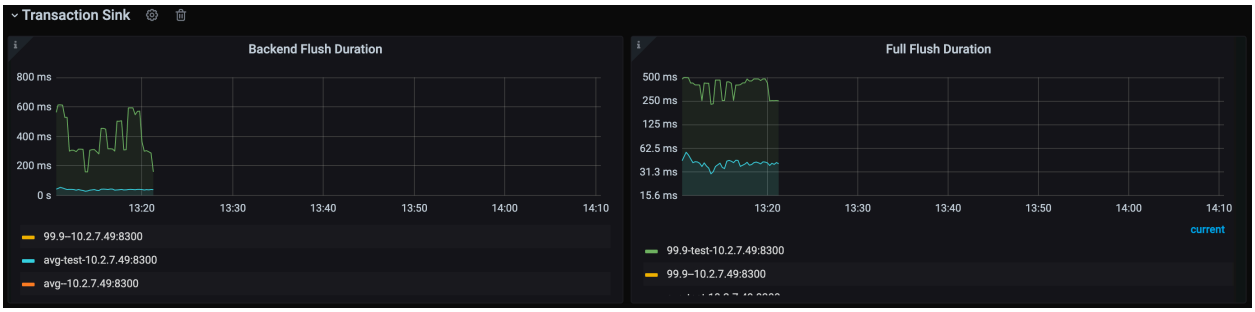


图 272: TiCDC Summary Dashboard - Transaction Sink metrics

- Backend Flush Duration：TiCDC Transaction Sink 模块在向下游执行一条 SQL 语句的耗时。通过观察该指标，能够判断下游的性能是否为同步速度的瓶颈。一般来说，p999 应该维持在 500 ms 内为佳，超过该值时，同步速度可能就会受到影响，引起 Changefeed checkpoint lag 上升。
- Full Flush Duration：TiCDC 中每个事务从 Sorter 排序完成直到发送到下游之间的总耗时。用该值减去 Backend Flush Duration 的值，即可得出一个事务在被执行到下游之前的总排队时长。如果排队时长较高，可以考虑给同步任务分配更多的内存 Quota。

13.9.4.1.5 MQ Sink 监控栏

MQ Sink 监控栏示意如下，该监控栏只有下游为 Kafka 时才有数据。

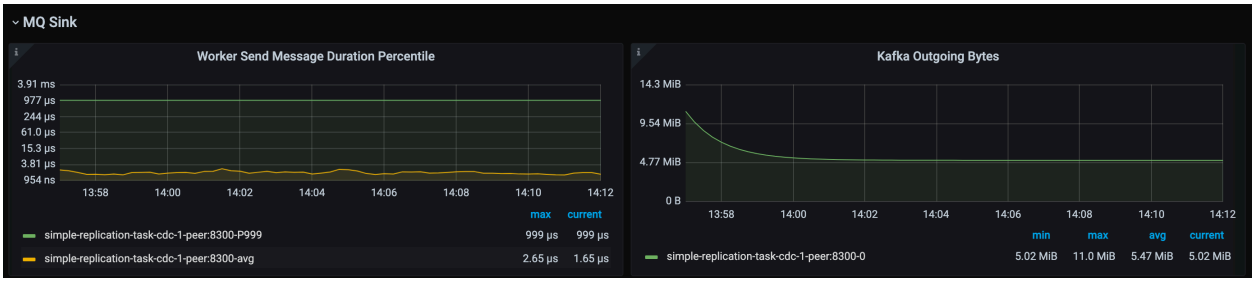


图 273: TiCDC Summary Dashboard - Transaction Sink metrics

- Worker Send Message Duration Percentile：TiCDC MQ Sink 的 Worker 往下游发送数据的延迟。
- Kafka Ongoing Bytes：TiCDC MQ Sink 往下游发送数据的速率。

13.9.4.1.6 Cloud Storage Sink 监控栏

Cloud Storage Sink 监控栏示意如下，该监控栏只有下游为 Cloud Storage 时才有数据。

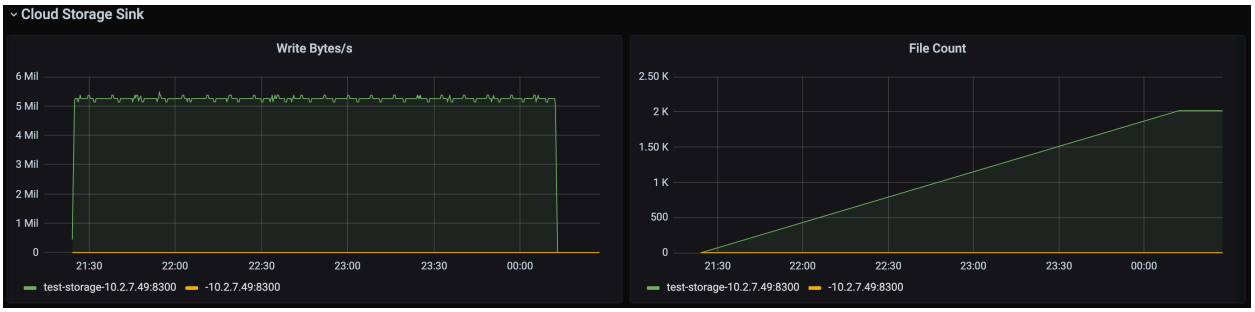


图 274: TiCDC Summary Dashboard - Transaction Sink metrics

- Write Bytes/s：Cloud Storage Sink 模块往下游写数据的速率。
- File Count：Cloud Storage Sink 模块写文件的总数量。

13.9.4.1.7 Redo 监控栏

Redo 监控栏示意如下，该监控栏只有在开启了 Redo Log 功能时才有数据。



图 275: TiCDC Summary Dashboard - Transaction Sink metrics

- Redo Write rows/s：Redo 模块每秒写数据的行数。当开启 Redo 功能时，如果发现同步任务的延迟上升，那么可以观察该指标是否和 Puller Output event/s 的值有较大差距。如果是，那么可能是由于 Redo 模块的写入能力不足造成延迟上升。
- Redo Write Byte/s：Redo 模块每秒写数据的速率。
- Redo flush log duration：Redo 模块往下游刷写数据的耗时。若该指标值较高，则可能是它影响了同步的速度。
- Redo flushall duration：数据变更停留在 Redo 模块中的总时长。

13.9.4.2 TiCDC 详细监控指标

使用 TiUP 部署 TiDB 集群时，一键部署的监控系统面板包含 TiCDC 面板。本文档对 TiCDC 监控面板上的各项指标进行详细说明。在日常运维中，运维人员可通过观察 TiCDC 面板上的指标了解 TiCDC 当前的状态。

本文档的对指标的介绍基于以下同步任务，即使用默认配置同步数据到 MySQL。

```
cdc cli changefeed create --server=http://10.0.10.25:8300 --sink-uri="mysql://root:123456@127
    ↪ .0.0.1:3306/" --changefeed-id="simple-replication-task"
```

下图显示了 TiCDC Dashboard 各监控面板：



图 276: TiCDC Dashboard - Overview

各监控面板说明如下：

- Server：TiDB 集群中 TiKV 节点和 TiCDC 节点的概要信息
- Changefeed：TiCDC 同步任务的详细信息
- Events：TiCDC 内部数据流转的详细信息
- TiKV：TiKV 中和 TiCDC 相关的详细信息

13.9.4.2.1 Server 面板

Server 面板示例如下：



图 277: TiCDC Dashboard - Server metrics

Server 面板的各指标说明如下：

- Uptime：TiKV 节点和 TiCDC 节点已经运行的时间
- Goroutine count：TiCDC 节点 Goroutine 的个数
- Open FD count：TiCDC 节点打开的文件句柄个数
- Ownership：TiCDC 集群中节点的当前状态
- Ownership history：TiCDC 集群中 Owner 节点的历史记录
- CPU usage：TiCDC 节点使用的 CPU
- Memory usage：TiCDC 节点使用的内存

13.9.4.2.2 Changefeed 面板

Changefeed 面板示例如下：



图 278: TiCDC Dashboard - Changefeed metrics 1

- Changefeed table count：一个同步任务中分配到各个 TiCDC 节点同步的数据表个数
- Processor resolved ts：TiCDC 节点内部状态中已同步的时间点
- Table resolved ts：同步任务中各数据表的同步进度
- Changefeed checkpoint：同步任务同步到下游的进度，正常情况下绿柱应和黄线相接
- PD etcd requests/s：TiCDC 节点每秒向 PD 读写数据的次数
- Exit error count/m：每分钟内导致同步中断的错误发生次数
- Changefeed checkpoint lag：同步任务上下游数据的进度差（以时间计算）
- Processor resolved ts lag：TiCDC 节点内部同步状态与上游的进度差（以时间计算）



图 279: TiCDC Dashboard - Changefeed metrics 2

- Sink write duration：TiCDC 将一个事务的更改写到下游的耗时直方图
- Sink write duration percentile：每秒钟中 95%、99% 和 99.9% 的情况下，TiCDC 将一个事务的更改写到下游所花费的时间
- Flush sink duration：TiCDC 异步刷写数据入下游的耗时直方图
- Flush sink duration percentile：每秒钟中 95%、99% 和 99.9% 的情况下，TiCDC 异步刷写数据入下游所花费的时间

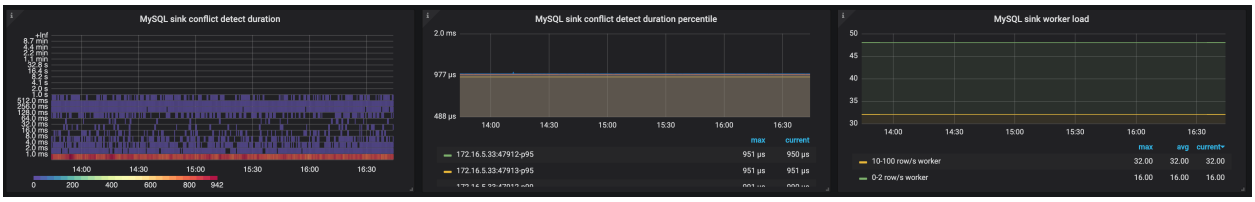


图 280: TiCDC Dashboard - Changefeed metrics 3

- MySQL sink conflict detect duration：MySQL 写入冲突检测耗时直方图
- MySQL sink conflict detect duration percentile：每秒钟中 95%、99% 和 99.9% 的情况下，MySQL 写入冲突检测耗时
- MySQL sink worker load：TiCDC 节点中写 MySQL 线程的负载情况

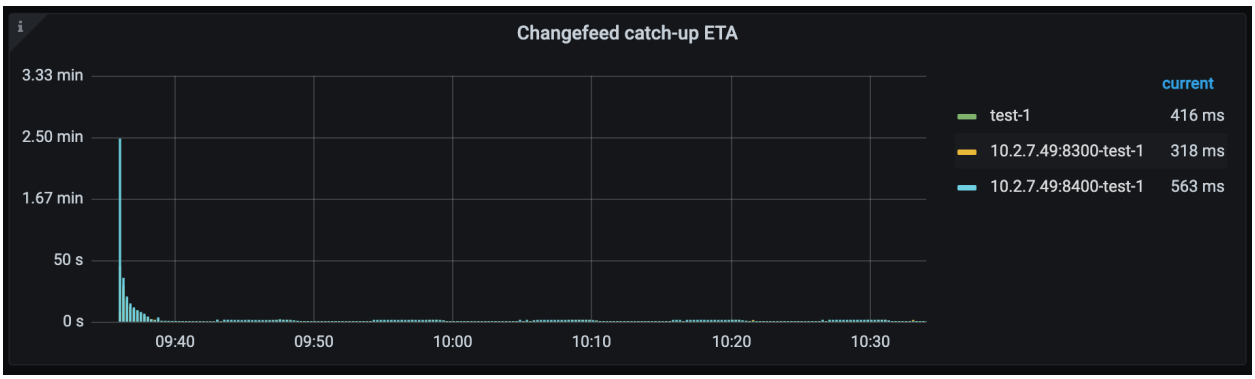
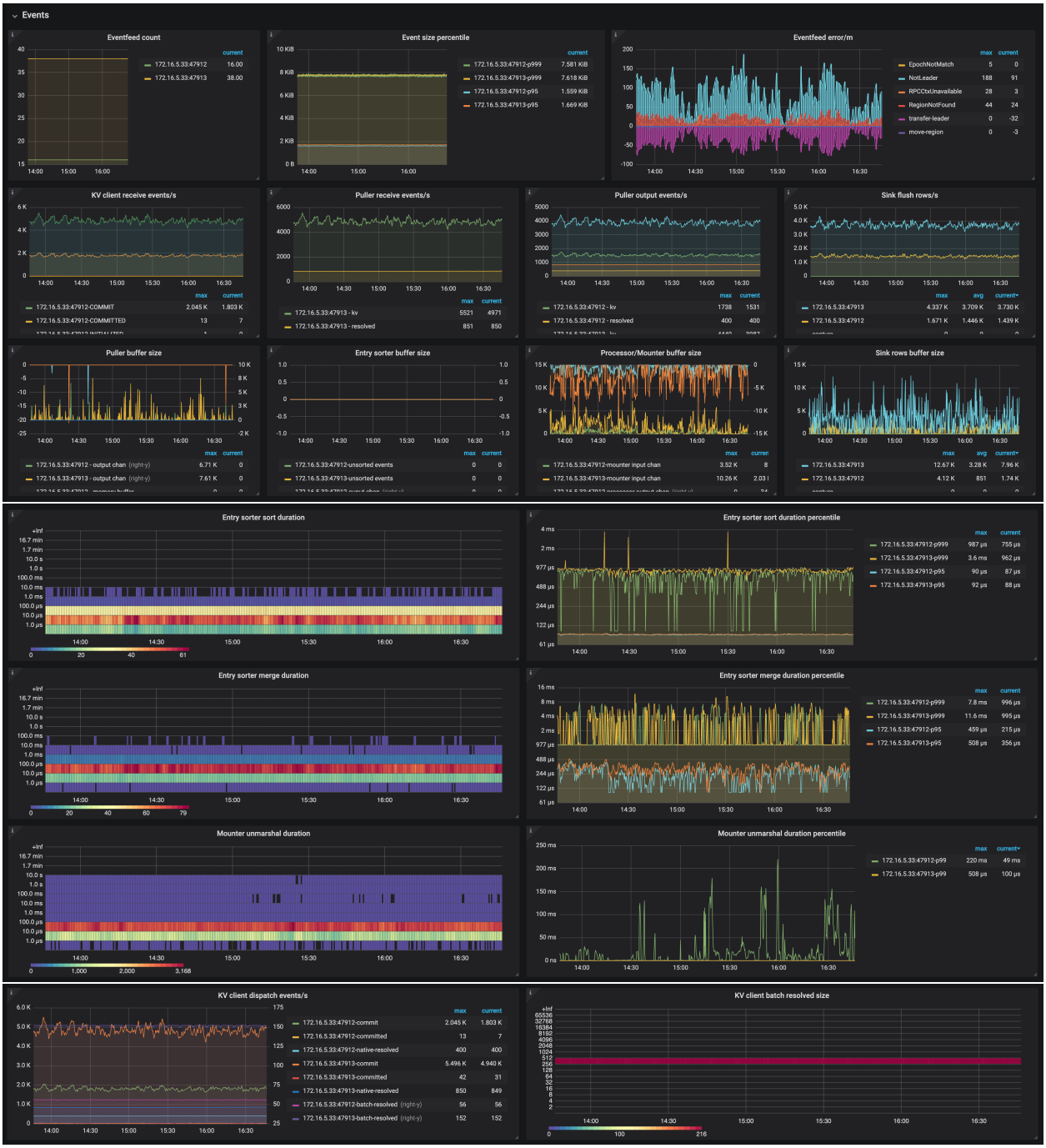


图 281: TiCDC Dashboard - Changefeed metrics 4

- Changefeed catch-up ETA：同步完上游写入的数据所需时间的估计值。当上游的写入速度大于 TiCDC 同步速度时，该值可能会异常的大。（由于 TiCDC 的同步速度受到较多因素制约，因此该值仅供参考，不能完全代表实际所需的同步时间。）

13.9.4.2.3 Events 面板

Events 面板示例如下：



Events 面板的各指标说明如下：

- Eventfeed count：TiCDC 节点中 Eventfeed RPC 的个数
- Event size percentile：每秒钟中 95% 和 99.9% 的情况下，TiCDC 收到的来自 TiKV 的数据变更消息大小
- Eventfeed error/m：TiCDC 节点中每分钟 Eventfeed RPC 遇到的错误个数
- KV client receive events/s：TiCDC 节点中 KV client 模块每秒收到来自 TiKV 的数据变更个数
- Puller receive events/s：TiCDC 节点中 Puller 模块每秒收到来自 KV client 模块的数据变更个数
- Puller output events/s：TiCDC 节点中 Puller 模块每秒输出到 Sorter 模块的数据变更个数
- Sink flush rows/s：TiCDC 节点每秒写到下游的数据变更的个数

- Puller buffer size：TiCDC 节点中缓存在 Puller 模块中的数据变更个数
- Entry sorter buffer size：TiCDC 节点中缓存在 Sorter 模块中的数据变更个数
- Processor/Mounter buffer size：TiCDC 节点中缓存在 Processor 模块和 Mounter 模块中的数据变更个数
- Sink row buffer size：TiCDC 节点中缓存在 Sink 模块中的数据变更个数
- Entry sorter sort duration：TiCDC 节点对数据变更进行排序的耗时直方图
- Entry sorter sort duration percentile：每秒钟中 95%，99% 和 99.9% 的情况下，TiCDC 排序数据变更所花费的时间
- Entry sorter merge duration：TiCDC 节点合并排序后的数据变更的耗时直方图
- Entry sorter merge duration percentile：每秒钟中 95%，99% 和 99.9% 的情况下，TiCDC 合并排序后的数据变更所花费的时间
- Mounter unmarshal duration：TiCDC 节点解码数据变更的耗时直方图
- Mounter unmarshal duration percentile：每秒钟中 95%，99% 和 99.9% 的情况下，TiCDC 解码数据变更所花费的时间
- KV client dispatch events/s：TiCDC 节点内部 KV client 模块每秒分发数据变更的个数
- KV client batch resolved size：TiKV 批量发给 TiCDC 的 resolved ts 消息的大小

13.9.4.2.4 TiKV 面板

TiKV 面板示例如下：



TiKV 面板的各指标说明如下：

- CDC endpoint CPU：TiKV 节点上 CDC endpoint 线程使用的 CPU
- CDC worker CPU：TiKV 节点上 CDC worker 线程使用的 CPU
- Min resolved ts：TiKV 节点上最小的 resolved ts
- Min resolved region：TiKV 节点上最小的 resolved ts 的 Region ID
- Resolved ts lag duration percentile：TiKV 节点上最小的 resolved ts 与当前时间的差距
- Initial scan duration：TiKV 节点与 TiCDC 建立链接时增量扫的耗时直方图
- Initial scan duration percentile：每秒钟中 95%、99% 和 99.9% 的情况下，TiKV 节点增量扫的耗时
- Memory without block cache：TiKV 节点在减去 RocksDB block cache 后使用的内存
- CDC pending bytes in memory：TiKV 节点中 CDC 模块使用的内存
- Captured region count：TiKV 节点上捕获数据变更的 Region 个数

13.9.4.3 TiCDC 集群监控报警规则

本文介绍了 TiCDC 组件的报警项及相应的报警规则。根据严重级别，报警项按照严重程度由高到低依次为：重要级别 (Critical)、警告级别 (Warning)。

13.9.4.3.1 重要级别报警项

对于重要级别的报警，需要密切关注异常指标。

`cdc_checkpoint_high_delay`

- 报警规则：

  (time() - ticdc_processor_checkpoint_ts / 1000) > 600

- 规则描述：

  TiCDC 某个同步任务延迟超过 10 分钟。

- 处理方法：

  参考TiCDC 同步任务出现中断的处理方法。

`cdc_resolvedts_high_delay`

- 报警规则：

  (time() - ticdc_processor_resolved_ts / 1000) > 300

- 规则描述：

  TiCDC 某个同步任务的 resolved ts 延迟超过 5 分钟。

- 处理方法：

  该告警与同步任务中断类似，可参考TiCDC 同步任务出现中断的处理方法。

`ticdc_processor_exit_with_error_count`

- 报警规则：

  ```
  changes(ticdc_processor_exit_with_error_count[1m])> 0
  ```

- 规则描述：

  TiCDC 某个同步任务报错退出。

- 处理方法：

  参考TiCDC 同步任务出现中断的处理方法。

13.9.4.3.2 警告级别报警项

警告级别的报警是对某一问题或错误的提醒。

`cdc_multiple_owners`

- 报警规则：

  ```
  sum(rate(ticdc_owner_ownership_counter[30s]))>= 2
  ```

- 规则描述：

  TiCDC 集群有多个 owner。

- 处理方法：

  收集 TiCDC 日志，定位原因。

`ticdc_mounter_unmarshal_and_mount_time_more_than_1s`

- 报警规则：

  ```
  histogram_quantile(0.9, rate(ticdc_mounter_unmarshal_and_mount_bucket[1m]))* 1000 > 1000
  ```

- 规则描述：

  TiCDC 某一同步任务解码变更数据的耗时超过 1 秒。

- 处理方法：

  收集 TiCDC 日志，定位原因。

`cdc_sink_execute_duration_time_more_than_10s`

- 报警规则：

  ```
  histogram_quantile(0.9, rate(ticdc_sink_txn_exec_duration_bucket[1m]))> 10
  ```

- 规则描述：

  TiCDC 某一同步任务写下游执行时间超过 10 秒。

- 处理方法：

  检查下游是否出现问题。

`cdc_processor_checkpoint_tso_no_change_for_1m`

- 报警规则：

  ```
  changes(ticdc_processor_checkpoint_ts[1m])< 1
  ```

- 规则描述：

  TiCDC 某一个同步任务进度超过 1 分钟没有推进。

- 处理方法：

  参考TiCDC 同步任务出现中断的处理方法。

`ticdc_puller_entry_sorter_sort_bucket`

- 报警规则：

  ```
  histogram_quantile(0.9, rate(ticdc_puller_entry_sorter_sort_bucket{}[1m]))> 1
  ```

- 规则描述：

  TiCDC puller entry sorter 排序延迟太高。

- 处理方法：

  收集 TiCDC 日志，定位原因。

`ticdc_puller_entry_sorter_merge_bucket`

- 报警规则：

  ```
  histogram_quantile(0.9, rate(ticdc_puller_entry_sorter_merge_bucket{}[1m]))> 1
  ```

- 规则描述：

  TiCDC puller entry sorter merge 延迟太高。

- 处理方法：

  收集 TiCDC 日志，定位原因。

`tikv_cdc_min_resolved_ts_no_change_for_1m`

- 报警规则：

  ```
  changes(tikv_cdc_min_resolved_ts[1m])< 1 and ON (instance)tikv_cdc_region_resolve_status{
  ↪ status="resolved"} > 0
  ```

- 规则描述：

  TiKV CDC 模块最小的 resolved ts 1 分钟没推进。

- 处理方法：

  收集 TiKV 日志，定位原因。

`tikv_cdc_scan_duration_seconds_more_than_10min`

- 报警规则：

  ```
  histogram_quantile(0.9, rate(tikv_cdc_scan_duration_seconds_bucket{}[1m]))> 600
  ```

- 规则描述：

  TiKV CDC 模块的增量扫描耗时超过 10 分钟。

- 处理方法：

  收集 TiCDC 监控和 TiKV 日志，定位原因。

`ticdc_sink_mysql_execution_error`

- 报警规则：

  ```
  changes(ticdc_sink_mysql_execution_error[1m])> 0
  ```

- 规则描述：

  TiCDC 某一同步任务写下游 MySQL 时遇到错误。

- 处理方法：

  MySQL 报错的情况较多，参考TiCDC 故障处理。

`ticdc_memory_abnormal`

- 报警规则：

  ```
  go_memstats_heap_alloc_bytes{job="ticdc"} > 1e+10
  ```

- 规则描述：

  TiCDC 堆内存使用量超过 10 GiB。

- 处理方法：

  收集 TiCDC 监控和 TiCDC 日志，定位原因。

13.9.4.4 TiCDC 架构设计与原理

13.9.4.4.1 TiCDC 软件架构

TiCDC 集群由多个 TiCDC 对等节点组成，是一种分布式无状态的架构设计。TiCDC 集群及节点内部组件的设计如下图所示：

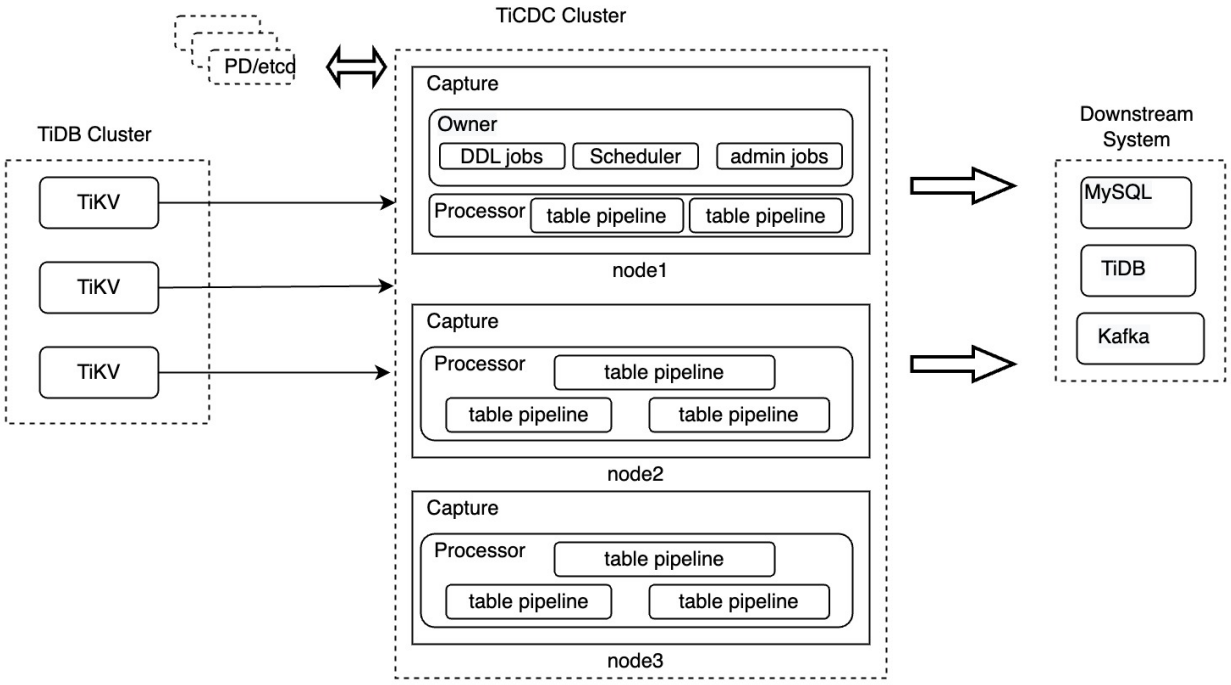


图 282: TiCDC architecture

13.9.4.4.2 组件介绍

上图中，TiCDC 集群由多个运行了 TiCDC 实例的节点组成，每个 TiCDC 实例都运行一个 Capture 进程。多个 Capture 进程中会有一个被选举成为 Owner Capture，负责完成 TiCDC 集群的负载调度、DDL 语句同步和其它管理任务。

每个 Capture 进程包含一个或多个 Processor 线程，用于同步上游 TiDB 集群中的表数据。由于 TiCDC 同步数据的最小单位是表，所以 Processor 是由多条 table pipeline 构成的。

每条 pipeline 包含 Puller、 Sorter、 Mounter 和 Sink 模块，如下图。

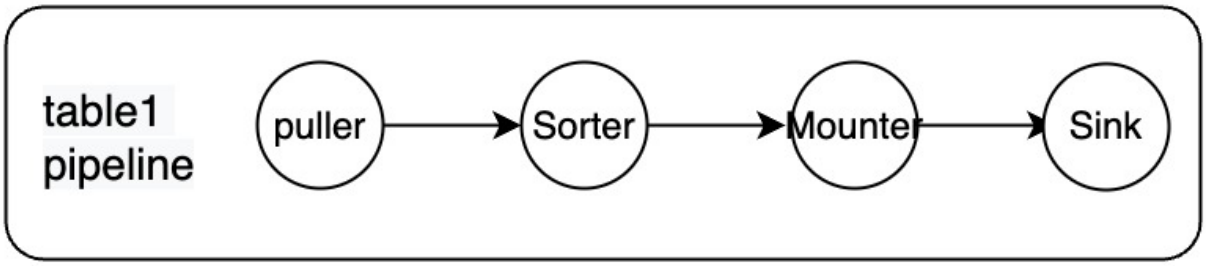


图 283: TiCDC architecture

各个模块之间是串行的关系，组合在一起完成从上游拉取、排序、加载和同步数据到下游的过程，其中：

- Puller：从 TiKV 节点获取 DDL 和行变更信息。

- Sorter：将接收到的变更在 TiCDC 内按照时间戳进行升序排序。
- Mounter：将变更按照对应的 Schema 信息转换成 TiCDC 可以处理的格式。
- Sink：将对应的变更应用到下游系统。

为了实现高可用，每个 TiCDC 集群都包含多个 TiCDC 节点，这些节点定期向 PD 集群中的 etcd 集群汇报自己的状态，并选举出其中一个节点作为 TiCDC 集群的 Owner。Owner 采用 etcd 统一存储状态来进行调度，并将调度结果直接写入 etcd。Processor 按照状态完成对应的任务，如果 Processor 所在节点出现异常，集群会将表调度到其他节点。如果 Owner 节点出现异常，其他节点的 Capture 进程会选举出新的 Owner，如下图所示：

TiCDC architecture

13.9.4.4.3 Changefeed 和 Task

TiCDC 中的 Changefeed 和 Task 是两个逻辑概念，前者是分配同步任务的方式，后者是同步任务拆分后的子任务名称，具体描述如下：

- Changefeed：代表一个同步任务，携带了需要同步的表信息，以及对应的下游信息和一些其他的配置信息。
- Task：当 TiCDC 接收到对应的同步任务之后，就会把这个任务拆分为若干个称之为 Task 的子任务，分配给 TiCDC 集群的各个节点上的 Capture 进程进行处理。

例如：

```
cdc cli changefeed create --server="http://127.0.0.1:8300" --sink-uri="kafka://127.0.0.1:9092/cdc
    ↪ -test?kafka-version=2.4.0&partition-num=6&max-message-bytes=67108864&replication-factor
    ↪ =1"
cat changefeed.toml
......
[sink]
dispatchers = [
    {matcher = ['test1.tab1', 'test2.tab2'], topic = "{schema}_{table}"},
    {matcher = ['test3.tab3', 'test4.tab4'], topic = "{schema}_{table}"},
]
```

关于命令中参数的详细释义，参考TiCDC Changefeed 配置参数。

以上命令创建了一个到 Kafka 集群的 Changefeed 任务，需要同步 `test1.tab1`、`test1.tab2`、`test3.tab3` 和 `test4.tab4` 四张表。TiCDC 接收到这个命令之后的处理流程如下：

1. TiCDC 将这个任务发送给 Owner Capture 进程。
2. Owner Capture 进程将这个任务的相关定义信息保存在 PD 的 etcd 中。
3. Owner Capture 将这个任务拆分成若干个 Task，并通知其他 Capture 进程各个 Task 需要完成的任务。
4. 各个 Capture 进程开始从对应的 TiKV 节点拉取信息，进行处理后完成同步。

如果将 Changefeed 和 Task 也包含到上文中提及的架构图，完整的 TiCDC 架构图如下：

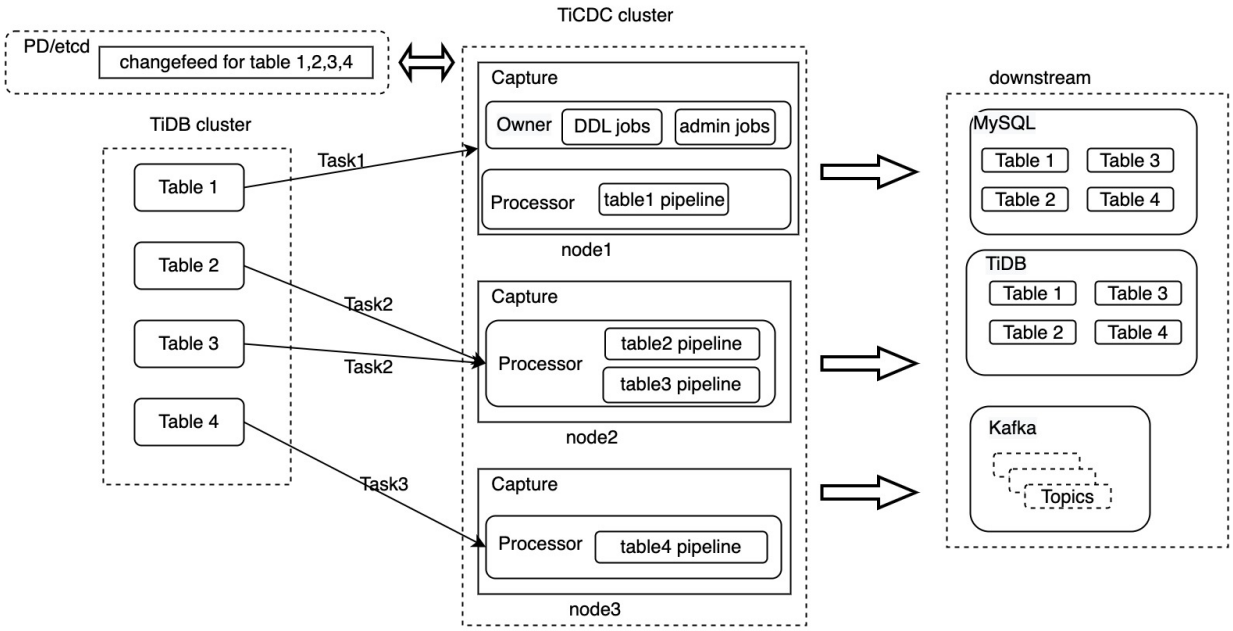


图 284: TiCDC architecture

上图创建了一个 Changefeed，需要同步 4 张表，这个 Changefeed 被拆分成了 3 个任务，均匀的分发到了 TiCDC 集群的 3 个 Capture 节点上，在 TiCDC 对这些数据进行了处理之后，数据同步到了下游的系统。

目前 TiCDC 支持的下游系统包含 MySQL、TiDB 和 Kafka。上面的过程只是讲解了 Changefeed 级别数据流转的基本过程，接下来，本文将以处理表 `table1` 的任务 Task1 为例，从更加详细的层面来描述 TiCDC 处理数据的过程：

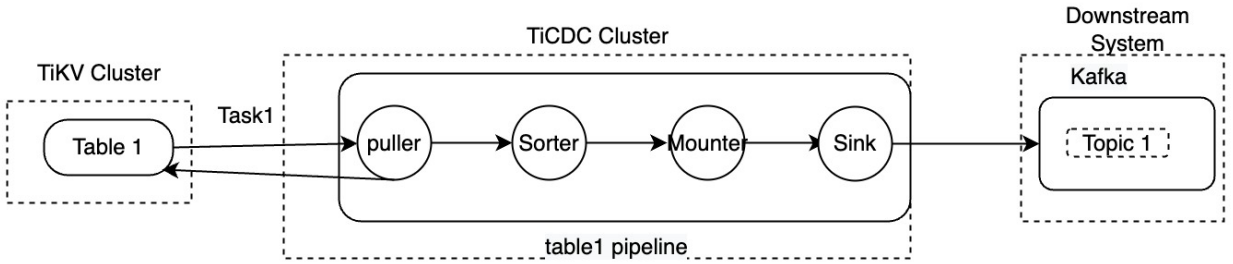


图 285: TiCDC architecture

1. 推流：发生数据改变时，TiKV 集群将数据主动推送给 Puller 模块。
2. 增量扫：Puller 模块在发现收到的数据改变不连续的时候，向 TiKV 节点主动拉取需要的数据。
3. 排序：Sorter 模块对获取的数据按照时间进行排序，并将排好序的数据发送给 Mounter。
4. 装载：Mounter 模块收到数据变更后，根据表的 schema 信息，将数据变更装载成 TiCDC sink 可以理解的格式。
5. 同步：Sink 模块根据下游的类型将数据变更同步到下游。

由于 TiCDC 的上游是支持事务的分布式关系型数据库 TiDB，在同步数据的时候，如何保证数据的一致性，以

及在同步多张表的时候，如何保证事务的一致性，都是很大的挑战。下面的章节会介绍 TiCDC 在确保事务特性时所使用的关键技术和概念。

13.9.4.4.4 TiCDC 关键概念

当 TiCDC 对应的下游系统是关系型数据库时，TiCDC 在同步数据的时候会确保单表事务的数据一致性，以及多表事务的最终一致性。另外，TiCDC 会确保上游 TiDB 集群发生过的数据变更能够被 TiCDC 向任意下游最少同步一次。

架构相关概念

- Capture：TiCDC 节点的运行进程，多个 Capture 进程构成了 TiCDC 集群，Capture 进程负责 TiKV 的数据变更的同步，包括接收和主动拉取两种方式，并向下游同步数据。
- Capture Owner：是一种 Capture 的角色，每个 TiCDC 集群同一时刻最多只存在一个 Capture Owner 角色，负责集群内部的调度。
- Processor：是 Capture 内部的逻辑线程，每个 Processor 负责处理同一个同步流中一个或多个 table 的数据。一个 Capture 节点可以运行多个 Processor。
- ChangeFeed：一个由用户启动的从上游 TiDB 同步到下游的任务，其中包含多个 Task，Task 会分布在不同的 Capture 节点进行数据同步处理。

时间戳相关概念

由于 TiCDC 需要确保数据被至少一次同步到下游，并且确保一定的一致性，因此引入了一系列时间戳（Timestamp，简称 TS）来对数据同步的状态进行描述。

ResolvedTS

这个时间戳在 TiKV 和 TiCDC 中都存在。

- TiKV 节点中的 ResolvedTS：代表某一个 Region leader 上开始时间最早的事务的开始时间，即：ResolvedTS = max(ResolvedTS, min(StartTS))。因为每个 TiDB 集群包含多个 TiKV 节点，所有 TiKV 节点上的 Region leader 的 ResolvedTS 的最小值，被称为 Global ResolvedTS。TiDB 集群确保 Global ResolvedTS 之前的事务都被提交了，或者可以认为这个时间戳之前不存在未提交的事务。
- TiCDC 节点中的 ResolvedTS：
  - table ResolvedTS：每张表都会有的表级别 ResolvedTS，表示这张表已经从 TiKV 接收到的数据改变的最低水位，可以简单的认为和 TiKV 节点上这张表的各个 Region 的 ResolvedTS 的最小值是相同的。
  - global ResolvedTS：各个 TiCDC 节点上的 Processor ResolvedTS 的最小值。由于 TiCDC 每个节点上都会存在一个或多个 Processor，每个 Processor 又对应多个 table pipeline。

  对于 TiCDC 节点来说，TiKV 节点发送过来的 ResolvedTS 信息是一种特殊的事件，它只包含一个格式为 `<resolvedTS: 时间戳>` 的特殊事件。通常情况下，ResolvedTS 满足以下约束：

```
table ResolvedTS >= global ResolvedTS
```

CheckpointTS

这个时间戳只在 TiCDC 中存在，它表示 TiCDC 已经同步给下游的数据的最低水位线，即 TiCDC 认为在这个时间戳之前的数据已经被同步到下游系统了。由于 TiCDC 同步数据的单位是表，所以 table CheckpointTS 表示表级别

的同步数据的水位线，Processor CheckpointTS 表示各个 Processor 中最小的 table CheckpointTS；Global checkpointTS 表示各个 Processor checkpointTS 中最低的 checkpointTS。通常情况下，Checkpoint TS 满足以下约束：

```
table CheckpointTS >= global CheckpointTS
```

因为 TiCDC 只会复制小于 global ResolvedTS 的数据到下游，所以存在下面的约束：

```
table ResolvedTS >= global ResolvedTS >= table CheckpointTS >= global CheckpointTS
```

随着数据的改变和事务的提交，TiKV 节点上的 ResolvedTS 会不断的向前推进，TiCDC 节点的 Puller 模块也会不断的收到 TiKV 推流过来的数据，并且根据收到的信息决定是否执行增量扫的过程，从而确保数据改变都能够被发送到 TiCDC 节点上。Sorter 模块则负责将 Puller 模块收到的信息按照时间戳进行升序排序，从而确保数据在表级别是满足一致性的。接下来，Mounter 模块把上游的数据改变装配成 Sink 模块可以消费的格式，并发送给 Sink，而 Sink 则负责把 CheckpointTS 到 ResolvedTS 之间的数据改变，按照发生的 TS 顺序同步到下游，并在下游返回后推进 CheckpointTS。

上面的内容只介绍了和 DML 语句相关的数据改变，并没有包含 DDL 相关的内容。下面对 DDL 语句相关的关键概念进行介绍。

Barrier TS

当系统发生 DDL 变更或者用户使用了 TiCDC 的 Syncpoint 时会产生的一个时间戳。

- DDL：这个时间戳会用来确保在这个 DDL 语句之前的改变都被应用到下游，之后执行对应的 DDL 语句，在 DDL 语句同步完成之后再同步其他的数据改变。由于 DDL 语句的处理是 Owner 角色的 Capture 负责的，DDL 语句对应的 Barrier TS 只会由 Owner 节点产生。
- Syncpoint：当你启用 TiCDC 的 Syncpoint 特性后，TiCDC 会根据你指定的间隔产生一个 Barrier TS。等所有表都同步到这个 Barrier TS 后，TiCDC 将此刻的 global CheckpointTS 作为 Primary Ts 插入下游的 TiDB 记录 tsMap 信息的表中，然后 TiCDC 才会继续向下游同步数据。

一个 Barrier TS 被生成后, TiCDC 会保证只有小于 Barrier TS 的数据会被复制到下游，并且保证小于 Barrier TS 的数据全部被复制到下游之前，同步任务不会再推进。Owner 不断地比较 global CheckpointTS 和 Barrier TS 的大小，确定小于 Barrier TS 的数据是否已经被同步完成。如果 global CheckpointTS = Barrier TS，执行完对应的操作（如 DDL 或者记录 global CheckpointTS 到下游）后，同步继续；否则需要继续等待所有小于 Barrier TS 数据的同步完成。

13.9.4.4.5 主要流程

本章节将介绍 TiCDC 软件的常见操作所对应的主要流程，以帮助你更好的理解 TiCDC 的工作原理。

注意，下面所描述的启动流程只存在于 TiCDC 进程内部，对于用户是完全透明的。因此，你在启动 TiCDC 进程时无需关心自己启动的是什么节点。

启动 TiCDC 节点

- 启动非 Owner 的 TiCDC 节点：
    1. 启动 Capture 进程。
    2. 启动 processor。
    3. 接收 Owner 下发的 Task 调度命令。

4. 根据调度命令启动或停止 tablePipeline。

- 启动 Owner 的 TiCDC 节点：

1. 启动 Capture 进程。
2. 当选 Owner 并启动对应的线程。
3. 读取 Changefeed 信息。
4. 启动 Changefeed 管理逻辑。
5. 根据 Changefeed 配置和最新 CheckpointTS，读取 TiKV 中的 schema 信息，确定需要被同步的表。
6. 读取各 Processor 当前同步的表的列表，分发需要添加的表。
7. 更新进度信息。

停止 TiCDC 节点

通常来说，停止 TiCDC 节点是为了对 TiCDC 进行升级，或者对 TiCDC 所在的节点进行一些计划的维护操作。停止 TiCDC 节点的流程如下：

1. 节点收到停止的信号。
2. 节点将自己的服务状态置为不可用状态。
3. 节点停止接收新的复制表任务。
4. 通知 Owner 节点将自己节点的数据复制任务转移到其他节点。
5. 复制任务转移到其他节点后，该节点停止完成。

13.9.4.5 TiCDC Server 配置

本文介绍 TiCDC 的命令行参数和配置文件定义。

13.9.4.5.1 `cdc server` 命令行参数说明

对于 `cdc server` 命令中可用选项解释如下：

- `addr`：TiCDC 的监听地址，提供服务的 HTTP API 查询地址和 Prometheus 查询地址，默认为 `127.0.0.1:8300`。
- `advertise-addr`：TiCDC 对外开放地址，供客户端访问。如果未设置该参数值，地址默认与 `addr` 相同。
- `pd`：TiCDC 监听的 PD 节点地址，用 , 来分隔多个 PD 节点地址。
- `config`：可选项，表示 TiCDC 使用的配置文件地址。TiCDC 从 v5.0.0 开始支持该选项，TiUP 从 v1.4.0 开始支持在部署 TiCDC 时使用该配置。配置文件的格式说明详见：TiCDC Changefeed 配置参数
- `data-dir`：指定 TiCDC 使用磁盘储存文件时的目录。目前 TiCDC 内部的排序引擎和 redo log 等特性会使用该目录储存临时文件，建议确保该目录所在设备的可用空间大于等于 500 GiB。如果你使用 TiUP，本选项可以通过配置`cdc_servers` 中的 `data_dir` 来指定或默认使用 `global` 中 `data_dir` 路径。
- `gc-ttl`：TiCDC 在 PD 设置的服务级别 GC safepoint 的 TTL (Time To Live) 时长，和 TiCDC 同步任务所能够停滞的时长。单位为秒，默认值为 86400，即 24 小时。注意：TiCDC 同步任务的停滞会影响 TiCDC GC safepoint 的推进，即会影响上游 TiDB GC 的推进，详情可以参考TiCDC GC safepoint 的完整行为。
- `log-file`：TiCDC 进程运行时日志的输出地址，未设置时默认为标准输出 (stdout)。
- `log-level`：TiCDC 进程运行时的日志级别，默认为 `"info"`。
- `ca`：TiCDC 创建 TLS 连接时使用的 CA 证书文件路径，PEM 格式，可选。
- `cert`：TiCDC 创建 TLS 连接时使用的证书文件路径，PEM 格式，可选。
- `cert-allowed-cn`：TiCDC 创建 TLS 连接时使用的通用名称文件路径，可选。

- key：TiCDC 创建 TLS 连接时使用的证书密钥文件路径，PEM 格式，可选。
- tz：TiCDC 服务使用的时区。TiCDC 在内部转换 TIMESTAMP 等时间数据类型和向下游同步数据时使用该时区，默认为进程运行本地时区。(注意如果同时指定 tz 参数和 sink-uri 中的 time-zone 参数，TiCDC 进程内部使用 tz 指定的时区，sink 向下游执行时使用 time-zone 指定的时区)
- cluster-id：TiCDC 集群的 ID。可选，默认值为 default。cluster-id 是 TiCDC 集群的唯一标识，拥有相同 cluster-id 的 TiCDC 节点同属一个集群。长度最大为 128，需要符合正则表达式 ^[a-zA-Z0-9]+(-[a-zA-Z0-9]+)*$，且不能是以下值：owner，capture，task，changefeed，job，meta。

13.9.4.5.2 cdc server 配置文件说明

对于 cdc server 命令中 config 参数指定的配置文件说明如下：

```
addr = "127.0.0.1:8300"
advertise-addr = ""
log-file = ""
log-level = "info"
data-dir = ""
gc-ttl = 86400 # 24 h
tz = "System"
cluster-id = "default"

[security]
  ca-path = ""
  cert-path = ""
  key-path = ""


capture-session-ttl = 10 # 10s
owner-flush-interval = 50000000 # 50 ms
processor-flush-interval = 50000000 # 50 ms
per-table-memory-quota = 10485760 # 10 MiB

[log]
  error-output = "stderr"
  [log.file]
    max-size = 300 # 300 MiB
    max-days = 0
    max-backups = 0


#### [kv-client]
####   worker-concurrent = 8
####   worker-pool-size = 0
####   region-scan-limit = 40
####   region-retry-duration = 60000000000
```

13.9.4.6 TiCDC Changefeed 命令行参数和配置参数

13.9.4.6.1 TiCDC Changefeed 命令行参数

本章节将以创建同步任务为例，介绍 TiCDC Changefeed 的命令行参数：

```
cdc cli changefeed create --server=http://10.0.10.25:8300 --sink-uri="mysql://root:123456@127
    ↪ .0.0.1:3306/" --changefeed-id="simple-replication-task"
```

```
Create changefeed successfully!
ID: simple-replication-task
Info: {"upstream_id":7178706266519722477,"namespace":"default","id":"simple-replication-task","
    ↪ sink_uri":"mysql://root:xxxxx@127.0.0.1:4000/?time-zone=","create_time":"2022-12-19T15
    ↪ :05:46.679218+08:00","start_ts":438156275634929669,"engine":"unified","config":{"
    ↪ case_sensitive":true,"enable_old_value":true,"force_replicate":false,"
    ↪ ignore_ineligible_table":false,"check_gc_safe_point":true,"enable_sync_point":true,"
    ↪ bdr_mode":false,"sync_point_interval":30000000000,"sync_point_retention":3600000000000,"
    ↪ filter":{"rules":["test.*"],"event_filters":null},"mounter":{"worker_num":16},"sink":{"
    ↪ protocol":"","schema_registry":"","csv":{"delimiter":",","quote":"\"","null":"\\N","
    ↪ include_commit_ts":false},"column_selectors":null,"transaction_atomicity":"none","
    ↪ encoder_concurrency":16,"terminator":"\r\n","date_separator":"none","
    ↪ enable_partition_separator":false},"consistent":{"level":"none","max_log_size":64,"
    ↪ flush_interval":2000,"storage":""}},"state":"normal","creator_version":"v6.5.0"}
```

- `--changefeed-id`：同步任务的 ID，格式需要符合正则表达式 `^[a-zA-Z0-9]+(\-[a-zA-Z0-9]+)*$`。如果不指定该 ID，TiCDC 会自动生成一个 UUID（version 4 格式）作为 ID。
- `--sink-uri`：同步任务下游的地址，需要按照以下格式进行配置，目前 scheme 支持 `mysql`、`tidb` 和 `kafka`。

  ```
  [scheme]://[userinfo@][host]:[port][/path]?[query_parameters]
  ```

  URI 中包含特殊字符时，如 `! * ' ( ) ; : @ & = + $ , / ? % # [ ]`，需要对 URI 特殊字符进行转义处理。你可以在 URI Encoder 中对 URI 进行转义。
- `--start-ts`：指定 changefeed 的开始 TSO。TiCDC 集群将从这个 TSO 开始拉取数据。默认为当前时间。
- `--target-ts`：指定 changefeed 的目标 TSO。TiCDC 集群拉取数据直到这个 TSO 停止。默认为空，即 TiCDC 不会自动停止。
- `--config`：指定 changefeed 配置文件。

13.9.4.6.2 TiCDC Changefeed 配置文件说明

本章节详细介绍了同步任务的配置。

```
#### 指定该 Changefeed 在 Capture Server 中内存配额的上限。对于超额使用部分，
#### 会在运行中被 Go runtime 优先回收。默认值为 `1073741824`，即 1 GB。
#### memory-quota = 1073741824
```

```
#### 指定配置文件中涉及的库名、表名是否为大小写敏感
#### 该配置会同时影响 filter 和 sink 相关配置，默认为 true
case-sensitive = true

#### 是否输出 old value，从 v4.0.5 开始支持，从 v5.0 开始默认为 true
enable-old-value = true

#### 是否开启 Syncpoint 功能，从 v6.3.0 开始支持，该功能默认关闭。
#### 从 v6.4.0 开始，使用 Syncpoint 功能需要同步任务拥有下游集群的 SYSTEM_VARIABLES_ADMIN 或者
    ↪ SUPER 权限。
#### 注意：该参数只有当下游为 Kafka 或存储服务时，才会生效。
#### enable-sync-point = false

#### Syncpoint 功能对齐上下游 snapshot 的时间间隔
#### 配置格式为 h m s，例如 "1h30m30s"
#### 默认值为 10m，最小值为 30s
#### 注意：该参数只有当下游为 Kafka 或存储服务时，才会生效。
#### sync-point-interval = "5m"

#### Syncpoint 功能在下游表中保存的数据的时长，超过这个时间的数据会被清理
#### 配置格式为 h m s，例如 "24h30m30s"
#### 默认值为 24h
#### 注意：该参数只有当下游为 Kafka 或存储服务时，才会生效。
#### sync-point-retention = "1h"

[mounter]
#### mounter 解码 KV 数据的线程数，默认值为 16
#### worker-num = 16

[filter]
#### 忽略指定 start_ts 的事务
#### ignore-txn-start-ts = [1, 2]

#### 过滤器规则
#### 过滤规则语法：https://docs.pingcap.com/zh/tidb/stable/table-filter#表库过滤语法
rules = ['*.*', '!test.*']

#### 忽略特定 start_ts 的事务
#### 默认值为空列表。
#### IgnoreTxnStartTs = []

#### 事件过滤器规则
#### 事件过滤器的详细配置规则可参考：https://docs.pingcap.com/zh/tidb/stable/ticdc-filter
#### 第一个事件过滤器规则
```

```
[[filter.event-filters]]
matcher = ["test.worker"] # matcher 是一个白名单，表示该过滤规则只应用于 test 库中的 worker 表
ignore-event = ["insert"] # 过滤掉 insert 事件
ignore-sql = ["^drop", "add column"] # 过滤掉以 "drop" 开头或者包含 "add column" 的 DDL
ignore-delete-value-expr = "name = 'john'" # 过滤掉包含 name = 'john' 条件的 delete DML
ignore-insert-value-expr = "id >= 100" # 过滤掉包含 id >= 100 条件的 insert DML
ignore-update-old-value-expr = "age < 18" # 过滤掉旧值 age < 18 的 update DML
ignore-update-new-value-expr = "gender = 'male'" # 过滤掉新值 gender = 'male' 的 update DML

#### 第二个事件过滤器规则
[[filter.event-filters]]
matcher = ["test.fruit"] # 该事件过滤器只应用于 test.fruit 表
ignore-event = ["drop table"] # 忽略 drop table 事件
ignore-sql = ["delete"] # 忽略 delete DML
ignore-insert-value-expr = "price > 1000 and origin = 'no where'" # 忽略包含 price > 1000 和
    ↪ origin = 'no where' 条件的 insert DML

[scheduler]
#### 将表按 Region 个数划分成多个同步范围，这些范围可由多个 TiCDC 节点同步。
#### 注意：该功能只在 Kafka changefeed 上生效，暂不支持 MySQL changefeed。
#### 默认为 "false"。设置为 "true" 以打开该功能。
enable-table-across-nodes = false

#### 打开该功能后，该功能会对 Region 个数大于 `region-threshold` 值的表生效。
region-threshold = 100000

#### 打开该功能后，该功能会对每分钟修改行数大于 `write-key-threshold` 值的表生效。
#### 注意：
#### * `write-key-threshold` 参数默认值为 0，代表该功能默认不会按表的修改行数来切分表的同步范围。
#### * 你可以根据集群负载来配置该参数，如 30000，代表当表每分钟的更新行数超过 30000 时，
    ↪ 该功能将会切分表的同步范围。
#### * 当 `region-threshold` 和 `write-key-threshold` 同时配置时，
####   TiCDC 将优先检查修改行数是否大于 `write-key-threshold`，
####   如果不超过，则再检查 Region 个数是否大于 `region-threshold`。
write-key-threshold = 0

[sink]
#### 对于 MQ 类的 Sink，可以通过 dispatchers 配置 event 分发器
#### 支持 partition 及 topic（从 v6.1 开始支持）两种 event 分发器。二者的详细说明见下一节。
#### matcher 的匹配语法和过滤器规则语法相同，matcher 匹配规则的详细说明见下一节。
#### 注意：该参数只有当下游为消息队列时，才会生效。
#### dispatchers = [
####     {matcher = ['test1.*', 'test2.*'], topic = "Topic 表达式 1", partition = "ts" },
####     {matcher = ['test3.*', 'test4.*'], topic = "Topic 表达式 2", partition = "index-value" },
####     {matcher = ['test1.*', 'test5.*'], topic = "Topic 表达式 3", partition = "table"},
```

```
####    {matcher = ['test6.*'], partition = "ts"}
#### ]

#### protocol 用于指定传递到下游的协议格式
#### 当下游类型是 Kafka 时，支持 canal-json、avro 两种协议。
#### 当下游类型是存储服务时，目前仅支持 canal-json、csv 两种协议。
#### 注意：该参数只有当下游为 Kafka 或存储服务时，才会生效。
#### protocol = "canal-json"

#### 以下三个配置项仅在同步到存储服务的 sink 中使用，在 MQ 和 MySQL 类 sink 中无需设置。
#### 换行符，用来分隔两个数据变更事件。默认值为空，表示使用 "\r\n" 作为换行符。
#### terminator = ''

#### 文件路径的日期分隔类型。可选类型有 `none`、`year`、`month` 和 `day`。默认值为 `none`，
    ↪ 即不使用日期分隔。详见 <https://docs.pingcap.com/zh/tidb/dev/ticdc-sink-to-cloud-storage#
    ↪ 数据变更记录>。
#### 注意：该参数只有当下游为存储服务时，才会生效。
date-separator = 'none'

#### 是否使用 partition 作为分隔字符串。默认值为 true，即一张表中各个 partition
    ↪ 的数据会分不同的目录来存储。建议保持该配置项为 true 以避免下游分区表可能丢数据的问题 <
    ↪ https://github.com/pingcap/tiflow/issues/8581>。使用示例详见 <https://docs.pingcap.com/zh
    ↪ /tidb/dev/ticdc-sink-to-cloud-storage#数据变更记录>。
#### 注意：该参数只有当下游为存储服务时，才会生效。
enable-partition-separator = true

#### Schema 注册表的 URL。
#### 注意：该参数只有当下游为消息队列时，才会生效。
#### schema-registry = "http://localhost:80801/subjects/{subject-name}/versions/{version-number}/
    ↪ schema"

#### 编码数据时所用编码器的线程数。
#### 默认值为 16。
#### 注意：该参数只有当下游为消息队列时，才会生效。
#### encoder-concurrency = 16

#### 是否开启 Kafka Sink V2。Kafka Sink V2 内部使用 kafka-go 实现。
#### 默认值为 false。
#### 注意：该参数只有当下游为消息队列时，才会生效。
#### enable-kafka-sink-v2 = false

#### 从 v6.5.0 开始，TiCDC 支持以 CSV 格式将数据变更记录保存至存储服务中，在 MQ 和 MySQL 类 sink
    ↪ 中无需设置。
[sink.csv]
#### 字段之间的分隔符。必须为 ASCII 字符，默认值为 `,`。
```

```
delimiter = ','
#### 用于包裹字段的引号字符。空值代表不使用引号字符。默认值为 `"`。
quote = '"'
#### CSV 中列为 NULL 时将以什么字符来表示。默认值为 `\N`。
null = '\N'
#### 是否在 CSV 行中包含 commit-ts。默认值为 false。
#### include-commit-ts = false

#### consistent 中的字段用于配置 Changefeed 的数据一致性。详细的信息，请参考 <https://docs.
    ↪ pingcap.com/tidb/stable/ticdc-sink-to-mysql#eventually-consistent-replication-in-disaster
    ↪ -scenarios>。
#### 注意：一致性相关参数只有当下游为数据库并且开启 redo log 功能时，才会生效。
[consistent]
#### 数据一致性级别。默认值为 "none"，可选值为 "none" 和 "eventual"。
#### 设置为 "none" 时将关闭 redo log。
level = "none"
#### redo log 的最大日志大小，单位为 MB。默认值为 64。
max-log-size = 64
#### 两次 redo log 刷新的时间间隔，单位为毫秒。默认值为 2000。
flush-interval = 2000
#### redo log 使用存储服务的 URI。默认值为空。
storage = ""
#### 是否将 redo log 存储到文件中。默认值为 false。
use-file-backend = false
```

13.9.4.7 输出数据协议

13.9.4.7.1 TiCDC Avro Protocol

Avro 是由 Apache Avro™ 定义的一种数据交换格式协议，Confluent Platform 选择它作为默认的数据交换格式。通过本文，你可以了解 TiCDC 对 Avro 数据格式的实现，包括 TiDB 扩展字段、Avro 数据格式定义，以及和 Confluent Schema Registry 的交互。

使用 Avro

当使用 Message Queue (MQ) 作为下游 Sink 时，你可以在 `sink-uri` 中指定使用 Avro。TiCDC 获取 TiDB 的 DML 事件，并将这些事件封装到 Avro Message，然后发送到下游。当 Avro 检测到 schema 变化时，会向 Schema Registry 注册最新的 schema。

使用 Avro 时的配置样例如下所示：

```
cdc cli changefeed create --server=http://127.0.0.1:8300 --changefeed-id="kafka-avro" --sink-uri
    ↪ ="kafka://127.0.0.1:9092/topic-name?protocol=avro" --schema-registry=http
    ↪ ://127.0.0.1:8081 --config changefeed_config.toml
```

```
[sink]
dispatchers = [
```

```
 {matcher = ['*.*'], topic = "tidb_{schema}_{table}"},
]
```

`--schema-registry` 的值支持 https 协议和 username:password 认证，比如`--schema-registry=https://username:password@schema-registry-uri.com`，username 和 password 必须经过 URL 编码。

TiDB 扩展字段

默认情况下，Avro 只收集在 DML 事件中发生数据变更的行的所有数据信息，不收集数据变更的类型和 TiDB 专有的 CommitTS 事务唯一标识信息。为了解决这个问题，TiCDC 在 Avro 协议格式中附加了 TiDB 扩展字段。当 `sink-uri` 中设置 `enable-tidb-extension` 为 `true`（默认为 `false`）后，TiCDC 生成 Avro 消息时会新增三个字段：

- `_tidb_op`：DML 的类型，“c”表示插入，“u”表示更新。
- `_tidb_commit_ts`：事务唯一标识信息。
- `_tidb_commit_physical_time`：事务标识信息中现实时间的时间戳。

配置样例如下所示：

```
cdc cli changefeed create --server=http://127.0.0.1:8300 --changefeed-id="kafka-avro-enable-
    ↪ extension" --sink-uri="kafka://127.0.0.1:9092/topic-name?protocol=avro&enable-tidb-
    ↪ extension=true" --schema-registry=http://127.0.0.1:8081 --config changefeed_config.toml
```

```
shell [sink] dispatchers = [ {matcher = ['*.*'], topic = "tidb_{schema}_{table}"}, ]
```

数据格式定义

TiCDC 会将一个 DML 事件转换为一个 kafka 事件，其中事件的 key 和 value 都按照 Avro 协议进行编码。

Key 数据格式

```
{
    "name":"{{TableName}}",
    "namespace":"{{Namespace}}",
    "type":"record",
    "fields":[
        {{ColumnValueBlock}},
        {{ColumnValueBlock}},
    ]
}
```

- `{{TableName}}` 是事件来源表的名称。
- `{{Namespace}}` 是 Avro 的命名空间。
- `{{ColumnValueBlock}}` 是每列数据的格式定义。

Key 中的 `fields` 只包含主键或唯一索引列。

Value 数据格式

```
{
    "name":"{{TableName}}",
    "namespace":"{{Namespace}}",
    "type":"record",
    "fields":[
        {{ColumnValueBlock}},
        {{ColumnValueBlock}},
    ]
}
```

Value 数据格式默认与 Key 数据格式相同，但是 Value 的 `fields` 中包含了所有的列，而不仅仅是主键列。

如果开启了TiDB 扩展字段，那么 Value 数据格式将会变成：

```
{
    "name":"{{TableName}}",
    "namespace":"{{Namespace}}",
    "type":"record",
    "fields":[
        {{ColumnValueBlock}},
        {{ColumnValueBlock}},
        {
            "name":"_tidb_op",
            "type":"string"
        },
        {
            "name":"_tidb_commit_ts",
            "type":"long"
        },
        {
            "name":"_tidb_commit_physical_time",
            "type":"long"
        }
    ]
}
```

相比于不打开 TiDB 扩展字段选项，多出了 `_tidb_op`, `_tidb_commit_ts`, `_tidb_commit_physical_time` 三个字段的定义。

Column 数据格式

Column 数据格式即 Key/Value 数据格式中的 `{{ColumnValueBlock}}` 部分，TiCDC 会根据 SQL Type 生成对应的 Column 数据格式。基础的 Column 数据格式是：

```
{
    "name":"{{ColumnName}}",
    "type":{
```

```
        "connect.parameters":{
            "tidb_type":"{{TIDB_TYPE}}"
        },
        "type":"{{AVRO_TYPE}}"
    }
}
```

如果一列可以为 NULL，那么 Column 数据格式是：

```
{
    "default":null,
    "name":"{{ColumnName}}",
    "type":[
        "null",
        {
            "connect.parameters":{
                "tidb_type":"{{TIDB_TYPE}}"
            },
            "type":"{{AVRO_TYPE}}"
        }
    ]
}
```

- `{{ColumnName}}` 表示列名。
- `{{TIDB_TYPE}}` 表示对应到 TiDB 中的类型，与原始的 SQL Type 不是一一对应关系。
- `{{AVRO_TYPE}}` 表示 avro spec 中的类型。

SQL TYPE	TIDB_TYPE	AVRO_TYPE	说明
BOOL	INT	int	
TINYINT	INT	int	当 TINYINT 为无符号值时，TIDB_TYPE 为 INT UNSIGNED。
SMALLINT	INT	int	当 SMALLINT 为无符号值时，TIDB_TYPE 为 INT UNSIGNED.
MEDIUMINT	INT	int	当 MEDIUMINT 为无符号值时，TIDB_TYPE 为 INT UNSIGNED。
INT	INT	int	当 INT 为无符号值时，TIDB_TYPE 为 INT UNSIGNED，AVRO_TYPE 为 long。
BIGINT	BIGINT	long	当 BIGINT 为无符号值时，TIDB_TYPE 为 BIGINT UNSIGNED。当 `avro-bigint-unsigned-handling-mode` 为 string 时，AVRO_TYPE 也为 string。
TINYBLOB	BLOB	bytes	
BLOB	BLOB	bytes	
MEDIUMBLOB	BLOB	bytes	
LONGBLOB	BLOB	bytes	
BINARY	BLOB	bytes	
VARBINARY	BLOB	bytes	
TINYTEXT	TEXT	string	
TEXT	TEXT	string	

SQL TYPE	TIDB_TYPE	AVRO_TYPE	说明
MEDIUMTEXT	TEXT	string	
LONGTEXT	TEXT	string	
CHAR	TEXT	string	
VARCHAR	TEXT	string	
FLOAT	FLOAT	double	
DOUBLE	DOUBLE	double	
DATE	DATE	string	
DATETIME	DATETIME	string	
TIMESTAMP	TIMESTAMP	string	
TIME	TIME	string	
YEAR	YEAR	int	
BIT	BIT	bytes	
JSON	JSON	string	
ENUM	ENUM	string	
SET	SET	string	
DECIMAL	DECIMAL	bytes	当 `avro-decimal-handling-mode` 为 string 时，AVRO_TYPE 也为 string。

对于 Avro 协议，另外两个 `sink-uri` 参数 `avro-decimal-handling-mode` 和 `avro-bigint-unsigned-handling-mode` 也会影响 Column 数据格式:

- `avro-decimal-handling-mode` 决定了如何处理 DECIMAL 字段，它有两个选项：
  - string：Avro 将 DECIMAL 字段以 string 的方式处理。
  - precise：Avro 将 DECIMAL 字段以字节的方式处理。
- `avro-bigint-unsigned-handling-mode` 决定了如何处理 BIGINT UNSIGNED 字段，它有两个选项：
  - string：Avro 将 BIGINT UNSIGNED 字段以 string 的方式处理。
  - long：Avro 将 BIGINT UNSIGNED 字段以 64 位有符号整数处理，大于 9223372036854775807 的值会发生溢出。

配置样例如下所示：

```
cdc cli changefeed create --server=http://127.0.0.1:8300 --changefeed-id="kafka-avro-string-
    ↪ option" --sink-uri="kafka://127.0.0.1:9092/topic-name?protocol=avro&avro-decimal-handling
    ↪ -mode=string&avro-bigint-unsigned-handling-mode=string" --schema-registry=http
    ↪ ://127.0.0.1:8081 --config changefeed_config.toml
```

```
[sink]
dispatchers = [
 {matcher = ['*.*'], topic = "tidb_{schema}_{table}"},
]
```

大多数的 SQL Type 都会映射成基础的 Column 数据格式，但有一些类型会在基础数据格式上拓展，提供更多的信息。

BIT(64)

```
{
    "name":"{{ColumnName}}",
    "type":{
        "connect.parameters":{
            "tidb_type":"BIT",
            "length":"64"
        },
        "type":"bytes"
    }
}
```

ENUM/SET(a,b,c)

```
{
    "name":"{{ColumnName}}",
    "type":{
        "connect.parameters":{
            "tidb_type":"ENUM/SET",
            "allowed":"a,b,c"
        },
        "type":"string"
    }
}
```

DECIMAL(10, 4)

```
{
    "name":"{{ColumnName}}",
    "type":{
        "connect.parameters":{
            "tidb_type":"DECIMAL",
        },
        "logicalType":"decimal",
        "precision":10,
        "scale":4,
        "type":"bytes"
    }
}
```

DDL 事件与 Schema 变更

Avro 并不会向下游生成 DDL 事件。Avro 会在每次 DML 事件发生时检测是否发生 schema 变更，如果发生了 schema 变更，Avro 会生成新的 schema，并尝试向 Schema Registry 注册。注册时，Schema Registry 会做兼容性检测，如果

此次 schema 变更没有通过兼容性检测，注册将会失败，TiCDC 并不会尝试解决 schema 的兼容性问题。

同时，即使 schema 变更通过兼容性检测并成功注册新版本，数据的生产者和消费者可能仍然需要升级才能正确工作。

比如，Confluent Schema Registry 默认的兼容性策略是 BACKWARD，在这种策略下，如果你在源表增加一个非空列，Avro 在生成新 schema 向 Schema Registry 注册时将会因为兼容性问题失败，这个时候 changefeed 将会进入 error 状态。

如需了解更多 schema 相关信息，请参阅 Schema Registry 的相关文档。

Topic 分发

Schema Registry 支持三种 Subject Name Strategy：TopicNameStrategy、 RecordNameStrategy 和 TopicRecordNameStrategy。目前 TiCDC Avro 只支持 TopicNameStrategy 一种，这意味着一个 kafka topic 只能接收一种数据格式的数据，所以 TiCDC Avro 禁止将多张表映射到同一个 topic。在创建 changefeed 时，如果配置的分发规则中，topic 规则不包含 `{schema}` 和 `{table}` 占位符，将会报错。

兼容性说明

- 在升级 TiCDC 集群到 v7.0.0 时，如果使用 Avro 同步的表包含 FLOAT 类型数据，请在升级前手动调整 Confluent Schema Registry 的兼容性策略为 None，使 changefeed 能够成功更新 schema。否则，在升级之后 changefeed 将无法更新 schema 并进入错误状态。详情请参考 GitHub issue #8490。

13.9.4.7.2 TiCDC Canal-JSON Protocol

Canal-JSON 是由 Alibaba Canal 定义的一种数据交换格式协议。通过本文，你可以了解 TiCDC 对 Canal-JSON 数据格式的实现，包括 TiDB 扩展字段、Canal-JSON 数据格式定义，以及和官方实现进行对比等相关内容。

使用 Canal-JSON

当使用 MQ (Message Queue) 作为下游 Sink 时，你可以在 `sink-uri` 中指定使用 Canal-JSON，TiCDC 将以 Event 为基本单位封装构造 Canal-JSON Message，向下游发送 TiDB 的数据变更事件。

Event 分为三类：

- DDL Event：代表 DDL 变更记录，在上游成功执行 DDL 语句后发出，DDL Event 会被发送到索引为 0 的 MQ Partition。
- DML Event：代表一行数据变更记录，在行变更发生时该类 Event 被发出，包含变更后该行的相关信息。
- WATERMARK Event：代表一个特殊的时间点，表示在这个时间点前收到的 Event 是完整的。仅适用于 TiDB 扩展字段，当你在 `sink-uri` 中设置 `enable-tidb-extension=true` 时生效。

使用 `Canal-JSON` 时的配置样例如下所示：

```
cdc cli changefeed create --server=http://127.0.0.1:8300 --changefeed-id="kafka-canal-json" --
    ↪ sink-uri="kafka://127.0.0.1:9092/topic-name?kafka-version=2.4.0&protocol=canal-json"
```

TiDB 扩展字段

Canal-JSON 协议本是为 MySQL 设计的，其中并不包含 TiDB 专有的 CommitTS 事务唯一标识等重要字段。为了解决这个问题，TiCDC 在 Canal-JSON 协议格式中附加了 TiDB 扩展字段。在 `sink-uri` 中设置 `enable-tidb-extension` 为 `true`（默认为 `false`）后，TiCDC 生成 Canal-JSON 消息时的行为如下：

- TiCDC 发送的 DML Event 和 DDL Event 类型消息中，将会含有一个名为 `_tidb` 的字段。
- TiCDC 将会发送 WATERMARK Event 消息。·

配置样例如下所示：

```
cdc cli changefeed create --server=http://127.0.0.1:8300 --changefeed-id="kafka-canal-json-enable
    ↪ -tidb-extension" --sink-uri="kafka://127.0.0.1:9092/topic-name?kafka-version=2.4.0&
    ↪ protocol=canal-json&enable-tidb-extension=true"
```

Message 格式定义

下面介绍 DDL Event、 DML Event 和 WATERMARK Event 的格式定义，以及消费端的数据解析。

DDL Event

TiCDC 会把一个 DDL Event 编码成如下 Canal-JSON 格式：

```
{
    "id": 0,
    "database": "test",
    "table": "",
    "pkNames": null,
    "isDdl": true,
    "type": "QUERY",
    "es": 1639633094670,
    "ts": 1639633095489,
    "sql": "drop database if exists test",
    "sqlType": null,
    "mysqlType": null,
    "data": null,
    "old": null,
    "_tidb": {     // TiDB 的扩展字段
        "commitTs": 163963309467037594
    }
}
```

以上 JSON 数据的字段解释如下：

字段	类型	说明
id	Number	TiCDC 默认值为 0
database	String	Row 所在的 Database 的名字
table	String	Row 所在的 Table 的名字
pkNames	Array	组成 primary key 的所有列的名字
isDdl	Bool	该条消息是否为 DDL 事件
type	String	Canal-JSON 定义的事件类型
es	Number	产生该条消息的事件发生时的 13 位（毫秒级）时间戳
ts	Number	TiCDC 生成该条消息时的 13 位（毫秒级）时间戳
sql	String	当 isDdl 为 true 时，记录对应的 DDL 语句

字段	类型	说明
sqlType	Object	当 isDdl 为 false 时，记录每一列数据类型在 Java 中的类型表示
mysqlType	object	当 isDdl 为 false 时，记录每一列数据类型在 MySQL 中的类型表示
data	Object	当 isDdl 为 false 时，记录每一列的名字及其数据值
old	Object	仅当该条消息由 Update 类型事件产生时，记录每一列的名字，和 Update 之前的数据值
_tidb	Object	TiDB 扩展字段，仅当 `enable-tidb-extension` 为 true 时才会存在。其中的 `commitTs` 值为造成 Row 变更的事务的 TSO

DML Event

对于一行 DML 数据变更事件，TiCDC 会将其编码成如下形式:

```
{
    "id": 0,
    "database": "test",
    "table": "tp_int",
    "pkNames": [
        "id"
    ],
    "isDdl": false,
    "type": "INSERT",
    "es": 1639633141221,
    "ts": 1639633142960,
    "sql": "",
    "sqlType": {
        "c_bigint": -5,
        "c_int": 4,
        "c_mediumint": 4,
        "c_smallint": 5,
        "c_tinyint": -6,
        "id": 4
    },
    "mysqlType": {
        "c_bigint": "bigint",
        "c_int": "int",
        "c_mediumint": "mediumint",
        "c_smallint": "smallint",
        "c_tinyint": "tinyint",
        "id": "int"
    },
    "data": [
        {
            "c_bigint": "9223372036854775807",
            "c_int": "2147483647",
```

```
            "c_mediumint": "8388607",
            "c_smallint": "32767",
            "c_tinyint": "127",
            "id": "2"
        }
    ],
    "old": null,
    "_tidb": {     // TiDB 的扩展字段
        "commitTs": 163963314122145239
    }
}
```

WATERMARK Event

仅当 enable-tidb-extension 为 true 时，TiCDC 才会发送 WATERMARK Event，其 type 字段值为 TIDB_WATERMARK。该类型事件具有 _tidb 字段，当前只含有 watermarkTs，其值为该 Event 发送时的 TSO。

当你收到一个该类型的事件，所有 commitTs 小于 watermarkTs 的事件均已发送完毕。因为 TiCDC 提供 At Least Once 语义，可能出现重复发送数据的情况。如果后续收到有 commitTs 小于 watermarkTs 的事件，可以忽略。

WATERMARK Event 的示例如下：

```
{
    "id": 0,
    "database": "",
    "table": "",
    "pkNames": null,
    "isDdl": false,
    "type": "TIDB_WATERMARK",
    "es": 1640007049196,
    "ts": 1640007050284,
    "sql": "",
    "sqlType": null,
    "mysqlType": null,
    "data": null,
    "old": null,
    "_tidb": {     // TiDB 的扩展字段
        "watermarkTs": 429918007904436226
    }
}
```

消费端数据解析

从上面的示例中可知，Canal-JSON 具有统一的数据格式，针对不同的事件类型，有不同的字段填充规则。消费者可以使用统一的方法对该 JSON 格式的数据进行解析，然后通过判断字段值的方式，来确定具体事件类型：

- 当 isDdl 为 true 时，该消息含有一条 DDL Event。

- 当 `isDdl` 为 false 时，需要对 `type` 字段加以判断。如果 `type` 为 `TIDB_WATERMARK`，可得知其为 WATERMARK Event，否则就是 DML Event。

字段说明

Canal-JSON 格式会在 `mysqlType` 字段和 `sqlType` 字段中记录对应的数据类型。

MySQL Type 字段

Canal-JSON 格式会在 `mysqlType` 字段中记录每一列的 MySQL Type 的字符串表示。相关详情可以参考TiDB Data Types。

SQL Type 字段

Canal-JSON 格式会在 `sqlType` 字段中记录每一列的 Java SQL Type，即每条数据在 JDBC 中对应的数据类型，其值可以通过 MySQL Type 和具体数据值计算得到。具体对应关系如下:

MySQL Type	Java SQL Type Code
Boolean	-6
Float	7
Double	8
Decimal	3
Char	1
Varchar	12
Binary	2004
Varbinary	2004
Tinytext	2005
Text	2005
Mediumtext	2005
Longtext	2005
Tinyblob	2004
Blob	2004
Mediumblob	2004
Longblob	2004
Date	91
Datetime	93
Timestamp	93
Time	92
Year	12
Enum	4
Set	-7
Bit	-7
JSON	12

整数类型

你需要考虑整数类型是否有 Unsigned 约束，以及当前取值大小，分别对应不同的 Java SQL Type Code。如下表所示。

MySQL Type String	Value Range	Java SQL Type Code
tinyint	[-128, 127]	-6
tinyint unsigned	[0, 127]	-6
tinyint unsigned	[128, 255]	5
smallint	[-32768, 32767]	5
smallint unsigned	[0, 32767]	5
smallint unsigned	[32768, 65535]	4
mediumint	[-8388608, 8388607]	4
mediumint unsigned	[0, 8388607]	4
mediumint unsigned	[8388608, 16777215]	4
int	[-2147483648, 2147483647]	4
int unsigned	[0, 2147483647]	4
int unsigned	[2147483648, 4294967295]	-5
bigint	[-9223372036854775808, 9223372036854775807]	-5
bigint unsigned	[0, 9223372036854775807]	-5
bigint unsigned	[9223372036854775808, 18446744073709551615]	3

TiCDC 涉及的 Java SQL Type 及其 Code 映射关系如下表所示。

Java SQL Type	Java SQL Type Code
CHAR	1
DECIMAL	3
INTEGER	4
SMALLINT	5
REAL	7
DOUBLE	8
VARCHAR	12
DATE	91
TIME	92
TIMESTAMP	93
BLOB	2004
CLOB	2005
BIGINT	-5
TINYINT	-6
Bit	-7

想要了解 Java SQL Type 的更多信息，请参考 Java SQL Class Types。

TiCDC Canal-JSON 和 Canal 官方实现对比

TiCDC 对 Canal-JSON 数据格式的实现，包括 `Update` 类型事件和 `mysqlType` 字段，和官方有些许不同。主要差异见下表。

差异点	TiCDC	Canal
Update 类型事件	old 字段包含所有列数据	old 字段仅包含被修改的列数据
mysqlType 字段	对于含有参数的类型，没有类型参数信息	对于含有参数的类型，会包含完整的参数信息

Update 类型事件

对于 Update 类型事件，Canal 官方实现中，old 字段仅包含被修改的列数据，而 TiCDC 的实现则包含所有列数据。

假设在上游 TiDB 按顺序执行如下 SQL 语句:

```
create table tp_int
(
    id          int auto_increment,
    c_tinyint   tinyint   null,
    c_smallint  smallint  null,
    c_mediumint mediumint null,
    c_int       int       null,
    c_bigint    bigint    null,
    constraint pk
        primary key (id)
);

insert into tp_int(c_tinyint, c_smallint, c_mediumint, c_int, c_bigint)
values (127, 32767, 8388607, 2147483647, 9223372036854775807);

update tp_int set c_int = 0, c_tinyint = 0 where c_smallint = 32767;
```

对于 update 语句，TiCDC 将会输出一条 type 为 UPDATE 的事件消息，如下所示。该 update 语句仅对 c_int 和 c_tinyint 两列进行了修改。输出事件消息的 old 字段，则包含所有列数据。

```
{
    "id": 0,
    ...
    "type": "UPDATE",
    ...
    "sqlType": {
        ...
    },
    "mysqlType": {
        ...
    },
    "data": [
        {
            "c_bigint": "9223372036854775807",
            "c_int": "0",
```

```
            "c_mediumint": "8388607",
            "c_smallint": "32767",
            "c_tinyint": "0",
            "id": "2"
        }
    ],
    "old": [                                // TiCDC 输出事件消息的 `old` 字段，则包含所有列数据
        ↪ 。
        {
            "c_bigint": "9223372036854775807",
            "c_int": "2147483647",          // 修改的列
            "c_mediumint": "8388607",
            "c_smallint": "32767",
            "c_tinyint": "127",             // 修改的列
            "id": "2"
        }
    ]
}
```

官方 Canal 输出事件消息的 old 字段仅包含被修改的列数据。示例如下。

```
{
    "id": 0,
    ...
    "type": "UPDATE",
    ...
    "sqlType": {
        ...
    },
    "mysqlType": {
        ...
    },
    "data": [
        {
            "c_bigint": "9223372036854775807",
            "c_int": "0",
            "c_mediumint": "8388607",
            "c_smallint": "32767",
            "c_tinyint": "0",
            "id": "2"
        }
    ],
    "old": [                                // Canal 输出事件消息的 `old` 字段，
        ↪ 仅包含被修改的列的数据。
        {
            "c_int": "2147483647",          // 修改的列
```

```
            "c_tinyint": "127",                 // 修改的列
        }
    ]
}
```

mysqlType 字段

对于 mysqlType 字段，Canal 官方实现中，对于含有参数的类型，会包含完整的参数信息，TiCDC 实现则没有类型参数信息。

在下面示例的表定义 SQL 语句中，如 decimal / char / varchar / enum 等类型，都含有参数。对比 TiCDC 和 Canal 官方实现分别生成的 Canal-JSON 格式数据可知，在 mysqlType 字段中的数据，TiCDC 实现只包含基本 MySQL Type。如果业务需要类型参数信息，需要你自行通过其他方式实现。

假设在上游数据库按顺序执行如下 SQL 语句:

```
create table t (
    id     int auto_increment,
    c_decimal    decimal(10, 4) null,
    c_char       char(16)       null,
    c_varchar    varchar(16)    null,
    c_binary     binary(16)     null,
    c_varbinary  varbinary(16) null,

    c_enum enum('a','b','c') null,
    c_set  set('a','b','c')  null,
    c_bit  bit(64)            null,
    constraint pk
        primary key (id)
);

insert into t (c_decimal, c_char, c_varchar, c_binary, c_varbinary, c_enum, c_set, c_bit)
values (123.456, "abc", "abc", "abc", "abc", 'a', 'a,b', b'1000001');
```

TiCDC 输出内容如下：

```
{
    "id": 0,
    ...
    "isDdl": false,
    "sqlType": {
        ...
    },
    "mysqlType": {
        "c_binary": "binary",
        "c_bit": "bit",
        "c_char": "char",
        "c_decimal": "decimal",
```

```
        "c_enum": "enum",
        "c_set": "set",
        "c_varbinary": "varbinary",
        "c_varchar": "varchar",
        "id": "int"
    },
    "data": [
        {
            ...
        }
    ],
    "old": null,
}
```

Canal 官方实现输出内容如下：

```
{
    "id": 0,
    ...
    "isDdl": false,
    "sqlType": {
        ...
    },
    "mysqlType": {
        "c_binary": "binary(16)",
        "c_bit": "bit(64)",
        "c_char": "char(16)",
        "c_decimal": "decimal(10, 4)",
        "c_enum": "enum('a','b','c')",
        "c_set": "set('a','b','c')",
        "c_varbinary": "varbinary(16)",
        "c_varchar": "varchar(16)",
        "id": "int"
    },
    "data": [
        {
            ...
        }
    ],
    "old": null,
}
```

TiCDC Canal-JSON 改动说明

`Delete` 类型事件中 `Old` 字段的变化说明

TiCDC 实现的 Canal-JSON 格式，v5.4.0 及以后版本的实现，和之前的有些许不同，具体如下：

- Delete 类型事件，Old 字段的内容发生了变化。

如下是一个 DELETE 事件的数据内容，在 v5.4.0 前的实现中，“old” 的内容和 “data” 相同，在 v5.4.0 及之后的实现中，“old” 将被设为 null。你可以通过 “data” 字段获取到被删除的数据。

```
{
    "id": 0,
    "database": "test",
    ...
    "type": "DELETE",
    ...
    "sqlType": {
        ...
    },
    "mysqlType": {
        ...
    },
    "data": [
        {
            "c_bigint": "9223372036854775807",
            "c_int": "0",
            "c_mediumint": "8388607",
            "c_smallint": "32767",
            "c_tinyint": "0",
            "id": "2"
        }
    ],
    "old": null,

    // 以下示例是 v5.4.0 之前的实现，`old` 内容等同于 `data` 内容
    "old": [
        {
            "c_bigint": "9223372036854775807",
            "c_int": "0",
            "c_mediumint": "8388607",
            "c_smallint": "32767",
            "c_tinyint": "0",
            "id": "2"
        }
    ]
}
```

13.9.4.7.3 TiCDC Open Protocol

概述

TiCDC Open Protocol 是一种行级别的数据变更通知协议，为监控、缓存、全文索引、分析引擎、异构数据库的主从复制等提供数据源。TiCDC 遵循 TiCDC Open Protocol，向 MQ(Message Queue) 等第三方数据媒介复制 TiDB 的数据变更。

TiCDC Open Protocol 以 Event 为基本单位向下游复制数据变更事件，Event 分为三类：

- Row Changed Event：代表一行的数据变化，在行发生变更时该 Event 被发出，包含变更后该行的相关信息。
- DDL Event：代表 DDL 变更，在上游成功执行 DDL 后发出，DDL Event 会广播到每一个 MQ Partition 中。
- Resolved Event：代表一个特殊的时间点，表示在这个时间点前的收到的 Event 是完整的。

协议约束

- 在绝大多数情况下，一个版本的 Row Changed Event 只会发出一次，但是特殊情况（节点故障、网络分区等）下，同一版本的 Row Changed Event 可能会多次发送。
- 同一张表中的每一个版本第一次发出的 Row Changed Event 在 Event 流中一定是按 TS (timestamp) 顺序递增的。
- Resolved Event 会被周期性的广播到各个 MQ Partition，Resolved Event 意味着任何 TS 小于 Resolved Event TS 的 Event 已经发送给下游。
- DDL Event 将被广播到各个 MQ Partition。
- 一行数据的多个 Row Changed Event 一定会被发送到同一个 MQ Partition 中。

Message 格式定义

一个 Message 中包含一个或多个 Event，按照以下格式排列：

Key:

Offset(Byte)	0~7	8~15	16~(15+ 长度 1)	…	…
参数	协议版本号	长度 1	Event Key1	长度 N	Event KeyN

Value:

Offset(Byte)	0~7	8~(7+ 长度 1)	…	…
参数	长度 1	Event Value1	长度 N	Event ValueN

- 长度N代表第 N 个 Key/Value 的长度
- 长度及协议版本号均为大端序 int64 类型
- 当前协议版本号为 1

Event 格式定义

本部分介绍 Row Changed Event、 DDL Event 和 Resolved Event 的格式定义。

Row Changed Event

- Key:

```
{
    "ts":<TS>,
    "scm":<Schema Name>,
    "tbl":<Table Name>,
    "t":1
}
```

参数	类型	说明
TS	Number	造成 Row 变更的事务的 TS
Schema Name	String	Row 所在的 Schema 的名字
Table Name	String	Row 所在的 Table 的名字

- Value:

  `Insert` 事件，输出新增的行数据。

```
{
    "u":{
        <Column Name>:{
            "t":<Column Type>,
            "h":<Where Handle>,
            "f":<Flag>,
            "v":<Column Value>
        },
        <Column Name>:{
            "t":<Column Type>,
            "h":<Where Handle>,
            "f":<Flag>,
            "v":<Column Value>
        }
    }
}
```

`Update` 事件，输出新增的行数据（“u”）以及修改前的行数据（“p”），仅当 Old Value 特性开启时，才会输出修改前的行数据。

```
{
    "u":{
        <Column Name>:{
            "t":<Column Type>,
            "h":<Where Handle>,
            "f":<Flag>,
            "v":<Column Value>
        },
        <Column Name>:{
```

```
            "t":<Column Type>,
            "h":<Where Handle>,
            "f":<Flag>,
            "v":<Column Value>
        }
    },
    "p":{
        <Column Name>:{
            "t":<Column Type>,
            "h":<Where Handle>,
            "f":<Flag>,
            "v":<Column Value>
        },
        <Column Name>:{
            "t":<Column Type>,
            "h":<Where Handle>,
            "f":<Flag>,
            "v":<Column Value>
        }
    }
}
```

Delete 事件，输出被删除的行数据。当 Old Value 特性开启时，Delete 事件中包含被删除的行数据中的所有列；当 Old Value 特性关闭时，Delete 事件中仅包含HandleKey 列。

```
{
    "d":{
        <Column Name>:{
            "t":<Column Type>,
            "h":<Where Handle>,
            "f":<Flag>,
            "v":<Column Value>
        },
        <Column Name>:{
            "t":<Column Type>,
            "h":<Where Handle>,
            "f":<Flag>,
            "v":<Column Value>
        }
    }
}
```

参数	类型	说明
Column Name	String	列名

参数	类型	说明
Column Type	Number	列类型，详见：Column 的类型码
Where Handle	Bool	表示该列是否可以作为 Where 筛选条件，当该列在表内具有唯一性时，Where Handle 为 true。
Flag	Number	列标志位，详见：列标志位
Column Value	Any	列值

DDL Event

- Key:

```
{
    "ts":<TS>,
    "scm":<Schema Name>,
    "tbl":<Table Name>,
    "t":2
}
```

参数	类型	说明
TS	Number	进行 DDL 变更的事务的 TS
Schema Name	String	DDL 变更的 Schema 的名字，可能为空字符串
Table Name	String	DDL 变更的 Table 的名字，可能为空字符串

- Value:

```
{
    "q":<DDL Query>,
    "t":<DDL Type>
}
```

参数	类型	说明
DDL Query	String	DDL Query SQL
DDL Type	String	DDL 类型，详见：DDL 的类型码

Resolved Event

- Key:

```
{
    "ts":<TS>,
    "t":3
}
```

参数	类型	说明
TS	Number	Resolved TS，任意小于该 TS 的 Event 已经发送完毕

- Value: None

Event 流的输出示例

本部分展示并描述 Event 流的输出日志。

假设在上游执行以下 SQL 语句，MQ Partition 数量为 2：

```
CREATE TABLE test.t1(id int primary key, val varchar(16));
```

如以下执行日志中的 Log 1、Log 3 所示，DDL Event 将被广播到所有 MQ Partition，Resolved Event 会被周期性地广播到各个 MQ Partition：

```
1. [partition=0] [key="{\"ts\":415508856908021766,\"scm\":\"test\",\"tbl\":\"t1\",\"t\":2}"] [
    ↪ value="{\"q\":\"CREATE TABLE test.t1(id int primary key, val varchar(16))\",\"t\":3}"]
2. [partition=0] [key="{\"ts\":415508856908021766,\"t\":3}"] [value=]
3. [partition=1] [key="{\"ts\":415508856908021766,\"scm\":\"test\",\"tbl\":\"t1\",\"t\":2}"] [
    ↪ value="{\"q\":\"CREATE TABLE test.t1(id int primary key, val varchar(16))\",\"t\":3}"]
4. [partition=1] [key="{\"ts\":415508856908021766,\"t\":3}"] [value=]
```

在上游执行以下 SQL 语句：

```
BEGIN;
INSERT INTO test.t1(id, val) VALUES (1, 'aa');
INSERT INTO test.t1(id, val) VALUES (2, 'aa');
UPDATE test.t1 SET val = 'bb' WHERE id = 2;
INSERT INTO test.t1(id, val) VALUES (3, 'cc');
COMMIT;
```

- 如以下执行日志中的 Log 5 和 Log 6 所示，同一张表内的 Row Changed Event 可能会根据主键被分派到不同的 Partition，但同一行的变更一定会分派到同一个 Partition，方便下游并发处理。
- 如 Log 6 所示，在一个事务内对同一行进行多次修改，只会发出一个 Row Changed Event。
- Log 8 是 Log 7 的重复 Event。 Row Changed Event 可能重复，但每个版本的 Event 第一次发出的次序一定是有序的。

```
5. [partition=0] [key="{\"ts\":415508878783938562,\"scm\":\"test\",\"tbl\":\"t1\",\"t\":1}"] [
    ↪ value="{\"u\":{\"id\":{\"t\":3,\"h\":true,\"v\":1},\"val\":{\"t\":15,\"v\":\"YWE=\"}}}"]
6. [partition=1] [key="{\"ts\":415508878783938562,\"scm\":\"test\",\"tbl\":\"t1\",\"t\":1}"] [
    ↪ value="{\"u\":{\"id\":{\"t\":3,\"h\":true,\"v\":2},\"val\":{\"t\":15,\"v\":\"YmI=\"}}}"]
7. [partition=0] [key="{\"ts\":415508878783938562,\"scm\":\"test\",\"tbl\":\"t1\",\"t\":1}"] [
    ↪ value="{\"u\":{\"id\":{\"t\":3,\"h\":true,\"v\":3},\"val\":{\"t\":15,\"v\":\"Y2M=\"}}}"]
8. [partition=0] [key="{\"ts\":415508878783938562,\"scm\":\"test\",\"tbl\":\"t1\",\"t\":1}"] [
    ↪ value="{\"u\":{\"id\":{\"t\":3,\"h\":true,\"v\":3},\"val\":{\"t\":15,\"v\":\"Y2M=\"}}}"]
```

在上游执行以下 SQL 语句：

```
BEGIN;
DELETE FROM test.t1 WHERE id = 1;
UPDATE test.t1 SET val = 'dd' WHERE id = 3;
UPDATE test.t1 SET id = 4, val = 'ee' WHERE id = 2;
COMMIT;
```

- Log 9 是 Delete 类型的 Row Changed Event，这种类型的 Event 只包含主键列或唯一索引列。
- Log 13 和 Log 14 是 Resolved Event。Resolved Event 意味着在这个 Partition 中，任意小于 Resolved TS 的 Event（包括 Row Changed Event 和 DDL Event）已经发送完毕。

```
9. [partition=0] [key="{\"ts\":415508881418485761,\"scm\":\"test\",\"tbl\":\"t1\",\"t\":1}"] [
    ↪ value="{\"d\":{\"id\":{\"t\":3,\"h\":true,\"v\":1}}}"]
10. [partition=1] [key="{\"ts\":415508881418485761,\"scm\":\"test\",\"tbl\":\"t1\",\"t\":1}"] [
    ↪ value="{\"d\":{\"id\":{\"t\":3,\"h\":true,\"v\":2}}}"]
11. [partition=0] [key="{\"ts\":415508881418485761,\"scm\":\"test\",\"tbl\":\"t1\",\"t\":1}"] [
    ↪ value="{\"u\":{\"id\":{\"t\":3,\"h\":true,\"v\":3},\"val\":{\"t\":15,\"v\":\"ZGQ=\"}}}"]
12. [partition=0] [key="{\"ts\":415508881418485761,\"scm\":\"test\",\"tbl\":\"t1\",\"t\":1}"] [
    ↪ value="{\"u\":{\"id\":{\"t\":3,\"h\":true,\"v\":4},\"val\":{\"t\":15,\"v\":\"ZWU=\"}}}"]
13. [partition=0] [key="{\"ts\":415508881038376963,\"t\":3}"] [value=]
14. [partition=1] [key="{\"ts\":415508881038376963,\"t\":3}"] [value=]
```

消费端协议解析

目前 TiCDC 没有提供 Open Protocol 协议解析的标准实现，但是提供了 Golang 版本和 Java 版本的解析例子。你可以参考本文档提供的数据格式和以下例子实现消费端协议解析。

- Golang 例子
- Java 例子

Column 的类型码

Column 的类型码用于标识 Row Changed Event 中列的数据类型。

类型	Code	输出示例	说明
TINYINT/BOOL	1	{ “t” :1, “v” :1}	
SMALLINT	2	{ “t” :2, “v” :1}	
INT	3	{ “t” :3, “v” :123}	
FLOAT	4	{ “t” :4, “v” :153.123}	
DOUBLE	5	{ “t” :5, “v” :153.123}	
NULL	6	{ “t” :6, “v” :null}	
TIMESTAMP	7	{ “t” :7, “v” : “1973-12-30 15:30:00” }	

类型	Code	输出示例	说明
BIGINT	8	{“t”:8,“v”:123}	
MEDIUMINT	9	{“t”:9,“v”:123}	
DATE	10/14	{“t”:10,“v”:“2000-01-01”}	
TIME	11	{“t”:11,“v”:“23:59:59”}	
DATETIME	12	{“t”:12,“v”:“2015-12-20 23:58:58”}	
YEAR	13	{“t”:13,“v”:1970}	
VARCHAR/VARBINARY	15/253	{“t”:15,“v”:“val测试”} / {“t”:15,“v”:“\\x89PNG\\r\\n\\x1a\\n”}	val 编码为 UTF-8；当上游类型为 VARBINARY 时，将对不可见的字符转义
BIT	16	{“t”:16,“v”:81}	
JSON	245	{“t”:245,“v”:“{\” key1\ “:\” value1\ “}”}	
DECIMAL	246	{“t”:246,“v”:“129012.1230000”}	
ENUM	247	{“t”:247,“v”:1}	
SET	248	{“t”:248,“v”:3}	

类型	Code	输出示例	说明
TINYTEXT/TINYBLOB	249	{“t”:249,“v”:“5rWL6K+VdGV4dA==”}	编码为Base64
MEDIUMTEXT/MEDIUMBLOB	250	{“t”:250,“v”:“5rWL6K+VdGV4dA==”}	编码为Base64
LONGTEXT/LONGBLOB	251	{“t”:251,“v”:“5rWL6K+VdGV4dA==”}	编码为Base64
TEXT/BLOB	252	{“t”:252,“v”:“5rWL6K+VdGV4dA==”}	编码为Base64

类型	Code	输出示例	说明
CHAR/BINARY	254	{ “t” :254, “v”:“valu 试” } / { “t” :254, “v”:“\\x89PNG\\r\\n\\x1a\\n” }	测编码为 UTF-8；当上游类型为 BINARY 时，将对不可见的字符转义
GEOMETRY	255		尚不支持

DDL 的类型码

DDL 的类型码用于标识 DDL Event 中的 DDL 语句的类型。

类型	Code
Create Schema	1
Drop Schema	2
Create Table	3
Drop Table	4
Add Column	5
Drop Column	6
Add Index	7
Drop Index	8

类型	Code
Add Foreign Key	9
Drop Foreign Key	10
Truncate Table	11
Modify Column	12
Rebase Auto ID	13
Rename Table	14
Set Default Value	15
Shard RowID	16
Modify Table Comment	17
Rename Index	18
Add Table Partition	19
Drop Table Partition	20
Create View	21
Modify Table Charset And Collate	22
Truncate Table Partition	23
Drop View	24
Recover Table	25
Modify Schema Charset And Collate	26
Lock Table	27
Unlock Table	28
Repair Table	29
Set TiFlash Replica	30
Update TiFlash Replica Status	31
Add Primary Key	32
Drop Primary Key	33
Create Sequence	34
Alter Sequence	35
Drop Sequence	36

列标志位

列标志位以 Bit flags 形式标记列的相关属性。

位移	值	名称	说明
1	0x01	BinaryFlag	该列是否为二进制编码列
2	0x02	HandleKeyFlag	该列是否为 Handle 列
3	0x04	GeneratedColumnFlag	该列是否为生成列
4	0x08	PrimaryKeyFlag	该列是否为主键列
5	0x10	UniqueKeyFlag	该列是否为唯一索引列
6	0x20	MultipleKeyFlag	该列是否为组合索引列
7	0x40	NullableFlag	该列是否为可空列
8	0x80	UnsignedFlag	该列是否为无符号列

示例：

若某列 Flag 值为 85，则代表这一列为可空列、唯一索引列、生成列、二进制编码列。

```
85 == 0b_101_0101
   == NullableFlag | UniqueKeyFlag | GeneratedColumnFlag | BinaryFlag
```

若某列 Flag 值为 46，则代表这一列为组合索引列、主键列、生成列、Handle 列。

```
46 == 0b_010_1110
   == MultipleKeyFlag | PrimaryKeyFlag | GeneratedColumnFlag | HandleKeyFlag
```

> 注意：
>
> - BinaryFlag 仅在列为 BLOB/TEXT（包括 TINYBLOB/TINYTEXT、BINARY/CHAR 等）类型时才有意义。当上游列为 BLOB 类型时，BinaryFlag 置 1；当上游列为 TEXT 类型时，BinaryFlag 置 0。
> - 若要同步上游的一张表，TiCDC 会选择一个有效索引作为 Handle Index。Handle Index 包含的列的 HandleKeyFlag 置 1。

13.9.4.7.4 TiCDC CSV Protocol

当使用云存储服务作为下游 sink 时，你可以使用 CSV 格式将 DML 事件发送到下游云存储服务。

使用 CSV

使用 CSV 时的配置样例如下所示：

```
cdc cli changefeed create --server=http://127.0.0.1:8300 --changefeed-id="csv-test" --sink-uri="
    ↪ s3://bucket/prefix" --config changefeed.toml
```

changefeed.toml 文件内容如下：

```
[sink]
protocol = "csv"
terminator = "\n"

[sink.csv]
delimiter = ','
quote = '"'
null = '\N'
include-commit-ts = true
```

数据保存的事务性约束

- 单个 CSV 文件中后一行数据的 commit-ts 大于等于前一行数据的 commit-ts。
- 单表的同一事务不会存储在不同的 CSV 文件中。

- 相同事务涉及的不同表会存储在不同的 CSV 文件中。

数据存储路径结构

关于数据存储路径结构的更多信息，请参考同步数据到存储服务。

数据格式定义

CSV 文件中，单行的每一列定义如下：

- 第一列：DML 操作指示符，取值包括 I、U 和 D。I 表示 INSERT，U 表示 UPDATE，D 表示 DELETE。
- 第二列：表名。
- 第三列：库名。
- 第四列：commit ts，即原始事务的 commit ts。该列为可选配置。
- 第五列至最后一列：变更数据的列，可为一列或多列。

假设某张表 hr.employee 的定义如下：

```
CREATE TABLE `employee` (
  `Id` int NOT NULL,
  `LastName` varchar(20) DEFAULT NULL,
  `FirstName` varchar(30) DEFAULT NULL,
  `HireDate` date DEFAULT NULL,
  `OfficeLocation` varchar(20) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

该表上的 DML 事件以 CSV 格式存储后如下所示：

```
"I","employee","hr",433305438660591626,101,"Smith","Bob","2014-06-04","New York"
"U","employee","hr",433305438660591627,101,"Smith","Bob","2015-10-08","Los Angeles"
"D","employee","hr",433305438660591629,101,"Smith","Bob","2017-03-13","Dallas"
"I","employee","hr",433305438660591630,102,"Alex","Alice","2017-03-14","Shanghai"
"U","employee","hr",433305438660591630,102,"Alex","Alice","2018-06-15","Beijing"
```

数据类型映射

MySQL 类型	CSV 类型	示例	描述
BOOLEAN/TINYINT/SMALLINT/INT/MEDIUMINT/BIGINT	Integer	123	-
FLOAT/DOUBLE	Float	153.123	-
NULL	Null	\N	-
TIMESTAMP/DATETIME	String	"1973-12-30 15:30:00.123456"	格式：yyyy-MM-dd HH:m
DATE	String	"2000-01-01"	格式：yyyy-MM-dd
TIME	String	"23:59:59"	格式：HH:mm:ss
YEAR	Integer	1970	-
VARCHAR/JSON/TINYTEXT/MEDIUMTEXT/LONGTEXT/TEXT/CHAR	String	"test"	以 UTF-8 编码输出
VARBINARY/TINYBLOB/MEDIUMBLOB/LONGBLOB/BLOB/BINARY	String	"6Zi/5pav"	以 Base64 编码输出
BIT	Integer	81	-
DECIMAL	String	"129012.1230000"	-

MySQL 类型	CSV 类型	示例	描述
ENUM	String	"a"	-
SET	String	"a,b"	-

13.9.4.8 TiCDC OpenAPI v2

TiCDC 提供 OpenAPI 功能，你可以通过 OpenAPI v2 对 TiCDC 集群进行查询和运维操作。OpenAPI 的功能是cdc cli工具的一个子集。

> **注意：**
>
> TiCDC OpenAPI v1 将在未来版本中被删除。推荐使用 TiCDC OpenAPI v2。

你可以通过 OpenAPI 完成 TiCDC 集群的如下运维操作：

- 获取 TiCDC 节点状态信息
- 检查 TiCDC 集群的健康状态
- 创建同步任务
- 删除同步任务
- 更新同步任务配置
- 查询同步任务列表
- 查询特定同步任务
- 暂停同步任务
- 恢复同步任务
- 查询同步子任务列表
- 查询特定同步子任务
- 查询 TiCDC 服务进程列表
- 驱逐 owner 节点
- 动态调整 TiCDC Server 日志级别

所有 API 的请求体与返回值统一使用 JSON 格式数据。请求如果成功，则统一返回 200 OK。本文档以下部分描述当前提供的 API 的具体使用方法。

在下文的示例描述中，假设 TiCDC server 的监听 IP 地址为 127.0.0.1，端口为 8300。在启动 TiCDC server 时，可以通过 --addr=ip:port 指定 TiCDC 绑定的 IP 地址和端口。

13.9.4.8.1 API 统一错误格式

对 API 发起请求后，如发生错误，返回错误信息的格式如下所示：

```
{
  "error_msg": "",
  "error_code": ""
}
```

如上所示，error_msg 描述错误信息，error_code 则是对应的错误码。

13.9.4.8.2 API List 接口统一返回格式

一个 API 请求如果返回是一个资源列表（例如，返回所有的服务进程 Captures），TiCDC 统一的返回格式如下：

```
{
  "total": 2,
  "items": [
    {
      "id": "d2912e63-3349-447c-90ba-wwww",
      "is_owner": true,
      "address": "127.0.0.1:8300"
    },
    {
      "id": "d2912e63-3349-447c-90ba-xxxx",
      "is_owner": false,
      "address": "127.0.0.1:8302"
    }
  ]
}
```

如上所示:

- total: 表示有一共有多少个资源。
- items: 当次请求返回的资源在这个数组里，数组所有的元素都是同种资源。

13.9.4.8.3 获取 TiCDC 节点状态信息

该接口是一个同步接口，请求成功会返回对应节点的状态信息。

请求 URI

```
GET /api/v2/status
```

使用样例

以下请求会获取 IP 地址为 127.0.0.1，端口号为 8300 的 TiCDC 节点的状态信息。

```
curl -X GET http://127.0.0.1:8300/api/v2/status
```

```
{
  "version": "v7.0.0-master-dirty",
  "git_hash": "10413bded1bdb2850aa6d7b94eb375102e9c44dc",
  "id": "d2912e63-3349-447c-90ba-72a4e04b5e9e",
  "pid": 1447,
  "is_owner": true,
  "liveness": 0
}
```

以上返回信息的字段解释如下：

- `version`：当前 TiCDC 版本号。
- `git_hash`：Git 哈希值。
- `id`：该节点的 capture ID。
- `pid`：该节点 capture 进程的 PID。
- `is_owner`：表示该节点是否是 owner。
- `liveness`: 该节点是否在线。0 表示正常，1 表示处于 `graceful shutdown` 状态。

13.9.4.8.4 检查 TiCDC 集群的健康状态

该接口是一个同步接口，在集群健康的时候会返回 `200 OK`。

请求 URI

```
GET /api/v2/health
```

使用样例

```
curl -X GET http://127.0.0.1:8300/api/v2/health
```

如果集群健康，则返回 `200 OK` 和一个空的 json `{}`：

```
{}
```

如果集群不健康，则返回错误信息。

13.9.4.8.5 创建同步任务

该接口用于向 TiCDC 提交一个同步任务，请求成功会返回 `200 OK`。该返回结果表示服务器收到了执行命令指示，并不代表命令被成功执行。

请求 URI

```
POST /api/v2/changefeeds
```

参数说明

```
{
  "changefeed_id": "string",
  "replica_config": {
    "bdr_mode": true,
    "case_sensitive": true,
    "check_gc_safe_point": true,
    "consistent": {
      "flush_interval": 0,
      "level": "string",
      "max_log_size": 0,
      "storage": "string"
    },
    "enable_old_value": true,
```

```
"enable_sync_point": true,
"filter": {
  "do_dbs": [
    "string"
  ],
  "do_tables": [
    {
      "database_name": "string",
      "table_name": "string"
    }
  ],
  "event_filters": [
    {
      "ignore_delete_value_expr": "string",
      "ignore_event": [
        "string"
      ],
      "ignore_insert_value_expr": "string",
      "ignore_sql": [
        "string"
      ],
      "ignore_update_new_value_expr": "string",
      "ignore_update_old_value_expr": "string",
      "matcher": [
        "string"
      ]
    }
  ],
  "ignore_dbs": [
    "string"
  ],
  "ignore_tables": [
    {
      "database_name": "string",
      "table_name": "string"
    }
  ],
  "ignore_txn_start_ts": [
    0
  ],
  "rules": [
    "string"
  ]
},
"force_replicate": true,
```

```
  "ignore_ineligible_table": true,
  "memory_quota": 0,
  "mounter": {
    "worker_num": 0
  },
  "sink": {
    "column_selectors": [
      {
        "columns": [
          "string"
        ],
        "matcher": [
          "string"
        ]
      }
    ],
    "csv": {
      "delimiter": "string",
      "include_commit_ts": true,
      "null": "string",
      "quote": "string"
    },
    "date_separator": "string",
    "dispatchers": [
      {
        "matcher": [
          "string"
        ],
        "partition": "string",
        "topic": "string"
      }
    ],
    "enable_partition_separator": true,
    "encoder_concurrency": 0,
    "protocol": "string",
    "schema_registry": "string",
    "terminator": "string",
    "transaction_atomicity": "string"
  },
  "sync_point_interval": "string",
  "sync_point_retention": "string"
},
"sink_uri": "string",
"start_ts": 0,
"target_ts": 0
```

```
}
```

参数说明如下：

参数名	说明
changefeed_id	STRING 类型，同步任务的 ID。（非必选）
replica_config	同步任务的配置参数。（非必选）
sink_uri	STRING 类型，同步任务下游的地址。（必选）
start_ts	UINT64 类型，指定 changefeed 的开始 TSO。TiCDC 集群将从这个 TSO 开始拉取数据。默认为当前时间。（非必选）
target_ts	UINT64 类型，指定 changefeed 的目标 TSO。达到这个 TSO 后，TiCDC 集群将停止拉取数据。默认为空，即 TiCDC 不会自动停止。（非必选）

changefeed_id、start_ts、target_ts、sink_uri 的含义和格式与使用 cli 创建同步任务中所作的解释相同，具体解释请参见该文档。需要注意，当在 sink_uri 中指定证书的路径时，须确保已将对应证书上传到对应的 TiCDC server 上。

replica_config 参数说明如下：

参数名	说明
bdr_mode	BOOLEAN 类型，是否开启双向同步复制。默认值为 false。（非必选）
case_sensitive	BOOLEAN 类型，过滤表名时大小写是否敏感，默认值为 true。（非必选）
check_gc_safe_point	BOOLEAN 类型，是否检查同步任务的开始时间早于 GC 时间，默认值为 true。（非必选）
consistent	Redo log 配置。（非必选）
enable_old_value	BOOLEAN 类型，是否输出 old value 值（即变更前的值），默认值为 true。（非必选）
enable_sync_point	BOOLEAN 类型，是否开启 sync point 功能。（非必选）
filter	filter 配置。（非必选）
force_replicate	BOOLEAN 类型，该值默认为 false，当指定为 true 时，同步任务会尝试强制同步没有唯一索引的表。（非必选）
ignore_ineligible_table ↪	BOOLEAN 类型，该值默认为 false，当指定为 true 时，同步任务会忽略无法进行同步的表。（非必选）
memory_quota	UINT64 类型，同步任务的内存 quota。（非必选）
mounter	同步任务 mounter 配置。（非必选）
sink	同步任务的sink配置。（非必选）
sync_point_interval	STRING 类型，注意返回值为 UINT64 类型的纳秒级时间，sync point 功能开启时，对齐上下游 snapshot 的时间间隔。默认值为 10m，最小值为 30s。（非必选）
sync_point_retention ↪	STRING 类型，注意返回值为 UINT64 类型的纳秒级时间，sync point 功能开启时，在下游表中保存的数据的时长，超过这个时间的数据会被清理。默认值为 24h。（非必选）

consistent 参数说明如下：

参数名	说明
flush_interval	UINT64 类型，redo log 文件 flush 间隔。（非必选）
level	STRING 类型，同步数据的一致性级别。（非必选）
max_log_size	UINT64 类型，redo log 的最大值。（非必选）
storage	STRING 类型，存储的目标地址。（非必选）

filter 参数说明如下：

参数名	说明
do_dbs	STRING ARRAY 类型，需要同步的数据库。（非必选）
do_tables	需要同步的表。（非必选）
ignore_dbs	STRING ARRAY 类型，不同步的数据库。（非必选）
ignore_tables	不同步的表。（非必选）
event_filters	event 过滤配置。（非必选）
ignore_txn_start_ts	UINT64 ARRAY 类型，指定之后会忽略指定 start_ts 的事务，如 [1, 2]。（非必选）
rules	STRING ARRAY 类型，表库过滤的规则，如 ['foo*.*', 'bar*.*']。详情请参考表库过滤。（非必选）

filter.event_filters 参数说明如下，可参考日志过滤器。

参数名	说明
ignore_delete_value_expr ↪	STRING ARRAY 类型，如 "name = 'john'" 表示过滤掉包含 name = 'john' 条件的 DELETE DML。（非必选）
ignore_event	STRING ARRAY 类型，如 ["insert"] 表示过滤掉 INSERT 事件。（非必选）
ignore_insert_value_expr ↪	STRING ARRAY 类型，如 "id >= 100" 表示过滤掉包含 id >= 100 条件的 INSERT DML。（非必选）
ignore_sql	STRING ARRAY 类型，如 ["^drop", "add column"] 表示过滤掉以 DROP 开头或者包含 ADD COLUMN 的 DDL。（非必选）
ignore_update_new_value_expr ↪	STRING ARRAY 类型，如 "gender = 'male'" 表示过滤掉新值 gender = 'male' 的 UPDATE DML。（非必选）
ignore_update_old_value_expr ↪	STRING ARRAY 类型，如 "age < 18" 表示过滤掉旧值 age < 18 的 UPDATE DML。（非必选）
matcher	STRING ARRAY 类型，是一个白名单，如 ["test.worker"]，表示该过滤规则只应用于 test 库中的 worker 表。（非必选）

mounter 参数说明如下：

参数名	说明
worker_num	INT 类型。Mounter 线程数，Mounter 用于解码 TiKV 输出的数据，默认值为 16。（非必选）

sink 参数说明如下：

参数名	说明
column_selectors	column selector 配置。(非必选)
csv	CSV 配置。(非必选)
date_separator	STRING 类型，文件路径的日期分隔类型。可选类型有 none、 year、 month 和 day。默认值为 none，即不使用日期分隔。(非必选)
dispatchers	事件分发配置数组。(非必选)
encoder_concurrency	INT 类型。MQ sink 中编码器的线程数。默认值为 16。(非必选)
protocol	STRING 类型，对于 MQ 类的 Sink，可以指定消息的协议格式。目前支持以下协议：canal-json、 open-protocol、 canal、 avro 和 maxwell。
schema_registry	STRING 类型，schema registry 地址。(非必选)
terminator	STRING 类型，换行符，用来分隔两个数据变更事件。默认值为空，表示使用 "\r\n" 作为换行符。(非必选)
transaction_atomicity	STRING 类型，事务一致性等级。(非必选)

sink.column_selectors 是一个数组，元素参数说明如下：

参数名	说明
columns	STRING ARRAY 类型，column 数组。
matcher	STRING ARRAY 类型，matcher 配置，匹配语法和过滤器规则的语法相同。

sink.csv 参数说明如下：

参数名	说明
delimiter	STRING 类型，字段之间的分隔符。必须为 ASCII 字符，默认值为 ,。
include_commit_ts	BOOLEAN 类型，是否在 CSV 行中包含 commit-ts。默认值为 false。
null	STRING 类型，如果这一列是 null，那这一列该如何表示。默认是用 \N 来表示。
quote	STRING 类型，用于包裹字段的引号字符。空值代表不使用引号字符。默认值为 "。

sink.dispatchers：对于 MQ 类的 Sink，可以通过该参数配置 event 分发器，支持以下分发器：default、 ts、 rowid、 table 。分发规则如下：

- default：有多个唯一索引 (包括主键) 时按照 table 模式分发；只有一个唯一索引 (或主键) 时按照 rowid 模式分发；如果开启了 old value 特性，按照 table 模式分发。
- ts：以行变更的 commitTs 做 Hash 计算并进行 event 分发。
- rowid：以所选的 HandleKey 列名和列值做 Hash 计算并进行 event 分发。
- table：以表的 schema 名和 table 名做 Hash 计算并进行 event 分发。

sink.dispatchers 是一个数组，元素参数说明如下：

参数名	说明
matcher	STRING ARRAY 类型，匹配语法和过滤器规则的语法相同。
partition	STRING 类型，事件分发的目标 partition。
topic	STRING 类型，事件分发的目标 topic。

使用样例

以下请求会创建一个 ID 为 test5，sink_uri 为 blackhole:// 的同步任务。

```
curl -X POST -H "'Content-type':'application/json'" http://127.0.0.1:8300/api/v2/changefeeds -d
    ↪ '{"changefeed_id":"test5","sink_uri":"blackhole://"}'
```

如果请求成功，则返回 200 OK。如果请求失败，则返回错误信息和错误码。

响应体格式

```
{
  "admin_job_type": 0,
  "checkpoint_time": "string",
  "checkpoint_ts": 0,
  "config": {
    "bdr_mode": true,
    "case_sensitive": true,
    "check_gc_safe_point": true,
    "consistent": {
      "flush_interval": 0,
      "level": "string",
      "max_log_size": 0,
      "storage": "string"
    },
    "enable_old_value": true,
    "enable_sync_point": true,
    "filter": {
      "do_dbs": [
        "string"
      ],
      "do_tables": [
        {
          "database_name": "string",
          "table_name": "string"
        }
      ],
      "event_filters": [
        {
          "ignore_delete_value_expr": "string",
          "ignore_event": [
```

```
          "string"
        ],
        "ignore_insert_value_expr": "string",
        "ignore_sql": [
          "string"
        ],
        "ignore_update_new_value_expr": "string",
        "ignore_update_old_value_expr": "string",
        "matcher": [
          "string"
        ]
      }
    ],
    "ignore_dbs": [
      "string"
    ],
    "ignore_tables": [
      {
        "database_name": "string",
        "table_name": "string"
      }
    ],
    "ignore_txn_start_ts": [
      0
    ],
    "rules": [
      "string"
    ]
  },
  "force_replicate": true,
  "ignore_ineligible_table": true,
  "memory_quota": 0,
  "mounter": {
    "worker_num": 0
  },
  "sink": {
    "column_selectors": [
      {
        "columns": [
          "string"
        ],
        "matcher": [
          "string"
        ]
      }
```

```
    ],
    "csv": {
      "delimiter": "string",
      "include_commit_ts": true,
      "null": "string",
      "quote": "string"
    },
    "date_separator": "string",
    "dispatchers": [
      {
        "matcher": [
          "string"
        ],
        "partition": "string",
        "topic": "string"
      }
    ],
    "enable_partition_separator": true,
    "encoder_concurrency": 0,
    "protocol": "string",
    "schema_registry": "string",
    "terminator": "string",
    "transaction_atomicity": "string"
  },
  "sync_point_interval": "string",
  "sync_point_retention": "string"
},
"create_time": "string",
"creator_version": "string",
"error": {
  "addr": "string",
  "code": "string",
  "message": "string"
},
"id": "string",
"resolved_ts": 0,
"sink_uri": "string",
"start_ts": 0,
"state": "string",
"target_ts": 0,
"task_status": [
  {
    "capture_id": "string",
    "table_ids": [
      0
```

```
      ]
    }
  ]
}
```

参数说明如下：

参数名	说明
admin_job_type	INTEGER 类型，admin 事件类型
checkpoint_time	STRING 类型，同步任务当前 checkpoint 的格式化时间表示
checkpoint_ts	STRING 类型，同步任务当前 checkpoint 的 TSO 表示
config	同步任务配置，结构和含义与创建同步任务中的 replica_config 配置项相同
create_time	STRING 类型，同步任务创建的时间
creator_version	STRING 类型，同步任务创建时 TiCDC 的版本
error	同步任务错误
id	STRING 类型，同步任务 ID
resolved_ts	UINT64 类型，同步任务 resolved ts
sink_uri	STRING 类型，同步任务的 sink uri
start_ts	UINT64 类型，同步任务 start ts
state	STRING 类型，同步任务状态，状态可分为 normal、stopped、error、failed、finished
target_ts	UINT64 类型，同步任务的 target ts
task_status	同步任务分发的详细状态

task_status 参数说明如下：

参数名	说明
capture_id	STRING 类型，Capture ID
table_ids	UINT64 ARRAY 类型，该 Capture 上正在同步的 table 的 ID

error 参数说明如下：

参数名	说明
addr	STRING 类型，Capture 地址
code	STRING 类型，错误码
message	STRING 类型，错误的详细信息

13.9.4.8.6 删除同步任务

该接口是幂等的（即其任意多次执行所产生的影响均与一次执行的影响相同），用于删除一个 changefeed 同步任务，请求成功会返回 200 OK。该返回结果表示服务器收到了执行命令指示，并不代表命令被成功执行。

请求 URI

```
DELETE /api/v2/changefeeds/{changefeed_id}
```

参数说明

路径参数

参数名	说明
changefeed_id	需要删除的同步任务 (changefeed) 的 ID

使用样例

以下请求会删除 ID 为 test1 的同步任务。

```
curl -X DELETE http://127.0.0.1:8300/api/v2/changefeeds/test1
```

如果请求成功，则返回 200 OK。如果请求失败，则返回错误信息和错误码。

13.9.4.8.7 更新同步任务配置

该接口用于更新一个同步任务，请求成功会返回 200 OK。该返回结果表示服务器收到了执行命令指示，并不代表命令被成功执行。

修改 changefeed 配置需要按照暂停任务 -> 修改配置 -> 恢复任务的流程。

请求 URI

PUT /api/v2/changefeeds/{changefeed_id}

参数说明

路径参数

参数名	说明
changefeed_id	需要更新的同步任务 (changefeed) 的 ID

请求体参数

```
{
  "replica_config": {
    "bdr_mode": true,
    "case_sensitive": true,
    "check_gc_safe_point": true,
    "consistent": {
      "flush_interval": 0,
      "level": "string",
      "max_log_size": 0,
      "storage": "string"
    },
    "enable_old_value": true,
    "enable_sync_point": true,
    "filter": {
```

```
    "do_dbs": [
      "string"
    ],
    "do_tables": [
      {
        "database_name": "string",
        "table_name": "string"
      }
    ],
    "event_filters": [
      {
        "ignore_delete_value_expr": "string",
        "ignore_event": [
          "string"
        ],
        "ignore_insert_value_expr": "string",
        "ignore_sql": [
          "string"
        ],
        "ignore_update_new_value_expr": "string",
        "ignore_update_old_value_expr": "string",
        "matcher": [
          "string"
        ]
      }
    ],
    "ignore_dbs": [
      "string"
    ],
    "ignore_tables": [
      {
        "database_name": "string",
        "table_name": "string"
      }
    ],
    "ignore_txn_start_ts": [
      0
    ],
    "rules": [
      "string"
    ]
  },
  "force_replicate": true,
  "ignore_ineligible_table": true,
  "memory_quota": 0,
```

```
    "mounter": {
      "worker_num": 0
    },
    "sink": {
      "column_selectors": [
        {
          "columns": [
            "string"
          ],
          "matcher": [
            "string"
          ]
        }
      ],
      "csv": {
        "delimiter": "string",
        "include_commit_ts": true,
        "null": "string",
        "quote": "string"
      },
      "date_separator": "string",
      "dispatchers": [
        {
          "matcher": [
            "string"
          ],
          "partition": "string",
          "topic": "string"
        }
      ],
      "enable_partition_separator": true,
      "encoder_concurrency": 0,
      "protocol": "string",
      "schema_registry": "string",
      "terminator": "string",
      "transaction_atomicity": "string"
    },
    "sync_point_interval": "string",
    "sync_point_retention": "string"
  },
  "sink_uri": "string",
  "target_ts": 0
}
```

目前仅支持通过 API 修改同步任务的如下配置。

参数名	说明
target_ts	UINT64 类型，指定 changefeed 的目标 TSO。(非必选)
sink_uri	STRING 类型，同步任务下游的地址。(非必选)
replica_config	sink 的配置参数, 必须是完整的配置。(非必选)

以上参数含义与创建同步任务中的参数相同，此处不再赘述。

使用样例

以下请求会更新 ID 为 test1 的同步任务的 target_ts 为 32。

```
curl -X PUT -H "'Content-type':'application/json'" http://127.0.0.1:8300/api/v2/changefeeds/
    ↪ test1 -d '{"target_ts":32}'
```

若是请求成功，则返回 200 OK，若请求失败，则返回错误信息和错误码。响应的 JSON 格式以及字段含义与创建同步任务中的响应参数相同，此处不再赘述。

13.9.4.8.8 查询同步任务列表

该接口是一个同步接口，请求成功会返回 TiCDC 集群中所有同步任务 (changefeed) 的基本信息。

请求 URI

```
GET /api/v2/changefeeds
```

参数说明

查询参数

参数名	说明
state	非必选，指定后将会只返回该状态的同步任务的信息

state 可选值为 all、 normal、 stopped、 error、 failed、 finished。

若不指定该参数，则默认返回处于 normal、 stopped、 failed 状态的同步任务基本信息。

使用样例

以下请求查询所有状态 (state) 为 normal 的同步任务的基本信息。

```
curl -X GET http://127.0.0.1:8300/api/v2/changefeeds?state=normal
```

```
{
  "total": 2,
  "items": [
    {
      "id": "test",
      "state": "normal",
      "checkpoint_tso": 439749918821711874,
      "checkpoint_time": "2023-02-27 23:46:52.888",
```

```
      "error": null
    },
    {
      "id": "test2",
      "state": "normal",
      "checkpoint_tso": 439749918821711874,
      "checkpoint_time": "2023-02-27 23:46:52.888",
      "error": null
    }
  ]
}
```

以上返回的信息的说明如下：

- `id`：同步任务的 ID。
- `state`：同步任务当前所处的状态。
- `checkpoint_tso`：同步任务当前 checkpoint 的 TSO 表示。
- `checkpoint_time`：同步任务当前 checkpoint 的格式化时间表示。
- `error`：同步任务的错误信息。

13.9.4.8.9 查询特定同步任务

该接口是一个同步接口，请求成功会返回指定同步任务 (changefeed) 的详细信息。

请求 URI

```
GET /api/v2/changefeeds/{changefeed_id}
```

参数说明

路径参数

参数名	说明
`changefeed_id`	需要查询的同步任务 (changefeed) 的 ID

使用样例

以下请求会查询 ID 为 `test1` 的同步任务的详细信息。

```
curl -X GET http://127.0.0.1:8300/api/v2/changefeeds/test1
```

响应的 JSON 格式以及字段含义与创建同步任务中的响应参数相同，此处不再赘述。

13.9.4.8.10 暂停同步任务

该接口暂停一个同步任务，请求成功会返回 `200 OK`。该返回结果表示服务器收到了执行命令指示，并不代表命令被成功执行。

请求 URI

`POST /api/v2/changefeeds/{changefeed_id}/pause`

参数说明

路径参数

参数名	说明
`changefeed_id`	需要暂停的同步任务 (changefeed) 的 ID

使用样例

以下请求会暂停 ID 为 `test1` 的同步任务。

```
curl -X POST http://127.0.0.1:8300/api/v2/changefeeds/test1/pause
```

如果请求成功，则返回 `200 OK`。如果请求失败，则返回错误信息和错误码。

13.9.4.8.11 恢复同步任务

该接口恢复一个同步任务，请求成功会返回 `200 OK`。该返回结果表示服务器收到了执行命令指示，并不代表命令被成功执行。

请求 URI

`POST /api/v2/changefeeds/{changefeed_id}/resume`

参数说明

路径参数

参数名	说明
`changefeed_id`	需要恢复的同步任务 (changefeed) 的 ID

请求体参数

```
{
  "overwrite_checkpoint_ts": 0
}
```

参数名	说明
`overwrite_checkpoint_ts`	UINT64 类型，恢复同步任务 (changefeed) 时重新指定的 checkpoint TSO

使用样例

以下请求会恢复 ID 为 `test1` 的同步任务。

```
curl -X POST http://127.0.0.1:8300/api/v2/changefeeds/test1/resume -d '{}'
```

如果请求成功，则返回 200 OK。如果请求失败，则返回错误信息和错误码。

13.9.4.8.12 查询同步子任务列表

该接口是一个同步接口，请求成功会返回当前 TiCDC 集群中的所有同步子任务 (processor) 的基本信息。

请求 URI

```
GET /api/v2/processors
```

使用样例

```
curl -X GET http://127.0.0.1:8300/api/v2/processors
```

```
{
  "total": 3,
  "items": [
    {
      "changefeed_id": "test2",
      "capture_id": "d2912e63-3349-447c-90ba-72a4e04b5e9e"
    },
    {
      "changefeed_id": "test1",
      "capture_id": "d2912e63-3349-447c-90ba-72a4e04b5e9e"
    },
    {
      "changefeed_id": "test",
      "capture_id": "d2912e63-3349-447c-90ba-72a4e04b5e9e"
    }
  ]
}
```

以上返回的信息的说明如下：

- changefeed_id：同步任务的 ID。
- capture_id：Capture 的 ID。

13.9.4.8.13 查询特定同步子任务

该接口是一个同步接口，请求成功会返回指定同步子任务 (processor) 的详细信息。

请求 URI

```
GET /api/v2/processors/{changefeed_id}/{capture_id}
```

参数说明

路径参数

参数名	说明
changefeed_id	需要查询的子任务的 Changefeed ID
capture_id	需要查询的子任务的 Capture ID

使用样例

以下请求查询 changefeed_id 为 test、capture_id 为 561c3784-77f0-4863-ad52-65a3436db6af 的同步子任务。一个同步子任务通过 changefeed_id 和 capture_id 来标识。

```
curl -X GET http://127.0.0.1:8300/api/v2/processors/test/561c3784-77f0-4863-ad52-65a3436db6af
```

```
{
  "table_ids": [
    80
  ]
}
```

以上返回的信息的说明如下：

- table_ids：在这个 capture 上同步的 table 的 ID。

13.9.4.8.14 查询 TiCDC 服务进程列表

该接口是一个同步接口，请求成功会返回当前 TiCDC 集群中的所有服务进程 (capture) 的基本信息。

请求 URI

```
GET /api/v2/captures
```

使用样例

```
curl -X GET http://127.0.0.1:8300/api/v2/captures
```

```
{
  "total": 1,
  "items": [
    {
      "id": "d2912e63-3349-447c-90ba-72a4e04b5e9e",
      "is_owner": true,
      "address": "127.0.0.1:8300"
    }
  ]
}
```

以上返回的信息的说明如下：

- id：capture 的 ID。
- is_owner：该 capture 是否是 owner。
- address：该 capture 的地址。

13.9.4.8.15 驱逐 owner 节点

该接口是一个异步的请求，请求成功会返回 200 OK。该返回结果表示服务器收到了执行命令指示，并不代表命令被成功执行。

请求 URI

```
POST /api/v2/owner/resign
```

使用样例

以下请求会驱逐 TiCDC 当前的 owner 节点，并会触发新一轮的选举，产生新的 owner 节点。

```
curl -X POST http://127.0.0.1:8300/api/v2/owner/resign
```

如果请求成功，则返回 200 OK。如果请求失败，则返回错误信息和错误码。

13.9.4.8.16 动态调整 TiCDC Server 日志级别

该接口是一个同步接口，请求成功会返回 200 OK。

请求 URI

```
POST /api/v2/log
```

请求参数

请求体参数

参数名	说明
log_level	想要设置的日志等级

log_level 支持 zap 提供的日志级别：“debug”、“info”、“warn”、“error”、“dpanic”、“panic”、“fatal”。

使用样例

```
curl -X POST -H "'Content-type':'application/json'" http://127.0.0.1:8300/api/v2/log -d '{"
    ↪ log_level":"debug"}'
```

如果请求成功，则返回 200 OK。如果请求失败，则返回错误信息和错误码。

13.9.4.9 TiCDC OpenAPI v1

> **注意：**
>
> TiCDC OpenAPI v1 将在未来版本中被删除。推荐使用TiCDC OpenAPI v2。

TiCDC 提供 OpenAPI 功能，你可以通过 OpenAPI 对 TiCDC 集群进行查询和运维操作。OpenAPI 的总体功能和cdc cli工具类似。

你可以通过 OpenAPI 完成 TiCDC 集群的如下运维操作：

- 获取 TiCDC 节点状态信息
- 检查 TiCDC 集群的健康状态
- 创建同步任务
- 删除同步任务
- 更新同步任务配置
- 查询同步任务列表
- 查询特定同步任务
- 暂停同步任务
- 恢复同步任务
- 查询同步子任务列表
- 查询特定同步子任务
- 查询 TiCDC 服务进程列表
- 驱逐 owner 节点
- 手动触发表的负载均衡
- 手动调度表到其他节点
- 动态调整 TiCDC Server 日志级别

所有 API 的请求体与返回值统一使用 JSON 格式数据。本文档以下部分描述当前提供的 API 的具体使用方法。

在下文的示例描述中，假设 TiCDC server 的监听 IP 地址为 `127.0.0.1`，端口为 8300（在启动 TiCDC server 时可以通过 `--addr=ip:port` 指定绑定的 IP 和端口）。

13.9.4.9.1 API 统一错误格式

对 API 发起的请求后，如发生错误，返回错误信息的格式如下所示：

```
{
    "error_msg": "",
    "error_code": ""
}
```

如上所示，`error_msg` 描述错误信息，`error_code` 则是对应的错误码。

13.9.4.9.2 获取 TiCDC 节点状态信息

该接口是一个同步接口，请求成功会返回对应节点的状态信息。

请求 URI

```
GET /api/v1/status
```

使用样例

以下请求会获取 IP 地址为 `127.0.0.1`，端口号为 8300 的 TiCDC 节点的状态信息。

```
curl -X GET http://127.0.0.1:8300/api/v1/status
```

```
{
    "version": "v5.2.0-master-dirty",
    "git_hash": "f191cd00c53fdf7a2b1c9308a355092f9bf8824e",
    "id": "c6a43c16-0717-45af-afd6-8b3e01e44f5d",
    "pid": 25432,
    "is_owner": true
}
```

以上返回信息的字段解释如下：

- `version`：当前 TiCDC 版本号。
- `git_hash`：Git 哈希值。
- `id`：该节点的 capture ID。
- `pid`：该节点 capture 进程的 PID。
- `is_owner`：表示该节点是否是 owner。

13.9.4.9.3 检查 TiCDC 集群的健康状态

该接口是一个同步接口，在集群健康的时候会返回 `200 OK`。

请求 URI

```
GET /api/v1/health
```

使用样例

```
curl -X GET http://127.0.0.1:8300/api/v1/health
```

13.9.4.9.4 创建同步任务

该接口是一个异步接口，请求成功会返回 `202 Accepted`。该返回结果只代表服务器答应执行该命令，不保证命令会被成功的执行。

请求 URI

```
POST /api/v1/changefeeds
```

参数说明

使用 API 创建同步任务可选的参数不如使用 `cli` 命令创建同步任务的参数完备，以下是该 API 支持的参数。

请求体参数

参数名	说明
`changefeed_id`	STRING 类型，同步任务的 ID。（非必选）
`start_ts`	UINT64 类型，指定 changefeed 的开始 TSO。（非必选）
`target_ts`	UINT64 类型，指定 changefeed 的目标 TSO。（非必选）
`sink_uri`	STRING 类型，同步任务下游的地址。（必选）
`force_replicate`	BOOLEAN 类型，是否强制同步没有唯一索引的表。（非必选）

参数名	说明
ignore_ineligible_table	BOOLEAN 类型，是否忽略无法进行同步的表。（非必选）
filter_rules	STRING 类型数组，表库过滤的规则。（非必选）
ignore_txn_start_ts	UINT64 类型数组，忽略指定 start_ts 的事务。（非必选）
mounter_worker_num	INT 类型，Mounter 线程数。（非必选）
sink_config	sink 的配置参数。（非必选）

changefeed_id、start_ts、target_ts、sink_uri 的含义和格式与使用 cli 创建同步任务中所作的解释相同，具体解释请参见该文档。需要注意，当在 sink_uri 中指定证书的路径时，须确保已将对应证书上传到对应的 TiCDC server 上。

下面会对一些需要补充说明的参数进行进一步阐述。

force_replicate：该值默认为 false，当指定为 true 时，同步任务会尝试强制同步没有唯一索引的表。

ignore_ineligible_table：该值默认为 false，当指定为 true 时，同步任务会忽略无法进行同步的表。

filter_rules：表库过滤的规则，如 filter_rules = ['foo*.*', 'bar*.*'] 详情参考表库过滤。

ignore_txn_start_ts：指定之后会忽略指定 start_ts 的事务，如 ignore-txn-start-ts = [1, 2]。

mounter_worker_num：Mounter 线程数，Mounter 用于解码 TiKV 输出的数据，默认值为 16。

sink_config：sink 的配置参数，如下

```
{
  "dispatchers":[
    {"matcher":["test1.*", "test2.*"], "dispatcher":"ts"},
    {"matcher":["test3.*", "test4.*"], "dispatcher":"rowid"}
  ],
  "protocol":"canal-json"
}
```

dispatchers：对于 MQ 类的 Sink，可以通过 dispatchers 配置 event 分发器，支持 default、ts、rowid、table 四种分发器，分发规则如下：

- default：有多个唯一索引（包括主键）时按照 table 模式分发；只有一个唯一索引（或主键）按照 rowid 模式分发；如果开启了 old value 特性，按照 table 分发。
- ts：以行变更的 commitTs 做 Hash 计算并进行 event 分发。
- rowid：以所选的 HandleKey 列名和列值做 Hash 计算并进行 event 分发。
- table：以表的 schema 名和 table 名做 Hash 计算并进行 event 分发。

matcher：匹配语法和过滤器规则语法相同。

protocol：对于 MQ 类的 Sink，可以指定消息的协议格式。目前支持 canal-json、open-protocol、canal、avro 和 maxwell 五种协议。

使用样例

以下请求会创建一个 ID 为 test5，sink_uri 为 blackhole:// 的同步任务。

```
curl -X POST -H "'Content-type':'application/json'" http://127.0.0.1:8300/api/v1/changefeeds -d
    ↪ '{"changefeed_id":"test5","sink_uri":"blackhole://"}'
```

若是请求成功，则返回 202 Accepted，若请求失败，则返回错误信息和错误码。

13.9.4.9.5 删除同步任务

该接口是一个异步接口，请求成功会返回 202 Accepted，它只代表服务器答应执行该命令，不保证命令会被成功的执行。

请求 URI

```
DELETE /api/v1/changefeeds/{changefeed_id}
```

参数说明

路径参数

参数名	说明
changefeed_id	需要删除的同步任务 (changefeed) 的 ID

使用样例

以下请求会删除 ID 为 test1 的同步任务。

```
curl -X DELETE http://127.0.0.1:8300/api/v1/changefeeds/test1
```

若是请求成功，则返回 202 Accepted，若请求失败，则返回错误信息和错误码。

13.9.4.9.6 更新同步任务配置

该接口是一个异步接口，请求成功会返回 202 Accepted，它只代表服务器答应执行该命令，不保证命令会被成功的执行。

修改 changefeed 配置需要按照暂停任务 -> 修改配置 -> 恢复任务的流程。

请求 URI

```
PUT /api/v1/changefeeds/{changefeed_id}
```

参数说明

路径参数

参数名	说明
changefeed_id	需要更新的同步任务 (changefeed) 的 ID

请求体参数

目前仅支持通过 API 修改同步任务的如下配置。

参数名	说明
target_ts	UINT64 类型，指定 changefeed 的目标 TSO。(非必选)
sink_uri	STRING 类型，同步任务下游的地址。(非必选)
filter_rules	STRING 类型数组，表库过滤的规则。(非必选)
ignore_txn_start_ts	UINT64 类型数组，忽略指定 start_ts 的事务。 (非必选)
mounter_worker_num	INT 类型，mounter 线程数。(非必选)
sink_config	sink 的配置参数。(非必选)

以上参数含义与创建同步任务中的参数相同，此处不再赘述。

使用样例

以下请求会更新 ID 为 test1 的同步任务的 mounter_worker_num 为 32。

```
curl -X PUT -H "'Content-type':'application/json'" http://127.0.0.1:8300/api/v1/changefeeds/
    ↪ test1 -d '{"mounter_worker_num":32}'
```

若是请求成功，则返回 202 Accepted，若请求失败，则返回错误信息和错误码。

13.9.4.9.7 查询同步任务列表

该接口是一个同步接口，请求成功会返回 TiCDC 集群中所有同步任务 (changefeed) 的基本信息。

请求 URI

```
GET /api/v1/changefeeds
```

参数说明

查询参数

参数名	说明
state	非必选，指定后将会只返回该状态的同步任务的信息

state 可选值为 all、 normal、 stopped、 error、 failed、 finished。

若不指定该参数，则默认返回处于 normal、 stopped、 failed 状态的同步任务基本信息。

使用样例

以下请求查询所有状态 (state) 为 normal 的同步任务的基本信息。

```
curl -X GET http://127.0.0.1:8300/api/v1/changefeeds?state=normal
```

```
[
    {
        "id": "test1",
        "state": "normal",
        "checkpoint_tso": 426921294362574849,
        "checkpoint_time": "2021-08-10 14:04:54.242",
```

```
        "error": null
    },
    {
        "id": "test2",
        "state": "normal",
        "checkpoint_tso": 426921294362574849,
        "checkpoint_time": "2021-08-10 14:04:54.242",
        "error": null
    }
]
```

此处对以上返回的信息做进一步阐述：

- `id`：同步任务的 ID
- `state`：同步任务当前所处的状态。
- `checkpoint_tso`：同步任务当前 checkpoint 的 TSO 表示。
- `checkpoint_time`：同步任务当前 checkpoint 的格式化时间表示。
- `error`：同步任务的错误信息。

13.9.4.9.8 查询特定同步任务

该接口是一个同步接口，请求成功会返回指定同步任务 (changefeed) 的详细信息。

请求 URI

```
GET /api/v1/changefeeds/{changefeed_id}
```

参数说明

路径参数

参数名	说明
`changefeed_id`	需要查询的同步任务 (changefeed) 的 ID

使用样例

以下请求会查询 ID 为 `test1` 的同步任务的详细信息。

```
curl -X GET http://127.0.0.1:8300/api/v1/changefeeds/test1
```

```
{
    "id": "test1",
    "sink_uri": "blackhole://",
    "create_time": "2021-08-10 11:41:30.642",
    "start_ts": 426919038970232833,
    "target_ts": 0,
    "checkpoint_tso": 426921014615867393,
    "checkpoint_time": "2021-08-10 13:47:07.093",
```

```
    "sort_engine": "unified",
    "state": "normal",
    "error": null,
    "error_history": null,
    "creator_version": "",
    "task_status": [
        {
            "capture_id": "d8924259-f52f-4dfb-97a9-c48d26395945",
            "table_ids": [
                63,
                65
            ],
            "table_operations": {}
        }
    ]
}
```

13.9.4.9.9 暂停同步任务

该接口是一个异步接口，请求成功会返回 `202 Accepted`，它只代表服务器答应执行该命令，不保证命令会被成功的执行。

请求 URI

```
POST /api/v1/changefeeds/{changefeed_id}/pause
```

参数说明

路径参数

参数名	说明
changefeed_id	需要暂停的同步任务 (changefeed) 的 ID

使用样例

以下请求会暂停 ID 为 `test1` 的同步任务。

```
curl -X POST http://127.0.0.1:8300/api/v1/changefeeds/test1/pause
```

若是请求成功，则返回 `202 Accepted`，若请求失败，则返回错误信息和错误码。

13.9.4.9.10 恢复同步任务

该接口是一个异步接口，请求成功会返回 `202 Accepted`，它只代表服务器答应执行该命令，不保证命令会被成功的执行。

请求 URI

```
POST /api/v1/changefeeds/{changefeed_id}/resume
```

参数说明

路径参数

参数名	说明
changefeed_id	需要恢复的同步任务 (changefeed) 的 ID

使用样例

以下请求会恢复 ID 为 test1 的同步任务。

```
curl -X POST http://127.0.0.1:8300/api/v1/changefeeds/test1/resume
```

若是请求成功，则返回 202 Accepted，若请求失败，则返回错误信息和错误码。

13.9.4.9.11 查询同步子任务列表

该接口是一个同步接口，请求成功会返回当前 TiCDC 集群中的所有同步子任务 (processor) 的基本信息。

请求 URI

```
GET /api/v1/processors
```

使用样例

```
curl -X GET http://127.0.0.1:8300/api/v1/processors
```

```
[
    {
        "changefeed_id": "test1",
        "capture_id": "561c3784-77f0-4863-ad52-65a3436db6af"
    }
]
```

13.9.4.9.12 查询特定同步子任务

该接口是一个同步接口，请求成功会返回指定同步子任务 (processor) 的详细信息。

请求 URI

```
GET /api/v1/processors/{changefeed_id}/{capture_id}
```

参数说明

路径参数

参数名	说明
changefeed_id	需要查询的子任务的 Changefeed ID
capture_id	需要查询的子任务的 Capture ID

使用样例

以下请求查询 changefeed_id 为 test、 capture_id 为 561c3784-77f0-4863-ad52-65a3436db6af 的同步子任务。一个同步子任务通过 changefeed_id 和 capture_id 来标识。

```
curl -X GET http://127.0.0.1:8300/api/v1/processors/test1/561c3784-77f0-4863-ad52-65a3436db6af
```

```
{
    "checkpoint_ts": 426919123303006208,
    "resolved_ts": 426919123369066496,
    "table_ids": [
        63,
        65
    ],
    "error": null
}
```

13.9.4.9.13 查询 TiCDC 服务进程列表

该接口是一个同步接口，请求成功会返回当前 TiCDC 集群中的所有服务进程 (capture) 的基本信息。

请求 URI

```
GET /api/v1/captures
```

使用样例

```
curl -X GET http://127.0.0.1:8300/api/v1/captures
```

```
[
    {
        "id": "561c3784-77f0-4863-ad52-65a3436db6af",
        "is_owner": true,
        "address": "127.0.0.1:8300"
    }
]
```

13.9.4.9.14 驱逐 owner 节点

该接口是一个异步的请求，请求成功会返回 202 Accepted，它只代表服务器答应执行该命令，不保证命令会被成功的执行。

请求 URI

```
POST /api/v1/owner/resign
```

使用样例

以下请求会驱逐 TiCDC 当前的 owner 节点，并会触发新一轮的选举，产生新的 owner 节点。

```
curl -X POST http://127.0.0.1:8300/api/v1/owner/resign
```

若是请求成功，则返回 202 Accepted，若请求失败，则返回错误信息和错误码。

13.9.4.9.15 手动触发表的负载均衡

该接口是一个异步的请求，请求成功会返回 202 Accepted它只代表服务器答应执行该命令，不保证命令会被成功的执行。

请求 URI

```
POST /api/v1/changefeeds/{changefeed_id}/tables/rebalance_table
```

参数说明

路径参数

参数名	说明
changefeed_id	进行调度的 Changefeed ID

使用样例

以下请求会触发 ID 为 test1 的 changefeed 表的负载均衡。

```
curl -X POST http://127.0.0.1:8300/api/v1/changefeeds/test1/tables/rebalance_table
```

若是请求成功，则返回 202 Accepted，若请求失败，则返回错误信息和错误码。

13.9.4.9.16 手动调度表到其他节点

该接口是一个异步的请求，请求成功会返回 202 Accepted，它只代表服务器答应执行该命令，不保证命令会被成功的执行。

请求 URI

```
POST /api/v1/changefeeds/{changefeed_id}/tables/move_table
```

参数说明

路径参数

参数名	说明
changefeed_id	进行调度的 Changefeed ID

请求体参数

参数名	说明
target_capture_id	目标 Capture ID
table_id	需要调度的 Table ID

参数名	说明

使用样例

以下请求会将 ID 为 test1 的 changefeed 中 ID 为 49 的 table 调度到 ID 为 6f19a6d9-0f8c-4dc9-b299-3ba7c0f216f5 的 capture 上去。

```
curl -X POST -H "'Content-type':'application/json'" http://127.0.0.1:8300/api/v1/changefeeds/
    ↪ changefeed-test1/tables/move_table -d '{"capture_id":"6f19a6d9-0f8c-4dc9-b299-3
    ↪ ba7c0f216f5","table_id":49}'
```

若是请求成功，则返回 202 Accepted，若请求失败，则返回错误信息和错误码。

13.9.4.9.17 动态调整 TiCDC Server 日志级别

该接口是一个同步接口，请求成功会返回 200 OK。

请求 URI

POST /api/v1/log

请求参数

请求体参数

参数名	说明
log_level	想要设置的日志等级

log_level 支持 zap 提供的日志级别：“debug”、“info”、“warn”、“error”、“dpanic”、“panic”、“fatal”。

使用样例

```
curl -X POST -H "'Content-type':'application/json'" http://127.0.0.1:8300/api/v1/log -d '{"
    ↪ log_level":"debug"}'
```

若是请求成功，则返回 200 OK，若请求失败，则返回错误信息和错误码。

13.9.4.10 Storage sink 消费程序设计

本文介绍如何设计和实现一个 TiDB 数据变更的消费程序。

> **注意：**
>
> 当前 Storage sink 无法处理 DROP DATABASE DDL，请你尽量避免执行该语句。如果需要执行 DROP DATABASE DDL，请在下游 MySQL 手动执行。

TiCDC 不提供消费存储服务的数据的标准实现。本文介绍一个基于 Golang 的消费示例程序，该示例程序能够读取存储服务中的数据并写入到兼容 MySQL 的下游数据库。你可以参考本文提供的数据格式和以下示例代码实现消费端。

Golang 示例代码

13.9.4.10.1 Consumer 设计

下图是 Consumer 的整体消费流程：

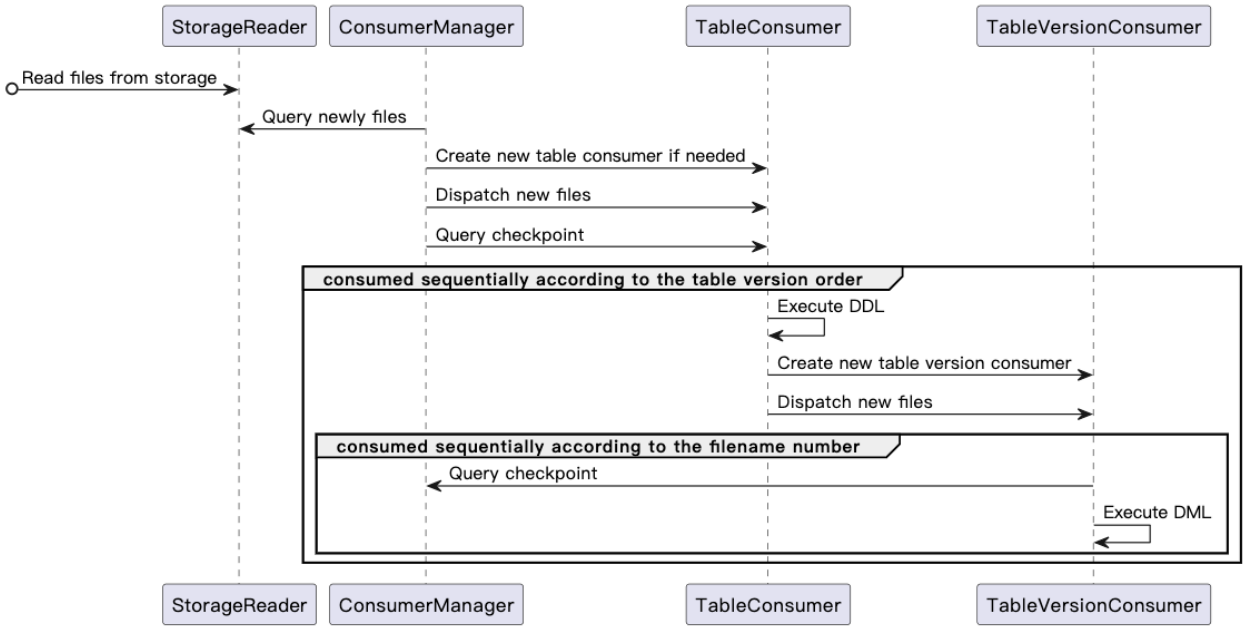


图 286: TiCDC storage consumer overview

以下是 Consumer 消费流程中的组件和功能定义，及其功能注释：

```
type StorageReader struct {
}

// Read the files from storage
// Add newly added files and delete files that not exist in storage
func (c *StorageReader) ReadFiles() {}

// Query newly added files and the latest checkpoint from storage, one file can only be returned
    ↪ once
func (c *StorageReader) ExposeNewFiles() (int64, []string) {}


// ConsumerManager is responsible for assigning tasks to TableConsumer.
// Different consumers can consume data concurrently,
```

```
// but data of one table must be processed by the same TableConsumer.
type ConsumerManager struct {
  // StorageCheckpoint is recorded in metadata file and it can be fetched by calling `
      ↪ StorageReader.ExposeNewFiles()`.
  // It indicates that the data whose transaction commit time is less than this checkpoint has
      ↪ been stored in storage
  StorageCheckpoint int64
  // it indicates where the consumer has consumed
  // ConsumerManager periodically collects TableConsumer.Checkpoint,
  // then Checkpoint is updated to the minimum value of all TableConsumer.Checkpoint
  Checkpoint int64

  tableFiles[schema][table]*TableConsumer
}

// Query newly files from StorageReader
// For new created table, create a TableConsumer for the new table
// If any, send new files to the corresponding TableConsumer
func (c *ConsumerManager) Dispatch() {}


type TableConsumer struct {
  // it indicates where this TableConsumer has consumed
  // Its initial value is ConsumerManager.Checkpoint
  // TableConsumer.Checkpoint is equal to TableVersionConsumer.Checkpoint
  Checkpoint int64

  schema,table string
  // Must be consumed sequentially according to the table version order
  verConsumers map[version int64]*TableVersionConsumer
  currentVer, previousVer int64
}

// Send newly files to the corresponding TableVersionConsumer
// For any DDL, assign a TableVersionConsumer for the new table version
func (tc *TableConsumer) Dispatch() {}

// If DDL query is empty or its tableVersion is less than TableConsumer.Checkpoint,
// - ignore this DDL, and consume the data under the table version
// Otherwise,
// - execute DDL first, and then consume the data under the table version
// - But for dropped table, self recycling after drop table DDL is executed
func (tc *TableConsumer) ExecuteDDL() {}
```

```
type TableVersionConsumer struct {
  // it indicates where the TableVersionConsumer has consumed
  // Its initial value is TableConsumer.Checkpoint
  Checkpoint int64

  schema,table,version string
  // For same table version, data in different partitions can be consumed concurrently
  # partitionNum int64
  // Must be consumed sequentially according to the data file number
  fileSet map[filename string]*TableVersionConsumer
  currentVersion
}

// If data commit ts is less than TableConsumer.Checkpoint
// or bigger than ConsumerManager.StorageCheckpoint,
// - ignore this data
// Otherwise,
// - process this data and write it to MySQL
func (tc *TableVersionConsumer) ExecuteDML() {}
```

13.9.4.10.2 DDL 事件的处理

举例来说，第一次遍历目录时目录内容如下：

```
├── metadata
└── test
    ├── tbl_1
    │   └── 437752935075545091
    │       ├── CDC000001.json
    │       └── schema.json
```

此时，首先需要解析 `schema.json` 文件中的表结构信息，从中获取 DDL Query 语句，并将其分为两种情况处理：

- 如果 DDL Query 为空或者 TableVersion 小于 consumer checkpoint 则跳过该语句。
- 否则，在下游 MySQL 数据库中执行获取到的 DDL 语句。

接着再开始同步数据文件 `CDC000001.json`。

例如，以下的 `test/tbl_1/437752935075545091/schema.json` 文件中 DDL Query 不为空：

```
{
    "Table":"test",
    "Schema":"tbl_1",
    "Version": 1,
    "TableVersion":437752935075545091,
    "Query": "create table tbl_1 (Id int primary key, LastName char(20), FirstName varchar(30),
        ↪ HireDate datetime, OfficeLocation Blob(20))",
```

```
    "TableColumns":[
        {
            "ColumnName":"Id",
            "ColumnType":"INT",
            "ColumnNullable":"false",
            "ColumnIsPk":"true"
        },
        {
            "ColumnName":"LastName",
            "ColumnType":"CHAR",
            "ColumnLength":"20"
        },
        {
            "ColumnName":"FirstName",
            "ColumnType":"VARCHAR",
            "ColumnLength":"30"
        },
        {
            "ColumnName":"HireDate",
            "ColumnType":"DATETIME"
        },
        {
            "ColumnName":"OfficeLocation",
            "ColumnType":"BLOB",
            "ColumnLength":"20"
        }
    ],
    "TableColumnsTotal":"5"
}
```

当程序再次遍历目录，发现该表新增了一个版本目录。程序先消费完 `test/tbl_1/437752935075545091` 目录下的所有文件，然后再消费新目录下的数据。

```
├── metadata
└── test
    ├── tbl_1
    │   ├── 437752935075545091
    │   │   ├── CDC000001.json
    │   │   └── schema.json
    │   └── 437752935075546092
    │   │   └── CDC000001.json
    │   │   └── schema.json
```

消费逻辑跟上述一致，先解析 `schema.json` 文件中的表结构信息，从中获取 DDL Query 语句并按不同情况处理，然后同步数据文件 `CDC000001.json`。

13.9.4.10.3 DML 事件的处理

在处理完 DDL 事件后，可以在 `{schema}/{table}/{table-version-separator}/` 目录下，根据具体的文件格式（CSV 或 Canal-JSON）并按照文件序号依次处理 DML 事件。

因为 TiCDC 提供 At Least Once 语义，可能出现重复发送数据的情况，所以需要在消费程序中比较数据事件的 commit ts 和 consumer checkpoint，并在 commit ts 小于 consumer checkpoint 的情况下进行去重处理。

13.9.4.11 TiCDC 兼容性

本文介绍了与 TiCDC 有关的一系列兼容性问题及其处理方案。

13.9.4.11.1 命令行参数和配置文件兼容性

- TiCDC v4.0.0 中移除了 `ignore-txn-commit-ts`，添加了 `ignore-txn-start-ts`，使用 start_ts 过滤事务。
- TiCDC v4.0.2 中移除了 `db-dbs/db-tables/ignore-dbs/ignore-tables`，添加了 `rules`，使用新版的数据库和数据表过滤规则，详细语法参考表库过滤。
- 自 TiCDC v6.2.0 开始，`cdc cli` 将通过 TiCDC 的 Open API 直接与 TiCDC server 进行交互，而不再需要访问 PD。`cdc cli` 子命令中的 `--pd` 参数被废除，增加了 `--server` 参数，用于指定 TiCDC Server 地址。请使用 `--server` 参数替代 `--pd` 参数。
- 从 v6.4.0 开始，TiCDC 使用 Syncpoint 功能需要同步任务拥有下游集群的 `SYSTEM_VARIABLES_ADMIN` 或者 `SUPER` 权限。

13.9.4.11.2 兼容性问题处理

本节介绍了兼容性相关的问题。

使用 TiCDC v5.0.0-rc 版本的 `cdc cli` 工具操作 v4.0.x 集群导致不兼容问题

使用 TiCDC v5.0.0-rc 版本的 `cdc cli` 工具操作 v4.0.x 版本的 TiCDC 集群时，可能会遇到如下异常情况：

- 若 TiCDC 集群版本为 v4.0.8 或以下，使用 v5.0.0-rc 版本的 `cdc cli` 创建同步任务 changefeed 时，可能导致 TiCDC 集群陷入异常状态，导致同步卡住。
- 若 TiCDC 集群版本为 v4.0.9 或以上，使用 v5.0.0-rc 版本的 `cdc cli` 创建同步任务 changefeed，会导致 Old Value 和 Unified Sorter 特性被非预期地默认开启。

处理方案：

使用和 TiCDC 集群版本对应的 `cdc` 可执行文件进行如下操作：

1. 删除使用 v5.0.0-rc 版本创建的 changefeed，例如：`tiup cdc:v4.0.9 cli changefeed remove -c xxxx --pd=xxxxx --force`。
2. 如果 TiCDC 同步已经卡住，重启 TiCDC 集群，例如：`tiup cluster restart <cluster_name> -R cdc`。
3. 重新创建 changefeed，例如：`tiup cdc:v4.0.9 cli changefeed create --sink-uri=xxxx --pd=xxx`。

> **注意：**
>
> 上述问题仅在 `cdc cli` 的版本是 v5.0.0-rc 时存在。未来其他 v5.0.x 版本的 `cdc cli` 可以兼容 v4.0.x 版本的集群。

`sort-dir` 及 `data-dir` 配置项的兼容性说明

`sort-dir` 配置项用于给 TiCDC 内部的排序器指定临时文件目录，其作用在各版本有过如下兼容性更改：

版本	sort-engine 的使用	说明	使用建议
版本	sort-engine 的使用	说明	使用建议
v4.0.11 及之前的 v4.0 版本，v5.0.0-rc	作为 changefeed 配置项，给 file sorter 和 unified Sorter 指定临时文件目录	在这些版本中，file sorter 和 unified sorter 均不是正式功能 (GA)，不推荐在生产环境中使用。如果有多个 changefeed 被配	不推荐在生产环境中使用 Unified Sorter

版本	sort-engine 的使用	说明	使用建议
v4.0.12，v4.0.13，v5.0.0 及 v5.0.1	作为 change-feed 配置项或 cdc server 配置项	在默认情况下 change-feed 的 sort-dir 配置不会生效，而 cdc server 的 sort-dir 配置默认为 /tmp/cdc_sort。建议生产环境下仅	需要通过 cdc server 命令行参数（或 TiUP）配置 sort-dir

版本	sort-engine 的使用	说明	使用建议
v4.0.14 及之后的 v4.0 版本，v5.0.3 及之后的 v5.0 版本，更新的版本	sort-dir 被弃用，建议配置 data-dir	data-dir 可以通过最新版本的 TiUP 进行配置。这些版本中 unified sorter 是默认开启的，升级时请确保 data-dir 已经被正	需要通过 cdc server 命令行参数（或 TiUP）配置 data-dir

版本	sort-engine 的使用	说明	使用建议
v6.0.0 及之后版本	data-dir 被用来存放 TiCDC 生成的临时文件	在该版本之后，TiCDC 默认采用 db sorter 作为内部的排序引擎，它使用 data-dir 作为磁盘目录。	需要通过 cdc server 命令行参数 --data-dir（或 TiUP）配置 data-dir

全局临时表兼容性说明

TiCDC 从 v5.3.0 开始支持全局临时表。

你需要使用 TiCDC v5.3.0 及以上版本同步全局临时表到下游。低于该版本，会导致表定义错误。

如果 TiCDC 的上游集群包含全局临时表，下游集群也必须是 TiDB 5.3.0 及以上版本，否则同步报错。

13.9.5 TiCDC 故障处理

本文档总结了使用 TiCDC 过程中常见的运行故障及解决方案。

> **注意：**
>
> 本文档 `cdc cli` 命令中指定 PD 地址为 `--server=http://127.0.0.1:8300`，在使用时需根据实际地址进行替换。

13.9.5.1 TiCDC 同步任务出现中断

13.9.5.1.1 如何判断 TiCDC 同步任务出现中断？

- 通过 Grafana 检查同步任务的 `changefeed checkpoint` 监控项。注意选择正确的 `changefeed id`。如果该值不发生变化或者查看 `checkpoint lag` 是否不断增大，可能同步任务出现中断。
- 通过 Grafana 检查 `exit error count` 监控项，该监控项大于 0 代表同步任务出现错误。
- 通过 `cdc cli changefeed list` 和 `cdc cli changefeed query` 命令查看同步任务的状态信息。任务状态为 `stopped` 代表同步中断，`error` 项会包含具体的错误信息。任务出错后可以在 TiCDC server 日志中搜索 `error on running processor` 查看错误堆栈，帮助进一步排查问题。
- 部分极端异常情况下 TiCDC 出现服务重启，可以在 TiCDC server 日志中搜索 FATAL 级别的日志排查问题。

13.9.5.1.2 如何查看 TiCDC 同步任务是否被人为终止？

可以使用 `cdc cli` 查询同步任务是否被人为终止。例如：

```
cdc cli changefeed query --server=http://127.0.0.1:8300 --changefeed-id 28c43ffc-2316-4f4f-a70b-
    ↪ d1a7c59ba79f
```

上述命令的输出中 `admin-job-type` 标志这个同步的任务的状态：

- 0: 任务进行中，没有被人为停止。
- 1: 任务暂停，停止任务后所有同步 `processor` 会结束退出，同步任务的配置和同步状态都会保留，可以从 `checkpoint-ts` 恢复任务。
- 2: 任务恢复，同步任务从 `checkpoint-ts` 继续同步。
- 3: 任务已删除，接口请求后会结束所有同步 `processor`，并清理同步任务配置信息。同步状态保留，只提供查询，没有其他实际功能。

13.9.5.1.3 如何处理 TiCDC 同步任务的中断？

目前已知可能发生的同步中断包括以下场景：

- 下游持续异常，TiCDC 多次重试后仍然失败。
  - 该场景下 TiCDC 会保存任务信息，由于 TiCDC 已经在 PD 中设置的 service GC safepoint，在 `gc-ttl` 的有效期内，同步任务 checkpoint 之后的数据不会被 TiKV GC 清理掉。
  - 处理方法：在下游恢复正常后，通过 HTTP 接口恢复同步任务。
- 因下游存在不兼容的 SQL 语句，导致同步不能继续。
  - 该场景下 TiCDC 会保存任务信息，由于 TiCDC 已经在 PD 中设置的 service GC safepoint，在 `gc-ttl` 的有效期内，同步任务 checkpoint 之后的数据不会被 TiKV GC 清理掉。
  - 处理方法：
    1. 先通过 `cdc cli changefeed query` 查询同步任务状态信息，记录 `checkpoint-ts` 值。
    2. 使用新的任务配置文件，增加`ignore-txn-start-ts` 参数跳过指定 `start-ts` 对应的事务。
    3. 通过 HTTP API 停止旧的同步任务，使用 `cdc cli changefeed create`，指定新的任务配置文件，指定 `start-ts` 为刚才记录的 `checkpoint-ts`，启动新的同步任务恢复同步。
- 在 v4.0.13 及之前的版本中 TiCDC 同步分区表存在问题，导致同步停止。
  - 该场景下 TiCDC 会保存任务信息，由于 TiCDC 已经在 PD 中设置的 service GC safepoint，在 `gc-ttl` 的有效期内，同步任务 checkpoint 之后的数据不会被 TiKV GC 清理掉。
  - 处理方法：
    1. 通过 `cdc cli changefeed pause -c <changefeed-id>` 暂停同步。
    2. 等待约一分钟后，通过 `cdc cli changefeed resume -c <changefeed-id>` 恢复同步。

13.9.5.1.4 同步任务中断，尝试再次启动后 TiCDC 发生 OOM，应该如何处理？

升级 TiDB 集群和 TiCDC 集群到最新版本。该 OOM 问题在 v4.0.14 及之后的 v4.0 版本，v5.0.2 及之后的 v5.0 版本，更新的版本上已得到缓解。

13.9.5.2 如何处理 TiCDC 创建同步任务或同步到 MySQL 时遇到 `Error 1298: Unknown or incorrect time zone: 'UTC'` 错误？

这是因为下游 MySQL 没有加载时区，可以通过 mysql_tzinfo_to_sql 命令加载时区，加载后就可以正常创建任务或同步任务。

```
mysql_tzinfo_to_sql /usr/share/zoneinfo | mysql -u root mysql -p
```

显示类似于下面的输出则表示导入已经成功：

```
Enter password:
Warning: Unable to load '/usr/share/zoneinfo/iso3166.tab' as time zone. Skipping it.
Warning: Unable to load '/usr/share/zoneinfo/leap-seconds.list' as time zone. Skipping it.
Warning: Unable to load '/usr/share/zoneinfo/zone.tab' as time zone. Skipping it.
Warning: Unable to load '/usr/share/zoneinfo/zone1970.tab' as time zone. Skipping it.
```

如果下游是特殊的 MySQL 环境（某种公有云 RDS 或某些 MySQL 衍生版本等），使用上述方式导入时区失败，就需要通过 sink-uri 中的 `time-zone` 参数指定下游的 MySQL 时区。可以首先在 MySQL 中查询其使用的时区：

```
show variables like '%time_zone%';
```

```
+------------------+--------+
| Variable_name    | Value  |
+------------------+--------+
| system_time_zone | CST    |
| time_zone        | SYSTEM |
+------------------+--------+
```

然后在创建同步任务和创建 TiCDC 服务时使用该时区：

```
cdc cli changefeed create --sink-uri="mysql://root@127.0.0.1:3306/?time-zone=CST" --server=http
    ↪ ://127.0.0.1:8300
```

> 注意：
>
> CST 可能是以下四个不同时区的缩写：
>
> - 美国中部时间：Central Standard Time (USA) UT-6:00
> - 澳大利亚中部时间：Central Standard Time (Australia) UT+9:30
> - 中国标准时间：China Standard Time UT+8:00
> - 古巴标准时间：Cuba Standard Time UT-4:00
>
> 在中国，CST 通常表示中国标准时间，使用时请注意甄别。

13.9.5.3 如何处理升级 TiCDC 后配置文件不兼容的问题？

请参阅配置文件兼容注意事项。

13.9.5.4 TiCDC 启动任务的 start-ts 时间戳与当前时间差距较大，任务执行过程中同步中断，出现错误 `[CDC:ErrBufferReachLimit]`，怎么办？

自 v4.0.9 起可以尝试开启 unified sorter 特性进行同步；或者使用 BR 工具进行一次增量备份和恢复，然后从新的时间点开启 TiCDC 同步任务。TiCDC 将会在后续版本中对该问题进行优化。

13.9.5.5 当 changefeed 的下游为类 MySQL 数据库时，TiCDC 执行了一个耗时较长的 DDL 语句，阻塞了所有其他 changefeed，应该怎样处理？

1. 首先暂停执行耗时较长的 DDL 的 changefeed。此时可以观察到，这个 changefeed 暂停后，其他的 changefeed 不再阻塞了。
2. 在 TiCDC log 中搜寻 `apply job` 字段，确认耗时较长的 DDL 的 `start-ts`。

3. 手动在下游执行该 DDL 语句，执行完毕后进行下面的操作。
4. 修改 changefeed 配置，将上述 `start-ts` 添加到 `ignore-txn-start-ts` 配置项中。
5. 恢复被暂停的 changefeed。

13.9.5.6 TiCDC 集群升级到 v4.0.8 之后，changefeed 报错 `[CDC:ErrKafkaInvalidConfig]Canal requires old value to be enabled`，为什么？

自 v4.0.8 起，如果 changefeed 使用 `canal-json`、`canal` 或者 `maxwell` 协议输出，TiCDC 会自动开启 Old Value 功能。但是，当 TiCDC 是从较旧版本升级到 v4.0.8 或以上版本时，在 changefeed 使用 `canal-json`、`canal` 或 `maxwell` 协议的同时 TiCDC 的 Old Value 功能会被禁用。此时，会出现该报错。可以按照以下步骤解决该报错：

1. 将 changefeed 配置文件中 `enable-old-value` 的值设为 `true`。
2. 使用 `cdc cli changefeed pause` 暂停同步任务。

```
cdc cli changefeed pause -c test-cf --server=http://127.0.0.1:8300
```

3. 使用 `cdc cli changefeed update` 更新原有 changefeed 的配置。

```
cdc cli changefeed update -c test-cf --server=http://127.0.0.1:8300 --sink-uri="mysql
    ↪ ://127.0.0.1:3306/?max-txn-row=20&worker-number=8" --config=changefeed.toml
```

4. 使用 `cdc cli changefeed resume` 恢复同步任务。

```
cdc cli changefeed resume -c test-cf --server=http://127.0.0.1:8300
```

13.9.5.7 使用 TiCDC 创建 changefeed 时报错 `[tikv:9006]GC life time is shorter than transaction duration, transaction starts at xx, GC safe point is yy`，该如何处理？

执行 `pd-ctl service-gc-safepoint --pd <pd-addrs>` 命令查询当前的 GC safepoint 与 service GC safepoint。如果 GC safepoint 小于 TiCDC changefeed 同步任务的开始时间戳 `start-ts`，可以直接在 `cdc cli create changefeed` 命令后加上 `--disable-gc-check` 参数创建 changefeed。

如果 `pd-ctl service-gc-safepoint --pd <pd-addrs>` 的结果中没有 `gc_worker service_id`：

- 如果 PD 的版本 <= v4.0.8，详见 PD issue #3128。
- 如果 PD 是由 v4.0.8 或更低版本滚动升级到新版，详见 PD issue #3366。
- 对于其他情况，请将上述命令执行结果反馈到 AskTUG 论坛。

13.9.5.8 使用 TiCDC 同步消息到 Kafka 时 Kafka 报错 `Message was too large`，该如何处理？

v4.0.8 或更低版本的 TiCDC，仅在 Sink URI 中为 Kafka 配置 `max-message-bytes` 参数不能有效控制输出到 Kafka 的消息大小，需要在 Kafka server 配置中加入如下配置以增加 Kafka 接收消息的字节数限制。

```
### broker 能接收消息的最大字节数
message.max.bytes=2147483648
### broker 可复制的消息的最大字节数
replica.fetch.max.bytes=2147483648
```

```
### 消费者端的可读取的最大消息字节数
fetch.message.max.bytes=2147483648
```

13.9.5.9 TiCDC 同步时，在下游执行 DDL 语句失败会有什么表现，如何恢复？

如果某条 DDL 语句执行失败，同步任务 (changefeed) 会自动停止，checkpoint-ts 断点时间戳为该条出错 DDL 语句的结束时间戳 (finish-ts) 减去一。如果希望让 TiCDC 在下游重试执行这条 DDL 语句，可以使用 `cdc cli changefeed resume` 恢复同步任务。例如：

```
cdc cli changefeed resume -c test-cf --server=http://127.0.0.1:8300
```

如果希望跳过这条出错的 DDL 语句，可以将 changefeed 的 start-ts 设为报错时的 checkpoint-ts 加上一，然后通过 `cdc cli changefeed create` 新建同步任务。假设报错时的 checkpoint-ts 为 415241823337054209，可以进行如下操作来跳过该 DDL 语句：

```
cdc cli changefeed remove --server=http://127.0.0.1:8300 --changefeed-id simple-replication-task
cdc cli changefeed create --server=http://127.0.0.1:8300 --sink-uri="mysql://root:123456@127
    ↪ .0.0.1:3306/" --changefeed-id="simple-replication-task" --sort-engine="unified" --start-
    ↪ ts 415241823337054210
```

13.9.6 TiCDC 常见问题解答

本文档总结了使用 TiCDC 时经常遇到的问题。

> **注意：**
>
> 本文档 `cdc cli` 命令中指定 server 地址为 `--server=http://127.0.0.1:8300`，在使用时你需要根据实际地址进行替换。

13.9.6.1 TiCDC 创建任务时如何选择 start-ts？

首先需要理解同步任务的 `start-ts` 对应于上游 TiDB 集群的一个 TSO，同步任务会从这个 TSO 开始请求数据。所以同步任务的 `start-ts` 需要满足以下两个条件：

- `start-ts` 的值需要大于 TiDB 集群当前的 `tikv_gc_safe_point`，否则创建任务时会报错。
- 启动任务时，需要保证下游已经具有 `start-ts` 之前的所有数据。对于同步到消息队列等场景，如果不需要保证上下游数据的一致，可根据业务场景放宽此要求。

如果不指定 `start-ts` 或者指定 `start-ts=0`，在启动任务的时候会去 PD 获取一个当前 TSO，并从该 TSO 开始同步。

13.9.6.2 为什么 TiCDC 创建任务时提示部分表不能同步？

在使用 `cdc cli changefeed create` 创建同步任务时会检查上游表是否符合同步要求。如果存在表不满足同步限制，会提示 `some tables are not eligible to replicate` 并列出这些不满足的表。如果选择 `Y` 或 `y` 则会继续创建同步任务，并且同步过程中自动忽略这些表的所有更新。如果选择其他输入，则不会创建同步任务。

13.9.6.3 如何查看 TiCDC 同步任务的状态？

可以使用 `cdc cli` 查询同步任务的状态。例如：

```
cdc cli changefeed list --server=http://127.0.0.1:8300
```

上述命令输出如下：

```
[{
    "id": "4e24dde6-53c1-40b6-badf-63620e4940dc",
    "summary": {
      "state": "normal",
      "tso": 417886179132964865,
      "checkpoint": "2020-07-07 16:07:44.881",
      "error": null
    }
}]
```

- `checkpoint`：即为 TiCDC 已经将该时间点前的数据同步到了下游。
- `state` 为该同步任务的状态：
    - `normal`：正常同步。
    - `stopped`：停止同步（手动暂停或出错）。
    - `removed`：已删除任务。

> **注意：**
>
> 该功能在 TiCDC 4.0.3 版本引入。

13.9.6.4 TiCDC 的 `gc-ttl` 是什么？

从 TiDB v4.0.0-rc.1 版本起，PD 支持外部服务设置服务级别 GC safepoint。任何一个服务可以注册更新自己服务的 GC safepoint。PD 会保证任何晚于该 GC safepoint 的 KV 数据不会在 TiKV 中被 GC 清理掉。

在 TiCDC 中启用了这一功能，用来保证 TiCDC 在不可用、或同步任务中断情况下，可以在 TiKV 内保留 TiCDC 需要消费的数据不被 GC 清理掉。

启动 TiCDC server 时可以通过 `gc-ttl` 指定 GC safepoint 的 TTL，也可以通过 TiUP 修改 TiCDC 的 `gc-ttl`，默认值为 24 小时。在 TiCDC 中这个值有如下两重含义：

- 当 TiCDC 服务全部停止后，由 TiCDC 在 PD 所设置的 GC safepoint 保存的最长时间。

- TiCDC 中某个同步任务中断或者被手动停止时所能停滞的最长时间，若同步任务停滞时间超过 `gc-ttl` 所设置的值，那么该同步任务就会进入 `failed` 状态，无法被恢复，并且不会继续影响 GC safepoint 的推进。

以上第二种行为是在 TiCDC v4.0.13 版本及之后版本中新增的。目的是为了防止 TiCDC 中某个同步任务停滞时间过长，导致上游 TiKV 集群的 GC safepoint 长时间不推进，保留的旧数据版本过多，进而影响上游集群性能。

> **注意：**
>
> 在某些应用场景中，比如使用 Dumpling/BR 全量同步后使用 TiCDC 接增量同步时，默认的 `gc-ttl` 为 24 小时可能无法满足需求。此时应该根据实际情况，在启动 TiCDC server 时指定 `gc-ttl` 的值。

13.9.6.5 TiCDC GC safepoint 的完整行为是什么？

TiCDC 服务启动后，如果有任务开始同步，TiCDC owner 会根据所有同步任务最小的 checkpoint-ts 更新到 PD service GC safepoint，service GC safepoint 可以保证该时间点及之后的数据不被 GC 清理掉。如果 TiCDC 中某个同步任务中断、或者被用户主动停止，则该任务的 checkpoint-ts 不会再改变，PD 对应的 service GC safepoint 最终会停滞在该任务的 checkpoint-ts 处不再更新。

如果该同步任务停滞的时间超过了 `gc-ttl` 指定的时长，那么该同步任务就会进入 `failed` 状态，并且无法被恢复，PD 对应的 service GC safepoint 就会继续推进。

TiCDC 为 service GC safepoint 设置的存活有效期为 24 小时，即 TiCDC 服务中断 24 小时内恢复能保证数据不因 GC 而丢失。

13.9.6.6 如何理解 TiCDC 时区和上下游数据库系统时区之间的关系？

	上游时区	TiCDC 时区	下游时区
配置方式	见时区支持	启动 ticdc server 时的 `--tz` 参数	sink-uri 中的 `time-zone` 参数

	上游时区	TiCDC 时区	下游时区
说明	上游 TiDB 的时区，影响 timestamp 类型的 DML 操作和与 timestamp 类型列相关的 DDL 操作。	TiCDC 会将假设上游 TiDB 的时区和 TiCDC 时区配置相同，对 timestamp 类型的列进行相关处理。	下游 MySQL 将按照下游的时区设置对 DML 和 DDL 操作中包含的 timestamp 进行处理。

注意：

请谨慎设置 TiCDC server 的时区，因为该时区会用于时间类型的转换。上游时区、TiCDC 时区和下游时区应该保持一致。TiCDC server 时区使用的优先级如下：

- 最优先使用 `--tz` 传入的时区。

- 没有 `--tz` 参数，会尝试读取 TZ 环境变量设置的时区。
- 如果还没有 TZ 环境变量，会从 TiCDC server 运行机器的默认时区。

13.9.6.7 创建同步任务时，如果不指定 `--config` 配置文件，TiCDC 的默认的行为是什么？

在使用 `cdc cli changefeed create` 命令时如果不指定 `--config` 参数，TiCDC 会按照以下默认行为创建同步任务：

- 同步所有的非系统表
- 开启 old value 功能
- 只同步包含有效索引的表

13.9.6.8 TiCDC 是否支持输出 Canal 格式的变更数据？

支持。要开启 Canal 格式输出，只需在 `--sink-uri` 中指定 protocol 为 `canal` 即可，例如：

```
cdc cli changefeed create --server=http://127.0.0.1:8300 --sink-uri="kafka://127.0.0.1:9092/cdc-
    ↪ test?kafka-version=2.4.0&protocol=canal" --config changefeed.toml
```

注意：

- 该功能在 TiCDC 4.0.2 版本引入。
- 目前 TiCDC 仅支持将 Canal 格式的变更数据输出到 MQ 类的 Sink（例如 Kafka）。

更多信息请参考TiCDC Changefeed 配置参数。

13.9.6.9 为什么 TiCDC 到 Kafka 的同步任务延时越来越大？

- 请参考如何查看 TiCDC 同步任务的状态？检查下同步任务的状态是否正常。
- 请适当调整 Kafka 的以下参数：
  - `message.max.bytes`，将 Kafka 的 `server.properties` 中该参数调大到 1073741824 (1 GB)。
  - `replica.fetch.max.bytes`，将 Kafka 的 `server.properties` 中该参数调大到 1073741824 (1 GB)。
  - `fetch.message.max.bytes`，适当调大 `consumer.properties` 中该参数，确保大于 `message.max.bytes`。

13.9.6.10 TiCDC 把数据同步到 Kafka 时，能在 TiDB 中控制单条消息大小的上限吗？

对于 Avro 和 Canal-JSON 格式，消息是以行变更为单位发送的，一条 Kafka Message 仅包含一条行变更。一般情况下，消息的大小不会超过 Kafka 单条消息上限，因此，一般不需要限制单条消息大小。如果单条 Kafka 消息大小确实超过 Kakfa 上限，请参考为什么 TiCDC 到 Kafka 的同步任务延时越来越大。

对于 Open Protocol 格式，一条 Kafka Message 可能包含多条行变更。因此，有可能存在某条 Kafka Message 消息过大。可以通过 `max-message-bytes` 控制每次向 Kafka broker 发送消息的最大数据量（可选，默认值 10 MB），通过 `max-batch-size` 参数指定每条 kafka 消息中变更记录的最大数量（可选，默认值 16）。

13.9.6.11 在一个事务中对一行进行多次修改，TiCDC 会输出多条行变更事件吗？

不会，在进行事务操作时，对于在一个事务内多次修改同一行的情况，TiDB 仅会将最新一次的修改结果发送给 TiKV。因此 TiCDC 仅能获取到最新一次修改的结果。

13.9.6.12 TiCDC 把数据同步到 Kafka 时，一条消息中会不会包含多种数据变更？

会，一条消息中可能出现多个 update 或 delete，update 和 delete 也有可能同时存在。

13.9.6.13 TiCDC 把数据同步到 Kafka 时，如何查看 TiCDC Open protocol 输出变更数据中的时间戳、表名和库名？

这些信息包含在 Kafka 消息的 Key 中，比如：

```
{
    "ts":<TS>,
    "scm":<Schema Name>,
    "tbl":<Table Name>,
    "t":1
}
```

更多信息请参考Open protocol Event 格式定义

13.9.6.14 TiCDC 把数据同步到 Kafka 时，如何确定一条消息中包含的数据变更发生在哪个时间点？

把 Kafka 消息的 Key 中的 ts 右移 18 位即得 unix timestamp。

13.9.6.15 TiCDC Open protocol 如何标示 null 值？

Open protocol 的输出中 type = 6 即为 null，比如：

类型	Code	输出示例	说明
Null	6	{"t":6,"v":null}	

更多信息请参考Open protocol Event 格式定义。

13.9.6.16 如何区分 TiCDC Open Protocol 中的 Row Changed Event 是 INSERT 事件还是 UPDATE 事件？

如果没有开启 Old Value 功能，你无法区分 TiCDC Open Protocol 中的 Row Changed Event 是 INSERT 事件还是 UPDATE 事件。如果开启了 Old Value 功能，则可以通过事件中的字段判断事件类型：

- 如果同时存在 "p" 和 "u" 字段为 UPDATE 事件
- 如果只存在 "u" 字段则为 INSERT 事件
- 如果只存在 "d" 字段则为 DELETE 事件

更多信息请参考Open protocol Row Changed Event 格式定义。

13.9.6.17 TiCDC 占用多少 PD 的存储空间

TiCDC 使用 PD 内部的 etcd 来存储元数据并定期更新。因为 etcd 的多版本并发控制 (MVCC) 以及 PD 默认的 compaction 间隔是 1 小时，TiCDC 占用的 PD 存储空间与 1 小时内元数据的版本数量成正比。在 v4.0.5、v4.0.6、v4.0.7 三个版本中 TiCDC 存在元数据写入频繁的问题，如果 1 小时内有 1000 张表创建或调度，就会用尽 etcd 的存储空间，出现 `etcdserver: mvcc: database space exceeded` 错误。出现这种错误后需要清理 etcd 存储空间，参考 etcd maintenance space-quota。如果你的 TiCDC 版本为 v4.0.5、v4.0.6 或 v4.0.7，建议升级到 v4.0.9 及以后版本。

13.9.6.18 TiCDC 支持同步大事务吗？有什么风险吗？

TiCDC 对大事务（大小超过 5 GB）提供部分支持，根据场景不同可能存在以下风险：

- 可能导致主从同步延迟大幅增高。
- 当 TiCDC 内部处理能力不足时，可能出现同步任务报错 `ErrBufferReachLimit`。
- 当 TiCDC 内部处理能力不足或 TiCDC 下游吞吐能力不足时，可能出现内存溢出 (OOM)。

从 v6.2 版本开始，TiCDC 支持拆分单表事务功能，可大幅降低同步大事务的延时和内存消耗。因此，在业务对事务原子性要求不高的场景下，建议通过设置 sink uri 参数`transaction-atomicity` 打开拆分事务功能以解决可能出现的同步延迟和 OOM 问题。

如果实际同步过程中仍然遇到了上述错误，建议将包含大事务部分的增量数据通过 BR 进行增量恢复，具体操作如下：

1. 记录因为大事务而终止的 changefeed 的 `checkpoint-ts`，将这个 TSO 作为 BR 增量备份的 `--lastbackupts`，并执行增量备份。
2. 增量备份结束后，可以在 BR 日志输出中找到类似 `["Full backup Failed summary : total backup ranges: 0, total success: 0, total failed: 0"] [BackupTS=421758868510212097]` 的日志，记录其中的 BackupTS。
3. 执行增量恢复。
4. 建立一个新的 changefeed，从 BackupTS 开始同步任务。
5. 删除旧的 changefeed。

13.9.6.19 同步 DDL 到下游 MySQL 5.7 时为什么时间类型字段默认值不一致？

比如上游 TiDB 的建表语句为 `create table test (id int primary key, ts timestamp)`，TiCDC 同步该语句到下游 MySQL 5.7，MySQL 使用默认配置，同步得到的表结构如下所示，timestamp 字段默认值会变成 `CURRENT_TIMESTAMP`：

```
mysql root@127.0.0.1:test> show create table test;
+-------+-----------------------------------------------------------------------------------------+
| Table | Create Table                                                                            |
+-------+-----------------------------------------------------------------------------------------+
| test  | CREATE TABLE `test` (                                                                   |
|       |   `id` int(11) NOT NULL,                                                                |
|       |   `ts` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,        |
|       |   PRIMARY KEY (`id`)                                                                    |
|       | ) ENGINE=InnoDB DEFAULT CHARSET=latin1                                                  |
```

```
+-------+-------------------------------------------------------------------------------------+
1 row in set
```

产生表结构不一致的原因是 explicit_defaults_for_timestamp 的默认值在 TiDB 和 MySQL 5.7 不同。从 TiCDC v5.0.1/v4.0.13 版本开始，同步到 MySQL 会自动设置 session 变量 explicit_defaults_for_timestamp = ↪ ON，保证同步时间类型时上下游行为一致。对于 v5.0.1/v4.0.13 以前的版本，同步时间类型时需要注意 explicit_defaults_for_timestamp 默认值不同带来的兼容性问题。

13.9.6.20 使用 TiCDC 创建同步任务时将 safe-mode 设置为 true 后，为什么上游的 INSERT/UPDATE 语句经 TiCDC 同步到下游后变为了 REPLACE INTO？

TiCDC 提供至少一次的数据同步保证，当下游有重复数据时，会引起写冲突。为了避免该问题，TiCDC 会将 INSERT 和 UPDATE 语句转成 REPLACE INTO 语句。该行为由 safe-mode 参数来控制。

在 v6.1.3 版本之前，safe-mode 默认为 true，即所有的 INSERT 和 UPDATE 语句都转成 REPLACE INTO 语句。在 v6.1.3 及之后版本，系统能自动判断下游是否存在重复数据，safe-mode 默认更改为 false，当系统判断下游无重复数据时，会直接同步 INSERT 和 UPDATE 语句。

13.9.6.21 数据同步下游的 Sink 为 TiDB 或 MySQL 时，下游数据库的用户需要哪些权限？

Sink 为 TiDB 或 MySQL 时，下游数据库的用户需要以下权限：

- Select
- Index
- Insert
- Update
- Delete
- Create
- Drop
- Alter
- Create View

如果要同步 recover table 到下游 TiDB，需要有 Super 权限。

13.9.6.22 为什么 TiCDC 需要使用磁盘，什么时候会写磁盘，TiCDC 能否利用内存缓存提升同步性能？

TiCDC 需要磁盘是为了缓冲上游写入高峰时下游消费不及时堆积的数据。TiCDC 正常运行期间都需要写入磁盘，但这通常不是同步吞吐和同步延时的瓶颈，写磁盘对延时影响在百毫秒内。TiCDC 也利用了内存来提升加速读取磁盘中的数据，以提升同步性能。

13.9.6.23 为什么在上游使用了 TiDB Lightning 和 BR 恢复了数据之后，TiCDC 同步会出现卡顿甚至卡住？

目前 TiCDC 还没完全适配 TiDB Lightning 和 BR，请避免在使用 TiCDC 同步的表上使用 TiDB Lightning 和 BR。

13.9.6.24 为什么恢复暂停的 changefeed 后，changefeed 同步延迟越来越高，数分钟后才恢复正常？

当 changefeed 启动时，为了补齐 changefeed 暂停期间产生的增量数据日志，TiCDC 需要扫描 TiKV 中数据的历史版本，待扫描完毕后，才能够继续推进复制过程，扫描过程可能长达数分钟到数十分钟。

13.9.6.25 在两个异地 TiDB 集群之间同步数据，如何部署 TiCDC？

建议将 TiCDC 部署在下游 TiDB 集群。这是因为，如果上下游网络延迟较大，例如超过 100 ms 时，由于 MySQL 传输协议的原因，TiCDC 向下游执行 SQL 的延迟会急剧增加，导致系统的吞吐下降。部署在下游能够极大缓解该问题。

13.9.6.26 如何理解 DML 和 DDL 语句之间的执行顺序？

按照 DML -> DDL -> DML 的顺序执行。在数据同步过程中，为了确保 DML 事件在下游执行时有对应正确的表结构，需要协调 DDL 和 DML 的执行顺序。目前 TiCDC 采用了简洁的方式处理该问题，会将 DDL ts 之前的 DML 都同步到下游之后，再同步 DDL。

13.9.6.27 如何对比上下游数据的一致性？

如果下游是 TiDB 集群或者 MySQL，我们推荐使用sync diff inspector 工具进行数据对比。

13.9.6.28 单表数据同步只能在一个 TiCDC 节点上运行，TiCDC 是否考虑使用多个节点同步多表数据？

目前正在开发中，未来 TiCDC 会支持按照 TiKV Region 粒度来同步数据变更日志，实现处理能力上的可扩展性。

13.9.6.29 上游有运行时间比较长的未提交事务，TiCDC 同步是否会被卡住？

TiDB 有事务超时的机制，当事务运行超过`max-txn-ttl` 后，会被 TiDB 强制回滚。TiCDC 遇到未提交的事务，会等待其提交后再继续同步其数据，因此会出现同步延迟。

13.9.6.30 为什么通过 TiDB Operator 部署的 TiCDC 集群无法使用 cdc cli 命令进行操作？

因为通过 TiDB Operator 部署的 TiCDC 集群的默认端口号为 8301, 而 cdc cli 命令默认连接的 cdc 服务器的端口号是 8300。在使用 cdc cli 操作 TiCDC 集群时，你需要显式地指定 `--server` 参数，如下：

```
./cdc cli changefeed list --server "127.0.0.1:8301"
[
  {
    "id": "4k-table",
    "namespace": "default",
    "summary": {
      "state": "stopped",
      "tso": 441832628003799353,
      "checkpoint": "2023-05-30 22:41:57.910",
      "error": null
    }
  },
  {
    "id": "big-table",
    "namespace": "default",
    "summary": {
      "state": "normal",
```

```
      "tso": 441872834546892882,
      "checkpoint": "2023-06-01 17:18:13.700",
      "error": null
    }
  }
]
```

13.9.7 TiCDC 术语表

本术语表提供 TiCDC 相关的术语和定义，这些术语会出现在 TiCDC 的日志、监控指标、配置和文档中。

13.9.7.1 B

13.9.7.1.1 变更数据

从上游 TiDB 集群写入 TiCDC 的数据，包括 DML 操作引发的数据变更和 DDL 操作引发的表结构变更。

13.9.7.2 C

13.9.7.2.1 Capture

单个 TiCDC 实例。多个 Capture 组成一个 TiCDC 集群，Capture 上运行集群中的同步任务。

13.9.7.2.2 Changefeed

TiCDC 中的单个同步任务。同步任务将一个 TiDB 集群中数张表的变更数据输出到一个指定的下游中。

13.9.7.3 O

13.9.7.3.1 Owner

一个特殊角色的Capture，负责管理 TiCDC 集群和调度 TiCDC 集群中的同步任务。该角色由 Capture 选举产生，在任意时刻最多只存在一个。

13.9.7.4 P

13.9.7.4.1 Processor

TiCDC 同步任务会在 TiCDC 实例上分配数据表，Processor 指这些数据表的同步处理单元。处理任务包括变更数据的拉取、排序、还原和分发。

13.10 TiDB Binlog

13.10.1 TiDB Binlog 简介

TiDB Binlog 是一个用于收集 TiDB 的 binlog，并提供准实时备份和同步功能的商业工具。

TiDB Binlog 支持以下功能场景：

- 数据同步：同步 TiDB 集群数据到其他数据库
- 实时备份和恢复：备份 TiDB 集群数据，同时可以用于 TiDB 集群故障时恢复

> **注意：**
>
> TiDB Binlog 与 TiDB v5.0 版本开始引入的一些特性不兼容，无法一起使用，详情参照注意事项。
> 建议使用TiCDC 替代 TiDB Binlog。

要快速了解 Binlog 的基本原理和使用方法，建议先观看下面的培训视频（时长 32 分钟）。注意本视频只为学习参考，具体操作步骤和最新功能，请以文档内容为准。

13.10.1.1 TiDB Binlog 整体架构

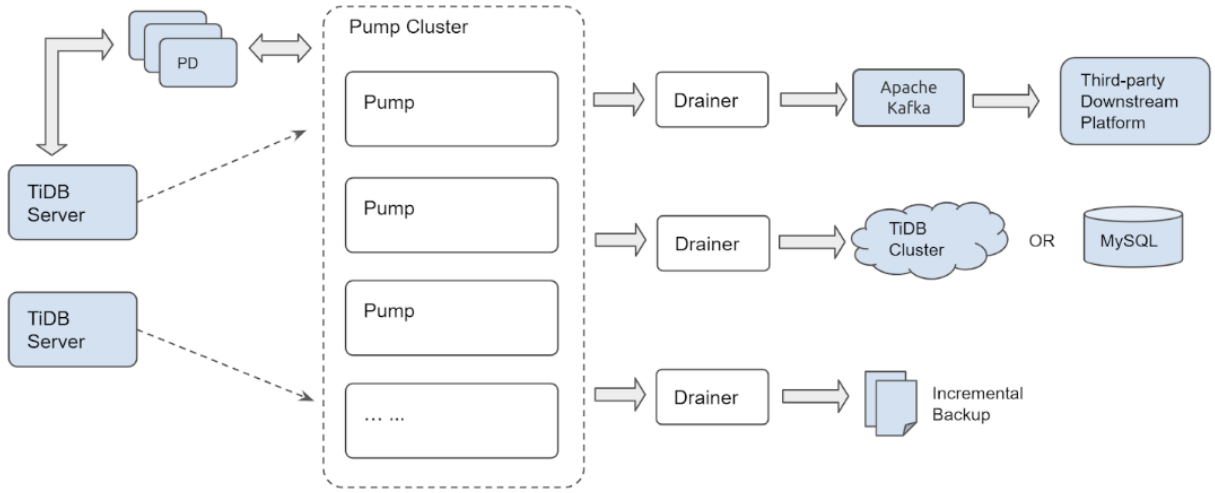


图 287: TiDB Binlog 架构

TiDB Binlog 集群主要分为 Pump 和 Drainer 两个组件，以及 binlogctl 工具：

13.10.1.1.1 Pump

Pump 用于实时记录 TiDB 产生的 Binlog，并将 Binlog 按照事务的提交时间进行排序，再提供给 Drainer 进行消费。

13.10.1.1.2 Drainer

Drainer 从各个 Pump 中收集 Binlog 进行归并，再将 Binlog 转化成 SQL 或者指定格式的数据，最终同步到下游。

13.10.1.1.3 binlogctl 工具

`binlogctl` 是一个 TiDB Binlog 配套的运维工具，具有如下功能：

- 获取 TiDB 集群当前的 TSO
- 查看 Pump/Drainer 状态
- 修改 Pump/Drainer 状态
- 暂停/下线 Pump/Drainer

13.10.1.2 主要特性

- 多个 Pump 形成一个集群，可以水平扩容。
- TiDB 通过内置的 Pump Client 将 Binlog 分发到各个 Pump。
- Pump 负责存储 Binlog，并将 Binlog 按顺序提供给 Drainer。
- Drainer 负责读取各个 Pump 的 Binlog，归并排序后发送到下游。
- Drainer 支持relay log 功能，通过 relay log 保证下游集群的一致性状态。

13.10.1.3 注意事项

- TiDB Binlog 和 TiDB 在 v5.1 版本中解决了 v5.0 版本中引入的聚簇索引与 TiDB Binlog 不兼容问题。在升级 TiDB Binlog 和 TiDB Server 到 v5.1 版本后：开启 TiDB Binlog 后，TiDB 支持创建聚簇索引表；聚簇索引表的数据插入、删除和更新操作支持通过 TiDB Binlog 同步到下游。对于同步聚簇索引表时需注意：
  - 如果从 v5.0 版本手动控制组件升级顺序进行升级，请确保先将 TiDB Binlog 升级至 v5.1 版本后再将 TiDB Server 升级至 v5.1 版本。
  - 推荐将上下游的 TiDB 系统变量`tidb_enable_clustered_index` 配置为一致的值来保证上下游 TiDB 聚簇索引表结构一致。
- TiDB Binlog 与 TiDB v5.0 版本开始引入的以下特性不兼容，无法一起使用：
  - TiDB 聚簇索引特性：开启 TiDB Binlog 后 TiDB 不允许创建非单个整数列作为主键的聚簇索引；已创建的聚簇索引表的数据插入、删除和更新动作不会通过 TiDB Binlog 同步到下游。如需同步聚簇索引表，请升级至 v5.1 版本或使用TiCDC；
  - TiDB 系统变量tidb_enable_async_commit：启用 TiDB Binlog 后，开启该选项无法获得性能提升。要获得性能提升，建议使用TiCDC 替代 TiDB Binlog。
  - TiDB 系统变量tidb_enable_1pc：启用 TiDB Binlog 后，开启该选项无法获得性能提升。要获得性能提升，建议使用TiCDC 替代 TiDB Binlog。
- Drainer 支持将 Binlog 同步到 MySQL、TiDB、Kafka 或者本地文件。如果需要将 Binlog 同步到其他 Drainer 不支持的类型的系统中，可以设置 Drainer 将 Binlog 同步到 Kafka，然后根据 binlog consumer protocol 进行定制处理，参考Binlog Consumer Client 用户文档。

- 如果 TiDB Binlog 用于增量恢复，可以设置配置项 db-type="file"，Drainer 会将 binlog 转化为指定的 proto buffer 格式的数据，再写入到本地文件中。这样就可以使用Reparo 恢复增量数据。

  关于 db-type 的取值，应注意：

  - 如果 TiDB 版本 < 2.1.9，则 db-type="pb"。
  - 如果 TiDB 版本 > = 2.1.9，则 db-type="file" 或 db-type="pb"。

- 如果下游为 MySQL/TiDB，数据同步后可以使用sync-diff-inspector 进行数据校验。

13.10.2 TiDB Binlog 教程

本文档主要介绍如何使用 TiDB Binlog 将数据从 TiDB 推送到 MariaDB 实例。文中的 TiDB Binlog 集群包含 Pump 和 Drainer 的单个节点，TiDB 集群包含 TiDB、TiKV 和 Placement Driver (PD) 各组件的单个节点。

希望上手实践 TiDB Binlog 工具的用户需要对TiDB 架构有一定的了解，最好有创建过 TiDB 集群的经验。该文档也有助于简单快速了解 TiDB Binlog 架构以及相关概念。

> **警告：**
>
> 该文档中部署 TiDB 的操作指导不适用于在生产或研发环境中部署 TiDB 的情况。

该文档假设用户使用的是现代 Linux 发行版本中的 x86-64。示例中使用的是 VMware 中运行的 CentOS 7 最小化安装。建议在一开始就进行清洁安装，以避免受现有环境中未知情况的影响。如果不想使用本地虚拟环境，也可以使用云服务启动 CentOS 7 VM。

13.10.2.1 TiDB Binlog 简介

TiDB Binlog 用于收集 TiDB 中二进制日志数据、提供实时数据备份和同步以及将 TiDB 集群的数据增量同步到下游。

TiDB Binlog 支持以下功能场景：

- 增量备份，将 TiDB 集群中的数据增量同步到另一个集群，或通过 Kafka 增量同步到选择的下游。
- 当使用 TiDB DM (Data Migration) 将数据从上游 MySQL 或者 MariaDB 迁移到 TiDB 集群时，可使用 TiDB Binlog 保持 TiDB 集群与其一个独立下游 MySQL 或 MariaDB 实例或集群同步。当 TiDB 集群上游数据迁移过程中出现问题，下游数据同步过程中可使用 TiDB Binlog 恢复数据到原先的状态。

更多信息参考TiDB Binlog Cluster 版本用户文档。

13.10.2.2 架构

TiDB Binlog 集群由 Pump 和 Drainer 两个组件组成。一个 Pump 集群中有若干个 Pump 节点。TiDB 实例连接到各个 Pump 节点并发送 binlog 数据到 Pump 节点。Pump 集群连接到 Drainer 节点，Drainer 将接收到的更新数据转换到某个特定下游（例如 Kafka、另一个 TiDB 集群或 MySQL 或 MariaDB Server）指定的正确格式。

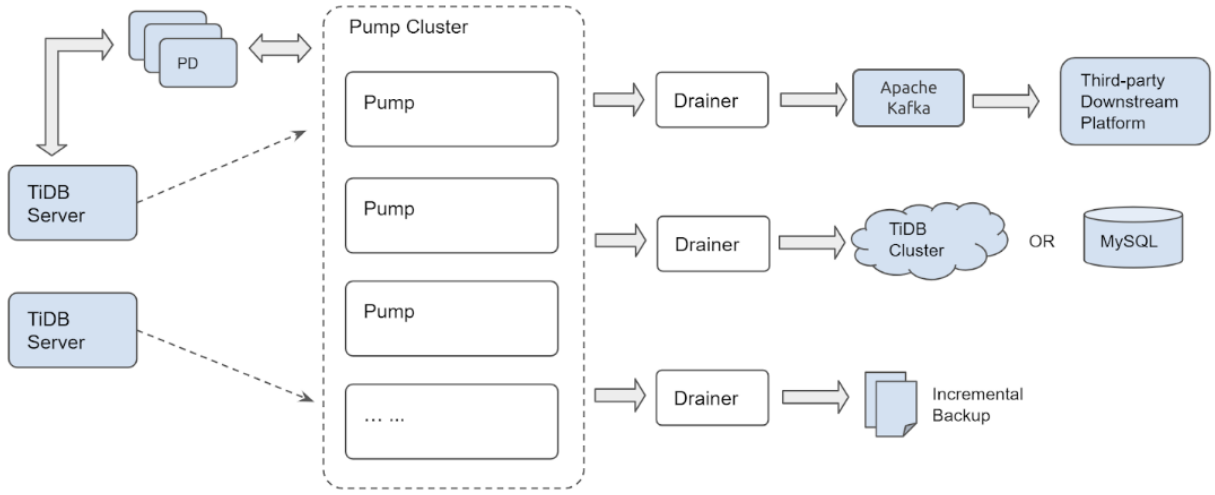


图 288: TiDB Binlog architecture

Pump 的集群架构能确保 TiDB 或 Pump 集群中有新的实例加入或退出时更新数据不会丢失。

13.10.2.3 安装

由于 RHEL/CentOS 7 的默认包装库中包括 MariaDB Server，本示例选择的是 MariaDB Server。后续除了安装服务器，也需要安装客户端。安装指令如下：

```
sudo yum install -y mariadb-server
```

```
curl -LO https://download.pingcap.org/tidb-community-server-v7.0.0-linux-amd64.tar.gz | tar xzf -
    ↪ &&
cd tidb-latest-linux-amd64
```

预期输出：

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  368M  100  368M    0     0  8394k      0  0:00:44  0:00:44 --:--:-- 11.1M
```

13.10.2.4 配置

通过执行以下步骤配置一个 TiDB 集群，该集群包括 pd-server、tikv-server 和 tidb-server 各组件的单个实例。

1. 填充配置文件：

```
printf > pd.toml %s\\n 'log-file="pd.log"' 'data-dir="pd.data"' &&
printf > tikv.toml %s\\n 'log-file="tikv.log"' '[storage]' 'data-dir="tikv.data"' '[pd]' '
    ↪ endpoints=["127.0.0.1:2379"]' '[rocksdb]' max-open-files=1024 '[raftdb]' max-open-
    ↪ files=1024 &&
```

```
printf > pump.toml %s\\n 'log-file="pump.log"' 'data-dir="pump.data"' 'addr="127.0.0.1:8250"
    ↪ ' 'advertise-addr="127.0.0.1:8250"' 'pd-urls="http://127.0.0.1:2379"' &&
printf > tidb.toml %s\\n 'store="tikv"' 'path="127.0.0.1:2379"' '[log.file]' 'filename="tidb
    ↪ .log"' '[binlog]' 'enable=true' &&
printf > drainer.toml %s\\n 'log-file="drainer.log"' '[syncer]' 'db-type="mysql"' '[syncer.
    ↪ to]' 'host="127.0.0.1"' 'user="root"' 'password=""' 'port=3306'
```

2. **查看配置细节：**

```
for f in *.toml; do echo "$f:"; cat "$f"; echo; done
```

预期输出：

```
drainer.toml:
log-file="drainer.log"
[syncer]
db-type="mysql"
[syncer.to]
host="127.0.0.1"
user="root"
password=""
port=3306

pd.toml:
log-file="pd.log"
data-dir="pd.data"

pump.toml:
log-file="pump.log"
data-dir="pump.data"
addr="127.0.0.1:8250"
advertise-addr="127.0.0.1:8250"
pd-urls="http://127.0.0.1:2379"

tidb.toml:
store="tikv"
path="127.0.0.1:2379"
[log.file]
filename="tidb.log"
[binlog]
enable=true

tikv.toml:
log-file="tikv.log"
[storage]
data-dir="tikv.data"
```

```
[pd]
endpoints=["127.0.0.1:2379"]
[rocksdb]
max-open-files=1024
[raftdb]
max-open-files=1024
```

13.10.2.5 启动程序

现在可启动各个组件。推荐启动顺序依次为 Placement Driver (PD)、 TiKV、 Pump（TiDB 发送 binlog 日志必须连接 Pump 服务）、 TiDB。

1. 启动所有服务：

```
./bin/pd-server --config=pd.toml &>pd.out &
```

```
[1] 20935
```

```
./bin/tikv-server --config=tikv.toml &>tikv.out &
```

```
[2] 20944
```

```
./pump --config=pump.toml &>pump.out &
```

```
[3] 21050
```

```
sleep 3 &&
./bin/tidb-server --config=tidb.toml &>tidb.out &
```

```
[4] 21058
```

2. 如果执行 `jobs`，可以看到后台正在运行的程序，列表如下：

```
jobs
```

```
[1]   Running                 ./bin/pd-server --config=pd.toml &>pd.out &
[2]   Running                 ./bin/tikv-server --config=tikv.toml &>tikv.out &
[3]-  Running                 ./pump --config=pump.toml &>pump.out &
[4]+  Running                 ./bin/tidb-server --config=tidb.toml &>tidb.out &
```

如果有服务启动失败（例如出现 “`Exit 1`” 而不是 “`Running`”），尝试重启单个组件。

13.10.2.6 连接

按以上步骤操作后，TiDB 的 4 个组件开始运行。接下来可以使用以下 MariaDB 或 MySQL 命令行客户端，通过 4000 端口连接到 TiDB 服务：

```
mysql -h 127.0.0.1 -P 4000 -u root -e 'select tidb_version();'
```

预期输出：

```
*************************** 1. row ***************************
tidb_version(): Release Version: v3.0.0-beta.1-154-gd5afff70c
Git Commit Hash: d5afff70cdd825d5fab125c8e52e686cc5fb9a6e
Git Branch: master
UTC Build Time: 2019-04-24 03:10:00
GoVersion: go version go1.12 linux/amd64
Race Enabled: false
TiKV Min Version: 2.1.0-alpha.1-ff3dd160846b7d1aed9079c389fc188f7f5ea13e
Check Table Before Drop: false
```

连接后 TiDB 集群已开始运行，pump 读取集群中的 binlog 数据，并在其数据目录中将 binlog 数据存储为 relay log。下一步是启动一个可供 `drainer` 写入的 MariaDB Server。

1. 启动 `drainer`：

    ```
    sudo systemctl start mariadb &&
    ./drainer --config=drainer.toml &>drainer.out &
    ```

    如果你的操作系统更易于安装 MySQL，只需保证监听 3306 端口。另外，可使用密码为空的“root”用户连接到 MySQL，或调整 `drainer.toml` 连接到 MySQL。

    ```
    mysql -h 127.0.0.1 -P 3306 -u root
    ```

    预期输出：

    ```
    Welcome to the MariaDB monitor.  Commands end with ; or \g.
    Your MariaDB connection id is 20
    Server version: 5.5.60-MariaDB MariaDB Server

    Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

    Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

    MariaDB [(none)]>
    ```

    ```
    show databases;
    ```

    预期输出：

```
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| test               |
| tidb_binlog        |
+--------------------+
5 rows in set (0.01 sec)
```

如下表格是包含 checkpoint 表格的 tidb_binlog 数据库。drainer 使用 checkpoint 表格，记录 TiDB 集群中的 binlog 已经更新到了哪个位置。

```
use tidb_binlog;
```

```
Database changed
```

```
select * from checkpoint;
```

```
+--------------------+-------------------------------------------+
| clusterID          | checkPoint                                |
+--------------------+-------------------------------------------+
| 6678715361817107733 | {"commitTS":407637466476445697,"ts-map":{}} |
+--------------------+-------------------------------------------+
1 row in set (0.00 sec)
```

打开另一个连接到 TiDB 的客户端，创建一个表格并插入几行数据。建议在 GNU Screen 软件中操作，从而同时打开多个客户端。

```
mysql -h 127.0.0.1 -P 4000 --prompt='TiDB [\d]> ' -u root
```

```
create database tidbtest;
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
use tidbtest;
```

```
Database changed
```

```
create table t1 (id int unsigned not null AUTO_INCREMENT primary key);
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
insert into t1 () values (),(),(),(),();
```

```
Query OK, 5 rows affected (0.01 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
select * from t1;
```

```
+----+
| id |
+----+
|  1 |
|  2 |
|  3 |
|  4 |
|  5 |
+----+
5 rows in set (0.00 sec)
```

切换回 MariaDB 客户端可看到新的数据库、新的表格和最近插入的行数据。

```
use tidbtest;
```

```
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
```

```
show tables;
```

```
+--------------------+
| Tables_in_tidbtest |
+--------------------+
| t1                 |
+--------------------+
1 row in set (0.00 sec)
```

```
select * from t1;
```

```
+----+
| id |
+----+
|  1 |
|  2 |
|  3 |
|  4 |
|  5 |
```

```
+-----+
5 rows in set (0.00 sec)
```

可看到查询 MariaDB 时插入到 TiDB 相同的行数据，表明 TiDB Binlog 安装成功。

13.10.2.7 binlogctl

加入到集群的 Pump 和 Drainer 的数据存储在 Placement Driver (PD) 中。binlogctl 可用于查询和修改状态信息。更多信息请参考binlogctl guide。

使用 `binlogctl` 查看集群中 Pump 和 Drainer 的当前状态：

```
./binlogctl -cmd drainers
```

```
[2019/04/11 17:44:10.861 -04:00] [INFO] [nodes.go:47] ["query node"] [type=drainer] [node="{
    ↪ NodeID: localhost.localdomain:8249, Addr: 192.168.236.128:8249, State: online,
    ↪ MaxCommitTS: 407638907719778305, UpdateTime: 2019-04-11 17:44:10 -0400 EDT}"]
```

```
./binlogctl -cmd pumps
```

```
[2019/04/11 17:44:13.904 -04:00] [INFO] [nodes.go:47] ["query node"] [type=pump] [node="{NodeID:
    ↪ localhost.localdomain:8250, Addr: 192.168.236.128:8250, State: online, MaxCommitTS:
    ↪ 407638914024079361, UpdateTime: 2019-04-11 17:44:13 -0400 EDT}"]
```

如果结束 Drainer 进程，集群会改进程设置 “已暂停（即集群等待 Drainer 重新加入）” 的状态。

```
pkill drainer &&
./binlogctl -cmd drainers
```

预期输出：

```
[2019/04/11 17:44:22.640 -04:00] [INFO] [nodes.go:47] ["query node"] [type=drainer] [node="{
    ↪ NodeID: localhost.localdomain:8249, Addr: 192.168.236.128:8249, State: paused,
    ↪ MaxCommitTS: 407638915597467649, UpdateTime: 2019-04-11 17:44:18 -0400 EDT}"]
```

使用 binlogctl 的 “NodeIDs” 可控制单个对应节点。在该情况下，Drainer 的节点 ID 是 “localhost.localdomain:8249”，Pump 的节点 ID 是 “localhost.localdomain:8250”。

本文档中的 binlogctl 主要用于集群重启。如果在 TiDB 集群中终止并尝试重启所有的进程，由于 Pump 无法连接 Drainer 且认为 Drainer 依旧 “在线”，Pump 会拒绝启动。这里的进程并不包括下游的 MySQL 或 MariaDB 或 Drainer。

以下有三个方案可解决上述问题：

- 使用 binlogctl 停止 Drainer，而不是结束进程：

```
./binlogctl --pd-urls=http://127.0.0.1:2379 --cmd=drainers &&
./binlogctl --pd-urls=http://127.0.0.1:2379 --cmd=pause-drainer --node-id=localhost.
    ↪ localdomain:8249
```

- 在启动 Pump 之前先启动 Drainer。
- 在启动 PD 之后但在启动 Drainer 和 Pump 之前，使用 binlogctl 更新已暂定 Drainer 的状态。

```
./binlogctl --pd-urls=http://127.0.0.1:2379 --cmd=update-drainer --node-id=localhost.
    ↪ localdomain:8249 --state=paused
```

13.10.2.8 清理

在 shell 终端里可启动创建集群的所有进程（pd-server、tikv-server、pump、tidb-server、drainer）。可通过在 shell 终端中执行 pkill -P $$ 停止 TiDB 集群服务和 TiDB Binlog 进程。按一定的顺序停止这些进程有利于留出足够的时间彻底关闭每个组件。

```
for p in tidb-server drainer pump tikv-server pd-server; do pkill "$p"; sleep 1; done
```

预期输出：

```
[4]-  Done                    ./bin/tidb-server --config=tidb.toml &>tidb.out
[5]+  Done                    ./drainer --config=drainer.toml &>drainer.out
[3]+  Done                    ./pump --config=pump.toml &>pump.out
[2]+  Done                    ./bin/tikv-server --config=tikv.toml &>tikv.out
[1]+  Done                    ./bin/pd-server --config=pd.toml &>pd.out
```

如果需要所有服务退出后重启集群，可以使用一开始启动服务的命令。如以上binlogctl 部分所述，需要先启动 Drainer 再启动 Pump，最后启动 tidb-server。

```
./bin/pd-server --config=pd.toml &>pd.out &
./bin/tikv-server --config=tikv.toml &>tikv.out &
./drainer --config=drainer.toml &>drainer.out &
sleep 3
./pump --config=pump.toml &>pump.out &
sleep 3
./bin/tidb-server --config=tidb.toml &>tidb.out &
```

如果有组件启动失败，请尝试单独重启该组件。

13.10.2.9 总结

本文档介绍了如何通过设置 TiDB Binlog，使用单个 Pump 和 Drainer 组成的集群同步 TiDB 集群数据到下游的 MariaDB。可以发现，TiDB Binlog 是用于获取处理 TiDB 集群中更新数据的综合性平台工具。

在更稳健的开发、测试或生产部署环境中，可以使用多个 TiDB 服务以实现高可用性和扩展性。使用多个 Pump 实例可以避免 Pump 集群中的问题影响发送到 TiDB 实例的应用流量。或者可以使用增加的 Drainer 实例同步数据到不同的下游或实现数据增量备份。

13.10.3 TiDB Binlog 集群部署

13.10.3.1 服务器要求

Pump 和 Drainer 均可部署和运行在 Intel x86-64 架构的 64 位通用硬件服务器平台上。在开发、测试和生产环境下，对服务器硬件配置的要求和建议如下：

服务	部署数量	CPU	磁盘	内存
Pump	3	8 核 +	SSD, 200 GB+	16G
Drainer	1	8 核 +	SAS, 100 GB+（如果输出 binlog 为本地文件，磁盘大小视保留数据天数而定）	16G

13.10.3.2 使用 TiUP 部署 TiDB Binlog

推荐使用 TiUP 部署 TiDB Binlog。即在使用 TiUP 部署 TiDB 时，在拓扑文件中添加 TiDB Binlog 的 drainer 和 pump 节点信息后，再随 TiDB 一起部署。详细部署方式参考TiUP 部署 TiDB 集群。

13.10.3.3 使用 Binary 部署 TiDB Binlog

13.10.3.3.1 下载 TiDB Binlog 安装包

TiDB Binlog 安装包位于 TiDB 离线工具包中。下载方式，请参考TiDB 工具下载。

13.10.3.3.2 使用样例

假设有三个 PD，一个 TiDB，另外有两台机器用于部署 Pump，一台机器用于部署 Drainer。各个节点信息如下：

```
TiDB="192.168.0.10"
PD1="192.168.0.16"
PD2="192.168.0.15"
PD3="192.168.0.14"
Pump="192.168.0.11"
Pump="192.168.0.12"
Drainer="192.168.0.13"
```

下面以此为例，说明 Pump/Drainer 的使用。

1. 使用 binary 部署 Pump

    - Pump 命令行参数说明（以在 “192.168.0.11” 上部署为例）

    ```
    Usage of Pump:
    -L string
        日志输出信息等级设置：debug，info，warn，error，fatal（默认 "info"）
    -V
        打印版本信息
    -addr string
        Pump 提供服务的 RPC 地址(-addr="192.168.0.11:8250")
    ```

```
-advertise-addr string
    Pump 对外提供服务的 RPC 地址(-advertise-addr="192.168.0.11:8250")
-config string
    配置文件路径，如果你指定了配置文件，Pump 会首先读取配置文件的配置;
    如果对应的配置在命令行参数里面也存在，Pump
        ↪ 就会使用命令行参数的配置来覆盖配置文件里的配置。
-data-dir string
    Pump 数据存储位置路径
-gc int
    Pump 只保留多少天以内的数据（默认 7）
-heartbeat-interval int
    Pump 向 PD 发送心跳间隔（单位 秒）
-log-file string
    log 文件路径
-log-rotate string
    log 文件切换频率，hour/day
-metrics-addr string
    Prometheus Pushgateway 地址，不设置则禁止上报监控信息
-metrics-interval int
    监控信息上报频率（默认 15，单位 秒）
-node-id string
    Pump 节点的唯一识别 ID，如果不指定，程序会根据主机名和监听端口自动生成
-pd-urls string
    PD 集群节点的地址 (-pd-urls="http://192.168.0.16:2379,http://192.168.0.15:2379,http
        ↪ ://192.168.0.14:2379")
-fake-binlog-interval int
    Pump 节点生成 fake binlog 的频率（默认 3，单位 秒）
```

- Pump 配置文件（以在 “192.168.0.11” 上部署为例）

```
# Pump Configuration

# Pump 绑定的地址
addr = "192.168.0.11:8250"

# Pump 对外提供服务的地址
advertise-addr = "192.168.0.11:8250"

# Pump 只保留多少天以内的数据（默认 7）
gc = 7

# Pump 数据存储位置路径
data-dir = "data.pump"

# Pump 向 PD 发送心跳的间隔（单位 秒）
heartbeat-interval = 2
```

```
# PD 集群节点的地址（英文逗号分割，中间不加空格）
pd-urls = "http://192.168.0.16:2379,http://192.168.0.15:2379,http://192.168.0.14:2379"

# [security]
# 如无特殊安全设置需要，该部分一般都注解掉
# 包含与集群连接的受信任 SSL CA 列表的文件路径
# ssl-ca = "/path/to/ca.pem"
# 包含与集群连接的 PEM 形式的 X509 certificate 的路径
# ssl-cert = "/path/to/drainer.pem"
# 包含与集群链接的 PEM 形式的 X509 key 的路径
# ssl-key = "/path/to/drainer-key.pem"

# [storage]
# 设置为 true（默认值）来保证可靠性，确保 binlog 数据刷新到磁盘
# sync-log = true

# 当可用磁盘容量小于该设置值时，Pump 将停止写入数据
# 42 MB -> 42000000, 42 mib -> 44040192
# default: 10 gib
# stop-write-at-available-space = "10 gib"

# Pump 内嵌的 LSM DB 设置，除非对该部分很了解，否则一般注解掉
# [storage.kv]
# block-cache-capacity = 8388608
# block-restart-interval = 16
# block-size = 4096
# compaction-L0-trigger = 8
# compaction-table-size = 67108864
# compaction-total-size = 536870912
# compaction-total-size-multiplier = 8.0
# write-buffer = 67108864
# write-L0-pause-trigger = 24
# write-L0-slowdown-trigger = 17
```

- 启动示例

```
./pump -config pump.toml
```

如果命令行参数与配置文件中的参数重合，则使用命令行设置的参数的值。

2. 使用 binary 部署 Drainer

- Drainer 命令行参数说明（以在 “192.168.0.13” 上部署为例）

```
Usage of Drainer
-L string
```

```
    日志输出信息等级设置：debug，info，warn，error，fatal（默认 "info"）
-V
    打印版本信息
-addr string
    Drainer 提供服务的地址(-addr="192.168.0.13:8249")
-c int
    同步下游的并发数，该值设置越高同步的吞吐性能越好（default 1）
-cache-binlog-count int
    缓存中的 binlog 数目限制（默认 8）
    如果上游的单个 binlog 较大导致 Drainer 出现 OOM 时，可尝试调小该值减少内存使用
-config string
    配置文件路径，Drainer 会首先读取配置文件的配置；
    如果对应的配置在命令行参数里面也存在，Drainer
        ↪ 就会使用命令行参数的配置来覆盖配置文件里面的配置
-data-dir string
    Drainer 数据存储位置路径（默认 "data.drainer"）
-dest-db-type string
    Drainer 下游服务类型（默认为 mysql，支持 tidb、kafka、file）
-detect-interval int
    向 PD 查询在线 Pump 的时间间隔（默认 10，单位 秒）
-disable-detect
    是否禁用冲突监测
-disable-dispatch
    是否禁用拆分单个 binlog 的 SQL 的功能，如果设置为 true，则每个 binlog
    按顺序依次还原成单个事务进行同步（下游服务类型为 MySQL，该项设置为 False）
-ignore-schemas string
    db 过滤列表（默认 "INFORMATION_SCHEMA,PERFORMANCE_SCHEMA,mysql,test"），
    不支持对 ignore schemas 的 table 进行 rename DDL 操作
-initial-commit-ts（默认为 `-1`）
    如果 Drainer 没有相关的断点信息，可以通过该项来设置相关的断点信息
    该参数值为 `-1` 时，Drainer 会自动从 PD 获取一个最新的时间戳
-log-file string
    log 文件路径
-log-rotate string
    log 文件切换频率，hour/day
-metrics-addr string
    Prometheus Pushgateway 地址，不设置则禁止上报监控信息
-metrics-interval int
    监控信息上报频率（默认 15，单位：秒）
-node-id string
    drainer 节点的唯一识别 ID，如果不指定，程序会根据主机名和监听端口自动生成
-pd-urls string
    PD 集群节点的地址 (-pd-urls="http://192.168.0.16:2379,http://192.168.0.15:2379,http
        ↪ ://192.168.0.14:2379")
-safe-mode
```

```
    是否开启安全模式使得下游 MySQL/TiDB 可被重复写入
    即将 insert 语句换为 replace 语句，将 update 语句拆分为 delete + replace 语句
-txn-batch int
    输出到下游数据库一个事务的 SQL 数量（默认 1）
```

- Drainer 配置文件（以在 “192.168.0.13” 上部署为例）

```
# Drainer Configuration.

# Drainer 提供服务的地址("192.168.0.13:8249")
addr = "192.168.0.13:8249"

# Drainer 对外提供服务的地址
advertise-addr = "192.168.0.13:8249"

# 向 PD 查询在线 Pump 的时间间隔（默认 10，单位 秒）
detect-interval = 10

# Drainer 数据存储位置路径（默认 "data.drainer"）
data-dir = "data.drainer"

# PD 集群节点的地址（英文逗号分割，中间不加空格）
pd-urls = "http://192.168.0.16:2379,http://192.168.0.15:2379,http://192.168.0.14:2379"

# log 文件路径
log-file = "drainer.log"

# Drainer 从 Pump 获取 binlog 时对数据进行压缩，值可以为 "gzip"，如果不配置则不进行压缩
# compressor = "gzip"

# [security]
# 如无特殊安全设置需要，该部分一般都注解掉
# 包含与集群连接的受信任 SSL CA 列表的文件路径
# ssl-ca = "/path/to/ca.pem"
# 包含与集群连接的 PEM 形式的 X509 certificate 的路径
# ssl-cert = "/path/to/pump.pem"
# 包含与集群链接的 PEM 形式的 X509 key 的路径
# ssl-key = "/path/to/pump-key.pem"

# Syncer Configuration
[syncer]
# 如果设置了该项，会使用该 sql-mode 解析 DDL 语句，此时如果下游是 MySQL 或 TiDB 则
# 下游的 sql-mode 也会被设置为该值
# sql-mode = "STRICT_TRANS_TABLES,NO_ENGINE_SUBSTITUTION"

# 输出到下游数据库一个事务的 SQL 语句数量（默认 20）
```

```
txn-batch = 20

# 同步下游的并发数，该值设置越高同步的吞吐性能越好（默认 16）
worker-count = 16

# 是否禁用拆分单个 binlog 的 SQL 的功能，如果设置为 true，则按照每个 binlog
# 顺序依次还原成单个事务进行同步（下游服务类型为 MySQL，该项设置为 False）
disable-dispatch = false

# safe mode 会使写下游 MySQL/TiDB 可被重复写入
# 会用 replace 替换 insert 语句，用 delete + replace 替换 update 语句
safe-mode = false

# Drainer 下游服务类型（默认为 mysql）
# 参数有效值为 "mysql", "tidb", "file", "kafka"
db-type = "mysql"

# 事务的 commit ts 若在该列表中，则该事务将被过滤，不会同步至下游
ignore-txn-commit-ts = []

# db 过滤列表（默认 "INFORMATION_SCHEMA,PERFORMANCE_SCHEMA,mysql,test"），
# 不支持对 ignore schemas 的 table 进行 rename DDL 操作
ignore-schemas = "INFORMATION_SCHEMA,PERFORMANCE_SCHEMA,mysql"

# replicate-do-db 配置的优先级高于 replicate-do-table。如果配置了相同的库名，
    ↪ 支持使用正则表达式进行配置。
# 以 '~' 开始声明使用正则表达式

# replicate-do-db = ["~^b.*","s1"]

# [syncer.relay]
# 保存 relay log 的目录，空值表示不开启。
# 只有下游是 TiDB 或 MySQL 时该配置才生效。
# log-dir = ""
# 每个文件的大小上限
# max-file-size = 10485760

# [[syncer.replicate-do-table]]
# db-name ="test"
# tbl-name = "log"

# [[syncer.replicate-do-table]]
# db-name ="test"
# tbl-name = "~^a.*"
```

```
# 忽略同步某些表
# [[syncer.ignore-table]]
# db-name = "test"
# tbl-name = "log"

# db-type 设置为 mysql 时，下游数据库服务器参数
[syncer.to]
host = "192.168.0.13"
user = "root"
# 如果你不想在配置文件中写明文密码，则可以使用 `./binlogctl -cmd encrypt -text string`
    ↪ 生成加密的密码
# 如果配置了 encrypted_password 且非空，那么配置的 password 不生效。encrypted_password
    ↪ 和 password 无法同时生效。
password = ""
encrypted_password = ""
port = 3306

[syncer.to.checkpoint]
# 当 checkpoint type 是 mysql 或 tidb 时可以开启该选项，以改变保存 checkpoint 的数据库
# schema = "tidb_binlog"
# 目前只支持 mysql 或者 tidb 类型。可以去掉注释来控制 checkpoint 保存的位置。
# db-type 默认的 checkpoint 保存方式是:
# mysql/tidb -> 对应的下游 mysql/tidb
# file/kafka -> file in `data-dir`
# type = "mysql"
# host = "127.0.0.1"
# user = "root"
# password = ""
# 使用 `./binlogctl -cmd encrypt -text string` 加密的密码
# encrypted_password 非空时 password 会被忽略
# encrypted_password = ""
# port = 3306

# db-type 设置为 file 时，存放 binlog 文件的目录
# [syncer.to]
# dir = "data.drainer"

# db-type 设置为 kafka 时，Kafka 相关配置
# [syncer.to]
# kafka-addrs 和 zookeeper-addrs 只需要一个，两者都有时程序会优先用 zookeeper 中的
    ↪ kafka 地址
# zookeeper-addrs = "127.0.0.1:2181"
# kafka-addrs = "127.0.0.1:9092"
# kafka-version = "0.8.2.0"
# 配置单条 broker request 中的最大 message 数（即 binlog 数），不配置或配置小于等于 0
```

```
    ↪ 时会使用默认值 1024
# kafka-max-messages = 1024
# 配置单条 broker request 的最大 size（单位为 Byte），默认为 1 GiB，最大可配置为 2 GiB
# kafka-max-message-size = 1073741824

# 保存 binlog 数据的 Kafka 集群的 topic 名称，默认值为 <cluster-id>_obinlog
# 如果运行多个 Drainer 同步数据到同一个 Kafka 集群，每个 Drainer 的 topic-name
    ↪ 需要设置不同的名称
# topic-name = ""
```

- 启动示例

> **注意：**
> 如果下游为 MySQL/TiDB，为了保证数据的完整性，在 Drainer 初次启动前需要获取 `initial-commit-ts` 的值，并进行全量数据的备份与恢复。

初次启动时使用参数 `initial-commit-ts`，命令如下：

```
./drainer -config drainer.toml -initial-commit-ts {initial-commit-ts}
```

如果命令行参数与配置文件中的参数重合，则使用命令行设置的参数的值。

注意：

- 在运行 TiDB 时，需要保证至少一个 Pump 正常运行。
- 通过给 TiDB 增加启动参数 `enable-binlog` 来开启 binlog 服务。尽量保证同一集群的所有 TiDB 都开启了 binlog 服务，否则在同步数据时可能会导致上下游数据不一致。如果要临时运行一个不开启 binlog 服务的 TiDB 实例，需要在 TiDB 的配置文件中设置 `run_ddl= false`。
- Drainer 不支持对 ignore schemas（在过滤列表中的 schemas）的 table 进行 rename DDL 操作。
- 在已有的 TiDB 集群中启动 Drainer，一般需要全量备份并且获取快照时间戳，然后导入全量备份，最后启动 Drainer 从对应的快照时间戳开始同步增量数据。
- 下游使用 MySQL 或 TiDB 时应当保证上下游数据库的 sql_mode 具有一致性，即下游数据库同步每条 SQL 语句时的 sql_mode 应当与上游数据库执行该条 SQL 语句时的 sql_mode 保持一致。可以在上下游分别执行 `select @@sql_mode;` 进行查询和比对。
- 如果存在上游 TiDB 能运行但下游 MySQL 不支持的 DDL 语句时（例如下游 MySQL 使用 InnoDB 引擎时同步语句 `CREATE TABLE t1(a INT)ROW_FORMAT=FIXED;`），Drainer 也会同步失败，此时可以在 Drainer 配置中跳过该事务，同时在下游手动执行兼容的语句，详见跳过事务。

13.10.4 TiDB Binlog 集群运维

本文首先介绍 Pump 和 Drainer 的状态及启动、退出的内部处理流程，然后说明如何通过 binlogctl 工具或者直接在 TiDB 执行 SQL 操作来管理 binlog 集群，最后的 FAQ 部分会介绍一些常见问题以及处理方法。

13.10.4.1 Pump/Drainer 的状态

Pump/Drainer 中状态的定义：

- online：正常运行中
- pausing：暂停中
- paused：已暂停
- closing：下线中
- offline：已下线

这些状态由 Pump/Drainer 服务自身进行维护，并定时将状态信息更新到 PD 中。

13.10.4.2 Pump/Drainer 的启动、退出流程

13.10.4.2.1 Pump

- 启动：Pump 启动时会通知所有 online 状态的 Drainer，如果通知成功，则 Pump 将状态设置为 online，否则 Pump 将报错，然后将状态设置为 paused 并退出进程。
- 退出：Pump 进程正常退出前要选择进入暂停或者下线状态；非正常退出（kill -9、进程 panic、宕机）都依然保持 online 状态。
  - 暂停：使用 kill（非 kill -9）、Ctrl+C 或者使用 binlogctl 的 pause-pump 命令都可以暂停 Pump。接收到暂停指令后，Pump 会变更状态为 pausing，并停止接受 binlog 的写请求，也停止向 Drainer 提供 binlog 数据。安全退出所有线程后，更新状态为 paused 然后退出进程。
  - 下线：仅在使用 binlogctl 的 offline-pump 命令的情况下才会下线 Pump。接收到下线指令后，Pump 会变更状态为 closing，并停止接受 binlog 的写请求。Pump 继续向 Drainer 提供 binlog，等待所有 binlog 数据都被 Drainer 消费后再将状态设置为 offline 并退出进程。

13.10.4.2.2 Drainer

- 启动：Drainer 启动时将状态设置为 online，并尝试从所有非 offline 状态的 Pump 获取 binlog，如果获取 binlog 失败，会不断尝试重新获取。
- 退出：Drainer 进程正常退出前要选择进入暂停或者下线状态；非正常退出（kill -9 、进程 panic、宕机）都依然保持 online 状态。
  - 暂停：使用 kill（非 kill -9）、Ctrl+C 或者使用 binlogctl 的 pause-drainer 命令都可以暂停 Drainer。接收到指令后，Drainer 会变更状态为 pausing，并停止从 Pump 获取 binlog。安全退出所有线程后，更新状态为 paused 然后退出进程。
  - 下线：仅在使用 binlogctl 的 offline-drainer 命令的情况下才会下线 Drainer。接收到下线指令后，Drainer 变更状态为 closing，并停止从 Pump 获取 binlog。安全退出所有线程后，更新状态为 offline 然后退出进程。

关于 Pump/Drainer 暂停、下线、状态查询、状态修改等具体的操作方法，参考如下 binlogctl 工具的使用方法介绍。

13.10.4.3 使用 binlogctl 工具管理 Pump/Drainer

binlogctl 支持如下这些功能：

- 查看 Pump/Drainer 状态
- 暂停/下线 Pump/Drainer
- Pump/Drainer 异常状态处理

详细的介绍和使用方法请参考binlogctl 工具。

13.10.4.4 使用 TiDB SQL 管理 Pump/Drainer

要查看和管理 binlog 相关的状态，可在 TiDB 中执行相应的 SQL 语句。

- 查看 TiDB 是否开启 binlog，0 代表关闭，1 代表开启

```
show variables like "log_bin";
```

```
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| log_bin       |  0    |
+---------------+-------+
```

- 查看 Pump/Drainer 状态

```
show pump status;
```

```
+--------|----------------|--------|--------------------|---------------------|
| NodeID |     Address    | State  |   Max_Commit_Ts    |    Update_Time      |
+--------|----------------|--------|--------------------|---------------------|
| pump1  | 127.0.0.1:8250 | Online | 408553768673342237 | 2019-05-01 00:00:01 |
+--------|----------------|--------|--------------------|---------------------|
| pump2  | 127.0.0.2:8250 | Online | 408553768673342335 | 2019-05-01 00:00:02 |
+--------|----------------|--------|--------------------|---------------------|
```

```
show drainer status;
```

```
+----------|----------------|--------|--------------------|---------------------|
|  NodeID  |     Address    | State  |   Max_Commit_Ts    |    Update_Time      |
+----------|----------------|--------|--------------------|---------------------|
| drainer1 | 127.0.0.3:8249 | Online | 408553768673342532 | 2019-05-01 00:00:03 |
+----------|----------------|--------|--------------------|---------------------|
| drainer2 | 127.0.0.4:8249 | Online | 408553768673345531 | 2019-05-01 00:00:04 |
+----------|----------------|--------|--------------------|---------------------|
```

- 异常情况下修改 Pump/Drainer 状态

```
change pump to node_state ='paused' for node_id 'pump1';
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
change drainer to node_state ='paused' for node_id 'drainer1';
```

```
Query OK, 0 rows affected (0.01 sec)
```

该 SQL 的功能和 binlogctl 中的 update-pump 和 update-drainer 命令的功能一样，因此也只有在 Pump/Drainer 异常的情况下使用。

注意：

1. 查看 binlog 开启状态以及 Pump/Drainer 状态的功能在 TiDB v2.1.7 及以上版本中支持。
2. 修改 Pump/Drainer 状态的功能在 TiDB v3.0.0-rc.1 及以上版本中支持。该功能只修改 PD 中存储的 Pump/Drainer 状态，如果需要暂停/下线节点，仍然需要使用 `binlogctl`。

13.10.5 TiDB Binlog 配置说明

本文档介绍 TiDB Binlog 的各项配置说明。

13.10.5.1 Pump

本节介绍 Pump 的配置项。可以在 Pump Configuration 中查看完整的 Pump 配置文件示例。

13.10.5.1.1 addr

- HTTP API 的监听地址，格式为 `host:port`。
- 默认：`"127.0.0.1:8250"`

13.10.5.1.2 advertise-addr

- 对外可访问的 HTTP API 地址。这个地址会被注册到 PD，格式为 `host:port`。
- 默认：与 `addr` 的配置相同。

13.10.5.1.3 socket

- HTTP API 监听的 Unix socket 地址。
- 默认：`""`

13.10.5.1.4 pd-urls

- 由逗号分隔的 PD URL 列表。如果指定了多个地址，PD 客户端在连接一个地址时出错时会自动尝试连接另一个地址。
- 默认：`"http://127.0.0.1:2379"`

13.10.5.1.5 data-dir

- 本地存放 binlog 及其索引的目录。
- 默认：`"data.pump"`

13.10.5.1.6 heartbeat-interval

- 心跳间隔，即每隔指定秒数向 PD 汇报最新的状态。
- 默认：2

13.10.5.1.7 gen-binlog-interval

- 指定写入 fake binlog 的间隔秒数。
- 默认：3

13.10.5.1.8 gc

- 指定 binlog 可在本地存储的天数（整型）。超过指定天数的 binlog 会被自动删除。
- 默认：7

13.10.5.1.9 log-file

- 保存日志文件的路径。如果为空，日志不会被保存。
- 默认：`""`

13.10.5.1.10 log-level

- Log 等级。
- 默认：`"info"`

13.10.5.1.11 node-id

- Pump 节点的 ID，用于在集群中识别这个进程。
- 默认：主机名:端口号，例如 `node-1:8250`。

13.10.5.1.12 security

以下是与安全相关的配置项。

ssl-ca

- 包含可信 SSL 证书或 CA 证书列表的文件路径，例如 `/path/to/ca.pem`。
- 默认：`""`

ssl-cert

- 包含 PEM 格式编码的 X509 证书文件路径，例如 `/path/to/pump.pem`。
- 默认：`""`

ssl-key

- 包含 PEM 格式编码的 X509 Key 文件路径，例如 `/path/to/pump-key.pem`。
- 默认：`""`

13.10.5.1.13 storage

以下是与存储相关的配置项。

sync-log

- 指定是否在每次批量写入 binlog 后使用 `fsync` 以确保数据安全。
- 默认：`true`

kv_chan_cap

- 在 Pump 接收写入请求时会先将请求放入一个缓冲区，该项指定缓冲区能存放的请求数量。
- 默认：1048576（即 2 的 20 次方）

slow_write_threshold

- 写入单个 binlog 的耗时超过该项设定的秒数就被认为是慢写入，并输出一条包含 `"take a long time to write binlog"` 的日志。
- 默认：1

stop-write-at-available-space

- 可用存储空间低于指定值时不再接收 binlog 写入请求。可以用例如 `900 MB`、`5 GB`、`12 GiB` 的格式指定空间大小。如果集群中 Pump 节点多于一个，那么在某个 Pump 节点因为空间不足而拒绝写入时，TiDB 端会自动写入到其他 Pump 节点。
- 默认：`10 GiB`

kv

目前 Pump 的存储是基于 GoLevelDB 实现的。`storage` 下还有一个 kv 子分组可以用于调整 GoLevelDB 的配置，支持的配置项包括：

- block-cache-capacity
- block-restart-interval
- block-size
- compaction-L0-trigger
- compaction-table-size
- compaction-total-size
- compaction-total-size-multiplier
- write-buffer
- write-L0-pause-trigger
- write-L0-slowdown-trigger

配置具体含义可在 GoLevelDB 文档中查看。

13.10.5.2 Drainer

本节介绍 Drainer 的配置项。可以在 Drainer Configuration 中查看完整的配置文件示例。

13.10.5.2.1 addr

- HTTP API 的监听的地址，格式为 `host:port`。
- 默认：`"127.0.0.1:8249"`

13.10.5.2.2 advertise-addr

- 对外可访问的 HTTP API 地址，这个地址会被注册到 PD，格式为 `host:port`。
- 默认：设定成与 `addr` 相同的配置

13.10.5.2.3 log-file

- 日志的文件保存路径。如果为空，日志不会被保存。
- 默认：`""`

13.10.5.2.4 log-level

- Log 等级。
- 默认：`"info"`

13.10.5.2.5 node-id

- Drainer 节点 ID，用于在集群中识别这个进程。
- 默认：主机名:端口号，例如 `node-1:8249`。

13.10.5.2.6 data-dir

- 用于存放 Drainer 运行中需要保存的文件的目录。
- 默认：`"data.drainer"`

13.10.5.2.7 detect-interval

- 每隔指定秒数从 PD 更新一次 Pumps 信息，以获取节点加入或离开等事件。
- 默认：5

13.10.5.2.8 pd-urls

- 由逗号分隔的 PD URL 列表。如果指定了多个地址，PD 客户端在连接一个地址出错时会自动尝试连接另一个地址。
- 默认：`"http://127.0.0.1:2379"`

13.10.5.2.9 initial-commit-ts

- 指定从哪个事务提交时间点（事务的 commit ts）之后开始同步。这个配置仅适用于初次开始同步的 Drainer 节点。如果下游已经有 checkpoint 存在，则会根据 checkpoint 里记录的时间进行同步。
- commit ts（即 commit timestamp）是 TiDB 事务的提交时间点。该时间点是从 PD 获取的全局唯一递增的时间戳，作为当前事务的唯一 ID。典型的 `initial-commit-ts` 配置可以通过以下方式获得：
  - BR 备份的元信息（即 backupmeta）中记录的 backup TS
  - Dumpling 备份的元信息（即 metadata）中记录的 Pos
  - PD Control 中 `tso` 命令返回的结果
- 默认：-1。Drainer 会从 PD 得到一个最新的 timestamp 作为初始时间。即从当前的时间点开始同步。

13.10.5.2.10 synced-check-time

- 通过 HTTP API 访问 `/status` 路径可以查询 Drainer 同步的状态。`synced-check-time` 指定距离上次成功同步的时间超过多少分钟可以认为是 `synced`，即同步完成。
- 默认：5

13.10.5.2.11 compressor

- 指定 Pump 与 Drainer 间的数据传输所用的压缩算法。目前仅支持一种算法，即 `gzip`。
- 默认：`""`，表示不压缩。

13.10.5.2.12 security

以下是与 Drainer 安全相关的配置。

ssl-ca

- 包含可信 SSL 证书或 CA 证书列表的文件路径，例如 `/path/to/ca.pem`。
- 默认：""

ssl-cert

- 包含 PEM 格式编码的 X509 证书文件路径，例如 `/path/to/drainer.pem`。
- 默认：""

ssl-key

- 包含 PEM 格式编码的 X509 Key 文件路径，例如 `/path/to/drainer-key.pem`。
- 默认：""

13.10.5.2.13 syncer

syncer 分组包含一些与同步下游相关的配置项。

db-type

下游类型，目前支持的选项有：

- `mysql`
- `tidb`
- `kafka`
- `file`

默认：`mysql`

sql-mode

- 当下游为 `mysql/tidb` 类型时，可以指定 SQL mode，如果超过一个 mode，则用逗号隔开。
- 默认：""

ignore-txn-commit-ts

- 同步时，该项所指定的 commit timestamp 的 binlog 会被忽略，例如 `[416815754209656834, 421349811963822081]`。
- 默认：`[]`

ignore-schemas

- 同步时忽略指定数据库的变更。如果超过一个需忽略的数据库，则用逗号隔开。如果一个 binlog 中的变更全部被过滤掉，则忽略整个 binlog。
- 默认：`"INFORMATION_SCHEMA,PERFORMANCE_SCHEMA,mysql"`

ignore-table

同步时忽略指定表格的变更。在 `toml` 中可以用以下方式指定多个需忽略的表格：

```
[[syncer.ignore-table]]
db-name = "test"
tbl-name = "log"

[[syncer.ignore-table]]
db-name = "test"
tbl-name = "audit"
```

如果一个 binlog 中的变更全部被过滤掉，则忽略整个 binlog。

默认：`[]`

replicate-do-db

- 指定要同步的数据库，例如 `[db1, db2]`。
- 默认：`[]`

replicate-do-table

指定要同步的表格，示例如下：

```
[[syncer.replicate-do-table]]
db-name ="test"
tbl-name = "log"

[[syncer.replicate-do-table]]
db-name ="test"
tbl-name = "~^a.*"
```

默认：`[]`

txn-batch

- 当下游为 `mysql/tidb` 类型时，会将 DML 分批执行。这个配置可以用于设置每个事务中包含多少个 DML。
- 默认：`20`

worker-count

- 当下游为 `mysql/tidb` 类型时，会并发执行 DML，`worker-count` 可以指定并发数。
- 默认：`16`

disable-dispatch

- 关掉并发，强制将 worker-count 置为 1。
- 默认：false

safe-mode

如果打开 Safe mode，Drainer 会对同步的变更作以下修改，使其变成可重入的操作：

- Insert 变为 Replace Into
- Update 变为 Delete 和 Replace Into

默认：false

13.10.5.2.14 syncer.to

不同类型的下游配置都在 syncer.to 分组。以下按配置类型进行介绍。

mysql/tidb

用于连接下游数据库的配置项：

- host：如果没有设置，会尝试检查环境变量 MYSQL_HOST，默认值为 "localhost"。
- port：如果没有设置，会尝试检查环境变量 MYSQL_PORT，默认值为 3306。
- user：如果没有设置，会尝试检查环境变量 MYSQL_USER，默认值为 "root"。
- password：如果没有设置，会尝试检查环境变量 MYSQL_PSWD，默认值为 ""。
- read-timeout：指定下游数据库连接的 IO 读取超时时间，默认值为 1m。如果 drainer 在一些耗时长的 DDL 上不断失败，你可以将这个变量设置为更大的值。

file

- dir：指定用于保存 binlog 的目录。如果不指定该项，则使用 data-dir。

kafka

当下游为 kafka 时，有效的配置包括：

- zookeeper-addrs
- kafka-addrs
- kafka-version
- kafka-max-messages
- kafka-max-message-size
- topic-name

13.10.5.2.15 syncer.to.checkpoint

- type：指定用哪种方式保存同步进度，目前支持的选项为 mysql、tidb 和 file。

  该配置选项默认与下游类型相同。例如 file 类型的下游 checkpoint 进度保存在本地文件 <data-dir>/savepoint 中，mysql 类型的下游进度保存在下游数据库。当明确指定要使用 mysql 或 tidb 保存同步进度时，需要指定以下配置项：

- schema：默认为 "tidb_binlog"。

> 注意：
> 在同个 TiDB 集群中部署多个 Drainer 时，需要为每个 Drainer 节点指定不同的 checkpoint schema，否则两个实例的同步进度会互相覆盖。

- host
- user
- password
- port

13.10.5.3 TiDB Binlog 配置说明

本文档介绍 TiDB Binlog 的各项配置说明。

13.10.5.3.1 Pump

本节介绍 Pump 的配置项。可以在 Pump Configuration 中查看完整的 Pump 配置文件示例。

addr

- HTTP API 的监听地址，格式为 host:port。
- 默认："127.0.0.1:8250"

advertise-addr

- 对外可访问的 HTTP API 地址。这个地址会被注册到 PD，格式为 host:port。
- 默认：与 addr 的配置相同。

socket

- HTTP API 监听的 Unix socket 地址。
- 默认：""

pd-urls

- 由逗号分隔的 PD URL 列表。如果指定了多个地址，PD 客户端在连接一个地址时出错时会自动尝试连接另一个地址。
- 默认：`"http://127.0.0.1:2379"`

data-dir

- 本地存放 binlog 及其索引的目录。
- 默认：`"data.pump"`

heartbeat-interval

- 心跳间隔，即每隔指定秒数向 PD 汇报最新的状态。
- 默认：2

gen-binlog-interval

- 指定写入 fake binlog 的间隔秒数。
- 默认：3

gc

- 指定 binlog 可在本地存储的天数（整型）。超过指定天数的 binlog 会被自动删除。
- 默认：7

log-file

- 保存日志文件的路径。如果为空，日志不会被保存。
- 默认：`""`

log-level

- Log 等级。
- 默认：`"info"`

node-id

- Pump 节点的 ID，用于在集群中识别这个进程。
- 默认：主机名:端口号，例如 node-1:8250。

security

以下是与安全相关的配置项。

ssl-ca

- 包含可信 SSL 证书或 CA 证书列表的文件路径，例如 `/path/to/ca.pem`。

- 默认：""

ssl-cert

- 包含 PEM 格式编码的 X509 证书文件路径，例如 `/path/to/pump.pem`。
- 默认：""

ssl-key

- 包含 PEM 格式编码的 X509 Key 文件路径，例如 `/path/to/pump-key.pem`。
- 默认：""

storage

以下是与存储相关的配置项。

sync-log

- 指定是否在每次批量写入 binlog 后使用 `fsync` 以确保数据安全。
- 默认：`true`

kv_chan_cap

- 在 Pump 接收写入请求时会先将请求放入一个缓冲区，该项指定缓冲区能存放的请求数量。
- 默认：1048576（即 2 的 20 次方）

slow_write_threshold

- 写入单个 binlog 的耗时超过该项设定的秒数就被认为是慢写入，并输出一条包含 `"take a long time to write binlog"` 的日志。
- 默认：1

stop-write-at-available-space

- 可用存储空间低于指定值时不再接收 binlog 写入请求。可以用例如 `900 MB`、`5 GB`、`12 GiB` 的格式指定空间大小。如果集群中 Pump 节点多于一个，那么在某个 Pump 节点因为空间不足而拒绝写入时，TiDB 端会自动写入到其他 Pump 节点。
- 默认：`10 GiB`

kv

目前 Pump 的存储是基于 GoLevelDB 实现的。`storage` 下还有一个 kv 子分组可以用于调整 GoLevelDB 的配置，支持的配置项包括：

- block-cache-capacity
- block-restart-interval

- block-size
- compaction-L0-trigger
- compaction-table-size
- compaction-total-size
- compaction-total-size-multiplier
- write-buffer
- write-L0-pause-trigger
- write-L0-slowdown-trigger

配置具体含义可在 GoLevelDB 文档中查看。

13.10.5.3.2 Drainer

本节介绍 Drainer 的配置项。可以在 Drainer Configuration 中查看完整的配置文件示例。

addr

- HTTP API 的监听的地址，格式为 `host:port`。
- 默认：`"127.0.0.1:8249"`

advertise-addr

- 对外可访问的 HTTP API 地址，这个地址会被注册到 PD，格式为 `host:port`。
- 默认：设定成与 `addr` 相同的配置

log-file

- 日志的文件保存路径。如果为空，日志不会被保存。
- 默认：`""`

log-level

- Log 等级。
- 默认：`"info"`

node-id

- Drainer 节点 ID，用于在集群中识别这个进程。
- 默认：主机名:端口号，例如 `node-1:8249`。

data-dir

- 用于存放 Drainer 运行中需要保存的文件的目录。
- 默认：`"data.drainer"`

detect-interval

- 每隔指定秒数从 PD 更新一次 Pumps 信息，以获取节点加入或离开等事件。
- 默认：5

pd-urls

- 由逗号分隔的 PD URL 列表。如果指定了多个地址，PD 客户端在连接一个地址出错时会自动尝试连接另一个地址。
- 默认："http://127.0.0.1:2379"

initial-commit-ts

- 指定从哪个事务提交时间点（事务的 commit ts）之后开始同步。这个配置仅适用于初次开始同步的 Drainer 节点。如果下游已经有 checkpoint 存在，则会根据 checkpoint 里记录的时间进行同步。
- commit ts（即 commit timestamp）是 TiDB 事务的提交时间点。该时间点是从 PD 获取的全局唯一递增的时间戳，作为当前事务的唯一 ID。典型的 initial-commit-ts 配置可以通过以下方式获得：
  - BR 备份的元信息（即 backupmeta）中记录的 backup TS
  - Dumpling 备份的元信息（即 metadata）中记录的 Pos
  - PD Control 中 tso 命令返回的结果
- 默认：-1。Drainer 会从 PD 得到一个最新的 timestamp 作为初始时间。即从当前的时间点开始同步。

synced-check-time

- 通过 HTTP API 访问 /status 路径可以查询 Drainer 同步的状态。synced-check-time 指定距离上次成功同步的时间超过多少分钟可以认为是 synced，即同步完成。
- 默认：5

compressor

- 指定 Pump 与 Drainer 间的数据传输所用的压缩算法。目前仅支持一种算法，即 gzip。
- 默认：""，表示不压缩。

security

以下是与 Drainer 安全相关的配置。

ssl-ca

- 包含可信 SSL 证书或 CA 证书列表的文件路径，例如 /path/to/ca.pem。
- 默认：""

ssl-cert

- 包含 PEM 格式编码的 X509 证书文件路径，例如 /path/to/drainer.pem。
- 默认：""

ssl-key

- 包含 PEM 格式编码的 X509 Key 文件路径，例如 `/path/to/drainer-key.pem`。
- 默认：""

syncer

syncer 分组包含一些与同步下游相关的配置项。

db-type

下游类型，目前支持的选项有：

- `mysql`
- `tidb`
- `kafka`
- `file`

默认：`mysql`

sql-mode

- 当下游为 `mysql/tidb` 类型时，可以指定 SQL mode，如果超过一个 mode，则用逗号隔开。
- 默认：""

ignore-txn-commit-ts

- 同步时，该项所指定的 commit timestamp 的 binlog 会被忽略，例如 `[416815754209656834, 421349811963822081]` 。
- 默认：`[]`

ignore-schemas

- 同步时忽略指定数据库的变更。如果超过一个需忽略的数据库，则用逗号隔开。如果一个 binlog 中的变更全部被过滤掉，则忽略整个 binlog。
- 默认：`"INFORMATION_SCHEMA,PERFORMANCE_SCHEMA,mysql"`

ignore-table

同步时忽略指定表格的变更。在 `toml` 中可以用以下方式指定多个需忽略的表格：

```
[[syncer.ignore-table]]
db-name = "test"
tbl-name = "log"

[[syncer.ignore-table]]
db-name = "test"
tbl-name = "audit"
```

如果一个 binlog 中的变更全部被过滤掉，则忽略整个 binlog。

默认：[]

replicate-do-db

- 指定要同步的数据库，例如 [db1, db2]。
- 默认：[]

replicate-do-table

指定要同步的表格，示例如下：

```
[[syncer.replicate-do-table]]
db-name ="test"
tbl-name = "log"

[[syncer.replicate-do-table]]
db-name ="test"
tbl-name = "~^a.*"
```

默认：[]

txn-batch

- 当下游为 mysql/tidb 类型时，会将 DML 分批执行。这个配置可以用于设置每个事务中包含多少个 DML。
- 默认：20

worker-count

- 当下游为 mysql/tidb 类型时，会并发执行 DML，worker-count 可以指定并发数。
- 默认：16

disable-dispatch

- 关掉并发，强制将 worker-count 置为 1。
- 默认：false

safe-mode

如果打开 Safe mode，Drainer 会对同步的变更作以下修改，使其变成可重入的操作：

- Insert 变为 Replace Into
- Update 变为 Delete 和 Replace Into

默认：false

syncer.to

不同类型的下游配置都在 syncer.to 分组。以下按配置类型进行介绍。

mysql/tidb

用于连接下游数据库的配置项：

- host：如果没有设置，会尝试检查环境变量 MYSQL_HOST，默认值为 "localhost"。
- port：如果没有设置，会尝试检查环境变量 MYSQL_PORT，默认值为 3306。
- user：如果没有设置，会尝试检查环境变量 MYSQL_USER，默认值为 "root"。
- password：如果没有设置，会尝试检查环境变量 MYSQL_PSWD，默认值为 ""。
- read-timeout：指定下游数据库连接的 IO 读取超时时间，默认值为 1m。如果 drainer 在一些耗时长的 DDL 上不断失败，你可以将这个变量设置为更大的值。

file

- dir：指定用于保存 binlog 的目录。如果不指定该项，则使用 data-dir。

kafka

当下游为 kafka 时，有效的配置包括：

- zookeeper-addrs
- kafka-addrs
- kafka-version
- kafka-max-messages
- kafka-max-message-size
- topic-name

syncer.to.checkpoint

- type：指定用哪种方式保存同步进度，目前支持的选项为 mysql、 tidb 和 file。

  该配置选项默认与下游类型相同。例如 file 类型的下游 checkpoint 进度保存在本地文件 <data-dir>/savepoint 中，mysql 类型的下游进度保存在下游数据库。当明确指定要使用 mysql 或 tidb 保存同步进度时，需要指定以下配置项：
- schema：默认为 "tidb_binlog"。

> 注意：
> 在同个 TiDB 集群中部署多个 Drainer 时，需要为每个 Drainer 节点指定不同的 checkpoint schema，否则两个实例的同步进度会互相覆盖。

- host
- user
- password
- port

13.10.6 TiDB Binlog 版本升级方法

如未特别指明，文中出现的 TiDB Binlog 均指最新的Cluster 版本。

本文介绍通过手动部署的 TiDB Binlog 的版本升级方法，另外有一小节介绍如何从更早的不兼容版本（Kafka/Local 版本）升级到最新版本。

13.10.6.1 手动部署

13.10.6.1.1 升级 Pump

对集群里的每个 Pump 逐一升级，确保集群中总有 Pump 可以接收 TiDB 发来的 Binlog。

1. 用新版本的 pump 替换原来的文件
2. 重启 Pump 进程

13.10.6.1.2 升级 Drainer

1. 用新版本的 `drainer` 替换原来的文件
2. 重启 Drainer 进程

13.10.6.2 从 Kafka/Local 版本升级到 Cluster 版本

新版本的 TiDB（v2.0.8-binlog、v2.1.0-rc.5 及以上版本）不兼容 Kafka 版本以及 Local 版本的 TiDB Binlog，集群升级到新版本后只能使用 Cluster 版本的 TiDB Binlog。如果在升级前已经使用了 Kafka / Local 版本的 TiDB Binlog，必须将其升级到 Cluster 版本。

TiDB Binlog 版本与 TiDB 版本的对应关系如下：

TiDB Binlog 版本	TiDB 版本	说明
Local	TiDB 1.0 及更低版本	
Kafka	TiDB 1.0 ~ TiDB 2.1 RC5	TiDB 1.0 支持 local 版本和 Kafka 版本的 TiDB Bin-log。

TiDB Binlog 版本	TiDB 版本	说明
Cluster	TiDB v2.0.8-binlog，TiDB 2.1 RC5 及更高版本	TiDB v2.0.8-binlog 是一个支持 Cluster 版本 TiDB Binlog 的 2.0 特殊版本。

13.10.6.2.1 升级流程

> **注意：**
>
> 如果能接受重新导全量数据，则可以直接废弃老版本，按TiDB Binlog 集群部署中的步骤重新部署。

如果想从原来的 checkpoint 继续同步，使用以下升级流程：

1. 部署新版本 Pump。
2. 暂停 TiDB 集群业务。
3. 更新 TiDB 以及配置，写 Binlog 到新的 Pump Cluster。
4. TiDB 集群重新接入业务。
5. 确认老版本的 Drainer 已经将老版本的 Pump 的数据完全同步到下游。

   查询 Drainer 的 `status` 接口，示例命令如下：

```
curl 'http://172.16.10.49:8249/status'
```

```
{"PumpPos":{"172.16.10.49:8250":{"offset":32686}},"Synced": true ,"DepositWindow":{"Upper
    ↪ ":398907800202772481,"Lower":398907799455662081}}
```

如果返回的 Synced 为 true，则可以认为 Binlog 数据已经全部同步到了下游。

6. 启动新版本 Drainer；
7. 下线老版本的 Pump、 Drainer 以及依赖的 Kafka 和 ZooKeeper。

13.10.7 TiDB Binlog 集群监控

成功部署 TiDB Binlog 集群后，可以进入 Grafana Web 界面（默认地址: http://grafana_ip:3000，默认账号：admin，密码：admin）查看 Pump 和 Drainer 的运行状态。

13.10.7.1 监控指标

13.10.7.1.1 Pump

metric 名称	说明
Storage Size	记录磁盘的总空间大小 (capacity)，以及可用磁盘空间大小 (available)
Metadata	记录每个 Pump 的可删除 binlog 的最大 tso (gc_tso)，以及保存的 binlog 的最大的 commit tso (max_commit_tso)。
Write Binlog QPS by Instance	每个 Pump 接收到的写 binlog 请求的 QPS
Write Binlog Latency	记录每个 Pump 写 binlog 的延迟时间
Storage Write Binlog Size	Pump 写 binlog 数据的大小
Storage Write Binlog Latency	Pump 中的 storage 模块写 binlog 数据的延迟

metric 名称	说明
Pump Storage Error By Type	Pump 遇到的 error 数量，按照 error 的类型进行统计
Query TiKV	Pump 通过 TiKV 查询事务状态的次数

13.10.7.1.2 Drainer

metric 名称	说明
Checkpoint TSO	Drainer 已经同步到下游的 binlog 的最大 TSO 对应的时间。可以通过该指标估算同步延迟时间
Pump Handle TSO	记录 Drainer 从各个 Pump 获取到的 binlog 的最大 TSO 对应的时间
Pull Binlog QPS by Pump NodeID	Drainer 从每个 Pump 获取 binlog 的 QPS
95% Binlog Reach Duration By Pump	记录 binlog 从写入 Pump 到被 Drainer 获取到这个过程的延迟时间
Error By Type	Drainer 遇到的 error 数量，按照 error 的类型进行统计
SQL Query Time	Drainer 在下游执行 SQL 的耗时
Drainer Event	各种类型 event 的数量，event 包括 ddl、insert、delete、update、flush、savepoint
Execute Time	写入 binlog 到同步下游模块所消耗的时间
95% Binlog Size	Drainer 从各个 Pump 获取到 binlog 数据的大小
DDL Job Count	Drainer 处理的 DDL 的数量
Queue Size	Drainer 内部工作队列大小

13.10.7.2 监控报警规则

本节介绍了 TiDB Binlog 组件的报警项及相应的报警规则。根据严重级别，报警项可分为三类，按照严重程度由高到低依次为：紧急级别 (Emergency)、重要级别 (Critical)、警告级别 (Warning)。

13.10.7.2.1 紧急级别报警项

紧急级别的报警通常由于服务停止或节点故障导致，此时需要马上进行人工干预操作。

`binlog_pump_storage_error_count`

- 报警规则：

  ```
  changes(binlog_pump_storage_error_count[1m])> 0
  ```

- 规则描述：

  Pump 写 binlog 到本地存储时失败。

- 处理方法：

  确认 `pump_storage_error` 监控是否存在错误，查看 Pump 日志确认原因。

13.10.7.2.2 重要级别报警项

对于重要级别的报警，需要密切关注异常指标。

`binlog_drainer_checkpoint_high_delay`

- 报警规则：

  ```
  (time()- binlog_drainer_checkpoint_tso / 1000)> 3600
  ```

- 规则描述：

  Drainer 同步落后延迟超过 1 个小时。

- 处理方法：

  - 判断从 Pump 获取数据是否太慢：

    监控 Pump handle tso 可以看每个 Pump 最近一条消息的时间，是不是有延迟特别大的 Pump，确认对应 Pump 正常运行。

  - 根据 Drainer event 和 Drainer execute latency 来判断是否下游同步太慢：

    * 如果 Drainer execute time 过大，则检查到目标库网络带宽和延迟，以及目标库状态。
    * 如果 Drainer execute time 不大，Drainer event 过小，则增加 work count 和 batch 进行重试。

  - 如果上面都不满足或者操作后没有改观，请从 PingCAP 官方或 TiDB 社区获取支持。

13.10.7.2.3 警告级别报警项

警告级别的报警是对某一问题或错误的提醒。

`binlog_pump_write_binlog_rpc_duration_seconds_bucket`

- 报警规则：

  ```
  histogram_quantile(0.9, rate(binlog_pump_rpc_duration_seconds_bucket{method="WriteBinlog"}[5
  ↪ m]))> 1
  ```

- 规则描述：

  Pump 处理 TiDB 写 Binlog 请求耗时过大。

- 处理方法：

  - 确认磁盘性能压力，通过 `node exported` 查看 disk performance 监控。
  - 如果 `disk latency` 和 `util` 都很低，请从 PingCAP 官方或 TiDB 社区获取支持。

`binlog_pump_storage_write_binlog_duration_time_bucket`

- 报警规则：

  ```
  histogram_quantile(0.9, rate(binlog_pump_storage_write_binlog_duration_time_bucket{type="
  ↪ batch"}[5m]))> 1
  ```

- 规则描述：

  Pump 写本地 binlog 到本地盘的耗时。

- 处理方法：

  确认 Pump 本地盘情况，进行修复。

`binlog_pump_storage_available_size_less_than_20G`

- 报警规则：

  `binlog_pump_storage_storage_size_bytes{type="available"} < 20 * 1024 * 1024 * 1024`

- 规则描述：

  Pump 剩余可用磁盘空间不足 20 G。

- 处理方法：

  监控确认 Pump 的 `gc_tso` 是否正常。如果不正常，调整 Pump 的 GC 时间配置或者下线对应 Pump。

`binlog_drainer_checkpoint_tso_no_change_for_1m`

- 报警规则：

  `changes(binlog_drainer_checkpoint_tso[1m])< 1`

- 规则描述：

  Drainer 的 `checkpoint` 在 1 分钟内没有更新。

- 处理方法：

  确认所有非下线的 Pump 是否正常运行。

`binlog_drainer_execute_duration_time_more_than_10s`

- 报警规则：

  `histogram_quantile(0.9, rate(binlog_drainer_execute_duration_time_bucket[1m]))> 10`

- 规则描述：

  Drainer 同步到 TiDB 的事务耗时。如果这个值过大，会影响 Drainer 同步。

- 处理方法：

  - 查看 TiDB 集群的状态。
  - 查看 Drainer 日志或监控，如果是 DDL 操作导致了该问题，则忽略。

13.10.8 Reparo 使用文档

Reparo 是 TiDB Binlog 的一个配套工具，用于增量的恢复。使用 TiDB Binlog 中的 Drainer 将 binlog 按照 protobuf 格式输出到文件，通过这种方式来备份增量数据。当需要恢复增量数据时，使用 Reparo 解析文件中的 binlog，并将其应用到 TiDB / MySQL 中。

Reparo 的安装包 `reparo` 位于 TiDB 离线工具包中。下载方式，请参考TiDB 工具下载。

13.10.8.1 Reparo 使用

13.10.8.1.1 命令行参数说明

```
Usage of Reparo:
-L string
    日志输出信息等级设置：debug, info, warn, error, fatal（默认值：info）。
-V 打印版本信息。
-c int
    同步下游的并发数，该值设置越高同步的吞吐性能越好（默认 16）。
-config string
    配置文件路径，如果指定了配置文件，Reparo 会首先读取配置文件的配置；
        ↪ 如果对应的配置在命令行参数里面也存在，Reparo
        ↪ 就会使用命令行参数的配置来覆盖配置文件里面的。
-data-dir string
    Drainer 输出的 protobuf 格式 binlog 文件的存储路径（默认值： data.drainer）。
-dest-type string
    下游服务类型。 取值为 print, mysql（默认值：print）。当值为 print 时，只做解析打印到标准输出
        ↪ ，不执行 SQL；如果为 mysql，则需要在配置文件内配置 host、port、user、password 等信息
        ↪ 。
-log-file string
    log 文件路径。
-log-rotate string
    log 文件切换频率，取值为 hour、day。
-start-datetime string
    用于指定开始恢复的时间点，格式为 “2006-01-02 15:04:05”。如果不设置该参数则从最早的 binlog
        ↪ 文件开始恢复。
-stop-datetime string
    用于指定结束恢复的时间点，格式同上。如果不设置该参数则恢复到最后一个 binlog 文件。
-safe-mode bool
    指定是否开启安全模式，开启后可支持反复同步。
-txn-batch int
    输出到下游数据库一个事务的 SQL 语句数量（默认 20）。
```

13.10.8.1.2 配置文件说明

```
### Drainer 输出的 protobuf 格式 binlog 文件的存储路径。
data-dir = "./data.drainer"

### 日志输出信息等级设置：debug, info, warn, error, fatal（默认值：info）。
log-level = "info"

### 使用 start-datetime 和 stop-datetime 来选择恢复指定时间范围内的 binlog，格式为 “2006-01-02
    ↪ 15:04:05”。
### start-datetime = ""
```

```
### stop-datetime = ""

### start-tso、stop-tso 分别对应 start-datetime 和 stop-datetime，也是用于恢复指定时间范围内的
    ↪ binlog，用 tso 的值来设置。如果已经设置了 start-datetime 和 stop-datetime，就不需要再设置
    ↪  start-tso 和 stop-tso。
### 在从全量或者上次增量位置继续同步时，start-tso 应当指定为全量 tso + 1 或者上次增量的 stop-tso
    ↪ + 1
### start-tso = 0
### stop-tso = 0

### 下游服务类型。 取值为 print, mysql（默认值：print）。当值为 print 时，只做解析打印到标准输出
    ↪ ，不执行 SQL；如果为 mysql，则需要在 [dest-db] 中配置 host、port、user、password 等信息。
dest-type = "mysql"

### 输出到下游数据库一个事务的 SQL 语句数量（默认 20）。
txn-batch = 20

### 同步下游的并发数，该值设置越高同步的吞吐性能越好（默认 16）。
worker-count = 16

### 安全模式配置。取值为 true 或 false（默认值：false）。当值为 true 时，Reparo 会将 update
    ↪ 语句拆分为 delete + replace 语句。
safe-mode = false

### replicate-do-db 和 replicate-do-table 用于指定恢复的库和表，replicate-do-db 的优先级高于
    ↪ replicate-do-table。支持使用正则表达式来配置，需要以 '~' 开始声明使用正则表达式。
### 注：replicate-do-db 和 replicate-do-table 使用方式与 Drainer 的使用方式一致。
### replicate-do-db = ["~^b.*","s1"]
### [[replicate-do-table]]
### db-name ="test"
### tbl-name = "log"
### [[replicate-do-table]]
### db-name ="test"
### tbl-name = "~^a.*"

### 如果 dest-type 设置为 mysql，需要配置 dest-db。
[dest-db]
host = "127.0.0.1"
port = 3309
user = "root"
password = ""
```

13.10.8.1.3 启动示例

```
./reparo -config reparo.toml
```

注意：

- data-dir 用于指定 Drainer 输出的 binlog 文件目录。
- start-datatime 和 start-tso 效果一样，只是时间格式上的区别，用于指定开始恢复的时间点；如果不指定，则默认在第一个 binlog 文件开始恢复。
- stop-datetime 和 stop-tso 效果一样，只是时间格式上的区别，用于指定结束恢复的时间点；如果不指定，则恢复到最后一个 binlog 文件的结尾。
- dest-type 指定目标类型，取值为 `mysql`、`print`。当值为 `mysql` 时，可以恢复到 MySQL/TiDB 等使用或兼容 MySQL 协议的数据库，需要在配置下面的 [dest-db] 填写数据库信息；当取值为 `print` 的时候，只是打印 binlog 信息，通常用于 debug，以及查看 binlog 的内容，此时不需要填写 `[dest-db]`。
- replicate-do-db 用于指定恢复的库，不指定的话，则全部都恢复。
- replicate-do-table 用于指定要恢复的表，不指定的话，则全部都恢复。

13.10.9 binlogctl 工具

Binlog Control（以下简称 binlogctl）是 TiDB Binlog 的命令行工具，用于管理 TiDB Binlog 集群。

binlogctl 支持如下这些功能：

- 查看 Pump/Drainer 状态
- 暂停/下线 Pump/Drainer
- Pump/Drainer 异常状态处理

使用 binlogctl 的场景：

- 同步出现故障/检查运行情况，需要查看 Pump/Drainer 的状态
- 维护集群，需要暂停/下线 Pump/Drainer
- Pump/Drainer 异常退出，状态没有更新，或者状态不符合预期，对业务造成影响

13.10.9.1 binlogctl 下载

注意：

建议使用的 Control 工具版本与集群版本保持一致。

binlogctl 的安装包位于 TiDB 离线工具包中。下载方式，请参考TiDB 工具下载。

13.10.9.2 binlogctl 使用说明

命令行参数：

```
Usage of binlogctl:
-V
输出 binlogctl 的版本信息
-cmd string
    命令模式，包括 "generate_meta"（已废弃），"pumps", "drainers", "update-pump" ,"update-drainer
        ↪ ", "pause-pump", "pause-drainer", "offline-pump", "offline-drainer"
-data-dir string
    保存 Drainer 的 checkpoint 的文件的路径（默认 "binlog_position"）（已废弃）
-node-id string
    Pump/Drainer 的 ID
-pd-urls string
    PD 的地址，如果有多个，则用"," 连接（默认 "http://127.0.0.1:2379"）
-ssl-ca string
    SSL CAs 文件的路径
-ssl-cert string
        PEM 格式的 X509 认证文件的路径
-ssl-key string
        PEM 格式的 X509 key 文件的路径
-time-zone string
    如果设置时区，在 "generate_meta" 模式下会打印出获取到的 tso 对应的时间。例如 "Asia/Shanghai"
        ↪ 为 CST 时区，"Local" 为本地时区
-show-offline-nodes
    在用 `-cmd pumps` 或 `-cmd drainers` 命令时使用，这两个命令默认不显示 offline 的节点，
        ↪ 仅当明确指定 `-show-offline-nodes` 时会显示
```

命令示例：

- 查询所有的 Pump/Drainer 的状态：

  设置 cmd 为 pumps 或者 drainers 来查看所有 Pump 或者 Drainer 的状态。例如：

```
bin/binlogctl -pd-urls=http://127.0.0.1:2379 -cmd pumps
```

```
[2019/04/28 09:29:59.016 +00:00] [INFO] [nodes.go:48] ["query node"] [type=pump] [node="{
    ↪ NodeID: 1.1.1.1:8250, Addr: pump:8250, State: online, MaxCommitTS:
    ↪ 408012403141509121, UpdateTime: 2019-04-28 09:29:57 +0000 UTC}"]
```

```
bin/binlogctl -pd-urls=http://127.0.0.1:2379 -cmd drainers
```

```
[2019/04/28 09:29:59.016 +00:00] [INFO] [nodes.go:48] ["query node"] [type=drainer] [node="{
    ↪ NodeID: 1.1.1.1:8249, Addr: 1.1.1.1:8249, State: online, MaxCommitTS:
    ↪ 408012403141509121, UpdateTime: 2019-04-28 09:29:57 +0000 UTC}"]
```

- 暂停/下线 Pump/Drainer

  binlogctl 提供以下命令暂停/下线服务：

cmd	说明	示例
pause-pump	暂停 Pump	`bin/binlogctl -pd-urls=http://127.0.0.1:2379 -cmd pause-pump -node-id ip-127-0-0-1:8250`
pause-drainer	暂停 Drainer	`bin/binlogctl -pd-urls=http://127.0.0.1:2379 -cmd pause-drainer -node-id ip-127-0-0-1:8249`
offline-pump	下线 Pump	`bin/binlogctl -pd-urls=http://127.0.0.1:2379 -cmd offline-pump -node-id ip-127-0-0-1:8250`
offline-drainer	下线 Drainer	`bin/binlogctl -pd-urls=http://127.0.0.1:2379 -cmd offline-drainer -node-id ip-127-0-0-1:8249`

  binlogctl 会发送 HTTP 请求给 Pump/Drainer，Pump/Drainer 收到命令后会主动执行对应的退出流程。

- 异常情况下修改 Pump/Drainer 的状态

  在服务正常运行以及符合流程的暂停、下线过程中，Pump/Drainer 的状态都是可以正确的。但是在一些异常情况下 Pump/Drainer 无法正确维护自己的状态，可能会影响数据同步任务，在这种情况下需要使用 binlogctl 修复状态信息。

  设置 cmd 为 update-pump 或者 update-drainer 来更新 Pump 或者 Drainer 的状态。Pump 和 Drainer 的状态可以为 paused 或者 offline。例如：

```
bin/binlogctl -pd-urls=http://127.0.0.1:2379 -cmd update-pump -node-id ip-127-0-0-1:8250 -state paused
```

> 注意：
>
> Pump/Drainer 在正常运行过程中会定期在 PD 中更新自己的状态，而这条命令是直接去修改 Pump/Drainer 保存在 PD 中的状态，所以在 Pump/Drainer 服务正常的情况下使用这些命令是没有意义的。仅在 Pump/Drainer 服务异常的情况下使用，具体哪些场景下使用这条命令可以参考 FAQ。

13.10.10 Binlog Consumer Client 用户文档

目前 Drainer 提供了多种输出方式，包括 MySQL、TiDB、file 等。但是用户往往有一些自定义的需求，比如输出到 Elasticsearch、Hive 等，这些需求 Drainer 现在还没有实现，因此 Drainer 增加了输出到 Kafka 的功能，将 binlog 数据解析后按一定的格式再输出到 Kafka 中，用户编写代码从 Kafka 中读出数据再进行处理。

13.10.10.1 配置 Kafka Drainer

修改 Drainer 的配置文件，设置输出为 Kafka，相关配置如下：

```
[syncer]
db-type = "kafka"
```

```
[syncer.to]
### Kafka 地址
kafka-addrs = "127.0.0.1:9092"
### Kafka 版本号
kafka-version = "2.4.0"
```

13.10.10.2 自定义开发

13.10.10.2.1 数据格式

首先需要了解 Drainer 写入到 Kafka 中的数据格式：

```
// Column 保存列的数据，针对数据的类型，保存在对应的变量中
message Column {
  // 数据是否为 null
  optional bool is_null = 1 [ default = false ];
  // 保存 int 类型的数据
  optional int64 int64_value = 2;
  // 保存 uint、enum, set 类型的数据
  optional uint64 uint64_value = 3;
  // 保存 float、double 类型的数据
  optional double double_value = 4;
  // 保存 bit、blob、binary、json 类型的数据
  optional bytes bytes_value = 5;
  // 保存 date、time、decimal、text、char 类型的数据
  optional string string_value = 6;
}

// ColumnInfo 保存列的信息，包括列名、类型、是否为主键
message ColumnInfo {
  optional string name = 1 [ (gogoproto.nullable) = false ];
  // MySQL 中小写的列字段类型
  // https://dev.mysql.com/doc/refman/8.0/en/data-types.html
  // numeric 类型：int bigint smallint tinyint float double decimal bit
  // string 类型：text longtext mediumtext char tinytext varchar
  // blob longblob mediumblob binary tinyblob varbinary
  // enum set
  // json 类型：json
  optional string mysql_type = 2 [ (gogoproto.nullable) = false ];
  optional bool is_primary_key = 3 [ (gogoproto.nullable) = false ];
}

// Row 保存一行的具体数据
message Row { repeated Column columns = 1; }
```

```
// MutationType 表示 DML 的类型
enum MutationType {
  Insert = 0;
  Update = 1;
  Delete = 2;
}

// Table 包含一个表的数据变更
message Table {
  optional string schema_name = 1;
  optional string table_name = 2;
  repeated ColumnInfo column_info = 3;
  repeated TableMutation mutations = 4;
}

// TableMutation 保存一行数据的变更
message TableMutation {
  required MutationType type = 1;
  // 修改后的数据
  required Row row = 2;
  // 修改前的数据，只对 Update MutationType 有效
  optional Row change_row = 3;
}

// DMLData 保存一个事务所有的 DML 造成的数据变更
message DMLData {
  // `tables` 包含事务中所有表的数据变更
  repeated Table tables = 1;
}

// DDLData 保存 DDL 的信息
message DDLData {
  // 当前使用的数据库
  optional string schema_name = 1;
  // 相关表
  optional string table_name = 2;
  // `ddl_query` 是原始的 DDL 语句 query
  optional bytes ddl_query = 3;
}

// BinlogType 为 Binlog 的类型，分为 DML 和 DDL
enum BinlogType {
  DML = 0; //  Has `dml_data`
  DDL = 1; //  Has `ddl_query`
```

```
}

// Binlog 保存一个事务所有的变更，Kafka 中保存的数据为该结构数据序列化后的结果
message Binlog {
  optional BinlogType type = 1 [ (gogoproto.nullable) = false ];
  optional int64 commit_ts = 2 [ (gogoproto.nullable) = false ];
  optional DMLData dml_data = 3;
  optional DDLData ddl_data = 4;
}
```

查看数据格式的具体定义，参见 secondary_binlog.proto。

13.10.10.2.2 Driver

TiDB-Tools 项目提供了用于读取 Kafka 中 binlog 数据的 Driver，具有如下功能：

- 读取 Kafka 的数据
- 根据 commit ts 查找 binlog 在 kafka 中的储存位置

使用该 Driver 时，用户需要配置如下信息：

- KafkaAddr：Kafka 集群的地址
- CommitTS：从哪个 commit ts 开始读取 binlog
- Offset：从 Kafka 哪个 offset 开始读取，如果设置了 CommitTS 就不用配置该参数
- ClusterID：TiDB 集群的 cluster ID
- Topic: Kafka Topic 名称，如果 Topic 名称为空，将会使用 drainer <ClusterID>_obinlog 中的默认名称

用户以包的形式引用 Driver 的代码即可使用，可以参考 Driver 中提供的示例代码来学习如何使用 Driver 以及 binlog 数据的解析，目前提供了两个例子：

- 使用该 Driver 将数据同步到 MySQL，该示例包含将 binlog 转化为 SQL 的具体方法
- 使用该 Driver 将数据打印出来

Driver 项目地址：Binlog Slave Driver。

> 注意：
>
> - 示例代码仅仅用于示范如何使用 Driver，如果需要用于生产环境需要优化代码。
> - 目前仅提供了 golang 版本的 Driver 以及示例代码。如果需要使用其他语言，用户需要根据 binlog 的 proto 文件生成相应语言的代码文件，并自行开发程序读取 Kafka 中的 binlog 数据、解析数据、输出到下游。也欢迎用户优化 example 代码，以及提交其他语言的示例代码到 TiDB-Tools。

13.10.11 TiDB Binlog Relay Log

Drainer 同步 binlog 时会拆分上游的事务，并将拆分的事务并发同步到下游。在极端情况下，上游集群不可用并且 Drainer 异常退出后，下游集群（MySQL 或 TiDB）可能处于数据不一致的中间状态。在此场景下，Drainer 借助 relay log 可以确保将下游集群同步到一个一致的状态。

13.10.11.1 Drainer 同步时的一致性状态

下游集群达到一致的状态是指：下游集群的数据等同于上游设置了 `tidb_snapshot = ts` 的快照。

checkpoint 状态一致性是指：Drainer checkpoint 通过 `consistent` 保存了同步的一致性状态。Drainer 运行时 `consistent` 为 `false`，Drainer 正常退出后 `consistent` 更新为 `true`。

查询下游 checkpoint 表的示例如下：

```
mysql> select  * from tidb_binlog.checkpoint;
+--------------------+-----------------------------------------------------------------+
| clusterID          | checkPoint                                                      |
+--------------------+-----------------------------------------------------------------+
| 6791641053252586769 | {"consistent":false,"commitTS":414529105591271429,"ts-map":{}} |
+--------------------+-----------------------------------------------------------------+
```

13.10.11.2 工作原理

Drainer 开启 relay log 后会先将 binlog event 写到磁盘上，然后再同步给下游集群。如果上游集群不可用，Drainer 可以通过读取 relay log 把下游集群恢复到一个一致的状态。

> **注意：**
>
> 若同时丢失 relay log 数据，该方法将不可用，不过这是概率极小的事件。此外可以使用 NFS 等网络文件系统来保证 relay log 的数据安全。

13.10.11.2.1 Drainer 从 relay log 消费 binlog 的触发场景

如果 Drainer 启动时无法连接到上游集群的 PD，并且探测到 checkpoint 的 `consistent = false`，此时会尝试读取 relay log，并将下游集群恢复到一致的状态。然后 Drainer 进程将 checkpoint 的 `consistent` 设置为 `true` 后主动退出。

13.10.11.2.2 Relay log 的清理（GC）机制

Drainer 在将数据同步到下游之前，会先将数据写入到 relay log 文件中。当一个 relay log 文件大小达到 10MB（默认）并且当前事务的 binlog 数据被写入完成后，Drainer 就会开始将数据写入到下一个 relay log 文件中。当 Drainer 将数据成功同步到下游后，就会自动清除当前正在写入的 relay log 文件以外其他已完成同步的 relay log 文件。

13.10.11.3 配置

在 Drainer 中添加以下配置来开启 relay log 功能：

```
[syncer.relay]
### 保存 relay log 的目录，空值表示不开启。
### 只有下游是 TiDB 或 MySQL 时该配置才有生效。
log-dir = "/dir/to/save/log"
### 单个 relay log 文件大小限制（单位：字节）。
### 超出该值后会将 binlog 数据写入到下一个 relay log 文件。
max-file-size = 10485760
```

13.10.12 集群间双向同步

> **警告：**
>
> 目前双向同步属于实验特性，尚未经过完备的测试，不建议在生产环境中使用该功能。

本文档介绍如何将一个 TiDB 集群的数据双向同步到另一个 TiDB 集群、双向同步的实现原理、如何开启双向同步、以及如何同步 DDL 操作。

13.10.12.1 使用场景

当用户需要在两个 TiDB 集群之间双向同步数据时，可使用 TiDB Binlog 进行操作。例如要将集群 A 的数据同步到集群 B，而且要将集群 B 的数据同步到集群 A。

> **注意：**
>
> 集群间双向同步的前提条件是，写入两个集群的数据必须保证无冲突，即在两个集群中，不会同时修改同一张表的同一主键和具有唯一索引的行。

使用场景示例图如下：

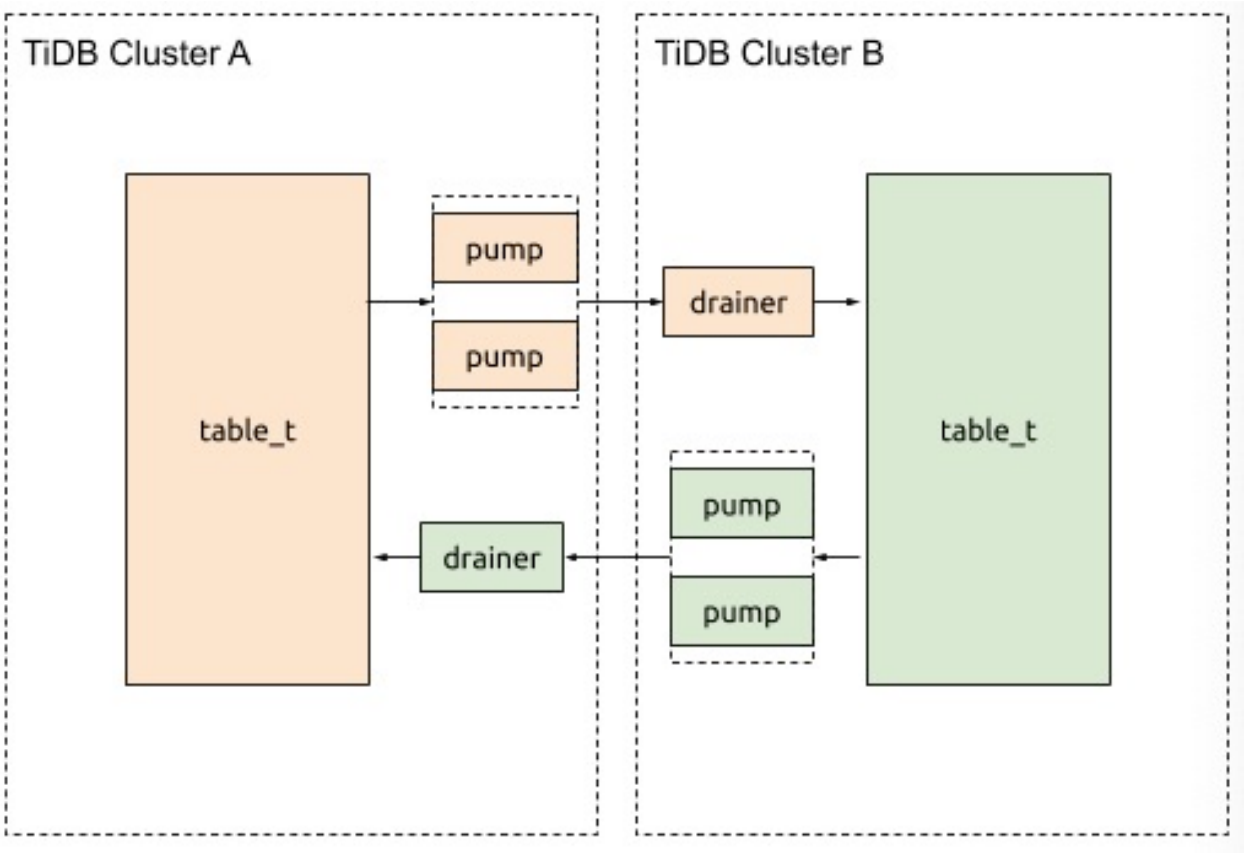


图 289: 使用场景示例图

13.10.12.2 实现原理

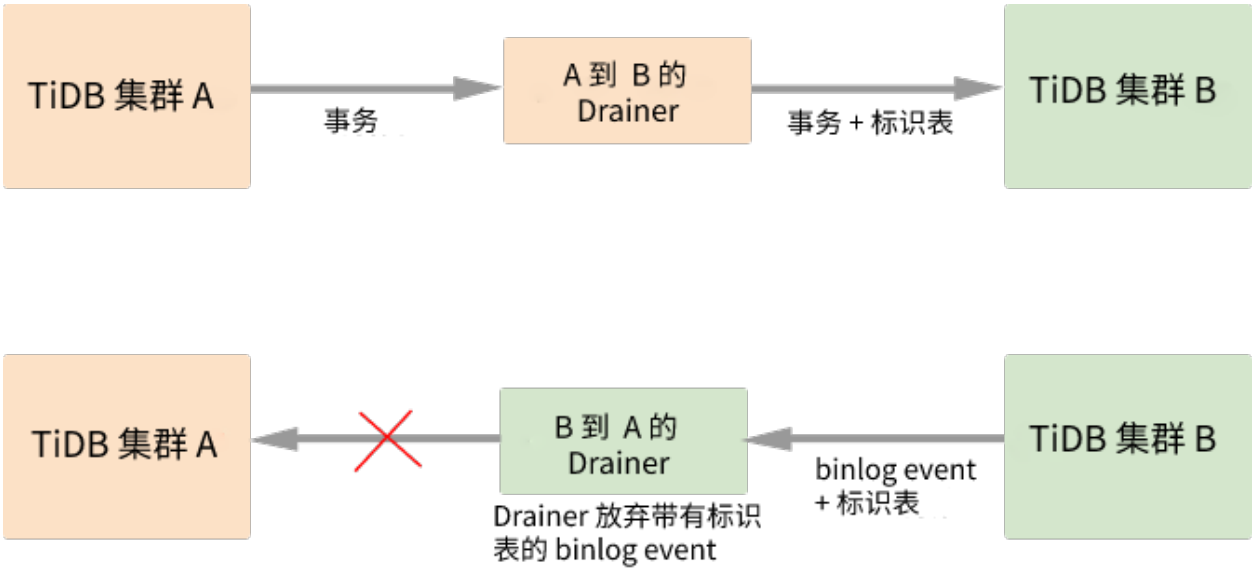


图 290: 原理示例图

在 A 和 B 两个集群间开启双向同步，则写入集群 A 的数据会同步到集群 B 中，然后这部分数据又会继续同步到集群 A，这样就会出现无限循环同步的情况。如上图所示，在同步数据的过程中 Drainer 对 binlog 加上标记，通过过滤掉有标记的 binlog 来避免循环同步。详细的实现流程如下：

1. 为两个集群分别启动 TiDB Binlog 同步程序。
2. 待同步的事务经过 A 的 Drainer 时，Drainer 为事务加入 `_drainer_repl_mark` 标识表，并在表中写入本次 DML event 更新，将事务同步至集群 B。
3. 集群 B 向集群 A 返回带有 `_drainer_repl_mark` 标识表的 binlog event。集群 B 的 Drainer 在解析该 binlog event 时发现带有 DML event 的标识表，放弃同步该 binlog event 到集群 A。

将集群 B 中的数据同步到集群 A 的流程与以上流程相同，两个集群可以互为上下游。

> 注意：
>
> - 更新 `_drainer_repl_mark` 标识表时，一定要有数据改动才会产生 binlog。
> - DDL 操作没有事务概念，因此采取单向同步的方案，见同步 DDL。

Drainer 与下游的每个连接可以使用一个 ID 以避免冲突。`channel_id` 用来表示进行双向同步的一个通道。A 和 B 两个集群进行双向同步的配置应使用相同的值。

当有添加或者删除列时，要同步到下游的数据可能会出现多列或者少列的情况。Drainer 通过添加配置来允许这种情况，会忽略多了的列值或者给少了的列写入默认值。

13.10.12.3 标识表

`_drainer_repl_mark` 标识表的结构如下：

```
CREATE TABLE `_drainer_repl_mark` (
  `id` bigint(20) NOT NULL,
  `channel_id` bigint(20) NOT NULL DEFAULT '0',
  `val` bigint(20) DEFAULT '0',
  `channel_info` varchar(64) DEFAULT NULL,
  PRIMARY KEY (`id`,`channel_id`)
);
```

Drainer 使用如下 SQL 语句更新 _drainer_repl_mark 可保证数据改动，从而保证产生 binlog：

```
update drainer_repl_mark set val = val + 1 where id = ? && channel_id = ?;
```

13.10.12.4 同步 DDL

因为 Drainer 无法为 DDL 操作加入标识表，所以采用单向同步的方式来同步 DDL 操作。

比如集群 A 到集群 B 开启了 DDL 同步，则集群 B 到集群 A 会关闭 DDL 同步。即 DDL 操作全部在 A 上执行。

> 注意：
>
> DDL 操作无法在两个集群上同时执行。执行 DDL 时，若同时有 DML 操作或者 DML binlog 没同步完，会可能出现 DML 同步的上下游表结构不一致的情况。

13.10.12.5 配置并开启双向同步

若要在集群 A 和集群 B 间进行双向同步，假设统一在集群 A 上执行 DDL。在集群 A 到集群 B 的同步路径上，向 Drainer 添加以下配置：

```
[syncer]
loopback-control = true
channel-id = 1 # 互相同步的两个集群配置相同的 ID。
sync-ddl = true # 需要同步 DDL 操作

[syncer.to]
### 1 表示 SyncFullColumn，2 表示 SyncPartialColumn。
### 若设为 SyncPartialColumn，Drainer 会允许下游表结构比当前要同步的数据多或者少列
### 并且去掉 SQL mode 的 STRICT_TRANS_TABLES，来允许少列的情况，并插入零值到下游。
sync-mode = 2

### 忽略 checkpoint 表。
[[syncer.ignore-table]]
db-name = "tidb_binlog"
tbl-name = "checkpoint"
```

在集群 B 到集群 A 的同步路径上，向 Drainer 添加以下配置：

```
[syncer]
loopback-control = true
channel-id = 1 # 互相同步的两个集群配置相同的 ID。
sync-ddl = false  # 不需要同步 DDL 操作。

[syncer.to]
### 1 表示 SyncFullColumn，2 表示 SyncPartialColumn。
### 若设为 SyncPartialColumn，Drainer 会允许下游表结构比当前要同步的数据多或者少列
### 并且去掉 SQL mode 的 STRICT_TRANS_TABLES，来允许少列的情况，并插入零值到下游。
sync-mode = 2

### 忽略 checkpoint 表。
[[syncer.ignore-table]]
db-name = "tidb_binlog"
tbl-name = "checkpoint"
```

13.10.13 TiDB Binlog 术语表

本文档介绍 TiDB Binlog 相关术语。

13.10.13.1 Binlog

在 TiDB Binlog 中，Binlog 通常指 TiDB 写的二进制日志数据，也指 Drainer 写到 Kafka 或者文件的二进制日志数据，前者与后者的格式不同。此外，TiDB 的 binlog 格式与 MySQL 的 binlog 格式也不同。

13.10.13.2 Binlog event

TiDB 写的 DML Binlog 有 3 种 event，分别为：INSERT、UPDATE 和 DELETE。在 Drainer 监控面板上可以看到同步数据对应的不同 event 的个数。

13.10.13.3 Checkpoint

Checkpoint 指保存的断点信息，记录了 Drainer 同步到下游的 commit-ts。Drainer 重启时可以读取 checkpoint，并从对应的 commit-ts 开始同步数据到下游。

13.10.13.4 Safe mode

Safe mode 指增量复制过程中，当表结构中存在主键或唯一索引时，用于支持幂等导入 DML 的模式。

该模式将来自上游的 INSERT 改写为 REPLACE，将 UPDATE 改写为 DELETE 与 REPLACE，然后再向下游执行。在 Drainer 启动的前 5 分钟，会自动启动 Safe mode；另外也可以在配置文件中通过 safe-mode 参数手动开启，但该配置仅在下游是 MySQL 或 TiDB 时有效。

13.10.14 故障诊断

13.10.14.1 TiDB Binlog 故障诊断

本文总结了在 TiDB Binlog 的使用过程中遇到问题的诊断流程，并指引用户通过监控、状态、日志等信息查找相应的解决方案。

如果你在使用 TiDB Binlog 时出现了异常，请尝试以下方式排查问题：

1. 查看各个监控指标是否异常，参见TiDB Binlog 集群监控。
2. 使用binlogctl 工具查看各个 Pump、 Drainer 的状态是否有异常。
3. 查看 Pump、 Drainer 日志中是否有 ERROR、 WARN，并根据详细的日志信息初步判断问题原因。

通过以上方式定位到问题后，在FAQ 以及常见错误及修复 中查找解决方案，如果没有查找到解决方案或者提供的解决方案无法解决问题，请提交 issue，或从 PingCAP 官方或 TiDB 社区获取支持。

13.10.14.2 TiDB Binlog 常见错误修复

本文档介绍 TiDB Binlog 中常见的错误以及修复方法。

13.10.14.2.1 Drainer 同步数据到 Kafka 时报错 “kafka server: Message was too large, server rejected it to avoid allocation error”

报错原因：如果在 TiDB 中执行了大事务，则生成的 binlog 数据会比较大，可能超过了 Kafka 的消息大小限制。

解决方法：需要调整 Kafka 的配置参数，如下所示。

```
message.max.bytes=1073741824
replica.fetch.max.bytes=1073741824
fetch.message.max.bytes=1073741824
```

13.10.14.2.2 Pump 报错 “no space left on device”

报错原因：本地磁盘空间不足，Pump 无法正常写 binlog 数据。

解决方法：需要清理磁盘空间，然后重启 Pump。

13.10.14.2.3 Pump 启动时报错 “fail to notify all living drainer”

报错原因：Pump 启动时需要通知所有 Online 状态的 Drainer，如果通知失败则会打印该错误日志。

解决方法：可以使用binlogctl 工具查看所有 Drainer 的状态是否有异常，保证 Online 状态的 Drainer 都在正常工作。如果某个 Drainer 的状态和实际运行情况不一致，则使用 binlogctl 修改状态，然后再重启 Pump。

13.10.14.2.4 TiDB Binlog 同步中发现数据丢失

需要确认所有 TiDB 实例均开启了 TiDB Binlog，并且运行状态正常。如果集群版本大于 v3.0，可以使用 `curl {TiDB_IP}:{STATUS_PORT}/info/all` 命令确认所有 TiDB 实例上的 TiDB Binlog 状态。

13.10.14.2.5 当上游事务较大时，Pump 报错 `rpc error: code = ResourceExhausted desc = trying to send message larger than max (2191430008 vs. 2147483647)`

出现该错误的原因是 TiDB 发送给 Pump 的 gRPC message 超过限值。可以在启动 Pump 时通过指定 `max-message-size` 来调整 Pump 可接受 gRPC message 的最大大小。

13.10.14.2.6 Drainer 输出 file 格式的增量数据，数据有什么清理机制吗？数据会被删除吗？

- 在 v3.0.x 版本的 Drainer 中，file 格式的增量数据没有任何清理机制。
- 在 v4.0.x 版本中，有基于时间的数据清理机制，详见 Drainer 的 `retention-time` 配置项。

13.10.15 TiDB Binlog 常见问题

本文介绍 TiDB Binlog 使用过程中的常见问题及解决方案。

13.10.15.1 开启 binog 对 TiDB 的性能有何影响？

- 对于查询无影响。
- 对于有写入或更新数据的事务有一点性能影响。延迟上，在 Prewrite 阶段要并发写一条 p-binlog 成功后才可以提交事务，一般写 binlog 比 KV Prewrite 快，所以不会增加延迟。可以在 Pump 的监控面板看到写 binlog 的响应时间。

13.10.15.2 TiDB Binlog 的同步延迟一般为多少？

TiDB Binlog 的同步延迟为秒级别，在非业务高峰时延迟一般为 3 秒左右。

13.10.15.3 Drainer 同步下游 TiDB/MySQL 的账号需要哪些权限？

Drainer 同步账号需要有如下权限：

- Insert
- Update
- Delete
- Create
- Drop
- Alter
- Execute
- Index
- Select
- Create View

13.10.15.4 Pump 磁盘快满了怎么办？

确认 GC 正常：

- 确认 pump 监控面板 gc_tso 时间是否与配置一致。

如 gc 正常以下调整可以降低单个 pump 需要的空间大小：

- 调整 pump GC 参数减少保留数据天数。
- 添加 pump 结点。

13.10.15.5 Drainer 同步中断怎么办？

使用以下 binlogctl 命令查看 Pump 状态是否正常，以及是否全部非 `offline` 状态的 Pump 都在正常运行。

```
binlogctl -cmd pumps
```

查看 Drainer 监控与日志是否有对应报错，根据具体问题进行处理。

13.10.15.6 Drainer 同步下游 TiDB/MySQL 慢怎么办？

特别关注以下监控项：

- 通过 Drainer 监控 drainer event，可以看到 Drainer 当前每秒同步 Insert/Update/Delete 事件到下游的速度。
- 通过 Drainer 监控 sql query time，可以看到 Drainer 在下游执行 SQL 的响应时间。

同步慢的可能原因与解决方案：

- 同步的数据库包含没有主键或者唯一索引的表，需要给表加上主键。
- Drainer 与下游之间延迟大，可以调大 Drainer `worker-count` 参数（跨机房同步建议将 Drainer 部署在下游）。
- 下游负载不高，可以尝试调大 Drainer `worker-count` 参数。

13.10.15.7 假如有一个 Pump crash 了会怎样？

Drainer 会因为获取不到这个 Pump 的数据没法同步数据到下游。如果这个 Pump 能恢复，Drainer 就能恢复同步。

如果 Pump 没法恢复，可采用以下方式进行处理：

1. 使用binlogctl 将该 Pump 状态修改为 `offline`（丢失这个 Pump 的数据）
2. Drainer 获取到的数据会丢失这个 Pump 上的数据，下游跟上游数据会不一致，需要重新做全量 + 增量同步。具体步骤如下：
    1. 停止当前 Drainer。
    2. 上游做全量备份。
    3. 清理掉下游数据，包括 checkpoint 表 `tidb_binlog.checkpoint`。
    4. 使用上游的全量备份恢复下游数据。
    5. 部署 Drainer，使用 `initialCommitTs=`{从全量备份获取快照的时间戳}。

13.10.15.8 什么是 checkpoint？

Checkpoint 记录了 Drainer 同步到下游的 commit-ts，Drainer 重启时可以读取 checkpoint 接着从对应 commit-ts 同步数据到下游。Drainer 日志 `["write save point"] [ts=411222863322546177]` 表示保存对应时间戳的 checkpoint。

下游类型不同，checkpoint 的保存方式也不同：

- 下游 MySQL/TiDB 保存在 `tidb_binlog.checkpoint` 表。
- 下游 kafka/file 保存在对应配置目录里的文件。

因为 kafka/file 的数据内容包含了 commit-ts，所以如果 checkpoint 丢失，可以消费下游最新的一条数据看写到下游数据的最新 commit-ts。

Drainer 启动的时候会去读取 checkpoint，如果读取不到，就会使用配置的 `initial-commit-ts` 做为初次启动开始的同步时间点。

13.10.15.9 Drainer 机器发生故障，下游数据还在，如何在新机器上重新部署 Drainer？

如果下游数据还在，只要保证能从对应 checkpoint 接着同步即可。

假如 checkpoint 还在，可采用以下方式进行处理：

1. 部署新的 Drainer 并启动即可（参考 checkpoint 介绍，Drainer 可以读取 checkpoint 接着同步）。
2. 使用binlogctl 将老的 Drainer 状态修改成 `offline`。

假如 checkpoint 不在，可以如下处理：

1. 获取之前 Drainer 的 checkpoint `commit-ts`，做为新部署 Drainer 的 `initial-commit-ts` 配置来部署新的 Drainer。
2. 使用binlogctl 将老的 Drainer 状态修改成 `offline`。

13.10.15.10 如何用全量 + binlog 备份文件来恢复一个集群？

1. 清理集群数据并使用全部备份恢复数据。
2. 使用 reparo 设置 `start-tso` = {全量备份文件快照时间戳 +1}，`end-ts` = 0（或者指定时间点），恢复到备份文件最新的数据。

13.10.15.11 主从同步开启 `ignore-error` 触发 critical error 后如何重新部署？

TiDB 配置开启 `ignore-error` 写 binlog 失败后触发 critical error 告警，后续都不会再写 binlog，所以会有 binlog 数据丢失。如果要恢复同步，需要如下处理：

1. 停止当前 Drainer。
2. 重启触发 critical error 的 `tidb-server` 实例重新开始写 binlog（触发 critical error 后不会再写 binlog 到 pump）。
3. 上游做全量备份。
4. 清理掉下游数据包括 checkpoint 表 `tidb_binlog.checkpoint`。
5. 使用上游的全量备份恢复下游。
6. 部署 Drainer，使用 `initialCommitTs` = {从全量备份获取快照的时间戳}。

13.10.15.12 同步时出现上游数据库支持但是下游数据库执行会出错的 DDL，应该怎么办？

1. 查看 drainer.log 日志，查找 `exec failed` 找到 Drainer 退出前最后一条执行失败的 DDL。
2. 将 DDL 改为下游兼容支持的版本，在下游数据库中手动执行。
3. 查看 drainer.log 日志，查找执行失败的 DDL 语句，可以查询到该 DDL 的 commit-ts。例如：

```
[2020/05/21 09:51:58.019 +08:00] [INFO] [syncer.go:398] ["add ddl item to syncer, you can
    ↪ add this commit ts to `ignore-txn-commit-ts` to skip this ddl if needed"] [sql="
    ↪ ALTER TABLE `test` ADD INDEX (`index1`)"] ["commit ts"=416815754209656834]。
```

4. 编辑 `drainer.toml` 配置文件，在 `ignore-txn-commit-ts` 项中添加该 commit-ts，重启 Drainer。

在绝大部分情况下，TiDB 和 MySQL 的语句都是兼容的。用户需要注意的是上下游的 `sql_mode` 应当保持一致。

13.10.15.13 在什么情况下暂停和下线 Pump/Drainer？

首先需要通过以下内容来了解 Pump/Drainer 的状态定义和启动、退出的流程。

暂停主要针对临时需要停止服务的场景，例如：

- 版本升级：停止进程后使用新的 binary 启动服务。
- 服务器维护：需要对服务器进行停机维护。退出进程，等维护完成后重启服务。

下线主要针对永久（或长时间）不再使用该服务的场景，例如：

- Pump 缩容：不再需要那么多 Pump 服务了，所以下线部分服务。
- 同步任务取消：不再需要将数据同步到某个下游，所以下线对应的 Drainer。
- 服务器迁移：需要将服务迁移到其他服务器。下线服务，在新的服务器上重新部署。

13.10.15.14 可以通过哪些方式暂停 Pump/Drainer？

- 直接 kill 进程。

> 注意：
> 不能使用 `kill -9` 命令，否则 Pump/Drainer 无法对信号进行处理。

- 如果 Pump/Drainer 运行在前台，则可以通过按下 Ctrl+C 来暂停。
- 使用 binlogctl 的 `pause-pump` 或 `pause-drainer` 命令。

13.10.15.15 可以使用 binlogctl 的 update-pump/update-drainer 命令来下线 Pump/Drainer 服务吗？

不可以。使用 update-pump/update-drainer 命令会直接修改 PD 中保存的状态信息，并且不会通知 Pump/Drainer 做相应的操作。使用不当时，可能会干扰数据同步，某些情况下还可能会造成数据不一致的严重后果。例如：

- 当 Pump 正常运行或者处于暂停状态时，如果使用 update-pump 将该 Pump 设置为 offline，那么 Drainer 会放弃获取处于 offline 状态的 Pump 的 binlog 数据，导致该 Pump 最新的 binlog 数据没有同步到 Drainer，造成上下游数据不一致。
- 当 Drainer 正常运行时，使用 update-drainer 命令将该 Drainer 设置为 offline。如果这时启动一个 Pump 节点，Pump 只会通知 online 状态的 Drainer，导致该 Drainer 没有及时获取到该 Pump 的 binlog 数据，造成上下游数据不一致。

13.10.15.16 可以使用 binlogctl 的 update-pump/update-drainer 命令来暂停 Pump/Drainer 服务吗？

不可以。update-pump/update-drainer 命令直接修改 PD 中保存的状态信息。执行这个命令并不会通知 Pump/Drainer 做相应的操作，而且使用不当会使数据同步中断，甚至造成数据丢失。

13.10.15.17 什么情况下使用 binlogctl 的 update-pump 命令设置 Pump 状态为 paused？

在某些异常情况下，Pump 没有正确维护自己的状态，实际上状态应该为 paused。这时可以使用 update-pump 对状态进行修正，例如：

- Pump 异常退出（可能由 panic 或者误操作执行 kill -9 命令直接 kill 掉进程导致），Pump 保存在 PD 中的状态仍然为 online。如果暂时不需要重启 Pump 恢复服务，可以使用 update-pump 把该 Pump 状态设置为 paused，避免对 TiDB 写 binlog 和 Drainer 获取 binlog 造成干扰。

13.10.15.18 什么情况下使用 binlogctl 的 update-drainer 命令设置 Drainer 状态为 paused？

在某些异常情况下，Drainer 没有正确维护自己的状态，，对数据同步造成了影响，实际上状态应该为 paused。这时可以使用 update-drainer 对状态进行修正，例如：

- Drainer 异常退出（出现 panic 直接退出进程，或者误操作执行 kill -9 命令直接 kill 掉进程），Drainer 保存在 PD 中的状态仍然为 online。当 Pump 启动时无法正常通知该 Drainer（报错 notify drainer ...），导致 Pump 无法正常运行。这个时候可以使用 update-drainer 将 Drainer 状态更新为 paused，再启动 Pump。

13.10.15.19 可以通过哪些方式下线 Pump/Drainer？

目前只可以使用 binlogctl 的 offline-pump 和 offline-drainer 命令来下线 Pump 和 Drainer。

13.10.15.20 什么情况下使用 binlogctl 的 update-pump 命令设置 Pump 状态为 offline?

> 警告：
>
> 仅在可以容忍 binlog 数据丢失、上下游数据不一致或者确认不再需要使用该 Pump 存储的 binlog 数据的情况下，才能使用 update-pump 修改 Pump 状态为 offline。

可以使用 update-pump 修改 Pump 状态为 offline 的情况有：

- 在某些情况下，Pump 异常退出进程，且无法恢复服务，同步就会中断。如果希望恢复同步且可以容忍部分 binlog 数据丢失，可以使用 update-pump 命令将该 Pump 状态设置为 offline，则 Drainer 会放弃拉取该 Pump 的 binlog 然后继续同步数据。
- 有从历史任务遗留下来且不再使用的 Pump 且进程已经退出（例如测试使用的服务），之后不再需要使用该服务，使用 update-pump 将该 Pump 设置为 offline。

在其他情况下一定要使用 offline-pump 命令让 Pump 走正常的下线处理流程。

13.10.15.21 Pump 进程已经退出，且状态为 paused，现在不想使用这个 Pump 节点了，能否用 binlogctl 的 update-pump 命令设置节点状态为 offline？

Pump 以 paused 状态退出进程时，不保证所有 binlog 数据被下游 Drainer 消费。所以这样做会有上下游数据不一致的风险。正确的做法是重新启动 Pump，然后使用 offline-pump 下线该 Pump。

13.10.15.22 什么情况下使用 binlogctl 的 update-drainer 命令设置 Drainer 状态为 offline？

- 有从历史任务遗留下来且不再使用的 Drainer 且进程已经退出（例如测试使用的服务），之后不再需要使用该服务，使用 update-drainer 将该 Drainer 设置为 offline。

13.10.15.23 可以使用 change pump、change drainer 等 SQL 操作来暂停或者下线 Pump/Drainer 服务吗？

目前还不支持。这种 SQL 操作会直接修改 PD 中保存的状态，在功能上等同于使用 binlogctl 的 update-pump、update-drainer 命令。如果需要暂停或者下线，仍然要使用 binlogctl。

13.10.15.24 TiDB 写入 binlog 失败导致 TiDB 卡住，日志中出现 listener stopped, waiting for manual stop

在 TiDB v3.0.12 以及之前，binlog 写入失败会导致 TiDB 报 fatal error。但是 TiDB 不会自动退出只是停止服务，看起来像服务卡住。TiDB 日志中可看到 listener stopped, waiting for manual stop。

遇到该问题需要根据具体情况判断是什么原因导致 binlog 写入失败。如果是 binlog 写入下游缓慢导致的，可以考虑扩容 Pump 或增加写 binlog 的超时时间。

TiDB 在 v3.0.13 版本中已优化此逻辑，写入 binlog 失败将使事务执行失败返回报错，而不会导致 TiDB 卡住。

13.10.15.25 TiDB 向 Pump 写入了重复的 binlog？

TiDB 在写入 binlog 失败或者超时的情况下，会重试将 binlog 写入到下一个可用的 Pump 节点直到写入成功。所以如果写入到某个 Pump 节点较慢，导致 TiDB 超时（默认 15s），此时 TiDB 判定写入失败并尝试写入下一个 Pump 节点。如果超时的 Pump 节点实际也写入成功，则会出现同一条 binlog 被写入到多个 Pump 节点。Drainer 在处理 binlog 的时候，会自动去重 TSO 相同的 binlog，所以这种重复的写入对下游无感知，不会对同步逻辑产生影响。

13.10.15.26 在使用全量 + 增量方式恢复的过程中，Reparo 中断了，可以使用日志里面最后一个 TSO 恢复同步吗？

可以。Reparo 不会在启动时自动开启 safe-mode 模式，需要手动操作：

1. Reparo 中断后，记录日志中最后一个 TSO，记为 `checkpoint-tso`。
2. 修改 Reparo 配置文件，将配置项 `start-tso` 设为 `checkpoint-tso + 1`，将 `stop-tso` 设为 `checkpoint-tso + 80,000,000,000`（大概是 `checkpoint-tso` 延后 5 分钟），将 `safe-mode` 设置为 `true`。启动 Reparo，Reparo 会将数据同步到 `stop-tso` 后自动停止。
3. Reparo 自动停止后，将 `start-tso` 设置为 `checkpoint tso + 80,000,000,001`，将 `stop-tso` 设置为 0，将 `safe-mode` 设为 `false`。启动 Reparo 继续同步。

13.11 PingCAP Clinic 诊断服务

13.11.1 PingCAP Clinic 诊断服务简介

PingCAP Clinic 诊断服务（以下简称为 PingCAP Clinic）是 PingCAP 为 TiDB 集群提供的诊断服务，支持对使用 TiUP 或 TiDB Operator 部署的集群进行远程定位集群问题和本地快速检查集群状态，用于从全生命周期确保 TiDB 集群稳定运行、预测可出现的集群问题、降低问题出现概率、快速定位并修复问题。

PingCAP Clinic 服务提供以下两个组件进行集群诊断：

- Diag 诊断客户端：部署在集群侧的开源工具，用于采集集群的诊断数据 (collect)、上传诊断数据到 Clinic Server、对集群进行本地快速健康检查 (check)。如需了解 Diag 工具可采集的详细的数据列表，请参阅PingCAP Clinic 数据采集说明。

> 注意：
> Diag 诊断客户端支持 TiDB v4.0 及以上的集群，不支持使用 TiDB Ansible 部署的集群。

- Clinic Server：部署在云端的云服务。Clinic Server 提供 SaaS 模式的诊断服务，不仅能接收上传到该组件的诊断数据，也可以提供在线诊断环境，用于存储、查看和诊断已上传的诊断数据，并提供集群诊断报告。根据数据存储的位置不同，Clinic Server 分为以下两个独立的服务：
  - Clinic Server 中国区，数据存储在 AWS 中国区（北京）。
  - Clinic Server 美国区，数据存储在 AWS 美国区。

13.11.1.1 使用场景

- 远程定位集群问题

  当集群出现无法快速修复的问题时，请从 PingCAP 官方或 TiDB 社区获取支持。当申请远程协助时，你需要先保存问题现场的各种诊断数据，然后将其转发给相关技术人员。此时，你可以使用 Diag 诊断客户端，对诊断数据进行一键采集，快速收集完整的诊断数据，替代复杂的手动数据采集操作。随后，你可以将其诊断数据上传到 Clinic Server，供 PingCAP 技术人员查看。Clinic Server 为诊断数据提供了安全的存储，并支持在线诊断，提升了技术人员进行问题定位的效率。

- 快速检查集群状态

  即使集群可以正常运行，也需要定期检查集群是否有潜在的稳定性风险。PingCAP Clinic 提供的本地和 Server 端的快速诊断功能，用于检查集群潜在的健康风险。

13.11.1.2 工作原理

本章节主要介绍 Diag 诊断客户端（以下简称为 Diag）采集集群诊断数据的工作原理。

首先，Diag 需要从部署工具 TiUP (tiup-cluster) 或 TiDB Operator (tidb-operator) 获取集群拓扑信息，然后通过不同的数据采集方式来采集不同类型的诊断数据，具体采集方式如下：

- 通过 SCP 传输服务器文件

  对于使用 TiUP 部署的集群，Diag 可通过 SCP (Secure copy protocol) 直接从目标组件的节点采集日志文件和配置文件。

- 通过 SSH 远程执行命令采集数据

  对于 TiUP 部署的集群，Diag 可以通过 SSH (Secure Shell) 连接到目标组件系统，并可执行 Insight 等命令获取系统信息，包括内核日志、内核参数、系统和硬件的基础信息等。

- 通过 HTTP 调用采集数据

  - 通过调用 TiDB 组件的 HTTP 接口，Diag 可获取 TiDB、TiKV、PD 等组件的实时配置采样信息与实时性能采样信息。
  - 通过调用 Prometheus 的 HTTP 接口，Diag 可获取报警信息和 metrics 监控数据。

- 通过 SQL 语句查询数据库参数

  通过 SQL 语句，Diag 可以查询 TiDB 数据库的系统参数等信息。对于这种方式，你需要在采集数据时额外提供访问 TiDB 数据库的用户名和密码。

13.11.1.3 Clinic Server 使用限制

注意：

- Clinic Server 诊断服务在 2022 年 7 月 15 日至 2024 年 7 月 14 日期间提供免费服务。后续如需收取相关费用，PingCAP Clinic 运营团队将在 2024 年 7 月 14 日前通过邮件通知用户。
- 如果需要调整使用限制，可以联系 PingCAP 技术支持。

诊断服务类型	使用限制
每个组织最多可以创建的集群数量	10 个
诊断数据存储容量	50 GB/集群
诊断数据最长存储时间	180 天
数据包最大大小	3 GB
诊断数据重建保存时间	最长 3 天

13.11.1.4 探索更多

- 在 TiUP 部署环境使用 PingCAP Clinic
  - 快速上手 PingCAP Clinic
  - 使用 PingCAP Clinic 诊断集群
  - 使用 PingCAP Clinic 生成诊断报告
  - PingCAP Clinic 数据采集说明
- 在 TiDB Operator 部署环境使用 PingCAP Clinic
  - 使用 PingCAP Clinic
  - PingCAP Clinic 数据采集说明

13.11.2 PingCAP Clinic 快速上手指南

本指南介绍如何使用 PingCAP Clinic 诊断服务（以下简称为 PingCAP Clinic）快速采集、上传、查看集群诊断数据。

PingCAP Clinic 由 Diag 诊断客户端（以下简称为 Diag）和 Clinic Server 云服务平台（以下简称为 Clinic Server）组成。关于 Diag 和 Clinic Server 的详细介绍，请参考PingCAP Clinic 诊断服务简介。

13.11.2.1 使用场景

- 当集群出现问题，需要远程咨询 PingCAP 技术支持时，为了提高定位和解决问题的效率，你可以使用 Diag 采集诊断数据，上传数据到 Clinic Server 并获取数据链接，然后将数据链接提供给技术支持人员。
- 在集群正常运行时，需要检查集群的运行状态，你可以使用 Diag 采集诊断数据，上传数据到 Clinic Server 并查看 Health Report 的结果。

> 注意：
>
> - 本文档提供的采集和上传数据的方式仅适用于使用TiUP 部署的集群。如需查看适用于使用 Operator 部署的集群，请参阅在 TiDB Operator 部署环境使用 PingCAP Clinic。
> - 通过 PingCAP Clinic 采集和上传的数据仅用于定位和诊断集群问题。

13.11.2.2 准备工作

在开始体验 PingCAP Clinic 功能之前，你需要先安装数据采集组件 Diag 并准备数据上传环境。

1. 在安装了 TiUP 的中控机上，一键安装 Diag：

```
tiup install diag
```

2. 登录 Clinic Server。

   登录 Clinic Server 中国区，选择 Sign in with AskTUG 进入 TiDB 社区 AskTUG 的登录界面。如果你尚未注册 AskTUG 账号，可以在该界面进行注册。

   登录 Clinic Server 美国区，选择 Sign in with TiDB Account 进入 TiDB Cloud Account 的登录界面。如果你尚未注册 TiDB Cloud 账号，可以在该界面进行注册。

> 注意：
>
> Clinic Server 只是通过 TiDB Cloud 账号进行 SSO 登录，并不要求用户必须使用 TiDB Cloud 服务。

3. 在 Clinic Server 中，依据页面提示创建组织 (Organization)。组织是一系列 TiDB 集群的集合。你可以在所创建的组织上上传诊断数据。

4. 获取用于上传数据的 Access Token（以下简称为 Token）。使用 Diag 上传数据时，你需要通过 Token 进行用户认证，以保证数据上传到组织后被安全地隔离。获取一个 Token 后，你可以重复使用该 Token，具体获取方法如下：

   点击 Cluster 页面右下角的图标，选择 Get Access Token For Diag Tool，在弹出窗口中点击 + 符号获取 Token 后，复制并保存 Token 信息。

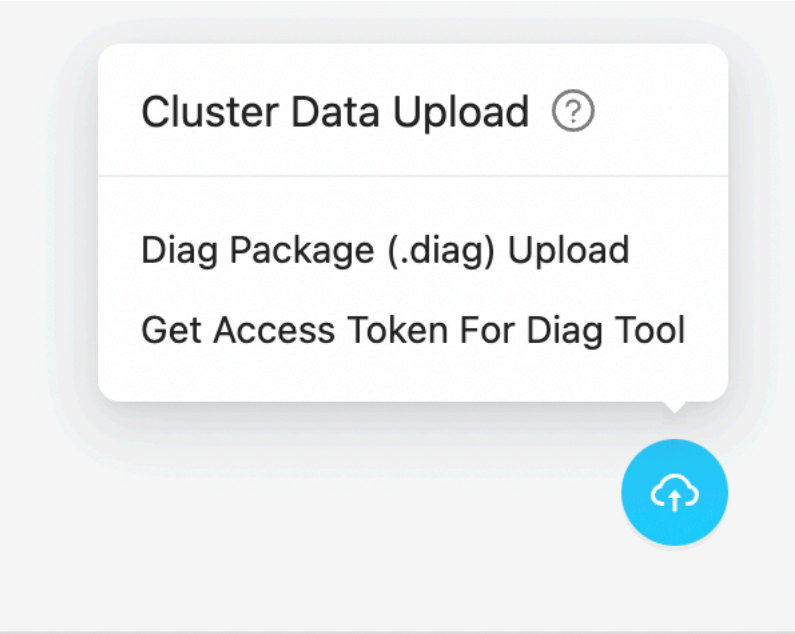


图 291: Token 示例

> 注意：
>
> - 为了确保数据的安全性，TiDB 只在创建 Token 时显示 Token 信息。如果丢失了 Token 信息，你可以删除旧 Token 后重新创建。
> - 该 Token 只用于数据上传，访问数据时不需要使用 Token。

5. 在 Diag 中设置 Token 和 region：

    - 设置 clinic.token：

    ```
    tiup diag config clinic.token ${token-value}
    ```

    - 设置 clinic.region：

    region 决定数据打包时使用的加密证书和上传的目标 Clinic Server 地址。参考以下命令，根据你的 Clinic Server 在 Diag 中设置 clinic.region。

    > 注意：
    > - Diag v0.9.0 及以后的版本支持自行设置 region。
    > - 对于 Diag v0.9.0 之前的版本，数据默认上传到中国区的 Clinic Server。
    > - 如果你的 Diag 是 v0.9.0 之前的版本，你可以通过 tiup update diag 命令将其升级至最新版后设置 region。

    对于 Clinic Server 中国区，参考以下命令，将 region 设置为 CN：

    ```
    tiup diag config clinic.region CN
    ```

    对于 Clinic Server 美国区，参考以下命令，将 region 设置为 US：

    ```
    tiup diag config clinic.region US
    ```

6. 开启日志脱敏配置（可选步骤）。

    TiDB 在提供详细的日志信息时可能会打印数据库的敏感信息（例如用户数据）。如果希望本地日志及上传到 Clinic Server 的日志中不带有敏感信息，你可以开启日志脱敏配置。具体操作请参考日志脱敏。

13.11.2.3 体验步骤

1. 运行 Diag，采集诊断数据。

    例如，如需采集从当前时间的 4 小时前到 2 小时前的诊断数据，可以运行以下命令：

    ```
    tiup diag collect ${cluster-name} -f="-4h" -t="-2h"
    ```

    运行 Diag 数据采集命令后，Diag 不会立即开始采集数据，而会在输出中提供预估数据量大小和数据存储路径，并询问你是否进行数据收集。如果确认要开始采集数据，请输入 Y。

    采集完成后，Diag 会提示采集数据所在的文件夹路径。

2. 将采集到的数据上传到 Clinic Server。

    > 注意：
    > 上传数据（数据包文件夹打包压缩后的文件）的大小不得超过 3 GB，否则会导致上传失败。

- 如果你的集群所在的网络可以访问互联网，你可以通过以下命令上传已采集的数据包文件夹：

```
tiup diag upload ${filepath}
```

完成上传后，Diag 会提示诊断数据的下载路径 Download URL。

> 注意：
> 使用该方式进行上传时，你需要使用 Diag v0.9.0 及以上版本。Diag 会在运行时提示版本信息。如果你早期安装过 v0.9.0 之前版本的 Diag，你可以通过 `tiup update diag` 命令将其一键升级至最新版。

- 如果你所在的集群无法访问互联网，需要先打包数据后进行上传。具体步骤，请参阅上传方式 2：打包后上传。

3. 完成数据上传后，通过上传输出结果中的 Download URL 获取诊断数据的链接。

   诊断数据默认包括集群名称、集群拓扑信息、诊断数据包中的日志内容和基于诊断数据包中的 metrics 信息重建的 Grafana Dashboard 信息。

   你可以通过这些数据自己查找并诊断集群问题，或者，你也可以将链接发给与你对接的 PingCAP 技术支持人员，以协助远程定位集群问题。

4. 查看 Health Report 结果。

   数据上传到 Clinic Server 后，后台将自动处理数据，在大概 5 ~ 15 分钟后生成 Health Report。用户可以打开诊断数据链接，点击下方的 “Health Report” 入口查看报告内容。

13.11.2.4 探索更多

- 在 TiUP 部署环境使用 PingCAP Clinic
  - PingCAP Clinic 诊断服务简介
  - 使用 PingCAP Clinic 诊断集群
  - 使用 PingCAP Clinic 生成诊断报告
  - PingCAP Clinic 数据采集说明
- 在 TiDB Operator 部署环境使用 PingCAP Clinic
  - 使用 PingCAP Clinic
  - PingCAP Clinic 数据采集说明

13.11.3 使用 PingCAP Clinic 诊断集群

对于使用 TiUP 部署的 TiDB 集群和 DM 集群，PingCAP Clinic 诊断服务（以下简称为 PingCAP Clinic）可以通过 Diag 诊断客户端（以下简称为 Diag）与 Clinic Server 云诊断平台（以下简称为 Clinic Server）实现远程定位集群问题和本地快速检查集群状态。

注意：

- 本文档仅适用于使用 TiUP 部署的集群。如需查看适用于使用 Operator 部署的集群，请参阅在 TiDB Operator 部署环境使用 PingCAP Clinic。
- PingCAP Clinic 暂时不支持对使用 TiDB Ansible 部署的集群进行数据采集。

13.11.3.1 使用场景

- 远程定位集群问题
  - 当集群出现问题，需要远程咨询 PingCAP 技术支持时，你可以先使用 Diag 采集诊断数据，然后将其数据上传到 Clinic Server，最后把数据链接提供给技术支持人员，协助远程定位集群问题。
  - 当集群出现问题，但无法马上进行问题分析时，你可以先使用 Diag 采集数据，并将其数据保存下来，用于自己后期进行问题分析。
- 本地快速检查集群状态

  即使集群可以正常运行，也需要定期检查集群是否有潜在的稳定性风险。PingCAP Clinic 提供的本地快速诊断功能，用于检查集群潜在的健康风险，本地诊断只覆盖配置项检查。如果需要更全面的检查，推荐上传诊断数据包到 Clinic Server，使用 Clinic Server 提供的 Health Report 对 Metrics、日志和配置项进行全面的快速检查。

13.11.3.2 准备工作

在使用 PingCAP Clinic 功能之前，你需要先安装数据采集组件 Diag 并准备数据上传环境。

1. 安装 Diag。
   - 如果你的中控机上已经安装了 TiUP，可以使用以下命令一键安装 Diag：

     ```
     tiup install diag
     ```

   - 若已安装了 Diag，你可以通过以下命令，将本地的 Diag 一键升级至最新版本：

     ```
     tiup update diag
     ```

注意：

- 对于离线集群，你需要离线部署 Diag 诊断客户端。具体方法，请参照离线部署 TiUP 组件：方式 2。
- Diag 诊断客户端仅包含在 v6.0.0 及后续版本的 TiDB Server 离线镜像包中。

2. 获取并设置用于上传数据的 Access Token（以下简称为 Token）。

   使用 Diag 上传采集到的数据时，你需要通过 Token 进行用户认证，以保证数据上传到组织后被安全地隔离。获取一个 Token 后，你可以重复使用该 Token。如果你已经获取过并在 Diag 上设置过 Token，可跳过此步骤。

   首先，通过以下方法获取 Token：

   - 登录 Clinic Server。

     Clinic Server 中国区，数据存储在亚马逊云服务中国区。

     Clinic Server 美国区，数据存储在亚马逊云服务美国区。

   - 点击 Cluster 页面右下角的图标，选择 Get Access Token For Diag Tool，在弹出窗口中点击 + 符号获取 Token 后，复制并保存 Token 信息。

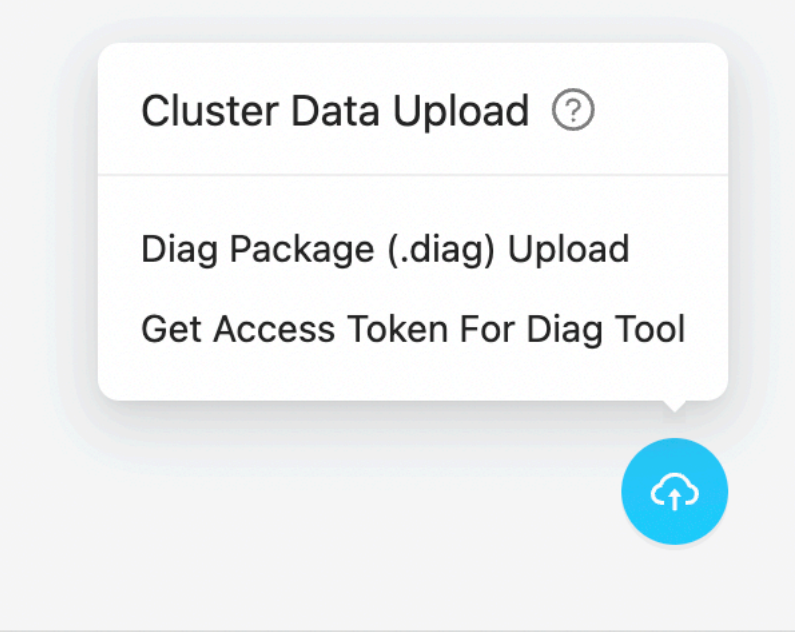


图 292: Token 示例

> 注意：
>
> - 如果你第一次访问 Clinic Server，请参考快速上手指南：准备数据上传环境的相关步骤。
> - 为了确保数据的安全性，TiDB 只在创建 Token 时显示 Token 信息。如果丢失了 Token 信息，你可以删除旧 Token 后重新创建。
> - Token 只用于上传数据。

   然后，参考以下命令，在 Diag 中设置该 Token：

```
tiup diag config clinic.token ${token-value}
```

3. 在 Diag 中设置 `region`。

`region` 决定数据打包时使用的加密证书和上传的目标 Clinic Server 地址。参考以下命令，根据你的 Clinic Server 在 Diag 中设置 `clinic.region`。

> **注意：**
>
> - Diag v0.9.0 及以后的版本支持 `region` 设置。
> - 对于 Diag v0.9.0 之前的版本，数据默认上传到 Clinic Server 中国区。
> - 如果你的 Diag 是 v0.9.0 之前的版本，你可以通过 `tiup update diag` 命令将其升级至最新版本后设置 `region`。

对于 Clinic Server 中国区，参考以下命令，将 `region` 设置为 `CN`：

```
tiup diag config clinic.region CN
```

对于 Clinic Server 美国区，参考以下命令，将 `region` 设置为 `US`：

```
tiup diag config clinic.region US
```

4. 开启日志脱敏配置（可选步骤）。

   TiDB 在提供详细的日志信息时可能会打印数据库的敏感信息（例如用户数据）。如果希望本地日志及上传到 Clinic Server 的日志中不带有敏感信息，你可以开启日志脱敏配置。具体操作请参考日志脱敏。

13.11.3.3 远程定位集群问题

你可以使用 Diag 快速抓取 TiDB 集群和 DM 集群的诊断数据，其中包括监控数据、配置信息等。

13.11.3.3.1 第 1 步：确定需要采集的数据

如需查看 Diag 支持采集的数据的详细列表，请参阅PingCAP Clinic 数据采集说明。

建议收集监控数据、配置信息等全量诊断数据，有助于提升后续诊断效率。具体方法，请参考采集集群的数据。

13.11.3.3.2 第 2 步：采集数据

你可以使用 Diag 采集使用 TiUP 部署的 TiDB 集群和 DM 集群的数据。

1. 运行 Diag 数据采集命令。

   例如，如需采集集群从当前时间的 4 小时前到 2 小时前的诊断数据，可以运行以下命令：

```
tiup diag collect ${cluster-name} -f="-4h" -t="-2h"
```

```
tiup diag collectdm ${dm-cluster-name} -f="-4h" -t="-2h"
```

   采集参数说明：

- `-f/--from`：指定采集时间的起始点。如果不指定该参数，默认起始点为当前时间的 2 小时前。如需修改时区，可使用 `-f="12:30 +0800"` 语法。如果没有在该参数中指定时区信息，如 `+0800`，则默认时区为 UTC。
- `-t/--to`：指定采集时间的结束点。如果不指定该参数，默认结束点为当前时刻。如需修改时区，可使用 `-f="12:30 +0800"` 语法。如果没有在该参数中指定时区信息，如 `+0800`，则默认时区为 UTC。

参数使用提示：

除了指定采集时间，你还可以使用 Diag 指定更多参数。如需查看所有参数，请使用 `tiup diag collect -h` 或 `tiup diag collectdm -h` 命令。

> 注意：
>
> - Diag 默认不收集系统变量数据 (db_vars)。如需收集该数据，你需要额外提供开启了系统变量可读权限的数据库用户名和密码。
> - Diag 默认不收集性能数据 (perf) 和 Debug 数据 (debug)。
> - 如需收集全量诊断数据，可以使用命令 `tiup diag collect <cluster-name> --include="system,monitor,log,config,db_vars,perf,debug"`。

- `-l`：传输文件时的带宽限制，单位为 Kbit/s，默认值为 100000（即 scp 的 `-l` 参数）。
- `-N/--node`：支持只收集指定节点的数据，格式为 `ip:port`。
- `--include`：只收集特定类型的数据，可选值为 `system`，`monitor`，`log`，`config`，`db_vars`。如需同时列出多种类型的数据，你可以使用逗号，来分割不同的数据类型。
- `--exclude`：不收集特定类型的数据，可选值为 `system`，`monitor`，`log`，`config`，`db_vars`。如需同时列出多种类型的数据，你可以使用逗号，来分割不同的数据类型。

运行 Diag 数据采集命令后，Diag 不会立即开始采集数据，而会在输出中提供预估数据量大小和数据存储路径，并询问你是否进行数据收集。例如：

```
Estimated size of data to collect:
Host                Size       Target
----                ----       ------
172.16.7.129:9090   43.57 MB   1775 metrics, compressed
172.16.7.87         0 B        /tidb-deploy/tidb-4000/log/tidb_stderr.log
... ...
172.16.7.179        325 B      /tidb-deploy/tikv-20160/conf/tikv.toml
Total               2.01 GB    (inaccurate)
These data will be stored in /home/user/diag-fNTnz5MGhr6
Do you want to continue? [y/N]: (default=N)
```

2. 如果确认要开始采集数据，请输入 Y。

   采集数据需要一定的时间，具体所需时间与需要收集的数据量有关。例如，在测试环境中收集 1 GB 数据，大概需要 10 分钟。

   采集完成后，Diag 会提示采集数据所在的文件夹路径。例如：

```
Collected data are stored in /home/user/diag-fNTnz5MGhr6
```

13.11.3.3.3 第 3 步：本地查看数据（可选步骤）

已收集的数据会根据其数据来源存储于独立的子目录中，这些子目录以机器名和端口号来命名。每个节点的配置、日志等文件的存放位置与在真实服务器中存放的相对路径相同，其中：

- 系统和硬件的基础信息：位于 `insight.json`
- 系统 `/etc/security/limits.conf` 中的内容：位于 `limits.conf`
- 内核参数列表：位于 `sysctl.conf`
- 内核日志：位于 `dmesg.log`
- 采集时的网络连接情况：位于 `ss.txt`
- 配置数据：位于每节点目录下的 `config.json`
- 集群本身的元信息：位于 `meta.yaml`（此文件位于采集数据存储目录的顶层）
- 监控数据：位于 `/monitor` 文件目录。默认经过压缩，无法直接查看。如需直接查看监控指标的 JSON 文件内容，可在采集时通过 `--compress-metrics=false` 参数禁用压缩。

13.11.3.3.4 第 4 步：上传数据

如需将集群诊断数据提供给 PingCAP 技术支持人员，请先将数据上传到 Clinic Server 后，再把获取到的数据访问链接发送给技术支持人员。Clinic Server 为 PingCAP Clinic 的云服务，可提供安全的诊断数据存储和共享。

根据集群的网络连接情况，你可以选择以下上传方式之一：

- 方式 1：如果集群所在的网络能访问互联网，你可以通过上传命令直接上传数据。
- 方式 2：如果集群所在的网络不能访问互联网，你需要打包后再上传数据。

> 注意：
>
> 如果在上传前没有在 Diag 中设置 Token 或 `region`，Diag 会提示上传失败，并提醒你进行设置。关于 Token 获取方法，请参考准备工作：第 2 步。

方式 1：直接上传

如果你的集群所在的网络可以访问互联网，你可以直接通过以下命令上传在第 2 步：采集数据中收集的数据包文件夹：

```
bash tiup diag upload
```

完成上传后，Diag 会提示诊断数据的下载路径 Download URL。你可以打开 Download URL 中的链接查看数据，也可以将 Download URL 中的链接发给与你对接的 PingCAP 技术支持人员。

方式 2：打包后上传

如果你的集群所在的网络无法访问互联网，你需要先在内网打包数据后，再将其数据包发送到网络连通的设备上进行上传。具体操作方法如下：

1. 打包在第 2 步：采集数据中采集的数据，并对其数据包进行压缩和加密：

```
tiup diag package ${filepath}
```

打包时，Diag 会同时对数据进行压缩和加密。在测试环境中，800 MB 数据压缩后变为 57 MB。示例输出如下：

```
Starting component `diag`: /root/.tiup/components/diag/v0.7.0/diag package diag-fNTnz5MGhr6
packaged data set saved to /home/user/diag-fNTnz5MGhr6.diag
```

完成打包后，数据包为 .diag 格式。只有上传到 Clinic Server 后，该数据包才能被解密并查看。如需直接转发已收集的数据，而不在 Clinic Server 中查看，你可以自行压缩后转发数据。

2. 使用可以访问互联网的机器上传数据压缩包。

```
tiup diag upload ${filepath}
```

完成上传后，Diag 会提示诊断数据的下载路径 Download URL。你可以打开 Download URL 中的链接，在 Clinic Server 页面进行数据查看，也可以将 Download URL 中的链接发给与你对接的 PingCAP 技术支持人员。

13.11.3.4 本地快速检查集群状态

你可以使用 Diag 对集群状态进行快速诊断。即使集群可以正常运行，也需要定期检查集群是否有潜在的稳定性风险。PingCAP Clinic 提供的本地快速诊断功能，用于检查集群潜在的健康风险，本地诊断只覆盖配置项检查。如果需要更全面的检查，推荐上传诊断数据包到 Clinic Server，使用 Clinic Server 提供的 Health Report 对 Metrics、日志和配置项进行全面的快速检查。

1. 采集配置数据：

```
tiup diag collect ${cluster-name} --include="config"
```

配置文件数据较小，采集后会默认存放至当前路径下。在测试环境中，对于一个 18 个节点的集群，配置文件数据量小于 10 KB。

2. 诊断配置数据：

```
tiup diag check ${subdir-in-output-data}
```

其中，`${subdir-in-output-data}` 为采集数据的存放路径，其路径中存放 `meta.yaml` 文件。

3. 查看诊断结果：

诊断结果会在命令行中返回，示例如下：

```
Starting component `diag`: /root/.tiup/components/diag/v0.7.0/diag check diag-fNTnz5MGhr6

# 诊断结果
lili 2022-01-24T09:33:57+08:00

## 1. 诊断集群名称等基础信息
- Cluster ID: 7047403704292855808
- Cluster Name: lili
- Cluster Version: v5.3.0
```

```
## 2. 诊断数据来源信息
- Sample ID: fNTnz5MGhr6
- Sampling Date: 2022-01-24T09:33:57+08:00
- Sample Content:: [system monitor log config]

## 3. 诊断结果信息，包括发现的可能的配置问题
In this inspection, 22 rules were executed.

The results of **1** rules were abnormal and needed to be further discussed with support
    ↪ team.

The following is the details of the abnormalities.

### 诊断结果摘要
The configuration rules are all derived from PingCAP's OnCall Service.

If the results of the configuration rules are found to be abnormal, they may cause the
    ↪ cluster to fail.

There were **1** abnormal results.

#### 诊断结果文档的保存路径
Rule Name: tidb-max-days
- RuleID: 100
- Variation: TidbConfig.log.file.max-days
- For more information, please visit: https://s.tidb.io/msmo6awg
- Check Result:
  TidbConfig_172.16.7.87:4000   TidbConfig.log.file.max-days:0   warning
  TidbConfig_172.16.7.86:4000   TidbConfig.log.file.max-days:0   warning
  TidbConfig_172.16.7.179:4000   TidbConfig.log.file.max-days:0   warning

Result report and record are saved at diag-fNTnz5MGhr6/report-220125153215
```

在上述示例诊断结果信息的最后一部分（即“#### 诊断结果文档的保存路径”）中，对于被发现的每一条潜在的配置问题，Diag 都会提供对应的知识库链接，以便查看详细的配置建议。在上面示例中，相关链接为 `https://s.tidb.io/msmo6awg`。

13.11.3.5 常见问题

1. 如果数据上传失败了，可以重新上传吗？

   可以。数据上传支持断点上传，如果上传失败了，可以直接再次上传。

2. 数据上传后，无法打开返回的数据访问链接，怎么办？

   你可以先尝试登录 Clinic Server。如果登录后依然无法打开链接，请确认你是否拥有访问该数据的权限。如果没有权限，你需要联系数据所有人给你添加权限后，重新登录 Clinic Server 并访问数据链接。

3. 上传到 Clinic Server 的数据后会保存多久？

   最长 180 天，用户可以随时通过 Clinic Server 页面删除自己上传的集群诊断数据。

13.11.4 使用 PingCAP Clinic 生成诊断报告

PingCAP Clinic Server 云诊断平台（以下简称为 Clinic Server）可以基于上传的诊断数据生成诊断报告，包括 Benchmark Report 和 Comparison Report，用于在特定场景下检查集群健康风险，并协助定位集群问题。本文介绍Benchmark Report 和Comparison Report 报告的使用场景和使用方法。

> **注意：**
>
> 使用 PingCAP Clinic 生成的报告是基于诊断数据中的 Metrics 数据和日志数据。要获取完整的诊断建议，你需要完整收集集群诊断数据。

13.11.4.1 使用场景

- 压力测试场景

  在性能测试或新业务压力接入测试阶段，需要对系统性能进行评估，并通过调整上层业务请求和下层数据库配制参数来提升整体性能。通过Benchmark Report，可以检查当前系统性能状态，发现风险问题，提出业务层和集群配置的优化建议。

- 性能对比检查场景

  很多场景需要对比集群各种关键指标，通过Comparison Report，可以快速了解机器不同时间段运行状态的差异。相对于手动对比，Comparison Report 能够提升效率和准确性。对比的内容包括：QPS 差异、延迟差异、性能指标差异、系统资源使用差异和关键日志差异等。需要使用 Comparison Report 对比性能的场景有：

  - 集群健康巡检时，对比本次性能与上次巡检的差异
  - 集群修改了关键参数或配置后，对比修改前后的差异
  - 集群升级后，对比升级前后的差异
  - 集群接入新业务后，对比接入新业务前后的差异

13.11.4.2 前提条件

要生成 PingCAP Clinic 报告，需要使用 Diag 诊断客户端采集集群诊断数据，并上传到 Clinic Server。具体使用 Diag 采集诊断数据的方法，在 TiUP 部署环境参见使用 PingCAP Clinic 诊断集群，TiDB Operator 部署环境参见使用 PingCAP Clinic。

13.11.4.3 使用方法

13.11.4.3.1 Benchmark Report

生成报告

- 生成默认报告

  Benchmark Report 会在数据包上传到 Clinic Server 后自动生成。由于生成报告需要先将相关诊断数据从数据包中解压提取，所以生成报告时间会在 5 分钟以上（数据包越大，所需时间越长）。

- 生成自定义报告

  默认生成的报告是基于数据包的起止时间，如果需要自定义生成报告的时间，可以按如下步骤进行自定义：

  1. 登录 Clinic Server（中国区、美国区）
  2. 打开需要生成诊断报告的集群页面
  3. 点击集群页面上的 Benchmark Report
  4. 在 Time Range 部分选择报告的开始 (Start date) 和结束时间 (End date)
  5. 点击 Create Report 生成自定义报告

解读报告

打开 Clinic Server 集群页，点击页面上的 Benchmark Report，在 All Reports 列表中点击 View 可以查看相应的报告，报告内容示例如下：

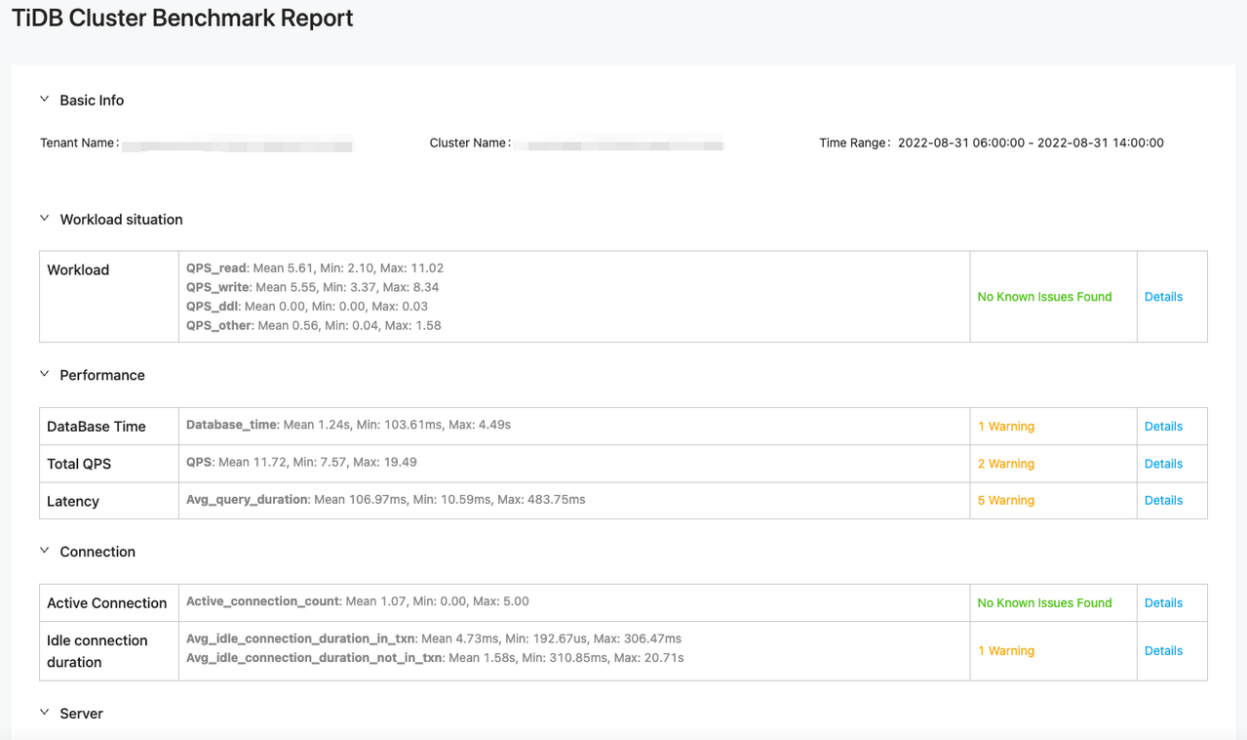


图 293: benchmark-report

> 注意：
>
> 报告功能会持续优化，实际生成的内容可能比下面的说明更丰富。

报告分为以下几个部分：

- Workload situation：业务侧性能指标结果和建议。该部分的建议主要是针对业务请求的调整。
- Performance：数据库主要性能指标分析和建议。该部分分析节点是否均衡、数据库时间 (DB Time) 中主要阶段的时延是否在合理区间，并基于业务压力和时延数据提供配置优化的建议。
- Connection：连接数据分析和建议。该部分分析各节点连接是否均衡、空闲连接是否异常。
- Server：各节点资源使用情况分析。该部分分析各节点资源使用的现状和趋势。对于资源使用不均衡，或在短时间内资源使用可能会超过阈值的情况显示 Warning。

13.11.4.3.2 Comparison Report

生成报告

生成 Comparison Report 的步骤如下：

1. 登录 Clinic Server（中国区、美国区）
2. 打开需要生成诊断报告的集群页面
3. 点击集群页面上的 Comparison Report
4. 选择两个对比时间段 T1 Range 和 T2 Range
5. 点击 Create Report 生成自定义报告

生成 Comparison Report 大约需要 5 ~ 30 分钟。因为报告需要处理日志数据，所以报告生成时间与对比时间段中日志数据量相关。

解读报告

Comparison Report 会基于所选的两个时间段的 Metrics 和日志进行对比分析。报告中 Metrics 的计算方法是对两个时间段相应的指标时间序列数据取均值，然后将均值进行对比。

打开 Clinic Server 集群页，点击页面上的 Comparison Report，在 All Reports 列表中点击 View 可以查看相应的报告，报告内容示例如下：

TiDB 集群诊断数据对比报告

基本信息

报告区间

T1.START_TIME	T1.END_TIME	T2.START_TIME	T2.END_TIME
2022-08-31 12:00:00	2022-08-31 16:00:00	2022-08-31 17:30:00	2022-08-31 19:40:00

主要指标对比

Metrics	T1 value	T2 value	Diff	View
QPS	12.98	15.75	21.35%	T1 T2
Connnection_count	20.61	23.15	12.31%	T1 T2
Active_connection_count	1.27	1.20	-5.37%	T1 T2
CPS	13.07	15.96	22.09%	T1 T2
Database_time	1.27s	1.20s	-5.37%	T1 T2
Avg_cluster_duration	96.76ms	74.13ms	-23.39%	T1 T2
P99_cluster_duration	474.62ms	405.31ms	-14.60%	T1 T2
TiDB_CPU_max	59.72%	60.82%	1.84%	T1 T2
TiKV_CPU_max	434.00%	812.08%	87.11%	T1 T2
TiKV_IO_MBps_max	12.18MB/s	312.46MB/s	2466.19%	T1 T2
TiKV_disk_usage_max	47.05%	47.07%	0.03%	T1 T2
Error_log_count	1142	502	-56.04%	T1 T2

图 294: comparison-report

注意：

报告功能会持续优化，实际生成的内容可能比下面的说明更丰富。

报告分为以下几个部分：

- 基本信息：列出 Comparison Report 对比的两个时间段信息。
- 主要指标对比：列出最关键的 12 个指标在前后两个时间段的对比。
- 业务指标对比：列出以下 5 个关键业务指标的对比：
  - QPS：每秒钟的请求次数。可以点击 More 对比不同请求类型的 QPS
  - Connection_count：连接数量。可以点击 More 对比不同 TiDB 节点的连接数量
  - Active_connection_count：活跃连接数量
  - CPS：每秒处理的命令数。可以点击 More 对比不同命令类型的 CPS
  - Plan_cache_OPS：所有 TiDB 实例每秒使用 Plan Cache 的查询数量
- 性能指标对比：列出系统中最重要的二十多个性能指标的对比，包括以下几个部分：
  - 基础指标：数据库时间和延迟等最基础的性能指标对比
  - 基于 SQL Phase 拆分的数据库时间对比
  - 数据库时间中耗时最长的 SQL 类型对比
  - SQL_execute_time 指标中耗时最长的请求类型对比
  - Duration 指标中耗时最长的 SQL 类型对比
  - KV request 细节数据对比
  - 其他性能指标对比
- 资源指标对比：列出 TiDB、TiKV、TiFlash 和 PD 节点的资源指标对比。不仅包括节点本身的对比，还会计算同类型节点资源使用的最大差值 delta，基于 delta 进行对比，展示节点不均衡状态的变化。
- 日志聚类对比：列出不同类型日志在前后两个时间段的数量，可以通过日志数量的变化了解系统内部任务运行情况的变化。

13.11.4.4 探索更多

- 在 TiUP 部署环境使用 PingCAP Clinic
  - 快速上手 PingCAP Clinic
  - PingCAP Clinic 诊断服务简介
  - 使用 PingCAP Clinic 诊断集群
  - PingCAP Clinic 数据采集说明
- 在 TiDB Operator 部署环境使用 PingCAP Clinic
  - 使用 PingCAP Clinic
  - PingCAP Clinic 数据采集说明

13.11.5 使用 PingCAP Clinic Diag 采集 SQL 查询计划信息

TiDB 在 v5.3.0 中引入了PLAN REPLAYER 命令，能够快速保存和恢复查询计划相关的信息，简化了排查优化器相关问题时提取排查信息的步骤。Clinic Diag 诊断客户端（下称 Diag）集成了 PLAN REPLAYER 功能，你可以使用 Diag 方便快捷地保存查询计划相关的数据。

13.11.5.1 使用说明

在排查 TiDB v4.0 及以上版本的集群问题时，你可以使用 Diag 将排查现场的相关数据以 ZIP 格式导出，并将导出的数据快速上传到 PingCAP Clinic 供技术支持人员查看。在使用 Diag 采集数据前，你需要提供需要采集的 SQL 语句文件。Diag 采集的数据相较于 TiDB 中 PLAN REPLAYER 功能采集的数据多了日志信息和集群信息，具体见全量数据采集的输出结果。

> **警告：**
>
> - 当前该功能为实验特性，不建议在生产环境中使用。
> - 暂时不支持采集使用 TiDB Operator 部署的 TiDB 集群的数据。

13.11.5.2 使用方法

本部分介绍如何使用 Diag 采集 SQL 查询计划信息。你需要先安装 Diag，再使用 Diag 采集数据。

13.11.5.2.1 安装 Diag

- 通过 TiUP 安装 Diag，默认安装最新版本：

```
tiup install diag
```

- 若已安装 Diag，请确保 Diag 版本 >= 0.7.3。

  你可以通过如下命令检查 Diag 版本：

```
tiup diag --version
```

  如果 Diag 版本不满足要求，可以通过如下命令升级到最新版本：

```
tiup update diag
```

13.11.5.2.2 全量数据采集

使用如下 `diag collect` 命令进行数据采集。你需要将 `<cluster-name>` 和 `<statement-filepath>` 占位符替换为实际的值：

```
diag collect <cluster-name> --profile=tidb-plan-replayer --explain-sql=<statement-filepath>
```

> **注意：**
>
> - 通过 `diag` 采集数据前，你需要通过 `--explain-sql` 指定 `sql-statement` 文件：
>   - 以上命令中 `<statement-filepath>` 是指 `sql-statement` 文件的路径。

  - 该文件包含了需要采集的 SQL 语句。
  - 如果有多条 SQL 语句需要分析，可以使用分号 ; 分隔。
  - 因为以上 diag 命令会新建会话进行查询，所以 SQL 文件里的 SQL 语句必须显示指明所使用的数据库，如 SELECT * FROM test.t1。
- PLAN REPLAYER 不会导出表中数据。
- 采集数据时只会执行 EXPLAIN，不会真正执行查询。因此采集时对数据库性能影响较小。

sql-statement 的文件内容示例：

```
SELECT * FROM test.t1;SELECT * FROM test.t2;
```

--explain-sql 指定的 sql-statement 文件内容说明：

- 如果有多条 SQL 语句需要分析，可以使用分号 ; 分隔。
- 因为以上 diag 命令会新建会话进行查询，所以 SQL 文件里的 SQL 语句必须显示指明所使用的数据库，如 SELECT * FROM test.t1。

输出结果

全量数据采集的输出结果包含以下集群现场信息：

序号	采集内容	调用的 Diagnostic collector	输出文件
1	TiDB 配置信息	config	tidb.toml
2	TiDB Session 系统变量	plan_replayer	plan_replayer.zip/variables.toml
3	TiDB 执行计划绑定信息 (SQL Binding)	sql_bind	sql_bind/global_bind.csv

序号	采集内容	调用的 Diagcollector	输出文件
4	sql-statement 中所包含的表结构	plan_replayer	plan_replayer.zip/schema/<db.table>.schema.txt
5	sql-statement 中所包含表的统计信息	plan_replayer	plan_replayer.zip/stats/<db.table>.json
6	EXPLAIN sql-statement 的结果	explain	explain/sql0

| 7 | 默认采集的集群信息

集群基础信息

Diag 本次采集记录

| default | cluster.json, meta.yaml, $collectid_diag_audit.log |

13.11.5.2.3 自定义数据采集

你可以自定义配置文件使 Diag 采集上述输出结果中的部分数据。下面为一个示例的配置文件 `tidb-plan-replayer.toml`：

```
name = "tidb-plan-replayer"
version = "0.0.1"
maintainers = [
    "pingcap"
]

### 要采集的数据类型
collectors = [
    "config",
    "sql_bind",
    "plan_replayer"
]

### 要采集的组件
roles = [
    "tidb"
]
```

通过 `--profile` 参数指定配置文件的路径来进行自定义采集。你需要将 `<cluster-name>`、`<profile-filepath>` 和 `<statement-filepath>` 替换为实际的值：

```
diag collect <cluster-name> --profile=<profile-filepath> --explain-sql=<statement-filepath>
```

13.11.5.2.4 将结果导入到 TiDB 集群

采集结果中的 `plan_replayer.zip` 可以通过 `PLAN REPLAYER LOAD` 语句直接导入到 TiDB 集群中进行信息查看，具体方法可参考使用 `PLAN REPLAYER` 导入集群信息

13.11.6 PingCAP Clinic 数据采集说明

本文提供了 PingCAP Clinic 诊断服务（以下简称为 PingCAP Clinic）在使用 TiUP 部署的 TiDB 集群和 DM 集群中能够采集的诊断数据类型，并列出了各个采集项对应的采集参数。当执行 Clinic Diag 诊断客户端（以下简称为 Diag）数据采集命令时，你可以依据需要采集的数据类型，在命令中添加所需的采集参数。

通过 PingCAP Clinic 在使用 TiUP 部署的集群中采集的数据仅用于诊断和分析集群问题。

Clinic Server 是部署在云端的云服务，根据数据存储的位置不同，分为以下两个独立的服务：

- Clinic Server 中国区：如果你把采集的数据上传到了 Clinic Server 中国区，这些数据将存储于 PingCAP 设立在 AWS 中国区（北京）的 S3 服务。PingCAP 对数据访问权限进行了严格的访问控制，只有经授权的内部技术人员可以访问该数据。

- Clinic Server 美国区：如果你把采集的数据上传到了 Clinic Server 美国区，这些数据将存储于 PingCAP 设立在 AWS 美国区的 S3 服务。PingCAP 对数据访问权限进行了严格的访问控制，只有经授权的内部技术人员可以访问该数据。

13.11.6.1 TiDB 集群

本节列出了 Diag 在使用 TiUP 部署的 TiDB 集群中能够采集的诊断数据类型。

13.11.6.1.1 TiDB 集群信息

诊断数据类型	输出文件	PingCAP Clinic 采集参数
集群基础信息，包括集群 ID	`cluster.json`	每次收集默认采集
集群详细信息	`meta.yaml`	每次收集默认采集

13.11.6.1.2 TiDB 诊断数据

诊断数据类型	输出文件	PingCAP Clinic 采集参数
日志	`tidb.log`	`--include=log`
Error 日志	`tidb_stderr.log`	`--include=log`
慢日志	`tidb_slow_query.log`	`--include=log`
配置文件	`tidb.toml`	`--include=config`
实时配置	`config.json`	`--include=config`

13.11.6.1.3 TiKV 诊断数据

诊断数据类型	输出文件	PingCAP Clinic 采集参数
日志	`tikv.log`	`--include=log`
Error 日志	`tikv_stderr.log`	`--include=log`
配置文件	`tikv.toml`	`--include=config`
实时配置	`config.json`	`--include=config`

13.11.6.1.4 PD 诊断数据

诊断数据类型	输出文件	PingCAP Clinic 采集参数
日志	`pd.log`	`--include=log`
Error 日志	`pd_stderr.log`	`--include=log`
配置文件	`pd.toml`	`--include=config`

诊断数据类型	输出文件	PingCAP Clinic 采集参数
实时配置	config.json	--include=config
tiup ctl:v< CLUSTER_VERSION > pd -u http://${pd IP}:${PORT} store 的输出结果	store.json	--include=config
tiup ctl:v< CLUSTER_VERSION > pd -u http://${pd IP}:${PORT} config placement -rule show 的输出结果	placement-rule.json	--include=config

13.11.6.1.5 TiFlash 诊断数据

诊断数据类型	输出文件	PingCAP Clinic 采集参数
日志	tiflash.log	--include=log
Error 日志	tiflash_stderr.log	--include=log

诊断数据类型	输出文件	PingCAP Clinic 采集参数
配置文件	`tiflash-learner.toml, tiflash-preprocessed.toml, tiflash.toml`	`--include=config`
实时配置	`config.json`	`--include=config`

13.11.6.1.6 TiCDC 诊断数据

诊断数据类型	输出文件	PingCAP Clinic 采集参数
日志	`ticdc.log`	`--include=log`
Error 日志	`ticdc_stderr.log`	`--include=log`
配置文件	`ticdc.toml`	`--include=config`
Debug 数据	`info.txt, status.txt, changefeeds.txt, captures.txt, processors.txt`	`--include=debug`（默认不采集）

13.11.6.1.7 Prometheus 监控数据

诊断数据类型	输出文件	PingCAP Clinic 采集参数
所有的 Metrics 数据	`{metric_name}.json`	`--include=monitor`
Alert 列表	`alerts.json`	`--include=monitor`

13.11.6.1.8 TiDB 系统变量

诊断数据类型	输出文件	PingCAP Clinic 采集参数
获取 TiDB 系统变量（默认不采集，采集需要额外提供数据库账号）	`mysql.tidb.csv`	`--include=db_vars`（默认不采集
	`global_variables.csv`	`--include=db_vars`（默认不采集

13.11.6.1.9 集群节点的系统信息

诊断数据类型	输出文件	PingCAP Clinic 采集参数
内核日志	`dmesg.log`	`--include=system`

诊断数据类型	输出文件	PingCAP Clinic 采集参数
系统和硬件的基础信息	`insight.json`	`--include=system`
系统 `/etc/security/limits.conf` 中的内容	`limits.conf`	`--include=system`
内核参数列表	`sysctl.conf`	`--include=system`
socket 统计信息（即 ss 的命令结果）	`ss.txt`	`--include=system`

13.11.6.2 DM 集群

本节列出了 Diag 在使用 TiUP 部署的 DM 集群中能够采集的诊断数据类型。

13.11.6.2.1 DM 集群信息

诊断数据类型	输出文件	PingCAP Clinic 采集参数
集群基础信息，包括集群 ID	`cluster.json`	每次收集默认采集
集群详细信息	`meta.yaml`	每次收集默认采集

13.11.6.2.2 dm-master 诊断数据

诊断数据类型	输出文件	PingCAP Clinic 采集参数
日志	`m-master.log`	`--include=log`
Error 日志	`dm-master_stderr.log`	`--include=log`
配置文件	`dm-master.toml`	`--include=config`

13.11.6.2.3 dm-worker 诊断数据

诊断数据类型	输出文件	PingCAP Clinic 采集参数
日志	`dm-worker.log`	`--include=log`
Error 日志	`dm-worker_stderr.log`	`--include=log`
配置文件	`dm-work.toml`	`--include=config`

13.11.6.2.4 Prometheus 监控数据

诊断数据类型	输出文件	PingCAP Clinic 采集参数
所有的 Metrics 数据	`{metric_name}.json`	`--include=monitor`
Alert 列表	`alerts.json`	`--include=monitor`

13.11.6.2.5 集群节点的系统信息

诊断数据类型	输出文件	PingCAP Clinic 采集参数
内核日志	`dmesg.log`	`--include=system`
系统和硬件基础信息	`insight.json`	`--include=system`
系统 `/etc/security/limits.conf` 中的内容	`limits.conf`	`--include=system`
内核参数列表	`sysctl.conf`	`--include=system`
socket 统计信息（即 ss 的命令结果）	`ss.txt`	`--include=system`

13.12 TiSpark

13.12.1 TiSpark 用户指南

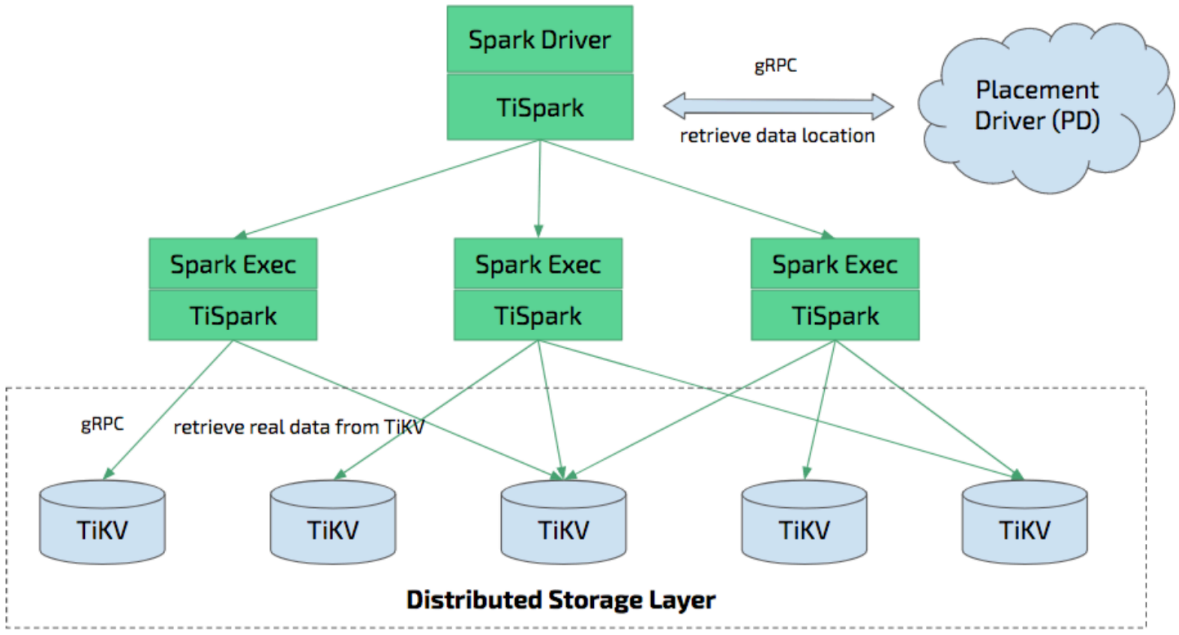


图 295: TiSpark 架构

13.12.1.1 TiSpark vs TiFlash

TiSpark 是 PingCAP 为解决用户复杂 OLAP 需求而推出的产品。它借助 Spark 平台，同时融合 TiKV 分布式集群的优势，和 TiDB 一起为用户一站式解决 HTAP (Hybrid Transactional/Analytical Processing) 的需求。

TiFlash 也是一个解决 HTAP 需求的产品。TiFlash 和 TiSpark 都允许使用多个主机在 OLTP 数据上执行 OLAP 查询。TiFlash 是列式存储，这提供了更高效的分析查询。TiFlash 和 TiSpark 可以同时使用。

13.12.1.2 TiSpark 是什么

TiSpark 依赖于 TiKV 集群和 Placement Driver (PD)，也需要你搭建一个 Spark 集群。本文简单介绍如何部署和使用 TiSpark。本文假设你对 Spark 有基本认知。你可以参阅 Apache Spark 官网了解 Spark 的相关信息。

TiSpark 深度整合了 Spark Catalyst 引擎，可以对计算进行精确的控制，使 Spark 能够高效地读取 TiKV 中的数据。TiSpark 还提供索引支持，帮助实现高速点查。

TiSpark 通过将计算下推到 TiKV 中提升了数据查询的效率，减少了 Spark SQL 需要处理的数据量，通过 TiDB 内置的统计信息选择最优的查询计划。

TiSpark 和 TiDB 可以让用户无需创建和维护 ETL，直接在同一个平台上进行事务和分析两种任务。这简化了系统架构，降低了运维成本。

用户可以在 TiDB 上使用 Spark 生态圈的多种工具进行数据处理，例如：

- TiSpark：数据分析和 ETL。
- TiKV：数据检索。
- 调度系统：生成报表。

除此之外，TiSpark 还提供了分布式写入 TiKV 的功能。与使用 Spark 结合 JDBC 写入 TiDB 的方式相比，分布式写入 TiKV 能够实现事务（要么全部数据写入成功，要么全部都写入失败）。

> **警告：**
>
> 由于 TiSpark 直接访问 TiKV，所以 TiDB Server 使用的访问控制机制并不适用于 TiSpark。TiSpark v2.5.0 及以上版本实现了部分鉴权与授权功能，具体信息请参考安全。

13.12.1.3 版本要求

- TiSpark 支持 Spark 2.3 或以上版本。
- TiSpark 需要 JDK 1.8 以及 Scala 2.11/2.12 版本。
- TiSpark 可以运行在任何 Spark 模式上，如 YARN、Mesos 以及 Standalone。

13.12.1.4 推荐配置

> **警告：**
>
> 此文所描述的使用 TiUP 部署 TiSpark 的方式已被废弃。

TiSpark 作为 Spark 的 TiDB 连接器，需要 Spark 集群的支持。本文仅提供部署 Spark 的参考建议，对于硬件以及部署细节，请参考 Spark 官方文档。

对于独立部署的 Spark 集群，可以参考如下建议配置：

- 建议为 Spark 分配 32G 以上的内存，并为操作系统和缓存保留至少 25% 的内存。
- 建议每台机器至少为 Spark 分配 8 到 16 核 CPU。起初，你可以设定将所有 CPU 核分配给 Spark。

可以参考如下的 spark-env.sh 配置文件：

```
SPARK_EXECUTOR_MEMORY = 32g
SPARK_WORKER_MEMORY = 32g
SPARK_WORKER_CORES = 8
```

13.12.1.5 获取 TiSpark

TiSpark 是 Spark 的第三方 jar 包，提供读写 TiKV 的能力。

13.12.1.5.1 获取 mysql-connector-j

由于 GPL 许可证的限制，TiSpark 不再提供 `mysql-connector-java` 的依赖。以下版本将不再包含 `mysql-connector-java`：

- TiSpark > 3.0.1
- TiSpark > 2.5.1 (TiSpark 2.5.x)
- TiSpark > 2.4.3 (TiSpark 2.4.x)

在使用 TiSpark 写入与鉴权时，仍需要 `mysql-connector-java` 依赖，因此你需要手动下载，并使用以下方式引入：

- 将 `mysql-connector-java` 放入 Spark jars 包中。
- 在你提交 Spark 任务时，引入 `mysql-connector-java`，详见以下示例:

```
spark-submit --jars tispark-assembly-3.0_2.12-3.1.0-SNAPSHOT.jar,mysql-connector-java-8.0.29.jar
```

13.12.1.5.2 选择 TiSpark 版本

你可以根据 TiDB 和 Spark 版本选择相应的 TiSpark 版本。

TiSpark 版本	TiDB、TiKV、PD 版本	Spark 版本	Scala 版本
2.4.x-scala_2.11	5.x, 4.x	2.3.x, 2.4.x	2.11
2.4.x-scala_2.12	5.x, 4.x	2.4.x	2.12
2.5.x	5.x, 4.x	3.0.x, 3.1.x	2.12
3.0.x	5.x, 4.x	3.0.x, 3.1.x, 3.2.x	2.12
3.1.x	6.x, 5.x, 4.x	3.0.x, 3.1.x, 3.2.x, 3.3.x	2.12

13.12.1.6 获取 TiSpark jar 包

你能用以下方式获取 jar 包：

- 从 maven 中央仓库获取，你可以搜索 `pingcap` 关键词。
- 从 TiSpark releases 获取。
- 通过以下步骤从源码构建：

1. 下载 TiSpark 源码：

```
git clone https://github.com/pingcap/tispark.git
cd tisapark
```

2. 在 TiSpark 根目录运行如下命令：

```
// add -Dmaven.test.skip=true to skip the tests
mvn clean install -Dmaven.test.skip=true
// or you can add properties to specify spark version
mvn clean install -Dmaven.test.skip=true -Pspark3.2.1
```

> 注意：
> 目前，你只能使用 java8 构架 TiSpark。运行 `mvn -version` 来检查 java 版本。

13.12.1.6.1 TiSpark jar 包的 artifact ID

注意不同版本的 TiSpark artifact ID 也不同：

TiSpark 版本	Artifact ID
2.4.x-${scala_version}, 2.5.0	tispark-assembly
2.5.1	tispark-assembly-${spark_version}
3.0.x, 3.1.x	tispark-assembly-${spark_version}-${scala_version}

13.12.1.7 快速开始

本章节将以 spark-shell 为例，介绍如何使用 TiSpark。请保证您已下载 Spark。

13.12.1.7.1 启动 spark-shell

在 `spark-defaults.conf` 中添加如下配置：

```
spark.sql.extensions  org.apache.spark.sql.TiExtensions
spark.tispark.pd.addresses  ${your_pd_adress}
spark.sql.catalog.tidb_catalog  org.apache.spark.sql.catalyst.catalog.TiCatalog
spark.sql.catalog.tidb_catalog.pd.addresses  ${your_pd_adress}
```

启动 spark-shell：

```
spark-shell --jars tispark-assembly-{version}.jar
```

13.12.1.7.2 获取 TiSpark 版本

在 spark-shell 中运行如下命令，可获取 TiSpark 版本信息：

```
spark.sql("select ti_version()").collect
```

13.12.1.7.3 使用 TiSpark 读取数据

可以通过 Spark SQL 从 TiKV 读取数据：

```
spark.sql("use tidb_catalog")
spark.sql("select count(*) from ${database}.${table}").show
```

13.12.1.7.4 使用 TiSpark 写入数据

通过 Spark DataSource API，可以在保证 ACID 前提下写入数据到 TiKV：

```
val tidbOptions: Map[String, String] = Map(
  "tidb.addr" -> "127.0.0.1",
  "tidb.password" -> "",
  "tidb.port" -> "4000",
  "tidb.user" -> "root"
)

val customerDF = spark.sql("select * from customer limit 100000")

customerDF.write
.format("tidb")
.option("database", "tpch_test")
.option("table", "cust_test_select")
.options(tidbOptions)
.mode("append")
.save()
```

详见 Data Source API User Guide。

在 TiSpark 3.1 之后，你还能通过 Spark SQL 写入 TiSpark 3.1。详见 Insert SQL。

13.12.1.7.5 通过 JDBC 数据源写入数据

你同样可以在不使用 TiSpark 的情况下使用 Spark JDBC 数据源写入 TiDB。

由于这超过了 TiSpark 的范畴，本文仅简单提供示例，详情请参考 JDBC To Other Databases。

```
import org.apache.spark.sql.execution.datasources.jdbc.JDBCOptions

val customer = spark.sql("select * from customer limit 100000")
// 为了平衡各节点以及提高并发数，你可以将数据源重新分区
```

```
val df = customer.repartition(32)
df.write
.mode(saveMode = "append")
.format("jdbc")
.option("driver", "com.mysql.jdbc.Driver")
// 替换为你的主机名和端口地址，并确保开启了重写批处理
.option("url", "jdbc:mysql://127.0.0.1:4000/test?rewriteBatchedStatements=true")
.option("useSSL", "false")
// 作为测试，建议设置为 150
.option(JDBCOptions.JDBC_BATCH_INSERT_SIZE, 150)
.option("dbtable", s"cust_test_select") // 数据库名和表名
.option("isolationLevel", "NONE") // 如果需要写入较大 Dataframe，推荐将 isolationLevel 设置为
    ↪ NONE
.option("user", "root") // TiDB 用户名
.save()
```

为了避免大事务导致 OOM 以及 TiDB 报 `ISOLATION LEVEL does not support` 错误（TiDB 目前仅支持 `REPEATABLE-READ`），推荐设置 `isolationLevel` 为 `NONE`。

13.12.1.7.6 使用 TiSpark 删除数据

可以使用 Spark SQL 删除 TiKV 数据：

```
spark.sql("use tidb_catalog")
spark.sql("delete from ${database}.${table} where xxx")
```

详情请参考 delete feature。

13.12.1.7.7 与其他数据源一起使用

你可以使用多个 catalog 从不同数据源读取数据：

```
// 从 Hive 读取
spark.sql("select * from spark_catalog.default.t").show

// Join Hive 表和 TiDB 表
spark.sql("select t1.id,t2.id from spark_catalog.default.t t1 left join tidb_catalog.test.t t2").
    ↪ show
```

13.12.1.8 TiSpark 配置

可以将如下参数配置在 `spark-defaults.conf` 中，也可以参考 Spark 其他配置，用同样的方式传入。

Key	Default value	Description
spark.tispark.pd.addresses	127.0.0.1:2379	PD 集群的地址，通过逗号分隔。
spark.tispark.grpc.framesize	2147483648	gRPC 的最大回复大小，单位 bytes（默认 2G）。
spark.tispark.grpc.timeout_in_sec	10	gRPC 超时时间，单位秒。
spark.tispark.plan.allow_agg_pushdown	true	是否运行聚合下推（为了避免 TiKV 节点繁忙）。
spark.tispark.plan.allow_index_read	true	是否在执行计划中开启 index（可能导致 TiKV 压力过大）。
spark.tispark.index.scan_batch_size	20000	在 index scan 的一次 batch 中 row keys 的数量。
spark.tispark.index.scan_concurrency	5	在 index scan 中获取 row keys 的最大线程数（每个 JVM 的所有任务共享）。
spark.tispark.table.scan_concurrency	512	在 table scan 中的最大线程数（每个 JVM 的所有任务共享）。

Key	Default value	Description
spark.tispark.request.command.priority	Low	可选项有 Low、Normal和High。修改配置会影响影响 TiKV 的资源分配。建议使用 Low，因为 OLTP 的负载不会被影响。
spark.tispark.coprocessor.codec_format	chblock	Coprocessor 的编码格式。可选项为 default、chblock 和 chunk。
spark.tispark.coprocessor.streaming	false	是否在获取响应时使用 streaming（实验性质）。
spark.tispark.plan.unsupported_pushdown_exprs		表达式清单，多个表达式用逗号分隔。为了防止老版本的 TiKV 不支持某些表达式，你可以禁止下推它们。
spark.tispark.plan.downgrade.index_threshold	1000000	如果 index scan 请求的范围超过了此限制，该 Region 请求会被降级为 table scan。降级默认关闭。
spark.tispark.show_rowid	false	是否在 row_id 存在时显示它。
spark.tispark.db_prefix		TiDB 数据库的前缀。该配置可用于在 TiSpark 2.4 中区分同名的 TiDB 和 Hive 数据库。
spark.tispark.request.isolation.level	SI	是否为底层的 TiKV 解锁。当你使用 “RC” 级别，你将忽略锁，得到比 tso 更小的最新版本数据。当你使用 “SI” 级别，你将进行解锁并根据该锁对应的事务提交与否获取数据。

Key	Default value	Description
spark.tispark.coprocessor.chunk_batch_size	1024	从 coprocessor 获取的一个 batch 的 Row 数量。
spark.tispark.isolation_read_engines	tikv	TiSpark 读引擎，多个引擎使用逗号分隔。未配置的存储引擎不会被读取。
spark.tispark.stale_read	optional	stale read 时间戳。详情请参考 stale read。
spark.tispark.tikv.tls_enable	false	是否开启 TiSpark TLS。
spark.tispark.tikv.trust_cert_collection		TiKV Client 的受信任根证书，用于验证 PD 证书。如 `/home/tispark/config/root.pem`，该文件需包含 X.509 证书。
spark.tispark.tikv.key_cert_chain		TiKV Client 的 X.509 格式的客户端证书链。如 `/home/tispark/config/client.pem`。
spark.tispark.tikv.key_file		TiKV Client 的 PKCS#8 私钥文件。如 `/home/tispark/client_pkcs8.key`。
spark.tispark.tikv.jks_enable	false	是否使用 `JAVA key store` 而不是 X.509 证书。

Key	Default value	Description
spark.tispark.tikv.jks_trust_path		TiKV Client JKS 格式的受信任根证书。由 keytool 生成，如 /home/tispark/config/tikv-truststore。
spark.tispark.tikv.jks_trust_password		spark.tispark.tikv.jks_trust_path 的密码。
spark.tispark.tikv.jks_key_path		TiKV Client JKS 格式的客户端证书。由 keytool 生成，如 /home/tispark/config/tikv-clientstore。
spark.tispark.tikv.jks_key_password		spark.tispark.tikv.jks_key_path 的密码。
spark.tispark.jdbc.tls_enable	false	是否开启 JDBC connector TLS。
spark.tispark.jdbc.server_cert_store		JDBC 的受信任根证书。由 keytool 生产的 Java keystore (JKS) 格式的证书。如 /home/tispark/config/jdbc-truststore。默认值为 “”，表示 TiSpark 不会校验 TiDB 服务端。
spark.tispark.jdbc.server_cert_password		spark.tispark.jdbc.server_cert_store 的密码。
spark.tispark.jdbc.client_cert_store		JDBC 的 PKCS#12 格式的客户端证书。这是由 keytool 生成的 JKS 格式证书。如 /home/tispark/config/jdbc-clientstore。默认值为 “”，表示 TiDB 不会校验 TiSpark。

Key	Default value	Description
spark.tispark.jdbc.client_cert_password		spark.tispark.jdbc.client_cert_store 的密码。
spark.tispark.tikv.tls_reload_interval	10s	重载证书的时间间隔。默认值为 10s。
spark.tispark.tikv.conn_recycle_time	60s	清理 TiKV 失效连接的时间间隔。默认时间为 60s。当重载证书开启时此配置才会生效。
spark.tispark.host_mapping		路由映射配置。用于配置公有 IP 地址和私有 IP 地址的映射。当 TiDB 在私有网络上运行时，你可以将一系列内部 IP 地址映射到公网 IP 地址以便 Spark 集群访问。其格式为 {Intranet IP1}:{Public IP1};{Intranet IP2}:{Public IP2}，例如 192.168.0.2:8.8.8.8;192.168.0.3:9.9.9.9。
spark.tispark.new_collation_enable		当 TiDB 开启 new collation，推荐将此配置设为true。当 TiDB 关闭 new collation，推荐将此配置设置为 false。在未配置的情况下，TiSpark 会依据 TiDB 版本自动配置 new collation。其规则为：当 TiDB 版本大于等于 v6.0.0 时为 true；否则为 false。
spark.tispark.replica_read	leader	读取副本的类型。可选值为 leader、follower、learner。可以同时指定多个类型，TiSpark 会根据顺序选择。
spark.tispark.replica_read.label		目标 TiKV 节点的标签。格式为 label_x=value_x,label_y=value_y，各项之间为“逻辑与”的关系。

13.12.1.8.1 TLS 配置

TiSpark TLS 分为两部分：TiKV Client TLS 以及 JDBC connector TLS。前者用于创建和 TiKV/PD 的 TLS 连接，后者用于创建与 TiDB 的 TLS 连接。

当配置 TiKV Client TLS 时，你需要以 X.509 格式的证书配置 tikv.trust_cert_collection、tikv.key_cert_chain 和 tikv.key_file；或者以 JKS 格式的证书配置 tikv.jks_enable，tikv.jks_trust_path 和 tikv.jks_key_path。

当配置 JDBC connector TLS 时，你需要配置 spark.tispark.jdbc.tls_enable，而 jdbc.server_cert_store 和 jdbc.client_cert_store 则是可选的。

TiSpark 目前仅持 TLS 1.2 and TLS 1.3。

- 如下是使用 X.509 证书配置 TiKV Client TLS 的例子：

```
spark.tispark.tikv.tls_enable                                    true
spark.tispark.tikv.trust_cert_collection                         /home/tispark/root.pem
spark.tispark.tikv.key_cert_chain                                /home/tispark/client.pem
spark.tispark.tikv.key_file                                      /home/tispark/client.key
```

- 如下是使用 JKS 配置 TiKV Client TLS 的例子：

```
spark.tispark.tikv.tls_enable                                    true
spark.tispark.tikv.jks_enable                                    true
spark.tispark.tikv.jks_key_path                                  /home/tispark/config/tikv-
    ↪ truststore
spark.tispark.tikv.jks_key_password                              tikv_trustore_password
spark.tispark.tikv.jks_trust_path                                /home/tispark/config/tikv-
    ↪ clientstore
spark.tispark.tikv.jks_trust_password                            tikv_clientstore_password
```

当你同时配置 JKS 和 X.509 证书时，JKS 优先级更高。因此，当你只想使用普通的 pem 证书时，不要同时设置 `spark.tispark.tikv.jks_enable=true`。

- 下面是一个配置 JDBC connector TLS 的例子：

```
spark.tispark.jdbc.tls_enable                                    true
spark.tispark.jdbc.server_cert_store                             /home/tispark/jdbc-truststore
spark.tispark.jdbc.server_cert_password                          jdbc_truststore_password
spark.tispark.jdbc.client_cert_store                             /home/tispark/jdbc-clientstore
spark.tispark.jdbc.client_cert_password                          jdbc_clientstore_password
```

- 对于如何开启 TiDB TLS，请参考Enable TLS between TiDB Clients and Servers。
- 对于如何生成 JAVA key store，请参考 Connecting Securely Using SSL。

13.12.1.8.2 时区配置

使用 `-Duser.timezone` 系统参数来配置时区（比如 `-Duser.timezone=GMT-7`）。时区会影响 `Timestamp` 数据类型。

请不要使用 `spark.sql.session.timeZone`。

13.12.1.9 特性

TiSpark 的主要特性如下：

特性支持	TiSpark 2.4.x	TiSpark 2.5.x	TiSpark 3.0.x	TiSpark 3.1.x
SQL select without tidb_catalog	✓	✓		
SQL select with tidb_catalog		✓	✓	✓
DataFrame append	✓	✓	✓	✓
DataFrame reads	✓	✓	✓	✓
SQL show databases	✓	✓	✓	✓
SQL show tables	✓	✓	✓	✓
SQL auth		✓	✓	✓
SQL delete			✓	✓
SQL insert				✓
TLS			✓	✓
DataFrame auth				✓

13.12.1.9.1 Expression index 支持

TiDB v5.0 开始支持expression index。

TiSpark 目前支持从 `expression index` 的表中获取数据，但 `expression index` 不会被 TiSpark 执行计划使用。

13.12.1.9.2 TiFlash 支持

TiSpark 能够通过配置 `spark.tispark.isolation_read_engines` 从 TiFlash 读取数据。

13.12.1.9.3 分区表支持

读分区表

TiSpark 目前支持读取 range 与 hash 分区表。

TiSpark 目前不支持 `partition table` 语法 `select col_name from table_name partition(partition_name)`。但是，你仍可以使用 `where` 条件过滤分区。

TiSpark 会根据分区类型、分区表达式以及具体 SQL 决定是否进行分区裁剪。目前，TiSpark 仅支持在 range 分区下，且在下列任一条件下进行分区裁剪：

- 列表达式，如 `partition by col1`。
- 形如 `YEAR($col)` 的 year 函数，其中 col 为列名且类型为 datetime、date 或能被解析为 datetime、date 的 string 字面量。
- 形如 `TO_DAYS($col)` 的 to_days 函数，其中 col 为列名且类型为 datetime、 date 或能被解析为 datetime、 date 的 string 字面量。

如果分区裁剪未被应用，TiSpark 将会读取所有分区表。

写分区表

目前, TiSpark 仅支持写入 range 与 hash 分区表，且需满足以下任一条件：

- 列表达式，如 `partition by col1`。

- 形如 YEAR($col) 的 year 函数，其中 col 为列名且类型为 datetime、date 或能被解析为 datetime、date 的 string 字面量。

> 注意：
>
> 目前，TiSpark 只支持在 utf8mb4_bin 字符集下写入分区表。

有两种方式写入分区表：

- 使用支持 replace 和 append 语义的 Datasource API 写入分区表。
- 使用 Spark SQL 删除语句。

13.12.1.9.4 安全

从 TiSpark v2.5.0 起，你可以通过 TiDB 对 TiSpark 进行鉴权与授权。

该功能默认关闭。要开启该功能，请在 Spark 配置文件 spark-defaults.conf 中添加以下配置项：

```
// 开启鉴权与授权功能
spark.sql.auth.enable true

// 配置 TiDB 信息
spark.sql.tidb.addr $your_tidb_server_address
spark.sql.tidb.port $your_tidb_server_port
spark.sql.tidb.user $your_tidb_server_user
spark.sql.tidb.password $your_tidb_server_password
```

更多详细信息，请参考 TiSpark 鉴权与授权指南。

13.12.1.9.5 其他特性

- 下推
- TiSpark 删除数据
- 历史读
- TiSpark with multiple catalogs
- TiSpark TLS
- TiSpark 执行计划

13.12.1.10 统计信息

TiSpark 可以使用 TiDB 的统计信息：

- 选择代价最低的索引访问
- 估算数据大小以决定是否进行广播优化

如果你希望 TiSpark 使用统计信息支持，需要确保所涉及的表已经被分析。参考统计信息简介了解如何进行表分析。

从 TiSpark 2.0 开始，统计信息将会默认被读取。

13.12.1.11 FAQ

详情请参考 TiSpark FAQ。

13.13 sync-diff-inspector

13.13.1 sync-diff-inspector 用户文档

sync-diff-inspector 是一个用于校验 MySQL / TiDB 中两份数据是否一致的工具。该工具提供了修复数据的功能 (适用于修复少量不一致的数据)。

主要功能：

- 对比表结构和数据
- 如果数据不一致，则生成用于修复数据的 SQL 语句
- 支持不同库名或表名的数据校验
- 支持分库分表场景下的数据校验
- 支持TiDB 主从集群的数据校验
- 支持从 TiDB DM 拉取配置的数据校验

你可通过以下方式下载 sync-diff-inspector：

- Binary 包。sync-diff-inspector 的安装包位于 TiDB 离线工具包中。下载方式，请参考TiDB 工具下载。
- Docker 镜像。执行以下命令进行下载：

```
docker pull pingcap/tidb-enterprise-tools:nightly
```

13.13.1.1 sync-diff-inspector 的使用限制

- 对于 MySQL 和 TiDB 之间的数据同步不支持在线校验，需要保证上下游校验的表中没有数据写入，或者保证某个范围内的数据不再变更，通过配置 `range` 来校验这个范围内的数据。
- FLOAT、 DOUBLE 等浮点数类型在 TiDB 和 MySQL 中的实现方式不同，在计算 checksum 时会分别取 6 位和 15 位有效数字。如果不使用该特性，需要设置 `ignore-columns` 忽略这些列的检查。
- 支持对不包含主键或者唯一索引的表进行校验，但是如果数据不一致，生成的用于修复的 SQL 可能无法正确修复数据。

13.13.1.2 sync-diff-inspector 所需的数据库权限

sync-diff-inspector 需要获取表结构信息、查询数据，需要的数据库权限如下：

- 上游数据库
  - SELECT（查数据进行对比）
  - SHOW_DATABASES（查看库名）
  - RELOAD（查看表结构）
- 下游数据库
  - SELECT（查数据进行对比）
  - SHOW_DATABASES（查看库名）
  - RELOAD（查看表结构）

13.13.1.3 配置文件说明

sync-diff-inspector 的配置总共分为五个部分：

- Global config：通用配置，包括校验的线程数量、是否输出修复 SQL、是否比对数据、是否跳过校验上游或下游不存在的表等。
- Datasource config：配置上下游数据库实例。
- Routes：上游多表名通过正则匹配下游单表名的规则。（可选）
- Task config：配置校验哪些表，如果有的表在上下游有一定的映射关系或者有一些特殊要求，则需要对指定的表进行配置。
- Table config：对具体表的特殊配置，例如指定范围、忽略的列等等。（可选）

下面是一个完整配置文件的说明：

- 提示：配置名后带 s 的配置项允许拥有多个配置值，因此需要使用方括号 [] 来包含配置值。

```
### Diff Configuration.

########################## Global config ##########################

### 检查数据的线程数量，上下游数据库的连接数会略大于该值
check-thread-count = 4

### 如果开启，若表存在不一致，则输出用于修复的 SQL 语句。
export-fix-sql = true

### 只对比表结构而不对比数据
check-struct-only = false

### 如果开启，会跳过校验上游或下游不存在的表。
```

```
skip-non-existing-table = false

########################### Datasource config ########################
[data-sources]
[data-sources.mysql1] # mysql1 是该数据库实例唯一标识的自定义 id，用于下面 task.source-instances/
    ↪ task.target-instance 中
    host = "127.0.0.1"
    port = 3306
    user = "root"
    password = "" # 设置连接上游数据库的密码，可为明文或 Base64 编码。

    # ( 可选 ) 使用映射规则来匹配上游多个分表，其中 rule1 和 rule2 在下面 Routes 配置栏中定义
    route-rules = ["rule1", "rule2"]

[data-sources.tidb0]
    host = "127.0.0.1"
    port = 4000
    user = "root"
    password = "" # 设置连接下游数据库的密码，可为明文或 Base64 编码。

    # ( 可选 ) 使用 TLS 连接 TiDB
    # security.ca-path = ".../ca.crt"
    # security.cert-path = ".../cert.crt"
    # security.key-path = ".../key.crt"

    # ( 可选 ) 使用 TiDB 的 snapshot 功能，如果开启的话会使用历史数据进行对比
    # snapshot = "386902609362944000"
    # 当 snapshot 设置为 "auto" 时，使用 TiCDC 在上下游的同步时间点，具体参考 <https://github.com
        ↪ /pingcap/tidb-tools/issues/663>
    # snapshot = "auto"

############################# Routes #############################
### 如果需要对比大量的不同库名或者表名的表的数据，或者用于校验上游多个分表与下游总表的数据，
    ↪ 可以通过 table-rule 来设置映射关系
### 可以只配置 schema 或者 table 的映射关系，也可以都配置
[routes]
[routes.rule1] # rule1 是该配置的唯一标识的自定义 id，用于上面 data-sources.route-rules 中
schema-pattern = "test_*"      # 匹配数据源的库名，支持通配符 "*" 和 "?"
table-pattern = "t_*"          # 匹配数据源的表名，支持通配符 "*" 和 "?"
target-schema = "test"         # 目标库名
target-table = "t" # 目标表名

[routes.rule2]
schema-pattern = "test2_*"      # 匹配数据源的库名，支持通配符 "*" 和 "?"
table-pattern = "t2_*"          # 匹配数据源的表名，支持通配符 "*" 和 "?"
```

```
target-schema = "test2"         # 目标库名
target-table = "t2" # 目标表名

########################### Task config ###########################
### 配置需要对比的*目标数据库*中的表
[task]
    # output-dir 会保存如下信息
    # 1 sql：检查出错误后生成的修复 SQL 文件，并且一个 chunk 对应一个文件
    # 2 log：sync-diff.log 保存日志信息
    # 3 summary：summary.txt 保存总结
    # 4 checkpoint：a dir 保存断点续传信息
    output-dir = "./output"

    # 上游数据库，内容是 data-sources 声明的唯一标识 id
    source-instances = ["mysql1"]

    # 下游数据库，内容是 data-sources 声明的唯一标识 id
    target-instance = "tidb0"

    # 需要比对的下游数据库的表，每个表需要包含数据库名和表名，两者由 `.` 隔开
    # 使用 ? 来匹配任意一个字符；使用 * 来匹配任意；详细匹配规则参考 golang regexp pkg: https://
        ↪ github.com/google/re2/wiki/Syntax
    target-check-tables = ["schema*.table*", "!c.*", "test2.t2"]

    # ( 可选 ) 对部分表的额外配置，其中 config1 在下面 Table config 配置栏中定义
    target-configs = ["config1"]

########################### Table config ###########################
### 对部分表进行特殊的配置，配置的表必须包含在 task.target-check-tables 中
[table-configs.config1] # config1 是该配置的唯一标识自定义 id，用于上面 task.target-configs 中
### 目标表名称，可以使用正则来匹配多个表，但不允许存在一个表同时被多个特殊配置匹配。
target-tables = ["schema*.test*", "test2.t2"]
# ( 可选 ) 指定检查的数据的范围，需要符合 sql 中 where 条件的语法
range = "age > 10 AND age < 20"
# ( 可选 ) 指定用于划分 chunk 的列，如果不配置该项，sync-diff-inspector 会选取一些合适的列 (
    ↪ 主键 / 唯一键 / 索引 )
index-fields = ["col1","col2"]
# ( 可选 ) 忽略某些列的检查，例如 sync-diff-inspector 目前还不支持的一些类型 ( json，bit，blob 等
    ↪ ) ，
### 或者是浮点类型数据在 TiDB 和 MySQL 中的表现可能存在差异，可以使用 ignore-columns
    ↪ 忽略检查这些列
ignore-columns = ["",""]
# ( 可选 ) 指定划分该表的 chunk 的大小，若不指定可以删去或者将其配置为 0。
chunk-size = 0
# ( 可选 ) 指定该表的 collation，若不指定可以删去或者将其配置为空字符串。
```

```
collation = ""
```

13.13.1.4 运行 sync-diff-inspector

执行如下命令：

```
./sync_diff_inspector --config=./config.toml
```

该命令最终会在 `config.toml` 中的 `output-dir` 输出目录输出本次比对的检查报告 `summary.txt` 和日志 `sync_diff.log`。在输出目录下还会生成由 `config.toml` 文件内容哈希值命名的文件夹，该文件夹下包括断点续传 checkpoint 结点信息以及数据存在不一致时生成的 SQL 修复数据。

13.13.1.4.1 前台输出

sync-diff-inspector 在执行过程中会往 `stdout` 发送进度信息。进度信息包括表的结构比较结果、表的数据比较结果以及进度条。

> **注意：**
>
> 为了达成显示效果，请保持显示窗口宽度在 80 字符以上。

```
A total of 2 tables need to be compared

Comparing the table structure of ``sbtest`.`sbtest96`` ... equivalent
Comparing the table structure of ``sbtest`.`sbtest99`` ... equivalent
Comparing the table data of ``sbtest`.`sbtest96`` ... failure
Comparing the table data of ``sbtest`.`sbtest99`` ...
_____________________________________________________________________________
Progress [============================================================>--] 98% 193/200
```

```
A total of 2 tables need to be compared

Comparing the table structure of ``sbtest`.`sbtest96`` ... equivalent
Comparing the table structure of ``sbtest`.`sbtest99`` ... equivalent
Comparing the table data of ``sbtest`.`sbtest96`` ... failure
Comparing the table data of ``sbtest`.`sbtest99`` ... failure
_____________________________________________________________________________
Progress [============================================================>] 100% 0/0
The data of `sbtest`.`sbtest99` is not equal
The data of `sbtest`.`sbtest96` is not equal

The rest of tables are all equal.
```

```
A total of 2 tables have been compared, 0 tables finished, 2 tables failed, 0 tables skipped.
The patch file has been generated in
        'output/fix-on-tidb2/'
You can view the comparison details through 'output/sync_diff.log'
```

13.13.1.4.2 输出文件

输出文件目录结构如下：

```
output/
|-- checkpoint # 保存断点续传信息
| |-- bbfec8cc8d1f58a5800e63aa73e5 # config hash 占位文件，标识该输出目录（output/）
    ↪ 对应的配置文件
| |-- DO_NOT_EDIT_THIS_DIR
| └-- sync_diff_checkpoints.pb # 断点续传信息
|
|-- fix-on-target # 保存用于修复不一致的 SQL 文件
| |-- xxx.sql
| |-- xxx.sql
| └-- xxx.sql
|
|-- summary.txt # 保存校验结果的总结
└-- sync_diff.log # 保存 sync-diff-inspector 执行过程中输出的日志信息
```

日志

sync-diff-inspector 的日志存放在 ${output}/sync_diff.log 中，其中 ${output} 是 config.toml 文件中 output-dir 的值。

校验进度

sync-diff-inspector 会在运行时定期（间隔 10s）输出校验进度到 checkpoint 中 (位于 ${output}/checkpoint/
↪ sync_diff_checkpoints.pb 中，其中 ${output} 是 config.toml 文件中 output-dir 的值。

校验结果

当校验结束时，sync-diff-inspector 会输出一份校验报告，位于 ${output}/summary.txt 中，其中 ${output} 是 config.toml 文件中 output-dir 的值。

```
+--------------------+---------+--------------------+----------------+---------+-----------+
|        TABLE       | RESULT  | STRUCTURE EQUALITY | DATA DIFF ROWS | UPCOUNT | DOWNCOUNT |
+--------------------+---------+--------------------+----------------+---------+-----------+
| `sbtest`.`sbtest99` | succeed | true               | +97/-97        |  999999 |    999999 |
| `sbtest`.`sbtest96` | succeed | true               | +0/-101        |  999999 |   1000100 |
| `sbtest`.`sbtest97` | skipped | false              | +999999/-0     |  999999 |         0 |
+--------------------+---------+--------------------+----------------+---------+-----------+
Time Cost: 16.75370462s
Average Speed: 113.277149MB/s
```

- TABLE：该列表示对应的数据库及表名
- RESULT：校验是否完成。如果设置了 skip-non-existing-table = true，对于上游或下游不存在的表，该列的值将为 skipped
- STRUCTURE EQUALITY：表结构是否相同
- DATA DIFF ROWS：即 rowAdd/rowDelete，表示该表修复需要增加/删除的行数

SQL 修复

校验过程中遇到不同的行，会生成修复数据的 SQL 语句。一个 chunk 如果出现数据不一致，就会生成一个以 chunk.Index 命名的 SQL 文件。文件位于 ${output}/fix-on-${instance} 文件夹下。其中 ${instance} 为 config.toml 中 task.target-instance 的值。

一个 SQL 文件会包含该 chunk 的所属表以及表示的范围信息。对每个修复 SQL 语句，有三种情况：

- 下游数据库缺失行，则是 REPLACE 语句
- 下游数据库冗余行，则是 DELETE 语句
- 下游数据库行部分数据不一致，则是 REPLACE 语句，但会在 SQL 文件中通过注释的方法标明不同的列

```
-- table: sbtest.sbtest99
-- range in sequence: (3690708) < (id) <= (3720581)
/*
  DIFF COLUMNS │   `K`   │                `C`                 │               `PAD`
╪═══════════╪═══════╪═══════════════════════════════════╪═══════════════════════════════════
    ↪
  source data  │ 2501808 │ 'hello'                           │ 'world'
╪═══════════╪═══════╪═══════════════════════════════════╪═══════════════════════════════════
    ↪
  target data  │ 5003616 │ '0709824117-9809973320-4456050422' │
      ↪ '1714066100-7057807621-1425865505'
╪═══════════╪═══════╪═══════════════════════════════════╪═══════════════════════════════════
    ↪
*/
REPLACE INTO `sbtest`.`sbtest99`(`id`,`k`,`c`,`pad`) VALUES (3700000,2501808,'hello','world');
```

13.13.1.5 注意事项

- sync-diff-inspector 在校验数据时会消耗一定的服务器资源，需要避免在业务高峰期间校验。
- 在数据对比前，需要注意表中的 collation 设置。如果表的主键或唯一键为 varchar 类型，且上下游数据库中 collation 设置不同，可能会因为排序问题导致最终校验结果不正确，需要在 sync-diff-inspector 的配置文件中增加 collation 设置。
- sync-diff-inspector 会优先使用 TiDB 的统计信息来划分 chunk，需要尽量保证统计信息精确，可以在业务空闲期手动执行 analyze table {table_name}。
- table-rule 的规则需要特殊注意，例如设置了 schema-pattern="test1"，table-pattern = "t_1"，target-schema="test2"，target-table = "t_2"，会对比 source 中的表 test1.t_1 和 target 中的表 test2.t_2。sync-diff-inspector 默认开启 sharding，如果 source 中还有表 test2.t_2，则会把 source 端的表 test1.t_1 和表 test2.t_2 作为 sharding 与 target 中的表 test2.t_2 进行一致性校验。
- 生成的 SQL 文件仅作为修复数据的参考，需要确认后再执行这些 SQL 修复数据。

13.13.2 不同库名或表名的数据校验

当你在使用TiDB DM 等同步工具时，可以设置 route-rules 将数据同步到下游指定表中。sync-diff-inspector 通过设置 rules 提供了校验不同库名、表名的表的功能。

下面是一个简单的配置文件说明，要了解完整配置，请参考sync-diff-inspector 用户文档。

```
########################### Datasource config ###########################
[data-sources.mysql1]
    host = "127.0.0.1"
    port = 3306
    user = "root"
    password = ""

    route-rules = ["rule1"]

[data-sources.tidb0]
    host = "127.0.0.1"
    port = 4000
    user = "root"
    password = ""

############################## Routes ############################
[routes.rule1]
schema-pattern = "test_1"      # 匹配数据源的库名，支持通配符 "*" 和 "?"
table-pattern = "t_1"          # 匹配数据源的表名，支持通配符 "*" 和 "?"
target-schema = "test_2"         # 目标库名
target-table = "t_2" # 目标表名
```

使用该配置会对下游的 test_2.t_2 与实例 mysql1 中的 test_1.t_1 进行校验。

如果需要校验大量的不同库名或者表名的表，也可以通过 rules 设置映射关系来简化配置。可以只配置 schema 或者 table 的映射关系，也可以都配置。例如上游库 test_1 中的所有表都同步到了下游的 test_2 库中，可以使用如下配置进行校验：

```
########################### Datasource config ###########################
[data-sources.mysql1]
    host = "127.0.0.1"
    port = 3306
    user = "root"
    password = ""

    route-rules = ["rule1"]

[data-sources.tidb0]
    host = "127.0.0.1"
    port = 4000
    user = "root"
```

```
    password = ""

############################ Routes ##########################
[routes.rule1]
schema-pattern = "test_1"      # 匹配数据源的库名，支持通配符 "*" 和 "?"
table-pattern = "*"            # 匹配数据源的表名，支持通配符 "*" 和 "?"
target-schema = "test_2"       # 目标库名
target-table = "t_2"           # 目标表名
```

13.13.2.1 注意事项

如果上游数据库有 `test_2.t_2` 也会被下游数据库匹配到。

13.13.3 分库分表场景下的数据校验

sync-diff-inspector 支持对分库分表场景进行数据校验。例如有多个 MySQL 实例，当你使用同步工具TiDB DM 同步到一个 TiDB 时，可以使用 sync-diff-inspector 对上下游数据进行校验。

13.13.3.1 使用 datasource config 进行配置

使用 `Datasource config` 对 `table-0` 进行特殊配置，设置对应 `rules`，配置上游表与下游表的映射关系。这种配置方式需要对所有分表进行设置，适合上游分表数量较少，且分表的命名规则没有规律的场景。场景如图所示：

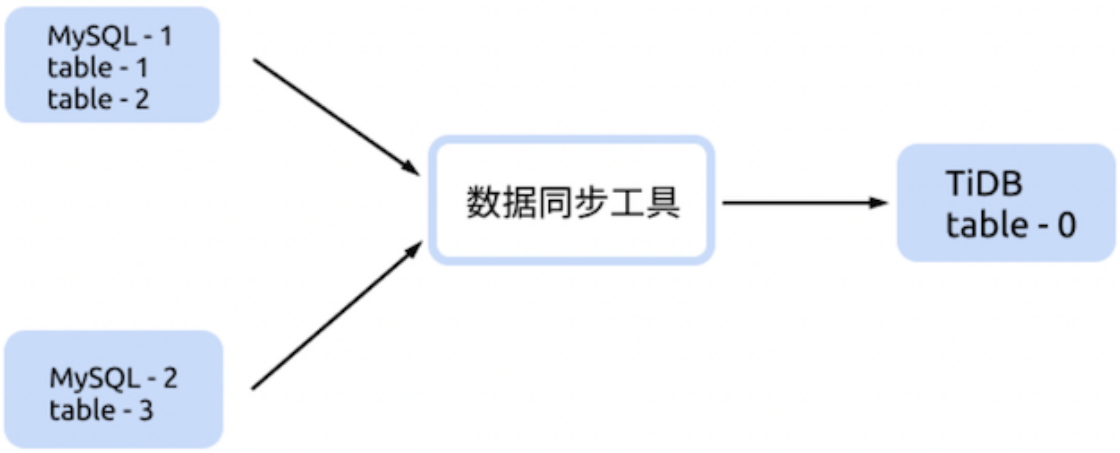


图 296: shard-table-sync-1

sync-diff-inspector 完整的示例配置如下：

```
### Diff Configuration.

############################ Global config ########################

### 检查数据的线程数量，上下游数据库的连接数会略大于该值
check-thread-count = 4

### 如果开启，若表存在不一致，则输出用于修复的 SQL 语句
export-fix-sql = true

### 只对比表结构而不对比数据
check-struct-only = false


########################### Datasource config ########################
[data-sources.mysql1]
    host = "127.0.0.1"
    port = 3306
    user = "root"
    password = ""

    route-rules = ["rule1"]

[data-sources.mysql2]
    host = "127.0.0.1"
    port = 3306
    user = "root"
    password = ""

    route-rules = ["rule2"]

[data-sources.tidb0]
    host = "127.0.0.1"
    port = 4000
    user = "root"
    password = ""

############################## Routes ############################
[routes.rule1]
schema-pattern = "test"        # 匹配数据源的库名，支持通配符 "*" 和 "?"
table-pattern = "table-[1-2]"  # 匹配数据源的表名，支持通配符 "*" 和 "?"
target-schema = "test"         # 目标库名
target-table = "table-0"       # 目标表名
```

```
[routes.rule2]
schema-pattern = "test"      # 匹配数据源的库名，支持通配符 "*" 和 "?"
table-pattern = "table-3"    # 匹配数据源的表名，支持通配符 "*" 和 "?"
target-schema = "test"       # 目标库名
target-table = "table-0"     # 目标表名

########################### Task config ###########################
[task]
    output-dir = "./output"

    source-instances = ["mysql1", "mysql2"]

    target-instance = "tidb0"

    # 需要比对的下游数据库的表，每个表需要包含数据库名和表名，两者由 `.` 隔开
    target-check-tables = ["test.table-0"]
```

当上游分表较多，且所有分表的命名都符合一定的规则时，则可以使用 `table-rules` 进行配置。场景如图所示：

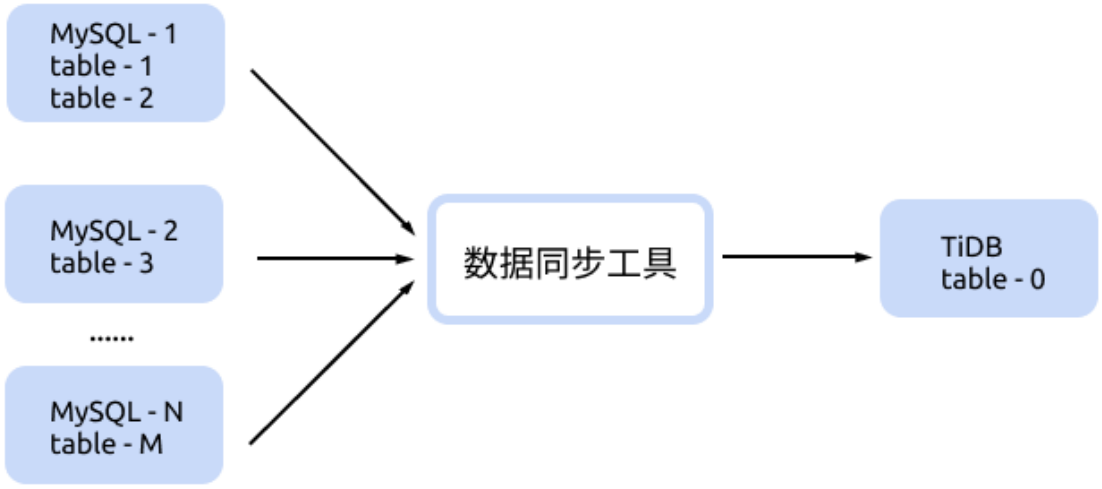


图 297: shard-table-sync-2

sync-diff-inspector 完整的示例配置如下：

```
### Diff Configuration.

########################### Global config ###########################

### 检查数据的线程数量，上下游数据库的连接数会略大于该值
check-thread-count = 4
```

```
### 如果开启，若表存在不一致，则输出用于修复的 SQL 语句
export-fix-sql = true

### 只对比表结构而不对比数据
check-struct-only = false


########################### Datasource config ########################
[data-sources.mysql1]
    host = "127.0.0.1"
    port = 3306
    user = "root"
    password = ""

    route-rules = ["rule1"]

[data-sources.mysql2]
    host = "127.0.0.1"
    port = 3306
    user = "root"
    password = ""

    route-rules = ["rule1"]

[data-sources.tidb0]
    host = "127.0.0.1"
    port = 4000
    user = "root"
    password = ""

############################## Routes ############################
[routes.rule1]
schema-pattern = "test"      # 匹配数据源的库名，支持通配符 "*" 和 "?"
table-pattern = "table-*"    # 匹配数据源的表名，支持通配符 "*" 和 "?"
target-schema = "test"       # 目标库名
target-table = "table-0"     # 目标表名

########################### Task config ##########################
[task]
    output-dir = "./output"

    source-instances = ["mysql1", "mysql2"]

    target-instance = "tidb0"
```

```
    # 需要比对的下游数据库的表，每个表需要包含数据库名和表名，两者由 `.` 隔开
    target-check-tables = ["test.table-0"]
```

13.13.3.2 注意事项

如果上游数据库有 `test.table-0` 也会被下游数据库匹配到。

13.13.4 TiDB 主从集群的数据校验

当你使用 TiCDC 搭建 TiDB 的主从集群时，可能会需要在不停止同步的情况下对上下游的数据进行一致性验证。在普通的同步模式中，TiCDC 只提供数据的最终一致性的保证，而无法确保在同步的过程中数据的一致性。因此，对动态变更的数据进行一致性校验非常困难，为了满足这一需求，TiCDC 提供了 Syncpoint 功能。

Syncpoint 通过利用 TiDB 提供的 snapshot 特性，让 TiCDC 在同步过程中维护了一个上下游具有一致性 snapshot 的 `ts-map`。把校验动态数据的一致性问题转化为了校验静态 snapshot 数据的一致性问题，达到了接近数据一致性实时校验的效果。

要开启 SyncPoint 功能，你可以在创建同步任务时把 TiCDC 的配置项 `enable-sync-point` 设置为 `true`。开启 Syncpoint 功能后，TiCDC 在数据的同步过程中会根据你所配置的 TiCDC 参数 `sync-point-interval` 来定时对齐上下游的 snapshot，并将上下游的 TSO 对应关系保存在下游的 `tidb_cdc.syncpoint_v1` 表中。

然后，你只需要在 sync-diff-inspector 中配置 `snapshot` 即可对 TiDB 主从集群的数据进行校验。以下 TiCDC 配置示例为创建的同步任务开启 Syncpoint 功能：

```
### 开启 SyncPoint
enable-sync-point = true

### 每隔 5 分钟对齐一次上下游的 snapshot
sync-point-interval = "5m"

### 每隔 1 小时清理一次下游 tidb_cdc.syncpoint_v1 表中的 ts-map 数据
sync-point-retention = "1h"
```

13.13.4.1 获取 ts-map

在下游 TiDB 中执行以下 SQL 语句，从结果中可以获取上游 TSO (primary_ts) 和下游 TSO (secondary_ts) 信息。

```
select * from tidb_cdc.syncpoint_v1;
```

```
+------------------+----------------+--------------------+--------------------+--------------------+
    ↪
| ticdc_cluster_id | changefeed     | primary_ts         | secondary_ts       | created_at
    ↪          |
+------------------+----------------+--------------------+--------------------+--------------------+
    ↪
```

```
| default          | test-2         | 435953225454059520 | 435953235516456963 | 2022-09-13
    ↪ 08:40:15 |
+------------------+----------------+--------------------+--------------------+---------------------+
    ↪
```

以上 syncpoint_v1 表中各列所代表的信息如下:

- ticdc_cluster_id：插入该条记录的 TiCDC 集群的 ID。
- changefeed：插入该条记录的 Changefeed 的 ID。由于不同的 TiCDC 集群可能会存在重名的 Changefeed，所以需要通过 TiCDC 集群 ID 和 Changefeed 的 ID 来确认一个 Changefeed 所插入的 ts-map。
- primary_ts：上游数据库 snapshot 的时间戳。
- secondary_ts：下游数据库 snapshot 的时间戳。
- created_at：插入该条记录的时间。

13.13.4.2 配置 snapshot

使用上一步骤获取的 ts-map 信息来配置上下游数据库的 snapshot 信息。其中的 Datasource config 部分示例配置如下：

```
########################## Datasource config ######################
[data-sources.uptidb]
    host = "172.16.0.1"
    port = 4000
    user = "root"
    password = ""
    snapshot = "435953225454059520"

[data-sources.downtidb]
    host = "172.16.0.2"
    port = 4000
    user = "root"
    password = ""
    snapshot = "435953235516456963"
```

13.13.4.3 注意事项

- TiCDC 在创建 Changefeed 前，请确保 TiCDC 的配置项 enable-sync-point 已设置为 true，这样才会开启 Syncpoint 功能，在下游保存 ts-map。完整的配置请参考TiCDC 同步任务配置文件描述。
- 需要调整 TiKV 的 GC 时间，保证在校验时 snapshot 对应的历史数据不会被执行 GC。建议调整为 1 个小时，在校验后再还原 GC 设置。
- 以上配置只展示了 Datasource config 部分，完整配置请参考sync-diff-inspector 用户文档。
- 从 v6.4.0 开始，TiCDC 使用 Syncpoint 功能需要同步任务拥有下游集群的 SYSTEM_VARIABLES_ADMIN 或者 SUPER 权限。

13.13.5 基于 DM 同步场景下的数据校验

当你在使用TiDB DM 等同步工具时，需要校验 DM 同步后数据的一致性。你可以从 DM-master 拉取指定 task-name 的配置，进行数据校验。

下面是一个简单的配置文件说明，要了解完整配置，请参考sync-diff-inspector 用户文档。

```
### Diff Configuration.

########################### Global config ##########################

### 检查数据的线程数量，上下游数据库的连接数会略大于该值
check-thread-count = 4

### 如果开启，若表存在不一致，则输出用于修复的 SQL 语句
export-fix-sql = true

### 只对比表结构而不对比数据
check-struct-only = false

### dm-master 的地址，格式为 "http://127.0.0.1:8261"
dm-addr = "http://127.0.0.1:8261"

### 指定 DM 的 `task-name`
dm-task = "test"

########################### Task config ##########################
[task]
    output-dir = "./output"

    # 需要比对的下游数据库的表，每个表需要包含数据库名和表名，两者由 `.` 隔开
    target-check-tables = ["hb_test.*"]
```

该配置在 dm-task = “test” 中，会对该任务下 hb_test 库的所有表进行检验，自动从 DM 配置中获取上游对下游库名的正则匹配，以校验 DM 同步后数据的一致性。

13.14 TiUniManager

13.14.1 TiUniManager 概览

TiUniManager 是为分布式数据库 TiDB 打造的管控平台软件和数据库运维管理平台，主要为 TiDB 提供数据库集群管理功能、主机管理功能和平台管理功能，涵盖了数据库运维人员 (DBA) 在 TiDB 上进行的常用运维操作，帮助 DBA 对 TiDB 进行自动化、自助化和可视化管理。

TiUniManager 可帮助 DBA 避免因人工操作失误导致的数据库故障，保障数据库安全、稳定、高效地运行，降低运维 TiDB 的难度，提升 DBA 工作效率。

自 v1.0.2 版本起，TiUniManager 正式开放源码，详见 GitHub 仓库 tiunimanager。

13.14.1.1 软件架构

TiUniManager 的软件架构图如下。

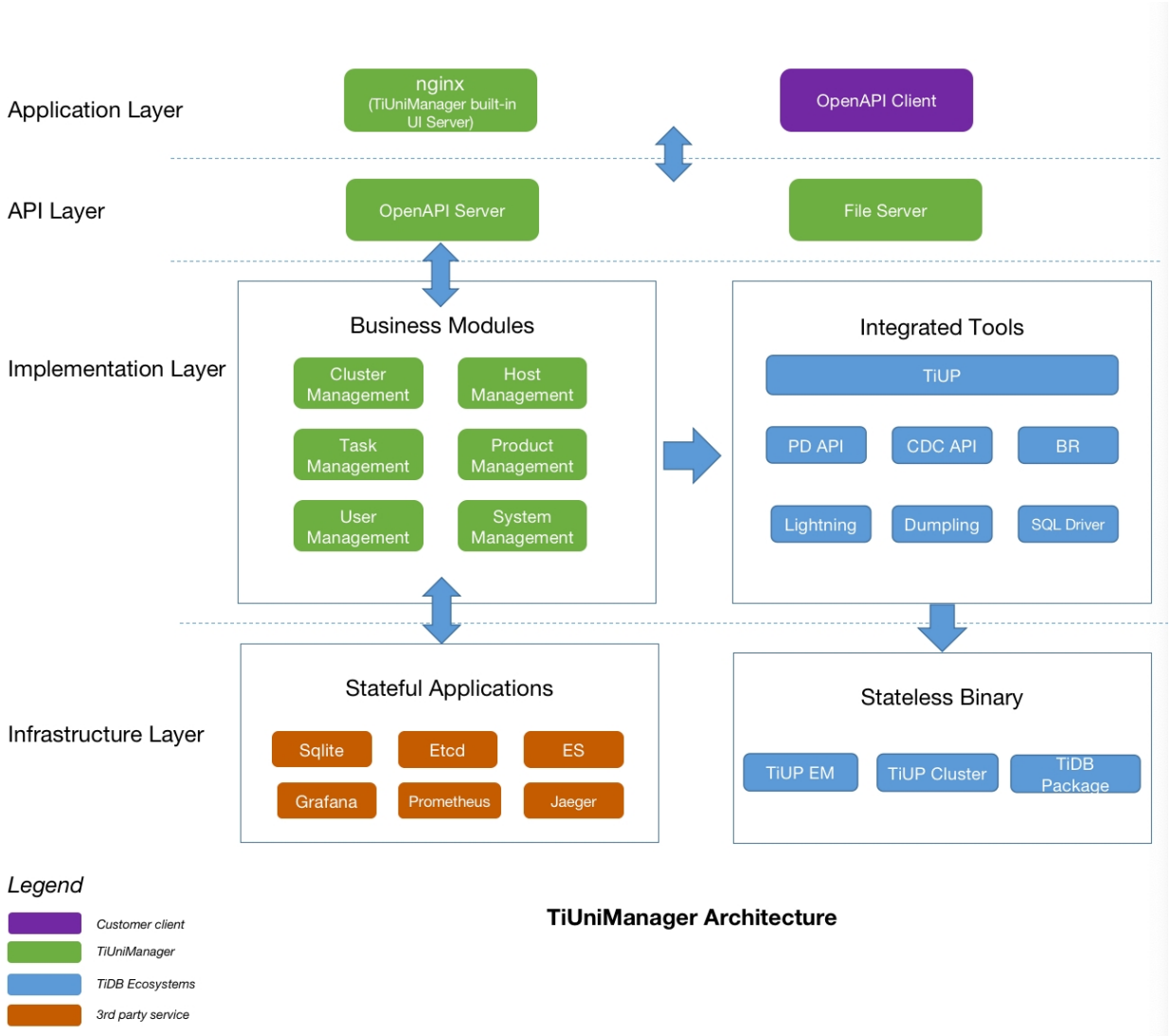


图 298: TiUniManager 架构

13.14.1.1.1 Web UI

TiUniManager Web UI 即用户可见的 TiUniManager 界面，展示用户输入以及服务端返回的数据。用户可与该界面进行交互，通过输入数据来获取想要看到的数据，通过界面操作完成对 TiDB 的部署运维相关工作。

13.14.1.1.2 OpenAPI

TiUniManager 提供数据库管理操作的 OpenAPI，帮助用户灵活完成自动化任务或第三方集成。当前 OpenAPI 为实验特性，不建议在生产环境中使用。

13.14.1.1.3 业务逻辑

TiUniManager 平台核心功能实现层，包括一键创建 TiDB 集群、删除集群、扩容集群、恢复集群、查看集群详情、集群监控报警、错误日志、慢查询日志等。实现层负责以下方面：

- 负责后台定期备份 TiDB 数据库并保存到分布式文件系统。
- 负责管理平台中所有主机信息的导入，主机的上线，记录主机资源使用情况。
- 负责管理平台自身的配置信息，元信息的备份，性能监控等。

13.14.1.2 核心特性

本节介绍 TiUniManager 的核心特性。

13.14.1.2.1 易用性

TiUniManager 提供 Web UI 界面交互模式，UI 界面即开即用。用户根据规划添加主机形成统一的资源池，通过 TiUniManager 可一键创建指定规格的 TiDB 实例，数分钟内就可部署完成集群并对外提供服务。

数据库管理员 (DBA) 根据业务发展情况可随时弹性扩容 TiDB 容量及服务能力。TiUniManager 集成 DBA 常用的运维操作，所有操作都通过可视化的方式操作，极大提升 DBA 运维 TiDB 效率及易用性。

13.14.1.2.2 高可靠性

TiUniManager 支持以下特性，保障数据高可靠：

- 支持自动备份 TiDB 数据库。用户可以设置自动备份的周期和时间。
- 支持按备份文件进行数据恢复，且支持恢复到新的实例。为避免在恢复数据过程中影响业务，数据验证通过后 TiUniManager 将业务流量切换到新实例，完成数据回溯。
- 支持将数据备份文件存储在分布式文件系统，通过多副本冗余和副本分布的不同策略（不同机架等）确保数据不会丢失。
- 支持将 TiDB 部署在不同的可用区。TiDB 自身采用多副本的数据存储，TiUniManager 根据调度策略将 TiDB 的副本放在不同的可用区，实现跨机房数据可靠性功能。

13.14.1.2.3 高可用性

TiUniManager 组件均提供高可用功能，无单点故障，确保平台高可用。

TiDB 自身采用多副本的数据存储，TiUniManager 根据调度策略将 TiDB 的副本放在不同的可用区，并采用业界先进的 Raft 多数派选举算法确保数据 100% 强一致性和高可用。可将副本跨地域部署在不同的数据中心，主副本故障时自动切换，无需人工介入，自动保障业务的连续性，实现真正意义上的异地多活。

13.14.1.3 使用场景

本节介绍 TiUniManager 的使用场景。

13.14.1.3.1 资源管理

- 支持导入主机
- 查看主机资源列表

13.14.1.3.2 集群管理

- 支持一键创建（部署）TiDB 集群，创建过程中可选择可用区、版本、规格、节点个数等
- 支持一键接管已有 TiDB 集群
- 支持一键删除 TiDB 集群
- 支持一键弹性扩容 TiDB 集群，可分别扩展计算和存储能力
- 支持一键备份与恢复集群数据
- 支持一键查看监控报警
- 支持一键升级集群版本
- 支持日志管理，包括服务日志和慢查询日志等
- 支持通过集群 ID、名称、标签等参数搜索实例

13.14.1.3.3 数据备份与恢复

- 支持设置自动备份策略
- 支持一键手工备份
- 支持一键删除备份记录
- 支持从备份记录恢复至新集群

13.14.1.3.4 数据导入与导出

- 支持从 CSV 文件导入数据到 TiDB 集群
- 支持从 SQL 文件导入数据到 TiDB 集群
- 支持从 TiDB 集群导出数据到 CSV 文件
- 支持从 TiDB 集群导出数据库到 SQL 文件

13.14.1.3.5 数据同步

- 支持从 TiDB 集群向下游（Kafaka、MySQL、TiDB）数据库同步数据

> **注意：**
>
> 数据同步功能仅支持 TiDB 5.2.2 及以上版本的集群。

13.14.1.3.6 克隆集群

- 支持克隆一个已有的 TiDB 集群，并创建一个新的 TiDB 集群
- 支持同步克隆和快照克隆两种数据策略

> **注意：**
>
> 同步克隆功能仅支持 TiDB 5.2.2 及以上版本的集群。

13.14.1.3.7 集群参数管理

- 提供 TiDB 版本对应的参数组模板。关于支持的 TiDB 版本，参见TiUniManager 支持的 TiDB 版本。
- 提供统一界面查看、修改 TiDB 集群参数

13.14.1.3.8 工作流任务管理

- 支持查看从 TiUniManager 发起的所有工作流任务记录列表
- 支持查看工作流任务详情

13.14.1.3.9 系统管理

- 支持 TiUniManager 系统监控和 TiUniManager 主机监控
- 支持查看 TiUniManager 系统日志
- 支持查看 TiUniManager 微服务 Trace 信息

13.14.2 TiUniManager 安装和运维指南

本文档介绍 TiUniManager 的安装部署流程以及运维方法，指导数据库管理员完成 TiUniManager 的部署和配置。

13.14.2.1 软硬件环境配置

本节介绍部署 TiUniManager 前需要满足的软硬件环境配置要求。

13.14.2.1.1 Linux 操作系统版本

要部署和运行 TiUniManager 服务，确保 Linux 操作系统的版本满足以下要求：

Linux 操作系统	版本
Red Hat Enterprise Linux	7.3 及以上的 7.x 版本
CentOS	7.3 及以上的 7.x 版本

13.14.2.1.2 中控机软件配置

TiUniManager 中控机是运行 TiUniManager 服务的中央控制节点。你可通过登录 TiUniManager 中控机上的 Web console 或 OpenAPI 完成对 TiDB 集群的日常管理。中控机的软件配置要求如下：

软件	版本
sshpass	1.06 及以上
TiUP	1.9.0 及以上

13.14.2.1.3 服务器配置

TiUniManager 支持部署和运行在 Intel x86-64 架构的 64 位通用硬件服务器平台上。对于开发、测试、及生产环境的服务器硬件配置（不包含操作系统 OS 本身的占用）有以下要求：

开发及测试环境

在开发及测试环境中，对服务器硬件配置要求如下：

组件	CPU	内存	硬盘类型	网络	实例数量（最低要求）
TiUniManager	8 核 +	16 GB+	SAS	万兆网卡（2 块最佳）	1

生产环境

在生产环境中，对服务器硬件配置要求如下：

组件	CPU	内存	硬盘类型	网络	实例数量（最低要求）
TiUniManager	48 核 +	128 GB+	SAS/SSD	万兆网卡（2 块最佳）	1

13.14.2.1.4 网络要求

TiUniManager 正常运行需要网络环境提供如下端口配置，管理员可根据实际环境中 TiUniManager 组件部署的方案，在网络侧和主机侧开放相关端口：

组件	默认端口	说明
Web server	4180 或 4183	HTTP 端口：4180HTTPS 端口：4183
OpenAPI server	4100 或 4103	OpenAPI 服务端口: 4100OpenAPI 监控端口：4103
Cluster server	4101 或 4104	集群服务端口
File server	4102 或 4105	文件上传或下载的服务器端口
etcd	4106 或 4107	etcd 服务端口
Elasticsearch server	4108	Elasticsearch 服务端口
Kibana server	4109	Kibana 服务端口
Prometheus	4110	Prometheus 服务端口
Grafana server	4111	Grafana 服务端口
Alertmanager server	4112 或 4113	告警管理服务端口
Jaeger(tracer server)	4114 到 4123	Jaeger 服务端口
node_exporter	4124	TiUniManager 主机系统上报信息的通信端口

13.14.2.2 客户端 Web 浏览器要求

你可在较新版本的常见桌面浏览器中使用 TiUniManager，浏览器的版本要求如下：

- Chrome > 79
- Firefox > 72
- Microsoft Edge > 79
- Safari > 14

> **注意：**
>
> 若使用旧版本浏览器或其他浏览器访问 TiUniManager，部分界面可能无法正常工作。

13.14.2.3 拓扑模板

在线部署 TiUniManager 前，你需要准备好 YAML 拓扑文件。TiUniManager 离线包中包含 YAML 拓扑文件模板。本节介绍用于部署 TiUniManager 拓扑配置模版。见 TiUniManager 拓扑配置模版 config.yaml（单机版）。

如果 TiUniManager 中控机通过用户名密钥访问 TiDB 资源机，需要参照指定 TiUniManager 中控机登录 TiDB 资源机的用户名和密钥，在配置文件 `config.yaml` 中指定用户名和密钥。

13.14.2.4 离线部署 TiUniManager

本节介绍如何在离线环境部署 TiUniManager。当前 TiUniManager 只支持通过离线部署。

1. 通过 `https://download.pingcap.org/em-enterprise-server-${version}-linux-amd64.tar.gz` 下载 TiUniManager 离线安装包。

   下载链接中的 `${version}` 为 TiUniManager 的版本号。例如，`v1.0.2` 版本的下载链接为 `https://download.pingcap.org/em-enterprise-server-v1.0.2-linux-amd64.tar.gz`。在下载时，你需要将链接中的 `${version}` 替换为目标版本号。

2. 发送 TiUniManager 离线安装包至 TiUniManager 中控机。

   离线安装包放置于 TiUniManager 中控机，使用具有 sudo 权限的账号执行后续操作。

3. 解压 TiUniManager 离线包。

```
tar xzvf em-enterprise-server-${version}-linux-amd64.tar.gz
```

4. 安装 TiUniManager。进入解压后的目录，执行 `install.sh` 脚本。

```
sudo sh em-enterprise-server-${version}-linux-amd64/install.sh <TiUniManager 中控机 IP>
```

5. 声明环境变量。

```
# 切换到 tidb 账号下
su - tidb

# 声明环境变量，使 tiup 工具生效
source /home/tidb/.bash_profile
```

6. 生成 tidb 帐户下的密钥。

```
# 切换到 tidb 账号下
su - tidb

# 生成 rsa 密钥
ssh-keygen -t rsa

# 复制密钥到 tiup_rsa
cp /home/tidb/.ssh/id_rsa /home/tidb/.ssh/tiup_rsa
```

7. 编辑拓扑配置文件。根据实际环境，你可编辑位于 /home/tidb/ 下的拓扑配置文件 config.yaml。

8. 导入 TiDB Server 离线镜像包。

   离线环境下，需要在 TiUniManager 中控机本地目录上导入 TiDB 离线镜像包，否则无法通过 TiUniManager 中控机完成对 TiDB 集群的日常管理。

```
# 切换到 tidb 账号下
su - tidb

# 下载解压 TiDB Server 离线镜像包，将 ${version} 手动替换为实际的 TiDB 版本号。

wget https://download.pingcap.org/tidb-community-server-${version}-linux-amd64.tar.gz
tar xzvf tidb-community-server-${version}-linux-amd64.tar.gz

# 导入离线镜像包
TIUP_HOME=/home/tidb/.tiup tiup mirror merge tidb-community-server-${version}-linux-amd64
```

   在 TiUniManager 中控机上查看本地 TiDB 镜像源。

```
# 切换到 tidb 账号下
su - tidb

# 查看 TiDB 镜像源地址
tiup mirror show
```

9. 执行命令部署 TiUniManager。

```
# 切换到 tidb 账号下
su - tidb

# 部署名称为 "em-test" 的 TiUniManager，注意这里的版本号不带 v，比如 v1.0.0 的版本号，
    ↪ 正确的输入是 1.0.0
TIUP_HOME=/home/tidb/.em tiup em deploy em-test <版本号> config.yaml -u <具有sudo权限的账号>
    ↪  -p

# 启动 TiUniManager
TIUP_HOME=/home/tidb/.em tiup em start em-test
```

13.14.2.5 指定 TiUniManager 中控机登录 TiDB 资源机的帐户和密钥

默认情况下，你可以通过 TiUniManager Web 控制台资源管理 > 导入主机页面提供的主机资源模板填写 TiDB 资源机帐户和密码。

如果用户环境中不允许账号密码方式登录 TiDB 资源机，可在 `config.yaml` 文件中进行配置，以帐户密钥方式登录 TiDB 资源机。`config.yaml` 中相关配置项如下：

配置描述	配置参数名	示例值
login-host-user	登录 TiDB 资源机的用户名	tidb
login-private-key-path	位于 TiUniManager 中控机上登录 TiDB 资源机使用的私钥路径	/home/tidb/.ssh/id_rsa
login-public-key-path	位于 TiUniManager 中控机登录 TiDB 资源机使用的公钥路径	/home/tidb/.ssh/id_rsa.pub

13.14.2.6 升级 TiUniManager（适用于 v1.0.1 及以上版本）

本节适用于安装过 TiUniManager v1.0.1 及以上版本，并打算升级 TiUniManager 的用户。如首次安装可跳过本节。

> 注意：
>
> - 本节介绍的方法仅适用于从 TiUniManager v1.0.1 升级至 v1.0.2 及以上版本，不适用于从 v1.0.0 升级至 v1.0.1 及以上版本。
> - 如果你已安装 TiUniManager v1.0.0 版本，想要升级至 v1.0.1 及以上版本，则需要先参考升级 TiUniManager v1.0.0 至 v1.0.1 进行升级，再参考本节内容从 v1.0.1 升级至较新的目标版本。
> - 在 v1.0.0 和 v1.0.1 中，TiUniManager 原名为 TiDB Enterprise Manager。自 v1.0.2 起正式更名为 TiUniManager。

1. 升级 TiUniManager 离线包。

   参考离线部署 TiUniManager 中第 1-3 步：通过 TiUniManager 产品团队获得最新 TiUniManager 离线包，发送 TiUniManager 离线安装包至目标集群的 TiUniManager 中控机，解压离线包。然后执行下面命令更新 TiUniManager 离线包。

   ```
   # user 为之前部署 TiUniManager 的帐户，默认为 tidb。${version} 为 TiUniManager 的版本号，
       ↪ 需要手动替换为实际的版本号。
   sudo sh em-enterprise-server-${version}-linux-amd64/update.sh <user>
   ```

2. 执行升级 TiUniManager 命令。

   ```
   # 切换到 tidb 账号下
   su - tidb

   # 将集群 <cluster-name> 升级到特定版本 <version>
   # 目前仅支持原地停机升级，并且暂不支持版本降级和回退。为安全起见，请按照 停止集群 > 备份
       ↪ tiunimanager 元数据 > 升级集群的流程操作
   TIUP_HOME=/home/tidb/.em tiup em upgrade <cluster-name> <version>
   ```

13.14.2.7 升级 TiUniManager v1.0.0 至 v1.0.1

本节介绍的方法适用于从 v1.0.0 升级至 v1.0.1。如需升级至更高版本，请先升级至 v1.0.1，再参考升级 TiUniManager（适用于 v1.0.1 及以上版本）进行升级。如首次安装可跳过本节。

> 注意：
>
> 在 v1.0.0 和 v1.0.1 中，TiUniManager 原名为 TiDB Enterprise Manager。自 v1.0.2 起正式更名为 TiUniManager。

1. 升级 TiUniManager 离线包。

   参考离线部署 TiUniManager 中第 1-3 步：通过 TiUniManager 产品团队获得最新 TiUniManager 离线包，发送 TiUniManager 离线安装包至目标集群的 TiUniManager 中控机，解压离线包。然后执行下面命令更新 TiUniManager 离线包。

   ```
   # user 为之前部署 TiUniManager 的帐户，默认为 tidb
   sudo sh em-enterprise-server-v1.0.1-linux-amd64/update.sh <user>
   ```

2. 做升级 TiUniManager 之前的准备。

   执行以下命令为 TiUniManager 升级做准备。以下命令会自动备份 TiUniManager v1.0.0 中的数据。请注意，执行完此命令后直至升级完成，请不要再通过 TiUniManager 界面做其它操作，以避免备份未包含最新数据导致升级后数据丢失的情况。

   ```
   # user 为之前部署 TiUniManager v1.0.0 的帐户，默认为 tidb
   # cluster-name 为当前部署的 TiUniManager 名称，默认为 em-test
   sudo sh em-enterprise-server-v1.0.1-linux-amd64/prepare_upgrade.sh <user> <TiUniManager
       ↪ 中控机 IP> <cluster-name>
   ```

3. 检查新的 TiUniManager 部署拓扑文件。

   如果你在部署 TiUniManager v1.0.0 时曾对 config.yaml 文件做了自定义修改，请参考下列子步骤查看 v1.0.0 的配置文件 config.yaml 和待部署 v1.0.1 配置文件 config-v1.0.1.yaml 的区别，并做相应修改。

   1. 切换至 tidb 账号。

      ```
      su - tidb
      ```

   2. 查看配置变更。

      对比 config.yaml 和 config-v1.0.1.yaml 生效内容的差异。如果差异仅为 db_path 所在行，则可跳过本步骤，无须执行后续子步骤（备份 db_path 以及更新 config-v1.0.1.yaml），可直接进入删除 TiUniManager v1.0.0 的步骤。如果差异不仅有 db_path 所在行，还存在其它差异，则需要继续执行后续子步骤（备份 db_path 以及更新 config-v1.0.1.yaml）。

      ```
      diff config.yaml config-v1.0.1.yaml
      ```

   3. 记下 config-v1.0.1.yaml 中的数据备份路径 db_path。

   db_path 为 TiUniManager 备份数据的路径。在“升级 TiUniManager 之前的准备”这一步中 TiUniManager 已经自动在 config-v1.0.1.yaml 中写好了默认的 v1.0.1 数据备份路径，此前已备份的 v1.0.0 数据将恢复到该路径下。

   ```
   查看 `config-v1.0.1.yaml` 中的 `db_path`，记下该行内容，在以下子步骤 5 中更新 `config-v1
       ↪ .0.1.yaml` 时需要用到该行内容。示例如下：

   ```yaml
   db_path: "/home/tidb/em-backup-20220511-154415/em.db"
   ```
   ```

4. 覆盖 config-v1.0.1.yaml。

将 TiUniManager v1.0.0 的 config.yaml 文件复制一份并覆盖 v1.0.1 的 config-v1.0.1.yaml。

```shell
cp config.yaml config-v1.0.1.yaml
```

5. 更新 config-v1.0.1.yaml。

覆盖配置文件后，将子步骤 3 中拷贝的 db_path 填写至 config-v1.0.1.yaml，注意空格的缩进。示例如下：

```
em_cluster_servers:
  - host: 127.0.0.1
    db_path: "/home/tidb/em-backup-20220511-154415/em.db"
```

4. 删除 TiUniManager v1.0.0。

```
# 切换到 tidb 账号下
su - tidb

# 查看现有 TiUniManager 服务
TIUP_HOME=/home/tidb/.em tiup em list

# 删除指定的 TiUniManager 服务
TIUP_HOME=/home/tidb/.em tiup em destroy <cluster-name>
```

5. 部署 TiUniManager v1.0.1。

```
# 切换到 tidb 账号下
su - tidb

# 更新 em 工具
TIUP_HOME=/home/tidb/.em tiup update em
TIUP_HOME=/home/tidb/.em tiup update --all

# 部署名称为 "em-test" 的 TiUniManager
TIUP_HOME=/home/tidb/.em tiup em deploy em-test 1.0.1 config-v1.0.1.yaml -u <具有sudo
    ↪ 权限的账号> -p

# 启动 TiUniManager
TIUP_HOME=/home/tidb/.em tiup em start em-test

# 将 db_path 注释掉。如果不注释，后续对 em 进行重启、扩容等操作时 TiUniManager
    ↪ 可能会将数据变更还原
sed -i "s/  db_path:/# db_path:/" /home/tidb/.em/storage/em/clusters/em-test/meta.yaml
```

6. 重建资源机器的 filebeat。

```
# 切换到 tidb 账号下
su - tidb

# 重建资源机器 filebeat
TIUP_HOME=/home/tidb/.em tiup em scale-out em-test /home/tidb/em-v1.0.0-backup/resource-
    ↪ filebeat-scale-out-topology.yaml --user tidb -i /home/tidb/.ssh/tiup_rsa --wait-
    ↪ timeout 360 --yes
```

7. 使集群日志功能生效。

   待所有上述步骤完成后，大部分的升级已完成。由于在升级过程中，TiUniManager 对资源机器的 filebeat 进行了重装。为了使得查看 TiDB 集群日志功能在 TiUniManager 中生效，你需要手动在 TiUniManager 界面操作重启 TiDB 集群以刷新配置。为避免影响用户服务，请尽量在业务低峰期操作。

13.14.2.8 备份恢复 TiUniManager 元信息

为提高 TiUniManager 可用性，应对容灾场景，你可对 TiUniManager 元信息进行备份，并恢复元信息到新的 TiUniManager 中控机。

1. 备份 TiUniManager 元信息。

```
# 切换到 tidb 账号下
su - tidb

# 将当前 TiUniManager 系统 <cluster-name> 的元数据备份到中控机的 <target-path>
TIUP_HOME=/home/tidb/.em tiup em backup <cluster-name> <target-path> -N <tiunimanager-ip>
```

2. 恢复 TiUniManager 元信息。

```
# 切换到 tidb 账号下
su - tidb

db_path: "/home/tidb/em.db"
```

   从备份的元数据中恢复到新集群，流程和部署新集群相同。唯一的区别是在集群 yaml 配置中，`em_cluster_servers` 里增加了 `db_path: "/home/tidb/em.db"`，详细见 TiUniManager 根据元数据恢复新集群拓扑配置模版 em.yaml（单机版）。

13.14.2.9 修改默认的集群备份路径

TiUniManager 默认的集群备份路径相关配置参数如下：

配置描述	配置参数名	参考值
TiDB 集群备份的存储类型（仅支持 NFS 或 S3）	BackupStorageType	‘s3’ 或 ‘nfs’
TiDB 集群备份的存储路径（S3 bucket 路径，或 NFS share 的绝对路径）	BackupStoragePath	‘bucketPath/backup’ (S3 路径示例) ‘/mnt/nfs-path’ (NFS 路径示例)
TiDB 集群备份在 S3 共享存储时，S3 的 AccessKey	BackupS3AccessKey	

配置描述	配置参数名	参考值
TiDB 集群备份在 S3 共享存储时，S3 的 SecretAccessKey	BackupS3SecretAccessKey	
TiDB 集群备份在 S3 共享存储时，S3 的 Endpoint（域名）	BackupS3Endpoint	

当前不支持通过 TiUniManager 界面修改备份路径。如需修改备份路径，需要通过 OpenAPI 修改配置参数，以修改配置参数 “BackupS3AccessKey” 为例：

1. 登录获取 user token。

```
curl -X 'POST' \ 'http://172.16.6.206:4180/api/v1/user/login' \ -H 'accept: application/json
    ↪ ' \ -H 'Content-Type: application/json' \ -d '{ "userName": "admin", "userPassword":
    ↪  "admin" }'
```

> **注意：**
> 你需要将以上命令中 172.16.6.206:4180 替换为实际环境中 TiUniManager 中控机的 IP 地址和 WebServer 服务端口。

2. 查看配置参数值。

```
curl -X GET "http://172.16.6.206:4100/api/v1/config/?configKey=BackupS3AccessKey" -H "
    ↪ Authorization: Bearer 6ea768e4-c0ec-4d48-b401-0f114ddc994c"
```

3. 修改配置参数值 BackupS3AccessKey 的值为 newValue。你可以将 newValue 替换为你期待的备份路径。

```
curl -X POST "http://172.16.6.206:4100/api/v1/config/update" -d "{ \"configKey\": \"
    ↪ BackupS3AccessKey\", \"configValue\": \"newValue\"}" -H "Authorization: Bearer 6
    ↪ ea768e4-c0ec-4d48-b401-0f114ddc994c"
```

13.14.2.10 修改默认的数据导入导出路径

TiUniManager 默认的导入导出路径在 TiUniManager 中控机上，细节如下：

配置描述	配置参数名	参考值
TiUniManager 共享存储中导入文件存储位置（建议配置为 NFS 共享存储）	importShareStoragePath	/home/tidb/import（备注：中控机 tidb 账号拥有该路径的读写权限）

配置描述	配置参数名	参考值
TiUniManager 共享存储中导出文件存储的位置（建议配置为 NFS 共享存储）	ExportShareStoragePath	/ ↪ home ↪ / ↪ tidb ↪ / ↪ export ↪ （备注：中控机 tidb 账号拥有该路径的读写权限）

当前 TiUniManager 不支持通过 TiUniManager 界面修改导入导出的路径。如需修导入导出路径，需要通过 OpenAPI 对配置进行修改，以修改 “ImportShareStoragePath” 为例：

1. 登录获取 user token。

```
curl -X 'POST' \ 'http://172.16.6.206:4180/api/v1/user/login' \ -H 'accept: application/json
    ↪ ' \ -H 'Content-Type: application/json' \ -d '{ "userName": "admin", "userPassword":
    ↪  "admin" }'
```

> **注意：**
> 你需要将以上命令中 172.16.6.206:4180 替换为实际环境中 TiUniManager 中控机的 IP 地址和 WebServer 服务端口。

2. 查看配置参数值。

```
curl -X GET "http://172.16.6.206:4100/api/v1/config/?configKey=ImportShareStoragePath" -H "
    ↪ Authorization: Bearer 6ea768e4-c0ec-4d48-b401-0f114ddc994c"
```

3. 修改配置参数 ImportShareStoragePath 的值为 newValue，你可以将 newValue 替换为你期待的导入导出路径。

```
curl -X POST "http://172.16.6.206:4100/api/v1/config/update" -d "{ \"configKey\": \"
    ↪ ImportShareStoragePath\", \"configValue\": \"newValue\"}" -H "Authorization: Bearer
    ↪ 6ea768e4-c0ec-4d48-b401-0f114ddc994c"
```

13.14.2.11 删除 TiUniManager

如要删除 TiUniManager 服务，请执行以下命令：

```
su - tidb

### 查看现有 TiUniManager 服务
TIUP_HOME=/home/tidb/.em tiup em list

### 删除指定的 TiUniManager 服务
TIUP_HOME=/home/tidb/.em tiup em destroy <cluster-name>
```

13.14.2.12 指定外部的 Elasticsearch 服务

Elasticsearch 是一个分布式、高扩展、高实时的搜索与数据分析引擎。它能很方便的使大量数据具有搜索、分析和探索的能力。目前 TiUniManager 主要依赖 Elasticsearch 来做日志数据的存储和检索。

TiUniManager 可以通过 `config.yaml` 文件中全局的 `external_elasticsearch_url` 参数来指定外部的 Elasticsearch 服务。默认值为空。

`external_elasticsearch_url` 参数值格式为 `<IP:Port>`，例如：`127.0.0.1:9200`。

在 `config.yaml` 文件中，`external_elasticsearch_url` 和 `elasticsearch_servers` 都是用来指定 Elasticsearch 部署信息的，不能同时指定。如果指定了 `external_elasticsearch_url` 的值，则需要注释掉 `elasticsearch_servers` 组件的配置信息注释。

配置格式参考 TiUniManager 拓扑配置模版 config.yaml（单机版）。

13.14.2.13 手动指定部署的 Elasticsearch 组件堆内存大小

部署 TiUniManager 的拓扑文件支持根据真实部署中控机资源大小指定 Elasticsearch 组件的堆内存大小。通过 `elasticsearch_servers` 下的 `heap_size` 参数进行指定。例如 16g。

`heap_size` 是选填参数，不手工指定会使用默认值。默认值为 4g。

配置格式参考 TiUniManager 拓扑配置模版 config.yaml（单机版）。

13.14.2.14 安装 Kibana 组件（可选）

Kibana 是一个针对 Elasticsearch 的开源分析及可视化平台，用来搜索、查看交互存储在 Elasticsearch 索引中的数据。使用 Kibana，可以通过各种图表进行高级数据分析及展示。

目前 TiUniManager 默认安装 Kibana，集成 Kibana 主要是做日志的展示，在控制台页面系统管理下的系统日志进行查看。

如果不想安装 kibana 组件，则直接在 `config.yaml` 文件中将 `kibana_servers` 组件相关的配置注释掉即可。

13.14.2.15 安全加固

完成 TiUniManager 安装后，建议用户通过系统防火墙为 Elasticsearch 和 Kibana 添加网络访问控制，仅允许 TiUniManager 中控机与上述组件通信，参考命令如下所示：

```
### 允许来自 TiUniManager 中控机的连接
iptables -A INPUT -p tcp -s <tiunimanager addr> --dport <elasticsearch port> -j ACCEPT
### 拒绝其它 IP 的访问
iptables -A INPUT -p tcp --dport <elasticsearch port> -j DROP
```

13.14.3 TiUniManager 快速操作指南

本文档介绍 TiUniManager 的常见操作场景以及基本的集群操作，包括将主机节点加入 TiDB 集群、创建 TiDB 集群、删除集群、接管集群。

在阅读本文档前，你需要先阅读TiUniManager 安装和运维手册完成 TiUniManager 的安装。

13.14.3.1 将主机节点加入 TiDB 集群

系统初始化或者扩容机器时，系统管理员需将主机信息导入 TiUniManager 平台，TiUniManager 将导入的机器加入到集群中，由平台统一管理。

13.14.3.1.1 前置条件

将主机节点加入 TiDB 集群前，需要保证：

- 已经登录 TiUniManager 控制台
- 已安装主机的操作系统和所依赖的软件，并通过测试

> 注意：
>
> - 你需要按照主机模板完整、正确地填写字段信息。详情参见本节中主机模板字段说明。
> - 导入主机时，TiDB 会对主机进行检查，参见TiDB 环境与系统配置检查。

TiUniManager 中控机通过 SSH 连接主机，默认连接端口为 22。如果环境中 SSH 端口不为默认的 22，可通过 OpenAPI 修改 `config_default_ssh_port` 参数来配置主机的默认登陆端口，该参数默认值为 22。以下示例通过 OpenAPI 修改 `config_default_ssh_port` 参数的值，从而修改主机的默认登陆端口：

1. 登录 TiUniManager 获取用户 Token。

```
curl -X 'POST' \ 'http://172.16.6.206:4180/api/v1/user/login' \ -H 'accept: application/json
    ↪ ' \ -H 'Content-Type: application/json' \ -d '{ "userName": "admin", "userPassword":
    ↪  "admin" }'
```

> **注意：**
>
> 你需要将以上命令中 172.16.6.206:4180 替换为实际环境中 TiUniManager 中控机的 IP 地址和 WebServer 服务端口。

2. 查看 config_default_ssh_port 的参数值。

```
curl -X GET "http://172.16.6.206:4100/api/v1/config/?configKey=config_default_ssh_port" -H "
    ↪ Authorization: Bearer 6ea768e4-c0ec-4d48-b401-0f114ddc994c"
```

3. 将 config_default_ssh_port 的值修改为 newValue。

```
curl -X POST "http://172.16.6.206:4100/api/v1/config/update" -d "{ \"configKey\": \"
    ↪ config_default_ssh_port\", \"configValue\": \"newValue\"}" -H "Authorization: Bearer
    ↪  6ea768e4-c0ec-4d48-b401-0f114ddc994c"
```

13.14.3.1.2 操作步骤

1. 进入资源管理页面。
2. 点击导入主机按钮。
3. 点击下载主机模板按钮。
4. 双击打开模板，并按照模板填入相应信息并保存。
5. 点击上传按钮，选择步骤 5 中编辑的文件，并上传该文件。
6. 点击确认按钮，确认导入主机信息。

13.14.3.1.3 主机模板字段说明

Hostname	主机名
IP	主机 IP 地址
Login username	主机用户名（选填，当通过用户名密码方式导入主机时，才需要填写）
Login password	主机密码（选填，当通过用户名密码方式导入主机时，才需要填写）
Region	区域 ID (产品初始化时设定的 Region ID）
Zone	可用区 ID (产品初始化时设定的 Zone ID）
Rack	机架 ID （产品初始化时设定的 Rack ID）
Arch	体系架构
OS Version	操作系统版本
Kernel	内核版本
vCPU	CPU 核数
Memory	内存大小
NIC	网卡规格
Cluster Purpose	区分主机用来部署什么类型的集群，包括 TiDB、 DM、 TiUniManager
Host Purpose	主机用途：用来区分部署的组件类型，包括：Compute、Storage、Schedule 三种用途。多种用途以逗号 “,” 连接，例如 ‘Compute,Storage,Schedule’

Hostname	主机名
Disk Type	磁盘类型，包括：NVMe SSD、SSD、SATA 三种类型
Disks	磁盘信息，包括磁盘名称、容量、状态、路径，示例：{“name”：“sda”，“capacity”：256，“status”：“Available”，“path”：“/mnt/sda”}

如果通过用户名密钥方式导入主机，需要在 TiUniManager 安装前配置用户名和密钥路径，参见指定 TiUniManager 中控机登录 TiDB 资源机的账户和密钥。

13.14.3.2 创建集群

TiUniManager 部署完成后，你可以通过 TiUniManager 创建 TiDB 集群，并自定义集群配置。

13.14.3.2.1 前置条件

- 已登录 TiUniManager 控制台
- 已完成主机资源导入

13.14.3.2.2 操作步骤

1. 进入集群管理 > 集群页面。
2. 点击创建集群按钮，跳转至创建实例的页面。
3. 选择集群创建模式。
4. 选择集群主机所在厂商、区域。
5. 输入以下数据库基本信息：
    - 数据库类型
    - CPU 体系架构
    - 数据库版本
    - 参数组
6. 设置数据库产品各组件的以下配置：
    - 所在可用区
    - 实例规格
    - 实例数量
7. 输入集群的以下基本信息：
    - 集群名称。集群名称必须是 4 - 64 个字符，可包含大小写字母、数字和连字符，并以字母或数字开头。
    - 集群标签
    - 数据库管理员 Root 的密码。密码必须是 8-64 个字符，可包含大小写字母、数字和可见的特殊字符（包括 `!@#$%^&*()_+=`）
    - 是否独占部署
8. 点击提交按钮，确认主机资源库存满足集群要求后，点击确认创建按钮。

13.14.3.3 删除集群

由于业务或其他原因，如果你不再需要某个已创建的实例，可以将此集群删除。

13.14.3.3.1 前置条件

- 已登录 TiUniManager 控制台。
- 待删除的 TiDB 集群已存在。

> **注意：**
>
> 删除集群同时将删除集群上自动备份的数据。如果需要某个备份数据，应在删除实例之前将该备份恢复到新实例上，数据恢复操作可参见备份管理 - 数据恢复。

13.14.3.3.2 操作步骤

1. 进入集群管理 > 集群页面。
2. 选择待删除的集群，点击集群 ID 进入集群详情页面。
3. 点击删除按钮。
4. 选择是否在删除集群前完成一次数据备份。
5. 选择是否保留手动备份数据。
6. 输入 “delete” 以确认删除，点击确认删除按钮。

13.14.3.4 接管集群

通过 TiUniManager 接管已经存在的 TiDB 集群，统一管理新老集群，提高集群管理效率。

13.14.3.4.1 前置条件

已经登录 TiUniManager 控制台。

13.14.3.4.2 操作步骤

1. 进入集群管理 > 集群页面。
2. 点击接管集群按钮，进入接管集群页面。
3. 输入接管集群的基本信息：集群名称、数据库用户名 root、数据库密码。
4. 输入接管集群中控配配置信息：
    - 接管集群中控机主机 IP 地址
    - 接管集群中控机 SSH 端口号
    - 接管集群中控机 SSH 用户名
    - 接管集群中控机 SSH 密码
    - 接管集群中控机 TiUP 路径（即 `.tiup` 目录所在路径，不含路径结尾的 `/`，例如 `/root/.tiup` 不能填写为 `/root/.tiup/`）
5. 导入接管集群的主机（导入主机流程参见将主机节点加入 TiDB 集群）。

13.14.4 操作指南

13.14.4.1 TiUniManager 登录与初始化

本文档介绍如何登录 TiUniManager、进行产品初始化，以及如何退出登录。

13.14.4.1.1 登录 TiUniManager 控制台

使用 TiUniManager 进行操作前，你需要先登录 TiUniManager 控制台。

要登录 TiUniManager 控制台，你需要先参考TiUniManager 安装手册 开启 TiUniManager 服务。

> 注意：
>
> 当前 TiUniManager 暂不支持找回密码。

按照以下步骤进行登录：

1. 在浏览器中输入 TiUniManager 地址，跳转至登录页面。
2. 在登录页面输入用户名和密码。默认用户名为 admin，密码为 admin。
3. 点击登录按钮进行登录并跳转至概览页面。

13.14.4.1.2 退出登录

已登录 TiUniManager 控制台的用户可以退出登录，返回登录页面。

> 注意：
>
> 退出登录后，TiUniManager 会清除缓存数据。如果再次登录同一用户，之前未保存的编辑信息会丢失。

操作步骤：点击用户图标，出现退出登录按钮，点击该按钮可退出登录并返回登录页面。

13.14.4.1.3 产品初始化

首次登录 TiUniManager 系统后，需要完成对 TiDB 所在数据中心、TiDB 产品组件、TiDB 版本信息的初始化。

数据中心当前默认为本地数据中心 (Local)，本地数据中心下按照区域 (Region) - 可用区 (Available Zone) 2 层层级来组织主机资源。

以下为主机资源组织结构示例：

```
- 本地数据中心（Local）
    - 区域 Region 1
        - 可用区 Zone1_1
```

```
      - 可用区 Zone1_2
  - 区域 Region 2
      - 可用区 Zone2_1
      - 可用区 Zone2_2
```

操作步骤

1. 登录 TiUniManager 控制台。
2. 进入数据中心初始化页面
    1. 自定义填写本地数据中心的厂商名称。
    2. 自定义填写区域 ID、区域名称。
    3. 自定义填写可用区 ID、可用区名称。
    4. 自定义填写 TiDB、PD、TiKV、TiFlash 实例规格。
3. 点击下一步进入产品组件初始化页面。
    5. 自定义填写组件名称。
    6. 设置组件端口范围。
4. 点击下一步进入产品版本初始化页面。选择希望安装的 TiDB 版本和架构。
5. 点击完成按钮。

13.14.4.2 TiUniManager 主机资源管理

本文档介绍如何通过 TiUniManager 管理主机资源，包括导入、查看、删除主机资源。

13.14.4.2.1 导入主机

系统初始化或者扩容机器时，系统管理员需将机器信息导入 TiUniManager 平台，TiUniManager 将导入的机器加入到集群中，由平台统一管理。

前置条件

将主机节点加入 TiDB 集群前，需要保证：

- 已经登录 TiUniManager 控制台
- 已安装主机的操作系统和所依赖的软件，并通过测试

> 注意：
>
> - 你需要按照主机模板完整、正确地填写字段信息。详情参见本节中主机模板字段说明。
> - 导入主机时，TiDB 会对主机进行检查，参见TiDB 环境与系统配置检查。

TiUniManager 中控机通过 SSH 连接主机，默认连接端口为 22。如果环境中 SSH 端口不为默认的 22，可通过 OpenAPI 修改 `config_default_ssh_port` 参数来配置主机的默认登陆端口，该参数默认值为 22。以下示例通过 OpenAPI 修改 `config_default_ssh_port` 参数的值，从而修改主机的默认登陆端口：

1. 登录 TiUniManager 获取用户 Token。

```
curl -X 'POST' \ 'http://172.16.6.206:4180/api/v1/user/login' \ -H 'accept: application/json
    ↪ ' \ -H 'Content-Type: application/json' \ -d '{ "userName": "admin", "userPassword":
    ↪  "admin" }'
```

> **注意：**
> 你需要将以上命令中 172.16.6.206:4180 替换为实际环境中 TiUniManager 中控机的 IP 地址和 WebServer 服务端口。

2. 查看 config_default_ssh_port 的参数值。

```
curl -X GET "http://172.16.6.206:4100/api/v1/config/?configKey=config_default_ssh_port" -H "
    ↪ Authorization: Bearer 6ea768e4-c0ec-4d48-b401-0f114ddc994c"
```

3. 将 config_default_ssh_port 的值修改为 newValue。

```
curl -X POST "http://172.16.6.206:4100/api/v1/config/update" -d "{ \"configKey\": \"
    ↪ config_default_ssh_port\", \"configValue\": \"newValue\"}" -H "Authorization: Bearer
    ↪  6ea768e4-c0ec-4d48-b401-0f114ddc994c"
```

操作步骤

1. 进入资源管理页面。
2. 点击导入主机按钮。
3. 点击下载主机模板按钮。
4. 双击打开模板，并按照模板填入相应信息并保存。
5. 点击上传按钮，选择步骤 5 中编辑的文件，并上传该文件。
6. 点击确认按钮，确认导入主机信息。

主机模板字段说明

Hostname	主机名
IP	主机 IP 地址
Login username	主机用户名（选填，当通过用户名密码方式导入主机时，才需要填写）
Login password	主机密码（选填，当通过用户名密码方式导入主机时，才需要填写）
Region	区域 ID (产品初始化时设定的 Region ID）
Zone	可用区 ID (产品初始化时设定的 Zone ID）
Rack	机架 ID （产品初始化时设定的 Rack ID）
Arch	体系架构
OS Version	操作系统版本
Kernel	内核版本
vCPU	CPU 核数
Memory	内存大小

Hostname	主机名
NIC	网卡规格
Cluster Purpose	区分主机用来部署什么类型的集群，包括 TiDB、 DM、 TiUniManager
Host Purpose	主机用途：用来区分部署的组件类型，包括：Compute、Storage、Schedule 三种用途。多种用途以逗号 “,” 连接，例如 ‘Compute,Storage,Schedule’
Disk Type	磁盘类型，包括：NVMe SSD、 SSD、 SATA 三种类型
Disks	磁盘信息，包括磁盘名称、容量、状态、路径，示例：{“name”: “sda”,“capacity”: 256,“status”: “Available”,“path”: “/mnt/sda”}

如果通过用户名密钥方式导入主机，需要在 TiUniManager 安装前配置用户名和密钥路径，参见指定 TiUniManager 中控机登录 TiDB 资源机的账户和密钥。

13.14.4.2.2 查看主机列表

查看 TiUniManager 当前管理的所有主机信息列表。

操作步骤如下：

1. 登录控制台。
2. 进入主机管理页面，在该页面即可看到主机列表。

13.14.4.2.3 删除主机

从 TiUniManager 主机资源池中删除主机。

操作步骤如下：

1. 登录控制台。
2. 进入资源管理页面。
3. 点击主机条目的删除。
4. 输入 “delete” 以确认删除操作。
5. 点击确认删除按钮。

13.14.4.3 TiUniManager 集群管理

本文档介绍如何通过 TiUniManager 对 TiDB 集群进行操作和管理。

13.14.4.3.1 创建集群

TiUniManager 部署完成后，你可以通过 TiUniManager 创建 TiDB 集群，并自定义集群配置。

进行操作前，确保以下条件已满足：

- 已登录 TiUniManager 控制台
- 已完成主机资源导入

创建集群的操作步骤如下：

1. 登录控制台。
2. 进入集群管理 > 集群页面。
3. 点击创建集群按钮，跳转至创建实例的页面。
4. 选择集群创建模式。
5. 选择集群主机所在厂商、区域。
6. 输入以下数据库基本信息：
    - 数据库类型
    - CPU 体系架构
    - 数据库版本
    - 参数组
7. 设置数据库产品各组件的以下配置：
    - 所在可用区
    - 实例规格
    - 实例数量
8. 输入集群的以下基本信息：
    - 集群名称。集群名称必须是 4-64 个字符，可包含大小写字母、数字和连字符，并以字母或数字开头
    - 集群标签
    - 数据库管理员 Root 的密码。密码必须是 8-64 个字符，可包含大小写字母、数字和可见的特殊字符（包括 `!@#$%^&*()_+=`）
    - 是否独占部署
9. 点击提交按钮，确认主机资源库存满足集群要求后，点击确认创建按钮。

13.14.4.3.2 删除集群

由于业务或其他原因，如果你不再需要某个已创建的实例，可以将某个集群删除。

进行操作前，确保以下条件已满足：

- 已登录 TiUniManager 控制台。
- 待删除的 TiDB 集群已存在。

> **注意：**
>
> 删除集群同时将删除集群上自动备份的数据。如果需要某个备份数据，应在删除实例之前将该备份恢复到新实例上。

删除集群的操作步骤如下：

1. 登录控制台。
2. 进入集群管理 > 集群页面。
3. 选择待删除的集群，点击集群 ID 进入集群详情页面。

4. 点击删除按钮。
5. 选择是否在删除集群前完成一次数据备份。
6. 选择是否保留手动备份数据。
7. 输入 “delete” 以确认删除，点击确认删除按钮。

13.14.4.3.3 扩容集群

集群运行过程中，如果由于业务或其他需求，你需要为集群添加计算引擎、存储引擎或调度引擎。

进行操作前，确保以下条件已满足：

- 已登录 TiUniManager 控制台。
- 集群已创建。

> 注意：
>
> - 集群配置由平台配置所限定。
> - 推荐的 PD 实例数量为 1 个、3 个、5 个或 7 个。
> - TiKV 实例数量不能低于 TiKV 副本数。

扩容的操作步骤如下：

1. 登录控制台。
2. 进入集群管理 > 集群页面。
3. 选择待扩容的集群，点击集群 ID 进入集群详情页面。
4. 点击扩容按钮进入扩容页面。
5. 根据业务需要，选择要扩容引擎数量或新增不同规格的引擎数量。
6. 点击提交按钮，确认资源库存满足扩容要求后，点击确认扩容。

13.14.4.3.4 缩容集群

在集群运行过程中，由于业务或其他需求，你需要从集群中删除多余的计算引擎、存储引擎或调度引擎。

进行操作前，确保以下条件已满足：

- 已登录 TiUniManager 控制台。
- 集群已创建。

> 注意：
>
> - 集群配置由平台配置所限定。
> - 当 TiDB、 PD、 TiKV 的实例数量最小为 1。当实例数等于 1 时，不能进行缩容操作。

- 推荐的 PD 实例数量为 1 个、3 个、5 个或 7 个。
- TiKV 实例数量不能小于 TiKV 副本数。对 TiKV 进行缩容时，缩容后的 TiKV 实例数量不能小于 TiKV 副本数。

缩容集群的操作步骤如下：

1. 登录控制台。
2. 进入集群管理 > 集群页面。
3. 选择待删除的集群，点击集群 ID 进入集群详情页面。
4. 点击待删除实例所在行的删除按钮。
5. 点击确认按钮。

13.14.4.3.5 查看集群列表

查看实例的基本信息、运行情况，并使用集群删除按钮按需删除集群。

查看集群列表的操作步骤如下：

1. 登录控制台。
2. 进集群管理 > 集群页面即可查看集群列表，若暂无集群，则列表为空。

13.14.4.3.6 搜索集群

在集群列表页面，你根据以下指标搜索集群：

- 集群运行状态（可选择）
- 集群 ID（需要输入集群 ID，暂不支持模糊搜索）
- 集群名称（需要输入集群名称，暂不支持模糊搜索）
- 标签（需要输入标签，暂不支持模糊搜索）

搜索集群前，确保以下条件已满足：

- 已登录平台控制台。
- 集群已创建。

搜索集群的操作步骤如下：

1. 进入集群详情页面。
2. 选择搜索指标。
3. 点击搜索按钮，确认搜索，集群详情列表展示搜索结果。

13.14.4.3.7 查看集群详情

你可通过 TiUniManager 查看集群配置、运行情况、计算、存储、调度等各引擎列表及信息详情。

进行操作前，确保以下条件已满足：

- 已登录 TiUniManager 控制台。
- 集群已创建。

查看集群详情的操作步骤如下：

1. 进入集群管理 > 集群页面。
2. 点击集群 ID 进入集群详情页面。

13.14.4.3.8 性能分析

你可通过 TiUniManager 查看集群的 SQL 语句分析、慢 SQL 查询分析、流量可视化、集群诊断。

进行操作前，确保以下条件已满足：

- 已登录 TiUniManager 控制台。
- 集群已创建。

性能分析的操作步骤如下：

1. 进入集群管理 > 集群页面
2. 点击集群 ID 进入集群详情页面，再点击性能分析标签页，你可看到 SQL 语句分析、慢查询日志、流量可视化、诊断四个子标签页。
3. 点击语句分析页签，进入 SQL 语句分析标签页。了解SQL 语句分析详情。
4. 点击慢查询分析页签，进入慢查询分析标签页。了解慢查询分析详情。
5. 点击流量可视化页签，进入流量可视化标签页。了解流量可视化详情
6. 点击集群诊断页签，进入集群诊断标签页。了解集群诊断 详情。

13.14.4.3.9 日志管理

在集群详情页面，你可以查看该集群各组件日志，也可以选择时间范围进行日志查询，或按照日志级别、实例 IP 地址查看日志。

进行操作前，确保以下条件已满足：

- 已登录 TiUniManager 控制台。
- 集群已创建。

日志管理的操作步骤如下：

1. 进入集群管理 > 集群页面。
2. 点击集群 ID 进入集群详情页面。

3. 点击日志管理页签，进入日志管理标签页，默认以时间倒序排序展示集群日志，并可按照以下维度进行日志过滤：
    - 按时间范围过滤
    - 按组件（计算、存储、调度）过滤
    - 按日志级别过滤
    - 按日志内容关键词过滤
    - 按实例 IP 地址过滤

13.14.4.3.10 集群监控

在 TiUniManager 中，你可通过集成的 Grafana 面板查看集群整体、组件、节点实例的运行情况（组件分别展示：PD、TiDB、TiKV）。

进行操作前，确保以下条件已满足：

- 已登录 TiUniManager 控制台。
- 集群已创建。

查看集群监控的操作步骤如下：

1. 进入集群管理 > 集群页面。
2. 点击集群 ID 进入集群详情页面，并点击集群监控页签。
3. 在集群监控页签中，默认显示 TiDB-Summary Dashboard。
4. 可通过选择 Dashboard 一级目录，列出所有可用 Dashboard。

13.14.4.3.11 告警管理

TiUniManager 集成了 TiDB 集群组件的报价规则，按照严重程度由高到低罗列。报警项可分为紧急级别 > 严重级别 > 警告级别三类。该分级适用于以下各组件的报警项。

TiDB 集群告警规则可参见在线文档：TiDB 集群报警规则

进行告警管理前，确保以下条件已满足：

- 已登录 TiUniManager 控制台。
- 集群已创建。

告警管理的操作步骤如下：

1. 进入集群管理 > 集群页面。
2. 点击集群 ID 进入集群详情页面，并点击告警管理页签。

13.14.4.3.12 参数管理

你可通过 TiUniManager 查看集群实例的参数信息，并修改集群的参数运行值。

进行操作前，确保以下条件已满足：

- 已登录 TiUniManager 控制台
- 集群已创建

参数管理的操作步骤如下：

1. 进入集群管理 > 集群页面。
2. 点击集群 ID 进入集群详情页面，并点击参数管理页签。
3. 选择要修改的参数项，点击编辑进行修改，完成后点击确认。
4. 点击保存按钮，使参数运行值生效。

13.14.4.3.13 备份管理 - 数据备份

TiUniManager 提供自动定期备份功能。集群创建成功并开始正常运行后，你可对集群设置定期备份时间，TiUniManager 会对集群数据进行定时备份，备份的数据将保存到你所设置的存储位置中（S3 兼容存储和 NFS 共享存储）。你也可以逐条删除不需要的备份数据文件。

进行操作前，确保以下条件已满足：

- 已经登录 TiUniManager 控制台。
- 集群已创建。

> 注意：
>
> - 备份保存的默认文件夹不可修改。

TiUniManager 默认的集群备份路径相关配置参数如下：

配置描述	配置参数名	参考值
TiDB 集群备份的存储类型（仅支持 NFS 或 S3）	BackupStorageType	‘s3’ 或 ‘nfs’
TiDB 集群备份的存储路径（S3 bucket 路径，或 NFS share 的绝对路径）	BackupStoragePath	‘bucketPath/backup’ (S3 路径示例） ‘/mnt/nfspath’ (NFS 路径示例）
TiDB 集群备份在 S3 共享存储时，S3 的 AccessKey	BackupS3AccessKey	”
TiDB 集群备份在 S3 共享存储时，S3 的 SecretAccessKey	BackupS3SecretAccessKey	
TiDB 集群备份在 S3 共享存储时,S3 的 Endpoint（域名）	BackupS3Endpoint	”

当前不支持通过 TiUniManager 界面修改备份路径。如需修改备份路径，需要通过 OpenAPI 修改配置参数，以修改配置参数 BackupS3AccessKey 为例：

1. 登录 TiUniManager 获取 user token。

```
curl -X 'POST' \ 'http://172.16.6.206:4180/api/v1/user/login' \ -H 'accept: application/json
    ↪ ' \ -H 'Content-Type: application/json' \ -d '{ "userName": "admin", "userPassword":
    ↪  "admin" }'
```

> 注意：
> 你需要将以上命令中 172.16.6.206:4180 替换为实际环境中 TiUniManager 中控机的 IP 地址和 WebServer 服务端口。

2. 查看配置参数值。

```
curl -X GET "http://172.16.6.206:4100/api/v1/config/?configKey=BackupS3AccessKey" -H "
    ↪ Authorization: Bearer 6ea768e4-c0ec-4d48-b401-0f114ddc994c"
```

3. 修改配置参数值。

```
curl -X POST "http://172.16.6.206:4100/api/v1/config/update" -d "{ \"configKey\": \"
    ↪ BackupS3AccessKey\", \"configValue\": \"newValue\"}" -H "Authorization: Bearer 6
    ↪ ea768e4-c0ec-4d48-b401-0f114ddc994c"
```

数据备份的操作步骤如下：

1. 进入集群管理 > 集群页面。
2. 点击集群 ID 进入集群详情页面，并点击备份管理页签。
3. 点击备份计划按钮，配置自动备份策略。
4. 点击手动备份按钮，立即开始手动备份。
5. 选择备份并点击删除按钮，确认删除选择的备份。
6. 查询某时间段的备份：
    - 默认以时间倒序排序展示备份
    - 查询某时间段的备份

13.14.4.3.14 备份管理 - 数据恢复

使用 TiUniManager 的备份数据恢复功能，将一个 TiDB 集群的备份数据恢复到一个新集群。

进行操作前，确保以下条件已满足：

- 已经登录控制台。
- 集群的备份记录已存在。

> 注意：
> 备份数据恢复到新创建的集群上，不会覆盖已创建实例。

数据恢复的操作步骤如下：

1. 进入集群管理 > 集群页面。
2. 点击集群 ID 进入集群详情页面，并点击备份管理页签。
3. 选择要恢复的备份记录，点击恢复，跳转恢复集群页面。
4. 设置新集群的名称、标签和数据库密码。
5. 选择新集群的 Region 区域。
6. 确认新集群的实例配置规格与数量。
7. 点击提交按钮，确认创建新集群并随后开始自动恢复实例。

13.14.4.3.15 数据同步

通过 TiUniManager，你可以使用 TiDB 增量数据同步功能，从配置了 TiCDC 组件的上游 TiDB 集群，将增量数据同步到下游系统（TiDB 数据库、MySQL 数据库或 Kafka）。

进行操作前，确保以下条件已满足：

- 已经登录控制台。

> 注意：
>
> 数据同步功能仅支持 TiDB v5.2.2 及其以上版本。

数据同步的操作步骤如下：

1. 进入集群管理 > 集群页面。
2. 点击集群 ID 进入集群详情页面，并点击数据同步页签。
3. 点击创建按钮创建数据同步任务。
4. 填写数据同步任务基本信息：
    - 任务名称
    - 同步任务开始 TSO 值（填入“0”则系统将从当前时刻获取起始 TSO）
    - 过滤规则（参见表库过滤语法）
5. 填写下游信息：
    - 下游类型（TiDB、MySQL、Kafka)
    - 下游 URL 或 IP 地址
    - 下游数据库服务端口
    - 下游数据库用户名
    - 下游数据库密码
    - 同步任务并发数
6. 点击提交按钮。

13.14.4.3.16 查看参数组

TiUniManager v1.0.0 为 TiDB v5.1.x、v5.2.x、v5.3.x、v5.4.x 提供内置的默认参数组（暂不支持 TiDB v6.0 参数组），你可查看参数组下的参数详情。

进行操作前，确保以下条件已满足：

- 已经登录控制台。

查看参数组步骤如下：

1. 进入集群管理 > 参数组页面，查看所有的参数组列表。
2. 点击参数组名称，查看参数组中详情：参数类别、参数名称、参数列表、默认值、参数描述。

13.14.4.3.17 编辑、复制、应用参数组

TiUniManager 为 TiDB v5.1.x、v5.2.x、v5.3.x、v5.4.x 提供内置的默认参数组，你可直接应用默认参数组到集群实例，也可复制默认参数组并根据实际运行情况对参数进行调优。

进行操作前，确保以下条件已满足：

- 已经登录控制台。

编辑、复制、应用参数组的操作步骤如下：

1. 进入集群管理 > 参数组页面。
2. 点击待修改的参数组对应的编辑操作，对参数组默认值进行修改。
3. 点击待复制的参数组对应的复制操作，将参数组复制产生一份新的参数组。
4. 点击参数组对应的应用操作，可将参数组应用到适配的集群。

13.14.4.3.18 接管集群

你可通过 TiUniManager 接管已有的 TiDB 集群。

进行操作前，确保以下条件已满足：

- 已经登录控制台。

接管集群的操作步骤如下：

1. 进入集群管理 > 集群页面。
2. 点击接管集群按钮，进入接管集群页面。
3. 输入接管集群的基本信息：集群名称、数据库用户名 root、数据库密码。
4. 输入接管集群中控配配置信息：
    - 接管集群中控机主机 IP 地址
    - 接管集群中控机 SSH 端口号
    - 接管集群中控机 SSH 用户名
    - 接管集群中控机 SSH 密码
    - 接管集群中控机 TiUP 路径（即 `.tiup` 目录所在路径，不含结尾的 `/`，例如 `/root/.tiup` ）
5. 导入接管集群的主机（导入主机流程参见导入主机）。

13.14.4.3.19 重启集群

你可在 TiUniManager 上重启指定的 TiDB 集群。

进行操作前，确保以下条件已满足：

- 已经登录控制台。

重启集群的操作步骤如下：

1. 进入集群管理 > 集群页面。
2. 点击待重启集群的重启操作。
3. 点击确认按钮开始重启集群。

13.14.4.3.20 停止集群

你可在 TiUniManager 上停止指定的 TiDB 集群。

进行操作前，确保以下条件已满足：

- 已经登录 TiUniManager 控制台。

停止集群的操作步骤如下：

1. 进入集群管理 > 集群页面。
2. 选择集群，点击停止。
3. 点击确认按钮，开始停止集群。

13.14.4.3.21 克隆集群

你可通过 TiUniManager 从一个 TiDB 集群克隆出一个新集群。新集群与源集群之间数量关系可配置为：

- 全量：克隆集群并与源集群建立主备关系，主集群实时向备集群同步数据
- 快照：克隆集群，但不建立主备关系

进行操作前，确保以下条件已满足：

- 已经登录 TiUniManager 控制台。
- 已有 TiDB 集群。

> 注意：
>
> 克隆集群功能仅支持 TiDB v5.2.2 及其以上版本。

克隆集群的操作步骤如下：

1. 进入集群管理 > 集群页面。
2. 选择要产生克隆集群的源集群，进入集群详情页面。
3. 点击克隆集群按钮。
4. 选择克隆策略。
5. 填写新集群配置信息。
6. 点击提交按钮。

13.14.4.3.22 切换主备集群

你可通过 TiUniManager 选择一个备集群，并将其切换为主集群，此时集群的原主集群将自动被切换为备集群。

进行操作前，确保以下条件已满足：

- 当前 TiDB 集群为备集群。

> 注意：
> 主备集群切换功能仅支持 TiDB v5.2.2 及其以上版本。

切换的操作步骤如下：

1. 进入集群管理 > 集群页面。
2. 选择备集群，进入备集群详情页面。
3. 点击备集群后的切换。
4. 确认备注集群切换信息后点击确认按钮

13.14.4.3.23 原地升级集群

你可通过 TiUniManager 将指定的 TiDB 集群升级为更高版本的集群。

进行操作前，确保以下条件已满足：

- 已经登录 TiUniManager 控制台。

原地升级集群的操作步骤如下：

1. 进入集群管理 > 集群页面。
2. 选择要升级的集群，进入集群详情页面。
3. 点击数据库版本编号后的升级操作，进入集群版本升级页面。
4. 选择目标版本、升级类型、升级方式，点击下一步按钮。
5. 查看对比和合并配置参数，选择参数值为原版本值以及新版本值，点击下一步按钮。
6. 确认合并后的配置参数，点击确认按钮。

13.14.4.3.24 巡检集群

你可通过 TiUniManager 巡检功能检查集群各个维度是否健康，包括集群资源分配是否泄露，Region 是否健康，参数配置是否正常等等。

进行操作前，确保以下条件已满足：

- 已经登录 TiUniManager 控制台

TiUniManager 于每天 23:00 自动进行集群巡检。

查看巡检结果的方法：在工作流任务管理中查看任务名为 “CheckPlatform” 的任务流，点击 checkHosts 步骤可以查看，如下图所示：

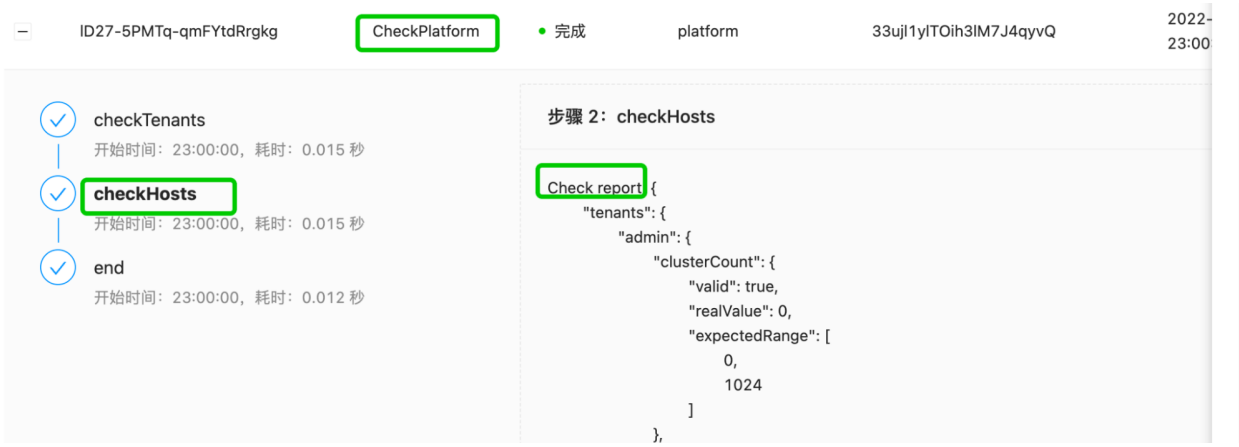


图 299: 集群巡检图

13.14.4.4 TiUniManager 数据导入与导出

本文档介绍如何使用 TiUniManager 从 TiDB 集群导入和导出数据。

13.14.4.4.1 导入数据

DBA 管理员可从 S3 兼容存储、本地存储、或 TiUniManager 共享存储上，导入 SQL 文件、CSV 文件的数据到 TiDB 集群。

TiUniManager 共享存储是用户事先设定好、用于保存导入导出文件的文件资源池，该文件资源池是一个可挂载于 TiUniManager 中控机的 NFS 目录，你可在安装 TiUniManager 后设置该路径。

导入前准备

进行操作前，确保已经登录 TiUniManager 控制台。

> 注意：
>
> - 从本地上传的源数据文件大小不能超过 2 GiB
> - TiUniManager 最多支持 3 个导入导出任务并行运行，后续任务在队列中等待

TiUniManager 默认的导入导出路径是在 TiUniManager 中控机上，细节如下：

配置描述	配置参数名	参考值
TiUniManager 共享存储中导入文件的存储位置（建议配置为 NFS 共享存储）	ImportShareStoragePath	/home/tidb/import（备注：中控机 tidb 账号拥有该路径的读写权限）
TiUniManager 共享存储中导出文件的存储位置（建议配置为 NFS 共享存储）	ExportShareStoragePath	/home/tidb/export（备注：中控机 tidb 账号拥有该路径的读写权限）

当前 TiUniManager 不支持通过前端界面修改导入路径。如需修导入路径，需要通过 OpenAPI 对配置进行修改，以修改 ImportShareStoragePath 为例：

1. 登录获取 user token。

```
curl -X 'POST' \ 'http://172.16.6.206:4180/api/v1/user/login' \ -H 'accept: application/json
    ↪ ' \ -H 'Content-Type: application/json' \ -d '{ "userName": "admin", "userPassword":
    ↪  "admin" }'
```

> 注意：
> 你需要将以上命令中·172.16.6.206:4180 替换为实际环境中 TiUniManager 中控机的 IP 地址和 WebServer 服务端口。

2. 查看配置参数值。

```
curl -X GET "http://172.16.6.206:4100/api/v1/config/?configKey=ImportShareStoragePath" -H "
    ↪ Authorization: Bearer 6ea768e4-c0ec-4d48-b401-0f114ddc994c"
```

3. 修改配置参数值为 newValue（根据实际的环境设置为新的存储路径）。

```
curl -X POST "http://172.16.6.206:4100/api/v1/config/update" -d "{ \"configKey\": \"
    ↪ ImportShareStoragePath\", \"configValue\": \"newValue\"}" -H "Authorization: Bearer
    ↪ 6ea768e4-c0ec-4d48-b401-0f114ddc994c"
```

导入数据操作步骤

1. 进入集群管理 > 导入导出页面。
2. 点击导入数据按钮，进入导入页面。
3. 输入目标集群信息：集群 ID、数据库用户名、数据库密码。
4. 选择源数据。存储位置可以是本地文件或 TiUniManager 共享存储或 S3 兼容存储。
5. 输入本次导入备注信息。
6. 点击创建导入任务开始导入。

13.14.4.4.2 导出数据

DBA 管理员可从 TiDB 集群将数据以 SQL 文件或 CSV 文件格式，导出至 S3 兼容存储、本地存储或 TiUniManager 共享存储，并可对导出数据进行筛选。数据筛选的方式包括：

- 按 SQL 语句对导出数据进行筛选
- 按指定库表对导出数据进行筛选

对于保存在 TiUniManager 共享存储上的导入数据，你可进一步下载至本地。

导出前准备

在导出数据前，确保已经登录 TiUniManager 控制台。

> 注意：
>
> - TiUniManager 最多支持 3 个导入导出任务并行运行，后续任务在队列中等待。
> - 从 TiUniManager 共享存储下载的导出数据文件大小不能超过 2 GiB。

当前 TiUniManager 前端不支持修改导入导出路径，如需修改导入导出路径，需要通过 OpenAPI 对配置进行修改，以修改 ExportShareStoragePath 为例：

1. 登录获取 user token。

```
curl -X 'POST' \ 'http://172.16.6.206:4180/api/v1/user/login' \ -H 'accept: application/json
    ↪ ' \ -H 'Content-Type: application/json' \ -d '{ "userName": "admin", "userPassword":
    ↪  "admin" }'
```

> 注意：
> 你需要将以上命令中·172.16.6.206:4180 替换为实际环境中 TiUniManager 中控机的 IP 地址和 WebServer 服务端口。

2. 查看配置参数值。

```
curl -X GET "http://172.16.6.206:4100/api/v1/config/?configKey=ExportShareStoragePath" -H "
    ↪ Authorization: Bearer 6ea768e4-c0ec-4d48-b401-0f114ddc994c"
```

3. 修改配置参数值为 newValue（根据实际环境设置为新的存储路径）。

```
curl -X POST "http://172.16.6.206:4100/api/v1/config/update" -d "{ \"configKey\": \"
    ↪ ExportShareStoragePath\", \"configValue\": \"newValue\"}" -H "Authorization: Bearer
    ↪ 6ea768e4-c0ec-4d48-b401-0f114ddc994c"
```

导出数据操作步骤

1. 进入集群管理 > 导入导出页面。
2. 点击导出数据按钮，进入导出数据页面。
3. 输入源集群信息：集群 ID、数据库用户名、数据库密码。
4. 选择导出目标位置：TiUniManager 共享存储或 S3 兼容存储。
5. 设置导出选项：导出文件格式，是否筛选数据及筛选条件。
6. 记录本次导出备注信息。
7. 点击创建导出任务开始导出数据。

13.14.4.4.3 查看数据的导入导出记录

你可以通过 TiUniManager 查看、删除数据的导入导出记录，并下载记录文件至本地。

在查看数据的导入导出记录前，确保已经登录 TiUniManager 控制台。

查看数据的导入导出记录的操作步骤如下：

1. 进入集群管理 > 导入导出页面。
2. 查看导入导出记录列表。
3. 选择导出记录的下载，可下载导出记录至本地。
4. 选择记录的删除，可删除导入导出记录。

13.14.4.5 TiUniManager 工作流任务查看

TiUniManager 提供工作流任务页面，你可以从该页面查看在 TiUniManager 控制台发起的用户任务进展，以及分析任务失败的原因。

查看工作流任务列表及详情前，确保已经登录 TiUniManager 控制台。

查看工作流任务列表及详情的步骤如下：

1. 进入工作量任务管理页面查看任务列表，页面以时间倒序排序展示集群工作量任务。
2. 选择要查看详情的任务，点击 + 图标，可进一步查看任务包含的详细步骤和执行结果。

13.14.4.6 TiUniManager 系统管理

本文档介绍如何通过 TiUniManager 管理系统。

13.14.4.6.1 查看系统监控

你可能需要查看 TiUniManager 系统监控，了解 TiUniManager 系统运行情况、TiUniManager 所在主机运行情况。

操作步骤如下：

1. 登录 TiUniManager 控制台。
2. 进入系统管理 > 系统监控页面。
3. 选择 EM Server，查看 TiUniManager 系统运行情况的 Dashboard。
4. 选择 Node Exporter，查看 TiUniManager 所在主机运行情况的 Dashboard。

13.14.4.6.2 查看系统日志

查看 TiUniManager 系统日志的步骤如下：

1. 登录 TiUniManager 控制台。
2. 进入系统管理 > 系统日志页面查看系统日志。

13.14.4.6.3 TiUniManager 告警规则

TiUniManager 默认包含以下告警规则，以便接受相应的告警通知。

work flow error alert

- 报警规则：`sum(increase(em_work_flow_total{flow_status="Error"}[5m]))by(service, biz_type, flow_name, flow_status)> 3`
- 规则描述：错误状态工作流数量超过 3 时告警。
- 处理方法：可能是逻辑问题。查看失败工作流，或者联系 TiUniManager 开发人员。

work flow node error alert

- 报警规则：`sum(increase(em_work_flow_node_total{flow_node_status="Error"}[5m]))by(service, biz_type, flow_name, flow_node, flow_node_status)> 3`
- 规则描述：错误状态工作流节点超过 3 时告警。
- 处理方法：可能是逻辑问题。查看失败工作流，或者联系 TiUniManager 开发人员。

cluster-server request error rate

- 报警规则：`sum(increase(em_micro_requests_total{service="cluster-server", code!="0"}[1m])/ increase(em_micro_requests_total{service="cluster-server"}[1m]))by(service, method)> 0.05`
- 规则描述：接口失败率超过 5% 时告警。
- 处理方法：可以查看系统日志或系统追踪，或者联系 TiUniManager 开发人员。

openapi-server request error rate

- 报警规则：`sum(increase(em_http_requests_total{service="openapi-server", code!="200"}[1m]) / increase(em_http_requests_total{service="openapi-server"}[1m]))by(service, handler, method)> 0.05`
- 规则描述：接口失败率超过 5% 时告警。
- 处理方法：可以查看系统日志或系统追踪，或者联系 TiUniManager 开发人员。

13.14.4.6.4 TiUniManager 告警设置

TiUniManager 告警设置支持钉钉、Email 等告警通道，具体见 List of notifiers supported by Grafana。在设置 TiUniManager 告警前，确保已登录 TiUniManager 控制台。

以下示例基于 Grafana v7.5.15，展示如何在 Grafana 上配置钉钉的告警通道。

1. 打开 Notification channels 配置页面，点击 New channel 创建通道。

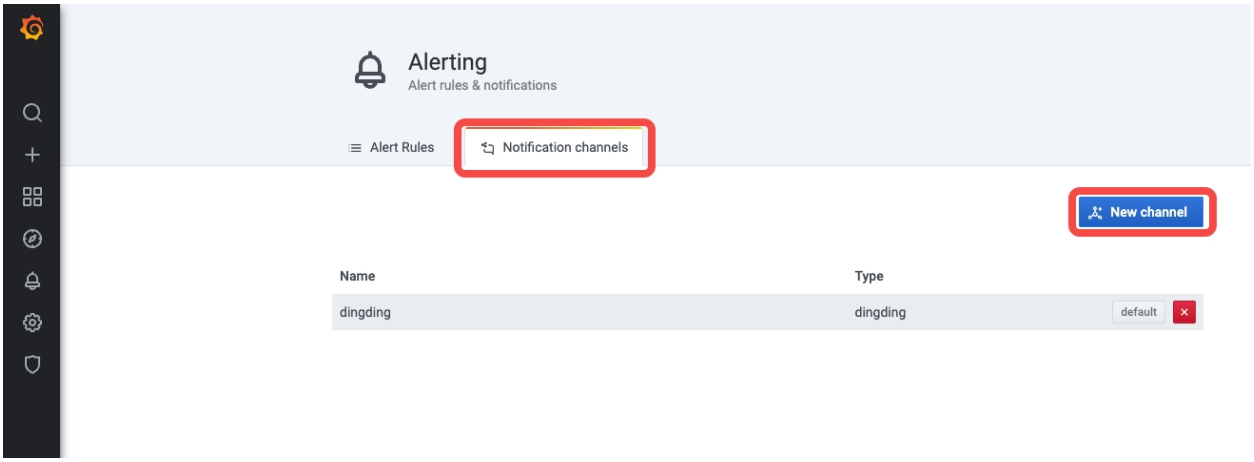


图 300: Notification channels - New channel

2. 配置消息接收方，示例如下：

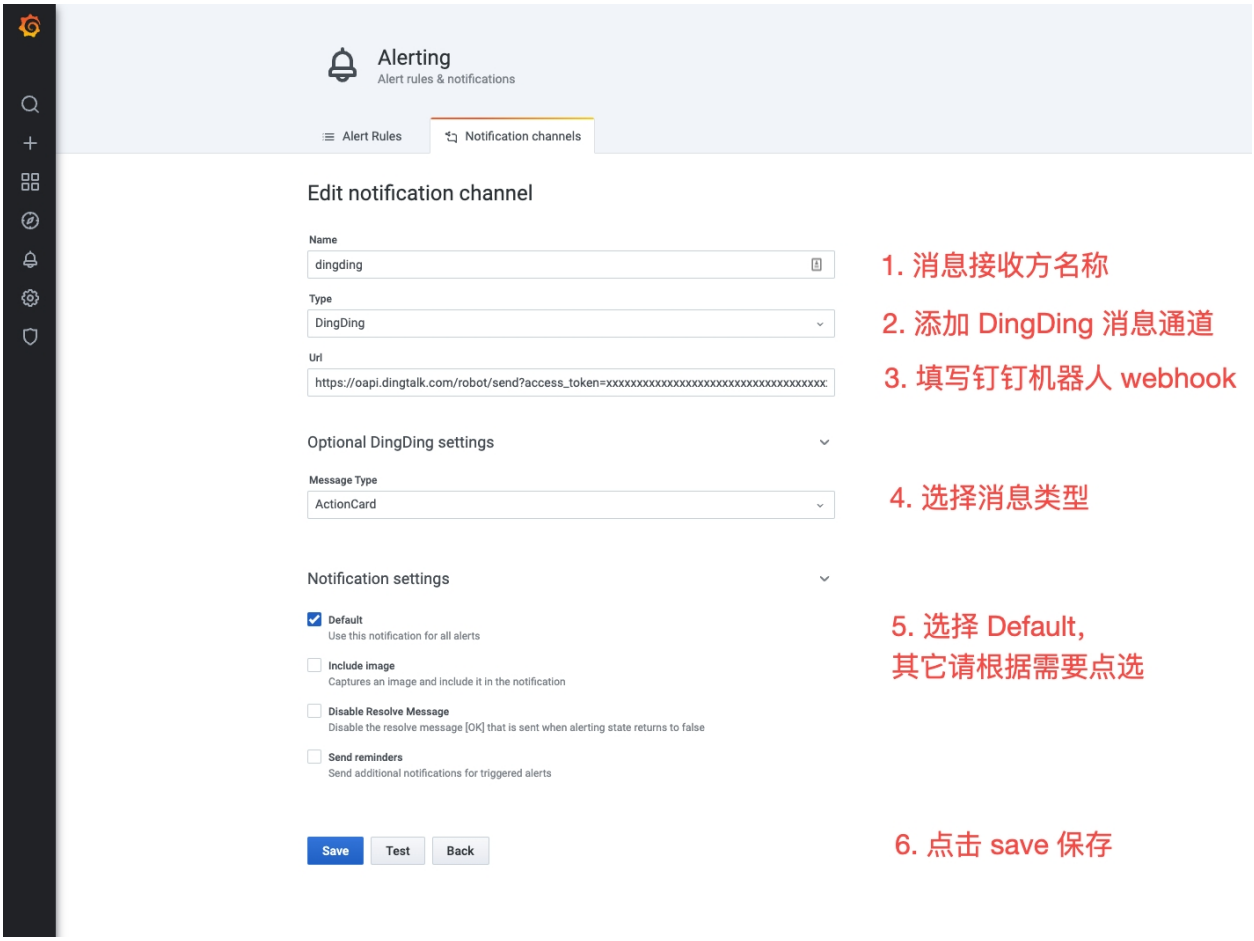


图 301: Notification channels - Edit config

若要接入钉钉自定义机器人，参考钉钉文档 - 自定义机器人接入。

13.14.5 TiUniManager 常见问题

本文档介绍 TiUniManager 的常见问题，帮助用户更好了解 TiUniManager。

13.14.5.1 产品基本 FAQ

13.14.5.1.1 TiUniManager 管理的集群可以部署在哪里？

TiUniManager 管理的集群可以部署在用户自建的数据中心或公有云。

13.14.5.1.2 TiUniManager 面向哪几类用户？

TiUniManager 面向三类用户：IT 系统管理员、数据库管理员 DBA、数据库应用开发人员。

13.14.5.1.3 TiUniManager 支持的 TiDB 版本有哪些？

见TiUniManager 支持的 TiDB 版本。

13.14.5.1.4 TiUniManager 与 TiUP 的关系是什么？

TiUniManager 使用 TiDB 及其生态工具、API 提供数据库管理产品功能，TiUP 是 TiUniManager 使用的工具之一。TiUniManager 产品架构图如下：

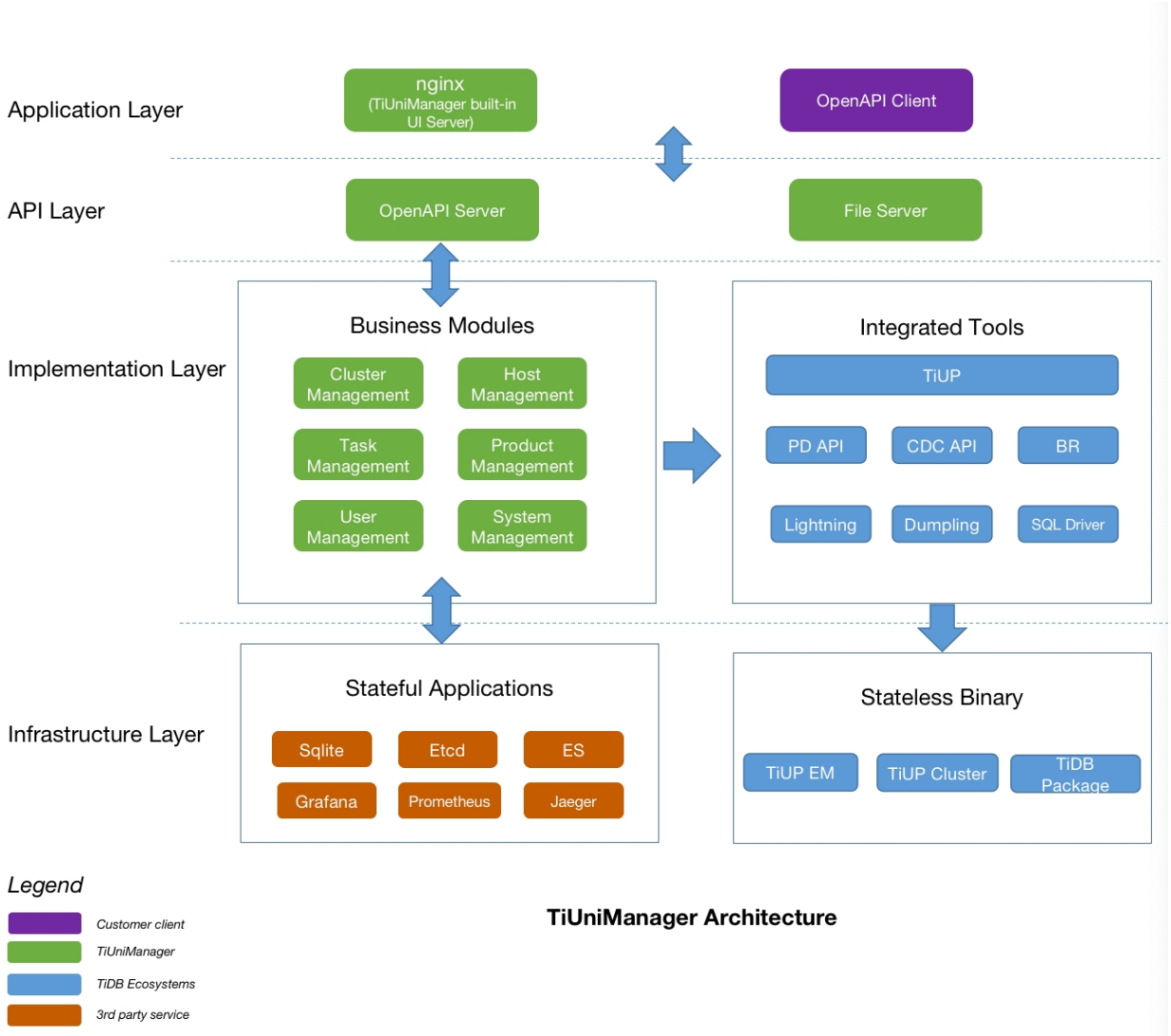


图 302: TiUniManager 架构图

13.14.5.2 主机资源管理 FAQ

13.14.5.2.1 何时导入主机资源到 TiUniManager？

在创建新集群或导入现有集群前，你需要将集群所需的主机资源导入到 TiUniManager。具体步骤参见TiUniManager 资源管理。

13.14.5.3 TiUniManager 安装部署 FAQ

13.14.5.3.1 TiUniManager 支持离线安装吗？

支持

13.14.5.3.2 TiUniManager 首次部署后，需要如何做产品初始化吗？

TiUniManager 首次部署后，需要对 TiDB 组件规格、数据中心 Zone 信息、主机规格信息进行初始化。

13.14.5.3.3 如何在命令行查看 EM 服务和 EM 部署的 TiDB 集群信息？

首先请先通过 `su - tidb` 命令切换到 `tidb` 账号下。

所有与 EM 工具相关的命令，请以 `TIUP_HOME=/home/tidb/.em tiup em <cmd>` 的方式执行，例如 `TIUP_HOME=/home/tidb/.em tiup em list`。

所有与 EM 部署的 TiDB 集群相关的命令，请以直接以 `tiup cluster <cmd>` 的方式执行，例如 `tiup cluster list`。

13.14.5.4 集群部署 FAQ

13.14.5.4.1 TiUniManager 支持在离线环境中部署 TiDB 集群吗？

支持

13.14.5.4.2 TiKV 实例数与 TiKV 的关系？

创建集群时，TiKV 副本数建议为 1 个、3 个、5 个、或 7 个。设置的 TiKV 实例数不能小于 TiKV 副本数。

13.14.5.4.3 建议的 PD 实例数量？

建议的 PD 实例数量为 1 个、3 个、5 个、或 7 个。

13.14.5.5 备份与恢复 FAQ

13.14.5.5.1 备份支持的存储类型有哪些？

TiUniManager 支持将数据备份至 S3 兼容存储和 NFS 共享存储。

13.14.5.5.2 如何设置 TiUniManager 的备份路径？

TiUniManager 目前不支持通过前端界面修改备份路径，修改备份路径方法参考TiUniManager 备份管理。

13.14.5.6 数据导入与导出 FAQ

13.14.5.6.1 可以从哪些上游导入数据到 TiUniManager？

你可以从 S3 兼容存储、本地存储、或 TiUniManager 共享存储上导入集群。

13.14.5.6.2 导出的数据可以保存在哪些存储上？

导出的数据可以保存至 S3 兼容存储或 TiUniManager 共享存储，对于保存在 TiUniManager 共享存储上的导入数据，你可进一步下载至本地。

13.14.5.6.3 从本地上传的导入文件大小有限制吗？

从本地上传的源数据文件大小不能超过 2 GB。

13.14.5.6.4 什么是 TiUniManager 共享存储？

TiUniManager 共享存储是用户事先设定好用于保存导入导出文件的文件资源池，该文件资源池是一个可挂载于 TiUniManager 中控机的 NFS 目录，该 NFS 路径由用户在 TiUniManager 安装后进行设置。

13.14.5.6.5 如何修改 TiUniManager 共享存储路径？

参见TiUniManager 数据导入与导出。

13.14.5.7 集群扩容缩容 FAQ

13.14.5.7.1 对组件实例进行缩容时有何限制？

TiDB、PD、TiKV 的实例数量最小为 1。当实例数等于 1 时，不能进行缩容操作。

13.14.5.7.2 TiKV 实例数与 TiKV 副本数有何关系？

TiKV 实例数量不能低于 TiKV 副本数。对 TiKV 进行缩容时，缩容后的 TiKV 实例数量不能小于 TiKV 副本数。

13.14.5.8 接管集群 FAQ

13.14.5.8.1 TiUniManager 接管集群前有什么注意事项吗？

建议用户在 TiUniManager 接管集群前，对集群进行一次数据备份。

13.14.5.8.2 接管集群时需要用户填写的“TiUP 部署路径”是什么？

“TiUP 部署路径”是被接管集群中 TiUP 的 home 路径。填写该路径时，请勿填写路径结尾的 /。

例如，请勿填写为 `/root/.tiup/`。

13.14.6 发布版本历史

13.14.6.1 TiUniManager 版本发布历史

TiUniManager 历史版本发布声明如下：

- v1.0.2
- v1.0.1
- v1.0.0

13.14.6.1.1 TiUniManager 支持的 TiDB 版本

当前 TiUniManager 仅支持 x86_64 版本的 TiDB。

TiUniManager 版本	支持的 TiDB 版本（仅 x86_64）
v1.0.2	v6.1.0、v6.0.0、v5.4.0、v5.3.0、v5.3.1、v5.2.0、v5.2.1、v5.2.2、v5.2.3、v5.1.0、v5.1.1、v5.1.2、v5.1.3、v5.1.4、v5.0.0、v5.0.1、v5.0.2、v5.0.3、v5.0.4、v5.0.5、v5.0.6
v1.0.1	v6.0.0、v5.4.0、v5.3.0、v5.3.1、v5.2.0、v5.2.1、v5.2.2、v5.2.3、v5.1.0、v5.1.1、v5.1.2、v5.1.3、v5.1.4、v5.0.0、v5.0.1、v5.0.2、v5.0.3、v5.0.4、v5.0.5、v5.0.6
v1.0.0	v5.4.0、v5.3.0、v5.3.1、v5.2.0、v5.2.1、v5.2.2、v5.2.3、v5.1.0、v5.1.1、v5.1.2、v5.1.3、v5.1.4、v5.0.0、v5.0.1、v5.0.2、v5.0.3、v5.0.4、v5.0.5、v5.0.6

13.14.6.2 TiUniManager v1.0.2 Release Notes

发布日期：2022 年 6 月 16 日

发布版本：1.0.2

自 v1.0.2 起，产品名由 TiDB Enterprise Manager 改为 TiUniManager，并正式开放源代码。在v1.0.1 版的基础上，v1.0.2 有如下更新：

- 新增对 TiDB v6.1.0 集群的支持
- 增加修改密码功能，首次安装后必须修改密码才能完成登录

TiUniManager v1.0.2 支持的 TiDB 版本如下：

- v6.1.0
- v6.0.0
- v5.4.0
- v5.3.0、v5.3.1
- v5.2.0、v5.2.1、v5.2.2、v5.2.3
- v5.1.0、v5.1.1、v5.1.2、v5.1.3、v5.1.4
- v5.0.0、v5.0.1、v5.0.2、v5.0.3、v5.0.4、v5.0.5、v5.0.6

13.14.6.3 TiDB Enterprise Manager v1.0.1 Release Notes

发布日期：2022 年 5 月 30 日

发布版本：1.0.1

> 注意：
>
> - TiDB Enterprise Manager v1.0.1 未开源。若要获取 v1.0.1 的安装包或相关支持，请联系 TiUniManager 团队。
> - 自 v1.0.2 起，TiDB Enterprise Manager 更名为 TiUniManager。

该版本在 TiDB Enterprise Manager 1.0.0 的基础上，进行了以下更新：

- 新增对 TiDB v6.0.0 集群的支持
- 优化 workflow 逻辑，支持重启继续执行
- 修复一些安全基线漏洞，提升了 TiDB Enterprise Manager 安全性
- 替换 Elasticsearch、Filebeat、Kibana、Grafana 为 OSS 版本

TiDB Enterprise Manager v1.0.1 支持的 TiDB 版本如下：

- v6.0.0
- v5.4.0

- v5.3.0、v5.3.1
- v5.2.0、v5.2.1、v5.2.2、v5.2.3
- v5.1.0、v5.1.1、v5.1.2、v5.1.3、v5.1.4
- v5.0.0、v5.0.1、v5.0.2、v5.0.3、v5.0.4、v5.0.5、v5.0.6

13.14.6.4 TiDB Enterprise Manager v1.0.0 Release Notes

发布日期：2022 年 4 月 7 日

TiDB Enterprise Manager **版本：**1.0.0

> **注意：**
>
> - TiDB Enterprise Manager v1.0.0 未开源。若要获取 v1.0.0 的安装包或相关支持，请联系 TiUniManager 团队。
> - 自 v1.0.2 起，TiDB Enterprise Manager 正式更名为 TiUniManager。

TiDB Enterprise Manager 是一款以 TiDB 数据库为核心的数据库管理平台，帮助用户在本地部署环境或公有云环境中管理 TiDB 集群。

TiDB Enterprise Manager 不仅提供对 TiDB 集群的全生命周期的可视化管理，也同时一站式提供 TiDB 数据库参数管理、数据库版本升级、克隆集群、主备集群切换、数据导入导出、数据同步、数据备份恢复服务，能有效提高 TiDB 集群运维效率，降低企业运维成本。

13.14.6.4.1 用户价值

分布式数据库管理面临集群规模化、管理场景多元化挑战，传统的命令行管理方式面临部署成本高、技术门槛高、维护成本高等诸多挑战，TiDB Enterprise Manager 能有效解决以下场景中的分布式数据库管理难题：

- 主机资源管理
- 部署集群
- 统一入口管理多套集群
- 升级集群
- 参数管理
- 备份恢复
- 数据导入导出
- 安全与合规
- 自动化与第三方集成
- 管理任务历史记录可查可追溯

13.14.6.4.2 主要功能

集群主机资源管理

- 统一构建 TiDB 集群 IT 基础设施资源池，主机资源（vCPU、内存、磁盘）总量与分配用量一目了然。

- 支持按集群类型、组件类型对主机资源分类，让最合适的主机资源匹配不同 TiDB 组件，提升主机资源利用率。
- 支持按集群独立部署与混合部署对资源进行分类，支持在不同主机资源上灵活快速部署多套 TiDB 集群。
- 支持主机位置信息，区域（Region) - 可用区（Available Zone) - 机架（Rack）三层结构清楚定义主机位置。

集群全生命周期管理

- 支持一站式管理多套 TiDB 集群。

  一个 Web 入口统一管理多套 TiDB 集群，同时查看多套集群日志、监控和告警。

- 支持快速灵活部署 OLTP 集群和 HTAP 集群：

  - 一页配置集群，避免直接修改拓扑和配置文件，有效降低集群配置错误
  - 一键创建集群，缩短 88% 集群创建时间
  - 规范创建流程，降低技术门槛，自动检查创建流程每一步结果，确保数据库集群健康上线，及时响应业务需求

- 支持扩容、缩容 TiDB 集群：

  - 支持用户根据业务需要灵活调整 TiDB 集群规模，让数据库服务能力与业务规模保持动态适配
  - 集群扩缩容过程中，确保 TiDB 组件实例数量符合最佳实践
  - 支持异步完成 TiDB 集群缩容，确保 Region 数据迁移与集群缩容有序完成

- 支持一键重启、停止、删除 TiDB 集群：

  - 支持一键完成 TiDB 集群日常操作
  - 删除集群前，提醒用户进行最终备份，确保数据安全
  - 优化删除集群操作的用户交互，防呆设计有效避免集群误删除

- 统一集成数据库监控告警和 TiDB Dashboard。

  统一 Web 入口实时监控多套 TiDB 数据库性能，及时发现热点和 SQL 慢查询，帮助数据库管理员跟踪数据库核心指标，提前发现数据库潜在风险。

- 支持统一查看 TiDB 集群日志。

  统一查看多套 TiDB 集群日志，支持按组件、日志级别、时间范围、日志内容、IP 地址过滤日志，帮助数据库管理员快速完成根因定位。

- 支持接管由 TiUP 安装的 TiDB 集群。一键接管存量 TiDB 集群，统一入口管理新老集群，提高集群管理效率。

克隆集群与一键主备切换

- 支持一键克隆 TiDB 集群，快速构建主备集群

  支持一键克隆 TiDB 集群，快速构建主备集群，自动构建数据同步任务，确保主备集群间数据保持同步，有效缩短 90% 主备集群创建时间。

- 支持一键切换主备集群：

  - 帮助管理员快速完成主备集群切换，自动实现数据同步任务切换，提高数据库容灾能力及业务连续性
  - 主备切换过程中确保至少有一个集群始终可读，确保数据库服务不中断
  - 主备切换过程中确保数据不丢失

TiDB 集群升级

- 支持 TiDB 集群可视化原地升级
  - 支持全流程可视化集群原地升级，简化数据库管理员的业务操作流程，提高集群升级成功率；
  - 提供集群升级前后参数对比，允许数据库管理员核查、选择参数，避免升级前后参数变化对集群性能的影响。

参数管理

- 提供 TiDB 版本对应参数组模板：
  - 根据 TiDB 最佳实践经验，内置 TiDB 版本对应的参数组模板，帮助数据库管理员快速完成集群参数优化
  - 支持用户可自定义参数组，沉淀积累运营中的最佳参数组合
  - 支持用户快速应用参数组至不同集群
- 提供统一界面查看、修改 TiDB 集群参数：
  - 统一管理不同接口的参数（SQL、TiUP、各组件 API），屏蔽参数管理接口细节
  - 帮助数据库管理员随时查看、调整 TiDB 集群运行参数，降低参数管理维护难度
  - 通过参数值有效校验和定期巡检，减少参数错误对集群性能带来的负面影响

数据备份与恢复

- 支持手动备份 TiDB 集群，帮助数据库管理员随时触发备份请求，支持查看备份进度。
- 支持设置 TiDB 集群的自动备份策略，帮助数据库管理员设置周期性、自动化的备份策略，满足数据安全、审计的合规要求。
- 支持从备份数据恢复创建新 TiDB 集群，备份数据与创建集群与无缝对接，快速恢复集群与数据。
- 支持备份历史记录，方便数据库管理员查看历史备份记录，检查备份任务状态。

数据导入与导出

- 支持数据导入 TiDB 集群，从不同存储（S3 兼容存储、NFS 存储、本地存储）快速导入全量数据到 TiDB 集群。
- 支持从 TiDB 集群导出数据：
  - 从 TiDB 集群导出 SQL 或 CSV 格式的数据，可存储与 S3 兼容的存储或 NFS 存储
  - 支持按库、表过滤导出数据
  - 支持按 SQL 语句过滤导出数据

- 支持跨 TiDB 集群导入和导出数据。

  不同 TiDB 集群之间，通过 TiDB Enterprise Manager 共享存储实现跨集群的数据全量导入与导出，完成数据流动与融合，打破数据孤岛，实现数据价值挖掘。

- 支持导入导出历史记录，方便数据库管理员查看数据导入导出历史记录，检查任务状态。

数据同步

支持 TiCDC 数据同步：

- 快速构建从 TiDB 集群到下游系统 (TiDB、 MySQL、 Kafka ）的数据同步任务
- 支持数据同步任务按库、表过滤
- 支持数据同步任务管理：创建、暂停、恢复、 删除

工作流任务管理

- 自动记录管理员操作，历史任务事后可追查、可回溯，帮助数据库管理员查看集群历史任务记录，检查工作流执行进度与结果，快速分析操作失败原因。
- 规范不同场景下数据库管理员操作流程。规范数据库管理员在各场景下的操作流程，避免人工操作错误与遗漏。
- 支持异步工作流任务。管理员任务工作量支持异步，确保用户操作体验。

OpenAPI（实验特性）

提供集群全生命周期 OpenAPI：

- 帮助管理员完成数据库管理自动化
- 实现 TiDB 管理信息与第三方平台集成

14 参考指南

14.1 架构

14.1.1 TiDB 整体架构

推荐先观看以下视频（时长约 14 分钟），快速了解 TiDB 的整体架构。

与传统的单机数据库相比，TiDB 具有以下优势：

- 纯分布式架构，拥有良好的扩展性，支持弹性的扩缩容
- 支持 SQL，对外暴露 MySQL 的网络协议，并兼容大多数 MySQL 的语法，在大多数场景下可以直接替换 MySQL
- 默认支持高可用，在少数副本失效的情况下，数据库本身能够自动进行数据修复和故障转移，对业务透明
- 支持 ACID 事务，对于一些有强一致需求的场景友好，例如：银行转账

- 具有丰富的工具链生态，覆盖数据迁移、同步、备份等多种场景

在内核设计上，TiDB 分布式数据库将整体架构拆分成了多个模块，各模块之间互相通信，组成完整的 TiDB 系统。对应的架构图如下：

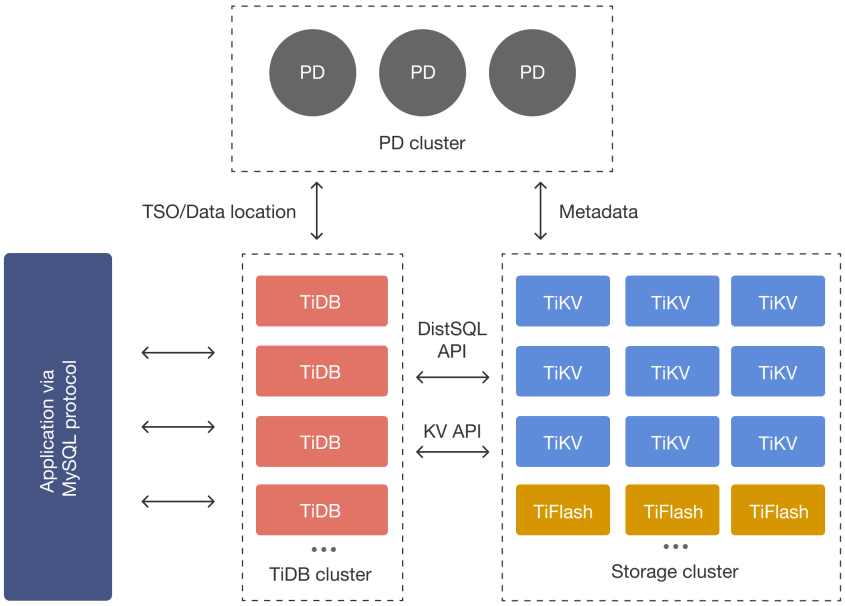


图 303: architecture

- TiDB Server：SQL 层，对外暴露 MySQL 协议的连接 endpoint，负责接受客户端的连接，执行 SQL 解析和优化，最终生成分布式执行计划。TiDB 层本身是无状态的，实践中可以启动多个 TiDB 实例，通过负载均衡组件（如 LVS、HAProxy 或 F5）对外提供统一的接入地址，客户端的连接可以均匀地分摊在多个 TiDB 实例上以达到负载均衡的效果。TiDB Server 本身并不存储数据，只是解析 SQL，将实际的数据读取请求转发给底层的存储节点 TiKV（或 TiFlash）。
- PD (Placement Driver) Server：整个 TiDB 集群的元信息管理模块，负责存储每个 TiKV 节点实时的数据分布情况和集群的整体拓扑结构，提供 TiDB Dashboard 管控界面，并为分布式事务分配事务 ID。PD 不仅存储元信息，同时还会根据 TiKV 节点实时上报的数据分布状态，下发数据调度命令给具体的 TiKV 节点，可以说是整个集群的“大脑”。此外，PD 本身也是由至少 3 个节点构成，拥有高可用的能力。建议部署奇数个 PD 节点。
- 存储节点
  - TiKV Server：负责存储数据，从外部看 TiKV 是一个分布式的提供事务的 Key-Value 存储引擎。存储数据的基本单位是 Region，每个 Region 负责存储一个 Key Range（从 StartKey 到 EndKey 的左闭右开区间）的数据，每个 TiKV 节点会负责多个 Region。TiKV 的 API 在 KV 键值对层面提供对分布式事务的原生支持，默认提供了 SI (Snapshot Isolation) 的隔离级别，这也是 TiDB 在 SQL 层面支持分布式事务的核心。TiDB 的 SQL 层做完 SQL 解析后，会将 SQL 的执行计划转换为对 TiKV API 的实际调用。所以，数据都存储在 TiKV 中。另外，TiKV 中的数据都会自动维护多副本（默认为三副本），天然支持高可用和自动故障转移。

- TiFlash：TiFlash 是一类特殊的存储节点。和普通 TiKV 节点不一样的是，在 TiFlash 内部，数据是以列式的形式进行存储，主要的功能是为分析型的场景加速。

14.1.2 TiDB 数据库的存储

本文主要介绍 TiKV 的一些设计思想和关键概念。

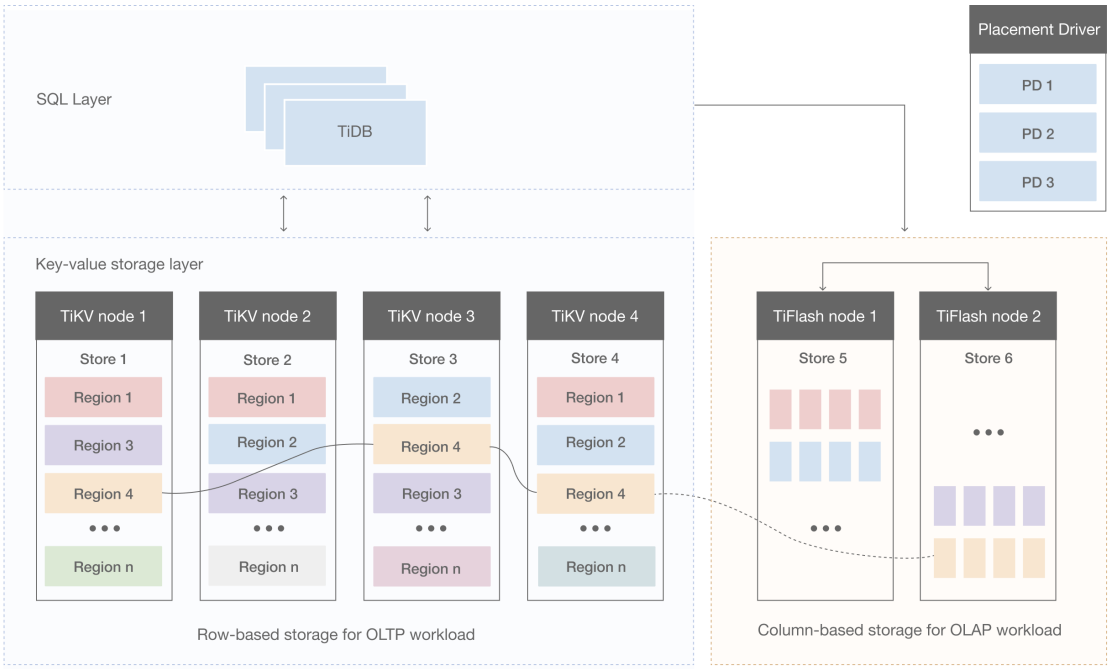


图 304: storage-architecture

14.1.2.1 Key-Value Pairs（键值对）

作为保存数据的系统，首先要决定的是数据的存储模型，也就是数据以什么样的形式保存下来。TiKV 的选择是 Key-Value 模型，并且提供有序遍历方法。

TiKV 数据存储的两个关键点：

1. 这是一个巨大的 Map（可以类比一下 C++ 的 std::map），也就是存储的是 Key-Value Pairs（键值对）
2. 这个 Map 中的 Key-Value pair 按照 Key 的二进制顺序有序，也就是可以 Seek 到某一个 Key 的位置，然后不断地调用 Next 方法以递增的顺序获取比这个 Key 大的 Key-Value。

注意，本文所说的 TiKV 的 KV 存储模型和 SQL 中的 Table 无关。本文不讨论 SQL 中的任何概念，专注于讨论如何实现 TiKV 这样一个高性能、高可靠性、分布式的 Key-Value 存储。

14.1.2.2 本地存储 (RocksDB)

任何持久化的存储引擎，数据终归要保存在磁盘上，TiKV 也不例外。但是 TiKV 没有选择直接向磁盘上写数据，而是把数据保存在 RocksDB 中，具体的数据落地由 RocksDB 负责。这个选择的原因是开发一个单机存储引擎工作量很大，特别是要做一个高性能的单机引擎，需要做各种细致的优化，而 RocksDB 是由 Facebook 开源的一个非常优秀的单机 KV 存储引擎，可以满足 TiKV 对单机引擎的各种要求。这里可以简单的认为 RocksDB 是一个单机的持久化 Key-Value Map。

14.1.2.3 Raft 协议

接下来 TiKV 的实现面临一件更难的事情：如何保证单机失效的情况下，数据不丢失，不出错？

简单来说，需要想办法把数据复制到多台机器上，这样一台机器无法服务了，其他的机器上的副本还能提供服务；复杂来说，还需要这个数据复制方案是可靠和高效的，并且能处理副本失效的情况。TiKV 选择了 Raft 算法。Raft 是一个一致性协议，本文只会对 Raft 做一个简要的介绍，细节问题可以参考它的论文。Raft 提供几个重要的功能：

- Leader（主副本）选举
- 成员变更（如添加副本、删除副本、转移 Leader 等操作）
- 日志复制

TiKV 利用 Raft 来做数据复制，每个数据变更都会落地为一条 Raft 日志，通过 Raft 的日志复制功能，将数据安全可靠地同步到复制组的每一个节点中。不过在实际写入中，根据 Raft 的协议，只需要同步复制到多数节点，即可安全地认为数据写入成功。

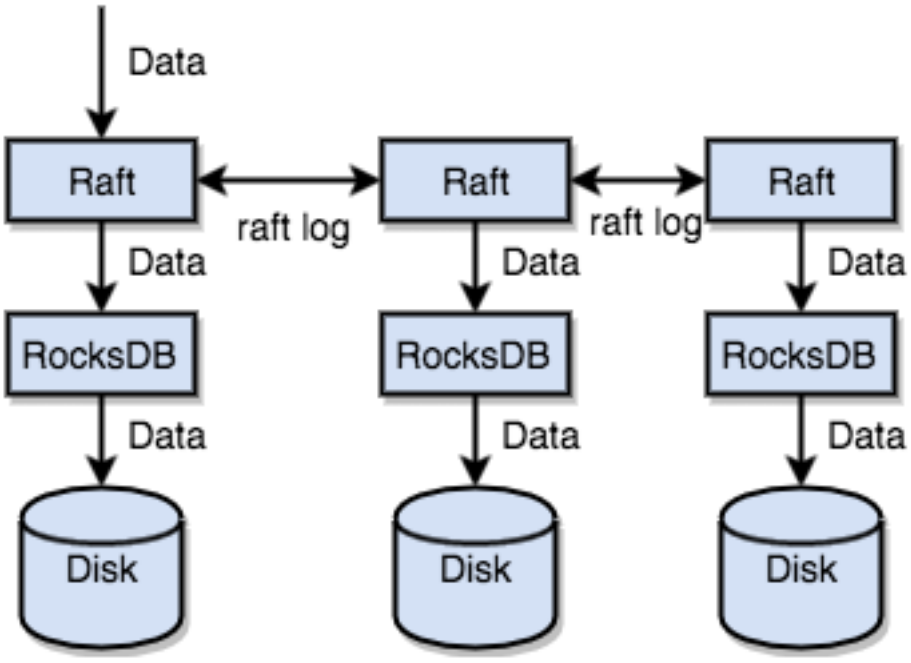


图 305: Raft in TiDB

总结一下，通过单机的 RocksDB，TiKV 可以将数据快速地存储在磁盘上；通过 Raft，将数据复制到多台机器上，以防单机失效。数据的写入是通过 Raft 这一层的接口写入，而不是直接写 RocksDB。通过实现 Raft，TiKV 变成

了一个分布式的 Key-Value 存储，少数几台机器宕机也能通过原生的 Raft 协议自动把副本补全，可以做到对业务无感知。

14.1.2.4 Region

首先，为了便于理解，在此节，假设所有的数据都只有一个副本。前面提到，TiKV 可以看做是一个巨大的有序的 KV Map，那么为了实现存储的水平扩展，数据将被分散在多台机器上。对于一个 KV 系统，将数据分散在多台机器上有两种比较典型的方案：

- Hash：按照 Key 做 Hash，根据 Hash 值选择对应的存储节点。
- Range：按照 Key 分 Range，某一段连续的 Key 都保存在一个存储节点上。

TiKV 选择了第二种方式，将整个 Key-Value 空间分成很多段，每一段是一系列连续的 Key，将每一段叫做一个 Region，可以用 [StartKey，EndKey) 这样一个左闭右开区间来描述。每个 Region 中保存的数据量默认维持在 96 MiB 左右（可以通过配置修改）。

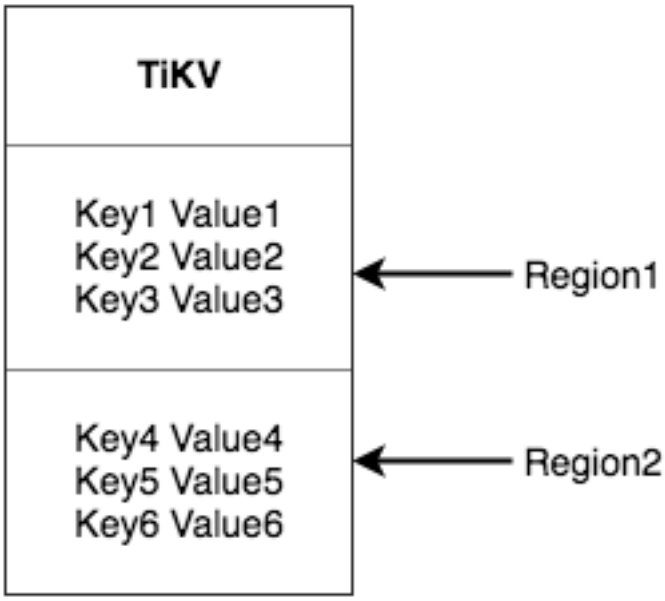


图 306: Region in TiDB

注意，这里的 Region 还是和 SQL 中的表没什么关系。这里的讨论依然不涉及 SQL，只和 KV 有关。

将数据划分成 Region 后，TiKV 将会做两件重要的事情：

- 以 Region 为单位，将数据分散在集群中所有的节点上，并且尽量保证每个节点上服务的 Region 数量差不多。
- 以 Region 为单位做 Raft 的复制和成员管理。

这两点非常重要：

- 先看第一点，数据按照 Key 切分成很多 Region，每个 Region 的数据只会保存在一个节点上面（暂不考虑多副本）。TiDB 系统会有一个组件 (PD) 来负责将 Region 尽可能均匀的散布在集群中所有的节点上，这样一方面实现了存储容量的水平扩展（增加新的节点后，会自动将其他节点上的 Region 调度过来），另一方面也实现了负载均衡（不会出现某个节点有很多数据，其他节点上没什么数据的情况）。同时为了保证上层客户端能够访问所需要的数据，系统中也会有一个组件 (PD) 记录 Region 在节点上面的分布情况，也就是通过任意一个 Key 就能查询到这个 Key 在哪个 Region 中，以及这个 Region 目前在哪个节点上（即 Key 的位置路由信息）。至于负责这两项重要工作的组件 (PD)，会在后续介绍。
- 对于第二点，TiKV 是以 Region 为单位做数据的复制，也就是一个 Region 的数据会保存多个副本，TiKV 将每一个副本叫做一个 Replica。Replica 之间是通过 Raft 来保持数据的一致，一个 Region 的多个 Replica 会保存在不同的节点上，构成一个 Raft Group。其中一个 Replica 会作为这个 Group 的 Leader，其他的 Replica 作为 Follower。默认情况下，所有的读和写都是通过 Leader 进行，读操作在 Leader 上即可完成，而写操作再由 Leader 复制给 Follower。

大家理解了 Region 之后，应该可以理解下面这张图：

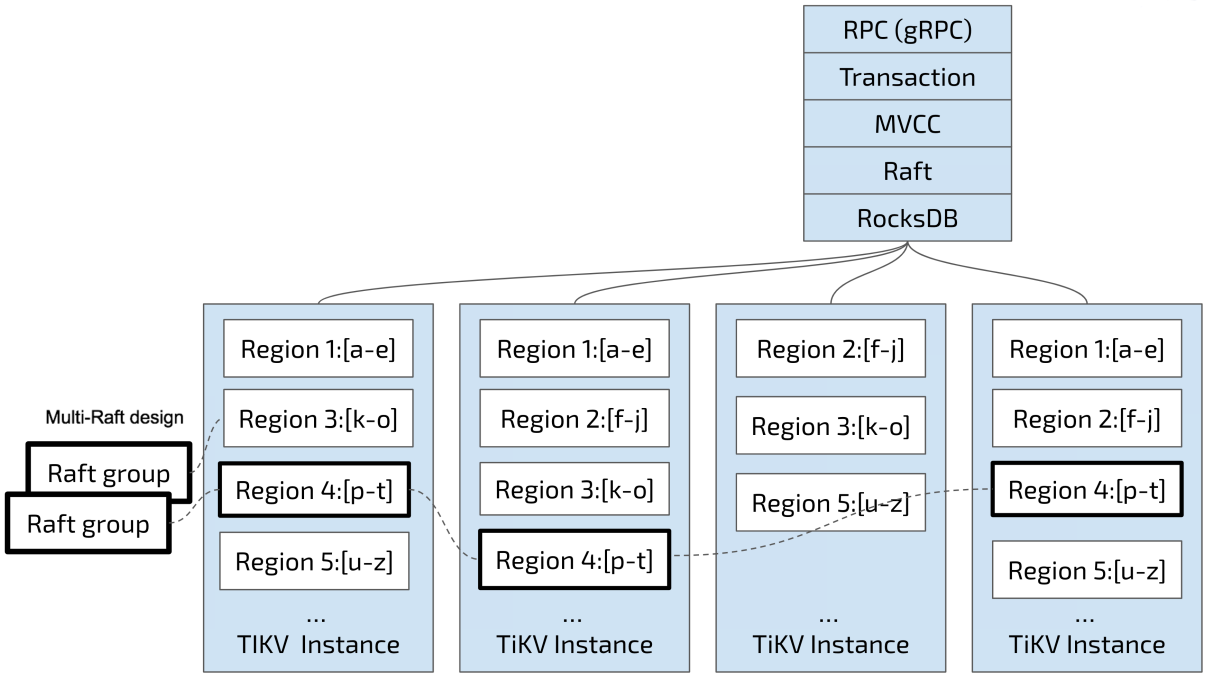


图 307: TiDB Storage

以 Region 为单位做数据的分散和复制，TiKV 就成为了一个分布式的具备一定容灾能力的 KeyValue 系统，不用再担心数据存不下，或者是磁盘故障丢失数据的问题。

14.1.2.5 MVCC

很多数据库都会实现多版本并发控制 (MVCC)，TiKV 也不例外。设想这样的场景：两个客户端同时去修改一个 Key 的 Value，如果没有数据的多版本控制，就需要对数据上锁，在分布式场景下，可能会带来性能以及死锁问题。TiKV 的 MVCC 实现是通过在 Key 后面添加版本号来实现，简单来说，没有 MVCC 之前，可以把 TiKV 看做这样的：

```
Key1 -> Value
Key2 -> Value
……
KeyN -> Value
```

有了 MVCC 之后，TiKV 的 Key 排列是这样的：

```
Key1_Version3 -> Value
Key1_Version2 -> Value
Key1_Version1 -> Value
……
Key2_Version4 -> Value
Key2_Version3 -> Value
Key2_Version2 -> Value
Key2_Version1 -> Value
……
KeyN_Version2 -> Value
KeyN_Version1 -> Value
……
```

注意，对于同一个 Key 的多个版本，版本号较大的会被放在前面，版本号小的会被放在后面（见Key-Value 一节，Key 是有序的排列），这样当用户通过一个 Key + Version 来获取 Value 的时候，可以通过 Key 和 Version 构造出 MVCC 的 Key，也就是 Key_Version。然后可以直接通过 RocksDB 的 SeekPrefix(Key_Version) API，定位到第一个大于等于这个 Key_Version 的位置。

14.1.2.6 分布式 ACID 事务

TiKV 的事务采用的是 Google 在 BigTable 中使用的事务模型：Percolator，TiKV 根据这篇论文实现，并做了大量的优化。详细介绍参见事务概览。

14.1.3 TiDB 数据库的计算

TiDB 在 TiKV 提供的分布式存储能力基础上，构建了兼具优异的交易处理能力与良好的数据分析能力的计算引擎。本文首先从数据映射算法入手介绍 TiDB 如何将库表中的数据映射到 TiKV 中的 (Key, Value) 键值对，然后描述 TiDB 元信息管理方式，最后介绍 TiDB SQL 层的主要架构。

对于计算层依赖的存储方案，本文只介绍基于 TiKV 的行存储结构。针对分析型业务的特点，TiDB 推出了作为 TiKV 扩展的列存储方案TiFlash。

14.1.3.1 表数据与 Key-Value 的映射关系

本小节介绍 TiDB 中数据到 (Key, Value) 键值对的映射方案。这里的数据主要包括以下两个方面：

- 表中每一行的数据，以下简称表数据
- 表中所有索引的数据，以下简称索引数据

14.1.3.1.1 表数据与 Key-Value 的映射关系

在关系型数据库中，一个表可能有很多列。要将一行中各列数据映射成一个 (Key, Value) 键值对，需要考虑如何构造 Key。首先，OLTP 场景下有大量针对单行或者多行的增、删、改、查等操作，要求数据库具备快速读取一行数据的能力。因此，对应的 Key 最好有一个唯一 ID（显示或隐式的 ID），以方便快速定位。其次，很多 OLAP 型查询需要进行全表扫描。如果能够将一个表中所有行的 Key 编码到一个区间内，就可以通过范围查询高效完成全表扫描的任务。

基于上述考虑，TiDB 中的表数据与 Key-Value 的映射关系作了如下设计：

- 为了保证同一个表的数据放在一起，方便查找，TiDB 会为每个表分配一个表 ID，用 `TableID` 表示。表 ID 是一个整数，在整个集群内唯一。
- TiDB 会为表中每行数据分配一个行 ID，用 `RowID` 表示。行 ID 也是一个整数，在表内唯一。对于行 ID，TiDB 做了一个小优化，如果某个表有整数型的主键，TiDB 会使用主键的值当做这一行数据的行 ID。

每行数据按照如下规则编码成 (Key, Value) 键值对：

```
Key:   tablePrefix{TableID}_recordPrefixSep{RowID}
Value: [col1, col2, col3, col4]
```

其中 `tablePrefix` 和 `recordPrefixSep` 都是特定的字符串常量，用于在 Key 空间内区分其他数据。其具体值在后面的小结中给出。

14.1.3.1.2 索引数据和 Key-Value 的映射关系

TiDB 同时支持主键和二级索引（包括唯一索引和非唯一索引）。与表数据映射方案类似，TiDB 为表中每个索引分配了一个索引 ID，用 `IndexID` 表示。

对于主键和唯一索引，需要根据键值快速定位到对应的 RowID，因此，按照如下规则编码成 (Key, Value) 键值对：

```
Key:   tablePrefix{tableID}_indexPrefixSep{indexID}_indexedColumnsValue
Value: RowID
```

对于不需要满足唯一性约束的普通二级索引，一个键值可能对应多行，需要根据键值范围查询对应的 RowID。因此，按照如下规则编码成 (Key, Value) 键值对：

```
Key:   tablePrefix{TableID}_indexPrefixSep{IndexID}_indexedColumnsValue_{RowID}
Value: null
```

14.1.3.1.3 映射关系小结

上述所有编码规则中的 `tablePrefix`、`recordPrefixSep` 和 `indexPrefixSep` 都是字符串常量，用于在 Key 空间内区分其他数据，定义如下：

```
tablePrefix     = []byte{'t'}
recordPrefixSep = []byte{'r'}
indexPrefixSep  = []byte{'i'}
```

另外请注意，上述方案中，无论是表数据还是索引数据的 Key 编码方案，一个表内所有的行都有相同的 Key 前缀，一个索引的所有数据也都有相同的前缀。这样具有相同的前缀的数据，在 TiKV 的 Key 空间内，是排列在一起的。因此只要小心地设计后缀部分的编码方案，保证编码前和编码后的比较关系不变，就可以将表数据或者索引数据有序地保存在 TiKV 中。采用这种编码后，一个表的所有行数据会按照 RowID 顺序地排列在 TiKV 的 Key 空间中，某一个索引的数据也会按照索引数据的具体的值（编码方案中的 indexedColumnsValue ）顺序地排列在 Key 空间内。

14.1.3.1.4 Key-Value 映射关系示例

最后通过一个简单的例子，来理解 TiDB 的 Key-Value 映射关系。假设 TiDB 中有如下这个表：

```
CREATE TABLE User (
    ID int,
    Name varchar(20),
    Role varchar(20),
    Age int,
    PRIMARY KEY (ID),
    KEY idxAge (Age)
);
```

假设该表中有 3 行数据：

```
1, "TiDB", "SQL Layer", 10
2, "TiKV", "KV Engine", 20
3, "PD", "Manager", 30
```

首先每行数据都会映射为一个 (Key, Value) 键值对，同时该表有一个 int 类型的主键，所以 RowID 的值即为该主键的值。假设该表的 TableID 为 10，则其存储在 TiKV 上的表数据为：

```
t10_r1 --> ["TiDB", "SQL Layer", 10]
t10_r2 --> ["TiKV", "KV Engine", 20]
t10_r3 --> ["PD", "Manager", 30]
```

除了主键外，该表还有一个非唯一的普通二级索引 idxAge，假设这个索引的 IndexID 为 1，则其存储在 TiKV 上的索引数据为：

```
t10_i1_10_1 --> null
t10_i1_20_2 --> null
t10_i1_30_3 --> null
```

以上例子展示了 TiDB 中关系模型到 Key-Value 模型的映射规则，以及选择该方案背后的考量。

14.1.3.2 元信息管理

TiDB 中每个 Database 和 Table 都有元信息，也就是其定义以及各项属性。这些信息也需要持久化，TiDB 将这些信息也存储在了 TiKV 中。

每个 Database/Table 都被分配了一个唯一的 ID，这个 ID 作为唯一标识，并且在编码为 Key-Value 时，这个 ID 都会编码到 Key 中，再加上 m_ 前缀。这样可以构造出一个 Key，Value 中存储的是序列化后的元信息。

除此之外，TiDB 还用一个专门的 (Key, Value) 键值对存储当前所有表结构信息的最新版本号。这个键值对是全局的，每次 DDL 操作的状态改变时其版本号都会加 1。目前，TiDB 把这个键值对持久化存储在 PD Server 中，其 Key 是 “/tidb/ddl/global_schema_version”，Value 是类型为 int64 的版本号值。TiDB 采用 Online Schema 变更算法，有一个后台线程在不断地检查 PD Server 中存储的表结构信息的版本号是否发生变化，并且保证在一定时间内一定能够获取版本的变化。

14.1.3.3 SQL 层简介

TiDB 的 SQL 层，即 TiDB Server，负责将 SQL 翻译成 Key-Value 操作，将其转发给共用的分布式 Key-Value 存储层 TiKV，然后组装 TiKV 返回的结果，最终将查询结果返回给客户端。

这一层的节点都是无状态的，节点本身并不存储数据，节点之间完全对等。

14.1.3.3.1 SQL 运算

最简单的方案就是通过上一节所述的表数据与 Key-Value 的映射关系方案，将 SQL 查询映射为对 KV 的查询，再通过 KV 接口获取对应的数据，最后执行各种计算。

比如 `select count(*)from user where name = "TiDB"` 这样一个 SQL 语句，它需要读取表中所有的数据，然后检查 `name` 字段是否是 `TiDB`，如果是的话，则返回这一行。具体流程如下：

1. 构造出 Key Range：一个表中所有的 RowID 都在 `[0, MaxInt64)` 这个范围内，使用 0 和 `MaxInt64` 根据行数据的 Key 编码规则，就能构造出一个 `[StartKey, EndKey)`的左闭右开区间。
2. 扫描 Key Range：根据上面构造出的 Key Range，读取 TiKV 中的数据。
3. 过滤数据：对于读到的每一行数据，计算 `name = "TiDB"` 这个表达式，如果为真，则向上返回这一行，否则丢弃这一行数据。
4. 计算 `Count(*)`：对符合要求的每一行，累计到 `Count(*)` 的结果上面。

整个流程示意图如下：

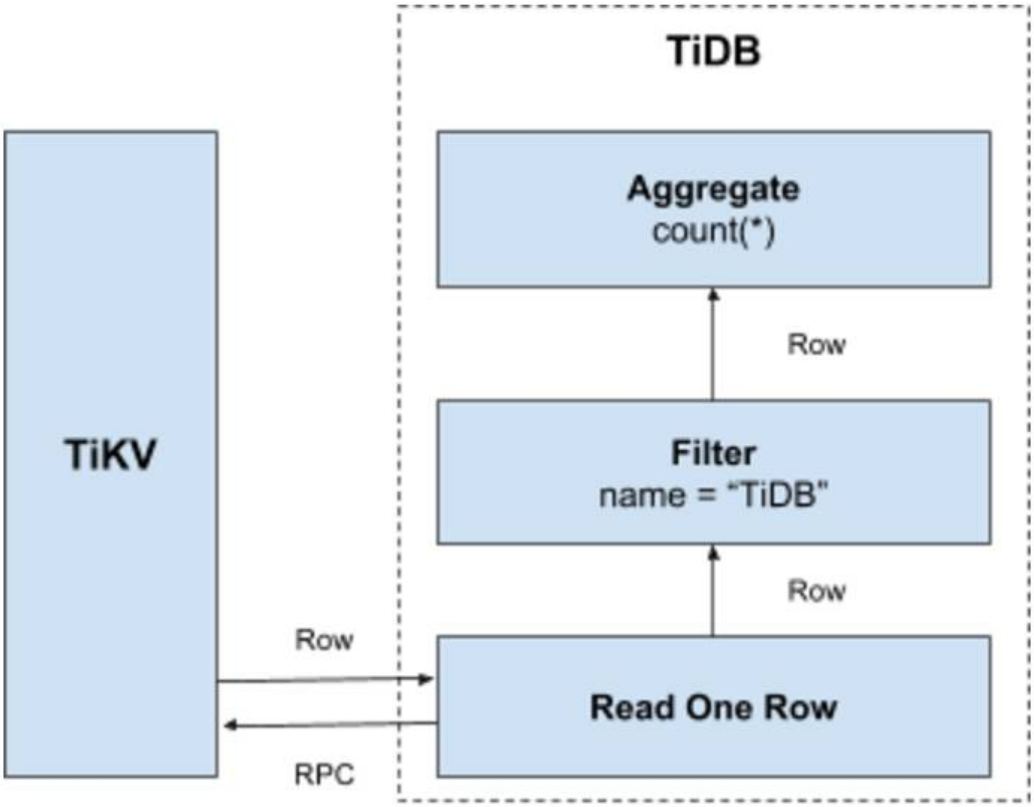


图 308: naive sql flow

这个方案是直观且可行的，但是在分布式数据库的场景下有一些显而易见的问题：

- 在扫描数据的时候，每一行都要通过 KV 操作从 TiKV 中读取出来，至少有一次 RPC 开销，如果需要扫描的数据很多，那么这个开销会非常大。
- 并不是所有的行都满足过滤条件 `name = "TiDB"`，如果不满足条件，其实可以不读取出来。
- 此查询只要求返回符合要求行的数量，不要求返回这些行的值。

14.1.3.3.2 分布式 SQL 运算

为了解决上述问题，计算应该需要尽量靠近存储节点，以避免大量的 RPC 调用。首先，SQL 中的谓词条件 `name = "TiDB"` 应被下推到存储节点进行计算，这样只需要返回有效的行，避免无意义的网络传输。然后，聚合函数 `Count(*)` 也可以被下推到存储节点，进行预聚合，每个节点只需要返回一个 `Count(*)` 的结果即可，再由 SQL 层将各个节点返回的 `Count(*)` 的结果累加求和。

以下是数据逐层返回的示意图：

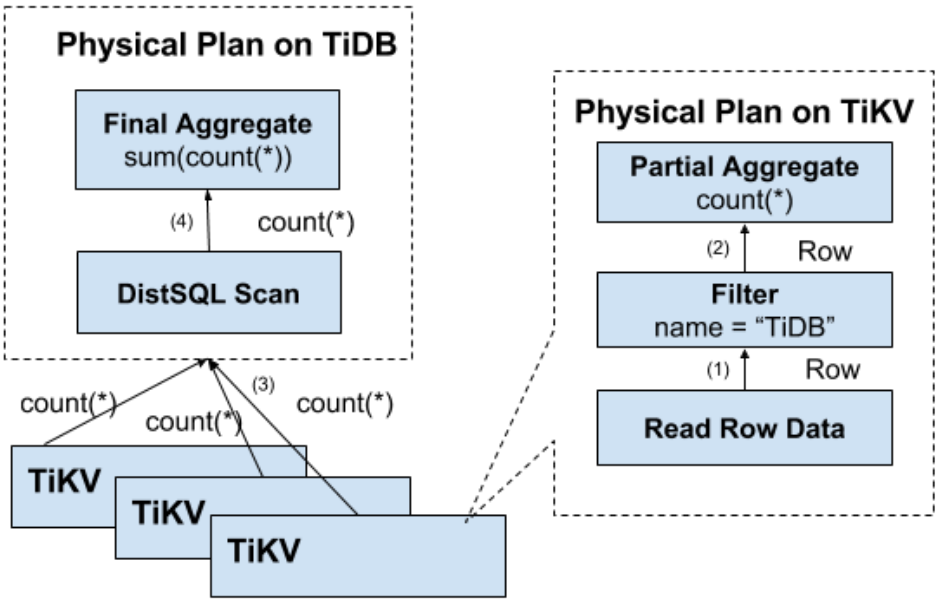


图 309: dist sql flow

14.1.3.3.3 SQL 层架构

通过上面的例子，希望大家对 SQL 语句的处理有一个基本的了解。实际上 TiDB 的 SQL 层要复杂得多，模块以及层次非常多，下图列出了重要的模块以及调用关系：

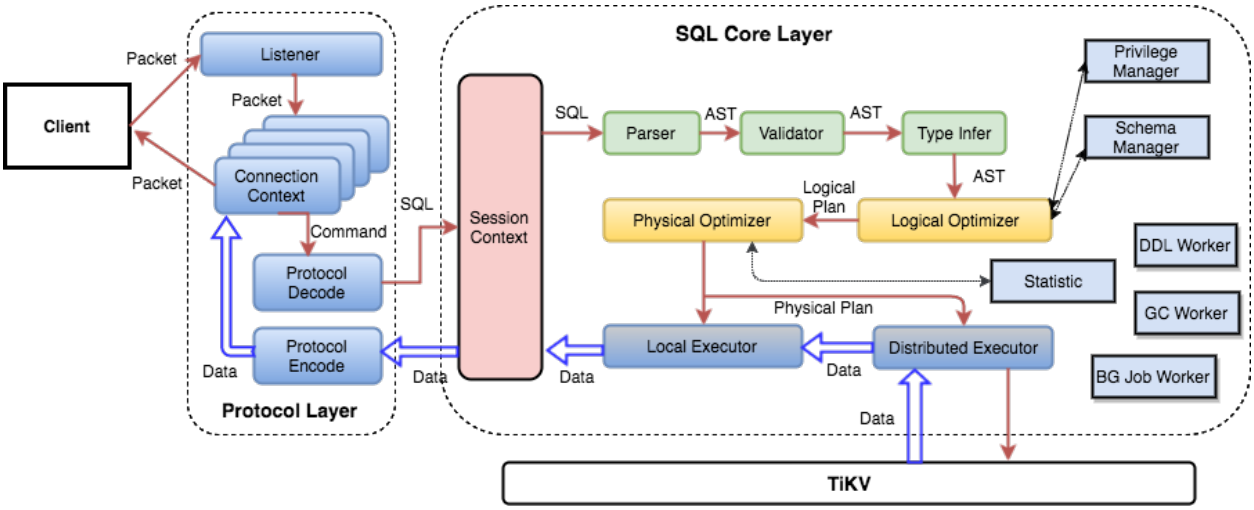


图 310: tidb sql layer

用户的 SQL 请求会直接或者通过 Load Balancer 发送到 TiDB Server，TiDB Server 会解析 MySQL Protocol Packet，获取请求内容，对 SQL 进行语法解析和语义分析，制定和优化查询计划，执行查询计划并获取和处理数据。数据全部存储在 TiKV 集群中，所以在这个过程中 TiDB Server 需要和 TiKV 交互，获取数据。最后 TiDB Server 需要将查询结果返回给用户。

14.1.4 TiDB 数据库的调度

PD (Placement Driver) 是 TiDB 集群的管理模块，同时也负责集群数据的实时调度。本文档介绍一下 PD 的设计思想和关键概念。

14.1.4.1 场景描述

TiKV 集群是 TiDB 数据库的分布式 KV 存储引擎，数据以 Region 为单位进行复制和管理，每个 Region 会有多个副本 (Replica)，这些副本会分布在不同的 TiKV 节点上，其中 Leader 负责读/写，Follower 负责同步 Leader 发来的 Raft log。

需要考虑以下场景：

- 为了提高集群的空间利用率，需要根据 Region 的空间占用对副本进行合理的分布。
- 集群进行跨机房部署的时候，要保证一个机房掉线，不会丢失 Raft Group 的多个副本。
- 添加一个节点进入 TiKV 集群之后，需要合理地将集群中其他节点上的数据搬到新增节点。
- 当一个节点掉线时，需要考虑快速稳定地进行容灾。
  - 从节点的恢复时间来看
    - 如果节点只是短暂掉线（重启服务），是否需要进行调度。
    - 如果节点是长时间掉线（磁盘故障，数据全部丢失），如何进行调度。
  - 假设集群需要每个 Raft Group 有 N 个副本，从单个 Raft Group 的副本个数来看
    - 副本数量不够（例如节点掉线，失去副本），需要选择适当的机器的进行补充。
    - 副本数量过多（例如掉线的节点又恢复正常，自动加入集群），需要合理的删除多余的副本。

- 读/写通过 Leader 进行，Leader 的分布只集中在少量几个节点会对集群造成影响。
- 并不是所有的 Region 都被频繁的访问，可能访问热点只在少数几个 Region，需要通过调度进行负载均衡。
- 集群在做负载均衡的时候，往往需要搬迁数据，这种数据的迁移可能会占用大量的网络带宽、磁盘 IO 以及 CPU，进而影响在线服务。

以上问题和场景如果多个同时出现，就不太容易解决，因为需要考虑全局信息。同时整个系统也是在动态变化的，因此需要一个中心节点，来对系统的整体状况进行把控和调整，所以有了 PD 这个模块。

14.1.4.2 调度的需求

对以上的问题和场景进行分类和整理，可归为以下两类：

第一类：作为一个分布式高可用存储系统，必须满足的需求，包括几种

- 副本数量不能多也不能少
- 副本需要根据拓扑结构分布在不同属性的机器上
- 节点宕机或异常能够自动合理快速地进行容灾

第二类：作为一个良好的分布式系统，需要考虑的地方包括

- 维持整个集群的 Leader 分布均匀
- 维持每个节点的储存容量均匀
- 维持访问热点分布均匀
- 控制负载均衡的速度，避免影响在线服务
- 管理节点状态，包括手动上线/下线节点

满足第一类需求后，整个系统将具备强大的容灾功能。满足第二类需求后，可以使得系统整体的资源利用率更高且合理，具备良好的扩展性。

为了满足这些需求，首先需要收集足够的信息，比如每个节点的状态、每个 Raft Group 的信息、业务访问操作的统计等；其次需要设置一些策略，PD 根据这些信息以及调度的策略，制定出尽量满足前面所述需求的调度计划；最后需要一些基本的操作，来完成调度计划。

14.1.4.3 调度的基本操作

调度的基本操作指的是为了满足调度的策略。上述调度需求可整理为以下三个操作：

- 增加一个副本
- 删除一个副本
- 将 Leader 角色在一个 Raft Group 的不同副本之间 transfer（迁移）

刚好 Raft 协议通过 `AddReplica`、`RemoveReplica`、`TransferLeader` 这三个命令，可以支撑上述三种基本操作。

14.1.4.4 信息收集

调度依赖于整个集群信息的收集，简单来说，调度需要知道每个 TiKV 节点的状态以及每个 Region 的状态。TiKV 集群会向 PD 汇报两类消息，TiKV 节点信息和 Region 信息：

每个 TiKV 节点会定期向 PD 汇报节点的状态信息

TiKV 节点 (Store) 与 PD 之间存在心跳包，一方面 PD 通过心跳包检测每个 Store 是否存活，以及是否有新加入的 Store；另一方面，心跳包中也会携带这个 Store 的状态信息，主要包括：

- 总磁盘容量
- 可用磁盘容量
- 承载的 Region 数量
- 数据写入/读取速度
- 发送/接受的 Snapshot 数量（副本之间可能会通过 Snapshot 同步数据）
- 是否过载
- labels 标签信息（标签是具备层级关系的一系列 Tag，能够感知拓扑信息）

通过使用 `pd-ctl` 可以查看到 TiKV Store 的状态信息。TiKV Store 的状态具体分为 Up，Disconnect，Offline，Down，Tombstone。各状态的关系如下：

- Up：表示当前的 TiKV Store 处于提供服务的状态。
- Disconnect：当 PD 和 TiKV Store 的心跳信息丢失超过 20 秒后，该 Store 的状态会变为 Disconnect 状态，当时间超过 `max-store-down-time` 指定的时间后，该 Store 会变为 Down 状态。
- Down：表示该 TiKV Store 与集群失去连接的时间已经超过了 `max-store-down-time` 指定的时间，默认 30 分钟。超过该时间后，对应的 Store 会变为 Down，并且开始在存活的 Store 上补足各个 Region 的副本。
- Offline：当对某个 TiKV Store 通过 PD Control 进行手动下线操作，该 Store 会变为 Offline 状态。该状态只是 Store 下线的中间状态，处于该状态的 Store 会将其上的所有 Region 搬离至其它满足搬迁条件的 Up 状态 Store。当该 Store 的 `leader_count` 和 `region_count` (在 PD Control 中获取) 均显示为 0 后，该 Store 会由 Offline 状态变为 Tombstone 状态。在 Offline 状态下，禁止关闭该 Store 服务以及其所在的物理服务器。下线过程中，如果集群里不存在满足搬迁条件的其它目标 Store（例如没有足够的 Store 能够继续满足集群的副本数量要求），该 Store 将一直处于 Offline 状态。
- Tombstone：表示该 TiKV Store 已处于完全下线状态，可以使用 `remove-tombstone` 接口安全地清理该状态的 TiKV。

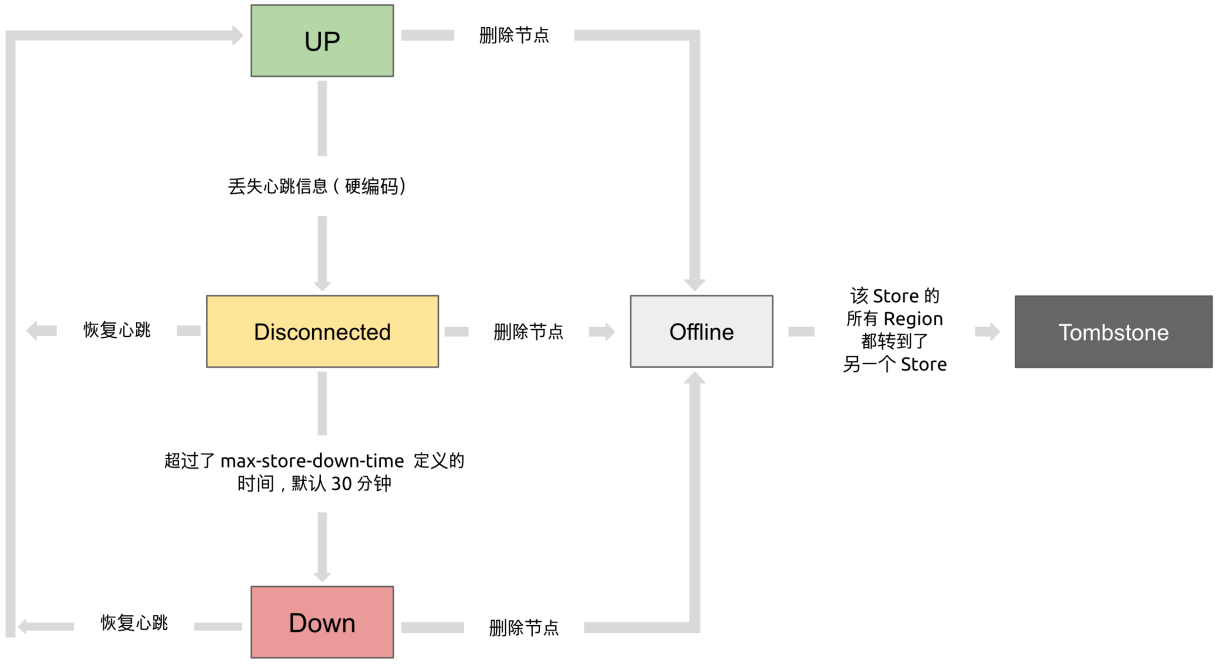


图 311: TiKV store status relationship

每个 Raft Group 的 Leader 会定期向 PD 汇报 Region 的状态信息

每个 Raft Group 的 Leader 和 PD 之间存在心跳包，用于汇报这个 Region 的状态，主要包括下面几点信息：

- Leader 的位置
- Followers 的位置
- 掉线副本的个数
- 数据写入/读取的速度

PD 不断的通过这两类心跳消息收集整个集群的信息，再以这些信息作为决策的依据。

除此之外，PD 还可以通过扩展的接口接受额外的信息，用来做更准确的决策。比如当某个 Store 的心跳包中断的时候，PD 并不能判断这个节点是临时失效还是永久失效，只能经过一段时间的等待（默认是 30 分钟），如果一直没有心跳包，就认为该 Store 已经下线，再决定需要将这个 Store 上面的 Region 都调度走。

但是有的时候，是运维人员主动将某台机器下线，这个时候，可以通过 PD 的管理接口通知 PD 该 Store 不可用，PD 就可以马上判断需要将这个 Store 上面的 Region 都调度走。

14.1.4.5 调度的策略

PD 收集了这些信息后，还需要一些策略来制定具体的调度计划。

一个 Region 的副本数量正确

当 PD 通过某个 Region Leader 的心跳包发现这个 Region 的副本数量不满足要求时，需要通过 Add/Remove Replica 操作调整副本数量。出现这种情况的可能原因是：

- 某个节点掉线，上面的数据全部丢失，导致一些 Region 的副本数量不足
- 某个掉线节点又恢复服务，自动接入集群，这样之前已经补足了副本的 Region 的副本数量过多，需要删除某个副本
- 管理员调整副本策略，修改了 max-replicas 的配置

一个 Raft Group 中的多个副本不在同一个位置

注意这里用的是『同一个位置』而不是『同一个节点』。在一般情况下，PD 只会保证多个副本不落在一个节点上，以避免单个节点失效导致多个副本丢失。在实际部署中，还可能出现下面这些需求：

- 多个节点部署在同一台物理机器上
- TiKV 节点分布在多个机架上，希望单个机架掉电时，也能保证系统可用性
- TiKV 节点分布在多个 IDC 中，希望单个机房掉电时，也能保证系统可用性

这些需求本质上都是某一个节点具备共同的位置属性，构成一个最小的『容错单元』，希望这个单元内部不会存在一个 Region 的多个副本。这个时候，可以给节点配置 labels 并且通过在 PD 上配置 location-labels 来指名哪些 label 是位置标识，需要在副本分配的时候尽量保证一个 Region 的多个副本不会分布在具有相同的位置标识的节点上。

副本在 Store 之间的分布均匀分配

由于每个 Region 的副本中存储的数据容量上限是固定的，通过维持每个节点上面副本数量的均衡，使得各节点间承载的数据更均衡。

Leader 数量在 Store 之间均匀分配

Raft 协议要求读取和写入都通过 Leader 进行，所以计算的负载主要在 Leader 上面，PD 会尽可能将 Leader 在节点间分散开。

访问热点数量在 Store 之间均匀分配

每个 Store 以及 Region Leader 在上报信息时携带了当前访问负载的信息，比如 Key 的读取/写入速度。PD 会检测出访问热点，且将其在节点之间分散开。

各个 Store 的存储空间占用大致相等

每个 Store 启动的时候都会指定一个 `Capacity` 参数，表明这个 Store 的存储空间上限，PD 在做调度的时候，会考虑节点的存储空间剩余量。

控制调度速度，避免影响在线服务

调度操作需要耗费 CPU、内存、磁盘 IO 以及网络带宽，需要避免对线上服务造成太大影响。PD 会对当前正在进行的操作数量进行控制，默认的速度控制是比较保守的，如果希望加快调度（比如停服务升级或者增加新节点，希望尽快调度），那么可以通过调节 PD 参数动态加快调度速度。

14.1.4.6 调度的实现

本节介绍调度的实现

PD 不断地通过 Store 或者 Leader 的心跳包收集整个集群信息，并且根据这些信息以及调度策略生成调度操作序列。每次收到 Region Leader 发来的心跳包时，PD 都会检查这个 Region 是否有待进行的操作，然后通过心跳包的回复消息，将需要进行的操作返回给 Region Leader，并在后面的心跳包中监测执行结果。

注意这里的操作只是给 Region Leader 的建议，并不保证一定能得到执行，具体是否会执行以及什么时候执行，由 Region Leader 根据当前自身状态来定。

14.2 存储引擎 TiKV

14.2.1 TiKV 简介

TiKV 是一个分布式事务型的键值数据库，提供了满足 ACID 约束的分布式事务接口，并且通过 Raft 协议保证了多副本数据一致性以及高可用。TiKV 作为 TiDB 的存储层，为用户写入 TiDB 的数据提供了持久化以及读写服务，同时还存储了 TiDB 的统计信息数据。

14.2.1.1 整体架构

与传统的整节点备份方式不同，TiKV 参考 Spanner 设计了 multi-raft-group 的副本机制。将数据按照 key 的范围划分成大致相等的切片（下文统称为 Region），每一个切片会有多个副本（通常是 3 个），其中一个副本是 Leader，提供读写服务。TiKV 通过 PD 对这些 Region 以及副本进行调度，以保证数据和读写负载都均匀地分散在各个 TiKV 上，这样的设计保证了整个集群资源的充分利用并且可以随着机器数量的增加水平扩展。

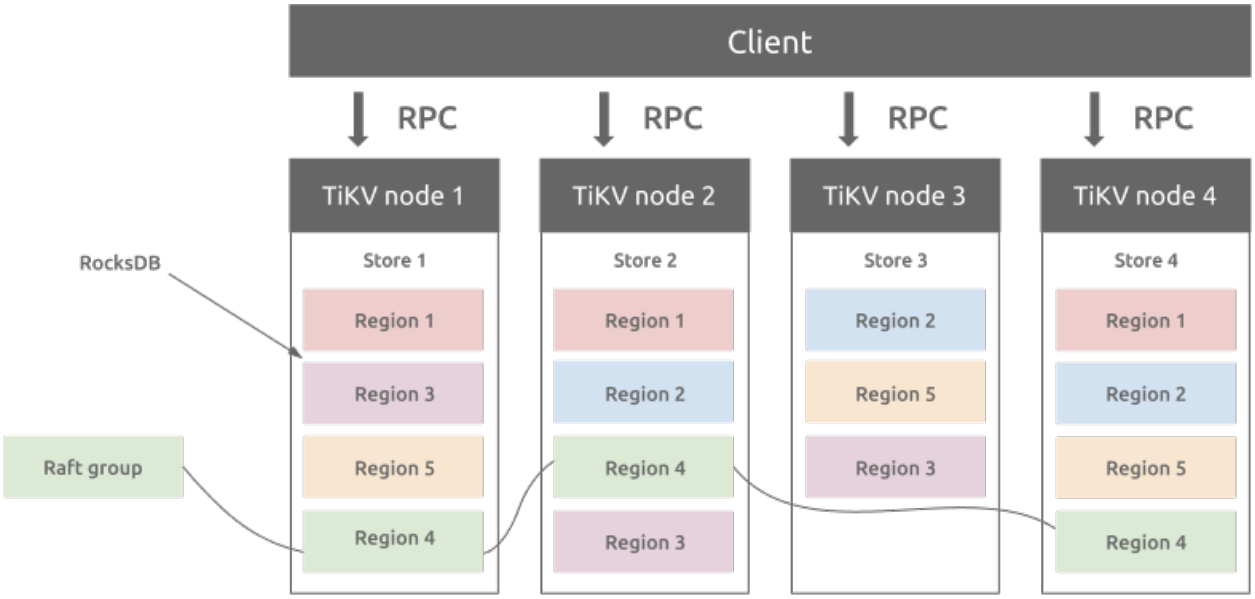


图 312: TiKV 架构

14.2.1.1.1 Region 与 RocksDB

虽然 TiKV 将数据按照范围切割成了多个 Region，但是同一个节点的所有 Region 数据仍然是不加区分地存储于同一个 RocksDB 实例上，而用于 Raft 协议复制所需要的日志则存储于另一个 RocksDB 实例。这样设计的原因是因为随机 I/O 的性能远低于顺序 I/O，所以 TiKV 使用同一个 RocksDB 实例来存储这些数据，以便不同 Region 的写入可以合并在一次 I/O 中。

14.2.1.1.2 Region 与 Raft 协议

Region 与副本之间通过 Raft 协议来维持数据一致性，任何写请求都只能在 Leader 上写入，并且需要写入多数副本后（默认配置为 3 副本，即所有请求必须至少写入两个副本成功）才会返回客户端写入成功。

TiKV 会尽量保持每个 Region 中保存的数据在一个合适的大小，目前默认是 96 MB，这样更有利于 PD 进行调度决策。当某个 Region 的大小超过一定限制（默认是 144 MiB）后，TiKV 会将它分裂为两个或者更多个 Region。同样，当某个 Region 因为大量的删除请求而变得太小时（默认是 20 MiB），TiKV 会将比较小的两个相邻 Region 合并为一个。

当 PD 需要把某个 Region 的一个副本从一个 TiKV 节点调度到另一个上面时，PD 会先为这个 Raft Group 在目标节点上增加一个 Learner 副本（虽然会复制 Leader 的数据，但是不会计入写请求的多数副本中）。当这个 Learner 副本的进度大致追上 Leader 副本时，Leader 会将它变更为 Follower，之后再移除操作节点的 Follower 副本，这样就完成了 Region 副本的一次调度。

Leader 副本的调度原理也类似，不过需要在目标节点的 Learner 副本变为 Follower 副本后，再执行一次 Leader Transfer，让该 Follower 主动发起一次选举成为新 Leader，之后新 Leader 负责删除旧 Leader 这个副本。

14.2.1.2 分布式事务

TiKV 支持分布式事务，用户（或者 TiDB）可以一次性写入多个 key-value 而不必关心这些 key-value 是否处于同一个数据切片 (Region) 上，TiKV 通过两阶段提交保证了这些读写请求的 ACID 约束，详见TiDB 乐观事务模型。

14.2.1.3 计算加速

TiKV 通过协处理器 (Coprocessor) 可以为 TiDB 分担一部分计算：TiDB 会将可以由存储层分担的计算下推。能否下推取决于 TiKV 是否可以支持相关下推。计算单元仍然是以 Region 为单位，即 TiKV 的一个 Coprocessor 计算请求中不会计算超过一个 Region 的数据。

14.2.2 RocksDB 简介

RocksDB 是由 Facebook 基于 LevelDB 开发的一款提供键值存储与读写功能的 LSM-tree 架构引擎。用户写入的键值对会先写入磁盘上的 WAL (Write Ahead Log)，然后再写入内存中的跳表（SkipList，这部分结构又被称作 MemTable）。LSM-tree 引擎由于将用户的随机修改（插入）转化为了对 WAL 文件的顺序写，因此具有比 B 树类存储引擎更高的写吞吐。

内存中的数据达到一定阈值后，会刷到磁盘上生成 SST 文件 (Sorted String Table)，SST 又分为多层（默认至多 6 层），每一层的数据达到一定阈值后会挑选一部分 SST 合并到下一层，每一层的数据是上一层的 10 倍（因此 90% 的数据存储在最后一层）。

RocksDB 允许用户创建多个 ColumnFamily，这些 ColumnFamily 各自拥有独立的内存跳表以及 SST 文件，但是共享同一个 WAL 文件，这样的好处是可以根据应用特点为不同的 ColumnFamily 选择不同的配置，但是又没有增加对 WAL 的写次数。

14.2.2.1 TiKV 架构

TiKV 的系统架构如下图所示：

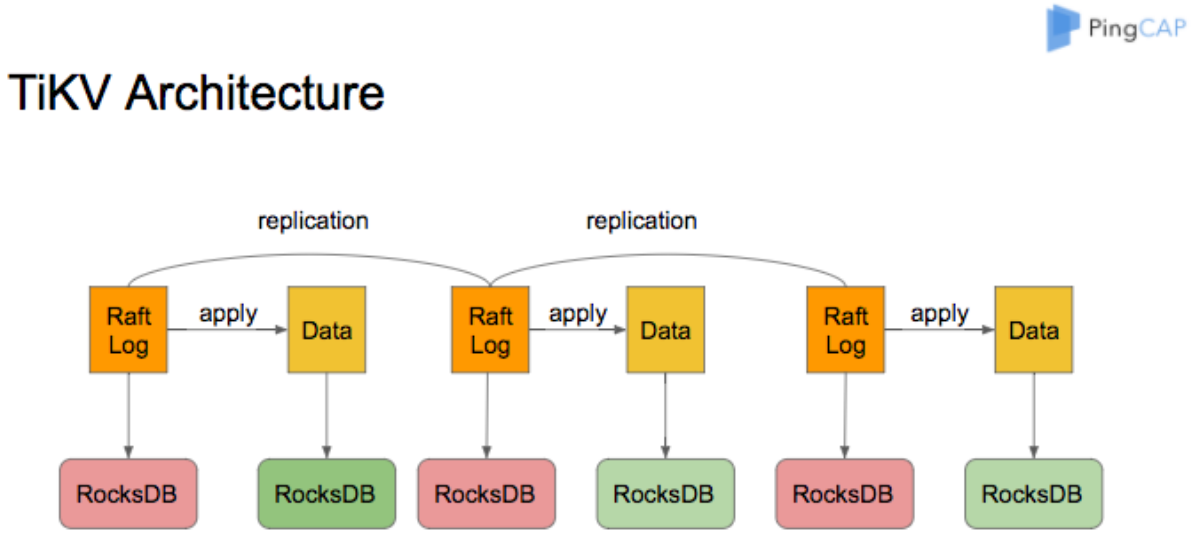


图 313: TiKV RocksDB

RocksDB 作为 TiKV 的核心存储引擎，用于存储 Raft 日志以及用户数据。每个 TiKV 实例中有两个 RocksDB 实例，一个用于存储 Raft 日志（通常被称为 raftdb），另一个用于存储用户数据以及 MVCC 信息（通常被称为 kvdb）。kvdb 中有四个 ColumnFamily：raft、lock、default 和 write：

- raft 列：用于存储各个 Region 的元信息。仅占极少量空间，用户可以不必关注。
- lock 列：用于存储悲观事务的悲观锁以及分布式事务的一阶段 Prewrite 锁。当用户的事务提交之后，lock cf 中对应的数据会很快删除掉，因此大部分情况下 lock cf 中的数据也很少（少于 1GB）。如果 lock cf 中的数据大量增加，说明有大量事务等待提交，系统出现了 bug 或者故障。
- write 列：用于存储用户真实的写入数据以及 MVCC 信息（该数据所属事务的开始时间以及提交时间）。当用户写入了一行数据时，如果该行数据长度小于 255 字节，那么会被存储 write 列中，否则的话该行数据会被存入到 default 列中。由于 TiDB 的非 unique 索引存储的 value 为空，unique 索引存储的 value 为主键索引，因此二级索引只会占用 writecf 的空间。
- default 列：用于存储超过 255 字节长度的数据。

14.2.2.2 RocksDB 的内存占用

为了提高读取性能以及减少对磁盘的读取，RocksDB 将存储在磁盘上的文件都按照一定大小切分成 block（默认是 64KB），读取 block 时先去内存中的 BlockCache 中查看该块数据是否存在，存在的话则可以直接从内存中读取而不必访问磁盘。

BlockCache 按照 LRU 算法淘汰低频访问的数据，TiKV 默认将系统总内存大小的 45% 用于 BlockCache，用户也可以自行修改 `storage.block-cache.capacity` 配置设置为合适的值，但是不建议超过系统总内存的 60%。

写入 RocksDB 中的数据会写入 MemTable，当一个 MemTable 的大小超过 128MB 时，会切换到一个新的 MemTable 来提供写入。TiKV 中一共有 2 个 RocksDB 实例，合计 4 个 ColumnFamily，每个 ColumnFamily 的单个 MemTable 大小限制是 128MB，最多允许 5 个 MemTable 存在，否则会阻塞前台写入，因此这部分占用的内存最多为 4 x 5 x 128MB = 2.5GB。这部分占用内存较少，不建议用户自行更改。

14.2.2.3 RocksDB 的空间占用

- 多版本：RocksDB 作为一个 LSM-tree 结构的键值存储引擎，MemTable 中的数据会首先被刷到 L0。L0 层的 SST 之间的范围可能存在重叠（因为文件顺序是按照生成的顺序排列），因此同一个 key 在 L0 中可能存在多个版本。当文件从 L0 合并到 L1 的时候，会按照一定大小（默认是 8MB）切割为多个文件，同一层的文件的范围互不重叠，所以 L1 及其以后的层每一层的 key 都只有一个版本。
- 空间放大：RocksDB 的每一层文件总大小都是上一层的 x 倍，在 TiKV 中这个配置默认是 10，因此 90% 的数据存储在最后一层，这也意味着 RocksDB 的空间放大不超过 1.11（L0 层的数据较少，可以忽略不计）。
- TiKV 的空间放大：TiKV 在 RocksDB 之上还有一层自己的 MVCC，当用户写入一个 key 的时候，实际上写入到 RocksDB 的是 key + commit_ts，也就是说，用户的更新和删除都是会写入新的 key 到 RocksDB。TiKV 每隔一段时间会删除旧版本的数据（通过 RocksDB 的 Delete 接口），因此可以认为用户存储在 TiKV 上的数据的实际空间放大为，1.11 加最近 10 分钟内写入的数据（假设 TiKV 回收旧版本数据足够及时）。详情见《TiDB in Action》。

14.2.2.4 RocksDB 后台线程与 Compact

RocksDB 中，将内存中的 MemTable 转化为磁盘上的 SST 文件，以及合并各个层级的 SST 文件等操作都是在后台线程池中执行的。后台线程池的默认大小是 8，当机器 CPU 数量小于等于 8 时，则后台线程池默认大小为 CPU 数量减一。通常来说，用户不需要更改这个配置。如果用户在一个机器上部署了多个 TiKV 实例，或者机器的读负载比较高而写负载比较低，那么可以适当调低 `rocksdb/max-background-jobs` 至 3 或者 4。

14.2.2.5 WriteStall

RocksDB 的 L0 与其他层不同，L0 的各个 SST 是按照生成顺序排列，各个 SST 之间的 key 范围存在重叠，因此查询的时候必须依次查询 L0 中的每一个 SST。为了不影响查询性能，当 L0 中的文件数量过多时，会触发 WriteStall 阻塞写入。

如果用户遇到了写延迟突然大幅度上涨，可以先查看 Grafana RocksDB KV 面板 WriteStall Reason 指标，如果是 L0 文件数量过多引起的 WriteStall，可以调整下面几个配置到 64，详细见《TiDB in Action》。

```
rocksdb.defaultcf.level0-slowdown-writes-trigger
rocksdb.writecf.level0-slowdown-writes-trigger
rocksdb.lockcf.level0-slowdown-writes-trigger
rocksdb.defaultcf.level0-stop-writes-trigger
rocksdb.writecf.level0-stop-writes-trigger
rocksdb.lockcf.level0-stop-writes-trigger
```

14.2.3 Titan 介绍

Titan 是基于 RocksDB 的高性能单机 key-value 存储引擎插件。

当 value 较大（1 KB 以上或 512 B 以上）的时候，Titan 在写、更新和点读等场景下性能都优于 RocksDB。但与此同时，Titan 会占用更多硬盘空间和部分舍弃范围查询。随着 SSD 价格的降低，Titan 的优势会更加突出，让用户更容易做出选择。

14.2.3.1 核心特性

- 支持将 value 从 LSM-tree 中分离出来单独存储，以降低写放大。
- 已有 RocksDB 实例可以平滑地升级到 Titan，这意味着升级过程不需要人工干预，并且不会影响线上服务。
- 100% 兼容目前 TiKV 所使用的所有 RocksDB 的特性。

14.2.3.2 适用场景

Titan 适合在以下场景中使用：

- 前台写入量较大，RocksDB 大量触发 compaction 消耗大量 I/O 带宽或者 CPU 资源，造成 TiKV 前台读写性能较差。
- 前台写入量较大，由于 I/O 带宽瓶颈或 CPU 瓶颈的限制，RocksDB compaction 进度落后较多频繁造成 write stall。
- 前台写入量较大，RocksDB 大量触发 compaction 造成 I/O 写入量较大，影响 SSD 盘的寿命。

开启 Titan 需要考虑以下前提条件：

- Value 较大。即 value 平均大小比较大，或者数据中大 value 的数据总大小占比比较大。目前 Titan 默认 1KB 以上大小的 value 是大 value，根据实际情况 512B 以上大小的 value 也可以看作是大 value。注：由于 TiKV Raft 层的限制，写入 TiKV 的 value 大小还是无法超过 8MB 的限制，可通过`raft-entry-max-size`配置项调整该限制。
- 没有范围查询或者对范围查询性能不敏感。Titan 存储数据的顺序性较差，所以相比 RocksDB 范围查询的性能较差，尤其是大范围查询。在测试中 Titan 范围查询性能相比 RocksDB 下降 40% 到数倍不等。
- 磁盘剩余空间足够，推荐为相同数据量下 RocksDB 磁盘占用的两倍。Titan 降低写放大是通过牺牲空间放大达到的。另外由于 Titan 逐个压缩 value，压缩率比 RocksDB（逐个压缩 block）要差。这两个因素一起造成 Titan 占用磁盘空间比 RocksDB 要多，这是正常现象。根据实际情况和不同的配置，Titan 磁盘空间占用可能会比 RocksDB 多一倍。

性能提升请参考 Titan 的设计与实现。

14.2.3.3 架构与实现

Titan 的基本架构如下图所示：

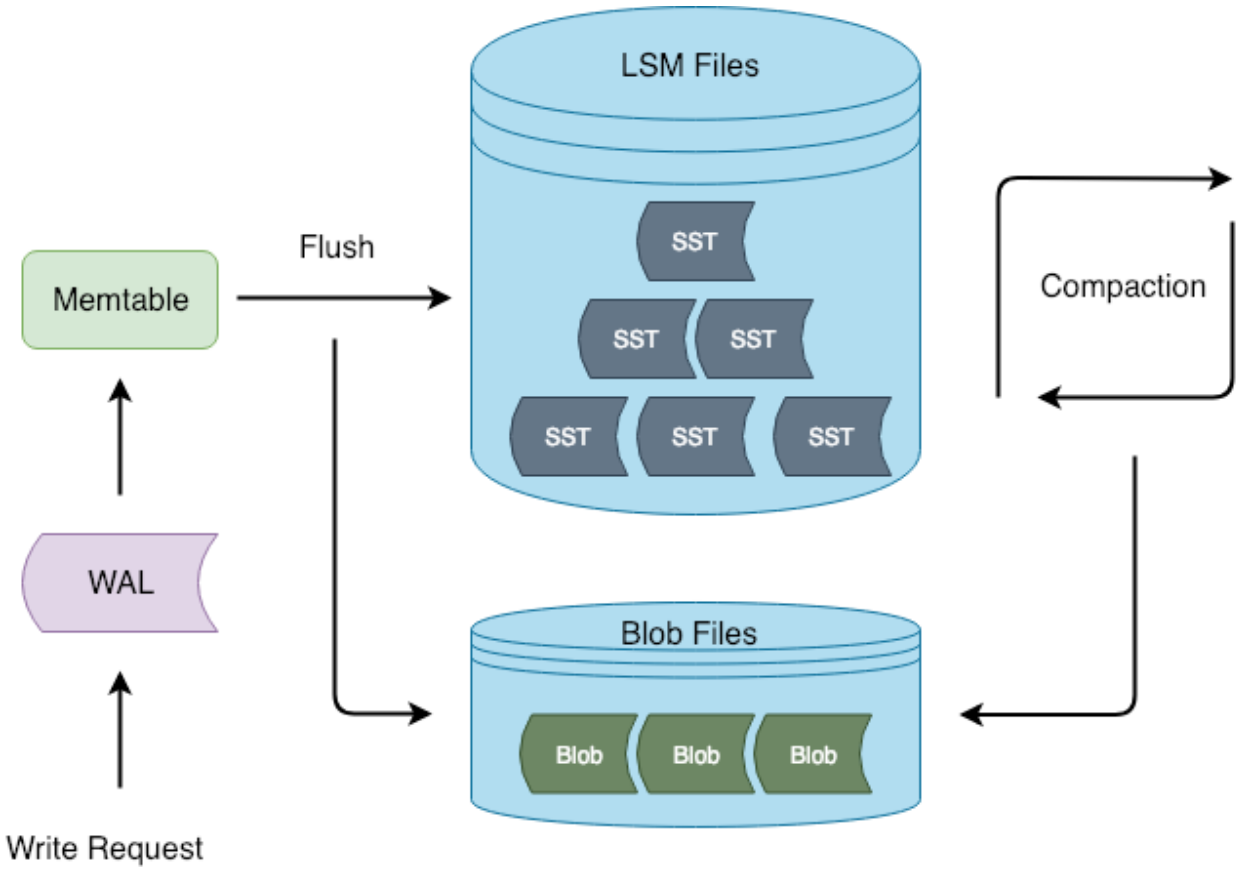


图 314: Architecture

Titan 在 Flush 和 Compaction 的时候将 value 分离出 LSM-tree，这样写入流程可以和 RocksDB 保持一致，减少对 RocksDB 的侵入性改动。

14.2.3.3.1 BlobFile

BlobFile 是用来存放从 LSM-tree 中分离出来的 value 的文件，其格式如下图所示：

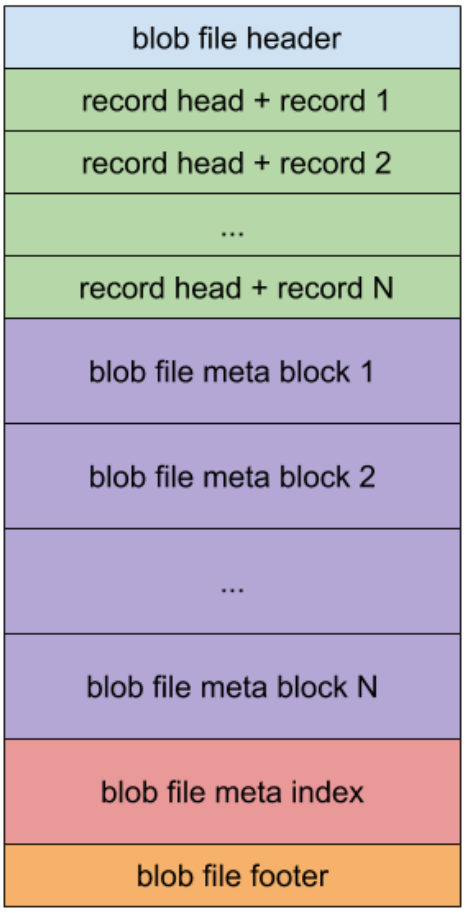


图 315: BlobFile

BlobFile 由 blob record 、meta block、meta index block 和 footer 组成。其中每个 blob record 用于存放一个 key-value 对；meta block 支持可扩展性，可以用来存放和 BlobFile 相关的一些属性；meta index block 用于检索 meta block。

BlobFile 的实现上有几点值得关注的地方：

- BlobFile 中的 key-value 是有序存放的，目的是在实现 iterator 的时候可以通过 prefetch 的方式提高顺序读取的性能。
- 每个 blob record 都保留了 value 对应的 user key 的拷贝，这样做的目的是在进行 GC 的时候，可以通过查询 user key 是否更新来确定对应 value 是否已经过期，但同时也带来了一定的写放大。
- BlobFile 支持 blob record 粒度的 compression，并且支持多种 compression algorithm，包括 Snappy、LZ4 和 Zstd 等，目前 Titan 默认使用的 compression algorithm 是 LZ4。

14.2.3.3.2 TitanTableBuilder

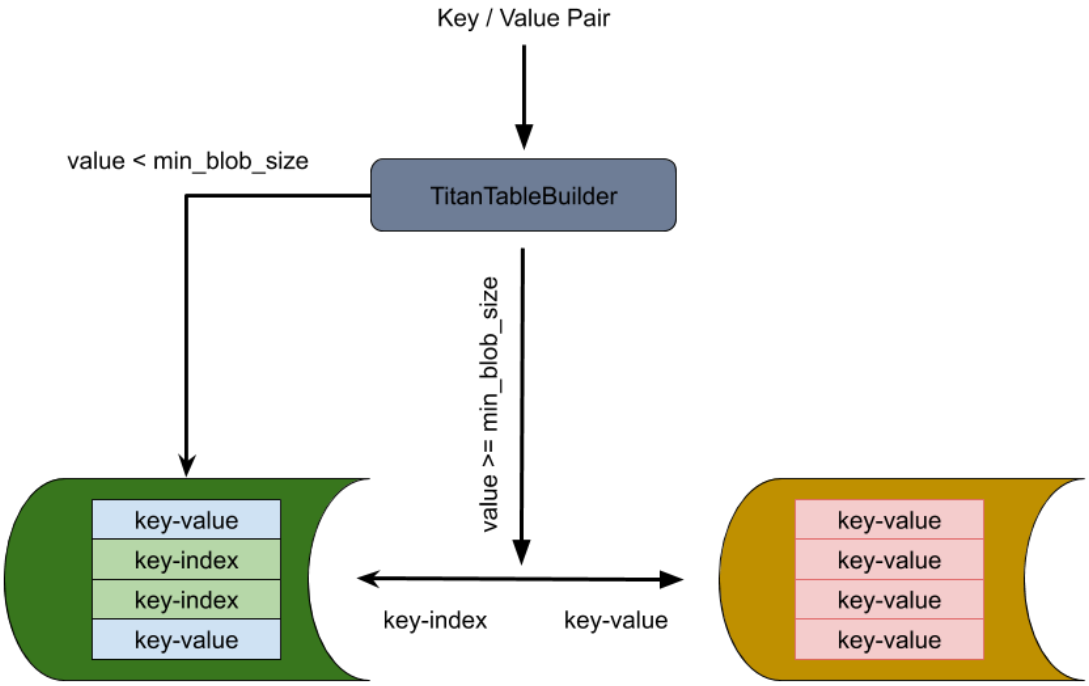


图 316: TitanTableBuilder

TitanTableBuilder 是实现分离 key-value 的关键，它通过判断 value size 的大小来决定是否将 value 分离到 BlobFile 中去。如果 value size 大于等于 `min_blob_size` 则将 value 分离到 BlobFile，并生成 index 写入 SST；如果 value size 小于 `min_blob_size` 则将 value 直接写入 SST。

该流程还支持将 Titan 降级回 RocksDB。在 RocksDB 做 compaction 的时候，将分离出来的 value 重新写回新生成的 SST 文件中。

14.2.3.4 Garbage Collection

Garbage Collection (GC) 的目的是回收空间。由于在 LSM-tree compaction 进行回收 key 时，储存在 blob 文件中的 value 并不会一同被删除，因此需要 GC 定期来将已经作废的 value 删除掉。在 Titan 中有两种 GC 方式可供选择：

- 定期整合重写 Blob 文件将作废的 value 剔除（传统 GC）
- 在 LSM-tree compaction 的时候同时进行 blob 文件的重写 (Level-Merge)

14.2.3.4.1 传统 GC

Titan 使用 RocksDB 的 TablePropertiesCollector 和 EventListener 来收集 GC 所需的统计信息。

TablePropertiesCollector

RocksDB 允许使用自定义的 TablePropertiesCollector 来搜集 SST 上的 properties 并写入到对应文件中去。Titan 通过一个自定义的 TablePropertiesCollector —— BlobFileSizeCollector 来搜集每个 SST 中有多少数据是存放在哪些 BlobFile 上的，将它收集到的 properties 命名为 BlobFileSizeProperties，它的工作流程和数据格式如下图所示：

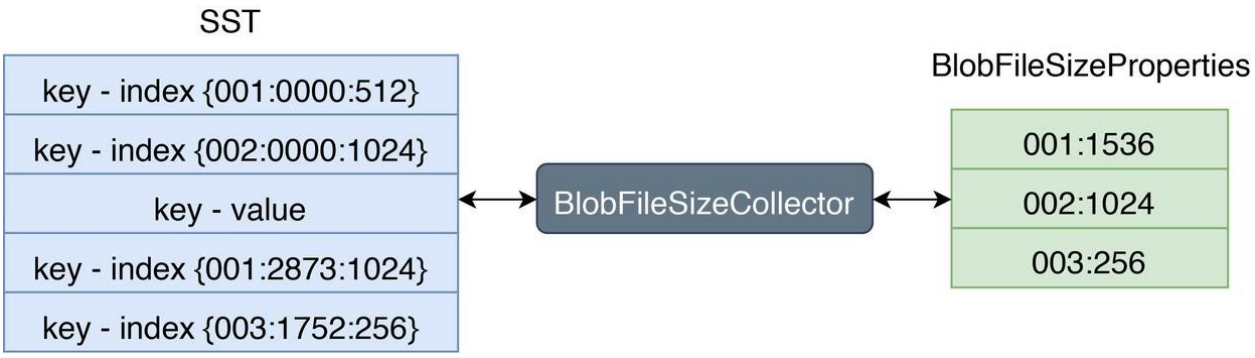


图 317: BlobFileSizeProperties

左边 SST 中 Index 的格式为：第一列代表 BlobFile 的文件 ID，第二列代表 blob record 在 BlobFile 中的 offset，第三列代表 blob record 的 size。右边 BlobFileSizeProperties 中的每一行代表一个 BlobFile 以及 SST 中有多少数据保存在这个 BlobFile 中，第一列代表 BlobFile 的文件 ID，第二列代表数据大小。

EventListener

RocksDB 是通过 Compaction 来丢弃旧版本数据以回收空间的，因此每次 Compaction 完成后 Titan 中的某些 BlobFile 中便可能有部分或全部数据过期。因此便可以通过监听 Compaction 事件来触发 GC，搜集比对 Compaction 中输入输出 SST 的 BlobFileSizeProperties 来决定挑选哪些 BlobFile 进行 GC。其流程大概如下图所示：

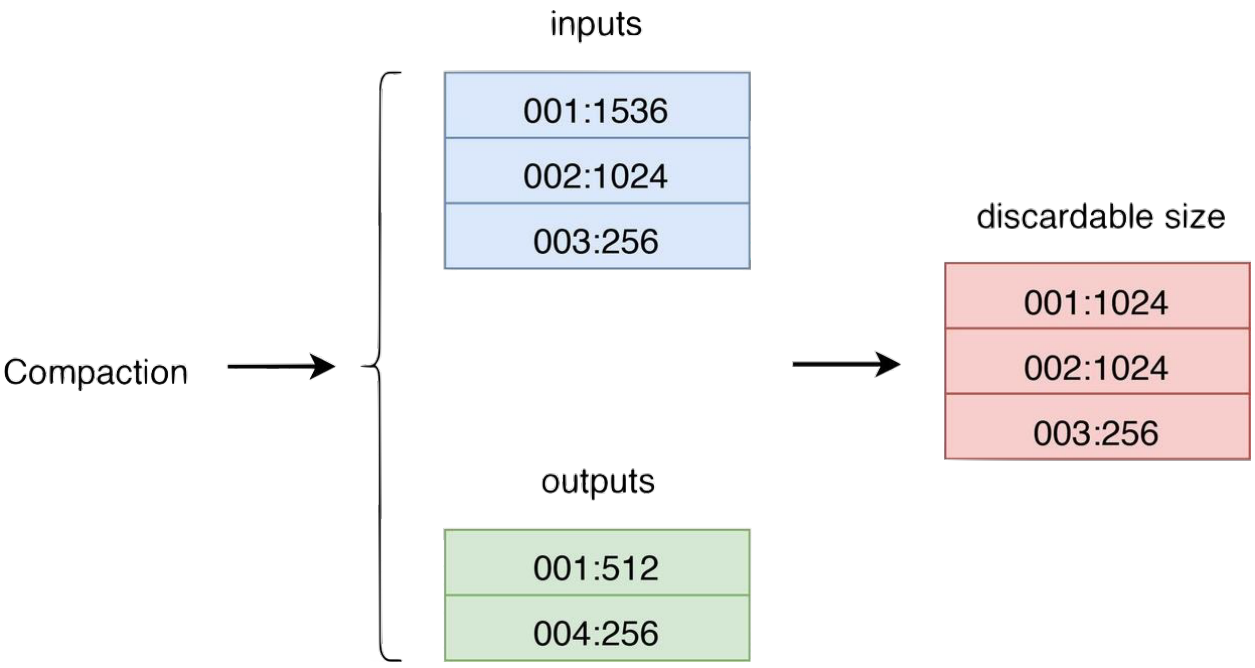


图 318: EventListener

inputs 代表参与 Compaction 的所有 SST 的 BlobFileSizeProperties，outputs 代表 Compaction 生成的所有 SST 的 BlobFileSizeProperties，discardable size 是通过计算 inputs 和 outputs 得出的每个 BlobFile 被丢弃的数据大小，第一列代表 BlobFile 的文件 ID，第二列代表被丢弃的数据大小。

Titan 会为每个有效的 BlobFile 在内存中维护一个 discardable size 变量，每次 Compaction 结束之后都对相应的 BlobFile 的 discardable size 变量进行累加。注意，在每次重启后会扫描一遍所有的 SST 的 BlobFileSizeProperties 重新构建每个有效 BlobFile 的 discardable size 变量。每次 GC 开始时就可以通过挑选 discardable size 最大的几个 BlobFile 来作为候选的文件。为了减小写放大，我们可以容忍一定的空间放大，所以 Titan 只有在 BlobFile 可丢弃的数据达到一定比例之后才会对其进行 GC。

GC 的方式就是对于这些选中的 BlobFile 文件，依次通过查询其中每个 value 相应的 key 的 blob index 是否存在或者更新来确定该 value 是否作废，最终将未作废的 value 归并排序生成新的 BlobFile，并将这些 value 更新后的 blob index 通过 WriteCallback 或者 MergeOperator 的方式写回到 SST 中。在完成 GC 后，这些原来的 BlobFile 文件并不会立即被删除，Titan 会在写回 blob index 后记录 RocksDB 最新的 sequence number，等到最旧 snapshot 的 sequence 超过这个记录的 sequence number 时 BlobFile 才能被删除。这个是因为在写回 blob index 后，还是可能通过之前的 snapshot 访问到老的 blob index，因此需要确保没有 snapshot 会访问到这个老的 blob index 后才能安全删除相应 BlobFile。

14.2.3.4.2 Level Merge

Level Merge 是 Titan 新加入的一种策略，它的核心思想是 LSM-tree 在进行 Compaction 的同时，对 SST 文件对应的 BlobFile 进行归并重写产生新的 BlobFile。其流程大概如下图所示：

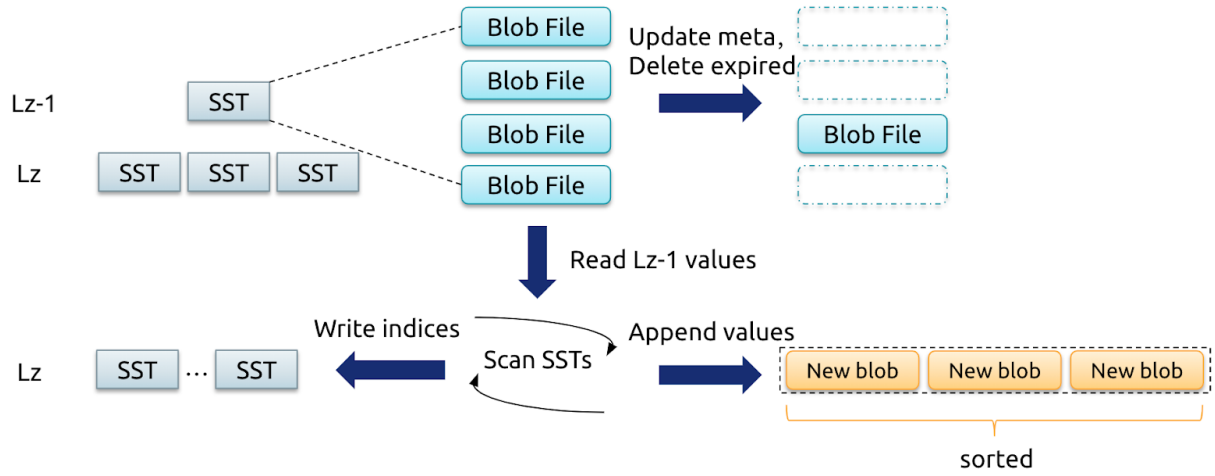


图 319: LevelMerge

Level z-1 和 Level z 的 SST 进行 Compaction 时会对 KV 对有序读写一遍，这时就可以对这些 SST 中所涉及的 BlobFile 的 value 有序写到新的 BlobFile 中，并在生成新的 SST 时将 key 的 blob index 进行更新。对于 Compaction 中被删除的 key，相应的 value 也不会写到新的 BlobFile 中，相当于完成了 GC。

相比于传统 GC，Level Merge 这种方式在 LSM-tree 进行 Compaction 的同时就完成了 Blob GC，不再需要查询 LSM-tree 的 blob index 情况和写回新 blob index 到 LSM-tree 中，减小了 GC 对前台操作影响。同时通过不断的重写 BlobFile，减小了 BlobFile 之间的相互重叠，提高系统整体有序性，也就是提高了 Scan 性能。当然将 BlobFile 以类似 tiering compaction 的方式分层会带来写放大，考虑到 LSM-tree 中 99% 的数据都落在最后两层，因此 Titan 仅对 LSM-tree 中 Compaction 到最后两层数据对应的 BlobFile 进行 Level Merge。

Range Merge

Range Merge 是基于 Level Merge 的一个优化。考虑如下两种情况，会导致最底层的有序性越来越差：

- 开启 `level_compaction_dynamic_level_bytes`，此时 LSM-tree 各层动态增长，随数据量增大最后一层的 sorted run 会越来越多。
- 某个 range 被频繁 Compaction 导致该 range 的 sorted runs 较多。

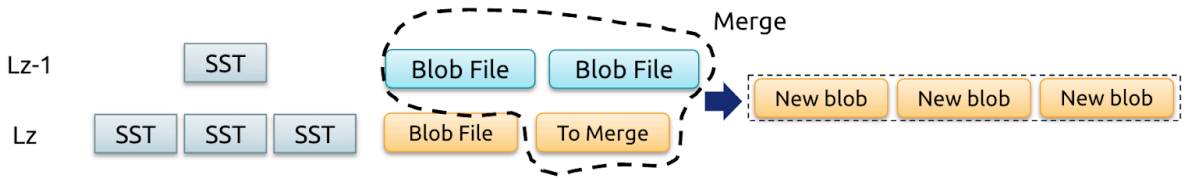


图 320: RangeMerge

因此需要通过 Range Merge 操作维持 sorted run 在一定水平，即在 OnCompactionComplete 时统计该 range 的 sorted run 数量，若数量过多则将涉及的 BlobFile 标记为 ToMerge，在下一次的 Compaction 中进行重写。

14.2.4 Titan 配置

本文档介绍如何通过Titan 配置项来开启、关闭 Titan、相关参数以及 Level Merge 功能。

14.2.4.1 开启 Titan

Titan 对 RocksDB 兼容，也就是说，使用 RocksDB 存储引擎的现有 TiKV 实例可以直接开启 Titan。

- 方法一：如果使用 TiUP 部署的集群，开启的方法是执行 `tiup cluster edit-config ${cluster-name}` 命令，再编辑 TiKV 的配置文件。编辑 TiKV 配置文件示例如下：

```
  tikv:
    rocksdb.titan.enabled: true
```

  重新加载配置，同时也会在线滚动重启 TiKV：

```
tiup cluster reload ${cluster-name} -R tikv
```

  具体命令，可参考通过 TiUP 修改配置参数。

- 方法二：直接编辑 TiKV 配置文件开启 Titan（生产环境不推荐）。

```
[rocksdb.titan]
enabled = true
```

开启 Titan 以后，原有的数据并不会马上移入 Titan 引擎，而是随着前台写入和 RocksDB compaction 的进行，逐步进行 key-value 分离并写入 Titan。可以通过观察 TiKV Details - Titan kv - blob file size 监控面版确认数据保存在 Titan 中部分的大小。

如果需要加速数据移入 Titan，可以通过 tikv-ctl 执行一次全量 compaction，具体参考手动 compact。

> **警告：**
>
> 在不开启 Titan 功能的情况下，RocksDB 无法读取已经迁移到 Titan 的数据。如果在打开过 Titan 的 TiKV 实例上错误地关闭了 Titan（误设置 `rocksdb.titan.enabled = false`），启动 TiKV 会失败，TiKV log 中出现 `You have disabled titan when its data directory is not empty` 错误。如需要关闭 Titan，参考关闭 Titan 一节。

14.2.4.2 相关参数介绍

使用 TiUP 调整参数，请参考修改配置参数。

- Titan GC 线程数。

  当从 TiKV Details - Thread CPU - RocksDB CPU 监控中观察到 Titan GC 线程长期处于满负荷状态时，应该考虑增加 Titan GC 线程池大小。

```
[rocksdb.titan]
max-background-gc = 1
```

- value 的大小阈值。

  当写入的 value 小于这个值时，value 会保存在 RocksDB 中，反之则保存在 Titan 的 blob file 中。根据 value 大小的分布，增大这个值可以使更多 value 保存在 RocksDB，读取这些小 value 的性能会稍好一些；减少这个值可以使更多 value 保存在 Titan 中，进一步减少 RocksDB compaction。

```
[rocksdb.defaultcf.titan]
min-blob-size = "1KB"
```

- Titan 中 value 所使用的压缩算法。Titan 中压缩是以 value 为单元的。

```
[rocksdb.defaultcf.titan]
blob-file-compression = "lz4"
```

- Titan 中 value 的缓存大小。

  更大的缓存能提高 Titan 读性能，但过大的缓存会造成 OOM。建议在数据库稳定运行后，根据监控把 RocksDB block cache (`storage.block-cache.capacity`) 设置为 store size 减去 blob file size 的大小，`blob-cache-size` 设置为 内存大小 * 50% 再减去 block cache 的大小。这是为了保证 block cache 足够缓存整个 RocksDB 的前提下，blob cache 尽量大。

```
[rocksdb.defaultcf.titan]
blob-cache-size = 0
```

- 当一个 blob file 中无用数据（相应的 key 已经被更新或删除）比例超过以下阈值时，将会触发 Titan GC。

```
discardable-ratio = 0.5
```

将此文件有用的数据重写到另一个文件。这个值可以估算 Titan 的写放大和空间放大的上界（假设关闭压缩）。公式是：

写放大上界 = 1 / discardable_ratio

空间放大上界 = 1 / (1 - discardable_ratio)

可以看到，减少这个阈值可以减少空间放大，但是会造成 Titan 更频繁 GC；增加这个值可以减少 Titan GC，减少相应的 I/O 带宽和 CPU 消耗，但是会增加磁盘空间占用。

- 以下选项限制 RocksDB compaction 的 I/O 速率，以达到在流量高峰时，限制 RocksDB compaction 减少其 I/O 带宽和 CPU 消耗对前台读写性能的影响。

  当开启 Titan 时，该选项限制 RocksDB compaction 和 Titan GC 的 I/O 速率总和。当发现在流量高峰时 RocksDB compaction 和 Titan GC 的 I/O 和/或 CPU 消耗过大，可以根据磁盘 I/O 带宽和实际写入流量适当配置这个选项。

```
[rocksdb]
rate-bytes-per-sec = 0
```

14.2.4.3 关闭 Titan

通过设置 `rocksdb.defaultcf.titan.blob-run-mode` 参数可以关闭 Titan。`blob-run-mode` 可以设置为以下几个值之一：

- 当设置为 `normal` 时，Titan 处于正常读写的状态。
- 当设置为 `read-only` 时，新写入的 value 不论大小均会写入 RocksDB。
- 当设置为 `fallback` 时，新写入的 value 不论大小均会写入 RocksDB，并且当 RocksDB 进行 compaction 时，会自动把所碰到的存储在 Titan blob file 中的 value 移回 RocksDB。

如果现有数据和未来数据均不再需要 Titan，可执行以下步骤完全关闭 Titan：

1. 更新需要关闭 Titan 的 TiKV 节点的配置。你可以通过以下两种方式之一更新 TiKV 配置：

   - 执行 `tiup cluster edit-config`，编辑配置文件，再执行 `tiup cluster reload -R tikv`。
   - 手动修改 TiKV 配置文件，然后重启 TiKV。

```
[rocksdb.defaultcf.titan]
blob-run-mode = "fallback"
discardable-ratio = 1.0
```

2. 使用 `tikv-ctl` 执行全量数据整理 (Compaction)。这一步骤将消耗大量 I/O 和 CPU 资源。

```
tikv-ctl --pd <PD_ADDR> compact-cluster --bottommost force
```

3. 数据整理结束后，通过 TiKV-Details/Titan - kv 监控面板确认 Blob file count 指标降为 0。

4. 更新 TiKV 节点的配置，关闭 Titan。

```
[rocksdb.titan]
enabled = false
```

14.2.4.4 Level Merge（实验功能）

TiKV 4.0 中 Titan 提供新的算法提升范围查询性能并降低 Titan GC 对前台写入性能的影响。这个新的算法称为Level Merge。Level Merge 可以通过以下选项开启：

```
[rocksdb.defaultcf.titan]
level-merge = true
```

开启 Level Merge 的好处如下：

- 大幅提升 Titan 的范围查询性能。
- 减少了 Titan GC 对前台写入性能的影响，提升写入性能。
- 减少 Titan 空间放大，减少磁盘空间占用（默认配置下的比较）。

相应地，Level Merge 的写放大会比 Titan 稍高，但依然低于原生的 RocksDB。

14.2.5 Partitioned Raft KV

> **警告：**
>
> Partitioned Raft KV 目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现 bug，请在 GitHub 上提 issue 反馈。

v6.6.0 之前，基于 Raft 的存储引擎，TiKV 使用单个 RocksDB 实例存储该 TiKV 实例所有 Region 的数据。

为了更平稳地支持更大的集群，从 v6.6.0 开始，TiDB 引入了一个全新的 TiKV 存储引擎，该引擎使用多个 RocksDB 实例来存储 TiKV 的 Region 数据，每个 Region 的数据都独立存储在单个 RocksDB 实例中。

新的 TiKV 引擎能够更好地控制 RocksDB 实例的文件数和层级，并实现了 Region 间数据操作的物理隔离，避免相互影响。同时，该引擎支持平稳管理更多的数据。你可以理解为，TiKV 通过分区管理多个 RocksDB 实例，这也是该特性 Partitioned Raft KV 名字的由来。

14.2.5.1 使用场景

如果你的 TiKV 集群有以下特点，可以考虑使用该功能：

- 需要在单个 TiKV 实例支持更多的数据。
- 具有大量写入吞吐。
- 需要频繁地扩缩容。

- 负载有较为严重的读写放大。
- TiKV 内存尚有富余。

该功能的主要优势在于，提高写入性能，加快扩缩容速度，以及在相同硬件下支持更多数据和更大的集群。

14.2.5.2 使用方法

要启用 Partitioned Raft KV，需要在创建集群时将配置项`storage.engine` 设为 `"partitioned-raft-kv"`。同时，在使用 Partitioned Raft KV 特性时，可以通过配置项`rocksdb.write-buffer-flush-oldest-first` 和`rocksdb.write-buffer-limit` 来控制 RocksDB 的内存使用。

14.2.5.3 使用限制

由于该功能为实验特性，目前有以下限制：

- 暂不支持 TiDB Lightning、 TiCDC、 BR、 Dumping 等数据导入、同步和备份工具。
- 暂不支持 tikv-ctl 命令行管理工具。
- 不支持同时和 TiFlash 使用。
- 需要在创建集群时启用，不支持在集群创建后开启。

14.3 存储引擎 TiFlash

14.3.1 TiFlash 简介

TiFlash 是 TiDB HTAP 形态的关键组件，它是 TiKV 的列存扩展，在提供了良好的隔离性的同时，也兼顾了强一致性。列存副本通过 Raft Learner 协议异步复制，但是在读取的时候通过 Raft 校对索引配合 MVCC 的方式获得 Snapshot Isolation 的一致性隔离级别。这个架构很好地解决了 HTAP 场景的隔离性以及列存同步的问题。

14.3.1.1 整体架构

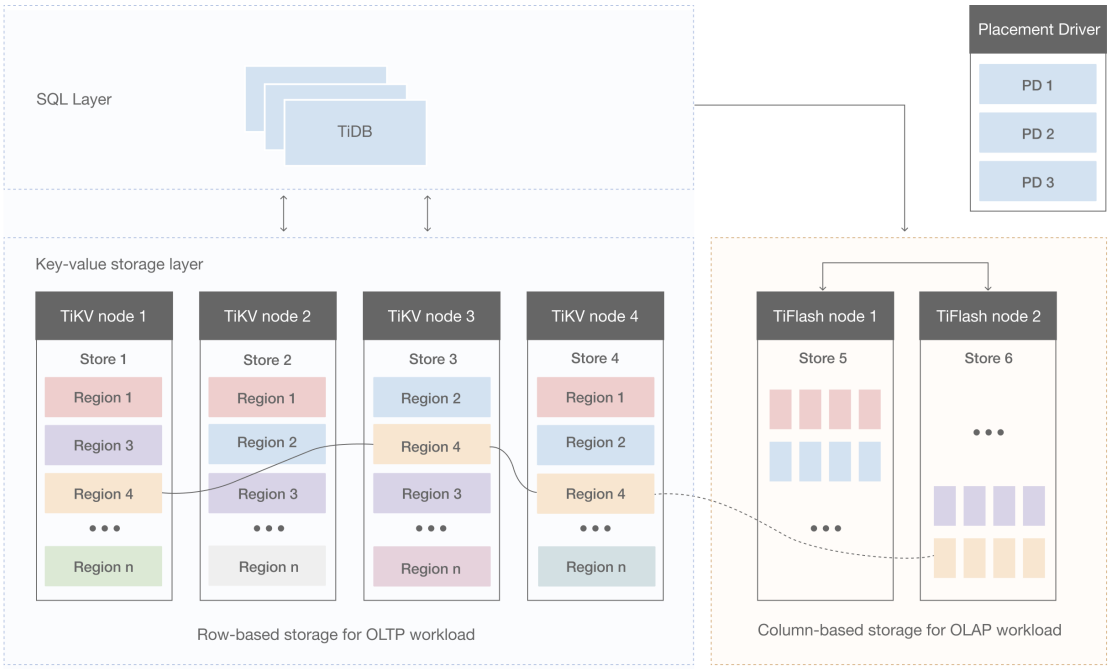


图 321: TiFlash 架构

上图为 TiDB HTAP 形态架构，其中包含 TiFlash 节点。

TiFlash 提供列式存储，且拥有借助 ClickHouse 高效实现的协处理器层。除此以外，它与 TiKV 非常类似，依赖同样的 Multi-Raft 体系，以 Region 为单位进行数据复制和分散（详情见《说存储》一文）。

TiFlash 以低消耗不阻塞 TiKV 写入的方式，实时复制 TiKV 集群中的数据，并同时提供与 TiKV 一样的一致性读取，且可以保证读取到最新的数据。TiFlash 中的 Region 副本与 TiKV 中完全对应，且会跟随 TiKV 中的 Leader 副本同时进行分裂与合并。

在 Linux AMD64 架构的硬件平台部署 TiFlash 时，CPU 必须支持 AVX2 指令集。确保命令 `cat /proc/cpuinfo | grep avx2` 有输出。而在 Linux ARM64 架构的硬件平台部署 TiFlash 时，CPU 必须支持 ARMv8 架构。确保命令 `cat /proc/cpuinfo | grep 'crc32' | grep 'asimd'` 有输出。通过使用向量扩展指令集，TiFlash 的向量化引擎能提供更好的性能。

TiFlash 可以兼容 TiDB 与 TiSpark，用户可以选择使用不同的计算引擎。

TiFlash 推荐使用和 TiKV 不同的节点以做到 Workload 隔离，但在无业务隔离的前提下，也可以选择与 TiKV 同节点部署。

TiFlash 暂时无法直接接受数据写入，任何数据必须先写入 TiKV 再同步到 TiFlash。TiFlash 以 learner 角色接入 TiDB 集群，TiFlash 支持表粒度的数据同步，部署后默认情况下不会同步任何数据，需要按照按表构建 TiFlash 副本一节完成指定表的数据同步。

TiFlash 主要包含三个组件，除了主要的存储引擎组件，另外包含 tiflash proxy 和 pd buddy 组件，其中 tiflash proxy 主要用于处理 Multi-Raft 协议通信的相关工作，pd buddy 负责与 PD 协同工作，将 TiKV 数据按表同步到 TiFlash。

对于按表构建 TiFlash 副本的流程，TiDB 接收到相应的 DDL 命令后 pd buddy 组件会通过 TiDB 的 status 端口获取到需要同步的数据表信息，然后会将需要同步的数据信息发送到 PD，PD 根据该信息进行相关的数据调度。

14.3.1.2 核心特性

TiFlash 主要有异步复制、一致性、智能选择、计算加速等几个核心特性。

14.3.1.2.1 异步复制

TiFlash 中的副本以特殊角色 (Raft Learner) 进行异步的数据复制。这表示当 TiFlash 节点宕机或者网络高延迟等状况发生时，TiKV 的业务仍然能确保正常进行。

这套复制机制也继承了 TiKV 体系的自动负载均衡和高可用：并不用依赖附加的复制管道，而是直接以多对多方式接收 TiKV 的数据传输；且只要 TiKV 中数据不丢失，就可以随时恢复 TiFlash 的副本。

14.3.1.2.2 一致性

TiFlash 提供与 TiKV 一样的快照隔离支持，且保证读取数据最新（确保之前写入的数据能被读取）。这个一致性是通过对数据进行复制进度校验做到的。

每次收到读取请求，TiFlash 中的 Region 副本会向 Leader 副本发起进度校对（一个非常轻的 RPC 请求），只有当进度确保至少所包含读取请求时间戳所覆盖的数据之后才响应读取。

14.3.1.2.3 智能选择

TiDB 可以自动选择使用 TiFlash 列存或者 TiKV 行存，甚至在同一查询内混合使用提供最佳查询速度。这个选择机制与 TiDB 选取不同索引提供查询类似：根据统计信息判断读取代价并作出合理选择。

14.3.1.2.4 计算加速

TiFlash 对 TiDB 的计算加速分为两部分：列存本身的读取效率提升以及为 TiDB 分担计算。其中分担计算的原理和 TiKV 的协处理器一致：TiDB 会将可以由存储层分担的计算下推。能否下推取决于 TiFlash 是否可以支持相关下推。具体介绍请参阅 “TiFlash 支持的计算下推”一节。

14.3.1.3 使用 TiFlash

TiFlash 部署完成后并不会自动同步数据，而需要手动指定需要同步的表。

你可以使用 TiDB 或者 TiSpark 读取 TiFlash，TiDB 适合用于中等规模的 OLAP 计算，而 TiSpark 适合大规模的 OLAP 计算，你可以根据自己的场景和使用习惯自行选择。具体参见：

- 构建 TiFlash 副本
- 使用 TiDB 读取 TiFlash
- 使用 TiSpark 读取 TiFlash
- 使用 MPP 模式

如果需要快速体验以 TPC-H 为例子，从导入到查询的完整流程，可以参考HTAP 快速上手指南。

14.3.1.4 另请参阅

- 全新部署一个包含 TiFlash 节点的集群，请参考使用 TiUP 部署 TiDB 集群
- 已有集群新增一个 TiFlash 节点，请参考扩容 TiFlash 节点
- TiFlash 常见运维操作
- TiFlash 性能调优
- TiFlash 配置参数介绍
- TiFlash 监控说明
- TiFlash 报警规则
- TiFlash 常见问题处理
- TiFlash 支持的计算下推
- TiFlash 数据校验
- TiFlash 兼容性说明

14.3.2 构建 TiFlash 副本

本文档介绍如何按表和库构建 TiFlash 副本，以及如何设置可用区来调度副本。

14.3.2.1 按表构建 TiFlash 副本

TiFlash 接入 TiKV 集群后，默认不会开始同步数据。可通过 MySQL 客户端向 TiDB 发送 DDL 命令来为特定的表建立 TiFlash 副本：

```
ALTER TABLE table_name SET TIFLASH REPLICA count;
```

该命令的参数说明如下：

- count 表示副本数，0 表示删除。

对于相同表的多次 DDL 命令，仅保证最后一次能生效。例如下面给出的操作 tpch50 表的两条 DDL 命令中，只有第二条删除副本的命令能生效：

为表建立 2 个副本：

```
ALTER TABLE `tpch50`.`lineitem` SET TIFLASH REPLICA 2;
```

删除副本：

```
ALTER TABLE `tpch50`.`lineitem` SET TIFLASH REPLICA 0;
```

注意事项：

- 假设有一张表 t 已经通过上述的 DDL 语句同步到 TiFlash，则通过以下语句创建的表也会自动同步到 TiFlash：

  ```
  CREATE TABLE table_name like t;
  ```

- 如果集群版本 < v4.0.6，若先对表创建 TiFlash 副本，再使用 TiDB Lightning 导入数据，会导致数据导入失败。需要在使用 TiDB Lightning 成功导入数据至表后，再对相应的表创建 TiFlash 副本。
- 如果集群版本以及 TiDB Lightning 版本均 >= v4.0.6，无论一个表是否已经创建 TiFlash 副本，你均可以使用 TiDB Lightning 导入数据至该表。但注意此情况会导致 TiDB Lightning 导入数据耗费的时间延长，具体取决于 TiDB Lightning 部署机器的网卡带宽、TiFlash 节点的 CPU 及磁盘负载、TiFlash 副本数等因素。
- 不推荐同步 1000 张以上的表，这会降低 PD 的调度性能。这个限制将在后续版本去除。
- v5.1 版本及后续版本将不再支持设置系统表的 replica。在集群升级前，需要清除相关系统表的 replica，否则升级到较高版本后将无法再修改系统表的 replica 设置。

14.3.2.1.1 查看表同步进度

可通过如下 SQL 语句查看特定表（通过 WHERE 语句指定，去掉 WHERE 语句则查看所有表）的 TiFlash 副本的状态：

```
SELECT * FROM information_schema.tiflash_replica WHERE TABLE_SCHEMA = '<db_name>' and TABLE_NAME
    ↪ = '<table_name>';
```

查询结果中：

- AVAILABLE 字段表示该表的 TiFlash 副本是否可用。1 代表可用，0 代表不可用。副本状态为可用之后就不再改变，如果通过 DDL 命令修改副本数则会重新计算同步进度。
- PROGRESS 字段代表同步进度，在 0.0~1.0 之间，1 代表至少 1 个副本已经完成同步。

14.3.2.2 按库构建 TiFlash 副本

类似于按表构建 TiFlash 副本的方式，你可以在 MySQL 客户端向 TiDB 发送 DDL 命令来为指定数据库中的所有表建立 TiFlash 副本：

```
ALTER DATABASE db_name SET TIFLASH REPLICA count;
```

在该命令中，count 表示 TiFlash 的副本数。当设置 count 值为 0 时，表示删除现有的 TiFlash 副本。

命令示例：

执行以下命令可以为 tpch50 库中的所有表建立 2 个 TiFlash 副本。

```
ALTER DATABASE `tpch50` SET TIFLASH REPLICA 2;
```

执行以下命令可以删除为 tpch50 库建立的 TiFlash 副本：

```
ALTER DATABASE `tpch50` SET TIFLASH REPLICA 0;
```

> 注意：

- 该命令实际是为用户执行一系列 DDL 操作，对资源要求比较高。如果在执行过程中出现中断，已经执行成功的操作不会回退，未执行的操作不会继续执行。
- 从命令执行开始到该库中所有表都已同步完成之前，不建议执行和该库相关的 TiFlash 副本数量设置或其他 DDL 操作，否则最终状态可能非预期。非预期场景包括：
  - 先设置 TiFlash 副本数量为 2，在库中所有的表都同步完成前，再设置 TiFlash 副本数量为 1，不能保证最终所有表的 TiFlash 副本数量都为 1 或都为 2。
  - 在命令执行到结束期间，如果在该库下创建表，则可能会对这些新增表创建 TiFlash 副本。
  - 在命令执行到结束期间，如果为该库下的表添加索引，则该命令可能陷入等待，直到添加索引完成。
- 该命令执行结束后，在该库中新建的表不会自动创建 TiFlash 副本。
- 该命令会跳过系统表、视图、临时表以及包含了 TiFlash 不支持字符集的表。

14.3.2.2.1 查看库同步进度

类似于按表构建，按库构建 TiFlash 副本的命令执行成功，不代表所有表都已同步完成。可以执行下面的 SQL 语句检查数据库中所有已设置 TiFlash Replica 表的同步进度：

```
SELECT * FROM information_schema.tiflash_replica WHERE TABLE_SCHEMA = '<db_name>';
```

可以执行下面的 SQL 语句检查数据库中尚未设置 TiFlash Replica 的表名：

```
SELECT TABLE_NAME FROM information_schema.tables where TABLE_SCHEMA = "<db_name>" and TABLE_NAME
    ↪ not in (SELECT TABLE_NAME FROM information_schema.tiflash_replica where TABLE_SCHEMA = "<
    ↪ db_name>");
```

14.3.2.3 加快 TiFlash 副本同步速度

新增 TiFlash 副本时，各个 TiKV 实例将进行全表数据扫描，并将扫描得到的数据快照发送给 TiFlash 从而形成副本。默认情况下，为了降低对 TiKV 及 TiFlash 线上业务的影响，TiFlash 新增副本速度较慢、占用资源较少。如果集群中 TiKV 及 TiFlash 的 CPU 和磁盘 IO 资源有富余，你可以按以下步骤操作来提升 TiFlash 副本同步速度：

1. 通过SQL 语句在线修改配置，临时调高各个 TiKV 及 TiFlash 实例的数据快照写入速度：

```
sql  -- 这两个参数默认值都为 100MiB，即用于副本同步的快照最大占用的磁盘带宽不超过 100MiB/s。
↪ SET CONFIG tikv `server.snap-io-max-bytes-per-sec` = '300MiB'; SET CONFIG tiflash `raftstore-
↪ proxy.server.snap-max-write-bytes-per-sec` = '300MiB';
```

以上 SQL 语句执行后，配置修改立即生效，无需重启集群。但由于副本同步速度还受到 PD 副本速度控制，因此当前你还无法观察到副本同步速度提升。

2. 使用PD Control 逐步放开新增副本速度限制：

TiFlash 默认新增副本速度是 30（每分钟大约 30 个 Region 将会新增 TiFlash 副本）。执行以下命令将调整所有 TiFlash 实例的新增副本速度到 60，即原来的 2 倍速度：

```
shell  tiup ctl:v<CLUSTER_VERSION> pd -u http://<PD_ADDRESS>:2379 store limit all engine tiflash
↪ 60 add-peer
```

> 上述命令中，需要将 v<CLUSTER_VERSION> 替换为该集群版本，例如 v6.5.0，<PD_ADDRESS>:2379 替换为任一 PD 节点的地址。替换后样例为：
>
> ```
> tiup ctl:v6.1.1 pd -u http://192.168.1.4:2379 store limit all engine tiflash 60 add
>     ↪ -peer
> ```

执行完毕后，几分钟内，你将观察到 TiFlash 节点的 CPU 及磁盘 IO 资源占用显著提升，TiFlash 将更快地创建副本。同时，TiKV 节点的 CPU 及磁盘 IO 资源占用也将有所上升。

如果此时 TiKV 及 TiFlash 节点的资源仍有富余，且线上业务的延迟没有显著上升，则可以考虑进一步放开调度速度，例如将新增副本的速度增加为原来的 3 倍：

```
shell  tiup ctl:v<CLUSTER_VERSION> pd -u http://<PD_ADDRESS>:2379 store limit all engine tiflash
↪ 90 add-peer
```

3. 在副本同步完毕后，恢复到默认配置，减少在线业务受到的影响。

执行以下 PD Control 命令可恢复默认的新增副本速度：

```
shell  tiup ctl:v<CLUSTER_VERSION> pd -u http://<PD_ADDRESS>:2379 store limit all engine tiflash
↪ 30 add-peer
```

执行以下 SQL 语句可恢复默认的数据快照写入速度：

```
sql  SET CONFIG tikv `server.snap-io-max-bytes-per-sec` = '100MiB'; SET CONFIG tiflash `raftstore
↪ -proxy.server.snap-max-write-bytes-per-sec` = '100MiB';
```

14.3.2.4 设置可用区

在配置副本时，如果为了考虑容灾，需要将 TiFlash 的不同数据副本分布到多个数据中心，则可以按如下步骤进行配置：

1. 在集群配置文件中为 TiFlash 节点指定 label：

```
tiflash_servers:
  - host: 172.16.5.81
    config:
      logger.level: "info"
    learner_config:
      server.labels:
```

```
      zone: "z1"
  - host: 172.16.5.82
    config:
      logger.level: "info"
    learner_config:
      server.labels:
        zone: "z1"
  - host: 172.16.5.85
    config:
      logger.level: "info"
    learner_config:
      server.labels:
        zone: "z2"
```

注：旧版本中的 `flash.proxy.labels` 配置无法处理可用区名字中的特殊字符，建议使用 `learner_config` 中的 `server.labels` 来进行配置。

2. 启动集群后，在创建副本时为副本调度指定 label，语法如下：

```
ALTER TABLE table_name SET TIFLASH REPLICA count LOCATION LABELS location_labels;
```

例如：

```
ALTER TABLE t SET TIFLASH REPLICA 2 LOCATION LABELS "zone";
```

3. 此时 PD 会根据设置的 label 进行调度，将表 t 的两个副本分别调度到两个可用区中。可以通过监控或 pd-ctl 来验证这一点：

```
> tiup ctl:v<CLUSTER_VERSION> pd -u http://<PD_ADDRESS>:2379 store

    ...

    "address": "172.16.5.82:23913",
    "labels": [
      { "key": "engine", "value": "tiflash"},
      { "key": "zone", "value": "z1" }
    ],
    "region_count": 4,

    ...

    "address": "172.16.5.81:23913",
    "labels": [
      { "key": "engine", "value": "tiflash"},
      { "key": "zone", "value": "z1" }
    ],
    "region_count": 5,
```

```
    ...

    "address": "172.16.5.85:23913",
    "labels": [
      { "key": "engine", "value": "tiflash"},
      { "key": "zone", "value": "z2" }
    ],
    "region_count": 9,

    ...
```

关于使用 label 进行副本调度划分可用区的更多内容，可以参考通过拓扑 label 进行副本调度，同城多数据中心部署 TiDB 与两地三中心部署。

14.3.3 使用 TiDB 读取 TiFlash

本文档介绍如何使用 TiDB 读取 TiFlash 副本。

TiDB 提供三种读取 TiFlash 副本的方式。如果添加了 TiFlash 副本，而没有做任何 engine 的配置，则默认使用 CBO 方式。

14.3.3.1 智能选择

对于创建了 TiFlash 副本的表，TiDB 优化器会自动根据代价估算选择是否使用 TiFlash 副本。具体有没有选择 TiFlash 副本，可以通过 desc 或 explain analyze 语句查看，例如：

```
desc select count(*) from test.t;
```

```
+-------------------------+---------+--------------+--------------+--------------------------------+
    ↪
| id                      | estRows | task         | access object | operator info
    ↪                  |
+-------------------------+---------+--------------+--------------+--------------------------------+
    ↪
| StreamAgg_9             | 1.00    | root         |               | funcs:count(1)->Column#4
    ↪       |
| └─TableReader_17        | 1.00    | root         |               | data:TableFullScan_16
    ↪          |
|   └─TableFullScan_16    | 1.00    | cop[tiflash] | table:t       | keep order:false, stats:
    ↪ pseudo |
+-------------------------+---------+--------------+--------------+--------------------------------+
    ↪
3 rows in set (0.00 sec)
```

```
explain analyze select count(*) from test.t;
```

```
+--------------------------+---------+---------+-------------+---------------+--------------------------------
    ↪
| id                       | estRows | actRows | task         | access object | execution info
    ↪                                                        | operator info                  |
    ↪ memory    | disk |
+--------------------------+---------+---------+-------------+---------------+--------------------------------
    ↪
| StreamAgg_9              | 1.00    | 1       | root         |               | time:83.8372ms,
    ↪ loops:2                                               | funcs:count(1)->Column#4       |
    ↪ 372 Bytes | N/A  |
| └─TableReader_17         | 1.00    | 1       | root         |               | time:83.7776ms,
    ↪ loops:2, rpc num: 1, rpc time:83.5701ms, proc keys:0 | data:TableFullScan_16          |
    ↪ 152 Bytes | N/A  |
|   └─TableFullScan_16     | 1.00    | 1       | cop[tiflash] | table:t       | tiflash_task:{
    ↪ time:43ms, loops:1, threads:1}, tiflash_scan:{...}     | keep order:false, stats:pseudo |
    ↪  N/A       | N/A  |
+--------------------------+---------+---------+-------------+---------------+--------------------------------
    ↪
```

cop[tiflash] 表示该任务会发送至 TiFlash 进行处理。如果没有选择 TiFlash 副本，可尝试通过 analyze table 语句更新统计信息后，再查看 explain analyze 结果。

需要注意的是，如果表仅有单个 TiFlash 副本且相关节点无法服务，智能选择模式下的查询会不断重试，需要指定 Engine 或者手工 Hint 来读取 TiKV 副本。

14.3.3.2 Engine 隔离

Engine 隔离是通过配置变量来指定所有的查询均使用指定 engine 的副本，可选 engine 为 “tikv”、“tidb” 和 “tiflash”（其中 “tidb” 表示 TiDB 内部的内存表区，主要用于存储一些 TiDB 系统表，用户不能主动使用），分别有 2 个配置级别：

1. TiDB 实例级别，即 INSTANCE 级别。在 TiDB 的配置文件添加如下配置项：

   ```
   [isolation-read]
   engines = ["tikv", "tidb", "tiflash"]
   ```

   实例级别的默认配置为 ["tikv", "tidb", "tiflash"]。

2. 会话级别，即 SESSION 级别。设置语句：

   ```
   set @@session.tidb_isolation_read_engines = "逗号分隔的 engine list";
   ```

   或者

   ```
   set SESSION tidb_isolation_read_engines = "逗号分隔的 engine list";
   ```

会话级别的默认配置继承自 TiDB 实例级别的配置。

最终的 engine 配置为会话级别配置，即会话级别配置会覆盖实例级别配置。比如实例级别配置了 “tikv”，而会话级别配置了 “tiflash”，则会读取 TiFlash 副本。当 engine 配置为 “tikv, tiflash”，即可以同时读取 TiKV 和 TiFlash 副本，优化器会自动选择。

> 注意：
>
> 由于TiDB Dashboard 等组件需要读取一些存储于 TiDB 内存表区的系统表，因此建议实例级别 engine 配置中始终加入 “tidb” engine。

如果查询中的表没有对应 engine 的副本，比如配置了 engine 为 “tiflash” 而该表没有 TiFlash 副本，则查询会报该表不存在该 engine 副本的错。

14.3.3.3 手工 Hint

手工 Hint 可以在满足 engine 隔离的前提下，强制 TiDB 对于某张或某几张表使用指定的副本，使用方法为：

```
select /*+ read_from_storage(tiflash[table_name]) */ ... from table_name;
```

如果在查询语句中对表设置了别名，在 Hint 语句中必须使用别名才能使 Hint 生效。比如：

```
select /*+ read_from_storage(tiflash[alias_a,alias_b]) */ ... from table_name_1 as alias_a,
    ↪ table_name_2 as alias_b where alias_a.column_1 = alias_b.column_2;
```

其中 `tiflash[]` 是提示优化器读取 TiFlash 副本，亦可以根据需要使用 `tikv[]` 来提示优化器读取 TiKV 副本。更多关于该 Hint 语句的语法可以参考READ_FROM_STORAGE。

如果 Hint 指定的表在指定的引擎上不存在副本，则 Hint 会被忽略，并产生 warning。另外 Hint 必须在满足 engine 隔离的前提下才会生效，如果 Hint 中指定的引擎不在 engine 隔离列表中，Hint 同样会被忽略，并产生 warning。

> 注意：
>
> MySQL 命令行客户端在 5.7.7 版本之前默认清除了 Optimizer Hints。如果需要在这些早期版本的客户端中使用 `Hint` 语法，需要在启动客户端时加上 `--comments` 选项，例如 `mysql -h 127.0.0.1 -P 4000 -uroot --comments`。

14.3.3.4 三种方式之间关系的总结

上述三种读取 TiFlash 副本的方式中，Engine 隔离规定了总的可使用副本 engine 的范围，手工 Hint 可以在该范围内进一步实现语句级别及表级别的细粒度的 engine 指定，最终由 CBO 在指定的 engine 范围内根据代价估算最终选取某个 engine 上的副本。

> **注意：**
>
> TiDB 4.0.3 版本之前，在非只读 SQL 语句中（比如 `INSERT INTO ... SELECT`、`SELECT ... FOR UPDATE`、`UPDATE ...`、`DELETE ...`）读取 TiFlash，行为是未定义。TiDB 4.0.3 以及后续的版本，TiDB 内部会对非只读 SQL 语句忽略 TiFlash 副本以保证数据写入、更新、删除的正确性。对应的，如果使用了智能选择的方式，TiDB 会自动选择非 TiFlash 副本；如果使用了 Engine 隔离的方式指定仅读取 TiFlash 副本，则查询会报错；而如果使用了手工 Hint 的方式，则 Hint 会被忽略。

14.3.4 使用 TiSpark 读取 TiFlash

TiSpark 目前提供类似 TiDB 中 engine 隔离的方式读取 TiFlash，方式是通过配置参数 `spark.tispark.isolation_read_engines`。参数值默认为 `tikv,tiflash`，表示根据 CBO 自动选择从 TiFlash 或从 TiKV 读取数据。如果将该参数值设置成 `tiflash`，表示强制从 TiFlash 读取数据。

> **注意：**
>
> 设为 `tiflash` 时，所有查询的表都会只读取 TiFlash 副本，设为 `tikv` 则只读取 TiKV 副本。设为 `tiflash` 时，要求查询所用到的表都必须已创建了 TiFlash 副本，对于未创建 TiFlash 副本的表的查询会报错。

可以使用以下任意一种方式进行设置：

1. 在 `spark-defaults.conf` 文件中添加：

```
spark.tispark.isolation_read_engines tiflash
```

2. 在启动 Spark shell 或 Thrift server 时，启动命令中添加 `--conf spark.tispark.isolation_read_engines=tiflash`
3. Spark shell 中实时设置：`spark.conf.set("spark.tispark.isolation_read_engines", "tiflash")`
4. Thrift server 通过 beeline 连接后实时设置：`set spark.tispark.isolation_read_engines=tiflash`

14.3.5 使用 MPP 模式

本文档介绍 TiFlash 的 MPP 模式及其使用方法。

TiFlash 支持 MPP 模式的查询执行，即在计算中引入跨节点的数据交换（data shuffle 过程）。TiDB 默认由优化器自动选择是否使用 MPP 模式，你可以通过修改变量 `tidb_allow_mpp` 和 `tidb_enforce_mpp` 的值来更改选择策略。

14.3.5.1 控制是否选择 MPP 模式

变量 tidb_allow_mpp 控制 TiDB 能否选择 MPP 模式执行查询。变量 tidb_enforce_mpp 控制是否忽略优化器代价估算，强制使用 TiFlash 的 MPP 模式执行查询。

这两个变量所有取值对应的结果如下：

	tidb_allow_mpp=off	tidb_allow_mpp=on（默认）
tidb_enforce_mpp=off（默认）	不使用 MPP 模式。	优化器根据代价估算选择。（默认）
tidb_enforce_mpp=on	不使用 MPP 模式。	TiDB 无视代价估算，选择 MPP 模式。

例如，如果你不想使用 MPP 模式，可以通过以下语句来设置：

```
set @@session.tidb_allow_mpp=0;
```

如果想要通过优化器代价估算来智能选择是否使用 MPP（默认情况），可以通过如下语句来设置：

```
set @@session.tidb_allow_mpp=1;
set @@session.tidb_enforce_mpp=0;
```

如果想要 TiDB 忽略优化器的代价估算，强制使用 MPP，可以通过如下语句来设置：

```
set @@session.tidb_allow_mpp=1;
set @@session.tidb_enforce_mpp=1;
```

Session 变量 tidb_enforce_mpp 的初始值等于这台 tidb-server 实例的enforce-mpp 配置项值（默认为 false）。在一个 TiDB 集群中，如果有若干台 tidb-server 实例只执行分析型查询，要确保它们能够选中 MPP 模式，你可以将它们的enforce-mpp 配置值修改为 true.

> 注意：
>
> tidb_enforce_mpp=1 在生效时，TiDB 优化器会忽略代价估算选择 MPP 模式。但如果存在其它不支持 MPP 的因素，例如没有 TiFlash 副本、TiFlash 副本同步未完成、语句中含有 MPP 模式不支持的算子或函数等，那么 TiDB 仍然不会选择 MPP 模式。
>
> 如果由于代价估算之外的原因导致 TiDB 优化器无法选择 MPP，在你使用 EXPLAIN 语句查看执行计划时，会返回警告说明原因，例如：

```
set @@session.tidb_enforce_mpp=1;
create table t(a int);
explain select count(*) from t;
show warnings;
```

```
+---------+------+-----------------------------------------------------------------------------+
    ↪
| Level   | Code | Message
    ↪                                                                            |
```

```
+---------+------+-----------------------------------------------------------------------------+
    ↪
| Warning | 1105 | MPP mode may be blocked because there aren't tiflash replicas of
    ↪  table `t`. |
+---------+------+-----------------------------------------------------------------------------+
    ↪
```

14.3.5.2 MPP 模式的算法支持

MPP 模式目前支持的物理算法有：Broadcast Hash Join、Shuffled Hash Join、Shuffled Hash Aggregation、Union All、TopN 和 Limit。算法的选择由优化器自动判断。通过 EXPLAIN 语句可以查看具体的查询执行计划。如果 EXPLAIN 语句的结果中出现 ExchangeSender 和 ExchangeReceiver 算子，表明 MPP 已生效。

以 TPC-H 测试集中的表结构为例：

```
mysql> explain select count(*) from customer c join nation n on c.c_nationkey=n.n_nationkey;
+--
    ↪ ----------------------------------------+------------+------------------+---------------+---------------
    ↪
| id                                    | estRows    | task              | access object |
    ↪ operator info                                                          |
+--
    ↪ ----------------------------------------+------------+------------------+---------------+---------------
    ↪
| HashAgg_23                            | 1.00       | root              |               |
    ↪ funcs:count(Column#16)->Column#15                                      |
| └─TableReader_25                       | 1.00       | root              |               |
    ↪ data:ExchangeSender_24                                                 |
|   └─ExchangeSender_24                  | 1.00       | batchCop[tiflash] |               |
    ↪ ExchangeType: PassThrough                                              |
|     └─HashAgg_12                       | 1.00       | batchCop[tiflash] |               |
    ↪ funcs:count(1)->Column#16                                              |
|       └─HashJoin_17                    | 3000000.00 | batchCop[tiflash] |               |
    ↪ inner join, equal:[eq(tpch.nation.n_nationkey, tpch.customer.c_nationkey)] |
|         ├─ExchangeReceiver_21(Build)   | 25.00      | batchCop[tiflash] |               |
    ↪                                                                        |
|         │ └─ExchangeSender_20          | 25.00      | batchCop[tiflash] |               |
    ↪ ExchangeType: Broadcast                                                |
|         │   └─TableFullScan_18         | 25.00      | batchCop[tiflash] | table:n       |
    ↪ keep order:false                                                       |
|         └─TableFullScan_22(Probe)      | 3000000.00 | batchCop[tiflash] | table:c       |
    ↪ keep order:false                                                       |
+--
    ↪ ----------------------------------------+------------+------------------+---------------+---------------
```

```
    ↪
9 rows in set (0.00 sec)
```

在执行计划中，出现了 ExchangeReceiver 和 ExchangeSender 算子。该执行计划表示 nation 表读取完毕后，经过 ExchangeSender 算子广播到各个节点中，与 customer 表先后进行 HashJoin 和 HashAgg 操作，再将结果返回至 TiDB 中。

TiFlash 提供了两个全局/会话变量决定是否选择 Broadcast Hash Join，分别为：

- tidb_broadcast_join_threshold_size，单位为 bytes。如果表大小（字节数）小于该值，则选择 Broadcast Hash Join 算法。否则选择 Shuffled Hash Join 算法。
- tidb_broadcast_join_threshold_count，单位为行数。如果 join 的对象为子查询，优化器无法估计子查询结果集大小，在这种情况下通过结果集行数判断。如果子查询的行数估计值小于该变量，则选择 Broadcast Hash Join 算法。否则选择 Shuffled Hash Join 算法。

14.3.5.3 MPP 模式访问分区表

如果希望使用 MPP 模式访问分区表，需要先开启动态裁剪模式。

示例如下：

```
mysql> DROP TABLE if exists test.employees;
Query OK, 0 rows affected, 1 warning (0.00 sec)
mysql> CREATE TABLE test.employees
(id int(11) NOT NULL,
 fname varchar(30) DEFAULT NULL,
 lname varchar(30) DEFAULT NULL,
 hired date NOT NULL DEFAULT '1970-01-01',
 separated date DEFAULT '9999-12-31',
 job_code int DEFAULT NULL,
 store_id int NOT NULL) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin
PARTITION BY RANGE (store_id)
(PARTITION p0 VALUES LESS THAN (6),
 PARTITION p1 VALUES LESS THAN (11),
 PARTITION p2 VALUES LESS THAN (16),
 PARTITION p3 VALUES LESS THAN (MAXVALUE));
Query OK, 0 rows affected (0.10 sec)

mysql> ALTER table test.employees SET tiflash replica 1;
Query OK, 0 rows affected (0.09 sec)
mysql> SET tidb_partition_prune_mode=static;
Query OK, 0 rows affected (0.00 sec)
mysql> explain SELECT count(*) FROM test.employees;
+--
    ↪ --------------------------------+----------+-------------------+-------------------------------+---------
    ↪
```

```
| id                              | estRows  | task              | access object
    ↪                 | operator info                     |
+--
    ↪ -------------------------------+----------+-------------------+---------------------------+--------
    ↪
| HashAgg_18                      | 1.00     | root              |
    ↪                                 | funcs:count(Column#10)->Column#9  |
| └─PartitionUnion_20             | 4.00     | root              |
    ↪                                 |                                   |
|   ├─StreamAgg_35                | 1.00     | root              |
    ↪                                 | funcs:count(Column#12)->Column#10 |
|   │ └─TableReader_36            | 1.00     | root              |
    ↪                                 | data:StreamAgg_26                 |
|   │   └─StreamAgg_26            | 1.00     | batchCop[tiflash] |
    ↪                                 | funcs:count(1)->Column#12         |
|   │     └─TableFullScan_34      | 10000.00 | batchCop[tiflash] | table:employees, partition:
    ↪ p0 | keep order:false, stats:pseudo    |
|   ├─StreamAgg_52                | 1.00     | root              |
    ↪                                 | funcs:count(Column#14)->Column#10 |
|   │ └─TableReader_53            | 1.00     | root              |
    ↪                                 | data:StreamAgg_43                 |
|   │   └─StreamAgg_43            | 1.00     | batchCop[tiflash] |
    ↪                                 | funcs:count(1)->Column#14         |
|   │     └─TableFullScan_51      | 10000.00 | batchCop[tiflash] | table:employees, partition:
    ↪ p1 | keep order:false, stats:pseudo    |
|   ├─StreamAgg_69                | 1.00     | root              |
    ↪                                 | funcs:count(Column#16)->Column#10 |
|   │ └─TableReader_70            | 1.00     | root              |
    ↪                                 | data:StreamAgg_60                 |
|   │   └─StreamAgg_60            | 1.00     | batchCop[tiflash] |
    ↪                                 | funcs:count(1)->Column#16         |
|   │     └─TableFullScan_68      | 10000.00 | batchCop[tiflash] | table:employees, partition:
    ↪ p2 | keep order:false, stats:pseudo    |
|   └─StreamAgg_86                | 1.00     | root              |
    ↪                                 | funcs:count(Column#18)->Column#10 |
|     └─TableReader_87            | 1.00     | root              |
    ↪                                 | data:StreamAgg_77                 |
|       └─StreamAgg_77            | 1.00     | batchCop[tiflash] |
    ↪                                 | funcs:count(1)->Column#18         |
|         └─TableFullScan_85      | 10000.00 | batchCop[tiflash] | table:employees, partition:
    ↪ p3 | keep order:false, stats:pseudo    |
+--
    ↪ -------------------------------+----------+-------------------+---------------------------+--------
    ↪
18 rows in set (0,00 sec)
```

```
mysql> SET tidb_partition_prune_mode=dynamic;
Query OK, 0 rows affected (0.00 sec)
mysql> explain SELECT count(*) FROM test.employees;
+--
    ↪ ---------------------------+----------+-------------+-----------------+-------------------------------
    ↪
| id                           | estRows  | task        | access object   | operator info
    ↪                                      |
+--
    ↪ ---------------------------+----------+-------------+-----------------+-------------------------------
    ↪
| HashAgg_17                   | 1.00     | root        |                 | funcs:count(Column
    ↪ #11)->Column#9                         |
| └─TableReader_19             | 1.00     | root        | partition:all   | data:
    ↪ ExchangeSender_18                       |
|   └─ExchangeSender_18        | 1.00     | mpp[tiflash] |                | ExchangeType:
    ↪ PassThrough                            |
|     └─HashAgg_8              | 1.00     | mpp[tiflash] |                | funcs:count(1)->
    ↪ Column#11                              |
|       └─TableFullScan_16     | 10000.00 | mpp[tiflash] | table:employees | keep order:false,
    ↪ stats:pseudo, PartitionTableScan:true |
+--
    ↪ ---------------------------+----------+-------------+-----------------+-------------------------------
    ↪
5 rows in set (0,00 sec)
```

14.3.6 TiFlash 存算分离架构与 S3 支持

> **警告：**
>
> TiFlash 存算分离架构目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现 bug，请在 GitHub 上提 issue 反馈。

TiFlash 默认使用存算一体的架构进行部署，即 TiFlash 节点既是存储节点，也是计算节点。从 TiDB v7.0.0 开始，TiFlash 支持存算分离架构，并将数据存储在 Amazon S3 或兼容 S3 API 的对象存储中（比如 MinIO）。

14.3.6.1 架构介绍

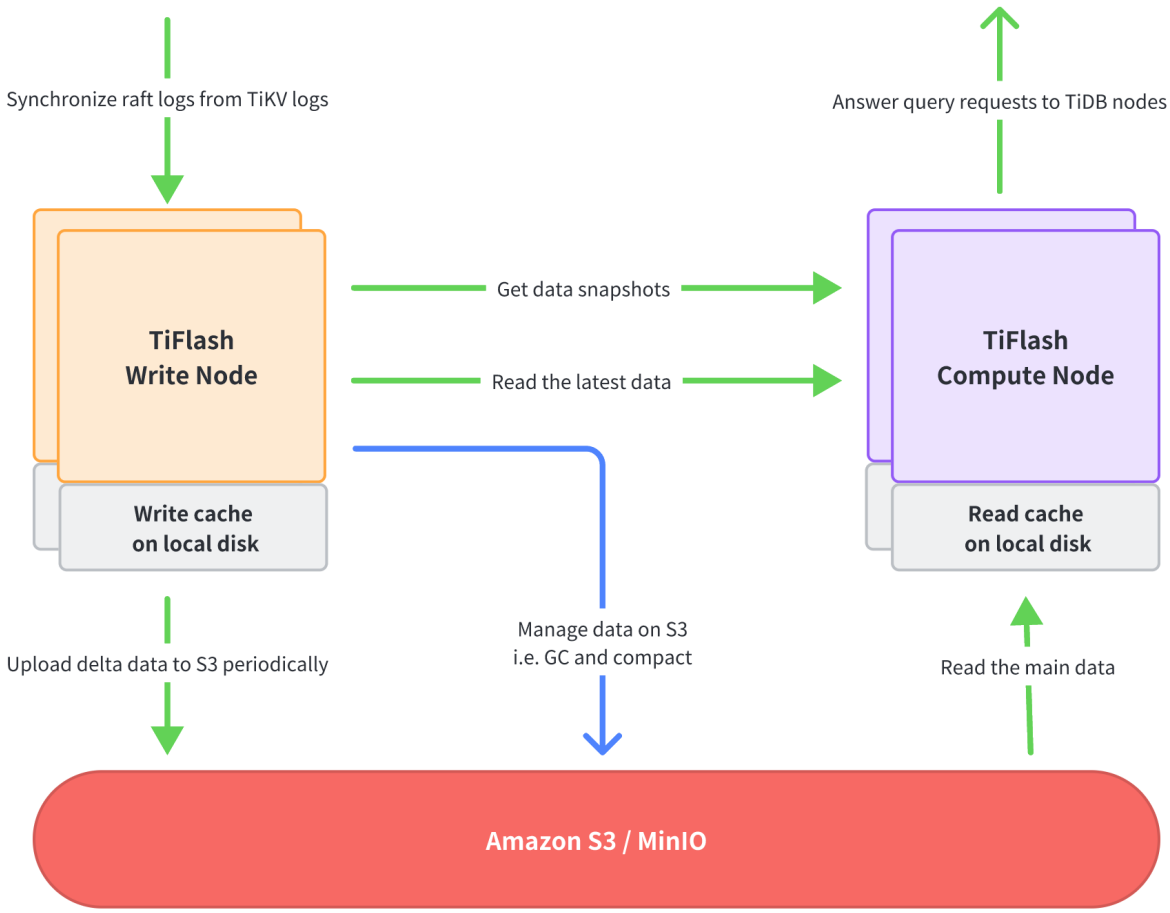


图 322: TiFlash Write and Compute Separation Architect

如图，在存算分离架构中，TiFlash 原有进程的不同部分的功能，被拆分到两种不同的节点中，分别是 Write Node 和 Compute Node。这两种节点可以分别部署，各自扩展，即你可以选择部署任意数量的 Write Node 或者 Compute Node。

- TiFlash Write Node

  负责接收 TiKV 的 Raft logs 数据，将数据转换成列存格式，并每隔一小段时间将这段时间的所有数据更新打包上传到 S3 中。此外，Write Node 也负责管理 S3 上的数据，比如不断整理数据使之具有更好的查询性能，以及删除无用的数据等。

  Write Node 利用本地磁盘（通常是 NVMe SSD）来缓存最新写入的数据，从而避免过多使用内存。

  Write Node 比原来存算一体的 TiFlash 节点有更快的扩容和缩容速度，即增加或者删除 Write Node 后，数据能更快地在 Write Node 之间达到平衡。原理是 Write Node 把所有数据存储到了 S3，运行时只需要在本地存储很少的数据。扩容和缩容本质上是 Region Peer 在节点间的迁移。当某个 Write Node 要将某个 Region Peer 移动到自己之上管理时，它只需要从 Region Peer 所在的 Write Node 上传到 S3 的最新文件中下载少量关于此 Region 的元数据，再从 TiKV 同步最近的 Region 更新，就可以追上 Region Leader 的进度，从而完成 Region Peer 的迁移。

- TiFlash Compute Node

  负责执行从 TiDB 节点发过来的查询请求。它首先访问 Write Node 以获取数据的快照 (data snapshots)，然后分别从 Write Node 读取最新的数据（即尚未上传到 S3 的数据），从 S3 读取剩下的大部分数据。

  Compute Node 利用本地磁盘（通常是 NVMe SSD）来作为数据文件的缓存，从而避免相同的数据反复从远端（Write Node 或者 S3）读取，以提高查询性能。

  Compute Node 是无状态节点，它拥有秒级的扩容和缩容速度。你可以利用这个特性降低成本：

  - 在查询负载较低时，减少 Compute Node 的数量，从而节省成本。在没有查询时，甚至可以停掉所有 Compute Node。
  - 在查询负载变高时，快速增加 Compute Node 的数量，保证查询性能。

14.3.6.2 使用场景

TiFlash 存算分离架构适用于高性价比的数据分析服务的场景。在这个架构下，存储和计算资源可以单独按需扩展。在这些场景将会有较大收益：

- 数据量虽然很大，但是只有少量数据被频繁查询；其他大部分数据属于冷数据，很少被查询。此时经常被查询的数据通常已被缓存在 Compute Node 的本地 SSD 上，可以提供较快查询性能；而其他大部分冷数据则存储在成本较低的 S3 或者其他对象存储上，从而节省存储成本。
- 计算资源需求有明显的波峰和波谷。比如重型的对账查询通常放在晚上执行，此时对计算资源要求较高，可以考虑临时扩展 Compute Node；其他时间可以用较少的 Compute Node 完成查询任务。

14.3.6.3 准备条件

1. 准备一个 S3 的 bucket，用于存储 TiFlash 数据。

   你也可以使用已有的 bucket，但需要为每个 TiDB 集群预留专门的 key 前缀。关于 S3 bucket 的更多信息，请参考 AWS 文档。

   也可以使用兼容 S3 的其他对象存储，比如 MinIO。

   TiFlash 使用的 S3 API 接口列表包括：

   - PutObject
   - GetObject
   - CopyObject
   - DeleteObject
   - ListObjectsV2
   - GetObjectTagging
   - PutBucketLifecycle

2. 给准备好的 S3 bucket 添加一个用于清理已删除数据的生命周期：

```
"ID": "tiflash-clean",
"Expiration": {
    "Days": 1
},
```

```
"Filter": {
    "And": {
        "Tags": [
            {
                "Value": "tiflash_deleted",
                "Key": "true"
            }
        ]
    }
}
```

14.3.6.4 使用方式

默认情况下，TiUP 会将 TiFlash 部署为存算一体架构。如需将 TiFlash 部署为存算分离架构，请参考以下步骤手动进行配置：

1. 确保 TiDB 集群中没有任何 TiFlash 节点。如果有，则需要将所有表的 TiFlash 副本数设置为 0，然后缩容掉所有 TiFlash 节点。比如：

```
SELECT * FROM INFORMATION_SCHEMA.TIFLASH_REPLICA; # 查询所有带有 TiFlash 副本的表
ALTER TABLE table_name SET TIFLASH REPLICA 0;     # 将所有表的 TiFlash 副本数设置为 0
```

```
tiup cluster scale-in mycuster -R tiflash # 缩容掉所有 TiFlash 节点
tiup cluster display mycluster            # 等待所有 TiFlash 节点进入 Tombstone 状态
tiup cluster prune mycluster              # 移除所有处于 Tombstone 状态的 TiFlash 节点
```

2. 准备 TiFlash 的拓扑配置文件，比如 scale-out.topo.yaml，配置内容如下：

```
tiflash_servers:
  # TiFlash 的拓扑配置中存在 storage.s3 配置，说明部署时使用存算分离架构
  # 配置了 flash.disaggregated_mode: tiflash_compute，则节点类型是 Compute Node；
  # 配置了 flash.disaggregated_mode: tiflash_write，则节点类型是 Write Node

  # 172.31.8.1~2 是 TiFlash Write Node
  - host: 172.31.8.1
    config:
      flash.disaggregated_mode: tiflash_write                # 这是一个 Write Node
      storage.s3.endpoint: http://s3.{region}.amazonaws.com # S3 的 endpoint 地址
      storage.s3.bucket: mybucket                            # TiFlash 的所有数据存储在这个
          ↪ bucket 中
      storage.s3.root: /cluster1_data                        # S3 bucket 中存储数据的根目录
      storage.s3.access_key_id: {ACCESS_KEY_ID}              # 访问 S3 的 ACCESS_KEY_ID
      storage.s3.secret_access_key: {SECRET_ACCESS_KEY}      # 访问 S3 的 SECRET_ACCESS_KEY
      storage.main.dir: ["/data1/tiflash/data"]              # Write Node 的本地数据目录，
          ↪ 和存算一体的配置方式相同
  - host: 172.31.8.2
```

```
  config:
    flash.disaggregated_mode: tiflash_write               # 这是一个 Write Node
    storage.s3.endpoint: http://s3.{region}.amazonaws.com # S3 的 endpoint 地址
    storage.s3.bucket: mybucket                           # TiFlash 的所有数据存储在这个
        ↪ bucket 中
    storage.s3.root: /cluster1_data                       # S3 bucket 中存储数据的根目录
    storage.s3.access_key_id: {ACCESS_KEY_ID}             # 访问 S3 的 ACCESS_KEY_ID
    storage.s3.secret_access_key: {SECRET_ACCESS_KEY}     # 访问 S3 的 SECRET_ACCESS_KEY
    storage.main.dir: ["/data1/tiflash/data"]             # Write Node 的本地数据目录，
        ↪ 和存算一体的配置方式相同

# 172.31.9.1~2 是 TiFlash Compute Node
- host: 172.31.9.1
  config:
    flash.disaggregated_mode: tiflash_compute             # 这是一个 Compute Node
    storage.s3.endpoint: http://s3.{region}.amazonaws.com # S3 的 endpoint 地址
    storage.s3.bucket: mybucket                           # TiFlash 的所有数据存储在这个
        ↪ bucket 中
    storage.s3.root: /cluster1_data                       # S3 bucket 中存储数据的根目录
    storage.s3.access_key_id: {ACCESS_KEY_ID}             # 访问 S3 的 ACCESS_KEY_ID
    storage.s3.secret_access_key: {SECRET_ACCESS_KEY}     # 访问 S3 的 SECRET_ACCESS_KEY
    storage.main.dir: ["/data1/tiflash/data"]             # Compute Node 的本地数据目录，
        ↪ 和存算一体的配置方式相同
    storage.remote.cache.dir: /data1/tiflash/cache        # Compute Node
        ↪ 的本地数据缓存目录
    storage.remote.cache.capacity: 858993459200           # 800 GiB
- host: 172.31.9.2
  config:
    flash.disaggregated_mode: tiflash_compute             # 这是一个 Compute Node
    storage.s3.endpoint: http://s3.{region}.amazonaws.com # S3 的 endpoint 地址
    storage.s3.bucket: mybucket                           # TiFlash 的所有数据存储在这个
        ↪ bucket 中
    storage.s3.root: /cluster1_data                       # S3 bucket 中存储数据的根目录
    storage.s3.access_key_id: {ACCESS_KEY_ID}             # 访问 S3 的 ACCESS_KEY_ID
    storage.s3.secret_access_key: {SECRET_ACCESS_KEY}     # 访问 S3 的 SECRET_ACCESS_KEY
    storage.main.dir: ["/data1/tiflash/data"]             # Compute Node 的本地数据目录，
        ↪ 和存算一体的配置方式相同
    storage.remote.cache.dir: /data1/tiflash/cache        # Compute Node
        ↪ 的本地数据缓存目录
    storage.remote.cache.capacity: 858993459200           # 800 GiB
```

- 注意以上 ACCESS_KEY_ID 和 SECRET_ACCESS_KEY 是直接写在配置文件中的。你也可以选择使用环境变量的方式单独配置。如果两种方式都配置了，环境变量的优先级高于配置文件。

  如需通过环境变量配置，请在所有部署了 TiFlash 进程的机器上，切换到启动 TiFlash 进程的用户环境（通常是 tidb），然后修改 ~/.bash_profile，增加这些配置：

```
export S3_ACCESS_KEY_ID={ACCESS_KEY_ID}
export S3_SECRET_ACCESS_KEY={SECRET_ACCESS_KEY}
```

- `storage.s3.endpoint` 支持使用 `http` 模式和 `https` 模式连接 S3，可以直接通过修改 URL 来选择。比如 `https://s3.{region}.amazonaws.com`。

3. 执行扩容 TiFlash 节点，并重新设置 TiFlash 副本数：

```
tiup cluster scale-out mycluster ./scale-out.topo.yaml
```

```
ALTER TABLE table_name SET TIFLASH REPLICA 1;
```

4. 修改 TiDB 配置，用存算分离的方式查询 TiFlash。

    1. 以编辑模式打开 TiDB 配置文件：

        ```
        shell tiup cluster edit-config mycluster
        ```

    2. 在 TiDB 配置文件中添加以下配置项：

        ```
        server_configs:
        tidb:
        disaggregated-tiflash: true   # 使用存算分离的方式查询 TiFlash
        ```

    3. 重启 TiDB:

        ```
        tiup cluster reload mycluster -R tidb
        ```

14.3.6.5 使用限制

- TiFlash 不支持在存算一体架构和存算分离架构之间原地切换。在切换架构前，需要将原有 TiFlash 节点全部删除。
- 从一种架构迁移到另外一种架构后，需要重新同步所有 TiFlash 的数据。
- 同一个 TiDB 集群只允许存在相同架构的 TiFlash 节点，不允许两种架构同时存在。
- 存算分离架构只支持使用 S3 API 的对象存储，存算一体架构只支持本地存储。
- 使用 S3 存储的情况下，TiFlash 节点无法获取非本节点文件的密钥，因此无法启用静态加密功能。

14.3.7 使用 FastScan 功能

本文档介绍通过使用 FastScan 来加速 Online Analytical Processing (OLAP) 场景中查询的方法。

默认情况下，TiFlash 能够保证查询结果精度以及数据一致性。如果使用 FastScan，TiFlash 可以实现更高效的查询性能，但不保证查询结果精度和数据一致性。

某些 OLAP 对查询结果精度可以容忍一定误差。如果对查询性能有更高要求，可以在 session 级别或 global 级别开启 FastScan 功能，你可以通过修改变量 `tiflash_fastscan` 的值来选择是否启用 FastScan 功能。

14.3.7.1 启用和禁用 FastScan

默认情况下，session 和 global 级别的变量 `tiflash_fastscan=OFF`，即没有开启 FastScan 功能。你可以通过以下语句来查看对应的变量信息。

```
show variables like 'tiflash_fastscan';
```

```
+------------------+-------+
| Variable_name    | Value |
+------------------+-------+
| tiflash_fastscan | OFF   |
+------------------+-------+
```

```
show global variables like 'tiflash_fastscan';
```

```
+------------------+-------+
| Variable_name    | Value |
+------------------+-------+
| tiflash_fastscan | OFF   |
+------------------+-------+
```

变量 `tiflash_fastscan` 支持 session 级别和 global 级别的修改。如果需要在当前 session 中启用 FastScan 功能，可以通过以下语句设置:

```
set session tiflash_fastscan=ON;
```

如果对 global 级别的 `tiflash_fastscan` 进行设置，设置后新建的会话中默认 session 和 global 变量 `tiflash_fastscan` 启用新值。设置方式如下：

```
set global tiflash_fastscan=ON;
```

可以用下面语句禁用 FastScan：

```
set session tiflash_fastscan=OFF;
set global tiflash_fastscan=OFF;
```

14.3.7.2 实现机制

TiFlash 存储层的数据主要存放在 Delta 层和 Stable 层。

在默认状态下（即未开启 FastScan 功能），TableScan 算子过程整体包括了以下步骤：

1. Read data：在 Delta 层和 Stable 层分别建立数据流，进行各自数据的读取。
2. Sort Merge：将步骤 1 中建立的数据流进行合并，并且将数据按照 (handle, version) 顺序排列返回。
3. Range Filter：根据读取范围限制，对步骤 2 中的数据进行过滤筛选并返回。
4. MVCC + Column Filter：对步骤 3 中的数据进行 MVCC 过滤，同时过滤掉不需要的列并返回。

FastScan 通过损失一定的数据一致性来获取更快的查询性能。FastScan 中的 TableScan 流程省略了上述过程中的第 2 步和第 4 步中 MVCC 的部分，从而提高查询性能。

14.3.8 TiFlash 支持的计算下推

本文档介绍 TiFlash 支持的计算下推。

14.3.8.1 支持下推的算子

TiFlash 支持部分算子的下推，支持的算子如下：

- TableScan：该算子从表中读取数据
- Selection：该算子对数据进行过滤
- HashAgg：该算子基于Hash Aggregation 算法对数据进行聚合运算
- StreamAgg：该算子基于Stream Aggregation 算法对数据进行聚合运算。StreamAgg 仅支持不带 GROUP BY 条件的列。
- TopN：该算子对数据求 TopN 运算
- Limit：该算子对数据进行 limit 运算
- Project：该算子对数据进行投影运算
- HashJoin：该算子基于Hash Join 算法对数据进行连接运算：
  - 只有在MPP 模式下才能被下推
  - 支持的 Join 类型包括 Inner Join、 Left Join、 Semi Join、 Anti Semi Join、 Left Semi Join、 Anti Left Semi Join
  - 对于上述类型，既支持带等值条件的连接，也支持不带等值条件的连接（即 Cartesian Join 或者 Null-aware Semi Join）；在计算 Cartesian Join 或者 Null-aware Semi Join 时，只会使用 Broadcast 算法，而不会使用 Shuffle Hash Join 算法
- Window：当前支持下推的窗口函数包括 ROW_NUMBER()、 RANK()、 DENSE_RANK()、 LEAD() 和 LAG()

在 TiDB 中，算子之间会呈现树型组织结构。一个算子能下推到 TiFlash 的前提条件，是该算子的所有子算子都能下推到 TiFlash。因为大部分算子都包含有表达式计算，当且仅当一个算子所包含的所有表达式均支持下推到 TiFlash 时，该算子才有可能下推给 TiFlash。

14.3.8.2 支持下推的表达式

表达式类型	运算
数学函数	+, -, /, *, %, >=, <=, =, !=, <, >, ROUND(), ABS(), FLOOR(int), CEIL(int), CEILING(int), SQRT(), LOG(), LOG2(), LOG10(), LN(), EXP(), POW(), SIGN(), RADIANS(), DEGREES(), CONV(), CRC32(), GREATEST(int/real), LEAST(int/real)
逻辑函数和算子	AND, OR, NOT, CASE WHEN, IF(), IFNULL(), ISNULL(), IN, LIKE, ILIKE, COALESCE, IS
位运算	& (bitand), | (bitor), ~ (bitneg), ^ (bitxor)
字符串函数	SUBSTR(), CHAR_LENGTH(), REPLACE(), CONCAT(), CONCAT_WS(), LEFT(), RIGHT(), ASCII(), LENGTH(), TRIM(), LTRIM(), RTRIM(), POSITION(), FORMAT(), LOWER(), UCASE(), UPPER(), SUBSTRING_INDEX(), LPAD(), RPAD(), STRCMP()

表达式类型	运算
正则函数和算子	REGEXP, REGEXP_LIKE(), REGEXP_INSTR(), REGEXP_SUBSTR(), REGEXP_REPLACE()
日期函数	DATE_FORMAT(), TIMESTAMPDIFF(), FROM_UNIXTIME(), UNIX_TIMESTAMP(int), UNIX_TIMESTAMP(decimal), STR_TO_DATE(date), STR_TO_DATE(datetime), DATEDIFF(), YEAR(), MONTH(), DAY(), EXTRACT(datetime), DATE(), HOUR(), MICROSECOND(), MINUTE(), SECOND(), SYSDATE(), DATE_ADD/ADDDATE(datetime, int), DATE_ADD/ADDDATE(string, int/real), DATE_SUB/SUBDATE(datetime, int), DATE_SUB/SUBDATE(string, int/real), QUARTER(), DAYNAME(), DAYOFMONTH(), DAYOFWEEK(), DAYOFYEAR(), LAST_DAY(), MONTHNAME(), TO_SECONDS(), TO_DAYS(), FROM_DAYS(), WEEKOFYEAR()
JSON 函数	JSON_LENGTH(), ->, ->>, JSON_EXTRACT()
转换函数	CAST(int AS DOUBLE), CAST(int AS DECIMAL), CAST(int AS STRING), CAST(int AS TIME), CAST(double AS INT), CAST(double AS DECIMAL), CAST(double AS STRING), CAST(double AS TIME), CAST(string AS INT), CAST(string AS DOUBLE), ↪ CAST(string AS DECIMAL), CAST(string AS TIME), CAST(decimal AS INT), CAST(decimal AS STRING), CAST(decimal AS TIME), CAST(time AS INT), CAST(time AS DECIMAL), CAST(time AS STRING), CAST(time AS REAL)
聚合函数	MIN(), MAX(), SUM(), COUNT(), AVG(), APPROX_COUNT_DISTINCT(), GROUP_CONCAT()
其他函数	INET_NTOA(), INET_ATON(), INET6_NTOA(), INET6_ATON()

14.3.8.3 下推限制

- 所有包含 Bit、 Set 和 Geometry 类型的表达式均不能下推到 TiFlash
- DATE_ADD()、 DATE_SUB()、 ADDDATE() 以及 SUBDATE() 中的 interval 类型只支持如下几种，如使用了其他类型的 interval，TiFlash 会在运行时报错。
  - DAY
  - WEEK
  - MONTH
  - YEAR

- HOUR
- MINUTE
- SECOND

如查询遇到不支持的下推计算，则需要依赖 TiDB 完成剩余计算，可能会很大程度影响 TiFlash 加速效果。对于暂不支持的算子/表达式，将会在后续版本中陆续支持。

类似 `MAX()` 这样的函数在聚合算子中支持下推，但是在窗口函数算子中还不支持下推。

14.3.8.4 示例

以下通过一些例子对下推算子和表达式到 TiFlash 进行说明。

14.3.8.4.1 示例 1：下推算子到 TiFlash 存储

```
CREATE TABLE t(id INT PRIMARY KEY, a INT);
ALTER TABLE t SET TIFLASH REPLICA 1;

EXPLAIN SELECT * FROM t LIMIT 3;

+--
    ↪ ----------------------------+---------+--------------+---------------+--------------------------------+
    ↪
| id                            | estRows | task         | access object | operator info
    ↪                   |
+--
    ↪ ----------------------------+---------+--------------+---------------+--------------------------------+
    ↪
| Limit_9                       | 3.00    | root         |               | offset:0, count:3
    ↪               |
| └─TableReader_17              | 3.00    | root         |               | data:ExchangeSender_16
    ↪           |
|   └─ExchangeSender_16         | 3.00    | mpp[tiflash] |               | ExchangeType:
    ↪ PassThrough       |
|     └─Limit_15                | 3.00    | mpp[tiflash] |               | offset:0, count:3
    ↪               |
|       └─TableFullScan_14      | 3.00    | mpp[tiflash] | table:t       | keep order:false,
    ↪ stats:pseudo |
+--
    ↪ ----------------------------+---------+--------------+---------------+--------------------------------+
    ↪
5 rows in set (0.18 sec)
```

在该查询中，算子 Limit 被下推到 TiFlash 对数据进行过滤，减少了网络传输数据量，进而减少网络传输开销。具体可查看以上示例中 `Limit_15` 算子的 `task` 列，其值为 `mpp[tiflash]`，表示该算子被下推到 TiFlash。

14.3.8.4.2 示例 2：下推表达式到 TiFlash 存储

```
CREATE TABLE t(id INT PRIMARY KEY, a INT);
ALTER TABLE t SET TIFLASH REPLICA 1;
INSERT INTO t(id,a) VALUES (1,2),(2,4),(11,2),(12,4),(13,4),(14,7);

EXPLAIN SELECT MAX(id + a) FROM t GROUP BY a;

+--
    ↪ ---------------------------------+---------+--------------+--------------+-----------------------------
    ↪
| id                                | estRows | task         | access object | operator info
    ↪                                                                  |
+--
    ↪ ---------------------------------+---------+--------------+--------------+-----------------------------
    ↪
| TableReader_45                      | 4.80    | root         |               | data:
    ↪ ExchangeSender_44                                                        |
| └─ExchangeSender_44                 | 4.80    | mpp[tiflash] |               | ExchangeType:
    ↪ PassThrough                                                    |
|   └─Projection_39                   | 4.80    | mpp[tiflash] |               | Column#3
    ↪                                                                |
|     └─HashAgg_37                    | 4.80    | mpp[tiflash] |               | group by:Column
    ↪ #9, funcs:max(Column#8)->Column#3                              |
|       └─Projection_46               | 6.00    | mpp[tiflash] |               | plus(test.t.id,
    ↪ test.t.a)->Column#8, test.t.a                                  |
|         └─ExchangeReceiver_23       | 6.00    | mpp[tiflash] |               |
    ↪                                                                  |
|           └─ExchangeSender_22       | 6.00    | mpp[tiflash] |               | ExchangeType:
    ↪ HashPartition, Hash Cols: [name: test.t.a, collate: binary] |
|             └─TableFullScan_21      | 6.00    | mpp[tiflash] | table:t       | keep order:false
    ↪ , stats:pseudo                                                 |
+--
    ↪ ---------------------------------+---------+--------------+--------------+-----------------------------
    ↪
8 rows in set (0.18 sec)
```

在该查询中，表达式 `id + a` 被下推到 TiFlash，从而能提前进行计算，减少网络传输数据量，进而减少网络传输开销，提升整体计算性能。具体可查看以上示例中 `operator` 列为 `plus(test.t.id, test.t.a)` 的行的 `task` 列，其值为 `mpp[tiflash]`，表示该表达式被下推到 TiFlash。

14.3.8.4.3 示例 3：下推限制

```
CREATE TABLE t(id INT PRIMARY KEY, a INT);
ALTER TABLE t SET TIFLASH REPLICA 1;
INSERT INTO t(id,a) VALUES (1,2),(2,4),(11,2),(12,4),(13,4),(14,7);
```

```
EXPLAIN SELECT id FROM t WHERE TIME(now()+ a) < '12:00:00';

+--
    ↪ -------------------------+---------+-------------+--------------+-------------------------------
    ↪
| id                          | estRows | task        | access object | operator info
    ↪                                                                                              |
+--
    ↪ -------------------------+---------+-------------+--------------+-------------------------------
    ↪
| Projection_4                | 4.80    | root        |               | test.t.id
    ↪                                                                                                 |
| └─Selection_6               | 4.80    | root        |               | lt(cast(time(cast(plus
    ↪ (20230110083056, test.t.a), var_string(20))), var_string(10)), "12:00:00") |
|   └─TableReader_11          | 6.00    | root        |               | data:ExchangeSender_10
    ↪                                                                                         |
|     └─ExchangeSender_10     | 6.00    | mpp[tiflash] |               | ExchangeType:
    ↪ PassThrough                                                                                  |
|       └─TableFullScan_9     | 6.00    | mpp[tiflash] | table:t       | keep order:false, stats
    ↪ :pseudo                                                                                 |
+--
    ↪ -------------------------+---------+-------------+--------------+-------------------------------
    ↪
5 rows in set, 3 warnings (0.20 sec)
```

分析执行计划可以发现，该查询在执行时只在 TiFlash 中进行了 TableFullScan，其他的函数计算和过滤均在 root 进行，并未下推至 TiFlash。

执行以下命令，可以查找不能下推的算子和表达式。

```
SHOW WARNINGS;

+--
    ↪ -------+------+-----------------------------------------------------------------------------------
    ↪
| Level   | Code | Message
    ↪
    ↪ |
+--
    ↪ -------+------+-----------------------------------------------------------------------------------
    ↪
| Warning | 1105 | Scalar function 'time'(signature: Time, return type: time) is not supported to
    ↪  push down to storage layer now.                    |
| Warning | 1105 | Scalar function 'cast'(signature: CastDurationAsString, return type:
    ↪ var_string(10)) is not supported to push down to tiflash now. |
```

```
| Warning | 1105 | Scalar function 'cast'(signature: CastDurationAsString, return type:
    ↪ var_string(10)) is not supported to push down to tiflash now. |
+--
    ↪ -------+------+-----------------------------------------------------------------------------------------
    ↪
3 rows in set (0.18 sec)
```

可以看出，该查询的表达式无法完全下推至 TiFlash，因为 Time 函数和 Cast 函数无法下推至 TiFlash。

14.3.8.4.4 示例 4：窗口函数

```
CREATE TABLE t(id INT PRIMARY KEY, c1 VARCHAR(100));
ALTER TABLE t SET TIFLASH REPLICA 1;
INSERT INTO t VALUES(1,"foo"),(2,"bar"),(3,"bar foo"),(10,"foo"),(20,"bar"),(30,"bar foo");

EXPLAIN SELECT id, ROW_NUMBER() OVER (PARTITION BY id > 10) FROM t;
+--
    ↪ -------------------------------+----------+--------------+---------------+-----------------------------
    ↪
| id                               | estRows  | task         | access object | operator info
    ↪
    ↪ |
+--
    ↪ -------------------------------+----------+--------------+---------------+-----------------------------
    ↪
| TableReader_30                   | 10000.00 | root         |               | MppVersion: 1,
    ↪ data:ExchangeSender_29
    ↪                                                                                 |
| └─ExchangeSender_29              | 10000.00 | mpp[tiflash] |               | ExchangeType:
    ↪ PassThrough
    ↪                                                                                      |
|   └─Projection_7                 | 10000.00 | mpp[tiflash] |               | test.t.id, Column
    ↪ #5, stream_count: 4
    ↪                                                                                        |
|     └─Window_28                  | 10000.00 | mpp[tiflash] |               | row_number()->
    ↪ Column#5 over(partition by Column#4 rows between current row and current row),
    ↪ stream_count: 4  |
|       └─Sort_14                  | 10000.00 | mpp[tiflash] |               | Column#4,
    ↪ stream_count: 4
    ↪                                                                                      |
|         └─ExchangeReceiver_13    | 10000.00 | mpp[tiflash] |               | stream_count: 4
    ↪
    ↪ |
|           └─ExchangeSender_12    | 10000.00 | mpp[tiflash] |               | ExchangeType:
    ↪ HashPartition, Compression: FAST, Hash Cols: [name: Column#4, collate: binary],
    ↪ stream_count: 4 |
```

```
|              └─Projection_10      | 10000.00 | mpp[tiflash] |                | test.t.id, gt(
    ↪ test.t.id, 10)->Column#4
    ↪                                                                          |
|                └─TableFullScan_11 | 10000.00 | mpp[tiflash] | table:t        | keep order:false,
    ↪  stats:pseudo
    ↪                                                                                  |
+--
    ↪ -------------------------------+----------+--------------+---------------+-------------------------------
    ↪
9 rows in set (0.0073 sec)
```

可以看到，Window 操作在 task 列中有一个 mpp[tiflash] 的值，表示 ROW_NUMBER()OVER (PARTITION BY id > ↪ 10) 操作能够被下推至 TiFlash。

```
CREATE TABLE t(id INT PRIMARY KEY, c1 VARCHAR(100));
ALTER TABLE t SET TIFLASH REPLICA 1;
INSERT INTO t VALUES(1,"foo"),(2,"bar"),(3,"bar foo"),(10,"foo"),(20,"bar"),(30,"bar foo");

EXPLAIN SELECT id, MAX(id) OVER (PARTITION BY id > 10) FROM t;
+--
    ↪ --------------------------+----------+-----------+---------------+-------------------------------------
    ↪
| id                          | estRows  | task      | access object | operator info
    ↪                                                 |
+--
    ↪ --------------------------+----------+-----------+---------------+-------------------------------------
    ↪
| Projection_6                | 10000.00 | root      |               | test.t1.id, Column#5
    ↪                                          |
| └─Shuffle_14                | 10000.00 | root      |               | execution info:
    ↪ concurrency:5, data sources:[Projection_8] |
|   └─Window_7                | 10000.00 | root      |               | max(test.t1.id)->Column#5
    ↪  over(partition by Column#4)      |
|     └─Sort_13               | 10000.00 | root      |               | Column#4
    ↪                                                    |
|       └─Projection_8        | 10000.00 | root      |               | test.t1.id, gt(test.t1.id
    ↪ , 10)->Column#4                   |
|         └─TableReader_10    | 10000.00 | root      |               | data:TableFullScan_9
    ↪                                          |
|           └─TableFullScan_9 | 10000.00 | cop[tikv] | table:t1      | keep order:false, stats:
    ↪ pseudo                              |
+--
    ↪ --------------------------+----------+-----------+---------------+-------------------------------------
    ↪
7 rows in set (0.0010 sec)
```

可以看到，Window 操作在 task 列中有一个 root 的值，表示 MAX(id)OVER (PARTITION BY id > 10) 操作不能被下推至 TiFlash。这是因为，MAX() 只支持作为聚合函数下推，而不支持作为窗口函数下推。

14.3.9 TiFlash 查询结果物化

> **警告：**
>
> 该功能目前是实验性功能，请注意使用场景限制。该功能会在未事先通知的情况下发生变化或删除。语法和实现可能会在 GA 前发生变化。如果发现 bug，请在 GitHub 上提交 issue 反馈。

本文介绍如何在同一个事务 (INSERT INTO SELECT) 中实现将 TiFlash 查询结果保存至某一指定的 TiDB 表中。

从 v6.5.0 起，TiDB 支持将 TiFlash 查询结果保存到数据表中，即物化了 TiFlash 的查询结果。执行 INSERT INTO SELECT 语句时，如果 TiDB 将 SELECT 子查询下推到了 TiFlash，TiFlash 的查询结果可以保存到 INSERT INTO 指定的 TiDB 表中。v6.5.0 之前的 TiDB 版本不允许此类行为，即通过 TiFlash 执行的查询必须是只读的，你需要从应用程序层面接收 TiFlash 返回的结果，然后另行在其它事务或处理中保存结果。

> **注意：**
>
> - 默认情况下 (tidb_allow_mpp = ON)，TiDB 优化器将依据查询代价智能选择下推查询到 TiKV 或 TiFlash。如需强制使用 TiFlash 查询，你可以设置系统变量tidb_enforce_mpp 为 ON。
> - 在实验特性阶段，该功能默认关闭。要开启此功能，请设置系统变量tidb_enable_tiflash_read_for_write_stmt 为 ON。

INSERT INTO SELECT 语法如下：

```
INSERT [LOW_PRIORITY | HIGH_PRIORITY] [IGNORE]
    [INTO] tbl_name
    [PARTITION (partition_name [, partition_name] ...)]
    [(col_name [, col_name] ...)]
    SELECT ...
    [ON DUPLICATE KEY UPDATE assignment_list]value:
    {expr | DEFAULT}

assignment:
    col_name = valueassignment_list:
    assignment [, assignment] ...
```

例如，通过以下 INSERT INTO SELECT 语句，你可以将 SELECT 子句中表 t1 的查询结果保存到表 t2 中：

```
INSERT INTO t2 (name, country)
SELECT app_name, country FROM t1;
```

14.3.9.1 典型和推荐的使用场景

- 高效的 BI 解决方案

  很多报表类应用有较重的分析查询，如果有很多用户同时打开和刷新报表，则会产生较多的查询请求。一个有效的解决方案是使用本功能在 TiDB 表中保存报表查询的结果，报表刷新时再从结果表中抽取数据，则可以避免多次重复的分析计算。同理，在保存历史分析记录的基础上，可以进一步优化长时间历史数据分析的计算量。例如，某报表 A 用于分析每日的销售利润，使用本功能你可以将报表 A 中每日的分析结果保存到某结果表 T 中。那么，在生成报表 B 分析过去一个月的销售利润时，可以直接使用表 T 中的每日分析结果数据，不仅大幅降低计算量也提升了查询响应速度，减轻系统负载。

- 使用 TiFlash 服务在线应用

  TiFlash 支持的并发请求数量视数据量和查询复杂度不同，但一般不会超过 100 QPS。你可以使用本功能保存 TiFlash 的查询结果，然后通过查询结果表来支持在线的高并发请求。后台的结果表数据更新可以以较低的频率进行，例如以 0.5 秒间隔更新结果表数据也远低于 TiFlash 的并发上限，同时仍然较好地保证了数据新鲜度。

14.3.9.2 执行过程

- 在 `INSERT INTO SELECT` 语句的执行过程中，TiFlash 首先将 `SELECT` 子句的查询结果返回到集群中某单一 TiDB server 节点，然后再写入目标表（可以有 TiFlash 副本）。
- `INSERT INTO SELECT` 语句的执行保证 ACID 特性。

14.3.9.3 限制

- TiDB 对 `INSERT INTO SELECT` 语句的内存限制可以通过系统变量`tidb_mem_quota_query` 调整。从 v6.5.0 版本开始，不推荐使用`txn-total-size-limit` 来控制事务内存大小，详见该配置项文档。

  更多信息，请参考TiDB 内存控制。

- TiDB 对 `INSERT INTO SELECT` 语句的并发没有硬性限制，但是推荐考虑以下用法：

  - 当“写事务”较大时，例如接近 1 GiB，建议控制并发不超过 10。
  - 当“写事务”较小时，例如小于 100 MiB，建议控制并发不超过 30。
  - 请基于测试和具体情况做出合理选择。

14.3.9.4 示例

数据定义：

```
CREATE TABLE detail_data (
    ts DATETIME,                -- 费用产生时间
    customer_id VARCHAR(20),    -- 客户 ID
    detail_fee DECIMAL(20,2));  -- 费用数额


CREATE TABLE daily_data (
```

```
    rec_date DATE,              -- 汇总数据的日期
    customer_id VARCHAR(20),    -- 客户 ID
    daily_fee DECIMAL(20,2));   -- 单日汇总费用

ALTER TABLE detail_data SET TIFLASH REPLICA 1;
ALTER TABLE daily_data SET TIFLASH REPLICA 1;

-- ... (detail_data 表不断增加数据)
INSERT INTO detail_data(ts,customer_id,detail_fee) VALUES
('2023-1-1 12:2:3', 'cus001', 200.86),
('2023-1-2 12:2:3', 'cus002', 100.86),
('2023-1-3 12:2:3', 'cus002', 2200.86),
('2023-1-4 12:2:3', 'cus003', 2020.86),
('2023-1-5 12:2:3', 'cus003', 1200.86),
('2023-1-6 12:2:3', 'cus002', 20.86);
```

每日分析数据保存：

```
SET @@tidb_enable_tiflash_read_for_write_stmt=ON;

INSERT INTO daily_data (rec_date, customer_id, daily_fee)
SELECT DATE(ts), customer_id, sum(detail_fee) FROM detail_data WHERE DATE(ts) = CURRENT_DATE()
    ↪ GROUP BY DATE(ts), customer_id;
```

基于日分析数据的月数据分析：

```
SELECT MONTH(rec_date), customer_id, sum(daily_fee) FROM daily_data GROUP BY MONTH(rec_date),
    ↪ customer_id;
```

将每日分析结果数据物化，保存到日数据结果表中。使用日数据结果表加速月数据分析，从而提升月数据分析效率。

14.3.10 延迟物化

> 警告：
>
> 该功能目前是实验性功能，其形式和使用方法可能会在未来版本中发生变化。

本文档介绍通过使用 TiFlash 延迟物化的方式来加速 Online Analytical Processing (OLAP) 场景的查询。

默认情况下，当收到一个查询请求时，TiFlash 会先读取该查询所需列的全部数据，然后再根据查询条件对数据进行过滤、聚合等计算任务。延迟物化是一种优化方式，它支持下推部分过滤条件到 TableScan 算子，即先扫描过滤条件相关的列数据，过滤得到符合条件的行后，再扫描这些行的其他列数据，继续后续计算，从而减少 IO 扫描和数据处理的计算量。

如果希望提升 OLAP 场景部分查询的性能，可以在 session 级别或 global 级别开启 TiFlash 延迟物化功能。你可以通过修改变量tidb_opt_enable_late_materialization 的值来选择是否启用 TiFlash 延迟物化功能。

启用 TiFlash 延迟物化功能后，TiDB 优化器会根据统计信息和查询的过滤条件，决定哪些过滤条件会被下推。优化器会优先考虑下推过滤率高的过滤条件，详细算法可以参考 RFC 文档。

例如：

```
EXPLAIN SELECT a, b, c FROM t1 WHERE a < 1;
```

```
+-------------------------+----------+--------------+---------------+-----------------------------------------------
    ↪
| id                      | estRows  | task         | access object | operator info
    ↪                                        |
+-------------------------+----------+--------------+---------------+-----------------------------------------------
    ↪
| TableReader_12          | 12288.00 | root         |               | MppVersion: 1, data:
    ↪ ExchangeSender_11                 |
| └─ExchangeSender_11     | 12288.00 | mpp[tiflash] |               | ExchangeType: PassThrough
    ↪                               |
|   └─TableFullScan_9     | 12288.00 | mpp[tiflash] | table:t1      | pushed down filter:lt(test
    ↪ .t1.a, 1), keep order:false |
+-------------------------+----------+--------------+---------------+-----------------------------------------------
    ↪
```

在此示例中，过滤条件 a < 1 被下推到了 TableScan 算子，TiFlash 会先读取列 a 的全部数据，然后根据过滤条件 a < 1 进行过滤，得到符合条件的行后，再读取这些行的列 b 和 c 的数据。

14.3.10.1 启用和禁用 TiFlash 延迟物化

默认情况下，session 和 global 级别的变量 tidb_opt_enable_late_materialization=OFF，即未开启 TiFlash 延迟物化功能。你可以通过以下语句来查看对应的变量信息。

```
SHOW VARIABLES LIKE 'tidb_opt_enable_late_materialization';
```

```
+---------------------------------------+-------+
| Variable_name                         | Value |
+---------------------------------------+-------+
| tidb_opt_enable_late_materialization | OFF   |
+---------------------------------------+-------+
```

```
SHOW GLOBAL VARIABLES LIKE 'tidb_opt_enable_late_materialization';
```

```
+---------------------------------------+-------+
| Variable_name                         | Value |
+---------------------------------------+-------+
| tidb_opt_enable_late_materialization | OFF   |
+---------------------------------------+-------+
```

变量 tidb_opt_enable_late_materialization 支持 session 级别和 global 级别的修改。

- 如果需要在当前 session 中启用 TiFlash 延迟物化功能，可以通过以下语句设置:

```
SET SESSION tidb_opt_enable_late_materialization=ON;
```

- 如果需要在 global 级别启用 TiFlash 延迟物化功能，可以通过以下语句设置：

```
SET GLOBAL tidb_opt_enable_late_materialization=ON;
```

设置后，新建的会话中 session 和 global 级别 tidb_opt_enable_late_materialization 都将默认启用新值。

如需禁用 TiFlash 延迟物化功能，可以通过以下语句设置：

```
SET SESSION tidb_opt_enable_late_materialization=OFF;
```

```
SET GLOBAL tidb_opt_enable_late_materialization=OFF;
```

14.3.10.2 实现机制

当有过滤条件下推到 TableScan 算子时，TableScan 算子的执行过程主要包括了以下步骤：

1. 读取 <handle, del_mark, version> 三列，执行多版本并发控制 (MVCC) 过滤，生成 MVCC Bitmap。
2. 读取过滤条件相关的列，过滤满足条件的行，生成 Filter Bitmap。
3. 将 MVCC Bitmap 和 Filter Bitmap 进行与操作 (AND)，生成 Final Bitmap。
4. 根据 Final Bitmap 读取剩余列的对应的行。
5. 合并第 2 步和第 4 步中读取的数据，返回结果。

14.3.11 TiFlash 数据落盘

本文介绍 TiFlash 计算过程中的数据落盘功能。

从 v7.0.0 起，TiFlash 支持在计算过程中将中间数据落盘以缓解内存压力。目前支持落盘的算子有：

- 带有等值关联条件的 Hash Join 算子
- 带有 GROUP BY key 的 Hash Aggregation 算子
- TopN 算子以及窗口函数中的 Sort 算子

14.3.11.1 TiFlash 数据落盘的触发机制

TiDB 提供了以下系统变量，来控制各算子数据落盘的阈值。当算子使用的内存超过阈值之后，TiFlash 会触发对应算子的落盘。

- tidb_max_bytes_before_tiflash_external_group_by
- tidb_max_bytes_before_tiflash_external_join
- tidb_max_bytes_before_tiflash_external_sort

14.3.11.2 示例

本示例构造一个占用大量内存的 SQL 语句来对 Hash Aggregation 的落盘功能进行演示：

1. 准备数据。创建一个包含 2 个节点的 TiFlash 集群，并导入 TPCH-100 的数据。
2. 执行以下语句。该语句不限制带 GROUP BY 的 Hash Aggregation 算子的内存使用量。

```
SET tidb_max_bytes_before_tiflash_external_group_by = 0;
SELECT
  l_orderkey,
  MAX(L_COMMENT),
  MAX(L_SHIPMODE),
  MAX(L_SHIPINSTRUCT),
  MAX(L_SHIPDATE),
  MAX(L_EXTENDEDPRICE)
FROM lineitem
GROUP BY l_orderkey
HAVING SUM(l_quantity) > 314;
```

3. 从 TiFlash 的日志中可以看到，该查询在单个 TiFlash 节点上需要消耗 29.55 GiB 内存：

```
[DEBUG] [MemoryTracker.cpp:69] ["Peak memory usage (total): 29.55 GiB."] [source=
    ↪ MemoryTracker] [thread_id=468]
```

4. 执行以下语句。该语句限制了带 GROUP BY 的 Hash Aggregation 算子的内存使用量，不超过 10737418240 (10 GiB)。

```
SET tidb_max_bytes_before_tiflash_external_group_by = 10737418240;
SELECT
  l_orderkey,
  MAX(L_COMMENT),
  MAX(L_SHIPMODE),
  MAX(L_SHIPINSTRUCT),
  MAX(L_SHIPDATE),
  MAX(L_EXTENDEDPRICE)
FROM lineitem
GROUP BY l_orderkey
HAVING SUM(l_quantity) > 314;
```

5. 从 TiFlash 的日志中可以看出，通过配置 `tidb_max_bytes_before_tiflash_external_group_by`，TiFlash 触发了中间结果落盘，显著减小了查询所需的内存。

```
[DEBUG] [MemoryTracker.cpp:69] ["Peak memory usage (total): 12.80 GiB."] [source=
    ↪ MemoryTracker] [thread_id=110]
```

14.3.11.3 注意

- 当 Hash Aggregation 算子不带 GROUP BY key 时，不支持落盘。即使该 Hash Aggregation 中含有 DISTINCT 聚合函数，也不能触发落盘。
- 目前阈值是针对单个算子来计算的。如果一个查询里有两个 Hash Aggregation 算子，并且阈值设置为 10 GiB，那么两个 Hash Aggregation 算子仅会在各自占用的内存超过 10 GiB 时才会触发数据落盘。
- 目前 Hash Aggregation 与 TopN/Sort 在 restore 阶段采用的是 merge aggregation 和 merge sort 的算法，所以这两个算子只会触发一次数据落盘。如果需要的内存特别大以至于在 restore 阶段内存使用仍然超阈值，不会再次触发落盘。
- 目前 Hash Join 采用基于分区的落盘策略，在 restore 阶段如果内存使用仍然超过阈值，会继续触发落盘。但是为了控制落盘的规模，落盘的轮数限制为三轮。如果第三轮落盘之后的 restore 阶段内存使用仍然超过阈值，则不会触发新的落盘。

14.3.12 TiFlash 数据校验

本文档介绍 TiFlash 的数据校验机制以及相关的工具。

14.3.12.1 使用场景

数据损坏通常意味着严重的硬件故障。在这种情形下，即使尝试自主修复，也会使得数据的可靠性下降。TiFlash 默认对数据文件进行基础的校验，使用固定的 City128 算法。一旦发现数据校验不符的情况，TiFlash 将立刻报错退出，避免因错误数据造成次生灾害。此时，您需要手动检查干预并重新同步数据，才可以恢复节点的使用。

自 v5.4.0 起，TiFlash 完善了数据校验功能，默认使用 XXH3 算法，并允许用户调整校验帧大小和校验算法。

14.3.12.2 校验机制简介

TiFlash 的数据校验功能基于 DTFile（即 DeltaTree File）提供。DTFile 是 TiFlash 落盘数据的存储文件，共有三版格式：

版本	状态	校验机制	备注
V1	已废弃	在数据文件中内嵌哈希值	
V2	v6.0.0 之前的默认格式	在数据文件中内嵌哈希值	在 V1 的基础上增加了列数据的统计信息
V3	v6.0.0 及之后的默认格式	包含元数据，标记数据校验，支持多种哈希算法	于 v5.4 版本引入

DTFile 存储在数据文件夹目录下的 stable 文件夹内。目前启用的格式均为文件夹形式，即具体数据均储存在名字类似 `dmf_<file id>` 的文件夹下的多个子文件中。

14.3.12.2.1 使用数据校验

TiFlash 支持自动和手动进行数据校验：

- 自动数据校验（`storage.format_version` 配置项）：
  - v6.0.0 之后默认使用 DTFile V3 版本校验机制。

  - v6.0.0 之前默认使用 DTFile V2 版本校验机制。
  - 如需切换版本校验机制，参见TiFlash 配置文件。默认配置经过大量测试，不推荐修改。
- 手动数据校验，参见DTTool 使用文档。

> **警告：**
>
> 设置使用 V3 版本后，新生成的 DTFile 将无法被 v5.4.0 以前 TiFlash 直接正常读取。v5.4.0 后 TiFlash 同时支持 V2，V3 版本，不会主动进行版本的升降级。如果需要迁移到新的版本，或者需要回退到旧的版本，需要手动使用 DTTool 进行版本切换。

14.3.12.2.2 校验工具

除了 TiFlash 在读取所需数据时进行的自动校验，在 v5.4.0 版本时还引入了手动检查文件完整性的工具，详情请见 DTTool 的使用文档。

14.3.13 TiFlash 兼容性说明

TiFlash 在以下情况与 TiDB 存在不兼容问题：

- TiFlash 计算层：
  - 不支持检查溢出的数值。例如将两个 BIGINT 类型的最大值相加 `9223372036854775807 + 9223372036854775807`，该计算在 TiDB 中预期的行为是返回错误 `ERROR 1690 (22003): BIGINT value is out of range`，但如果该计算在 TiFlash 中进行，则会得到溢出的结果 `-2` 且无报错。
  - 不支持从 TiKV 读取数据。
  - 目前 TiFlash 中的 sum 函数不支持传入字符串类型的参数，但 TiDB 在编译时无法检测出这种情况。所以当执行类似于 `select sum(string_col)from t` 的语句时，TiFlash 会报错 `[FLASH:Coprocessor: Unimplemented] CastStringAsReal is not supported.`。要避免这类报错，需要手动把 SQL 改写成 `select sum(cast(string_col as double))from t`。
  - TiFlash 目前的 Decimal 除法计算和 TiDB 存在不兼容的情况。例如在进行 Decimal 相除的时候，TiFlash 会始终按照编译时推断出来的类型进行计算，而 TiDB 则在计算过程中采用精度高于编译时推断出来的类型。这导致在一些带有 Decimal 除法的 SQL 语句在 TiDB + TiKV 上的执行结果会和 TiDB + TiFlash 上的执行结果不一样，示例如下：

```
mysql> create table t (a decimal(3,0), b decimal(10, 0));
Query OK, 0 rows affected (0.07 sec)

mysql> insert into t values (43, 1044774912);
Query OK, 1 row affected (0.03 sec)

mysql> alter table t set tiflash replica 1;
Query OK, 0 rows affected (0.07 sec)
```

```
mysql> set session tidb_isolation_read_engines='tikv';
Query OK, 0 rows affected (0.00 sec)

mysql> select a/b, a/b + 0.0000000000001 from t where a/b;
+--------+------------------------+
| a/b    | a/b + 0.0000000000001 |
+--------+------------------------+
| 0.0000 |       0.0000000410001 |
+--------+------------------------+
1 row in set (0.00 sec)

mysql> set session tidb_isolation_read_engines='tiflash';
Query OK, 0 rows affected (0.00 sec)

mysql> select a/b, a/b + 0.0000000000001 from t where a/b;
Empty set (0.01 sec)
```

以上示例中，在 TiDB 和 TiFlash 中，`a/b` 在编译期推导出来的类型都为 `Decimal(7,4)`，而在 `Decimal(7,4)` 的约束下，`a/b` 返回的结果应该为 `0.0000`。但是在 TiDB 中，`a/b` 运行期的精度比 `Decimal(7,4)` 高，所以原表中的数据没有被 `where a/b` 过滤掉。而在 TiFlash 中 `a/b` 在运行期也是采用 `Decimal(7,4)` 作为结果类型，所以原表中的数据被 `where a/b` 过滤掉了。

14.4 系统变量

TiDB 系统变量的行为与 MySQL 相似，变量的作用范围可以是会话级别有效 (Session Scope) 或全局范围有效 (Global Scope)。其中：

- 对 SESSION 作用域变量的更改，设置后只影响当前会话。
- 对 GLOBAL 作用域变量的更改，设置后立即生效。如果该变量也有 SESSION 作用域，已经连接的所有会话 (包括当前会话) 将继续使用会话当前的 SESSION 变量值。
- 要设置变量值，可使用SET 语句。

```
## 以下两个语句等价地改变一个 Session 变量
SET tidb_distsql_scan_concurrency = 10;
SET SESSION tidb_distsql_scan_concurrency = 10;

## 以下两个语句等价地改变一个 Global 变量
SET @@global.tidb_distsql_scan_concurrency = 10;
SET  GLOBAL tidb_distsql_scan_concurrency = 10;
```

注意：

部分 GLOBAL 作用域的变量会持久化到 TiDB 集群中。文档中的变量有一个“是否持久化到集群”的说明，可以为“是”或者“否”。

- 对于持久化到集群的变量，当该全局变量被修改后，会通知所有 TiDB 服务器刷新其系统变量缓存。在集群中增加一个新的 TiDB 服务器时，或者重启现存的 TiDB 服务器时，都将自动使用该持久化变量。
- 对于不持久化到集群的变量，对变量的修改只对当前连接的 TiDB 实例生效。如果需要保留设置过的值，需要在 `tidb.toml` 配置文件中声明。

此外，由于应用和连接器通常需要读取 MySQL 变量，为了兼容这一需求，在 TiDB 中，部分 MySQL 的变量既可读取也可设置。例如，尽管 JDBC 连接器不依赖于查询缓存 (query cache) 的行为，但仍然可以读取和设置查询缓存。

注意：

变量取较大值并不总会带来更好的性能。由于大部分变量对单个连接生效，设置变量时，还应考虑正在执行语句的并发连接数量。

确定安全值时，应考虑变量的单位：

- 如果单位为线程，安全值通常取决于 CPU 核的数量。
- 如果单位为字节，安全值通常小于系统内存的总量。
- 如果单位为时间，单位可能为秒或毫秒。

单位相同的多个变量可能会争夺同一组资源。

14.4.1 变量参考

14.4.1.1 `allow_auto_random_explicit_insert` 从 v4.0.3 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 是否允许在 INSERT 语句中显式指定含有 AUTO_RANDOM 属性的列的值。

14.4.1.2 `auto_increment_increment`

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：1
- 范围：[1, 65535]
- 控制 AUTO_INCREMENT 自增值字段的自增步长。该变量常与 `auto_increment_offset` 一起使用。

14.4.1.3 auto_increment_offset

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：1
- 范围：[1, 65535]
- 控制 AUTO_INCREMENT 自增值字段的初始值。该变量常与 auto_increment_increment 一起使用。示例如下：

```
mysql> CREATE TABLE t1 (a int not null primary key auto_increment);
Query OK, 0 rows affected (0.10 sec)

mysql> set auto_increment_offset=1;
Query OK, 0 rows affected (0.00 sec)

mysql> set auto_increment_increment=3;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO t1 VALUES (),(),(),();
Query OK, 4 rows affected (0.04 sec)
Records: 4  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM t1;
+----+
| a  |
+----+
|  1 |
|  4 |
|  7 |
| 10 |
+----+
4 rows in set (0.00 sec)
```

14.4.1.4 autocommit

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 用于设置在非显式事务时是否自动提交事务。更多信息，请参见事务概述。

14.4.1.5 block_encryption_mode

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是

- 类型：枚举型
- 默认值：aes-128-ecb
- 可选值：aes-128-ecb、aes-192-ecb、aes-256-ecb、aes-128-cbc、aes-192-cbc、aes-256-cbc、aes-128-ofb、aes-192-ofb、aes-256-ofb、aes-128-cfb、aes-192-cfb、aes-256-cfb
- 该变量用于设置 AES_ENCRYPT() 和 AES_DECRYPT() 函数的加密模式。

14.4.1.6 character_set_client

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：utf8mb4
- 这个变量表示从客户端发出的数据所用的字符集。有关更多 TiDB 支持的字符集和排序规则，参阅字符集和排序规则文档。如果需要更改字符集，建议使用SET NAMES 语句。

14.4.1.7 character_set_connection

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：utf8mb4
- 若没有为字符串常量指定字符集，该变量表示这些字符串常量所使用的字符集。

14.4.1.8 character_set_database

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：utf8mb4
- 该变量表示当前默认在用数据库的字符集，不建议设置该变量。选择新的默认数据库后，服务器会更改该变量的值。

14.4.1.9 character_set_results

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：utf8mb4
- 该变量表示数据发送至客户端时所使用的字符集。

14.4.1.10 character_set_server

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：utf8mb4
- 当 CREATE SCHEMA 中没有指定字符集时，该变量表示这些新建的表结构所使用的字符集。

14.4.1.11 collation_connection

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：utf8mb4_bin
- 该变量表示连接中所使用的排序规则。与 MySQL 中的 collation_connection 一致。

14.4.1.12 collation_database

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：utf8mb4_bin
- 该变量表示当前数据库默认所使用的排序规则。与 MySQL 中的 collation_database 一致。不建议设置此变量，当前使用的数据库变动时，此变量会被 TiDB 修改。

14.4.1.13 collation_server

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：utf8mb4_bin
- 该变量表示创建数据库时默认的排序规则。

14.4.1.14 cte_max_recursion_depth

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：1000
- 范围：[0, 4294967295]
- 这个变量用于控制公共表表达式的最大递归深度。

14.4.1.15 datadir

- 作用域：NONE
- 默认值：使用的组件和部署方式不同，默认值也不同。
  - /tmp/tidb：如果你将--store 设置为 "unistore" 或没有设置 --store，则默认值为 /tmp/tidb。
  - ${pd-ip}:${pd-port}：如果你设置的存储引擎是 TiKV（如果使用 TiUP 和 TiDB Operator 部署，则默认的存储引擎为 TiKV），则默认值为 ${pd-ip}:${pd-port}。
- 这个变量表示数据存储的位置，位置可以是本地路径 /tmp/tidb。如果数据存储在 TiKV 上，则可以是指向 PD 服务器的路径。变量值的格式为 ${pd-ip}:${pd-port}，表示 TiDB 在启动时连接到的 PD 服务器。

14.4.1.16 ddl_slow_threshold

- 作用域：GLOBAL
- 是否持久化到集群：否，仅作用于当前连接的 TiDB 实例
- 类型：整数型
- 默认值：300
- 取值范围：[0, 2147483647]
- 单位：毫秒
- 耗时超过该阈值的 DDL 操作会被输出到日志。

14.4.1.17 default_authentication_plugin

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：mysql_native_password
- 可选值：mysql_native_password，caching_sha2_password，tidb_sm3_password，tidb_auth_token
- tidb_auth_token 认证方式仅用于 TiDB Cloud 内部实现，不要设置为该值。
- 服务器和客户端建立连接时，这个变量用于设置服务器对外通告的默认身份验证方式。如要了解该变量的其他可选值，参见可用的身份验证插件。
- 若要在用户登录时使用 tidb_sm3_password 插件，需要使用 TiDB-JDBC 进行连接。

14.4.1.18 default_password_lifetime 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：0
- 取值范围：[0, 65535]
- 该变量用于设置全局自动密码过期策略，默认值为 0，即禁用全局自动密码过期。如果设置该变量的值为正整数 N，则表示允许的密码生存期为 N，即必须在 N 天之内更改密码。

14.4.1.19 default_week_format

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：0
- 取值范围：[0, 7]
- 设置 WEEK() 函数使用的周格式。

14.4.1.20 disconnect_on_expired_password 从 v6.5.0 版本开始引入

- 作用域：GLOBAL

- 类型：布尔型
- 默认值：ON
- 该变量是一个只读变量，用来显示 TiDB 是否会直接断开密码已过期用户的连接。当其值为 ON，表示 TiDB 会断开密码已过期用户的连接。当其值为 OFF，表示 TiDB 会将密码已过期用户的连接置于“沙盒模式”，允许该用户建立连接并执行密码重置操作。
- 如果需要改变 TiDB 对密码已过期用户连接的处理方式，请在 TiDB 配置文件中的 [security] 部分修改disconnect-on-expired-password 选项。

14.4.1.21 error_count

- 作用域：SESSION
- 默认值：0
- 类型：整数型
- 表示上一条生成消息的 SQL 语句中的错误数。该变量为只读变量。

14.4.1.22 foreign_key_checks

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：在 v6.6.0 之前版本中为 OFF，在 v6.6.0 及之后的版本中为 ON。
- 表示是否开启外键约束检查。

14.4.1.23 group_concat_max_len

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：1024
- 取值范围：[4, 18446744073709551615]
- 表示 GROUP_CONCAT() 函数中，项目的最大缓冲区大小。

14.4.1.24 have_openssl

- 作用域：NONE
- 类型：布尔型
- 默认值：DISABLED
- 用于 MySQL 兼容性的只读变量。当服务器启用 TLS 时，服务器将其设置为 YES。

14.4.1.25 have_ssl

- 作用域：NONE
- 类型：布尔型
- 默认值：DISABLED
- 用于 MySQL 兼容性的只读变量。当服务器启用 TLS 时，服务器将其设置为 YES。

14.4.1.26 hostname

- 作用域：NONE
- 默认值：(系统主机名)
- 这个变量一个只读变量，表示 TiDB server 的主机名。

14.4.1.27 identity 从 v5.3.0 版本开始引入

- 该变量为变量last_insert_id 的别名。

14.4.1.28 init_connect

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：“”
- 用户首次连接到 TiDB 服务器时，init_connect 特性允许 TiDB 自动执行一条或多条 SQL 语句。如果你有 CONNECTION_ADMIN 或者 SUPER 权限，这些 SQL 语句将不会被自动执行。如果这些语句执行报错，你的用户连接将被终止。

14.4.1.29 innodb_lock_wait_timeout

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：50
- 范围：[1, 3600]
- 单位：秒
- 悲观事务语句等锁时间。

14.4.1.30 interactive_timeout

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：28800
- 范围：[1, 31536000]
- 单位：秒
- 该变量表示交互式用户会话的空闲超时。交互式用户会话是指使用 CLIENT_INTERACTIVE 选项调用 mysql_real_connect() API 建立的会话 (例如：MySQL shell 和 MySQL client)。该变量与 MySQL 完全兼容。

14.4.1.31 last_insert_id 从 v5.3.0 版本开始引入

- 作用域：SESSION
- 类型：整数型
- 默认值：0
- 取值范围：[0, 9223372036854775807]
- 返回由 INSERT 语句产生的最新 AUTO_INSCRENT 或者 AUTO_RANDOM 值，与 LAST_INSERT_ID() 的返回的结果相同。与 MySQL 中的 last_insert_id 一致。

14.4.1.32 last_plan_from_binding 从 v4.0 版本开始引入

- 作用域：SESSION
- 默认值：OFF
- 该变量用来显示上一条执行的语句所使用的执行计划是否来自 binding 的执行计划。

14.4.1.33 last_plan_from_cache 从 v4.0 版本开始引入

- 作用域：SESSION
- 默认值：OFF
- 这个变量用来显示上一个 execute 语句所使用的执行计划是不是直接从 plan cache 中取出来的。

14.4.1.34 last_sql_use_alloc 从 v6.4.0 版本开始引入

- 作用域：SESSION
- 默认值：OFF
- 这个变量是一个只读变量，用来显示上一个语句是否使用了缓存的 Chunk 对象 (Chunk allocation)。

14.4.1.35 license

- 作用域：NONE
- 默认值：Apache License 2.0
- 这个变量表示 TiDB 服务器的安装许可证。

14.4.1.36 log_bin

- 作用域：NONE
- 类型：布尔型
- 默认值：OFF
- 该变量表示是否使用TiDB Binlog。

14.4.1.37 max_connections

- 作用域：GLOBAL
- 是否持久化到集群：否，仅作用于当前连接的 TiDB 实例
- 类型：整数型
- 默认值：0
- 取值范围：[0, 100000]
- 该变量表示 TiDB 中同时允许的最大客户端连接数，用于资源控制。
- 默认情况下，该变量值为 0 表示不限制客户端连接数。当本变量的值大于 0 且客户端连接数到达此值时，TiDB 服务端将会拒绝新的客户端连接。

14.4.1.38 max_execution_time

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：0
- 范围：[0, 2147483647]
- 单位：毫秒
- 语句最长执行时间。默认值 (0) 表示无限制。

> 注意：
>
> - max_execution_time 目前只用于控制只读语句的最大执行时长，实际精度在 100ms 级别，而非更准确的毫秒级别。
> - 对于使用了MAX_EXECUTION_TIME Hint 的 SQL 语句，这些语句的最长执行时间将不受该变量限制，而是由该 Hint 进行限制。你也可以使用该 Hint 来创建 SQL 绑定，详情请参考SQL 操作常见问题。

14.4.1.39 max_prepared_stmt_count

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：-1
- 范围：[-1, 1048576]
- 指定当前实例中PREPARE 语句的最大数量。
- 值为 -1 时表示不对实例中的 PREPARE 语句数量进行限制。
- 如果将变量值设为超过上限 1048576，则使用上限值 1048576：

```
mysql> SET GLOBAL max_prepared_stmt_count = 1048577;
Query OK, 0 rows affected, 1 warning (0.01 sec)

mysql> SHOW WARNINGS;
+---------+------+-----------------------------------------------------------------+
| Level   | Code | Message                                                         |
+---------+------+-----------------------------------------------------------------+
| Warning | 1292 | Truncated incorrect max_prepared_stmt_count value: '1048577' |
+---------+------+-----------------------------------------------------------------+
1 row in set (0.00 sec)

mysql> SHOW GLOBAL VARIABLES LIKE 'max_prepared_stmt_count';
+-------------------------+---------+
| Variable_name           | Value   |
```

```
+-------------------------+---------+
| max_prepared_stmt_count | 1048576 |
+-------------------------+---------+
1 row in set (0.00 sec)
```

14.4.1.40 max_allowed_packet 从 v6.1.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：67108864
- 取值范围：[1024, 1073741824]
- 该变量取值应为 1024 的整数倍。若取值无法被 1024 整除，则会提示 warning 并向下取整。例如设置为 1025 时，则 TiDB 中的实际取值为 1024。
- 服务器端和客户端在一次传送数据包的过程中所允许最大的数据包大小，单位为字节。
- 该变量的行为与 MySQL 兼容。

14.4.1.41 mpp_exchange_compression_mode 从 v6.6.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：UNSPECIFIED
- 可选值：NONE，FAST，HIGH_COMPRESSION，UNSPECIFIED
- 该变量用于选择 MPP Exchange 算子的数据压缩模式，当 TiDB 选择版本号为 1 的 MPP 执行计划时生效。该变量值的含义如下：
  - UNSPECIFIED：表示未指定，TiDB 将自动选择压缩模式，当前 TiDB 自动选择 FAST 模式
  - NONE：不使用数据压缩
  - FAST：快速模式，整体性能较好，压缩比小于 HIGH_COMPRESSION
  - HIGH_COMPRESSION：高压缩比模式

14.4.1.42 mpp_version 从 v6.6.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：UNSPECIFIED
- 可选值：UNSPECIFIED，0，1
- 该变量用于指定不同版本的 MPP 执行计划。指定后，TiDB 会选择指定版本的 MPP 执行计划。该变量值含义如下：
  - UNSPECIFIED：表示未指定，此时 TiDB 自动选择最新版本 1。
  - 0：兼容所有 TiDB 集群版本，MPP 版本大于 0 的新特性均不会生效。
  - 1：从 v6.6.0 版本开始引入，用于开启 TiFlash 带压缩的数据交换，详情参见MPP Version 和 Exchange 数据压缩。

14.4.1.43 password_history 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：0
- 范围：[0, 4294967295]
- 该变量用于建立密码重用策略，使 TiDB 基于密码更改次数限制密码的重复使用。该变量默认值为 0，表示禁用基于密码更改次数的密码重用策略。当设置该变量为一个正整数 N 时，表示不允许重复使用最近 N 次使用过的密码。

14.4.1.44 password_reuse_interval 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：0
- 范围：[0, 4294967295]
- 该变量用于建立密码重用策略，使 TiDB 基于经过时间限制密码重复使用。该变量默认值为 0，表示禁用基于密码经过时间的密码重用策略。当设置该变量为一个正整数 N 时，表示不允许重复使用最近 N 天内使用过的密码。

14.4.1.45 plugin_dir

- 作用域：GLOBAL
- 是否持久化到集群：否，仅作用于当前连接的 TiDB 实例
- 默认值：“”
- 指定加载插件的目录。

14.4.1.46 plugin_load

- 作用域：GLOBAL
- 是否持久化到集群：否，仅作用于当前连接的 TiDB 实例
- 默认值：“”
- 指定 TiDB 启动时加载的插件，多个插件之间用逗号（,）分隔。

14.4.1.47 port

- 作用域：NONE
- 默认值：4000
- 范围：[0, 65535]
- 使用 MySQL 协议时 tidb-server 监听的端口。

14.4.1.48 rand_seed1

- 作用域：SESSION
- 默认值：0
- 范围：[0, 2147483647]
- 该变量用于为 SQL 函数 RAND() 中使用的随机值生成器添加种子。
- 该变量的行为与 MySQL 兼容。

14.4.1.49 rand_seed2

- 作用域：SESSION
- 默认值：0
- 范围：[0, 2147483647]
- 该变量用于为 SQL 函数 RAND() 中使用的随机值生成器添加种子。
- 该变量的行为与 MySQL 兼容。

14.4.1.50 require_secure_transport 从 v6.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 该变量控制是否所有 TiDB 的连接都在本地 socket 上进行通信，或使用 TLS。详情见为 TiDB 客户端服务端间通信开启加密传输。
- 该变量设置为 ON 时，必须使用开启 TLS 的会话连接到 TiDB，防止在 TLS 配置不正确时出现锁定的情况。
- 在 v6.1.0 之前这个开关通过 TiDB 配置文件 (security.require-secure-transport) 进行配置，升级到 v6.1.0 时会自动继承原有设置。

14.4.1.51 skip_name_resolve 从 v5.2.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 该变量控制 tidb-server 实例是否将主机名作为连接握手的一部分来解析。
- 当 DNS 不可靠时，可以启用该变量来提高网络性能。

> 注意：
>
> 当 skip_name_resolve 设置为 ON 时，身份信息中包含主机名的用户将无法登录服务器。例如：
>
> ```
> CREATE USER 'appuser'@'apphost' IDENTIFIED BY 'app-password';
> ```
>
> 该示例中，建议将 apphost 替换为 IP 地址或通配符（%）。

14.4.1.52 socket

- 作用域：NONE
- 默认值：“”
- 使用 MySQL 协议时，tidb-server 所监听的本地 unix 套接字文件。

14.4.1.53 sql_log_bin

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 表示是否将更改写入 TiDB Binlog。

> 注意：
>
> 不建议将 sql_log_bin 设置为全局变量，因为 TiDB 的未来版本可能只允许将其设置为会话变量。

14.4.1.54 sql_mode

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值:ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO ↪ ,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION
- 这个变量控制许多 MySQL 兼容行为。详情见SQL 模式。

14.4.1.55 sql_require_primary_key 从 v6.3.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 这个变量用于控制表是否必须有主键。启用该变量后，如果在没有主键的情况下创建或修改表，将返回错误。
- 该功能基于 MySQL 8.0 的特性 sql_require_primary_key。
- 强烈推荐在使用 TiCDC 时启用改变量，因为同步数据变更至 MySQL sink 时要求表必须有主键。

14.4.1.56 sql_select_limit 从 v4.0.2 版本开始引入

- 作用域：SESSION | GLOBAL

- 是否持久化到集群：是
- 默认值：18446744073709551615
- 范围：[0, 18446744073709551615]
- 单位：行
- SELECT 语句返回的最大行数。

14.4.1.57 ssl_ca

- 作用域：NONE
- 默认值：“”
- 证书颁发机构 (CA) 文件的位置。若文件不存在，则变量值为空。该变量的值由 TiDB 配置项ssl-ca 定义。

14.4.1.58 ssl_cert

- 作用域：NONE
- 默认值：“”
- 用于 SSL/TLS 连接的证书文件的位置。若文件不存在，则变量值为空。该变量的值由 TiDB 配置项ssl-cert 定义。

14.4.1.59 ssl_key

- 作用域：NONE
- 默认值：“”
- 用于 SSL/TLS 连接的私钥文件的位置。若文件不存在，则变量值为空。该变量的值由 TiDB 配置项ssl-key 定义。

14.4.1.60 system_time_zone

- 作用域：NONE
- 默认值：(随系统)
- 该变量显示首次引导启动 TiDB 时的系统时区。另请参阅time_zone。

14.4.1.61 tidb_adaptive_closest_read_threshold 从 v6.3.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：4096
- 取值范围：[0, 9223372036854775807]
- 单位：字节
- 这个变量用于控制当replica-read 设置为 closest-adaptive 时，优先将读请求发送至 TiDB server 所在区域副本的阈值。当读请求预估的返回结果的大小超过此阈值时，TiDB 会将读请求优先发送至同一可用区的副本，否则会发送至 leader 副本。

14.4.1.62 tidb_allow_batch_cop 从 v4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：1
- 范围：[0, 2]
- 这个变量用于控制 TiDB 向 TiFlash 发送 coprocessor 请求的方式，有以下几种取值：
  - 0：从不批量发送请求
  - 1：aggregation 和 join 的请求会进行批量发送
  - 2：所有的 cop 请求都会批量发送

14.4.1.63 tidb_allow_fallback_to_tikv 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：“”
- 这个变量表示将 TiKV 作为备用存储引擎的存储引擎列表。当该列表中的存储引擎发生故障导致 SQL 语句执行失败时，TiDB 会使用 TiKV 作为存储引擎再次执行该 SQL 语句。目前支持设置该变量为 “” 或者 “tiflash”。如果设置该变量为 “tiflash”，当 TiFlash 返回超时错误（对应的错误码为 ErrTiFlashServerTimeout）时，TiDB 会使用 TiKV 作为存储引擎再次执行该 SQL 语句。

14.4.1.64 tidb_allow_function_for_expression_index 从 v5.2.0 版本开始引入

- 作用域：NONE
- 默认值：json_array、json_array_append、json_array_insert、json_contains、json_contains_path、json_depth、json_extract、json_insert、json_keys、json_length、json_merge_patch、json_merge_preserve ↪ 、json_object、json_pretty、json_quote、json_remove、json_replace、json_search、json_set、json_storage_size、json_type、json_unquote、json_valid、lower、md5、reverse、tidb_shard、upper、vitess_hash
- 这个变量用于显示创建表达式索引所允许使用的函数。

14.4.1.65 tidb_allow_mpp 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用于控制是否使用 TiFlash 的 MPP 模式执行查询，可以设置的值包括：
  - 0 或 OFF，代表从不使用 MPP 模式
  - 1 或 ON，代表由优化器根据代价估算选择是否使用 MPP 模式（默认）

MPP 是 TiFlash 引擎提供的分布式计算框架，允许节点之间的数据交换并提供高性能、高吞吐的 SQL 算法。MPP 模式选择的详细说明参见控制是否选择 MPP 模式。

14.4.1.66 tidb_allow_remove_auto_inc 从 v2.1.18 和 v3.0.4 版本开始引入

- 作用域：SESSION
- 默认值：OFF
- 这个变量用来控制是否允许通过 ALTER TABLE MODIFY 或 ALTER TABLE CHANGE 来移除某个列的 AUTO_INCREMENT 属性。默认 (OFF) 为不允许。

14.4.1.67 tidb_analyze_partition_concurrency

> 警告：
>
> 当前版本中该变量控制的功能尚未完全生效，请保留默认值。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：1
- 这个变量用于 TiDB analyze 分区表时，对分区表统计信息进行读写的并发度。

14.4.1.68 tidb_analyze_version 从 v5.1.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：2
- 范围：[1, 2]
- 这个变量用于控制 TiDB 收集统计信息的行为。
- 在 v5.3.0 及之后的版本中，该变量的默认值为 2，具体可参照统计信息简介文档。如果从 v5.3.0 之前版本的集群升级至 v5.3.0 及之后的版本，tidb_analyze_version 的默认值不发生变化。

14.4.1.69 tidb_auto_analyze_end_time

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：23:59 +0000
- 这个变量用来设置一天中允许自动 ANALYZE 更新统计信息的结束时间。例如，只允许在 UTC 时间的凌晨 1:00 至 3:00 之间自动更新统计信息，可以设置如下：

  ```
  - tidb_auto_analyze_start_time='01:00 +0000'
  - tidb_auto_analyze_end_time='03:00 +0000'
  ```

14.4.1.70 tidb_auto_analyze_partition_batch_size 从 v6.4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：1
- 范围：[1, 1024]
- 用于设置 TiDB 自动 analyze 分区表（即自动收集分区表上的统计信息）时，每次同时 analyze 分区的个数。
- 若该变量值小于分区表的分区数，则 TiDB 会分多批自动 analyze 该分区表的所有分区。若该变量值大于等于分区表的分区数，则 TiDB 会同时 analyze 该分区表的所有分区。
- 若分区表个数远大于该变量值，且自动 analyze 花费时间较长，可调大该参数的值以减少耗时。

14.4.1.71 tidb_auto_analyze_ratio

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：0.5
- 这个变量用来设置 TiDB 在后台自动执行ANALYZE TABLE 更新统计信息的阈值。0.5 指的是当表中超过 50% 的行被修改时，触发自动 ANALYZE 更新。可以指定 tidb_auto_analyze_start_time 和 tidb_auto_analyze_end_time 来限制自动 ANALYZE 的时间

> 注意：
>
> 当系统变量 tidb_enable_auto_analyze 设置为 ON 时，TiDB 才会触发 auto_analyze。

14.4.1.72 tidb_auto_analyze_start_time

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：00:00 +0000
- 这个变量用来设置一天中允许自动 ANALYZE 更新统计信息的开始时间。例如，只允许在 UTC 时间的凌晨 1:00 至 3:00 之间自动更新统计信息，可以设置如下：

  - tidb_auto_analyze_start_time='01:00 +0000'
  - tidb_auto_analyze_end_time='03:00 +0000'

14.4.1.73 tidb_auto_build_stats_concurrency 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：1
- 范围：[1, 256]
- 这个变量用来设置执行统计信息自动更新的并发度。

14.4.1.74 tidb_backoff_lock_fast

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：10
- 范围：[1, 2147483647]
- 这个变量用来设置读请求遇到锁的 backoff 时间。

14.4.1.75 tidb_backoff_weight

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：2
- 范围：[0, 2147483647]
- 这个变量用来给 TiDB 的 backoff 最大时间增加权重，即内部遇到网络或其他组件 (TiKV, PD) 故障时，发送重试请求的最大重试时间。可以通过这个变量来调整最大重试时间，最小值为 1。

  例如，TiDB 向 PD 取 TSO 的基础超时时间是 15 秒，当 tidb_backoff_weight = 2 时，取 TSO 的最大超时时间为：基础时间 * 2 等于 30 秒。

  在网络环境较差的情况下，适当增大该变量值可以有效缓解因为超时而向应用端报错的情况；而如果应用端希望更快地接到报错信息，则应该尽量减小该变量的值。

14.4.1.76 tidb_batch_commit

> 警告：
>
> 不建议开启此变量。

- 作用域：SESSION
- 类型：布尔型
- 默认值：OFF
- 该变量控制是否启用已废弃的 batch-commit 特性。当该变量开启时，事务可能会通过分组一些语句被拆分为多个事务，并被非原子地提交。不推荐使用这种方式。

14.4.1.77 tidb_batch_delete

> 警告：
>
> 该变量与废弃的 batch-dml 特性相关，可能会导致数据损坏。因此，不建议开启该变量来使用 batch-dml。作为替代，请使用非事务 DML 语句。

- 作用域：SESSION
- 类型：布尔型
- 默认值：OFF
- 该变量控制是否启用已废弃的 batch-dml 特性中的 batch-delete 特性。当该变量开启时，DELETE 语句可能会被拆分为多个事务，并被非原子地提交。要使该特性生效，还需要开启 tidb_enable_batch_dml 并将 tidb_dml_batch_size 的值设置为正数。不推荐使用这种方式。

14.4.1.78 tidb_batch_insert

> 警告：
>
> 该变量与废弃的 batch-dml 特性相关，可能会导致数据损坏。因此，不建议开启该变量来使用 batch-dml。作为替代，请使用非事务 DML 语句。

- 作用域：SESSION
- 类型：布尔型
- 默认值：OFF
- 该变量控制是否启用已废弃的 batch-dml 特性中的 batch-insert 特性。当该变量开启时，INSERT 语句可能会被拆分为多个事务，并被非原子地提交。要使该特性生效，还需要开启 tidb_enable_batch_dml 并将 tidb_dml_batch_size 的值设置为正数。不推荐使用这种方式。

14.4.1.79 tidb_batch_pending_tiflash_count 从 v6.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：4000
- 范围：[0, 4294967295]
- 使用 ALTER DATABASE SET TIFLASH REPLICA 语句为 TiFlash 添加副本时，能容许的不可用表的个数上限。如果超过该上限，则会停止或者以非常慢的速度为库中的剩余表设置 TiFlash 副本。

14.4.1.80 tidb_broadcast_join_threshold_count 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：10240
- 范围：[0, 9223372036854775807]
- 单位：行
- 如果 join 的对象为子查询，优化器无法估计子查询结果集大小，在这种情况下通过结果集行数判断。如果子查询的行数估计值小于该变量，则选择 Broadcast Hash Join 算法。否则选择 Shuffled Hash Join 算法。

14.4.1.81 tidb_broadcast_join_threshold_size 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：104857600 (100 MiB)
- 范围：[0, 9223372036854775807]
- 单位：字节
- 如果表大小（字节数）小于该值，则选择 Broadcast Hash Join 算法。否则选择 Shuffled Hash Join 算法。

14.4.1.82 tidb_build_stats_concurrency

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 单位：线程
- 默认值：4
- 取值范围：[1, 256]
- 这个变量用来设置 ANALYZE 语句执行时并发度。
- 当这个变量被设置得更大时，会对其它的查询语句执行性能产生一定影响。

14.4.1.83 tidb_capture_plan_baselines 从 v4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 这个变量用于控制是否开启自动捕获绑定功能。该功能依赖 Statement Summary，因此在使用自动绑定之前需打开 Statement Summary 开关。
- 开启该功能后会定期遍历一次 Statement Summary 中的历史 SQL 语句，并为至少出现两次的 SQL 语句自动创建绑定。

14.4.1.84 tidb_cdc_write_source 从 v6.5.0 版本开始引入

- 作用域：SESSION
- 是否持久化到集群：否
- 类型：整数型
- 默认值：0
- 范围：[0, 15]
- 当变量非 0 时，该 SESSION 写入的数据将被视为是由 TiCDC 写入的。这个变量仅由 TiCDC 设置，任何时候都不应该手动调整该变量。

14.4.1.85 tidb_check_mb4_value_in_utf8

- 作用域：GLOBAL

- 是否持久化到集群：否，仅作用于当前连接的 TiDB 实例
- 默认值：ON
- 设置该变量为 ON 可强制只存储基本多文种平面 (BMP) 编码区段内的 utf8 字符值。若要存储 BMP 区段外的 utf8 值，推荐使用 utf8mb4 字符集。
- 早期版本的 TiDB 中 (v2.1.x)，utf8 检查更为宽松。如果你的 TiDB 集群是从早期版本升级的，推荐关闭该变量，详情参阅升级与升级后常见问题。

14.4.1.86 tidb_committer_concurrency 从 v6.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：128
- 范围：[1, 10000]
- 在单个事务的提交阶段，用于执行提交操作相关请求的 goroutine 数量。
- 若提交的事务过大，事务提交时的流控队列等待耗时可能会过长。此时，可以通过调大该配置项来加速提交。
- 在 v6.1.0 之前这个开关通过 TiDB 配置文件 (performance.committer-concurrency) 进行配置，升级到 v6.1.0 时会自动继承原有设置。

14.4.1.87 tidb_checksum_table_concurrency

- 作用域：SESSION
- 类型：整数型
- 默认值：4
- 取值范围：[1, 256]
- 单位：线程
- 这个变量用来设置ADMIN CHECKSUM TABLE 语句执行时扫描索引的并发度。当这个变量被设置得更大时，会对其它的查询语句执行性能产生一定影响。

14.4.1.88 tidb_config

- 作用域：GLOBAL
- 是否持久化到集群：否，仅作用于当前连接的 TiDB 实例
- 默认值：“”
- 这个变量是一个只读变量，用来获取当前 TiDB Server 的配置信息。

14.4.1.89 tidb_constraint_check_in_place

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：OFF

- 该变量仅适用于乐观事务模型。悲观事务模式中的行为由tidb_constraint_check_in_place_pessimistic 控制。
- 当这个变量设置为 OFF 时，唯一索引的重复值检查会被推迟到事务提交时才进行。这有助于提高性能，但对于某些应用，可能导致非预期的行为。详情见约束。
  - 乐观事务模型下将 tidb_constraint_check_in_place 设置为 OFF：

```
create table t (i int key);
insert into t values (1);
begin optimistic;
insert into t values (1);
```

```
Query OK, 1 row affected
```

```
tidb> commit; -- 事务提交时才检查
```

```
ERROR 1062 : Duplicate entry '1' for key 't.PRIMARY'
```

  - 乐观事务模型下将 tidb_constraint_check_in_place 设置为 ON：

```
set @@tidb_constraint_check_in_place=ON;
begin optimistic;
insert into t values (1);
```

```
ERROR 1062 : Duplicate entry '1' for key 't.PRIMARY'
```

14.4.1.90 tidb_constraint_check_in_place_pessimistic 从 v6.3.0 版本开始引入

- 作用域：SESSION
- 类型：布尔型
- 默认值：当配置项pessimistic-txn.constraint-check-in-place-pessimistic 为默认值 true 时，该变量的默认值为 ON。当配置项pessimistic-txn.constraint-check-in-place-pessimistic 为 false 时，该变量的默认值为 OFF。
- 该变量仅适用于悲观事务模型。乐观事务模式中的行为由tidb_constraint_check_in_place 控制。
- 当这个变量设置为 OFF 时，唯一约束检查会被推迟到下一次需要对这个索引加锁的语句执行时，或事务提交时才进行。这有助于提高性能，但对于某些应用，可能导致非预期的行为。详情见约束。
- 关闭该变量可能会导致悲观事务中返回 LazyUniquenessCheckFailure 报错。返回该错误时，TiDB 将会回滚当前事务。
- 关闭该变量后，悲观事务中不支持使用SAVEPOINT 功能。
- 关闭该变量时，commit 语句可能会报出 Write conflict 错误或 Duplicate entry 错误，两种错误都意味着事务回滚。

- 悲观事务模型下将 tidb_constraint_check_in_place_pessimistic 设置为 OFF：

```
set @@tidb_constraint_check_in_place_pessimistic=OFF;
create table t (i int key);
insert into t values (1);
begin pessimistic;
insert into t values (1);
```

```
Query OK, 1 row affected
```

```
tidb> commit; -- 事务提交时才检查
```

```
ERROR 1062 : Duplicate entry '1' for key 't.PRIMARY'
```

- 悲观事务模型下将 tidb_constraint_check_in_place_pessimistic 设置为 ON：

```
set @@tidb_constraint_check_in_place_pessimistic=ON;
begin pessimistic;
insert into t values (1);
```

```
ERROR 1062 : Duplicate entry '1' for key 't.PRIMARY'
```

14.4.1.91 tidb_cost_model_version 从 v6.2.0 版本开始引入

> 注意：
>
> - 自 v6.5.0 开始，新创建的 TiDB 集群默认使用 Cost Model Version 2。如果从 v6.4.0 及之前版本的集群升级到 v6.5.0 及之后的版本，tidb_cost_model_version 的值不发生变化。
> - 切换代价模型版本可能会引起查询计划的变动。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：2
- 取值范围：[1, 2]
- 可选值：
  - 1：使用 Cost Model Version 1 代价模型。TiDB v6.4.0 及之前的版本默认使用 Cost Model Version 1。
  - 2：使用 Cost Model Version 2 代价模型。TiDB v6.5.0 正式发布了代价模型Cost Model Version 2，在内部测试中比 Version 1 版本的代价模型更加准确。
- 代价模型会影响优化器对计划的选择，具体可见代价模型。

14.4.1.92 tidb_current_ts

- 作用域：SESSION
- 类型：整数型
- 默认值：0
- 取值范围：[0, 9223372036854775807]
- 这个变量是一个只读变量，用来获取当前事务的时间戳。

14.4.1.93 tidb_ddl_disk_quota 从 v6.3.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：107374182400 (100 GiB)
- 范围：[107374182400, 1125899906842624] ([100 GiB, 1 PiB])
- 单位：字节
- 这个变量仅在tidb_ddl_enable_fast_reorg 开启的情况下生效，用于设置创建索引的回填过程中本地存储空间的使用限制。

14.4.1.94 tidb_ddl_enable_fast_reorg 从 v6.3.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用来控制是否开启添加索引加速功能，来提升创建索引回填过程的速度。开启该变量对于数据量较大的表有一定的性能提升。
- 要验证已经完成的 ADD INDEX 操作是否使用了添加索引加速功能，可以执行ADMIN SHOW DDL JOBS 语句查看 JOB_TYPE 一列中是否含有 ingest 字样。

> **警告：**
>
> 目前，PITR 恢复会额外处理日志备份时间段内通过索引加速功能创建的索引，以达到兼容效果。详细内容请参考索引加速功能为什么与 PITR 功能不兼容。

> **注意：**
>
> - 要使用索引加速功能，你需要提供一个可写且具有足够空余空间的临时路径temp-dir。如果 temp-dir 无法使用，TiDB 会退回到非加速的索引创建方式。建议将 temp-dir 挂载在 SSD 磁盘上。
> - 在升级到 v6.5.0 及以上版本时，请确保 TiDB 的temp-dir 路径已正确挂载了 SSD 磁盘，并确保运行 TiDB 的操作系统用户对该目录有读写权限，否则在运行时可能产生不可预知的问题。该参数是 TiDB 的配置参数，设置后需要重启 TiDB 才能生效。因此，在升级前提前进行设置，可以避免再次重启。

14.4.1.95 tidb_ddl_distribute_reorg 从 v6.6.0 版本开始引入

> 警告：
>
> - 该功能目前为实验特性。不推荐在生产环境中开启该功能。
> - 当前启用此功能后，在 DDL reorg 阶段遇到某些异常只会做简单重试，还没有兼容 DDL 操作的重试方式，即目前无法依据tidb_ddl_error_count_limit 的大小控制重试次数。

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 这个变量用于控制是否开启分布式执行 DDL reorg 阶段，来提升此阶段的速度。目前此开关只对 ADD INDEX 语句有效。开启该变量对于数据量较大的表有一定的性能提升。分布式 DDL 会通过 DDL 动态资源管控，控制 DDL 的 CPU 使用量，来防止对线上业务产生影响。
- 要验证已经完成的ADD INDEX操作是否使用了此功能，可以查看mysql.tidb_background_subtask_history 表是否有对应任务。

14.4.1.96 tidb_ddl_error_count_limit

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：512
- 范围：[0, 9223372036854775807]
- 这个变量用来控制 DDL 操作失败重试的次数。失败重试次数超过该参数的值后，会取消出错的 DDL 操作。

14.4.1.97 tidb_ddl_flashback_concurrency 从 v6.3.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：64
- 范围：[1, 256]
- 这个变量用来控制FLASHBACK CLUSTER TO TIMESTAMP 的并发数。

14.4.1.98 tidb_ddl_reorg_batch_size

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：256

- 范围：[32, 10240]
- 单位：行
- 这个变量用来设置 DDL 操作 re-organize 阶段的 batch size。比如 ADD INDEX 操作，需要回填索引数据，通过并发 tidb_ddl_reorg_worker_cnt 个 worker 一起回填数据，每个 worker 以 batch 为单位进行回填。
  - 如果 ADD INDEX 操作时有较多 UPDATE 操作或者 REPLACE 等更新操作，batch size 越大，事务冲突的概率也会越大，此时建议调小 batch size 的值，最小值是 32。
  - 在没有事务冲突的情况下，batch size 可设为较大值（需要参考 worker 数量，见线上负载与 ADD INDEX 相互影响测试），这样回填数据的速度更快，但是 TiKV 的写入压力也会变大。

14.4.1.99 tidb_ddl_reorg_priority

- 作用域：SESSION
- 类型：枚举型
- 默认值：PRIORITY_LOW
- 可选值：PRIORITY_LOW、PRIORITY_NORMAL、PRIORITY_HIGH
- 这个变量用来设置 ADD INDEX 操作 re-organize 阶段的执行优先级，可设置为 PRIORITY_LOW/PRIORITY_NORMAL/PRIORITY_HIGH。

14.4.1.100 tidb_ddl_reorg_worker_cnt

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：4
- 范围：[1, 256]
- 单位：线程
- 这个变量用来设置 DDL 操作 re-organize 阶段的并发度。

14.4.1.101 tidb_default_string_match_selectivity 从 v6.2.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：浮点型
- 默认值：0.8
- 范围：[0, 1]
- 这个变量用来设置过滤条件中的 like、rlike、regexp 函数在行数估算时的默认选择率，以及是否对这些函数启用 TopN 辅助估算。
- TiDB 总是会尝试利用统计信息对过滤条件中的 like 进行估算，但是当 like 匹配的字符串太复杂时，或者面对 rlike 或 regexp 时，往往无法充分利用统计信息，转而使用 0.8 作为选择率，造成行数估算的误差较大。
- 该变量可以用于修改这个行为，当变量被设为 0 以外的值时，会使用变量的值而不是默认的 0.8 作为选择率。

- 如果将该变量的值设为 0，TiDB 在对上述三个函数进行行数估算时，会尝试利用统计信息中的 TopN 进行求值来提高估算精度，同时也会考虑统计信息中的 NULL 数。求值操作预计会造成少量性能损耗。这个功能生效的前提是统计信息是在tidb_analyze_version 设为 2 时收集的。
- 当该变量的值被设为默认值以外的值的时候，会对 not like、not rlike、not regexp 的行数估算也进行相应的调整。

14.4.1.102 tidb_disable_txn_auto_retry

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用来设置是否禁用显式的乐观事务自动重试，设置为 ON 时，不会自动重试，如果遇到事务冲突需要在应用层重试。

  如果将该变量的值设为 OFF，TiDB 将会自动重试事务，这样在事务提交时遇到的错误更少。需要注意的是，这样可能会导致数据更新丢失。

  这个变量不会影响自动提交的隐式事务和 TiDB 内部执行的事务，它们依旧会根据 tidb_retry_limit 的值来决定最大重试次数。

  关于是否需要禁用自动重试，请参考重试的局限性。

  该变量只适用于乐观事务，不适用于悲观事务。悲观事务的重试次数由max_retry_count 控制。

14.4.1.103 tidb_distsql_scan_concurrency

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：15
- 范围：[1, 256]
- 单位：线程
- 这个变量用来设置 scan 操作的并发度。
- AP 类应用适合较大的值，TP 类应用适合较小的值。对于 AP 类应用，最大值建议不要超过所有 TiKV 节点的 CPU 核数。
- 若表的分区较多可以适当调小该参数（取决于扫描数据量的大小以及扫描频率），避免 TiKV 内存溢出 (OOM)。

14.4.1.104 tidb_dml_batch_size

> 警告：
>
> 该变量与废弃的 batch-dml 特性相关，可能会导致数据损坏。因此，不建议开启该变量来使用 batch-dml。作为替代，请使用非事务 DML 语句。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：0
- 范围：[0, 2147483647]
- 单位：行
- 这个变量的值大于 0 时，TiDB 会将 INSERT 语句在更小的事务中批量提交。这样可减少内存使用，确保大批量修改时事务大小不会达到 txn-total-size-limit 限制。
- 只有变量值为 0 时才符合 ACID 要求。否则无法保证 TiDB 的原子性和隔离性要求。
- 要使该特性生效，还需要开启 tidb_enable_batch_dml，以及至少开启 tidb_batch_insert 和 tidb_batch_delete 中的一个。

> 注意：
>
> 自 v7.0.0 起，tidb_dml_batch_size 对LOAD DATA 语句不再生效。

14.4.1.105 tidb_enable_1pc 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 指定是否在只涉及一个 Region 的事务上启用一阶段提交特性。比起传统两阶段提交，一阶段提交能大幅降低事务提交延迟并提升吞吐。

> 注意：
>
> - 对于新创建的集群，默认值为 ON。对于升级版本的集群，如果升级前是 v5.0 以下版本，升级后默认值为 OFF。
> - 启用 TiDB Binlog 后，开启该选项无法获得性能提升。要获得性能提升，建议使用TiCDC 替代 TiDB Binlog。
> - 启用该参数仅意味着一阶段提交成为可选的事务提交模式，实际由 TiDB 自行判断选择最合适的提交模式进行事务提交。

14.4.1.106 tidb_enable_analyze_snapshot 从 v6.2.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 该变量控制 ANALYZE 读取历史时刻的数据还是读取最新的数据。当该变量设置为 ON 时，ANALYZE 读取 ANALYZE 开始时刻的历史数据。当该变量设置为 OFF 时，ANALYZE 读取最新的数据。
- 在 v5.2 之前，ANALYZE 读取最新的数据。v5.2 至 v6.1 版本 ANALYZE 读取 ANALYZE 开始时刻的历史数据。

> 警告：
>
> 如果 ANALYZE 读取 ANALYZE 开始时刻的历史数据，长时间的 AUTO ANALYZE 可能会因为历史数据被 GC 而出现 GC life time is shorter than transaction duration 的报错。

14.4.1.107 tidb_enable_async_commit 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 该变量控制是否启用 Async Commit 特性，使事务两阶段提交的第二阶段于后台异步进行。开启本特性能降低事务提交的延迟。

> 注意：
>
> - 对于新创建的集群，默认值为 ON。对于升级版本的集群，如果升级前是 v5.0 以下版本，升级后默认值为 OFF。
> - 启用 TiDB Binlog 后，开启该选项无法获得性能提升。要获得性能提升，建议使用TiCDC 替代 TiDB Binlog。
> - 启用该参数仅意味着 Async Commit 成为可选的事务提交模式，实际由 TiDB 自行判断选择最合适的提交模式进行事务提交。

14.4.1.108 tidb_enable_auto_analyze 从 v6.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 该变量控制 TiDB 是否以后台操作自动更新表的统计信息。
- 在 v6.1.0 之前这个开关通过 TiDB 配置文件 (performance.run-auto-analyze) 进行配置，升级到 v6.1.0 时会自动继承原有设置。

14.4.1.109 tidb_enable_auto_increment_in_generated

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 这个变量用于控制是否允许在创建生成列或者表达式索引时引用自增列。

14.4.1.110 tidb_enable_batch_dml

> **警告：**
>
> 该变量与废弃的 batch-dml 特性相关，可能会导致数据损坏。因此，不建议开启该变量来使用 batch-dml。作为替代，请使用非事务 DML 语句。

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 该变量控制是否启用废弃的 batch-dml 特性。启用该变量后，部分语句可能会被拆分为多个事务执行，这是非原子性的，使用时需谨慎。使用 batch-dml 时，必须确保正在操作的数据没有并发操作。要使该变量生效，还需要为 `tidb_batch_dml_size` 指定一个正值，并启用 `tidb_batch_insert` 和 `tidb_batch_delete` 中的至少一个。

14.4.1.111 tidb_enable_cascades_planner

> **警告：**
>
> 目前 cascades planner 为实验特性，不建议在生产环境中使用。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 这个变量用于控制是否开启 cascades planner。

14.4.1.112 tidb_enable_chunk_rpc 从 v4.0 版本开始引入

- 作用域：SESSION
- 默认值：ON
- 这个变量用来设置是否启用 Coprocessor 的 Chunk 数据编码格式。

14.4.1.113 tidb_enable_clustered_index 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 可选值：OFF，ON，INT_ONLY

- 这个变量用于控制默认情况下表的主键是否使用聚簇索引。“默认情况”即不显式指定 CLUSTERED /NONCLUSTERED 关键字的情况。可设置为 OFF/ON/INT_ONLY。
  - OFF 表示所有主键默认使用非聚簇索引。
  - ON 表示所有主键默认使用聚簇索引。
  - INT_ONLY 此时的行为受配置项 alter-primary-key 控制。如果该配置项取值为 true，则所有主键默认使用非聚簇索引；如果该配置项取值为 false，则由单个整数类型的列构成的主键默认使用聚簇索引，其他类型的主键默认使用非聚簇索引。

14.4.1.114 tidb_enable_collect_execution_info

- 作用域：GLOBAL
- 是否持久化到集群：否，仅作用于当前连接的 TiDB 实例
- 默认值：ON
- 这个变量用于控制是否同时将各个执行算子的执行信息记录入 slow query log 中。

14.4.1.115 tidb_enable_column_tracking 从 v5.4.0 版本开始引入

> 警告：
>
> 收集 PREDICATE COLUMNS 的统计信息目前为实验特性，不建议在生产环境中使用。

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 这个变量用于控制是否开启 TiDB 对 PREDICATE COLUMNS 的收集。关闭该变量后，之前收集的 PREDICATE COLUMNS 会被清除。详情见收集部分列的统计信息。

14.4.1.116 tidb_enable_ddl 从 v6.3.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：否，仅作用于当前连接的 TiDB 实例
- 默认值：ON
- 可选值：OFF，ON
- 用于设置该 TiDB 实例是否可以成为 DDL owner。若当前 TiDB 集群中只有一台 TiDB 实例，则不能禁止该实例成为 DDL owner，即不能设置为 OFF。

14.4.1.117 tidb_enable_enhanced_security

- 作用域：NONE
- 默认值：OFF

- 这个变量表示所连接的 TiDB 服务器是否启用了安全增强模式 (SEM)。若要改变该变量值，你需要在 TiDB 服务器的配置文件中修改 enable-sem 项的值，并重启 TiDB 服务器。
- 安全增强模式受安全增强式 Linux 等系统设计的启发，削减拥有 MySQL SUPER 权限的用户能力，转而使用细粒度的 RESTRICTED 权限作为替代。这些细粒度的 RESTRICTED 权限如下：
  - RESTRICTED_TABLES_ADMIN：能够写入 mysql 库中的系统表，能查看 information_schema 表上的敏感列。
  - RESTRICTED_STATUS_ADMIN：能够在 SHOW STATUS 命令中查看敏感内容。
  - RESTRICTED_VARIABLES_ADMIN：能够在 SHOW [GLOBAL] VARIABLES 和 SET 命令中查看和设置包含敏感内容的变量。
  - RESTRICTED_USER_ADMIN：能够阻止其他用户更改或删除用户帐户。
  - RESTRICTED_CONNECTION_ADMIN：能够阻止其它用户使用 KILL 语句终止连接。

14.4.1.118 tidb_enable_external_ts_read 从 v6.4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 当此变量设置为 ON 时，TiDB 会读取tidb_external_ts 指定时间戳前的历史数据。

14.4.1.119 tidb_external_ts 从 v6.4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：0
- 当tidb_enable_external_ts_read 设置为 ON 时，TiDB 会依据该变量指定的时间戳读取历史数据。

14.4.1.120 tidb_restricted_read_only 从 v5.2.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 可选值：OFF 和 ON
- tidb_restricted_read_only和tidb_super_read_only 的作用相似。在大多数情况下，你只需要使用tidb_super_read_only 即可。
- 拥有 SUPER 或 SYSTEM_VARIABLES_ADMIN 权限的用户可以修改该变量。如果 TiDB 开启了安全增强模式，你还需要额外的 RESTRICTED_VARIABLES_ADMIN 权限才能读取或修改该变量。
- tidb_restricted_read_only 的设置将影响tidb_super_read_only 的值：
  - 当设置 tidb_restricted_read_only 为 ON 时，tidb_super_read_only 的将自动被设置为 ON。

  - 当设置 tidb_restricted_read_only 为 OFF 时，tidb_super_read_only 的值将不受影响。
  - 当 tidb_restricted_read_only 为 ON 时，tidb_super_read_only 的值无法设置为 OFF。

- 对于 TiDB 的 DBaaS 供应商，当 TiDB 为另一个数据库的下游数据库时，如果要将整个 TiDB 集群设置为只读模式，你需要开启安全增强模式 并将 tidb_restricted_read_only 设置为 ON，从而防止你的用户通过tidb_super_read_only 将 TiDB 集群设置为可写。实现方法：首先开启安全增强模式，然后由你（作为 DBaaS 的控制面）使用一个 admin 用户控制 tidb_restricted_read_only（需要拥有 SYSTEM_VARIABLES_ADMIN 和 RESTRICTED_VARIABLES_ADMIN 权限），由你的数据库用户使用 root 用户控制tidb_super_read_only（需要拥有 SUPER 权限）。
- 该变量可以控制整个集群的只读状态。开启后（即该值为 ON），整个集群中的 TiDB 服务器都将进入只读状态，只有 SELECT、USE、SHOW 等不会修改数据的语句才能被执行，其他如 INSERT、UPDATE 等语句会被拒绝执行。
- 该变量开启只读模式只保证整个集群最终进入只读模式，当变量修改状态还没被同步到其他 TiDB 服务器时，尚未同步的 TiDB 仍然停留在非只读模式。
- 在执行 SQL 语句之前，TiDB 会检查集群的只读标志。从 v6.2.0 起，在提交 SQL 语句之前，TiDB 也会检查该标志，从而防止在服务器被置于只读模式后某些长期运行的auto commit 语句可能修改数据的情况。
- 在变量开启时，对于尚未提交的事务：
  - 如果有尚未提交的只读事务，可正常提交该事务。
  - 如果尚未提交的事务为非只读事务，在事务内执行写入的 SQL 语句会被拒绝。
  - 如果尚未提交的事务已经有数据改动，其提交也会被拒绝。
- 当集群开启只读模式后，所有用户（包括 SUPER 用户）都无法执行可能写入数据的 SQL 语句，除非该用户被显式地授予了 RESTRICTED_REPLICA_WRITER_ADMIN 权限。

14.4.1.121 tidb_enable_exchange_partition

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 该变量用于设置是否启用exchange partitions with tables 特性。默认值为 ON，即默认开启该功能。
- 该变量自 v6.3.0 开始废弃，其取值将固定为默认值 ON，即默认开启 exchange partitions with tables。

14.4.1.122 tidb_enable_extended_stats

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 该变量指定 TiDB 是否收集扩展统计信息来指导优化器。

14.4.1.1.123 tidb_enable_fast_analyze

> **警告：**
>
> 目前快速分析功能为实验特性，不建议在生产环境中使用。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 这个变量用来控制是否启用统计信息快速分析功能。默认值 0 表示不开启。
- 快速分析功能开启后，TiDB 会随机采样约 10000 行的数据来构建统计信息。因此在数据分布不均匀或者数据量比较少的情况下，统计信息的准确度会比较低。这可能导致执行计划不优，比如选错索引。如果可以接受普通 ANALYZE 语句的执行时间，则推荐关闭快速分析功能。

14.4.1.1.124 tidb_enable_foreign_key 从 v6.3.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：在 v6.6.0 之前版本中为 OFF，在 v6.6.0 及之后的版本中为 ON。
- 这个变量用于控制是否开启 FOREIGN KEY 特性。

14.4.1.1.125 tidb_enable_gc_aware_memory_track

> **警告：**
>
> 该变量为 TiDB 内部调试变量，可能会在未来版本中删除，请勿设置该变量。

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 这个变量用于控制是否开启可感知到垃圾回收的内存追踪 (GC-Aware memory track)。

14.4.1.1.126 tidb_enable_non_prepared_plan_cache

> **警告：**
>
> 非 Prepare 语句执行计划缓存目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现 bug，请在 GitHub 上提 issue 反馈。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 这个变量用来控制是否开启非 Prepare 语句执行计划缓存。

14.4.1.127 tidb_enable_gogc_tuner 从 v6.4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 该变量来用控制是否开启 GOGC Tuner。

14.4.1.128 tidb_enable_historical_stats

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 这个变量用来控制是否开启历史统计信息。默认值为 ON 表示默认开启历史统计信息。

14.4.1.129 tidb_enable_historical_stats_for_capture

> 警告：
>
> 当前版本中该变量控制的功能尚未完全生效，请保留默认值。

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 这个变量用来控制 PLAN REPLAYER CAPTURE 抓取的内容是否默认带历史统计信息。默认值为 OFF，表示默认不带历史统计信息。

14.4.1.130 tidb_enable_index_merge 从 v4.0 版本开始引入

> 注意：

- 当集群从 v4.0.0 以下版本升级到 v5.4.0 及以上版本时，该变量开关默认关闭，防止升级后计划发生变化导致回退。
- 当集群从 v4.0.0 及以上版本升级到 v5.4.0 及以上版本时，该变量开关保持升级前的状态。
- 对于 v5.4.0 及以上版本的新建集群，该变量开关默认开启。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用于控制是否开启 index merge 功能。

14.4.1.131 tidb_enable_index_merge_join

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 表示是否启用 IndexMergeJoin 算子。
- 该变量为 TiDB 内部变量，不推荐使用，否则可能会造成数据正确性问题。

14.4.1.132 tidb_enable_legacy_instance_scope 从 v6.0.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用于允许使用 SET SESSION 对 INSTANCE 作用域的变量进行设置，用法同 SET GLOBAL。
- 为了兼容之前的 TiDB 版本，该变量值默认为 ON。

14.4.1.133 tidb_enable_list_partition 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用来设置是否开启 LIST (COLUMNS)TABLE PARTITION 特性。

14.4.1.134 tidb_enable_local_txn

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 该变量用于一个未发布的特性，请勿修改该变量值。

14.4.1.135 tidb_enable_metadata_lock 从 v6.3.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用来设置是否开启元数据锁特性。需要注意，在设置该变量时，集群中不能有 DDL 任务，以免造成非预期数据正确性、一致性问题。

14.4.1.136 tidb_enable_mutation_checker 从 v6.0.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 这个变量用于设置是否开启 mutation checker。mutation checker 是一项在 DML 语句执行过程中进行的数据索引一致性校验，校验报错会回滚当前语句。开启该校验会导致 CPU 使用轻微上升。详见数据索引一致性报错。
- 对于新创建的 v6.0.0 及以上的集群，默认值为 ON。对于升级版本的集群，如果升级前是低于 v6.0.0 的版本，升级后默认值为 OFF。

14.4.1.137 tidb_enable_new_cost_interface 从 v6.2.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- TiDB v6.2.0 对代价模型的实现进行了代码层面的重构，这个变量用来控制是否使用重构后的代价模型Cost Model Version 2。
- 重构后的代价模型使用完全一样的代价公式，因此不会引起计划选择的变动，此开关默认打开。
- 从 v6.1 升级至 v6.2 的用户，此开关保持升级前的 OFF 状态，此时建议直接打开；对于从 v6.1 之前版本升级至 v6.2 的用户，此开关默认为 ON。

14.4.1.138 tidb_enable_new_only_full_group_by_check 从 v6.1.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 该变量用于控制 TiDB 执行 ONLY_FULL_GROUP_BY 检查时的行为。有关 ONLY_FULL_GROUP_BY 的信息可以参考 MySQL 文档。在 v6.1 中 TiDB 对该项检查做了更严格正确的处理。
- 由于可能存在版本升级造成的兼容性问题，在 v6.1 中该变量默认值是 OFF，即默认关闭。

14.4.1.139 tidb_enable_noop_functions 从 v4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 默认情况下，用户尝试将某些语法用于尚未实现的功能时，TiDB 会报错。若将该变量值设为 ON，TiDB 则自动忽略此类功能不可用的情况，即不会报错。若用户无法更改 SQL 代码，可考虑将变量值设为 ON。
- 启用 noop 函数可以控制以下行为：
  - LOCK IN SHARE MODE 语法
  - SQL_CALC_FOUND_ROWS 语法
  - START TRANSACTION READ ONLY 和 SET TRANSACTION READ ONLY 语法
  - tx_read_only、transaction_read_only、offline_mode、super_read_only、read_only 以及 sql_auto_is_null 系统变量
  - GROUP BY <expr> ASC|DESC 语法

> **警告：**
>
> 该变量只有在默认值 OFF 时，才算是安全的。因为设置 tidb_enable_noop_functions=1 后，TiDB 会自动忽略某些语法而不报错，这可能会导致应用程序出现异常行为。例如，允许使用语法 START TRANSACTION READ ONLY 时，事务仍会处于读写模式。

14.4.1.140 tidb_enable_noop_variables 从 v6.2.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 若该变量值为 OFF，TiDB 具有以下行为：
  - 使用 SET 设置 noop 的系统变量时会报 "setting *variable_name* has no effect in TiDB" 的警告。
  - SHOW [SESSION | GLOBAL] VARIABLES 的结果不显示 noop 的系统变量。
  - 使用 SELECT 读取 noop 的系统变量时会报 "variable *variable_name* has no effect in TiDB" 的警告。
- 你可以通过 SELECT * FROM INFORMATION_SCHEMA.CLIENT_ERRORS_SUMMARY_GLOBAL; 语句来检查 TiDB 实例是否曾设置和读取 noop 系统变量。

14.4.1.141 tidb_enable_null_aware_anti_join 从 v6.3.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：在 v7.0.0 之前版本中为 OFF，即默认关闭。在 v7.0.0 及之后的版本中为 ON，即默认开启。

- 这个变量用于控制 TiDB 对特殊集合算子 NOT IN 和 != ALL 引导的子查询产生的 ANTI JOIN 是否采用 Null Aware Hash Join 的执行方式。
- 从旧版本升级到 v7.0.0 及之后版本，该功能自动开启，即该变量的值修改为默认值 ON。

14.4.1.142 tidb_enable_outer_join_reorder 从 v6.1.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 自 v6.1.0 起，TiDB 的Join Reorder 算法开始支持 Outer Join。该变量用于控制是否启用 Outer Join 的 Join Reorder。
- 对于从较低版本升级到当前版本的 TiDB：
  - 如果升级前 TiDB 的版本低于 v6.1.0，升级后该变量的默认值为 ON。
  - 如果升级前 TiDB 的版本等于或大于 v6.1.0，升级后该变量的默认值跟随升级前的设定值。

14.4.1.143 tidb_enable_inl_join_inner_multi_pattern 从 v7.0.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 该变量用于控制当内表上有 Selection/Projection 算子时是否支持 Index Join。 OFF 表示不支持。

14.4.1.144 tidb_enable_ordered_result_mode

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 指定是否对最终的输出结果进行自动排序。
- 例如，开启该变量后，TiDB 会将 SELECT a, MAX(b)FROM t GROUP BY a 处理为 SELECT a, MAX(b)FROM t GROUP BY a ORDER BY a, MAX(b)。

14.4.1.145 tidb_enable_paging 从 v5.4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON

- 这个变量用于控制是否使用分页 (paging) 方式发送 Coprocessor 请求。对于 [v5.4.0, v6.2.0) 区间的 TiDB 版本，该变量只对 IndexLookup 算子生效；对于 v6.2.0 以及之后的版本，该变量对全局生效。从 v6.4.0 版本开始，该变量默认值由 OFF 改成 ON。
- 适用场景：
  - 推荐在所有偏 OLTP 的场景下使用 paging。
  - 对于使用 IndexLookUp 和 Limit 并且 Limit 无法下推到 IndexScan 上的读请求，可能会出现读请求的延迟高、TiKV 的 Unified read pool CPU 使用率高的情况。在这种情况下，由于 Limit 算子只需要少部分数据，开启tidb_enable_paging 能够减少处理数据的数量，从而降低延迟、减少资源消耗。
  - 对于Dumpling 数据导出或者全表扫描这类的场景，开启 paging 后可以有效降低 TiDB 进程的内存消耗。

> 注意：
>
> 对于偏 OLAP 的场景，并且以 TiKV 而非 TiFlash 作为存储引擎时，开启 paging 可能导致部分场景下性能回退。此时，你可以考虑通过该变量关闭 paging 或者通过系统变量tidb_min_paging_size 和tidb_max_paging_size 调整 paging size 的行数范围。

14.4.1.146 tidb_enable_parallel_apply 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：0
- 这个变量用于控制是否开启 Apply 算子并发，并发数由 tidb_executor_concurrency 变量控制。Apply 算子用来处理关联子查询且默认无并发，所以执行速度较慢。打开 Apply 并发开关可增加并发度，提高执行速度。目前默认关闭。

14.4.1.147 tidb_enable_pipelined_window_function

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 该变量指定是否对窗口函数采用流水线的执行算法。

14.4.1.148 tidb_enable_plan_cache_for_param_limit 从 v6.6.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 这个变量用来控制 Prepared Plan Cache 是否缓存 LIMIT 后面带变量 (LIMIT ?) 的执行计划。目前不支持缓存 LIMIT 后面带变量且变量值大于 10000 的执行计划。

14.4.1.149 tidb_enable_plan_cache_for_subquery 从 v7.0.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 这个变量用来控制 Prepared Plan Cache 是否缓存包含子查询的查询。

14.4.1.150 tidb_enable_plan_replayer_capture

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 这个变量用来控制是否开启PLAN REPLAYER CAPTURE 功能。默认值 ON 代表开启 PLAN REPLAYER CAPTURE 功能。

14.4.1.151 tidb_enable_plan_replayer_continuous_capture 从 v7.0.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 这个变量用来控制是否开启PLAN REPLAYER CONTINUOUS CAPTURE 功能。默认值 OFF 代表关闭功能。

14.4.1.152 tidb_enable_prepared_plan_cache 从 v6.1.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用来控制是否开启Prepared Plan Cache。开启后，对 Prepare、Execute 请求的执行计划会进行缓存，以便在后续执行时跳过查询计划优化这个步骤，获得性能上的提升。
- 在 v6.1.0 之前这个开关通过 TiDB 配置文件 (prepared-plan-cache.enabled) 进行配置，升级到 v6.1.0 时会自动继承原有设置。

14.4.1.153 tidb_enable_prepared_plan_cache_memory_monitor 从 v6.4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用来控制是否统计 Prepared Plan Cache 中所缓存的执行计划占用的内存。具体可见Prepared Plan Cache 的内存管理。

14.4.1.154 tidb_enable_pseudo_for_outdated_stats 从 v5.3.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 这个变量用来控制优化器在一张表上的统计信息过期时的行为。
- 统计信息过期的判断标准：最近一次对某张表执行 ANALYZE 获得统计信息后，该表数据被修改的行数大于该表总行数的 80%，便可判定该表的统计信息已过期。该比例可通过pseudo-estimate-ratio 配置参数调整。
- 默认情况下（即该变量值为 OFF 时），某张表上的统计信息过期后，优化器仍会使用该表上的统计信息。将该变量值设为 ON 时，当统计信息过期后，优化器认为该表上除总行数以外的统计信息不再可靠，转而使用 pseudo 统计信息。
- 如果表数据修改较频繁，没有及时对表执行 ANALYZE，但又希望执行计划保持稳定，推荐将该变量值设为 OFF。

14.4.1.155 tidb_enable_rate_limit_action

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 这个变量控制是否为读数据的算子开启动态内存控制功能。读数据的算子默认启用tidb_distsql_scan_concurrency ↪ 所允许的最大线程数来读取数据。当单条 SQL 语句的内存使用每超过tidb_mem_quota_query 一次，读数据的算子会停止一个线程。
- 当读数据的算子只剩 1 个线程且当单条 SQL 语句的内存使用继续超过tidb_mem_quota_query 时，该 SQL 语句会触发其它的内存控制行为，例如落盘。
- 该变量在单条查询仅涉及读数据的情况下，对内存控制效果较好。若还存在额外的计算操作（如连接、聚合等），打开该变量可能会导致内存不受 tidb_mem_quota_query 控制，加剧 OOM 风险。

14.4.1.156 tidb_enable_resource_control 从 v6.6.0 版本开始引入

> 警告：
>
> 资源管控 目前为实验性特性，此变量定义可能在之后发生变化或者删除。

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 类型：布尔型
- 该变量是资源管控特性的开关。该变量设置为 ON 时，集群支持应用按照资源组做资源隔离。

14.4.1.157 tidb_enable_reuse_chunk 从 v6.4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 可选值：OFF，ON
- 该变量用于控制 TiDB 是否启用 Chunk 对象缓存。如果为 ON，则优先使用缓存中的 Chunk 对象，缓存中找不到申请的对象时才会从系统内存中申请。如果为 OFF，则直接从系统内存中申请 Chunk 对象。

14.4.1.158 tidb_enable_slow_log

- 作用域：GLOBAL
- 是否持久化到集群：否，仅作用于当前连接的 TiDB 实例
- 默认值：ON
- 这个变量用于控制是否开启 slow log 功能。

14.4.1.159 tidb_enable_stmt_summary 从 v3.0.4 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用来控制是否开启 statement summary 功能。如果开启，SQL 的耗时等执行信息将被记录到系统表 information_schema.STATEMENTS_SUMMARY 中，用于定位和排查 SQL 性能问题。

14.4.1.160 tidb_enable_strict_double_type_check 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用来控制是否可以用 DOUBLE 类型的无效定义创建表。该设置的目的是提供一个从 TiDB 早期版本升级的方法，因为早期版本在验证类型方面不太严格。
- 该变量的默认值 ON 与 MySQL 兼容。

例如，由于无法保证浮点类型的精度，现在将 DOUBLE(10) 类型视为无效。将 tidb_enable_strict_double_type_check ↪ 更改为 OFF 后，将会创建表。如下所示：

```
CREATE TABLE t1 (id int, c double(10));
ERROR 1149 (42000): You have an error in your SQL syntax; check the manual that corresponds to
    ↪ your MySQL server version for the right syntax to use
SET tidb_enable_strict_double_type_check = 'OFF';
Query OK, 0 rows affected (0.00 sec)
CREATE TABLE t1 (id int, c double(10));
Query OK, 0 rows affected (0.09 sec)
```

> **注意：**
>
> 该设置仅适用于 DOUBLE 类型，因为 MySQL 允许为 FLOAT 类型指定精度。从 MySQL 8.0.17 开始已弃用此行为，不建议为 FLOAT 或 DOUBLE 类型指定精度。

14.4.1.161 tidb_enable_table_partition

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 可选值：OFF，ON，AUTO
- 这个变量用来设置是否开启 TABLE PARTITION 特性。目前变量支持以下三种值：
  - 默认值 ON 表示开启 TiDB 当前已实现了的分区表类型，目前 Range partition、 Hash partition 以及 Range column 单列的场景会生效。
  - AUTO 目前作用和 ON 一样。
  - OFF 表示关闭 TABLE PARTITION 特性，此时语法还是保持兼容，只是创建的表并不是真正的分区表，而是普通的表。

14.4.1.162 tidb_enable_telemetry 从 v4.0.2 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 这个变量用于动态地控制 TiDB 遥测功能是否开启，当前版本默认关闭 TiDB 的遥测功能。当所有 TiDB 实例都设置配置项enable-telemetry 为 false 时，将忽略该系统变量，并总是关闭 TiDB 遥测功能。参阅遥测了解该功能详情。

14.4.1.163 tidb_enable_tiflash_read_for_write_stmt 从 v6.3.0 版本开始引入

> **警告：**
>
> 当前版本中该变量控制的功能是实验性功能，暂不建议在生产环境中使用。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF

- 这个变量用于控制包含增删改的 SQL 语句中的读取操作能否下推到 TiFlash，比如：
  - INSERT INTO SELECT 语句中的 SELECT 查询（典型应用场景为TiFlash 查询结果物化）
  - UPDATE 和 DELETE 语句中的 WHERE 条件过滤

14.4.1.164 tidb_enable_tmp_storage_on_oom

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 设置是否在单条 SQL 语句的内存使用超出系统变量tidb_mem_quota_query 限制时为某些算子启用临时磁盘。
- 在 v6.3.0 之前这个开关可通过 TiDB 配置文件中的 oom-use-tmp-storage 项进行配置。在升级到 v6.3.0 及更新的版本后，集群会自动使用原 oom-use-tmp-storage 的值来初始化该开关，配置文件中 oom-use-tmp-storage 的新设置不再影响该开关。

14.4.1.165 tidb_enable_top_sql 从 v5.4.0 版本开始引入

> **警告：**
>
> Top SQL 目前是实验性功能，不建议在生产环境中使用。

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 这个变量用于控制是否开启Top SQL 特性。

14.4.1.166 tidb_enable_tso_follower_proxy 从 v5.3.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 这个变量用来开启 TSO Follower Proxy 特性。当该值为 OFF 时，TiDB 仅会从 PD leader 获取 TSO。开启该特性之后，TiDB 在获取 TSO 时会将请求均匀地发送到所有 PD 节点上，通过 PD follower 转发 TSO 请求，从而降低 PD leader 的 CPU 压力。
- 适合开启 TSO Follower Proxy 的场景：
  - PD leader 因高压力的 TSO 请求而达到 CPU 瓶颈，导致 TSO RPC 请求的延迟较高。
  - 集群中的 TiDB 实例数量较多，且调高tidb_tso_client_batch_max_wait_time 并不能缓解 TSO RPC 请求延迟高的问题。

> 注意：
>
> 如果 PD leader 的 TSO RPC 延迟升高，但其现象并非由 CPU 使用率达到瓶颈而导致（可能存在网络等问题），此时，打开 TSO Follower Proxy 可能会导致 TiDB 的语句执行延迟上升，从而影响集群的 QPS 表现。

14.4.1.167 tidb_enable_unsafe_substitute 从 v6.3.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 这个变量用于控制是否对生成列中表达式替换使用不安全的替换方式。默认值为 OFF，即默认关闭不安全的替换方式。详情见生成列。

14.4.1.168 tidb_enable_vectorized_expression 从 v4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用于控制是否开启向量化执行。

14.4.1.169 tidb_enable_window_function

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用来控制是否开启窗口函数的支持。默认值 1 代表开启窗口函数的功能。
- 由于窗口函数会使用一些保留关键字，可能导致原先可以正常执行的 SQL 语句在升级 TiDB 后无法被解析语法，此时可以将 tidb_enable_window_function 设置为 OFF。

14.4.1.170 tidb_enforce_mpp 从 v5.1 版本开始引入

- 作用域：SESSION
- 默认值：OFF（表示关闭）。如需修改此变量的默认值，请配置performance.enforce-mpp 参数。
- 这个变量用于控制是否忽略优化器代价估算，强制使用 TiFlash 的 MPP 模式执行查询，可以设置的值包括：
  - 0 或 OFF，代表不强制使用 MPP 模式（默认）
  - 1 或 ON，代表将忽略代价估算，强制使用 MPP 模式。注意：只有当 tidb_allow_mpp=true 时该设置才生效。

MPP 是 TiFlash 引擎提供的分布式计算框架，允许节点之间的数据交换并提供高性能、高吞吐的 SQL 算法。MPP 模式选择的详细说明参见控制是否选择 MPP 模式。

14.4.1.171 tidb_evolve_plan_baselines 从 v4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 这个变量用于控制是否启用自动演进绑定功能。该功能的详细介绍和使用方法可以参考自动演进绑定。
- 为了减少自动演进对集群的影响，可以进行以下配置：
  - 设置 tidb_evolve_plan_task_max_time，限制每个执行计划运行的最长时间，其默认值为 600s；
  - 设置tidb_evolve_plan_task_start_time 和 tidb_evolve_plan_task_end_time，限制运行演进任务的时间窗口，默认值分别为 00:00 +0000 和 23:59 +0000。

14.4.1.172 tidb_evolve_plan_task_end_time 从 v4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：23:59 +0000
- 这个变量用来设置一天中允许自动演进的结束时间。

14.4.1.173 tidb_evolve_plan_task_max_time 从 v4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：600
- 范围：[-1, 9223372036854775807]
- 单位：秒
- 该变量用于限制自动演进功能中，每个执行计划运行的最长时间。

14.4.1.174 tidb_evolve_plan_task_start_time 从 v4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：00:00 +0000
- 这个变量用来设置一天中允许自动演进的开始时间。

14.4.1.175 tidb_executor_concurrency 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：5
- 范围：[1, 256]
- 单位：线程

变量用来统一设置各个 SQL 算子的并发度，包括：

- index lookup
- index lookup join
- hash join
- hash aggregation（partial 和 final 阶段）
- window
- projection

tidb_executor_concurrency 整合了已有的系统变量，方便管理。这些变量所列如下：

- tidb_index_lookup_concurrency
- tidb_index_lookup_join_concurrency
- tidb_hash_join_concurrency
- tidb_hashagg_partial_concurrency
- tidb_hashagg_final_concurrency
- tidb_projection_concurrency
- tidb_window_concurrency

v5.0 后，用户仍可以单独修改以上系统变量（会有废弃警告），且修改只影响单个算子。后续通过 tidb_executor_concurrency 的修改也不会影响该算子。若要通过 tidb_executor_concurrency 来管理所有算子的并发度，需要将以上所列变量的值设置为 -1。

对于从 v5.0 之前的版本升级到 v5.0 的系统，如果用户对上述所列变量的值没有做过改动（即 tidb_hash_join_concurrency 值为 5，其他值为 4），则会自动转为使用 tidb_executor_concurrency 来统一管理算子并发度。如果用户对上述变量的值做过改动，则沿用之前的变量对相应的算子做并发控制。

14.4.1.176 tidb_expensive_query_time_threshold

- 作用域：GLOBAL
- 是否持久化到集群：否，仅作用于当前连接的 TiDB 实例
- 默认值：60
- 范围：[10, 2147483647]
- 单位：秒
- 这个变量用来控制打印 expensive query 日志的阈值时间，默认值是 60 秒。expensive query 日志和慢日志的差别是，慢日志是在语句执行完后才打印，expensive query 日志可以把正在执行中的语句且执行时间超过阈值的语句及其相关信息打印出来。

14.4.1.177 tidb_force_priority

- 作用域：GLOBAL
- 是否持久化到集群：否，仅作用于当前连接的 TiDB 实例
- 类型：枚举型
- 默认值：NO_PRIORITY
- 可选值：NO_PRIORITY、LOW_PRIORITY、DELAYED、HIGH_PRIORITY

- 这个变量用于改变 TiDB server 上执行的语句的默认优先级。例如，你可以通过设置该变量来确保正在执行 OLAP 查询的用户优先级低于正在执行 OLTP 查询的用户。
- 默认值 NO_PRIORITY 表示不强制改变执行语句的优先级。

14.4.1.178 tidb_gc_concurrency 从 v5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：-1
- 范围：[1, 256]
- 单位：线程
- 这个变量用于指定 GC 在Resolve Locks（清理锁）步骤中线程的数量。默认值 -1 表示由 TiDB 自主判断运行 GC 要使用的线程的数量。

14.4.1.179 tidb_gc_enable 从 v5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用于控制是否启用 TiKV 的垃圾回收 (GC) 机制。如果不启用 GC 机制，系统将不再清理旧版本的数据，因此会有损系统性能。

14.4.1.180 tidb_gc_life_time 从 v5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：10m0s
- 范围：[10m0s, 8760h0m0s]
- 这个变量用于指定每次进行垃圾回收 (GC) 时保留数据的时限。变量值为 Go 的 Duration 字符串格式。每次进行 GC 时，将以当前时间减去该变量的值作为 safe point。

> Note:
>
> - 在数据频繁更新的场景下，将 tidb_gc_life_time 的值设置得过大（如数天甚至数月）可能会导致一些潜在的问题，如：
>   - 占用更多的存储空间。
>   - 大量的历史数据可能会在一定程度上影响系统性能，尤其是范围的查询（如 select count(*)from t）。
> - 如果一个事务的运行时长超过了 tidb_gc_life_time 配置的值，在 GC 时，为了使这个事务可以继续正常运行，系统会保留从这个事务开始时间 start_ts 以来的数据。例如，如果 tidb_gc_life_time 的值配置为 10 分钟，且在一次 GC 时，集群正在运行的事务中最早开始的那个事务已经运行了 15 分钟，那么本次 GC 将保留最近 15 分钟的数据。

14.4.1.181 tidb_gc_max_wait_time 从 v6.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：86400
- 范围：[600, 31536000]
- 单位：秒
- 这个变量用于指定活跃事务阻碍 GC safe point 推进的最大时间。每次进行 GC 时，默认 GC safe point 不会超过正在执行中的事务的开始时间。如果活跃事务运行时间未超过该值，GC safe point 会一直被阻塞不更新，直到活跃事务运行时间超过该值 safe point 才会正常推进。

14.4.1.182 tidb_gc_run_interval 从 v5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：10m0s
- 范围：[10m0s, 8760h0m0s]
- 这个变量用于指定垃圾回收 (GC) 运行的时间间隔。变量值为 Go 的 Duration 字符串格式，如"1h30m"、"15m"等。

14.4.1.183 tidb_gc_scan_lock_mode 从 v5.0 版本开始引入

> 警告：
>
> Green GC 目前是实验性功能，不建议在生产环境中使用。

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：LEGACY
- 可设置为：PHYSICAL，LEGACY
  - LEGACY：使用旧的扫描方式，即禁用 Green GC。
  - PHYSICAL：使用物理扫描方式，即启用 Green GC。
- 这个变量用于指定垃圾回收 (GC) 的 Resolve Locks（清理锁）步骤中扫描锁的方式。当变量值设置为 LEGACY 时，TiDB 以 Region 为单位进行扫描。当变量值设置为 PHYSICAL 时，每个 TiKV 节点分别绕过 Raft 层直接扫描数据，可以有效地缓解在启用Hibernate Region 功能时，GC 唤醒全部 Region 的影响，从而提升 Resolve Locks（清理锁）这个步骤的执行速度。

14.4.1.184 tidb_general_log

- 作用域：GLOBAL

- 是否持久化到集群：否，仅作用于当前连接的 TiDB 实例
- 默认值：OFF
- 这个变量用来设置是否在日志里记录所有的 SQL 语句。该功能默认关闭。如果系统运维人员在定位问题过程中需要追踪所有 SQL 记录，可考虑开启该功能。
- 在 TiDB 配置项 log.level 为 "info" 或 "debug" 时，通过查询 "GENERAL_LOG" 字符串可以定位到该功能在日志中的所有记录。日志会记录以下内容：
  - conn：当前会话对应的 ID
  - user：当前会话用户
  - schemaVersion：当前 schema 版本
  - txnStartTS：当前事务的开始时间戳
  - forUpdateTS：事务模式为悲观事务时，SQL 语句的当前时间戳。悲观事务内发生写冲突时，会重试当前执行语句，该时间戳会被更新。重试次数由 max-retry-count 配置。事务模式为乐观事务时，该条目与 txnStartTS 等价。
  - isReadConsistency：当前事务隔离级别是否是读已提交 (RC)
  - current_db：当前数据库名
  - txn_mode：事务模式。可选值：OPTIMISTIC（乐观事务模式），或 PESSIMISTIC（悲观事务模式）
  - sql：当前查询对应的 SQL 语句

14.4.1.185 tidb_non_prepared_plan_cache_size

> 警告：
>
> 非 Prepare 语句执行计划缓存目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现 bug，请在 GitHub 上提 issue 反馈。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：100
- 范围：[1, 100000]
- 这个变量用来控制非 Prepare 语句执行计划缓存最多能够缓存的计划数量。

14.4.1.186 tidb_generate_binary_plan 从 v6.2.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 这个变量用于指定是否在 slow log 和 statement summary 里包含以二进制格式编码的执行计划。
- 开启该变量后，即可在 TiDB Dashboard 中查看查询的图形化执行计划。注意，TiDB Dashboard 只显示变量开启时产生的查询的执行计划。
- 用 select tidb_decode_binary_plan('xxx...') SQL 语句可以从编码后的执行计划解析出具体的执行计划。

14.4.1.187 tidb_gogc_tuner_threshold 从 v6.4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：0.6
- 范围：[0, 0.9)
- 这个变量用来控制 GOGC Tuner 自动调节的最大内存阈值，超过阈值后 GOGC Tuner 会停止工作。

14.4.1.188 tidb_guarantee_linearizability 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 此变量控制异步提交 (Async Commit) 中提交时间戳的计算方式。默认情况下（使用 ON 值），两阶段提交从 PD 服务器请求一个新的时间戳，并使用该时间戳计算最终提交的时间戳，这样可保证所有并发事务可线性化。
- 如果将该变量值设为 OFF，从 PD 获取时间戳的操作会被省掉，这种情况下只保证因果一致性但不保证线性一致性。详情请参考 PingCAP 博文 Async Commit 原理介绍。
- 对于需要只保证因果一致性的场景，可将此变量设为 OFF 以提升性能。

14.4.1.189 tidb_hash_exchange_with_new_collation

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 该值表示是否在开启 new collation 的集群里生成 MPP hash partition exchange 算子。true 表示生成此算子，false表示不生成。
- 该变量为 TiDB 内部变量，不推荐设置该变量。

14.4.1.190 tidb_hash_join_concurrency

> 警告：
>
> 从 v5.0 版本开始，该变量被废弃。请使用tidb_executor_concurrency 进行设置。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：-1
- 范围：[1, 256]
- 单位：线程
- 这个变量用来设置 hash join 算法的并发度。
- 默认值 -1 表示使用 tidb_executor_concurrency 的值。

14.4.1.191 tidb_hashagg_final_concurrency

> **警告：**
>
> 从 v5.0 版本开始，该变量被废弃。请使用`tidb_executor_concurrency`进行设置。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：-1
- 范围：[1, 256]
- 单位：线程
- 这个变量用来设置并行 hash aggregation 算法 final 阶段的执行并发度。对于聚合函数参数不为 distinct 的情况，HashAgg 分为 partial 和 final 阶段分别并行执行。
- 默认值 -1 表示使用 tidb_executor_concurrency 的值。

14.4.1.192 tidb_hashagg_partial_concurrency

> **警告：**
>
> 从 v5.0 版本开始，该变量被废弃。请使用`tidb_executor_concurrency`进行设置。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：-1
- 范围：[1, 256]
- 单位：线程
- 这个变量用来设置并行 hash aggregation 算法 partial 阶段的执行并发度。对于聚合函数参数不为 distinct 的情况，HashAgg 分为 partial 和 final 阶段分别并行执行。
- 默认值 -1 表示使用 tidb_executor_concurrency 的值。

14.4.1.193 tidb_historical_stats_duration 从 v6.6.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：Duration
- 默认值：168h，即 7 天
- 这个变量用来控制历史统计信息在存储中的保留时间。

14.4.1.194 tidb_ignore_prepared_cache_close_stmt 从 v6.0.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 这个变量用来设置是否忽略关闭 Prepared Statement 的指令。
- 如果变量值设为 ON，Binary 协议的 COM_STMT_CLOSE 信号和文本协议的DEALLOCATE PREPARE 语句都会被忽略。

14.4.1.195 tidb_index_join_batch_size

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：25000
- 范围：[1, 2147483647]
- 单位：行
- 这个变量用来设置 index lookup join 操作的 batch 大小，AP 类应用适合较大的值，TP 类应用适合较小的值。

14.4.1.196 tidb_index_join_double_read_penalty_cost_rate 从 v6.6.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：浮点数
- 默认值：0
- 范围：[0, 18446744073709551615]
- 这个变量用来设置是否给选择 index join 增加一些惩罚性的代价，以降低优化器选择 index join 操作的倾向，从而增加选择其他 join 方式的倾向，例如如选择 hash join 和 tiflash join 等。
- 优化器选择 index join 可能触发较多的回表请求，造成较多的资源开销，此时可以通过设置这个变量，来减少优化器选择 index join 的倾向。
- 这个变量只有在tidb_cost_model_version 设置为 2 时生效。

14.4.1.197 tidb_index_lookup_concurrency

> **警告：**
>
> 从 v5.0 版本开始，该变量被废弃。请使用tidb_executor_concurrency 进行设置。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：-1
- 范围：[1, 256]
- 单位：线程
- 这个变量用来设置 index lookup 操作的并发度，AP 类应用适合较大的值，TP 类应用适合较小的值。
- 默认值 -1 表示使用 tidb_executor_concurrency 的值。

14.4.1.198 tidb_index_lookup_join_concurrency

> **警告：**
>
> 从 v5.0 版本开始，该变量被废弃。请使用tidb_executor_concurrency 进行设置。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：-1
- 范围：[1, 256]
- 单位：线程
- 这个变量用来设置 index lookup join 算法的并发度。
- 默认值 -1 表示使用 tidb_executor_concurrency 的值。

14.4.1.199 tidb_index_merge_intersection_concurrency 从 v6.5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：-1
- 范围：[1, 256]
- 这个变量用来设置索引合并进行交集操作时的最大并发度，仅在以动态裁剪模式访问分区表时有效。实际并发度为 tidb_index_merge_intersection_concurrency 与分区表分区数目两者中较小的值。
- 默认值 -1 表示使用tidb_executor_concurrency 的值。

14.4.1.200 tidb_index_lookup_size

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：20000
- 范围：[1, 2147483647]
- 单位：行
- 这个变量用来设置 index lookup 操作的 batch 大小，AP 类应用适合较大的值，TP 类应用适合较小的值。

14.4.1.201 tidb_index_serial_scan_concurrency

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：1
- 范围：[1, 256]
- 单位：线程
- 这个变量用来设置顺序 scan 操作的并发度，AP 类应用适合较大的值，TP 类应用适合较小的值。

14.4.1.202 tidb_init_chunk_size

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：32
- 范围：[1, 32]
- 单位：行
- 这个变量用来设置执行过程中初始 chunk 的行数。默认值是 32，可设置的范围是 1 ~ 32。

14.4.1.203 tidb_isolation_read_engines 从 v4.0 版本开始引入

- 作用域：SESSION
- 默认值：tikv,tiflash,tidb
- 这个变量用于设置 TiDB 在读取数据时可以使用的存储引擎列表。

14.4.1.204 tidb_last_ddl_info 从 v6.0.0 版本开始引入

- 作用域：SESSION
- 默认值：“”
- 类型：字符串
- 该变量为只读变量，TiDB 内部使用该变量获取当前会话中上一个 DDL 操作的信息。
  - “query”：上一个 DDL 查询字符串。
  - “seq_num”：每个 DDL 操作的序列号，用于标识 DDL 操作的顺序。

14.4.1.205 tidb_last_query_info 从 v4.0.14 版本开始引入

- 作用域：SESSION
- 默认值：“”
- 这是一个只读变量。用于在 TiDB 内部查询上一条 DML 语句的事务信息。查询的事务信息包括：
  - txn_scope：事务的作用域，可能为 global 或 local。
  - start_ts：事务开始的时间戳。
  - for_update_ts：先前执行的 DML 语句的 for_update_ts 信息。这是 TiDB 用于测试的内部术语。通常，你可以忽略此信息。
  - error：错误消息（如果有）。

14.4.1.206 tidb_last_txn_info 从 v4.0.9 版本开始引入

- 作用域：SESSION
- 类型：字符串
- 此变量用于获取当前会话中最后一个事务的信息。这是一个只读变量。事务信息包括：
  - 事务的范围
  - 开始时间戳和提交时间戳
  - 事务的提交模式，可能是两阶段提交，一阶段提交，或者异步提交
  - 事务从异步提交或一阶段提交到两阶段提交的回退信息
  - 遇到的错误

14.4.1.207 tidb_last_plan_replayer_token 从 v6.3.0 版本开始引入

- 作用域：SESSION
- 类型：字符串
- 这个变量是一个只读变量，用于获取当前会话中最后一个 PLAN REPLAYER dump 的结果。

14.4.1.208 tidb_load_based_replica_read_threshold 从 v7.0.0 版本开始引入

> **警告：**
>
> 当前版本中该变量控制的功能尚未完全生效，请保留默认值。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值："0s"
- 范围：[0s, 1h]
- 类型：字符串
- 这个变量用来设置基于负载的 replica read 的触发阈值。当 leader 节点的预估排队时间超过阈值时，TiDB 会优先从 follower 节点读取数据。格式为时间，例如 "100ms" 或 "1s"。

14.4.1.209 tidb_log_file_max_days 从 v5.3.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：否，仅作用于当前连接的 TiDB 实例
- 默认值：0
- 范围：[0, 2147483647]
- 这个变量可以调整当前 TiDB 实例上日志的最大保留天数。默认值是实例配置文件中指定的值，见配置项max-days。此变量只影响当前 TiDB 实例上的配置，重启后丢失，且配置文件不受影响。

14.4.1.210 tidb_low_resolution_tso

- 作用域：SESSION
- 默认值：OFF
- 这个变量用来设置是否启用低精度 TSO 特性。开启该功能之后，新事务会使用一个每 2s 更新一次的 TS 来读取数据。
- 主要场景是在可以容忍读到旧数据的情况下，降低小的只读事务获取 TSO 的开销。

14.4.1.211 tidb_nontransactional_ignore_error 从 v6.1.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是

- 类型：布尔型
- 默认值：OFF
- 这个变量用来设置是否在非事务语句中立刻返回错误。当设为 OFF 时，在碰到第一个报错的 batch 时，非事务 DML 语句即中止，取消其后的所有 batch，返回错误。当设为 ON 时，当某个 batch 执行报错时，其后的 batch 会继续执行，直到所有 batch 执行完毕，返回结果时把这些错误合并后返回。

14.4.1.212 tidb_max_auto_analyze_time 从 v6.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：43200
- 范围：[0, 2147483647]
- 单位：秒
- 这个变量用于指定自动 ANALYZE 的最大执行时间。当执行时间超出指定的时间时，自动 ANALYZE 会被终止。当该变量值为 0 时，自动 ANALYZE 没有最大执行时间的限制。

14.4.1.213 tidb_max_bytes_before_tiflash_external_group_by 从 v7.0.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：-1
- 范围：[-1, 9223372036854775807]
- 这个变量用于指定 TiFlash 中带有 GROUP BY 的 Hash Aggregation 算子的最大内存使用量，单位为 byte，超过该值之后 TiFlash 会触发 Hash Aggregation 算子的落盘。当该变量值为 -1 时，TiDB 不传递该变量给 TiFlash。只有该变量值大于等于 0 时，TiDB 才会传递该变量给 TiFlash。该变量为 0 时表示内存使用无限制，即 TiFlash Hash Aggregation 算子不会触发落盘。详情见TiFlash 数据落盘。

> 注意：
>
> - 假设一个 TiDB 集群有多个 TiFlash 节点，Aggregation 通常会在多个 TiFlash 节点上分布式执行。该变量控制的是单个 TiFlash 节点中 Aggregation 算子的最大内存使用量。
> - 当该变量设置为 -1 时，TiFlash 将根据自身配置项max_bytes_before_external_group_by 的值来决定 Aggregation 算子的最大内存使用量。

14.4.1.214 tidb_max_bytes_before_tiflash_external_join 从 v7.0.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型

- 默认值：-1
- 范围：[-1，9223372036854775807]
- 这个变量用于指定 TiFlash 中带等值关联条件的 Hash Join 算子的最大内存使用量，单位为 byte，超过该值之后 TiFlash 会触发 Hash Join 算子的落盘。当该变量值为 -1 时，TiDB 不传递该变量给 TiFlash。只有该变量值大于等于 0 时，TiDB 才会传递该变量给 TiFlash。该变量为 0 时表示内存使用无限制，即 TiFlash Hash Join 算子不会触发落盘。详情见TiFlash 数据落盘。

> 注意：
>
> - 假设一个 TiDB 集群有多个 TiFlash 节点，Join 通常会在多个 TiFlash 节点上分布式执行。该变量控制的是单个 TiFlash 节点中 Join 算子的最大内存使用量。
> - 当该变量设置为 -1 时，TiFlash 将根据自身配置项`max_bytes_before_external_join` 的值来决定 Join 算子的最大内存使用量。

14.4.1.215 tidb_max_bytes_before_tiflash_external_sort 从 v7.0.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：-1
- 范围：[-1，9223372036854775807]
- 这个变量用于指定 TiFlash 中带 topN 和 sort 算子的最大内存使用量，单位为 byte，超过该值之后 TiFlash 会触发 topN 和 sort 算子的落盘。当该变量值为 -1 时，TiDB 不传递该变量给 TiFlash。只有该变量值大于等于 0 时，TiDB 才会传递该变量给 TiFlash。该变量为 0 时表示内存使用无限制，即 TiFlash topN 和 sort 算子不会触发落盘。详情见TiFlash 数据落盘。

> 注意：
>
> - 假设一个 TiDB 集群有多个 TiFlash 节点，TopN 和 Sort 通常会在多个 TiFlash 节点中分布式执行。该变量控制的是单个 TiFlash 节点中 TopN 和 Sort 算子的最大内存使用量。
> - 当该变量设置为 -1 时，TiFlash 将根据自身配置项`max_bytes_before_external_sort` 的值来决定 TopN 和 Sort 算子的最大内存使用量。

14.4.1.216 tidb_max_chunk_size

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：1024
- 范围：[32，2147483647]

- 单位：行
- 这个变量用来设置执行过程中一个 chunk 最大的行数，设置过大可能引起缓存局部性的问题。

14.4.1.217 tidb_max_delta_schema_count

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：1024
- 范围：[100, 16384]
- 这个变量用来设置缓存 schema 版本信息（对应版本修改的相关 table IDs）的个数限制，可设置的范围 100 - 16384。此变量在 2.1.18 及之后版本支持。

14.4.1.218 tidb_max_paging_size 从 v6.3.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：50000
- 范围：[1, 9223372036854775807]
- 单位：行
- 这个变量用来设置 coprocessor 协议中 paging size 的最大的行数。请合理设置该值，设置过小，TiDB 与 TiKV 的 RPC 交互会更频繁；设置过大，导数据和全表扫等特定场景会占用更多内存。该变量的默认值对于 OLTP 场景较友好，如果业务只使用了 TiKV 作为存储引擎，当执行偏 OLAP 的负载时，可以考虑将变量值调大，有可能获得更好的性能。

14.4.1.219 tidb_max_tiflash_threads 从 v6.1.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：-1
- 范围：[-1, 256]
- 单位：线程
- TiFlash 中 request 执行的最大并发度。默认值为 -1，表示该系统变量无效。0 表示由 TiFlash 系统自动设置该值。

14.4.1.220 tidb_mem_oom_action 从 v6.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：CANCEL
- 可选值：CANCEL，LOG
- 该变量控制当单个查询使用的内存超过限制 (tidb_mem_quota_query) 且不能再利用临时磁盘时，TiDB 所采取的操作。详情见TiDB 内存控制。
- 该变量默认值为 CANCEL，但在 TiDB v4.0.2 及之前的版本中，默认值为 LOG。
- 在 v6.1.0 之前这个开关通过 TiDB 配置文件 (oom-action) 进行配置，升级到 v6.1.0 时会自动继承原有设置。

14.4.1.221 tidb_mem_quota_analyze 从 v6.1.0 版本开始引入

> **警告：**
>
> 目前限制 ANALYZE 的内存使用量为实验特性，在生产环境中使用时可能存在内存统计有误差的情况。

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：-1
- 单位：字节
- 取值范围：[-1, 9223372036854775807]
- 这个变量用来控制 TiDB 更新统计信息时的最大总内存占用，包括用户执行的ANALYZE TABLE 和 TiDB 后台自动执行的统计信息更新任务。当总的内存占用超过这个阈值时，用户执行的 ANALYZE 会被终止退出，并通过错误信息提示用户尝试更小的采样率或稍后重试。如果 TiDB 后台自动执行的统计信息更新任务因内存超限而退出，且使用的采样率高于默认值，则会使用默认采样率重试一次。当该变量值为负数或零时，TiDB 不对更新统计信息的前后台任务进行内存限制。

> **注意：**
>
> 只有在 TiDB 的启动配置文件中开启了 run-auto-analyze 选项，该 TiDB 集群才会触发 auto_analyze。

14.4.1.222 tidb_mem_quota_apply_cache 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：33554432 (32 MiB)
- 范围：[0, 9223372036854775807]
- 单位：字节
- 这个变量用来设置 Apply 算子中局部 Cache 的内存使用阈值。
- Apply 算子中局部 Cache 用来加速 Apply 算子的计算，该变量可以设置 Apply Cache 的内存使用阈值。设置变量值为 0 可以关闭 Apply Cache 功能。

14.4.1.223 tidb_mem_quota_binding_cache 从 v6.0.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：67108864 (64 MiB)

- 范围：[0, 2147483647]
- 单位：字节
- 这个变量用来设置存放 binding 的缓存的内存使用阈值。
- 如果一个系统创建或者捕获了过多的绑定，导致绑定所使用的内存空间超过该阈值，TiDB 会在日志中增加警告日志进行提示。这种情况下，缓存无法存放所有可用的绑定，并且无法保证哪些绑定存在于缓存中，因此，可能存在一些查询无法使用可用绑定的情况。此时，可以调大该变量的值，从而保证所有可用绑定都能正常使用。修改变量值以后，需要执行命令 admin reload bindings 重新加载绑定，确保变更生效。

14.4.1.224 tidb_mem_quota_query

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：1073741824 (1 GiB)
- 范围：[-1, 9223372036854775807]
- 单位：字节
- 在 v6.1.0 之前的版本中，作用域为 SESSION。v6.1.0 及之后的版本，作用域变更为 SESSION | GLOBAL。
- 在 v6.5.0 之前的版本中，该变量用来设置单条查询的内存使用限制，如果单条查询执行过程中使用的内存量超过该阈值，会触发系统变量tidb_mem_oom_action 中指定的行为。
- 在 v6.5.0 及之后的版本中，该变量用来设置单个会话整体的内存使用限制，如果某个会话执行过程中使用的内存量超过该阈值，会触发系统变量tidb_mem_oom_action 中指定的行为。需要注意的是，自 v6.5.0 版本开始，会话的内存使用量包含会话中事务所消耗的内存。v6.5.0 及之后版本对事务内存的控制行为有所变化，详见txn-total-size-limit。
- 当变量值为 0 或 -1 时，表示内存阈值为正无穷。此外，当变量值小于 128 时，将默认被设置为 128。

14.4.1.225 tidb_memory_debug_mode_alarm_ratio

- 作用域：SESSION
- 类型：浮点型
- 默认值：0
- 该变量表示在 TiDB memory debug 模式下，允许的内存统计误差值。
- 该变量用于 TiDB 内部测试，不推荐修改该变量值。

14.4.1.226 tidb_memory_debug_mode_min_heap_inuse

- 作用域：SESSION
- 类型：整数型
- 默认值：0
- 该变量用于 TiDB 内部测试，不推荐修改该变量值，因为开启后会影响 TiDB 的性能。
- 配置此参数后，TiDB 会进入 memory debug 模式进行内存追踪准确度的分析。TiDB 会在后续执行 SQL 语句的过程中频繁触发 GC，并将实际内存使用和内存统计值做对比。若当前内存使用大于 tidb_memory_debug_mode_min_heap_inuse 且内存统计误差超过 tidb_memory_debug_mode_alarm_ratio，则会输出相关内存信息到日志和文件中。

14.4.1.227 tidb_memory_usage_alarm_ratio

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：0.7
- 范围：[0.0, 1.0]
- 这个变量用于设置触发 tidb-server 内存告警的内存使用比率。默认情况下，当 TiDB 内存使用量超过总内存的 70% 且满足报警条件时，TiDB 会打印报警日志。
- 当配置该变量的值为 0 或 1 时，表示关闭内存阈值报警功能。
- 当配置该变量为 0 到 1 之间的值时，表示开启内存阈值报警功能：
  - 如果系统变量tidb_server_memory_limit 等于 0,则内存报警阈值为 tidb_memory_usage_alarm_ratio * 系统内存大小。
  - 如果系统变量 tidb_server_memory_limit 被设置为大于 0,则内存报警阈值为 tidb_memory_usage_alarm_ratio * tidb_server_memory_limit。

14.4.1.228 tidb_memory_usage_alarm_keep_record_num 从 v6.4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：5
- 范围：[1, 10000]
- 当 tidb-server 内存占用超过内存报警阈值并触发报警时，TiDB 默认只保留最近 5 次报警时所生成的状态文件。你可以通过该变量调整该次数。

14.4.1.229 tidb_merge_join_concurrency

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：1
- 取值范围：[1, 256]
- 设置 MergeJoin 算子执行查询时的并发度。
- 不推荐设置该变量，修改该变量值可能会造成数据正确性问题。

14.4.1.230 tidb_merge_partition_stats_concurrency

> **警告：**
>
> 当前版本中该变量控制的功能尚未完全生效，请保留默认值。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：1
- 这个变量用于 TiDB analyze 分区表时，对分区表统计信息进行合并时的并发度。

14.4.1.231 tidb_metric_query_range_duration 从 v4.0 版本开始引入

- 作用域：SESSION
- 默认值：60
- 范围：[10, 216000]
- 单位：秒
- 这个变量设置了查询 METRIC_SCHEMA 时生成的 Prometheus 语句的 range duration。

14.4.1.232 tidb_metric_query_step 从 v4.0 版本开始引入

- 作用域：SESSION
- 默认值：60
- 范围：[10, 216000]
- 单位：秒
- 这个变量设置了查询 METRIC_SCHEMA 时生成的 Prometheus 语句的 step。

14.4.1.233 tidb_min_paging_size 从 v6.2.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：128
- 范围：[1, 9223372036854775807]
- 单位：行
- 这个变量用来设置 coprocessor 协议中 paging size 的最小的行数。请合理设置该值，设置过小，TiDB 与 TiKV 的 RPC 交互会更频繁；设置过大，IndexLookup 带 Limit 场景会出现性能下降。该变量的默认值对于 OLTP 场景较友好，如果业务只使用了 TiKV 作为存储引擎，当执行偏 OLAP 的负载时，可以考虑将变量值调大，有可能获得更好的性能。

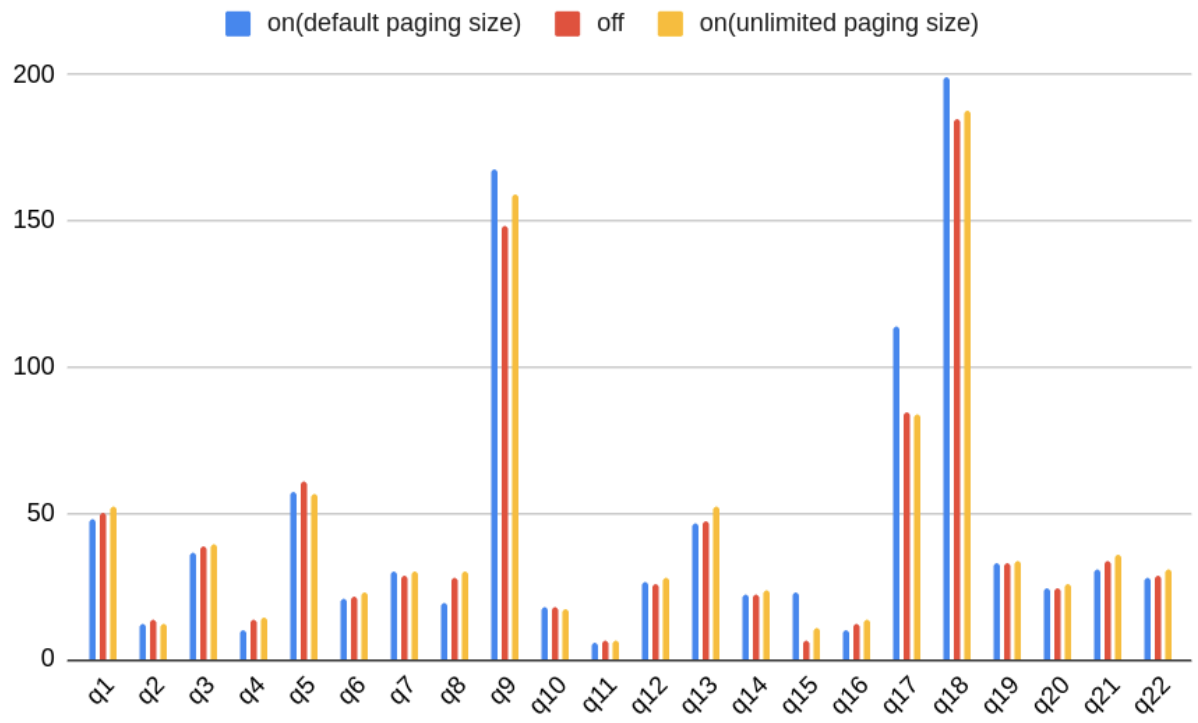


图 323: Paging size impact on TPCH

开启tidb_enable_paging 时，tidb_min_paging_size 和tidb_max_paging_size 对 TPCH 的性能影响如上图所示，纵轴是执行时间，越小越好。

14.4.1.234 tidb_mpp_store_fail_ttl

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：Duration
- 默认值：60s
- 刚重启的 TiFlash 可能不能正常提供服务。为了防止查询失败，TiDB 会限制 tidb-server 向刚重启的 TiFlash 节点发送查询。这个变量表示刚重启的 TiFlash 不被发送请求的时间范围。

14.4.1.235 tidb_multi_statement_mode 从 v4.0.11 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 可选值：OFF，ON，WARN
- 该变量用于控制是否在同一个 COM_QUERY 调用中执行多个查询。
- 为了减少 SQL 注入攻击的影响，TiDB 目前默认不允许在同一 COM_QUERY 调用中执行多个查询。该变量可用作早期 TiDB 版本的升级路径选项。该变量值与是否允许多语句行为的对照表如下：

客户端设置	tidb_multi_statement_mode 值	是否允许多语句
Multiple Statements = ON	OFF	允许
Multiple Statements = ON	ON	允许
Multiple Statements = ON	WARN	允许
Multiple Statements = OFF	OFF	不允许
Multiple Statements = OFF	ON	允许
Multiple Statements = OFF	WARN	允许 + 警告提示

注意：

只有默认值 OFF 才是安全的。如果用户业务是专为早期 TiDB 版本而设计的，那么需要将该变量值设为 ON。如果用户业务需要多语句支持，建议用户使用客户端提供的设置，不要使用 tidb_multi_statement_mode 变量进行设置。

- go-sql-driver (multiStatements)
- Connector/J (allowMultiQueries)
- PHP mysqli (mysqli_multi_query)

14.4.1.236 tidb_opt_agg_push_down

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 这个变量用来设置优化器是否执行聚合函数下推到 Join，Projection 和 UnionAll 之前的优化操作。当查询中聚合操作执行很慢时，可以尝试设置该变量为 ON。

14.4.1.237 tidb_opt_broadcast_cartesian_join

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：1
- 范围：[0, 2]
- 表示是否允许 Broadcast Cartesian Join 算法。
- 值为 0 时表示不允许使用 Broadcast Cartesian Join 算法。值为 1 时表示根据tidb_broadcast_join_threshold_count ↪ 的行数阈值确定是否允许使用 Broadcast Cartesian Join 算法。值为 2 时表示总是允许 Broadcast Cartesian Join 算法，即使表的大小超过了该阈值。
- 该变量是 TiDB 内部使用的变量，不推荐修改该变量的值。

14.4.1.238 tidb_opt_concurrency_factor

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：浮点数
- 范围：[0, 18446744073709551615]
- 默认值：3.0
- 表示在 TiDB 中开启一个 Golang goroutine 的 CPU 开销。该变量是代价模型内部使用的变量，不建议修改该变量的值。

14.4.1.239 tidb_opt_copcpu_factor

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：浮点数
- 范围：[0, 18446744073709551615]
- 默认值：3.0
- 表示 TiKV 协处理器处理一行数据的 CPU 开销。该变量是代价模型内部使用的变量，不建议修改该变量的值。

14.4.1.240 tidb_opt_correlation_exp_factor

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：1
- 范围：[0, 2147483647]
- 当交叉估算方法不可用时，会采用启发式估算方法。这个变量用来控制启发式方法的行为。当值为 0 时不用启发式估算方法，大于 0 时，该变量值越大，启发式估算方法越倾向 index scan，越小越倾向 table scan。

14.4.1.241 tidb_opt_correlation_threshold

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：0.9
- 这个变量用来设置优化器启用交叉估算 row count 方法的阈值。如果列和 handle 列之间的顺序相关性超过这个阈值，就会启用交叉估算方法。
- 交叉估算方法可以简单理解为，利用这个列的直方图来估算 handle 列需要扫的行数。

14.4.1.242 tidb_opt_cpu_factor

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是

- 类型：浮点数
- 范围：[0, 2147483647]
- 默认值：3.0
- 表示 TiDB 处理一行数据的 CPU 开销。该变量是代价模型内部使用的变量，不建议修改该变量的值。

14.4.1.243 tidb_opt_derive_topn 从 v7.0.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 表示是否开启从窗口函数中推导 TopN 或 Limit 的优化规则。

14.4.1.244 tidb_opt_desc_factor

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：浮点数
- 范围：[0, 18446744073709551615]
- 默认值：3.0
- 表示降序扫描时，TiKV 在磁盘上扫描一行数据的开销。该变量是代价模型内部使用的变量，不建议修改该变量的值。

14.4.1.245 tidb_opt_disk_factor

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：浮点数
- 范围：[0, 18446744073709551615]
- 默认值：1.5
- 表示 TiDB 往临时磁盘读写一个字节数据的 I/O 开销。该变量是代价模型内部使用的变量，不建议修改该变量的值。

14.4.1.246 tidb_opt_distinct_agg_push_down

- 作用域：SESSION
- 默认值：OFF
- 这个变量用来设置优化器是否执行带有 Distinct 的聚合函数（比如 select count(distinct a)from t）下推到 Coprocessor 的优化操作。当查询中带有 Distinct 的聚合操作执行很慢时，可以尝试设置该变量为 1。

在以下示例中，tidb_opt_distinct_agg_push_down 开启前，TiDB 需要从 TiKV 读取所有数据，并在 TiDB 侧执行 distinct。tidb_opt_distinct_agg_push_down 开启后，distinct a 被下推到了 Coprocessor，在 HashAgg_5 里新增里一个 group by 列 test.t.a。

```
mysql> desc select count(distinct a) from test.t;
+--
    ↪ ----------------------+----------+-----------+--------------+--------------------------------
    ↪
| id                      | estRows  | task      | access object | operator info
    ↪                            |
+--
    ↪ ----------------------+----------+-----------+--------------+--------------------------------
    ↪
| StreamAgg_6             | 1.00     | root      |               | funcs:count(distinct test.t.a)
    ↪ ->Column#4 |
| └─TableReader_10        | 10000.00 | root      |               | data:TableFullScan_9
    ↪                       |
|   └─TableFullScan_9     | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo           |
+--
    ↪ ----------------------+----------+-----------+--------------+--------------------------------
    ↪
3 rows in set (0.01 sec)

mysql> set session tidb_opt_distinct_agg_push_down = 1;
Query OK, 0 rows affected (0.00 sec)

mysql> desc select count(distinct a) from test.t;
+--
    ↪ ------------------------+----------+-----------+--------------+--------------------------------
    ↪
| id                        | estRows  | task      | access object | operator info
    ↪                              |
+--
    ↪ ------------------------+----------+-----------+--------------+--------------------------------
    ↪
| HashAgg_8                 | 1.00     | root      |               | funcs:count(distinct test.t.
    ↪ a)->Column#3 |
| └─TableReader_9           | 1.00     | root      |               | data:HashAgg_5
    ↪                             |
|   └─HashAgg_5             | 1.00     | cop[tikv] |               | group by:test.t.a,
    ↪                          |
|     └─TableFullScan_7     | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo           |
+--
    ↪ ------------------------+----------+-----------+--------------+--------------------------------
    ↪
4 rows in set (0.00 sec)
```

14.4.1.247 tidb_opt_enable_correlation_adjustment

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用来控制优化器是否开启交叉估算。

14.4.1.248 tidb_opt_enable_late_materialization 从 v7.0.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 这个变量用来控制是否启用TiFlash 延迟物化功能。
- 默认情况下，如果 SELECT 语句中包含过滤条件（WHERE 子句），TiFlash 会先扫描查询所需列的全部数据后再进行过滤。当设置该变量为 ON 开启 TiFlash 延迟物化功能时，TiFlash 可以先扫描过滤条件相关的列数据，过滤得到符合条件的行后，再扫描这些行的其他列数据，继续后续计算，从而减少 IO 扫描和数据处理的计算量。

14.4.1.249 tidb_opt_force_inline_cte 从 v6.3.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 这个变量用来控制是否强制 inline CTE。默认值为 OFF，即默认不强制 inline CTE。注意，此时依旧可以通过 MERGE() hint 来开启个别 CTE 的 inline。如果设置为 ON，则当前 session 中所有查询的 CTE（递归 CTE 除外）都会 inline。

14.4.1.250 tidb_opt_advanced_join_hint 从 v7.0.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 这个变量用来控制包括HASH_JOIN() Hint、MERGE_JOIN() Hint 等用于控制连接算法的 Join Method Hint 是否会影响 Join Reorder 的优化过程，包括LEADING() Hint 的使用。默认值为 ON，即默认不影响。如果设置为 OFF，在一些同时使用 Join Method Hint 和 LEADING() Hint 的场景下可能会产生冲突。

> 注意：
>
> v7.0.0 之前的版本行为和将该变量设置为 OFF 的行为一致。为确保向前兼容，从旧版本升级到 v7.0.0 及之后版本的集群，该变量会被设置成 OFF。为了获取更灵活的 Hint 行为，强烈建议在确保无性能回退的情况下，将该变量切换为 ON。

14.4.1.251 tidb_opt_insubq_to_join_and_agg

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用来设置是否开启优化规则：将子查询转成 join 和 aggregation。

  例如，打开这个优化规则后，会将下面子查询做如下变化：

```
select * from t where t.a in (select aa from t1);
```

  将子查询转成如下 join：

```
select t.* from t, (select aa from t1 group by aa) tmp_t where t.a = tmp_t.aa;
```

  如果 t1 在列 aa 上有 unique 且 not null 的限制，可以直接改写为如下，不需要添加 aggregation。

```
select t.* from t, t1 where t.a=t1.aa;
```

14.4.1.252 tidb_opt_join_reorder_threshold

- 作用域: SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：0
- 范围：[0, 2147483647]
- 这个变量用来控制 TiDB Join Reorder 算法的选择。当参与 Join Reorder 的节点个数大于该阈值时，TiDB 选择贪心算法，小于该阈值时 TiDB 选择动态规划 (dynamic programming) 算法。
- 目前对于 OLTP 的查询，推荐保持默认值。对于 OLAP 的查询，推荐将变量值设为 10~15 来获得 AP 场景下更好的连接顺序。

14.4.1.253 tidb_opt_limit_push_down_threshold

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：100
- 范围：[0, 2147483647]
- 这个变量用来设置将 Limit 和 TopN 算子下推到 TiKV 的阈值。
- 如果 Limit 或者 TopN 的取值小于等于这个阈值，则 Limit 和 TopN 算子会被强制下推到 TiKV。该变量可以解决部分由于估算误差导致 Limit 或者 TopN 无法被下推的问题。

14.4.1.254 tidb_opt_memory_factor

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：浮点数
- 范围：[0, 2147483647]
- 默认值：0.001
- 表示 TiDB 存储一行数据的内存开销。该变量是代价模型内部使用的变量，不建议修改该变量的值。

14.4.1.255 tidb_opt_mpp_outer_join_fixed_build_side 从 v5.1.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 当该变量值为 ON 时，左连接始终使用内表作为构建端，右连接始终使用外表作为构建端。将该变量值设为 OFF 后，外连接可以灵活选择任意一边表作为构建端。

14.4.1.256 tidb_opt_network_factor

- 作用域：SESSION | GLOBAL
- 是否持久化
- 类型：浮点数
- 范围：[0, 2147483647]
- 默认值：1.0
- 表示传输 1 比特数据的网络净开销。该变量是代价模型内部使用的变量，不建议修改该变量的值。

14.4.1.257 tidb_opt_ordering_index_selectivity_threshold 从 v7.0.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：浮点数
- 默认值：0
- 范围：[0, 1]
- 用于当 SQL 中存在 ORDER BY 和 LIMIT 子句且带有过滤条件时，控制优化器选择索引的行为。
- 对于此类查询，优化器会考虑选择对应的索引来满足 ORDER BY 和 LIMIT 子句（即使这个索引并不满足任何过滤条件）。但是由于数据分布的复杂性，优化器在这种场景下可能会选择不优的索引。
- 该变量表示一个阈值。当存在索引能满足过滤条件，且其选择率估算值低于该阈值时，优化器会避免选择用于满足 ORDER BY 和 LIMIT 的索引，而优先选择用于满足过滤条件的索引。
- 例如，当把该变量设为 0 时，优化器保持默认行为；当设为 1 时，优化器总是优先选择满足过滤条件的索引，避免选择满足 ORDER BY 和 LIMIT 的索引。
- 在以下示例中，t 表共有 1,000,000 行数据。使用 b 列上的索引时，其估算行数是大约 8,748 行，因此其选择率估算值大约是 0.0087。默认情况下，优化器选择了 a 列上的索引。而将该变量设为 0.01 之后，由于 b 列上的索引的选择率 (0.0087) 低于 0.01，优化器选择了 b 列上的索引。

```
> EXPLAIN SELECT * FROM t WHERE b <= 9000 ORDER BY a LIMIT 1;
+--
    ↪ ---------------------------------+---------+-----------+---------------------+-----------------
    ↪
| id                                | estRows | task      | access object       | operator info
    ↪      |
+--
    ↪ ---------------------------------+---------+-----------+---------------------+-----------------
    ↪
| Limit_12                          | 1.00    | root      |                     | offset:0,
    ↪ count:1  |
| └─Projection_25                    | 1.00    | root      |                      | test.t.a,
    ↪ test.t.b |
|   └─IndexLookUp_24                 | 1.00    | root      |                      |
    ↪                      |
|     ├─IndexFullScan_21(Build)      | 114.30  | cop[tikv] | table:t, index:ia(a) | keep order:
    ↪ true    |
|     └─Selection_23(Probe)          | 1.00    | cop[tikv] |                      | le(test.t.b,
    ↪ 9000) |
|       └─TableRowIDScan_22          | 114.30  | cop[tikv] | table:t              | keep order:
    ↪ false   |
+--
    ↪ ---------------------------------+---------+-----------+---------------------+-----------------
    ↪

> SET SESSION tidb_opt_ordering_index_selectivity_threshold = 0.01;

> EXPLAIN SELECT * FROM t WHERE b <= 9000 ORDER BY a LIMIT 1;
+--
    ↪ ---------------------------------+---------+-----------+---------------------+-----------------
    ↪
| id                                | estRows | task      | access object       | operator info
    ↪                      |
+--
    ↪ ---------------------------------+---------+-----------+---------------------+-----------------
    ↪
| TopN_9                            | 1.00    | root      |                     | test.t.a,
    ↪ offset:0, count:1         |
| └─IndexLookUp_20                   | 1.00    | root      |                      |
    ↪                                  |
|   ├─IndexRangeScan_17(Build)       | 8748.62 | cop[tikv] | table:t, index:ib(b) | range:[-inf
    ↪ ,9000], keep order:false |
|   └─TopN_19(Probe)                 | 1.00    | cop[tikv] |                      | test.t.a,
    ↪ offset:0, count:1         |
```

```
|     └─TableRowIDScan_18          | 8748.62 | cop[tikv] | table:t              | keep order:
    ↪ false                    |
+--
    ↪ ---------------------------------+---------+-----------+----------------------+-------------------------
    ↪
```

14.4.1.258 tidb_opt_prefer_range_scan 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 将该变量值设为 ON 后，优化器总是偏好区间扫描而不是全表扫描。
- 在以下示例中，tidb_opt_prefer_range_scan 开启前，TiDB 优化器需要执行全表扫描。tidb_opt_prefer_range_scan ↪ 开启后，优化器选择了索引区间扫描。

```
explain select * from t where age=5;
+-------------------------+------------+-----------+---------------+-------------------+
| id                      | estRows    | task      | access object | operator info     |
+-------------------------+------------+-----------+---------------+-------------------+
| TableReader_7           | 1048576.00 | root      |               | data:Selection_6  |
| └─Selection_6           | 1048576.00 | cop[tikv] |               | eq(test.t.age, 5) |
|   └─TableFullScan_5     | 1048576.00 | cop[tikv] | table:t       | keep order:false  |
+-------------------------+------------+-----------+---------------+-------------------+
3 rows in set (0.00 sec)

set session tidb_opt_prefer_range_scan = 1;

explain select * from t where age=5;
+--
    ↪ ----------------------------+------------+-----------+----------------------------+-------------------
    ↪
| id                           | estRows    | task      | access object              | operator
    ↪  info                 |
+--
    ↪ ----------------------------+------------+-----------+----------------------------+-------------------
    ↪
| IndexLookUp_7                | 1048576.00 | root      |                            |
    ↪                                |
| ├─IndexRangeScan_5(Build)     | 1048576.00 | cop[tikv] | table:t, index:idx_age(age) | range
    ↪ :[5,5], keep order:false |
| └─TableRowIDScan_6(Probe)     | 1048576.00 | cop[tikv] | table:t                    | keep
    ↪ order:false              |
+--
    ↪ ----------------------------+------------+-----------+----------------------------+-------------------
    ↪
```

```
3 rows in set (0.00 sec)
```

14.4.1.259 tidb_opt_projection_push_down 从 v6.1.0 版本开始引入

- 作用域：SESSION
- 类型：布尔型
- 默认值：OFF
- 指定是否允许优化器将 Projection 算子下推到 TiKV 或者 TiFlash。

14.4.1.260 tidb_opt_range_max_size 从 v6.4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：67108864 (64 MiB)
- 取值范围：[0, 9223372036854775807]
- 单位：字节
- 该变量用于指定优化器构造扫描范围的内存用量上限。当该变量为 0 时，表示对扫描范围没有内存限制。如果构造精确的扫描范围会超出内存用量限制，优化器会使用更宽松的扫描范围（例如 [[NULL,+inf]] ）。如果执行计划中未使用精确的扫描范围，可以调大该变量的值让优化器构造精确的扫描范围。

该变量的使用示例如下：

tidb_opt_range_max_size 使用示例

查看该变量的默认值，即优化器构造扫描范围最多使用 64 MiB 内存。

```
SELECT @@tidb_opt_range_max_size;
```

```
+----------------------------+
| @@tidb_opt_range_max_size |
+----------------------------+
| 67108864                   |
+----------------------------+
1 row in set (0.01 sec)
```

```
EXPLAIN SELECT * FROM t use index (idx) WHERE a IN (10,20,30) AND b IN (40,50,60);
```

在 64 MiB 的内存最大限制约束下，优化器构造出精确的扫描范围 [10 40,10 40], [10 50,10 50], [10 60,10 ↪ 60], [20 40,20 40], [20 50,20 50], [20 60,20 60], [30 40,30 40], [30 50,30 50], [30 60,30 60]，见如下执行计划返回结果。

```
+--
    ↪ ----------------------------+---------+-----------+-------------------------+-------------------------
    ↪
```

```
| id                            | estRows | task      | access object            | operator info
    ↪
    ↪ |
+--
    ↪ ----------------------------+---------+-----------+--------------------------+----------------------------
    ↪
| IndexLookUp_7                 | 0.90    | root      |                          |
    ↪
    ↪ |
| ├─IndexRangeScan_5(Build)     | 0.90    | cop[tikv] | table:t, index:idx(a, b) | range:[10
    ↪ 40,10 40], [10 50,10 50], [10 60,10 60], [20 40,20 40], [20 50,20 50], [20 60,20 60], [30
    ↪  40,30 40], [30 50,30 50], [30 60,30 60], keep order:false, stats:pseudo |
| └─TableRowIDScan_6(Probe)     | 0.90    | cop[tikv] | table:t                  | keep order:
    ↪ false, stats:pseudo
    ↪
    ↪ |
+--
    ↪ ----------------------------+---------+-----------+--------------------------+----------------------------
    ↪
3 rows in set (0.00 sec)
```

现将优化器构造扫描范围的内存用量上限设为 1500 字节。

```
SET @@tidb_opt_range_max_size = 1500;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
EXPLAIN SELECT * FROM t USE INDEX (idx) WHERE a IN (10,20,30) AND b IN (40,50,60);
```

在 1500 字节内存的最大限制约束下，优化器构造出了更宽松的扫描范围 `[10,10], [20,20], [30,30]`，并用 warning 提示用户构造精确的扫描范围所需的内存用量超出了 `tidb_opt_range_max_size` 的限制。

```
+--
    ↪ ----------------------------+---------+-----------+--------------------------+----------------------------
    ↪
| id                            | estRows | task      | access object            | operator info
    ↪                                          |
+--
    ↪ ----------------------------+---------+-----------+--------------------------+----------------------------
    ↪
| IndexLookUp_8                 | 0.09    | root      |                          |
    ↪                                                    |
| ├─Selection_7(Build)          | 0.09    | cop[tikv] |                          | in(test.t.b,
    ↪ 40, 50, 60)                                |
| │ └─IndexRangeScan_5          | 30.00   | cop[tikv] | table:t, index:idx(a, b) | range
    ↪ :[10,10], [20,20], [30,30], keep order:false, stats:pseudo |
```

```
| └─TableRowIDScan_6(Probe)     | 0.09    | cop[tikv] | table:t                  | keep order:
    ↪ false, stats:pseudo                                  |
+--
    ↪ ----------------------------+---------+-----------+--------------------------+----------------------
    ↪
4 rows in set, 1 warning (0.00 sec)
```

```
SHOW WARNINGS;
```

```
+--
    ↪ -------+------+-----------------------------------------------------------------------------------
    ↪
| Level   | Code | Message
    ↪
    ↪ |
+--
    ↪ -------+------+-----------------------------------------------------------------------------------
    ↪
| Warning | 1105 | Memory capacity of 1500 bytes for 'tidb_opt_range_max_size' exceeded when
    ↪ building ranges. Less accurate ranges such as full range are chosen |
+--
    ↪ -------+------+-----------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

再将优化器构造扫描范围的内存用量上限设为 100 字节。

```
set @@tidb_opt_range_max_size = 100;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
EXPLAIN SELECT * FROM t USE INDEX (idx) WHERE a IN (10,20,30) AND b IN (40,50,60);
```

在 100 字节的内存最大限制约束下，优化器选择了 IndexFullScan，并用 warning 提示用户构造精确的扫描范围所需的内存超出了 tidb_opt_range_max_size 的限制。

```
+--
    ↪ ----------------------------+----------+-----------+--------------------------+----------------------
    ↪
| id                            | estRows  | task      | access object            | operator info
    ↪                                        |
+--
    ↪ ----------------------------+----------+-----------+--------------------------+----------------------
    ↪
| IndexLookUp_8                 | 8000.00  | root      |                          |
    ↪                                                  |
```

```
| ├─Selection_7(Build)          | 8000.00  | cop[tikv] |                          | in(test.t.a,
    ↪  10, 20, 30), in(test.t.b, 40, 50, 60) |
| │ └─IndexFullScan_5           | 10000.00 | cop[tikv] | table:t, index:idx(a, b) | keep order:
    ↪ false, stats:pseudo                    |
| └─TableRowIDScan_6(Probe)     | 8000.00  | cop[tikv] | table:t                  | keep order:
    ↪ false, stats:pseudo                    |
+--
    ↪ ----------------------------+----------+-----------+--------------------------+----------------------
    ↪
4 rows in set, 1 warning (0.00 sec)
```

```
SHOW WARNINGS;
```

```
+--
    ↪ -------+------+-----------------------------------------------------------------------------------------
    ↪
| Level   | Code | Message
    ↪
    ↪ |
+--
    ↪ -------+------+-----------------------------------------------------------------------------------------
    ↪
| Warning | 1105 | Memory capacity of 100 bytes for 'tidb_opt_range_max_size' exceeded when
    ↪ building ranges. Less accurate ranges such as full range are chosen |
+--
    ↪ -------+------+-----------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

14.4.1.261 tidb_opt_scan_factor

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：浮点数
- 范围：[0, 2147483647]
- 默认值：1.5
- 表示升序扫描时，TiKV 在磁盘上扫描一行数据的开销。该变量是代价模型内部使用的变量，不建议修改该变量的值。

14.4.1.262 tidb_opt_seek_factor

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是

- 类型：浮点数
- 范围：[0, 2147483647]
- 默认值：20
- 表示 TiDB 从 TiKV 请求数据的初始开销。该变量是代价模型内部使用的变量，不建议修改该变量的值。

14.4.1.263 tidb_opt_skew_distinct_agg 从 v6.2.0 版本开始引入

> **注意：**
>
> 开启该变量带来的查询性能优化仅对 TiFlash 有效。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 这个变量用来设置优化器是否将带有 DISTINCT 的聚合函数（例如 `SELECT b, count(DISTINCT a)FROM t GROUP BY b`）改写为两层聚合函数（例如 `SELECT b, count(a)FROM (SELECT b, a FROM t GROUP BY b, a)t GROUP BY b`）。当聚合列有严重的数据倾斜，且 DISTINCT 列有很多不同的值时，这种改写能够避免查询执行过程中的数据倾斜，从而提升查询性能。

14.4.1.264 tidb_opt_three_stage_distinct_agg 从 v6.3.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 该变量用于控制在 MPP 模式下是否将 COUNT(DISTINCT) 聚合改写为三阶段分布式执行的聚合。
- 该变量目前仅对只有一个 COUNT(DISTINCT) 的聚合生效。

14.4.1.265 tidb_opt_tiflash_concurrency_factor

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：浮点数
- 范围：[0, 2147483647]
- 默认值：24.0
- 表示 TiFlash 计算的并发数。该变量是代价模型内部使用的变量，不建议修改该变量的值。

14.4.1.266 tidb_opt_write_row_id

- 作用域：SESSION
- 类型：布尔型

- 默认值：OFF
- 这个变量用来设置是否允许 INSERT、REPLACE 和 UPDATE 操作 _tidb_rowid 列，默认是不允许操作。该选项仅用于 TiDB 工具导数据时使用。

14.4.1.267 tidb_optimizer_selectivity_level

- 作用域：SESSION
- 类型：整数型
- 默认值：0
- 范围：[0, 2147483647]
- 控制优化器估算逻辑的更迭。更改该变量值后，优化器的估算逻辑会产生较大的改变。目前该变量的有效值只有 0，不建议设为其它值。

14.4.1.268 tidb_partition_prune_mode 从 v5.1 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：枚举型
- 默认值：dynamic
- 可选值：static、dynamic、static-only、dynamic-only
- 这个变量用来设置是否开启分区表动态裁剪模式。默认值为 dynamic。但是注意，dynamic 模式仅在表级别汇总统计信息（即 GlobalStats）收集完成的情况下生效。如果选择了 dynamic 但 GlobalStats 未收集完成，TiDB 会仍采用 static 模式。关于 GlobalStats 更多信息，请参考动态裁剪模式下的分区表统计信息。关于动态裁剪模式更多信息，请参考分区表动态裁剪模式。

14.4.1.269 tidb_persist_analyze_options 从 v5.4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用于控制是否开启ANALYZE 配置持久化特性。

14.4.1.270 tidb_pessimistic_txn_fair_locking 从 v7.0.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 是否对悲观锁启用加强的悲观锁唤醒模型。该模型可严格控制悲观锁单点冲突场景下事务的唤醒顺序，避免无效唤醒，大大降低原有唤醒机制中的随机性对事务延迟带来的不确定性。如果业务场景中遇到了单点悲观锁冲突频繁的情况（如高频更新同一行数据等），并进而引起语句重试频繁、尾延迟高，甚至偶尔发生 pessimistic lock retry limit reached 错误，可以尝试开启该变量来解决问题。
- 对于从 v7.0.0 以前的版本升级到 v7.0.0 或更新版本的 TiDB 集群，该选项默认关闭。

注意：

- 视具体业务场景的不同，启用该选项可能对存在频繁锁冲突的事务造成一定程度的吞吐下降（平均延迟上升）。
- 该选项目前仅对需要上锁单个 key 的语句有效。如果一个语句需要对多行同时上锁，则该选项不会对此类语句生效。
- 该功能从 v6.6.0 版本引入。在 v6.6.0 版本中，该功能由变量 tidb_pessimistic_txn_aggressive_locking 控制，默认关闭。

14.4.1.271 tidb_placement_mode 从 v6.0.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：STRICT
- 可选值：STRICT，IGNORE
- 该变量用于控制 DDL 语句是否忽略Placement Rules in SQL 指定的放置规则。变量值为 IGNORE 时将忽略所有放置规则选项。
- 该变量可由逻辑转储或逻辑恢复工具使用，确保即使绑定了不合适的放置规则，也始终可以成功创建表。这类似于 mysqldump 将 SET FOREIGN_KEY_CHECKS=0; 写入每个转储文件的开头部分。

14.4.1.272 tidb_pprof_sql_cpu 从 v4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：否，仅作用于当前连接的 TiDB 实例
- 默认值：0
- 范围：[0, 1]
- 这个变量用来控制是否在 profile 输出中标记出对应的 SQL 语句，用于定位和排查性能问题。

14.4.1.273 tidb_opt_prefix_index_single_scan 从 v6.4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用于控制 TiDB 优化器是否将某些过滤条件下推到前缀索引，尽量避免不必要的回表，从而提高查询性能。
- 将该变量设置为 ON 时，会将过滤条件下推到前缀索引。此时，假设一张表中 col 列是索引前缀列，查询语句中的 col is null 或者 col is not null 条件会被归为索引上的过滤条件，而不是回表时的过滤条件，从而避免不必要的回表。

该变量的使用示例

创建一张带前缀索引的表：

```
CREATE TABLE t (a INT, b VARCHAR(10), c INT, INDEX idx_a_b(a, b(5)));
```

此时关闭 tidb_opt_prefix_index_single_scan：

```
SET tidb_opt_prefix_index_single_scan = 'OFF';
```

对于以下查询，执行计划使用了前缀索引 idx_a_b 但需要回表（出现了 IndexLookUp 算子）。

```
EXPLAIN FORMAT='brief' SELECT COUNT(1) FROM t WHERE a = 1 AND b IS NOT NULL;
+--
    ↪ ----------------------------+---------+-----------+-----------------------------+----------------------
    ↪
| id                            | estRows | task      | access object               | operator
    ↪ info                                          |
+--
    ↪ ----------------------------+---------+-----------+-----------------------------+----------------------
    ↪
| HashAgg                       | 1.00    | root      |                             | funcs:
    ↪ count(Column#8)->Column#5                       |
| └─IndexLookUp                 | 1.00    | root      |                              |
    ↪                                                 |
|   ├─IndexRangeScan(Build)     | 99.90   | cop[tikv] | table:t, index:idx_a_b(a, b) | range:[1
    ↪ -inf,1 +inf], keep order:false, stats:pseudo |
|   └─HashAgg(Probe)            | 1.00    | cop[tikv] |                              | funcs:
    ↪ count(1)->Column#8                              |
|     └─Selection               | 99.90   | cop[tikv] |                              | not(
    ↪ isnull(test.t.b))                                |
|       └─TableRowIDScan        | 99.90   | cop[tikv] | table:t                      | keep
    ↪ order:false, stats:pseudo                       |
+--
    ↪ ----------------------------+---------+-----------+-----------------------------+----------------------
    ↪
6 rows in set (0.00 sec)
```

此时打开 tidb_opt_prefix_index_single_scan：

```
SET tidb_opt_prefix_index_single_scan = 'ON';
```

开启该变量后，对于以下查询，执行计划使用了前缀索引 idx_a_b 且不需要回表。

```
EXPLAIN FORMAT='brief' SELECT COUNT(1) FROM t WHERE a = 1 AND b IS NOT NULL;
+--
    ↪ ------------------------+---------+-----------+-----------------------------+-------------------------
    ↪
| id                          | estRows | task      | access object               | operator info
    ↪                                          |
```

```
+--
    ↪ ----------------------+---------+-----------+-----------------------------+-----------------------------
    ↪
| StreamAgg                | 1.00    | root      |                             | funcs:count(
    ↪ Column#7)->Column#5                         |
| └─IndexReader            | 1.00    | root      |                              | index:
    ↪ StreamAgg                                        |
|   └─StreamAgg            | 1.00    | cop[tikv] |                              | funcs:count(1)
    ↪ ->Column#7                                |
|     └─IndexRangeScan     | 99.90   | cop[tikv] | table:t, index:idx_a_b(a, b) | range:[1 -inf
    ↪ ,1 +inf], keep order:false, stats:pseudo |
+--
    ↪ ----------------------+---------+-----------+-----------------------------+-----------------------------
    ↪
4 rows in set (0.00 sec)
```

14.4.1.274 tidb_prepared_plan_cache_memory_guard_ratio 从 v6.1.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：0.1
- 范围：[0, 1]
- 这个变量用来控制 Prepared Plan Cache 触发内存保护机制的阈值，具体可见Prepared Plan Cache 的内存管理。
- 在 v6.1.0 之前这个开关通过 TiDB 配置文件 (prepared-plan-cache.memory-guard-ratio) 进行配置，升级到 v6.1.0 时会自动继承原有设置。

14.4.1.275 tidb_prepared_plan_cache_size 从 v6.1.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：100
- 范围：[1, 100000]
- 这个变量用来控制单个 SESSION 的 Prepared Plan Cache 最多能够缓存的计划数量，具体可见Prepared Plan Cache 的内存管理。
- 在 v6.1.0 之前这个开关通过 TiDB 配置文件 (prepared-plan-cache.capacity) 进行配置，升级到 v6.1.0 时会自动继承原有设置。

14.4.1.276 tidb_projection_concurrency

> **警告：**
>
> 从 v5.0 版本开始，该变量被废弃。请使用tidb_executor_concurrency 进行设置。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：-1
- 范围：[-1, 256]
- 单位：线程
- 这个变量用来设置 Projection 算子的并发度。
- 默认值 -1 表示使用 tidb_executor_concurrency 的值。

14.4.1.277 tidb_query_log_max_len

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：4096 (4 KiB)
- 范围：[0, 1073741824]
- 单位：字节
- 该变量控制 SQL 语句输出的最大长度。当一条 SQL 语句的输出长度大于 tidb_query_log_max_len 时，输出将会被截断。
- 在 v6.1.0 之前这个开关也可以通过 TiDB 配置文件 (log.query-log-max-len) 进行配置，升级到 v6.1.0 后仅可通过系统变量配置。

14.4.1.278 tidb_rc_read_check_ts 从 v6.0.0 版本开始引入

> 警告：
>
> - 该特性与replica-read 尚不兼容，开启 tidb_rc_read_check_ts 的读请求无法使用replica-read，请勿同时开启两项特性。
> - 如果客户端使用游标操作，建议不开启 tidb_rc_read_check_ts 这一特性，避免前一批返回数据已经被客户端使用而语句最终会报错的情况。
> - 自 v7.0.0 版本开始，该变量对于使用 prepared statement 协议下 cursor fetch read 游标模式不再生效。

- 作用域：GLOBAL
- 是否持久化到集群：否，仅作用于当前连接的 TiDB 实例
- 默认值：OFF
- 该变量用于优化时间戳的获取，适用于悲观事务 READ-COMMITTED 隔离级别下读写冲突较少的场景，开启此变量可以避免获取全局 timestamp 带来的延迟和开销，并优化事务内读语句延迟。
- 如果读写冲突较为严重，开启此功能会增加额外开销和延迟，造成性能回退。更详细的说明，请参考读已提交隔离级别 (Read Committed) 文档。

14.4.1.279 tidb_rc_write_check_ts 从 v6.3.0 版本开始引入

> 警告：
>
> 该特性与replica-read 尚不兼容。开启本变量后，客户端发送的所有请求都将无法使用 replica-read，因此请勿同时开启 tidb_rc_write_check_ts 和 replica-read。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 该变量用于优化时间戳的获取，适用于悲观事务 READ-COMMITTED 隔离级别下点写冲突较少的场景。开启此变量可以避免点写语句获取全局时间戳带来的延迟和开销。目前该变量适用的点写语句包括 UPDATE、DELETE、SELECT ..... FOR UPDATE 三种类型。点写语句是指将主键或者唯一键作为过滤条件且最终执行算子包含 POINT-GET 的写语句。
- 如果点写冲突较为严重，开启此变量会增加额外开销和延迟，造成性能回退。更详细的说明，请参考读已提交隔离级别 (Read Committed) 文档。

14.4.1.280 tidb_read_consistency New in v5.4.0

- 作用域：SESSION
- 类型：字符串
- 默认值：strict
- 此变量用于控制自动提交的读语句的读一致性。
- 如果将变量值设置为 weak，则直接跳过读语句遇到的锁，读的执行可能会更快，这就是弱一致性读模式。但在该模式下，事务语义（例如原子性）和分布式一致性（线性一致性）并不能得到保证。
- 如果用户场景中需要快速返回自动提交的读语句，并且可接受弱一致性的读取结果，则可以使用弱一致性读取模式。

14.4.1.281 tidb_read_staleness 从 v5.4.0 版本开始引入

- 作用域：SESSION
- 默认值：0
- 范围 [-2147483648, 0]
- 这个变量用于设置当前会话允许读取的历史数据范围。设置后，TiDB 会从参数允许的范围内选出一个尽可能新的时间戳，并影响后继的所有读操作。比如，如果该变量的值设置为 -5，TiDB 会在 5 秒时间范围内，保证 TiKV 拥有对应历史版本数据的情况下，选择尽可能新的一个时间戳。

14.4.1.282 tidb_record_plan_in_slow_log

- 作用域：GLOBAL
- 是否持久化到集群：否，仅作用于当前连接的 TiDB 实例
- 默认值：ON
- 这个变量用于控制是否在 slow log 里包含慢查询的执行计划。

14.4.1.283 tidb_redact_log

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 这个变量用于控制在记录 TiDB 日志和慢日志时，是否将 SQL 中的用户信息遮蔽。
- 将该变量设置为 1 即开启后，假设执行的 SQL 为 `insert into t values (1,2)`，在日志中记录的 SQL 会是 `insert into t values (?,?)`，即用户输入的信息被遮蔽。

14.4.1.284 tidb_regard_null_as_point 从 v5.4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用来控制优化器是否可以将包含 null 的等值条件作为前缀条件来访问索引。
- 该变量默认开启。开启后，该变量可以使优化器减少需要访问的索引数据量，从而提高查询的执行速度。例如，在有多列索引 `index(a, b)` 且查询条件为 `a<=>null and b=1` 的情况下，优化器可以同时使用查询条件中的 `a<=>null` 和 `b=1` 进行索引访问。如果关闭该变量，因为 `a<=>null and b=1` 包含 null 的等值条件，优化器不会使用 `b=1` 进行索引访问。

14.4.1.285 tidb_remove_orderby_in_subquery 从 v6.1.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 指定是否在子查询中移除 `ORDER BY` 子句。

14.4.1.286 tidb_replica_read 从 v4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：`leader`
- 可选值：`leader`、`follower`、`leader-and-follower`、`prefer-leader`、`closest-replicas`、`closest-adaptive` 和 `learner`。其中，`learner` 从 v6.6.0 开始引入。
- 这个变量用于控制 TiDB 的 Follower Read 功能的行为。
- 关于使用方式与实现原理，见Follower Read。

14.4.1.287 tidb_retry_limit

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：10

- 范围：[-1，9223372036854775807]
- 这个变量用来设置乐观事务的最大重试次数。一个事务执行中遇到可重试的错误（例如事务冲突、事务提交过慢或表结构变更）时，会根据该变量的设置进行重试。注意当 tidb_retry_limit = 0 时，也会禁用自动重试。该变量仅适用于乐观事务，不适用于悲观事务。

14.4.1.288 tidb_row_format_version

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：2
- 范围：[1，2]
- 控制新保存数据的表数据格式版本。TiDB v4.0 中默认使用版本号为 2 的新表数据格式保存新数据。
- 但如果从 4.0.0 之前的版本升级到 4.0.0，不会改变表数据格式版本，TiDB 会继续使用版本为 1 的旧格式写入表中，即只有新创建的集群才会默认使用新表数据格式。
- 需要注意的是修改该变量不会对已保存的老数据产生影响，只会对修改变量后的新写入数据使用对应版本格式保存。

14.4.1.289 tidb_scatter_region

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- TiDB 默认会在建表时为新表分裂 Region。开启该变量后，会在建表语句执行时，同步打散刚分裂出的 Region。适用于批量建表后紧接着批量写入数据，能让刚分裂出的 Region 先在 TiKV 分散而不用等待 PD 进行调度。为了保证后续批量写入数据的稳定性，建表语句会等待打散 Region 完成后再返回建表成功，建表语句执行时间会是该变量关闭时的数倍。
- 如果建表时设置了 SHARD_ROW_ID_BITS 和 PRE_SPLIT_REGIONS，建表成功后会均匀切分出指定数量的 Region。

14.4.1.290 tidb_server_memory_limit 从 v6.4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：80%
- 取值范围：
  - 你可以将该变量值设为百分比格式，表示内存用量占总内存的百分比，取值范围为 [1%，99%]。
  - 你还可以将变量值设为内存大小，取值范围为 0 以及 [536870912，9223372036854775807]，单位为 Byte。支持带单位的内存格式“KB|MB|GB|TB”。0 值表示不设内存限制。
  - 当设置的内存值小于 512 MB 且不为 0 时，TiDB 将会使用 512 MB 作为替代。

- 该变量指定 TiDB 实例的内存限制。TiDB 会在内存用量达到该限制时，对当前内存用量最高的 SQL 语句进行取消 (Cancel) 操作。在该 SQL 语句被成功 Cancel 掉后，TiDB 会尝试调用 Golang GC 立刻回收内存，以最快速度缓解内存压力。
- 只有内存使用大于 `tidb_server_memory_limit_sess_min_size` 的 SQL 语句会被选定为最优先被 Cancel 的 SQL 语句。
- 目前 TiDB 一次只能 Cancel 一条 SQL 语句。如果 TiDB 完全 Cancel 掉一条 SQL 语句并回收资源后，内存使用仍然大于该变量所设限制，TiDB 会开始下一次 Cancel 操作。

14.4.1.291 `tidb_server_memory_limit_gc_trigger` 从 v6.4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：70%
- 取值范围：[50%, 99%]
- TiDB 尝试触发 GC 的阈值。当 TiDB 的内存使用达到 `tidb_server_memory_limit` 值 * `tidb_server_memory_limit_gc_trigger` 值时，则会主动触发一次 Golang GC。在一分钟之内只会主动触发一次 GC。

14.4.1.292 `tidb_server_memory_limit_sess_min_size` 从 v6.4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：134217728（即 128 MB）
- 取值范围：[128, 9223372036854775807]，单位为 Byte。支持带单位的内存格式“KB|MB|GB|TB”。
- 开启内存限制后，TiDB 会终止当前实例上内存用量最高的 SQL 语句。本变量指定此情况下 SQL 语句被终止的最小内存用量。如果 TiDB 实例的内存超限是由许多内存使用量不明显的会话导致的，可以适当调小该变量值，使得更多会话成为 Cancel 的对象。

14.4.1.293 `tidb_shard_allocate_step` 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：9223372036854775807
- 范围：[1, 9223372036854775807]
- 该变量设置为AUTO_RANDOM 或SHARD_ROW_ID_BITS 属性列分配的最大连续 ID 数。通常，AUTO_RANDOM ID 或带有 SHARD_ROW_ID_BITS 属性的行 ID 在一个事务中是增量和连续的。你可以使用该变量来解决大事务场景下的热点问题。

14.4.1.294 `tidb_simplified_metrics`

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：OFF
- 该变量开启后，TiDB 将不会收集或记录 Grafana 面板未使用到的 metrics。

14.4.1.295 tidb_skip_ascii_check 从 v5.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 这个变量用来设置是否校验 ASCII 字符的合法性。
- 校验 ASCII 字符会损耗些许性能。当你确认输入的字符串为有效的 ASCII 字符时，可以将其设置为 ON。

14.4.1.296 tidb_skip_isolation_level_check

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 开启这个开关之后，如果对 tx_isolation 赋值一个 TiDB 不支持的隔离级别，不会报错，有助于兼容其他设置了（但不依赖于）不同隔离级别的应用。

```
tidb> set tx_isolation='serializable';
ERROR 8048 (HY000): The isolation level 'serializable' is not supported. Set
    ↪ tidb_skip_isolation_level_check=1 to skip this error
tidb> set tidb_skip_isolation_level_check=1;
Query OK, 0 rows affected (0.00 sec)

tidb> set tx_isolation='serializable';
Query OK, 0 rows affected, 1 warning (0.00 sec)
```

14.4.1.297 tidb_skip_utf8_check

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 这个变量用来设置是否校验 UTF-8 字符的合法性。
- 校验 UTF-8 字符会损耗些许性能。当你确认输入的字符串为有效的 UTF-8 字符时，可以将其设置为 ON。

> **注意：**
>
> 跳过字符检查可能会使 TiDB 检测不到应用写入的非法 UTF-8 字符，进一步导致执行 ANALYZE 时解码错误，以及引入其他未知的编码问题。如果应用不能保证写入字符串的合法性，不建议跳过该检查。

14.4.1.298 tidb_slow_log_threshold

- 作用域：GLOBAL
- 是否持久化到集群：否，仅作用于当前连接的 TiDB 实例
- 默认值：300
- 范围：[-1, 9223372036854775807]
- 单位：毫秒
- 输出慢日志的耗时阈值。当查询大于这个值，就会当做是一个慢查询，输出到慢查询日志。默认为 300 ms。

14.4.1.299 tidb_slow_query_file

- 作用域：SESSION
- 默认值：“”
- 查询 INFORMATION_SCHEMA.SLOW_QUERY 只会解析配置文件中 slow-query-file 设置的慢日志文件名，默认是“tidb-slow.log”。但如果想要解析其他的日志文件，可以通过设置 session 变量 tidb_slow_query_file 为具体的文件路径，然后查询 INFORMATION_SCHEMA.SLOW_QUERY 就会按照设置的路径去解析慢日志文件。更多详情可以参考SLOW_QUERY 文档。

14.4.1.300 tidb_snapshot

- 作用域：SESSION
- 默认值：“”
- 这个变量用来设置当前会话期待读取的历史数据所处时刻。比如当设置为 "2017-11-11 20:20:20" 时或者一个 TSO 数字“400036290571534337”，当前会话将能读取到该时刻的数据。

14.4.1.301 tidb_source_id 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：1
- 范围：[1, 15]
- 这个变量用来设置在双向复制系统内不同集群的 ID。

14.4.1.302 tidb_stats_cache_mem_quota 从 v6.1.0 版本开始引入

> 警告：
>
> 该变量为实验特性，不推荐在生产环境中使用。

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：0
- 这个变量用于控制 TiDB 内部统计信息缓存使用内存的上限。

14.4.1.303 tidb_stats_load_sync_wait 从 v5.4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：100
- 单位：毫秒
- 范围：[0, 2147483647]
- 这个变量用于控制是否开启统计信息的同步加载模式（为 0 代表不开启，即为异步加载模式），以及开启的情况下，SQL 执行同步加载完整统计信息等待多久后会超时。更多信息，请参考统计信息的加载。

14.4.1.304 tidb_stats_load_pseudo_timeout 从 v5.4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用于控制统计信息同步加载超时后，SQL 是执行失败（OFF），还是退回使用 pseudo 的统计信息（ON）。

14.4.1.305 tidb_stmt_summary_enable_persistent 从 v6.6.0 版本开始引入

> **警告：**
>
> statements summary 持久化目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现 bug，请在 GitHub 上提 issue 反馈。

- 作用域：GLOBAL
- 类型：布尔型
- 默认值：OFF
- 只读变量。表示是否开启statement summary tables 持久化。该变量的值与配置文件中tidb_stmt_summary_enable_persistent 的取值相同。

14.4.1.306 tidb_stmt_summary_filename 从 v6.6.0 版本开始引入

> **警告：**
>
> statements summary 持久化目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现 bug，请在 GitHub 上提 issue 反馈。

- 作用域：GLOBAL
- 类型：字符串

- 默认值：`"tidb-statements.log"`
- 只读变量。表示当开启statement summary tables 持久化后持久化数据所写入的文件。该变量的值与配置文件中`tidb_stmt_summary_filename` 的取值相同。

14.4.1.307 `tidb_stmt_summary_file_max_backups` 从 v6.6.0 版本开始引入

> **警告：**
>
> statements summary 持久化目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现 bug，请在 GitHub 上提 issue 反馈。

- 作用域：GLOBAL
- 类型：整数型
- 默认值：0
- 只读变量。表示当开启statement summary tables 持久化后持久化数据文件的最大数量限制。该变量的值与配置文件中`tidb_stmt_summary_file_max_backups` 的取值相同。

14.4.1.308 `tidb_stmt_summary_file_max_days` 从 v6.6.0 版本开始引入

> **警告：**
>
> statements summary 持久化目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现 bug，请在 GitHub 上提 issue 反馈。

- 作用域：GLOBAL
- 类型：整数型
- 默认值：3
- 单位：天
- 只读变量。表示当开启statement summary tables 持久化后持久化数据文件所保留的最大天数。该变量的值与配置文件中`tidb_stmt_summary_file_max_days` 的取值相同。

14.4.1.309 `tidb_stmt_summary_file_max_size` 从 v6.6.0 版本开始引入

> **警告：**
>
> statements summary 持久化目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现 bug，请在 GitHub 上提 issue 反馈。

- 作用域：GLOBAL
- 类型：整数型
- 默认值：64
- 单位：MiB
- 只读变量。表示当开启statement summary tables 持久化后持久化数据单个文件的大小限制。该变量的值与配置文件中tidb_stmt_summary_file_max_size 的取值相同。

14.4.1.310 tidb_stmt_summary_history_size 从 v4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：24
- 范围：[0, 255]
- 这个变量设置了statement summary tables 的历史记录容量。

14.4.1.311 tidb_stmt_summary_internal_query 从 v4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 这个变量用来控制是否在statement summary tables 中包含 TiDB 内部 SQL 的信息。

14.4.1.312 tidb_stmt_summary_max_sql_length 从 v4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：4096
- 范围：[0, 2147483647]
- 这个变量控制statement summary tables 显示的 SQL 字符串长度。

14.4.1.313 tidb_stmt_summary_max_stmt_count 从 v4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：3000
- 范围：[1, 32767]
- 这个变量设置了statement summary tables 在内存中保存的语句的最大数量。

14.4.1.314 tidb_stmt_summary_refresh_interval 从 v4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：1800

- 范围：[1, 2147483647]
- 单位：秒
- 这个变量设置了statement summary tables 的刷新时间。

14.4.1.315 tidb_store_batch_size

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：4
- 范围：[0, 25000]
- 设置 IndexLookUp 算子回表时多个 Coprocessor Task 的 batch 大小。0 代表不使用 batch。当 IndexLookUp 算子的回表 Task 数量特别多，出现极长的慢查询时，可以适当调大该参数以加速查询。

14.4.1.316 tidb_streamagg_concurrency

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：1
- 设置 StreamAgg 算子执行查询时的并发度。
- 不推荐设置该变量，修改该变量值可能会造成数据正确性问题。

14.4.1.317 tidb_top_sql_max_meta_count 从 v6.0.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：5000
- 范围：[1, 10000]
- 这个变量用于控制Top SQL 每分钟最多收集 SQL 语句类型的数量。

14.4.1.318 tidb_top_sql_max_time_series_count 从 v6.0.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：100
- 范围：[1, 5000]
- 这个变量用于控制Top SQL 每分钟保留消耗负载最大的前多少条 SQL（即 Top N) 的数据。

> **注意：**
>
> TiDB Dashboard 中的 Top SQL 页面目前只显示消耗负载最多的 5 类 SQL 查询，这与 tidb_top_sql_max_time_series_count 的配置无关。

14.4.1.319 tidb_store_limit 从 v3.0.4 和 v4.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：0
- 范围：[0, 9223372036854775807]
- 这个变量用于限制 TiDB 同时向 TiKV 发送的请求的最大数量，0 表示没有限制。

14.4.1.320 tidb_super_read_only 从 v5.3.1 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值: OFF。
- tidb_super_read_only 用于实现对 MySQL 变量 super_read_only 的替代。然而，由于 TiDB 是一个分布式数据库，开启 tidb_super_read_only 后数据库各个 TiDB 服务器进入只读模式的时刻不是强一致的，而是最终一致的。
- 拥有 SUPER 或 SYSTEM_VARIABLES_ADMIN 权限的用户可以修改该变量。
- 该变量可以控制整个集群的只读状态。开启后（即该值为 ON），整个集群中的 TiDB 服务器都将进入只读状态，只有 SELECT、USE、SHOW 等不会修改数据的语句才能被执行，其他如 INSERT、UPDATE 等语句会被拒绝执行。
- 该变量开启只读模式只保证整个集群最终进入只读模式，当变量修改状态还没被同步到其他 TiDB 服务器时，尚未同步的 TiDB 仍然停留在非只读模式。
- 在执行 SQL 语句之前，TiDB 会检查集群的只读标志。从 v6.2.0 起，在提交 SQL 语句之前，TiDB 也会检查该标志，从而防止在服务器被置于只读模式后某些长期运行的auto commit 语句可能修改数据的情况。
- 在变量开启时，对于尚未提交的事务：
  - 如果有尚未提交的只读事务，可正常提交该事务。
  - 如果尚未提交的事务为非只读事务，在事务内执行写入的 SQL 语句会被拒绝。
  - 如果尚未提交的事务已经有数据改动，其提交也会被拒绝。
- 当集群开启只读模式后，所有用户（包括 SUPER 用户）都无法执行可能写入数据的 SQL 语句，除非该用户被显式地授予了 RESTRICTED_REPLICA_WRITER_ADMIN 权限。
- 当系统变量tidb_restricted_read_only 为 ON 时，tidb_super_read_only 的值会受到tidb_restricted_read_only 的影响。详情请参见tidb_restricted_read_only 中的描述。

14.4.1.321 tidb_sysdate_is_now 从 v6.0.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 这个变量用于控制 SYSDATE 函数能否替换为 NOW 函数，其效果与 MYSQL 中的 sysdate-is-now 一致。

14.4.1.322 tidb_sysproc_scan_concurrency 从 v6.5.0 版本开始引入

- 作用域：GLOBAL

- 是否持久化到集群：是
- 类型：整数型
- 默认值：1
- 范围：[1, 256]
- 这个变量用来设置 TiDB 执行内部 SQL 语句（例如统计信息自动更新）时 scan 操作的并发度。

14.4.1.323 tidb_table_cache_lease 从 v6.0.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：3
- 范围：[1, 10]
- 单位：秒
- 这个变量用来控制缓存表的 lease 时间，默认值是 3 秒。该变量值的大小会影响缓存表的修改。在缓存表上执行修改操作后，最长可能出现 tidb_table_cache_lease 变量值时长的等待。如果业务表为只读表，或者能接受很高的写入延迟，则可以将该变量值调大，从而增加缓存的有效时间，减少 lease 续租的频率。

14.4.1.324 tidb_tmp_table_max_size 从 v5.3 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：67108864
- 范围：[1048576, 137438953472]
- 单位：字节
- 这个变量用于限制单个临时表的最大大小，临时表超出该大小后报错。

14.4.1.325 tidb_track_aggregate_memory_usage

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：布尔型
- 默认值：ON
- 本变量控制 TiDB 是否跟踪聚合函数的内存使用情况。

> **警告：**
>
> 如果禁用该变量，TiDB 可能无法准确跟踪内存使用情况，并且无法控制对应 SQL 语句的内存使用。

14.4.1.326 tidb_tso_client_batch_max_wait_time 从 v5.3.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：0
- 范围：[0, 10]
- 单位：毫秒
- 这个变量用来设置 TiDB 向 PD 请求 TSO 时进行一次攒批操作的最大等待时长。默认值为 0，即不进行额外的等待。
- 在向 PD 获取 TSO 请求时，TiDB 使用的 PD Client 会一次尽可能多地收集同一时刻的 TSO 请求，将其攒批合并成一个 RPC 请求后再发送给 PD，从而减轻 PD 的压力。
- 将这个变量值设置为非 0 后，TiDB 会在每一次攒批结束前进行一个最大时长为其值的等待，目的是为了收集到更多的 TSO 请求，从而提高攒批效果。
- 适合调高这个变量值的场景：
  - PD leader 因高压力的 TSO 请求而达到 CPU 瓶颈，导致 TSO RPC 请求的延迟较高。
  - 集群中 TiDB 实例的数量不多，但每一台 TiDB 实例上的并发量较高。
- 在实际使用中，推荐将该变量尽可能设置为一个较小的值。

> 注意：
>
> 如果 PD leader 的 TSO RPC 延迟升高，但其现象并非由 CPU 使用率达到瓶颈而导致（可能存在网络等问题），此时，调高 tidb_tso_client_batch_max_wait_time 可能会导致 TiDB 的语句执行延迟上升，影响集群的 QPS 表现。

14.4.1.327 tidb_ttl_delete_rate_limit 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：0
- 范围：[0, 9223372036854775807]
- 这个变量用来对每个 TiDB 节点的 TTL 删除操作进行限流。其值代表了在 TTL 任务中单个节点每秒允许 DELETE 语句执行的最大次数。当此变量设置为 0 时，则表示不做限制。更多信息，请参考Time to Live。

14.4.1.328 tidb_ttl_delete_batch_size 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：100
- 范围：[1, 10240]
- 这个变量用于设置 TTL 任务中单个删除事务中允许删除的最大行数。更多信息，请参考Time to Live。

14.4.1.329 tidb_ttl_delete_worker_count 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：4
- 范围：[1, 256]
- 这个变量用于设置每个 TiDB 节点上 TTL 删除任务的最大并发数。更多信息，请参考Time to Live。

14.4.1.330 tidb_ttl_job_enable 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 类型：布尔型
- 这个变量用于控制是否启动 TTL 后台清理任务。如果设置为 OFF，所有具有 TTL 属性的表会自动停止对过期数据的清理。更多信息，请参考Time to Live。

14.4.1.331 tidb_ttl_scan_batch_size 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：500
- 范围：[1, 10240]
- 这个变量用于设置 TTL 任务中用来扫描过期数据的每个 SELECT 语句的 LIMIT 的值。更多信息，请参考Time to Live。

14.4.1.332 tidb_ttl_scan_worker_count 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：4
- 范围：[1, 256]
- 这个变量用于设置每个 TiDB 节点 TTL 扫描任务的最大并发数。更多信息，请参考Time to Live。

14.4.1.333 tidb_ttl_job_schedule_window_start_time 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 类型：时间
- 是否持久化到集群：是
- 默认值：00:00 +0000
- 这个变量用于控制 TTL 后台清理任务的调度窗口的起始时间。请谨慎调整此参数，过小的窗口有可能会造成过期数据的清理无法完成。更多信息，请参考Time to Live。

14.4.1.334 tidb_ttl_job_schedule_window_end_time 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 类型：时间
- 是否持久化到集群：是
- 默认值：23:59 +0000
- 这个变量用于控制 TTL 后台清理任务的调度窗口的结束时间。请谨慎调整此参数，过小的窗口有可能会造成过期数据的清理无法完成。更多信息，请参考Time to Live。

14.4.1.335 tidb_ttl_running_tasks 从 v7.0.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：-1
- 范围：-1 或 [1, 256]
- 这个变量用于限制整个集群内 TTL 任务的并发量。-1 表示与 TiKV 节点的数量相同。更多信息，请参考Time to Live。

14.4.1.336 tidb_txn_assertion_level 从 v6.0.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：FAST
- 可选值：OFF，FAST，STRICT
- 这个变量用于设置 assertion 级别。assertion 是一项在事务提交过程中进行的数据索引一致性校验，它对正在写入的 key 是否存在进行检查。如果不符则说明数据索引不一致，会导致事务 abort。详见数据索引一致性报错。
- 对于新创建的 v6.0.0 及以上的集群，默认值为 FAST。对于升级版本的集群，如果升级前是低于 v6.0.0 的版本，升级后默认值为 OFF。
  - OFF: 关闭该检查。
  - FAST: 开启大多数检查项，对性能几乎无影响。
  - STRICT: 开启全部检查项，当系统负载较高时，对悲观事务的性能有较小影响。

14.4.1.337 tidb_txn_commit_batch_size 从 v6.2.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：16384

- 范围：[1, 1073741824]
- 单位：字节
- 这个变量用于控制 TiDB 向 TiKV 发送的事务提交请求的批量大小。如果业务负载的大部分事务都有大量的写操作，适当调大该变量可以提高批处理的效果。但需要注意的是，设置过大将会超过 TiKV 的raft-entry-max-size 限制，导致提交失败。

14.4.1.338 tidb_txn_mode

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：pessimistic
- 可选值：pessimistic，optimistic
- 这个变量用于设置事务模式。TiDB v3.0 支持了悲观事务，自 v3.0.8 开始，默认使用悲观事务模式。
- 但如果从 3.0.7 及之前的版本升级到 >= 3.0.8 的版本，不会改变默认事务模式，即只有新创建的集群才会默认使用悲观事务模式。
- 将该变量设置为 “optimistic” 或 “” 时，将会使用乐观事务模式。

14.4.1.339 tidb_use_plan_baselines 从 v4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用于控制是否开启执行计划绑定功能，默认打开，可通过赋值 OFF 来关闭。关于执行计划绑定功能的使用可以参考执行计划绑定文档。

14.4.1.340 tidb_wait_split_region_finish

- 作用域：SESSION
- 默认值：ON
- 由于打散 Region 的时间可能比较长，主要由 PD 调度以及 TiKV 的负载情况所决定。这个变量用来设置在执行 SPLIT REGION 语句时，是否同步等待所有 Region 都打散完成后再返回结果给客户端。
  - 默认 ON 代表等待打散完成后再返回结果
  - OFF 代表不等待 Region 打散完成就返回。
- 需要注意的是，在 Region 打散期间，对正在打散 Region 上的写入和读取的性能会有一定影响，对于批量写入、导数据等场景，还是建议等待 Region 打散完成后再开始导数据。

14.4.1.341 tidb_wait_split_region_timeout

- 作用域：SESSION
- 默认值：300
- 范围：[1, 2147483647]
- 单位：秒
- 这个变量用来设置 SPLIT REGION 语句的执行超时时间，默认值是 300 秒，如果超时还未完成，就返回一个超时错误。

14.4.1.342 tidb_window_concurrency 从 v4.0 版本开始引入

> **警告：**
>
> 从 v5.0 版本开始，该变量被废弃。请使用tidb_executor_concurrency 进行设置。

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：-1
- 范围：[1, 256]
- 单位：线程
- 这个变量用于设置 window 算子的并行度。
- 默认值 -1 表示使用 tidb_executor_concurrency 的值。

14.4.1.343 tiflash_fastscan 从 v6.3.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 类型：布尔型
- 如果开启FastScan 功能（设置为 ON 时），TiFlash 可以提供更高效的查询性能，但不保证查询结果的精度和数据一致性。

14.4.1.344 tiflash_fine_grained_shuffle_batch_size 从 v6.2.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：8192
- 范围：[1, 18446744073709551615]
- 细粒度 shuffle 功能开启时，下推到 TiFlash 的窗口函数可以并行执行。该变量控制发送端发送数据的攒批大小。
- 对性能影响：如果该值设置过小，例如极端值 1，会导致每个 Block 都进行一次网络传输。如果设置过大，例如极端值整个表的行数，会导致接收端大部分时间都在等待数据，无法流水线计算。可以观察 TiFlash 接收端收到的行数分布情况，如果大部分线程接收的行数很少，例如只有几百行，可以增加该值以达到减少网络开销的目的。

14.4.1.345 tiflash_fine_grained_shuffle_stream_count 从 v6.2.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 类型：整数型

- 默认值：0
- 范围：[-1, 1024]
- 当窗口函数下推到 TiFlash 执行时，可以通过该变量控制窗口函数执行的并行度。不同取值含义：
  - -1: 表示不使用细粒度 shuffle 功能，下推到 TiFlash 的窗口函数以单线程方式执行
  - 0: 表示使用细粒度 shuffle 功能。如果tidb_max_tiflash_threads 有效 (大于 0),则 tiflash_fine_grained_shuffle_str
    ↪ 会自动取值为tidb_max_tiflash_threads，否则为默认值 8。最终在 TiFlash 上窗口函数的实际并发度为：min(tiflash_fine_grained_shuffle_stream_count，TiFlash 节点物理线程数)
  - 大于 0: 表示使用细粒度 shuffle 功能，下推到 TiFlash 的窗口函数会以多线程方式执行，并发度为：min(tiflash_fine_grained_shuffle_stream_count, TiFlash 节点物理线程数)
- 理论上窗口函数的性能会随着该值的增加线性提升。但是如果设置的值超过实际的物理线程数，反而会导致性能下降。

14.4.1.346 time_zone

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：SYSTEM
- 数据库所使用的时区。这个变量值可以写成时区偏移的形式，如 '-8:00'，也可以写成一个命名时区，如 'America/Los_Angeles'。
- 默认值 SYSTEM 表示时区应当与系统主机的时区相同。系统的时区可通过system_time_zone 获取。

14.4.1.347 timestamp

- 作用域：SESSION
- 类型：浮点数
- 默认值：0
- 取值范围：[0, 2147483647]
- 一个 Unix 时间戳。变量值非空时，表示 CURRENT_TIMESTAMP()、NOW() 等函数的时间戳。该变量通常用于数据恢复或数据复制。

14.4.1.348 transaction_isolation

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：REPEATABLE-READ
- 可选值：READ-UNCOMMITTED，READ-COMMITTED，REPEATABLE-READ，SERIALIZABLE
- 这个变量用于设置事务隔离级别。TiDB 为了兼容 MySQL，支持可重复读 (REPEATABLE-READ)，但实际的隔离级别是快照隔离。详情见事务隔离级别。

14.4.1.349 tx_isolation

这个变量是 transaction_isolation 的别名。

14.4.1.350 tx_isolation_one_shot

> 注意：
> 该变量仅用于 TiDB 内部实现，不推荐设置该变量。

在 TiDB 内部实现中，TiDB 解释器会将 `SET TRANSACTION ISOLATION LEVEL [READ COMMITTED| REPEATABLE READ | ...]` 语句转化为 `SET @@SESSION.TX_ISOLATION_ONE_SHOT = [READ COMMITTED| REPEATABLE READ | ...]`。

14.4.1.351 tx_read_ts

- 作用域：SESSION
- 默认值：“”
- 在 Stale Read 场景下，该会话变量用于帮助记录 Stable Read TS 值。
- 该变量仅用于 TiDB 内部实现，不推荐设置该变量。

14.4.1.352 txn_scope

- 作用域：SESSION
- 默认值：`global`
- 可选值：`global` 和 `local`
- 该变量用于设置当前会话下事务为全局事务（设为 `global`）还是局部事务（设为 `local`）。
- 该变量仅用于 TiDB 内部实现，不推荐设置该变量。

14.4.1.353 validate_password.check_user_name 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 类型：布尔型
- 该变量是密码复杂度策略检查中的一个检查项，用于进行密码与用户名匹配检查。只有validate_password.enable 开启时，该变量才生效。
- 当该变量生效且为 ON 时，如果设置账户密码，TiDB 会将密码与当前会话账户的用户名部分（不包含主机名部分）进行比较，如果匹配则拒绝该密码。
- 该变量独立于validate_password.policy，即不受密码复杂度检测强度的控制。

14.4.1.354 validate_password.dictionary 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是

- 默认值：“”
- 类型：字符串
- 该变量是密码复杂度策略检查中的一个检查项，用于进行密码与字典字符串匹配检查。只有当validate_password.enable 开启且validate_password.policy 设置为 2 (STRONG) 时，该变量才生效。
- 该变量是一个长字符串，长度不超过 1024，字符串内容可包含一个或多个在密码中不允许出现的单词，每个单词之间采用英文分号（;）分隔。
- 默认情况下，该变量为空值，不执行字典检查。要进行字典检查，该变量值必须包含待匹配的单词。配置了该变量后，在设置账户密码时，TiDB 会将长度为 4 到 100 的密码的每个子字符串与该变量中配置的单词进行比较。任何匹配都会导致密码被拒绝。比较不区分大小写。

14.4.1.355 validate_password.enable 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 默认值：OFF
- 类型：布尔型
- 该变量是密码复杂度策略检查的开关。该变量设置为 ON 后，当设置账户密码时，TiDB 才会进行密码复杂度的各项检查。

14.4.1.356 validate_password.length 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：8
- 范围：[0, 2147483647]
- 该变量是密码复杂度策略检查中的一个检查项，用于限定密码的最小长度，默认最小长度为 8。只有validate_password.enable 开启时，该变量才生效。
- 设置该变量时有最小值要求，最小值由其他几个相关的系统变量控制，即该变量的值不能设置为小于此表达式的值：validate_password.number_count + validate_password.special_char_count + (2 * validate_password.mixed_case_count)。
- 当用户修改validate_password.number_count、validate_password.special_char_count、validate_password.mixed_case_count 后导致表达式的值大于 validate_password.length 时，validate_password.length 将自动被修改为满足表达式的最小值。

14.4.1.357 validate_password.mixed_case_count 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：1
- 范围：[0, 2147483647]
- 该变量是密码复杂度策略检查中的一个检查项，用于限定密码中至少需要包含多少个大写字符和小写字符。只有当validate_password.enable 开启且validate_password.policy 大于或等于 1 (MEDIUM) 时，该变量才生效。

- 对于给定的 validate_password.mixed_case_count 值，密码中的小写字符数和大写字符数都不能少于该值。例如，值为 1 时，密码中至少需要 1 个小写字母，至少需要 1 个大写字母。

14.4.1.358 validate_password.number_count 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：1
- 范围：[0, 2147483647]
- 该变量是密码复杂度策略检查中的一个检查项，用于限定密码中至少需要包含多少个数字字符。只有当validate_password.enable 开启且validate_password.policy 大于或等于 1 (MEDIUM) 时，该变量才生效。

14.4.1.359 validate_password.policy 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：枚举型
- 默认值：1
- 可选值：[0, 1, 2]
- 该变量是密码复杂度策略检查的强度策略，该变量影响其他密码复杂度系统变量（前缀为 validate_password）在密码检查时是否生效，但是 validate_password.check_user_name 除外。只有validate_password.enable 开启时，该变量才生效。
- 该变量可以使用数值 0、1、2 或相应的符号值 LOW、MEDIUM、STRONG，密码强度策略对应的检查项如下：
  - 0 或者 LOW：检查密码长度。
  - 1 或者 MEDIUM：检查密码长度，检查密码中数字、小写字符、大写字符、特殊字符数量。
  - 2 或者 STRONG：检查密码长度，检查密码中数字、小写字符、大写字符、特殊字符数量，检查密码字典匹配。

14.4.1.360 validate_password.special_char_count 从 v6.5.0 版本开始引入

- 作用域：GLOBAL
- 是否持久化到集群：是
- 类型：整数型
- 默认值：1
- 范围：[0, 2147483647]
- 该变量是密码复杂度策略检查中的一个检查项，用于限定密码中至少需要包含多少个特殊字符。只有当validate_password.enable 开启且validate_password.policy 大于或等于 1 (MEDIUM) 时，该变量才生效。

14.4.1.361 version

- 作用域：NONE
- 默认值：5.7.25-TiDB-(tidb version)
- 这个变量的值是 MySQL 的版本和 TiDB 的版本，例如 ‘5.7.25-TiDB-v4.0.0-beta.2-716-g25e003253’。

14.4.1.362 version_comment

- 作用域：NONE
- 默认值：(string)
- 这个变量的值是 TiDB 版本号的其他信息，例如 ‘TiDB Server (Apache License 2.0) Community Edition, MySQL 5.7 compatible’。

14.4.1.363 version_compile_os

- 作用域：NONE
- 默认值：(string)
- 这个变量值是 TiDB 所在操作系统的名称。

14.4.1.364 version_compile_machine

- 作用域：NONE
- 默认值：(string)
- 这个变量值是运行 TiDB 的 CPU 架构的名称。

14.4.1.365 wait_timeout

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：28800
- 范围：[0, 31536000]
- 单位：秒
- 这个变量表示用户会话的空闲超时。0 代表没有时间限制。

14.4.1.366 warning_count

- 作用域：SESSION
- 默认值：0
- 这个只读变量表示之前执行语句中出现的警告数。

14.4.1.367 windowing_use_high_precision

- 作用域：SESSION | GLOBAL
- 是否持久化到集群：是
- 默认值：ON
- 这个变量用于控制计算窗口函数时是否采用高精度模式。

14.5 配置文件参数

14.5.1 TiDB 配置文件描述

TiDB 配置文件比命令行参数支持更多的选项。你可以在 config/config.toml.example 找到默认值的配置文件，重命名为 `config.toml` 即可。本文档只介绍未包含在命令行参数中的参数。

> Tip:
>
> 如果你需要调整配置项的值，请参考修改配置参数进行操作。

14.5.1.0.1 `split-table`

- 为每个 table 建立单独的 Region。
- 默认值：true
- 如果需要创建大量的表（例如 10 万张以上），建议将此参数设置为 false。

14.5.1.0.2 `tidb-max-reuse-chunk` 从 v6.4.0 版本开始引入

- 用于控制每个连接最多缓存的 Chunk 对象数。配置过大会增加 OOM 的风险。
- 默认值：64
- 最小值：0
- 最大值：2147483647

14.5.1.0.3 `tidb-max-reuse-column` 从 v6.4.0 版本开始引入

- 用于控制每个连接最多缓存的 column 对象数。配置过大会增加 OOM 的风险。
- 默认值：256
- 最小值：0
- 最大值：2147483647

14.5.1.0.4 `token-limit`

- 可以同时执行请求的 session 个数
- 类型：Integer
- 默认值：1000
- 最小值：1
- 最大值（64 位平台）：18446744073709551615
- 最大值（32 位平台）：4294967295

14.5.1.0.5 temp-dir 从 v6.3.0 版本开始引入

- TiDB 用于存放临时数据的路径。如果一个功能需要使用 TiDB 节点的本地存储，TiDB 将把对应数据临时存放在这个目录下。
- 在创建索引的过程中，如果开启了创建索引加速，那么新创建索引需要回填的数据会被先存放在 TiDB 本地临时存储路径，然后批量导入到 TiKV，从而提升索引创建速度。
- 默认值：“/tmp/tidb”

> **注意：**
>
> 如果目录不存在，TiDB 在启动时会自动创建该目录。如果目录创建失败，或者 TiDB 对该目录没有读写权限，Fast Online DDL 在运行时可能产生不可预知的问题。

14.5.1.0.6 oom-use-tmp-storage

> **警告：**
>
> 自 v6.3.0 起，该配置项被废弃，其功能由系统变量tidb_enable_tmp_storage_on_oom 代替。集群升级到 v6.3.0 及之后的版本后，会自动继承升级前的 oom-use-tmp-storage 设置，升级后再设置 oom-use-tmp-storage 将不生效。

- 设置是否在单条 SQL 语句的内存使用超出系统变量tidb_mem_quota_query 限制时为某些算子启用临时磁盘。
- 默认值：true

14.5.1.0.7 tmp-storage-path

- 单条 SQL 语句的内存使用超出系统变量tidb_mem_quota_query 限制时，某些算子的临时磁盘存储位置。
- 默认值：<操作系统临时文件夹>/<操作系统用户ID>_tidb/MC4wLjAuMDo0MDAwLzAuMC4wLjA6MTAwODA=/tmp-storage。其中 MC4wLjAuMDo0MDAwLzAuMC4wLjA6MTAwODA= 是对 <host>:<port>/<statusHost>:<statusPort> 进行 Base64 编码的输出结果。
- 此配置仅在系统变量tidb_enable_tmp_storage_on_oom 的值为 ON 时有效。

14.5.1.0.8 tmp-storage-quota

- tmp-storage-path 存储使用的限额。
- 单位：Byte
- 当单条 SQL 语句使用临时磁盘，导致 TiDB server 的总体临时磁盘总量超过 tmp-storage-quota 时，当前 SQL 操作会被取消，并返回 Out Of Global Storage Quota! 错误。

- 当 tmp-storage-quota 小于 0 时则没有上述检查与限制。
- 默认值：-1
- 当 tmp-storage-path 的剩余可用容量低于 tmp-storage-quota 所定义的值时，TiDB server 启动时将会报出错误并退出。

14.5.1.0.9 lease

- DDL 租约超时时间。
- 默认值：45s
- 单位：秒

14.5.1.0.10 compatible-kill-query

- 设置 KILL 语句的兼容性。
- 默认值：false
- compatible-kill-query 仅在enable-global-kill 为 false 时生效。
- 当enable-global-kill 为 false 时，compatible-kill-query 控制杀死一条查询时是否需要加上 TIDB 关键词。
  - compatible-kill-query 为 false 时，TiDB 中 KILL xxx 的行为和 MySQL 中的行为不同。为杀死一条查询，在 TiDB 中需要加上 TIDB 关键词，即 KILL TIDB xxx。
  - compatible-kill-query 为 true 时，为杀死一条查询，在 TiDB 中无需加上 TIDB 关键词。强烈不建议设置 compatible-kill-query 为 true，除非你确定客户端将始终连接到同一个 TiDB 节点。这是因为当你在默认的 MySQL 客户端按下 Control+C 时，客户端会开启一个新连接，并在这个新连接中执行 KILL 语句。此时，如果客户端和 TiDB 之间存在代理，新连接可能会被路由到其他 TiDB 节点，从而错误地终止其他会话。
- 当enable-global-kill 为 true 时，KILL xxx 和 KILL TIDB xxx 的作用相同，但是暂不支持 Control+C 终止查询。
- 关于 KILL 语句的更多信息，请参考KILL [TIDB]。

14.5.1.0.11 check-mb4-value-in-utf8

- 开启检查 utf8mb4 字符的开关，如果开启此功能，字符集是 utf8，且在 utf8 插入 mb4 字符，系统将会报错。
- 默认值：true
- 自 v6.1.0 起，utf8mb4 字符检查改为通过 TiDB 配置项 instance.tidb_check_mb4_value_in_utf8 或系统变量 tidb_check_mb4_value_in_utf8 进行设置。check-mb4-value-in-utf8 仍可使用，但如果同时设置了 check-mb4-value-in-utf8 与 instance.tidb_check_mb4_value_in_utf8，TiDB 将采用 instance.tidb_check_mb4_value_in_utf8 的值。

14.5.1.0.12 treat-old-version-utf8-as-utf8mb4

- 将旧表中的 utf8 字符集当成 utf8mb4 的开关。
- 默认值：true

14.5.1.0.13 alter-primary-key（已废弃）

- 用于控制添加或者删除主键功能。
- 默认值：false
- 默认情况下，不支持增删主键。将此变量被设置为 true 后，支持增删主键功能。不过对在此开关开启前已经存在的表，且主键是整型类型时，即使之后开启此开关也不支持对此列表删除主键。

> 注意：
>
> 该配置项已被废弃，目前仅在 `@@tidb_enable_clustered_index` 取值为 `INT_ONLY` 时生效。如果需要增删主键，请在建表时使用 `NONCLUSTERED` 关键字代替。要了解关于 `CLUSTERED` 主键的详细信息，请参考聚簇索引。

14.5.1.0.14 server-version

- 用来修改 TiDB 在以下情况下返回的版本号：
  - 当使用内置函数 `VERSION()` 时。
  - 当与客户端初始连接，TiDB 返回带有服务端版本号的初始握手包时。具体可以查看 MySQL 初始握手包的描述。
- 默认值：“”
- 默认情况下，TiDB 版本号格式为：`5.7.${mysql_latest_minor_version}-TiDB-${tidb_version}`。

> 注意：
>
> `server-version` 的值会被 TiDB 节点用于验证当前 TiDB 的版本。因此在进行 TiDB 集群升级前，请将 `server-version` 的值设置为空或者当前 TiDB 真实的版本值，避免出现非预期行为。

14.5.1.0.15 repair-mode

- 用于开启非可信修复模式，启动该模式后，可以过滤 `repair-table-list` 名单中坏表的加载。
- 默认值：false
- 默认情况下，不支持修复语法，默认启动时会加载所有表信息。

14.5.1.0.16 repair-table-list

- 配合 `repair-mode` 为 true 时使用，用于列出实例中需要修复的坏表的名单，该名单的写法为 [“db.table1”, “db.table2”, ……]。
- 默认值：[]
- 默认情况下，该 list 名单为空，表示没有所需修复的坏表信息。

14.5.1.0.17 new_collations_enabled_on_first_bootstrap

- 用于开启新的 collation 支持
- 默认值：true
- 注意：该配置项只有在初次初始化集群时生效，初始化集群后，无法通过更改该配置项打开或关闭新的 collation 框架。

14.5.1.0.18 max-server-connections

- TiDB 中同时允许的最大客户端连接数，用于资源控制。
- 默认值：0
- 默认情况下，TiDB 不限制客户端连接数。当本配置项的值大于 0 且客户端连接数到达此值时，TiDB 服务端将会拒绝新的客户端连接。
- 自 v6.2.0 起，客户端连接数已改用配置项instance.max_connections 或系统变量max_connections 进行设置。max-server-connections 仍可使用，但如果同时设置了 max-server-connections 与 instance.
  ↪ max_connections，TiDB 将采用 instance.max_connections 的值。

14.5.1.0.19 max-index-length

- 用于设置新建索引的长度限制。
- 默认值：3072
- 单位：Byte
- 目前的合法值范围 [3072，3072*4]。MySQL 和 TiDB v3.0.11 之前版本（不包含 v3.0.11）没有此配置项，不过都对新建索引的长度做了限制。MySQL 对此的长度限制为 3072，TiDB 在 v3.0.7 以及之前版本该值为 3072*4，在 v3.0.7 之后版本（包含 v3.0.8、v3.0.9 和 v3.0.10）的该值为 3072。为了与 MySQL 和 TiDB 之前版本的兼容，添加了此配置项。

14.5.1.0.20 table-column-count-limit 从 v5.0 版本开始引入

- 用于设置单个表中列的数量限制
- 默认值：1017
- 目前的合法值范围 [1017，4096]。

14.5.1.0.21 index-limit 从 v5.0 版本开始引入

- 用于设置单个表中索引的数量限制
- 默认值：64
- 目前的合法值范围 [64，512]。

14.5.1.0.22 enable-telemetry 从 v4.0.2 版本开始引入

- 是否开启 TiDB 遥测功能。
- 默认值：false

- 如果在 TiDB 实例上该配置项设为 `true`，该 TiDB 实例上将开启遥测功能，且`tidb_enable_telemetry` 系统变量生效。
- 如果所有 TiDB 实例上该选项都设置为 `false`，那么将完全禁用 TiDB 遥测功能，且忽略 `tidb_enable_telemetry` 系统变量。参阅遥测了解该功能详情。

14.5.1.0.23 `enable-tcp4-only` 从 v5.0 版本开始引入

- 控制是否只监听 TCP4。
- 默认值：false
- 当使用 LVS 为 TiDB 做负载均衡时，可开启此配置项。这是因为 LVS 的 TOA 模块可以通过 TCP4 协议从 TCP 头部信息中解析出客户端的真实 IP。

14.5.1.0.24 `enable-enum-length-limit` 从 v5.0 版本开始引入

- 是否限制单个 ENUM 元素和单个 SET 元素的最大长度
- 默认值：true
- 当该配置项值为 `true` 时，ENUM 和 SET 单个元素的最大长度为 255 个字符，与 MySQL 8 兼容；当该配置项值为 `false` 时，不对单个元素的长度进行限制，与 TiDB v5.0 之前的版本兼容。

14.5.1.0.25 `graceful-wait-before-shutdown` 从 v5.0 版本开始引入

- 指定关闭服务器时 TiDB 等待的秒数，使得客户端有时间断开连接。
- 默认值：0
- 在 TiDB 等待服务器关闭期间，HTTP 状态会显示失败，使得负载均衡器可以重新路由流量。

14.5.1.0.26 `enable-global-kill` 从 v6.1.0 版本开始引入

- 用于开启 Global Kill（跨节点终止查询或连接）功能。
- 默认值：true
- 当该配置项值为 `true` 时，`KILL` 语句和 `KILL TIDB` 语句均能跨节点终止查询或连接，无需担心错误地终止其他查询或连接。当你使用客户端连接到任何一个 TiDB 节点执行 `KILL` 语句或 `KILL TIDB` 语句时，该语句会被转发给对应的 TiDB 节点。当客户端和 TiDB 中间有代理时，`KILL` 语句或 `KILL TIDB` 语句也会被转发给对应的 TiDB 节点执行。目前暂时不支持在 `enable-global-kill` 为 `true` 时用 MySQL 命令行 ctrl+c 终止查询或连接。关于 `KILL` 语句的更多信息，请参考KILL [TIDB]。

14.5.1.0.27 `initialize-sql-file` 从 v6.6.0 版本开始引入

- 用于指定 TiDB 集群初次启动时执行的 SQL 脚本。
- 默认值：“”
- 脚本中的所有 SQL 语句将以最高权限执行，不会进行权限检查。若指定的 SQL 脚本无法正确执行，可能导致 TiDB 集群启动失败。
- 通常用于修改系统变量的值、创建用户或分配权限等。

14.5.1.0.28 enable-forwarding 从 v5.0.0 版本开始引入

- 控制 TiDB 中的 PD client 以及 TiKV client 在疑似网络隔离的情况下是否通过 follower 将请求转发给 leader。
- 默认值：false
- 如果确认环境存在网络隔离的可能，开启这个参数可以减少服务不可用的窗口期。
- 如果无法准确判断隔离、网络中断、宕机等情况，这个机制存在误判情况从而导致可用性、性能降低。如果网络中从未发生过网络故障，不推荐开启此选项。

14.5.1.0.29 enable-table-lock 从 v4.0.0 版本开始引入

> 警告：
>
> 表级锁 (Table Lock) 为实验特性，不建议在生产环境中使用。

- 控制是否开启表级锁特性。
- 默认值：false
- 表级锁用于协调多个 session 之间对同一张表的并发访问。目前已支持的锁种类包括 READ、WRITE 和 WRITE LOCAL。当该配置项为 false 时，执行 LOCK TABLES 和 UNLOCK TABLES 语句不会生效，并且会报“LOCK/UNLOCK TABLES is not supported”的警告。更多信息，请参考LOCK TABLES 和 UNLOCK TABLES。

14.5.1.0.30 labels

- 指定服务器标签，例如 `{ zone = "us-west-1", dc = "dc1", rack = "rack1", host = "tidb1" }`。
- 默认值：`{}`

> 注意：
>
> - 标签 zone 在 TiDB 中具有特殊用途，用于指定服务器所在的区域信息，当设置 zone 为非空值时，对应的值会被自动用于txn-score 和Follower read 等功能。
> - 标签 group 在 TiDB Operator 中具有特殊用途。对于使用TiDB Operator 部署的集群，建议不要手动指定此标签。

14.5.1.1 log

日志相关的配置项。

14.5.1.1.1 level

- 指定日志的输出级别，可选项为 [debug, info, warn, error, fatal]
- 默认值：“info”

14.5.1.1.2 format

- 指定日志输出的格式，可选项为 [json, text]。
- 默认值：“text”

14.5.1.1.3 enable-timestamp

- 是否在日志中输出时间戳。
- 默认值：null
- 如果设置为 false，那么日志里面将不会输出时间戳。

> 注意：
>
> - 考虑后向兼容性，原来的配置项 disable-timestamp 仍然有效，但如果和 enable-timestamp 配置的值在语义上冲突（例如在配置中把 enable-timestamp 和 disable-timestamp 同时设置为 true），则 TiDB 会忽略 disable-timestamp 的值。
> - 当前 TiDB 默认使用 disable-timestamp 来决定是否在日志中输出时间戳，此时 enable-timestamp 的值为 null。
> - 在未来的版本中，disable-timestamp 配置项将被彻底移除，请废弃 disable-timestamp 的用法，使用语义上更易于理解的 enable-timestamp。

14.5.1.1.4 enable-slow-log

- 是否开启慢查询日志
- 默认值：true
- 可以设置成 true 或 false 来启用或禁用慢查询日志。
- 自 v6.1.0 起，已改用配置项 instance.tidb_enable_slow_log 或系统变量 tidb_enable_slow_log 来设置是否开启慢查询日志。enable-slow-log 仍可使用，但如果同时设置了 enable-slow-log 与 instance.tidb_enable_slow_log，TiDB 将采用 instance.tidb_enable_slow_log 的值。

14.5.1.1.5 slow-query-file

- 慢查询日志的文件名。
- 默认值：“tidb-slow.log”。注：由于 TiDB V2.1.8 更新了慢日志格式，所以将慢日志单独输出到了慢日志文件。V2.1.8 之前的版本，该变量的默认值是 “”。
- 设置后，慢查询日志会单独输出到该文件。

14.5.1.1.6 slow-threshold

- 输出慢日志的耗时阈值。
- 默认值：300

- 单位：毫秒
- 当查询大于这个值，就会当做是一个慢查询，输出到慢查询日志。
- 自 v6.1.0 起，已改用配置项 instance.tidb_slow_log_threshold 或系统变量 tidb_slow_log_threshold 来设置输出慢日志的耗时阈值。slow-threshold 仍可使用，但如果同时设置了 slow-threshold 与 instance.tidb_slow_log_threshold，TiDB 将采用 instance.tidb_slow_log_threshold 的值。

14.5.1.1.7 record-plan-in-slow-log

- 在慢日志中记录执行计划
- 默认值：1
- 自 v6.1.0 起，已改用配置项 instance.tidb_record_plan_in_slow_log 或系统变量 tidb_record_plan_in_slow_log 来设置在慢日志中记录执行计划。record-plan-in-slow-log 仍可使用，但如果同时设置了 record-plan-in-slow-log 与 instance.tidb_record_plan_in_slow_log，TiDB 将采用 instance.tidb_record_plan_in_slow_log 的值。

14.5.1.1.8 expensive-threshold

- 输出 expensive 操作的行数阈值。
- 默认值：10000
- 当查询的行数（包括中间结果，基于统计信息）大于这个值，该操作会被认为是 expensive 查询，并输出一个前缀带有 [EXPENSIVE_QUERY] 的日志。

14.5.1.2 log.file

日志文件相关的配置项。

filename

- 一般日志文件名字。
- 默认值：“”
- 如果设置，会输出一般日志到这个文件。

max-size

- 日志文件的大小限制。
- 默认值：300
- 单位：MB
- 最大设置上限为 4096。

max-days

- 日志最大保留的天数。
- 默认值：0
- 默认不清理；如果设置了参数值，在 max-days 之后 TiDB 会清理过期的日志文件。

`max-backups`

- 保留的日志的最大数量。
- 默认值：0
- 默认全部保存；如果设置为 7，会最多保留 7 个老的日志文件。

14.5.1.3 security

安全相关配置。

14.5.1.3.1 enable-sem

- 启用安全增强模式 (SEM)。
- 默认值：false
- 可以通过系统变量tidb_enable_enhanced_security 获取安全增强模式的状态。

14.5.1.3.2 ssl-ca

- PEM 格式的受信任 CA 的证书文件路径。
- 默认值：“”
- 当同时设置了该选项和 --ssl-cert、--ssl-key 选项时，TiDB 将在客户端出示证书的情况下根据该选项指定的受信任的 CA 列表验证客户端证书。若验证失败，则连接会被终止。
- 即使设置了该选项，若客户端没有出示证书，则安全连接仍然继续，不会进行客户端证书验证。

14.5.1.3.3 ssl-cert

- PEM 格式的 SSL 证书文件路径。
- 默认值：“”
- 当同时设置了该选项和 --ssl-key 选项时，TiDB 将接受（但不强制）客户端使用 TLS 安全地连接到 TiDB。
- 若指定的证书或私钥无效，则 TiDB 会照常启动，但无法接受安全连接。

14.5.1.3.4 ssl-key

- PEM 格式的 SSL 证书密钥文件路径，即 --ssl-cert 所指定的证书的私钥。
- 默认值：“”
- 目前 TiDB 不支持加载由密码保护的私钥。

14.5.1.3.5 cluster-ssl-ca

- CA 根证书，用于用 tls 连接 TiKV/PD
- 默认值：“”

14.5.1.3.6 cluster-ssl-cert

- ssl 证书文件路径，用于用 tls 连接 TiKV/PD
- 默认值：“”

14.5.1.3.7 cluster-ssl-key

- ssl 私钥文件路径，用于用 tls 连接 TiKV/PD
- 默认值：“”

14.5.1.3.8 spilled-file-encryption-method

- 内存落盘文件的加密方式。
- 默认值："plaintext"，表示不进行加密。
- 可选值："plaintext"、"aes128-ctr"。

14.5.1.3.9 auto-tls

- 控制 TiDB 启动时是否自动生成 TLS 证书。
- 默认值：false

14.5.1.3.10 tls-version

- 设置用于连接 MySQL 协议的最低 TLS 版本。
- 默认值：“”，支持 TLSv1.1 及以上版本。
- 可选值："TLSv1.0"、"TLSv1.1"、"TLSv1.2" 和 "TLSv1.3"

14.5.1.3.11 auth-token-jwks 从 v6.4.0 版本开始引入

> **警告：**
>
> tidb_auth_token 认证方式仅用于 TiDB Cloud 内部实现，不要修改该配置。

- 设置 tidb_auth_token 认证方式的 JSON Web Key Sets (JWKS) 的本地文件路径。
- 默认值：“”

14.5.1.3.12 auth-token-refresh-interval 从 v6.4.0 版本开始引入

> **警告：**
>
> tidb_auth_token 认证方式仅用于 TiDB Cloud 内部实现，不要修改该配置。

- 设置 tidb_auth_token 认证方式的 JWKS 刷新时间间隔。
- 默认值：1h

14.5.1.3.13 disconnect-on-expired-password 从 v6.5.0 版本开始引入

- 对于密码已过期的用户，通过 disconnect-on-expired-password 控制 TiDB 服务端是否直接断开该用户的连接。
- 默认值：true
- 默认值为 “true” 表示 TiDB 服务端将直接断开密码已过期用户的连接。设置为 “false” 时，TiDB 服务端将密码已过期用户的连接置于 “沙盒模式”，允许该用户建立连接并执行密码重置操作。

14.5.1.3.14 session-token-signing-cert 从 v6.4.0 版本开始引入

> **警告：**
>
> 该配置与一个未发布的特性相关。请勿设置该配置。

- 默认值：“”

14.5.1.3.15 session-token-signing-key 从 v6.4.0 版本开始引入

> **警告：**
>
> 该配置与一个未发布的特性相关。请勿设置该配置。

- 默认值：“”

14.5.1.4 performance

性能相关配置。

14.5.1.4.1 max-procs

- TiDB 的 CPU 使用数量。
- 默认值：0
- 默认值为 0 表示使用机器上所有的 CPU；如果设置成 n，那么 TiDB 会使用 n 个 CPU 数量。

14.5.1.4.2 server-memory-quota 从 v4.0.9 版本开始引入

> **警告：**
>
> 自 v6.5.0 起，该配置项被废弃。请使用tidb_server_memory_limit 系统变量进行设置。

- 设置 tidb-server 实例的最大内存用量，单位为字节。
- 默认值：0
- 默认值为 0 表示无内存限制。

14.5.1.4.3 txn-entry-size-limit 从 v5.0 版本开始引入

- TiDB 单行数据的大小限制
- 默认值：6291456
- 单位：Byte
- 事务中单个 key-value 记录的大小限制。若超出该限制，TiDB 将会返回 entry too large 错误。该配置项的最大值不超过 125829120（表示 120MB）。
- 注意，TiKV 有类似的限制。若单个写入请求的数据量大小超出raft-entry-max-size，默认为 8MB，TiKV 会拒绝处理该请求。当表的一行记录较大时，需要同时修改这两个配置。
- max_allowed_packet (MySQL 协议的最大数据包大小) 的默认值为 67108864（64 MiB）。如果一行记录的大小超过 max_allowed_packet，该行记录会被截断。
- txn-total-size-limit（TiDB 单个事务大小限制）的默认值为 100 MiB。如果将 txn-entry-size-limit 的值设置为 100 MiB 以上，需要相应地调大 txn-total-size-limit 的值。

14.5.1.4.4 txn-total-size-limit

- TiDB 单个事务大小限制
- 默认值：104857600
- 单位：Byte
- 单个事务中，所有 key-value 记录的总大小不能超过该限制。该配置项的最大值不超过 1099511627776（表示 1TB）。注意，如果使用了以 Kafka 为下游消费者的 binlog，如：arbiter 集群，该配置项的值不能超过 1073741824（表示 1GB），因为这是 Kafka 的处理单条消息的最大限制，超过该限制 Kafka 将会报错。
- 在 v6.5.0 及之后的版本中，不再推荐使用该配置项，事务的内存大小会被累计计入所在会话的内存使用量中，并由tidb_mem_quota_query 变量在单个会话内存超阈值时采取控制行为。为了向前兼容，由低版本升级至 v6.5.0 及更高版本时，该配置项的行为如下所述:

  - 若该配置项未设置，或设置为默认值 (104857600)，升级后事务内存大小将会计入所在会话的内存使用中，由 `tidb_mem_quota_query` 变量控制。
  - 若该配置项未设为默认值 (104857600)，升级前后该配置项仍生效，对单个事务大小的限制行为不会发生变化，事务内存大小不由 `tidb_mem_quota_query` 控制。

14.5.1.4.5 max-txn-ttl

- 单个事务持锁的最长时间，超过该时间，该事务的锁可能会被其他事务清除，导致该事务无法成功提交。
- 默认值：3600000
- 单位：毫秒
- 超过此时间的事务只能执行提交或者回滚，提交不一定能够成功。

14.5.1.4.6 stmt-count-limit

- TiDB 单个事务允许的最大语句条数限制。
- 默认值：5000
- 在一个事务中，超过 `stmt-count-limit` 条语句后还没有 rollback 或者 commit，TiDB 将会返回 `statement count 5001 exceeds the transaction limitation, autocommit = false` 错误。该限制只在可重试的乐观事务中生效，如果使用悲观事务或者关闭了事务重试，事务中的语句数将不受此限制。

14.5.1.4.7 tcp-keep-alive

- TiDB 在 TCP 层开启 keepalive。
- 默认值：true

14.5.1.4.8 tcp-no-delay

- 控制 TiDB 是否在 TCP 层开启 TCP_NODELAY。开启后，TiDB 将禁用 TCP/IP 协议中的 Nagle 算法，允许小数据包的发送，可以降低网络延时，适用于延时敏感型且数据传输量比较小的应用。
- 默认值：true

14.5.1.4.9 cross-join

- 默认值：true
- 默认可以执行在做 join 时两边表没有任何条件（where 字段）的语句；如果设置为 false，则有这样的 join 语句出现时，server 会拒绝执行

14.5.1.4.10 stats-lease

- TiDB 重载统计信息，更新表行数，检查是否需要自动 analyze，利用 feedback 更新统计信息以及加载列的统计信息的时间间隔。
- 默认值：3s

  - 每隔 stats-lease 时间，TiDB 会检查统计信息是否有更新，如果有会将其更新到内存中
  - 每隔 20 * stats-lease 时间，TiDB 会将 DML 产生的总行数以及修改的行数变化更新到系统表中
  - 每隔 stats-lease 时间，TiDB 会检查是否有表或者索引需要自动 analyze
  - 每隔 stats-lease 时间，TiDB 会检查是否有列的统计信息需要被加载到内存中
  - 每隔 200 * stats-lease 时间，TiDB 会将内存中缓存的 feedback 写入系统表中
  - 每隔 5 * stats-lease 时间，TiDB 会读取系统表中的 feedback，更新内存中缓存的统计信息
- 当 stats-lease 为 0s 时，TiDB 会以 3s 的时间间隔周期性的读取系统表中的统计信息并更新内存中缓存的统计信息。但不会自动修改统计信息相关系统表，具体来说，TiDB 不再自动修改这些表：
  - mysql.stats_meta：TiDB 不再自动记录事务中对某张表的修改行数，也不会更新到这个系统表中
  - mysql.stats_histograms/mysql.stats_buckets 和 mysql.stats_top_n：TiDB 不再自动 analyze 和主动更新统计信息
  - mysql.stats_feedback：TiDB 不再根据被查询的数据反馈的部分统计信息更新表和索引的统计信息

14.5.1.4.11 pseudo-estimate-ratio

- 修改过的行数/表的总行数的比值，超过该值时系统会认为统计信息已经过期，会采用 pseudo 的统计信息。
- 默认值：0.8
- 最小值：0
- 最大值：1

14.5.1.4.12 force-priority

- 把所有的语句优先级设置为 force-priority 的值。
- 默认值：NO_PRIORITY
- 可选值：默认值 NO_PRIORITY 表示不强制改变执行语句的优先级，其它优先级从低到高可设置为 LOW_PRIORITY、DELAYED 或 HIGH_PRIORITY。
- 自 v6.1.0 起，已改用配置项instance.tidb_force_priority 或系统变量tidb_force_priority 来将所有语句优先级设为 force-priority 的值。force-priority 仍可使用，但如果同时设置了 force-priority 与 instance.tidb_force_priority，TiDB 将采用 instance.tidb_force_priority 的值。

14.5.1.4.13 distinct-agg-push-down

- 设置优化器是否执行将带有 Distinct 的聚合函数（比如 select count(distinct a)from t）下推到 Coprocessor 的优化操作。
- 默认值：false
- 该变量作为系统变量tidb_opt_distinct_agg_push_down 的初始值。

14.5.1.4.14 enforce-mpp

- 用于控制是否忽略优化器代价估算，强制使用 TiFlash 的 MPP 模式执行查询。
- 默认值：false
- 该配置项可以控制系统变量tidb_enforce_mpp 的初始值。例如，当设置该配置项为 true 时，tidb_enforce_mpp 的默认值为 ON。

14.5.1.4.15 stats-load-concurrency 从 v5.4.0 版本开始引入

> **警告：**
>
> 统计信息同步加载功能目前为实验性特性，不建议在生产环境中使用。

- TiDB 统计信息同步加载功能可以并发处理的最大列数
- 默认值：5
- 目前的合法值范围：[1, 128]

14.5.1.4.16 stats-load-queue-size 从 v5.4.0 版本开始引入

> **警告：**
>
> 统计信息同步加载功能目前为实验性特性，不建议在生产环境中使用。

- 用于设置 TiDB 统计信息同步加载功能最多可以缓存多少列的请求
- 默认值：1000
- 目前的合法值范围：[1, 100000]

14.5.1.4.17 enable-stats-cache-mem-quota 从 v6.1.0 版本开始引入

> **警告：**
>
> 该变量为实验特性，不推荐在生产环境中使用。

- 用于控制 TiDB 是否开启统计信息缓存的内存上限。
- 默认值：false

14.5.1.5 opentracing

opentracing 的相关的设置。

14.5.1.5.1 enable

- 开启 opentracing 跟踪 TiDB 部分组件的调用开销。注意开启后会有一定的性能损失。
- 默认值：false

14.5.1.5.2 rpc-metrics

- 开启 rpc metrics。
- 默认值：false

14.5.1.6 opentracing.sampler

opentracing.sampler 相关的设置。

14.5.1.6.1 type

- opentracing 采样器的类型。字符串取值大小写不敏感。
- 默认值：“const”
- 可选值：“const”，“probabilistic”，“ratelimiting”，remote”

14.5.1.6.2 param

- 采样器参数。
  - 对于 const 类型，可选值为 0 或 1，表示是否开启。
  - 对于 probabilistic 类型，参数为采样概率，可选值为 0 到 1 之间的浮点数。
  - 对于 ratelimiting 类型，参数为每秒采样 span 的个数。
  - 对于 remote 类型，参数为采样概率，可选值为 0 到 1 之间的浮点数。
- 默认值：1.0

14.5.1.6.3 sampling-server-url

- jaeger-agent 采样服务器的 HTTP URL 地址。
- 默认值：“”

14.5.1.6.4 max-operations

- 采样器可追踪的最大操作数。如果一个操作没有被追踪，会启用默认的 probabilistic 采样器。
- 默认值：0

14.5.1.6.5 sampling-refresh-interval

- 控制远程轮询 jaeger-agent 采样策略的频率。
- 默认值：0

14.5.1.7 opentracing.reporter

opentracing.reporter 相关的设置。

14.5.1.7.1 queue-size

- reporter 在内存中记录 spans 个数的队列容量。
- 默认值：0

14.5.1.7.2 buffer-flush-interval

- reporter 缓冲区的刷新频率。
- 默认值：0

14.5.1.7.3 log-spans

- 是否为所有提交的 span 打印日志。
- 默认值：false

14.5.1.7.4 local-agent-host-port

- reporter 向 jaeger-agent 发送 span 的地址。
- 默认值：“”

14.5.1.8 tikv-client

14.5.1.8.1 grpc-connection-count

- 跟每个 TiKV 之间建立的最大连接数。
- 默认值：4

14.5.1.8.2 grpc-keepalive-time

- TiDB 与 TiKV 节点之间 rpc 连接 keepalive 时间间隔，如果超过该值没有网络包，grpc client 会 ping 一下 TiKV 查看是否存活。
- 默认值：10
- 单位：秒

14.5.1.8.3 grpc-keepalive-timeout

- TiDB 与 TiKV 节点 rpc keepalive 检查的超时时间
- 默认值：3
- 单位：秒

14.5.1.8.4 grpc-compression-type

- 控制 TiDB 向 TiKV 节点传输数据使用的压缩算法类型。默认值为 “none” 即不压缩。修改为 “gzip” 可以使用 gzip 算法压缩数据。
- 默认值：“none”
- 可选值：“none”，“gzip”

14.5.1.8.5 commit-timeout

- 执行事务提交时，最大的超时时间。
- 默认值：41s
- 这个值必须是大于两倍 Raft 选举的超时时间。

14.5.1.8.6 max-batch-size

- 批量发送 rpc 封包的最大数量，如果不为 0，将使用 BatchCommands api 发送请求到 TiKV，可以在并发度高的情况降低 rpc 的延迟，推荐不修改该值。
- 默认值：128

14.5.1.8.7 max-batch-wait-time

- 等待 max-batch-wait-time 纳秒批量将此期间的数据包封装成一个大包发送给 TiKV 节点，仅在 tikv-client.max-batch-size 值大于 0 时有效，不推荐修改该值。
- 默认值：0
- 单位：纳秒

14.5.1.8.8 batch-wait-size

- 批量向 TiKV 发送的封包最大数量，不推荐修改该值。
- 默认值：8
- 若此值为 0 表示关闭此功能。

14.5.1.8.9 overload-threshold

- TiKV 的负载阈值，如果超过此阈值，会收集更多的 batch 封包，来减轻 TiKV 的压力。仅在 tikv-client.max-batch-size 值大于 0 时有效，不推荐修改该值。
- 默认值：200

14.5.1.9 tikv-client.copr-cache 从 v4.0.0 版本开始引入

本部分介绍 Coprocessor Cache 相关的配置项。

14.5.1.9.1 capacity-mb

- 缓存的总数据量大小。当缓存空间满时，旧缓存条目将被逐出。值为 0.0 时表示关闭 Coprocessor Cache。
- 默认值：1000.0
- 单位：MB
- 类型：Float

14.5.1.10 binlog

TiDB Binlog 相关配置。

14.5.1.10.1 enable

- binlog 开关。
- 默认值：false

14.5.1.10.2 write-timeout

- 写 binlog 的超时时间，推荐不修改该值。
- 默认值：15s
- 单位：秒

14.5.1.10.3 ignore-error

- 忽略写 binlog 发生的错误时处理开关，推荐不修改该值。
- 默认值：false
- 如果设置为 `true`，发生错误时，TiDB 会停止写入 binlog，并且在监控项 `tidb_server_critical_error_total` 上计数加 1；如果设置为 `false`，写入 binlog 失败，会停止整个 TiDB 的服务。

14.5.1.10.4 binlog-socket

- binlog 输出网络地址。
- 默认值：“”

14.5.1.10.5 strategy

- binlog 输出时选择 pump 的策略，仅支持 hash，range 方法。
- 默认值：“range”

14.5.1.11 status

TiDB 服务状态相关配置。

14.5.1.11.1 report-status

- 开启 HTTP API 服务的开关。
- 默认值：true

14.5.1.11.2 record-db-qps

- 输出与 database 相关的 QPS metrics 到 Prometheus 的开关。
- 默认值：false

14.5.1.12 pessimistic-txn

悲观事务使用方法请参考TiDB 悲观事务模式。

14.5.1.12.1 max-retry-count

- 悲观事务中单个语句最大重试次数，重试次数超过该限制，语句执行将会报错。
- 默认值：256

14.5.1.12.2 deadlock-history-capacity

- 单个 TiDB 节点的INFORMATION_SCHEMA.DEADLOCKS 表最多可记录的死锁事件个数。当表的容量已满时，如果再次发生死锁错误，最早的一次死锁错误的信息将从表中移除。
- 默认值：10
- 最小值：0
- 最大值：10000

14.5.1.12.3 deadlock-history-collect-retryable

- 控制INFORMATION_SCHEMA.DEADLOCKS 表中是否收集可重试的死锁错误信息。详见 DEADLOCKS 表文档的可重试的死锁错误小节。
- 默认值：false

14.5.1.12.4 pessimistic-auto-commit

- 用来控制开启全局悲观事务模式下 (tidb_txn_mode='pessimistic') 时，自动提交的事务使用的事务模式。默认情况下，即使开启全局悲观事务模式，自动提交事务依然使用乐观事务模式来执行。当开启该配置项后（设置为 true），在全局悲观事务模式下，自动提交事务将也使用悲观事务模式执行。行为与其他显式提交的悲观事务相同。
- 对于存在冲突的场景，开启本开关可以将自动提交事务纳入全局等锁管理中，从而避免死锁，改善冲突造成死锁带来的时延尖刺。
- 对于不存在冲突的场景，如果有大量自动提交事务（例如自动提交事务数量占业务数量的比例超过一半甚至更多，需要根据实际情况分析）且单个事务操作数据量较大的情况下，开启该配置项会造成性能回退。例如，自动提交的 INSERT INTO SELECT 语句。
- 默认值：false

14.5.1.12.5 constraint-check-in-place-pessimistic 从 v6.4.0 版本开始引入

- 用来控制系统变量tidb_constraint_check_in_place_pessimistic 的默认值。
- 默认值：true

14.5.1.13 isolation-read

读取隔离相关的配置项。

14.5.1.13.1 engines

- 用于控制 TiDB 节点允许从哪种类型的引擎读取数据。
- 默认值：[“tikv”，“tiflash”，“tidb”]，表示由优化器自动选择存储引擎。
- 可选值：“tikv”，“tiflash”，“tidb” 的组合，如：[“tikv”，“tidb”]、[“tiflash”，“tidb”]。

14.5.1.14 instance

14.5.1.14.1 tidb_enable_collect_execution_info

- 用于控制是否同时将各个执行算子的执行信息记录入 slow query log 中。
- 默认值：true
- 在 v6.1.0 之前，该功能通过配置项 enable-collect-execution-info 进行设置。

14.5.1.14.2 tidb_enable_slow_log

- 是否开启慢查询日志。
- 默认值：true
- 可以设置成 true 或 false 来启用或禁用慢查询日志。
- 在 v6.1.0 之前，该功能通过配置项 enable-slow-log 进行设置。

14.5.1.14.3 tidb_slow_log_threshold

- 输出慢日志的耗时阈值。
- 默认值：300
- 范围：[-1, 9223372036854775807]
- 单位：毫秒
- 当查询大于这个值，就会当做是一个慢查询，输出到慢查询日志。
- 在 v6.1.0 之前，该功能通过配置项 slow-threshold 进行设置。

14.5.1.14.4 tidb_record_plan_in_slow_log

- 在慢日志中记录执行计划。
- 默认值：1
- 0 表示关闭，1 表示开启，默认开启，该值作为系统变量tidb_record_plan_in_slow_log 的初始值。
- 在 v6.1.0 之前，该功能通过配置项 record-plan-in-slow-log 进行设置。

14.5.1.14.5 tidb_force_priority

- 把所有的语句优先级设置为系统变量 `tidb_force_priority` 的值。
- 默认值：NO_PRIORITY
- 默认值 NO_PRIORITY 表示不强制改变执行语句的优先级，其它优先级从低到高可设置为 LOW_PRIORITY、DELAYED 或 HIGH_PRIORITY。
- 在 v6.1.0 之前，该功能通过配置项 `force-priority` 进行设置。

14.5.1.14.6 max_connections

- TiDB 中同时允许的最大客户端连接数，用于资源控制。
- 默认值：0
- 取值范围：`[0, 100000]`
- 默认情况下，TiDB 不限制客户端连接数。当本配置项的值大于 0 且客户端连接数到达此值时，TiDB 服务端将会拒绝新的客户端连接。
- 该值作为系统变量`max_connections` 的初始值。
- 在 v6.2.0 之前，该功能通过配置项 `max-server-connections` 进行设置。

14.5.1.14.7 tidb_enable_ddl

- 用于表示该 tidb-server 是否可以成为 DDL owner。
- 默认值：true
- 该值作为系统变量`tidb_enable_ddl` 的初始值。
- 在 v6.3.0 之前，该功能由配置项 `run-ddl` 进行设置。

14.5.1.14.8 tidb_stmt_summary_enable_persistent 从 v6.6.0 版本开始引入

> **警告：**
>
> statements summary 持久化目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现 bug，请在 GitHub 上提 issue 反馈。

- 用于控制是否开启 statements summary 持久化。
- 默认值：false
- 详情参考持久化 statements summary。

14.5.1.14.9 tidb_stmt_summary_filename 从 v6.6.0 版本开始引入

> **警告：**
>
> statements summary 持久化目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现 bug，请在 GitHub 上提 issue 反馈。

- 当开启了 statements summary 持久化时，该配置用于指定持久化数据所写入的文件。
- 默认值：“tidb-statements.log”

14.5.1.14.10 `tidb_stmt_summary_file_max_days` 从 v6.6.0 版本开始引入

> **警告：**
>
> statements summary 持久化目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现 bug，请在 GitHub 上提 issue 反馈。

- 当开启了 statements summary 持久化时，该配置用于指定持久化数据文件所保留的最大天数。
- 默认值：3
- 单位：天
- 可结合数据保留时长需求与磁盘空间占用适当调整。

14.5.1.14.11 `tidb_stmt_summary_file_max_size` 从 v6.6.0 版本开始引入

> **警告：**
>
> statements summary 持久化目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现 bug，请在 GitHub 上提 issue 反馈。

- 当开启了 statements summary 持久化时，该配置用于限制持久化数据单个文件的大小。
- 默认值：64
- 单位：MiB
- 可结合数据保留时长需求与磁盘空间占用适当调整。

14.5.1.14.12 `tidb_stmt_summary_file_max_backups` 从 v6.6.0 版本开始引入

> **警告：**
>
> statements summary 持久化目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现 bug，请在 GitHub 上提 issue 反馈。

- 当开启了 statements summary 持久化时，该配置用于限制持久化数据文件最大数量，0 表示不限制。
- 默认值：0
- 可结合数据保留时长需求与磁盘空间占用适当调整。

14.5.1.15 proxy-protocol

PROXY 协议相关的配置项。

14.5.1.15.1 networks

- 允许使用 PROXY 协议连接 TiDB 的代理服务器地址列表。
- 默认值：“”
- 通常情况下，通过反向代理使用 TiDB 时，TiDB 会将反向代理服务器的 IP 地址视为客户端 IP 地址。对于支持 PROXY 协议的反向代理（如 HAProxy），开启 PROXY 协议后能让反向代理透传客户端真实的 IP 地址给 TiDB。
- 配置该参数后，TiDB 将允许配置的源 IP 地址使用 PROXY 协议连接到 TiDB，且拒绝这些源 IP 地址使用非 PROXY 协议连接。若该参数为空，则任何源 IP 地址都不能使用 PROXY 协议连接到 TiDB。地址可以使用 IP 地址格式 (192.168.1.50) 或者 CIDR 格式 (192.168.1.0/24)，并可用 , 分隔多个地址，或用 * 代表所有 IP 地址。

> **警告：**
>
> 需谨慎使用 * 符号，因为 * 允许来自任何 IP 的客户端自行汇报其 IP 地址，从而可能引入安全风险。另外，* 可能导致部分直接连接 TiDB 的内部组件无法使用，例如 TiDB Dashboard。

14.5.1.15.2 fallbackable 从 v6.5.1 版本开始引入

- 用于控制是否启用 PROXY 协议回退模式。如果设置为 `true`，TiDB 可以接受属于 `proxy-protocol.networks` 的客户端使用非 PROXY 协议规范或者没有发送 PROXY 协议头的客户端连接。默认情况下，TiDB 仅接受属于 `proxy-protocol.networks` 的客户端发送 PROXY 协议头的客户端连接。
- 默认：`false`

14.5.1.16 experimental

experimental 部分为 TiDB 实验功能相关的配置。该部分从 v3.1.0 开始引入。

14.5.1.16.1 allow-expression-index 从 v4.0.0 版本开始引入

- 用于控制是否能创建表达式索引。自 v5.2.0 版本起，如果表达式中的函数是安全的，你可以直接基于该函数创建表达式索引，不需要打开该配置项。如果要创建基于其他函数的表达式索引，可以打开该配置项，但可能存在正确性问题。通过查询 `tidb_allow_function_for_expression_index` 变量可得到能直接用于创建表达式的安全函数。
- 默认值：false

14.5.2 TiKV 配置文件描述

TiKV 配置文件比命令行参数支持更多的选项。你可以在 etc/config-template.toml 找到默认值的配置文件，重命名为 config.toml 即可。

本文档只阐述未包含在命令行参数中的参数，命令行参数参见TiKV 配置参数。

> Tip:
>
> 如果你需要调整配置项的值，请参考修改配置参数进行操作。

14.5.2.1 全局配置

14.5.2.1.1 `abort-on-panic`

- 设置 TiKV panic 时是否调用 `abort()` 退出进程。此选项影响 TiKV 是否允许系统生成 core dump 文件。
  - 如果此配置项值为 false，当 TiKV panic 时，TiKV 调用 `exit()` 退出进程。
  - 如果此配置项值为 true，当 TiKV panic 时，TiKV 调用 `abort()` 退出进程。此时 TiKV 允许系统在退出时生成 core dump 文件。要生成 core dump 文件，你还需要进行 core dump 相关的系统配置（比如打开 `ulimit -c` 和配置 core dump 路径，不同操作系统配置方式不同）。建议将 core dump 生成路径设置在 TiKV 数据的不同磁盘分区，避免 core dump 文件占用磁盘空间过大，造成 TiKV 磁盘空间不足。
- 默认值：false

14.5.2.1.2 `slow-log-file`

- 存储慢日志的文件。
- 如果未设置本项但设置了 `log.file.filename`，慢日志将输出至 `log.file.filename` 指定的日志文件中。
- 如果本项和 `log.file.filename` 均未设置，所有日志默认输出到 `"stderr"`。
- 如果同时设置了两项，普通日志会输出至 `log.file.filename` 指定的日志文件中，而慢日志则会输出至本配置项指定的日志文件中。
- 默认值：“”

14.5.2.1.3 `slow-log-threshold`

- 输出慢日志的阈值。处理时间超过该阈值后会输出慢日志。
- 默认值：“1s”

14.5.2.1.4 memory-usage-limit

- TiKV 实例的内存使用限制。当 TiKV 的内存使用量接近此阈值时，内部缓存会被清除以释放内存。
- 在大多数情况下，TiKV 实例被设置为占系统可用总内存的 75%，因此你不需要显式指定此配置项。剩余 25% 的内存用于操作系统的页缓存，详情参见storage.block-cache.capacity。
- 在单个物理机上部署多个 TiKV 节点时，你也不需要设置此配置项。在这种情况下，TiKV 实例使用 `5/3 * block-cache.capacity` 的内存。
- 不同系统内存容量的默认值如下：
  - system=8G block-cache=3.6G memory-usage-limit=6G page-cache=2G
  - system=16G block-cache=7.2G memory-usage-limit=12G page-cache=4G
  - system=32G block-cache=14.4G memory-usage-limit=24G page-cache=8G

14.5.2.2 log 从 v5.4.0 版本开始引入

日志相关的配置项。

自 v5.4.0 版本起，废弃原 log 参数 `log-rotation-timespan`，并将 `log-level`、`log-format`、`log-file`、`log-rotation-size` 变更为下列参数，与 TiDB 的 log 参数保持一致。如果只设置了原参数、且把其值设为非默认值，原参数与新参数会保持兼容；如果同时设置了原参数和新参数，则会使用新参数。

14.5.2.2.1 level 从 v5.4.0 版本开始引入

- 日志等级。
- 可选值：“debug”，“info”，“warn”，“error”，“fatal”
- 默认值：“info”

14.5.2.2.2 format 从 v5.4.0 版本开始引入

- 日志的格式。
- 可选值：“json”，“text”
- 默认值：“text”

14.5.2.2.3 enable-timestamp 从 v5.4.0 版本开始引入

- 是否开启日志中的时间戳。
- 可选值：“true”，“false”
- 默认值：“true”

14.5.2.3 log.file 从 v5.4.0 版本开始引入

日志文件相关的配置项。

14.5.2.3.1 filename 从 v5.4.0 版本开始引入

- log 文件。如果未设置该参数，日志会默认输出到 "stderr"；如果设置了该参数，log 会输出到对应的文件中。
- 默认值： “”

14.5.2.3.2 max-size 从 v5.4.0 版本开始引入

- 单个 log 文件最大大小，超过设定的参数值后，系统自动切分成多个文件。
- 默认值：300
- 最大值：4096
- 单位：MiB

14.5.2.3.3 max-days 从 v5.4.0 版本开始引入

- 保留 log 文件的最长天数。
  - 如果未设置本参数或把此参数设置为默认值 0，TiKV 不清理 log 文件。
  - 如果把此参数设置为非 0 的值，在 max-days 之后，TiKV 会清理过期的日志文件。
- 默认值：0

14.5.2.3.4 max-backups 从 v5.4.0 版本开始引入

- 可保留的 log 文件的最大数量。
  - 如果未设置本参数或把此参数设置为默认值 0，TiKV 会保存所有的 log 文件；
  - 如果把此参数设置为非 0 的值，TiKV 最多会保留 max-backups 中指定的数量的旧日志文件。比如，如果该值设置为 7，TiKV 最多会保留 7 个旧的日志文件。
- 默认值：0

14.5.2.3.5 pd.enable-forwarding 从 v5.0.0 版本开始引入

- 控制 TiKV 中的 PD client 在疑似网络隔离的情况下是否通过 follower 将请求转发给 leader。
- 默认值：false
- 如果确认环境存在网络隔离的可能，开启这个参数可以减少服务不可用的窗口期。
- 如果无法准确判断隔离、网络中断、宕机等情况，这个机制存在误判情况从而导致可用性、性能降低。如果网络中从未发生过网络故障，不推荐开启此选项。

14.5.2.4 server

服务器相关的配置项。

14.5.2.4.1 addr

- 服务器监听的 IP 地址和端口号。
- 默认值：`"127.0.0.1:20160"`

14.5.2.4.2 advertise-addr

- 用于客户端通信的对外访问地址。
- 如果没有设置该配置项，则使用 `addr` 的值。
- 默认值：`""`

14.5.2.4.3 status-addr

- 通过 HTTP 直接报告 TiKV 状态的地址。

> **警告：**
> 如果该值暴露在公网，TiKV 服务器的状态可能会泄露。

- 要禁用 `status-addr`，请将该值设置为 `""`。
- 默认值：`"127.0.0.1:20180"`

14.5.2.4.4 status-thread-pool-size

- HTTP API 服务的工作线程数量。
- 默认值：1
- 最小值：1

14.5.2.4.5 grpc-compression-type

- gRPC 消息的压缩算法。
- 可选值：`"none"`、`"deflate"`、`"gzip"`
- 默认值：`"none"`

14.5.2.4.6 grpc-concurrency

- gRPC 工作线程的数量。调整 gRPC 线程池的大小时，请参考TiKV 线程池调优。
- 默认值：5
- 最小值：1

14.5.2.4.7 grpc-concurrent-stream

- 一个 gRPC 链接中最多允许的并发请求数量。
- 默认值：1024
- 最小值：1

14.5.2.4.8 grpc-memory-pool-quota

- gRPC 可使用的内存大小限制。
- 默认值：无限制
- 建议仅在出现内存不足 (OOM) 的情况下限制内存使用。需要注意，限制内存使用可能会导致卡顿。

14.5.2.4.9 grpc-raft-conn-num

- TiKV 节点之间用于 Raft 通讯的链接最大数量。
- 默认值：1
- 最小值：1

14.5.2.4.10 max-grpc-send-msg-len

- 设置可发送的最大 gRPC 消息长度。
- 默认值：10485760
- 单位：Bytes
- 最大值：2147483647

14.5.2.4.11 grpc-stream-initial-window-size

- gRPC stream 的 window 大小。
- 默认值：2MB
- 单位：KB|MB|GB
- 最小值：1KB

14.5.2.4.12 grpc-keepalive-time

- gRPC 发送 keep alive ping 消息的间隔时长。
- 默认值：10s
- 最小值：1s

14.5.2.4.13 grpc-keepalive-timeout

- 关闭 gRPC 链接的超时时长。
- 默认值：3s
- 最小值：1s

14.5.2.4.14 concurrent-send-snap-limit

- 同时发送 snapshot 的最大个数。
- 默认值：32
- 最小值：1

14.5.2.4.15 concurrent-recv-snap-limit

- 同时接受 snapshot 的最大个数。
- 默认值：32
- 最小值：1

14.5.2.4.16 end-point-recursion-limit

- endpoint 下推查询请求解码消息时，最多允许的递归层数。
- 默认值：1000
- 最小值：1

14.5.2.4.17 end-point-request-max-handle-duration

- endpoint 下推查询请求处理任务最长允许的时长。
- 默认值：60s
- 最小值：1s

14.5.2.4.18 snap-io-max-bytes-per-sec

- 处理 snapshot 时最大允许使用的磁盘带宽。
- 默认值：100MB
- 单位：KB|MB|GB
- 最小值：1KB

14.5.2.4.19 enable-request-batch

- 控制是否开启批处理请求。
- 默认值：true

14.5.2.4.20 labels

- 指定服务器属性，例如 `{ zone = "us-west-1", disk = "ssd" }`。
- 默认值：{}

14.5.2.4.21 background-thread-count

- 后台线程池的工作线程数量，包括 endpoint 线程、 BR 线程、 split check 线程、 Region 线程以及其他延迟不敏感的任务线程。
- 默认值：当 CPU 核数小于 16 时，默认值为 2。否则，默认值为 3。

14.5.2.4.22 end-point-slow-log-threshold

- endpoint 下推查询请求输出慢日志的阈值，处理时间超过阈值后会输出慢日志。
- 默认值：1s
- 最小值：0

14.5.2.4.23 raft-client-queue-size

- 该配置项指定 TiKV 中发送 Raft 消息的缓冲区大小。如果存在消息发送不及时导致缓冲区满、消息被丢弃的情况，可以适当调大该配置项值以提升系统运行的稳定性。
- 默认值：8192

14.5.2.4.24 simplify-metrics 从 v6.2.0 版本开始引入

- 是否精简返回的监控指标 Metrics 数据。设置为 `true` 后，TiKV 可以通过过滤部分 Metrics 采样数据以减少每次请求返回的 Metrics 数据量。
- 默认值：false

14.5.2.4.25 forward-max-connections-per-address 从 v5.0.0 版本开始引入

- 设置服务与转发请求的连接池大小。设置过小会影响请求的延迟和负载均衡。
- 默认值：4

14.5.2.5 readpool.unified

统一处理读请求的线程池相关的配置项。该线程池自 4.0 版本起取代原有的 storage 和 coprocessor 线程池。

14.5.2.5.1 min-thread-count

- 统一处理读请求的线程池最少的线程数量。
- 默认值：1

14.5.2.5.2 max-thread-count

- 统一处理读请求的线程池最多的线程数量，即 UnifyReadPool 线程池的大小。调整该线程池的大小时，请参考TiKV 线程池调优。
- 可调整范围：[min-thread-count, MAX(4, CPU quota * 10)]。其中，MAX(4, CPU quota * 10) 表示：如果 CPU 配额乘 10 小于 4，取 4；如果 CPU 配额乘 10 大于 4，即 CPU 配额大于 0.4，则取 CPU 配额乘 10。
- 默认值：MAX(4, CPU quota * 0.8)

> 注意：
>
> 增加线程数量会导致上下文切换增多，可能会导致性能下降，因此不推荐修改此配置。

14.5.2.5.3 stack-size

- 统一处理读请求的线程池中线程的栈大小。
- 类型：整数 + 单位
- 默认值：10MB
- 单位：KB|MB|GB
- 最小值：2MB
- 最大值：在系统中执行 ulimit -sH 命令后，输出的千字节数。

14.5.2.5.4 max-tasks-per-worker

- 统一处理读请求的线程池中单个线程允许积压的最大任务数量，超出后会返回 Server Is Busy。
- 默认值：2000
- 最小值：2

14.5.2.5.5 auto-adjust-pool-size 从 v6.3.0 版本开始引入

- 是否开启自动调整线程池的大小。开启此配置可以基于当前的 CPU 使用情况，自动调整统一处理读请求的线程池 (UnifyReadPool) 的大小，优化 TiKV 的读性能。目前线程池自动调整的范围为：[max-thread-count, MAX(4, CPU)](上限与max-thread-count 可设置的最大值相同)。
- 默认值：false

14.5.2.6 readpool.storage

存储线程池相关的配置项。

14.5.2.6.1 use-unified-pool

- 是否使用统一的读取线程池（在readpool.unified 中配置）处理存储请求。该选项值为 false 时，使用单独的存储线程池。通过本节 (readpool.storage) 中的其余配置项配置单独的线程池。
- 默认值：如果本节 (readpool.storage) 中没有其他配置，默认为 true。否则，为了升级兼容性，默认为 false，请根据需要更改readpool.unified 中的配置后再启用该选项。

14.5.2.6.2 high-concurrency

- 处理高优先级读请求的线程池线程数量。
- 当 8 ≤ cpu num ≤ 16 时，默认值为 cpu_num * 0.5；当 cpu num 小于 8 时，默认值为 4；当 cpu num 大于 16 时，默认值为 8。
- 最小值：1

14.5.2.6.3 normal-concurrency

- 处理普通优先级读请求的线程池线程数量。
- 当 8 ≤ cpu num ≤ 16 时，默认值为 cpu_num * 0.5；当 cpu num 小于 8 时，默认值为 4；当 cpu num 大于 16 时，默认值为 8。
- 最小值：1

14.5.2.6.4 low-concurrency

- 处理低优先级读请求的线程池线程数量。
- 当 8 ≤ cpu num ≤ 16 时，默认值为 cpu_num * 0.5；当 cpu num 小于 8 时，默认值为 4；当 cpu num 大于 16 时，默认值为 8。
- 最小值：1

14.5.2.6.5 max-tasks-per-worker-high

- 高优先级线程池中单个线程允许积压的最大任务数量，超出后会返回 Server Is Busy。
- 默认值：2000
- 最小值：2

14.5.2.6.6 max-tasks-per-worker-normal

- 普通优先级线程池中单个线程允许积压的最大任务数量，超出后会返回 Server Is Busy。
- 默认值：2000
- 最小值：2

14.5.2.6.7 max-tasks-per-worker-low

- 低优先级线程池中单个线程允许积压的最大任务数量，超出后会返回 Server Is Busy。
- 默认值：2000
- 最小值：2

14.5.2.6.8 stack-size

- Storage 读线程池中线程的栈大小。
- 类型：整数 + 单位
- 默认值：10MB
- 单位：KB|MB|GB
- 最小值：2MB
- 最大值：在系统中执行 `ulimit -sH` 命令后，输出的千字节数。

14.5.2.7 readpool.coprocessor

协处理器线程池相关的配置项。

14.5.2.7.1 use-unified-pool

- 是否使用统一的读取线程池（在`readpool.unified` 中配置）处理协处理器请求。该选项值为 false 时，使用单独的协处理器线程池。通过本节 (`readpool.coprocessor`) 中的其余配置项配置单独的线程池。
- 默认值：如果本节 (`readpool.coprocessor`) 中没有其他配置，默认为 true。否则，为了升级兼容性，默认为 false，请根据需要更改`readpool.unified` 中的配置后再启用该选项。

14.5.2.7.2 high-concurrency

- 处理高优先级 Coprocessor 请求（如点查）的线程池线程数量。
- 默认值：CPU * 0.8
- 最小值：1

14.5.2.7.3 normal-concurrency

- 处理普通优先级 Coprocessor 请求的线程池线程数量。
- 默认值：CPU * 0.8
- 最小值：1

14.5.2.7.4 low-concurrency

- 处理低优先级 Coprocessor 请求（如扫表）的线程池线程数量。
- 默认值：CPU * 0.8
- 最小值：1

14.5.2.7.5 max-tasks-per-worker-high

- 高优先级线程池中单个线程允许积压的任务数量，超出后会返回 Server Is Busy。
- 默认值：2000
- 最小值：2

14.5.2.7.6 max-tasks-per-worker-normal

- 普通优先级线程池中单个线程允许积压的任务数量，超出后会返回 Server Is Busy。
- 默认值：2000
- 最小值：2

14.5.2.7.7 max-tasks-per-worker-low

- 低优先级线程池中单个线程允许积压的任务数量，超出后会返回 Server Is Busy。
- 默认值：2000
- 最小值：2

14.5.2.7.8 stack-size

- Coprocessor 线程池中线程的栈大小。
- 默认值：10MB
- 单位：KB|MB|GB
- 最小值：2MB
- 最大值：在系统中执行 `ulimit -sH` 命令后，输出的千字节数。

14.5.2.8 storage

存储相关的配置项。

14.5.2.8.1 data-dir

- RocksDB 存储路径。
- 默认值：`"./"`

14.5.2.8.2 engine 从 v6.6.0 版本开始引入

> 警告：
>
> 该功能目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现 bug，请在 GitHub 上提 issue 反馈。

- 设置存储引擎类型。该配置只能在创建新集群时指定，且后续无法更改。
- 默认值：`"raft-kv"`
- 可选值：
  - `"raft-kv"`：TiDB v6.6.0 之前版本的默认存储引擎。
  - `"partitioned-raft-kv"`：TiDB v6.6.0 新引入的存储引擎。

14.5.2.8.3 scheduler-concurrency

- scheduler 内置一个内存锁机制，防止同时对一个 key 进行操作。每个 key hash 到不同的槽。
- 默认值：524288
- 最小值：1

14.5.2.8.4 scheduler-worker-pool-size

- Scheduler 线程池中线程的数量。Scheduler 线程主要负责写入之前的事务一致性检查工作。如果 CPU 核心数量大于等于 16，默认为 8；否则默认为 4。调整 scheduler 线程池的大小时，请参考TiKV 线程池调优。
- 默认值：4
- 可调整范围：`[1, MAX(4, CPU)]`。其中，`MAX(4, CPU)` 表示：如果 CPU 核心数量小于 4，取 4；如果 CPU 核心数量大于 4，则取 CPU 核心数量。

14.5.2.8.5 scheduler-pending-write-threshold

- 写入数据队列的最大值，超过该值之后对于新的写入 TiKV 会返回 Server Is Busy 错误。
- 默认值：100MB
- 单位：MB|GB

14.5.2.8.6 enable-async-apply-prewrite

- 控制异步提交 (Async Commit) 事务在应用 prewrite 请求之前是否响应 TiKV 客户端。开启该配置项可以降低 apply 耗时较高时的延迟，或者减少 apply 耗时不稳定时的延迟抖动。
- 默认值：`false`

14.5.2.8.7 reserve-space

- TiKV 启动时会预留一块空间用于保护磁盘空间。当磁盘剩余空间小于该预留空间时，TiKV 会限制部分写操作。预留空间形式上分为两个部分：预留空间的 80% 用作磁盘空间不足时的运维操作所需要的额外磁盘空间，剩余的 20% 为磁盘临时文件。在回收空间的过程中，如果额外使用的磁盘空间过多，导致存储耗尽时，该临时文件会成为恢复服务的最后一道防御。
- 临时文件名为 `space_placeholder_file`，位于 `storage.data-dir` 目录下。当 TiKV 因磁盘空间耗尽而下线时，重启 TiKV 会自动删除该临时文件，并自动尝试回收空间。
- 当剩余空间不足时，TiKV 不会创建该临时文件。防御的有效性与预留空间的大小有关。预留空间大小的计算方式为磁盘容量的 5% 与该配置项之间的最大值。当该配置项的值为 0MB 时，TiKV 会关闭磁盘防护功能。
- 默认值：5GB
- 单位：MB|GB

14.5.2.8.8 enable-ttl

警告：

- 你只能在部署新的 TiKV 集群时将 `enable-ttl` 的值设置为 `true` 或 `false`，不能在已有的 TiKV 集群中修改该配置项的值。由于该配置项为 `true` 和 `false` 的 TiKV 集群所存储的数据格式不相同，如果你在已有的 TiKV 集群中修改该配置项的值，会造成不同格式的数据存储在同一个集群，导致重启对应的 TiKV 集群时 TiKV 报“can’t enable ttl on a non-ttl instance”错误。
- 你只能在 TiKV 集群中使用 `enable-ttl`，不能在有 TiDB 节点的集群中使用该配置项（即在此类集群中把 `enable-ttl` 设置为 `true`），否则会导致数据损坏、TiDB 集群升级失败等严重后果。

- TTL 即 Time to live。数据超过 TTL 时间后会被自动删除。用户需在客户端写入请求中指定 TTL。不指定 TTL 即表明相应数据不会被自动删除。
- 默认值：false

14.5.2.8.9 `ttl-check-poll-interval`

- 回收数据物理空间的检查周期。如果数据超过了 TTL 时间，数据的物理空间会在检查时被强制回收。
- 默认值：12h
- 最小值：0s

14.5.2.8.10 `background-error-recovery-window` 从 v6.1.0 版本开始引入

- RocksDB 检测到可恢复的后台错误后，所允许的最长恢复时间。如果后台 SST 文件出现损坏，RocksDB 在检测到故障 SST 文件所属的 Peer 后，会通过心跳上报到 PD。PD 随后会进行调度操作移除该 Peer。最后故障 SST 文件将会被直接删除，随后 TiKV 后台恢复正常。
- 在恢复操作完成之前，损坏的 SST 文件将一直存在。此时 RocksDB 可以继续写入新的内容，但读到损坏的数据范围时会返回错误。
- 如果恢复操作未能在该时间窗口内完成，TiKV 会崩溃。
- 默认值：1h

14.5.2.8.11 `api-version` 从 v6.1.0 版本开始引入

- TiKV 作为 RawKV 存储数据时使用的存储格式与接口版本。
- 可选值：
  - 1：使用 API V1。不对客户端传入的数据进行编码，而是原样存储。在 v6.1.0 之前的版本，TiKV 都使用 API V1。
  - 2：使用 API V2：
    * 数据采用多版本并发控制 (MVCC) 方式存储，其中时间戳由 tikv-server 从 PD 获取（即 TSO）。
    * 数据根据使用方式划分范围，支持单一集群 TiDB、事务 KV、RawKV 应用共存。

    * 需要同时设置 `storage.enable-ttl = true`。由于 API V2 支持 TTL 特性，因此强制要求打开 `enable-ttl` 以避免这个参数出现歧义。
    * 启用 API V2 后需要在集群中额外部署至少一个 tidb-server 以回收过期数据。该 tidb-server 可同时提供数据库读写服务。可以部署多个 tidb-server 以保证高可用。
    * 需要客户端的支持。请参考对应客户端的 API V2 使用说明。
    * 从 v6.2.0 版本开始，你可以通过 RawKV CDC 组件实现 RawKV 的 Change Data Capture (CDC)。

- 默认值：1

> 警告：
>
> - 由于 API V1 和 API V2 底层存储格式不同，因此仅当 TiKV 中只有 TiDB 数据时，可以平滑启用或关闭 API V2。其他情况下，需要新建集群，并使用 TiKV Backup & Restore 工具进行数据迁移。
> - 启用 API V2 后，不能将 TiKV 集群回退到 v6.1.0 之前的版本，否则可能导致数据损坏。

14.5.2.9 storage.block-cache

RocksDB 多个 CF 之间共享 block cache 的配置选项。

14.5.2.9.1 capacity

- 共享 block cache 的大小。
- 默认值：系统总内存大小的 45%
- 单位：KB|MB|GB

14.5.2.10 storage.flow-control

在 scheduler 层进行流量控制代替 RocksDB 的 write stall 机制，可以避免 write stall 机制卡住 Raftstore 或 Apply 线程导致的次生问题。本节介绍 TiKV 流量控制机制相关的配置项。

14.5.2.10.1 enable

- 是否开启流量控制机制。开启后，TiKV 会自动关闭 KvDB 的 write stall 机制，还会关闭 RaftDB 中除 memtable 以外的 write stall 机制。
- 默认值：true

14.5.2.10.2 memtables-threshold

- 当 KvDB 的 memtable 的个数达到该阈值时，流控机制开始工作。当 `enable` 的值为 `true` 时，会覆盖 `rocksdb.(defaultcf|writecf|lockcf).max-write-buffer-number` 的配置。
- 默认值：5

14.5.2.10.3 l0-files-threshold

- 当 KvDB 的 L0 文件个数达到该阈值时，流控机制开始工作。当 enable 的值为 true 时，会覆盖 rocksdb.(defaultcf|writecf|lockcf).level0-slowdown-writes-trigger的配置。
- 默认值：20

14.5.2.10.4 soft-pending-compaction-bytes-limit

- 当 KvDB 的 pending compaction bytes 达到该阈值时，流控机制开始拒绝部分写入请求，报错 ServerIsBusy。当 enable 的值为 true 时，会覆盖 rocksdb.(defaultcf|writecf|lockcf).soft-pending-compaction-bytes-limit 的配置。
- 默认值：“192GB”

14.5.2.10.5 hard-pending-compaction-bytes-limit

- 当 KvDB 的 pending compaction bytes 达到该阈值时，流控机制拒绝所有写入请求，报错 ServerIsBusy。当 enable 的值为 true 时，会覆盖 rocksdb.(defaultcf|writecf|lockcf).hard-pending-compaction-bytes-limit 的配置。
- 默认值：“1024GB”

14.5.2.11 storage.io-rate-limit

I/O rate limiter 相关的配置项。

14.5.2.11.1 max-bytes-per-sec

- 限制服务器每秒从磁盘读取数据或写入数据的最大 I/O 字节数，I/O 类型由下面的 mode 配置项决定。达到该限制后，TiKV 倾向于放缓后台操作为前台操作节流。该配置项值应设为磁盘的最佳 I/O 带宽，例如云盘厂商指定的最大 I/O 带宽。
- 默认值：“0MB”

14.5.2.11.2 mode

- 确定哪些类型的 I/O 操作被计数并受 max-bytes-per-sec 阈值的限流。当前 TiKV 只支持 write-only 只写模式。
- 可选值："read-only"，"write-only"，"all-io"
- 默认值："write-only"

14.5.2.12 pd

14.5.2.12.1 endpoints

- PD 的地址。当指定多个地址时，需要用逗号，分隔。
- 默认值：["127.0.0.1:2379"]

14.5.2.12.2 retry-interval

- 初始化 PD 连接时的重试间隔。
- 默认值：`"300ms"`

14.5.2.12.3 retry-log-every

- 指定 PD 客户端在观察到错误时跳过报错的频率。例如，当配置项值为 5 时，每次 PD 观察到错误时，将跳过 4 次报错，直到第 5 次错误时才报告。
- 要禁用此功能，请将值设置为 1。
- 默认值：10

14.5.2.12.4 retry-max-count

- 初始化 PD 连接的最大重试次数。
- 要禁用重试，请将该值设置为 0。要解除重试次数的限制，请将该值设置为 -1。
- 默认值：-1

14.5.2.13 raftstore

raftstore 相关的配置项。

14.5.2.13.1 prevote

- 开启 Prevote 的开关，开启有助于减少隔离恢复后对系统造成的抖动。
- 默认值：true

14.5.2.13.2 capacity

- 存储容量，即允许的最大数据存储大小。如果没有设置，则使用当前磁盘容量。如果要将多个 TiKV 实例部署在同一块物理磁盘上，需要在 TiKV 配置中添加该参数，参见混合部署的关键参数介绍。
- 默认值：0
- 单位：KB|MB|GB

14.5.2.13.3 raftdb-path

- raft 库的路径，默认存储在 storage.data-dir/raft 下。
- 默认值：“”

14.5.2.13.4 raft-base-tick-interval

> 注意：
> 该配置项不支持通过 SQL 语句查询，但支持在配置文件中进行配置。

- 状态机 tick 一次的间隔时间。
- 默认值：1s
- 最小值：大于 0

14.5.2.13.5 raft-heartbeat-ticks

> 注意：
> 该配置项不支持通过 SQL 语句查询，但支持在配置文件中进行配置。

- 发送心跳时经过的 tick 个数，即每隔 raft-base-tick-interval * raft-heartbeat-ticks 时间发送一次心跳。
- 默认值：2
- 最小值：大于 0

14.5.2.13.6 raft-election-timeout-ticks

> 注意：
> 该配置项不支持通过 SQL 语句查询，但支持在配置文件中进行配置。

- 发起选举时经过的 tick 个数，即如果处于无主状态，大约经过 raft-base-tick-interval * raft-election-timeout-ticks 时间以后发起选举。
- 默认值：10
- 最小值：raft-heartbeat-ticks

14.5.2.13.7 raft-min-election-timeout-ticks

> 注意：
> 该配置项不支持通过 SQL 语句查询，但支持在配置文件中进行配置。

- 发起选举时至少经过的 tick 个数，如果为 0，则表示使用 raft-election-timeout-ticks，不能比 raft-election-timeout-ticks 小。
- 默认值：0
- 最小值：0

14.5.2.13.8 raft-max-election-timeout-ticks

> 注意：
> 该配置项不支持通过 SQL 语句查询，但支持在配置文件中进行配置。

- 发起选举时最多经过的 tick 个数，如果为 0，则表示使用 raft-election-timeout-ticks * 2。
- 默认值：0
- 最小值：0

14.5.2.13.9 raft-max-size-per-msg

> 注意：
> 该配置项不支持通过 SQL 语句查询，但支持在配置文件中进行配置。

- 产生的单个消息包的大小限制，软限制。
- 默认值：1MB
- 最小值：大于 0
- 最大值: 3GB
- 单位：KB|MB|GB

14.5.2.13.10 raft-max-inflight-msgs

> 注意：
> 该配置项不支持通过 SQL 语句查询，但支持在配置文件中进行配置。

- 待确认的日志个数，如果超过这个数量，Raft 状态机会减缓发送日志的速度。
- 默认值：256
- 最小值：大于 0
- 最大值: 16384

14.5.2.13.11 raft-entry-max-size

- 单个日志最大大小，硬限制。
- 默认值：8MB
- 最小值：0
- 单位：MB|GB

14.5.2.13.12 raft-log-compact-sync-interval 从 v5.3 版本开始引入

- 压缩非必要 Raft 日志的时间间隔。
- 默认值：“2s”
- 最小值：“0s”

14.5.2.13.13 raft-log-gc-tick-interval

- 删除 Raft 日志的轮询任务调度间隔时间，0 表示不启用。
- 默认值：“3s”
- 最小值：“0s”

14.5.2.13.14 raft-log-gc-threshold

- 允许残余的 Raft 日志个数，这是一个软限制。
- 默认值：50
- 最小值：1

14.5.2.13.15 raft-log-gc-count-limit

- 允许残余的 Raft 日志个数，这是一个硬限制。
- 默认值：3/4 Region 大小所能容纳的日志个数，按照每个日志 1 MiB 计算
- 最小值：0

14.5.2.13.16 raft-log-gc-size-limit

- 允许残余的 Raft 日志大小，这是一个硬限制。
- 默认值：Region 大小的 3/4
- 最小值：大于 0

14.5.2.13.17 raft-log-reserve-max-ticks 从 v5.3 版本开始引入

- 超过本配置项设置的的 tick 数后，即使剩余 Raft 日志的数量没有达到 raft-log-gc-threshold 设置的值，TiKV 也会进行 GC 操作。
- 默认值：6
- 最小值：大于 0

14.5.2.13.18 raft-engine-purge-interval

- 清除旧的 TiKV 日志文件的间隔时间，以尽快回收磁盘空间。Raft 引擎是可替换的组件，因此某些功能或优化的实现需要清除 TiKV 日志文件。
- 默认值：`"10s"`

14.5.2.13.19 raft-entry-cache-life-time

- 内存中日志 cache 允许的最长残留时间。
- 默认值：30s
- 最小值：0

14.5.2.13.20 hibernate-regions

- 打开或关闭静默 Region。打开后，如果 Region 长时间处于非活跃状态，即被自动设置为静默状态。静默状态的 Region 可以降低 Leader 和 Follower 之间心跳信息的系统开销。可以通过 `peer-stale-state-check-interval` 调整 Leader 和 Follower 之间的心跳间隔。
- 默认值：v5.0.2 及以后版本默认值为 true，v5.0.2 以前的版本默认值为 false

14.5.2.13.21 split-region-check-tick-interval

- 检查 Region 是否需要分裂的时间间隔，0 表示不启用。
- 默认值：10s
- 最小值：0

14.5.2.13.22 region-split-check-diff

- 允许 Region 数据超过指定大小的最大值。
- 默认值：Region 大小的 1/16
- 最小值：0

14.5.2.13.23 region-compact-check-interval

- 检查是否需要人工触发 RocksDB compaction 的时间间隔，0 表示不启用。
- 默认值：5m
- 最小值：0

14.5.2.13.24 region-compact-check-step

- 每轮校验人工 compaction 时，一次性检查的 Region 个数。
- 默认值：100
- 最小值：0

14.5.2.13.25 region-compact-min-tombstones

- 触发 RocksDB compaction 需要的 tombstone 个数。
- 默认值：10000
- 最小值：0

14.5.2.13.26 region-compact-tombstones-percent

- 触发 RocksDB compaction 需要的 tombstone 所占比例。
- 默认值：30
- 最小值：1
- 最大值：100

14.5.2.13.27 report-region-buckets-tick-interval 从 v6.1.0 版本开始引入

> **警告：**
>
> report-region-buckets-tick-interval 是 TiDB 在 v6.1.0 中引入的实验特性，不建议在生产环境中使用。

- 启用 enable-region-bucket 后，该配置项设置 TiKV 向 PD 上报 bucket 信息的间隔时间。
- 默认值：10s

14.5.2.13.28 pd-heartbeat-tick-interval

- 触发 Region 对 PD 心跳的时间间隔，0 表示不启用。
- 默认值：1m
- 最小值：0

14.5.2.13.29 pd-store-heartbeat-tick-interval

- 触发 store 对 PD 心跳的时间间隔，0 表示不启用。
- 默认值：10s
- 最小值：0

14.5.2.13.30 snap-mgr-gc-tick-interval

- 触发回收过期 snapshot 文件的时间间隔，0 表示不启用。
- 默认值：1m
- 最小值：0

14.5.2.13.31 snap-gc-timeout

- snapshot 文件的最长保存时间。
- 默认值：4h
- 最小值：0

14.5.2.13.32 snap-generator-pool-size 从 v5.4.0 版本开始引入

- 用于配置 snap-generator 线程池的大小。
- 为了让 TiKV 在恢复场景下加快 Region 生成 Snapshot 的速度，需要调大对应 Worker 的 snap-generator 线程数量。可通过本配置项调大对应线程的数量。
- 默认值：2
- 最小值：1

14.5.2.13.33 lock-cf-compact-interval

- 触发对 lock CF compact 检查的时间间隔。
- 默认值：10m
- 最小值：0

14.5.2.13.34 lock-cf-compact-bytes-threshold

- 触发对 lock CF 进行 compact 的大小。
- 默认值：256MB
- 最小值：0
- 单位：MB

14.5.2.13.35 notify-capacity

- Region 消息队列的最长长度。
- 默认值：40960
- 最小值：0

14.5.2.13.36 messages-per-tick

- 每轮处理的消息最大个数。
- 默认值：4096
- 最小值：0

14.5.2.13.37 max-peer-down-duration

- 副本允许的最长未响应时间，超过将被标记为 down，后续 PD 会尝试将其删掉。
- 默认值：10m
- 最小值：当 Hibernate Region 功能启用时，为 peer-stale-state-check-interval * 2；Hibernate Region 功能关闭时，为 0。

14.5.2.13.38 max-leader-missing-duration

- 允许副本处于无主状态的最长时间，超过将会向 PD 校验自己是否已经被删除。
- 默认值：2h
- 最小值：> abnormal-leader-missing-duration

14.5.2.13.39 abnormal-leader-missing-duration

- 允许副本处于无主状态的时间，超过将视为异常，标记在 metrics 和日志中。
- 默认值：10m
- 最小值：> peer-stale-state-check-interval

14.5.2.13.40 peer-stale-state-check-interval

- 触发检验副本是否处于无主状态的时间间隔。
- 默认值：5m
- 最小值：> 2 * election-timeout

14.5.2.13.41 leader-transfer-max-log-lag

- 尝试转移领导权时被转移者允许的最大日志缺失个数。
- 默认值：128
- 最小值：10

14.5.2.13.42 max-snapshot-file-raw-size 从 v6.1.0 版本开始引入

- 当 snapshot 文件大于该配置项指定的大小时，snapshot 文件会被切割为多个文件。
- 默认值：100MiB
- 最小值：100MiB

14.5.2.13.43 snap-apply-batch-size

- 当导入 snapshot 文件需要写数据时，内存写缓存的大小。
- 默认值：10MB
- 最小值：0
- 单位：MB

14.5.2.13.44 consistency-check-interval

> **警告：**
>
> 开启一致性检查对集群性能有影响，并且和 TiDB GC 操作不兼容，不建议在生产环境中使用。

- 触发一致性检查的时间间隔，0 表示不启用。
- 默认值：0s
- 最小值：0

14.5.2.13.45 raft-store-max-leader-lease

- Region 主可信任期的最长时间。
- 默认值：9s
- 最小值：0

14.5.2.13.46 right-derive-when-split

- 为 true 时，以最大分裂 key 为起点的 Region 复用原 Region 的 key；否则以原 Region 起点 key 作为起点的 Region 复用原 Region 的 key。
- 默认值：true

14.5.2.13.47 merge-max-log-gap

- 进行 merge 时，允许的最大日志缺失个数。
- 默认值：10
- 最小值：> raft-log-gc-count-limit

14.5.2.13.48 merge-check-tick-interval

- 触发 merge 完成检查的时间间隔。
- 默认值：2s
- 最小值：大于 0

14.5.2.13.49 use-delete-range

- 开启 rocksdb delete_range 接口删除数据的开关。
- 默认值：false

14.5.2.13.50 cleanup-import-sst-interval

- 触发检查过期 SST 文件的时间间隔，0 表示不启用。
- 默认值：10m
- 最小值：0

14.5.2.13.51 local-read-batch-size

- 一轮处理读请求的最大个数。
- 默认值：1024
- 最小值：大于 0

14.5.2.13.52 apply-yield-write-size 从 v6.4.0 版本开始引入

- Apply 线程每一轮处理单个状态机写入的最大数据量，这是个软限制。
- 默认值：32KiB
- 最小值：大于 0
- 单位：KiB|MiB|GiB

14.5.2.13.53 apply-max-batch-size

- Raft 状态机由 BatchSystem 批量执行数据写入请求，该配置项指定每批可执行请求的最多 Raft 状态机个数。
- 默认值：256
- 最小值：大于 0
- 最大值: 10240

14.5.2.13.54 apply-pool-size

- Apply 线程池负责把数据落盘至磁盘。该配置项为 Apply 线程池中线程的数量，即 Apply 线程池的大小。调整 Apply 线程池的大小时，请参考TiKV 线程池调优。
- 默认值：2
- 可调整范围：[1, CPU * 10]

14.5.2.13.55 store-max-batch-size

- Raft 状态机由 BatchSystem 批量执行把日志落盘至磁盘的请求，该配置项指定每批可执行请求的最多 Raft 状态机个数。
- 如果开启 hibernate-regions，默认值为 256；如果关闭 hibernate-regions，默认值为 1024
- 最小值：大于 0
- 最大值: 10240

14.5.2.13.56 store-pool-size

- 表示处理 Raft 的线程池中线程的数量，即 Raftstore 线程池的大小。调整该线程池的大小时，请参考TiKV 线程池调优。
- 默认值：2
- 可调整范围：[1, CPU * 10]

14.5.2.13.57 store-io-pool-size 从 v5.3.0 版本开始引入

- 表示处理 Raft I/O 任务的线程池中线程的数量，即 StoreWriter 线程池的大小。调整该线程池的大小时，请参考TiKV 线程池调优。
- 默认值：0
- 最小值：0

14.5.2.13.58 future-poll-size

- 驱动 future 的线程池中线程的数量。
- 默认值：1
- 最小值：大于 0

14.5.2.13.59 cmd-batch

- 对请求进行攒批的控制开关，开启后可显著提升写入性能。
- 默认值：true

14.5.2.13.60 inspect-interval

- TiKV 每隔一段时间会检测 Raftstore 组件的延迟情况，该配置项设置检测的时间间隔。当检测的延迟超过该时间，该检测会被记为超时。
- 根据超时的检测延迟的比例计算判断 TiKV 是否为慢节点。
- 默认值：500ms
- 最小值：1ms

14.5.2.13.61 raft-write-size-limit 从 v5.3.0 版本开始引入

- 触发 Raft 数据写入的阈值。当数据大小超过该配置项值，数据会被写入磁盘。当 store-io-pool-size 的值为 0 时，该配置项不生效。
- 默认值：1MB
- 最小值：0

14.5.2.13.62 report-min-resolved-ts-interval 从 v6.0.0 版本开始引入

- 设置 PD leader 收到 Resolved TS 的间隔时间。如果该值设置为 0，表示禁用该功能。
- 默认值：在 v6.3.0 之前版本中为 "0s"，在 v6.3.0 及之后的版本中为 "1s"，即最小正值。
- 最小值：0
- 单位：秒

14.5.2.14 coprocessor

Coprocessor 相关的配置项。

14.5.2.14.1 split-region-on-table

- 开启按 table 分裂 Region 的开关，建议仅在 TiDB 模式下使用。
- 默认值：false

14.5.2.14.2 batch-split-limit

- 批量分裂 Region 的阈值，调大该值可加速分裂 Region。
- 默认值：10
- 最小值：1

14.5.2.14.3 region-max-size

- Region 容量空间最大值，超过时系统分裂成多个 Region。
- 默认值：`region-split-size / 2 * 3`
- 单位：KiB|MiB|GiB

14.5.2.14.4 region-split-size

- 分裂后新 Region 的大小，此值属于估算值。
- 默认值：96MiB
- 单位：KiB|MiB|GiB

14.5.2.14.5 region-max-keys

- Region 最多允许的 key 的个数，超过时系统分裂成多个 Region。
- 默认值：`region-split-keys / 2 * 3`

14.5.2.14.6 region-split-keys

- 分裂后新 Region 的 key 的个数，此值属于估算值。
- 默认值：960000

14.5.2.14.7 consistency-check-method

- 指定数据一致性检查的方法。
- 要对 MVCC 数据进行一致性检查，设置该值为 `"mvcc"`。要对原始数据进行一致性检查，设置该值为 `"raw"`。
- 默认值：`"mvcc"`

14.5.2.15 coprocessor-v2

14.5.2.15.1 coprocessor-plugin-directory

- 已编译 coprocessor 插件所在目录的路径。TiKV 会自动加载该目录下的插件。
- 如果未设置该配置项，则 coprocessor 插件会被禁用。
- 默认值：`"./coprocessors"`

14.5.2.15.2 enable-region-bucket 从 v6.1.0 版本开始引入

- 是否将 Region 划分为更小的区间 bucket，并且以 bucket 作为并发查询单位，以提高扫描数据的并发度。bucket 的详细设计可见 Dynamic size Region。
- 默认值：false

> 警告：
>
> - enable-region-bucket 是 TiDB 在 v6.1.0 中引入的实验特性，不建议在生产环境中使用。
> - 这个参数仅在 region-split-size 调到两倍 region-bucket-size 及以上时才有意义，否则不会真正生成 bucket。
> - 将 region-split-size 调大可能会有潜在的性能回退、数据调度缓慢的风险。

14.5.2.15.3 region-bucket-size 从 v6.1.0 版本开始引入

- 设置 enable-region-bucket 启用时 bucket 的预期大小。
- 默认值：96MiB

> 警告：
>
> region-bucket-size 是 TiDB 在 v6.1.0 中引入的实验特性，不建议在生产环境中使用。

14.5.2.16 rocksdb

RocksDB 相关的配置项。

14.5.2.16.1 max-background-jobs

- RocksDB 后台线程个数。调整 RocksDB 线程池的大小时，请参考TiKV 线程池调优。
- 默认值：
  - CPU 核数为 10 时，默认值为 9
  - CPU 核数为 8 时，默认值为 7
  - CPU 核数为 N 时，默认值为 max(2, min(N - 1, 9))
- 最小值：2

14.5.2.16.2 max-background-flushes

- RocksDB 用于刷写 memtable 的最大后台线程数量。

- 默认值：
  - CPU 核数为 10 时，默认值为 3
  - CPU 核数为 8 时，默认值为 2
  - CPU 核数为 N 时，默认值为 `[(max-background-jobs + 3)/ 4]`
- 最小值：1

14.5.2.16.3 max-sub-compactions

- RocksDB 进行 subcompaction 的并发个数。
- 默认值：3
- 最小值：1

14.5.2.16.4 max-open-files

- RocksDB 可以打开的文件总数。
- 默认值：40960
- 最小值：-1

14.5.2.16.5 max-manifest-file-size

- RocksDB Manifest 文件最大大小。
- 默认值：128MB
- 最小值：0
- 单位：B|KB|MB|GB

14.5.2.16.6 create-if-missing

- 自动创建 DB 开关。
- 默认值：true

14.5.2.16.7 wal-recovery-mode

- 预写式日志 (WAL, Write Ahead Log) 的恢复模式。
- 可选值：
  - `"tolerate-corrupted-tail-records"`：容忍并丢弃位于日志尾部的不完整的数据 (trailing data)。
  - `"absolute-consistency"`：当发现待恢复的日志中有被损坏的日志时，放弃恢复所有日志。
  - `"point-in-time"`：按顺序恢复日志。遇到第一个损坏的日志时，停止恢复剩余的日志。
  - `"skip-any-corrupted-records"`：灾难后恢复。跳过日志中的损坏记录，尽可能多地恢复数据。
- 默认值：`"point-in-time"`

14.5.2.16.8 wal-dir

- WAL 存储目录，默认：“tmp/tikv/store”。
- 默认值：/tmp/tikv/store

14.5.2.16.9 wal-ttl-seconds

- 归档 WAL 生存周期，超过该值时，系统会删除相关 WAL。
- 默认值：0
- 最小值：0
- 单位：秒

14.5.2.16.10 wal-size-limit

- 归档 WAL 大小限制，超过该值时，系统会删除相关 WAL。
- 默认值：0
- 最小值：0
- 单位：B|KB|MB|GB

14.5.2.16.11 max-total-wal-size

- RocksDB WAL 总大小限制，即 data-dir 目录下 *.log 文件的大小总和。
- 默认值："4GB"

14.5.2.16.12 stats-dump-period

- 将统计信息输出到日志中的间隔时间。
- 默认值：10m

14.5.2.16.13 compaction-readahead-size

- 开启 RocksDB compaction 过程中的预读功能，该项指定预读数据的大小。如果使用的是机械磁盘，建议该值至少为 2MB。
- 默认值：0
- 最小值：0
- 单位：B|KB|MB|GB

14.5.2.16.14 writable-file-max-buffer-size

- WritableFileWrite 所使用的最大的 buffer 大小。
- 默认值：1MB
- 最小值：0
- 单位：B|KB|MB|GB

14.5.2.16.15 use-direct-io-for-flush-and-compaction

- 决定后台 flush 或者 compaction 的读写是否设置 O_DIRECT 的标志。该选项对性能的影响：开启 O_DIRECT 可以绕过并防止污染操作系统 buffer cache，但后续文件读取需要把内容重新读到 buffer cache。
- 默认值：false

14.5.2.16.16 rate-bytes-per-sec

- RocksDB compaction rate limiter 的限制速率。
- 默认值：10GB
- 最小值：0
- 单位：B|KB|MB|GB

14.5.2.16.17 rate-limiter-refill-period

- 控制 I/O 令牌的刷新频率。较小的值可以减少 I/O 尖刺，但会增加 CPU 开销。
- 默认值："100ms"

14.5.2.16.18 rate-limiter-mode

- RocksDB 的 compaction rate limiter 模式。
- 可选值：“read-only”，“write-only”，“all-io”
- 默认值：“write-only”

14.5.2.16.19 rate-limiter-auto-tuned 从 v5.0 版本开始引入

- 控制是否依据最近的负载量自动优化 RocksDB 的 compaction rate limiter 配置。此配置项开启后，compaction pending bytes 监控指标值会比一般情况下稍微高些。
- 默认值：true

14.5.2.16.20 enable-pipelined-write

- 控制是否开启 Pipelined Write。开启时会使用旧的 Pipelined Write，关闭时会使用新的 Pipelined Commit 机制。
- 默认值：false

14.5.2.16.21 bytes-per-sync

- 异步 Sync 限速速率。
- 默认值：1MB
- 最小值：0
- 单位：B|KB|MB|GB

14.5.2.16.22 wal-bytes-per-sync

- WAL Sync 限速速率，默认：512KB。
- 默认值：512KB
- 最小值：0
- 单位：B|KB|MB|GB

14.5.2.16.23 info-log-max-size

- Info 日志的最大大小。
- 默认值：1GB
- 最小值：0
- 单位：B|KB|MB|GB

14.5.2.16.24 info-log-roll-time

- 日志截断间隔时间，如果为 0s 则不截断。
- 默认值：0s

14.5.2.16.25 info-log-keep-log-file-num

- 保留日志文件最大个数。
- 默认值：10
- 最小值：0

14.5.2.16.26 info-log-dir

- 日志存储目录。
- 默认值：“”

14.5.2.16.27 info-log-level

- RocksDB 的日志级别。
- 默认值："info"

14.5.2.16.28 write-buffer-flush-oldest-first 从 v6.6.0 版本开始引入

> **警告：**
>
> 该功能目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现 bug，请在 GitHub 上提 issue 反馈。

- 设置当 RocksDB 当前 memtable 内存占用达到阈值之后的 Flush 策略。
- 默认值：`false`
- 可选值：
  - `false`：Flush 策略是优先选择数据量大的 memtable 落盘到 SST。
  - `true`：Flush 策略是优先选择最早的 memtable 落盘到 SST。该策略可以清除冷数据的 memtable，用于有明显冷热数据的场景。

14.5.2.16.29 `write-buffer-limit` 从 v6.6.0 版本开始引入

> **警告：**
>
> 该功能目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现 bug，请在 GitHub 上提 issue 反馈。

- 设置单个 TiKV 中所有 RocksDB 实例使用的 memtable 的总内存上限，默认值为本机内存的 25%，推荐配置不低于 5 GiB 的内存。该配置只对 Partitioned Raft KV (storage.engine="partitioned-raft-kv") 生效。
- 默认值：25%
- 单位：KiB|MiB|GiB

14.5.2.17 rocksdb.titan

Titan 相关的配置项。

14.5.2.17.1 `enabled`

- 开启 Titan 开关。
- 默认值：false

14.5.2.17.2 `dirname`

- Titan Blob 文件存储目录。
- 默认值：titandb

14.5.2.17.3 `disable-gc`

- 关闭 Titan 对 Blob 文件的 GC 的开关。
- 默认值：false

14.5.2.17.4 `max-background-gc`

- Titan 后台 GC 的线程个数。
- 默认值：4
- 最小值：1

14.5.2.18 rocksdb.defaultcf | rocksdb.writecf | rocksdb.lockcf

rocksdb defaultcf、 rocksdb writecf 和 rocksdb lockcf 相关的配置项。

14.5.2.18.1 block-size

- 一个 RocksDB block 的默认大小。
- defaultcf 默认值：32KB
- writecf 默认值：32KB
- lockcf 默认值：16KB
- 最小值：1KB
- 单位：KB|MB|GB

14.5.2.18.2 block-cache-size

> **警告：**
>
> 从 v6.6.0 起，该配置项被废弃。

- 一个 RocksDB block 的默认缓存大小。
- defaultcf 默认值：机器总内存 * 25%
- writecf 默认值：机器总内存 * 15%
- lockcf 默认值：机器总内存 * 2%
- 最小值：0
- 单位：KB|MB|GB

14.5.2.18.3 disable-block-cache

- 开启 block cache 开关。
- 默认值：false

14.5.2.18.4 cache-index-and-filter-blocks

- 开启缓存 index 和 filter 的开关。
- 默认值：true

14.5.2.18.5 pin-l0-filter-and-index-blocks

- 控制第 0 层 SST 文件的 index block 和 filter block 是否常驻在内存中的开关。
- 默认值：true

14.5.2.18.6 use-bloom-filter

- 开启 bloom filter 的开关。
- 默认值：true

14.5.2.18.7 optimize-filters-for-hits

- 开启优化 filter 的命中率的开关。
- defaultcf 默认值：true
- writecf 默认值：false
- lockcf 默认值：false

14.5.2.18.8 whole-key-filtering

- 开启将整个 key 放到 bloom filter 中的开关。
- defaultcf 默认值：true
- writecf 默认值：false
- lockcf 默认值：true

14.5.2.18.9 bloom-filter-bits-per-key

- bloom filter 为每个 key 预留的长度。
- 默认值：10
- 单位：字节

14.5.2.18.10 block-based-bloom-filter

- 开启每个 block 建立 bloom filter 的开关。
- 默认值：false

14.5.2.18.11 read-amp-bytes-per-bit

- 开启读放大统计的开关，0：不开启，> 0 开启。
- 默认值：0
- 最小值：0

14.5.2.18.12 compression-per-level

- 每一层默认压缩算法。
- defaultcf 的默认值：[“no”，“no”，“lz4”，“lz4”，“lz4”，“zstd”，“zstd”]
- writecf 的默认值：[“no”，“no”，“lz4”，“lz4”，“lz4”，“zstd”，“zstd”]
- lockcf 的默认值：[“no”，“no”，“no”，“no”，“no”，“no”，“no”]

14.5.2.18.13 bottommost-level-compression

- 设置最底层的压缩算法。该设置将覆盖 compression-per-level 的设置。
- 因为最底层并非从数据开始写入 LSM-tree 起就直接采用 compression-per-level 数组中的最后一个压缩算法，使用 bottommost-level-compression 可以让最底层从一开始就使用压缩效果最好的压缩算法。
- 如果不想设置最底层的压缩算法，可以将该配置项的值设为 disable。
- 默认值：“zstd”

14.5.2.18.14 write-buffer-size

- memtable 大小。
- defaultcf 默认值："128MB"
- writecf 默认值："128MB"
- lockcf 默认值："32MB"
- 最小值：0
- 单位：KB|MB|GB

14.5.2.18.15 max-write-buffer-number

- 最大 memtable 个数。当 storage.flow-control.enable 的值为 true 时，storage.flow-control.memtables
  ↪ -threshold 会覆盖此配置。
- 默认值：5
- 最小值：0

14.5.2.18.16 min-write-buffer-number-to-merge

- 触发 flush 的最小 memtable 个数。
- 默认值：1
- 最小值：0

14.5.2.18.17 max-bytes-for-level-base

- base level (L1) 最大字节数，一般设置为 memtable 大小 4 倍。当 L1 的数据量大小达到 max-bytes-for-level
  ↪ -base 限定的值的时候，会触发 L1 的 SST 文件和 L2 中有 overlap 的 SST 文件进行 compaction。
- defaultcf 默认值："512MB"
- writecf 默认值："512MB"
- lockcf 默认值："128MB"
- 最小值：0
- 单位：KB|MB|GB
- 建议 max-bytes-for-level-base 的取值和 L0 的数据量大致相等，以减少不必要的 compaction。假如压缩方式为“no:no:lz4:lz4:lz4:lz4:lz4”，那么 max-bytes-for-level-base 的值应该是 write-buffer-size * 4，因为 L0 和 L1 均没有压缩，且 L0 触发 compaction 的条件是 SST 文件的个数到达 4（默认值）。当 L0 和 L1 都发生了 compaction 时，需要分析 RocksDB 的日志了解由一个 memtable 压缩成的 SST 文件的大小。如果文件大小为 32MB，那么 max-bytes-for-level-base 的值建议设为 32MB * 4 = 128MB。

14.5.2.18.18 target-file-size-base

- base level 的目标文件大小。当 enable-compaction-guard 的值为 true 时，compaction-guard-max-output-file-size 会覆盖此配置。
- 默认值：8MB
- 最小值：0
- 单位：KB|MB|GB

14.5.2.18.19 level0-file-num-compaction-trigger

- 触发 compaction 的 L0 文件最大个数。
- defaultcf 默认值：4
- writecf 默认值：4
- lockcf 默认值：1
- 最小值：0

14.5.2.18.20 level0-slowdown-writes-trigger

- 触发 write stall 的 L0 文件最大个数。当 storage.flow-control.enable 的值为 true 时，storage.flow-control.l0-files-threshold 会覆盖此配置。
- 默认值：20
- 最小值：0

14.5.2.18.21 level0-stop-writes-trigger

- 完全阻停写入的 L0 文件最大个数。
- 默认值：36
- 最小值：0

14.5.2.18.22 max-compaction-bytes

- 一次 compaction 最大写入字节数，默认 2GB。
- 默认值：2GB
- 最小值：0
- 单位：KB|MB|GB

14.5.2.18.23 compaction-pri

- 优先处理 compaction 的类型。
- 可选值：
  - "by-compensated-size"：根据大小顺序，优先对大文件进行 compaction。
  - "oldest-largest-seq-first"：根据时间顺序，优先对数据更新时间晚的文件进行 compaction。当你只在小范围内更新部分热点键 (hot keys) 时，可以使用此配置。

  - `"oldest-smallest-seq-first"`：根据时间顺序，优先对长时间没有被 compact 到下一级的文件进行 compaction。如果你在大范围内随机更新了部分热点键，使用该配置可以轻微缓解写放大。
  - `"min-overlapping-ratio"`：根据重叠比例，优先对在不同层之间文件重叠比例高的文件进行 compaction，即一个文件在 下一层的大小/本层的大小 的值越小，compaction 的优先级越高。在诸多场景下，该配置可以有效缓解写放大。

- 默认值：
  - `defaultcf` 和 `writecf` 的默认值：`"min-overlapping-ratio"`
  - `lockcf` 的默认值：`"by-compensated-size"`

14.5.2.18.24 dynamic-level-bytes

- 开启 dynamic level bytes 优化的开关。
- 默认值：true

14.5.2.18.25 num-levels

- RocksDB 文件最大层数。
- 默认值：7

14.5.2.18.26 max-bytes-for-level-multiplier

- 每一层的默认放大倍数。
- 默认值：10

14.5.2.18.27 compaction-style

- compaction 方法。
- 可选值：“level”，“universal”，“fifo”
- 默认值：“level”

14.5.2.18.28 disable-auto-compactions

- 是否关闭自动 compaction。
- 默认值：false

14.5.2.18.29 soft-pending-compaction-bytes-limit

- pending compaction bytes 的软限制。当 `storage.flow-control.enable` 的值为 `true` 时，`storage.flow-control.soft-pending-compaction-bytes-limit` 会覆盖此配置。
- 默认值：192GB
- 单位：KB|MB|GB

14.5.2.18.30 hard-pending-compaction-bytes-limit

- pending compaction bytes 的硬限制。当 `storage.flow-control.enable` 的值为 `true` 时，`storage.flow-control.hard-pending-compaction-bytes-limit` 会覆盖此配置。
- 默认值：256GB
- 单位：KB|MB|GB

14.5.2.18.31 enable-compaction-guard

- 设置 compaction guard 的启用状态。compaction guard 优化通过使用 TiKV Region 边界分割 SST 文件，帮助降低 compaction I/O，让 TiKV 能够输出较大的 SST 文件，并且在迁移 Region 时及时清理过期数据。
- `defaultcf` 默认值：`true`
- `writecf` 默认值：`true`
- `lockcf` 默认值：`false`

14.5.2.18.32 compaction-guard-min-output-file-size

- 设置 compaction guard 启用时 SST 文件大小的最小值，防止 SST 文件过小。
- 默认值：8MB
- 单位：KB|MB|GB

14.5.2.18.33 compaction-guard-max-output-file-size

- 设置 compaction guard 启用时 SST 文件大小的最大值，防止 SST 文件过大。对于同一列族，此配置项的值会覆盖 `target-file-size-base`。
- 默认值：128MB
- 单位：KB|MB|GB

14.5.2.18.34 format-version 从 v6.2.0 版本开始引入

- 设置 SST 文件的格式版本。该配置项只影响新写入的表，对于已经存在的表，版本信息会从 footer 中读取。
- 可选值：
  - 0：适用于所有 TiKV 版本。默认 checksum 类型为 CRC32。该版本不支持修改 checksum 类型。
  - 1：适用于所有 TiKV 版本。支持使用非默认的 checksum 类型，例如 xxHash。只有在 checksum 类型不是 CRC32 时，RocksDB 才会写入数据。（0 版本会自动升级）
  - 2：适用于所有 TiKV 版本。更改了压缩块的编码方式，使用 LZ4、BZip2 和 Zlib 压缩。
  - 3：适用于 TiKV v2.1 及以上版本。更改了索引块中 key 的编码方式。
  - 4：适用于 TiKV v3.0 及以上版本。更改了索引块中 value 的编码方式。
  - 5：适用于 TiKV v6.1 及以上版本。全量和分区 filter 采用一种具有不同模式的、更快、更准确的 Bloom filter 实现。
- 默认值：2

14.5.2.19 rocksdb.defaultcf.titan

rocksdb defaultcf titan 相关的配置项。

14.5.2.19.1 min-blob-size

- 最小存储在 Blob 文件中 value 大小，低于该值的 value 还是存在 LSM-Tree 中。
- 默认值：1KB
- 最小值：0
- 单位：KB|MB|GB

14.5.2.19.2 blob-file-compression

- Blob 文件所使用的压缩算法，可选值：no、 snappy、 zlib、 bz2、 lz4、 lz4hc、 zstd。
- 默认值：lz4

14.5.2.19.3 blob-cache-size

- Blob 文件的 cache 大小。
- 默认值：0GB
- 最小值：0
- 单位：KB|MB|GB

14.5.2.19.4 min-gc-batch-size

- 做一次 GC 所要求的最低 Blob 文件大小总和。
- 默认值：16MB
- 最小值：0
- 单位：KB|MB|GB

14.5.2.19.5 max-gc-batch-size

- 做一次 GC 所要求的最高 Blob 文件大小总和。
- 默认值：64MB
- 最小值：0
- 单位：KB|MB|GB

14.5.2.19.6 discardable-ratio

- Blob 文件 GC 的触发比例，如果某 Blob 文件中的失效 value 的比例高于该值才可能被 GC 选中。
- 默认值：0.5
- 最小值：0
- 最大值：1

14.5.2.19.7 sample-ratio

- 进行 GC 时，对 Blob 文件进行采样时读取数据占整个文件的比例。
- 默认值：0.1
- 最小值：0
- 最大值：1

14.5.2.19.8 merge-small-file-threshold

- Blob 文件的大小小于该值时，无视 discardable-ratio 仍可能被 GC 选中。
- 默认值：8MB
- 最小值：0
- 单位：KB|MB|GB

14.5.2.19.9 blob-run-mode

- Titan 的运行模式选择。
- 可选值：
  - “normal”：value size 超过 min-blob-size 的数据会写入到 blob 文件。
  - “read_only”：不再写入新数据到 blob，原有 blob 内的数据仍然可以读取。
  - “fallback”：将 blob 内的数据写回 LSM。
- 默认值：“normal”

14.5.2.19.10 level-merge

- 是否通过开启 level-merge 来提升读性能，副作用是写放大会比不开启更大。
- 默认值：false

14.5.2.20 raftdb

raftdb 相关配置项。

14.5.2.20.1 max-background-jobs

- RocksDB 后台线程个数。调整 RocksDB 线程池的大小时，请参考TiKV 线程池调优。
- 默认值：4
- 最小值：2

14.5.2.20.2 max-sub-compactions

- RocksDB 进行 subcompaction 的并发数。
- 默认值：2
- 最小值：1

14.5.2.20.3 max-open-files

- RocksDB 可以打开的文件总数。
- 默认值：40960
- 最小值：-1

14.5.2.20.4 max-manifest-file-size

- 单个 RocksDB Manifest 文件的最大大小。
- 默认值："20MB"
- 最小值：0
- 单位：B|KB|MB|GB

14.5.2.20.5 create-if-missing

- 如果值为 true，当数据库不存在时将自动创建。
- 默认值：true

14.5.2.20.6 stats-dump-period

- 输出统计信息到日志的时间间隔。
- 默认值：10m

14.5.2.20.7 wal-dir

- 存储 Raft RocksDB WAL 文件的目录，即 WAL 的绝对路径。请勿将该配置项设置为与rocksdb.wal-dir 相同的值。
- 如果未设置该配置项，日志文件将存储在与数据相同的目录中。
- 如果机器上有两个磁盘，将 RocksDB 数据和 WAL 日志存储在不同磁盘上可以提高性能。
- 默认值：""

14.5.2.20.8 wal-ttl-seconds

- 归档的 WAL 文件的保留时间。当超过该值时，系统将删除这些文件。
- 默认值：0
- 最小值：0
- 单位：秒

14.5.2.20.9 wal-size-limit

- 归档 WAL 文件的大小限制。当超过该值时，系统将删除这些文件。
- 默认值：0
- 最小值：0
- 单位：B|KB|MB|GB

14.5.2.20.10 max-total-wal-size

- RocksDB WAL 文件的最大总大小。
- 默认值：`"4GB"`

14.5.2.20.11 compaction-readahead-size

- 控制在 RocksDB compaction 时是否开启预读取功能，并指定预读取数据的大小。
- 如果使用机械硬盘，建议将该值至少设置为 2MB。
- 默认值：`0`
- 最小值：`0`
- 单位：`B|KB|MB|GB`

14.5.2.20.12 writable-file-max-buffer-size

- WriteableFileWrite 中使用的最大缓冲区大小。
- 默认值：`"1MB"`
- 最小值：`0`
- 单位：`B|KB|MB|GB`

14.5.2.20.13 use-direct-io-for-flush-and-compaction

- 控制是否在后台刷新和 compaction 时使用 `O_DIRECT` 进行读写。启用 `O_DIRECT` 的性能影响：它可以绕过和防止操作系统缓存污染，但是后续文件读取需要重新读取内容到缓存中。
- 默认值：`false`

14.5.2.20.14 enable-pipelined-write

- 控制是否开启 Pipelined Write。开启时会使用旧的 Pipelined Write，关闭时会使用新的 Pipelined Commit 机制。
- 默认值：`true`

14.5.2.20.15 allow-concurrent-memtable-write

- 控制是否开启并发 memtable 写入。
- 默认值：`true`

14.5.2.20.16 bytes-per-sync

- 异步 Sync 限速速率。
- 默认值：`"1MB"`
- 最小值：`0`
- 单位：`B|KB|MB|GB`

14.5.2.20.17 wal-bytes-per-sync

- WAL Sync 限速速率。
- 默认值：`"512KB"`
- 最小值：`0`
- 单位：`B|KB|MB|GB`

14.5.2.20.18 info-log-max-size

- Info 日志的最大大小。
- 默认值：`"1GB"`
- 最小值：`0`
- 单位：`B|KB|MB|GB`

14.5.2.20.19 info-log-roll-time

- Info 日志截断间隔时间，如果为 `"0s"` 则不截断。
- 默认值：`"0s"`

14.5.2.20.20 info-log-keep-log-file-num

- RaftDB 中保存的 Info 日志文件的最大数量。
- 默认值：`10`
- 最小值：`0`

14.5.2.20.21 info-log-dir

- Info 日志存储的目录。
- 默认值：`""`

14.5.2.20.22 info-log-level

- RaftDB 的日志级别。
- 默认值：`"info"`

14.5.2.21 raft-engine

Raft Engine 相关的配置项。

> **注意：**

- 第一次开启 Raft Engine 时，TiKV 会将原有的 RocksDB 数据转移至 Raft Engine 中。因此，TiKV 的启动时间会比较长，你需要额外等待几十秒。
- 如果你要将 TiDB 集群降级至 v5.4.0 以前的版本（不含 v5.4.0），你需要在降级之前先关闭 Raft Engine（即把 `enable` 配置项设置为 `false`，并重启 TiKV 使配置生效），否则会导致集群降级后无法正常开启。

14.5.2.21.1 enable

- 决定是否使用 Raft Engine 来存储 Raft 日志。开启该配置项后，`raftdb` 的配置不再生效。
- 默认值：`true`

14.5.2.21.2 dir

- 存储 Raft 日志文件的目录。如果该目录不存在，则在启动 TiKV 时创建该目录。
- 如果未设置此配置，则使用 `{data-dir}/raft-engine`。
- 如果你的机器上有多个磁盘，建议将 Raft Engine 的数据存储在单独的磁盘上，以提高 TiKV 性能。
- 默认值：`""`

14.5.2.21.3 batch-compression-threshold

- 指定日志批处理的阈值大小。大于此配置的日志批次将被压缩。如果将此配置项设置为 `0`，则禁用压缩。
- 默认值：`"8KB"`

14.5.2.21.4 bytes-per-sync

- 指定缓存写入的最大累积大小。当超过此配置值时，缓存的写入将被刷写到磁盘。
- 如果将此配置项设置为 `0`，则禁用增量同步。
- 默认值：`"4MB"`

14.5.2.21.5 target-file-size

- 指定日志文件的最大大小。当日志文件大于此值时，将对其进行轮转。
- 默认值：`"128MB"`

14.5.2.21.6 purge-threshold

- 指定主日志队列的阈值大小。当超过此配置值时，将对主日志队列执行垃圾回收。
- 此参数可用于调整 Raft Engine 的空间占用大小。
- 默认值：`"10GB"`

14.5.2.21.7 recovery-mode

- 确定在日志恢复过程中如何处理文件损坏。
- 可选值：`"absolute-consistency"`, `"tolerate-tail-corruption"`, `"tolerate-any-corruption"`
- 默认值：`"tolerate-tail-corruption"`

14.5.2.21.8 recovery-read-block-size

- 恢复期间读取日志文件的最小 I/O 大小。
- 默认值：`"16KB"`
- 最小值：`"512B"`

14.5.2.21.9 recovery-threads

- 用于扫描和恢复日志文件的线程数。
- 默认值：4
- 最小值：1

14.5.2.21.10 memory-limit

- 指定 Raft Engine 使用内存的上限。
- 当该配置项未设置时，Raft Engine 默认使用系统总内存的 15%。
- 默认值：系统总内存 * 15%

14.5.2.21.11 format-version 从 v6.3.0 版本开始引入

> **注意：**
>
> format-version 的值设置为 2 后，如果你需要将 TiKV 集群降级至 v6.3.0 以前的版本，你需要在降级之前执行如下操作：
>
> 1. 关闭 Raft Engine。将enable 配置项设置为 false，并重启 TiKV 使配置生效。
> 2. 将 format-version 的值重新设置为 1。
> 3. 重新打开 Raft Engine，即把 enable 配置项重设为 true，并重启 TiKV 使配置生效。

- 指定 Raft Engine 的日志文件格式版本。
- 可选值：
  - 1：v6.3.0 以前的默认日志文件格式。v6.1.0 及以后版本的 TiKV 可以读取该格式。
  - 2：支持日志回收。v6.3.0 及以后版本的 TiKV 可以读取该格式。
- 默认值：2

14.5.2.21.12 enable-log-recycle 从 v6.3.0 版本开始引入

> **注意：**
>
> 仅在 `format-version` 的值大于等于 2 时，该配置项才生效。

- 控制 Raft Engine 是否回收过期的日志文件。该配置项启用时，Raft Engine 将保留逻辑上被清除的日志文件，用于日志回收，减少写负载的长尾延迟。
- 默认值：`true`

14.5.2.21.13 prefill-for-recycle 从 v7.0.0 版本开始引入

> **注意：**
>
> 仅在 `enable-log-recycle` 的值为 `true` 时，该配置项才生效。

- 控制 Raft Engine 是否自动生成空的日志文件用于日志回收。该配置项启用时，Raft Engine 将在初始化时自动填充一批空日志文件用于日志回收，保证日志回收在初始化后立即生效。
- 默认值：`false`

14.5.2.22 security

安全相关配置项。

14.5.2.22.1 ca-path

- CA 文件路径。
- 默认值：“”

14.5.2.22.2 cert-path

- 包含 X.509 证书的 PEM 文件路径。
- 默认值：“”

14.5.2.22.3 key-path

- 包含 X.509 key 的 PEM 文件路径。
- 默认值：“”

14.5.2.22.4 cert-allowed-cn

- 客户端提供的证书中，可接受的 X.509 通用名称列表。仅当提供的通用名称与列表中的条目之一完全匹配时，才会允许其请求。
- 默认值：`[]`。这意味着默认情况下禁用客户端证书 CN 检查。

14.5.2.22.5 redact-info-log 从 v4.0.8 版本开始引入

- 若开启该选项，日志中的用户数据会以 ? 代替。
- 默认值：`false`

14.5.2.23 security.encryption

静态加密 (TDE) 有关的配置项。

14.5.2.23.1 data-encryption-method

- 数据文件的加密方法。
- 可选值：`"plaintext"`，`"aes128-ctr"`，`"aes192-ctr"`，`"aes256-ctr"`，`"sm4-ctr"`（从 v6.3.0 开始支持）
- 选择 `"plaintext"` 以外的值则表示启用加密功能。此时必须指定主密钥。
- 默认值：`"plaintext"`

14.5.2.23.2 data-key-rotation-period

- 指定 TiKV 轮换数据密钥的频率。
- 默认值：`7d`

14.5.2.23.3 enable-file-dictionary-log

- 启用优化，以减少 TiKV 管理加密元数据时的 I/O 操作和互斥锁竞争。
- 此配置参数默认启用，为避免可能出现的兼容性问题，请参考静态加密 - TiKV 版本间兼容性。
- 默认值：`true`

14.5.2.23.4 master-key

- 指定启用加密时的主密钥。若要了解如何配置主密钥，可以参考静态加密 - 配置加密。

14.5.2.23.5 previous-master-key

- 指定轮换新主密钥时的旧主密钥。旧主密钥的配置格式与主密钥相同。若要了解如何配置主密钥，可以参考静态加密 - 配置加密。

14.5.2.24 import

用于 TiDB Lightning 导入及 BR 恢复相关的配置项。

14.5.2.24.1 num-threads

- 处理 RPC 请求的线程数量。
- 默认值：8
- 最小值：1

14.5.2.24.2 stream-channel-window

- Stream channel 的窗口大小。当 channel 满时，Stream 会被阻塞。
- 默认值：128

14.5.2.24.3 memory-use-ratio 从 v6.5.0 版本开始引入

- 从 v6.5.0 开始，PITR 支持直接将备份日志文件读取到缓存中，然后进行恢复。此配置项用来配置 PITR 恢复中可用内存与系统总内存的占比。
- 可调整范围：[0.0, 0.5]
- 默认值：0.3，表示系统 30% 的内存可用于 PITR 恢复；当为 0.0 时，表示通过下载日志文件到本地进行 PITR 恢复。

> **注意：**
>
> 在小于 v6.5.0 的版本中，PITR 仅支持将备份文件下载到本地进行恢复。

14.5.2.25 gc

14.5.2.25.1 batch-keys

- 一次 GC 操作中的 key 的数量。
- 默认值：512

14.5.2.25.2 max-write-bytes-per-sec

- GC 工作线程每秒可以写入 RocksDB 的最大字节数。
- 如果设置为 0，则没有限制。
- 默认值："0"

14.5.2.25.3 enable-compaction-filter 从 v5.0 版本开始引入

- 是否开启 GC in Compaction Filter 特性。
- 默认值：true

14.5.2.25.4 ratio-threshold

- 触发 GC 的垃圾比例阈值。
- 默认值：1.1

14.5.2.26 backup

用于 BR 备份相关的配置项。

14.5.2.26.1 num-threads

- 处理备份的工作线程数量。
- 默认值：CPU * 0.5，但最大为 8
- 可调整范围：[1, CPU]
- 最小值：1

14.5.2.26.2 batch-size

- 一次备份的数据范围数量。
- 默认值：8

14.5.2.26.3 sst-max-size

- 备份 SST 文件大小的阈值。如果 TiKV Region 中备份文件的大小超过该阈值，则将该文件备份到 Region 分割的多个 Region 文件中，每个分割 Region 中的文件大小均为 `sst-max-size`（或略大）。
- 例如，当 Region `[a,e)` 中备份文件大小超过 `sst-max-size` 时，该文件会被备份到多个 Region 范围中，分别为 Region `[a,b)`、`[b,c)`、`[c,d)` 和 `[d,e)`，并且 `[a,b)`、`[b,c)` 和 `[c,d)` 的大小均为 `sst-max-size`（或略大）。
- 默认值："144MB"

14.5.2.26.4 enable-auto-tune 从 v5.4 版本开始引入

- 在集群资源占用率较高的情况下，是否允许 BR 自动限制备份使用的资源，减少对集群的影响。详情见自动调节。
- 默认值：true

14.5.2.26.5 s3-multi-part-size 从 v5.3.2 版本开始引入

> 注意：
>
> 引入该配置项是为了解决备份期间遇到的 S3 限流导致备份失败的问题。该问题已通过优化 BR 备份数据存储的目录结构得到解决。因此，该配置项自 v6.1.1 起开始废弃，不再推荐使用。

- 备份阶段 S3 分块上传的块大小。可通过调整该参数来控制备份时发往 S3 的请求数量。
- TiKV 备份数据到 S3 时，如果备份文件大于该配置项的值，会自动进行分块上传。根据压缩率的不同，96 MiB Region 产生的备份文件大约在 10 MiB~30 MiB 之间。
- 默认值：5MiB

14.5.2.27 backup.hadoop

14.5.2.27.1 home

- 指定 HDFS shell 命令的位置，并且允许 TiKV 找到该 shell 命令。该配置项与环境变量 `$HADOOP_HOME` 有相同的效果。
- 默认值：""

14.5.2.27.2 linux-user

- 指定 TiKV 运行 HDFS shell 命令的 Linux 用户。
- 如果未设置该配置项，TiKV 会使用当前 Linux 用户。
- 默认值：""

14.5.2.28 log-backup

用于日志备份相关的配置项。

14.5.2.28.1 enable 从 v6.2.0 版本开始引入

- 用于开启日志备份功能。
- 默认值：true

14.5.2.28.2 file-size-limit 从 v6.2.0 版本开始引入

- 日志备份任务中，保存到存储的备份文件大小。
- 默认值：256MiB
- 注意：一般情况下，`file-size-limit` 的值会大于外部存储上显示的备份文件大小，这是因为备份文件在上传时会被压缩。

14.5.2.28.3 `initial-scan-pending-memory-quota` 从 v6.2.0 版本开始引入

- 日志备份任务在扫描增量数据时，用于存放扫描数据的缓存大小。
- 默认值：`min(机器总内存 * 10%, 512 MB)`

14.5.2.28.4 `initial-scan-rate-limit` 从 v6.2.0 版本开始引入

- 日志备份任务在扫描增量数据时的吞吐限流参数。
- 默认值：60，即默认限流 60 MB/s

14.5.2.28.5 `max-flush-interval` 从 v6.2.0 版本开始引入

- 日志备份任务将备份数据写入到外部存储的最大间隔时间。
- 默认值：3min

14.5.2.28.6 `num-threads` 从 v6.2.0 版本开始引入

- 日志备份功能占用的线程数目。
- 默认值：CPU * 0.5
- 可调整范围：[2, 12]

14.5.2.28.7 `temp-path` 从 v6.2.0 版本开始引入

- 日志文件存放的临时目录，日志文件预先写入临时目录，然后 flush 到外部存储中。
- 默认值：`${deploy-dir}/data/log-backup-temp`

14.5.2.29 cdc

用于 TiCDC 捕捉变更数据相关的配置项。

14.5.2.29.1 `min-ts-interval`

- 定期推进 Resolved TS 的时间间隔。
- 默认值：200ms

14.5.2.29.2 `old-value-cache-memory-quota`

- 缓存在内存中的 TiCDC Old Value 的条目占用内存的上限。
- 默认值：512MB

14.5.2.29.3 `sink-memory-quota`

- 缓存在内存中的 TiCDC 数据变更事件占用内存的上限。
- 默认值：512MB

14.5.2.29.4 incremental-scan-speed-limit

- 增量扫描历史数据的速度上限。
- 默认值：128MB，即 128MB 每秒。

14.5.2.29.5 incremental-scan-threads

- 增量扫描历史数据任务的线程个数。
- 默认值：4，即 4 个线程

14.5.2.29.6 incremental-scan-concurrency

- 增量扫描历史数据任务的最大并发执行个数。
- 默认值：6，即最多并发执行 6 个任务
- 注意：incremental-scan-concurrency 需要大于等于 incremental-scan-threads，否则 TiKV 启动会报错。

14.5.2.30 resolved-ts

用于维护 Resolved TS 以服务 Stale Read 请求的相关配置项。

14.5.2.30.1 enable

- 是否为所有 Region 维护 Resolved TS。
- 默认值：true

14.5.2.30.2 advance-ts-interval

- 定期推进 Resolved TS 的时间间隔。
- 默认值：20s

14.5.2.30.3 scan-lock-pool-size

- 初始化 Resolved TS 时 TiKV 扫描 MVCC（多版本并发控制）锁数据的线程个数。
- 默认值：2，即 2 个线程

14.5.2.31 pessimistic-txn

悲观事务使用方法请参考TiDB 悲观事务模式。

14.5.2.31.1 wait-for-lock-timeout

- 悲观事务在 TiKV 中等待其他事务释放锁的最长时间。若超时则会返回错误给 TiDB 并由 TiDB 重试加锁，语句最长等锁时间由 innodb_lock_wait_timeout 控制。
- 默认值：1s
- 最小值：1ms

14.5.2.31.2 wake-up-delay-duration

- 悲观事务释放锁时，只会唤醒等锁事务中 start_ts 最小的事务，其他事务将会延迟 wake-up-delay-duration 之后被唤醒。
- 默认值：20ms

14.5.2.31.3 pipelined

- 开启流水线式加悲观锁流程。开启该功能后，TiKV 在检测数据满足加锁要求后，立刻通知 TiDB 执行后面的请求，并异步写入悲观锁，从而降低大部分延迟，显著提升悲观事务的性能。但有较低概率出现悲观锁异步写入失败的情况，可能会导致悲观事务提交失败。
- 默认值：true

14.5.2.31.4 in-memory 从 v6.0.0 版本开始引入

- 开启内存悲观锁功能。开启该功能后，悲观事务会尽可能在 TiKV 内存中存储悲观锁，而不将悲观锁写入磁盘，也不将悲观锁同步给其他副本，从而提升悲观事务的性能。但有较低概率出现悲观锁丢失的情况，可能会导致悲观事务提交失败。
- 默认值：true
- 注意：in-memory 仅在 pipelined 为 true 时生效。

14.5.2.32 quota

用于请求限流 (Quota Limiter) 相关的配置项。

14.5.2.32.1 max-delay-duration 从 v6.0.0 版本开始引入

- 单次读写请求被强制等待的最大时间。
- 默认值：500ms
- 推荐设置：一般使用默认值即可。如果实例出现了内存溢出或者是剧烈的性能抖动，可以设置为 1S，使得请求被延迟调节的时间不超过 1 秒。

14.5.2.32.2 前台限流

用于前台限流相关的配置项。

当 TiKV 部署的机型资源有限（如 4v CPU，16 G 内存）时，如果 TiKV 前台处理的读写请求量过大，以至于占用 TiKV 后台处理请求所需的 CPU 资源，最终影响 TiKV 性能的稳定性。此时，你可以使用前台限流相关的 quota 配置项以限制前台各类请求占用的 CPU 资源。触发该限制的请求会被强制等待一段时间以让出 CPU 资源。具体等待时间与新增请求量相关，最多不超过max-delay-duration 的值。

foreground-cpu-time 从 v6.0.0 版本开始引入

- 限制处理 TiKV 前台读写请求所使用的 CPU 资源使用量，这是一个软限制。
- 默认值：0（即无限制）

- 单位：millicpu（当该参数值为 1500 时，前端请求会消耗 1.5v CPU）。
- 推荐设置：对于 4 核以上的实例，使用默认值 0 即可；对 4 核实例，设置为 1000 到 1500 之间的值能取得比较均衡的效果；对 2 核实例，则不要超过 1200。

`foreground-write-bandwidth` 从 v6.0.0 版本开始引入

- 限制前台事务写入的带宽，这是一个软限制。
- 默认值：0KB（即无限制）
- 推荐设置：除非因为 `foreground-cpu-time` 设置不足以对写带宽做限制，一般情况下本配置项使用默认值 0 即可；否则，在 4 核及 4 核以下规格实例上，建议设置在 50MB 以下。

`foreground-read-bandwidth` 从 v6.0.0 版本开始引入

- 限制前台事务读取数据和 Coprocessor 读取数据的带宽，这是一个软限制。
- 默认值：0KB（即无限制）
- 推荐设置：除非因为 `foreground-cpu-time` 设置不足以对读带宽做限制，一般情况本配置项使用默认值 0 即可；否则，在 4 核及 4 核以下规格实例上，建议设置在 20MB 以内。

14.5.2.32.3 后台限流

用于后台限流相关的配置项。

当 TiKV 部署的机型资源有限（如 4v CPU，16 G 内存）时，如果 TiKV 后台处理的计算或者读写请求量过大，以至于占用 TiKV 前台处理请求所需的 CPU 资源，最终影响 TiKV 性能的稳定性。此时，你可以使用后台限流相关的 quota 配置项以限制后台各类请求占用的 CPU 资源。触发该限制的请求会被强制等待一段时间以让出 CPU 资源。具体等待时间与新增请求量相关，最多不超过`max-delay-duration` 的值。

> **警告：**
>
> - 后台限流是 TiDB 在 v6.2.0 中引入的实验特性，不建议在生产环境中使用。
> - 该功能仅适合在资源有限的环境中使用，以保证 TiKV 在该环境下可以长期稳定地运行。如果在资源丰富的机型环境中开启该功能，可能会导致读写请求量达到峰值时 TiKV 的性能下降的问题。

`background-cpu-time` 从 v6.2.0 版本开始引入

- 限制处理 TiKV 后台读写请求所使用的 CPU 资源使用量，这是一个软限制。
- 默认值：0（即无限制）
- 单位：millicpu（当该参数值为 1500 时，后端请求会消耗 1.5v CPU）。

`background-write-bandwidth` 从 v6.2.0 版本开始引入

> **注意：**
>
> 该配置项可以通过 `SHOW CONFIG` 查询到，但暂未生效。设置该配置项的值不生效。

- 限制后台事务写入的带宽，这是一个软限制。
- 默认值：0KB（即无限制）

`background-read-bandwidth` 从 v6.2.0 版本开始引入

> **注意：**
>
> 该配置项可以通过 `SHOW CONFIG` 查询到，但暂未生效。设置该配置项的值不生效。

- 限制后台事务读取数据和 Coprocessor 读取数据的带宽，这是一个软限制。
- 默认值：0KB（即无限制）

`enable-auto-tune` 从 v6.2.0 版本开始引入

- 是否支持 quota 动态调整。如果打开该配置项，TiKV 会根据 TiKV 实例的负载情况动态调整对后台请求的限制 quota。
- 默认值：false（即关闭动态调整）

14.5.2.33 causal-ts 从 v6.1.0 版本开始引入

用于 TiKV API V2（`storage.api-version = 2`）中时间戳获取相关的配置项。

为了降低写请求延迟，TiKV 会定期获取一批时间戳缓存在本地，避免频繁访问 PD，并容忍短时间的 TSO 服务故障。

14.5.2.33.1 `alloc-ahead-buffer` 从 v6.4.0 版本开始引入

- 预分配 TSO 缓存大小（以时长计算）。
- 表示 TiKV 将按照这个参数指定的时长，预分配 TSO 缓存。TiKV 会根据前一周期的使用情况，预估并请求满足 `alloc-ahead-buffer` 时长所需要的 TSO 数量，缓存在本地。
- 这个参数通常用于提高 TiKV API V2 (`storage.api-version = 2`) 对 PD 故障的容忍度。
- 调大这个参数会增加 TSO 消耗，并增加 TiKV 的内存开销。为了获得足够的 TSO，建议同时调小 PD 的`tso-update-physical-interval` 参数。
- 根据测试，默认配置下，当 PD 主节点由于故障切换到其节点时，写请求会短暂出现延迟增大和 QPS 下降（幅度约 15%）。
- 如果希望业务不受影响，可以尝试采用以下配置：

  - causal-ts.alloc-ahead-buffer = "6s"
  - causal-ts.renew-batch-max-size = 65536
  - causal-ts.renew-batch-min-size = 2048
  - 在 PD 中配置 tso-update-physical-interval = "1ms"

- 默认值：3s

14.5.2.33.2 renew-interval

- 更新本地缓存时间戳的周期。
- TiKV 会每隔 renew-interval 发起一次时间戳更新，并根据前一周期的使用情况以及 alloc-ahead-buffer 参数，来调整时间戳的缓存数量。这个参数配置过大会导致不能及时反映最新的 TiKV 负载变化。而配置过小则会增加 PD 的负载。如果写流量剧烈变化、频繁出现时间戳耗尽、写延迟增加，可以适当调小这个参数，但需要同时关注 PD 的负载情况。
- 默认值：100ms

14.5.2.33.3 renew-batch-min-size

- 单次时间戳请求的最小数量。
- TiKV 会根据前一周期的使用情况以及 alloc-ahead-buffer 参数设置，来调整时间戳的缓存数量。如果 TSO 需求量较低，TiKV 会降低单次 TSO 请求量，直至等于 renew-batch-min-size。如果业务中经常出现突发的大流量写入，可以适当调大这个参数。注意这个参数是单个 tikv-server 的缓存大小，如果配置过大、而同时集群中 tikv-server 较多，会导致 TSO 消耗过快。
- Grafana TiKV-Raw 面板下 Causal timestamp 中的 TSO batch size 是根据业务负载动态调整后的本地缓存数量。可以参考该监控指标值调整这个参数的大小。
- 默认值：100

14.5.2.33.4 renew-batch-max-size 从 v6.4.0 版本开始引入

- 单次时间戳请求的最大数量。
- 在默认的一个 TSO 物理时钟更新周期内 (50ms)，PD 最多提供 262144 个 TSO，超过这个数量后 PD 会暂缓 TSO 请求的处理。这个配置用于避免 PD 的 TSO 消耗殆尽、影响其他业务的使用。如果增大这个参数，建议同时减小 PD 的tso-update-physical-interval 参数，以获得足够的 TSO。
- 默认值：8192

14.5.2.34 resource-control

资源控制 (Resource Control) 在 TiKV 存储层相关的配置项。

14.5.2.34.1 enabled 从 v6.6.0 版本开始引入

- 是否支持对用户前台的读写请求按照对应的资源组配额做优先级调度。有关 TiDB 资源组和资源管控的信息，请参考TiDB 资源管控

- 在 TiDB 侧开启tidb_enable_resource_control 全局变量的情况下，开启这个配置项才有意义。此配置参数开启后，TiKV 会使用优先级队列对排队的用户前台读写请求做调度，调度的优先级和请求所在资源组已经消费的资源量反相关，和对应资源组的配额正相关。
- 默认值：true（即开启按照资源组配额调度）

14.5.3 TiFlash 配置参数

本文介绍了与部署使用 TiFlash 相关的配置参数。

14.5.3.1 PD 调度参数

可通过pd-ctl 调整参数。如果你使用 TiUP 部署，可以用 `tiup ctl:v<CLUSTER_VERSION> pd` 代替 `pd-ctl -u <pd_ip:pd_port>` 命令。

- replica-schedule-limit：用来控制 replica 相关 operator 的产生速度（涉及到下线、补副本的操作都与该参数有关）

> 注意：
>
> 不要超过 `region-schedule-limit`，否则会影响正常 TiKV 之间的 Region 调度。

- store-balance-rate：用于限制每个 TiKV store 或 TiFlash store 的 Region 调度速度。注意这个参数只对新加入集群的 store 有效，如果想立刻生效请用下面的方式。

> 注意：
>
> 4.0.2 版本之后（包括 4.0.2 版本）废弃了 `store-balance-rate` 参数且 `store limit` 命令有部分变化。该命令变化的细节请参考store-limit 文档。

```
- 使用 `pd-ctl -u <pd_ip:pd_port> store limit <store_id> <value>` 命令单独设置某个 store 的
    ↪ Region 调度速度。（`store_id` 可通过 `pd-ctl -u <pd_ip:pd_port> store` 命令获得）
    ↪ 如果没有单独设置，则继承 `store-balance-rate` 的设置。你也可以使用 `pd-ctl -u <pd_ip:
    ↪ pd_port> store limit` 命令查看当前设置值。
```

- replication.location-labels：用来表示 TiKV 实例的拓扑关系，其中 key 的顺序代表了不同标签的层次关系。在 TiFlash 开启的情况下需要使用pd-ctl config placement-rules 来设置默认值，详细可参考geo-distributed-deployment-topology。

14.5.3.2 TiFlash 配置参数

> Tip:
>
> 如果你需要调整配置项的值，请参考修改配置参数进行操作。

14.5.3.2.1 配置文件 tiflash.toml

```
#### TiFlash TCP/HTTP 等辅助服务的监听 host。建议配置成 0.0.0.0，即监听本机所有 IP 地址。
listen_host = "0.0.0.0"
#### TiFlash TCP 服务的端口
tcp_port = 9000
#### TiFlash HTTP 服务的端口
http_port = 8123
#### 数据块元信息的内存 cache 大小限制，通常不需要修改
mark_cache_size = 5368709120
#### 数据块 min-max 索引的内存 cache 大小限制，通常不需要修改
minmax_index_cache_size = 5368709120
#### DeltaIndex 内存 cache 大小限制，默认为 0，代表没有限制
delta_index_cache_size = 0

#### TiFlash 数据的存储路径。如果有多个目录，以英文逗号分隔。
#### 从 v4.0.9 版本开始，不推荐使用 path 及 path_realtime_mode 参数。推荐使用 [storage]
    ↪ 下的配置项代替，这样在多盘部署的场景下能更好地利用节点性能。
#### 从 v5.2.0 版本开始，如果要使用配置项 storage.io_rate_limit，需要同时将 TiFlash
    ↪ 的数据存储路径设置为 storage.main.dir。
#### 当 [storage] 配置项存在的情况下，path 和 path_realtime_mode 两个配置会被忽略。
### path = "/tidb-data/tiflash-9000"
#### 或
### path = "/ssd0/tidb-data/tiflash,/ssd1/tidb-data/tiflash,/ssd2/tidb-data/tiflash"
#### 默认为 false。如果设为 true，且 path 配置了多个目录，表示在第一个目录存放最新数据，
    ↪ 在其他目录存放较旧的数据。
### path_realtime_mode = false

#### TiFlash 临时文件的存放路径。默认使用 [`path` 或者 `storage.latest.dir` 的第一个目录] + "/tmp
    ↪ "
### tmp_path = "/tidb-data/tiflash-9000/tmp"

#### 存储路径相关配置，从 v4.0.9 开始生效
[storage]

    ## DTFile 储存文件格式
    ## * format_version = 2 v6.0.0 以前版本的默认文件格式
    ## * format_version = 3 v6.0.0 及 v6.1.x 版本的默认文件格式，具有更完善的检验功能
```

```
## * format_version = 4 v6.2.0 及以后版本的默认文件格式，优化了写放大问题，
    ↪ 同时减少了后台线程消耗
# format_version = 4

[storage.main]
## 用于存储主要的数据，该目录列表中的数据占总数据的 90% 以上。
dir = [ "/tidb-data/tiflash-9000" ]
## 或
# dir = [ "/ssd0/tidb-data/tiflash", "/ssd1/tidb-data/tiflash" ]

## storage.main.dir 存储目录列表中每个目录的最大可用容量。
## * 在未定义配置项，或者列表中全填 0 时，会使用目录所在的硬盘容量
## * 以 byte 为单位。目前不支持如 "10GB" 的设置
## * capacity 列表的长度应当与 dir 列表长度保持一致
## 例如:
# capacity = [ 10737418240, 10737418240 ]

[storage.latest]
## 用于存储最新的数据，大约占总数据量的 10% 以内，需要较高的 IOPS。
## 默认情况该项可留空。在未配置或者为空列表的情况下，会使用 storage.main.dir 的值。
# dir = [ ]
## storage.latest.dir 存储目录列表中，每个目录的最大可用容量。
# capacity = [ 10737418240, 10737418240 ]

## [storage.io_rate_limit] 相关配置从 v5.2.0 开始引入。
[storage.io_rate_limit]
## 该配置项是 I/O 限流功能的开关，默认关闭。TiFlash 的 I/O
    ↪ 限流功能适用于磁盘带宽较小且磁盘带宽大小明确的云盘场景。
## I/O 限流功能限制下的读写流量总带宽，单位为 Byte，默认值为 0，即默认关闭 I/O 限流功能。
# max_bytes_per_sec = 0
## max_read_bytes_per_sec 和 max_write_bytes_per_sec 的含义和 max_bytes_per_sec 类似，分别指
    ↪ I/O 限流功能限制下的读流量总带宽和写流量总带宽。
## 分别用两个配置项控制读写带宽限制，适用于一些读写带宽限制分开计算的云盘，例如 GCP 上的
    ↪ persistent disk。
## 当 max_bytes_per_sec 配置不为 0 时，优先使用 max_bytes_per_sec。
# max_read_bytes_per_sec = 0
# max_write_bytes_per_sec = 0

## 下面的参数用于控制不同 I/O 流量类型分配到的带宽权重，一般不需要调整。
## TiFlash 内部将 I/O 请求分成 4 种类型：前台写、后台写、前台读、后台读。
## I/O 限流初始化时，TiFlash 会根据下面的权重 (weight) 比例分配带宽。
## 以下默认配置表示每一种流量将获得 25 / (25 + 25 + 25 + 25) = 25% 的权重。
## 如果将 weight 配置为 0，则对应的 I/O 操作不会被限流。
# foreground_write_weight = 25
# background_write_weight = 25
```

```
    # foreground_read_weight = 25
    # background_read_weight = 25
    ## TiFlash 支持根据当前的 I/O 负载情况自动调整各种 I/O 类型的限流带宽，有可能会超过设置的权重
        ↪ 。
    ## auto_tune_sec 表示自动调整的执行间隔，单位为秒。设为 0 表示关闭自动调整。
    # auto_tune_sec = 5

    ## 下面的配置只针对存算分离模式生效，详细请参考 TiFlash 存算分离架构与 S3 支持文档 https://
        ↪ docs.pingcap.com/zh/tidb/v7.0/tiflash-disaggregated-and-s3
    # [storage.s3]
    # endpoint: http://s3.{region}.amazonaws.com # S3 的 endpoint 地址
    # bucket: mybucket                           # TiFlash 的所有数据存储在这个 bucket 中
    # root: /cluster1_data                       # S3 bucket 中存储数据的根目录
    # access_key_id: {ACCESS_KEY_ID}             # 访问 S3 的 ACCESS_KEY_ID
    # secret_access_key: {SECRET_ACCESS_KEY}     # 访问 S3 的 SECRET_ACCESS_KEY

    # [storage.remote.cache]
    # dir: /data1/tiflash/cache        # TiFlash Compute Node 的本地数据缓存目录
    # capacity: 858993459200           # 800 GiB

[flash]
    tidb_status_addr = tidb status 端口地址 # 多个地址以逗号分割
    service_addr =  TiFlash raft 服务 和 coprocessor 服务监听地址

    ## 下面的配置只针对存算分离模式生效，详情请参考 TiFlash 存算分离架构与 S3 支持文档 https://
        ↪ docs.pingcap.com/zh/tidb/v7.0/tiflash-disaggregated-and-s3
    # disaggregated_mode = tiflash_write # 可选值为 tiflash_write 或者 tiflash_compute

### 多个 TiFlash 节点会选一个 master 来负责往 PD 增删 placement rule，通过 flash.flash_cluster
    ↪ 中的参数控制。
[flash.flash_cluster]
    refresh_interval = master 定时刷新有效期
    update_rule_interval = master 定时向 tidb 获取 TiFlash 副本状态并与 pd 交互
    master_ttl = master 选出后的有效期
    cluster_manager_path = pd buddy 所在目录的绝对路径
    log = pd buddy log 路径

[flash.proxy]
    addr = proxy 监听地址，不填则默认是 127.0.0.1:20170
    advertise-addr = 外部访问 addr 的地址，不填则默认是 "addr"
    data-dir = proxy 数据存储路径
    config = proxy 配置文件路径
    log-file = proxy log 路径
    log-level = proxy log 级别，默认是 "info"
    status-addr = 拉取 proxy metrics | status 信息的监听地址，不填则默认是 127.0.0.1:20292
```

```
    advertise-status-addr = 外部访问 status-addr 的地址，不填则默认是 "status-addr"

[logger]
    ## log 级别（支持 "trace"、"debug"、"info"、"warn"、"error"），默认是 "debug"
    level = "debug"
    log = TiFlash log 路径
    errorlog = TiFlash error log 路径
    ## 单个日志文件的大小，默认是 "100M"
    size = "100M"
    ## 最多保留日志文件个数，默认是 10
    count = 10

[raft]
    pd_addr = pd 服务地址 # 多个地址以逗号隔开

[status]
    ## Prometheus 拉取 metrics 信息的端口，默认是 8234
    metrics_port = 8234

[profiles]

[profiles.default]
    ## 存储引擎的 segment 分裂是否使用逻辑分裂。使用逻辑分裂可以减小写放大，
        ↪ 但是会造成一定程度的硬盘空间回收不及时。默认为 false。
    ## 在 v6.2.0 以及后续版本，强烈建议保留默认值 `false`，不要将其修改为 `true`。
        ↪ 具体请参考已知问题 [#5576](https://github.com/pingcap/tiflash/issues/5576)。
    # dt_enable_logical_split = false

    ## 单次查询过程中，节点对中间数据的内存限制
    ## 设置为整数时，单位为 byte，比如 34359738368 表示 32 GiB 的内存限制，0 表示无限制
    ## 设置为 [0.0, 1.0) 之间的浮点数时，指节点总内存的比值，比如 0.8 表示总内存的 80%，0.0
        ↪ 表示无限制
    ## 默认值为 0，表示不限制
    ## 当查询试图申请超过限制的内存时，查询终止执行并且报错
    max_memory_usage = 0

    ## 所有查询过程中，节点对中间数据的内存限制
    ## 设置为整数时，单位为 byte，比如 34359738368 表示 32 GiB 的内存限制，0 表示无限制
    ## 设置为 [0.0, 1.0) 之间的浮点数时，指节点总内存的比值，比如 0.8 表示总内存的 80%，0.0
        ↪ 表示无限制
    ## 默认值为 0.8，表示总内存的 80%
    ## 当查询试图申请超过限制的内存时，查询终止执行并且报错
    max_memory_usage_for_all_queries = 0.8

    ## 从 v5.0 引入，表示 TiFlash Coprocessor 最多同时执行的 cop 请求数量。
```

```
    ↪ 如果请求数量超过了该配置指定的值，多出的请求会排队等待。如果设为 0 或不设置，
    ↪ 则使用默认值，即物理核数的两倍。
cop_pool_size = 0

## 从 v5.0 引入，表示 TiFlash Coprocessor 最多同时执行的 batch 请求数量。
    ↪ 如果请求数量超过了该配置指定的值，多出的请求会排队等待。如果设为 0 或不设置，
    ↪ 则使用默认值，即物理核数的两倍。
batch_cop_pool_size = 0

## 从 v6.1 引入，指定 TiFlash 执行来自 TiDB 的 ALTER TABLE ... COMPACT 请求时，
    ↪ 能同时并行处理的请求数量。
## 如果这个值没有设置或设为了 0，则会采用默认值（1）。
manual_compact_pool_size = 1

## 从 v5.4.0 引入，表示是否启用弹性线程池，这项功能可以显著提高 TiFlash 在高并发场景的 CPU
    ↪ 利用率。默认为 true。
# enable_elastic_threadpool = true

## TiFlash 存储引擎的压缩算法，支持 LZ4、zstd 和 LZ4HC，大小写不敏感。默认使用 LZ4 算法。
dt_compression_method = "LZ4"

## TiFlash 存储引擎的压缩级别，默认为 1。
## 如果 dt_compression_method 设置为 LZ4，推荐将该值设为 1；
## 如果 dt_compression_method 设置为 zstd，推荐将该值设为 -1 或 1，设置为 -1 的压缩率更小，
    ↪ 但是读性能会更好；
## 如果 dt_compression_method 设置为 LZ4HC，推荐将该值设为 9。
dt_compression_level = 1

## 从 v6.2.0 引入，表示 PageStorage 单个数据文件中有效数据的最低比例。
    ↪ 当某个数据文件的有效数据比例低于该值时，会触发 GC 对该文件的数据进行整理。默认为
    ↪ 0.5。
dt_page_gc_threshold = 0.5

## 从 v7.0.0 引入，表示带 group by key 的 HashAggregation 算子在触发 spill 之前的最大可用内存
    ↪ ，超过该阈值之后 HashAggregation 会采用 spill to disk 的方式来减小内存使用。默认值为
    ↪ 0，表示内存使用无限制，即不会触发 spill。
max_bytes_before_external_group_by = 0

## 从 v7.0.0 引入，表示 sort/topN 算子在触发 spill 之前的最大可用内存，超过该阈值之后 sort/
    ↪ TopN 会采用 spill to disk 的方式来减小内存使用。默认值为 0，表示内存使用无限制，
    ↪ 即不会触发 spill。
max_bytes_before_external_sort = 0

## 从 v7.0.0 引入，表示带等值 join 条件的 HashJoin 算子在触发 spill 之前的最大可用内存，
    ↪ 超过该阈值之后 HashJoin 算子会采用 spill to disk 的方式来减小内存使用。默认值为 0，
```

```
        ↪ 表示内存使用无限制，即不会触发 spill。
    max_bytes_before_external_join = 0

#### 安全相关配置，从 v4.0.5 开始生效
[security]
    ## 从 v5.0 引入，控制是否开启日志脱敏
    ## 若开启该选项，日志中的用户数据会以 `?` 代替显示
    ## 注意，tiflash-learner 对应的安全配置选项为 `security.redact-info-log`，需要在 tiflash-
        ↪ learner.toml 中另外开启
    # redact_info_log = false

    ## 包含可信 SSL CA 列表的文件路径。如果你设置了该值，`cert_path` 和 `key_path`
        ↪ 中的路径也需要填写
    # ca_path = "/path/to/ca.pem"
    ## 包含 PEM 格式的 X509 certificate 文件路径
    # cert_path = "/path/to/tiflash-server.pem"
    ## 包含 PEM 格式的 X509 key 文件路径
    # key_path = "/path/to/tiflash-server-key.pem"
```

14.5.3.2.2 配置文件 tiflash-learner.toml

```
[server]
    engine-addr = 外部访问 TiFlash coprocessor 服务的地址

[raftstore]
    ## 处理 Raft 数据落盘的线程池中线程的数量
    apply-pool-size = 4
    ## 处理 Raft 的线程池中线程的数量，即 Raftstore 线程池的大小。
    store-pool-size = 4
    ## 控制处理 snapshot 的线程数，默认为 2。设为 0 则关闭多线程优化
    snap-handle-pool-size = 2
    ## 控制 raft store 持久化 WAL 的最小间隔。通过适当增大延迟以减少 IOPS 占用，默认为 "4ms"，
        ↪ 设为 "0ms" 则关闭该优化。
    store-batch-retry-recv-timeout = "4ms"

[security]
    ## 从 v5.0 引入，控制是否开启日志脱敏
    ## 若开启该选项，日志中的用户数据会以 `?` 代替显示
    ## 默认值为 false
    redact-info-log = false

[security.encryption]
    ## 数据文件的加密方法。
    ## 可选值为 "aes128-ctr"、"aes192-ctr"、"aes256-ctr"、"sm4-ctr"（仅 v6.4.0 及之后版本）和 "
        ↪ plaintext"。
```

```
    ## 默认值为 "plaintext"，即默认不开启加密功能。选择 "plaintext" 以外的值则表示启用加密功能。
        ↪ 此时必须指定主密钥。
    data-encryption-method = "aes128-ctr"
    ## 轮换密钥的频率，默认值：`7d`。
    data-key-rotation-period = "168h" # 7 days

[security.encryption.master-key]
    ## 指定启用加密时的主密钥。若要了解如何配置主密钥，可以参考《静态加密 - 配置加密》：https://
        ↪ docs.pingcap.com/zh/tidb/v7.0/encryption-at-rest#配置加密

[security.encryption.previous-master-key]
    ## 指定轮换新主密钥时的旧主密钥。旧主密钥的配置格式与主密钥相同。若要了解如何配置主密钥，
        ↪ 可以参考《静态加密 - 配置加密》：https://docs.pingcap.com/zh/tidb/v7.0/encryption-at-
        ↪ rest#配置加密
```

除以上几项外，其余功能参数和 TiKV 的配置相同。需要注意的是：key 为 engine 的 label 是保留项，不可手动配置。

14.5.3.2.3 通过拓扑 label 进行副本调度

TiFlash 设置可用区

14.5.3.2.4 多盘部署

TiFlash 支持单节点多盘部署。如果你的部署节点上有多块硬盘，可以通过以下的方式配置参数，提高节点的硬盘 I/O 利用率。TiUP 中参数配置格式参照详细 TiFlash 配置模版。

TiDB 集群版本低于 v4.0.9

TiDB v4.0.9 之前的版本中，TiFlash 只支持将存储引擎中的主要数据分布在多盘上。通过 path（TiUP 中为 data_dir）和 path_realtime_mode 这两个参数配置多盘部署。

多个数据存储目录在 path 中以英文逗号分隔，比如 /nvme_ssd_a/data/tiflash,/sata_ssd_b/data/tiflash,/
↪ sata_ssd_c/data/tiflash。如果你的节点上有多块硬盘，推荐把性能最好的硬盘目录放在最前面，以更好地利用节点性能。

如果节点上有多块相同规格的硬盘，可以把 path_realtime_mode 参数留空（或者把该值明确地设为 false）。这表示数据会在所有的存储目录之间进行均衡。但由于最新的数据仍然只会被写入到第一个目录，因此该目录所在的硬盘会较其他硬盘繁忙。

如果节点上有多块规格不一致的硬盘，推荐把 path_relatime_mode 参数设置为 true，并且把性能最好的硬盘目录放在 path 参数内的最前面。这表示第一个目录只会存放最新数据，较旧的数据会在其他目录之间进行均衡。注意此情况下，第一个目录规划的容量大小需要占总容量的约 10%。

TiDB 集群版本为 v4.0.9 及以上

TiDB v4.0.9 及之后的版本中，TiFlash 支持将存储引擎的主要数据和新数据都分布在多盘上。多盘部署时，推荐使用 [storage] 中的参数，以更好地利用节点的 I/O 性能。但 TiFlash 仍然支持TiDB 集群版本低于 v4.0.9 中的参数。

如果节点上有多块相同规格的硬盘，推荐把硬盘目录填到 `storage.main.dir` 列表中，`storage.latest.dir` 列表留空。TiFlash 会在所有存储目录之间分摊 I/O 压力以及进行数据均衡。

如果节点上有多块规格不同的硬盘，推荐把 I/O 性能较好的硬盘目录配置在 `storage.latest.dir` 中，把 I/O 性能较一般的硬盘目录配置在 `storage.main.dir` 中。例如节点上有一块 NVMe-SSD 硬盘加上两块 SATA-SSD 硬盘，你可以把 `storage.latest.dir` 设为 `["/nvme_ssd_a/data/tiflash"]` 以及把 `storage.main.dir` 设为 `["/sata_ssd_b/data/tiflash", "/sata_ssd_c/data/tiflash"]`。TiFlash 会根据两个目录列表分别进行 I/O 压力分摊及数据均衡。需要注意此情况下，`storage.latest.dir` 中规划的容量大小需要占总规划容量的约 10%。

> **警告：**
>
> `[storage]` 参数从 TiUP v1.2.5 版本开始支持。如果你的 TiDB 版本为 v4.0.9 及以上，请确保你的 TiUP 版本不低于 v1.2.5，否则 `[storage]` 中定义的数据目录不会被 TiUP 纳入管理。

14.5.4 PD 配置文件描述

PD 配置文件比命令行参数支持更多的选项。你可以在 conf/config.toml 找到默认的配置文件。

本文档只阐述未包含在命令行参数中的参数，命令行参数参见PD 配置参数。

> Tip:
>
> 如果你需要调整配置项的值，请参考修改配置参数进行操作。

14.5.4.0.1 name

- PD 节点名称。
- 默认值：`"pd"`
- 如果你需要启动多个 PD，一定要给 PD 使用不同的名字。

14.5.4.0.2 data-dir

- PD 存储数据路径。
- 默认值：`"default.${name}"`

14.5.4.0.3 client-urls

- PD 监听的客户端 URL 列表。
- 默认值：`"http://127.0.0.1:2379"`
- 如果部署一个集群，client URLs 必须指定当前主机的 IP 地址，例如 `"http://192.168.100.113:2379"`，如果是运行在 Docker 则需要指定为 `"http://0.0.0.0:2379"`。

14.5.4.0.4 advertise-client-urls

- 用于外部访问 PD 的 URL 列表。
- 默认值：`"${client-urls}"`
- 在某些情况下，例如 Docker 或者 NAT 网络环境，客户端并不能通过 PD 自己监听的 client URLs 来访问到 PD，这时候，你就可以设置 advertise URLs 来让客户端访问。
- 例如，Docker 内部 IP 地址为 `172.17.0.1`，而宿主机的 IP 地址为 `192.168.100.113` 并且设置了端口映射 `-p 2379:2379`，那么可以设置为 `advertise-client-urls="http://192.168.100.113:2379"`，客户端可以通过 `http://192.168.100.113:2379` 来找到这个服务。

14.5.4.0.5 peer-urls

- PD 节点监听其他 PD 节点的 URL 列表。
- 默认：`"http://127.0.0.1:2380"`
- 如果部署一个集群，peer URLs 必须指定当前主机的 IP 地址，例如 `"http://192.168.100.113:2380"`，如果是运行在 Docker 则需要指定为 `"http://0.0.0.0:2380"`。

14.5.4.0.6 advertise-peer-urls

- 用于其他 PD 节点访问某个 PD 节点的 URL 列表。
- 默认值：`"${peer-urls}"`
- 在某些情况下，例如 Docker 或者 NAT 网络环境，其他节点并不能通过 PD 自己监听的 peer URLs 来访问到 PD，这时候，你就可以设置 advertise URLs 来让其他节点访问
- 例如，Docker 内部 IP 地址为 `172.17.0.1`，而宿主机的 IP 地址为 `192.168.100.113` 并且设置了端口映射 `-p 2380:2380`，那么可以设置为 `advertise-peer-urls="http://192.168.100.113:2380"`，其他 PD 节点可以通过 `http://192.168.100.113:2380` 来找到这个服务。

14.5.4.0.7 initial-cluster

- 初始化 PD 集群配置。
- 默认值：`"{name}=http://{advertise-peer-url}"`
- 例如，如果 name 是“pd”，并且 advertise-peer-urls 是 `"http://192.168.100.113:2380"`，那么 `initial-cluster` 就是 `"pd=http://192.168.100.113:2380"`。
- 如果启动三台 PD，那么 `initial-cluster` 可能就是 `pd1=http://192.168.100.113:2380, pd2=http://192.168.100.114:2380, pd3=192.168.100.115:2380`。

14.5.4.0.8 initial-cluster-state

- 集群初始状态
- 默认值：“new”

14.5.4.0.9 initial-cluster-token

- 用于在集群初始化阶段标识不同的集群。
- 默认值：“pd-cluster”
- 如果先后部署多个集群，且多个集群有相同配置的节点，应指定不同的 token 来隔离不同的集群。

14.5.4.0.10 lease

- PD Leader Key 租约超时时间，超时系统重新选举 Leader。
- 默认值：3
- 单位：秒

14.5.4.0.11 quota-backend-bytes

- 元信息数据库存储空间的大小，默认 8GiB。
- 默认值：8589934592

14.5.4.0.12 auto-compaction-mod

- 元信息数据库自动压缩的模式，可选项为 periodic（按周期），revision（按版本数）。
- 默认值：periodic

14.5.4.0.13 auto-compaction-retention

- compaction-mode 为 periodic 时为元信息数据库自动压缩的间隔时间；compaction-mode 设置为 revision 时为自动压缩的版本数。
- 默认值：1h

14.5.4.0.14 force-new-cluster

- 强制让该 PD 以一个新集群启动，且修改 raft 成员数为 1。
- 默认值：false

14.5.4.0.15 tso-update-physical-interval

- TSO 物理时钟更新周期。
- 在默认的一个 TSO 物理时钟更新周期内 (50ms)，PD 最多提供 262144 个 TSO。如果需要更多的 TSO，可以将这个参数调小。最小值为 1ms。
- 缩短这个参数会增加 PD 的 CPU 消耗。根据测试，相比 50ms 更新周期，更新周期为 1ms 时，PD 的 CPU 占用率 (CPU usage) 将增加约 10%。
- 默认值：50ms
- 最小值：1ms

14.5.4.1 pd-server

pd-server 相关配置项。

14.5.4.1.1 server-memory-limit 从 v6.6.0 版本开始引入

> **警告：**
>
> 在当前版本中，该配置项为实验特性，不建议在生产环境中使用。

- PD 实例的内存限制比例。0 值表示不设内存限制。
- 默认值：0
- 最小值：0
- 最大值：0.99

14.5.4.1.2 server-memory-limit-gc-trigger 从 v6.6.0 版本开始引入

> **警告：**
>
> 在当前版本中，该配置项为实验特性，不建议在生产环境中使用。

- PD 尝试触发 GC 的阈值比例。当 PD 的内存使用达到 server-memory-limit 值 * server-memory-limit-gc-trigger 值时，则会主动触发一次 Golang GC。在一分钟之内只会主动触发一次 GC。
- 默认值：0.7
- 最小值：0.5
- 最大值：0.99

14.5.4.1.3 enable-gogc-tuner 从 v6.6.0 版本开始引入

> **警告：**
>
> 在当前版本中，该配置项为实验特性，不建议在生产环境中使用。

- 是否开启 GOGC Tuner。
- 默认值：false

14.5.4.1.4 gc-tuner-threshold 从 v6.6.0 版本开始引入

> **警告：**
>
> 在当前版本中，该配置项为实验特性，不建议在生产环境中使用。

- GOGC Tuner 自动调节的最大内存阈值比例，即 server-memory-limit 值 * server-memory-limit-gc-trigger 值，超过阈值后 GOGC Tuner 会停止工作。
- 默认值：0.6
- 最小值：0
- 最大值：0.9

14.5.4.1.5 flow-round-by-digit 从 v5.1 版本开始引入

- 默认值：3
- PD 会对流量信息的末尾数字进行四舍五入处理，减少 Region 流量信息变化引起的统计信息更新。该配置项用于指定对 Region 流量信息的末尾进行四舍五入的位数。例如流量 100512 会归约到 101000。默认值为 3。该配置替换了 trace-region-flow。

> **注意：**
>
> 如果是从 v4.0 升级至当前版本，升级后的 flow-round-by-digit 行为和升级前的 trace-region-flow 行为默认保持一致：如果升级前 trace-region-flow 为 false，则升级后 flow-round-by-digit 为 127；如果升级前 trace-region-flow 为 true，则升级后 flow-round-by-digit 为 3。

14.5.4.1.6 min-resolved-ts-persistence-interval 从 v6.0.0 版本开始引入

- 设置 PD leader 对集群中 Resolved TS 最小值进行持久化的间隔时间。如果该值设置为 0，表示禁用该功能。
- 默认值：在 v6.3.0 之前版本中为 "0s"，在 v6.3.0 及之后的版本中为 "1s"，即最小正值。
- 最小值："0s"
- 单位：秒

> **注意：**
>
> 对于从 v6.0.0~v6.2.0 升级上来的集群，min-resolved-ts-persistence-interval 的默认值在升级后将不会发生变化，即仍然为 "0s"。若要开启该功能，需要手动修改该配置项的值。

14.5.4.2 security

安全相关配置项。

14.5.4.2.1 cacert-path

- CA 文件路径
- 默认值：“”

14.5.4.2.2 cert-path

- 包含 X509 证书的 PEM 文件路径
- 默认值：“”

14.5.4.2.3 key-path

- 包含 X509 key 的 PEM 文件路径
- 默认值：“”

14.5.4.2.4 redact-info-log 从 v5.0 版本开始引入

- 控制 PD 日志脱敏的开关
- 该配置项值设为 true 时将对 PD 日志脱敏，遮蔽日志中的用户信息。
- 默认值：false

14.5.4.3 log

日志相关的配置项。

14.5.4.3.1 level

- 指定日志的输出级别。
- 可选值：“debug”，“info”，“warn”，“error”，“fatal”
- 默认值：“info”

14.5.4.3.2 format

- 日志格式。
- 可选值：“text”，“json”
- 默认值：“text”

14.5.4.3.3 disable-timestamp

- 是否禁用日志中自动生成的时间戳。
- 默认值：false

14.5.4.4 log.file

日志文件相关的配置项。

14.5.4.4.1 max-size

- 单个日志文件最大大小，超过该值系统自动切分成多个文件。
- 默认值：300
- 单位：MiB
- 最小值为 1

14.5.4.4.2 max-days

- 日志保留的最长天数。
- 如果未设置本参数或把本参数设置为默认值 0，PD 不清理日志文件。
- 默认：0

14.5.4.4.3 max-backups

- 日志文件保留的最大个数。
- 如果未设置本参数或把本参数设置为默认值 0，PD 会保留所有的日志文件。
- 默认：0

14.5.4.5 metric

监控相关的配置项。

14.5.4.5.1 interval

- 向 Prometheus 推送监控指标数据的间隔时间。
- 默认：15s

14.5.4.6 schedule

调度相关的配置项。

14.5.4.6.1 max-merge-region-size

- 控制 Region Merge 的 size 上限，当 Region Size 大于指定值时 PD 不会将其与相邻的 Region 合并。
- 默认：20
- 单位：MiB

14.5.4.6.2 max-merge-region-keys

- 控制 Region Merge 的 key 上限，当 Region key 大于指定值时 PD 不会将其与相邻的 Region 合并。
- 默认：200000

14.5.4.6.3 patrol-region-interval

- 控制 replicaChecker 检查 Region 健康状态的运行频率，越短则运行越快，通常状况不需要调整
- 默认：10ms

14.5.4.6.4 split-merge-interval

- 控制对同一个 Region 做 split 和 merge 操作的间隔，即对于新 split 的 Region 一段时间内不会被 merge。
- 默认：1h

14.5.4.6.5 max-snapshot-count

- 控制单个 store 最多同时接收或发送的 snapshot 数量，调度受制于这个配置来防止抢占正常业务的资源。
- 默认：64

14.5.4.6.6 max-pending-peer-count

- 控制单个 store 的 pending peer 上限，调度受制于这个配置来防止在部分节点产生大量日志落后的 Region。
- 默认值：64

14.5.4.6.7 max-store-down-time

- PD 认为失联 store 无法恢复的时间，当超过指定的时间没有收到 store 的心跳后，PD 会在其他节点补充副本。
- 默认值：30m

14.5.4.6.8 max-store-preparing-time 从 v6.1.0 版本开始引入

- 控制 store 上线阶段的最长等待时间。在 store 的上线阶段，PD 可以查询该 store 的上线进度。当超过该配置项指定的时间后，PD 会认为该 store 已完成上线，无法再次查询这个 store 的上线进度，但是不影响 Region 向这个新上线 store 的迁移。通常用户无需修改该配置项。
- 默认值：48h

14.5.4.6.9 leader-schedule-limit

- 同时进行 leader 调度的任务个数。
- 默认值：4

14.5.4.6.10 region-schedule-limit

- 同时进行 Region 调度的任务个数
- 默认值：2048

14.5.4.6.11 hot-region-schedule-limit

- 控制同时进行的 hot Region 任务。该配置项独立于 Region 调度。
- 默认值：4

14.5.4.6.12 hot-region-cache-hits-threshold

- 设置识别热点 Region 所需的分钟数。只有当 Region 处于热点状态持续时间超过此分钟数时，PD 才会参与热点调度。
- 默认值：3

14.5.4.6.13 replica-schedule-limit

- 同时进行 replica 调度的任务个数。
- 默认值：64

14.5.4.6.14 merge-schedule-limit

- 同时进行的 Region Merge 调度的任务，设置为 0 则关闭 Region Merge。
- 默认值：8

14.5.4.6.15 high-space-ratio

- 设置 store 空间充裕的阈值。当节点的空间占用比例小于该阈值时，PD 调度时会忽略节点的剩余空间，主要根据实际数据量进行均衡。此配置仅在 `region-score-formula-version = v1` 时生效。
- 默认值：0.7
- 最小值：大于 0
- 最大值：小于 1

14.5.4.6.16 low-space-ratio

- 设置 store 空间不足的阈值。当某个节点的空间占用比例超过该阈值时，PD 会尽可能避免往该节点迁移数据，同时主要根据节点剩余空间大小进行调度，避免对应节点的磁盘空间被耗尽。
- 默认值：0.8
- 最小值：大于 0
- 最大值：小于 1

14.5.4.6.17 tolerant-size-ratio

- 控制 balance 缓冲区大小。
- 默认值：0（为 0 为自动调整缓冲区大小）
- 最小值：0

14.5.4.6.18 enable-cross-table-merge

- 设置是否开启跨表 merge。
- 默认值：true

14.5.4.6.19 region-score-formula-version 从 v5.0 版本开始引入

- 设置 Region 算分公式版本。
- 默认值：v2
- 可选值：v1，v2。v2 相比于 v1，变化会更平滑，空间回收引起的调度抖动情况会得到改善。

> **注意：**
>
> 如果是从 v4.0 升级至当前版本，默认不自动开启该算分公式新版本，以保证升级前后 PD 行为一致。若想切换算分公式的版本，使用需要手动通过 pd-ctl 设置切换，详见PD Control 文档。

14.5.4.6.20 enable-joint-consensus 从 v5.0 版本开始引入

- 是否使用 Joint Consensus 进行副本调度。关闭该特性时，PD 将采用一次调度一个副本的方式进行调度。
- 默认值：true

14.5.4.6.21 enable-diagnostic 从 v6.3.0 版本开始引入

- 是否开启诊断功能。开启特性时，PD 将会记录调度中的一些状态来帮助诊断。开启时会略微影响调度速度，在 Store 数量较多时会消耗较大内存。
- 默认值：false

14.5.4.6.22 hot-regions-write-interval 从 v5.4.0 版本开始引入

- 设置 PD 存储 Hot Region 信息时间间隔。
- 默认值：10m

> **注意：**
>
> Hot Region 的信息一般 3 分钟更新一次。如果设置时间间隔小于 3 分钟，中间部分的更新可能没有意义。

14.5.4.6.23 hot-regions-reserved-days 从 v5.4.0 版本开始引入

- 设置 PD 保留的 Hot Region 信息的最长时间。单位为天。
- 默认值: 7

14.5.4.7 replication

副本相关的配置项。

14.5.4.7.1 max-replicas

- 所有副本数量，即 leader 与 follower 数量之和。默认为 3，即 1 个 leader 和 2 个 follower。当此配置被在线修改后，PD 会在后台通过调度使得 Region 的副本数量符合配置。
- 默认值：3

14.5.4.7.2 location-labels

- TiKV 集群的拓扑信息。
- 默认值：[]
- 配置集群拓扑

14.5.4.7.3 isolation-level

- TiKV 集群的最小强制拓扑隔离级别。
- 默认值：“”
- 配置集群拓扑

14.5.4.7.4 strictly-match-label

- 打开强制 TiKV Label 和 PD 的 location-labels 是否匹配的检查
- 默认值：false

14.5.4.7.5 enable-placement-rules

- 打开 placement-rules
- 默认值：true
- 参考Placement Rules 使用文档

14.5.4.8 label-property

标签相关的配置项。

14.5.4.8.1 key

- 拒绝 leader 的 store 带有的 label key。
- 默认值：“”

14.5.4.8.2 value

- 拒绝 leader 的 store 带有的 label value。
- 默认值：“”

14.5.4.9 dashboard

PD 中内置的TiDB Dashboard 相关配置项。

14.5.4.9.1 tidb-cacert-path

- CA 根证书文件路径。可配置该路径来使用 TLS 连接 TiDB 的 SQL 服务。
- 默认值：“”

14.5.4.9.2 tidb-cert-path

- SSL 证书文件路径。可配置该路径来使用 TLS 连接 TiDB 的 SQL 服务。
- 默认值：“”

14.5.4.9.3 tidb-key-path

- SSL 私钥文件路径。可配置该路径来使用 TLS 连接 TiDB 的 SQL 服务。
- 默认值：“”

14.5.4.9.4 public-path-prefix

- 通过反向代理访问 TiDB Dashboard 时，配置反向代理提供服务的路径前缀。
- 默认值：“/dashboard”
- 若不通过反向代理访问 TiDB Dashboard，请勿配置该项，否则可能导致 TiDB Dashboard 无法正常访问。关于该配置的详细使用场景，参见通过反向代理使用 TiDB Dashboard。

14.5.4.9.5 enable-telemetry

- 是否启用 TiDB Dashboard 遥测功能。
- 默认值：false
- 参阅遥测了解该功能详情。

14.5.4.10 replication-mode

Region 同步模式相关的配置项。更多详情，请参阅启用自适应同步模式。

14.5.4.11 Controllor

PD 中内置的Resource Control 相关的配置项。

14.5.4.11.1 degraded-mode-wait-duration

- 触发降级模式需要等待的时间。降级模式是指在 Local Token Bucket (LTB) 和 Global Token Bucket (GTB) 失联的情况下，LTB 将回退到默认的资源组配置，不再有 GTB 授权 token，从而保证在网络隔离或者异常情况下，服务不受影响。
- 默认值: 0s
- 默认为不开启降级模式

14.5.4.11.2 request-unit

下面是Request Unit (RU) 相关的配置项。

`read-base-cost`

- 每次读请求转换成 RU 的基准系数
- 默认值: 0.25

`write-base-cost`

- 每次写请求转换成 RU 的基准系数
- 默认值: 1

`read-cost-per-byte`

- 读流量转换成 RU 的基准系数
- 默认值: 1/(64 * 1024)
- 1 RU = 64 KiB 读取字节

`write-cost-per-byte`

- 写流量转换成 RU 的基准系数
- 默认值: 1/1024
- 1 RU = 1 KiB 写入字节

`read-cpu-ms-cost`

- CPU 转换成 RU 的基准系数
- 默认值: 1/3
- 1 RU = 3 毫秒 CPU 时间

14.6 CLI

14.6.1 TiKV Control 使用说明

TiKV Control（以下简称 tikv-ctl）是 TiKV 的命令行工具，用于管理 TiKV 集群。它的安装目录如下：

- 如果是使用 TiUP 部署的集群，在 `~/.tiup/components/ctl/{VERSION}/` 目录下。

14.6.1.1 通过 TiUP 使用 TiKV Control

> **注意：**
>
> 建议使用的 Control 工具版本与集群版本保持一致。

`tikv-ctl` 也集成在了 `tiup` 命令中。执行以下命令，即可调用 `tikv-ctl` 工具：

```
tiup ctl:v<CLUSTER_VERSION> tikv
```

```
Starting component `ctl`: /home/tidb/.tiup/components/ctl/v4.0.8/ctl tikv
TiKV Control (tikv-ctl)
Release Version:   4.0.8
Edition:           Community
Git Commit Hash:   83091173e960e5a0f5f417e921a0801d2f6635ae
Git Commit Branch: heads/refs/tags/v4.0.8
UTC Build Time:    2020-10-30 08:40:33
Rust Version:      rustc 1.42.0-nightly (0de96d37f 2019-12-19)
Enable Features:   jemalloc mem-profiling portable sse protobuf-codec
Profile:           dist_release

A tool for interacting with TiKV deployments.

USAGE:
    TiKV Control (tikv-ctl) [FLAGS] [OPTIONS] [SUBCOMMAND]

FLAGS:
    -h, --help                    Prints help information
        --skip-paranoid-checks    Skip paranoid checks when open rocksdb
    -V, --version                 Prints version information

OPTIONS:
        --ca-path <ca-path>              Set the CA certificate path
        --cert-path <cert-path>          Set the certificate path
        --config <config>                TiKV config path, by default it's <deploy-dir>/conf/tikv
            ↪ .toml
```

```
        --data-dir <data-dir>           TiKV data directory path, check <deploy-dir>/scripts/run
            ↪ .sh to get it
        --decode <decode>               Decode a key in escaped format
        --encode <encode>               Encode a key in escaped format
        --to-hex <escaped-to-hex>       Convert an escaped key to hex key
        --to-escaped <hex-to-escaped>   Convert a hex key to escaped key
        --host <host>                   Set the remote host
        --key-path <key-path>           Set the private key path
        --log-level <log-level>         Set the log level [default:warn]
        --pd <pd>                       Set the address of pd

SUBCOMMANDS:
    bad-regions           Get all regions with corrupt raft
    cluster               Print the cluster id
    compact               Compact a column family in a specified range
    compact-cluster       Compact the whole cluster in a specified range in one or more column
        ↪ families
    consistency-check     Force a consistency-check for a specified region
    decrypt-file          Decrypt an encrypted file
    diff                  Calculate difference of region keys from different dbs
    dump-snap-meta        Dump snapshot meta file
    encryption-meta       Dump encryption metadata
    fail                  Inject failures to TiKV and recovery
    help                  Prints this message or the help of the given subcommand(s)
    metrics               Print the metrics
    modify-tikv-config    Modify tikv config, eg. tikv-ctl --host ip:port modify-tikv-config -n
                          rocksdb.defaultcf.disable-auto-compactions -v true
    mvcc                  Print the mvcc value
    print                 Print the raw value
    raft                  Print a raft log entry
    raw-scan              Print all raw keys in the range
    recover-mvcc          Recover mvcc data on one node by deleting corrupted keys
    recreate-region       Recreate a region with given metadata, but alloc new id for it
    region-properties     Show region properties
    scan                  Print the range db range
    size                  Print region size
    split-region          Split the region
    store                 Print the store id
    tombstone             Set some regions on the node to tombstone by manual
    unsafe-recover        Unsafely recover the cluster when the majority replicas are failed
```

你可以在 `tiup ctl:v<CLUSTER_VERSION> tikv` 后面再接上相应的参数与子命令。

14.6.1.2 通用参数

tikv-ctl 提供以下两种运行模式：

- 远程模式。通过 `--host` 选项接受 TiKV 的服务地址作为参数。在此模式下，如果 TiKV 启用了 SSL，则 tikv-ctl 也需要指定相关的证书文件，例如：

```
tikv-ctl --ca-path ca.pem --cert-path client.pem --key-path client-key.pem --host
    ↪ 127.0.0.1:20160 <subcommands>
```

  某些情况下，tikv-ctl 与 PD 进行通信，而不与 TiKV 通信。此时你需要使用 `--pd` 选项而非 `--host` 选项，例如：

```
tikv-ctl --pd 127.0.0.1:2379 compact-cluster
```

```
store:"127.0.0.1:20160" compact db:KV cf:default range:([], []) success!
```

- 本地模式：
  - 通过 `--data-dir` 选项来指定本地 TiKV 数据的目录路径。
  - 通过 `--config` 选项来指定本地 TiKV 配置文件到路径。

在此模式下，需要停止正在运行的 TiKV 实例。

以下如无特殊说明，所有命令都同时支持这两种模式。

除此之外，tikv-ctl 还有两个简单的命令 `--to-hex` 和 `--to-escaped`，用于对 key 的形式作简单的变换。一般使用 escaped 形式，示例如下：

```
tikv-ctl --to-escaped 0xaaff
```

```
\252\377
```

```
tikv-ctl --to-hex "\252\377"
```

```
AAFF
```

> 注意：
>
> 在命令行上指定 escaped 形式的 key 时，需要用双引号引起来，否则 bash 会将反斜杠吃掉，导致结果错误。

14.6.1.3 各项子命令及部分参数、选项

下面逐一对 tikv-ctl 支持的子命令进行举例说明。有的子命令支持很多可选参数，要查看全部细节，可运行 `tikv-ctl --help <subcommand>`。

14.6.1.3.1 查看 Raft 状态机的信息

`raft` 子命令可以查看 Raft 状态机在某一时刻的状态。状态信息包括 RegionLocalState、 RaftLocalState 和 RegionApplyState 三个结构体，及某一条 log 对应的 Entries。

可以使用 `region` 和 `log` 两个子命令分别查询以上信息。两条子命令都同时支持远程模式和本地模式。

对于 `region` 子命令：

- 要查看指定的 Region，可在命令中使用 `-r` 参数，多个 Region 以 , 分隔。也可以使用 `--all-regions` 参数来返回所有 Region（`-r` 与 `--all-regions` 不能同时使用）
- 要限制输出的 Region 的数量，可在命令中使用 `--limit` 参数（默认为 16）
- 要查询某个 key 范围中包含哪些 Region，可在命令中使用 `--start` 和 `--end` 参数（默认不限范围，采用 Hex 格式）

需要输出 ID 为 1239 的 Region 时，用法及输出内容如下所示：

```
tikv-ctl --host 127.0.0.1:20160 raft region -r 1239
```

```
"region id": 1239
"region state": {
    id: 1239,
    start_key: 7480000000000000FF4E5F728000000000FF1443770000000000FA,
    end_key: 7480000000000000FF4E5F728000000000FF21C4420000000000FA,
    region_epoch: {conf_ver: 1 version: 43},
    peers: [ {id: 1240 store_id: 1 role: Voter} ]
}
"raft state": {
    hard_state {term: 8 vote: 5 commit: 7}
    last_index: 8)
}
"apply state": {
    applied_index: 8 commit_index: 8 commit_term: 8
    truncated_state {index: 5 term: 5}
}
```

需要查询某个 key 范围中包含哪些 Region 时，用法及输出内容如下所示：

- 当 key 范围包含在某个 Region 中时，将会输出该 Region 信息。
- 当 key 范围精准到某个 Region 的范围时，以上述 Region 1239 为例：当给定的 key 范围为 Region 1239 的范围时，由于 Region 范围为左闭右开区间，并且 Region 1009 以 Region 1239 的 end_key 作为 `start_key`，因此会同时输出 Region 1009 和 Region 1239 的信息。

```
tikv-ctl --host 127.0.0.1:20160 raft region --start 7480000000000000
    ↪ FF4E5F728000000000FF1443770000000000FA --end 7480000000000000
    ↪ FF4E5F728000000000FF21C4420000000000FA
```

```
"region state": {
    id: 1009
    start_key: 7480000000000000FF4E5F728000000000FF21C4420000000000FA,
    end_key: 7480000000000000FF5000000000000000F8,
    ...
}
"region state": {
    id: 1239
    start_key: 7480000000000000FF4E5F728000000000FF06C6D60000000000FA,
    end_key: 7480000000000000FF4E5F728000000000FF1443770000000000FA,
    ...
}
```

14.6.1.3.2 查看 Region 的大小

使用 `size` 命令可以查看 Region 的大小：

```
tikv-ctl --data-dir /path/to/tikv size -r 2
```

```
region id: 2
cf default region size: 799.703 MB
cf write region size: 41.250 MB
cf lock region size: 27616
```

14.6.1.3.3 扫描查看给定范围的 MVCC

`scan` 命令的 `--from` 和 `--to` 参数接受两个 escaped 形式的 raw key，并用 `--show-cf` 参数指定只需要查看哪些列族。

```
tikv-ctl --data-dir /path/to/tikv scan --from 'zm' --limit 2 --show-cf lock,default,write
```

```
key: zmBootstr\377a\377pKey\000\000\377\000\000\373\000\000\000\000\000\377\000\000s
    ↪ \000\000\000\000\000\372
        write cf value: start_ts: 399650102814441473 commit_ts: 399650102814441475 short_value:
            ↪ "20"
key: zmDB:29\000\000\377\000\374\000\000\000\000\000\000\377\000H\000\000\000\000\000\000\371
        write cf value: start_ts: 399650105239273474 commit_ts: 399650105239273475 short_value:
            ↪ "\000\000\000\000\000\000\000\002"
        write cf value: start_ts: 399650105199951882 commit_ts: 399650105213059076 short_value:
            ↪ "\000\000\000\000\000\000\000\001"
```

14.6.1.3.4 查看给定 key 的 MVCC

与 `scan` 命令类似，`mvcc` 命令可以查看给定 key 的 MVCC：

```
tikv-ctl --data-dir /path/to/tikv mvcc -k "zmDB
    ↪ :29\000\000\377\000\374\000\000\000\000\000\000\377\000H\000\000\000\000\000\000\371" --
    ↪ show-cf=lock,write,default
```

```
key: zmDB:29\000\000\377\000\374\000\000\000\000\000\000\377\000H\000\000\000\000\000\000\371
        write cf value: start_ts: 399650105239273474 commit_ts: 399650105239273475 short_value:
            ↪ "\000\000\000\000\000\000\000\002"
        write cf value: start_ts: 399650105199951882 commit_ts: 399650105213059076 short_value:
            ↪ "\000\000\000\000\000\000\000\001"
```

> **注意：**
>
> 该命令中，key 同样需要是 escaped 形式的 raw key。

14.6.1.3.5 扫描 raw key

使用 `raw-scan` 命令，TiKV 可直接在 RocksDB 中扫描 raw key。

> **注意：**
>
> 如果要扫描数据 key，需要在 key 前添加 `'z'` 前缀。

- 要指定扫描范围，可在 `raw-scan` 命令中使用 `--from` 和 `--to` 参数（默认不限范围）
- 要限制能够打印出的 key 的数量（默认为 30），可在命令中使用 `--limit` 参数
- 要指定扫描的 CF，可在命令中使用 `--cf` 参数（可选值为 `default`，`write`，`lock`）

```
tikv-ctl --data-dir /var/lib/tikv raw-scan --from 'zt' --limit 2 --cf default
```

```
key: "zt\200\000\000\000\000\000\000\377\005_r
    ↪ \200\000\000\000\000\377\000\000\001\000\000\000\000\000\000\372\372b2,^\033\377\364", value:
    ↪  "\010\002\002\002%\010\004\002\010root\010\006\002\000\010\010\t\002\010\n\t
    ↪ \002\010\014\t\002\010\016\t\002\010\020\t\002\010\022\t\002\010\024\t\002\010\026\t
    ↪ \002\010\030\t\002\010\032\t\002\010\034\t\002\010\036\t\002\010 \t\002\010\"\t\002\010s\
    ↪ t\002\010&\t\002\010(\t\002\010*\t\002\010,\t\002\010.\t\002\0100\t\002\0102\t\002\0104\t
    ↪ \002"
key: "zt\200\000\000\000\000\000\000\377\025_r
    ↪ \200\000\000\000\000\377\000\000\023\000\000\000\000\000\000\372\372b2,^\033\377\364", value:
    ↪  "\010\002\002&slow_query_log_file\010\004\002P/usr/local/mysql/data/localhost-slow.log"

Total scanned keys: 2
```

14.6.1.3.6 打印某个 key 的值

打印某个 key 的值需要用到 `print` 命令。示例从略。

14.6.1.3.7 打印 Region 的 properties 信息

为了记录 Region 的状态信息，TiKV 将一些数据写入 Region 的 SST 文件中。你可以用子命令 `region-properties` 运行 tikv-ctl 来查看这些 properties 信息。例如：

```
tikv-ctl --host localhost:20160 region-properties -r 2
```

```
num_files: 0
num_entries: 0
num_deletes: 0
mvcc.min_ts: 18446744073709551615
mvcc.max_ts: 0
mvcc.num_rows: 0
mvcc.num_puts: 0
mvcc.num_versions: 0
mvcc.max_row_versions: 0
middle_key_by_approximate_size:
```

这些 properties 信息可以用于检查某个 Region 是否健康或者修复不健康的 Region。例如，使用 `middle_key_approximate_size` 可以手动分裂 Region。

14.6.1.3.8 手动 compact 单个 TiKV 的数据

`compact` 命令可以对单个 TiKV 进行手动 compact。

- `--from` 和 `--to` 选项以 escaped raw key 形式指定 compact 的范围。如果没有设置，表示 compact 整个 TiKV。
- `--region` 选项指定 compact Region 的范围。如果设置，则 `--from` 和 `--to` 选项会被忽略。
- `-c` 选项指定 column family 名称，默认值为 `default`，可选值为 `default`、`lock` 和 `write`。
- `-d` 选项指定要 compact 的 RocksDB，默认值为 `kv`，可选值为 `kv` 和 `raft`。
- `--threads` 选项可以指定 compact 的并发数，默认值是 8。一般来说，并发数越大，compact 的速度越快，但是也会对服务造成影响，所以需要根据情况选择合适的并发数。
- `--bottommost` 选项可以指定 compact 是否包括最下层的文件。可选值为 `default`、`skip` 和 `force`，默认为 `default`。
  - `default` 表示只有开启了 Compaction Filter 时 compact 才会包括最下层文件。
  - `skip` 表示 compact 不包括最下层文件。
  - `force` 表示 compact 总是包括最下层文件。
- 在本地模式 compact data，执行如下命令：

```
tikv-ctl --data-dir /path/to/tikv compact -d kv
```

- 在远程模式 compact data，执行如下命令：

```
tikv-ctl --host ip:port compact -d kv
```

14.6.1.3.9 手动 compact 整个 TiKV 集群的数据

`compact-cluster` 命令可以对整个 TiKV 集群进行手动 compact。该命令参数的含义和使用与 `compact` 命令一样，唯一的区别如下：

- 使用 `compact-cluster` 命令时，通过 `--pd` 指定 PD 所在的地址，以便 `tikv-ctl` 可以找到集群中的所有 TiKV 节点作为 compact 目标。
- 使用 `compact` 命令时，通过 `--data-dir` 或者 `--host` 指定单个 TiKV 作为 compact 目标。

14.6.1.3.10 设置一个 Region 副本为 tombstone 状态

`tombstone` 命令常用于没有开启 sync-log，因为机器掉电导致 Raft 状态机丢失部分写入的情况。它可以在一个 TiKV 实例上将一些 Region 的副本设置为 Tombstone 状态，从而在重启时跳过这些 Region，避免因为这些 Region 的副本的 Raft 状态机损坏而无法启动服务。这些 Region 应该在其他 TiKV 上有足够多的健康的副本以便能够继续通过 Raft 机制进行读写。

一般情况下，可以先在 PD 上将 Region 的副本通过 `remove-peer` 命令删除掉：

```
pd-ctl>> operator add remove-peer <region_id> <store_id>
```

然后再用 tikv-ctl 在那个 TiKV 实例上将 Region 的副本标记为 tombstone 以便跳过启动时对他的健康检查：

```
tikv-ctl --data-dir /path/to/tikv tombstone -p 127.0.0.1:2379 -r <region_id>
```

```
success!
```

但是有些情况下，当不能方便地从 PD 上移除这个副本时，可以指定 tikv-ctl 的 `--force` 选项来强制设置它为 tombstone：

```
tikv-ctl --data-dir /path/to/tikv tombstone -p 127.0.0.1:2379 -r <region_id>,<region_id> --force
```

```
success!
```

> 注意：
>
> - 该命令只支持本地模式
> - -p 选项的参数指定 PD 的 endpoints，无需 `http` 前缀。指定 PD 的 endpoints 是为了询问 PD 是否可以安全切换至 Tombstone 状态。

14.6.1.3.11 向 TiKV 发出 consistency-check 请求

consistency-check 命令用于在某个 Region 对应的 Raft 副本之间进行一致性检查。如果检查失败，TiKV 自身会 panic。如果 --host 指定的 TiKV 不是这个 Region 的 Leader，则会报告错误。

```
tikv-ctl --host 127.0.0.1:20160 consistency-check -r 2
```

```
success!
```

```
tikv-ctl --host 127.0.0.1:20161 consistency-check -r 2
```

```
DebugClient::check_region_consistency: RpcFailure(RpcStatus { status: Unknown, details: Some("
    ↪ StringError(\"Leader is on store 1\")") })
```

> 注意：
>
> - 目前 consistency-check 与 TiDB GC 操作不兼容，存在误报错误的可能，因此不建议使用该命令。
> - 该命令只支持远程模式。
> - 即使该命令返回了成功信息，也需要检查是否有 TiKV panic 了。因为该命令只是向 Leader 请求进行一致性检查，但整个检查流程是否成功并不能在客户端知道。

14.6.1.3.12 Dump snapshot 元文件

这条子命令可以用于解析指定路径下的 Snapshot 元文件并打印结果。

14.6.1.3.13 打印 Raft 状态机出错的 Region

前面 tombstone 命令可以将 Raft 状态机出错的 Region 设置为 Tombstone 状态，避免 TiKV 启动时对它们进行检查。在运行 tombstone 命令之前，可使用 bad-regions 命令找到出错的 Region，以便将多个工具组合起来进行自动化的处理。

```
tikv-ctl --data-dir /path/to/tikv bad-regions
```

```
all regions are healthy
```

命令执行成功后会打印以上信息，否则会打印出有错误的 Region 列表。目前可以检出的错误包括 last index、commit index 和 apply index 之间的不匹配，以及 Raft log 的丢失。其他一些情况，比如 Snapshot 文件损坏等仍然需要后续的支持。

14.6.1.3.14 查看 Region 属性

- 本地查看部署在 /path/to/tikv 的 TiKV 上面 Region 2 的 properties 信息：

```
tikv-ctl --data-dir /path/to/tikv/data region-properties -r 2
```

- 在线查看运行在 127.0.0.1:20160 的 TiKV 上面 Region 2 的 properties 信息：

```
tikv-ctl --host 127.0.0.1:20160 region-properties -r 2
```

14.6.1.3.15 动态修改 TiKV 的配置

使用 modify-tikv-config 命令可以动态修改配置参数。目前可动态修改的 TiKV 配置与具体的修改行为与 SQL 动态修改配置功能相同，可参考在线修改 TiKV 配置。

- -n 用于指定完整的配置名。支持动态修改的配置名可以参考在线修改 TiKV 配置中支持的配置项列表。
- -v 用于指定配置值。

设置 shared block cache 的大小：

```
tikv-ctl --host ip:port modify-tikv-config -n storage.block-cache.capacity -v 10GB
```

```
success
```

当禁用 shared block cache 时，为 write CF 设置 block cache size：

```
tikv-ctl --host ip:port modify-tikv-config -n rocksdb.writecf.block-cache-size -v 256MB
```

```
success
```

```
tikv-ctl --host ip:port modify-tikv-config -n raftdb.defaultcf.disable-auto-compactions -v true
```

```
success
```

```
tikv-ctl --host ip:port modify-tikv-config -n raftstore.sync-log -v false
```

```
success
```

如果 compaction 的流量控制导致待 compact 数据量 (compaction pending bytes) 堆积，可以禁用 rate-limiter-auto-tuned 配置项或调高 compaction 相关的流量阈值。示例如下：

```
tikv-ctl --host ip:port modify-tikv-config -n rocksdb.rate-limiter-auto-tuned -v false
```

```
success
```

```
tikv-ctl --host ip:port modify-tikv-config -n rocksdb.rate-bytes-per-sec -v "1GB"
```

```
success
```

14.6.1.3.16 强制 Region 从多副本失败状态恢复服务（弃用）

> **警告：**
>
> 不推荐使用该功能，恢复需求可通过 pd-ctl 的 Online Unsafe Recovery 功能实现。它提供了一键式自动恢复的能力，无需停止服务等额外操作，具体使用方式请参考Online Unsafe Recovery 使用文档。

unsafe-recover remove-fail-stores 命令可以将故障机器从指定 Region 的 peer 列表中移除。运行命令之前，需要目标 TiKV 先停掉服务以便释放文件锁。

-s 选项接受多个以逗号分隔的 store_id，并使用 -r 参数来指定包含的 Region。如果要对某一个 store 上的全部 Region 都执行这个操作，可简单指定 --all-regions。

> **警告：**
>
> - 此功能使用不当可能导致集群难以恢复，存在风险。请悉知潜在的风险，尽量避免在生产环境中使用。
> - 如果使用 --all-regions，必须在剩余所有连入集群的 store 上执行此命令。需要保证这些健康的 store 都停掉服务后再进行恢复，否则期间 Region 副本之间的 peer 列表不一致会导致执行 split-region 或者 remove-peer 时报错进而引起其他元数据的不一致，最终引发 Region 不可用。
> - 一旦执行了 remove-fail-stores，不可再重新启动被移除的节点并将其加入集群，否则会导致元数据的不一致，最终引发 Region 不可用。

```
tikv-ctl --data-dir /path/to/tikv unsafe-recover remove-fail-stores -s 3 -r 1001,1002
```

```
success!
```

```
tikv-ctl --data-dir /path/to/tikv unsafe-recover remove-fail-stores -s 4,5 --all-regions
```

之后启动 TiKV，这些 Region 便可以使用剩下的健康副本继续提供服务了。此命令常用于多个 TiKV store 损坏或被删除的情况。

> **注意：**
>
> - 一般来说，您需要为指定 Region 的 peers 所在的每个 store 运行此命令。
> - 该命令只支持本地模式。在运行成功后，会打印 success!。

14.6.1.3.17 恢复损坏的 MVCC 数据

recover-mvcc 命令用于 MVCC 数据损坏导致 TiKV 无法正常运行的情况。为了从不同种类的不一致情况中恢复，该命令会交叉检查 3 个 CF（“default”，“write”，“lock”）。

- -r 选项可以通过 region_id 指定包含的 Region。
- -p 选项可以指定 PD 的 endpoints。

```
tikv-ctl --data-dir /path/to/tikv recover-mvcc -r 1001,1002 -p 127.0.0.1:2379
```

```
success!
```

> 注意：
>
> - 该命令只支持本地模式。在运行成功后，会打印 success!。
> - -p 选项指定 PD 的 endpoint，不使用 http 前缀，用于查询指定的 region_id 是否有效。
> - 对于指定 Region 的 peers 所在的每个 store，均须执行该命令。

14.6.1.3.18 Ldb 命令

ldb 命令行工具提供多种数据访问以及数据库管理命令。下方列出了一些示例用法。详细信息请在运行 tikv-ctl ldb 命令时查看帮助消息或查阅 RocksDB 文档。

数据访问序列的示例如下：

用 HEX 格式 dump 现有 RocksDB 数据：

```
tikv-ctl ldb --hex --db=/tmp/db dump
```

Dump 现有 RocksDB 的声明：

```
tikv-ctl ldb --hex manifest_dump --path=/tmp/db/MANIFEST-000001
```

您可以通过 --column_family=<string> 指定查询的目标列族。

通过 --try_load_options 命令加载数据库选项文件以打开数据库。在数据库运行时，建议您保持该命令为开启的状态。如果您使用默认配置打开数据库，LSM-tree 存储组织可能会出现混乱，且无法自动恢复。

14.6.1.3.19 打印加密元数据

encryption-meta 命令用于打印加密元数据。该子命令可以打印两种加密元数据：数据文件的加密信息，以及所有的数据加密密钥。

使用 encryption-meta dump-file 子命令打印数据文件的加密信息。你需要创建一个 TiKV 配置文件用以指定 TiKV 的数据目录：

```
### conf.toml
[storage]
data-dir = "/path/to/tikv/data"
```

--path 选项可以指定数据文件的绝对或者相对路径。如果指定的文件是明文存储的，本命令有可能没有输出。如果不指定 --path 选项，本命令打印所有数据文件的加密信息。

```
tikv-ctl --config=./conf.toml encryption-meta dump-file --path=/path/to/tikv/data/db/CURRENT
/path/to/tikv/data/db/CURRENT: key_id: 9291156302549018620 iv: E3C2FDBF63FC03BFC28F265D7E78283F
    ↪ method: Aes128Ctr
```

使用 encryption-meta dump-key 打印数据加密密钥。使用本命令的时候，除了在 TiKV 配置文件中指定 TiKV 的数据目录以外，还需要指定当前的主加密密钥。请参阅静态加密文档关于配置 TiKV 主加密密钥的说明。使用本命令时 security.encryption.previous-master-key 配置项不生效，即使配置文件中使用了该配置，本命令也不会触发更换主加密密钥。

```
### conf.toml
[storage]
data-dir = "/path/to/tikv/data"

[security.encryption.master-key]
type = "kms"
key-id = "0987dcba-09fe-87dc-65ba-ab0987654321"
region = "us-west-2"
```

注意如果使用了 AWS KMS 作为主加密密钥，使用本命令时 tikv-ctl 需要该 KMS 密钥的访问权限。KMS 访问权限可以通过环境变量、AWS 默认配置文件或 IAM 的方式传递给 tikv-ctl。详情请参阅相关 AWS 文档。

--ids 选项可以指定以逗号分隔的数据加密密钥 id 列表。如果不指定 --ids 选项，本命令打印所有的数据加密密钥，以及最新的数据加密密钥的 id。

本命令会输出一个警告，提示本命令会泄漏敏感数据。根据提示输入 “I consent” 即可。

```
tikv-ctl --config=./conf.toml encryption-meta dump-key
This action will expose encryption key(s) as plaintext. Do not output the result in file on disk.
Type "I consent" to continue, anything else to exit: I consent
current key id: 9291156302549018620
9291156302549018620: key: 8B6B6B8F83D36BE2467ED55D72AE808B method: Aes128Ctr creation_time:
    ↪ 1592938357
```

```
tikv-ctl --config=./conf.toml encryption-meta dump-key --ids=9291156302549018620
This action will expose encryption key(s) as plaintext. Do not output the result in file on disk.
Type "I consent" to continue, anything else to exit: I consent
9291156302549018620: key: 8B6B6B8F83D36BE2467ED55D72AE808B method: Aes128Ctr creation_time:
    ↪ 1592938357
```

> **注意：**
>
> 本命令会以明文方式打印数据加密密钥。在生产环境中，请勿将本命令的输出重定向到磁盘文件中。即使使用以后删除该文件也不能保证文件内容从磁盘中干净清除。

14.6.1.3.20 打印损坏的 SST 文件信息

TiKV 中损坏的 SST 文件会导致 TiKV 进程崩溃。在 TiDB v6.1.0 之前，损坏的 SST 文件会导致 TiKV 进程立即崩溃。从 TiDB v6.1.0 起，TiKV 进程会在 SST 文件损坏 1 小时后崩溃。

为了方便清理掉这些 SST 文件，你可以先使用 `bad-ssts` 命令打印出损坏的 SST 文件信息。

> **注意：**
>
> 执行此命令前，请保证关闭当前运行的 TiKV 实例。

```
tikv-ctl --data-dir </path/to/tikv> bad-ssts --pd <endpoint>
```

```
--------------------------------------------------------
corruption info:
data/tikv-21107/db/000014.sst: Corruption: Bad table magic number: expected 9863518390377041911,
    ↪ found 759105309091689679 in data/tikv-21107/db/000014.sst
sst meta:
14:552997[1 .. 5520]['0101' seq:1, type:1 .. '7
    ↪ A7480000000000000FF0F5F728000000000FF000216000000000FAFA13AB33020BFFFA' seq:2032, type
    ↪ :1] at level 0 for Column family "default"  (ID 0)
it isn't easy to handle local data, start key:0101
overlap region:
RegionInfo { region: id: 4 end_key: 7480000000000000FF0500000000000000F8 region_epoch { conf_ver:
    ↪  1 version: 2 } peers { id: 5 store_id: 1 }, leader: Some(id: 5 store_id: 1) }
suggested operations:
tikv-ctl ldb --db=data/tikv-21107/db unsafe_remove_sst_file "data/tikv-21107/db/000014.sst"
tikv-ctl --db=data/tikv-21107/db tombstone -r 4 --pd <endpoint>
--------------------------------------------------------
corruption analysis has completed
```

通过上面的输出，你可以看到损坏的 SST 文件和损坏原因等信息先被打印出，然后是相关的元信息。

- 在 `sst meta` 输出部分，14 表示 SST 文件号，552997 表示文件大小，紧随其后的是最小和最大的序列号 (seq) 等其它元信息。
- `overlap region` 部分为损坏 SST 文件所在 Region 的信息。该信息是从 PD 组件获取的。
- `suggested operations` 部分为你清理损坏的 SST 文件提供建议操作。你可以参考这些建议的命令，清理文件，并重新启动该 TiKV 实例。

14.6.2 PD Control 使用说明

PD Control 是 PD 的命令行工具，用于获取集群状态信息和调整集群。

14.6.2.1 安装方式

> **注意：**
>
> 建议使用的 Control 工具版本与集群版本保持一致。

14.6.2.1.1 使用 TiUP

可直接通过 `tiup ctl:v<CLUSTER_VERSION> pd -u http://<pd_ip>:<pd_port> [-i]` 使用。

14.6.2.1.2 下载安装包

如需下载最新版本的 `pd-ctl`，直接下载 TiDB 安装包即可。`pd-ctl` 位于 TiDB 安装包的 `ctl-{version}-linux-{arch}.tar.gz` 包中。

安装包	操作系统	架构	SHA256 校验和
`https://download.pingcap.org/tidb-community-server-{version}-linux-amd64.tar.gz` (pd-ctl)	Linux	amd64	`https://download.pingcap.org/tidb-community-server-{version}-linux-amd64.tar.gz.sha256`
`https://download.pingcap.org/tidb-community-server-{version}-linux-arm64.tar.gz` (pd-ctl)	Linux	arm64	`https://download.pingcap.org/tidb-community-server-{version}-linux-arm64.tar.gz.sha256`

> **注意：**
>
> 下载链接中的 `{version}` 为 TiDB 的版本号。例如，amd64 架构的 `v7.0.0` 版本的下载链接为 `https://download.pingcap.org/tidb-community-server-v7.0.0-linux-amd64.tar.gz`。

14.6.2.1.3 源码编译

1. Go 1.20 或以上版本
2. 在 PD 项目根目录使用 `make` 或者 `make pd-ctl` 命令进行编译，生成 bin/pd-ctl

14.6.2.2 简单例子

单命令模式：

```
tiup ctl:v<CLUSTER_VERSION> pd store -u http://127.0.0.1:2379
```

交互模式：

```
tiup ctl:v<CLUSTER_VERSION> pd -i -u http://127.0.0.1:2379
```

使用环境变量：

```
export PD_ADDR=http://127.0.0.1:2379 &&
tiup ctl:v<CLUSTER_VERSION> pd
```

使用 TLS 加密：

```
tiup ctl:v<CLUSTER_VERSION> pd -u https://127.0.0.1:2379 --cacert="path/to/ca" --cert="path/to/
    ↪ cert" --key="path/to/key"
```

14.6.2.3 命令行参数 (flags)

14.6.2.3.1 `--cacert`

- 指定 PEM 格式的受信任 CA 证书的文件路径
- 默认值： “”

14.6.2.3.2 `--cert`

- 指定 PEM 格式的 SSL 证书的文件路径
- 默认值： “”

14.6.2.3.3 `--detach / -d`

- 使用单命令行模式（不进入 readline）
- 默认值：true

14.6.2.3.4 `--help / -h`

- 输出帮助信息
- 默认值：false

14.6.2.3.5 `--interact/-i`

- 使用交互模式（进入 readline）
- 默认值：false

14.6.2.3.6 --key

- 指定 PEM 格式的 SSL 证书密钥文件路径，即 --cert 所指定的证书的私钥
- 默认值：“”

14.6.2.3.7 --pd/-u

- 指定 PD 的地址
- 默认地址：http://127.0.0.1:2379
- 环境变量：PD_ADDR

14.6.2.3.8 --version/-V

- 打印版本信息并退出
- 默认值：false

14.6.2.4 命令 (command)

14.6.2.4.1 cluster

用于显示集群基本信息。

示例：

```
cluster
```

```
{
  "id": 6493707687106161130,
  "max_peer_count": 3
}
```

14.6.2.4.2 config [show | set <option> <value> | placement-rules]

用于显示或调整配置信息。示例如下。

显示 scheduling 的相关 config 信息：

```
config show
```

```
{
  "replication": {
    "enable-placement-rules": "true",
    "isolation-level": "",
    "location-labels": "",
    "max-replicas": 3,
    "strictly-match-label": "false"
```

```
  },
  "schedule": {
    "enable-cross-table-merge": "true",
    "high-space-ratio": 0.7,
    "hot-region-cache-hits-threshold": 3,
    "hot-region-schedule-limit": 4,
    "leader-schedule-limit": 4,
    "leader-schedule-policy": "count",
    "low-space-ratio": 0.8,
    "max-merge-region-keys": 200000,
    "max-merge-region-size": 20,
    "max-pending-peer-count": 64,
    "max-snapshot-count": 64,
    "max-store-down-time": "30m0s",
    "merge-schedule-limit": 8,
    "patrol-region-interval": "10ms",
    "region-schedule-limit": 2048,
    "region-score-formula-version": "v2",
    "replica-schedule-limit": 64,
    "scheduler-max-waiting-operator": 5,
    "split-merge-interval": "1h0m0s",
    "tolerant-size-ratio": 0
  }
}
```

显示所有的 config 信息：

```
config show all
```

显示 replication 的相关 config 信息：

```
config show replication
```

```
{
  "max-replicas": 3,
  "isolation-level": "",
  "location-labels": "",
  "strictly-match-label": "false",
  "enable-placement-rules": "true"
}
```

显示目前集群版本，是目前集群 TiKV 节点的最低版本，并不对应 binary 的版本：

```
config show cluster-version
```

```
"5.2.2"
```

- max-snapshot-count 控制单个 store 最多同时接收或发送的 snapshot 数量，调度受制于这个配置来防止抢占正常业务的资源。当需要加快补副本或 balance 速度时可以调大这个值。

  设置最大 snapshot 为 64：

```
config set max-snapshot-count 64
```

- max-pending-peer-count 控制单个 store 的 pending peer 上限，调度受制于这个配置来防止在部分节点产生大量日志落后的 Region。需要加快补副本或 balance 速度可以适当调大这个值，设置为 0 则表示不限制。

  设置最大 pending peer 数量为 64：

```
config set max-pending-peer-count 64
```

- max-merge-region-size 控制 Region Merge 的 size 上限（单位是 MiB）。当 Region Size 大于指定值时 PD 不会将其与相邻的 Region 合并。设置为 0 表示不开启 Region Merge 功能。

  设置 Region Merge 的 size 上限为 16 MiB：

```
config set max-merge-region-size 16
```

- max-merge-region-keys 控制 Region Merge 的 keyCount 上限。当 Region KeyCount 大于指定值时 PD 不会将其与相邻的 Region 合并。

  设置 Region Merge 的 keyCount 上限为 50000：

```
config set max-merge-region-keys 50000
```

- split-merge-interval 控制对同一个 Region 做 split 和 merge 操作的间隔，即对于新 split 的 Region 一段时间内不会被 merge。

  设置 split 和 merge 的间隔为 1 天：

```
config set split-merge-interval 24h
```

- enable-one-way-merge 用于控制是否只允许和相邻的后一个 Region 进行合并。当设置为 false 时，PD 允许与相邻的前后 Region 进行合并。

  设置只允许和相邻的后一个 Region 合并：

```
config set enable-one-way-merge true
```

- enable-cross-table-merge 用于开启跨表 Region 的合并。当设置为 false 时，PD 不会合并不同表的 Region。该选项只在键类型为“table”时生效。

  设置允许跨表合并：

```
config set enable-cross-table-merge true
```

- key-type 用于指定集群的键编码类型。支持的类型有 ["table", "raw", "txn"]，默认值为“table”。

  - 如果集群中不存在 TiDB 实例，key-type 的值为“raw”或“txn”。此时，无论 enable-cross-table-merge 设置为何，PD 均可以跨表合并 Region。

  - 如果集群中存在 TiDB 实例，key-type 的值应当为 “table”。此时，enable-cross-table-merge 的设置决定了 PD 是否能跨表合并 Region。如果 key-type 的值为 “raw”，placement rules 不生效。

  启用跨表合并：

```
config set key-type raw
```

- region-score-formula-version 用于设置 Region 算分公式的版本，支持的值有 ["v1", "v2"]。v2 版本公式有助于减少上下线等场景下冗余的 balance Region 调度。

  开启 v2 版本 Region 算分公式：

```
config set region-score-formula-version v2
```

- patrol-region-interval 控制 replicaChecker 检查 Region 健康状态的运行频率，越短则运行越快，通常状况不需要调整。

  设置 replicaChecker 的运行频率为 10 毫秒：

```
config set patrol-region-interval 10ms
```

- max-store-down-time 为 PD 认为失联 store 无法恢复的时间，当超过指定的时间没有收到 store 的心跳后，PD 会在其他节点补充副本。

  设置 store 心跳丢失 30 分钟开始补副本：

```
config set max-store-down-time 30m
```

- max-store-preparing-time 控制 store 上线阶段的最长等待时间。在 store 的上线阶段，PD 可以查询该 store 的上线进度。当超过该配置项指定的时间后，PD 会认为该 store 已完成上线，无法再次查询这个 store 的上线进度，但是不影响 Region 向这个新上线 store 的迁移。通常用户无需修改该配置项。

  设置 store 上线阶段最多等待 4 小时：

```
config set max-store-preparing-time 4h
```

- 通过调整 leader-schedule-limit 可以控制同时进行 leader 调度的任务个数。这个值主要影响 leader balance 的速度，值越大调度得越快，设置为 0 则关闭调度。Leader 调度的开销较小，需要的时候可以适当调大。

  最多同时进行 4 个 leader 调度：

```
config set leader-schedule-limit 4
```

- 通过调整 region-schedule-limit 可以控制同时进行 Region 调度的任务个数。这个值可以避免创建过多的 Region balance operator。默认值为 2048，对所有大小的集群都足够。设置为 0 则关闭调度。Region 调度的速度通常受到 store-limit 的限制，但除非你熟悉该设置，否则不推荐自定义该参数。

  最多同时进行 2 个 Region 调度：

```
config set region-schedule-limit 2
```

- 通过调整 `replica-schedule-limit` 可以控制同时进行 replica 调度的任务个数。这个值主要影响节点挂掉或者下线的时候进行调度的速度，值越大调度得越快，设置为 0 则关闭调度。Replica 调度的开销较大，所以这个值不宜调得太大。注意：该参数通常保持为默认值。如需调整，需要根据实际情况反复尝试设置该值大小。

  最多同时进行 4 个 replica 调度：

```
config set replica-schedule-limit 4
```

- `merge-schedule-limit` 控制同时进行的 Region Merge 调度的任务，设置为 0 则关闭 Region Merge。Merge 调度的开销较大，所以这个值不宜调得过大。注意：该参数通常保持为默认值。如需调整，需要根据实际情况反复尝试设置该值大小。

  最多同时进行 16 个 merge 调度：

```
config set merge-schedule-limit 16
```

- `hot-region-schedule-limit` 控制同时进行的 Hot Region 调度的任务，设置为 0 则关闭调度。这个值不宜调得过大，否则可能对系统性能造成影响。注意：该参数通常保持为默认值。如需调整，需要根据实际情况反复尝试设置该值大小。

  最多同时进行 4 个 Hot Region 调度：

```
config set hot-region-schedule-limit 4
```

- `hot-region-cache-hits-threshold` 用于设置识别热点 Region 所需的分钟数，只有 Region 处于热点状态持续时间超过该分钟数后，才能参与热点调度。

- `tolerant-size-ratio` 控制 balance 缓冲区大小。当两个 store 的 leader 或 Region 的得分差距小于指定倍数的 Region size 时，PD 会认为此时 balance 达到均衡状态。

  设置缓冲区为约 20 倍平均 RegionSize：

```
config set tolerant-size-ratio 20
```

- `low-space-ratio` 用于设置 store 空间不足的阈值。当节点的空间占用比例超过指定值时，PD 会尽可能避免往对应节点迁移数据，同时主要针对剩余空间大小进行调度，避免对应节点磁盘空间被耗尽。

  设置空间不足阈值为 0.9：

```
config set low-space-ratio 0.9
```

- `high-space-ratio` 用于设置 store 空间充裕的阈值，此配置仅的在 `region-score-formula-version = v1` 时生效。当节点的空间占用比例小于指定值时，PD 调度时会忽略剩余空间这个指标，主要针对实际数据量进行均衡。

  设置空间充裕阈值为 0.5：

```
config set high-space-ratio 0.5
```

- `cluster-version` 集群的版本，用于控制某些 Feature 是否开启，处理兼容性问题。通常是集群正常运行的所有 TiKV 节点中的最低版本，需要回滚到更低的版本时才进行手动设置。

  设置 cluster version 为 1.0.8：

```
config set cluster-version 1.0.8
```

- leader-schedule-policy 用于选择 Leader 的调度策略，可以选择按照 size 或者 count 来进行调度。
- scheduler-max-waiting-operator 用于控制每个调度器同时存在的 operator 的个数。
- enable-remove-down-replica 用于开启自动删除 DownReplica 的特性。当设置为 false 时，PD 不会自动清理宕机状态的副本。
- enable-replace-offline-replica 用于开启迁移 OfflineReplica 的特性。当设置为 false 时，PD 不会迁移下线状态的副本。
- enable-make-up-replica 用于开启补充副本的特性。当设置为 false 时，PD 不会为副本数不足的 Region 补充副本。
- enable-remove-extra-replica 用于开启删除多余副本的特性。当设置为 false 时，PD 不会为副本数过多的 Region 删除多余副本。
- enable-location-replacement 用于开启隔离级别检查。当设置为 false 时，PD 不会通过调度来提升 Region 副本的隔离级别。
- enable-debug-metrics 用于开启 debug 的 metrics。当设置为 true 时，PD 会开启一些 metrics，比如 balance-tolerant-size 等。
- enable-placement-rules 用于开启 placement rules，在 v5.0 及以上的版本默认开启。
- store-limit-mode 用于控制 store 限速机制的模式。主要有两种模式：auto 和 manual。auto 模式下会根据 load 自动进行平衡调整（实验性功能）。
- PD 会对流量信息的末尾数字进行四舍五入处理，减少 Region 流量信息变化引起的统计信息更新。该配置项用于指定对 Region 流量信息的末尾进行四舍五入的位数。例如流量 100512 会归约到 101000。默认值为 3。该配置替换了 trace-region-flow。

  示例：将 flow-round-by-digit 的值设为 4：

```
config set flow-round-by-digit 4
```

14.6.2.4.3 config placement-rules [disable | enable | load | save | show | rule-group]

关于 config placement-rules 的具体用法，参考Placement Rules 使用文档。

14.6.2.4.4 health

用于显示集群健康信息。示例如下。

显示健康信息：

```
health
```

```
[
  {
    "name": "pd",
    "member_id": 13195394291058371180,
    "client_urls": [
      "http://127.0.0.1:2379"
      ......
    ],
    "health": true
  }
  ......
]
```

14.6.2.4.5 hot [read | write | store| history <start_time> <end_time> [<key> <value>]]

用于显示集群热点信息。示例如下。

显示读热点信息：

```
hot read
```

显示写热点信息：

```
hot write
```

显示所有 store 的读写信息：

```
hot store
```

显示历史读写热点信息:

```
hot history startTime endTime [ <name> <value> ]
```

例如查询时间 1629294000000 到 1631980800000 （毫秒）之间的历史热点 Region 信息:

```
hot history 1629294000000 1631980800000
```

```
{
  "history_hot_region": [
    {
      "update_time": 1630864801948,
      "region_id": 103,
      "peer_id": 1369002,
      "store_id": 3,
      "is_leader": true,
      "is_learner": false,
      "hot_region_type": "read",
```

```
      "hot_degree": 152,
      "flow_bytes": 0,
      "key_rate": 0,
      "query_rate": 305,
      "start_key": "7480000000000000FF5300000000000000F8",
      "end_key": "7480000000000000FF5600000000000000F8"
    },
    ...
  ]
}
```

对于参数的值为数组的请用 `x, y, ...` 的形式进行参数值的设置，所有支持的参数如下所示:

```
hot history 1629294000000 1631980800000 hot_region_type read region_id 1,2,3 store_id 1,2,3
    ↪ peer_id 1,2,3 is_leader true is_learner true
```

```
{
  "history_hot_region": [
    {
      "update_time": 1630864801948,
      "region_id": 103,
      "peer_id": 1369002,
      "store_id": 3,
      "is_leader": true,
      "is_learner": false,
      "hot_region_type": "read",
      "hot_degree": 152,
      "flow_bytes": 0,
      "key_rate": 0,
      "query_rate": 305,
      "start_key": "7480000000000000FF5300000000000000F8",
      "end_key": "7480000000000000FF5600000000000000F8"
    },
    ...
  ]
}
```

14.6.2.4.6 label [store <name> <value>]

用于显示集群标签信息。示例如下。

显示所有 label：

```
label
```

显示所有包含 label 为 “zone”：“cn” 的 store：

```
label store zone cn
```

14.6.2.4.7 member [delete | leader_priority | leader [show | resign | transfer <member_name>]]

用于显示 PD 成员信息，删除指定成员，设置成员的 leader 优先级。示例如下。

显示所有成员的信息：

```
member
```

```
{
  "header": {......},
  "members": [......],
  "leader": {......},
  "etcd_leader": {......},
}
```

下线“pd2”：

```
member delete name pd2
```

```
Success!
```

使用 id 下线节点：

```
member delete id 1319539429105371180
```

```
Success!
```

显示 leader 的信息：

```
member leader show
```

```
{
  "name": "pd",
  "member_id": 13155432540099656863,
  "peer_urls": [......],
  "client_urls": [......]
}
```

将 leader 从当前成员移走：

```
member leader resign
```

```
......
```

将 leader 迁移至指定成员：

```
member leader transfer pd3
```

```
......
```

14.6.2.4.8 operator [check | show | add | remove]

用于显示和控制调度操作。

示例：

```
>> operator show                                    // 显示所有的 operators
>> operator show admin                              // 显示所有的 admin operators
>> operator show leader                             // 显示所有的 leader operators
>> operator show region                             // 显示所有的 Region operators
>> operator add add-peer 1 2                        // 在 store 2 上新增 Region 1 的一个副本
>> operator add add-learner 1 2                     // 在 store 2 上新增 Region 1 的一个
    ↪ learner 副本
>> operator add remove-peer 1 2                     // 移除 store 2 上的 Region 1 的一个副本
>> operator add transfer-leader 1 2                 // 把 Region 1 的 leader 调度到 store 2
>> operator add transfer-region 1 2 3 4             // 把 Region 1 调度到 store 2,3,4
>> operator add transfer-peer 1 2 3                 // 把 Region 1 在 store 2 上的副本调度到
    ↪ store 3
>> operator add merge-region 1 2                    // 将 Region 1 与 Region 2 合并
>> operator add split-region 1 --policy=approximate // 将 Region 1 对半拆分成两个 Region，
    ↪ 基于粗略估计值
>> operator add split-region 1 --policy=scan        // 将 Region 1 对半拆分成两个 Region，
    ↪ 基于精确扫描值
>> operator remove 1                                // 把 Region 1 的调度操作删掉
>> operator check 1                                 // 查看 Region 1 相关 operator 的状态
```

其中，Region 的分裂都是尽可能地从靠近中间的位置开始。对这个位置的选择支持两种策略，即 scan 和 approximate。它们之间的区别是，前者通过扫描这个 Region 的方式来确定中间的 key，而后者是通过查看 SST 文件中记录的统计信息，来得到近似的位置。一般来说，前者更加精确，而后者消耗更少的 I/O，可以更快地完成。

14.6.2.4.9 ping

用于显示ping PD 所需要花费的时间

示例：

```
ping
```

```
time: 43.12698ms
```

14.6.2.4.10 region <region_id> [--jq="<query string>"]

用于显示 Region 信息。使用 jq 格式化输出请参考jq 格式化 json 输出示例。示例如下。

显示所有 Region 信息：

```
region
```

```
{
  "count": 1,
  "regions": [......]
}
```

显示 Region id 为 2 的信息：

```
region 2
```

```
{
  "id": 2,
  "start_key": "7480000000000000FF1D00000000000000F8",
  "end_key": "7480000000000000FF1F00000000000000F8",
  "epoch": {
    "conf_ver": 1,
    "version": 15
  },
  "peers": [
    {
      "id": 40,
      "store_id": 3
    }
  ],
  "leader": {
    "id": 40,
    "store_id": 3
  },
  "written_bytes": 0,
  "read_bytes": 0,
  "written_keys": 0,
  "read_keys": 0,
  "approximate_size": 1,
  "approximate_keys": 0
}
```

14.6.2.4.11 region key [--format=raw|encode|hex] <key>

用于查询某个 key 位于哪一个 Region 上，支持 raw、encoding 和 hex 格式。使用 encoding 格式时，key 需要使用单引号。

Hex 格式（默认）示例：

```
region key 7480000000000000FF1300000000000000F8
{
  "region": {
    "id": 2,
    ......
  }
}
```

Raw 格式示例：

```
region key --format=raw abc
```

```
{
  "region": {
    "id": 2,
    ......
  }
}
```

Encoding 格式示例：

```
region key --format=encode 't\200\000\000\000\000\000\000\377\035_r\200\000\000\000\000\377\017U
    ↪ \320\000\000\000\000\000\372'
```

```
{
  "region": {
    "id": 2,
    ......
  }
}
```

14.6.2.4.12 region scan

用于获取所有 Region。

示例：

```
region scan
```

```
{
  "count": 20,
  "regions": [......],
}
```

14.6.2.4.13 region sibling <region_id>

用于查询某个 Region 相邻的 Region。

示例：

```
region sibling 2
```

```
{
  "count": 2,
  "regions": [......],
}
```

14.6.2.4.14 region keys [--format=raw|encode|hex] <start_key> <end_key> <limit>

用于查询某个 key 范围内的所有 Region。支持不带 endKey 的范围。`limit` 的默认值是 16，设为 -1 则表示无数量限制。示例如下：

显示从 a 开始的所有 Region 信息，数量上限为 16：

```
region keys --format=raw a
```

```
{
  "count": 16,
  "regions": [......],
}
```

显示 [a, z) 范围内的所有 Region 信息，数量上限为 16：

```
region keys --format=raw a z
```

```
{
  "count": 16,
  "regions": [......],
}
```

显示 [a, z) 范围内的所有 Region 信息，无数量上限：

```
region keys --format=raw a z -1
```

```
{
  "count": ...,
  "regions": [......],
}
```

显示从 a 开始的所有 Region 信息，数量上限为 20：

```
region keys --format=raw a "" 20
```

```
{
  "count": 20,
  "regions": [......],
}
```

14.6.2.4.15 region store <store_id>

用于查询某个 store 上面所有的 Region。

示例：

```
region store 2
```

```
{
  "count": 10,
  "regions": [......],
}
```

14.6.2.4.16 region topread [limit]

用于查询读流量最大的 Region。 limit 的默认值是 16。

示例：

```
region topread
```

```
{
  "count": 16,
  "regions": [......],
}
```

14.6.2.4.17 region topwrite [limit]

用于查询写流量最大的 Region。 limit 的默认值是 16。

示例：

```
region topwrite
```

```
{
  "count": 16,
  "regions": [......],
}
```

14.6.2.4.18 region topconfver [limit]

用于查询 conf version 最大的 Region。 limit 的默认值是 16。

示例：

```
region topconfver
```

```
{
  "count": 16,
  "regions": [......],
}
```

14.6.2.4.19 region topversion [limit]

用于查询 version 最大的 Region。 limit 的默认值是 16。

示例：

```
region topversion
```

```
{
  "count": 16,
  "regions": [......],
}
```

14.6.2.4.20 region topsize [limit]

用于查询 approximate size 最大的 Region。 limit 的默认值是 16。

示例：

```
region topsize
```

```
{
  "count": 16,
  "regions": [......],
}
```

14.6.2.4.21 region check [miss-peer | extra-peer | down-peer | pending-peer | offline-peer | empty-region | hist-size | hist-keys] [--jq="<query string>"]

用于查询处于异常状态的 Region，使用 jq 格式化输出请参考jq 格式化 JSON 输出示例。

各类型的意义如下：

- miss-peer：缺副本的 Region
- extra-peer：多副本的 Region

- down-peer：有副本状态为 Down 的 Region
- pending-peer：有副本状态为 Pending 的 Region

示例：

```
region check miss-peer
```

```
{
  "count": 2,
  "regions": [......],
}
```

14.6.2.4.22 scheduler [show | add | remove | pause | resume | config | describe]

用于显示和控制调度策略。

示例：

```
>> scheduler show                                    // 显示所有已经创建的 schedulers
>> scheduler add grant-leader-scheduler 1            // 把 store 1 上的所有 Region 的 leader
    ↪ 调度到 store 1
>> scheduler add evict-leader-scheduler 1            // 把 store 1 上的所有 Region 的 leader
    ↪ 从 store 1 调度出去
>> scheduler config evict-leader-scheduler           // v4.0.0 起，展示该调度器具体在哪些
    ↪ store 上
>> scheduler add shuffle-leader-scheduler            // 随机交换不同 store 上的 leader
>> scheduler add shuffle-region-scheduler            // 随机调度不同 store 上的 Region
>> scheduler add evict-slow-store-scheduler          // 当有且仅有一个 slow store 时将该
    ↪ store 上的所有 Region 的 leader 驱逐出去
>> scheduler remove grant-leader-scheduler-1         // 把对应的调度器删掉，`-1` 对应 store
    ↪ ID
>> scheduler pause balance-region-scheduler 10       // 暂停运行 balance-region 调度器 10 秒
>> scheduler pause all 10                            // 暂停运行所有的调度器 10 秒
>> scheduler resume balance-region-scheduler         // 继续运行 balance-region 调度器
>> scheduler resume all                              // 继续运行所有的调度器
>> scheduler config balance-hot-region-scheduler     // 显示 balance-hot-region 调度器的配置
>> scheduler describe balance-region-scheduler       // 显示 balance-region
    ↪ 的运行状态和相应的诊断信息
```

14.6.2.4.23 scheduler describe balance-region-scheduler

用于查看 balance-region-scheduler 的运行状态和相应的诊断信息。

从 TiDB v6.3.0 起，PD 为 balance-region-scheduler 和 balance-leader-scheduler 提供了运行状态和简要诊断信息的功能，其余 scheduler 和 checker 暂未支持。你可以通过 pd-ctl 修改enable-diagnostic 配置项开启该功能。

调度器运行状态有以下几种类型：

- disabled：表示当前调度器不可用或被移除。
- paused：表示当前调度器暂停工作。
- scheduling：表示当前调度器正在生成调度。
- pending：表示当前调度器无法产生调度。pending 状态的调度器，会返回一个概览信息，来帮助用户诊断。概览信息包含了 store 的一些状态信息，解释了它们为什么不能被选中进行调度。
- normal：表示当前调度器无需进行调度。

14.6.2.4.24 scheduler config balance-leader-scheduler

用于查看和控制 balance-leader-scheduler 策略。

从 TiDB v6.0.0 起，PD 为 balance-leader-scheduler 引入了 Batch 参数，用于控制 balance-leader 执行任务的速度。你可以通过 pd-ctl 修改 balance-leader batch 配置项设置该功能。

在 v6.0.0 前，PD 不带有该配置（即 balance-leader batch=1）。在 v6.0.0 或更高版本中，balance-leader batch 的默认值为 4。如果你想为该配置项设置大于 4 的值，你需要同时调大scheduler-max-waiting-operator（默认值 5）。同时调大两个配置项后，你才能体验预期的加速效果。

```
scheduler config balance-leader-scheduler set batch 3  // 将 balance-leader
    ↪ 调度器可以批量执行的算子大小设置为 3
```

14.6.2.4.25 scheduler config balance-hot-region-scheduler

用于查看和控制 balance-hot-region-scheduler 策略。

示例：

```
scheduler config balance-hot-region-scheduler  // 显示 balance-hot-region 调度器的所有配置
{
  "min-hot-byte-rate": 100,
  "min-hot-key-rate": 10,
  "min-hot-query-rate": 10,
  "max-zombie-rounds": 3,
  "max-peer-number": 1000,
  "byte-rate-rank-step-ratio": 0.05,
  "key-rate-rank-step-ratio": 0.05,
  "query-rate-rank-step-ratio": 0.05,
  "count-rank-step-ratio": 0.01,
  "great-dec-ratio": 0.95,
  "minor-dec-ratio": 0.99,
  "src-tolerance-ratio": 1.05,
  "dst-tolerance-ratio": 1.05,
  "read-priorities": [
    "query",
    "byte"
  ],
  "write-leader-priorities": [
    "key",
```

```
    "byte"
  ],
  "write-peer-priorities": [
    "byte",
    "key"
  ],
  "strict-picking-store": "true",
  "enable-for-tiflash": "true",
  "rank-formula-version": "v2"
}
```

- `min-hot-byte-rate` 指计数的最小字节数，通常为 100。

```
scheduler config balance-hot-region-scheduler set min-hot-byte-rate 100
```

- `min-hot-key-rate` 指计数的最小 key 数，通常为 10。

```
scheduler config balance-hot-region-scheduler set min-hot-key-rate 10
```

- `min-hot-query-rate` 指计数的最小 query 数，通常为 10。

```
scheduler config balance-hot-region-scheduler set min-hot-query-rate 10
```

- `max-zombie-rounds` 指一个 operator 可被纳入 pending influence 所允许的最大心跳次数。如果将它设置为更大的值，更多的 operator 可能会被纳入 pending influence。通常用户不需要修改这个值。pending influence 指的是在调度中产生的、但仍生效的影响。

```
scheduler config balance-hot-region-scheduler set max-zombie-rounds 3
```

- `max-peer-number` 指最多要被解决的 peer 数量。这个配置可避免调度器处理速度过慢。

```
scheduler config balance-hot-region-scheduler set max-peer-number 1000
```

- `byte-rate-rank-step-ratio`、`key-rate-rank-step-ratio`、`query-rate-rank-step-ratio` 和 `count-rank-step-ratio` 分别控制 byte、key、query 和 count 的 step ranks。rank-step-ratio 决定了计算 rank 时的 step 值。`great-dec-ratio` 和 `minor-dec-ratio` 控制 dec 的 rank。通常用户不需要修改这些配置项。

```
scheduler config balance-hot-region-scheduler set byte-rate-rank-step-ratio 0.05
```

- `src-tolerance-ratio` 和 `dst-tolerance-ratio` 是期望调度器的配置项。`tolerance-ratio` 的值越小，调度就越容易。当出现冗余调度时，你可以适当调大这个值。

```
scheduler config balance-hot-region-scheduler set src-tolerance-ratio 1.1
```

- `read-priorities`、`write-leader-priorities` 和 `write-peer-priorities` 用于控制调度器优先从哪些维度进行热点均衡，支持配置两个维度。

  - `read-priorities` 和 `write-leader-priorities` 用于控制调度器在处理 read 和 write-leader 类型的热点时优先均衡的维度，可选的维度有 `query`、`byte` 和 `key`。
  - `write-peer-priorities` 用于控制调度器在处理 write-peer 类型的热点时优先均衡的维度，支持配置 `byte` 和 `key` 维度。

> **注意：**
>
> 若集群的所有组件未全部升级到 v5.2 及以上版本，`query` 维度的配置不生效，部分组件升级完成后调度器仍默认优先从 `byte` 和 `key` 维度进行热点均衡，集群的所有组件全部升级完成后，也会继续保持这样的兼容配置，可通过 `pd-ctl` 查看实时配置。通常用户不需要修改这些配置项。

```
scheduler config balance-hot-region-scheduler set read-priorities query,byte
```

- `strict-picking-store` 是控制热点调度搜索空间的开关，通常为打开。该配置项仅影响 `rank-formula-version` 为 v1 时的行为。当打开时，热点调度的目标是保证所配置的两个维度的热点均衡。当关闭后，热点调度只保证处于第一优先级的维度的热点均衡表现更好，但可能会导致其他维度的热点不再那么均衡。通常用户不需要修改这个配置项。

```
scheduler config balance-hot-region-scheduler set strict-picking-store true
```

- `rank-formula-version` 适用于热点调度，其用来确定调度策略的算法版本，支持的值有 `["v1", "v2"]`。目前该配置的默认值为 v2。
  - v1 版本为 v6.3.0 之前的策略，主要关注调度是否降低了不同 Store 之间的负载差值，以及是否在另一维度引入副作用。
  - v2 版本是 v6.3.0 引入的实验特性算法，在 v6.4.0 正式发布，主要关注 Store 之间均衡度的提升率，同时降低了对副作用的关注度。对比 `strict-picking-store` 为 `true` 的 v1 算法，v2 版本更注重优先均衡第一维度。对比 `strict-picking-store` 为 `false` 的 v1 算法，v2 版本兼顾了第二维度的均衡。
  - `strict-picking-store` 为 `true` 的 v1 版本算法较为保守，只有当存在两个维度的负载都偏高的 Store 时才能产生调度。在特定场景下有可能因为维度冲突导致无法继续均衡，需要将 `strict-picking-store` 改为 `false` 才能在第一维度取得更好的均衡效果。v2 版本算法则可以在两个维度都取得更好的均衡效果，并减少无效调度。

```
scheduler config balance-hot-region-scheduler set rank-formula-version v2
```

- `enable-for-tiflash` 是控制热点调度是否对 TiFlash 生效的开关。通常为打开，关闭后将不会产生 TiFlash 实例之间的热点调度。

```
scheduler config balance-hot-region-scheduler set enable-for-tiflash true
```

14.6.2.4.26 service-gc-safepoint

用于查询当前的 GC safepoint 与 service GC safepoint，输出结果示例如下：

```
{
  "service_gc_safe_points": [
    {
      "service_id": "gc_worker",
      "expired_at": 9223372036854775807,
      "safe_point": 439923410637160448
    }
  ],
  "gc_safe_point": 0
}
```

14.6.2.4.27 store [delete | cancel-delete | label | weight | remove-tombstone | limit] <store_id> [--jq="<query string>"]

使用 jq 格式化输出请参考jq 格式化 json 输出示例。

查询 store

显示所有 store 信息：

```
store
```

```
{
  "count": 3,
  "stores": [...]
}
```

获取 id 为 1 的 store：

```
store 1
```

```
......
```

下线 store

下线 id 为 1 的 store：

```
store delete 1
```

执行 `store cancel-delete` 命令，你可以撤销已使用 `store delete` 下线并处于 `Offline` 状态的 store。撤销后，该 store 会从 `Offline` 状态变为 `Up` 状态。注意，`store cancel-delete` 命令无法使 `Tombstone` 状态的 store 变回 `Up` 状态。

撤销通过 `store delete` 下线 id 为 1 的 store：

```
store cancel-delete 1
```

删除所有 Tombstone 状态的 store：

```
store remove-tombstone
```

> 注意：
>
> 若下线过程中切换了 PD leader，需要使用 `store limit` 命令修改store 调度限速。

管理 store label

`store label` 命令用于管理 store label。

- 为 id 为 1 的 store 设置键为 `"zone"`、值为 `"cn"` 的 label：

```
store label 1 zone=cn
```

- 更新 id 为 1 的 store 的 label：

```
store label 1 zone=us
```

- 通过 `--rewrite` 选项重写 id 为 1 的 store 的所有 label，之前的 label 会被覆盖：

```
store label 1 region=us-est-1 disk=ssd --rewrite
```

- 删除 id 为 1 的 store 的键为 `"disk"` 的 label ：

```
store label 1 disk --delete
```

> 注意：
>
> - store 的 label 更新方法使用的是合并策略。如果修改了 TiKV 配置文件中的 store label，进程重启之后，PD 会将自身存储的 store label 与其进行合并更新，并持久化合并后的结果。
> - 如果希望使用 TiUP 统一管理 store label，你可以在集群重启前，使用 PD Control 的 `store label <id> --force` 命令将 PD 存储的 store label 清空。

设置 store weight

将 id 为 1 的 store 的 leader weight 设为 5，Region weight 设为 10：

```
store weight 1 5 10
```

设置 store 调度限速

通过 `store-limit`，你可以设置 store 的调度速度。关于 `store limit` 的原理和使用方法，请参考`store limit`。

```
>> store limit                          // 显示所有 store 添加和删除 peer 的速度上限
>> store limit add-peer                 // 显示所有 store 添加 peer 的速度上限
>> store limit remove-peer              // 显示所有 store 删除 peer 的速度上限
>> store limit all 5                    // 设置所有 store 添加和删除 peer 的速度上限为每分钟 5 个
>> store limit 1 5                      // 设置 store 1 添加和删除 peer 的速度上限为每分钟 5 个
>> store limit all 5 add-peer           // 设置所有 store 添加 peer 的速度上限为每分钟 5 个
>> store limit 1 5 add-peer             // 设置 store 1 添加 peer 的速度上限为每分钟 5 个
>> store limit 1 5 remove-peer          // 设置 store 1 删除 peer 的速度上限为每分钟 5 个
>> store limit all 5 remove-peer        // 设置所有 store 删除 peer 的速度上限为每分钟 5 个
```

注意：

使用 pd-ctl 可以查看 TiKV 节点的状态信息，即 Up，Disconnect，Offline，Down，或 Tombstone。如需查看各个状态之间的关系，请参考TiKV Store 状态之间的关系。

14.6.2.4.28 log [fatal | error | warn | info | debug]

用于设置 PD leader 的日志级别。

```
log warn
```

14.6.2.4.29 tso

用于解析 TSO 到物理时间和逻辑时间。示例如下。

解析 TSO：

```
tso 395181938313123110
```

```
system:  2017-10-09 05:50:59 +0800 CST
logic:  120102
```

14.6.2.4.30 unsafe remove-failed-stores [store-ids | show]

警告：

- 此功能为有损恢复，无法保证数据和数据索引完整性。
- 建议在 TiDB 团队支持下进行相关操作，操作不当可能导致集群难以恢复。

用于在多数副本永久损坏造成数据不可用时进行有损恢复。示例如下。详见Online Unsafe Recovery。

执行 Online Unsafe Recovery，移除永久损坏的节点 (Store):

```
unsafe remove-failed-stores 101,102,103
```

```
Success!
```

显示正在运行的 Online Unsafe Recovery 的当前状态或历史状态。

```
unsafe remove-failed-stores show
```

```
[
  "Collecting cluster info from all alive stores, 10/12.",
  "Stores that have reports to PD: 1, 2, 3, ...",
  "Stores that have not reported to PD: 11, 12",
]
```

14.6.2.5 jq 格式化 JSON 输出示例

14.6.2.5.1 简化 store 的输出

```
store --jq=".stores[].store | { id, address, state_name}"
```

```
{"id":1,"address":"127.0.0.1:20161","state_name":"Up"}
{"id":30,"address":"127.0.0.1:20162","state_name":"Up"}
...
```

14.6.2.5.2 查询节点剩余空间

```
store --jq=".stores[] | {id: .store.id, available: .status.available}"
```

```
{"id":1,"available":"10 GiB"}
{"id":30,"available":"10 GiB"}
...
```

14.6.2.5.3 查询状态不为 Up 的所有节点

```
store --jq='.stores[].store | select(.state_name!="Up") | { id, address, state_name}'
```

```
{"id":1,"address":"127.0.0.1:20161""state_name":"Offline"}
{"id":5,"address":"127.0.0.1:20162""state_name":"Offline"}
...
```

14.6.2.5.4 查询所有的 TiFlash 节点

```
store --jq='.stores[].store | select(.labels | length>0 and contains([{"key":"engine","value":"
    ↪ tiflash"}])) | { id, address, state_name}'
```

```
{"id":1,"address":"127.0.0.1:20161""state_name":"Up"}
{"id":5,"address":"127.0.0.1:20162""state_name":"Up"}
...
```

14.6.2.5.5 查询 Region 副本的分布情况

```
region --jq=".regions[] | {id: .id, peer_stores: [.peers[].store_id]}"
```

```
{"id":2,"peer_stores":[1,30,31]}
{"id":4,"peer_stores":[1,31,34]}
...
```

14.6.2.5.6 根据副本数过滤 Region

例如副本数不为 3 的所有 Region：

```
region --jq=".regions[] | {id: .id, peer_stores: [.peers[].store_id] | select(length != 3)}"
```

```
{"id":12,"peer_stores":[30,32]}
{"id":2,"peer_stores":[1,30,31,32]}
```

14.6.2.5.7 根据副本 store ID 过滤 Region

例如在 store30 上有副本的所有 Region：

```
region --jq=".regions[] | {id: .id, peer_stores: [.peers[].store_id] | select(any(.==30))}"
```

```
{"id":6,"peer_stores":[1,30,31]}
{"id":22,"peer_stores":[1,30,32]}
...
```

还可以像这样找出在 store30 或 store31 上有副本的所有 Region：

```
region --jq=".regions[] | {id: .id, peer_stores: [.peers[].store_id] | select(any(.==(30,31)))}"
```

```
{"id":16,"peer_stores":[1,30,34]}
{"id":28,"peer_stores":[1,30,32]}
{"id":12,"peer_stores":[30,32]}
...
```

14.6.2.5.8 恢复数据时寻找相关 Region

例如当 [store1, store30, store31] 宕机时不可用时，我们可以通过查找所有 Down 副本数量大于正常副本数量的所有 Region：

```
region --jq=".regions[] | {id: .id, peer_stores: [.peers[].store_id] | select(length as $total |
    ↪ map(if .==(1,30,31) then . else empty end) | length>=$total-length) }"
```

```
{"id":2,"peer_stores":[1,30,31,32]}
{"id":12,"peer_stores":[30,32]}
{"id":14,"peer_stores":[1,30,32]}
...
```

或者在 [store1, store30, store31] 无法启动时，找出 store1 上可以安全手动移除数据的 Region。我们可以这样过滤出所有在 store1 上有副本并且没有其他 DownPeer 的 Region：

```
region --jq=".regions[] | {id: .id, peer_stores: [.peers[].store_id] | select(length>1 and any
    ↪ (.==1) and all(.!=(30,31)))}"
```

```
{"id":24,"peer_stores":[1,32,33]}
```

在 [store30, store31] 宕机时，找出能安全地通过创建 `remove-peer` Operator 进行处理的所有 Region，即有且仅有一个 DownPeer 的 Region：

```
region --jq=".regions[] | {id: .id, remove_peer: [.peers[].store_id] | select(length>1) | map(if
    ↪ .==(30,31) then . else empty end) | select(length==1)}"
```

```
{"id":12,"remove_peer":[30]}
{"id":4,"remove_peer":[31]}
{"id":22,"remove_peer":[30]}
...
```

14.6.3 TiDB Control 使用说明

TiDB Control 是 TiDB 的命令行工具，用于获取 TiDB 状态信息，多用于调试。本文介绍了 TiDB Control 的主要功能和各个功能的使用方法。

> **注意：**
>
> TiDB Control 主要用于诊断调试，不保证和 TiDB 未来引入的新特性完全兼容。因此不推荐客户在应用程序开发或工具开发中利用 TiDB Control 获取结果。

14.6.3.1 获取 TiDB Control

本节提供了两种方式获取 TiDB Control 工具。

> **注意：**
>
> 建议使用的 Control 工具版本与集群版本保持一致。

14.6.3.1.1 通过 TiUP 安装

在安装 TiUP 之后，可以使用 `tiup ctl:v<CLUSTER_VERSION> tidb` 命令来获取 TiDB Control 的二进制程序以及运行 TiDB Control。

14.6.3.1.2 从源代码编译安装

编译环境要求：Go 1.20 或以上版本

编译步骤：在 TiDB Control 项目根目录，使用 `make` 命令进行编译，生成 tidb-ctl。

编译文档：帮助文档在 doc 文件夹下，如丢失或需要更新，可通过 `make doc` 命令生成帮助文档。

14.6.3.2 使用介绍

`tidb-ctl` 的使用由命令（包括子命令）、选项和参数组成。命令即不带 - 或者 -- 的字符，选项即带有 - 或者 -- 的字符，参数即命令或选项字符后紧跟的传递给命令和选项的字符。

如：`tidb-ctl schema in mysql -n db`

- schema: 命令
- in: schema 的子命令
- mysql: in 的参数
- -n: 选项
- db: -n 的参数

目前，TiDB Control 包含以下子命令。

- `tidb-ctl base64decode` 用于 BASE64 解码
- `tidb-ctl decoder` 用于 KEY 解码
- `tidb-ctl etcd` 用于操作 etcd
- `tidb-ctl log` 用于格式化日志文件，将单行的堆栈信息展开
- `tidb-ctl mvcc` 用于获取 MVCC 信息
- `tidb-ctl region` 用于获取 Region 信息
- `tidb-ctl schema` 用于获取 Schema 信息
- `tidb-ctl table` 用于获取 Table 信息

14.6.3.2.1 获取帮助

`tidb-ctl -h/--help` 用于获取帮助信息。tidb-ctl 由多层命令组成，tidb-ctl 及其所有子命令都可以通过 `-h/--help` 来获取使用帮助。

以获取 Schema 信息为例：

通过 `tidb-ctl schema -h` 可以获取这个子命令的使用帮助。schema 有两个子命令 ——in 和 tid。in 用来通过数据库名获取数据库中所有表的表结构，tid 用来通过全数据库唯一的 table_id 获取表的表结构。

14.6.3.2.2 全局参数

`tidb-ctl` 有 4 个与连接相关的全局参数，分别为：

- `--host` TiDB 服务地址
- `--port` TiDB status 端口
- `--pdhost` PD 服务地址
- `--pdport` PD 服务端口
- `--ca` 连接使用的 TLS CA 文件路径
- `--ssl-key` 连接使用的 TLS 密钥文件路径
- `--ssl-cert` 连接使用的 TLS 证书文件路径

其中 `--pdhost` 和 `--pdport` 主要是用于 `etcd` 子命令，例如：`tidb-ctl etcd ddlinfo`。如不添加地址和端口将使用默认值，TiDB/PD 服务默认的地址是 127.0.0.1（服务地址只能使用 IP 地址），TiDB 服务端口默认的端口是 10080，PD 服务端口默认的端口是 2379 连接选项是全局选项，适用于以下所有命令。

14.6.3.2.3 schema 命令

in 子命令

in 子命令用来通过数据库名获取数据库中所有表的表结构。

`tidb-ctl schema in {数据库名}`

如：`tidb-ctl schema in mysql` 将得到以下结果

```
[
    {
        "id": 13,
        "name": {
            "O": "columns_priv",
            "L": "columns_priv"
        },
              ...
        "update_timestamp": 399494726837600268,
        "ShardRowIDBits": 0,
        "Partition": null
    }
]
```

结果将以 json 形式展示，内容较长，这里做了截断。

如希望指定表名，可以使用 `tidb-ctl schema in {数据库名} -n {表名}` 进行过滤。

如：`tidb-ctl schema in mysql -n db` 将得到 mysql 库中 db 表的表结构。结果如下：

```
{
    "id": 9,
    "name": {
        "O": "db",
        "L": "db"
    },
    ...
    "Partition": null
}
```

这里同样做了截断。

如使用的 TiDB 地址不为默认地址和端口，可以使用命令行参数 `--host`, `--port` 选项，如：`tidb-ctl --host 172.16.55.88 --port 8898 schema in mysql -n db`。

tid 子命令

tid 子命令用来通过表的 id 获取数据库中表的表结构。

通过使用 in 子命令查询到数据库中表的 id，之后可以通过 tid 子命令查看表的详细信息。

例如，查询到 `mysql.stat_meta` 表的 id 是 21，可以通过 `tidb-ctl schema tid -i 21` 查看表的详细信息。

```
{
 "id": 21,
 "name": {
  "O": "stats_meta",
  "L": "stats_meta"
 },
 "charset": "utf8mb4",
 "collate": "utf8mb4_bin",
  ...
}
```

同 in 子命令一样，如果使用的 TiDB 地址不是默认的地址和端口，需要通过 `--host` 和 `--port` 参数指定 TiDB 的地址和 status 端口。

14.6.3.2.4 base64decode 命令

base64decode 用来解码 base64 数据。

基本用法：

```
tidb-ctl base64decode [base64_data]
tidb-ctl base64decode [db_name.table_name] [base64_data]
tidb-ctl base64decode [table_id] [base64_data]
```

1. 准备环境，执行以下 SQL

```
use test;
create table t (a int, b varchar(20),c datetime default current_timestamp , d timestamp
    ↪ default current_timestamp, unique index(a));
insert into t (a,b,c) values(1,"哈哈 hello",NULL);
alter table t add column e varchar(20);
```

2. 用 HTTP API 接口获取 MVCC 数据

```
curl "http://$IP:10080/mvcc/index/test/t/a/1?a=1"
```

```
{
 "info": {
  "writes": [
   {
    "start_ts": 407306449994645510,
    "commit_ts": 407306449994645513,
    "short_value": "AAAAAAAAAAE="    # unique index a 存的值是对应行的 handle id.
   }
  ]
 }
}%
```

```
curl "http://$IP:10080/mvcc/key/test/t/1"
```

```
{
 "info": {
  "writes": [
   {
    "start_ts": 407306588892692486,
    "commit_ts": 407306588892692489,
    "short_value": "CAIIAggEAhjlk4jlk4ggaGVsbG8IBgAICAmAgIDwjYuuORk="  # handle id 为 1
        ↪ 的行数据。
   }
  ]
 }
}%
```

3. 用 base64decode 解码 handle id (uint64).

```
tidb-ctl base64decode AAAAAAAAAAE=
```

```
hex: 0000000000000001
uint64: 1
```

4. 用 base64decode 解码行数据。

```
./tidb-ctl base64decode test.t CAIIAggEAhjlk4jlk4ggaGVsbG8IBgAICAmAgIDwjYuuORk=
```

```
a:      1
b:      哈哈 hello
c is NULL
d:      2019-03-28 05:35:30
e not found in data
```

如果 test.t 的 table id 是 60，你也可以使用下列命令获得同样结果：

```
./tidb-ctl base64decode 60 CAIIAggEAhjlk4jlk4ggaGVsbG8IBgAICAmAgIDwjYuuORk=
```

```
a:      1
b:      哈哈 hello
c is NULL
d:      2019-03-28 05:35:30
e not found in data
```

14.6.3.2.5 decoder 命令

- 以下示例解码 row key，index key 类似。

```
./tidb-ctl decoder "t\x00\x00\x00\x00\x00\x00\x00\x1c_r\x00\x00\x00\x00\x00\x00\x00\xfa"
```

```
format: table_row
table_id: -9223372036854775780
row_id: -9223372036854775558
```

- 以下示例解码 value

```
./tidb-ctl decoder AhZoZWxsbyB3b3JsZAiAEA==
```

```
format: index_value
index_value[0]: {type: bytes, value: hello world}
index_value[1]: {type: bigint, value: 1024}
```

14.6.3.2.6 etcd 命令

- tidb-ctl etcd ddlinfo 获取 DDL 信息。
- tidb-ctl etcd putkey KEY VALUE 添加 KEY VALUE 到 etcd（所有的 KEY 会添加到 /tidb/ddl/all_schema_versions
  ↪ / 之下）。

```
tidb-ctl etcd putkey "foo" "bar"
```

实际是添加 KEY 为 /tidb/ddl/all_schema_versions/foo，VALUE 为 bar 的键值对到 etcd 中。

- tidb-ctl etcd delkey 删除 etcd 中的 KEY，只有前缀以 /tidb/ddl/fg/owner/ 和 /tidb/ddl/all_schema_versions/ 开头才允许被删除。

```
tidb-ctl etcd delkey "/tidb/ddl/fg/owner/foo" &&
tidb-ctl etcd delkey "/tidb/ddl/all_schema_versions/bar"
```

14.6.3.2.7 log 命令

TiDB 错误日志的堆栈信息是一行的格式，可以使用 tidb-ctl log 将堆栈信息格式化成多行形式。

14.6.3.2.8 keyrange 命令

keyrange 子命令用于查询全局或表相关的关键 key range 信息，以十六进制形式输出。

- 使用 tidb-ctl keyrange 命令查看全局的关键 key range。

```
tidb-ctl keyrange
```

```
global ranges:
  meta: (6d, 6e)
  table: (74, 75)
```

- 添加 --encode 选项可以显示 encode 过的 key（与 TiKV 及 PD 中的格式相同）。

```
tidb-ctl keyrange --encode
```

```
global ranges:
  meta: (6d00000000000000f8, 6e00000000000000f8)
  table: (7400000000000000f8, 7500000000000000f8)
```

- 使用 tidb-ctl keyrange --database={db} --table={tbl} 命令查看全局和表相关的关键 key range。

```
tidb-ctl keyrange --database test --table ttt
```

```
global ranges:
  meta: (6d, 6e)
  table: (74, 75)
table ttt ranges: (NOTE: key range might be changed after DDL)
  table: (74800000000000002f, 748000000000000030)
  table indexes: (74800000000000002f5f69, 74800000000000002f5f72)
    index c2: (74800000000000002f5f698000000000000001, 748000000000000002
        ↪ f5f698000000000000002)
    index c3: (74800000000000002f5f698000000000000002, 748000000000000002
        ↪ f5f698000000000000003)
    index c4: (74800000000000002f5f698000000000000003, 748000000000000002
        ↪ f5f698000000000000004)
  table rows: (74800000000000002f5f72, 748000000000000030)
```

14.6.4 PD Recover 使用文档

PD Recover 是对 PD 进行灾难性恢复的工具，用于恢复无法正常启动或服务的 PD 集群。

14.6.4.1 安装 PD Recover

要使用 PD Recover，你可以从源代码编译，也可以直接下载 TiDB 工具包。

14.6.4.1.1 从源代码编译

- Go：PD Recover 使用了 Go 模块，请安装 Go 1.20 或以上版本。
- 在 PD 根目录下，运行 `make pd-recover` 命令来编译源代码并生成 `bin/pd-recover`。

> 注意：
>
> 一般来说，用户不需要编译源代码，因为发布的二进制文件或 Docker 中已包含 PD Recover 工具。开发者可以参考以上步骤来编译源代码。

14.6.4.1.2 下载 TiDB 工具包

PD Recover 的安装包位于 TiDB 离线工具包中。下载方式，请参考TiDB 工具下载。

下面介绍两种重建集群的方式：从存活的 PD 节点重建和完全重建。

14.6.4.2 方式一：从存活的 PD 节点重建集群

当 PD 集群的大多数节点发生灾难性故障时，集群将无法提供服务。当还有 PD 节点存活时，可以选择一个存活的 PD 节点，通过强制修改 Raft Group 的成员，使该节点重新恢复服务。具体操作步骤如下：

14.6.4.2.1 第 1 步：停止所有节点

停止集群中的 TiDB、 TiKV 和 TiFlash 服务进程，以防止在恢复过程中与 PD 参数交互，造成数据错乱或其他无法挽救的异常状况。

14.6.4.2.2 第 2 步：启动存活的 PD 节点

使用启动参数 `--force-new-cluster` 拉起该存活的 PD 节点，如：

```
./bin/pd-server --force-new-cluster --name=pd-127.0.0.10-2379 --client-urls=http://0.0.0.0:2379
    ↪ --advertise-client-urls=http://127.0.0.1:2379 --peer-urls=http://0.0.0.0:2380 --advertise
    ↪ -peer-urls=http://127.0.0.1:2380 --config=conf/pd.toml
```

14.6.4.2.3 第 3 步：使用 pd-recover 修复元数据

该方法是利用少数派 PD 节点恢复服务，但由于该节点可能存在数据落后的情况，因此对于 alloc_id 和 tso 等数据，一旦发生回退，可能导致集群数据错乱或不可用。为确保该节点能提供正确的分配 ID 和 TSO 等服务，需要使用 pd-recover 修改元数据。具体命令参考：

```
./bin/pd-recover --from-old-member --endpoints=http://127.0.0.1:2379 # 指定对应的 PD 地址
```

> 注意：
>
> 该步骤会自动将存储中的 alloc_id 增加一个安全值 100000000。这将导致后续集群中分配的 ID 偏大。
>
> 此外，pd-recover 不会修改 TSO。因此，在执行此步骤之前，请确保本地时间晚于故障发生时间，并且确认故障前 PD 组件之间已开启 NTP 时钟同步服务。如果未开启，则需要将本地时钟调整到一个未来的时间，以确保 TSO 不会回退。

14.6.4.2.4 第 4 步：重启这个 PD

当上一步出现 recovery is successful 的提示信息后，重启 PD。

14.6.4.2.5 第 5 步：扩容 PD 并启动集群

通过部署工具扩容 PD，并启动集群中的其他组件。至此服务恢复。

14.6.4.3 方式二：完全重建 PD 集群

该方式适用于所有 PD 的数据都丢失，但 TiDB、TiKV 和 TiFlash 等其他组件数据都还存在的情况。

14.6.4.3.1 第 1 步：获取 Cluster ID

一般在 PD、TiKV 或 TiDB 的日志中都可以获取 Cluster ID。你可以直接在服务器上查看日志以获取 Cluster ID。

从 PD 日志获取 Cluster ID（推荐）

使用以下命令，从 PD 日志中获取 Cluster ID：

```
cat {{/path/to}}/pd.log | grep "init cluster id"
```

```
[2019/10/14 10:35:38.880 +00:00] [INFO] [server.go:212] ["init cluster id"] [cluster-id
    ↪ =6747551640615446306]
...
```

或者也可以从 TiDB 或 TiKV 的日志中获取。

从 TiDB 日志获取 Cluster ID

使用以下命令，从 TiDB 日志中获取 Cluster ID：

```
cat {{/path/to}}/tidb.log | grep "init cluster id"
```

```
2019/10/14 19:23:04.688 client.go:161: [info] [pd] init cluster id 6747551640615446306
...
```

从 TiKV 日志获取 Cluster ID

使用以下命令，从 TiKV 日志中获取 Cluster ID：

```
cat {{/path/to}}/tikv.log | grep "connect to PD cluster"
```

```
[2019/10/14 07:06:35.278 +00:00] [INFO] [tikv-server.rs:464] ["connect to PD cluster
    ↪ 6747551640615446306"]
...
```

14.6.4.3.2 第 2 步：获取已分配 ID

在指定已分配 ID 时，需指定一个比当前最大的已分配 ID 更大的值。可以从监控中获取已分配 ID，也可以直接在服务器上查看日志。

从监控中获取已分配 ID（推荐）

要从监控中获取已分配的 ID，需要确保你所查看的监控指标是上一任 PD Leader 的指标。可从 PD Dashboard 中 Current ID allocation 面板获取最大的已分配 ID。

从 PD 日志获取已分配 ID

要从 PD 日志中获取分配的 ID，需要确保你所查看的日志是上一任 PD Leader 的日志。运行以下命令获取最大的已分配 ID：

```
cat {{/path/to}}/pd*.log | grep "idAllocator allocates a new id" |  awk -F'=' '{print $2}' | awk
    ↪ -F']' '{print $1}' | sort -r -n | head -n 1
```

```
4000
...
```

你也可以在所有 PD server 中运行上述命令，找到最大的值。

14.6.4.3.3 第 3 步：部署一套新的 PD 集群

部署新的 PD 集群之前，需要停止当前的 PD 集群，然后删除旧的数据目录（或者用 `--data-dir` 指定新的数据目录）。

14.6.4.3.4 第 4 步：使用 pd-recover

只需在一个 PD 节点上执行 `pd-recover` 即可。

```
./pd-recover -endpoints http://10.0.1.13:2379 -cluster-id 6747551640615446306 -alloc-id 10000
```

14.6.4.3.5 第 5 步：重启整个集群

当出现 `recovery is successful` 的提示信息时，重启整个集群。

14.6.4.4 常见问题

14.6.4.4.1 获取 Cluster ID 时发现有多个 Cluster ID

新建 PD 集群时，会生成新的 Cluster ID。可以通过日志判断旧集群的 Cluster ID。

14.6.4.4.2 执行 pd-recover 时返回错误 `dial tcp 10.0.1.13:2379: connect: connection refused`

执行 pd-recover 时需要 PD 提供服务，请先部署并启动 PD 集群。

14.7 命令行参数

14.7.1 TiDB 配置参数

在启动 TiDB 时，你可以使用命令行参数或环境变量来配置 TiDB。

要快速了解 TiDB 的参数体系与参数作用域，建议先观看下面的培训视频（时长 17 分钟）。

本文将详细介绍 TiDB 的命令行启动参数。TiDB 的默认端口为 4000（客户端请求）与 10080（状态报告）。

14.7.1.1 `--advertise-address`

- 登录 TiDB 的 IP 地址
- 默认：“”
- 必须确保用户和集群中的其他机器都能够访问到该 IP 地址

14.7.1.2 `--config`

- 配置文件
- 默认：“”
- 如果你指定了配置文件，TiDB 会首先读取配置文件的配置。如果对应的配置在命令行参数里面也存在，TiDB 就会使用命令行参数的配置来覆盖配置文件中的配置。详细的配置项请参阅TiDB 配置文件描述。

14.7.1.3 `--config-check`

- 检查配置文件的有效性并退出
- 默认：false

14.7.1.4 `--config-strict`

- 增强配置文件的有效性
- 默认：false

14.7.1.5 --cors

- 用于设置 TiDB HTTP 状态服务的 Access-Control-Allow-Origin
- 默认：“”

14.7.1.6 --enable-binlog

- 开启或关闭 TiDB 中 binlog 的生成
- 默认：false

14.7.1.7 --host

- TiDB 服务监听的 host
- 默认：“0.0.0.0”
- 0.0.0.0 默认会监听所有的网卡地址。如果有多块网卡，可以指定对外提供服务的网卡，如 192.168.100.113

14.7.1.8 --initialize-insecure

- 在不安全模式下启动 tidb-server
- 默认：true

14.7.1.9 --initialize-secure

- 在安全模式下启动 tidb-server
- 默认：false

14.7.1.10 --initialize-sql-file

- 用于指定 TiDB 集群初次启动时执行的 SQL 脚本。参考配置项 initialize-sql-file
- 默认：“”

14.7.1.11 -L

- Log 级别
- 默认：“info”
- 可选：“debug”，“info”，“warn”，“error”，“fatal”

14.7.1.12 --lease

- Schema lease 的持续时间。除非你知道更改该值带来的后果，否则你的更改操作是危险的。
- 默认：45s

14.7.1.13 --log-file

- Log 文件
- 默认：“”
- 如果未设置该参数，log 会默认输出到“stderr”；如果设置了该参数，log 会输出到对应的文件中。

14.7.1.14 --log-slow-query

- 慢查询日志文件路径
- 默认：“”
- 如果未设置该参数，log 会默认输出到 --log-file 指定的文件中

14.7.1.15 --metrics-addr

- Prometheus Pushgateway 地址
- 默认：“”
- 如果该参数为空，TiDB 不会将统计信息推送给 Pushgateway。参数格式示例：--metrics-addr =192.168.100.115:9091

14.7.1.16 --metrics-interval

- 推送统计信息到 Prometheus Pushgateway 的时间间隔
- 默认：15s
- 设置为 0 表示不推送统计信息给 Pushgateway。示例：--metrics-interval=2 指每两秒推送到 Pushgateway

14.7.1.17 -P

- TiDB 服务监听端口
- 默认：“4000”
- TiDB 服务会使用该端口接受 MySQL 客户端发来的请求

14.7.1.18 --path

- 对于本地存储引擎“unistore”来说，path 指定的是实际的数据存放路径
- 当 --store = tikv 时，必须指定 path；当 --store = unistore 时，如果不指定 path，会使用默认值。
- 对于“TiKV”存储引擎来说，path 指定的是实际的 PD 地址。假如在 192.168.100.113:2379、192.168.100.114:2379 和 192.168.100.115:2379 上面部署了 PD，那么 path 为“192.168.100.113:2379, 192.168.100.114:2379, 192.168.100.115:2379”
- 默认：“/tmp/tidb”
- 可以通过 tidb-server --store=unistore --path="" 来启动一个纯内存引擎的 TiDB

14.7.1.19 --proxy-protocol-networks

- 允许使用 PROXY 协议连接 TiDB 的代理服务器地址列表。
- 默认：“”
- 通常情况下，通过反向代理使用 TiDB 时，TiDB 会将反向代理服务器的 IP 地址视为客户端 IP 地址。对于支持 PROXY 协议的反向代理（如 HAProxy），开启 PROXY 协议后能让反向代理透传客户端真实的 IP 地址给 TiDB。
- 配置该参数后，TiDB 将允许配置的源 IP 地址使用 PROXY 协议连接到 TiDB，且拒绝这些源 IP 地址使用非 PROXY 协议连接。其他地址可以使用非 PROXY 协议连接到 TiDB。若该参数为空，则任何源 IP 地址都不能使用 PROXY 协议连接到 TiDB。地址可以使用 IP 地址格式 (192.168.1.50) 或者 CIDR 格式 (192.168.1.0/24)，并可用 , 分隔多个地址，或用 * 代表所有 IP 地址。

> **警告：**
>
> 需谨慎使用 * 符号，因为它可能引入安全风险，允许来自任何 IP 的客户端自行汇报其 IP 地址。另外，当 `--proxy-protocol-fallbackable` 设置为 `true` 以外的值时，使用 * 可能会导致部分直接连接 TiDB 的内部组件无法使用，例如 TiDB Dashboard。

> **注意：**
>
> 如果使用 AWS 的 Network Load Balancer (NLB) 并开启 PROXY 协议，需要设置 NLB 的 `target group` 属性：将 `proxy_protocol_v2.client_to_server.header_place` 设为 `on_first_ack`。同时向 AWS 的 Support 提工单开通此功能的支持。注意，AWS NLB 在开启 PROXY 协议后，客户端将无法获取服务器端的握手报文，因此报文会一直阻塞到客户端超时。这是因为，NLB 默认只在客户端发送数据之后才会发送 PROXY 的报文，而在客户端发送数据包之前，服务器端发送的任何数据包都会在内网被丢弃。

14.7.1.20 --proxy-protocol-fallbackable

- 用于控制是否启用 PROXY 协议回退模式。如果设置为 `true`，TiDB 可以接受属于 `--proxy-protocol-networks` 的客户端使用非 PROXY 协议规范或者没有发送 PROXY 协议头的客户端连接。默认情况下，TiDB 仅接受属于 `--proxy-protocol-networks` 的客户端发送 PROXY 协议头的客户端连接。
- 默认：`false`

14.7.1.21 --proxy-protocol-header-timeout

- PROXY 协议请求头读取超时时间
- 默认：5
- 单位：秒

警告：

自 v6.3.0 起，该参数被废弃。因为自 v6.3.0 起，读取 PROXY 协议报头的操作会在第一次读取网络数据时进行，废弃该参数可避免影响首次读取网络数据时设置的超时时间。

注意：

不要将该参数配置为 0。除非特殊情况，一般使用默认值即可。

14.7.1.22 --report-status

- 用于打开或者关闭服务状态监听端口
- 默认：true
- 将参数值设置为 `true` 表明开启状态监听端口；设置为 `false` 表明关闭状态监听端口

14.7.1.23 --run-ddl

- tidb-server 是否运行 DDL 语句，集群内至少需要有一台 tidb-server 设置该参数
- 默认：true
- 值可以为 `true` 或者 `false`。设置为 `true` 表明自身会运行 DDL；设置为 `false` 表明自身不会运行 DDL

14.7.1.24 --socket string

- TiDB 服务使用 unix socket file 方式接受外部连接
- 默认：“”
- 例如可以使用 “/tmp/tidb.sock” 来打开 unix socket file

14.7.1.25 --status

- TiDB 服务状态监听端口
- 默认：“10080”
- 该端口用于展示 TiDB 内部数据，包括 prometheus 统计和 pprof
- Prometheus 统计可以通过 `http://host:status_port/metrics` 访问
- pprof 数据可以通过 `http://host:status_port/debug/pprof` 访问

14.7.1.26 --status-host

- TiDB 服务状态监听 host
- 默认：“0.0.0.0”

14.7.1.27 --store

- 用来指定 TiDB 底层使用的存储引擎
- 默认：“unistore”
- 可以选择“unistore”（本地存储引擎）或者“tikv”（分布式存储引擎）

14.7.1.28 --temp-dir

- TiDB 用于存放临时文件的目录
- 默认：“/tmp/tidb”

14.7.1.29 --token-limit

- TiDB 中同时允许运行的 Session 数量，用于流量控制
- 默认：1000
- 如果当前运行的连接多于该 token-limit，那么请求会阻塞，等待已经完成的操作释放 Token

14.7.1.30 -V

- 输出 TiDB 的版本
- 默认：“”

14.7.1.31 --plugin-dir

- plugin 存放目录
- 默认：“/data/deploy/plugin”

14.7.1.32 --plugin-load

- 需要加载的 plugin 名称，多个 plugin 以“,”逗号分隔
- 默认：“”

14.7.1.33 --affinity-cpus

- 设置 TiDB server CPU 亲和性，以“,”逗号分隔，例如“1,2,3”
- 默认：“”

14.7.1.34 --repair-mode

- 是否开启修复模式，仅用于数据修复场景
- 默认：false

14.7.1.35 --repair-list

- 修复模式下需要修复的表名
- 默认：“”

14.7.2 TiKV 配置参数

TiKV 的命令行参数支持一些可读性好的单位转换。

- 文件大小（以 bytes 为单位）：KB, MB, GB, TB, PB（也可以全小写）
- 时间（以毫秒为单位）：ms, s, m, h

14.7.2.1 -A, --addr

- TiKV 监听地址。
- 默认：“127.0.0.1:20160”
- 如果部署一个集群，--addr 必须指定当前主机的 IP 地址，例如“192.168.100.113:20160”，如果是运行在 docker 则需要指定为“0.0.0.0:20160”。

14.7.2.2 --advertise-addr

- TiKV 对外访问地址。
- 默认：${addr}
- 在某些情况下，比如 Docker 或者 NAT 网络环境，客户端并不能通过 --addr 的地址来访问到 TiKV。这时候，你可以设置 --advertise-addr 来让客户端访问 TiKV。
- 例如，docker 内部 IP 地址为 172.17.0.1，而宿主机的 IP 地址为 192.168.100.113 并且设置了端口映射 -p 20160:20160，那么可以设置为 --advertise-addr=“192.168.100.113:20160”，客户端可以通过 192.168.100.113:20160 来找到这个服务。

14.7.2.3 --status-addr

- TiKV 服务状态监听端口。
- 默认：“20180”
- Prometheus 统计可以通过 http://host:status_port/metrics 访问。
- Profile 数据可以通过 http://host:status_port/debug/pprof/profile 访问。

14.7.2.4 --advertise-status-addr

- TiKV 对外访问服务状态地址。
- 默认：使用 --status-addr
- 在某些情况下，例如 docker 或者 NAT 网络环境，客户端并不能通过 --status-addr 的地址来访问到 TiKV。此时，你可以设置 --advertise-status-addr 来让客户端访问 TiKV。
- 例如，Docker 内部 IP 地址为 172.17.0.1，而宿主机的 IP 地址为 192.168.100.113 并且设置了端口映射 -p 20180:20180，那么可以设置 --advertise-status-addr="192.168.100.113:20180"，客户端可以通过 192.168.100.113:20180 来找到这个服务。

14.7.2.5 -C, --config

- 配置文件。
- 默认：“”
- 如果你指定了配置文件，TiKV 会首先读取配置文件的配置。然后如果对应的配置在命令行参数里面也存在，TiKV 就会使用命令行参数的配置来覆盖配置文件里面的。

14.7.2.6 --capacity

- TiKV 存储数据的容量。
- 默认：0（无限）
- PD 需要使用这个值来对整个集群做 balance 操作。（提示：你可以使用 10GB 来替代 10737418240，从而简化参数的传递）。

14.7.2.7 --config-info <FORMAT>

- 按照指定的 FORMAT 输出各个配置项的取值信息并退出。
- FORMAT 可选值：json。
- 目前仅支持以 JSON 格式输出每个配置项的名字 (Name)、默认值 (DefaultValue) 和当前配置值 (ValueInFile)。当执行此命令时，若同时指定了 -C 或 --config 参数，则对应的配置文件包含的配置项会同时输出当前配置值和默认值，其他未指定的配置项仅输出默认值，示例如下：

```
json {   "Component": "TiKV Server", "Version": "6.2.0", "Parameters": [ { "Name": "log-level",
↪ "DefaultValue": "info", "ValueInFile": "warn" },   {      "Name": "log-file",  "DefaultValue
↪ ": "" },    ...   ]  }
```

14.7.2.8 --data-dir

- TiKV 数据存储路径。
- 默认：“/tmp/tikv/store”

14.7.2.9 -L

- Log 级别。
- 默认：“info”
- 可选值：“debug”，“info”，“warn”，“error”，“fatal”

14.7.2.10 --log-file

- Log 文件。
- 默认：“”
- 如果没设置这个参数，log 会默认输出到“stderr”，如果设置了，log 就会输出到对应的文件里面。

14.7.2.11 --pd

- PD 地址列表。
- 默认：“”
- TiKV 必须使用这个值连接 PD，才能正常工作。使用逗号来分隔多个 PD 地址，例如：192.168.100.113:2379, 192.168.100.114:2379, 192.168.100.115:2379。

14.7.3 TiFlash 命令行参数

本文介绍了 TiFlash 的命令行启动参数。

14.7.3.1 server --config-file

- 指定 TiFlash 的配置文件路径
- 默认：“”
- 必须指定配置文件，详细的配置项请参阅TiFlash 配置参数

14.7.3.2 dttool migrate

- 迁移 DTFile 的文件格式（用于测试和原地降级）。数据迁移的单位为单个 DTFile。如果想进行整表迁移，通常需要定位到所有形如 `<data dir>/t_<table id>/stable/dmf_<file id>` 的路径，逐一进行迁移。可以结合脚本来自动进行这一操作。
- 使用场景：
  - 当需要从开启了数据校验功能 (`storage.format_version` >= 3) 的 TiFlash v5.4 及以上版本降级回以前的版本时，可以使用此工具完成数据格式降级。
  - 当升级到 TiFlash v5.4 及以上，并希望对存量数据也加上数据校验功能以加固数据检验时，可以使用此工具完成数据格式升级。
  - 测试不同配置的 DTFile 空间占用和读取速度。
- 参数：
  - `--imitative`：当不使用 DTFile 的加密功能时，可以使用本选项避免使用配置文件和连接 PD。
  - `--version`：DTFile 的版本，可选值为 1 和 2，默认为 2。1 为传统格式，2 为新版 checksum 对应的 DTFile 格式。
  - `--algorithm`：检验哈希算法，可选值为 xxh3，city128，crc32，crc64，none，默认为 xxh3，仅在 version=2 时有用。
  - `--frame`：校验帧大小，默认为 1048576，仅在 version=2 时有用。
  - `--compression`：目标压缩算法，可选值为 LZ4（默认）、LZ4HC、zstd 和 none。
  - `--level`：目标压缩等级，不指定则根据压缩算法默认使用推荐的压缩级别。如果 `compression` 设置为 LZ4 或 `zstd`，则默认设置为 1；如果 `compression` 设置为 LZ4HC，则默认设置为 9。
  - `--config-file`：dttool migrate 的配置文件应当与 server 模式下的配置文件保持一致。见 `--imitative` 选项。
  - `--file-id`：对应 DTFile 的 ID，如 `dmf_123` 对应的 ID 是 123。

- `--workdir`：指向 `dmf_xxx` 的父级目录。
- `--dry`：空跑模式，只输出迁移过程。
- `--nokeep`：不保留原数据。不开启该选项时，会产生 `dmf_xxx.old` 文件。

> **警告：**
>
> 虽然 TiFlash 可以读取自定义压缩算法和压缩等级的 DTFile，但目前正式支持的只有默认压缩等级的 LZ4 算法。自定义压缩参数并未经过大量测试，仅作实验。

> **注意：**
>
> 为保证安全 DTTool 在迁移模式下会尝试对工作目录进行加锁，因此同一工作目录下同一时间只能有一个 DTTool 执行迁移工作。如果您在中途强制停止 DTTool，可能会因锁未释放导致后面在运行 DTTool 时工具拒绝进行迁移工作。如果您遇到这种情况，在保证安全的前提下，可以手动删除工作目录下的 LOCK 文件来释放锁。

14.7.3.3 `dttool bench`

- 提供 DTFile 的简单 IO 速度测试。
- 参数：
  - `--version`：DTFile 的版本，见dttool migrate 对应参数。
  - `--algorithm`：检验哈希算法，见dttool migrate 对应参数。
  - `--frame`：校验帧大小，见dttool migrate 对应参数。
  - `--column`：测试表宽度，默认为 100。
  - `--size`：测试表长度，默认为 1000。
  - `--field`：测试表字段长度上限，默认为 1024。
  - `--random`：随机数种子。如未提供，该值从系统熵池抽取。
  - `--encryption`：启用加密功能。
  - `--repeat`：性能测试采样次数，默认为 5。
  - `--workdir`：临时数据文件夹，应指向需要测试的文件系统下的路径，默认为 /tmp/test。

14.7.3.4 `dttool inspect`

- 检查 DTFile 的完整性。数据校验的单位为单个 DTFile。如果想进行整表校验，通常需要定位到所有形如 `<data dir>/t_<table id>/stable/dmf_<file id>` 的路径，逐一进行校验。可以结合脚本来自动进行这一操作。
- 使用场景：
  - 完成格式升降级后进行完整性检测。

- 将原有数据文件搬迁至新环境后进行完整性检测。

- 参数：

  - `--config-file`：dttool bench 的配置文件，见dttool migrate 对应参数。
  - `--check`：进行哈希校验。
  - `--file-id`：对应 DTFile 的 ID，见dttool migrate 对应参数。
  - `--imitative`：模拟数据库上下文，见dttool migrate 对应参数。
  - `--workdir`：数据文件夹，见dttool migrate 对应参数。

14.7.4 PD 配置参数

PD 可以通过命令行参数或环境变量配置。

14.7.4.1 `--advertise-client-urls`

- 用于外部访问 PD 的 URL 列表。
- 默认：`${client-urls}`
- 在某些情况下，例如 Docker 或者 NAT 网络环境，客户端并不能通过 PD 自己监听的 client URLs 来访问到 PD，这时候，你就可以设置 advertise URLs 来让客户端访问。
- 例如，Docker 内部 IP 地址为 `172.17.0.1`，而宿主机的 IP 地址为 `192.168.100.113` 并且设置了端口映射 `-p 2379:2379`，那么可以设置为 `--advertise-client-urls="http://192.168.100.113:2379"`，客户端可以通过 `http://192.168.100.113:2379` 来找到这个服务。

14.7.4.2 `--advertise-peer-urls`

- 用于其他 PD 节点访问某个 PD 节点的 URL 列表。
- 默认：`${peer-urls}`
- 在某些情况下，例如 Docker 或者 NAT 网络环境，其他节点并不能通过 PD 自己监听的 peer URLs 来访问到 PD，这时候，你就可以设置 advertise URLs 来让其他节点访问。
- 例如，Docker 内部 IP 地址为 `172.17.0.1`，而宿主机的 IP 地址为 `192.168.100.113` 并且设置了端口映射 `-p 2380:2380`，那么可以设置为 `advertise-peer-urls="http://192.168.100.113:2380"`，其他 PD 节点可以通过 `http://192.168.100.113:2380` 来找到这个服务。

14.7.4.3 `--client-urls`

- PD 监听的客户端 URL 列表。
- 默认：`http://127.0.0.1:2379`
- 如果部署一个集群，`--client-urls` 必须指定当前主机的 IP 地址，例如 `http://192.168.100.113:2379"`，如果是运行在 Docker 则需要指定为 `http://0.0.0.0:2379`。

14.7.4.4 --peer-urls

- PD 节点监听其他 PD 节点的 URL 列表。
- 默认：http://127.0.0.1:2380
- 如果部署一个集群，--peer-urls 必须指定当前主机的 IP 地址，例如 http://192.168.100.113:2380，如果是运行在 Docker 则需要指定为 http://0.0.0.0:2380。

14.7.4.5 --config

- 配置文件。
- 默认：“”
- 如果你指定了配置文件，PD 会首先读取配置文件的配置。然后如果对应的配置在命令行参数里面也存在，PD 就会使用命令行参数的配置来覆盖配置文件里面的。

14.7.4.6 --data-dir

- PD 存储数据路径。
- 默认：default.${name}

14.7.4.7 --initial-cluster

- 初始化 PD 集群配置。
- 默认："{name}=http://{advertise-peer-url}"
- 例如，如果 name 是“pd”，并且 advertise-peer-urls 是 http://192.168.100.113:2380，那么 initial-cluster 就是 pd=http://192.168.100.113:2380。
- 如果你需要启动三台 PD，那么 initial-cluster 可能就是 pd1=http://192.168.100.113:2380, pd2=http://192.168.100.114:2380, pd3=192.168.100.115:2380。

14.7.4.8 --join

- 动态加入 PD 集群。
- 默认：“”
- 如果你想动态将一台 PD 加入集群，你可以使用 --join="${advertise-client-urls}"，advertise-client-url 是当前集群里面任意 PD 的 advertise-client-url，你也可以使用多个 PD 的，需要用逗号分隔。

14.7.4.9 -L

- Log 级别。
- 默认：“info”
- 可选：“debug”，“info”，“warn”，“error”，“fatal”

14.7.4.10 --log-file

- Log 文件。
- 默认：“”
- 如果没设置这个参数，log 会默认输出到“stderr”，如果设置了，log 就会输出到对应的文件里面。

14.7.4.11 --log-rotate

- 是否开启日志切割。
- 默认：true
- 当值为 true 时，按照 PD 配置文件中 [log.file] 信息执行。

14.7.4.12 --name

- 当前 PD 的名字。
- 默认：“pd”
- 如果你需要启动多个 PD，一定要给 PD 使用不同的名字

14.7.4.13 --cacert

- CA 文件路径，用于开启 TLS。
- 默认：“”

14.7.4.14 --cert

- 包含 X509 证书的 PEM 文件路径，用户开启 TLS。
- 默认：“”

14.7.4.15 --key

- 包含 X509 key 的 PEM 文件路径，用于开启 TLS。
- 默认：“”

14.7.4.16 --metrics-addr

- 指定 Prometheus Pushgateway 的地址。
- 默认：“”
- 如果留空，则不开启 Prometheus Push。

14.7.4.17 --force-new-cluster

- 强制使用当前节点创建新的集群。
- 默认：false
- 仅用于在 PD 丢失多数副本的情况下恢复服务，可能会产生部分数据丢失。

14.7.4.18 -V, --version

- 输出版本信息并退出。

14.8 监控指标

14.8.1 Overview 面板重要监控指标详解

使用 TiUP 部署 TiDB 集群时，一键部署监控系统 (Prometheus & Grafana)，监控架构参见TiDB 监控框架概述。

目前 Grafana Dashboard 整体分为 PD、 TiDB、 TiKV、 Node_exporter、 Overview、 Performance_overview 等。

对于日常运维，我们单独挑选出重要的 Metrics 放在 Overview 页面，方便日常运维人员观察集群组件 (PD, TiDB, TiKV) 使用状态以及集群使用状态。

以下为 Overview Dashboard 监控说明：

14.8.1.1 Services Port Status

- Services Up：各服务在线节点数量

14.8.1.2 PD

- PD role：当前 PD 的角色
- Storage capacity：TiDB 集群总可用数据库空间大小
- Current storage size：TiDB 集群目前已用数据库空间大小，TiKV 多副本的空间占用也会包含在内
- Normal stores：处于正常状态的节点数目
- Abnormal stores：处于异常状态的节点数目，正常情况应当为 0
- Number of Regions：当前集群的 Region 总量，请注意 Region 数量与副本数无关
- 99% completed_cmds_duration_seconds：单位时间内，99% 的 pd-server 请求执行时间小于监控曲线的值，一般 <= 5ms
- Handle_requests_duration_seconds：PD 发送请求的网络耗时
- Region health：每个 Region 的状态，通常情况下，pending 的 peer 应该少于 100，miss 的 peer 不能一直大于 0
- Hot write Region’s leader distribution：每个 TiKV 实例上是写入热点的 leader 的数量
- Hot read Region’s leader distribution：每个 TiKV 实例上是读取热点的 leader 的数量
- Region heartbeat report：TiKV 向 PD 发送的心跳个数
- 99% Region heartbeat latency：99% 的情况下，心跳的延迟

14.8.1.3 TiDB

- Statement OPS：不同类型 SQL 语句每秒执行的数量。按 SELECT、 INSERT、 UPDATE 等来统计
- Duration：执行的时间
  - 客户端网络请求发送到 TiDB，到 TiDB 执行结束后返回给客户端的时间。一般情况下，客户端请求都是以 SQL 语句的形式发送，但也可以包含 COM_PING、COM_SLEEP、COM_STMT_FETCH、COM_SEND_LONG_DATA 之类的命令执行的时间

  - 由于 TiDB 支持 Multi-Query，因此，可以接受客户端一次性发送的多条 SQL 语句，如：`select 1; select 1; select 1;`。此时，统计的执行时间是所有 SQL 执行完之后的总时间

- CPS By Instance：每个 TiDB 实例上的命令统计。按照命令和执行结果成功或失败来统计
- Failed Query OPM：每个 TiDB 实例上，每秒钟执行 SQL 语句发生错误按照错误类型的统计（例如语法错误、主键冲突等）。包含了错误所属的模块和错误码
- Connection count：每个 TiDB 的连接数
- Memory Usage：每个 TiDB 实例的内存使用统计，分为进程占用内存和 Golang 在堆上申请的内存
- Transaction OPS：每秒事务执行数量统计
- Transaction Duration：事务执行的时间
- KV Cmd OPS：KV 命令执行数量统计
- KV Cmd Duration 99：KV 命令执行的时间
- PD TSO OPS：TiDB 每秒从 PD 获取 TSO 的数量
- PD TSO Wait Duration：TiDB 等待从 PD 获取 TS 的时间
- TiClient Region Error OPS：TiKV 返回 Region 相关错误信息的数量
- Lock Resolve OPS：TiDB 清理锁操作的数量。当 TiDB 的读写请求遇到锁时，会尝试进行锁清理
- Load Schema Duration：TiDB 从 TiKV 获取 Schema 的时间
- KV Backoff OPS：TiKV 返回错误信息的数量

14.8.1.4 TiKV

- leader：各个 TiKV 节点上 Leader 的数量分布
- region：各个 TiKV 节点上 Region 的数量分布
- CPU：各个 TiKV 节点的 CPU 使用率
- Memory：各个 TiKV 节点的内存使用量
- store size：每个 TiKV 实例的使用的存储空间的大小
- cf size：每个列族的大小
- channel full：每个 TiKV 实例上 channel full 错误的数量，正常情况下应当为 0
- server report failures：每个 TiKV 实例上报错的消息个数，正常情况下应当为 0
- scheduler pending commands：每个 TiKV 实例上 pending 命令的个数
- coprocessor executor count：TiKV 每秒收到的 coprocessor 操作数量，按照 coprocessor 类型统计
- coprocessor request duration：处理 coprocessor 读请求所花费的时间
- raft store CPU：raftstore 线程的 CPU 使用率，线程数量默认为 2（通过 `raftstore.store-pool-size` 配置）。如果单个线程使用率超过 80%，说明使用率很高
- Coprocessor CPU：coprocessor 线程的 CPU 使用率

14.8.1.5 System Info

- Vcores：CPU 核心数量
- Memory：内存总大小
- CPU Usage：CPU 使用率，最大为 100%
- Load [1m]：1 分钟的负载情况
- Memory Available：剩余内存大小
- Network Traffic：网卡流量统计
- TCP Retrans：TCP 重传数量统计
- IO Util：磁盘使用率，最高为 100%，一般到 80% - 90% 就需要考虑加节点

14.8.1.6 图例



图 324: overview

14.8.2 Performance Overview 面板重要监控指标详解

使用 TiUP 部署 TiDB 集群时，你可以一键部署监控系统 (Prometheus & Grafana)。监控架构参见TiDB 监控框架概述。

目前 Grafana Dashboard 整体分为 PD、TiDB、TiKV、Node_exporter、Overview、Performance_overview 等。

Performance Overview Dashboard 按总分结构对 TiDB、TiKV、PD 的性能指标进行编排组织，包含了以下三部分内容：

- 总的概览：数据库时间和 SQL 执行时间概览。通过颜色优化法，你可以快速识别数据库负载特征和性能瓶颈。
- 资源负载：关键指标和资源利用率，包含数据库 QPS、应用和数据库的连接信息和请求命令类型、数据库内部 TSO 和 KV 请求 OPS、TiDB 和 TiKV 的资源使用概况。
- 自上而下的延迟分解：Query 延迟和连接空闲时间对比、Query 延迟分解、execute 阶段 TSO 请求和 KV 请求的延迟、TiKV 内部写延迟的分解等。

借助 Performance Overview Dashboard，你可以高效地进行性能分析，确认用户响应时间的瓶颈是否在数据库中。如果数据库是整个系统的瓶颈，通过数据库时间概览和 SQL 延迟的分解，定位数据库内部的瓶颈点，并进行针对性的优化。详情请参考TiDB 性能分析和优化方法。

以下为 Performance Overview Dashboard 监控说明：

14.8.2.1 Performance Overview

14.8.2.1.1 Database Time by SQL Type

- database time: 每秒的总数据库时间
- sql_type: 每种 SQL 语句每秒消耗的数据库时间

14.8.2.1.2 Database Time by SQL Phase

- database time: 每秒的总数据库时间
- get token/parse/compile/execute: 4 个 SQL 处理阶段每秒消耗的数据库时间

execute 执行阶段为绿色，其他三个阶段偏红色系，如果非绿色的颜色占比明显，意味着在执行阶段之外数据库消耗了过多时间，需要进一步分析根源。

14.8.2.1.3 SQL Execute Time Overview

- execute time: execute 阶段每秒消耗的数据库时间
- tso_wait: execute 阶段每秒同步等待 TSO 的时间
- kv request type: execute 阶段每秒等待每种 KV 请求类型的时间，总的 KV request 等待时间可能超过 execute time，因为 KV request 是并发的。
- tiflash_mpp: execute 阶段每秒 TiFlash 请求处理时间。

绿色系标识代表常规的写 KV 请求（例如 Prewrite 和 Commit），蓝色系标识代表常规的读 KV 请求，紫色系标识代表 TiFlash MPP 请求，其他色系标识需要注意的问题。例如，悲观锁加锁请求为红色，TSO 等待为深褐色。如果非蓝色系或者非绿色系占比明显，意味着执行阶段存在异常的瓶颈。例如，当发生严重锁冲突时，红色的悲观锁时间会占比明显；当负载中 TSO 等待的消耗时间过长时，深褐色会占比明显。

14.8.2.1.4 QPS

QPS：按 SELECT、 INSERT、 UPDATE 等类型统计所有 TiDB 实例上每秒执行的 SQL 语句数量

14.8.2.1.5 CPS By Type

CPS By Type：按照类型统计所有 TiDB 实例每秒处理的命令数（Command Per Second）

14.8.2.1.6 Queries Using Plan Cache OPS

- avg-hit：所有 TiDB 实例每秒执行计划缓存的命中次数
- avg-miss：所有 TiDB 实例每秒执行计划缓存的未命中次数

`avg-hit + avg-miss` 等于 StmtExecute 每秒执行次数。

14.8.2.1.7 KV/TSO Request OPS

- kv request total: 所有 TiDB 实例每秒总的 KV 请求数量
- kv request by type: 按 `Get`、`Prewrite`、`Commit` 等类型统计在所有 TiDB 实例每秒的请求数据
- tso - cmd：在所有 TiDB 实例每秒 tso cmd 的请求数量
- tso - request：在所有 TiDB 实例每秒 tso request 的请求数量

通常 tso - cmd 除以 tso - request 等于平均请求的 batch 大小。

14.8.2.1.8 KV Request Time By Source

- kv request total time: 所有 TiDB 实例每秒总的 KV 和 TiFlash 请求处理时间。
- 每种 KV 请求和请求来源组成柱状堆叠图，`external` 标识正常业务的请求，`internal` 标识内部活动的请求（比如 DDL、auto analyze 等请求）。

14.8.2.1.9 TiDB CPU

- avg：所有 TiDB 实例平均 CPU 利用率
- delta：所有 TiDB 实例中最大 CPU 利用率减去所有 TiDB 实例中最小 CPU 利用率
- max：所有 TiDB 实例中最大 CPU 利用率

14.8.2.1.10 TiKV CPU/IO MBps

- CPU-Avg：所有 TiKV 实例平均 CPU 利用率
- CPU-Delta：所有 TiKV 实例中最大 CPU 利用率减去所有 TiKV 实例中最小 CPU 利用率
- CPU-MAX：所有 TiKV 实例中最大 CPU 利用率
- IO-Avg：所有 TiKV 实例平均 MBps
- IO-Delta：所有 TiKV 实例中最大 MBps 减去所有 TiKV 实例中最小 MBps
- IO-MAX：所有 TiKV 实例中最大 MBps

14.8.2.1.11 Duration

- Duration：执行时间解释
  - 从客户端网络请求发送到 TiDB，到 TiDB 执行结束后返回给客户端的时间。一般情况下，客户端请求都是以 SQL 语句的形式发送，但也可以包含 `COM_PING`、`COM_SLEEP`、`COM_STMT_FETCH`、`COM_SEND_LONG_DATA` 之类的命令执行时间。

  - 由于 TiDB 支持 Multi-Query，因此，客户端可以一次性发送多条 SQL 语句，如 `select 1; select 1; select 1;`。此时的执行时间是所有 SQL 语句执行完成的总时间。

- avg：所有请求命令的平均执行时间
- 99：所有请求命令的 P99 执行时间
- avg by type：按 SELECT、 INSERT、 UPDATE 类型统计所有 TiDB 实例上所有请求命令的平均执行时间

14.8.2.1.12 Connection Idle Duration

Connection Idle Duration 指空闲连接的持续时间。

- avg-in-txn：处于事务中，空闲连接的平均持续时间
- avg-not-in-txn：没有处于事务中，空闲连接的平均持续时间
- 99-in-txn：处于事务中，空闲连接的 P99 持续时间

14.8.2.1.13 Connection Count

- total：所有 TiDB 节点的总连接数
- active connections：所有 TiDB 节点的总活跃连接数
- tidb-{node-number}-peer：各个 TiDB 节点的连接数
- disconnection/s：集群每秒断开连接的数量
- 99-not-in-txn：没有处于事务中，空闲连接的 P99 持续时间

14.8.2.1.14 Parse Duration、 Compile Duration 和 Execute Duration

- Parse Duration：SQL 语句解析耗时统计
- Compile Duration：将解析后的 SQL AST 编译成执行计划的耗时
- Execution Duration：执行 SQL 语句执行计划耗时

这三个时间指标均包含均所有 TiDB 实例的平均值和 P99 值。

14.8.2.1.15 Avg TiDB KV Request Duration

按 `Get`、 `Prewrite`、 `Commit` 等类型统计在所有 TiDB 实例 KV 请求的平均执行时间。

14.8.2.1.16 Avg TiKV GRPC Duration

按 `get`、 `kv_prewrite`、 `kv_commit` 等类型统计所有 TiKV 实例对 gRPC 请求的平均执行时间。

14.8.2.1.17 PD TSO Wait/RPC Duration

- wait - avg：所有 TiDB 实例等待从 PD 返回 TSO 的平均时间
- rpc - avg：所有 TiDB 实例从向 PD 发送获取 TSO 的请求到接收到 TSO 的平均耗时
- wait - 99：所有 TiDB 实例等待从 PD 返回 TSO 的 P99 时间
- rpc - 99：所有 TiDB 实例从向 PD 发送获取 TSO 的请求到接收到 TSO 的 P99 耗时

14.8.2.1.18 Storage Async Write Duration、Store Duration 和 Apply Duration

- Storage Async Write Duration：异步写所花费的时间
- Store Duration：异步写 Store 步骤所花费的时间
- Apply Duration：异步写 Apply 步骤所花费的时间

这三个时间指标都包含所有 TiKV 实例的平均值和 P99 值

平均 Storage async write duration = 平均 Store Duration + 平均 Apply Duration

14.8.2.1.19 Append Log Duration、Commit Log Duration 和 Apply Log Duration

- Append Log Duration：Raft append 日志所花费的时间
- Commit Log Duration：Raft commit 日志所花费的时间
- Apply Log Duration：Raft apply 日志所花费的时间

这三个时间指标均包含所有 TiKV 实例的平均值和 P99 值。

14.8.2.1.20 图例



图 325: performance overview

14.8.2.2 TiFlash

- CPU：每个 TiFlash 实例 CPU 的使用率
- Memory：每个 TiFlash 实例内存的使用情况
- IO utilization：每个 TiFlash 实例的 IO 使用率

- MPP Query count：每个 TiFlash 实例每秒 MPP 查询数量
- Request QPS：所有 TiFlash 实例收到的 coprocessor 请求数量。
  - `batch`：batch 请求数量
  - `batch_cop`：batch 请求中的 coprocessor 请求数量
  - `cop`：直接通过 coprocessor 接口发送的 coprocessor 请求数量
  - `cop_dag`：所有 coprocessor 请求中 dag 请求数量
  - `super_batch`：开启 super batch 特性的请求数量
- Executor QPS：所有 TiFlash 实例收到的请求中，每种 dag 算子的数量，其中 `table_scan` 是扫表算子，`selection` 是过滤算子，`aggregation` 是聚合算子，`top_n` 是 TopN 算子，`limit` 是 limit 算子
- Request Duration Overview：每秒所有 TiFlash 实例所有请求类型总处理时间的堆叠图
- Request Duration：所有 TiFlash 实例每种 MPP 和 coprocessor 请求类型的总处理时间，此时间为接收到该 coprocessor 请求至请求应答完毕的时间，包含平均和 P99 处理延迟
- Request Handle Duration：所有 TiFlash 实例每种 MPP 和 coprocessor 请求的处理时间，此时间为该 coprocessor 请求从开始执行到结束的时间，包含平均和 P99 延迟
- Raft Wait Index Duration：所有 TiFlash 实例在进行 wait_index 消耗的时间，即拿到 read_index 请求后，等待本地的 Region index >= read_index 所花费的时间
- Raft Batch Read Index Duration：所有 TiFlash 实例在进行 read_index 消耗的时间，主要消耗在于和 Region leader 的交互和重试时间
- Write Throughput By Instance：每个实例写入数据的吞吐量，包括 apply Raft 数据日志以及 Raft 快照的写入吞吐量
- Write flow：所有 TiFlash 实例磁盘写操作的流量
- Read flow：所有 TiFlash 实例磁盘读操作的流量

14.8.2.3 CDC

- CPU usage：TiCDC 节点的 CPU 使用情况
- Memory usage：TiCDC 节点的内存使用情况
- Goroutine count：TiCDC 节点 Goroutine 的个数
- Changefeed checkpoint lag：同步任务上下游数据的进度差（以时间单位秒计算）
- Changefeed resolved ts lag：TiCDC 节点内部同步状态与上游的进度差（以时间单位秒计算）
- The status of changefeeds：changefeed 的状态
  - 0：Normal
  - 1：Error
  - 2：Failed
  - 3：Stopped

  - 4：Finished
  - -1：Unknown

- Puller output events/s：TiCDC 节点中 Puller 模块每秒输出到 Sorter 模块的数据变更行数
- Sorter output events/s：TiCDC 节点中 Sorter 模块每秒输出到 Mounter 模块的行数
- Mounter output events/s：TiCDC 节点中 Mounter 模块每秒输出到 Sink 模块的行数
- Table sink output events/s：TiCDC 节点中 Table Sorter 模块每秒输出到 Sink 模块的行数
- SinkV2 - Sink flush rows/s：TiCDC 节点中 Sink 模块每秒输出到下游的行数
- Transaction Sink Full Flush Duration：TiCDC 节点中 MySQL Sink 写下游事务的平均延迟和 p999 延迟
- MQ Worker Send Message Duration Percentile：下游为 Kafka 时 MQ worker 发送消息的延迟
- Kafka Outgoing Bytes：MQ Workload 写下游事务的流量

14.8.3 TiDB 重要监控指标详解

使用 TiUP 部署 TiDB 集群时，你可以一键部署监控系统 (Prometheus & Grafana)，参考监控架构TiDB 监控框架概述。

目前 Grafana Dashboard 整体分为 PD、TiDB、TiKV、Node_exporter、Overview、Performance_overview 等。TiDB 分为 TiDB 和 TiDB Summary 面板，两个面板的区别如下：

- TiDB 面板：提供尽可能全面的信息，供排查集群异常。
- TiDB Summary 面板：将 TiDB 面板中用户最为关心的部分抽取出来，并做了些许修改。主要用于提供数据库日常运行中用户关心的数据，如 QPS、TPS、响应延迟等，以便作为外部展示、汇报用的监控信息。

以下为 TiDB Dashboard 关键监控指标的说明：

14.8.3.1 关键指标说明

14.8.3.1.1 Query Summary

- Duration：执行时间
  - 客户端网络请求发送到 TiDB，到 TiDB 执行结束后返回给客户端的时间。一般情况下，客户端请求都是以 SQL 语句的形式发送，但也可以包含 COM_PING、COM_SLEEP、COM_STMT_FETCH、COM_SEND_LONG_DATA 之类的命令执行时间。
  - 由于 TiDB 支持 Multi-Query，因此，客户端可以一次性发送多条 SQL 语句，如 `select 1; select 1; select 1;`。此时的执行时间是所有 SQL 语句执行完之后的总时间。
- Command Per Second：TiDB 按照执行结果成功或失败来统计每秒处理的命令数。
- QPS：按 SELECT、INSERT、UPDATE 类型统计所有 TiDB 实例上每秒执行的 SQL 语句数量。
- CPS By Instance：按照命令和执行结果成功或失败来统计每个 TiDB 实例上的命令。
- Failed Query OPM：每个 TiDB 实例上，对每分钟执行 SQL 语句发生的错误按照错误类型进行统计（例如语法错误、主键冲突等）。包含了错误所属的模块和错误码。

- Slow query：慢查询的处理时间（整个慢查询耗时、Coprocessor 耗时、Coprocessor 调度等待时间），慢查询分为 internal 和 general SQL 语句。
- Connection Idle Duration：空闲连接的持续时间。
- 999/99/95/80 Duration：不同类型的 SQL 语句执行耗时（不同百分位）。

14.8.3.1.2 Query Detail

- Duration 80/95/99/999 By Instance：每个 TiDB 实例执行 SQL 语句的耗时（不同百分位）。
- Failed Query OPM Detail：每个 TiDB 实例上，对每分钟执行 SQL 语句发生的错误按照错误类型进行统计（例如语法错误、主键冲突等）。
- Internal SQL OPS：整个 TiDB 集群内部 SQL 语句执行的 QPS。内部 SQL 语句是指 TiDB 内部自动执行的 SQL 语句，一般由用户 SQL 语句来触发或者内部定时任务触发。

14.8.3.1.3 Server

- Uptime：每个 TiDB 实例的运行时间。
- Memory Usage：每个 TiDB 实例的内存使用，分为进程占用内存和 Golang 在堆上申请的内存。
- CPU Usage：每个 TiDB 实例的 CPU 使用。
- Connection Count：每个 TiDB 的连接数。
- Open FD Count：每个 TiDB 实例的打开的文件描述符数量。
- Disconnection Count：每个 TiDB 实例断开连接的数量。
- Event OPM：每个 TiDB 实例关键事件，例如 start，close，graceful-shutdown，kill，hang 等。
- Goroutine Count：每个 TiDB 实例的 Goroutine 数量。
- Prepare Statement Count：每个 TiDB 实例现存的 `Prepare` 语句数以及总数。
- Keep Alive OPM：每个 TiDB 实例每分钟刷新监控的次数，通常不需要关注。
- Panic And Critical Error：TiDB 中出现的 Panic、Critical Error 数量。
- Time Jump Back OPS：每个 TiDB 实例上每秒操作系统时间回跳的次数。
- Get Token Duration：每个连接获取 Token 的耗时。
- Skip Binlog Count：TiDB 写入 Binlog 失败的数量。
- Client Data Traffic：TiDB 和客户端的数据流量。

14.8.3.1.4 Transaction

- Transaction OPS：每秒事务的执行数量
- Duration：事务执行时间
- Transaction Statement Num：事务中的 SQL 语句数量
- Transaction Retry Num：事务重试次数
- Session Retry Error OPS：事务重试时每秒遇到的错误数量，分为重试失败和超过最大重试次数两种类型
- Commit Token Wait Duration：事务提交时的流控队列等待时间。当出现较长等待时，代表提交事务过大，正在限流。如果系统还有资源可以使用，可以通过增大系统变量 `tidb_committer_concurrency` 的值来加速提交
- KV Transaction OPS：每个 TiDB 内部每秒执行的事务数量
  - 一个用户的事务，在 TiDB 内部可能会触发多次事务执行，其中包含，内部元数据的读取，用户事务原子性地多次重试执行等

  - TiDB 内部的定时任务也会通过事务来操作数据库，这部分也包含在这个面板里

- KV Transaction Duration：每个 TiDB 内部执行事务的耗时
- Transaction Regions Num：事务操作的 Region 数量
- Transaction Write KV Num Rate and Sum：事务写入 KV 的速率总和
- Transaction Write KV Num：事务操作的 KV 数量
- Statement Lock Keys：单个语句的加锁个数
- Send HeartBeat Duration：事务发送心跳的时间间隔
- Transaction Write Size Bytes Rate and sum：事务写入字节数的速率总和
- Transaction Write Size Bytes：事务写入的数据大小
- Acquire Pessimistic Locks Duration：加锁所消耗的时间
- TTL Lifetime Reach Counter：事务的 TTL 寿命上限。TTL 上限默认值 1 小时，它的含义是从悲观事务第一次加锁，或者乐观事务的第一个 prewrite 开始，超过了 1 小时。可以通过修改 TiDB 配置文件中 `max-txn-ttl` 来改变 TTL 寿命上限
- Load Safepoint OPS：加载 Safepoint 的次数。Safepoint 作用是在事务读数据时，保证不读到 Safepoint 之前的数据，保证数据安全。因为，Safepoint 之前的数据有可能被 GC 清理掉
- Pessimistic Statement Retry OPS：悲观语句重试次数。当语句尝试加锁时，可能遇到写入冲突，此时，语句会重新获取新的 snapshot 并再次加锁
- Transaction Types Per Seconds：每秒采用两阶段提交 (2PC)、异步提交（Async Commit) 和一阶段提交 (1PC) 机制的事务数量，提供成功和失败两种数量

14.8.3.1.5 Executor

- Parse Duration：SQL 语句解析耗时统计。
- Compile Duration：将解析后的 SQL AST 编译成执行计划的耗时。
- Execution Duration：执行 SQL 语句执行计划耗时。
- Expensive Executor OPS：每秒消耗系统资源比较多的算子。包括 Merge Join、Hash Join、Index Look Up Join、Hash Agg、Stream Agg、Sort、TopN 等。
- Queries Using Plan Cache OPS：每秒使用 Plan Cache 的查询数量。
- Plan Cache Miss OPS：每秒出现 Plan Cache Miss 的数量。
- Plan Cache Memory Usage：每个 TiDB 实例上所有 Plan Cache 缓存的执行计划占用的总内存。
- Plan Cache Plan Num：每个 TiDB 实例上所有 Plan Cache 缓存的执行计划总数。

14.8.3.1.6 Distsql

- Distsql Duration：Distsql 处理的时长
- Distsql QPS：每秒 Distsql 的数量
- Distsql Partial QPS：每秒 Partial Results 的数量
- Scan Keys Num：每个 Query 扫描的 Key 的数量
- Scan Keys Partial Num：每一个 Partial Result 扫描的 Key 的数量
- Partial Num：每个 SQL 语句 Partial Results 的数量

14.8.3.1.7 KV Errors

- KV Backoff Duration：KV 每个请求重试的总时间。TiDB 向 TiKV 的请求都有重试机制，这里统计的是向 TiKV 发送请求时遇到错误重试的总时间
- TiClient Region Error OPS：TiKV 返回 Region 相关错误信息的数量
- KV Backoff OPS：TiKV 返回错误信息的数量
- Lock Resolve OPS：TiDB 清理锁操作的数量。当 TiDB 的读写请求遇到锁时，会尝试进行锁清理
- Other Errors OPS：其他类型的错误数量，包括清锁和更新 SafePoint

14.8.3.1.8 KV Request

- KV Request OPS：KV Request 根据 TiKV 显示执行次数
- KV Request Duration 99 by store：根据 TiKV 显示 KV Request 执行时间
- KV Request Duration 99 by type：根据类型显示 KV Request 的执行时间

14.8.3.1.9 PD Client

- PD Client CMD OPS：PD Client 每秒执行命令的数量
- PD Client CMD Duration：PD Client 执行命令耗时
- PD Client CMD Fail OPS：PD Client 每秒执行命令失败的数量
- PD TSO OPS：TiDB 每秒从 PD 获取 TSO 的数量
- PD TSO Wait Duration：TiDB 等待从 PD 返回 TSO 的时间
- PD TSO RPC Duration：TiDB 从向 PD 发送获取 TSO 的请求到接收到 TSO 的耗时
- Start TSO Wait Duration：TiDB 从向 PD 发送获取 start tso 请求开始到开始等待 tso 返回的时间

14.8.3.1.10 Schema Load

- Load Schema Duration：TiDB 从 TiKV 获取 Schema 的时间
- Load Schema OPS：TiDB 从 TiKV 每秒获取 Schema 的数量
- Schema Lease Error OPM：Schema Lease 出错统计，包括 change 和 outdate 两种，change 代表 schema 发生了变化，outdate 代表无法更新 schema，属于较严重错误，出现 outdate 错误时会报警
- Load Privilege OPS：TiDB 从 TiKV 每秒获取权限信息的数量

14.8.3.1.11 DDL

- DDL Duration 95：DDL 语句处理时间的 95% 分位
- Batch Add Index Duration 100：创建索引时每个 Batch 所花费的最大时间
- DDL Waiting Jobs Count：等待的 DDL 任务数量
- DDL META OPM：DDL 每分钟获取 META 的次数
- DDL Worker Duration 99：每个 DDL worker 执行时间的 99% 分位
- Deploy Syncer Duration：Schema Version Syncer 初始化，重启，清空等操作耗时
- Owner Handle Syncer Duration：DDL Owner 在执行更新，获取以及检查 Schema Version 的耗时
- Update Self Version Duration：Schema Version Syncer 更新版本信息耗时
- DDL OPM：DDL 语句的每秒执行次数
- DDL backfill progress in percentage：backfill DDL 任务的进度展示

14.8.3.1.12 Statistics

- Auto Analyze Duration 95：自动 ANALYZE 耗时
- Auto Analyze QPS：自动 ANALYZE 数量
- Stats Inaccuracy Rate：统计信息不准确度
- Pseudo Estimation OPS：使用假的统计信息优化 SQL 的数量
- Dump Feedback OPS：存储统计信息 Feedback 的数量
- Store Query Feedback QPS：存储合并查询的 Feedback 信息的每秒操作数量，该操作在 TiDB 内存中进行
- Significant Feedback：重要的 Feedback 更新统计信息的数量
- Update Stats OPS：利用 Feedback 更新统计信息的数量
- Fast Analyze Status 100：快速收集统计信息的状态

14.8.3.1.13 Owner

- New ETCD Session Duration 95：创建一个新的 etcd 会话花费的时间。TiDB 通过 etcd client 连接 PD 中的 etcd 保存/读取部分元数据信息。这里记录了创建会话花费的时间
- Owner Watcher OPS：DDL owner watch PD 的 etcd 的元数据的 goroutine 的每秒操作次数

14.8.3.1.14 Meta

- AutoID QPS：AutoID 相关操作的数量统计，包括全局 ID 分配、单个 Table AutoID 分配、单个 Table AutoID Rebase 三种操作
- AutoID Duration：AutoID 相关操作的耗时
- Region Cache Error OPS：TiDB 缓存的 region 信息每秒遇到的错误次数
- Meta Operations Duration 99：元数据操作延迟

14.8.3.1.15 GC

- Worker Action OPM：GC 相关操作的数量，包括 run_job，resolve_lock，delete_range 等操作
- Duration 99：GC 相关操作的耗时
- Config：GC 的数据保存时长 (life time) 和 GC 运行间隔 (run interval) 配置
- GC Failure OPM：GC 相关操作失败的数量
- Delete Range Failure OPM：Delete range 失败的次数
- Too Many Locks Error OPM：GC 清锁过多错误的数量
- Action Result OPM：GC 相关操作结果数量
- Delete Range Task Status：Delete range 的任务状态，包含完成和失败状态
- Push Task Duration 95：将 GC 子任务推送给 GC worker 的耗时

14.8.3.1.16 Batch Client

- Pending Request Count by TiKV：TiKV 批量消息处理的等待数量
- Wait Duration 95: 批量消息处理的等待时间。
- Batch Client Unavailable Duration 95：批处理客户端的不可用时长。
- No Available Connection Counter：批处理客户端不可用的连接数。

14.8.3.1.17 TTL

- TiDB CPU Usage: 每个 TiDB 实例的 CPU 使用。
- TiKV IO MBps: 每个 TiKV 实例的 I/O 吞吐量。
- TiKV CPU: 每个 TiKV 实例的 CPU 使用。
- TTL QPS By Type：TTL 任务产生的不同类型语句的 QPS 信息。
- TTL Insert Rows Per Second: 每秒钟向 TTL 表插入的数据行数。
- TTL Processed Rows Per Second：TTL 任务每秒处理的过期数据的行数。
- TTL Insert Rows Per Hour: 每小时总共向 TTL 表插入的行数。
- TTL Delete Rows Per Hour: 每小时总共删除的过期行数。
- TTL Scan/Delete Query Duration：TTL 的扫描/删除语句的执行时间。
- TTL Scan/Delete Worker Time By Phase：TTL 内部工作线程的不同阶段所占用的时间。
- TTL Job Count By Status：当前正在执行的 TTL 任务的数量。
- TTL Task Count By Status：当前正在执行的 TTL 子任务的数量。

14.8.4 PD 重要监控指标详解

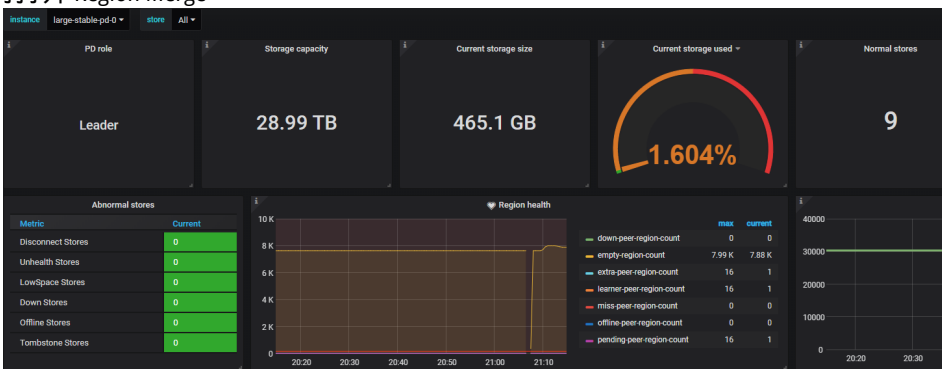
使用 TiUP 部署 TiDB 集群时，一键部署监控系统 (Prometheus & Grafana)，监控架构参见TiDB 监控框架概述。

目前 Grafana Dashboard 整体分为 PD、 TiDB、 TiKV、 Node_exporter、 Overview、 Performance_overview 等。

对于日常运维，我们通过观察 PD 面板上的 Metrics，可以了解 PD 当前的状态。

以下为 PD Dashboard 监控说明：

- PD role：当前 PD 的角色
- Storage capacity：TiDB 集群总可用数据库空间大小
- Current storage size：TiDB 集群目前已用数据库空间大小
- Current storage usage：TiDB 集群存储空间的使用率
- Normal stores：处于正常状态的节点数目
- Number of Regions：当前集群的 Region 总量
- Abnormal stores：处于异常状态的节点数目，正常情况应当为 0
- Region health：集群所有 Region 的状态。通常情况下，pending 或 down 的 peer 应该少于 100，miss 的 peer 不能一直大于 0，empty Region 过多需及时打开 Region Merge
- Current peer count：当前集群 peer 的总量



14.8.4.1 Cluster

- PD scheduler config：PD 调度配置列表
- Cluster ID：集群的 cluster id，唯一标识
- Current TSO：当前分配 TSO 的物理时间戳部分
- Current ID allocation：当前可分配 ID 的最大值
- Region label isolation level：不同 label 所在的 level 的 Region 数量
- Label distribution：集群中 TiKV 节点的 label 分布情况
- Store Limit：Store 的调度限流状态

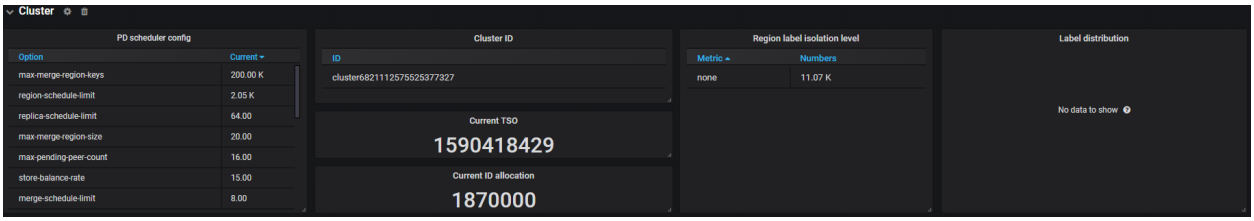


图 326: PD Dashboard - Cluster metrics

14.8.4.2 Operator

- Schedule operator create：新创建的不同 operator 的数量，单位 opm 代表一分钟内创建的个数
- Schedule operator check：已检查的 operator 的次数，主要检查是否当前步骤已经执行完成，如果是，则执行下一个步骤
- Schedule operator finish：已完成调度的 operator 的数量
- Schedule operator timeout：已超时的 operator 的数量
- Schedule operator replaced or canceled：已取消或者被替换的 operator 的数量
- Schedule operators count by state：不同状态的 operator 的数量
- Operator finish duration：已完成的 operator 所花费的最长时间
- Operator step duration：已完成的 operator 的步骤所花费的最长时间

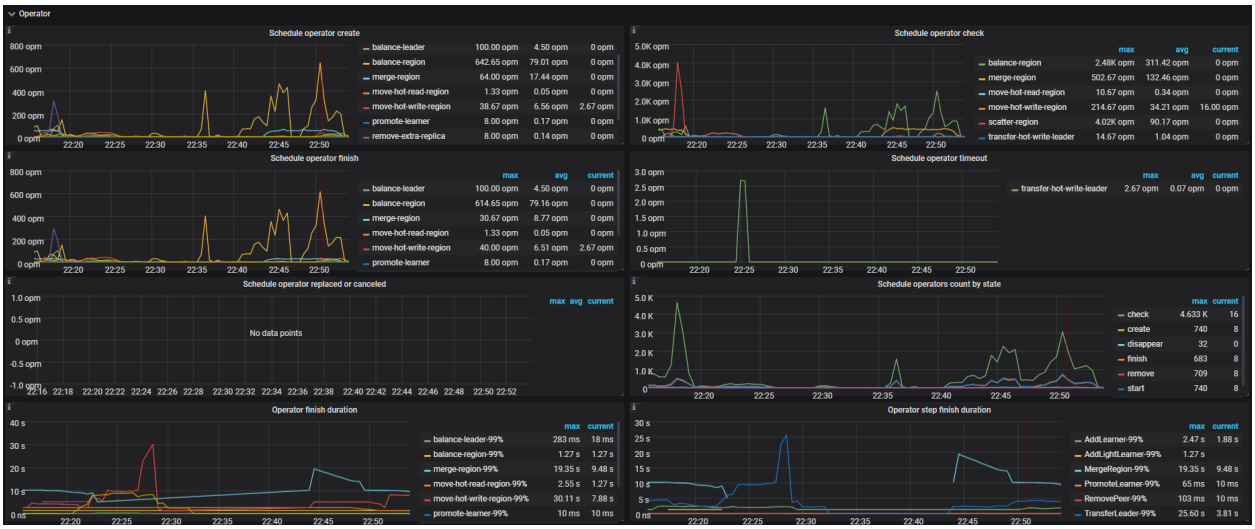


图 327: PD Dashboard - Operator metrics

14.8.4.3 Statistics - Balance

- Store capacity：每个 TiKV 实例的总的空间大小
- Store available：每个 TiKV 实例的可用空间大小
- Store used：每个 TiKV 实例的已使用空间大小
- Size amplification：每个 TiKV 实例的空间放大比率
- Size available ratio：每个 TiKV 实例的可用空间比率
- Store leader score：每个 TiKV 实例的 leader 分数
- Store Region score：每个 TiKV 实例的 Region 分数
- Store leader size：每个 TiKV 实例上所有 leader 的大小
- Store Region size：每个 TiKV 实例上所有 Region 的大小
- Store leader count：每个 TiKV 实例上所有 leader 的数量
- Store Region count：每个 TiKV 实例上所有 Region 的数量



图 328: PD Dashboard - Balance metrics

14.8.4.4 Statistics - hot write

- Hot Region’s leader distribution：每个 TiKV 实例上成为写入热点的 leader 的数量
- Total written bytes on hot leader Regions：每个 TiKV 实例上所有成为写入热点的 leader 的总的写入流量大小
- Hot write Region’s peer distribution：每个 TiKV 实例上成为写入热点的 peer 的数量

- Total written bytes on hot peer Regions：每个 TiKV 实例上所有成为写入热点的 peer 的写入流量大小
- Store Write rate bytes：每个 TiKV 实例总的写入的流量
- Store Write rate keys：每个 TiKV 实例总的写入 keys
- Hot cache write entry number：每个 TiKV 实例进入热点统计模块的 peer 的数量
- Selector events：热点调度中选择器的事件发生次数
- Direction of hotspot move leader：热点调度中 leader 的调度方向，正数代表调入，负数代表调出
- Direction of hotspot move peer：热点调度中 peer 的调度方向，正数代表调入，负数代表调出



图 329: PD Dashboard - Hot write metrics

14.8.4.5 Statistics - hot read

- Hot Region’s peer distribution：每个 TiKV 实例上成为读取热点的 peer 的数量
- Total read bytes on hot peer Regions：每个 TiKV 实例上所有成为读取热点的 peer 的总的读取流量大小
- Store read rate bytes：每个 TiKV 实例总的读取的流量
- Store read rate keys：每个 TiKV 实例总的读取 keys
- Hot cache read entry number：每个 TiKV 实例进入热点统计模块的 peer 的数量



图 330: PD Dashboard - Hot read metrics

14.8.4.6 Scheduler

- Scheduler is running：所有正在运行的 scheduler
- Balance leader movement：leader 移动的详细情况
- Balance Region movement：Region 移动的详细情况
- Balance leader event：balance leader 的事件数量
- Balance Region event：balance Region 的事件数量
- Balance leader scheduler：balance-leader scheduler 的状态
- Balance Region scheduler：balance-region scheduler 的状态
- Replica checker：replica checker 的状态
- Rule checker：rule checker 的状态
- Region merge checker：merge checker 的状态
- Filter target：尝试选择 Store 作为调度 taget 时没有通过 Filter 的计数
- Filter source：尝试选择 Store 作为调度 source 时没有通过 Filter 的计数
- Balance Direction：Store 被选作调度 target 或 source 的次数

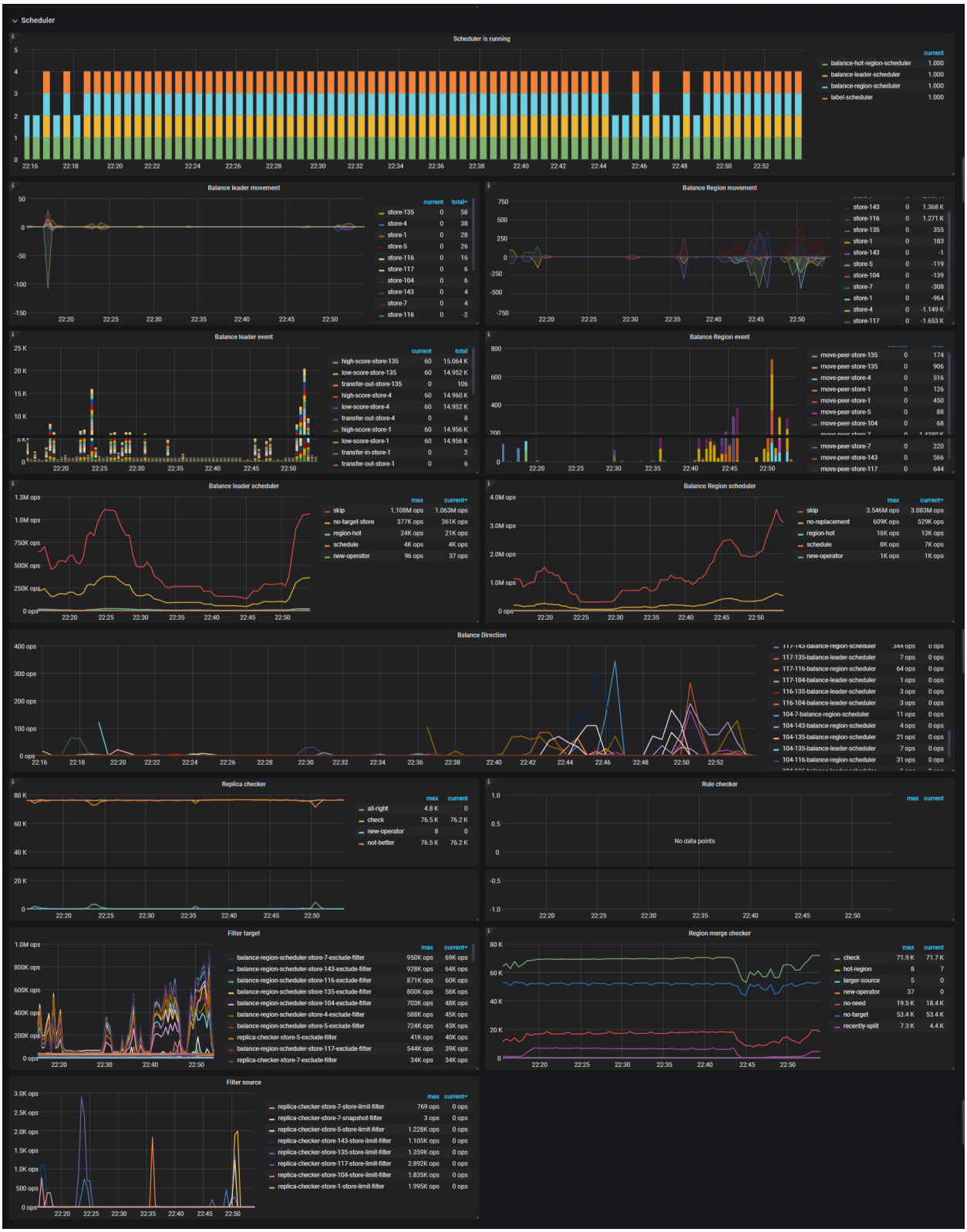


图 331: PD Dashboard - Scheduler metrics

14.8.4.7 gRPC

- Completed commands rate：gRPC 命令的完成速率
- 99% Completed commands duration：99% 命令的最长消耗时间



图 332: PD Dashboard - gRPC metrics

14.8.4.8 etcd

- Handle transactions count：etcd 的事务个数
- 99% Handle transactions duration：99% 的情况下，处理 etcd 事务所需花费的时间
- 99% WAL fsync duration：99% 的情况下，持久化 WAL 所需花费的时间，这个值通常应该小于 1s
- 99% Peer round trip time seconds：99% 的情况下，etcd 的网络延时，这个值通常应该小于 1s
- etcd disk WAL fsync rate：etcd 持久化 WAL 的速率
- Raft term：当前 Raft 的 term
- Raft committed index：最后一次 commit 的 Raft index
- Raft applied index：最后一次 apply 的 Raft index



图 333: PD Dashboard - etcd metrics

14.8.4.9 TiDB

- PD Server TSO handle time and Client recv time：从 PD 开始处理 TSO 请求到 client 端接收到 TSO 的总耗时
- Handle requests count：TiDB 的请求数量
- Handle requests duration：每个请求所花费的时间，99% 的情况下，应该小于 100ms

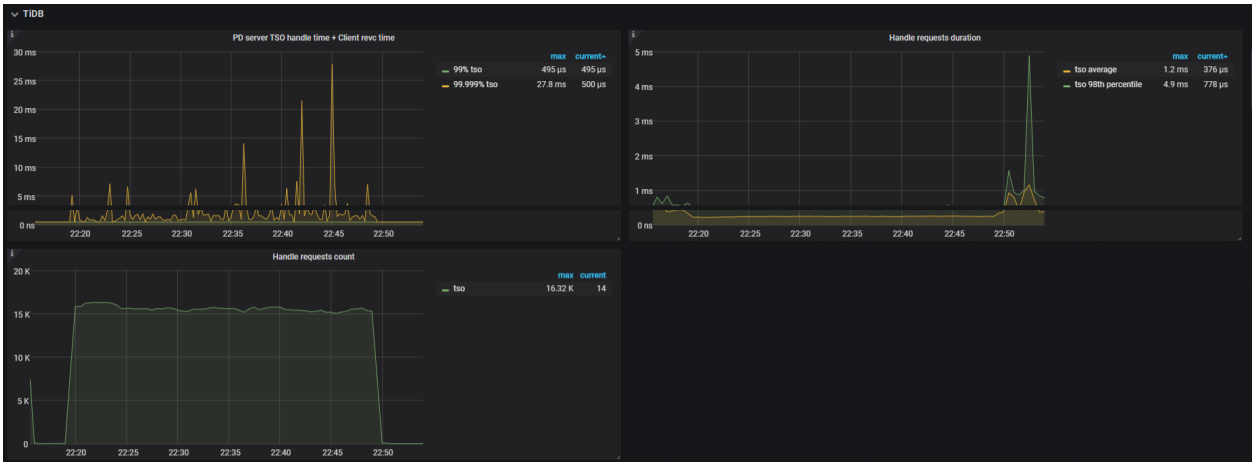


图 334: PD Dashboard - TiDB metrics

14.8.4.10 Heartbeat

- Heartbeat region event QPS：心跳处理 region 的 QPS，包括更新缓存和持久化
- Region heartbeat report：TiKV 向 PD 发送的心跳个数
- Region heartbeat report error：TiKV 向 PD 发送的异常的心跳个数
- Region heartbeat report active：TiKV 向 PD 发送的正常的心跳个数
- Region schedule push：PD 向 TiKV 发送的调度命令的个数
- 99% Region heartbeat latency：99% 的情况下，心跳的延迟

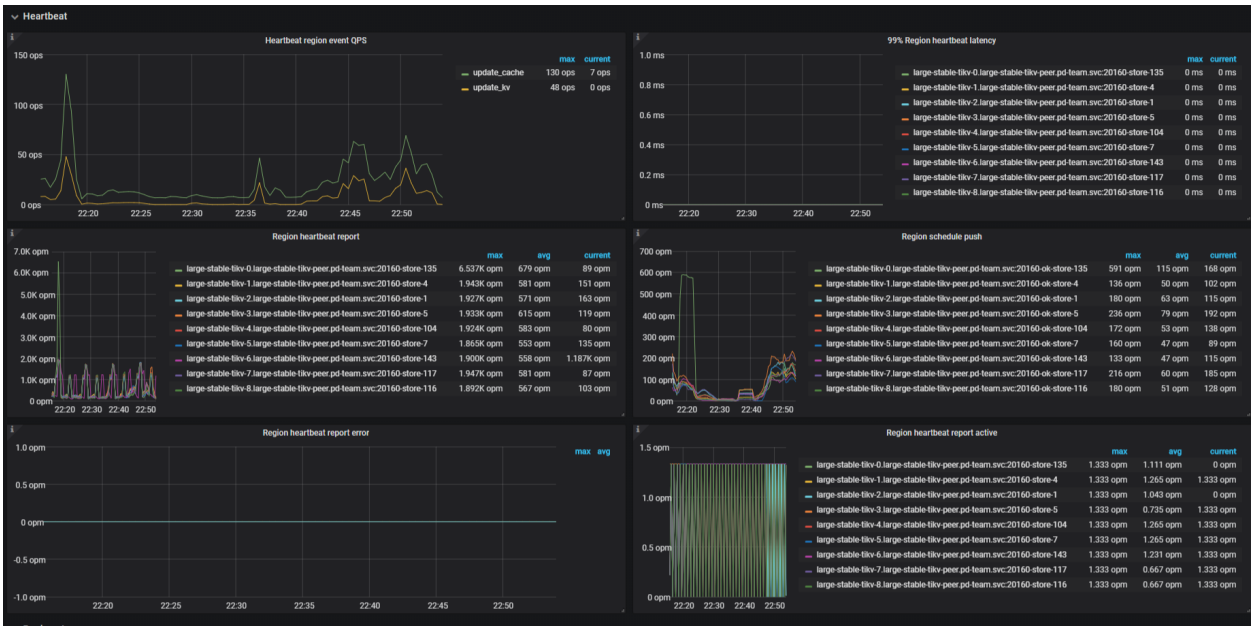


图 335: PD Dashboard - Heartbeat metrics

14.8.4.11 Region storage

- Syncer Index：Leader 记录 Region 变更历史的最大 index
- history last index：Follower 成功同步的 Region 变更历史的 index



图 336: PD Dashboard - Region storage

14.8.5 TiKV 监控指标详解

使用 TiUP 部署 TiDB 集群时，一键部署监控系统 (Prometheus & Grafana)，监控架构参见TiDB 监控框架概述。

目前 Grafana Dashboard 整体分为 PD、 TiDB、 TiKV、 Node_exporter、 Overview、 Performance_overview 等。

14.8.5.1 TiKV-Details 面板

对于日常运维，通过观察 TiKV-Details 面板上的指标，可以了解 TiKV 当前的状态。根据性能地图可以检查集群的状态是否符合预期。

以下为 TiKV-Details 默认的监控信息：

14.8.5.1.1 Cluster

- Store size：每个 TiKV 实例的使用的存储空间的大小
- Available size：每个 TiKV 实例的可用的存储空间的大小
- Capacity size：每个 TiKV 实例的存储容量的大小
- CPU：每个 TiKV 实例 CPU 的使用率
- Memory：每个 TiKV 实例内存的使用情况
- IO utilization：每个 TiKV 实例 IO 的使用率
- MBps：每个 TiKV 实例写入和读取的数据量大小
- QPS：每个 TiKV 实例上各种命令的 QPS
- Errps：每个 TiKV 实例上 gRPC 消息失败的速率
- leader：每个 TiKV 实例 leader 的个数
- Region：每个 TiKV 实例 Region 的个数
- Uptime：自上次重启以来 TiKV 正常运行的时间

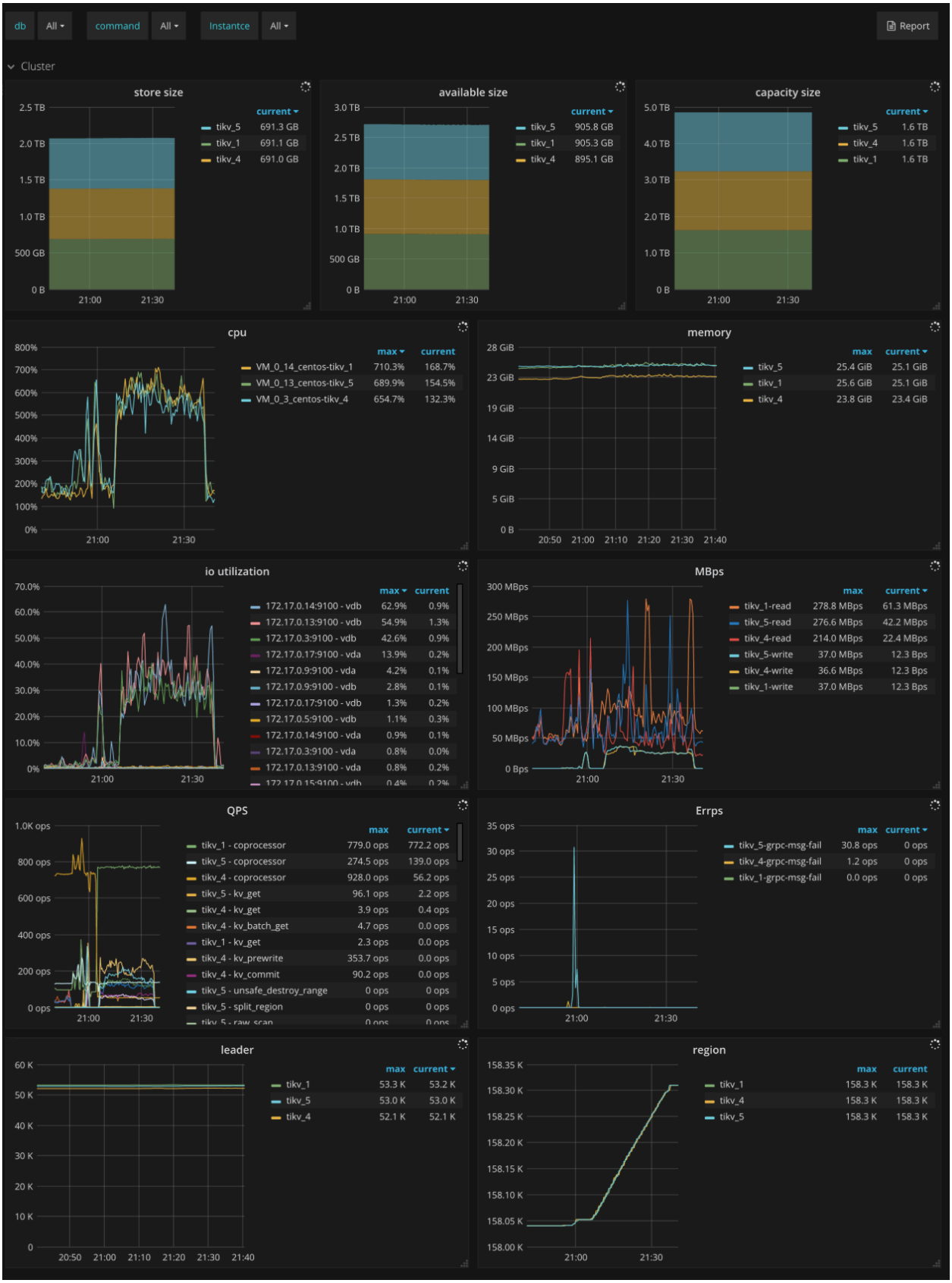


图 337: TiKV Dashboard - Cluster metrics

14.8.5.1.2 Errors

- Critical error：严重错误的数量
- Server is busy：各种会导致 TiKV 实例暂时不可用的事件个数，如 write stall，channel full 等，正常情况下应当为 0
- Server report failures：server 报错的消息个数，正常情况下应当为 0
- Raftstore error：每个 TiKV 实例上 raftstore 发生错误的个数
- Scheduler error：每个 TiKV 实例上 scheduler 发生错误的个数
- Coprocessor error：每个 TiKV 实例上 coprocessor 发生错误的个数
- gRPC message error：每个 TiKV 实例上 gRPC 消息发生错误的个数
- Leader drop：每个 TiKV 实例上 drop leader 的个数
- Leader missing：每个 TiKV 实例上 missing leader 的个数
- Log Replication Rejected：每个 TiKV 实例上由于内存不足而拒绝 logappend 消息的个数



图 338: TiKV Dashboard - Errors metrics

14.8.5.1.3 Server

- CF size：每个列族的大小
- Store size：每个 TiKV 实例的使用的存储空间的大小
- Channel full：每个 TiKV 实例上 channel full 错误的数量，正常情况下应当为 0
- Active written leaders：各个 TiKV 实例中正在被写入的 Leader 的数量
- Approximate Region size：每个 Region 近似的大小
- Approximate Region size Histogram：每个 Region 近似大小的直方图
- Region average written keys：每个 TiKV 实例上所有 Region 的平均 key 写入个数
- Region average written bytes：每个 TiKV 实例上所有 Region 的平均写入大小

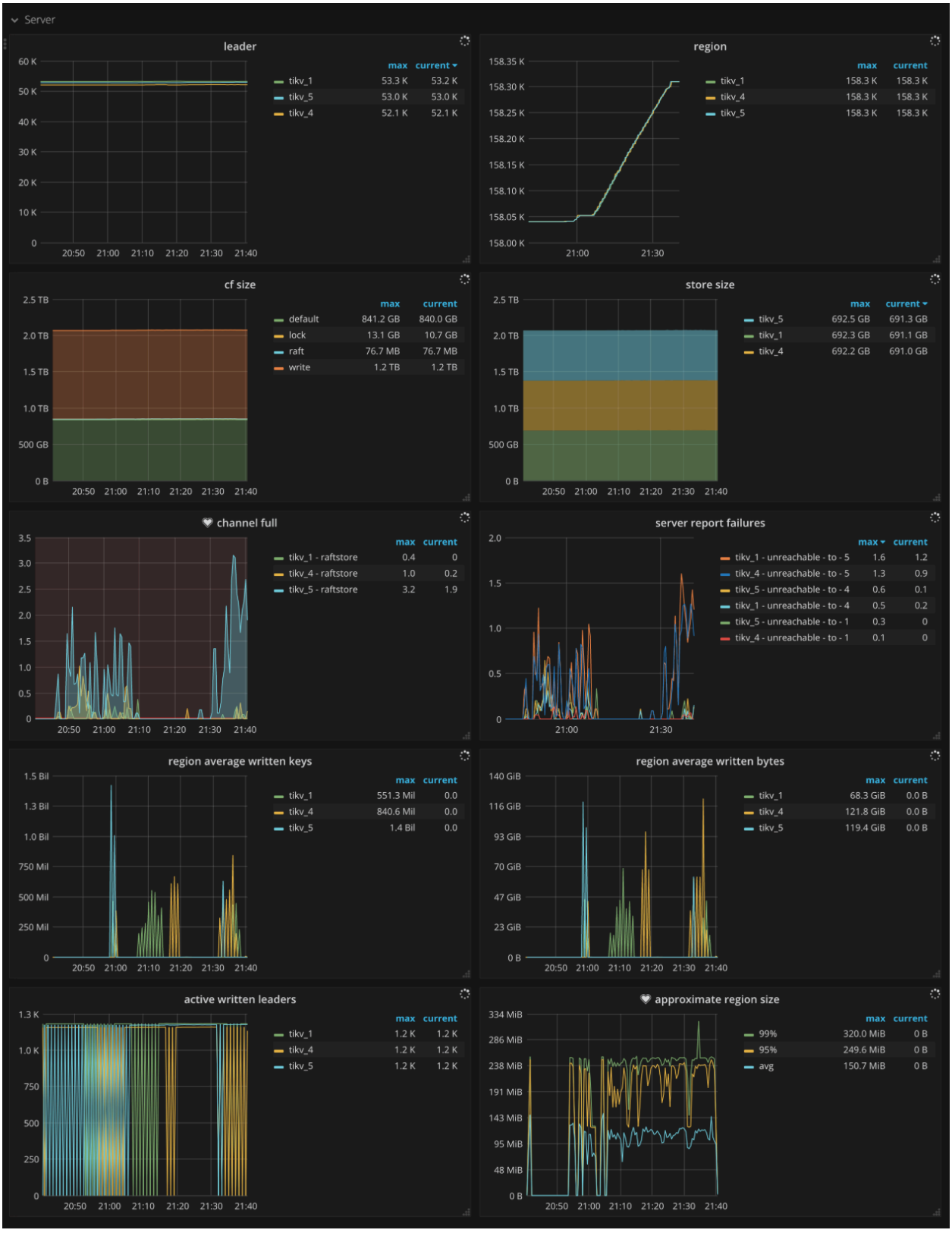


图 339: TiKV Dashboard - Server metrics

14.8.5.1.4 gRPC

- gRPC message count：每种 gRPC 请求的速度
- gRPC message failed：失败的 gRPC 请求的速度
- 99% gRPC message duration：99% gRPC 请求的执行时间小于该值
- Average gRPC message duration：gRPC 请求平均的执行时间
- gRPC batch size：TiDB 与 TiKV 之间 grpc 请求的 batch 大小
- raft message batch size：TiKV 与 TiKV 之间 raft 消息的 batch 大小
- gRPC request sources QPS: 不同 gRPC 请求来源的速度
- gRPC request sources duration: 不同 gRPC 请求来源的执行总时间
- gRPC resource group QPS: 不同 resource group 的 gRPC 请求速度

14.8.5.1.5 Thread CPU

- Raft store CPU：raftstore 线程的 CPU 使用率，通常应低于 80% * `raftstore.store-pool-size`
- Async apply CPU：async apply 线程的 CPU 使用率，通常应低于 90% * `raftstore.apply-pool-size`
- Store writer CPU：async io 线程的 CPU 使用率，通常应低于 90% * `raftstore.store-io-pool-size`
- gRPC poll CPU：gRPC 线程的 CPU 使用率，通常应低于 80% * `server.grpc-concurrency`
- Scheduler worker CPU：scheduler worker 线程的 CPU 使用率，通常应低于 90% * `storage.scheduler-worker-pool-size`
- Storage ReadPool CPU：storage read pool 线程的 CPU 使用率
- Unified read pool CPU：unified read pool 线程的 CPU 使用率
- RocksDB CPU：RocksDB 线程的 CPU 使用率
- Coprocessor CPU：coprocessor 线程的 CPU 使用率
- GC worker CPU：GC worker 线程的 CPU 使用率
- BackGround worker CPU：background worker 线程的 CPU 使用率
- Import CPU：Import 线程的 CPU 使用率
- Backup Worker CPU：Backup 线程的 CPU 使用率
- CDC Worker CPU：CDC Worker 线程的 CPU 使用率
- CDC endpoint CPU：CDC endpoint 的 CPU 使用率
- Raftlog fetch worker CPU：Async raft log fetcher worker 的 CPU 使用率
- TSO Worker CPU: TSO Worker 线程的 CPU 使用率

14.8.5.1.6 PD

- PD requests：TiKV 发送给 PD 的请求速度
- PD request duration (average)：TiKV 发送给 PD 的请求处理的平均时间
- PD heartbeats：发送给 PD 的心跳的速度
- PD validate peers：TiKV 发送给 PD 用于验证 TiKV 的 peer 有效的消息的速度

14.8.5.1.7 Raft IO

- Apply log duration：Raft apply 日志所花费的时间
- Apply log duration per server：每个 TiKV 实例上 Raft apply 日志所花费的时间

- Append log duration：Raft append 日志所花费的时间
- Append log duration per server：每个 TiKV 实例上 Raft append 日志所花费的时间
- Commit log duration：Raft commit 日志所花费的时间
- Commit log duration per server：每个 TiKV 实例上 Raft commit 日志所花费的时间



图 340: TiKV Dashboard - Raft IO metrics

14.8.5.1.8 Raft process

- Ready handled：Raft 中不同 ready 类型的 ops
  - count: 批量处理 ready 的 ops
  - has_ready_region: 获得 ready 的 Region 的 ops
  - pending_region: 被检查是否获得 ready 的 Region 的 ops，v3.0.0 后废弃
  - message: ready 内待发送 message 的 ops
  - append: ready 内 Raft log entry 的 ops
  - commit: ready 内 committed Raft log entry 的 ops
  - snapshot: 携带 snapshot 的 ready 的 ops
- Max Duration of Raft store events：raftstore 处理事件最慢一次所花费的时间
- Replica read lock checking duration：处理 Replica Read 时检查 lock 所花费的时间
- Peer msg length distribution：每个 TiKV 中每个 region 一次性处理 Peer 消息的个数，消息越多说明 peer 越繁忙。

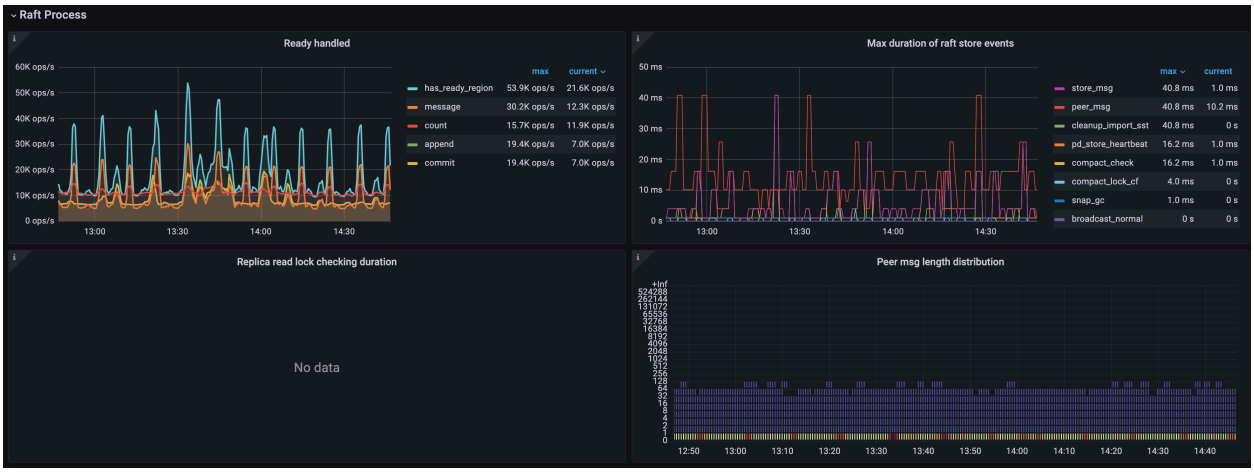


图 341: TiKV Dashboard - Raft process metrics

14.8.5.1.9 Raft message

- Sent messages per server：每个 TiKV 实例发送 Raft 消息的 ops
- Flush messages per server：每个 TiKV 实例中 raft client 往外 flush Raft 消息的 ops
- Receive messages per server：每个 TiKV 实例接受 Raft 消息的 ops
- Messages：发送不同类型的 Raft 消息的 ops
- Vote：Raft 投票消息发送的 ops
- Raft dropped messages：每秒钟丢弃不同类型的 Raft 消息的个数

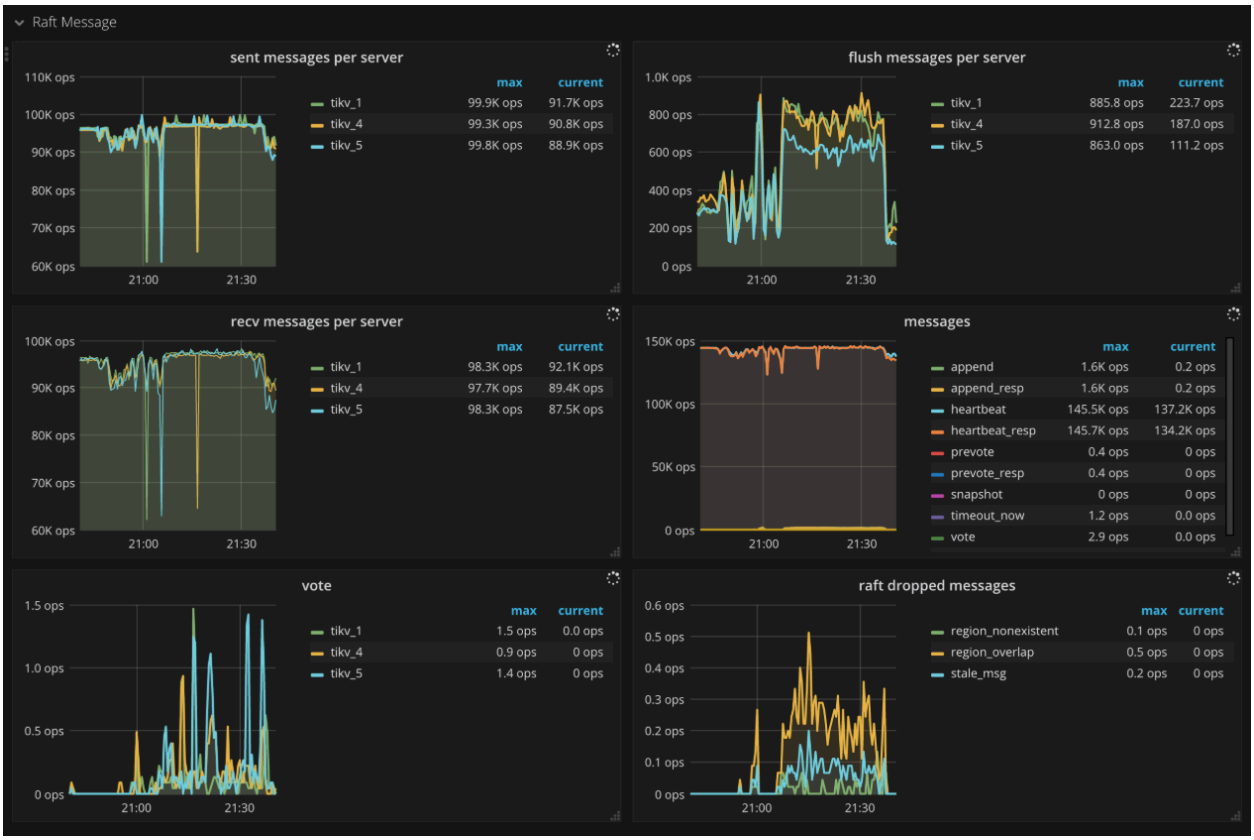


图 342: TiKV Dashboard - Raft message metrics

14.8.5.1.10 Raft propose

- Raft apply proposals per ready：在一个 batch 内，apply proposal 时每个 ready 中包含 proposal 的个数的直方图
- Raft read/write proposals：不同类型的 proposal 的 ops
- Raft read proposals per server：每个 TiKV 实例发起读 proposal 的 ops
- Raft write proposals per server：每个 TiKV 实例发起写 proposal 的 ops
- Propose wait duration：proposal 的等待时间的直方图
- Propose wait duration per server：每个 TiKV 实例上每个 proposal 的等待时间的直方图
- Apply wait duration：apply 的等待时间的直方图
- Apply wait duration per server：每个 TiKV 实例上每个 apply 的等待时间的直方图
- Raft log speed：peer propose 日志的平均速度

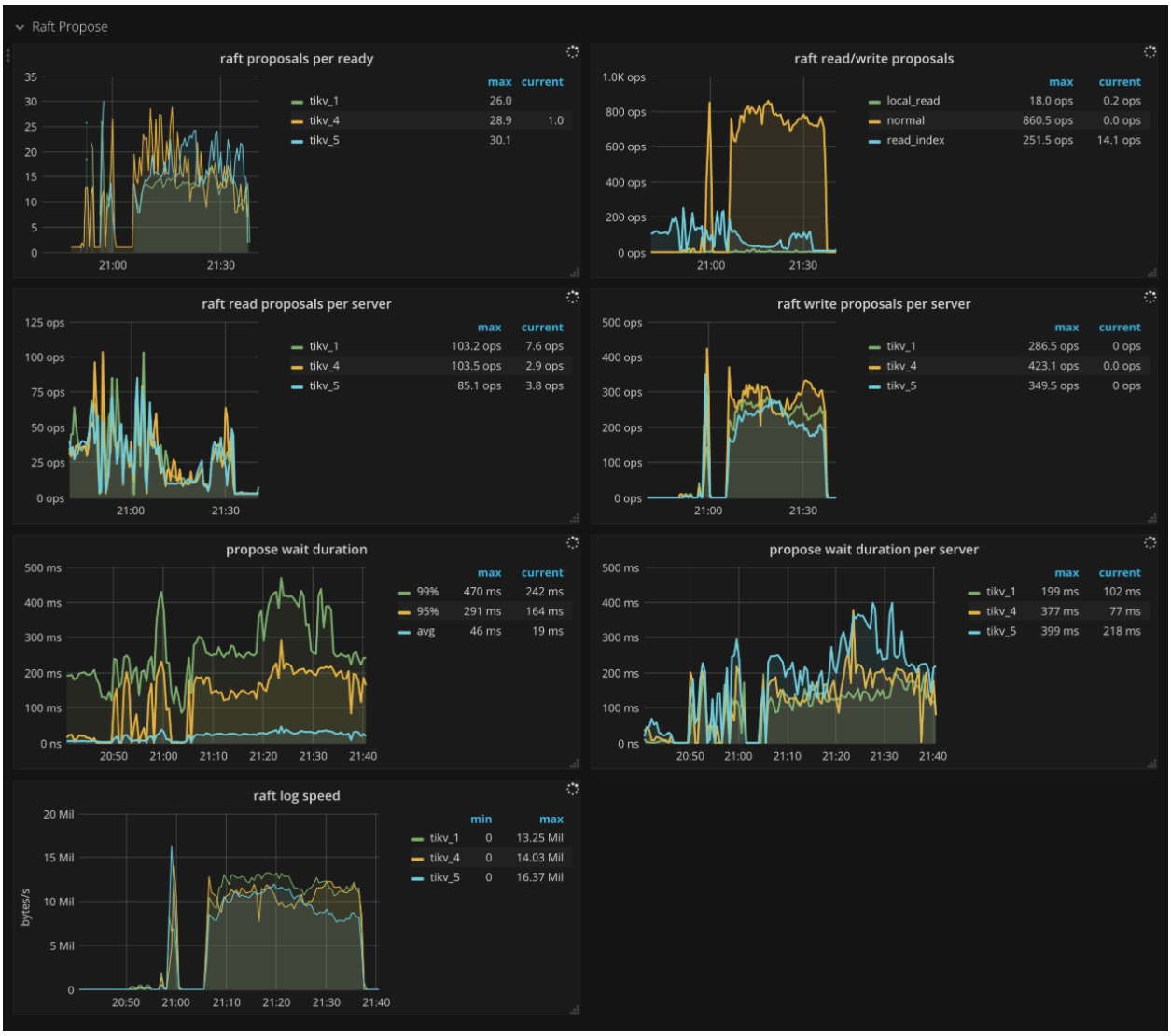


图 343: TiKV Dashboard - Raft propose metrics

14.8.5.1.11 Raft admin

- Admin proposals：admin proposal 的 ops
- Admin apply：apply 命令的 ops
- Check split：split check 命令的 ops
- 99.99% Check split duration：99.99% 的情况下，split check 所需花费的时间

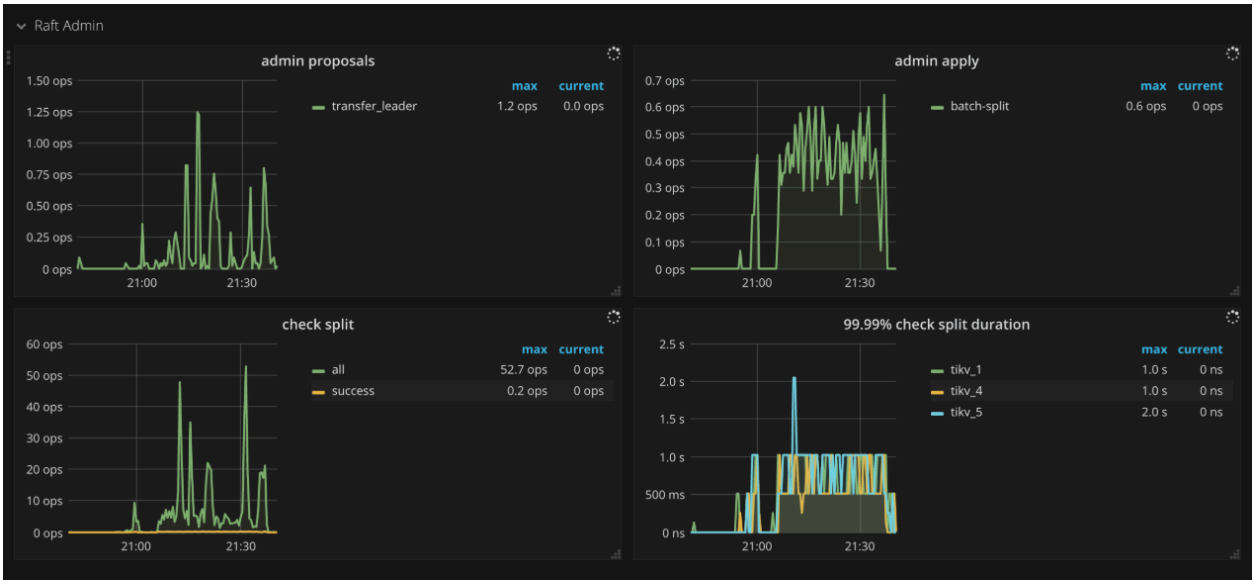


图 344: TiKV Dashboard - Raft admin metrics

14.8.5.1.12 Local reader

- Local reader requests：所有请求的总数以及 local read 线程拒绝的请求数量



图 345: TiKV Dashboard - Local reader metrics

14.8.5.1.13 Unified Read Pool

- Time used by level：在 unified read pool 中每个级别使用的时间，级别 0 指小查询

- Level 0 chance：在 unified read pool 中调度的 level 0 任务的比例
- Running tasks：在 unified read pool 中并发运行的任务数量

14.8.5.1.14 Storage

- Storage command total：收到不同命令的 ops
- Storage async request error：异步请求出错的 ops
- Storage async snapshot duration：异步处理 snapshot 所花费的时间，99% 的情况下，应该小于 1s
- Storage async write duration：异步写所花费的时间，99% 的情况下，应该小于 1s

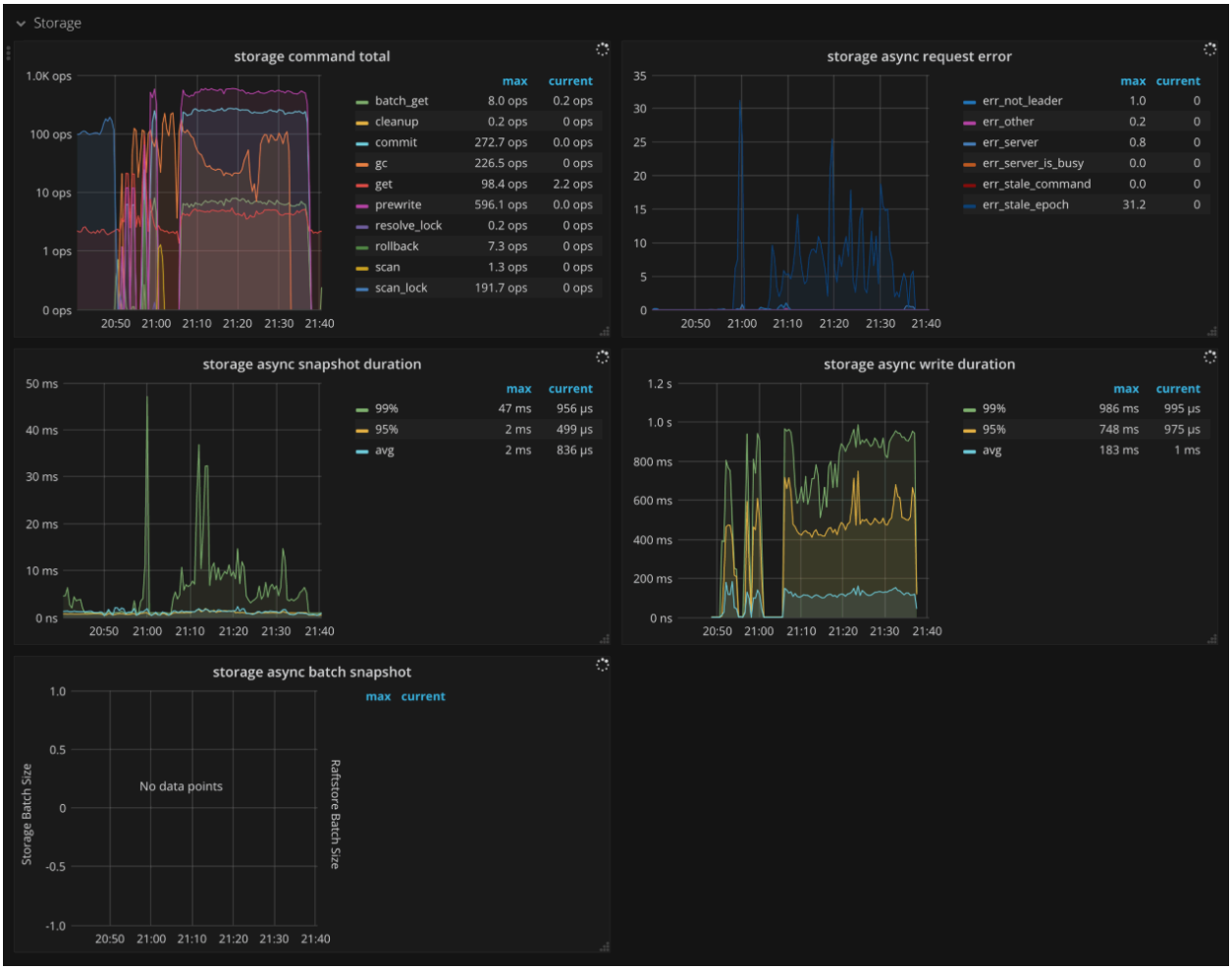


图 346: TiKV Dashboard - Storage metrics

14.8.5.1.15 Scheduler

- Scheduler stage total：每种命令不同阶段的 ops，正常情况下，不会在短时间内出现大量的错误
- Scheduler writing bytes：每个 TiKV 实例正在处理的命令的写入字节数量
- Scheduler priority commands：不同优先级命令的 ops

- Scheduler pending commands：每个 TiKV 实例上 pending 命令的 ops

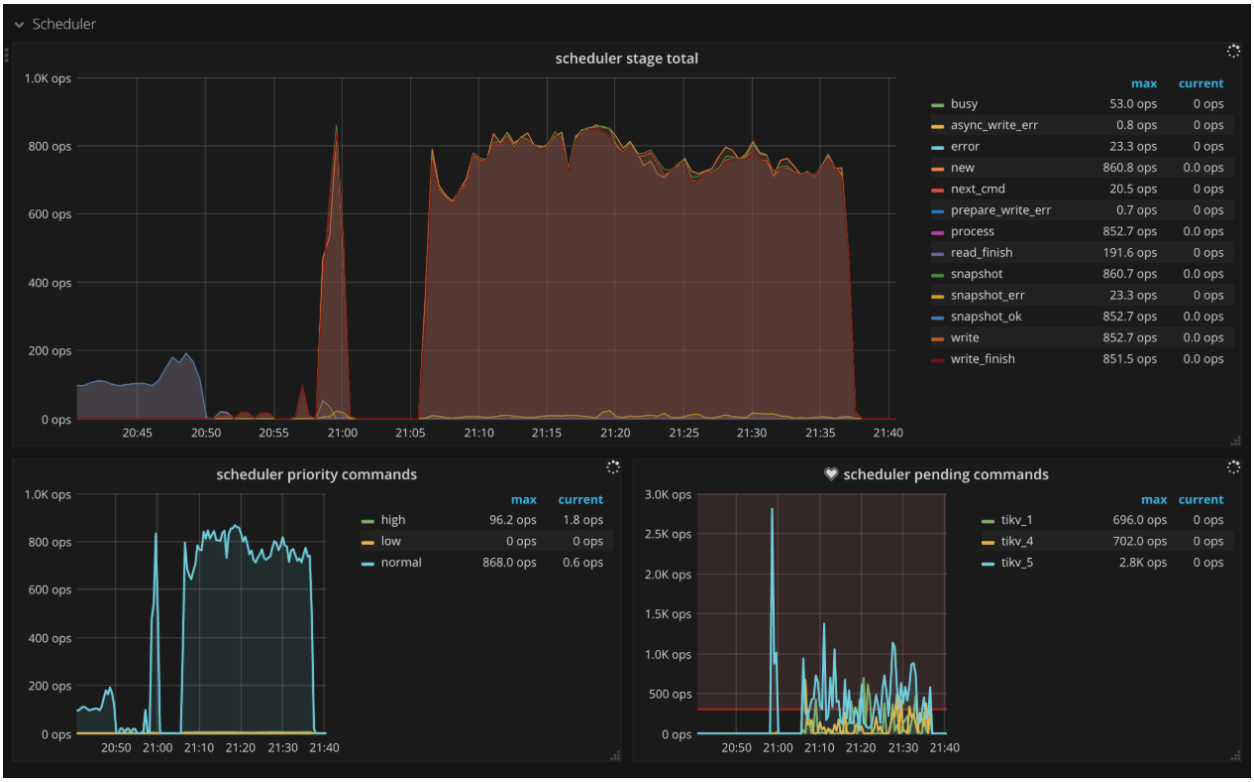


图 347: TiKV Dashboard - Scheduler metrics

14.8.5.1.16 Scheduler - commit

- Scheduler stage total：commit 中每个命令所处不同阶段的 ops，正常情况下，不会在短时间内出现大量的错误
- Scheduler command duration：执行 commit 命令所需花费的时间，正常情况下，应该小于 1s
- Scheduler latch wait duration：由于 latch wait 造成的时间开销，正常情况下，应该小于 1s
- Scheduler keys read：commit 命令读取 key 的个数
- Scheduler keys written：commit 命令写入 key 的个数
- Scheduler scan details：执行 commit 命令时，扫描每个 CF 中 key 的详细情况
- Scheduler scan details [lock]：执行 commit 命令时，扫描每个 lock CF 中 key 的详细情况
- Scheduler scan details [write]：执行 commit 命令时，扫描每个 write CF 中 key 的详细情况
- Scheduler scan details [default]：执行 commit 命令时，扫描每个 default CF 中 key 的详细情况



图 348: TiKV Dashboard - Scheduler commit metrics

14.8.5.1.17 Scheduler - pessimistic_rollback

- Scheduler stage total：pessimistic_rollback 中每个命令所处不同阶段的 ops，正常情况下，不会在短时间内出现大量的错误
- Scheduler command duration：执行 pessimistic_rollback 命令所需花费的时间，正常情况下，应该小于 1s
- Scheduler latch wait duration：由于 latch wait 造成的时间开销，正常情况下，应该小于 1s
- Scheduler keys read：pessimistic_rollback 命令读取 key 的个数
- Scheduler keys written：pessimistic_rollback 命令写入 key 的个数

- Scheduler scan details：执行 pessimistic_rollback 命令时，扫描每个 CF 中 key 的详细情况
- Scheduler scan details [lock]：执行 pessimistic_rollback 命令时，扫描每个 lock CF 中 key 的详细情况
- Scheduler scan details [write]：执行 pessimistic_rollback 命令时，扫描每个 write CF 中 key 的详细情况
- Scheduler scan details [default]：执行 pessimistic_rollback 命令时，扫描每个 default CF 中 key 的详细情况

14.8.5.1.18 Scheduler - prewrite

- Scheduler stage total：prewrite 中每个命令所处不同阶段的 ops，正常情况下，不会在短时间内出现大量的错误
- Scheduler command duration：执行 prewrite 命令所需花费的时间，正常情况下，应该小于 1s
- Scheduler latch wait duration：由于 latch wait 造成的时间开销，正常情况下，应该小于 1s
- Scheduler keys read：prewrite 命令读取 key 的个数
- Scheduler keys written：prewrite 命令写入 key 的个数
- Scheduler scan details：执行 prewrite 命令时，扫描每个 CF 中 key 的详细情况
- Scheduler scan details [lock]：执行 prewrite 命令时，扫描每个 lock CF 中 key 的详细情况
- Scheduler scan details [write]：执行 prewrite 命令时，扫描每个 write CF 中 key 的详细情况
- Scheduler scan details [default]：执行 prewrite 命令时，扫描每个 default CF 中 key 的详细情况

14.8.5.1.19 Scheduler - rollback

- Scheduler stage total：rollback 中每个命令所处不同阶段的 ops，正常情况下，不会在短时间内出现大量的错误
- Scheduler command duration：执行 rollback 命令所需花费的时间，正常情况下，应该小于 1s
- Scheduler latch wait duration：由于 latch wait 造成的时间开销，正常情况下，应该小于 1s
- Scheduler keys read：rollback 命令读取 key 的个数
- Scheduler keys written：rollback 命令写入 key 的个数
- Scheduler scan details：执行 rollback 命令时，扫描每个 CF 中 key 的详细情况
- Scheduler scan details [lock]：执行 rollback 命令时，扫描每个 lock CF 中 key 的详细情况
- Scheduler scan details [write]：执行 rollback 命令时，扫描每个 write CF 中 key 的详细情况
- Scheduler scan details [default]：执行 rollback 命令时，扫描每个 default CF 中 key 的详细情况

14.8.5.1.20 GC

- GC tasks：由 gc_worker 处理的 GC 任务的个数
- GC tasks Duration：执行 GC 任务时所花费的时间
- TiDB GC seconds：TiDB 执行 GC 花费的时间
- TiDB GC worker actions：TiDB GC worker 的不同 action 的个数
- TiKV AutoGC Working：Auto GC 管理器的工作状态
- ResolveLocks Progress：GC 第一阶段 (ResolveLocks) 的进度
- TiKV Auto GC Progress：GC 第二阶段的进度
- GC speed：GC 每秒删除的 key 的数量
- TiKV Auto GC SafePoint：TiKV GC 的 safe point 的数值，safe point 为当前 GC 的时间戳
- GC lifetime：TiDB 设置的 GC lifetime
- GC interval：TiDB 设置的 GC 间隔
- GC in Compaction Filter：write CF 的 Compaction Filter 中已过滤版本的数量

14.8.5.1.21 Snapshot

- Rate snapshot message：发送 Raft snapshot 消息的速率
- 99% Handle snapshot duration：99% 的情况下，处理 snapshot 所需花费的时间
- Snapshot state count：不同状态的 snapshot 的个数
- 99.99% Snapshot size：99.99% 的 snapshot 的大小
- 99.99% Snapshot KV count：99.99% 的 snapshot 包含的 key 的个数

14.8.5.1.22 Task

- Worker handled tasks：worker 每秒钟处理的任务的数量
- Worker pending tasks：当前 worker 中，每秒钟 pending 和 running 的任务的数量，正常情况下，应该小于 1000
- FuturePool handled tasks：future pool 每秒钟处理的任务的数量
- FuturePool pending tasks：当前 future pool 中，每秒钟 pending 和 running 的任务的数量

14.8.5.1.23 Coprocessor Overview

- Request duration：从收到 coprocessor 请求到处理结束所消耗的总时间
- Total Requests：每种类型的总请求的 ops
- Handle duration：每分钟实际处理 coprocessor 请求所消耗的时间的直方图
- Total Request Errors：Coprocessor 每秒请求错误的数量，正常情况下，短时间内不应该有大量的错误
- Total KV Cursor Operations：各种类型的 KV cursor 操作的总数量的 ops，例如 select、index、analyze_table、analyze_index、checksum_table、checksum_index 等
- KV Cursor Operations：每秒各种类型的 KV cursor 操作的数量，以直方图形式显示
- Total RocksDB Perf Statistics：RocksDB 性能统计数据
- Total Response Size：coprocessor 回应的数据大小

14.8.5.1.24 Coprocessor Detail

- Handle duration：每秒钟实际处理 coprocessor 请求所消耗的时间的直方图
- 95% Handle duration by store：每秒钟中 95% 的情况下，每个 TiKV 实例处理 coprocessor 请求所花费的时间
- Wait duration：coprocessor 每秒钟内请求的等待时间，99.99% 的情况下，应该小于 10s
- 95% Wait duration by store：每秒钟 95% 的情况下，每个 TiKV 实例上 coprocessor 请求的等待时间
- Total DAG Requests：DAG 请求的总数量的 ops
- Total DAG Executors：DAG executor 的总数量的 ops
- Total Ops Details (Table Scan)：coprocessor 中请求为 select 的 scan 过程中每秒钟各种事件发生的次数
- Total Ops Details (Index Scan)：coprocessor 中请求为 index 的 scan 过程中每秒钟各种事件发生的次数
- Total Ops Details by CF (Table Scan)：coprocessor 中对于每个 CF 请求为 select 的 scan 过程中每秒钟各种事件发生的次数
- Total Ops Details by CF (Index Scan)：coprocessor 中对于每个 CF 请求为 index 的 scan 过程中每秒钟各种事件发生的次数

14.8.5.1.25 Threads

- Threads state：TiKV 线程的状态
- Threads IO：TiKV 各个线程的 I/O 流量
- Thread Voluntary Context Switches：TiKV 线程自主切换的次数
- Thread Nonvoluntary Context Switches：TiKV 线程被动切换的次数

14.8.5.1.26 RocksDB - kv/raft

- Get operations：get 操作的 ops
- Get duration：get 操作的耗时
- Seek operations：seek 操作的 ops
- Seek duration：seek 操作的耗时
- Write operations：write 操作的 ops
- Write duration：write 操作的耗时
- WAL sync operations：sync WAL 操作的 ops
- Write WAL duration：write 操作中写 WAL 的耗时
- WAL sync duration：sync WAL 操作的耗时
- Compaction operations：compaction 和 flush 操作的 ops
- Compaction duration：compaction 和 flush 操作的耗时
- SST read duration：读取 SST 所需的时间
- Write stall duration：由于 write stall 造成的时间开销，正常情况下应为 0
- Memtable size：每个 CF 的 memtable 的大小
- Memtable hit：memtable 的命中率
- Block cache size：block cache 的大小。如果将 `shared block cache` 禁用，即为每个 CF 的 block cache 的大小
- Block cache hit：block cache 的命中率
- Block cache flow：不同 block cache 操作的流量
- Block cache operations 不同 block cache 操作的个数
- Keys flow：不同操作造成的 key 的流量
- Total keys：每个 CF 中 key 的个数
- Read flow：不同读操作的流量
- Bytes/Read：每次读的大小
- Write flow：不同写操作的流量
- Bytes/Write：每次写的大小
- Compaction flow：compaction 相关的流量
- Compaction pending bytes：等待 compaction 的大小
- Read amplification：每个 TiKV 实例的读放大
- Compression ratio：每一层的压缩比
- Number of snapshots：每个 TiKV 的 snapshot 的数量
- Oldest snapshots duration：最旧的 snapshot 保留的时间
- Number files at each level：每一层的文件个数
- Ingest SST duration seconds：ingest SST 所花费的时间
- Stall conditions changed of each CF：每个 CF stall 的原因

14.8.5.1.27 Raft Engine

- Operations
  - write：Raft Engine 每秒写操作的次数
  - read_entry：Raft Engine 每秒读 raft 日志的次数
  - read_message：Raft Engine 每秒读 raft 元数据的次数
- Write duration：Raft Engine 写操作的耗时，该耗时基本接近写入这些数据所包含的磁盘 IO 的 latency 之和
- Flow
  - write：Raft Engine 写流量
  - rewrite append：重写 append 日志的流量
  - rewrite rewrite：重写 rewrite 日志的流量
- Write Duration Breakdown (99%)
  - wal：写 Raft Engine WAL 的延迟
  - wait：写入前等待时间
  - apply：apply 到内存的时间
- Bytes/Written 每次写入对应的 bytes
- WAL Duration Breakdown (P99%)：写 WAL 内部各个阶段所花的时间
- File Count
  - append：Raft Engine 用于 append 数据的文件个数
  - rewrite：Raft Engine 用于 rewrite 的文件个数（rewrite 类似于 RocksDB 的 compaction）
- Entry Count
  - rewrite：Raft Engine 中已经 rewrite 的记录条数
  - append：Raft Engine 中已经 append 的记录条数

14.8.5.1.28 Titan - All

- Blob file count：Titan blob 文件的数量
- Blob file size：Titan blob 文件总大小
- Live blob size：有效 blob record 的总大小
- Blob cache hit：Titan 的 blob cache 命中率
- Iter touched blob file count：单个 Iterator 所涉及到 blob 文件的数量
- Blob file discardable ratio distribution：blob 文件的失效 blob record 比例的分布情况
- Blob key size：Titan 中 blob key 的大小
- Blob value size：Titan 中 blob value 的大小
- Blob get operations：blob 的 get 操作的数量
- Blob get duration：blob 的 get 操作的耗时
- Blob iter operations：blob 的 iter 操作的耗时
- Blob seek duration：blob 的 seek 操作的耗时
- Blob next duration：blob 的 next 操作的耗时
- Blob prev duration：blob 的 prev 操作的耗时
- Blob keys flow：Titan blob 读写的 key 数量
- Blob bytes flow：Titan blob 读写的 bytes 数量
- Blob file read duration：blob 文件的读取耗时

- Blob file write duration：blob 文件的写入耗时
- Blob file sync operations：blob 文件 sync 次数
- Blob file sync duration：blob 文件 sync 耗时
- Blob GC action：Titan GC 细分动作的次数
- Blob GC duration：Titan GC 的耗时
- Blob GC keys flow：Titan GC 读写的 key 数量
- Blob GC bytes flow：Titan GC 读写的 bytes 数量
- Blob GC input file size：Titan GC 输入文件的大小
- Blob GC output file size：Titan GC 输出文件的大小
- Blob GC file count：Titan GC 涉及的 blob 文件数量

14.8.5.1.29 Pessimistic Locking

- Lock Manager Thread CPU：lock manager 的线程 CPU 使用率
- Lock Manager Handled tasks：lock manager 处理的任务数量
- Waiter lifetime duration：事务等待锁释放的时间
- Wait table：wait table 的状态信息，包括锁的数量和等锁事务的数量
- Deadlock detect duration：处理死锁检测请求的耗时
- Detect error：死锁检测遇到的错误数量，包含死锁的数量
- Deadlock detector leader：死锁检测器 leader 所在节点的信息
- Total pessimistic locks memory size：内存悲观锁占用内存的总大小
- In-memory pessimistic locking result：将悲观锁仅保存到内存的结果，其中 full 表示因为超过内存限制而无法将悲观锁保存至内存的次数

14.8.5.1.30 Memory

- Allocator Stats：内存分配器的统计信息

14.8.5.1.31 Backup

- Backup CPU：backup 的线程 CPU 使用率
- Range Size：backup range 的大小直方图
- Backup Duration：backup 的耗时
- Backup Flow：backup 总的字节大小
- Disk Throughput：实例磁盘的吞吐量
- Backup Range Duration：backup range 的耗时
- Backup Errors：backup 中发生的错误数量

14.8.5.1.32 Encryption

- Encryption data keys：正在使用的加密 data key 的总数量
- Encrypted files：被加密的文件数量
- Encryption initialized：显示加密是否被启用，1 代表已经启用

- Encryption meta files size：加密相关的元数据文件的大小
- Encrypt/decrypt data nanos：每次加密/解密数据的耗时的直方图
- Read/write encryption meta duration：每秒钟读写加密文件所耗费的时间

14.8.5.1.33 面板常见参数的解释

gRPC 消息类型

1. 使用事务型接口的命令：

    - kv_get：事务型的 get 命令，获取指定 ts 能读到的最新版本数据
    - kv_scan：扫描连续的一段数据
    - kv_prewrite：2PC 的第一阶段，预写入事务要提交的数据
    - kv_pessimistic_lock：对 key 加悲观锁，防止其他事务修改
    - kv_pessimistic_rollback：删除 key 上的悲观锁
    - kv_txn_heart_beat：更新悲观事务或大事务的 `lock_ttl` 以防止其被回滚
    - kv_check_txn_status：检查事务的状态
    - kv_commit：2PC 的第二阶段，提交 prewrite 阶段写入的数据
    - kv_cleanup：回滚一个事务（此命令将会在 4.0 中废除）
    - kv_batch_get：与 `kv_get` 类似，一次性获取批量 key 的 value
    - kv_batch_rollback：批量回滚多个预写的事务
    - kv_scan_lock：扫描所有版本号在 `max_version` 之前的锁，用于清理过期的事务
    - kv_resolve_lock：根据事务状态，提交或回滚事务的锁
    - kv_gc：触发垃圾回收
    - kv_delete_range：从 TiKV 中删除连续的一段数据

2. 非事务型的裸命令：

    - raw_get：获取 key 所对应的 value
    - raw_batch_get：获取一批 key 所对应的 value
    - raw_scan：扫描一段连续的数据
    - raw_batch_scan：扫描多段连续的数据
    - raw_put：写入一个 key/value 对
    - raw_batch_put：直接写入一批 key/value 对
    - raw_delete：删除一个 key/value 对
    - raw_batch_delete：删除一批 key/value 对
    - raw_delete_range：删除连续的一段区间

14.8.5.2 TiKV-FastTune 面板

当 TiKV 出现 QPS 抖动、延迟抖动、延迟增加趋势等性能问题时，你可以查看 TiKV-FastTune 面板。TiKV-FastTune 包括多组子面板，可帮助你诊断性能问题，尤其适用于集群中写入负载较大的场景。

当出现写入相关的性能问题时，可以先在 Grafana 中查看 TiDB 相关的面板。如果问题出在存储端，打开 TiKV-FastTune 面板，浏览并检查上面的每个指标。

在 TiKV-FastTune 的面板中，指标标题描述了性能问题的可能成因。要验证成因是否正确，你需要检查具体的图表曲线。

左边 Y 轴表示存储端的 write-RPC QPS，右边 Y 轴上的一组图是倒置绘制的。如果左边 Y 轴的曲线形状与右边的形状匹配，则指标标题描述的问题成因是正确的。

有关该面板的具体监控项以及解释，参考 TiKV-FastTune 用户手册（英文）。

14.8.6 TiFlash 集群监控

使用 TiUP 部署 TiDB 集群时，一键部署监控系统 (Prometheus & Grafana)，监控架构参见TiDB 监控框架概述。

目前 Grafana Dashboard 整体分为 PD、 TiDB、 TiKV、 Node_exporter、 Overview 等。

TiFlash 面板一共包括 TiFlash-Summary、 TiFlash-Proxy-Summary、 TiFlash-Proxy-Details。通过面板上的指标，可以了解 TiFlash 当前的状态。其中 TiFlash-Proxy-Summary、 TiFlash-Proxy-Details 主要为 TiFlash 的 Raft 层信息，其监控指标信息可参考TiKV 监控指标详解。

> 注意：
>
> 低版本的 TiFlash 监控信息较不完善，如有需要推荐使用 v4.0.5 或更高版本的 TiDB 集群。

以下为 TiFlash-Summary 默认的监控信息：

14.8.6.1 Server

- Store size：每个 TiFlash 实例的使用的存储空间的大小。
- Available size：每个 TiFlash 实例的可用的存储空间的大小。
- Capacity size：每个 TiFlash 实例的存储容量的大小。
- Uptime：自上次重启以来 TiFlash 正常运行的时间。
- Memory：每个 TiFlash 实例内存的使用情况。
- CPU Usage：每个 TiFlash 实例 CPU 的使用率。
- FSync OPS：每个 TiFlash 实例每秒进行 fsync 操作的次数。
- File Open OPS：每个 TiFlash 实例每秒进行 open 操作的次数。
- Opened File Count：当前每个 TiFlash 实例打开的文件句柄数。

> 注意：
>
> Store size、FSync OPS、File Open OPS、Opened File Count 目前仅包含了 TiFlash 存储层的统计指标，未包括 TiFlash-Proxy 内的信息。

14.8.6.2 Coprocessor

- Request QPS：所有 TiFlash 实例收到的 coprocessor 请求数量。其中 batch 是 batch 请求数量，batch_cop 是 batch 请求中的 coprocessor 请求数量，cop 是直接通过 coprocessor 接口发送的 coprocessor 请求数量，cop_dag 是所有 coprocessor 请求中 dag 请求数量，super_batch 是开启 super batch 特性的请求数量。

- Executor QPS：所有 TiFlash 实例收到的请求中，每种 dag 算子的数量，其中 table_scan 是扫表算子，selection 是过滤算子，aggregation 是聚合算子，top_n 是 TopN 算子，limit 是 limit 算子。
- Request Duration：所有 TiFlash 实例处理 coprocessor request 总时间，总时间为接收到该 coprocessor 请求至请求应答完毕的时间。
- Error QPS：所有 TiFlash 实例处理 coprocessor 请求的错误数量。其中 meet_lock 为读取的数据有锁，region_not_found 为 Region 不存在，epoch_not_match 为读取的 Region epoch 与本地不一致，kv_client_error 为与 TiKV 通信产生的错误，internal_error 为 TiFlash 内部系统错误，other 为其他错误。
- Request Handle Duration：所有 TiFlash 实例处理 coprocessor 请求处理时间，处理时间为该 coprocessor 请求开始执行到执行结束的时间。
- Response Bytes/Seconds：所有 TiFlash 实例应答总字节数。
- Cop task memory usage：所有 TiFlash 实例处理 coprocessor 请求占用的总内存。
- Handling Request Number：所有 TiFlash 实例正在处理的 coprocessor 请求数量之和。请求的分类与 Request QPS 中的分类相同。
- Threads of RPC：每个 TiFlash 实例使用的实时 RPC 线程数。
- Max Threads of RPC：最近一段时间每个 TiFlash 实例使用的 RPC 线程数峰值。
- Threads：每个 TiFlash 实例使用的实时线程数。
- Max Threads：最近一段时间每个 TiFlash 实例使用的线程数峰值。

14.8.6.3 Task Scheduler

- Min TSO：每个 TiFlash 实例上正在运行的查询语句中的最小 TSO，该值确保具有最小 TSO 的查询可以被调度。如果当前没有正在运行的查询，则该值为 `uint64` 整数型最大值。
- Estimated Thread Usage and Limit：每个 TiFlash 实例上正在运行的所有任务占用的线程估值，以及该实例上任务调度器设置的估算线程用量的软限制和硬限制。
- Active and Waiting Queries Count：每个 TiFlash 实例上正在运行的查询数量和正在等待的查询数量。
- Active and Waiting Tasks Count：每个 TiFlash 实例上正在运行的任务数量和正在等待的任务数量。
- Hard Limit Exceeded Count：每个 TiFlash 实例上运行中任务的估算线程用量超过了设置的硬限制的次数。
- Task Waiting Duration：每个 TiFlash 实例上任务从初始化到被调度的等待时长。

14.8.6.4 DDL

- Schema Version：每个 TiFlash 实例目前缓存的 schema 版本。
- Schema Apply OPM：所有 TiFlash 实例每分钟 apply 同步 TiDB schema diff 的次数。diff apply 是正常的单次 apply 过程，如果 diff apply 失败，则 failed apply +1，并回退到 full apply，拉取最新的 schema 信息以更新 TiFlash 的 schema 版本。
- Schema Internal DDL OPM：所有 TiFlash 实例每分钟执行的内部 DDL 次数。
- Schema Apply Duration：所有 TiFlash 实例 apply schema 消耗的时间。

14.8.6.5 Storage

- Write Command OPS：所有 TiFlash 实例存储层每秒收到的写请求数量。
- Write Amplification：每个 TiFlash 实例写放大倍数（实际磁盘写入量/逻辑数据写入量）。total 为自此次启动以来的写放大倍数，5min 为最近 5 分钟内的写放大倍数。
- Read Tasks OPS：每个 TiFlash 实例每秒存储层内部读取任务的数量。

- Rough Set Filter Rate：每个 TiFlash 实例最近 1 分钟内读取的 packet 数被存储层粗糙索引过滤的比例。
- Internal Tasks OPS：所有 TiFlash 实例每秒进行内部数据整理任务的次数。
- Internal Tasks Duration：所有 TiFlash 实例进行内部数据整理任务消耗的时间。
- Page GC Tasks OPM：所有 TiFlash 实例每分钟进行 Delta 部分数据整理任务的次数。
- Page GC Tasks Duration：所有 TiFlash 实例进行 Delta 部分数据整理任务消耗的时间分布。
- Disk Write OPS：所有 TiFlash 实例每秒进行磁盘写入的次数。
- Disk Read OPS：所有 TiFlash 实例每秒进行磁盘读取的次数。
- Write flow：所有 TiFlash 实例磁盘写操作的流量。
- Read flow：所有 TiFlash 实例磁盘读操作的流量。

> 注意：
>
> 目前这部分监控指标仅包含了 TiFlash 存储层的统计指标，未包括 TiFlash-Proxy 内的信息。

14.8.6.6 Storage Write Stall

- Write & Delta Management Throughput：所有实例写入及数据整理的吞吐量。
  - `throughput_write` 表示通过 Raft 进行数据同步的吞吐量。
  - `throughput_delta-management` 表示数据整理的吞吐量。
  - `total_write` 表示自上次启动以来的总写入字节数。
  - `total_delta-management` 表示自上次启动以来数据整理的总字节数。
- Write Stall Duration：每个实例写入和移除 Region 数据产生的卡顿时长。
- Write Throughput By Instance：每个实例写入数据的吞吐量，包括 apply Raft 数据日志以及 Raft 快照的写入吞吐量。
- Write Command OPS By Instance：每个实例收到各种命令的总计数。
  - `write block` 表示通过 Raft 同步数据日志。
  - `delete_range` 表示从该实例中删除一些 Region 或移动一些 Region 到该实例中。
  - `ingest` 表示这些 Region 的快照被应用到这个实例中。

14.8.6.7 Raft

- Read Index OPS：每个 TiFlash 实例每秒触发 read_index 请求的次数，等于请求触发的 Region 总数。
- Read Index Duration：所有 TiFlash 实例在进行 read_index 消耗的时间，主要消耗在于和 Region leader 的交互和重试时间。
- Wait Index Duration：所有 TiFlash 实例在进行 wait_index 消耗的时间，即拿到 read_index 请求后，等待本地的 Region index >= read_index 所花费的时间。

14.8.7 TiCDC 详细监控指标

使用 TiUP 部署 TiDB 集群时，一键部署的监控系统面板包含 TiCDC 面板。本文档对 TiCDC 监控面板上的各项指标进行详细说明。在日常运维中，运维人员可通过观察 TiCDC 面板上的指标了解 TiCDC 当前的状态。

本文档的对指标的介绍基于以下同步任务，即使用默认配置同步数据到 MySQL。

```
cdc cli changefeed create --server=http://10.0.10.25:8300 --sink-uri="mysql://root:123456@127
    ↪ .0.0.1:3306/" --changefeed-id="simple-replication-task"
```

下图显示了 TiCDC Dashboard 各监控面板：



图 349: TiCDC Dashboard - Overview

各监控面板说明如下：

- Server：TiDB 集群中 TiKV 节点和 TiCDC 节点的概要信息
- Changefeed：TiCDC 同步任务的详细信息
- Events：TiCDC 内部数据流转的详细信息
- TiKV：TiKV 中和 TiCDC 相关的详细信息

14.8.7.1 Server 面板

Server 面板示例如下：



图 350: TiCDC Dashboard - Server metrics

Server 面板的各指标说明如下：

- Uptime：TiKV 节点和 TiCDC 节点已经运行的时间
- Goroutine count：TiCDC 节点 Goroutine 的个数
- Open FD count：TiCDC 节点打开的文件句柄个数

- Ownership：TiCDC 集群中节点的当前状态
- Ownership history：TiCDC 集群中 Owner 节点的历史记录
- CPU usage：TiCDC 节点使用的 CPU
- Memory usage：TiCDC 节点使用的内存

14.8.7.2 Changefeed 面板

Changefeed 面板示例如下：



图 351: TiCDC Dashboard - Changefeed metrics 1

- Changefeed table count：一个同步任务中分配到各个 TiCDC 节点同步的数据表个数
- Processor resolved ts：TiCDC 节点内部状态中已同步的时间点
- Table resolved ts：同步任务中各数据表的同步进度
- Changefeed checkpoint：同步任务同步到下游的进度，正常情况下绿柱应和黄线相接
- PD etcd requests/s：TiCDC 节点每秒向 PD 读写数据的次数
- Exit error count/m：每分钟内导致同步中断的错误发生次数
- Changefeed checkpoint lag：同步任务上下游数据的进度差（以时间计算）
- Processor resolved ts lag：TiCDC 节点内部同步状态与上游的进度差（以时间计算）



图 352: TiCDC Dashboard - Changefeed metrics 2

- Sink write duration：TiCDC 将一个事务的更改写到下游的耗时直方图
- Sink write duration percentile：每秒钟中 95%、99% 和 99.9% 的情况下，TiCDC 将一个事务的更改写到下游所花费的时间
- Flush sink duration：TiCDC 异步刷写数据入下游的耗时直方图
- Flush sink duration percentile：每秒钟中 95%、99% 和 99.9% 的情况下，TiCDC 异步刷写数据入下游所花费的时间

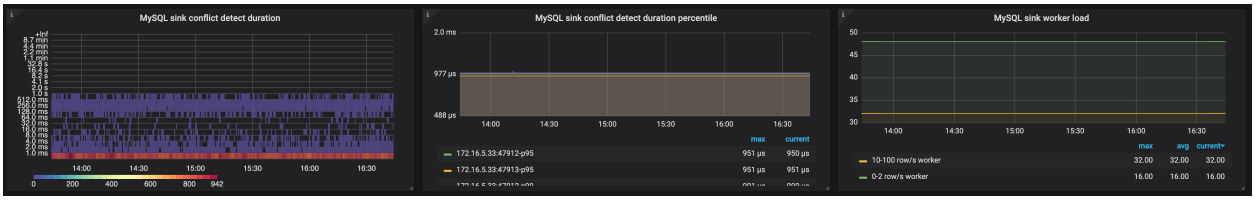


图 353: TiCDC Dashboard - Changefeed metrics 3

- MySQL sink conflict detect duration：MySQL 写入冲突检测耗时直方图
- MySQL sink conflict detect duration percentile：每秒钟中 95%、99% 和 99.9% 的情况下，MySQL 写入冲突检测耗时
- MySQL sink worker load：TiCDC 节点中写 MySQL 线程的负载情况

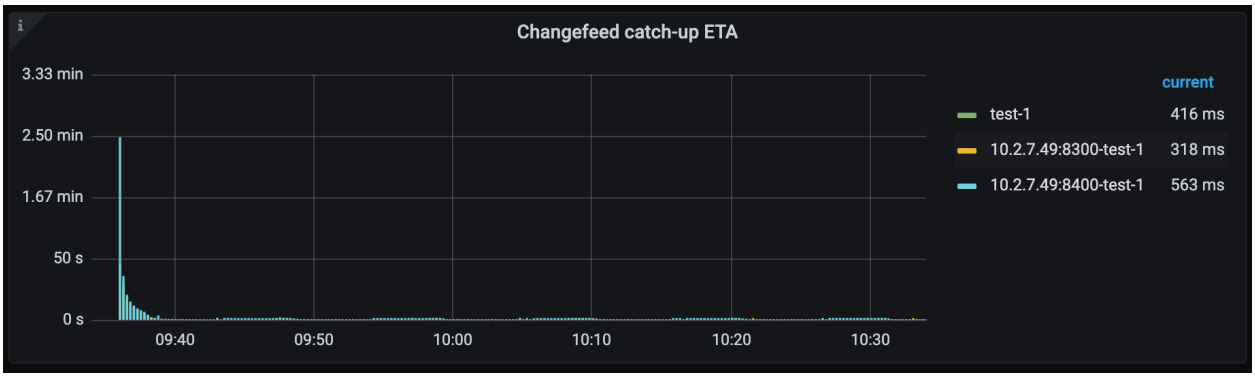


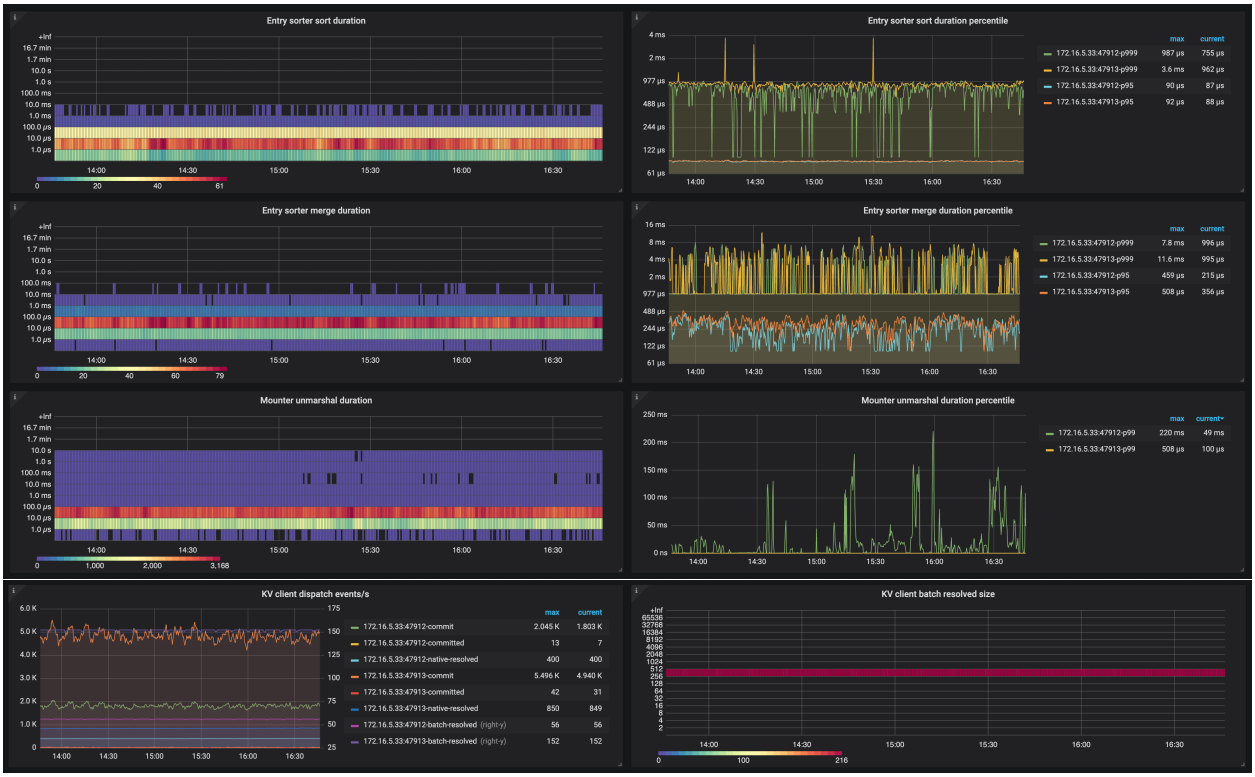
图 354: TiCDC Dashboard - Changefeed metrics 4

- Changefeed catch-up ETA：同步完上游写入的数据所需时间的估计值。当上游的写入速度大于 TiCDC 同步速度时，该值可能会异常的大。(由于 TiCDC 的同步速度受到较多因素制约，因此该值仅供参考，不能完全代表实际所需的同步时间。)

14.8.7.3 Events 面板

Events 面板示例如下：





Events 面板的各指标说明如下：

- Eventfeed count：TiCDC 节点中 Eventfeed RPC 的个数
- Event size percentile：每秒钟中 95% 和 99.9% 的情况下，TiCDC 收到的来自 TiKV 的数据变更消息大小
- Eventfeed error/m：TiCDC 节点中每分钟 Eventfeed RPC 遇到的错误个数
- KV client receive events/s：TiCDC 节点中 KV client 模块每秒收到来自 TiKV 的数据变更个数
- Puller receive events/s：TiCDC 节点中 Puller 模块每秒收到来自 KV client 模块的数据变更个数
- Puller output events/s：TiCDC 节点中 Puller 模块每秒输出到 Sorter 模块的数据变更个数
- Sink flush rows/s：TiCDC 节点每秒写到下游的数据变更的个数
- Puller buffer size：TiCDC 节点中缓存在 Puller 模块中的数据变更个数
- Entry sorter buffer size：TiCDC 节点中缓存在 Sorter 模块中的数据变更个数
- Processor/Mounter buffer size：TiCDC 节点中缓存在 Processor 模块和 Mounter 模块中的数据变更个数
- Sink row buffer size：TiCDC 节点中缓存在 Sink 模块中的数据变更个数
- Entry sorter sort duration：TiCDC 节点对数据变更进行排序的耗时直方图
- Entry sorter sort duration percentile：每秒钟中 95%，99% 和 99.9% 的情况下，TiCDC 排序数据变更所花费的时间
- Entry sorter merge duration：TiCDC 节点合并排序后的数据变更的耗时直方图
- Entry sorter merge duration percentile：每秒钟中 95%，99% 和 99.9% 的情况下，TiCDC 合并排序后的数据变更所花费的时间
- Mounter unmarshal duration：TiCDC 节点解码数据变更的耗时直方图
- Mounter unmarshal duration percentile：每秒钟中 95%，99% 和 99.9% 的情况下，TiCDC 解码数据变更所花费的时间
- KV client dispatch events/s：TiCDC 节点内部 KV client 模块每秒分发数据变更的个数
- KV client batch resolved size：TiKV 批量发给 TiCDC 的 resolved ts 消息的大小

14.8.7.4 TiKV 面板

TiKV 面板示例如下：



TiKV 面板的各指标说明如下：

- CDC endpoint CPU：TiKV 节点上 CDC endpoint 线程使用的 CPU
- CDC worker CPU：TiKV 节点上 CDC worker 线程使用的 CPU
- Min resolved ts：TiKV 节点上最小的 resolved ts
- Min resolved region：TiKV 节点上最小的 resolved ts 的 Region ID
- Resolved ts lag duration percentile：TiKV 节点上最小的 resolved ts 与当前时间的差距
- Initial scan duration：TiKV 节点与 TiCDC 建立链接时增量扫的耗时直方图
- Initial scan duration percentile：每秒钟中 95%、99% 和 99.9% 的情况下，TiKV 节点增量扫的耗时
- Memory without block cache：TiKV 节点在减去 RocksDB block cache 后使用的内存
- CDC pending bytes in memory：TiKV 节点中 CDC 模块使用的内存
- Captured region count：TiKV 节点上捕获数据变更的 Region 个数

14.8.8 资源管控 (Resource Control) 监控指标详解

使用 TiUP 部署 TiDB 集群时，可以一键部署监控系统 (Prometheus & Grafana)。监控架构请参见TiDB 监控框架概述。

目前 Grafana Dashboard 整体分为 PD、 TiDB、 TiKV、 Node_exporter、 Overview、 Performance_overview 等。

如果你的集群配置了Resource Control ，通过观察 Resource Control 面板上的 Metrics，你可以了解当前集群整体的资源消耗状态。

TiDB 使用令牌桶算法 做流控，正如资源管控实现机制 (RFC: Global Resource Control in TiDB) 中所描述：一个 TiDB 节点可能存在多个 Resource Group（资源组），将在 PD 端的 GAC（Global Admission Control）进行流控。每个 TiDB 节点中的本地令牌桶（Local Token Buckets）将定期（默认 5 秒）与 PD 端的 GAC 进行通信，以重新配置本地令牌。其中的本地令牌桶（Local Token Buckets）具体实现为 Resource Controller Client。

以下为 Resource Control 关键监控指标的说明。

14.8.8.1 Request Unit 相关指标

- RU：以 Resource Group（资源组）为单位进行实时统计的Request Unit (RU) 消耗信息。`total` 为当前所有 Resource Group 消耗的 Request Unit 之和。每个 Resource Group 的 Request Unit 消耗等于其读消耗 (Read Request Unit) 和写消耗 (Write Request Unit) 之和。
- RU Per Query：平均每个 SQL 语句消耗的 Request Unit 数量。计算方法是将前述 Request Unit 监控指标除以当前每秒执行的 SQL 语句数量。
- RRU：以 Resource Group 为单位进行实时统计的读请求 Read Request Unit 消耗信息。`total` 为当前所有 Resource Group 消耗的 Read Request Unit 之和。
- RRU Per Query：平均每个 SQL 语句消耗的 Read Request Unit 数量。计算方法是将前述 Read Request Unit 监控指标除以当前每秒执行的 SQL 语句数量。
- WRU：以 Resource Group 为单位进行实时统计的写请求 Write Request Unit 消耗信息。`total` 为当前所有 Resource Group 消耗的 Write Request Unit 之和。
- WRU Per Query：平均每个 SQL 语句消耗的 Write Request Unit 数量。计算方法是将前述 Write Request Unit 监控指标除以当前每秒执行的 SQL 语句数量。

14.8.8.2 Resource 相关指标

- KV Request Count：以 Resource Group（资源组）为单位进行实时统计的 KV 请求数量，区分了读和写两种类型。`total` 为当前所有 Resource Group 涉及的 KV 请求数量之和。
- KV Request Count Per Query：平均每个 SQL 语句涉及的读写 KV 请求数量。计算方法是将前述 KV Request Count 监控指标除以当前每秒执行的 SQL 语句数量。
- Bytes Read：以 Resource Group 为单位进行实时统计的读取数据量。`total` 为当前所有 Resource Group 读取数据量之和。
- Bytes Read Per Query：平均每个 SQL 语句的读取数据量。将前述 Bytes Read 监控指标除以当前每秒执行的 SQL 语句数量。
- Bytes Written：以 Resource Group 为单位进行实时统计的写入数据量。`total` 为当前所有 Resource Group 写入数据量之和。
- Bytes Written Per Query：平均每个 SQL 语句的写入数据量。计算方法是将前述 Bytes Written 监控指标除以当前每秒执行的 SQL 语句数量。
- KV CPU Time：以 Resource Group 为单位进行实时统计的 KV 层 CPU 时间消耗。`total` 为当前所有 Resource Group 消耗 KV 层 CPU 时间之和。
- SQL CPU Time：以 Resource Group 为单位进行实时统计的 SQL 层 CPU 时间消耗。`total` 为当前所有 Resource Group 消耗 SQL 层 CPU 时间之和。

14.8.8.3 Resource Controller Client 相关指标

- Active Resource Groups：实时统计各个 Resource Controller Client 的 Resource Groups 数量。
- Total KV Request Count：以 Resource Group 为单位，实时统计各个 Resource Controller Client 的 KV 请求数量。`total` 为 Resource Controller Client 下 KV 请求数量之和。
- Failed KV Request Count：以 Resource Group 为单位，实时统计各个 Resource Controller Client 的 KV 失败请求数量。`total` 为 Resource Controller Client 下 KV 失败请求数量之和。
- Successful KV Request Count：以 Resource Group 为单位，实时统计各个 Resource Controller Client 的 KV 成功请求数量。`total` 为 Resource Controller Client 下 KV 成功请求数量之和。
- Successful KV Request Wait Duration (99/90)：以 Resource Group 为单位，实时统计各个 Resource Controller Client 成功 KV 请求的等待时间（不同百分位）。
- Token Request Handle Duration (999/99)：以 Resource Group 为单位，实时统计各个 Resource Controller Client 向 Server 端申请 Token 等待的响应时间（不同百分位）。
- Token Request Count：以 Resource Group 为单位，实时统计各个 Resource Controller Client 向 Server 端申请 Token 的次数。`successful` 和 `failed` 分别为 Resource Controller Client 下申请 Token 成功和失败数量之和。

14.9 安全加固

14.9.1 为 TiDB 客户端服务端间通信开启加密传输

TiDB 服务端与客户端之间默认采用非加密连接，因而具备监视信道流量能力的第三方可以知悉 TiDB 服务端与客户端之间发送和接受的数据，包括但不限于查询语句内容、查询结果等。若信道是不可信的，例如客户端是通过公网连接到 TiDB 服务端的，则非加密连接容易造成信息泄露，建议使用加密连接确保安全性。

TiDB 服务端支持启用基于 TLS（传输层安全）协议的加密连接，协议与 MySQL 加密连接一致，现有 MySQL Client 如 MySQL Shell 和 MySQL 驱动等能直接支持。TLS 的前身是 SSL，因而 TLS 有时也被称为 SSL，但由于 SSL 协议有已知安全漏洞，TiDB 实际上并未支持。TiDB 支持的 TLS/SSL 协议版本为 TLSv1.0、TLSv1.1、TLSv1.2 和 TLSv1.3。

使用加密连接后，连接将具有以下安全性质：

- 保密性：流量明文被加密，无法被窃听；
- 完整性：流量明文无法被篡改；
- 身份验证（可选）：客户端和服务端能验证双方身份，避免中间人攻击。

要为 TiDB 客户端与服务端间的通信开启 TLS 加密传输，首先需要在 TiDB 服务端通过配置开启 TLS 加密连接的支持，然后通过配置客户端应用程序使用 TLS 加密连接。一般情况下，如果服务端正确配置了 TLS 加密连接支持，客户端库都会自动启用 TLS 加密传输。

另外，与 MySQL 相同，TiDB 也支持在同一 TCP 端口上开启 TLS 连接或非 TLS 连接。对于开启了 TLS 连接支持的 TiDB 服务端，客户端既可以选择通过加密连接安全地连接到该 TiDB 服务端，也可以选择使用普通的非加密连接。如需使用加密连接，你可以通过以下方式进行配置：

- 通过配置系统变量 `require_secure_transport` 要求所有用户必须使用加密连接来连接到 TiDB。
- 通过在创建用户 (`create user`)，或修改已有用户 (`alter user`) 时指定 `REQUIRE SSL` 要求指定用户必须使用加密连接来连接 TiDB。以创建用户为例：

```
CREATE USER 'u1'@'%' IDENTIFIED BY 'my_random_password' REQUIRE SSL;
```

> 注意：
>
> 如果登录用户已配置使用TiDB 证书鉴权功能校验用户证书，也会隐式要求对应用户必须使用加密连接连接 TiDB。

14.9.1.1 配置 TiDB 服务端启用安全连接

要启用安全连接，请参考以下相关参数说明：

- auto-tls：启用证书自动生成功能（从 v5.2.0 开始）
- ssl-cert：指定 SSL 证书文件路径
- ssl-key：指定证书文件对应的私钥
- ssl-ca：可选，指定受信任的 CA 证书文件路径
- tls-version：可选，指定最低 TLS 版本，例如 TLSv1.2

auto tls 支持安全连接，但不提供客户端证书验证。有关证书验证和控制证书生成方式的说明，请参考下面配置 ssl-cert，ssl-key 和 ssl-ca 变量的建议：

- 在启动 TiDB 时，至少需要在配置文件中同时指定 ssl-cert 和 ssl-key 参数，才能在 TiDB 服务端开启安全连接。还可以指定 ssl-ca 参数进行客户端身份验证（请参见配置启用身份验证章节）。
- 参数指定的文件都为 PEM 格式。另外目前 TiDB 尚不支持加载有密码保护的私钥，因此必须提供一个没有密码的私钥文件。若提供的证书或私钥无效，则 TiDB 服务端将照常启动，但并不支持客户端加密连接到 TiDB 服务端。
- 若证书参数无误，则 TiDB 在启动时将会输出 secure connection is enabled，否则 TiDB 会输出 secure connection is NOT ENABLED。

> 注意：
>
> v5.2.0 版本之前，你可以使用 mysql_ssl_rsa_setup --datadir=./certs 生成证书。mysql_ssl_rsa_setup 工具是 MySQL Server 的一部分。

14.9.1.2 配置 MySQL Client 使用安全连接

MySQL 5.7 及以上版本自带的客户端默认尝试使用安全连接，若服务端不支持安全连接则自动退回到使用非安全连接；MySQL 5.7 以下版本自带的客户端默认采用非安全连接。

可以通过命令行参数修改客户端的连接行为，包括：

- 强制使用安全连接，若服务端不支持安全连接则连接失败 (--ssl-mode=REQUIRED)

- 尝试使用安全连接，若服务端不支持安全连接则退回使用不安全连接
- 使用不安全连接 (--ssl-mode=DISABLED)

除此参数外，MySQL 8.0 客户端有两种 SSL 模式：

- --ssl-mode=VERIFY_CA：根据 --ssl-ca 签发的 CA 验证来自服务器的证书。
- --ssl-mode=VERIFY_IDENTITY：与 VERIFY_CA 相同，但也验证所连接的主机名是否与证书匹配。

详细信息请参阅 MySQL 文档中关于客户端配置安全连接的部分。

14.9.1.3 配置启用身份验证

若在 TiDB 服务端或 MySQL Client 中未指定 ssl-ca 参数，则默认不会进行客户端或服务端身份验证，无法抵御中间人攻击，例如客户端可能会“安全地”连接到了一个伪装的服务端。可以在服务端和客户端中配置 ssl-ca 参数进行身份验证。一般情况下只需验证服务端身份，但也可以验证客户端身份进一步增强安全性。

- 若要使 MySQL Client 验证 TiDB 服务端身份，TiDB 服务端需至少配置 ssl-cert 和 ssl-key 参数，客户端需至少指定 --ssl-ca 参数，且 --ssl-mode 至少为 VERIFY_CA。必须确保 TiDB 服务端配置的证书 (ssl-cert) 是由客户端 --ssl-ca 参数所指定的 CA 所签发的，否则身份验证失败。
- 若要使 TiDB 服务端验证 MySQL Client 身份，TiDB 服务端需配置 ssl-cert、ssl-key、ssl-ca 参数，客户端需至少指定 --ssl-cert、--ssl-key 参数。必须确保服务端配置的证书和客户端配置的证书都是由服务端配置指定的 ssl-ca 签发的。
- 若要进行双向身份验证，请同时满足上述要求。

默认情况，服务端对客户端的身份验证是可选的。若客户端在 TLS 握手时未出示自己的身份证书，也能正常建立 TLS 连接。但也可以通过在创建用户 (create user)，赋予权限 (grant) 或修改已有用户 (alter user) 时指定 require x509 要求客户端需进行身份验证，以创建用户为例：

```
create user 'u1'@'%'  require x509;
```

> 注意：
>
> 如果登录用户已配置使用TiDB 证书鉴权功能校验用户证书，也会隐式要求对应用户需进行身份验证。

14.9.1.4 检查当前连接是否是加密连接

可以通过 SHOW STATUS LIKE "%Ssl%"; 了解当前连接的详细情况，包括是否使用了安全连接、安全连接采用的加密协议、TLS 版本号等。

以下是一个安全连接中执行该语句的结果。由于客户端支持的 TLS 版本号和加密协议会有所不同，执行结果相应地也会有所变化。

```
SHOW STATUS LIKE "%Ssl%";
```

```
......
| Ssl_verify_mode | 5                           |
| Ssl_version     | TLSv1.2                     |
| Ssl_cipher      | ECDHE-RSA-AES128-GCM-SHA256  |
......
```

除此以外，对于 MySQL 自带客户端，还可以使用 STATUS 或 `\s` 语句查看连接情况：

```
\s
```

```
...
SSL: Cipher in use is ECDHE-RSA-AES128-GCM-SHA256
...
```

14.9.1.5 支持的 TLS 版本及密钥交换协议和加密算法

TiDB 支持的 TLS 版本及密钥交换协议和加密算法由 Golang 官方库决定。

你使用的客户端库和操作系统加密策略也可能会影响到支持的协议版本和密码套件。

14.9.1.5.1 支持的 TLS 版本

- TLSv1.0（默认禁用）
- TLSv1.1
- TLSv1.2
- TLSv1.3

可以使用配置项 `tls-version` 来限制 TLS 版本。

实际可用的 TLS 版本取决于操作系统的加密策略、 MySQL 客户端版本和客户端的 SSL/TLS 库。

14.9.1.5.2 支持的密钥交换协议及加密算法

- TLS_RSA_WITH_AES_128_CBC_SHA
- TLS_RSA_WITH_AES_256_CBC_SHA
- TLS_RSA_WITH_AES_128_CBC_SHA256
- TLS_RSA_WITH_AES_128_GCM_SHA256
- TLS_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256

- TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_RSA_WITH_CHACHA20_POLY1305_SHA256
- TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256
- TLS_AES_128_GCM_SHA256
- TLS_AES_256_GCM_SHA384
- TLS_CHACHA20_POLY1305_SHA256

14.9.1.6 重加载证书、密钥和 CA

在需要替换证书、密钥或 CA 时，可以在完成对应文件替换后，对运行中的 TiDB 实例执行`ALTER INSTANCE RELOAD TLS` 语句从原配置的证书 (`ssl-cert`)、密钥 (`ssl-key`) 和 CA (`ssl-ca`) 的路径重新加证书、密钥和 CA，而无需重启 TiDB 实例。

新加载的证书密钥和 CA 将在语句执行成功后对新建立的连接生效，不会影响语句执行前已建立的连接。

14.9.1.7 监控

自 TiDB v5.2.0 版本起，你可以使用 `Ssl_server_not_after` 和 `Ssl_server_not_before` 状态变量监控证书有效期的起止时间。

```
SHOW GLOBAL STATUS LIKE 'Ssl\_server\_not\_%';
```

```
+-----------------------+--------------------------+
| Variable_name         | Value                    |
+-----------------------+--------------------------+
| Ssl_server_not_after  | Nov 28 06:42:32 2021 UTC |
| Ssl_server_not_before | Aug 30 06:42:32 2021 UTC |
+-----------------------+--------------------------+
2 rows in set (0.0076 sec)
```

14.9.1.8 另请参阅

- 为 TiDB 组件间通信开启加密传输

14.9.2 为 TiDB 组件间通信开启加密传输

本部分介绍如何为 TiDB 集群内各组件间开启加密传输。一旦开启，以下组件间均将使用加密传输：

- TiDB、 TiKV、 PD、 TiFlash 之间的通讯
- TiDB Control 与 TiDB，TiKV Control 与 TiKV，PD Control 与 PD
- TiKV、 PD、 TiDB、 TiFlash 各自集群内内部通讯

目前暂不支持只开启其中部分组件的加密传输。

14.9.2.1 配置开启加密传输

1. 准备证书。

   推荐为 TiDB、TiKV、PD 分别准备一个 Server 证书，并保证可以相互验证，而它们的 Control 工具则可选择共用 Client 证书。

   有多种工具可以生成自签名证书，如 `openssl`，`easy-rsa`，`cfssl`。

   这里提供一个使用 `openssl` 生成证书的示例：生成自签名证书。

2. 配置证书。

   - TiDB

     在 `config` 文件或命令行参数中设置：

     ```
     [security]
     # Path of file that contains list of trusted SSL CAs for connection with cluster
         ↪ components.
     cluster-ssl-ca = "/path/to/ca.pem"
     # Path of file that contains X509 certificate in PEM format for connection with cluster
         ↪  components.
     cluster-ssl-cert = "/path/to/tidb-server.pem"
     # Path of file that contains X509 key in PEM format for connection with cluster
         ↪ components.
     cluster-ssl-key = "/path/to/tidb-server-key.pem"
     ```

   - TiKV

     在 `config` 文件或命令行参数中设置，并设置相应的 URL 为 https：

     ```
     [security]
     # set the path for certificates. Empty string means disabling secure connectoins.
     ca-path = "/path/to/ca.pem"
     cert-path = "/path/to/tikv-server.pem"
     key-path = "/path/to/tikv-server-key.pem"
     ```

   - PD

     在 `config` 文件或命令行参数中设置，并设置相应的 URL 为 https：

     ```
     [security]
     # Path of file that contains list of trusted SSL CAs. if set, following four settings
         ↪ shouldn't be empty
     cacert-path = "/path/to/ca.pem"
     # Path of file that contains X509 certificate in PEM format.
     cert-path = "/path/to/pd-server.pem"
     # Path of file that contains X509 key in PEM format.
     key-path = "/path/to/pd-server-key.pem"
     ```

   - TiFlash（从 v4.0.5 版本开始引入）

     在 `tiflash.toml` 文件中设置，将 `http_port` 项改为 `https_port`:

```
[security]
# Path of file that contains list of trusted SSL CAs. if set, following four settings
    ↪ shouldn't be empty
ca_path = "/path/to/ca.pem"
# Path of file that contains X509 certificate in PEM format.
cert_path = "/path/to/tiflash-server.pem"
# Path of file that contains X509 key in PEM format.
key_path = "/path/to/tiflash-server-key.pem"
```

在 tiflash-learner.toml 文件中设置，

```
[security]
# Sets the path for certificates. The empty string means that secure connections are
    ↪ disabled.
ca-path = "/path/to/ca.pem"
cert-path = "/path/to/tiflash-server.pem"
key-path = "/path/to/tiflash-server-key.pem"
```

- TiCDC

  在 config 文件中设置

```
[security]
ca-path = "/path/to/ca.pem"
cert-path = "/path/to/cdc-server.pem"
key-path = "/path/to/cdc-server-key.pem"
```

  或者在启动命令行中设置，并设置相应的 URL 为 https：

```
cdc server --pd=https://127.0.0.1:2379 --log-file=ticdc.log --addr=0.0.0.0:8301 --
    ↪ advertise-addr=127.0.0.1:8301 --ca=/path/to/ca.pem --cert=/path/to/ticdc-cert.
    ↪ pem --key=/path/to/ticdc-key.pem
```

此时 TiDB 集群各个组件间已开启加密传输。

> 注意：
>
> 若 TiDB 集群各个组件间开启加密传输后，在使用 tidb-ctl、 tikv-ctl 或 pd-ctl 工具连接集群时，需要指定 client 证书，示例：

```
./tidb-ctl -u https://127.0.0.1:10080 --ca /path/to/ca.pem --ssl-cert /path/to/client.pem --
    ↪ ssl-key /path/to/client-key.pem
```

```
tiup ctl:v<CLUSTER_VERSION> pd -u https://127.0.0.1:2379 --cacert /path/to/ca.pem --cert /
    ↪ path/to/client.pem --key /path/to/client-key.pem
```

```
./tikv-ctl --host="127.0.0.1:20160" --ca-path="/path/to/ca.pem" --cert-path="/path/to/client
    ↪ .pem" --key-path="/path/to/clinet-key.pem"
```

14.9.2.2 认证组件调用者身份

通常被调用者除了校验调用者提供的密钥、证书和 CA 有效性外，还需要校验调用方身份以防止拥有有效证书的非法访问者进行访问（例如：TiKV 只能被 TiDB 访问，需阻止拥有合法证书但非 TiDB 的其他访问者访问 TiKV）。

如希望进行组件调用者身份认证，需要在生证书时通过 Common Name 标识证书使用者身份，并在被调用者配置检查证书 Common Name 列表来检查调用者身份。

- TiDB

  在 config 文件或命令行参数中设置：

```
[security]
cluster-verify-cn = [
  "TiDB-Server",
  "TiKV-Control",
]
```

- TiKV

  在 config 文件或命令行参数中设置：

```
[security]
cert-allowed-cn = [
    "TiDB-Server", "PD-Server", "TiKV-Control", "RawKvClient1",
]
```

- PD

  在 config 文件或命令行参数中设置：

```
[security]
cert-allowed-cn = ["TiKV-Server", "TiDB-Server", "PD-Control"]
```

- TiFlash（从 v4.0.5 版本开始引入）

  在 tiflash.toml 文件中设置：

```
[security]
cert_allowed_cn = ["TiKV-Server", "TiDB-Server"]
```

  在 tiflash-learner.toml 文件中设置：

```
[security]
cert-allowed-cn = ["PD-Server", "TiKV-Server", "TiFlash-Server"]
```

14.9.2.3 证书重新加载

- 如果 TiDB 集群部署在本地的数据中心，TiDB、PD、TiKV、TiFlash、TiCDC 和各种 client 在每次新建相互通讯的连接时都会重新读取当前的证书和密钥文件内容，实现证书和密钥的重新加载，无需重启 TiDB 集群。
- 如果 TiDB 集群部署在自己管理的 Cloud，TLS 证书的签发需要与云服务商的证书管理服务集成，TiDB、PD、TiKV、TiFlash、TiCDC 组件的 TLS 证书支持自动轮换，无需重启 TiDB 集群。

14.9.2.4 证书有效期

你可以自定义 TiDB 集群中各组件 TLS 证书的有效期。例如，使用 OpenSSL 签发生成 TLS 证书时，可以通过 days 参数设置有效期，详见生成自签名证书。

14.9.2.5 另请参阅

- 为 TiDB 客户端服务端间通信开启加密传输

14.9.3 生成自签名证书

> **注意：**
>
> 要在 TiDB 客户端与服务端间通信开启加密传输，你只需配置 `auto-tls` 参数。

本文档提供使用 `openssl` 生成自签名证书的一个示例，用户也可以根据自己的需求生成符合要求的证书和密钥。

假设实例集群拓扑如下：

Name	Host IP	Services
node1	172.16.10.11	PD1, TiDB1
node2	172.16.10.12	PD2
node3	172.16.10.13	PD3
node4	172.16.10.14	TiKV1
node5	172.16.10.15	TiKV2
node6	172.16.10.16	TiKV3

14.9.3.1 安装 OpenSSL

对于 Debian 或 Ubuntu 操作系统：

```
apt install openssl
```

对于 RedHat 或 CentOS 操作系统：

```
yum install openssl
```

也可以参考 OpenSSL 官方的下载文档进行安装。

14.9.3.2 生成 CA 证书

CA 的作用是签发证书。实际情况中，请联系你的管理员签发证书或者使用信任的 CA 机构。CA 会管理多个证书对，这里只需生成原始的一对证书，步骤如下：

1. 生成 root 密钥：

```
openssl genrsa -out root.key 4096
```

2. 生成 root 证书：

```
openssl req -new -x509 -days 1000 -key root.key -out root.crt
```

3. 验证 root 证书：

```
openssl x509 -text -in root.crt -noout
```

14.9.3.3 签发各个组件的证书

14.9.3.3.1 集群中可能使用到的证书

- tidb certificate 由 TiDB 使用，为其他组件和客户端验证 TiDB 身份。
- tikv certificate 由 TiKV 使用，为其他组件和客户端验证 TiKV 身份。
- pd certificate 由 PD 使用，为其他组件和客户端验证 PD 身份。
- client certificate 用于 PD、TiKV、TiDB 验证客户端。例如 `pd-ctl`，`tikv-ctl` 等。

14.9.3.3.2 给 TiKV 实例签发证书

给 TiKV 实例签发证书的步骤如下：

1. 生成该证书对应的私钥：

```
openssl genrsa -out tikv.key 2048
```

2. 拷贝一份 OpenSSL 的配置模板文件。

   模板文件可能存在多个位置，请以实际位置为准：

```
cp /usr/lib/ssl/openssl.cnf .
```

   如果不知道实际位置，请在根目录下查找：

```
find / -name openssl.cnf
```

3. 编辑 `openssl.cnf`，在 `[ req ]` 字段下加入 `req_extensions = v3_req`，然后在 `[ v3_req ]` 字段下加入 `subjectAltName = @alt_names`。最后新建一个字段，并编辑 SAN 的信息：

```
[ alt_names ]
IP.1 = 127.0.0.1
IP.2 = 172.16.10.14
IP.3 = 172.16.10.15
IP.4 = 172.16.10.16
```

4. 保存 openssl.cnf 文件后，生成证书请求文件（在这一步也可以为该证书指定 Common Name，其作用是让服务端验证接入的客户端的身份，各个组件默认不会开启验证，需要在配置文件中启用该功能才生效）：

```
openssl req -new -key tikv.key -out tikv.csr -config openssl.cnf
```

5. 签发生成证书：

```
openssl x509 -req -days 365 -CA root.crt -CAkey root.key -CAcreateserial -in tikv.csr -out
    ↪ tikv.crt -extensions v3_req -extfile openssl.cnf
```

6. 验证证书携带 SAN 字段信息（可选）：

```
openssl x509 -text -in tikv.crt -noout
```

7. 确认在当前目录下得到如下文件：

```
root.crt
tikv.crt
tikv.key
```

为其它 TiDB 组件签发证书的过程类似，此文档不再赘述。

14.9.3.3.3 为客户端签发证书

为客户端签发证书的步骤如下。

1. 生成该证书对应的私钥：

```
openssl genrsa -out client.key 2048
```

2. 生成证书请求文件（在这一步也可以为该证书指定 Common Name，其作用是让服务端验证接入的客户端的身份，默认不会开启对各个组件的验证，需要在配置文件中启用该功能才生效）

```
openssl req -new -key client.key -out client.csr
```

3. 签发生成证书：

```
openssl x509 -req -days 365 -CA root.crt -CAkey root.key -CAcreateserial -in client.csr -out
    ↪  client.crt
```

14.9.4 静态加密

> 注意：
>
> 如果集群部署在 AWS 上并在使用 EBS (Elastic Block Store) 存储数据，建议使用 EBS 加密，详细信息请参考 AWS 文档 - EBS 加密。如果集群部署在 AWS 上，但未使用 EBS 存储（例如使用本地 NVMe 存储），则建议使用本文中介绍的静态加密方式。

静态加密 (encryption at rest) 即在存储数据时进行数据加密。对于数据库，静态加密功能也叫透明数据加密 (TDE)，区别于传输数据加密 (TLS) 或使用数据加密（很少使用）。SSD 驱动器、文件系统、云供应商等都可进行静态加密。但区别于这些加密方式，若 TiKV 在存储数据前就进行数据加密，攻击者则必须通过数据库的身份验证才能访问数据。例如，即使攻击者获得物理机的访问权限时，也无法通过复制磁盘上的文件来访问数据。

14.9.4.1 各 TiDB 组件支持的加密方式

在一个 TiDB 集群中，不同的组件使用不同的加密方式。本节介绍 TiKV、TiFlash、PD 和 Backup & Restore (BR) 等不同 TiDB 组件支持的加密方式。

部署好一个 TiDB 集群后，大部分用户数据会存储在 TiKV 和 TiFlash 节点上。此外，一些元数据会存储在 PD 节点上，例如用作为 TiKV Region 边界的二级索引密钥。为了能够充分发挥静态加密的优势，所有组件都需要启用加密功能。此外，进行加密时，也需要对备份、日志文件和通过网络传输的数据进行加密。

14.9.4.1.1 TiKV

TiKV 支持静态加密，即在 CTR 模式下使用 AES 或 SM4 对数据文件进行透明加密。要启用静态加密，用户需提供一个加密密钥，即主密钥。TiKV 自动轮换 (rotate) 用于加密实际数据文件的密钥，主密钥则可以由用户手动轮换。请注意，静态加密仅加密静态数据（即磁盘上的数据），而不加密网络传输中的数据。建议 TLS 与静态加密一起使用。

可以选择将 AWS KMS (Key Management Service) 用于云上部署或本地部署，也可以指定将密钥以明文形式存储在文件中。

TiKV 当前不从核心转储 (core dumps) 中排除加密密钥和用户数据。建议在使用静态加密时禁用 TiKV 进程的核心转储，该功能目前无法由 TiKV 独立处理。

TiKV 使用文件的绝对路径来跟踪已加密的数据文件。一旦 TiKV 节点开启了加密功能，用户就不应更改数据文件的路径配置，例如 `storage.data-dir`，`raftstore.raftdb-path`，`rocksdb.wal-dir` 和 `raftdb.wal-dir`。

SM4 加密只在 v6.3.0 及之后版本的 TiKV 上支持。v6.3.0 之前的 TiKV 仅支持 AES 加密。SM4 加密会对吞吐造成 50% 到 80% 的回退。

14.9.4.1.2 TiFlash

TiFlash 支持静态加密。数据密钥由 TiFlash 生成。TiFlash（包括 TiFlash Proxy）写入的所有文件，包括数据文件、Schema 文件、临时文件等，均由当前数据密钥加密。TiFlash 支持的加密算法、加密配置方法（配置项在`tiflash-learner.toml` 中）和监控项含义等均与 TiKV 一致。

如果 TiFlash 中部署了 Grafana 组件，可以查看 TiFlash-Proxy-Details -> Encryption。

SM4 加密只在 v6.4.0 及之后版本的 TiFlash 上支持。v6.4.0 之前的 TiFlash 仅支持 AES 加密。

14.9.4.1.3 PD

PD 的静态加密为实验特性，其配置方式与 TiKV 相同。

14.9.4.1.4 BR 备份

BR 支持对备份到 S3 的数据进行 S3 服务端加密 (SSE)。BR S3 服务端加密也支持使用用户自行创建的 AWS KMS 密钥进行加密，详细信息请参考BR S3 服务端加密。

14.9.4.1.5 日志

TiKV，TiDB 和 PD 信息日志中可能包含用于调试的用户数据。信息日志不会被加密，建议开启日志脱敏功能。

14.9.4.2 TiKV 静态加密

TiKV 当前支持的加密算法包括 AES128-CTR、AES192-CTR、AES256-CTR 和 SM4-CTR (仅 v6.3.0 及之后版本)。TiKV 使用信封加密 (envelop encryption)，所以启用加密后，TiKV 使用以下两种类型的密钥：

- 主密钥 (master key)：主密钥由用户提供，用于加密 TiKV 生成的数据密钥。用户在 TiKV 外部进行主密钥的管理。
- 数据密钥 (data key)：数据密钥由 TiKV 生成，是实际用于加密的密钥。

多个 TiKV 实例可共用一个主密钥。在生产环境中，推荐通过 AWS KMS 提供主密钥。首先通过 AWS KMS 创建用户主密钥 (CMK)，然后在配置文件中将 CMK 密钥的 ID 提供给 TiKV。TiKV 进程在运行时可以通过 IAM 角色访问 KMS CMK。如果 TiKV 无法访问 KMS CMK，TiKV 就无法启动或重新启动。详情参阅 AWS 文档中的 KMS and IAM。

用户也可以通过文件形式提供主密钥。该文件须包含一个用十六进制字符串编码的 256 位（32 字节）密钥，并以换行符结尾（即 `\n`），且不包含其他任何内容。将密钥存储在磁盘上会泄漏密钥，因此密钥文件仅适合存储在 RAM 内存的 `tempfs` 中。

数据密钥由 TiKV 传递给底层存储引擎（即 RocksDB）。RocksDB 写入的所有文件，包括 SST 文件，WAL 文件和 MANIFEST 文件，均由当前数据密钥加密。TiKV 使用的其他临时文件（可能包括用户数据）也由相同的数据密钥加密。默认情况下，TiKV 每周自动轮换数据密钥，但是该时间段是可配置的。密钥轮换时，TiKV 不会重写全部现有文件来替换密钥，但如果集群的写入量恒定，则 RocksDB compaction 会将使用最新的数据密钥对数据重新加密。TiKV 跟踪密钥和加密方法，并使用密钥信息对读取的内容进行解密。

无论用户配置了哪种数据加密方法，数据密钥都使用 AES256-GCM 算法进行加密，以方便对主密钥进行验证。所以当使用文件而不是 KMS 方式指定主密钥时，主密钥必须为 256 位（32 字节）。

14.9.4.2.1 创建密钥

如需在 AWS 上创建一个密钥，请执行以下步骤：

1. 进入 AWS 控制台的 AWS KMS。
2. 确保在控制台的右上角选择正确的区域。
3. 点击创建密钥，选择对称 (Symmetric) 作为密钥类型。
4. 为钥匙设置一个别名。

你也可以使用 AWS CLI 执行该操作：

```
aws --region us-west-2 kms create-key
aws --region us-west-2 kms create-alias --alias-name "alias/tidb-tde" --target-key-id 0987dcba-09
    ↪ fe-87dc-65ba-ab0987654321
```

需要在第二条命令中输入的 `--target-key-id` 是第一条命令的结果。

14.9.4.2.2 配置加密

要启用加密，你可以在 TiKV 和 PD 的配置文件中添加加密部分：

```
[security.encryption]
data-encryption-method = "aes128-ctr"
data-key-rotation-period = "168h" # 7 days
```

`data-encryption-method` 的可选值为 `"aes128-ctr"`、`"aes192-ctr"`、`"aes256-ctr"`、`"sm4-ctr"` (仅 v6.3.0 及之后版本) 和 `"plaintext"`。默认值为 `"plaintext"`，即默认不开启加密功能。`data-key-rotation-period` 指定 TiKV 轮换密钥的频率。可以为新 TiKV 集群或现有 TiKV 集群开启加密，但只有启用后写入的数据才保证被加密。要禁用加密，请在配置文件中删除 `data-encryption-method`，或将该参数值为 `"plaintext"`，然后重启 TiKV。若要替换加密算法，则将 `data-encryption-method` 替换成已支持的加密算法，然后重启 TiKV。替换加密算法后，旧加密算法生成的加密文件会随着新数据的写入逐渐被重写成新加密算法所生成的加密文件。

如果启用了加密（即 `data-encryption-method` 的值不是 `"plaintext"`），则必须指定主密钥。要使用 AWS KMS 方式指定为主密钥，请在 `[security.encryption]` 部分之后添加 `[security.encryption.master-key]` 部分：

```
[security.encryption.master-key]
type = "kms"
key-id = "0987dcba-09fe-87dc-65ba-ab0987654321"
region = "us-west-2"
endpoint = "https://kms.us-west-2.amazonaws.com"
```

`key-id` 指定 KMS CMK 的密钥 ID。`region` 为 KMS CMK 的 AWS 区域名。`endpoint` 通常无需指定，除非你在使用非 AWS 提供的 AWS KMS 兼容服务或需要使用 KMS VPC endpoint。

你也可以使用 AWS 多区域键。为此，你需要在一个特定的区域设置一个主键，并在需要的区域中添加副本密钥。

若要使用文件方式指定主密钥，主密钥配置应如下所示：

```
[security.encryption.master-key]
type = "file"
path = "/path/to/key/file"
```

以上示例中，`path` 为密钥文件的路径。该文件须包含一个 256 位（32 字节）的十六进制字符串，并以换行符结尾（即 `\n`），且不包含其他任何内容。密钥文件示例如下：

```
3b5896b5be691006e0f71c3040a29495ddcad20b14aff61806940ebd780d3c62
```

14.9.4.2.3 轮换主密钥

要轮换主密钥，你必须在配置中同时指定新的主密钥和旧的主密钥，然后重启 TiKV。用 `security.encryption.master-key` 指定新的主密钥，用 `security.encryption.previous-master-key` 指定旧的主密钥。`security.encryption.previous-master-key` 配置的格式与 `encryption.master-key` 相同。重启时，TiKV 必须能同时访问旧主密钥和新主密钥，但一旦 TiKV 启动并运行，就只需访问新密钥。此后，可以将 `encryption.previous-master-key` 项保留在配置文件中。即使重启时，TiKV 也只会在无法使用新的主密钥解密现有数据时尝试使用旧密钥。

TiKV 当前不支持在线轮换主密钥，因此你需要重启 TiKV 进行主密钥轮换。建议对运行中的、提供在线查询的 TiKV 集群进行滚动重启。

轮换 KMS CMK 的配置示例如下：

```
[security.encryption.master-key]
type = "kms"
key-id = "50a0c603-1c6f-11e6-bb9e-3fadde80ce75"
region = "us-west-2"

[security.encryption.previous-master-key]
type = "kms"
key-id = "0987dcba-09fe-87dc-65ba-ab0987654321"
region = "us-west-2"
```

14.9.4.2.4 监控和调试

要监控静态加密（如果 TiKV 中部署了 Grafana 组件），可以查看 TiKV-Details -> Encryption 面板中的监控项：

- Encryption initialized：如果在 TiKV 启动期间初始化了加密，则为 1，否则为 0。进行主密钥轮换时可通过该监控项确认主密钥轮换是否已完成。
- Encryption data keys：现有数据密钥的数量。每次轮换数据密钥后，该数字都会增加 1。通过此监控指标可以监测数据密钥是否按预期轮换。
- Encrypted files：当前的加密数据文件数量。为先前未加密的集群启用加密时，将此数量与数据目录中的当前数据文件进行比较，可通过此监控指标估计已经被加密的数据量。
- Encryption meta file size：加密元数据文件的大小。
- Read/Write encryption meta duration：对用于加密的元数据进行操作带来的额外开销。

在调试方面，可使用 `tikv-ctl` 命令查看加密元数据（例如使用的加密方法和数据密钥列表）。该操作可能会暴露密钥，因此不推荐在生产环境中使用。详情参阅TiKV Control。

14.9.4.2.5 TiKV 版本间兼容性

为了减少 TiKV 管理加密元数据造成的 I/O 操作和互斥锁竞争引发的开销，TiKV v4.0.9 对此进行了优化。此优化可以通过 TiKV 配置文件中的 `security.encryption.enable-file-dictionary-log` 参数启用或关闭。此配置参数仅在 TiKV v4.0.9 或更高版本中生效。

该配置项默认开启，此时 TiKV v4.0.8 或更早期的版本无法识别加密元数据的数据格式。例如，假设你正在使用 TiKV v4.0.9 或更高版本，开启了静态加密和默认开启了 `enable-file-dictionary-log` 配置。如果将集群降级到 TiKV v4.0.8 或更早版本，TiKV 将无法启动，并且信息日志中会有类似如下的报错：

```
[2020/12/07 07:26:31.106 +08:00] [ERROR] [mod.rs:110] ["encryption: failed to load file
    ↪ dictionary."]
[2020/12/07 07:26:33.598 +08:00] [FATAL] [lib.rs:483] ["called `Result::unwrap()` on an `Err`
    ↪ value: Other(\"[components/encryption/src/encrypted_file/header.rs:18]: unknown version
    ↪ 2\")"]
```

为了避免上面所示错误，你可以首先将 `security.encryption.enable-file-dictionary-log` 设置为 `false`，然后启动 TiKV v4.0.9 或更高版本。TiKV 成功启动后，加密元数据的数据格式将降级为 TiKV 早期版本可以识别的格式。此时，你可再将 TiKV 集群降级到较早的版本。

14.9.4.3 TiFlash 静态加密

TiFlash 当前支持的加密算法与 TiKV 一致，包括 AES128-CTR、AES192-CTR、AES256-CTR 和 SM4-CTR（仅 v6.4.0 及之后版本）。TiFlash 同样使用信封加密 (envelop encryption)，所以启用加密后，TiFlash 使用以下两种类型的密钥：

- 主密钥 (master key)：主密钥由用户提供，用于加密 TiFlash 生成的数据密钥。用户在 TiFlash 外部进行主密钥的管理。
- 数据密钥 (data key)：数据密钥由 TiFlash 生成，是实际用于加密的密钥。

多个 TiFlash 实例可共用一个主密钥，并且也可以和 TiKV 共用一个主密钥。在生产环境中，推荐通过 AWS KMS 提供主密钥。另外，你也可以通过文件形式提供主密钥。具体的主密钥生成方式和格式均与 TiKV 相同。

TiFlash 使用数据密钥加密所有落盘的数据文件，包括数据文件、Schmea 文件和计算过程中产生的临时数据文件等。默认情况下，TiFlash 每周自动轮换数据密钥，该轮换周期也可根据需要自定义配置。密钥轮换时，TiFlash 不会重写全部现有文件来替换密钥，但如果集群的写入量恒定，则后台 compaction 任务将会用最新的数据密钥对数据重新加密。TiFlash 跟踪密钥和加密方法，并使用密钥信息对读取的内容进行解密。

14.9.4.3.1 创建密钥

如需在 AWS 上创建一个密钥，请参考 TiKV 密钥的创建方式。

14.9.4.3.2 配置加密

启用加密的配置，可以在 `tiflash-learner.toml` 文件中添加以下内容：

```
[security.encryption]
data-encryption-method = "aes128-ctr"
data-key-rotation-period = "168h" # 7 days
```

或者，在 TiUP 集群模板文件中添加以下内容：

```
server_configs:
  tiflash-learner:
    security.encryption.data-encryption-method: "aes128-ctr"
    security.encryption.data-key-rotation-period: "168h" # 7 days
```

data-encryption-method 的可选值为 "aes128-ctr"、"aes192-ctr"、"aes256-ctr"、"sm4-ctr" (仅 v6.4.0 及之后版本) 和 "plaintext"。默认值为 "plaintext"，即默认不开启加密功能。data-key-rotation-period 指定 TiFlash 轮换密钥的频率。可以为新 TiFlash 集群或现有 TiFlash 集群开启加密，但只有启用后写入的数据才保证被加密。要禁用加密，请在配置文件中删除 data-encryption-method，或将该参数值为 "plaintext"，然后重启 TiFlash。若要替换加密算法，则将 data-encryption-method 替换成已支持的加密算法，然后重启 TiFlash。替换加密算法后，旧加密算法生成的加密文件会随着新数据的写入逐渐被重写成新加密算法所生成的加密文件。

如果启用了加密 (即 data-encryption-method 的值不是 "plaintext")，则必须指定主密钥。要使用 AWS KMS 方式指定为主密钥，配置方式如下：

在 tiflash-learner.toml 配置文件的 [security.encryption] 部分之后添加 [security.encryption.master-key]：

```
[security.encryption.master-key]
type = "kms"
key-id = "0987dcba-09fe-87dc-65ba-ab0987654321"
region = "us-west-2"
endpoint = "https://kms.us-west-2.amazonaws.com"
```

或者，在 TiUP 集群模板文件中添加以下内容：

```
server_configs:
  tiflash-learner:
    security.encryption.master-key.type: "kms"
    security.encryption.master-key.key-id: "0987dcba-09fe-87dc-65ba-ab0987654321"
    security.encryption.master-key.region: "us-west-2"
    security.encryption.master-key.endpoint: "https://kms.us-west-2.amazonaws.com"
```

上述配置项的含义与 TiKV 均相同。

若要使用文件方式指定主密钥，可以在 tiflash-learner.toml 配置文件中添加以下内容：

```
[security.encryption.master-key]
type = "file"
path = "/path/to/key/file"
```

或者，在 TiUP 集群模板文件中添加以下内容：

```
server_configs:
  tiflash-learner:
    security.encryption.master-key.type: "file"
    security.encryption.master-key.path: "/path/to/key/file"
```

上述配置项的含义以及密钥文件的内容格式与 TiKV 均相同。

14.9.4.3.3 轮换主密钥

TiFlash 轮换主秘钥的方法与 TiKV 相同。TiFlash 当前也不支持在线轮换主密钥，因此你需要重启 TiFlash 进行主密钥轮换。建议对运行中的、提供在线查询的 TiFlash 集群进行滚动重启。

要轮换 KMS CMK，可以在 tiflash-learner.toml 配置文件中添加以下内容：

```
[security.encryption.master-key]
type = "kms"
key-id = "50a0c603-1c6f-11e6-bb9e-3fadde80ce75"
region = "us-west-2"

[security.encryption.previous-master-key]
type = "kms"
key-id = "0987dcba-09fe-87dc-65ba-ab0987654321"
region = "us-west-2"
```

或者，在 TiUP 集群模板文件中添加以下内容：

```
server_configs:
  tiflash-learner:
    security.encryption.master-key.type: "kms"
    security.encryption.master-key.key-id: "50a0c603-1c6f-11e6-bb9e-3fadde80ce75"
    security.encryption.master-key.region: "us-west-2"
    security.encryption.previous-master-key.type: "kms"
    security.encryption.previous-master-key.key-id: "0987dcba-09fe-87dc-65ba-ab0987654321"
    security.encryption.previous-master-key.region: "us-west-2"
```

14.9.4.3.4 监控和调试

要监控静态加密（如果 TiFlash 中部署了 Grafana 组件），可以查看 TiFlash-Proxy-Details -> Encryption 面板中的监控项，其中监控项的含义与 TiKV 相同。

在调试方面，由于 TiFlash 复用了 TiKV 管理加密元数据的逻辑，因此也可使用 `tikv-ctl` 命令查看加密元数据（例如使用的加密方法和数据密钥列表）。该操作可能会暴露密钥，因此不推荐在生产环境中使用。详情参阅TiKV Control。

14.9.4.3.5 TiFlash 版本间兼容性

TiFlash 在 v4.0.9 同样对加密元数据操作进行了优化，其兼容性要求与 TiKV 相同，详情参考TiKV 版本间兼容性。

14.9.4.4 BR S3 服务端加密

使用 BR 备份数据到 S3 时，若要启用 S3 服务端加密，需要传递 `--s3.sse` 参数并将参数值设置为 `aws:kms`。S3 将使用自己的 KMS 密钥进行加密。示例如下：

```
./br backup full --pd <pd-address> --storage "s3://<bucket>/<prefix>" --s3.sse aws:kms
```

若要使用用户创建和拥有的自定义 AWS KMS CMK，需另外传递 `--s3.sse-kms-key-id` 参数。此时，BR 进程和集群中的所有 TiKV 节点都需访问该 KMS CMK（例如，通过 AWS IAM），并且该 KMS CMK 必须与存储备份的 S3 bucket 位于同一 AWS 区域。建议通过 AWS IAM 向 BR 进程和 TiKV 节点授予对 KMS CMK 的访问权限。参见 AWS 文档中的 IAM。示例如下：

```
./br backup full --pd <pd-address> --storage "s3://<bucket>/<prefix>" --s3.sse aws:kms --s3.sse-
    ↪ kms-key-id 0987dcba-09fe-87dc-65ba-ab0987654321
```

恢复备份时，不需要也不可指定 `--s3.sse` 和 `--s3.sse-kms-key-id` 参数。S3 将自动相应进行解密。用于恢复备份数据的 BR 进程和集群中的 TiKV 节点也需要访问 KMS CMK，否则恢复将失败。示例如下：

```
./br restore full --pd <pd-address> --storage "s3://<bucket>/<prefix>"
```

14.9.5 为 TiDB 落盘文件开启加密

当系统变量tidb_enable_tmp_storage_on_oom 为 ON 时，如果单条 SQL 语句的内存使用超出系统变量tidb_mem_quota_query 的限制，某些算子可以将执行时的中间结果作为临时文件落盘保存，直到查询执行完成之后将它们删除。

用户可以开启落盘文件加密功能，防止攻击者通过读取临时文件来访问数据。

14.9.5.1 配置方法

要启用落盘文件加密功能，可以在 TiDB 配置文件中的 `[security]` 部分，配置spilled-file-encryption-method 选项：

```
[security]
spilled-file-encryption-method = "aes128-ctr"
```

`spilled-file-encryption-method` 的可选值为 `aes128-ctr` 和 `plaintext`。默认值为 `plaintext`，表示不启用加密。

14.9.6 日志脱敏

TiDB 在提供详细的日志信息时，可能会把数据库敏感的数据（例如用户数据）打印出来，造成数据安全方面的风险。因此 TiDB、TiKV、PD 等组件各提供了一个配置项开关，开关打开后，会隐藏日志中包含的用户数据值。

14.9.6.1 TiDB 组件日志脱敏

TiDB 侧的日志脱敏需要将global.tidb_redact_log 的值设为 1。该变量值默认为 0，即关闭脱敏。

可以通过 `set` 语法，设置全局系统变量 `tidb_redact_log`，示例如下：

```
set @@global.tidb_redact_log=1;
```

设置后，所有新 session 产生的日志都会脱敏：

```
create table t (a int, unique key (a));
Query OK, 0 rows affected (0.00 sec)

insert into t values (1),(1);
ERROR 1062 (23000): Duplicate entry '1' for key 't.a'
```

打印出的错误日志如下：

```
[2020/10/20 11:45:49.539 +08:00] [INFO] [conn.go:800] ["command dispatched failed"] [conn=5] [
    ↪ connInfo="id:5, addr:127.0.0.1:57222 status:10, collation:utf8_general_ci,  user:root"] [
    ↪ command=Query] [status="inTxn:0, autocommit:1"] [sql="insert into t values ( ? ) , ( ? )
    ↪ "] [txn_mode=OPTIMISTIC] [err="[kv:1062]Duplicate entry '?' for key 't.a'"]
```

从以上报错日志可以看到，开启 tidb_redact_log 后，报错信息里的敏感内容被隐藏掉了（目前是用问号替代）。TiDB 日志中会把敏感信息隐藏掉，以此规避数据安全风险。

14.9.6.2 TiKV 组件日志脱敏

TiKV 侧的日志脱敏需要将security.redact-info-log 的值设为 true。该配置项值默认为 false，即关闭脱敏。

14.9.6.3 PD 组件日志脱敏

PD 侧的日志脱敏需要将security.redact-info-log 的值设为 true。该配置项值默认为 false，即关闭脱敏。

14.9.6.4 TiFlash 组件日志脱敏

TiFlash 侧的日志脱敏需要将 tiflash-server 中security.redact_info_log 配置项的值以及 tiflash-learner 中security
↪ .redact-info-log 配置项的值均设为 true。两配置项默认值均为 false，即关闭脱敏。

14.10 权限

14.10.1 与 MySQL 安全特性差异

TiDB 支持与 MySQL 5.7 类似的安全特性，同时 TiDB 还支持 MySQL 8.0 的部分安全特性。TiDB 的安全特性在实现上与 MySQL 存在差异。

14.10.1.1 不支持的安全功能特性

- 不支持列级别权限设置。
- 不支持权限属性 max_questions，max_updated 以及 max_user_connections。
- 不支持密码修改验证策略，修改密码时需要验证当前密码。
- 不支持双密码策略。
- 不支持随机密码生成策略。
- 不支持多因素身份验证。

14.10.1.2 与 MySQL 有差异的安全特性详细说明

14.10.1.2.1 密码过期策略

针对密码过期策略功能，TiDB 与 MySQL 的比较如下：

- MySQL 5.7 和 8.0 支持密码过期策略管理功能。
- TiDB 从 v6.5.0 起支持密码过期策略管理功能。

TiDB 的密码过期策略功能与 MySQL 保持一致，但是在密码过期处理机制上存在以下差异：

- MySQL 5.7 和 8.0 在密码过期后是否将客户端的连接限制为“沙盒模式”，由客户端和服务端设置的组合确定。
- TiDB 在密码过期后是否将客户端的连接限制为“沙盒模式”，仅由 TiDB 配置文件中的 `[security]` 部分的`disconnect-on-expired-password` 选项确定。

14.10.1.2.2 密码复杂度策略

针对密码复杂度策略功能，TiDB 与 MySQL 的比较如下：

- MySQL 5.7 以 validate_password 插件的形式实现了密码复杂度策略管理功能。
- MySQL 8.0 重新以 validate_password 组件的形式实现了密码复杂度策略管理功能。
- TiDB 从 v6.5.0 起内置实现了密码复杂度策略管理功能。

因此，功能实现上存在以下差异：

- 密码复杂度策略功能如何启用：
  - MySQL 5.7 以 validate_password 插件的形式实现，需要进行插件的安装以启用密码复杂度策略管理。
  - MySQL 8.0 以 validate_password 组件的形式实现，需要进行组件的安装以启用密码复杂度策略管理。
  - TiDB 内置实现了密码复杂度策略管理，支持通过系统变量`validate_password.enable` 启用密码复杂度策略管理。
- 密码字典功能：
  - MySQL 5.7 通过变量 `validate_password_dictionary_file` 指定一个文件路径，在文件中写入密码中不允许包含的单词。
  - MySQL 8.0 通过变量 `validate_password.dictionary_file` 指定一个文件路径，在文件中写入密码中不允许包含的单词。
  - TiDB 通过变量`validate_password.dictionary` 指定一个字符串，在该字符串中写入密码中不允许包含的单词。

14.10.1.2.3 密码连续错误限制登录策略

针对密码连续错误限制登录策略功能，TiDB 与 MySQL 的比较如下：

- MySQL 5.7 不支持密码连续错误限制登录策略管理功能。
- MySQL 8.0 支持密码连续错误限制登录策略管理功能。
- TiDB 从 v6.5.0 起支持密码连续错误限制登录策略管理功能。

因为用户的失败尝试次数和锁定状态需要做到全局一致，而 TiDB 是分布式数据库，不能像 MySQL 在服务端的内存中记录失败尝试次数和锁定状态，所以实现机制存在以下差异：

- 用户未被自动锁定，失败尝试次数的计数重置场景：
  - MySQL 8.0：
    * 服务器重启时，所有用户失败尝试次数的计数都会被重置。
    * 执行 `FLUSH PRIVILEGES` 时，所有用户失败尝试次数的计数都会被重置。
    * 对该用户执行 `ALTER USER ... ACCOUNT UNLOCK` 解锁命令时。
    * 该用户登录成功时。
  - TiDB：
    * 对该用户执行 `ALTER USER ... ACCOUNT UNLOCK` 解锁命令时。
    * 该用户登录成功时。
- 账户被自动锁定后的解锁场景：
  - MySQL 8.0：
    * 服务器重启时，所有用户的自动锁定标识都会被重置。
    * 执行 `FLUSH PRIVILEGES` 时，所有用户的自动锁定标识都会被重置。
    * 该用户锁定时间结束，这种情况下，用户的自动锁定标识将在下次登录尝试时重置。
    * 对该用户执行 `ALTER USER ... ACCOUNT UNLOCK` 解锁命令时。
  - TiDB：
    * 该用户锁定时间结束，这种情况下，用户的自动锁定标识将在下次登录尝试时重置。
    * 对该用户执行 `ALTER USER ... ACCOUNT UNLOCK` 解锁命令时。

14.10.1.2.4 密码重用策略

针对密码重用策略功能，TiDB 与 MySQL 的比较如下：

- MySQL 5.7 不支持密码重用策略管理功能。
- MySQL 8.0 支持密码重用策略管理功能。
- TiDB 从 v6.5.0 起支持密码重用策略管理功能。

TiDB 的密码重用策略功能与 MySQL 一致，在实现密码重用策略时都增加了系统表 `mysql.password_history`，但 TiDB 与 MySQL 在删除 `mysql.user` 系统表中不存在的用户时存在以下差异：

- 场景：没有正确创建用户（例如：`user01`），而通过 `INSERT INTO mysql.password_history VALUES (...)` 命令直接向 `mysql.password_history` 系统表中添加一条 `user01` 的记录，此时系统表 `mysql.user` 中没有 `user01` 的记录。对该用户执行 `DROP USER` 操作时，TiDB 和 MySQL 状态不一致。
- 差异点：
  - MySQL：执行 `DROP USER user01` 时，在 `mysql.user` 和 `mysql.password_history` 系统表中匹配 `user01`，若在两个系统表或其中一个系统表中匹配成功，则 `DROP USER` 操作可以正常执行，不会报错。
  - TiDB：执行 `DROP USER user01` 时，只在 `mysql.user` 系统表中匹配 `user01`，若没有匹配成功，则 `DROP USER` 操作执行失败，返回报错。此时如果需要成功执行 `DROP USER user01` 操作，删除 `mysql.password_history` 中 `user01` 的记录，请使用 `DROP USER IF EXISTS user01`。

14.10.1.3 可用的身份验证插件

TiDB 支持多种身份验证方式。通过使用CREATE USER 语句和ALTER USER 语句，即可创建新用户或更改 TiDB 权限系统内的已有用户。TiDB 身份验证方式与 MySQL 兼容，其名称与 MySQL 保持一致。

TiDB 目前支持的身份验证方式可在以下的表格中查找到。服务器和客户端建立连接时，如要指定服务器对外通告的默认验证方式，可通过default_authentication_plugin 变量进行设置。tidb_sm3_password 为仅在 TiDB 支持的 SM3 身份验证方式，使用该方式登录的用户需要使用 TiDB-JDBC。tidb_auth_token 为仅用于 TiDB Cloud 内部的基于 JSON Web Token (JWT) 的认证方式。

针对 TLS 身份验证，TiDB 目前采用不同的配置方案。具体情况请参见为 TiDB 客户端服务端间通信开启加密传输。

身份验证方式	支持
mysql_native_password	是
sha256_password	否
caching_sha2_password	是（5.2.0 版本起）
auth_socket	是（5.3.0 版本起）
tidb_sm3_password	是（6.3.0 版本起）
tidb_auth_token	是（6.4.0 版本起）
TLS 证书	是
LDAP	否
PAM	否
ed25519 (MariaDB)	否
GSSAPI (MariaDB)	否
FIDO	否

14.10.2 权限管理

TiDB 支持 MySQL 5.7 的权限管理系统，包括 MySQL 的语法和权限类型。同时 TiDB 还支持 MySQL 8.0 的以下特性：

- 从 TiDB 3.0 开始，支持 SQL 角色。
- 从 TiDB 5.1 开始，支持动态权限。

本文档主要介绍 TiDB 权限相关操作、各项操作需要的权限以及权限系统的实现。

14.10.2.1 权限相关操作

14.10.2.1.1 授予权限

授予 xxx 用户对数据库 test 的读权限：

```
GRANT SELECT ON test.* TO 'xxx'@'%';
```

为 xxx 用户授予所有数据库，全部权限：

```
GRANT ALL PRIVILEGES ON *.* TO 'xxx'@'%';
```

默认情况下，如果指定的用户不存在，GRANT 语句将报错。该行为受 SQL Mode 中的 NO_AUTO_CREATE_USER 控制。

```
SET sql_mode=DEFAULT;
Query OK, 0 rows affected (0.00 sec)

SELECT @@sql_mode;
+--
    ↪ ----------------------------------------------------------------------------------------------------------------
    ↪
| @@sql_mode
    ↪
    ↪ |
+--
    ↪ ----------------------------------------------------------------------------------------------------------------
    ↪
| ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO,
    ↪ NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION |
+--
    ↪ ----------------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)

SELECT * FROM mysql.user WHERE user='idontexist';
Empty set (0.00 sec)

GRANT ALL PRIVILEGES ON test.* TO 'idontexist';
ERROR 1105 (HY000): You are not allowed to create a user with GRANT

SELECT user,host,authentication_string FROM mysql.user WHERE user='idontexist';
Empty set (0.00 sec)
```

在下面的例子中，由于没有将 SQL Mode 设置为 NO_AUTO_CREATE_USER，用户 idontexist 会被自动创建且密码为空。不推荐使用这种方式，因为会带来安全风险：如果用户名拼写错误，会导致新用户被创建且密码为空。

```
SET @@sql_mode='ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,
    ↪ ERROR_FOR_DIVISION_BY_ZERO,NO_ENGINE_SUBSTITUTION';
Query OK, 0 rows affected (0.00 sec)

SELECT @@sql_mode;
+--
    ↪ ----------------------------------------------------------------------------------------------------------------
    ↪
| @@sql_mode
    ↪
    ↪ |
+--
```

```
    ↪ ------------------------------------------------------------------------------------------------------------------------
    ↪
| ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO,
    ↪ NO_ENGINE_SUBSTITUTION |
+--
    ↪ ------------------------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)

SELECT * FROM mysql.user WHERE user='idontexist';
Empty set (0.00 sec)

GRANT ALL PRIVILEGES ON test.* TO 'idontexist';
Query OK, 1 row affected (0.05 sec)

SELECT user,host,authentication_string FROM mysql.user WHERE user='idontexist';
+------------+------+-----------------------+
| user       | host | authentication_string |
+------------+------+-----------------------+
| idontexist | %    |                       |
+------------+------+-----------------------+
1 row in set (0.01 sec)
```

GRANT 还可以模糊匹配地授予用户数据库的权限：

```
GRANT ALL PRIVILEGES ON `te%`.* TO genius;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
SELECT user,host,db FROM mysql.db WHERE user='genius';
```

```
+--------|------|-----+
| user   | host | db  |
+--------|------|-----+
| genius | %    | te% |
+--------|------|-----+
1 row in set (0.00 sec)
```

这个例子中通过 % 模糊匹配，所有 te 开头的数据库，都被授予了权限。

14.10.2.1.2 收回权限

REVOKE 语句与 GRANT 对应：

```
REVOKE ALL PRIVILEGES ON `test`.* FROM 'genius'@'localhost';
```

> 注意：
>
> REVOKE 收回权限时只做精确匹配，若找不到记录则报错。而 GRANT 授予权限时可以使用模糊匹配。

```
REVOKE ALL PRIVILEGES ON `te%`.* FROM 'genius'@'%';
```

```
ERROR 1141 (42000): There is no such grant defined for user 'genius' on host '%'
```

关于模糊匹配和转义，字符串和 identifier：

```
GRANT ALL PRIVILEGES ON `te\%`.* TO 'genius'@'localhost';
```

```
Query OK, 0 rows affected (0.00 sec)
```

上述例子是精确匹配名为 te% 的数据库，注意使用 \ 转义字符。

以单引号包含的部分，是一个字符串。以反引号包含的部分，是一个 identifier。注意下面的区别：

```
GRANT ALL PRIVILEGES ON 'test'.* TO 'genius'@'localhost';
```

```
ERROR 1064 (42000): You have an error in your SQL syntax; check the
manual that corresponds to your MySQL server version for the right
syntax to use near ''test'.* to 'genius'@'localhost'' at line 1
```

```
GRANT ALL PRIVILEGES ON `test`.* TO 'genius'@'localhost';
```

```
Query OK, 0 rows affected (0.00 sec)
```

如果想将一些特殊的关键字做为表名，可以用反引号包含起来。比如：

```
CREATE TABLE `select` (id int);
```

```
Query OK, 0 rows affected (0.27 sec)
```

14.10.2.1.3 查看为用户分配的权限

SHOW GRANTS 语句可以查看为用户分配了哪些权限。例如：

查看当前用户的权限：

```
SHOW GRANTS;
```

```
+-------------------------------------------------------------+
| Grants for User                                             |
+-------------------------------------------------------------+
| GRANT ALL PRIVILEGES ON *.* TO 'root'@'%' WITH GRANT OPTION |
+-------------------------------------------------------------+
```

或者：

```
SHOW GRANTS FOR CURRENT_USER();
```

查看某个特定用户的权限：

```
SHOW GRANTS FOR 'user'@'host';
```

例如，创建一个用户 rw_user@192.168.% 并为其授予 test.write_table 表的写权限，和全局读权限。

```
CREATE USER `rw_user`@`192.168.%`;
GRANT SELECT ON *.* TO `rw_user`@`192.168.%`;
GRANT INSERT, UPDATE ON `test`.`write_table` TO `rw_user`@`192.168.%`;
```

查看用户 rw_user@192.168.% 的权限。

```
SHOW GRANTS FOR `rw_user`@`192.168.%`;
```

```
+-----------------------------------------------------------------+
| Grants for rw_user@192.168.%                                    |
+-----------------------------------------------------------------+
| GRANT Select ON *.* TO 'rw_user'@'192.168.%'                    |
| GRANT Insert,Update ON test.write_table TO 'rw_user'@'192.168.%' |
+-----------------------------------------------------------------+
```

14.10.2.1.4 动态权限

从 v5.1 开始，TiDB 支持 MySQL 8.0 中的动态权限特性。动态权限用于限制 SUPER 权限，实现对某些操作更细粒度的访问。例如，系统管理员可以使用动态权限来创建一个只能执行 BACKUP 和 RESTORE 操作的用户帐户。

动态权限包括：

- BACKUP_ADMIN
- RESTORE_ADMIN
- SYSTEM_USER
- SYSTEM_VARIABLES_ADMIN
- ROLE_ADMIN
- CONNECTION_ADMIN
- PLACEMENT_ADMIN 允许创建、删除和修改放置策略 (placement policy)。
- DASHBOARD_CLIENT 允许登录 TiDB Dashboard。

- RESTRICTED_TABLES_ADMIN 允许在 SEM 打开的情况下查看系统表。
- RESTRICTED_STATUS_ADMIN 允许在 SEM 打开的情况下查看SHOW [GLOBAL|SESSION] STATUS 中的状态变量。
- RESTRICTED_VARIABLES_ADMIN 允许在 SEM 打开的情况下查看所有系统变量。
- RESTRICTED_USER_ADMIN 不允许在 SEM 打开的情况下使用 SUPER 用户撤销访问权限。
- RESTRICTED_CONNECTION_ADMIN 允许 KILL 属于 RESTRICTED_USER_ADMIN 用户的连接。该权限对 KILL 和 KILL TIDB 语句生效。
- RESTRICTED_REPLICA_WRITER_ADMIN 允许权限拥有者在 TiDB 集群开启了只读模式的情况下不受影响地执行写入或更新操作，详见tidb_restricted_read_only 配置项。

若要查看全部的动态权限，请执行 SHOW PRIVILEGES 语句。由于用户可使用插件来添加新的权限，因此可分配的权限列表可能因用户的 TiDB 安装情况而异。

14.10.2.2 SUPER 权限

- 拥有 SUPER 权限的用户能完成几乎所有的操作，默认情况下只有 root 用户拥有该权限。请谨慎向其它用户授予 SUPER 权限。
- SUPER 权限在 MySQL 8.0 中被认为是过时的，可以通过动态权限替代 SUPER 权限进行更细粒度的权限控制。

14.10.2.3 TiDB 各操作需要的权限

TiDB 用户目前拥有的权限可以在 INFORMATION_SCHEMA.USER_PRIVILEGES 表中查找到。例如：

```
SELECT * FROM INFORMATION_SCHEMA.USER_PRIVILEGES WHERE grantee = "'root'@'%'";
+------------+---------------+-------------------------+--------------+
| GRANTEE    | TABLE_CATALOG | PRIVILEGE_TYPE          | IS_GRANTABLE |
+------------+---------------+-------------------------+--------------+
| 'root'@'%' | def           | Select                  | YES          |
| 'root'@'%' | def           | Insert                  | YES          |
| 'root'@'%' | def           | Update                  | YES          |
| 'root'@'%' | def           | Delete                  | YES          |
| 'root'@'%' | def           | Create                  | YES          |
| 'root'@'%' | def           | Drop                    | YES          |
| 'root'@'%' | def           | Process                 | YES          |
| 'root'@'%' | def           | References              | YES          |
| 'root'@'%' | def           | Alter                   | YES          |
| 'root'@'%' | def           | Show Databases          | YES          |
| 'root'@'%' | def           | Super                   | YES          |
| 'root'@'%' | def           | Execute                 | YES          |
| 'root'@'%' | def           | Index                   | YES          |
| 'root'@'%' | def           | Create User             | YES          |
| 'root'@'%' | def           | Create Tablespace       | YES          |
| 'root'@'%' | def           | Trigger                 | YES          |
| 'root'@'%' | def           | Create View             | YES          |
| 'root'@'%' | def           | Show View               | YES          |
| 'root'@'%' | def           | Create Role             | YES          |
```

```
| 'root'@'%' | def           | Drop Role                | YES          |
| 'root'@'%' | def           | CREATE TEMPORARY TABLES | YES          |
| 'root'@'%' | def           | LOCK TABLES              | YES          |
| 'root'@'%' | def           | CREATE ROUTINE           | YES          |
| 'root'@'%' | def           | ALTER ROUTINE            | YES          |
| 'root'@'%' | def           | EVENT                    | YES          |
| 'root'@'%' | def           | SHUTDOWN                 | YES          |
| 'root'@'%' | def           | RELOAD                   | YES          |
| 'root'@'%' | def           | FILE                     | YES          |
| 'root'@'%' | def           | CONFIG                   | YES          |
| 'root'@'%' | def           | REPLICATION CLIENT       | YES          |
| 'root'@'%' | def           | REPLICATION SLAVE        | YES          |
+------------+---------------+--------------------------+--------------+
31 rows in set (0.00 sec)
```

14.10.2.3.1 ALTER

- 对于所有的 ALTER 语句，均需要用户对所操作的表拥有 ALTER 权限。
- 除 ALTER...DROP 和 ALTER...RENAME TO 外，均需要对所操作表拥有 INSERT 和 CREATE 权限。
- 对于 ALTER...DROP 语句，需要对表拥有 DROP 权限。
- 对于 ALTER...RENAME TO 语句，需要对重命名前的表拥有 DROP 权限，对重命名后的表拥有 CREATE 和 INSERT 权限。

> **注意：**
>
> 根据 MySQL 5.7 文档中的说明，对表进行 ALTER 操作需要 INSERT 和 CREATE 权限，但在 MySQL 5.7.25 版本实际情况中，该操作仅需要 ALTER 权限。目前，TiDB 中的 ALTER 权限与 MySQL 实际行为保持一致。

14.10.2.3.2 BACKUP

需要拥有 SUPER 或者 BACKUP_ADMIN 权限。

14.10.2.3.3 CREATE DATABASE

需要拥有全局 CREATE 权限。

14.10.2.3.4 CREATE INDEX

需要对所操作的表拥有 INDEX 权限。

14.10.2.3.5 CREATE TABLE

需要对要创建的表所在的数据库拥有 CREATE 权限；若使用 CREATE TABLE...LIKE... 需要对相关的表拥有 SELECT 权限。

14.10.2.3.6 CREATE VIEW

需要拥有 CREATE VIEW 权限。

> 注意：
> 如果当前登录用户与创建视图的用户不同，除需要 CREATE VIEW 权限外，还需要 SUPER 权限。

14.10.2.3.7 DROP DATABASE

需要对数据库拥有 DROP 权限。

14.10.2.3.8 DROP INDEX

需要对所操作的表拥有 INDEX 权限。

14.10.2.3.9 DROP TABLES

需要对所操作的表拥有 DROP 权限。

14.10.2.3.10 LOAD DATA

LOAD DATA 需要对所操作的表拥有 INSERT 权限。执行 REPLACE INTO 语句还需要对所操作的表拥有 DELETE 权限。

14.10.2.3.11 TRUNCATE TABLE

需要对所操作的表拥有 DROP 权限。

14.10.2.3.12 RENAME TABLE

需要对重命名前的表拥有 ALTER 和 DROP 权限，对重命名后的表拥有 CREATE 和 INSERT 权限。

14.10.2.3.13 ANALYZE TABLE

需要对所操作的表拥有 INSERT 和 SELECT 权限。

14.10.2.3.14 SHOW

SHOW CREATE TABLE 需要任意一种权限。

SHOW CREATE VIEW 需要 SHOW VIEW 权限。

SHOW GRANTS 需要拥有对 mysql 数据库的 SELECT 权限。如果是使用 SHOW GRANTS 查看当前用户权限，则不需要任何权限。

SHOW PROCESSLIST 需要 SUPER 权限来显示属于其他用户的连接。

14.10.2.3.15 CREATE ROLE/USER

CREATE ROLE 需要 CREATE ROLE 权限。

CREATE USER 需要 CREATE USER 权限

14.10.2.3.16 DROP ROLE/USER

DROP ROLE 需要 DROP ROLE 权限。

DROP USER 需要 CREATE USER 权限

14.10.2.3.17 ALTER USER

ALTER USER 需要 CREATE USER 权限。

14.10.2.3.18 GRANT

GRANT 需要 GRANT 权限并且拥有 GRANT 所赋予的权限。

如果在 GRANTS 语句中创建用户，需要有 CREATE USER 权限。

GRANT ROLE 操作需要拥有 SUPER 或者 ROLE_ADMIN 权限。

14.10.2.3.19 REVOKE

REVOKE 需要 GRANT 权限并且拥有 REVOKE 所指定要撤销的权限。

REVOKE ROLE 操作需要拥有 SUPER 或者 ROLE_ADMIN 权限。

14.10.2.3.20 SET GLOBAL

使用 SET GLOBAL 设置全局变量需要拥有 SUPER 或者 SYSTEM_VARIABLES_ADMIN 权限。

14.10.2.3.21 ADMIN

需要拥有 SUPER 权限。

14.10.2.3.22 SET DEFAULT ROLE

需要拥有 SUPER 权限。

14.10.2.3.23 KILL

使用 KILL 终止其他用户的会话需要拥有 SUPER 或者 CONNECTION_ADMIN 权限。

14.10.2.3.24 CREATE RESOURCE GROUP

需要拥有 SUPER 或者 RESOURCE_GROUP_ADMIN 权限。

14.10.2.3.25 ALTER RESOURCE GROUP

需要拥有 SUPER 或者 RESOURCE_GROUP_ADMIN 权限。

14.10.2.3.26 DROP RESOURCE GROUP

需要拥有 SUPER 或者 RESOURCE_GROUP_ADMIN 权限。

14.10.2.4 权限系统的实现

14.10.2.4.1 授权表

以下几张系统表是非常特殊的表，权限相关的数据全部存储在这几张表内。

- mysql.user：用户账户，全局权限
- mysql.db：数据库级别的权限
- mysql.tables_priv：表级别的权限
- mysql.columns_priv：列级别的权限，当前暂不支持

这几张表包含了数据的生效范围和权限信息。例如，mysql.user 表的部分数据：

```
SELECT User,Host,Select_priv,Insert_priv FROM mysql.user LIMIT 1;
```

```
+------|------|-------------|-------------+
| User | Host | Select_priv | Insert_priv |
+------|------|-------------|-------------+
| root | %    | Y           | Y           |
+------|------|-------------|-------------+
1 row in set (0.00 sec)
```

这条记录中，Host 和 User 决定了 root 用户从任意主机 (%) 发送过来的连接请求可以被接受，而 Select_priv 和 Insert_priv 表示用户拥有全局的 Select 和 Insert 权限。mysql.user 这张表里面的生效范围是全局的。

mysql.db 表里面包含的 Host 和 User 决定了用户可以访问哪些数据库，权限列的生效范围是数据库。

理论上，所有权限管理相关的操作，都可以通过直接对授权表的 CRUD 操作完成。

实现层面其实也只是包装了一层语法糖。例如删除用户会执行：

```
DELETE FROM mysql.user WHERE user='test';
```

但是，不推荐手动修改授权表，建议使用 DROP USER 语句：

```
DROP USER 'test';
```

14.10.2.4.2 连接验证

当客户端发送连接请求时，TiDB 服务器会对登录操作进行验证。验证过程先检查 mysql.user 表，当某条记录的 User 和 Host 和连接请求匹配上了，再去验证 authentication_string。用户身份基于两部分信息，发起连接的客户端的 Host，以及用户名 User。如果 User 不为空，则用户名必须精确匹配。

User+Host 可能会匹配 user 表里面多行，为了处理这种情况，user 表的行是排序过的，客户端连接时会依次去匹配，并使用首次匹配到的那一行做权限验证。排序是按 Host 在前，User 在后。

14.10.2.4.3 请求验证

连接成功之后，请求验证会检测执行操作是否拥有足够的权限。

对于数据库相关请求 (INSERT, UPDATE)，先检查 mysql.user 表里面的用户全局权限，如果权限够，则直接可以访问。如果全局权限不足，则再检查 mysql.db 表。

user 表的权限是全局的，并且不管默认数据库是哪一个。比如 user 里面有 DELETE 权限，任何一行，任何的表，任何的数据库。

db表里面，User 为空是匹配匿名用户，User 里面不能有通配符。Host 和 Db 列里面可以有 % 和 _，可以模式匹配。

user 和 db 读到内存也是排序的。

tables_priv 和 columns_priv 中使用 % 是类似的，但是在Db, Table_name, Column_name 这些列不能包含 %。加载进来时排序也是类似的。

14.10.2.4.4 生效时机

TiDB 启动时，将一些权限检查的表加载到内存，之后使用缓存的数据来验证权限。系统会周期性的将授权表从数据库同步到缓存，生效则是由同步的周期决定，目前这个值设定的是 5 分钟。

修改了授权表，如果需要立即生效，可以手动调用：

```
FLUSH PRIVILEGES;
```

14.10.3 TiDB 用户账户管理

本文档主要介绍如何管理 TiDB 用户账户。

要快速了解 TiDB 如何进行认证与赋权并创建与管理用户账户，建议先观看下面的培训视频（时长 22 分钟）。注意本视频只作为学习参考，如需了解具体的用户账户管理方法，请参考本文档的内容。

14.10.3.1 用户名和密码

TiDB 将用户账户存储在 mysql.user 系统表里面。每个账户由用户名和 host 作为标识。每个账户可以设置一个密码。每个用户名最长为 32 个字符。

通过 MySQL 客户端连接到 TiDB 服务器，通过指定的账户和密码登录：

```
mysql --port 4000 --user xxx --password
```

使用缩写的命令行参数则是：

```
mysql -P 4000 -u xxx -p
```

14.10.3.2 添加用户

添加用户有两种方式：

- 通过标准的用户管理的 SQL 语句创建用户以及授予权限，比如 CREATE USER 和 GRANT。
- 直接通过 INSERT、UPDATE 和 DELETE 操作授权表。不推荐使用这种方式添加用户，因为容易导致修改不完整。

除以上两种方法外，你还可以使用第三方图形化界面工具来添加用户。

```
CREATE USER [IF NOT EXISTS] user [IDENTIFIED BY 'auth_string'];
```

设置登录密码后，auth_string 会被 TiDB 经过加密存储在 mysql.user 表中。

```
CREATE USER 'test'@'127.0.0.1' IDENTIFIED BY 'xxx';
```

TiDB 的用户账户名由一个用户名和一个主机名组成。账户名的语法为 'user_name'@'host_name'。

- user_name 大小写敏感。
- host_name 可以是一个主机名或 IP 地址。主机名或 IP 地址中允许使用通配符 % 和 _。例如，名为 '%' 的主机名可以匹配所有主机，'192.168.1.%' 可以匹配子网中的所有主机。

host 支持模糊匹配，比如：

```
CREATE USER 'test'@'192.168.10.%';
```

允许 test 用户从 192.168.10 子网的任何一个主机登录。

如果没有指定 host，则默认是所有 IP 均可登录。如果没有指定密码，默认为空：

```
CREATE USER 'test';
```

等价于：

```
CREATE USER 'test'@'%' IDENTIFIED BY '';
```

为一个不存在的用户授权时，是否会自动创建用户的行为受 sql_mode 影响。如果 sql_mode 中包含 NO_AUTO_CREATE_USER，则 GRANT 不会自动创建用户并报错。

假设 sql_mode 不包含 NO_AUTO_CREATE_USER，下面的例子用 CREATE USER 和 GRANT 语句创建了四个账户：

```
CREATE USER 'finley'@'localhost' IDENTIFIED BY 'some_pass';
```

```
GRANT ALL PRIVILEGES ON *.* TO 'finley'@'localhost' WITH GRANT OPTION;
```

```
CREATE USER 'finley'@'%' IDENTIFIED BY 'some_pass';
```

```
GRANT ALL PRIVILEGES ON *.* TO 'finley'@'%' WITH GRANT OPTION;
```

```
CREATE USER 'admin'@'localhost' IDENTIFIED BY 'admin_pass';
```

```
GRANT RELOAD,PROCESS ON *.* TO 'admin'@'localhost';
```

```
CREATE USER 'dummy'@'localhost';
```

使用 SHOW GRANTS 可以看到为一个用户授予的权限：

```
SHOW GRANTS FOR 'admin'@'localhost';
```

```
+-------------------------------------------------------+
| Grants for admin@localhost                            |
+-------------------------------------------------------+
| GRANT RELOAD, PROCESS ON *.* TO 'admin'@'localhost' |
+-------------------------------------------------------+
```

14.10.3.3 删除用户

使用 DROP USER 语句可以删除用户，例如：

```
DROP USER 'test'@'localhost';
```

这个操作会清除用户在 mysql.user 表里面的记录项，并且清除在授权表里面的相关记录。

14.10.3.4 保留用户账户

TiDB 在数据库初始化时会生成一个 'root'@'%' 的默认账户。

14.10.3.5 设置资源限制

暂不支持。

14.10.3.6 设置密码

TiDB 将密码存在 mysql.user 系统数据库里面。只有拥有 CREATE USER 权限，或者拥有 mysql 数据库权限（INSERT 权限用于创建，UPDATE 权限用于更新）的用户才能够设置或修改密码。

- 在 CREATE USER 创建用户时通过 IDENTIFIED BY 指定密码：

```
CREATE USER 'test'@'localhost' IDENTIFIED BY 'mypass';
```

- 为一个已存在的账户修改密码，可以通过 SET PASSWORD FOR 或者 ALTER USER 语句完成：

```
SET PASSWORD FOR 'root'@'%' = 'xxx';
```

或者：

```
ALTER USER 'test'@'localhost' IDENTIFIED BY 'mypass';
```

14.10.3.7 忘记 root 密码

1. 修改 TiDB 配置文件：
    1. 登录其中一台 tidb-server 实例所在的机器。
    2. 进入 TiDB 节点的部署目录下的 conf 目录，找到 tidb.toml 配置文件。
    3. 在配置文件的 security 部分添加配置项 skip-grant-table。如无 security 部分，则将以下两行内容添加至 tidb.toml 配置文件尾部：

    ```
    [security]
    skip-grant-table = true
    ```

2. 终止该 tidb-server 的进程：
    1. 查看 tidb-server 的进程：

    ```
    ps aux | grep tidb-server
    ```

    2. 找到 tidb-server 对应的进程 ID (PID) 并使用 kill 命令停掉该进程：

    ```
    kill -9 <pid>
    ```

3. 使用修改之后的配置启动 TiDB：

> 注意：
> 设置 skip-grant-table 之后，启动 TiDB 进程会增加操作系统用户检查，只有操作系统的 root 用户才能启动 TiDB 进程。

    1. 进入 TiDB 节点部署目录下的 scripts 目录。

2. 切换到操作系统 root 账号。
3. 在前台执行目录中的 run_tidb.sh 脚本。
4. 在新的终端窗口中使用 root 登录后修改密码：

```
mysql -h 127.0.0.1 -P 4000 -u root
```

4. 停止运行 run_tidb.sh 脚本，并去掉第 1 步中在 TiDB 配置文件中添加的内容，等待 tidb-server 自启动。

14.10.3.8 FLUSH PRIVILEGES

用户以及权限相关的信息都存储在 TiKV 服务器中，TiDB 在进程内部会缓存这些信息。一般通过 CREATE USER，GRANT 等语句来修改相关信息时，可在整个集群迅速生效。如果遇到网络或者其它因素影响，由于 TiDB 会周期性地更新缓存信息，正常情况下，最多 15 分钟左右生效。

如果授权表已被直接修改，则不会通知 TiDB 节点更新缓存，运行如下命令可使改动立即生效：

```
FLUSH PRIVILEGES;
```

详情参见权限管理。

14.10.4 TiDB 密码管理

为了保护用户密码的安全，从 TiDB v6.5.0 开始支持密码管理能力：

- 密码复杂度策略：要求用户设置强密码，以防止出现空密码、弱密码。
- 密码过期策略：要求用户定期修改密码。
- 密码重用策略：限制用户重复使用旧密码。
- 密码连续错误限制登录策略：连续多次密码错误导致登录失败后，临时锁定用户，限制该用户继续尝试登录。

14.10.4.1 TiDB 身份验证凭据存储

密码作为一种用户身份凭据，在用户登录到服务端时用于身份验证，确保用户身份的合法性。本文中描述的密码是指由 TiDB 生成、存储、验证的内部凭据，TiDB 将用户密码被存储到 mysql.user 系统表中，以下身份验证插件涉及本文的密码管理功能：

- mysql_native_password
- caching_sha2_password
- tidb_sm3_password

有关 TiDB 支持身份验证插件的更多信息，请查看可用的身份验证插件。

14.10.4.2 密码复杂度策略

在 TiDB 中，密码复杂度检查默认未开启。通过配置密码复杂度相关的系统变量，你可以开启密码复杂度检查，并确保为账户设置的密码符合密码复杂度策略。

密码复杂度策略支持以下功能：

- 对采用明文方式设置用户密码的 SQL 语句（包括 `CREATE USER`、`ALTER USER`、`SET PASSWORD`），系统会根据密码复杂度策略检查该密码，如果该密码不符合要求，则拒绝该密码。
- 可以使用 SQL 函数 `VALIDATE_PASSWORD_STRENGTH()` 评估给定密码的强度。

> 注意：
>
> - 对于 `CREATE USER` 语句，即使该账户最初被锁定，也必须提供满足密码复杂度策略的密码，否则将账户解锁后，该账户可以使用不符合密码复杂度策略的密码访问 TiDB。
> - 对密码复杂度策略的变更不影响已存在的密码，只会对新设置的密码产生影响。

通过以下 SQL 语句，你可以查看所有密码复杂度策略相关的系统变量：

```
mysql> SHOW VARIABLES LIKE 'validate_password.%';
+--------------------------------------+--------+
| Variable_name                        | Value  |
+--------------------------------------+--------+
| validate_password.check_user_name    | ON     |
| validate_password.dictionary         |        |
| validate_password.enable             | OFF    |
| validate_password.length             | 8      |
| validate_password.mixed_case_count   | 1      |
| validate_password.number_count       | 1      |
| validate_password.policy             | MEDIUM |
| validate_password.special_char_count | 1      |
+--------------------------------------+--------+
8 rows in set (0.00 sec)
```

关于这些变量的详细解释，请查阅系统变量文档。

14.10.4.2.1 配置密码复杂度策略

密码复杂度策略相关的系统变量的配置方式如下：

开启密码复杂度策略检查：

```
SET GLOBAL validate_password.enable = ON;
```

设置不允许密码与当前用户名相同：

```
SET GLOBAL validate_password.check_user_name = ON;
```

设置密码复杂度的检查等级为 LOW：

```
SET GLOBAL validate_password.policy = LOW;
```

设置密码最小长度为 10：

```
SET GLOBAL validate_password.length = 10;
```

设置密码中至少含有 2 个数字，至少含有 1 个大写和小写字符，至少含有 1 个特殊字符：

```
SET GLOBAL validate_password.number_count = 2;
SET GLOBAL validate_password.mixed_case_count = 1;
SET GLOBAL validate_password.special_char_count = 1;
```

设置密码字典功能，要求密码中不允许包含 mysql 或 abcd：

```
SET GLOBAL validate_password.dictionary = 'mysql;abcd';
```

> 注意：
>
> - validate_password.dictionary 是一个长字符串，长度不超过 1024，字符串内容可包含一个或多个在密码中不允许出现的单词，每个单词之间采用英文分号（;）分隔。
> - 密码字典功能进行单词比较时，不区字符分大小写。

14.10.4.2.2 密码复杂度检查示例

配置系统变量 validate_password.enable = ON 后，TiDB 将开启密码复杂度检查。以下为一些典型的检查示例：

按照默认密码复杂度策略，检测用户明文密码，若设置的密码不符合复杂度策略要求，则设置失败。

```
ALTER USER 'test'@'localhost' IDENTIFIED BY 'abc';
ERROR 1819 (HY000): Require Password Length: 8
```

TiDB 进行密码复杂度检查时，不检查散列后的密码。

```
ALTER USER 'test'@'localhost' IDENTIFIED WITH mysql_native_password AS '*0
    ↪ D3CED9BEC10A777AEC23CCC353A8C08A633045E';
Query OK, 0 rows affected (0.01 sec)
```

创建一个最初被锁定的账户时，也必须设置符合密码复杂度策略的密码，否则创建失败。

```
CREATE USER 'user02'@'localhost' ACCOUNT LOCK;
ERROR 1819 (HY000): Require Password Length: 8
```

14.10.4.2.3 密码强度评估函数

使用 VALIDATE_PASSWORD_STRENGTH() 函数评估给定密码的强度，该函数接受一个密码参数，并返回一个从 0（弱）到 100（强）的整数。

> **注意：**
>
> 密码强度是基于当前已配置的密码复杂度策略进行评估的，密码复杂度配置改变后，同一个密码的评估结果可能与之前不同。

VALIDATE_PASSWORD_STRENGTH() 函数使用示例如下：

```
SELECT VALIDATE_PASSWORD_STRENGTH('weak');
+-----------------------------------+
| VALIDATE_PASSWORD_STRENGTH('weak') |
+-----------------------------------+
|                                25 |
+-----------------------------------+
1 row in set (0.01 sec)

SELECT VALIDATE_PASSWORD_STRENGTH('lessweak$_@123');
+---------------------------------------------+
| VALIDATE_PASSWORD_STRENGTH('lessweak$_@123') |
+---------------------------------------------+
|                                          50 |
+---------------------------------------------+
1 row in set (0.01 sec)

SELECT VALIDATE_PASSWORD_STRENGTH('N0Tweak$_@123!');
+---------------------------------------------+
| VALIDATE_PASSWORD_STRENGTH('N0Tweak$_@123!') |
+---------------------------------------------+
|                                         100 |
+---------------------------------------------+
1 row in set (0.01 sec)
```

14.10.4.3 密码过期策略

TiDB 支持通过设置密码过期策略，要求用户定期修改密码，从而提高密码的安全性。可以手动将指定账户的密码设置为过期，也可以建立密码自动过期策略。自动过期策略分为全局级别和账户级别，管理员可以在全局级别建立密码过期策略，也可以使用账户级别密码过期策略覆盖全局级别策略。设置密码过期策略的权限要求如下：

- 具有 SUPER 或者 CREATE USER 权限的数据库管理员可以手动设置密码过期。

- 具有 SUPER 或者 CREATE USER 权限的数据库管理员可以设置账户级别自动密码过期策略。
- 具有 SUPER 或者 SYSTEM_VARIABLES_ADMIN 权限的数据库管理员可以设置全局级别自动密码过期策略。

14.10.4.3.1 手动密码过期

要手动设置账户密码过期，请使用 CREATE USER 或 ALTER USER 语句。

```
ALTER USER 'test'@'localhost' PASSWORD EXPIRE;
```

当账户密码被管理员手动设置过期后，必须修改该账户密码才能解除密码过期，不支持取消手动过期。

对于通过 CREATE ROLE 语句创建的角色，由于角色不需要设置密码，所以该角色对应的密码字段为空，此时对应的 password_expired 属性为 'Y'，即该角色的密码处于手动过期状态。如此设计的目的是防止出现角色锁定状态被解除后，该角色以空密码登录到 TiDB，当该角色被 ALTER USER ... ACCOUNT UNLOCK 命令解锁后，此时该角色处于可登录的状态，但是密码为空；因此 TiDB 通过 password_expired 属性，使得角色的密码处于手动过期状态，从而强制要求为该角色设置一个有效的密码。

```
CREATE ROLE testrole;
Query OK, 0 rows affected (0.01 sec)

SELECT user,password_expired,Account_locked FROM mysql.user WHERE user = 'testrole';
+----------+------------------+----------------+
| user     | password_expired | Account_locked |
+----------+------------------+----------------+
| testrole | Y                | Y              |
+----------+------------------+----------------+
1 row in set (0.02 sec)
```

14.10.4.3.2 自动密码过期

自动密码过期是基于密码使用期限和密码被允许的生存期来判断的。

- 密码使用期限：从最近一次密码更改日期到当前日期的时间间隔。系统表 mysql.user 中会记录最近一次修改密码的时间。
- 密码被允许的生存期：一个密码被设置后，其可以正常用于登录 TiDB 的天数。

如果密码使用期限大于其被允许的生存期，服务器会自动将密码视为已过期。

TiDB 支持在全局级别和账户级别设置自动密码过期：

- 全局级别自动密码过期

  你可以设置系统变量default_password_lifetime 来控制密码生存期。该变量默认值为 0，表示禁用自动密码过期。如果设置该变量的值为正整数 N，则表示允许的密码生存期为 N 天，即必须在 N 天之内更改密码。

  全局自动密码过期策略适用于所有未设置账户级别覆盖的账户。

  以下示例建立全局自动密码过期策略，密码有效期为 180 天：

```
SET GLOBAL default_password_lifetime = 180;
```

- 账户级别自动密码过期

  要为个人账户建立自动密码过期策略，请使用 CREATE USER 或 ALTER USER 语句的 PASSWORD EXPIRE 选项。

  以下示例要求用户密码每 90 天更改一次：

```
CREATE USER 'test'@'localhost' PASSWORD EXPIRE INTERVAL 90 DAY;
ALTER USER 'test'@'localhost' PASSWORD EXPIRE INTERVAL 90 DAY;
```

  在账户级别禁用自动密码过期策略：

```
CREATE USER 'test'@'localhost' PASSWORD EXPIRE NEVER;
ALTER USER 'test'@'localhost' PASSWORD EXPIRE NEVER;
```

  移除指定账户的账户级别自动密码过期策略，使其遵循于全局自动密码过期策略：

```
CREATE USER 'test'@'localhost' PASSWORD EXPIRE DEFAULT;
ALTER USER 'test'@'localhost' PASSWORD EXPIRE DEFAULT;
```

14.10.4.3.3 密码过期策略检查机制

当客户端连接成功后，服务端将按顺序进行以下检查，判断账号密码是否过期：

1. 服务器检查密码是否已手动过期。
2. 若密码没有手动过期，服务器根据自动密码过期策略检查密码使用期限是否大于其允许的生存期。如果是，服务器认为密码已过期。

14.10.4.3.4 密码过期处理机制

TiDB 支持密码过期策略控制。当密码过期后，服务器要么断开客户端的连接，要么将客户端限制为“沙盒模式”。“沙盒模式”下，TiDB 服务端接受密码过期账户的连接，但是连接成功后只允许该用户执行重置密码的操作。

TiDB 服务端可以控制是否将密码已过期用户的连接限制为“沙盒模式”。你可以在 TiDB 配置文件中的 [security] 部分，配置disconnect-on-expired-password 选项：

```
[security]
disconnect-on-expired-password = true
```

- 默认情况下，disconnect-on-expired-password 为 true，表示当密码过期后，服务器将直接断开客户端的连接。
- 如果配置 disconnect-on-expired-password 为 false，则服务端处于沙盒模式，服务端允许用户建立连接，但只能执行密码重置操作，密码重置后将允许用户正常执行各类 SQL 语句。

当 disconnect-on-expired-password 为 true 时，TiDB 将拒绝密码已过期用户的连接，此时可以通过如下方法修改密码：

- 普通用户密码过期，可以由管理员用户通过 SQL 语句修改该用户的密码。
- 管理员密码过期，可以由其他管理员用户通过 SQL 语句修改该用户的密码。
- 如果管理员密码过期，且无法寻求其他管理员帮助修改该用户的密码，此时可以采用 skip-grant-table 机制修改该用户密码，具体可参看忘记密码流程。

14.10.4.4 密码重用策略

TiDB 支持限制重复使用以前的密码。密码重用策略可以基于密码更改的次数或经过的时间，也可以同时基于两者。密码重用策略分为全局级别和账户级别。你可以在全局级别建立密码重用策略，也可以使用账户级别密码重用策略覆盖全局策略。

TiDB 会记录账户的历史密码，并限制从该历史记录中选择新密码：

- 如果密码重用策略基于密码更改次数，则新密码不得与指定数量的历史密码相同。例如，如果密码的最小更改次数设置为 3，则新密码不能与最近 3 个密码中的任何一个相同。
- 如果密码重用策略基于经过的时间，则新密码不得与历史记录中指定天数内使用过的密码相同。例如，如果密码重用间隔设置为 60，则新密码不能与最近 60 天内使用过的密码相同。

> 注意：
>
> 空密码不计入密码历史记录，可以随时重复使用。

14.10.4.4.1 全局级别密码重用策略

要在全局范围内建立密码重用策略，请使用password_history 和password_reuse_interval 系统变量。

例如，建立全局密码重用策略，禁止重复使用最近 6 个密码或最近 365 天的密码：

```
SET GLOBAL password_history = 6;
SET GLOBAL password_reuse_interval = 365;
```

全局密码重用策略适用于所有未在账户级别设置过密码重用策略的账户。

14.10.4.4.2 账户级别密码重用策略

要建立账户级别密码重用策略，请使用 CREATE USER 或 ALTER USER 语句的 PASSWORD HISTORY 和 PASSWORD REUSE INTERVAL 选项。

示例：

禁止重复使用最近 5 次使用过的密码：

```
CREATE USER 'test'@'localhost' PASSWORD HISTORY 5;
ALTER USER 'test'@'localhost' PASSWORD HISTORY 5;
```

禁止重复使用最近 365 天内使用过的密码：

```
CREATE USER 'test'@'localhost' PASSWORD REUSE INTERVAL 365 DAY;
ALTER USER 'test'@'localhost' PASSWORD REUSE INTERVAL 365 DAY;
```

如需组合两种类型的重用策略，请一起使用 PASSWORD HISTORY 和 PASSWORD REUSE INTERVAL：

```
CREATE USER 'test'@'localhost'
  PASSWORD HISTORY 5
  PASSWORD REUSE INTERVAL 365 DAY;
ALTER USER 'test'@'localhost'
  PASSWORD HISTORY 5
  PASSWORD REUSE INTERVAL 365 DAY;
```

移除指定账户的账户级别密码重用策略，使其遵循于全局密码重用策略：

```
CREATE USER 'test'@'localhost'
  PASSWORD HISTORY DEFAULT
  PASSWORD REUSE INTERVAL DEFAULT;
ALTER USER 'test'@'localhost'
  PASSWORD HISTORY DEFAULT
  PASSWORD REUSE INTERVAL DEFAULT;
```

> 注意：
>
> - 如果多次设置密码重用策略，则最后一次设置的值生效。
> - PASSWORD HISTORY 和 PASSWORD REUSE INTERVAL 选项的默认值为 0，表示禁用该项重用策略。
> - 在修改用户名时，TiDB 会将 mysql.password_history 系统表中原用户名的历史密码记录迁移到新用户名的记录中。

14.10.4.5 密码连续错误限制登录策略

TiDB 支持限制账户持续尝试登录，防止用户密码被暴力破解。当账户连续登录失败次数过多时，账户将被临时锁定。

> 注意：
>
> - 只支持账户级别的密码连续错误限制登录策略，不支持全局级别的策略。
> - 登录失败是指客户端在连接尝试期间未能提供正确的密码，不包括由于未知用户或网络问题等原因而导致的连接失败。
> - 对用户启用密码连续错误限制登录策略后，将增加该用户登录时的检查步骤，此时会影响该用户登录相关操作的性能，尤其是高并发登录场景。

14.10.4.5.1 配置密码连续错误限制登录策略

每个账户的登录失败次数和锁定时间是可配置的，你可以使用 CREATE USER、ALTER USER 语句的 FAILED_LOGIN_ATTEMPTS 和 PASSWORD_LOCK_TIME 选项。FAILED_LOGIN_ATTEMPTS 和 PASSWORD_LOCK_TIME 选项的可设置值如下：

- FAILED_LOGIN_ATTEMPTS：N。表示连续登录失败 N 次后，账户将被临时锁定。N 取值范围为 0 到 32767。
- PASSWORD_LOCK_TIME：N | UNBOUNDED。N 表示登录失败后，账户将被临时锁定 N 天。UNBOUNDED 表明锁定时间无限期，账户必须被手动解锁。N 取值范围为 0 到 32767。

> 注意：
>
> - 允许单条 SQL 语句只设置 FAILED_LOGIN_ATTEMPTS 或 PASSWORD_LOCK_TIME 中的一个选项，这时密码连续错误限制登录策略不会实质生效。
> - 只有当账户的 FAILED_LOGIN_ATTEMPTS 和 PASSWORD_LOCK_TIME 都不为 0 时，系统才会跟踪该账户的失败登录次数并执行临时锁定。

配置密码连续错误限制登录策略的示例如下：

新建一个用户，并配置密码连续错误限制登录策略，当该用户密码连续错误 3 次时，临时锁定 3 天：

```
CREATE USER 'test1'@'localhost' IDENTIFIED BY 'password' FAILED_LOGIN_ATTEMPTS 3
    ↪ PASSWORD_LOCK_TIME 3;
```

修改用户的密码连续错误限制登录策略，当该用户密码连续错误 4 次时，无限期锁定，直到账户被手动解锁：

```
ALTER USER 'test2'@'localhost' FAILED_LOGIN_ATTEMPTS 4 PASSWORD_LOCK_TIME UNBOUNDED;
```

关闭用户的密码连续错误限制登录策略：

```
ALTER USER 'test3'@'localhost' FAILED_LOGIN_ATTEMPTS 0 PASSWORD_LOCK_TIME 0;
```

14.10.4.5.2 锁定账户解锁

在以下场景中，TiDB 将重置用户密码连续错误次数的计数：

- 对该用户执行 ALTER USER ... ACCOUNT UNLOCK 解锁命令时。
- 该用户登录成功时。

当用户因密码连续多次错误触发账户锁定后，以下情况下可以解锁账户：

- 该用户锁定时间结束，这种情况下，用户的自动锁定标识将在下次登录尝试时重置。
- 对该用户执行 ALTER USER ... ACCOUNT UNLOCK 解锁命令时。

注意：

当用户因密码连续失败次数达到设定值而被自动锁定时，如果修改该账户的密码连续错误限制登录策略，需要注意：

- 修改该用户的登录失败次数 FAILED_LOGIN_ATTEMPTS，用户的自动锁定状态不会改变。修改后的连续登录失败次数，将在用户解锁后再次尝试登录时生效。
- 修改该用户的锁定时间 PASSWORD_LOCK_TIME，用户的自动锁定状态不会改变。修改后的用户锁定时间，将在账户再次登录尝试时检查是否满足此时的锁定时间要求，如果锁定时间结束就会解锁该用户。

14.10.5 基于角色的访问控制

TiDB 的基于角色的访问控制 (RBAC) 系统的实现类似于 MySQL 8.0 的 RBAC 系统。TiDB 兼容大部分 MySQL RBAC 系统的语法。

本文档主要介绍 TiDB 基于角色的访问控制相关操作及实现。

14.10.5.1 角色访问控制相关操作

角色是一系列权限的集合。用户可以创建角色、删除角色、将权限赋予角色；也可以将角色授予给其他用户，被授予的用户在启用角色后，可以得到角色所包含的权限。

14.10.5.1.1 创建角色

创建角色 app_developer，app_read 和 app_write：

```
CREATE ROLE 'app_developer', 'app_read', 'app_write';
```

角色名的格式和规范可以参考TiDB 用户账户管理。

角色会被保存在 mysql.user 表中，角色名称的主机名部分（如果省略）默认为 '%'。如果表中有同名角色或用户，角色会创建失败并报错。创建角色的用户需要拥有 CREATE ROLE 或 CREATE USER 权限。

14.10.5.1.2 授予角色权限

为角色授予权限和为用户授予权限操作相同，可参考TiDB 权限管理。

为 app_read 角色授予数据库 app_db 的读权限：

```
GRANT SELECT ON app_db.* TO 'app_read'@'%';
```

为 app_write 角色授予数据库 app_db 的写权限：

```
GRANT INSERT, UPDATE, DELETE ON app_db.* TO 'app_write'@'%';
```

为 app_developer 角色授予 app_db 数据库的全部权限：

```
GRANT ALL ON app_db.* TO 'app_developer';
```

14.10.5.1.3 将角色授予给用户

假设有一个用户拥有开发者角色，可以对 app_db 的所有操作权限；另外有两个用户拥有 app_db 的只读权限；还有一个用户拥有 app_db 的读写权限。

首先用 CREATE USER 来创建用户。

```
CREATE USER 'dev1'@'localhost' IDENTIFIED BY 'dev1pass';
CREATE USER 'read_user1'@'localhost' IDENTIFIED BY 'read_user1pass';
CREATE USER 'read_user2'@'localhost' IDENTIFIED BY 'read_user2pass';
CREATE USER 'rw_user1'@'localhost' IDENTIFIED BY 'rw_user1pass';
```

然后使用 GRANT 授予用户对应的角色。

```
GRANT 'app_developer' TO 'dev1'@'localhost';
GRANT 'app_read' TO 'read_user1'@'localhost', 'read_user2'@'localhost';
GRANT 'app_read', 'app_write' TO 'rw_user1'@'localhost';
```

用户执行将角色授予给其他用户或者收回角色的命令，需要用户拥有 SUPER 权限。将角色授予给用户时并不会启用该角色，启用角色需要额外的操作。

以下操作可能会形成一个“关系环”：

```
CREATE USER 'u1', 'u2';
CREATE ROLE 'r1', 'r2';

GRANT 'u1' TO 'u1';
GRANT 'r1' TO 'r1';

GRANT 'r2' TO 'u2';
GRANT 'u2' TO 'r2';
```

TiDB 允许这种多层授权关系存在，可以使用多层授权关系实现权限继承。

14.10.5.1.4 查看角色拥有的权限

可以通过 SHOW GRANTS 语句查看用户被授予了哪些角色。当用户查看其他用户权限相关信息时，需要对 mysql 数据库拥有 SELECT 权限。

```
SHOW GRANTS FOR 'dev1'@'localhost';
```

```
+-------------------------------------------------+
| Grants for dev1@localhost                       |
+-------------------------------------------------+
| GRANT USAGE ON *.* TO `dev1`@`localhost`        |
```

```
| GRANT `app_developer`@`%` TO `dev1`@`localhost` |
+-------------------------------------------------+
```

可以通过使用 SHOW GRANTS 的 USING 选项来查看角色对应的权限。

```
SHOW GRANTS FOR 'dev1'@'localhost' USING 'app_developer';
```

```
+-----------------------------------------------------------+
| Grants for dev1@localhost                                 |
+-----------------------------------------------------------+
| GRANT USAGE ON *.* TO `dev1`@`localhost`                  |
| GRANT ALL PRIVILEGES ON `app_db`.* TO `dev1`@`localhost` |
| GRANT `app_developer`@`%` TO `dev1`@`localhost`           |
+-----------------------------------------------------------+
```

```
SHOW GRANTS FOR 'rw_user1'@'localhost' USING 'app_read', 'app_write';
```

```
+-------------------------------------------------------------------------------+
| Grants for rw_user1@localhost                                                 |
+-------------------------------------------------------------------------------+
| GRANT USAGE ON *.* TO `rw_user1`@`localhost`                                  |
| GRANT SELECT, INSERT, UPDATE, DELETE ON `app_db`.* TO `rw_user1`@`localhost` |
| GRANT `app_read`@`%`,`app_write`@`%` TO `rw_user1`@`localhost`                |
+-------------------------------------------------------------------------------+
```

```
SHOW GRANTS FOR 'read_user1'@'localhost' USING 'app_read';
```

```
+-----------------------------------------------------------+
| Grants for read_user1@localhost                           |
+-----------------------------------------------------------+
| GRANT USAGE ON *.* TO `read_user1`@`localhost`            |
| GRANT SELECT ON `app_db`.* TO `read_user1`@`localhost` |
| GRANT `app_read`@`%` TO `read_user1`@`localhost`          |
+-----------------------------------------------------------+
```

可以使用 SHOW GRANTS 或 SHOW GRANTS FOR CURRENT_USER() 查看当前用户的权限。这两个语句有细微的差异，SHOW GRANTS 会显示当前用户的启用角色的权限，而 SHOW GRANTS FOR CURRENT_USER() 则不会显示启用角色的权限。

14.10.5.1.5 设置默认启用角色

角色在授予给用户之后，并不会生效；只有在用户启用了某些角色之后，才可以使用角色拥有的权限。

可以对用户设置默认启用的角色；用户在登录时，默认启用的角色会被自动启用。

```
SET DEFAULT ROLE
    {NONE | ALL | role [, role ] ...}
    TO user [, user ]
```

比如将 app_read 和 app_wirte 设置为 rw_user1@localhost 的默认启用角色：

```
SET DEFAULT ROLE app_read, app_write TO 'rw_user1'@'localhost';
```

将 dev1@localhost 的所有角色，设为其默认启用角色：

```
SET DEFAULT ROLE ALL TO 'dev1'@'localhost';
```

关闭 dev1@localhost 的所有默认启用角色：

```
SET DEFAULT ROLE NONE TO 'dev1'@'localhost';
```

需要注意的是，设置为默认启用角色的角色必须已经授予给那个用户。

14.10.5.1.6 在当前 session 启用角色

除了使用 SET DEFAULT ROLE 启用角色外，TiDB 还提供让用户在当前 session 启用某些角色的功能。

```
SET ROLE {
    DEFAULT
  | NONE
  | ALL
  | ALL EXCEPT role [, role ] ...
  | role [, role ] ...
}
```

例如，登录 rw_user1 后，为当前用户启用角色 app_read 和 app_write，仅在当前 session 有效：

```
SET ROLE 'app_read', 'app_write';
```

启用当前用户的默认角色：

```
SET ROLE DEFAULT
```

启用授予给当前用户的所有角色：

```
SET ROLE ALL
```

不启用任何角色：

```
SET ROLE NONE
```

启用除 app_read 外的角色：

```
SET ROLE ALL EXCEPT 'app_read'
```

> **注意：**
>
> 使用 SET ROLE 启用的角色只有在当前 session 才会有效。

14.10.5.1.7 查看当前启用角色

当前用户可以通过 CURRENT_ROLE() 函数查看当前用户启用了哪些角色。

例如，先对 rw_user1'@'localhost 设置默认角色：

```
SET DEFAULT ROLE ALL TO 'rw_user1'@'localhost';
```

用 rw_user1@localhost 登录后：

```
SELECT CURRENT_ROLE();
```

```
+--------------------------------+
| CURRENT_ROLE()                 |
+--------------------------------+
| `app_read`@`%`,`app_write`@`%` |
+--------------------------------+
```

```
SET ROLE 'app_read'; SELECT CURRENT_ROLE();
```

```
+----------------+
| CURRENT_ROLE() |
+----------------+
| `app_read`@`%` |
+----------------+
```

14.10.5.1.8 收回角色

解除角色 app_read 与用户 read_user1@localhost、read_user2@localhost 的授权关系。

```
REVOKE 'app_read' FROM 'read_user1'@'localhost', 'read_user2'@'localhost';
```

解除角色 app_read、app_write 与用户 rw_user1@localhost 的授权关系。

```
REVOKE 'app_read', 'app_write' FROM 'rw_user1'@'localhost';
```

解除角色授权具有原子性，如果在撤销授权操作中失败会回滚。

14.10.5.1.9 收回权限

REVOKE 语句与 GRANT 对应，可以使用 REVOKE 来撤销 app_write 的权限。

```
REVOKE INSERT, UPDATE, DELETE ON app_db.* FROM 'app_write';
```

具体可参考TiDB 权限管理。

14.10.5.1.10 删除角色

删除角色 app_read 和 app_write：

```
DROP ROLE 'app_read', 'app_write';
```

这个操作会清除角色在 mysql.user 表里面的记录项，并且清除在授权表里面的相关记录，解除和其相关的授权关系。执行删除角色的用户需要拥有 DROP ROLE 或 DROP USER 权限。

14.10.5.1.11 授权表

在原有的四张系统权限表的基础上，角色访问控制引入了两张新的系统表：

- mysql.role_edges：记录角色与用户的授权关系
- mysql.default_roles：记录每个用户默认启用的角色

以下是 mysql.role_edges 所包含的数据。

```
select * from mysql.role_edges;
```

```
+-----------+-----------+---------+---------+-------------------+
| FROM_HOST | FROM_USER | TO_HOST | TO_USER | WITH_ADMIN_OPTION |
+-----------+-----------+---------+---------+-------------------+
| %         | r_1       | %       | u_1     | N                 |
+-----------+-----------+---------+---------+-------------------+
1 row in set (0.00 sec)
```

其中 FROM_HOST 和 FROM_USER 分别表示角色的主机名和用户名，TO_HOST 和 TO_USER 分别表示被授予角色的用户的主机名和用户名。

mysql.default_roles 中包含了每个用户默认启用了哪些角色。

```
select * from mysql.default_roles;
```

```
+------+------+-------------------+-------------------+
| HOST | USER | DEFAULT_ROLE_HOST | DEFAULT_ROLE_USER |
+------+------+-------------------+-------------------+
| %    | u_1  | %                 | r_1               |
| %    | u_1  | %                 | r_2               |
+------+------+-------------------+-------------------+
2 rows in set (0.00 sec)
```

HOST 和 USER 分别表示用户的主机名和用户名，DEFAULT_ROLE_HOST 和 DEFAULT_ROLE_USER 分别表示默认启用的角色的主机名和用户名。

14.10.5.1.12 其他

由于基于角色的访问控制模块和用户管理以及权限管理结合十分紧密，因此需要参考一些操作的细节：

- TiDB 权限管理
- TiDB 用户账户管理

14.10.6 TiDB 证书鉴权使用指南

TiDB 支持基于证书鉴权的登录方式。采用这种方式，TiDB 对不同用户签发证书，使用加密连接来传输数据，并在用户登录时验证证书。相比 MySQL 用户常用的用户名密码验证方式，与 MySQL 相兼容的证书鉴权方式更安全，因此越来越多的用户使用证书鉴权来代替用户名密码验证。

在 TiDB 上使用证书鉴权的登录方法，可能需要进行以下操作：

- 创建安全密钥和证书
- 配置 TiDB 和客户端使用的证书
- 配置登录时需要校验的用户证书信息
- 更新和替换证书

本文介绍了如何进行证书鉴权的上述几个操作。

14.10.6.1 创建安全密钥和证书

目前推荐使用 OpenSSL 来生成密钥和证书，生成证书的过程和为 TiDB 客户端服务端间通信开启加密传输过程类似，下面更多演示如何在证书中配置更多需校验的属性字段。

14.10.6.1.1 生成 CA 密钥和证书

1. 执行以下命令生成 CA 密钥：

```
sudo openssl genrsa 2048 > ca-key.pem
```

命令执行后输出以下结果：

```
Generating RSA private key, 2048 bit long modulus (2 primes)
.....................+++++
.............................................+++++
e is 65537 (0x010001)
```

2. 执行以下命令生成该密钥对应的证书：

```
sudo openssl req -new -x509 -nodes -days 365000 -key ca-key.pem -out ca-cert.pem
```

3. 输入证书细节信息，示例如下：

```
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:California
Locality Name (eg, city) []:San Francisco
Organization Name (eg, company) [Internet Widgits Pty Ltd]:PingCAP Inc.
Organizational Unit Name (eg, section) []:TiDB
Common Name (e.g. server FQDN or YOUR name) []:TiDB admin
Email Address []:s@pingcap.com
```

> 注意：
> 以上信息中，: 后的文字为用户输入的信息。

14.10.6.1.2 生成服务端密钥和证书

1. 执行以下命令生成服务端的密钥：

```
sudo openssl req -newkey rsa:2048 -days 365000 -nodes -keyout server-key.pem -out server-req
    ↪ .pem
```

2. 输入证书细节信息，示例如下：

```
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:California
Locality Name (eg, city) []:San Francisco
Organization Name (eg, company) [Internet Widgits Pty Ltd]:PingCAP Inc.
Organizational Unit Name (eg, section) []:TiKV
Common Name (e.g. server FQDN or YOUR name) []:TiKV Test Server
Email Address []:k@pingcap.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

3. 执行以下命令生成服务端的 RSA 密钥：

```
sudo openssl rsa -in server-key.pem -out server-key.pem
```

输出结果如下：

```
writing RSA key
```

4. 使用 CA 证书签名来生成服务端的证书：

```
sudo openssl x509 -req -in server-req.pem -days 365000 -CA ca-cert.pem -CAkey ca-key.pem -
    ↪ set_serial 01 -out server-cert.pem
```

输出结果示例如下：

```
Signature ok
subject=C = US, ST = California, L = San Francisco, O = PingCAP Inc., OU = TiKV, CN = TiKV
    ↪ Test Server, emailAddress = k@pingcap.com
Getting CA Private Key
```

> 注意：
> 以上结果中，用户登录时 TiDB 将强制检查 `subject` 部分的信息是否一致。

14.10.6.1.3 生成客户端密钥和证书

生成服务端密钥和证书后，需要生成客户端使用的密钥和证书。通常需要为不同的用户生成不同的密钥和证书。

1. 执行以下命令生成客户端的密钥：

```
sudo openssl req -newkey rsa:2048 -days 365000 -nodes -keyout client-key.pem -out client-req
    ↪ .pem
```

2. 输入证书细节信息，示例如下：

```
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:California
Locality Name (eg, city) []:San Francisco
Organization Name (eg, company) [Internet Widgits Pty Ltd]:PingCAP Inc.
Organizational Unit Name (eg, section) []:TiDB
Common Name (e.g. server FQDN or YOUR name) []:tpch-user1
Email Address []:zz@pingcap.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

3. 执行以下命令生成客户端 RSA 证书：

```
sudo openssl rsa -in client-key.pem -out client-key.pem
```

以上命令的输出结果如下：

```
writing RSA key
```

4. 执行以下命令，使用 CA 证书签名来生成客户端证书：

```
sudo openssl x509 -req -in client-req.pem -days 365000 -CA ca-cert.pem -CAkey ca-key.pem -
    ↪ set_serial 01 -out client-cert.pem
```

输出结果示例如下：

```
Signature ok
subject=C = US, ST = California, L = San Francisco, O = PingCAP Inc., OU = TiDB, CN = tpch-
    ↪ user1, emailAddress = zz@pingcap.com
Getting CA Private Key
```

> 注意：
> 以上结果中，`subject` 部分后的信息会被用来在 `require` 中配置和要求验证。

14.10.6.1.4 验证证书

执行以下命令验证证书：

```
openssl verify -CAfile ca-cert.pem server-cert.pem client-cert.pem
```

如果验证通过，会显示以下信息：

```
server-cert.pem: OK
client-cert.pem: OK
```

14.10.6.2 配置 TiDB 和客户端使用证书

在生成证书后，需要在 TiDB 中配置服务端所使用的证书，同时让客户端程序使用客户端证书。

14.10.6.2.1 配置 TiDB 服务端

修改 TiDB 配置文件中的 `[security]` 段。这一步指定 CA 证书、服务端密钥和服务端证书存放的路径。可将 `path/to/server-cert.pem`、`path/to/server-key.pem` 和 `path/to/ca-cert.pem` 替换成实际的路径。

```
[security]
ssl-cert ="path/to/server-cert.pem"
ssl-key ="path/to/server-key.pem"
ssl-ca="path/to/ca-cert.pem"
```

启动 TiDB 日志。如果日志中有以下内容，即代表配置生效：

```
[INFO] [server.go:264] ["secure connection is enabled"] ["client verification enabled"=true]
```

14.10.6.2.2 配置客户端程序

配置客户端程序，让客户端使用客户端密钥和证书来登录 TiDB。

以 MySQL 客户端为例，可以通过指定 ssl-cert、ssl-key、ssl-ca 来使用新的 CA 证书、客户端密钥和证书：

```
mysql -utest -h0.0.0.0 -P4000 --ssl-cert /path/to/client-cert.new.pem --ssl-key /path/to/client-
    ↪ key.new.pem --ssl-ca /path/to/ca-cert.pem
```

> 注意：
>
> /path/to/client-cert.new.pem、/path/to/client-key.new.pem 和 /path/to/ca-cert.pem 是 CA 证书、客户端密钥和客户端存放的路径。可将以上命令中的这些部分替换为实际的路径。

14.10.6.3 配置登录时需要校验的用户证书信息

使用客户端连接 TiDB 进行授权配置。先获取需要验证的用户证书信息，再对这些信息进行配置。

14.10.6.3.1 获取用户证书信息

用户证书信息可由 require subject、require issuer、require san 和 require cipher 来指定，用于检查 X.509 certificate attributes。

- require subject：指定用户在连接时需要提供客户端证书的 subject 内容。指定该选项后，不需要再配置 require ssl 或 x509。配置内容对应生成客户端密钥和证书中的录入信息。

  可以执行以下命令来获取该项的信息：

  ```
  openssl x509 -noout -subject -in client-cert.pem | sed 's/.\{8\}//'  | sed 's/, /\//g' | sed
      ↪  's/ = /=/g' | sed 's/^/\//'
  ```

- require issuer：指定签发用户证书的 CA 证书的 subject 内容。配置内容对应生成 CA 密钥和证书中的录入信息。

  可以执行以下命令来获取该项的信息：

  ```
  openssl x509 -noout -subject -in ca-cert.pem | sed 's/.\{8\}//'  | sed 's/, /\//g' | sed 's/
      ↪  = /=/g' | sed 's/^/\//'
  ```

- require san：指定签发用户证书的 CA 证书的 Subject Alternative Name 内容。配置内容对应生成客户端证书使用的openssl.cnf 配置文件的 alt_names 信息。

  - 可以执行以下命令来获取已生成证书中的 require san 项的信息：

    ```
    openssl x509 -noout -extensions subjectAltName -in client.crt
    ```

  - require san 目前支持以下 Subject Alternative Name 检查项：

    * URI
    * IP
    * DNS
  - 多个检查项可通过逗号连接后进行配置。例如，对用户 u1 进行以下配置：

```
create user 'u1'@'%' require san 'DNS:d1,URI:spiffe://example.org/myservice1,URI:spiffe
    ↪ ://example.org/myservice2'
```

    以上配置只允许用户 u1 使用 URI 项为 spiffe://example.org/myservice1 或 spiffe://example.org
    ↪ /myservice2、DNS 项为 d1 的证书登录 TiDB。

- require cipher：配置该项检查客户端支持的 cipher method。可以使用以下语句来查看支持的列表：

```
SHOW SESSION STATUS LIKE 'Ssl_cipher_list';
```

14.10.6.3.2 配置用户证书信息

获取用户证书信息（require subject, require issuer、require san 和 require cipher）后，可在创建用户、赋予权限或更改用户时配置用户证书信息。将以下命令中的 <replaceable> 替换为对应的信息。可以选择配置其中一项或多项，使用空格或 and 分隔。

- 可以在创建用户 (create user) 时配置登录时需要校验的证书信息：

```
create user 'u1'@'%'  require issuer '<replaceable>' subject '<replaceable>' san '<
    ↪ replaceable>' cipher '<replaceable>';
```

- 可以在赋予权限 (grant) 时配置登录时需要校验的证书信息：

```
grant all on *.* to 'u1'@'%' require issuer '<replaceable>' subject '<replaceable>' san '<
    ↪ replaceable>' cipher '<replaceable>';
```

- 还可以在修改已有用户 (alter user) 时配置登录时需要校验的证书信息：

```
alter user 'u1'@'%' require issuer '<replaceable>' subject '<replaceable>' san '<replaceable
    ↪ >' cipher '<replaceable>';
```

配置完成后，用户在登录时 TiDB 会验证以下内容：

- 使用 SSL 登录，且证书为服务器配置的 CA 证书所签发
- 证书的 Issuer 信息和权限配置里的信息相匹配
- 证书的 Subject 信息和权限配置里的信息相匹配
- 证书的 Subject Alternative Name 信息和权限配置里的信息相匹配

全部验证通过后用户才能登录，否则会报 ERROR 1045 (28000): Access denied 错误。登录后，可以通过以下命令来查看当前链接是否使用证书登录、TLS 版本和 Cipher 算法。

连接 MySQL 客户端并执行：

```
\s
```

返回结果如下：

```
--------------
mysql  Ver 15.1 Distrib 10.4.10-MariaDB, for Linux (x86_64) using readline 5.1

Connection id:       1
Current database:    test
Current user:        root@127.0.0.1
SSL:                 Cipher in use is TLS_AES_256_GCM_SHA384
```

然后执行：

```
show variables like '%ssl%';
```

返回结果如下：

```
+---------------+---------------------------------+
| Variable_name | Value                           |
+---------------+---------------------------------+
| ssl_cert      | /path/to/server-cert.pem        |
| ssl_ca        | /path/to/ca-cert.pem            |
| have_ssl      | YES                             |
| have_openssl  | YES                             |
| ssl_key       | /path/to/server-key.pem         |
+---------------+---------------------------------+
6 rows in set (0.067 sec)
```

14.10.6.4 更新和替换证书

证书和密钥通常会周期性更新。下文介绍更新密钥和证书的流程。

CA 证书是客户端和服务端相互校验的依据，所以如果需要替换 CA 证书，则需要生成一个组合证书来在替换期间同时支持客户端和服务器上新旧证书的验证，并优先替换客户端和服务端的 CA 证书，再替换客户端和服务端的密钥和证书。

14.10.6.4.1 更新 CA 密钥和证书

1. 以替换 CA 密钥为例（假设 `ca-key.pem` 被盗），将旧的 CA 密钥和证书进行备份：

    ```
    mv ca-key.pem ca-key.old.pem && \
    mv ca-cert.pem ca-cert.old.pem
    ```

2. 生成新的 CA 密钥：

```
sudo openssl genrsa 2048 > ca-key.pem
```

3. 用新的密钥生成新的 CA 证书：

```
sudo openssl req -new -x509 -nodes -days 365000 -key ca-key.pem -out ca-cert.new.pem
```

> 注意：
> 生成新的 CA 证书是为了替换密钥和证书，保证在线用户不受影响。所以以上命令中填写的附加信息必须与已配置的 `require issuer` 信息一致。

4. 生成组合 CA 证书：

```
cat ca-cert.new.pem ca-cert.old.pem > ca-cert.pem
```

之后使用新生成的组合 CA 证书并重启 TiDB Server，此时服务端可以同时接受和使用新旧 CA 证书。

之后先将所有客户端用的 CA 证书也替换为新生成的组合 CA 证书，使客户端能同时和使用新旧 CA 证书。

14.10.6.4.2 更新客户端密钥和证书

> 注意：
> 需要将集群中所有服务端和客户端使用的 CA 证书都替换为新生成的组合 CA 证书后才能开始进行以下步骤。

1. 生成新的客户端 RSA 密钥：

```
sudo openssl req -newkey rsa:2048 -days 365000 -nodes -keyout client-key.new.pem -out client
    ↪ -req.new.pem && \
sudo openssl rsa -in client-key.new.pem -out client-key.new.pem
```

> 注意：
> 以上命令是为了替换密钥和证书，保证在线用户不受影响，所以以上命令中填写的附加信息必须与已配置的 `require subject` 信息一致。

2. 使用新的组合 CA 证书和新 CA 密钥生成新客户端证书：

```
sudo openssl x509 -req -in client-req.new.pem -days 365000 -CA ca-cert.pem -CAkey ca-key.pem
    ↪  -set_serial 01 -out client-cert.new.pem
```

3. 让客户端使用新的客户端密钥和证书来连接 TiDB（以 MySQL 客户端为例）：

```
mysql -utest -h0.0.0.0 -P4000 --ssl-cert /path/to/client-cert.new.pem --ssl-key /path/to/
    ↪ client-key.new.pem --ssl-ca /path/to/ca-cert.pem
```

> 注意：
>
> /path/to/client-cert.new.pem、/path/to/client-key.new.pem 和 /path/to/ca-cert.pem 是 CA 证书、客户端密钥和客户端存放的路径。可将以上命令中的这些部分替换为实际的路径。

14.10.6.4.3 更新服务端密钥和证书

1. 生成新的服务端 RSA 密钥：

```
sudo openssl req -newkey rsa:2048 -days 365000 -nodes -keyout server-key.new.pem -out server
    ↪ -req.new.pem && \
sudo openssl rsa -in server-key.new.pem -out server-key.new.pem
```

2. 使用新的组合 CA 证书和新 CA 密钥生成新服务端证书：

```
sudo openssl x509 -req -in server-req.new.pem -days 365000 -CA ca-cert.pem -CAkey ca-key.pem
    ↪  -set_serial 01 -out server-cert.new.pem
```

3. 配置 TiDB 使用上面新生成的服务端密钥和证书并重启。参见配置 TiDB 服务端。

14.11 SQL

14.11.1 SQL 语言结构和语法

14.11.1.1 属性

14.11.1.1.1 AUTO_INCREMENT

本文介绍列属性 AUTO_INCREMENT 的基本概念、实现原理、自增相关的特性，以及使用限制。

> 注意：
>
> 使用 AUTO_INCREMENT 可能会给生产环境带热点问题，因此推荐使用AUTO_RANDOM 代替。详情请参考TiDB 热点问题处理。

基本概念

AUTO_INCREMENT 是用于自动填充缺省列值的列属性。当 INSERT 语句没有指定 AUTO_INCREMENT 列的具体值时，系统会自动地为该列分配一个值。

出于性能原因，自增编号是系统批量分配给每台 TiDB 服务器的值（默认 3 万个值），因此自增编号能保证唯一性，但分配给 INSERT 语句的值仅在单台 TiDB 服务器上具有单调性。

> 注意：
>
> 如果要求自增编号在所有 TiDB 实例上具有单调性，并且你的 TiDB 版本在 v6.5.0 及以上，推荐使用MySQL 兼容模式。

```
CREATE TABLE t(id int PRIMARY KEY AUTO_INCREMENT, c int);
```

```
INSERT INTO t(c) VALUES (1);
INSERT INTO t(c) VALUES (2);
INSERT INTO t(c) VALUES (3), (4), (5);
```

```
SELECT * FROM t;
+----+---+
| id | c |
+----+---+
| 1  | 1 |
| 2  | 2 |
| 3  | 3 |
| 4  | 4 |
| 5  | 5 |
+----+---+
5 rows in set (0.01 sec)
```

此外，AUTO_INCREMENT 还支持显式指定列值的插入语句，此时 TiDB 会保存显式指定的值：

```
INSERT INTO t(id, c) VALUES (6, 6);
```

```
SELECT * FROM t;
+----+---+
| id | c |
+----+---+
| 1  | 1 |
| 2  | 2 |
| 3  | 3 |
| 4  | 4 |
| 5  | 5 |
| 6  | 6 |
```

```
+----+---+
6 rows in set (0.01 sec)
```

以上用法和 MySQL 的 AUTO_INCREMENT 用法一致。但在隐式分配的具体值方面，TiDB 和 MySQL 之间具有较为显著的差异。

实现原理

TiDB 实现 AUTO_INCREMENT 隐式分配的原理是，对于每一个自增列，都使用一个全局可见的键值对用于记录当前已分配的最大 ID。由于分布式环境下的节点通信存在一定开销，为了避免写请求放大的问题，每个 TiDB 节点在分配 ID 时，都申请一段 ID 作为缓存，用完之后再去取下一段，而不是每次分配都向存储节点申请。例如，对于以下新建的表：

```
CREATE TABLE t(id int UNIQUE KEY AUTO_INCREMENT, c int);
```

假设集群中有两个 TiDB 实例 A 和 B，如果向 A 和 B 分别对 t 执行一条插入语句：

```
INSERT INTO t (c) VALUES (1)
```

实例 A 可能会缓存 [1,30000] 的自增 ID，而实例 B 则可能缓存 [30001,60000] 的自增 ID。各自实例缓存的 ID 将随着执行将来的插入语句被作为缺省值，顺序地填充到 AUTO_INCREMENT 列中。

基本特性

唯一性保证

> 警告：
>
> 在集群中有多个 TiDB 实例时，如果表结构中有自增 ID，建议不要混用显式插入和隐式分配（即自增列的缺省值和自定义值），否则可能会破坏隐式分配值的唯一性。

例如在上述示例中，依次执行如下操作：

1. 客户端向实例 B 插入一条将 id 设置为 2 的语句 INSERT INTO t VALUES (2, 1)，并执行成功。
2. 客户端向实例 A 发送 INSERT 语句 INSERT INTO t (c)(1)，这条语句中没有指定 id 的值，所以会由 A 分配。当前 A 缓存了 [1, 30000] 这段 ID，可能会分配 2 为自增 ID 的值，并把本地计数器加 1。而此时数据库中已经存在 id 为 2 的数据，最终返回 Duplicated Error 错误。

单调性保证

TiDB 保证 AUTO_INCREMENT 自增值在单台服务器上单调递增。以下示例在一台服务器上生成连续的 AUTO_INCREMENT 自增值 1-3：

```
CREATE TABLE t (a int PRIMARY KEY AUTO_INCREMENT, b timestamp NOT NULL DEFAULT NOW());
INSERT INTO t (a) VALUES (NULL), (NULL), (NULL);
SELECT * FROM t;
```

```
Query OK, 0 rows affected (0.11 sec)
Query OK, 3 rows affected (0.02 sec)
Records: 3  Duplicates: 0  Warnings: 0
+---+---------------------+
| a | b                   |
+---+---------------------+
| 1 | 2020-09-09 20:38:22 |
| 2 | 2020-09-09 20:38:22 |
| 3 | 2020-09-09 20:38:22 |
+---+---------------------+
3 rows in set (0.00 sec)
```

TiDB 能保证自增值的单调性，但并不能保证其连续性。参考以下示例：

```
CREATE TABLE t (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, a VARCHAR(10), cnt INT NOT NULL
    ↪ DEFAULT 1, UNIQUE KEY (a));
INSERT INTO t (a) VALUES ('A'), ('B');
SELECT * FROM t;
INSERT INTO t (a) VALUES ('A'), ('C') ON DUPLICATE KEY UPDATE cnt = cnt + 1;
SELECT * FROM t;
```

```
Query OK, 0 rows affected (0.00 sec)

Query OK, 2 rows affected (0.00 sec)
Records: 2  Duplicates: 0  Warnings: 0

+----+------+-----+
| id | a    | cnt |
+----+------+-----+
|  1 | A    |   1 |
|  2 | B    |   1 |
+----+------+-----+
2 rows in set (0.00 sec)

Query OK, 3 rows affected (0.00 sec)
Records: 2  Duplicates: 1  Warnings: 0

+----+------+-----+
| id | a    | cnt |
+----+------+-----+
|  1 | A    |   2 |
|  2 | B    |   1 |
|  4 | C    |   1 |
+----+------+-----+
3 rows in set (0.00 sec)
```

在以上示例 `INSERT INTO t (a)VALUES ('A'), ('C')ON DUPLICATE KEY UPDATE cnt = cnt + 1;` 语句中，自增值 3 被分配为 A 键对应的 id 值，但实际上 3 并未作为 id 值插入进表中。这是因为该 INSERT 语句包含一个重复键 A，使得自增序列不连续，出现了间隙。该行为尽管与 MySQL 不同，但仍是合法的。MySQL 在其他情况下也会出现自增序列不连续的情况，例如事务被中止和回滚时。

AUTO_ID_CACHE

如果在另一台服务器上执行插入操作，那么 AUTO_INCREMENT 值的顺序可能会剧烈跳跃，这是由于每台服务器都有各自缓存的 AUTO_INCREMENT 自增值。

```
CREATE TABLE t (a INT PRIMARY KEY AUTO_INCREMENT, b TIMESTAMP NOT NULL DEFAULT NOW());
INSERT INTO t (a) VALUES (NULL), (NULL), (NULL);
INSERT INTO t (a) VALUES (NULL);
SELECT * FROM t;
```

```
Query OK, 1 row affected (0.03 sec)

+---------+---------------------+
| a       | b                   |
+---------+---------------------+
|       1 | 2020-09-09 20:38:22 |
|       2 | 2020-09-09 20:38:22 |
|       3 | 2020-09-09 20:38:22 |
| 2000001 | 2020-09-09 20:43:43 |
+---------+---------------------+
4 rows in set (0.00 sec)
```

以下示例在最先的一台服务器上执行一个插入 INSERT 操作，生成 AUTO_INCREMENT 值 4。因为这台服务器上仍有剩余的 AUTO_INCREMENT 缓存值可用于分配。在该示例中，值的顺序不具有全局单调性：

```
INSERT INTO t (a) VALUES (NULL);
Query OK, 1 row affected (0.01 sec)

SELECT * FROM t ORDER BY b;
+---------+---------------------+
| a       | b                   |
+---------+---------------------+
|       1 | 2020-09-09 20:38:22 |
|       2 | 2020-09-09 20:38:22 |
|       3 | 2020-09-09 20:38:22 |
| 2000001 | 2020-09-09 20:43:43 |
|       4 | 2020-09-09 20:44:43 |
+---------+---------------------+
5 rows in set (0.00 sec)
```

AUTO_INCREMENT 缓存不会持久化，重启会导致缓存值失效。以下示例中，最先的一台服务器重启后，向该服务器执行一条插入操作：

```
INSERT INTO t (a) VALUES (NULL);
Query OK, 1 row affected (0.01 sec)

SELECT * FROM t ORDER BY b;
+---------+---------------------+
| a       | b                   |
+---------+---------------------+
|       1 | 2020-09-09 20:38:22 |
|       2 | 2020-09-09 20:38:22 |
|       3 | 2020-09-09 20:38:22 |
| 2000001 | 2020-09-09 20:43:43 |
|       4 | 2020-09-09 20:44:43 |
| 2030001 | 2020-09-09 20:54:11 |
+---------+---------------------+
6 rows in set (0.00 sec)
```

TiDB 服务器频繁重启可能导致 AUTO_INCREMENT 缓存值被快速消耗。在以上示例中，最先的一台服务器本来有可用的缓存值 [5-3000]。但重启后，这些值便丢失了，无法进行重新分配。

用户不应指望 AUTO_INCREMENT 值保持连续。在以下示例中，一台 TiDB 服务器的缓存值为 [2000001-2030000]。当手动插入值 2029998 时，TiDB 取用了一个新缓存区间的值：

```
INSERT INTO t (a) VALUES (2029998);
Query OK, 1 row affected (0.01 sec)

INSERT INTO t (a) VALUES (NULL);
Query OK, 1 row affected (0.01 sec)

INSERT INTO t (a) VALUES (NULL);
Query OK, 1 row affected (0.00 sec)

INSERT INTO t (a) VALUES (NULL);
Query OK, 1 row affected (0.02 sec)

INSERT INTO t (a) VALUES (NULL);
Query OK, 1 row affected (0.01 sec)

SELECT * FROM t ORDER BY b;
+---------+---------------------+
| a       | b                   |
+---------+---------------------+
|       1 | 2020-09-09 20:38:22 |
|       2 | 2020-09-09 20:38:22 |
|       3 | 2020-09-09 20:38:22 |
| 2000001 | 2020-09-09 20:43:43 |
|       4 | 2020-09-09 20:44:43 |
```

```
| 2030001 | 2020-09-09 20:54:11 |
| 2029998 | 2020-09-09 21:08:11 |
| 2029999 | 2020-09-09 21:08:11 |
| 2030000 | 2020-09-09 21:08:11 |
| 2060001 | 2020-09-09 21:08:11 |
| 2060002 | 2020-09-09 21:08:11 |
+---------+---------------------+
11 rows in set (0.00 sec)
```

以上示例插入 2030000 后，下一个值为 2060001，即顺序出现跳跃。这是因为另一台 TiDB 服务器获取了中间缓存区间 [2030001-2060000]。当部署有多台 TiDB 服务器时，AUTO_INCREMENT 值的顺序会出现跳跃，因为对缓存值的请求是交叉出现的。

缓存大小控制

TiDB 自增 ID 的缓存大小在早期版本中是对用户透明的。从 v3.1.2、v3.0.14 和 v4.0.rc-2 版本开始，TiDB 引入了 AUTO_ID_CACHE 表选项来允许用户自主设置自增 ID 分配缓存的大小。例如：

```
CREATE TABLE t(a int AUTO_INCREMENT key) AUTO_ID_CACHE 100;
Query OK, 0 rows affected (0.02 sec)

INSERT INTO t VALUES();
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

SELECT * FROM t;
+---+
| a |
+---+
| 1 |
+---+
1 row in set (0.01 sec)
```

此时如果将该列的自增缓存无效化，重新进行隐式分配：

```
DELETE FROM t;
Query OK, 1 row affected (0.01 sec)

RENAME TABLE t to t1;
Query OK, 0 rows affected (0.01 sec)

INSERT INTO t1 VALUES()
Query OK, 1 row affected (0.00 sec)

SELECT * FROM t;
+-----+
| a   |
```

```
+-----+
| 101 |
+-----+
1 row in set (0.00 sec)
```

可以看到再一次分配的值为 101，说明该表的自增 ID 分配缓存的大小为 100。

此外如果在批量插入的 INSERT 语句中所需连续 ID 长度超过 AUTO_ID_CACHE 的长度时，TiDB 会适当调大缓存以便能够保证该语句的正常插入。

自增步长和偏移量设置

从 v3.0.9 和 v4.0.rc-1 开始，和 MySQL 的行为类似，自增列隐式分配的值遵循 session 变量 @@auto_increment_increment 和 @@auto_increment_offset 的控制，其中自增列隐式分配的值 (ID) 将满足式子 (ID - auto_increment_offset)% auto_increment_increment == 0。

MySQL 兼容模式

从 v6.4.0 开始，TiDB 实现了中心化分配自增 ID 的服务，可以支持 TiDB 实例不缓存数据，而是每次请求都访问中心化服务获取 ID。

当前中心化分配服务内置在 TiDB 进程，类似于 DDL Owner 的工作模式。有一个 TiDB 实例将充当 “主” 的角色提供 ID 分配服务，而其它的 TiDB 实例将充当 “备” 角色。当 “主” 节点发生故障时，会自动进行 “主备切换”，从而保证中心化服务的高可用。

MySQL 兼容模式的使用方式是，建表时将 AUTO_ID_CACHE 设置为 1：

```
CREATE TABLE t(a int AUTO_INCREMENT key) AUTO_ID_CACHE 1;
```

> 注意：
>
> 在 TiDB 各个版本中，AUTO_ID_CACHE 设置为 1 都表明 TiDB 不再缓存 ID，但是不同版本的实现方式不一样：
>
> - 对于 TiDB v6.4.0 之前的版本，由于每次分配 ID 都需要通过一个 TiKV 事务完成 AUTO_INCREMENT 值的持久化修改，因此设置 AUTO_ID_CACHE 为 1 会出现性能下降。
> - 对于 v6.4.0 及以上版本，由于引入了中心化的分配服务，AUTO_INCREMENT 值的修改只是在 TiDB 服务进程中的一个内存操作，相较于之前版本更快。
> - 将 AUTO_ID_CACHE 设置为 1 表示 TiDB 使用默认的缓存大小 30000。

使用 MySQL 兼容模式后，能保证 ID 唯一、单调递增，行为几乎跟 MySQL 完全一致。即使跨 TiDB 实例访问，ID 也不会出现回退。只有当中心化服务的 “主” TiDB 实例异常崩溃时，才有可能造成少量 ID 不连续。这是因为主备切换时，“备” 节点需要丢弃一部分之前的 “主” 节点可能已经分配的 ID，以保证 ID 不出现重复。

使用限制

目前在 TiDB 中使用 AUTO_INCREMENT 有以下限制：

- 对于 v6.6.0 及更早的 TiDB 版本，定义的列必须为主键或者索引前缀。

- 只能定义在类型为整数、 FLOAT 或 DOUBLE 的列上。
- 不支持与列的默认值 DEFAULT 同时指定在同一列上。
- 不支持使用 ALTER TABLE 来添加 AUTO_INCREMENT 属性。
- 支持使用 ALTER TABLE 来移除 AUTO_INCREMENT 属性。但从 TiDB 2.1.18 和 3.0.4 版本开始，TiDB 通过 session 变量 @@tidb_allow_remove_auto_inc 控制是否允许通过 ALTER TABLE MODIFY 或 ALTER TABLE CHANGE 来移除列的 AUTO_INCREMENT 属性，默认是不允许移除。
- ALTER TABLE 需要 FORCE 选项来将 AUTO_INCREMENT 设置为较小的值。
- 将 AUTO_INCREMENT 设置为小于 MAX(<auto_increment_column>) 的值会导致重复键，因为预先存在的值不会被跳过。

14.11.1.1.2 AUTO_RANDOM 从 v3.1.0 版本开始引入

使用场景

由于 AUTO_RANDOM 的值具有随机性和唯一性，因此 AUTO_RANDOM 通常用于代替AUTO_INCREMENT，以避免 TiDB 分配连续的 ID 值造成单个存储节点的写热点问题。如果当前表的 AUTO_INCREMENT 列是主键列，且列类型为 BIGINT，可以通过 ALTER TABLE t MODIFY COLUMN id BIGINT AUTO_RANDOM(5); 从 AUTO_INCREMENT 切换成 AUTO_RANDOM。

关于如何在高并发写入场景下调优 TiDB，请参阅TiDB 高并发写入场景最佳实践。

基本概念

AUTO_RANDOM 是应用在 BIGINT 类型列的属性，用于列值的自动分配。其自动分配的值满足随机性和唯一性。

以下语句均可创建包含 AUTO_RANDOM 列的表，其中 AUTO_RANDOM 列必须被包含在主键中，并且是主键的第一列。

```
CREATE TABLE t (a BIGINT AUTO_RANDOM, b VARCHAR(255), PRIMARY KEY (a));
CREATE TABLE t (a BIGINT PRIMARY KEY AUTO_RANDOM, b VARCHAR(255));
CREATE TABLE t (a BIGINT AUTO_RANDOM(6), b VARCHAR(255), PRIMARY KEY (a));
CREATE TABLE t (a BIGINT AUTO_RANDOM(5, 54), b VARCHAR(255), PRIMARY KEY (a));
CREATE TABLE t (a BIGINT AUTO_RANDOM(5, 54), b VARCHAR(255), PRIMARY KEY (a, b));
```

AUTO_RANDOM 关键字可以被包裹在 TiDB 可执行注释中，注释语法请参考TiDB 可执行注释。

```
CREATE TABLE t (a bigint /*T![auto_rand] AUTO_RANDOM */, b VARCHAR(255), PRIMARY KEY (a));
CREATE TABLE t (a bigint PRIMARY KEY /*T![auto_rand] AUTO_RANDOM */, b VARCHAR(255));
CREATE TABLE t (a BIGINT /*T![auto_rand] AUTO_RANDOM(6) */, b VARCHAR(255), PRIMARY KEY (a));
CREATE TABLE t (a BIGINT  /*T![auto_rand] AUTO_RANDOM(5, 54) */, b VARCHAR(255), PRIMARY KEY (a))
    ↪ ;
```

在用户执行 INSERT 语句时：

- 如果语句中显式指定了 AUTO_RANDOM 列的值，则该值会被正常插入到表中。
- 如果语句中没有显式指定 AUTO_RANDOM 列的值，TiDB 会自动生成一个随机的值插入到表中。

```
tidb> CREATE TABLE t (a BIGINT PRIMARY KEY AUTO_RANDOM, b VARCHAR(255));
Query OK, 0 rows affected, 1 warning (0.01 sec)
```

```
tidb> INSERT INTO t(a, b) VALUES (1, 'string');
Query OK, 1 row affected (0.00 sec)

tidb> SELECT * FROM t;
+---+--------+
| a | b      |
+---+--------+
| 1 | string |
+---+--------+
1 row in set (0.01 sec)

tidb> INSERT INTO t(b) VALUES ('string2');
Query OK, 1 row affected (0.00 sec)

tidb> INSERT INTO t(b) VALUES ('string3');
Query OK, 1 row affected (0.00 sec)

tidb> SELECT * FROM t;
+---------------------+---------+
| a                   | b       |
+---------------------+---------+
|                   1 | string  |
| 1152921504606846978 | string2 |
| 4899916394579099651 | string3 |
+---------------------+---------+
3 rows in set (0.00 sec)
```

TiDB 自动分配的 AUTO_RANDOM(S, R) 列值共有 64 位：

- S 表示分片位的数量，取值范围是 1 到 15。默认为 5。
- R 表示自动分配值域的总长度，取值范围是 32 到 64。默认为 64。

AUTO_RANDOM 列值的具体结构如下：

总位数	符号位	保留位	分片位	自增位
64 bits	0/1 bit	(64-R) bits	S bits	(R-1-S) bits

- 符号位的长度由该列是否存在 UNSIGNED 属性决定：存在则为 0，否则为 1。
- 保留位的长度为 64-R，保留位的内容始终为 0。
- 分片位的内容通过计算当前事务的开始时间的哈希值而得。要使用不同的分片位数量（例如 10），可以在建表时指定 AUTO_RANDOM(10)。
- 自增位的值保存在存储引擎中，按顺序分配，每次分配完值后会自增 1。自增位保证了 AUTO_RANDOM 列值全局唯一。当自增位耗尽后，再次自动分配时会报 Failed to read auto-increment value from

↪ storage engine 的错误。

> 注意：
>
> 分片位长度 (S) 的选取：
>
> - 由于总位数固定为 64 位，分片位的数量会影响到自增位的数量：当分片位数增加时，自增位数会减少，反之亦然。因此，你需要权衡 “自动分配值的随机性” 以及 “可用空间”。
> - 最佳实践是将分片位设置为 `log(2, x)`，其中 x 为当前集群存储引擎的数量。例如，一个 TiDB 集群中存在 16 个 TiKV，分片位可以设置为 `log(2, 16)`，即 4。在所有 Region 被均匀调度到各个 TiKV 上以后，此时大批量写入的负载可被均匀分布到不同 TiKV 节点，以实现资源最大化利用。
>
> 值域长度 (R) 的选取：
>
> - 通常，在应用程序的数值类型无法表示完整的 64 位整数时需要设置 R 参数。
> - 例如：JSON number 类型的取值范围为 `[-(2^53)+1, (2^53)-1]`，而 TiDB 很容易会为 `AUTO_RANDOM(5)` 的列分配超出该范围的整数，导致应用程序读取该列的值时出现异常。此时，你可以用 `AUTO_RANDOM(5, 54)` 代替 `AUTO_RANDOM(5)`，使得 TiDB 不会分配出大于 9007199254740991 (2^53-1) 的整数。

`AUTO_RANDOM` 列隐式分配的值会影响 `last_insert_id()`。可以使用 `SELECT last_insert_id()` 获取上一次 TiDB 隐式分配的 ID。

要查看某张含有 `AUTO_RANDOM` 属性的表的分片位数量，除了 `SHOW CREATE TABLE` 以外，还可以在系统表 `INFORMATION_SCHEMA.TABLES` 中 `TIDB_ROW_ID_SHARDING_INFO` 一列中查到模式为 `PK_AUTO_RANDOM_BITS=x` 的值，其中 x 为分片位的数量。

创建完一张含有 `AUTO_RANDOM` 属性的表后，可以使用 `SHOW WARNINGS` 查看当前表可支持的最大隐式分配的次数：

```
CREATE TABLE t (a BIGINT AUTO_RANDOM, b VARCHAR(255), PRIMARY KEY (a));
SHOW WARNINGS;
```

输出结果如下：

```
+-------+------+-----------------------------------------------------------+
| Level | Code | Message                                                   |
+-------+------+-----------------------------------------------------------+
| Note  | 1105 | Available implicit allocation times: 288230376151711743 |
+-------+------+-----------------------------------------------------------+
1 row in set (0.00 sec)
```

使用限制

目前在 TiDB 中使用 `AUTO_RANDOM` 有以下限制：

- 要使用显式插入的功能，需要将系统变量 @@allow_auto_random_explicit_insert 的值设置为 1（默认值为 0）。不建议自行显式指定含有 AUTO_RANDOM 列的值。不恰当地显式赋值，可能会导致该表提前耗尽用于自动分配的数值。
- 该属性必须指定在 BIGINT 类型的主键列上，否则会报错。此外，当主键属性为 NONCLUSTERED 时，即使是整型主键列，也不支持使用 AUTO_RANDOM。要了解关于 CLUSTERED 主键的详细信息，请参考聚簇索引。
- 不支持使用 ALTER TABLE 来修改 AUTO_RANDOM 属性，包括添加或移除该属性。
- 支持将 AUTO_INCREMENT 属性改为 AUTO_RANDOM 属性。但在 AUTO_INCREMENT 的列数据最大值已接近 BIGINT 类型最大值的情况下，修改可能会失败。
- 不支持修改含有 AUTO_RANDOM 属性的主键列的列类型。
- 不支持与 AUTO_INCREMENT 同时指定在同一列上。
- 不支持与列的默认值 DEFAULT 同时指定在同一列上。
- AUTO_RANDOM 列的数据很难迁移到 AUTO_INCREMENT 列上，因为 AUTO_RANDOM 列自动分配的值通常都很大。

14.11.1.1.3 SHARD_ROW_ID_BITS

本文介绍表属性 SHARD_ROW_ID_BITS，它用来设置隐式 _tidb_rowid 分片数量的 bit 位数。

基本概念

对于非聚簇索引主键或没有主键的表，TiDB 会使用一个隐式的自增 rowid。大量执行 INSERT 插入语句时会把数据集中写入单个 Region，造成写入热点。

通过设置 SHARD_ROW_ID_BITS，可以把 rowid 打散写入多个不同的 Region，缓解写入热点问题。

- SHARD_ROW_ID_BITS = 4 表示 16 个分片
- SHARD_ROW_ID_BITS = 6 表示 64 个分片
- SHARD_ROW_ID_BITS = 0 表示默认值 1 个分片

关于 SHARD_ROW_ID_BITS 的更多使用信息，可参考使用 SHARD_ROW_ID_BITS 处理热点表。

语句示例

```
CREATE TABLE t (
    id INT PRIMARY KEY NONCLUSTERED
) SHARD_ROW_ID_BITS = 4;
```

```
ALTER TABLE t SHARD_ROW_ID_BITS = 4;
```

14.11.1.2 字面值

TiDB 字面值包括字符字面值、数值字面值、时间日期字面值、十六进制、二进制字面值和 NULL 字面值。以下分别对这些字面值进行一一介绍。

14.11.1.2.1 String Literals

String Literals 是一个 bytes 或者 characters 的序列，两端被单引号 ' 或者双引号 " 包围，例如：

```
'example string'
"example string"
```

如果字符串是连续的，会被合并为一个独立的 string。以下表示是一样的：

```
'a string'
'a' ' ' 'string'
"a" ' ' "string"
```

如果开启了 ANSI_QUOTES SQL MODE，那么只有单引号内的会被认为是 String Literals，对于双引号内的字符串，会被认为是一个 identifier。

字符串分为以下两种：

- 二进制字符串 (binary string)：由字节序列构成，它的 charset 和 collation 都是 `binary`，在互相比较时利用字节作为单位。
- 非二进制字符串：由字符序列构成，有除 `binary` 以外的多种 charset 和 collation，在相互比较时用字符（一个字符可能包含多个字节，取决于 charset 的选择）作为单位。

一个 String Literal 可以拥有一个可选的 `character set introducer` 和 `COLLATE clause`，可以用来指定特定的 charset 和 collation。

```
[_charset_name]'string' [COLLATE collation_name]
```

例如：

```
SELECT _latin1'string';
SELECT _binary'string';
SELECT _utf8'string' COLLATE utf8_bin;
```

你可以使用 `N'literal'` 或者 `n'literal'` 来创建使用 national character set 的字符串，下列语句是一样的：

```
SELECT N'some text';
SELECT n'some text';
SELECT _utf8'some text';
```

要在字符串中表示某些特殊字符，可以利用转义字符进行转义：

转义字符	含义
\0	ASCII NUL (X’00’) 字符
\’	单引号
\“	双引号
\b	退格符号
\n	换行符
\r	回车符
\t	tab 符（制表符）
\z	ASCII 26 (Ctrl + Z)
\\	反斜杠 \
\%	%
_	_

如果要在 ' 包围的字符串中表示 "，或者在 " 包围的字符串中表示 '，可以不使用转义字符。

更多细节见 MySQL 官方文档。

14.11.1.2.2 Numeric Literals

数值字面值包括 integer 跟 Decimal 类型跟浮点数字面值。

integer 可以包括 . 作为小数点分隔，数字前可以有 - 或者 + 来表示正数或者负数。

精确数值字面值可以表示为如下格式：`1`，`.2`，`3.4`，`-5`，`-6.78`，`+9.10`.

科学记数法也是被允许的，表示为如下格式：`1.2E3`，`1.2E-3`，`-1.2E3`，`-1.2E-3`。

更多细节见 MySQL 官方文档。

14.11.1.2.3 Date and Time Literals

Date 跟 Time 字面值有几种格式，例如用字符串表示，或者直接用数字表示。在 TiDB 里面，当 TiDB 期望一个 Date 的时候，它会把 `'2017-08-24'`，`'20170824'`，20170824 当做是 Date。

TiDB 的 Date 值有以下几种格式：

- `'YYYY-MM-DD'` 或者 `'YY-MM-DD'`，这里的 - 分隔符并不是严格的，可以是任意的标点符号。比如 `'2017-08-24'`，`'2017&08&24'`，`'2012@12^31'` 都是一样的。唯一需要特别对待的是 '.' 号，它被当做是小数点，用于分隔整数和小数部分。

  Date 和 Time 部分可以被 'T' 分隔，它的作用跟空格符是一样的，例如 `2017-8-24 10:42:00` 跟 `2017-8-24 T10:42:00` 是一样的。

- `'YYYYMMDDHHMMSS'` 或者 `'YYMMDDHHMMSS'`，例如 `'20170824104520'` 和 `'170824104520'` 被当做是 `'2017-08-24 10:45:20'`，但是如果你提供了一个超过范围的值，例如`'170824304520'`，那这就不是一个有效的 Date 字面值。需要注意 `YYYYMMDD HHMMSS`、`YYYYMMDD HH:MM:DD` 和 `YYYY-MM-DD HHMMSS` 等不正确的格式会插入失败。

- YYYYMMDDHHMMSS 或者 YYMMDDHHMMSS，注意这里没有单引号或者双引号，是一个数字。例如 20170824104520 表示 `'2017-08-24 10:45:20'`。

DATETIME 或者 TIMESTAMP 值可以接一个小数部分，用来表示微秒（精度最多到小数点后 6 位），用小数点 . 分隔。

如果 Date 的 year 部分只有两个数字，这是有歧义的（推荐使用四个数字的格式），TiDB 会尝试用以下的规则来解释：

- year 值如果在 70-99 范围，那么被转换成 1970-1999。
- year 值如果在 00-69 范围，那么被转换成 2000-2069。

对于小于 10 的 month 或者 day 值，`'2017-8-4'` 跟 `'2017-08-04'` 是一样的。对于 Time 也是一样，比如 `'2017-08-24 1:2:3'` 跟 `'2017-08-24 01:02:03'`是一样的。

在需要 Date 或者 Time 的语境下，对于数值，TiDB 会根据数值的长度来选定指定的格式：

- 6 个数字，会被解释为 YYMMDD。
- 12 个数字，会被解释为 YYMMDDHHMMSS。
- 8 个数字，会解释为 YYYYMMDD。
- 14 个数字，会被解释为 YYYYMMDDHHMMSS。

对于 Time 类型，TiDB 用以下格式来表示：

- 'D HH:MM:SS'，或者 'HH:MM:SS'，'HH:MM'，'D HH:MM'，'D HH'，'SS'。这里的 D 表示 days，合法的范围是 0-34。
- 数值 HHMMSS，例如 231010 被解释为'23:10:10'。
- 数值 SS，MMSS，HHMMSS 都是可以被当做 Time。

Time 类型的小数点也是 .，精度最多小数点后 6 位。

更多细节见 MySQL 官方文档。

14.11.1.2.4 Boolean Literals

常量 TRUE 和 FALSE 等于 1 和 0，它是大小写不敏感的。

```
SELECT TRUE, true, tRuE, FALSE, FaLsE, false;
```

```
+------+------+------+-------+-------+-------+
| TRUE | true | tRuE | FALSE | FaLsE | false |
+------+------+------+-------+-------+-------+
|    1 |    1 |    1 |     0 |     0 |     0 |
+------+------+------+-------+-------+-------+
1 row in set (0.00 sec)
```

14.11.1.2.5 Hexadecimal Literals

十六进制字面值是有 X 和 0x 前缀的字符串，后接表示十六进制的数字。注意 0x 是大小写敏感的，不能表示为 0X。

例如：

```
X'ac12'
X'12AC'
x'ac12'
x'12AC'
0xac12
0x12AC
```

以下是不合法的十六进制字面值：

```
X'1z' (z 不是合法的十六进制值)
0X12AC (0X 必须用小写的 0x)
```

对于使用 X'val' 格式的十六进制字面值，val 必须包含偶数个字符，如果 val 的长度是奇数（比如 X’A’、X’11A’），可以在前面补一个 0 来避免语法错误。

```
select X'aff';
```

```
ERROR 1105 (HY000): line 0 column 13 near ""hex literal: invalid hexadecimal format, must even
    ↪ numbers, but 3 (total length 13)
```

```
select X'0aff';
```

```
+---------+
| X'0aff' |
+---------+
| 0x0aff  |
+---------+
1 row in set (0.00 sec)
```

默认情况，十六进制字面值是一个二进制字符串。

如果需要将一个字符串或者数字转换为十六进制字面值，可以使用内建函数 HEX()：

```
SELECT HEX('TiDB');
```

```
+-------------+
| HEX('TiDB') |
+-------------+
| 54694442    |
+-------------+
1 row in set (0.01 sec)
```

```
SELECT X'54694442';
```

```
+-------------+
| X'54694442' |
+-------------+
| TiDB        |
+-------------+
1 row in set (0.00 sec)
```

14.11.1.2.6 Bit-Value Literals

位值字面值用 b 或者 0b 做前缀，后接以 0 和 1 组成的二进制数字。其中 0b 是区分大小写的，0B 则会报错。

合法的 Bit-value：

- b’01’

- B’01’
- 0b01

非法的 Bit-value：

- b'2'（2 不是二进制数值，必须为 0 或 1）
- 0B01（0B 必须是小写 0b）

默认情况，位值字面值是一个二进制字符串。

Bit-value 是作为二进制返回的，所以输出到 MySQL Client 可能会无法显示，如果要转换为可打印的字符，可以使用内建函数 BIN() 或者 HEX()：

```
CREATE TABLE t (b BIT(8));
INSERT INTO t SET b = b'00010011';
INSERT INTO t SET b = b'1110';
INSERT INTO t SET b = b'100101';
```

```
SELECT b+0, BIN(b), HEX(b) FROM t;
```

```
+------+--------+--------+
| b+0  | BIN(b) | HEX(b) |
+------+--------+--------+
|   19 | 10011  | 13     |
|   14 | 1110   | E      |
|   37 | 100101 | 25     |
+------+--------+--------+
3 rows in set (0.00 sec)
```

14.11.1.2.7 NULL Values

NULL 代表数据为空，它是大小写不敏感的，与 \N（大小写敏感）同义。

> 注意：
>
> NULL 跟 0 并不一样，跟空字符串 '' 也不一样。

14.11.1.3 Schema 对象名

本文介绍 TiDB SQL 语句中的模式对象名。

模式对象名用于命名 TiDB 中所有的模式对象，包括 database、table、index、column、alias 等等。在 SQL 语句中，可以通过标识符 (identifier) 来引用这些对象。

标识符可以被反引号包裹，即 SELECT * FROM t 也可以写成 SELECT * FROM `t`。但如果标识符中存在至少一个特殊符号，或者它是一个保留关键字，那就必须使用反引号包裹来引用它所代表的模式对象。

```
SELECT * FROM `table` WHERE `table`.id = 20;
```

如果 SQL MODE 中设置了 ANSI_QUOTES，那么 TiDB 会将被双引号 " 包裹的字符串识别为 identifier。

```
MySQL [test]> CREATE TABLE "test" (a varchar(10));
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to
    ↪ your TiDB version for the right syntax to use line 1 column 19 near ""test" (a varchar
    ↪ (10))"

MySQL [test]> SET SESSION sql_mode='ANSI_QUOTES';
Query OK, 0 rows affected (0.000 sec)

MySQL [test]> CREATE TABLE "test" (a varchar(10));
Query OK, 0 rows affected (0.012 sec)
```

如果要在被引用的标识符中使用反引号这个字符，则需要重复两次反引号，例如创建一个表 a `b：

```
CREATE TABLE `a``b` (a int);
```

在 select 语句中，alias 部分可以用标识符或者字符串：

```
SELECT 1 AS `identifier`, 2 AS 'string';
```

```
+------------+--------+
| identifier | string |
+------------+--------+
|          1 |      2 |
+------------+--------+
1 row in set (0.00 sec)
```

更多细节，请参考 MySQL 文档。

14.11.1.3.1 Identifier Qualifiers

Object Names（对象名字）有时可以被限定或者省略。例如在创建表的时候可以省略数据库限定名：

```
CREATE TABLE t (i int);
```

如果之前没有使用 USE 或者连接参数来设定数据库，会报 ERROR 1046 (3D000): No database selected 错误。此时可以指定数据库限定名：

```
CREATE TABLE test.t (i int);
```

. 的左右两端可以出现空格，table_name.col_name 等于 table_name . col_name。

如果要引用这个模式对象，那么请使用：

```
`table_name`.`col_name`
```

而不是：

```
`table_name.col_name`
```

更多细节，请参考 MySQL 文档。

14.11.1.4 关键字

本文介绍 TiDB 的关键字，对保留字和非保留字作出区分，并汇总所有的关键字以供查询使用。

关键字是 SQL 语句中具有特殊含义的单词，例如 SELECT，UPDATE，DELETE 等等。它们之中有的能够直接作为标识符，被称为非保留关键字（简称非保留字），但有需要经过特殊处理才能作为标识符的字，被称为保留关键字（简称保留字）。但是，也存在一些特殊的非保留关键字，有时候可能也需要进行特殊处理，推荐你将它们当作保留关键字处理。

对于保留字，必须使用反引号包裹，才能作为标识符被使用。例如：

```
CREATE TABLE select (a INT);
```

```
ERROR 1105 (HY000): line 0 column 19 near " (a INT)" (total length 27)
```

```
CREATE TABLE `select` (a INT);
```

```
Query OK, 0 rows affected (0.09 sec)
```

而非保留字则不需要反引号也能直接作为标识符。例如 BEGIN 和 END 是非保留字，以下语句能够正常执行：

```
CREATE TABLE `select` (BEGIN int, END int);
```

```
Query OK, 0 rows affected (0.09 sec)
```

有一种特殊情况，如果使用了限定符 .，那么也不需要用反引号：

```
CREATE TABLE test.select (BEGIN int, END int);
```

```
Query OK, 0 rows affected (0.08 sec)
```

14.11.1.4.1 关键字列表

下表列出了 TiDB 中所有的关键字。其中保留字用 (R) 来标识。窗口函数的保留字用 (R-Window) 来标识。需要用反引号 ` 包裹的特殊非保留字用 (S) 来标识。

A

- ACCOUNT
- ACTION
- ADD (R)
- ADMIN (R)

- ADVISE
- AFTER
- AGAINST
- AGO
- ALGORITHM
- ALL (R)
- ALTER (R)
- ALWAYS
- ANALYZE (R)
- AND (R)
- ANY
- AS (R)
- ASC (R)
- ASCII
- AUTO_ID_CACHE
- AUTO_INCREMENT
- AUTO_RANDOM
- AUTO_RANDOM_BASE
- AVG
- AVG_ROW_LENGTH

B

- BACKEND
- BACKUP
- BACKUPS
- BEGIN
- BETWEEN (R)
- BIGINT (R)
- BINARY (R)
- BINDING
- BINDINGS
- BINLOG
- BIT
- BLOB (R)
- BLOCK
- BOOL
- BOOLEAN
- BOTH (R)
- BTREE
- BUCKETS (R)
- BUILTINS (R)
- BY (R)
- BYTE

C

- CACHE
- CANCEL (R)
- CAPTURE
- CASCADE (R)
- CASCADED
- CASE (R)
- CHAIN
- CHANGE (R)
- CHAR (R)
- CHARACTER (R)
- CHARSET
- CHECK (R)
- CHECKPOINT
- CHECKSUM
- CIPHER
- CLEANUP
- CLIENT
- CMSKETCH (R)
- COALESCE
- COLLATE (R)
- COLLATION
- COLUMN (R)
- COLUMNS
- COLUMN_FORMAT
- COMMENT
- COMMIT
- COMMITTED
- COMPACT
- COMPRESSED
- COMPRESSION
- CONCURRENCY
- CONFIG
- CONNECTION
- CONSISTENT
- CONSTRAINT (R)
- CONTEXT
- CONVERT (R)
- CPU
- CREATE (R)
- CROSS (R)
- CSV_BACKSLASH_ESCAPE
- CSV_DELIMITER
- CSV_HEADER
- CSV_NOT_NULL
- CSV_NULL

- CSV_SEPARATOR
- CSV_TRIM_LAST_SEPARATORS
- CUME_DIST (R-Window)
- CURRENT
- CURRENT_DATE (R)
- CURRENT_ROLE (R)
- CURRENT_TIME (R)
- CURRENT_TIMESTAMP (R)
- CURRENT_USER (R)
- CYCLE

D

- DATA
- DATABASE (R)
- DATABASES (R)
- DATE
- DATETIME
- DAY
- DAY_HOUR (R)
- DAY_MICROSECOND (R)
- DAY_MINUTE (R)
- DAY_SECOND (R)
- DDL (R)
- DEALLOCATE
- DECIMAL (R)
- DEFAULT (R)
- DEFINER
- DELAYED (R)
- DELAY_KEY_WRITE
- DELETE (R)
- DENSE_RANK (R-Window)
- DEPTH (R)
- DESC (R)
- DESCRIBE (R)
- DIRECTORY
- DISABLE
- DISCARD
- DISK
- DISTINCT (R)
- DISTINCTROW (R)
- DIV (R)
- DO
- DOUBLE (R)
- DRAINER (R)

- DROP (R)
- DUAL (R)
- DUPLICATE
- DYNAMIC

E

- ELSE (R)
- ENABLE
- ENCLOSED (R)
- ENCRYPTION
- END
- ENFORCED
- ENGINE
- ENGINES
- ENUM
- ERROR
- ERRORS
- ESCAPE
- ESCAPED (R)
- EVENT
- EVENTS
- EVOLVE
- EXCEPT (R)
- EXCHANGE
- EXCLUSIVE
- EXECUTE
- EXISTS (R)
- EXPANSION
- EXPIRE
- EXPLAIN (R)
- EXTENDED

F

- FALSE (R)
- FAULTS
- FIELDS
- FILE
- FIRST
- FIRST_VALUE (R-Window)
- FIXED
- FLOAT (R)
- FLUSH
- FOLLOWING

- FOR (R)
- FORCE (R)
- FOREIGN (R)
- FORMAT
- FROM (R)
- FULL
- FULLTEXT (R)
- FUNCTION

G

- GENERAL
- GENERATED (R)
- GLOBAL
- GRANT (R)
- GRANTS
- GROUP (R)
- GROUPS (R-Window)

H

- HASH
- HAVING (R)
- HIGH_PRIORITY (R)
- HISTORY
- HOSTS
- HOUR
- HOUR_MICROSECOND (R)
- HOUR_MINUTE (R)
- HOUR_SECOND (R)

I

- IDENTIFIED
- IF (R)
- IGNORE (R)
- IMPORT
- IMPORTS
- IN (R)
- INCREMENT
- INCREMENTAL
- INDEX (R)
- INDEXES
- INFILE (R)
- INNER (R)

- INSERT (R)
- INSERT_METHOD
- INSTANCE
- INT (R)
- INT1 (R)
- INT2 (R)
- INT3 (R)
- INT4 (R)
- INT8 (R)
- INTEGER (R)
- INTERVAL (R)
- INTO (R)
- INVISIBLE
- INVOKER
- IO
- IPC
- IS (R)
- ISOLATION
- ISSUER

J

- JOB (R)
- JOBS (R)
- JOIN (R)
- JSON

K

- KEY (R)
- KEYS (R)
- KEY_BLOCK_SIZE
- KILL (R)

L

- LABELS
- LAG (R-Window)
- LANGUAGE
- LAST
- LASTVAL
- LAST_BACKUP
- LAST_VALUE (R-Window)
- LEAD (R-Window)
- LEADING (R)

- LEFT (R)
- LESS
- LEVEL
- LIKE (R)
- LIMIT (R)
- LINEAR (R)
- LINES (R)
- LIST
- LOAD (R)
- LOCAL
- LOCALTIME (R)
- LOCALTIMESTAMP (R)
- LOCATION
- LOCK (R)
- LOGS
- LONG (R)
- LONGBLOB (R)
- LONGTEXT (R)
- LOW_PRIORITY (R)

M

- MASTER
- MATCH (R)
- MAXVALUE (R)
- MAX_CONNECTIONS_PER_HOUR
- MAX_IDXNUM
- MAX_MINUTES
- MAX_QUERIES_PER_HOUR
- MAX_ROWS
- MAX_UPDATES_PER_HOUR
- MAX_USER_CONNECTIONS
- MB
- MEDIUMBLOB (R)
- MEDIUMINT (R)
- MEDIUMTEXT (R)
- MEMORY
- MERGE
- MICROSECOND
- MINUTE
- MINUTE_MICROSECOND (R)
- MINUTE_SECOND (R)
- MINVALUE
- MIN_ROWS
- MOD (R)

- MODE
- MODIFY
- MONTH

N

- NAMES
- NATIONAL
- NATURAL (R)
- NCHAR
- NEVER
- NEXT
- NEXTVAL
- NO
- NOCACHE
- NOCYCLE
- NODEGROUP
- NODE_ID (R)
- NODE_STATE (R)
- NOMAXVALUE
- NOMINVALUE
- NONE
- NOT (R)
- NOWAIT
- NO_WRITE_TO_BINLOG (R)
- NTH_VALUE (R-Window)
- NTILE (R-Window)
- NULL (R)
- NULLS
- NUMERIC (R)
- NVARCHAR

O

- OFFSET
- ON (R)
- ONLINE
- ONLY
- ON_DUPLICATE
- OPEN
- OPTIMISTIC (R)
- OPTIMIZE (R)
- OPTION (R)
- OPTIONALLY (R)
- OR (R)

- ORDER (R)
- OUTER (R)
- OUTFILE (R)
- OVER (R-Window)

P

- PACK_KEYS
- PAGE
- PARSER
- PARTIAL
- PARTITION (R)
- PARTITIONING
- PARTITIONS
- PASSWORD
- PERCENT_RANK (R-Window)
- PER_DB
- PER_TABLE
- PESSIMISTIC (R)
- PLACEMENT (S)
- PLUGINS
- PRECEDING
- PRECISION (R)
- PREPARE
- PRE_SPLIT_REGIONS
- PRIMARY (R)
- PRIVILEGES
- PROCEDURE (R)
- PROCESS
- PROCESSLIST
- PROFILE
- PROFILES
- PUMP (R)

Q

- QUARTER
- QUERIES
- QUERY
- QUICK

R

- RANGE (R)
- RANK (R-Window)

- RATE_LIMIT
- READ (R)
- REAL (R)
- REBUILD
- RECOVER
- REDUNDANT
- REFERENCES (R)
- REGEXP (R)
- REGION (R)
- REGIONS (R)
- RELEASE (R)
- RELOAD
- REMOVE
- RENAME (R)
- REORGANIZE
- REPAIR
- REPEAT (R)
- REPEATABLE
- REPLACE (R)
- REPLICA
- REPLICATION
- REQUIRE (R)
- RESPECT
- RESTORE
- RESTORES
- RESTRICT (R)
- REVERSE
- REVOKE (R)
- RIGHT (R)
- RLIKE (R)
- ROLE
- ROLLBACK
- ROUTINE
- ROW (R)
- ROWS (R-Window)
- ROW_COUNT
- ROW_FORMAT
- ROW_NUMBER (R-Window)
- RTREE

S

- SAMPLES (R)
- SECOND
- SECONDARY_ENGINE

- SECONDARY_LOAD
- SECONDARY_UNLOAD
- SECOND_MICROSECOND (R)
- SECURITY
- SELECT (R)
- SEND_CREDENTIALS_TO_TIKV
- SEPARATOR
- SEQUENCE
- SERIAL
- SERIALIZABLE
- SESSION
- SET (R)
- SETVAL
- SHARD_ROW_ID_BITS
- SHARE
- SHARED
- SHOW (R)
- SHUTDOWN
- SIGNED
- SIMPLE
- SKIP_SCHEMA_FILES
- SLAVE
- SLOW
- SMALLINT (R)
- SNAPSHOT
- SOME
- SOURCE
- SPATIAL (R)
- SPLIT (R)
- SQL (R)
- SQL_BIG_RESULT (R)
- SQL_BUFFER_RESULT
- SQL_CACHE
- SQL_CALC_FOUND_ROWS (R)
- SQL_NO_CACHE
- SQL_SMALL_RESULT (R)
- SQL_TSI_DAY
- SQL_TSI_HOUR
- SQL_TSI_MINUTE
- SQL_TSI_MONTH
- SQL_TSI_QUARTER
- SQL_TSI_SECOND
- SQL_TSI_WEEK
- SQL_TSI_YEAR
- SSL (R)

- START
- STARTING (R)
- STATS (R)
- STATS_AUTO_RECALC
- STATS_BUCKETS (R)
- STATS_HEALTHY (R)
- STATS_HISTOGRAMS (R)
- STATS_META (R)
- STATS_PERSISTENT
- STATS_SAMPLE_PAGES
- STATUS
- STORAGE
- STORED (R)
- STRAIGHT_JOIN (R)
- STRICT_FORMAT
- SUBJECT
- SUBPARTITION
- SUBPARTITIONS
- SUPER
- SWAPS
- SWITCHES
- SYSTEM_TIME

T

- TABLE (R)
- TABLES
- TABLESPACE
- TABLE_CHECKSUM
- TEMPORARY
- TEMPTABLE
- TERMINATED (R)
- TEXT
- THAN
- THEN (R)
- TIDB (R)
- TIFLASH (R)
- TIKV_IMPORTER
- TIME
- TIMESTAMP
- TINYBLOB (R)
- TINYINT (R)
- TINYTEXT (R)
- TO (R)
- TOKEN_ISSUER

- TOPN (R)
- TRACE
- TRADITIONAL
- TRAILING (R)
- TRANSACTION
- TRIGGER (R)
- TRIGGERS
- TRUE (R)
- TRUNCATE
- TYPE

U

- UNBOUNDED
- UNCOMMITTED
- UNDEFINED
- UNICODE
- UNION (R)
- UNIQUE (R)
- UNKNOWN
- UNLOCK (R)
- UNSIGNED (R)
- UPDATE (R)
- USAGE (R)
- USE (R)
- USER
- USING (R)
- UTC_DATE (R)
- UTC_TIME (R)
- UTC_TIMESTAMP (R)

V

- VALIDATION
- VALUE
- VALUES (R)
- VARBINARY (R)
- VARCHAR (R)
- VARCHARACTER (R)
- VARIABLES
- VARYING (R)
- VIEW
- VIRTUAL (R)
- VISIBLE

W

- WARNINGS
- WEEK
- WEIGHT_STRING
- WHEN (R)
- WHERE (R)
- WIDTH (R)
- WINDOW (R-Window)
- WITH (R)
- WITHOUT
- WRITE (R)

X

- X509
- XOR (R)

Y

- YEAR
- YEAR_MONTH (R)

Z

- ZEROFILL (R)

14.11.1.5 用户自定义变量

> **警告：**
>
> 当前该功能为实验特性，不建议在生产环境中使用。

本文介绍 TiDB 的用户自定义变量的概念，以及设置和读取用户自定义变量的方法。

用户自定义变量格式为 `@var_name`。组成 `var_name` 的字符可以是任何能够组成标识符 (identifier) 的字符，包括数字 `0-9`、字母 `a-zA-Z`、下划线 `_`、美元符号 `$` 以及 UTF-8 字符。此外，还包括英文句号 `.`。用户自定义变量是大小写不敏感的。

用户自定义变量跟 session 绑定，当前设置的用户变量只在当前连接中可见，其他客户端连接无法查看。

14.11.1.5.1 设置用户自定义变量

用 `SET` 语句可以设置用户自定义变量，语法为 `SET @var_name = expr [, @var_name = expr] ...;`。例如：

```
SET @favorite_db = 'TiDB';
```

```
SET @a = 'a', @b = 'b', @c = 'c';
```

其中赋值符号还可以使用 `:=`。例如：

```
SET @favorite_db := 'TiDB';
```

赋值符号右边的内容可以是任意合法的表达式。例如：

```
SET @c = @a + @b;
```

```
SET @c = b'1000001' + b'1000001';
```

14.11.1.5.2 读取用户自定义变量

要读取一个用户自定义变量，可以使用 SELECT 语句查询：

```
SELECT @a1, @a2, @a3
```

```
+------+------+------+
| @a1  | @a2  | @a3  |
+------+------+------+
|    1 |    2 |    4 |
+------+------+------+
```

还可以在 SELECT 语句中赋值：

```
SELECT @a1, @a2, @a3, @a4 := @a1+@a2+@a3;
```

```
+------+------+------+--------------------+
| @a1  | @a2  | @a3  | @a4 := @a1+@a2+@a3 |
+------+------+------+--------------------+
|    1 |    2 |    4 |                  7 |
+------+------+------+--------------------+
```

其中变量 `@a4` 在被修改或关闭连接之前，值始终为 7。

如果设置用户变量时用了十六进制字面量或者二进制字面量，TiDB 会把它当成二进制字符串。如果要将其设置成数字，那么可以手动加上 CAST 转换，或者在表达式中使用数字的运算符：

```
SET @v1 = b'1000001';
SET @v2 = b'1000001'+0;
SET @v3 = CAST(b'1000001' AS UNSIGNED);
```

```
SELECT @v1, @v2, @v3;
```

```
+------+------+------+
| @v1  | @v2  | @v3  |
+------+------+------+
| A    | 65   | 65   |
+------+------+------+
```

如果获取一个没有设置过的变量，会返回一个 NULL：

```
SELECT @not_exist;
```

```
+------------+
| @not_exist |
+------------+
| NULL       |
+------------+
```

除了 SELECT 读取用户自定义变量以外，常见的用法还有 PREPARE 语句，例如：

```
SET @s = 'SELECT SQRT(POW(?,2) + POW(?,2)) AS hypotenuse';
PREPARE stmt FROM @s;
SET @a = 6;
SET @b = 8;
EXECUTE stmt USING @a, @b;
```

```
+------------+
| hypotenuse |
+------------+
|         10 |
+------------+
```

用户自定义变量的内容不会在 SQL 语句中被当成标识符，例如：

```
SELECT * FROM t;
```

```
+---+
| a |
+---+
| 1 |
+---+
```

```
SET @col = "`a`";
SELECT @col FROM t;
```

```
+------+
| @col |
```

```
+------+
| `a`  |
+------+
```

14.11.1.5.3 MySQL 兼容性

除 SELECT ... INTO <variable> 外，MySQL 和 TiDB 支持的语法相同。

更多细节，请参考 MySQL 文档。

14.11.1.6 表达式语法 (Expression Syntax)

表达式是一个或多个值、操作符或函数的组合。在 TiDB 中，表达式主要使用在 SELECT 语句的各个子句中，包括 Group by 子句、Where 子句、Having 子句、Join 条件以及窗口函数等。此外，部分 DDL 语句也会使用到表达式，例如建表时默认值的设置、生成列的设置，分区规则等。

表达式包含几种类型：

- 标识符，可参考模式对象名。
- 谓词、数值、字符串、日期表达式等，这些类型的字面值也是表达式。
- 函数调用，窗口函数等。可参考函数和操作符概述和窗口函数。
- 其他，包括 paramMarker（即 ? ）、系统变量和用户变量、CASE 表达式等。

以下规则是表达式的语法，该语法基于 TiDB parser 的 parser.y 文件中所定义的规则。此外，下列语法图的可导航版本请参考 TiDB SQL 语法图。

```
Expression ::=
    ( singleAtIdentifier assignmentEq | 'NOT' | Expression ( logOr | 'XOR' | logAnd ) )
        ↪ Expression
|   'MATCH' '(' ColumnNameList ')' 'AGAINST' '(' BitExpr FulltextSearchModifierOpt ')'
|   PredicateExpr ( IsOrNotOp 'NULL' | CompareOp ( ( singleAtIdentifier assignmentEq )?
    ↪ PredicateExpr | AnyOrAll SubSelect ) )* ( IsOrNotOp ( trueKwd | falseKwd | 'UNKNOWN' ) )?

PredicateExpr ::=
    BitExpr ( BetweenOrNotOp BitExpr 'AND' BitExpr )* ( InOrNotOp ( '(' ExpressionList ')' |
        ↪ SubSelect ) | LikeOrNotOp SimpleExpr LikeEscapeOpt | RegexpOrNotOp SimpleExpr )?

BitExpr ::=
    BitExpr ( ( '|' | '&' | '<<' | '>>' | '*' | '/' | '%' | 'DIV' | 'MOD' | '^' ) BitExpr | ( '+'
        ↪  | '-' ) ( BitExpr | "INTERVAL" Expression TimeUnit ) )
|   SimpleExpr

SimpleExpr ::=
    SimpleIdent ( ( '->' | '->>' ) stringLit )?
|   FunctionCallKeyword
|   FunctionCallNonKeyword
|   FunctionCallGeneric
```

```
|   SimpleExpr ( 'COLLATE' CollationName | pipes SimpleExpr )
|   WindowFuncCall
|   Literal
|   paramMarker
|   Variable
|   SumExpr
|   ( '!' | '~' | '-' | '+' | 'NOT' | 'BINARY' ) SimpleExpr
|   'EXISTS'? SubSelect
|   ( ( '(' ( ExpressionList ',' )? | 'ROW' '(' ExpressionList ',' ) Expression | builtinCast '('
    ↪  Expression 'AS' CastType | ( 'DEFAULT' | 'VALUES' ) '(' SimpleIdent | 'CONVERT' '('
    ↪ Expression ( ',' CastType | 'USING' CharsetName ) ) ')'
|   'CASE' ExpressionOpt WhenClause+ ElseOpt 'END'
```

14.11.1.7 注释语法

本文档介绍 TiDB 支持的注释语法。

TiDB 支持三种注释风格：

- 用 # 注释一行：

```
SELECT 1+1;     # 注释文字
```

```
+------+
| 1+1  |
+------+
|    2 |
+------+
1 row in set (0.00 sec)
```

- 用 -- 注释一行：

```
SELECT 1+1;     -- 注释文字
```

```
+------+
| 1+1  |
+------+
|    2 |
+------+
1 row in set (0.00 sec)
```

用 -- 注释时，必须要在其之后留出至少一个空格，否则注释不生效：

```
SELECT 1+1--1;
```

```
+--------+
| 1+1--1 |
+--------+
|      3 |
+--------+
1 row in set (0.01 sec)
```

- 用 /* */ 注释一块，可以注释多行：

```
SELECT 1 /* 这是行内注释文字 */ + 1;
```

```
+--------+
| 1  + 1 |
+--------+
|      2 |
+--------+
1 row in set (0.01 sec)
```

```
SELECT 1+
/*
/*> 这是一条
/*> 多行注释
/*> */
    1;
```

```
+--------------------+
| 1+


        1 |
+--------------------+
|                  2 |
+--------------------+
1 row in set (0.001 sec)
```

14.11.1.7.1 MySQL 兼容的注释语法

TiDB 也跟 MySQL 保持一致，支持一种 C 风格注释的变体：

```
/*! Specific code */
```

或者

```
/*!50110 Specific code */
```

和 MySQL 一样，TiDB 会执行注释中的语句。

例如：`SELECT /*! STRAIGHT_JOIN */ col1 FROM table1,table2 WHERE ...`

在 TiDB 中，这种写法等价于 `SELECT STRAIGHT_JOIN col1 FROM table1,table2 WHERE ...`

如果注释中指定了 Server 版本号，例如 `/*!50110 KEY_BLOCK_SIZE=1024 */`，在 MySQL 中表示只有 MySQL 的版本大于等于 5.1.10 才会处理这个 comment 中的内容。但是在 TiDB 中，这个 MySQL 版本号不会起作用，所有的 comment 都被会处理。

14.11.1.7.2 TiDB 可执行的注释语法

TiDB 也有独立的注释语法，称为 TiDB 可执行注释语法。主要分为两种：

- `/*T! Specific code */`：该语法只能被 TiDB 解析执行，而在其他数据库中会被忽略。
- `/*T![feature_id] Specific code */`：该语法用于保证 TiDB 不同版本之间的兼容性。只有在当前版本中实现了 `feature_id` 对应的功能特性的 TiDB，才会试图解析该注释里的 SQL 片段。例如 v3.1.1 中引入了 AUTO_RANDOM 特性，该版本能够将 `/*T![auto_rand] auto_random */` 解析为 `auto_random`；而 v3.0.0 中没有实现 AUTO_RANDOM 特性，则上述 SQL 语句片段会被忽略。注意前几个字符 `/*T![` 中，各字符之间没有任何空格。

14.11.1.7.3 优化器注释语法

还有一种注释会被当做是优化器 Hint 特殊对待：

```
SELECT /*+ hint */ FROM ...;
```

TiDB 支持的相关优化器 hint 详见Optimizer Hints。

> **注意：**
>
> 在 MySQL 客户端中，TiDB 可执行注释语法会被默认当成注释被清除掉。在 MySQL 客户端 5.7.7 之前的版本中，Hint 也会被默认当成注释被清除掉。推荐在启动客户端时加上 `--comments` 选项，例如 `mysql -h 127.0.0.1 -P 4000 -uroot --comments`。

更多细节，请参考 MySQL 文档。

14.11.2 SQL 语句

14.11.2.1 ADD COLUMN

`ALTER TABLE.. ADD COLUMN` 语句用于在已有表中添加列。在 TiDB 中，`ADD COLUMN` 为在线操作，不会阻塞表中的数据读写。

14.11.2.1.1 语法图

```
AlterTableStmt
         ::= 'ALTER' 'IGNORE'? 'TABLE' TableName AddColumnSpec ( ',' AddColumnSpec )*

TableName ::=
    Identifier ('.' Identifier)?

AddColumnSpec
         ::= 'ADD' 'COLUMN' 'IF NOT EXISTS'? ColumnName ColumnType ColumnOption+ ( 'FIRST' | '
             ↪ AFTER' ColumnName )?

ColumnType
         ::= NumericType
           | StringType
           | DateAndTimeType
           | 'SERIAL'

ColumnOption
         ::= 'NOT'? 'NULL'
           | 'AUTO_INCREMENT'
           | 'PRIMARY'? 'KEY' ( 'CLUSTERED' | 'NONCLUSTERED' )?
           | 'UNIQUE' 'KEY'?
           | 'DEFAULT' ( NowSymOptionFraction | SignedLiteral | NextValueForSequence )
           | 'SERIAL' 'DEFAULT' 'VALUE'
           | 'ON' 'UPDATE' NowSymOptionFraction
           | 'COMMENT' stringLit
           | ( 'CONSTRAINT' Identifier? )? 'CHECK' '(' Expression ')' ( 'NOT'? ( 'ENFORCED' | '
               ↪ NULL' ) )?
           | 'GENERATED' 'ALWAYS' 'AS' '(' Expression ')' ( 'VIRTUAL' | 'STORED' )?
           | 'REFERENCES' TableName ( '(' IndexPartSpecificationList ')' )? Match?
               ↪ OnDeleteUpdateOpt
           | 'COLLATE' CollationName
           | 'COLUMN_FORMAT' ColumnFormat
           | 'STORAGE' StorageMedia
           | 'AUTO_RANDOM' ( '(' LengthNum ')' )?
```

14.11.2.1.2 示例

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT);
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
INSERT INTO t1 VALUES (NULL);
```

```
Query OK, 1 row affected (0.02 sec)
```

```
SELECT * FROM t1;
```

```
+----+
| id |
+----+
|  1 |
+----+
1 row in set (0.00 sec)
```

```
ALTER TABLE t1 ADD COLUMN c1 INT NOT NULL;
```

```
Query OK, 0 rows affected (0.28 sec)
```

```
SELECT * FROM t1;
```

```
+----+----+
| id | c1 |
+----+----+
|  1 |  0 |
+----+----+
1 row in set (0.00 sec)
```

```
ALTER TABLE t1 ADD c2 INT NOT NULL AFTER c1;
```

```
Query OK, 0 rows affected (0.28 sec)
```

```
SELECT * FROM t1;
```

```
+----+----+----+
| id | c1 | c2 |
+----+----+----+
|  1 |  0 |  0 |
+----+----+----+
1 row in set (0.00 sec)
```

14.11.2.1.3 MySQL 兼容性

- 不支持将新添加的列设为 PRIMARY KEY。
- 不支持将新添加的列设为 AUTO_INCREMENT。
- 对添加生成列有局限性，具体可参考：生成列局限性。

14.11.2.1.4 另请参阅

- ADD INDEX
- CREATE TABLE

14.11.2.2 ADD INDEX

`ALTER TABLE.. ADD INDEX` 语句用于在已有表中添加一个索引。在 TiDB 中，`ADD INDEX` 为在线操作，不会阻塞表中的数据读写。

> **警告：**
>
> - 在升级 TiDB 集群的过程中，请勿执行 DDL 语句，否则可能会出现行为未定义的问题。
> - 集群中有 DDL 语句正在被执行时（通常为 `ADD INDEX` 和列类型变更等耗时较久的 DDL 语句），请勿进行升级操作。在升级前，建议使用`ADMIN SHOW DDL` 命令查看集群中是否有正在进行的 DDL Job。如需升级，请等待 DDL 执行完成或使用`ADMIN CANCEL DDL` 命令取消该 DDL Job 后再进行升级。

14.11.2.2.1 语法图

```
AlterTableStmt
         ::= 'ALTER' 'IGNORE'? 'TABLE' TableName AddIndexSpec ( ',' AddIndexSpec )*

AddIndexSpec
         ::= 'ADD' ( ( 'PRIMARY' 'KEY' | ( 'KEY' | 'INDEX' ) 'IF NOT EXISTS'? | 'UNIQUE' ( 'KEY'
             ↪ | 'INDEX' )? ) ( ( Identifier? 'USING' | Identifier 'TYPE' ) IndexType )? | '
             ↪ FULLTEXT' ( 'KEY' | 'INDEX' )? IndexName ) '(' IndexPartSpecification ( ','
             ↪ IndexPartSpecification )* ')' IndexOption*

IndexPartSpecification
         ::= ( ColumnName ( '(' LengthNum ')' )? | '(' Expression ')' ) ( 'ASC' | 'DESC' )

IndexOption
         ::= 'KEY_BLOCK_SIZE' '='? LengthNum
           | IndexType
           | 'WITH' 'PARSER' Identifier
           | 'COMMENT' stringLit
           | 'VISIBLE'
           | 'INVISIBLE'

IndexType
         ::= 'BTREE'
           | 'HASH'
```

```
          | 'RTREE'
```

14.11.2.2.2 示例

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, c1 INT NOT NULL);
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
INSERT INTO t1 (c1) VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 5 rows affected (0.03 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
EXPLAIN SELECT * FROM t1 WHERE c1 = 3;
```

```
+-------------------------+----------+-----------+---------------+--------------------------------+
    ↪
| id                      | estRows  | task      | access object | operator info
    ↪                   |
+-------------------------+----------+-----------+---------------+--------------------------------+
    ↪
| TableReader_7           | 10.00    | root      |               | data:Selection_6
    ↪               |
| └─Selection_6           | 10.00    | cop[tikv] |               | eq(test.t1.c1, 3)
    ↪              |
|   └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t1      | keep order:false, stats:
    ↪ pseudo |
+-------------------------+----------+-----------+---------------+--------------------------------+
    ↪
3 rows in set (0.00 sec)
```

```
ALTER TABLE t1 ADD INDEX (c1);
```

```
Query OK, 0 rows affected (0.30 sec)
```

```
EXPLAIN SELECT * FROM t1 WHERE c1 = 3;
```

```
+------------------------+---------+-----------+-----------------------+-----------------------------------
    ↪
| id                     | estRows | task      | access object         | operator info
    ↪                              |
+------------------------+---------+-----------+-----------------------+-----------------------------------
    ↪
```

```
| IndexReader_6           | 0.01    | root      |                       | index:IndexRangeScan_5
    ↪                       |
| └─IndexRangeScan_5      | 0.01    | cop[tikv] | table:t1, index:c1(c1) | range:[3,3], keep
    ↪ order:false, stats:pseudo |
+-------------------------+---------+-----------+------------------------+-----------------------------------------------
    ↪
2 rows in set (0.00 sec)
```

14.11.2.2.3 MySQL 兼容性

- 不支持 FULLTEXT，HASH 和 SPATIAL 索引。
- 不支持降序索引（类似于 MySQL 5.7）。
- 无法向表中添加 CLUSTERED 类型的 PRIMARY KEY。要了解关于 CLUSTERED 主键的详细信息，请参考聚簇索引。

14.11.2.2.4 另请参阅

- 索引的选择
- 错误索引的解决方案
- CREATE INDEX
- DROP INDEX
- RENAME INDEX
- ALTER INDEX
- ADD COLUMN
- CREATE TABLE
- EXPLAIN

14.11.2.3 ADMIN

ADMIN 语句是 TiDB 扩展语法，用于查看 TiDB 自身的状态，并对 TiDB 中的表数据进行校验。本文介绍了下列与 ADMIN 相关的扩展语句：

- ADMIN RELOAD
- ADMIN PLUGIN
- ADMIN ... BINDINGS
- ADMIN REPAIR TABLE
- ADMIN SHOW SLOW

14.11.2.3.1 ADMIN 与 DDL 相关的扩展语句

语句	功能描述
ADMIN CANCEL DDL JOBS	取消当前正在运行的 DDL 作业
ADMIN CHECKSUM TABLE	计算表中所有行和索引的 CRC64 校验和

语句	功能描述
[ADMIN CHECK [TABLE|INDEX]](#admin-check-[table	index])
[ADMIN SHOW DDL [JOBS|QUERIES]](#admin-show-ddl-[jobs	job-queries])
ADMIN SHOW TELEMETRY	显示通过遥测功能收集到并分享给 PingCAP 的使用信息。

14.11.2.3.2 ADMIN RELOAD 语句

```
ADMIN RELOAD expr_pushdown_blacklist;
```

以上语句用于重新加载表达式下推的黑名单。

```
ADMIN RELOAD opt_rule_blacklist;
```

以上语句用于重新加载逻辑优化规则的黑名单。

14.11.2.3.3 ADMIN PLUGIN 语句

```
ADMIN PLUGINS ENABLE plugin_name [, plugin_name] ...;
```

以上语句用于启用 plugin_name 插件。

```
ADMIN PLUGINS DISABLE plugin_name [, plugin_name] ...;
```

以上语句用于禁用 plugin_name 插件。

14.11.2.3.4 ADMIN ... BINDINGS 语句

```
ADMIN FLUSH bindings;
```

以上语句用于持久化 SQL Plan 绑定的信息。

```
ADMIN CAPTURE bindings;
```

以上语句可以将出现超过一次的 selectexecution-plan 语句生成 SQL Plan 的绑定。

```
ADMIN EVOLVE bindings;
```

开启自动绑定功能后，每隔 bind-info-lease（默认值为 3s）触发一次 SQL Plan 绑定信息的演进。以上语句用于主动触发此演进，SQL Plan 绑定详情可参考：执行计划管理。

```
ADMIN RELOAD bindings;
```

以上语句用于重新加载 SQL Plan 绑定的信息。

14.11.2.3.5 ADMIN REPAIR TABLE 语句

```
ADMIN REPAIR TABLE tbl_name CREATE TABLE STATEMENT;
```

ADMIN REPAIR TABLE tbl_name CREATE TABLE STATEMENT 用于在极端情况下，对存储层中的表的元信息进行非可信的覆盖。“非可信”是指需要人为保证原表的元信息可以完全由 CREATE TABLE STATEMENT 提供。该语句需要打开配置文件项中的repair-mode 开关，并且需要确保所修复的表名在repair-table-list 名单中。

14.11.2.3.6 ADMIN SHOW SLOW 语句

```
ADMIN SHOW SLOW RECENT N;
```

```
ADMIN SHOW SLOW TOP [INTERNAL | ALL] N;
```

这两种语句的具体操作详情可参考：admin show slow 语句。

14.11.2.3.7 语句概览

```
AdminStmt ::=
    'ADMIN' ( 'SHOW' ( 'DDL' ( 'JOBS' Int64Num? WhereClauseOptional | 'JOB' 'QUERIES' NumList )?
        ↪ | TableName 'NEXT_ROW_ID' | 'SLOW' AdminShowSlow ) | 'CHECK' ( 'TABLE' TableNameList
        ↪ | 'INDEX' TableName Identifier ( HandleRange ( ',' HandleRange )* )? ) | 'RECOVER' '
        ↪ INDEX' TableName Identifier | 'CLEANUP' ( 'INDEX' TableName Identifier | 'TABLE' '
        ↪ LOCK' TableNameList ) | 'CHECKSUM' 'TABLE' TableNameList | 'CANCEL' 'DDL' 'JOBS'
        ↪ NumList | 'RELOAD' ( 'EXPR_PUSHDOWN_BLACKLIST' | 'OPT_RULE_BLACKLIST' | 'BINDINGS' )
        ↪ | 'PLUGINS' ( 'ENABLE' | 'DISABLE' ) PluginNameList | 'REPAIR' 'TABLE' TableName
        ↪ CreateTableStmt | ( 'FLUSH' | 'CAPTURE' | 'EVOLVE' ) 'BINDINGS' )
```

14.11.2.3.8 使用示例

执行以下命令，可查看正在执行的 DDL 任务中最近 10 条已经完成的 DDL 任务。未指定 NUM 时，默认只显示最近 10 条已经执行完的 DDL 任务。

```
ADMIN SHOW DDL jobs;
```

```
+--------+---------+------------+---------------+----------------------+-----------+----------+-----------+------
    ↪
| JOB_ID | DB_NAME | TABLE_NAME | JOB_TYPE      | SCHEMA_STATE         | SCHEMA_ID | TABLE_ID |
    ↪ ROW_COUNT | START_TIME                        | END_TIME                          | STATE
    ↪          |
+--------+---------+------------+---------------+----------------------+-----------+----------+-----------+------
    ↪
| 45     | test    | t1         | add index     | write reorganization | 32        | 37       | 0
    ↪          | 2019-01-10 12:38:36.501 +0800 CST |                                   |
    ↪ running       |
```

```
| 44     | test    | t1          | add index      | none                 | 32        | 37       | 0
    ↪           | 2019-01-10 12:36:55.18 +0800 CST  | 2019-01-10 12:36:55.852 +0800 CST |
    ↪ rollback done |
| 43     | test    | t1          | add index      | public               | 32        | 37       | 6
    ↪           | 2019-01-10 12:35:13.66 +0800 CST  | 2019-01-10 12:35:14.925 +0800 CST | synced
    ↪          |
| 42     | test    | t1          | drop index     | none                 | 32        | 37       | 0
    ↪           | 2019-01-10 12:34:35.204 +0800 CST | 2019-01-10 12:34:36.958 +0800 CST | synced
    ↪          |
| 41     | test    | t1          | add index      | public               | 32        | 37       | 0
    ↪           | 2019-01-10 12:33:22.62 +0800 CST  | 2019-01-10 12:33:24.625 +0800 CST | synced
    ↪          |
| 40     | test    | t1          | drop column    | none                 | 32        | 37       | 0
    ↪           | 2019-01-10 12:33:08.212 +0800 CST | 2019-01-10 12:33:09.78 +0800 CST  | synced
    ↪          |
| 39     | test    | t1          | add column     | public               | 32        | 37       | 0
    ↪           | 2019-01-10 12:32:55.42 +0800 CST  | 2019-01-10 12:32:56.24 +0800 CST  | synced
    ↪          |
| 38     | test    | t1          | create table   | public               | 32        | 37       | 0
    ↪           | 2019-01-10 12:32:41.956 +0800 CST | 2019-01-10 12:32:43.956 +0800 CST | synced
    ↪          |
| 36     | test    |             | drop table     | none                 | 32        | 34       | 0
    ↪           | 2019-01-10 11:29:59.982 +0800 CST | 2019-01-10 11:30:00.45 +0800  CST | synced
    ↪          |
| 35     | test    |             | create table   | public               | 32        | 34       | 0
    ↪           | 2019-01-10 11:29:40.741 +0800 CST | 2019-01-10 11:29:41.682 +0800 CST | synced
    ↪          |
| 33     | test    |             | create schema  | public               | 32        | 0        | 0
    ↪           | 2019-01-10 11:29:22.813 +0800 CST | 2019-01-10 11:29:23.954 +0800 CST | synced
    ↪          |
+--------+---------+------------+--------------------+----------------+-----------+----------+-----------+------
    ↪
```

执行以下命令，可查看正在执行的 DDL 任务中最近 5 条已经执行完的 DDL 任务：

```
ADMIN SHOW DDL JOBS 5;
```

```
+--------+---------+------------+---------------+----------------------+-----------+----------+-----------+------
    ↪
| JOB_ID | DB_NAME | TABLE_NAME | JOB_TYPE      | SCHEMA_STATE         | SCHEMA_ID | TABLE_ID |
    ↪ ROW_COUNT | START_TIME                        | END_TIME                           | STATE
    ↪           |
+--------+---------+------------+---------------+----------------------+-----------+----------+-----------+------
    ↪
| 45     | test    | t1          | add index     | write reorganization | 32        | 37       | 0
    ↪           | 2019-01-10 12:38:36.501 +0800 CST |                                    |
```

```
    ↪ running       |
| 44     | test    | t1         | add index     | none                 | 32        | 37       | 0
    ↪          | 2019-01-10 12:36:55.18 +0800 CST  | 2019-01-10 12:36:55.852 +0800 CST |
    ↪ rollback done |
| 43     | test    | t1         | add index     | public               | 32        | 37       | 6
    ↪          | 2019-01-10 12:35:13.66 +0800 CST  | 2019-01-10 12:35:14.925 +0800 CST | synced
    ↪         |
| 42     | test    | t1         | drop index    | none                 | 32        | 37       | 0
    ↪          | 2019-01-10 12:34:35.204 +0800 CST | 2019-01-10 12:34:36.958 +0800 CST | synced
    ↪         |
| 41     | test    | t1         | add index     | public               | 32        | 37       | 0
    ↪          | 2019-01-10 12:33:22.62 +0800 CST  | 2019-01-10 12:33:24.625 +0800 CST | synced
    ↪         |
| 40     | test    | t1         | drop column   | none                 | 32        | 37       | 0
    ↪          | 2019-01-10 12:33:08.212 +0800 CST | 2019-01-10 12:33:09.78 +0800 CST  | synced
    ↪         |
+--------+---------+------------+---------------+----------------------+-----------+----------+-----------+------
    ↪
```

执行以下命令，可查看 test 数据库中未执行完成的 DDL 任务，包括正在执行中以及最近 5 条已经执行完但是执行失败的 DDL 任务。

```
ADMIN SHOW DDL JOBS 5 WHERE state != 'synced' AND db_name = 'test';
```

```
+--------+---------+------------+---------------+----------------------+-----------+----------+-----------+------
    ↪
| JOB_ID | DB_NAME | TABLE_NAME | JOB_TYPE      | SCHEMA_STATE         | SCHEMA_ID | TABLE_ID |
    ↪ ROW_COUNT | START_TIME                        | END_TIME                          | STATE
    ↪          |
+--------+---------+------------+---------------+----------------------+-----------+----------+-----------+------
    ↪
| 45     | test    | t1         | add index     | write reorganization | 32        | 37       | 0
    ↪          | 2019-01-10 12:38:36.501 +0800 CST |                                   |
    ↪ running       |
| 44     | test    | t1         | add index     | none                 | 32        | 37       | 0
    ↪          | 2019-01-10 12:36:55.18 +0800 CST  | 2019-01-10 12:36:55.852 +0800 CST |
    ↪ rollback done |
+--------+---------+------------+---------------+----------------------+-----------+----------+-----------+------
    ↪
```

- JOB_ID：每个 DDL 操作对应一个 DDL 作业，JOB_ID 全局唯一。
- DB_NAME：执行 DDL 操作的数据库的名称。
- TABLE_NAME：执行 DDL 操作的表的名称。
- JOB_TYPE：DDL 操作的类型。

- SCHEMA_STATE：schema 的当前状态。如果 JOB_TYPE 是 add index，则为 index 的状态；如果是 add column，则为 column 的状态，如果是 create table，则为 table 的状态。常见的状态有以下几种：
  - none：表示不存在。一般 drop 操作或者 create 操作失败回滚后，会变为 none 状态。
  - delete only、write only、delete reorganization、write reorganization：这四种状态是中间状态。由于中间状态转换很快，一般操作中看不到这几种状态，只有执行 add index 操作时能看到处于 write reorganization 状态，表示正在添加索引数据。
  - public：表示存在且可用。一般 create table 和 add index/column 等操作完成后，会变为 public 状态，表示新建的 table/column/index 可以正常读写了。
- SCHEMA_ID：执行 DDL 操作的数据库的 ID。
- TABLE_ID：执行 DDL 操作的表的 ID。
- ROW_COUNT：执行 add index 操作时，当前已经添加完成的数据行数。
- START_TIME：DDL 操作的开始时间。
- END_TIME：DDL 操作的结束时间。
- STATE：DDL 操作的状态。常见的状态有以下几种：
  - none：表示该操作任务已经进入 DDL 作业队列中，但尚未执行，因为还在排队等待前面的 DDL 作业完成。另一种原因可能是执行 drop 操作后，会变为 none 状态，但是很快会更新为 synced 状态，表示所有 TiDB 实例都已经同步到该状态。
  - running：表示该操作正在执行。
  - synced：表示该操作已经执行成功，且所有 TiDB 实例都已经同步该状态。
  - rollback done：表示该操作执行失败，回滚完成。
  - rollingback：表示该操作执行失败，正在回滚。
  - cancelling：表示正在取消该操作。这个状态只有在用 ADMIN CANCEL DDL JOBS 命令取消 DDL 作业时才会出现。

14.11.2.3.9 MySQL 兼容性

ADMIN 语句是 TiDB 对于 MySQL 语法的扩展。

14.11.2.4 ADMIN CANCEL DDL

ADMIN CANCEL DDL 语句用于取消当前正在运行的 DDL 作业。可以通过ADMIN SHOW DDL JOBS 语句获取 DDL 作业的 job_id。

用于取消已经提交但未执行完成的 DDL 任务。取消完成后，执行 DDL 任务的 SQL 语句会返回 ERROR 8214 (HY000) : Cancelled DDL job 的错误。取消一个已经执行完成的 DDL 任务会在 RESULT 列看到 DDL Job:90 not found 的错误，表示该任务已从 DDL 等待队列中被移除。

14.11.2.4.1 语法图

```
AdminStmt ::=
    'ADMIN' ( 'SHOW' ( 'DDL' ( 'JOBS' Int64Num? WhereClauseOptional | 'JOB' 'QUERIES' NumList )?
        ↪ | TableName 'NEXT_ROW_ID' | 'SLOW' AdminShowSlow ) | 'CHECK' ( 'TABLE' TableNameList
        ↪ | 'INDEX' TableName Identifier ( HandleRange ( ',' HandleRange )* )? ) | 'RECOVER' '
        ↪ INDEX' TableName Identifier | 'CLEANUP' ( 'INDEX' TableName Identifier | 'TABLE' '
        ↪ LOCK' TableNameList ) | 'CHECKSUM' 'TABLE' TableNameList | 'CANCEL' 'DDL' 'JOBS'
```

```
        ↪ NumList | 'RELOAD' ( 'EXPR_PUSHDOWN_BLACKLIST' | 'OPT_RULE_BLACKLIST' | 'BINDINGS' )
        ↪ | 'PLUGINS' ( 'ENABLE' | 'DISABLE' ) PluginNameList | 'REPAIR' 'TABLE' TableName
        ↪ CreateTableStmt | ( 'FLUSH' | 'CAPTURE' | 'EVOLVE' ) 'BINDINGS' )

NumList ::=
    Int64Num ( ',' Int64Num )*
```

14.11.2.4.2 示例

可以通过 ADMIN CANCEL DDL JOBS 语句取消当前正在运行的 DDL 作业，并返回对应作业是否取消成功：

```
ADMIN CANCEL DDL JOBS job_id [, job_id] ...;
```

如果取消失败，会显示失败的具体原因。

> 注意：
>
> - 只有该操作可以取消 DDL 作业，其他所有的操作和环境变更（例如机器重启、集群重启）都不会取消 DDL 作业。
> - 该操作可以同时取消多个 DDL 作业，可以通过ADMIN SHOW DDL JOBS 语句来获取 DDL 作业的 job_id。
> - 如果希望取消的作业已经执行完毕，取消操作将失败。

14.11.2.4.3 MySQL 兼容性

ADMIN CANCEL DDL 语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.4.4 另请参阅

- ADMIN SHOW DDL [JOBS|QUERIES]

14.11.2.5 ADMIN CHECKSUM TABLE

ADMIN CHECKSUM TABLE 语句用于计算表中所有行和索引的 CRC64 校验和。在 TiDB Lightning 等程序中，可通过此语句来确保导入操作成功。

14.11.2.5.1 语法图

```
AdminStmt ::=
    'ADMIN' ( 'SHOW' ( 'DDL' ( 'JOBS' Int64Num? WhereClauseOptional | 'JOB' 'QUERIES' NumList )?
        ↪ | TableName 'NEXT_ROW_ID' | 'SLOW' AdminShowSlow ) | 'CHECK' ( 'TABLE' TableNameList
        ↪ | 'INDEX' TableName Identifier ( HandleRange ( ',' HandleRange )* )? ) | 'RECOVER' '
        ↪ INDEX' TableName Identifier | 'CLEANUP' ( 'INDEX' TableName Identifier | 'TABLE' '
```

```
        ↪ LOCK' TableNameList ) | 'CHECKSUM' 'TABLE' TableNameList | 'CANCEL' 'DDL' 'JOBS'
        ↪ NumList | 'RELOAD' ( 'EXPR_PUSHDOWN_BLACKLIST' | 'OPT_RULE_BLACKLIST' | 'BINDINGS' )
        ↪ | 'PLUGINS' ( 'ENABLE' | 'DISABLE' ) PluginNameList | 'REPAIR' 'TABLE' TableName
        ↪ CreateTableStmt | ( 'FLUSH' | 'CAPTURE' | 'EVOLVE' ) 'BINDINGS' )

TableNameList ::=
    TableName ( ',' TableName )*
```

14.11.2.5.2 示例

创建表 `t1`：

```
CREATE TABLE t1(id INT PRIMARY KEY);
```

插入一些数据：

```
INSERT INTO t1 VALUES (1),(2),(3);
```

计算表 `t1` 的校验和：

```
ADMIN CHECKSUM TABLE t1;
```

输出结果示例如下：

```
+---------+------------+---------------------+-----------+-------------+
| Db_name | Table_name | Checksum_crc64_xor  | Total_kvs | Total_bytes |
+---------+------------+---------------------+-----------+-------------+
| test    | t1         | 10909174369497628533 |         3 |          75 |
+---------+------------+---------------------+-----------+-------------+
1 row in set (0.00 sec)
```

14.11.2.5.3 MySQL 兼容性

`ADMIN CHECKSUM TABLE` 语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.6 ADMIN CHECK [TABLE|INDEX]

`ADMIN CHECK [TABLE|INDEX]` 语句用于校验表中数据和对应索引的一致性，不支持校验外键约束。

14.11.2.6.1 语法图

```
AdminStmt ::=
    'ADMIN' ( 'SHOW' ( 'DDL' ( 'JOBS' Int64Num? WhereClauseOptional | 'JOB' 'QUERIES' NumList )?
        ↪ | TableName 'NEXT_ROW_ID' | 'SLOW' AdminShowSlow ) | 'CHECK' ( 'TABLE' TableNameList
        ↪ | 'INDEX' TableName Identifier ( HandleRange ( ',' HandleRange )* )? ) | 'RECOVER' '
        ↪ INDEX' TableName Identifier | 'CLEANUP' ( 'INDEX' TableName Identifier | 'TABLE' '
        ↪ LOCK' TableNameList ) | 'CHECKSUM' 'TABLE' TableNameList | 'CANCEL' 'DDL' 'JOBS'
```

```
        ↪ NumList | 'RELOAD' ( 'EXPR_PUSHDOWN_BLACKLIST' | 'OPT_RULE_BLACKLIST' | 'BINDINGS' )
        ↪ | 'PLUGINS' ( 'ENABLE' | 'DISABLE' ) PluginNameList | 'REPAIR' 'TABLE' TableName
        ↪ CreateTableStmt | ( 'FLUSH' | 'CAPTURE' | 'EVOLVE' ) 'BINDINGS' )

TableNameList ::=
    TableName ( ',' TableName )*
```

14.11.2.6.2 示例

可以通过 ADMIN CHECK TABLE 语句校验 tbl_name 表中所有数据和对应索引的一致性：

```
ADMIN CHECK TABLE tbl_name [, tbl_name] ...;
```

若通过一致性校验，则返回空的查询结果；否则返回数据不一致的错误信息。

```
ADMIN CHECK INDEX tbl_name idx_name;
```

以上语句用于对 tbl_name 表中 idx_name 索引对应列数据和索引数据进行一致性校验。若通过校验，则返回空的查询结果；否则返回数据不一致的错误信息。

```
ADMIN CHECK INDEX tbl_name idx_name (lower_val, upper_val) [, (lower_val, upper_val)] ...;
```

以上语句用于对 tbl_name 表中 idx_name 索引对应列数据和索引数据进行一致性校验，并且指定了需要检查的数据范围。若通过校验，则返回空的查询结果；否则返回数据不一致的错误信息。

14.11.2.6.3 MySQL 兼容性

ADMIN CHECK [TABLE|INDEX] 语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.6.4 另请参阅

- ADMIN REPAIR

14.11.2.7 ADMIN CLEANUP INDEX

ADMIN CLEANUP INDEX 语句用于在表发生行数据和索引的一致性故障时，删除表中多余的索引，使表的行数据和索引恢复一致状态。注意，该语法尚不支持外键约束。

14.11.2.7.1 语法图

```
AdminCleanupStmt ::=
    'ADMIN' 'CLEANUP' ( 'INDEX' TableName IndexName | 'TABLE' 'LOCK' TableNameList )

TableNameList ::=
    TableName ( ',' TableName )*
```

14.11.2.7.2 示例

假设由于一些原因（例如灾难恢复场景，集群中丢失了部分行数据），数据库中的 tbl 表出现数据和索引不一致现象：

```
SELECT * FROM tbl;
ERROR 1105 (HY000): inconsistent index idx handle count 3 isn't equal to value count 2

ADMIN CHECK INDEX tbl idx ;
ERROR 1105 (HY000): handle &kv.CommonHandle{encoded:[]uint8{0x1, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0
    ↪ x0, 0x0, 0xf8}, colEndOffsets:[]uint16{0xa}}, index:types.Datum{k:0x5, decimal:0x0,
    ↪ length:0x0, i:0, collation:"utf8mb4_bin", b:[]uint8{0x0}, x:interface {}(nil)} != record
    ↪ :<nil>
```

从 SELECT 查询的错误信息可以看到，tbl 表中包含 2 条行数据和 3 条索引数据，这意味着行数据与索引数据出现了不一致故障，同时至少有 1 条索引处于悬空状态。此时可以使用 ADMIN CLEANUP INDEX 语句删除悬空的索引：

```
ADMIN CLEANUP INDEX tbl idx;
```

执行结果示例如下：

```
ADMIN CLEANUP INDEX tbl idx;
+---------------+
| REMOVED_COUNT |
+---------------+
|             1 |
+---------------+
```

此时，可以重新使用 ADMIN CHECK INDEX 语句检查数据索引的一致性，验证数据是否恢复到正常状态：

```
ADMIN CHECK INDEX tbl idx;
Query OK, 0 rows affected (0.01 sec)
```

注意：

当由于副本丢失导致数据索引出现不一致时：

- 通常行数据与索引数据都有丢失，此时需要同时使用 ADMIN CLEANUP INDEX 与ADMIN ↪ RECOVER INDEX 语句保证行数据与索引数据的一致性。
- ADMIN CLEANUP INDEX 总是单并发执行。在表数据量大时，建议通过重建索引的方式进行索引数据的恢复。
- ADMIN CLEANUP INDEX 在执行期间，不会为对应的表或索引记录加锁，TiDB 允许其他的会话同时对表记录进行更新或修改。然而，在此情况下，ADMIN CLEANUP INDEX 可能无法正确处理所有表记录。因此，在执行 ADMIN CLEANUP INDEX 时，请避免同时修改表数据。
- 若你使用 TiDB 企业版，此时建议提交工单联系 PingCAP 支持工程师进行处理。

> ADMIN CLEANUP INDEX 命令不具有原子性：若该命令在执行期间中断，建议重新执行直到成功。

14.11.2.7.3 MySQL 兼容性

ADMIN CLEANUP INDEX 语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.7.4 另请参阅

- ADMIN CHECK TABLE/INDEX
- ADMIN RECOVER INDEX

14.11.2.8 ADMIN RECOVER INDEX

ADMIN RECOVER INDEX 语句用于在表发生行数据和索引的一致性故障时，根据表中多余的索引，使表的行数据和索引重新恢复到一致状态。注意，该语法尚不支持外键约束。

14.11.2.8.1 语法图

```
AdminCleanupStmt ::=
    'ADMIN' 'RECOVER' 'INDEX' TableName IndexName
```

14.11.2.8.2 示例

假设由于一些原因（例如灾难恢复场景，集群中丢失了部分索引数据），数据库中的 tbl 表出现行数据和索引不一致现象：

```
SELECT * FROM tbl;
ERROR 1105 (HY000): inconsistent index idx handle count 2 isn't equal to value count 3

ADMIN CHECK INDEX tbl idx ;
ERROR 1105 (HY000): handle &kv.CommonHandle{encoded:[]uint8{0x1, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0
    ↪ x0, 0x0, 0xf8}, colEndOffsets:[]uint16{0xa}}, index:types.Datum{k:0x5, decimal:0x0,
    ↪ length:0x0, i:0, collation:"utf8mb4_bin", b:[]uint8{0x0}, x:interface {}(nil)} != record
    ↪ :<nil>
```

从 SELECT 查询的错误信息可以看到，tbl 表中包含 3 条行数据和 2 条索引数据，这意味着行数据与索引数据出现了不一致故障，同时至少有 1 条行数据缺少了对应的索引。此时可以使用 ADMIN RECOVER INDEX 语句补充缺少的索引：

```
ADMIN RECOVER INDEX tbl idx;
```

执行结果示例如下：

```
ADMIN RECOVER INDEX tbl idx;
+-------------+------------+
| ADDED_COUNT | SCAN_COUNT |
+-------------+------------+
|           1 |          3 |
+-------------+------------+
1 row in set (0.00 sec)
```

此时，可以重新使用 ADMIN CHECK INDEX 语句对数据索引的一致性进行检查，验证数据恢复到正常状态：

```
ADMIN CHECK INDEX tbl idx;
Query OK, 0 rows affected (0.01 sec)
```

> 注意：
>
> 当由于副本丢失导致数据索引出现不一致时：
>
> - 通常行数据与索引数据都有丢失，此时需要同时使用ADMIN CLEANUP INDEX 与 ADMIN RECOVER INDEX 语句保证行数据与索引数据的一致性。
> - ADMIN RECOVER INDEX 总是单并发执行。在表数据量大时，建议通过重建索引的方式进行索引数据的恢复。
> - ADMIN RECOVER INDEX 在执行期间，不会为对应的表或索引记录加锁，这意味着 TiDB 允许同时有其他的会话对表记录进行更新或修改。然而，在此情况下 ADMIN RECOVER INDEX 可能无法正确处理所有数据。因此，在执行 ADMIN RECOVER INDEX 时，请避免同时修改表数据。
> - 若你使用 TiDB 企业版，此时建议提交工单联系 PingCAP 支持工程师进行处理。
>
> ADMIN RECOVER INDEX 命令不具有原子性：若该命令在执行期间中断，建议重新执行直到成功。

14.11.2.8.3 MySQL 兼容性

ADMIN RECOVER INDEX 语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.8.4 另请参阅

- ADMIN CHECK TABLE/INDEX
- ADMIN CLEANUP INDEX

14.11.2.9 ADMIN SHOW DDL [JOBS|JOB QUERIES]

ADMIN SHOW DDL [JOBS|JOB QUERIES] 语句显示了正在运行和最近完成的 DDL 作业的信息。

14.11.2.9.1 语法图

```
AdminStmt ::=
    'ADMIN' ( 'SHOW' ( 'DDL' ( 'JOBS' Int64Num? WhereClauseOptional | 'JOB' 'QUERIES' NumList | '
        ↪ JOB' 'QUERIES' 'LIMIT' m 'OFFSET' n )? | TableName 'NEXT_ROW_ID' | 'SLOW'
        ↪ AdminShowSlow ) | 'CHECK' ( 'TABLE' TableNameList | 'INDEX' TableName Identifier (
        ↪ HandleRange ( ',' HandleRange )* )? ) | 'RECOVER' 'INDEX' TableName Identifier | '
        ↪ CLEANUP' ( 'INDEX' TableName Identifier | 'TABLE' 'LOCK' TableNameList ) | 'CHECKSUM'
        ↪  'TABLE' TableNameList | 'CANCEL' 'DDL' 'JOBS' NumList | 'RELOAD' ( '
        ↪ EXPR_PUSHDOWN_BLACKLIST' | 'OPT_RULE_BLACKLIST' | 'BINDINGS' ) | 'PLUGINS' ( 'ENABLE'
        ↪  | 'DISABLE' ) PluginNameList | 'REPAIR' 'TABLE' TableName CreateTableStmt | ( 'FLUSH
        ↪ ' | 'CAPTURE' | 'EVOLVE' ) 'BINDINGS' )

NumList ::=
    Int64Num ( ',' Int64Num )*

WhereClauseOptional ::=
    WhereClause?
```

14.11.2.9.2 示例

ADMIN SHOW DDL

可以通过 ADMIN SHOW DDL 语句查看当前正在运行的 DDL 作业状态，包括当前 schema 版本号、Owner 的 DDL ID 和地址、正在执行的 DDL 任务和 SQL、当前 TiDB 实例的 DDL ID。

```
ADMIN SHOW DDL;
```

```
ADMIN SHOW DDL;
+--
    ↪ ----------+--------------------------------------+--------------+-------------+----------------------
    ↪
| SCHEMA_VER | OWNER_ID                             | OWNER_ADDRESS | RUNNING_JOBS | SELF_ID
    ↪                              | QUERY |
+--
    ↪ ----------+--------------------------------------+--------------+-------------+----------------------
    ↪
|         26 | 2d1982af-fa63-43ad-a3d5-73710683cc63 | 0.0.0.0:4000  |              | 2d1982af-
    ↪ fa63-43ad-a3d5-73710683cc63 |       |
+--
    ↪ ----------+--------------------------------------+--------------+-------------+----------------------
    ↪
1 row in set (0.00 sec)
```

ADMIN SHOW DDL JOBS

ADMIN SHOW DDL JOBS 语句用于查看当前 DDL 作业队列中的所有结果（包括正在运行以及等待运行的任务）以及已执行完成的 DDL 作业队列中的最近十条结果。该语句的返回结果字段描述如下：

- JOB_ID：每个 DDL 操作对应一个 DDL 任务，JOB_ID 全局唯一。
- DB_NAME：执行 DDL 操作的数据库的名称。
- TABLE_NAME：执行 DDL 操作的表的名称。
- JOB_TYPE：DDL 操作的类型。
- SCHEMA_STATE：DDL 所操作的 schema 对象的当前状态。如果 JOB_TYPE 是 ADD INDEX，则为索引的状态；如果是 ADD COLUMN，则为列的状态；如果是 CREATE TABLE，则为表的状态。常见的状态有以下几种：
  - none：表示不存在。一般 DROP 操作或者 CREATE 操作失败回滚后，会变为 none 状态。
  - delete only、write only、delete reorganization、write reorganization：这四种状态是中间状态，具体含义请参考TiDB 中在线 DDL 异步变更的原理。由于中间状态转换很快，一般操作中看不到这几种状态，只有执行 ADD INDEX 操作时能看到处于 write reorganization 状态，表示正在添加索引数据。
  - public：表示存在且对用户可用。一般 CREATE TABLE 和 ADD INDEX（或 ADD COLUMN）等操作完成后，会变为 public 状态，表示新建的表、列、索引可以正常读写了。
- SCHEMA_ID：执行 DDL 操作的数据库的 ID。
- TABLE_ID：执行 DDL 操作的表的 ID。
- ROW_COUNT：执行 ADD INDEX 操作时，当前已经添加完成的数据行数。
- START_TIME：DDL 操作的开始时间。
- STATE：DDL 操作的状态。常见的状态有以下几种：
  - queueing：表示该操作任务已经进入 DDL 任务队列中，但尚未执行，因为还在排队等待前面的 DDL 任务完成。另一种原因可能是执行 DROP 操作后，会变为 none 状态，但是很快会更新为 synced 状态，表示所有 TiDB 实例都已经同步到该状态。
  - running：表示该操作正在执行。
  - synced：表示该操作已经执行成功，且所有 TiDB 实例都已经同步该状态。
  - rollback done：表示该操作执行失败，回滚完成。
  - rollingback：表示该操作执行失败，正在回滚。
  - cancelling：表示正在取消该操作。这个状态只有在用 ADMIN CANCEL DDL JOBS 命令取消 DDL 任务时才会出现。

示例如下：

```
ADMIN SHOW DDL JOBS;
```

```
+--
    ↪ ------+---------+--------------------+-------------+----------------------+-----------+----------+------
    ↪
| JOB_ID | DB_NAME | TABLE_NAME         | JOB_TYPE     | SCHEMA_STATE          | SCHEMA_ID |
    ↪ TABLE_ID | ROW_COUNT | CREATE_TIME          | START_TIME          | END_TIME            |
    ↪ STATE    |
+--
    ↪ ------+---------+--------------------+-------------+----------------------+-----------+----------+------
    ↪
|     59 | test    | t1                 | add index    | write reorganization |         1 |
    ↪       55 |     88576 | 2020-08-17 07:51:58 | 2020-08-17 07:51:58 | NULL                |
    ↪ running |
```

```
|     60 | test    | t2                 | add index     | none                 |         1 |
    ↪        57 |         0 | 2020-08-17 07:51:59 | 2020-08-17 07:51:59 | NULL                |
    ↪ none    |
|     58 | test    | t2                 | create table | public               |         1 |
    ↪        57 |         0 | 2020-08-17 07:41:28 | 2020-08-17 07:41:28 | 2020-08-17 07:41:28 |
    ↪ synced  |
|     56 | test    | t1                 | create table | public               |         1 |
    ↪        55 |         0 | 2020-08-17 07:41:02 | 2020-08-17 07:41:02 | 2020-08-17 07:41:02 |
    ↪ synced  |
|     54 | test    | t1                 | drop table    | none                 |         1 |
    ↪        50 |         0 | 2020-08-17 07:41:02 | 2020-08-17 07:41:02 | 2020-08-17 07:41:02 |
    ↪ synced  |
|     53 | test    | t1                 | drop index    | none                 |         1 |
    ↪        50 |         0 | 2020-08-17 07:35:44 | 2020-08-17 07:35:44 | 2020-08-17 07:35:44 |
    ↪ synced  |
|     52 | test    | t1                 | add index     | public               |         1 |
    ↪        50 |    451010 | 2020-08-17 07:34:43 | 2020-08-17 07:34:43 | 2020-08-17 07:35:16 |
    ↪ synced  |
|     51 | test    | t1                 | create table | public               |         1 |
    ↪        50 |         0 | 2020-08-17 07:34:02 | 2020-08-17 07:34:02 | 2020-08-17 07:34:02 |
    ↪ synced  |
|     49 | test    | t1                 | drop table    | none                 |         1 |
    ↪        47 |         0 | 2020-08-17 07:34:02 | 2020-08-17 07:34:02 | 2020-08-17 07:34:02 |
    ↪ synced  |
|     48 | test    | t1                 | create table | public               |         1 |
    ↪        47 |         0 | 2020-08-17 07:33:37 | 2020-08-17 07:33:37 | 2020-08-17 07:33:37 |
    ↪ synced  |
|     46 | mysql   | stats_extended     | create table | public               |         3 |
    ↪        45 |         0 | 2020-08-17 06:42:38 | 2020-08-17 06:42:38 | 2020-08-17 06:42:38 |
    ↪ synced  |
|     44 | mysql   | opt_rule_blacklist | create table | public               |         3 |
    ↪        43 |         0 | 2020-08-17 06:42:38 | 2020-08-17 06:42:38 | 2020-08-17 06:42:38 |
    ↪ synced  |
+--
    ↪ ------+---------+--------------------+--------------+----------------------+-----------+----------+------
    ↪
12 rows in set (0.00 sec)
```

由上述 ADMIN 查询结果可知：

- job_id 为 59 的 DDL 作业当前正在进行中（STATE 列显示为 running）。SCHEMA_STATE 列显示了表当前处于 write reorganization 状态，一旦任务完成，将更改为 public，以便用户会话可以公开观察到状态变更。end_time 列显示为 NULL，表明当前作业的完成时间未知。
- job_id 为 60 的 JOB_TYPE 显示为 add index，表明正在排队等待 job_id 为 59 的作业完成。当作业 59 完成时，作业 60 的 STATE 将更改为 running。

- 对于破坏性的更改（例如删除索引或删除表），当作业完成时，SCHEMA_STATE 将变为 none。对于附加更改，SCHEMA_STATE 将变为 public。

若要限制表中显示的行数，可以指定 NUM 和 WHERE 条件：

```
ADMIN SHOW DDL JOBS [NUM] [WHERE where_condition];
```

- NUM：用于查看已经执行完成的 DDL 作业队列中最近 NUM 条结果；未指定时，默认值为 10。
- WHERE：WHERE 子句，用于添加过滤条件。

ADMIN SHOW DDL JOB QUERIES

ADMIN SHOW DDL JOB QUERIES 语句用于查看 job_id 对应的 DDL 任务的原始 SQL 语句：

```
ADMIN SHOW DDL JOBS;
ADMIN SHOW DDL JOB QUERIES 51;
```

```
ADMIN SHOW DDL JOB QUERIES 51;
+-------------------------------------------------------------+
| QUERY                                                       |
+-------------------------------------------------------------+
| CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY auto_increment) |
+-------------------------------------------------------------+
1 row in set (0.02 sec)
```

只能在 DDL 历史作业队列中最近十条结果中搜索与 job_id 对应的正在运行中的 DDL 作业。

ADMIN SHOW DDL JOB QUERIES LIMIT m OFFSET n

ADMIN SHOW DDL JOB QUERIES LIMIT m OFFSET n 语句用于查看指定范围 [n+1, n+m] 的 job_id 对应的 DDL 任务的原始 SQL 语句：

```
ADMIN SHOW DDL JOB QUERIES LIMIT m;           # -- 取出前 m 行
ADMIN SHOW DDL JOB QUERIES LIMIT n, m;        # -- 取出第 n+1 到 n+m 行
ADMIN SHOW DDL JOB QUERIES LIMIT m OFFSET n;  # -- 取出第 n+1 到 n+m 行
```

以上语法中 n 和 m 都是非负整数。语法的具体示例如下：

```
ADMIN SHOW DDL JOB QUERIES LIMIT 3;  # Retrieve first 3 rows
+--------+-------------------------------------------------------------+
| JOB_ID | QUERY                                                       |
+--------+-------------------------------------------------------------+
|     59 | ALTER TABLE t1 ADD INDEX index2 (col2)                      |
|     60 | ALTER TABLE t2 ADD INDEX index1 (col1)                      |
|     58 | CREATE TABLE t2 (id INT NOT NULL PRIMARY KEY auto_increment) |
+--------+-------------------------------------------------------------+
3 rows in set (0.00 sec)
```

```
ADMIN SHOW DDL JOB QUERIES LIMIT 6, 2;  # Retrieve rows 7-8
+--------+-------------------------------------------------------------------------------+
| JOB_ID | QUERY                                                                         |
+--------+-------------------------------------------------------------------------------+
|     52 | ALTER TABLE t1 ADD INDEX index1 (col1)                                        |
|     51 | CREATE TABLE IF NOT EXISTS t1 (id INT NOT NULL PRIMARY KEY auto_increment) |
+--------+-------------------------------------------------------------------------------+
3 rows in set (0.00 sec)
```

```
ADMIN SHOW DDL JOB QUERIES LIMIT 3 OFFSET 4;  # Retrieve rows 5-7
+--------+----------------------------------------+
| JOB_ID | QUERY                                  |
+--------+----------------------------------------+
|     54 | DROP TABLE IF EXISTS t3                |
|     53 | ALTER TABLE t1 DROP INDEX index1       |
|     52 | ALTER TABLE t1 ADD INDEX index1 (col1) |
+--------+----------------------------------------+
3 rows in set (0.00 sec)
```

该语句可以在 DDL 历史作业队列任意指定范围中搜索与 job_id 对应的正在运行中的 DDL 作业，没有 ADMIN SHOW DDL JOB QUERIES 语句的最近 10 条结果的限制。

14.11.2.9.3 MySQL 兼容性

ADMIN SHOW DDL [JOBS|JOB QUERIES] 语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.9.4 另请参阅

- ADMIN CANCEL DDL

14.11.2.10 ADMIN SHOW TELEMETRY

ADMIN SHOW TELEMETRY 语句用于查看通过遥测功能收集到并分享给 PingCAP 的使用信息。

14.11.2.10.1 语法图

```
AdminStmt ::=
    'ADMIN' ( 'SHOW' ( 'DDL' ( 'JOBS' Int64Num? WhereClauseOptional | 'JOB' 'QUERIES' NumList )?
        ↪ | TableName 'NEXT_ROW_ID' | 'SLOW' AdminShowSlow | 'TELEMETRY' ) | 'CHECK' ( 'TABLE'
        ↪ TableNameList | 'INDEX' TableName Identifier ( HandleRange ( ',' HandleRange )* )? )
        ↪ | 'RECOVER' 'INDEX' TableName Identifier | 'CLEANUP' ( 'INDEX' TableName Identifier |
        ↪  'TABLE' 'LOCK' TableNameList ) | 'CHECKSUM' 'TABLE' TableNameList | 'CANCEL' 'DDL' '
        ↪ JOBS' NumList | 'RELOAD' ( 'EXPR_PUSHDOWN_BLACKLIST' | 'OPT_RULE_BLACKLIST' | '
        ↪ BINDINGS' ) | 'PLUGINS' ( 'ENABLE' | 'DISABLE' ) PluginNameList | 'REPAIR' 'TABLE'
        ↪ TableName CreateTableStmt | ( 'FLUSH' | 'CAPTURE' | 'EVOLVE' ) 'BINDINGS' )
```

14.11.2.10.2 示例

```
ADMIN SHOW TELEMETRY\G
```

```
*************************** 1. row ***************************
 TRACKING_ID: a1ba1d97-b940-4d5b-a9d5-ddb0f2ac29e7
 LAST_STATUS: {
  "check_at": "2021-08-11T08:23:38+02:00",
  "is_error": false,
  "error_msg": "",
  "is_request_sent": true
}
DATA_PREVIEW: {
  "hardware": [
    {
      "instanceType": "tidb",
      "listenHostHash": "4b84b15bff6ee5796152495a230e45e3d7e947d9",
      "listenPort": "4000",
      "cpu": {
        "cache": "8192",
        "cpuFrequency": "2301.00MHz",
        "cpuLogicalCores": "8",
        "cpuPhysicalCores": "4"
      },
      "memory": {
        "capacity": "16410021888"
      },
      "disk": {
        "ebbca862689fa9fef7c55c3112e375c4ce575fe4": {
          "deviceName": "ebbca862689fa9fef7c55c3112e375c4ce575fe4",
          "free": "624438726656",
          "freePercent": "0.61",
          "fstype": "btrfs",
          "opts": "bind,rw,relatime",
          "path": "fb365c1216b59e1cfc86950425867007a60f4435",
          "total": "1022488477696",
          "used": "397115568128",
          "usedPercent": "0.39"
        },
        "nvme0n1p1": {
          "deviceName": "nvme0n1p1",
          "free": "582250496",
          "freePercent": "0.93",
          "fstype": "vfat",
          "opts": "rw,relatime",
          "path": "0fc8c8d71702d81a02e216fb6ef19f4dda4973df",
```

```
        "total": "627900416",
        "used": "45649920",
        "usedPercent": "0.07"
      },
      "nvme0n1p2": {
        "deviceName": "nvme0n1p2",
        "free": "701976576",
        "freePercent": "0.74",
        "fstype": "ext4",
        "opts": "rw,relatime",
        "path": "/boot",
        "total": "1023303680",
        "used": "250863616",
        "usedPercent": "0.26"
      }
    }
  },
  {
    "instanceType": "pd",
    "listenHostHash": "4b84b15bff6ee5796152495a230e45e3d7e947d9",
    "listenPort": "2379",
    "cpu": {
      "cache": "8192",
      "cpuFrequency": "2301.00MHz",
      "cpuLogicalCores": "8",
      "cpuPhysicalCores": "4"
    },
    "memory": {
      "capacity": "16410021888"
    },
    "disk": {
      "ebbca862689fa9fef7c55c3112e375c4ce575fe4": {
        "deviceName": "ebbca862689fa9fef7c55c3112e375c4ce575fe4",
        "free": "624438726656",
        "freePercent": "0.61",
        "fstype": "btrfs",
        "opts": "bind,rw,relatime",
        "path": "fb365c1216b59e1cfc86950425867007a60f4435",
        "total": "1022488477696",
        "used": "397115568128",
        "usedPercent": "0.39"
      },
      "nvme0n1p1": {
        "deviceName": "nvme0n1p1",
        "free": "582250496",
```

```
        "freePercent": "0.93",
        "fstype": "vfat",
        "opts": "rw,relatime",
        "path": "0fc8c8d71702d81a02e216fb6ef19f4dda4973df",
        "total": "627900416",
        "used": "45649920",
        "usedPercent": "0.07"
      },
      "nvme0n1p2": {
        "deviceName": "nvme0n1p2",
        "free": "701976576",
        "freePercent": "0.74",
        "fstype": "ext4",
        "opts": "rw,relatime",
        "path": "/boot",
        "total": "1023303680",
        "used": "250863616",
        "usedPercent": "0.26"
      }
    }
  },
  {
    "instanceType": "tikv",
    "listenHostHash": "4b84b15bff6ee5796152495a230e45e3d7e947d9",
    "listenPort": "20160",
    "cpu": {
      "cpuFrequency": "3730MHz",
      "cpuLogicalCores": "8",
      "cpuPhysicalCores": "4",
      "cpuVendorId": "GenuineIntel",
      "l1CacheLineSize": "64",
      "l1CacheSize": "32768",
      "l2CacheLineSize": "64",
      "l2CacheSize": "262144",
      "l3CacheLineSize": "64",
      "l3CacheSize": "8388608"
    },
    "memory": {
      "capacity": "16803861504"
    },
    "disk": {
      "36e7dfacbb83843f83075d78aeb4cf850a4882a1": {
        "deviceName": "36e7dfacbb83843f83075d78aeb4cf850a4882a1",
        "free": "624438726656",
        "freePercent": "0.61",
```

```
        "fstype": "btrfs",
        "path": "fb365c1216b59e1cfc86950425867007a60f4435",
        "total": "1022488477696",
        "used": "398049751040",
        "usedPercent": "0.39"
      },
      "nvme0n1p1": {
        "deviceName": "nvme0n1p1",
        "free": "582250496",
        "freePercent": "0.93",
        "fstype": "vfat",
        "path": "0fc8c8d71702d81a02e216fb6ef19f4dda4973df",
        "total": "627900416",
        "used": "45649920",
        "usedPercent": "0.07"
      },
      "nvme0n1p2": {
        "deviceName": "nvme0n1p2",
        "free": "701976576",
        "freePercent": "0.69",
        "fstype": "ext4",
        "path": "/boot",
        "total": "1023303680",
        "used": "321327104",
        "usedPercent": "0.31"
      }
    }
  },
  {
    "instanceType": "tiflash",
    "listenHostHash": "4b84b15bff6ee5796152495a230e45e3d7e947d9",
    "listenPort": "3930",
    "cpu": {
      "cpuFrequency": "3400MHz",
      "cpuLogicalCores": "8",
      "cpuPhysicalCores": "4",
      "l1CacheLineSize": "64",
      "l1CacheSize": "32768",
      "l2CacheLineSize": "64",
      "l2CacheSize": "262144",
      "l3CacheLineSize": "64",
      "l3CacheSize": "8388608"
    },
    "memory": {
      "capacity": "16410021888"
```

```
    },
    "disk": {
      "36e7dfacbb83843f83075d78aeb4cf850a4882a1": {
        "deviceName": "36e7dfacbb83843f83075d78aeb4cf850a4882a1",
        "free": "624438726656",
        "freePercent": "0.61",
        "fstype": "btrfs",
        "path": "fb365c1216b59e1cfc86950425867007a60f4435",
        "total": "1022488477696",
        "used": "398049751040",
        "usedPercent": "0.39"
      }
    }
  }
],
"instances": [
  {
    "instanceType": "tidb",
    "listenHostHash": "4b84b15bff6ee5796152495a230e45e3d7e947d9",
    "listenPort": "4000",
    "statusHostHash": "4b84b15bff6ee5796152495a230e45e3d7e947d9",
    "statusPort": "10080",
    "version": "5.1.1",
    "gitHash": "797bddd25310ed42f0791c8eccb78be8cce2f502",
    "startTime": "2021-08-11T08:23:38+02:00",
    "upTime": "22.210217487s"
  },
  {
    "instanceType": "pd",
    "listenHostHash": "4b84b15bff6ee5796152495a230e45e3d7e947d9",
    "listenPort": "2379",
    "statusHostHash": "4b84b15bff6ee5796152495a230e45e3d7e947d9",
    "statusPort": "2379",
    "version": "5.1.1",
    "gitHash": "7cba1912b317a533e18b16ea2ba9a14ed2891129",
    "startTime": "2021-08-11T08:23:32+02:00",
    "upTime": "28.210220368s"
  },
  {
    "instanceType": "tikv",
    "listenHostHash": "4b84b15bff6ee5796152495a230e45e3d7e947d9",
    "listenPort": "20160",
    "statusHostHash": "4b84b15bff6ee5796152495a230e45e3d7e947d9",
    "statusPort": "20180",
    "version": "5.1.1",
```

```
      "gitHash": "4705d7c6e9c42d129d3309e05911ec6b08a25a38",
      "startTime": "2021-08-11T08:23:33+02:00",
      "upTime": "27.210221447s"
    },
    {
      "instanceType": "tiflash",
      "listenHostHash": "4b84b15bff6ee5796152495a230e45e3d7e947d9",
      "listenPort": "3930",
      "statusHostHash": "4b84b15bff6ee5796152495a230e45e3d7e947d9",
      "statusPort": "20292",
      "version": "v5.1.1",
      "gitHash": "c8fabfb50fe28db17cc5118133a69be255c40efd",
      "startTime": "2021-08-11T08:23:40+02:00",
      "upTime": "20.210222452s"
    }
  ],
  "hostExtra": {
    "cpuFlags": [
      "fpu",
      "vme",
      "de",
      "pse",
      "tsc",
      "msr",
      "pae",
      "mce",
      "cx8",
      "apic",
      "sep",
      "mtrr",
      "pge",
      "mca",
      "cmov",
      "pat",
      "pse36",
      "clflush",
      "dts",
      "acpi",
      "mmx",
      "fxsr",
      "sse",
      "sse2",
      "ss",
      "ht",
      "tm",
```

```
        "pbe",
        "syscall",
        "nx",
        "pdpe1gb",
        "rdtscp",
        "lm",
        "constant_tsc",
        "art",
        "arch_perfmon",
        "pebs",
        "bts",
        "rep_good",
        "nopl",
        "xtopology",
        "nonstop_tsc",
        "cpuid",
        "aperfmperf",
        "pni",
        "pclmulqdq",
        "dtes64",
        "monitor",
        "ds_cpl",
        "vmx",
        "est",
        "tm2",
        "ssse3",
        "sdbg",
        "fma",
        "cx16",
        "xtpr",
        "pdcm",
        "pcid",
        "sse4_1",
        "sse4_2",
        "x2apic",
        "movbe",
        "popcnt",
        "tsc_deadline_timer",
        "aes",
        "xsave",
        "avx",
        "f16c",
        "rdrand",
        "lahf_lm",
        "abm",
```

```
        "3dnowprefetch",
        "cpuid_fault",
        "epb",
        "invpcid_single",
        "ssbd",
        "ibrs",
        "ibpb",
        "stibp",
        "ibrs_enhanced",
        "tpr_shadow",
        "vnmi",
        "flexpriority",
        "ept",
        "vpid",
        "ept_ad",
        "fsgsbase",
        "tsc_adjust",
        "sgx",
        "bmi1",
        "avx2",
        "smep",
        "bmi2",
        "erms",
        "invpcid",
        "mpx",
        "rdseed",
        "adx",
        "smap",
        "clflushopt",
        "intel_pt",
        "xsaveopt",
        "xsavec",
        "xgetbv1",
        "xsaves",
        "dtherm",
        "ida",
        "arat",
        "pln",
        "pts",
        "hwp",
        "hwp_notify",
        "hwp_act_window",
        "hwp_epp",
        "md_clear",
        "flush_l1d",
```

```
      "arch_capabilities"
    ],
    "cpuModelName": "Intel(R) Core(TM) i7-1051OU CPU @ 1.8OGHz",
    "os": "linux",
    "platform": "fedora",
    "platformFamily": "fedora",
    "platformVersion": "34",
    "kernelVersion": "5.13.5-200.fc34.x86_64",
    "kernelArch": "x86_64",
    "virtualizationSystem": "kvm",
    "virtualizationRole": "host"
  },
  "reportTimestamp": 1628663040,
  "trackingId": "a1ba1d97-b940-4d5b-a9d5-ddb0f2ac29e7",
  "featureUsage": {
    "txn": {
      "asyncCommitUsed": true,
      "onePCUsed": true,
      "txnCommitCounter": {
        "twoPC": 9,
        "asyncCommit": 0,
        "onePC": 0
      }
    },
    "clusterIndex": {},
    "temporaryTable": false,
    "cte": {
      "nonRecursiveCTEUsed": 0,
      "recursiveUsed": 0,
      "nonCTEUsed": 13
    }
  },
  "windowedStats": [],
  "slowQueryStats": {
    "slowQueryBucket": {}
  }
}
1 row in set (0.0259 sec)
```


14.11.2.10.3 MySQL 兼容性

`ADMIN` 语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.10.4 另请参阅

- 遥测
- 系统变量 tidb_enable_telemetry

14.11.2.11 ALTER DATABASE

ALTER DATABASE 用于修改指定或当前数据库的默认字符集和排序规则。ALTER SCHEMA 跟 ALTER DATABASE 操作效果一样。

14.11.2.11.1 语法图

```
AlterDatabaseStmt ::=
    'ALTER' 'DATABASE' DBName? DatabaseOptionList

DatabaseOption ::=
    DefaultKwdOpt ( CharsetKw '='? CharsetName | 'COLLATE' '='? CollationName | 'ENCRYPTION' '='?
        ↪ EncryptionOpt )
```

14.11.2.11.2 示例

修改数据库 test 的字符集为 utf8mb4：

```
ALTER DATABASE test DEFAULT CHARACTER SET = utf8mb4;
```

```
Query OK, 0 rows affected (0.00 sec)
```

目前 TiDB 只支持部分的字符集和排序规则，详情参阅字符集支持。

14.11.2.11.3 MySQL 兼容性

ALTER DATABASE 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.11.4 另请参阅

- CREATE DATABASE
- SHOW DATABASES

14.11.2.12 ALTER INDEX

ALTER INDEX 语句用于修改索引的可见性，可以将索引设置为 Visible 或者 Invisible。设置为 Invisible 的索引即不可见索引 (Invisible Index) 由 DML 语句维护，不会被查询优化器使用。

14.11.2.12.1 语法图

```
AlterTableStmt
         ::= 'ALTER' 'IGNORE'? 'TABLE' TableName AlterIndexSpec ( ',' AlterIndexSpec )*

AlterIndexSpec
         ::= 'ALTER' 'INDEX' Identifier ( 'VISIBLE' | 'INVISIBLE' )
```

14.11.2.12.2 示例

可以通过 `ALTER TABLE ... ALTER INDEX ...` 语句，修改索引的可见性：

```
CREATE TABLE t1 (c1 INT, UNIQUE(c1));
ALTER TABLE t1 ALTER INDEX c1 INVISIBLE;
```

```
Query OK, 0 rows affected (0.02 sec)
```

```
SHOW CREATE TABLE t1;
```

```
+--
    ↪ -----+----------------------------------------------------------------------------------------------------
    ↪
| Table | Create Table
                                      |
+--
    ↪ -----+----------------------------------------------------------------------------------------------------
    ↪
| t1    | CREATE TABLE `t1` (
  `c1` int(11) DEFAULT NULL,
  UNIQUE KEY `c1` (`c1`) /*!80000 INVISIBLE */
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin |
+--
    ↪ -----+----------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

优化器将无法使用 `c1` 这个不可见的索引：

```
EXPLAIN SELECT c1 FROM t1 ORDER BY c1;
```

```
+--
    ↪ ----------------------+----------+-----------+---------------+--------------------------------+
    ↪
| id                      | estRows  | task      | access object | operator info
    ↪                   |
```

```
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
| Sort_4                  | 10000.00 | root      |               | test.t1.c1:asc
    ↪                 |
| └─TableReader_8         | 10000.00 | root      |               | data:TableFullScan_7
    ↪           |
|   └─TableFullScan_7     | 10000.00 | cop[tikv] | table:t1      | keep order:false, stats:
    ↪ pseudo |
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
3 rows in set (0.00 sec)
```

作为对比，c2 是可见的索引，优化器将可以使用索引：

```
EXPLAIN SELECT c2 FROM t1 ORDER BY c2;
```

```
+--
    ↪ ----------------------+----------+-----------+-----------------------+------------------------------+
    ↪
| id                      | estRows  | task      | access object         | operator info
    ↪                 |
+--
    ↪ ----------------------+----------+-----------+-----------------------+------------------------------+
    ↪
| IndexReader_13          | 10000.00 | root      |                       | index:IndexFullScan_12
    ↪         |
| └─IndexFullScan_12      | 10000.00 | cop[tikv] | table:t1, index:c2(c2) | keep order:true,
    ↪ stats:pseudo |
+--
    ↪ ----------------------+----------+-----------+-----------------------+------------------------------+
    ↪
2 rows in set (0.00 sec)
```

即使用 SQL Hint USE INDEX 强制使用索引，优化器也无法使用不可见索引，否则 SQL 语句会报错：

```
SELECT * FROM t1 USE INDEX(c1);
```

```
ERROR 1176 (42000): Key 'c1' doesn't exist in table 't1'
```

> 注意：
>
> “不可见”是仅仅对优化器而言的，不可见索引仍然可以被修改或删除。

```
ALTER TABLE t1 DROP INDEX c1;
```

```
Query OK, 0 rows affected (0.02 sec)
```

14.11.2.12.3 MySQL 兼容性

- TiDB 中的不可见索引是基于 MySQL 8.0 中的同等特性构建的。
- 与 MySQL 类似，TiDB 不允许将主键索引设为不可见。
- MySQL 中提供的优化器开关 `use_invisible_indexes=on` 可将所有的不可见索引重新设为可见。该功能在 TiDB 中不可用。

14.11.2.12.4 另请参阅

- CREATE TABLE
- CREATE INDEX
- ADD INDEX
- DROP INDEX
- RENAME INDEX

14.11.2.13 ALTER INSTANCE

`ALTER INSTANCE` 语句用于对单个 TiDB 实例进行变更操作。目前 TiDB 仅支持 `RELOAD TLS` 子句。

14.11.2.13.1 RELOAD TLS

`ALTER INSTANCE RELOAD TLS` 语句用于从原配置的证书 (`ssl-cert`)、密钥 (`ssl-key`) 和 CA (`ssl-ca`) 的路径重新加证书、密钥和 CA。

新加载的证书密钥和 CA 将在语句执行成功后对新建立的连接生效，不会影响语句执行前已建立的连接。

在重加载遇到错误时默认会报错返回且继续使用变更前的密钥和证书，但在添加可选的 `NO ROLLBACK ON ERROR` 后遇到错误将不报错并以关闭 TLS 安全连接功能的方式处理后续请求。

14.11.2.13.2 语法图

```
AlterInstanceStmt ::=
    'ALTER' 'INSTANCE' InstanceOption

InstanceOption ::=
    'RELOAD' 'TLS' ('NO' 'ROLLBACK' 'ON' 'ERROR')?
```

14.11.2.13.3 示例

```
ALTER INSTANCE RELOAD TLS;
```

14.11.2.13.4 MySQL 兼容性

仅支持从原配置路径重加载，不支持动态修改加载路径，也不支持动态启用启动 TiDB 时未开启的 TLS 加密连接功能。

14.11.2.13.5 另请参阅

为 TiDB 客户端服务端间通信开启加密传输

14.11.2.14 ALTER PLACEMENT POLICY

`ALTER PLACEMENT POLICY` 用于修改已创建的放置策略。此修改会自动更新至所有使用这些放置策略的表和分区。

`ALTER PLACEMENT POLICY` 会完全替换之前定义的规则，而不会和之前的规则合并，比如在下面的例子中，`FOLLOWERS=4` 就被 `ALTER PLACEMENT POLICY` 语句覆盖了：

```
CREATE PLACEMENT POLICY p1 FOLLOWERS=4;
ALTER PLACEMENT POLICY p1 PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1";
```

14.11.2.14.1 语法图

```
AlterPolicyStmt ::=
    "ALTER" "PLACEMENT" "POLICY" IfExists PolicyName PlacementOptionList

PolicyName ::=
    Identifier

PlacementOptionList ::=
    PlacementOption
|   PlacementOptionList PlacementOption
|   PlacementOptionList ',' PlacementOption

PlacementOption ::=
    CommonPlacementOption
|   SugarPlacementOption
|   AdvancedPlacementOption

CommonPlacementOption ::=
    "FOLLOWERS" EqOpt LengthNum

SugarPlacementOption ::=
    "PRIMARY_REGION" EqOpt stringLit
|   "REGIONS" EqOpt stringLit
|   "SCHEDULE" EqOpt stringLit

AdvancedPlacementOption ::=
```

```
    "LEARNERS" EqOpt LengthNum
|   "CONSTRAINTS" EqOpt stringLit
|   "LEADER_CONSTRAINTS" EqOpt stringLit
|   "FOLLOWER_CONSTRAINTS" EqOpt stringLit
|   "LEARNER_CONSTRAINTS" EqOpt stringLit
```

14.11.2.14.2 示例

> **注意：**
>
> 如要查看所在集群中可用的区域，见SHOW PLACEMENT LABELS。如果未看到任何可用的区域，此 TiKV 集群在部署时可能未正确设置标签 (label)。

```
CREATE PLACEMENT POLICY p1 PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1";
CREATE TABLE t1 (i INT) PLACEMENT POLICY=p1; -- 绑定放置策略 p1 到表 t1。
ALTER PLACEMENT POLICY p1 PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1,us-west-2"
    ↪ FOLLOWERS=4; -- t1 上的放置规则会自动更新。
SHOW CREATE PLACEMENT POLICY p1\G;
```

```
Query OK, 0 rows affected (0.08 sec)

Query OK, 0 rows affected (0.10 sec)

***************************[ 1. row ]***************************
Policy        | p1
Create Policy | CREATE PLACEMENT POLICY `p1` PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-
    ↪ west-1,us-west-2" FOLLOWERS=4
1 row in set (0.00 sec)
```

14.11.2.14.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.14.4 另请参阅

- Placement Rules in SQL
- SHOW PLACEMENT
- CREATE PLACEMENT POLICY
- DROP PLACEMENT POLICY

14.11.2.15 ALTER RESOURCE GROUP

`ALTER RESOURCE GROUP` 语句用于在当前所选数据库中修改资源组。

14.11.2.15.1 语法图

```
AlterResourceGroupStmt ::=
   "ALTER" "RESOURCE" "GROUP" IfExists ResourceGroupName ResourceGroupOptionList

IfExists ::=
    ('IF' 'EXISTS')?

ResourceGroupName ::=
   Identifier

ResourceGroupOptionList ::=
    DirectResourceGroupOption
|   ResourceGroupOptionList DirectResourceGroupOption
|   ResourceGroupOptionList ',' DirectResourceGroupOption

DirectResourceGroupOption ::=
    "RU_PER_SEC" EqOpt stringLit
|   "PRIORITY" EqOpt ResourceGroupPriorityOption
|   "BURSTABLE"

ResourceGroupPriorityOption ::=
    LOW
|   MEDIUM
|   HIGH
```

TiDB 支持以下 DirectResourceGroupOption, 其中Request Unit (RU) 是 TiDB 对 CPU、IO 等系统资源统一抽象的单位。

参数	含义	举例
RU_PER_SEC	每秒 RU 填充的速度	RU_PER_SEC = 500 表示此资源组每秒回填 500 个 RU。
PRIORITY	任务在 TiKV 上处理的绝对优先级	PRIORITY = HIGH 表示优先级高。若未指定则默认为 MEDI
BURSTABLE	允许对应的资源组超出配额后使用空余的系统资源。	

> **注意：**
>
> ALTER RESOURCE GROUP 语句只能在全局变量tidb_enable_resource_control 参数设置为 ON 时才能执行。

14.11.2.15.2 示例

创建一个名为 rg1 的资源组，并修改它的属性。

```
DROP RESOURCE GROUP IF EXISTS rg1;
```

```
Query OK, 0 rows affected (0.22 sec)
```

```
CREATE RESOURCE GROUP IF NOT EXISTS rg1
  RU_PER_SEC = 100
  BURSTABLE;
```

```
Query OK, 0 rows affected (0.08 sec)
```

```
SELECT * FROM information_schema.resource_groups WHERE NAME ='rg1';
```

```
+------+------------+----------+-----------+
| NAME | RU_PER_SEC | PRIORITY | BURSTABLE |
+------+------------+----------+-----------+
| rg1  |        100 | MEDIUM   | YES       |
+------+------------+----------+-----------+
1 rows in set (1.30 sec)
```

```
ALTER RESOURCE GROUP rg1
  RU_PER_SEC = 200
  PRIORITY = LOW;
```

```
Query OK, 0 rows affected (0.08 sec)
```

```
SELECT * FROM information_schema.resource_groups WHERE NAME ='rg1';
```

```
+------+------------+----------+-----------+
| NAME | RU_PER_SEC | PRIORITY | BURSTABLE |
+------+------------+----------+-----------+
| rg1  |        200 | LOW      | NO        |
+------+------------+----------+-----------+
1 rows in set (1.30 sec)
```

14.11.2.15.3 MySQL 兼容性

MySQL 也支持 ALTER RESOURCE GROUP，但是接受的参数和 TiDB 不同，两者并不兼容。

14.11.2.15.4 另请参阅

- DROP RESOURCE GROUP
- CREATE RESOURCE GROUP
- RU

14.11.2.16 ALTER TABLE

`ALTER TABLE` 语句用于对已有表进行修改，以符合新表结构。`ALTER TABLE` 语句可用于：

- `ADD`，`DROP`，或`RENAME` 索引
- `ADD`，`DROP`，`MODIFY` 或`CHANGE` 列
- `COMPACT` 表数据

14.11.2.16.1 语法图

```
AlterTableStmt ::=
    'ALTER' IgnoreOptional 'TABLE' TableName (
        AlterTableSpecListOpt AlterTablePartitionOpt |
        'ANALYZE' 'PARTITION' PartitionNameList ( 'INDEX' IndexNameList )? AnalyzeOptionListOpt |
        'COMPACT' ( 'PARTITION' PartitionNameList )? 'TIFLASH' 'REPLICA'
    )

TableName ::=
    Identifier ('.' Identifier)?

AlterTableSpec ::=
    TableOptionList
|   'SET' 'TIFLASH' 'REPLICA' LengthNum LocationLabelList
|   'CONVERT' 'TO' CharsetKw ( CharsetName | 'DEFAULT' ) OptCollate
|   'ADD' ( ColumnKeywordOpt IfNotExists ( ColumnDef ColumnPosition | '(' TableElementList ')' )
    ↪ | Constraint | 'PARTITION' IfNotExists NoWriteToBinLogAliasOpt (
    ↪ PartitionDefinitionListOpt | 'PARTITIONS' NUM ) )
|   ( ( 'CHECK' | 'TRUNCATE' ) 'PARTITION' | ( 'OPTIMIZE' | 'REPAIR' | 'REBUILD' ) 'PARTITION'
    ↪ NoWriteToBinLogAliasOpt ) AllOrPartitionNameList
|   'COALESCE' 'PARTITION' NoWriteToBinLogAliasOpt NUM
|   'DROP' ( ColumnKeywordOpt IfExists ColumnName RestrictOrCascadeOpt | 'PRIMARY' 'KEY' |  '
    ↪ PARTITION' IfExists PartitionNameList | ( KeyOrIndex IfExists | 'CHECK' ) Identifier | '
    ↪ FOREIGN' 'KEY' IfExists Symbol )
|   'EXCHANGE' 'PARTITION' Identifier 'WITH' 'TABLE' TableName WithValidationOpt
|   ( 'IMPORT' | 'DISCARD' ) ( 'PARTITION' AllOrPartitionNameList )? 'TABLESPACE'
|   'REORGANIZE' 'PARTITION' NoWriteToBinLogAliasOpt ReorganizePartitionRuleOpt
|   'ORDER' 'BY' AlterOrderItem ( ',' AlterOrderItem )*
|   ( 'DISABLE' | 'ENABLE' ) 'KEYS'
|   ( 'MODIFY' ColumnKeywordOpt IfExists | 'CHANGE' ColumnKeywordOpt IfExists ColumnName )
    ↪ ColumnDef ColumnPosition
|   'ALTER' ( ColumnKeywordOpt ColumnName ( 'SET' 'DEFAULT' ( SignedLiteral | '(' Expression ')'
    ↪ ) | 'DROP' 'DEFAULT' ) | 'CHECK' Identifier EnforcedOrNot | 'INDEX' Identifier
    ↪ IndexInvisible )
|   'RENAME' ( ( 'COLUMN' | KeyOrIndex ) Identifier 'TO' Identifier | ( 'TO' | '='? | 'AS' )
    ↪ TableName )
|   LockClause
```

```
|   AlgorithmClause
|   'FORCE'
|   ( 'WITH' | 'WITHOUT' ) 'VALIDATION'
|   'SECONDARY_LOAD'
|   'SECONDARY_UNLOAD'
|   ( 'AUTO_INCREMENT' | 'AUTO_ID_CACHE' | 'AUTO_RANDOM_BASE' | 'SHARD_ROW_ID_BITS' ) EqOpt
    ↪ LengthNum
|   ( 'CACHE' | 'NOCACHE' )
|   (
        'TTL' EqOpt TimeColumnName '+' 'INTERVAL' Expression TimeUnit (TTLEnable EqOpt ( 'ON' | '
            ↪ OFF' ))?
        | 'REMOVE' 'TTL'
        | TTLEnable EqOpt ( 'ON' | 'OFF' )
        | TTLJobInterval EqOpt stringLit
    )
|   PlacementPolicyOption

PlacementPolicyOption ::=
    "PLACEMENT" "POLICY" EqOpt PolicyName
|   "PLACEMENT" "POLICY" (EqOpt | "SET") "DEFAULT"
```

14.11.2.16.2 示例

创建一张表，并插入初始数据：

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, c1 INT NOT NULL);
INSERT INTO t1 (c1) VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 0 rows affected (0.11 sec)
Query OK, 5 rows affected (0.03 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

执行以下查询需要扫描全表，因为 c1 列未被索引：

```
EXPLAIN SELECT * FROM t1 WHERE c1 = 3;
```

```
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
| id                    | estRows  | task      | access object | operator info
    ↪                  |
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
| TableReader_7           | 10.00    | root      |               | data:Selection_6
    ↪               |
```

```
| └─Selection_6           | 10.00    | cop[tikv] |               | eq(test.t1.c1, 3)
    ↪              |
|   └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t1      | keep order:false, stats:
    ↪ pseudo |
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
3 rows in set (0.00 sec)
```

你可以使用ALTER TABLE .. ADD INDEX 语句在 t1 表上添加索引。添加后，EXPLAIN 的分析结果显示 SELECT * ↪ FROM t1 WHERE c1 = 3; 查询已使用效率更高的索引范围扫描：

```
ALTER TABLE t1 ADD INDEX (c1);
EXPLAIN SELECT * FROM t1 WHERE c1 = 3;
```

```
Query OK, 0 rows affected (0.30 sec)
+--
    ↪ ----------------------+---------+-----------+-----------------------+-------------------------------
    ↪
| id                    | estRows | task      | access object         | operator info
    ↪                               |
+--
    ↪ ----------------------+---------+-----------+-----------------------+-------------------------------
    ↪
| IndexReader_6           | 10.00   | root      |                       | index:IndexRangeScan_5
    ↪                         |
| └─IndexRangeScan_5      | 10.00   | cop[tikv] | table:t1, index:c1(c1) | range:[3,3], keep
    ↪ order:false, stats:pseudo |
+--
    ↪ ----------------------+---------+-----------+-----------------------+-------------------------------
    ↪
2 rows in set (0.00 sec)
```

TiDB 允许用户为 DDL 操作指定使用某一种 ALTER 算法。这仅为一种指定，并不改变实际的用于更改表的算法。如果你只想在群集的高峰时段允许即时 DDL 更改，则 ALTER 算法会很有用。示例如下：

```
ALTER TABLE t1 DROP INDEX c1, ALGORITHM=INSTANT;
```

```
Query OK, 0 rows affected (0.24 sec)
```

如果某一 DDL 操作要求使用 INPLACE 算法，而用户指定 ALGORITHM=INSTANT，会导致报错：

```
ALTER TABLE t1 ADD INDEX (c1), ALGORITHM=INSTANT;
```

```
ERROR 1846 (0A000): ALGORITHM=INSTANT is not supported. Reason: Cannot alter table by INSTANT.
    ↪ Try ALGORITHM=INPLACE.
```

但如果为 INPLACE 操作指定 ALGORITHM=COPY，会产生警告而非错误，这是因为 TiDB 将该指定解读为该算法或更好的算法。由于 TiDB 使用的算法可能不同于 MySQL，所以这一行为可用于 MySQL 兼容性。

```
ALTER TABLE t1 ADD INDEX (c1), ALGORITHM=COPY;
SHOW WARNINGS;
```

```
Query OK, 0 rows affected, 1 warning (0.25 sec)
+--
    ↪ -----+------+-------------------------------------------------------------------------------------------------
    ↪
| Level | Code | Message
    ↪                                                                                                          |
+--
    ↪ -----+------+-------------------------------------------------------------------------------------------------
    ↪
| Error | 1846 | ALGORITHM=COPY is not supported. Reason: Cannot alter table by COPY. Try
    ↪ ALGORITHM=INPLACE. |
+--
    ↪ -----+------+-------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

14.11.2.16.3 MySQL 兼容性

TiDB 中的 ALTER TABLE 语法主要存在以下限制：

- 使用 ALTER TABLE 语句修改一个表的多个模式对象（如列、索引）时：
  - 不允许在多个更改中指定同一个模式对象。
  - TiDB 根据执行前的表结构检查合法性。例如 ALTER TABLE t ADD COLUMN c1 INT, ADD COLUMN c2 ↪ INT AFTER c1; 会报错，因为表结构中不存在名字为 c1 的列。
  - TiDB 的执行顺序是从左往右逐个执行更改，该行为在个别场景下和 MySQL 不兼容。
- 不支持主键列上Reorg-Data 类型的变更。
- 不支持分区表上的列类型变更。
- 不支持生成列上的列类型变更。
- 不支持部分数据类型（例如，部分时间类型、Bit、Set、Enum、JSON 等）的变更，因为 TiDB 中的 CAST 函数与 MySQL 的行为存在兼容性问题。
- 不支持空间数据类型。
- ALTER TABLE t CACHE | NOCACHE 不是 MySQL 标准语法，而是 TiDB 扩展功能，参见缓存表。

其它限制可参考：TiDB 中 DDL 语句与 MySQL 的兼容性情况。

14.11.2.16.4 另请参阅

- 与 MySQL 兼容性对比

- ALTER TABLE ··· COMPACT
- ADD COLUMN
- DROP COLUMN
- ADD INDEX
- DROP INDEX
- RENAME INDEX
- ALTER INDEX
- CREATE TABLE
- DROP TABLE
- SHOW CREATE TABLE

14.11.2.17 ALTER TABLE ··· COMPACT

TiDB 存储节点在后台会自动发起数据整理（Compaction）。数据整理时，表中的物理数据会被重写，如清理已删除的数据、合并多版本数据等，从而可以获得更高的访问性能，并减少磁盘空间占用。使用 `ALTER TABLE ... COMPACT` 语句可以立即对指定的表进行数据整理，而无需等待后台触发。

该语句执行时不会阻塞现有 SQL 语句的执行或 TiDB 功能的使用，包括事务、DDL、GC 等，也不会改变通过 SQL 语句访问获得的数据内容。该语句执行时会消耗一定量的 IO 及 CPU 资源，请注意选择合适的时机执行，如资源空闲时段，避免对业务造成负面影响。

该语句会等待表中所有副本都数据整理完毕后才结束运行并返回。在执行过程中，你可以通过`KILL` 语句安全地中断本张表的数据整理过程。中断不会破坏数据一致性或丢失数据，也不会影响后续重新发起或自动触发后台数据整理。

目前该语句仅支持对 TiFlash 进行数据整理，不支持对 TiKV 进行数据整理。

14.11.2.17.1 语法图

```
AlterTableCompactStmt ::=
    'ALTER' 'TABLE' TableName 'COMPACT' ( 'PARTITION' PartitionNameList )? ( 'TIFLASH' 'REPLICA'
        ↪ )?
```

自 v6.2.0 起，语法中 `TIFLASH REPLICA` 部分可以被省略。省略后语句含义不变，同样只对 TiFlash 列存有效。

14.11.2.17.2 示例

对表中 TiFlash 副本进行数据整理

假设目前有一张 `employees` 表具有 4 个分区，且具有 2 个 TiFlash 副本：

```
CREATE TABLE employees (
    id INT NOT NULL,
    hired DATE NOT NULL DEFAULT '1970-01-01',
    store_id INT
)
PARTITION BY LIST (store_id) (
    PARTITION pNorth VALUES IN (1, 2, 3, 4, 5),
    PARTITION pEast VALUES IN (6, 7, 8, 9, 10),
```

```
    PARTITION pWest VALUES IN (11, 12, 13, 14, 15),
    PARTITION pCentral VALUES IN (16, 17, 18, 19, 20)
);

ALTER TABLE employees SET TIFLASH REPLICA 2;
```

执行以下语句可对 employees 表上所有分区的 2 个 TiFlash 副本立即进行数据整理：

```
ALTER TABLE employees COMPACT TIFLASH REPLICA;
```

对分区表中指定分区的 TiFlash 副本进行数据整理

假设目前有一张 employees 表具有 4 个分区，且具有 2 个 TiFlash 副本：

```
CREATE TABLE employees (
    id INT NOT NULL,
    hired DATE NOT NULL DEFAULT '1970-01-01',
    store_id INT
)
PARTITION BY LIST (store_id) (
    PARTITION pNorth VALUES IN (1, 2, 3, 4, 5),
    PARTITION pEast VALUES IN (6, 7, 8, 9, 10),
    PARTITION pWest VALUES IN (11, 12, 13, 14, 15),
    PARTITION pCentral VALUES IN (16, 17, 18, 19, 20)
);

ALTER TABLE employees SET TIFLASH REPLICA 2;
```

执行以下语句可对 employees 表上 pNorth、pEast 这两个分区的 2 个 TiFlash 副本立即进行数据整理：

```
ALTER TABLE employees COMPACT PARTITION pNorth, pEast TIFLASH REPLICA;
```

14.11.2.17.3 并发度

ALTER TABLE ... COMPACT 语句会同时对表中所有副本发起数据整理操作。

为了避免数据整理操作对在线业务造成显著影响，在同一时间，每个 TiFlash 实例默认只会处理一张表的数据整理操作（后台自动触发的数据整理除外）。这意味着，若你同时对多张表执行 ALTER TABLE ... COMPACT 语句，则它们在同一个 TiFlash 实例上会排队依次执行，而非同时执行。

你可以修改 TiFlash 配置文件参数manual_compact_pool_size，以更大资源占用为代价、获得更大的表级别并发度。例如，参数指定为 2 时，可以同时对 2 张表进行数据整理。

14.11.2.17.4 观测数据整理进度

你可以通过 INFORMATION_SCHEMA.TIFLASH_TABLES 表中 TOTAL_DELTA_ROWS 列来观测 TiFlash 存储引擎上数据整理的进度，或判断是否有必要对某张表发起数据整理。TOTAL_DELTA_ROWS 的值越大，说明还能被整理的数据越多；若 TOTAL_DELTA_ROWS 为 0，说明表中所有数据都处于最佳状态，无需整理。

示例：查询普通表的数据整理状态

```
USE test;

CREATE TABLE foo(id INT);

ALTER TABLE foo SET TIFLASH REPLICA 1;

SELECT TOTAL_DELTA_ROWS, TOTAL_STABLE_ROWS FROM INFORMATION_SCHEMA.TIFLASH_TABLES
    WHERE IS_TOMBSTONE = 0 AND
    `TIDB_DATABASE` = "test" AND `TIDB_TABLE` = "foo";
+------------------+------------------+
| TOTAL_DELTA_ROWS | TOTAL_STABLE_ROWS |
+------------------+------------------+
|                0 |                0 |
+------------------+------------------+

INSERT INTO foo VALUES (1), (3), (7);

SELECT TOTAL_DELTA_ROWS, TOTAL_STABLE_ROWS FROM INFORMATION_SCHEMA.TIFLASH_TABLES
    WHERE IS_TOMBSTONE = 0 AND
    `TIDB_DATABASE` = "test" AND `TIDB_TABLE` = "foo";
+------------------+------------------+
| TOTAL_DELTA_ROWS | TOTAL_STABLE_ROWS |
+------------------+------------------+
|                3 |                0 |
+------------------+------------------+
-- 新写入的数据可被整理

ALTER TABLE foo COMPACT TIFLASH REPLICA;

SELECT TOTAL_DELTA_ROWS, TOTAL_STABLE_ROWS FROM INFORMATION_SCHEMA.TIFLASH_TABLES
    WHERE IS_TOMBSTONE = 0 AND
    `TIDB_DATABASE` = "test" AND `TIDB_TABLE` = "foo";
+------------------+------------------+
| TOTAL_DELTA_ROWS | TOTAL_STABLE_ROWS |
+------------------+------------------+
|                0 |                3 |
+------------------+------------------+
-- 所有数据都处于最佳整理状态
```

示例：查询分区表的数据整理状态

```
USE test;

CREATE TABLE employees
    (id INT NOT NULL, store_id INT)
```

```
    PARTITION BY LIST (store_id) (
        PARTITION pNorth VALUES IN (1, 2, 3, 4, 5),
        PARTITION pEast VALUES IN (6, 7, 8, 9, 10),
        PARTITION pWest VALUES IN (11, 12, 13, 14, 15),
        PARTITION pCentral VALUES IN (16, 17, 18, 19, 20)
    );

ALTER TABLE employees SET TIFLASH REPLICA 1;

INSERT INTO employees VALUES (1, 1), (6, 6), (10, 10);

SELECT PARTITION_NAME, TOTAL_DELTA_ROWS, TOTAL_STABLE_ROWS
    FROM INFORMATION_SCHEMA.TIFLASH_TABLES t, INFORMATION_SCHEMA.PARTITIONS p
    WHERE t.IS_TOMBSTONE = 0 AND t.TABLE_ID = p.TIDB_PARTITION_ID AND
    p.TABLE_SCHEMA = "test" AND p.TABLE_NAME = "employees";
+----------------+------------------+--------------------+
| PARTITION_NAME | TOTAL_DELTA_ROWS | TOTAL_STABLE_ROWS |
+----------------+------------------+--------------------+
| pNorth         |                1 |                  0 |
| pEast          |                2 |                  0 |
| pWest          |                0 |                  0 |
| pCentral       |                0 |                  0 |
+----------------+------------------+--------------------+
-- 一些分区上有数据可被整理

ALTER TABLE employees COMPACT TIFLASH REPLICA;

SELECT PARTITION_NAME, TOTAL_DELTA_ROWS, TOTAL_STABLE_ROWS
    FROM INFORMATION_SCHEMA.TIFLASH_TABLES t, INFORMATION_SCHEMA.PARTITIONS p
    WHERE t.IS_TOMBSTONE = 0 AND t.TABLE_ID = p.TIDB_PARTITION_ID AND
    p.TABLE_SCHEMA = "test" AND p.TABLE_NAME = "employees";
+----------------+------------------+--------------------+
| PARTITION_NAME | TOTAL_DELTA_ROWS | TOTAL_STABLE_ROWS |
+----------------+------------------+--------------------+
| pNorth         |                0 |                  1 |
| pEast          |                0 |                  2 |
| pWest          |                0 |                  0 |
| pCentral       |                0 |                  0 |
+----------------+------------------+--------------------+
-- 所有分区上的数据都处于最佳整理状态
```

注意：

- 若数据整理的过程中发生了数据更新，你可能观察到数据整理完毕后 TOTAL_DELTA_ROWS 仍为非零值。这是正常现象，表明这些更新部分没有被整理到。若你想对这部分更新数据也进行整理，可再次执行 ALTER TABLE ... COMPACT 语句。
- TOTAL_DELTA_ROWS 的单位是数据版本数，而非数据行数。例如，插入一行记录再删除该行后，TOTAL_DELTA_ROWS 会增加 2。

14.11.2.17.5 兼容性

MySQL 兼容性

ALTER TABLE ... COMPACT 语法是 TiDB 引入的对标准 SQL 语法的扩展。尽管没有对应的 MySQL 语法，但你仍然可通过 MySQL 各版本客户端，或各个遵循 MySQL 协议的数据库驱动执行该语句。

TiDB Binlog 及 TiCDC 兼容性

ALTER TABLE ... COMPACT 语句不会导致逻辑数据变化，因而不会被 TiDB Binlog 及 TiCDC 同步到下游。

14.11.2.17.6 另请参阅

- ALTER TABLE
- KILL

14.11.2.18 ALTER USER

ALTER USER 语句用于更改 TiDB 权限系统内的已有用户。和 MySQL 一样，在 TiDB 权限系统中，用户是用户名和用户名所连接主机的组合。因此，可创建一个用户 'newuser2'@'192.168.1.1'，使其只能通过 IP 地址 192.168.1.1 进行连接。相同的用户名从不同主机登录时可能会拥有不同的权限。

14.11.2.18.1 语法图

```
AlterUserStmt ::=
    'ALTER' 'USER' IfExists (UserSpecList RequireClauseOpt ConnectionOptions PasswordOption
        ↪ LockOption AttributeOption | 'USER' '(' ')' 'IDENTIFIED' 'BY' AuthString)
        ↪ ResourceGroupNameOption

UserSpecList ::=
    UserSpec ( ',' UserSpec )*

UserSpec ::=
    Username AuthOption

Username ::=
    StringName ('@' StringName | singleAtIdentifier)? | 'CURRENT_USER' OptionalBraces

AuthOption ::=
```

```
    ( 'IDENTIFIED' ( 'BY' ( AuthString | 'PASSWORD' HashString ) | 'WITH' StringName ( 'BY'
        ↪ AuthString | 'AS' HashString )? ) )?

PasswordOption ::= ( 'PASSWORD' 'EXPIRE' ( 'DEFAULT' | 'NEVER' | 'INTERVAL' N 'DAY' )? | '
    ↪ PASSWORD' 'HISTORY' ( 'DEFAULT' | N ) | 'PASSWORD' 'REUSE' 'INTERVAL' ( 'DEFAULT' | N '
    ↪ DAY' ) | 'FAILED_LOGIN_ATTEMPTS' N | 'PASSWORD_LOCK_TIME' ( N | 'UNBOUNDED' ) )*

LockOption ::= ( 'ACCOUNT' 'LOCK' | 'ACCOUNT' 'UNLOCK' )?

AttributeOption ::= ( 'COMMENT' CommentString | 'ATTRIBUTE' AttributeString )?

ResourceGroupNameOption::= ( 'RESOURCE' 'GROUP' Identifier)?
```

14.11.2.18.2 示例

```
CREATE USER 'newuser' IDENTIFIED BY 'newuserpassword';
```

```
Query OK, 1 row affected (0.01 sec)
```

```
SHOW CREATE USER 'newuser';
```

```
+-----------------------------------------------------------------------------------------------------------------------
    ↪
| CREATE USER for newuser@%
    ↪
    ↪ |
+-----------------------------------------------------------------------------------------------------------------------
    ↪
| CREATE USER 'newuser'@'%' IDENTIFIED WITH 'mysql_native_password' AS '*5806
    ↪ E04BBEE79E1899964C6A04D68BCA69B1A879' REQUIRE NONE PASSWORD EXPIRE DEFAULT ACCOUNT UNLOCK
    ↪  |
+-----------------------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

修改用户基本信息

修改用户 newuser 的密码：

```
ALTER USER 'newuser' IDENTIFIED BY 'newnewpassword';
```

```
Query OK, 0 rows affected (0.02 sec)
```

```
SHOW CREATE USER 'newuser';
```

```
+-----------------------------------------------------------------------------------------------------------------------
    ↪
| CREATE USER for newuser@%
    ↪
    ↪ |
+-----------------------------------------------------------------------------------------------------------------------
    ↪
| CREATE USER 'newuser'@'%' IDENTIFIED WITH 'mysql_native_password' AS '*
    ↪ FB8A1EA1353E8775CA836233E367FBDFCB37BE73' REQUIRE NONE PASSWORD EXPIRE DEFAULT ACCOUNT
    ↪ UNLOCK |
+-----------------------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

锁定用户 `newuser`：

```
ALTER USER 'newuser' ACCOUNT LOCK;
```

```
Query OK, 0 rows affected (0.02 sec)
```

修改 `newuser` 的属性：

```
ALTER USER 'newuser' ATTRIBUTE '{"newAttr": "value", "deprecatedAttr": null}';
SELECT * FROM information_schema.user_attributes;
```

```
+-----------+------+--------------------------+
| USER      | HOST | ATTRIBUTE                |
+-----------+------+--------------------------+
| newuser   | %    | {"newAttr": "value"}     |
+-----------+------+--------------------------+
1 rows in set (0.00 sec)
```

通过 `ALTER USER ... COMMENT` 修改用户 `newuser` 的注释：

```
ALTER USER 'newuser' COMMENT 'Here is the comment';
SELECT * FROM information_schema.user_attributes;
```

```
+-----------+------+-----------------------------------------------------------+
| USER      | HOST | ATTRIBUTE                                                 |
+-----------+------+-----------------------------------------------------------+
| newuser   | %    | {"comment": "Here is the comment", "newAttr": "value"} |
+-----------+------+-----------------------------------------------------------+
1 rows in set (0.00 sec)
```

通过 `ALTER USER ... ATTRIBUTE` 删除用户 `newuser` 的注释：

```
ALTER USER 'newuser' ATTRIBUTE '{"comment": null}';
SELECT * FROM information_schema.user_attributes;
```

```
+-----------+------+---------------------------+
| USER      | HOST | ATTRIBUTE                 |
+-----------+------+---------------------------+
| newuser   | %    | {"newAttr": "value"}      |
+-----------+------+---------------------------+
1 rows in set (0.00 sec)
```

通过 `ALTER USER ... PASSWORD EXPIRE NEVER` 修改用户 `newuser` 的自动密码过期策略为永不过期：

```
ALTER USER 'newuser' PASSWORD EXPIRE NEVER;
```

```
Query OK, 0 rows affected (0.02 sec)
```

通过 `ALTER USER ... PASSWORD REUSE INTERVAL ... DAY` 修改用户 `newuser` 的密码重用策略为不允许重复使用最近 `90` 天内使用过的密码：

```
ALTER USER 'newuser' PASSWORD REUSE INTERVAL 90 DAY;
```

```
Query OK, 0 rows affected (0.02 sec)
```

修改用户绑定的资源组

通过 `ALTER USER ... RESOURCE GROUP` 修改用户 `newuser` 的资源组到 `rg1`：

```
ALTER USER 'newuser' RESOURCE GROUP rg1;
```

```
Query OK, 0 rows affected (0.02 sec)
```

查看当前用户绑定的资源组：

```
SELECT USER, JSON_EXTRACT(User_attributes, "$.resource_group") FROM mysql.user WHERE user = "
    ↪ newuser";
```

```
+---------+-------------------------------------------------+
| USER    | JSON_EXTRACT(User_attributes, "$.resource_group") |
+---------+-------------------------------------------------+
| newuser | "rg1"                                           |
+---------+-------------------------------------------------+
1 row in set (0.02 sec)
```

取消用户绑定的资源组，即将用户绑定的资源组设置为空。取消后，用户会绑定到 `default` 资源组。

```
ALTER USER 'newuser' RESOURCE GROUP ``;
SELECT USER, JSON_EXTRACT(User_attributes, "$.resource_group") FROM mysql.user WHERE user = "
    ↪ newuser";
```

```
+---------+-----------------------------------------------------+
| USER    | JSON_EXTRACT(User_attributes, "$.resource_group") |
+---------+-----------------------------------------------------+
| newuser | ""                                                  |
+---------+-----------------------------------------------------+
1 row in set (0.02 sec)
```

14.11.2.18.3 另请参阅

- Security Compatibility with MySQL
- CREATE USER
- DROP USER
- SHOW CREATE USER

14.11.2.19 ANALYZE

ANALYZE 语句用于更新 TiDB 在表和索引上留下的统计信息。执行大批量更新或导入记录后，或查询执行计划不是最佳时，建议运行 ANALYZE。

当 TiDB 逐渐发现这些统计数据与预估不一致时，也会自动更新其统计数据。

目前 TiDB 收集统计信息分为全量收集和增量收集两种方式，分别通过 ANALYZE TABLE 和 ANALYZE INCREMENTAL TABLE 语句来实现。关于这两种语句的详细使用方式，可参考统计信息简介。

14.11.2.19.1 语法图

```
AnalyzeTableStmt ::=
    'ANALYZE' ( 'TABLE' ( TableNameList ( 'ALL COLUMNS' | 'PREDICATE COLUMNS' ) | TableName ( '
        ↪ INDEX' IndexNameList? | AnalyzeColumnOption | 'PARTITION' PartitionNameList ( 'INDEX'
        ↪  IndexNameList? | AnalyzeColumnOption )? )? ) | 'INCREMENTAL' 'TABLE' TableName ( '
        ↪ PARTITION' PartitionNameList )? 'INDEX' IndexNameList? ) AnalyzeOptionListOpt

AnalyzeOptionListOpt ::=
( WITH AnalyzeOptionList )?

AnalyzeOptionList ::=
AnalyzeOption ( ',' AnalyzeOption )*

AnalyzeOption ::=
( NUM ( 'BUCKETS' | 'TOPN' | ( 'CMSKETCH' ( 'DEPTH' | 'WIDTH' ) ) | 'SAMPLES' ) ) | ( FLOATNUM '
    ↪ SAMPLERATE' )

AnalyzeColumnOption ::=
( 'ALL COLUMNS' | 'PREDICATE COLUMNS' | 'COLUMNS' ColumnNameList )
```

```
TableNameList ::=
    TableName (',' TableName)*

TableName ::=
    Identifier ( '.' Identifier )?

ColumnNameList ::=
    Identifier ( ',' Identifier )*

IndexNameList ::=
    Identifier ( ',' Identifier )*

PartitionNameList ::=
    Identifier ( ',' Identifier )*
```

14.11.2.19.2 示例

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, c1 INT NOT NULL);
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
INSERT INTO t1 (c1) VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 5 rows affected (0.03 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
ALTER TABLE t1 ADD INDEX (c1);
```

```
Query OK, 0 rows affected (0.30 sec)
```

```
EXPLAIN SELECT * FROM t1 WHERE c1 = 3;
```

```
+-----------------------+---------+-----------+-----------------------+-----------------------------------
    ↪
| id                    | estRows | task      | access object         | operator info
    ↪                               |
+-----------------------+---------+-----------+-----------------------+-----------------------------------
    ↪
| IndexReader_6         | 10.00   | root      |                       | index:IndexRangeScan_5
    ↪                       |
| └─IndexRangeScan_5    | 10.00   | cop[tikv] | table:t1, index:c1(c1) | range:[3,3], keep
    ↪ order:false, stats:pseudo |
+-----------------------+---------+-----------+-----------------------+-----------------------------------
    ↪
2 rows in set (0.00 sec)
```

```
analyze table t1;
```

```
Query OK, 0 rows affected (0.13 sec)
```

```
EXPLAIN SELECT * FROM t1 WHERE c1 = 3;
```

```
+-------------------------+---------+-----------+-----------------------+------------------------------+
    ↪
| id                      | estRows | task      | access object         | operator info
    ↪                 |
+-------------------------+---------+-----------+-----------------------+------------------------------+
    ↪
| IndexReader_6           | 1.00    | root      |                       | index:IndexRangeScan_5
    ↪        |
| └─IndexRangeScan_5      | 1.00    | cop[tikv] | table:t1, index:c1(c1) | range:[3,3], keep
    ↪ order:false |
+-------------------------+---------+-----------+-----------------------+------------------------------+
    ↪
2 rows in set (0.00 sec)
```

14.11.2.19.3 MySQL 兼容性

- ANALYZE TABLE 在语法上与 MySQL 类似。但 ANALYZE TABLE 在 TiDB 上的执行时间可能长得多，因为它的内部运行方式不同。
- 在 MySQL 上不支持 ANALYZE INCREMENTAL TABLE 语句，它的使用可参考增量收集文档。

TiDB 与 MySQL 在以下方面存在区别：所收集的统计信息，以及查询执行过程中统计信息是如何被使用的。虽然 TiDB 中的 ANALYZE 语句在语法上与 MySQL 类似，但存在以下差异：

- 执行 ANALYZE TABLE 时，TiDB 可能不包含最近提交的更改。若对行进行了批量更改，在执行 ANALYZE TABLE 之前，你可能需要先执行 sleep(1)，这样统计信息更新才能反映这些更改。参见 #16570。
- ANALYZE TABLE 在 TiDB 中的执行时间比在 MySQL 中的执行时间要长得多。但你可以通过执行 SET GLOBAL ↪ tidb_enable_fast_analyze=1 来启用快速分析，这样能部分抵消这种执行上的性能差异。快速分析利用了采样，会导致统计信息的准确性降低。因此快速分析仍是一项实验特性。

14.11.2.19.4 另请参阅

- EXPLAIN
- EXPLAIN ANALYZE

14.11.2.20 BACKUP

BACKUP 语句用于对 TiDB 集群执行分布式备份操作。

BACKUP 语句使用的引擎与BR 相同，但备份过程是由 TiDB 本身驱动，而非单独的 BR 工具。BR 工具的优势和警告也适用于 BACKUP 语句。

执行 BACKUP 需要 BACKUP_ADMIN 或 SUPER 权限。此外，执行备份的 TiDB 节点和集群中的所有 TiKV 节点都必须有对目标存储的读或写权限。

BACKUP 语句开始执行后将会被阻塞，直到整个备份任务完成、失败或取消。因此，执行 BACKUP 时需要准备一个持久的连接。如需取消任务，可执行KILL TIDB QUERY 语句。

一次只能执行一个 BACKUP 和RESTORE 任务。如果 TiDB server 上已经在执行一个 BACKUP 或 RESTORE 语句，新的 BACKUP 将等待前面所有的任务完成后再执行。

14.11.2.20.1 语法图

```
BackupStmt ::=
    "BACKUP" BRIETables "TO" stringLit BackupOption*

BRIETables ::=
    "DATABASE" ( '*' | DBName (',' DBName)* )
|   "TABLE" TableNameList

BackupOption ::=
    "RATE_LIMIT" '='? LengthNum "MB" '/' "SECOND"
|   "CONCURRENCY" '='? LengthNum
|   "CHECKSUM" '='? Boolean
|   "SEND_CREDENTIALS_TO_TIKV" '='? Boolean
|   "LAST_BACKUP" '='? BackupTSO
|   "SNAPSHOT" '='? ( BackupTSO | LengthNum TimestampUnit "AGO" )

Boolean ::=
    NUM | "TRUE" | "FALSE"

BackupTSO ::=
    LengthNum | stringLit
```

14.11.2.20.2 示例

备份数据库

```
BACKUP DATABASE `test` TO 'local:///mnt/backup/2020/04/';
```

```
+--
    ↪ ---------------------------+-----------+-----------------+---------------------+---------------------+
    ↪
```

```
| Destination                  | Size      | BackupTS         | Queue Time          | Execution
    ↪ Time      |
+--
    ↪ --------------------------+-----------+------------------+---------------------+---------------------+
    ↪
| local:///mnt/backup/2020/04/ | 248665063 | 416099531454472 | 2020-04-12 23:09:48 | 2020-04-12
    ↪ 23:09:48 |
+--
    ↪ --------------------------+-----------+------------------+---------------------+---------------------+
    ↪
1 row in set (58.453 sec)
```

上述示例中，test 数据库被备份到本地，数据以 SST 文件的形式存储在分布于所有 TiDB 和 TiKV 节点的 /mnt/backup/2020/04/ 目录中。

输出结果的第一行描述如下：

列名	描述
Destination	目标存储的 URL
Size	备份文件的总大小，单位为字节
BackupTS	创建备份时的快照 TSO（用于增量备份）
Queue Time	BACKUP 任务开始排队的时间戳（当前时区）
Execution Time	BACKUP 任务开始执行的时间戳（当前时区）

备份表

```
BACKUP TABLE `test`.`sbtest01` TO 'local:///mnt/backup/sbtest01/';
```

```
BACKUP TABLE sbtest02, sbtest03, sbtest04 TO 'local:///mnt/backup/sbtest/';
```

备份集群

```
BACKUP DATABASE * TO 'local:///mnt/backup/full/';
```

注意，备份中不包含系统表（mysql.*、INFORMATION_SCHEMA.*、PERFORMANCE_SCHEMA.* 等）。

外部存储

BR 支持备份数据到 Amazon S3 或 Google Cloud Storage (GCS)：

```
BACKUP DATABASE `test` TO 's3://example-bucket-2020/backup-05/?access-key={YOUR_ACCESS_KEY}&
    ↪ secret-access-key={YOUR_SECRET_KEY}';
```

有关详细的 URL 语法，见外部存储 URI 格式。

当运行在云环境中时，不能分发凭证，可设置 SEND_CREDENTIALS_TO_TIKV 选项为 FALSE：

```
BACKUP DATABASE `test` TO 's3://example-bucket-2020/backup-05/'
    SEND_CREDENTIALS_TO_TIKV = FALSE;
```

性能调优

如果你需要减少网络带宽占用，可以通过 RATE_LIMIT 来限制每个 TiKV 节点的平均上传速度。

默认情况下，每个 TiKV 节点上运行 4 个备份线程。可以通过 CONCURRENCY 选项来调整这个值。

在备份完成之前，BACKUP 将对集群上的数据进行校验，以验证数据的正确性。如果你确信无需进行校验，可以通过 CHECKSUM 选项禁用这一步骤。

```
BACKUP DATABASE `test` TO 's3://example-bucket-2020/backup-06/'
    RATE_LIMIT = 120 MB/SECOND
    CONCURRENCY = 8
    CHECKSUM = FALSE;
```

快照

可以指定一个时间戳、TSO 或相对时间，来备份历史数据。

```
-- 相对时间
BACKUP DATABASE `test` TO 'local:///mnt/backup/hist01'
    SNAPSHOT = 36 HOUR AGO;
-- 时间戳（当前时区）
BACKUP DATABASE `test` TO 'local:///mnt/backup/hist02'
    SNAPSHOT = '2020-04-01 12:00:00';
-- TSO
BACKUP DATABASE `test` TO 'local:///mnt/backup/hist03'
    SNAPSHOT = 415685305958400;
```

对于相对时间，支持以下时间单位：

- MICROSECOND（微秒）
- SECOND（秒）
- MINUTE（分钟）
- HOUR（小时）
- DAY（天）
- WEEK（周）

注意，相对时间的单位遵循 SQL 标准，永远使用单数。

增量备份

提供 LAST_BACKUP 选项，只备份从上一次备份到当前快照之间的增量数据。

```
-- 时间戳（当前时区）
BACKUP DATABASE `test` TO 'local:///mnt/backup/hist02'
    LAST_BACKUP = '2020-04-01 12:00:00';

-- TSO
BACKUP DATABASE `test` TO 'local:///mnt/backup/hist03'
    LAST_BACKUP = 415685305958400;
```

14.11.2.20.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.20.4 另请参阅

- RESTORE
- SHOW BACKUPS

14.11.2.21 BATCH

BATCH 语句将一个 DML 语句拆成多个语句在内部执行，因此不保证事务的原子性和隔离性，是一个“非事务”语句。

目前 BATCH 语句支持 INSERT、 REPLACE、 UPDATE、 DELETE。

BATCH 语句在某一列将 DML 语句涉及的范围划分为多个区间，在每个区间执行一条 SQL。

详细的说明和使用限制见非事务语句。

在涉及多表 join 时，BATCH 语法中指定拆分列时需要指明完整的路径以避免歧义，如：

```
BATCH ON test.t2.id LIMIT 1 INSERT INTO t SELECT t2.id, t2.v, t3.v FROM t2 JOIN t3 ON t2.k = t3.k
    ↪ ;
```

上面这条语句的拆分列用 `test.t2.id` 指明，不具有歧义。如果写成如下 `id` 的形式，则会报错：

```
BATCH ON id LIMIT 1 INSERT INTO t SELECT t2.id, t2.v, t3.v FROM t2 JOIN t3 ON t2.k = t3.k;

Non-transactional DML, shard column must be fully specified
```

14.11.2.21.1 语法图

```
NonTransactionalDMLStmt ::=
    'BATCH' ( 'ON' ColumnName )? 'LIMIT' NUM DryRunOptions? ShardableStmt

DryRunOptions ::=
    'DRY' 'RUN' 'QUERY'?

ShardableStmt ::=
    DeleteFromStmt
|   UpdateStmt
|   InsertIntoStmt
|   ReplaceIntoStmt
```

14.11.2.21.2 MySQL 兼容性

BATCH 语句是 TiDB 独有的语句，与 MySQL 不兼容。

14.11.2.21.3 另请参阅

- 非事务语句

14.11.2.22 BEGIN

BEGIN 语句用于在 TiDB 内启动一个新事务，类似于 START TRANSACTION 和 SET autocommit=0 语句。

在没有 BEGIN 语句的情况下，每个语句默认在各自的事务中自动提交，从而确保 MySQL 兼容性。

14.11.2.22.1 语法图

```
BeginTransactionStmt ::=
    'BEGIN' ( 'PESSIMISTIC' | 'OPTIMISTIC' )?
|   'START' 'TRANSACTION' ( 'READ' ( 'WRITE' | 'ONLY' ( 'WITH' 'TIMESTAMP' 'BOUND' TimestampBound
    ↪  )? ) | 'WITH' 'CONSISTENT' 'SNAPSHOT' )?
```

14.11.2.22.2 示例

```
CREATE TABLE t1 (a int NOT NULL PRIMARY KEY);
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
BEGIN;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
INSERT INTO t1 VALUES (1);
```

```
Query OK, 1 row affected (0.00 sec)
```

```
COMMIT;
```

```
Query OK, 0 rows affected (0.01 sec)
```

14.11.2.22.3 MySQL 兼容性

TiDB 支持 BEGIN PESSIMISTIC 或 BEGIN OPTIMISTIC 的语法扩展，用户可以为某一个事务覆盖默认的事务模型。

14.11.2.22.4 另请参阅

- COMMIT
- ROLLBACK
- START TRANSACTION
- TiDB 乐观事务模型
- TiDB 悲观事务模式

14.11.2.23 CALIBRATE RESOURCE

`CALIBRATE RESOURCE` 语句用于预估并输出当前集群的`Request Unit (RU)` 的容量。

14.11.2.23.1 语法图

```
CalibrateResourceStmt ::= 'CALIBRATE' 'RESOURCE'
```

14.11.2.23.2 示例

```
CALIBRATE RESOURCE;
+-------+
| QUOTA |
+-------+
| 68569 |
+-------+
1 row in set (0.03 sec)
```

> **注意：**
>
> 集群 RU 的容量会随集群的拓扑结构和各个组件软硬件配置的变化而变化，每个集群实际能消耗的 RU 还与实际的负载相关。此预估值仅供参考，可能会与实际的最大值存在偏差。

14.11.2.24 CHANGE COLUMN

`ALTER TABLE.. CHANGE COLUMN` 语句用于在已有表上更改列，包括对列进行重命名，和将数据改为兼容类型。

从 v5.1.0 版本起，TiDB 开始支持 Reorg 数据的类型变更，包括但不限于：

- 从 varchar 转换为 bigint
- decimal 精度修改
- 从 varchar(10) 到 varchar(5) 的长度压缩

14.11.2.24.1 语法图

```
AlterTableStmt
         ::= 'ALTER' 'IGNORE'? 'TABLE' TableName ChangeColumnSpec ( ',' ChangeColumnSpec )*

ChangeColumnSpec
         ::= 'CHANGE' ColumnKeywordOpt 'IF EXISTS' ColumnName ColumnName ColumnType ColumnOption*
             ↪  ( 'FIRST' | 'AFTER' ColumnName )?

ColumnType
         ::= NumericType
```

```
            | StringType
            | DateAndTimeType
            | 'SERIAL'

ColumnOption
         ::= 'NOT'? 'NULL'
           | 'AUTO_INCREMENT'
           | 'PRIMARY'? 'KEY' ( 'CLUSTERED' | 'NONCLUSTERED' )?
           | 'UNIQUE' 'KEY'?
           | 'DEFAULT' ( NowSymOptionFraction | SignedLiteral | NextValueForSequence )
           | 'SERIAL' 'DEFAULT' 'VALUE'
           | 'ON' 'UPDATE' NowSymOptionFraction
           | 'COMMENT' stringLit
           | ( 'CONSTRAINT' Identifier? )? 'CHECK' '(' Expression ')' ( 'NOT'? ( 'ENFORCED' | '
               ↪ NULL' ) )?
           | 'GENERATED' 'ALWAYS' 'AS' '(' Expression ')' ( 'VIRTUAL' | 'STORED' )?
           | 'REFERENCES' TableName ( '(' IndexPartSpecificationList ')' )? Match?
               ↪ OnDeleteUpdateOpt
           | 'COLLATE' CollationName
           | 'COLUMN_FORMAT' ColumnFormat
           | 'STORAGE' StorageMedia
           | 'AUTO_RANDOM' ( '(' LengthNum ')' )?

ColumnName ::=
    Identifier ( '.' Identifier ( '.' Identifier )? )?
```

14.11.2.24.2 示例

```
CREATE TABLE t1 (id int not null primary key AUTO_INCREMENT, col1 INT);
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
INSERT INTO t1 (col1) VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 5 rows affected (0.02 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
ALTER TABLE t1 CHANGE col1 col2 INT;
```

```
Query OK, 0 rows affected (0.09 sec)
```

```
ALTER TABLE t1 CHANGE col2 col3 BIGINT, ALGORITHM=INSTANT;
```

```
Query OK, 0 rows affected (0.08 sec)
```

```
ALTER TABLE t1 CHANGE col3 col4 BIGINT, CHANGE id id2 INT NOT NULL;
```

```
ERROR 1105 (HY000): can't run multi schema change
```

```
CREATE TABLE t (a int primary key);
ALTER TABLE t CHANGE COLUMN a a VARCHAR(10);
```

```
ERROR 8200 (HY000): Unsupported modify column: column has primary key flag
```

```
CREATE TABLE t (c1 INT, c2 INT, c3 INT) partition by range columns(c1) ( partition p0 values less
    ↪  than (10), partition p1 values less than (maxvalue));
ALTER TABLE t CHANGE COLUMN c1 c1 DATETIME;
```

```
ERROR 8200 (HY000): Unsupported modify column: table is partition table
```

```
CREATE TABLE t (a INT, b INT as (a+1));
ALTER TABLE t CHANGE COLUMN b b VARCHAR(10);
```

```
ERROR 8200 (HY000): Unsupported modify column: column is generated
```

```
CREATE TABLE t (a DECIMAL(13, 7));
ALTER TABLE t CHANGE COLUMN a a DATETIME;
```

```
ERROR 8200 (HY000): Unsupported modify column: change from original type decimal(13,7) to
    ↪ datetime is currently unsupported yet
```

14.11.2.24.3 MySQL 兼容性

- 不支持主键列上Reorg-Data 类型的变更。
- 不支持分区表上的列类型变更。
- 不支持生成列上的列类型变更。
- 不支持部分数据类型（例如，部分时间类型、Bit、Set、Enum、JSON 等）的变更，因为 TiDB 中 CAST 函数与 MySQL 的行为存在兼容性问题。

14.11.2.24.4 另请参阅

- CREATE TABLE
- SHOW CREATE TABLE
- ADD COLUMN
- DROP COLUMN
- MODIFY COLUMN

14.11.2.25 CHANGE DRAINER

CHANGE DRAINER 语句用于修改集群中 Drainer 的状态信息。

> **注意：**
>
> Drainer 在正常运行时会自动上报状态到 PD，仅在 Drainer 处于异常情况导致实际状态与 PD 中保存的状态信息不一致时，使用该语句修改 PD 中存储的 Drainer 状态信息。

14.11.2.25.1 示例

```
SHOW DRAINER STATUS;
```

```
+----------|----------------|--------|--------------------|---------------------|
|  NodeID  |     Address    | State  |   Max_Commit_Ts    |    Update_Time      |
+----------|----------------|--------|--------------------|---------------------|
| drainer1 | 127.0.0.3:8249 | Online | 408553768673342532 | 2019-04-30 00:00:03 |
+----------|----------------|--------|--------------------|---------------------|
| drainer2 | 127.0.0.4:8249 | Online | 408553768673345531 | 2019-05-01 00:00:04 |
+----------|----------------|--------|--------------------|---------------------|
2 rows in set (0.00 sec)
```

可以看出 drainer1 已经超过一天没有更新状态，该 Drainer 处于异常状态，但是 State 仍然为 Online，使用 CHANGE DRAINER 将该 Drainer 状态修改为 paused：

```
CHANGE DRAINER TO NODE_STATE ='paused' FOR NODE_ID 'drainer1';
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
SHOW DRAINER STATUS;
```

```
+----------|----------------|--------|--------------------|---------------------|
|  NodeID  |     Address    | State  |   Max_Commit_Ts    |    Update_Time      |
+----------|----------------|--------|--------------------|---------------------|
| drainer1 | 127.0.0.3:8249 | Paused | 408553768673342532 | 2019-04-30 00:00:03 |
+----------|----------------|--------|--------------------|---------------------|
| drainer2 | 127.0.0.4:8249 | Online | 408553768673345531 | 2019-05-01 00:00:04 |
+----------|----------------|--------|--------------------|---------------------|
2 rows in set (0.00 sec)
```

14.11.2.25.2 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.25.3 另请参阅

- SHOW PUMP STATUS
- SHOW DRAINER STATUS
- CHANGE PUMP STATUS

14.11.2.26 CHANGE PUMP

CHANGE PUMP 语句用于修改集群中 Pump 的状态信息。

> **注意：**
>
> Pump 在正常运行时会自动上报状态到 PD，仅在 Pump 处于异常情况导致实际状态与 PD 中保存的状态信息不一致时，使用该语句修改 PD 中存储的 Pump 状态信息。

14.11.2.26.1 示例

```
SHOW PUMP STATUS;
```

```
+--------|----------------|--------|--------------------|---------------------|
| NodeID |     Address    | State  |   Max_Commit_Ts    |    Update_Time      |
+--------|----------------|--------|--------------------|---------------------|
| pump1  | 127.0.0.1:8250 | Online | 408553768673342237 | 2019-04-30 00:00:01 |
+--------|----------------|--------|--------------------|---------------------|
| pump2  | 127.0.0.2:8250 | Online | 408553768673342335 | 2019-05-01 00:00:02 |
+--------|----------------|--------|--------------------|---------------------|
2 rows in set (0.00 sec)
```

可以看出 pump1 已经超过一天没有更新状态，该 Pump 处于异常状态，但是 State 仍然为 Online，使用 CHANGE PUMP 将该 Pump 状态修改为 paused：

```
CHANGE PUMP TO NODE_STATE ='paused' FOR NODE_ID 'pump1';
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
SHOW PUMP STATUS;
```

```
+--------|----------------|--------|--------------------|---------------------|
| NodeID |     Address    | State  |   Max_Commit_Ts    |    Update_Time      |
+--------|----------------|--------|--------------------|---------------------|
| pump1  | 127.0.0.1:8250 | Paused | 408553768673342237 | 2019-04-30 00:00:01 |
+--------|----------------|--------|--------------------|---------------------|
| pump2  | 127.0.0.2:8250 | Online | 408553768673342335 | 2019-05-01 00:00:02 |
+--------|----------------|--------|--------------------|---------------------|
2 rows in set (0.00 sec)
```

14.11.2.26.2 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.26.3 另请参阅

- SHOW PUMP STATUS
- SHOW DRAINER STATUS
- CHANGE DRAINER STATUS

14.11.2.27 COMMIT

COMMIT 语句用于在 TiDB 服务器内部提交事务。

在不使用 BEGIN 或 START TRANSACTION 语句的情况下，TiDB 中每一个查询语句本身也会默认作为事务处理，自动提交，确保了与 MySQL 的兼容。

14.11.2.27.1 语法图

```
CommitStmt ::=
    'COMMIT' CompletionTypeWithinTransaction?

CompletionTypeWithinTransaction ::=
    'AND' ( 'CHAIN' ( 'NO' 'RELEASE' )? | 'NO' 'CHAIN' ( 'NO'? 'RELEASE' )? )
|   'NO'? 'RELEASE'
```

14.11.2.27.2 示例

```
CREATE TABLE t1 (a int NOT NULL PRIMARY KEY);
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
START TRANSACTION;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
INSERT INTO t1 VALUES (1);
```

```
Query OK, 1 row affected (0.00 sec)
```

```
COMMIT;
```

```
Query OK, 0 rows affected (0.01 sec)
```

14.11.2.27.3 MySQL 兼容性

- 目前，默认情况下 TiDB 使用元数据锁 (MDL) 来防止 DDL 语句修改事务使用的表，这与 MySQL 中的元数据锁在行为上有一些差别，具体可以参考元数据锁。
- TiDB 3.0.8 及更新版本默认使用悲观事务模式。在乐观事务模型下，需要考虑到修改的行已被另一个事务修改，导致 COMMIT 语句可能执行失败的情况。
- 启用乐观事务模型后，UNIQUE 和 PRIMARY KEY 约束检查将延迟直至语句提交。当 COMMIT 语句失败时，这可能导致其他问题。可通过设置 tidb_constraint_check_in_place=ON 来改变该行为。
- TiDB 解析但忽略 ROLLBACK AND [NO] RELEASE 语法。在 MySQL 中，使用该语法可在提交事务后立即断开客户端会话。在 TiDB 中，建议使用客户端程序的 mysql_close() 来实现该功能。
- TiDB 解析但忽略 ROLLBACK AND [NO] CHAIN 语法。在 MySQL 中，使用该语法可在提交当前事务时立即以相同的隔离级别开启新事务。在 TiDB 中，推荐直接开启新事务。

14.11.2.27.4 另请参阅

- START TRANSACTION
- ROLLBACK
- BEGIN
- 事务的惰性检查

14.11.2.28 CREATE [GLOBAL|SESSION] BINDING

CREATE [GLOBAL|SESSION] BINDING 语句用于在 TiDB 中创建新的执行计划绑定。绑定可用于将优化器 Hint 插入语句中，而无需更改底层查询。

BINDING 语句可以在 GLOBAL 或者 SESSION 作用域内创建执行计划绑定。在不指定作用域时，默认的作用域为 SESSION。

被绑定的 SQL 语句会被参数化后存储到系统表中。在处理 SQL 查询时，只要参数化后的 SQL 语句和系统表中某个被绑定的 SQL 语句一致，并且系统变量 tidb_use_plan_baselines 的值为 ON (其默认值为 ON)，即可使用相应的优化器 Hint。如果存在多个可匹配的执行计划，优化器会从中选择代价最小的一个进行绑定。更多信息，请参考创建绑定。

14.11.2.28.1 语法图

```
CreateBindingStmt ::=
    'CREATE' GlobalScope 'BINDING' ( 'FOR' BindableStmt 'USING' BindableStmt
|   'FROM' 'HISTORY' 'USING' 'PLAN' 'DIGEST' PlanDigest )

GlobalScope ::=
    ( 'GLOBAL' | 'SESSION' )?

BindableStmt ::=
    ( SelectStmt | UpdateStmt | InsertIntoStmt | ReplaceIntoStmt | DeleteStmt )
```

14.11.2.28.2 示例

你可以根据 SQL 或历史执行计划创建绑定。

下面的示例演示如何根据 SQL 创建绑定。

```
CREATE TABLE t1 (
    id INT NOT NULL PRIMARY KEY auto_increment,
    b INT NOT NULL,
    pad VARBINARY(255),
    INDEX(b)
   );
Query OK, 0 rows affected (0.07 sec)

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM dual;
Query OK, 1 row affected (0.01 sec)
Records: 1  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 8 rows affected (0.00 sec)
Records: 8  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 1000 rows affected (0.04 sec)
Records: 1000  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 100000 rows affected (1.74 sec)
Records: 100000  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 100000 rows affected (2.15 sec)
Records: 100000  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 100000 rows affected (2.64 sec)
Records: 100000  Duplicates: 0  Warnings: 0
```

```
SELECT SLEEP(1);
+----------+
| SLEEP(1) |
+----------+
|        0 |
+----------+
1 row in set (1.00 sec)

ANALYZE TABLE t1;
Query OK, 0 rows affected (1.33 sec)

EXPLAIN ANALYZE SELECT * FROM t1 WHERE b = 123;
+--
    ↪ ----------------------------+---------+---------+-----------+---------------------+-----------------
    ↪
| id                            | estRows | actRows | task      | access object       |
    ↪ execution info                                                          | operator info
    ↪                       | memory         | disk |
+--
    ↪ ----------------------------+---------+---------+-----------+---------------------+-----------------
    ↪
| IndexLookUp_10                | 583.00  | 297     | root      |                     | time
    ↪ :10.545072ms, loops:2, rpc num: 1, rpc time:398.359µs, proc keys:297  |
    ↪                                     | 109.1484375 KB | N/A  |
| ├─IndexRangeScan_8(Build)     | 583.00  | 297     | cop[tikv] | table:t1, index:b(b) | time:0s
    ↪ , loops:4                                                               | range:[123,123],
    ↪ keep order:false | N/A            | N/A  |
| └─TableRowIDScan_9(Probe)     | 583.00  | 297     | cop[tikv] | table:t1             | time:12
    ↪ ms, loops:4                                                             | keep order:false
    ↪                    | N/A            | N/A  |
+--
    ↪ ----------------------------+---------+---------+-----------+---------------------+-----------------
    ↪
3 rows in set (0.02 sec)

CREATE SESSION BINDING FOR
    SELECT * FROM t1 WHERE b = 123
   USING
    SELECT * FROM t1 IGNORE INDEX (b) WHERE b = 123;
Query OK, 0 rows affected (0.00 sec)

EXPLAIN ANALYZE  SELECT * FROM t1 WHERE b = 123;
+--
    ↪ ----------------------+-----------+---------+-----------+---------------+-----------------------------
```

```
    ↪
| id                    | estRows   | actRows | task      | access object | execution info
    ↪                                                                  | operator info      |
    ↪ memory        | disk |
+--
    ↪ ----------------------+----------+---------+----------+--------------+-------------------------------
    ↪
| TableReader_7           | 583.00    | 297     | root      |               | time:222.32506ms,
    ↪ loops:2, rpc num: 1, rpc time:222.078952ms, proc keys:301010 | data:Selection_6   |
    ↪ 88.6640625 KB | N/A  |
| └─Selection_6           | 583.00    | 297     | cop[tikv] |               | time:224ms, loops
    ↪ :298                                                          | eq(test.t1.b, 123) | N/A
    ↪           | N/A  |
|   └─TableFullScan_5     | 301010.00 | 301010  | cop[tikv] | table:t1      | time:220ms, loops
    ↪ :298                                                          | keep order:false   | N/A
    ↪           | N/A  |
+--
    ↪ ----------------------+----------+---------+----------+--------------+-------------------------------
    ↪
3 rows in set (0.22 sec)

SHOW SESSION BINDINGS\G
*************************** 1. row ***************************
Original_sql: select * from t1 where b = ?
    Bind_sql: SELECT * FROM t1 IGNORE INDEX (b) WHERE b = 123
  Default_db: test
      Status: using
 Create_time: 2020-05-22 14:38:03.456
 Update_time: 2020-05-22 14:38:03.456
     Charset: utf8mb4
   Collation: utf8mb4_0900_ai_ci
1 row in set (0.00 sec)

DROP SESSION BINDING FOR SELECT * FROM t1 WHERE b = 123;
Query OK, 0 rows affected (0.00 sec)

EXPLAIN ANALYZE  SELECT * FROM t1 WHERE b = 123;
+--
    ↪ ---------------------------+---------+---------+----------+---------------------+-----------------
    ↪
| id                          | estRows | actRows | task      | access object       |
    ↪ execution info                                                          | operator info
    ↪                     | memory         | disk |
+--
    ↪ ---------------------------+---------+---------+----------+---------------------+-----------------
```

```
    ↪
| IndexLookUp_10                | 583.00  | 297     | root      |                       | time
    ↪ :5.31206ms, loops:2, rpc num: 1, rpc time:665.927µs, proc keys:297  |
    ↪                                     | 109.1484375 KB | N/A  |
| ├─IndexRangeScan_8(Build)     | 583.00  | 297     | cop[tikv] | table:t1, index:b(b) | time:0s
    ↪ , loops:4                                                          | range:[123,123], keep
    ↪ order:false | N/A            | N/A  |
| └─TableRowIDScan_9(Probe)     | 583.00  | 297     | cop[tikv] | table:t1             | time:0s
    ↪ , loops:4                                                          | keep order:false
    ↪                   | N/A            | N/A  |
+--
    ↪ ----------------------------+---------+---------+-----------+----------------------+-----------------
    ↪
3 rows in set (0.01 sec)
```

下面的示例演示如何根据历史执行计划创建绑定。

```
mysql> CREATE TABLE t(id INT PRIMARY KEY , a INT, KEY(a));
Query OK, 0 rows affected (0.06 sec)

mysql> SELECT /*+ IGNORE_INDEX(t, a) */ * FROM t WHERE a = 1;
Empty set (0.01 sec)

mysql> SELECT plan_digest FROM INFORMATION_SCHEMA.STATEMENTS_SUMMARY WHERE QUERY_SAMPLE_TEXT = '
    ↪ SELECT /*+ IGNORE_INDEX(t, a) */ * FROM t WHERE a = 1';
+------------------------------------------------------------------+
| plan_digest                                                      |
+------------------------------------------------------------------+
| 4e3159169cc63c14b139a4e7d72eae1759875c9a9581f94bb2079aae961189cb |
+------------------------------------------------------------------+
1 row in set (0.01 sec)

mysql> CREATE BINDING FROM HISTORY USING PLAN DIGEST '4
    ↪ e3159169cc63c14b139a4e7d72eae1759875c9a9581f94bb2079aae961189cb';
Query OK, 0 rows affected (0.02 sec)

mysql> SELECT * FROM t WHERE a = 1;
Empty set (0.01 sec)

mysql> SELECT @@LAST_PLAN_FROM_BINDING;
+--------------------------+
| @@LAST_PLAN_FROM_BINDING |
+--------------------------+
|                        1 |
+--------------------------+
1 row in set (0.01 sec)
```

14.11.2.28.3 MySQL 兼容性

`CREATE [GLOBAL|SESSION] BINDING` 语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.28.4 另请参阅

- DROP [GLOBAL|SESSION] BINDING
- SHOW [GLOBAL|SESSION] BINDINGS
- ANALYZE
- Optimizer Hints
- 执行计划管理 (SPM)

14.11.2.29 CREATE DATABASE

`CREATE DATABASE` 语句用于在 TiDB 上创建新数据库。按照 SQL 标准，“数据库” 一词在 MySQL 术语中最接近“schema”。

14.11.2.29.1 语法图

```
CreateDatabaseStmt ::=
    'CREATE' 'DATABASE' IfNotExists DBName DatabaseOptionListOpt

IfNotExists ::=
    ( 'IF' 'NOT' 'EXISTS' )?

DBName ::=
    Identifier

DatabaseOptionListOpt ::=
    DatabaseOptionList?

DatabaseOptionList ::=
    DatabaseOption ( ','? DatabaseOption )*

DatabaseOption ::=
    DefaultKwdOpt ( CharsetKw '='? CharsetName | 'COLLATE' '='? CollationName | 'ENCRYPTION' '='?
        ↪  EncryptionOpt )
|   DefaultKwdOpt PlacementPolicyOption

PlacementPolicyOption ::=
    "PLACEMENT" "POLICY" EqOpt PolicyName
|   "PLACEMENT" "POLICY" (EqOpt | "SET") "DEFAULT"
```

14.11.2.29.2 语法说明

CREATE DATABASE 用于创建数据库，并可以指定数据库的默认属性（如数据库默认字符集、排序规则）。CREATE SCHEMA 跟 CREATE DATABASE 操作效果一样。

```
CREATE {DATABASE | SCHEMA} [IF NOT EXISTS] db_name
    [create_specification] ...

create_specification:
    [DEFAULT] CHARACTER SET [=] charset_name
  | [DEFAULT] COLLATE [=] collation_name
```

当创建已存在的数据库且不指定使用 IF NOT EXISTS 时会报错。

create_specification 选项用于指定数据库具体的 CHARACTER SET 和 COLLATE。目前 TiDB 只支持部分的字符集和排序规则，请参照字符集支持。

14.11.2.29.3 示例

```
CREATE DATABASE mynewdatabase;
```

```
Query OK, 0 rows affected (0.09 sec)
```

```
USE mynewdatabase;
```

```
Database changed
```

```
CREATE TABLE t1 (a int);
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
SHOW TABLES;
```

```
+-------------------------+
| Tables_in_mynewdatabase |
+-------------------------+
| t1                      |
+-------------------------+
1 row in set (0.00 sec)
```

14.11.2.29.4 MySQL 兼容性

CREATE DATABASE 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.29.5 另请参阅

- USE
- ALTER DATABASE
- DROP DATABASE
- SHOW DATABASES

14.11.2.30 CREATE INDEX

`CREATE INDEX` 语句用于在已有表中添加新索引，功能等同于 `ALTER TABLE .. ADD INDEX`。包含该语句提供了 MySQL 兼容性。

14.11.2.30.1 语法图

```
CreateIndexStmt ::=
    'CREATE' IndexKeyTypeOpt 'INDEX' IfNotExists Identifier IndexTypeOpt 'ON' TableName '('
        ↪ IndexPartSpecificationList ')' IndexOptionList IndexLockAndAlgorithmOpt

IndexKeyTypeOpt ::=
    ( 'UNIQUE' | 'SPATIAL' | 'FULLTEXT' )?

IfNotExists ::=
    ( 'IF' 'NOT' 'EXISTS' )?

IndexTypeOpt ::=
    IndexType?

IndexPartSpecificationList ::=
    IndexPartSpecification ( ',' IndexPartSpecification )*

IndexOptionList ::=
    IndexOption*

IndexLockAndAlgorithmOpt ::=
    ( LockClause AlgorithmClause? | AlgorithmClause LockClause? )?

IndexType ::=
    ( 'USING' | 'TYPE' ) IndexTypeName

IndexPartSpecification ::=
    ( ColumnName OptFieldLen | '(' Expression ')' ) Order

IndexOption ::=
    'KEY_BLOCK_SIZE' '='? LengthNum
|   IndexType
|   'WITH' 'PARSER' Identifier
```

```
|   'COMMENT' stringLit
|   IndexInvisible

IndexTypeName ::=
    'BTREE'
|   'HASH'
|   'RTREE'

ColumnName ::=
    Identifier ( '.' Identifier ( '.' Identifier )? )?

OptFieldLen ::=
    FieldLen?

IndexNameList ::=
    ( Identifier | 'PRIMARY' )? ( ',' ( Identifier | 'PRIMARY' ) )*

KeyOrIndex ::=
    'Key' | 'Index'
```

14.11.2.30.2 示例

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, c1 INT NOT NULL);
```

```
Query OK, 0 rows affected (0.10 sec)
```

```
INSERT INTO t1 (c1) VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 5 rows affected (0.02 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
EXPLAIN SELECT * FROM t1 WHERE c1 = 3;
```

```
+-------------------------+----------+-----------+---------------+--------------------------------+
    ↪
| id                      | estRows  | task      | access object | operator info
    ↪                  |
+-------------------------+----------+-----------+---------------+--------------------------------+
    ↪
| TableReader_7           | 10.00    | root      |               | data:Selection_6
    ↪               |
| └─Selection_6           | 10.00    | cop[tikv] |               | eq(test.t1.c1, 3)
    ↪              |
|   └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t1      | keep order:false, stats:
    ↪ pseudo |
```

```
+-------------------------+----------+-----------+--------------+--------------------------------+
    ↪
3 rows in set (0.00 sec)
```

```
CREATE INDEX c1 ON t1 (c1);
```

```
Query OK, 0 rows affected (0.30 sec)
```

```
EXPLAIN SELECT * FROM t1 WHERE c1 = 3;
```

```
+------------------------+---------+-----------+-----------------------+-----------------------------------
    ↪
| id                     | estRows | task      | access object         | operator info
    ↪                              |
+------------------------+---------+-----------+-----------------------+-----------------------------------
    ↪
| IndexReader_6          | 10.00   | root      |                       | index:IndexRangeScan_5
    ↪                        |
| └─IndexRangeScan_5     | 10.00   | cop[tikv] | table:t1, index:c1(c1) | range:[3,3], keep
    ↪ order:false, stats:pseudo |
+------------------------+---------+-----------+-----------------------+-----------------------------------
    ↪
2 rows in set (0.00 sec)
```

```
ALTER TABLE t1 DROP INDEX c1;
```

```
Query OK, 0 rows affected (0.30 sec)
```

```
CREATE UNIQUE INDEX c1 ON t1 (c1);
```

```
Query OK, 0 rows affected (0.31 sec)
```

14.11.2.30.3 表达式索引

在一些场景中，查询的条件往往是基于某个表达式进行过滤。在这些场景中，一般的索引不能生效，执行查询只能遍历整个表，导致查询性能较差。表达式索引是一种特殊的索引，能将索引建立于表达式上。在创建了表达式索引后，基于表达式的查询便可以使用上索引，极大提升查询的性能。

假设要基于 `lower(col1)` 这个表达式建立索引，示例的 SQL 语句如下：

```
CREATE INDEX idx1 ON t1 ((lower(col1)));
```

或者等价的语句：

```
ALTER TABLE t1 ADD INDEX idx1((lower(col1)));
```

还可以在建表的同时指定表达式索引：

```
CREATE TABLE t1(col1 char(10), col2 char(10), index((lower(col1))));
```

注意：

表达式索引中的表达式需要用 (和) 包围起来，否则会报语法错误。

删除表达式索引与删除普通索引的方法一致：

```
DROP INDEX idx1 ON t1;
```

注意：

表达式索引涉及众多表达式。为了确保正确性，当前仅允许经充分测试的一部分函数用于创建表达式索引，即生产环境中仅允许表达式中包含这些函数。这些函数可以通过查询变量 `tidb_allow_function_for_expression_index` 得到。在后续版本中，这些函数会持续增加。目前允许的函数如下:

```
json_array, json_array_append, json_array_insert, json_contains, json_contains_path
    ↪ , json_depth, json_extract, json_insert, json_keys, json_length,
    ↪ json_merge_patch, json_merge_preserve, json_object, json_pretty, json_quote
    ↪ , json_remove, json_replace, json_search, json_set, json_storage_size,
    ↪ json_type, json_unquote, json_valid, lower, md5, reverse, tidb_shard, upper
    ↪ , vitess_hash
```

对于以上列表之外的函数，由于未完成充分测试，当前仍为实验特性，不建议在生产环境中使用。其他的表达式例如运算符、`cast` 和 `case when` 也同样为实验特性，不建议在生产环境中使用。如果仍然希望使用，可以在TiDB 配置文件中进行以下设置：

```
allow-expression-index = true
```

表达式索引不能为主键。

表达式索引中的表达式不能包含以下内容：

- 易变函数，例如 `rand()` 和 `now()` 等。
- 系统变量以及用户变量。
- 子查询。
- AUTO_INCREMENT 属性的列。一个例外是设置系统变量 `tidb_enable_auto_increment_in_generated` 为 `true` 后，可以去掉该限制。
- 窗口函数。
- row 函数。例如 `create table t (j json, key k (((j,j))));`。
- 聚合函数。

表达式索引将隐式占用名字，`_V$_{index_name}_{index_offset}`，如果已有相同名字的列存在，创建表达式索引将报错。如果后续新增相同名字的列，也会报错。

在表达式索引中，表达式的函数参数个数必须正确。

当索引的表达式使用了字符串相关的函数时，受返回类型以及其长度的影响，创建表达式索引可能会失败。这时可以使用 `cast()` 函数显式指定返回的类型以及长度。例如表达式 `repeat(a, 3)`，为了能根据该表达式建立表达式索引，需要将表达式改写为 `cast(repeat(a, 3)as char(20))` 这样的形式。

当查询语句中的表达式与表达式索引中的表达式一致时，优化器可以为该查询选择使用表达式索引。依赖于统计信息，某些情况下优化器不一定选择表达式索引。这时可以通过 hint 指定强制使用表达式索引。

在以下示例中，假设建立在 `lower(col1)` 表达式上的索引为 `idx`。

当读取的结果为相同的表达式时，可以使用表达式索引。例如：

```
SELECT lower(col1) FROM t;
```

当过滤的条件中有相同的表达式时，可以使用表达式索引。例如：

```
SELECT * FROM t WHERE lower(col1) = "a";
SELECT * FROM t WHERE lower(col1) > "a";
SELECT * FROM t WHERE lower(col1) BETWEEN "a" AND "b";
SELECT * FROM t WHERE lower(col1) in ("a", "b");
SELECT * FROM t WHERE lower(col1) > "a" AND lower(col1) < "b";
SELECT * FROM t WHERE lower(col1) > "b" OR lower(col1) < "a";
```

当查询按照相同的表达式进行排序时，可以使用表达式索引。例如：

```
SELECT * FROM t ORDER BY lower(col1);
```

当聚合函数或者 `GROUP BY` 中包含相同的表达式时，可以使用表达式索引。例如：

```
SELECT max(lower(col1)) FROM t;
SELECT min(col1) FROM t GROUP BY lower(col1);
```

要查看表达式索引对应的表达式，可执行 `show index` 或查看系统表 `information_schema.tidb_indexes` 以及 `information_schema.STATISTICS` 表，输出中 `Expression` 这一列显示对应的表达式。对于非表达式索引，该列的值为 `NULL`。

维护表达式索引的代价比一般的索引更高，因为在插入或者更新每一行时都需要计算出表达式的值。因为表达式的值已经存储在索引中，所以当优化器选择表达式索引时，表达式的值就不需要再计算。因此，当查询速度比插入速度和更新速度更重要时，可以考虑建立表达式索引。

表达式索引的语法和限制与 MySQL 相同，是通过将索引建立在隐藏的虚拟生成列 (generated virtual column) 上来实现的。因此所支持的表达式继承了虚拟生成列的所有限制。

14.11.2.30.4 多值索引

> **警告：**
>
> 当前该功能为实验特性，不建议在生产环境中使用。

多值索引是一种定义在数组列上的二级索引。在普通索引中，一条索引记录对应一条数据记录 (1:1)。而在多值索引中，存在多条索引记录对应一条数据记录 (N:1)。多值索引用于索引 JSON 数组。例如，一个定义在 `zipcode` 字段上的多值索引会对每一个 `zipcode` 中的记录产生一条索引记录。

```
{
    "user":"Bob",
    "user_id":31,
    "zipcode":[94477,94536]
}
```

创建多值索引

创建多值索引与创建表达式索引的方法一致。在索引定义中使用 `CAST(... AS ... ARRAY)` 表达式来创建一个多值索引。

```
mysql> CREATE TABLE customers (
    id BIGINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name CHAR(10),
    custinfo JSON,
    INDEX zips((CAST(custinfo->'$.zipcode' AS UNSIGNED ARRAY)))
);
```

多值索引可以被定义为唯一索引：

```
mysql> CREATE TABLE customers (
    id BIGINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name CHAR(10),
    custinfo JSON,
    UNIQUE INDEX zips( (CAST(custinfo->'$.zipcode' AS UNSIGNED ARRAY)))
);
```

当被定义为唯一索引时，试图插入重复数据将会报错：

```
mysql> INSERT INTO customers VALUES (1, 'pingcap', '{"zipcode": [1,2]}');
Query OK, 1 row affected (0.01 sec)

mysql> INSERT INTO customers VALUES (1, 'pingcap', '{"zipcode": [2,3]}');
ERROR 1062 (23000): Duplicate entry '2' for key 'customers.zips'
```

允许同一条记录存在重复的值，不同记录出现重复值时将报错：

```
-- 插入成功
mysql> INSERT INTO t1 VALUES('[1,1,2]');
mysql> INSERT INTO t1 VALUES('[3,3,3,4,4,4]');

-- 插入失败
mysql> INSERT INTO t1 VALUES('[1,2]');
mysql> INSERT INTO t1 VALUES('[2,3]');
```

多值索引也可以被定义为复合索引：

```
mysql> CREATE TABLE customers (
    id BIGINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name CHAR(10),
    custinfo JSON,
    INDEX zips(name, (CAST(custinfo->'$.zipcode' AS UNSIGNED ARRAY)))
);
```

当被定义为复合索引时，多值部分可以出现在任意位置，但是只能出现一次。

```
mysql> CREATE TABLE customers (
    id BIGINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name CHAR(10),
    custinfo JSON,
    INDEX zips(name, (CAST(custinfo->'$.zipcode' AS UNSIGNED ARRAY)), (CAST(custinfo->'$.zipcode'
        ↪  AS UNSIGNED ARRAY)))
);
ERROR 1235 (42000): This version of TiDB doesn't yet support 'more than one multi-valued key part
    ↪  per index'.
```

写入的数据必须与多值索引的定义类型完全匹配，否则数据写入失败：

```
-- zipcode 字段中的所有元素必须为 UNSIGNED 类型
mysql> INSERT INTO customers VALUES (1, 'pingcap', '{"zipcode": [-1]}');
ERROR 3752 (HY000): Value is out of range for expression index 'zips' at row 1

mysql> INSERT INTO customers VALUES (1, 'pingcap', '{"zipcode": ["1"]}'); -- 与 MySQL 不兼容
ERROR 3903 (HY000): Invalid JSON value for CAST for expression index 'zips'

mysql> INSERT INTO customers VALUES (1, 'pingcap', '{"zipcode": [1]}');
Query OK, 1 row affected (0.00 sec)
```

使用多值索引

请参考索引的选择。

特性与限制

- 如果是空 JSON 数组，则不会有对应的索引记录。

- CAST(... AS ... ARRAY) 中的目标类型不能是 BINARY、JSON、YEAR、FLOAT、DOUBLE、DECIMAL。其中源类型必须是 JSON。
- 无法使用多值索引进行排序。
- 只允许在 JSON 数组上建立多值索引。
- 多值索引不可以作为主键或外键。
- 多值索引使用额外的存储空间为：平均每行数组元素个数 * 普通二级索引使用空间。
- 相比于普通索引，DML 会对多值索引产生更多的索引记录的修改，因此多值索引会带来比普通索引更大的性能影响。
- 由于多值索引是一种特殊的表达式索引，因此具有表达式索引的限制。
- 由于多值索引的统计信息暂时还没有被收集，多值索引的选择率基于固定假设。当查询命中多个多值索引时，可能无法选取最优的索引。在这种情况下，建议用优化器提示use_index_merge 来固定执行计划。具体使用方式请参考使用多值索引。
- 条件复杂的查询有可能无法选择到多值索引，多值索引支持的条件模式请参考使用多值索引。

工具兼容性

多值索引为实验特性，与 TiDB 相关的工具或功能存在一些已知的兼容性问题：

- LOAD DATA 语句和TiDB Lightning 不支持向含有多值索引的表中导入数据。
- DM 和TiCDC 不支持迁移、同步含有多值索引的表。
- sync-diff-inspector 不支持对含有多值索引的表进行数据校验。

14.11.2.30.5 不可见索引

不可见索引 (Invisible Indexes) 不会被查询优化器使用：

```
CREATE TABLE t1 (c1 INT, c2 INT, UNIQUE(c2));
CREATE UNIQUE INDEX c1 ON t1 (c1) INVISIBLE;
```

具体可以参考ALTER INDEX。

14.11.2.30.6 相关系统变量

和 CREATE INDEX 语句相关的系统变量有 tidb_ddl_enable_fast_reorg、tidb_ddl_reorg_worker_cnt 、tidb_ddl_reorg_batch_size 、tidb_ddl_reorg_priority 和 tidb_enable_auto_increment_in_generated，具体可以参考系统变量。

14.11.2.30.7 MySQL 兼容性

- TiDB 支持解析 FULLTEXT 和 SPATIAL 语法，但尚不支持使用 FULLTEXT，HASH 和 SPATIAL 索引。
- 不支持降序索引（类似于 MySQL 5.7）。
- 无法向表中添加 CLUSTERED 类型的 PRIMARY KEY。要了解关于 CLUSTERED 主键的详细信息，请参考聚簇索引。
- 表达式索引与视图存在兼容性问题。通过视图进行查询时，无法使用上表达式索引。

- 表达式索引与 Binding 存在兼容性问题。当表达式索引中的表达式存在常量时，对应查询所建的 Binding 会扩大范围。假设表达式索引中的表达式为 `a+1`，对应的查询条件为 `a+1 > 2`。则建立的 Binding 为 `a+? > ?`，这会导致像 `a+2 > 2` 这样的查询也会强制使用表达式索引，得到一个较差的执行计划。这同样影响 SQL Plan Management (SPM) 中的捕获和演进功能。
- 多值索引写入的数据必须与定义类型完全匹配，否则数据写入失败。详见创建多值索引。

14.11.2.30.8 另请参阅

- 索引的选择
- 错误索引的解决方案
- ADD INDEX
- DROP INDEX
- RENAME INDEX
- ALTER INDEX
- ADD COLUMN
- CREATE TABLE
- EXPLAIN

14.11.2.31 CREATE PLACEMENT POLICY

`CREATE PLACEMENT POLICY` 用于创建命名的放置策略，随后可以将该策略分配给表、分区或数据库。

14.11.2.31.1 语法图

```
CreatePolicyStmt ::=
    "CREATE" "PLACEMENT" "POLICY" IfNotExists PolicyName PlacementOptionList

PolicyName ::=
    Identifier

PlacementOptionList ::=
    PlacementOption
|   PlacementOptionList PlacementOption
|   PlacementOptionList ',' PlacementOption

PlacementOption ::=
    CommonPlacementOption
|   SugarPlacementOption
|   AdvancedPlacementOption

CommonPlacementOption ::=
    "FOLLOWERS" EqOpt LengthNum

SugarPlacementOption ::=
    "PRIMARY_REGION" EqOpt stringLit
```

```
|   "REGIONS" EqOpt stringLit
|   "SCHEDULE" EqOpt stringLit

AdvancedPlacementOption ::=
    "LEARNERS" EqOpt LengthNum
|   "CONSTRAINTS" EqOpt stringLit
|   "LEADER_CONSTRAINTS" EqOpt stringLit
|   "FOLLOWER_CONSTRAINTS" EqOpt stringLit
|   "LEARNER_CONSTRAINTS" EqOpt stringLit
```

14.11.2.31.2 示例

> **注意：**
>
> 如要查看所在集群中可用的区域，见SHOW PLACEMENT LABELS。如果未看到任何可用的区域，此 TiKV 集群在部署时可能未正确设置标签 (label)。

```
CREATE PLACEMENT POLICY p1 PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1" FOLLOWERS=4;
CREATE TABLE t1 (a INT) PLACEMENT POLICY=p1;
SHOW CREATE PLACEMENT POLICY p1;
```

```
Query OK, 0 rows affected (0.08 sec)

Query OK, 0 rows affected (0.10 sec)

+--------+-------------------------------------------------------------------------------------------------------+
    ↪
| Policy | Create Policy
    ↪                                                                                                      |
+--------+-------------------------------------------------------------------------------------------------------+
    ↪
| p1     | CREATE PLACEMENT POLICY `p1` PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1"
    ↪ FOLLOWERS=4 |
+--------+-------------------------------------------------------------------------------------------------------+
    ↪
1 row in set (0.00 sec)
```

14.11.2.31.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.31.4 另请参阅

- Placement Rules in SQL
- SHOW PLACEMENT
- ALTER PLACEMENT POLICY
- DROP PLACEMENT POLICY

14.11.2.32 CREATE RESOURCE GROUP

`CREATE RESOURCE GROUP` 语句用于在当前所选数据库中创建资源组。

14.11.2.32.1 语法图

```
CreateResourceGroupStmt ::=
   "CREATE" "RESOURCE" "GROUP" IfNotExists ResourceGroupName ResourceGroupOptionList

IfNotExists ::=
    ('IF' 'NOT' 'EXISTS')?

ResourceGroupName ::=
   Identifier

ResourceGroupOptionList ::=
    DirectResourceGroupOption
|   ResourceGroupOptionList DirectResourceGroupOption
|   ResourceGroupOptionList ',' DirectResourceGroupOption

DirectResourceGroupOption ::=
    "RU_PER_SEC" EqOpt stringLit
|   "PRIORITY" EqOpt ResourceGroupPriorityOption
|   "BURSTABLE"

ResourceGroupPriorityOption ::=
    LOW
|   MEDIUM
|   HIGH
```

资源组的 `ResourceGroupName` 是全局唯一的，不允许重复。

TiDB 支持以下 `DirectResourceGroupOption`, 其中Request Unit (RU) 是 TiDB 对 CPU、IO 等系统资源统一抽象的单位。

参数	含义	举例
`RU_PER_SEC`	每秒 RU 填充的速度	`RU_PER_SEC = 500` 表示此资源组每秒回填 500 个 RU。
`PRIORITY`	任务在 TiKV 上处理的绝对优先级	`PRIORITY = HIGH` 表示优先级高。若未指定，则默认为 ME
`BURSTABLE`	允许对应的资源组超出配额后使用空余的系统资源。	

注意：

- CREATE RESOURCE GROUP 语句只能在全局变量tidb_enable_resource_control 设置为 ON 时才能执行。
- TiDB 集群在初始化时会自动创建 default 资源组，其 RU_PER_SEC 的默认值为 UNLIMITED (等同于 INT 类型最大值，即 2147483647)，且为 BURSTABLE 模式。所有未绑定资源组的请求都将自动绑定至此资源组。在新建配置其他资源组时，建议根据实际情况修改 default 资源组的配置。

14.11.2.32.2 示例

创建 rg1 和 rg2 两个资源组。

```
DROP RESOURCE GROUP IF EXISTS rg1;
```

```
Query OK, 0 rows affected (0.22 sec)
```

```
CREATE RESOURCE GROUP IF NOT EXISTS rg1
  RU_PER_SEC = 100
  PRIORITY = HIGH
  BURSTABLE;
```

```
Query OK, 0 rows affected (0.08 sec)
```

```
CREATE RESOURCE GROUP IF NOT EXISTS rg2
  RU_PER_SEC = 200;
```

```
Query OK, 0 rows affected (0.08 sec)
```

```
SELECT * FROM information_schema.resource_groups WHERE NAME ='rg1' or NAME = 'rg2';
```

```
+------+------------+----------+-----------+
| NAME | RU_PER_SEC | PRIORITY | BURSTABLE |
+------+------------+----------+-----------+
| rg1  |        100 | HIGH     | YES       |
| rg2  |        200 | MEDIUM   | NO        |
+------+------------+----------+-----------+
2 rows in set (1.30 sec)
```

14.11.2.32.3 MySQL 兼容性

MySQL 也支持 CREATE RESOURCE GROUP，但是接受的参数和 TiDB 不同，两者并不兼容。

14.11.2.32.4 另请参阅

- DROP RESOURCE GROUP
- ALTER RESOURCE GROUP
- ALTER USER RESOURCE GROUP
- RU

14.11.2.33 CREATE ROLE

`CREATE ROLE` 语句是基于角色的访问控制 (RBAC) 操作的一部分，用于创建新角色并将新角色分配给用户。

14.11.2.33.1 语法图

```
CreateRoleStmt ::=
    'CREATE' 'ROLE' IfNotExists RoleSpec (',' RoleSpec)*

IfNotExists ::=
    ('IF' 'NOT' 'EXISTS')?

RoleSpec ::=
    Rolename
```

14.11.2.33.2 示例

以 `root` 用户连接 TiDB：

```
mysql -h 127.0.0.1 -P 4000 -u root
```

创建新角色 `analyticsteam` 和新用户 `jennifer`：

```
CREATE ROLE analyticsteam;
Query OK, 0 rows affected (0.02 sec)

GRANT SELECT ON test.* TO analyticsteam;
Query OK, 0 rows affected (0.02 sec)

CREATE USER jennifer;
Query OK, 0 rows affected (0.01 sec)

GRANT analyticsteam TO jennifer;
Query OK, 0 rows affected (0.01 sec)
```

以 `jennifer` 用户连接 TiDB：

```
mysql -h 127.0.0.1 -P 4000 -u jennifer
```

需要注意的是，默认情况下，用户 jennifer 需要执行 SET ROLE analyticsteam 语句才能使用与 analyticsteam 角色相关联的权限：

```
SHOW GRANTS;
+-------------------------------------------+
| Grants for User                           |
+-------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'      |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+-------------------------------------------+
2 rows in set (0.00 sec)

SHOW TABLES in test;
ERROR 1044 (42000): Access denied for user 'jennifer'@'%' to database 'test'
SET ROLE analyticsteam;
Query OK, 0 rows affected (0.00 sec)

SHOW GRANTS;
+-------------------------------------------+
| Grants for User                           |
+-------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'      |
| GRANT SELECT ON test.* TO 'jennifer'@'%'  |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+-------------------------------------------+
3 rows in set (0.00 sec)

SHOW TABLES IN test;
+----------------+
| Tables_in_test |
+----------------+
| t1             |
+----------------+
1 row in set (0.00 sec)
```

以 root 用户连接 TiDB：

```
mysql -h 127.0.0.1 -P 4000 -u root
```

执行 SET DEFAULT ROLE 语句将用户 jennifer 与 analyticsteam 角色相关联：

```
SET DEFAULT ROLE analyticsteam TO jennifer;
Query OK, 0 rows affected (0.02 sec)
```

以 jennifer 用户连接 TiDB：

```
mysql -h 127.0.0.1 -P 4000 -u jennifer
```

此时 jennifer 用户无需执行 SET ROLE 语句就能拥有 analyticsteam 角色相关联的权限：

```
SHOW GRANTS;
+-------------------------------------------+
| Grants for User                           |
+-------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'      |
| GRANT SELECT ON test.* TO 'jennifer'@'%'  |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+-------------------------------------------+
3 rows in set (0.00 sec)

SHOW TABLES IN test;
+----------------+
| Tables_in_test |
+----------------+
| t1             |
+----------------+
1 row in set (0.00 sec)
```

14.11.2.33.3 MySQL 兼容性

CREATE ROLE 语句与 MySQL 8.0 的“角色”功能完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.33.4 另请参阅

- DROP ROLE
- GRANT <role>
- REVOKE <role>
- SET ROLE
- SET DEFAULT ROLE
- 基于角色的访问控制

14.11.2.34 CREATE SEQUENCE

CREATE SEQUENCE 语句用于在 TiDB 中创建序列对象。序列是一种与表、视图对象平级的数据库对象，用于生成自定义的序列化 ID。

14.11.2.34.1 语法图

```
CreateSequenceStmt ::=
    'CREATE' 'SEQUENCE' IfNotExists TableName CreateSequenceOptionListOpt
        ↪ CreateTableOptionListOpt

IfNotExists ::=
```

```
    ('IF' 'NOT' 'EXISTS')?

TableName ::=
    Identifier ('.' Identifier)?

CreateSequenceOptionListOpt ::=
    SequenceOption*

SequenceOptionList ::=
    SequenceOption

SequenceOption ::=
    ( 'INCREMENT' ( '='? | 'BY' ) | 'START' ( '='? | 'WITH' ) | ( 'MINVALUE' | 'MAXVALUE' | '
        ↪ CACHE' ) '='? ) SignedNum
|   'NOMINVALUE'
|   'NO' ( 'MINVALUE' | 'MAXVALUE' | 'CACHE' | 'CYCLE' )
|   'NOMAXVALUE'
|   'NOCACHE'
|   'CYCLE'
|   'NOCYCLE'
```

14.11.2.34.2 语法说明

```
CREATE [TEMPORARY] SEQUENCE [IF NOT EXISTS] sequence_name
    [ INCREMENT [ BY | = ] increment ]
    [ MINVALUE [=] minvalue | NO MINVALUE | NOMINVALUE ]
    [ MAXVALUE [=] maxvalue | NO MAXVALUE | NOMAXVALUE ]
    [ START [ WITH | = ] start ]
    [ CACHE [=] cache | NOCACHE | NO CACHE]
    [ CYCLE | NOCYCLE | NO CYCLE]
    [table_options]
```

14.11.2.34.3 参数说明

参数	默认值	描述
TEMPORARY ↪	false ↪	TiDB 暂时不支持 TEMPORARY ↪ 选项，仅在语法上做兼容。
INCREMENT ↪	1	指定序列的步长。其正负值可以控制序列的增长方向。

参数	默认值	描述
MINVALUE	或 -9223372036854775807	指定序列的最小值。当 INCREMENT > 0 时，默认值为 1；当 INCREMENT < 0 时，默认值为 -9223372036854775807。
MAXVALUE	9223372036854775806 或 -1	指定序列的最大值。当 INCREMENT > 0 时，默认值为 9223372036854775806；当 INCREMENT < 0 时，默认值为 -1。

参数	默认值	描述
START ↪	MINVALUE ↪ 或 MAXVALUE ↪	指定序列的初始值。当 INCREMENT ↪ > 0 时，默认值为 MINVALUE ↪ ; 当 INCREMENT ↪ < 0 时，默认值为 MAXVALUE ↪ 。
CACHE ↪	1000	指定每个 TiDB 本地缓存序列的大小。

参数	默认值	描述
CYCLE ↪	NO ↪ CYCLE ↪	指定序列用完之后是否要循环使用。在 CYCLE ↪ 的情况下，当 INCREMENT ↪ > 0 时，序列用完后的后续起始值为 MINVALUE ↪ ；当 INCREMENT ↪ < 0 时，序列用完后的后续起始值为 MAXVALUE ↪ 。

14.11.2.34.4 SEQUENCE 函数

主要通过表达式函数来操纵序列的使用。

- NEXTVAL 或 NEXT VALUE FOR

本质上都是 nextval() 函数，获取序列对象的下一个有效值，其参数为序列的 identifier。

- LASTVAL

  lastval() 函数，用于获取本会话上一个使用过的值。如果没有值，则为 NULL，其参数为序列的 identifier。

- SETVAL

  setval() 函数，用于设置序列的增长。其第一参数为序列的 identifier，第二个参数为 num。

> **注意：**
>
> 在 TiDB 序列的实现中，SETVAL 函数并不能改变序列增长的初始步调或循环步调。在 SETVAL 之后只会返回符合步调规律的下一个有效的序列值。

14.11.2.34.5 示例

- 创建一个默认参数的序列对象。

```
CREATE SEQUENCE seq;
```

```
Query OK, 0 rows affected (0.06 sec)
```

- 使用 nextval() 函数获取序列对象的下一个值。

```
SELECT nextval(seq);
```

```
+--------------+
| nextval(seq) |
+--------------+
|            1 |
+--------------+
1 row in set (0.02 sec)
```

- 使用 lastval() 函数获取本会话上一次调用序列对象所产生的值。

```
SELECT lastval(seq);
```

```
+--------------+
| lastval(seq) |
+--------------+
|            1 |
+--------------+
1 row in set (0.02 sec)
```

- 使用 `setval()` 函数设置序列对象当前值的位置。

```
SELECT setval(seq, 10);
```

```
+-----------------+
| setval(seq, 10) |
+-----------------+
|              10 |
+-----------------+
1 row in set (0.01 sec)
```

- 也可使用 `next value for` 语法获取序列的下一个值。

```
SELECT next value for seq;
```

```
+--------------------+
| next value for seq |
+--------------------+
|                 11 |
+--------------------+
1 row in set (0.00 sec)
```

- 创建一个默认自定义参数的序列对象。

```
CREATE SEQUENCE seq2 start 3 increment 2 minvalue 1 maxvalue 10 cache 3;
```

```
Query OK, 0 rows affected (0.01 sec)
```

- 当本会话还未使用过序列对象时，`lastval()` 函数返回 NULL 值。

```
SELECT lastval(seq2);
```

```
+---------------+
| lastval(seq2) |
+---------------+
|          NULL |
+---------------+
1 row in set (0.01 sec)
```

- 序列对象 `nextval()` 的第一个有效值为 `start` 值。

```
SELECT nextval(seq2);
```

```
+---------------+
| nextval(seq2) |
+---------------+
|             3 |
```

```
+---------------+
1 row in set (0.00 sec)
```

- 使用 `setval()` 虽然可以改变序列对象当前值的位置，但是无法改变下一个值的等差规律。

```
SELECT setval(seq2, 6);
```

```
+-----------------+
| setval(seq2, 6) |
+-----------------+
|               6 |
+-----------------+
1 row in set (0.00 sec)
```

- 使用 `nextval()` 下一个值获取时，会遵循序列定义的等差规律。

```
SELECT next value for seq2;
```

```
+---------------------+
| next value for seq2 |
+---------------------+
|                   7 |
+---------------------+
1 row in set (0.00 sec)
```

- 可以将序列的下一个值作为列的默认值来使用。

```
CREATE TABLE t(a int default next value for seq2);
```

```
Query OK, 0 rows affected (0.02 sec)
```

- 下列示例中，因为没有指定值，会直接获取 `seq2` 的默认值来使用。

```
INSERT into t values();
```

```
Query OK, 1 row affected (0.00 sec)
```

```
SELECT * from t;
```

```
+------+
| a    |
+------+
|    9 |
+------+
1 row in set (0.00 sec)
```

- 下列示例中，因为没有指定值，会直接获取 seq2 的默认值来使用。由于 seq2 的下一个值超过了上述示例 (CREATE SEQUENCE seq2 start 3 increment 2 minvalue 1 maxvalue 10 cache 3;) 的定义范围，所以会显示报错。

```
INSERT into t values();
```

```
ERROR 4135 (HY000): Sequence 'test.seq2' has run out
```

14.11.2.34.6 MySQL 兼容性

该语句是 TiDB 的扩展，序列的实现借鉴自 MariaDB。

除了 SETVAL 函数外，其他函数的 “步调 (progressions)” 与 MariaDB 一致。这里的步调是指，序列中的数在定义之后会产生一定的等差关系。SETVAL 虽然可以将序列的当前值进行移动设置，但是后续出现的值仍会遵循原有的等差关系。

示例如下：

```
1, 3, 5, ...            // 序列遵循起始为 1、步长为 2 的等差关系。
select setval(seq, 6)   // 设置序列的当前值为 6。
7, 9, 11, ...           // 后续产生值仍会遵循这个等差关系。
```

在 CYCLE 模式下，序列的起始值第一轮为 start，后续轮次将会是 MinValue (increment > 0) 或 MaxValue (increment < 0)。

14.11.2.34.7 另请参阅

- DROP SEQUENCE
- SHOW CREATE SEQUENCE

14.11.2.35 CREATE TABLE LIKE

CREATE TABLE LIKE 语句用于复制已有表的定义，但不复制任何数据。

14.11.2.35.1 语法图

```
CreateTableLikeStmt ::=
    'CREATE' OptTemporary 'TABLE' IfNotExists TableName LikeTableWithOrWithoutParen OnCommitOpt

OptTemporary ::=
    ( 'TEMPORARY' | ('GLOBAL' 'TEMPORARY') )?

LikeTableWithOrWithoutParen ::=
    'LIKE' TableName
|   '(' 'LIKE' TableName ')'

OnCommitOpt ::=
    ('ON' 'COMMIT' 'DELETE' 'ROWS')?
```

14.11.2.35.2 示例

```
CREATE TABLE t1 (a INT NOT NULL);
```

```
Query OK, 0 rows affected (0.13 sec)
```

```
INSERT INTO t1 VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 5 rows affected (0.02 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
SELECT * FROM t1;
```

```
+---+
| a |
+---+
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
+---+
5 rows in set (0.00 sec)
```

```
CREATE TABLE t2 LIKE t1;
```

```
Query OK, 0 rows affected (0.10 sec)
```

```
SELECT * FROM t2;
```

```
Empty set (0.00 sec)
```

14.11.2.35.3 Region 的预切分

如果被复制的表定义了 PRE_SPLIT_REGIONS 属性，则通过 CREATE TABLE LIKE 语句复制的表，会继承该属性并在建表时预切分 Region。关于 PRE_SPLIT_REGIONS 属性的说明，参见CREATE TABLE 语句。

14.11.2.35.4 MySQL 兼容性

CREATE TABLE LIKE 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.35.5 另请参阅

- CREATE TABLE
- SHOW CREATE TABLE

14.11.2.36 CREATE TABLE

CREATE TABLE 语句用于在当前所选数据库中创建新表，与 MySQL 中 CREATE TABLE 语句的行为类似。另可参阅单独的 CREATE TABLE LIKE 文档。

14.11.2.36.1 语法图

```
CreateTableStmt ::=
    'CREATE' OptTemporary 'TABLE' IfNotExists TableName ( TableElementListOpt
        ↪ CreateTableOptionListOpt PartitionOpt DuplicateOpt AsOpt CreateTableSelectOpt |
        ↪ LikeTableWithOrWithoutParen ) OnCommitOpt

OptTemporary ::=
    ( 'TEMPORARY' | ('GLOBAL' 'TEMPORARY') )?

IfNotExists ::=
    ('IF' 'NOT' 'EXISTS')?

TableName ::=
    Identifier ('.' Identifier)?

TableElementListOpt ::=
    ( '(' TableElementList ')' )?

TableElementList ::=
    TableElement ( ',' TableElement )*

TableElement ::=
    ColumnDef
|   Constraint

ColumnDef ::=
    ColumnName ( Type | 'SERIAL' ) ColumnOptionListOpt

ColumnOptionListOpt ::=
    ColumnOption*

ColumnOptionList ::=
    ColumnOption*

ColumnOption ::=
    'NOT'? 'NULL'
|   'AUTO_INCREMENT'
|   PrimaryOpt 'KEY'
|   'UNIQUE' 'KEY'?
|   'DEFAULT' DefaultValueExpr
```

```
|   'SERIAL' 'DEFAULT' 'VALUE'
|   'ON' 'UPDATE' NowSymOptionFraction
|   'COMMENT' stringLit
|   ConstraintKeywordOpt 'CHECK' '(' Expression ')' EnforcedOrNotOrNotNullOpt
|   GeneratedAlways 'AS' '(' Expression ')' VirtualOrStored
|   ReferDef
|   'COLLATE' CollationName
|   'COLUMN_FORMAT' ColumnFormat
|   'STORAGE' StorageMedia
|   'AUTO_RANDOM' OptFieldLen

Constraint ::=
    IndexDef
|   ForeignKeyDef

IndexDef ::=
    ( 'INDEX' | 'KEY' ) IndexName? '(' KeyPartList ')' IndexOption?

KeyPartList ::=
    KeyPart ( ',' KeyPart )*

KeyPart ::=
    ColumnName ( '(' Length ')')? ( 'ASC' | 'DESC' )?
|   '(' Expression ')' ( 'ASC' | 'DESC' )?

IndexOption ::=
    'COMMENT' String
|   ( 'VISIBLE' | 'INVISIBLE' )

ForeignKeyDef
         ::= ( 'CONSTRAINT' Identifier )? 'FOREIGN' 'KEY'
             Identifier? '(' ColumnName ( ',' ColumnName )* ')'
             'REFERENCES' TableName '(' ColumnName ( ',' ColumnName )* ')'
             ( 'ON' 'DELETE' ReferenceOption )?
             ( 'ON' 'UPDATE' ReferenceOption )?

ReferenceOption
         ::= 'RESTRICT'
           | 'CASCADE'
           | 'SET' 'NULL'
           | 'SET' 'DEFAULT'
           | 'NO' 'ACTION'

CreateTableOptionListOpt ::=
    TableOptionList?
```

```
PartitionOpt ::=
    ( 'PARTITION' 'BY' PartitionMethod PartitionNumOpt SubPartitionOpt PartitionDefinitionListOpt
        ↪  )?

DuplicateOpt ::=
    ( 'IGNORE' | 'REPLACE' )?

TableOptionList ::=
    TableOption ( ','? TableOption )*

TableOption ::=
    PartDefOption
|   DefaultKwdOpt ( CharsetKw EqOpt CharsetName | 'COLLATE' EqOpt CollationName )
|   ( 'AUTO_INCREMENT' | 'AUTO_ID_CACHE' | 'AUTO_RANDOM_BASE' | 'AVG_ROW_LENGTH' | 'CHECKSUM' | '
    ↪ TABLE_CHECKSUM' | 'KEY_BLOCK_SIZE' | 'DELAY_KEY_WRITE' | 'SHARD_ROW_ID_BITS' | '
    ↪ PRE_SPLIT_REGIONS' ) EqOpt LengthNum
|   ( 'CONNECTION' | 'PASSWORD' | 'COMPRESSION' ) EqOpt stringLit
|   RowFormat
|   ( 'STATS_PERSISTENT' | 'PACK_KEYS' ) EqOpt StatsPersistentVal
|   ( 'STATS_AUTO_RECALC' | 'STATS_SAMPLE_PAGES' ) EqOpt ( LengthNum | 'DEFAULT' )
|   'STORAGE' ( 'MEMORY' | 'DISK' )
|   'SECONDARY_ENGINE' EqOpt ( 'NULL' | StringName )
|   'UNION' EqOpt '(' TableNameListOpt ')'
|   'ENCRYPTION' EqOpt EncryptionOpt
|    'TTL' EqOpt TimeColumnName '+' 'INTERVAL' Expression TimeUnit (TTLEnable EqOpt ( 'ON' | 'OFF
    ↪ ' ))? (TTLJobInterval EqOpt stringLit)?
|   PlacementPolicyOption

OnCommitOpt ::=
    ('ON' 'COMMIT' 'DELETE' 'ROWS')?

PlacementPolicyOption ::=
    "PLACEMENT" "POLICY" EqOpt PolicyName
|   "PLACEMENT" "POLICY" (EqOpt | "SET") "DEFAULT"
```

TiDB 支持以下 table_option。TiDB 会解析并忽略其他 table_option 参数，例如 AVG_ROW_LENGTH、CHECKSUM、COMPRESSION、CONNECTION、DELAY_KEY_WRITE、ENGINE、KEY_BLOCK_SIZE、MAX_ROWS、MIN_ROWS、ROW_FORMAT 和 STATS_PERSISTENT。

参数	含义	举例
AUTO_INCREMENT	自增字段初始值	AUTO_INCREMENT = 5
SHARD_ROW_ID_BITS	用来设置隐式 _tidb_rowid 的分片数量的 bit 位数	SHARD_ROW_ID_BITS = 4

参数	含义	举例
PRE_SPLIT_REGIONS	用来在建表时预先均匀切分 2^(PRE_SPLIT_REGIONS) 个 Region	PRE_SPLIT_REGIONS = 4
AUTO_ID_CACHE	用来指定 Auto ID 在 TiDB 实例中 Cache 的大小，默认情况下 TiDB 会根据 Auto ID 分配速度自动调整	AUTO_ID_CACHE = 200
AUTO_RANDOM_BASE	用来指定 AutoRandom 自增部分的初始值，该参数可以被认为属于内部接口的一部分，对于用户而言请忽略	AUTO_RANDOM_BASE = 0
CHARACTER SET	指定该表所使用的字符集	CHARACTER SET = ‘utf8mb4’
COLLATE	指定该表所使用的字符集排序规则	COLLATE = ‘utf8mb4_bin’
COMMENT	注释信息	COMMENT = ‘comment info’

注意：

在 TiDB 配置文件中，split-table 默认开启。当该配置项开启时，建表操作会为每个表建立单独的 Region，详情参见TiDB 配置文件描述。

14.11.2.36.2 示例

创建一张简单表并插入一行数据：

```
CREATE TABLE t1 (a int);
DESC t1;
SHOW CREATE TABLE t1\G
INSERT INTO t1 (a) VALUES (1);
SELECT * FROM t1;
```

```
mysql> drop table if exists t1;
Query OK, 0 rows affected (0.23 sec)

mysql> CREATE TABLE t1 (a int);
Query OK, 0 rows affected (0.09 sec)

mysql> DESC t1;
+-------+---------+------+------+---------+-------+
| Field | Type    | Null | Key  | Default | Extra |
+-------+---------+------+------+---------+-------+
| a     | int(11) | YES  |      | NULL    |       |
+-------+---------+------+------+---------+-------+
1 row in set (0.00 sec)

mysql> SHOW CREATE TABLE t1\G
```

```
*************************** 1. row ***************************
       Table: t1
Create Table: CREATE TABLE `t1` (
  `a` int(11) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin
1 row in set (0.00 sec)

mysql> INSERT INTO t1 (a) VALUES (1);
Query OK, 1 row affected (0.03 sec)

mysql> SELECT * FROM t1;
+------+
| a    |
+------+
|    1 |
+------+
1 row in set (0.00 sec)
```

删除一张表。如果该表不存在，就建一张表：

```
DROP TABLE IF EXISTS t1;
CREATE TABLE IF NOT EXISTS t1 (
 id BIGINT NOT NULL PRIMARY KEY auto_increment,
 b VARCHAR(200) NOT NULL
);
DESC t1;
```

```
mysql> DROP TABLE IF EXISTS t1;
Query OK, 0 rows affected (0.22 sec)
mysql> CREATE TABLE IF NOT EXISTS t1 (
          id BIGINT NOT NULL PRIMARY KEY auto_increment,
          b VARCHAR(200) NOT NULL
         );
Query OK, 0 rows affected (0.08 sec)
mysql> DESC t1;
+-------+--------------+------+------+---------+----------------+
| Field | Type         | Null | Key  | Default | Extra          |
+-------+--------------+------+------+---------+----------------+
| id    | bigint(20)   | NO   | PRI  | NULL    | auto_increment |
| b     | varchar(200) | NO   |      | NULL    |                |
+-------+--------------+------+------+---------+----------------+
2 rows in set (0.00 sec)
```

14.11.2.36.3 MySQL 兼容性

- 支持除空间类型以外的所有数据类型。
- 不支持 FULLTEXT，HASH 和 SPATIAL 索引。
- 为了与 MySQL 兼容，index_col_name 属性支持 length 选项，最大长度默认限制为 3072 字节。此长度限制可以通过配置项 max-index-length 更改，具体请参阅TiDB 配置文件描述。
- 为了与 MySQL 兼容，TiDB 会解析但忽略 index_col_name 属性的 [ASC | DESC] 索引排序选项。
- COMMENT 属性不支持 WITH PARSER 选项。
- TiDB 在单个表中默认支持 1017 列，最大可支持 4096 列。InnoDB 中相应的数量限制为 1017 列，MySQL 中的硬限制为 4096 列。详情参阅TiDB 使用限制。
- 当前仅支持 Range、 Hash 和 Range Columns（单列）类型的分区表，详情参阅分区表。
- TiDB 会解析并忽略 CHECK 约束，与 MySQL 5.7 相兼容。详情参阅CHECK 约束。

14.11.2.36.4 另请参阅

- 数据类型
- DROP TABLE
- CREATE TABLE LIKE
- SHOW CREATE TABLE

14.11.2.37 CREATE USER

CREATE USER 语句用于创建带有指定密码的新用户。和 MySQL 一样，在 TiDB 权限系统中，用户是用户名和用户名所连接主机的组合。因此，可创建一个用户 'newuser2'@'192.168.1.1'，使其只能通过 IP 地址 192.168.1.1 进行连接。相同的用户名从不同主机登录时可能会拥有不同的权限。

14.11.2.37.1 语法图

```
CreateUserStmt ::=
    'CREATE' 'USER' IfNotExists UserSpecList RequireClauseOpt ConnectionOptions PasswordOption
        ↪ LockOption AttributeOption ResourceGroupNameOption

IfNotExists ::=
    ('IF' 'NOT' 'EXISTS')?

UserSpecList ::=
    UserSpec ( ',' UserSpec )*

UserSpec ::=
    Username AuthOption

AuthOption ::=
    ( 'IDENTIFIED' ( 'BY' ( AuthString | 'PASSWORD' HashString ) | 'WITH' StringName ( 'BY'
        ↪ AuthString | 'AS' HashString )? ) )?

StringName ::=
    stringLit
```

```
|   Identifier

PasswordOption ::= ( 'PASSWORD' 'EXPIRE' ( 'DEFAULT' | 'NEVER' | 'INTERVAL' N 'DAY' )? | '
    ↪ PASSWORD' 'HISTORY' ( 'DEFAULT' | N ) | 'PASSWORD' 'REUSE' 'INTERVAL' ( 'DEFAULT' | N '
    ↪ DAY' ) | 'FAILED_LOGIN_ATTEMPTS' N | 'PASSWORD_LOCK_TIME' ( N | 'UNBOUNDED' ) )*

LockOption ::= ( 'ACCOUNT' 'LOCK' | 'ACCOUNT' 'UNLOCK' )?

AttributeOption ::= ( 'COMMENT' CommentString | 'ATTRIBUTE' AttributeString )?

ResourceGroupNameOption::= ( 'RESOURCE' 'GROUP' Identifier)?
```

14.11.2.37.2 示例

创建一个密码为 newuserpassword 的用户。

```
CREATE USER 'newuser' IDENTIFIED BY 'newuserpassword';
```

```
Query OK, 1 row affected (0.04 sec)
```

创建一个只能在 192.168.1.1 登录的用户，密码为 newuserpassword。

```
CREATE USER 'newuser2'@'192.168.1.1' IDENTIFIED BY 'newuserpassword';
```

```
Query OK, 1 row affected (0.02 sec)
```

创建一个要求在登录时使用 TLS 连接的用户。

```
CREATE USER 'newuser3'@'%' IDENTIFIED BY 'newuserpassword' REQUIRE SSL;
```

```
Query OK, 1 row affected (0.02 sec)
```

创建一个要求在登录时提供指定客户端证书的用户。

```
CREATE USER 'newuser4'@'%' IDENTIFIED BY 'newuserpassword' REQUIRE ISSUER '/C=US/ST=California/L=
    ↪ San Francisco/O=PingCAP';
```

```
Query OK, 1 row affected (0.02 sec)
```

创建一个初始状态下被锁住的用户。

```
CREATE USER 'newuser5'@'%' ACCOUNT LOCK;
```

```
Query OK, 1 row affected (0.02 sec)
```

创建一个带注释的用户。

```
CREATE USER 'newuser6'@'%' COMMENT 'This user is created only for test';
SELECT * FROM information_schema.user_attributes;
```

```
+-----------+------+-----------------------------------------------------+
| USER      | HOST | ATTRIBUTE                                           |
+-----------+------+-----------------------------------------------------+
| newuser6  | %    | {"comment": "This user is created only for test"} |
+-----------+------+-----------------------------------------------------+
1 rows in set (0.00 sec)
```

创建一个具有邮箱 (`email`) 属性的用户。

```
CREATE USER 'newuser7'@'%' ATTRIBUTE '{"email": "user@pingcap.com"}';
SELECT * FROM information_schema.user_attributes;
```

```
+-----------+------+-----------------------------------------------------+
| USER      | HOST | ATTRIBUTE                                           |
+-----------+------+-----------------------------------------------------+
| newuser7  | %    | {"email": "user@pingcap.com"} |
+-----------+------+-----------------------------------------------------+
1 rows in set (0.00 sec)
```

创建一个禁止重复使用最近 5 次密码的用户。

```
CREATE USER 'newuser8'@'%' PASSWORD HISTORY 5;
```

```
Query OK, 1 row affected (0.02 sec)
```

创建一个密码已经手动过期的用户。

```
CREATE USER 'newuser9'@'%' PASSWORD EXPIRE;
```

```
Query OK, 1 row affected (0.02 sec)
```

创建一个使用资源组 `rg1` 的用户：

```
CREATE USER 'newuser7'@'%' RESOURCE GROUP rg1;
SELECT USER, HOST, USER_ATTRIBUTES FROM MYSQL.USER WHERE USER='newuser7';
```

```
+-----------+------+-----------------------------------------------------+
| USER      | HOST | USER_ATTRIBUTES                                     |
+-----------+------+-----------------------------------------------------+
| newuser7  | %    | {"resource_group": "rg1"} |
+-----------+------+-----------------------------------------------------+
1 rows in set (0.00 sec)
```

14.11.2.37.3 MySQL 兼容性

- TiDB 不支持 `WITH MAX_QUERIES_PER_HOUR`、`WITH MAX_UPDATES_PER_HOUR`、`WITH MAX_USER_CONNECTIONS` 等 `CREATE` 选项。
- TiDB 不支持 `DEFAULT ROLE` 选项。
- TiDB 不支持 `PASSWORD EXPIRE`、`PASSWORD HISTORY` 等有关密码限制的 `CREATE` 选项。
- 对于 TiDB 尚不支持的 `CREATE` 选项。这些选项可被解析，但会被忽略。

14.11.2.37.4 另请参阅

- Security Compatibility with MySQL
- DROP USER
- SHOW CREATE USER
- ALTER USER
- Privilege Management

14.11.2.38 CREATE VIEW

使用 `CREATE VIEW` 语句将 `SELECT` 语句保存为类似于表的可查询对象。TiDB 中的视图是非物化的，这意味着在查询视图时，TiDB 将在内部重写查询，以将视图定义与 SQL 查询结合起来。

14.11.2.38.1 语法图

```
CreateViewStmt ::=
    'CREATE' OrReplace ViewAlgorithm ViewDefiner ViewSQLSecurity 'VIEW' ViewName ViewFieldList '
        ↪ AS' CreateViewSelectOpt ViewCheckOption

OrReplace ::=
    ( 'OR' 'REPLACE' )?

ViewAlgorithm ::=
    ( 'ALGORITHM' '=' ( 'UNDEFINED' | 'MERGE' | 'TEMPTABLE' ) )?

ViewDefiner ::=
    ( 'DEFINER' '=' Username )?

ViewSQLSecurity ::=
    ( 'SQL' 'SECURITY' ( 'DEFINER' | 'INVOKER' ) )?

ViewName ::= TableName

ViewFieldList ::=
    ( '(' Identifier ( ',' Identifier )* ')' )?

ViewCheckOption ::=
    ( 'WITH' ( 'CASCADED' | 'LOCAL' ) 'CHECK' 'OPTION' )?
```

14.11.2.38.2 示例

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, c1 INT NOT NULL);
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
INSERT INTO t1 (c1) VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 5 rows affected (0.03 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
CREATE VIEW v1 AS SELECT * FROM t1 WHERE c1 > 2;
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
SELECT * FROM t1;
```

```
+----+----+
| id | c1 |
+----+----+
|  1 |  1 |
|  2 |  2 |
|  3 |  3 |
|  4 |  4 |
|  5 |  5 |
+----+----+
5 rows in set (0.00 sec)
```

```
SELECT * FROM v1;
```

```
+----+----+
| id | c1 |
+----+----+
|  3 |  3 |
|  4 |  4 |
|  5 |  5 |
+----+----+
3 rows in set (0.00 sec)
```

```
INSERT INTO t1 (c1) VALUES (6);
```

```
Query OK, 1 row affected (0.01 sec)
```

```
SELECT * FROM v1;
```

```
+----+----+
| id | c1 |
+----+----+
|  3 |  3 |
|  4 |  4 |
|  5 |  5 |
|  6 |  6 |
+----+----+
4 rows in set (0.00 sec)
```

```
INSERT INTO v1 (c1) VALUES (7);
```

```
ERROR 1105 (HY000): insert into view v1 is not supported now.
```

14.11.2.38.3 MySQL 兼容性

- 目前 TiDB 中的任何视图都不可被插入，也不可被更新（即不支持 INSERT VIEW，也不支持 UPDATE VIEW）。WITH CHECK OPTION 只做了语法兼容但不生效。
- 目前 TiDB 中的视图不支持 ALTER VIEW，但可以使用 CREATE OR REPLACE 替代。
- 目前 ALGORITHM 字段在 TiDB 中只做了语法兼容但不生效，TiDB 目前只支持 MERGE 算法。

14.11.2.38.4 另请参阅

- DROP VIEW
- CREATE TABLE
- SHOW CREATE TABLE
- DROP TABLE

14.11.2.39 DEALLOCATE

DEALLOCATE 语句用于为服务器端预处理语句提供 SQL 接口。

14.11.2.39.1 语法图

```
DeallocateStmt ::=
    DeallocateSym 'PREPARE' Identifier

DeallocateSym ::=
    'DEALLOCATE'
|   'DROP'

Identifier ::=
    identifier
```

```
|   UnReservedKeyword
|   NotKeywordToken
|   TiDBKeyword
```

14.11.2.39.2 示例

```
PREPARE mystmt FROM 'SELECT ? as num FROM DUAL';
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
SET @number = 5;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
EXECUTE mystmt USING @number;
```

```
+------+
| num  |
+------+
| 5    |
+------+
1 row in set (0.00 sec)
```

```
DEALLOCATE PREPARE mystmt;
```

```
Query OK, 0 rows affected (0.00 sec)
```

14.11.2.39.3 MySQL 兼容性

DEALLOCATE 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.39.4 另请参阅

- PREPARE
- EXECUTE

14.11.2.40 DELETE

DELETE 语句用于从指定的表中删除行。

14.11.2.40.1 语法图

```
DeleteFromStmt ::=
    'DELETE' TableOptimizerHints PriorityOpt QuickOptional IgnoreOptional ( 'FROM' ( TableName
        ↪ TableAsNameOpt IndexHintListOpt WhereClauseOptional OrderByOptional LimitClause |
        ↪ TableAliasRefList 'USING' TableRefs WhereClauseOptional ) | TableAliasRefList 'FROM'
        ↪ TableRefs WhereClauseOptional )
```

14.11.2.40.2 示例

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, c1 INT NOT NULL);
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
INSERT INTO t1 (c1) VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 5 rows affected (0.03 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
SELECT * FROM t1;
```

```
+----+----+
| id | c1 |
+----+----+
|  1 |  1 |
|  2 |  2 |
|  3 |  3 |
|  4 |  4 |
|  5 |  5 |
+----+----+
5 rows in set (0.00 sec)
```

```
DELETE FROM t1 WHERE id = 4;
```

```
Query OK, 1 row affected (0.02 sec)
```

```
SELECT * FROM t1;
```

```
+----+----+
| id | c1 |
+----+----+
|  1 |  1 |
|  2 |  2 |
|  3 |  3 |
```

```
|  5 |  5 |
+----+----+
4 rows in set (0.00 sec)
```

14.11.2.40.3 MySQL 兼容性

DELETE 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.40.4 另请参阅

- INSERT
- SELECT
- UPDATE
- REPLACE

14.11.2.41 DESC

DESC 语句是EXPLAIN 的别名。包含该语句提供了 MySQL 兼容性。

14.11.2.42 DESCRIBE

DESCRIBE 语句为EXPLAIN 的别名。包含该语句提供了 MySQL 兼容性。

14.11.2.43 DO

DO 语句用于执行表达式，但不返回任何结果。大部分情况下，DO 相当于不返回结果的 `SELECT expr, ...`。

> **注意：**
>
> DO 只能执行表达式，所以不是所有能够用 SELECT 的地方都能用 DO 替换。例如 `DO id FROM t1` 就是不是合法的 SQL 语句，因为它引用了一张表。

DO 在 MySQL 中的一个主要应用场景是存储过程或者触发器。因为 TiDB 当前不支持存储过程和触发器，所以 DO 的实际使用场景较少。

14.11.2.43.1 语法图

```
DoStmt   ::= 'DO' ExpressionList

ExpressionList ::=
    Expression ( ',' Expression )*

Expression ::=
```

```
    ( singleAtIdentifier assignmentEq | 'NOT' | Expression ( logOr | 'XOR' | logAnd ) )
        ↪ Expression
|   'MATCH' '(' ColumnNameList ')' 'AGAINST' '(' BitExpr FulltextSearchModifierOpt ')'
|   PredicateExpr ( IsOrNotOp 'NULL' | CompareOp ( ( singleAtIdentifier assignmentEq )?
    ↪ PredicateExpr | AnyOrAll SubSelect ) )* ( IsOrNotOp ( trueKwd | falseKwd | 'UNKNOWN' ) )?
```

14.11.2.43.2 示例

这条 SELECT 语句会暂停执行，但同时也会返回一个结果集。

```
SELECT SLEEP(5);
```

```
+----------+
| SLEEP(5) |
+----------+
|        0 |
+----------+
1 row in set (5.00 sec)
```

如果使用 DO 的话，语句同样会暂停，但不会返回结果集。

```
DO SLEEP(5);
```

```
Query OK, 0 rows affected (5.00 sec)
```

```
DO SLEEP(1), SLEEP(1.5);
```

```
Query OK, 0 rows affected (2.50 sec)
```

14.11.2.43.3 MySQL 兼容性

DO 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.43.4 另请参阅

- SELECT

14.11.2.44 DROP [GLOBAL|SESSION] BINDING

DROP BINDING 语句用于删除指定的 SQL 绑定。绑定可用于将优化器 Hint 插入语句中，而无需更改底层查询。

BINDING 语句可以在 GLOBAL 或者 SESSION 作用域内删除执行计划绑定。在不指定作用域时，默认的作用域为 SESSION。

14.11.2.44.1 语法图

```
DropBindingStmt ::=
    'DROP' GlobalScope 'BINDING' 'FOR' ( BindableStmt ( 'USING' BindableStmt )?
|   'SQL' 'DIGEST' SqlDigest)

GlobalScope ::=
    ( 'GLOBAL' | 'SESSION' )?

BindableStmt ::=
    ( SelectStmt | UpdateStmt | InsertIntoStmt | ReplaceIntoStmt | DeleteStmt )
```

14.11.2.44.2 示例

你可以根据 SQL 语句或 `sql_digest` 删除绑定。

下面的示例演示如何根据 SQL 语句删除绑定。

```
CREATE TABLE t1 (
    id INT NOT NULL PRIMARY KEY auto_increment,
    b INT NOT NULL,
    pad VARBINARY(255),
    INDEX(b)
   );
Query OK, 0 rows affected (0.07 sec)

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM dual;
Query OK, 1 row affected (0.01 sec)
Records: 1  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 8 rows affected (0.00 sec)
Records: 8  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 1000 rows affected (0.04 sec)
Records: 1000  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
```

```
Query OK, 100000 rows affected (1.74 sec)
Records: 100000  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 100000 rows affected (2.15 sec)
Records: 100000  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 100000 rows affected (2.64 sec)
Records: 100000  Duplicates: 0  Warnings: 0

SELECT SLEEP(1);
+----------+
| SLEEP(1) |
+----------+
|        0 |
+----------+
1 row in set (1.00 sec)

ANALYZE TABLE t1;
Query OK, 0 rows affected (1.33 sec)

EXPLAIN ANALYZE SELECT * FROM t1 WHERE b = 123;
+--
    ↪ ----------------------------+---------+---------+-----------+---------------------+-----------------
    ↪
| id                          | estRows | actRows | task      | access object       |
    ↪ execution info                                                                | operator info
    ↪                       | memory         | disk |
+--
    ↪ ----------------------------+---------+---------+-----------+---------------------+-----------------
    ↪
| IndexLookUp_10              | 583.00  | 297     | root      |                     | time
    ↪ :10.545072ms, loops:2, rpc num: 1, rpc time:398.359µs, proc keys:297  |
    ↪                                     | 109.1484375 KB | N/A  |
| ├─IndexRangeScan_8(Build)     | 583.00  | 297     | cop[tikv] | table:t1, index:b(b) | time:0s
    ↪ , loops:4                                                             | range:[123,123],
    ↪ keep order:false | N/A            | N/A  |
| └─TableRowIDScan_9(Probe)     | 583.00  | 297     | cop[tikv] | table:t1             | time:12
    ↪ ms, loops:4                                                           | keep order:false
    ↪                   | N/A            | N/A  |
+--
    ↪ ----------------------------+---------+---------+-----------+---------------------+-----------------
```

```
    ↪
3 rows in set (0.02 sec)

CREATE SESSION BINDING FOR
    SELECT * FROM t1 WHERE b = 123
   USING
    SELECT * FROM t1 IGNORE INDEX (b) WHERE b = 123;
Query OK, 0 rows affected (0.00 sec)

EXPLAIN ANALYZE  SELECT * FROM t1 WHERE b = 123;
+--
    ↪ ----------------------+----------+---------+----------+--------------+------------------------------
    ↪
| id                    | estRows   | actRows | task      | access object | execution info
    ↪                                                                  | operator info      |
    ↪ memory        | disk |
+--
    ↪ ----------------------+----------+---------+----------+--------------+------------------------------
    ↪
| TableReader_7           | 583.00    | 297     | root      |               | time:222.32506ms,
    ↪ loops:2, rpc num: 1, rpc time:222.078952ms, proc keys:301010 | data:Selection_6   |
    ↪ 88.6640625 KB | N/A  |
| └─Selection_6           | 583.00    | 297     | cop[tikv] |               | time:224ms, loops
    ↪ :298                                                          | eq(test.t1.b, 123) | N/A
    ↪           | N/A  |
|   └─TableFullScan_5     | 301010.00 | 301010  | cop[tikv] | table:t1      | time:220ms, loops
    ↪ :298                                                          | keep order:false   | N/A
    ↪           | N/A  |
+--
    ↪ ----------------------+----------+---------+----------+--------------+------------------------------
    ↪
3 rows in set (0.22 sec)

SHOW SESSION BINDINGS\G
*************************** 1. row ***************************
Original_sql: select * from t1 where b = ?
    Bind_sql: SELECT * FROM t1 IGNORE INDEX (b) WHERE b = 123
  Default_db: test
      Status: using
 Create_time: 2020-05-22 14:38:03.456
 Update_time: 2020-05-22 14:38:03.456
     Charset: utf8mb4
   Collation: utf8mb4_0900_ai_ci
1 row in set (0.00 sec)
```

```
DROP SESSION BINDING FOR SELECT * FROM t1 WHERE b = 123;
Query OK, 0 rows affected (0.00 sec)

EXPLAIN ANALYZE  SELECT * FROM t1 WHERE b = 123;
+--
    ↪ ----------------------------+---------+---------+-----------+----------------------+-----------------
    ↪
| id                            | estRows | actRows | task      | access object        |
    ↪ execution info                                                                  | operator info
    ↪                       | memory         | disk |
+--
    ↪ ----------------------------+---------+---------+-----------+----------------------+-----------------
    ↪
| IndexLookUp_10                | 583.00  | 297     | root      |                      | time
    ↪ :5.31206ms, loops:2, rpc num: 1, rpc time:665.927µs, proc keys:297  |
    ↪                                     | 109.1484375 KB | N/A  |
| ├─IndexRangeScan_8(Build)     | 583.00  | 297     | cop[tikv] | table:t1, index:b(b) | time:0s
    ↪ , loops:4                                                                       | range:[123,123], keep
    ↪ order:false | N/A             | N/A  |
| └─TableRowIDScan_9(Probe)     | 583.00  | 297     | cop[tikv] | table:t1             | time:0s
    ↪ , loops:4                                                                       | keep order:false
    ↪                     | N/A             | N/A  |
+--
    ↪ ----------------------------+---------+---------+-----------+----------------------+-----------------
    ↪
3 rows in set (0.01 sec)

SHOW SESSION BINDINGS\G
Empty set (0.00 sec)
```

下面的示例演示如何根据 `sql_digest` 删除绑定。

```
mysql> CREATE TABLE t(id INT PRIMARY KEY , a INT, KEY(a));
Query OK, 0 rows affected (0.06 sec)

mysql> SELECT /*+ IGNORE_INDEX(t, a) */ * FROM t WHERE a = 1;
Empty set (0.01 sec)

mysql> SELECT plan_digest FROM INFORMATION_SCHEMA.STATEMENTS_SUMMARY WHERE QUERY_SAMPLE_TEXT = '
    ↪ SELECT /*+ IGNORE_INDEX(t, a) */ * FROM t WHERE a = 1';
+------------------------------------------------------------------+
| plan_digest                                                      |
+------------------------------------------------------------------+
| 4e3159169cc63c14b139a4e7d72eae1759875c9a9581f94bb2079aae961189cb |
+------------------------------------------------------------------+
1 row in set (0.01 sec)
```

```
mysql> CREATE BINDING FROM HISTORY USING PLAN DIGEST '4
    ↪ e3159169cc63c14b139a4e7d72eae1759875c9a9581f94bb2079aae961189cb';
Query OK, 0 rows affected (0.02 sec)

mysql> SELECT * FROM t WHERE a = 1;
Empty set (0.01 sec)

mysql> SELECT @@LAST_PLAN_FROM_BINDING;
+--------------------------+
| @@LAST_PLAN_FROM_BINDING |
+--------------------------+
|                        1 |
+--------------------------+
1 row in set (0.01 sec)

mysql> SHOW BINDINGS\G;
*************************** 1. row ***************************
Original_sql: select * from `test` . `t` where `a` = ?
    Bind_sql: SELECT /*+ use_index(@`sel_1` `test`.`t` ) ignore_index(`t` `a`)*/ * FROM `test`.`t
        ↪ ` WHERE `a` = 1
  Default_db: test
      Status: enabled
 Create_time: 2022-12-14 15:26:22.277
 Update_time: 2022-12-14 15:26:22.277
     Charset: utf8mb4
   Collation: utf8mb4_general_ci
      Source: history
  Sql_digest: 6909a1bbce5f64ade0a532d7058dd77b6ad5d5068aee22a531304280de48349f
 Plan_digest: 4e3159169cc63c14b139a4e7d72eae1759875c9a9581f94bb2079aae961189cb
1 row in set (0.02 sec)

ERROR:
No query specified

mysql> DROP BINDING FOR SQL DIGEST '6909
    ↪ a1bbce5f64ade0a532d7058dd77b6ad5d5068aee22a531304280de48349f';
Query OK, 0 rows affected (0.00 sec)

mysql> SHOW BINDINGS\G;
Empty set (0.01 sec)

ERROR:
No query specified
```

14.11.2.44.3 MySQL 兼容性

`DROP [GLOBAL|SESSION] BINDING` 语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.44.4 另请参阅

- CREATE [GLOBAL|SESSION] BINDING
- SHOW [GLOBAL|SESSION] BINDINGS
- ANALYZE
- Optimizer Hints
- 执行计划管理 (SPM)

14.11.2.45 DROP COLUMN

`DROP COLUMN` 语句用于从指定的表中删除列。在 TiDB 中，`COLUMN` 为在线操作，不会阻塞表中的数据读写。

14.11.2.45.1 语法图

```
AlterTableStmt
         ::= 'ALTER' 'IGNORE'? 'TABLE' TableName DropColumnSpec ( ',' DropColumnSpec )*

DropColumnSpec
         ::= 'DROP' 'COLUMN'? 'IF EXISTS'? ColumnName ( 'RESTRICT' | 'CASCADE' )?

ColumnName
         ::= Identifier ( '.' Identifier ( '.' Identifier )? )?
```

14.11.2.45.2 示例

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, col1 INT NOT NULL, col2 INT NOT NULL
    ↪ );
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
INSERT INTO t1 (col1,col2) VALUES (1,1),(2,2),(3,3),(4,4),(5,5);
```

```
Query OK, 5 rows affected (0.02 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
SELECT * FROM t1;
```

```
+----+------+------+
| id | col1 | col2 |
+----+------+------+
|  1 |    1 |    1 |
```

```
|  2 |    2 |    2 |
|  3 |    3 |    3 |
|  4 |    4 |    4 |
|  5 |    5 |    5 |
+----+------+------+
5 rows in set (0.01 sec)
```

```
ALTER TABLE t1 DROP COLUMN col1, DROP COLUMN col2;
```

```
ERROR 1105 (HY000): can't run multi schema change
```

```
SELECT * FROM t1;
```

```
+----+------+------+
| id | col1 | col2 |
+----+------+------+
|  1 |    1 |    1 |
|  2 |    2 |    2 |
|  3 |    3 |    3 |
|  4 |    4 |    4 |
|  5 |    5 |    5 |
+----+------+------+
5 rows in set (0.00 sec)
```

```
ALTER TABLE t1 DROP COLUMN col1;
```

```
Query OK, 0 rows affected (0.27 sec)
```

```
SELECT * FROM t1;
```

```
+----+------+
| id | col2 |
+----+------+
|  1 |    1 |
|  2 |    2 |
|  3 |    3 |
|  4 |    4 |
|  5 |    5 |
+----+------+
5 rows in set (0.00 sec)
```

14.11.2.45.3 MySQL 兼容性

- 目前不支持删除主键列或组合索引相关列。

14.11.2.45.4 另请参阅

- ADD COLUMN
- SHOW CREATE TABLE
- CREATE TABLE

14.11.2.46 DROP DATABASE

DROP DATABASE 语句用于永久删除指定的数据库，以及删除所有在 schema 中创建的表和视图。与被删数据库相关联的用户权限不受影响。

14.11.2.46.1 语法图

```
DropDatabaseStmt ::=
    'DROP' 'DATABASE' IfExists DBName

IfExists ::= ( 'IF' 'EXISTS' )?
```

14.11.2.46.2 示例

```
SHOW DATABASES;
```

```
+--------------------+
| Database           |
+--------------------+
| INFORMATION_SCHEMA |
| PERFORMANCE_SCHEMA |
| mysql              |
| test               |
+--------------------+
4 rows in set (0.00 sec)
```

```
DROP DATABASE test;
```

```
Query OK, 0 rows affected (0.25 sec)
```

```
SHOW DATABASES;
```

```
+--------------------+
| Database           |
+--------------------+
| INFORMATION_SCHEMA |
| PERFORMANCE_SCHEMA |
| mysql              |
+--------------------+
3 rows in set (0.00 sec)
```

14.11.2.46.3 MySQL 兼容性

DROP DATABASE 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.46.4 另请参阅

- CREATE DATABASE
- ALTER DATABASE

14.11.2.47 DROP INDEX

DROP INDEX 语句用于从指定的表中删除索引，并在 TiKV 中将空间标记为释放。

14.11.2.47.1 语法图

```
DropIndexStmt ::=
    "DROP" "INDEX" IfExists Identifier "ON" TableName IndexLockAndAlgorithmOpt

IfExists ::=
    ( 'IF' 'EXISTS' )?

IndexLockAndAlgorithmOpt ::=
    ( LockClause AlgorithmClause? | AlgorithmClause LockClause? )?
```

14.11.2.47.2 示例

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, c1 INT NOT NULL);
```

```
Query OK, 0 rows affected (0.10 sec)
```

```
INSERT INTO t1 (c1) VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 5 rows affected (0.02 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
EXPLAIN SELECT * FROM t1 WHERE c1 = 3;
```

```
+-------------------------+----------+-----------+--------------+--------------------------------+
    ↪
| id                      | estRows  | task      | access object | operator info
    ↪                   |
+-------------------------+----------+-----------+--------------+--------------------------------+
    ↪
| TableReader_7           | 10.00    | root      |               | data:Selection_6
    ↪               |
```

```
| └─Selection_6           | 10.00    | cop[tikv] |               | eq(test.t1.c1, 3)
    ↪              |
|   └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t1      | keep order:false, stats:
    ↪ pseudo |
+-------------------------+----------+-----------+---------------+--------------------------------+
    ↪
3 rows in set (0.00 sec)
```

```
CREATE INDEX c1 ON t1 (c1);
```

```
Query OK, 0 rows affected (0.30 sec)
```

```
EXPLAIN SELECT * FROM t1 WHERE c1 = 3;
```

```
+-----------------------+---------+-----------+-----------------------+-----------------------------------
    ↪
| id                    | estRows | task      | access object         | operator info
    ↪                               |
+-----------------------+---------+-----------+-----------------------+-----------------------------------
    ↪
| IndexReader_6         | 0.01    | root      |                       | index:IndexRangeScan_5
    ↪                       |
| └─IndexRangeScan_5    | 0.01    | cop[tikv] | table:t1, index:c1(c1) | range:[3,3], keep
    ↪ order:false, stats:pseudo |
+-----------------------+---------+-----------+-----------------------+-----------------------------------
    ↪
2 rows in set (0.00 sec)
```

```
DROP INDEX c1 ON t1;
```

```
Query OK, 0 rows affected (0.30 sec)
```

14.11.2.47.3 MySQL 兼容性

- 不支持删除 CLUSTERED 类型的 PRIMARY KEY。要了解关于 CLUSTERED 主键的详细信息，请参考聚簇索引。

14.11.2.47.4 另请参阅

- SHOW INDEX
- CREATE INDEX
- ADD INDEX
- RENAME INDEX
- ALTER INDEX

14.11.2.48 DROP PLACEMENT POLICY

`DROP PLACEMENT POLICY` 用于删除已创建的放置策略。

14.11.2.48.1 语法图

```
DropPolicyStmt ::=
    "DROP" "PLACEMENT" "POLICY" IfExists PolicyName

PolicyName ::=
    Identifier
```

14.11.2.48.2 示例

删除放置规则时，确保该策略未被任何表或分区引用，否则会删除失败。

```
CREATE PLACEMENT POLICY p1 FOLLOWERS=4;
CREATE TABLE t1 (a INT PRIMARY KEY) PLACEMENT POLICY=p1;
DROP PLACEMENT POLICY p1;  -- 该语句执行失败，因为放置规则 p1 被引用。

-- 查看引用放置规则的表和分区。
SELECT table_schema, table_name FROM information_schema.tables WHERE tidb_placement_policy_name='
    ↪ p1';
SELECT table_schema, table_name FROM information_schema.partitions WHERE
    ↪ tidb_placement_policy_name='p1';

ALTER TABLE t1 PLACEMENT POLICY=default;  -- 移除表 t1 上的默认放置规则。
DROP PLACEMENT POLICY p1;  -- 执行成功。
```

```
Query OK, 0 rows affected (0.10 sec)

Query OK, 0 rows affected (0.11 sec)

ERROR 8241 (HY000): Placement policy 'p1' is still in use

+--------------+------------+
| table_schema | table_name |
+--------------+------------+
| test         | t1         |
+--------------+------------+
1 row in set (0.00 sec)

Empty set (0.01 sec)

Query OK, 0 rows affected (0.08 sec)
```

```
Query OK, 0 rows affected (0.21 sec)
```

14.11.2.48.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.48.4 另请参阅

- Placement Rules in SQL
- SHOW PLACEMENT
- CREATE PLACEMENT POLICY
- ALTER PLACEMENT POLICY

14.11.2.49 DROP RESOURCE GROUP

`DROP RESOURCE GROUP` 语句用于在当前所选数据库中删除资源组。

14.11.2.49.1 语法图

```
DropResourceGroupStmt ::=
    "DROP" "RESOURCE" "GROUP" IfExists ResourceGroupName

IfExists ::=
    ('IF' 'EXISTS')?

ResourceGroupName ::=
    Identifier
```

> **注意：**
>
> - `DROP RESOURCE GROUP` 语句只能在全局变量`tidb_enable_resource_control` 设置为 `ON` 时才能执行。
> - `default` 资源组为系统保留的资源组，不支持删除。

14.11.2.49.2 示例

删除名为 `rg1` 的资源组：

```
DROP RESOURCE GROUP IF EXISTS rg1;
```

```
Query OK, 0 rows affected (0.22 sec)
```

```
CREATE RESOURCE GROUP IF NOT EXISTS rg1 RU_PER_SEC = 500 BURSTABLE;
```

```
Query OK, 0 rows affected (0.08 sec)
```

```
SELECT * FROM information_schema.resource_groups WHERE NAME ='rg1';
```

```
+------+------------+----------+-----------+
| NAME | RU_PER_SEC | PRIORITY | BURSTABLE |
+------+------------+----------+-----------+
| rg1  |        500 | MEDIUM   | YES       |
+------+------------+----------+-----------+
1 row in set (0.01 sec)
```

```
DROP RESOURCE GROUP IF EXISTS rg1;
```

```
Query OK, 1 rows affected (0.09 sec)
```

```
SELECT * FROM information_schema.resource_groups WHERE NAME ='rg1';
```

```
Empty set (0.00 sec)
```

14.11.2.49.3 MySQL 兼容性

MySQL 也支持 DROP RESOURCE GROUP，但 TiDB 不支持 FORCE 参数。

14.11.2.49.4 另请参阅

- ALTER RESOURCE GROUP
- CREATE RESOURCE GROUP
- RU

14.11.2.50 DROP ROLE

使用 DROP ROLE 语句可删除已用 CREATE ROLE 语句创建的角色。

14.11.2.50.1 语法图

```
DropRoleStmt ::=
    'DROP' 'ROLE' ( 'IF' 'EXISTS' )? RolenameList

RolenameList ::=
    Rolename ( ',' Rolename )*
```

14.11.2.50.2 示例

以 root 用户连接 TiDB：

```
mysql -h 127.0.0.1 -P 4000 -u root
```

创建新角色 analyticsteam 和新用户 jennifer：

```
CREATE ROLE analyticsteam;
Query OK, 0 rows affected (0.02 sec)

GRANT SELECT ON test.* TO analyticsteam;
Query OK, 0 rows affected (0.02 sec)

CREATE USER jennifer;
Query OK, 0 rows affected (0.01 sec)

GRANT analyticsteam TO jennifer;
Query OK, 0 rows affected (0.01 sec)
```

以 jennifer 用户连接 TiDB：

```
mysql -h 127.0.0.1 -P 4000 -u jennifer
```

需要注意的是，默认情况下，用户 jennifer 需要执行 SET ROLE analyticsteam 语句才能使用与角色相关联的权限：

```
SHOW GRANTS;
+-----------------------------------------------+
| Grants for User                               |
+-----------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'          |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+-----------------------------------------------+
2 rows in set (0.00 sec)

SHOW TABLES in test;
ERROR 1044 (42000): Access denied for user 'jennifer'@'%' to database 'test'
SET ROLE analyticsteam;
Query OK, 0 rows affected (0.00 sec)

SHOW GRANTS;
+-----------------------------------------------+
| Grants for User                               |
+-----------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'          |
| GRANT SELECT ON test.* TO 'jennifer'@'%'      |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
```

```
+-------------------------------------------+
3 rows in set (0.00 sec)

SHOW TABLES IN test;
+----------------+
| Tables_in_test |
+----------------+
| t1             |
+----------------+
1 row in set (0.00 sec)
```

以 root 用户连接 TiDB：

```
mysql -h 127.0.0.1 -P 4000 -u root
```

执行 SET DEFAULT ROLE 语句将用户 jennifer 与 analyticsteam 角色相关联：

```
SET DEFAULT ROLE analyticsteam TO jennifer;
Query OK, 0 rows affected (0.02 sec)
```

以 jennifer 用户连接 TiDB：

```
mysql -h 127.0.0.1 -P 4000 -u jennifer
```

此时 jennifer 用户无需执行 SET ROLE 语句就能拥有 analyticsteam 角色相关联的权限：

```
SHOW GRANTS;
+-------------------------------------------+
| Grants for User                           |
+-------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'      |
| GRANT SELECT ON test.* TO 'jennifer'@'%'  |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+-------------------------------------------+
3 rows in set (0.00 sec)

SHOW TABLES IN test;
+----------------+
| Tables_in_test |
+----------------+
| t1             |
+----------------+
1 row in set (0.00 sec)
```

以 root 用户连接 TiDB：

```
mysql -h 127.0.0.1 -P 4000 -u root
```

删除角色 analyticsteam：

```
DROP ROLE analyticsteam;
Query OK, 0 rows affected (0.02 sec)
```

jennifer 用户不再具有与 analyticsteam 关联的默认角色，或不能再将 analyticsteam 设为启用角色：

以 jennifer 用户连接 TiDB：

```
mysql -h 127.0.0.1 -P 4000 -u jennifer
```

查看 jennifer 用户的权限：

```
SHOW GRANTS;
+--------------------------------------+
| Grants for User                      |
+--------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%' |
+--------------------------------------+
1 row in set (0.00 sec)

SET ROLE analyticsteam;
ERROR 3530 (HY000): `analyticsteam`@`%` is is not granted to jennifer@%
```

14.11.2.50.3 MySQL 兼容性

DROP ROLE 语句与 MySQL 8.0 的角色功能完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.50.4 另请参阅

- CREATE ROLE
- GRANT <role>
- REVOKE <role>
- SET ROLE
- SET DEFAULT ROLE
- 基于角色的访问控制

14.11.2.51 DROP SEQUENCE

DROP SEQUENCE 语句用于删除序列对象。

14.11.2.51.1 语法图

```
DropSequenceStmt ::=
    'DROP' 'SEQUENCE' IfExists TableNameList

IfExists ::= ( 'IF' 'EXISTS' )?
```

```
TableNameList ::=
    TableName ( ',' TableName )*

TableName ::=
    Identifier ('.' Identifier)?
```

14.11.2.51.2 示例

```
DROP SEQUENCE seq;
```

```
Query OK, 0 rows affected (0.10 sec)
```

```
DROP SEQUENCE seq, seq2;
```

```
Query OK, 0 rows affected (0.03 sec)
```

14.11.2.51.3 MySQL 兼容性

该语句是 TiDB 的扩展，序列的实现借鉴自 MariaDB。

14.11.2.51.4 另请参阅

- CREATE SEQUENCE
- SHOW CREATE SEQUENCE

14.11.2.52 DROP STATS

`DROP STATS` 语句用于从当前所选定的数据库中删除选定表的统计信息。

14.11.2.52.1 语法图

```
DropStatsStmt ::=
    'DROP' 'STATS' TableNameList

TableNameList ::=
    TableName ( ',' TableName )*

TableName ::=
    Identifier ('.' Identifier)?
```

14.11.2.52.2 示例

```
CREATE TABLE t(a INT);
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
SHOW STATS_META WHERE db_name='test' and table_name='t';
```

```
+---------+------------+----------------+---------------------+--------------+-----------+
| Db_name | Table_name | Partition_name | Update_time         | Modify_count | Row_count |
+---------+------------+----------------+---------------------+--------------+-----------+
| test    | t          |                | 2020-05-25 20:34:33 |            0 |         0 |
+---------+------------+----------------+---------------------+--------------+-----------+
1 row in set (0.00 sec)
```

```
DROP STATS t;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
SHOW STATS_META WHERE db_name='test' and table_name='t';
```

```
Empty set (0.00 sec)
```

14.11.2.52.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.52.4 另请参阅

- 统计信息简介

14.11.2.53 DROP TABLE

`DROP TABLE` 语句用于从当前所选的数据库中删除表。如果表不存在则会报错，除非使用 `IF EXISTS` 修饰符。

14.11.2.53.1 语法图

```
DropTableStmt ::=
    'DROP' OptTemporary TableOrTables IfExists TableNameList RestrictOrCascadeOpt

OptTemporary ::=
    ( 'TEMPORARY' | ('GLOBAL' 'TEMPORARY') )?

TableOrTables ::=
    'TABLE'
```

```
|   'TABLES'

TableNameList ::=
    TableName ( ',' TableName )*
```

14.11.2.53.2 删除临时表

删除普通表和临时表的用法如下：

- `DROP TEMPORARY TABLE` 只能删除本地临时表
- `DROP GLOBAL TEMPORARY TABLE` 只能删除全局临时表
- `DROP TABLE` 可以删除普通表或临时表

14.11.2.53.3 示例

```
CREATE TABLE t1 (a INT);
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
DROP TABLE t1;
```

```
Query OK, 0 rows affected (0.22 sec)
```

```
DROP TABLE table_not_exists;
```

```
ERROR 1051 (42S02): Unknown table 'test.table_not_exists'
```

```
DROP TABLE IF EXISTS table_not_exists;
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
CREATE VIEW v1 AS SELECT 1;
```

```
Query OK, 0 rows affected (0.10 sec)
```

```
DROP TABLE v1;
```

```
Query OK, 0 rows affected (0.23 sec)
```

14.11.2.53.4 MySQL 兼容性

- 在尝试删除不存在的表时，使用 `IF EXISTS` 删除表不会返回警告。Issue #7867
- 目前 `RESTRICT` 和 `CASCADE` 仅在语法上支持。

14.11.2.53.5 另请参阅

- CREATE TABLE
- SHOW CREATE TABLE
- SHOW TABLES

14.11.2.54 DROP USER

DROP USER 语句用于从 TiDB 系统数据库中删除用户。如果用户不存在，使用关键词 IF EXISTS 可避免出现警告。执行 DROP USER 语句需要拥有 CREATE USER 权限。

14.11.2.54.1 语法图

```
DropUserStmt ::=
    'DROP' 'USER' ( 'IF' 'EXISTS' )? UsernameList

Username ::=
    StringName ('@' StringName | singleAtIdentifier)? | 'CURRENT_USER' OptionalBraces
```

14.11.2.54.2 示例

```
DROP USER 'idontexist';
```

```
ERROR 1396 (HY000): Operation DROP USER failed for idontexist@%
```

```
DROP USER IF EXISTS 'idontexist';
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
CREATE USER 'newuser' IDENTIFIED BY 'mypassword';
```

```
Query OK, 1 row affected (0.02 sec)
```

```
GRANT ALL ON test.* TO 'newuser';
```

```
Query OK, 0 rows affected (0.03 sec)
```

```
SHOW GRANTS FOR 'newuser';
```

```
+-----------------------------------------------+
| Grants for newuser@%                          |
+-----------------------------------------------+
| GRANT USAGE ON *.* TO 'newuser'@'%'           |
| GRANT ALL PRIVILEGES ON test.* TO 'newuser'@'%' |
+-----------------------------------------------+
2 rows in set (0.00 sec)
```

```
REVOKE ALL ON test.* FROM 'newuser';
```

```
Query OK, 0 rows affected (0.03 sec)
```

```
SHOW GRANTS FOR 'newuser';
```

```
+-------------------------------------+
| Grants for newuser@%                |
+-------------------------------------+
| GRANT USAGE ON *.* TO 'newuser'@'%' |
+-------------------------------------+
1 row in set (0.00 sec)
```

```
DROP USER 'newuser';
```

```
Query OK, 0 rows affected (0.14 sec)
```

```
SHOW GRANTS FOR 'newuser';
```

```
ERROR 1141 (42000): There is no such grant defined for user 'newuser' on host '%'
```

14.11.2.54.3 MySQL 兼容性

- 在 TiDB 中删除不存在的用户时，使用 `IF EXISTS` 可避免出现警告。Issue #10196

14.11.2.54.4 另请参阅

- CREATE USER
- ALTER USER
- SHOW CREATE USER
- Privilege Management

14.11.2.55 DROP VIEW

`DROP VIEW` 语句用于从当前所选定的数据库中删除视图对象。视图所引用的基表不受影响。

14.11.2.55.1 语法图

```
DropViewStmt ::=
    'DROP' 'VIEW' ( 'IF' 'EXISTS' )? TableNameList RestrictOrCascadeOpt

TableNameList ::=
    TableName ( ',' TableName )*
```

```
TableName ::=
    Identifier ('.' Identifier)?
```

14.11.2.55.2 示例

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, c1 INT NOT NULL);
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
INSERT INTO t1 (c1) VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 5 rows affected (0.03 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
CREATE VIEW v1 AS SELECT * FROM t1 WHERE c1 > 2;
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
SELECT * FROM t1;
```

```
+----+----+
| id | c1 |
+----+----+
|  1 |  1 |
|  2 |  2 |
|  3 |  3 |
|  4 |  4 |
|  5 |  5 |
+----+----+
5 rows in set (0.00 sec)
```

```
SELECT * FROM v1;
```

```
+----+----+
| id | c1 |
+----+----+
|  3 |  3 |
|  4 |  4 |
|  5 |  5 |
+----+----+
3 rows in set (0.00 sec)
```

```
DROP VIEW v1;
```

```
Query OK, 0 rows affected (0.23 sec)
```

```
SELECT * FROM t1;
```

```
+----+----+
| id | c1 |
+----+----+
|  1 |  1 |
|  2 |  2 |
|  3 |  3 |
|  4 |  4 |
|  5 |  5 |
+----+----+
5 rows in set (0.00 sec)
```

14.11.2.55.3 MySQL 兼容性

DROP VIEW 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.55.4 See also

- CREATE VIEW
- DROP TABLE

14.11.2.56 EXECUTE

EXECUTE 语句为服务器端预处理语句提供 SQL 接口。

14.11.2.56.1 语法图

```
ExecuteStmt ::=
    'EXECUTE' Identifier ( 'USING' UserVariable ( ',' UserVariable )* )?
```

14.11.2.56.2 示例

```
PREPARE mystmt FROM 'SELECT ? as num FROM DUAL';
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
SET @number = 5;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
EXECUTE mystmt USING @number;
```

```
+------+
| num  |
+------+
| 5    |
+------+
1 row in set (0.00 sec)
```

```
DEALLOCATE PREPARE mystmt;
```

```
Query OK, 0 rows affected (0.00 sec)
```

14.11.2.56.3 MySQL 兼容性

EXECUTE 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.56.4 另请参阅

- PREPARE
- DEALLOCATE

14.11.2.57 EXPLAIN ANALYZE

EXPLAIN ANALYZE 语句的工作方式类似于 EXPLAIN，主要区别在于前者实际上会执行语句。这样可以将查询计划中的估计值与执行时所遇到的实际值进行比较。如果估计值与实际值显著不同，那么应考虑在受影响的表上运行 ANALYZE TABLE。

> **注意：**
>
> 在使用 EXPLAIN ANALYZE 执行 DML 语句时，数据的修改操作会被正常执行。但目前 DML 语句还无法展示执行计划。

14.11.2.57.1 语法图

```
ExplainSym ::=
    'EXPLAIN'
|   'DESCRIBE'
|    'DESC'

ExplainStmt ::=
```

```
    ExplainSym ( TableName ColumnName? | 'ANALYZE'? ExplainableStmt | 'FOR' 'CONNECTION' NUM | '
        ↪ FORMAT' '=' ( stringLit | ExplainFormatType ) ( 'FOR' 'CONNECTION' NUM |
        ↪ ExplainableStmt ) )

ExplainableStmt ::=
    SelectStmt
|   DeleteFromStmt
|   UpdateStmt
|   InsertIntoStmt
|   ReplaceIntoStmt
|   UnionStmt
```

14.11.2.57.2 EXPLAIN ANALYZE 输出格式

和 EXPLAIN 不同，EXPLAIN ANALYZE 会执行对应的 SQL 语句，记录其运行时信息，和执行计划一并返回出来。因此，可以将 EXPLAIN ANALYZE 视为 EXPLAIN 语句的扩展。EXPLAIN ANALYZE 语句的返回结果相比 EXPLAIN，增加了 actRows，execution info，memory，disk 这几列信息：

属性名	含义
actRows	算子实际输出的数据条数。
execution info	算子的实际执行信息。time 表示从进入算子到离开算子的全部 wall time，包括所有子算子操作的全部执行时间。如果该算子被父算子多次调用 (loops)，这个时间就是累积的时间。loops 是当前算子被父算子调用的次数。
memory	算子占用内存空间的大小。
disk	算子占用磁盘空间的大小。

14.11.2.57.3 示例

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, c1 INT NOT NULL);
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
INSERT INTO t1 (c1) VALUES (1), (2), (3);
```

```
Query OK, 3 rows affected (0.02 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
EXPLAIN ANALYZE SELECT * FROM t1 WHERE id = 1;
```

```
+-------------+---------+---------+------+---------------+-------------------------------------------------------------------------
    ↪
```

```
| id          | estRows | actRows | task | access object | execution info
    ↪                                                  | operator info | memory | disk |
+-------------+---------+---------+------+---------------+-------------------------------------------------------------------
    ↪
| Point_Get_1 | 1.00    | 1       | root | table:t1      | time:757.205µs, loops:2, Get:{num_rpc
    ↪ :1, total_time:697.051µs} | handle:1      | N/A    | N/A  |
+-------------+---------+---------+------+---------------+-------------------------------------------------------------------
    ↪
1 row in set (0.01 sec)
```

```
EXPLAIN ANALYZE SELECT * FROM t1;
```

```
+------------------+----------+---------+-----------+--------------+--------------------------------------------------------------
    ↪
| id               | estRows  | actRows | task      | access object | execution info
    ↪
    ↪ | operator info                  | memory    | disk |
+------------------+----------+---------+-----------+--------------+--------------------------------------------------------------
    ↪
| TableReader_5     | 10000.00 | 3       | root      |               | time:278.2µs, loops:2,
    ↪ cop_task: {num: 1, max: 437.6µs, proc_keys: 3, rpc_num: 1, rpc_time: 423.9µs,
    ↪ copr_cache_hit_ratio: 0.00}
    ↪
    ↪ | data:TableFullScan_4           | 251 Bytes | N/A  |
| └─TableFullScan_4 | 10000.00 | 3       | cop[tikv] | table:t1      | tikv_task:{time:0s, loops
    ↪ :1}, scan_detail: {total_process_keys: 3, total_process_keys_size: 111, total_keys: 4,
    ↪ rocksdb: {delete_skipped_count: 0, key_skipped_count: 3, block: {cache_hit_count: 0,
    ↪ read_count: 0, read_byte: 0 Bytes}}} | keep order:false, stats:pseudo | N/A       | N/A
    ↪ |
+------------------+----------+---------+-----------+--------------+--------------------------------------------------------------
    ↪
2 rows in set (0.00 sec)
```

14.11.2.57.4 算子执行信息介绍

`execution info` 信息除了基本的 `time` 和 `loop` 信息外，还包含算子特有的执行信息，主要包含了该算子发送 RPC 请求的耗时信息以及其他步骤的耗时。

Point_Get

`Point_Get` 算子可能包含以下执行信息：

- `Get:{num_rpc:1, total_time:697.051µs}`：向 TiKV 发送 `Get` 类型的 RPC 请求的数量 (`num_rpc`) 和所有 RPC 请求的总耗时 (`total_time`)。
- `ResolveLock:{num_rpc:1, total_time:12.117495ms}`：读数据遇到锁后，进行 resolve lock 的时间。一般在读写冲突的场景下会出现。

- `regionMiss_backoff:{num:11, total_time:2010 ms},tikvRPC_backoff:{num:11, total_time:10691 ms}`：RPC 请求失败后，会在等待 backoff 的时间后重试，包括了 backoff 的类型（如 regionMiss，tikvRPC），backoff 等待的总时间 (total_time) 和 backoff 的总次数 (num)。

Batch_Point_Get

`Batch_Point_get` 算子的执行信息和 `Point_Get` 算子类似，不过 `Batch_Point_Get` 一般向 TiKV 发送 `BatchGet` 类型的 RPC 请求来读取数据。

`BatchGet:{num_rpc:2, total_time:83.13µs}`：向 TiKV 发送 `BatchGet` 类型的 RPC 请求的数量 (num_rpc) 和所有 RPC 请求的总耗时 (total_time)。

TableReader

`TableReader` 算子可能包含以下执行信息：

```
cop_task: {num: 6, max: 1.07587ms, min: 844.312µs, avg: 919.601µs, p95: 1.07587ms, max_proc_keys:
    ↪  16, p95_proc_keys: 16, tot_proc: 1ms, tot_wait: 1ms, rpc_num: 6, rpc_time: 5.313996ms,
    ↪ copr_cache_hit_ratio: 0.00}
```

- `cop_task`：包含 cop task 的相关信息，如：
  - `num`：cop task 的数量
  - `max,min,avg,p95`：所有 cop task 中执行时间的最大值，最小值，平均值和 P95 值。
  - `max_proc_keys, p95_proc_keys`：所有 cop task 中 tikv 扫描 kv 数据的最大值，P95 值，如果 max 和 p95 的值差距很大，说明数据分布不太均匀。
  - `rpc_num, rpc_time`：向 TiKV 发送 Cop 类型的 RPC 请求总数量和总时间。
  - `copr_cache_hit_ratio`：cop task 请求的 Coprocessor Cache 缓存命中率。Coprocessor Cache 配置。
- `backoff`：包含不同类型的 backoff 以及等待总耗时。

Insert

`Insert` 算子可能包含以下执行信息：

```
prepare:109.616µs, check_insert:{total_time:1.431678ms, mem_insert_time:667.878µs, prefetch:763.8
    ↪ µs, rpc:{BatchGet:{num_rpc:1, total_time:699.166µs},Get:{num_rpc:1, total_time:378.276µs
    ↪ }}}
```

- `prepare`：准备写入前的耗时，包括表达式，默认值相关的计算等。
- `check_insert`：这个信息一般出现在 `insert ignore` 和 `insert on duplicate` 语句中，包含冲突检查和写入 TiDB 事务缓存的耗时。注意，这个耗时不包含事务提交的耗时。具体包含以下信息：
  - `total_time`：`check_insert` 步骤的总耗时。
  - `mem_insert_time`：将数据写入 TiDB 事务缓存的耗时。
  - `prefetch`：从 TiKV 中获取需要检查冲突的数据的耗时，该步骤主要是向 TiKV 发送 `BatchGet` 类型的 RPC 请求的获取数据。
  - `rpc`：向 TiKV 发送 RPC 请求的总耗时，一般包含 `BatchGet` 和 `Get` 两种类型的 RPC 耗时，其中：
    * `BatchGet` 请求是 `prefetch` 步骤发送的 RPC 请求。

    * Get 请求是 insert on duplicate 语句在执行 duplicate update 时发送的 RPC 请求。

- backoff：包含不同类型的 backoff 以及等待总耗时。

IndexJoin

IndexJoin 算子有 1 个 outer worker 和 N 个 inner worker 并行执行，其 join 结果的顺序和 outer table 的顺序一致，具体执行流程如下：

1. Outer worker 读取 N 行 outer table 的数据，然后包装成一个 task 发送给 result channel 和 inner worker channel。
2. Inner worker 从 inner worker channel 里面接收 task，然后根据 task 生成需要读取 inner table 的 key ranges 范围，然后读取相应范围的 inner table 行数据，并生成一个 inner table row 的 hash table。
3. IndexJoin 的主线程从 result channel 中接收 task，然后等待 inner worker 执行完这个 task。
4. IndexJoin 的主线程用 outer table rows 和 inner table rows 的 hash table 做 join。

IndexJoin 算子包含以下执行信息：

```
inner:{total:4.297515932s, concurrency:5, task:17, construct:97.96291ms, fetch:4.164310088s,
    ↪ build:35.219574ms}, probe:53.574945ms
```

- inner：inner worker 的执行信息，具体如下：
  - total：inner worker 的总耗时。
  - concurrency：inner worker 的数量。
  - task：inner worker 处理 task 的总数量。
  - construct：inner worker 读取 task 对应的 inner table rows 之前的准备时间。
  - fetch：inner worker 读取 inner table rows 的总耗时。
  - build: inner worker 构造 inner table rows 对应的 hash table 的总耗时。
- probe：IndexJoin 主线程用 outer table rows 和 inner table rows 的 hash table 做 join 的总耗时。

IndexHashJoin

IndexHashJoin 算子和 IndexJoin 算子执行流程类似，也有 1 个 outer worker 和 N 个 inner worker 并行执行，但是其 join 结果的顺序是不和 outer table 一致。具体执行流程如下：

1. Outer worker 读取 N 行 out table 的数据，然后包装成一个 task 发送给 inner worker channel。
2. Inner worker 从 inner worker channel 里面接收 task，然后做以下三件事情，其中步骤 a 和 b 是并行执行。
   a. 用 outer table rows 生成一个 hash table。
   b. 根据 task 生成 key 的范围后，读取 inner table 相应范围的行数据。
   c. 用 inner table rows 和 outer table rows 的 hash table 做 join，然后把 join 结果发送给 result channel。
3. IndexHashJoin 的主线程从 result channel 中接收 join 结果。

IndexHashJoin 算子包含以下执行信息：

```
inner:{total:4.429220003s, concurrency:5, task:17, construct:96.207725ms, fetch:4.239324006s,
    ↪ build:24.567801ms, join:93.607362ms}
```

- `inner`：inner worker 的执行信息，具体如下：
  - `total`：inner worker 的总耗时。
  - `concurrency`：inner worker 的数量。
  - `task`：inner worker 处理 task 的总数量。
  - `construct`：inner worker 读取 task 对应的 inner table rows 之前的准备时间。
  - `fetch`：inner worker 读取 inner table rows 的总耗时。
  - build: inner worker 构造 outer table rows 对应的 hash table 的总耗时。
  - `join`: inner worker 用 inner table rows 和 outer table rows 的 hash table 做 join 的总耗时。

HashJoin

`HashJoin` 算子有一个 inner worker，一个 outer worker 和 N 个 join worker，其具体执行逻辑如下：

1. inner worker 读取 inner table rows 并构造 hash table。
2. outer worker 读取 outer table rows，然后包装成 task 发送给 join worker。
3. 等待第 1 步的 hash table 构造完成。
4. join worker 用 task 里面的 outer table rows 和 hash table 做 join，然后把 join 结果发送给 result channel。
5. `HashJoin` 的主线程从 result channel 中接收 join 结果。

`HashJoin` 算子包含以下执行信息：

```
build_hash_table:{total:146.071334ms, fetch:110.338509ms, build:35.732825ms}, probe:{concurrency
    ↪ :5, total:857.162518ms, max:171.48271ms, probe:125.341665ms, fetch:731.820853ms}
```

- `build_hash_table`: 读取 inner table 的数据并构造 hash table 的执行信息：
  - `total`：总耗时。
  - `fetch`：读取 inner table 数据的总耗时。
  - `build`：构造 hash table 的总耗时。
- `probe`: join worker 的执行信息：
  - `concurrency`：join worker 的数量。
  - `total`：所有 join worker 执行的总耗时。
  - `max`：单个 join worker 执行的最大耗时。
  - `probe`: 用 outer table rows 和 hash table 做 join 的总耗时。
  - `fetch`：join worker 等待读取 outer table rows 数据的总耗时。

TableFullScan (TiFlash)

在 TiFlash 节点上执行的 `TableFullScan` 算子包含以下执行信息：

```
tiflash_scan: {
  dtfile: {
    total_scanned_packs: 2,
    total_skipped_packs: 1,
    total_scanned_rows: 16000,
    total_skipped_rows: 8192,
```

```
    total_rough_set_index_load_time: 2ms,
    total_read_time: 20ms
  },
  total_create_snapshot_time: 1ms
}
```

- dtfile：扫表过程中与 DTFile （即 DeltaTree File ）相关的信息；这基本反映了 TiFlash 在 Stable 层数据的读取情况。
  - total_scanned_packs：DTFile 内累计读取的 Pack 的数量；Pack 是 TiFlash DTFile 读取的最小粒度，默认情况下每 8192 行构成一个 Pack。
  - total_skipped_packs：DTFile 内累计跳过的 Pack 的数量；Pack 会由于 WHERE 条件命中粗糙索引或主键范围过滤而被跳过。
  - total_scanned_rows：DTFile 内累计读取的行数；若存在 MVCC 带来的多版本更新或删除，则每个版本独立计数。
  - total_skipped_rows：DTFile 内累计跳过的行数。
  - total_rs_index_load_time：读取 DTFile 粗糙索引的累计耗时。
  - total_read_time：读取 DTFile 数据的累计耗时。
- total_create_snapshot_time：扫表过程中创建快照的总耗时。

lock_keys 执行信息

在悲观事务中执行 DML 语句时，算子的执行信息还可能包含 lock_keys 的执行信息，示例如下：

```
lock_keys: {time:94.096168ms, region:6, keys:8, lock_rpc:274.503214ms, rpc_count:6}
```

- time：执行 lock_keys 操作的总耗时。
- region：执行 lock_keys 操作涉及的 Region 数量。
- keys：需要 Lock 的 Key 的数量。
- lock_rpc：向 TiKV 发送 Lock 类型的 RPC 总耗时。因为可以并行发送多个 RPC 请求，所以总 RPC 耗时可能比 lock_keys 操作总耗时大。
- rpc_count：向 TiKV 发送 Lock 类型的 RPC 总数量。

commit_txn 执行信息

在 autocommit=1 的事务中执行写入类型的 DML 语句时，算子的执行信息还会包括事务提交的耗时信息，示例如下：

```
commit_txn: {prewrite:48.564544ms, wait_prewrite_binlog:47.821579, get_commit_ts:4.277455ms,
    ↪ commit:50.431774ms, region_num:7, write_keys:16, write_byte:536}
```

- prewrite：事务 2PC 提交阶段中 prewrite 阶段的耗时。
- wait_prewrite_binlog:：等待写 prewrite Binlog 的耗时。
- get_commit_ts：获取事务提交时间戳的耗时。
- commit：事务 2PC 提交阶段中，commit 阶段的耗时。

- write_keys：事务中写入 key 的数量。
- write_byte：事务中写入 key-value 的总字节数量，单位是 byte。

RU (Request Unit) 消耗

Request Unit (RU) 是资源管控对系统资源统一抽象的计量单位。执行计划顶层算子的 execution info 会显示 SQL 整体的 RU 消耗。

```
RU:273.842670
```

> 注意：
>
> 该值仅表示本次执行的实际 RU 消耗。由于受缓存的影响（比如下推计算结果缓存），同一个 SQL 在每次执行时消耗的 RU 可能会不同。

其它常见执行信息

Coprocessor 算子通常包含 cop_task 和 tikv_task 两部分执行时间信息。前者是 TiDB 端记录的时间，从发出请求到接收回复；后者是 TiKV Coprocessor 算子自己记录的时间。两者相差较大可能说明在等待、gRPC 或网络上耗时较长。

14.11.2.57.5 MySQL 兼容性

EXPLAIN ANALYZE 是 MySQL 8.0 的功能，但该语句在 MySQL 中的输出格式和可能的执行计划都与 TiDB 有较大差异。

14.11.2.57.6 另请参阅

- Understanding the Query Execution Plan
- EXPLAIN
- ANALYZE TABLE
- TRACE

14.11.2.58 EXPLAIN

EXPLAIN 语句仅用于显示查询的执行计划，而不执行查询。EXPLAIN ANALYZE 可执行查询，补充 EXPLAIN 语句。如果 EXPLAIN 的输出与预期结果不匹配，可考虑在查询的每个表上执行 ANALYZE TABLE。

语句 DESC 和 DESCRIBE 是 EXPLAIN 的别名。EXPLAIN <tableName> 的替代用法记录在SHOW [FULL] COLUMNS FROM 下。

TiDB 支持 EXPLAIN [options] FOR CONNECTION connection_id，但与 MySQL 的 EXPLAIN FOR 有一些区别，请参见EXPLAIN FOR CONNECTION。

14.11.2.58.1 语法图

```
ExplainSym ::=
    'EXPLAIN'
|   'DESCRIBE'
|   'DESC'

ExplainStmt ::=
    ExplainSym ( TableName ColumnName? | 'ANALYZE'? ExplainableStmt | 'FOR' 'CONNECTION' NUM | '
        ↪ FORMAT' '=' ( stringLit | ExplainFormatType ) ( 'FOR' 'CONNECTION' NUM |
        ↪ ExplainableStmt ) )

ExplainableStmt ::=
    SelectStmt
|   DeleteFromStmt
|   UpdateStmt
|   InsertIntoStmt
|   ReplaceIntoStmt
|   UnionStmt
```

14.11.2.58.2 EXPLAIN 输出格式

> **注意：**
>
> 使用 MySQL 客户端连接到 TiDB 时，为避免输出结果在终端中换行，可先执行 `pager less -S` 命令。执行命令后，新的 EXPLAIN 的输出结果不再换行，可按右箭头 → 键水平滚动阅读输出结果。

> **注意：**
>
> 在执行计划返回结果中，自 v6.4.0 版本起，特定算子（即 IndexJoin 和 Apply 算子的 Probe 端所有子节点）的 estRows 字段意义与 v6.4.0 之前的有所不同。细节请参考TiDB 执行计划概览。

目前 TiDB 的 EXPLAIN 会输出 5 列，分别是：id，estRows，task，access object，operator info。执行计划中每个算子都由这 5 列属性来描述，EXPLAIN 结果中每一行描述一个算子。每个属性的具体含义如下：

属性名	含义
id	算子的 ID，是算子在整个执行计划中唯一的标识。在 TiDB 2.1 中，ID 会格式化地显示算子的树状结构。数据从孩子结点流向父亲结点，每个算子的父亲结点有且仅有一个。
estRows	算子预计将会输出的数据条数，基于统计信息以及算子的执行逻辑估算而来。在 4.0 之前叫 count。

属性名	含义
task	算子属于的 task 种类。目前的执行计划分成为两种 task，一种叫 root task，在 tidb-server 上执行，一种叫 cop task，在 TiKV 或者 TiFlash 上并行执行。当前的执行计划在 task 级别的拓扑关系是一个 root task 后面可以跟许多 cop task，root task 使用 cop task 的输出结果作为输入。cop task 中执行的也即是 TiDB 下推到 TiKV 或者 TiFlash 上的任务，每个 cop task 分散在 TiKV 或者 TiFlash 集群中，由多个进程共同执行。
access object	算子所访问的数据项信息。包括表 `table`，表分区 `partition` 以及使用的索引 `index`（如果有）。只有直接访问数据的算子才拥有这些信息。
operator info	算子的其它信息。各个算子的 operator info 各有不同，可参考下面的示例解读。

14.11.2.58.3 示例

```
EXPLAIN SELECT 1;
```

```
+------------------+---------+------+---------------+---------------+
| id               | estRows | task | access object | operator info |
+------------------+---------+------+---------------+---------------+
| Projection_3     | 1.00    | root |               | 1->Column#1   |
| └─TableDual_4    | 1.00    | root |               | rows:1        |
+------------------+---------+------+---------------+---------------+
2 rows in set (0.00 sec)
```

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, c1 INT NOT NULL);
```

```
Query OK, 0 rows affected (0.10 sec)
```

```
INSERT INTO t1 (c1) VALUES (1), (2), (3);
```

```
Query OK, 3 rows affected (0.02 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
EXPLAIN SELECT * FROM t1 WHERE id = 1;
```

```
+-------------+---------+------+---------------+---------------+
| id          | estRows | task | access object | operator info |
+-------------+---------+------+---------------+---------------+
| Point_Get_1 | 1.00    | root | table:t1      | handle:1      |
+-------------+---------+------+---------------+---------------+
1 row in set (0.00 sec)
```

```
DESC SELECT * FROM t1 WHERE id = 1;
```

```
+-------------+---------+------+---------------+---------------+
| id          | estRows | task | access object | operator info |
+-------------+---------+------+---------------+---------------+
| Point_Get_1 | 1.00    | root | table:t1      | handle:1      |
+-------------+---------+------+---------------+---------------+
1 row in set (0.00 sec)
```

```
DESCRIBE SELECT * FROM t1 WHERE id = 1;
```

```
+-------------+---------+------+---------------+---------------+
| id          | estRows | task | access object | operator info |
+-------------+---------+------+---------------+---------------+
| Point_Get_1 | 1.00    | root | table:t1      | handle:1      |
+-------------+---------+------+---------------+---------------+
1 row in set (0.00 sec)
```

```
EXPLAIN INSERT INTO t1 (c1) VALUES (4);
```

```
+----------+---------+------+---------------+---------------+
| id       | estRows | task | access object | operator info |
+----------+---------+------+---------------+---------------+
| Insert_1 | N/A     | root |               | N/A           |
+----------+---------+------+---------------+---------------+
1 row in set (0.00 sec)
```

```
EXPLAIN UPDATE t1 SET c1=5 WHERE c1=3;
```

```
+--------------------------+---------+-----------+---------------+--------------------------------+
    ↪
| id                       | estRows | task      | access object | operator info
    ↪                  |
+--------------------------+---------+-----------+---------------+--------------------------------+
    ↪
| Update_4                 | N/A     | root      |               | N/A
    ↪                        |
| └─TableReader_8           | 0.00    | root      |               | data:Selection_7
    ↪               |
|   └─Selection_7           | 0.00    | cop[tikv] |               | eq(test.t1.c1, 3)
    ↪               |
|     └─TableFullScan_6     | 3.00    | cop[tikv] | table:t1      | keep order:false, stats:
    ↪ pseudo |
+--------------------------+---------+-----------+---------------+--------------------------------+
    ↪
4 rows in set (0.00 sec)
```

```
EXPLAIN DELETE FROM t1 WHERE c1=3;
```

```
+--------------------------+---------+-----------+---------------+--------------------------------+
    ↪
| id                       | estRows | task      | access object | operator info
    ↪                  |
+--------------------------+---------+-----------+---------------+--------------------------------+
    ↪
| Delete_4                 | N/A     | root      |               | N/A
    ↪                            |
| └─TableReader_8           | 0.00    | root      |               | data:Selection_7
    ↪               |
|   └─Selection_7           | 0.00    | cop[tikv] |               | eq(test.t1.c1, 3)
    ↪              |
|     └─TableFullScan_6     | 3.00    | cop[tikv] | table:t1      | keep order:false, stats:
    ↪ pseudo |
+--------------------------+---------+-----------+---------------+--------------------------------+
    ↪
4 rows in set (0.01 sec)
```

在 EXPLAIN 语句中，你可以使用 FORMAT = "xxx" 语法指定输出内容的格式。目前支持的输出格式如下：

FORMAT	作用
未指定	未指定 FORMAT 时，默认输出格式为 row
row	EXPLAIN 语句将以表格格式输出结果。更多信息，可参阅TiDB 执行计划概览
brief	EXPLAIN 语句输出结果中的算子 ID 将被简化，较之未指定 FORMAT 时输出结果的算子 ID 更为简化
dot	EXPLAIN 语句将输出 DOT 格式的执行计划，可以通过 dot 程序（在 graphviz 包中）生成 PNG 文件
tidb_json	EXPLAIN 语句将输出 JSON 格式的执行计划，算子信息存放在一个 JSON 数组中

在 EXPLAIN 中指定 FORMAT = "brief" 时，示例如下:

```
EXPLAIN FORMAT = "brief" DELETE FROM t1 WHERE c1=3;
```

```
+------------------------+---------+-----------+---------------+--------------------------------+
    ↪
| id                     | estRows | task      | access object | operator info
    ↪ |
+------------------------+---------+-----------+---------------+--------------------------------+
    ↪
| Delete                 | N/A     | root      |               | N/A
    ↪ |
| └─TableReader           | 0.00    | root      |               | data:Selection
    ↪                |
```

```
|   └─Selection           | 0.00    | cop[tikv] |               | eq(test.t1.c1, 3)
    ↪              |
|     └─TableFullScan     | 3.00    | cop[tikv] | table:t1      | keep order:false, stats:pseudo
    ↪  |
+-------------------------+---------+-----------+---------------+--------------------------------+
    ↪
4 rows in set (0.001 sec)
```

除 MySQL 标准结果格式外，TiDB 还支持 DotGraph，需要在 EXPLAIN 中指定 FORMAT = "dot"，示例如下：

```
create table t(a bigint, b bigint);
explain format = "dot" select A.a, B.b from t A join t B on A.a > B.b where A.a < 10;
```

```
+-----------------------------------------------------------------------------------------------------------------------
    ↪
| dot contents
    ↪
    ↪ |
+-----------------------------------------------------------------------------------------------------------------------
    ↪
|
digraph Projection_8 {
subgraph cluster8{
node [style=filled, color=lightgrey]
color=black
label = "root"
"Projection_8" -> "HashJoin_9"
"HashJoin_9" -> "TableReader_13"
"HashJoin_9" -> "Selection_14"
"Selection_14" -> "TableReader_17"
}
subgraph cluster12{
node [style=filled, color=lightgrey]
color=black
label = "cop"
"Selection_12" -> "TableFullScan_11"
}
subgraph cluster16{
node [style=filled, color=lightgrey]
color=black
label = "cop"
"Selection_16" -> "TableFullScan_15"
}
"TableReader_13" -> "Selection_12"
"TableReader_17" -> "Selection_16"
}
```

```
 |
+-----------------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

如果你的计算机上安装了 dot 程序，可使用以下方法生成 PNG 文件：

```
dot xx.dot -T png -O
```

The xx.dot is the result returned by the above statement.

如果你的计算机上未安装 dot 程序，可将结果复制到本网站以获取树形图：

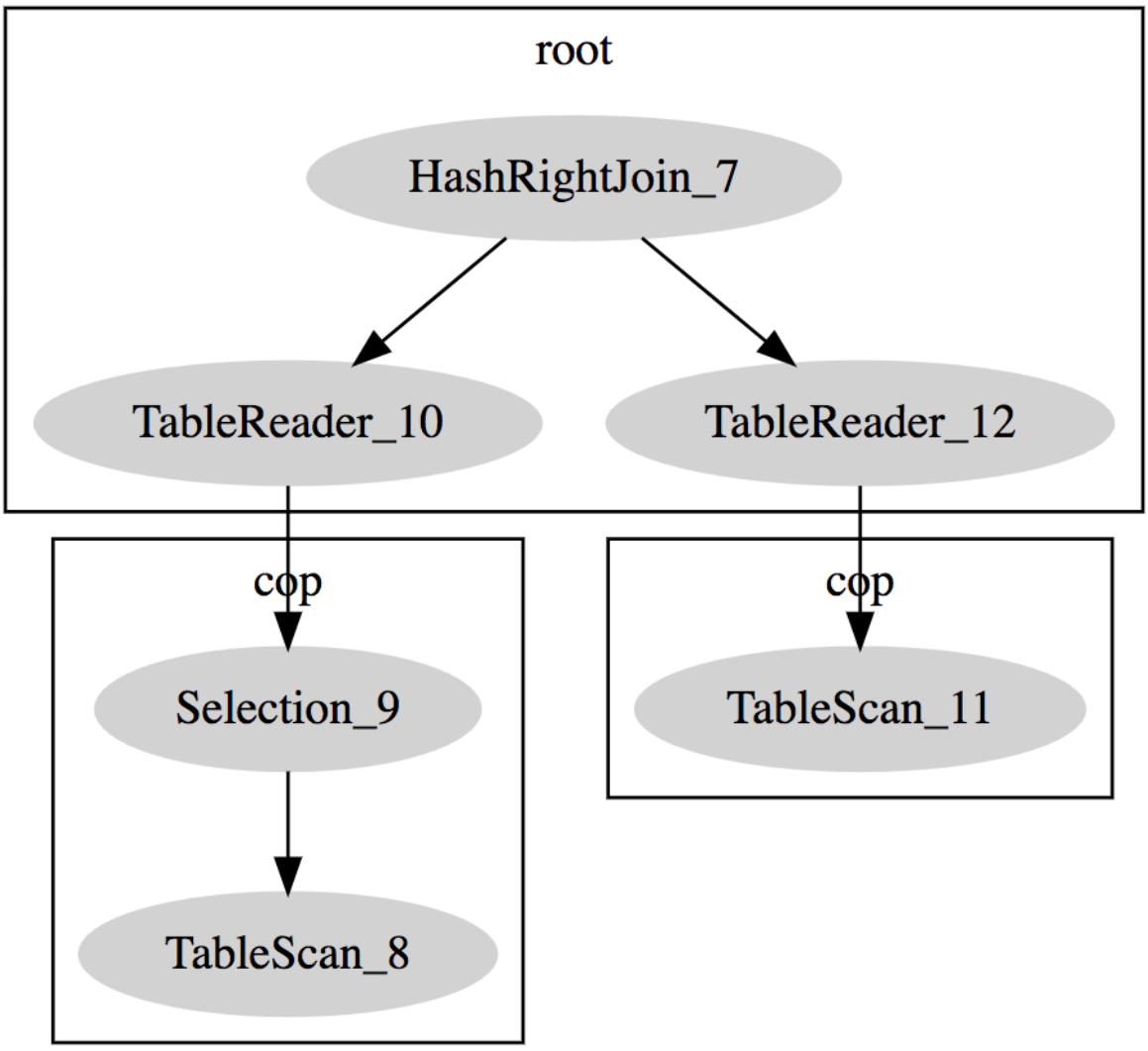


图 355: Explain Dot

在 EXPLAIN 中指定 FORMAT = "tidb_json" 时，示例如下：

```
CREATE TABLE t(id int primary key, a int, b int, key(a));
EXPLAIN FORMAT = "tidb_json" SELECT id FROM t WHERE a = 1;
```

```
+-----------------------------------------------------------------------------------------------------------------
    ↪
| TiDB_JSON
    ↪
    ↪ |
+-----------------------------------------------------------------------------------------------------------------
    ↪
| [
    {
        "id": "Projection_4",
        "estRows": "10.00",
        "taskType": "root",
        "operatorInfo": "test.t.id",
        "subOperators": [
            {
                "id": "IndexReader_6",
                "estRows": "10.00",
                "taskType": "root",
                "operatorInfo": "index:IndexRangeScan_5",
                "subOperators": [
                    {
                        "id": "IndexRangeScan_5",
                        "estRows": "10.00",
                        "taskType": "cop[tikv]",
                        "accessObject": "table:t, index:a(a)",
                        "operatorInfo": "range:[1,1], keep order:false, stats:pseudo"
                    }
                ]
            }
        ]
    }
]
 |
+-----------------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.01 sec)
```

在输出结果中，id、estRows、taskType、accessObject、operatorInfo 与默认格式输出的列含义相同。subOperators 是存放子节点的数组，子节点的字段及含义与父节点均相同。若某字段缺失，则说明该字段为空。

14.11.2.58.4 MySQL 兼容性

- EXPLAIN 的格式和 TiDB 中潜在的执行计划都与 MySQL 有很大不同。
- TiDB 不支持 FORMAT=JSON 或 FORMAT=TREE 选项。
- TiDB 支持的 FORMAT=tidb_json 是对当前默认 EXPLAIN 格式的 JSON 编码，与 MySQL 的 FORMAT=JSON 结果的格式、字段信息都不同。

14.11.2.58.5 EXPLAIN FOR CONNECTION

EXPLAIN FOR CONNECTION 用于获得一个连接中当前正在执行 SQL 的执行计划或者是最后执行 SQL 的执行计划，其输出格式与 EXPLAIN 完全一致。但 TiDB 中的实现与 MySQL 不同，除了输出格式之外，还有以下区别：

- 如果连接处于睡眠状态，MySQL 返回为空，而 TiDB 返回的是最后执行的查询计划。
- 如果获取当前会话连接的执行计划，MySQL 会报错，而 TiDB 会正常返回。
- MySQL 的文档中指出，MySQL 要求登录用户与被查询的连接相同，或者拥有 PROCESS 权限，而 TiDB 则要求登录用户与被查询的连接相同，或者拥有 SUPER 权限。

14.11.2.58.6 另请参阅

- 理解 TiDB 执行计划
- EXPLAIN ANALYZE
- ANALYZE TABLE
- TRACE

14.11.2.59 FLASHBACK CLUSTER TO TIMESTAMP

TiDB v6.4.0 引入了 FLASHBACK CLUSTER TO TIMESTAMP 语法，其功能是将集群的数据恢复到特定的时间点。

> **注意：**
>
> FLASHBACK CLUSTER TO TIMESTAMP 是用最新的时间戳写入特定时间点的旧数据，但不会删除当前数据，所以在使用前请确保集群有足够的存储空间来同时容纳旧数据和当前数据。

14.11.2.59.1 语法

```
FLASHBACK CLUSTER TO TIMESTAMP '2022-09-21 16:02:50';
```

语法图

```
FlashbackToTimestampStmt ::=
    "FLASHBACK" "CLUSTER" "TO" "TIMESTAMP" stringLit
```

14.11.2.59.2 注意事项

- FLASHBACK 指定的时间点需要在 Garbage Collection (GC) life time 时间内。你可以使用系统变量 tidb_gc_life_time 配置数据的历史版本的保留时间（默认值是 10m0s）。可以使用以下 SQL 语句查询当前的 safePoint，即 GC 已经清理到的时间点：

```
SELECT * FROM mysql.tidb WHERE variable_name = 'tikv_gc_safe_point';
```

- 执行 FLASHBACK CLUSTER SQL 语句的用户需要有 SUPER 权限。
- FLASHBACK CLUSTER 不支持回退修改 PD 相关信息的 DDL，如 ALTER TABLE ATTRIBUTE、ALTER TABLE REPLICA、CREATE PLACEMENT POLICY 等。
- FLASHBACK CLUSTER 指定的时间点不能存在未执行完成的 DDL 记录。若存在，TiDB 会拒绝该 DDL 操作。
- 在执行 FLASHBACK CLUSTER TO TIMESTAMP 前，TiDB 会主动断开所有相关表上的连接，并禁止对这些表进行读写操作，直到 FLASHBACK CLUSTER 完成。
- FLASHBACK CLUSTER TO TIMESTAMP 命令不能取消，一旦开始执行 TiDB 会一直重试，直到成功。
- 在 FLASHBACK CLUSTER 执行期间，若有数据备份需求，只支持使用BR 命令行工具进行快照备份，并需要指定早于 FLASHBACK CLUSTER 开始时间的 BackupTS。同时，在执行 FLASHBACK CLUSTER 期间，开启日志备份的操作会失败，请等待 FLASHBACK CLUSTER 结束后再尝试开启日志备份。
- 若 FLASHBACK CLUSTER 导致了元信息（表结构、库结构）的回滚，则相关的修改不会被 TiCDC 同步。因此，用户需主动暂停任务，待 FLASHBACK CLUSTER 完成后将上下游的 schema 定义手动同步一致，然后重新创建 TiCDC changefeed。

14.11.2.59.3 示例

恢复新插入的数据：

```
mysql> CREATE TABLE t(a INT);
Query OK, 0 rows affected (0.09 sec)

mysql> SELECT * FROM t;
Empty set (0.01 sec)

mysql> SELECT now();
+---------------------+
| now()               |
+---------------------+
| 2022-09-28 17:24:16 |
+---------------------+
1 row in set (0.02 sec)

mysql> INSERT INTO t VALUES (1);
Query OK, 1 row affected (0.02 sec)
```

```
mysql> SELECT * FROM t;
+------+
| a    |
+------+
|    1 |
+------+
1 row in set (0.01 sec)

mysql> FLASHBACK CLUSTER TO TIMESTAMP '2022-09-28 17:24:16';
Query OK, 0 rows affected (0.20 sec)

mysql> SELECT * FROM t;
Empty set (0.00 sec)
```

如果 FLASHBACK CLUSTER 指定的时间点有未完成的 DDL 记录，那么 FLASHBACK CLUSTER 将执行失败：

```
mysql> ALTER TABLE t ADD INDEX k(a);
Query OK, 0 rows affected (0.56 sec)

mysql> ADMIN SHOW DDL JOBS 1;
+--
    ↪ ------+---------+----------------------+------------------------+--------------+-----------+----------+-
    ↪
| JOB_ID | DB_NAME | TABLE_NAME           | JOB_TYPE               | SCHEMA_STATE | SCHEMA_ID |
    ↪ TABLE_ID | ROW_COUNT | CREATE_TIME          | START_TIME             | END_TIME             |
    ↪ STATE  |
+--
    ↪ ------+---------+----------------------+------------------------+--------------+-----------+----------+-
    ↪
|     84 | test    | t                    | add index /* ingest */ | public       |         2 |
    ↪       82 |         0 | 2023-01-29 14:33:11 | 2023-01-29 14:33:11 | 2023-01-29 14:33:12 |
    ↪ synced |
+--
    ↪ ------+---------+----------------------+------------------------+--------------+-----------+----------+-
    ↪
1 rows in set (0.01 sec)

mysql> FLASHBACK CLUSTER TO TIMESTAMP '2023-01-29 14:33:12';
ERROR 1105 (HY000): Detected another DDL job at 2023-01-29 14:33:12 +0800 CST, can't do flashback
```

可以通过日志查看 FLASHBACK 执行进度，具体的日志如下所示：

```
[2022/10/09 17:25:59.316 +08:00] [INFO] [cluster.go:463] ["flashback cluster stats"] ["complete
    ↪ regions"=9] ["total regions"=10] []
```

14.11.2.59.4 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.60 FLASHBACK DATABASE

TiDB v6.4.0 引入了 FLASHBACK DATABASE 语法，其功能是在 Garbage Collection (GC) life time 时间内，可以用 FLASHBACK DATABASE 语句来恢复被 DROP 删除的数据库以及数据。

可以使用系统变量tidb_gc_life_time 配置数据的历史版本的保留时间（默认值是 10m0s）。可以使用以下 SQL 语句查询当前的 safePoint，即 GC 已经清理到的时间点：

```
SELECT * FROM mysql.tidb WHERE variable_name = 'tikv_gc_safe_point';
```

只要一个数据库是在 tikv_gc_safe_point 时间之后被 DROP，即可以用 FLASHBACK DATABASE 语法来恢复。

14.11.2.60.1 语法

```
FLASHBACK DATABASE DBName [TO newDBName]
```

语法图

```
FlashbackDatabaseStmt ::=
    'FLASHBACK' DatabaseSym DBName FlashbackToNewName

FlashbackToNewName ::=
    ( 'TO' Identifier )?
```

14.11.2.60.2 注意事项

- 如果数据库被删除的时间超过了 GC life time (tikv_gc_safe_point)，就无法使用 FLASHBACK DATABASE 语句来恢复被删除的数据了，否则会返回错误。错误类似于 ERROR 1105 (HY000): Can't find dropped database 'test' in GC safe point 2022-11-06 16:10:10 +0800 CST。
- 不能用 FLASHBACK DATABASE 多次恢复同一个被删除的数据库，因为 FLASHBACK DATABASE 所恢复数据库的 schema ID 和原被删除数据库的 schema ID 一致，多次恢复同一数据库会导致重复的 schema ID。在 TiDB 中，所有数据库的 schema ID 必须全局唯一。
- 在开启 TiDB Binlog 时，使用 FLASHBACK DATABASE 需要注意以下情况：
  - 下游从集群也需要支持 FLASHBACK DATABASE。
  - 从集群的 GC life time 一定要长于主集群的 GC life time。否则上下游同步存在的延迟可能也会造成下游恢复数据失败。
  - 如果 TiDB Binlog 同步出错，则需要在 TiDB Binlog 中过滤掉该数据库，同时手动全量重新导入该数据库的数据。

14.11.2.60.3 示例

- 恢复被 DROP 删除的 test 数据库：

```
DROP DATABASE test;
```

```
FLASHBACK DATABASE test;
```

- 恢复被 DROP 删除的 test 数据库并重命名为 test1：

```
DROP DATABASE test;
```

```
FLASHBACK DATABASE test TO test1;
```

14.11.2.60.4 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.61 FLASHBACK TABLE

在 TiDB 4.0 中，引入了 FLASHBACK TABLE 语法，其功能是在 Garbage Collection (GC) life time 时间内，可以用 FLASHBACK TABLE 语句来恢复被 DROP 或 TRUNCATE 删除的表以及数据。

可以使用系统变量tidb_gc_life_time 配置数据的历史版本的保留时间（默认值是 10m0s）。可以使用以下 SQL 语句查询当前的 safePoint，即 GC 已经清理到的时间点：

```
sql  SELECT * FROM mysql.tidb WHERE variable_name = 'tikv_gc_safe_point';
```

只要被 DROP 或 TRUNCATE 删除的表是在 tikv_gc_safe_point 时间之后，都能用 FLASHBACK TABLE 语法来恢复。

14.11.2.61.1 语法

```
FLASHBACK TABLE table_name [TO other_table_name]
```

语法图

```
FlashbackTableStmt ::=
    'FLASHBACK' 'TABLE' TableName FlashbackToNewName

TableName ::=
    Identifier ( '.' Identifier )?

FlashbackToNewName ::=
    ( 'TO' Identifier )?
```

14.11.2.61.2 注意事项

如果删除了一张表并过了 GC lifetime，就不能再用 FLASHBACK TABLE 语句来恢复被删除的数据了，否则会返回错误，错误类似于 `Can't find dropped/truncated table 't' in GC safe point 2020-03-16 16:34:52 +0800 CST`。

在开启 TiDB Binlog 时使用 FLASHBACK TABLE 需要注意以下情况：

- 下游从集群也支持 FLASHBACK TABLE
- 从集群的 GC lifetime 一定要长于主集群的 GC lifetime。上下游同步存在的延迟可能也会造成下游恢复数据失败。

如果 Binlog 同步出错，则需要在 Binlog 过滤掉该表，同时手动全量重新导入该表的数据。

14.11.2.61.3 示例

- 恢复被 DROP 删除的表数据：

```
DROP TABLE t;
```

```
FLASHBACK TABLE t;
```

- 恢复被 TRUNCATE 的表数据，由于被 TRUNCATE 的表还存在，所以需要重命名被恢复的表，否则会报错表 t 已存在。

```
TRUNCATE TABLE t;
```

```
FLASHBACK TABLE t TO t1;
```

14.11.2.61.4 工作原理

TiDB 在删除表时，实际上只删除了表的元信息，并将需要删除的表数据（行数据和索引数据）写一条数据到 `mysql.gc_delete_range` 表。TiDB 后台的 GC Worker 会定期从 `mysql.gc_delete_range` 表中取出超过 GC lifetime 相关范围的 key 进行删除。

所以，FLASHBACK TABLE 只需要在 GC Worker 还没删除表数据前，恢复表的元信息并删除 `mysql.gc_delete_range` 表中相应的行记录即可。恢复表的元信息可以用 TiDB 的快照读实现。具体的快照读内容可以参考读取历史数据文档。下面是 FLASHBACK TABLE t TO t1 的工作流程：

1. 从 DDL History job 中查找 `drop table` 或者 `truncate table` 类型的操作，且操作的表名是 t 的第一个 DDL job，若没找到，则返回错误。
2. 检查 DDL job 的开始时间，是否在 `tikv_gc_safe_point` 之前。如果是`tikv_gc_safe_point` 之前，说明被 DROP 或 TRUNCATE 删除的表已经被 GC 清理掉，返回错误。
3. 用 DDL job 的开始时间作为 snapshot 读取历史数据，读取表的元信息。
4. 删除 `mysql.gc_delete_range` 中和表 t 相关的 GC 任务。
5. 将表的元信息中的 name 修改成 `t1`，并用该元信息新建一个表。注意：这里只是修改了表名，但是 table ID 不变，依旧是之前被删除的表 t 的 table ID。

可以发现，从表 t 被删除，到表 t 被 FLASHBACK 恢复到 t1，一直都是对表的元信息进行操作，而表的用户数据一直未被修改过。被恢复的表 t1 和之前被删除的表 t 的 table ID 相同，所以表 t1 才能读取表t 的用户数据。

> **注意：**
>
> 不能用 FLASHBACK 多次恢复同一个被删除的表，因为 FLASHBACK 所恢复表的 table ID 还是被删除表的 table ID，而 TiDB 要求所有还存在的表 table ID 必须全局唯一。

FLASHBACK TABLE 是通过快照读获取表的元信息后，再走一次类似于 CREATE TABLE 的建表流程，所以 FLASHBACK TABLE 实际上也是一种 DDL 操作。

14.11.2.61.5 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.62 FLUSH PRIVILEGES

FLUSH PRIVILEGES 语句可触发 TiDB 从权限表中重新加载权限的内存副本。在对如 mysql.user 一类的表进行手动编辑后，应当执行 FLUSH PRIVILEGES。使用如 GRANT 或 REVOKE 一类的权限语句后，不需要执行 FLUSH PRIVILEGES 语句。执行 FLUSH PRIVILEGES 语句的用户需要拥有 RELOAD 权限。

14.11.2.62.1 语法图

```
FlushStmt ::=
    'FLUSH' NoWriteToBinLogAliasOpt FlushOption

NoWriteToBinLogAliasOpt ::=
    ( 'NO_WRITE_TO_BINLOG' | 'LOCAL' )?

FlushOption ::=
    'PRIVILEGES'
|   'STATUS'
|    'TIDB' 'PLUGINS' PluginNameList
|    'HOSTS'
|   LogTypeOpt 'LOGS'
|   TableOrTables TableNameListOpt WithReadLockOpt
```

14.11.2.62.2 示例

```
FLUSH PRIVILEGES;
```

```
Query OK, 0 rows affected (0.01 sec)
```

14.11.2.62.3 MySQL 兼容性

`FLUSH PRIVILEGES` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.62.4 另请参阅

- `GRANT <privileges>`
- `REVOKE <privileges>`
- Privilege Management

14.11.2.63 FLUSH STATUS

`FLUSH STATUS` 语句用于提供 MySQL 兼容性，但在 TiDB 上并无作用。因为 TiDB 使用 Prometheus 和 Grafana 而非 `SHOW STATUS` 来进行集中度量收集。

14.11.2.63.1 语法图

```
FlushStmt ::=
    'FLUSH' NoWriteToBinLogAliasOpt FlushOption

NoWriteToBinLogAliasOpt ::=
    ( 'NO_WRITE_TO_BINLOG' | 'LOCAL' )?

FlushOption ::=
    'PRIVILEGES'
|   'STATUS'
|    'TIDB' 'PLUGINS' PluginNameList
|    'HOSTS'
|   LogTypeOpt 'LOGS'
|   TableOrTables TableNameListOpt WithReadLockOpt
```

14.11.2.63.2 示例

```
show status;
```

```
+--------------------+--------------------------------------+
| Variable_name      | Value                                |
+--------------------+--------------------------------------+
| Ssl_cipher_list    |                                      |
| server_id          | 93e2e07d-6bb4-4a1b-90b7-e035fae154fe |
| ddl_schema_version | 141                                  |
| Ssl_verify_mode    | 0                                    |
| Ssl_version        |                                      |
| Ssl_cipher         |                                      |
+--------------------+--------------------------------------+
6 rows in set (0.01 sec)
```

```
show global status;
```

```
+--------------------+--------------------------------------+
| Variable_name      | Value                                |
+--------------------+--------------------------------------+
| Ssl_cipher         |                                      |
| Ssl_cipher_list    |                                      |
| Ssl_verify_mode    | 0                                    |
| Ssl_version        |                                      |
| server_id          | 93e2e07d-6bb4-4a1b-90b7-e035fae154fe |
| ddl_schema_version | 141                                  |
+--------------------+--------------------------------------+
6 rows in set (0.00 sec)
```

```
flush status;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
show status;
```

```
+--------------------+--------------------------------------+
| Variable_name      | Value                                |
+--------------------+--------------------------------------+
| Ssl_cipher         |                                      |
| Ssl_cipher_list    |                                      |
| Ssl_verify_mode    | 0                                    |
| Ssl_version        |                                      |
| ddl_schema_version | 141                                  |
| server_id          | 93e2e07d-6bb4-4a1b-90b7-e035fae154fe |
+--------------------+--------------------------------------+
6 rows in set (0.00 sec)
```

14.11.2.63.3 MySQL 兼容性

- FLUSH STATUS 语句仅用于提供 MySQL 兼容性。

14.11.2.63.4 另请参阅

- SHOW [GLOBAL | SESSION] STATUS

14.11.2.64 FLUSH TABLES

FLUSH TABLES 语句用于提供 MySQL 兼容性，但在 TiDB 中并无有效用途。

14.11.2.64.1 语法图

```
FlushStmt ::=
    'FLUSH' NoWriteToBinLogAliasOpt FlushOption

NoWriteToBinLogAliasOpt ::=
    ( 'NO_WRITE_TO_BINLOG' | 'LOCAL' )?

FlushOption ::=
    'PRIVILEGES'
|   'STATUS'
|    'TIDB' 'PLUGINS' PluginNameList
|    'HOSTS'
|    LogTypeOpt 'LOGS'
|    TableOrTables TableNameListOpt WithReadLockOpt

LogTypeOpt ::=
    ( 'BINARY' | 'ENGINE' | 'ERROR' | 'GENERAL' | 'SLOW' )?

TableOrTables ::=
    'TABLE'
|   'TABLES'

TableNameListOpt ::=
    TableNameList?

WithReadLockOpt ::=
    ( 'WITH' 'READ' 'LOCK' )?
```

14.11.2.64.2 示例

```
FLUSH TABLES;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
FLUSH TABLES WITH READ LOCK;
```

```
ERROR 1105 (HY000): FLUSH TABLES WITH READ LOCK is not supported.  Please use @@tidb_snapshot
```

14.11.2.64.3 MySQL 兼容性

- TiDB 没有 MySQL 中的表缓存这一概念。所以，`FLUSH TABLES` 因 MySQL 兼容性会在 TiDB 中解析出但会被忽略掉。
- 因为 TiDB 目前不支持锁表，所以`FLUSH TABLES WITH READ LOCK` 语句会产生错误。建议使用Historical reads来实现锁表。

14.11.2.64.4 另请参阅

- Read historical data

14.11.2.65 GRANT <privileges>

GRANT <privileges> 语句用于为 TiDB 中已存在的用户分配权限。TiDB 中的权限系统同 MySQL 一样，都基于数据库/表模式来分配凭据。执行 GRANT <privileges> 语句需要拥有分配的权限，并且拥有 GRANT OPTION 权限。

14.11.2.65.1 语法图

```
GrantStmt ::=
    'GRANT' PrivElemList 'ON' ObjectType PrivLevel 'TO' UserSpecList RequireClauseOpt
        ↪ WithGrantOptionOpt

PrivElemList ::=
    PrivElem ( ',' PrivElem )*

PrivElem ::=
    PrivType ( '(' ColumnNameList ')' )?

PrivType ::=
    'ALL' 'PRIVILEGES'?
|    'ALTER' 'ROUTINE'?
|   'CREATE' ( 'USER' | 'TEMPORARY' 'TABLES' | 'VIEW' | 'ROLE' | 'ROUTINE' )?
|   'TRIGGER'
|   'DELETE'
|   'DROP' 'ROLE'?
|    'PROCESS'
|   'EXECUTE'
|   'INDEX'
|   'INSERT'
|   'SELECT'
|   'SUPER'
|   'SHOW' ( 'DATABASES' | 'VIEW' )
|   'UPDATE'
|    'GRANT' 'OPTION'
|    'REFERENCES'
|    'REPLICATION' ( 'SLAVE' | 'CLIENT' )
|    'USAGE'
|   'RELOAD'
|   'FILE'
|   'CONFIG'
|   'LOCK' 'TABLES'
|   'EVENT'
|   'SHUTDOWN'
```

```
ObjectType ::=
    'TABLE'?

PrivLevel ::=
    '*' ( '.' '*' )?
|    Identifier ( '.' ( '*' | Identifier ) )?

UserSpecList ::=
    UserSpec ( ',' UserSpec )*
```

14.11.2.65.2 示例

```
CREATE USER 'newuser' IDENTIFIED BY 'mypassword';
```

```
Query OK, 1 row affected (0.02 sec)
```

```
GRANT ALL ON test.* TO 'newuser';
```

```
Query OK, 0 rows affected (0.03 sec)
```

```
SHOW GRANTS FOR 'newuser';
```

```
+-------------------------------------------------+
| Grants for newuser@%                            |
+-------------------------------------------------+
| GRANT USAGE ON *.* TO 'newuser'@'%'             |
| GRANT ALL PRIVILEGES ON test.* TO 'newuser'@'%' |
+-------------------------------------------------+
2 rows in set (0.00 sec)
```

14.11.2.65.3 MySQL 兼容性

- 与 MySQL 类似，USAGE 权限表示登录 TiDB 服务器的能力。
- 目前不支持列级权限。
- 与 MySQL 类似，不存在 NO_AUTO_CREATE_USER sql 模式时，GRANT 语句将在用户不存在时自动创建一个空密码的新用户。删除此 sql-mode（默认情况下已启用）会带来安全风险。
- GRANT <privileges> 语句执行成功后，在 TiDB 中语句执行的结果会在当前连接立即生效，而 MySQL 中部分权限的结果需要等到之后的连接才生效。见 TiDB #39356。

14.11.2.65.4 另请参阅

- GRANT <role>
- REVOKE <privileges>
- SHOW GRANTS
- 权限管理

14.11.2.66 GRANT <role>

GRANT <role> 语句用于将之前创建的角色授予给现有用户。用户可以通过 SET ROLE <rolename> 语句拥有角色权限，或者通过 SET ROLE ALL 语句拥有被授予的所有角色。

14.11.2.66.1 语法图

```
GrantRoleStmt ::=
    'GRANT' RolenameList 'TO' UsernameList

RolenameList ::=
    Rolename ( ',' Rolename )*

UsernameList ::=
    Username ( ',' Username )*
```

14.11.2.66.2 示例

以 root 用户连接 TiDB：

```
mysql -h 127.0.0.1 -P 4000 -u root
```

创建新角色 analyticsteam 和新用户 jennifer：

```
CREATE ROLE analyticsteam;
Query OK, 0 rows affected (0.02 sec)

GRANT SELECT ON test.* TO analyticsteam;
Query OK, 0 rows affected (0.02 sec)

CREATE USER jennifer;
Query OK, 0 rows affected (0.01 sec)

GRANT analyticsteam TO jennifer;
Query OK, 0 rows affected (0.01 sec)
```

以 jennifer 用户连接 TiDB：

```
mysql -h 127.0.0.1 -P 4000 -u jennifer
```

需要注意的是，默认情况下，用户 jennifer 需要执行 SET ROLE analyticsteam 语句才能使用与 analyticsteam 角色相关联的权限：

```
SHOW GRANTS;
+-------------------------------------------+
| Grants for User                           |
+-------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'      |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+-------------------------------------------+
2 rows in set (0.00 sec)

SHOW TABLES in test;
ERROR 1044 (42000): Access denied for user 'jennifer'@'%' to database 'test'
SET ROLE analyticsteam;
Query OK, 0 rows affected (0.00 sec)

SHOW GRANTS;
+-------------------------------------------+
| Grants for User                           |
+-------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'      |
| GRANT SELECT ON test.* TO 'jennifer'@'%'  |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+-------------------------------------------+
3 rows in set (0.00 sec)

SHOW TABLES IN test;
+----------------+
| Tables_in_test |
+----------------+
| t1             |
+----------------+
1 row in set (0.00 sec)
```

以 root 用户连接 TiDB：

```
mysql -h 127.0.0.1 -P 4000 -u root
```

执行 SET DEFAULT ROLE 语句将用户 jennifer 与 analyticsteam 角色相关联：

```
SET DEFAULT ROLE analyticsteam TO jennifer;
Query OK, 0 rows affected (0.02 sec)
```

以 jennifer 用户连接 TiDB：

```
mysql -h 127.0.0.1 -P 4000 -u jennifer
```

此时 jennifer 用户无需执行 SET ROLE 语句就能拥有 analyticsteam 角色相关联的权限：

```
SHOW GRANTS;
+-------------------------------------------+
| Grants for User                           |
+-------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'      |
| GRANT SELECT ON test.* TO 'jennifer'@'%'  |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+-------------------------------------------+
3 rows in set (0.00 sec)

SHOW TABLES IN test;
+----------------+
| Tables_in_test |
+----------------+
| t1             |
+----------------+
1 row in set (0.00 sec)
```

14.11.2.66.3 MySQL 兼容性

GRANT <role> 语句与 MySQL 8.0 的角色功能完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.66.4 另请参阅

- GRANT <privileges>
- CREATE ROLE
- DROP ROLE
- REVOKE <role>
- SET ROLE
- SET DEFAULT ROLE
- 基于角色的访问控制

14.11.2.67 INSERT

使用 INSERT 语句在表中插入新行。

14.11.2.67.1 语法图

```
InsertIntoStmt ::=
    'INSERT' TableOptimizerHints PriorityOpt IgnoreOptional IntoOpt TableName
        ↪ PartitionNameListOpt InsertValues OnDuplicateKeyUpdate

TableOptimizerHints ::=
```

```
    hintComment?

PriorityOpt ::=
    ( 'LOW_PRIORITY' | 'HIGH_PRIORITY' | 'DELAYED' )?

IgnoreOptional ::=
    'IGNORE'?

IntoOpt  ::= 'INTO'?

TableName ::=
    Identifier ( '.' Identifier )?

PartitionNameListOpt ::=
    ( 'PARTITION' '(' Identifier ( ',' Identifier )* ')' )?

InsertValues ::=
    '(' ( ColumnNameListOpt ')' ( ValueSym ValuesList | SelectStmt | '(' SelectStmt ')' |
        ↪ UnionStmt ) | SelectStmt ')' )
|   ValueSym ValuesList
|   SelectStmt
|   UnionStmt
|   'SET' ColumnSetValue? ( ',' ColumnSetValue )*

OnDuplicateKeyUpdate ::=
    ( 'ON' 'DUPLICATE' 'KEY' 'UPDATE' AssignmentList )?
```

14.11.2.67.2 示例

```
CREATE TABLE t1 (a INT);
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
CREATE TABLE t2 LIKE t1;
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
INSERT INTO t1 VALUES (1);
```

```
Query OK, 1 row affected (0.02 sec)
```

```
INSERT INTO t1 (a) VALUES (1);
```

```
Query OK, 1 row affected (0.01 sec)
```

```
INSERT INTO t2 SELECT * FROM t1;
```

```
Query OK, 2 rows affected (0.01 sec)
Records: 2  Duplicates: 0  Warnings: 0
```

```
SELECT * FROM t1;
```

```
+------+
| a    |
+------+
|    1 |
|    1 |
+------+
2 rows in set (0.00 sec)
```

```
SELECT * FROM t2;
```

```
+------+
| a    |
+------+
|    1 |
|    1 |
+------+
2 rows in set (0.00 sec)
```

```
INSERT INTO t2 VALUES (2),(3),(4);
```

```
Query OK, 3 rows affected (0.02 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
SELECT * FROM t2;
```

```
+------+
| a    |
+------+
|    1 |
|    1 |
|    2 |
|    3 |
|    4 |
+------+
5 rows in set (0.00 sec)
```

14.11.2.67.3 MySQL 兼容性

INSERT 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.67.4 另请参阅

- DELETE
- SELECT
- UPDATE
- REPLACE

14.11.2.68 KILL

KILL 语句可以终止当前 TiDB 集群中任意一个 TiDB 实例中的某个连接。

14.11.2.68.1 语法图

```
KillStmt ::= 'KILL' 'TIDB'? ( 'CONNECTION' | 'QUERY' )? CONNECTION_ID
```

14.11.2.68.2 示例

查询当前集群中所有活跃查询，并终止其中某一个连接：

```
SELECT ID, USER, INSTANCE, INFO FROM INFORMATION_SCHEMA.CLUSTER_PROCESSLIST;
```

```
+--------------------+------+----------------+-------------------------------------------------------------------
    ↪
| ID                 | USER | INSTANCE       | INFO
    ↪                                                                                 |
+--------------------+------+----------------+-------------------------------------------------------------------
    ↪
| 8306449708033769879 | root | 127.0.0.1:10082 | select sleep(30), 'foo'
    ↪                                                      |
| 5857102839209263511 | root | 127.0.0.1:10080 | select sleep(50)
    ↪                                                                  |
| 5857102839209263513 | root | 127.0.0.1:10080 | SELECT ID, USER, INSTANCE, INFO FROM
    ↪ INFORMATION_SCHEMA.CLUSTER_PROCESSLIST |
+--------------------+------+----------------+-------------------------------------------------------------------
    ↪
```

```
KILL 5857102839209263511;
```

```
Query OK, 0 rows affected (0.00 sec)
```

14.11.2.68.3 MySQL 兼容性

- MySQL 的 KILL 语句仅能终止当前连接的 MySQL 实例上的连接，TiDB 的 KILL 语句能终止整个集群中任意一个 TiDB 实例上的连接。
- 暂时不支持使用 MySQL 命令行 ctrl+c 终止查询或连接。

14.11.2.68.4 行为变更说明

TiDB 从 v6.1.0 起新增 Global Kill 功能（由enable-global-kill 配置项控制，默认启用）。启用 Global Kill 功能时，KILL 语句和 KILL TIDB 语句均能跨节点终止查询或连接，且无需担心错误地终止其他查询或连接。当你使用客户端连接到任何一个 TiDB 节点执行 KILL 语句或 KILL TIDB 语句时，该语句会被转发给对应的 TiDB 节点。当客户端和 TiDB 中间有代理时，KILL 及 KILL TIDB 语句也会被转发给对应的 TiDB 节点执行。

对于 TiDB v6.1.0 之前的版本，或未启用 Global Kill 功能时：

- KILL 语句与 MySQL 不兼容，负载均衡器后面通常放有多个 TiDB 服务器，这种不兼容有助于防止在错误的 TiDB 服务器上终止连接。你需要显式地增加 TIDB 后缀，通过执行 KILL TIDB 语句来终止当前连接的 TiDB 实例上的其他连接。
- 强烈不建议在配置文件里设置compatible-kill-query = true，除非你确定客户端将始终连接到同一个 TiDB 节点。这是因为当你在默认的 MySQL 客户端按下 ctrl+c 时，客户端会开启一个新连接，并在这个新连接中执行 KILL 语句。此时，如果客户端和 TiDB 中间有代理，新连接可能会被路由到其他的 TiDB 节点，从而错误地终止其他会话。
- KILL TIDB 语句是 TiDB 的扩展语法，其功能与 MySQL 命令 KILL [CONNECTION|QUERY] 和 MySQL 命令行 ctrl+c 相同。在同一个 TiDB 节点上，你可以安全地使用 KILL TIDB 语句。

14.11.2.68.5 另请参阅

- SHOW [FULL] PROCESSLIST
- CLUSTER_PROCESSLIST

14.11.2.69 LOAD DATA

LOAD DATA 语句用于将数据批量加载到 TiDB 表中。

在 v7.0.0 版本支持以下功能：

- 支持从 S3、GCS 导入数据
- 新增参数 FIELDS DEFINED NULL BY

> **警告：**
>
> v7.0.0 新增支持从 S3、GCS 导入数据和 FIELDS DEFINED NULL BY 参数为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现 bug，请在 GitHub 上提 issue 反馈。

14.11.2.69.1 语法图

```
LoadDataStmt ::=
    'LOAD' 'DATA' LocalOpt 'INFILE' stringLit DuplicateOpt 'INTO' 'TABLE' TableName CharsetOpt
        ↪ Fields Lines IgnoreLines ColumnNameOrUserVarListOptWithBrackets LoadDataSetSpecOpt

LocalOpt ::= ('LOCAL')?

Fields ::=
    ('TERMINATED' 'BY' stringLit
    | ('OPTIONALLY')? 'ENCLOSED' 'BY' stringLit
    | 'ESCAPED' 'BY' stringLit
    | 'DEFINED' 'NULL' 'BY' stringLit ('OPTIONALLY' 'ENCLOSED')?)?
```

14.11.2.69.2 参数说明

LOCAL

你可以使用 LOCAL 来指定导入位于客户端的数据文件，此时传入文件参数必须为客户端文件系统路径。

S3/GCS 路径

如果你不指定 LOCAL，则文件参数必须是有效的 S3/GCS URI 路径，详见外部存储。

当数据文件存储在 S3/GCS 上时，你可以导入单个文件，也可使用通配符 * 来匹配需要导入的多个文件。注意通配符不会递归处理子目录下相关的文件。以数据存储在 S3 为例，示例如下:

- 导入单个文件：s3://<bucket-name>/path/to/data/foo.csv
- 导入指定路径下的所有文件：s3://<bucket-name>/path/to/data/*
- 导入指定路径下的所有以 .csv 结尾的文件：s3://<bucket-name>/path/to/data/*.csv
- 导入指定路径下所有以 foo 为前缀的文件：s3://<bucket-name>/path/to/data/foo*
- 导入指定路径下以 foo 为前缀、以 .csv 结尾的文件：s3://<bucket-name>/path/to/data/foo*.csv

Fields、Lines、Ignore Lines

你可以使用 Fields 和 Lines 参数来指定如何处理数据格式：

- 使用 FIELDS TERMINATED BY 来指定数据的分隔符号。
- 使用 FIELDS ENCLOSED BY 来指定数据的包围符号。
- 如果你希望以某个字符为结尾来切分行数据，可以使用 LINES TERMINATED BY 来指定行的终止符。

可以使用 DEFINED NULL BY 来指定数据文件中如何表示 NULL 值。

- 与 MySQL 行为一致，如果 ESCAPED BY 不为空时，例如是默认值 \，那么 \N 会被认为是 NULL 值。
- 如果使用 DEFINED NULL BY，例如 DEFINED NULL BY 'my-null'，my-null 会被认为是 NULL 值。
- 如果使用 DEFINED NULL BY ... OPTIONALLY ENCLOSED，例如 DEFINED NULL BY 'my-null' OPTIONALLY ↪ ENCLOSED，my-null 和 "my-null"（假设 ENCLOSED BY '"'）会被认为是 NULL 值。

- 如果没有使用 DEFINED NULL BY 或者 DEFINED NULL BY ... OPTIONALLY ENCLOSED，但使用了 ENCLOSED BY，例如 ENCLOSED BY '"'，那么 NULL 会被认为是 NULL 值。这个行为与 MySQL 一致。
- 其他情况不会被认为是 NULL 值。

例如对于以下格式的数据：

```
"bob","20","street 1"\r\n
"alice","33","street 1"\r\n
```

如果想分别提取 bob、20、street 1，可以指定数据的分隔符号为 ','，数据的包围符号为 '\"'。可以写成：

```
FIELDS TERMINATED BY ',' ENCLOSED BY '\"' LINES TERMINATED BY '\r\n'
```

当不指定处理数据的参数时，将按以下参数处理：

```
FIELDS TERMINATED BY '\t' ENCLOSED BY '' ESCAPED BY '\\'
LINES TERMINATED BY '\n' STARTING BY ''
```

你可以通过 IGNORE <number> LINES 参数来忽略文件开始的 <number> 行。例如，可以使用 IGNORE 1 LINES 来忽略文件的第一行。

14.11.2.69.3 示例

通过 LOAD DATA 导入数据，指定数据的分隔符为逗号，忽略包围数据的引号，并且忽略文件的第一行数据。

如果此时遇到 ERROR 1148 (42000): the used command is not allowed with this TiDB version 报错信息。可以参考文档解决：ERROR 1148 (42000): the used command is not allowed with this TiDB version 问题的处理方法

```
LOAD DATA LOCAL INFILE '/mnt/evo970/data-sets/bikeshare-data/2017Q4-capitalbikeshare-tripdata.csv
    ↪ ' INTO TABLE trips FIELDS TERMINATED BY ',' ENCLOSED BY '\"' LINES TERMINATED BY '\r\n'
    ↪ IGNORE 1 LINES (duration, start_date, end_date, start_station_number, start_station,
    ↪ end_station_number, end_station, bike_number, member_type);
```

```
Query OK, 815264 rows affected (39.63 sec)
Records: 815264  Deleted: 0  Skipped: 0  Warnings: 0
```

LOAD DATA 也支持使用十六进制 ASCII 字符表达式或二进制 ASCII 字符表达式作为 FIELDS ENCLOSED BY 和 FIELDS TERMINATED BY 的参数。示例如下：

```
LOAD DATA LOCAL INFILE '/mnt/evo970/data-sets/bikeshare-data/2017Q4-capitalbikeshare-tripdata.csv
    ↪ ' INTO TABLE trips FIELDS TERMINATED BY x'2c' ENCLOSED BY b'100010' LINES TERMINATED BY '
    ↪ \r\n' IGNORE 1 LINES (duration, start_date, end_date, start_station_number, start_station
    ↪ , end_station_number, end_station, bike_number, member_type);
```

以上示例中 x'2c' 是字符 , 的十六进制表示，b'100010' 是字符 " 的二进制表示。

14.11.2.69.4 MySQL 兼容性

TiDB 中的 LOAD DATA 语句语法上兼容 MySQL（除字符集选项被解析但会被忽略以外）。若发现任何语法兼容性差异，请在 GitHub 上提交 issue。

> 注意：
>
> - 在 TiDB v4.0.0 之前的版本中，LOAD DATA 语句每 20000 行进行一次提交。
> - 从 TiDB v4.0.0 开始一直到 TiDB v6.6.0 的版本，TiDB 默认在一个事务中提交所有行。
> - 从 TiDB v4.0.0 及以前版本升级后，可能出现 `ERROR 8004 (HY000)at line 1: Transaction is too large, size: 100000058` 错误。要解决该问题，建议调大 `tidb.toml` 文件中的`txn-total-size-limit` 值。如果无法增加此限制，还可以将`tidb_dml_batch_size` 的值设置为 20000 来恢复升级前的行为。注意自 v7.0.0 起，`tidb_dml_batch_size` 对 LOAD DATA 语句不再生效。
> - 无论以多少行为一个事务提交，LOAD DATA 都不会被显式事务中的`ROLLBACK` 语句回滚。
> - LOAD DATA 语句始终以乐观事务模式执行，不受 TiDB 事务模式设置的影响。

14.11.2.69.5 另请参阅

- INSERT
- 乐观事务模型
- 悲观事务模式

14.11.2.70 LOAD STATS

LOAD STATS 语句用于将统计信息加载到 TiDB 中。

14.11.2.70.1 语法图

```
LoadStatsStmt ::=
    'LOAD' 'STATS' stringLit
```

14.11.2.70.2 参数说明

用户直接指定统计信息文件路径，统计信息文件可通过访问 API `http://${tidb-server-ip}:${tidb-server-status-port}/stats/dump/${db_name}/${table_name}` 进行下载。

路径可以是相对路径，也可以是绝对路径，如果是相对路径，会从启动 `tidb-server` 的路径为起点寻找对应文件。

下面是一个绝对路径的例子：

```
LOAD STATS '/tmp/stats.json';
```

```
Query OK, 0 rows affected (0.00 sec)
```

14.11.2.70.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.70.4 另请参阅

- 统计信息

14.11.2.71 LOCK STATS

`LOCK STATS` 语句用于锁定统计信息。

> **警告：**
>
> 锁定统计信息目前为实验特性，不建议在生产环境中使用。

14.11.2.71.1 语法图

```
LockStatsStmt ::=
    'LOCK' 'STATS' TableNameList

TableNameList ::=
    TableName (',' TableName)*

TableName ::=
    Identifier ( '.' Identifier )?
```

14.11.2.71.2 示例

创建表 `t`，插入一些数据，在未锁定表 `t` 的统计信息的情况下成功执行 `ANALYZE` 语句。

```
mysql> create table t(a int, b int);
Query OK, 0 rows affected (0.03 sec)

mysql> insert into t values (1,2), (3,4), (5,6), (7,8);
Query OK, 4 rows affected (0.00 sec)
Records: 4  Duplicates: 0  Warnings: 0

mysql> analyze table t;
Query OK, 0 rows affected, 1 warning (0.02 sec)

mysql> show warnings;
+-------+------+----------------------------------------------------------------------+
| Level | Code | Message                                                              |
```

```
+-------+------+--------------------------------------------------------------------+
| Note  | 1105 | Analyze use auto adjusted sample rate 1.000000 for table test.t |
+-------+------+--------------------------------------------------------------------+
1 row in set (0.00 sec)
```

锁定表 t 的统计信息，执行 ANALYZE 语句，warning 提示跳过对表 t 的 ANALYZE。

```
mysql> lock stats t;
Query OK, 0 rows affected (0.00 sec)

mysql> show stats_locked;
+---------+------------+----------------+--------+
| Db_name | Table_name | Partition_name | Status |
+---------+------------+----------------+--------+
| test    | t          |                | locked |
+---------+------------+----------------+--------+
1 row in set (0.01 sec)

mysql> analyze table t;
Query OK, 0 rows affected, 2 warnings (0.00 sec)

mysql> show warnings;
+---------+------+--------------------------------------------------------------------+
| Level   | Code | Message                                                            |
+---------+------+--------------------------------------------------------------------+
| Note    | 1105 | Analyze use auto adjusted sample rate 1.000000 for table test.t |
| Warning | 1105 | skip analyze locked table: t                                       |
+---------+------+--------------------------------------------------------------------+
2 rows in set (0.00 sec)
```

解锁表 t 的统计信息，成功执行 ANALYZE 语句。

```
mysql> unlock stats t;
Query OK, 0 rows affected (0.01 sec)

mysql> analyze table t;
Query OK, 0 rows affected, 1 warning (0.03 sec)

mysql> show warnings;
+-------+------+--------------------------------------------------------------------+
| Level | Code | Message                                                            |
+-------+------+--------------------------------------------------------------------+
| Note  | 1105 | Analyze use auto adjusted sample rate 1.000000 for table test.t |
+-------+------+--------------------------------------------------------------------+
1 row in set (0.00 sec)
```

14.11.2.71.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.71.4 另请参阅

- 统计信息
- UNLOCK STATS
- SHOW STATS_LOCKED

14.11.2.72 LOCK TABLES 和 UNLOCK TABLES

> **警告：**
>
> LOCK TABLES 和 UNLOCK TABLES 目前为实验特性，不建议在生产环境中使用。

客户端会话可以使用 LOCK TABLES 语句获取表锁，以便和其他会话合作访问表，或者防止其他会话修改表。会话只能为自己获取或释放锁。一个会话无法为另一个会话获取表锁或释放另一个会话持有的表锁。

LOCK TABLES 可以为当前客户端会话获取表锁。你可以获取普通表的表锁，但你必须拥有锁定对象的 LOCK TABLES 和 SELECT 权限。

UNLOCK TABLES 显式释放当前会话持有的所有表锁。LOCK TABLES 在获取新锁之前会隐式释放当前会话持有的所有表锁。

表锁可以防止其他会话的读取或写入。持有 WRITE 锁的会话可以执行表级操作，例如 DROP TABLE 或 TRUNCATE TABLE。

> **警告：**
>
> 开启表锁功能需要在所有 TiDB 实例的配置文件中设置enable-table-lock 为 true。

14.11.2.72.1 语法图

```
LockTablesDef
         ::= 'LOCK' ( 'TABLES' | 'TABLE' ) TableName LockType ( ',' TableName LockType)*


UnlockTablesDef
         ::= 'UNLOCK' 'TABLES'

LockType
         ::= 'READ' ('LOCAL')?
           | 'WRITE' ('LOCAL')?
```

14.11.2.72.2 获取表锁

你可以在会话中使用 LOCK TABLES 语句获取表锁。表锁有以下类型：

READ 锁：

- 持有 READ 锁的会话可以读表，但不能写入。
- 多个会话可以同时获取同一个表的 READ 锁。
- 其他会话可以在不显式获取 READ 锁的情况下读表。

READ LOCAL 锁只是语法兼容 MySQL，实际上并不支持。

WRITE 锁：

- 持有 WRITE 锁的会话可以读取和写入表。
- 只有持有 WRITE 锁的会话才能访问该表。在释放锁之前，其他会话不能读取或写入该表。

WRITE LOCAL 锁：

- 持有 WRITE LOCAL 锁的会话可以读取和写入表。
- 只有持有 WRITE LOCAL 锁的会话才能写入该表，但其他会话依然可以读取该表的数据。

如果 LOCK TABLES 语句想要获取的表锁被其他会话持有且必须等待锁释放时，则 LOCK TABLES 语句会执行报错，例如：

```
> LOCK TABLES t1 read;
ERROR 8020 (HY000): Table 't1' was locked in WRITE by server: f4799bcb-cad7-4285-8a6d-23
    ↪ d3555173f1_session: 2199023255959
```

以上错误信息表明，在 TiDB f4799bcb-cad7-4285-8a6d-23d3555173f1 中，ID 为 2199023255959 的会话已经持有表 t1 的 WRITE 锁，所以，当前会话无法获取表 t1 的 READ 锁。

不能在单个 LOCK TABLES 语句中多次获取同一个表的锁。

```
> LOCK TABLES t WRITE, t READ;
ERROR 1066 (42000): Not unique table/alias: 't'
```

14.11.2.72.3 释放表锁

释放会话持有的表锁时，将同时释放该会话所有持有的表锁。会话可以显式或隐式释放表锁：

- 会话可以使用 UNLOCK TABLES 语句显式释放其持有的所有表锁。
- 当会话使用 LOCK TABLES 语句来获取表锁，而且同时已经持有其他表锁时，则在获取新表锁之前，将隐式释放其已经持有的所有表锁。

当客户端会话的连接终止（无论是正常还是异常），TiDB 都会隐式释放会话持有的所有表锁。如果客户端重新连接，锁将不再有效。因此，通常不建议在客户端使用自动重新连接，因为在开启自动重新连接时，如果发生重新连接，任何表锁或当前事务都将丢失，而且不会通知客户端。禁用自动重新连接后，如果连接断开，则客户端会话发出的下一条语句将会收到报错信息。客户端可以检测到错误并采取适当的操作，例如重新获取锁或重做事务。

14.11.2.72.4 表锁的使用限制和条件

你可以安全地使用 KILL 语句终止已经持有表锁的会话。

不支持获取以下数据库中表的表锁：

- INFORMATION_SCHEMA
- PERFORMANCE_SCHEMA
- METRICS_SCHEMA
- mysql

14.11.2.72.5 和 MySQL 的兼容性

获取表锁

- 在 TiDB 中，如果会话 A 已经持有了一个表的表锁，另一个会话 B 对该表写入时会报错；但 MySQL 会阻塞会话 B 对该表的写入，直到会话 A 释放该表锁。其他会话对该表的锁请求会被阻塞直到当前会话释放 WRITE 锁。
- 在 TiDB 中，如果 LOCK TABLES 语句想要获取的表锁被其他会话持有且必须等待锁释放时，LOCK TABLES 语句会执行报错；但 MySQL 会阻塞 LOCK TABLES 语句的执行，直到成功获取想要的表锁。
- 在 TiDB 中，使用 LOCK TABLES 语句获取表锁的作用域是整个集群；但 MySQL 中表锁的作用域是单个 MySQL 服务器，与 NDB 群集不兼容。

释放表锁

在 TiDB 的会话中显示开启一个事务时（例如使用 BEGIN 语句），TiDB 不会隐式释放当前会话已经持有的表锁；但 MySQL 会隐式释放当前会话已经持有的表锁。

14.11.2.73 MODIFY COLUMN

ALTER TABLE .. MODIFY COLUMN 语句用于修改已有表上的列，包括列的数据类型和属性。若要同时重命名，可改用CHANGE COLUMN 语句。

从 v5.1.0 版本起，TiDB 开始支持 Reorg 类型变更，包括但不限于：

- 从 VARCHAR 转换为 BIGINT
- DECIMAL 精度修改
- 从 VARCHAR(10) 到 VARCHAR(5) 的长度压缩

14.11.2.73.1 语法图

```
AlterTableStmt
         ::= 'ALTER' 'IGNORE'? 'TABLE' TableName ModifyColumnSpec ( ',' ModifyColumnSpec )*

ModifyColumnSpec
         ::= 'MODIFY' ColumnKeywordOpt 'IF EXISTS' ColumnName ColumnType ColumnOption* ( 'FIRST'
             ↪ | 'AFTER' ColumnName )?
```

```
ColumnType
         ::= NumericType
           | StringType
           | DateAndTimeType
           | 'SERIAL'

ColumnOption
         ::= 'NOT'? 'NULL'
           | 'AUTO_INCREMENT'
           | 'PRIMARY'? 'KEY' ( 'CLUSTERED' | 'NONCLUSTERED' )?
           | 'UNIQUE' 'KEY'?
           | 'DEFAULT' ( NowSymOptionFraction | SignedLiteral | NextValueForSequence )
           | 'SERIAL' 'DEFAULT' 'VALUE'
           | 'ON' 'UPDATE' NowSymOptionFraction
           | 'COMMENT' stringLit
           | ( 'CONSTRAINT' Identifier? )? 'CHECK' '(' Expression ')' ( 'NOT'? ( 'ENFORCED' | '
               ↪ NULL' ) )?
           | 'GENERATED' 'ALWAYS' 'AS' '(' Expression ')' ( 'VIRTUAL' | 'STORED' )?
           | 'REFERENCES' TableName ( '(' IndexPartSpecificationList ')' )? Match?
               ↪ OnDeleteUpdateOpt
           | 'COLLATE' CollationName
           | 'COLUMN_FORMAT' ColumnFormat
           | 'STORAGE' StorageMedia
           | 'AUTO_RANDOM' ( '(' LengthNum ')' )?

ColumnName ::=
    Identifier ( '.' Identifier ( '.' Identifier )? )?
```

14.11.2.73.2 示例

Meta-Only Change

```
CREATE TABLE t1 (id int not null primary key AUTO_INCREMENT, col1 INT);
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
INSERT INTO t1 (col1) VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 5 rows affected (0.02 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
ALTER TABLE t1 MODIFY col1 BIGINT;
```

```
Query OK, 0 rows affected (0.09 sec)
```

```
SHOW CREATE TABLE t1\G;
```

```
*************************** 1. row ***************************
       Table: t1
Create Table: CREATE TABLE `t1` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `col1` bigint(20) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin AUTO_INCREMENT=30001
1 row in set (0.00 sec)
```

Reorg-Data Change

```
CREATE TABLE t1 (id int not null primary key AUTO_INCREMENT, col1 INT);
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
INSERT INTO t1 (col1) VALUES (12345),(67890);
```

```
Query OK, 2 rows affected (0.00 sec)
Records: 2  Duplicates: 0  Warnings: 0
```

```
ALTER TABLE t1 MODIFY col1 VARCHAR(5);
```

```
Query OK, 0 rows affected (2.52 sec)
```

```
SHOW CREATE TABLE t1\G;
```

```
*************************** 1. row ***************************
       Table: t1
CREATE TABLE `t1` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `col1` varchar(5) DEFAULT NULL,
  PRIMARY KEY (`id`) /*T![clustered_index] CLUSTERED */
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin AUTO_INCREMENT=30001
1 row in set (0.00 sec)
```

注意：

- 当所变更的类型与已经存在的数据行产生冲突时，TiDB 会进行报错处理。在上述例子中，TiDB 将进行如下报错：

```
alter table t1 modify column col1 varchar(4); ERROR 1406 (22001): Data Too Long,
↪ field len 4, data len 5
```

- 由于和 Async Commit 功能兼容，DDL 在开始进入到 Reorg Data 前会有一定时间（约 2.5s）的等待处理：

```
Query OK, 0 rows affected (2.52 sec)
```

14.11.2.73.3 MySQL 兼容性

- 不支持修改主键列上需要 Reorg-Data 的类型，但是支持修改 Meta-Only 的类型。例如：

```
CREATE TABLE t (a int primary key);
ALTER TABLE t MODIFY COLUMN a VARCHAR(10);
ERROR 8200 (HY000): Unsupported modify column: column has primary key flag
```

```
CREATE TABLE t (a int primary key);
ALTER TABLE t MODIFY COLUMN a INT(10) UNSIGNED;
ERROR 8200 (HY000): Unsupported modify column: column has primary key flag
```

```
CREATE TABLE t (a int primary key);
ALTER TABLE t MODIFY COLUMN a bigint;
Query OK, 0 rows affected (0.01 sec)
```

- 不支持修改生成列的类型。例如：

```
CREATE TABLE t (a INT, b INT as (a+1));
ALTER TABLE t MODIFY COLUMN b VARCHAR(10);
ERROR 8200 (HY000): Unsupported modify column: column is generated
```

- 不支持修改分区表上的列类型。例如：

```
CREATE TABLE t (c1 INT, c2 INT, c3 INT) partition by range columns(c1) ( partition p0 values
    ↪  less than (10), partition p1 values less than (maxvalue));
ALTER TABLE t MODIFY COLUMN c1 DATETIME;
ERROR 8200 (HY000): Unsupported modify column: table is partition table
```

- 不支持部分数据类型（例如，部分时间类型、Bit、Set、Enum、JSON 等）的变更，因为 TiDB cast 函数与 MySQL 的行为有一些兼容性问题。例如：

```
CREATE TABLE t (a DECIMAL(13, 7));
ALTER TABLE t MODIFY COLUMN a DATETIME;
ERROR 8200 (HY000): Unsupported modify column: change from original type decimal(13,7) to
    ↪ datetime is currently unsupported yet
```

14.11.2.73.4 另请参阅

- CREATE TABLE
- SHOW CREATE TABLE
- ADD COLUMN
- DROP COLUMN
- CHANGE COLUMN

14.11.2.74 PREPARE

PREPARE 语句为服务器端预处理语句提供 SQL 接口。

14.11.2.74.1 语法图

```
PreparedStmt ::=
    'PREPARE' Identifier 'FROM' PrepareSQL

PrepareSQL ::=
    stringLit
|   UserVariable
```

14.11.2.74.2 示例

```
PREPARE mystmt FROM 'SELECT ? as num FROM DUAL';
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
SET @number = 5;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
EXECUTE mystmt USING @number;
```

```
+------+
| num  |
+------+
| 5    |
+------+
1 row in set (0.00 sec)
```

```
DEALLOCATE PREPARE mystmt;
```

```
Query OK, 0 rows affected (0.00 sec)
```

14.11.2.74.3 MySQL 兼容性

PREPARE 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.74.4 另请参阅

- EXECUTE
- DEALLOCATE

14.11.2.75 RECOVER TABLE

RECOVER TABLE 的功能是恢复被删除的表及其数据。在 DROP TABLE 后，在 GC life time 时间内，可以用 RECOVER TABLE 语句恢复被删除的表以及其数据。

14.11.2.75.1 语法

```
RECOVER TABLE table_name
```

```
RECOVER TABLE BY JOB JOB_ID
```

语法图

```
RecoverTableStmt ::=
    'RECOVER' 'TABLE' ( 'BY' 'JOB' Int64Num | TableName Int64Num? )

TableName ::=
    Identifier ( '.' Identifier )?

Int64Num ::= NUM

NUM ::= intLit
```

14.11.2.75.2 注意事项

如果删除表后并过了 GC lifetime，就不能再用 RECOVER TABLE 来恢复被删除的表了，执行 RECOVER TABLE 语句会返回类似错误：`snapshot is older than GC safe point 2019-07-10 13:45:57 +0800 CST`。

对于 3.0.0 及之后的 TiDB 版本，不推荐在使用 TiDB Binlog 的情况下使用 RECOVER TABLE 功能。

TiDB Binlog 在 3.0.1 支持 RECOVER TABLE 后，可在下面的情况下使用 RECOVER TABLE：

- 3.0.1+ 版本的 TiDB Binlog
- 主从集群都使用 TiDB 3.0
- 从集群 GC lifetime 一定要长于主集群（不过由于上下游同步的延迟，可能也会造成下游 recover 失败）

TiDB Binlog 同步错误处理

当使用 TiDB Binlog 同步工具时，上游 TiDB 使用 RECOVER TABLE 后，TiDB Binlog 可能会因为下面几个原因造成同步中断：

- 下游数据库不支持 RECOVER TABLE 语句。类似错误：`check the manual that corresponds to your MySQL server version for the right syntax to use near 'RECOVER TABLE table_name'`。
- 上下游数据库的 GC lifetime 不一样。类似错误：`snapshot is older than GC safe point 2019-07-10 13:45:57 +0800 CST`。
- 上下游数据库的同步延迟。类似错误：`snapshot is older than GC safe point 2019-07-10 13:45:57 +0800 CST`。

只能通过重新全量导入被删除的表来恢复 TiDB Binlog 的数据同步。

14.11.2.75.3 示例

- 根据表名恢复被删除的表。

```
DROP TABLE t;
```

```
RECOVER TABLE t;
```

根据表名恢复被删除的表需满足以下条件：

  - 最近 DDL JOB 历史中找到的第一个 DROP TABLE 操作，且
  - DROP TABLE 所删除的表的名称与 RECOVER TABLE 语句指定表名相同

- 根据删除表时的 DDL JOB ID 恢复被删除的表。

  如果第一次删除表 t 后，又新建了一个表 t，然后又把新建的表 t 删除了，此时如果想恢复最开始删除的表 t，就需要用到指定 DDL JOB ID 的语法了。

```
DROP TABLE t;
```

```
ADMIN SHOW DDL JOBS 1;
```

上面这个语句用来查找删除表 t 时的 DDL JOB ID，这里是 53：

```
+--------+---------+------------+------------+--------------+-----------+----------+-----------+-----------
    ↪
| JOB_ID | DB_NAME | TABLE_NAME | JOB_TYPE   | SCHEMA_STATE | SCHEMA_ID | TABLE_ID |
    ↪ ROW_COUNT | START_TIME                         | STATE  |
+--------+---------+------------+------------+--------------+-----------+----------+-----------+-----------
    ↪
| 53     | test    |            | drop table | none         | 1         | 41       | 0
    ↪         | 2019-07-10 13:23:18.277 +0800 CST | synced |
+--------+---------+------------+------------+--------------+-----------+----------+-----------+-----------
    ↪
```

```
RECOVER TABLE BY JOB 53;
```

根据删除表时的 DDL JOB ID 恢复被删除的表，会直接用 DDL JOB ID 找到被删除表进行恢复。如果指定的 DDL JOB ID 的 DDL JOB 不是 DROP TABLE 类型，会报错。

14.11.2.75.4 原理

TiDB 在删除表时，实际上只删除了表的元信息，并将需要删除的表数据（行数据和索引数据）写一条数据到 `mysql.gc_delete_range` 表。TiDB 后台的 GC Worker 会定期从 `mysql.gc_delete_range` 表中取出超过 GC lifetime 相关范围的 key 进行删除。

所以，RECOVER TABLE 只需要在 GC Worker 还没删除表数据前，恢复表的元信息并删除 `mysql.gc_delete_range` 表中相应的行记录就可以了。恢复表的元信息可以用 TiDB 的快照读实现。具体的快照读内容可以参考读取历史数据文档。

TiDB 中表的恢复是通过快照读获取表的元信息后，再走一次类似于 `CREATE TABLE` 的建表流程，所以 RECOVER TABLE 实际上也是一种 DDL。

14.11.2.75.5 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.76 RENAME INDEX

`ALTER TABLE .. RENAME INDEX` 语句用于对已有索引进行重命名。这在 TiDB 中是即时操作的，仅需更改元数据。

14.11.2.76.1 语法图

```
AlterTableStmt
         ::= 'ALTER' 'IGNORE'? 'TABLE' TableName RenameIndexSpec ( ',' RenameIndexSpec )*

RenameIndexSpec
         ::= 'RENAME' ( 'KEY' | 'INDEX' ) Identifier 'TO' Identifier
```

14.11.2.76.2 示例

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, c1 INT NOT NULL, INDEX col1 (c1));
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
SHOW CREATE TABLE t1;
```

```
*************************** 1. row ***************************
       Table: t1
Create Table: CREATE TABLE `t1` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `c1` int(11) NOT NULL,
  PRIMARY KEY (`id`),
  KEY `col1` (`c1`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin
1 row in set (0.00 sec)
```

```
ALTER TABLE t1 RENAME INDEX col1 TO c1;
```

```
Query OK, 0 rows affected (0.09 sec)
```

```
SHOW CREATE TABLE t1;
```

```
*************************** 1. row ***************************
       Table: t1
Create Table: CREATE TABLE `t1` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `c1` int(11) NOT NULL,
  PRIMARY KEY (`id`),
  KEY `c1` (`c1`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin
1 row in set (0.00 sec)
```

14.11.2.76.3 MySQL 兼容性

`RENAME INDEX` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.76.4 另请参阅

- SHOW CREATE TABLE
- CREATE INDEX
- DROP INDEX
- SHOW INDEX
- ALTER INDEX

14.11.2.77 RENAME TABLE

`RENAME TABLE` 语句用于对已有表进行重命名。

14.11.2.77.1 语法图

```
RenameTableStmt ::=
    'RENAME' 'TABLE' TableToTable ( ',' TableToTable )*

TableToTable ::=
    TableName 'TO' TableName
```

14.11.2.77.2 示例

```
CREATE TABLE t1 (a int);
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
SHOW TABLES;
```

```
+----------------+
| Tables_in_test |
+----------------+
| t1             |
+----------------+
1 row in set (0.00 sec)
```

```
RENAME TABLE t1 TO t2;
```

```
Query OK, 0 rows affected (0.08 sec)
```

```
SHOW TABLES;
```

```
+----------------+
| Tables_in_test |
+----------------+
| t2             |
+----------------+
1 row in set (0.00 sec)
```

14.11.2.77.3 MySQL 兼容性

RENAME TABLE 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.77.4 另请参阅

- CREATE TABLE
- SHOW TABLES
- ALTER TABLE

14.11.2.78 RENAME USER

RENAME USER 语句用于对已有用户进行重命名。

14.11.2.78.1 语法图

```
RenameUserStmt ::=
    'RENAME' 'USER' UserToUser ( ',' UserToUser )*

UserToUser ::=
    Username 'TO' Username

Username ::=
    StringName ('@' StringName | singleAtIdentifier)? | 'CURRENT_USER' OptionalBraces
```

14.11.2.78.2 示例

```
CREATE USER 'newuser' IDENTIFIED BY 'mypassword';
```

```
Query OK, 1 row affected (0.02 sec)
```

```
SHOW GRANTS FOR 'newuser';
```

```
+-----------------------------------+
| Grants for newuser@%              |
+-----------------------------------+
| GRANT USAGE ON *.* TO 'newuser'@'%' |
+-----------------------------------+
1 row in set (0.00 sec)
```

```
RENAME USER 'newuser' TO 'testuser';
```

```
Query OK, 0 rows affected (0.08 sec)
```

```
SHOW GRANTS FOR 'testuser';
```

```
+-----------------------------------+
| Grants for testuser@%             |
+-----------------------------------+
| GRANT USAGE ON *.* TO 'testuser'@'%' |
+-----------------------------------+
1 row in set (0.00 sec)
```

```
SHOW GRANTS FOR 'newuser';
```

```
ERROR 1141 (42000): There is no such grant defined for user 'newuser' on host '%'
```

14.11.2.78.3 MySQL 兼容性

RENAME USER 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.78.4 另请参阅

- CREATE USER
- SHOW GRANTS
- DROP USER

14.11.2.79 REPLACE

从语义上看，REPLACE 语句是 DELETE 语句和 INSERT 语句的结合，可用于简化应用程序代码。

14.11.2.79.1 语法图

```
ReplaceIntoStmt ::=
    'REPLACE' PriorityOpt IntoOpt TableName PartitionNameListOpt InsertValues

PriorityOpt ::=
    ( 'LOW_PRIORITY' | 'HIGH_PRIORITY' | 'DELAYED' )?

IntoOpt ::= 'INTO'?

TableName ::=
    Identifier ( '.' Identifier )?

PartitionNameListOpt ::=
    ( 'PARTITION' '(' Identifier ( ',' Identifier )* ')' )?

InsertValues ::=
    '(' ( ColumnNameListOpt ')' ( ValueSym ValuesList | SelectStmt | '(' SelectStmt ')' |
        ↪ UnionStmt ) | SelectStmt ')' )
|   ValueSym ValuesList
|   SelectStmt
|   UnionStmt
|   'SET' ColumnSetValue? ( ',' ColumnSetValue )*
```

14.11.2.79.2 示例

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, c1 INT NOT NULL);
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
INSERT INTO t1 (c1) VALUES (1), (2), (3);
```

```
Query OK, 3 rows affected (0.02 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
SELECT * FROM t1;
```

```
+----+----+
| id | c1 |
+----+----+
|  1 |  1 |
|  2 |  2 |
|  3 |  3 |
+----+----+
3 rows in set (0.00 sec)
```

```
REPLACE INTO t1 (id, c1) VALUES(3, 99);
```

```
Query OK, 2 rows affected (0.01 sec)
```

```
SELECT * FROM t1;
```

```
+----+----+
| id | c1 |
+----+----+
|  1 |  1 |
|  2 |  2 |
|  3 | 99 |
+----+----+
3 rows in set (0.00 sec)
```

14.11.2.79.3 MySQL 兼容性

REPLACE 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.79.4 另请参阅

- DELETE
- INSERT
- SELECT
- UPDATE

14.11.2.80 RESTORE

RESTORE 语句用于执行分布式恢复，把BACKUP 语句生成的备份文件恢复到 TiDB 集群中。

RESTORE 语句使用的引擎与BR 相同，但恢复过程是由 TiDB 本身驱动，而非单独的 BR 工具。BR 工具的优势和警告也适用于 RESTORE 语句。需要注意的是，RESTORE 语句目前不遵循 ACID 原则。

执行 RESTORE 语句前，确保集群已满足以下要求：

- 集群处于“下线”状态，当前的 TiDB 会话是唯一在访问待恢复表的活跃 SQL 连接。
- 执行全量恢复时，确保即将恢复的表不存在于集群中，因为现有的数据可能被覆盖，从而导致数据与索引不一致。
- 执行增量恢复时，表的状态应该与创建备份时 LAST_BACKUP 时间戳的状态完全一致。

执行 RESTORE 需要 RESTORE_ADMIN 或 SUPER 权限。此外，执行恢复操作的 TiDB 节点和集群中的所有 TiKV 节点都必须有对目标存储的读权限。

RESTORE 语句开始执行后将会被阻塞，直到整个恢复任务完成、失败或取消。因此，执行 RESTORE 时需要准备一个持久的连接。如需取消任务，可执行KILL TIDB QUERY 语句。

一次只能执行一个 BACKUP 和 RESTORE 任务。如果 TiDB server 上已经在执行一个 BACKUP 或 RESTORE 语句，新的 RESTORE 将等待前面所有的任务完成后再执行。

RESTORE 只能在“tikv”存储引擎上使用，如果使用“unistore”存储引擎，RESTORE 操作会失败。

14.11.2.80.1 语法图

```
RestoreStmt ::=
    "RESTORE" BRIETables "FROM" stringLit RestoreOption*

BRIETables ::=
    "DATABASE" ( '*' | DBName (',' DBName)* )
|   "TABLE" TableNameList

RestoreOption ::=
    "RATE_LIMIT" '='? LengthNum "MB" '/' "SECOND"
|   "CONCURRENCY" '='? LengthNum
|   "CHECKSUM" '='? Boolean
|   "SEND_CREDENTIALS_TO_TIKV" '='? Boolean

Boolean ::=
    NUM | "TRUE" | "FALSE"
```

14.11.2.80.2 示例

从备份文件恢复

```
RESTORE DATABASE * FROM 'local:///mnt/backup/2020/04/';
```

```
+--
    ↪ ---------------------------+-----------+----------+--------------------+--------------------+
    ↪
| Destination                  | Size      | BackupTS | Queue Time         | Execution Time
    ↪      |
+--
    ↪ ---------------------------+-----------+----------+--------------------+--------------------+
    ↪
| local:///mnt/backup/2020/04/ | 248665063 | 0        | 2020-04-21 17:16:55 | 2020-04-21 17:16:55
    ↪  |
+--
    ↪ ---------------------------+-----------+----------+--------------------+--------------------+
    ↪
1 row in set (28.961 sec)
```

上述示例中，所有数据均从本地的备份文件中恢复到集群中。RESTORE 从 SST 文件里读取数据，SST 文件存储在所有 TiDB 和 TiKV 节点的 `/mnt/backup/2020/04/` 目录下。

输出结果的第一行描述如下：

列名	描述
Destination	读取的目标存储 URL
Size	备份文件的总大小，单位为字节
BackupTS	不适用
Queue Time	RESTORE 任务开始排队的时间戳（当前时区）
Execution Time	RESTORE 任务开始执行的时间戳（当前时区）

部分恢复

你可以指定恢复部分数据库或部分表数据。如果备份文件中缺失了某些数据库或表，缺失的部分将被忽略。此时，RESTORE 语句不进行任何操作即完成执行。

```
RESTORE DATABASE `test` FROM 'local:///mnt/backup/2020/04/';
```

```
RESTORE TABLE `test`.`sbtest01`, `test`.`sbtest02` FROM 'local:///mnt/backup/2020/04/';
```

外部存储

BR 支持从 Amazon S3 或 Google Cloud Storage (GCS) 恢复数据：

```
RESTORE DATABASE * FROM 's3://example-bucket-2020/backup-05/';
```

有关详细的 URL 语法，见外部存储 URI 格式。

当运行在云环境中时，不能分发凭证，可设置 SEND_CREDENTIALS_TO_TIKV 选项为 FALSE：

```
RESTORE DATABASE * FROM 's3://example-bucket-2020/backup-05/'
    SEND_CREDENTIALS_TO_TIKV = FALSE;
```

性能调优

如果你需要减少网络带宽占用，可以通过 RATE_LIMIT 来限制每个 TiKV 节点的平均下载速度。

默认情况下，每个 TiKV 节点上运行 128 个恢复线程。可以通过 CONCURRENCY 选项来调整这个值。

在恢复完成之前，RESTORE 将对备份文件中的数据进行校验，以验证数据的正确性。如果你确信无需进行校验，可以通过 CHECKSUM 选项禁用这一步骤。

```
RESTORE DATABASE * FROM 's3://example-bucket-2020/backup-06/'
    RATE_LIMIT = 120 MB/SECOND
    CONCURRENCY = 64
    CHECKSUM = FALSE;
```

增量恢复

增量恢复没有特殊的语法。TiDB 将识别备份文件属于全量备份或增量备份，然后执行对应的恢复操作，用户只需按照正确顺序进行增量恢复。

假设按照如下方式创建一个备份任务：

```
BACKUP DATABASE `test` TO 's3://example-bucket/full-backup'  SNAPSHOT = 413612900352000;
BACKUP DATABASE `test` TO 's3://example-bucket/inc-backup-1' SNAPSHOT = 414971854848000
    ↪ LAST_BACKUP = 413612900352000;
BACKUP DATABASE `test` TO 's3://example-bucket/inc-backup-2' SNAPSHOT = 416353458585600
    ↪ LAST_BACKUP = 414971854848000;
```

在恢复备份时，需要采取同样的顺序：

```
RESTORE DATABASE * FROM 's3://example-bucket/full-backup';
RESTORE DATABASE * FROM 's3://example-bucket/inc-backup-1';
RESTORE DATABASE * FROM 's3://example-bucket/inc-backup-2';
```

14.11.2.80.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.80.4 另请参阅

- BACKUP
- SHOW RESTORES

14.11.2.81 REVOKE <privileges>

REVOKE <privileges> 语句用于删除已有用户的权限。执行 REVOKE <privileges> 语句需要拥有分配的权限，并且拥有 GRANT OPTION 权限。

14.11.2.81.1 语法图

```
GrantStmt ::=
    'GRANT' PrivElemList 'ON' ObjectType PrivLevel 'TO' UserSpecList RequireClauseOpt
        ↪ WithGrantOptionOpt

PrivElemList ::=
    PrivElem ( ',' PrivElem )*

PrivElem ::=
    PrivType ( '(' ColumnNameList ')' )?

PrivType ::=
    'ALL' 'PRIVILEGES'?
|   'ALTER' 'ROUTINE'?
|   'CREATE' ( 'USER' | 'TEMPORARY' 'TABLES' | 'VIEW' | 'ROLE' | 'ROUTINE' )?
|    'TRIGGER'
|   'DELETE'
|    'DROP' 'ROLE'?
|    'PROCESS'
|    'EXECUTE'
|   'INDEX'
|   'INSERT'
|   'SELECT'
|   'SUPER'
|    'SHOW' ( 'DATABASES' | 'VIEW' )
|   'UPDATE'
|   'GRANT' 'OPTION'
|   'REFERENCES'
|   'REPLICATION' ( 'SLAVE' | 'CLIENT' )
|   'USAGE'
|    'RELOAD'
|   'FILE'
|   'CONFIG'
|   'LOCK' 'TABLES'
|    'EVENT'
|   'SHUTDOWN'

ObjectType ::=
    'TABLE'?

PrivLevel ::=
    '*' ( '.' '*' )?
|    Identifier ( '.' ( '*' | Identifier ) )?

UserSpecList ::=
```

```
    UserSpec ( ',' UserSpec )*
```

14.11.2.81.2 示例

```
CREATE USER 'newuser' IDENTIFIED BY 'mypassword';
```

```
Query OK, 1 row affected (0.02 sec)
```

```
GRANT ALL ON test.* TO 'newuser';
```

```
Query OK, 0 rows affected (0.03 sec)
```

```
SHOW GRANTS FOR 'newuser';
```

```
+-------------------------------------------------+
| Grants for newuser@%                            |
+-------------------------------------------------+
| GRANT USAGE ON *.* TO 'newuser'@'%'             |
| GRANT ALL PRIVILEGES ON test.* TO 'newuser'@'%' |
+-------------------------------------------------+
2 rows in set (0.00 sec)
```

```
REVOKE ALL ON test.* FROM 'newuser';
```

```
Query OK, 0 rows affected (0.03 sec)
```

```
SHOW GRANTS FOR 'newuser';
```

```
+-----------------------------------+
| Grants for newuser@%              |
+-----------------------------------+
| GRANT USAGE ON *.* TO 'newuser'@'%' |
+-----------------------------------+
1 row in set (0.00 sec)
```

```
DROP USER 'newuser';
```

```
Query OK, 0 rows affected (0.14 sec)
```

```
SHOW GRANTS FOR 'newuser';
```

```
ERROR 1141 (42000): There is no such grant defined for user 'newuser' on host '%'
```

14.11.2.81.3 MySQL 兼容性

`REVOKE <privileges>` 语句执行成功后，在 TiDB 中语句执行的结果会在当前连接立即生效，而 MySQL 中部分权限的结果需要等到之后的连接才生效。见 TiDB #39356。

14.11.2.81.4 另请参阅

- GRANT <privileges>
- SHOW GRANTS
- Privilege Management

14.11.2.82 REVOKE <role>

`REVOKE <role>` 语句用于从指定的用户（或用户列表）中收回之前授予的角色。

14.11.2.82.1 语法图

```
RevokeRoleStmt ::=
    'REVOKE' RolenameList 'FROM' UsernameList

RolenameList ::=
    Rolename ( ',' Rolename )*

UsernameList ::=
    Username ( ',' Username )*
```

14.11.2.82.2 示例

以 root 用户连接 TiDB：

```
mysql -h 127.0.0.1 -P 4000 -u root
```

创建新角色 analyticsteam 和新用户 jennifer：

```
CREATE ROLE analyticsteam;
Query OK, 0 rows affected (0.02 sec)

GRANT SELECT ON test.* TO analyticsteam;
Query OK, 0 rows affected (0.02 sec)

CREATE USER jennifer;
Query OK, 0 rows affected (0.01 sec)

GRANT analyticsteam TO jennifer;
Query OK, 0 rows affected (0.01 sec)
```

以 jennifer 用户连接 TiDB：

```
mysql -h 127.0.0.1 -P 4000 -u jennifer
```

需要注意的是，默认情况下，用户 jennifer 需要执行 SET ROLE analyticsteam 语句才能使用与角色相关联的权限：

```
SHOW GRANTS;
+-----------------------------------------------+
| Grants for User                               |
+-----------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'          |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+-----------------------------------------------+
2 rows in set (0.00 sec)

SHOW TABLES in test;
ERROR 1044 (42000): Access denied for user 'jennifer'@'%' to database 'test'
SET ROLE analyticsteam;
Query OK, 0 rows affected (0.00 sec)

SHOW GRANTS;
+-----------------------------------------------+
| Grants for User                               |
+-----------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'          |
| GRANT SELECT ON test.* TO 'jennifer'@'%'      |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+-----------------------------------------------+
3 rows in set (0.00 sec)

SHOW TABLES IN test;
+----------------+
| Tables_in_test |
+----------------+
| t1             |
+----------------+
1 row in set (0.00 sec)
```

以 root 用户连接 TiDB：

```
mysql -h 127.0.0.1 -P 4000 -u root
```

执行 SET DEFAULT ROLE 语句将用户 jennifer 与 analyticsteam 角色相关联：

```
SET DEFAULT ROLE analyticsteam TO jennifer;
Query OK, 0 rows affected (0.02 sec)
```

以 jennifer 用户连接 TiDB：

```
mysql -h 127.0.0.1 -P 4000 -u jennifer
```

此时 jennifer 用户无需执行 SET ROLE 语句就能拥有 analyticsteam 角色相关联的权限：

```
SHOW GRANTS;
+-------------------------------------------+
| Grants for User                           |
+-------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'      |
| GRANT SELECT ON test.* TO 'jennifer'@'%'  |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+-------------------------------------------+
3 rows in set (0.00 sec)

SHOW TABLES IN test;
+----------------+
| Tables_in_test |
+----------------+
| t1             |
+----------------+
1 row in set (0.00 sec)
```

以 root 用户连接 TiDB：

```
mysql -h 127.0.0.1 -P 4000 -u root
```

收回角色 analyticsteam：

```
REVOKE analyticsteam FROM jennifer;
Query OK, 0 rows affected (0.01 sec)
```

以 jennifer 用户连接 TiDB：

```
mysql -h 127.0.0.1 -P 4000 -u jennifer
```

查看 jennifer 用户的权限：

```
SHOW GRANTS;
+--------------------------------------+
| Grants for User                      |
+--------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%' |
+--------------------------------------+
1 row in set (0.00 sec)
```

14.11.2.82.3 MySQL 兼容性

`REVOKE <role>` 语句与 MySQL 8.0 的角色功能完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.82.4 另请参阅

- `CREATE ROLE`
- `DROP ROLE`
- `GRANT <role>`
- `SET ROLE`
- `SET DEFAULT ROLE`
- 基于角色的访问控制

14.11.2.83 ROLLBACK

`ROLLBACK` 语句用于还原 TiDB 内当前事务中的所有更改，作用与 `COMMIT` 语句相反。

14.11.2.83.1 语法图

```
RollbackStmt ::=
    'ROLLBACK' CompletionTypeWithinTransaction?

CompletionTypeWithinTransaction ::=
    'AND' ( 'CHAIN' ( 'NO' 'RELEASE' )? | 'NO' 'CHAIN' ( 'NO'? 'RELEASE' )? )
|   'NO'? 'RELEASE'
```

14.11.2.83.2 示例

```
CREATE TABLE t1 (a INT NOT NULL PRIMARY KEY);
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
BEGIN;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
INSERT INTO t1 VALUES (1);
```

```
Query OK, 1 row affected (0.00 sec)
```

```
ROLLBACK;
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
SELECT * FROM t1;
```

```
Empty set (0.01 sec)
```

14.11.2.83.3 MySQL 兼容性

- TiDB 解析但忽略 ROLLBACK AND [NO] RELEASE 语法。在 MySQL 中，使用该语法可在回滚事务后立即断开客户端会话。在 TiDB 中，建议使用客户端程序的 mysql_close() 来实现该功能。
- TiDB 解析但忽略 ROLLBACK AND [NO] CHAIN 语法。在 MySQL 中，使用该语法可在回滚当前事务时立即以相同的隔离级别开启新事务。在 TiDB 中，推荐直接开启新事务。

14.11.2.83.4 另请参阅

- SAVEPOINT
- COMMIT
- BEGIN
- START TRANSACTION

14.11.2.84 SAVEPOINT

SAVEPOINT 是 TiDB 从 v6.2.0 开始支持的特性，语法如下：

```
SAVEPOINT identifier
ROLLBACK TO [SAVEPOINT] identifier
RELEASE SAVEPOINT identifier
```

> **警告：**
>
> SAVEPOINT 特性不支持与 TiDB Binlog 一起使用，也不支持在关闭tidb_constraint_check_in_place_pessimistic 的悲观事务中使用。

- SAVEPOINT 语句用于在当前事务中，设置一个指定名字保存点。如果已经存在相同名字的保存点，就删除已有的保存点并设置新的保存点。
- ROLLBACK TO SAVEPOINT 语句将事务回滚到指定名称的事务保存点，而不终止该事务。当前事务在设置保存点后，对表数据所做的修改将在回滚中撤销，且删除事务保存点之后的所有保存点。在悲观事务中，对于已经持有的悲观锁不会回滚，而是在事务结束时才释放。

  如果 ROLLBACK TO SAVEPOINT 语句中指定名称的保存点不存在，则会返回以下错误信息：

  ```
  ERROR 1305 (42000): SAVEPOINT identifier does not exist
  ```

- RELEASE SAVEPOINT 语句将从当前事务中删除指定名称及之后的所有保存点，而不会提交或回滚当前事务。如果指定名称的保存点不存在，则会返回以下错误信息：

  ```
  ERROR 1305 (42000): SAVEPOINT identifier does not exist
  ```

  当事务提交或者回滚后，事务中所有保存点都会被删除。

14.11.2.84.1 示例

创建表 t1：

```
CREATE TABLE t1 (a int NOT NULL PRIMARY KEY);
```

```
Query OK, 0 rows affected (0.12 sec)
```

开始当前事务：

```
BEGIN;
```

```
Query OK, 0 rows affected (0.00 sec)
```

向表中插入数据并设置保存点 sp1：

```
INSERT INTO t1 VALUES (1);
```

```
Query OK, 1 row affected (0.00 sec)
```

```
SAVEPOINT sp1;
```

```
Query OK, 0 rows affected (0.01 sec)
```

向表中再次插入数据并设置保存点 sp2：

```
INSERT INTO t1 VALUES (2);
```

```
Query OK, 1 row affected (0.00 sec)
```

```
SAVEPOINT sp2;
```

```
Query OK, 0 rows affected (0.01 sec)
```

释放保存点 sp2：

```
RELEASE SAVEPOINT sp2;
```

```
Query OK, 0 rows affected (0.01 sec)
```

回滚至保存点 sp1：

```
ROLLBACK TO SAVEPOINT sp1;
```

```
Query OK, 0 rows affected (0.01 sec)
```

提交事务并查询表格，发现表中仅有 sp1 前插入的数据：

```
COMMIT;
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
SELECT * FROM t1;
```

```
+---+
| a |
+---+
| 1 |
+---+
1 row in set
```

14.11.2.84.2 MySQL 兼容性

使用 `ROLLBACK TO SAVEPOINT` 语句将事物回滚到指定保存点时，MySQL 会释放该保存点之后才持有的锁，但在 TiDB 悲观事务中，不会立即释放该保存点之后才持有的锁，而是等到事务提交或者回滚时，才释放全部持有的锁。

14.11.2.84.3 另请参阅

- COMMIT
- ROLLBACK
- START TRANSACTION
- TiDB 乐观事务模型
- TiDB 悲观事务模式

14.11.2.85 SELECT

`SELECT` 语句用于从 TiDB 读取数据。

14.11.2.85.1 语法图

SelectStmt:

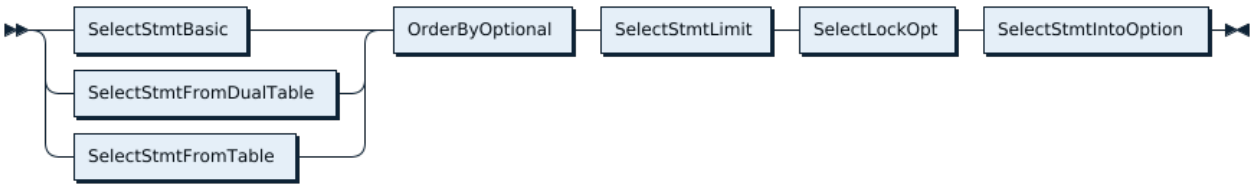


图 356: SelectStmt

FromDual:

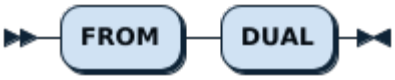


图 357: FromDual

WhereClauseOptional:

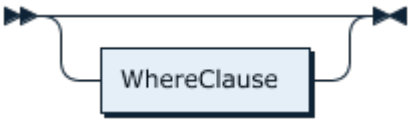


图 358: WhereClauseOptional

SelectStmtOpts:

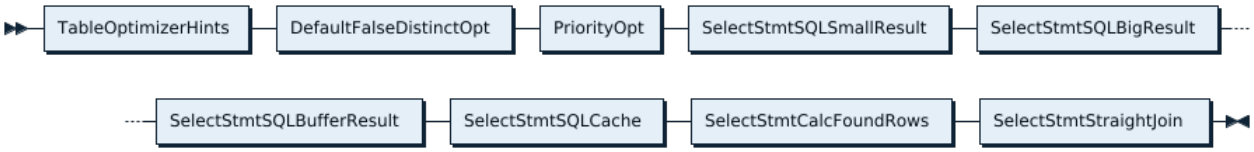


图 359: SelectStmtOpts

SelectStmtFieldList:

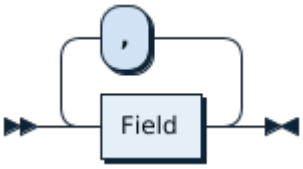


图 360: SelectStmtFieldList

TableRefsClause:

```
TableRefsClause ::=
    TableRef AsOfClause? ( ',' TableRef AsOfClause? )*

AsOfClause ::=
    'AS' 'OF' 'TIMESTAMP' Expression
```

SelectStmtGroup:



图 361: SelectStmtGroup

HavingClause:

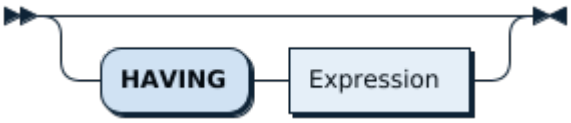


图 362: HavingClause

OrderByOptional:

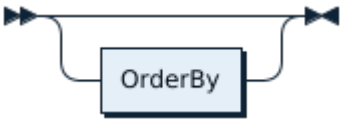


图 363: OrderByOptional

SelectStmtLimit:

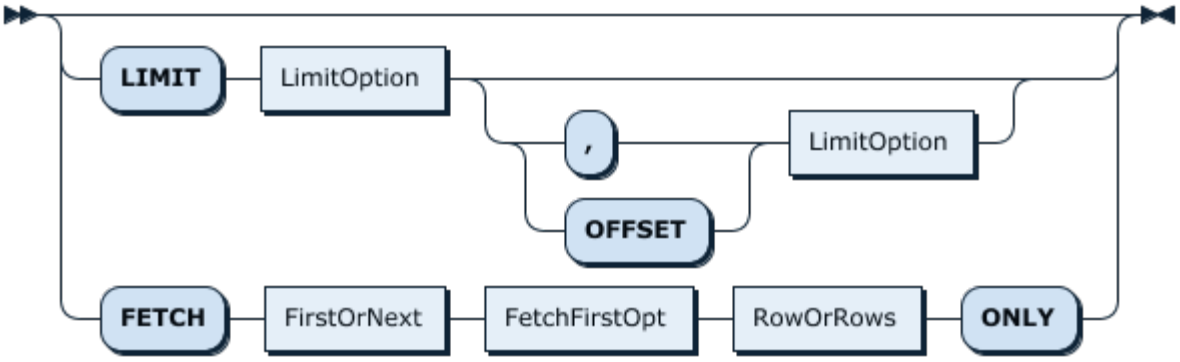


图 364: SelectStmtLimit

FirstOrNext:

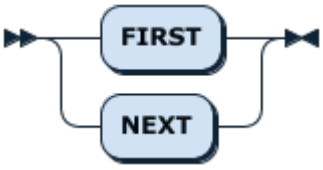


图 365: FirstOrNext

FetchFirstOpt:



图 366: FetchFirstOpt

RowOrRows:

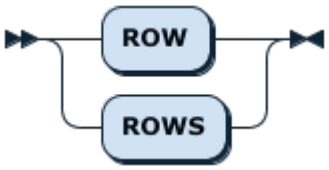


图 367: RowOrRows

SelectLockOpt:

```
SelectLockOpt ::=
    ( ( 'FOR' 'UPDATE' ( 'OF' TableList )? 'NOWAIT'? )
|   ( 'LOCK' 'IN' 'SHARE' 'MODE' ) )?

TableList ::=
    TableName ( ',' TableName )*
```

WindowClauseOptional

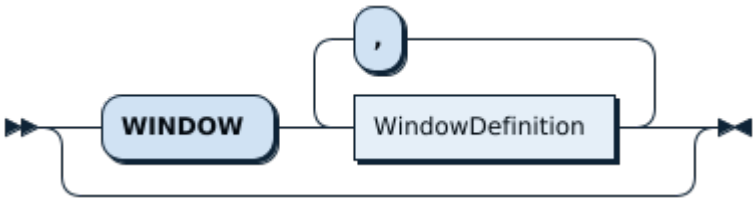


图 368: WindowClauseOptional

TableSampleOpt

```
TableSampleOpt ::=
    'TABLESAMPLE' 'REGIONS()'
```

14.11.2.85.2 语法元素说明

语法元素	说明
TableOptimizerHints	用于控制优化器行为的 Hint，具体可参见Optimizer Hints
ALL、DISTINCT、DISTINCTROW	查询结果集中可能会包含重复值。指定 DISTINCT/DISTINCTROW 则在查询结果中过滤掉重复的行；指定 ALL 则列出所有的行。默认为 ALL。
HIGH_PRIORITY	该语句为高优先级语句，TiDB 在执行阶段会优先处理这条语句
SQL_CALC_FOUND_ROWS	TiDB 不支持该语法，并报错（若tidb_enable_noop_functions 值设为 1 则不会报错）
SQL_CACHE、 SQL_NO_CACHE	是否把请求结果缓存到 TiKV (RocksDB) 的 BlockCache 中。对于一次性的大数据量的查询，比如 count(*) 查询，为了避免冲掉 BlockCache 中用户的热点数据，建议填上 SQL_NO_CACHE

语法元素	说明
STRAIGHT_JOIN	STRAIGHT_JOIN 会强制优化器按照 FROM 子句中所使用的表的顺序做联合查询。当优化器选择的 Join 顺序并不优秀时，你可以使用这个语法来加速查询的执行
select_expr	投影操作列表，一般包括列名、表达式，或者是用 ‘*’ 表示全部列
FROM table_references	表示数据来源，数据来源可以是一个表 (select * from t;) 或者是多个表 (select * from t1 join t2;) 或者是 0 个表 (select 1+1 from dual;，等价于 select 1+1;)
WHERE where_condition	Where 子句用于设置过滤条件，查询结果中只会包含满足条件的数据
GROUP BY	GroupBy 子句用于对查询结果集进行分组
HAVING where_condition	Having 子句与 Where 子句作用类似，Having 子句可以让过滤 GroupBy 后的各种数据，Where 子句用于在聚合前过滤记录。
ORDER BY	OrderBy 子句用于指定结果排序顺序，可以按照列、表达式或者是 select_expr 列表中某个位置的字段进行排序。
LIMIT	Limit 子句用于限制结果条数。Limit 接受一个或两个数字参数，如果只有一个参数，那么表示返回数据的最大行数；如果是两个参数，那么第一个参数表示返回数据的第一行的偏移量 (第一行数据的偏移量是 0)，第二个参数指定返回数据的最大条目数。另支持 FETCH FIRST/NEXT n ROW/ROWS ONLY 语法，与 LIMIT n 效果相同，其中 n 可省略，省略时与 LIMIT 1 效果相同。
Window window_definition	窗口函数的相关语法，用来进行一些分析型计算的操作，详情可见窗口函数
FOR UPDATE	对查询结果集所有行上锁 (对于在查询条件内，但是不在结果集的行，将不会加锁，如事务启动后由其他事务写入的行)，以监测其他事务对这些的并发修改。使用乐观事务模型时，语句执行期间不会检测锁，因此，不会像 PostgreSQL 之类的数据库一样，在当前事务结束前阻止其他事务执行 UPDATE、DELETE 和 SELECT FOR UPDATE。在事务的提交阶段 SELECT FOR UPDATE 读到的行，也会进行两阶段提交，因此，它们也可以参与事务冲突检测。如发生写入冲突，那么包含 SELECT FOR UPDATE 语句的事务会提交失败。如果没有冲突，事务将成功提交，当提交结束时，这些被加锁的行，会产生一个新版本，可以让其他尚未提交的事务，在将来提交时发现写入冲突。若使用悲观事务，则行为与其他数据库基本相同，不一致之处参考和 MySQL InnoDB 的差异。TiDB 支持 FOR UPDATE NOWAIT 语法，详情可见TiDB 中悲观事务模式的行为。
LOCK IN SHARE MODE	TiDB 出于兼容性解析这个语法，但是不做任何处理
TABLESAMPLE	从表中获取一些行的样本数据。

14.11.2.85.3 示例

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, c1 INT NOT NULL);
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
INSERT INTO t1 (c1) VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 5 rows affected (0.03 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
SELECT * FROM t1;
```

```
+----+----+
| id | c1 |
+----+----+
|  1 |  1 |
|  2 |  2 |
|  3 |  3 |
|  4 |  4 |
|  5 |  5 |
+----+----+
5 rows in set (0.00 sec)
```

下面这个例子使用 `tiup bench tpcc prepare` 生成的数据，其中第一个查询展示了 `TABLESAMPLE` 的用法。

```
mysql> SELECT AVG(s_quantity), COUNT(s_quantity) FROM stock TABLESAMPLE REGIONS();
+-----------------+------------------+
| AVG(s_quantity) | COUNT(s_quantity) |
+-----------------+------------------+
|         59.5000 |                4 |
+-----------------+------------------+
1 row in set (0.00 sec)

mysql> SELECT AVG(s_quantity), COUNT(s_quantity) FROM stock;
+-----------------+------------------+
| AVG(s_quantity) | COUNT(s_quantity) |
+-----------------+------------------+
|         54.9729 |           1000000 |
+-----------------+------------------+
1 row in set (0.52 sec)
```

14.11.2.85.4 MySQL 兼容性

- 不支持 `SELECT ... INTO @variable` 语法。
- 不支持 `SELECT ... GROUP BY ... WITH ROLLUP` 语法。
- 不支持 MySQL 5.7 中支持的 `SELECT .. GROUP BY expr` 语法，而是匹配 MySQL 8.0 的行为，不按照默认的顺序进行排序。
- `SELECT ... TABLESAMPLE ...` 是 TiDB 的扩展语法，MySQL 不支持该语法。

14.11.2.85.5 另请参阅

- INSERT
- DELETE
- UPDATE
- REPLACE

14.11.2.86 SET DEFAULT ROLE

`SET DEFAULT ROLE` 语句默认设置将特定角色应用于用户。因此，用户不必执行 `SET ROLE <rolename>` 或 `SET ROLE ALL` 语句，也可以自动具有与角色相关联的权限。

14.11.2.86.1 语法图

SetDefaultRoleStmt:



图 369: SetDefaultRoleStmt

SetDefaultRoleOpt:

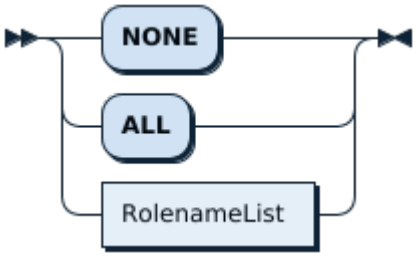


图 370: SetDefaultRoleOpt

RolenameList:

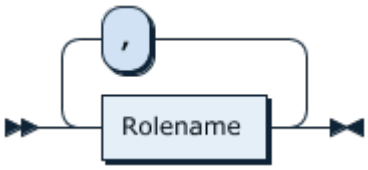


图 371: RolenameList

UsernameList:

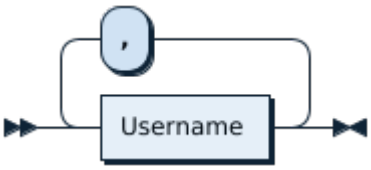


图 372: UsernameList

14.11.2.86.2 示例

以 root 用户连接 TiDB：

```
mysql -h 127.0.0.1 -P 4000 -u root
```

创建新角色 analyticsteam 和新用户 jennifer：

```
CREATE ROLE analyticsteam;
Query OK, 0 rows affected (0.02 sec)

GRANT SELECT ON test.* TO analyticsteam;
Query OK, 0 rows affected (0.02 sec)

CREATE USER jennifer;
Query OK, 0 rows affected (0.01 sec)

GRANT analyticsteam TO jennifer;
Query OK, 0 rows affected (0.01 sec)
```

以 jennifer 用户连接 TiDB：

```
mysql -h 127.0.0.1 -P 4000 -u jennifer
```

需要注意的是，默认情况下，用户 jennifer 需要执行 SET ROLE analyticsteam 语句才能使用与 analyticsteam 角色相关联的权限：

```
SHOW GRANTS;
+-----------------------------------------------+
| Grants for User                               |
+-----------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'          |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+-----------------------------------------------+
2 rows in set (0.00 sec)

SHOW TABLES in test;
ERROR 1044 (42000): Access denied for user 'jennifer'@'%' to database 'test'
SET ROLE analyticsteam;
Query OK, 0 rows affected (0.00 sec)
```

```
SHOW GRANTS;
+-------------------------------------------+
| Grants for User                           |
+-------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'      |
| GRANT Select ON test.* TO 'jennifer'@'%'  |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+-------------------------------------------+
3 rows in set (0.00 sec)

SHOW TABLES IN test;
+----------------+
| Tables_in_test |
+----------------+
| t1             |
+----------------+
1 row in set (0.00 sec)
```

以 root 用户连接 TiDB：

```
mysql -h 127.0.0.1 -P 4000 -u root
```

执行 SET DEFAULT ROLE 语句将用户 jennifer 与 analyticsteam 角色相关联：

```
SET DEFAULT ROLE analyticsteam TO jennifer;
Query OK, 0 rows affected (0.02 sec)
```

以 jennifer 用户连接 TiDB：

```
mysql -h 127.0.0.1 -P 4000 -u jennifer
```

此时 jennifer 用户无需执行 SET ROLE 语句就能拥有 analyticsteam 角色相关联的权限：

```
SHOW GRANTS;
+-------------------------------------------+
| Grants for User                           |
+-------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'      |
| GRANT Select ON test.* TO 'jennifer'@'%'  |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+-------------------------------------------+
3 rows in set (0.00 sec)

SHOW TABLES IN test;
+----------------+
| Tables_in_test |
```

```
+----------------+
| t1             |
+----------------+
1 row in set (0.00 sec)
```

SET DEFAULT ROLE 语句不会自动将相关角色授予 (GRANT) 用户。若尝试为 jennifer 尚未被授予的角色执行 SET DEFAULT ROLE 语句会导致以下错误：

```
SET DEFAULT ROLE analyticsteam TO jennifer;
ERROR 3530 (HY000): `analyticsteam`@`%` is is not granted to jennifer@%
```

14.11.2.86.3 MySQL 兼容性

SET DEFAULT ROLE 语句与 MySQL 8.0 的角色功能完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.86.4 另请参阅

- CREATE ROLE
- DROP ROLE
- GRANT <role>
- REVOKE <role>
- SET ROLE
- 基于角色的访问控制

14.11.2.87 SET [NAMES|CHARACTER SET]

SET NAMES，SET CHARACTER SET 和 SET CHARSET 语句用于修改当前连接的变量 character_set_client，character_set_results 和 character_set_connection。

14.11.2.87.1 语法图

SetNamesStmt:



图 373: SetNamesStmt

VariableAssignmentList:

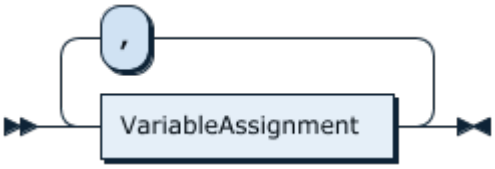


图 374: VariableAssignmentList

VariableAssignment:

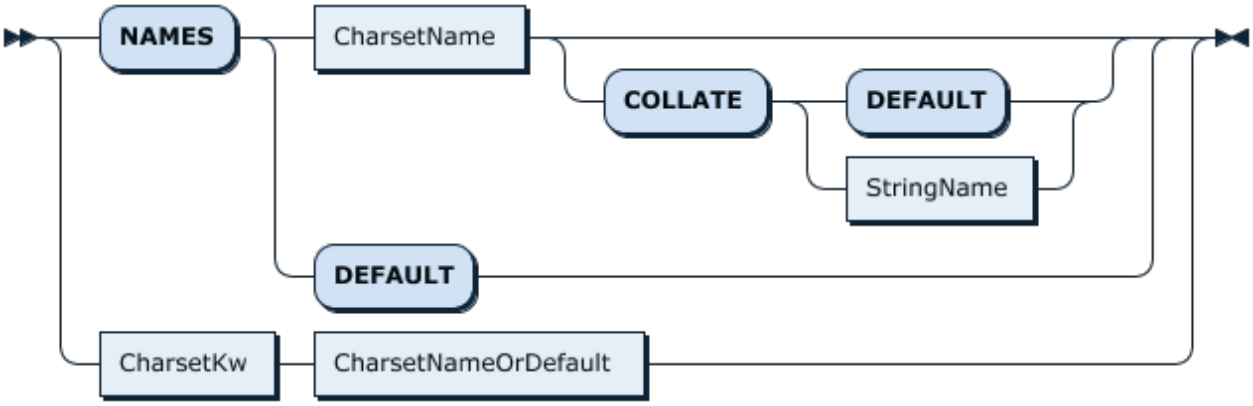


图 375: VariableAssignment

CharsetName:



图 376: CharsetName

StringName:



图 377: StringName

CharsetKw:

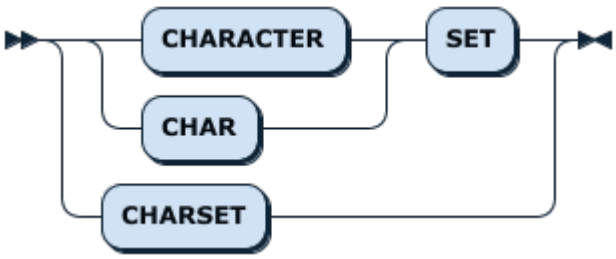


图 378: CharsetKw

CharsetNameOrDefault:

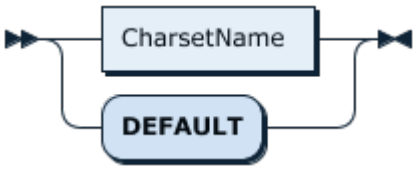


图 379: CharsetNameOrDefault

14.11.2.87.2 示例

```
SHOW VARIABLES LIKE 'character_set%';
```

```
+--------------------------+----------------------------------------------------------+
| Variable_name            | Value                                                    |
+--------------------------+----------------------------------------------------------+
| character_sets_dir       | /usr/local/mysql-5.6.25-osx10.8-x86_64/share/charsets/ |
| character_set_connection | utf8mb4                                                  |
| character_set_system     | utf8                                                     |
| character_set_results    | utf8mb4                                                  |
| character_set_client     | utf8mb4                                                  |
| character_set_database   | utf8mb4                                                  |
| character_set_filesystem | binary                                                   |
| character_set_server     | utf8mb4                                                  |
+--------------------------+----------------------------------------------------------+
8 rows in set (0.01 sec)
```

```
SET NAMES utf8;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
SHOW VARIABLES LIKE 'character_set%';
```

```
+--------------------------+----------------------------------------------------------+
| Variable_name            | Value                                                    |
+--------------------------+----------------------------------------------------------+
| character_sets_dir       | /usr/local/mysql-5.6.25-osx10.8-x86_64/share/charsets/ |
| character_set_connection | utf8                                                     |
| character_set_system     | utf8                                                     |
| character_set_results    | utf8                                                     |
| character_set_client     | utf8                                                     |
| character_set_server     | utf8mb4                                                  |
| character_set_database   | utf8mb4                                                  |
| character_set_filesystem | binary                                                   |
+--------------------------+----------------------------------------------------------+
8 rows in set (0.00 sec)
```

```
SET CHARACTER SET utf8mb4;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
SHOW VARIABLES LIKE 'character_set%';
```

```
+--------------------------+----------------------------------------------------------+
| Variable_name            | Value                                                    |
+--------------------------+----------------------------------------------------------+
| character_set_connection | utf8mb4                                                  |
| character_set_system     | utf8                                                     |
| character_set_results    | utf8mb4                                                  |
| character_set_client     | utf8mb4                                                  |
| character_sets_dir       | /usr/local/mysql-5.6.25-osx10.8-x86_64/share/charsets/ |
| character_set_database   | utf8mb4                                                  |
| character_set_filesystem | binary                                                   |
| character_set_server     | utf8mb4                                                  |
+--------------------------+----------------------------------------------------------+
8 rows in set (0.00 sec)
```

14.11.2.87.3 MySQL 兼容性

`SET [NAMES|CHARACTER SET]` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.87.4 另请参阅

- SHOW [GLOBAL|SESSION] VARIABLES
- SET <variable>
- Character Set Support

14.11.2.88 SET PASSWORD

`SET PASSWORD` 语句用于更改 TiDB 系统数据库中的用户密码。

14.11.2.88.1 语法图

SetStmt:

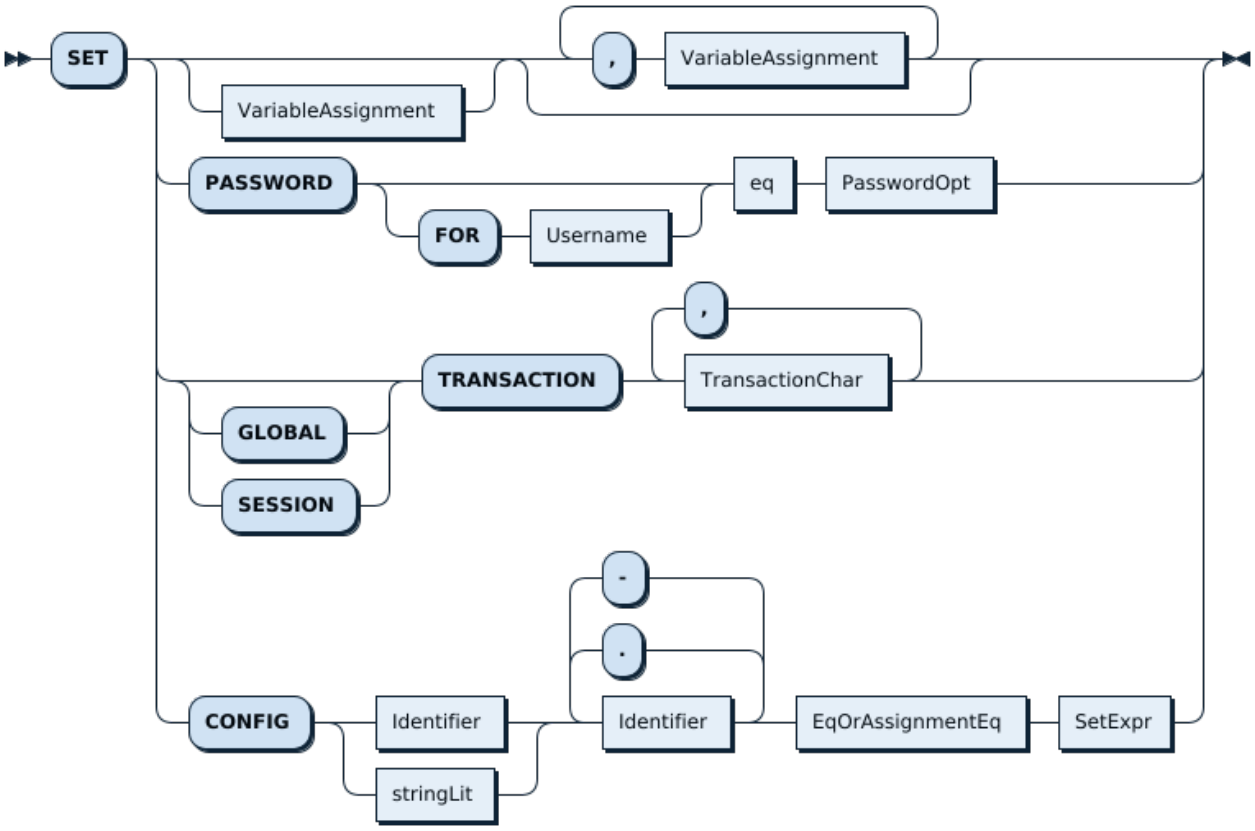


图 380: SetStmt

14.11.2.88.2 示例

```
SET PASSWORD='test';
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
CREATE USER 'newuser' IDENTIFIED BY 'test';
```

```
Query OK, 1 row affected (0.00 sec)
```

```
SHOW CREATE USER 'newuser';
```

```
+-----------------------------------------------------------------------------------------------------------------
    ↪
| CREATE USER for newuser@%
    ↪
    ↪ |
+-----------------------------------------------------------------------------------------------------------------
    ↪
```

```
| CREATE USER 'newuser'@'%' IDENTIFIED WITH 'mysql_native_password' AS '*94
    ↪ BDCEBE19083CE2A1F959FD02F964C7AF4CFC29' REQUIRE NONE PASSWORD EXPIRE DEFAULT ACCOUNT
    ↪ UNLOCK |
+-----------------------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

```
SET PASSWORD FOR 'newuser' = 'test';
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
SHOW CREATE USER 'newuser';
```

```
+-----------------------------------------------------------------------------------------------------------------------
    ↪
| CREATE USER for newuser@%
    ↪
    ↪ |
+-----------------------------------------------------------------------------------------------------------------------
    ↪
| CREATE USER 'newuser'@'%' IDENTIFIED WITH 'mysql_native_password' AS '*94
    ↪ BDCEBE19083CE2A1F959FD02F964C7AF4CFC29' REQUIRE NONE PASSWORD EXPIRE DEFAULT ACCOUNT
    ↪ UNLOCK |
+-----------------------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

```
SET PASSWORD FOR 'newuser' = PASSWORD('test');
```

上述语法是早期 MySQL 版本的过时语法。

```
Query OK, 0 rows affected (0.00 sec)
```

```
SHOW CREATE USER 'newuser';
```

```
+-----------------------------------------------------------------------------------------------------------------------
    ↪
| CREATE USER for newuser@%
    ↪
    ↪ |
+-----------------------------------------------------------------------------------------------------------------------
    ↪
| CREATE USER 'newuser'@'%' IDENTIFIED WITH 'mysql_native_password' AS '*94
    ↪ BDCEBE19083CE2A1F959FD02F964C7AF4CFC29' REQUIRE NONE PASSWORD EXPIRE DEFAULT ACCOUNT
    ↪ UNLOCK |
```

```
+-----------------------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

14.11.2.88.3 MySQL 兼容性

SET PASSWORD 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.88.4 另请参阅

- CREATE USER
- Privilege Management

14.11.2.89 SET RESOURCE GROUP

SET RESOURCE GROUP 用于设置当前会话使用的资源组。

14.11.2.89.1 语法图

SetResourceGroupStmt:

```
SetResourceGroupStmt ::=
   "SET" "RESOURCE" "GROUP" ResourceGroupName

ResourceGroupName ::=
   Identifier
```

14.11.2.89.2 示例

创建一个用户 user1，创建两个资源组 rg1 和 rg2，并将用户 user1 绑定资源组 rg1。

```
CREATE USER 'user1';
CREATE RESOURCE GROUP 'rg1' RU_PER_SEC = 1000;
ALTER USER 'user1' RESOURCE GROUP `rg1`;
```

使用 user1 登录，查看当前用户绑定的资源组。

```
SELECT CURRENT_RESOURCE_GROUP();
```

```
+--------------------------+
| CURRENT_RESOURCE_GROUP() |
+--------------------------+
| rg1                      |
+--------------------------+
1 row in set (0.00 sec)
```

执行 SET RESOURCE GROUP 将当前会话的资源组设置为 rg2。

```
SET RESOURCE GROUP `rg2`;
SELECT CURRENT_RESOURCE_GROUP();
```

```
+--------------------------+
| CURRENT_RESOURCE_GROUP() |
+--------------------------+
| rg2                      |
+--------------------------+
1 row in set (0.00 sec)
```

执行 SET RESOURCE GROUP 设置当前会话使用默认资源组。

```
SET RESOURCE GROUP ``;
SELECT CURRENT_RESOURCE_GROUP();
```

```
+--------------------------+
| CURRENT_RESOURCE_GROUP() |
+--------------------------+
| default                  |
+--------------------------+
1 row in set (0.00 sec)
```

14.11.2.89.3 MySQL 兼容性

MySQL 也支持 SET RESOURCE GROUP，但是接受的参数和 TiDB 不同，两者并不兼容。

14.11.2.89.4 另请参阅

- CREATE RESOURCE GROUP
- DROP RESOURCE GROUP
- ALTER RESOURCE GROUP
- 使用资源管控 (Resource Control) 实现资源隔离

14.11.2.90 SET ROLE

SET ROLE 用于在当前用户会话中启用角色。使用 SET ROLE 启用角色后，用户可以使用这些角色的权限。

14.11.2.90.1 语法图

SetRoleStmt:



图 381: SetRoleStmt

SetRoleOpt:

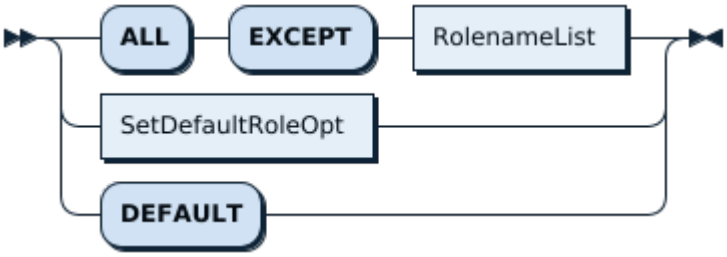


图 382: SetRoleOpt

SetDefaultRoleOpt:

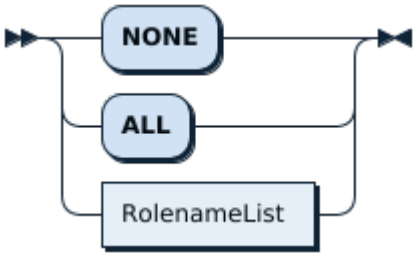


图 383: SetDefaultRoleOpt

14.11.2.90.2 示例

创建一个用户 `'u1'@'%'`，创建三个角色 `'r1'@'%'`、`'r2'@'%'` 和 `'r3'@'%'` 并将这些角色授予给 `'u1'@'%'`。将 `'u1'@'%'` 的默认启用角色设置为 `'r1'@'%'`。

```
CREATE USER 'u1'@'%';
CREATE ROLE 'r1', 'r2', 'r3';
GRANT 'r1', 'r2', 'r3' TO 'u1'@'%';
SET DEFAULT ROLE 'r1' TO 'u1'@'%';
```

使用 `'u1'@'%'` 登录，执行 `SET ROLE` 将启用角色设置为 `ALL`。

```
SET ROLE ALL;
SELECT CURRENT_ROLE();
```

```
+----------------------------+
| CURRENT_ROLE()             |
+----------------------------+
| `r1`@`%`,`r2`@`%`,`r3`@`%` |
+----------------------------+
1 row in set (0.000 sec)
```

执行 `SET ROLE` 将启用角色设置为 `'r2'` 和 `'r3'`。

```
SET ROLE 'r2', 'r3';
SELECT CURRENT_ROLE();
```

```
+-------------------+
| CURRENT_ROLE()    |
+-------------------+
| `r2`@`%`,`r3`@`%` |
+-------------------+
1 row in set (0.000 sec)
```

执行 SET ROLE 将启用角色设置为 DEFAULT。

```
SET ROLE DEFAULT;
SELECT CURRENT_ROLE();
```

```
+----------------+
| CURRENT_ROLE() |
+----------------+
| `r1`@`%`       |
+----------------+
1 row in set (0.000 sec)
```

执行 SET ROLE 将启用角色设置为 NONE。

```
SET ROLE NONE;
SELECT CURRENT_ROLE();
```

```
+----------------+
| CURRENT_ROLE() |
+----------------+
|                |
+----------------+
1 row in set (0.000 sec)
```

14.11.2.90.3 MySQL 兼容性

SET ROLE 语句与 MySQL 8.0 的角色功能完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.90.4 另请参阅

- CREATE ROLE
- DROP ROLE
- GRANT <role>
- REVOKE <role>
- SET DEFAULT ROLE
- 基于角色的访问控制

14.11.2.91 SET TRANSACTION

SET TRANSACTION 语句用于在 GLOBAL 或 SESSION 的基础上更改当前的隔离级别，是 SET transaction_isolation ↪ ='new-value' 的替代语句，提供 MySQL 和 SQL 标准的兼容性。

14.11.2.91.1 语法图

```
SetStmt ::=
    'SET' ( VariableAssignmentList |
    'PASSWORD' ('FOR' Username)? '=' PasswordOpt |
    ( 'GLOBAL'| 'SESSION' )? 'TRANSACTION' TransactionChars |
    'CONFIG' ( Identifier | stringLit) ConfigItemName EqOrAssignmentEq SetExpr )

TransactionChars ::=
    ( 'ISOLATION' 'LEVEL' IsolationLevel | 'READ' 'WRITE' | 'READ' 'ONLY' AsOfClause? )

IsolationLevel ::=
    ( 'REPEATABLE' 'READ' | 'READ' ( 'COMMITTED' | 'UNCOMMITTED' ) | 'SERIALIZABLE' )

AsOfClause ::=
    ( 'AS' 'OF' 'TIMESTAMP' Expression)
```

14.11.2.91.2 示例

```
SHOW SESSION VARIABLES LIKE 'transaction_isolation';
```

```
+-----------------------+-----------------+
| Variable_name         | Value           |
+-----------------------+-----------------+
| transaction_isolation | REPEATABLE-READ |
+-----------------------+-----------------+
1 row in set (0.00 sec)
```

```
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
SHOW SESSION VARIABLES LIKE 'transaction_isolation';
```

```
+-----------------------+----------------+
| Variable_name         | Value          |
+-----------------------+----------------+
| transaction_isolation | READ-COMMITTED |
+-----------------------+----------------+
1 row in set (0.01 sec)
```

```
SET SESSION transaction_isolation = 'REPEATABLE-READ';
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
SHOW SESSION VARIABLES LIKE 'transaction_isolation';
```

```
+-----------------------+-----------------+
| Variable_name         | Value           |
+-----------------------+-----------------+
| transaction_isolation | REPEATABLE-READ |
+-----------------------+-----------------+
1 row in set (0.00 sec)
```

14.11.2.91.3 MySQL 兼容性

- TiDB 支持仅在语法中将事务设置为只读的功能。
- 不支持隔离级别 READ-UNCOMMITTED 和 SERIALIZABLE。
- 通过快照隔离 (Snapshot Isolation) 技术，实现乐观事务的 REPEATABLE-READ 隔离级别，和 MySQL 兼容。
- 在悲观事务中，TiDB 支持与 MySQL 兼容的 REPEATABLE-READ 和 READ-COMMITTED 两种隔离级别。具体描述详见Isolation Levels。

14.11.2.91.4 另请参阅

- SET [GLOBAL|SESSION] <variable>
- Isolation Levels

14.11.2.92 SET [GLOBAL|SESSION] <variable>

SET [GLOBAL|SESSION] 语句用于在 SESSION 或 GLOBAL 的范围内，对某个 TiDB 的内置变量进行更改。

> **注意：**
>
> 与 MySQL 类似，对 GLOBAL 变量的更改不适用于已有连接或本地连接，只有新会话才会反映值的变化。

14.11.2.92.1 语法图

SetStmt:

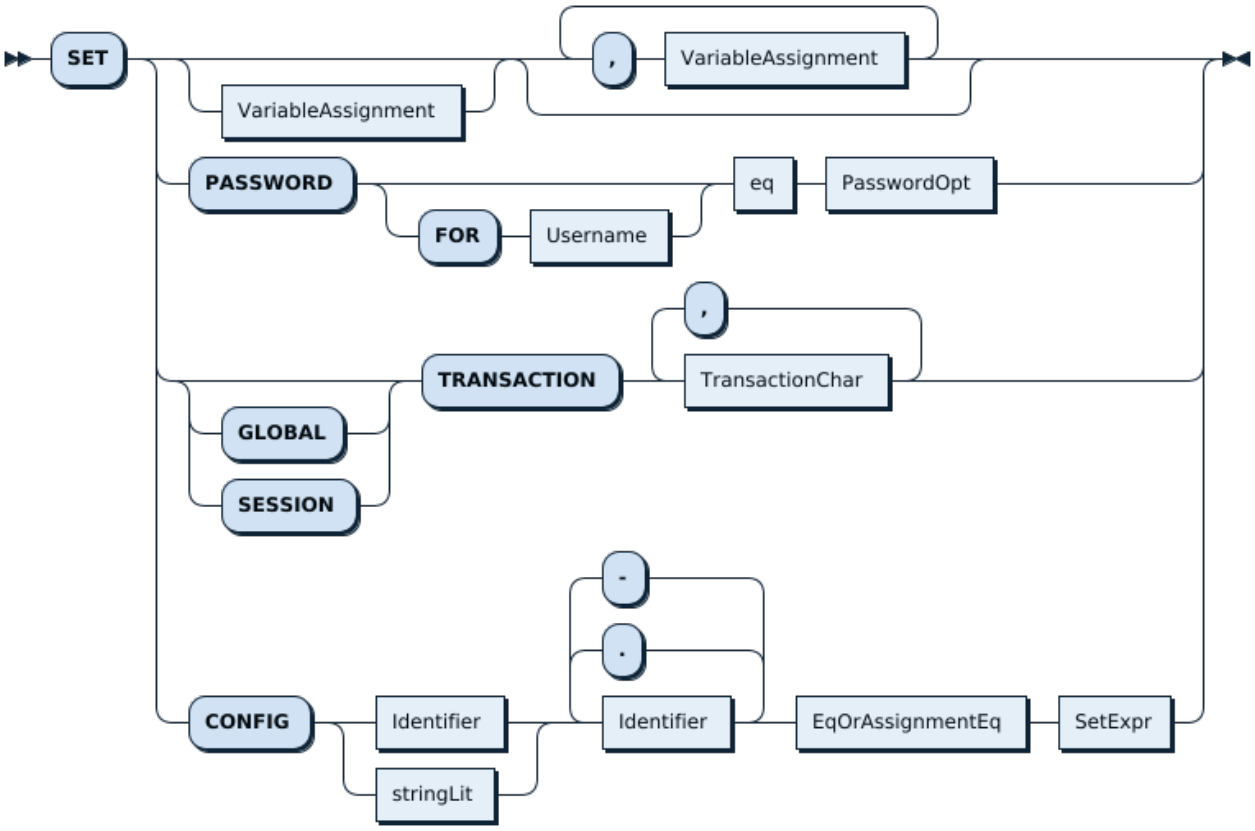


图 384: SetStmt

VariableAssignment:

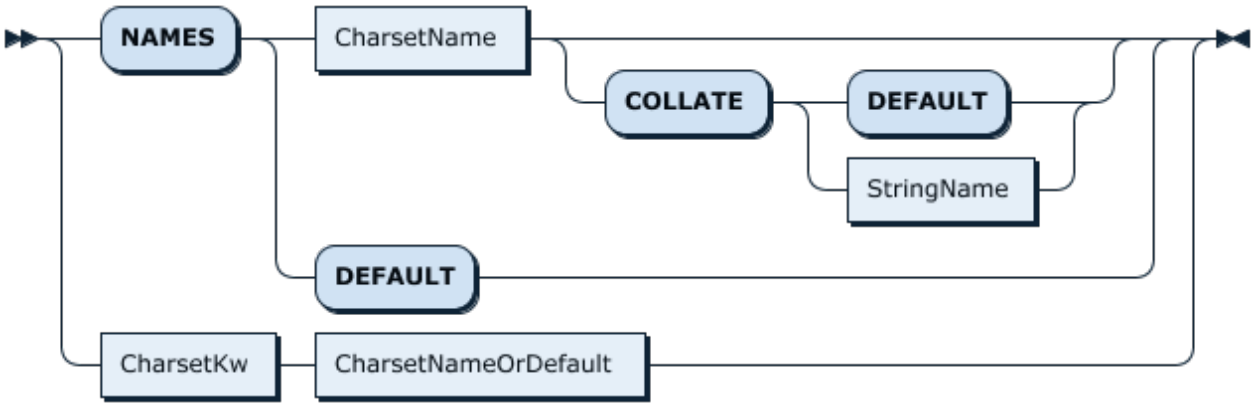


图 385: VariableAssignment

14.11.2.92.2 示例

获取 `sql_mode` 的值：

```
SHOW GLOBAL VARIABLES LIKE 'sql_mode';
```

```
+---------------+-----------------------------------------------------------------------------------------------------------
    ↪
| Variable_name | Value
    ↪
    ↪ |
+---------------+-----------------------------------------------------------------------------------------------------------
    ↪
| sql_mode      | ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,
    ↪ ERROR_FOR_DIVISION_BY_ZERO,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION |
+---------------+-----------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

```
SHOW SESSION VARIABLES LIKE 'sql_mode';
```

```
+---------------+-----------------------------------------------------------------------------------------------------------
    ↪
| Variable_name | Value
    ↪
    ↪ |
+---------------+-----------------------------------------------------------------------------------------------------------
    ↪
| sql_mode      | ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,
    ↪ ERROR_FOR_DIVISION_BY_ZERO,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION |
+---------------+-----------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

更新全局的 sql_mode：

```
SET GLOBAL sql_mode = 'STRICT_TRANS_TABLES,NO_AUTO_CREATE_USER';
```

```
Query OK, 0 rows affected (0.03 sec)
```

检查更新之后的 sql_mode 的取值，可以看到 SESSION 级别的值没有更新：

```
SHOW GLOBAL VARIABLES LIKE 'sql_mode';
```

```
+---------------+-----------------------------------------+
| Variable_name | Value                                   |
+---------------+-----------------------------------------+
| sql_mode      | STRICT_TRANS_TABLES,NO_AUTO_CREATE_USER |
+---------------+-----------------------------------------+
1 row in set (0.00 sec)
```

```
SHOW SESSION VARIABLES LIKE 'sql_mode';
```

```
+---------------+-------------------------------------------------------------------------------------------------
    ↪
| Variable_name | Value
    ↪
    ↪ |
+---------------+-------------------------------------------------------------------------------------------------
    ↪
| sql_mode      | ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,
    ↪ ERROR_FOR_DIVISION_BY_ZERO,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION |
+---------------+-------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

SET SESSION 则可以立即生效：

```
SET SESSION sql_mode = 'STRICT_TRANS_TABLES,NO_AUTO_CREATE_USER';
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
SHOW SESSION VARIABLES LIKE 'sql_mode';
```

```
+---------------+-----------------------------------------+
| Variable_name | Value                                   |
+---------------+-----------------------------------------+
| sql_mode      | STRICT_TRANS_TABLES,NO_AUTO_CREATE_USER |
+---------------+-----------------------------------------+
1 row in set (0.00 sec)
```

14.11.2.92.3 MySQL 兼容性

使用 SET [GLOBAL|SESSION] <variable> 更改系统变量上，TiDB 与 MySQL 存在以下差异

- 与 MySQL 不同，TiDB 中使用 SET GLOBAL 所作的修改会应用于集群中的全部 TiDB 实例。而在 MySQL 中，修改不会应用于副本。
- TiDB 中的若干变量可读又可设置，这是与 MySQL 相兼容的要求，因为应用程序和连接器常读取 MySQL 变量。例如：JDBC 连接器同时读取和设置缓存查询的参数，尽管并不依赖这一行为。
- 即使在 TiDB 服务器重启后，SET GLOBAL 的更改也仍然有效。这样，TiDB 中的 SET GLOBAL 更类似于 MySQL 8.0 及更高版本中的 SET PERSIST。

14.11.2.92.4 另请参阅

- SHOW [GLOBAL|SESSION] VARIABLES

14.11.2.93 SHOW [BACKUPS|RESTORES]

SHOW [BACKUPS|RESTORES] 语句会列出所有队列中或正在执行的BACKUP 和RESTORE 任务。

查询 BACKUP 任务时，使用 SHOW BACKUPS 语句。查询 RESTORE 任务时，使用 SHOW RESTORES 语句。执行两个语句均需要 SUPER 权限。

14.11.2.93.1 语法图

```
ShowBRIEStmt ::=
    "SHOW" ("BACKUPS" | "RESTORES") ShowLikeOrWhere?

ShowLikeOrWhere ::=
    "LIKE" SimpleExpr
|   "WHERE" Expression
```

14.11.2.93.2 示例

在一个连接中，执行以下命令备份数据库：

```
BACKUP DATABASE `test` TO 's3://example-bucket/backup-01/';
```

在备份完成之前，在新的连接中执行 SHOW BACKUPS：

```
SHOW BACKUPS;
```

```
+--
    ↪ ------------------------------+---------+----------+--------------------+--------------------+---------
    ↪
| Destination                    | State    | Progress | Queue_Time          | Execution_Time
    ↪       | Finish_Time | Connection |
+--
    ↪ ------------------------------+---------+----------+--------------------+--------------------+---------
    ↪
| s3://example-bucket/backup-01/ | Backup   | 98.38    | 2020-04-12 23:09:03 | 2020-04-12 23:09:25
    ↪  |        NULL |           4 |
+--
    ↪ ------------------------------+---------+----------+--------------------+--------------------+---------
    ↪
1 row in set (0.00 sec)
```

输出结果的第一行描述如下：

列名	描述
Destination	目标存储的 URL（为避免泄露密钥，所有参数均不显示）
State	任务状态
Progress	当前状态的进度（百分比）

列名	描述
Queue Time	任务开始排队的时间
Execution Time	任务开始执行的时间；对于队列中任务，该值为 0000-00-00 00:00:00
Finish_Time	(暂不适用)
Connection	运行任务的连接 ID

连接 ID 可用于在KILL TIDB QUERY 语句中取消备份/恢复任务：

```
KILL TIDB QUERY 4;
```

```
Query OK, 0 rows affected (0.00 sec)
```

过滤

在 LIKE 子句中使用通配符，可以按目标存储 URL 筛选任务：

```
SHOW BACKUPS LIKE 's3://%';
```

使用 WHERE 子句，可以按列筛选任务：

```
SHOW BACKUPS WHERE `Progress` < 25.0;
```

14.11.2.93.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.93.4 另请参阅

- BACKUP
- RESTORE

14.11.2.94 SHOW ANALYZE STATUS

SHOW ANALYZE STATUS 语句提供 TiDB 正在执行的统计信息收集任务以及有限条历史任务记录。

从 TiDB v6.1.0 起，执行 SHOW ANALYZE STATUS 语句将显示集群级别的任务，且 TiDB 重启后仍能看到重启之前的任务记录。在 TiDB v6.1.0 之前，执行 SHOW ANALYZE STATUS 语句仅显示实例级别的任务，且 TiDB 重启后任务记录会被清空。

从 TiDB v6.1.0 起，你可以通过系统表 mysql.analyze_jobs 查看过去 7 天内的历史记录。

14.11.2.94.1 语法图

```
ShowAnalyzeStatusStmt ::= 'SHOW' 'ANALYZE' 'STATUS' ShowLikeOrWhereOpt

ShowLikeOrWhereOpt ::= 'LIKE' SimpleExpr | 'WHERE' Expression
```

14.11.2.94.2 示例

```
mysql> create table t(x int, index idx(x)) partition by hash(x) partitions 2;
Query OK, 0 rows affected (0.69 sec)

mysql> set @@tidb_analyze_version = 1;
Query OK, 0 rows affected (0.00 sec)

mysql> analyze table t;
Query OK, 0 rows affected (0.20 sec)

mysql> show analyze status;
+--
    ↪ ------------+------------+----------------+-------------------+---------------+---------------------+---
    ↪
| Table_schema | Table_name | Partition_name | Job_info          | Processed_rows | Start_time
    ↪          | End_time            | State    | Fail_reason | Instance       | Process_ID |
+--
    ↪ ------------+------------+----------------+-------------------+---------------+---------------------+---
    ↪
| test         | t          | p1             | analyze index idx |              0 | 2022-05-27
    ↪ 11:29:46 | 2022-05-27 11:29:46 | finished | NULL        | 127.0.0.1:4000 |       NULL |
| test         | t          | p0             | analyze index idx |              0 | 2022-05-27
    ↪ 11:29:46 | 2022-05-27 11:29:46 | finished | NULL        | 127.0.0.1:4000 |       NULL |
| test         | t          | p1             | analyze columns   |              0 | 2022-05-27
    ↪ 11:29:46 | 2022-05-27 11:29:46 | finished | NULL        | 127.0.0.1:4000 |       NULL |
| test         | t          | p0             | analyze columns   |              0 | 2022-05-27
    ↪ 11:29:46 | 2022-05-27 11:29:46 | finished | NULL        | 127.0.0.1:4000 |       NULL |
+--
    ↪ ------------+------------+----------------+-------------------+---------------+---------------------+---
    ↪
4 rows in set (0.01 sec)

mysql> set @@tidb_analyze_version = 2;
Query OK, 0 rows affected (0.00 sec)

mysql> analyze table t;
Query OK, 0 rows affected, 2 warnings (0.03 sec)

mysql> show analyze status;
+--
    ↪ ------------+------------+----------------+-------------------------------------------------------------
    ↪
| Table_schema | Table_name | Partition_name | Job_info
    ↪                                                          | Processed_rows | Start_time
    ↪          | End_time            | State    | Fail_reason | Instance       | Process_ID |
```

```
+--
    ↪ ------------+------------+---------------+-------------------------------------------------------------
    ↪
| test         | t          | p1             | analyze table all columns with 256 buckets, 500
    ↪ topn, 1 samplerate |              0 | 2022-05-27 11:30:12 | 2022-05-27 11:30:12 |
    ↪ finished | NULL        | 127.0.0.1:4000 |       NULL |
| test         | t          | p0             | analyze table all columns with 256 buckets, 500
    ↪ topn, 1 samplerate |              0 | 2022-05-27 11:30:12 | 2022-05-27 11:30:12 |
    ↪ finished | NULL        | 127.0.0.1:4000 |       NULL |
| test         | t          | p1             | analyze index idx
    ↪                                                 |              0 | 2022-05-27 11:29:46 |
    ↪  2022-05-27 11:29:46 | finished | NULL        | 127.0.0.1:4000 |       NULL |
| test         | t          | p0             | analyze index idx
    ↪                                                 |              0 | 2022-05-27 11:29:46 |
    ↪  2022-05-27 11:29:46 | finished | NULL        | 127.0.0.1:4000 |       NULL |
| test         | t          | p1             | analyze columns
    ↪                                                    |              0 | 2022-05-27 11:29:46
    ↪  | 2022-05-27 11:29:46 | finished | NULL        | 127.0.0.1:4000 |       NULL |
| test         | t          | p0             | analyze columns
    ↪                                                    |              0 | 2022-05-27 11:29:46
    ↪  | 2022-05-27 11:29:46 | finished | NULL        | 127.0.0.1:4000 |       NULL |
+--
    ↪ ------------+------------+---------------+-------------------------------------------------------------
    ↪
6 rows in set (0.00 sec)
```

14.11.2.94.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.94.4 另请参阅

- ANALYZE_STATUS 表

14.11.2.95 SHOW [GLOBAL|SESSION] BINDINGS

SHOW BINDINGS 语句用于显示创建过的 SQL 绑定的相关信息。BINDING 语句可以在 GLOBAL 或者 SESSION 作用域内显示执行计划绑定。在不指定作用域时，默认的作用域为 SESSION。

14.11.2.95.1 语法图

ShowStmt:

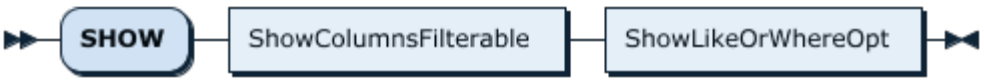


图 386: ShowStmt

ShowTargetFilterable:

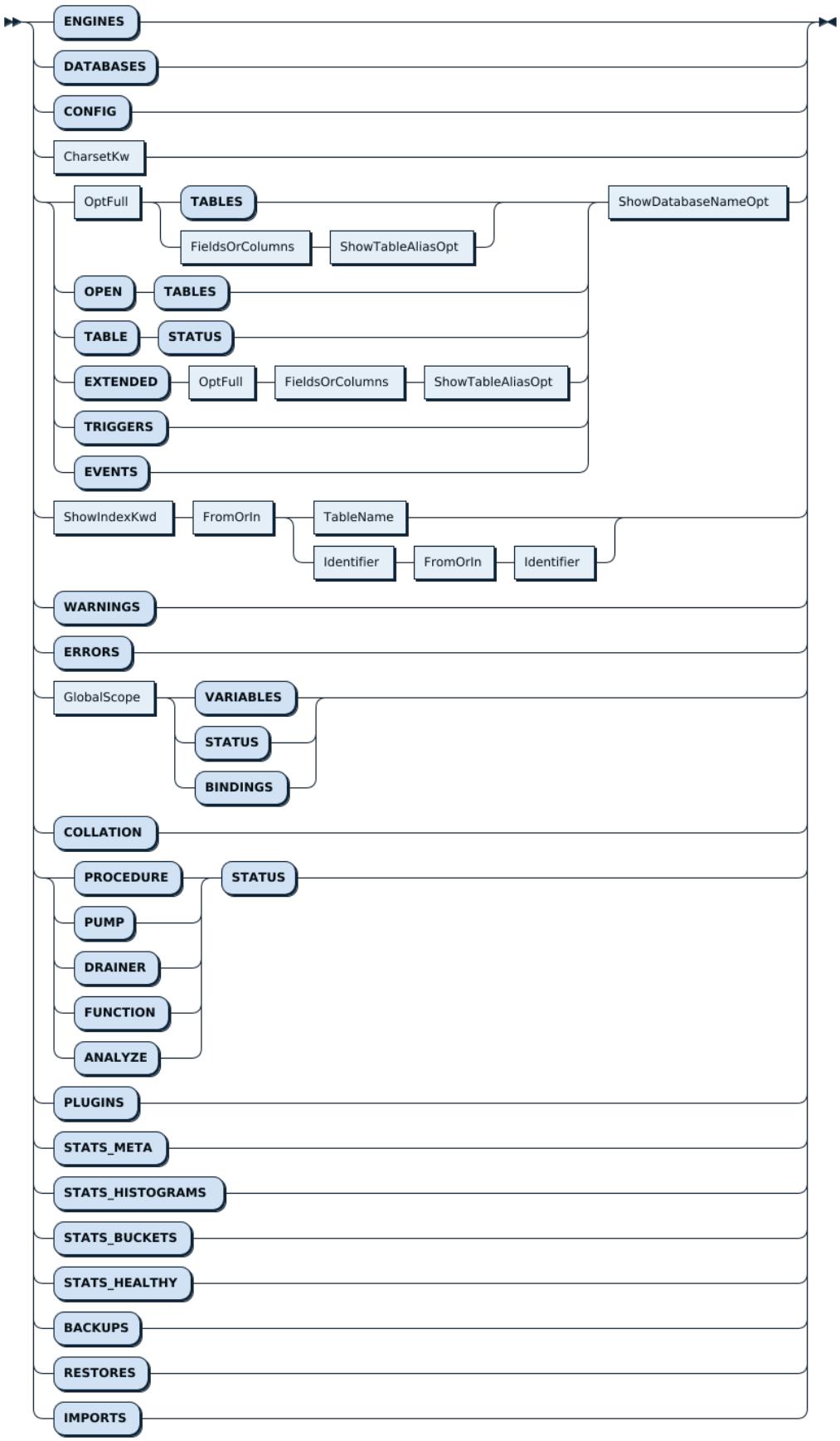


图 387: ShowTargetFilterable

GlobalScope:



图 388: GlobalScope

ShowLikeOrWhereOpt

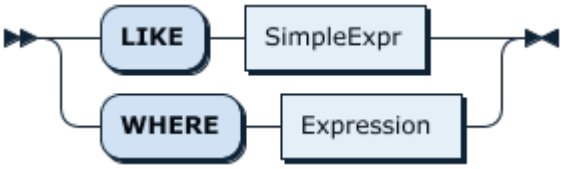


图 389: ShowLikeOrWhereOpt

14.11.2.95.2 语法说明

```
SHOW [GLOBAL | SESSION] BINDINGS [ShowLikeOrWhereOpt];
```

该语句会输出 GLOBAL 或者 SESSION 作用域内的执行计划绑定，在不指定作用域时默认作用域为 SESSION。目前 SHOW BINDINGS 会输出 8 列，具体如下：

列名	说明
original_sql	参数化后的原始 SQL
bind_sql	带 Hint 的绑定 SQL
default_db	默认数据库名
status	状态，包括 using（正在使用）、deleted（已删除）、invalid（无效）、rejected（演进时被拒绝）和 pending verify（等待演进验证）
create_time	创建时间
update_time	更新时间
charset	字符集
collation	排序规则

列名	说明
source	创建方式，包括 manual（由 create [global] binding 生成）、capture（由 tidb 自动创建生成）和 evolve（由 tidb 自动演进生成）

14.11.2.95.3 示例

```
CREATE TABLE t1 (
    id INT NOT NULL PRIMARY KEY auto_increment,
    b INT NOT NULL,
    pad VARBINARY(255),
    INDEX(b)
   );
Query OK, 0 rows affected (0.07 sec)

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM dual;
Query OK, 1 row affected (0.01 sec)
Records: 1  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 8 rows affected (0.00 sec)
Records: 8  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 1000 rows affected (0.04 sec)
Records: 1000  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 100000 rows affected (1.74 sec)
Records: 100000  Duplicates: 0  Warnings: 0
```

```
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 100000 rows affected (2.15 sec)
Records: 100000  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 100000 rows affected (2.64 sec)
Records: 100000  Duplicates: 0  Warnings: 0

SELECT SLEEP(1);
+----------+
| SLEEP(1) |
+----------+
|        0 |
+----------+
1 row in set (1.00 sec)

ANALYZE TABLE t1;
Query OK, 0 rows affected (1.33 sec)

EXPLAIN ANALYZE SELECT * FROM t1 WHERE b = 123;
+--
    ↪ ----------------------------+---------+---------+-----------+---------------------+-----------------
    ↪
| id                          | estRows | actRows | task      | access object       |
    ↪ execution info                                                                | operator info
    ↪                       | memory         | disk |
+--
    ↪ ----------------------------+---------+---------+-----------+---------------------+-----------------
    ↪
| IndexLookUp_10                | 583.00  | 297     | root      |                     | time
    ↪ :10.545072ms, loops:2, rpc num: 1, rpc time:398.359µs, proc keys:297  |
    ↪                                     | 109.1484375 KB | N/A  |
| ├─IndexRangeScan_8(Build)     | 583.00  | 297     | cop[tikv] | table:t1, index:b(b) | time:0s
    ↪ , loops:4                                                             | range:[123,123],
    ↪ keep order:false | N/A            | N/A  |
| └─TableRowIDScan_9(Probe)     | 583.00  | 297     | cop[tikv] | table:t1             | time:12
    ↪ ms, loops:4                                                           | keep order:false
    ↪                   | N/A            | N/A  |
+--
    ↪ ----------------------------+---------+---------+-----------+---------------------+-----------------
    ↪
3 rows in set (0.02 sec)
```

```
CREATE SESSION BINDING FOR
    SELECT * FROM t1 WHERE b = 123
   USING
    SELECT * FROM t1 IGNORE INDEX (b) WHERE b = 123;
Query OK, 0 rows affected (0.00 sec)

EXPLAIN ANALYZE  SELECT * FROM t1 WHERE b = 123;
+--
    ↪ ----------------------+----------+---------+----------+---------------+-------------------------------
    ↪
| id                    | estRows   | actRows | task      | access object | execution info
    ↪                                                                  | operator info      |
    ↪ memory        | disk |
+--
    ↪ ----------------------+----------+---------+----------+---------------+-------------------------------
    ↪
| TableReader_7           | 583.00    | 297     | root      |               | time:222.32506ms,
    ↪ loops:2, rpc num: 1, rpc time:222.078952ms, proc keys:301010 | data:Selection_6   |
    ↪ 88.6640625 KB | N/A  |
| └─Selection_6           | 583.00    | 297     | cop[tikv] |               | time:224ms, loops
    ↪ :298                                                          | eq(test.t1.b, 123) | N/A
    ↪           | N/A  |
|   └─TableFullScan_5     | 301010.00 | 301010  | cop[tikv] | table:t1      | time:220ms, loops
    ↪ :298                                                          | keep order:false   | N/A
    ↪           | N/A  |
+--
    ↪ ----------------------+----------+---------+----------+---------------+-------------------------------
    ↪
3 rows in set (0.22 sec)

SHOW SESSION BINDINGS\G
*************************** 1. row ***************************
Original_sql: select * from t1 where b = ?
    Bind_sql: SELECT * FROM t1 IGNORE INDEX (b) WHERE b = 123
  Default_db: test
      Status: using
 Create_time: 2020-05-22 14:38:03.456
 Update_time: 2020-05-22 14:38:03.456
     Charset: utf8mb4
   Collation: utf8mb4_0900_ai_ci
1 row in set (0.00 sec)
```

14.11.2.95.4 MySQL 兼容性

`SHOW [GLOBAL|SESSION] BINDINGS` 语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.95.5 另请参阅

- CREATE [GLOBAL|SESSION] BINDING
- DROP [GLOBAL|SESSION] BINDING
- ANALYZE
- Optimizer Hints
- 执行计划管理 (SPM)

14.11.2.96 SHOW BUILTINS

SHOW BUILTINS 语句用于列出 TiDB 中所有的内置函数。

14.11.2.96.1 语法图

ShowBuiltinsStmt:

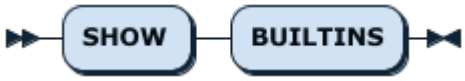


图 390: ShowBuiltinsStmt

14.11.2.96.2 示例

```
SHOW BUILTINS;
```

```
+----------------------------+
| Supported_builtin_functions |
+----------------------------+
| abs                        |
| acos                       |
| adddate                    |
| addtime                    |
| aes_decrypt                |
| aes_encrypt                |
| and                        |
| any_value                  |
| ascii                      |
| asin                       |
| atan                       |
| atan2                      |
| benchmark                  |
| bin                        |
| bit_count                  |
| bit_length                 |
| bitand                     |
| bitneg                     |
```

```
| bitor                      |
| bitxor                     |
| case                       |
| ceil                       |
| ceiling                    |
| char_func                  |
| char_length                |
| character_length           |
| charset                    |
| coalesce                   |
| coercibility               |
| collation                  |
| compress                   |
| concat                     |
| concat_ws                  |
| connection_id              |
| conv                       |
| convert                    |
| convert_tz                 |
| cos                        |
| cot                        |
| crc32                      |
| curdate                    |
| current_date               |
| current_role               |
| current_time               |
| current_timestamp          |
| current_user               |
| curtime                    |
| database                   |
| date                       |
| date_add                   |
| date_format                |
| date_sub                   |
| datediff                   |
| day                        |
| dayname                    |
| dayofmonth                 |
| dayofweek                  |
| dayofyear                  |
| decode                     |
| default_func               |
| degrees                    |
| des_decrypt                |
| des_encrypt                |
```

```
| div                          |
| elt                          |
| encode                       |
| encrypt                      |
| eq                           |
| exp                          |
| export_set                   |
| extract                      |
| field                        |
| find_in_set                  |
| floor                        |
| format                       |
| format_bytes                 |
| format_nano_time             |
| found_rows                   |
| from_base64                  |
| from_days                    |
| from_unixtime                |
| ge                           |
| get_format                   |
| get_lock                     |
| getparam                     |
| getvar                       |
| greatest                     |
| gt                           |
| hex                          |
| hour                         |
| if                           |
| ifnull                       |
| in                           |
| inet6_aton                   |
| inet6_ntoa                   |
| inet_aton                    |
| inet_ntoa                    |
| insert_func                  |
| instr                        |
| intdiv                       |
| interval                     |
| is_free_lock                 |
| is_ipv4                      |
| is_ipv4_compat               |
| is_ipv4_mapped               |
| is_ipv6                      |
| is_used_lock                 |
| isfalse                      |
```

```
| isnull                    |
| istrue                    |
| json_array                |
| json_array_append         |
| json_array_insert         |
| json_contains             |
| json_contains_path        |
| json_depth                |
| json_extract              |
| json_insert               |
| json_keys                 |
| json_length               |
| json_merge                |
| json_merge_patch          |
| json_merge_preserve       |
| json_object               |
| json_pretty               |
| json_quote                |
| json_remove               |
| json_replace              |
| json_search               |
| json_set                  |
| json_storage_size         |
| json_type                 |
| json_unquote              |
| json_valid                |
| last_day                  |
| last_insert_id            |
| lastval                   |
| lcase                     |
| le                        |
| least                     |
| left                      |
| leftshift                 |
| length                    |
| like                      |
| ln                        |
| load_file                 |
| localtime                 |
| localtimestamp            |
| locate                    |
| log                       |
| log10                     |
| log2                      |
| lower                     |
```

```
| lpad                      |
| lt                        |
| ltrim                     |
| make_set                  |
| makedate                  |
| maketime                  |
| master_pos_wait           |
| md5                       |
| microsecond               |
| mid                       |
| minus                     |
| minute                    |
| mod                       |
| month                     |
| monthname                 |
| mul                       |
| name_const                |
| ne                        |
| nextval                   |
| not                       |
| now                       |
| nulleq                    |
| oct                       |
| octet_length              |
| old_password              |
| or                        |
| ord                       |
| password_func             |
| period_add                |
| period_diff               |
| pi                        |
| plus                      |
| position                  |
| pow                       |
| power                     |
| quarter                   |
| quote                     |
| radians                   |
| rand                      |
| random_bytes              |
| regexp                    |
| release_all_locks         |
| release_lock              |
| repeat                    |
| replace                   |
```

```
| reverse                    |
| right                      |
| rightshift                 |
| round                      |
| row_count                  |
| rpad                       |
| rtrim                      |
| schema                     |
| sec_to_time                |
| second                     |
| session_user               |
| setval                     |
| setvar                     |
| sha                        |
| sha1                       |
| sha2                       |
| sign                       |
| sin                        |
| sleep                      |
| sm3                        |
| space                      |
| sqrt                       |
| str_to_date                |
| strcmp                     |
| subdate                    |
| substr                     |
| substring                  |
| substring_index            |
| subtime                    |
| sysdate                    |
| system_user                |
| tan                        |
| tidb_decode_key            |
| tidb_decode_plan           |
| tidb_is_ddl_owner          |
| tidb_parse_tso             |
| tidb_version               |
| time                       |
| time_format                |
| time_to_sec                |
| timediff                   |
| timestamp                  |
| timestampadd               |
| timestampdiff              |
| to_base64                  |
```

```
| to_days                       |
| to_seconds                    |
| trim                          |
| truncate                      |
| ucase                         |
| unaryminus                    |
| uncompress                    |
| uncompressed_length           |
| unhex                         |
| unix_timestamp                |
| upper                         |
| user                          |
| utc_date                      |
| utc_time                      |
| utc_timestamp                 |
| uuid                          |
| uuid_short                    |
| validate_password_strength    |
| version                       |
| week                          |
| weekday                       |
| weekofyear                    |
| weight_string                 |
| xor                           |
| year                          |
| yearweek                      |
+-------------------------------+
268 rows in set (0.00 sec)
```

14.11.2.96.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.97 SHOW CHARACTER SET

`SHOW CHARACTER SET` 语句提供 TiDB 中可用字符集的静态列表。此列表不反映当前连接或用户的任何属性。

14.11.2.97.1 语法图

ShowCharsetStmt:



图 391: ShowCharsetStmt

CharsetKw:

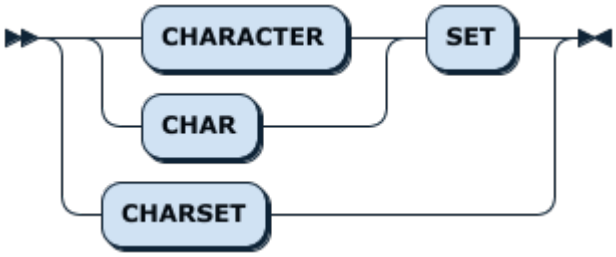


图 392: CharsetKw

14.11.2.97.2 示例

```
SHOW CHARACTER SET;
```

```
+---------+---------------+-------------------+--------+
| Charset | Description   | Default collation | Maxlen |
+---------+---------------+-------------------+--------+
| utf8    | UTF-8 Unicode | utf8_bin          |      3 |
| utf8mb4 | UTF-8 Unicode | utf8mb4_bin       |      4 |
| ascii   | US ASCII      | ascii_bin         |      1 |
| latin1  | Latin1        | latin1_bin        |      1 |
| binary  | binary        | binary            |      1 |
+---------+---------------+-------------------+--------+
5 rows in set (0.00 sec)
```

14.11.2.97.3 MySQL 兼容性

SHOW CHARACTER SET 语句功能与 MySQL 完全兼容。注意，TiDB 中字符集的默认排序规则与 MySQL 有所不同，具体可以参考与 MySQL 兼容性对比。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.97.4 另请参阅

- SHOW COLLATION
- 字符集和排序规则

14.11.2.98 SHOW COLLATION

SHOW COLLATION 语句用于提供一个静态的排序规则列表，确保与 MySQL 客户端库的兼容性。

> **注意：**
>
> SHOW COLLATION 所展示的排序规则列表与 TiDB 集群是否开启新排序规则框架有关，详情请见TiDB 字符集和排序规则。

14.11.2.98.1 语法图

ShowCollationStmt:

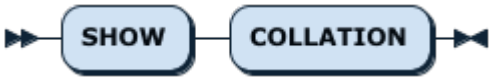


图 393: ShowCollationStmt

14.11.2.98.2 示例

若未开启新排序规则框架，仅展示二进制排序规则：

```
SHOW COLLATION;
```

```
+-------------+---------+------+---------+----------+---------+
| Collation   | Charset | Id   | Default | Compiled | Sortlen |
+-------------+---------+------+---------+----------+---------+
| utf8mb4_bin | utf8mb4 |   46 | Yes     | Yes      |       1 |
| latin1_bin  | latin1  |   47 | Yes     | Yes      |       1 |
| binary      | binary  |   63 | Yes     | Yes      |       1 |
| ascii_bin   | ascii   |   65 | Yes     | Yes      |       1 |
| utf8_bin    | utf8    |   83 | Yes     | Yes      |       1 |
+-------------+---------+------+---------+----------+---------+
5 rows in set (0.02 sec)
```

若开启了新排序规则框架，则在二进制排序规则之外，额外支持 `utf8_general_ci` 和 `utf8mb4_general_ci` 两种大小写和口音不敏感的排序规则：

```
SHOW COLLATION;
```

```
+--------------------+---------+------+---------+----------+---------+
| Collation          | Charset | Id   | Default | Compiled | Sortlen |
+--------------------+---------+------+---------+----------+---------+
| ascii_bin          | ascii   |   65 | Yes     | Yes      |       1 |
| binary             | binary  |   63 | Yes     | Yes      |       1 |
| gbk_bin            | gbk     |   87 |         | Yes      |       1 |
| gbk_chinese_ci     | gbk     |   28 | Yes     | Yes      |       1 |
| latin1_bin         | latin1  |   47 | Yes     | Yes      |       1 |
| utf8_bin           | utf8    |   83 | Yes     | Yes      |       1 |
| utf8_general_ci    | utf8    |   33 |         | Yes      |       1 |
| utf8_unicode_ci    | utf8    |  192 |         | Yes      |       1 |
| utf8mb4_bin        | utf8mb4 |   46 | Yes     | Yes      |       1 |
| utf8mb4_general_ci | utf8mb4 |   45 |         | Yes      |       1 |
| utf8mb4_unicode_ci | utf8mb4 |  224 |         | Yes      |       1 |
```

```
+--------------------+---------+------+---------+----------+---------+
11 rows in set (0.001 sec)
```

14.11.2.98.3 MySQL 兼容性

SHOW COLLATION 语句功能与 MySQL 完全兼容。注意，TiDB 中字符集的默认排序规则与 MySQL 有所不同，具体可参考与 MySQL 兼容性对比。如发现任何其他兼容性差异，请在 GitHub 上提交 issue。

14.11.2.98.4 另请参阅

- SHOW CHARACTER SET
- 字符集和排序规则

14.11.2.99 SHOW [FULL] COLUMNS FROM

SHOW [FULL] COLUMNS FROM <table_name> 语句用于以表格格式描述表或视图中的列。可选关键字 FULL 用于显示当前用户对该列的权限，以及表定义中的 comment。

SHOW [FULL] FIELDS FROM <table_name>、DESC <table_name>、DESCRIBE <table_name> 和 EXPLAIN <table_name> 语句都是 SHOW [FULL] COLUMNS FROM 的别名。

> **注意：**
>
> DESC TABLE <table_name>、DESCRIBE TABLE <table_name> 和 EXPLAIN TABLE <table_name> 与上面的语句并不等价，它们是DESC SELECT * FROM <table_name> 的别名。

14.11.2.99.1 语法图

ShowStmt:

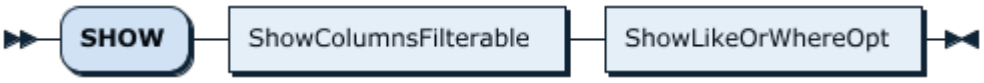


图 394: ShowStmt

ShowColumnsFilterable:



图 395: ShowColumnsFilterable

OptFull:



图 396: OptFull

FieldsOrColumns:



图 397: FieldsOrColumns

ShowTableAliasOpt:

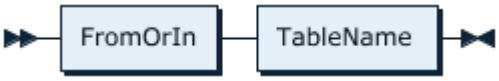


图 398: ShowTableAliasOpt

FromOrIn:

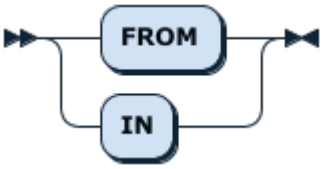


图 399: FromOrIn

TableName:

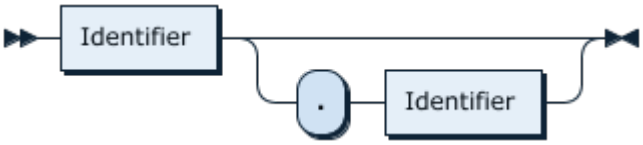


图 400: TableName

ShowDatabaseNameOpt:

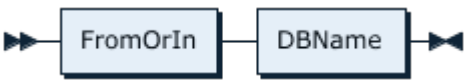


图 401: ShowDatabaseNameOpt

DBName:



图 402: DBName

ShowLikeOrWhereOpt:

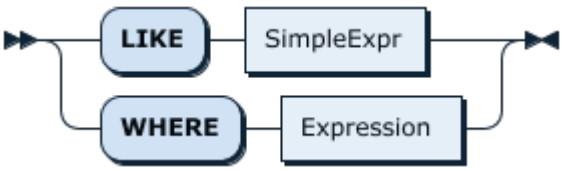


图 403: ShowLikeOrWhereOpt

14.11.2.99.2 示例

```
create view v1 as select 1;
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
show columns from v1;
```

```
+-------+-----------+------+------+---------+-------+
| Field | Type      | Null | Key  | Default | Extra |
+-------+-----------+------+------+---------+-------+
| 1     | bigint(1) | YES  |      | NULL    |       |
+-------+-----------+------+------+---------+-------+
1 row in set (0.00 sec)
```

```
desc v1;
```

```
+-------+-----------+------+------+---------+-------+
| Field | Type      | Null | Key  | Default | Extra |
+-------+-----------+------+------+---------+-------+
| 1     | bigint(1) | YES  |      | NULL    |       |
+-------+-----------+------+------+---------+-------+
1 row in set (0.00 sec)
```

```
describe v1;
```

```
+-------+-----------+------+------+---------+-------+
| Field | Type      | Null | Key  | Default | Extra |
+-------+-----------+------+------+---------+-------+
```

```
| 1     | bigint(1) | YES  |      | NULL    |       |
+-------+-----------+------+------+---------+-------+
1 row in set (0.00 sec)
```

```
explain v1;
```

```
+-------+-----------+------+------+---------+-------+
| Field | Type      | Null | Key  | Default | Extra |
+-------+-----------+------+------+---------+-------+
| 1     | bigint(1) | YES  |      | NULL    |       |
+-------+-----------+------+------+---------+-------+
1 row in set (0.00 sec)
```

```
show fields from v1;
```

```
+-------+-----------+------+------+---------+-------+
| Field | Type      | Null | Key  | Default | Extra |
+-------+-----------+------+------+---------+-------+
| 1     | bigint(1) | YES  |      | NULL    |       |
+-------+-----------+------+------+---------+-------+
1 row in set (0.00 sec)
```

```
show full columns from v1;
```

```
+-------+-----------+-----------+------+------+---------+-------+---------------------------------+---------+
    ↪
| Field | Type      | Collation | Null | Key  | Default | Extra | Privileges
    ↪                     | Comment |
+-------+-----------+-----------+------+------+---------+-------+---------------------------------+---------+
    ↪
| 1     | bigint(1) | NULL      | YES  |      | NULL    |       | select,insert,update,references
    ↪  |         |
+-------+-----------+-----------+------+------+---------+-------+---------------------------------+---------+
    ↪
1 row in set (0.00 sec)
```

```
show full columns from mysql.user;
```

```
+-----------------------+---------------+-------------+------+------+---------+-------+-----------------------------
    ↪
| Field                 | Type          | Collation   | Null | Key  | Default | Extra |
    ↪ Privileges                      | Comment |
+-----------------------+---------------+-------------+------+------+---------+-------+-----------------------------
    ↪
```

```
| Host                  | char(255)     | utf8mb4_bin | NO   | PRI  | NULL    |       | select,
    ↪ insert,update,references |         |
| User                  | char(32)      | utf8mb4_bin | NO   | PRI  | NULL    |       | select,
    ↪ insert,update,references |         |
| authentication_string | text          | utf8mb4_bin | YES  |      | NULL    |       | select,
    ↪ insert,update,references |         |
| plugin                | char(64)      | utf8mb4_bin | YES  |      | NULL    |       | select,
    ↪ insert,update,references |         |
| Select_priv           | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Insert_priv           | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Update_priv           | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Delete_priv           | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Create_priv           | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Drop_priv             | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Process_priv          | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Grant_priv            | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| References_priv       | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Alter_priv            | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Show_db_priv          | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Super_priv            | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Create_tmp_table_priv | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Lock_tables_priv      | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Execute_priv          | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Create_view_priv      | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Show_view_priv        | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Create_routine_priv   | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
```

```
| Alter_routine_priv     | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Index_priv             | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Create_user_priv       | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Event_priv             | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Repl_slave_priv        | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Repl_client_priv       | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Trigger_priv           | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Create_role_priv       | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Drop_role_priv         | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Account_locked         | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Shutdown_priv          | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Reload_priv            | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| FILE_priv              | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Config_priv            | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Create_Tablespace_Priv | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| User_attributes        | json          | NULL        | YES  |      | NULL    |       | select,
    ↪ insert,update,references |         |
+------------------------+---------------+-------------+------+------+---------+-------+-------------------------
    ↪
38 rows in set (0.00 sec)
```

14.11.2.99.3 MySQL 兼容性

`SHOW [FULL] COLUMNS FROM` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.99.4 另请参阅

- SHOW CREATE TABLE

14.11.2.100 SHOW CONFIG

`SHOW CONFIG` 语句用于展示 TiDB 各个组件当前正在应用的配置，请注意，配置与系统变量作用于不同维度，请不要混淆，如果希望获取系统变量信息，请使用SHOW VARIABLES 语法。

14.11.2.100.1 语法图

ShowStmt:

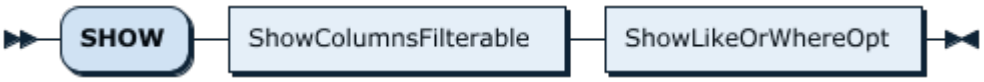


图 404: ShowStmt

ShowTargetFilterable:

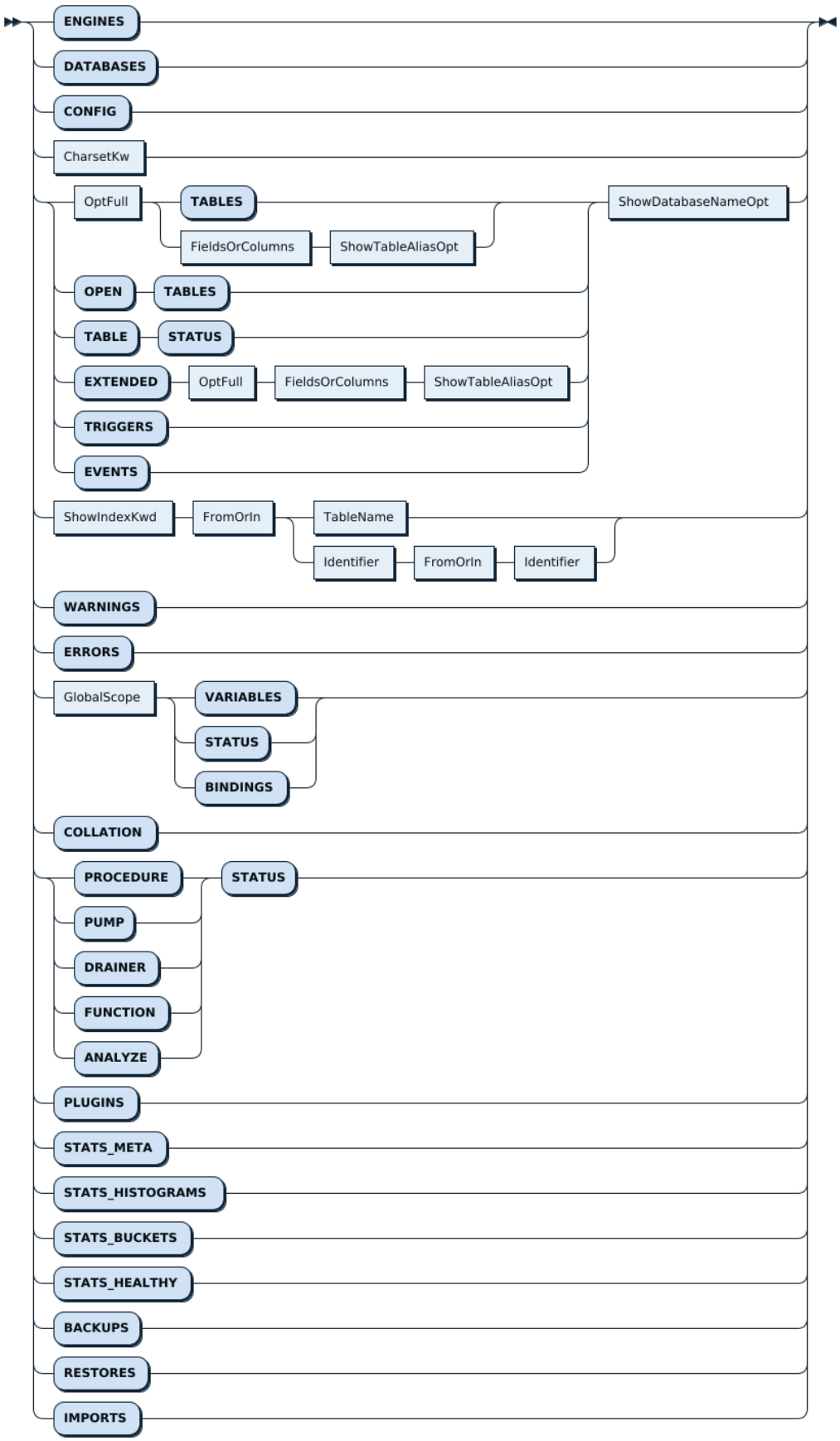


图 405: ShowTargetFilterable

14.11.2.100.2 示例

显示所有配置：

```
SHOW CONFIG;
```

```
+------+----------------+--------------------------------------------------+-----------------------------------
    ↪
| Type | Instance       | Name                                             | Value
    ↪                                                                  |
+------+----------------+--------------------------------------------------+-----------------------------------
    ↪
| tidb | 127.0.0.1:4000 | advertise-address                                | 127.0.0.1
    ↪                                                                |
| tidb | 127.0.0.1:4000 | binlog.binlog-socket                             |
    ↪                                                                  |
| tidb | 127.0.0.1:4000 | binlog.enable                                    | false
    ↪                                                                |
...
120 rows in set (0.01 sec)
```

显示 type 是 tidb 的配置：

```
SHOW CONFIG WHERE type = 'tidb' AND name = 'advertise-address';
```

```
+------+----------------+-------------------+-----------+
| Type | Instance       | Name              | Value     |
+------+----------------+-------------------+-----------+
| tidb | 127.0.0.1:4000 | advertise-address | 127.0.0.1 |
+------+----------------+-------------------+-----------+
1 row in set (0.05 sec)
```

也可以用 LIKE 子句来显示 type 是 tidb 的配置：

```
SHOW CONFIG LIKE 'tidb';
```

```
+------+----------------+--------------------------------------------------+-----------------------------------
    ↪
| Type | Instance       | Name                                             | Value
    ↪                                                                  |
+------+----------------+--------------------------------------------------+-----------------------------------
    ↪
| tidb | 127.0.0.1:4000 | advertise-address                                | 127.0.0.1
    ↪                                                                |
| tidb | 127.0.0.1:4000 | binlog.binlog-socket                             |
    ↪                                                                  |
| tidb | 127.0.0.1:4000 | binlog.enable                                    | false
    ↪                                                                |
```

```
...
40 rows in set (0.01 sec)
```

14.11.2.100.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.100.4 另请参阅

- SHOW VARIABLES

14.11.2.101 SHOW CREATE PLACEMENT POLICY

`SHOW CREATE PLACEMENT POLICY` 语句可用于查看放置策略当前的定义，并在另一个 TiDB 集群中重新创建该策略。

14.11.2.101.1 语法图

```
ShowCreatePlacementPolicyStmt ::=
    "SHOW" "CREATE" "PLACEMENT" "POLICY" PolicyName

PolicyName ::=
    Identifier
```

14.11.2.101.2 示例

```
CREATE PLACEMENT POLICY p1 PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1" FOLLOWERS=4;
CREATE TABLE t1 (a INT) PLACEMENT POLICY=p1;
SHOW CREATE PLACEMENT POLICY p1\G;
```

```
Query OK, 0 rows affected (0.08 sec)

Query OK, 0 rows affected (0.10 sec)

***************************[ 1. row ]***************************
Policy        | p1
Create Policy | CREATE PLACEMENT POLICY `p1` PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-
    ↪ west-1" FOLLOWERS=4
1 row in set (0.00 sec)
```

14.11.2.101.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.101.4 另请参阅

- Placement Rules in SQL
- SHOW PLACEMENT
- CREATE PLACEMENT POLICY
- ALTER PLACEMENT POLICY
- DROP PLACEMENT POLICY

14.11.2.102 SHOW CREATE RESOURCE GROUP

`SHOW CREATE RESOURCE GROUP` 语句可用于查看资源组当前的定义。

14.11.2.102.1 语法图

```
ShowCreateResourceGroupStmt ::=
    "SHOW" "CREATE" "RESOURCE" "GROUP" ResourceGroupName

ResourceGroupName ::=
    Identifier
```

14.11.2.102.2 示例

查看资源组 `rg1` 当前的定义：

```
CREATE RESOURCE GROUP rg1 RU_PER_SEC=100;
Query OK, 0 rows affected (0.10 sec)
```

```
SHOW CREATE RESOURCE GROUP rg1;
***************************[ 1. row ]***************************
+----------------+-----------------------------------------------------------------+
| Resource_Group | Create Resource Group                                           |
+----------------+-----------------------------------------------------------------+
| rg1            | CREATE RESOURCE GROUP `rg1` RU_PER_SEC=100 PRIORITY=MEDIUM |
+----------------+-----------------------------------------------------------------+
1 row in set (0.01 sec)
```

14.11.2.102.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.102.4 另请参阅

- TiDB RESOURCE CONTROL
- CREATE RESOURCE GROUP
- ALTER RESOURCE GROUP
- DROP RESOURCE GROUP

14.11.2.103 SHOW CREATE SEQUENCE

`SHOW CREATE SEQUENCE` 语句用于查看一个序列的详细信息，类似于 `SHOW CREATE TABLE` 语句。

14.11.2.103.1 语法图

ShowCreateSequenceStmt:



图 406: ShowCreateSequenceStmt

TableName:

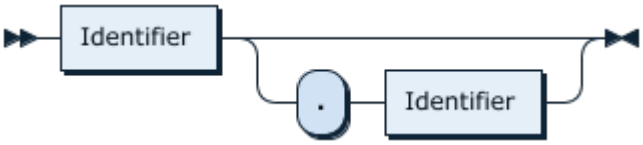


图 407: TableName

14.11.2.103.2 示例

```
CREATE SEQUENCE seq;
```

```
Query OK, 0 rows affected (0.03 sec)
```

```
SHOW CREATE SEQUENCE seq;
```

```
+-------+-------------------------------------------------------------------------------------------------------------
    ↪
| Table | Create Table
    ↪
    ↪ |
+-------+-------------------------------------------------------------------------------------------------------------
    ↪
| seq   | CREATE SEQUENCE `seq` start with 1 minvalue 1 maxvalue 9223372036854775806 increment by
    ↪  1 cache 1000 nocycle ENGINE=InnoDB |
+-------+-------------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

14.11.2.103.3 MySQL 兼容性

该语句是 TiDB 的扩展，序列的实现借鉴自 MariaDB。

14.11.2.103.4 另请参阅

- CREATE SEQUENCE
- DROP SEQUENCE

14.11.2.104 SHOW CREATE TABLE

`SHOW CREATE TABLE` 语句用于显示用 SQL 重新创建已有表的确切语句。

14.11.2.104.1 语法图

ShowCreateTableStmt:

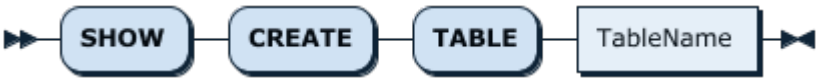


图 408: ShowCreateTableStmt

TableName:

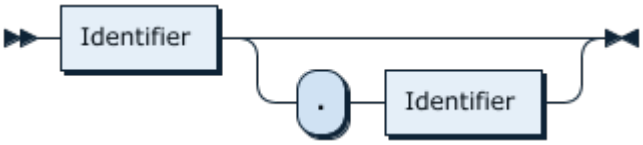


图 409: TableName

14.11.2.104.2 示例

```
CREATE TABLE t1 (a INT);
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
SHOW CREATE TABLE t1;
```

```
+-------+-----------------------------------------------------------------------------------------------------------------
    ↪
| Table | Create Table
    ↪
    ↪ |
+-------+-----------------------------------------------------------------------------------------------------------------
    ↪
| t1    | CREATE TABLE `t1` (
  `a` int(11) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin |
+-------+-----------------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

14.11.2.104.3 MySQL 兼容性

`SHOW CREATE TABLE` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.104.4 另请参阅

- CREATE TABLE
- DROP TABLE
- SHOW TABLES
- SHOW COLUMNS FROM

14.11.2.105 SHOW CREATE DATABASE

`SHOW CREATE DATABASE` 语句用于显示用 SQL 重新创建已有库的确切语句。`SHOW CREATE SCHEMA` 与其同义。

14.11.2.105.1 语法图

ShowCreateDatabaseStmt:

```
ShowCreateDatabaseStmt ::=
    "SHOW" "CREATE" "DATABASE" | "SCHEMA" ("IF" "NOT" "EXISTS")? DBName
```

14.11.2.105.2 示例

```
CREATE DATABASE test;
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
SHOW CREATE DATABASE test;
```

```
+----------+-----------------------------------------------------------------+
| Database | Create Database                                                 |
+----------+-----------------------------------------------------------------+
| test     | CREATE DATABASE `test` /*!40100 DEFAULT CHARACTER SET utf8mb4 */ |
+----------+-----------------------------------------------------------------+
1 row in set (0.00 sec)
```

```
SHOW CREATE SCHEMA IF NOT EXISTS test;
```

```
+----------+-----------------------------------------------------------------------------------------------+
    ↪
| Database | Create Database
    ↪                                                                                |
+----------+-----------------------------------------------------------------------------------------------+
    ↪
```

```
| test     | CREATE DATABASE /*!32312 IF NOT EXISTS*/ `test` /*!40100 DEFAULT CHARACTER SET
    ↪ utf8mb4 */ |
+----------+-----------------------------------------------------------------------------------------+
    ↪
1 row in set (0.00 sec)
```

14.11.2.105.3 MySQL 兼容性

`SHOW CREATE DATABASE` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.105.4 另请参阅

- CREATE TABLE
- DROP TABLE
- SHOW TABLES
- SHOW COLUMNS FROM

14.11.2.106 SHOW CREATE USER

`SHOW CREATE USER` 语句用于显示如何使用 `CREATE USER` 语法来重新创建用户。

14.11.2.106.1 语法图

ShowCreateUserStmt:

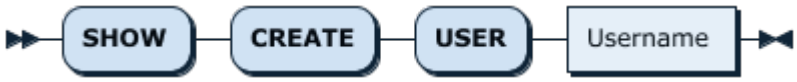


图 410: ShowCreateUserStmt

Username:

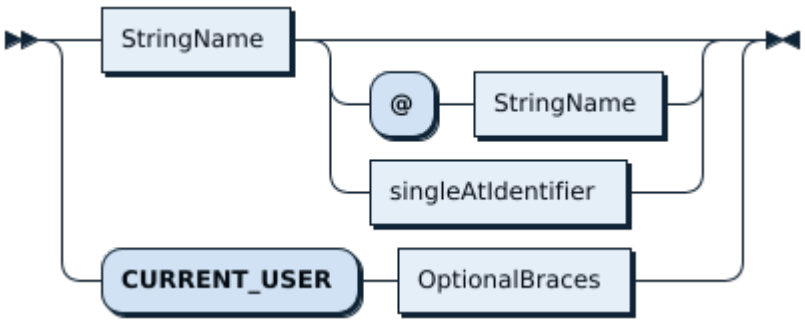


图 411: Username

14.11.2.106.2 示例

```
SHOW CREATE USER 'root';
```

```
+-----------------------------------------------------------------------------------------------------------------------------------
    ↪
| CREATE USER for root@%
    ↪
    ↪ |
+-----------------------------------------------------------------------------------------------------------------------------------
    ↪
| CREATE USER 'root'@'%' IDENTIFIED WITH 'mysql_native_password' AS '' REQUIRE NONE PASSWORD
    ↪ EXPIRE DEFAULT ACCOUNT UNLOCK |
+-----------------------------------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)

mysql> SHOW GRANTS FOR 'root';
+-----------------------------------------+
| Grants for root@%                       |
+-----------------------------------------+
| GRANT ALL PRIVILEGES ON *.* TO 'root'@'%' |
+-----------------------------------------+
1 row in set (0.00 sec)
```

14.11.2.106.3 MySQL 兼容性

- SHOW CREATE USER 的输出结果旨在匹配 MySQL，但 TiDB 尚不支持若干 CREATE 选项。尚未支持的选项在语句执行过程中会被解析但会被跳过执行。详情可参阅 [security compatibility]。

14.11.2.106.4 另请参阅

- CREATE USER
- SHOW GRANTS
- DROP USER

14.11.2.107 SHOW DATABASES

SHOW DATABASES 语句用于显示当前用户有权访问的数据库列表。当前用户无权访问的数据库将从列表中隐藏。information_schema 数据库始终出现在列表的最前面。

SHOW SCHEMAS 是 SHOW DATABASES 语句的别名。

14.11.2.107.1 语法图

ShowDatabasesStmt:



图 412: ShowDatabasesStmt

ShowLikeOrWhereOpt:

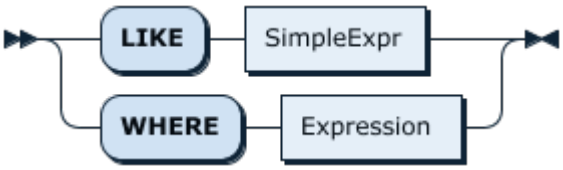


图 413: ShowLikeOrWhereOpt

14.11.2.107.2 示例

```
SHOW DATABASES;
```

```
+--------------------+
| Database           |
+--------------------+
| INFORMATION_SCHEMA |
| PERFORMANCE_SCHEMA |
| mysql              |
| test               |
+--------------------+
4 rows in set (0.00 sec)
```

```
CREATE DATABASE mynewdb;
```

```
Query OK, 0 rows affected (0.10 sec)
```

```
SHOW DATABASES;
```

```
+--------------------+
| Database           |
+--------------------+
| INFORMATION_SCHEMA |
| PERFORMANCE_SCHEMA |
| mynewdb            |
| mysql              |
| test               |
```

```
+--------------------+
5 rows in set (0.00 sec)
```

14.11.2.107.3 MySQL 兼容性

`SHOW DATABASES` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.107.4 另请参阅

- SHOW SCHEMAS
- DROP DATABASE
- CREATE DATABASE

14.11.2.108 SHOW DRAINER STATUS

`SHOW DRAINER STATUS` 语句用于显示集群中所有 Drainer 的状态信息。

14.11.2.108.1 示例

```
SHOW DRAINER STATUS;
```

```
+----------|----------------|--------|--------------------|---------------------|
|  NodeID  |     Address    | State  |   Max_Commit_Ts    |    Update_Time      |
+----------|----------------|--------|--------------------|---------------------|
| drainer1 | 127.0.0.3:8249 | Online | 408553768673342532 | 2019-05-01 00:00:03 |
+----------|----------------|--------|--------------------|---------------------|
| drainer2 | 127.0.0.4:8249 | Online | 408553768673345531 | 2019-05-01 00:00:04 |
+----------|----------------|--------|--------------------|---------------------|
2 rows in set (0.00 sec)
```

14.11.2.108.2 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.108.3 另请参阅

- SHOW PUMP STATUS
- CHANGE PUMP STATUS
- CHANGE DRAINER STATUS

14.11.2.109 SHOW ENGINES

`SHOW ENGINES` 语句用于列出所有支持的存储引擎，该语法仅提供 MySQL 兼容性。

14.11.2.109.1 语法图

ShowEnginesStmt:

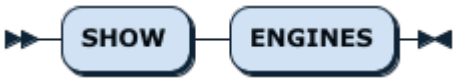


图 414: ShowEnginesStmt

```
SHOW ENGINES;
```

14.11.2.109.2 示例

```
SHOW ENGINES;
```

```
+--------+---------+----------------------------------------------------------------+--------------+------+----------
    ↪
| Engine | Support | Comment                                                        | Transactions |
    ↪ XA   | Savepoints |
+--------+---------+----------------------------------------------------------------+--------------+------+----------
    ↪
| InnoDB | DEFAULT | Supports transactions, row-level locking, and foreign keys | YES          |
    ↪ YES  | YES        |
+--------+---------+----------------------------------------------------------------+--------------+------+----------
    ↪
1 row in set (0.00 sec)
```

14.11.2.109.3 MySQL 兼容性

- SHOW ENGINES 语句始终只返回 InnoDB 作为其支持的引擎。但 TiDB 内部通常使用 TiKV 作为存储引擎。

14.11.2.110 SHOW ERRORS

SHOW ERRORS 语句用于显示已执行语句中的错误。一旦先前的语句成功执行，就会清除错误缓冲区，这时 SHOW ERRORS 会返回一个空集。

当前的 sql_mode 很大程度决定了哪些语句会产生错误与警告。

14.11.2.110.1 语法图

ShowErrorsStmt:

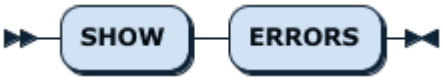


图 415: ShowErrorsStmt

14.11.2.110.2 示例

```
select invalid;
```

```
ERROR 1054 (42S22): Unknown column 'invalid' in 'field list'
```

```
create invalid;
```

```
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to
    ↪ your TiDB version for the right syntax to use line 1 column 14 near "invalid"
```

```
SHOW ERRORS;
```

```
+-------+------+-----------------------------------------------------------------------------------------------
    ↪
| Level | Code | Message
    ↪
    ↪ |
+-------+------+-----------------------------------------------------------------------------------------------
    ↪
| Error | 1054 | Unknown column 'invalid' in 'field list'
    ↪
    ↪ |
| Error | 1064 | You have an error in your SQL syntax; check the manual that corresponds to your
    ↪ TiDB version for the right syntax to use line 1 column 14 near "invalid"  |
+-------+------+-----------------------------------------------------------------------------------------------
    ↪
2 rows in set (0.00 sec)
```

```
CREATE invalid2;
```

```
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to
    ↪ your TiDB version for the right syntax to use line 1 column 15 near "invalid2"
```

```
SELECT 1;
```

```
+------+
| 1    |
+------+
|    1 |
+------+
1 row in set (0.00 sec)
```

```
SHOW ERRORS;
```

```
Empty set (0.00 sec)
```

14.11.2.110.3 MySQL 兼容性

`SHOW ERRORS` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.110.4 另请参阅

- SHOW WARNINGS

14.11.2.111 SHOW [FULL] FIELDS FROM

`SHOW [FULL] FIELDS FROM` 是`SHOW [FULL] COLUMNS FROM` 的别名。包含该语句提供了 MySQL 兼容性。

14.11.2.112 SHOW GRANTS

`SHOW GRANTS` 语句用于显示与用户关联的权限列表。与在 MySQL 中一样，`USAGE` 权限表示登录 TiDB 的能力。

14.11.2.112.1 语法图

ShowGrantsStmt:

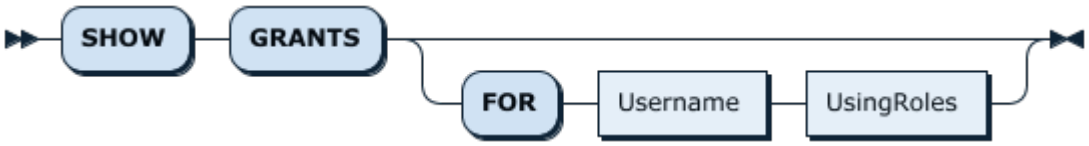


图 416: ShowGrantsStmt

Username:

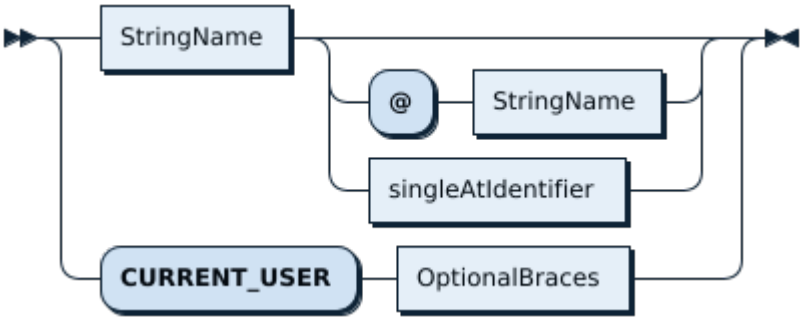


图 417: Username

UsingRoles:



图 418: UsingRoles

RolenameList:

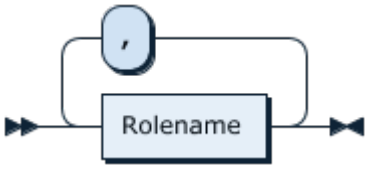


图 419: RolenameList

Rolename:

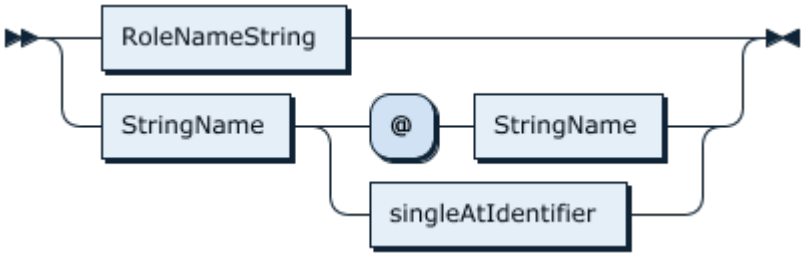


图 420: Rolename

14.11.2.112.2 示例

```
SHOW GRANTS;
```

```
+----------------------------------------+
| Grants for User                        |
+----------------------------------------+
| GRANT ALL PRIVILEGES ON *.* TO 'root'@'%' |
+----------------------------------------+
1 row in set (0.00 sec)
```

```
SHOW GRANTS FOR 'u1';
```

```
ERROR 1141 (42000): There is no such grant defined for user 'u1' on host '%'
```

```
CREATE USER u1;
```

```
Query OK, 1 row affected (0.04 sec)
```

```
GRANT SELECT ON test.* TO u1;
```

```
Query OK, 0 rows affected (0.04 sec)
```

```
SHOW GRANTS FOR u1;
```

```
+-----------------------------------+
| Grants for u1@%                   |
+-----------------------------------+
| GRANT USAGE ON *.* TO 'u1'@'%'    |
| GRANT Select ON test.* TO 'u1'@'%' |
+-----------------------------------+
2 rows in set (0.00 sec)
```

14.11.2.112.3 MySQL 兼容性

`SHOW GRANTS` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.112.4 另请参阅

- SHOW CREATE USER
- GRANT

14.11.2.113 SHOW INDEX [FROM|IN]

`SHOW INDEX [FROM|IN]` 语句是`SHOW INDEXES [FROM|IN]` 的别名。包含该语句提供了 MySQL 兼容性。

14.11.2.114 SHOW INDEXES [FROM|IN]

`SHOW INDEXES [FROM|IN]` 语句用于列出指定表上的索引。`SHOW INDEX [FROM | IN]` 和 `SHOW KEYS [FROM | IN]` 是该语句的别名。包含该语句提供了 MySQL 兼容性。

14.11.2.114.1 语法图

ShowIndexStmt:

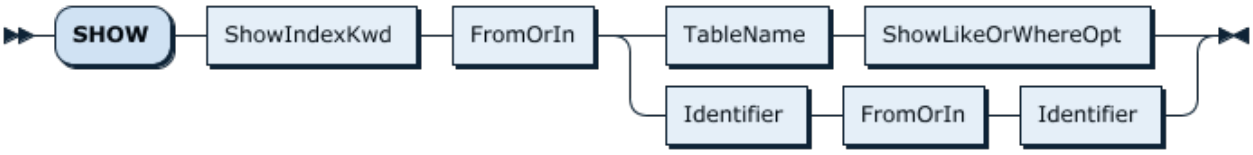


图 421: ShowIndexStmt

ShowIndexKwd:

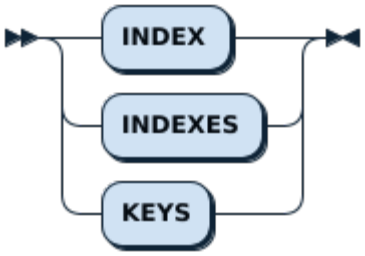


图 422: ShowIndexKwd

FromOrIn:

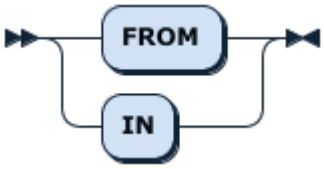


图 423: FromOrIn

TableName:

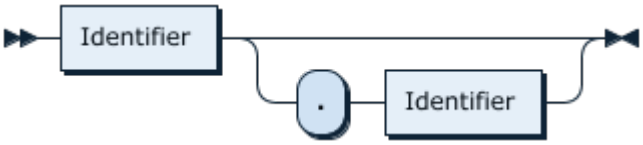


图 424: TableName

ShowLikeOrWhereOpt:

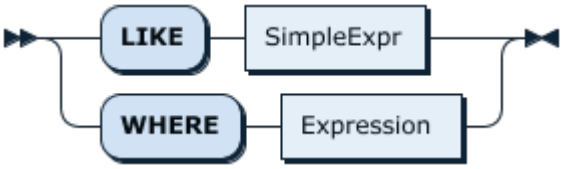


图 425: ShowLikeOrWhereOpt

14.11.2.114.2 示例

```
CREATE TABLE t1 (id int not null primary key AUTO_INCREMENT, col1 INT, INDEX(col1));
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
SHOW INDEXES FROM t1;
```

```
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+-----
    ↪
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part
    ↪  | Packed | Null | Index_type | Comment | Index_comment | Visible | Expression |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+-----
    ↪
| t1    |          0 | PRIMARY  |            1 | id          | A         |           0 |     NULL
    ↪  | NULL   |      | BTREE      |         |               | YES
    | NULL       |
| t1    |          1 | col1     |            1 | col1        | A         |           0 |     NULL
    ↪  | NULL   | YES  | BTREE      |         |               | YES
    | NULL       |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+-----
    ↪
2 rows in set (0.00 sec)
```

```
SHOW INDEX FROM t1;
```

```
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+-----
    ↪
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part
    ↪  | Packed | Null | Index_type | Comment | Index_comment | Visible | Expression |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+-----
    ↪
| t1    |          0 | PRIMARY  |            1 | id          | A         |           0 |     NULL
    ↪  | NULL   |      | BTREE      |         |               | YES
    | NULL       |
| t1    |          1 | col1     |            1 | col1        | A         |           0 |     NULL
    ↪  | NULL   | YES  | BTREE      |         |               | YES
    | NULL       |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+-----
    ↪
2 rows in set (0.00 sec)
```

```
SHOW KEYS FROM t1;
```

```
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+-----
    ↪
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part
    ↪  | Packed | Null | Index_type | Comment | Index_comment | Visible | Expression |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+-----
    ↪
| t1    |          0 | PRIMARY  |            1 | id          | A         |           0 |     NULL
    ↪  | NULL   |      | BTREE      |         |               | YES
```

```
    | NULL       |
| t1    |          1 | col1     |            1 | col1        | A         |           0 |     NULL
    ↪  | NULL    | YES  | BTREE      |          |               | YES
    | NULL       |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+-----
    ↪
2 rows in set (0.00 sec)
```

14.11.2.114.3 MySQL 兼容性

MySQL 中的 `Cardinality` 列返回该索引上不同值的个数，而 TiDB 中的 `Cardinality` 列始终返回 0。

14.11.2.114.4 另请参阅

- SHOW CREATE TABLE
- DROP INDEX
- CREATE INDEX

14.11.2.115 SHOW KEYS [FROM|IN]

`SHOW KEYS [FROM|IN]` 语句是`SHOW INDEXES [FROM|IN]` 语句的别名。包含该语句提供了 MySQL 兼容性。

14.11.2.116 SHOW MASTER STATUS

`SHOW MASTER STATUS` 语句用于显示集群当前最新的 TSO 信息。

14.11.2.116.1 示例

```
SHOW MASTER STATUS;
```

```
+-------------+--------------------+--------------+------------------+-------------------+
| File        | Position           | Binlog_Do_DB | Binlog_Ignore_DB | Executed_Gtid_Set |
+-------------+--------------------+--------------+------------------+-------------------+
| tidb-binlog | 416916363252072450 |              |                  |                   |
+-------------+--------------------+--------------+------------------+-------------------+
1 row in set (0.00 sec)
```

14.11.2.116.2 MySQL 兼容性

`SHOW MASTER STATUS` 语句与 MySQL 兼容，但是执行结果有差异，在 MySQL 中执行结果为 binlog 的位置信息，而在 TiDB 中为最新的 TSO 信息。

14.11.2.116.3 另请参阅

- SHOW PUMP STATUS
- SHOW DRAINER STATUS
- CHANGE PUMP STATUS
- CHANGE DRAINER STATUS

14.11.2.117 SHOW PLACEMENT

SHOW PLACEMENT 汇总了所有放置策略 (placement policy)，并用统一的形式呈现相关信息。

本语句返回结果中的 Scheduling_State 列标识了 Placement Driver (PD) 在当前对象上的调度进度，有以下可能结果：

- PENDING: PD 没有进行调度。可能的原因之一是放置规则虽然语法上正确，但集群拓扑并不满足。比如指定 FOLLOWERS=4 但只有 3 个可用作 follower 的 TiKV 实例。
- INPROGRESS: PD 正在进行调度。
- SCHEDULED: PD 调度完成。

14.11.2.117.1 语法图

```
ShowStmt ::=
    "PLACEMENT"
```

14.11.2.117.2 示例

```
CREATE PLACEMENT POLICY p1 PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1" FOLLOWERS=4;
CREATE TABLE t1 (a INT) PLACEMENT POLICY=p1;
SHOW PLACEMENT;
```

```
Query OK, 0 rows affected (0.01 sec)

Query OK, 0 rows affected (0.00 sec)

+--
    ↪ -------------+-------------------------------------------------------------------------+-------------------
    ↪
| Target        | Placement                                                               |
    ↪ Scheduling_State |
+--
    ↪ -------------+-------------------------------------------------------------------------+-------------------
    ↪
| POLICY p1     | PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1" FOLLOWERS=4 | NULL
    ↪              |
| DATABASE test | PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1" FOLLOWERS=4 |
    ↪ INPROGRESS       |
```

```
| TABLE test.t1 | PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1" FOLLOWERS=4 |
    ↪ INPROGRESS       |
+--
    ↪ -------------+-------------------------------------------------------------------------+-----------------+
    ↪
4 rows in set (0.00 sec)
```

14.11.2.117.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.117.4 另请参阅

- Placement Rules in SQL
- SHOW PLACEMENT FOR
- CREATE PLACEMENT POLICY

14.11.2.118 SHOW PLACEMENT FOR

SHOW PLACEMENT FOR 用于汇总所有放置策略 (placement policy)，并用统一的形式呈现特定表、数据库或分区的信息。

本语句返回结果中的 Scheduling_State 列标识了 Placement Driver (PD) 在当前对象上的调度进度，有以下可能的结果：

- PENDING: PD 没有进行调度。可能的原因之一是放置规则虽然语法上正确，但集群拓扑并不满足。比如指定 FOLLOWERS=4 但只有 3 个可用作 follower 的 TiKV 实例。
- INPROGRESS: PD 正在进行调度。
- SCHEDULED: PD 调度完成。

14.11.2.118.1 语法图

```
ShowStmt ::=
    "PLACEMENT" "FOR" ShowPlacementTarget

ShowPlacementTarget ::=
    DatabaseSym DBName
|   "TABLE" TableName
|   "TABLE" TableName "PARTITION" Identifier
```

14.11.2.118.2 示例

```
CREATE PLACEMENT POLICY p1 PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1" FOLLOWERS=4;
use test;
ALTER DATABASE test PLACEMENT POLICY=p1;
```

```
CREATE TABLE t1 (a INT);
SHOW PLACEMENT FOR DATABASE test;
SHOW PLACEMENT FOR TABLE t1;
SHOW CREATE TABLE t1\G;
CREATE TABLE t3 (a INT) PARTITION BY RANGE (a) (PARTITION p1 VALUES LESS THAN (10), PARTITION p2
    ↪ VALUES LESS THAN (20));
SHOW PLACEMENT FOR TABLE t3 PARTITION p1\G;
```

```
Query OK, 0 rows affected (0.02 sec)

Query OK, 0 rows affected (0.00 sec)

Query OK, 0 rows affected (0.01 sec)

+--
    ↪ -------------+-------------------------------------------------------------------------+------------------+
    ↪
| Target        | Placement                                                               |
    ↪ Scheduling_State |
+--
    ↪ -------------+-------------------------------------------------------------------------+------------------+
    ↪
| DATABASE test | PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1" FOLLOWERS=4 |
    ↪ INPROGRESS       |
+--
    ↪ -------------+-------------------------------------------------------------------------+------------------+
    ↪
1 row in set (0.00 sec)

+---------------+-------------+------------------+
| Target        | Placement   | Scheduling_State |
+---------------+-------------+------------------+
| TABLE test.t1 | FOLLOWERS=4 | INPROGRESS       |
+---------------+-------------+------------------+
1 row in set (0.00 sec)

***************************[ 1. row ]***************************
Table        | t1
Create Table | CREATE TABLE `t1` (
  `a` int(11) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin /*T![placement] PLACEMENT POLICY=`p1`
    ↪  */
1 row in set (0.00 sec)

***************************[ 1. row ]***************************
```

```
Target           | TABLE test.t3 PARTITION p1
Placement        | PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1" FOLLOWERS=4
Scheduling_State | PENDING
1 row in set (0.00 sec)
```

14.11.2.118.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.118.4 另请参阅

- Placement Rules in SQL
- SHOW PLACEMENT
- CREATE PLACEMENT POLICY

14.11.2.119 SHOW PLACEMENT LABELS

`SHOW PLACEMENT LABELS` 汇总可用于放置规则 (Placement Rules) 的标签 (label) 和值。

14.11.2.119.1 语法图

```
ShowStmt ::=
    "PLACEMENT" "LABELS"
```

14.11.2.119.2 示例

```
SHOW PLACEMENT LABELS;
```

```
+--------+----------------+
| Key    | Values         |
+--------+----------------+
| region | ["us-east-1"]  |
| zone   | ["us-east-1a"] |
+--------+----------------+
2 rows in set (0.00 sec)
```

14.11.2.119.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.119.4 另请参阅

- Placement Rules in SQL
- SHOW PLACEMENT
- CREATE PLACEMENT POLICY

14.11.2.120 SHOW PLUGINS

`SHOW PLUGINS` 用于查看 TiDB 安装的插件，各个插件运行的状态以及插件版本信息。

14.11.2.120.1 语法图

ShowStmt:

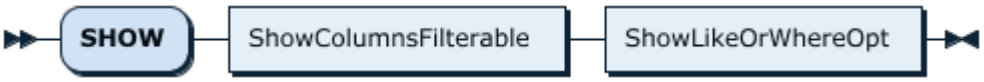


图 426: ShowStmt

ShowTargetFilterable:

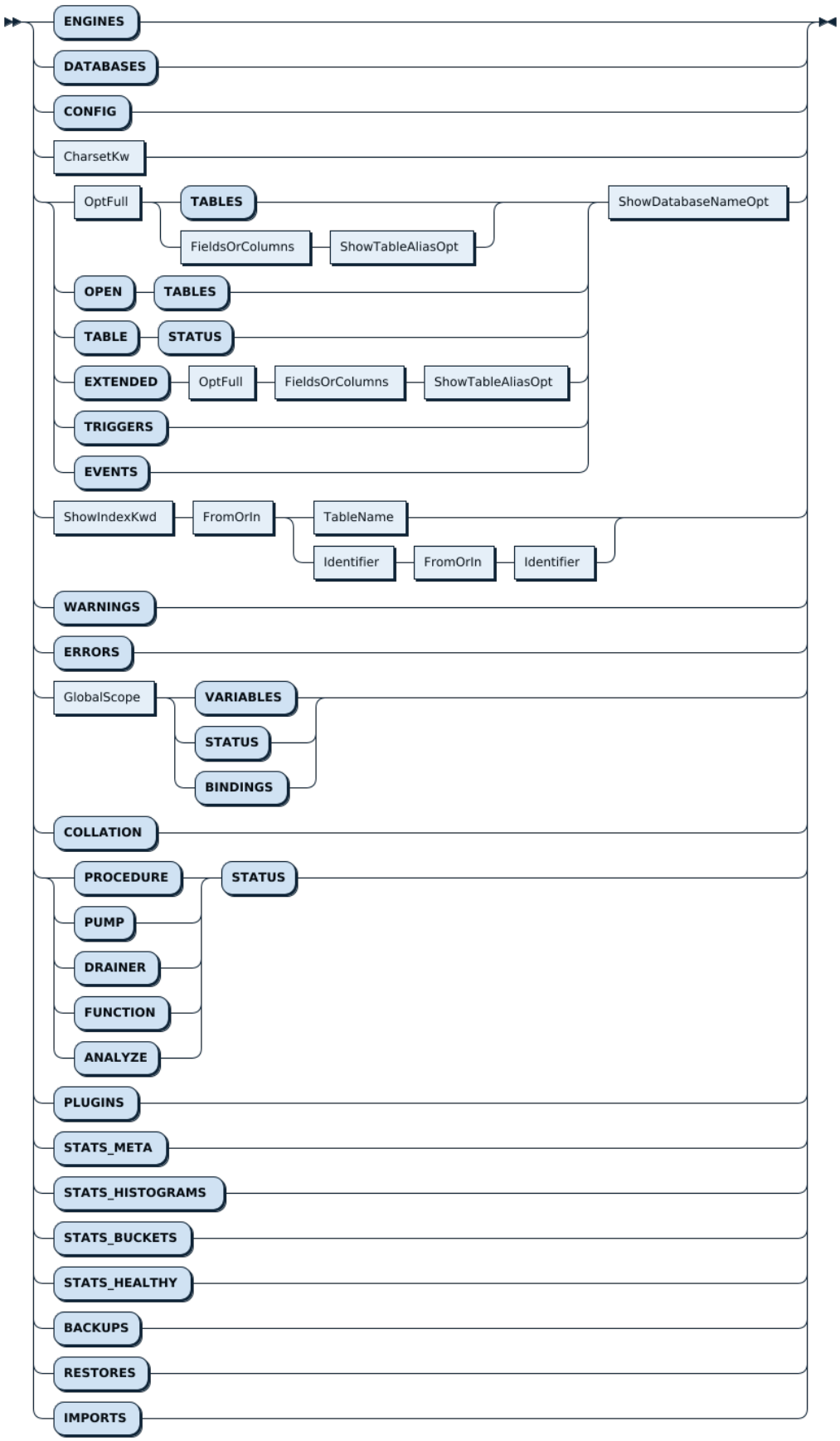


图 427: ShowTargetFilterable

14.11.2.120.2 示例

```
SHOW PLUGINS;
```

```
+-------+--------------+-------+----------------------------+---------+---------+
| Name  | Status       | Type  | Library                    | License | Version |
+-------+--------------+-------+----------------------------+---------+---------+
| audit | Ready-enable | Audit | /tmp/tidb/plugin/audit-1.so |         | 1       |
+-------+--------------+-------+----------------------------+---------+---------+
1 row in set (0.000 sec)
```

SHOW PLUGINS 也支持模糊匹配：

```
SHOW PLUGINS LIKE 'a%';
```

```
+-------+--------------+-------+----------------------------+---------+---------+
| Name  | Status       | Type  | Library                    | License | Version |
+-------+--------------+-------+----------------------------+---------+---------+
| audit | Ready-enable | Audit | /tmp/tidb/plugin/audit-1.so |         | 1       |
+-------+--------------+-------+----------------------------+---------+---------+
1 row in set (0.000 sec)
```

14.11.2.120.3 MySQL 兼容性

SHOW PLUGINS 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.121 SHOW PRIVILEGES

SHOW PRIVILEGES 语句用于显示 TiDB 中可分配权限的列表。此列表为静态列表，不反映当前用户的权限。

14.11.2.121.1 语法图

ShowStmt:

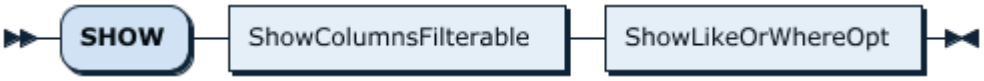


图 428: ShowStmt

14.11.2.121.2 示例

```
show privileges;
```

```
mysql> show privileges;
+--
    ↪ ------------------------------+----------------------------------------+-------------------------------
    ↪
```

```
| Privilege                        | Context                                | Comment
    ↪                                                  |
+--
    ↪ -------------------------------+---------------------------------------+-------------------------------
    ↪
| Alter                            | Tables                                 | To alter the table
    ↪                                        |
| Alter routine                    | Functions,Procedures                   | To alter or drop
    ↪ stored functions/procedures          |
| Config                           | Server Admin                           | To use SHOW CONFIG
    ↪ and SET CONFIG statements          |
| Create                           | Databases,Tables,Indexes               | To create new
    ↪ databases and tables                     |
| Create routine                   | Databases                              | To use CREATE
    ↪ FUNCTION/PROCEDURE                       |
| Create role                      | Server Admin                           | To create new roles
    ↪                                       |
| Create temporary tables          | Databases                              | To use CREATE
    ↪ TEMPORARY TABLE                          |
| Create view                      | Tables                                 | To create new views
    ↪                                       |
| Create user                      | Server Admin                           | To create new users
    ↪                                       |
| Delete                           | Tables                                 | To delete existing
    ↪ rows                                 |
| Drop                             | Databases,Tables                       | To drop databases,
    ↪ tables, and views                    |
| Drop role                        | Server Admin                           | To drop roles
    ↪                                           |
| Event                            | Server Admin                           | To create, alter,
    ↪ drop and execute events              |
| Execute                          | Functions,Procedures                   | To execute stored
    ↪ routines                             |
| File                             | File access on server                  | To read and write
    ↪ files on the server                  |
| Grant option                     | Databases,Tables,Functions,Procedures | To give to other
    ↪ users those privileges you possess   |
| Index                            | Tables                                 | To create or drop
    ↪ indexes                              |
| Insert                           | Tables                                 | To insert data into
    ↪ tables                               |
| Lock tables                      | Databases                              | To use LOCK TABLES (
    ↪ together with SELECT privilege)   |
| Process                          | Server Admin                           | To view the plain
    ↪ text of currently executing queries |
```

```
| Proxy                          | Server Admin                          | To make proxy user
    ↪ possible                          |
| References                     | Databases,Tables                      | To have references on
    ↪  tables                          |
| Reload                         | Server Admin                          | To reload or refresh
    ↪ tables, logs and privileges      |
| Replication client             | Server Admin                          | To ask where the
    ↪ slave or master servers are          |
| Replication slave              | Server Admin                          | To read binary log
    ↪ events from the master            |
| Select                         | Tables                                | To retrieve rows from
    ↪  table                           |
| Show databases                 | Server Admin                          | To see all databases
    ↪ with SHOW DATABASES              |
| Show view                      | Tables                                | To see views with
    ↪ SHOW CREATE VIEW                   |
| Shutdown                       | Server Admin                          | To shut down the
    ↪ server                             |
| Super                          | Server Admin                          | To use KILL thread,
    ↪ SET GLOBAL, CHANGE MASTER, etc.   |
| Trigger                        | Tables                                | To use triggers
    ↪                                      |
| Create tablespace              | Server Admin                          | To create/alter/drop
    ↪ tablespaces                      |
| Update                         | Tables                                | To update existing
    ↪ rows                             |
| Usage                          | Server Admin                          | No privileges - allow
    ↪  connect only                    |
| BACKUP_ADMIN                   | Server Admin                          |
    ↪                                                    |
| RESTORE_ADMIN                  | Server Admin                          |
    ↪                                                    |
| SYSTEM_USER                    | Server Admin                          |
    ↪                                                    |
| SYSTEM_VARIABLES_ADMIN         | Server Admin                          |
    ↪                                                    |
| ROLE_ADMIN                     | Server Admin                          |
    ↪                                                    |
| CONNECTION_ADMIN               | Server Admin                          |
    ↪                                                    |
| PLACEMENT_ADMIN                | Server Admin                          |
    ↪                                                    |
| DASHBOARD_CLIENT               | Server Admin                          |
    ↪                                                    |
```

```
| RESTRICTED_TABLES_ADMIN         | Server Admin                          |
    ↪                                                          |
| RESTRICTED_STATUS_ADMIN         | Server Admin                          |
    ↪                                                          |
| RESTRICTED_VARIABLES_ADMIN      | Server Admin                          |
    ↪                                                          |
| RESTRICTED_USER_ADMIN           | Server Admin                          |
    ↪                                                          |
| RESTRICTED_CONNECTION_ADMIN     | Server Admin                          |
    ↪                                                          |
| RESTRICTED_REPLICA_WRITER_ADMIN | Server Admin                          |
    ↪                                                          |
| RESOURCE_GROUP_ADMIN            | Server Admin                          |
    ↪                                                          |
+--
    ↪ -------------------------------+---------------------------------------+-------------------------------
    ↪
49 rows in set (0.00 sec)
```

14.11.2.121.3 MySQL 兼容性

SHOW PRIVILEGES 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.121.4 另请参阅

- SHOW GRANTS
- GRANT <privileges>

14.11.2.122 SHOW [FULL] PROCESSLIST

SHOW [FULL] PROCESSLIST 语句列出连接到相同 TiDB 服务器的当前会话。

14.11.2.122.1 语法图

ShowProcesslistStmt:



图 429: ShowProcesslistStmt

OptFull:



图 430: OptFull

14.11.2.122.2 示例

```
SHOW PROCESSLIST;
```

```
+------+------+-----------------+------+---------+------+------------+------------------+
| Id   | User | Host            | db   | Command | Time | State      | Info             |
+------+------+-----------------+------+---------+------+------------+------------------+
|    5 | root | 127.0.0.1:45970 | test | Query   |    0 | autocommit | SHOW PROCESSLIST |
+------+------+-----------------+------+---------+------+------------+------------------+
1 rows in set (0.00 sec)
```

以上返回结果中的主要字段描述如下：

- Command：SQL 语句的类型，通常值为 Query。
- Time：SQL 语句开始执行的时间。
- State：SQL 语句的状态。常见的值是 autocommit，表示该 SQL 语句是自动提交的。in transaction 表示该 SQL 语句处于事务中。
- Info：表示具体的 SQL 文本。除非指定可选关键字 FULL，否则文本会被截断。

14.11.2.122.3 MySQL 兼容性

- TiDB 中的 State 列是非描述性的。在 TiDB 中，将状态表示为单个值更复杂，因为查询是并行执行的，而且每个 Go 线程在任一时刻都有不同的状态。

14.11.2.122.4 另请参阅

- KILL [TIDB]

14.11.2.123 SHOW PROFILES

SHOW PROFILES 语句目前只会返回空结果。

14.11.2.123.1 语法图

ShowStmt:

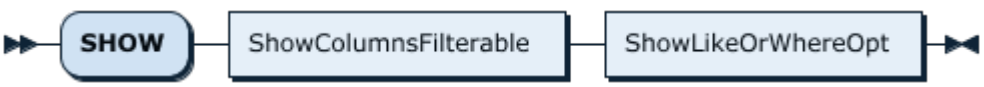


图 431: ShowStmt

14.11.2.123.2 示例

```
SHOW PROFILES
```

```
Empty set (0.00 sec)
```

14.11.2.123.3 MySQL 兼容性

该语句仅与 MySQL 兼容，无其他作用。执行 SHOW PROFILES 始终返回空结果。

14.11.2.124 SHOW PUMP STATUS

SHOW PUMP STATUS 语句用于显示集群中所有 Pump 的状态信息。

14.11.2.124.1 示例

```
SHOW PUMP STATUS;
```

```
+--------|----------------|--------|--------------------|---------------------|
| NodeID |     Address    | State  |   Max_Commit_Ts    |    Update_Time      |
+--------|----------------|--------|--------------------|---------------------|
| pump1  | 127.0.0.1:8250 | Online | 408553768673342237 | 2019-05-01 00:00:01 |
+--------|----------------|--------|--------------------|---------------------|
| pump2  | 127.0.0.2:8250 | Online | 408553768673342335 | 2019-05-01 00:00:02 |
+--------|----------------|--------|--------------------|---------------------|
2 rows in set (0.00 sec)
```

14.11.2.124.2 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.124.3 另请参阅

- SHOW DRAINER STATUS
- CHANGE PUMP STATUS
- CHANGE DRAINER STATUS

14.11.2.125 SHOW SCHEMAS

SHOW SCHEMAS 语句是SHOW DATABASES 的别名。包含该语句提供了 MySQL 兼容性。

14.11.2.126 SHOW STATS_HEALTHY

SHOW STATS_HEALTHY 语句可以预估统计信息的准确度，也就是健康度。健康度低的表可能会生成次优查询执行计划。

可以通过执行 ANALYZE 表命令来改善表的健康度。当表的健康度下降到低于tidb_auto_analyze_ratio 时，则会自动执行 ANALYZE 命令。

14.11.2.126.1 语法图

ShowStmt

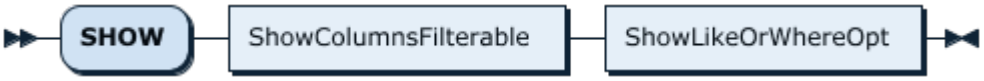


图 432: ShowStmt

ShowTargetFilterable

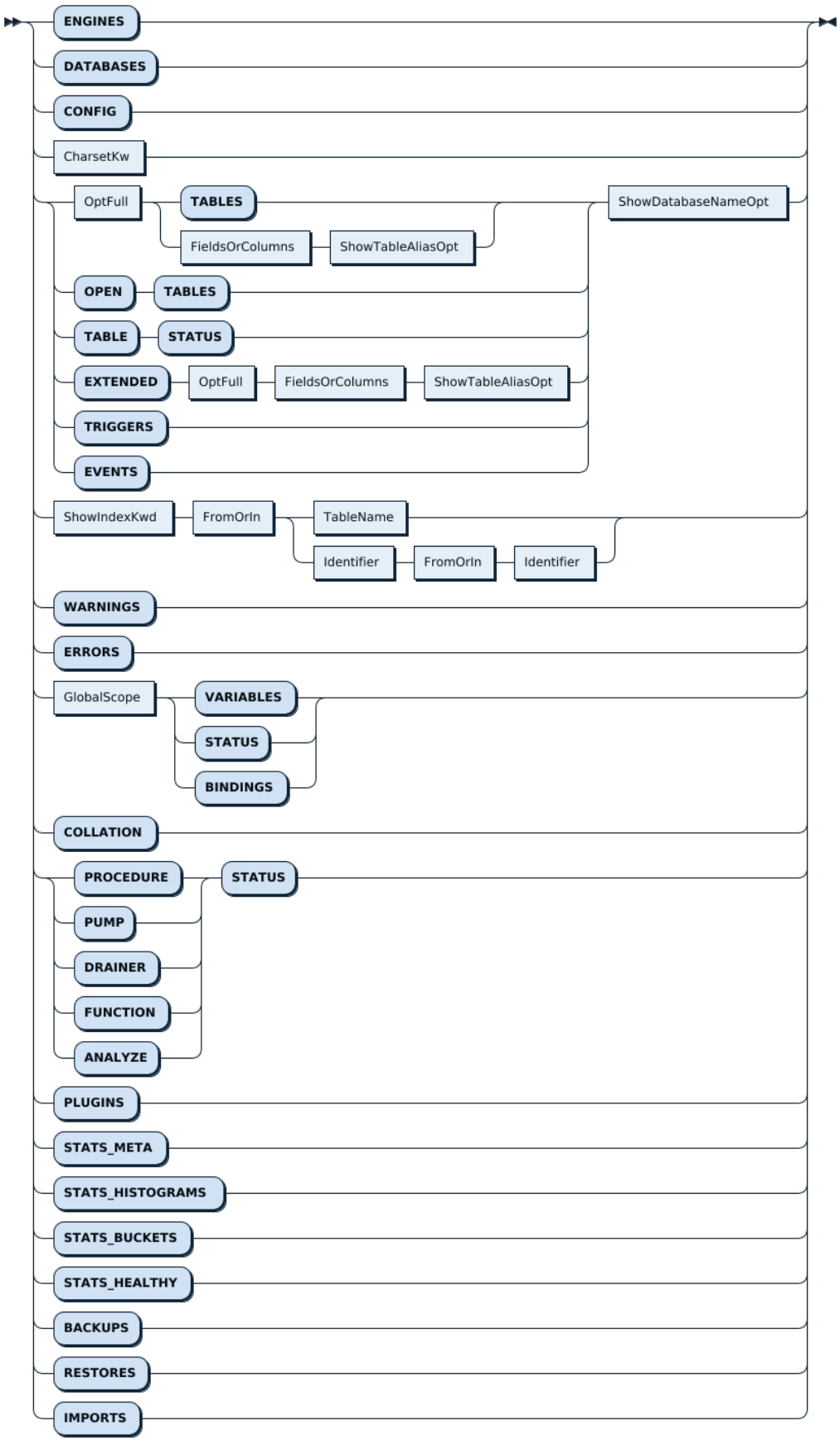


图 433: ShowTargetFilterable

ShowLikeOrWhereOpt

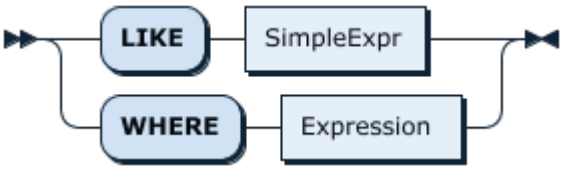


图 434: ShowLikeOrWhereOpt

14.11.2.126.2 示例

加载示例数据并运行 ANALYZE 命令：

```
CREATE TABLE t1 (
 id INT NOT NULL PRIMARY KEY auto_increment,
 b INT NOT NULL,
 pad VARBINARY(255),
 INDEX(b)
);
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM dual;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
SELECT SLEEP(1);
ANALYZE TABLE t1;
SHOW STATS_HEALTHY; # should be 100% healthy
```

```
SHOW STATS_HEALTHY;
```

```
+---------+------------+----------------+---------+
| Db_name | Table_name | Partition_name | Healthy |
+---------+------------+----------------+---------+
| test    | t1         |                |     100 |
+---------+------------+----------------+---------+
1 row in set (0.00 sec)
```

执行批量更新来删除大约 30% 的记录，然后检查统计信息的健康度：

```
DELETE FROM t1 WHERE id BETWEEN 101010 AND 201010; # delete about 30% of records
SHOW STATS_HEALTHY;
```

```
SHOW STATS_HEALTHY;
+---------+------------+----------------+---------+
| Db_name | Table_name | Partition_name | Healthy |
+---------+------------+----------------+---------+
| test    | t1         |                |      50 |
+---------+------------+----------------+---------+
1 row in set (0.00 sec)
```

14.11.2.126.3 MySQL 兼容性

`SHOW STATS_HEALTHY` 语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.126.4 另请参阅

- ANALYZE
- 统计信息简介

14.11.2.127 SHOW STATS_HISTOGRAMS

你可以使用 `SHOW STATS_HISTOGRAMS` 语句查看统计信息中直方图的相关信息。

14.11.2.127.1 语法图

ShowStmt

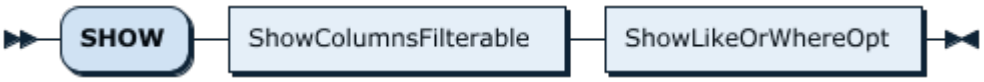


图 435: ShowStmt

ShowTargetFiltertable

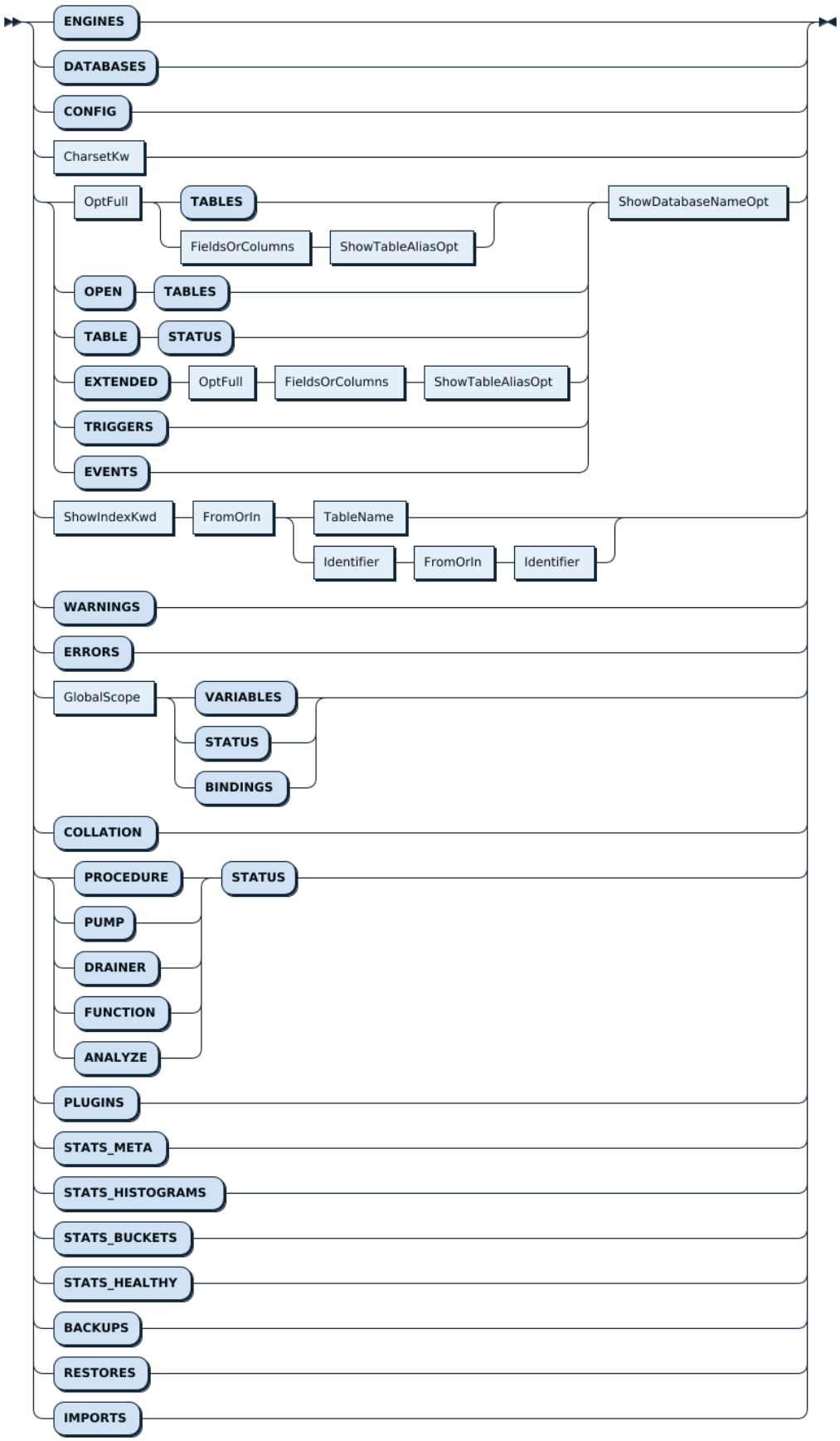


图 436: ShowTargetFilterable

ShowLikeOrWhereOpt

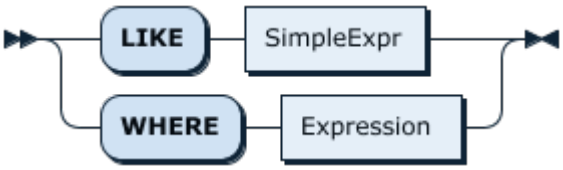


图 437: ShowLikeOrWhereOpt

14.11.2.127.2 示例

```
show stats_histograms;
```

```
+--
    ↪ -------+------------+----------------+-------------+----------+--------------------+-----------------+---
    ↪
| Db_name | Table_name | Partition_name | Column_name | Is_index | Update_time         |
    ↪ Distinct_count | Null_count | Avg_col_size | Correlation |
+--
    ↪ -------+------------+----------------+-------------+----------+--------------------+-----------------+---
    ↪
| test    | t          |                | a           |        0 | 2020-05-25 19:20:00 |
    ↪              7 |          0 |            1 |           1 |
| test    | t2         |                | a           |        0 | 2020-05-25 19:20:01 |
    ↪              6 |          0 |            8 |           0 |
| test    | t2         |                | b           |        0 | 2020-05-25 19:20:01 |
    ↪              6 |          0 |         1.67 |           1 |
+--
    ↪ -------+------------+----------------+-------------+----------+--------------------+-----------------+---
    ↪
3 rows in set (0.00 sec)
```

```
show stats_histograms where table_name = 't2';
```

```
+--
    ↪ -------+------------+----------------+-------------+----------+--------------------+-----------------+---
    ↪
| Db_name | Table_name | Partition_name | Column_name | Is_index | Update_time         |
    ↪ Distinct_count | Null_count | Avg_col_size | Correlation |
+--
    ↪ -------+------------+----------------+-------------+----------+--------------------+-----------------+---
    ↪
| test    | t2         |                | b           |        0 | 2020-05-25 19:20:01 |
    ↪              6 |          0 |         1.67 |           1 |
| test    | t2         |                | a           |        0 | 2020-05-25 19:20:01 |
    ↪              6 |          0 |            8 |           0 |
```

```
+--
    ↪ -------+------------+----------------+-------------+----------+--------------------+-----------+---
    ↪
2 rows in set (0.00 sec)
```

14.11.2.127.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.127.4 另请参阅

- ANALYZE
- 统计信息介绍

14.11.2.128 SHOW STATS_LOCKED

`SHOW STATS_LOCKED` 语句显示统计信息被锁定的表。

> **警告：**
>
> 锁定统计信息目前为实验特性，不建议在生产环境中使用。

14.11.2.128.1 语法图

```
ShowStatsLockedStmt ::= 'SHOW' 'STATS_LOCKED' ShowLikeOrWhereOpt

ShowLikeOrWhereOpt ::= 'LIKE' SimpleExpr | 'WHERE' Expression
```

14.11.2.128.2 示例

创建表 `t`，插入一些数据，在未锁定表 `t` 的统计信息的情况下成功执行 `ANALYZE` 语句。

```
mysql> create table t(a int, b int);
Query OK, 0 rows affected (0.03 sec)

mysql> insert into t values (1,2), (3,4), (5,6), (7,8);
Query OK, 4 rows affected (0.00 sec)
Records: 4  Duplicates: 0  Warnings: 0

mysql> analyze table t;
Query OK, 0 rows affected, 1 warning (0.02 sec)

mysql> show warnings;
```

```
+-------+------+-----------------------------------------------------------------+
| Level | Code | Message                                                         |
+-------+------+-----------------------------------------------------------------+
| Note  | 1105 | Analyze use auto adjusted sample rate 1.000000 for table test.t |
+-------+------+-----------------------------------------------------------------+
1 row in set (0.00 sec)
```

锁定表 t 的统计信息，执行 ANALYZE 语句，使用 SHOW STATS_LOCKED 可查看表 t 的统计信息已被锁定。warning 提示跳过对表 t 的 ANALYZE。

```
mysql> lock stats t;
Query OK, 0 rows affected (0.00 sec)

mysql> show stats_locked;
+---------+------------+----------------+--------+
| Db_name | Table_name | Partition_name | Status |
+---------+------------+----------------+--------+
| test    | t          |                | locked |
+---------+------------+----------------+--------+
1 row in set (0.01 sec)

mysql> analyze table t;
Query OK, 0 rows affected, 2 warnings (0.00 sec)

mysql> show warnings;
+---------+------+-----------------------------------------------------------------+
| Level   | Code | Message                                                         |
+---------+------+-----------------------------------------------------------------+
| Note    | 1105 | Analyze use auto adjusted sample rate 1.000000 for table test.t |
| Warning | 1105 | skip analyze locked table: t                                    |
+---------+------+-----------------------------------------------------------------+
2 rows in set (0.00 sec)
```

解锁表 t 的统计信息，成功执行 ANALYZE 语句。

```
mysql> unlock stats t;
Query OK, 0 rows affected (0.01 sec)

mysql> analyze table t;
Query OK, 0 rows affected, 1 warning (0.03 sec)

mysql> show warnings;
+-------+------+-----------------------------------------------------------------+
| Level | Code | Message                                                         |
+-------+------+-----------------------------------------------------------------+
| Note  | 1105 | Analyze use auto adjusted sample rate 1.000000 for table test.t |
```

```
+-------+------+--------------------------------------------------------------------+
1 row in set (0.00 sec)
```

14.11.2.128.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.128.4 另请参阅

- 统计信息
- LOCK STATS
- UNLOCK STATS

14.11.2.129 SHOW STATS_META

你可以通过 `SHOW STATS_META` 来查看表的总行数以及修改的行数等信息，可以通过 ShowLikeOrWhere 来筛选需要的信息。

目前 `SHOW STATS_META` 会输出 6 列，具体如下：

列名	说明
db_name	数据库名
table_name	表名
partition_name	分区名
update_time	更新时间
modify_count	修改的行数
row_count	总行数

> **注意：**
>
> 在 TiDB 根据 DML 语句自动更新总行数以及修改的行数时，`update_time` 也会被更新，因此并不能认为 `update_time` 是最近一次发生 Analyze 的时间。

14.11.2.129.1 语法图

ShowStmt

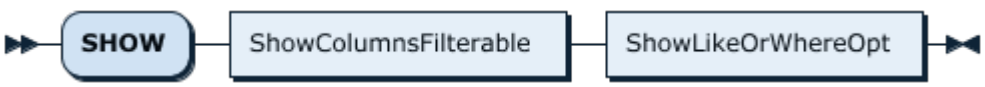


图 438: ShowStmt

ShowTargetFiltertable

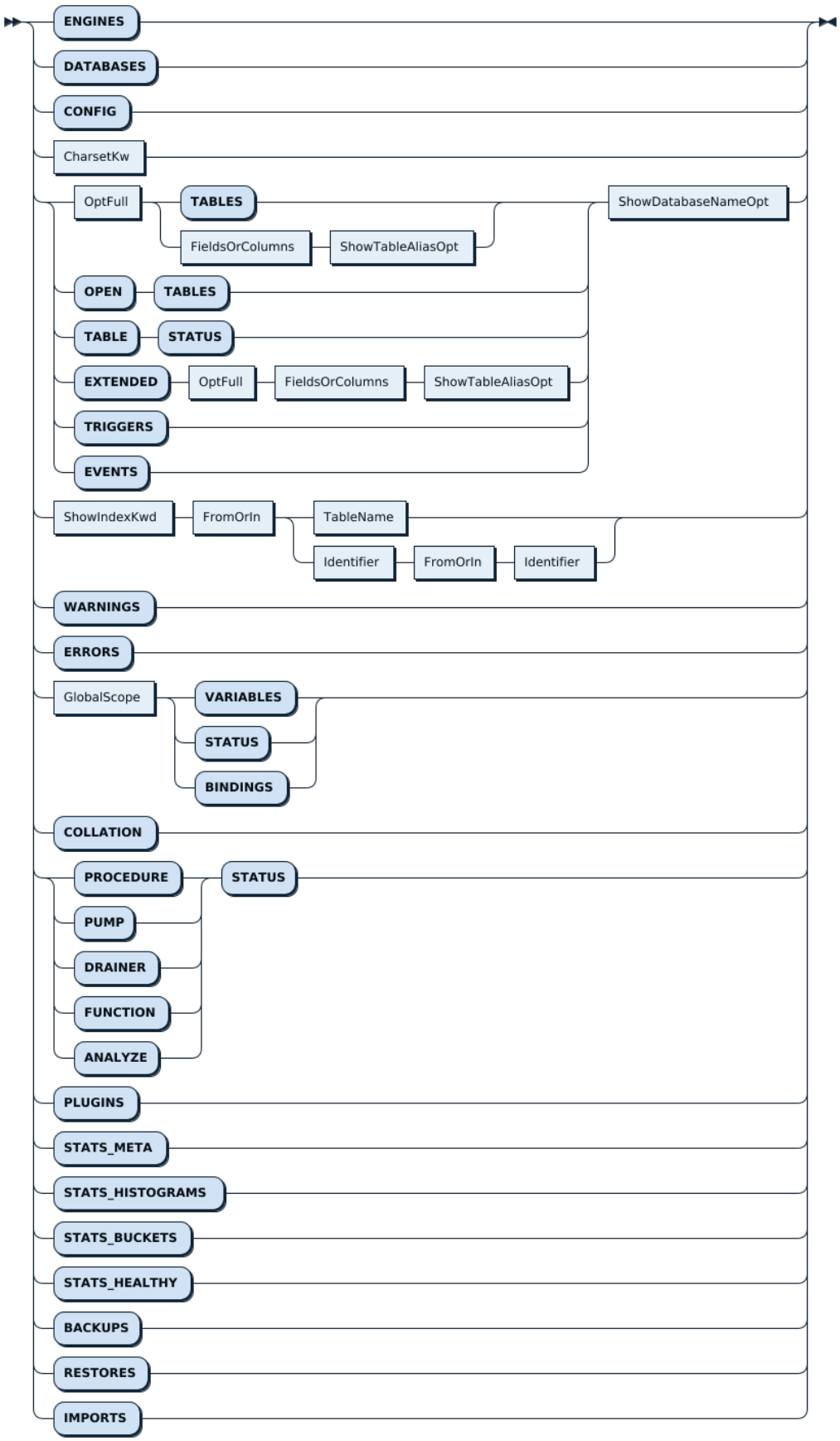


图 439: ShowTargetFilterable

ShowLikeOrWhereOpt

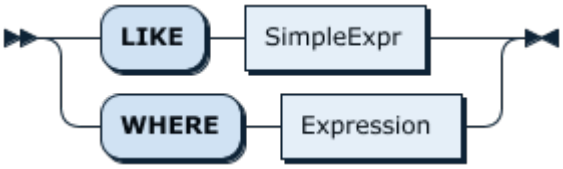


图 440: ShowLikeOrWhereOpt

14.11.2.129.2 示例

```
show stats_meta;
```

```
+---------+------------+----------------+---------------------+--------------+-----------+
| Db_name | Table_name | Partition_name | Update_time         | Modify_count | Row_count |
+---------+------------+----------------+---------------------+--------------+-----------+
| test    | t0         |                | 2020-05-15 16:58:00 |            0 |         0 |
| test    | t1         |                | 2020-05-15 16:58:04 |            0 |         0 |
| test    | t2         |                | 2020-05-15 16:58:11 |            0 |         0 |
| test    | s          |                | 2020-05-22 19:46:43 |            0 |         0 |
| test    | t          |                | 2020-05-25 12:04:21 |            0 |         0 |
+---------+------------+----------------+---------------------+--------------+-----------+
5 rows in set (0.00 sec)
```

```
show stats_meta where table_name = 't2';
```

```
+---------+------------+----------------+---------------------+--------------+-----------+
| Db_name | Table_name | Partition_name | Update_time         | Modify_count | Row_count |
+---------+------------+----------------+---------------------+--------------+-----------+
| test    | t2         |                | 2020-05-15 16:58:11 |            0 |         0 |
+---------+------------+----------------+---------------------+--------------+-----------+
1 row in set (0.00 sec)
```

14.11.2.129.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.129.4 另请参阅

- ANALYZE
- 统计信息介绍

14.11.2.130 SHOW [GLOBAL|SESSION] STATUS

`SHOW [GLOBAL|SESSION] STATUS` 语句用于提供 MySQL 兼容性，对 TiDB 没有作用。因为 TiDB 使用 Prometheus 和 Grafana 而非 `SHOW STATUS` 来进行集中度量收集。

14.11.2.130.1 语法图

ShowStmt:

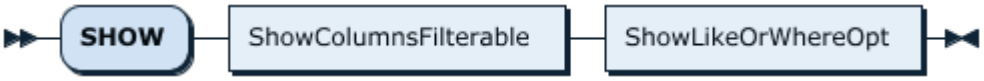


图 441: ShowStmt

ShowTargetFilterable:

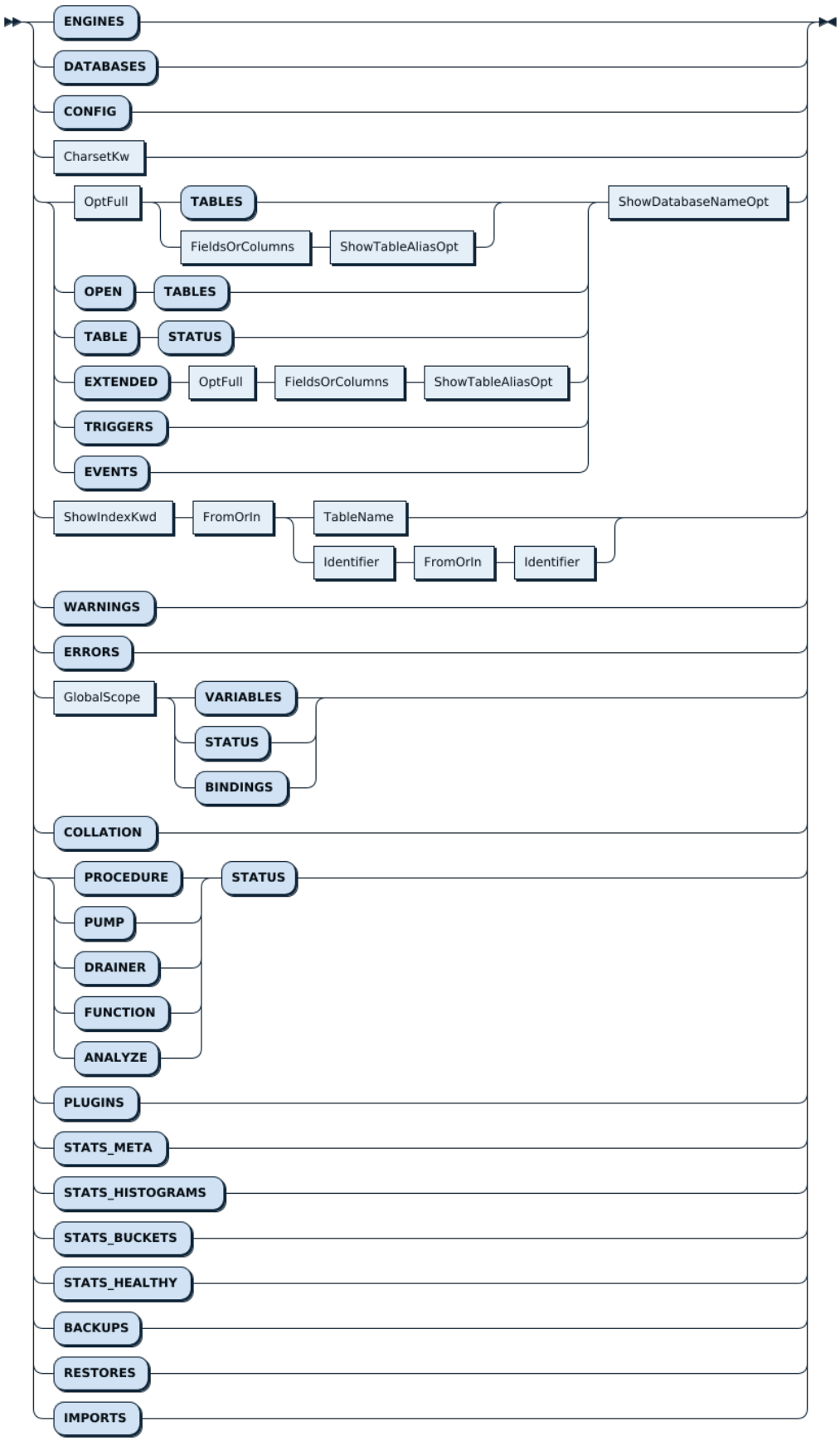


图 442: ShowTargetFilterable

GlobalScope:



图 443: GlobalScope

14.11.2.130.2 示例

```
show status;
```

```
+--------------------+--------------------------------------+
| Variable_name      | Value                                |
+--------------------+--------------------------------------+
| Ssl_cipher_list    |                                      |
| server_id          | 93e2e07d-6bb4-4a1b-90b7-e035fae154fe |
| ddl_schema_version | 141                                  |
| Ssl_verify_mode    | 0                                    |
| Ssl_version        |                                      |
| Ssl_cipher         |                                      |
+--------------------+--------------------------------------+
6 rows in set (0.01 sec)
```

```
show global status;
```

```
+--------------------+--------------------------------------+
| Variable_name      | Value                                |
+--------------------+--------------------------------------+
| Ssl_cipher         |                                      |
| Ssl_cipher_list    |                                      |
| Ssl_verify_mode    | 0                                    |
| Ssl_version        |                                      |
| server_id          | 93e2e07d-6bb4-4a1b-90b7-e035fae154fe |
| ddl_schema_version | 141                                  |
+--------------------+--------------------------------------+
6 rows in set (0.00 sec)
```

14.11.2.130.3 MySQL 兼容性

`SHOW [GLOBAL|SESSION] STATUS` 语句仅用于提供 MySQL 兼容性。

14.11.2.130.4 另请参阅

- FLUSH STATUS

14.11.2.131 SHOW TABLE NEXT_ROW_ID

SHOW TABLE NEXT_ROW_ID 语句用于显示用表中某些特殊列的详情，主要包含以下几种类型：

- TiDB 创建的 AUTO_INCREMENT 类型列，即 _tidb_rowid 列
- 用户创建的 AUTO_INCREMENT 类型列
- 用户创建的AUTO_RANDOM 类型列
- 用户创建的SEQUENCE 对象信息

14.11.2.131.1 语法图

ShowTableNextRowIDStmt:



图 444: ShowTableNextRowIDStmt

TableName:

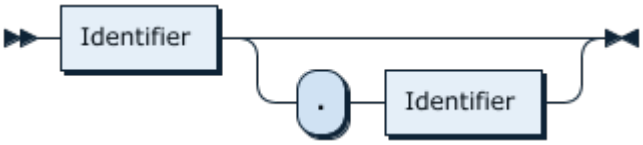


图 445: TableName

14.11.2.131.2 示例

对于新建的表，由于没有任何的 Row ID 分配，NEXT_GLOBAL_ROW_ID 值为 1

```
create table t(a int);
Query OK, 0 rows affected (0.06 sec)
```

```
show table t next_row_id;
+---------+------------+-------------+--------------------+
| DB_NAME | TABLE_NAME | COLUMN_NAME | NEXT_GLOBAL_ROW_ID |
+---------+------------+-------------+--------------------+
| test    | t          | _tidb_rowid |                  1 |
+---------+------------+-------------+--------------------+
1 row in set (0.00 sec)
```

表中写入了数据，负责写入的 TiDB Server 一次性向存储层请求了 30000 个 ID 缓存起来，NEXT_GLOBAL_ROW_ID 值为 30001

```
insert into t values (), (), ();
Query OK, 3 rows affected (0.02 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
show table t next_row_id;
+---------+------------+-------------+--------------------+
| DB_NAME | TABLE_NAME | COLUMN_NAME | NEXT_GLOBAL_ROW_ID |
+---------+------------+-------------+--------------------+
| test    | t          | _tidb_rowid |              30001 |
+---------+------------+-------------+--------------------+
1 row in set (0.00 sec)
```

14.11.2.131.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.131.4 另请参阅

- CREATE TABLE
- AUTO_RANDOM
- CREATE_SEQUENCE

14.11.2.132 SHOW TABLE REGIONS

SHOW TABLE REGIONS 语句用于显示 TiDB 中某个表的 Region 信息。

14.11.2.132.1 语法

```
SHOW TABLE [table_name] REGIONS [WhereClauseOptional];

SHOW TABLE [table_name] INDEX [index_name] REGIONS [WhereClauseOptional];
```

语法图

ShowTableRegionStmt:

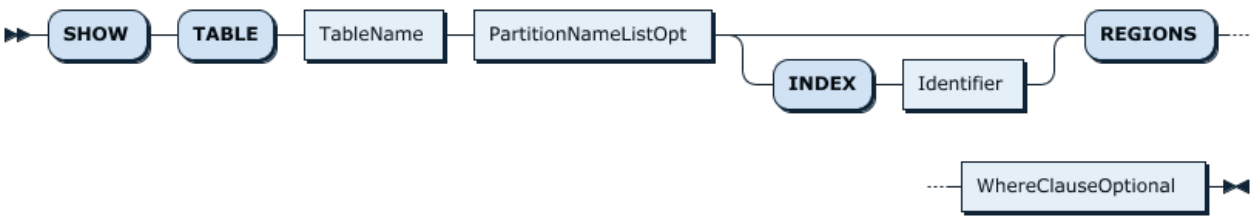


图 446: ShowTableRegionStmt

TableName:

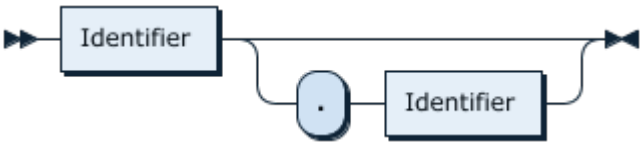


图 447: TableName

PartitionNameListOpt:

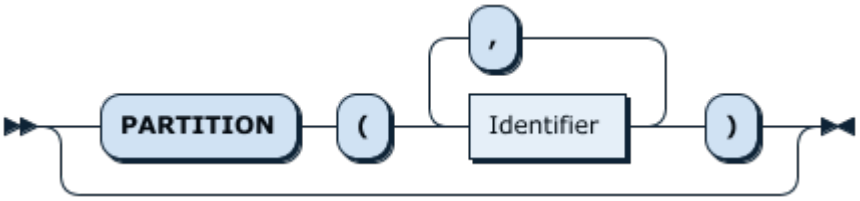


图 448: PartitionNameListOpt

WhereClauseOptional:

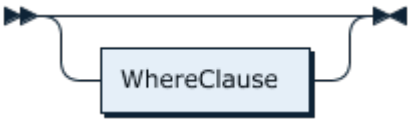


图 449: WhereClauseOptional

WhereClause:



图 450: WhereClause

`SHOW TABLE REGIONS` 会返回如下列：

- `REGION_ID`：Region 的 ID。
- `START_KEY`：Region 的 Start key。
- `END_KEY`：Region 的 End key。
- `LEADER_ID`：Region 的 Leader ID。
- `LEADER_STORE_ID`：Region leader 所在的 store (TiKV) ID。
- `PEERS`：Region 所有副本的 ID。
- `SCATTERING`：Region 是否正在调度中。1 表示正在调度。
- `WRITTEN_BYTES`：估算的 Region 在 1 个心跳周期内的写入数据量大小，单位是 byte。
- `READ_BYTES`：估算的 Region 在 1 个心跳周期内的读数据量大小，单位是 byte。
- `APPROXIMATE_SIZE(MB)`：估算的 Region 的数据量大小，单位是 MB。

- APPROXIMATE_KEYS：估算的 Region 内 Key 的个数。
- SCHEDULING_CONSTRAINTS：Region 所在的表或者分区所关联的放置策略的规则。
- SCHEDULING_STATE：关联了放置策略的 Region 的当前调度状态。

> 注意：
>
> WRITTEN_BYTES，READ_BYTES，APPROXIMATE_SIZE(MB)，APPROXIMATE_KEYS 的值是 PD 根据 Region 的心跳汇报信息统计，估算出来的数据，所以不是精确的数据。

14.11.2.132.2 示例

创建一个示例表，并在若干 Region 中填充足够的数据量：

```
CREATE TABLE t1 (
 id INT NOT NULL PRIMARY KEY auto_increment,
 b INT NOT NULL,
 pad1 VARBINARY(1024),
 pad2 VARBINARY(1024),
 pad3 VARBINARY(1024)
);
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(1024), RANDOM_BYTES(1024),
    ↪ RANDOM_BYTES(1024) FROM dual;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(1024), RANDOM_BYTES(1024),
    ↪ RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(1024), RANDOM_BYTES(1024),
    ↪ RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(1024), RANDOM_BYTES(1024),
    ↪ RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(1024), RANDOM_BYTES(1024),
    ↪ RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(1024), RANDOM_BYTES(1024),
    ↪ RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(1024), RANDOM_BYTES(1024),
    ↪ RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(1024), RANDOM_BYTES(1024),
    ↪ RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(1024), RANDOM_BYTES(1024),
    ↪ RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(1024), RANDOM_BYTES(1024),
    ↪ RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(1024), RANDOM_BYTES(1024),
    ↪ RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(1024), RANDOM_BYTES(1024),
    ↪ RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
```

```
SELECT SLEEP(5);
SHOW TABLE t1 REGIONS;
```

结果显示示例表被切分成多个 Regions。REGION_ID、START_KEY 和 END_KEY 可能不完全匹配：

```
SHOW TABLE t1 REGIONS;
```

```
+-----------+--------------+--------------+-----------+-----------------+-------+------------+---------------+---
    ↪
| REGION_ID | START_KEY    | END_KEY      | LEADER_ID | LEADER_STORE_ID | PEERS | SCATTERING |
    ↪ WRITTEN_BYTES | READ_BYTES | APPROXIMATE_SIZE(MB) | APPROXIMATE_KEYS |
    ↪ SCHEDULING_CONSTRAINTS | SCHEDULING_STATE |
+-----------+--------------+--------------+-----------+-----------------+-------+------------+---------------+---
    ↪
|        94 | t_75_        | t_75_r_31717 |        95 |               1 | 95    |          0 |
    ↪             0 |           0 |                  112 |           207465 |
    ↪                          |                  |
|        96 | t_75_r_31717 | t_75_r_63434 |        97 |               1 | 97    |          0 |
    ↪             0 |           0 |                   97 |                0 |
    ↪                          |                  |
|         2 | t_75_r_63434 |              |         3 |               1 | 3     |          0 |
    ↪     269323514 |   66346110 |                  245 |           162020 |
    ↪                          |                  |
+-----------+--------------+--------------+-----------+-----------------+-------+------------+---------------+---
    ↪
3 rows in set (0.00 sec)
```

解释：

上面 START_KEY 列的值 t_75_r_31717 和 END_KEY 列的值 t_75_r_63434 表示主键在 31717 和 63434 之间的数据存储在该 Region 中。t_75_ 是前缀，表示这是表格 (t) 的 Region，75 是表格的内部 ID。若 START_KEY 或 END_KEY 的一对键值为空，分别表示负无穷大或正无穷大。

TiDB 会根据需要自动重新平衡 Regions。建议使用 SPLIT TABLE REGION 语句手动进行平衡：

```
SPLIT TABLE t1 BETWEEN (31717) AND (63434) REGIONS 2;
```

```
+--------------------+----------------------+
| TOTAL_SPLIT_REGION | SCATTER_FINISH_RATIO |
+--------------------+----------------------+
|                  1 |                    1 |
+--------------------+----------------------+
1 row in set (42.34 sec)
```

```
SHOW TABLE t1 REGIONS;
```

```
+--
    ↪ ---------+--------------+--------------+-----------+-----------------+-------+------------+--------------
    ↪
| REGION_ID | START_KEY     | END_KEY       | LEADER_ID | LEADER_STORE_ID | PEERS | SCATTERING |
    ↪ WRITTEN_BYTES | READ_BYTES | APPROXIMATE_SIZE(MB) | APPROXIMATE_KEYS |
    ↪ SCHEDULING_CONSTRAINTS | SCHEDULING_STATE |
+--
    ↪ ---------+--------------+--------------+-----------+-----------------+-------+------------+--------------
    ↪
|        94 | t_75_         | t_75_r_31717 |        95 |               1 | 95    |          0 |
    ↪             0 |           0 |                  112 |           207465 |
    ↪                           |                  |
|        98 | t_75_r_31717 | t_75_r_47575 |        99 |               1 | 99    |          0 |
    ↪          1325 |           0 |                   53 |            12052 |
    ↪                           |                  |
|        96 | t_75_r_47575 | t_75_r_63434 |        97 |               1 | 97    |          0 |
    ↪          1526 |           0 |                   48 |                0 |
    ↪                           |                  |
|         2 | t_75_r_63434 |              |         3 |               1 | 3     |          0 |
    ↪             0 |    55752049 |                   60 |                0 |
    ↪                           |                  |
+--
    ↪ ---------+--------------+--------------+-----------+-----------------+-------+------------+--------------
    ↪
4 rows in set (0.00 sec)
```

上面的输出结果显示 Region 96 被切分，并创建一个新的 Region 98。切分操作不会影响表中的其他 Region。输出结果同样证实：

- TOTAL_SPLIT_REGION 表示新切的 Region 数量。以上示例新切了 1 个 Region。
- SCATTER_FINISH_RATIO 表示新切的 Region 的打散成功率，1.0 表示都已经打散了。

更详细的示例如下：

```
show table t regions;
```

```
+--
    ↪ ---------+--------------+--------------+-----------+-----------------+--------------+-----------+------
    ↪
| REGION_ID | START_KEY     | END_KEY       | LEADER_ID | LEADER_STORE_ID | PEERS         |
    ↪ SCATTERING | WRITTEN_BYTES | READ_BYTES | APPROXIMATE_SIZE(MB) | APPROXIMATE_KEYS |
    ↪ SCHEDULING_CONSTRAINTS | SCHEDULING_STATE |
+--
    ↪ ---------+--------------+--------------+-----------+-----------------+--------------+-----------+------
    ↪
```

```
| 102        | t_43_r        | t_43_r_20000 | 118        | 7                | 105, 118, 119 | 0
    ↪           | 0             | 0           | 1                    | 0                |
    ↪                         |                  |
| 106        | t_43_r_20000 | t_43_r_40000 | 120        | 7                | 107, 108, 120 | 0
    ↪           | 23            | 0           | 1                    | 0                |
    ↪                         |                  |
| 110        | t_43_r_40000 | t_43_r_60000 | 112        | 9                | 112, 113, 121 | 0
    ↪           | 0             | 0           | 1                    | 0                |
    ↪                         |                  |
| 114        | t_43_r_60000 | t_43_r_80000 | 122        | 7                | 115, 122, 123 | 0
    ↪           | 35            | 0           | 1                    | 0                |
    ↪                         |                  |
| 3          | t_43_r_80000 |              | 93         | 8                | 5, 73, 93     | 0
    ↪           | 0             | 0           | 1                    | 0                |
    ↪                         |                  |
| 98         | t_43_        | t_43_r       | 99         | 1                | 99, 100, 101  | 0
    ↪           | 0             | 0           | 1                    | 0                |
    ↪                         |                  |
+--
    ↪ ---------+--------------+--------------+-----------+----------------+---------------+-------------+------
    ↪
6 rows in set
```

解释：

- Region 102 的 START_KEY 和 END_KEY 中，t_43 是表数据前缀和 table ID，_r 是表 t record 数据的前缀，索引数据的前缀是 _i，所以 Region 102 的 START_KEY 和 END_KEY 表示用来存储 [-inf, 20000) 之前的 record 数据。其他 Region (106, 110, 114, 3) 的存储范围依次类推。
- Region 98 用来存储索引数据存储。表 t 索引数据的起始 key 是 t_43_i，处于 Region 98 的存储范围内。

查看表 t 在 store 1 上的 region，用 where 条件过滤。

```
show table t regions where leader_store_id =1;
```

```
+-----------+-----------+---------+-----------+----------------+--------------+------------+---------------+----
    ↪
| REGION_ID | START_KEY | END_KEY | LEADER_ID | LEADER_STORE_ID | PEERS        | SCATTERING |
    ↪ WRITTEN_BYTES | READ_BYTES | APPROXIMATE_SIZE(MB) | APPROXIMATE_KEYS |
    ↪ SCHEDULING_CONSTRAINTS | SCHEDULING_STATE |
+-----------+-----------+---------+-----------+----------------+--------------+------------+---------------+----
    ↪
| 98        | t_43_     | t_43_r  | 99        | 1              | 99, 100, 101 | 0          | 0
    ↪             | 0          | 1                    | 0               |
    ↪                         |                  |
+-----------+-----------+---------+-----------+----------------+--------------+------------+---------------+----
    ↪
```

用 SPLIT TABLE REGION 语法切分索引数据的 Region，下面语法把表 t 的索引 name 数据在 [a,z] 范围内切分成 2 个 Region。

```
split table t index name between ("a") and ("z") regions 2;
```

```
+--------------------+----------------------+
| TOTAL_SPLIT_REGION | SCATTER_FINISH_RATIO |
+--------------------+----------------------+
| 2                  | 1.0                  |
+--------------------+----------------------+
1 row in set
```

现在表 t 一共有 7 个 Region，其中 5 个 Region (102, 106, 110, 114, 3) 用来存表 t 的 record 数据，另外 2 个 Region (135, 98) 用来存 name 索引的数据。

```
show table t regions;
```

```
+-----------+-----------------------------+-----------------------------+-----------+-----------------+-----------
    ↪
| REGION_ID | START_KEY                   | END_KEY                     | LEADER_ID |
    ↪ LEADER_STORE_ID | PEERS         | SCATTERING | WRITTEN_BYTES | READ_BYTES |
    ↪ APPROXIMATE_SIZE(MB) | APPROXIMATE_KEYS | SCHEDULING_CONSTRAINTS | SCHEDULING_STATE |
+-----------+-----------------------------+-----------------------------+-----------+-----------------+-----------
    ↪
| 102       | t_43_r                      | t_43_r_20000                | 118       | 7
    ↪                | 105, 118, 119 | 0          | 0             | 0          | 1
    ↪                    | 0               |                         |                  |
| 106       | t_43_r_20000                | t_43_r_40000                | 120       | 7
    ↪                | 108, 120, 126 | 0          | 0             | 0          | 1
    ↪                    | 0               |                         |                  |
| 110       | t_43_r_40000                | t_43_r_60000                | 112       | 9
    ↪                | 112, 113, 121 | 0          | 0             | 0          | 1
    ↪                    | 0               |                         |                  |
| 114       | t_43_r_60000                | t_43_r_80000                | 122       | 7
    ↪                | 115, 122, 123 | 0          | 35            | 0          | 1
    ↪                    | 0               |                         |                  |
| 3         | t_43_r_80000                |                             | 93        | 8
    ↪                | 73, 93, 128   | 0          | 0             | 0          | 1
    ↪                    | 0               |                         |                  |
| 135       | t_43_i_1_                   | t_43_i_1_016d80000000000000 | 139       | 2
    ↪                | 138, 139, 140 | 0          | 35            | 0          | 1
    ↪                    | 0               |                         |                  |
| 98        | t_43_i_1_016d80000000000000 | t_43_r                      | 99        | 1
    ↪                | 99, 100, 101  | 0          | 0             | 0          | 1
    ↪                    | 0               |                         |                  |
+-----------+-----------------------------+-----------------------------+-----------+-----------------+-----------
    ↪
```

```
7 rows in set
```

14.11.2.132.3 MySQL 兼容性

`SHOW TABLE REGIONS` 语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.132.4 另请参阅

- SPLIT REGION
- CREATE TABLE

14.11.2.133 SHOW TABLE STATUS

`SHOW TABLE STATUS` 语句用于显示 TiDB 中表的各种统计信息。如果显示统计信息过期，建议运行`ANALYZE TABLE`。

14.11.2.133.1 语法图

ShowTableStatusStmt:

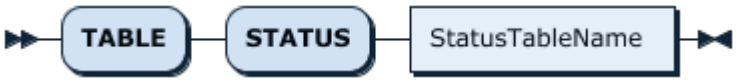


图 451: ShowTableStatusStmt

FromOrIn:

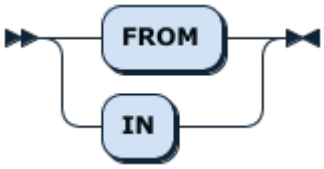


图 452: FromOrIn

StatusTableName:



图 453: StatusTableName

14.11.2.133.2 示例

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, c1 INT NOT NULL);
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
INSERT INTO t1 (c1) VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 5 rows affected (0.02 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
SHOW TABLE STATUS LIKE 't1';
```

```
*************************** 1. row ***************************
           Name: t1
         Engine: InnoDB
        Version: 10
     Row_format: Compact
           Rows: 0
 Avg_row_length: 0
    Data_length: 0
Max_data_length: 0
   Index_length: 0
      Data_free: 0
 Auto_increment: 30001
    Create_time: 2019-04-19 08:32:06
    Update_time: NULL
     Check_time: NULL
      Collation: utf8mb4_bin
       Checksum:
 Create_options:
        Comment:
1 row in set (0.00 sec)
```

```
analyze table t1;
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
SHOW TABLE STATUS LIKE 't1';
```

```
*************************** 1. row ***************************
           Name: t1
         Engine: InnoDB
        Version: 10
     Row_format: Compact
           Rows: 5
 Avg_row_length: 16
    Data_length: 80
Max_data_length: 0
   Index_length: 0
```

```
      Data_free: 0
 Auto_increment: 30001
    Create_time: 2019-04-19 08:32:06
    Update_time: NULL
     Check_time: NULL
      Collation: utf8mb4_bin
       Checksum:
 Create_options:
        Comment:
1 row in set (0.00 sec)
```

14.11.2.133.3 MySQL 兼容性

`SHOW TABLE STATUS` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.133.4 另请参阅

- SHOW TABLES
- CREATE TABLE
- DROP TABLE
- SHOW CREATE TABLE

14.11.2.134 SHOW [FULL] TABLES

`SHOW [FULL] TABLES` 语句用于显示当前所选数据库中表和视图的列表。可选关键字 `FULL` 说明表的类型是 `BASE TABLE` 还是 `VIEW`。

若要在不同的数据库中显示表，可使用 `SHOW TABLES IN DatabaseName` 语句。

14.11.2.134.1 语法图

ShowTablesStmt:



图 454: ShowTablesStmt

OptFull:

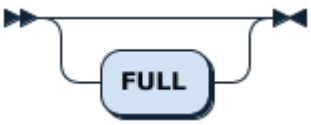


图 455: OptFull

ShowDatabaseNameOpt:

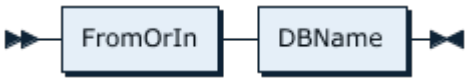


图 456: ShowDatabaseNameOpt

ShowLikeOrWhereOpt:

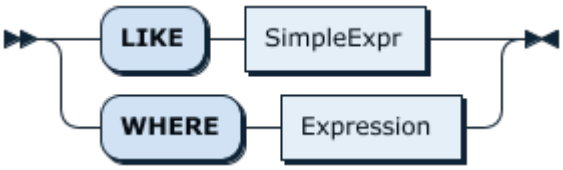


图 457: ShowLikeOrWhereOpt

14.11.2.134.2 示例

```
mysql> CREATE TABLE t1 (a int);
Query OK, 0 rows affected (0.12 sec)

mysql> CREATE VIEW v1 AS SELECT 1;
Query OK, 0 rows affected (0.10 sec)

mysql> SHOW TABLES;
+----------------+
| Tables_in_test |
+----------------+
| t1             |
| v1             |
+----------------+
2 rows in set (0.00 sec)

mysql> SHOW FULL TABLES;
+----------------+------------+
| Tables_in_test | Table_type |
+----------------+------------+
| t1             | BASE TABLE |
| v1             | VIEW       |
+----------------+------------+
2 rows in set (0.00 sec)

mysql> SHOW TABLES IN mysql;
+-------------------------+
| Tables_in_mysql         |
+-------------------------+
```

```
| GLOBAL_VARIABLES        |
| bind_info               |
| columns_priv            |
| db                      |
| default_roles           |
| expr_pushdown_blacklist |
| gc_delete_range         |
| gc_delete_range_done    |
| global_priv             |
| help_topic              |
| opt_rule_blacklist      |
| role_edges              |
| stats_buckets           |
| stats_feedback          |
| stats_histograms        |
| stats_meta              |
| stats_top_n             |
| tables_priv             |
| tidb                    |
| user                    |
+-------------------------+
20 rows in set (0.00 sec)
```

14.11.2.134.3 MySQL 兼容性

`SHOW [FULL] TABLES` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.134.4 另请参阅

- CREATE TABLE
- DROP TABLE
- SHOW CREATE TABLE

14.11.2.135 SHOW [GLOBAL|SESSION] VARIABLES

`SHOW [GLOBAL|SESSION] VARIABLES` 语句用于显示 `GLOBAL` 或 `SESSION` 范围的变量列表。如果未指定范围，则应用默认范围 `SESSION`。

14.11.2.135.1 语法图

ShowStmt:

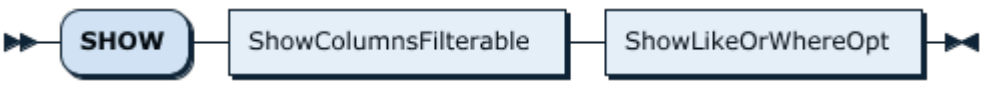


图 458: ShowStmt

ShowTargetFilterable:

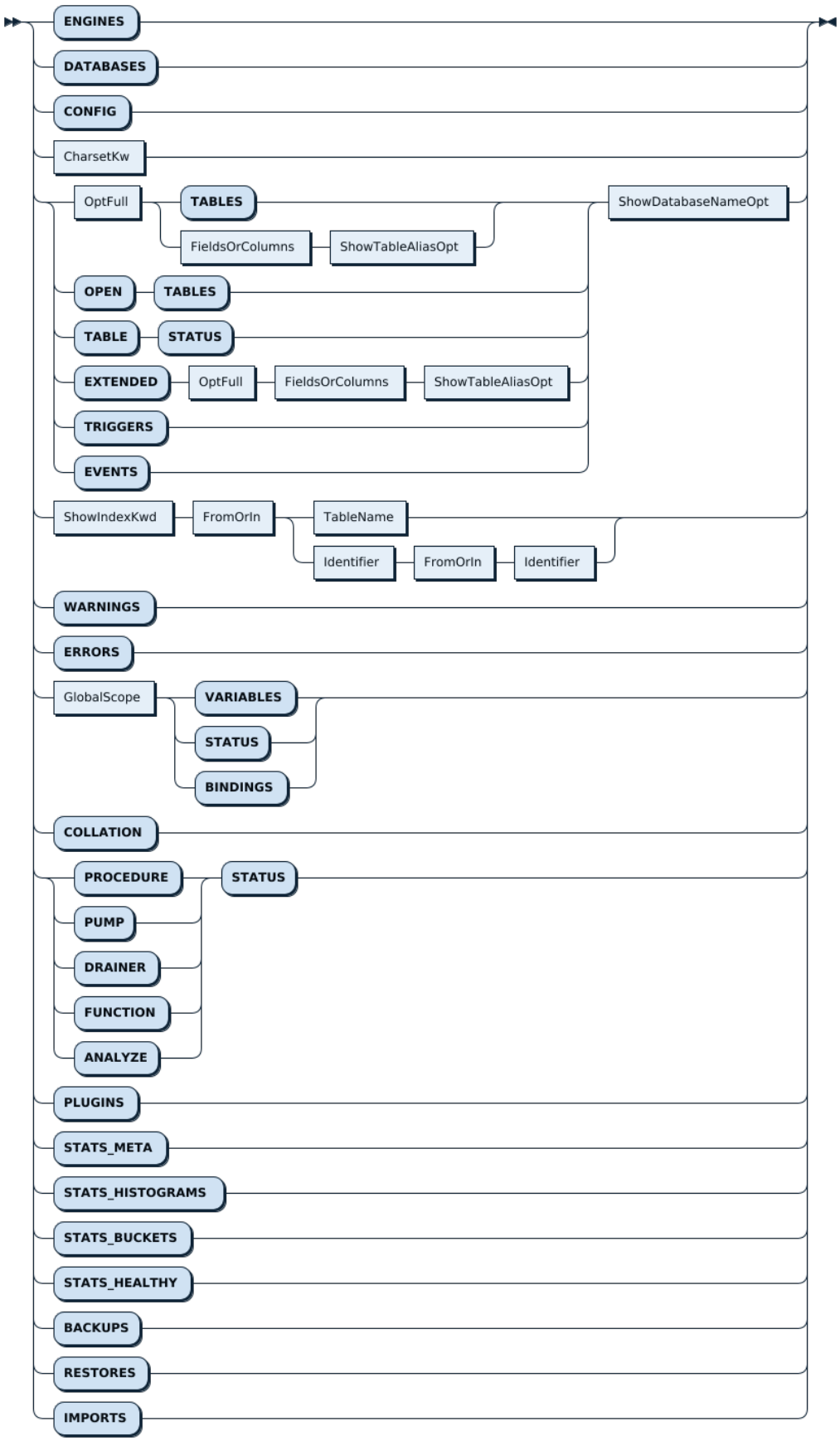


图 459: ShowTargetFilterable

GlobalScope:



图 460: GlobalScope

14.11.2.135.2 示例

查看 TiDB 定义的专用系统变量，关于这些变量的含义参见系统变量和语法。

```
SHOW GLOBAL VARIABLES LIKE 'tidb%';
```

```
+-----------------------------------+---------------------+
| Variable_name                     | Value               |
+-----------------------------------+---------------------+
| tidb_allow_batch_cop              | 0                   |
| tidb_allow_remove_auto_inc        | 0                   |
| tidb_auto_analyze_end_time        | 23:59 +0000         |
| tidb_auto_analyze_ratio           | 0.5                 |
| tidb_auto_analyze_start_time      | 00:00 +0000         |
| tidb_backoff_lock_fast            | 100                 |
| tidb_backoff_weight               | 2                   |
| tidb_batch_commit                 | 0                   |
| tidb_batch_delete                 | 0                   |
| tidb_build_stats_concurrency      | 4                   |
| tidb_capture_plan_baselines       | off                 |
| tidb_check_mb4_value_in_utf8      | 1                   |
| tidb_checksum_table_concurrency   | 4                   |
| tidb_config                       |                     |
| tidb_constraint_check_in_place    | 0                   |
| tidb_current_ts                   | 0                   |
| tidb_ddl_error_count_limit        | 512                 |
| tidb_ddl_reorg_batch_size         | 256                 |
| tidb_ddl_reorg_priority           | PRIORITY_LOW        |
| tidb_ddl_reorg_worker_cnt         | 4                   |
| tidb_disable_txn_auto_retry       | 1                   |
| tidb_distsql_scan_concurrency     | 15                  |
| tidb_dml_batch_size               | 20000               |
| tidb_enable_cascades_planner      | 0                   |
| tidb_enable_chunk_rpc             | 1                   |
| tidb_enable_collect_execution_info | 1                   |
| tidb_enable_fast_analyze          | 0                   |
```

```
| tidb_enable_index_merge             | 0                    |
| tidb_enable_noop_functions          | 0                    |
| tidb_enable_radix_join              | 0                    |
| tidb_enable_slow_log                | 1                    |
| tidb_enable_stmt_summary            | 1                    |
| tidb_enable_table_partition          | on                   |
| tidb_enable_vectorized_expression   | 1                    |
| tidb_enable_window_function         | 1                    |
| tidb_evolve_plan_baselines          | off                  |
| tidb_evolve_plan_task_end_time      | 23:59 +0000          |
| tidb_evolve_plan_task_max_time      | 600                  |
| tidb_evolve_plan_task_start_time    | 00:00 +0000          |
| tidb_expensive_query_time_threshold | 60                   |
| tidb_force_priority                 | NO_PRIORITY          |
| tidb_general_log                    | 0                    |
| tidb_hash_join_concurrency          | 5                    |
| tidb_hashagg_final_concurrency      | 4                    |
| tidb_hashagg_partial_concurrency    | 4                    |
| tidb_index_join_batch_size          | 25000                |
| tidb_index_lookup_concurrency       | 4                    |
| tidb_index_lookup_join_concurrency  | 4                    |
| tidb_index_lookup_size              | 20000                |
| tidb_index_serial_scan_concurrency  | 1                    |
| tidb_init_chunk_size                | 32                   |
| tidb_isolation_read_engines         | tikv, tiflash, tidb  |
| tidb_low_resolution_tso             | 0                    |
| tidb_max_chunk_size                 | 1024                 |
| tidb_max_delta_schema_count         | 1024                 |
| tidb_mem_quota_hashjoin             | 34359738368          |
| tidb_mem_quota_indexlookupjoin      | 34359738368          |
| tidb_mem_quota_indexlookupreader    | 34359738368          |
| tidb_mem_quota_mergejoin            | 34359738368          |
| tidb_mem_quota_nestedloopapply      | 34359738368          |
| tidb_mem_quota_query                | 1073741824           |
| tidb_mem_quota_sort                 | 34359738368          |
| tidb_mem_quota_topn                 | 34359738368          |
| tidb_metric_query_range_duration    | 60                   |
| tidb_metric_query_step              | 60                   |
| tidb_opt_agg_push_down              | 0                    |
| tidb_opt_concurrency_factor         | 3                    |
| tidb_opt_copcpu_factor              | 3                    |
| tidb_opt_correlation_exp_factor     | 1                    |
| tidb_opt_correlation_threshold      | 0.9                  |
| tidb_opt_cpu_factor                 | 3                    |
| tidb_opt_desc_factor                | 3                    |
```

```
| tidb_opt_disk_factor                | 1.5                 |
| tidb_opt_distinct_agg_push_down     | 0                   |
| tidb_opt_insubq_to_join_and_agg     | 1                   |
| tidb_opt_join_reorder_threshold     | 0                   |
| tidb_opt_memory_factor              | 0.001               |
| tidb_opt_network_factor             | 1                   |
| tidb_opt_scan_factor                | 1.5                 |
| tidb_opt_seek_factor                | 20                  |
| tidb_opt_write_row_id               | 0                   |
| tidb_optimizer_selectivity_level    | 0                   |
| tidb_pprof_sql_cpu                  | 0                   |
| tidb_projection_concurrency         | 4                   |
| tidb_query_log_max_len              | 4096                |
| tidb_record_plan_in_slow_log        | 1                   |
| tidb_replica_read                   | leader              |
| tidb_retry_limit                    | 10                  |
| tidb_row_format_version             | 2                   |
| tidb_scatter_region                 | 0                   |
| tidb_skip_isolation_level_check     | 0                   |
| tidb_skip_utf8_check                | 0                   |
| tidb_slow_log_threshold             | 300                 |
| tidb_slow_query_file                | tidb-slow.log       |
| tidb_snapshot                       |                     |
| tidb_stmt_summary_history_size      | 24                  |
| tidb_stmt_summary_internal_query    | 0                   |
| tidb_stmt_summary_max_sql_length    | 4096                |
| tidb_stmt_summary_max_stmt_count    | 3000                |
| tidb_stmt_summary_refresh_interval  | 1800                |
| tidb_store_limit                    | 0                   |
| tidb_txn_mode                       |                     |
| tidb_use_plan_baselines             | on                  |
| tidb_wait_split_region_finish       | 1                   |
| tidb_wait_split_region_timeout      | 300                 |
| tidb_window_concurrency             | 4                   |
+-------------------------------------+---------------------+
108 rows in set (0.01 sec)
```

```
SHOW GLOBAL VARIABLES LIKE 'time_zone%';
```

```
+---------------+--------+
| Variable_name | Value  |
+---------------+--------+
| time_zone     | SYSTEM |
+---------------+--------+
1 row in set (0.00 sec)
```

14.11.2.135.3 MySQL 兼容性

`SHOW [GLOBAL|SESSION] VARIABLES` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.135.4 另请参阅

- `SET [GLOBAL|SESSION]`

14.11.2.136 SHOW WARNINGS

`SHOW WARNINGS` 语句用于显示当前客户端连接中已执行语句的报错列表。与在 MySQL 中一样，`sql_mode` 极大地影响哪些语句会导致错误与警告。

14.11.2.136.1 语法图

ShowWarningsStmt:



图 461: ShowWarningsStmt

14.11.2.136.2 示例

```
CREATE TABLE t1 (a INT UNSIGNED);
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
INSERT INTO t1 VALUES (0);
```

```
Query OK, 1 row affected (0.02 sec)
```

```
SELECT 1/a FROM t1;
```

```
+------+
| 1/a  |
+------+
| NULL |
+------+
1 row in set, 1 warning (0.00 sec)
```

```
SHOW WARNINGS;
```

```
+---------+------+---------------+
| Level   | Code | Message       |
+---------+------+---------------+
| Warning | 1365 | Division by 0 |
+---------+------+---------------+
1 row in set (0.00 sec)
```

```
INSERT INTO t1 VALUES (-1);
```

```
ERROR 1264 (22003): Out of range value for column 'a' at row 1
```

```
SELECT * FROM t1;
```

```
+------+
| a    |
+------+
|    0 |
+------+
1 row in set (0.00 sec)
```

```
SET sql_mode='';
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
INSERT INTO t1 VALUES (-1);
```

```
Query OK, 1 row affected, 1 warning (0.01 sec)
```

```
SHOW WARNINGS;
```

```
+---------+------+---------------------------+
| Level   | Code | Message                   |
+---------+------+---------------------------+
| Warning | 1690 | constant -1 overflows int |
+---------+------+---------------------------+
1 row in set (0.00 sec)
```

```
SELECT * FROM t1;
```

```
+------+
| a    |
+------+
|    0 |
```

```
|    0 |
+------+
2 rows in set (0.00 sec)
```

14.11.2.136.3 MySQL 兼容性

SHOW WARNINGS 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.136.4 另请参阅

- SHOW ERRORS

14.11.2.137 SHUTDOWN

SHUTDOWN 语句用于在 TiDB 中执行停机操作。执行 SHUTDOWN 语句需要用户拥有 SHUTDOWN privilege。

14.11.2.137.1 语法图

Statement:



图 462: Statement

14.11.2.137.2 示例

```
SHUTDOWN;
```

```
Query OK, 0 rows affected (0.00 sec)
```

14.11.2.137.3 MySQL 兼容性

> **注意：**
>
> 由于 TiDB 是分布式数据库，因此 TiDB 中的停机操作停止的是客户端连接的 TiDB 实例，而不是整个 TiDB 集群。

SHUTDOWN 语句与 MySQL 不完全兼容。如发现任何其他兼容性差异，请在 GitHub 上提交 issue。

14.11.2.138 Split Region 使用文档

在 TiDB 中新建一个表后，默认会单独切分出 1 个Region 来存储这个表的数据，这个默认行为由配置文件中的 `split-table` 控制。当这个 Region 中的数据超过默认 Region 大小限制后，这个 Region 会开始分裂成 2 个 Region。

上述情况中，如果在新建的表上发生大批量写入，则会造成热点，因为开始只有一个 Region，所有的写请求都发生在该 Region 所在的那台 TiKV 上。

为解决上述场景中的热点问题，TiDB 引入了预切分 Region 的功能，即可以根据指定的参数，预先为某个表切分出多个 Region，并打散到各个 TiKV 上去。

14.11.2.138.1 语法图

SplitRegionStmt:



图 463: SplitRegionStmt

SplitSyntaxOption:

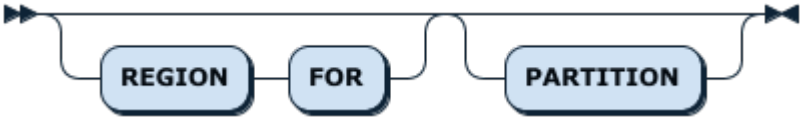


图 464: SplitSyntaxOption

TableName:

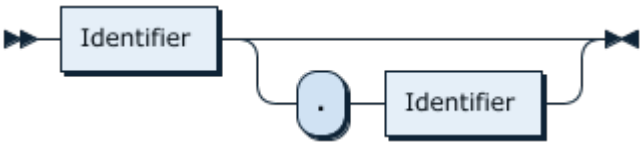


图 465: TableName

PartitionNameListOpt:

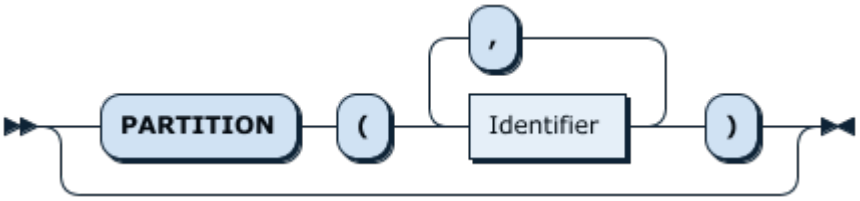


图 466: PartitionNameListOpt

SplitOption:

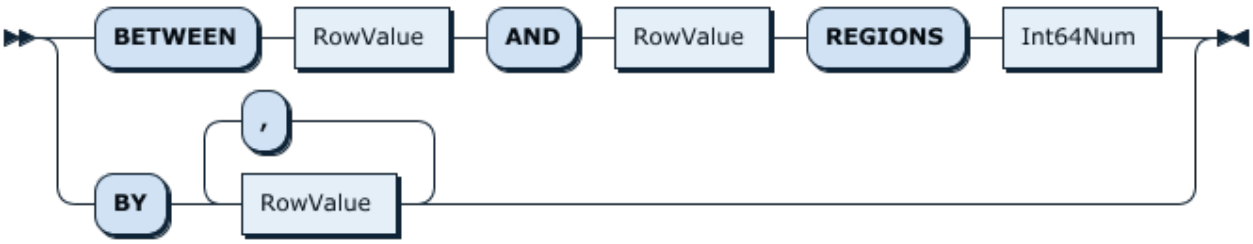


图 467: SplitOption

RowValue:



图 468: RowValue

Int64Num:

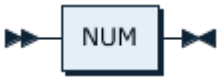


图 469: Int64Num

14.11.2.138.2 Split Region 的使用

Split Region 有 2 种不同的语法，具体如下：

- 均匀切分的语法：

```
SPLIT TABLE table_name [INDEX index_name] BETWEEN (lower_value) AND (upper_value) REGIONS
    ↪ region_num
```

`BETWEEN lower_value AND upper_value REGIONS region_num` 语法是通过指定数据的上、下边界和 Region 数量，然后在上、下边界之间均匀切分出 `region_num` 个 Region。

- 不均匀切分的语法：

```
SPLIT TABLE table_name [INDEX index_name] BY (value_list) [, (value_list)] ...
```

`BY value_list…` 语法将手动指定一系列的点，然后根据这些指定的点切分 Region，适用于数据不均匀分布的场景。

`SPLIT` 语句的返回结果示例如下：

```
+--------------------+----------------------+
| TOTAL_SPLIT_REGION | SCATTER_FINISH_RATIO |
+--------------------+----------------------+
| 4                  | 1.0                  |
+--------------------+----------------------+
```

- TOTAL_SPLIT_REGION：表示新增预切分的 Region 数量。
- SCATTER_FINISH_RATIO：表示新增预切分 Region 中，打散完成的比率。如 1.0 表示全部完成。0.5表示只有一半的 Region 已经打散完成，剩下的还在打散过程中。

> 注意：
>
> 以下会话变量会影响 SPLIT 语句的行为，需要特别注意：
>
> - tidb_wait_split_region_finish：打散 Region 的时间可能较长，由 PD 调度以及 TiKV 的负载情况所决定。这个变量用来设置在执行 SPLIT REGION 语句时，是否同步等待所有 Region 都打散完成后再返回结果给客户端。默认 1 代表等待打散完成后再返回结果。0 代表不等待 Region 打散完成就返回结果。
> - tidb_wait_split_region_timeout：这个变量用来设置 SPLIT REGION 语句的执行超时时间，单位是秒，默认值是 300 秒，如果超时还未完成 Split 操作，就返回一个超时错误。

Split Table Region

表中行数据的 key 由 table_id 和 row_id 编码组成，格式如下：

```
t[table_id]_r[row_id]
```

例如，当 table_id 是 22，row_id 是 11 时：

```
t22_r11
```

同一表中行数据的 table_id 是一样的，但 row_id 肯定不一样，所以可以根据 row_id 来切分 Region。

均匀切分

由于 row_id 是整数，所以根据指定的 lower_value、upper_value 以及 region_num，可以推算出需要切分的 key。TiDB 先计算 step(step = (upper_value - lower_value)/region_num)，然后在 lower_value 和 upper_value 之间每隔 step 区间切一次，最终切出 region_num 个 Region。

例如，对于表 t，如果想要从 minInt64~maxInt64 之间均匀切割出 16 个 Region，可以用以下语句：

```
SPLIT TABLE t BETWEEN (-9223372036854775808) AND (9223372036854775807) REGIONS 16;
```

该语句会把表 t 从 minInt64 到 maxInt64 之间均匀切割出 16 个 Region。如果已知主键的范围没有这么大，比如只会在 0~1000000000 之间，那可以用 0 和 1000000000 分别代替上面的 minInt64 和 maxInt64 来切分 Region。

```
SPLIT TABLE t BETWEEN (0) AND (1000000000) REGIONS 16;
```

不均匀切分

如果已知数据不是均匀分布的，比如想要 -inf ~ 10000 切一个 Region，10000 ~ 90000 切一个 Region，90000 ~ +inf 切一个 Region，可以通过手动指定点来切分 Region，示例如下：

```
SPLIT TABLE t BY (10000), (90000);
```

Split Index Region

表中索引数据的 key 由 `table_id`、`index_id` 以及索引列的值编码组成，格式如下：

```
t[table_id]_i[index_id][index_value]
```

例如，当 `table_id` 是 22，`index_id` 是 5，`index_value` 是 abc 时：

```
t22_i5abc
```

同一表中同一索引数据的 `table_id` 和 `index_id` 是一样的，所以要根据 `index_value` 切分索引 Region。

均匀切分

索引均匀切分与行数据均匀切分的原理一样，只是计算 step 的值较为复杂，因为 `index_value` 可能不是整数。

`upper` 和 `lower` 的值会先编码成 byte 数组，去掉 `lower` 和 `upper` byte 数组的最长公共前缀后，从 `lower` 和 `upper` 各取前 8 字节转成 uint64，再计算 `step = (upper - lower)/num`。计算出 step 后再将 step 编码成 byte 数组，添加到之前 `upper`和 `lower`的最长公共前缀后面组成一个 key 后去做切分。示例如下：

如果索引 idx 的列也是整数类型，可以用如下 SQL 语句切分索引数据：

```
SPLIT TABLE t INDEX idx BETWEEN (-9223372036854775808) AND (9223372036854775807) REGIONS 16;
```

该语句会把表 t 中 idx 索引数据 Region 从 `minInt64` 到 `maxInt64` 之间均匀切割出 16 个 Region。

如果索引 idx1 的列是 varchar 类型，希望根据前缀字母来切分索引数据：

```
SPLIT TABLE t INDEX idx1 BETWEEN ("a") AND ("z") REGIONS 25;
```

该语句会把表 t 中 idx1 索引数据的 Region 从 a~z 切成 25 个 Region，region1 的范围是 [minIndexValue, b)，region2 的范围是 [b, c)，……，region25 的范围是 [y, maxIndexValue)。对于 idx1 索引以 a 为前缀的数据都会写到 region1，以 b 为前缀的索引数据都会写到 region2，以此类推。

上面的切分方法，以 y 和 z 前缀的索引数据都会写到 region 25，因为 z 并不是一个上界，真正的上界是 z 在 ASCII 码中的下一位 `{`，所以更准确的切分方法如下：

```
SPLIT TABLE t INDEX idx1 BETWEEN ("a") AND ("{") REGIONS 26;
```

该语句会把表 t 中 idx1 索引数据的 Region 从 a~{ 切成 26 个 Region，region1 的范围是 [minIndexValue, b)，region2 的范围是 [b, c)，……，region25 的范围是 [y,z)，region26 的范围是 [z, maxIndexValue)。

如果索引 idx2 的列是 timestamp/datetime 等时间类型，希望根据时间区间，按年为间隔切分索引数据，示例如下：

```
SPLIT TABLE t INDEX idx2 BETWEEN ("2010-01-01 00:00:00") AND ("2020-01-01 00:00:00") REGIONS 10;
```

该语句会把表 t 中 idx2 的索引数据 Region 从 2010-01-01 00:00:00 到 2020-01-01 00:00:00 切成 10 个 Region。region1 的范围是从 [minIndexValue, 2011-01-01 00:00:00)，region2 的范围是 [2011-01-01 00:00:00, 2012-01-01 00:00:00)……

如果希望按照天为间隔切分索引，示例如下：

```
SPLIT TABLE t INDEX idx2 BETWEEN ("2020-06-01 00:00:00") AND ("2020-07-01 00:00:00") REGIONS 30;
```

该语句会将表 t 中 idx2 索引位于 2020 年 6 月份的数据按天为间隔切分成 30 个 Region。

其他索引列类型的切分方法也是类似的。

对于联合索引的数据 Region 切分，唯一不同的是可以指定多个 column 的值。

比如索引 idx3 (a, b) 包含 2 列，a 是 timestamp，b 是 int。如果只想根据 a 列做时间范围的切分，可以用切分单列时间索引的 SQL 语句来切分，lower_value 和 upper_velue 中不指定 b 列的值即可。

```
SPLIT TABLE t INDEX idx3 BETWEEN ("2010-01-01 00:00:00") AND ("2020-01-01 00:00:00") REGIONS 10;
```

如果想在时间相同的情况下，根据 b 列再做一次切分，在切分时指定 b 列的值即可。

```
SPLIT TABLE t INDEX idx3 BETWEEN ("2010-01-01 00:00:00", "a") AND ("2010-01-01 00:00:00", "z")
    ↪ REGIONS 10;
```

该语句在 a 列时间前缀相同的情况下，根据 b 列的值从 a~z 切了 10 个 Region。如果指定的 a 列的值不相同，那么可能不会用到 b 列的值。

如果表的主键为非聚簇索引NONCLUSTERED，切分 Region 时需要用反引号 ` 来转义 PRIMARY 关键字。例如：

```
SPLIT TABLE t INDEX `PRIMARY` BETWEEN (-9223372036854775808) AND (9223372036854775807) REGIONS
    ↪ 16;
```

不均匀切分

索引数据也可以根据用户指定的索引值来做切分。

假如有 idx4 (a,b)，其中 a 列是 varchar 类型，b 列是 timestamp 类型。

```
SPLIT TABLE t1 INDEX idx4 BY ("a", "2000-01-01 00:00:01"), ("b", "2019-04-17 14:26:19"), ("c", ""
    ↪ );
```

该语句指定了 3 个值，会切分出 4 个 Region，每个 Region 的范围如下。

```
region1  [ minIndexValue               , ("a", "2000-01-01 00:00:01"))
region2  [("a", "2000-01-01 00:00:01") , ("b", "2019-04-17 14:26:19"))
region3  [("b", "2019-04-17 14:26:19") , ("c", "")                    )
region4  [("c", "")                    , maxIndexValue                )
```

Split 分区表的 Region

预切分分区表的 Region 在使用上和普通表一样，差别是会为每一个 partition 都做相同的切分。

- 均匀切分的语法如下：

```
SPLIT [PARTITION] TABLE t [PARTITION] [(partition_name_list...)] [INDEX index_name] BETWEEN
    ↪ (lower_value) AND (upper_value) REGIONS region_num
```

- 不均匀切分的语法如下：

```
SPLIT [PARTITION] TABLE table_name [PARTITION (partition_name_list...)] [INDEX index_name]
    ↪ BY (value_list) [, (value_list)] ...
```

Split 分区表的 Region 示例

1. 首先创建一个分区表。如果你要建一个 Hash 分区表，分成 2 个 partition，示例语句如下：

```
create table t (a int,b int,index idx(a)) partition by hash(a) partitions 2;
```

此时建完表后会为每个 partition 都单独 split 一个 Region，用 SHOW TABLE REGIONS 语法查看该表的 Region 如下：

```
show table t regions;
```

```
+--
    ↪ ---------+-----------+---------+-----------+-----------------+------------------+------------+------
    ↪
| REGION_ID | START_KEY | END_KEY | LEADER_ID | LEADER_STORE_ID | PEERS            |
    ↪ SCATTERING | WRITTEN_BYTES | READ_BYTES | APPROXIMATE_SIZE(MB) | APPROXIMATE_KEYS |
+--
    ↪ ---------+-----------+---------+-----------+-----------------+------------------+------------+------
    ↪
| 1978      | t_1400_   | t_1401_ | 1979      | 4               | 1979, 1980, 1981 | 0
    ↪          | 0             | 0          | 1                   | 0                |
| 6         | t_1401_   |         | 17        | 4               | 17, 18, 21       | 0
    ↪          | 223           | 0          | 1                   | 0                |
+--
    ↪ ---------+-----------+---------+-----------+-----------------+------------------+------------+------
    ↪
```

2. 用 SPLIT 语法为每个 partition 切分 Region。如果你要将各个 partition 的 [0,10000] 范围内的数据切分成 4 个 Region，示例语句如下：

```
split partition table t between (0) and (10000) regions 4;
```

其中，0 和 10000 分别代表你想要打散的热点数据对应的上、下边界的 row_id。

> 注意：
> 此示例仅适用于数据热点均匀分布的场景。如果热点数据在你指定的数据范围内是不均匀分布的，请参考Split 分区表的 Region 中不均匀切分的语法。

3. 用 SHOW TABLE REGIONS 语法查看该表的 Region。如下会发现该表现在一共有 10 个 Region，每个 partition 分别有 5 个 Region，其中 4 个 Region 是表的行数据，1 个 Region 是表的索引数据。

```
show table t regions;
```

```
+--
    ↪ ---------+--------------+--------------+-----------+----------------+--------------------+---------
    ↪
| REGION_ID | START_KEY     | END_KEY       | LEADER_ID | LEADER_STORE_ID | PEERS
    ↪            | SCATTERING | WRITTEN_BYTES | READ_BYTES | APPROXIMATE_SIZE(MB) |
    ↪ APPROXIMATE_KEYS |
+--
    ↪ ---------+--------------+--------------+-----------+----------------+--------------------+---------
    ↪
| 1998      | t_1400_r      | t_1400_r_2500 | 2001      | 5               | 2000, 2001, 2015
    ↪  | 0          | 132           | 0          | 1                    | 0
    ↪                |
| 2006      | t_1400_r_2500 | t_1400_r_5000 | 2016      | 1               | 2007, 2016, 2017
    ↪  | 0          | 35            | 0          | 1                    | 0
    ↪                |
| 2010      | t_1400_r_5000 | t_1400_r_7500 | 2012      | 2               | 2011, 2012, 2013
    ↪  | 0          | 35            | 0          | 1                    | 0
    ↪                |
| 1978      | t_1400_r_7500 | t_1401_       | 1979      | 4               | 1979, 1980, 1981
    ↪  | 0          | 621           | 0          | 1                    | 0
    ↪                |
| 1982      | t_1400_       | t_1400_r      | 2014      | 3               | 1983, 1984, 2014
    ↪  | 0          | 35            | 0          | 1                    | 0
    ↪                |
| 1990      | t_1401_r      | t_1401_r_2500 | 1992      | 2               | 1991, 1992, 2020
    ↪  | 0          | 120           | 0          | 1                    | 0
    ↪                |
| 1994      | t_1401_r_2500 | t_1401_r_5000 | 1997      | 5               | 1996, 1997, 2021
    ↪  | 0          | 129           | 0          | 1                    | 0
    ↪                |
| 2002      | t_1401_r_5000 | t_1401_r_7500 | 2003      | 4               | 2003, 2023, 2022
    ↪  | 0          | 141           | 0          | 1                    | 0
    ↪                |
| 6         | t_1401_r_7500 |               | 17        | 4               | 17, 18, 21
    ↪       | 0          | 601           | 0          | 1                    | 0
    ↪                |
| 1986      | t_1401_       | t_1401_r      | 1989      | 5               | 1989, 2018, 2019
    ↪  | 0          | 123           | 0          | 1                    | 0
    ↪                |
+--
    ↪ ---------+--------------+--------------+-----------+----------------+--------------------+---------
```

```
    ↪
```

4. 如果你要给每个分区的索引切分 Region，如将索引 idx 的 [1000,10000] 范围切分成 2 个 Region，示例语句如下：

```
split partition table t index idx between (1000) and (10000) regions 2;
```

Split 单个分区的 Region 示例

可以单独指定要切分的 partition，示例如下：

1. 首先创建一个分区表。如果你要建一个 Range 分区表，分成 3 个 partition，示例语句如下：

```
create table t ( a int, b int, index idx(b)) partition by range( a ) (
    partition p1 values less than (10000),
    partition p2 values less than (20000),
    partition p3 values less than (MAXVALUE) );
```

2. 如果你要将 p1 分区的 [0,10000] 范围内的数据预切分 2 个 Region，示例语句如下：

```
split partition table t partition (p1) between (0) and (10000) regions 2;
```

3. 如果你要将 p2 分区的 [10000,20000] 范围内的数据预切分 2 个 Region，示例语句如下：

```
split partition table t partition (p2) between (10000) and (20000) regions 2;
```

4. 用 SHOW TABLE REGIONS 语法查看该表的 Region 如下：

```
show table t regions;
```

```
+--
    ↪ ---------+----------------+----------------+-----------+-----------------+------------------+-------
    ↪
| REGION_ID | START_KEY      | END_KEY        | LEADER_ID | LEADER_STORE_ID | PEERS
    ↪            | SCATTERING | WRITTEN_BYTES | READ_BYTES | APPROXIMATE_SIZE(MB) |
    ↪ APPROXIMATE_KEYS |
+--
    ↪ ---------+----------------+----------------+-----------+-----------------+------------------+-------
    ↪
| 2040      | t_1406_        | t_1406_r_5000  | 2045      | 3               | 2043, 2045,
    ↪ 2044 | 0           | 0              | 0          | 1                 | 0
    ↪                |
| 2032      | t_1406_r_5000  | t_1407_        | 2033      | 4               | 2033, 2034,
    ↪ 2035 | 0           | 0              | 0          | 1                 | 0
    ↪                |
| 2046      | t_1407_        | t_1407_r_15000 | 2048      | 2               | 2047, 2048,
    ↪ 2050 | 0           | 35             | 0          | 1                 | 0
    ↪                |
```

```
| 2036       | t_1407_r_15000 | t_1408_         | 2037       | 4                 | 2037, 2038,
    ↪ 2039 | 0           | 0              | 0           | 1                   | 0
    ↪                 |
| 6          | t_1408_         |                 | 17         | 4                 | 17, 18, 21
    ↪        | 0           | 214            | 0           | 1                   | 0
    ↪                 |
+--
    ↪ ---------+---------------+---------------+-----------+-----------------+-------------------+-------
    ↪
```

5. 如果你要将 p1 和 p2 分区的索引 idx 的 [0,20000] 范围预切分 2 个 Region，示例语句如下：

```
split partition table t partition (p1,p2) index idx between (0) and (20000) regions 2;
```

14.11.2.138.3 pre_split_regions

使用带有 SHARD_ROW_ID_BITS 的表时，如果希望建表时就均匀切分 Region，可以考虑配合 PRE_SPLIT_REGIONS 一起使用，用来在建表成功后就开始预均匀切分 2^(PRE_SPLIT_REGIONS) 个 Region。

> **注意：**
>
> PRE_SPLIT_REGIONS 必须小于等于 SHARD_ROW_ID_BITS。

以下全局变量会影响 PRE_SPLIT_REGIONS 的行为，需要特别注意：

- tidb_scatter_region：该变量用于控制建表完成后是否等待预切分和打散 Region 完成后再返回结果。如果建表后有大批量写入，需要设置该变量值为 1，表示等待所有 Region 都切分和打散完成后再返回结果给客户端。否则未打散完成就进行写入会对写入性能影响有较大的影响。

pre_split_regions 示例

```
create table t (a int, b int,index idx1(a)) shard_row_id_bits = 4 pre_split_regions=2;
```

该语句在建表后，会对这个表 t 预切分出 4 + 1 个 Region。4 (2^2) 个 Region 是用来存 table 的行数据的，1 个 Region 是用来存 idx1 索引的数据。

4 个 table Region 的范围区间如下：

```
region1:   [ -inf      ,  1<<61 )
region2:   [ 1<<61     ,  2<<61 )
region3:   [ 2<<61     ,  3<<61 )
region4:   [ 3<<61     ,  +inf  )
```

14.11.2.138.4 注意事项

Split Region 语句切分的 Region 会受到 PD 中Region merge 调度的控制，需要动态修改 Region merge 相关的配置项，避免新切分的 Region 不久后又被 PD 重新合并的情况。

14.11.2.138.5 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.138.6 另请参阅

- SHOW TABLE REGIONS
- Session 变量：tidb_scatter_region，tidb_wait_split_region_finish 和 tidb_wait_split_region_timeout
  ↪ .

14.11.2.139 START TRANSACTION

START TRANSACTION 语句用于在 TiDB 内部启动新事务。它类似于语句 BEGIN。

在没有 START TRANSACTION 语句的情况下，每个语句都会在各自的事务中自动提交，这样可确保 MySQL 兼容性。

14.11.2.139.1 语法图

BeginTransactionStmt:

```
BeginTransactionStmt ::=
    'BEGIN' ( 'PESSIMISTIC' | 'OPTIMISTIC' )?
|   'START' 'TRANSACTION' ( 'READ' ( 'WRITE' | 'ONLY' ( ( 'WITH' 'TIMESTAMP' 'BOUND'
    ↪ TimestampBound )? | AsOfClause ) ) | 'WITH' 'CONSISTENT' 'SNAPSHOT' | 'WITH' 'CAUSAL' '
    ↪ CONSISTENCY' 'ONLY' )?

AsOfClause ::=
    ( 'AS' 'OF' 'TIMESTAMP' Expression)
```

14.11.2.139.2 示例

```
CREATE TABLE t1 (a int NOT NULL PRIMARY KEY);
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
START TRANSACTION;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
INSERT INTO t1 VALUES (1);
```

```
Query OK, 1 row affected (0.00 sec)
```

```
COMMIT;
```

```
Query OK, 0 rows affected (0.01 sec)
```

14.11.2.139.3 MySQL 兼容性

- 执行 START TRANSACTION 在 TiDB 中开启事务并立即生成快照。而在 MySQL 中，执行 START TRANSACTION 会开启事务但不会立即生成快照。TiDB 中的 START TRANSACTION 等同于 MySQL 中的 START TRANSACTION WITH CONSISTENT SNAPSHOT。
- 为与 MySQL 兼容，TiDB 会解析 START TRANSACTION READ ONLY 语句，但解析后 TiDB 仍允许写入操作。

14.11.2.139.4 另请参阅

- COMMIT
- ROLLBACK
- BEGIN
- START TRANSACTION WITH CAUSAL CONSISTENCY ONLY

14.11.2.140 TABLE

当不需要聚合或复杂的过滤操作时，可以用 TABLE 语句代替 SELECT * FROM。

14.11.2.140.1 语法图

```
TableStmt ::=
    "TABLE" Table ( "ORDER BY" Column )? ( "LIMIT" NUM )?
```

14.11.2.140.2 示例

创建表 t1：

```
CREATE TABLE t1(id INT PRIMARY KEY);
```

插入一些数据：

```
INSERT INTO t1 VALUES (1),(2),(3);
```

查看表 t1 的数据：

```
TABLE t1;
```

```
+----+
| id |
+----+
|  1 |
|  2 |
|  3 |
+----+
3 rows in set (0.01 sec)
```

查询 t1 表，并按 id 字段倒序排列结果：

```
TABLE t1 ORDER BY id DESC;
```

```
+----+
| id |
+----+
|  3 |
|  2 |
|  1 |
+----+
3 rows in set (0.01 sec)
```

查询表 t1 中的前 1 条记录：

```
TABLE t1 LIMIT 1;
```

```
+----+
| id |
+----+
|  1 |
+----+
1 row in set (0.01 sec)
```

14.11.2.140.3 MySQL 兼容性

TABLE 语句是从 MySQL 8.0.19 开始引入的。

14.11.2.140.4 另请参阅

- SELECT
- TABLE statements in MySQL

14.11.2.141 TRACE

TRACE 语句用于提供查询执行的详细信息，可通过 TiDB 服务器状态端口所公开的图形界面进行查看。

14.11.2.141.1 语法图

TraceStmt:

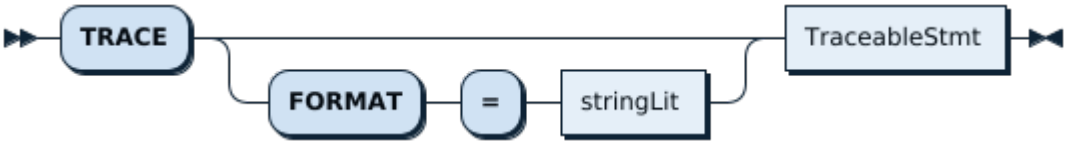


图 470: TraceStmt

TraceableStmt:

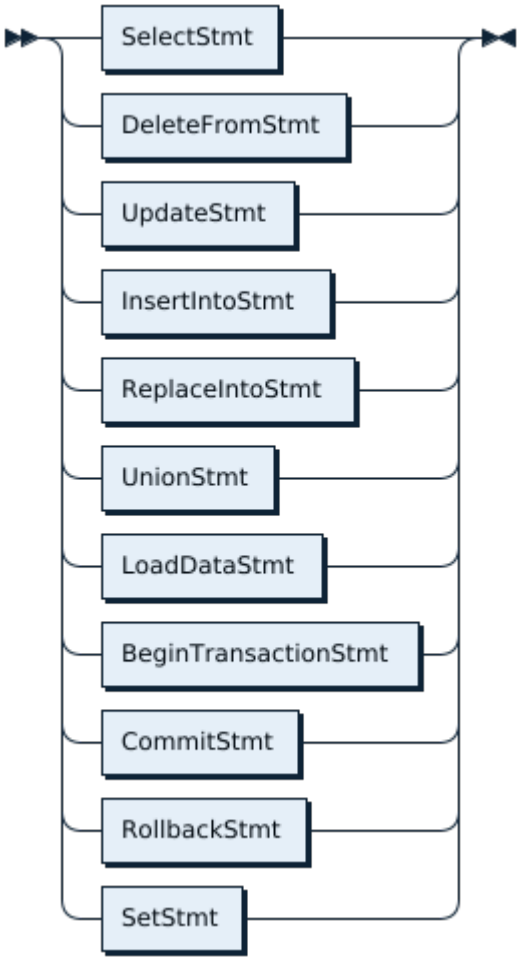


图 471: TraceableStmt

14.11.2.141.2 示例

```
trace format='row' select * from mysql.user;
```

```
+-----------------------------------------+-----------------+------------+
| operation                               | startTS         | duration   |
```

```
+-----------------------------------------+-----------------+------------+
| trace                                   | 17:03:31.938237 | 886.086µs  |
|   ├─session.Execute                     | 17:03:31.938247 | 507.812µs  |
|   │ ├─session.ParseSQL                  | 17:03:31.938254 | 22.504µs   |
|   │ ├─executor.Compile                  | 17:03:31.938321 | 278.931µs  |
|   │ │ └─session.getTxnFuture            | 17:03:31.938337 | 1.515µs    |
|   │ └─session.runStmt                   | 17:03:31.938613 | 109.578µs  |
|   │   ├─TableReaderExecutor.Open        | 17:03:31.938645 | 50.657µs   |
|   │   │ └─distsql.Select                | 17:03:31.938666 | 21.066µs   |
|   │   │   └─RPCClient.SendRequest       | 17:03:31.938799 | 158.411µs  |
|   │   └─session.CommitTxn               | 17:03:31.938705 | 12.06µs    |
|   │     └─session.doCommitWitRetry      | 17:03:31.938709 | 2.437µs    |
|   ├─*executor.TableReaderExecutor.Next  | 17:03:31.938781 | 224.327µs  |
|   └─*executor.TableReaderExecutor.Next  | 17:03:31.939019 | 6.266µs    |
+-----------------------------------------+-----------------+------------+
13 rows in set (0.00 sec)
```

```
trace format='json' select * from mysql.user;
```

可将 JSON 格式的跟踪文件粘贴到跟踪查看器中。查看器可通过 TiDB 状态端口访问：

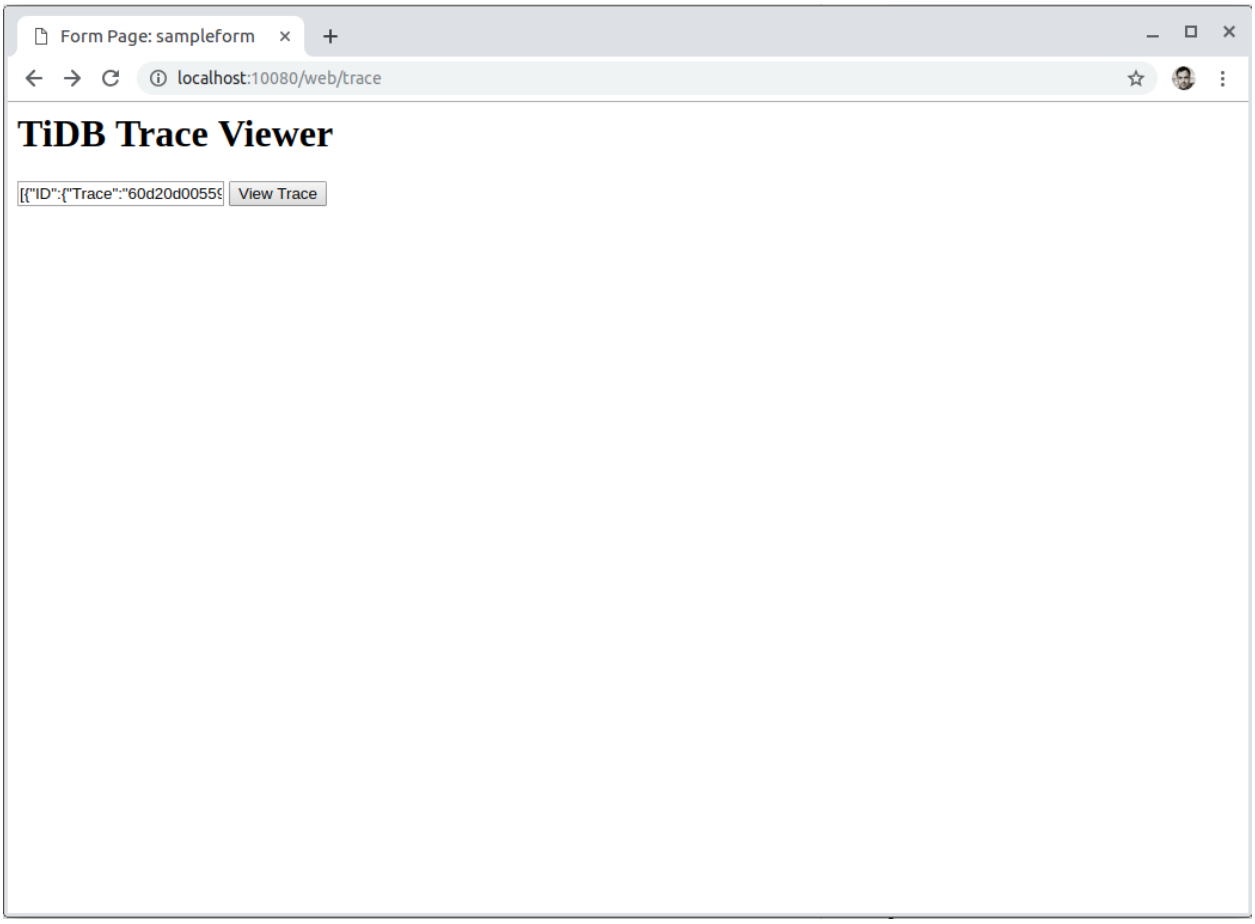


图 472: TiDB Trace Viewer-1

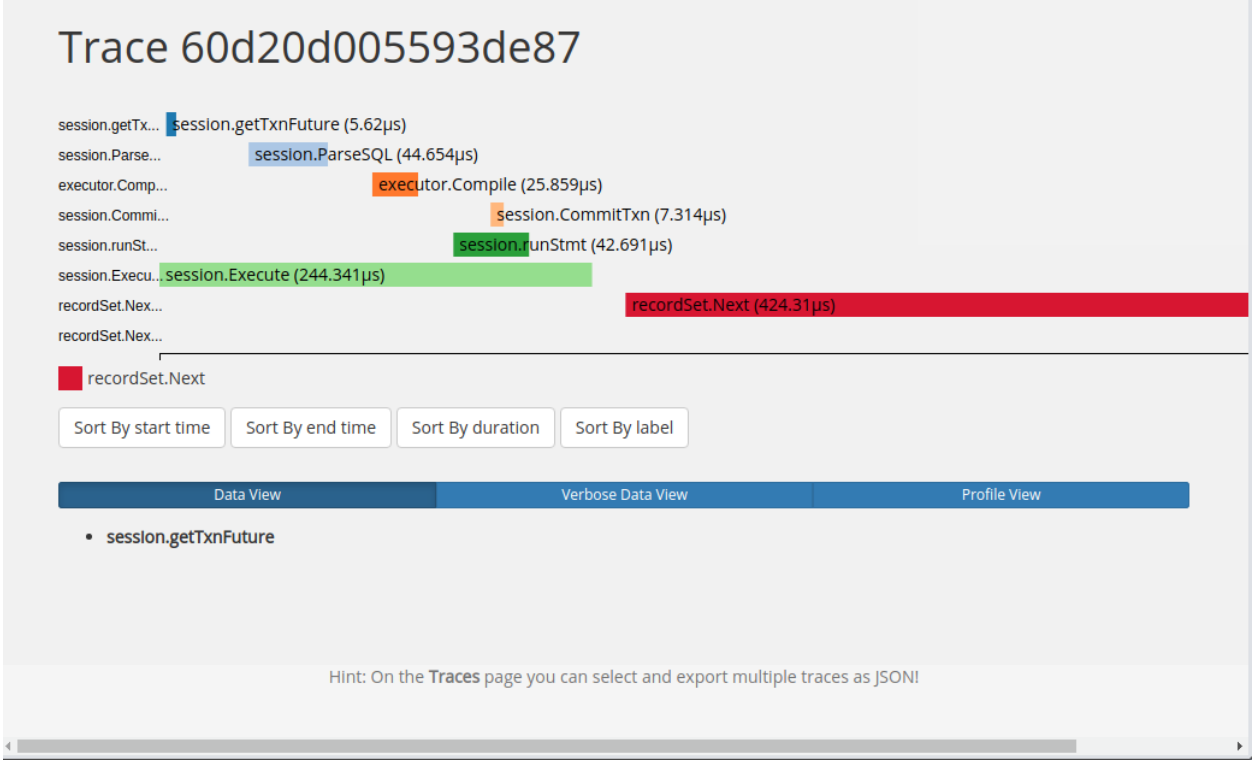


图 473: TiDB Trace Viewer-2

14.11.2.141.3 MySQL 兼容性

TRACE 语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.141.4 另请参阅

- EXPLAIN ANALYZE

14.11.2.142 TRUNCATE

TRUNCATE 语句以非事务方式从表中删除所有数据。可认为 TRUNCATE 语句同 DROP TABLE + CREATE TABLE 组合在语义上相同，定义与 DROP TABLE 语句相同。

TRUNCATE TABLE tableName 和 TRUNCATE tableName 均为有效语法。

14.11.2.142.1 语法图

TruncateTableStmt:



图 474: TruncateTableStmt

OptTable:



图 475: OptTable

TableName:

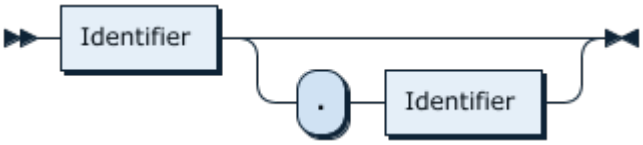


图 476: TableName

14.11.2.142.2 示例

```
CREATE TABLE t1 (a INT NOT NULL PRIMARY KEY);
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
INSERT INTO t1 VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 5 rows affected (0.01 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
SELECT * FROM t1;
```

```
+---+
| a |
+---+
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
+---+
5 rows in set (0.00 sec)
```

```
TRUNCATE t1;
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
SELECT * FROM t1;
```

```
Empty set (0.00 sec)
```

```
INSERT INTO t1 VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 5 rows affected (0.01 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
TRUNCATE TABLE t1;
```

```
Query OK, 0 rows affected (0.11 sec)
```

14.11.2.142.3 MySQL 兼容性

TRUNCATE 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.142.4 另请参阅

- DROP TABLE
- DELETE
- CREATE TABLE
- SHOW CREATE TABLE

14.11.2.143 UNLOCK STATS

UNLOCK STATS 语句用于解锁统计信息被锁定的表。

> **警告：**
>
> 锁定统计信息目前为实验特性，不建议在生产环境中使用。

14.11.2.143.1 语法图

```
UnlockStatsStmt ::=
    'UNLOCK' 'STATS' TableNameList

TableNameList ::=
    TableName (',' TableName)*

TableName ::=
    Identifier ( '.' Identifier )?
```

14.11.2.143.2 示例

创建表 t，插入一些数据，在未锁定表 t 的统计信息的情况下成功执行 ANALYZE 语句。

```
mysql> create table t(a int, b int);
Query OK, 0 rows affected (0.03 sec)

mysql> insert into t values (1,2), (3,4), (5,6), (7,8);
Query OK, 4 rows affected (0.00 sec)
Records: 4  Duplicates: 0  Warnings: 0

mysql> analyze table t;
Query OK, 0 rows affected, 1 warning (0.02 sec)

mysql> show warnings;
+-------+------+-----------------------------------------------------------------+
| Level | Code | Message                                                         |
+-------+------+-----------------------------------------------------------------+
| Note  | 1105 | Analyze use auto adjusted sample rate 1.000000 for table test.t |
+-------+------+-----------------------------------------------------------------+
1 row in set (0.00 sec)
```

锁定表 t 的统计信息，执行 ANALYZE 语句，warning 提示跳过对表 t 的 ANALYZE。

```
mysql> lock stats t;
Query OK, 0 rows affected (0.00 sec)

mysql> show stats_locked;
+---------+------------+----------------+--------+
| Db_name | Table_name | Partition_name | Status |
+---------+------------+----------------+--------+
| test    | t          |                | locked |
+---------+------------+----------------+--------+
1 row in set (0.01 sec)

mysql> analyze table t;
Query OK, 0 rows affected, 2 warnings (0.00 sec)

mysql> show warnings;
+---------+------+-----------------------------------------------------------------+
| Level   | Code | Message                                                         |
+---------+------+-----------------------------------------------------------------+
| Note    | 1105 | Analyze use auto adjusted sample rate 1.000000 for table test.t |
| Warning | 1105 | skip analyze locked table: t                                    |
+---------+------+-----------------------------------------------------------------+
2 rows in set (0.00 sec)
```

解锁表 t 的统计信息，成功执行 ANALYZE 语句。

```
mysql> unlock stats t;
Query OK, 0 rows affected (0.01 sec)

mysql> analyze table t;
Query OK, 0 rows affected, 1 warning (0.03 sec)

mysql> show warnings;
+-------+------+-----------------------------------------------------------------+
| Level | Code | Message                                                         |
+-------+------+-----------------------------------------------------------------+
| Note  | 1105 | Analyze use auto adjusted sample rate 1.000000 for table test.t |
+-------+------+-----------------------------------------------------------------+
1 row in set (0.00 sec)
```

14.11.2.143.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

14.11.2.143.4 另请参阅

- 统计信息
- LOCK STATS
- SHOW STATS_LOCKED

14.11.2.144 UPDATE

UPDATE 语句用于修改指定表中的数据。

14.11.2.144.1 语法图

UpdateStmt:

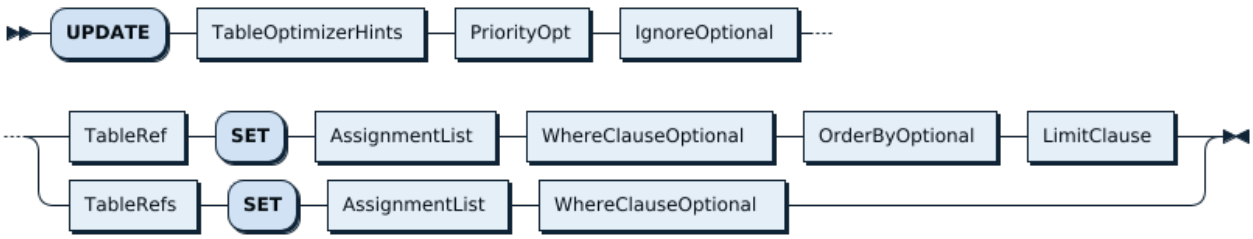


图 477: UpdateStmt

PriorityOpt:

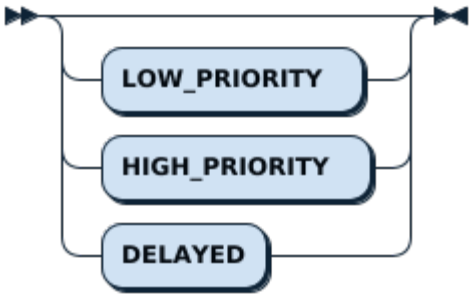


图 478: PriorityOpt

TableRef:

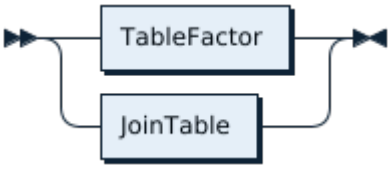


图 479: TableRef

TableRefs:



图 480: TableRefs

AssignmentList:

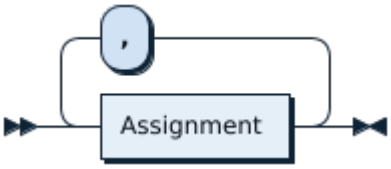


图 481: AssignmentList

WhereClauseOptional:

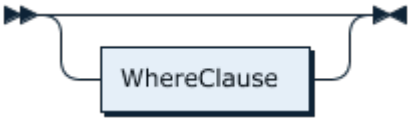


图 482: WhereClauseOptional

14.11.2.144.2 示例

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, c1 INT NOT NULL);
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
INSERT INTO t1 (c1) VALUES (1), (2), (3);
```

```
Query OK, 3 rows affected (0.02 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
SELECT * FROM t1;
```

```
+----+----+
| id | c1 |
+----+----+
|  1 |  1 |
|  2 |  2 |
|  3 |  3 |
+----+----+
3 rows in set (0.00 sec)
```

```
UPDATE t1 SET c1=5 WHERE c1=3;
```

```
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

```
SELECT * FROM t1;
```

```
+----+----+
| id | c1 |
+----+----+
|  1 |  1 |
|  2 |  2 |
|  3 |  5 |
+----+----+
3 rows in set (0.00 sec)
```

14.11.2.144.3 MySQL 兼容性

在计算表达式中的列时，TiDB 总使用原始的值。例如：

```
CREATE TABLE t (a int, b int);
INSERT INTO t VALUES (1,2);
UPDATE t SET a = a+1,b=a;
```

在 MySQL 中，b 列的值会被更新成 2，因为 b 列被设置为与 a 列相同，而 a（最初是 1）在同一条语句中被更新成了 2。

TiDB 遵守标准的 SQL 行为，这里将 b 列值更新成 1。

14.11.2.144.4 另请参阅

- INSERT
- SELECT
- DELETE
- REPLACE

14.11.2.145 USE

USE 语句可为用户会话选择当前数据库。

14.11.2.145.1 语法图

UseStmt:



图 483: UseStmt

DBName:



图 484: DBName

14.11.2.145.2 示例

```
USE mysql;
```

```
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
```

```
SHOW TABLES;
```

```
+-------------------------+
| Tables_in_mysql         |
+-------------------------+
| GLOBAL_VARIABLES        |
```

```
| bind_info                |
| columns_priv             |
| db                       |
| default_roles            |
| expr_pushdown_blacklist |
| gc_delete_range          |
| gc_delete_range_done     |
| global_priv              |
| help_topic               |
| opt_rule_blacklist       |
| role_edges               |
| stats_buckets            |
| stats_feedback           |
| stats_histograms         |
| stats_meta               |
| stats_top_n              |
| tables_priv              |
| tidb                     |
| user                     |
+--------------------------+
20 rows in set (0.01 sec)
```

```
CREATE DATABASE newtest;
```

```
Query OK, 0 rows affected (0.10 sec)
```

```
USE newtest;
```

```
Database changed
```

```
SHOW TABLES;
```

```
Empty set (0.00 sec)
```

```
CREATE TABLE t1 (a int);
```

```
Query OK, 0 rows affected (0.10 sec)
```

```
SHOW TABLES;
```

```
+-------------------+
| Tables_in_newtest |
+-------------------+
| t1                |
+-------------------+
1 row in set (0.00 sec)
```

14.11.2.145.3 MySQL 兼容性

USE 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

14.11.2.145.4 另请参阅

- CREATE DATABASE
- SHOW TABLES

14.11.2.146 WITH

公共表表达式 (CTE) 是一个临时的中间结果集，能够在 SQL 语句中引用多次，提高 SQL 语句的可读性与执行效率。在 TiDB 中可以通过 WITH 语句使用公共表表达式。

14.11.2.146.1 语法图

WithClause:

```
WithClause ::=
        "WITH" WithList
|       "WITH" recursive WithList
```

WithList:

```
WithList ::=
        WithList ',' CommonTableExpr
|       CommonTableExpr
```

CommonTableExpr:

```
CommonTableExpr ::=
        Identifier IdentListWithParenOpt "AS" SubSelect
```

IdentListWithParenOpt:

```
IdentListWithParenOpt ::=
( '(' IdentList ')' )?
```

14.11.2.146.2 示例

非递归的 CTE：

```
WITH CTE AS (SELECT 1, 2) SELECT * FROM cte t1, cte t2;
```

```
+---+---+---+---+
| 1 | 2 | 1 | 2 |
+---+---+---+---+
| 1 | 2 | 1 | 2 |
```

```
+---+---+---+---+
1 row in set (0.00 sec)
```

递归的 CTE：

```
WITH RECURSIVE cte(a) AS (SELECT 1 UNION SELECT a+1 FROM cte WHERE a < 5) SELECT * FROM cte;
```

```
+---+
| a |
+---+
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
+---+
5 rows in set (0.00 sec)
```

14.11.2.146.3 MySQL 兼容性

- 在严格模式下，当递归部分算出的数据长度超过初始部分的数据长度时，TiDB 会返回警告，而 MySQL 会返回错误。在非严格模式下，TiDB 与 MySQL 行为一致。
- 递归 CTE 所使用的数据类型由初始部分决定。初始部分的数据类型在某些情况（例如函数）下与 MySQL 并不完全一致。
- 多个 UNION / UNION ALL 情况下，MySQL 不允许 UNION 后面加 UNION ALL，TiDB 允许。
- 如果 CTE 的定义存在问题，TiDB 会报错，而 MySQL 在未引用的情况下不报错。

14.11.2.146.4 另请参阅

- SELECT
- INSERT
- DELETE
- UPDATE
- REPLACE

14.11.3 数据类型

14.11.3.1 数据类型概述

TiDB 支持除空间类型 (SPATIAL) 之外的所有 MySQL 数据类型，包括数值型类型、字符串类型、时间和日期类型、JSON 类型。

数据类型定义一般为 `T(M[, D])`，其中：

- T 表示具体的类型。

- M 在整数类型中表示最大显示长度；在浮点数或者定点数中表示精度；在字符类型中表示最大长度。M 的最大值取决于具体的类型。
- D 表示浮点数、定点数的小数位长度。
- fsp 在时间和日期类型里的 TIME、DATETIME 以及 TIMESTAMP 中表示秒的精度，其取值范围是 0 到 6。值为 0 表示没有小数部分。如果省略，则默认精度为 0。

14.11.3.2 数据类型的默认值

在一个数据类型描述中的 DEFAULT value 段描述了一个列的默认值。这个默认值必须是常量，不可以是一个函数或者是表达式。但是对于时间类型，可以例外的使用 NOW、CURRENT_TIMESTAMP、LOCALTIME、LOCALTIMESTAMP 等函数作为 DATETIME 或者 TIMESTAMP 的默认值。

BLOB、TEXT 以及 JSON 不可以设置默认值。

如果一个列的定义中没有 DEFAULT 的设置。TiDB 按照如下的规则决定：

- 如果该类型可以使用 NULL 作为值，那么这个列会在定义时添加隐式的默认值设置 DEFAULT NULL。
- 如果该类型无法使用 NULL 作为值，那么这个列在定义时不会添加隐式的默认值设置。

对于一个设置了 NOT NULL 但是没有显式设置 DEFAULT 的列，当 INSERT、REPLACE 没有涉及到该列的值时，TiDB 根据当时的 SQL_MODE 进行不同的行为：

- 如果此时是 strict sql mode，在事务中的语句会导致事务失败并回滚，非事务中的语句会直接报错。
- 如果此时不是 strict sql mode，TiDB 会为这列赋值为列数据类型的隐式默认值。

此时隐式默认值的设置按照如下规则：

- 对于数值类型，它们的默认值是 0。当有 AUTO_INCREMENT 参数时，默认值会按照增量情况赋予正确的值。
- 对于除了时间戳外的日期时间类型，默认值会是该类型的“零值”。时间戳类型的默认值会是当前的时间。
- 对于除枚举以外的字符串类型，默认值会是空字符串。对于枚举类型，默认值是枚举中的第一个值。

14.11.3.3 数值类型

TiDB 支持 MySQL 所有的数值类型，按照精度可以分为：

- 整数类型（精确值）
- 浮点类型（近似值）
- 定点类型（精确值）

14.11.3.3.1 整数类型

TiDB 支持 MySQL 所有的整数类型，包括 INTEGER/INT、TINYINT、SMALLINT、MEDIUMINT 以及 BIGINT，完整信息参考这篇文档。

字段说明：

语法元素	说明
M	类型显示宽度，可选
UNSIGNED	无符号数，如果不加这个标识，则为有符号数
ZEROFILL	补零标识，如果有这个标识，TiDB 会自动给类型增加 UNSIGNED 标识，但是没有做补零的操作

类型定义

BIT 类型

比特值类型。M 表示比特位的长度，取值范围从 1 到 64，其默认值是 1。

```
BIT[(M)]
```

BOOLEAN 类型

布尔类型，别名为 BOOL，和 TINYINT(1) 等价。零值被认为是 False，非零值认为是 True。在 TiDB 内部，True 存储为 1，False 存储为 0。

```
BOOLEAN
```

TINYINT 类型

TINYINT 类型。有符号数的范围是 [-128，127]。无符号数的范围是 [0，255]。

```
TINYINT[(M)] [UNSIGNED] [ZEROFILL]
```

SMALLINT 类型

SMALLINT 类型。有符号数的范围是 [-32768，32767]。无符号数的范围是 [0，65535]。

```
SMALLINT[(M)] [UNSIGNED] [ZEROFILL]
```

MEDIUMINT 类型

MEDIUMINT 类型。有符号数的范围是 [-8388608，8388607]。无符号数的范围是 [0，16777215]。

```
MEDIUMINT[(M)] [UNSIGNED] [ZEROFILL]
```

INTEGER 类型

INTEGER 类型，别名 INT。有符号数的范围是 [-2147483648，2147483647]。无符号数的范围是 [0，4294967295]。

```
INT[(M)] [UNSIGNED] [ZEROFILL]
```

或者：

```
INTEGER[(M)] [UNSIGNED] [ZEROFILL]
```

BIGINT 类型

BIGINT 类型。有符号数的范围是 [-9223372036854775808，9223372036854775807]。无符号数的范围是 [0，
↪ 18446744073709551615]。

```
BIGINT[(M)] [UNSIGNED] [ZEROFILL]
```

存储空间以及取值范围

每种类型对存储空间的需求以及最大/最小值如下表所示：

类型	存储空间	最小值 (有符号/无符号)	最大值 (有符号/无符号)
TINYINT	1	-128 / 0	127 / 255
SMALLINT	2	-32768 / 0	32767 / 65535
MEDIUMINT	3	-8388608 / 0	8388607 / 16777215
INT	4	-2147483648 / 0	2147483647 / 4294967295
BIGINT	8	-9223372036854775808 / 0	9223372036854775807 / 18446744073709551615

14.11.3.3.2 浮点类型

TiDB 支持 MySQL 所有的浮点类型，包括 FLOAT、DOUBLE，完整信息参考这篇文档。

字段说明：

语法元素	说明
M	小数总位数
D	小数点后位数
UNSIGNED	无符号数，如果不加这个标识，则为有符号数
ZEROFILL	补零标识，如果有这个标识，TiDB 会自动给类型增加 UNSIGNED 标识

类型定义

FLOAT 类型

单精度浮点数。允许的值范围为 -2^128 ~ +2^128，也即 -3.402823466E+38 到 -1.175494351E-38、0 和 1.175494351E-38 到 3.402823466E+38。这些是基于 IEEE 标准的理论限制。实际的范围根据硬件或操作系统的不同可能稍微小些。

FLOAT(p) 类型中，p 表示精度（以位数表示）。只使用该值来确定是否结果列的数据类型为 FLOAT 或 DOUBLE。如果 p 为从 0 到 24，数据类型变为没有 M 或 D 值的 FLOAT。如果 p 为从 25 到 53，数据类型变为没有 M 或 D 值的 DOUBLE。结果列范围与本节前面描述的单精度 FLOAT 或双精度 DOUBLE 数据类型相同。

```
FLOAT[(M,D)] [UNSIGNED] [ZEROFILL]
FLOAT(p) [UNSIGNED] [ZEROFILL]
```

注意：

- 与在 MySQL 中一样，FLOAT 数据类型用于存储近似值。对于类似货币的精确值，建议使用 DECIMAL 类型。

- 在 TiDB 中，FLOAT 数据类型默认的精度是 8 位，这与 MySQL 不同。在 MySQL 中，FLOAT 数据类型默认的精度是 6 位。例如，同时往 MySQL 和 TiDB 中类型为 FLOAT 的列中插入数据 123456789 和 1.23456789。在 MySQL 中查询对应的值，将会得到结果 123457000 和 1.23457。而在 TiDB 中查询对应的值，将会得到 123456790 和 1.2345679。

DOUBLE 类型

双精度浮点数，别名为 DOUBLE PRECISION。允许的值范围为：-2^1024 ~ +2^1024，也即是 -1.7976931348623157E+308 到 -2.2250738585072014E-308、0 和 2.2250738585072014E-308 到 1.7976931348623157E+308。这些是基于 IEEE 标准的理论限制。实际的范围根据硬件或操作系统的不同可能稍微小些。

```
DOUBLE[(M,D)] [UNSIGNED] [ZEROFILL]
DOUBLE PRECISION [(M,D)] [UNSIGNED] [ZEROFILL], REAL[(M,D)] [UNSIGNED] [ZEROFILL]
```

警告：

与在 MySQL 中一样，DOUBLE 数据类型存储近似值。对于货币之类的精确值，建议使用 DECIMAL 类型。

注意：

当 TiDB 将用科学计数法表示的双精度浮点数转换到 CHAR 类型时，其结果在显示上与 MySQL 不一致，详情参见Cast 函数和操作符。

存储空间

每种类型对存储空间的需求如下表所示：

类型	存储空间
FLOAT	4
FLOAT(p)	如果 0 <= p <= 24 为 4 个字节，如果 25 <= p <= 53 为 8 个字节
DOUBLE	8

14.11.3.3.3 定点类型

TiDB 支持 MySQL 所有的定点类型，包括 DECIMAL、NUMERIC，完整信息参考这篇文档。

字段说明：

语法元素	说明
M	小数总位数
D	小数点后位数
UNSIGNED	无符号数，如果不加这个标识，则为有符号数
ZEROFILL	补零标识，如果有这个标识，TiDB 会自动给类型增加 UNSIGNED 标识

类型定义

DECIMAL 类型

定点数，别名为 NUMERIC。M 是小数位数（精度）的总数，D 是小数点（标度）后面的位数。小数点和 -（负数）符号不包括在 M 中。如果 D 是 0，则值没有小数点或分数部分。如果 D 被省略，默认是 0。如果 M 被省略，默认是 10。

```
DECIMAL[(M[,D])] [UNSIGNED] [ZEROFILL]
NUMERIC[(M[,D])] [UNSIGNED] [ZEROFILL]
```

14.11.3.4 日期和时间类型

TiDB 支持 MySQL 所有的日期和时间类型，包括DATE、TIME、DATETIME、TIMESTAMP 以及YEAR。完整信息可以参考 MySQL 中的时间和日期类型。

每种类型都有有效值范围，值为 0 表示无效值。此外，TIMESTAMP 和 DATETIME 类型能自动生成新的时间值。

关于日期和时间值类型，需要注意：

- 日期部分必须是“年 -月 -日”的格式（例如 1998-09-04），而不是“月 -日 -年”或“日 -月 -年”的格式。
- 如果日期的年份部分是 2 位数，TiDB 会根据年份为两位数的具体规则进行转换。
- 如果格式必须是数值类型，TiDB 会自动将日期或时间值转换为数值类型。例如：

```
SELECT NOW(), NOW()+0, NOW(3)+0;
```

```
+---------------------+----------------+--------------------+
| NOW()               | NOW()+0        | NOW(3)+0           |
+---------------------+----------------+--------------------+
| 2012-08-15 09:28:00 | 20120815092800 | 20120815092800.889 |
+---------------------+----------------+--------------------+
```

- TiDB 可以自动将无效值转换同一类型的零值。是否进行转换取决于 SQL 模式的设置。比如：

```
show create table t1;
```

```
+--
    ↪ -----+-------------------------------------------------------------------------------------------------
    ↪
```

```
| Table | Create Table
    ↪
    ↪ |
+--
    ↪ -----+----------------------------------------------------------------------------------------------------------
    ↪
| t1    | CREATE TABLE `t1` (
  `a` time DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin |
+--
    ↪ -----+----------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

```
select @@sql_mode;
```

```
+--
    ↪ ----------------------------------------------------------------------------------------------------------
    ↪
| @@sql_mode
    ↪
    ↪ |
+--
    ↪ ----------------------------------------------------------------------------------------------------------
    ↪
| ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,
    ↪ ERROR_FOR_DIVISION_BY_ZERO,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION |
+--
    ↪ ----------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

```
insert into t1 values (`2090-11-32:22:33:44`);
```

```
ERROR 1292 (22007): Truncated incorrect time value: `2090-11-32:22:33:44`
```

```
set @@sql_mode=``;
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
insert into t1 values (`2090-11-32:22:33:44`);
```

```
Query OK, 1 row affected, 1 warning (0.01 sec)
```

```
select * from t1;
```

```
+----------+
| a        |
+----------+
| 00:00:00 |
+----------+
1 row in set (0.01 sec)
```

- SQL 模式的不同设置，会改变 TiDB 对格式的要求。
- 如果 SQL 模式的 NO_ZERO_DATE 被禁用，TiDB 允许 DATE 和 DATETIME 列中的月份或日期为零。例如，2009-00-00 或 2009-01-00。如果使用函数计算这种日期类型，例如使用 DATE_SUB() 或 DATE_ADD() 函数，计算结果可能不正确。
- 默认情况下，TiDB 启用 NO_ZERO_DATE SQL 模式。该模式可以避免存储像 0000-00-00 这样的零值。

不同类型的零值如下表所示：

数据类型	零值
DATE	0000-00-00
TIME	00:00:00
DATETIME	0000-00-00 00:00:00
TIMESTAMP	0000-00-00 00:00:00
YEAR	0000

如果 SQL 模式允许使用无效的 DATE、DATETIME、TIMESTAMP 值，无效值会自动转换为相应的零值（0000-00-00 或 0000-00-00 00:00:00）。

14.11.3.4.1 类型定义

DATE 类型

DATE 类型只包含日期部分，不包含时间部分。DATE 类型的格式为 YYYY-MM-DD，支持的范围是 0000-01-01 到 9999-12-31。

```
DATE
```

TIME 类型

TIME 类型的格式为 HH:MM:SS[.fraction]，支持的范围是 -838:59:59.000000 到 838:59:59.000000。TIME 不仅可用于指示一天内的时间，还可用于指两个事件之间的时间间隔。fsp 参数表示秒精度，取值范围为：0 ~ 6，默认值为 0。

```
TIME[(fsp)]
```

> **注意：**
>
> 注意 TIME 的缩写形式。例如，11:12 表示 11:12:00 而不是 00:11:12。但是，1112 表示 00:11:12。这些差异取决于：字符的存在与否。

DATETIME 类型

DATETIME 类型是日期和时间的组合，格式为 YYYY-MM-DD HH:MM:SS[.fraction]。支持的范围是 0000-01-01 00:00:00.000000 到 9999-12-31 23:59:59.999999。fsp 参数表示秒精度，取值范围为 0~6，默认值为 0。TiDB 支持字符串或数字转换为 DATETIME 类型。

```
DATETIME[(fsp)]
```

TIMESTAMP 类型

TIMESTAMP 类型是日期和时间的组合，支持的范围是 UTC 时间从 1970-01-01 00:00:01.000000 到 2038-01-19 03:14:07.999999。fsp 参数表示秒精度，取值范围为 0~6，默认值为 0。在 TIMESTAMP 中，不允许零出现在月份部分或日期部分，唯一的例外是零值本身 0000-00-00 00:00:00。

```
TIMESTAMP[(fsp)]
```

时区处理

当存储 TIMESTAMP 时，TiDB 会将当前时区的 TIMESTAMP 值转换为 UTC 时区。当读取 TIMESTAMP 时，TiDB 将存储的 TIMESTAMP 值从 UTC 时区转换为当前时区（注意：DATETIME 不会这样处理）。每次连接的默认时区是服务器的本地时区，可以通过环境变量 time_zone 进行修改。

> **警告：**
>
> 和 MySQL 一样，TIMESTAMP 数据类型受 2038 年问题的影响。如果存储的值大于 2038，建议使用 DATETIME 类型。

YEAR 类型

YEAR 类型的格式为 YYYY，支持的值范围是 1901 到 2155，也支持零值 0000。

```
YEAR[(4)]
```

YEAR 类型遵循以下格式规则：

- 如果是四位数的数值，支持的范围是 1901 至 2155。
- 如果是四位数的字符串，支持的范围是 '1901' 到 '2155'。
- 如果是 1~99 之间的一位数或两位数的数字，1~69 换算成 2001_{2069}，$_{70}99$ 换算成 1970~1999。
- 支持 '0' 到 '99' 之间的一位数或两位数字符串的范围
- 将数值 0 转换为 0000，将字符串 '0' 或 '00' 转换为 '2000'。

无效的 YEAR 值会自动转换为 0000（如果用户没有使用 NO_ZERO_DATE SQL 模式）。

14.11.3.4.2 自动初始化和更新 TIMESTAMP 或 DATETIME

带有 TIMESTAMP 或 DATETIME 数据类型的列可以自动初始化为或更新为当前时间。

对于表中任何带有 TIMESTAMP 或 DATETIME 数据类型的列，你可以设置默认值，或自动更新为当前时间戳。

在定义列的时候，TIMESTAMP 和 DATETIME 可以通过 DEFAULT CURRENT_TIMESTAMP 和 ON UPDATE CURRENT_TIMESTAMP 来设置。DEFAULT 也可以设置为一个特定的值，例如 DEFAULT 0 或 DEFAULT '2000-01-01 00:00:00'。

```
CREATE TABLE t1 (
    ts TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    dt DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);
```

除非指定 DATETIME 的值为 NOT NULL，否则默认 DATETIME 的值为 NULL。指定 DATETIME 的值为 NOT NULL 时，如果没有设置默认值，则默认值为 0。

```
CREATE TABLE t1 (
    dt1 DATETIME ON UPDATE CURRENT_TIMESTAMP,          -- default NULL
    dt2 DATETIME NOT NULL ON UPDATE CURRENT_TIMESTAMP -- default 0
);
```

14.11.3.4.3 时间值的小数部分

DATETIME 和 TIMESTAMP 值最多可以有 6 位小数，精确到微秒。如果包含小数部分，值的格式为 YYYY-MM-DD HH:MM:SS[.fraction]，小数部分的范围为 000000 到999999。必须使用小数点分隔小数部分与其他部分。

- 使用 type_name(fsp) 可以定义精确到小数的列，其中 type_name 可以是TIME、DATETIME 或 TIMESTAMP。例如：

  ```
  CREATE TABLE t1 (t TIME(3), dt DATETIME(6));
  ```

  fsp 范围是 0 到 6。

  0 表示没有小数部分。如果省略了 fsp，默认为 0。

- 当插入包含小数部分的 TIME、DATETIME 或 TIMESTAMP 时，如果小数部分的位数过少或过多，可能需要进行四舍五入。例如：

  ```
  CREATE TABLE fractest( c1 TIME(2), c2 DATETIME(2), c3 TIMESTAMP(2) );
  ```

  ```
  Query OK, 0 rows affected (0.33 sec)
  ```

  ```
  INSERT INTO fractest VALUES
       > ('17:51:04.777', '2014-09-08 17:51:04.777',   '2014-09-08 17:51:04.777');
  ```

  ```
  Query OK, 1 row affected (0.03 sec)
  ```

```
SELECT * FROM fractest;
```

```
+-------------|-------------------------|-------------------------+
| c1          | c2                      | c3                      |
+-------------|-------------------------|-------------------------+
| 17:51:04.78 | 2014-09-08 17:51:04.78 | 2014-09-08 17:51:04.78 |
+-------------|-------------------------|-------------------------+
1 row in set (0.00 sec)
```

14.11.3.4.4 日期和时间类型的转换

在日期和时间类型之间进行转换时，有些转换可能会导致信息丢失。例如，DATE、DATETIME 和 TIMESTAMP 都有各自的有效值范围。TIMESTAMP 不能早于 UTC 时间的 1970 年，也不能晚于 UTC 时间的 2038-01-19 03:14:07。根据这个规则，1968-01-01 对于 DATE 或 DATETIME 是有效的，但当 1968-01-01 转换为 TIMESTAMP 时，就会变成 0。

DATE 的转换：

- 当 DATE 转换为 DATETIME 或 TIMESTAMP 时，会添加时间部分 00:00:00，因为 DATE 不包含任何时间信息。
- 当 DATE 转换为 TIME 时，结果是 00:00:00。

DATETIME 或 TIMESTAMP 的转换：

- 当 DATETIME 或 TIMESTAMP 转换为 DATE 时，时间和小数部分将被舍弃。例如，1999-12-31 23:59:59.499 被转换为 1999-12-31。
- 当 DATETIME 或 TIMESTAMP 转换为 TIME 时，日期部分被舍弃，因为 TIME 不包含任何日期信息。

如果要将 TIME 转换为其他时间和日期格式，日期部分会自动指定为 CURRENT_DATE()。最终的转换结果是由 TIME 和 CURRENT_DATE() 组成的日期。也就是说，如果 TIME 的值超出了 00:00:00 到 23:59:59 的范围，那么转换后的日期部分并不表示当前的日期。

当 TIME 转换为 DATE 时，转换过程类似，时间部分被舍弃。

使用 CAST() 函数可以显式地将值转换为 DATE 类型。例如：

```
date_col = CAST(datetime_col AS DATE)
```

将 TIME 和 DATETIME 转换为数字格式。例如：

```
SELECT CURTIME(), CURTIME()+0, CURTIME(3)+0;
```

```
+-----------|-------------|--------------+
| CURTIME() | CURTIME()+0 | CURTIME(3)+0 |
+-----------|-------------|--------------+
| 09:28:00  |       92800 |    92800.887 |
+-----------|-------------|--------------+
```

```
SELECT NOW(), NOW()+0, NOW(3)+0;
```

```
+---------------------|----------------|--------------------+
| NOW()               | NOW()+0        | NOW(3)+0           |
+---------------------|----------------|--------------------+
| 2012-08-15 09:28:00 | 20120815092800 | 20120815092800.889 |
+---------------------|----------------|--------------------+
```

14.11.3.4.5 年份为两位数

如果日期中包含年份为两位数，这个年份是有歧义的，并不显式地表示实际年份。

对于 DATETIME、DATE 和 TIMESTAMP 类型，TiDB 使用如下规则来消除歧义。

- 将 01 至 69 之间的值转换为 2001 至 2069 之间的值。
- 将 70 至 99 之间的值转化为 1970 至 1999 之间的值。

上述规则也适用于 YEAR 类型，但有一个例外。将数字 00 插入到 YEAR(4) 中时，结果是 0000 而不是 2000。

如果想让结果是 2000，需要指定值为 2000。

对于 MIN() 和 MAX() 等函数，年份为两位数时可能会得到错误的计算结果。建议年份为四位数时使用这类函数。

14.11.3.5 字符串类型

TiDB 支持 MySQL 所有的字符串类型，包括 CHAR、VARCHAR、BINARY、VARBINARY、BLOB、TEXT、ENUM 以及 SET，完整信息参考这篇文档。

14.11.3.5.1 类型定义

CHAR 类型

定长字符串。CHAR 列的长度固定为创建表时声明的长度。M 表示列长度（字符的个数，不是字节的个数）。长度可以为从 0 到 255 的任何值。和 VARCHAR 类型不同，CHAR 列在写入时会对数据末尾的空格进行截断。

```
[NATIONAL] CHAR[(M)] [CHARACTER SET charset_name] [COLLATE collation_name]
```

VARCHAR 类型

变长字符串。M 表示最大列长度（字符的最大个数）。VARCHAR 的空间占用大小不得超过 65535 字节。在选择 VARCHAR 长度时，应当根据最长的行的大小和使用的字符集确定。

对于不同的字符集，单个字符所占用的空间可能有所不同。以下表格是各个字符集下单个字符占用的字节数，以及 VARCHAR 列长度的取值范围：

字符集	单个字符字节数	VARCHAR 最大列长度的取值范围
ascii	1	(0, 65535]

字符集	单个字符字节数	VARCHAR 最大列长度的取值范围
latin1	1	(0, 65535]
binary	1	(0, 65535]
utf8	3	(0, 21845]
utf8mb4	4	(0, 16383]

```
[NATIONAL] VARCHAR(M) [CHARACTER SET charset_name] [COLLATE collation_name]
```

TEXT 类型

文本串。最大列长为 65,535 字节。可选的 M 参数以字符为单位，用于自动选择 TEXT 列的最合适类型。例如 TEXT(60) 会产生一个 TINYTEXT 数据类型，最多可容纳 255 字节，即容纳一个 60 字符的 UTF-8 字符串，每个字符最多包含 4 字节（即 4×60=240）。不推荐使用 M 参数。

```
TEXT[(M)] [CHARACTER SET charset_name] [COLLATE collation_name]
```

TINYTEXT 类型

类似于TEXT，区别在于最大列长度为 255。

```
TINYTEXT [CHARACTER SET charset_name] [COLLATE collation_name]
```

MEDIUMTEXT 类型

类似于TEXT，区别在于最大列长度为 16,777,215。但由于txn-entry-size-limit 的限制，TiDB 中默认单行存储最大不超过 6 MiB，可通过配置项将该限制调整至 120 MiB。

```
MEDIUMTEXT [CHARACTER SET charset_name] [COLLATE collation_name]
```

LONGTEXT 类型

类似于TEXT，区别在于最大列长度为 4,294,967,295。但由于txn-entry-size-limit 的限制，TiDB 中默认单行存储最大不超过 6 MiB，可通过配置项将该限制调整至 120 MiB。

```
LONGTEXT [CHARACTER SET charset_name] [COLLATE collation_name]
```

BINARY 类型

类似于CHAR，区别在于 BINARY 存储的是二进制字符串。

```
BINARY(M)
```

VARBINARY 类型

类似于VARCHAR，区别在于 VARBINARY 存储的是二进制字符串。

```
VARBINARY(M)
```

BLOB 类型

二进制大文件。M 表示最大列长度，单位是字节，范围是 0 到 65535。

```
BLOB[(M)]
```

TINYBLOB 类型

类似于BLOB，区别在于最大列长度为 255。

```
TINYBLOB
```

MEDIUMBLOB 类型

类似于BLOB，区别在于最大列长度为 16,777,215。但由于txn-entry-size-limit 的限制，TiDB 中默认单行存储最大不超过 6 MiB，可通过配置项将该限制调整至 120 MiB。

```
MEDIUMBLOB
```

LONGBLOB 类型

类似于BLOB，区别在于最大列长度为 4,294,967,295。但由于txn-entry-size-limit 的限制，TiDB 中默认单行存储最大不超过 6 MiB，可通过配置项将该限制调整至 120 MiB。

```
LONGBLOB
```

ENUM 类型

枚举类型是一个字符串，它只能有一个值的字符串对象。其值必须是从一个固定集合中选取，这个固定集合在创建表的时候定义，语法是：

```
ENUM('value1','value2',...) [CHARACTER SET charset_name] [COLLATE collation_name]
```

例如：

```
ENUM('apple', 'orange', 'pear')
```

枚举类型的值在 TiDB 内部使用数值来存储，每个值会按照定义的顺序转换为一个数字，比如上面的例子中，每个字符串值都会映射为一个数字：

值	数字
NULL	NULL
”	0
‘apple’	1
‘orange’	2
‘pear’	3

更多信息参考 MySQL 枚举文档。

SET 类型

集合类型是一个包含零个或多个值的字符串，其中每个值必须是从一个固定集合中选取，这个固定集合在创建表的时候定义，语法是：

```
SET('value1','value2',...) [CHARACTER SET charset_name] [COLLATE collation_name]
```

例如：

```
SET('1', '2') NOT NULL
```

上面的例子中，这列的有效值可以是：

```
''
'1'
'2'
'1,2'
```

集合类型的值在 TiDB 内部会转换为一个 Int64 数值，每个元素是否存在用一个二进制位的 0/1 值来表示，比如这个例子 `SET('a','b','c','d')`，每一个元素都被映射为一个数字，且每个数字的二进制表示只会有一位是 1：

成员	十进制表示	二进制表示
‘a’	1	0001
‘b’	2	0010
‘c’	4	0100
‘d’	8	1000

这样对于值为 `('a', 'c')` 的元素，其二进制表示即为 0101。

更多信息参考 MySQL 集合文档。

14.11.3.6 JSON 类型

JSON 类型可以存储 JSON 这种半结构化的数据，相比于直接将 JSON 存储为字符串，它的好处在于：

1. 使用 Binary 格式进行序列化，对 JSON 的内部字段的查询、解析加快；
2. 多了 JSON 合法性验证的步骤，只有合法的 JSON 文档才可以放入这个字段中；

JSON 字段本身上，并不能创建索引，但是可以对 JSON 文档中的某个子字段创建索引。例如：

```
CREATE TABLE city (
    id INT PRIMARY KEY,
    detail JSON,
    population INT AS (JSON_EXTRACT(detail, '$.population')),
    index index_name (population)
);
INSERT INTO city (id,detail) VALUES (1, '{"name": "Beijing", "population": 100}');
SELECT id FROM city WHERE population >= 100;
```

14.11.3.6.1 使用限制

- 目前 TiDB 仅支持下推部分 JSON 函数到 TiFlash。详情请参考TiFlash 支持下推的表达式。
- TiDB Backup & Restore（BR）在 v6.3.0 版本对 JSON 列的数据的编码进行了修改。因此不建议使用 BR 恢复包含 JSON 列的数据到 v6.3.0 之前的 TiDB 集群。
- 请勿使用任何同步工具同步非标准 JSON 类型（例如 DATE、DATETIME、TIME 等）的数据。

14.11.3.6.2 MySQL 兼容性

- 当使用二进制类型数据创建 JSON 时，目前 MySQL 会将其误标记为 STRING 类型，而 TiDB 会保持正确的二进制类型。

```
CREATE TABLE test(a json);
INSERT INTO test SELECT json_objectagg('a', b'01010101');

-- 在 TiDB 中，执行以下 SQL 语句返回结果如下所示。在 MySQL 中，执行以下 SQL 语句的结果为 `0,
    ↪  1`。
mysql> SELECT JSON_EXTRACT(JSON_OBJECT('a', b'01010101'), '$.a') = "base64:type15:VQ==" AS
    ↪ r1, JSON_EXTRACT(a, '$.a') = "base64:type15:VQ==" AS r2 FROM test;
+------+------+
| r1   | r2   |
+------+------+
|    0 |    0 |
+------+------+
1 row in set (0.01 sec)
```

详情可见此 issue。

- 当将 ENUM 或 SET 数据类型转换为 JSON 时，TiDB 会检查其格式正确性。例如，当执行下面的 SQL 语句时，TiDB 中会报错：

```
CREATE TABLE t(e ENUM('a'));
INSERT INTO t VALUES ('a');
mysql> SELECT CAST(e AS JSON) FROM t;
ERROR 3140 (22032): Invalid JSON text: The document root must not be followed by other
    ↪ values.
```

详情可见此 issue。

- TiDB 支持使用 ORDER BY 对 JSON Array 或 JSON Object 进行排序。

  当使用 ORDER BY 对 JSON Array 或 JSON Object 进行排序时，MySQL 会返回一个警告，且排序结果与比较运算结果不一致：

```
CREATE TABLE t(j JSON);
INSERT INTO t VALUES ('[1,2,3,4]');
INSERT INTO t VALUES ('[5]');
```

```
mysql> SELECT j FROM t WHERE j < JSON_ARRAY(5);
+--------------+
| j            |
+--------------+
| [1, 2, 3, 4] |
+--------------+
1 row in set (0.00 sec)

-- 在 TiDB 中，执行以下 SQL 语句返回结果如下所示。在 MySQL 中，执行以下 SQL 语句会返回警告
    ↪ “This version of MySQL doesn't yet support 'sorting of non-scalar JSON values'.
    ↪ ”，且排序结果与 `<` 比较结果不一致。
mysql> SELECT j FROM t ORDER BY j;
+--------------+
| j            |
+--------------+
| [1, 2, 3, 4] |
| [5]          |
+--------------+
2 rows in set (0.00 sec)
```

详情可见此 issue。

- 在 INSERT JSON 列时，TiDB 会将值隐式转换为 JSON：

```
CREATE TABLE t(col JSON);

-- 在 TiDB 中，执行以下 INSERT 语句成功。在 MySQL 中，执行以下 INSERT 语句将返回 Invalid
    ↪ JSON text 错误。
INSERT INTO t VALUES (3);
```

有关 JSON 的更多信息，可以参考JSON 函数和生成列。

14.11.4 函数与操作符

14.11.4.1 函数和操作符概述

TiDB 中函数和操作符使用方法与 MySQL 基本一致，详情参见：Functions and Operators。

在 SQL 语句中，表达式可用于诸如 SELECT 语句的 ORDER BY 或 HAVING 子句，SELECT/DELETE/UPDATE 语句的 WHERE 子句，或 SET 语句之类的地方。

可使用字面值，列名，NULL，内置函数，操作符等来书写表达式。其中有些表达式下推到 TiKV 上执行，详见下推到 TiKV 的表达式列表。

14.11.4.2 表达式求值的类型转换

TiDB 中表达式求值的类型转换与 MySQL 基本一致，详情参见 MySQL 表达式求值的类型转换。

14.11.4.3 操作符

操作符名	功能描述
AND, &&	逻辑与
=	赋值（可用于 SET 语句中，或用于 UPDATE 语句的 SET 中）
:=	赋值
BETWEEN ... AND ...	判断值满足范围
BINARY	将一个字符串转换为一个二进制字符串
&	位与
~	位非
|	位或
^	按位异或
CASE	case 操作符
DIV	整数除
/	除法
=	相等比较
<=>	空值安全型相等比较
>	大于
>=	大于或等于
IS	判断一个值是否等于一个布尔值
IS NOT	判断一个值是否不等于一个布尔值
IS NOT NULL	非空判断
IS NULL	空值判断
<<	左移
<	小于
<=	小于或等于
LIKE	简单模式匹配
ILIKE	大小写不敏感的简单模式匹配（TiDB 支持但 MySQL 不支持）
-	减
lstinlineMOD!	求余
NOT, !	取反
NOT BETWEEN ... AND ...	判断值是否不在范围内
!=, <>	不等于
NOT LIKE	不符合简单模式匹配
NOT REGEXP	不符合正则表达式模式匹配
||, OR	逻辑或
+	加
REGEXP	使用正则表达式进行模式匹配
>>	右移
RLIKE	REGEXP 同义词
*	乘
-	取反符号
XOR	逻辑亦或

14.11.4.3.1 不支持的操作符

- SOUNDS LIKE

14.11.4.3.2 操作符优先级

操作符优先级显示在以下列表中，从最高优先级到最低优先级。同一行显示的操作符具有相同的优先级。

```
INTERVAL
BINARY
!
- (unary minus), ~ (unary bit inversion)
^
*, /, DIV, %, MOD
-, +
<<, >>
&
|
= (comparison), <=>, >=, >, <=, <, <>, !=, IS, LIKE, REGEXP, IN
BETWEEN, CASE, WHEN, THEN, ELSE
NOT
AND, &&
XOR
OR, ||
= (assignment), :=
```

详情参见这里。

14.11.4.3.3 比较方法和操作符

操作符名	功能描述
BETWEEN ... AND ...	判断值是否在范围内
COALESCE()	返回第一个非空值
=	相等比较
<=>	空值安全型相等比较
>	大于
>=	大于或等于
GREATEST()	返回最大值
IN()	判断值是否在一个值的集合内
INTERVAL()	返回一个小于第一个参数的参数的下标
IS	判断是否等于一个布尔值
IS NOT	判断是否不等于一个布尔值
IS NOT NULL	非空判断
IS NULL	空值判断
ISNULL()	判断参数是否为空

操作符名	功能描述
LEAST()	返回最小值
<	小于
<=	小于或等于
LIKE	简单模式匹配
ILIKE	大小写不敏感的简单模式匹配（TiDB 支持但 MySQL 不支持）
NOT BETWEEN ... AND ...	判断值是否不在范围内
!=, <>	不等于
NOT IN()	判断值是否不在一个值的集合内
NOT LIKE	不满足简单模式匹配
STRCMP()	比较两个字符串

详情参见这里。

14.11.4.3.4 逻辑操作符

操作符名	功能描述
AND, &&	逻辑与
NOT, !	逻辑非
||, OR	逻辑或
XOR	逻辑异或

详情参见这里。

14.11.4.3.5 赋值操作符

操作符名	功能描述
=	赋值（可用于 SET 语句中，或用于 UPDATE 语句的 SET 中）
:=	赋值

详情参见这里。

14.11.4.3.6 MySQL 兼容性

- MySQL 不支持 ILIKE 操作符。

14.11.4.4 控制流程函数

TiDB 支持使用 MySQL 5.7 中提供的所有控制流程函数。

函数名	功能描述
CASE	Case 操作符
IF()	构建 if/else
IFNULL()	构建 Null if/else
NULLIF()	如果 expr1 = expr2，返回 NULL

14.11.4.5 字符串函数

TiDB 支持使用大部分 MySQL 5.7 中提供的字符串函数、一部分 MySQL 8.0 中提供的字符串函数和一部分 Oracle 21 所提供的函数。

关于 Oracle 函数和 TiDB 函数的对照关系，请参考Oracle 与 TiDB 函数和语法差异对照。

14.11.4.5.1 支持的函数

函数名	功能描述
ASCII()	返回最左字符的数值
BIN()	返回一个数的二进制值的字符串表示
BIT_LENGTH()	返回字符串的位长度
CHAR()	返回由整数的代码值所给出的字符组成的字符串
CHAR_LENGTH()	返回字符串的字符长度
CHARACTER_LENGTH()	与 CHAR_LENGTH() 功能相同
CONCAT()	返回连接的字符串
CONCAT_WS()	返回由分隔符连接的字符串
ELT()	返回指定位置的字符串
EXPORT_SET()	返回一个字符串，其中值位中设置的每个位，可以得到一个 on 字符串，而每个未设置的位，可以得到
FIELD()	返回参数在后续参数中出现的第一个位置
FIND_IN_SET()	返回第一个参数在第二个参数中出现的位置
FORMAT()	返回指定小数位数格式的数字
FROM_BASE64()	解码 base-64 表示的字符串，并返回结果
HEX()	返回一个十进制数或字符串值的 16 进制表示
INSERT()	在指定位置插入一个子字符串，最多不超过指定字符数
INSTR()	返回第一次出现的子字符串的索引
LCASE()	与 LOWER() 功能相同
LEFT()	返回最左侧指定长度的字符
LENGTH()	返回字符串长度，单位为字节
LIKE	进行简单模式匹配
LOCATE()	返回第一次出现的子字符串的位置
LOWER()	返回全小写的参数
LPAD()	返回字符串参数，左侧添加指定字符串
LTRIM()	去掉前缀空格
MAKE_SET()	返回一组用逗号分隔的字符串，这些字符串的位数与给定的 bits 参数对应
MID()	返回一个以指定位置开始的子字符串
NOT LIKE	否定简单模式匹配

函数名	功能描述
NOT REGEXP	REGEXP 的否定形式
OCT()	返回一个数值的八进制表示，形式为字符串
OCTET_LENGTH()	与 LENGTH() 功能相同
ORD()	返回该参数最左侧字符的字符编码
POSITION()	与 LOCATE() 功能相同
QUOTE()	使参数逃逸，为了在 SQL 语句中使用
REGEXP	使用正则表达式匹配模式
REGEXP_INSTR()	返回满足正则的子字符串的第一个索引位置（与 MySQL 不完全兼容，具体请参考正则函数与 MySQL 的兼
REGEXP_LIKE()	判断字符串是否满足正则表达式（与 MySQL 不完全兼容，具体请参考正则函数与 MySQL 的兼容性）
REGEXP_REPLACE()	替换满足正则表达式的子字符串（与 MySQL 不完全兼容，具体请参考正则函数与 MySQL 的兼容性）
REGEXP_SUBSTR()	返回满足正则表达式的子字符串（与 MySQL 不完全兼容，具体请参考正则函数与 MySQL 的兼容性）
REPEAT()	以指定次数重复一个字符串
REPLACE()	替换所有出现的指定字符串
REVERSE()	反转字符串里的所有字符
RIGHT()	返回指定数量的最右侧的字符
RLIKE	与 REGEXP 功能相同
RPAD()	以指定次数添加字符串
RTRIM()	去掉后缀空格
SPACE()	返回指定数量的空格，形式为字符串
STRCMP()	比较两个字符串
SUBSTR()	返回指定的子字符串
SUBSTRING()	返回指定的子字符串
SUBSTRING_INDEX()	从一个字符串中返回指定出现次数的定界符之前的子字符串
TO_BASE64()	返回转化为 base-64 表示的字符串参数
TRANSLATE()	将字符串中出现的所有指定字符替换为其它字符。这个函数不会像 Oracle 一样将空字符串视为NULL
TRIM()	去掉前缀和后缀空格
UCASE()	与 UPPER() 功能相同
UNHEX()	返回一个数的十六进制表示，形式为字符串
UPPER()	参数转换为大写形式
WEIGHT_STRING()	返回字符串的权重

14.11.4.5.2 不支持的函数

- LOAD_FILE()
- MATCH()
- SOUNDEX()

14.11.4.5.3 正则函数与 MySQL 的兼容性

本节介绍 TiDB 中正则函数 REGEXP_INSTR()、REGEXP_LIKE()、REGEXP_REPLACE()、REGEXP_SUBSTR() 与 MySQL 的兼容情况。

语法兼容性

MySQL 的实现使用的是 ICU (International Components for Unicode) 库，TiDB 的实现使用的是 RE2 库，两个库之间的语法差异可以查阅 ICU 文档和 RE2 文档。

匹配模式 `match_type` 兼容性

TiDB 与 MySQL 在 `match_type` 上的差异：

- TiDB 中 `match_type` 可选值为：`"c"`、`"i"`、`"m"`、`"s"`。MySQL 中 `match_type` 可选值为：`"c"`、`"i"`、`"m"`、`"n"`、`"u"`。
- TiDB 中 `"s"` 对应 MySQL 中的 `"n"`，即 `.` 字符匹配行结束符。

  例如：MySQL 中 `SELECT REGEXP_LIKE(a, b, "n")FROM t1;` 在 TiDB 中需要修改为 `SELECT REGEXP_LIKE(a, b, "s")FROM t1;`。
- TiDB 不支持 `match_type` 为 `"u"`。

数据类型兼容性

TiDB 与 MySQL 在二进制字符串 (binary string) 数据类型上的差异：

- MySQL 8.0.22 及以上版本中正则函数不支持二进制字符串，具体信息可查看 MySQL 文档。但在实际使用过程中，如果所有参数或者返回值的数据类型都是二进制字符串，则正则函数可以正常使用，否则报错。
- TiDB 目前完全禁止使用二进制字符串，无论什么情况都会报错。

其它兼容性

TiDB 与 MySQL 在替换空字符串上的差异：

下面以 `REGEXP_REPLACE("", "^$", "123")` 为例：

- MySQL 不会对空串进行替换，其结果为 `""`。
- TiDB 会对空串进行替换，其结果为 `"123"`。

14.11.4.6 数值函数与操作符

TiDB 支持使用 MySQL 5.7 中提供的所有数值函数与操作符。

14.11.4.6.1 算术操作符

操作符名	功能描述
+	加号
-	减号
*	乘号
/	除号
DIV	整数除法
lstinlineMOD!	模运算，取余
-	更改参数符号

14.11.4.6.2 数学函数

函数名	功能描述
POW()	返回参数的指定乘方的结果值
POWER()	返回参数的指定乘方的结果值
EXP()	返回 e（自然对数的底）的指定乘方后的值
SQRT()	返回非负数的二次方根
LN()	返回参数的自然对数
LOG()	返回第一个参数的自然对数
LOG2()	返回参数以 2 为底的对数
LOG10()	返回参数以 10 为底的对数
PI()	返回 pi 的值
TAN()	返回参数的正切值
COT()	返回参数的余切值
SIN()	返回参数的正弦值
COS()	返回参数的余弦值
ATAN()	返回参数的反正切值
ATAN2(), ATAN()	返回两个参数的反正切值
ASIN()	返回参数的反正弦值
ACOS()	返回参数的反余弦值
RADIANS()	返回由度转化为弧度的参数
DEGREES()	返回由弧度转化为度的参数
MOD()	返回余数
ABS()	返回参数的绝对值
CEIL()	返回不小于参数的最小整数值
CEILING()	返回不小于参数的最小整数值
FLOOR()	返回不大于参数的最大整数值
ROUND()	返回参数最近似的整数或指定小数位数的数值
RAND()	返回一个随机浮点值
SIGN()	返回参数的符号
CONV()	不同数基间转换数字，返回数字的字符串表示
TRUNCATE()	返回被舍位至指定小数位数的数字
CRC32()	计算循环冗余码校验值并返回一个 32 位无符号值

14.11.4.7 日期和时间函数

TiDB 支持使用 MySQL 5.7 中提供的所有日期和时间函数。

> **注意：**
>
> - MySQL 常常会接受格式不正确的日期和时间值。例如，`'2020-01-01\n\t01:01:01'` 和 `'2020-01_01\n\t01:01'` 被视为有效的日期和时间值。

- TiDB 会尽量与 MySQL 的行为保持一致，但可能无法在所有情况下完全匹配。建议使用正确的格式化日期，TiDB 文档中未记录将如何处理格式不正确的值。

14.11.4.7.1 日期时间函数表

函数名	功能描述
ADDDATE()	将时间间隔添加到日期上
ADDTIME()	时间数值相加
CONVERT_TZ()	转换时区
CURDATE()	返回当前日期
CURRENT_DATE(), CURRENT_DATE	与 CURDATE() 同义
CURRENT_TIME(), CURRENT_TIME	与 CURTIME() 同义
CURRENT_TIMESTAMP(), CURRENT_TIMESTAMP	与 NOW() 同义
CURTIME()	返回当前时间
DATE()	从日期或日期/时间表达式中提取日期部分
DATE_ADD()	将时间间隔添加到日期上
DATE_FORMAT()	返回满足指定格式的日期/时间
DATE_SUB()	从日期减去指定的时间间隔
DATEDIFF()	返回两个日期间隔的天数
DAY()	与 DAYOFMONTH() 同义
DAYNAME()	返回星期名称
DAYOFMONTH()	返回参数对应的天数部分 (1-31)
DAYOFWEEK()	返回参数对应的星期下标
DAYOFYEAR()	返回参数代表一年的哪一天 (1-366)
EXTRACT()	提取日期/时间中的单独部分
FROM_DAYS()	将天数转化为日期
FROM_UNIXTIME()	将 Unix 时间戳格式化为日期
GET_FORMAT()	返回满足日期格式的字符串
HOUR()	提取日期/时间表达式中的小时部分
LAST_DAY	返回参数中月份的最后一天
LOCALTIME(), LOCALTIME	与 NOW() 同义
LOCALTIMESTAMP, LOCALTIMESTAMP()	与 NOW() 同义
MAKEDATE()	根据给定的年份和一年中的天数生成一个日期
MAKETIME()	根据给定的时、分、秒生成一个时间
MICROSECOND()	返回参数的微秒部分
MINUTE()	返回参数的分钟部分
MONTH()	返回参数的月份部分
MONTHNAME()	返回参数的月份名称
NOW()	返回当前日期和时间
PERIOD_ADD()	在年 -月表达式上添加一段时间 (数个月)
PERIOD_DIFF()	返回间隔的月数
QUARTER()	返回参数对应的季度 (1-4)

函数名	功能描述
SEC_TO_TIME()	将秒数转化为 ‘HH:MM:SS’ 的格式
SECOND()	返回秒数 (0-59)
STR_TO_DATE()	将字符串转化为日期
SUBDATE()	当传入三个参数时作为 DATE_SUB() 的同义
SUBTIME()	从一个时间中减去一段时间
SYSDATE()	返回该方法执行时的时间
TIME()	返回参数的时间表达式部分
TIME_FORMAT()	格式化时间
TIME_TO_SEC()	返回参数对应的秒数
TIMEDIFF()	返回时间间隔
TIMESTAMP()	传入一个参数时候, 该方法返回日期或日期/时间表达式，传入两个参数时候，返回
TIMESTAMPADD()	在日期/时间表达式上增加一段时间间隔
TIMESTAMPDIFF()	从日期/时间表达式中减去一段时间间隔
TO_DAYS()	将参数转化对应的天数 (从第 0 年开始)
TO_SECONDS()	将日期或日期/时间参数转化为秒数 (从第 0 年开始)
UNIX_TIMESTAMP()	返回一个 Unix 时间戳
UTC_DATE()	返回当前的 UTC 日期
UTC_TIME()	返回当前的 UTC 时间
UTC_TIMESTAMP()	返回当前的 UTC 日期和时间
WEEK()	返回参数所在的一年中的星期数
WEEKDAY()	返回星期下标
WEEKOFYEAR()	返回参数在日历中对应的一年中的星期数
YEAR()	返回参数对应的年数
YEARWEEK()	返回年数和星期数

14.11.4.7.2 MySQL 兼容性

TiDB 支持 `str_to_date()` 函数，但是无法解析所有的日期和时间值。此外，TiDB 不支持以下日期和时间格式化选项：

格式	说明
“%a”	星期名的缩写（例如 Sun..Sat）
“%D”	带英文后缀的月份日期（例如 0th，1st，2nd，3rd）
“%U”	星期 (00..53)，星期日是每周的第一天；WEEK() mode 0
“%u”	星期 (00..53)，星期一是每周的第一天；WEEK() mode 1
“%V”	星期 (01..53)，星期日是每周的第一天；WEEK() mode 2；和 “%X” 一起使用
“%v”	星期 (01..53)，星期一是每周的第一天；WEEK() mode 3；和 “%x” 一起使用
“%W”	星期名（例如 Sunday..Saturday）
“%w”	一周中的天名 (0=Sunday..6=Saturday)
“%X”	星期天是每周第一天的年份，数字类型，四位数字
“%x”	星期一是每周第一天的年份，数字类型，四位数字

更多信息，参见 GitHub Issue #30082。

14.11.4.7.3 相关系统变量

`default_week_format` 变量影响 `WEEK()` 函数。

14.11.4.8 位函数和操作符

TiDB 支持使用 MySQL 5.7 中提供的所有位函数和操作符。

位函数和操作符表

函数和操作符名	功能描述
BIT_COUNT()	返回参数二进制表示中为 1 的个数
&	位与
~	按位取反
|	位或
[^](https://dev.mysql.com/doc/refman/5.7/en/bit-functions.html#operator_bitwise-xor)	位亦或
<<	左移
>>	右移

14.11.4.9 Cast 函数和操作符

Cast 函数和操作符用于将某种数据类型的值转换为另一种数据类型。TiDB 支持使用 MySQL 5.7 中提供的所有 Cast 函数和操作符。

14.11.4.9.1 Cast 函数和操作符表

函数和操作符名	功能描述
BINARY	将一个字符串转换成一个二进制字符串
CAST()	将一个值转换成一个确定类型
CONVERT()	将一个值转换成一个确定类型

> **注意：**
>
> TiDB 和 MySQL 对于 `SELECT CAST(MeN AS CHAR)`（或者等价的 `SELECT CONVERT(MeM, CHAR)`）的结果显示不一致，其中 `MeN` 是用科学计数法表示的双精度浮点数。MySQL 在 `-15 <= N <= 14` 时显示完整数值，在 `N < -15` 或 `N > 14` 时显示科学计数法。而 TiDB 始终显示完整数值。例如，MySQL 对于 `SELECT CAST(3.1415e15 AS CHAR)` 的显示结果为 `3.1415e15`，而 TiDB 的显示结果为 `3141500000000000`。

14.11.4.10 加密和压缩函数

TiDB 支持使用 MySQL 5.7 中提供的大部分加密和压缩函数。

14.11.4.10.1 支持的函数

函数名	功能描述
MD5()	计算字符串的 MD5 校验和
PASSWORD()	计算并返回密码字符串
RANDOM_BYTES()	返回随机字节向量
SHA1(), SHA()	计算 SHA-1 160 位校验和
SHA2()	计算 SHA-2 校验和
SM3()	计算 SM3 校验和（MySQL 中暂不支持该函数）
AES_DECRYPT()	使用 AES 解密
AES_ENCRYPT()	使用 AES 加密
COMPRESS()	返回经过压缩的二进制字符串
UNCOMPRESS()	解压缩字符串
UNCOMPRESSED_LENGTH()	返回字符串解压后的长度
VALIDATE_PASSWORD_STRENGTH()	确定密码强度

14.11.4.10.2 相关系统变量

block_encryption_mode 变量设置 AES_ENCRYPT() 和 AES_DECRYPT() 所使用的加密模式。

14.11.4.10.3 不支持的函数

- DES_DECRYPT()、DES_ENCRYPT()、OLD_PASSWORD() 和 ENCRYPT()：这些函数在 MySQL 5.7 中被废弃，并且已在 MySQL 8.0 中移除。
- 只在 MySQL 企业版中支持的函数。见 Issue #2632。

14.11.4.11 锁函数

TiDB 支持 MySQL 5.7 中的大部分用户级锁函数。

14.11.4.11.1 支持的函数

函数名	功能描述
GET_LOCK(lockName, timeout)	获取咨询锁。lockName 参数不得超过 64 个字符。在超时前，TiDB 最长等待 timeout 秒并返回失败。
RELEASE_LOCK(lockName)	释放先前获取的锁。lockName 参数不得超过 64 个字符。
RELEASE_ALL_LOCKS()	释放当前会话持有的所有锁。

14.11.4.11.2 MySQL 兼容性

- TiDB 允许的最短超时时间为 1 秒，最长超时时间为 1 小时（即 3600 秒）。而 MySQL 允许最短 0 秒和最长无限超时（timeout=-1）。TiDB 会自动将超出范围的值转换为最接近的允许值，timeout=-1 会被转换为 3600 秒。
- TiDB 不会自动检测用户级锁导致的死锁。死锁会话将在 1 小时内超时，但你也可以在任一受影响的会话上使用 KILL 语句手动终止死锁。你还可以通过始终用相同顺序获取用户级锁的方法来防止死锁。
- 在 TiDB 中，锁对集群中所有 TiDB 服务器生效。而在 MySQL Cluster 和 Group Replication 中，锁只对本地单个服务器生效。

14.11.4.11.3 不支持的函数

- IS_FREE_LOCK()
- IS_USED_LOCK()

14.11.4.12 信息函数

TiDB 支持使用 MySQL 5.7 中提供的大部分信息函数。

14.11.4.12.1 支持的函数

函数名	功能描述
BENCHMARK()	循环执行一个表达式
CONNECTION_ID()	返回当前连接的连接 ID (线程 ID)
CURRENT_RESOURCE_GROUP()	返回当前连接的资源组名
CURRENT_USER(), CURRENT_USER	返回当前用户的用户名和主机名
DATABASE()	返回默认 (当前) 的数据库名
FOUND_ROWS()	该函数返回对于一个包含 LIMIT 的 SELECT 查询语句，在不包含 LIMIT 的情况下回返回的记录数
LAST_INSERT_ID()	返回最后一条 INSERT 语句中自增列的值
ROW_COUNT()	影响的行数
SCHEMA()	与 DATABASE() 同义
SESSION_USER()	与 USER() 同义
SYSTEM_USER()	与 USER() 同义
USER()	返回客户端提供的用户名和主机名
VERSION()	返回当前 MySQL 服务器的版本信息

14.11.4.12.2 不支持的函数

- `CHARSET()`
- `COERCIBILITY()`
- `COLLATION()`

14.11.4.13 JSON 函数

TiDB 支持 MySQL 5.7 GA 版本发布的大多数 JSON 函数。

14.11.4.13.1 创建 JSON 值的函数

函数	功能描述
JSON_ARRAY([val[, val] ···])	根据一系列元素创建一个 JSON 文档
JSON_OBJECT(key, val[, key, val] ···)	根据一系列 K/V 对创建一个 JSON 文档
JSON_QUOTE(string)	返回一个字符串，该字符串为带引号的 JSON 值

14.11.4.13.2 搜索 JSON 值的函数

函数	功能描述
JSON_CONTAINS(target, candidate[, path])	通过返回 1 或 0 来表示目标 JSON 文档中是否包含给定的 candidate JSON 文档
JSON_CONTAINS_PATH(json_doc, one_or_all, path[, path] ···)	通过返回 0 或 1 来表示一个 JSON 文档在给定路径是否包含数据
JSON_EXTRACT(json_doc, path[, path] ···)	从 JSON 文档中解出某一路径对应的子文档
->	返回执行路径后面的 JSON 列的值；`JSON_EXTRACT(doc, path_literal)` 的别名
->>	返回执行路径后面的 JSON 列的值和转义后的结果；`JSON_UNQUOTE(JSON_EXTRACT(doc, path_literal))` 的别名
JSON_KEYS(json_doc[, path])	返回从 JSON 对象的顶级值作为 JSON array 的键，如果给定了路径参数，则从选定路径中获取顶级键
JSON_SEARCH(json_doc, one_or_all, search_str[, escape_char[, path] ···])	返回指定字符在 JSON 文档中的路径
value MEMBER OF(json_array)	如果传入值是 JSON array 中的一个元素，返回 1，否则返回 0
JSON_OVERLAPS(json_doc1, json_doc2)	表示两个 JSON 文档中是否包含公共部分。返回 1 表示两个 JSON 文档中包含公共部分，否则返回 0

14.11.4.13.3 修改 JSON 值的函数

函数	功能描述
JSON_APPEND(json_doc, path, value)	JSON_ARRAY_APPEND 的别名
JSON_ARRAY_APPEND(json_doc, path, value)	将值追加到指定路径的 JSON 数组的末尾
JSON_ARRAY_INSERT(json_doc, path, val[, path, val] ...)	将数组插入 JSON 文档，并返回修改后的文档
JSON_INSERT(json_doc, path, val[, path, val] ...)	在 JSON 文档中在某一路径下插入子文档
JSON_MERGE(json_doc, json_doc[, json_doc] ...)	已废弃的 JSON_MERGE_PRESERVE 别名
JSON_MERGE_PATCH(json_doc, json_doc[, json_doc] ...)	合并 JSON 文档
JSON_MERGE_PRESERVE(json_doc, json_doc[, json_doc] ...)	将两个或多个 JSON 文档合并成一个文档，并返回合并结果
JSON_REMOVE(json_doc, path[, path] ...)	移除 JSON 文档中某一路径下的子文档
JSON_REPLACE(json_doc, path, val[, path, val] ...)	替换 JSON 文档中的某一路径下的子文档
JSON_SET(json_doc, path, val[, path, val] ...)	在 JSON 文档中为某一路径设置子文档
JSON_UNQUOTE(json_val)	去掉 JSON 值外面的引号，返回结果为字符串
JSON_ARRAY_APPEND(json_doc, path, val[, path, val] ...)	将值添加到 JSON 文档指定数组的末尾，并返回添加结果

函数	功能描述
JSON_ARRAY_INSERT(json_doc, path, val[, path, val] ...)	将值插入到 JSON 文档的指定位置，并返回插入结果

14.11.4.13.4 返回 JSON 值属性的函数

函数	功能描述
JSON_DEPTH(json_doc)	返回 JSON 文档的最大深度
JSON_LENGTH(json_doc[, path])	返回 JSON 文档的长度；如果路径参数已定，则返回该路径下值的长度
JSON_TYPE(json_val)	检查某 JSON 文档内部内容的类型
JSON_VALID(json_doc)	检查 JSON 文档内容是否有效；用于将列转换为 JSON 类型之前对该列进行检查

14.11.4.13.5 效用函数

函数	功能描述
JSON_PRETTY(json_doc)	格式化 JSON 文档
JSON_STORAGE_FREE(json_doc)	返回该 JSON 对象的存储空间中空闲的字节数。由于 TiDB 采用与 MySQL 完全不同的存储结构，本函数对合法的 JSON 值总是返回 0，主要用于兼容 MySQL 8.0
JSON_STORAGE_SIZE(json_doc)	返回存储 JSON 值所需的大致字节大小，由于不考虑 TiKV 压缩的字节大小，因此函数的输出与 MySQL 不严格兼容

14.11.4.13.6 聚合函数

函数	功能描述
JSON_ARRAYAGG(key)	提供指定列 key 的聚合
JSON_OBJECTAGG(key, value)	提供给定两列键值对的聚合

14.11.4.13.7 另请参阅

- JSON Function Reference
- JSON Data Type

14.11.4.14 GROUP BY 聚合函数

本文将详细介绍 TiDB 支持的聚合函数。

14.11.4.14.1 TiDB 支持的聚合函数

TiDB 支持的 MySQL GROUP BY 聚合函数如下所示：

函数名	功能描述
COUNT()	返回检索到的行的数目
COUNT(DISTINCT)	返回不同值的数目
SUM()	返回和
AVG()	返回平均值
MAX()	返回最大值
MIN()	返回最小值
GROUP_CONCAT()	返回连接的字符串
VARIANCE(), VAR_POP()	返回总体标准方差
STD(), STDDEV(), STDDEV_POP	返回总体标准差
VAR_SAMP()	返回采样方差
STDDEV_SAMP()	返回采样标准方差
JSON_OBJECTAGG(key, value)	将结果集返回为单个含 (key, value) 键值对的 JSON object

> 注意：
>
> - 除非另有说明，否则聚合函数默认忽略 NULL 值。
> - 如果在不包含 GROUP BY 子句的语句中使用聚合函数，则相当于对所有行进行分组。

另外，TiDB 还支持以下聚合函数：

- APPROX_PERCENTILE(expr, constant_integer_expr)

  该函数用于计算 expr 值的百分位数。参数 constant_integer_expr 是一个取值为区间 [1,100] 内整数的常量表达式，表示百分数。一个百分位数 Pk（k为百分数）表示数据集中至少有 k% 的数据小于等于 Pk。

该函数中，表达式的返回结果必须为数值类型或日期与时间类型。函数不支持计算其他类型的返回结果，并直接返回 NULL。

以下是一个计算第 50 百分位数的例子：

```
drop table if exists t;
create table t(a int);
insert into t values(1), (2), (3);
```

```
select approx_percentile(a, 50) from t;
```

```
+--------------------------+
| approx_percentile(a, 50) |
+--------------------------+
|                        2 |
+--------------------------+
1 row in set (0.00 sec)
```

上述聚合函数除 GROUP_CONCAT() 和 APPROX_PERCENTILE() 以外，均可作为窗口函数使用。

14.11.4.14.2 GROUP BY 修饰符

TiDB 目前不支持 GROUP BY 修饰符，例如 WITH ROLLUP，将来会提供支持。详情参阅 #4250。

14.11.4.14.3 对 SQL 模式的支持

TiDB 支持 SQL 模式 ONLY_FULL_GROUP_BY，当启用该模式时，TiDB 拒绝不明确的非聚合列的查询。例如，以下查询在启用 ONLY_FULL_GROUP_BY 时是不合规的，因为 SELECT 列表中的非聚合列 “b” 在 GROUP BY 语句中不显示：

```
drop table if exists t;
create table t(a bigint, b bigint, c bigint);
insert into t values(1, 2, 3), (2, 2, 3), (3, 2, 3);
```

```
select a, b, sum(c) from t group by a;
```

```
+------+------+--------+
| a    | b    | sum(c) |
+------+------+--------+
|    1 |    2 |      3 |
|    2 |    2 |      3 |
|    3 |    2 |      3 |
+------+------+--------+
3 rows in set (0.01 sec)
```

```
set sql_mode = 'ONLY_FULL_GROUP_BY';
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
select a, b, sum(c) from t group by a;
```

```
ERROR 1055 (42000): Expression #2 of SELECT list is not in GROUP BY clause and contains
    ↪ nonaggregated column 'b' which is not functionally dependent on columns in GROUP BY
    ↪ clause; this is incompatible with sql_mode=only_full_group_by
```

目前，TiDB 默认开启 SQL 模式ONLY_FULL_GROUP_BY。

与 MySQL 的区别

TiDB 目前实现的 ONLY_FULL_GROUP_BY 没有 MySQL 5.7 严格。例如，假设我们执行以下查询，希望结果按 “c” 排序：

```
drop table if exists t;
create table t(a bigint, b bigint, c bigint);
insert into t values(1, 2, 1), (1, 2, 2), (1, 3, 1), (1, 3, 2);
select distinct a, b from t order by c;
```

要对结果进行排序，必须先清除重复。但选择保留哪一行会影响 c 的保留值，也会影响排序，并使其具有任意性。

在 MySQL 中，ORDER BY 表达式需至少满足以下条件之一，否则 DISTINCT 和 ORDER BY 查询将因不合规而被拒绝：

- 表达式等同于 SELECT 列表中的一个。
- 表达式引用并属于查询选择表的所有列都是 SELECT 列表的元素。

但是在 TiDB 中，上述查询是合规的，详情参阅 #4254。

TiDB 中另一个标准 SQL 的扩展允许 HAVING 子句中的引用使用 SELECT 列表中的别名表达式。例如：以下查询返回在 orders 中只出现一次的 name 值

```
select name, count(name) from orders
group by name
having count(name) = 1;
```

这个 TiDB 扩展允许在聚合列的 HAVING 子句中使用别名：

```
select name, count(name) as c from orders
group by name
having c = 1;
```

标准 SQL 只支持 GROUP BY 子句中的列表达式，以下语句不合规，因为 FLOOR(value/100) 是一个非列表达式：

```
select id, floor(value/100)
from tbl_name
group by id, floor(value/100);
```

TiDB 对标准 SQL 的扩展支持 GROUP BY 子句中非列表达式，认为上述语句合规。

标准 SQL 也不支持 GROUP BY 子句中使用别名。TiDB 对标准 SQL 的扩展支持使用别名，查询的另一种写法如下：

```
select id, floor(value/100) as val
from tbl_name
group by id, val;
```

14.11.4.14.4 相关系统变量

group_concat_max_len 变量设置 GROUP_CONCAT() 函数中的最大项目数。

14.11.4.15 窗口函数

TiDB 中窗口函数的使用方法与 MySQL 8.0 基本一致，详情可参见 MySQL 窗口函数。由于窗口函数会使用一些保留关键字，可能导致原先可以正常执行的 SQL 语句在升级 TiDB 后无法被解析语法，此时可以将 tidb_enable_window_function 设置为 0，该参数的默认值为 1。

本页列出的窗口函数可以下推到 TiFlash。

TiDB 支持除 GROUP_CONCAT() 和 APPROX_PERCENTILE() 以外的所有GROUP BY 聚合函数。此外，TiDB 支持的其他窗口函数如下：

函数名	功能描述
CUME_DIST()	返回一组值中的累积分布
DENSE_RANK()	返回分区中当前行的排名，并且排名是连续的
FIRST_VALUE()	当前窗口中第一行的表达式值
LAG()	分区中当前行前面第 N 行的表达式值
LAST_VALUE()	当前窗口中最后一行的表达式值
LEAD()	分区中当前行后面第 N 行的表达式值
NTH_VALUE()	当前窗口中第 N 行的表达式值
NTILE()	将分区划分为 N 桶，为分区中的每一行分配桶号
PERCENT_RANK()	返回分区中小于当前行的百分比
RANK()	返回分区中当前行的排名，排名可能不连续
ROW_NUMBER()	返回分区中当前行的编号

14.11.4.16 其他函数

TiDB 支持使用 MySQL 5.7 中提供的大部分其他函数。

14.11.4.16.1 支持的函数

函数名	功能描述
ANY_VALUE()	在 ONLY_FULL_GROUP_BY 模式下，防止带有 GROUP BY 的语句报错
BIN_TO_UUID()	将通用唯一识别码 (UUID) 从二进制格式转换为文本格式
DEFAULT()	返回表的某一列的默认值

函数名	功能描述
INET_ATON()	将 IP 地址转换为数值
INET_NTOA()	将数值转换为 IP 地址
INET6_ATON()	将 IPv6 地址转换为数值
INET6_NTOA()	将数值转换为 IPv6 地址
IS_IPV4()	判断参数是否为 IPv4 地址
IS_IPV4_COMPAT()	判断参数是否为兼容 IPv4 的地址
IS_IPV4_MAPPED()	判断参数是否为 IPv4 映射的地址
IS_IPV6()	判断参数是否为 IPv6 地址
NAME_CONST()	可以用于重命名列名
SLEEP()	让语句暂停执行几秒时间
UUID()	返回一个通用唯一识别码 (UUID)
UUID_TO_BIN()	将 UUID 从文本格式转换为二进制格式
VALUES()	定义 INSERT 语句使用的值

14.11.4.16.2 不支持的函数

函数名	功能描述
UUID_SHORT()	基于特定假设提供唯一的 UUID，目前这些假设在 TiDB 中不存在，详见 TiDB #4620
MASTER_WAIT_POS()	与 MySQL 同步相关

14.11.4.17 精度数学

TiDB 中精度数学计算与 MySQL 中基本一致。详情请参见：Precision Math

- 数值类型
- DECIMAL 数据类型的特性

14.11.4.17.1 数值类型

精确数值运算的范围包括精确值数据类型（整型和 DECIMAL 类型），以及精确值数字字面量。近似值数据类型和近似值数字字面量被作为浮点数来处理。

精确值数字字面量包含整数部分或小数部分，或二者都包含。精确值数字字面量可以包含符号位。例如：1, .2, 3.4, -5, -6.78, +9.10。

近似值数字字面量以一个包含尾数和指数的科学计数法表示（基数为 10）。其中尾数和指数可以分别或同时带有符号位。例如：1.2E3, 1.2E-3, -1.2E3, -1.2E-3。

两个看起来相似的数字可能会被以不同的方式进行处理。例如：2.34 是精确值（定点数），而 2.3E0 是近似值（浮点数）。

DECIMAL 数据类型是定点数类型，其运算是精确计算。FLOAT 和 DOUBLE 数据类型是浮点类型，其运算是近似计算。

14.11.4.17.2 DECIMAL 数据类型的特性

本节讨论 DECIMAL 数据类型的特性，主要涉及以下几点：

1. 最大位数
2. 存储格式
3. 存储要求

DECIMAL 列的声明语法为 DECIMAL(M, D)。其中参数值意义及其范围如下：

- M 表示最大的数字位数（精度），1<= M <= 65。
- D 表示小数点右边数字的位数（标度）。1 <= D <= 30 且不大于 M。

M 的最大值 65 表示 DECIMAL 值的计算精确到 65 位数字。该精度同样适用于其精确值字面量。

DECIMAL 列的值采用二进制进行存储，其将每 9 位十进制数字包装成 4 个字节。其中整数和小数部分分别确定所需的存储空间。如果数字位数为 9 的倍数，则每 9 位十进制数字各采用 4 个字节进行存储，对于剩余不足 9 位的数字，所需的存储空间如下表所示

剩余数字位数	存储所需字节数
0	0
1–2	1
3–4	2
5–6	3
7–9	4

例如：

- 定义类型为 DECIMAL(18, 9) 的列，其小数点两侧均各包含 9 位十进制数字。因此，分别需要 4 个字节的存储空间。
- 定义类型为 DECIMAL(20, 6) 的列，其小数部分包含 6 位十进制数字，整数部分包含 14 位十进制数字。整数部分中 9 位数字需要 4 个字节进行存储，其余 5 位数字需要 3 个字节进行存储。小数部分 6 位数字需要 3 个字节进行存储。

DECIMAL 列不存储前导的字符 + 或字符 - 或数字 0。如果将 +0003.1 插入到 DECIMAL(5, 1) 列中，则将其存储为 3.1。对于负数，不存储字符 - 的字面值。

DECIMAL 列不允许插入大于列定义的隐含范围的值。例如：DECIMAL(3, 0) 列范围为 -999 到 999。DECIMAL(M, D) 列小数点左边部分最多支持 M-D 位数字。

有关 DECIMAL 值的内部格式完整说明，请参阅 TiDB 源码文件 types/mydecimal.go。

14.11.4.17.3 表达式计算

在涉及精度数学计算的表达式中，TiDB 会尽可能不做任何修改的使用每个输入的数值。比如：在计算比较函数时，参与运算的数字将不做任何改变。在严格 SQL 模式下，向一个数据列插入一个值时，如果该值处于这

一列的值域范围内，这个值将直接不做任何修改的直接插入进去，提取这个值的时候，取得的值和插入的值将会是同一个值。当处于非严格 SQL 模式时，TiDB 会允许数据插入过程中发生的数据截断。

处理数值类型表达式取决于这个表达式参数的具体值：

- 当表达式参数中包含近似值时，这个表达式的结果也是近似值，TiDB 会使用浮点数对应的计算逻辑返回一个浮点数的结果。
- 当表达式参数中不包含任何近似值时（也就是说表达式的参数全部是精确值），如果某个精确值包含小数部分，TiDB 会对这个表达式使用 DECIMAL 对应的计算逻辑，返回一个 DECIMAL 的结果，精确到 65 位数字。
- 其他情况下，表达式只会包含整数参数，这个表达式的结果也是精确的，TiDB 会使用整数对应的计算逻辑返回一个整数结果，精度和 BIGINT 保持一致（64 位）。

如果数值类型表达式中包含字符串参数，这些字符串参数将被转换成双精度浮点数，这个表达式的计算结果将是个近似值。

向一个数值类型列插入数据的具体行为会受到 SQL 模式的影响。接下来的讨论将围绕严格模式以及 ERROR_FOR_DIVISION_BY_ZERO 模式展开，如果要打开所有的限制，可以简单的使用 TRADITIONAL 模式，这个模式将同时使用严格模式以及 ERROR_FOR_DIVISION_BY_ZERO 模式：

```
SET sql_mode = 'TRADITIONAL';
```

向一个具有精确值类型（DECIMAL 或者整数类型）的列插入数据时，如果插入的数据位于该列的值域范围内将使用该数据的精确值。如果该数据的小数部分太长，将会发生数值修约，这时会有 warning 产生，具体内容可以看“数值修约”。

如果该数据整数部分太长：

- 如果没有开启严格模式，这个值会被截断并产生一个 warning。
- 如果开启了严格模式，将会产生一个数据溢出的 error。

如果向一个数值类型列插入字符串，如果该字符串中包含非数值部分，TiDB 将这样做类型转换：

- 在严格模式下，没有以数字开头的字符串（即使是一个空字符串）不能被被用作数字值并会返回一个 error 或者是 warning。
- 以数字开头的字符串可以被转换，不过末尾的非数字部分会被截断。如果被截断的部分包含的不全是空格，在严格模式下这回产生一个 error 或者 warning。

默认情况下，如果计算的过程中发生了除数是 0 的现象将会得到一个 NULL 结果，并且不会有 warning 产生。通过设置适当的 SQL 模式，除以 0 的操作可以被限制。当设置 ERROR_FOR_DIVISION_BY_ZERO SQL 模式时，TiDB 的行为是：

- 如果设置了严格 SQL 模式，INSERT 和 UPDATE 的过程中如果发生了除以 0 的操作，正在进行的 INSERT 或者 UPDATE 操作会被禁止，并且会返回一个 error。
- 如果没有设置严格 SQL 模式，除以 0 的操作仅会返回一个 warning。

假设我们有如下的 SQL 语句：

```
INSERT INTO t SET i = 1/0;
```

不同的 SQL 模式将会导致不同的结果如下：

sql_mode 的值	结果
”	没有 warning，没有 error，i 被设为 NULL
strict	没有 warning，没有 error，i 被设为 NULL
ERROR_FOR_DIVISION_BY_ZERO	有 warning，没有 error，i 被设为 NULL
strict, ERROR_FOR_DIVISION_BY_ZERO	有 error，插入失败

14.11.4.17.4 数值修约

round() 函数的结果取决于他的参数是否是精确值：

- 如果参数是精确值，round() 函数将使用四舍五入的规则。
- 如果参数是一个近似值，round() 表达式的结果可能和 MySQL 不太一样。

```
SELECT ROUND(2.5), ROUND(25E-1);
```

```
+------------+--------------+
| ROUND(2.5) | ROUND(25E-1) |
+------------+--------------+
|          3 |            3 |
+------------+--------------+
1 row in set (0.00 sec)
```

向一个 DECIMAL 或者整数类型列插入数据时，round 的规则将采用 round half away from zero 的方式：

```
CREATE TABLE t (d DECIMAL(10,0));
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
INSERT INTO t VALUES(2.5),(2.5E0);
```

```
Query OK, 2 rows affected, 2 warnings (0.00 sec)
```

```
SELECT d FROM t;
```

```
+------+
| d    |
+------+
|    3 |
|    3 |
+------+
2 rows in set (0.00 sec)
```

14.11.4.18 集合运算

TiDB 支持三种集合运算：并集 (UNION)，差集 (EXCEPT) 和交集 (INTERSECT)。最小的集合单位是一个SELECT 语句。

14.11.4.18.1 并集 (UNION)

数学上，两个集合 A 和 B 的并集是含有所有属于 A 或属于 B 的元素。下面是一个 UNION 的例子：

```
SELECT 1 UNION SELECT 2;
+---+
| 1 |
+---+
| 2 |
| 1 |
+---+
2 rows in set (0.00 sec)
```

TiDB 支持 UNION ALL 和 UNION DISTINCT 并集，两者区别在于 UNION DISTINCT 会对并集结果去重复，而 UNION ↪ ALL 不会。TiDB 中默认使用 UNION DISTINCT。

```
CREATE TABLE t1 (a int);
CREATE TABLE t2 (a int);
INSERT INTO t1 VALUES (1),(2);
INSERT INTO t2 VALUES (1),(3);
```

UNION DISTINCT与 UNION ALL 的结果分别如下：

```
SELECT * FROM t1 UNION DISTINCT SELECT * FROM t2;
+---+
| a |
+---+
| 1 |
| 2 |
| 3 |
+---+
3 rows in set (0.00 sec)

SELECT * FROM t1 UNION ALL SELECT * FROM t2;
+---+
| a |
+---+
| 1 |
| 2 |
| 1 |
| 3 |
+---+
4 rows in set (0.00 sec)
```

14.11.4.18.2 差集 (EXCEPT)

若 A 和 B 是集合，则 A 与 B 的差集是由所有属于 A 但不属于 B 的元素组成的集合。

```
SELECT * FROM t1 EXCEPT SELECT * FROM t2;
+---+
| a |
+---+
| 2 |
+---+
1 rows in set (0.00 sec)
```

差集 (EXCEPT) 暂时不支持 EXCEPT ALL。

14.11.4.18.3 交集 (INTERSECT)

数学上，两个集合 A 和 B 的交集是含有所有既属于 A 又属于 B 的元素，而且没有其他元素的集合。

```
SELECT * FROM t1 INTERSECT SELECT * FROM t2;
+---+
| a |
+---+
| 1 |
+---+
1 rows in set (0.00 sec)
```

交集 (INTERSECT) 暂时不支持 INTERSECT ALL。交集 (INTERSECT) 的计算优先级大于差集 (EXCEPT) 和并集 (UNION)。

```
SELECT * FROM t1 UNION ALL SELECT * FROM t1 INTERSECT SELECT * FROM t2;
+---+
| a |
+---+
| 1 |
| 1 |
| 2 |
+---+
3 rows in set (0.00 sec)
```

14.11.4.18.4 括号优先

TiDB 支持使用括号修改集合运算的优先级，如同四则运算中先计算括号部分，集合运算也先计算括号内的部分。

```
(SELECT * FROM t1 UNION ALL SELECT * FROM t1) INTERSECT SELECT * FROM t2;
+---+
| a |
+---+
| 1 |
+---+
1 rows in set (0.00 sec)
```

14.11.4.18.5 与 ORDER BY 和 Limit 结合

TiDB 支持单独为整个集合运算进行 ORDER BY 或者 LIMIT。

```
(SELECT * FROM t1 UNION ALL SELECT * FROM t1 INTERSECT SELECT * FROM t2) ORDER BY a LIMIT 2;
+---+
| a |
+---+
| 1 |
| 1 |
+---+
2 rows in set (0.00 sec)
```

14.11.4.19 下推到 TiKV 的表达式列表

当 TiDB 从 TiKV 中读取数据的时候，TiDB 会尽量下推一些表达式运算到 TiKV 中，从而减少数据传输量以及 TiDB 单一节点的计算压力。本文将介绍 TiDB 已支持下推的表达式，以及如何禁止下推特定表达式。

TiFlash 也支持本页列出的函数和算子下推。

14.11.4.19.1 已支持下推的表达式列表

表达式分类	具体操作
逻辑运算	AND (&&), OR (||), NOT (!), XOR
位运算	&, ~, |, ^, <<, >>
比较运算	<, <=, =, != (<>), >, >=, <=>, BETWEEN ··· AND ···, COALESCE(), IN(), INTERVAL(), IS NOT NULL, IS NOT, IS NULL, IS, ISNULL(), LIKE, NOT BETWEEN ··· AND ···, NOT IN(), NOT LIKE, STRCMP()
数值运算	+, -, *, /, DIV, % (MOD), -, ABS(), ACOS(), ASIN(), ATAN(), ATAN2(), ATAN(), CEIL(), CEILING(), CONV(), COS(), COT(), CRC32(), DEGREES(), EXP(), FLOOR(), LN(), LOG(), LOG10(), LOG2(), MOD(), PI(), POW(), POWER(), RADIANS(), RAND(), ROUND(), SIGN(), SIN(), SQRT()
控制流运算	CASE, IF(), IFNULL()
JSON 运算	JSON_ARRAY([val[, val] ···]), JSON_CONTAINS(target, candidate[, path]), JSON_EXTRACT(json_doc, path[, path] ···), JSON_INSERT(json_doc, path, val[, path, val] ···), JSON_LENGTH(json_doc[, path]), JSON_MERGE(json_doc, json_doc[, json_doc] ···), JSON_OBJECT([key, val[, key, val] ···]), JSON_REMOVE(json_doc, path[, path] ···), JSON_REPLACE(json_doc, path, val[, path, val] ···), JSON_SET(json_doc, path, val[, path, val] ···), JSON_TYPE(json_val), JSON_UNQUOTE(json_val), JSON_VALID(val)

表达式分类	具体操作
日期运算	DATE(), DATE_FORMAT(), DATEDIFF(), DAYOFMONTH(), DAYOFWEEK(), DAYOFYEAR(), FROM_DAYS(), HOUR(), MAKEDATE(), MAKETIME(), MICROSECOND(), MINUTE(), MONTH(), MONTHNAME(), PERIOD_ADD(), PERIOD_DIFF(), SEC_TO_TIME(), SECOND(), SYSDATE(), TIME_TO_SEC(), TIMEDIFF(), WEEK(), WEEKOFYEAR(), YEAR()
字符串函数	ASCII(), BIT_LENGTH(), CHAR(), CHAR_LENGTH(), CONCAT(), CONCAT_WS(), ELT(), FIELD(), HEX(), LENGTH(), LIKE, LTRIM(), MID(), NOT LIKE, NOT REGEXP, REGEXP, REPLACE(), REVERSE(), RIGHT(), RTRIM(), SPACE(), STRCMP(), SUBSTR(), SUBSTRING()
聚合函数	COUNT(), COUNT(DISTINCT), SUM(), AVG(), MAX(), MIN(), VARIANCE(), VAR_POP(), STD(), STDDEV(), STDDEV_POP, VAR_SAMP(), STDDEV_SAMP(), JSON_ARRAYAGG(key), JSON_OBJECTAGG(key, value)
加密和压缩函数	MD5(), SHA1(), SHA(), UNCOMPRESSED_LENGTH()
Cast 函数	CAST(), CONVERT()
其他函数	UUID()

14.11.4.19.2 禁止特定表达式下推

当已支持下推的表达式列表中的函数和运算符，或特定的数据类型（仅限ENUM 类型和BIT 类型）的计算过程因下推而出现异常时，你可以使用黑名单功能禁止其下推，从而快速恢复 TiDB 业务。具体而言，你可以将函数名、运算符名，或数据列类型加入黑名单 `mysql.expr_pushdown_blacklist` 中，以禁止特定表达式下推。具体方法，请参阅表达式下推黑名单。

14.11.4.20 TiDB 特有的函数

以下函数为 TiDB 中特有的函数，与 MySQL 不兼容：

函数名	函数说明
`TIDB_BOUNDED_STALENES` `↪ ()`	$IDB_BOUNDED_STALENESS 函数指示 TiDB 在指定时间范围内读取尽可能新的数据。参见使用 AS OF TIMESTAMP 语法读取历史数据。
`TIDB_DECODE_KEY(` `↪ str)`	TIDB_DECODE_KEY 函数用于将 TiDB 编码的键输入解码为包含 `_tidb_rowid` 和 `table_id` 的 JSON 结构。你可以在一些系统表和日志输出中找到 TiDB 的编码键。
`TIDB_DECODE_PLAN(` `↪ str)`	TIDB_DECODE_PLAN 函数用于解码 TiDB 执行计划。
`TIDB_IS_DDL_OWNER` `↪ ()`	TIDB_IS_DDL_OWNER 函数用于检查你连接的 TiDB 实例是否是 DDL Owner。DDL Owner 代表集群中所有其他节点执行 DDL 语句的 TiDB 实例。

函数名	函数说明
TIDB_PARSE_TSO(num ↪)	TIDB_PARSE_TSO 函数用于从 TiDB TSO 时间戳中提取物理时间戳。参见tidb_current_ts。
TIDB_VERSION()	TIDB_VERSION 函数用于获取当前连接的 TiDB 服务器版本和构建详细信息。
TIDB_DECODE_SQL_DIGEST ↪ (digests, ↪ stmtTruncateLength ↪)	TIDB_DECODE_SQL_DIGESTS 函数用于在集群中查询一组 SQL Digest 所对应的 SQL 语句的归一化形式（即去除格式和参数后的形式）。
VITESS_HASH(str)	VITESS_HASH 函数返回与 Vitess 的 HASH 函数兼容的字符串哈希值，有助于从 Vitess 迁移数据。
TIDB_SHARD()	TIDB_SHARD 函数用于创建一个 SHARD INDEX 来打散热点索引。SHARD INDEX 是一种以 TIDB_SHARD 函数为前缀的表达式索引。

14.11.4.20.1 示例

下面为部分以上函数的示例。

TIDB_DECODE_KEY

TIDB_DECODE_KEY 函数用于将 TiDB 编码的键输入解码为包含 _tidb_rowid 和 table_id 的 JSON 结构。你可以在一些系统表和日志输出中找到 TiDB 的编码键。

以下示例中，表 t1 有一个隐藏的 rowid，该 rowid 由 TiDB 生成。语句中使用了 TIDB_DECODE_KEY 函数。结果显示，隐藏的 rowid 被解码后并输出，这是典型的非聚簇主键结果。

```
SELECT START_KEY, TIDB_DECODE_KEY(START_KEY) FROM information_schema.tikv_region_status WHERE
    ↪ table_name='t1' AND REGION_ID=2\G
```

```
*************************** 1. row ***************************
                START_KEY: 7480000000000000FF3B5F728000000000FF1DE3F10000000000FA
TIDB_DECODE_KEY(START_KEY): {"_tidb_rowid":1958897,"table_id":"59"}
1 row in set (0.00 sec)
```

以下示例中，表 t2 有一个复合聚簇主键。由 JSON 输出可知，输出结果的 handle 项中包含了主键部分两列的信息，即两列的名称和对应的值。

```
show create table t2\G
```

```
*************************** 1. row ***************************
       Table: t2
Create Table: CREATE TABLE `t2` (
  `id` binary(36) NOT NULL,
  `a` tinyint(3) unsigned NOT NULL,
  `v` varchar(512) DEFAULT NULL,
  PRIMARY KEY (`a`,`id`) /*T![clustered_index] CLUSTERED */
```

```
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin
1 row in set (0.001 sec)
```

```
select * from information_schema.tikv_region_status where table_name='t2' limit 1\G
```

```
*************************** 1. row ***************************
               REGION_ID: 48
               START_KEY: 7480000000000000
                    ↪ FF3E5F720400000000FF0000000601633430FF3338646232FF2D64FF3531632D3131FF65FF622D386337352DF
                    ↪
                 END_KEY:
                TABLE_ID: 62
                 DB_NAME: test
              TABLE_NAME: t2
                IS_INDEX: 0
                INDEX_ID: NULL
              INDEX_NAME: NULL
           EPOCH_CONF_VER: 1
            EPOCH_VERSION: 38
            WRITTEN_BYTES: 0
               READ_BYTES: 0
         APPROXIMATE_SIZE: 136
         APPROXIMATE_KEYS: 479905
  REPLICATIONSTATUS_STATE: NULL
REPLICATIONSTATUS_STATEID: NULL
1 row in set (0.005 sec)
```

```
select tidb_decode_key('7480000000000000
    ↪ FF3E5F720400000000FF0000000601633430FF3338646232FF2D64FF3531632D3131FF65FF622D386337352DFFFF3830653635303
    ↪ ');
```

```
+--
    ↪ ----------------------------------------------------------------------------------------------------------
    ↪
| tidb_decode_key('7480000000000000
    ↪ FF3E5F720400000000FF0000000601633430FF3338646232FF2D64FF3531632D3131FF65FF622D386337352DFFFF3830653635303
    ↪ ') |
+--
    ↪ ----------------------------------------------------------------------------------------------------------
    ↪
| {"handle":{"a":"6","id":"c4038db2-d51c-11eb-8c75-80e65018a9be"},"table_id":62}
    ↪
    ↪ |
+--
    ↪ ----------------------------------------------------------------------------------------------------------
    ↪
```

```
1 row in set (0.001 sec)
```

TIDB_DECODE_PLAN

你可以在慢查询日志中找到编码形式的 TiDB 执行计划，然后使用 TIDB_DECODE_PLAN() 函数将编码的执行计划解码为易读的形式。

该函数很有用，因为在执行语句时 TiDB 会捕获执行计划。重新执行 EXPLAIN 中的语句可能会产生不同的结果，因为数据分布和统计数据会随着时间的推移而变化。

```
SELECT tidb_decode_plan('8
    ↪ QIYMAkzMV83CQEH8E85LjAOCWRhdGE6U2VsZWN0aW9uXzYJOTYwCXRpbWU6NzEzLjHCtXMsIGxvb3BzOjIsIGNvcF90YXNrOiB7bnVtOi
    ↪ ')\G
```

```
*************************** 1. row ***************************
  tidb_decode_plan('8
      ↪ QIYMAkzMV83CQEH8E85LjAOCWRhdGE6U2VsZWN0aW9uXzYJOTYwCXRpbWU6NzEzLjHCtXMsIGxvb3BzOjIsIGNvcF90YXNrOiB7bnVt
      ↪ :     id                    task         estRows    operator info
      ↪                             actRows    execution info
      ↪
      ↪ memory     disk
    TableReader_7          root         319.04     data:Selection_6                          960
        ↪         time:713.1µs, loops:2, cop_task: {num: 1, max: 568.5µs, proc_keys: 0, rpc_num
        ↪ : 1, rpc_time: 549.1µs, copr_cache_hit_ratio: 0.00}    3.99 KB    N/A
      └─Selection_6          cop[tikv]    319.04     lt(test.t.a, 10000)
        ↪ 960        tikv_task:{time:313.8µs, loops:960}
        ↪
        ↪ N/A        N/A
        └─TableFullScan_5    cop[tikv]    960        table:t, keep order:false, stats:pseudo
          ↪ 960        tikv_task:{time:153µs, loops:960}
          ↪
          ↪ N/A        N/A
```

TIDB_PARSE_TSO

TIDB_PARSE_TSO 函数用于从 TiDB TSO 时间戳中提取物理时间戳。

TSO 指 Time Stamp Oracle，是 PD (Placement Driver) 为每个事务提供的单调递增的时间戳。TSO 是一串数字，包含以下两部分：

- 一个物理时间戳
- 一个逻辑计数器

```
BEGIN;
SELECT TIDB_PARSE_TSO(@@tidb_current_ts);
ROLLBACK;
```

```
+----------------------------------+
| TIDB_PARSE_TSO(@@tidb_current_ts) |
+----------------------------------+
| 2021-05-26 11:33:37.776000       |
+----------------------------------+
1 row in set (0.0012 sec)
```

以上示例使用 TIDB_PARSE_TSO 函数从 tidb_current_ts 会话变量提供的可用时间戳编号中提取物理时间戳。因为每个事务都会分配到时间戳，所以此函数在事务中运行。

TIDB_VERSION

TIDB_VERSION 函数用于获取当前连接的 TiDB 服务器版本和构建详细信息。向 GitHub 上提交 issue 时，你可使用此函数获取相关信息。

```
SELECT TIDB_VERSION()\G
```

```
*************************** 1. row ***************************
TIDB_VERSION(): Release Version: v5.1.0-alpha-13-gd5e0ed0aa-dirty
Edition: Community
Git Commit Hash: d5e0ed0aaed72d2f2dfe24e9deec31cb6cb5fdf0
Git Branch: master
UTC Build Time: 2021-05-24 14:39:20
GoVersion: go1.13
Race Enabled: false
TiKV Min Version: v3.0.0-60965b006877ca7234adaced7890d7b029ed1306
Check Table Before Drop: false
1 row in set (0.00 sec)
```

TIDB_DECODE_SQL_DIGESTS

TIDB_DECODE_SQL_DIGESTS() 函数用于在集群中查询一组 SQL Digest 所对应的 SQL 语句的归一化形式（即去除格式和参数后的形式）。函数接受 1 个或 2 个参数：

- digests：字符串类型，该参数应符合 JSON 字符串数组的格式，数组中的每个字符串应为一个 SQL Digest。
- stmtTruncateLength：可选参数，整数类型，用来限制返回结果中每条 SQL 语句的长度，超过指定的长度会被截断。0 表示不限制长度。

返回一个字符串，符合 JSON 字符串数组的格式，数组中的第 i 项为参数 digests 中的第 i 个元素所对应的语句。如果参数 digests 中的某一项不是一个有效的 SQL Digest 或系统无法查询到其对应的 SQL 语句，则返回结果中对应项为 null。如果指定了截断长度（stmtTruncateLength > 0），则返回结果中每条超过该长度的语句，保留前 stmtTruncateLength 个字符，并在尾部增加 "..." 后缀表示发生了截断。如果参数 digests 为 NULL，则函数的返回值为 NULL。

注意：

- 仅持有 PROCESS 权限的用户可以使用该函数。
- TIDB_DECODE_SQL_DIGESTS 执行时，TiDB 内部从 Statement Summary 一系列表中查询每个 SQL Digest 所对应的语句，因而并不能保证对任意 SQL Digest 都总是能查询到对应的语句，只有在集群中执行过的语句才有可能被查询到，且是否能查询到受 Statement Summary 表相关配置的影响。有关 Statement Summary 表的详细说明，参见Statement Summary Tables。
- 该函数开销较大，在行数很多的查询中（比如在规模较大、比较繁忙的集群上查询 information_schema.cluster_tidb_trx 全表时）直接使用该函数可能导致查询运行时间较长。请谨慎使用。
  - 该函数开销大的原因是，其每次被调用时，都会在内部发起对 STATEMENTS_SUMMARY、STATEMENTS_SUMMARY_HISTORY、CLUSTER_STATEMENTS_SUMMARY 和 CLUSTER_STATEMENTS_SUMMARY_HISTORY 这几张表的查询，且其中涉及 UNION 操作。且该函数目前不支持向量化，即对于多行数据调用该函数时，对每行都会独立进行一次上述的查询。

```
set @digests = '["e6f07d43b5c21db0fbb9a31feac2dc599787763393dd5acbfad80e247eb02ad5","38
    ↪ b03afa5debbdf0326a014dbe5012a62c51957f1982b3093e748460f8b00821","
    ↪ e5796985ccafe2f71126ed6c0ac939ffa015a8c0744a24b7aee6d587103fd2f7"]';

select tidb_decode_sql_digests(@digests);
```

```
+------------------------------------+
| tidb_decode_sql_digests(@digests)  |
+------------------------------------+
| ["begin",null,"select * from `t`"] |
+------------------------------------+
1 row in set (0.00 sec)
```

上面的例子中，参数是一个包含 3 个 SQL Digest 的 JSON 数组，其对应的 SQL 语句分别为查询结果中给出的三项。但是其中第二条 SQL Digest 所对应的 SQL 语句未能从集群中找到，因而结果中的第二项为 null。

```
select tidb_decode_sql_digests(@digests, 10);
```

```
+---------------------------------------+
| tidb_decode_sql_digests(@digests, 10) |
+---------------------------------------+
| ["begin",null,"select * f..."]        |
+---------------------------------------+
1 row in set (0.01 sec)
```

上述调用指定了第二个参数（即截断长度）为 10，而查询结果中的第三条语句的长度大于 10，因而仅保留了前 10 个字符，并在尾部添加了 "..." 表示发生了截断。

另请参阅：

- Statement Summary Tables
- INFORMATION_SCHEMA.TIDB_TRX

TIDB_SHARD

TIDB_SHARD 函数用于创建一个 SHARD INDEX 来打散热点索引。SHARD INDEX 是一种以 TIDB_SHARD 函数为前缀的表达式索引。

- 创建方式：

  使用 uk((tidb_shard(a)), a)) 为字段 a 创建一个 SHARD INDEX。当二级唯一索引 uk((tidb_shard(a)), a)) 的索引字段 a 上存在因单调递增或递减而产生的热点时，索引的前缀 tidb_shard(a) 会打散热点，从而提升集群可扩展性。

- 适用场景：

  - 二级唯一索引上 key 值存在单调递增或递减导致的写入热点，且该索引包含的列是整型。
  - 业务中 SQL 语句根据该二级索引的全部字段做等值查询，查询可以是单独的 SELECT，也可以是 UPDATE，DELETE 等产生的内部查询，等值查询包括 a = 1 或 a IN (1, 2, ...) 两种方式。

- 使用限制：

  - 非等值查询无法使用索引。
  - 查询条件中 AND 和 OR 混合且最外层是 AND 算子时无法使用 SHARD INDEX。
  - GROUP BY 无法使用 SHARD INDEX。
  - ORDER BY 无法使用 SHARD INDEX。
  - ON 子句无法使用 SHARD INDEX。
  - WHERE 子查询无法使用 SHARD INDEX。
  - SHARD INDEX 只能打散整型字段的唯一索引。
  - SHARD INDEX 联合索引可能失效。
  - SHARD INDEX 无法走 FastPlan 流程，影响优化器性能。
  - SHARD INDEX 无法使用执行计划缓存。

TIDB_SHARD 函数的使用示例如下：

- 使用 TIDB_SHARD 函数计算 SHARD 值

  以下示例说明如何使用 TIDB_SHARD 函数计算 12373743746 的 SHARD 值。

```
SELECT TIDB_SHARD(12373743746);
```

- 计算得出 SHARD 值为：

```
+-------------------------+
| TIDB_SHARD(12373743746) |
+-------------------------+
|                     184 |
+-------------------------+
1 row in set (0.00 sec)
```

- 使用 TIDB_SHARD 函数创建 SHARD INDEX

```
CREATE TABLE test(id INT PRIMARY KEY CLUSTERED, a INT, b INT, UNIQUE KEY uk((tidb_shard(a)),
    ↪  a));
```

14.11.4.21 Oracle 与 TiDB 函数和语法差异对照

本文档提供了 Oracle 与 TiDB 的函数和语法差异对照，方便你根据 Oracle 函数查找对应的 TiDB 函数，了解 Oracle 与 TiDB 语法差异。

> **注意：**
>
> 本文的内容是基于 Oracle 12.2.0.1.0 和 TiDB v5.4.0，其他版本可能存在差异。

14.11.4.21.1 函数对照表

下表列出了 Oracle 与 TiDB 部分函数的对照表。

函数	Oracle 语法	TiDB 语法	说明
转换数据类型			
	TO_NUMBER(key)		
	TO_CHAR(key)		
	CONVERT(key,dataType)	TiDB 支持转换为下面类型：BINARY、CHAR、DATE、DATETIME、TIME、SIGNED INTEGER、UNSIGNED INTEGER 和 DECIMAL。	
日期类型转换为字符串类型			
	TO_CHAR(SYSDATE,'yyyy-MM-dd hh24:mi:ss')		
	TO_CHAR(SYSDATE,'yyyy-MM-dd')		
	DATE_FORMAT(NOW(),'%Y-%m-%d %H:%i:%s')		
	DATE_FORMAT(NOW(),'%Y-%m-%d')		
		TiDB 的格式化字符串大小写敏感。	
字符串类型转换为日期类型			
	TO_DATE('2021-05-28 17:31:37','yyyy-MM-dd hh24:mi:ss')		
	TO_DATE('2021-05-28','yyyy-MM-dd hh24:mi:ss')		
	STR_TO_DATE('2021-05-28 17:31:37','%Y-%m-%d %H:%i:%s')		
	STR_TO_DATE('2021-05-28','%Y-%m-%d%T')		

| TiDB 的格式化字符串大小写敏感。| | 获取系统当前时间（精确到秒）| SYSDATE | NOW() | | | 获取当前时间（精确到微秒）| SYSTIMESTAMP | CURRENT_TIMESTAMP(6) | | | 获取两个日期相差的天数 | date1 - date2 | DATEDIFF(date1, date2) | | | 获取两个日期间隔月份 | MONTHS_BETWEEN(ENDDATE,SYSDATE) | TIMESTAMPDIFF(MONTH,SYSDATE,ENDDATE) | Oracle 中 MONTHS_BETWEEN() 函数与 TiDB 中 TIMESTAMPDIFF() 函数的结果会有误差。TIMESTAMPDIFF() 只保留整数月。使用时需要注意，两个函数的参数位置相反。| | 日期增加/减少 n 天 | DATEVAL + n | DATE_ADD(dateVal,INTERVAL n DAY) | n 可为负数。| | 日期增加/减少 n 月 | ADD_MONTHS(dateVal,n)| DATE_ADD(dateVal,INTERVAL n MONTH) | n 可为负数。| | 获取日期到日 | TRUNC(SYSDATE) |

```
CAST(NOW()AS DATE)
```

```
DATE_FORMAT(NOW(),'%Y-%m-%d')
```

| TiDB 中 CAST 与 DATE_FORMAT 结果一致。| | 获取日期当月第一天 | TRUNC(SYSDATE,'mm') | DATE_ADD(CURDATE(),interval - day(CURDATE())+ 1 day) | | | 截取数据 | TRUNC(2.136)= 2 TRUNC(2.136,2)= 2.13 | TRUNCATE(2.136,0)= 2 TRUNCATE(2.136,2)= 2.13 | 数据精度保留，直接截取相应小数位，不涉及四舍五入。| | 获取序列下一个值 | sequence_name.NEXTVAL | NEXTVAL(sequence_name) | | | 获取随机序列值 | SYS_GUID() | UUID() | TiDB 返回一个通用唯一识别码 (UUID)。| | 左/右外连接 | SELECT * FROM a, b WHERE a.id = b.id(+);SELECT * FROM a, b WHERE a.id(+)= b.id; | SELECT * FROM a LEFT JOIN b ON a.id = b.id;SELECT * FROM a RIGHT JOIN b ON a.id = b.id; | 关联查询时，TiDB 不支持使用 (+) 实现左/右关联，只能通过 LEFT JOIN 或 RIGHT JOIN 实现。| | NVL() | NVL(key,val) | IFNULL(key,val) | 如果该字段值为 NULL，则返回 val 值，否则返回该字段的值。| | NVL2() | NVL2(key, val1, val2) | IF(key is NULL, val1, val2) | 如果该字段值非 NULL，则返回 val1 值，否则返回 val2 值。| | DECODE() |

```
DECODE(key,val1,val2,val3)
```

```
DECODE(value,if1,val1,if2,val2,...,ifn,valn,val)
```

|

```
IF(key=val1,val2,val3)
```

```
CASE WHEN value=if1 THEN val1 WHEN value=if2 THEN val2,...,WHEN value=ifn THEN valn ELSE val END
```

|

如果该字段值等于 val1，则返回 val2，否则返回 val3。

当该字段值满足条件 1 (if1) 时，返回 val1，满足条件 2 (if2) 时，返回 val2，满足条件 3 (if3) 时，返回 val3。

| | 拼接字符串 a 和 b | ‘a’ || ‘b’ | CONCAT('a','b') | | | 获取字符串长度 | LENGTH(str) | CHAR_LENGTH(str) | | | 获取子串 | SUBSTR('abcdefg',0,2)= 'ab' SUBSTR('abcdefg',1,2)= 'ab' | SUBSTRING('abcdefg',0,2)= ''SUBSTRING('abcdefg',1,2)= 'ab' |

Oracle 中起始位置 0 与 1 作用一样。

TiDB 中 0 开始获取的子串为空，若需从字符串的起始位置开始，则应从 1 开始。

| | 字符串在源字符串中的位置 | INSTR('abcdefg','b',1,1) | INSTR('abcdefg','b') | 从字符串 'abcdefg' 第一个字符开始查询，返回 'b' 字符串第一次出现的位置。| | 字符串在源字符串中的位置 | INSTR('stst','s',1,2) | LENGTH(SUBSTRING_INDEX('stst','s',2))+ 1 | 从字符串 'stst' 第一个字符开始查找，返回 's' 字符第二次出现的位置。| | 字符串在源字符串中的位置 | INSTR('abcabc','b',2,1) | LOCATE('b','abcabc',2) | 从字符串 'abcabc' 第二个字符开始查询，返回 'b' 字符第一次出现的位置。| | 列合并为行 | LISTAGG(CONCAT(E.dimensionid,'---',E.DIMENSIONNAME),'***')within GROUP(ORDER BY DIMENSIONNAME) | GROUP_CONCAT(CONCAT(E.dimensionid,'---',E.DIMENSIONNAME)ORDER BY DIMENSIONNAME SEPARATOR '***') | 将一列字段

合并为一行并根据 *** 符号进行分割。| | ASCII 值转化为对应字符 | CHR(n) | CHAR(n) | Oracle 中制表符 (CHR(9))、换行符 (CHR(10))、 回车符 (CHR(13)) 对应 TiDB 中的 CHAR(9)、 CHAR(10)、 CHAR(13)。 |

14.11.4.21.2 语法差异

本节介绍 Oracle 部分语法与 TiDB 的差异。

字符串语法

Oracle 中字符串只能使用单引号 (’)。例如 'a'。

TiDB 中字符串可以使用单引号 (’) 或双引号 (“”)。例如 'a' 或 "a"。

NULL 与空字符串的区分

Oracle 中不区分 NULL 和空字符串 ''，即 NULL 与 '' 是等价的。

TiDB 中区分 NULL 和空字符串 ''。

INSERT 语句中读写同一张表

Oracle 支持 INSERT 语句中读写同一张表。例如：

```
INSERT INTO table1 VALUES (feild1,(SELECT feild2 FROM table1 WHERE...))
```

TiDB 不支持 INSERT 语句中读写同一张表。例如：

```
INSERT INTO table1 VALUES (feild1,(SELECT T.fields2 FROM table1 T WHERE...))
```

获取前 n 行数据

Oracle 通过 ROWNUM <= n 获取前 n 行数据。例如，ROWNUM <= 10。

TiDB 通过 LIMIT n 获取前 n 行数据。例如，LIMIT 10。Hibernate Query Language (HQL) 方式运行带 LIMIT 的 SQL 语句会出现错误，需要将 Hibernate 的运行方式改为 SQL 方式运行。

UPDATE 语句多表更新

Oracle 多表更新时不需要列出具体的字段更新关系。例如：

```
UPDATE test1 SET(test1.name,test1.age) = (SELECT test2.name,test2.age FROM test2 WHERE test2.id=
    ↪ test1.id)
```

TiDB 多表更新时需要在 SET 时把具体的字段更新关系都列出来。例如：

```
UPDATE test1,test2 SET test1.name=test2.name,test1.age=test2.age WHERE test1.id=test2.id
```

派生表别名

Oracle 多表查询时，派生表可以不起别名。例如：

```
SELECT * FROM (SELECT * FROM test)
```

TiDB 多表查询时，每一个派生出来的表都必须有一个自己的别名。例如：

```
SELECT * FROM (SELECT * FROM test) t
```

差集运算

Oracle 使用 MINUS 进行差集运算。例如：

```
SELECT * FROM t1 MINUS SELECT * FROM t2
```

TiDB 不支持 MINUS，需要改写为 EXCEPT 进行差集运算。例如：

```
SELECT * FROM t1 EXCEPT SELECT * FROM t2
```

注释语法

Oracle 中注释语法为 --注释，其中 -- 后面不需要空格。

TiDB 中注释语法为 -- 注释，其中 -- 后面需要有一个空格。

分页查询

Oracle 分页查询时 OFFSET m 表示跳过 m 行数据，FETCH NEXT n ROWS ONLY 表示取 n 条数据。例如：

```
SELECT * FROM tables OFFSET 0 ROWS FETCH NEXT 2000 ROWS ONLY
```

TiDB 使用 LIMIT n OFFSET m 等价改写 OFFSET m ROWS FETCH NEXT n ROWS ONLY。例如：

```
SELECT * FROM tables LIMIT 2000 OFFSET 0
```

ORDER BY 语句对 NULL 的排序规则

Oracle 中 ORDER BY 语句对 NULL 的排序规则：

- ORDER BY COLUMN ASC 时，NULL 默认被放在最后。
- ORDER BY COLUMN DESC 时，NULL 默认被放在最前。
- ORDER BY COLUMN [ASC|DESC] NULLS FIRST 时，强制 NULL 放在最前，非 NULL 的值仍然按声明顺序 ASC|DESC 进行排序。
- ORDER BY COLUMN [ASC|DESC] NULLS LAST 时，强制 NULL 放在最后，非 NULL 的值仍然按声明顺序 ASC|DESC 进行排序。

TiDB 中 ORDER BY 语句对 NULL 的排序规则：

- ORDER BY COLUMN ASC 时，NULL 默认被放在最前。
- ORDER BY COLUMN DESC 时，NULL 默认被放在最后。

下表是 Oracle 与 TiDB 中等价 ORDER BY 语句示例：

Oracle 中的 ORDER BY	TiDB 中的 ORDER BY
SELECT * FROM t1 ORDER BY name NULLS FIRST;	SELECT * FROM t1 ORDER BY name;

Oracle 中的 ORDER BY	TiDB 中的 ORDER BY
SELECT * FROM t1 ORDER BY name DESC NULLS LAST;	SELECT * FROM t1 ORDER BY name DESC ;
SELECT * FROM t1 ORDER BY name DESC NULLS FIRST;	SELECT * FROM t1 ORDER BY ISNULL(name)DESC, name DESC;
SELECT * FROM t1 ORDER BY name ASC NULLS LAST;	SELECT * FROM t1 ORDER BY ISNULL(name), name;

14.11.5 聚簇索引

聚簇索引 (clustered index) 是 TiDB 从 v5.0 开始支持的特性，用于控制含有主键的表数据的存储方式。通过使用聚簇索引，TiDB 可以更好地组织数据表，从而提高某些查询的性能。有些数据库管理系统也将聚簇索引称为“索引组织表” (index-organized tables)。

目前 TiDB 中含有主键的表分为以下两类：

- NONCLUSTERED，表示该表的主键为非聚簇索引。在非聚簇索引表中，行数据的键由 TiDB 内部隐式分配的 _tidb_rowid 构成，而主键本质上是唯一索引，因此非聚簇索引表存储一行至少需要两个键值对，分别为
  - _tidb_rowid（键）- 行数据（值）
  - 主键列数据（键）- _tidb_rowid（值）
- CLUSTERED，表示该表的主键为聚簇索引。在聚簇索引表中，行数据的键由用户给定的主键列数据构成，因此聚簇索引表存储一行至少只要一个键值对，即
  - 主键列数据（键）- 行数据（值）

> **注意：**
>
> TiDB 仅支持根据表的主键来进行聚簇操作。聚簇索引启用时，“主键”和“聚簇索引”两个术语在一些情况下可互换使用。主键指的是约束（一种逻辑属性），而聚簇索引描述的是数据存储的物理实现。

14.11.5.1 使用场景

相较于非聚簇索引表，聚簇索引表在以下几个场景中，性能和吞吐量都有较大优势：

- 插入数据时会减少一次从网络写入索引数据。
- 等值条件查询仅涉及主键时会减少一次从网络读取数据。

- 范围条件查询仅涉及主键时会减少多次从网络读取数据。
- 等值或范围条件查询仅涉及主键的前缀时会减少多次从网络读取数据。

另一方面，聚簇索引表也存在一定的劣势：

- 批量插入大量取值相邻的主键时，可能会产生较大的写热点问题。
- 当使用大于 64 位的数据类型作为主键时，可能导致表数据需要占用更多的存储空间。该现象在存在多个二级索引时尤为明显。

14.11.5.2 使用方法

14.11.5.2.1 创建聚簇索引表

从 TiDB 版本 5.0 开始，要指定一个表的主键是否使用聚簇索引，可以在 `CREATE TABLE` 语句中将 `CLUSTERED` 或者 `NONCLUSTERED` 非保留关键字标注在 `PRIMARY KEY` 后面，例如：

```
CREATE TABLE t (a BIGINT PRIMARY KEY CLUSTERED, b VARCHAR(255));
CREATE TABLE t (a BIGINT PRIMARY KEY NONCLUSTERED, b VARCHAR(255));
CREATE TABLE t (a BIGINT KEY CLUSTERED, b VARCHAR(255));
CREATE TABLE t (a BIGINT KEY NONCLUSTERED, b VARCHAR(255));
CREATE TABLE t (a BIGINT, b VARCHAR(255), PRIMARY KEY(a, b) CLUSTERED);
CREATE TABLE t (a BIGINT, b VARCHAR(255), PRIMARY KEY(a, b) NONCLUSTERED);
```

注意，列定义中的 `KEY` 和 `PRIMARY KEY` 含义相同。

此外，TiDB 支持使用可执行的注释语法指定聚簇索引属性：

```
CREATE TABLE t (a BIGINT PRIMARY KEY /*T![clustered_index] CLUSTERED */, b VARCHAR(255));
CREATE TABLE t (a BIGINT PRIMARY KEY /*T![clustered_index] NONCLUSTERED */, b VARCHAR(255));
CREATE TABLE t (a BIGINT, b VARCHAR(255), PRIMARY KEY(a, b) /*T![clustered_index] CLUSTERED */);
CREATE TABLE t (a BIGINT, b VARCHAR(255), PRIMARY KEY(a, b) /*T![clustered_index] NONCLUSTERED */
    ↪ );
```

对于未显式指定该关键字的语句，默认行为受系统变量@@global.tidb_enable_clustered_index 影响。该变量有三个取值：

- `OFF` 表示所有主键默认使用非聚簇索引。
- `ON` 表示所有主键默认使用聚簇索引。
- `INT_ONLY` 此时的行为受配置项 `alter-primary-key` 控制。如果该配置项取值为 `true`，则所有主键默认使用非聚簇索引；如果该配置项取值为 `false`，则由单个整数类型的列构成的主键默认使用聚簇索引，其他类型的主键默认使用非聚簇索引。

系统变量 `@@global.tidb_enable_clustered_index` 本身的默认值为 `ON`。

14.11.5.2.2 添加、删除聚簇索引

目前 TiDB 不支持在建表之后添加或删除聚簇索引，也不支持聚簇索引和非聚簇索引的互相转换。例如：

```
ALTER TABLE t ADD PRIMARY KEY(b, a) CLUSTERED; -- 暂不支持
ALTER TABLE t DROP PRIMARY KEY;      -- 如果主键为聚簇索引，则不支持
ALTER TABLE t DROP INDEX `PRIMARY`; -- 如果主键为聚簇索引，则不支持
```

14.11.5.2.3 添加、删除非聚簇索引

TiDB 支持在建表之后添加或删除非聚簇索引。此时可以选择显式指定 NONCLUSTERED 关键字或省略关键字：

```
ALTER TABLE t ADD PRIMARY KEY(b, a) NONCLUSTERED;
ALTER TABLE t ADD PRIMARY KEY(b, a); -- 不指定关键字，则为非聚簇索引
ALTER TABLE t DROP PRIMARY KEY;
ALTER TABLE t DROP INDEX `PRIMARY`;
```

14.11.5.2.4 查询主键是否为聚簇索引

可通过以下方式来确定一张表的主键是否使用了聚簇索引：

- 执行语句 SHOW CREATE TABLE。
- 执行语句 SHOW INDEX FROM。
- 查询系统表 information_schema.tables 中的 TIDB_PK_TYPE 列。

通过 SHOW CREATE TABLE 查看，PRIMARY KEY 的属性可能为 CLUSTERED 或 NONCLUSTERED：

```
mysql> SHOW CREATE TABLE t;
+--
    ↪ -----+----------------------------------------------------------------------------------------------------
    ↪
| Table | Create Table
    ↪
    ↪ |
+--
    ↪ -----+----------------------------------------------------------------------------------------------------
    ↪
| t     | CREATE TABLE `t` (
  `a` bigint(20) NOT NULL,
  `b` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`a`) /*T![clustered_index] CLUSTERED */
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin |
+--
    ↪ -----+----------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.01 sec)
```

通过 SHOW INDEX FROM 查看，Clustered 一列可能的结果为 Yes 或 No：

```
mysql> SHOW INDEX FROM t;
+--
    ↪ -----+------------+----------+--------------+-------------+-----------+-------------+----------+--------+
    ↪
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part
    ↪  | Packed | Null | Index_type | Comment | Index_comment | Visible | Expression |
    ↪ Clustered |
+--
    ↪ -----+------------+----------+--------------+-------------+-----------+-------------+----------+--------+
    ↪
| t     |          0 | PRIMARY  |            1 | a           | A         |           0 |     NULL
    ↪  | NULL   |      | BTREE      |         |               | YES     | NULL       | YES
    ↪       |
+--
    ↪ -----+------------+----------+--------------+-------------+-----------+-------------+----------+--------+
    ↪
1 row in set (0.01 sec)
```

查询 information_schema.tables 系统表中的 TIDB_PK_TYPE 列，可能的结果为 CLUSTERED 或 NONCLUSTERED：

```
mysql> SELECT TIDB_PK_TYPE FROM information_schema.tables WHERE table_schema = 'test' AND
    ↪ table_name = 't';
+--------------+
| TIDB_PK_TYPE |
+--------------+
| CLUSTERED    |
+--------------+
1 row in set (0.03 sec)
```

14.11.5.3 限制

目前 TiDB 的聚簇索引具有以下几类限制：

- 明确不支持且没有支持计划的使用限制：
  - 不支持与SHARD_ROW_ID_BITS 一起使用；PRE_SPLIT_REGIONS 在聚簇索引表上不生效。
  - 不支持对聚簇索引表进行降级。如需降级，请使用逻辑备份工具迁移数据。
- 尚未支持，但未来有计划支持的使用限制：
  - 尚未支持通过 ALTER TABLE 语句增加、删除、修改聚簇索引。
- 特定版本的限制：
  - 在 v5.0 版本中，聚簇索引不支持与 TiDB Binlog 一起使用。开启 TiDB Binlog 后，TiDB 只允许创建单个整数列作为主键的聚簇索引；已创建的聚簇索引表的数据插入、删除和更新动作不会通过 TiDB Binlog 同步到下游。如需同步聚簇索引表，请升级至 v5.1 版本或使用TiCDC。

开启 TiDB Binlog 之后，要创建的聚簇索引如果不是由单个整数列构成，会报以下错误：

```
mysql> CREATE TABLE t (a VARCHAR(255) PRIMARY KEY CLUSTERED);
ERROR 8200 (HY000): Cannot create clustered index table when the binlog is ON
```

与 SHARD_ROW_ID_BITS 一起使用时会报以下错误：

```
mysql> CREATE TABLE t (a VARCHAR(255) PRIMARY KEY CLUSTERED) SHARD_ROW_ID_BITS = 3;
ERROR 8200 (HY000): Unsupported shard_row_id_bits for table with primary key as row id
```

14.11.5.4 兼容性

14.11.5.4.1 升降级兼容性

TiDB 支持对聚簇索引表的升级兼容，但不支持降级兼容，即高版本 TiDB 聚簇索引表的数据在低版本 TiDB 上不可用。

聚簇索引在 TiDB v3.0 和 v4.0 中已完成部分支持，当表中存在单个整数列作为主键时默认启用，即：

- 表设置了主键
- 主键只有一列
- 主键的数据类型为整数类型

TiDB v5.0 完成了所有类型主键的支持，但默认行为与 TiDB v3.0 和 v4.0 保持一致。要修改默认行为，请设置系统变量 @@tidb_enable_clustered_index 为 ON 或 OFF。

14.11.5.4.2 MySQL 兼容性

TiDB 支持使用可执行注释的语法来包裹 CLUSTERED 或 NONCLUSTERED 关键字，且 SHOW CREATE TABLE 的结果均包含 TiDB 特有的可执行注释，这些注释在 MySQL 或低版本的 TiDB 中会被忽略。

14.11.5.4.3 TiDB 数据迁移工具兼容性

聚簇索引仅与 v5.0 及以后版本的以下数据迁移工具兼容：

- 备份与恢复工具 BR、 Dumpling、 TiDB Lightning。
- 数据迁移和同步工具 DM、 TiCDC。

v5.0 的 BR 不能通过备份恢复将非聚簇索引表转换成聚簇索引表，反之亦然。

14.11.5.4.4 与 TiDB 其他特性的兼容性

在非单整数列作为主键的表中，从非聚簇索引变为聚簇索引之后，在 v5.0 之前版本的 TiDB 能够执行的 SPLIT TABLE BY/BETWEEN 语句在 v5.0 及以后版本的 TiDB 上不再可用，原因是行数据键的构成发生了变化。在聚簇索引表上执行 SPLIT TABLE BY/BETWEEN 时需要依据主键列指定值，而不是指定一个整数值。例如：

```
mysql> create table t (a int, b varchar(255), primary key(a, b) clustered);
Query OK, 0 rows affected (0.01 sec)

mysql> split table t between (0) and (1000000) regions 5;
ERROR 1105 (HY000): Split table region lower value count should be 2

mysql> split table t by (0), (50000), (100000);
ERROR 1136 (21S01): Column count doesn't match value count at row 0

mysql> split table t between (0, 'aaa') and (1000000, 'zzz') regions 5;
+--------------------+----------------------+
| TOTAL_SPLIT_REGION | SCATTER_FINISH_RATIO |
+--------------------+----------------------+
|                  4 |                    1 |
+--------------------+----------------------+
1 row in set (0.00 sec)

mysql> split table t by (0, ''), (50000, ''), (100000, '');
+--------------------+----------------------+
| TOTAL_SPLIT_REGION | SCATTER_FINISH_RATIO |
+--------------------+----------------------+
|                  3 |                    1 |
+--------------------+----------------------+
1 row in set (0.01 sec)
```

AUTO_RANDOM 属性只能在聚簇索引表上使用。在非聚簇索引上使用 AUTO_RANDOM 会报以下错误：

```
mysql> create table t (a bigint primary key nonclustered auto_random);
ERROR 8216 (HY000): Invalid auto random: column a is not the integer primary key, or the primary
    ↪ key is nonclustered
```

14.11.6 约束

TiDB 支持的约束与 MySQL 的基本相同。

14.11.6.1 非空约束

TiDB 支持的非空约束规则与 MySQL 支持的一致。例如：

```
CREATE TABLE users (
 id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
 age INT NOT NULL,
 last_login TIMESTAMP
);
```

```
INSERT INTO users (id,age,last_login) VALUES (NULL,123,NOW());
```

```
Query OK, 1 row affected (0.02 sec)
```

```
INSERT INTO users (id,age,last_login) VALUES (NULL,NULL,NOW());
```

```
ERROR 1048 (23000): Column 'age' cannot be null
```

```
INSERT INTO users (id,age,last_login) VALUES (NULL,123,NULL);
```

```
Query OK, 1 row affected (0.03 sec)
```

- 第一条 INSERT 语句成功，因为对于定义为 AUTO_INCREMENT 的列，允许 NULL 作为其特殊值。TiDB 将为其分配下一个自动值。
- 第二条 INSERT 语句失败，因为 age 列被定义为 NOT NULL。
- 第三条 INSERT 语句成功，因为 last_login 列没有被明确地指定为 NOT NULL。默认允许 NULL 值。

14.11.6.2 CHECK 约束

TiDB 会解析并忽略 CHECK 约束。该行为与 MySQL 5.7 的相兼容。

示例如下：

```
DROP TABLE IF EXISTS users;
CREATE TABLE users (
 id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
 username VARCHAR(60) NOT NULL,
 UNIQUE KEY (username),
 CONSTRAINT min_username_length CHECK (CHARACTER_LENGTH(username) >=4)
);
INSERT INTO users (username) VALUES ('a');
SELECT * FROM users;
```

14.11.6.3 唯一约束

唯一约束是指唯一索引和主键列中所有的非空值都是唯一的。

14.11.6.3.1 乐观事务

在 TiDB 的乐观事务中，默认会对唯一约束进行惰性检查。通过在事务提交时再进行批量检查，TiDB 能够减少网络开销、提升性能。例如：

```
DROP TABLE IF EXISTS users;
CREATE TABLE users (
 id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
 username VARCHAR(60) NOT NULL,
 UNIQUE KEY (username)
);
INSERT INTO users (username) VALUES ('dave'), ('sarah'), ('bill');
```

乐观事务模式下且 tidb_constraint_check_in_place=OFF：

```
BEGIN OPTIMISTIC;
INSERT INTO users (username) VALUES ('jane'), ('chris'), ('bill');
```

```
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
INSERT INTO users (username) VALUES ('steve'),('elizabeth');
```

```
Query OK, 2 rows affected (0.00 sec)
Records: 2  Duplicates: 0  Warnings: 0
```

```
COMMIT;
```

```
ERROR 1062 (23000): Duplicate entry 'bill' for key 'users.username'
```

在以上乐观事务的示例中，唯一约束的检查推迟到事务提交时才进行。由于 bill 值已经存在，这一行为导致了重复键错误。

你可通过设置tidb_constraint_check_in_place 为 ON 停用此行为（该变量仅适用于乐观事务，悲观事务需通过 tidb_constraint_check_in_place_pessimistic 设置）。当 tidb_constraint_check_in_place 设置为 ON 时，TiDB 会在执行语句时就对唯一约束进行检查。例如：

```
DROP TABLE IF EXISTS users;
CREATE TABLE users (
 id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
 username VARCHAR(60) NOT NULL,
 UNIQUE KEY (username)
);
INSERT INTO users (username) VALUES ('dave'), ('sarah'), ('bill');
```

```
SET tidb_constraint_check_in_place = ON;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
BEGIN OPTIMISTIC;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
INSERT INTO users (username) VALUES ('jane'), ('chris'), ('bill');
```

```
ERROR 1062 (23000): Duplicate entry 'bill' for key 'users.username'
```

第一条 INSERT 语句导致了重复键错误。这会造成额外的网络通信开销，并可能降低插入操作的吞吐量。

14.11.6.3.2 悲观事务

在 TiDB 的悲观事务中，默认在执行任何一条需要插入或更新唯一索引的 SQL 语句时都会进行唯一约束检查：

```
DROP TABLE IF EXISTS users;
CREATE TABLE users (
 id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
 username VARCHAR(60) NOT NULL,
 UNIQUE KEY (username)
);
INSERT INTO users (username) VALUES ('dave'), ('sarah'), ('bill');

BEGIN PESSIMISTIC;
INSERT INTO users (username) VALUES ('jane'), ('chris'), ('bill');
```

```
ERROR 1062 (23000): Duplicate entry 'bill' for key 'users.username'
```

对于悲观事务，你可以设置变量tidb_constraint_check_in_place_pessimistic 为 OFF 来推迟唯一约束检查，到下一次对该唯一索引项加锁时或事务提交时再进行检查，同时也跳过对该悲观锁加锁，以获得更好的性能。此时需要注意：

- 由于推迟了唯一约束检查，TiDB 可能会读取到不满足唯一约束的结果，执行 COMMIT 语句时可能返回 Duplicate entry 错误。返回该错误时，TiDB 会回滚当前事务。

  下面这个例子跳过了对 bill 的加锁，因此 TiDB 可能读到不满足唯一性约束的结果：

  ```
  SET tidb_constraint_check_in_place_pessimistic = OFF;
  BEGIN PESSIMISTIC;
  INSERT INTO users (username) VALUES ('jane'), ('chris'), ('bill'); -- Query OK, 3 rows
      ↪ affected
  SELECT * FROM users FOR UPDATE;
  ```

  TiDB 读到了不满足唯一性约束的结果：有两个 bill。

  ```
  +----+----------+
  | id | username |
  +----+----------+
  | 1  | dave     |
  | 2  | sarah    |
  ```

```
| 3  | bill     |
| 7  | jane     |
| 8  | chris    |
| 9  | bill     |
+----+----------+
```

此时，如果提交事务，TiDB 将进行唯一约束检查，报出 Duplicate entry 错误并回滚事务。

```
COMMIT;
```

```
ERROR 1062 (23000): Duplicate entry 'bill' for key 'users.username'
```

- 关闭该变量时，如果在事务中写入数据，执行 COMMIT 语句可能会返回 Write conflict 错误。返回该错误时，TiDB 会回滚当前事务。

  在下面这个例子中，当有并发事务写入时，跳过悲观锁导致事务提交时报出 Write conflict 错误并回滚。

```
DROP TABLE IF EXISTS users;
CREATE TABLE users (
id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
username VARCHAR(60) NOT NULL,
UNIQUE KEY (username)
);

SET tidb_constraint_check_in_place_pessimistic = OFF;
BEGIN PESSIMISTIC;
INSERT INTO users (username) VALUES ('jane'), ('chris'), ('bill'); -- Query OK, 3 rows
    ↪ affected
```

然后另一个会话中写入了 bill：

```
INSERT INTO users (username) VALUES ('bill'); -- Query OK, 1 row affected
```

在第一个会话中提交时，TiDB 会报出 Write conflict 错误。

```
COMMIT;
```

```
ERROR 9007 (HY000): Write conflict, txnStartTS=435688780611190794, conflictStartTS
    ↪ =435688783311536129, conflictCommitTS=435688783311536130, key={tableID=74, indexID
    ↪ =1, indexValues={bill, }} primary={tableID=74, indexID=1, indexValues={bill, }},
    ↪ reason=LazyUniquenessCheck [try again later]
```

- 关闭该变量时，如果多个悲观事务之间存在写冲突，悲观锁可能会在其它悲观事务提交时被强制回滚，因此产生 PessimisticLockNotFound 错误。发生该错误时，说明该业务不适合推迟悲观事务的唯一约束检查，应考虑调整业务避免冲突，或在发生错误后重试事务。

- 关闭该变量会导致悲观事务中可能报出错误 8147：LazyUniquenessCheckFailure。

> **注意：**
> 返回 8147 错误时当前事务回滚。

下面的例子在 INSERT 语句执行时跳过了一次加锁后，在 DELETE 语句执行时对该唯一索引加锁并检查，即会在该语句报错：

```
SET tidb_constraint_check_in_place_pessimistic = OFF;
BEGIN PESSIMISTIC;
INSERT INTO users (username) VALUES ('jane'), ('chris'), ('bill'); -- Query OK, 3 rows
    ↪ affected
DELETE FROM users where username = 'bill';
```

```
ERROR 8147 (23000): transaction aborted because lazy uniqueness check is enabled and an
    ↪ error occurred: [kv:1062]Duplicate entry 'bill' for key 'users.username'
```

- 关闭该变量时，1062 Duplicate entry 报错不一定是当前执行的 SQL 语句所发生的错误。因此，在一个事务操作多个表，且这些表有同名索引时，请注意 1062 报错信息中提示的是哪个表的哪个索引发生了错误。

14.11.6.4 主键约束

与 MySQL 行为一样，主键约束包含了唯一约束，即创建了主键约束相当于拥有了唯一约束。此外，TiDB 其他的主键约束规则也与 MySQL 相似。例如：

```
CREATE TABLE t1 (a INT NOT NULL PRIMARY KEY);
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
CREATE TABLE t2 (a INT NULL PRIMARY KEY);
```

```
ERROR 1171 (42000): All parts of a PRIMARY KEY must be NOT NULL; if you need NULL in a key, use
    ↪ UNIQUE instead
```

```
CREATE TABLE t3 (a INT NOT NULL PRIMARY KEY, b INT NOT NULL PRIMARY KEY);
```

```
ERROR 1068 (42000): Multiple primary key defined
```

```
CREATE TABLE t4 (a INT NOT NULL, b INT NOT NULL, PRIMARY KEY (a,b));
```

```
Query OK, 0 rows affected (0.10 sec)
```

分析：

- 表 t2 创建失败，因为定义为主键的列 a 不能允许 NULL 值。
- 表 t3 创建失败，因为一张表只能有一个主键。
- 表 t4 创建成功，因为虽然只能有一个主键，但 TiDB 支持定义一个多列组合作为复合主键。

除上述规则外，TiDB 目前仅支持对 NONCLUSTERED 的主键进行添加和删除操作。例如：

```
CREATE TABLE t5 (a INT NOT NULL, b INT NOT NULL, PRIMARY KEY (a,b) CLUSTERED);
ALTER TABLE t5 DROP PRIMARY KEY;
```

```
ERROR 8200 (HY000): Unsupported drop primary key when the table is using clustered index
```

```
CREATE TABLE t5 (a INT NOT NULL, b INT NOT NULL, PRIMARY KEY (a,b) NONCLUSTERED);
ALTER TABLE t5 DROP PRIMARY KEY;
```

```
Query OK, 0 rows affected (0.10 sec)
```

要了解关于 CLUSTERED 主键的详细信息，请参考聚簇索引。

14.11.6.5 外键约束

> **注意：**
>
> TiDB 从 v6.6.0 开始支持外键约束。在 v6.6.0 之前，TiDB 支持创建和删除外键约束，但外键约束并不生效。升级到 v6.6.0 后，可以先删除不生效的外键后再创建外键使外键约束生效。

TiDB 支持创建外键约束。例如：

```
CREATE TABLE users (
 id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
 doc JSON
);
CREATE TABLE orders (
 id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
 user_id INT NOT NULL,
 doc JSON,
 FOREIGN KEY fk_user_id (user_id) REFERENCES users(id)
);
```

```
SELECT table_name, column_name, constraint_name, referenced_table_name, referenced_column_name
FROM information_schema.key_column_usage WHERE table_name IN ('users', 'orders');
```

```
+------------+-------------+-----------------+-----------------------+------------------------+
| table_name | column_name | constraint_name | referenced_table_name | referenced_column_name |
```

```
+------------+-------------+-----------------+----------------------+------------------------+
| users      | id          | PRIMARY         | NULL                 | NULL                   |
| orders     | id          | PRIMARY         | NULL                 | NULL                   |
| orders     | user_id     | fk_user_id      | users                | id                     |
+------------+-------------+-----------------+----------------------+------------------------+
3 rows in set (0.00 sec)
```

TiDB 也支持使用 ALTER TABLE 命令来删除外键 (DROP FOREIGN KEY) 和添加外键 (ADD FOREIGN KEY)：

```
ALTER TABLE orders DROP FOREIGN KEY fk_user_id;
ALTER TABLE orders ADD FOREIGN KEY fk_user_id (user_id) REFERENCES users(id);
```

14.11.7 生成列

> **警告：**
>
> 当前该功能为实验特性，不建议在生产环境中使用。

本文介绍生成列的概念以及用法。

14.11.7.1 生成列的基本概念

与一般的列不同，生成列的值由列定义中表达式计算得到。对生成列进行插入或更新操作时，并不能对之赋值，只能使用 DEFAULT。

生成列包括存储生成列和虚拟生成列。存储生成列会将计算得到的值存储起来，在读取时不需要重新计算。虚拟生成列不会存储其值，在读取时会重新计算。存储生成列和虚拟生成列相比，前者在读取时性能更好，但是要占用更多的磁盘空间。

无论是存储生成列还是虚拟列，都可以在其上面建立索引。

14.11.7.2 生成列的应用

生成列的主要的作用之一：从 JSON 数据类型中解出数据，并为该数据建立索引。

MySQL 5.7 及 TiDB 都不能直接为 JSON 类型的列添加索引，即不支持在如下表结构中的 address_info 上建立索引：

```
CREATE TABLE person (
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    address_info JSON,
    KEY (address_info)
);
```

如果要为 JSON 列某个字段添加索引，可以抽取该字段为生成列。

以 city 这一 address_info 中的字段为例，可以为其建立一个虚拟生成列并添加索引：

```
CREATE TABLE person (
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    address_info JSON,
    city VARCHAR(64) AS (JSON_UNQUOTE(JSON_EXTRACT(address_info, '$.city'))), -- 虚拟生成列
    -- city VARCHAR(64) AS (JSON_UNQUOTE(JSON_EXTRACT(address_info, '$.city'))) VIRTUAL, --
        ↪ 虚拟生成列
    -- city VARCHAR(64) AS (JSON_UNQUOTE(JSON_EXTRACT(address_info, '$.city'))) STORED, --
        ↪ 存储生成列
    KEY (city)
);
```

该表中，city 列是一个虚拟生成列。并且在该列上建立了索引。以下语句能够利用索引加速语句的执行速度：

```
SELECT name, id FROM person WHERE city = 'Beijing';
```

```
EXPLAIN SELECT name, id FROM person WHERE city = 'Beijing';
```

```
+---------------------------------+---------+-----------+---------------------------------+-----------------------
    ↪
| id                              | estRows | task      | access object                   |
    ↪ operator info                                       |
+---------------------------------+---------+-----------+---------------------------------+-----------------------
    ↪
| Projection_4                    | 10.00   | root      |                                 | test.
    ↪ person.name, test.person.id                         |
| └─IndexLookUp_10                | 10.00   | root      |                                 |
    ↪                                                     |
|   ├─IndexRangeScan_8(Build)     | 10.00   | cop[tikv] | table:person, index:city(city) | range
    ↪ :["Beijing","Beijing"], keep order:false, stats:pseudo |
|   └─TableRowIDScan_9(Probe)     | 10.00   | cop[tikv] | table:person                    | keep
    ↪ order:false, stats:pseudo                           |
+---------------------------------+---------+-----------+---------------------------------+-----------------------
    ↪
```

从执行计划中，可以看出使用了 city 这个索引来读取满足 city = 'Beijing' 这个条件的行的 HANDLE，再用这个 HANDLE 来读取该行的数据。

如果 $.city 路径中无数据，则 JSON_EXTRACT 返回 NULL。如果想增加约束，city 列必须是 NOT NULL，则可按照以下方式定义虚拟生成列：

```
CREATE TABLE person (
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
```

```
    address_info JSON,
    city VARCHAR(64) AS (JSON_UNQUOTE(JSON_EXTRACT(address_info, '$.city'))) NOT NULL,
    KEY (city)
);
```

14.11.7.3 生成列在 INSERT 和 UPDATE 语句中的行为

INSERT 和 UPDATE 语句都会检查生成列计算得到的值是否满足生成列的定义。未通过有效性检测的行会返回错误：

```
INSERT INTO person (name, address_info) VALUES ('Morgan', JSON_OBJECT('Country', 'Canada'));
```

```
ERROR 1048 (23000): Column 'city' cannot be null
```

14.11.7.4 索引生成列替换

当查询中出现的某个表达式与一个含索引的生成列严格同等时，TiDB 会将这个表达式替换为对应的生成列，这样就可以在生成查询计划时考虑使用这个索引。

下面的例子为 a+1 这个表达式创建生成列并添加索引，从而加速查询。其中，a 的列类型是 int，而 a+1 的列类型是 bigint。如果将生成列的类型改为 int，就不会发生替换。关于类型转换的规则，可以参见表达式求值的类型转换。

```
create table t(a int);
desc select a+1 from t where a+1=3;
```

```
+--
    ↪ -------------------------+----------+-----------+---------------+-------------------------------+
    ↪
| id                        | estRows  | task      | access object | operator info
    ↪                  |
+--
    ↪ -------------------------+----------+-----------+---------------+-------------------------------+
    ↪
| Projection_4              | 8000.00  | root      |               | plus(test.t.a, 1)->Column#3
    ↪    |
| └─TableReader_7           | 8000.00  | root      |               | data:Selection_6
    ↪               |
|   └─Selection_6           | 8000.00  | cop[tikv] |               | eq(plus(test.t.a, 1), 3)
    ↪       |
|     └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo |
+--
    ↪ -------------------------+----------+-----------+---------------+-------------------------------+
    ↪
4 rows in set (0.00 sec)
```

```
alter table t add column b bigint as (a+1) virtual;
alter table t add index idx_b(b);
desc select a+1 from t where a+1=3;
```

```
+--
    ↪ ---------------------+---------+-----------+-----------------------+-------------------------------
    ↪
| id                    | estRows | task      | access object           | operator info
    ↪                               |
+--
    ↪ ---------------------+---------+-----------+-----------------------+-------------------------------
    ↪
| IndexReader_6           | 10.00   | root      |                         | index:IndexRangeScan_5
    ↪                         |
| └─IndexRangeScan_5      | 10.00   | cop[tikv] | table:t, index:idx_b(b) | range:[3,3], keep
    ↪ order:false, stats:pseudo |
+--
    ↪ ---------------------+---------+-----------+-----------------------+-------------------------------
    ↪
2 rows in set (0.01 sec)
```

> **注意：**
>
> 若待替换的表达式类型和生成列类型都是字符类型，但两种类型长度不同时，仍可通过将系统变量tidb_enable_unsafe_substitute 设置为 ON 来允许其替换。配置该系统变量时，需要保证生成列计算得到的值严格满足生成列的定义，否则，可能因为长度不同，导致数据截断得到错误的结果。详情见 GitHub issue #35490。

14.11.7.5 生成列的局限性

目前生成列有以下局限性：

- 不能通过 ALTER TABLE 增加存储生成列；
- 不能通过 ALTER TABLE 将存储生成列转换为普通列，也不能将普通列转换成存储生成列；
- 不能通过 ALTER TABLE 修改存储生成列的生成列表达式；
- 并未支持所有的JSON 函数；
- 目前仅当生成列是虚拟生成列时索引生成列替换规则有效，暂不支持将表达式替换为存储生成列，但仍然可以通过直接使用该生成列本身来使用索引。

14.11.8 SQL 模式

TiDB 服务器采用不同 SQL 模式来操作，且不同客户端可以应用不同模式。SQL 模式定义 TiDB 支持哪些 SQL 语法及执行哪种数据验证检查。

TiDB 启动之后采用 SET [SESSION | GLOBAL] sql_mode='modes' 设置 SQL 模式。设置 GLOBAL 级别的 SQL 模式时用户需要有 SUPER 权限，并且只会影响到从设置 SQL 模式开始后续新建立的连接（注：老连接不受影响）。SESSION 级别的 SQL 模式的变化只会影响当前的客户端。

Modes 是用逗号（','）间隔开的一系列不同的模式。使用 SELECT @@sql_mode 语句查询当前 SQL 模式，SQL 模式默认值：ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,
↪ ERROR_FOR_DIVISION_BY_ZERO,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION。

14.11.8.1 重要的 sql_mode 值

- ANSI：符合标准 SQL，对数据进行校验，如果不符合定义类型或长度，对数据类型调整或截断保存，且返回 warning 警告。
- STRICT_TRANS_TABLES：严格模式，对数据进行严格校验，当数据出现错误时，无法被插入到表中，并且返回错误。
- TRADITIONAL：采用此模式使 TiDB 的行为像“传统”SQL 数据库系统，当在列中插入不正确的值时“给出错误而不是警告”，一旦发现错误立即放弃 INSERT 或 UPDATE。

14.11.8.2 SQL mode 列表，如下

名称	含义
PIPES_AS_CONCAT	将“||”视为字符串连接操作符（+）(同 CONCAT())，而不视为 OR（支持)

名称	含义
ANSI_QUOTES	将"视为识别符，如果启用ANSI_QUOTES，只单引号内的会被认为是String Liter-als，双引号被解释为识别符，因此不能用双引号来引用字符串（支持）

名称	含义
IGNORE_SPACE	若开启该模式，系统忽略空格。例如：“user”和“user “是相同的（支持）

名称	含义
ONLY_FULL_GROUP_BY	如果 GROUP BY 出现的列并没有在 SELECT，HAVING，ORDER BY 中出现，此 SQL 不合法，因为不在 GROUP BY 中的列被查询展示出来不符合正常现象 (支持)

名称	含义
NO_UNSIGNED_SUBTRACTION	在减运算中，如果某个操作数没有符号，不要将结果标记为 UNSIGNED（支持）
NO_DIR_IN_CREATE	创建表时，忽视所有 INDEX DIRECTORY 和 DATA DIRECTORY 指令，该选项仅对从复制服务器有用（仅语法支持）

名称	含义
NO_KEY_OPTIONS	使用 SHOW CREATE TABLE 时不会输出 MySQL 特有的语法部分，如 ENGINE，使用 mysqldump 跨 DB 种类迁移的时需要考虑此选项（仅语法支持）

名称	含义
NO_FIELD_OPTIONS	使用 SHOW CREATE TABLE 时不会输出 MySQL 特有的语法部分，如 ENGINE，使用 mysqldump 跨 DB 种类迁移的时需要考虑此选项（仅语法支持）

名称	含义
NO_TABLE_OPTIONS	使用 SHOW CRE-ATE TABLE 时不会输出 MySQL 特有的语法部分，如 EN-GINE，使用 mysql-dump 跨 DB 种类迁移的时需要考虑此选项（仅语法支持）

名称	含义
NO_AUTO_VALUE_ON_ZERO	若启用该模式，在 AUTO_INCREMENT 列的处理传入的值是 0 或者具体数值时系统直接将该值写入此列，传入 NULL 时系统自动生成下一个序列号（支持）
NO_BACKSLASH_ESCAPES	若启用该模式，\ 反斜杠符号仅代表它自己（支持）

名称	含义
STRICT_TRANS_TABLES	对于事务存储引擎启用严格模式，insert非法值之后，回滚整条语句（支持）
STRICT_ALL_TABLES	对于事务型表，写入非法值之后，回滚整个事务语句（支持）

名称	含义
NO_ZERO_IN_DATE	在严格模式，不接受月或日部分为 0 的日期。如果使用 IGNORE 选项，我们为类似的日期插入’0000-00-00’。在非严格模式，可以接受该日期，但会生成警告（支持）

名称	含义
NO_ZERO_DATE	在严格模式，不要将’0000-00-00’做为合法日期。你仍然可以用 IGNORE 选项插入零日期。在非严格模式，可以接受该日期，但会生成警告（支持）

名称	含义
ALLOW_INVALID_DATES	不检查全部日期的合法性，仅检查月份值在 1 到 12 及日期值在 1 到 31 之间，仅适用于 DATE 和 DATA-TIME 列，TIMES-TAMP 列需要全部检查其合法性（支持）

名称	含义
ERROR_FOR_DIVISION_BY_ZERO	若启用该模式，在 INSERT 或 UPDATE 过程中，被除数为 0 值时，系统产生错误若未启用该模式，被除数为 0 值时，系统产生警告，并用 NULL 代替（支持）

名称	含义
NO_AUTO_CREATE_USER	防止 GRANT 自动创建新用户，但指定密码除外（支持）

名称	含义
HIGH_NOT_PRECEDENCE	NOT 操作符的优先级是表达式。例如：NOT a BETWEEN b AND c 被解释为 NOT (a BETWEEN b AND c)。在部份旧版本 MySQL 中，表达式被解释为 (NOT a) BETWEEN b AND c（支持）

名称	含义
NO_ENGINE_SUBSTITUTION	如果需要的存储引擎被禁用或未编译，可以防止自动替换存储引擎（仅语法支持）
PAD_CHAR_TO_FULL_LENGTH	若启用该模式，系统对于 CHAR 类型不会截断尾部空格（仅语法支持。该模式在 MySQL 8.0 中已废弃。）

名称	含义
REAL_AS_FLOAT	将 REAL 视为 FLOAT 的同义词，而不是 DOUBLE 的同义词（支持）
POSTGRESQL	等同于 PIPES_AS_CONCAT、ANSI_QUOTES、IGNORE_SPACE、NO_KEY_OPTIONS、NO_TABLE_OPTIONS、NO_FIELD_OPTIONS（仅语法支持）
MSSQL	等同于 PIPES_AS_CONCAT、ANSI_QUOTES、IGNORE_SPACE、NO_KEY_OPTIONS、NO_TABLE_OPTIONS、NO_FIELD_OPTIONS（仅语法支持）

名称	含义
DB2	等同于 PIPES_AS_CONCAT、ANSI_QUOTES、IGNORE_SPACE、NO_KEY_OPTIONS、NO_TABLE_OPTIONS、NO_FIELD_OPTIONS（仅语法支持）
MAXDB	等同于 PIPES_AS_CONCAT、ANSI_QUOTES、IGNORE_SPACE、NO_KEY_OPTIONS、NO_TABLE_OPTIONS、NO_FIELD_OPTIONS、NO_AUTO_CREATE_USER（支持）
MySQL32	等同于 NO_FIELD_OPTIONS、HIGH_NOT_PRECEDENCE（仅语法支持）
MYSQL40	等同于 NO_FIELD_OPTIONS、HIGH_NOT_PRECEDENCE（仅语法支持）

名称	含义
ANSI	等同于 REAL_AS_FLOAT、PIPES_AS_CONCAT、ANSI_QUOTES、IGNORE_SPACE (仅语法支持)
TRADITIONAL	等同于 STRICT_TRANS_TABLES、STRICT_ALL_TABLES、NO_ZERO_IN_DATE、NO_ZERO_DATE、ERROR_FOR_DIVISION_BY_ZERO、NO_AUTO_CREATE_USER (仅语法支持)
ORACLE	等同于 PIPES_AS_CONCAT、ANSI_QUOTES、IGNORE_SPACE、NO_KEY_OPTIONS、NO_TABLE_OPTIONS、NO_FIELD_OPTIONS、NO_AUTO_CREATE_USER (仅语法支持)

14.11.9 表属性

表属性是 TiDB 从 5.3.0 版本开始引入的新特性，用于为表或分区添加特定的属性，以对表或分区执行相应属性对应的操作，例如可以利用表属性控制 Region 的合并。

注意：

- 目前 TiDB 仅支持为表或分区添加 merge_option 属性，用于控制 Region 合并。该属性仅能处理部分热点问题。如需了解更多的热点问题处理相关内容，请参阅TiDB 热点问题处理。
- 当使用 TiDB Binlog 或 TiCDC 进行同步或者使用 BR 进行增量备份时，同步和备份会跳过设置表属性的 DDL 语句。如需在下游或者备份集群使用表属性，需要在下游或者备份集群手动执行该 DDL 语句以设置表属性。

14.11.9.1 使用方法

表属性为 key=value 的形式，多个属性需要用逗号分隔。具体示例如下，其中 t 为所要修改的表名，p 为所要修改的分区名，[]内部为可选项。

- 设置表或分区属性

```
ALTER TABLE t [PARTITION p] ATTRIBUTES [=] 'key=value[, key1=value1...]';
```

- 重置表或分区属性

```
ALTER TABLE t [PARTITION p] ATTRIBUTES [=] DEFAULT;
```

- 查看全部表及分区属性

```
SELECT * FROM information_schema.attributes;
```

- 查看某一张表或分区配置的属性

```
SELECT * FROM information_schema.attributes WHERE id='schema/t[/p]';
```

- 查看拥有某属性的所有表及分区

```
SELECT * FROM information_schema.attributes WHERE attributes LIKE '%key%';
```

14.11.9.2 覆盖关系

为分区表配置的属性会对表的所有分区生效。一种例外情况是，如果分区表和分区都配置了相同属性但属性值不同，分区属性将覆盖分区表属性。例如，当分区表 t 配置属性 key=value，同时分区 p 配置属性 key=value1 时：

```
ALTER TABLE t ATTRIBUTES[=]'key=value';
ALTER TABLE t PARTITION p ATTRIBUTES[=]'key=value1';
```

分区 p 实际生效的属性为 key=value1。

14.11.9.3 使用表属性控制 Region 合并

14.11.9.3.1 使用场景

当写入或读取数据存在热点时，可以使用表属性控制 Region 合并，通过为表或分区添加 `merge_option` 属性，将其设置为 deny 来解决。以下介绍了两种使用场景。

新建表或分区的写入热点问题

在对某张新建表或某个新建分区写入数据存在热点问题时，通常需要使用分裂打散 Region 的操作避免写入热点，但由于新建表或分区的分裂操作实际产生的是空 Region，如果分裂打散操作距离写入存在一定时间间隔，则 Region 可能会被合并，从而导致无法真正规避写入热点问题。此时可以为表或分区添加 `merge_option` 属性，设置为 deny 来解决问题。

只读场景下周期性读热点问题

在只读场景下，如果是通过手动分裂 Region 缓解某张表或分区的周期性读热点问题，且不希望热点消失后手动分裂的 Region 被合并。此时可以为表或分区添加 `merge_option` 属性，设置为 deny 来解决问题。

14.11.9.3.2 使用方法

使用方法如下，其中 `t` 为所要修改的表名，`p` 为所要修改的分区名。

- 禁止属于某个表的 Region 被合并

```
ALTER TABLE t ATTRIBUTES 'merge_option=deny';
```

- 允许属于某个表的 Region 被合并

```
ALTER TABLE t ATTRIBUTES 'merge_option=allow';
```

- 重置某个表的属性

```
ALTER TABLE t ATTRIBUTES DEFAULT;
```

- 禁止属于某个分区的 Region 被合并

```
ALTER TABLE t PARTITION p ATTRIBUTES 'merge_option=deny';
```

- 允许属于某个分区的 Region 被合并

```
ALTER TABLE t PARTITION p attributes 'merge_option=allow';
```

- 查看所有配置了 `merge_option` 属性的表或分区

```
SELECT * FROM information_schema.attributes WHERE attributes LIKE '%merge_option%';
```

14.11.9.3.3 覆盖关系

```
ALTER TABLE t ATTRIBUTES 'merge_option=deny';
ALTER TABLE t PARTITION p ATTRIBUTES 'merge_option=allow';
```

同时配置上述两个属性时，实际分区 p 的 Region 可以被合并。当分区的属性被重置时，分区 p 则会继承表 t 的属性，Region 无法被合并。

> 注意：
>
> - 如果目前只存在分区表的属性，即使配置 merge_option=allow，分区也会默认按照实际分区数量切分成多个 Region。如需合并所有 Region，则需要重置该分区表的属性。
> - 使用该属性需要注意 PD 的参数split-merge-interval 的配置。如果没有配置 merge_option，Region 在超过 split-merge-interval 指定的时间后满足条件即可合并。如果配置了 merge_option，则超过指定时间后会根据 merge_option 的配置情况再决定是否可以合并。

14.11.10 事务

14.11.10.1 TiDB 事务概览

TiDB 支持分布式事务，提供乐观事务与悲观事务两种事务模式。TiDB 3.0.8 及以后版本，TiDB 默认采用悲观事务模式。

本文主要介绍涉及事务的常用语句、显式/隐式事务、事务的隔离级别和惰性检查，以及事务大小的限制。

常用的变量包括autocommit、 tidb_disable_txn_auto_retry、 tidb_retry_limit 以及tidb_txn_mode。

> 注意：
>
> 变量tidb_disable_txn_auto_retry 和tidb_retry_limit 仅适用于乐观事务，不适用于悲观事务。

14.11.10.1.1 常用事务语句

开启事务

要显式地开启一个新事务，既可以使用BEGIN 语句，也可以使用START TRANSACTION 语句，两者效果相同。

语法：

```
BEGIN;
```

```
START TRANSACTION;
```

```
START TRANSACTION WITH CONSISTENT SNAPSHOT;
```

```
START TRANSACTION WITH CAUSAL CONSISTENCY ONLY;
```

如果执行以上语句时，当前 Session 正处于一个事务的中间过程，那么系统会先自动提交当前事务，再开启一个新的事务。

> 注意：
>
> 与 MySQL 不同的是，TiDB 在执行完上述语句后即会获取当前数据库快照，而 MySQL 的 BEGIN 和 START TRANSACTION 是在开启事务后的第一个从 InnoDB 读数据的 SELECT 语句（非 SELECT FOR UPDATE）后获取快照，START TRANSACTION WITH CONSISTENT SNAPSHOT 是语句执行时获取快照。因此，TiDB 中的 BEGIN、START TRANSACTION 和 START TRANSACTION WITH CONSISTENT SNAPSHOT 都等效为 MySQL 中的 START TRANSACTION WITH CONSISTENT SNAPSHOT。

提交事务

COMMIT 语句用于提交 TiDB 在当前事务中进行的所有修改。

语法：

```
COMMIT;
```

> 建议：
>
> 启用乐观事务前，请确保应用程序可正确处理 COMMIT 语句可能返回的错误。如果不确定应用程序将会如何处理，建议改为使用悲观事务。

回滚事务

ROLLBACK 语句用于回滚并撤销当前事务的所有修改。

语法：

```
ROLLBACK;
```

如果客户端连接中止或关闭，也会自动回滚该事务。

14.11.10.1.2 自动提交

为满足 MySQL 兼容性的要求，在默认情况下，TiDB 将在执行语句后立即进行 autocommit（自动提交）。

举例：

```
mysql> CREATE TABLE t1 (
          id INT NOT NULL PRIMARY KEY auto_increment,
          pad1 VARCHAR(100)
         );
Query OK, 0 rows affected (0.09 sec)

mysql> SELECT @@autocommit;
+--------------+
| @@autocommit |
+--------------+
| 1            |
+--------------+
1 row in set (0.00 sec)

mysql> INSERT INTO t1 VALUES (1, 'test');
Query OK, 1 row affected (0.02 sec)

mysql> ROLLBACK;
Query OK, 0 rows affected (0.01 sec)

mysql> SELECT * FROM t1;
+----+------+
| id | pad1 |
+----+------+
|  1 | test |
+----+------+
1 row in set (0.00 sec)
```

以上示例中，ROLLBACK 语句没产生任何效果。由于 INSERT 语句是在自动提交的情况下执行的，等同于以下单语句事务：

```
START TRANSACTION;
INSERT INTO t1 VALUES (1, 'test');
COMMIT;
```

如果已显式地启动事务，则不适用自动提交。以下示例，ROLLBACK 语句成功撤回了 INSERT 语句：

```
mysql> CREATE TABLE t2 (
          id INT NOT NULL PRIMARY KEY auto_increment,
          pad1 VARCHAR(100)
         );
Query OK, 0 rows affected (0.10 sec)

mysql> SELECT @@autocommit;
+--------------+
| @@autocommit |
```

```
+--------------+
| 1            |
+--------------+
1 row in set (0.00 sec)

mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO t2 VALUES (1, 'test');
Query OK, 1 row affected (0.02 sec)

mysql> ROLLBACK;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM t2;
Empty set (0.00 sec)
```

autocommit 是一个系统变量，可以基于 Session 或 Global 进行修改。

举例：

```
SET autocommit = 0;
```

```
SET GLOBAL autocommit = 0;
```

14.11.10.1.3 显式事务和隐式事务

> **注意：**
>
> 有些语句是隐式提交的。例如，执行 [BEGIN|START TRANCATION] 语句时，TiDB 会隐式提交上一个事务，并开启一个新的事务以满足 MySQL 兼容性的需求。详情参见 implicit commit。

TiDB 可以显式地使用事务（通过 [BEGIN|START TRANSACTION]/COMMIT 语句定义事务的开始和结束）或者隐式地使用事务 (SET autocommit = 1)。

在自动提交状态下，使用 [BEGIN|START TRANSACTION] 语句会显式地开启一个事务，同时也会禁用自动提交，使隐式事务变成显式事务。直到执行 COMMIT 或 ROLLBACK 语句时才会恢复到此前默认的自动提交状态。

对于 DDL 语句，会自动提交并且不能回滚。如果运行 DDL 的时候，正在一个事务的中间过程中，会先自动提交当前事务，再执行 DDL。

14.11.10.1.4 惰性检查

执行 DML 语句时，乐观事务默认不会检查主键约束或唯一约束，而是在 COMMIT 事务时进行这些检查。

举例：

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY);
INSERT INTO t1 VALUES (1);
BEGIN OPTIMISTIC;
INSERT INTO t1 VALUES (1); -- MySQL 返回错误；TiDB 返回成功。
INSERT INTO t1 VALUES (2);
COMMIT; -- MySQL 提交成功；TiDB 返回错误，事务回滚。
SELECT * FROM t1; -- MySQL 返回 1 2；TiDB 返回 1。
```

```
mysql> CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY);
Query OK, 0 rows affected (0.10 sec)

mysql> INSERT INTO t1 VALUES (1);
Query OK, 1 row affected (0.02 sec)

mysql> BEGIN OPTIMISTIC;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO t1 VALUES (1); -- MySQL 返回错误；TiDB 返回成功。
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO t1 VALUES (2);
Query OK, 1 row affected (0.00 sec)

mysql> COMMIT; -- MySQL 提交成功；TiDB 返回错误，事务回滚。
ERROR 1062 (23000): Duplicate entry '1' for key 't1.PRIMARY'
mysql> SELECT * FROM t1; -- MySQL 返回 1 2；TiDB 返回 1。
+----+
| id |
+----+
|  1 |
+----+
1 row in set (0.01 sec)
```

惰性检查优化通过批处理约束检查并减少网络通信来提升性能。可以通过设置`tidb_constraint_check_in_place ↪ = ON`禁用该行为。

> 注意：
>
> - 本优化仅适用于乐观事务。
> - 本优化仅对普通的 `INSERT` 语句生效，对 `INSERT IGNORE` 和 `INSERT ON DUPLICATE KEY ↪ UPDATE` 不会生效。

14.11.10.1.5 语句回滚

TiDB 支持语句执行失败后的原子性回滚。如果语句报错，则所做的修改将不会生效。该事务将保持打开状态，并且在发出 COMMIT 或 ROLLBACK 语句之前可以进行其他修改。

```
CREATE TABLE test (id INT NOT NULL PRIMARY KEY);
BEGIN;
INSERT INTO test VALUES (1);
INSERT INTO tset VALUES (2);  -- tset 拼写错误，使该语句执行出错。
INSERT INTO test VALUES (1),(2);  -- 违反 PRIMARY KEY 约束，语句不生效。
INSERT INTO test VALUES (3);
COMMIT;
SELECT * FROM test;
```

```
mysql> CREATE TABLE test (id INT NOT NULL PRIMARY KEY);
Query OK, 0 rows affected (0.09 sec)

mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO test VALUES (1);
Query OK, 1 row affected (0.02 sec)

mysql> INSERT INTO tset VALUES (2);  -- tset 拼写错误，使该语句执行出错。
ERROR 1146 (42S02): Table 'test.tset' doesn't exist
mysql> INSERT INTO test VALUES (1),(2);  -- 违反 PRIMARY KEY 约束，语句不生效。
ERROR 1062 (23000): Duplicate entry '1' for key 'test.PRIMARY'
mysql> INSERT INTO test VALUES (3);
Query OK, 1 row affected (0.00 sec)

mysql> COMMIT;
Query OK, 0 rows affected (0.01 sec)

mysql> SELECT * FROM test;
+----+
| id |
+----+
|  1 |
|  3 |
+----+
2 rows in set (0.00 sec)
```

以上例子中，INSERT 语句执行失败之后，事务保持打开状态。最后的 INSERT 语句执行成功，并且提交了修改。

14.11.10.1.6 事务限制

由于底层存储引擎的限制，TiDB 要求单行不超过 6 MB。可以将一行的所有列根据类型转换为字节数并加和来估算单行大小。

TiDB 同时支持乐观事务与悲观事务，其中乐观事务是悲观事务的基础。由于乐观事务是先将修改缓存在私有内存中，因此，TiDB 对于单个事务的容量做了限制。

TiDB 中，单个事务的总大小默认不超过 100 MB，这个默认值可以通过配置文件中的配置项 `txn-total-size-limit` 进行修改，最大支持 1 TB。单个事务的实际大小限制还取决于服务器剩余可用内存的大小，执行事务时 TiDB 进程的内存消耗相对于事务大小会存在一定程度的放大，最大可能达到提交事务大小的 2 到 3 倍以上。

在 4.0 以前的版本，TiDB 限制了单个事务的键值对的总数量不超过 30 万条，从 4.0 版本起 TiDB 取消了这项限制。

> 注意：
>
> 通常，用户会开启 TiDB Binlog 将数据向下游进行同步。某些场景下，用户会使用消息中间件来消费同步到下游的 binlog，例如 Kafka。
>
> 以 Kafka 为例，Kafka 的单条消息处理能力的上限是 1 GB。因此，当把 `txn-total-size-limit` 设置为 1 GB 以上时，可能出现事务在 TiDB 中执行成功，但下游 Kafka 报错的情况。为避免这种情况出现，请用户根据最终消费者的限制来决定 `txn-total-size-limit` 的实际大小。例如：下游使用了 Kafka，则 `txn-total-size-limit` 不应超过 1 GB。

14.11.10.1.7 因果一致性事务

> 注意：
>
> 因果一致性事务只在启用 Async Commit 特性和一阶段提交特性时生效。关于这两个特性的启用情况，请参见`tidb_enable_async_commit` 系统变量介绍和`tidb_enable_1pc` 系统变量介绍。

TiDB 支持开启因果一致性的事务。因果一致性的事务在提交时无需向 PD 获取时间戳，所以提交延迟更低。开启因果一致性事务的语法为：

```
START TRANSACTION WITH CAUSAL CONSISTENCY ONLY;
```

默认情况下，TiDB 保证线性一致性。在线性一致性的情况下，如果事务 2 在事务 1 提交完成后提交，逻辑上事务 2 就应该在事务 1 后发生。

因果一致性弱于线性一致性。在因果一致性的情况下，只有事务 1 和事务 2 加锁或写入的数据有交集时（即事务 1 和事务 2 存在数据库可知的因果关系时），才能保证事务的提交顺序与事务的发生顺序保持一致。目前暂不支持传入数据库外部的因果关系。

采用因果一致性的两个事务有以下特性：

- 有潜在因果关系的事务之间的逻辑顺序与物理提交顺序一致

- 无因果关系的事务之间的逻辑顺序与物理提交顺序不保证一致
- 不加锁的读取不产生因果关系

有潜在因果关系的事务之间的逻辑顺序与物理提交顺序一致

假设事务 1 和事务 2 都采用因果一致性，并先后执行如下语句：

事务 1	事务 2
START TRANSACTION WITH CAUSAL CONSISTENCY ONLY	START TRANSACTION WITH CAUSAL CONSISTENCY ONLY
x = SELECT v FROM t WHERE id = 1 FOR UPDATE	
UPDATE t set v = $(x + 1) WHERE id = 2	
COMMIT	
	UPDATE t SET v = 2 WHERE id = 1
	COMMIT

上面的例子中，事务 1 对 `id = 1` 的记录加了锁，事务 2 的事务对 `id = 1` 的记录进行了修改，所以事务 1 和事务 2 有潜在的因果关系。所以即使用因果一致性开启事务，只要事务 2 在事务 1 提交成功后才提交，逻辑上事务 2 就必定比事务 1 晚发生。因此，不存在某个事务读到了事务 2 对 `id = 1` 记录的修改，但却没有读到事务 1 对 `id = 2` 记录的修改的情况。

无因果关系的事务之间的逻辑顺序与物理提交顺序不保证一致

假设 `id = 1` 和 `id = 2` 的记录最初值都为 0，事务 1 和事务 2 都采用因果一致性，并先后执行如下语句：

事务 1	事务 2	事务 3
START TRANSAC-TION WITH CAUSAL CONSIS-TENCY ONLY	START TRANSAC-TION WITH CAUSAL CONSIS-TENCY ONLY	
UPDATE t set v = 3 WHERE id = 2		
	UPDATE t SET v = 2 WHERE id = 1	
		BEGIN
COMMIT		
	COMMIT	
		SELECT v FROM t WHERE id IN (1, 2)

在本例中，事务 1 不读取 `id = 1` 的记录。此时事务 1 和事务 2 没有数据库可知的因果关系。如果使用因果一致性开启事务，即使物理时间上事务 2 在事务 1 提交完成后才开始提交，TiDB 也不保证逻辑上事务 2 比事务 1 晚发生。

此时如果有一个事务 3 在事务 1 提交前开启，并在事务 2 提交后读取 `id = 1` 和 `id = 2` 的记录，事务 3 可能读到 `id = 1` 的值为 2 但是 `id = 2` 的值为 0。

不加锁的读取不产生因果关系

假设事务 1 和事务 2 都采用因果一致性，并先后执行如下语句：

事务 1	事务 2
START TRANSAC-TION WITH CAUSAL CONSIS-TENCY ONLY	START TRANSAC-TION WITH CAUSAL CONSIS-TENCY ONLY

事务 1	事务 2
	UPDATE t SET v = 2 WHERE id = 1
SELECT v FROM t WHERE id = 1	
UPDATE t set v = 3 WHERE id = 2	
	COMMIT
COMMIT	

如本例所示，不加锁的读取不产生因果关系。事务 1 和事务 2 产生了写偏斜的异常，如果他们有业务上的因果关系，则是不合理的。所以本例中，使用因果一致性的事务 1 和事务 2 没有确定的逻辑顺序。

14.11.10.2 TiDB 事务隔离级别

事务隔离级别是数据库事务处理的基础，ACID 中的 “I”，即 Isolation，指的就是事务的隔离性。

SQL-92 标准定义了 4 种隔离级别：读未提交 (READ UNCOMMITTED)、读已提交 (READ COMMITTED)、可重复读 (REPEATABLE READ)、串行化 (SERIALIZABLE)。详见下表：

Isolation Level	Dirty Write	Dirty Read	Fuzzy Read	Phantom
READ UNCOMMITTED	Not Possible	Possible	Possible	Possible
READ COMMITTED	Not Possible	Not possible	Possible	Possible
REPEATABLE READ	Not Possible	Not possible	Not possible	Possible
SERIALIZABLE	Not Possible	Not possible	Not possible	Not possible

TiDB 实现了快照隔离 (Snapshot Isolation, SI) 级别的一致性。为与 MySQL 保持一致，又称其为 “可重复读”。该隔离级别不同于ANSI 可重复读隔离级别和MySQL 可重复读隔离级别。

> 注意：
>
> 在 TiDB v3.0 中，事务的自动重试功能默认为禁用状态。不建议开启自动重试功能，因为可能导致事务隔离级别遭到破坏。更多关于事务自动重试的文档说明，请参考事务重试。
>
> 从 TiDB v3.0.8 版本开始，新创建的 TiDB 集群会默认使用悲观事务模式，悲观事务中的当前读 (for update 读) 为不可重复读，关于悲观事务使用注意事项，请参考悲观事务模式

14.11.10.2.1 可重复读隔离级别 (Repeatable Read)

当事务隔离级别为可重复读时，只能读到该事务启动时已经提交的其他事务修改的数据，未提交的数据或在事务启动后其他事务提交的数据是不可见的。对于本事务而言，事务语句可以看到之前的语句做出的修改。

对于运行于不同节点的事务而言，不同事务启动和提交的顺序取决于从 PD 获取时间戳的顺序。

处于可重复读隔离级别的事务不能并发的更新同一行，当事务提交时发现该行在该事务启动后，已经被另一个已提交的事务更新过，那么该事务会回滚。示例如下：

```
create table t1(id int);
insert into t1 values(0);

start transaction;              |               start transaction;
select * from t1;               |               select * from t1;
update t1 set id=id+1;          |               update t1 set id=id+1; -- 如果使用悲观事务，
    ↪ 则后一个执行的 update 语句会等锁，直到持有锁的事务提交或者回滚释放行锁。
commit;                         |
                                |               commit; -- 事务提交失败，回滚。如果使用悲观事务，
                                    ↪ 可以提交成功。
```

与 ANSI 可重复读隔离级别的区别

尽管名称是可重复读隔离级别，但是 TiDB 中可重复读隔离级别和 ANSI 可重复隔离级别是不同的。按照 A Critique of ANSI SQL Isolation Levels 论文中的标准，TiDB 实现的是论文中的快照隔离级别。该隔离级别不会出现狭义上的幻读 (A3)，但不会阻止广义上的幻读 (P3)，同时，SI 还会出现写偏斜，而 ANSI 可重复读隔离级别不会出现写偏斜，会出现幻读。

与 MySQL 可重复读隔离级别的区别

MySQL 可重复读隔离级别在更新时并不检验当前版本是否可见，也就是说，即使该行在事务启动后被更新过，同样可以继续更新。这种情况在 TiDB 使用乐观事务时会导致事务回滚，导致事务最终失败，而 TiDB 默认的悲观事务和 MySQL 是可以更新成功的。

14.11.10.2.2 读已提交隔离级别 (Read Committed)

从 TiDB v4.0.0-beta 版本开始，TiDB 支持使用 Read Committed 隔离级别。由于历史原因，当前主流数据库的 Read Committed 隔离级别本质上都是 Oracle 定义的一致性读隔离级别。TiDB 为了适应这一历史原因，悲观事务中的 Read Committed 隔离级别的实质行为也是一致性读。

> **注意：**
>
> Read Committed 隔离级别仅在悲观事务模式下生效。在乐观事务模式下设置事务隔离级别为 Read Committed 将不会生效，事务将仍旧使用可重复读隔离级别。

从 v6.0.0 版本开始，TiDB 支持使用系统变量 `tidb_rc_read_check_ts` 对读写冲突较少情况下优化时间戳的获取。开启此变量后，SELECT 语句会尝试使用前一个有效的时间戳进行数据读取，初始值为事务的 `start_ts`。

- 如果整个读取过程没有遇到更新的数据版本，则返回结果给客户端且 SELECT 语句执行成功。
- 如果读取过程中遇到更新的数据版本：
  - 如果当前 TiDB 尚未向客户端回复数据，则尝试重新获取一个新的时间戳重试此语句。
  - 如果 TiDB 已经向客户端返回部分数据，则 TiDB 会向客户端报错。每次向客户端回复的数据量受 tidb_init_chunk_size 和 tidb_max_chunk_size 控制。

在使用 READ-COMMITTED 隔离级别且单个事务中 SELECT 语句较多、读写冲突较少的场景，可通过开启此变量来避免获取全局 timestamp 带来的延迟和开销。

从 v6.3.0 版本开始，TiDB 支持通过开启系统变量tidb_rc_write_check_ts 对点写冲突较少情况下优化时间戳的获取。开启此变量后，点写语句会尝试使用当前事务有效的时间戳进行数据读取和加锁操作，且在读取数据时按照开启tidb_rc_read_check_ts 的方式读取数据。目前该变量适用的点写语句包括 UPDATE、DELETE、SELECT ..... FOR UPDATE 三种类型。点写语句是指将主键或者唯一键作为过滤条件且最终执行算子包含 POINT-GET 的写语句。目前这三种点写语句的共同点是会先根据 key 值做点查，如果 key 存在再加锁，如果不存在则直接返回空集。

- 如果点写语句的整个读取过程中没有遇到更新的数据版本，则继续使用当前事务的时间戳进行加锁。
  - 如果加锁过程中遇到因时间戳旧而导致写冲突，则重新获取最新的全局时间戳进行加锁。
  - 如果加锁过程中没有遇到写冲突或其他错误，则加锁成功。
- 如果读取过程中遇到更新的数据版本，则尝试重新获取一个新的时间戳重试此语句。

在使用 READ-COMMITTED 隔离级别且单个事务中点写语句较多、点写冲突较少的场景，可通过开启此变量来避免获取全局时间戳带来的延迟和开销。

与 MySQL Read Committed 隔离级别的区别

MySQL 的 Read Committed 隔离级别大部分符合一致性读特性，但其中存在某些特例，如半一致性读 (semi-consistent read)，TiDB 没有兼容这个特殊行为。

14.11.10.2.3 更多阅读

- TiDB 的乐观事务模型
- TiDB 新特性漫谈 -悲观事务
- TiDB 新特性 -白话悲观锁
- TiKV 的 MVCC (Multi-Version Concurrency Control) 机制

14.11.10.3 TiDB 乐观事务模型

乐观事务模型下，将修改冲突视为事务提交的一部分。因此并发事务不常修改同一行时，可以跳过获取行锁的过程进而提升性能。但是并发事务频繁修改同一行（冲突）时，乐观事务的性能可能低于悲观事务。

启用乐观事务前，请确保应用程序可正确处理 COMMIT 语句可能返回的错误。如果不确定应用程序将会如何处理，建议改为使用悲观事务。

> **注意：**
>
> 自 v3.0.8 开始，TiDB 集群默认使用悲观事务模式。但如果从 3.0.7 及之前版本创建的集群升级到 3.0.8 及之后的版本，不会改变默认事务模式，即只有新创建的集群才会默认使用悲观事务模式。

14.11.10.3.1 乐观事务原理

为支持分布式事务，TiDB 中乐观事务使用两阶段提交协议，流程如下：

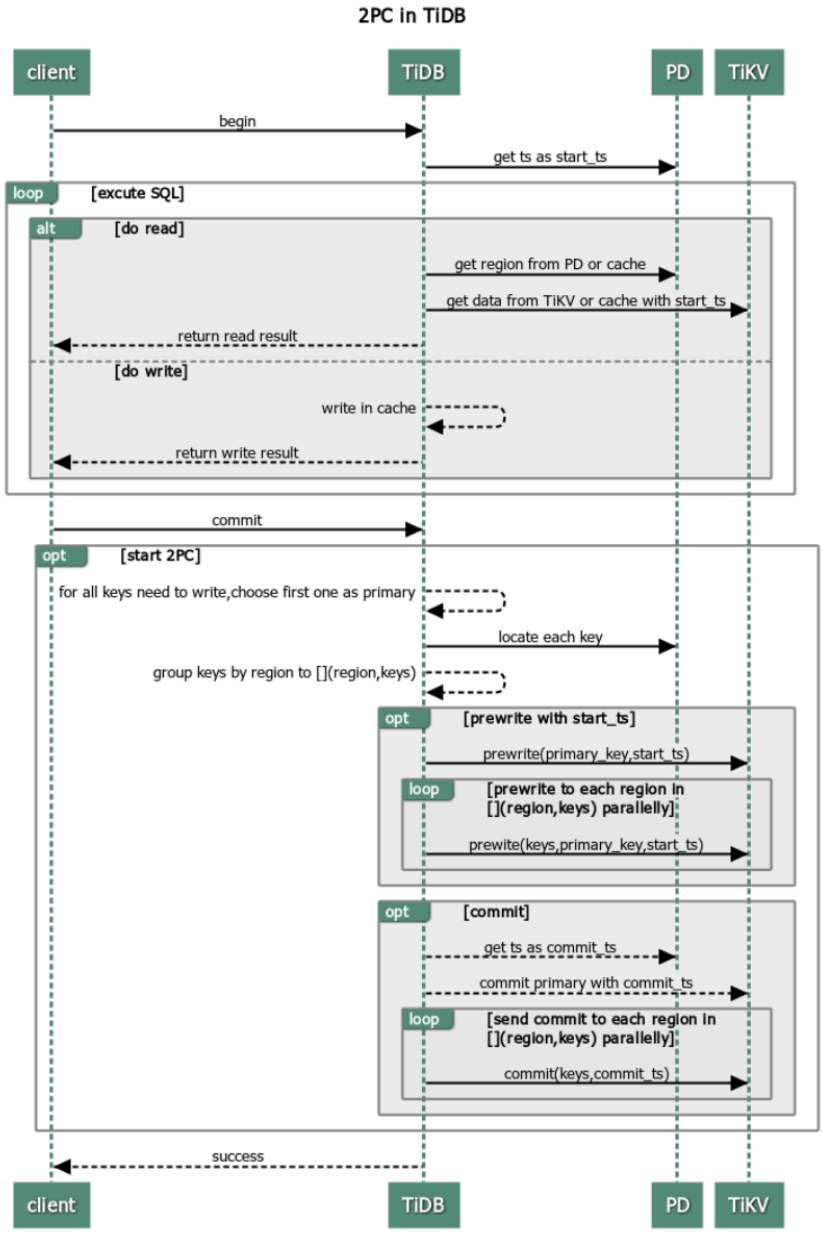


图 485: TiDB 中的两阶段提交

1. 客户端开始一个事务。

   TiDB 从 PD 获取一个全局唯一递增的时间戳作为当前事务的唯一事务 ID，这里称为该事务的 `start_ts`。TiDB 实现了多版本并发控制 (MVCC)，因此 `start_ts` 同时也作为该事务获取的数据库快照版本。该事务只能读到此 `start_ts` 版本可以读到的数据。

2. 客户端发起读请求。

   1. TiDB 从 PD 获取数据路由信息，即数据具体存在哪个 TiKV 节点上。
   2. TiDB 从 TiKV 获取 `start_ts` 版本下对应的数据。

3. 客户端发起写请求。

   TiDB 校验写入数据是否符合约束（如数据类型是否正确、是否符合非空约束等）。校验通过的数据将存放在 TiDB 中该事务的私有内存里。

4. 客户端发起 commit。

5. TiDB 开始两阶段提交，在保证事务原子性的前提下，进行数据持久化。

   1. TiDB 从当前要写入的数据中选择一个 Key 作为当前事务的 Primary Key。
   2. TiDB 从 PD 获取所有数据的写入路由信息，并将所有的 Key 按照所有的路由进行分类。
   3. TiDB 并发地向所有涉及的 TiKV 发起 prewrite 请求。TiKV 收到 prewrite 数据后，检查数据版本信息是否存在冲突或已过期。符合条件的数据会被加锁。
   4. TiDB 收到所有 prewrite 响应且所有 prewrite 都成功。
   5. TiDB 向 PD 获取第二个全局唯一递增版本号，定义为本次事务的 `commit_ts`。
   6. TiDB 向 Primary Key 所在 TiKV 发起第二阶段提交。TiKV 收到 commit 操作后，检查数据合法性，清理 prewrite 阶段留下的锁。
   7. TiDB 收到两阶段提交成功的信息。

6. TiDB 向客户端返回事务提交成功的信息。

7. TiDB 异步清理本次事务遗留的锁信息。

14.11.10.3.2 优缺点分析

通过分析 TiDB 中事务的处理流程，可以发现 TiDB 事务有如下优点：

- 实现原理简单，易于理解。
- 基于单实例事务实现了跨节点事务。
- 锁管理实现了去中心化。

但 TiDB 事务也存在以下缺点：

- 两阶段提交使网络交互增多。
- 需要一个中心化的分配时间戳服务。
- 事务数据量过大时易导致内存暴涨。

14.11.10.3.3 事务的重试

使用乐观事务模型时，在高冲突率的场景中，事务容易发生写写冲突而导致提交失败。MySQL 使用悲观事务模型，在执行写入类型的 SQL 语句的过程中进行加锁并且在 Repeatable Read 隔离级别下使用了当前读的机制，能够读取到最新的数据，所以提交时一般不会出现异常。为了降低应用改造难度，TiDB 提供了数据库内部自动重试机制。

重试机制

当事务提交时，如果发现写写冲突，TiDB 内部重新执行包含写操作的 SQL 语句。你可以通过设置 `tidb_disable_txn_auto_retry = OFF` 开启自动重试，并通过 `tidb_retry_limit` 设置重试次数：

```
#### 设置是否禁用自动重试，默认为 “on”，即不重试。
tidb_disable_txn_auto_retry = OFF
#### 控制重试次数，默认为 “10”。只有自动重试启用时该参数才会生效。
#### 当 “tidb_retry_limit= 0” 时，也会禁用自动重试。
tidb_retry_limit = 10
```

你也可以修改当前 Session 或 Global 的值：

- Session 级别设置：

```
SET tidb_disable_txn_auto_retry = OFF;
```

```
SET tidb_retry_limit = 10;
```

- Global 级别设置：

```
SET GLOBAL tidb_disable_txn_auto_retry = OFF;
```

```
SET GLOBAL tidb_retry_limit = 10;
```

> 注意：
>
> tidb_retry_limit 变量决定了事务重试的最大次数。当它被设置为 0 时，所有事务都不会自动重试，包括自动提交的单语句隐式事务。这是彻底禁用 TiDB 中自动重试机制的方法。禁用自动重试后，所有冲突的事务都会以最快的方式上报失败信息（包含 try again later）给应用层。

重试的局限性

TiDB 默认不进行事务重试，因为重试事务可能会导致更新丢失，从而破坏可重复读的隔离级别。

事务重试的局限性与其原理有关。事务重试可概括为以下三个步骤：

1. 重新获取 start_ts。
2. 重新执行包含写操作的 SQL 语句。
3. 再次进行两阶段提交。

第二步中，重试时仅重新执行包含写操作的 SQL 语句，并不涉及读操作的 SQL 语句。但是当前事务中读到数据的时间与事务真正开始的时间发生了变化，写入的版本变成了重试时获取的 start_ts 而非事务一开始时获取的 start_ts。因此，当事务中存在依赖查询结果来更新的语句时，重试将无法保证事务原本可重复读的隔离级别，最终可能导致结果与预期出现不一致。

如果业务可以容忍事务重试导致的异常，或并不关注事务是否以可重复读的隔离级别来执行，则可以开启自动重试。

14.11.10.3.4 冲突检测

作为一个分布式系统，TiDB 在内存中的冲突检测是在 TiKV 中进行，主要发生在 prewrite 阶段。因为 TiDB 集群是一个分布式系统，TiDB 实例本身无状态，实例之间无法感知到彼此的存在，也就无法确认自己的写入与别的 TiDB 实例是否存在冲突，所以会在 TiKV 这一层检测具体的数据是否有冲突。

具体配置如下：

```
#### scheduler 内置一个内存锁机制，防止同时对一个 Key 进行操作。
#### 每个 Key hash 到不同的 slot。（默认为 2048000）
scheduler-concurrency = 2048000
```

此外，TiKV 支持监控等待 latch 的时间：

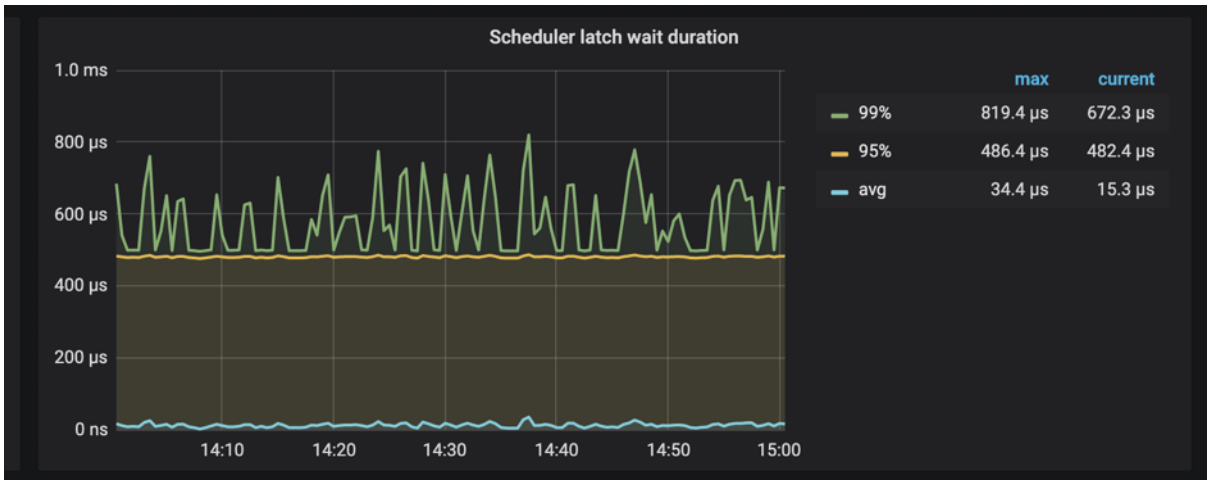


图 486: Scheduler latch wait duration

当 Scheduler latch wait duration 的值特别高时，说明大量时间消耗在等待锁的请求上。如果不存在底层写入慢的问题，基本上可以判断该段时间内冲突比较多。

14.11.10.3.5 更多阅读

- Percolator 和 TiDB 事务算法

14.11.10.4 TiDB 悲观事务模式

为了使 TiDB 的使用方式更加贴近传统数据库，降低用户迁移的成本，TiDB 自 v3.0 版本开始在乐观事务模型的基础上支持了悲观事务模式。本文将介绍 TiDB 悲观事务的相关特性。

> 注意：
>
> 自 v3.0.8 开始，新创建的 TiDB 集群默认使用悲观事务模式。但如果从 v3.0.7 版本及之前创建的集群升级到 >= v3.0.8 的版本，则不会改变默认的事务模式，即只有新创建的集群才会默认使用悲观事务模式。

14.11.10.4.1 事务模式的修改方法

你可以使用tidb_txn_mode 系统变量设置事务模式。执行以下命令，即可使整个集群中所有新创建 session 执行的所有显示事务（即非 autocommit 的事务）进入悲观事务模式：

```
SET GLOBAL tidb_txn_mode = 'pessimistic';
```

除此之外，还可以执行以下 SQL 语句显式地开启悲观事务：

```
BEGIN PESSIMISTIC;
```

```
BEGIN /*T! PESSIMISTIC */;
```

BEGIN PESSIMISTIC; 和 BEGIN OPTIMISTIC; 等语句的优先级高于 tidb_txn_mode 系统变量。使用这两个语句开启的事务，会忽略系统变量，从而支持悲观、乐观事务混合使用。

14.11.10.4.2 悲观事务模式的行为

悲观事务的行为和 MySQL 基本一致（不一致之处详见和 MySQL InnoDB 的差异）：

- 悲观事务中引入快照读和当前读的概念：
  - 快照读是一种不加锁读，读的是该事务开始时刻前已提交的版本。SELECT 语句中的读是快照读。
  - 当前读是一种加锁读，读取的是最新已提交的版本，UPDATE、DELETE、INSERT、SELECT FOR UPDATE 语句中的读是当前读。

  以下示例是对快照读和当前读的详细说明：

session 1	session 2
CREATE TABLE t (a INT);	
INSERT INTO T VALUES(1);	
BEGIN PESSIMISTIC;	
UPDATE t SET a = a + 1;	
	BEGIN PESSIMISTIC;
	SELECT * FROM t; – 使用
COMMIT; – 释放锁，session 3 的 SELECT FOR UPDATE 操作获得锁，使用当前读，读到最新已提交的版本 (a=2)	
	SELECT * FROM t; – 使用

- 悲观锁会在事务提交或回滚时释放。其他尝试修改这一行的写事务会被阻塞，等待悲观锁的释放。其他尝试读取这一行的事务不会被阻塞，因为 TiDB 采用多版本并发控制机制 (MVCC)。
- 需要检查唯一性约束的悲观锁可以通过设置系统变量tidb_constraint_check_in_place_pessimistic 控制是否跳过，详见约束。
- 如果多个事务尝试获取各自的锁，会出现死锁，并被检测器自动检测到。其中一个事务会被随机终止掉并返回兼容 MySQL 的错误码 1213。

- 通过 innodb_lock_wait_timeout 变量，设置事务等锁的超时时间（默认值为 50，单位为秒）。等锁超时后返回兼容 MySQL 的错误码 1205。如果多个事务同时等待同一个锁释放，会大致按照事务 start ts 顺序获取锁。
- 乐观事务和悲观事务可以共存，事务可以任意指定使用乐观模式或悲观模式来执行。
- 支持 FOR UPDATE NOWAIT 语法，遇到锁时不会阻塞等锁，而是返回兼容 MySQL 的错误码 3572。
- 如果 Point Get 和 Batch Point Get 算子没有读到数据，依然会对给定的主键或者唯一键加锁，阻塞其他事务对相同主键唯一键加锁或者进行写入操作。
- 支持 FOR UPDATE OF TABLES 语法，对于存在多表 join 的语句，只对 OF TABLES 中包含的表关联的行进行悲观锁加锁操作。

14.11.10.4.3 和 MySQL InnoDB 的差异

1. 有些 WHERE 子句中使用了 range，TiDB 在执行这类 DML 语句和 SELECT FOR UPDATE 语句时，不会阻塞 range 内并发的 DML 语句的执行。

   举例：

   ```
   CREATE TABLE t1 (
    id INT NOT NULL PRIMARY KEY,
    pad1 VARCHAR(100)
   );
   INSERT INTO t1 (id) VALUES (1),(5),(10);
   ```

   ```
   BEGIN /*T! PESSIMISTIC */;
   SELECT * FROM t1 WHERE id BETWEEN 1 AND 10 FOR UPDATE;
   ```

   ```
   BEGIN /*T! PESSIMISTIC */;
   INSERT INTO t1 (id) VALUES (6); -- 仅 MySQL 中出现阻塞。
   UPDATE t1 SET pad1='new value' WHERE id = 5; -- MySQL 和 TiDB 处于等待阻塞状态。
   ```

   产生这一行为是因为 TiDB 当前不支持 gap locking（间隙锁）。

2. TiDB 不支持 SELECT LOCK IN SHARE MODE。

   使用这个语句执行的时候，效果和没有加锁是一样的，不会阻塞其他事务的读写。

3. DDL 可能会导致悲观事务提交失败。

   MySQL 在执行 DDL 语句时，会被正在执行的事务阻塞住，而在 TiDB 中 DDL 操作会成功，造成悲观事务提交失败：ERROR 1105 (HY000): Information schema is changed. [try again later]。TiDB 事务执行过程中并发执行 TRUNCATE TABLE 语句，可能会导致事务报错 table doesn't exist。

4. START TRANSACTION WITH CONSISTENT SNAPSHOT 之后，MySQL 仍然可以读取到之后在其他事务创建的表，而 TiDB 不能。

5. autocommit 事务优先采用乐观事务提交。

   使用悲观事务模式时，autocommit 事务首先尝试使用开销更小的乐观事务模式提交。如果发生了写冲突，重试时才会使用悲观事务提交。所以 `tidb_retry_limit = 0` 时，autocommit 事务遇到写冲突仍会报 `Write Conflict` 错误。

   自动提交的 `SELECT FOR UPDATE` 语句不会等锁。

6. 对语句中 `EMBEDDED SELECT` 读到的相关数据不会加锁。

7. 垃圾回收 (GC) 不会影响到正在执行的事务，但悲观事务的执行时间仍有上限，默认为 1 小时，可通过 TiDB 配置文件 `[performance]` 类别下的 `max-txn-ttl` 修改。

14.11.10.4.4 隔离级别

TiDB 在悲观事务模式下支持了 2 种隔离级别：

1. 默认使用与 MySQL 行为相同的可重复读隔离级别 (Repeatable Read)。

> **注意：**
> 在这种隔离级别下，DML 操作会基于已提交的最新数据来执行，行为与 MySQL 相同，但与 TiDB 乐观事务不同，请参考与 MySQL 可重复读隔离级别的区别。

2. 使用`SET TRANSACTION` 语句可将隔离级别设置为读已提交隔离级别 (Read Committed)。

14.11.10.4.5 悲观事务提交流程

TiDB 悲观锁复用了乐观锁的两阶段提交逻辑，重点在 DML 执行时做了改造。

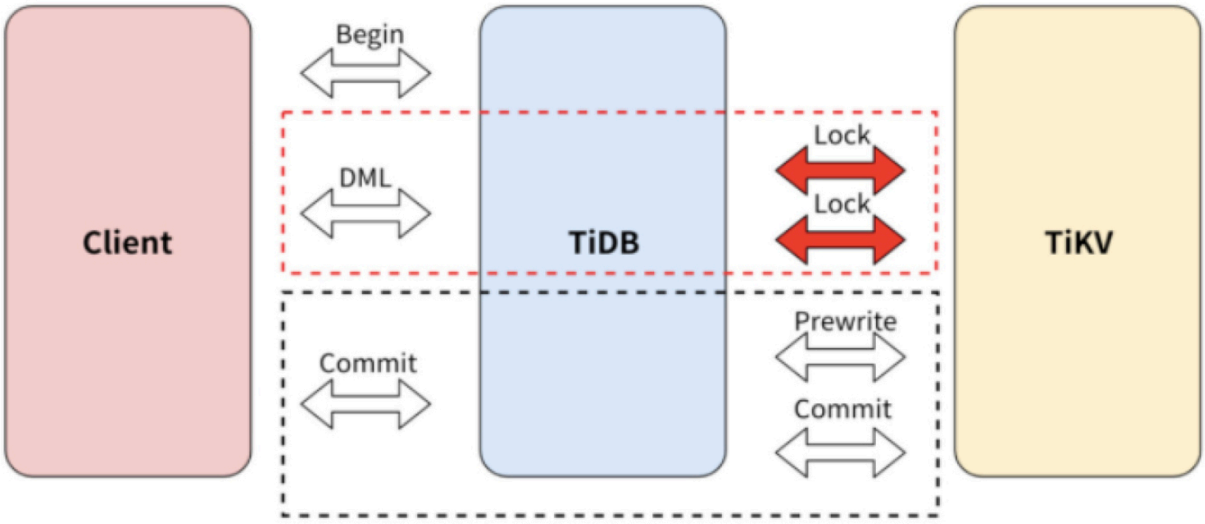


图 487: TiDB 悲观事务的提交流程

在两阶段提交之前增加了 Acquire Pessimistic Lock 阶段，简要步骤如下。

1. （同乐观锁）TiDB 收到来自客户端的 begin 请求，获取当前时间戳作为本事务的 StartTS。
2. TiDB 收到来自客户端的更新数据的请求：TiDB 向 TiKV 发起加悲观锁请求，该锁持久化到 TiKV。
3. （同乐观锁）客户端发起 commit，TiDB 开始执行与乐观锁一样的两阶段提交。

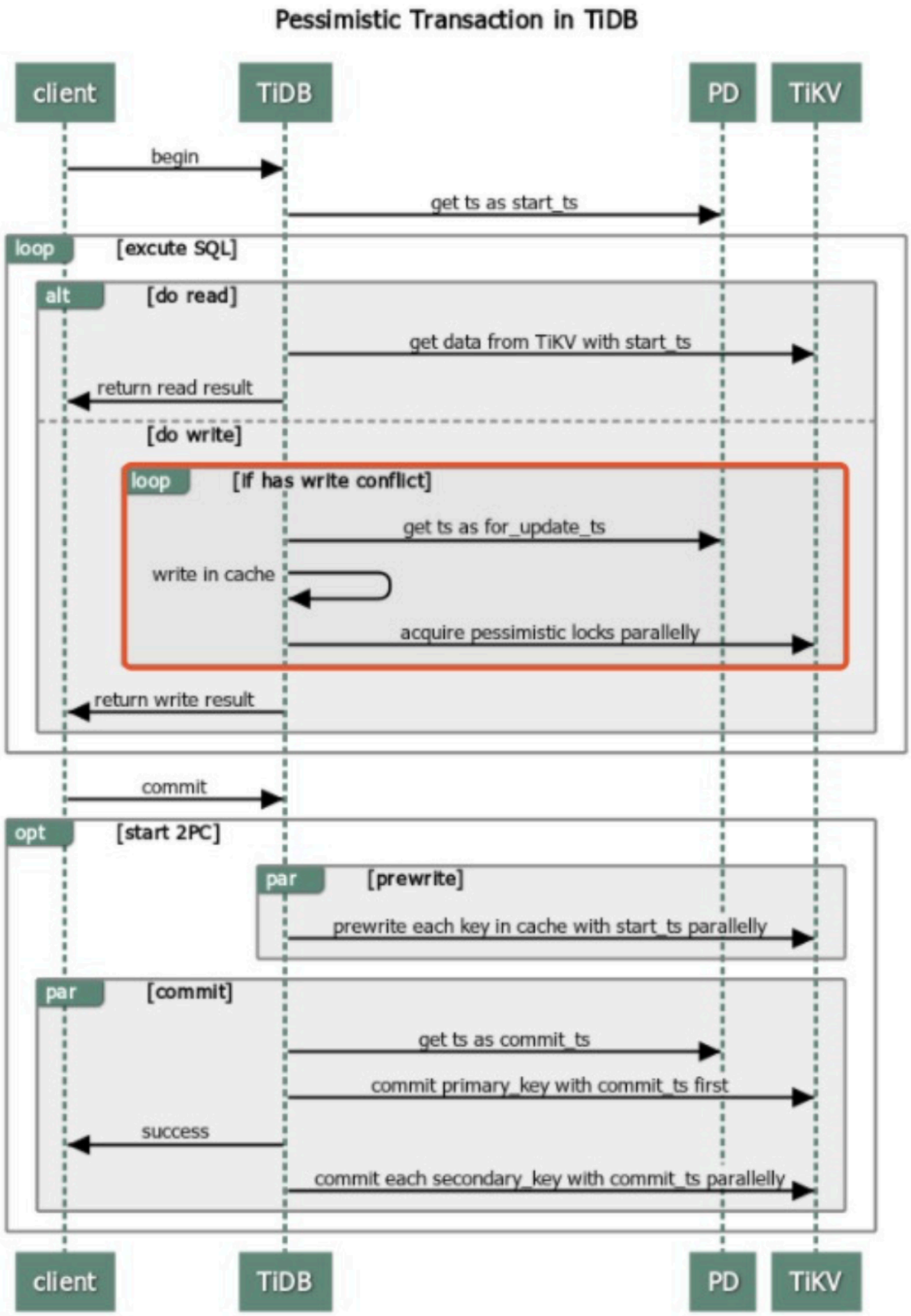


图 488: TiDB 中的悲观事务

相关细节本节不再赘述，详情可阅读 TiDB 悲观锁实现原理。

14.11.10.4.6 Pipelined 加锁流程

加悲观锁需要向 TiKV 写入数据，要经过 Raft 提交并 apply 后才能返回，相比于乐观事务，不可避免的会增加部分延迟。为了降低加锁的开销，TiKV 实现了 pipelined 加锁流程：当数据满足加锁要求时，TiKV 立刻通知 TiDB 执行后面的请求，并异步写入悲观锁，从而降低大部分延迟，显著提升悲观事务的性能。但当 TiKV 出现网络隔离或者节点宕机时，悲观锁异步写入有可能失败，从而产生以下影响：

- 无法阻塞修改相同数据的其他事务。如果业务逻辑依赖加锁或等锁机制，业务逻辑的正确性将受到影响。
- 有较低概率导致事务提交失败，但不会影响事务正确性。

如果业务逻辑依赖加锁或等锁机制，或者即使在集群异常情况下也要尽可能保证事务提交的成功率，应关闭 pipelined 加锁功能。

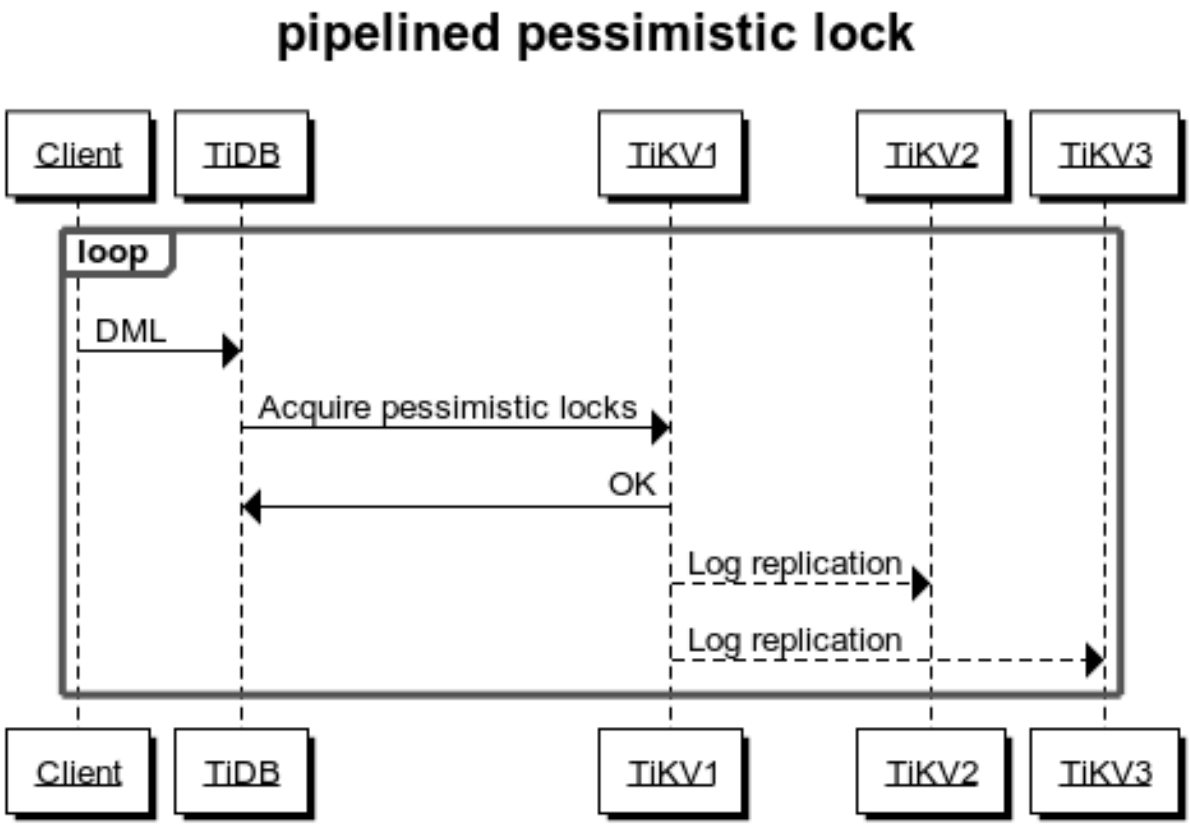


图 489: Pipelined pessimistic lock

该功能默认开启，可修改 TiKV 配置关闭：

```
[pessimistic-txn]
pipelined = false
```

若集群是 v4.0.9 及以上版本，也可通过在线修改 TiKV 配置功能动态关闭该功能：

```
set config tikv pessimistic-txn.pipelined='false';
```

14.11.10.4.7 内存悲观锁

TiKV 在 v6.0.0 中引入了内存悲观锁功能。开启内存悲观锁功能后，悲观锁通常只会被存储在 Region leader 的内存中，而不会将锁持久化到磁盘，也不会通过 Raft 协议将锁同步到其他副本，因此可以大大降低悲观事务加锁的开销，提升悲观事务的吞吐并降低延迟。

当内存悲观锁占用的内存达到 Region 或节点的阈值时，加悲观锁会回退为使用pipelined 加锁流程。当 Region 发生合并或 leader 迁移时，为避免悲观锁丢失，TiKV 会将内存悲观锁写入磁盘并同步到其他副本。

内存悲观锁实现了和pipelined 加锁类似的表现，即集群无异常时不影响加锁表现，但当 TiKV 出现网络隔离或者节点宕机时，事务加的悲观锁可能丢失。

如果业务逻辑依赖加锁或等锁机制，或者即使在集群异常情况下也要尽可能保证事务提交的成功率，应关闭内存悲观锁功能。

该功能默认开启。如要关闭，可修改 TiKV 配置：

```
[pessimistic-txn]
in-memory = false
```

也可通过在线修改 TiKV 配置功能动态关闭该功能：

```
set config tikv pessimistic-txn.in-memory='false';
```

14.11.10.5 非事务 DML 语句

本文档介绍非事务 DML 语句的使用场景、使用方法、使用限制和使用该功能的常见问题。

非事务 DML 语句是将一个普通 DML 语句拆成多个 SQL 语句（即多个 batch）执行，以牺牲事务的原子性和隔离性为代价，增强批量数据处理场景下的性能和易用性。

通常，对于消耗内存过多的大事务，你需要在应用中拆分 SQL 语句以绕过事务大小限制。非事务 DML 语句将这一过程集成到 TiDB 内核中，实现等价的效果。非事务 DML 语句的执行效果可以通过拆分 SQL 语句的结果来理解，DRY RUN 语法提供了预览拆分后语句的功能。

非事务 DML 语句包括：

- INSERT INTO ... SELECT
- REPLACE INTO .. SELECT
- UPDATE
- DELETE

详细的语法介绍见BATCH。

> 注意：
>
> - 非事务 DML 语句不保证该语句的原子性和隔离性，不能认为它和原始 DML 语句等价。
> - 在任意 DML 语句改写为非事务 DML 语句后，不应假设其行为与原来一致。
> - 使用非事务 DML 前需要分析其拆分后的语句是否会互相影响。

14.11.10.5.1 使用场景

在大批量的数据处理场景，用户经常需要对一大批数据执行相同的操作。如果直接使用一个 SQL 语句执行操作，很可能导致事务大小超过限制，而大事务会明显影响执行性能。

批量数据处理操作往往和在线业务操作不具有时间或数据上的交集。没有并发操作时，隔离性是不必要的。如果将批量数据操作设计成幂等的，或者易于重试的，那么原子性也是不必要的。如果你的业务满足这两个条件，那么可以考虑使用非事务 DML 语句。

非事务 DML 语句用于在特定场景下绕过大事务的事务大小限制，用一条语句完成原本需要拆分成多个事务完成的任务，且执行效率更高，占用资源更少。

例如，对于一个删除过期数据的需求，在确保没有任何业务会访问过期数据时，适合使用非事务 DML 语句来提升删除性能。

14.11.10.5.2 前提条件

要使用非事务 DML 语句，必须满足以下条件：

- 确保该语句不需要原子性，即允许执行结果中，一部分行被修改，而一部分行没有被修改。
- 确保该语句具有幂等性，或是做好准备根据错误信息对部分数据重试。如果系统变量 `tidb_redact_log = 1` 且 `tidb_nontransactional_ignore_error = 1`，则该语句必须是幂等的。否则语句部分失败时，无法准确定位失败的部分。
- 确保该语句将要操作的数据没有其它并发的写入，即不被其它语句同时更新。否则可能出现漏写、多写、重复修改同一行等非预期的现象。
- 确保该语句不会修改语句自身会读取的内容，否则后续的 batch 读到之前 batch 写入的内容，容易引起非预期的情况。
  - 在使用非事务 `INSERT INTO ... SELECT` 处理同一张表时，尽量不要在插入时修改拆分列，否则可能因为多个 batch 读取到同一行，导致重复插入：
    * 不推荐使用 `BATCH ON test.t.id LIMIT 10000 INSERT INTO t SELECT id+1, value FROM t;`
    * 推荐使用 `BATCH ON test.t.id LIMIT 10000 INSERT INTO t SELECT id, value FROM t;`
    * 当 `id` 列具有 `AUTO_INCREMENT` 属性时，推荐使用 `BATCH ON test.t.id LIMIT 10000 INSERT INTO t(value)SELECT value FROM t;`
  - 在使用非事务 `UPDATE`、`INSERT ... ON DUPLICATE KEY UPDATE`、`REPLACE INTO` 时，拆分列不应该在语句中更新：

    * 例如，对于一条非事务 UPDATE 语句，拆分后的 SQL 依次执行，前一 batch 的修改提交后被后一 batch 读到，导致同一行数据被多次修改。
    * 这类语句不支持 `BATCH ON test.t.id LIMIT 10000 UPDATE t SET test.t.id = test.t.id-1;`
    * 不推荐使用 `BATCH ON test.t.id LIMIT 1 INSERT INTO t SELECT id+1, value FROM t ON DUPLICATE KEY UPDATE id = id + 1;`
  - 拆分列也不应该用于 Join key。例如，下面示例将拆分列 `test.t.id` 作为 Join key，导致一个非事务 UPDATE 语句多次更新同一行：

```
CREATE TABLE t(id int, v int, key(id));
CREATE TABLE t2(id int, v int, key(id));
INSERT INTO t VALUES (1, 1), (2, 2), (3, 3);
INSERT INTO t2 VALUES (1, 1), (2, 2), (4, 4);
BATCH ON test.t.id LIMIT 1 UPDATE t JOIN t2 ON t.id = t2.id SET t2.id = t2.id+1;
SELECT * FROM t2; -- (4, 1) (4, 2) (4, 4)
```

- 确认该语句满足使用限制。
- 不建议在该 DML 语句将要读写的表上同时进行并发的 DDL 操作。

> **警告：**
>
> 如果同时开启了 `tidb_redact_log` 和 `tidb_nontransactional_ignore_error`，你可能无法完整得知每个 batch 的错误信息，无法只对失败的 batch 进行重试。因此，如果同时开启了这两个系统变量，该非事务 DML 语句必须是幂等的。

14.11.10.5.3 使用示例

使用非事务 DML 语句

以下部分通过示例说明非事务 DML 语句的使用方法：

创建一张表 `t`，表结构如下：

```
CREATE TABLE t(id int, v int, key(id));
```

```
Query OK, 0 rows affected
```

向表 `t` 中插入一些数据。

```
INSERT INTO t VALUES (1,2),(2,3),(3,4),(4,5),(5,6);
```

```
Query OK, 5 rows affected
```

以下操作使用非事务 DML 语句，删除表 `t` 的 `v` 列上小于整数 6 的行。该语句将以 2 为 batch size，以 `id` 列为划分列，拆分为两个 SQL 语句执行。

```
BATCH ON id LIMIT 2 DELETE FROM t where v < 6;
```

```
+----------------+---------------+
| number of jobs | job status    |
+----------------+---------------+
| 2              | all succeeded |
+----------------+---------------+
1 row in set
```

查看非事务 DML 语句的删除结果。

```
SELECT * FROM t;
```

```
+----+---+
| id | v |
+----+---+
| 5  | 6 |
+----+---+
1 row in set
```

以下示例说明多表 join 的使用方法。首先创建表 t2 并插入数据。

```
CREATE TABLE t2(id int, v int, key(id));
INSERT INTO t2 VALUES (1,1), (3,3), (5,5);
```

然后进行涉及多表 join 的更新（表 t 和 t2）。需要注意的是，指定拆分列时需要完整的数据库名、表名和列名（test.t._tidb_rowid)。

```
BATCH ON test.t._tidb_rowid LIMIT 1 UPDATE t JOIN t2 ON t.id = t2.id SET t2.id = t2.id+1;
```

查看更新后表的数据：

```
SELECT * FROM t2;
```

```
+----+---+
| id | v |
+----+---+
| 1  | 1 |
| 3  | 3 |
| 6  | 5 |
+----+---+
```

查看非事务 DML 语句的执行进度

非事务 DML 语句执行过程中，可以通过 SHOW PROCESSLIST 查看执行进度，返回结果中的 Time 表示当前 batch 执行的耗时。日志、慢日志等也会记录每个拆分后的语句在整个非事务 DML 语句中的进度。例如：

```
show processlist;
```

```
+--
    ↪ ----+------+--------------------+--------+---------+------+------------+-------------------------------
    ↪
| Id   | User | Host               | db     | Command | Time | State      | Info
    ↪
    ↪ |
+--
    ↪ ----+------+--------------------+--------+---------+------+------------+-------------------------------
    ↪
| 1203 | root | 100.64.10.62:52711 | test   | Query   | 0    | autocommit | /* job 506/500000 */
    ↪ DELETE FROM `test`.`t1` WHERE `test`.`t1`.`_tidb_rowid` BETWEEN 2271 AND 2273 |
| 1209 | root | 100.64.10.62:52735 | <null> | Query   | 0    | autocommit | show full processlist
    ↪                                                                                |
+--
    ↪ ----+------+--------------------+--------+---------+------+------------+-------------------------------
    ↪
```

终止一个非事务 DML 语句

通过 `KILL TIDB <processlist_id>` 终止一个非事务语句时，TiDB 会取消当前正在执行的 batch 之后的所有 batch。执行结果信息需要从日志里获得。

关于 `KILL TiDB` 的更多信息，参见KILL。

查询非事务 DML 语句中划分 batch 的语句

要查询非事务 DML 语句中用于划分 batch 的语句，你可在非事务 DML 语句中添加 `DRY RUN QUERY`。添加后，TiDB 不实际执行这个查询和后续的 DML 操作。

下面这条语句查询 `BATCH ON id LIMIT 2 DELETE FROM t WHERE v < 6` 这条非事务 DML 语句内用于划分 batch 的查询语句。

```
BATCH ON id LIMIT 2 DRY RUN QUERY DELETE FROM t WHERE v < 6;
```

```
+-----------------------------------------------------------------------------+
| query statement                                                             |
+-----------------------------------------------------------------------------+
| SELECT `id` FROM `test`.`t` WHERE (`v` < 6) ORDER BY IF(ISNULL(`id`),0,1),`id` |
+-----------------------------------------------------------------------------+
1 row in set
```

查询非事务 DML 语句中首末 batch 对应的语句

要查询非事务 DML 语句中第一个和最后一个 batch 对应的实际 DML 语句，你可在语句中添加 `DRY RUN`。添加后，TiDB 只划分 batch，不执行这些 SQL 语句。因为 batch 数量可能很多，不显示全部 batch，只显示第一个和最后一个 batch。

```
BATCH ON id LIMIT 2 DRY RUN DELETE FROM t where v < 6;
```

```
+-----------------------------------------------------------------------+
| split statement examples                                              |
+-----------------------------------------------------------------------+
| DELETE FROM `test`.`t` WHERE (`id` BETWEEN 1 AND 2 AND (`v` < 6)) |
| DELETE FROM `test`.`t` WHERE (`id` BETWEEN 3 AND 4 AND (`v` < 6)) |
+-----------------------------------------------------------------------+
2 rows in set
```

在非事务 DML 语句中使用 Optimizer Hint

对于 DELETE 语句原本支持的 Optimizer Hint，非事务 DELETE 语句也同样支持，hint 位置与普通 DELETE 语句中的位置相同：

```
BATCH ON id LIMIT 2 DELETE /*+ USE_INDEX(t)*/ FROM t where v < 6;
```

14.11.10.5.4 最佳实践

建议按照以下步骤执行非事务 DML 语句：

1. 选择合适的划分列。建议使用整数或字符串类型。
2. 在非事务 DML 语句中添加 DRY RUN QUERY，手动执行查询，确认 DML 语句影响的数据范围是否大体正确。
3. 在非事务 DML 语句中添加 DRY RUN，手动执行查询，检查拆分后的语句和执行计划。需要关注：
    - 一条拆分后的语句是否有可能读到之前的语句执行写入的结果，否则容易造成异常现象。
    - 索引选择效率。
    - 由 TiDB 自动选择的拆分列是否可能会被修改。
4. 执行非事务 DML 语句。
5. 如果报错，从报错信息或日志中获取具体失败的数据范围，进行重试或手动处理。

14.11.10.5.5 参数说明

参数	说明	默认值	是否必填	建议值

参数	说明	默认值	是否必填	建议值

划分列	用于划分 batch 的列，例如以上非事务 DML 语句 `BATCH ON id LIMIT 2 DELETE FROM t WHERE v < 3171 6`	TiDB 尝试自动选择（不建议）	否	选择可以最高效地满足 `WHERE` 条件的列

参数	说明	默认值	是否必填	建议值
Batch size	用于控制每个 batch 的大小，batch 即 DML 操作拆分成的 SQL 语句个数，例如以上非事务 DML 语句 BATCH ↪ ↪ ON ↪ ↪ id ↪ ↪ LIMIT ↪ ↪ 2 3172 ↪ ↪ DELETE ↪	N/A	是	1000 ~ 1000000，过小和过大都会导致性能下降

参数	说明	默认值	是否必填	建议值

划分列的选择

非事务 DML 语句需要用一个列作为数据分批的标准，该列即为划分列。为获得更高的执行效率，划分列必须能够利用索引。不同的索引和划分列所导致的执行效率可能有数十倍的差别。选择划分列时，可以考虑以下建议：

- 当你对业务数据分布有一定了解时，根据 WHERE 条件，选择划分 batch 后，划分范围较小的列。
  - 在理想情况下，WHERE 条件可以利用划分列的索引，降低每个 batch 需要扫描的数据量。例如有一个交易表，记录了每一笔交易的开始和结束时间，你希望删除结束时间在一个月之前的所有交易记录。如果在交易的开始时间有索引，且已知交易的开始和结束时间通常相差不大，那么可以选择开始时间作为划分列。
  - 在不太理想的情况下：划分列的数据分布与 WHERE 条件完全无关，无法利用划分列的索引来减少数据扫描的范围。
- 有聚簇索引时，建议用主键作为划分列，这样语句执行效率更高（包括 INT 主键和 _tidb_rowid）。
- 选择重复值较少的列。

你可以不指定划分列，TiDB 默认会使用 handle 的第一列作为划分列。但如果聚簇索引主键的第一列是非事务 DML 语句不支持的数据类型（即 ENUM，BIT，SET，JSON），TiDB 会报错。你可根据业务需要选择合适的划分列。

Batch size 的选择

非事务 DML 语句中，batch size 越大，拆分出来的 SQL 语句越少，每个 SQL 语句执行起来越慢。最优的 batch size 依据 workload 而定。根据经验值，推荐从 50000 开始尝试。过小和过大的 batch size 都会导致执行效率下降。

每个 batch 的信息存储在内存里，因此 batch 数量过多会显著增加内存消耗。这也是 batch size 不能过小的原因之一。非事务语句用于存储 batch 信息消耗的内存上限与tidb_mem_quota_query 相同，超出这个限制时触发的操作由系统变量tidb_mem_oom_action 控制。

14.11.10.5.6 使用限制

非事务 DML 语句的硬性限制，不满足这些条件时 TiDB 会报错。

- DML 语句不能包含 ORDER BY 或 LIMIT 字句。
- 不支持子查询或集合操作。
- 用于拆分的列必须被索引。该索引可以是单列的索引，或是一个联合索引的第一列。
- 必须在autocommit 模式中使用。
- 不能在开启了 batch-dml 时使用。
- 不能在设置了tidb_snapshot 时使用。
- 不能与 prepare 语句一起使用。
- 划分列不支持 ENUM，BIT，SET，JSON 类型。
- 不支持用在临时表上。
- 不支持公共表表达式。

14.11.10.5.7 控制 batch 执行失败

非事务 DML 语句不满足原子性，可能存在一些 batch 成功，一些 batch 失败的情况。系统变量tidb_nontransactional_ignore_error
↪ 控制非事务 DML 语句处理错误的行为。

一个例外是，如果第一个 batch 就执行失败，有很大概率是语句本身有错，此时整个非事务语句会直接返回一个错误。

14.11.10.5.8 实现原理

非事务 DML 语句的实现原理，是将原本需要在用户侧手动执行的 SQL 语句拆分工作内置为 TiDB 的一个功能，简化用户操作。要理解非事务 DML 语句的行为，可以将其想象成一个用户脚本进行了如下操作：

对于非事务 DML `BATCH ON $C$ LIMIT $N$ DELETE FROM ... WHERE $P$`，其中 C 为用于拆分的列，N 为 batch size，P 为筛选条件。

1. TiDB 根据原始语句的筛选条件 P，和指定的用于拆分的列 C，查询出所有满足 P 的 C。对这些 C 排序后按 N 分成多个分组 $B_1 \dots B_k$。对所有 B_i，保留它的第一个和最后一个 C，记为 S_i 和 E_i。这一步所执行的查询语句，可以通过DRY RUN QUERY 查看。
2. B_i 所涉及的数据就是满足 P_i: C BETWEEN S_i AND E_i 的一个子集。可以通过 P_i 来缩小每个 batch 需要处理的数据范围。
3. 对 B_i，将上面这个条件嵌入原始语句的 WHERE 条件，使其变为 WHERE (P_i) AND (P)。这一步的执行结果，可以通过DRY RUN 查看。
4. 对所有 batch，依次执行新的语句。收集每个分组的错误并合并，在所有分组结束后作为整个非事务 DML 语句的结果返回。

14.11.10.5.9 与 batch-dml 的异同

batch-dml 是一种在 DML 语句执行期间将一个事务拆成多个事务提交的机制。

> 注意：
>
> batch-dml 功能使用不当时，存在数据索引不一致风险。该功能已被废弃，因此不建议使用。

非事务 DML 语句尚不能替代所有的 batch-dml 使用场景。它们的主要区别有：

- 性能：在划分效率较高的情况下，非事务 DML 语句和 batch-dml 性能接近。在划分效率较低的情况下，非事务 DML 语句可能会明显慢于 batch-dml。
- 稳定性：batch-dml 极易因为使用不当导致数据索引不一致问题。而非事务 DML 语句不会导致数据索引不一致问题。但使用不当时，非事务 DML 语句与原始语句不等价，应用可能观察到和预期不符的现象。详见常见问题。

14.11.10.5.10 常见问题

实际的 batch 大小和指定的 batch size 不一样

在非事务 DML 语句的执行过程中，最后一个 batch 处理的数据量可能会小于 batch size。

在划分列有重复值时，每个 batch 会将当前 batch 中划分列的最后一个元素的所有重复值全部加入当前 batch 中，因此这个 batch 的行数可能会多于 batch size。

另外，在有其它并发的数据写入时，也可能导致每个 batch 实际处理的行数和 batch size 不一致。

执行时出现报错 `Failed to restore the delete statement, probably because of unsupported type of the shard column`

划分列的类型暂时不支持 ENUM、BIT、SET、JSON 类型，请尝试重新指定一个划分列。推荐使用整数或字符串类型的列。如果划分列不是这些类型，请从 PingCAP 官方或 TiDB 社区获取支持。

非事务 DELETE 出现和普通的 DELETE 不等价的“异常”行为

非事务 DML 语句和这个 DML 语句的原始形式并不等价，这可能是由以下原因导致的：

- 有并发的其它写入。
- 非事务 DML 语句修改了语句自身会读取的值。
- 在每个 batch 上实际执行的 SQL 语句由于改变了 WHERE 条件，可能会导致执行计划以及表达式计算顺序不同，由此导致了执行结果不一样。
- DML 语句中含有非确定性的操作。

14.11.10.5.11 兼容信息

非事务语句是 TiDB 独有的功能，与 MySQL 不兼容。

14.11.10.5.12 探索更多

- BATCH 语法
- tidb_nontransactional_ignore_error

14.11.11 视图

TiDB 支持视图，视图是一张虚拟表，该虚拟表的结构由创建视图时的 SELECT 语句定义。使用视图一方面可以对用户只暴露安全的字段及数据，进而保证底层表的敏感字段及数据的安全。另一方面，将频繁出现的复杂查询定义为视图，可以使复杂查询更加简单便捷。

14.11.11.1 查询视图

查询一个视图和查询一张普通表类似。但是 TiDB 在真正执行查询视图时，会将视图展开成创建视图时定义的 SELECT 语句，进而执行展开后的查询语句。

14.11.11.2 查看视图的相关信息

通过以下方式，可以查看 view 相关的信息。

14.11.11.2.1 使用 SHOW CREATE TABLE view_name 或 SHOW CREATE VIEW view_name 语句

示例：

```
show create view v;
```

使用该语句可以查看 view 对应的创建语句，及创建 view 时对应的 character_set_client 及 collation_connection 系统变量值。

```
+--
    ↪ ----+------------------------------------------------------------------------------------------------------------
    ↪
| View | Create View
    ↪
    ↪ | character_set_client | collation_connection |
+--
    ↪ ----+------------------------------------------------------------------------------------------------------------
    ↪
| v    | CREATE ALGORITHM=UNDEFINED DEFINER=`root`@`127.0.0.1` SQL SECURITY DEFINER VIEW `v` (`a
    ↪ `) AS SELECT `s`.`a` FROM `test`.`t` LEFT JOIN `test`.`s` ON `t`.`a`=`s`.`a` | utf8
    ↪                  | utf8_general_ci      |
+--
    ↪ ----+------------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

14.11.11.2.2 查询 INFORMATION_SCHEMA.VIEWS 表

示例：

```
select * from information_schema.views;
```

通过查询该表可以查看 view 的相关元信息，如 TABLE_CATALOG、TABLE_SCHEMA、TABLE_NAME、VIEW_DEFINITION、CHECK_OPTION、IS_UPDATABLE、DEFINER、SECURITY_TYPE、CHARACTER_SET_CLIENT、COLLATION_CONNECTION 等。

```
+--
    ↪ -------------+--------------+------------+-----------------------------------------------------------------
    ↪
| TABLE_CATALOG | TABLE_SCHEMA | TABLE_NAME | VIEW_DEFINITION
    ↪                                                          | CHECK_OPTION | IS_UPDATABLE |
    ↪ DEFINER        | SECURITY_TYPE | CHARACTER_SET_CLIENT | COLLATION_CONNECTION |
+--
    ↪ -------------+--------------+------------+-----------------------------------------------------------------
    ↪
| def           | test         | v          | SELECT `s`.`a` FROM `test`.`t` LEFT JOIN `test`.`s`
    ↪  ON `t`.`a`=`s`.`a` | CASCADED     | NO           | root@127.0.0.1 | DEFINER       | utf8
    ↪                   | utf8_general_ci      |
```

```
+--
    ↪ -------------+--------------+------------+-----------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

14.11.11.2.3 查询 HTTP API

示例：

```
curl http://127.0.0.1:10080/schema/test/v
```

通过访问 `http://{TiDBIP}:10080/schema/{db}/{view}` 可以得到对应 view 的所有元信息。

```
{
 "id": 122,
 "name": {
  "O": "v",
  "L": "v"
 },
 "charset": "utf8",
 "collate": "utf8_general_ci",
 "cols": [
  {
   "id": 1,
   "name": {
    "O": "a",
    "L": "a"
   },
   "offset": 0,
   "origin_default": null,
   "default": null,
   "default_bit": null,
   "default_is_expr": false,
   "generated_expr_string": "",
   "generated_stored": false,
   "dependences": null,
   "type": {
    "Tp": 0,
    "Flag": 0,
    "Flen": 0,
    "Decimal": 0,
    "Charset": "",
    "Collate": "",
    "Elems": null
   },
   "state": 5,
```

```
   "comment": "",
   "hidden": false,
   "version": 0
  }
 ],
 "index_info": null,
 "fk_info": null,
 "state": 5,
 "pk_is_handle": false,
 "is_common_handle": false,
 "comment": "",
 "auto_inc_id": 0,
 "auto_id_cache": 0,
 "auto_rand_id": 0,
 "max_col_id": 1,
 "max_idx_id": 0,
 "update_timestamp": 416801600091455490,
 "ShardRowIDBits": 0,
 "max_shard_row_id_bits": 0,
 "auto_random_bits": 0,
 "pre_split_regions": 0,
 "partition": null,
 "compression": "",
 "view": {
  "view_algorithm": 0,
  "view_definer": {
   "Username": "root",
   "Hostname": "127.0.0.1",
   "CurrentUser": false,
   "AuthUsername": "root",
   "AuthHostname": "%"
  },
  "view_security": 0,
  "view_select": "SELECT `s`.`a` FROM `test`.`t` LEFT JOIN `test`.`s` ON `t`.`a`=`s`.`a`",
  "view_checkoption": 1,
  "view_cols": null
 },
 "sequence": null,
 "Lock": null,
 "version": 3,
 "tiflash_replica": null
}
```


14.11.11.3 示例

以下例子将创建一个视图，并在该视图上进行查询，最后删除该视图。

```
create table t(a int, b int);
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
insert into t values(1, 1),(2,2),(3,3);
```

```
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
create table s(a int);
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
insert into s values(2),(3);
```

```
Query OK, 2 rows affected (0.01 sec)
Records: 2  Duplicates: 0  Warnings: 0
```

```
create view v as select s.a from t left join s on t.a = s.a;
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
select * from v;
```

```
+------+
| a    |
+------+
| NULL |
|    2 |
|    3 |
+------+
3 rows in set (0.00 sec)
```

```
drop view v;
```

```
Query OK, 0 rows affected (0.02 sec)
```

14.11.11.4 局限性

目前 TiDB 中的视图有以下局限性：

- 不支持物化视图。
- TiDB 中视图为只读视图，不支持对视图进行 UPDATE、INSERT、DELETE、TRUNCATE 等写入操作。
- 对已创建的视图仅支持 DROP 的 DDL 操作，即 DROP [VIEW | TABLE]。

14.11.11.5 扩展阅读

- 创建视图
- 删除视图

14.11.12 分区表

本文介绍 TiDB 的分区表。

14.11.12.1 分区类型

本节介绍 TiDB 中的分区类型。当前支持的类型包括Range 分区、Range COLUMNS 分区、Range INTERVAL 分区、List 分区、List COLUMNS 分区、Hash 分区和Key 分区。

- Range 分区、Range COLUMNS 分区、List 分区和 List COLUMNS 分区可以用于解决业务中大量删除带来的性能问题，支持快速删除分区。
- Hash 分区和 Key 分区可以用于大量写入场景下的数据打散。与 Hash 分区相比，Key 分区支持多列打散和非整数类型字段的打散。

14.11.12.1.1 Range 分区

一个表按 Range 分区是指，对于表的每个分区中包含的所有行，按分区表达式计算的值都落在给定的范围内。Range 必须是连续的，并且不能有重叠，通过使用 VALUES LESS THAN 进行定义。

下列场景中，假设你要创建一个人事记录的表：

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE DEFAULT '9999-12-31',
    job_code INT,
    store_id INT NOT NULL
);
```

你可以根据需求按各种方式进行 Range 分区。其中一种方式是按 store_id 列进行分区：

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE DEFAULT '9999-12-31',
    job_code INT,
    store_id INT NOT NULL
)
```

```
PARTITION BY RANGE (store_id) (
    PARTITION p0 VALUES LESS THAN (6),
    PARTITION p1 VALUES LESS THAN (11),
    PARTITION p2 VALUES LESS THAN (16),
    PARTITION p3 VALUES LESS THAN (21)
);
```

在这个分区模式中，所有 store_id 为 1 到 5 的员工，都存储在分区 p0 里面，store_id 为 6 到 10 的员工则存储在分区 p1 里面。Range 分区要求，分区的定义必须是有序的，按从小到大递增。

新插入一行数据 (72, 'Tom', 'John', '2015-06-25', NULL, NULL, 15) 将会落到分区 p2 里面。但如果你插入一条 store_id 大于 20 的记录，则会报错，因为 TiDB 无法知晓应该将它插入到哪个分区。这种情况下，可以在建表时使用最大值：

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE DEFAULT '9999-12-31',
    job_code INT,
    store_id INT NOT NULL
)

PARTITION BY RANGE (store_id) (
    PARTITION p0 VALUES LESS THAN (6),
    PARTITION p1 VALUES LESS THAN (11),
    PARTITION p2 VALUES LESS THAN (16),
    PARTITION p3 VALUES LESS THAN MAXVALUE
);
```

MAXVALUE 表示一个比所有整数都大的整数。现在，所有 store_id 列大于等于 16 的记录都会存储在 p3 分区中。

你也可以按员工的职位编号进行分区，也就是使用 job_code 列的值进行分区。假设两位数字编号是用于普通员工，三位数字编号是用于办公室以及客户支持，四位数字编号是管理层职位，那么你可以这样建表：

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE DEFAULT '9999-12-31',
    job_code INT,
    store_id INT NOT NULL
)
```

```
PARTITION BY RANGE (job_code) (
    PARTITION p0 VALUES LESS THAN (100),
    PARTITION p1 VALUES LESS THAN (1000),
    PARTITION p2 VALUES LESS THAN (10000)
);
```

在这个例子中，所有普通员工存储在 p0 分区，办公室以及支持人员在 p1 分区，管理者在 p2 分区。

除了可以按 store_id 切分，你还可以按日期切分。例如，假设按员工离职的年份进行分区：

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE DEFAULT '9999-12-31',
    job_code INT,
    store_id INT
)

PARTITION BY RANGE ( YEAR(separated) ) (
    PARTITION p0 VALUES LESS THAN (1991),
    PARTITION p1 VALUES LESS THAN (1996),
    PARTITION p2 VALUES LESS THAN (2001),
    PARTITION p3 VALUES LESS THAN MAXVALUE
);
```

在 Range 分区中，可以基于 timestamp 列的值分区，并使用 unix_timestamp() 函数，例如：

```
CREATE TABLE quarterly_report_status (
    report_id INT NOT NULL,
    report_status VARCHAR(20) NOT NULL,
    report_updated TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
)

PARTITION BY RANGE ( UNIX_TIMESTAMP(report_updated) ) (
    PARTITION p0 VALUES LESS THAN ( UNIX_TIMESTAMP('2008-01-01 00:00:00') ),
    PARTITION p1 VALUES LESS THAN ( UNIX_TIMESTAMP('2008-04-01 00:00:00') ),
    PARTITION p2 VALUES LESS THAN ( UNIX_TIMESTAMP('2008-07-01 00:00:00') ),
    PARTITION p3 VALUES LESS THAN ( UNIX_TIMESTAMP('2008-10-01 00:00:00') ),
    PARTITION p4 VALUES LESS THAN ( UNIX_TIMESTAMP('2009-01-01 00:00:00') ),
    PARTITION p5 VALUES LESS THAN ( UNIX_TIMESTAMP('2009-04-01 00:00:00') ),
    PARTITION p6 VALUES LESS THAN ( UNIX_TIMESTAMP('2009-07-01 00:00:00') ),
    PARTITION p7 VALUES LESS THAN ( UNIX_TIMESTAMP('2009-10-01 00:00:00') ),
    PARTITION p8 VALUES LESS THAN ( UNIX_TIMESTAMP('2010-01-01 00:00:00') ),
    PARTITION p9 VALUES LESS THAN (MAXVALUE)
);
```

对于 timestamp 列，使用其它的分区表达式是不允许的。

Range 分区在下列条件之一或者多个都满足时，尤其有效：

- 删除旧数据。如果你使用之前的 employees 表的例子，你可以简单使用 `ALTER TABLE employees DROP PARTITION p0;` 删除所有在 1991 年以前停止继续在这家公司工作的员工记录。这会比使用 `DELETE FROM employees WHERE YEAR(separated)<= 1990;` 执行快得多。
- 使用包含时间或者日期的列，或者是其它按序生成的数据。
- 频繁查询分区使用的列。例如执行这样的查询 `EXPLAIN SELECT COUNT(*)FROM employees WHERE separated BETWEEN '2000-01-01' AND '2000-12-31' GROUP BY store_id;` 时，TiDB 可以迅速确定，只需要扫描 p2 分区的数据，因为其它的分区不满足 where 条件。

14.11.12.1.2 Range COLUMNS 分区

Range COLUMNS 分区是 Range 分区的一种变体。你可以使用一个或者多个列作为分区键，分区列的数据类型可以是整数 (integer)、字符串（CHAR/VARCHAR），DATE 和 DATETIME。不支持使用任何表达式。

假设你想要按名字进行分区，并且能够轻松地删除旧的无效数据，那么你可以创建一个表格，如下所示：

```
CREATE TABLE t (
  valid_until datetime,
  name varchar(255) CHARACTER SET ascii,
  notes text
)
PARTITION BY RANGE COLUMNS(name, valid_until)
(PARTITION `p2022-g` VALUES LESS THAN ('G','2023-01-01 00:00:00'),
 PARTITION `p2023-g` VALUES LESS THAN ('G','2024-01-01 00:00:00'),
 PARTITION `p2022-m` VALUES LESS THAN ('M','2023-01-01 00:00:00'),
 PARTITION `p2023-m` VALUES LESS THAN ('M','2024-01-01 00:00:00'),
 PARTITION `p2022-s` VALUES LESS THAN ('S','2023-01-01 00:00:00'),
 PARTITION `p2023-s` VALUES LESS THAN ('S','2024-01-01 00:00:00'))
```

该语句将按名字和年份的范围 `[('', ''), ('G', '2023-01-01 00:00:00')), [('G', '2023-01-01 00:00:00'), ('G', '2024-01-01 00:00:00')), [('G', '2024-01-01 00:00:00'), ('M', '2023-01-01 00:00:00')), [('M', '2023-01-01 00:00:00'), ('M', '2024-01-01 00:00:00')), [('M', '2024-01-01 00:00:00'), ('S', '2023-01-01 00:00:00')), [('S', '2023-01-01 00:00:00'), ('S', '2024-01-01 00:00:00'))` 进行分区，删除无效数据，同时仍然可以在 name 和 valid_until 列上进行分区裁剪。其中，`[,)` 是一个左闭右开区间，比如 `[('G', '2023-01-01 00:00:00'), ('G', '2024-01-01 00:00:00'))`，表示 name 为 `'G'`，年份包含 2023-01-01 00:00:00 并大于 2023-01-01 00:00:00 但小于 2024-01-01 00:00:00 的数据，其中不包含 `(G, 2024-01-01 00:00:00)`。

14.11.12.1.3 Range INTERVAL 分区

TiDB v6.3.0 新增了 Range INTERVAL 分区特性，作为语法糖（syntactic sugar）引入。Range INTERVAL 分区是对 Range 分区的扩展。你可以使用特定的间隔（interval）轻松创建分区。

> **警告：**
>
> 该功能目前是实验性功能，请注意使用场景限制。该功能会在未事先通知的情况下发生变化或删除。语法和实现可能会在 GA 前发生变化。如果发现 bug，请提 Issues ·pingcap/tidb 反馈。

其语法如下：

```
PARTITION BY RANGE [COLUMNS] (<partitioning expression>)
INTERVAL (<interval expression>)
FIRST PARTITION LESS THAN (<expression>)
LAST PARTITION LESS THAN (<expression>)
[NULL PARTITION]
[MAXVALUE PARTITION]
```

示例：

```
CREATE TABLE employees (
    id int unsigned NOT NULL,
    fname varchar(30),
    lname varchar(30),
    hired date NOT NULL DEFAULT '1970-01-01',
    separated date DEFAULT '9999-12-31',
    job_code int,
    store_id int NOT NULL
) PARTITION BY RANGE (id)
INTERVAL (100) FIRST PARTITION LESS THAN (100) LAST PARTITION LESS THAN (10000) MAXVALUE
    ↪ PARTITION
```

该示例创建的表与如下 SQL 语句相同：

```
CREATE TABLE `employees` (
  `id` int unsigned NOT NULL,
  `fname` varchar(30) DEFAULT NULL,
  `lname` varchar(30) DEFAULT NULL,
  `hired` date NOT NULL DEFAULT '1970-01-01',
  `separated` date DEFAULT '9999-12-31',
  `job_code` int DEFAULT NULL,
  `store_id` int NOT NULL
)
PARTITION BY RANGE (`id`)
(PARTITION `P_LT_100` VALUES LESS THAN (100),
 PARTITION `P_LT_200` VALUES LESS THAN (200),
...
 PARTITION `P_LT_9900` VALUES LESS THAN (9900),
```

```
 PARTITION `P_LT_10000` VALUES LESS THAN (10000),
 PARTITION `P_MAXVALUE` VALUES LESS THAN (MAXVALUE))
```

Range INTERVAL 还可以配合Range COLUMNS 分区一起使用。如下面的示例：

```
CREATE TABLE monthly_report_status (
    report_id int NOT NULL,
    report_status varchar(20) NOT NULL,
    report_date date NOT NULL
)
PARTITION BY RANGE COLUMNS (report_date)
INTERVAL (1 MONTH) FIRST PARTITION LESS THAN ('2000-01-01') LAST PARTITION LESS THAN ('2025-01-01
    ↪ ')
```

该示例创建的表与如下 SQL 语句相同：

```
CREATE TABLE `monthly_report_status` (
  `report_id` int(11) NOT NULL,
  `report_status` varchar(20) NOT NULL,
  `report_date` date NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin
PARTITION BY RANGE COLUMNS(`report_date`)
(PARTITION `P_LT_2000-01-01` VALUES LESS THAN ('2000-01-01'),
 PARTITION `P_LT_2000-02-01` VALUES LESS THAN ('2000-02-01'),
...
 PARTITION `P_LT_2024-11-01` VALUES LESS THAN ('2024-11-01'),
 PARTITION `P_LT_2024-12-01` VALUES LESS THAN ('2024-12-01'),
 PARTITION `P_LT_2025-01-01` VALUES LESS THAN ('2025-01-01'))
```

可选参数 NULL PARTITION 会创建一个分区，其中分区表达式推导出的值为 NULL 的数据会放到该分区。在分区表达式中，NULL 会被认为是小于任何其他值。参见分区对 NULL 值的处理。

可选参数 MAXVALUE PARTITION 会创建一个最后的分区，其值为 PARTITION P_MAXVALUE VALUES LESS THAN (
↪ MAXVALUE)。

ALTER INTERVAL 分区

INTERVAL 分区还增加了添加和删除分区的更加简单易用的语法。

下面的语句会变更第一个分区，该语句会删除所有小于给定表达式的分区，使匹配的分区成为新的第一个分区。它不会影响 NULL PARTITION。

```
ALTER TABLE table_name FIRST PARTITION LESS THAN (<expression>)
```

下面的语句会变更最后一个分区，该语句会添加新的分区，分区范围扩大到给定的表达式的值。如果存在 MAXVALUE PARTITION，则该语句不会生效，因为它需要数据重组。

```
ALTER TABLE table_name LAST PARTITION LESS THAN (<expression>)
```

INTERVAL 分区相关细节和限制

- INTERVAL 分区特性仅涉及 CREATE/ALTER TABLE 语法。元数据保持不变，因此使用该新语法创建或变更的表仍然兼容 MySQL。
- 为保持兼容 MySQL，SHOW CREATE TABLE 的输出格式保持不变。
- 遵循 INTERVAL 的存量表可以使用新的 ALTER 语法。不需要使用 INTERVAL 语法重新创建这些表。
- 对于 RANGE COLUMNS，仅支持整数 (INTEGER) 类型、日期 (DATE) 和日期时间 (DATETIME) 列类型。

14.11.12.1.4 List 分区

在创建 List 分区表之前，需要先将 session 变量 tidb_enable_list_partition 的值设置为 ON。

```
set @@session.tidb_enable_list_partition = ON
```

此外，还需保证 tidb_enable_table_partition 变量已开启（默认开启）。

List 分区和 Range 分区有很多相似的地方。不同之处主要在于 List 分区中，对于表的每个分区中包含的所有行，按分区表达式计算的值属于给定的数据集合。每个分区定义的数据集合有任意个值，但不能有重复的值，可通过 PARTITION ... VALUES IN (...) 子句对值进行定义。

假设你要创建一张人事记录表，示例如下：

```
CREATE TABLE employees (
    id INT NOT NULL,
    hired DATE NOT NULL DEFAULT '1970-01-01',
    store_id INT
);
```

假如一共有 20 个商店分布在 4 个地区，如下表所示：

```
| Region  | Store ID Numbers     |
| ------- | -------------------- |
| North   | 1, 2, 3, 4, 5        |
| East    | 6, 7, 8, 9, 10       |
| West    | 11, 12, 13, 14, 15   |
| Central | 16, 17, 18, 19, 20   |
```

如果想把同一个地区商店员工的人事数据都存储在同一个分区中，你可以根据 store_id 来创建 List 分区：

```
CREATE TABLE employees (
    id INT NOT NULL,
    hired DATE NOT NULL DEFAULT '1970-01-01',
    store_id INT
)
PARTITION BY LIST (store_id) (
    PARTITION pNorth VALUES IN (1, 2, 3, 4, 5),
    PARTITION pEast VALUES IN (6, 7, 8, 9, 10),
    PARTITION pWest VALUES IN (11, 12, 13, 14, 15),
    PARTITION pCentral VALUES IN (16, 17, 18, 19, 20)
);
```

这样就能方便地在表中添加或删除与特定区域相关的记录。例如，假设东部地区 (East) 所有的商店都卖给了另一家公司，所有该地区商店员工相关的行数据都可以通过 `ALTER TABLE employees TRUNCATE PARTITION pEast` 删除，这比等效的 `DELETE` 语句 `DELETE FROM employees WHERE store_id IN (6, 7, 8, 9, 10)` 执行起来更加高效。

使用 `ALTER TABLE employees DROP PARTITION pEast` 也能删除所有这些行，但同时也会从表的定义中删除分区 `pEast`。那样你还需要使用 `ALTER TABLE ... ADD PARTITION` 语句来还原表的原始分区方案。

与 Range 分区的情况不同，List 分区没有类似的 `MAXVALUE` 分区来存储所有不属于其他 partition 的值。分区表达式的所有期望值都应包含在 `PARTITION ... VALUES IN (...)` 子句中。如果 `INSERT` 语句要插入的值不匹配分区的列值，该语句将执行失败并报错，如下例所示：

```
test> CREATE TABLE t (
        a INT,
        b INT
      )
      PARTITION BY LIST (a) (
        PARTITION p0 VALUES IN (1, 2, 3),
        PARTITION p1 VALUES IN (4, 5, 6)
      );
Query OK, 0 rows affected (0.11 sec)

test> INSERT INTO t VALUES (7, 7);
ERROR 1525 (HY000): Table has no partition for value 7
```

要忽略以上类型的错误，可以通过使用 `IGNORE` 关键字。使用该关键字后，就不会插入包含不匹配分区列值的行，但是会插入任何具有匹配值的行，并且不会报错：

```
test> TRUNCATE t;
Query OK, 1 row affected (0.00 sec)

test> INSERT IGNORE INTO t VALUES (1, 1), (7, 7), (8, 8), (3, 3), (5, 5);
Query OK, 3 rows affected, 2 warnings (0.01 sec)
Records: 5  Duplicates: 2  Warnings: 2

test> select * from t;
+------+------+
| a    | b    |
+------+------+
|    5 |    5 |
|    1 |    1 |
|    3 |    3 |
+------+------+
3 rows in set (0.01 sec)
```

14.11.12.1.5 List COLUMNS 分区

List COLUMNS 分区是 List 分区的一种变体，可以将多个列用作分区键，并且可以将整数类型以外的数据类型的列用作分区列。你还可以使用字符串类型、DATE 和 DATETIME 类型的列。

假设商店员工分别来自以下 12 个城市，想要根据相关规定分成 4 个区域，如下表所示：

```
| Region | Cities                          |
| :----- | ------------------------------- |
| 1      | LosAngeles,Seattle, Houston     |
| 2      | Chicago, Columbus, Boston       |
| 3      | NewYork, LongIsland, Baltimore  |
| 4      | Atlanta, Raleigh, Cincinnati    |
```

使用列表列分区，你可以为员工数据创建一张表，将每行数据存储在员工所在城市对应的分区中，如下所示：

```
CREATE TABLE employees_1 (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE DEFAULT '9999-12-31',
    job_code INT,
    store_id INT,
    city VARCHAR(15)
)
PARTITION BY LIST COLUMNS(city) (
    PARTITION pRegion_1 VALUES IN('LosAngeles', 'Seattle', 'Houston'),
    PARTITION pRegion_2 VALUES IN('Chicago', 'Columbus', 'Boston'),
    PARTITION pRegion_3 VALUES IN('NewYork', 'LongIsland', 'Baltimore'),
    PARTITION pRegion_4 VALUES IN('Atlanta', 'Raleigh', 'Cincinnati')
);
```

与 List 分区不同的是，你不需要在 COLUMNS() 子句中使用表达式来将列值转换为整数。

List COLUMNS 分区也可以使用 DATE 和 DATETIME 类型的列进行分区，如以下示例中所示，该示例使用与先前的 employees_1 表相同的名称和列，但根据 hired 列采用 List COLUMNS 分区：

```
CREATE TABLE employees_2 (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE DEFAULT '9999-12-31',
    job_code INT,
    store_id INT,
    city VARCHAR(15)
)
PARTITION BY LIST COLUMNS(hired) (
    PARTITION pWeek_1 VALUES IN('2020-02-01', '2020-02-02', '2020-02-03',
```

```
        '2020-02-04', '2020-02-05', '2020-02-06', '2020-02-07'),
    PARTITION pWeek_2 VALUES IN('2020-02-08', '2020-02-09', '2020-02-10',
        '2020-02-11', '2020-02-12', '2020-02-13', '2020-02-14'),
    PARTITION pWeek_3 VALUES IN('2020-02-15', '2020-02-16', '2020-02-17',
        '2020-02-18', '2020-02-19', '2020-02-20', '2020-02-21'),
    PARTITION pWeek_4 VALUES IN('2020-02-22', '2020-02-23', '2020-02-24',
        '2020-02-25', '2020-02-26', '2020-02-27', '2020-02-28')
);
```

另外，你也可以在 COLUMNS() 子句中添加多个列，例如：

```
CREATE TABLE t (
    id int,
    name varchar(10)
)
PARTITION BY LIST COLUMNS(id,name) (
     partition p0 values IN ((1,'a'),(2,'b')),
     partition p1 values IN ((3,'c'),(4,'d')),
     partition p3 values IN ((5,'e'),(null,null))
);
```

14.11.12.1.6 Hash 分区

Hash 分区主要用于保证数据均匀地分散到一定数量的分区里面。在 Range 分区中你必须为每个分区指定值的范围；在 Hash 分区中，你只需要指定分区的数量。

创建 Hash 分区表时，需要在 CREATE TABLE 后面添加 PARTITION BY HASH (expr)，其中 expr 是一个返回整数的表达式。当这一列的类型是整数类型时，它可以是一个列名。此外，你很可能还需要加上 PARTITIONS num，其中 num 是一个正整数，表示将表划分多少分区。

下面的语句将创建一个 Hash 分区表，按 store_id 分成 4 个分区：

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE DEFAULT '9999-12-31',
    job_code INT,
    store_id INT
)

PARTITION BY HASH(store_id)
PARTITIONS 4;
```

如果不指定 PARTITIONS num，默认的分区数量为 1。

你也可以使用一个返回整数的 SQL 表达式。例如，你可以按入职年份分区：

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE DEFAULT '9999-12-31',
    job_code INT,
    store_id INT
)

PARTITION BY HASH( YEAR(hired) )
PARTITIONS 4;
```

最高效的 Hash 函数是作用在单列上，并且函数的单调性是跟列的值是一样递增或者递减的。

例如，date_col 是类型为 DATE 的列，表达式 TO_DAYS(date_col) 的值是直接随 date_col 的值变化的。YEAR(date_col) 跟 TO_DAYS(date_col) 就不太一样，因为不是每次 date_col 变化时 YEAR(date_col) 都会得到不同的值。

作为对比，假设我们有一个类型是 INT 的 int_col 的列。考虑一下表达式 POW(5-int_col,3)+ 6，这并不是一个比较好的 Hash 函数，因为随着 int_col 的值的变化，表达式的结果不会成比例地变化。改变 int_col 的值会使表达式的结果的值变化巨大。例如，int_col 从 5 变到 6 表达式的结果变化是 -1，但是从 6 变到 7 的时候表达式的值的变化是 -7。

总而言之，表达式越接近 y = cx 的形式，它越是适合作为 Hash 函数。因为表达式越是非线性的，在各个分区上面的数据的分布越是倾向于不均匀。

理论上，Hash 分区也是可以做分区裁剪的。而实际上对于多列的情况，实现很难并且计算很耗时。因此，不推荐 Hash 分区在表达式中涉及多列。

使用 PARTITIION BY HASH 的时候，TiDB 通过表达式的结果做“取余”运算，决定数据落在哪个分区。换句话说，如果分区表达式是 expr，分区数是 num，则由 MOD(expr, num) 决定存储的分区。假设 t1 定义如下：

```
CREATE TABLE t1 (col1 INT, col2 CHAR(5), col3 DATE)
    PARTITION BY HASH( YEAR(col3) )
    PARTITIONS 4;
```

向 t1 插入一行数据，其中 col3 列的值是 ‘2005-09-15’，这条数据会被插入到分区 1 中：

```
MOD(YEAR('2005-09-01'),4)
=  MOD(2005,4)
=  1
```

14.11.12.1.7 Key 分区

TiDB 从 v7.0.0 开始支持 Key 分区。在 v7.0.0 之前的版本中，创建 Key 分区表时，TiDB 会将其创建为非分区表并给出告警。

Key 分区与 Hash 分区都可以保证将数据均匀地分散到一定数量的分区里面，区别是 Hash 分区只能根据一个指定的整数表达式或字段进行分区，而 Key 分区可以根据字段列表进行分区，且 Key 分区的分区字段不局限于整数类型。TiDB Key 分区表的 Hash 算法与 MySQL 不一样，因此表的数据分布也不一样。

创建 Key 分区表时，你需要在 `CREATE TABLE` 后面添加 `PARTITION BY KEY (columList)`，其中 `columnList` 是字段列表，可以包含一个或多个字段。每个字段的类型可以是除 BLOB、JSON、GEOMETRY 之外的任意类型（请注意 TiDB 不支持 GEOMETRY 类型）。此外，你很可能还需要加上 `PARTITIONS num`，其中 `num` 是一个正整数，表示将表划分多少个分区；或者加上分区名的定义，例如，加上 `(PARTITION p0, PARTITION p1)` 代表将表划分为两个分区，分区名为 `p0` 和 `p1`。

下面的语句将创建一个 Key 分区表，按 `store_id` 分成 4 个分区：

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE DEFAULT '9999-12-31',
    job_code INT,
    store_id INT
)

PARTITION BY KEY(store_id)
PARTITIONS 4;
```

如果不指定 `PARTITIONS num`，默认的分区数量为 1。

你也可以根据 VARCHAR 等非整数字段创建 Key 分区表。下面的语句按 `fname` 将表分成 4 个分区：

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE DEFAULT '9999-12-31',
    job_code INT,
    store_id INT
)

PARTITION BY KEY(fname)
PARTITIONS 4;
```

你还可以根据多列字段创建 Key 分区表。下面的语句按 `fname`、`store_id` 将表分成 4 个分区：

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
```

```
    separated DATE DEFAULT '9999-12-31',
    job_code INT,
    store_id INT
)

PARTITION BY KEY(fname, store_id)
PARTITIONS 4;
```

目前，TiDB 不支持分区字段列表 `PARTITION BY KEY` 为空的 Key 分区表。下面的语句将创建一个非分区表，并向客户端返回 `Unsupported partition type KEY, treat as normal table` 警告。

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE DEFAULT '9999-12-31',
    job_code INT,
    store_id INT
)

PARTITION BY KEY()
PARTITIONS 4;
```

14.11.12.1.8 TiDB 对 Linear Hash 分区的处理

在 v6.4.0 之前，如果在 TiDB 上执行 MySQL Linear Hash 分区 的 DDL 语句，TiDB 只能创建非分区表。在这种情况下，如果你仍然想要在 TiDB 中创建分区表，你需要修改这些 DDL 语句。

从 v6.4.0 起，TiDB 支持解析 MySQL 的 `PARTITION BY LINEAR HASH` 语法，但会忽略其中的 `LINEAR` 关键字。你可以直接在 TiDB 中执行现有的 MySQL Linear Hash 分区的 SQL 语句，而无需修改。

- 对于 MySQL Linear Hash 分区的 `CREATE` 语句，TiDB 将创建一个常规的非线性 Hash 分区表（注意 TiDB 内部实际不存在 Linear Hash 分区表）。如果分区数是 2 的幂，该分区表中行的分布情况与 MySQL 相同。如果分区数不是 2 的幂，该分区表中行的分布情况与 MySQL 会有所差异。这是因为 TiDB 中非线性分区表使用简单的“分区模数”，而线性分区表使用“模数的下一个 2 次方并会折叠分区数和下一个 2 次方之间的值”。详情请见 #38450。
- 对于 MySQL Linear Hash 分区的其他 SQL 语句，TiDB 将正常返回对应的 Hash 分区的查询结果。但当分区数不是 2 的幂（意味着分区表中行的分布情况与 MySQL 不同）时，分区选择、`TRUNCATE PARTITION`、`EXCHANGE PARTITION` 返回的结果将和 MySQL 有所差异。

14.11.12.1.9 TiDB 对 Linear Key 分区的处理

TiDB 从 v7.0.0 开始支持 Key 分区，并支持解析 MySQL 的 `PARTITION BY LINEAR KEY` 语法，但会忽略其中的 `LINEAR` 关键字，只采用非线性 Hash 算法。

在 v7.0.0 之前的版本中，创建 Key 分区表时，TiDB 会将其创建为非分区表并给出告警。

14.11.12.1.10 分区对 NULL 值的处理

TiDB 允许计算结果为 NULL 的分区表达式。注意，NULL 不是一个整数类型，NULL 小于所有的整数类型值，正如 ORDER BY 的规则一样。

Range 分区对 NULL 的处理

如果插入一行到 Range 分区表，它的分区列的计算结果是 NULL，那么这一行会被插入到最小的那个分区。

```
CREATE TABLE t1 (
    c1 INT,
    c2 VARCHAR(20)
)

PARTITION BY RANGE(c1) (
    PARTITION p0 VALUES LESS THAN (0),
    PARTITION p1 VALUES LESS THAN (10),
    PARTITION p2 VALUES LESS THAN MAXVALUE
);
```

```
Query OK, 0 rows affected (0.09 sec)
```

```
select * from t1 partition(p0);
```

```
+------|--------+
| c1   | c2     |
+------|--------+
| NULL | mothra |
+------|--------+
1 row in set (0.00 sec)
```

```
select * from t1 partition(p1);
```

```
Empty set (0.00 sec)
```

```
select * from t1 partition(p2);
```

```
Empty set (0.00 sec)
```

删除 p0 后验证：

```
alter table t1 drop partition p0;
```

```
Query OK, 0 rows affected (0.08 sec)
```

```
select * from t1;
```

```
Empty set (0.00 sec)
```

Hash 分区对 NULL 的处理

在 Hash 分区中 NULL 值的处理有所不同，如果分区表达式的计算结果为 NULL，它会被当作 0 值处理。

```
CREATE TABLE th (
    c1 INT,
    c2 VARCHAR(20)
)

PARTITION BY HASH(c1)
PARTITIONS 2;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
INSERT INTO th VALUES (NULL, 'mothra'), (0, 'gigan');
```

```
Query OK, 2 rows affected (0.04 sec)
```

```
select * from th partition (p0);
```

```
+------|--------+
| c1   | c2     |
+------|--------+
| NULL | mothra |
|    0 | gigan  |
+------|--------+
2 rows in set (0.00 sec)
```

```
select * from th partition (p1);
```

```
Empty set (0.00 sec)
```

可以看到，插入的记录 `(NULL, 'mothra')` 跟 `(0, 'gigan')` 落在了同一个分区。

> **注意：**
>
> 这里 Hash 分区对 NULL 的处理跟 MySQL 的文档描述一致，但是跟 MySQL 的实际行为并不一致。也就是说，MySQL 的文档跟它的实现并不一致。
>
> TiDB 的最终行为以本文档描述为准。

Key 分区对 NULL 的处理

在 Key 分区中 NULL 值的处理与 Hash 分区一致：如果分区字段的值为 NULL，它会被当作 0 值处理。

14.11.12.2 分区管理

对于 RANGE、 RANGE COLUMNS、 LIST、 LIST COLUMNS 分区表，你可以进行以下分区管理操作：

- 使用 ALTER TABLE <表名> ADD PARTITION (<分区说明>) 语句添加分区。
- 使用 ALTER TABLE <表名> DROP PARTITION <分区列表> 删除分区。
- 使用 ALTER TABLE <表名> TRUNCATE PARTITION <分区列表> 语句清空分区里的数据。TRUNCATE PARTITION 的逻辑与TRUNCATE TABLE 相似，但它的操作对象为分区。
- 使用 ALTER TABLE <表名> REORGANIZE PARTITION <分区列表> INTO (<新的分区说明>)语句对分区进行合并、拆分、或者其他修改。

对于 HASH 和 KEY 分区表，目前只支持 ALTER TABLE ... TRUNCATE PARTITION 分区管理语句，不支持 COALESCE PARTITION 和 ADD PARTITION 语句。

EXCHANGE PARTITION 语句用来交换分区和非分区表，类似于重命名表如 RENAME TABLE t1 TO t1_tmp, t2 TO t1, t1_tmp TO t2 的操作。

例如，ALTER TABLE partitioned_table EXCHANGE PARTITION p1 WITH TABLE non_partitioned_table 交换的是 p1 分区的 partitioned_table 表和 non_partitioned_table 表。

确保要交换入分区中的所有行与分区定义匹配；否则，交换将失败。

请注意对于以下 TiDB 专有的特性，当表结构中包含这些特性时，在 TiDB 中使用 EXCHANGE PARTITION 功能不仅需要满足 MySQL 的 EXCHANGE PARTITION 条件，还要保证这些专有特性对于分区表和非分区表的定义相同。

- Placement Rules in SQL：Placement Policy 定义相同。
- TiFlash：TiFlash Replica 数量相同。
- 聚簇索引：分区表和非分区表要么都是聚簇索引 (CLUSTERED)，要么都不是聚簇索引 (NONCLUSTERED)。

此外，EXCHANGE PARTITION 和其他组件兼容性上存在一些限制，需要保证分区表和非分区表的一致性：

- TiFlash：TiFlash Replica 定义不同时，无法执行 EXCHANGE PARTITION 操作。
- TiCDC：分区表和非分区表都有主键或者唯一键时，TiCDC 同步 EXCHANGE PARTITION 操作；反之 TiCDC 将不会同步。
- TiDB Lightning 和 BR：使用 TiDB Lightning 导入或使用 BR 恢复的过程中，不要执行 EXCHANGE PARTITION 操作。

14.11.12.2.1 管理 List 分区、 List COLUMNS 分区、 Range 分区、 Range COLUMNS 分区

本小节将以如下 SQL 语句创建的分区表为例，介绍如何管理 Range 分区和 List 分区。

```
CREATE TABLE members (
    id int,
    fname varchar(255),
    lname varchar(255),
    dob date,
    data json
)
PARTITION BY RANGE (YEAR(dob)) (
 PARTITION pBefore1950 VALUES LESS THAN (1950),
```

```
 PARTITION p1950 VALUES LESS THAN (1960),
 PARTITION p1960 VALUES LESS THAN (1970),
 PARTITION p1970 VALUES LESS THAN (1980),
 PARTITION p1980 VALUES LESS THAN (1990),
 PARTITION p1990 VALUES LESS THAN (2000));

CREATE TABLE member_level (
 id int,
 level int,
 achievements json
)
PARTITION BY LIST (level) (
 PARTITION l1 VALUES IN (1),
 PARTITION l2 VALUES IN (2),
 PARTITION l3 VALUES IN (3),
 PARTITION l4 VALUES IN (4),
 PARTITION l5 VALUES IN (5));
```

删除分区

```
ALTER TABLE members DROP PARTITION p1990;

ALTER TABLE member_level DROP PARTITION l5;
```

清空分区

```
ALTER TABLE members TRUNCATE PARTITION p1980;

ALTER TABLE member_level TRUNCATE PARTITION l4;
```

添加分区

```
ALTER TABLE members ADD PARTITION (PARTITION `p1990to2010` VALUES LESS THAN (2010));

ALTER TABLE member_level ADD PARTITION (PARTITION l5_6 VALUES IN (5,6));
```

对于 Range 分区表，ADD PARTITION 只能在分区列表的最后添加新的分区。与分区列表中已有的分区相比，你需要将新分区的 VALUES LESS THAN 定义为更大的值。否则，执行该语句时将会报错。

```
ALTER TABLE members ADD PARTITION (PARTITION p1990 VALUES LESS THAN (2000));
```

```
ERROR 1493 (HY000): VALUES LESS THAN value must be strictly increasing for each partition
```

重组分区

拆分分区：

```
ALTER TABLE members REORGANIZE PARTITION `p1990to2010` INTO
(PARTITION p1990 VALUES LESS THAN (2000),
 PARTITION p2000 VALUES LESS THAN (2010),
 PARTITION p2010 VALUES LESS THAN (2020),
 PARTITION p2020 VALUES LESS THAN (2030),
 PARTITION pMax VALUES LESS THAN (MAXVALUE));

ALTER TABLE member_level REORGANIZE PARTITION l5_6 INTO
(PARTITION l5 VALUES IN (5),
 PARTITION l6 VALUES IN (6));
```

合并分区：

```
ALTER TABLE members REORGANIZE PARTITION pBefore1950,p1950 INTO (PARTITION pBefore1960 VALUES
    ↪ LESS THAN (1960));

ALTER TABLE member_level REORGANIZE PARTITION l1,l2 INTO (PARTITION l1_2 VALUES IN (1,2));
```

修改分区表定义：

```
ALTER TABLE members REORGANIZE PARTITION pBefore1960,p1960,p1970,p1980,p1990,p2000,p2010,p2020,
    ↪ pMax INTO
(PARTITION p1800 VALUES LESS THAN (1900),
 PARTITION p1900 VALUES LESS THAN (2000),
 PARTITION p2000 VALUES LESS THAN (2100));

ALTER TABLE member_level REORGANIZE PARTITION l1_2,l3,l4,l5,l6 INTO
(PARTITION lOdd VALUES IN (1,3,5),
 PARTITION lEven VALUES IN (2,4,6));
```

在重组分区时，需要注意以下关键点：

- 重组分区（包括合并或拆分分区）只能修改分区定义，无法修改分区表类型。例如，无法将 List 类型修改为 Range 类型，或将 Range COLUMNS 类型修改为 Range 类型。
- 对于 Range 分区表，你只能对表中相邻的分区进行重组：

```
ALTER TABLE members REORGANIZE PARTITION p1800,p2000 INTO (PARTITION p2000 VALUES LESS THAN
    ↪ (2100));
```

```
ERROR 8200 (HY000): Unsupported REORGANIZE PARTITION of RANGE; not adjacent partitions
```

- 对于 Range 分区表，如需修改 Range 定义中的最大值，必须保证 VALUES LESS THAN 中新定义的值大于现有分区中的所有值。否则，TiDB 将报错，提示现有的行值对应不到分区。

```
INSERT INTO members VALUES (313, "John", "Doe", "2022-11-22", NULL);
ALTER TABLE members REORGANIZE PARTITION p2000 INTO (PARTITION p2000 VALUES LESS THAN (2050)
    ↪ ); -- 执行成功，因为 2050 包含了现有的所有行
ALTER TABLE members REORGANIZE PARTITION p2000 INTO (PARTITION p2000 VALUES LESS THAN (2020)
    ↪ ); -- 执行失败，因为 2022 将对应不到分区
```

```
ERROR 1526 (HY000): Table has no partition for value 2022
```

- 对于 List 分区表，如需修改分区定义中的数据集合，必须保证新的数据集合能覆盖到该分区中现有的所有值，否则 TiDB 将报错。

```
INSERT INTO member_level (id, level) values (313, 6);
ALTER TABLE member_level REORGANIZE PARTITION lEven INTO (PARTITION lEven VALUES IN (2,4));
```

```
ERROR 1526 (HY000): Table has no partition for value 6
```

14.11.12.2.2 管理 Hash 分区

跟 Range 分区不同，Hash 分区不支持 DROP PARTITION。

目前 TiDB 的实现暂时不支持 ALTER TABLE ... COALESCE PARTITION。对于暂不支持的分区管理语句，TiDB 会返回错误。

```
ALTER TABLE MEMBERS OPTIMIZE PARTITION p0;
```

```
ERROR 8200 (HY000): Unsupported optimize partition
```

14.11.12.2.3 Key 分区管理

目前 TiDB 中 Key 分区支持的分区管理语句只有 ALTER TABLE ... TRUNCATE PARTITION。

```
ALTER TABLE members TRUNCATE PARTITION p0;
```

```
Query OK, 0 rows affected (0.03 sec)
```

对于暂不支持的分区管理语句，TiDB 会返回错误。

```
ALTER TABLE members OPTIMIZE PARTITION p0;
```

```
ERROR 8200 (HY000): Unsupported optimize partition
```

14.11.12.3 分区裁剪

有一个优化叫做“分区裁剪”，它基于一个非常简单的概念：不需要扫描那些匹配不上的分区。

假设创建一个分区表 t1：

```
CREATE TABLE t1 (
    fname VARCHAR(50) NOT NULL,
    lname VARCHAR(50) NOT NULL,
    region_code TINYINT UNSIGNED NOT NULL,
    dob DATE NOT NULL
)

PARTITION BY RANGE( region_code ) (
    PARTITION p0 VALUES LESS THAN (64),
    PARTITION p1 VALUES LESS THAN (128),
    PARTITION p2 VALUES LESS THAN (192),
    PARTITION p3 VALUES LESS THAN MAXVALUE
);
```

如果你想获得这个 select 语句的结果：

```
SELECT fname, lname, region_code, dob
    FROM t1
    WHERE region_code > 125 AND region_code < 130;
```

很显然，结果必然是在分区 p1 或者 p2 里面，也就是说，我们只需要在 p1 和 p2 里面去搜索匹配的行。去掉不必要的分区就是所谓的裁剪。优化器如果能裁剪掉一部分的分区，则执行会快于处理整个不做分区的表的相同查询。

优化器可以通过 where 条件裁剪的两个场景：

- partition_column = constant
- partition_column IN (constant1, constant2, ⋯, constantN)

分区裁剪暂不支持 LIKE 语句。

14.11.12.3.1 分区裁剪生效的场景

1. 分区裁剪需要使用分区表上面的查询条件，所以根据优化器的优化规则，如果查询条件不能下推到分区表，则相应的查询语句无法执行分区裁剪。

   例如：

   ```
   create table t1 (x int) partition by range (x) (
       partition p0 values less than (5),
       partition p1 values less than (10));
   create table t2 (x int);
   ```

```
explain select * from t1 left join t2 on t1.x = t2.x where t2.x > 5;
```

在这个查询中，外连接可以简化成内连接，然后由 `t1.x = t2.x` 和 `t2.x > 5` 可以推出条件 `t1.x > 5`，于是可以分区裁剪并且只使用 `p1` 分区。

```
explain select * from t1 left join t2 on t1.x = t2.x and t2.x > 5;
```

这个查询中的 `t2.x > 5` 条件不能下推到 `t1` 分区表上面，因此 `t1` 无法分区裁剪。

2. 由于分区裁剪的规则优化是在查询计划的生成阶段，对于执行阶段才能获取到过滤条件的场景，无法利用分区裁剪的优化。

   例如：

```
create table t1 (x int) partition by range (x) (
    partition p0 values less than (5),
    partition p1 values less than (10));
```

```
explain select * from t2 where x < (select * from t1 where t2.x < t1.x and t2.x < 2);
```

这个查询每从 `t2` 读取一行，都会去分区表 `t1` 上进行查询，理论上这时会满足 `t1.x > val` 的过滤条件，但实际上由于分区裁剪只作用于查询计划生成阶段，而不是执行阶段，因而不会做裁剪。

3. 由于当前实现中的一处限制，对于查询条件无法下推到 TiKV 的表达式，不支持分区裁剪。

   对于一个函数表达式 `fn(col)`，如果 TiKV 支持这个函数 `fn`，则在查询优化做谓词下推的时候，`fn(col)` 会被推到叶子节点（也就是分区），因而能够执行分区裁剪。

   如果 TiKV 不支持 `fn`，则优化阶段不会把 `fn(col)` 推到叶子节点，而是在叶子上面连接一个 Selection 节点，分区裁剪的实现没有处理这种父节点的 Selection 中的条件，因此对不能下推到 TiKV 的表达式不支持分区裁剪。

4. 对于 Hash 和 Key 分区类型，只有等值比较的查询条件能够支持分区裁剪。

5. 对于 Range 分区类型，分区表达式必须是 `col` 或者 `fn(col)` 的简单形式，查询条件是 `>`、`<`、`=`、`>=`、`<=` 时才能支持分区裁剪。如果分区表达式是 `fn(col)` 形式，还要求 `fn` 必须是单调函数，才有可能分区裁剪。

   这里单调函数是指某个函数 `fn` 满足条件：对于任意 `x y`，如果 `x > y`，则 `fn(x)> fn(y)`。

   这种是严格递增的单调函数，非严格递增的单调函数也可以符合分区裁剪要求，只要函数 `fn` 满足：对于任意 `x y`，如果 `x > y`，则 `fn(x)>= fn(y)`。

   理论上所有满足单调条件（严格或者非严格）的函数都是可以支持分区裁剪。实际上，目前 TiDB 已经支持的单调函数只有：

   - `UNIX_TIMESTAMP()`
   - `TO_DAYS()`

   例如，分区表达式是简单列的情况：

```
create table t (id int) partition by range (id) (
    partition p0 values less than (5),
    partition p1 values less than (10));
select * from t where id > 6;
```

分区表达式是 fn(col) 的形式，fn 是我们支持的单调函数 to_days：

```
create table t (dt datetime) partition by range (to_days(id)) (
    partition p0 values less than (to_days('2020-04-01')),
    partition p1 values less than (to_days('2020-05-01')));
select * from t where dt > '2020-04-18';
```

有一处例外是 floor(unix_timestamp(ts)) 作为分区表达式，TiDB 针对这个场景做了特殊处理，可以支持分区裁剪。

```
create table t (ts timestamp(3) not null default current_timestamp(3))
partition by range (floor(unix_timestamp(ts))) (
    partition p0 values less than (unix_timestamp('2020-04-01 00:00:00')),
    partition p1 values less than (unix_timestamp('2020-05-01 00:00:00')));
select * from t where ts > '2020-04-18 02:00:42.123';
```

14.11.12.4 分区选择

SELECT 语句中支持分区选择。实现通过使用一个 PARTITION 选项实现。

```
SET @@sql_mode = '';

CREATE TABLE employees  (
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    fname VARCHAR(25) NOT NULL,
    lname VARCHAR(25) NOT NULL,
    store_id INT NOT NULL,
    department_id INT NOT NULL
)

PARTITION BY RANGE(id)  (
    PARTITION p0 VALUES LESS THAN (5),
    PARTITION p1 VALUES LESS THAN (10),
    PARTITION p2 VALUES LESS THAN (15),
    PARTITION p3 VALUES LESS THAN MAXVALUE
);

INSERT INTO employees VALUES
    ('', 'Bob', 'Taylor', 3, 2), ('', 'Frank', 'Williams', 1, 2),
    ('', 'Ellen', 'Johnson', 3, 4), ('', 'Jim', 'Smith', 2, 4),
    ('', 'Mary', 'Jones', 1, 1), ('', 'Linda', 'Black', 2, 3),
    ('', 'Ed', 'Jones', 2, 1), ('', 'June', 'Wilson', 3, 1),
    ('', 'Andy', 'Smith', 1, 3), ('', 'Lou', 'Waters', 2, 4),
    ('', 'Jill', 'Stone', 1, 4), ('', 'Roger', 'White', 3, 2),
    ('', 'Howard', 'Andrews', 1, 2), ('', 'Fred', 'Goldberg', 3, 3),
    ('', 'Barbara', 'Brown', 2, 3), ('', 'Alice', 'Rogers', 2, 2),
    ('', 'Mark', 'Morgan', 3, 3), ('', 'Karen', 'Cole', 3, 2);
```

你可以查看存储在分区 p1 中的行：

```
SELECT * FROM employees PARTITION (p1);
```

```
+----|-------|--------|----------|---------------+
| id | fname | lname  | store_id | department_id |
+----|-------|--------|----------|---------------+
|  5 | Mary  | Jones  |        1 |             1 |
|  6 | Linda | Black  |        2 |             3 |
|  7 | Ed    | Jones  |        2 |             1 |
|  8 | June  | Wilson |        3 |             1 |
|  9 | Andy  | Smith  |        1 |             3 |
+----|-------|--------|----------|---------------+
5 rows in set (0.00 sec)
```

如果希望获得多个分区中的行，可以提供分区名的列表，用逗号隔开。例如，`SELECT * FROM employees PARTITION (p1, p2)` 返回分区 p1 和 p2 的所有行。

使用分区选择时，仍然可以使用 where 条件，以及 ORDER BY 和 LIMIT 等选项。使用 HAVING 和 GROUP BY 等聚合选项也是支持的。

```
SELECT * FROM employees PARTITION (p0, p2)
    WHERE lname LIKE 'S%';
```

```
+----|-------|-------|----------|---------------+
| id | fname | lname | store_id | department_id |
+----|-------|-------|----------|---------------+
|  4 | Jim   | Smith |        2 |             4 |
| 11 | Jill  | Stone |        1 |             4 |
+----|-------|-------|----------|---------------+
2 rows in set (0.00 sec)
```

```
SELECT id, CONCAT(fname, ' ', lname) AS name
    FROM employees PARTITION (p0) ORDER BY lname;
```

```
+----|----------------+
| id | name           |
+----|----------------+
|  3 | Ellen Johnson  |
|  4 | Jim Smith      |
|  1 | Bob Taylor     |
|  2 | Frank Williams |
+----|----------------+
4 rows in set (0.06 sec)
```

```
SELECT store_id, COUNT(department_id) AS c
```

```
    FROM employees PARTITION (p1,p2,p3)
    GROUP BY store_id HAVING c > 4;
```

```
+---|----------+
| c | store_id |
+---|----------+
| 5 |        2 |
| 5 |        3 |
+---|----------+
2 rows in set (0.00 sec)
```

分支选择支持所有类型的分区表，无论是 Range 分区或是 Hash 分区等。对于 Hash 分区，如果没有指定分区名，会自动使用 p0、p1、p2、……、或 pN-1 作为分区名。

在 INSERT ... SELECT 的 SELECT 中也是可以使用分区选择的。

14.11.12.5 分区的约束和限制

本节介绍当前 TiDB 分区表的一些约束和限制。

14.11.12.5.1 分区键，主键和唯一键

本节讨论分区键，主键和唯一键之间的关系。一句话总结它们之间的关系要满足的规则：分区表的每个唯一键，必须包含分区表达式中用到的所有列。

> every unique key on the table must use every column in the table’s partitioning expression.

这里所指的唯一也包含了主键，因为根据主键的定义，主键必须是唯一的。例如，下面这些建表语句就是无效的：

```
CREATE TABLE t1 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    UNIQUE KEY (col1, col2)
)

PARTITION BY HASH(col3)
PARTITIONS 4;

CREATE TABLE t2 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
```

```
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    UNIQUE KEY (col1),
    UNIQUE KEY (col3)
)

PARTITION BY HASH(col1 + col3)
PARTITIONS 4;
```

它们都是有唯一键但没有包含所有分区键的。

下面是一些合法的语句的例子：

```
CREATE TABLE t1 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    UNIQUE KEY (col1, col2, col3)
)

PARTITION BY HASH(col3)
PARTITIONS 4;

CREATE TABLE t2 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    UNIQUE KEY (col1, col3)
)

PARTITION BY HASH(col1 + col3)
PARTITIONS 4;
```

下例中会产生一个报错：

```
CREATE TABLE t3 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    UNIQUE KEY (col1, col2),
    UNIQUE KEY (col3)
)

PARTITION BY HASH(col1 + col3)
```

```
    PARTITIONS 4;
```

```
ERROR 1491 (HY000): A PRIMARY KEY must include all columns in the table's partitioning function
```

原因是 `col1` 和 `col3` 出现在分区键中，但是几个唯一键定义并没有完全包含它们，做如下修改后语句即为合法：

```
CREATE TABLE t3 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    UNIQUE KEY (col1, col2, col3),
    UNIQUE KEY (col1, col3)
)

PARTITION BY HASH(col1 + col3)
    PARTITIONS 4;
```

下面这个表就没法做分区了，因为无论如何都不可能找到满足条件的分区键：

```
CREATE TABLE t4 (
    col1 INT NOT NULL,
    col2 INT NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    UNIQUE KEY (col1, col3),
    UNIQUE KEY (col2, col4)
);
```

根据定义，主键也是唯一键，下面两个建表语句是无效的：

```
CREATE TABLE t5 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    PRIMARY KEY(col1, col2)
)

PARTITION BY HASH(col3)
PARTITIONS 4;

CREATE TABLE t6 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
```

```
    col4 INT NOT NULL,
    PRIMARY KEY(col1, col3),
    UNIQUE KEY(col2)
)

PARTITION BY HASH( YEAR(col2) )
PARTITIONS 4;
```

以上两个例子中，主键都没有包含分区表达式中的全部的列，在主键中补充缺失列后语句即为合法：

```
CREATE TABLE t5 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    PRIMARY KEY(col1, col2, col3)
)

PARTITION BY HASH(col3)
PARTITIONS 4;

CREATE TABLE t6 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    PRIMARY KEY(col1, col2, col3),
    UNIQUE KEY(col2)
)

PARTITION BY HASH( YEAR(col2) )
PARTITIONS 4;
```

如果既没有主键，也没有唯一键，则不存在这个限制。

DDL 变更时，添加唯一索引也需要考虑到这个限制。比如创建了这样一个表：

```
CREATE TABLE t_no_pk (c1 INT, c2 INT)
    PARTITION BY RANGE(c1) (
        PARTITION p0 VALUES LESS THAN (10),
        PARTITION p1 VALUES LESS THAN (20),
        PARTITION p2 VALUES LESS THAN (30),
        PARTITION p3 VALUES LESS THAN (40)
    );
```

```
Query OK, 0 rows affected (0.12 sec)
```

通过 ALTER TABLE 添加非唯一索引是可以的。但是添加唯一索引时，唯一索引里面必须包含 c1 列。

使用分区表时，前缀索引是不能指定为唯一属性的：

```
CREATE TABLE t (a varchar(20), b blob,
    UNIQUE INDEX (a(5)))
    PARTITION by range columns (a) (
    PARTITION p0 values less than ('aaaaa'),
    PARTITION p1 values less than ('bbbbb'),
    PARTITION p2 values less than ('ccccc'));
```

```
ERROR 1503 (HY000): A UNIQUE INDEX must include all columns in the table's partitioning function
```

14.11.12.5.2 关于函数的分区限制

只有以下函数可以用于分区表达式：

```
ABS()
CEILING()
DATEDIFF()
DAY()
DAYOFMONTH()
DAYOFWEEK()
DAYOFYEAR()
EXTRACT() (see EXTRACT() function with WEEK specifier)
FLOOR()
HOUR()
MICROSECOND()
MINUTE()
MOD()
MONTH()
QUARTER()
SECOND()
TIME_TO_SEC()
TO_DAYS()
TO_SECONDS()
UNIX_TIMESTAMP() (with TIMESTAMP columns)
WEEKDAY()
YEAR()
YEARWEEK()
```

14.11.12.5.3 兼容性

目前 TiDB 支持 Range 分区、Range Columns 分区、List 分区、List COLUMNS 分区、Hash 分区和 Key 分区，其它的 MySQL 分区类型尚不支持。

对于 Key 分区，目前不支持分区字段为空的场景。

分区管理方面，只要底层实现可能会涉及数据挪动的操作，目前都暂不支持。包括且不限于：调整 Hash 分区表的分区数量，修改 Range 分区表的范围，合并分区等。

对于暂不支持的分区类型，在 TiDB 中建表时会忽略分区信息，以普通表的形式创建，并且会报 Warning。

Load Data 暂时不支持分区选择。

```
create table t (id int, val int) partition by hash(id) partitions 4;
```

普通的 Load Data 操作在 TiDB 中是支持的，如下：

```
load local data infile "xxx" into t ...
```

但 Load Data 不支持分区选择操作：

```
load local data infile "xxx" into t partition (p1)...
```

对于分区表，`select * from t` 的返回结果是分区之间无序的。这跟 MySQL 不同，MySQL 的返回结果是分区之间有序，分区内部无序。

```
create table t (id int, val int) partition by range (id) (
    partition p0 values less than (3),
    partition p1 values less than (7),
    partition p2 values less than (11));
```

```
Query OK, 0 rows affected (0.10 sec)
```

```
insert into t values (1, 2), (3, 4),(5, 6),(7,8),(9,10);
```

```
Query OK, 5 rows affected (0.01 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

TiDB 每次返回结果会不同，例如：

```
select * from t;
```

```
+------|------+
| id   | val  |
+------|------+
|    7 |    8 |
|    9 |   10 |
|    1 |    2 |
|    3 |    4 |
|    5 |    6 |
+------|------+
5 rows in set (0.00 sec)
```

MySQL 的返回结果：

```
select * from t;
```

```
+------|------+
| id   | val  |
+------|------+
|    1 |    2 |
|    3 |    4 |
|    5 |    6 |
|    7 |    8 |
|    9 |   10 |
+------|------+
5 rows in set (0.00 sec)
```

环境变量 tidb_enable_list_partition 可以控制是否启用分区表功能。如果该变量设置为 OFF，则建表时会忽略分区信息，以普通表的方式建表。

该变量仅作用于建表，已经建表之后再修改该变量无效。详见系统变量和语法。

14.11.12.5.4 动态裁剪模式

TiDB 访问分区表有两种模式，dynamic 和 static。从 v6.3.0 开始，默认使用 dynamic 模式。但是注意，dynamic 模式仅在表级别汇总统计信息（即 GlobalStats）收集完成的情况下生效。如果选择了 dynamic 但 GlobalStats 未收集完成，TiDB 会仍采用 static 模式。关于 GlobalStats 更多信息，请参考动态裁剪模式下的分区表统计信息。

```
set @@session.tidb_partition_prune_mode = 'dynamic'
```

普通查询和手动 analyze 使用的是 session 级别的 tidb_partition_prune_mode 设置，后台的 auto-analyze 使用的是 global 级别的 tidb_partition_prune_mode 设置。

静态裁剪模式下，分区表使用的是分区级别的统计信息，而动态裁剪模式下，分区表用的是表级别的汇总统计信息。

从 static 静态裁剪模式切到 dynamic 动态裁剪模式时，需要手动检查和收集统计信息。在刚切换到 dynamic 时，分区表上仍然只有分区的统计信息，需要等到全局 dynamic 动态裁剪模式开启后的下一次 auto-analyze 周期，才会更新生成汇总统计信息。

```
set session tidb_partition_prune_mode = 'dynamic';
show stats_meta where table_name like "t";
```

```
+---------+------------+----------------+---------------------+--------------+-----------+
| Db_name | Table_name | Partition_name | Update_time         | Modify_count | Row_count |
+---------+------------+----------------+---------------------+--------------+-----------+
| test    | t          | p0             | 2022-05-27 20:23:34 |            1 |         2 |
| test    | t          | p1             | 2022-05-27 20:23:34 |            2 |         4 |
| test    | t          | p2             | 2022-05-27 20:23:34 |            2 |         4 |
+---------+------------+----------------+---------------------+--------------+-----------+
3 rows in set (0.01 sec)
```

为保证开启全局 dynamic 动态裁剪模式时，SQL 可以用上正确的统计信息，此时需要手动触发一次 analyze 来更新汇总统计信息，可以通过 analyze 表或者单个分区来更新。

```
analyze table t partition p1;
show stats_meta where table_name like "t";
```

```
+---------+------------+----------------+---------------------+--------------+-----------+
| Db_name | Table_name | Partition_name | Update_time         | Modify_count | Row_count |
+---------+------------+----------------+---------------------+--------------+-----------+
| test    | t          | global         | 2022-05-27 20:50:53 |            0 |         5 |
| test    | t          | p0             | 2022-05-27 20:23:34 |            1 |         2 |
| test    | t          | p1             | 2022-05-27 20:50:52 |            0 |         2 |
| test    | t          | p2             | 2022-05-27 20:50:08 |            0 |         2 |
+---------+------------+----------------+---------------------+--------------+-----------+
4 rows in set (0.00 sec)
```

若 analyze 过程中提示如下 warning，说明分区的统计信息之间存在不一致，需要重新收集分区或整个表统计信息。

```
| Warning | 8244 | Build table: `t` column: `a` global-level stats failed due to missing
    ↪ partition-level column stats, please run analyze table to refresh columns of all
    ↪ partitions
```

也可以使用脚本来统一更新所有的分区表统计信息，详见为动态裁剪模式更新所有分区表的统计信息。

表级别统计信息准备好后，即可开启全局的动态裁剪模式。全局动态裁剪模式，对全局所有的 SQL 和对后台的统计信息自动收集（即 auto analyze）起作用。

```
set global tidb_partition_prune_mode = dynamic
```

在 static 模式下，TiDB 用多个算子单独访问每个分区，然后通过 Union 将结果合并起来。下面例子进行了一个简单的读取操作，可以发现 TiDB 用 Union 合并了对应两个分区的结果：

```
mysql> create table t1(id int, age int, key(id)) partition by range(id) (
          partition p0 values less than (100),
          partition p1 values less than (200),
          partition p2 values less than (300),
          partition p3 values less than (400));
Query OK, 0 rows affected (0.01 sec)

mysql> explain select * from t1 where id < 150;
```

```
+------------------------------+----------+-----------+-----------------------+-------------------------------
    ↪
| id                           | estRows  | task      | access object         | operator info
    ↪                  |
+------------------------------+----------+-----------+-----------------------+-------------------------------
    ↪
```

```
| PartitionUnion_9              | 6646.67  | root      |                         |
    ↪                                  |
| ├─TableReader_12              | 3323.33  | root      |                         | data:
    ↪ Selection_11               |
| │ └─Selection_11              | 3323.33  | cop[tikv] |                         | lt(test.t1.id,
    ↪ 150)            |
| │   └─TableFullScan_10        | 10000.00 | cop[tikv] | table:t1, partition:p0 | keep order:
    ↪ false, stats:pseudo |
| └─TableReader_18              | 3323.33  | root      |                         | data:
    ↪ Selection_17               |
|   └─Selection_17              | 3323.33  | cop[tikv] |                         | lt(test.t1.id,
    ↪ 150)            |
|     └─TableFullScan_16        | 10000.00 | cop[tikv] | table:t1, partition:p1 | keep order:
    ↪ false, stats:pseudo |
+-------------------------------+----------+-----------+-------------------------+-------------------------------
    ↪
7 rows in set (0.00 sec)
```

在 dynamic 模式下，每个算子都支持直接访问多个分区，所以 TiDB 不再使用 Union。

```
mysql> set @@session.tidb_partition_prune_mode = 'dynamic';
Query OK, 0 rows affected (0.00 sec)

mysql> explain select * from t1 where id < 150;
+--
    ↪ ----------------------+----------+-----------+-----------------+-----------------------------+
    ↪
| id                      | estRows  | task      | access object   | operator info
    ↪                  |
+--
    ↪ ----------------------+----------+-----------+-----------------+-----------------------------+
    ↪
| TableReader_7           | 3323.33  | root      | partition:p0,p1 | data:Selection_6
    ↪               |
| └─Selection_6           | 3323.33  | cop[tikv] |                 | lt(test.t1.id, 150)
    ↪            |
|   └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t1        | keep order:false, stats:
    ↪ pseudo |
+--
    ↪ ----------------------+----------+-----------+-----------------+-----------------------------+
    ↪
3 rows in set (0.00 sec)
```

从以上查询结果可知，执行计划中的 Union 消失了，分区裁剪依然生效，且执行计划只访问了 p0 和 p1 两个分区。

dynamic 模式让执行计划更简单清晰，省略 Union 操作可提高执行效率，还可避免 Union 并发管理的问题。此外 dynamic 模式下，执行计划可以使用 IndexJoin 的方式，这在 static 模式下是无法实现的。请看下面的例子：

示例一：以下示例在 static 模式下执行计划带 IndexJoin 的查询。

```
mysql> create table t1 (id int, age int, key(id)) partition by range(id)
          (partition p0 values less than (100),
           partition p1 values less than (200),
           partition p2 values less than (300),
           partition p3 values less than (400));
Query OK, 0 rows affected (0,08 sec)
mysql> create table t2 (id int, code int);

Query OK, 0 rows affected (0.01 sec)

mysql> set @@tidb_partition_prune_mode = 'static';
Query OK, 0 rows affected (0.00 sec)

mysql> explain select /*+ TIDB_INLJ(t1, t2) */ t1.* from t1, t2 where t2.code = 0 and t2.id = t1.
    ↪ id;
+--
    ↪ ------------------------------+----------+-----------+-----------------------+-----------------------------
    ↪
| id                            | estRows  | task      | access object         | operator info
    ↪                                |
+--
    ↪ ------------------------------+----------+-----------+-----------------------+-----------------------------
    ↪
| HashJoin_13                   | 12.49    | root      |                       | inner join,
    ↪ equal:[eq(test.t1.id, test.t2.id)] |
| ├─TableReader_42(Build)       | 9.99     | root      |                       | data:
    ↪ Selection_41                           |
| │ └─Selection_41              | 9.99     | cop[tikv] |                       | eq(test.t2.
    ↪ code, 0), not(isnull(test.t2.id))   |
| │   └─TableFullScan_40        | 10000.00 | cop[tikv] | table:t2              | keep order:
    ↪ false, stats:pseudo                |
| └─PartitionUnion_15(Probe)    | 39960.00 | root      |                       |
    ↪                                                 |
|   ├─TableReader_18            | 9990.00  | root      |                       | data:
    ↪ Selection_17                           |
|   │ └─Selection_17            | 9990.00  | cop[tikv] |                       | not(isnull(
    ↪ test.t1.id))                       |
|   │   └─TableFullScan_16      | 10000.00 | cop[tikv] | table:t1, partition:p0 | keep order:
    ↪ false, stats:pseudo                |
|   ├─TableReader_24            | 9990.00  | root      |                       | data:
    ↪ Selection_23                           |
```

```
|   |  └─Selection_23             | 9990.00  | cop[tikv] |                         | not(isnull(
    ↪ test.t1.id))                          |
|   |    └─TableFullScan_22       | 10000.00 | cop[tikv] | table:t1, partition:p1 | keep order:
    ↪ false, stats:pseudo                   |
|   ├─TableReader_30              | 9990.00  | root      |                         | data:
    ↪ Selection_29                             |
|   |  └─Selection_29             | 9990.00  | cop[tikv] |                         | not(isnull(
    ↪ test.t1.id))                          |
|   |    └─TableFullScan_28       | 10000.00 | cop[tikv] | table:t1, partition:p2 | keep order:
    ↪ false, stats:pseudo                   |
|   └─TableReader_36              | 9990.00  | root      |                         | data:
    ↪ Selection_35                             |
|     └─Selection_35              | 9990.00  | cop[tikv] |                         | not(isnull(
    ↪ test.t1.id))                          |
|       └─TableFullScan_34        | 10000.00 | cop[tikv] | table:t1, partition:p3 | keep order:
    ↪ false, stats:pseudo                   |
+--
    ↪ ------------------------------+----------+-----------+-------------------------+-------------------------
    ↪
17 rows in set, 1 warning (0.00 sec)

mysql> show warnings;
+--
    ↪ -------+------+-----------------------------------------------------------------------------------------+
    ↪
| Level   | Code | Message
    ↪                                                                                                  |
+--
    ↪ -------+------+-----------------------------------------------------------------------------------------+
    ↪
| Warning | 1815 | Optimizer Hint /*+ INL_JOIN(t1, t2) */ or /*+ TIDB_INLJ(t1, t2) */ is
    ↪ inapplicable |
+--
    ↪ -------+------+-----------------------------------------------------------------------------------------+
    ↪
1 row in set (0,00 sec)
```

从以上示例一结果可知，即使使用了 TIDB_INLJ 的 hint，也无法使得带分区表的查询选上带 IndexJoin 的执行计划。

示例二：以下示例在 dynamic 模式下尝试执行计划带 IndexJoin 的查询。

```
mysql> set @@tidb_partition_prune_mode = 'dynamic';
Query OK, 0 rows affected (0.00 sec)

mysql> explain select /*+ TIDB_INLJ(t1, t2) */ t1.* from t1, t2 where t2.code = 0 and t2.id = t1.
```

```
    ↪ id;
+--
    ↪ ------------------------------+----------+-----------+-----------------------+----------------------------
    ↪
| id                            | estRows  | task      | access object         | operator info
    ↪
    ↪
    ↪ |
+--
    ↪ ------------------------------+----------+-----------+-----------------------+----------------------------
    ↪
| IndexJoin_11                  | 12.49    | root      |                       | inner join,
    ↪ inner:IndexLookUp_10, outer key:test.t2.id, inner key:test.t1.id, equal cond:eq(test.t2.
    ↪ id, test.t1.id) |
| ├─TableReader_16(Build)       | 9.99     | root      |                       | data:
    ↪ Selection_15
    ↪
    ↪ |
| │ └─Selection_15              | 9.99     | cop[tikv] |                       | eq(test.t2.
    ↪ code, 0), not(isnull(test.t2.id))
    ↪                                                                                 |
| │   └─TableFullScan_14        | 10000.00 | cop[tikv] | table:t2              | keep order:
    ↪ false, stats:pseudo
    ↪                                                                                       |
| └─IndexLookUp_10(Probe)       | 12.49    | root      | partition:all         |
    ↪
    ↪ |
|   ├─Selection_9(Build)        | 12.49    | cop[tikv] |                       | not(isnull(
    ↪ test.t1.id))
    ↪
    ↪ |
|   │ └─IndexRangeScan_7        | 12.50    | cop[tikv] | table:t1, index:id(id) | range:
    ↪ decided by [eq(test.t1.id, test.t2.id)], keep order:false, stats:pseudo
    ↪                                       |
|   └─TableRowIDScan_8(Probe)   | 12.49    | cop[tikv] | table:t1              | keep order:
    ↪ false, stats:pseudo
    ↪                                                                               |
+--
    ↪ ------------------------------+----------+-----------+-----------------------+----------------------------
    ↪
8 rows in set (0.00 sec)
```

从示例二结果可知，开启 dynamic 模式后，带 IndexJoin 的计划在执行查询时被选上。

目前，静态和动态裁剪模式都不支持执行计划缓存。

为动态裁剪模式更新所有分区表的统计信息

1. 找到所有的分区表：

```
SELECT DISTINCT CONCAT(TABLE_SCHEMA,'.', TABLE_NAME)
    FROM information_schema.PARTITIONS
    WHERE TIDB_PARTITION_ID IS NOT NULL
    AND TABLE_SCHEMA NOT IN ('INFORMATION_SCHEMA', 'mysql', 'sys', 'PERFORMANCE_SCHEMA', '
        ↪ METRICS_SCHEMA');
```

```
+-------------------------------------+
| concat(TABLE_SCHEMA,'.',TABLE_NAME) |
+-------------------------------------+
| test.t                              |
+-------------------------------------+
1 row in set (0.02 sec)
```

2. 生成所有分区表的更新统计信息的语句：

```
SELECT DISTINCT CONCAT('ANALYZE TABLE ',TABLE_SCHEMA,'.',TABLE_NAME,' ALL COLUMNS;')
    FROM information_schema.PARTITIONS
    WHERE TIDB_PARTITION_ID IS NOT NULL
    AND TABLE_SCHEMA NOT IN ('INFORMATION_SCHEMA','mysql','sys','PERFORMANCE_SCHEMA','
        ↪ METRICS_SCHEMA');
```

```
+----------------------------------------------------------------------+
| concat('ANALYZE TABLE ',TABLE_SCHEMA,'.',TABLE_NAME,' ALL COLUMNS;') |
+----------------------------------------------------------------------+
| ANALYZE TABLE test.t ALL COLUMNS;                                    |
+----------------------------------------------------------------------+
1 row in set (0.01 sec)
```

可以按需将 `ALL COLUMNS` 改为实际需要的列。

3. 将批量更新语句导出到文件：

```
mysql --host xxxx --port xxxx -u root -p -e "SELECT DISTINCT CONCAT('ANALYZE TABLE ',
    ↪ TABLE_SCHEMA,'.',TABLE_NAME,' ALL COLUMNS;') \
    FROM information_schema.PARTITIONS \
    WHERE TIDB_PARTITION_ID IS NOT NULL \
    AND TABLE_SCHEMA NOT IN ('INFORMATION_SCHEMA','mysql','sys','PERFORMANCE_SCHEMA','
        ↪ METRICS_SCHEMA');" | tee gatherGlobalStats.sql
```

4. 执行批量更新：

在运行 source 命令之前处理 SQL 文件：

```
sed -i "" '1d' gatherGlobalStats.sql --- mac
sed -i '1d' gatherGlobalStats.sql --- linux
```

```
SET session tidb_partition_prune_mode = dynamic;
source gatherGlobalStats.sql
```

14.11.13 临时表

TiDB 在 v5.3.0 版本中引入了临时表功能。该功能针对业务中间计算结果的临时存储问题，让用户免于频繁地建表和删表等操作。用户可将业务上的中间计算数据存入临时表，用完数据后 TiDB 自动清理回收临时表。这避免了用户业务过于复杂，减少了表管理开销，并提升了性能。

本文介绍了 TiDB 临时表的使用场景、临时表类型、使用示例、限制临时表内存占用的方法、与其他 TiDB 功能的兼容性限制。

14.11.13.1 使用场景

TiDB 临时表主要应用于以下业务场景：

- 缓存业务的中间临时数据，计算完成后将数据转储至普通表，临时表会自动释放。
- 短期内对同一数据进行多次 DML 操作。例如在电商购物车应用中，添加、修改、删除商品及完成结算，并移除购物车信息。
- 快速批量导入中间临时数据，提升导入临时数据的性能。
- 批量更新数据。将数据批量导入到数据库的临时表，修改完成后再导出到文件。

14.11.13.2 临时表类型

TiDB 的临时表分为本地临时表和全局临时表：

- 本地临时表的表定义和表内数据只对当前会话可见，适用于暂存会话内的中间数据。
- 全局临时表的表定义对整个 TiDB 集群可见，表内数据只对当前事务可见，适用于暂存事务内的中间数据。

14.11.13.3 本地临时表

本地临时表的语义与 MySQL 临时表一致，它有以下特性：

- 本地临时表的表定义不持久化，只在创建该表的会话内可见，其他会话无法访问该本地临时表
- 不同会话可以创建同名的本地临时表，各会话只会读写该会话内创建的本地临时表
- 本地临时表的数据对会话内的所有事务可见
- 在会话结束后，该会话创建的本地临时表会被自动删除
- 本地临时表可以与普通表同名，此时在 DDL 和 DML 语句中，普通表被隐藏，直到本地临时表被删除

用户可通过 `CREATE TEMPORARY TABLE` 语句创建本地临时表，通过 `DROP TABLE` 或 `DROP TEMPORARY TABLE` 语句删除本地临时表。

不同于 MySQL，TiDB 本地临时表都是外部表，SQL 语句不会创建内部临时表。

14.11.13.3.1 本地临时表使用示例

> **注意：**
>
> - 使用 TiDB 中的临时表前，注意临时表与其他 TiDB 功能的兼容性限制以及与 MySQL 临时表的兼容性。
> - 如果在 v5.3.0 升级前创建了本地临时表，这些临时表实际为普通表，在升级后也会被 TiDB 当成普通表处理。

假设已存在普通表 users:

```
CREATE TABLE users (
    id BIGINT,
    name VARCHAR(100),
    PRIMARY KEY(id)
);
```

在会话 A 中创建本地临时表 users，不会有名字冲突。会话 A 访问 users 时，访问的是本地临时表 users。

```
CREATE TEMPORARY TABLE users (
    id BIGINT,
    name VARCHAR(100),
    city VARCHAR(50),
    PRIMARY KEY(id)
);
```

```
Query OK, 0 rows affected (0.01 sec)
```

此时将数据插入 users，插入到的是会话 A 中的本地临时表 users。

```
INSERT INTO users(id, name, city) VALUES(1001, 'Davis', 'LosAngeles');
```

```
Query OK, 1 row affected (0.00 sec)
```

```
SELECT * FROM users;
```

```
+------+-------+------------+
| id   | name  | city       |
+------+-------+------------+
| 1001 | Davis | LosAngeles |
+------+-------+------------+
1 row in set (0.00 sec)
```

在会话 B 中创建本地临时表 users，不会与普通表 users 冲突，也不会与会话 A 中的本地临时表 users 冲突。会话 B 内访问 users 时，访问的是会话 B 内创建的本地临时表 users 数据。

```
CREATE TEMPORARY TABLE users (
    id BIGINT,
    name VARCHAR(100),
    city VARCHAR(50),
    PRIMARY KEY(id)
);
```

```
Query OK, 0 rows affected (0.01 sec)
```

此时将数据插入 users，插入到的是会话 B 中的本地临时表 users。

```
INSERT INTO users(id, name, city) VALUES(1001, 'James', 'NewYork');
```

```
Query OK, 1 row affected (0.00 sec)
```

```
SELECT * FROM users;
```

```
+------+-------+---------+
| id   | name  | city    |
+------+-------+---------+
| 1001 | James | NewYork |
+------+-------+---------+
1 row in set (0.00 sec)
```

14.11.13.3.2 与 MySQL 临时表的兼容性

TiDB 本地临时表的以下特性与限制与 MySQL 一致：

- 创建、删除本地临时表时，不会自动提交当前事务
- 删除本地临时表所在的 schema 后，临时表不会被删除，仍然可以读写
- 创建本地临时表需要 CREATE TEMPORARY TABLES 权限，随后对该表的所有操作不需要权限
- 本地临时表不支持外键和分区表
- 不支持基于本地临时表创建视图
- SHOW [FULL] TABLES 不显示本地临时表

TiDB 本地临时表与 MySQL 临时表有以下方面不兼容：

- TiDB 本地临时表不支持 ALTER TABLE
- TiDB 本地临时表忽略 ENGINE 表选项，始终在 TiDB 内存中暂存临时表数据，并且有内存限制
- 当声明存储引擎为 MEMORY 时，TiDB 本地临时表没有 MEMORY 存储引擎的限制
- 当声明存储引擎为 INNODB 或 MYISAM 时，TiDB 本地临时表忽略 InnoDB 临时表特有的系统变量
- MySQL 不允许在同一条 SQL 中多次引用同一张临时表，而 TiDB 本地临时表没有该限制
- MySQL 中用于显示临时表的 information_schema.INNODB_TEMP_TABLE_INFO 表在 TiDB 中不存在。TiDB 暂无用于显示本地临时表的系统表。
- TiDB 没有内部临时表，MySQL 针对内部临时表的系统变量对 TiDB 不生效

14.11.13.4 全局临时表

全局临时表是 TiDB 的扩展功能，它有以下特性：

- 全局临时表的表定义会持久化，对所有会话可见
- 全局临时表的数据只对当前的事务内可见，事务结束后数据自动清空
- 全局临时表不能与普通表同名

用户可通过 `CREATE GLOBAL TEMPORARY TABLE` 语句创建全局临时表，语句末尾要加上 `ON COMMIT DELETE ROWS`。可通过 `DROP TABLE` 或 `DROP GLOBAL TEMPORARY TABLE` 语句删除全局临时表。

14.11.13.4.1 全局临时表使用示例

> 注意：
>
> - 使用 TiDB 中的临时表前，注意临时表与其他 TiDB 功能的兼容性限制。
> - 如果在 v5.3.0 或以上版本中创建了全局临时表，这些临时表在降级后会被当作普通表处理，导致数据错误。

在会话 A 中创建全局临时表 `users`：

```
CREATE GLOBAL TEMPORARY TABLE users (
    id BIGINT,
    name VARCHAR(100),
    city VARCHAR(50),
    PRIMARY KEY(id)
) ON COMMIT DELETE ROWS;
```

```
Query OK, 0 rows affected (0.01 sec)
```

写入 `users` 的数据对当前事务可见：

```
BEGIN;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
INSERT INTO users(id, name, city) VALUES(1001, 'Davis', 'LosAngeles');
```

```
Query OK, 1 row affected (0.00 sec)
```

```
SELECT * FROM users;
```

```
+------+-------+------------+
| id   | name  | city       |
+------+-------+------------+
| 1001 | Davis | LosAngeles |
+------+-------+------------+
1 row in set (0.00 sec)
```

事务结束后数据自动被清空：

```
COMMIT;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
SELECT * FROM users;
```

```
Empty set (0.00 sec)
```

会话 A 创建了 users 后，会话 B 也可以读写该表：

```
SELECT * FROM users;
```

```
Empty set (0.00 sec)
```

> 注意：
>
> 如果事务是自动提交的，插入的数据在 SQL 语句执行结束后会被自动清空，导致后续 SQL 执行查找不到结果。因此应该使用非自动提交的事务读写全局临时表。

14.11.13.5 限制临时表的内存占用

无论定义表时声明的 ENGINE 是哪种存储引擎，本地临时表和全局临时表的数据都只暂存在 TiDB 实例的内存中，不持久化。

为了避免内存溢出，用户可通过系统变量tidb_tmp_table_max_size 限制每张临时表的大小。当临时表大小超过限制后 TiDB 会报错。tidb_tmp_table_max_size 的默认值是 64MB。

例如，将每张临时表的大小限制为 256MB：

```
SET GLOBAL tidb_tmp_table_max_size=268435456;
```

14.11.13.6 与其他 TiDB 功能的兼容性限制

以下是本地临时表和全局临时表都不支持的功能：

- 不支持 AUTO_RANDOM 列
- 不支持 SHARD_ROW_ID_BITS 和 PRE_SPLIT_REGIONS 表选项
- 不支持分区表
- 不支持 SPLIT REGION 语句
- 不支持 ADMIN CHECK TABLE 和 ADMIN CHECKSUM TABLE 语句
- 不支持 FLASHBACK TABLE 和 RECOVER TABLE 语句
- 不支持以临时表为源表执行 CREATE TABLE LIKE 语句
- 不支持 Stale Read
- 不支持外键
- 不支持 SQL binding
- 不支持添加 TiFlash 副本
- 不支持在临时表上创建视图
- 不支持 Placement Rules
- 包含临时表的执行计划不会被 prepared plan cache 缓存

以下是只有本地临时表不支持的功能：

- 不支持通过系统变量 tidb_snapshot 读取历史数据

14.11.13.7 TiDB 数据迁移工具支持

本地临时表只对当前会话可见，因此本地临时表不会被 TiDB 数据迁移工具导出、备份、同步。

全局临时表的表定义全局可见，因此全局临时表的表定义会被 TiDB 数据迁移工具导出、备份、同步，但不导出数据。

> 注意：
>
> - TiCDC 必须使用 v5.3.0 及以上版本同步，否则下游集群的表定义错误
> - BR 必须使用 v5.3.0 及以上版本备份，否则备份后的表定义错误
> - 导入的集群、恢复后的集群、同步的下游集群需要支持全局临时表，否则报错

14.11.13.8 另请参阅

- CREATE TABLE
- CREATE TABLE LIKE
- DROP TABLE

14.11.14 缓存表

TiDB 在 v6.0.0 版本中引入了缓存表功能。该功能适用于频繁被访问且很少被修改的热点小表，即把整张表的数据加载到 TiDB 服务器的内存中，直接从内存中获取表数据，避免从 TiKV 获取表数据，从而提升读性能。

本文介绍了 TiDB 缓存表的使用场景、使用示例、与其他 TiDB 功能的兼容性限制。

14.11.14.1 使用场景

TiDB 缓存表功能适用于以下特点的表：

- 表的数据量不大
- 只读表，或者几乎很少修改
- 表的访问很频繁，期望有更好的读性能

当表的数据量不大，访问又特别频繁的情况下，数据会集中在 TiKV 一个 Region 上，形成热点，从而影响性能。因此，TiDB 缓存表的典型使用场景如下：

- 配置表，业务通过该表读取配置信息
- 金融场景中的存储汇率的表，该表不会实时更新，每天只更新一次
- 银行分行或者网点信息表，该表很少新增记录项

以配置表为例，当业务重启的瞬间，全部连接一起加载配置，会造成较高的数据库读延迟。如果使用了缓存表，则可以解决这样的问题。

14.11.14.2 使用示例

本节通过示例介绍缓存表的使用方法。

14.11.14.2.1 将普通表设为缓存表

假设已存在普通表 `users`:

```
CREATE TABLE users (
    id BIGINT,
    name VARCHAR(100),
    PRIMARY KEY(id)
);
```

通过 `ALTER TABLE` 语句，可以将这张表设置成缓存表：

```
ALTER TABLE users CACHE;
```

```
Query OK, 0 rows affected (0.01 sec)
```

14.11.14.2.2 验证是否为缓存表

要验证一张表是否为缓存表，使用 SHOW CREATE TABLE 语句。如果为缓存表，返回结果中会带有 CACHED ON 属性：

```
SHOW CREATE TABLE users;
```

```
+-------+-----------------------
```

而再次执行 trace，返回结果中不再有 regionRequest.SendReqCtx，表示 TiDB 已经不再从 TiKV 读取数据，而是直接从内存中读取：

```
+----------------------------------------+-----------------+------------+
| operation                              | startTS         | duration   |
+----------------------------------------+-----------------+------------+
| trace                                  | 17:47:40.533888 | 453.547µs  |
|   ├─session.ExecuteStmt                | 17:47:40.533894 | 402.341µs  |
|   │ ├─executor.Compile                 | 17:47:40.533903 | 205.54µs   |
|   │ └─session.runStmt                  | 17:47:40.534141 | 132.084µs  |
|   │   └─TableReaderExecutor.Open       | 17:47:40.534202 | 14.749µs   |
|   ├─*executor.UnionScanExec.Next       | 17:47:40.534306 | 3.21µs     |
|   └─*executor.UnionScanExec.Next       | 17:47:40.534316 | 1.219µs    |
+----------------------------------------+-----------------+------------+
7 rows in set (0.00 sec)
```

注意，读取缓存表会使用 UnionScan 算子，所以通过 explain 查看缓存表的执行计划时，可能会在结果中看到 UnionScan：

```
+--
    ↪ ----------------------+---------+-----------+---------------+------------------------------+
    ↪
| id                    | estRows | task      | access object | operator info
    ↪ |
+--
    ↪ ----------------------+---------+-----------+---------------+------------------------------+
    ↪
| UnionScan_5             | 1.00    | root      |               |
    ↪ |
| └─TableReader_7         | 1.00    | root      |               | data:TableFullScan_6
    ↪           |
|   └─TableFullScan_6     | 1.00    | cop[tikv] | table:users   | keep order:false, stats:pseudo
    ↪  |
+--
    ↪ ----------------------+---------+-----------+---------------+------------------------------+
    ↪
3 rows in set (0.00 sec)
```

14.11.14.2.3 往缓存表写入数据

缓存表支持写入数据。例如，往 users 表中插入一条记录：

```
INSERT INTO users(id, name) VALUES(1001, 'Davis');
```

```
Query OK, 1 row affected (0.00 sec)
```

```
SELECT * FROM users;
```

```
+------+-------+
| id   | name  |
+------+-------+
| 1001 | Davis |
+------+-------+
1 row in set (0.00 sec)
```

> 注意：
>
> 往缓存表写入数据时，有可能出现秒级别的写入延迟。延迟的时长由全局环境变量tidb_table_cache_lease 控制。你可根据实际业务能否承受此限制带来的延迟，决定是否适合使用缓存表功能。例如，对于完全只读的场景，可以将 tidb_table_cache_lease 调大：
>
> ```
> set @@global.tidb_table_cache_lease = 10;
> ```
>
> 缓存表的写入延时高是受到实现的限制。存在多个 TiDB 实例时，一个 TiDB 实例并不知道其它的 TiDB 实例是否缓存了数据，如果该实例直接修改了表数据，而其它 TiDB 实例依然读取旧的缓存数据，就会读到错误的结果。为了保证数据正确性，缓存表的实现使用了一套基于 lease 的复杂机制：读操作在缓存数据同时，还会对于缓存设置一个有效期，也就是 lease。在 lease 过期之前，无法对数据执行修改操作。因为修改操作必须等待 lease 过期，所以会出现写入延迟。

缓存表相关的元信息存储在 mysql.table_cache_meta 表中。这张表记录了所有缓存表的 ID、当前的锁状态 lock_type，以及锁租约 lease 相关的信息。这张表仅供 TiDB 内部使用，不建议用户修改该表，否则可能导致不可预期的错误。

```
SHOW CREATE TABLE mysql.table_cache_meta\G
*************************** 1. row ***************************
       Table: table_cache_meta
Create Table: CREATE TABLE `table_cache_meta` (
  `tid` bigint(11) NOT NULL DEFAULT '0',
  `lock_type` enum('NONE','READ','INTEND','WRITE') NOT NULL DEFAULT 'NONE',
  `lease` bigint(20) NOT NULL DEFAULT '0',
  `oldReadLease` bigint(20) NOT NULL DEFAULT '0',
  PRIMARY KEY (`tid`) /*T![clustered_index] CLUSTERED */
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin
1 row in set (0.00 sec)
```

14.11.14.2.4 将缓存表恢复为普通表

> **注意：**
>
> 对缓存表执行 DDL 语句会失败。若要对缓存表执行 DDL 语句，需要先去掉缓存属性，将缓存表设回普通表后，才能对其执行 DDL 语句。

```
TRUNCATE TABLE users;
```

```
ERROR 8242 (HY000): 'Truncate Table' is unsupported on cache tables.
```

```
mysql> ALTER TABLE users ADD INDEX k_id(id);
```

```
ERROR 8242 (HY000): 'Alter Table' is unsupported on cache tables.
```

使用 `ALTER TABLE t NOCACHE` 语句可以将缓存表恢复成普通表：

```
ALTER TABLE users NOCACHE
```

```
Query OK, 0 rows affected (0.00 sec)
```

14.11.14.3 缓存表大小限制

由于 TiDB 将整张缓存表的数据加载到 TiDB 进程的内存中，并且执行修改操作后缓存会失效，需要重新加载，所以 TiDB 缓存表只适用于表比较小的场景。

目前 TiDB 对于每张缓存表的大小限制为 64 MB。如果表的数据超过了 64 MB，执行 `ALTER TABLE t CACHE` 会失败。

14.11.14.4 与其他 TiDB 功能的兼容性限制

以下是缓存表不支持的功能：

- 不支持对分区表执行 `ALTER TABLE t CACHE` 操作
- 不支持对临时表执行 `ALTER TABLE t CACHE` 操作
- 不支持对视图执行 `ALTER TABLE t CACHE` 操作
- 不支持 Stale Read 功能
- 不支持对缓存表直接做 DDL 操作，需要先通过 `ALTER TABLE t NOCACHE` 将缓存表改回普通表后再进行 DDL 操作。

以下是缓存表无法使用缓存的场景：

- 设置系统变量 `tidb_snapshot` 读取历史数据
- 执行修改操作期间，已有缓存会失效，直到数据被再次加载

14.11.14.5 TiDB 数据迁移工具兼容性

缓存表并不是标准的 MySQL 功能，而是 TiDB 扩展。只有 TiDB 能识别 `ALTER TABLE ... CACHE` 语句。所有的 TiDB 数据迁移工具均不支持缓存表功能，包括 Backup & Restore (BR)、TiCDC、Dumpling 等组件，它们会将缓存表当作普通表处理。

这意味着，备份恢复一张缓存表时，它会变成一张普通表。如果下游集群是另一套 TiDB 集群并且你希望继续使用缓存表功能，可以对下游集群中的表执行 `ALTER TABLE ... CACHE` 手动开启缓存表功能。

14.11.14.6 另请参阅

- ALTER TABLE
- System Variables

14.11.15 外键约束

从 v6.6.0 开始，TiDB 支持外键以及外键约束功能，外键允许跨表交叉引用相关数据，外键约束则可以保证相关数据的一致性。

> **注意：**
>
> 外键功能通常适用于为中小规模的数据提供完整性和一致性约束校验，但是在大数据量和分布式数据库系统下，使用外键可能会导致严重的性能问题，并对系统产生不可预知的影响。如果计划使用外键，请进行充分验证后谨慎使用。

外键是在子表中定义的，语法如下：

```
ForeignKeyDef
         ::= ( 'CONSTRAINT' Identifier )? 'FOREIGN' 'KEY'
             Identifier? '(' ColumnName ( ',' ColumnName )* ')'
             'REFERENCES' TableName '(' ColumnName ( ',' ColumnName )* ')'
             ( 'ON' 'DELETE' ReferenceOption )?
             ( 'ON' 'UPDATE' ReferenceOption )?

ReferenceOption
         ::= 'RESTRICT'
           | 'CASCADE'
           | 'SET' 'NULL'
           | 'SET' 'DEFAULT'
           | 'NO' 'ACTION'
```

14.11.15.1 命名

外键的命名遵循以下规则：

- 如果在 CONSTRAINT identifier 语句中指定了名称，则使用该名称。
- 如果 CONSTRAINT identifier 语句未指定名称，但在 FOREIGN KEY identifier 语句中指定了名称，则使用 FOREIGN KEY identifier 定义的名称。
- 如果 CONSTRAINT identifier 和 FOREIGN KEY identifier 语句都没有指定名称，则会自动生成一个名称，例如 fk_1、fk_2、fk_3 等。
- 外键名称必须在当前表中唯一，否则创建时会报错 ERROR 1826: Duplicate foreign key constraint name 'fk'。

14.11.15.2 限制

创建外键时需要满足以下条件：

- 父表和子表都不能是临时表。
- 用户需要对父表有 REFERENCES 权限。
- 外键中的列和引用的父表中的列必须是相同的数据类型，并具有相同的大小、精度、长度、字符集 (charset) 和排序规则 (collation)。
- 外键中的列不能引用自身。
- 外键中的列和引用的父表中的列必须有相同的索引，并且索引中的列顺序必须与外键的列顺序一样，这样才能在执行外键约束检查时使用索引来避免全表扫描。
  - 如果父表中没有对应的外键索引，则会报错 ERROR 1822: Failed to add the foreign key constraint. Missing index for constraint 'fk' in the referenced table 't'。
  - 如果子表中没有对应的外键索引，则会自动创建一个索引，索引名和外键名一样。
- 不支持在 BLOB 和 TEXT 类型的列上创建外键。
- 不支持在分区表上创建外键。
- 不支持在虚拟生成列 (VIRTUAL GENERATED COLUMNS) 上创建外键。

14.11.15.3 引用操作

当 UPDATE 或 DELETE 操作影响父表中的外键值时，其在子表中相匹配的外键值取决于外键定义中 ON UPDATE 和 ON DELETE 定义的引用操作，引用操作包括：

- CASCADE：当 UPDATE 或 DELETE 父表中的行数据时，自动级联更新或删除子表中的匹配行数据。级联操作会用深度优先方式执行。
- SET NULL：当 UPDATE 或 DELETE 父表中的行数据时，自动将子表中匹配的外键列数据设置为 NULL。
- RESTRICT：如果子表中存在外键匹配的行数据，则拒绝 UPDATE 或 DELETE 父表的操作。
- NO ACTION：行为和 RESTRICT 一样。
- SET DEFAULT：行为和 RESTRICT 一样。

如果父表中没有匹配的外键值，则拒绝 INSERT 或 UPDATE 子表的操作。

如果外键定义中没有指定 ON DELETE 或者 ON UPDATE，则默认的行为是 NO ACTION。

如果外键是定义在 STORED GENERATED COLUMN 上的，则不支持使用 CASCADE、SET NULL 和 SET DEFAULT 引用操作。

14.11.15.4 外键使用示例

下面的示例通过单列外键关联父表和子表：

```
CREATE TABLE parent (
    id INT KEY
);

CREATE TABLE child (
    id INT,
    pid INT,
    INDEX idx_pid (pid),
    FOREIGN KEY (pid) REFERENCES parent(id) ON DELETE CASCADE
);
```

下面是一个更复杂的示例，其中 `product_order` 表有两个外键分别引用其他两个表。一个外键引用 `product` 表中的两列索引。另一个引用 `customer` 表中的单列索引：

```
CREATE TABLE product (
    category INT NOT NULL,
    id INT NOT NULL,
    price DECIMAL(20,10),
    PRIMARY KEY(category, id)
);

CREATE TABLE customer (
    id INT KEY
);

CREATE TABLE product_order (
    id INT NOT NULL AUTO_INCREMENT,
    product_category INT NOT NULL,
    product_id INT NOT NULL,
    customer_id INT NOT NULL,

    PRIMARY KEY(id),
    INDEX (product_category, product_id),
    INDEX (customer_id),

    FOREIGN KEY (product_category, product_id)
      REFERENCES product(category, id)
      ON UPDATE CASCADE ON DELETE RESTRICT,

    FOREIGN KEY (customer_id)
      REFERENCES customer(id)
);
```

14.11.15.5 新增外键约束

可以使用下面 ALTER TABLE 语句来新增一个外键约束：

```
ALTER TABLE table_name
    ADD [CONSTRAINT [identifier]] FOREIGN KEY
    [identifier] (col_name, ...)
    REFERENCES tbl_name (col_name,...)
    [ON DELETE reference_option]
    [ON UPDATE reference_option]
```

外键可以是自引用的，即引用同一个表。使用 ALTER TABLE 向表添加外键约束时，请先在外键引用父表的列上创建索引。

14.11.15.6 删除外键约束

可以使用下面 ALTER TABLE 语句来删除一个外键约束：

```
ALTER TABLE table_name DROP FOREIGN KEY fk_identifier;
```

如果外键约束在创建时定义了名称，则可以引用该名称来删除外键约束。否则，只能引用自动生成的约束名称进行删除。你可以使用 SHOW CREATE TABLE 查看外键名称：

```
mysql> SHOW CREATE TABLE child\G
*************************** 1. row ***************************
       Table: child
Create Table: CREATE TABLE `child` (
  `id` int(11) DEFAULT NULL,
  `pid` int(11) DEFAULT NULL,
  KEY `idx_pid` (`pid`),
  CONSTRAINT `fk_1` FOREIGN KEY (`pid`) REFERENCES `test`.`parent` (`id`) ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin
```

14.11.15.7 外键约束检查

TiDB 支持是否开启外键约束检查，由系统变量foreign_key_checks 控制，其默认值是 ON，即开启外键约束检查，它有 GLOBAL 和 SESSION 两种作用域。在一般的操作中保持该变量开启可以保证外键引用关系的完整性。

关闭外键约束检查的作用如下：

- 当删除一个被外键引用的父表时，只有关闭外键约束检查时才能删除成功。
- 当给数据库导入数据时，创建表的顺序可能和外键依赖顺序不一样而导致创建表报错，只有关闭外键约束检查时才能创建表成功，另外，导入数据时关闭外键约束检查也能加快导数据的速度。
- 当给数据库导入数据时，先导入子表的数据会报错，只有关闭外键约束检查，才能确保顺利导入子表数据。
- 执行有关外键的 ALTER TABLE 操作时，关闭外键约束检查才能执行成功。

当关闭关键约束检查时，不会执行外键约束检查以及引用操作，但以下场景除外：

- 如果执行 ALTER TABLE 会导致外键定义不正确，则依然会执行报错。
- 删除外键所需的索引时，需要先删除外键，否则删除外键会执行报错。
- 创建外键时，如果不符合外键的条件或限制，则依然会执行报错。

14.11.15.8 锁

在 INSERT 或者 UPDATE 子表时，外键约束会检查父表中是否存在对应的外键值，并对父表中的该行数据上锁，避免该外键值被其他操作删除，导致破坏外键约束。这里的上锁行为等同于对父表中外键值所在行做 SELECT FOR UPDATE 操作。

因为 TiDB 目前暂不支持 LOCK IN SHARE MODE，所以，在并发写入子表场景，如果引用的外键值大部分都一样，可能会有比较严重的锁冲突。建议在大批量写入子表数据时，关闭foreign_key_checks。

14.11.15.9 外键的定义和元信息

你可以使用SHOW CREATE TABLE 语句查看外键的定义：

```
mysql> SHOW CREATE TABLE child\G
*************************** 1. row ***************************
       Table: child
Create Table: CREATE TABLE `child` (
  `id` int(11) DEFAULT NULL,
  `pid` int(11) DEFAULT NULL,
  KEY `idx_pid` (`pid`),
  CONSTRAINT `fk_1` FOREIGN KEY (`pid`) REFERENCES `test`.`parent` (`id`) ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin
```

你可以使用以下任一系统表获取有关外键的信息：

- INFORMATION_SCHEMA.KEY_COLUMN_USAGE
- INFORMATION_SCHEMA.TABLE_CONSTRAINTS
- INFORMATION_SCHEMA.REFERENTIAL_CONSTRAINTS

下面提供了查询示例：

从 INFORMATION_SCHEMA.KEY_COLUMN_USAGE 系统表中获取有关的外键信息：

```
mysql> SELECT TABLE_SCHEMA, TABLE_NAME, COLUMN_NAME, CONSTRAINT_NAME FROM INFORMATION_SCHEMA.
    ↪ KEY_COLUMN_USAGE WHERE REFERENCED_TABLE_SCHEMA IS NOT NULL;
+--------------+---------------+------------------+-----------------+
| TABLE_SCHEMA | TABLE_NAME    | COLUMN_NAME      | CONSTRAINT_NAME |
+--------------+---------------+------------------+-----------------+
| test         | child         | pid              | fk_1            |
| test         | product_order | product_category | fk_1            |
| test         | product_order | product_id       | fk_1            |
| test         | product_order | customer_id      | fk_2            |
+--------------+---------------+------------------+-----------------+
```

从 INFORMATION_SCHEMA.TABLE_CONSTRAINTS 系统表中获取有关的外键信息：

```
mysql> SELECT * FROM INFORMATION_SCHEMA.TABLE_CONSTRAINTS WHERE CONSTRAINT_TYPE='FOREIGN KEY'\G
***************************[ 1. row ]***************************
CONSTRAINT_CATALOG | def
CONSTRAINT_SCHEMA  | test
CONSTRAINT_NAME    | fk_1
TABLE_SCHEMA       | test
TABLE_NAME         | child
CONSTRAINT_TYPE    | FOREIGN KEY
```

从 INFORMATION_SCHEMA.REFERENTIAL_CONSTRAINTS 系统表中获取有关的外键信息：

```
mysql> SELECT * FROM INFORMATION_SCHEMA.REFERENTIAL_CONSTRAINTS\G
***************************[ 1. row ]***************************
CONSTRAINT_CATALOG        | def
CONSTRAINT_SCHEMA         | test
CONSTRAINT_NAME           | fk_1
UNIQUE_CONSTRAINT_CATALOG | def
UNIQUE_CONSTRAINT_SCHEMA  | test
UNIQUE_CONSTRAINT_NAME    | PRIMARY
MATCH_OPTION              | NONE
UPDATE_RULE               | NO ACTION
DELETE_RULE               | CASCADE
TABLE_NAME                | child
REFERENCED_TABLE_NAME     | parent
```

14.11.15.10 查看带有外键的执行计划

你可以使用 EXPLAIN 语句查看执行计划。Foreign_Key_Check 算子是执行 DML 语句时，执行外键约束检查的算子。

```
mysql> explain insert into child values (1,1);
+-----------------------+---------+------+---------------+-------------------------------+
| id                    | estRows | task | access object | operator info                 |
+-----------------------+---------+------+---------------+-------------------------------+
| Insert_1              | N/A     | root |               | N/A                           |
| └─Foreign_Key_Check_3 | 0.00    | root | table:parent  | foreign_key:fk_1, check_exist |
+-----------------------+---------+------+---------------+-------------------------------+
```

你可以使用 EXPLAIN ANALYZE 语句查看外键引用行为的执行。Foreign_Key_Cascade 算子是执行 DML 语句时，执行外键引用行为的算子。

```
mysql> explain analyze delete from parent where id = 1;
+--
    ↪ -------------------------------+---------+---------+-----------+-------------------------------+------
    ↪
```

```
| id                              | estRows | actRows | task      | access object
    ↪                   | execution info
    ↪
    ↪ | operator info                                 | memory    | disk |
+--
    ↪ --------------------------------+---------+---------+-----------+-----------------------------------+------
    ↪
| Delete_2                        | N/A     | 0       | root      |
    ↪                                   | time:117.3µs, loops:1
    ↪
    ↪ | N/A                                           | 380 Bytes | N/A  |
| ├─Point_Get_1                    | 1.00    | 1       | root      | table:parent
    ↪                     | time:63.6µs, loops:2, Get:{num_rpc:1, total_time:29.9µs}
    ↪
    ↪ | handle:1                                      | N/A       | N/A  |
| └─Foreign_Key_Cascade_3          | 0.00    | 0       | root      | table:child, index:idx_pid
    ↪      | total:1.28ms, foreign_keys:1
    ↪
    ↪ | foreign_key:fk_1, on_delete:CASCADE           | N/A       | N/A  |
|   └─Delete_7                     | N/A     | 0       | root      |
    ↪                                   | time:904.8µs, loops:1
    ↪
    ↪ | N/A                                           | 1.11 KB   | N/A  |
|     └─IndexLookUp_11             | 10.00   | 1       | root      |
    ↪                                   | time:869.5µs, loops:2, index_task: {total_time: 371.1µs
    ↪ , fetch_handle: 357.3µs, build: 1.25µs, wait: 12.5µs}, table_task: {total_time: 382.6µs,
    ↪ num: 1, concurrency: 5}                     |
    ↪                                             | 9.13 KB   | N/A  |
|       ├─IndexRangeScan_9(Build)  | 10.00   | 1       | cop[tikv] | table:child, index:idx_pid(
    ↪ pid) | time:351.2µs, loops:3, cop_task: {num: 1, max: 282.3µs, proc_keys: 0, rpc_num: 1,
    ↪ rpc_time: 263µs, copr_cache_hit_ratio: 0.00, distsql_concurrency: 15}, tikv_task:{time
    ↪ :220.2µs, loops:0}   | range:[1,1], keep order:false, stats:pseudo | N/A       | N/A  |
|       └─TableRowIDScan_10(Probe) | 10.00   | 1       | cop[tikv] | table:child
    ↪                     | time:223.9µs, loops:2, cop_task: {num: 1, max: 168.8µs, proc_keys:
    ↪ 0, rpc_num: 1, rpc_time: 154.5µs, copr_cache_hit_ratio: 0.00, distsql_concurrency: 15},
    ↪ tikv_task:{time:145.6µs, loops:0} | keep order:false, stats:pseudo              | N/A
    ↪       | N/A  |
+--
    ↪ --------------------------------+---------+---------+-----------+-----------------------------------+------
    ↪
```

14.11.15.11 兼容性

14.11.15.11.1 TiDB 版本间兼容性

TiDB 在 v6.6.0 之前已经支持创建外键的语法，但创建的外键并不生效。如果将之前创建的 TiDB 集群升级到 v6.6.0 及之后的版本，之前创建的外键依然是不生效的，可以先删除不生效的外键后再创建外键使外键约束生效。只有在 v6.6.0 及之后版本中新创建的外键才生效。你可以使用 SHOW CREATE TABLE 语句查看外键是否生效，不生效的外键会有一条 /* FOREIGN KEY INVALID */ 注释。

```
mysql> SHOW CREATE TABLE child\G
***************************[ 1. row ]***************************
Table        | child
Create Table | CREATE TABLE `child` (
  `id` int(11) DEFAULT NULL,
  `pid` int(11) DEFAULT NULL,
  KEY `idx_pid` (`pid`),
  CONSTRAINT `fk_1` FOREIGN KEY (`pid`) REFERENCES `test`.`parent` (`id`) ON DELETE CASCADE /*
      ↪ FOREIGN KEY INVALID */
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin
```

14.11.15.11.2 与 TiDB 工具的兼容性

- TiDB Binlog 不支持外键功能。
- DM 不兼容外键功能。DM v6.6.0 在同步数据到下游 TiDB 时，会显式关闭下游 TiDB 的foreign_key_checks，所以由外键产生的级联操作不会从上游同步到下游，进而导致上下游数据不一致。这与 v6.6.0 之前版本 DM 的行为一致。
- TiCDC v6.6.0 兼容外键功能。旧版本的 TiCDC 在同步带外键的表时，可能会报错，建议使用 v6.6.0 之前版本 TiCDC 时先关闭下游 TiDB 集群的 foreign_key_checks。
- BR v6.6.0 兼容外键功能。之前版本的 BR 在恢复带外键的表到 v6.6.0 及之后版本的集群时，可能会报错，建议先关闭下游 TiDB 集群的 foreign_key_checks 后再恢复集群。
- TiDB Lightning 导入数据到 TiDB 前，建议先关闭 TiDB 集群的 foreign_key_checks。
- Dumpling 兼容外键功能。
- sync-diff-inspector 在对比上下游数据时，如果上下游数据库的版本不一样，且下游 TiDB 中存在不生效的外键，则 sync-diff-inspector 可能会报上下游表结构不一致的错误。因为 TiDB v6.6.0 会对表结构中不生效的外键添加一条 /* FOREIGN KEY INVALID */ 注释。

14.11.15.11.3 与 MySQL 的兼容性

创建外键未指定名称时，TiDB 自动生成的外键名称和 MySQL 不一样。例如 TiDB 生成的外键名称为 fk_1、fk_2、fk_3 等，MySQL 生成的外键名称为 table_name_ibfk_1、table_name_ibfk_2、table_name_ibfk_3 等。

MySQL 和 TiDB 均能解析但会忽略以内联 REFERENCES 的方式定义的外键。只有当 REFERENCES 作为 FOREIGN KEY 定义的一部分时，才会进行检查和执行。下面的示例在定义外键约束时只使用了 REFERENCES：

```
CREATE TABLE parent (
    id INT KEY
);

CREATE TABLE child (
    id INT,
```

```
    pid INT REFERENCES parent(id)
);

SHOW CREATE TABLE child;
```

输出结果显示 child 表不包含任何外键：

```
+-------+-----------------------------------------------------------------+
| Table | Create Table                                                    |
+-------+-----------------------------------------------------------------+
| child | CREATE TABLE `child` (                                          |
|       |   `id` int(11) DEFAULT NULL,                                    |
|       |   `pid` int(11) DEFAULT NULL                                    |
|       | ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin |
+-------+-----------------------------------------------------------------+
```

14.11.16 字符集和排序

14.11.16.1 字符集和排序规则

本文介绍了 TiDB 中支持的字符集和排序规则。

14.11.16.1.1 字符集和排序规则的概念

字符集 (character set) 是符号与编码的集合。TiDB 中的默认字符集是 utf8mb4，与 MySQL 8.0 及更高版本中的默认字符集匹配。

排序规则 (collation) 是在字符集中比较字符以及字符排序顺序的规则。例如，在二进制排序规则中，比较 A 和 a 的结果是不一样的：

```
SET NAMES utf8mb4 COLLATE utf8mb4_bin;
SELECT 'A' = 'a';
SET NAMES utf8mb4 COLLATE utf8mb4_general_ci;
SELECT 'A' = 'a';
```

```
SELECT 'A' = 'a';
```

```
+-----------+
| 'A' = 'a' |
+-----------+
|         0 |
+-----------+
1 row in set (0.00 sec)
```

```
SET NAMES utf8mb4 COLLATE utf8mb4_general_ci;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
SELECT 'A' = 'a';
```

```
+-----------+
| 'A' = 'a' |
+-----------+
|         1 |
+-----------+
1 row in set (0.00 sec)
```

TiDB 默认使用二进制排序规则。这一点与 MySQL 不同，MySQL 默认使用不区分大小写的排序规则。

14.11.16.1.2 支持的字符集和排序规则

目前 TiDB 支持以下字符集：

```
SHOW CHARACTER SET;
```

```
+---------+---------------------------------+--------------------+--------+
| Charset | Description                     | Default collation | Maxlen |
+---------+---------------------------------+--------------------+--------+
| ascii   | US ASCII                        | ascii_bin          |      1 |
| binary  | binary                          | binary             |      1 |
| gbk     | Chinese Internal Code Specification | gbk_bin            |      2 |
| latin1  | Latin1                          | latin1_bin         |      1 |
| utf8    | UTF-8 Unicode                   | utf8_bin           |      3 |
| utf8mb4 | UTF-8 Unicode                   | utf8mb4_bin        |      4 |
+---------+---------------------------------+--------------------+--------+
6 rows in set (0.00 sec)
```

TiDB 支持以下排序规则：

```
SHOW COLLATION;
```

```
+--------------------+---------+------+---------+----------+---------+
| Collation          | Charset | Id   | Default | Compiled | Sortlen |
+--------------------+---------+------+---------+----------+---------+
| ascii_bin          | ascii   |   65 | Yes     | Yes      |       1 |
| binary             | binary  |   63 | Yes     | Yes      |       1 |
| gbk_bin            | gbk     |   87 |         | Yes      |       1 |
| gbk_chinese_ci     | gbk     |   28 | Yes     | Yes      |       1 |
| latin1_bin         | latin1  |   47 | Yes     | Yes      |       1 |
| utf8_bin           | utf8    |   83 | Yes     | Yes      |       1 |
| utf8_general_ci    | utf8    |   33 |         | Yes      |       1 |
```

```
| utf8_unicode_ci    | utf8    |  192 |         | Yes      |       1 |
| utf8mb4_bin        | utf8mb4 |   46 | Yes     | Yes      |       1 |
| utf8mb4_general_ci | utf8mb4 |   45 |         | Yes      |       1 |
| utf8mb4_unicode_ci | utf8mb4 |  224 |         | Yes      |       1 |
+--------------------+---------+------+---------+----------+---------+
11 rows in set (0.00 sec)
```

> **警告：**
>
> TiDB 会错误地将 `latin1` 视为 `utf8` 的子集。当用户存储不同于 `latin1` 和 `utf8` 编码的字符时，可能会导致意外情况出现。因此强烈建议使用 `utf8mb4` 字符集。详情参阅 TiDB #18955。

> **注意：**
>
> TiDB 中的默认排序规则（后缀为 `_bin` 的二进制排序规则）与 MySQL 中的默认排序规则不同，后者通常是一般排序规则，后缀为 `_general_ci`。当用户指定了显式字符集，但依赖于待选的隐式默认排序规则时，这个差异可能导致兼容性问题。

利用以下的语句可以查看字符集对应的排序规则（以下是新的排序规则框架）下的结果：

```
SHOW COLLATION WHERE Charset = 'utf8mb4';
```

```
+--------------------+---------+------+---------+----------+---------+
| Collation          | Charset | Id   | Default | Compiled | Sortlen |
+--------------------+---------+------+---------+----------+---------+
| utf8mb4_bin        | utf8mb4 |   46 | Yes     | Yes      |       1 |
| utf8mb4_general_ci | utf8mb4 |   45 |         | Yes      |       1 |
| utf8mb4_unicode_ci | utf8mb4 |  224 |         | Yes      |       1 |
+--------------------+---------+------+---------+----------+---------+
3 rows in set (0.00 sec)
```

TiDB 对 GBK 字符集的支持详情见GBK。

14.11.16.1.3 TiDB 中的 utf8 和 utf8mb4

MySQL 限制字符集 `utf8` 为最多 3 个字节。这足以存储在基本多语言平面 (BMP) 中的字符，但不足以存储表情符号 (emoji) 等字符。因此，建议改用字符集`utf8mb4`。

默认情况下，TiDB 同样限制字符集 `utf8` 为最多 3 个字节，以确保 TiDB 中创建的数据可以在 MySQL 中顺利恢复。你可以禁用此功能，方法是将系统变量`tidb_check_mb4_value_in_utf8` 的值更改为 `OFF`。

以下示例演示了在表中插入 4 字节的表情符号字符（emoji 字符）时的默认行为。`utf8` 字符集下 `INSERT` 语句不能执行，`utf8mb4` 字符集下可以执行 `INSERT` 语句：

```
CREATE TABLE utf8_test (
     c char(1) NOT NULL
    ) CHARACTER SET utf8;
```

```
Query OK, 0 rows affected (0.09 sec)
```

```
CREATE TABLE utf8m4_test (
     c char(1) NOT NULL
    ) CHARACTER SET utf8mb4;
```

```
Query OK, 0 rows affected (0.09 sec)
```

```
INSERT INTO utf8_test VALUES ('');
```

```
ERROR 1366 (HY000): incorrect utf8 value f09f9889( ) for column c
```

```
INSERT INTO utf8m4_test VALUES ('');
```

```
Query OK, 1 row affected (0.02 sec)
```

```
SELECT char_length(c), length(c), c FROM utf8_test;
```

```
Empty set (0.01 sec)
```

```
SELECT char_length(c), length(c), c FROM utf8m4_test;
```

```
+----------------+-----------+------+
| char_length(c) | length(c) | c    |
+----------------+-----------+------+
|              1 |         4 |      |
+----------------+-----------+------+
1 row in set (0.00 sec)
```

14.11.16.1.4 不同范围的字符集和排序规则

字符集和排序规则可以在设置在不同的层次。

数据库的字符集和排序规则

每个数据库都有相应的字符集和排序规则。数据库的字符集和排序规则可以通过以下语句来设置：

```
CREATE DATABASE db_name
    [[DEFAULT] CHARACTER SET charset_name]
    [[DEFAULT] COLLATE collation_name]
```

```
ALTER DATABASE db_name
    [[DEFAULT] CHARACTER SET charset_name]
    [[DEFAULT] COLLATE collation_name]
```

在这里 DATABASE 可以跟 SCHEMA 互换使用。

不同的数据库之间可以使用不一样的字符集和排序规则。

通过系统变量 character_set_database 和 collation_database 可以查看到当前数据库的字符集以及排序规则：

```
CREATE SCHEMA test1 CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci;
```

```
Query OK, 0 rows affected (0.09 sec)
```

```
USE test1;
```

```
Database changed
```

```
SELECT @@character_set_database, @@collation_database;
```

```
+--------------------------|----------------------+
| @@character_set_database | @@collation_database |
+--------------------------|----------------------+
| utf8mb4                  | utf8mb4_general_ci   |
+--------------------------|----------------------+
1 row in set (0.00 sec)
```

```
CREATE SCHEMA test2 CHARACTER SET latin1 COLLATE latin1_bin;
```

```
Query OK, 0 rows affected (0.09 sec)
```

```
USE test2;
```

```
Database changed
```

```
SELECT @@character_set_database, @@collation_database;
```

```
+--------------------------|----------------------+
| @@character_set_database | @@collation_database |
+--------------------------|----------------------+
| latin1                   | latin1_bin           |
+--------------------------|----------------------+
1 row in set (0.00 sec)
```

在 INFORMATION_SCHEMA 中也可以查看到这两个值：

```
SELECT DEFAULT_CHARACTER_SET_NAME, DEFAULT_COLLATION_NAME
FROM INFORMATION_SCHEMA.SCHEMATA WHERE SCHEMA_NAME = 'db_name';
```

表的字符集和排序规则

表的字符集和排序规则可以通过以下语句来设置：

```
CREATE TABLE tbl_name (column_list)
    [[DEFAULT] CHARACTER SET charset_name]
    [COLLATE collation_name]]

ALTER TABLE tbl_name
    [[DEFAULT] CHARACTER SET charset_name]
    [COLLATE collation_name]
```

例如：

```
CREATE TABLE t1(a int) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci;
```

```
Query OK, 0 rows affected (0.08 sec)
```

如果表的字符集和排序规则没有设置，那么数据库的字符集和排序规则就作为其默认值。

列的字符集和排序规则

列的字符集和排序规则的语法如下：

```
col_name {CHAR | VARCHAR | TEXT} (col_length)
    [CHARACTER SET charset_name]
    [COLLATE collation_name]

col_name {ENUM | SET} (val_list)
    [CHARACTER SET charset_name]
    [COLLATE collation_name]
```

如果列的字符集和排序规则没有设置，那么表的字符集和排序规则就作为其默认值。

字符串的字符集和排序规则

每一个字符串都对应一个字符集和一个排序规则，在使用字符串时指此选项可选，如下：

```
[_charset_name]'string' [COLLATE collation_name]
```

示例如下：

```
SELECT 'string';
SELECT _utf8mb4'string';
SELECT _utf8mb4'string' COLLATE utf8mb4_general_ci;
```

规则如下：

- 规则 1：如果指定了 CHARACTER SET charset_name 和 COLLATE collation_name，则直接使用 charset_name 字符集和 collation_name 排序规则。
- 规则 2：如果指定了 CHARACTER SET charset_name 且未指定 COLLATE collation_name，则使用 charset_name 字符集和 charset_name 对应的默认排序规则。
- 规则 3：如果 CHARACTER SET charset_name 和 COLLATE collation_name 都未指定，则使用 character_set_connection ↪ 和 collation_connection 系统变量给出的字符集和排序规则。

客户端连接的字符集和排序规则

- 服务器的字符集和排序规则可以通过系统变量 character_set_server 和 collation_server 获取。
- 数据库的字符集和排序规则可以通过环境变量 character_set_database 和 collation_database 获取。

对于每一个客户端的连接，也有相应的变量表示字符集和排序规则：character_set_connection 和 collation_connection。

character_set_client 代表客户端的字符集。在返回结果前，服务端会把结果根据 character_set_results 转换成对应的字符集，包括结果的元信息等。

可以用以下的语句来影响这些跟客户端相关的字符集变量：

- SET NAMES 'charset_name' [COLLATE 'collation_name']

  SET NAMES 用来设定客户端会在之后的请求中使用的字符集。SET NAMES utf8mb4 表示客户端会在接下来的请求中，都使用 utf8mb4 字符集。服务端也会在之后返回结果的时候使用 utf8mb4 字符集。SET NAMES 'charset_name' 语句其实等于下面语句的组合：

```
SET character_set_client = charset_name;
SET character_set_results = charset_name;
SET character_set_connection = charset_name;
```

  COLLATE 是可选的，如果没有提供，将会用 charset_name 对应的默认排序规则设置 collation_connection ↪ 。

- SET CHARACTER SET 'charset_name'

  跟 SET NAMES 类似，等价于下面语句的组合：

```
SET character_set_client = charset_name;
SET character_set_results = charset_name;
SET charset_connection = @@charset_database;
SET collation_connection = @@collation_database;
```

14.11.16.1.5 服务器、数据库、表、列、字符串的字符集和排序规则的优先级

优先级从高到低排列顺序为：

字符串 > 列 > 表 > 数据库 > 服务器

14.11.16.1.6 字符集和排序规则的通用选择规则

- 规则 1：如果指定了 CHARACTER SET charset_name 和 COLLATE collation_name，则直接使用 charset_name 字符集和 collation_name 排序规则。
- 规则 2：如果指定了 CHARACTER SET charset_name 且未指定 COLLATE collation_name，则使用 charset_name 字符集和 charset_name 对应的默认排序规则。
- 规则 3：如果 CHARACTER SET charset_name 和 COLLATE collation_name 都未指定，则使用更高优先级的字符集和排序规则。

14.11.16.1.7 字符合法性检查

当指定的字符集为 utf8 或 utf8mb4 时，TiDB 仅支持合法的 utf8 字符。对于不合法的字符，会报错：incorrect utf8 value。该字符合法性检查与 MySQL 8.0 兼容，与 MySQL 5.7 及以下版本不兼容。

如果不希望报错，可以通过 set @@tidb_skip_utf8_check=1; 跳过字符检查。

> 注意：
>
> 跳过字符检查可能会使 TiDB 检测不到应用写入的非法 UTF-8 字符，进一步导致执行 ANALYZE 时解码错误，以及引入其他未知的编码问题。如果应用不能保证写入字符串的合法性，不建议跳过该检查。

14.11.16.1.8 排序规则支持

排序规则的语法支持和语义支持受到配置项new_collations_enabled_on_first_bootstrap 的影响。这里语法支持和语义支持有所区别。语法支持是指 TiDB 能够解析和设置排序规则；而语义支持是指 TiDB 能够在比较字符串时正确地使用排序规则。

在 4.0 版本之前，TiDB 只提供了旧的排序规则框架，能够在语法上支持的绝大部分 MySQL 排序规则，但语义上所有的排序规则都当成二进制排序规则。

4.0 版本中，TiDB 增加了新的排序规则框架用于在语义上支持不同的排序规则，保证字符串比较时严格遵循对应的排序规则，详情请见下文。

旧框架下的排序规则支持

在 4.0 版本之前，TiDB 中可以指定大部分 MySQL 中的排序规则，并把这些排序规则按照默认排序规则处理，即以编码字节序为字符定序。和 MySQL 不同的是，TiDB 不会处理字符末尾的空格，因此会造成以下的行为区别：

```
CREATE TABLE t(a varchar(20) charset utf8mb4 collate utf8mb4_general_ci PRIMARY KEY);
```

```
Query OK, 0 rows affected
```

```
INSERT INTO t VALUES ('A');
```

```
Query OK, 1 row affected
```

```
INSERT INTO t VALUES ('a');
```

```
Query OK, 1 row affected
```

以上语句，在 TiDB 会执行成功，而在 MySQL 中，由于 utf8mb4_general_ci 大小写不敏感，报错 Duplicate
↪ entry 'a'。

```
INSERT INTO t VALUES ('a ');
```

```
Query OK, 1 row affected
```

以上语句，在 TiDB 会执行成功，而在 MySQL 中，由于补齐空格比较，报错 Duplicate entry 'a '。

新框架下的排序规则支持

TiDB 4.0 新增了完整的排序规则支持框架，从语义上支持了排序规则，并新增了配置开关 new_collations_enabled_on_first_boot
↪ ，在集群初次初始化时决定是否启用新排序规则框架。如需启用新排序规则框架，可将
new_collations_enabled_on_first_bootstrap 的值设为 true，详情参见new_collations_enabled_on_first_bootstrap
↪ 。要在该配置开关打开之后初始化集群，可以通过 mysql.tidb 表中的 new_collation_enabled 变量确认是
否启用了新排序规则框架：

```
SELECT VARIABLE_VALUE FROM mysql.tidb WHERE VARIABLE_NAME='new_collation_enabled';
```

```
+----------------+
| VARIABLE_VALUE |
+----------------+
| True           |
+----------------+
1 row in set (0.00 sec)
```

在新的排序规则框架下，TiDB 能够支持 utf8_general_ci、utf8mb4_general_ci、utf8_unicode_ci、
utf8mb4_unicode_ci、gbk_chinese_ci 和 gbk_bin 这几种排序规则，与 MySQL 兼容。

使用 utf8_general_ci、utf8mb4_general_ci、utf8_unicode_ci 、utf8mb4_unicode_ci 和 gbk_chinese_ci 中任
一种时，字符串之间的比较是大小写不敏感 (case-insensitive) 和口音不敏感 (accent-insensitive) 的。同时，TiDB 还
修正了排序规则的 PADDING 行为：

```
CREATE TABLE t(a varchar(20) charset utf8mb4 collate utf8mb4_general_ci PRIMARY KEY);
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
INSERT INTO t VALUES ('A');
```

```
Query OK, 1 row affected (0.00 sec)
```

```
INSERT INTO t VALUES ('a');
```

```
ERROR 1062 (23000): Duplicate entry 'a' for key 't.PRIMARY'
```

TiDB 兼容了 MySQL 的 case insensitive collation。

```
INSERT INTO t VALUES ('a ');
```

```
ERROR 1062 (23000): Duplicate entry 'a ' for key 't.PRIMARY'
```

TiDB 修正了 PADDING 行为，与 MySQL 兼容。

> **注意：**
>
> TiDB 中 padding 的实现方式与 MySQL 的不同。在 MySQL 中，padding 是通过补齐空格实现的。而在 TiDB 中 padding 是通过裁剪掉末尾的空格来实现的。两种做法在绝大多数情况下是一致的，唯一的例外是字符串尾部包含小于空格 (0x20) 的字符时，例如 `'a' < 'a\t'` 在 TiDB 中的结果为 1，而在 MySQL 中，其等价于 `'a ' < 'a\t'`，结果为 0。

14.11.16.1.9 表达式中排序规则的 Coercibility 值

如果一个表达式涉及多个不同排序规则的子表达式时，需要对计算时用的排序规则进行推断，规则如下：

- 显式 COLLATE 子句的 coercibility 值为 0。
- 如果两个字符串的排序规则不兼容，这两个字符串 concat 结果的 coercibility 值为 1。
- 列或者 CAST()、CONVERT() 和 BINARY() 的排序规则的 coercibility 值为 2。
- 系统常量（USER() 或者 VERSION() 返回的字符串）的 coercibility 值为 3。
- 常量的 coercibility 值为 4。
- 数字或者中间变量的 coercibility 值为 5。
- NULL 或者由 NULL 派生出的表达式的 coercibility 值为 6。

在推断排序规则时，TiDB 优先使用 coercibility 值较低的表达式的排序规则。如果 coercibility 值相同，则按以下优先级确定排序规则：

binary > utf8mb4_bin > (utf8mb4_general_ci = utf8mb4_unicode_ci) > utf8_bin > (utf8_general_ci = utf8_unicode_ci) > latin1_bin > ascii_bin

以下情况 TiDB 无法推断排序规则并报错：

- 如果两个子表达式的排序规则不相同，而且表达式的 coercibility 值都为 0。
- 如果两个子表达式的排序规则不兼容，而且表达式的返回类型为 String 类。

14.11.16.1.10 COLLATE 子句

TiDB 支持使用 COLLATE 子句来指定一个表达式的排序规则，该表达式的 coercibility 值为 0，具有最高的优先级。示例如下：

```
SELECT 'a' = _utf8mb4 'A' collate utf8mb4_general_ci;
```

```
+-----------------------------------------------+
| 'a' = _utf8mb4 'A' collate utf8mb4_general_ci |
+-----------------------------------------------+
|                                             1 |
+-----------------------------------------------+
1 row in set (0.00 sec)
```

更多细节，参考 Connection Character Sets and Collations。

14.11.16.2 GBK

TiDB 从 v5.4.0 开始支持 GBK 字符集。本文档介绍 TiDB 对 GBK 字符集的支持和兼容情况。

```
SHOW CHARACTER SET WHERE CHARSET = 'gbk';
+---------+-----------------------------------+--------------------+--------+
| Charset | Description                       | Default collation | Maxlen |
+---------+-----------------------------------+--------------------+--------+
| gbk     | Chinese Internal Code Specification | gbk_bin           |      2 |
+---------+-----------------------------------+--------------------+--------+
1 row in set (0.00 sec)

SHOW COLLATION WHERE CHARSET = 'gbk';
+----------------+---------+------+---------+----------+---------+
| Collation      | Charset | Id   | Default | Compiled | Sortlen |
+----------------+---------+------+---------+----------+---------+
| gbk_bin        | gbk     |   87 |         | Yes      |       1 |
+----------------+---------+------+---------+----------+---------+
1 rows in set (0.00 sec)
```

14.11.16.2.1 与 MySQL 的兼容性

本节介绍 TiDB 中 GBK 字符集与 MySQL 的兼容情况。

排序规则兼容性

MySQL 的字符集默认排序规则是 gbk_chinese_ci。与 MySQL 不同，TiDB GBK 字符集的默认排序规则为 gbk_bin。另外，TiDB 支持的 gbk_bin 与 MySQL 支持的 gbk_bin 排序规则也不一致，TiDB 是将 GBK 转换成 UTF8MB4 然后做二进制排序。

如果要使 TiDB 兼容 MySQL 的 GBK 字符集排序规则，你需要在初次初始化 TiDB 集群时设置 TiDB 配置项new_collations_enabled_on_first_bootstrap 为 true 来开启新的排序规则框架。

开启新的排序规则框架后，如果查看 GBK 字符集对应的排序规则，你可以看到 TiDB GBK 默认排序规则已经切换为 gbk_chinese_ci。

```
SHOW CHARACTER SET WHERE CHARSET = 'gbk';
+---------+-----------------------------------+-------------------+--------+
| Charset | Description                       | Default collation | Maxlen |
+---------+-----------------------------------+-------------------+--------+
| gbk     | Chinese Internal Code Specification | gbk_chinese_ci    |      2 |
+---------+-----------------------------------+-------------------+--------+
1 row in set (0.00 sec)

SHOW COLLATION WHERE CHARSET = 'gbk';
+----------------+---------+------+---------+----------+---------+
| Collation      | Charset | Id   | Default | Compiled | Sortlen |
+----------------+---------+------+---------+----------+---------+
| gbk_bin        | gbk     |   87 |         | Yes      |       1 |
| gbk_chinese_ci | gbk     |   28 | Yes     | Yes      |       1 |
+----------------+---------+------+---------+----------+---------+
2 rows in set (0.00 sec)
```

非法字符兼容性

- 在系统变量character_set_client 和character_set_connection 不同时设置为 gbk 的情况下，TiDB 处理非法字符的方式与 MySQL 一致。
- 在 character_set_client 和 character_set_connection 同时为 gbk 的情况下，TiDB 处理非法字符的方式与 MySQL 有所区别。
  - MySQL 处理非法 GBK 字符集时，对读和写操作的处理方式不同。
  - TiDB 处理非法 GBK 字符集时，对读和写操作的处理方式相同。TiDB 在严格模式下读写非法 GBK 字符都会报错，在非严格模式下，读写非法 GBK 字符都会用 ? 替换。

例如，当 SET NAMES gbk 时，如果分别在 MySQL 和 TiDB 上通过 CREATE TABLE gbk_table(a VARCHAR(32) ↪ CHARACTER SET gbk) 语句建表，然后按照下表中的 SQL 语句进行操作，就能看到具体的区别。

数据库	如果设置的 SQL 模式包含 STRICT_ALL_TABLES 或 STRICT_TRANS_TABLES	如果设置的 SQL 模式不包含 STRICT_ALL_TABLES 和 STRICT_TRANS_TABLES
MySQL	SELECT HEX('一a'); e4b88061INSERT INTO ↪ gbk_table values('一a'); Incorrect Error	SELECT HEX('一a'); e4b88061INSERT INTO gbk_table VALUES('一a'); ↪ SELECT HEX(a)FROM gbk_table; e4b8
TiDB	SELECT HEX('一a'); Incorrect ErrorINSERT ↪ INTO gbk_table VALUES('一a'); Incorrect Error	SELECT HEX('一a'); e4b83fINSERT INTO gbk_table VALUES('一a'); ↪ SELECT HEX(a)FROM gbk_table; e4b83f

说明：该表中 SELECT HEX('一a'); 在 utf8mb4 字节集下的结果为 e4b88061。

其它

- 目前 TiDB 不支持通过 ALTER TABLE 语句将其它字符集类型改成 gbk 或者从 gbk 转成其它字符集类型。
- TiDB 不支持使用 _gbk，比如：

```
sql CREATE TABLE t(a CHAR(10)CHARSET BINARY); Query OK, 0 rows affected (0.00 sec)INSERT INTO t
↪ VALUES (_gbk'啊'); ERROR 1115 (42000): Unsupported character introducer: 'gbk'
```

- 对于 ENUM 和 SET 类型中的二进制字符，TiDB 目前都会将其作为 utf8mb4 字符集处理。

14.11.16.2.2 组件兼容性

- TiFlash 目前不支持 GBK 字符集。
- TiDB Data Migration (DM) 在 v5.4.0 之前不支持将 charset=GBK 的表迁移到 TiDB。
- TiDB Lightning 在 v5.4.0 之前不支持导入 charset=GBK 的表。
- TiCDC 在 v6.1.0 之前不支持同步 charset=GBK 的表。另外，任何版本的 TiCDC 都不支持同步 charset=GBK 的表到 6.1.0 之前的 TiDB 集群。
- TiDB Backup & Restore（BR）在 v5.4.0 之前不支持恢复 charset=GBK 的表。另外，任何版本的 BR 都不支持恢复 charset=GBK 的表到 5.4.0 之前的 TiDB 集群。

14.11.17 Placement Rules in SQL

Placement Rules in SQL 特性用于通过 SQL 接口配置数据在 TiKV 集群中的放置位置。通过该功能，用户可以将表和分区指定部署至不同的地域、机房、机柜、主机。适用场景包括低成本优化数据高可用策略、保证本地的数据副本可用于本地 Stale Read 读取、遵守数据本地要求等。

> **注意：**
>
> Placement Rules in SQL 底层的实现依赖 PD 提供的放置规则 (placement rules) 功能，参考Placement Rules 使用文档。在 Placement Rules in SQL 语境下，放置规则既可以代指绑定对象的放置策略 (placement policy)，也可以代指 TiDB 发给 PD 的放置规则。

该功能可以实现以下业务场景：

- 合并多个不同业务的数据库，大幅减少数据库常规运维管理的成本
- 增加重要数据的副本数，提高业务可用性和数据可靠性
- 将最新数据存入 NVMe，历史数据存入 SSD，降低归档数据存储成本
- 把热点数据的 leader 放到高性能的 TiKV 实例上
- 将冷数据分离到不同的存储中以提高可用性
- 支持物理隔离不同用户之间的计算资源，满足实例内部不同用户的隔离需求，以及不同混合负载 CPU、I/O、内存等资源隔离的需求

14.11.17.1 指定放置规则

指定放置规则，首先需要通过`CREATE PLACEMENT POLICY`语句创建放置策略 (placement policy)。

```
CREATE PLACEMENT POLICY myplacementpolicy PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-west-1
    ↪ ";
```

然后可以使用 `CREATE TABLE` 或者 `ALTER TABLE` 将规则绑定至表或分区表，这样就在表或分区上指定了放置规则：

```
CREATE TABLE t1 (a INT) PLACEMENT POLICY=myplacementpolicy;
CREATE TABLE t2 (a INT);
ALTER TABLE t2 PLACEMENT POLICY=myplacementpolicy;
```

`PLACEMENT POLICY` 为全局作用域，不与任何数据库表结构相关联。因此，通过 `CREATE TABLE` 指定放置规则时，无需任何额外的权限。

要修改放置策略，你可以使用`ALTER PLACEMENT POLICY`语句。修改将传播到所有绑定此放置策略的对象。

```
ALTER PLACEMENT POLICY myplacementpolicy FOLLOWERS=5;
```

要删除没有绑定任何分区或表的放置策略，你可以使用`DROP PLACEMENT POLICY`：

```
DROP PLACEMENT POLICY myplacementpolicy;
```

14.11.17.2 查看放置规则

如果一张表绑定了放置规则，你可以用`SHOW CREATE TABLE`来查看。还可以用`SHOW CREATE PLACEMENT POLICY`来查看已经创建的放置策略。

```
tidb> SHOW CREATE TABLE t1\G
*************************** 1. row ***************************
       Table: t1
Create Table: CREATE TABLE `t1` (
  `a` int(11) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin /*T![placement] PLACEMENT POLICY=`
    ↪ myplacementpolicy` */
1 row in set (0.00 sec)
tidb> SHOW CREATE PLACEMENT POLICY myplacementpolicy\G
*************************** 1. row ***************************
       Policy: myplacementpolicy
Create Policy: CREATE PLACEMENT POLICY myplacementpolicy PRIMARY_REGION="us-east-1" REGIONS="us-
    ↪ east-1,us-west-1"
1 row in set (0.00 sec)
```

你也可以用`INFORMATION_SCHEMA.PLACEMENT_POLICIES`系统表查看所有放置策略的定义。

```
tidb> select * from information_schema.placement_policies\G
***************************[ 1. row ]***************************
```

```
POLICY_ID             | 1
CATALOG_NAME          | def
POLICY_NAME           | p1
PRIMARY_REGION        | us-east-1
REGIONS               | us-east-1,us-west-1
CONSTRAINTS           |
LEADER_CONSTRAINTS    |
FOLLOWER_CONSTRAINTS  |
LEARNER_CONSTRAINTS   |
SCHEDULE              |
FOLLOWERS             | 4
LEARNERS              | 0
1 row in set
```

`information_schema.tables` 表和 `information_schema.partitions` 表也有一列 `tidb_placement_policy_name`，用于展示所有绑定了放置规则的对象：

```
SELECT * FROM information_schema.tables WHERE tidb_placement_policy_name IS NOT NULL;
SELECT * FROM information_schema.partitions WHERE tidb_placement_policy_name IS NOT NULL;
```

所有绑定规则的对象都是异步调度的。可以用`SHOW PLACEMENT` 来查看放置规则的调度进度。

14.11.17.3 放置选项参考

注意：

- 放置选项依赖于正确地指定在每个 TiKV 节点配置中的标签 (label)。例如，`PRIMARY_REGION` 选项依赖 TiKV 中的 `region` 标签。若要查看当前 TiKV 集群中所有可用的标签，可执行`SHOW PLACEMENT LABELS` 语句。

```
mysql> show placement labels;
+--------+----------------+
| Key    | Values         |
+--------+----------------+
| disk   | ["ssd"]        |
| region | ["us-east-1"]  |
| zone   | ["us-east-1a"] |
+--------+----------------+
3 rows in set (0.00 sec)
```

- 使用 `CREATE PLACEMENT POLICY` 创建放置规则时，TiDB 不会检查标签是否存在，而是在绑定表的时候进行检查。

选项名	描述
PRIMARY_REGION	Raft leader 被放置在有 region 标签的节点上，且这些 region 标签匹配本选项的值。
REGIONS	Raft followers 被放置在有 region 标签的节点上，且这些 region 标签匹配本选项的值。
SCHEDULE	用于调度 follower 放置位置的策略。可选值为 EVEN（默认值）或 MAJORITY_IN_PRIMARY。
FOLLOWERS	Follower 的数量。例如 FOLLOWERS=2 表示数据有 3 个副本（2 个 follower 和 1 个 leader）。

除以上配置选项外，你还可以使用高级配置，详细介绍见高级放置选项。

选项名	描述
CONSTRAINTS	适用于所有角色 (role) 的约束列表。例如，CONSTRAINTS="[+disk=ssd]"。
LEADER_CONSTRAINTS	仅适用于 leader 的约束列表。
FOLLOWER_CONSTRAINTS	仅适用于 follower 的约束列表。
LEARNER_CONSTRAINTS	仅适用于 learner 的约束列表。
LEARNERS	指定 learner 的数量。
SURVIVAL_PREFERENCE	指定按 label 容灾等级的优先级放置副本。例如 SURVIVAL_PREFERENCE="[region, zone, host]"。

14.11.17.4 示例

14.11.17.4.1 增加副本数

max-replicas 配置项的默认值为 3。如要为特定的表调大该值，可使用配置策略，示例如下：

```
CREATE PLACEMENT POLICY fivereplicas FOLLOWERS=4;
CREATE TABLE t1 (a INT) PLACEMENT POLICY=fivereplicas;
```

注意，PD 配置中包含了 leader 数和 follower 数。因此，5 个副本为 4 个 follower + 1 个 leader。

对于以上示例，你还可使用 PRIMARY_REGION 和 REGIONS 选项来描述 follower 的放置规则：

```
CREATE PLACEMENT POLICY eastandwest PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-east-2,us-
    ↪ west-1" SCHEDULE="MAJORITY_IN_PRIMARY" FOLLOWERS=4;
CREATE TABLE t1 (a INT) PLACEMENT POLICY=eastandwest;
```

SCHEDULE 选项指示 TiDB 如何平衡 follower。该选项默认的 EVEN 调度规则确保 follower 在所有区域内分布平衡。

如要保证在主区域内 (us-east-1) 放置足够多的 follower 副本，你可以使用 MAJORITY_IN_PRIMARY 调度规则来使该区域的 follower 达到指定数量。该调度牺牲一些可用性来换取更低的事务延迟。如果主区域宕机，MAJORITY_IN_PRIMARY 无法提供自动故障转移。

14.11.17.4.2 为分区表指定放置规则

除了给表绑定放置策略之外，你还可以给表分区绑定放置策略。示例如下：

```
CREATE PLACEMENT POLICY p1 FOLLOWERS=5;
CREATE PLACEMENT POLICY europe PRIMARY_REGION="eu-central-1" REGIONS="eu-central-1,eu-west-1";
```

```
CREATE PLACEMENT POLICY northamerica PRIMARY_REGION="us-east-1" REGIONS="us-east-1";

SET tidb_enable_list_partition = 1;
CREATE TABLE t1 (
  country VARCHAR(10) NOT NULL,
  userdata VARCHAR(100) NOT NULL
) PLACEMENT POLICY=p1 PARTITION BY LIST COLUMNS (country) (
  PARTITION pEurope VALUES IN ('DE', 'FR', 'GB') PLACEMENT POLICY=europe,
  PARTITION pNorthAmerica VALUES IN ('US', 'CA', 'MX') PLACEMENT POLICY=northamerica,
  PARTITION pAsia VALUES IN ('CN', 'KR', 'JP')
);
```

如果分区没有绑定任何放置策略，分区将尝试继承表上可能存在的策略。比如，pEurope 分区将会应用 europe 策略，而 pAsia 分区将会应用表 t1 的放置策略 p1。如果 t1 没有绑定任何策略，pAsia 就不会应用任何策略。

给分区绑定放置策略后，你可以更改指定分区的放置策略。示例如下：

```
ALTER TABLE t1 PARTITION pEurope PLACEMENT POLICY=p1;
```

14.11.17.4.3 为数据库配置默认的放置规则

你可以为某个数据库指定默认的放置策略，类似于为数据库设置默认字符集或排序规则。如果没有指定其他选项，就会使用数据库上指定的配置。示例如下：

```
CREATE PLACEMENT POLICY p1 PRIMARY_REGION="us-east-1" REGIONS="us-east-1,us-east-2";  --
    ↪ 创建放置策略

CREATE PLACEMENT POLICY p2 FOLLOWERS=4;

CREATE PLACEMENT POLICY p3 FOLLOWERS=2;

CREATE TABLE t1 (a INT);  -- 创建表 t1，且未指定放置规则。

ALTER DATABASE test PLACEMENT POLICY=p2;  -- 更改默认的放置规则，但更改不影响已有的表 t1。

CREATE TABLE t2 (a INT);  -- 创建表 t2，默认的放置策略 p2 在 t2 上生效。

CREATE TABLE t3 (a INT) PLACEMENT POLICY=p1;  -- 创建表 t3。因为语句中已经指定了其他放置规则，
    ↪ 默认的 p2 策略在 t3 上不生效。

ALTER DATABASE test PLACEMENT POLICY=p3;  -- 再次更改默认的放置规则，此更改不影响已有的表。

CREATE TABLE t4 (a INT);  -- 创建表 t4，默认的放置策略 p3 生效。

ALTER PLACEMENT POLICY p3 FOLLOWERS=3; -- 绑定策略 p3 的表，也就是 t4，会采用 FOLLOWERS=3。
```

注意分区与表之间的继承和这里的继承不同。改变表的放置策略，也会让分区应用新的策略。但是只有建表时没有指定放置策略的时候，表才会从数据库继承放置策略，且之后再改变数据库也不影响已经继承的表。

14.11.17.4.4 高级放置选项

放置选项 PRIMARY_REGION、REGIONS 和 SCHEDULE 可满足数据放置的基本需求，但会缺乏一些灵活性。在较复杂的场景下，若需要更灵活地放置数据，可以使用高级放置选项 CONSTRAINTS 和 FOLLOWER_CONSTRAINTS。PRIMARY_REGION、REGIONS 和 SCHEDULE 选项不可与 CONSTRAINTS 选项同时指定，否则会报错。

以下示例设置了一个约束，要求数据必须位于某个 TiKV 节点，且该节点的 disk 标签必须匹配特定的值：

```
CREATE PLACEMENT POLICY storageonnvme CONSTRAINTS="[+disk=nvme]";
CREATE PLACEMENT POLICY storageonssd CONSTRAINTS="[+disk=ssd]";
CREATE PLACEMENT POLICY companystandardpolicy CONSTRAINTS="";

CREATE TABLE t1 (id INT, name VARCHAR(50), purchased DATE)
PLACEMENT POLICY=companystandardpolicy
PARTITION BY RANGE( YEAR(purchased) ) (
  PARTITION p0 VALUES LESS THAN (2000) PLACEMENT POLICY=storageonssd,
  PARTITION p1 VALUES LESS THAN (2005),
  PARTITION p2 VALUES LESS THAN (2010),
  PARTITION p3 VALUES LESS THAN (2015),
  PARTITION p4 VALUES LESS THAN MAXVALUE PLACEMENT POLICY=storageonnvme
);
```

该约束可通过列表格式（[+disk=ssd]）或字典格式（{+disk=ssd: 1,+disk=nvme: 2}）指定。

在列表格式中，约束以键值对列表格式。键以 + 或 - 开头。+disk=nvme 表示 disk 标签必须设为 nvme，-disk=nvme 表示 disk 标签值不能为 nvme。

在字典格式中，约束还指定了适用于该规则的多个实例。例如，FOLLOWER_CONSTRAINTS="{+region=us-east-1: 1,+region=us-east-2: 1,+region=us-west-1: 1}"；表示 1 个 follower 位于 us-east-1，1 个 follower 位于 us-east-2，1 个 follower 位于 us-west-1。再例如，FOLLOWER_CONSTRAINTS='{"+region=us-east-1,+disk=nvme": 1,"+region=us-west-1": 1}'；表示 1 个 follower 位于 us-east-1 区域中有 nvme 硬盘的机器上，1 个 follower 位于 us-west-1。

> **注意：**
>
> 字典和列表格式都基于 YAML 解析，但 YAML 语法有些时候不能被正常解析。例如 YAML 会把 "{+disk=ssd:1,+disk=nvme:2}"（: 后无空格）错误地解析成 '{"+disk=ssd:1": null, "+disk=nvme:2": null}'，不符合预期。但 "{+disk=ssd: 1,+disk=nvme: 2}"（: 后有空格）能被正确解析成 '{"+disk=ssd": 1, "+disk=nvme": 2}'。

14.11.17.4.5 生存偏好

在创建或修改放置策略时，你可以使用 SURVIVAL_PREFERENCES 选项设置数据的生存能力偏好。

例如，假设你的 TiDB 集群分布在 3 个 zone（即可用区），且每个可用区的 host（即节点）上混合部署了多个 TiKV 实例。在为该集群创建放置策略时，假设 SURVIVAL_PREFERENCES 的设置如下：

```
CREATE PLACEMENT POLICY multiaz SURVIVAL_PREFERENCES="[zone, host]";
CREATE PLACEMENT POLICY singleaz CONSTRAINTS="[+zone=zone1]" SURVIVAL_PREFERENCES="[host]";
```

创建好放置策略后，你可以按需将放置策略绑定到对应的表上：

- 对于绑定了 multiaz 放置策略的表，数据将以 3 副本的形式放置在不同的可用区里，优先满足跨 zone 级别数据隔离的生存目标，再满足跨 host 级别的数据隔离的生存目标。
- 对于绑定了 singleaz 放置策略的表，数据会优先以 3 副本的形式全部放置在 zone1 这个可用区里，再满足跨 host 级别的数据隔离的生存目标。

> 注意：
>
> SURVIVAL_PREFERENCES 和 PD 中的 location-labels 是等价的，更多信息可以参考通过拓扑 label 进行副本调度。

14.11.17.5 工具兼容性

工具名称	最低兼容版本	说明
Backup & Restore (BR)	6.0	BR 在 v6.0 之前不支持放置规则的备份与恢复，见恢复 Placement Rule 到集群时为什么会报
TiDB Lightning	暂时不兼容	导入包含放置策略的数据时会报错
TiCDC	6.0	忽略放置规则，不同步规则到下游集群
TiDB Binlog	6.0	忽略放置规则，不同步规则到下游集群

14.11.17.6 使用限制

目前已知 Placement Rules in SQL 特性存在以下限制：

- 临时表不支持放置规则。
- 设置 PRIMARY_REGION 和 REGIONS 时允许存在语法糖。但在未来版本中，我们计划为 PRIMARY_RACK、PRIMARY_ZONE 和 PRIMARY_HOST 添加变体支持，见 issue #18030。
- 放置规则仅保证静态数据被放置在正确的 TiKV 节点上。该规则不保证传输中的数据（通过用户查询或内部操作）只出现在特定区域内。

14.11.18 系统表

14.11.18.1 mysql Schema

mysql 库里存储的是 TiDB 系统表。该设计类似于 MySQL 中的 mysql 库，其中 mysql.user 之类的表可以直接编辑。该库还包含许多 MySQL 的扩展表。

14.11.18.1.1 权限系统表

这些系统表里面包含了用户账户以及相应的授权信息：

- `user` 用户账户，全局权限，以及其它一些非权限的列
- `db` 数据库级别的权限
- `tables_priv` 表级的权限
- `columns_priv` 列级的权限
- `password_history` 记录密码更改历史
- `default_roles` 默认启用的角色
- `global_grants` 动态权限
- `global_priv` 基于证书的认证信息
- `role_edges` 角色之间的关系信息

14.11.18.1.2 服务端帮助信息系统表

- `help_topic` 目前为空

14.11.18.1.3 统计信息相关系统表

- `stats_buckets` 统计信息的桶
- `stats_histograms` 统计信息的直方图
- `stats_top_n` 统计信息的 TopN
- `stats_meta` 表的元信息，比如总行数和修改数
- `stats_extended` 扩展统计信息，比如列之间的顺序相关性
- `stats_feedback` 统计信息的查询反馈
- `stats_fm_sketch` 统计信息列的直方图 FMSketch 分布
- `analyze_options` 各个表默认的 `analyze` 参数
- `column_stats_usage` 列统计信息的使用情况
- `schema_index_usage` 索引的使用情况
- `analyze_jobs` 正在执行的统计信息收集任务以及过去 7 天内的历史任务记录

14.11.18.1.4 执行计划相关系统表

- `bind_info` 执行计划的绑定信息
- `capture_plan_baselines_blacklist` 关于自动绑定执行计划对象的黑名单

14.11.18.1.5 GC Worker 相关系统表

- `gc_delete_range` 需要被 GC worker 定期删除的 KV 范围段
- `gc_delete_range_done` 已经被删除的 KV 范围段

14.11.18.1.6 缓存表使用的系统表

- `table_cache_meta` 存储了缓存表的元信息

14.11.18.1.7 TTL 相关系统表

- `mysql.tidb_ttl_table_status` 所有 TTL 表的上一次执行与正在执行的 TTL 任务
- `mysql.tidb_ttl_task` 正在执行的 TTL 子任务
- `mysql.tidb_ttl_job_history` 过去 90 天内 TTL 任务的执行历史

14.11.18.1.8 其它系统表

- `GLOBAL_VARIABLES` 全局系统变量表
- `tidb` 用于 TiDB 在 bootstrap 的时候记录相关版本信息
- `expr_pushdown_blacklist` 表达式下推的黑名单
- `opt_rule_blacklist` 逻辑优化规则的黑名单
- `table_cache_meta` 缓存表的信息

14.11.18.2 INFORMATION_SCHEMA

14.11.18.2.1 Information Schema

Information Schema 提供了一种查看系统元数据的 ANSI 标准方法。除了包含与 MySQL 兼容的表外，TiDB 还提供了许多自定义的 `INFORMATION_SCHEMA` 表。

许多 `INFORMATION_SCHEMA` 表都有相应的 `SHOW` 命令。查询 `INFORMATION_SCHEMA` 的好处是可以在表之间进行 `join` 操作。

与 MySQL 兼容的表

表名	描述
CHARACTER_SETS	提供 TiDB 支持的字符集列表。
COLLATIONS	提供 TiDB 支持的排序规则列表。
COLLATION_CHARACTER_SET_APPLICABILITY	说明哪些排序规则适用于哪些字符集。
COLUMNS	提供所有表中列的列表。
COLUMN_PRIVILEGES	TiDB 未实现，返回零行。
COLUMN_STATISTICS	TiDB 未实现，返回零行。
ENGINES	提供支持的存储引擎列表。
EVENTS	TiDB 未实现，返回零行。
FILES	TiDB 未实现，返回零行。
GLOBAL_STATUS	TiDB 未实现，返回零行。
GLOBAL_VARIABLES	TiDB 未实现，返回零行。
KEY_COLUMN_USAGE	描述列的键约束，例如主键约束。
OPTIMIZER_TRACE	TiDB 未实现，返回零行。
PARAMETERS	TiDB 未实现，返回零行。
PARTITIONS	提供表分区的列表。
PLUGINS	TiDB 未实现，返回零行。
PROCESSLIST	提供与 `SHOW PROCESSLIST` 命令类似的信息。
PROFILING	TiDB 未实现，返回零行。
REFERENTIAL_CONSTRAINTS	提供有关 `FOREIGN KEY` 约束的信息。

表名	描述
ROUTINES	TiDB 未实现，返回零行。
SCHEMATA	提供与 SHOW DATABASES 命令类似的信息。
SCHEMA_PRIVILEGES	TiDB 未实现，返回零行。
SESSION_STATUS	TiDB 未实现，返回零行。
SESSION_VARIABLES	提供与 SHOW SESSION VARIABLES 命令类似的功能。
STATISTICS	提供有关表索引的信息。
TABLES	提供当前用户可见的表的列表。类似于 SHOW TABLES。
TABLESPACES	TiDB 未实现，返回零行。
TABLE_CONSTRAINTS	提供有关主键、唯一索引和外键的信息。
TABLE_PRIVILEGES	TiDB 未实现，返回零行。
TRIGGERS	TiDB 未实现，返回零行。
USER_ATTRIBUTES	汇总用户的注释和属性信息。
USER_PRIVILEGES	汇总与当前用户相关的权限。
VARIABLES_INFO	提供 TiDB 系统变量的信息。
VIEWS	提供当前用户可见的视图列表。类似于 SHOW FULL TABLES WHERE table_type = '

TiDB 中的扩展表

表名	描述
ANALYZE_STATUS	提供有关收集统计信息的任务的信息。
CLIENT_ERRORS_SUMMARY_BY_HOST	汇总由客户端请求生成并返回给客户端的错误和警告。
CLIENT_ERRORS_SUMMARY_BY_USER	汇总由客户端产生的错误和警告。
CLIENT_ERRORS_SUMMARY_GLOBAL	汇总由客户端产生的错误和警告。
CLUSTER_CONFIG	提供有关整个 TiDB 集群的配置设置的详细信息。
CLUSTER_DEADLOCKS	提供 DEADLOCKS 表的集群级别的视图。
CLUSTER_HARDWARE	提供在每个 TiDB 组件上发现的底层物理硬件的详细信息。
CLUSTER_INFO	提供当前集群拓扑的详细信息。
CLUSTER_LOAD	提供集群中 TiDB 服务器的当前负载信息。
CLUSTER_LOG	提供整个 TiDB 集群的日志。
CLUSTER_MEMORY_USAGE	提供 MEMORY_USAGE 表的集群级别的视图。
CLUSTER_MEMORY_USAGE_OPS_HISTORY	提供 MEMORY_USAGE_OPS_HISTORY 表的集群级别的视图。
CLUSTER_PROCESSLIST	提供 PROCESSLIST 表的集群级别的视图。
CLUSTER_SLOW_QUERY	提供 SLOW_QUERY 表的集群级别的视图。
CLUSTER_STATEMENTS_SUMMARY	提供 STATEMENTS_SUMMARY 表的集群级别的视图。
CLUSTER_STATEMENTS_SUMMARY_HISTORY	提供 STATEMENTS_SUMMARY_HISTORY 表的集群级别的视图。
CLUSTER_TIDB_TRX	提供 TIDB_TRX 表的集群级别的视图。
CLUSTER_SYSTEMINFO	提供集群中服务器的内核参数配置的详细信息。
DATA_LOCK_WAITS	提供 TiKV 服务器上的等锁信息。
DDL_JOBS	提供与 ADMIN SHOW DDL JOBS 类似的输出。
DEADLOCKS	提供 TiDB 节点上最近发生的数次死锁错误的信息。
INSPECTION_RESULT	触发内部诊断检查。
INSPECTION_RULES	进行的内部诊断检查的列表。
INSPECTION_SUMMARY	重要监视指标的摘要报告。

表名	描述
MEMORY_USAGE	提供当前 TiDB 实例的内存使用情况。
MEMORY_USAGE_OPS_HISTORY	提供当前 TiDB 实例内存相关的历史操作和执行依据。
METRICS_SUMMARY	从 Prometheus 获取的指标的摘要。
METRICS_SUMMARY_BY_LABEL	参见 METRICS_SUMMARY 表。
METRICS_TABLES	为 METRICS_SCHEMA 中的表提供 PromQL 定义。
PLACEMENT_POLICIES	提供所有放置策略的定义信息。
SEQUENCES	描述了基于 MariaDB 实现的 TiDB 序列。
SLOW_QUERY	提供当前 TiDB 服务器上慢查询的信息。
STATEMENTS_SUMMARY	类似于 MySQL 中的 performance_schema 语句摘要。
STATEMENTS_SUMMARY_HISTORY	类似于 MySQL 中的 performance_schema 语句摘要历史。
TABLE_STORAGE_STATS	提供存储的表的大小的详细信息。
TIDB_HOT_REGIONS	提供有关哪些 Region 访问次数最多的统计信息。
TIDB_HOT_REGIONS_HISTORY	提供有关哪些 Region 访问次数最多的历史统计信息。
TIDB_INDEXES	提供有关 TiDB 表的索引信息。
TIDB_SERVERS_INFO	提供 TiDB 服务器的列表
TIDB_TRX	提供 TiDB 节点上正在执行的事务的信息。
TIFLASH_REPLICA	提供有关 TiFlash 副本的详细信息。
TIKV_REGION_PEERS	提供 Region 存储位置的详细信息。
TIKV_REGION_STATUS	提供 Region 的统计信息。
TIKV_STORE_STATUS	提供 TiKV 服务器的基本信息。

14.11.18.2.2 ANALYZE_STATUS

ANALYZE_STATUS 表提供正在执行的收集统计信息的任务以及有限条历史任务记录。

从 TiDB v6.1.0 起，ANALYZE_STATUS 表显示集群级别的任务，且 TiDB 重启后仍能看到重启之前的任务记录。在 TiDB v6.1.0 之前，ANALYZE_STATUS 仅显示实例级别的任务，且 TiDB 重启后任务记录会被清空。

从 TiDB v6.1.0 起，可以通过系统表 `mysql.analyze_jobs` 查看过去 7 天内的历史任务记录。

```
USE information_schema;
DESC analyze_status;
```

```
+----------------+---------------------+------+------+---------+-------+
| Field          | Type                | Null | Key  | Default | Extra |
+----------------+---------------------+------+------+---------+-------+
| TABLE_SCHEMA   | varchar(64)         | YES  |      | NULL    |       |
| TABLE_NAME     | varchar(64)         | YES  |      | NULL    |       |
| PARTITION_NAME | varchar(64)         | YES  |      | NULL    |       |
| JOB_INFO       | longtext            | YES  |      | NULL    |       |
| PROCESSED_ROWS | bigint(64) unsigned | YES  |      | NULL    |       |
| START_TIME     | datetime            | YES  |      | NULL    |       |
| END_TIME       | datetime            | YES  |      | NULL    |       |
| STATE          | varchar(64)         | YES  |      | NULL    |       |
| FAIL_REASON    | longtext            | YES  |      | NULL    |       |
```

```
| INSTANCE        | varchar(512)         | YES  |      | NULL    |        |
| PROCESS_ID      | bigint(64) unsigned | YES  |      | NULL    |        |
+---------------+---------------------+------+------+---------+-------+
11 rows in set (0.00 sec)
```

```
SELECT * FROM information_schema.analyze_status;
```

```
+--
    ↪ ------------+------------+---------------+-------------------------------------------------------------------
    ↪
| TABLE_SCHEMA | TABLE_NAME | PARTITION_NAME | JOB_INFO
    ↪                                                                | PROCESSED_ROWS | START_TIME
    ↪          | END_TIME            | STATE    | FAIL_REASON | INSTANCE        | PROCESS_ID |
+--
    ↪ ------------+------------+---------------+-------------------------------------------------------------------
    ↪
| test         | t          | p1             | analyze table all columns with 256 buckets, 500
    ↪ topn, 1 samplerate |              0 | 2022-05-27 11:30:12 | 2022-05-27 11:30:12 |
    ↪ finished | NULL        | 127.0.0.1:4000 |       NULL |
| test         | t          | p0             | analyze table all columns with 256 buckets, 500
    ↪ topn, 1 samplerate |              0 | 2022-05-27 11:30:12 | 2022-05-27 11:30:12 |
    ↪ finished | NULL        | 127.0.0.1:4000 |       NULL |
| test         | t          | p1             | analyze index idx
    ↪                                                  |              0 | 2022-05-27 11:29:46 |
    ↪  2022-05-27 11:29:46 | finished | NULL        | 127.0.0.1:4000 |       NULL |
| test         | t          | p0             | analyze index idx
    ↪                                                  |              0 | 2022-05-27 11:29:46 |
    ↪  2022-05-27 11:29:46 | finished | NULL        | 127.0.0.1:4000 |       NULL |
| test         | t          | p1             | analyze columns
    ↪                                                    |              0 | 2022-05-27 11:29:46
    ↪  | 2022-05-27 11:29:46 | finished | NULL        | 127.0.0.1:4000 |       NULL |
| test         | t          | p0             | analyze columns
    ↪                                                    |              0 | 2022-05-27 11:29:46
    ↪  | 2022-05-27 11:29:46 | finished | NULL        | 127.0.0.1:4000 |       NULL |
+--
    ↪ ------------+------------+---------------+-------------------------------------------------------------------
    ↪
6 rows in set (0.00 sec)
```

ANALYZE_STATUS 表中列的含义如下：

- TABLE_SCHEMA：表所属的数据库的名称。
- TABLE_NAME：表的名称。
- PARTITION_NAME：分区表的名称。

- JOB_INFO：ANALYZE 任务的信息。如果分析索引，该信息会包含索引名。当 tidb_analyze_version =2 时，该信息会包含采样率等配置项。
- PROCESSED_ROWS：已经处理的行数。
- START_TIME：ANALYZE 任务的开始时间。
- END_TIME：ANALYZE 任务的结束时间。
- STATE：ANALYZE 任务的执行状态。其值可以是 pending、 running、 finished 或者 failed。
- FAIL_REASON：任务失败的原因。如果执行成功则为 NULL。
- INSTANCE：执行任务的 TiDB 实例。
- PROCESS_ID：执行任务的 process ID。

14.11.18.2.3 CLIENT_ERRORS_SUMMARY_BY_HOST

CLIENT_ERRORS_SUMMARY_BY_HOST 表汇总了已返回给连接到 TiDB 服务器的客户端的 SQL 错误和警告。这些错误和警告包括：

- 格式错误的 SQL 语句。
- 除以零错误。
- 尝试插入超出范围或重复的键值。
- 权限错误。
- 表不存在。

以上错误通过 MySQL 服务器协议返回给客户端，此时应用程序应在客户端采取适当操作。INFORMATION_SCHEMA ↪ .CLIENT_ERRORS_SUMMARY_BY_HOST 表提供了一种有效方法，能够在应用程序没有正确处理（或记录）TiDB 服务器返回的错误的情况下检查错误。

由于 CLIENT_ERRORS_SUMMARY_BY_HOST 会基于每个远程主机汇总错误，在诊断其中一台应用程序服务器比其他服务器生成更多错误的情况时很有用。可能的情况包括：

- 过时的 MySQL 客户端库。
- 过时的应用程序（可能是在推出新部署时遗漏了此服务器）。
- 用户权限中“主机”部分的使用不正确。
- 网络连接不可靠，导致更多超时或连接断开。

可以使用 FLUSH CLIENT_ERRORS_SUMMARY 语句重置汇总的计数。所汇总的是每个 TiDB 服务器的本地数据，并且只保留在内存中。如果 TiDB 服务器重新启动，会丢失汇总信息。

```
USE INFORMATION_SCHEMA;
DESC CLIENT_ERRORS_SUMMARY_BY_HOST;
```

输出结果如下：

```
+---------------+---------------+------+------+---------+-------+
| Field         | Type          | Null | Key  | Default | Extra |
+---------------+---------------+------+------+---------+-------+
| HOST          | varchar(255)  | NO   |      | NULL    |       |
| ERROR_NUMBER  | bigint(64)    | NO   |      | NULL    |       |
| ERROR_MESSAGE | varchar(1024) | NO   |      | NULL    |       |
```

```
| ERROR_COUNT   | bigint(64)    | NO   |      | NULL    |       |
| WARNING_COUNT | bigint(64)    | NO   |      | NULL    |       |
| FIRST_SEEN    | timestamp     | YES  |      | NULL    |       |
| LAST_SEEN     | timestamp     | YES  |      | NULL    |       |
+---------------+---------------+------+------+---------+-------+
7 rows in set (0.00 sec)
```

字段说明如下：

- HOST：客户端的远程主机。
- ERROR_NUMBER：返回的与 MySQL 兼容的错误码。
- ERROR_MESSAGE：与错误码匹配的错误消息（预处理语句形式）。
- ERROR_COUNT：此错误返回到客户端主机的次数。
- WARNING_COUNT：此警告返回到客户端主机的次数。
- FIRST_SEEN：首次从客户端主机看到此错误（或警告）的时间。
- LAST_SEEN：最近一次从客户端主机看到此错误（或警告）的时间。

以下示例显示了客户端连接到本地 TiDB 服务器时生成的警告。执行 FLUSH CLIENT_ERRORS_SUMMARY 语句后，会重置汇总。

```
SELECT 0/0;
SELECT * FROM CLIENT_ERRORS_SUMMARY_BY_HOST;
FLUSH CLIENT_ERRORS_SUMMARY;
SELECT * FROM CLIENT_ERRORS_SUMMARY_BY_HOST;
```

输出结果如下：

```
+-----+
| 0/0 |
+-----+
| NULL |
+-----+
1 row in set, 1 warning (0.00 sec)

+--
    ↪ ---------+--------------+---------------+-------------+---------------+---------------------+-----------
    ↪
| HOST      | ERROR_NUMBER | ERROR_MESSAGE | ERROR_COUNT | WARNING_COUNT | FIRST_SEEN          |
    ↪ LAST_SEEN           |
+--
    ↪ ---------+--------------+---------------+-------------+---------------+---------------------+-----------
    ↪
| 127.0.0.1 |         1365 | Division by 0 |           0 |             1 | 2021-03-18 12:51:54 |
    ↪ 2021-03-18 12:51:54 |
+--
    ↪ ---------+--------------+---------------+-------------+---------------+---------------------+-----------
    ↪
```

```
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

Empty set (0.00 sec)
```

14.11.18.2.4 CLIENT_ERRORS_SUMMARY_BY_USER

CLIENT_ERRORS_SUMMARY_BY_USER 表汇总了已返回给连接到 TiDB 服务器的客户端的 SQL 错误和警告。这些错误和警告包括：

- 格式错误的 SQL 语句。
- 除以零错误。
- 尝试插入超出范围或重复的键值。
- 权限错误。
- 表不存在。

以上错误通过 MySQL 服务器协议返回给客户端，此时应用程序应在客户端采取适当操作。INFORMATION_SCHEMA.CLIENT_ERRORS_SUMMARY_BY_USER 表提供了一种有效方法，能够在应用程序没有正确处理（或记录）TiDB 服务器返回的错误的情况下检查错误。

因为 CLIENT_ERRORS_SUMMARY_BY_USER 会基于每个用户汇总错误，所以在诊断一个用户服务器比其他服务器产生更多错误的方案时很有用。可能的情况包括：

- 权限错误。
- 表或关系对象丢失。
- SQL 语法不正确，或应用程序和 TiDB 版本之间不兼容。

可以使用 FLUSH CLIENT_ERRORS_SUMMARY 语句重置汇总的计数。所汇总的是每个 TiDB 服务器的本地数据，并且只保留在内存中。如果 TiDB 服务器重新启动，会丢失汇总信息。

```
USE INFORMATION_SCHEMA;
DESC CLIENT_ERRORS_SUMMARY_BY_USER;
```

```
+---------------+---------------+------+------+---------+-------+
| Field         | Type          | Null | Key  | Default | Extra |
+---------------+---------------+------+------+---------+-------+
| USER          | varchar(64)   | NO   |      | NULL    |       |
| ERROR_NUMBER  | bigint(64)    | NO   |      | NULL    |       |
| ERROR_MESSAGE | varchar(1024) | NO   |      | NULL    |       |
| ERROR_COUNT   | bigint(64)    | NO   |      | NULL    |       |
| WARNING_COUNT | bigint(64)    | NO   |      | NULL    |       |
| FIRST_SEEN    | timestamp     | YES  |      | NULL    |       |
| LAST_SEEN     | timestamp     | YES  |      | NULL    |       |
+---------------+---------------+------+------+---------+-------+
7 rows in set (0.00 sec)
```

字段说明如下：

- USER：已认证的用户。
- ERROR_NUMBER：返回的与 MySQL 兼容的错误码。
- ERROR_MESSAGE：与错误码匹配的错误消息（预处理语句形式）。
- ERROR_COUNT：返回此错误给用户的次数。
- WARNING_COUNT：返回此警告给用户的次数。
- FIRST_SEEN：首次向用户发送此错误（或警告）的时间。
- LAST_SEEN：最近一次向用户发送此错误（或警告）的时间。

以下示例显示了客户端连接到本地 TiDB 服务器时生成的警告。执行 FLUSH CLIENT_ERRORS_SUMMARY 语句后，会重置汇总。

```
SELECT 0/0;
SELECT * FROM CLIENT_ERRORS_SUMMARY_BY_USER;
FLUSH CLIENT_ERRORS_SUMMARY;
SELECT * FROM CLIENT_ERRORS_SUMMARY_BY_USER;
```

输出结果如下：

```
+-----+
| 0/0 |
+-----+
| NULL |
+-----+
1 row in set, 1 warning (0.00 sec)

+--
    ↪ ----+--------------+---------------+-------------+---------------+---------------------+----------------
    ↪
| USER | ERROR_NUMBER | ERROR_MESSAGE | ERROR_COUNT | WARNING_COUNT | FIRST_SEEN          |
    ↪ LAST_SEEN           |
+--
    ↪ ----+--------------+---------------+-------------+---------------+---------------------+----------------
    ↪
| root |         1365 | Division by 0 |           0 |             1 | 2021-03-18 13:05:36 |
    ↪ 2021-03-18 13:05:36 |
+--
    ↪ ----+--------------+---------------+-------------+---------------+---------------------+----------------
    ↪
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

Empty set (0.00 sec)
```

14.11.18.2.5 CLIENT_ERRORS_SUMMARY_GLOBAL

CLIENT_ERRORS_SUMMARY_GLOBAL 表汇总了已返回给连接到 TiDB 服务器的客户端的 SQL 错误和警告。这些错误和警告包括：

- 格式错误的 SQL 语句。
- 除以零错误。
- 尝试插入超出范围或重复的键值。
- 权限错误。
- 表不存在。

以上错误通过 MySQL 服务器协议返回给客户端，此时应用程序应在客户端采取适当操作。information_schema ↪ .CLIENT_ERRORS_SUMMARY_BY_GLOBAL 表提供了高级概述，在应用程序无法正确处理（或记录）TiDB 服务器返回的错误的情况下很有用。

可以使用 FLUSH CLIENT_ERRORS_SUMMARY 语句重置汇总的计数。所汇总的是每个 TiDB 服务器的本地数据，并且只保留在内存中。如果 TiDB 服务器重新启动，会丢失汇总信息。

```
USE information_schema;
DESC CLIENT_ERRORS_SUMMARY_GLOBAL;
```

输出结果如下：

```
+---------------+---------------+------+------+---------+-------+
| Field         | Type          | Null | Key  | Default | Extra |
+---------------+---------------+------+------+---------+-------+
| ERROR_NUMBER  | bigint(64)    | NO   |      | NULL    |       |
| ERROR_MESSAGE | varchar(1024) | NO   |      | NULL    |       |
| ERROR_COUNT   | bigint(64)    | NO   |      | NULL    |       |
| WARNING_COUNT | bigint(64)    | NO   |      | NULL    |       |
| FIRST_SEEN    | timestamp     | YES  |      | NULL    |       |
| LAST_SEEN     | timestamp     | YES  |      | NULL    |       |
+---------------+---------------+------+------+---------+-------+
6 rows in set (0.00 sec)
```

字段说明如下：

- ERROR_NUMBER：返回的与 MySQL 兼容的错误码。
- ERROR_MESSAGE：与错误码匹配的错误消息（预处理语句形式）。
- ERROR_COUNT：返回此错误的次数。
- WARNING_COUNT：返回此警告的次数。
- FIRST_SEEN：首次返回此错误（或警告）的时间。
- LAST_SEEN：最近一次发送此错误（或警告）的时间。

以下示例显示了客户端连接到本地 TiDB 服务器时生成的警告。执行 FLUSH CLIENT_ERRORS_SUMMARY 语句后，会重置汇总。

```
SELECT 0/0;
SELECT * FROM CLIENT_ERRORS_SUMMARY_GLOBAL;
FLUSH CLIENT_ERRORS_SUMMARY;
SELECT * FROM CLIENT_ERRORS_SUMMARY_GLOBAL;
```

输出结果如下：

```
+-----+
| 0/0 |
+-----+
| NULL |
+-----+
1 row in set, 1 warning (0.00 sec)

+--
    ↪ ------------+---------------+-------------+---------------+---------------------+---------------------+
    ↪
| ERROR_NUMBER | ERROR_MESSAGE | ERROR_COUNT | WARNING_COUNT | FIRST_SEEN          | LAST_SEEN
    ↪            |
+--
    ↪ ------------+---------------+-------------+---------------+---------------------+---------------------+
    ↪
|         1365 | Division by 0 |           0 |             1 | 2021-03-18 13:10:51 | 2021-03-18
    ↪ 13:10:51 |
+--
    ↪ ------------+---------------+-------------+---------------+---------------------+---------------------+
    ↪
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.00 sec)

Empty set (0.00 sec)
```

14.11.18.2.6 CHARACTER_SETS

CHARACTER_SETS 表提供字符集相关的信息。TiDB 目前仅支持部分字符集。

```
USE INFORMATION_SCHEMA;
DESC CHARACTER_SETS;
```

输出结果如下：

```
+----------------------+-------------+------+------+---------+-------+
| Field                | Type        | Null | Key  | Default | Extra |
+----------------------+-------------+------+------+---------+-------+
| CHARACTER_SET_NAME   | varchar(32) | YES  |      | NULL    |       |
```

```
| DEFAULT_COLLATE_NAME | varchar(32) | YES  |      | NULL    |       |
| DESCRIPTION          | varchar(60) | YES  |      | NULL    |       |
| MAXLEN               | bigint(3)   | YES  |      | NULL    |       |
+----------------------+-------------+------+------+---------+-------+
4 rows in set (0.00 sec)
```

查看 CHARACTER_SETS 表的信息：

```
SELECT * FROM `CHARACTER_SETS`;
```

输出结果如下：

```
+--------------------+----------------------+-----------------------------------+--------+
| CHARACTER_SET_NAME | DEFAULT_COLLATE_NAME | DESCRIPTION                       | MAXLEN |
+--------------------+----------------------+-----------------------------------+--------+
| ascii              | ascii_bin            | US ASCII                          |      1 |
| binary             | binary               | binary                            |      1 |
| gbk                | gbk_chinese_ci       | Chinese Internal Code Specification |      2 |
| latin1             | latin1_bin           | Latin1                            |      1 |
| utf8               | utf8_bin             | UTF-8 Unicode                     |      3 |
| utf8mb4            | utf8mb4_bin          | UTF-8 Unicode                     |      4 |
+--------------------+----------------------+-----------------------------------+--------+
6 rows in set (0.00 sec)
```

CHARACTER_SETS 表中列的含义如下：

- CHARACTER_SET_NAME：字符集名称
- DEFAULT_COLLATE_NAME：字符集的默认排序规则名称
- DESCRIPTION：字符集的描述信息
- MAXLEN：该字符集存储一个字符所需要的最大字节数

14.11.18.2.7 CLUSTER_CONFIG

CLUSTER_CONFIG 表用于获取集群当前所有组件实例的配置。在 TiDB 早期的版本，用户需要逐个访问各个实例的 HTTP API 才能收集到所有组件配置。TiDB v4.0 后，该表的引入提高了易用性。

```
USE information_schema;
DESC cluster_config;
```

```
+----------+--------------+------+------+---------+-------+
| Field    | Type         | Null | Key  | Default | Extra |
+----------+--------------+------+------+---------+-------+
| TYPE     | varchar(64)  | YES  |      | NULL    |       |
| INSTANCE | varchar(64)  | YES  |      | NULL    |       |
| KEY      | varchar(256) | YES  |      | NULL    |       |
| VALUE    | varchar(128) | YES  |      | NULL    |       |
+----------+--------------+------+------+---------+-------+
```

字段解释：

- TYPE：节点的类型，可取值为 tidb，pd 和 tikv。
- INSTANCE：节点的服务地址。
- KEY：配置项名。
- VALUE：配置项值。

以下示例查询 TiKV 实例的 coprocessor 相关配置：

```
SELECT * FROM cluster_config WHERE type='tikv' AND `key` LIKE 'coprocessor%';
```

```
+------+-----------------+-----------------------------------+---------+
| TYPE | INSTANCE        | KEY                               | VALUE   |
+------+-----------------+-----------------------------------+---------+
| tikv | 127.0.0.1:20165 | coprocessor.batch-split-limit     | 10      |
| tikv | 127.0.0.1:20165 | coprocessor.region-max-keys       | 1440000 |
| tikv | 127.0.0.1:20165 | coprocessor.region-max-size       | 144MiB  |
| tikv | 127.0.0.1:20165 | coprocessor.region-split-keys     | 960000  |
| tikv | 127.0.0.1:20165 | coprocessor.region-split-size     | 96MiB   |
| tikv | 127.0.0.1:20165 | coprocessor.split-region-on-table | false   |
+------+-----------------+-----------------------------------+---------+
6 rows in set (0.00 sec)
```

14.11.18.2.8 CLUSTER_HARDWARE

集群硬件表 CLUSTER_HARDWARE 提供了集群各实例所在服务器的硬件信息。

```
USE information_schema;
DESC cluster_hardware;
```

```
+-------------+--------------+------+------+---------+-------+
| Field       | Type         | Null | Key  | Default | Extra |
+-------------+--------------+------+------+---------+-------+
| TYPE        | varchar(64)  | YES  |      | NULL    |       |
| INSTANCE    | varchar(64)  | YES  |      | NULL    |       |
| DEVICE_TYPE | varchar(64)  | YES  |      | NULL    |       |
| DEVICE_NAME | varchar(64)  | YES  |      | NULL    |       |
| NAME        | varchar(256) | YES  |      | NULL    |       |
| VALUE       | varchar(128) | YES  |      | NULL    |       |
+-------------+--------------+------+------+---------+-------+
6 rows in set (0.00 sec)
```

字段解释：

- TYPE：对应集群信息表information_schema.cluster_info 中的 TYPE 字段，可取值为 tidb，pd 和 tikv。

- INSTANCE：对应于集群信息表information_schema.cluster_info 中的 INSTANCE 字段。
- DEVICE_TYPE：硬件类型。目前可以查询的硬件类型有 cpu、memory、disk 和 net。
- DEVICE_NAME：硬件名。对于不同的 DEVICE_TYPE，DEVICE_NAME 的取值不同。
  - cpu：硬件名为 cpu。
  - memory：硬件名为 memory。
  - disk：磁盘名。
  - net：网卡名。
- NAME：硬件不同的信息名，比如 cpu 有 cpu-logical-cores 和 cpu-physical-cores 两个信息名，表示逻辑核心数量和物理核心数量。
- VALUE：对应硬件信息的值。例如磁盘容量和 CPU 核数。

查询集群 CPU 信息的示例如下：

```
SELECT * FROM cluster_hardware WHERE device_type='cpu' AND device_name='cpu' AND name LIKE '%
    ↪ cores';
```

```
+------+-----------------+-------------+-------------+--------------------+-------+
| TYPE | INSTANCE        | DEVICE_TYPE | DEVICE_NAME | NAME               | VALUE |
+------+-----------------+-------------+-------------+--------------------+-------+
| tidb | 0.0.0.0:4000    | cpu         | cpu         | cpu-logical-cores  | 16    |
| tidb | 0.0.0.0:4000    | cpu         | cpu         | cpu-physical-cores | 8     |
| pd   | 127.0.0.1:2379  | cpu         | cpu         | cpu-logical-cores  | 16    |
| pd   | 127.0.0.1:2379  | cpu         | cpu         | cpu-physical-cores | 8     |
| tikv | 127.0.0.1:20165 | cpu         | cpu         | cpu-logical-cores  | 16    |
| tikv | 127.0.0.1:20165 | cpu         | cpu         | cpu-physical-cores | 8     |
+------+-----------------+-------------+-------------+--------------------+-------+
6 rows in set (0.03 sec)
```

14.11.18.2.9 CLUSTER_INFO

集群拓扑表 CLUSTER_INFO 提供集群当前的拓扑信息，以及各个节点的版本信息、版本对应的 Git Hash、各节点的启动时间、各实例的运行时间。

```
USE information_schema;
desc cluster_info;
```

```
+----------------+-------------+------+------+---------+-------+
| Field          | Type        | Null | Key  | Default | Extra |
+----------------+-------------+------+------+---------+-------+
| TYPE           | varchar(64) | YES  |      | NULL    |       |
| INSTANCE       | varchar(64) | YES  |      | NULL    |       |
| STATUS_ADDRESS | varchar(64) | YES  |      | NULL    |       |
| VERSION        | varchar(64) | YES  |      | NULL    |       |
| GIT_HASH       | varchar(64) | YES  |      | NULL    |       |
| START_TIME     | varchar(32) | YES  |      | NULL    |       |
```

```
| UPTIME         | varchar(32) | YES  |      | NULL    |       |
| SERVER_ID      | bigint(21)  | YES  |      | NULL    |       |
+----------------+-------------+------+------+---------+-------+
8 rows in set (0.01 sec)
```

字段解释：

- TYPE：节点类型，目前节点的可取值为 tidb，pd 和 tikv。
- INSTANCE：实例地址，为 IP:PORT 格式的字符串。
- STATUS_ADDRESS：HTTP API 的服务地址。部分 tikv-ctl、pd-ctl 或 tidb-ctl 命令会使用到 HTTP API 和该地址。用户也可以通过该地址获取一些额外的集群信息，详情可参考 HTTP API 文档。
- VERSION：对应节点的语义版本号。TiDB 版本为了兼容 MySQL 的版本号，以 ${mysql-version}-${tidb-version} 的格式展示版本号。
- GIT_HASH：编译节点版本时的 Git Commit Hash，用于识别两个节点是否是绝对一致的版本。
- START_TIME：对应节点的启动时间。
- UPTIME：对应节点已经运行的时间。
- SERVER_ID：对应节点的服务器 ID。

```
SELECT * FROM cluster_info;
```

```
+--
    ↪ ----+----------------+----------------+-------------+----------------------------------------+------
    ↪
| TYPE | INSTANCE        | STATUS_ADDRESS   | VERSION      | GIT_HASH
    ↪                                    | START_TIME                | UPTIME              |
+--
    ↪ ----+----------------+----------------+-------------+----------------------------------------+------
    ↪
| tidb | 0.0.0.0:4000    | 0.0.0.0:10080    | 4.0.0-beta.2 | 0
    ↪ df3b74f55f8f8fbde39bbd5d471783f49dc10f7 | 2020-07-05T09:25:53-06:00 | 26h39m4.352862693s
    ↪  |
| pd   | 127.0.0.1:2379  | 127.0.0.1:2379   | 4.1.0-alpha  | 1
    ↪ ad59bcbf36d87082c79a1fffa3b0895234ac862 | 2020-07-05T09:25:47-06:00 | 26h39m10.352868103s
    ↪  |
| tikv | 127.0.0.1:20165 | 127.0.0.1:20180 | 4.1.0-alpha  |
    ↪ b45e052df8fb5d66aa8b3a77b5c992ddbfbb79df | 2020-07-05T09:25:50-06:00 | 26h39m7.352869963s
    ↪   |
+--
    ↪ ----+----------------+----------------+-------------+----------------------------------------+------
    ↪
3 rows in set (0.00 sec)
```

14.11.18.2.10 CLUSTER_LOAD

集群负载表 CLUSTER_LOAD 提供集群各个实例所在服务器的当前负载信息。

```
USE information_schema;
DESC cluster_load;
```

```
+-------------+--------------+------+------+---------+-------+
| Field       | Type         | Null | Key  | Default | Extra |
+-------------+--------------+------+------+---------+-------+
| TYPE        | varchar(64)  | YES  |      | NULL    |       |
| INSTANCE    | varchar(64)  | YES  |      | NULL    |       |
| DEVICE_TYPE | varchar(64)  | YES  |      | NULL    |       |
| DEVICE_NAME | varchar(64)  | YES  |      | NULL    |       |
| NAME        | varchar(256) | YES  |      | NULL    |       |
| VALUE       | varchar(128) | YES  |      | NULL    |       |
+-------------+--------------+------+------+---------+-------+
6 rows in set (0.00 sec)
```

字段解释：

- TYPE：对应于节点信息表information_schema.cluster_info 中的 TYPE 字段，可取值为 tidb，pd 和 tikv。
- INSTANCE：对应于节点信息表information_schema.cluster_info 中的 INSTANCE 字段。
- DEVICE_TYPE：硬件类型，目前可以查询的硬件类型有 cpu、memory、disk 和 net。
- DEVICE_NAME：硬件名。对于不同的 DEVICE_TYPE，DEVICE_NAME 取值不同。
  - cpu：硬件名为 cpu。
  - disk：磁盘名。
  - net：网卡名。
  - memory：硬件名为 memory。
- NAME：不同的负载类型。例如 cpu 有 load1/load5/load15 三个负载类型，分别表示 cpu 在 1min/5min/15min 内的平均负载。
- VALUE：硬件负载的值，例如 cpu 在 1min/5min/15min 内的平均负载。

查询集群当前的 CPU 负载信息示例如下：

```
SELECT * FROM cluster_load WHERE device_type='cpu' AND device_name='cpu';
```

```
+------+-----------------+-------------+-------------+--------+-------+
| TYPE | INSTANCE        | DEVICE_TYPE | DEVICE_NAME | NAME   | VALUE |
+------+-----------------+-------------+-------------+--------+-------+
| tidb | 0.0.0.0:4000    | cpu         | cpu         | load1  | 0.13  |
| tidb | 0.0.0.0:4000    | cpu         | cpu         | load5  | 0.25  |
| tidb | 0.0.0.0:4000    | cpu         | cpu         | load15 | 0.31  |
| pd   | 127.0.0.1:2379  | cpu         | cpu         | load1  | 0.13  |
| pd   | 127.0.0.1:2379  | cpu         | cpu         | load5  | 0.25  |
| pd   | 127.0.0.1:2379  | cpu         | cpu         | load15 | 0.31  |
| tikv | 127.0.0.1:20165 | cpu         | cpu         | load1  | 0.13  |
| tikv | 127.0.0.1:20165 | cpu         | cpu         | load5  | 0.25  |
```

```
| tikv | 127.0.0.1:20165 | cpu         | cpu         | load15 | 0.31  |
+------+-----------------+-------------+-------------+--------+-------+
9 rows in set (1.50 sec)
```

14.11.18.2.11 CLUSTER_LOG

集群日志表 CLUSTER_LOG 表用于查询集群当前所有 TiDB/PD/TiKV 节点日志。它通过将查询条件下推到各个节点，降低了日志查询对集群的影响。该表的查询性能优于 grep 命令。

TiDB 4.0 版本之前，要获取集群的日志，用户需要逐个登录各个节点汇总日志。TiDB 4.0 的集群日志表提供了一个全局且时间有序的日志搜索结果，为跟踪全链路事件提供了便利的手段。例如按照某一个 `region id` 搜索日志，可以查询该 Region 生命周期内的所有日志；类似地，通过慢日志的 `txn id` 搜索全链路日志，可以查询该事务在各个节点扫描的 key 数量以及流量等信息。

```
USE information_schema;
DESC cluster_log;
```

```
+----------+------------------+------+------+---------+-------+
| Field    | Type             | Null | Key  | Default | Extra |
+----------+------------------+------+------+---------+-------+
| TIME     | varchar(32)      | YES  |      | NULL    |       |
| TYPE     | varchar(64)      | YES  |      | NULL    |       |
| INSTANCE | varchar(64)      | YES  |      | NULL    |       |
| LEVEL    | varchar(8)       | YES  |      | NULL    |       |
| MESSAGE  | var_string(1024) | YES  |      | NULL    |       |
+----------+------------------+------+------+---------+-------+
5 rows in set (0.00 sec)
```

字段解释：

- `TIME`：日志打印时间。
- `TYPE`：节点的类型，可取值为 `tidb`，`pd` 和 `tikv`。
- `INSTANCE`：节点的服务地址。
- `LEVEL`：日志级别。
- `MESSAGE`：日志内容。

> 注意：
>
> - 日志表的所有字段都会下推到对应节点执行，所以为了降低使用集群日志表的开销，必须指定搜索关键字以及时间范围，然后尽可能地指定更多的条件。例如 `select * from cluster_log where message like '%ddl%' and time > '2020-05-18 20:40:00' and time<'2020-05-18 21:40:00' and type='tidb'`。

- message 字段支持 like 和 regexp 正则表达式，对应的 pattern 会编译为 regexp。同时指定多个 message 条件，相当于 grep 命令的 pipeline 形式，例如：select * from cluster_log where message like 'coprocessor%' and message regexp '.*slow.*' and time > '2020-05-18 20:40:00' and time<'2020-05-18 21:40:00' 相当于在集群所有节点执行 grep 'coprocessor' xxx.log | grep -E '.*slow.*'。

查询某个 DDL 的执行过程示例如下：

```
SELECT time,instance,left(message,150) FROM cluster_log WHERE message LIKE '%ddl%job%ID.80%' AND type='tidb' AND time > '2020-05-18 20:40:00' AND time < '2020-05-18 21:40:00'
```

```
+----------------------+---------------+--------------------------------------------------------------------------------------------------------------------------------------------------------------+
| time                 | instance      | left(message,150)                                                                                                                                            |
+----------------------+---------------+--------------------------------------------------------------------------------------------------------------------------------------------------------------+
| 2020/05/18 21:37:54.784 | 127.0.0.1:4002 | [ddl_worker.go:261] ["[ddl] add DDL jobs"] ["batch count"=1] [jobs="ID:80, Type:create table, State:none, SchemaState:none, SchemaID:1, TableID:79, Ro |
| 2020/05/18 21:37:54.784 | 127.0.0.1:4002 | [ddl.go:477] ["[ddl] start DDL job"] [job="ID:80, Type:create table, State:none, SchemaState:none, SchemaID:1, TableID:79, RowCount:0, ArgLen:1, start |
| 2020/05/18 21:37:55.327 | 127.0.0.1:4000 | [ddl_worker.go:568] ["[ddl] run DDL job"] [worker="worker 1, tp general"] [job="ID:80, Type:create table, State:none, SchemaState:none, SchemaID:1, Ta |
| 2020/05/18 21:37:55.381 | 127.0.0.1:4000 | [ddl_worker.go:763] ["[ddl] wait latest schema version changed"] [worker="worker 1, tp general"] [ver=70] ["take time"=50.809848ms] [job ="ID:80, Type: |
| 2020/05/18 21:37:55.382 | 127.0.0.1:4000 | [ddl_worker.go:359] ["[ddl] finish DDL job"] [worker ="worker 1, tp general"] [job="ID:80, Type:create table, State:synced, SchemaState:public , SchemaI |
| 2020/05/18 21:37:55.786 | 127.0.0.1:4002 | [ddl.go:509] ["[ddl] DDL job is finished"] [jobID =80]                                                                                                       |
+----------------------+---------------+--------------------------------------------------------------------------------------------------------------------------------------------------------------+
```

上面查询结果记录了一个 DDL 执行的过程：

- 用户将 DDL JOB ID 为 80 的请求发给 127.0.0.1:4002 TiDB 节点。
- 127.0.0.1:4000 TiDB 节点处理这个 DDL 请求，说明此时 127.0.0.1:4000 节点是 DDL owner。
- DDL JOB ID 为 80 的请求处理完成。

14.11.18.2.12 CLUSTER_SYSTEMINFO

内核参数表 CLUSTER_SYSTEMINFO 用于查询集群所有实例所在服务器的内核配置信息。目前支持查询 sysctl 的信息。

```
USE information_schema;
DESC cluster_systeminfo;
```

```
+-------------+--------------+------+------+---------+-------+
| Field       | Type         | Null | Key  | Default | Extra |
+-------------+--------------+------+------+---------+-------+
| TYPE        | varchar(64)  | YES  |      | NULL    |       |
| INSTANCE    | varchar(64)  | YES  |      | NULL    |       |
| SYSTEM_TYPE | varchar(64)  | YES  |      | NULL    |       |
| SYSTEM_NAME | varchar(64)  | YES  |      | NULL    |       |
| NAME        | varchar(256) | YES  |      | NULL    |       |
| VALUE       | varchar(128) | YES  |      | NULL    |       |
+-------------+--------------+------+------+---------+-------+
6 rows in set (0.00 sec)
```

字段解释：

- TYPE：对应于节点信息表information_schema.cluster_info 中的 TYPE 字段，可取值为 tidb，pd 和 tikv。
- INSTANCE：对应于节点信息表information_schema.cluster_info 中的 INSTANCE 字段。
- SYSTEM_TYPE：系统类型，目前可以查询的系统类型有 system。
- SYSTEM_NAME：目前可以查询的 SYSTEM_NAME 为 sysctl。
- NAME：sysctl 对应的配置名。
- VALUE：sysctl 对应配置项的值。

查询集群所有服务器的内核版本示例如下：

```
SELECT * FROM cluster_systeminfo WHERE name LIKE '%kernel.osrelease%'
```

```
+--
    ↪ ----+------------------+-------------+-------------+------------------+-------------------------+
    ↪
| TYPE | INSTANCE          | SYSTEM_TYPE | SYSTEM_NAME | NAME             | VALUE
    ↪                       |
+--
    ↪ ----+------------------+-------------+-------------+------------------+-------------------------+
    ↪
| tidb | 172.16.5.40:4008  | system      | sysctl      | kernel.osrelease | 3.10.0-862.14.4.el7.
    ↪ x86_64 |
```

```
| pd   | 172.16.5.40:20379 | system      | sysctl      | kernel.osrelease | 3.10.0-862.14.4.el7.
    ↪ x86_64 |
| tikv | 172.16.5.40:21150 | system      | sysctl      | kernel.osrelease | 3.10.0-862.14.4.el7.
    ↪ x86_64 |
+--
    ↪ ----+------------------+-------------+-------------+------------------+----------------------------+
    ↪
```

14.11.18.2.13 COLLATIONS

COLLATIONS 表提供了 CHARACTER_SETS 表中字符集对应的排序规则列表。目前 TiDB 包含该表仅为兼容 MySQL。

```
USE information_schema;
DESC collations;
```

```
+--------------------+-------------+------+------+---------+-------+
| Field              | Type        | Null | Key  | Default | Extra |
+--------------------+-------------+------+------+---------+-------+
| COLLATION_NAME     | varchar(32) | YES  |      | NULL    |       |
| CHARACTER_SET_NAME | varchar(32) | YES  |      | NULL    |       |
| ID                 | bigint(11)  | YES  |      | NULL    |       |
| IS_DEFAULT         | varchar(3)  | YES  |      | NULL    |       |
| IS_COMPILED        | varchar(3)  | YES  |      | NULL    |       |
| SORTLEN            | bigint(3)   | YES  |      | NULL    |       |
+--------------------+-------------+------+------+---------+-------+
6 rows in set (0.00 sec)
```

```
SELECT * FROM collations WHERE character_set_name='utf8mb4';
```

```
+--------------------+--------------------+------+------------+-------------+---------+
| COLLATION_NAME     | CHARACTER_SET_NAME | ID   | IS_DEFAULT | IS_COMPILED | SORTLEN |
+--------------------+--------------------+------+------------+-------------+---------+
| utf8mb4_bin        | utf8mb4            |   46 | Yes        | Yes         |       1 |
| utf8mb4_general_ci | utf8mb4            |   45 |            | Yes         |       1 |
| utf8mb4_unicode_ci | utf8mb4            |  224 |            | Yes         |       1 |
+--------------------+--------------------+------+------------+-------------+---------+
3 rows in set (0.001 sec)
```

COLLATIONS 表中列的含义如下：

- COLLATION_NAME：排序规则名称
- CHARACTER_SET_NAME：排序规则所属的字符集名称
- ID：排序规则的 ID
- IS_DEFAULT：该排序规则是否是所属字符集的默认排序规则
- IS_COMPILED：字符集是否编译到服务器中
- SORTLEN：排序规则在对字符进行排序时，所分配内存的最小长度

14.11.18.2.14 COLLATION_CHARACTER_SET_APPLICABILITY

COLLATION_CHARACTER_SET_APPLICABILITY 表将排序规则映射至适用的字符集名称。和 COLLATIONS 表一样，包含此表只是为了兼容 MySQL。

```
USE INFORMATION_SCHEMA;
DESC COLLATION_CHARACTER_SET_APPLICABILITY;
```

输出结果如下：

```
+--------------------+-------------+------+------+---------+-------+
| Field              | Type        | Null | Key  | Default | Extra |
+--------------------+-------------+------+------+---------+-------+
| COLLATION_NAME     | varchar(32) | NO   |      | NULL    |       |
| CHARACTER_SET_NAME | varchar(32) | NO   |      | NULL    |       |
+--------------------+-------------+------+------+---------+-------+
2 rows in set (0.00 sec)
```

查看 COLLATION_CHARACTER_SET_APPLICABILITY 表中 utf8mb4 字符集的排序规则映射：

```
SELECT * FROM COLLATION_CHARACTER_SET_APPLICABILITY WHERE character_set_name='utf8mb4';
```

输出结果如下：

```
+--------------------+--------------------+
| COLLATION_NAME     | CHARACTER_SET_NAME |
+--------------------+--------------------+
| utf8mb4_bin        | utf8mb4            |
| utf8mb4_general_ci | utf8mb4            |
| utf8mb4_unicode_ci | utf8mb4            |
+--------------------+--------------------+
3 rows in set (0.00 sec)
```

COLLATION_CHARACTER_SET_APPLICABILITY 表中列的含义如下：

- COLLATION_NAME：排序规则名称
- CHARACTER_SET_NAME：排序规则所属的字符集名称

14.11.18.2.15 COLUMNS

COLUMNS 表提供了表的所有列的信息。

```
USE INFORMATION_SCHEMA;
DESC COLUMNS;
```

输出结果如下：

```
+--------------------------+---------------+------+------+---------+-------+
| Field                    | Type          | Null | Key  | Default | Extra |
+--------------------------+---------------+------+------+---------+-------+
| TABLE_CATALOG            | varchar(512)  | YES  |      | NULL    |       |
| TABLE_SCHEMA             | varchar(64)   | YES  |      | NULL    |       |
| TABLE_NAME               | varchar(64)   | YES  |      | NULL    |       |
| COLUMN_NAME              | varchar(64)   | YES  |      | NULL    |       |
| ORDINAL_POSITION         | bigint(64)    | YES  |      | NULL    |       |
| COLUMN_DEFAULT           | text          | YES  |      | NULL    |       |
| IS_NULLABLE              | varchar(3)    | YES  |      | NULL    |       |
| DATA_TYPE                | varchar(64)   | YES  |      | NULL    |       |
| CHARACTER_MAXIMUM_LENGTH | bigint(21)    | YES  |      | NULL    |       |
| CHARACTER_OCTET_LENGTH   | bigint(21)    | YES  |      | NULL    |       |
| NUMERIC_PRECISION        | bigint(21)    | YES  |      | NULL    |       |
| NUMERIC_SCALE            | bigint(21)    | YES  |      | NULL    |       |
| DATETIME_PRECISION       | bigint(21)    | YES  |      | NULL    |       |
| CHARACTER_SET_NAME       | varchar(32)   | YES  |      | NULL    |       |
| COLLATION_NAME           | varchar(32)   | YES  |      | NULL    |       |
| COLUMN_TYPE              | text          | YES  |      | NULL    |       |
| COLUMN_KEY               | varchar(3)    | YES  |      | NULL    |       |
| EXTRA                    | varchar(30)   | YES  |      | NULL    |       |
| PRIVILEGES               | varchar(80)   | YES  |      | NULL    |       |
| COLUMN_COMMENT           | varchar(1024) | YES  |      | NULL    |       |
| GENERATION_EXPRESSION    | text          | NO   |      | NULL    |       |
+--------------------------+---------------+------+------+---------+-------+
21 rows in set (0.00 sec)
```

创建表 `test.t1`，并查询 `COLUMNS` 表的信息：

```
CREATE TABLE test.t1 (a int);
SELECT * FROM COLUMNS WHERE table_schema='test' AND TABLE_NAME='t1'\G
```

输出结果如下：

```
*************************** 1. row ***************************
           TABLE_CATALOG: def
            TABLE_SCHEMA: test
              TABLE_NAME: t1
             COLUMN_NAME: a
        ORDINAL_POSITION: 1
          COLUMN_DEFAULT: NULL
             IS_NULLABLE: YES
               DATA_TYPE: int
CHARACTER_MAXIMUM_LENGTH: NULL
  CHARACTER_OCTET_LENGTH: NULL
       NUMERIC_PRECISION: 11
```

```
           NUMERIC_SCALE: 0
       DATETIME_PRECISION: NULL
       CHARACTER_SET_NAME: NULL
           COLLATION_NAME: NULL
              COLUMN_TYPE: int(11)
               COLUMN_KEY:
                    EXTRA:
               PRIVILEGES: select,insert,update,references
           COLUMN_COMMENT:
    GENERATION_EXPRESSION:
1 row in set (0.02 sec)
```

COLUMNS 表中列的含义如下：

- TABLE_CATALOG：包含列的表所属的目录的名称。该值始终为 def。
- TABLE_SCHEMA：包含列的表所属的数据库的名称。
- TABLE_NAME：包含列的表的名称。
- COLUMN_NAME：列的名称。
- ORDINAL_POSITION：表中列的位置。
- COLUMN_DEFAULT：列的默认值。如果列的显式默认值为 NULL，或者列定义中不包含 default 子句，则此值为 NULL。
- IS_NULLABLE：列的可空性。如果列中可以存储空值，则该值为 YES，否则为 NO。
- DATA_TYPE：列的数据类型。
- CHARACTER_MAXIMUM_LENGTH：对于字符串列，以字符为单位的最大长度。
- CHARACTER_OCTET_LENGTH：对于字符串列，以字节为单位的最大长度。
- NUMERIC_PRECISION：对于数字列，为数字精度。
- NUMERIC_SCALE：对于数字列，为数字刻度。
- DATETIME_PRECISION：对于时间列，小数秒精度。
- CHARACTER_SET_NAME：对于字符串列，字符集名称。
- COLLATION_NAME：对于字符串列，排序规则名称。
- COLUMN_TYPE：列类型。
- COLUMN_KEY：该列是否被索引。具体显示如下：
  - 如果此值为空，则该列要么未被索引，要么被索引且是多列非唯一索引中的第二列。
  - 如果此值是 PRI，则该列是主键，或者是多列主键中的一列。
  - 如果此值是 UNI，则该列是唯一索引的第一列。
  - 如果此值是 MUL，则该列是非唯一索引的第一列，在该列中允许给定值的多次出现。
- EXTRA：关于给定列的任何附加信息。
- PRIVILEGES：当前用户对该列拥有的权限。目前在 TiDB 中，此值为定值，一直为 select,insert,update ↪ ,references。
- COLUMN_COMMENT：列定义中包含的注释。
- GENERATION_EXPRESSION：对于生成的列，显示用于计算列值的表达式。对于未生成的列为空。

对应的 SHOW 语句如下：

```
SHOW COLUMNS FROM t1 FROM test;
```

输出结果如下：

```
+-------+---------+------+------+---------+-------+
| Field | Type    | Null | Key  | Default | Extra |
+-------+---------+------+------+---------+-------+
| a     | int(11) | YES  |      | NULL    |       |
+-------+---------+------+------+---------+-------+
1 row in set (0.00 sec)
```

14.11.18.2.16 DATA_LOCK_WAITS

DATA_LOCK_WAITS 表展示了集群中所有 TiKV 节点上当前正在发生的等锁情况，包括悲观锁的等锁情况和乐观事务被阻塞的信息。

```
USE information_schema;
DESC data_lock_waits;
```

```
+-----------------------+---------------------+------+------+---------+-------+
| Field                 | Type                | Null | Key  | Default | Extra |
+-----------------------+---------------------+------+------+---------+-------+
| KEY                   | text                | NO   |      | NULL    |       |
| KEY_INFO              | text                | YES  |      | NULL    |       |
| TRX_ID                | bigint(21) unsigned | NO   |      | NULL    |       |
| CURRENT_HOLDING_TRX_ID | bigint(21) unsigned | NO   |      | NULL    |       |
| SQL_DIGEST            | varchar(64)         | YES  |      | NULL    |       |
| SQL_DIGEST_TEXT       | text                | YES  |      | NULL    |       |
+-----------------------+---------------------+------+------+---------+-------+
```

DATA_LOCK_WAITS 表中各列的字段含义如下：

- KEY：正在发生等锁的 key，以十六进制编码的形式显示。
- KEY_INFO：对 KEY 进行解读得出的一些详细信息，见KEY_INFO。
- TRX_ID：正在等锁的事务 ID，即 start_ts。
- CURRENT_HOLDING_TRX_ID：当前持有锁的事务 ID，即 start_ts。
- SQL_DIGEST：当前正在等锁的事务中被阻塞的 SQL 语句的 Digest。
- SQL_DIGEST_TEXT：当前正在等锁的事务中被阻塞的 SQL 语句的归一化形式，即去除了参数和格式的 SQL 语句。与 SQL_DIGEST 对应。

> 警告：
>
> - 仅拥有 PROCESS 权限的用户可以查询该表。
> - 由于实现限制，目前对于乐观事务被阻塞情况的 SQL_DIGEST 和 SQL_DIGEST_TEXT 字段为 null。如果需要知道导致阻塞的 SQL 语句，可以将此表与CLUSTER_TIDB_TRX 进行 JOIN 来获得对应事务的所有 SQL 语句。

- DATA_LOCK_WAITS 表中的信息是在查询时，从所有 TiKV 节点实时获取的。目前，即使加上了 WHERE 查询条件，也无法避免对所有 TiKV 节点都进行信息收集。如果集群规模很大、负载很高，查询该表有造成性能抖动的潜在风险，因此请根据实际情况使用。
- 来自不同 TiKV 节点的信息不能保证是同一时间点的快照。
- SQL_DIGEST 列中的信息（SQL Digest）为 SQL 语句进行归一化后计算得到的哈希值。SQL_DIGEST_TEXT 列中的信息为内部从 Statements Summary 系列表中查询得到，因而存在内部查询不到对应语句的可能性。关于 SQL Digest 和 Statements Summary 相关表的详细说明，请参阅Statement Summary Tables。

KEY_INFO

KEY_INFO 列中展示了对 KEY 列中所给出的 key 的详细信息，以 JSON 格式给出。其包含的信息如下：

- "db_id"：该 key 所属的数据库（schema）的 ID。
- "db_name"：该 key 所属的数据库（schema）的名称。
- "table_id"：该 key 所属的表的 ID。
- "table_name"：该 key 所属的表的名称。
- "partition_id"：该 key 所在的分区（partition）的 ID。
- "partition_name"：该 key 所在的分区（partition）的名称。
- "handle_type"：该 row key（即储存一行数据的 key）的 handle 类型，其可能的值有：
  - "int"：handle 为 int 类型，即 handle 为 row ID
  - "common"：非 int64 类型的 handle，在启用 clustered index 时非 int 类型的主键会显示为此类型
  - "unknown"：当前暂不支持的 handle 类型
- "handle_value"：handle 的值。
- "index_id"：该 index key（即储存索引的 key）所属的 index ID。
- "index_name"：该 index key 所属的 index 名称。
- "index_values"：该 index key 中的 index value。

其中，不适用或当前无法查询到的信息会被省略。比如，row key 的信息中不会包含 index_id、index_name 和 index_values；index key 不会包含 handle_type 和 handle_value；非分区表不会显示 partition_id 和 partition_name；已经被删除掉的表中的 key 的信息无法获取 table_name、db_id、db_name、index_name 等 schema 信息，且无法区分是否为分区表。

注意：

如果一个 key 来自一张启用了分区的表，而在查询时，由于某些原因（例如，其所属的表已经被删除）导致无法查询其所属的 schema 信息，则其所属的分区的 ID 可能会出现在 table_id 字段中。这是因为，TiDB 对不同分区的 key 的编码方式与对几张独立的表的 key 的编码方式一致，因而在缺失 schema 信息时无法确认该 key 属于一张未分区的表还是某张表的一个分区。

示例

```
select * from information_schema.data_lock_waits\G
```

```
*************************** 1. row ***************************
                  KEY: 748000000000000355F728000000000000001
             KEY_INFO: {"db_id":1,"db_name":"test","table_id":53,"table_name":"t","handle_type":
                 ↪ "int","handle_value":"1"}
               TRX_ID: 426790594290122753
CURRENT_HOLDING_TRX_ID: 426790590082449409
            SQL_DIGEST: 38b03afa5debbdf0326a014dbe5012a62c51957f1982b3093e748460f8b00821
       SQL_DIGEST_TEXT: update `t` set `v` = `v` + ? where `id` = ?
1 row in set (0.01 sec)
```

以上查询结果显示，ID 为 426790594290122753 的事务在执行 Digest 为 "38b03afa5debbdf0326a014dbe5012a62c51957f1982b3093e7
↪ "、形如 update `t` set `v` = `v` + ? where `id` = ? 的语句的过程中，试图在 "748000000000000355
↪ F728000000000000001" 这个 key 上获取悲观锁，但是该 key 上的锁目前被 ID 为 426790590082449409 的事务持有。

14.11.18.2.17 DDL_JOBS

DDL_JOBS 表为ADMIN SHOW DDL JOBS 命令提供了一个 INFORMATION_SCHEMA 接口。它提供了 TiDB 集群中 DDL 操作的当前状态和简短历史记录。

```
USE information_schema;
DESC ddl_jobs;
```

```
+--------------+-------------+------+------+---------+-------+
| Field        | Type        | Null | Key  | Default | Extra |
+--------------+-------------+------+------+---------+-------+
| JOB_ID       | bigint(21)  | YES  |      | NULL    |       |
| DB_NAME      | varchar(64) | YES  |      | NULL    |       |
| TABLE_NAME   | varchar(64) | YES  |      | NULL    |       |
| JOB_TYPE     | varchar(64) | YES  |      | NULL    |       |
| SCHEMA_STATE | varchar(64) | YES  |      | NULL    |       |
| SCHEMA_ID    | bigint(21)  | YES  |      | NULL    |       |
| TABLE_ID     | bigint(21)  | YES  |      | NULL    |       |
| ROW_COUNT    | bigint(21)  | YES  |      | NULL    |       |
| START_TIME   | datetime    | YES  |      | NULL    |       |
| END_TIME     | datetime    | YES  |      | NULL    |       |
| STATE        | varchar(64) | YES  |      | NULL    |       |
| QUERY        | varchar(64) | YES  |      | NULL    |       |
+--------------+-------------+------+------+---------+-------+
12 rows in set (0.00 sec)
```

```
SELECT * FROM ddl_jobs LIMIT 3\G
```

```
*************************** 1. row ***************************
      JOB_ID: 44
     DB_NAME: mysql
  TABLE_NAME: opt_rule_blacklist
    JOB_TYPE: create table
SCHEMA_STATE: public
   SCHEMA_ID: 3
    TABLE_ID: 43
   ROW_COUNT: 0
  START_TIME: 2020-07-06 15:24:27
    END_TIME: 2020-07-06 15:24:27
       STATE: synced
       QUERY: CREATE TABLE IF NOT EXISTS mysql.opt_rule_blacklist (
        name char(100) NOT NULL
    );
*************************** 2. row ***************************
      JOB_ID: 42
     DB_NAME: mysql
  TABLE_NAME: expr_pushdown_blacklist
    JOB_TYPE: create table
SCHEMA_STATE: public
   SCHEMA_ID: 3
    TABLE_ID: 41
   ROW_COUNT: 0
  START_TIME: 2020-07-06 15:24:27
    END_TIME: 2020-07-06 15:24:27
       STATE: synced
       QUERY: CREATE TABLE IF NOT EXISTS mysql.expr_pushdown_blacklist (
        name char(100) NOT NULL,
        store_type char(100) NOT NULL DEFAULT 'tikv,tiflash,tidb',
        reason varchar(200)
    );
*************************** 3. row ***************************
      JOB_ID: 40
     DB_NAME: mysql
  TABLE_NAME: stats_top_n
    JOB_TYPE: create table
SCHEMA_STATE: public
   SCHEMA_ID: 3
    TABLE_ID: 39
   ROW_COUNT: 0
  START_TIME: 2020-07-06 15:24:26
    END_TIME: 2020-07-06 15:24:27
       STATE: synced
```

```
       QUERY: CREATE TABLE if not exists mysql.stats_top_n (
        table_id bigint(64) NOT NULL,
        is_index tinyint(2) NOT NULL,
        hist_id bigint(64) NOT NULL,
        value longblob,
        count bigint(64) UNSIGNED NOT NULL,
        index tbl(table_id, is_index, hist_id)
    );
3 rows in set (0.01 sec)
```

14.11.18.2.18 DEADLOCKS

DEADLOCKS 表提供当前 TiDB 节点上最近发生的若干次死锁错误的信息。

```
USE INFORMATION_SCHEMA;
DESC DEADLOCKS;
```

输出结果如下：

```
+-------------------------+---------------------+------+------+---------+-------+
| Field                   | Type                | Null | Key  | Default | Extra |
+-------------------------+---------------------+------+------+---------+-------+
| DEADLOCK_ID             | bigint(21)          | NO   |      | NULL    |       |
| OCCUR_TIME              | timestamp(6)        | YES  |      | NULL    |       |
| RETRYABLE               | tinyint(1)          | NO   |      | NULL    |       |
| TRY_LOCK_TRX_ID         | bigint(21) unsigned | NO   |      | NULL    |       |
| CURRENT_SQL_DIGEST      | varchar(64)         | YES  |      | NULL    |       |
| CURRENT_SQL_DIGEST_TEXT | text                | YES  |      | NULL    |       |
| KEY                     | text                | YES  |      | NULL    |       |
| KEY_INFO                | text                | YES  |      | NULL    |       |
| TRX_HOLDING_LOCK        | bigint(21) unsigned | NO   |      | NULL    |       |
+-------------------------+---------------------+------+------+---------+-------+
```

DEADLOCKS 表中需要用多行来表示同一个死锁事件，每行显示参与死锁的其中一个事务的信息。当该 TiDB 节点记录了多次死锁错误时，需要按照 DEADLOCK_ID 列来区分，相同的 DEADLOCK_ID 表示同一个死锁事件。需要注意，DEADLOCK_ID 并不保证全局唯一，也不会持久化，因而其只能在同一个结果集里表示同一个死锁事件。

DEADLOCKS 表中各列的字段含义如下：

- DEADLOCK_ID：死锁事件的 ID。当表内存在多次死锁错误的信息时，需要使用该列来区分属于不同死锁错误的行。
- OCCUR_TIME：发生该次死锁错误的时间。
- RETRYABLE：该次死锁错误是否可重试。关于可重试的死锁错误的说明，参见可重试的死锁错误小节。
- TRY_LOCK_TRX_ID：试图上锁的事务 ID，即事务的 start_ts。
- CURRENT_SQL_DIGEST：试图上锁的事务中当前正在执行的 SQL 语句的 Digest。
- CURRENT_SQL_DIGEST_TEXT：试图上锁的事务中当前正在执行的 SQL 语句的归一化形式。

- KEY：该事务试图上锁、但是被阻塞的 key，以十六进制编码的形式显示。
- KEY_INFO：对 KEY 进行解读得出的一些详细信息，详见KEY_INFO。
- TRX_HOLDING_LOCK：该 key 上当前持锁并导致阻塞的事务 ID，即事务的 start_ts。

要调整 DEADLOCKS 表中可以容纳的死锁事件数量，可通过 TiDB 配置文件中的pessimistic-txn.deadlock-history-capacity 配置项进行调整，默认容纳最近 10 次死锁错误的信息。

> 注意：
>
> - 仅拥有 PROCESS 权限的用户可以查询该表。
> - CURRENT_SQL_DIGEST 列中的信息 (SQL Digest) 为 SQL 语句进行归一化后计算得到的哈希值。CURRENT_SQL_DIGEST_TEXT 列中的信息为内部从 Statements Summary 系列表中查询得到，因而存在内部查询不到对应语句的可能性。关于 SQL Digest 和 Statements Summary 相关表的详细说明，请参阅Statement Summary Tables。

KEY_INFO

KEY_INFO 列中展示了对 KEY 列中所给出的 key 的详细信息，以 JSON 格式给出。其包含的信息如下：

- "db_id"：该 key 所属的数据库 (schema) 的 ID。
- "db_name"：该 key 所属的数据库 (schema) 的名称。
- "table_id"：该 key 所属的表的 ID。
- "table_name"：该 key 所属的表的名称。
- "partition_id"：该 key 所在的分区 (partition) 的 ID。
- "partition_name"：该 key 所在的分区 (partition) 的名称。
- "handle_type"：该 row key（即储存一行数据的 key）的 handle 类型，其可能的值有：
  - "int"：handle 为 int 类型，即 handle 为 row ID
  - "common"：非 int64 类型的 handle，在启用 clustered index 时非 int 类型的主键会显示为此类型
  - "unknown"：当前暂不支持的 handle 类型
- "handle_value"：handle 的值。
- "index_id"：该 index key（即储存索引的 key）所属的 index ID。
- "index_name"：该 index key 所属的 index 名称。
- "index_values"：该 index key 中的 index value。

其中，不适用或当前无法查询到的信息会被省略。比如，row key 的信息中不会包含 index_id、index_name 和 index_values；index key 不会包含 handle_type 和 handle_value；非分区表不会显示 partition_id 和 partition_name；已经被删除掉的表中的 key 的信息无法获取 table_name、db_id、db_name、index_name 等 schema 信息，且无法区分是否为分区表。

> 注意：

> 如果一个 key 来自一张启用了分区的表，而在查询时，由于某些原因（例如，其所属的表已经被删除）导致无法查询其所属的 schema 信息，则其所属的分区的 ID 可能会出现在 `table_id` 字段中。这是因为，TiDB 对不同分区的 key 的编码方式与对几张独立的表的 key 的编码方式一致，因而在缺失 schema 信息时无法确认该 key 属于一张未分区的表还是某张表的一个分区。

可重试的死锁错误

> 注意：
>
> DEADLOCKS 表中默认不收集可重试的死锁错误的信息。如果需要收集，可通过 TiDB 配置文件中的`pessimistic-txn.deadlock-history-collect-retryable` 配置项进行调整。

当事务 A 被另一个事务 B 已经持有的锁阻塞，而事务 B 直接或间接地被当前事务 A 持有的锁阻塞，将会引发一个死锁错误。这里：

- 情况一：事务 B 可能（直接或间接地）被事务 A 开始后到被阻塞前这段时间内已经执行完成的语句产生的锁阻塞
- 情况二：事务 B 也可能被事务 A 目前正在执行的语句阻塞

对于情况一，TiDB 将会向事务 A 的客户端报告死锁错误，并终止该事务；而对于情况二，事务 A 当前正在执行的语句将在 TiDB 内部被自动重试。例如，假设事务 A 执行了如下语句：

```
UPDATE t SET v = v + 1 WHERE id = 1 OR id = 2;
```

事务 B 则先后执行如下两条语句：

```
UPDATE t SET v = 4 WHERE id = 2;
UPDATE t SET v = 2 WHERE id = 1;
```

那么如果事务 A 先后对 `id = 1` 和 `id = 2` 的两行分别上锁，且两个事务以如下时序运行：

1. 事务 A 对 `id = 1` 的行上锁
2. 事务 B 执行第一条语句并对 `id = 2` 的行上锁
3. 事务 B 执行第二条语句试图对 `id = 1` 的行上锁，被事务 A 阻塞
4. 事务 A 试图对 `id = 2` 的行上锁，被 B 阻塞，形成死锁

对于情况二，由于事务 A 阻塞其它事务的语句也是当前正在执行的语句，因而可以解除当前语句所上的悲观锁（使得事务 B 可以继续运行），并重试当前语句。TiDB 内部使用 key 的哈希值来判断是否属于这种情况。

当可重试的死锁发生时，内部自动重试并不会引起事务报错，因而对客户端透明，但是这种情况的频繁发生可能影响性能。当这种情况发生时，在 TiDB 的日志中可以观察到 `single statement deadlock, retry statement` 字样的日志。

示例 1

假设有如下表定义和初始数据：

```
CREATE TABLE t (id int primary key, v int);
INSERT INTO t VALUES (1, 10), (2, 20);
```

使两个事务按如下顺序执行：

事务 1	事务 2	说明
UPDATE t SET v = 11 WHERE id = 1;		
	UPDATE t SET v = 21 WHERE id = 2;	
UPDATE t SET v = 12 WHERE id = 2;		事务 1 阻塞
	UPDATE t SET v = 22 WHERE id = 1;	事务 2 报出死锁错误

接下来，事务 2 将报出死锁错误。此时，查询 DEADLOCKS 表：

```
SELECT * FROM INFORMATION_SCHEMA.DEADLOCKS;
```

输出结果如下：

```
+--
    ↪ -----------+----------------------------+-----------+--------------------+-------------------------------
    ↪
| DEADLOCK_ID | OCCUR_TIME                 | RETRYABLE | TRY_LOCK_TRX_ID    | CURRENT_SQL_DIGEST
    ↪                                              | CURRENT_SQL_DIGEST_TEXT                 |
    ↪  KEY                                    | KEY_INFO
    ↪
    ↪ | TRX_HOLDING_LOCK   |
+--
    ↪ -----------+----------------------------+-----------+--------------------+-------------------------------
    ↪
|           1 | 2021-08-05 11:09:03.230341 |         0 | 426812829645406216 | 22230766411
    ↪ edb40f27a68dadefc63c6c6970d5827f1e5e22fc97be2c4d8350d | update `t` set `v` = ? where `id`
    ↪  = ? ; | 74800000000000000355F728000000000000002 | {"db_id":1,"db_name":"test","table_id"
    ↪ :53,"table_name":"t","handle_type":"int","handle_value":"2"} | 426812829645406217 |
|           1 | 2021-08-05 11:09:03.230341 |         0 | 426812829645406217 | 22230766411
    ↪ edb40f27a68dadefc63c6c6970d5827f1e5e22fc97be2c4d8350d | update `t` set `v` = ? where `id`
    ↪  = ? ; | 74800000000000000355F728000000000000001 | {"db_id":1,"db_name":"test","table_id"
    ↪ :53,"table_name":"t","handle_type":"int","handle_value":"1"} | 426812829645406216 |
+--
    ↪ -----------+----------------------------+-----------+--------------------+-------------------------------
    ↪
```

该表中产生了两行数据，两行的 DEADLOCK_ID 字段皆为 1，表示这两行数据包含同一次死锁错误的信息。第一行显示 ID 为 426812829645406216 的事务，在 "74800000000000000355F728000000000000002" 这个 key 上，被 ID 为 "426812829645406217" 的事务阻塞了；第二行则显示 ID 为 "426812829645406217" 的事务在 "74800000000000000355F728000000000000001" 这个 key 上被 ID 为 426812829645406216 的事务阻塞了，构成了相互阻塞的状态，形成了死锁。

示例 2

假设查询 DEADLOCKS 表得到了如下结果集：

```
+--
    ↪ -----------+----------------------------+-----------+-------------------+-----------------------------
    ↪
| DEADLOCK_ID | OCCUR_TIME                 | RETRYABLE | TRY_LOCK_TRX_ID    | CURRENT_SQL_DIGEST
    ↪                                              | CURRENT_SQL_DIGEST_TEXT                 |
    ↪  KEY                                    | KEY_INFO
    ↪
    ↪ | TRX_HOLDING_LOCK   |
+--
    ↪ -----------+----------------------------+-----------+-------------------+-----------------------------
    ↪
|           1 | 2021-08-05 11:09:03.230341 |         0 | 426812829645406216 | 22230766411
    ↪ edb40f27a68dadefc63c6c6970d5827f1e5e22fc97be2c4d8350d | update `t` set `v` = ? where `id`
    ↪  = ? ; | 74800000000000000355F728000000000000002 | {"db_id":1,"db_name":"test","table_id"
    ↪ :53,"table_name":"t","handle_type":"int","handle_value":"2"} | 426812829645406217 |
|           1 | 2021-08-05 11:09:03.230341 |         0 | 426812829645406217 | 22230766411
    ↪ edb40f27a68dadefc63c6c6970d5827f1e5e22fc97be2c4d8350d | update `t` set `v` = ? where `id`
    ↪  = ? ; | 74800000000000000355F728000000000000001 | {"db_id":1,"db_name":"test","table_id"
    ↪ :53,"table_name":"t","handle_type":"int","handle_value":"1"} | 426812829645406216 |
|           2 | 2021-08-05 11:09:21.252154 |         0 | 426812832017809412 | 22230766411
    ↪ edb40f27a68dadefc63c6c6970d5827f1e5e22fc97be2c4d8350d | update `t` set `v` = ? where `id`
    ↪  = ? ; | 74800000000000000355F728000000000000002 | {"db_id":1,"db_name":"test","table_id"
    ↪ :53,"table_name":"t","handle_type":"int","handle_value":"2"} | 426812832017809413 |
|           2 | 2021-08-05 11:09:21.252154 |         0 | 426812832017809413 | 22230766411
    ↪ edb40f27a68dadefc63c6c6970d5827f1e5e22fc97be2c4d8350d | update `t` set `v` = ? where `id`
    ↪  = ? ; | 74800000000000000355F728000000000000003 | {"db_id":1,"db_name":"test","table_id"
    ↪ :53,"table_name":"t","handle_type":"int","handle_value":"3"} | 426812832017809414 |
|           2 | 2021-08-05 11:09:21.252154 |         0 | 426812832017809414 | 22230766411
    ↪ edb40f27a68dadefc63c6c6970d5827f1e5e22fc97be2c4d8350d | update `t` set `v` = ? where `id`
    ↪  = ? ; | 74800000000000000355F728000000000000001 | {"db_id":1,"db_name":"test","table_id"
    ↪ :53,"table_name":"t","handle_type":"int","handle_value":"1"} | 426812832017809412 |
+--
    ↪ -----------+----------------------------+-----------+-------------------+-----------------------------
    ↪
```

以上查询结果中的 DEADLOCK_ID 列表明，前两行共同表示一次死锁错误的信息，两条事务相互等待构成了死锁；而后三行共同表示另一次死锁信息，三个事务循环等待构成了死锁。

CLUSTER_DEADLOCKS

CLUSTER_DEADLOCKS 表返回整个集群上每个 TiDB 节点中最近发生的数次死锁错误的信息，即将每个节点上的 DEADLOCKS 表内的信息合并在一起。CLUSTER_DEADLOCKS 还包含额外的 INSTANCE 列展示所属节点的 IP 地址和端口，用以区分不同的 TiDB 节点。

需要注意的是，由于 DEADLOCK_ID 并不保证全局唯一，所以在 CLUSTER_DEADLOCKS 表的查询结果中，需要 INSTANCE 和 DEADLOCK_ID 两个字段共同区分结果集中的不同死锁错误的信息。

```
USE INFORMATION_SCHEMA;
DESC CLUSTER_DEADLOCKS;
```

```
+-------------------------+---------------------+------+------+---------+-------+
| Field                   | Type                | Null | Key  | Default | Extra |
+-------------------------+---------------------+------+------+---------+-------+
| INSTANCE                | varchar(64)         | YES  |      | NULL    |       |
| DEADLOCK_ID             | bigint(21)          | NO   |      | NULL    |       |
| OCCUR_TIME              | timestamp(6)        | YES  |      | NULL    |       |
| RETRYABLE               | tinyint(1)          | NO   |      | NULL    |       |
| TRY_LOCK_TRX_ID         | bigint(21) unsigned | NO   |      | NULL    |       |
| CURRENT_SQL_DIGEST      | varchar(64)         | YES  |      | NULL    |       |
| CURRENT_SQL_DIGEST_TEXT | text                | YES  |      | NULL    |       |
| KEY                     | text                | YES  |      | NULL    |       |
| KEY_INFO                | text                | YES  |      | NULL    |       |
| TRX_HOLDING_LOCK        | bigint(21) unsigned | NO   |      | NULL    |       |
+-------------------------+---------------------+------+------+---------+-------+
```

14.11.18.2.19 ENGINES

ENGINES 表提供了关于存储引擎的信息。从和 MySQL 兼容性上考虑，TiDB 会一直将 InnoDB 描述为唯一支持的引擎。此外，ENGINES 表中其它列值也都是定值。

```
USE information_schema;
DESC engines;
```

```
+--------------+-------------+------+------+---------+-------+
| Field        | Type        | Null | Key  | Default | Extra |
+--------------+-------------+------+------+---------+-------+
| ENGINE       | varchar(64) | YES  |      | NULL    |       |
| SUPPORT      | varchar(8)  | YES  |      | NULL    |       |
| COMMENT      | varchar(80) | YES  |      | NULL    |       |
| TRANSACTIONS | varchar(3)  | YES  |      | NULL    |       |
| XA           | varchar(3)  | YES  |      | NULL    |       |
| SAVEPOINTS   | varchar(3)  | YES  |      | NULL    |       |
+--------------+-------------+------+------+---------+-------+
6 rows in set (0.00 sec)
```

```
SELECT * FROM engines;
```

```
+--
    ↪ ------+---------+----------------------------------------------------------------+--------------+------+-----
    ↪
```

```
| ENGINE | SUPPORT | COMMENT                                                
```

字段解释：

- RULE：诊断规则名称，目前实现了以下规则：
  - config：配置一致性以及合理性检测。如果同一个配置在不同实例不一致，会生成 warning 诊断结果。
  - version：版本一致性检测。如果同一类型的实例版本不同，会生成 critical 诊断结果。
  - node-load：服务器负载检测。如果当前系统负载太高，会生成对应的 warning 诊断结果。
  - critical-error：系统各个模块定义了严重的错误，如果某一个严重错误在对应时间段内超过阈值，会生成 warning 诊断结果。
  - threshold-check：诊断系统会对一些关键指标进行阈值判断，如果超过阈值会生成对应的诊断信息。
- ITEM：每一个规则会对不同的项进行诊断，该字段表示对应规则下面的具体诊断项。
- TYPE：诊断的实例类型，可取值为 tidb，pd 和 tikv。
- INSTANCE：诊断的具体实例地址。
- STATUS_ADDRESS：实例的 HTTP API 服务地址。
- VALUE：针对这个诊断项得到的值。
- REFERENCE：针对这个诊断项的参考值（阈值）。如果 VALUE 超过阈值，就会产生对应的诊断信息。
- SEVERITY：严重程度，取值为 warning 或 critical。
- DETAILS：诊断的详细信息，可能包含进一步调查的 SQL 或文档链接。

诊断示例

对当前时间的集群进行诊断。

```
SELECT * FROM information_schema.inspection_result\G
```

```
***************************[ 1. row ]***************************
RULE      | config
ITEM      | log.slow-threshold
TYPE      | tidb
INSTANCE  | 172.16.5.40:4000
VALUE     | 0
REFERENCE | not 0
SEVERITY  | warning
DETAILS   | slow-threshold = 0 will record every query to slow log, it may affect performance
***************************[ 2. row ]***************************
RULE      | version
ITEM      | git_hash
TYPE      | tidb
INSTANCE  |
VALUE     | inconsistent
REFERENCE | consistent
SEVERITY  | critical
DETAILS   | the cluster has 2 different tidb version, execute the sql to see more detail: select
    ↪ * from information_schema.cluster_info where type='tidb'
```

```
***************************[ 3. row ]***************************
RULE      | threshold-check
ITEM      | storage-write-duration
TYPE      | tikv
INSTANCE  | 172.16.5.40:23151
VALUE     | 130.417
REFERENCE | < 0.100
SEVERITY  | warning
DETAILS   | max duration of 172.16.5.40:23151 tikv storage-write-duration was too slow
***************************[ 4. row ]***************************
RULE      | threshold-check
ITEM      | rocksdb-write-duration
TYPE      | tikv
INSTANCE  | 172.16.5.40:20151
VALUE     | 108.105
REFERENCE | < 0.100
SEVERITY  | warning
DETAILS   | max duration of 172.16.5.40:20151 tikv rocksdb-write-duration was too slow
```

上述诊断结果发现了以下几个问题：

- 第一行表示 TiDB 的 `log.slow-threshold` 配置值为 0，可能会影响性能。
- 第二行表示集群中有 2 个不同的 TiDB 版本
- 第三、四行表示 TiKV 的写入延迟太大，期望时间是不超过 0.1s，但实际值远超预期。

诊断集群在时间段“2020-03-26 00:03:00”，“2020-03-26 00:08:00”的问题。指定时间范围需要使用 `/*+ time_range ()*/` 的 SQL Hint，参考下面的查询示例：

```
select /*+ time_range("2020-03-26 00:03:00", "2020-03-26 00:08:00") */ * from information_schema.inspection_result\G
```

```
***************************[ 1. row ]***************************
RULE      | critical-error
ITEM      | server-down
TYPE      | tidb
INSTANCE  | 172.16.5.40:4009
VALUE     |
REFERENCE |
SEVERITY  | critical
DETAILS   | tidb 172.16.5.40:4009 restarted at time '2020/03/26 00:05:45.670'
***************************[ 2. row ]***************************
RULE      | threshold-check
ITEM      | get-token-duration
TYPE      | tidb
INSTANCE  | 172.16.5.40:10089
VALUE     | 0.234
```

```
REFERENCE | < 0.001
SEVERITY  | warning
DETAILS   | max duration of 172.16.5.40:10089 tidb get-token-duration is too slow
```

上面的诊断结果发现了以下问题：

- 第一行表示 172.16.5.40:4009 TiDB 实例在 2020/03/26 00:05:45.670 发生了重启。
- 第二行表示 172.16.5.40:10089 TiDB 实例的最大的 get-token-duration 时间为 0.234s，期望时间是小于 0.001s。

也可以指定条件，比如只查询 critical 严重级别的诊断结果：

```
select * from information_schema.inspection_result where severity='critical';
```

只查询 critical-error 规则的诊断结果:

```
select * from information_schema.inspection_result where rule='critical-error';
```

诊断规则介绍

诊断模块内部包含一系列的规则，这些规则会通过查询已有的监控表和集群信息表，对结果和阈值进行对比。如果结果超过阈值将生成 warning 或 critical 的结果，并在 details 列中提供相应信息。

可以通过查询 inspection_rules 系统表查询已有的诊断规则:

```
select * from information_schema.inspection_rules where type='inspection';
```

```
+-----------------+------------+---------+
| NAME            | TYPE       | COMMENT |
+-----------------+------------+---------+
| config          | inspection |         |
| version         | inspection |         |
| node-load       | inspection |         |
| critical-error  | inspection |         |
| threshold-check | inspection |         |
+-----------------+------------+---------+
```

config 诊断规则

config 诊断规则通过查询 CLUSTER_CONFIG 系统表，执行以下两个诊断规则：

- 检测相同组件的配置值是否一致，并非所有配置项都会有一致性检查，下面是一致性检查的白名单：

```
// TiDB 配置一致性检查白名单
port
status.status-port
host
path
advertise-address
```

```
status.status-port
log.file.filename
log.slow-query-file
tmp-storage-path

// PD 配置一致性检查白名单
advertise-client-urls
advertise-peer-urls
client-urls
data-dir
log-file
log.file.filename
metric.job
name
peer-urls

// TiKV 配置一致性检查白名单
server.addr
server.advertise-addr
server.status-addr
log-file
raftstore.raftdb-path
storage.data-dir
storage.block-cache.capacity
```

- 检测以下配置项的值是否符合预期。

组件	配置项	预期值
TiDB	log.slow-threshold	大于 0
TiKV	raftstore.sync-log	true

version 诊断规则

version 诊断规则通过查询 CLUSTER_INFO 系统表，检测相同组件的版本 hash 是否一致。示例如下：

```
SELECT * FROM information_schema.inspection_result WHERE rule='version'\G
```

```
***************************[ 1. row ]***************************
RULE      | version
ITEM      | git_hash
TYPE      | tidb
INSTANCE  |
VALUE     | inconsistent
REFERENCE | consistent
SEVERITY  | critical
```

```
DETAILS   | the cluster has 2 different tidb versions, execute the sql to see more detail: SELECT
    ↪  * FROM information_schema.cluster_info WHERE type='tidb'
```

`critical-error` 诊断规则

`critical-error` 诊断规则执行以下两个诊断规则：

- 通过查询metrics schema 数据库中相关的监控系统表，检测集群是否有出现以下比较严重的错误：

组件	错误名字	相关监控表	错误说明
TiDB	panic-count	tidb_panic_total_count	TiDB 出现 panic 错误
TiDB	binlog-error	tidb_binlog_error_total_count	TiDB 写 binlog 时出现的错误
TiKV	critical-error	tikv_critical_error_total_coun	TiKV 的 critical error
TiKV	scheduler-is-busy	tikv_scheduler_is_busy_total_count	TiKV 的 scheduler 太忙，会导致 TiKV 临时不可用
TiKV	coprocessor-is-busy	tikv_coprocessor_is_busy_total_count	TiKV 的 coprocessor 太忙
TiKV	channel-is-full	tikv_channel_full_total_count	TiKV 出现 channel full 的错误

组件	错误名字	相关监控表	错误说明
TiKV	tikv_engine_write_stall	tikv_engine_write_stall	TiKV 出现写入 stall 的错误

- 通过查询 `metrics_schema.up` 监控表和 `CLUSTER_LOG` 系统表，检查是否有组件发生重启。

`threshold-check` 诊断规则

`threshold-check` 诊断规则通过查询metrics schema 数据库中相关的监控系统表，检测集群中以下指标是否超出阈值：

组件	监控指标	相关监控表	预期值	说明
TiDB	tso-duration	pd_tso_wait_duration	小于 50 ms	获取事务 TSO 时间戳的等待耗时
TiDB	get-token-duration	tidb_get_token_duration	小于 1 ms	查询获取 token 的耗时，相关的 TiDB 配置参数是 token-limit
TiDB	load-schema-duration	tidb_load_schema_duration	小于 1 s	TiDB 更新表元信息的耗时

组件	监控指标	相关监控表	预期值	说明
TiKV	scheduler-cmd-duration	tikv_scheduler_command_duration	小于 0.1 s	TiKV 执行 KV cmd 请求的耗时
TiKV	handle-snapshot-duration	tikv_handle_snapshot_duration	小于 30 s	TiKV 处理 snap-shot 的耗时
TiKV	storage-write-duration	tikv_storage_async_request_duration	小于 0.1 s	TiKV 写入的延迟
TiKV	storage-snapshot-duration	tikv_storage_async_request_duration	小于 50 ms	TiKV 获取 snap-shot 的耗时
TiKV	rocksdb-write-duration	tikv_engine_write_duration	小于 100 ms	TiKV RocksDB 的写入延迟
TiKV	rocksdb-get-duration	tikv_engine_max_get_duration	小于 50 ms	TiKV RocksDB 的读取延迟
TiKV	rocksdb-seek-duration	tikv_engine_max_seek_duration	小于 50 ms	TiKV RocksDB 执行 seek 的延迟
TiKV	scheduler-pending-cmd-coun	tikv_scheduler_pending_commands	小于 1000	TiKV 中被阻塞的命令数量

组件	监控指标	相关监控表	预期值	说明
TiKV	index-block-cache-hit	tikv_block_index_cache_hit	大于 0.95	TiKV 中 index block 缓存的命中率
TiKV	filter-block-cache-hit	tikv_block_filter_cache_hit	大于 0.95	TiKV 中 filter block 缓存的命中率
TiKV	data-block-cache-hit	tikv_block_data_cache_hit	大于 0.80	TiKV 中 data block 缓存的命中率
TiKV	leader-score-balance	pd_scheduler_store_status	小于 0.05	检测各个 TiKV 实例的 leader score 是否均衡，期望实例间的差异小于 5%

组件	监控指标	相关监控表	预期值	说明
TiKV	region-score-balance	pd_scheduler_store_status	小于 0.05	检测各个 TiKV 实例的 Region score 是否均衡，期望实例间的差异小于 5%
TiKV	store-available-balance	pd_scheduler_store_status	小于 0.2	检测各个 TiKV 实例的存储可用空间大小是否均衡，期望实例间的差异小于 20%

组件	监控指标	相关监控表	预期值	说明
TiKV	region-count	pd_scheduler_store_status	小于 20000	检测各个 TiKV 实例的 Region 数量，期望单个实例的 Region 数量小于 20000
PD	region-health	pd_region_health	小于 100	检测集群中处于调度中间状态的 Region 数量，期望总数小于 100

另外还会检测 TiKV 实例的以下 thread cpu usage 是否过高:

- scheduler-worker-cpu
- coprocessor-normal-cpu
- coprocessor-high-cpu
- coprocessor-low-cpu
- grpc-cpu
- raftstore-cpu
- apply-cpu
- storage-readpool-normal-cpu
- storage-readpool-high-cpu
- storage-readpool-low-cpu
- split-check-cpu

TiDB 内置的诊断规则还在不断的完善改进中，如果你也想到了一些诊断规则，非常欢迎在 tidb repository 下提 PR 或 Issue。

14.11.18.2.21 INSPECTION_RULES

INSPECTION_RULES 表提供在检查结果中运行哪些诊断测试的信息，示例用法参见inspection-result 表。

```
USE information_schema;
DESC inspection_rules;
```

```
+---------+--------------+------+------+---------+-------+
| Field   | Type         | Null | Key  | Default | Extra |
+---------+--------------+------+------+---------+-------+
| NAME    | varchar(64)  | YES  |      | NULL    |       |
| TYPE    | varchar(64)  | YES  |      | NULL    |       |
| COMMENT | varchar(256) | YES  |      | NULL    |       |
+---------+--------------+------+------+---------+-------+
3 rows in set (0.00 sec)
```

```
SELECT * FROM inspection_rules;
```

```
+-----------------+------------+---------+
| NAME            | TYPE       | COMMENT |
+-----------------+------------+---------+
| config          | inspection |         |
| version         | inspection |         |
| node-load       | inspection |         |
| critical-error  | inspection |         |
| threshold-check | inspection |         |
| ddl             | summary    |         |
| gc              | summary    |         |
| pd              | summary    |         |
| query-summary   | summary    |         |
| raftstore       | summary    |         |
| read-link       | summary    |         |
| rocksdb         | summary    |         |
| stats           | summary    |         |
| wait-events     | summary    |         |
| write-link      | summary    |         |
+-----------------+------------+---------+
15 rows in set (0.00 sec)
```

14.11.18.2.22 INSPECTION_SUMMARY

在部分场景下，用户只需要关注特定链路或模块的监控汇总。例如当前 Coprocessor 配置的线程池为 8，如果 Coprocessor 的 CPU 使用率达到了 750%，就可以确定存在风险，或者可能提前成为瓶颈。但是部分监控会因为

用户的 workload 不同而差异较大，所以难以定义确定的阈值。排查这部分场景的问题也非常重要，所以 TiDB 提供了 inspection_summary 来进行链路汇总。

诊断汇总表 information_schema.inspection_summary 的表结构如下：

```
USE information_schema;
DESC inspection_summary;
```

```
+--------------+--------------+------+------+---------+-------+
| Field        | Type         | Null | Key  | Default | Extra |
+--------------+--------------+------+------+---------+-------+
| RULE         | varchar(64)  | YES  |      | NULL    |       |
| INSTANCE     | varchar(64)  | YES  |      | NULL    |       |
| METRICS_NAME | varchar(64)  | YES  |      | NULL    |       |
| LABEL        | varchar(64)  | YES  |      | NULL    |       |
| QUANTILE     | double       | YES  |      | NULL    |       |
| AVG_VALUE    | double(22,6) | YES  |      | NULL    |       |
| MIN_VALUE    | double(22,6) | YES  |      | NULL    |       |
| MAX_VALUE    | double(22,6) | YES  |      | NULL    |       |
| COMMENT      | varchar(256) | YES  |      | NULL    |       |
+--------------+--------------+------+------+---------+-------+
9 rows in set (0.00 sec)
```

字段解释：

- RULE：汇总规则。由于规则在持续添加，最新的规则列表可以通过 select * from inspection_rules where type='summary' 查询。
- INSTANCE：监控的具体实例。
- METRICS_NAME：监控表的名字。
- QUANTILE：对于包含 QUANTILE 的监控表有效，可以通过谓词下推指定多个百分位，例如 select * from inspection_summary where rule='ddl' and quantile in (0.80, 0.90, 0.99, 0.999) 来汇总 DDL 相关监控，查询百分位为 80/90/99/999 的结果。AVG_VALUE、MIN_VALUE、MAX_VALUE 分别表示聚合的平均值、最小值、最大值。
- COMMENT：对应监控的解释。

> 注意：
>
> 由于汇总所有结果有一定开销，建议在 SQL 的谓词中显示指定的 rule 以减小开销。例如 select * from inspection_summary where rule in ('read-link', 'ddl') 会汇总读链路和 DDL 相关的监控。

使用示例:

诊断结果表和诊断监控汇总表都可以通过 hint 的方式指定诊断的时间范围，例如 select /*+ time_range ('2020-03-07 12:00:00','2020-03-07 13:00:00')*/* from inspection_summary 是对 2020-03-07 12:00:00 -

2020-03-07 13:00:00 时间段的监控汇总。和监控汇总表一样，inspection_summary 系统表也可以通过对比两个不同时间段的数据，快速发现差异较大的监控项。

以下为一个例子，对比以下两个时间段，读系统链路的监控项:

- (2020-01-16 16:00:54.933, 2020-01-16 16:10:54.933)
- (2020-01-16 16:10:54.933, 2020-01-16 16:20:54.933)

```
SELECT
  t1.avg_value / t2.avg_value AS ratio,
  t1.*,
  t2.*
FROM
  (
    SELECT
      /*+ time_range("2020-01-16 16:00:54.933", "2020-01-16 16:10:54.933")*/ *
    FROM information_schema.inspection_summary WHERE rule='read-link'
  ) t1
  JOIN
  (
    SELECT
      /*+ time_range("2020-01-16 16:10:54.933", "2020-01-16 16:20:54.933")*/ *
    FROM information_schema.inspection_summary WHERE rule='read-link'
  ) t2
  ON t1.metrics_name = t2.metrics_name
  and t1.instance = t2.instance
  and t1.label = t2.label
ORDER BY
  ratio DESC;
```

14.11.18.2.23 KEY_COLUMN_USAGE

KEY_COLUMN_USAGE 表描述了列的键约束，比如主键约束。

```
USE information_schema;
DESC key_column_usage;
```

```
+------------------------------+--------------+------+------+---------+-------+
| Field                        | Type         | Null | Key  | Default | Extra |
+------------------------------+--------------+------+------+---------+-------+
| CONSTRAINT_CATALOG           | varchar(512) | NO   |      | NULL    |       |
| CONSTRAINT_SCHEMA            | varchar(64)  | NO   |      | NULL    |       |
| CONSTRAINT_NAME              | varchar(64)  | NO   |      | NULL    |       |
| TABLE_CATALOG                | varchar(512) | NO   |      | NULL    |       |
| TABLE_SCHEMA                 | varchar(64)  | NO   |      | NULL    |       |
| TABLE_NAME                   | varchar(64)  | NO   |      | NULL    |       |
```

```
| COLUMN_NAME                    | varchar(64)  | NO   |      | NULL    |       |
| ORDINAL_POSITION               | bigint(10)   | NO   |      | NULL    |       |
| POSITION_IN_UNIQUE_CONSTRAINT | bigint(10)   | YES  |      | NULL    |       |
| REFERENCED_TABLE_SCHEMA        | varchar(64)  | YES  |      | NULL    |       |
| REFERENCED_TABLE_NAME          | varchar(64)  | YES  |      | NULL    |       |
| REFERENCED_COLUMN_NAME         | varchar(64)  | YES  |      | NULL    |       |
+--------------------------------+--------------+------+------+---------+-------+
12 rows in set (0.00 sec)
```

```
SELECT * FROM key_column_usage WHERE table_schema='mysql' and table_name='user';
```

```
*************************** 1. row ***************************
           CONSTRAINT_CATALOG: def
            CONSTRAINT_SCHEMA: mysql
              CONSTRAINT_NAME: PRIMARY
                TABLE_CATALOG: def
                 TABLE_SCHEMA: mysql
                   TABLE_NAME: user
                  COLUMN_NAME: Host
             ORDINAL_POSITION: 1
POSITION_IN_UNIQUE_CONSTRAINT: NULL
      REFERENCED_TABLE_SCHEMA: NULL
        REFERENCED_TABLE_NAME: NULL
       REFERENCED_COLUMN_NAME: NULL
*************************** 2. row ***************************
           CONSTRAINT_CATALOG: def
            CONSTRAINT_SCHEMA: mysql
              CONSTRAINT_NAME: PRIMARY
                TABLE_CATALOG: def
                 TABLE_SCHEMA: mysql
                   TABLE_NAME: user
                  COLUMN_NAME: User
             ORDINAL_POSITION: 2
POSITION_IN_UNIQUE_CONSTRAINT: NULL
      REFERENCED_TABLE_SCHEMA: NULL
        REFERENCED_TABLE_NAME: NULL
       REFERENCED_COLUMN_NAME: NULL
2 rows in set (0.00 sec)
```

KEY_COLUMN_USAGE 表中列的含义如下：

- CONSTRAINT_CATALOG：约束所属的目录的名称。该值始终为 def。
- CONSTRAINT_SCHEMA：约束所属的数据库的名称。
- CONSTRAINT_NAME：约束名称。
- TABLE_CATALOG：表所属目录的名称。该值始终为 def。

- TABLE_SCHEMA：表所属的架构数据库的名称。
- TABLE_NAME：具有约束的表的名称。
- COLUMN_NAME：具有约束的列的名称。
- ORDINAL_POSITION：列在约束中的位置，而不是列在表中的位置。列位置从 1 开始编号。
- POSITION_IN_UNIQUE_CONSTRAINT：唯一约束和主键约束为空。对于外键约束，此列是被引用的表的键的序号位置。
- REFERENCED_TABLE_SCHEMA：约束引用的数据库的名称。目前在 TiDB 中，除了外键约束，其它约束此列的值都为 nil。
- REFERENCED_TABLE_NAME：约束引用的表的名称。目前在 TiDB 中，除了外键约束，其它约束此列的值都为 nil。
- REFERENCED_COLUMN_NAME：约束引用的列的名称。目前在 TiDB 中，除了外键约束，其它约束此列的值都为 nil。

14.11.18.2.24 MEMORY_USAGE

MEMORY_USAGE 表描述了 TiDB 实例当前的内存使用情况。

```
USE information_schema;
DESC memory_usage;
```

```
+--------------------+-------------+------+------+---------+-------+
| Field              | Type        | Null | Key  | Default | Extra |
+--------------------+-------------+------+------+---------+-------+
| MEMORY_TOTAL       | bigint(21)  | NO   |      | NULL    |       |
| MEMORY_LIMIT       | bigint(21)  | NO   |      | NULL    |       |
| MEMORY_CURRENT     | bigint(21)  | NO   |      | NULL    |       |
| MEMORY_MAX_USED    | bigint(21)  | NO   |      | NULL    |       |
| CURRENT_OPS        | varchar(50) | YES  |      | NULL    |       |
| SESSION_KILL_LAST  | datetime    | YES  |      | NULL    |       |
| SESSION_KILL_TOTAL | bigint(21)  | NO   |      | NULL    |       |
| GC_LAST            | datetime    | YES  |      | NULL    |       |
| GC_TOTAL           | bigint(21)  | NO   |      | NULL    |       |
| DISK_USAGE         | bigint(21)  | NO   |      | NULL    |       |
| QUERY_FORCE_DISK   | bigint(21)  | NO   |      | NULL    |       |
+--------------------+-------------+------+------+---------+-------+
11 rows in set (0.000 sec)
```

```
SELECT * FROM information_schema.memory_usage;
```

```
+--
    ↪ ------------+--------------+----------------+-----------------+-------------+-------------------+------
    ↪
| MEMORY_TOTAL | MEMORY_LIMIT | MEMORY_CURRENT | MEMORY_MAX_USED | CURRENT_OPS |
    ↪ SESSION_KILL_LAST    | SESSION_KILL_TOTAL | GC_LAST             | GC_TOTAL | DISK_USAGE |
    ↪ QUERY_FORCE_DISK |
```

```
+--
    ↪ ------------+--------------+----------------+-----------------+-------------+---------------------+------
    ↪
|  33674170368 |  10737418240 |     5097644032 |     10826604544 | NULL        | 2022-10-17
    ↪ 22:47:47 |                   1 | 2022-10-17 22:47:47 |       20 |           0 |
    ↪                0 |
+--
    ↪ ------------+--------------+----------------+-----------------+-------------+---------------------+------
    ↪
2 rows in set (0.002 sec)
```

MEMORY_USAGE 表中列的含义如下：

- MEMORY_TOTAL：TiDB 的可用内存总量，单位为 byte。
- MEMORY_LIMIT：TiDB 的内存使用限制，单位为 byte。其值与系统变量tidb_server_memory_limit 的值相同。
- MEMORY_CURRENT：TiDB 当前的内存使用量，单位为 byte。
- MEMORY_MAX_USED：从 TiDB 启动到当前的最大内存使用量，单位为 byte。
- CURRENT_OPS：“shrinking” | null。“shrinking” 表示 TiDB 正在执行收缩内存用量的操作。
- SESSION_KILL_LAST：上一次终止会话的时间戳。
- SESSION_KILL_TOTAL：从 TiDB 启动到当前累计终止会话的次数。
- GC_LAST：上一次由内存使用触发 Golang GC 的时间戳。
- GC_TOTAL：从 TiDB 启动到当前累计由内存使用触发 Golang GC 的次数。
- DISK_USAGE：当前数据落盘的硬盘使用量，单位为 byte。
- QUERY_FORCE_DISK：从 TiDB 启动到当前累计的落盘次数。

14.11.18.2.25 MEMORY_USAGE_OPS_HISTORY

MEMORY_USAGE_OPS_HISTORY 表描述了 TiDB 实例内存相关的历史操作和执行依据。

```
USE information_schema;
DESC memory_usage_ops_history;
```

```
+----------------+---------------------+------+------+---------+-------+
| Field          | Type                | Null | Key  | Default | Extra |
+----------------+---------------------+------+------+---------+-------+
| TIME           | datetime            | NO   |      | NULL    |       |
| OPS            | varchar(20)         | NO   |      | NULL    |       |
| MEMORY_LIMIT   | bigint(21)          | NO   |      | NULL    |       |
| MEMORY_CURRENT | bigint(21)          | NO   |      | NULL    |       |
| PROCESSID      | bigint(21) unsigned | YES  |      | NULL    |       |
| MEM            | bigint(21) unsigned | YES  |      | NULL    |       |
| DISK           | bigint(21) unsigned | YES  |      | NULL    |       |
| CLIENT         | varchar(64)         | YES  |      | NULL    |       |
| DB             | varchar(64)         | YES  |      | NULL    |       |
| USER           | varchar(16)         | YES  |      | NULL    |       |
```

```
| SQL_DIGEST      | varchar(64)         | YES  |      | NULL    |       |
| SQL_TEXT        | varchar(256)        | YES  |      | NULL    |       |
+-----------------+---------------------+------+------+---------+-------+
12 rows in set (0.000 sec)
```

```
SELECT * FROM information_schema.memory_usage_ops_history;
```

```
+--
    ↪ -------------------+-------------+--------------+----------------+---------------------+------------+----
    ↪
| TIME                | OPS          | MEMORY_LIMIT | MEMORY_CURRENT | PROCESSID           | MEM
    ↪        | DISK | CLIENT           | DB    | USER | SQL_DIGEST
    ↪                                                          | SQL_TEXT
    ↪                                                                |
+--
    ↪ -------------------+-------------+--------------+----------------+---------------------+------------+----
    ↪
| 2022-10-17 22:46:25 | SessionKill |  10737418240 |    10880237568 | 6718275530455515543 |
    ↪ 7905028235 |    0 | 127.0.0.1:34394 | test | root | 146
    ↪ b3d812852663a20635fbcf02be01688f52c8d433dafec0d496a14f0b59df6 | desc analyze select *
    ↪ from t t1 join t t2 on t1.a=t2.a order by t1.a |
+--
    ↪ -------------------+-------------+--------------+----------------+---------------------+------------+----
    ↪
2 rows in set (0.002 sec)
```

MEMORY_USAGE_OPS_HISTORY 表中列的含义如下：

- TIME：终止会话的时间戳。
- OPS：“SessionKill”
- MEMORY_LIMIT：TiDB 终止会话时的内存使用限制，单位为 byte。其值和系统变量tidb_server_memory_limit相同。
- MEMORY_CURRENT：TiDB 当前的内存使用量，单位为 byte。
- PROCESSID：被终止会话的客户连接 ID。
- MEM：被终止的会话已使用的内存使用量，单位是 byte。
- DISK：被终止的会话已使用的硬盘使用量，单位是 byte。
- CLIENT：被终止的会话的客户连接的地址。
- DB：被终止的会话所连接的数据库名。
- USER：被终止的会话的用户名。
- SQL_DIGEST：被终止的会话正在执行 SQL 语句的 digest。
- SQL_TEXT: 被终止的会话正在执行的 SQL 语句。

14.11.18.2.26 METRICS_SUMMARY

由于 TiDB 集群的监控指标数量较多，为了方便用户从众多监控中找出异常的监控项，TiDB 4.0 提供了以下监控汇总表：

- information_schema.metrics_summary
- information_schema.metrics_summary_by_label

这两张表用于汇总所有监控数据，用户排查各个监控指标会更有效率。其中 information_schema.metrics_summary_by_label 会对不同的 label 进行区分统计。

```
USE information_schema;
DESC metrics_summary;
```

```
+--------------+--------------+------+------+---------+-------+
| Field        | Type         | Null | Key  | Default | Extra |
+--------------+--------------+------+------+---------+-------+
| METRICS_NAME | varchar(64)  | YES  |      | NULL    |       |
| QUANTILE     | double       | YES  |      | NULL    |       |
| SUM_VALUE    | double(22,6) | YES  |      | NULL    |       |
| AVG_VALUE    | double(22,6) | YES  |      | NULL    |       |
| MIN_VALUE    | double(22,6) | YES  |      | NULL    |       |
| MAX_VALUE    | double(22,6) | YES  |      | NULL    |       |
| COMMENT      | varchar(256) | YES  |      | NULL    |       |
+--------------+--------------+------+------+---------+-------+
7 rows in set (0.00 sec)
```

字段解释：

- METRICS_NAME：监控表名。
- QUANTILE：百分位。可以通过 SQL 语句指定 QUANTILE，例如：
  - select * from metrics_summary where quantile=0.99 指定查看百分位为 0.99 的数据。
  - select * from metrics_summary where quantile in (0.80, 0.90, 0.99, 0.999) 同时查看百分位为 0.80, 0.90, 0.99, 0.999 的数据。
- SUM_VALUE、AVG_VALUE、MIN_VALUE、MAX_VALUE 分别表示总和、平均值、最小值、最大值。
- COMMENT：对应监控的解释。

具体查询示例：

查询 '2020-03-08 13:23:00', '2020-03-08 13:33:00' 时间范围内 TiDB 集群中平均耗时最高的三组监控项。可直接查询 information_schema.metrics_summary 表，并通过 /*+ time_range()*/ 这个 hint 来指定时间范围，构造的 SQL 语句如下：

```
SELECT /*+ time_range('2020-03-08 13:23:00','2020-03-08 13:33:00') */ *
FROM information_schema.metrics_summary
WHERE metrics_name LIKE 'tidb%duration'
 AND avg_value > 0
 AND quantile = 0.99
ORDER BY avg_value DESC
LIMIT 3\G
```

```
***************************[ 1. row ]***************************
METRICS_NAME | tidb_get_token_duration
QUANTILE     | 0.99
SUM_VALUE    | 8.972509
AVG_VALUE    | 0.996945
MIN_VALUE    | 0.996515
MAX_VALUE    | 0.997458
COMMENT      |  The quantile of Duration (us) for getting token, it should be small until
    ↪ concurrency limit is reached(second)
***************************[ 2. row ]***************************
METRICS_NAME | tidb_query_duration
QUANTILE     | 0.99
SUM_VALUE    | 0.269079
AVG_VALUE    | 0.007272
MIN_VALUE    | 0.000667
MAX_VALUE    | 0.01554
COMMENT      | The quantile of TiDB query durations(second)
***************************[ 3. row ]***************************
METRICS_NAME | tidb_kv_request_duration
QUANTILE     | 0.99
SUM_VALUE    | 0.170232
AVG_VALUE    | 0.004601
MIN_VALUE    | 0.000975
MAX_VALUE    | 0.013
COMMENT      | The quantile of kv requests durations by store
```

类似的，查询 `metrics_summary_by_label` 监控汇总表示例如下：

```
SELECT /*+ time_range('2020-03-08 13:23:00','2020-03-08 13:33:00') */ *
FROM information_schema.metrics_summary_by_label
WHERE metrics_name LIKE 'tidb%duration'
 AND avg_value > 0
 AND quantile = 0.99
ORDER BY avg_value DESC
LIMIT 10\G
```

```
***************************[ 1. row ]***************************
INSTANCE     | 172.16.5.40:10089
METRICS_NAME | tidb_get_token_duration
LABEL        |
QUANTILE     | 0.99
SUM_VALUE    | 8.972509
AVG_VALUE    | 0.996945
MIN_VALUE    | 0.996515
MAX_VALUE    | 0.997458
```

```
COMMENT      |  The quantile of Duration (us) for getting token, it should be small until
    ↪ concurrency limit is reached(second)
***************************[ 2. row ]***************************
INSTANCE     | 172.16.5.40:10089
METRICS_NAME | tidb_query_duration
LABEL        | Select
QUANTILE     | 0.99
SUM_VALUE    | 0.072083
AVG_VALUE    | 0.008009
MIN_VALUE    | 0.007905
MAX_VALUE    | 0.008241
COMMENT      | The quantile of TiDB query durations(second)
***************************[ 3. row ]***************************
INSTANCE     | 172.16.5.40:10089
METRICS_NAME | tidb_query_duration
LABEL        | Rollback
QUANTILE     | 0.99
SUM_VALUE    | 0.072083
AVG_VALUE    | 0.008009
MIN_VALUE    | 0.007905
MAX_VALUE    | 0.008241
COMMENT      | The quantile of TiDB query durations(second)
```

前文提到 `metrics_summary_by_label` 表结构相对于 `metrics_summary` 多了一列 `LABEL`。以上面查询结果的第 2、3 行分别表示 `tidb_query_duration` 的 `Select` 和 `Rollback` 类型的语句平均耗时非常高。

除以上示例之外，监控汇总表可以通过对比两个时间段的全链路监控，迅速找出监控数据中变化最大的模块，快速定位瓶颈。以下示例对比两个时间段的所有监控（其中 t1 为 baseline），并按照差别最大的监控排序：

- 时间段 t1：`("2020-03-03 17:08:00", "2020-03-03 17:11:00")`
- 时间段 t2：`("2020-03-03 17:18:00", "2020-03-03 17:21:00")`

对两个时间段的监控按照 `METRICS_NAME` 进行 join，并按照差异值大小排序。其中 `TIME_RANGE` 是用于指定查询时间的 hint。

```
SELECT GREATEST(t1.avg_value,t2.avg_value)/LEAST(t1.avg_value,
         t2.avg_value) AS ratio,
         t1.metrics_name,
         t1.avg_value as t1_avg_value,
         t2.avg_value as t2_avg_value,
         t2.comment
FROM
    (SELECT /*+ time_range("2020-03-03 17:08:00", "2020-03-03 17:11:00")*/ *
    FROM information_schema.metrics_summary ) t1
JOIN
    (SELECT /*+ time_range("2020-03-03 17:18:00", "2020-03-03 17:21:00")*/ *
```

```
    FROM information_schema.metrics_summary ) t2
    ON t1.metrics_name = t2.metrics_name
ORDER BY ratio DESC LIMIT 10;
```

```
+--
    ↪ --------------+----------------------------------------+---------------+-----------------+----------
    ↪
| ratio          | metrics_name                             | t1_avg_value   | t2_avg_value     |
    ↪  comment
    ↪                                                                                          |
+--
    ↪ --------------+----------------------------------------+---------------+-----------------+----------
    ↪
| 5865.59537065  | tidb_slow_query_cop_process_total_time   |       0.016333 |        95.804724 |
    ↪  The total time of TiDB slow query statistics with slow query total cop process time(
    ↪ second) |
| 3648.74109023  | tidb_distsql_partial_scan_key_total_num  |   10865.666667 |  39646004.4394   |
    ↪  The total num of distsql partial scan key numbers
    ↪                                          |
|  267.002351165 | tidb_slow_query_cop_wait_total_time      |       0.003333 |         0.890008 |
    ↪  The total time of TiDB slow query statistics with slow query total cop wait time(second)
    ↪     |
|  192.43267836  | tikv_cop_total_response_total_size       | 2515333.66667  | 484032394.445    |
    ↪
    ↪
    ↪ |
|  192.43267836  | tikv_cop_total_response_size_per_seconds |   41922.227778 |   8067206.57408  |
    ↪
    ↪
    ↪ |
|  152.780296296 | tidb_distsql_scan_key_total_num          |    5304.333333 |    810397.618317 |
    ↪  The total num of distsql scan numbers
    ↪                                                    |
|  126.042290167 | tidb_distsql_execution_total_time        |       0.421622 |        53.142143 |
    ↪  The total time of distsql execution(second)
    ↪                                                  |
|  105.164020657 | tikv_cop_scan_details                    |     134.450733 |     14139.379665 |
    ↪
    ↪
    ↪ |
|  105.164020657 | tikv_cop_scan_details_total              |    8067.043981 |    848362.77991  |
    ↪
    ↪
    ↪ |
|  101.635495394 | tikv_cop_scan_keys_num                   |    1070.875    |    108838.91113  |
```

```
    ↪
    ↪
    ↪ |
+--
    ↪ --------------+----------------------------------------+---------------+-----------------+-----------
    ↪
```

上面查询结果表示：

- t2 时间段内的 tidb_slow_query_cop_process_total_time（TiDB 慢查询中的 cop process 耗时）比 t1 时间段高了 5865 倍。
- t2 时间段内的 tidb_distsql_partial_scan_key_total_num（TiDB 的 distsql 请求扫描 key 的数量）比 t1 时间段高了 3648 倍。
- t2 时间段内的 tidb_slow_query_cop_wait_total_time（TiDB 慢查询中的 cop 请求排队等待的耗时）比 t1 时间段高了 267 倍。
- t2 时间段内的 tikv_cop_total_response_size（TiKV 的 cop 请求结果的大小）比 t1 时间段高了 192 倍。
- t2 时间段内的 tikv_cop_scan_details（TiKV 的 cop 请求的 scan）比 t1 时间段高了 105 倍。

综上，可以马上知道 t2 时间段的 cop 请求要比 t2 时间段高很多，导致 TiKV 的 Coprocessor 过载，出现了 cop task 等待，可以猜测可能是 t2 时间段出现了一些大查询，或者是查询较多的负载。

实际上，在 t1 ~ t2 整个时间段内都在跑 go-ycsb 的压测，然后在 t2 时间段跑了 20 个 tpch 的查询，所以是因为 tpch 大查询导致了出现很多的 cop 请求。

14.11.18.2.27 METRICS_TABLES

METRICS_TABLES 表为METRICS_SCHEMA 数据库中的每个视图提供 PromQL（Prometheus 查询语言）定义。

```
USE INFORMATION_SCHEMA;
DESC METRICS_TABLES;
```

输出结果如下：

```
+------------+--------------+------+------+---------+-------+
| Field      | Type         | Null | Key  | Default | Extra |
+------------+--------------+------+------+---------+-------+
| TABLE_NAME | varchar(64)  | YES  |      | NULL    |       |
| PROMQL     | varchar(64)  | YES  |      | NULL    |       |
| LABELS     | varchar(64)  | YES  |      | NULL    |       |
| QUANTILE   | double       | YES  |      | NULL    |       |
| COMMENT    | varchar(256) | YES  |      | NULL    |       |
+------------+--------------+------+------+---------+-------+
```

表 METRICS_TABLES 的字段解释：

- TABLE_NAME：对应于 METRICS_SCHEMA 中的表名。

- PROMQL：监控表的主要原理是将 SQL 映射成 PromQL，并将 Prometheus 结果转换成 SQL 查询结果。这个字段是 PromQL 的表达式模板，查询监控表数据时使用查询条件改写模板中的变量，生成最终的查询表达式。
- LABELS：监控定义的 label，每一个 label 对应监控表中的一列。SQL 中如果包含对应列的过滤，对应的 PromQL 也会改变。
- QUANTILE：百分位。对于直方图类型的监控数据，指定一个默认百分位。如果值为 0，表示该监控表对应的监控不是直方图。
- COMMENT：对这个监控表的注释。

```
SELECT * FROM METRICS_TABLES LIMIT 5\G
```

输出结果如下：

```
*************************** 1. row ***************************
TABLE_NAME: abnormal_stores
    PROMQL: sum(pd_cluster_status{ type=~"store_disconnected_count|store_unhealth_count|
        ↪ store_low_space_count|store_down_count|store_offline_count|store_tombstone_count"})
    LABELS: instance,type
  QUANTILE: 0
   COMMENT:
*************************** 2. row ***************************
TABLE_NAME: etcd_disk_wal_fsync_rate
    PROMQL: delta(etcd_disk_wal_fsync_duration_seconds_count{$LABEL_CONDITIONS}[$RANGE_DURATION])
    LABELS: instance
  QUANTILE: 0
   COMMENT: The rate of writing WAL into the persistent storage
*************************** 3. row ***************************
TABLE_NAME: etcd_wal_fsync_duration
    PROMQL: histogram_quantile($QUANTILE, sum(rate(etcd_disk_wal_fsync_duration_seconds_bucket{
        ↪ $LABEL_CONDITIONS}[$RANGE_DURATION])) by (le,instance))
    LABELS: instance
  QUANTILE: 0.99
   COMMENT: The quantile time consumed of writing WAL into the persistent storage
*************************** 4. row ***************************
TABLE_NAME: etcd_wal_fsync_total_count
    PROMQL: sum(increase(etcd_disk_wal_fsync_duration_seconds_count{$LABEL_CONDITIONS}[
        ↪ $RANGE_DURATION])) by (instance)
    LABELS: instance
  QUANTILE: 0
   COMMENT: The total count of writing WAL into the persistent storage
*************************** 5. row ***************************
TABLE_NAME: etcd_wal_fsync_total_time
    PROMQL: sum(increase(etcd_disk_wal_fsync_duration_seconds_sum{$LABEL_CONDITIONS}[
        ↪ $RANGE_DURATION])) by (instance)
    LABELS: instance
  QUANTILE: 0
```

```
   COMMENT: The total time of writing WAL into the persistent storage
5 rows in set (0.00 sec)
```

14.11.18.2.28 PARTITIONS

PARTITIONS 表提供有关分区表的信息。

```
USE INFORMATION_SCHEMA;
DESC PARTITIONS;
```

输出结果如下：

```
+-------------------------------+--------------+------+------+---------+-------+
| Field                         | Type         | Null | Key  | Default | Extra |
+-------------------------------+--------------+------+------+---------+-------+
| TABLE_CATALOG                 | varchar(512) | YES  |      | NULL    |       |
| TABLE_SCHEMA                  | varchar(64)  | YES  |      | NULL    |       |
| TABLE_NAME                    | varchar(64)  | YES  |      | NULL    |       |
| PARTITION_NAME                | varchar(64)  | YES  |      | NULL    |       |
| SUBPARTITION_NAME             | varchar(64)  | YES  |      | NULL    |       |
| PARTITION_ORDINAL_POSITION    | bigint(21)   | YES  |      | NULL    |       |
| SUBPARTITION_ORDINAL_POSITION | bigint(21)   | YES  |      | NULL    |       |
| PARTITION_METHOD              | varchar(18)  | YES  |      | NULL    |       |
| SUBPARTITION_METHOD           | varchar(12)  | YES  |      | NULL    |       |
| PARTITION_EXPRESSION          | longtext     | YES  |      | NULL    |       |
| SUBPARTITION_EXPRESSION       | longtext     | YES  |      | NULL    |       |
| PARTITION_DESCRIPTION         | longtext     | YES  |      | NULL    |       |
| TABLE_ROWS                    | bigint(21)   | YES  |      | NULL    |       |
| AVG_ROW_LENGTH                | bigint(21)   | YES  |      | NULL    |       |
| DATA_LENGTH                   | bigint(21)   | YES  |      | NULL    |       |
| MAX_DATA_LENGTH               | bigint(21)   | YES  |      | NULL    |       |
| INDEX_LENGTH                  | bigint(21)   | YES  |      | NULL    |       |
| DATA_FREE                     | bigint(21)   | YES  |      | NULL    |       |
| CREATE_TIME                   | datetime     | YES  |      | NULL    |       |
| UPDATE_TIME                   | datetime     | YES  |      | NULL    |       |
| CHECK_TIME                    | datetime     | YES  |      | NULL    |       |
| CHECKSUM                      | bigint(21)   | YES  |      | NULL    |       |
| PARTITION_COMMENT             | varchar(80)  | YES  |      | NULL    |       |
| NODEGROUP                     | varchar(12)  | YES  |      | NULL    |       |
| TABLESPACE_NAME               | varchar(64)  | YES  |      | NULL    |       |
| TIDB_PARTITION_ID             | bigint(21)   | YES  |      | NULL    |       |
| TIDB_PLACEMENT_POLICY_NAME    | varchar(64)  | YES  |      | NULL    |       |
+-------------------------------+--------------+------+------+---------+-------+
27 rows in set (0.00 sec)
```

```
CREATE TABLE test.t1 (id INT NOT NULL PRIMARY KEY) PARTITION BY HASH (id) PARTITIONS 2;
SELECT * FROM PARTITIONS WHERE table_schema='test' AND table_name='t1'\G
```

输出结果如下：

```
*************************** 1. row ***************************
                TABLE_CATALOG: def
                 TABLE_SCHEMA: test
                   TABLE_NAME: t1
              PARTITION_NAME: p0
           SUBPARTITION_NAME: NULL
   PARTITION_ORDINAL_POSITION: 1
SUBPARTITION_ORDINAL_POSITION: NULL
             PARTITION_METHOD: HASH
          SUBPARTITION_METHOD: NULL
         PARTITION_EXPRESSION: `id`
      SUBPARTITION_EXPRESSION: NULL
        PARTITION_DESCRIPTION:
                   TABLE_ROWS: 0
               AVG_ROW_LENGTH: 0
                  DATA_LENGTH: 0
              MAX_DATA_LENGTH: 0
                 INDEX_LENGTH: 0
                    DATA_FREE: 0
                  CREATE_TIME: 2022-12-14 06:09:33
                  UPDATE_TIME: NULL
                   CHECK_TIME: NULL
                     CHECKSUM: NULL
           PARTITION_COMMENT:
                    NODEGROUP: NULL
              TABLESPACE_NAME: NULL
           TIDB_PARTITION_ID: 89
   TIDB_PLACEMENT_POLICY_NAME: NULL
*************************** 2. row ***************************
                TABLE_CATALOG: def
                 TABLE_SCHEMA: test
                   TABLE_NAME: t1
              PARTITION_NAME: p1
           SUBPARTITION_NAME: NULL
   PARTITION_ORDINAL_POSITION: 2
SUBPARTITION_ORDINAL_POSITION: NULL
             PARTITION_METHOD: HASH
          SUBPARTITION_METHOD: NULL
         PARTITION_EXPRESSION: `id`
      SUBPARTITION_EXPRESSION: NULL
```

```
        PARTITION_DESCRIPTION:
                   TABLE_ROWS: 0
               AVG_ROW_LENGTH: 0
                  DATA_LENGTH: 0
              MAX_DATA_LENGTH: 0
                 INDEX_LENGTH: 0
                    DATA_FREE: 0
                  CREATE_TIME: 2022-12-14 06:09:33
                  UPDATE_TIME: NULL
                   CHECK_TIME: NULL
                     CHECKSUM: NULL
            PARTITION_COMMENT:
                    NODEGROUP: NULL
              TABLESPACE_NAME: NULL
            TIDB_PARTITION_ID: 90
   TIDB_PLACEMENT_POLICY_NAME: NULL
2 rows in set (0.00 sec)
```

14.11.18.2.29 PLACEMENT_POLICIES

PLACEMENT_POLICIES 表展示所有放置策略 (placement policy) 的信息，见Placement Rules in SQL。

```
USE information_schema;
DESC placement_policies;
```

```
+----------------------+---------------+------+-----+---------+-------+
| Field                | Type          | Null | Key | Default | Extra |
+----------------------+---------------+------+-----+---------+-------+
| POLICY_ID            | bigint(64)    | NO   |     | <null>  |       |
| CATALOG_NAME         | varchar(512)  | NO   |     | <null>  |       |
| POLICY_NAME          | varchar(64)   | NO   |     | <null>  |       |
| PRIMARY_REGION       | varchar(1024) | YES  |     | <null>  |       |
| REGIONS              | varchar(1024) | YES  |     | <null>  |       |
| CONSTRAINTS          | varchar(1024) | YES  |     | <null>  |       |
| LEADER_CONSTRAINTS   | varchar(1024) | YES  |     | <null>  |       |
| FOLLOWER_CONSTRAINTS | varchar(1024) | YES  |     | <null>  |       |
| LEARNER_CONSTRAINTS  | varchar(1024) | YES  |     | <null>  |       |
| SCHEDULE             | varchar(20)   | YES  |     | <null>  |       |
| FOLLOWERS            | bigint(64)    | YES  |     | <null>  |       |
| LEARNERS             | bigint(64)    | YES  |     | <null>  |       |
+----------------------+---------------+------+-----+---------+-------+
12 rows in set (0.00 sec)
```

示例

PLACEMENT_POLICIES 表只展示放置策略 (placement policy) 的信息。如需查看所有信息的规范版本（放置策略，以及绑定放置策略的对象），请改用 SHOW PLACEMENT 语句：

```
CREATE TABLE t1 (a INT);
CREATE PLACEMENT POLICY p1 primary_region="us-east-1" regions="us-east-1";
CREATE TABLE t3 (a INT) PLACEMENT POLICY=p1;
SHOW PLACEMENT; -- 显示所有信息，包含 t3。
SELECT * FROM information_schema.placement_policies; -- 只显示放置策略，不包含 t3。
```

```
Query OK, 0 rows affected (0.09 sec)

Query OK, 0 rows affected (0.11 sec)

Query OK, 0 rows affected (0.08 sec)

+---------------+-------------------------------------------------+------------------+
| Target        | Placement                                       | Scheduling_State |
+---------------+-------------------------------------------------+------------------+
| POLICY p1     | PRIMARY_REGION="us-east-1" REGIONS="us-east-1" | NULL             |
| TABLE test.t3 | PRIMARY_REGION="us-east-1" REGIONS="us-east-1" | SCHEDULED        |
+---------------+-------------------------------------------------+------------------+
2 rows in set (0.00 sec)

+--
    ↪ ---------+--------------+-------------+----------------+-----------+-------------+----------------------+--
    ↪
| POLICY_ID | CATALOG_NAME | POLICY_NAME | PRIMARY_REGION | REGIONS   | CONSTRAINTS |
    ↪ LEADER_CONSTRAINTS | FOLLOWER_CONSTRAINTS | LEARNER_CONSTRAINTS | SCHEDULE | FOLLOWERS |
    ↪ LEARNERS |
+--
    ↪ ---------+--------------+-------------+----------------+-----------+-------------+----------------------+--
    ↪
| 1         | def          | p1          | us-east-1      | us-east-1 |             |
    ↪                    |                      |                      |          | 2         |
    ↪ 0        |
+--
    ↪ ---------+--------------+-------------+----------------+-----------+-------------+----------------------+--
    ↪
1 rows in set (0.00 sec)
```

14.11.18.2.30 PROCESSLIST

PROCESSLIST 和 SHOW PROCESSLIST 的功能一样，都是查看当前正在处理的请求。

PROCESSLIST 表比 SHOW PROCESSLIST 的结果多出下面几列：

- DIGEST 列：显示 SQL 语句的 digest。
- MEM 列：显示正在处理的请求已使用的内存，单位是 byte。
- DISK 列：显示磁盘空间使用情况，单位是 byte。
- TxnStart列：显示事务的开始时间。
- RESOURCE_GROUP列：显示对应的资源组名称。

```
USE information_schema;
DESC processlist;
```

```
+--------------------+---------------------+------+------+---------+-------+
| Field              | Type                | Null | Key  | Default | Extra |
+--------------------+---------------------+------+------+---------+-------+
| ID                 | bigint(21) unsigned | NO   |      | 0       |       |
| USER               | varchar(16)         | NO   |      |         |       |
| HOST               | varchar(64)         | NO   |      |         |       |
| DB                 | varchar(64)         | YES  |      | NULL    |       |
| COMMAND            | varchar(16)         | NO   |      |         |       |
| TIME               | int(7)              | NO   |      | 0       |       |
| STATE              | varchar(7)          | YES  |      | NULL    |       |
| INFO               | longtext            | YES  |      | NULL    |       |
| DIGEST             | varchar(64)         | YES  |      |         |       |
| MEM                | bigint(21) unsigned | YES  |      | NULL    |       |
| DISK               | bigint(21) unsigned | YES  |      | NULL    |       |
| TxnStart           | varchar(64)         | NO   |      |         |       |
| RESOURCE_GROUP     | varchar(32)         | NO   |      |         |       |
+--------------------+---------------------+------+------+---------+-------+
13 rows in set (0.00 sec)
```

```
SELECT * FROM processlist\G
```

```
*************************** 1. row ***************************
                ID: 2300033189772525975
              USER: root
              HOST: 127.0.0.1:51289
                DB: NULL
           COMMAND: Query
              TIME: 0
             STATE: autocommit
              INFO: SELECT * FROM processlist
            DIGEST: dbfaa16980ec628011029f0aaf0d160f4b040885240dfc567bf760d96d374f7e
               MEM: 0
              DISK: 0
          TxnStart:
    RESOURCE_GROUP: rg1
1 row in set (0.00 sec)
```

PROCESSLIST 表各列的含义如下：

- ID 列：客户连接 ID。
- USER 列：执行当前 PROCESS 的用户名。
- HOST 列：客户连接的地址。
- DB 列：当前连接的默认数据库名。
- COMMAND 列：当前 PROCESS 执行的命令类型。
- TIME 列：当前 PROCESS 的已经执行的时间，单位是秒。
- STATE 列：当前连接的状态。
- INFO 列：正在处理的请求语句。
- DIGEST 列：SQL 语句的 digest。
- MEM 列：正在处理的请求已使用的内存，单位是 byte。
- DISK 列：磁盘空间使用情况，单位是 byte。
- TxnStart列：显示事务的开始时间。
- RESOURCE_GROUP列：显示对应的资源组名称。

CLUSTER_PROCESSLIST

CLUSTER_PROCESSLIST 是 PROCESSLIST 对应的集群系统表，用于查询集群中所有 TiDB 节点的 PROCESSLIST 信息。CLUSTER_PROCESSLIST 表结构上比 PROCESSLIST 多一列 INSTANCE，表示该行数据来自的 TiDB 节点地址。

```
SELECT * FROM cluster_processlist;
```

```
+--
    ↪ ---------------+-----+------+----------+------+---------+------+------------+----------------------------
    ↪
| INSTANCE         | ID  | USER | HOST     | DB   | COMMAND | TIME | STATE      | INFO
    ↪                                                | MEM | TxnStart
    ↪                                 | RESOURCE_GROUP |
+--
    ↪ ---------------+-----+------+----------+------+---------+------+------------+----------------------------
    ↪


| 10.0.1.22:10080 | 150 | u1   | 10.0.1.1 | test | Query   | 0    | autocommit | select count(*)
    ↪ from usertable                        | 372 | 05-28 03:54:21.230(416976223923077223) |
    ↪ default        |
| 10.0.1.22:10080 | 138 | root | 10.0.1.1 | test | Query   | 0    | autocommit | SELECT * FROM
    ↪ information_schema.cluster_processlist | 0   | 05-28 03:54:21.230(416976223923077220) |
    ↪ rg1            |
| 10.0.1.22:10080 | 151 | u1   | 10.0.1.1 | test | Query   | 0    | autocommit | select count(*)
    ↪ from usertable                        | 372 | 05-28 03:54:21.230(416976223923077224) | rg2
    ↪             |
| 10.0.1.21:10080 | 15  | u2   | 10.0.1.1 | test | Query   | 0    | autocommit | select max(
    ↪ field0) from usertable                    | 496 | 05-28 03:54:21.230(416976223923077222)
    ↪ | default        |
```

```
| 10.0.1.21:10080 | 14   | u2    | 10.0.1.1 | test | Query    | 0     | autocommit | select max(
    ↪ field0) from usertable                    | 496 | 05-28 03:54:21.230(416976223923077225)
    ↪ | default         |
+--
    ↪ ---------------+-----+------+----------+------+---------+------+------------+-------------------------
    ↪
```

14.11.18.2.31 REFERENTIAL_CONSTRAINTS

REFERENTIAL_CONSTRAINTS 表提供 TiDB 表之间 FOREIGN KEY 关系的信息。

```
USE INFORMATION_SCHEMA;
DESC REFERENTIAL_CONSTRAINTS;
```

输出结果如下：

```
+--------------------------+--------------+------+------+---------+-------+
| Field                    | Type         | Null | Key  | Default | Extra |
+--------------------------+--------------+------+------+---------+-------+
| CONSTRAINT_CATALOG       | varchar(512) | NO   |      | NULL    |       |
| CONSTRAINT_SCHEMA        | varchar(64)  | NO   |      | NULL    |       |
| CONSTRAINT_NAME          | varchar(64)  | NO   |      | NULL    |       |
| UNIQUE_CONSTRAINT_CATALOG | varchar(512) | NO   |      | NULL    |       |
| UNIQUE_CONSTRAINT_SCHEMA  | varchar(64)  | NO   |      | NULL    |       |
| UNIQUE_CONSTRAINT_NAME    | varchar(64)  | YES  |      | NULL    |       |
| MATCH_OPTION             | varchar(64)  | NO   |      | NULL    |       |
| UPDATE_RULE              | varchar(64)  | NO   |      | NULL    |       |
| DELETE_RULE              | varchar(64)  | NO   |      | NULL    |       |
| TABLE_NAME               | varchar(64)  | NO   |      | NULL    |       |
| REFERENCED_TABLE_NAME    | varchar(64)  | NO   |      | NULL    |       |
+--------------------------+--------------+------+------+---------+-------+
11 rows in set (0.00 sec)
```

```
CREATE TABLE test.parent (
  id INT NOT NULL AUTO_INCREMENT,
  PRIMARY KEY (id)
);

CREATE TABLE test.child (
  id INT NOT NULL AUTO_INCREMENT,
  name varchar(255) NOT NULL,
  parent_id INT DEFAULT NULL,
  PRIMARY KEY (id),
  CONSTRAINT fk_parent FOREIGN KEY (parent_id) REFERENCES parent (id) ON UPDATE CASCADE ON DELETE
      ↪  RESTRICT
);
```

```
SELECT * FROM REFERENTIAL_CONSTRAINTS\G
```

输出结果如下：

```
*************************** 1. row ***************************
       CONSTRAINT_CATALOG: def
        CONSTRAINT_SCHEMA: test
          CONSTRAINT_NAME: fk_parent
UNIQUE_CONSTRAINT_CATALOG: def
 UNIQUE_CONSTRAINT_SCHEMA: test
   UNIQUE_CONSTRAINT_NAME: PRIMARY
             MATCH_OPTION: NONE
              UPDATE_RULE: CASCADE
              DELETE_RULE: RESTRICT
               TABLE_NAME: child
    REFERENCED_TABLE_NAME: parent
1 row in set (0.00 sec)
```

14.11.18.2.32 RESOURCE_GROUPS

> 警告:
>
> 资源管控是 TiDB 在 v6.6.0 中引入的实验特性，其语法或者行为表现在 GA 前可能会发生变化。

RESOURCE_GROUPS 表展示所有资源组 (resource group) 的信息，见使用资源管控 (Resource Control) 实现资源隔离。

```
USE information_schema;
DESC resource_groups;
```

```
+------------+-------------+------+------+---------+-------+
| Field      | Type        | Null | Key  | Default | Extra |
+------------+-------------+------+------+---------+-------+
| NAME       | varchar(32) | NO   |      | NULL    |       |
| RU_PER_SEC | bigint(21)  | YES  |      | NULL    |       |
| PRIORITY   | varchar(6)  | YES  |      | NULL    |       |
| BURSTABLE  | varchar(3)  | YES  |      | NULL    |       |
+------------+-------------+------+------+---------+-------+
3 rows in set (0.00 sec)
```

示例

```
SELECT * FROM information_schema.resource_groups; -- 查看资源组，TiDB 默认预置 `default` 资源组
```

```
+---------+------------+----------+-----------+
| NAME    | RU_PER_SEC | PRIORITY | BURSTABLE |
+---------+------------+----------+-----------+
| default | UNLIMITED  | MEDIUM   | YES       |
+---------+------------+----------+-----------+
```

```
CREATE RESOURCE GROUP rg1 RU_PER_SEC=1000; -- 创建资源组 `rg1`
```

```
Query OK, 0 rows affected (0.34 sec)
```

```
SHOW CREATE RESOURCE GROUP rg1; -- 显示 `rg1` 资源组的定义
```

```
+----------------+-------------------------------------------------------------------+
| Resource_Group | Create Resource Group                                             |
+----------------+-------------------------------------------------------------------+
| rg1            | CREATE RESOURCE GROUP `rg1` RU_PER_SEC=1000 PRIORITY="MEDIUM" |
+----------------+-------------------------------------------------------------------+
1 row in set (0.00 sec)
```

```
SELECT * FROM information_schema.resource_groups WHERE NAME = 'rg1'; -- 查看资源组 `rg1`
```

```
+------+------------+----------+-----------+
| NAME | RU_PER_SEC | PRIORITY | BURSTABLE |
+------+------------+----------+-----------+
| rg1  |       1000 | MEDIUM   | NO        |
+------+------------+----------+-----------+
1 row in set (0.00 sec)
```

RESOURCE_GROUPS 表中列的含义如下：

- NAME：资源组名称。
- RU_PER_SEC：资源组的回填速度，单位为每秒回填的Request Unit (RU) 数量。
- PRIORITY：任务在 TiKV 上处理的绝对优先级。不同的资源按照 PRIORITY 的设置进行调度，PRIORITY 高的任务会被优先调度。如果资源组的 PRIORITY 相同，则会根据 RU_PER_SEC 的配置按比例调度。如果不指定 PRIORITY，资源组的默认优先级为 MEDIUM。
- BURSTABLE：是否允许此资源组超额使用剩余的系统资源。

> **注意：**
>
> TiDB 集群在初始化时会自动创建 default 资源组，其 RU_PER_SEC 的默认值为 UNLIMITED (等同于 INT 类型最大值，即 2147483647)，且为 BURSTABLE 模式。所有未绑定资源组的请求都将自动绑定至此资源组。在新建配置其他资源组时，建议根据实际情况修改 default 资源组的配置。

14.11.18.2.33 SCHEMATA

SCHEMATA 表提供了关于数据库的信息。表中的数据与 SHOW DATABASES 语句的执行结果等价。

```
USE information_schema;
desc SCHEMATA;
```

```
+----------------------------+--------------+------+------+---------+-------+
| Field                      | Type         | Null | Key  | Default | Extra |
+----------------------------+--------------+------+------+---------+-------+
| CATALOG_NAME               | varchar(512) | YES  |      | NULL    |       |
| SCHEMA_NAME                | varchar(64)  | YES  |      | NULL    |       |
| DEFAULT_CHARACTER_SET_NAME | varchar(64)  | YES  |      | NULL    |       |
| DEFAULT_COLLATION_NAME     | varchar(32)  | YES  |      | NULL    |       |
| SQL_PATH                   | varchar(512) | YES  |      | NULL    |       |
+----------------------------+--------------+------+------+---------+-------+
5 rows in set (0.00 sec)
```

```
SELECT * FROM SCHEMATA;
```

```
+--
    ↪ ------------+--------------------+----------------------------+------------------------+-----------+
    ↪
| CATALOG_NAME | SCHEMA_NAME        | DEFAULT_CHARACTER_SET_NAME | DEFAULT_COLLATION_NAME |
    ↪ SQL_PATH |
+--
    ↪ ------------+--------------------+----------------------------+------------------------+-----------+
    ↪
| def          | INFORMATION_SCHEMA | utf8mb4                    | utf8mb4_bin            | NULL
    ↪      |
| def          | METRICS_SCHEMA     | utf8mb4                    | utf8mb4_bin            | NULL
    ↪      |
| def          | mysql              | utf8mb4                    | utf8mb4_bin            | NULL
    ↪      |
| def          | PERFORMANCE_SCHEMA | utf8mb4                    | utf8mb4_bin            | NULL
    ↪      |
| def          | test               | utf8mb4                    | utf8mb4_bin            | NULL
    ↪      |
+--
    ↪ ------------+--------------------+----------------------------+------------------------+-----------+
    ↪
5 rows in set (0.00 sec)
```

SCHEMATA 表各列字段含义如下：

- CATALOG_NAME：数据库归属的目录名，该列值永远为 def。

- SCHEMA_NAME：数据库的名字。
- DEFAULT_CHARACTER_SET_NAME：数据库的默认字符集。
- DEFAULT_COLLATION_NAME：数据库的默认 collation。
- SQL_PATH：该项值永远为 NULL。

14.11.18.2.34 SEQUENCES

SEQUENCES 表提供了有关序列的信息。TiDB 中序列的功能是参照 MariaDB 中的类似功能来实现的。

```
USE INFORMATION_SCHEMA;
DESC SEQUENCES;
```

输出结果如下：

```
+-----------------+--------------+------+------+---------+-------+
| Field           | Type         | Null | Key  | Default | Extra |
+-----------------+--------------+------+------+---------+-------+
| TABLE_CATALOG   | varchar(512) | NO   |      | NULL    |       |
| SEQUENCE_SCHEMA | varchar(64)  | NO   |      | NULL    |       |
| SEQUENCE_NAME   | varchar(64)  | NO   |      | NULL    |       |
| CACHE           | tinyint(0)   | NO   |      | NULL    |       |
| CACHE_VALUE     | bigint(21)   | YES  |      | NULL    |       |
| CYCLE           | tinyint(0)   | NO   |      | NULL    |       |
| INCREMENT       | bigint(21)   | NO   |      | NULL    |       |
| MAX_VALUE       | bigint(21)   | YES  |      | NULL    |       |
| MIN_VALUE       | bigint(21)   | YES  |      | NULL    |       |
| START           | bigint(21)   | YES  |      | NULL    |       |
| COMMENT         | varchar(64)  | YES  |      | NULL    |       |
+-----------------+--------------+------+------+---------+-------+
11 rows in set (0.00 sec)
```

创建一个名为 test.seq 的序列，并查询这个序列的下一个值：

```
CREATE SEQUENCE test.seq;
SELECT nextval(test.seq);
```

输出结果如下：

```
+-------------------+
| nextval(test.seq) |
+-------------------+
|                 1 |
+-------------------+
1 row in set (0.01 sec)
```

查询数据库中的所有序列：

```
SELECT * FROM SEQUENCES\G
```

输出结果如下：

```
*************************** 1. row ***************************
  TABLE_CATALOG: def
SEQUENCE_SCHEMA: test
  SEQUENCE_NAME: seq
          CACHE: 1
    CACHE_VALUE: 1000
          CYCLE: 0
      INCREMENT: 1
      MAX_VALUE: 9223372036854775806
      MIN_VALUE: 1
          START: 1
        COMMENT:
1 row in set (0.00 sec)
```

14.11.18.2.35 SESSION_VARIABLES

SESSION_VARIABLES 表提供了关于 session 变量的信息。表中的数据跟 SHOW SESSION VARIABLES 语句执行结果类似。

```
USE INFORMATION_SCHEMA;
DESC SESSION_VARIABLES;
```

输出结果如下：

```
+----------------+---------------+------+------+---------+-------+
| Field          | Type          | Null | Key  | Default | Extra |
+----------------+---------------+------+------+---------+-------+
| VARIABLE_NAME  | varchar(64)   | YES  |      | NULL    |       |
| VARIABLE_VALUE | varchar(1024) | YES  |      | NULL    |       |
+----------------+---------------+------+------+---------+-------+
2 rows in set (0.00 sec)
```

查看 SESSION_VARIABLES 表的前 10 条信息：

```
SELECT * FROM SESSION_VARIABLES ORDER BY variable_name LIMIT 10;
```

输出结果如下：

```
+-----------------------------------+------------------+
| VARIABLE_NAME                     | VARIABLE_VALUE   |
+-----------------------------------+------------------+
| allow_auto_random_explicit_insert | OFF              |
| auto_increment_increment          | 1                |
| auto_increment_offset             | 1                |
| autocommit                        | ON               |
```

```
| automatic_sp_privileges           | 1                |
| avoid_temporal_upgrade            | OFF              |
| back_log                          | 80               |
| basedir                           | /usr/local/mysql |
| big_tables                        | OFF              |
| bind_address                      | *                |
+-----------------------------------+------------------+
10 rows in set (0.00 sec)
```

SESSION_VARIABLES 表各列字段含义如下：

- VARIABLE_NAME：数据库中 session 级变量的名称。
- VARIABLE_VALUE：数据库中对应该 session 变量名的具体值。

14.11.18.2.36 SLOW_QUERY

SLOW_QUERY 表中提供了当前节点的慢查询相关的信息，其内容通过解析当前节点的 TiDB 慢查询日志而来，列名和慢日志中的字段名是一一对应。关于如何使用该表调查和改善慢查询，请参考慢查询日志文档。

```
USE INFORMATION_SCHEMA;
DESC slow_query;
```

输出结果示例如下：

```
+------------------------------+---------------------+------+------+---------+-------+
| Field                        | Type                | Null | Key  | Default | Extra |
+------------------------------+---------------------+------+------+---------+-------+
| Time                         | timestamp(6)        | NO   | PRI  | NULL    |       |
| Txn_start_ts                 | bigint(20) unsigned | YES  |      | NULL    |       |
| User                         | varchar(64)         | YES  |      | NULL    |       |
| Host                         | varchar(64)         | YES  |      | NULL    |       |
| Conn_ID                      | bigint(20) unsigned | YES  |      | NULL    |       |
| Exec_retry_count             | bigint(20) unsigned | YES  |      | NULL    |       |
| Exec_retry_time              | double              | YES  |      | NULL    |       |
| Query_time                   | double              | YES  |      | NULL    |       |
| Parse_time                   | double              | YES  |      | NULL    |       |
| Compile_time                 | double              | YES  |      | NULL    |       |
| Rewrite_time                 | double              | YES  |      | NULL    |       |
| Preproc_subqueries           | bigint(20) unsigned | YES  |      | NULL    |       |
| Preproc_subqueries_time      | double              | YES  |      | NULL    |       |
| Optimize_time                | double              | YES  |      | NULL    |       |
| Wait_TS                      | double              | YES  |      | NULL    |       |
| Prewrite_time                | double              | YES  |      | NULL    |       |
| Wait_prewrite_binlog_time    | double              | YES  |      | NULL    |       |
| Commit_time                  | double              | YES  |      | NULL    |       |
| Get_commit_ts_time           | double              | YES  |      | NULL    |       |
```

```
| Commit_backoff_time           | double              | YES  |     | NULL    |       |
| Backoff_types                 | varchar(64)         | YES  |     | NULL    |       |
| Resolve_lock_time             | double              | YES  |     | NULL    |       |
| Local_latch_wait_time         | double              | YES  |     | NULL    |       |
| Write_keys                    | bigint(22)          | YES  |     | NULL    |       |
| Write_size                    | bigint(22)          | YES  |     | NULL    |       |
| Prewrite_region               | bigint(22)          | YES  |     | NULL    |       |
| Txn_retry                     | bigint(22)          | YES  |     | NULL    |       |
| Cop_time                      | double              | YES  |     | NULL    |       |
| Process_time                  | double              | YES  |     | NULL    |       |
| Wait_time                     | double              | YES  |     | NULL    |       |
| Backoff_time                  | double              | YES  |     | NULL    |       |
| LockKeys_time                 | double              | YES  |     | NULL    |       |
| Request_count                 | bigint(20) unsigned | YES  |     | NULL    |       |
| Total_keys                    | bigint(20) unsigned | YES  |     | NULL    |       |
| Process_keys                  | bigint(20) unsigned | YES  |     | NULL    |       |
| Rocksdb_delete_skipped_count  | bigint(20) unsigned | YES  |     | NULL    |       |
| Rocksdb_key_skipped_count     | bigint(20) unsigned | YES  |     | NULL    |       |
| Rocksdb_block_cache_hit_count | bigint(20) unsigned | YES  |     | NULL    |       |
| Rocksdb_block_read_count      | bigint(20) unsigned | YES  |     | NULL    |       |
| Rocksdb_block_read_byte       | bigint(20) unsigned | YES  |     | NULL    |       |
| DB                            | varchar(64)         | YES  |     | NULL    |       |
| Index_names                   | varchar(100)        | YES  |     | NULL    |       |
| Is_internal                   | tinyint(1)          | YES  |     | NULL    |       |
| Digest                        | varchar(64)         | YES  |     | NULL    |       |
| Stats                         | varchar(512)        | YES  |     | NULL    |       |
| Cop_proc_avg                  | double              | YES  |     | NULL    |       |
| Cop_proc_p90                  | double              | YES  |     | NULL    |       |
| Cop_proc_max                  | double              | YES  |     | NULL    |       |
| Cop_proc_addr                 | varchar(64)         | YES  |     | NULL    |       |
| Cop_wait_avg                  | double              | YES  |     | NULL    |       |
| Cop_wait_p90                  | double              | YES  |     | NULL    |       |
| Cop_wait_max                  | double              | YES  |     | NULL    |       |
| Cop_wait_addr                 | varchar(64)         | YES  |     | NULL    |       |
| Mem_max                       | bigint(20)          | YES  |     | NULL    |       |
| Disk_max                      | bigint(20)          | YES  |     | NULL    |       |
| KV_total                      | double              | YES  |     | NULL    |       |
| PD_total                      | double              | YES  |     | NULL    |       |
| Backoff_total                 | double              | YES  |     | NULL    |       |
| Write_sql_response_total      | double              | YES  |     | NULL    |       |
| Result_rows                   | bigint(22)          | YES  |     | NULL    |       |
| Warnings                      | longtext            | YES  |     | NULL    |       |
| Backoff_Detail                | varchar(4096)       | YES  |     | NULL    |       |
| Prepared                      | tinyint(1)          | YES  |     | NULL    |       |
| Succ                          | tinyint(1)          | YES  |     | NULL    |       |
```

```
| IsExplicitTxn                | tinyint(1)          | YES  |      | NULL    |       |
| IsWriteCacheTable            | tinyint(1)          | YES  |      | NULL    |       |
| Plan_from_cache              | tinyint(1)          | YES  |      | NULL    |       |
| Plan_from_binding            | tinyint(1)          | YES  |      | NULL    |       |
| Has_more_results             | tinyint(1)          | YES  |      | NULL    |       |
| Plan                         | longtext            | YES  |      | NULL    |       |
| Plan_digest                  | varchar(128)        | YES  |      | NULL    |       |
| Binary_plan                  | longtext            | YES  |      | NULL    |       |
| Prev_stmt                    | longtext            | YES  |      | NULL    |       |
| Query                        | longtext            | YES  |      | NULL    |       |
+------------------------------+---------------------+------+------+---------+-------+
74 rows in set (0.001 sec)
```

CLUSTER_SLOW_QUERY table

CLUSTER_SLOW_QUERY 表中提供了集群所有节点的慢查询相关的信息，其内容通过解析 TiDB 慢查询日志而来，该表使用上和 SLOW_QUERY 表一样。CLUSTER_SLOW_QUERY 表结构上比 SLOW_QUERY 多一列 INSTANCE，表示该行慢查询信息来自的 TiDB 节点地址。关于如何使用该表调查和改善慢查询，请参考慢查询日志文档。

```
DESC CLUSTER_SLOW_QUERY;
```

输出结果示例如下：

```
+------------------------------+---------------------+------+------+---------+-------+
| Field                        | Type                | Null | Key  | Default | Extra |
+------------------------------+---------------------+------+------+---------+-------+
| INSTANCE                     | varchar(64)         | YES  |      | NULL    |       |
| Time                         | timestamp(6)        | NO   | PRI  | NULL    |       |
| Txn_start_ts                 | bigint(20) unsigned | YES  |      | NULL    |       |
| User                         | varchar(64)         | YES  |      | NULL    |       |
| Host                         | varchar(64)         | YES  |      | NULL    |       |
| Conn_ID                      | bigint(20) unsigned | YES  |      | NULL    |       |
| Exec_retry_count             | bigint(20) unsigned | YES  |      | NULL    |       |
| Exec_retry_time              | double              | YES  |      | NULL    |       |
| Query_time                   | double              | YES  |      | NULL    |       |
| Parse_time                   | double              | YES  |      | NULL    |       |
| Compile_time                 | double              | YES  |      | NULL    |       |
| Rewrite_time                 | double              | YES  |      | NULL    |       |
| Preproc_subqueries           | bigint(20) unsigned | YES  |      | NULL    |       |
| Preproc_subqueries_time      | double              | YES  |      | NULL    |       |
| Optimize_time                | double              | YES  |      | NULL    |       |
| Wait_TS                      | double              | YES  |      | NULL    |       |
| Prewrite_time                | double              | YES  |      | NULL    |       |
| Wait_prewrite_binlog_time    | double              | YES  |      | NULL    |       |
| Commit_time                  | double              | YES  |      | NULL    |       |
| Get_commit_ts_time           | double              | YES  |      | NULL    |       |
| Commit_backoff_time          | double              | YES  |      | NULL    |       |
```

```
| Backoff_types                | varchar(64)         | YES  |     | NULL    |       |
| Resolve_lock_time            | double              | YES  |     | NULL    |       |
| Local_latch_wait_time        | double              | YES  |     | NULL    |       |
| Write_keys                   | bigint(22)          | YES  |     | NULL    |       |
| Write_size                   | bigint(22)          | YES  |     | NULL    |       |
| Prewrite_region              | bigint(22)          | YES  |     | NULL    |       |
| Txn_retry                    | bigint(22)          | YES  |     | NULL    |       |
| Cop_time                     | double              | YES  |     | NULL    |       |
| Process_time                 | double              | YES  |     | NULL    |       |
| Wait_time                    | double              | YES  |     | NULL    |       |
| Backoff_time                 | double              | YES  |     | NULL    |       |
| LockKeys_time                | double              | YES  |     | NULL    |       |
| Request_count                | bigint(20) unsigned | YES  |     | NULL    |       |
| Total_keys                   | bigint(20) unsigned | YES  |     | NULL    |       |
| Process_keys                 | bigint(20) unsigned | YES  |     | NULL    |       |
| Rocksdb_delete_skipped_count | bigint(20) unsigned | YES  |     | NULL    |       |
| Rocksdb_key_skipped_count    | bigint(20) unsigned | YES  |     | NULL    |       |
| Rocksdb_block_cache_hit_count | bigint(20) unsigned | YES  |     | NULL    |       |
| Rocksdb_block_read_count     | bigint(20) unsigned | YES  |     | NULL    |       |
| Rocksdb_block_read_byte      | bigint(20) unsigned | YES  |     | NULL    |       |
| DB                           | varchar(64)         | YES  |     | NULL    |       |
| Index_names                  | varchar(100)        | YES  |     | NULL    |       |
| Is_internal                  | tinyint(1)          | YES  |     | NULL    |       |
| Digest                       | varchar(64)         | YES  |     | NULL    |       |
| Stats                        | varchar(512)        | YES  |     | NULL    |       |
| Cop_proc_avg                 | double              | YES  |     | NULL    |       |
| Cop_proc_p90                 | double              | YES  |     | NULL    |       |
| Cop_proc_max                 | double              | YES  |     | NULL    |       |
| Cop_proc_addr                | varchar(64)         | YES  |     | NULL    |       |
| Cop_wait_avg                 | double              | YES  |     | NULL    |       |
| Cop_wait_p90                 | double              | YES  |     | NULL    |       |
| Cop_wait_max                 | double              | YES  |     | NULL    |       |
| Cop_wait_addr                | varchar(64)         | YES  |     | NULL    |       |
| Mem_max                      | bigint(20)          | YES  |     | NULL    |       |
| Disk_max                     | bigint(20)          | YES  |     | NULL    |       |
| KV_total                     | double              | YES  |     | NULL    |       |
| PD_total                     | double              | YES  |     | NULL    |       |
| Backoff_total                | double              | YES  |     | NULL    |       |
| Write_sql_response_total     | double              | YES  |     | NULL    |       |
| Result_rows                  | bigint(22)          | YES  |     | NULL    |       |
| Warnings                     | longtext            | YES  |     | NULL    |       |
| Backoff_Detail               | varchar(4096)       | YES  |     | NULL    |       |
| Prepared                     | tinyint(1)          | YES  |     | NULL    |       |
| Succ                         | tinyint(1)          | YES  |     | NULL    |       |
| IsExplicitTxn                | tinyint(1)          | YES  |     | NULL    |       |
```

```
| IsWriteCacheTable             | tinyint(1)          | YES  |      | NULL    |       |
| Plan_from_cache               | tinyint(1)          | YES  |      | NULL    |       |
| Plan_from_binding             | tinyint(1)          | YES  |      | NULL    |       |
| Has_more_results              | tinyint(1)          | YES  |      | NULL    |       |
| Plan                          | longtext            | YES  |      | NULL    |       |
| Plan_digest                   | varchar(128)        | YES  |      | NULL    |       |
| Binary_plan                   | longtext            | YES  |      | NULL    |       |
| Prev_stmt                     | longtext            | YES  |      | NULL    |       |
| Query                         | longtext            | YES  |      | NULL    |       |
+-------------------------------+---------------------+------+------+---------+-------+
75 rows in set (0.001 sec)
```

查询集群系统表时，TiDB 也会将相关计算下推给其他节点执行，而不是把所有节点的数据都取回来，可以查看执行计划，如下：

```
DESC SELECT COUNT(*) FROM CLUSTER_SLOW_QUERY WHERE user = 'u1';
```

输出结果示例如下：

```
+--
    ↪ -------------------------+---------+-----------+-------------------------+-----------------------------
    ↪
| id                          | estRows  | task      | access object           | operator info
    ↪                                      |
+--
    ↪ -------------------------+---------+-----------+-------------------------+-----------------------------
    ↪
| StreamAgg_7                 | 1.00     | root      |                         | funcs:count(1)->
    ↪ Column#75                            |
| └─TableReader_13            | 10.00    | root      |                         | data:Selection_12
    ↪                                      |
|   └─Selection_12            | 10.00    | cop[tidb] |                         | eq(
    ↪ INFORMATION_SCHEMA.cluster_slow_query.user, "u1") |
|     └─TableFullScan_11 | 10000.00 | cop[tidb] | table:CLUSTER_SLOW_QUERY | keep order:false,
    ↪ stats:pseudo                         |
+--
    ↪ -------------------------+---------+-----------+-------------------------+-----------------------------
    ↪
4 rows in set (0.00 sec)
```

上面执行计划表示，会将 `user = u1` 条件下推给其他的 (cop) TiDB 节点执行，也会把聚合算子（即上面输出结果中的 StreamAgg 算子）下推。

目前由于没有对系统表收集统计信息，所以有时会导致某些聚合算子不能下推，导致执行较慢，用户可以通过手动指定聚合下推的 SQL HINT 来将聚合算子下推，示例如下：

```
SELECT /*+ AGG_TO_COP() */ COUNT(*) FROM CLUSTER_SLOW_QUERY GROUP BY user;
```

14.11.18.2.37 STATISTICS

STATISTICS 表提供了关于表索引的信息。

```
USE information_schema;
DESC statistics;
```

```
+---------------+---------------+------+------+---------+-------+
| Field         | Type          | Null | Key  | Default | Extra |
+---------------+---------------+------+------+---------+-------+
| TABLE_CATALOG | varchar(512)  | YES  |      | NULL    |       |
| TABLE_SCHEMA  | varchar(64)   | YES  |      | NULL    |       |
| TABLE_NAME    | varchar(64)   | YES  |      | NULL    |       |
| NON_UNIQUE    | varchar(1)    | YES  |      | NULL    |       |
| INDEX_SCHEMA  | varchar(64)   | YES  |      | NULL    |       |
| INDEX_NAME    | varchar(64)   | YES  |      | NULL    |       |
| SEQ_IN_INDEX  | bigint(2)     | YES  |      | NULL    |       |
| COLUMN_NAME   | varchar(21)   | YES  |      | NULL    |       |
| COLLATION     | varchar(1)    | YES  |      | NULL    |       |
| CARDINALITY   | bigint(21)    | YES  |      | NULL    |       |
| SUB_PART      | bigint(3)     | YES  |      | NULL    |       |
| PACKED        | varchar(10)   | YES  |      | NULL    |       |
| NULLABLE      | varchar(3)    | YES  |      | NULL    |       |
| INDEX_TYPE    | varchar(16)   | YES  |      | NULL    |       |
| COMMENT       | varchar(16)   | YES  |      | NULL    |       |
| INDEX_COMMENT | varchar(1024) | YES  |      | NULL    |       |
| IS_VISIBLE    | varchar(3)    | YES  |      | NULL    |       |
| Expression    | varchar(64)   | YES  |      | NULL    |       |
+---------------+---------------+------+------+---------+-------+
18 rows in set (0.00 sec)
```

STATISTICS 表中列的含义如下：

- TABLE_CATALOG：包含索引的表所属的目录的名称。这个值总是 def。
- TABLE_SCHEMA：包含索引的表所属的数据库的名称。
- TABLE_NAME：包含索引的表的名称。
- NON_UNIQUE：如果索引不能包含重复项，则为 0；如果可以，则为 1。
- INDEX_SCHEMA：索引所属的数据库的名称。
- INDEX_NAME：索引的名称。如果索引是主键，那么名称总是 PRIMARY。
- SEQ_IN_INDEX：索引中的列序号，从 1 开始。
- COLUMN_NAME：列名。请参见表达式列的说明。
- COLLATION：列在索引中的排序方式。取值可以是 A（升序）、D（降序）或 NULL（未排序）。
- CARDINALITY：TiDB 未使用该字段。该字段的值总是 0。
- SUB_PART：索引的前缀。如果只对列的部分前缀进行索引，则为索引字符的数量；如果对整个列进行索引，则为 NULL。
- PACKED：TiDB 未使用该字段。这个值总是 NULL。

- NULLABLE：如果列可能包含 NULL 值，则值为 YES；如果不包含，则值为 ''。
- INDEX_TYPE：索引的类型。
- COMMENT：其他与索引有关的信息。
- INDEX_COMMENT：在创建索引时为索引提供的带有注释属性的任何注释。
- IS_VISIBLE：优化器能否使用该索引。
- Expression 对于非表达式部分的索引键，这个值为 NULL；对于表达式部分的索引键，这个值为表达式本身。可参考表达式索引

下列语句是等价的：

```
SELECT * FROM INFORMATION_SCHEMA.STATISTICS
  WHERE table_name = 'tbl_name'
  AND table_schema = 'db_name'

SHOW INDEX
  FROM tbl_name
  FROM db_name
```

14.11.18.2.38 TABLES

TABLES 表提供了数据库里关于表的信息。

```
USE information_schema;
DESC tables;
```

```
+---------------------------+---------------+------+------+----------+-------+
| Field                     | Type          | Null | Key  | Default  | Extra |
+---------------------------+---------------+------+------+----------+-------+
| TABLE_CATALOG             | varchar(512)  | YES  |      | NULL     |       |
| TABLE_SCHEMA              | varchar(64)   | YES  |      | NULL     |       |
| TABLE_NAME                | varchar(64)   | YES  |      | NULL     |       |
| TABLE_TYPE                | varchar(64)   | YES  |      | NULL     |       |
| ENGINE                    | varchar(64)   | YES  |      | NULL     |       |
| VERSION                   | bigint(21)    | YES  |      | NULL     |       |
| ROW_FORMAT                | varchar(10)   | YES  |      | NULL     |       |
| TABLE_ROWS                | bigint(21)    | YES  |      | NULL     |       |
| AVG_ROW_LENGTH            | bigint(21)    | YES  |      | NULL     |       |
| DATA_LENGTH               | bigint(21)    | YES  |      | NULL     |       |
| MAX_DATA_LENGTH           | bigint(21)    | YES  |      | NULL     |       |
| INDEX_LENGTH              | bigint(21)    | YES  |      | NULL     |       |
| DATA_FREE                 | bigint(21)    | YES  |      | NULL     |       |
| AUTO_INCREMENT            | bigint(21)    | YES  |      | NULL     |       |
| CREATE_TIME               | datetime      | YES  |      | NULL     |       |
| UPDATE_TIME               | datetime      | YES  |      | NULL     |       |
| CHECK_TIME                | datetime      | YES  |      | NULL     |       |
| TABLE_COLLATION           | varchar(32)   | NO   |      | utf8_bin |       |
```

```
| CHECKSUM                  | bigint(21)    | YES  |      | NULL     |       |
| CREATE_OPTIONS            | varchar(255)  | YES  |      | NULL     |       |
| TABLE_COMMENT             | varchar(2048) | YES  |      | NULL     |       |
| TIDB_TABLE_ID             | bigint(21)    | YES  |      | NULL     |       |
| TIDB_ROW_ID_SHARDING_INFO | varchar(255)  | YES  |      | NULL     |       |
+---------------------------+---------------+------+------+----------+-------+
23 rows in set (0.00 sec)
```

```
SELECT * FROM tables WHERE table_schema='mysql' AND table_name='user'\G
```

```
*************************** 1. row ***************************
            TABLE_CATALOG: def
             TABLE_SCHEMA: mysql
               TABLE_NAME: user
               TABLE_TYPE: BASE TABLE
                   ENGINE: InnoDB
                  VERSION: 10
               ROW_FORMAT: Compact
               TABLE_ROWS: 0
           AVG_ROW_LENGTH: 0
              DATA_LENGTH: 0
          MAX_DATA_LENGTH: 0
             INDEX_LENGTH: 0
                DATA_FREE: 0
           AUTO_INCREMENT: NULL
              CREATE_TIME: 2020-07-05 09:25:51
              UPDATE_TIME: NULL
               CHECK_TIME: NULL
          TABLE_COLLATION: utf8mb4_bin
                 CHECKSUM: NULL
           CREATE_OPTIONS:
            TABLE_COMMENT:
            TIDB_TABLE_ID: 5
TIDB_ROW_ID_SHARDING_INFO: NULL
1 row in set (0.00 sec)
```

下列语句是等价的：

```
SELECT table_name FROM INFORMATION_SCHEMA.TABLES
  WHERE table_schema = 'db_name'
  [AND table_name LIKE 'wild']

SHOW TABLES
  FROM db_name
  [LIKE 'wild']
```

TABLES 表各列字段含义如下：

- TABLE_CATALOG：表所属的目录的名称。该值始终为 def。
- TABLE_SCHEMA：表所属数据库的名称。
- TABLE_NAME：表的名称。
- TABLE_TYPE：表的类型。
- ENGINE：存储引擎类型。该值暂为 ‘InnoDB’。
- VERSION：版本，默认值为 10。
- ROW_FORMAT：行格式。该值暂为 ‘Compact’。
- TABLE_ROWS：统计信息中该表所存的行数。
- AVG_ROW_LENGTH：该表中所存数据的平均行长度。平均行长度 = DATA_LENGTH / 统计信息中的行数。
- DATA_LENGTH：数据长度。数据长度 = 统计信息中的行数 × 元组各列存储长度和，这里尚未考虑 TiKV 的副本数。
- MAX_DATA_LENGTH：最大数据长度。该值暂为 0，表示没有最大数据长度的限制。
- INDEX_LENGTH：索引长度。索引长度 = 统计信息中的行数 × 索引元组各列长度和，这里尚未考虑 TiKV 的副本数。
- DATA_FREE：空间碎片。该值暂为 0。
- AUTO_INCREMENT：该表中自增主键自动增量的当前值。
- CREATE_TIME：该表的创建时间。
- UPDATE_TIME：该表的更新时间。
- CHECK_TIME：该表的检查时间。
- TABLE_COLLATION：该表的字符校验编码集。
- CHECKSUM：校验和。
- CREATE_OPTIONS：创建选项。
- TABLE_COMMENT：表的注释、备注。

表中的信息大部分定义自 MySQL，只有两列是 TiDB 新增的：

- TIDB_TABLE_ID：标识表的内部 ID，该 ID 在一个 TiDB 集群内部唯一。
- TIDB_ROW_ID_SHARDING_INFO：标识表的 Sharding 类型，可能的值为：
  - "NOT_SHARDED"：表未被 Shard。
  - "NOT_SHARDED(PK_IS_HANDLE)"：一个定义了整型主键的表未被 Shard。
  - "PK_AUTO_RANDOM_BITS={bit_number}"：一个定义了整型主键的表由于定义了 AUTO_RANDOM 而被 Shard。
  - "SHARD_BITS={bit_number}"：表使用 SHARD_ROW_ID_BITS={bit_number} 进行了 Shard。
  - NULL：表属于系统表或 View，无法被 Shard。

14.11.18.2.39 TABLE_CONSTRAINTS

TABLE_CONSTRAINTS 表记录了表的约束信息。

```
USE information_schema;
DESC table_constraints;
```

```
+--------------------+--------------+------+------+---------+-------+
| Field              | Type         | Null | Key  | Default | Extra |
+--------------------+--------------+------+------+---------+-------+
| CONSTRAINT_CATALOG | varchar(512) | YES  |      | NULL    |       |
| CONSTRAINT_SCHEMA  | varchar(64)  | YES  |      | NULL    |       |
| CONSTRAINT_NAME    | varchar(64)  | YES  |      | NULL    |       |
| TABLE_SCHEMA       | varchar(64)  | YES  |      | NULL    |       |
| TABLE_NAME         | varchar(64)  | YES  |      | NULL    |       |
| CONSTRAINT_TYPE    | varchar(64)  | YES  |      | NULL    |       |
+--------------------+--------------+------+------+---------+-------+
6 rows in set (0.00 sec)
```

```
SELECT * FROM table_constraints WHERE constraint_type='UNIQUE';
```

```
+--
    ↪ ------------------+--------------------+-------------------------+--------------------+-----------------
    ↪
| CONSTRAINT_CATALOG | CONSTRAINT_SCHEMA  | CONSTRAINT_NAME         | TABLE_SCHEMA       |
    ↪ TABLE_NAME                          | CONSTRAINT_TYPE |
+--
    ↪ ------------------+--------------------+-------------------------+--------------------+-----------------
    ↪
| def                | mysql              | name                    | mysql              |
    ↪ help_topic                          | UNIQUE          |
| def                | mysql              | tbl                     | mysql              |
    ↪ stats_meta                          | UNIQUE          |
| def                | mysql              | tbl                     | mysql              |
    ↪ stats_histograms                    | UNIQUE          |
| def                | mysql              | tbl                     | mysql              |
    ↪ stats_buckets                       | UNIQUE          |
| def                | mysql              | delete_range_index      | mysql              |
    ↪ gc_delete_range                     | UNIQUE          |
| def                | mysql              | delete_range_done_index | mysql              |
    ↪ gc_delete_range_done                | UNIQUE          |
| def                | PERFORMANCE_SCHEMA | SCHEMA_NAME             | PERFORMANCE_SCHEMA |
    ↪ events_statements_summary_by_digest | UNIQUE          |
+--
    ↪ ------------------+--------------------+-------------------------+--------------------+-----------------
    ↪
7 rows in set (0.01 sec)
```

`TABLE_CONSTRAINTS` 表中列的含义如下：

- `CONSTRAINT_CATALOG`：约束所属的目录的名称。这个值总是 `def`。

- CONSTRAINT_SCHEMA：约束所属的数据库的名称。
- CONSTRAINT_NAME：约束的名称。
- TABLE_NAME：表的名称。
- CONSTRAINT_TYPE：约束的类型。取值可以是 UNIQUE、PRIMARY KEY 或者 FOREIGN KEY。UNIQUE 和 PRIMARY KEY 信息与 SHOW INDEX 语句的执行结果类似。

14.11.18.2.40 TABLE_STORAGE_STATS

TABLE_STORAGE_STATS 表提供有关由存储引擎 (TiKV) 存储的表大小的信息。

```
USE INFORMATION_SCHEMA;
DESC TABLE_STORAGE_STATS;
```

输出结果如下：

```
+--------------------+-------------+------+------+---------+-------+
| Field              | Type        | Null | Key  | Default | Extra |
+--------------------+-------------+------+------+---------+-------+
| TABLE_SCHEMA       | varchar(64) | YES  |      | NULL    |       |
| TABLE_NAME         | varchar(64) | YES  |      | NULL    |       |
| TABLE_ID           | bigint(21)  | YES  |      | NULL    |       |
| PEER_COUNT         | bigint(21)  | YES  |      | NULL    |       |
| REGION_COUNT       | bigint(21)  | YES  |      | NULL    |       |
| EMPTY_REGION_COUNT | bigint(21)  | YES  |      | NULL    |       |
| TABLE_SIZE         | bigint(64)  | YES  |      | NULL    |       |
| TABLE_KEYS         | bigint(64)  | YES  |      | NULL    |       |
+--------------------+-------------+------+------+---------+-------+
8 rows in set (0.00 sec)
```

```
CREATE TABLE test.t1 (id INT);
INSERT INTO test.t1 VALUES (1);
SELECT * FROM TABLE_STORAGE_STATS WHERE table_schema = 'test' AND table_name = 't1'\G
```

输出结果如下：

```
*************************** 1. row ***************************
      TABLE_SCHEMA: test
        TABLE_NAME: t1
          TABLE_ID: 56
        PEER_COUNT: 1
      REGION_COUNT: 1
EMPTY_REGION_COUNT: 1
        TABLE_SIZE: 1
        TABLE_KEYS: 0
1 row in set (0.00 sec)
```

TABLE_STORAGE_STATS 表各列字段含义如下：

- TABLE_SCHEMA：表所在 schema 名字
- TABLE_NAME：表名
- TABLE_ID：表 ID
- PEER_COUNT：副本个数
- REGION_COUNT：表所在的 Region 个数
- EMPTY_REGION_COUNT：没有该表数据的 Region 个数
- TABLE_SIZE：数据量大小，单位 MiB
- TABLE_KEYS：表记录个数

14.11.18.2.41 TIDB_HOT_REGIONS

TIDB_HOT_REGIONS 表提供了关于当前热点 Region 的相关信息。历史热点信息可以在TIDB_HOT_REGIONS_HISTORY 表查看。

```
USE information_schema;
DESC tidb_hot_regions;
```

```
+----------------+-------------+------+------+---------+-------+
| Field          | Type        | Null | Key  | Default | Extra |
+----------------+-------------+------+------+---------+-------+
| TABLE_ID       | bigint(21)  | YES  |      | NULL    |       |
| INDEX_ID       | bigint(21)  | YES  |      | NULL    |       |
| DB_NAME        | varchar(64) | YES  |      | NULL    |       |
| TABLE_NAME     | varchar(64) | YES  |      | NULL    |       |
| INDEX_NAME     | varchar(64) | YES  |      | NULL    |       |
| REGION_ID      | bigint(21)  | YES  |      | NULL    |       |
| TYPE           | varchar(64) | YES  |      | NULL    |       |
| MAX_HOT_DEGREE | bigint(21)  | YES  |      | NULL    |       |
| REGION_COUNT   | bigint(21)  | YES  |      | NULL    |       |
| FLOW_BYTES     | bigint(21)  | YES  |      | NULL    |       |
+----------------+-------------+------+------+---------+-------+
10 rows in set (0.00 sec)
```

TIDB_HOT_REGIONS 表各列字段含义如下：

- TABLE_ID：热点 Region 所在表的 ID。
- INDEX_ID：热点 Region 所在索引的 ID。
- DB_NAME：热点 Region 所在数据库对象的数据库名。
- TABLE_NAME：热点 Region 所在表的名称。
- INDEX_NAME：热点 Region 所在索引的名称。
- REGION_ID：热点 Region 的 ID。
- TYPE：热点 Region 的类型。
- MAX_HOT_DEGREE：该 Region 的最大热度。
- REGION_COUNT：所在实例的热点 Region 数量。
- FLOW_BYTES：该 Region 内读写的字节数量。

14.11.18.2.42 TIDB_HOT_REGIONS_HISTORY

TIDB_HOT_REGIONS_HISTORY 表提供了关于历史热点 Region 的相关信息，这些信息由 PD 定期存储在本地，存储间隔为hot-regions-write-interval 的值（默认值为 10m），历史热点信息保持的期限为hot-regions-reserved-days 的值（默认值为 7），详情参见PD 配置文件描述。

```
USE information_schema;
DESC tidb_hot_regions_history;
```

```
+-------------+--------------+------+------+---------+-------+
| Field       | Type         | Null | Key  | Default | Extra |
+-------------+--------------+------+------+---------+-------+
| UPDATE_TIME | timestamp(6) | YES  |      | NULL    |       |
| DB_NAME     | varchar(64)  | YES  |      | NULL    |       |
| TABLE_NAME  | varchar(64)  | YES  |      | NULL    |       |
| TABLE_ID    | bigint(21)   | YES  |      | NULL    |       |
| INDEX_NAME  | varchar(64)  | YES  |      | NULL    |       |
| INDEX_ID    | bigint(21)   | YES  |      | NULL    |       |
| REGION_ID   | bigint(21)   | YES  |      | NULL    |       |
| STORE_ID    | bigint(21)   | YES  |      | NULL    |       |
| PEER_ID     | bigint(21)   | YES  |      | NULL    |       |
| IS_LEARNER  | tinyint(1)   | NO   |      | 0       |       |
| IS_LEADER   | tinyint(1)   | NO   |      | 0       |       |
| TYPE        | varchar(64)  | YES  |      | NULL    |       |
| HOT_DEGREE  | bigint(21)   | YES  |      | NULL    |       |
| FLOW_BYTES  | double       | YES  |      | NULL    |       |
| KEY_RATE    | double       | YES  |      | NULL    |       |
| QUERY_RATE  | double       | YES  |      | NULL    |       |
+-------------+--------------+------+------+---------+-------+
16 rows in set (0.00 sec)
```

TIDB_HOT_REGIONS_HISTORY 表各列字段含义如下：

- UPDATE_TIME：热点 Region 更新时间。
- DB_NAME：热点 Region 所在数据库对象的数据库名。
- TABLE_ID：热点 Region 所在表的 ID。
- TABLE_NAME：热点 Region 所在表的名称。
- INDEX_NAME：热点 Region 所在索引的名称。
- INDEX_ID：热点 Region 所在索引的 ID。
- REGION_ID：热点 Region 的 ID。
- STORE_ID：热点 Region 所在 TiKV Store 的 ID。
- PEER_ID：热点 Region 对应的副本 Peer 的 ID。
- IS_LEARNER：PEER 是否是 LEARNER。
- IS_LEADER：PEER 是否是 LEADER。
- TYPE：热点 Region 的类型。
- HOT_DEGREE：该热点 Region 的热度。

- FLOW_BYTES：该 Region 内读写的字节数量。
- KEY_RATE：该 Region 内读写的 Key 数量。
- QUERY_RATE：该 Region 内读写的 Query 数量。

> 注意：
>
> TIDB_HOT_REGIONS_HISTORY 表的 UPDATE_TIME、REGION_ID、STORE_ID、PEER_ID、IS_LEARNER、IS_LEADER、TYPE 字段会下推到 PD 服务器执行。所以为了降低使用该表的开销，必须指定搜索时间范围，然后尽可能地指定更多的条件。例如 `select * from tidb_hot_regions_history where store_id = 11 and update_time > '2020-05-18 20:40:00' and update_time < '2020-05-18 21:40:00' and type='write'`。

常见使用场景

- 查询一段时间内的所有热点 Regions。将以下语句中的 update_time 替换为实际所需的时间即可。

```
SELECT * FROM INFORMATION_SCHEMA.TIDB_HOT_REGIONS_HISTORY WHERE update_time >'2021-08-18 21:40:00' and update_time <'2021-09-19 00:00:00';
```

> 注意：
>
> UPDATE_TIME 同样支持 Unix 时间戳。例如：`update_time >TIMESTAMP('2012-08-18 21:40:00')` 或 `update_time > FROM_UNIXTIME(1629294000.000)`。

- 查询某张表指定时间内的热点 Regions，替换 update_time、table_name 即可。

```
SELECT * FROM INFORMATION_SCHEMA.TIDB_HOT_REGIONS_HISTORY WHERE update_time >'2021-08-18 21:40:00' and update_time <'2021-09-19 00:00:00' and TABLE_NAME = 'table_name';
```

- 查询某张表指定时间内热点 Regions 的分布，替换 update_time、table_name 即可。

```
SELECT count(region_id) cnt, store_id FROM INFORMATION_SCHEMA.TIDB_HOT_REGIONS_HISTORY WHERE update_time >'2021-08-18 21:40:00' and update_time <'2021-09-19 00:00:00' and table_name = 'table_name' GROUP BY STORE_ID ORDER BY cnt DESC;
```

- 查询某张表指定时间内热点 Leader Regions 的分布，替换 update_time、table_name 即可。

```
SELECT count(region_id) cnt, store_id FROM INFORMATION_SCHEMA.TIDB_HOT_REGIONS_HISTORY WHERE update_time >'2021-08-18 21:40:00' and update_time <'2021-09-19 00:00:00' and table_name = 'table_name' and is_leader=1 GROUP BY STORE_ID ORDER BY cnt DESC;
```

- 查询某张表指定时间内热点 Index Regions 的分布，替换 update_time、table_name 即可。

```
SELECT count(region_id) cnt, index_name, store_id FROM INFORMATION_SCHEMA.
    ↪ TIDB_HOT_REGIONS_HISTORY WHERE update_time >'2021-08-18 21:40:00' and update_time <'
    ↪ 2021-09-19 00:00:00' and table_name = 'table_name' group by index_name, store_id
    ↪ order by index_name,cnt desc;
```

- 查询某张表指定时间内热点 Index Leader Regions 的分布，替换 update_time、table_name 即可。

```
SELECT count(region_id) cnt, index_name, store_id FROM INFORMATION_SCHEMA.
    ↪ TIDB_HOT_REGIONS_HISTORY WHERE update_time >'2021-08-18 21:40:00' and update_time <'
    ↪ 2022-09-19 00:00:00' and table_name = 'table_name' and is_leader=1 group by
    ↪ index_name, store_id order by index_name,cnt desc;
```

14.11.18.2.43 TIDB_INDEXES

TIDB_INDEXES 记录了所有表中的 INDEX 信息。

```
USE information_schema;
DESC tidb_indexes;
```

```
+---------------+---------------+------+------+---------+-------+
| Field         | Type          | Null | Key  | Default | Extra |
+---------------+---------------+------+------+---------+-------+
| TABLE_SCHEMA  | varchar(64)   | YES  |      | NULL    |       |
| TABLE_NAME    | varchar(64)   | YES  |      | NULL    |       |
| NON_UNIQUE    | bigint(21)    | YES  |      | NULL    |       |
| KEY_NAME      | varchar(64)   | YES  |      | NULL    |       |
| SEQ_IN_INDEX  | bigint(21)    | YES  |      | NULL    |       |
| COLUMN_NAME   | varchar(64)   | YES  |      | NULL    |       |
| SUB_PART      | bigint(21)    | YES  |      | NULL    |       |
| INDEX_COMMENT | varchar(2048) | YES  |      | NULL    |       |
| Expression    | varchar(64)   | YES  |      | NULL    |       |
| INDEX_ID      | bigint(21)    | YES  |      | NULL    |       |
| IS_VISIBLE    | varchar(64)   | YES  |      | NULL    |       |
| CLUSTERED     | varchar(64)   | YES  |      | NULL    |       |
+---------------+---------------+------+------+---------+-------+
12 rows in set (0.00 sec)
```

INDEX_ID 是 TiDB 为每个索引分配的唯一 ID。它可以与从另一个表或 API 获得的 INDEX_ID 一起执行 join 操作。

例如，你可以在SLOW_QUERY 表 中获取某些慢查询所涉及的 TABLE_ID 和 INDEX_ID，然后使用以下 SQL 语句获取特定索引信息：

```
SELECT
 tidb_indexes.*
FROM
 tidb_indexes,
```

```
 tables
WHERE
  tidb_indexes.table_schema = tables.table_schema
 AND tidb_indexes.table_name = tidb_indexes.table_name
 AND tables.tidb_table_id = ?
 AND index_id = ?
```

TIDB_INDEXES 表中列的含义如下：

- TABLE_SCHEMA：索引所在表的所属数据库的名称。
- TABLE_NAME：索引所在表的名称。
- NON_UNIQUE：如果索引是唯一的，则为 0，否则为 1。
- KEY_NAME：索引的名称。如果索引是主键，则名称为 PRIMARY。
- SEQ_IN_INDEX：索引中列的顺序编号，从 1 开始。
- COLUMN_NAME：索引所在的列名。
- SUB_PART：索引前缀长度。如果列是部分被索引，则该值为被索引的字符数量，否则为 NULL。
- INDEX_COMMENT：创建索引时以 COMMENT 标注的注释。
- INDEX_ID：索引的 ID。
- IS_VISIBLE：索引是否可见。
- CLUSTERED：是否为聚簇索引。

14.11.18.2.44 TIDB_SERVERS_INFO

TIDB_SERVERS_INFO 表提供了 TiDB 集群中 TiDB 服务器的信息（即 tidb-server 进程）。

```
USE INFORMATION_SCHEMA;
DESC TIDB_SERVERS_INFO;
```

输出结果如下：

```
+---------------+--------------+------+------+---------+-------+
| Field         | Type         | Null | Key  | Default | Extra |
+---------------+--------------+------+------+---------+-------+
| DDL_ID        | varchar(64)  | YES  |      | NULL    |       |
| IP            | varchar(64)  | YES  |      | NULL    |       |
| PORT          | bigint(21)   | YES  |      | NULL    |       |
| STATUS_PORT   | bigint(21)   | YES  |      | NULL    |       |
| LEASE         | varchar(64)  | YES  |      | NULL    |       |
| VERSION       | varchar(64)  | YES  |      | NULL    |       |
| GIT_HASH      | varchar(64)  | YES  |      | NULL    |       |
| BINLOG_STATUS | varchar(64)  | YES  |      | NULL    |       |
| LABELS        | varchar(128) | YES  |      | NULL    |       |
+---------------+--------------+------+------+---------+-------+
9 rows in set (0.00 sec)
```

查看 TIDB_SERVERS_INFO 表的信息：

```
SELECT * FROM TIDB_SERVERS_INFO\G
```

输出结果如下：

```
*************************** 1. row ***************************
      DDL_ID: 771c169d-0a3a-48ea-a93c-a4d6751d3674
          IP: 0.0.0.0
        PORT: 4000
 STATUS_PORT: 10080
       LEASE: 45s
     VERSION: 5.7.25-TiDB-v6.5.0
    GIT_HASH: 827d8ff2d22ac4c93ae1b841b79d468211e1d393
BINLOG_STATUS: Off
      LABELS:
1 row in set (0.006 sec)
```

14.11.18.2.45 TIDB_TRX

TIDB_TRX 表提供了当前 TiDB 节点上正在执行的事务的信息。

```
USE INFORMATION_SCHEMA;
DESC TIDB_TRX;
```

输出结果如下：

```
+--
    ↪ ----------------------+--------------------------------------------------------------------------+------+------+-
    ↪
| Field                  | Type                                                                     |
    ↪ Null | Key  | Default | Extra |
+--
    ↪ ----------------------+--------------------------------------------------------------------------+------+------+-
    ↪
| ID                     | bigint(21) unsigned                                                      | NO
    ↪   | PRI  | NULL    |       |
| START_TIME             | timestamp(6)                                                             | YES
    ↪   |      | NULL    |       |
| CURRENT_SQL_DIGEST     | varchar(64)                                                              | YES
    ↪   |      | NULL    |       |
| CURRENT_SQL_DIGEST_TEXT | text                                                                    | YES
    ↪   |      | NULL    |       |
| STATE                  | enum('Idle','Running','LockWaiting','Committing','RollingBack')          | YES
    ↪   |      | NULL    |       |
| WAITING_START_TIME     | timestamp(6)                                                             | YES
    ↪   |      | NULL    |       |
```

```
| MEM_BUFFER_KEYS         | bigint(64)                                                
```

注意：

- 仅拥有 PROCESS 权限的用户可以获取该表中的完整信息。没有 PROCESS 权限的用户则只能查询到当前用户所执行的事务的信息。
- CURRENT_SQL_DIGEST 和 ALL_SQL_DIGESTS 列中的信息 (SQL Digest) 为 SQL 语句进行归一化后计算得到的哈希值。CURRENT_SQL_DIGEST_TEXT 列中的信息和函数 TIDB_DECODE_SQL_DIGESTS 所得到的结果均为内部从 Statements Summary 系列表中查询得到，因而存在内部查询不到对应语句的可能性。关于 SQL Digest 和 Statements Summary 相关表的详细说明，请参阅Statement Summary Tables。
- TIDB_DECODE_SQL_DIGESTS 函数调用开销较大，如果对大量事务的信息调用该函数查询历史 SQL，可能查询耗时较长。如果集群规模较大、同一时刻并发运行的事务较多，请避免直接在查询 TIDB_TRX 全表的同时直接将该函数用于 ALL_SQL_DIGEST 列（即尽量避免 SELECT *, tidb_decode_sql_digests(all_sql_digests)FROM TIDB_TRX 这样的用法）。
- 目前 TIDB_TRX 表暂不支持显示 TiDB 内部事务相关的信息。

示例

查看 TIDB_TRX 表中的信息：

```
SELECT * FROM INFORMATION_SCHEMA.TIDB_TRX\G
```

输出结果如下：

```
*************************** 1. row ***************************
                   ID: 426789913200689153
           START_TIME: 2021-08-04 10:51:54.883000
    CURRENT_SQL_DIGEST: NULL
CURRENT_SQL_DIGEST_TEXT: NULL
                STATE: Idle
    WAITING_START_TIME: NULL
       MEM_BUFFER_KEYS: 1
      MEM_BUFFER_BYTES: 29
            SESSION_ID: 7
                  USER: root
                    DB: test
       ALL_SQL_DIGESTS: ["e6f07d43b5c21db0fbb9a31feac2dc599787763393dd5acbfad80e247eb02ad5","04
           ↪ fa858fa491c62d194faec2ab427261cc7998b3f1ccf8f6844febca504cb5e9","
           ↪ b83710fa8ab7df8504920e8569e48654f621cf828afbe7527fd003b79f48da9e"]
*************************** 2. row ***************************
                   ID: 426789921471332353
           START_TIME: 2021-08-04 10:52:26.433000
    CURRENT_SQL_DIGEST: 38b03afa5debbdf0326a014dbe5012a62c51957f1982b3093e748460f8b00821
CURRENT_SQL_DIGEST_TEXT: update `t` set `v` = `v` + ? where `id` = ?
                STATE: LockWaiting
```

```
     WAITING_START_TIME: 2021-08-04 10:52:35.106568
        MEM_BUFFER_KEYS: 0
       MEM_BUFFER_BYTES: 0
             SESSION_ID: 9
                   USER: root
                     DB: test
        ALL_SQL_DIGESTS: ["e6f07d43b5c21db0fbb9a31feac2dc599787763393dd5acbfad80e247eb02ad5","38
            ↪ b03afa5debbdf0326a014dbe5012a62c51957f1982b3093e748460f8b00821"]
2 rows in set (0.01 sec)
```

此示例的查询结果表示：当前节点有两个运行中的事务，第一个事务正在闲置状态（STATE 为 Idle ↪ ，CURRENT_SQL_DIGEST 为 NULL），该事务已经执行过 3 条语句（ALL_SQL_DIGESTS 列表中有三条记录，分别为执行过的 3 条语句的 Digest）；第二个事务正在执行一条语句并正在等锁（STATE 为 LockWaiting，WAITING_START_TIME 显示了等锁开始的时间），该事务已经执行过两条语句，当前正在执行的语句形如 "update `t` set `v` = `v` + ? where `id` = ?"。

```
SELECT id, all_sql_digests, tidb_decode_sql_digests(all_sql_digests) AS all_sqls FROM
    ↪ INFORMATION_SCHEMA.TIDB_TRX\G
```

输出结果如下：

```
*************************** 1. row ***************************
             id: 426789913200689153
all_sql_digests: ["e6f07d43b5c21db0fbb9a31feac2dc599787763393dd5acbfad80e247eb02ad5","04
    ↪ fa858fa491c62d194faec2ab427261cc7998b3f1ccf8f6844febca504cb5e9","
    ↪ b83710fa8ab7df8504920e8569e48654f621cf828afbe7527fd003b79f48da9e"]
       all_sqls: ["begin","insert into `t` values ( ... )","update `t` set `v` = `v` + ?"]
*************************** 2. row ***************************
             id: 426789921471332353
all_sql_digests: ["e6f07d43b5c21db0fbb9a31feac2dc599787763393dd5acbfad80e247eb02ad5","38
    ↪ b03afa5debbdf0326a014dbe5012a62c51957f1982b3093e748460f8b00821"]
       all_sqls: ["begin","update `t` set `v` = `v` + ? where `id` = ?"]
```

此查询对 TIDB_TRX 表的 ALL_SQL_DIGESTS 列调用了TIDB_DECODE_SQL_DIGESTS 函数，将 SQL Digest 的数组通过系统内部的查询转换成归一化 SQL 语句的数组，以便于直观地获取事务历史执行过的语句的信息。但是需要注意上述查询扫描了 TIDB_TRX 全表，并对每一行都调用了 TIDB_DECODE_SQL_DIGESTS 函数；而 TIDB_DECODE_SQL_DIGESTS 函数调用的开销很大，所以如果集群中并发事务数量较多，请尽量避免这种查询。

CLUSTER_TIDB_TRX

TIDB_TRX 表仅提供单个 TiDB 节点中正在执行的事务信息。如果要查看整个集群上所有 TiDB 节点中正在执行的事务信息，需要查询 CLUSTER_TIDB_TRX 表。与 TIDB_TRX 表的查询结果相比，CLUSTER_TIDB_TRX 表的查询结果额外包含了 INSTANCE 字段。INSTANCE 字段展示了集群中各节点的 IP 地址和端口，用于区分事务所在的 TiDB 节点。

```
USE INFORMATION_SCHEMA;
DESC CLUSTER_TIDB_TRX;
```

输出结果如下：

```
+--
    ↪ ----------------------+------------------------------------------------------------------+------+------+-
    ↪
| Field                   | Type                                                             |
    ↪ Null | Key  | Default | Extra |
+--
    ↪ ----------------------+------------------------------------------------------------------+------+------+-
    ↪
| INSTANCE                | varchar(64)                                                      | YES
    ↪   |      | NULL    |       |
| ID                      | bigint(21) unsigned                                              | NO
    ↪   | PRI  | NULL    |       |
| START_TIME              | timestamp(6)                                                     | YES
    ↪   |      | NULL    |       |
| CURRENT_SQL_DIGEST      | varchar(64)                                                      | YES
    ↪   |      | NULL    |       |
| CURRENT_SQL_DIGEST_TEXT | text                                                             | YES
    ↪   |      | NULL    |       |
| STATE                   | enum('Idle','Running','LockWaiting','Committing','RollingBack') | YES
    ↪   |      | NULL    |       |
| WAITING_START_TIME      | timestamp(6)                                                     | YES
    ↪   |      | NULL    |       |
| MEM_BUFFER_KEYS         | bigint(64)                                                       | YES
    ↪   |      | NULL    |       |
| MEM_BUFFER_BYTES        | bigint(64)                                                       | YES
    ↪   |      | NULL    |       |
| SESSION_ID              | bigint(21) unsigned                                              | YES
    ↪   |      | NULL    |       |
| USER                    | varchar(16)                                                      | YES
    ↪   |      | NULL    |       |
| DB                      | varchar(64)                                                      | YES
    ↪   |      | NULL    |       |
| ALL_SQL_DIGESTS         | text                                                             | YES
    ↪   |      | NULL    |       |
| RELATED_TABLE_IDS       | text                                                             | YES
    ↪   |      | NULL    |       |
+--
    ↪ ----------------------+------------------------------------------------------------------+------+------+-
    ↪
14 rows in set (0.00 sec)
```

14.11.18.2.46 TIFLASH_REPLICA

TIFLASH_REPLICA 表提供了有关可用的 TiFlash 副本的信息。

```
USE INFORMATION_SCHEMA;
DESC TIFLASH_REPLICA;
```

输出结果如下：

```
+-----------------+-------------+------+------+---------+-------+
| Field           | Type        | Null | Key  | Default | Extra |
+-----------------+-------------+------+------+---------+-------+
| TABLE_SCHEMA    | varchar(64) | YES  |      | NULL    |       |
| TABLE_NAME      | varchar(64) | YES  |      | NULL    |       |
| TABLE_ID        | bigint(21)  | YES  |      | NULL    |       |
| REPLICA_COUNT   | bigint(64)  | YES  |      | NULL    |       |
| LOCATION_LABELS | varchar(64) | YES  |      | NULL    |       |
| AVAILABLE       | tinyint(1)  | YES  |      | NULL    |       |
| PROGRESS        | double      | YES  |      | NULL    |       |
+-----------------+-------------+------+------+---------+-------+
7 rows in set (0.01 sec)
```

TIFLASH_REPLICA 表中各列的字段含义如下：

- TABLE_SCHEMA：表所属的数据库的名称。
- TABLE_NAME：表的名称。
- TABLE_ID：表的内部 ID，该 ID 在一个 TiDB 集群内部唯一。
- REPLICA_COUNT：TiFlash 副本数。
- LOCATION_LABELS：设置 TiFlash 副本时设置的 LocationLabelList。
- AVAILABLE：表的 TiFlash 副本是否可用。1 代表可用，TiDB 优化器将依据查询代价智能选择下推查询到 TiKV 或 TiFlash；0 代表不可用，TiDB 将不会下推查询到 TiFlash。副本状态变为可用之后就不再改变。
- PROGRESS：TiFlash 副本同步进度，范围是 [0, 1]，精确到小数点后两位，刷新的精度为分钟级。当 AVAILABLE 字段为 1 时，如果 PROGRESS 小于 1，表示 TiFlash 副本落后 TiKV 较多，下推到 TiFlash 的查询很可能会因为等待数据同步超时而失败。

14.11.18.2.47 TIFLASH_SEGMENTS

> **警告：**
>
> 请不要在生产环境中使用该系统表，因为该表字段信息尚未固定下来，可能会在 TiDB 未来版本中发生变化。

TIFLASH_SEGMENTS 表提供了 TiFlash 内部数据表的统计信息。

```
USE information_schema;
DESC tiflash_segments;
```

```
+------------------------------+-------------+------+------+---------+-------+
| Field                        | Type        | Null | Key  | Default | Extra |
+------------------------------+-------------+------+------+---------+-------+
| DATABASE                     | varchar(64) | YES  |      | NULL    |       |
| TABLE                        | varchar(64) | YES  |      | NULL    |       |
| TIDB_DATABASE                | varchar(64) | YES  |      | NULL    |       |
| TIDB_TABLE                   | varchar(64) | YES  |      | NULL    |       |
| TABLE_ID                     | bigint(64)  | YES  |      | NULL    |       |
| IS_TOMBSTONE                 | bigint(64)  | YES  |      | NULL    |       |
| SEGMENT_ID                   | bigint(64)  | YES  |      | NULL    |       |
| RANGE                        | varchar(64) | YES  |      | NULL    |       |
| EPOCH                        | bigint(64)  | YES  |      | NULL    |       |
| ROWS                         | bigint(64)  | YES  |      | NULL    |       |
| SIZE                         | bigint(64)  | YES  |      | NULL    |       |
| DELTA_RATE                   | double      | YES  |      | NULL    |       |
| DELTA_MEMTABLE_ROWS          | bigint(64)  | YES  |      | NULL    |       |
| DELTA_MEMTABLE_SIZE          | bigint(64)  | YES  |      | NULL    |       |
| DELTA_MEMTABLE_COLUMN_FILES  | bigint(64)  | YES  |      | NULL    |       |
| DELTA_MEMTABLE_DELETE_RANGES | bigint(64)  | YES  |      | NULL    |       |
| DELTA_PERSISTED_PAGE_ID      | bigint(64)  | YES  |      | NULL    |       |
| DELTA_PERSISTED_ROWS         | bigint(64)  | YES  |      | NULL    |       |
| DELTA_PERSISTED_SIZE         | bigint(64)  | YES  |      | NULL    |       |
| DELTA_PERSISTED_COLUMN_FILES | bigint(64)  | YES  |      | NULL    |       |
| DELTA_PERSISTED_DELETE_RANGES| bigint(64)  | YES  |      | NULL    |       |
| DELTA_CACHE_SIZE             | bigint(64)  | YES  |      | NULL    |       |
| DELTA_INDEX_SIZE             | bigint(64)  | YES  |      | NULL    |       |
| STABLE_PAGE_ID               | bigint(64)  | YES  |      | NULL    |       |
| STABLE_ROWS                  | bigint(64)  | YES  |      | NULL    |       |
| STABLE_SIZE                  | bigint(64)  | YES  |      | NULL    |       |
| STABLE_DMFILES               | bigint(64)  | YES  |      | NULL    |       |
| STABLE_DMFILES_ID_0          | bigint(64)  | YES  |      | NULL    |       |
| STABLE_DMFILES_ROWS          | bigint(64)  | YES  |      | NULL    |       |
| STABLE_DMFILES_SIZE          | bigint(64)  | YES  |      | NULL    |       |
| STABLE_DMFILES_SIZE_ON_DISK  | bigint(64)  | YES  |      | NULL    |       |
| STABLE_DMFILES_PACKS         | bigint(64)  | YES  |      | NULL    |       |
| TIFLASH_INSTANCE             | varchar(64) | YES  |      | NULL    |       |
+------------------------------+-------------+------+------+---------+-------+
33 rows in set (0.00 sec)
```

`TIFLASH_SEGMENTS` 表中各列的字段含义如下：

- `DATABASE`：Segment 所属表在 TiFlash 内部所属的数据库的名称。
- `TABLE`：Segment 所属表在 TiFlash 内部的名称。
- `TIDB_DATABASE`：Segment 所属表所属的数据库的名称。
- `TIDB_TABLE`：Segment 所属表的名称。

- TABLE_ID：Segment 所属表的内部 ID，该 ID 在一个 TiDB 集群内部唯一。
- IS_TOMBSTONE：表示该 Segment 所属表是否处于待回收状态，为 1 表示该表处于待回收状态，为 0 则表示处于正常状态。
- SEGMENT_ID：Segment ID，该 ID 在每张表内部唯一。
- RANGE：Segment 包含数据的范围。
- EPOCH：Segment 的更新版本号，每个 Segment 的版本号单调递增。
- ROWS：Segment 的数据总行数。
- SIZE：Segment 的数据总大小（单位：字节）。
- DELTA_RATE：Delta 层的数据总行数占 Segment 数据总行数的比例。
- DELTA_MEMTABLE_ROWS：Delta 层缓存中的数据总行数。
- DELTA_MEMTABLE_SIZE：Delta 层缓存中的数据总大小（单位：字节）。
- DELTA_MEMTABLE_COLUMN_FILES：Delta 层缓存中 Column File 数量。
- DELTA_MEMTABLE_DELETE_RANGES：Delta 层缓存中 Delete Range 数量。
- DELTA_PERSISTED_PAGE_ID：Delta 层数据在磁盘上存储的 ID。
- DELTA_PERSISTED_ROWS：Delta 层已落盘的数据总行数。
- DELTA_PERSISTED_SIZE：Delta 层已落盘的数据总大小（单位：字节）。
- DELTA_PERSISTED_COLUMN_FILES：Delta 层已落盘 Column File 数量。
- DELTA_PERSISTED_DELETE_RANGES：Delta 层已落盘 Delete Range 数量。
- DELTA_CACHE_SIZE：Delta 层缓存大小（单位：字节）。
- DELTA_INDEX_SIZE：Delta 层索引大小（单位：字节）。
- STABLE_PAGE_ID：Stable 层数据在磁盘上存储的标识。
- STABLE_ROWS：Stable 层的数据总行数。
- STABLE_SIZE：Stable 层的数据总大小（单位：字节）。
- STABLE_DMFILES：Stable 层 DMFile 数量。
- STABLE_DMFILES_ID_0：Stable 层第一个 DMFile 在磁盘上存储的 ID。
- STABLE_DMFILES_ROWS：Stable 层 DMFile 中的数据总行数。
- STABLE_DMFILES_SIZE：Stable 层 DMFile 中的数据总大小（单位：字节）。
- STABLE_DMFILES_SIZE_ON_DISK：Stable 层 DMFile 中的数据在磁盘占据的空间大小（单位：字节）。
- STABLE_DMFILES_PACKS：Stable 层 DMFile 中包含的 Pack 数量。
- TIFLASH_INSTANCE：TiFlash 实例地址。

14.11.18.2.48 TIFLASH_TABLES

> **警告：**
>
> 请不要在生产环境中使用该系统表，因为该表字段信息尚未固定下来，可能会在 TiDB 未来版本中发生变化。

TIFLASH_TABLES 表提供了 TiFlash 内部数据表的统计信息。

```
USE information_schema;
DESC tiflash_tables;
```

```
+-----------------------------------------+--------------+------+------+---------+-------+
| Field                                   | Type         | Null | Key  | Default | Extra |
+-----------------------------------------+--------------+------+------+---------+-------+
| DATABASE                                | varchar(64)  | YES  |      | NULL    |       |
| TABLE                                   | varchar(64)  | YES  |      | NULL    |       |
| TIDB_DATABASE                           | varchar(64)  | YES  |      | NULL    |       |
| TIDB_TABLE                              | varchar(64)  | YES  |      | NULL    |       |
| TABLE_ID                                | bigint(64)   | YES  |      | NULL    |       |
| IS_TOMBSTONE                            | bigint(64)   | YES  |      | NULL    |       |
| SEGMENT_COUNT                           | bigint(64)   | YES  |      | NULL    |       |
| TOTAL_ROWS                              | bigint(64)   | YES  |      | NULL    |       |
| TOTAL_SIZE                              | bigint(64)   | YES  |      | NULL    |       |
| TOTAL_DELETE_RANGES                     | bigint(64)   | YES  |      | NULL    |       |
| DELTA_RATE_ROWS                         | double       | YES  |      | NULL    |       |
| DELTA_RATE_SEGMENTS                     | double       | YES  |      | NULL    |       |
| DELTA_PLACED_RATE                       | double       | YES  |      | NULL    |       |
| DELTA_CACHE_SIZE                        | bigint(64)   | YES  |      | NULL    |       |
| DELTA_CACHE_RATE                        | double       | YES  |      | NULL    |       |
| DELTA_CACHE_WASTED_RATE                 | double       | YES  |      | NULL    |       |
| DELTA_INDEX_SIZE                        | bigint(64)   | YES  |      | NULL    |       |
| AVG_SEGMENT_ROWS                        | double       | YES  |      | NULL    |       |
| AVG_SEGMENT_SIZE                        | double       | YES  |      | NULL    |       |
| DELTA_COUNT                             | bigint(64)   | YES  |      | NULL    |       |
| TOTAL_DELTA_ROWS                        | bigint(64)   | YES  |      | NULL    |       |
| TOTAL_DELTA_SIZE                        | bigint(64)   | YES  |      | NULL    |       |
| AVG_DELTA_ROWS                          | double       | YES  |      | NULL    |       |
| AVG_DELTA_SIZE                          | double       | YES  |      | NULL    |       |
| AVG_DELTA_DELETE_RANGES                 | double       | YES  |      | NULL    |       |
| STABLE_COUNT                            | bigint(64)   | YES  |      | NULL    |       |
| TOTAL_STABLE_ROWS                       | bigint(64)   | YES  |      | NULL    |       |
| TOTAL_STABLE_SIZE                       | bigint(64)   | YES  |      | NULL    |       |
| TOTAL_STABLE_SIZE_ON_DISK               | bigint(64)   | YES  |      | NULL    |       |
| AVG_STABLE_ROWS                         | double       | YES  |      | NULL    |       |
| AVG_STABLE_SIZE                         | double       | YES  |      | NULL    |       |
| TOTAL_PACK_COUNT_IN_DELTA               | bigint(64)   | YES  |      | NULL    |       |
| MAX_PACK_COUNT_IN_DELTA                 | bigint(64)   | YES  |      | NULL    |       |
| AVG_PACK_COUNT_IN_DELTA                 | double       | YES  |      | NULL    |       |
| AVG_PACK_ROWS_IN_DELTA                  | double       | YES  |      | NULL    |       |
| AVG_PACK_SIZE_IN_DELTA                  | double       | YES  |      | NULL    |       |
| TOTAL_PACK_COUNT_IN_STABLE              | bigint(64)   | YES  |      | NULL    |       |
| AVG_PACK_COUNT_IN_STABLE                | double       | YES  |      | NULL    |       |
| AVG_PACK_ROWS_IN_STABLE                 | double       | YES  |      | NULL    |       |
| AVG_PACK_SIZE_IN_STABLE                 | double       | YES  |      | NULL    |       |
| STORAGE_STABLE_NUM_SNAPSHOTS            | bigint(64)   | YES  |      | NULL    |       |
```

```
| STORAGE_STABLE_OLDEST_SNAPSHOT_LIFETIME    | double       | YES  |      | NULL    |       |
| STORAGE_STABLE_OLDEST_SNAPSHOT_THREAD_ID   | bigint(64)   | YES  |      | NULL    |       |
| STORAGE_STABLE_OLDEST_SNAPSHOT_TRACING_ID | varchar(128) | YES  |      | NULL    |       |
| STORAGE_DELTA_NUM_SNAPSHOTS                | bigint(64)   | YES  |      | NULL    |       |
| STORAGE_DELTA_OLDEST_SNAPSHOT_LIFETIME     | double       | YES  |      | NULL    |       |
| STORAGE_DELTA_OLDEST_SNAPSHOT_THREAD_ID    | bigint(64)   | YES  |      | NULL    |       |
| STORAGE_DELTA_OLDEST_SNAPSHOT_TRACING_ID   | varchar(128) | YES  |      | NULL    |       |
| STORAGE_META_NUM_SNAPSHOTS                 | bigint(64)   | YES  |      | NULL    |       |
| STORAGE_META_OLDEST_SNAPSHOT_LIFETIME      | double       | YES  |      | NULL    |       |
| STORAGE_META_OLDEST_SNAPSHOT_THREAD_ID     | bigint(64)   | YES  |      | NULL    |       |
| STORAGE_META_OLDEST_SNAPSHOT_TRACING_ID    | varchar(128) | YES  |      | NULL    |       |
| BACKGROUND_TASKS_LENGTH                    | bigint(64)   | YES  |      | NULL    |       |
| TIFLASH_INSTANCE                           | varchar(64)  | YES  |      | NULL    |       |
+--------------------------------------------+--------------+------+------+---------+-------+
54 rows in set (0.00 sec)
```

TIFLASH_TABLES 表中各列的字段含义如下：

- DATABASE：表在 TiFlash 内部所属的数据库的名称。
- TABLE：表在 TiFlash 内部的名称。
- TIDB_DATABASE：表所属的数据库的名称。
- TIDB_TABLE：表的名称。
- TABLE_ID：表的内部 ID，该 ID 在一个 TiDB 集群内部唯一。
- IS_TOMBSTONE：表示该表是否处于待回收状态，为 1 表示该表处于待回收状态，为 0 则表示处于正常状态。
- SEGMENT_COUNT：表的 Segment 数量，其中 Segment 为 TiFlash 内部管理数据的单元。
- TOTAL_ROWS：表的总行数。
- TOTAL_SIZE：表的总大小（单位：字节）。
- TOTAL_DELETE_RANGES：表当前 Delete Range 的数量。
- DELTA_RATE_ROWS：表 Delta 层数据总行数占该表总行数的比例。
- DELTA_RATE_SEGMENTS：表所有 Segment 中包含非空 Delta 层的 Segment 的比例。
- DELTA_PLACED_RATE：表 Delta 层数据行中已经完成 Index 构建的比例。
- DELTA_CACHE_SIZE：表 Delta 层缓存大小（单位：字节）。
- DELTA_CACHE_RATE：表 Delta 层数据的缓存比例。
- DELTA_CACHE_WASTED_RATE：表 Delta 层缓存中无效数据的比例。
- DELTA_INDEX_SIZE：Delta Index 占用的内存大小（单位：字节）。
- AVG_SEGMENT_ROWS：表所有 Segment 的平均行数。
- AVG_SEGMENT_SIZE：表所有 Segment 的平均大小（单位：字节）。
- DELTA_COUNT：表中包含 Delta 层的 Segment 的个数。
- TOTAL_DELTA_ROWS：所有 Delta 层的数据总行数。
- TOTAL_DELTA_SIZE：所有 Delta 层的数据总大小（单位：字节）。
- AVG_DELTA_ROWS：所有 Delta 层的平均数据行数。
- AVG_DELTA_SIZE：所有 Delta 层的平均数据大小（单位：字节）。
- AVG_DELTA_DELETE_RANGES：所有 Delta 层的平均 Delete Range 操作个数。
- STABLE_COUNT：表中包含非空 Stable 层的 Segment 的个数。

- TOTAL_STABLE_ROWS：所有 Stable 层的数据总行数。
- TOTAL_STABLE_SIZE：所有 Stable 层的数据总大小（单位：字节）。
- TOTAL_STABLE_SIZE_ON_DISK：所有 Stable 层的数据在磁盘占据的空间大小（单位：字节）。
- AVG_STABLE_ROWS：所有 Stable 层的平均数据行数。
- AVG_STABLE_SIZE：所有 Stable 层的平均数据大小（单位：字节）。
- TOTAL_PACK_COUNT_IN_DELTA：所有 Delta 层的 Column File 总数量。
- MAX_PACK_COUNT_IN_DELTA：单个 Delta 层包含 Column File 数量的最大值。
- AVG_PACK_COUNT_IN_DELTA：所有 Delta 层的平均 Column File 数量。
- AVG_PACK_ROWS_IN_DELTA：Delta 层所有 Column File 的平均数据行数。
- AVG_PACK_SIZE_IN_DELTA：Delta 层所有 Column File 的平均数据大小（单位：字节）。
- TOTAL_PACK_COUNT_IN_STABLE：所有 Stable 层的 Pack 总数量。
- AVG_PACK_COUNT_IN_STABLE：所有 Stable 层的平均 Pack 数量。
- AVG_PACK_ROWS_IN_STABLE：Stable 层所有 Pack 的平均数据行数。
- AVG_PACK_SIZE_IN_STABLE：Stable 层所有 Pack 的平均数据大小（单位：字节）。
- STORAGE_STABLE_NUM_SNAPSHOTS：Stable 层 Snapshot 数量。
- STORAGE_STABLE_OLDEST_SNAPSHOT_LIFETIME：Stable 层最早 Snapshot 的持续时间（单位：秒）。
- STORAGE_STABLE_OLDEST_SNAPSHOT_THREAD_ID：Stable 层最早 Snapshot 对应的线程 ID。
- STORAGE_STABLE_OLDEST_SNAPSHOT_TRACING_ID：Stable 层最早 Snapshot 对应的跟踪 ID。
- STORAGE_DELTA_NUM_SNAPSHOTS：Delta 层 Snapshot 数量。
- STORAGE_DELTA_OLDEST_SNAPSHOT_LIFETIME：Delta 层最早 Snapshot 的持续时间（单位：秒）。
- STORAGE_DELTA_OLDEST_SNAPSHOT_THREAD_ID：Delta 层最早 Snapshot 对应的线程 ID。
- STORAGE_DELTA_OLDEST_SNAPSHOT_TRACING_ID：Delta 层最早 Snapshot 对应的跟踪 ID。
- STORAGE_META_NUM_SNAPSHOTS：Meta 元信息 Snapshot 数量。
- STORAGE_META_OLDEST_SNAPSHOT_LIFETIME：Meta 元信息最早 Snapshot 的持续时间。（单位：秒）
- STORAGE_META_OLDEST_SNAPSHOT_THREAD_ID：Meta 元信息最早 Snapshot 对应的线程 ID。
- STORAGE_META_OLDEST_SNAPSHOT_TRACING_ID：Meta 元信息最早 Snapshot 对应的跟踪 ID。
- BACKGROUND_TASKS_LENGTH：后台任务队列长度。
- TIFLASH_INSTANCE：TiFlash 实例地址。

14.11.18.2.49 TIKV_REGION_PEERS

TIKV_REGION_PEERS 表提供了 TiKV 中单个 Region 节点的详细信息，比如它是一个 learner 还是一个 leader。

```
USE INFORMATION_SCHEMA;
DESC TIKV_REGION_PEERS;
```

输出结果如下：

```
+--------------+-------------+------+------+---------+-------+
| Field        | Type        | Null | Key  | Default | Extra |
+--------------+-------------+------+------+---------+-------+
| REGION_ID    | bigint(21)  | YES  |      | NULL    |       |
| PEER_ID      | bigint(21)  | YES  |      | NULL    |       |
| STORE_ID     | bigint(21)  | YES  |      | NULL    |       |
| IS_LEARNER   | tinyint(1)  | NO   |      | 0       |       |
| IS_LEADER    | tinyint(1)  | NO   |      | 0       |       |
| STATUS       | varchar(10) | YES  |      | 0       |       |
```

```
| DOWN_SECONDS | bigint(21)  | YES  |      | 0       |       |
+--------------+-------------+------+------+---------+-------+
7 rows in set (0.01 sec)
```

例如，使用以下 SQL 语句，你可以查询 WRITTEN_BYTES 最大的前 3 个 Region 所在的 TiKV 地址：

```
SELECT
  address,
  tikv.address,
  region.region_id
FROM
  TIKV_STORE_STATUS tikv,
  TIKV_REGION_PEERS peer,
  (SELECT * FROM tikv_region_status ORDER BY written_bytes DESC LIMIT 3) region
WHERE
  region.region_id = peer.region_id
  AND peer.is_leader = 1
  AND peer.store_id = tikv.store_id;
```

TIKV_REGION_PEERS 表各列含义如下：

- REGION_ID：REGION 的 ID。
- PEER_ID：REGION 中对应的副本 Peer 的 ID。
- STORE_ID：REGION 所在 TiKV Store 的 ID。
- IS_LEARNER：Peer 是否是 LEARNER。
- IS_LEADER：Peer 是否是 LEADER。
- STATUS：Peer 的状态，一共有 3 种状态：
  - PENDING：暂时不可用状态。
  - DOWN：下线转态，该 Peer 不再提供服务。
  - NORMAL：正常状态。
- DOWN_SECONDS：处于下线状态的时间，单位是秒。

14.11.18.2.50 TIKV_REGION_STATUS

TIKV_REGION_STATUS 表通过 PD 的 API 展示 TiKV Region 的基本信息，比如 Region ID、开始和结束键值以及读写流量。

```
USE information_schema;
DESC tikv_region_status;
```

```
+---------------------------+-------------+------+------+---------+-------+
| Field                     | Type        | Null | Key  | Default | Extra |
+---------------------------+-------------+------+------+---------+-------+
| REGION_ID                 | bigint(21)  | YES  |      | NULL    |       |
| START_KEY                 | text        | YES  |      | NULL    |       |
```

```
| END_KEY                    | text        | YES  |      | NULL    |       |
| TABLE_ID                   | bigint(21)  | YES  |      | NULL    |       |
| DB_NAME                    | varchar(64) | YES  |      | NULL    |       |
| TABLE_NAME                 | varchar(64) | YES  |      | NULL    |       |
| IS_INDEX                   | tinyint(1)  | NO   |      | 0       |       |
| INDEX_ID                   | bigint(21)  | YES  |      | NULL    |       |
| INDEX_NAME                 | varchar(64) | YES  |      | NULL    |       |
| EPOCH_CONF_VER             | bigint(21)  | YES  |      | NULL    |       |
| EPOCH_VERSION              | bigint(21)  | YES  |      | NULL    |       |
| WRITTEN_BYTES              | bigint(21)  | YES  |      | NULL    |       |
| READ_BYTES                 | bigint(21)  | YES  |      | NULL    |       |
| APPROXIMATE_SIZE           | bigint(21)  | YES  |      | NULL    |       |
| APPROXIMATE_KEYS           | bigint(21)  | YES  |      | NULL    |       |
| REPLICATIONSTATUS_STATE    | varchar(64) | YES  |      | NULL    |       |
| REPLICATIONSTATUS_STATEID | bigint(21)  | YES  |      | NULL    |       |
+----------------------------+-------------+------+------+---------+-------+
17 rows in set (0.00 sec)
```

TIKV_REGION_STATUS 表中列的含义如下：

- REGION_ID：Region 的 ID。
- START_KEY：Region 的起始 key 的值。
- END_KEY：Region 的末尾 key 的值。
- TABLE_ID：Region 所属的表的 ID。
- DB_NAME：TABLE_ID 所属的数据库的名称。
- TABLE_NAME：Region 所属的表的名称。
- IS_INDEX：Region 数据是否是索引，0 代表不是索引，1 代表是索引。如果当前 Region 同时包含表数据和索引数据，会有多行记录，IS_INDEX 分别是 0 和 1。
- INDEX_ID：Region 所属的索引的 ID。如果 IS_INDEX 为 0，这一列的值就为 NULL。
- INDEX_NAME：Region 所属的索引的名称。如果 IS_INDEX 为 0，这一列的值就为 NULL。
- EPOCH_CONF_VER：Region 的配置的版本号，在增加或减少 peer 时版本号会递增。
- EPOCH_VERSION：Region 的当前版本号，在分裂或合并时版本号会递增。
- WRITTEN_BYTES：已经往 Region 写入的数据量 (bytes)。
- READ_BYTES：已经从 Region 读取的数据量 (bytes)。
- APPROXIMATE_SIZE：Region 的近似数据量 (MB)。
- APPROXIMATE_KEYS：Region 中 key 的近似数量。
- REPLICATIONSTATUS_STATE:Region 当前的同步状态,可能为 UNKNOWN/SIMPLE_MAJORITY/INTEGRITY_OVER_LABEL ↪ 其中一种状态。
- REPLICATIONSTATUS_STATEID：REPLICATIONSTATUS_STATE 对应的标识符。

通过在 EPOCH_CONF_VER、WRITTEN_BYTES 和 READ_BYTES 列上执行 ORDER BY X LIMIT Y 操作，你可以在 pd-ctl 中实现 top confver、top read 和 top write 操作。

你可以使用以下 SQL 语句查询写入数据最多的前 3 个 Region：

```
SELECT * FROM tikv_region_status ORDER BY written_bytes DESC LIMIT 3;
```

14.11.18.2.51 TIKV_STORE_STATUS

TIKV_STORE_STATUS 表通过 PD 的 API 显示了 TiKV 节点的一些基本信息，例如在集群中分配的 ID、地址和端口、状态、容量以及当前节点的 Region leader 的数量。

```
USE INFORMATION_SCHEMA;
DESC TIKV_STORE_STATUS;
```

输出结果如下：

```
+------------------+-------------+------+------+---------+-------+
| Field            | Type        | Null | Key  | Default | Extra |
+------------------+-------------+------+------+---------+-------+
| STORE_ID         | bigint(21)  | YES  |      | NULL    |       |
| ADDRESS          | varchar(64) | YES  |      | NULL    |       |
| STORE_STATE      | bigint(21)  | YES  |      | NULL    |       |
| STORE_STATE_NAME | varchar(64) | YES  |      | NULL    |       |
| LABEL            | json        | YES  |      | NULL    |       |
| VERSION          | varchar(64) | YES  |      | NULL    |       |
| CAPACITY         | varchar(64) | YES  |      | NULL    |       |
| AVAILABLE        | varchar(64) | YES  |      | NULL    |       |
| LEADER_COUNT     | bigint(21)  | YES  |      | NULL    |       |
| LEADER_WEIGHT    | double      | YES  |      | NULL    |       |
| LEADER_SCORE     | double      | YES  |      | NULL    |       |
| LEADER_SIZE      | bigint(21)  | YES  |      | NULL    |       |
| REGION_COUNT     | bigint(21)  | YES  |      | NULL    |       |
| REGION_WEIGHT    | double      | YES  |      | NULL    |       |
| REGION_SCORE     | double      | YES  |      | NULL    |       |
| REGION_SIZE      | bigint(21)  | YES  |      | NULL    |       |
| START_TS         | datetime    | YES  |      | NULL    |       |
| LAST_HEARTBEAT_TS | datetime   | YES  |      | NULL    |       |
| UPTIME           | varchar(64) | YES  |      | NULL    |       |
+------------------+-------------+------+------+---------+-------+
19 rows in set (0.00 sec)
```

TIKV_STORE_STATUS 表中列的含义如下：

- STORE_ID：Store 的 ID。
- ADDRESS：Store 的地址。
- STORE_STATE：Store 状态的标识符，与 STORE_STATE_NAME 相对应。
- STORE_STATE_NAME：Store 状态的名字，为 Up、Offline、Tombstone 中的一种。
- LABEL：给 Store 设置的标签。
- VERSION：Store 的版本号。
- CAPACITY：Store 的存储容量。
- AVAILABLE：Store 的剩余存储空间。
- LEADER_COUNT：Store 上的 leader 的数量。
- LEADER_WEIGHT：Store 的 leader 权重。

- LEADER_SCORE：Store 的 leader 评分。
- LEADER_SIZE：Store 上的所有 leader 的近似总数据量 (MB)。
- REGION_COUNT：Store 上的 Region 总数。
- REGION_WEIGHT：Store 的 Region 权重。
- REGION_SCORE：Store 的 Region 评分。
- REGION_SIZE：Store 上的所有 Region 的近似总数据量 (MB)。
- START_TS：Store 启动时的时间戳。
- LAST_HEARTBEAT_TS：Store 上次发出心跳的时间戳。
- UPTIME：Store 启动以来的总时间。

14.11.18.2.52 USER_ATTRIBUTES

USER_ATTRIBUTES 表提供了用户的注释和属性。该表的数据根据 mysql.user 系统表生成。

```
USE information_schema;
DESC user_attributes;
```

```
+-----------+--------------+------+------+---------+-------+
| Field     | Type         | Null | Key  | Default | Extra |
+-----------+--------------+------+------+---------+-------+
| USER      | varchar(32)  | NO   |      | NULL    |       |
| HOST      | varchar(255) | NO   |      | NULL    |       |
| ATTRIBUTE | longtext     | YES  |      | NULL    |       |
+-----------+--------------+------+------+---------+-------+
3 rows in set (0.00 sec)
```

USER_ATTRIBUTES 表中列的含义如下：

- USER：用户名。
- HOST：用户可用于连接的主机。百分号（%）表示主机名不受限制。
- ATTRIBUTE：通过CREATE USER 或ALTER USER 语句设置的用户相关的注释和属性。

示例：

```
CREATE USER testuser1 COMMENT 'This user is created only for test';
CREATE USER testuser2 ATTRIBUTE '{"email": "user@pingcap.com"}';
SELECT * FROM information_schema.user_attributes;
```

```
+-----------+------+-----------------------------------------------------+
| USER      | HOST | ATTRIBUTE                                           |
+-----------+------+-----------------------------------------------------+
| root      | %    | NULL                                                |
| testuser1 | %    | {"comment": "This user is created only for test"} |
| testuser2 | %    | {"email": "user@pingcap.com"}                     |
+-----------+------+-----------------------------------------------------+
3 rows in set (0.00 sec)
```

14.11.18.2.53 USER_PRIVILEGES

USER_PRIVILEGES 表提供了关于全局权限的信息。该表的数据根据 mysql.user 系统表生成。

```
USE INFORMATION_SCHEMA;
DESC USER_PRIVILEGES;
```

输出结果如下：

```
+----------------+--------------+------+------+---------+-------+
| Field          | Type         | Null | Key  | Default | Extra |
+----------------+--------------+------+------+---------+-------+
| GRANTEE        | varchar(81)  | YES  |      | NULL    |       |
| TABLE_CATALOG  | varchar(512) | YES  |      | NULL    |       |
| PRIVILEGE_TYPE | varchar(64)  | YES  |      | NULL    |       |
| IS_GRANTABLE   | varchar(3)   | YES  |      | NULL    |       |
+----------------+--------------+------+------+---------+-------+
4 rows in set (0.00 sec)
```

查看 USER_PRIVILEGES 表的信息：

```
SELECT * FROM USER_PRIVILEGES;
```

输出结果如下：

```
+------------+---------------+-------------------------+--------------+
| GRANTEE    | TABLE_CATALOG | PRIVILEGE_TYPE          | IS_GRANTABLE |
+------------+---------------+-------------------------+--------------+
| 'root'@'%' | def           | SELECT                  | YES          |
| 'root'@'%' | def           | INSERT                  | YES          |
| 'root'@'%' | def           | UPDATE                  | YES          |
| 'root'@'%' | def           | DELETE                  | YES          |
| 'root'@'%' | def           | CREATE                  | YES          |
| 'root'@'%' | def           | DROP                    | YES          |
| 'root'@'%' | def           | PROCESS                 | YES          |
| 'root'@'%' | def           | REFERENCES              | YES          |
| 'root'@'%' | def           | ALTER                   | YES          |
| 'root'@'%' | def           | SHOW DATABASES          | YES          |
| 'root'@'%' | def           | SUPER                   | YES          |
| 'root'@'%' | def           | EXECUTE                 | YES          |
| 'root'@'%' | def           | INDEX                   | YES          |
| 'root'@'%' | def           | CREATE USER             | YES          |
| 'root'@'%' | def           | CREATE TABLESPACE       | YES          |
| 'root'@'%' | def           | TRIGGER                 | YES          |
| 'root'@'%' | def           | CREATE VIEW             | YES          |
| 'root'@'%' | def           | SHOW VIEW               | YES          |
| 'root'@'%' | def           | CREATE ROLE             | YES          |
| 'root'@'%' | def           | DROP ROLE               | YES          |
```

```
| 'root'@'%' | def           | CREATE TEMPORARY TABLES | YES          |
| 'root'@'%' | def           | LOCK TABLES             | YES          |
| 'root'@'%' | def           | CREATE ROUTINE          | YES          |
| 'root'@'%' | def           | ALTER ROUTINE           | YES          |
| 'root'@'%' | def           | EVENT                   | YES          |
| 'root'@'%' | def           | SHUTDOWN                | YES          |
| 'root'@'%' | def           | RELOAD                  | YES          |
| 'root'@'%' | def           | FILE                    | YES          |
| 'root'@'%' | def           | CONFIG                  | YES          |
| 'root'@'%' | def           | REPLICATION CLIENT      | YES          |
| 'root'@'%' | def           | REPLICATION SLAVE       | YES          |
+------------+---------------+-------------------------+--------------+
31 rows in set (0.00 sec)
```

USER_PRIVILEGES 表中列的含义如下：

- GRANTEE：被授权的用户名称，格式为 'user_name'@'host_name'。
- TABLE_CATALOG：表所属的目录的名称。该值始终为 def。
- PRIVILEGE_TYPE：被授权的权限类型，每行只列一个权限。
- IS_GRANTABLE：如果用户有 GRANT OPTION 的权限，则为 YES，否则为 NO。

14.11.18.2.54 VARIABLES_INFO

VARIABLES_INFO 可用于查看当前 TiDB 集群或实例的系统变量默认值、当前值以及作用域等信息。

```
USE information_schema;
DESC variables_info;
```

```
+-----------------+---------------------+------+------+---------+-------+
| Field           | Type                | Null | Key  | Default | Extra |
+-----------------+---------------------+------+------+---------+-------+
| VARIABLE_NAME   | varchar(64)         | NO   |      | NULL    |       |
| VARIABLE_SCOPE  | varchar(64)         | NO   |      | NULL    |       |
| DEFAULT_VALUE   | varchar(64)         | NO   |      | NULL    |       |
| CURRENT_VALUE   | varchar(64)         | NO   |      | NULL    |       |
| MIN_VALUE       | bigint(64)          | YES  |      | NULL    |       |
| MAX_VALUE       | bigint(64) unsigned | YES  |      | NULL    |       |
| POSSIBLE_VALUES | varchar(256)        | YES  |      | NULL    |       |
| IS_NOOP         | varchar(64)         | NO   |      | NULL    |       |
+-----------------+---------------------+------+------+---------+-------+
8 rows in set (0.00 sec)
```

```
SELECT * FROM variables_info ORDER BY variable_name LIMIT 3;
```

```
+--
    ↪ --------------------------------+---------------+--------------+--------------+-----------+----------
    ↪
| VARIABLE_NAME                      | VARIABLE_SCOPE | DEFAULT_VALUE | CURRENT_VALUE | MIN_VALUE
    ↪ | MAX_VALUE | POSSIBLE_VALUES | IS_NOOP |
+--
    ↪ --------------------------------+---------------+--------------+--------------+-----------+----------
    ↪
| allow_auto_random_explicit_insert | SESSION,GLOBAL | OFF           | OFF           |      NULL
    ↪ |      NULL | NULL             | NO      |
| auto_increment_increment          | SESSION,GLOBAL | 1             | 1             |         1
    ↪ |     65535 | NULL             | NO      |
| auto_increment_offset             | SESSION,GLOBAL | 1             | 1             |         1
    ↪ |     65535 | NULL             | NO      |
+--
    ↪ --------------------------------+---------------+--------------+--------------+-----------+----------
    ↪
3 rows in set (0.01 sec)
```

VARIABLES_INFO 表中列的含义如下：

- VARIABLE_NAME：系统变量名称。
- VARIABLE_SCOPE：系统变量的作用域。SESSION 表示当前 session 可见；INSTANCE 表示当前 TiDB 实例可见；GLOBAL 表示集群内可见。
- DEFAULT_VALUE：系统变量的默认值。
- CURRENT_VALUE：系统变量的当前值。如果应用范围中包含 SESSION，则显示当前 session 的值。
- MIN_VALUE：数值类型的系统变量允许的最小值。如果变量值为非数值类型，则为 NULL。
- MAX_VALUE：数值类型的系统变量允许的最大值。如果变量值为非数值类型，则为 NULL。
- POSSIBLE_VALUES：系统变量所有可能的值。如果变量值不可枚举，则为 NULL。
- IS_NOOP：是否为 noop 的系统变量。

14.11.18.2.55 VIEWS

VIEWS 表提供了关于 SQL 视图的信息。

```
USE INFORMATION_SCHEMA;
DESC VIEWS;
```

输出结果如下：

```
+----------------------+--------------+------+------+---------+-------+
| Field                | Type         | Null | Key  | Default | Extra |
+----------------------+--------------+------+------+---------+-------+
| TABLE_CATALOG        | varchar(512) | NO   |      | NULL    |       |
| TABLE_SCHEMA         | varchar(64)  | NO   |      | NULL    |       |
| TABLE_NAME           | varchar(64)  | NO   |      | NULL    |       |
```

```
| VIEW_DEFINITION      | longtext     | NO   |     | NULL    |       |
| CHECK_OPTION         | varchar(8)   | NO   |     | NULL    |       |
| IS_UPDATABLE         | varchar(3)   | NO   |     | NULL    |       |
| DEFINER              | varchar(77)  | NO   |     | NULL    |       |
| SECURITY_TYPE        | varchar(7)   | NO   |     | NULL    |       |
| CHARACTER_SET_CLIENT | varchar(32)  | NO   |     | NULL    |       |
| COLLATION_CONNECTION | varchar(32)  | NO   |     | NULL    |       |
+----------------------+--------------+------+-----+---------+-------+
10 rows in set (0.00 sec)
```

创建视图并查询 VIEWS 表信息：

```
CREATE VIEW test.v1 AS SELECT 1;
SELECT * FROM VIEWS\G
```

输出结果如下：

```
*************************** 1. row ***************************
       TABLE_CATALOG: def
        TABLE_SCHEMA: test
          TABLE_NAME: v1
     VIEW_DEFINITION: SELECT 1
        CHECK_OPTION: CASCADED
        IS_UPDATABLE: NO
             DEFINER: root@127.0.0.1
       SECURITY_TYPE: DEFINER
CHARACTER_SET_CLIENT: utf8mb4
COLLATION_CONNECTION: utf8mb4_0900_ai_ci
1 row in set (0.00 sec)
```

VIEWS 表中列的含义如下：

- TABLE_CATALOG：视图所属的目录的名称。该值始终为 def。
- TABLE_SCHEMA：视图所属的数据库的名称。
- TABLE_NAME：视图名称。
- VIEW_DEFINITION：视图的定义，由创建视图时 SELECT 部分的语句组成。
- CHECK_OPTION：CHECK_OPTION 的值。取值为 NONE、CASCADE 或 LOCAL。
- IS_UPDATABLE：UPDATE/INSERT/DELETE 是否对该视图可用。在 TiDB，始终为 NO。
- DEFINER：视图的创建者用户名称，格式为 'user_name'@'host_name'。
- SECURITY_TYPE：SQL SECURITY 的值，取值为 DEFINER 或 INVOKER。
- CHARACTER_SET_CLIENT：在视图创建时 session 变量 character_set_client 的值。
- COLLATION_CONNECTION：在视图创建时 session 变量 collation_connection 的值。

14.11.18.3 Metrics Schema

METRICS_SCHEMA 是基于 Prometheus 中 TiDB 监控指标的一组视图。每个表的 PromQL（Prometheus 查询语言）的源均可在INFORMATION_SCHEMA.METRICS_TABLES 表中找到。

```
use metrics_schema;
SELECT * FROM uptime;
SELECT * FROM information_schema.metrics_tables WHERE table_name='uptime'\G
```

```
+----------------------------+-----------------+------------+--------------------+
| time                       | instance        | job        | value              |
+----------------------------+-----------------+------------+--------------------+
| 2020-07-06 15:26:26.203000 | 127.0.0.1:10080 | tidb       | 123.60300016403198 |
| 2020-07-06 15:27:26.203000 | 127.0.0.1:10080 | tidb       | 183.60300016403198 |
| 2020-07-06 15:26:26.203000 | 127.0.0.1:20180 | tikv       | 123.60300016403198 |
| 2020-07-06 15:27:26.203000 | 127.0.0.1:20180 | tikv       | 183.60300016403198 |
| 2020-07-06 15:26:26.203000 | 127.0.0.1:2379  | pd         | 123.60300016403198 |
| 2020-07-06 15:27:26.203000 | 127.0.0.1:2379  | pd         | 183.60300016403198 |
| 2020-07-06 15:26:26.203000 | 127.0.0.1:9090  | prometheus | 123.72300004959106 |
| 2020-07-06 15:27:26.203000 | 127.0.0.1:9090  | prometheus | 183.72300004959106 |
+----------------------------+-----------------+------------+--------------------+
8 rows in set (0.00 sec)
*************************** 1. row ***************************
TABLE_NAME: uptime
    PROMQL: (time() - process_start_time_seconds{$LABEL_CONDITIONS})
    LABELS: instance,job
  QUANTILE: 0
   COMMENT: TiDB uptime since last restart(second)
1 row in set (0.00 sec)
```

```
show tables;
```

```
+-------------------------------------------------+
| Tables_in_metrics_schema                        |
+-------------------------------------------------+
| abnormal_stores                                 |
| etcd_disk_wal_fsync_rate                        |
| etcd_wal_fsync_duration                         |
| etcd_wal_fsync_total_count                      |
| etcd_wal_fsync_total_time                       |
| go_gc_count                                     |
| go_gc_cpu_usage                                 |
| go_gc_duration                                  |
| go_heap_mem_usage                               |
| go_threads                                      |
| goroutines_count                                |
| node_cpu_usage                                  |
| node_disk_available_size                        |
| node_disk_io_util                               |
```

```
| node_disk_iops                                    |
| node_disk_read_latency                            |
| node_disk_size                                    |
..
| tikv_storage_async_request_total_time             |
| tikv_storage_async_requests                       |
| tikv_storage_async_requests_total_count           |
| tikv_storage_command_ops                          |
| tikv_store_size                                   |
| tikv_thread_cpu                                   |
| tikv_thread_nonvoluntary_context_switches         |
| tikv_thread_voluntary_context_switches            |
| tikv_threads_io                                   |
| tikv_threads_state                                |
| tikv_total_keys                                   |
| tikv_wal_sync_duration                            |
| tikv_wal_sync_max_duration                        |
| tikv_worker_handled_tasks                         |
| tikv_worker_handled_tasks_total_num               |
| tikv_worker_pending_tasks                         |
| tikv_worker_pending_tasks_total_num               |
| tikv_write_stall_avg_duration                     |
| tikv_write_stall_max_duration                     |
| tikv_write_stall_reason                           |
| up                                                |
| uptime                                            |
+---------------------------------------------------+
626 rows in set (0.00 sec)
```

METRICS_SCHEMA 是监控相关的 summary 表的数据源，例如metrics_summary、metrics_summary_by_label和inspection_summary。

14.11.18.3.1 更多例子

下面以 metrics_schema 中的 tidb_query_duration 监控表为例，介绍监控表相关的使用和原理，其他的监控表原理均类似。

查询 information_schema.metrics_tables 中关于 tidb_query_duration 表相关的信息如下：

```
select * from information_schema.metrics_tables where table_name='tidb_query_duration';
```

```
+--
    ↪ -------------------+--------------------------------------------------------------------------------------------
    ↪
| TABLE_NAME           | PROMQL
    ↪
    ↪ | LABELS             | QUANTILE | COMMENT                                      |
```

```
+--
    ↪ -------------------+-----------------------------------------------------------------------------------
    ↪
| tidb_query_duration | histogram_quantile($QUANTILE, sum(rate(
    ↪ tidb_server_handle_query_duration_seconds_bucket{$LABEL_CONDITIONS}[$RANGE_DURATION])) by
    ↪  (le,sql_type,instance)) | instance,sql_type | 0.9      | The quantile of TiDB query
    ↪ durations(second) |
+--
    ↪ -------------------+-----------------------------------------------------------------------------------
    ↪
```

- TABLE_NAME：对应于 metrics_schema 中的表名，这里表名是 tidb_query_duration。
- PROMQL：因为监控表的原理是将 SQL 映射成 PromQL 后向 Prometheus 请求数据，并将 Prometheus 返回的结果转换成 SQL 查询结果。该字段是 PromQL 的表达式模板，查询监控表数据时使用查询条件改写模板中的变量，生成最终的查询表达式。
- LABELS：监控项定义的 label，tidb_query_duration 有两个 label，分别是 instance 和 sql_type。
- QUANTILE：百分位。直方图类型的监控数据会指定一个默认百分位。如果值为 0，表示该监控表对应的监控不是直方图。tidb_query_duration 默认查询 0.9，也就是 P90 的监控值。
- COMMENT：对这个监控表的解释。可以看出 tidb_query_duration 表是用来查询 TiDB query 执行的百分位时间，如 P999/P99/P90 的查询耗时，单位是秒。

再来看 tidb_query_duration 的表结构：

```
show create table metrics_schema.tidb_query_duration;
```

```
+--
    ↪ -------------------+-----------------------------------------------------------------------------------
    ↪
| Table               | Create Table
    ↪
    ↪ |
+--
    ↪ -------------------+-----------------------------------------------------------------------------------
    ↪
| tidb_query_duration | CREATE TABLE `tidb_query_duration` (
    ↪                                                                                  |
|                     |   `time` datetime unsigned DEFAULT CURRENT_TIMESTAMP,
    ↪                                                                  |
|                     |   `instance` varchar(512) DEFAULT NULL,
    ↪                                                                                |
|                     |   `sql_type` varchar(512) DEFAULT NULL,
    ↪                                                                                |
|                     |   `quantile` double unsigned DEFAULT '0.9',
    ↪                                                                          |
|                     |   `value` double unsigned DEFAULT NULL
    ↪                                                                                 |
```

```
|                     | ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin COMMENT='The
    ↪ quantile of TiDB query durations(second)' |
+--
    ↪ -------------------+-----------------------------------------------------------------------------
    ↪
```

- time：监控项的时间。
- instance 和 sql_type：是 tidb_query_duration 这个监控项的 label。instance 表示监控的地址，sql_type 表示执行 SQL 的类似。
- quantile，百分位，直方图类型的监控都会有该列，表示查询的百分位时间，如 quantile=0.9 就是查询 P90 的时间。
- value：监控项的值。

下面是查询时间 [2020-03-25 23:40:00, 2020-03-25 23:42:00] 范围内的 P99 的 TiDB Query 耗时：

```
select * from metrics_schema.tidb_query_duration where value is not null and time>='2020-03-25
    ↪ 23:40:00' and time <= '2020-03-25 23:42:00' and quantile=0.99;
```

```
+---------------------+-------------------+----------+----------+----------------+
| time                | instance          | sql_type | quantile | value          |
+---------------------+-------------------+----------+----------+----------------+
| 2020-03-25 23:40:00 | 172.16.5.40:10089 | Insert   | 0.99     | 0.509929485256 |
| 2020-03-25 23:41:00 | 172.16.5.40:10089 | Insert   | 0.99     | 0.494690793986 |
| 2020-03-25 23:42:00 | 172.16.5.40:10089 | Insert   | 0.99     | 0.493460506934 |
| 2020-03-25 23:40:00 | 172.16.5.40:10089 | Select   | 0.99     | 0.152058493415 |
| 2020-03-25 23:41:00 | 172.16.5.40:10089 | Select   | 0.99     | 0.152193879678 |
| 2020-03-25 23:42:00 | 172.16.5.40:10089 | Select   | 0.99     | 0.140498483232 |
| 2020-03-25 23:40:00 | 172.16.5.40:10089 | internal | 0.99     | 0.47104        |
| 2020-03-25 23:41:00 | 172.16.5.40:10089 | internal | 0.99     | 0.11776        |
| 2020-03-25 23:42:00 | 172.16.5.40:10089 | internal | 0.99     | 0.11776        |
+---------------------+-------------------+----------+----------+----------------+
```

以上查询结果的第一行意思是，在 2020-03-25 23:40:00 时，在 TiDB 实例 172.16.5.40:10089 上，Insert 类型的语句的 P99 执行时间是 0.509929485256 秒。其他各行的含义类似，sql_type 列的其他值含义如下：

- Select：表示执行的 select 类型的语句。
- internal：表示 TiDB 的内部 SQL 语句，一般是统计信息更新，获取全局变量相关的内部语句。

进一步再查看上面语句的执行计划如下：

```
desc select * from metrics_schema.tidb_query_duration where value is not null and time>='
    ↪ 2020-03-25 23:40:00' and time <= '2020-03-25 23:42:00' and quantile=0.99;
```

```
+--
    ↪ ---------------+----------+------+-------------------------+-------------------------------------
    ↪
```

```
| id                | estRows   | task | access object              | operator info
    ↪
    ↪ |
+--
    ↪ ----------------+----------+------+----------------------------+-----------------------------------------
    ↪
| Selection_5       | 8000.00   | root |                            | not(isnull(Column#5))
    ↪
    ↪ |
| └─MemTableScan_6 | 10000.00 | root | table:tidb_query_duration | PromQL:histogram_quantile
    ↪ (0.99, sum(rate(tidb_server_handle_query_duration_seconds_bucket{}[60s])) by (le,sql_type
    ↪ ,instance)), start_time:2020-03-25 23:40:00, end_time:2020-03-25 23:42:00, step:1m0s |
+--
    ↪ ----------------+----------+------+----------------------------+-----------------------------------------
    ↪
```

可以发现执行计划中有一个 PromQL，以及查询监控的 start_time 和 end_time，还有 step 值，在实际执行时，TiDB 会调用 Prometheus 的 query_range HTTP API 接口来查询监控数据。

从以上结果可知，在 [2020-03-25 23:40:00, 2020-03-25 23:42:00] 时间范围内，每个 label 只有三个时间的值，间隔时间是 1 分钟，即执行计划中的 step 值。该间隔时间由以下两个 session 变量决定：

- tidb_metric_query_step：查询的分辨率步长。从 Prometheus 的 query_range 接口查询数据时需要指定 start_time，end_time 和 step，其中 step 会使用该变量的值。
- tidb_metric_query_range_duration：查询监控时，会将 PROMQL 中的 $RANGE_DURATION 替换成该变量的值，默认值是 60 秒。

如果想要查看不同时间粒度的监控项的值，用户可以修改上面两个 session 变量后查询监控表，示例如下：

首先修改两个 session 变量的值，将时间粒度设置为 30 秒。

> **注意：**
>
> Prometheus 支持查询的最小粒度为 30 秒。

```
set @@tidb_metric_query_step=30;
set @@tidb_metric_query_range_duration=30;
```

再查询 tidb_query_duration 监控如下，可以发现在三分钟时间范围内，每个 label 有六个时间的值，每个值时间间隔是 30 秒。

```
select * from metrics_schema.tidb_query_duration where value is not null and time>='2020-03-25
    ↪ 23:40:00' and time <= '2020-03-25 23:42:00' and quantile=0.99;
```

```
+---------------------+------------------+----------+----------+-----------------+
| time                | instance         | sql_type | quantile | value           |
+---------------------+------------------+----------+----------+-----------------+
| 2020-03-25 23:40:00 | 172.16.5.40:10089 | Insert   | 0.99     | 0.483285651924  |
| 2020-03-25 23:40:30 | 172.16.5.40:10089 | Insert   | 0.99     | 0.484151462113  |
| 2020-03-25 23:41:00 | 172.16.5.40:10089 | Insert   | 0.99     | 0.504576        |
| 2020-03-25 23:41:30 | 172.16.5.40:10089 | Insert   | 0.99     | 0.493577384561  |
| 2020-03-25 23:42:00 | 172.16.5.40:10089 | Insert   | 0.99     | 0.49482474311   |
| 2020-03-25 23:40:00 | 172.16.5.40:10089 | Select   | 0.99     | 0.189253402185  |
| 2020-03-25 23:40:30 | 172.16.5.40:10089 | Select   | 0.99     | 0.184224951851  |
| 2020-03-25 23:41:00 | 172.16.5.40:10089 | Select   | 0.99     | 0.151673410553  |
| 2020-03-25 23:41:30 | 172.16.5.40:10089 | Select   | 0.99     | 0.127953838989  |
| 2020-03-25 23:42:00 | 172.16.5.40:10089 | Select   | 0.99     | 0.127455434547  |
| 2020-03-25 23:40:00 | 172.16.5.40:10089 | internal | 0.99     | 0.0624          |
| 2020-03-25 23:40:30 | 172.16.5.40:10089 | internal | 0.99     | 0.12416         |
| 2020-03-25 23:41:00 | 172.16.5.40:10089 | internal | 0.99     | 0.0304          |
| 2020-03-25 23:41:30 | 172.16.5.40:10089 | internal | 0.99     | 0.06272         |
| 2020-03-25 23:42:00 | 172.16.5.40:10089 | internal | 0.99     | 0.0629333333333 |
+---------------------+------------------+----------+----------+-----------------+
```

最后查看执行计划，也会发现执行计划中的 PromQL 以及 step 的值都已经变成了 30 秒。

```
desc select * from metrics_schema.tidb_query_duration where value is not null and time>='
    ↪ 2020-03-25 23:40:00' and time <= '2020-03-25 23:42:00' and quantile=0.99;
```

```
+--
    ↪ ----------------+----------+------+--------------------------+-----------------------------------------
    ↪
| id                | estRows  | task | access object            | operator info
    ↪
    ↪ |
+--
    ↪ ----------------+----------+------+--------------------------+-----------------------------------------
    ↪
| Selection_5       | 8000.00  | root |                          | not(isnull(Column#5))
    ↪
    ↪ |
| └─MemTableScan_6 | 10000.00 | root | table:tidb_query_duration | PromQL:histogram_quantile
    ↪ (0.99, sum(rate(tidb_server_handle_query_duration_seconds_bucket{}[30s])) by (le,sql_type
    ↪ ,instance)), start_time:2020-03-25 23:40:00, end_time:2020-03-25 23:42:00, step:30s |
+--
    ↪ ----------------+----------+------+--------------------------+-----------------------------------------
    ↪
```

14.11.19 元数据锁

本文介绍了 TiDB 中的元数据锁。

14.11.19.1 元数据锁的概念

在 TiDB 中，对元数据对象的更改采用的是在线异步变更算法。事务在执行时会获取开始时对应的元数据快照。如果事务执行过程中相关表上发生了元数据的更改，为了保证数据的一致性，TiDB 会返回 `Information schema is changed` 的错误，导致用户事务提交失败。

为了解决这个问题，在 TiDB v6.3.0 中，online DDL 算法中引入了元数据锁特性。通过协调表元数据变更过程中 DML 语句和 DDL 语句的优先级，让执行中的 DDL 语句等待持有旧版本元数据的 DML 语句提交，尽可能避免 DML 语句报错。

14.11.19.2 适用场景

元数据锁适用于所有的 DDL 语句，包括但不限于：

- `ADD INDEX`
- `ADD COLUMN`
- `DROP COLUMN`
- `DROP INDEX`
- `DROP PARTITION`
- `TRUNCATE TABLE`
- `EXCHANGE PARTITION`
- `CHANGE COLUMN`
- `MODIFY COLUMN`

使用元数据锁机制会给 TiDB DDL 任务的执行带来一定的性能影响。为了降低元数据锁对 DDL 任务的影响，下列场景不需要加元数据锁：

- 开启了 auto-commit 的查询语句
- 开启了 Stale Read 功能
- 访问临时表

14.11.19.3 使用元数据锁

在 v6.5.0 及之后的版本中，TiDB 默认开启元数据锁特性。当集群从 v6.5.0 之前的版本升级到 v6.5.0 及之后的版本时，TiDB 会自动开启元数据锁功能。如果需要关闭元数据锁，你可以将系统变量`tidb_enable_metadata_lock` 设置为 OFF。

14.11.19.4 元数据锁的影响

- 对于 DML 语句来说，元数据锁不会导致 DML 语句被阻塞，因此也不会存在死锁的问题。
- 开启元数据锁后，事务中某个元数据对象的元数据信息在第一次访问时确定，之后不再变化。
- 对于 DDL 语句来说，在进行元数据状态变更时，会被涉及相关元数据的旧事务所阻塞。例如以下的执行流程：

Session 1	Session 2
CREATE TABLE t (a INT);	
INSERT INTO t VALUES(1);	
BEGIN;	
	ALTER TABLE t ADD COLUMN b INT
SELECT * FROM t;（采用 t 表当前的元数据版本，返回 (a=1，b=NULL)，同时给表 t 加锁）	
	ALTER TABLE t ADD COLUMN c INT

在可重复读隔离级别下，如果从事务开始到确定一个表的元数据过程中，执行了加索引或者变更列类型等需要更改数据的 DDL，则有以下表现：

Session 1	Session 2
CREATE TABLE t (a INT);	
INSERT INTO t VALUES(1);	
BEGIN;	
	ALTER TABLE t ADD INDEX idx(a);
SELECT * FROM t;（索引 idx 不可用）	
COMMIT;	
BEGIN;	
	ALTER TABLE t MODIFY COLUMN a CHAR(10);
SELECT * FROM t;(报错 Error 8028: Information schema is changed)	

14.11.19.5 元数据锁的可观测性

TiDB v6.3.0 引入了 mysql.tidb_mdl_view 视图，可以用于查看当前阻塞的 DDL 的相关信息。

> 注意：
>
> 查询 mysql.tidb_mdl_view 视图需要有 PROCESS 权限。

下面以给表 t 添加索引为例，假设有 DDL 语句 ALTER TABLE t ADD INDEX idx(a)：

```
SELECT * FROM mysql.tidb_mdl_view\G
*************************** 1. row ***************************
    JOB_ID: 141
   DB_NAME: test
TABLE_NAME: t
     QUERY: ALTER TABLE t ADD INDEX idx(a)
SESSION ID: 2199023255957
  TxnStart: 08-30 16:35:41.313(435643624013955072)
SQL_DIGESTS: ["begin","select * from `t`"]
1 row in set (0.02 sec)
```

可以从上面的输出结果中了解到，有一个 SESSION_ID 为 2199023255957 的事务阻塞了该添加索引 DDL 的执行。该事务执行的 SQL 语句如 SQL_DIGESTS 中所示，即 ["begin","select * from `t`"]。如果想要使被阻塞的 DDL 能够继续执行，可以通过如下 Global KILL 命令中止 SESSION_ID 为 2199023255957 的事务：

```
mysql> KILL 2199023255957;
Query OK, 0 rows affected (0.00 sec)
```

中止该事务后，再次查询 mysql.tidb_mdl_view 视图。此时，查询结果不再显示上面的事务信息，说明 DDL 不再被阻塞：

```
SELECT * FROM mysql.tidb_mdl_view\G
Empty set (0.01 sec)
```

14.11.19.6 元数据锁的原理

14.11.19.6.1 问题描述

TiDB 中 DDL 操作使用的是 online DDL 模式。一个 DDL 语句在执行过程中，需要修改定义的对象元数据版本可能会进行多次小版本变更，而元数据在线异步变更的算法只论证了相邻的两个小版本之间是兼容的，即在相邻的两个元数据版本间操作，不会破坏 DDL 变更对象所存储的数据一致性。

以添加索引为例，DDL 语句的状态会经历 None -> Delete Only，Delete Only -> Write Only，Write Only -> Write Reorg，Write Reorg -> Public 这四个变化。

以下的提交流程将违反“相邻的两个小版本之间是兼容的”约束：

事务	所用的版本	集群最新版本	版本差
txn1	None	None	0
txn2	DeleteOnly	DeleteOnly	0
txn3	WriteOnly	WriteOnly	0
txn4	None	WriteOnly	2
txn5	WriteReorg	WriteReorg	0
txn6	WriteOnly	WriteReorg	1
txn7	Public	Public	0

其中 txn4 提交时采用的元数据版本与集群最新的元数据版本相差了两个版本，会影响数据正确性。

14.11.19.6.2 实现

引入元数据锁会保证整个 TiDB 集群中的所有事务所用的元数据版本最多相差一个版本。为此：

- 执行 DML 语句时，TiDB 会在事务上下文中记录该 DML 语句访问的元数据对象，例如表、视图，以及对应的元数据版本。事务提交时会清空这些记录。
- DDL 语句进行状态变更时，会向所有的 TiDB 节点推送最新版本的元数据。如果一个 TiDB 节点上所有与这次状态变更相关的事务使用的元数据版本与当前元数据版本之差小于 2，则称这个 TiDB 节点获得了

该元数据对象的元数据锁。当集群中的所有 TiDB 节点都获得了该元数据对象的元数据锁后，才能进行下一次状态变更。

14.12 UI

14.12.1 TiDB Dashboard

14.12.1.1 TiDB Dashboard 介绍

TiDB Dashboard 是 TiDB 自 4.0 版本起提供的图形化界面，可用于监控及诊断 TiDB 集群。TiDB Dashboard 内置于 TiDB 的 PD 组件中，无需独立部署。

> **注意：**
>
> TiDB v6.5.0 且 TiDB Operator v1.4.0 之后，在 Kubernetes 上支持将 TiDB Dashboard 作为独立的 Pod 部署。具体信息，参考 TiDB Operator 部署独立的 TiDB Dashboard。

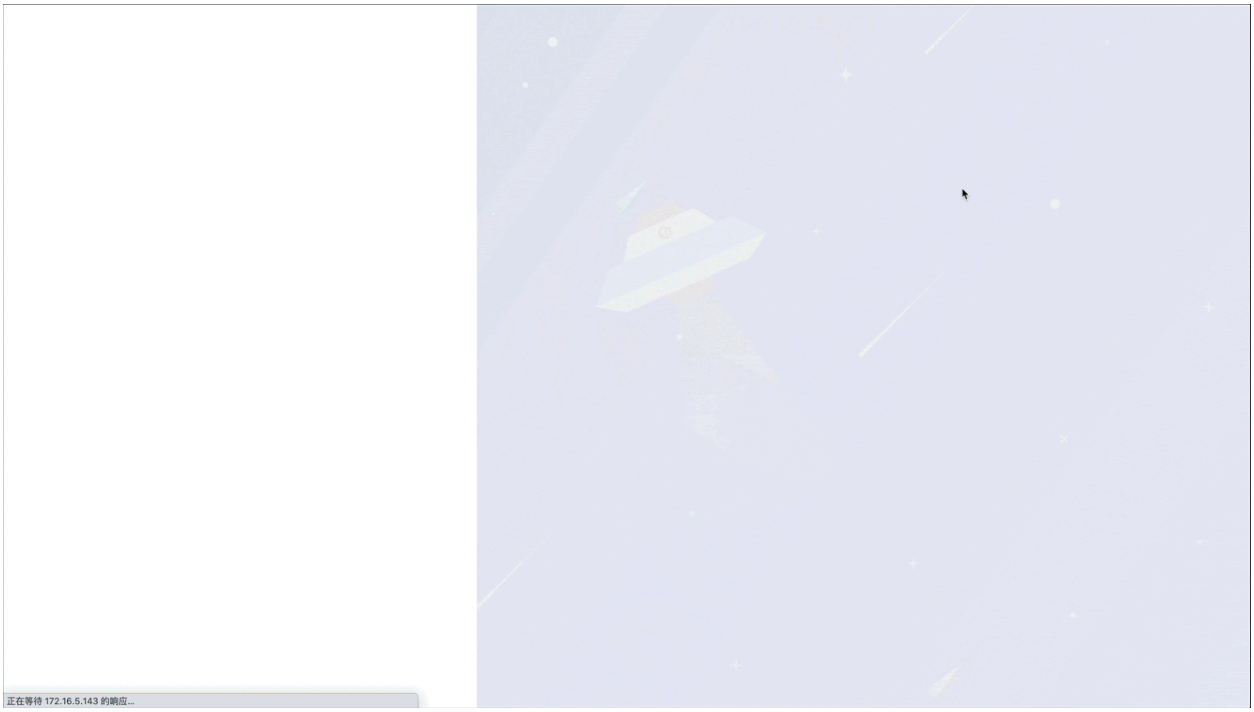


图 490: 界面

TiDB Dashboard 在 GitHub 上开源。

以下列出了 TiDB Dashboard 的主要功能，可分别点击小节内的链接进一步了解详情。

14.12.1.1.1 了解集群整体运行概况

查看集群整体 QPS 数值、执行耗时、消耗资源最多的几类 SQL 语句等概况信息。

参阅概况页面了解详情。

14.12.1.1.2 查看组件及主机运行状态

查看整个集群中 TiDB、TiKV、PD、TiFlash 组件的运行状态及其所在主机的运行状态。

参阅集群信息页面了解详情。

14.12.1.1.3 分析集群读写流量分布及趋势变化

通过热力图形式可视化地展示整个集群中读写流量随时间的变化情况，及时发现业务模式的变化，或定位性能不均衡的热点所在。

参阅流量可视化页面了解详情。

14.12.1.1.4 列出所有 SQL 查询的耗时等执行信息

列出所有 SQL 语句在集群上执行情况，了解各个阶段的执行时间、总运行次数等信息，帮助用户分析和定位集群中最消耗资源的查询，优化整体性能。

参阅SQL 语句分析页面了解详情。

14.12.1.1.5 详细了解耗时较长的 SQL 语句的执行信息

列出所有耗时较长的 SQL 语句文本及其执行信息，帮助用户定位 SQL 语句性能缓慢或发生性能抖动的原因。

参阅慢查询页面了解详情。

14.12.1.1.6 诊断常见集群问题并生成报告

自动判断集群中是否存在一些常见的风险（如配置不一致）或问题，生成报告并给出操作建议，或对比集群在不同时间段的各个指标状态，供用户分析可能存在问题的方向。

参阅集群诊断页面了解详情。

14.12.1.1.7 查询所有组件日志

按关键字、时间范围等条件快速搜索集群中所有运行实例的日志，并可打包下载到本地。

参阅日志搜索页面了解详情。

14.12.1.1.8 收集分析各个组件的性能数据

高级调试功能：无需第三方工具，在线地对各个组件进行性能分析，剖析组件实例在分析时间段内执行的各种内部操作及比例。

参阅实例性能分析页面了解详情。

14.12.1.2 运维

14.12.1.2.1 部署 TiDB Dashboard

TiDB Dashboard 界面内置于 TiDB 4.0 或更高版本的 PD 组件中，无需额外部署。只需部署标准 TiDB 集群，TiDB Dashboard 就会原生集成。

> **注意：**
>
> TiDB v6.5.0 且 TiDB Operator v1.4.0 之后，在 Kubernetes 上支持将 TiDB Dashboard 作为独立的 Pod 部署。具体信息，参考 TiDB Operator 部署独立的 TiDB Dashboard。

请参阅下列文档了解如何部署标准 TiDB 集群：

- 快速试用 TiDB 集群
- 生产环境部署
- Kubernetes 环境部署

> **注意：**
>
> TiDB Dashboard 目前不能在低于 4.0 版本的集群中部署或使用。

多 PD 实例部署

当集群中部署了多个 PD 实例时，其中仅有一个 PD 实例会固定地提供 TiDB Dashboard 服务。

各个 PD 首次运行时会自动协商出其中某一个实例提供 TiDB Dashboard 服务。协商完毕后，无论重启或扩容，都会固定在这个实例上运行 TiDB Dashboard 服务，除非该实例被手动缩容。其他 PD 实例不会运行 TiDB Dashboard 服务。这个协商过程无需用户介入，会自动完成。

当用户访问不提供 TiDB Dashboard 服务的 PD 实例时，浏览器将会收到重定向指令，自动引导用户重新访问提供了 TiDB Dashboard 服务的 PD 实例，从而能正常使用。流程如下图所示。

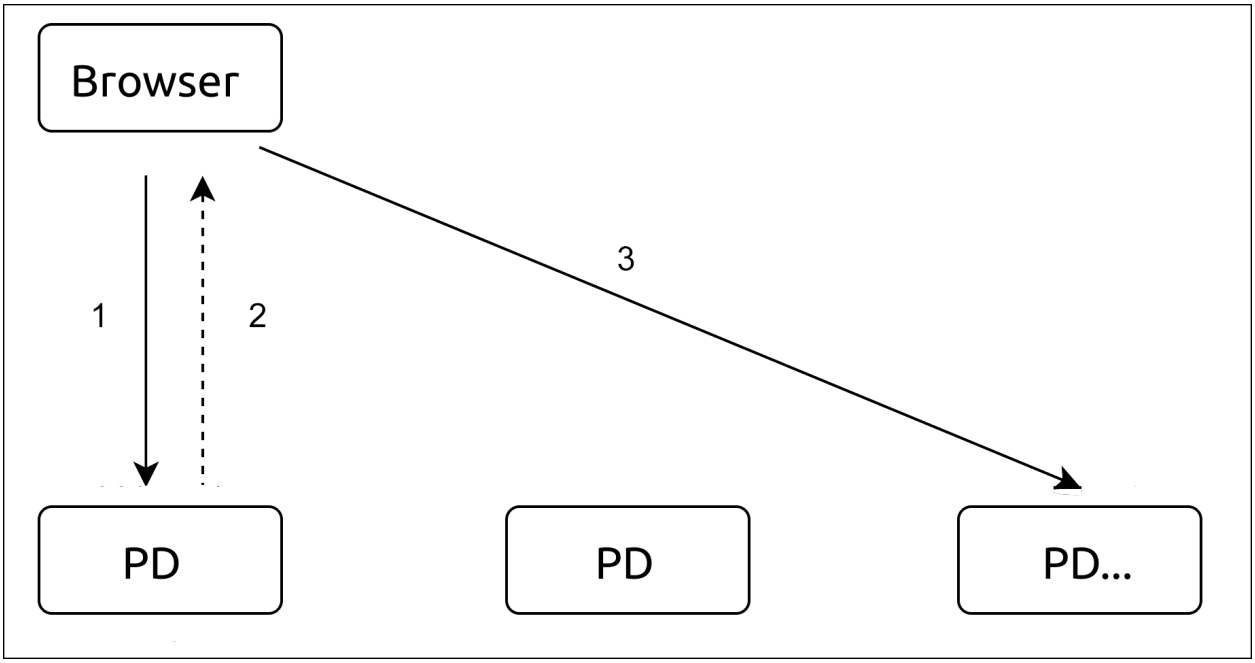


图 491: 流程示意

> **注意：**
>
> 提供 TiDB Dashboard 服务的 PD 实例不一定与 PD leader 一致。

查询实际运行 TiDB Dashboard 服务的 PD 实例

使用 TiUP 部署时，对于已启动的集群，可通过 `tiup cluster display` 命令查看哪个 PD 节点提供了 TiDB Dashboard 服务（将 CLUSTER_NAME 替换为集群名称）：

```
tiup cluster display CLUSTER_NAME --dashboard
```

输出样例如下：

```
http://192.168.0.123:2379/dashboard/
```

> **注意：**
>
> 该功能在 TiUP Cluster v1.0.3 或更高版本部署工具中提供。
>
> 升级 TiUP Cluster 步骤
>
> ```
> tiup update --self
> tiup update cluster --force
> ```

切换其他 PD 实例提供 TiDB Dashboard 服务

使用 TiUP 部署时，对于已启动的集群，可使用 `tiup ctl:v<CLUSTER_VERSION> pd` 命令切换其他 PD 实例运行 TiDB Dashboard，或在禁用 TiDB Dashboard 的情况下重新指定一个 PD 实例运行 TiDB Dashboard：

```
tiup ctl:v<CLUSTER_VERSION> pd -u http://127.0.0.1:2379 config set dashboard-address http
    ↪ ://9.9.9.9:2379
```

其中：

- 将 `127.0.0.1:2379` 替换为任意 PD 实例的 IP 和端口
- 将 `9.9.9.9:2379` 替换为想运行 TiDB Dashboard 服务的新 PD 实例的 IP 和端口

修改完毕后，可使用 `tiup cluster display` 命令确认修改是否生效（将 CLUSTER_NAME 替换为集群名称）：

```
tiup cluster display CLUSTER_NAME --dashboard
```

> 警告：
>
> 切换 TiDB Dashboard 将会丢失之前 TiDB Dashboard 实例所存储的本地数据，包括流量可视化历史、历史搜索记录等。

禁用 TiDB Dashboard

使用 TiUP 部署时，对于已启动的集群，可使用 `tiup ctl:v<CLUSTER_VERSION> pd` 命令在所有 PD 实例上禁用 TiDB Dashboard（将 `127.0.0.1:2379` 替换为任意 PD 实例的 IP 和端口）：

```
tiup ctl:v<CLUSTER_VERSION> pd -u http://127.0.0.1:2379 config set dashboard-address none
```

禁用 TiDB Dashboard 后，查询哪个 PD 实例提供 TiDB Dashboard 服务将会失败：

```
Error: TiDB Dashboard is disabled
```

浏览器访问任意 PD 实例的 TiDB Dashboard 地址也将提示失败：

```
Dashboard is not started.
```

重新启用 TiDB Dashboard

使用 TiUP 部署时，对于已启动的集群，可使用 `tiup ctl:v<CLUSTER_VERSION> pd` 命令，要求 PD 重新协商出某一个实例运行 TiDB Dashboard（将 `127.0.0.1:2379` 替换为任意 PD 实例的 IP 和端口）：

```
tiup ctl:v<CLUSTER_VERSION> pd -u http://127.0.0.1:2379 config set dashboard-address auto
```

修改完毕后，使用 `tiup cluster display` 命令查看 PD 自动协商出的 TiDB Dashboard 实例地址（将 CLUSTER_NAME 替换为集群名称）：

```
tiup cluster display CLUSTER_NAME --dashboard
```

还可以通过手动指定哪个 PD 实例运行 TiDB Dashboard 服务的方式重新启用 TiDB Dashboard，具体操作参见上文切换其他 PD 实例提供 TiDB Dashboard 服务。

> 警告：
>
> 若新启用的 TiDB Dashboard 实例与禁用前的实例不一致，将会丢失之前 TiDB Dashboard 实例所存储的本地数据，包括流量可视化历史、历史搜索记录等。

下一步

- 参阅访问 TiDB Dashboard 章节了解如何访问及登录集群上的 TiDB Dashboard 界面。
- 参阅提高 TiDB Dashboard 安全性章节了解如何增强 TiDB Dashboard 的安全性，如配置防火墙等。

14.12.1.2.2 通过反向代理使用 TiDB Dashboard

你可以使用反向代理将 TiDB Dashboard 服务安全从内部网络提供给外部网络。

操作步骤

第 1 步：获取实际 TiDB Dashboard 地址

当集群中部署有多个 PD 实例时，其中仅有一个 PD 实例会真正运行 TiDB Dashboard，因此需要确保反向代理的上游 (Upstream) 指向了正确的地址。关于该机制的详情，可参阅TiDB Dashboard 多 PD 实例部署章节。

使用 TiUP 部署工具时，操作命令如下（将 CLUSTER_NAME 替换为集群名称）：

```
tiup cluster display CLUSTER_NAME --dashboard
```

输出即为实际 TiDB Dashboard 地址。样例如下：

```
http://192.168.0.123:2379/dashboard/
```

> 注意：
>
> 该功能在 TiUP Cluster v1.0.3 或更高版本部署工具中提供。
>
> 升级 TiUP Cluster 步骤
>
> ```
> tiup update --self
> tiup update cluster --force
> ```

第 2 步：配置反向代理

使用 HAProxy 反向代理

HAProxy 作为反向代理时，方法如下：

1. 以在 8033 端口反向代理 TiDB Dashboard 为例，在 HAProxy 配置文件中，新增如下配置：

“‘haproxy frontend tidb_dashboard_front bind *:8033 use_backend tidb_dashboard_back if { path /dashboard } or { path_beg /dashboard/ }

backend tidb_dashboard_back mode http server tidb_dashboard 192.168.0.123:2379 “‘

其中 192.168.0.123:2379 需替换为第 1 步：获取实际 TiDB Dashboard 地址中取得的 TiDB Dashboard 实际地址中的 IP 及端口部分。

> 警告：
>
> 请务必保留 use_backend 指令中的 if 部分，确保只有该路径下的服务会被反向代理，否则将引入安全风险。参见提高 TiDB Dashboard 安全性。

2. 重启 HAProxy，以使配置生效。
3. 测试反向代理是否生效：访问 HAProxy 所在机器的 8033 端口下 /dashboard/ 地址，如 http://example.com:8033/dashboard/，即可访问 TiDB Dashboard。

使用 NGINX 反向代理

NGINX 作为反向代理时，方法如下：

1. 以在 8033 端口反向代理 TiDB Dashboard 为例，在 NGINX 配置文件中，新增如下配置：

```
nginx  server {  listen 8033;  location /dashboard/ { proxy_pass http://192.168.0.123:2379/dashboard
↪ /; } }
```

其中 http://192.168.0.123:2379/dashboard/ 需替换为第 1 步：获取实际 TiDB Dashboard 地址中取得的 TiDB Dashboard 实际地址。

> 警告：
>
> 请务必保留 proxy_pass 指令中的 /dashboard/ 路径，确保只有该路径下的服务会被反向代理，否则将引入安全风险。参见提高 TiDB Dashboard 安全性。

2. 重新载入 NGINX 以使配置生效：

```
shell  sudo nginx -s reload
```

3. 测试反向代理是否生效：访问 NGINX 所在机器的 8033 端口下 /dashboard/ 地址，如 http://example.com
↪ :8033/dashboard/，即可访问 TiDB Dashboard。

自定义路径前缀

TiDB Dashboard 默认在 /dashboard/ 路径下提供服务，即使是反向代理也是如此，例如 http://example.com:8033/dashboard/。若要配置反向代理以非默认的路径提供 TiDB Dashboard 服务，例如 http://example.com:8033/foo/ 或 http://example.com:8033/，可参考以下步骤。

第 1 步：修改 PD 配置指定 TiDB Dashboard 服务路径前缀

修改 PD 配置中 [dashboard] 类别的 public-path-prefix 配置项，可指定服务路径前缀。该配置修改后需要重启 PD 实例生效。

以 TiUP 部署且希望运行在 http://example.com:8033/foo/ 为例，可指定以下配置：

```
server_configs:
  pd:
    dashboard.public-path-prefix: /foo
```

使用 TiUP 部署全新集群时修改配置

若要全新部署集群，可在 TiUP 拓扑文件 topology.yaml 中加入上述配置项后进行部署，具体步骤参阅TiUP 部署文档。

使用 TiUP 修改已部署集群的配置

1. 以编辑模式打开该集群的配置文件（将 CLUSTER_NAME 替换为集群名称）

```
shell  tiup cluster edit-config CLUSTER_NAME
```

2. 在 server_configs 的 pd 配置下修改或新增配置项，若没有 server_configs 请在最顶层新增：

```
yaml  server_configs: pd:     dashboard.public-path-prefix: /foo
```

修改完成后的配置文件类似于：

```
yaml  server_configs: pd:     dashboard.public-path-prefix: /foo global: user: tidb ...
```

或

```
yaml  monitored:  ...  server_configs: tidb: ...  tikv: ...  pd:     dashboard.public-path-
↪ prefix: /foo ...
```

3. 滚动重启所有 PD 实例生效配置（将 CLUSTER_NAME 替换为集群名称）

```
shell  tiup cluster reload CLUSTER_NAME -R pd
```

详情请参阅TiUP 常见运维操作 - 修改配置参数。

若希望运行在根路径（如 http://example.com:8033/）下，相应的配置为：

```
server_configs:
  pd:
    dashboard.public-path-prefix: /
```

> **警告：**
>
> 修改自定义路径前缀生效后，直接访问将不能正常使用 TiDB Dashboard，您只能通过和路径前缀匹配的反向代理访问。

第 2 步：修改反向代理配置

使用 HAProxy 反向代理

以 `http://example.com:8033/foo/` 为例，HAProxy 配置如下：

```
frontend tidb_dashboard_front
  bind *:8033
  use_backend tidb_dashboard_back if { path /foo } or { path_beg /foo/ }

backend tidb_dashboard_back
  mode http
  http-request set-path %[path,regsub(^/foo/?,/dashboard/)]
  server tidb_dashboard 192.168.0.123:2379
```

其中 `192.168.0.123:2379` 需替换为第 1 步：获取实际 TiDB Dashboard 地址中取得的 TiDB Dashboard 实际地址中的 IP 及端口部分。

> **警告：**
>
> 请务必保留 `use_backend` 指令中的 `if` 部分，确保只有该路径下的服务会被反向代理，否则将引入安全风险。参见提高 TiDB Dashboard 安全性。

若希望运行在根路径（如 `http://example.com:8033/`），HAProxy 配置如下：

```
frontend tidb_dashboard_front
  bind *:8033
  use_backend tidb_dashboard_back

backend tidb_dashboard_back
  mode http
  http-request set-path /dashboard%[path]
  server tidb_dashboard 192.168.0.123:2379
```

修改配置并重启 HAProxy 后即可生效。

使用 NGINX 反向代理

以 `http://example.com:8033/foo/` 为例，相应的 NGINX 配置为：

```
server {
  listen 8033;
  location /foo/ {
    proxy_pass http://192.168.0.123:2379/dashboard/;
  }
}
```

其中 `http://192.168.0.123:2379/dashboard/` 需替换为第 1 步：获取实际 TiDB Dashboard 地址中取得的 TiDB Dashboard 实际地址。

> **警告：**
>
> 请务必保留 `proxy_pass` 指令中的 `/dashboard/` 路径，确保只有该路径下的服务会被反向代理，否则将引入安全风险。参见提高 TiDB Dashboard 安全性。

若希望运行在根路径（如 `http://example.com:8033/`），NGINX 配置为：

```
server {
  listen 8033;
  location / {
    proxy_pass http://192.168.0.123:2379/dashboard/;
  }
}
```

修改配置并重启 NGINX 后即可生效：

```
sudo nginx -s reload
```

下一步

参阅提高 TiDB Dashboard 安全性文档了解如何增强 TiDB Dashboard 的安全性，如配置防火墙等。

14.12.1.2.3 TiDB Dashboard 用户管理

TiDB Dashboard 与 TiDB 使用相同的用户权限体系和登录验证方式。你可以通过控制和管理 TiDB SQL 用户，从而限制和约束这些用户对 TiDB Dashboard 的访问。本文描述了 TiDB SQL 用户访问 TiDB Dashboard 所需的最小权限，并提供了如何创建最小权限 SQL 用户、如何通过 RBAC 授权 SQL 用户登录的示例。

要了解如何控制和管理 TiDB SQL 用户，请参见TiDB 用户账户管理。

所需权限说明

- 当所连接的 TiDB 服务器未启用安全增强模式 (SEM) 时，要访问 TiDB Dashboard，SQL 用户应当拥有以下所有权限：
  - PROCESS

  - SHOW DATABASES
  - CONFIG
  - DASHBOARD_CLIENT

- 当所连接的 TiDB 服务器启用了安全增强模式 (SEM) 时，要访问 TiDB Dashboard，SQL 用户应当拥有以下所有权限：

  - PROCESS
  - SHOW DATABASES
  - CONFIG
  - DASHBOARD_CLIENT
  - RESTRICTED_TABLES_ADMIN
  - RESTRICTED_STATUS_ADMIN
  - RESTRICTED_VARIABLES_ADMIN

- 若希望 SQL 用户在登录 TiDB Dashboard 后允许修改界面上的各项配置，SQL 用户还应当拥有以下权限：

  - SYSTEM_VARIABLES_ADMIN

- 若希望 SQL 用户在登录 TiDB Dashboard 后允许使用快速绑定执行计划功能。

  - SYSTEM_VARIABLES_ADMIN
  - SUPER

> **注意：**
>
> 拥有 `ALL PRIVILEGES` 或 `SUPER` 等粗粒度高权限的用户同样可以登录 TiDB Dashboard。出于最小权限原则，强烈建议创建用户时仅使用上述精细权限，从而防止用户执行非预期操作。请参阅权限管理了解这些权限的详细信息。

如果登录 TiDB Dashboard 时指定的 SQL 用户未满足上述权限需求，则登录将失败，如下图所示：

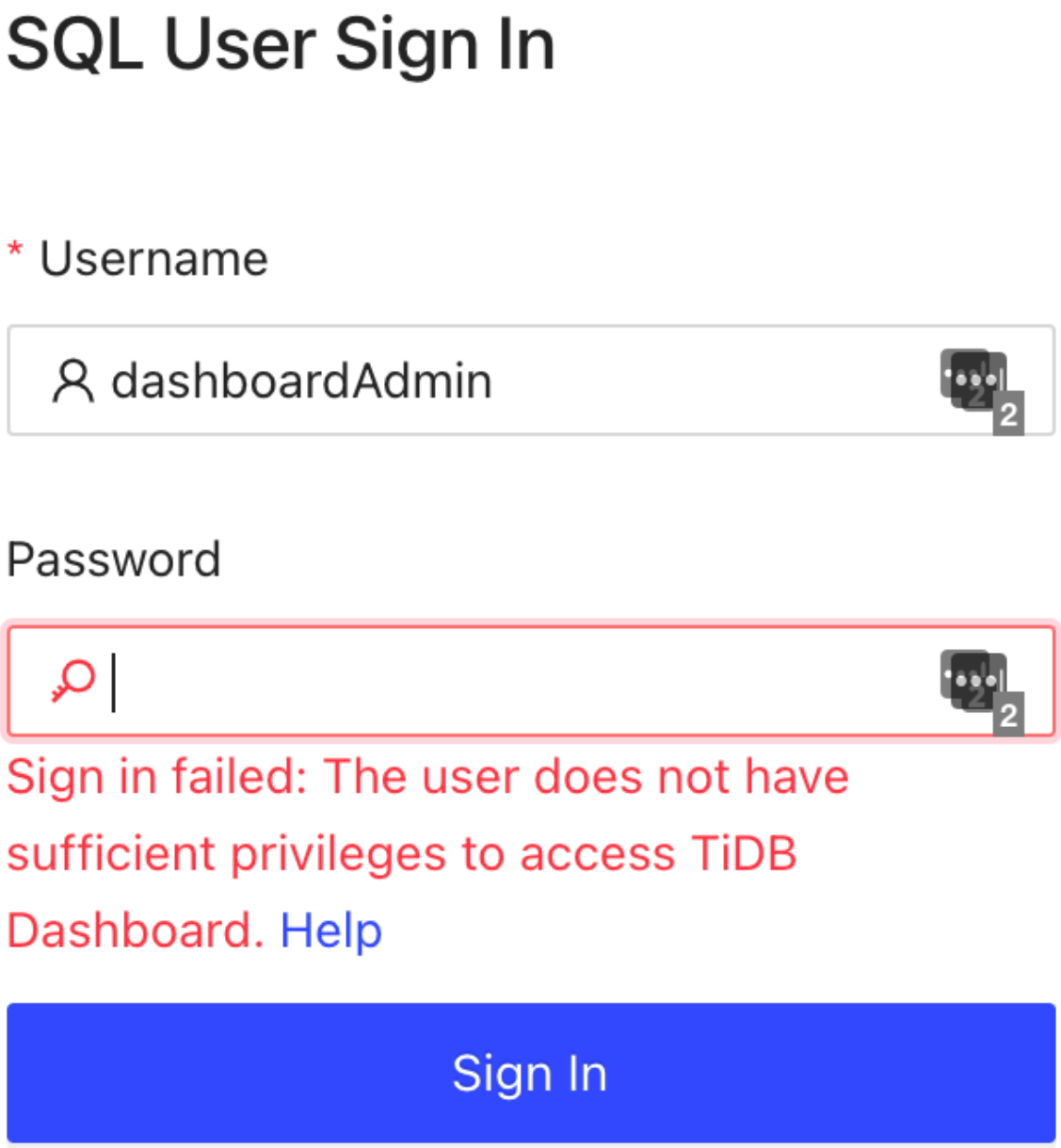


图 492: insufficient-privileges

示例：创建一个最小权限 SQL 用户用于登录 TiDB Dashboard

- 当所连接的 TiDB 服务器未启用安全增强模式 (SEM) 时，你可以通过执行以下示例 SQL 语句创建一个允许登录 TiDB Dashboard 的 SQL 用户 `dashboardAdmin`：

```
CREATE USER 'dashboardAdmin'@'%' IDENTIFIED BY '<YOUR_PASSWORD>';
GRANT PROCESS, CONFIG ON *.* TO 'dashboardAdmin'@'%';
GRANT SHOW DATABASES ON *.* TO 'dashboardAdmin'@'%';
```

```
GRANT DASHBOARD_CLIENT ON *.* TO 'dashboardAdmin'@'%';

-- 如果要使自定义的 SQL 用户能修改 TiDB Dashboard 界面上的各项配置，可以增加以下权限
GRANT SYSTEM_VARIABLES_ADMIN ON *.* TO 'dashboardAdmin'@'%';

-- 如果要使用快速绑定执行计划（具体参见 https://docs.pingcap.com/zh/tidb/v7.0/dashboard-
    ↪ statement-details#快速绑定执行计划）功能，可以增加以下权限
GRANT SYSTEM_VARIABLES_ADMIN ON *.* TO 'dashboardAdmin'@'%';
GRANT SUPER ON *.* TO 'dashboardAdmin'@'%';
```

- 当所连接的 TiDB 服务器启用了安全增强模式 (SEM) 时，先关闭 SEM，然后执行以下示例 SQL 语句创建一个允许登录 TiDB Dashboard 的 SQL 用户 dashboardAdmin，创建完成后，再重新开启 SEM：

```
CREATE USER 'dashboardAdmin'@'%' IDENTIFIED BY '<YOUR_PASSWORD>';
GRANT PROCESS, CONFIG ON *.* TO 'dashboardAdmin'@'%';
GRANT SHOW DATABASES ON *.* TO 'dashboardAdmin'@'%';
GRANT DASHBOARD_CLIENT ON *.* TO 'dashboardAdmin'@'%';
GRANT RESTRICTED_STATUS_ADMIN ON *.* TO 'dashboardAdmin'@'%';
GRANT RESTRICTED_TABLES_ADMIN ON *.* TO 'dashboardAdmin'@'%';
GRANT RESTRICTED_VARIABLES_ADMIN ON *.* TO 'dashboardAdmin'@'%';

-- 如果要使自定义的 SQL 用户能修改 TiDB Dashboard 界面上的各项配置，可以增加以下权限
GRANT SUPER ON *.* TO 'dashboardAdmin'@'%';

-- 如果要使用快速绑定执行计划（具体参见 https://docs.pingcap.com/zh/tidb/v7.0/dashboard-
    ↪ statement-details#快速绑定执行计划）功能，可以增加以下权限
GRANT SYSTEM_VARIABLES_ADMIN ON *.* TO 'dashboardAdmin'@'%';
GRANT SUPER ON *.* TO 'dashboardAdmin'@'%';
```

示例：通过 RBAC 授权 SQL 用户登录 TiDB Dashboard

以下示例演示了如何在基于角色的访问控制 (RBAC) 机制下创建角色及用户来登录 TiDB Dashboard。

1. 创建一个包含 TiDB Dashboard 所有功能所需权限的角色 dashboard_access：

```
CREATE ROLE 'dashboard_access';
GRANT PROCESS, CONFIG ON *.* TO 'dashboard_access'@'%';
GRANT SHOW DATABASES ON *.* TO 'dashboard_access'@'%';
GRANT DASHBOARD_CLIENT ON *.* TO 'dashboard_access'@'%';
GRANT SYSTEM_VARIABLES_ADMIN ON *.* TO 'dashboard_access'@'%';
GRANT SUPER ON *.* TO 'dashboard_access'@'%';
```

2. 为其他用户授权 dashboard_access 角色并设置为默认启用：

```
CREATE USER 'dashboardAdmin'@'%' IDENTIFIED BY '<YOUR_PASSWORD>';
GRANT 'dashboard_access' TO 'dashboardAdmin'@'%';
```

```
-- 需要默认启用 dashboard_access 角色
SET DEFAULT ROLE dashboard_access to 'dashboardAdmin'@'%';
```

完成以上步骤后，可以用 dashboardAdmin 用户登录 TiDB Dashboard。

登录 TiDB Dashboard

创建满足 TiDB Dashboard 权限要求的 SQL 用户后，你可以使用该用户登录 TiDB Dashboard。

14.12.1.2.4 提高 TiDB Dashboard 安全性

尽管访问 TiDB Dashboard 需要登录，但它被设计为默认供可信的用户实体访问。当你希望将 TiDB Dashboard 提供给外部网络用户或不可信用户访问时，需要注意采取以下措施以避免安全漏洞。

提高 TiDB 用户安全性

为 root 用户设置强密码

TiDB Dashboard 的账号体系与 TiDB SQL 用户一致。默认部署情况下，TiDB 的 root 用户没有密码，因而访问 TiDB Dashboard 也不需要密码验证。这将会给恶意访问者极大的集群权限，包括执行特权 SQL 等。

建议的措施：

- 为 TiDB 的 root 用户设置一个强密码（请参见TiDB 用户账户管理了解详情），或禁用 root 账户。

为 TiDB Dashboard 创建最小权限用户

TiDB Dashboard 的账号体系与 TiDB SQL 用户一致，并基于 TiDB SQL 用户的权限进行 TiDB Dashboard 授权验证。TiDB Dashboard 所需的权限较少，甚至可以只有只读权限。可以基于最小权限原则配置合适的用户访问 TiDB Dashboard，减少高权限用户的使用场景。

建议的措施：

- 为访问 TiDB Dashboard 创建一个最小权限的 SQL 用户，并用该用户登录 TiDB Dashboard，避免使用高权限用户，提升安全性。请参见TiDB Dashboard 用户管理了解详情。

使用防火墙阻止不可信访问

> 注意：
>
> TiDB v6.5.0 且 TiDB Operator v1.4.0 之后，在 Kubernetes 上支持将 TiDB Dashboard 作为独立的 Pod 部署。在 TiDB Operator 环境，可直接访问该 Pod 的 IP 来打开 TiDB Dashboard，该端口不与其他 PD 内部特权接口关联，对外提供该端口不需要额外的防火墙操作。具体信息，参考 TiDB Operator 部署独立的 TiDB Dashboard。

TiDB Dashboard 通过 PD Client 端口提供服务，默认为 http://IP:2379/dashboard/。尽管 TiDB Dashboard 需要验证身份，但 PD Client 端口上承载的其他 PD 内部特权接口不需要验证身份，且能进行特权操作，例如 http://IP:2379/pd/api/v1/members。因此，将 PD Client 端口直接暴露给外部网络具有极大的风险。

建议的措施：

1. 使用防火墙禁止组件外部网络或不可信网络访问任何 PD 组件的 Client 端口。

注意，TiDB、TiKV 等组件需要通过 PD Client 端口与 PD 组件进行通信，因此请勿对组件内部网络阻止访问，这将导致集群不可用。

2. 参见通过反向代理使用 TiDB Dashboard 了解如何配置反向代理将 TiDB Dashboard 服务在另一个端口上安全地提供给外部网络。

如何在多 PD 实例部署时开放 TiDB Dashboard 端口访问

> **警告：**
>
> 本章节描述了一个不安全的访问方案，仅供测试环境使用。在生产环境中，不要使用本方案。请参见本文的其余章节。

在测试环境中，您可能需要配置防火墙开放 TiDB Dashboard 端口供外部访问。

当部署了多个 PD 实例时，其中仅有一个 PD 实例会真正运行 TiDB Dashboard，访问其他 PD 实例时会发生浏览器重定向，因此需要确保防火墙配置了正确的 IP 地址。关于该机制的详情，可参阅TiDB Dashboard 多 PD 实例部署章节。

使用 TiUP 部署工具时，可使用以下命令查看实际运行 TiDB Dashboard 的 PD 实例地址（将 CLUSTER_NAME 替换为集群名称）：

```
tiup cluster display CLUSTER_NAME --dashboard
```

输出即为实际 TiDB Dashboard 地址。

> **注意：**
>
> 该功能在 TiUP Cluster v1.0.3 或更高版本部署工具中提供。
>
> 升级 TiUP Cluster 步骤
>
> ```
> tiup update --self
> tiup update cluster --force
> ```

以下是一个样例输出：

```
http://192.168.0.123:2379/dashboard/
```

在这个样例中，需要为防火墙配置开放 IP 192.168.0.123 的 2379 端口入站访问，并通过 http://192.168.0.123:2379/dashboard/ 访问 TiDB Dashboard。

反向代理仅代理 TiDB Dashboard

如前文所述，PD Client 端口下提供的服务不仅有 TiDB Dashboard（位于 http://IP:2379/dashboard/），还有其他 PD 内部特权接口（如 http://IP:2379/pd/api/v1/members）。因此，使用反向代理将 TiDB Dashboard 提供给外部网络时，应当确保仅提供 /dashboard 前缀下的服务，而非该端口下所有服务，避免外部网络能通过反向代理访问到 PD 内部特权接口。

建议的措施：

- 参见通过反向代理使用 TiDB Dashboard 了解安全且推荐的反向代理配置。

为反向代理开启 TLS

为了进一步加强传输层安全性，可以为反向代理开启 TLS，甚至可以引入 mTLS 实现访问用户的证书验证。

请参阅 NGINX 文档或 HAProxy 文档了解如何为它们开启 TLS。

其他建议的安全措施

- 为 TiDB 组件间通信开启加密传输
- 为 TiDB 客户端服务端间通信开启加密传输

14.12.1.3 访问 TiDB Dashboard

通过浏览器访问 http://127.0.0.1:2379/dashboard/（将 127.0.0.1:2379 替换为实际 PD 实例的地址和端口）即可打开 TiDB Dashboard。

> 注意：
>
> TiDB v6.5.0 且 TiDB Operator v1.4.0 之后，在 Kubernetes 上支持将 TiDB Dashboard 作为独立的 Pod 部署。在 TiDB Operator 环境，可直接访问该 Pod 的 IP 来打开 TiDB Dashboard。具体信息，参考 TiDB Operator 部署独立的 TiDB Dashboard。

14.12.1.3.1 多 PD 实例访问

当集群中部署有多个 PD 实例、且您可以直接访问到每个 PD 实例地址和端口时，可以简单地将 http://127.0.0.1:2379/dashboard/ 地址中的 127.0.0.1:2379 替换为集群中任意一个 PD 实例的地址和端口进行访问。

> 注意：
>
> 当处于防火墙或反向代理等环境下、无法直接访问每个 PD 实例时，可能会无法访问 TiDB Dashboard。这通常是防火墙或反向代理没有正确配置导致的。可参阅通过反向代理使用 TiDB Dashboard 或提高 TiDB Dashboard 安全性章节了解如何在多 PD 实例情况下正确配置防火墙或反向代理规则。

14.12.1.3.2 浏览器兼容性

TiDB Dashboard 可在常见的、更新及时的桌面浏览器中使用，具体版本号为：

- Chrome >= 77
- Firefox >= 68
- Edge >= 17

> **注意：**
>
> 若使用旧版本浏览器或其他浏览器访问 TiDB Dashboard，部分界面可能不能正常工作。

14.12.1.3.3 登录

访问 TiDB Dashboard 将会显示用户登录界面。

- 可使用 TiDB 的 root 用户登录。
- 如果创建了自定义 SQL 用户，也可以使用自定义的 SQL 用户和密码登录。

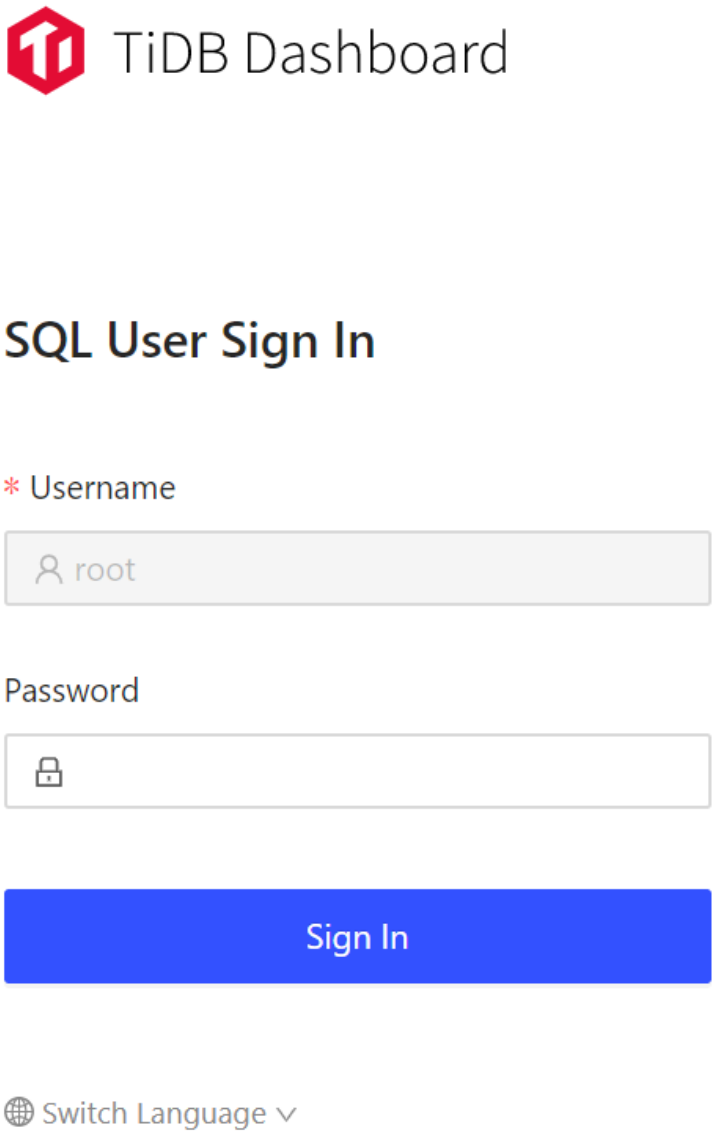


图 493: 登录界面

如果存在以下情况，则可能会登录失败：

- TiDB root 用户不存在
- PD 未启动或无法访问
- TiDB 未启动或无法访问
- root 密码错误

登录后，24 小时内将保持自动登录状态。参见登出章节了解如何登出用户。

14.12.1.3.4 切换语言

TiDB Dashboard 目前支持以下语言：

- 简体中文
- 英文

在登录界面中，可点击 Switch Language 下拉框切换界面显示语言：

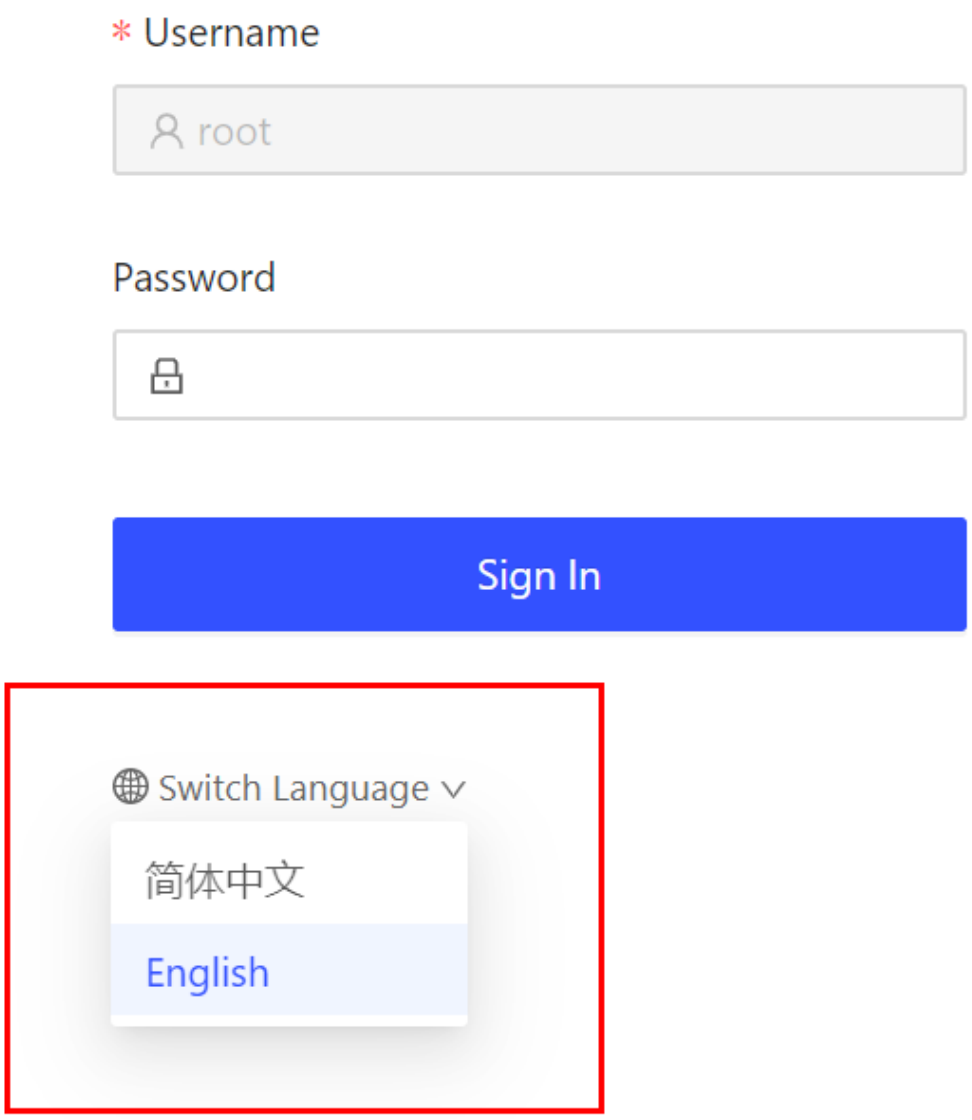


图 494: 切换语言

14.12.1.3.5 登出

登录后，在左侧导航处点击登录用户名，可切换到用户页面。在用户页面点击登出 (Logout) 按钮即可登出当前用户。登出后，需重新输入用户名密码。

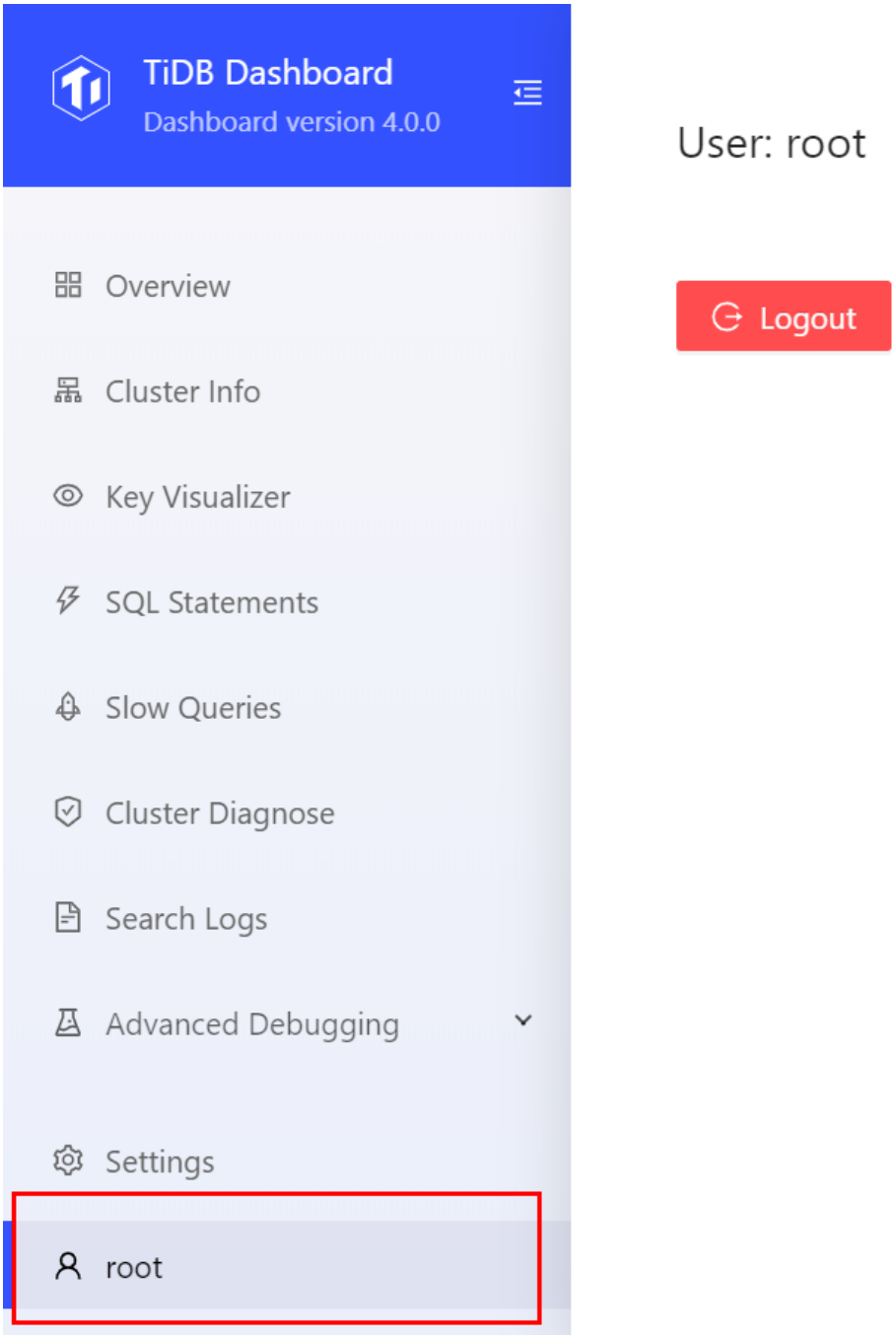


图 495: 登出

14.12.1.4 TiDB Dashboard 概况页面

该页面显示了整个集群的概况，包含以下信息：

- 整个集群的 QPS
- 整个集群的查询延迟

- 最近一段时间内累计耗时最多的若干 SQL 语句
- 最近一段时间内运行时间超过一定阈值的慢查询
- 各个实例的节点数和状态
- 监控及告警信息

14.12.1.4.1 访问

登录 TiDB Dashboard 后默认进入该页面，也可以左侧导航条点击概况 (Overview) 进入：

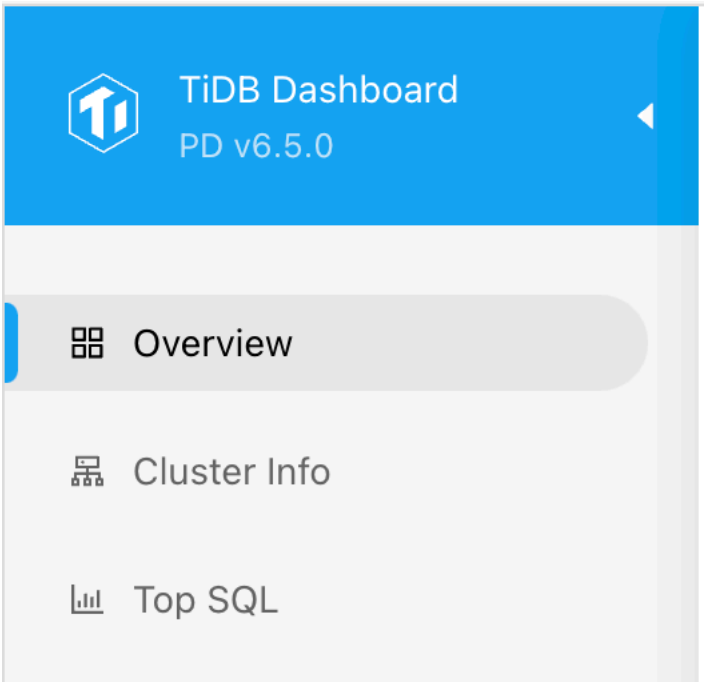


图 496: 访问

14.12.1.4.2 QPS

该区域显示最近一小时整个集群的每秒成功和失败查询数量：

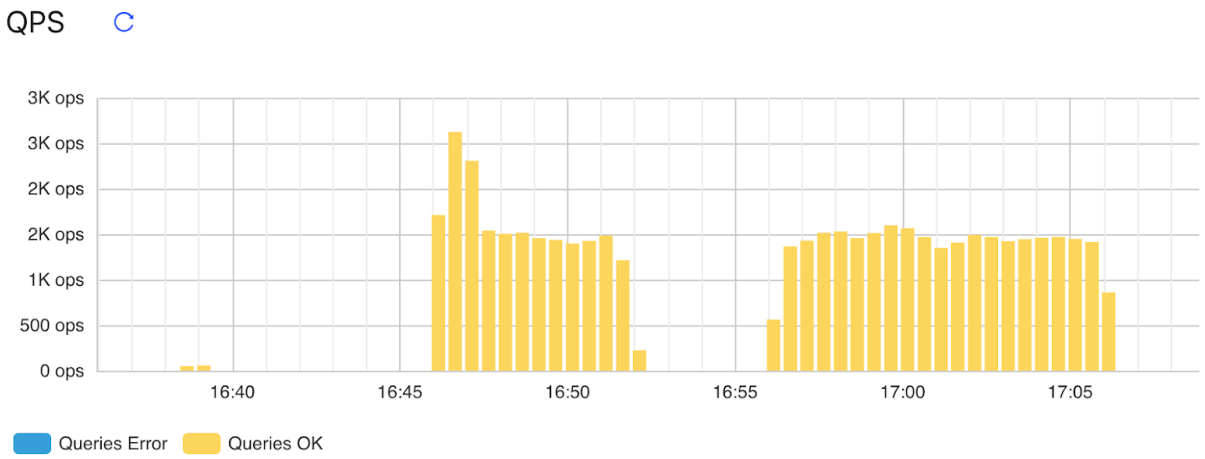


图 497: 界面

> **注意：**
>
> 该功能仅在部署了 Prometheus 监控组件的集群上可用，未部署监控组件的情况下会显示为失败。

14.12.1.4.3 延迟

该区域显示最近一小时整个集群中 99.9%、99% 和 90% 查询的延迟：

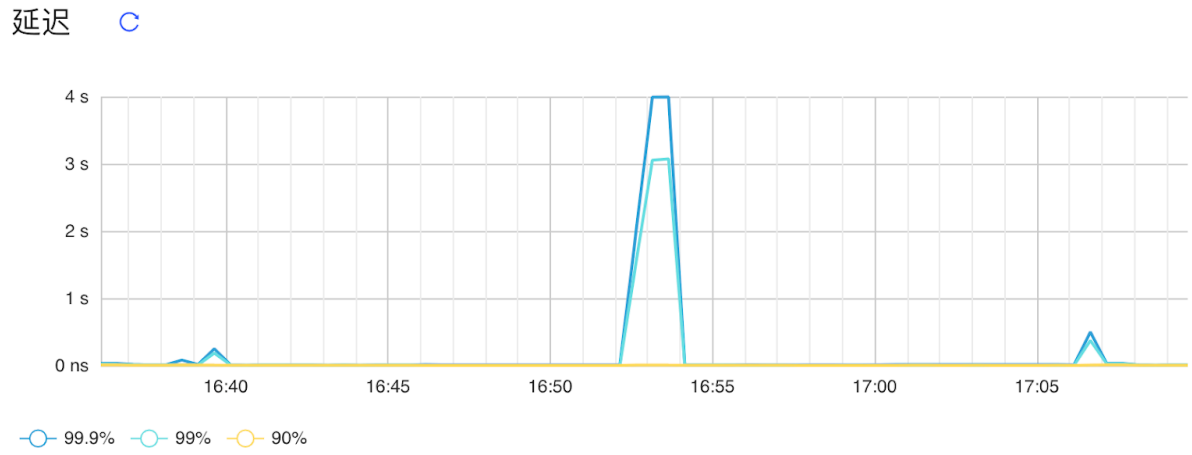


图 498: 界面

> **注意：**
>
> 该功能仅在部署了 Prometheus 监控组件的集群上可用，未部署监控组件的情况下会显示为失败。

14.12.1.4.4 Top SQL 语句

该区域显示最近一段时间内整个群集中累计耗时最长的 10 类 SQL 语句。查询参数不一样但结构一样的 SQL 会归为同一类 SQL 语句，在同一行中显示：

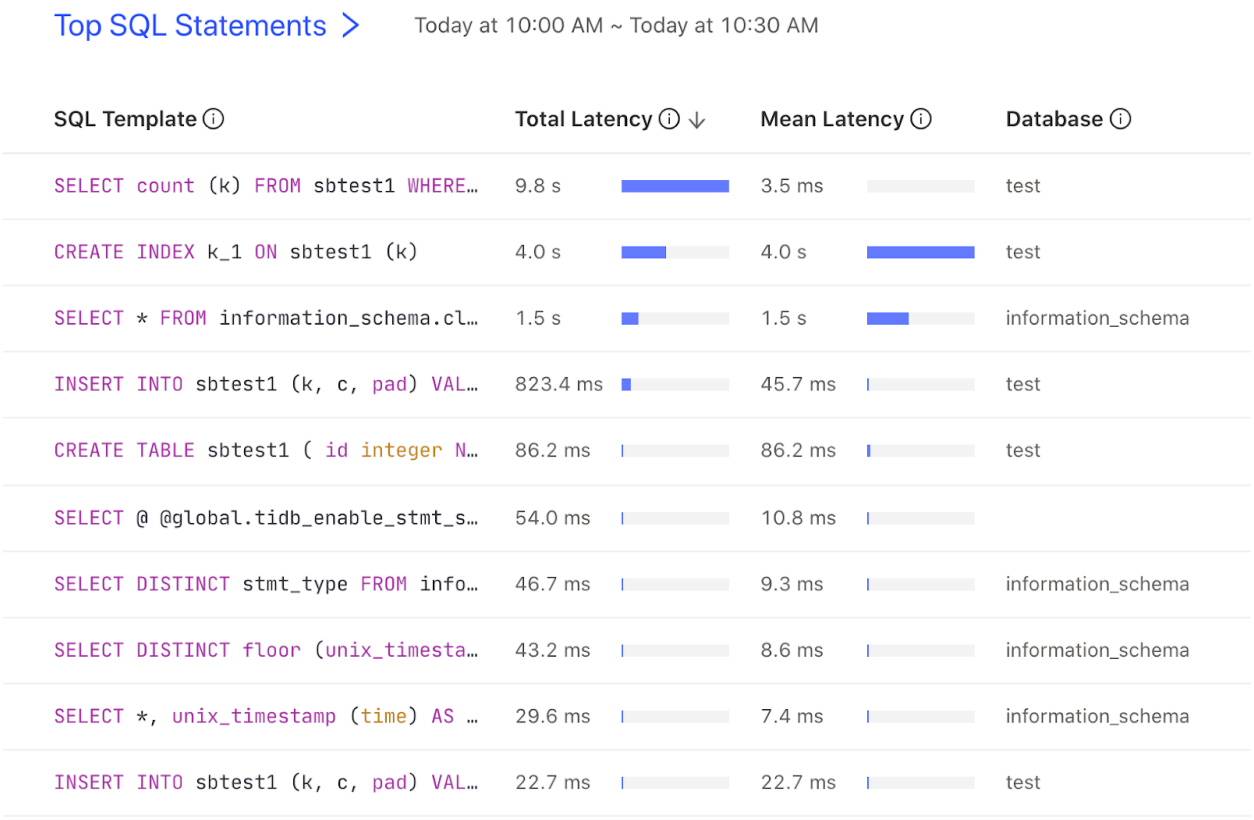


图 499: 界面

该区域显示的内容与SQL 语句分析页面一致，可点击 Top SQL 语句 (Top SQL Statements) 标题查看完整列表。关于该表格中各列详情，见SQL 语句分析页面。

> **注意：**
>
> 该功能仅在开启了 SQL 语句分析功能的集群上可用。

14.12.1.4.5 最近的慢查询

该区域默认显示最近 30 分钟内整个集群中最新的 10 条慢查询：



图 500: 界面

默认情况下运行时间超过 300ms 的 SQL 查询即会被计为慢查询并显示在该表格中。可通过调整tidb_slow_log_threshold 变量或 TiDB slow-threshold 参数调整阈值。

该区域显示的内容与慢查询页面一致，可点击最近的慢查询 (Recent Slow Queries) 标题查看完整列表。关于该表格中各列详情，见慢查询页面。

> 注意：
>
> 该功能仅在配置开启了慢查询日志的集群中可用，使用 TiUP 部署的集群默认开启慢查询日志。

14.12.1.4.6 实例

该区域汇总显示了整个集群中 TiDB、TiKV、PD、TiFlash 的总实例数量及异常实例数量：

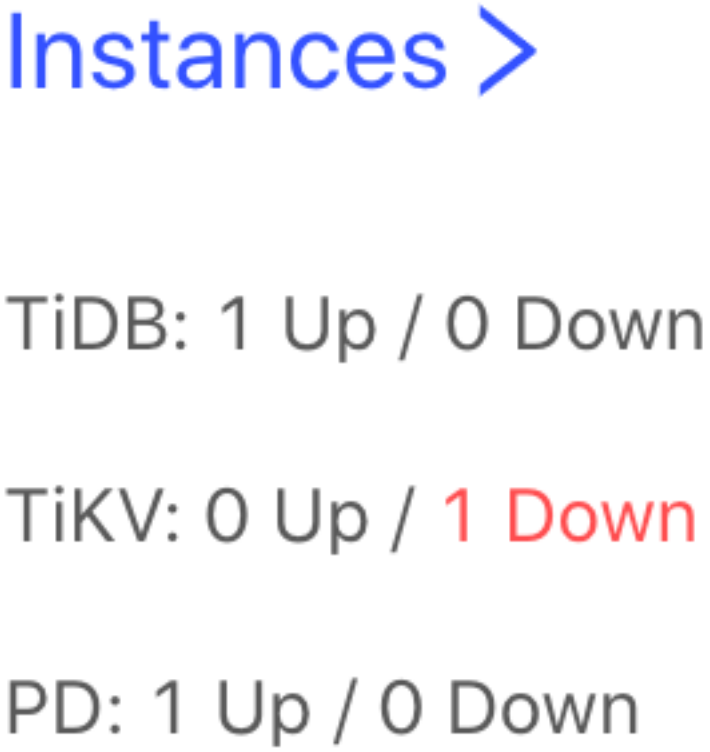


图 501: 界面

状态描述如下：

- Up：实例运行正常（含下线中的存储实例）。
- Down：实例运行异常，例如网络无法连接、进程已崩溃等。

点击实例标题可进入集群信息页面查看各个实例的详细运行状态。

14.12.1.4.7 监控和告警

该区域提供了便捷的链接方便用户查看详细监控或告警：

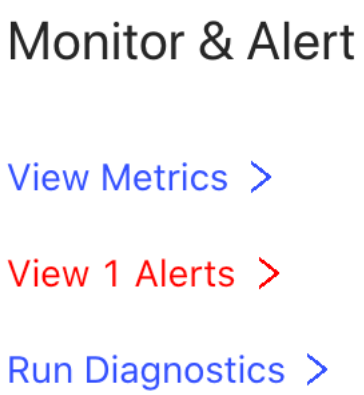


图 502: 界面

- 查看监控链接：点击后跳转至 Grafana 页面，可查看集群详细监控信息。关于 Grafana 监控面板中各个详细监控指标的解释，参见监控指标文档。
- 查看告警链接：点击后跳转至 AlertManager 页面，可查看集群详细告警信息。当集群中已有告警时，告警数量将会直接显示在链接文本上。
- 运行诊断链接：点击后跳转至集群诊断页面，参见集群诊断页面了解详情。

> 注意：
>
> 查看监控链接仅在集群中部署了 Grafana 节点时可用，查看告警链接仅在集群中部署了 AlertManager 节点时可用。

14.12.1.5 TiDB Dashboard 集群信息页面

该页面上允许用户查看整个集群中 TiDB、 TiKV、 PD、 TiFlash 组件的运行状态及其所在主机的运行状态。

14.12.1.5.1 访问

可以通过以下两种方法访问集群信息页面：

- 登录后，左侧导航条点击集群信息 (Cluster Info)：

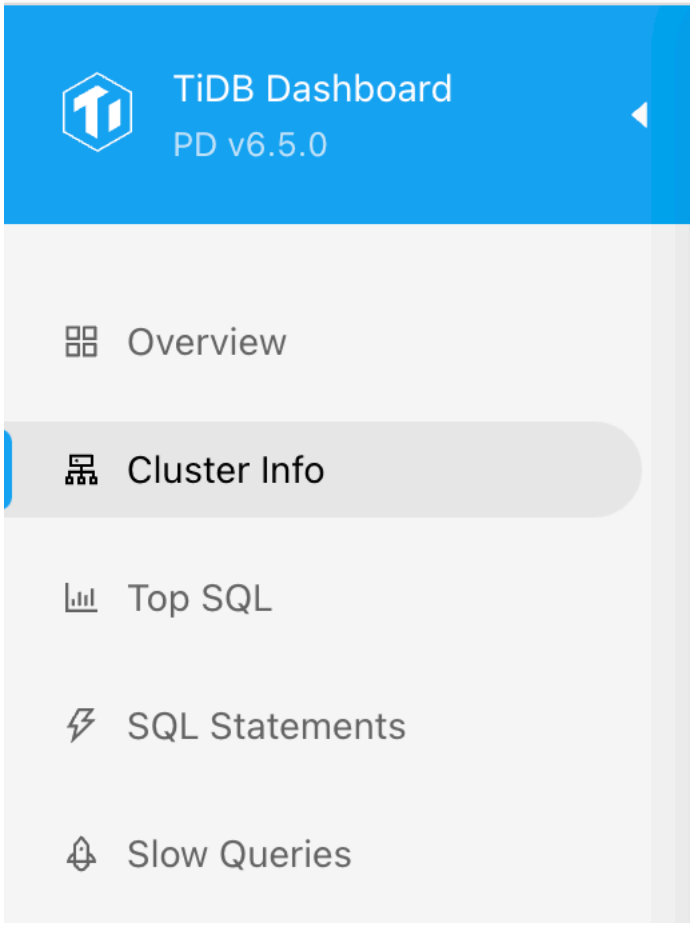


图 503: 访问

- 在浏览器中访问 http://127.0.0.1:2379/dashboard/#/cluster_info/instance（将 127.0.0.1:2379 替换为实际 PD 实例地址和端口）。

14.12.1.5.2 实例列表

点击实例 (Instances) 可查看实例列表：

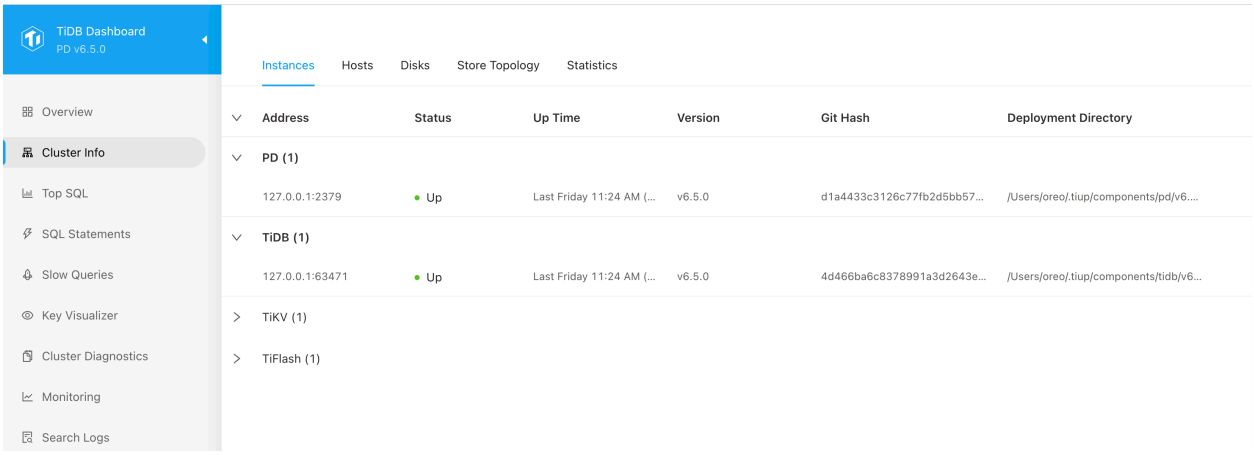


图 504: 实例

实例列表列出了该集群中 TiDB、 TiKV、 PD 和 TiFlash 组件所有实例的概况信息。

表格包含以下列：

- 地址 (Address)：实例地址
- 状态 (Status)：实例的运行状态
- 启动时间 (Up Time)：实例的启动时间
- 版本 (Version)：实例版本号
- Git 哈希值 (Git Hash)：实例二进制对应的 Git 哈希值
- 部署路径 (Deployment Directory)：实例二进制文件所在目录路径

实例运行状态 (Status)

实例可能处于如下任一运行状态：

- 在线 (Up)：实例正常运行。
- 离线 (Down) 或无法访问 (Unreachable)：实例未启动或对应主机存在网络问题。
- 已缩容下线 (Tombstone)：实例上的数据已被完整迁出并缩容完毕。仅 TiKV 或 TiFlash 实例存在该状态。
- 下线中 (Leaving)：实例上的数据正在被迁出并缩容。仅 TiKV 或 TiFlash 实例存在该状态。
- 未知 (Unknown)：未知的实例运行状态。

> 注意：
>
> - TiDB Dashboard 显示的 `Leaving`、 PD API 返回的 `Offline` 以及 TiUP 显示的 `Pending Offline` 这三个状态的含义相同。
> - 表格中部分列仅在实例处于在线状态 (Up) 时能显示。

实例运行状态来自于 PD 的调度信息。更详细的描述请参考TiDB 数据库的调度 - 信息收集。

14.12.1.5.3 主机列表

点击主机 (Hosts) 可查看主机列表：

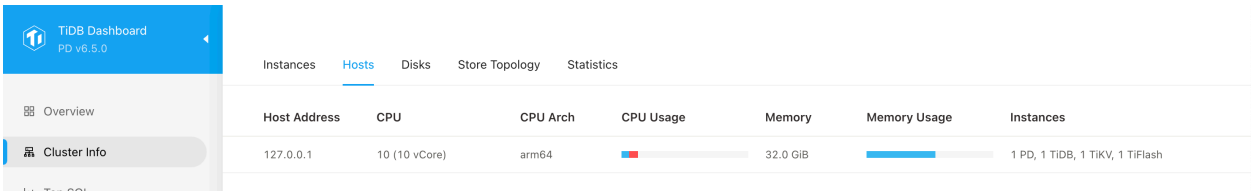


图 505: 主机

主机列表列出了该集群中 TiDB、 TiKV、 PD 和 TiFlash 组件所有实例对应主机的运行情况。

表格包含以下列：

- 主机地址 (Host Address)：主机 IP 地址
- CPU：主机 CPU 逻辑核心数
- CPU 使用率 (CPU Usage)：主机当前 1 秒的用户态和内核态 CPU 使用率
- 物理内存 (Memory)：主机总计的物理内存大小
- 内存使用率 (Memory Usage)：主机当前内存使用率

> 注意：
>
> 主机列表信息由各个实例进程给出，因此当主机上所有实例都处于离线状态时，该主机信息将无法显示。

14.12.1.5.4 磁盘列表

点击磁盘 (Disks) 可查看磁盘列表：

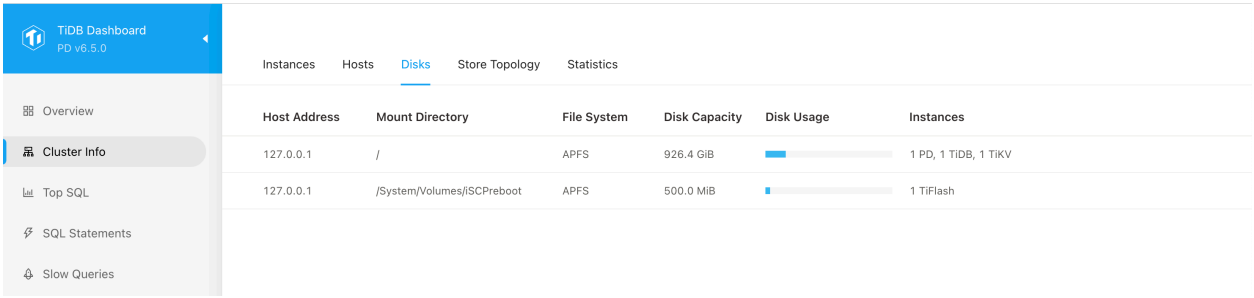


图 506: 磁盘

磁盘列表列出了该集群中 TiDB、 TiKV、 PD 和 TiFlash 组件所有实例对应主机磁盘的情况。

表格包含以下列：

- 主机地址 (Host Address)：主机 IP 地址
- 磁盘挂载点 (Mount Directory)：主机上运行实例所在磁盘的挂载路径
- 文件系统 (File System)：主机上运行实例所在磁盘的文件系统类型
- 磁盘容量 (Disk Capacity)：主机上运行实例所在磁盘的总空间大小
- 磁盘使用率 (Disk Usage)：主机上运行实例所在磁盘的空间使用率
- 实例 (Instance)：主机上运行的实例

14.12.1.6 TiDB Dashboard Top SQL 页面

TiDB Dashboard 的 Top SQL 功能允许你可视化地监控和探索数据库中各个 SQL 语句在执行过程中的 CPU 开销情况，从而对数据库性能问题进行优化和处理。Top SQL 持续收集各个 TiDB 及 TiKV 实例每秒的实时 CPU 负载等数据（按 SQL 类型分别统计），并存储至多 30 天。你可以通过 Top SQL 展示的图表及表格快速分析某个 TiDB 或 TiKV 实例在某段时间中高 CPU 负载是来自于哪些 SQL 语句。

Top SQL 具有以下功能：

- 通过图表及表格，可视化地展示 CPU 开销最多的 5 类 SQL 语句。
- 展示每秒请求数、平均延迟、查询计划等详细执行信息。
- 支持统计所有正在执行、尚未执行完毕的 SQL 语句。
- 支持查看集群中指定 TiDB 及 TiKV 实例的情况。

14.12.1.6.1 推荐适用场景

Top SQL 适用于分析性能问题。以下列举了一些典型的 Top SQL 适用场景：

- 通过监控图发现集群中有个别 TiKV 实例的 CPU 非常高，期望了解 CPU 热点来自于哪些 SQL 语句，以便对其进行优化、更好地利用上分布式资源。
- 集群整体 CPU 占用率非常高、数据库查询缓慢，期望快速知悉目前哪些 SQL 语句开销了最多的 CPU 资源，以便对它们进行优化。
- 集群整体 CPU 占用率突然发生了显著变化，期望了解变化前后主要的 SQL 资源开销区别。
- 分析集群当前最消耗资源的 SQL 语句情况，希望对它们进行优化以便降低硬件开支。

Top SQL 不能用于解答与性能无关的问题，例如数据正确性或异常崩溃问题。

当前 Top SQL 仍然处于早期阶段，功能正在持续加强。以下列举了一些目前暂不支持的场景，供参考：

- 暂时不支持分析 Top 5 以外 SQL 语句的开销情况（如多业务混合时）。
- 暂时不支持按 User、Database 等不同维度分析 Top N SQL 语句的开销情况。
- 暂时不支持分析并非由于 CPU 负载高导致的数据库性能问题，例如锁冲突。

14.12.1.6.2 访问页面

你可以通过以下任一方式访问 Top SQL 页面：

- 登录 TiDB Dashboard 后，在左侧导航栏中点击 Top SQL

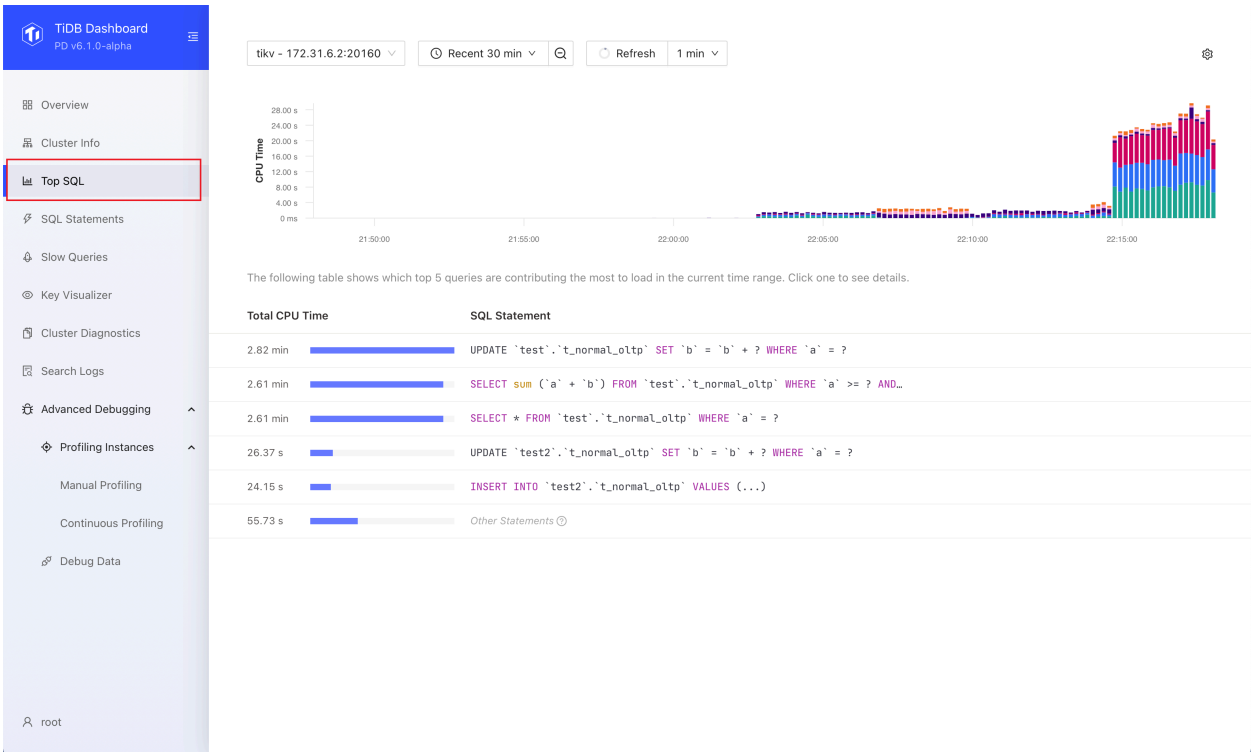


图 507: Top SQL

- 在浏览器中访问 http://127.0.0.1:2379/dashboard/#/topsql（将 127.0.0.1:2379 替换为实际 PD 实例地址和端口）。

14.12.1.6.3 启用 Top SQL

> **注意：**
>
> 要使用 Top SQL，你需要使用 TiUP（v1.9.0 及以上版本）或 TiDB Operator（v1.3.0 及以上版本）部署或升级集群。如果你已经使用旧版本 TiUP 或 TiDB Operator 进行了集群升级，请参见FAQ 进行处理。

Top SQL 开启后会对集群性能产生轻微的影响（平均 3% 以内），因此该功能默认关闭。你可以通过以下方法启用 Top SQL：

1. 访问Top SQL 页面。
2. 点击打开设置 (Open Settings)。在右侧设置 (Settings) 页面，将启用特性 (Enable Feature) 下方的开关打开。
3. 点击保存 (Save)。

你仅能看到开启功能之后的 CPU 负载细节情况，在开启功能之前的 CPU 负载细节无法在界面上呈现。另外，数据有至多 1 分钟左右的延迟，因此你可能需要等待片刻才能看到数据。

除了通过图形化界面以外，你也可以配置 TiDB 系统变量tidb_enable_top_sql 来启用 Top SQL 功能：

```
SET GLOBAL tidb_enable_top_sql = 1;
```

14.12.1.6.4 使用 Top SQL

以下是 Top SQL 的常用步骤：

1. 访问Top SQL 页面。
2. 选择一个你想要观察负载的具体 TiDB 或 TiKV 实例。

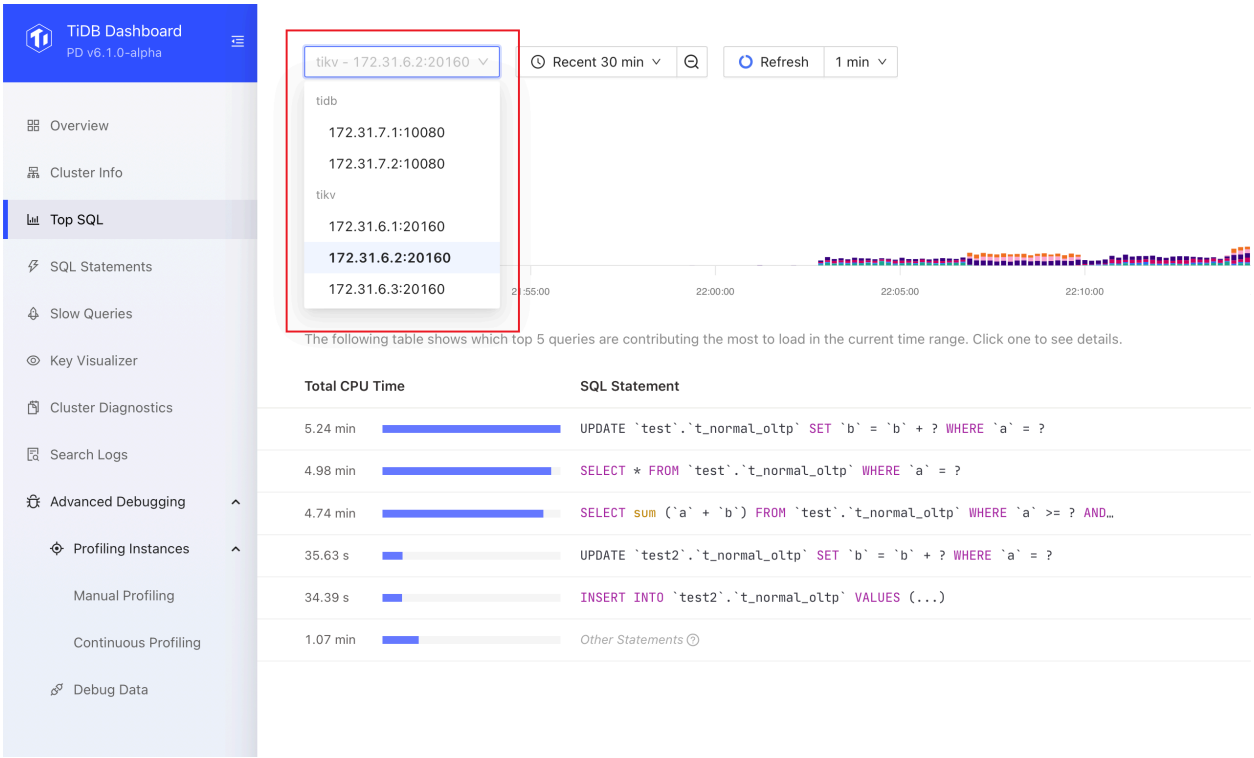


图 508: 选择实例

如果你不知道要观察哪一个 TiDB 或 TiKV 实例，可以选择任意一个实例。在集群 CPU 开销非常不均衡的情况下，你可以首先通过 Grafana 中的 CPU 监控来确定具体期望观察的实例。

3. 观察 Top SQL 呈现的 Top 5 图表及表格。

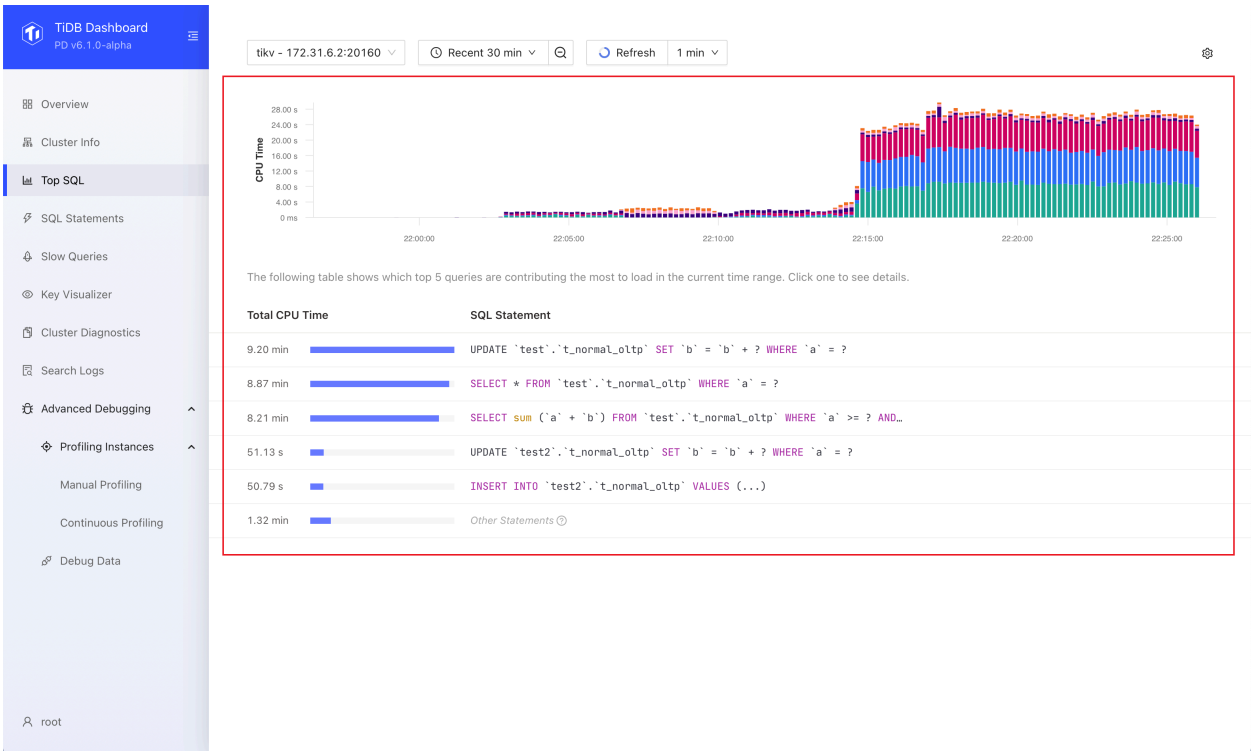


图 509: 图表表格

柱状图中色块的大小代表了 SQL 语句在该时刻消耗的 CPU 资源的多少，不同颜色区分了不同类型的 SQL 语句。大多数情况下，你都应该关注图表中相应时间范围内 CPU 资源开销较大的 SQL 语句。

4. 点击表格中的某一个 SQL 语句后，可以展开查看该语句不同执行计划的执行情况，例如 Call/sec（平均每秒请求数）、Scan Indexes/sec（平均每秒扫描索引数）等。

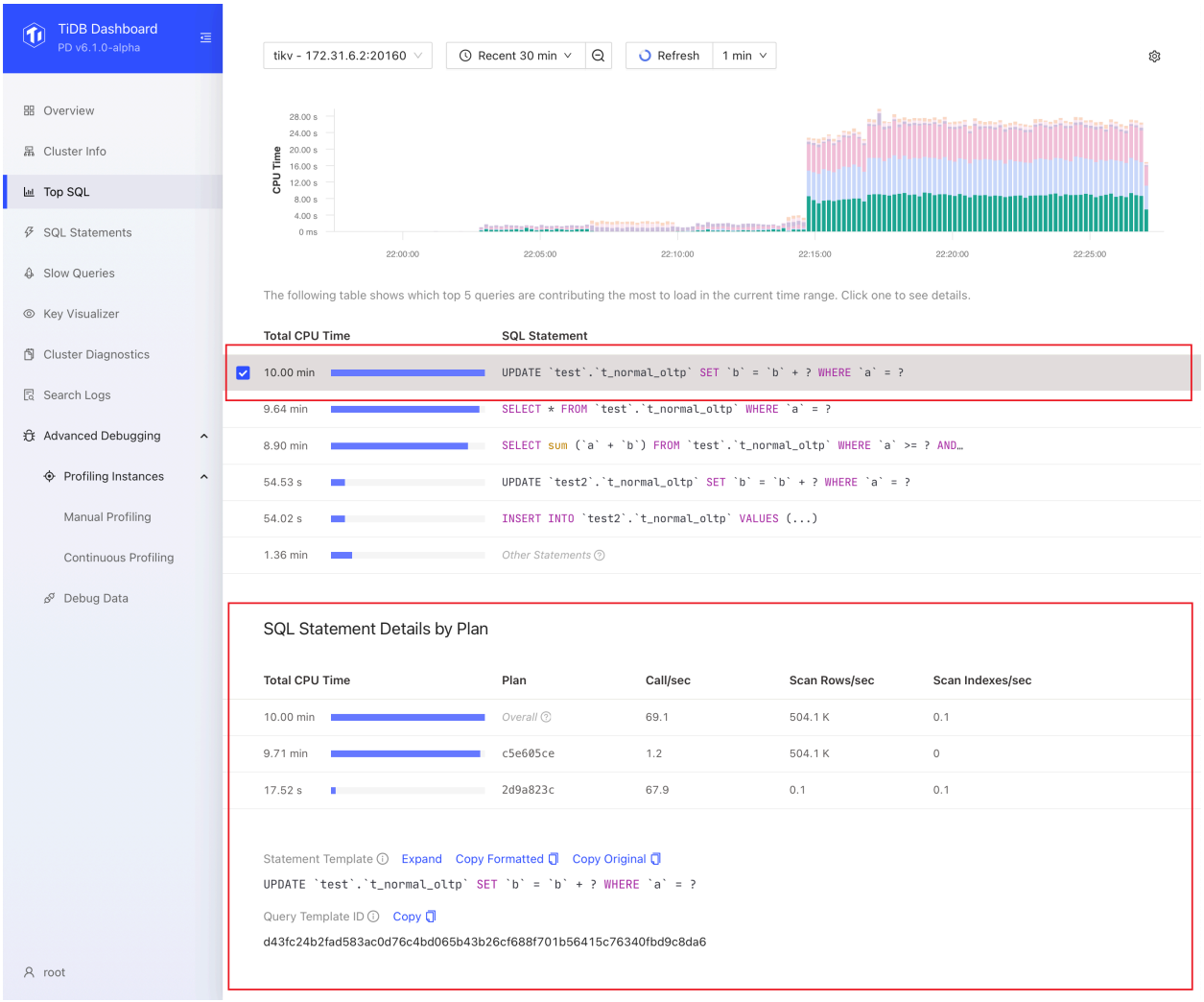


图 510: 详情

5. 基于这些初步线索，进一步在SQL 语句分析或慢查询界面中了解该 SQL 语句开销大量 CPU 资源、或扫大量数据的详细原因。

除此以外：

- 你可以在时间选择器中调整时间范围，或在图表中框选一个时间范围，来更精确、细致地观察问题。更小的时间范围将能提供细节更丰富的数据，数据最高精度可达 1 秒。



图 511: 修改时间范围

- 如果图表中显示的数据已过时，你可以点击刷新 (Refresh) 按钮，或在刷新 (Refresh) 下拉列表中选择自动刷新。

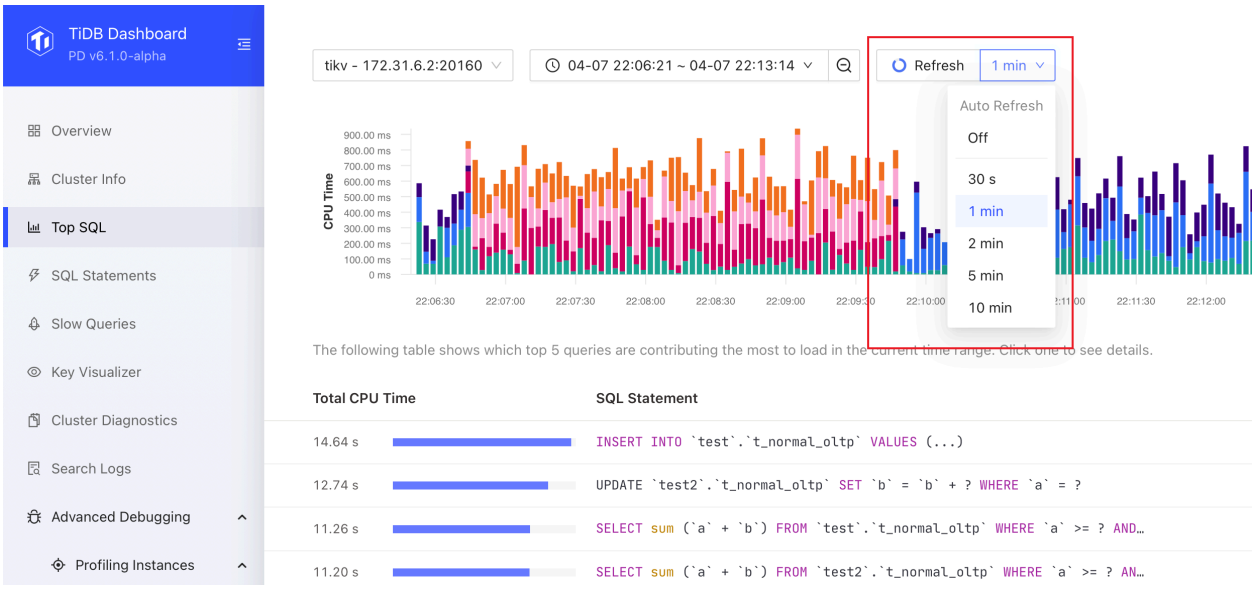


图 512: 刷新

14.12.1.6.5 停用 Top SQL

你可以通过以下步骤停用该功能：

1. 访问Top SQL 页面。
2. 点击右上角齿轮按钮打开设置界面，将启用特性 (Enable Feature) 下方的开关关闭。
3. 点击保存 (Save)。
4. 在弹出的关闭 Top SQL 功能 (Disable Top SQL Feature) 对话框中，点击确认 (Disable)。

除了通过图形化界面以外，你也可以配置 TiDB 系统变量tidb_enable_top_sql 来停用 Top SQL 功能：

```
SET GLOBAL tidb_enable_top_sql = 0;
```

14.12.1.6.6 常见问题

1. 界面上提示“集群中未启动必要组件 NgMonitoring” 无法启用功能

请参见TiDB Dashboard FAQ。

2. 该功能开启后对集群是否有性能影响？

该功能对集群性能有轻微影响。根据我们的测算，该功能对集群的平均性能影响小于 3%。

3. 该功能目前是什么状态？

该功能是正式特性，在生产环境中可用。

4. 界面中显示的其他语句（Other Statements）是什么意思？

其他所有非 Top 5 语句产生的 CPU 开销或执行情况都会被统计在该项中。你可以基于这一项了解 Top 5 的 SQL 语句开销在整体所有 SQL 语句的 CPU 开销中的比例。

5. Top SQL 展示的 CPU 开销总和与进程的实际 CPU 开销是什么关系？

它们之间有很强的相关性，但不完全一致。以 TiKV 为例，TiKV 的 CPU 开销还可能来自于其他副本的数据同步写入，这些开销不会被计入 Top SQL。但总的来说，Top SQL 中开销比例比较大的 SQL 语句实际的 CPU 资源开销也确实会更大。

6. Top SQL 图表的纵坐标是什么意思？

代表消耗 CPU 资源的多少。消耗资源越多的 SQL 语句，该值越大。在绝大多数情况下，你都不需要关心纵坐标具体数值的含义。

7. 还没有执行完毕的 SQL 语句会被统计到吗？

会。Top SQL 图表上所展示的每一时刻 CPU 开销比例即为这一时刻所有正在运行的 SQL 语句的 CPU 开销情况。

14.12.1.7 TiDB Dashboard 流量可视化页面

流量可视化页面 (Key Visualizer) 可用于分析 TiDB 集群的使用模式和排查流量热点。该页面可视化地呈现了 TiDB 集群一段时间的流量情况。

14.12.1.7.1 访问页面

可以通过以下两种方法访问 Key Visualizer 流量可视化页面：

- 登录 TiDB Dashboard 后，点击左侧导航条的流量可视化 (Key Visualizer)：

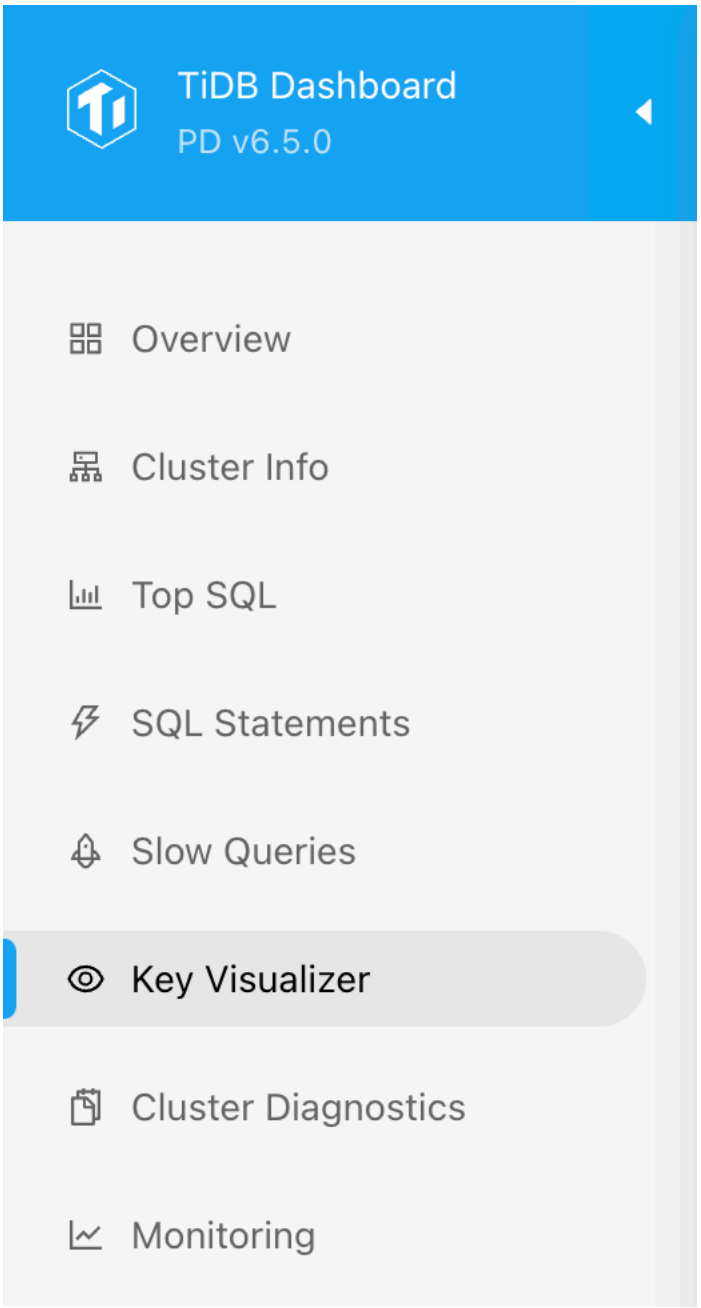


图 513: 访问

- 在浏览器中访问 `http://127.0.0.1:2379/dashboard/#/keyviz`（将 `127.0.0.1:2379` 替换为实际 PD 实例地址和端口）。

14.12.1.7.2 界面示例

流量可视化页面示例如下：

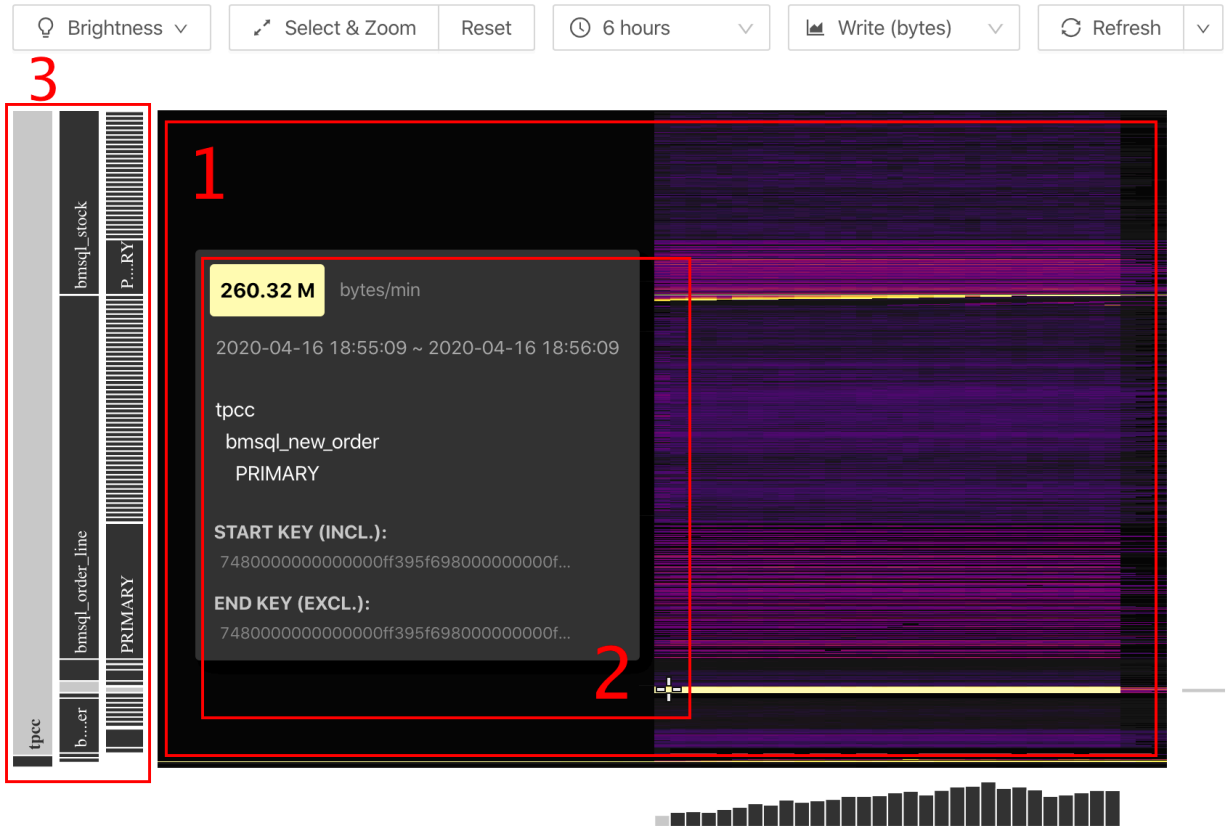


图 514: Key Visualizer 示例图

从以上流量可视化界面可以观察到以下信息：

- 一个大型热力图，显示整体访问流量随时间的变化情况。
- 热力图某个坐标的详细信息。
- 左侧为表、索引等标识信息。

14.12.1.7.3 基本概念

本节介绍流量可视化涉及的一些基本概念。

Region

在 TiDB 集群中，数据以分布式的方式存储在所有的 TiKV 实例中。TiKV 在逻辑上是一个巨大且有序的 KV Map。整个 Key-Value 空间分成很多 Region，每一个 Region 是一系列连续的 Key。

> 注意：
>
> 关于 Region 的详细介绍，请参考三篇文章了解 TiDB 技术内幕 - 说存储

热点

在使用 TiDB 的过程中，热点是一个典型的现象，它表现为大量的流量都在读写一小块数据。由于连续的数据往往由同一个 TiKV 实例处理，因此热点对应的 TiKV 实例的性能就成为了整个业务的性能瓶颈。常见的热点场景有使用自增主键连续写入相邻数据导致的写入表数据热点、时间索引下写入相邻时间数据导致的写入表索引热点等。

> 注意：
>
> 热点问题详情请参阅 TiDB 热点问题详解。

热力图

热力图是流量可视化页面的核心，它显示了一个指标随时间的变化。热力图的横轴 X 是时间，纵轴 Y 则是按 Key 排序的连续 Region，横跨 TiDB 集群上所有数据库和数据表。颜色越暗 (cold) 表示该区域的 Region 在这个时间段上读写流量较低，颜色越亮 (hot) 表示读写流量越高，即越热。

Region 压缩

一个 TiDB 集群中，Region 的数量可能多达数十万。在屏幕上是难以显示这么多 Region 信息的。因此，在一张热力图中，Region 会被压缩到约 1500 个连续范围，每个范围称为 Bucket。在一个热力图上，热的实例更需要关注，因此 Region 压缩总是倾向于将流量较小的大量 Region 压缩为一个 Bucket，而尽量让高流量的 Region 独立成 Bucket。

14.12.1.7.4 使用介绍

本节介绍如何使用流量可视化页面。

设置

首次使用流量可视化页面需要先通过设置页面手动开启此功能。参考页面指引，点击打开设置 (Open Settings) 即可打开设置页面：

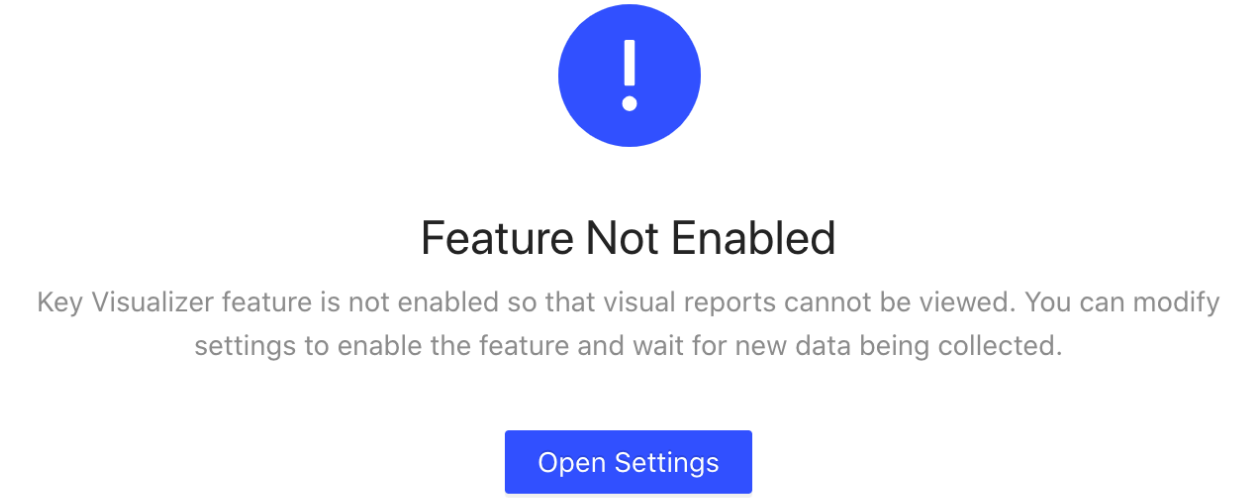


图 515: 功能未开启

在功能已经开启时，可通过右上角的设置图标打开设置页面：

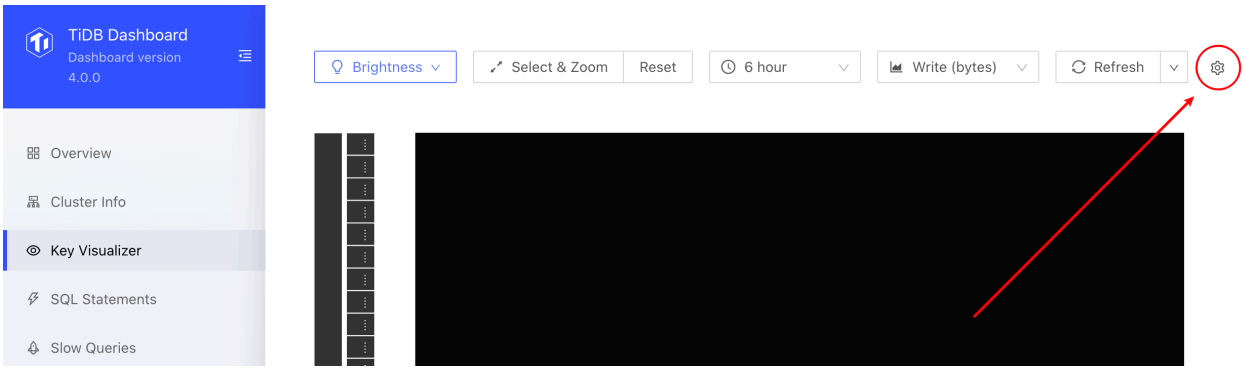


图 516: 设置按钮

设置页面如下图所示：

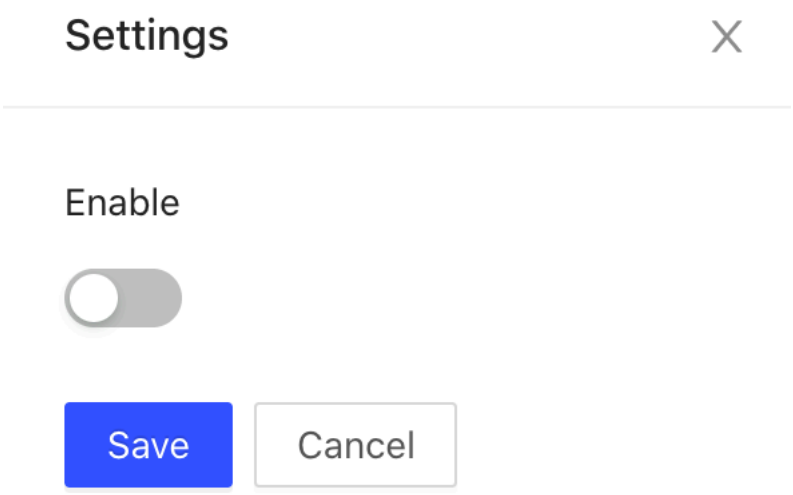


图 517: 设置页面

通过开关设定好是否开启收集，并点击保存 (Save) 后生效。开启后，在界面上观察到工具栏已经可以使用：

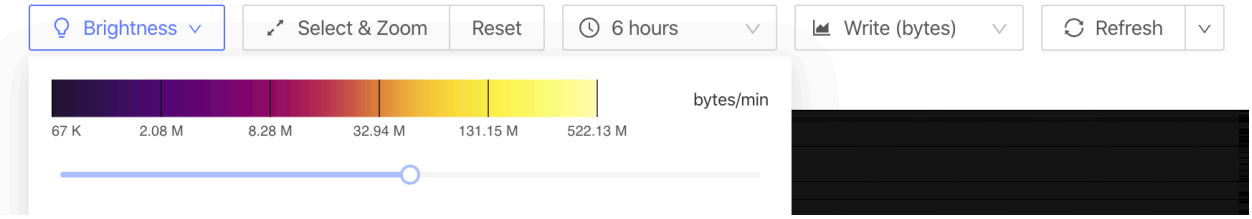


图 518: 工具栏

功能开启后，后台会持续收集数据，稍等一段时间即可看到热力图。

观察某一段时间或者 Region 范围

打开流量可视化页面时，默认会显示最近六小时整个数据库内的热力图。其中，越靠近右侧（当前时间）时，每列 Bucket 对应的时间间隔会越小。如果你想观察某个特定时间段或者特定的 Region 范围，则可以通过放大来获得更多细节。具体操作描述如下：

- 在热力图中向上或向下滚动。
- 点击框选 (Select & Zoom) 按钮，然后点击并拖动以选择要放大的区域。

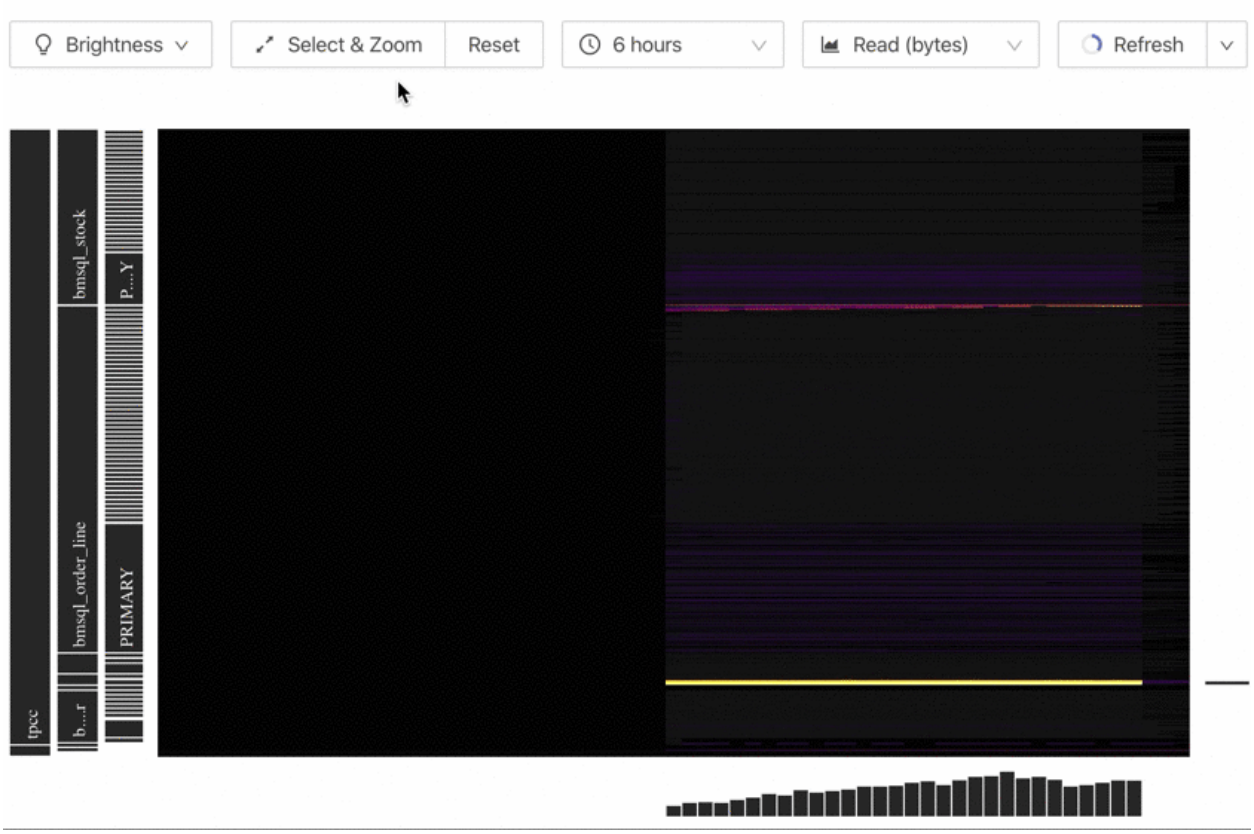


图 519: 框选

- 点击重置 (Reset) 按钮，将 Region 范围重置为整个数据库。
- 点击「时间选择框」(界面的 6 hour 处)，重新选择观察时间段。

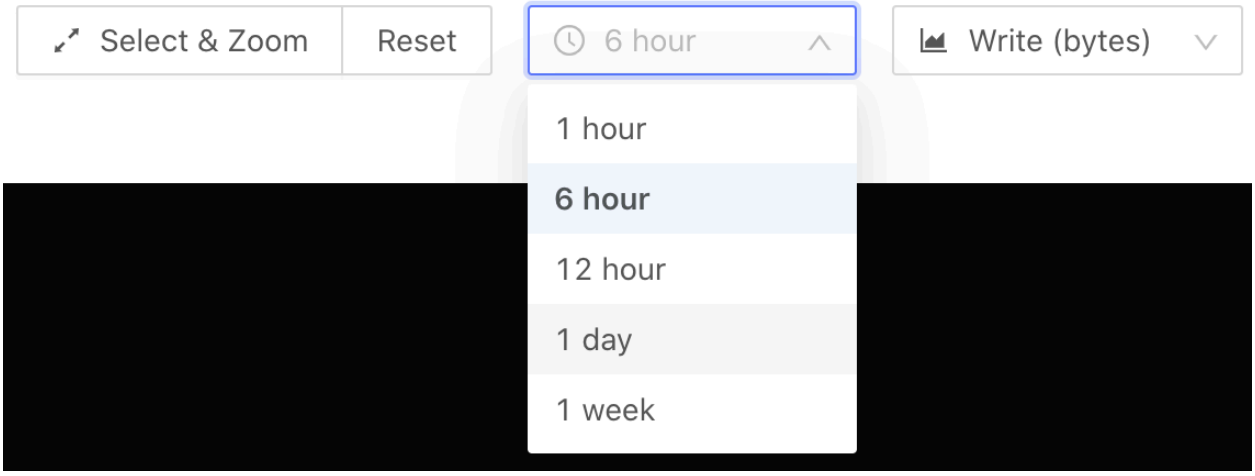


图 520: 时间选择

> 注意：
>
> 进行后三种操作，将引起热力图的重新绘制。你可能观察到热力图与放大前有较大差异。这是一个正常的现象，可能是由于在进行局部观察时，Region 压缩的粒度发生了变化，或者是局部范围内，“热”的基准发生了改变。

调整亮度

热力图使用颜色的明暗来表达一个 Bucket 的流量高低，颜色越暗 (cold) 表示该区域的 Region 在这个时间段上读写流量较低，颜色越亮 (hot) 表示读写流量越高，即越热。如果热力图中的颜色太亮或太暗，则可能很难观察到细节。此时，可以点击调整亮度 (Brightness) 按钮，然后通过滑块来调节页面的亮度。

> 注意：
>
> 在显示一个区域内的热力图时，会根据区域内的流量情况来界定冷热。当整个区域流量较为平均时，即使整体流量在数值上很低，你依然有可能观察到较大的亮色区域。请注意一定要结合数值一起分析。

选择指标

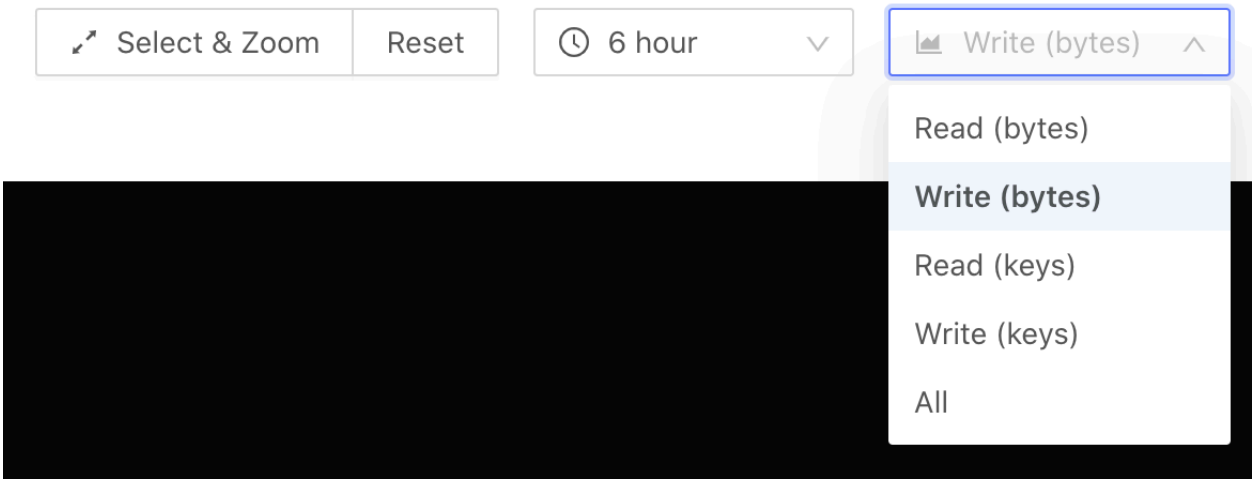


图 521: 指标选择

你可以通过「指标选择框」(以上界面中 Write (bytes) 处) 来查看你关心的指标：

- `Read (bytes)`：读取字节量
- `Write (bytes)`：写入字节量
- `Read (keys)`：读取次数

- `Write (keys)`：写入次数
- `All`：所有（读写流量的总和）

刷新与自动刷新

可以通过点击刷新 (Refresh) 按钮来重新获得基于当前时间的热力图。当需要实时观察数据库的流量分布情况时，可以点击按钮右侧的向下箭头，选择一个固定的时间间隔让热力图按此间隔自动刷新。

> 注意：
>
> 如果进行了时间范围或者 Region 范围的调整，自动刷新会被关闭。

查看详情

可以将鼠标悬停在你所关注的 Bucket 上，来查看这个区域的详细信息：

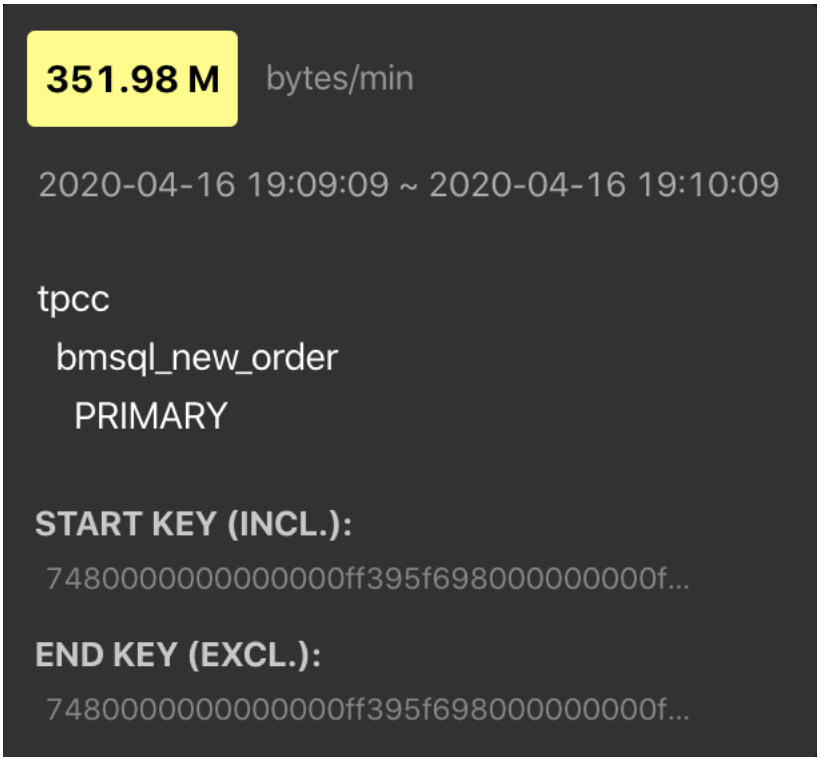


图 522: Bucket 详细信息

如果需要复制某个信息，可以进行点击 Bucket。此时相关详细信息的页面会被暂时钉住。点击你关注的信息，即可将其复制到剪切板：

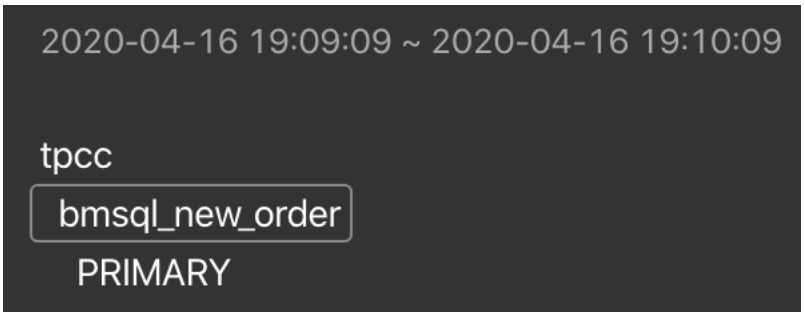


图 523: 复制 Bucket 详细信息

14.12.1.7.5 常见热力图解读

本章节选取了 Key Visualizer 中常见的四种热力图进行解读。

均衡：期望结果

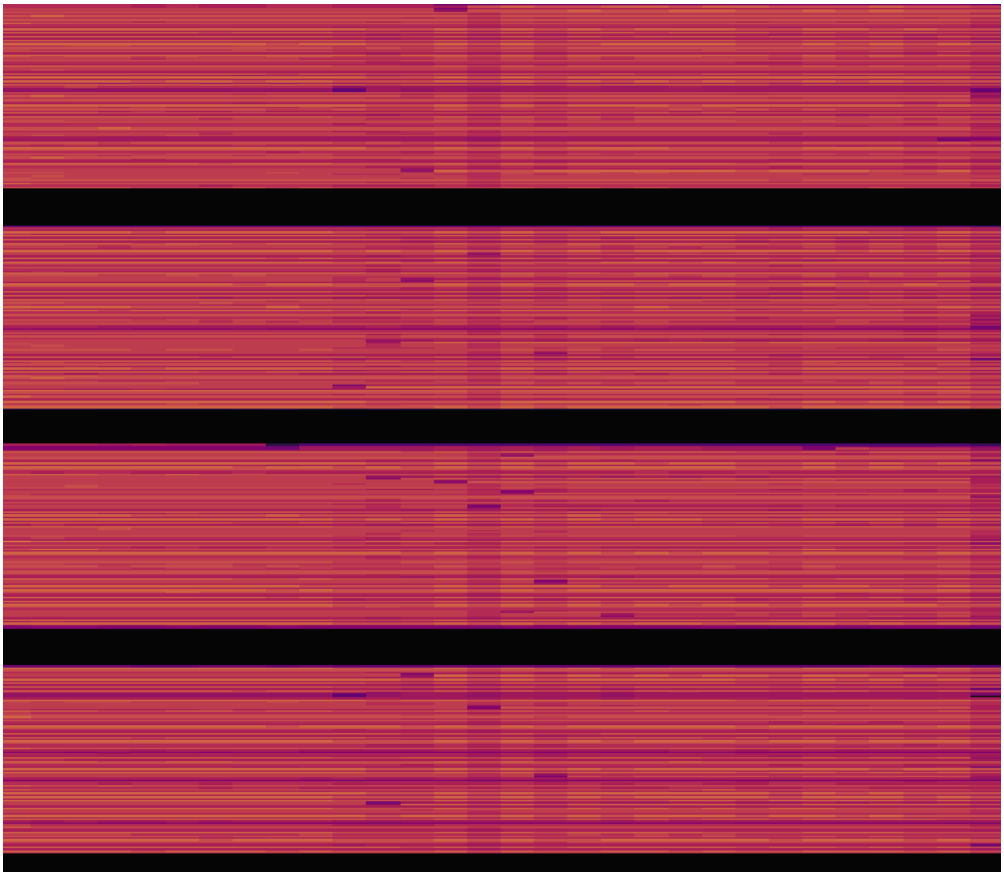


图 524: 均衡结果图

如上图所示，热力图颜色均匀或者深色和亮色混合良好，说明读取或写入在时间和 Region 空间范围上都分布得比较均衡，访问压力均匀地分摊在所有的机器上。这种负载是最适合分布式数据库的。

X 轴明暗交替：需要关注高峰期的资源情况

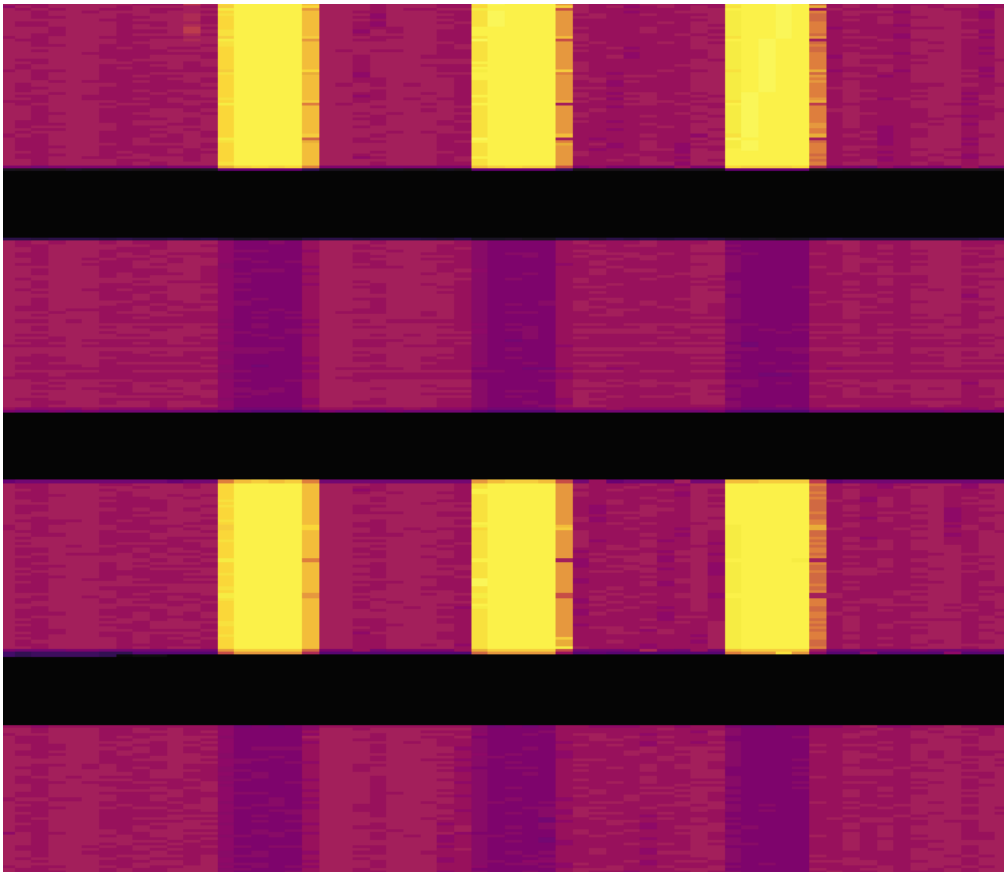


图 525: X 轴明暗交替

如上图所示，热力图在 X 轴（时间）上表现出明暗交替，但 Y 轴 (Region) 则比较均匀，说明读取或写入负载具有周期性的变化。这种情况可能出现在周期性的定时任务场景，如大数据平台每天定时从 TiDB 中抽取数据。一般来说可以关注一下使用高峰时期资源是否充裕。

Y 轴明暗交替：需要关注产生的热点聚集程度

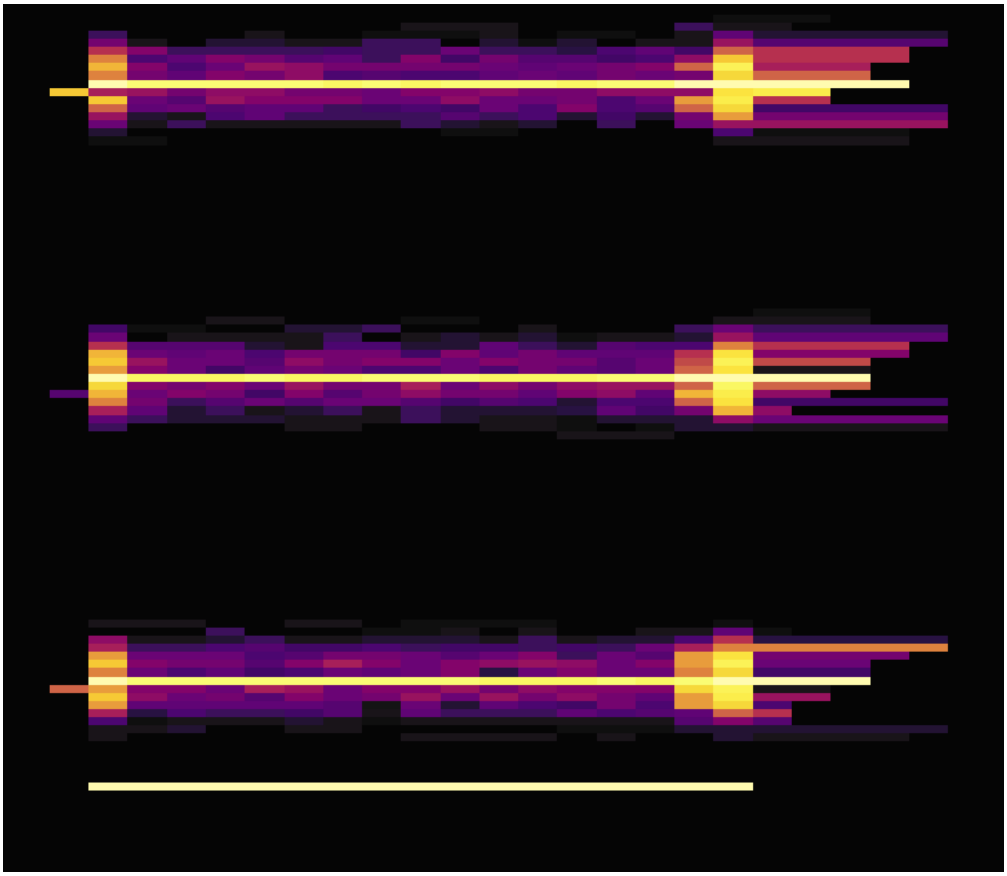


图 526: Y 轴明暗交替

如上图所示，热力图包含几个明亮的条纹，从 Y 轴来看条纹周围都是暗的，这表明明亮条纹区域的 Region 有很高的读写流量，可以从业务角度观察一下是否符合预期。例如，所有业务都关联用户表的情况下，用户表的整体流量就会很高，那么在热力图中表现为亮色区域就非常合理。

另外，明亮区域的高度（Y 轴方向的粗细）非常关键。由于 TiKV 自身拥有以 Region 为单位的热点平衡机制，因此涉及热点的 Region 越多其实越能有利于在所有 TiKV 实例上均衡流量。明亮条纹越粗、数量越多则意味着热点越分散、更多的 TiKV 能得到利用；明亮条纹越细、数量越少意味着热点越集中、热点 TiKV 越显著、越需要人工介入并关注。

明亮斜线：需要关注业务模式

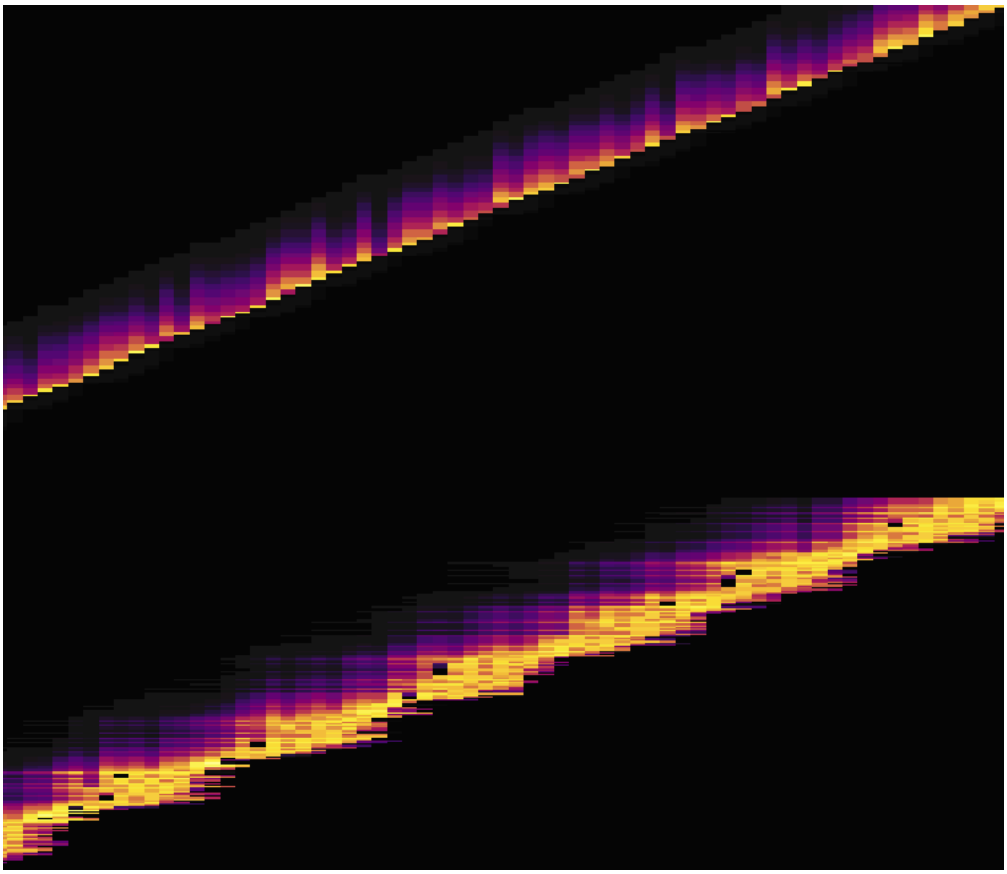


图 527: 明亮斜线

如上图所示，热力图显示了明亮的斜线，表明读写的 Region 是连续的。这种场景常常出现在带索引的数据导入或者扫描阶段。例如，向自增 ID 的表进行连续写入等等。图中明亮部分对应的 Region 是读写流量的热点，往往会成为整个集群的性能问题所在。这种时候，可能需要业务重新调整主键，尽可能打散以将压力分散在多个 Region 上，或者选择将业务任务安排在低峰期。

> **注意：**
>
> 这里只是列出了几种常见的热力图模式。流量可视化页面中实际展示的是整个集群上所有数据库、数据表的热力图，因此非常有可能在不同的区域观察到不同的热力图模式，也可能观察到多种热力图模式的混合结果。使用的时候应当视实际情况灵活判断。

14.12.1.7.6 解决热点问题

TiDB 内置了不少帮助缓解常见热点问题的功能，深入了解请参考TiDB 高并发写入场景最佳实践。

14.12.1.8 TiDB Dashboard 监控关系图

TiDB Dashboard 监控关系图是 TiDB v4.0.7 起提供的新功能，可以将集群中各个内部流程的耗时监控数据绘制为关系图，帮助用户快速了解集群中各个环节的耗时及关系。

14.12.1.8.1 访问关系图

登录 TiDB Dashboard 后点击左侧导航的集群诊断 (Cluster Diagnostics) 可以进入此功能页面：

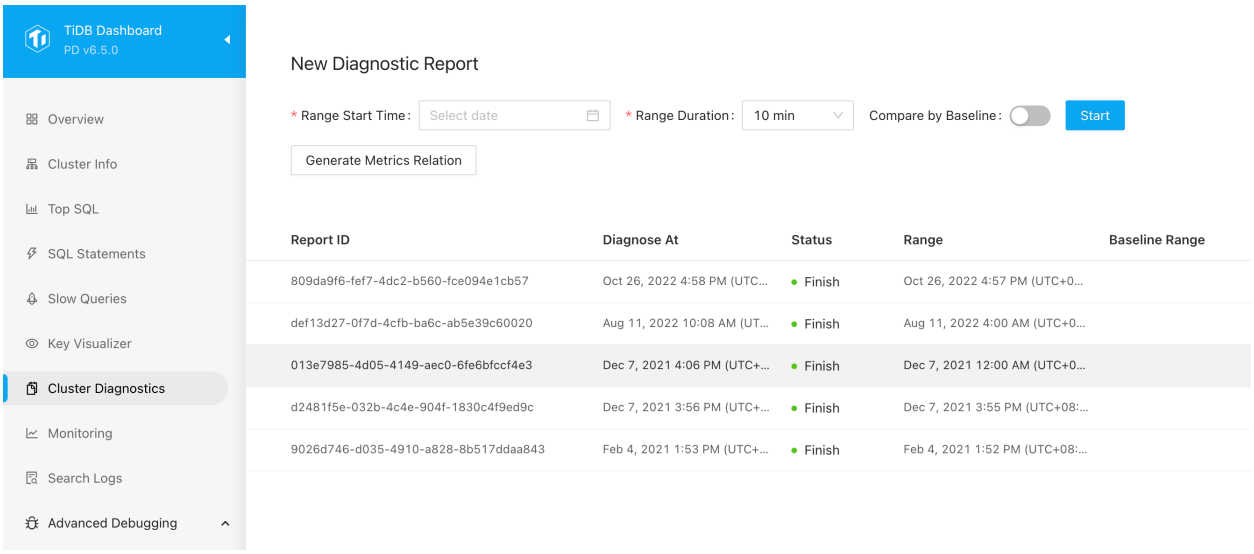


图 528: 生成监控关系图首页

设置区间起始时间 (Range Start Time) 和区间长度 (Range Duration) 参数后，点击生成监控关系图 (Generate Metrics Relation) 按钮后，会进入监控关系图页面。

14.12.1.8.2 关系图解读

下面是一份监控耗时关系图示例，描述的是某个 TiDB 集群在 2020-07-29 16:36:00 开始往后 5 分钟内，TiDB 集群中各个监控的总耗时比例，以及各项监控之间的关系。

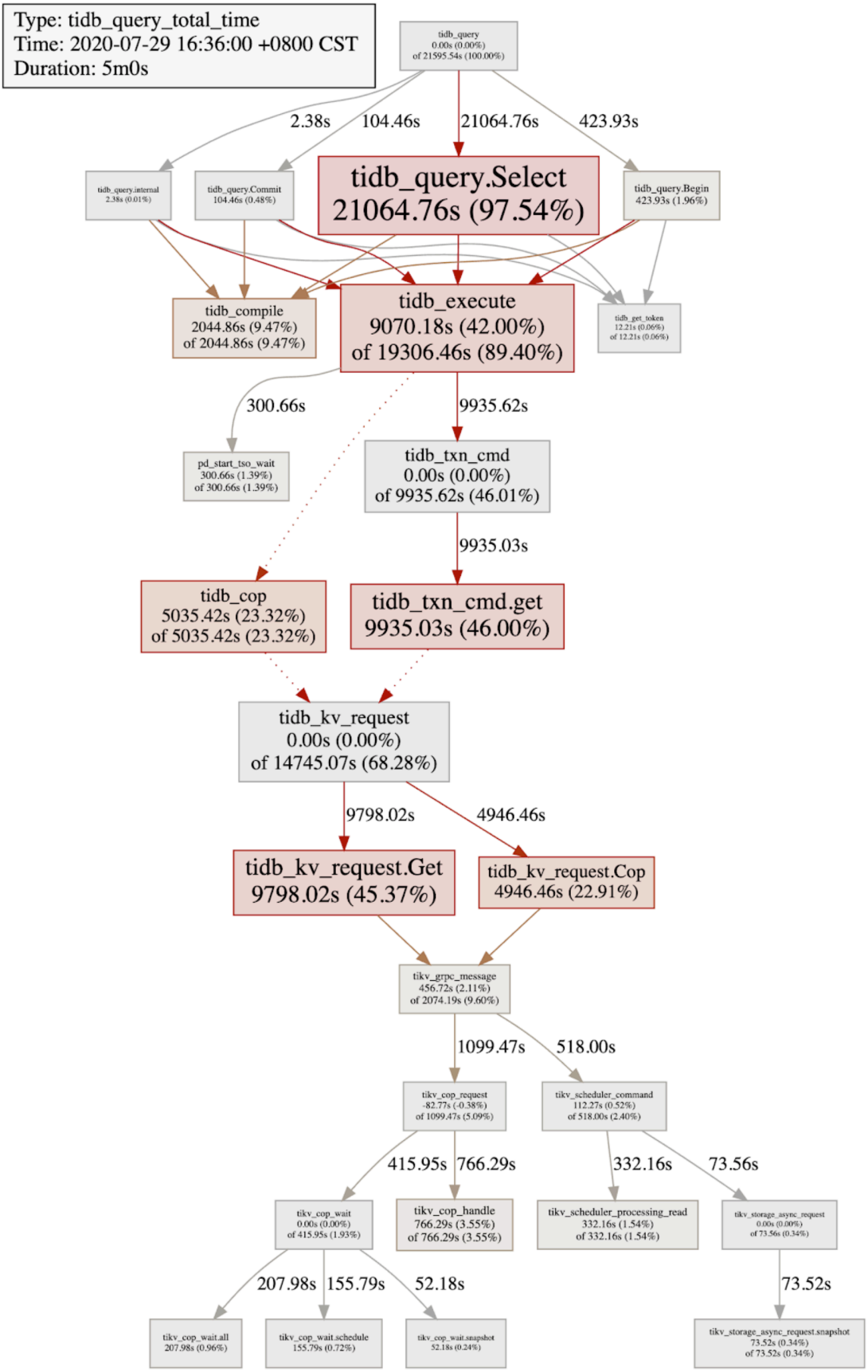


图 529: 监控关系图示例

例如以下 tidb_execute 节点监控图表示：tidb_execute 监控项的总耗时为 19306.46 秒，占总查询耗时的 89.4%，其中 tidb_execute 节点自身的耗时是 9070.18 秒，占总查询耗时的 42%。将鼠标悬停在该方框上，可以看到监控项的注释说明，总耗时、平均耗时、平均 P99 耗时等详细信息。

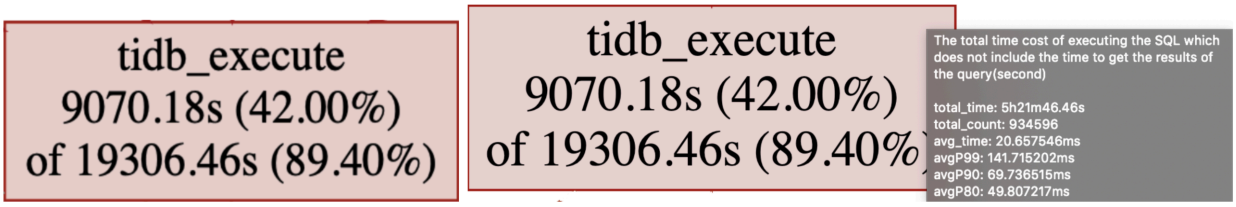


图 530: 监控关系图 tidb_execute 节点示例

节点的含义

每个方框节点代表一个监控项，包含了以下信息：

- 监控项的名称
- 监控项的总耗时
- 监控项总耗时和查询总耗时的比例

节点监控的总耗时 = 节点自身的耗时 + 子节点的耗时，所以某些节点监控图会显示节点自身的耗时和总耗时的比例。例如 tidb_execute 监控：

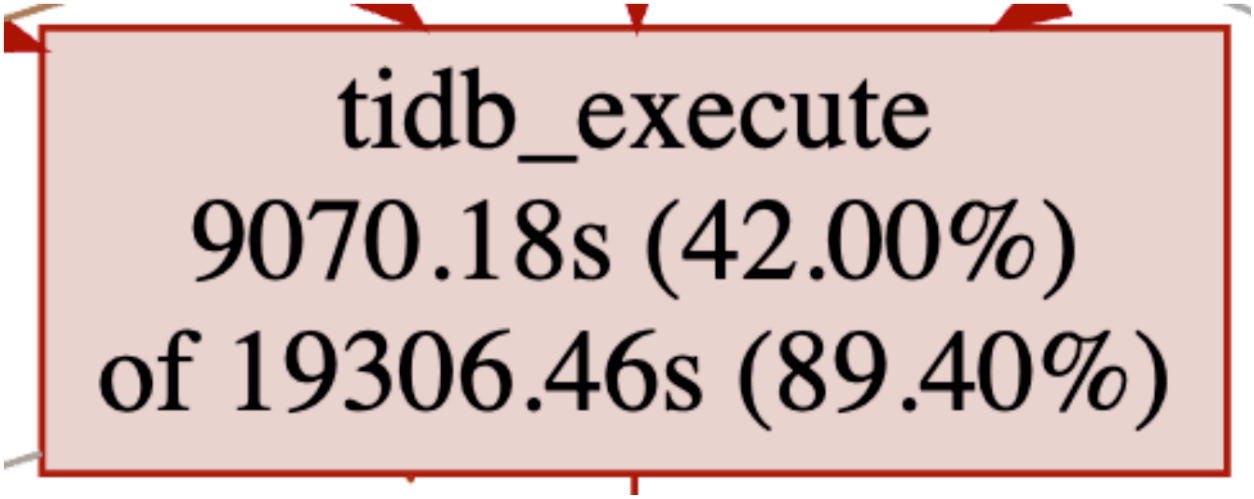


图 531: 监控关系图 tidb_execute 节点示例

- tidb_execute 是监控项的名字。该监控是指一条 SQL 请求在 TiDB 执行引擎中的执行耗时。
- 19306.46s 表示 tidb_execute 监控项消耗的总时间为 19306.46 秒。89.40% 表示 19306.46 秒占所有 SQL 查询总耗时（包括用户 SQL 和 TiDB 内部的 SQL）的比例为 89.40%。查询总耗时是 tidb_query 监控项的总耗时。

- `9070.18s` 表示 `tidb_execute` 节点自身总执行耗时是 9070.18 秒，其余部分是被其子节点消耗的时间。`42.00%` 表示 9070.18 秒占所有查询总耗时的比例为 42.00%。

将鼠标悬停在该节点后，会显示监控项的更多详细信息：

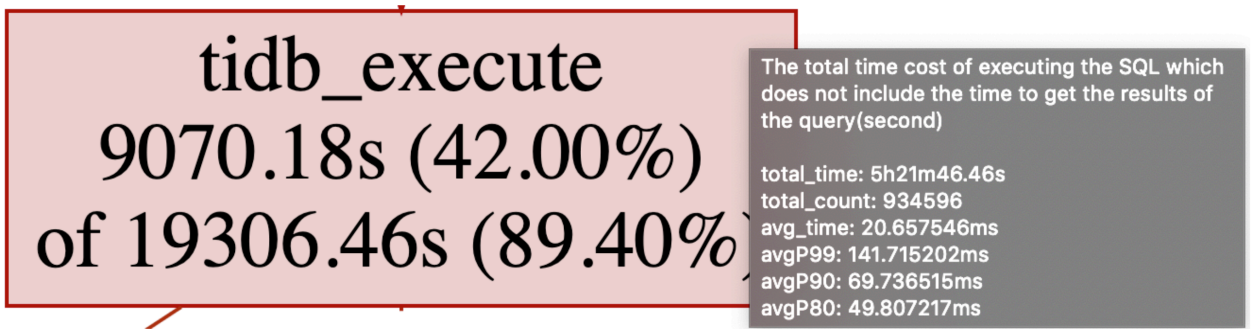


图 532: 监控关系图 tidb_execute 节点注释

上图信息为该项监控的注释说明，包括总耗时、总次数、平均耗时和平均 P99、P90、P80 耗时。

监控项之间的父子关系

下面以 `tidb_execute` 监控为例介绍该监控项相关的子节点：

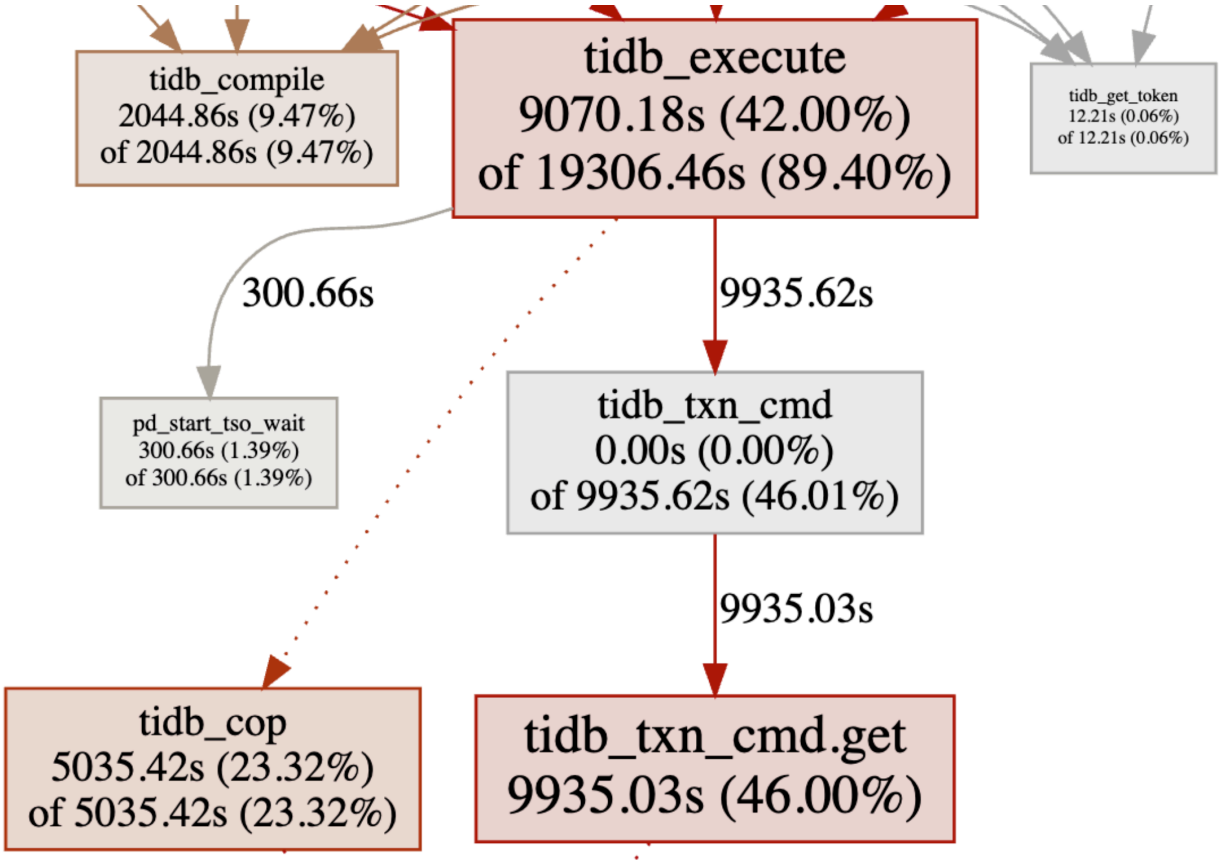


图 533: 监控关系图 tidb_execute 节点注释

可以看到，tidb_execute 包含两个子节点，分别是：

- pd_start_tso_wait：等待事务 start_tso 的耗时，其总耗时是 300.66 秒。
- tidb_txn_cmd：TiDB 执行事务相关命令的耗时，其总耗时是 9935.62 秒。

另外，tidb_execute 还有一条虚线箭头指向 tidb_cop 监控，这里虚线箭头的含义如下：

tidb_execute 包含 tidb_cop 监控的耗时，但是 cop 请求有可能并发执行。例如对两个表的进行 join 查询的 execute 耗时为 60 秒，其中 join 的两个表会并行地执行 cop 扫表请求。假如 cop 请求执行时间分别为 40 秒和 30 秒，那 cop 请求的总耗时是 70 秒，但是 execute 执行耗时只有 60 秒。所以如果父节点的耗时不完全包含子节点的耗时，就用虚线箭头来指向子节点。

> 注意：
>
> 当节点有虚线箭头指向的子节点时，该节点的本身的耗时是不准确的。例如 tidb_execute 监控中，tidb_execute 节点本身的耗时为 9070.18 = 19306.46 - 300.66 - 9935.62。这里 tidb_cop 节点的耗时并不会计入子节点耗时的计算，但实际上，tidb_execute 监控本身的耗时 9070.18 秒中包含了 tidb_cop 一部分监控节点的耗时，但无法确认具体包含了多少耗时。

tidb_kv_request 及其父节点

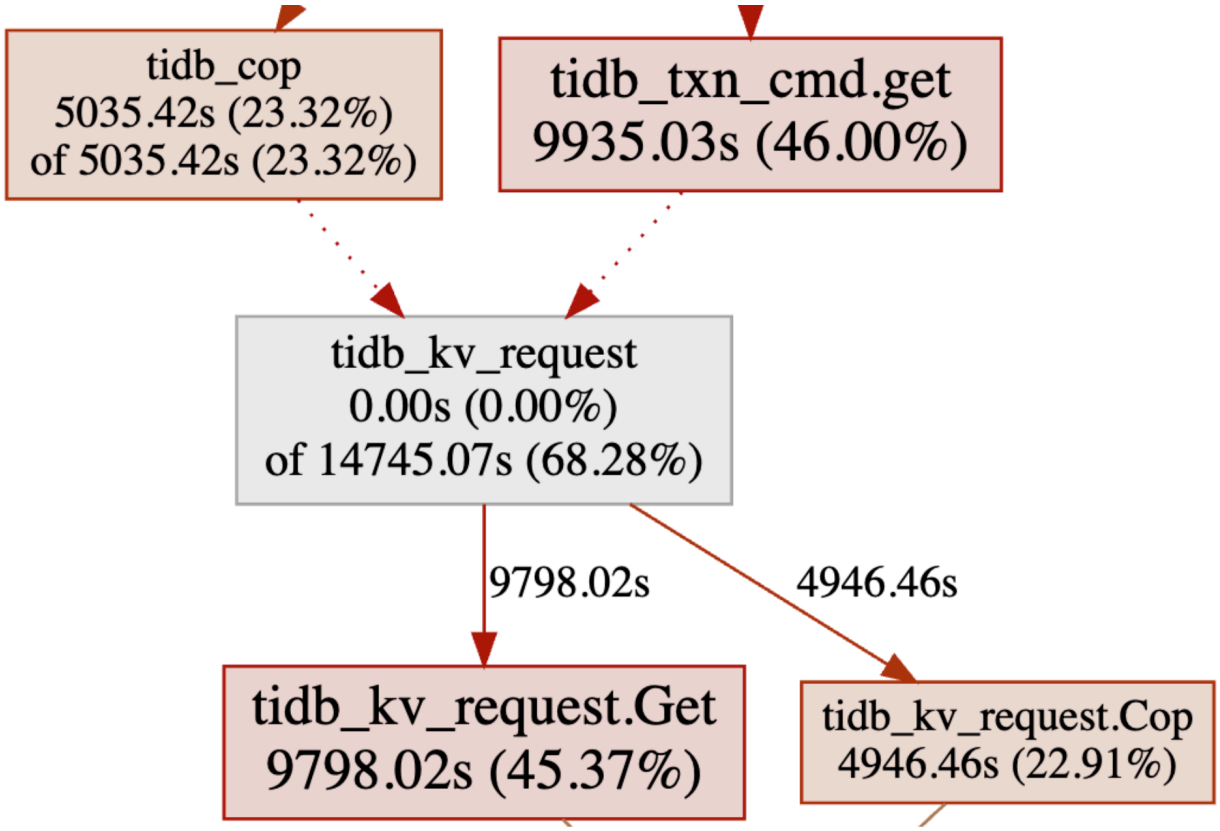


图 534: 监控关系图虚线节点关系

tidb_kv_request 的父节点 tidb_cop 和 tidb_txn_cmd.get 都用虚拟箭头指向 tidb_kv_request，这里表示：

- tidb_cop 的耗时包含部分 tidb_kv_request 的耗时
- tidb_txn_cmd.get 的耗时也包含部分的 tidb_kv_request 的耗时。

但是 tidb_cop 具体有多少耗时是 tidb_kv_request 消耗的，无法进行确认。

- tidb_kv_request.Get：TiDB 发送 Get 类型的 kv 请求的耗时。
- tidb_kv_request.Cop：TiDB 发送 Cop 类型的 kv 请求的耗时。

tidb_kv_request 与 tidb_kv_request.Get 和 tidb_kv_request.Cop 并不是父节点包含子节点的关系，而是组成关系。子节点的名称前缀是父节点的名称加上 .xxx，即为父节点的子类。这里可以理解为，TiDB 发送 kv 请求的总耗时为 14745.07 秒，其中 Get 和 Cop 类型的 kv 请求的总耗时分别为 9798.02 秒和 4946.46 秒。

14.12.1.9 SQL 语句分析

14.12.1.9.1 TiDB Dashboard SQL 语句分析执行详情页面

该页面可以查看所有 SQL 语句在集群上执行情况，常用于分析总耗时或单次耗时执行耗时较长的 SQL 语句。

在该页面中，结构一致的 SQL 查询（即使查询参数不一致）都会被归为同一个 SQL 语句，例如 `SELECT * FROM employee WHERE id IN (1, 2, 3)` 和 `select * from EMPLOYEE where ID in (4, 5)` 都属于同一 SQL 语句 `select * from employee where id in (...)`。

访问列表页面

可以通过以下两种方法访问 SQL 语句分析页面：

- 登录后，左侧导航条点击“SQL 语句分析”(SQL Statements)。

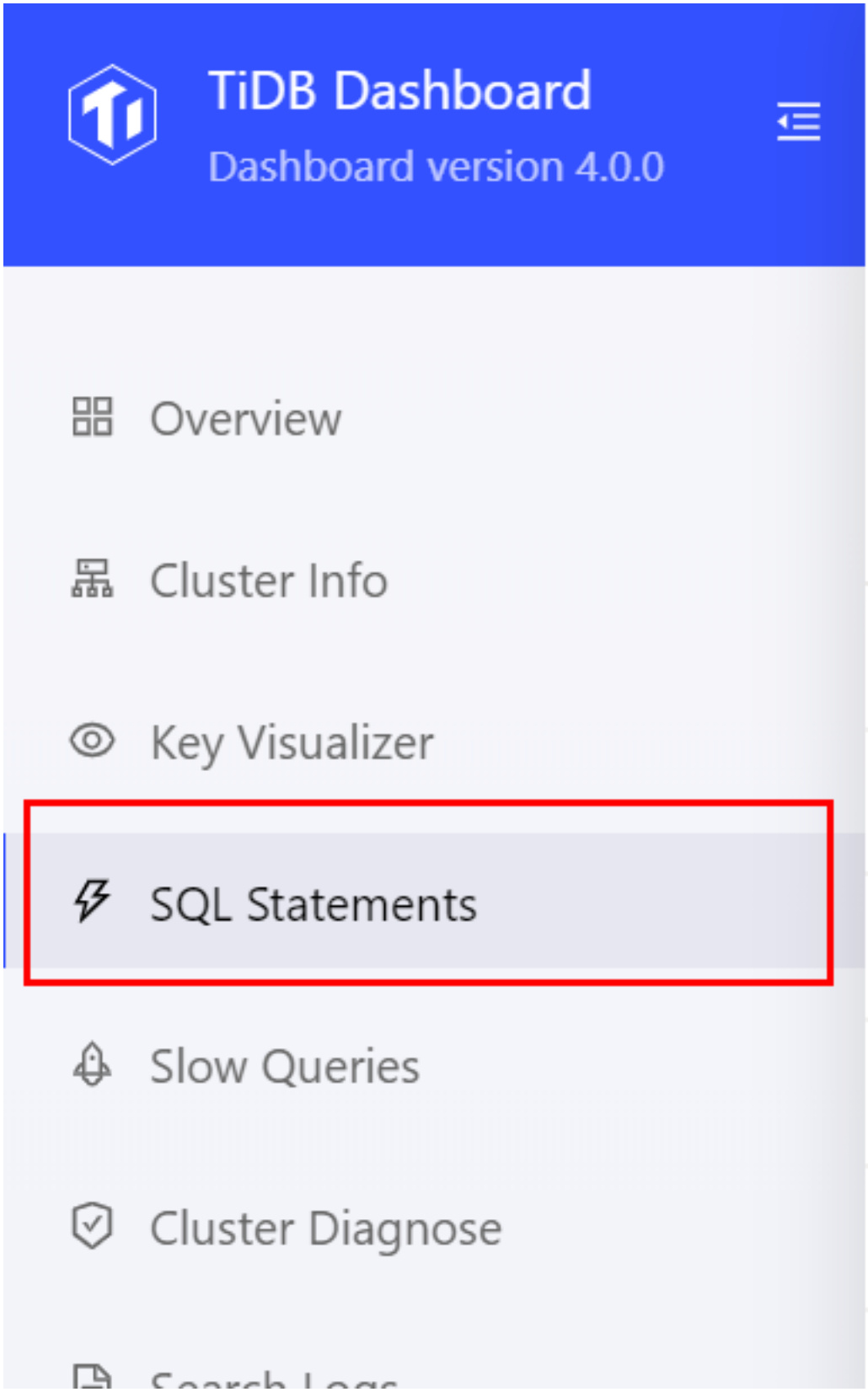

TiDB Dashboard
Dashboard version 4.0.0
Overview
Cluster Info
Key Visualizer
SQL Statements
Slow Queries
Cluster Diagnose

图 535: 访问

- 在浏览器中访问 http://127.0.0.1:2379/dashboard/#/statement（将 127.0.0.1:2379 替换为实际 PD 实例地址和端口）。

SQL 语句分析页面所展示的所有数据都来自于 TiDB Statement 系统表，参见Statement Summary Tables 文档了解该系统表的详细情况。

> **注意：**
>
> 在 SQL 语句分析页面的 Mean Latency 列中，蓝色条表示平均耗时。如果某条 SQL 语句的蓝色条上有黄色的线，黄色线的左右两侧分别代表最近一个收集周期内该 SQL 语句耗时的最小值和最大值。

修改列表过滤条件

页面顶部可修改显示的时间范围、按执行 SQL 所在数据库进行过滤，或按 SQL 类型进行过滤，如下所示。默认显示最近一个收集周期（默认最近 30 分钟）内的所有 SQL 语句执行情况。

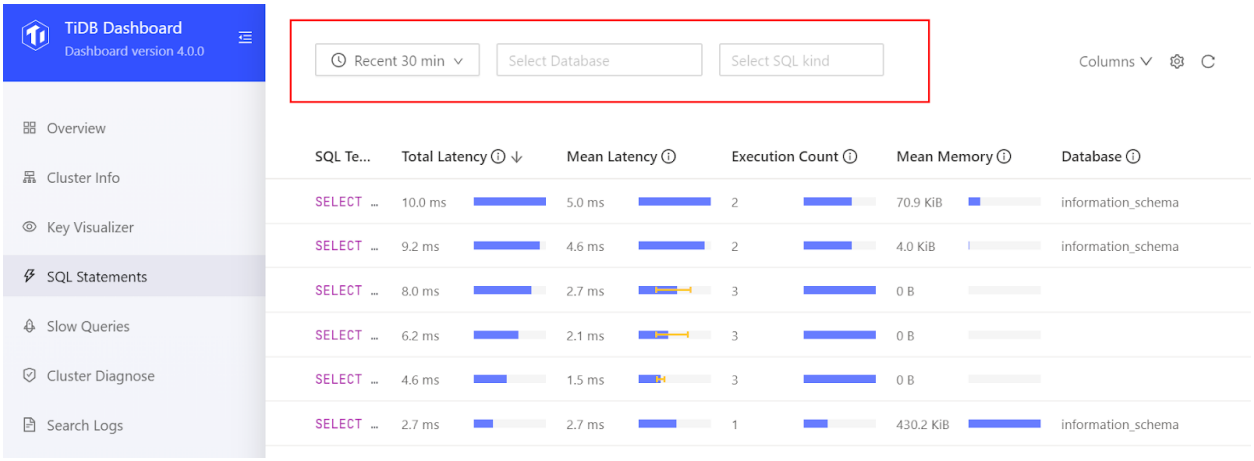


图 536: 修改过滤条件

显示其他字段

页面顶部选择列 (Columns) 选项中可选择显示更多列，可将鼠标移动到列名右侧的 (i) 图标处查看列的说明：

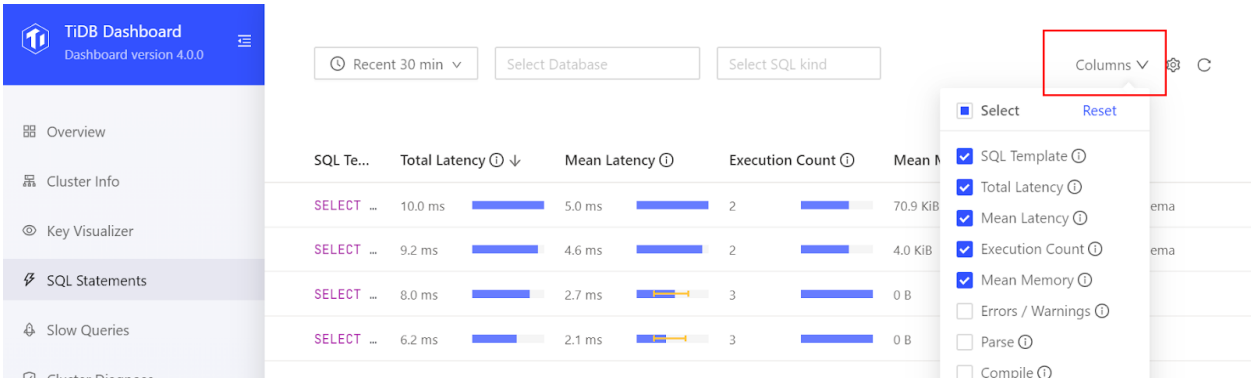


图 537: 选择列

修改列表排序依据

列表默认以累计耗时 (Total Latency) 从高到低进行排序，点击不同的列标题可以修改排序依据或切换排序顺序：

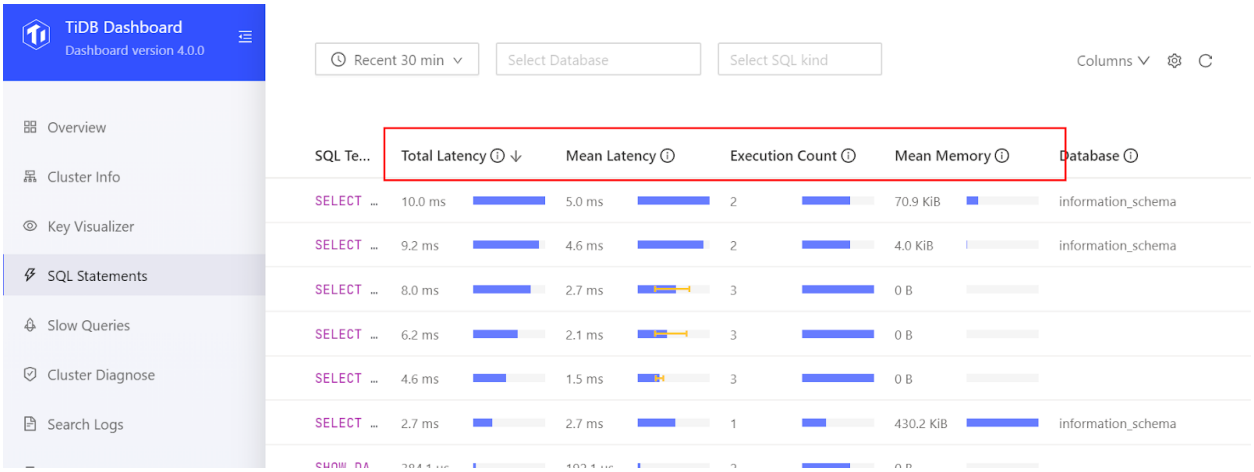


图 538: 修改列排序

修改数据收集设置

在列表页面，点击顶部右侧的设置 (Settings) 按钮，即可对 SQL 语句分析功能进行设置：

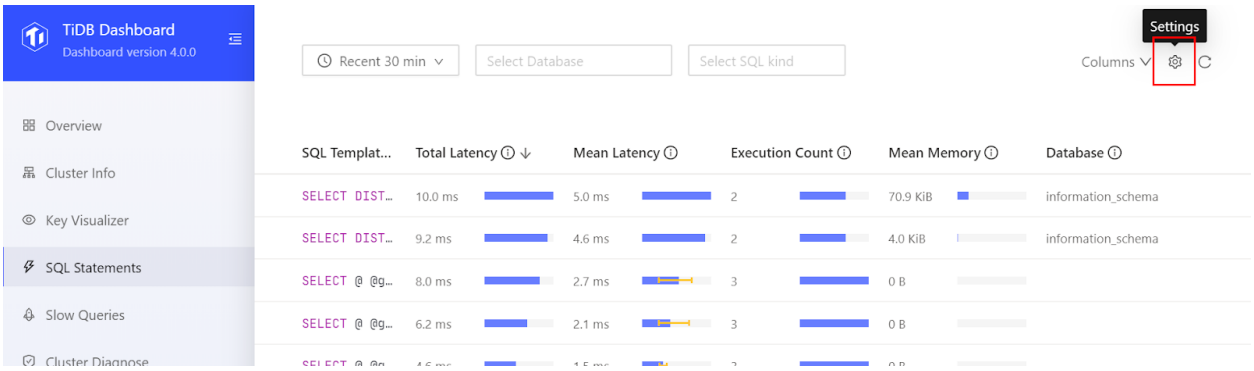


图 539: 设置入口

点击后设置界面如下图所示：

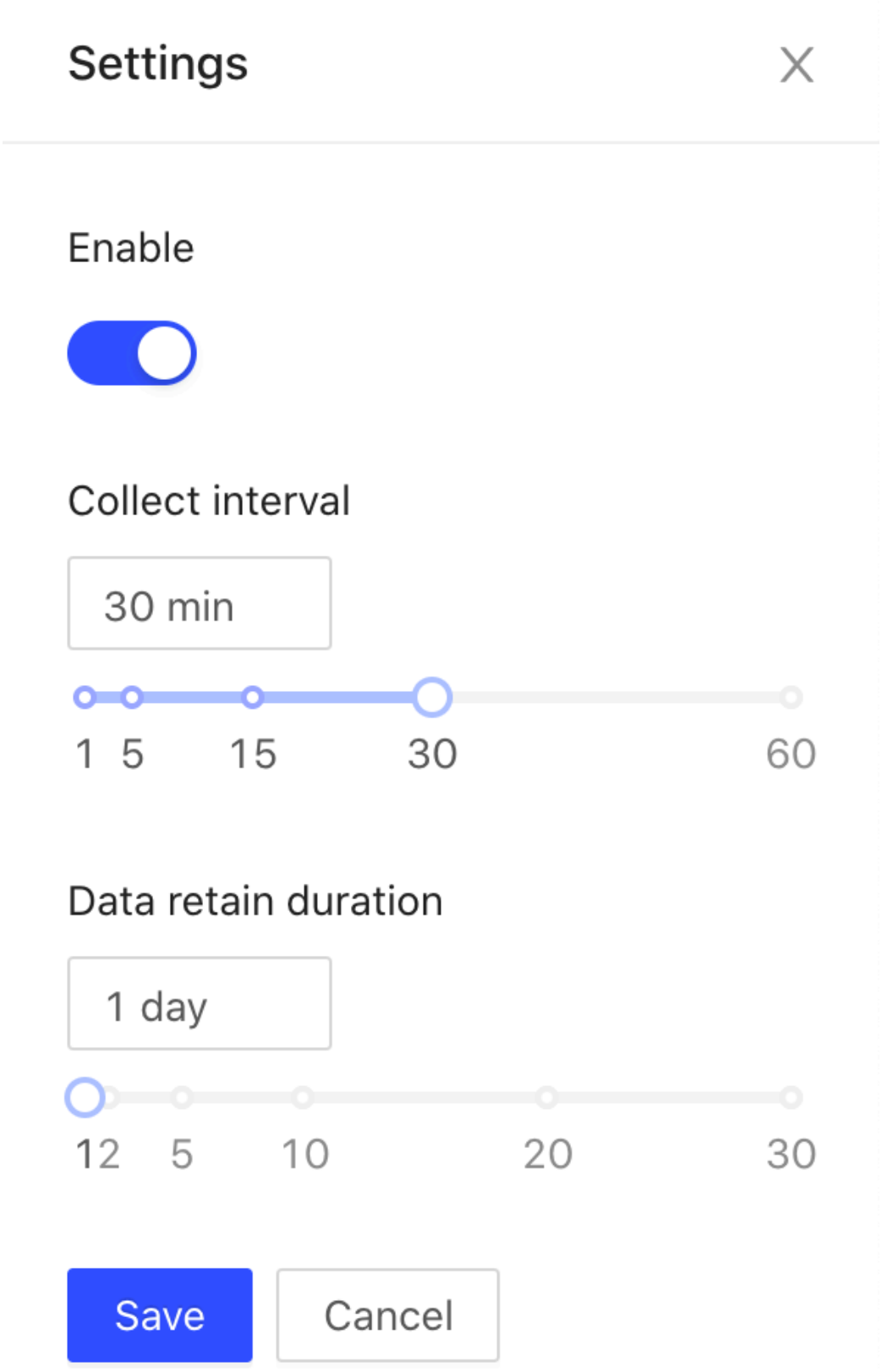

Settings
Enable
Collect interval
30 min
1 5 15 30 60
Data retain duration
1 day
12 5 10 20 30
Save
Cancel

图 540: 设置

在设置中可以选择关闭或开启 SQL 语句分析功能。在开启 SQL 语句分析功能时可以修改以下选项：

- 数据收集周期：默认 30 分钟，每次进行 SQL 语句分析的时间长度。SQL 语句分析功能每次对一段时间范围内的所有 SQL 语句进行汇总统计。

  如果这个时间范围过长，则统计的粒度粗，不利用定位问题；如果太短，则统计的粒度细，方便定位问题，但会导致在相同的数据保留时间内产生更多的记录，产生更多的内存占用。因此需要根据实际情况调整，在需要定位问题时适当地将值调低。

- 数据保留时间：默认 1 天，统计信息保留的时间，超过这个时间的数据会被从系统表中删除。

参见Statement Summary Tables 参数设置文档了解详细情况。

> 注意：
>
> 由于 Statement 系统表只存放在内存中，关闭此功能后，系统表中的数据会将清空。
>
> 数据收集周期和保留时间的值会影响内存占用，因此建议根据实际情况调整，保留时间不宜设置过大。

Others

`tidb_stmt_summary_max_stmt_count` 控制 Statement Summary 系统表保存的 SQL 种类数量。当 SQL 种类超过该值时，会移除最近没有出现的 SQL。这些 SQL 将会被 `DIGEST` 为 `NULL` 的行数据统计。`DIGEST` 为 `NULL` 的行数据在 TiDB Dashboard SQL 语句分析列表页面中将会显示为 `Others`，如下所示：

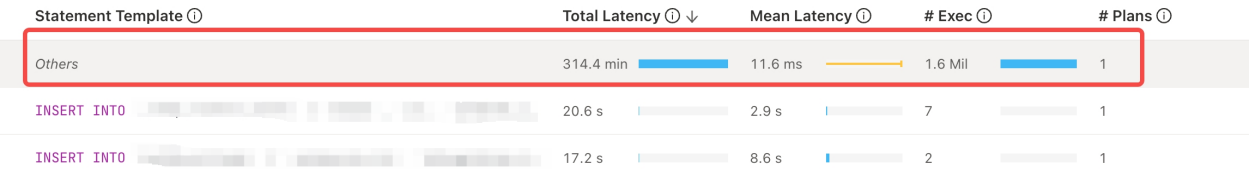


图 541: Others 行

下一步

阅读查看执行详情章节了解如何进一步查看 SQL 语句的详细执行情况。

14.12.1.9.2 TiDB Dashboard SQL 语句分析执行详情页面

在列表中点击任意一行可以进入该 SQL 语句的详情页查看更详细的信息，此信息包括三大部分：

- SQL 语句概况：包括 SQL 模板、SQL 模板 ID、当前查看的时间范围、执行计划个数、执行所在的数据库以及快速绑定执行计划功能（下图区域 1）

- 执行计划列表：如果一个 SQL 语句有多个执行计划，则显示执行计划列表。除了文本信息显示，TiDB 自 v6.2.0 开始引入图形化执行计划，通过图形化的执行计划，你可以更清晰地了解一个语句的具体算子和对应的内容。可以选择不同的执行计划，在列表和图形下方会显示该执行计划详情（下图区域 2）
- 执行计划详情：显示选中的执行计划的详细信息，具体见下一小节（下图区域 3）

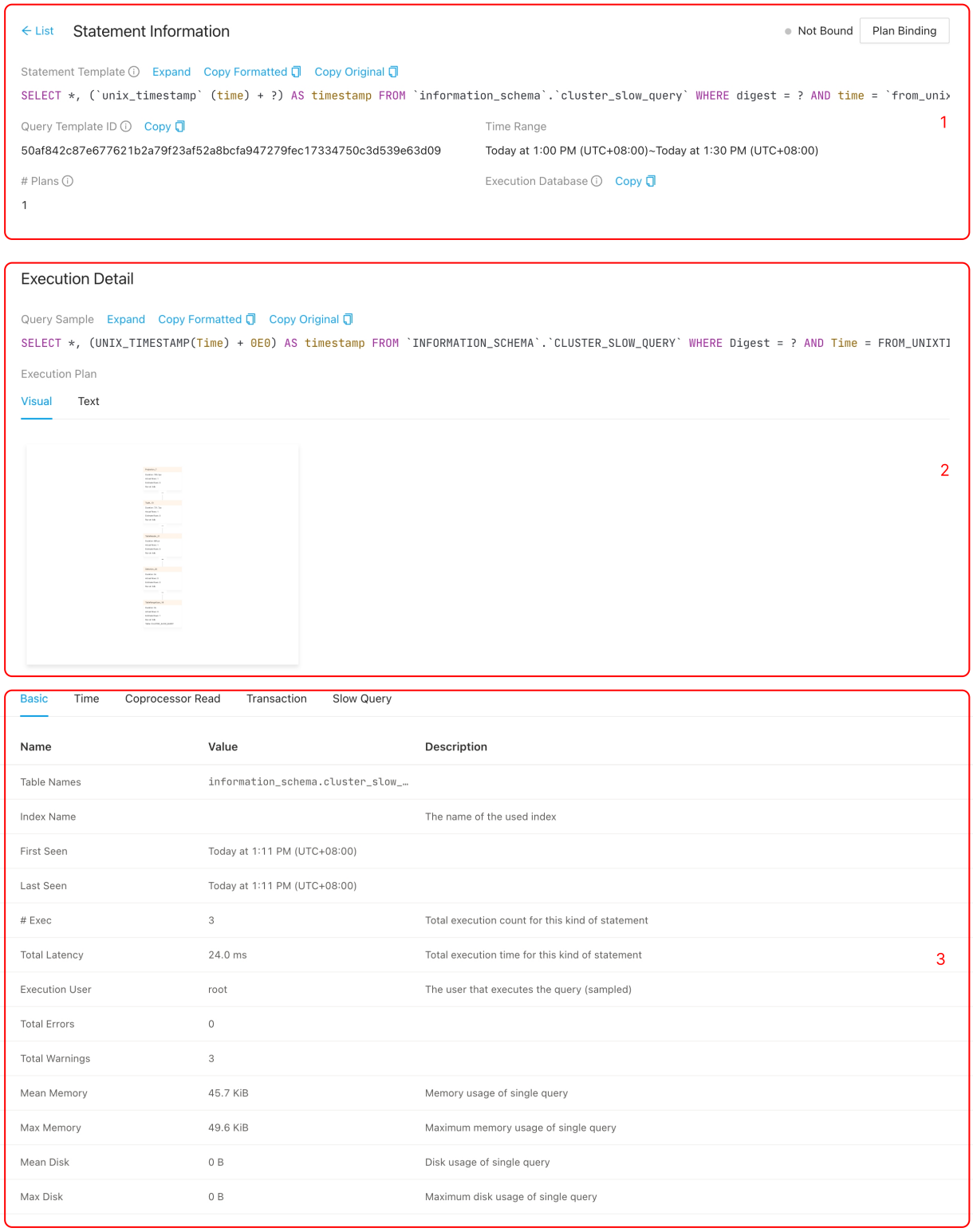

List
Statement Information
Not Bound
Plan Binding
Statement Template
Expand
Copy Formatted
Copy Original
SELECT *, (`unix_timestamp` (time) + ?) AS timestamp FROM `information_schema`.`cluster_slow_query` WHERE digest = ? AND time = `from_uni
Query Template ID
Copy
50af842c87e677621b2a79f23af52a8bcfa947279fec17334750c3d539e63d09
Time Range
Today at 1:00 PM (UTC+08:00)~Today at 1:30 PM (UTC+08:00)
1
Plans
1
Execution Database
Copy
Execution Detail
Query Sample
Expand
Copy Formatted
Copy Original
SELECT *, (UNIX_TIMESTAMP(Time) + 0E0) AS timestamp FROM `INFORMATION_SCHEMA`.`CLUSTER_SLOW_QUERY` WHERE Digest = ? AND Time = FROM_UNIXTI
Execution Plan
Visual
Text
2
Basic
Time
Coprocessor Read
Transaction
Slow Query
Name
Value
Description
Table Names
information_schema.cluster_slow_…
Index Name
The name of the used index
First Seen
Today at 1:11 PM (UTC+08:00)
Last Seen
Today at 1:11 PM (UTC+08:00)
Exec
3
Total execution count for this kind of statement
Total Latency
24.0 ms
Total execution time for this kind of statement
3
Execution User
root
The user that executes the query (sampled)
Total Errors
0
Total Warnings
3
Mean Memory
45.7 KiB
Memory usage of single query
Max Memory
49.6 KiB
Maximum memory usage of single query
Mean Disk
0 B
Disk usage of single query
Max Disk
0 B
Maximum disk usage of single query

图 542: 详情

快速绑定执行计划

自 v6.6.0 起，TiDB 引入了快速绑定执行计划的功能。你可以在 TiDB Dashboard 中，快速完成 SQL 语句与特定计划的绑定。

使用方式

绑定一个执行计划

1. 点击执行计划绑定 (Plan Binding)，弹出弹窗。

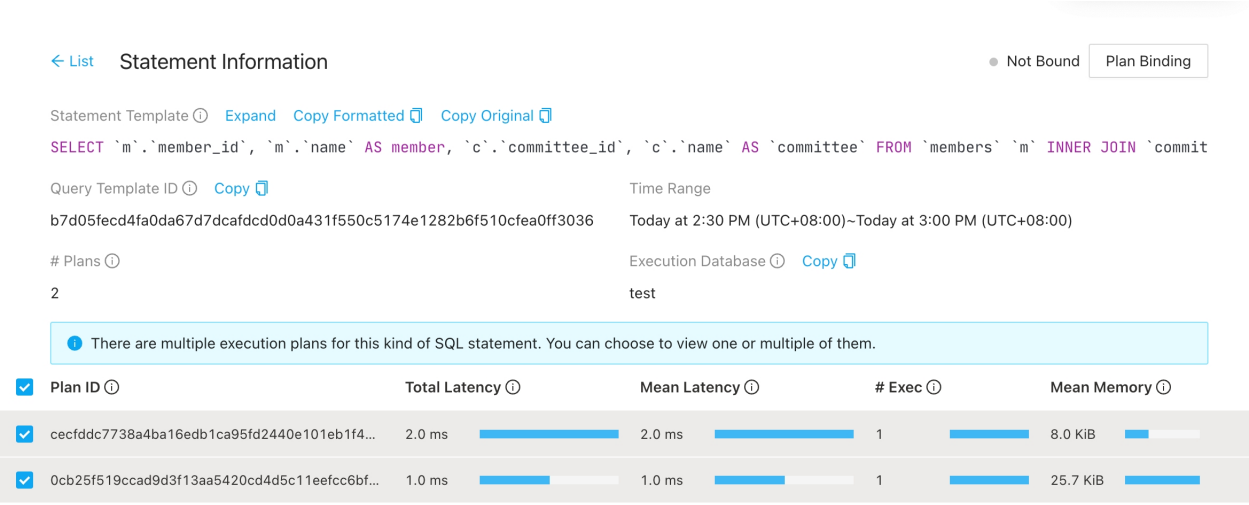


图 543: 快速绑定执行计划 -未绑定 -入口

2. 选择一个需要绑定的执行计划，点击绑定 (Bind)。

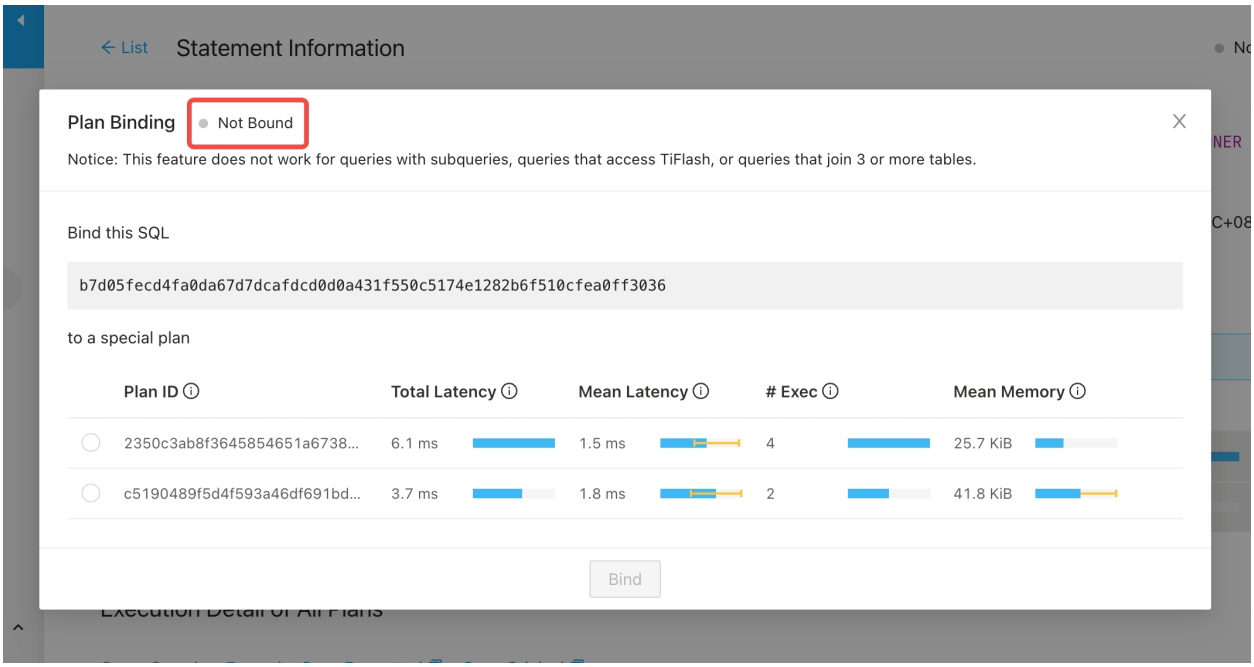


图 544: 快速绑定执行计划 -弹窗

3. 绑定成功后，可以看到已绑定 (Bound) 提示。

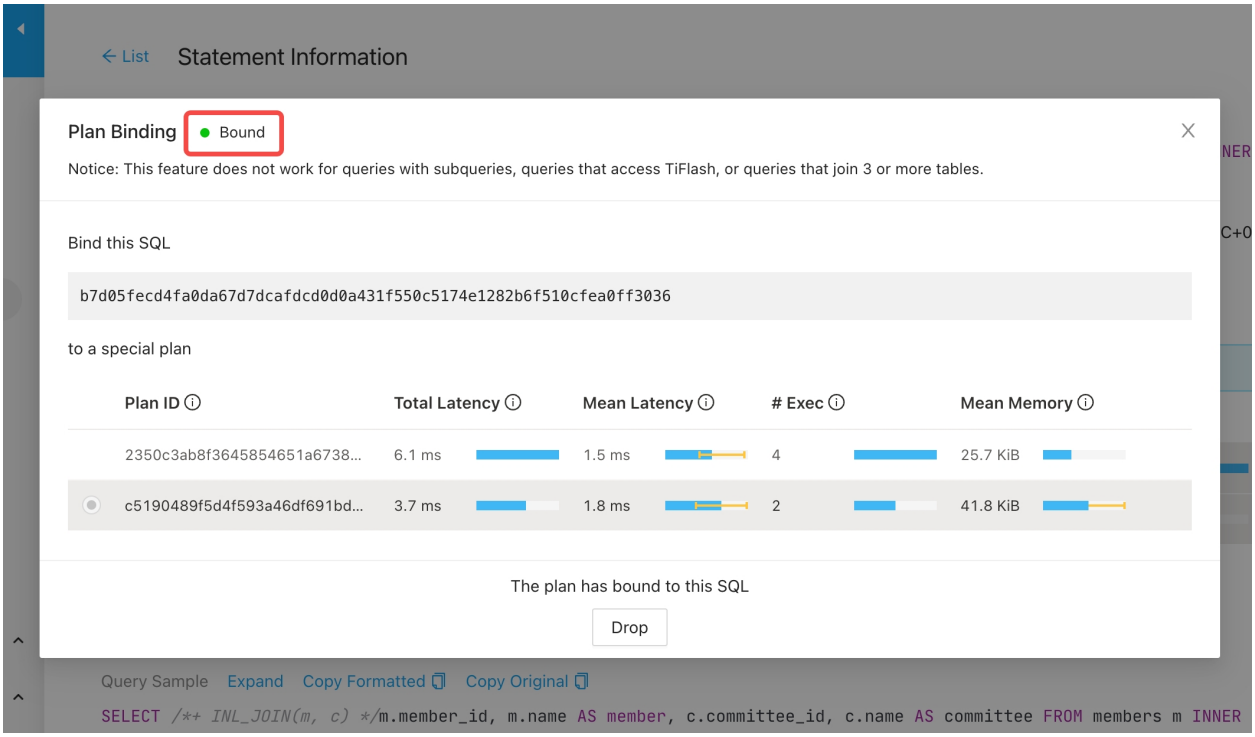


图 545: 快速绑定执行计划 -弹窗 -绑定成功

取消已有的执行计划绑定

1. 在已经绑定了执行计划的 SQL 语句页面上，点击执行计划绑定 (Plan Binding)，弹出弹窗。

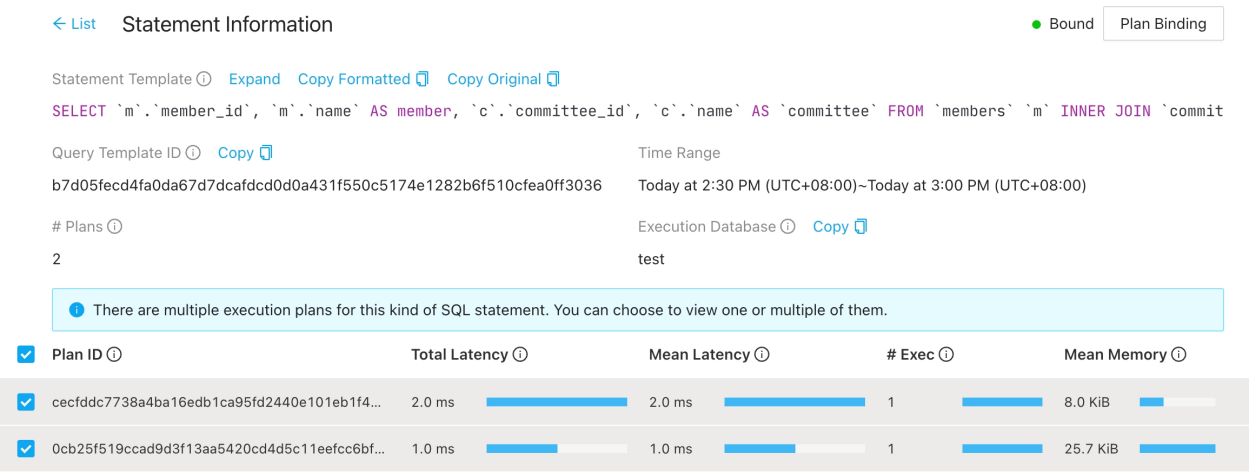


图 546: 快速绑定执行计划 -已绑定 -入口

2. 点击取消绑定 (Drop)。

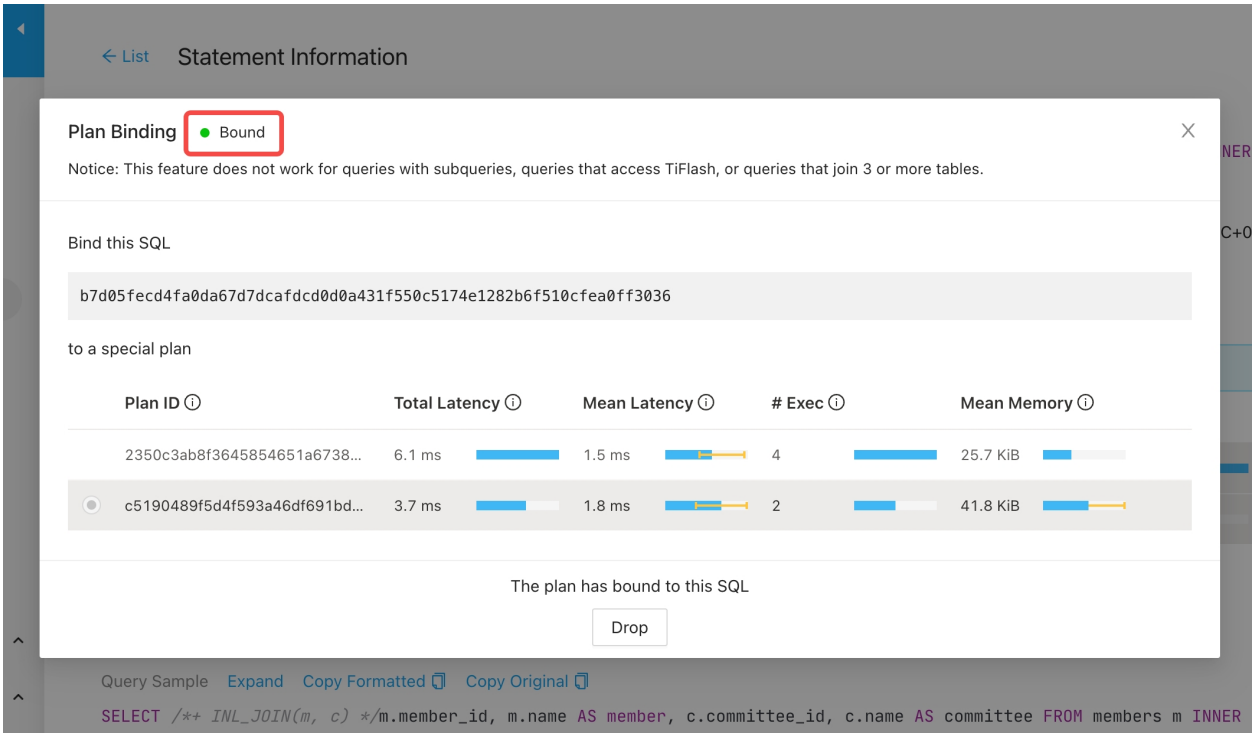


图 547: 快速绑定执行计划 -弹窗 -绑定成功

3. 取消成功后，可以看到未绑定 (Not Bound) 提示。

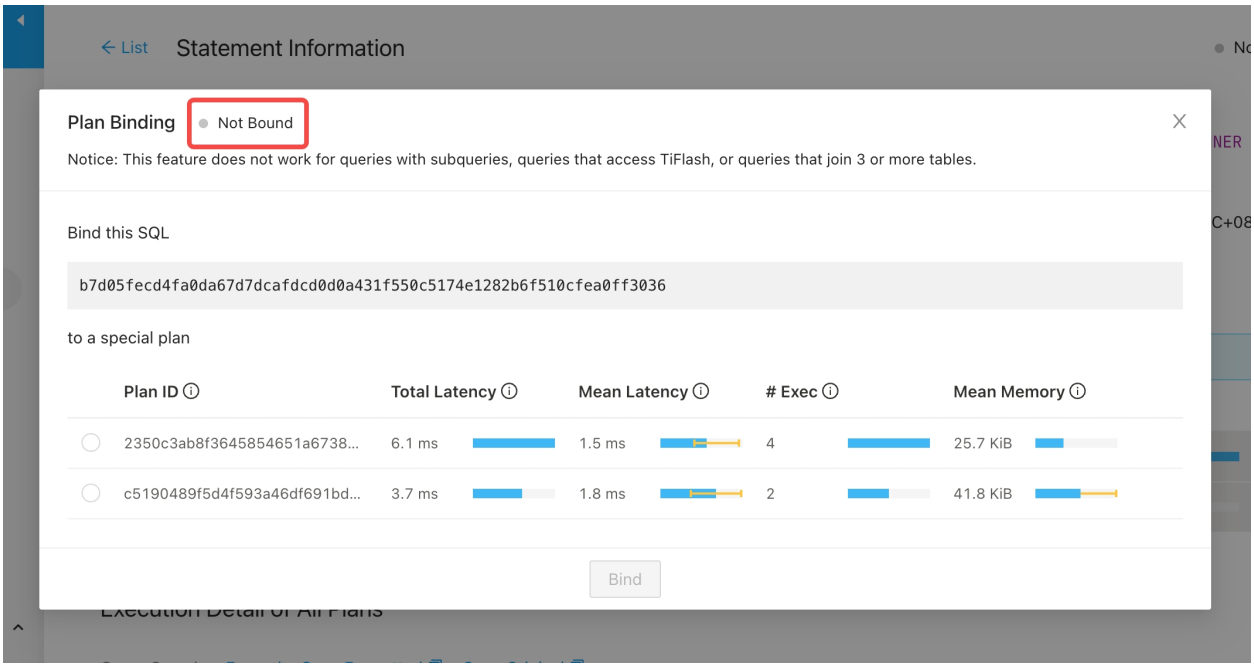


图 548: 快速绑定执行计划 -弹窗

使用限制

目前还不支持通过 TiDB Dashboard 绑定下列语句的执行计划：

- 非 SELECT、DELETE、UPDATE、INSERT、REPLACE 类型语句
- 带有子查询的查询
- 访问 TiFlash 的查询
- 对三张表或更多表进行 Join 的查询

目前该功能需用户拥有 SUPER 权限才可使用。如果在使用过程中提示权限不足，请参考TiDB Dashboard 用户管理补充所需权限。

执行计划详情

执行计划详情包括以下内容：

- SQL 样本：该计划对应的实际执行的某一条 SQL 语句文本。时间范围内任何出现过的 SQL 都可能作为 SQL 样本。
- 执行计划：执行计划的完整内容，有图形和文本两种展示形式。参阅理解 TiDB 执行计划文档了解如何解读执行计划。如果选择了多个执行计划，则显示的是其中任意一个。
- 其他关于该 SQL 的基本信息、执行时间、Coprocessor 读取、事务、慢查询等信息，可点击相应标签页标题切换。

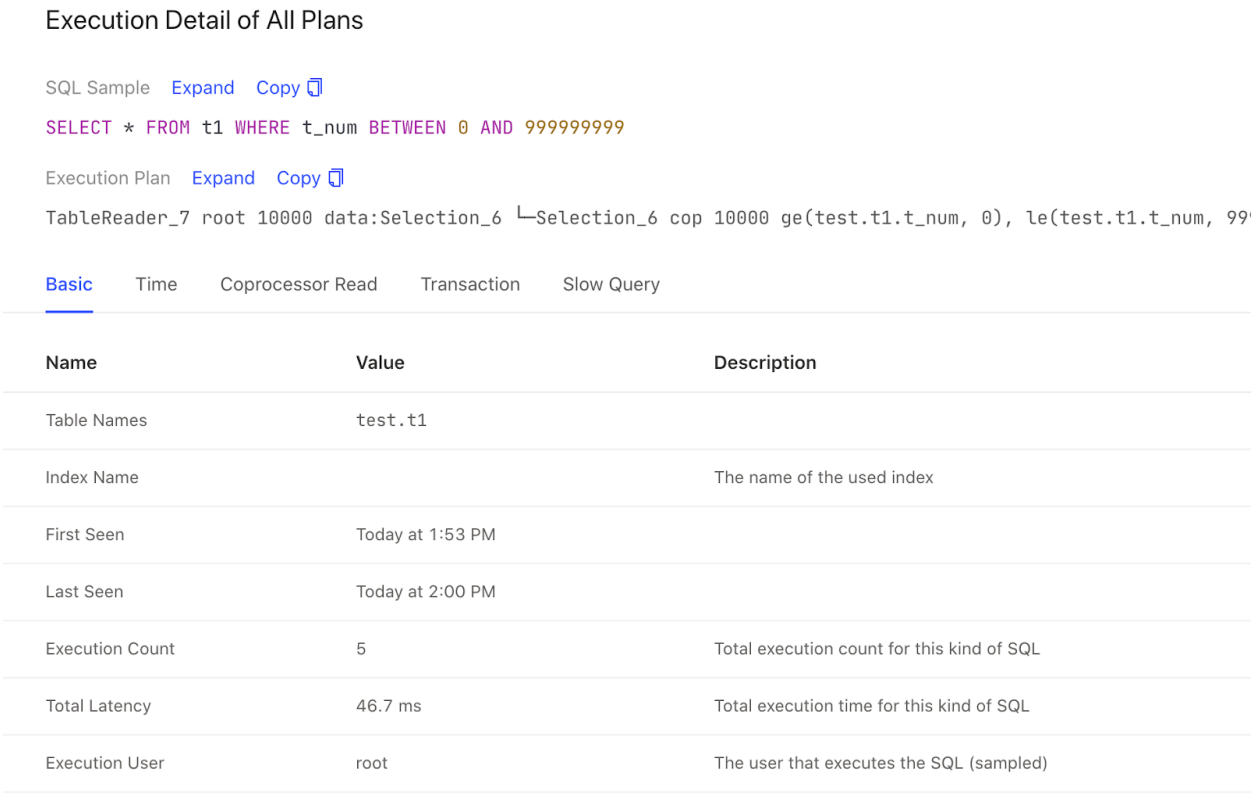


图 549: 执行计划详情

基本信息

包含关于表名、索引名、执行次数、累计耗时等信息。描述 (Description) 列对各个字段进行了具体描述。

Basic | Time | Coprocessor Read | Transaction | Slow Query

Name	Value	Description
Table Names	test.t1	
Index Name		The name of the used index
First Seen	Today at 1:53 PM	
Last Seen	Today at 2:00 PM	
Execution Count	5	Total execution count for this kind of SQL
Total Latency	46.7 ms	Total execution time for this kind of SQL
Execution User	root	The user that executes the SQL (sampled)
Total Errors	0	
Total Warnings	0	
Mean Memory	111.9 KiB	Memory usage of single SQL
Max Memory	236.3 KiB	Maximum memory usage of single SQL

图 550: 基本信息

执行时间

显示执行计划执行的各阶段所耗费时间。

> **注意：**
>
> 由于单个 SQL 语句内部可能有并行执行的操作，因此各阶段累加时间可能超出该 SQL 语句的实际执行时间。

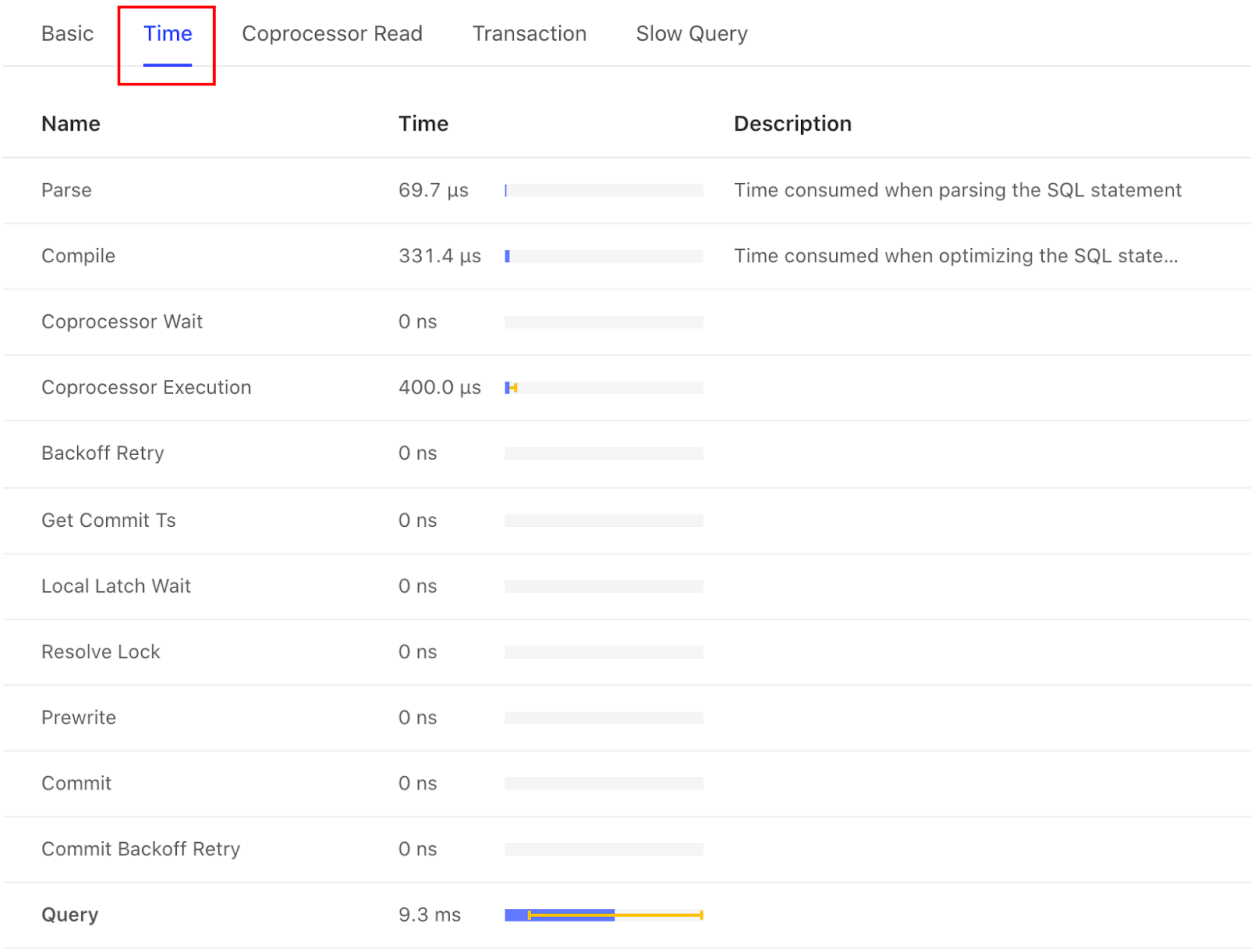


图 551: 执行时间

Coprocessor 读取

显示 Coprocessor 读取的相关信息。

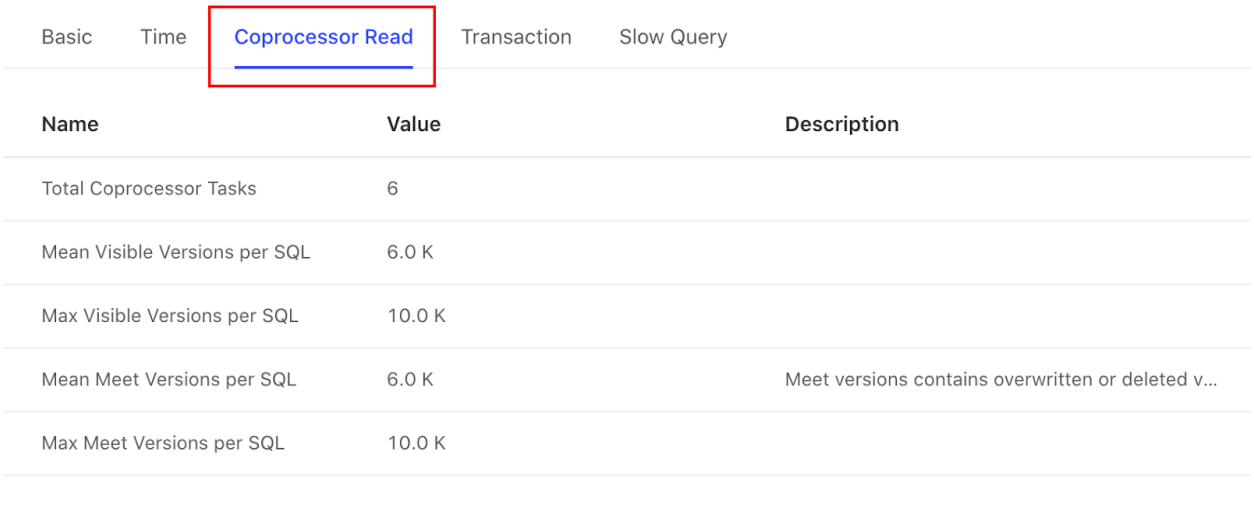


图 552: Coprocessor 读取

事务

显示执行计划与事务相关的信息，比如平均写入 key 个数，最大写入 key 个数等。

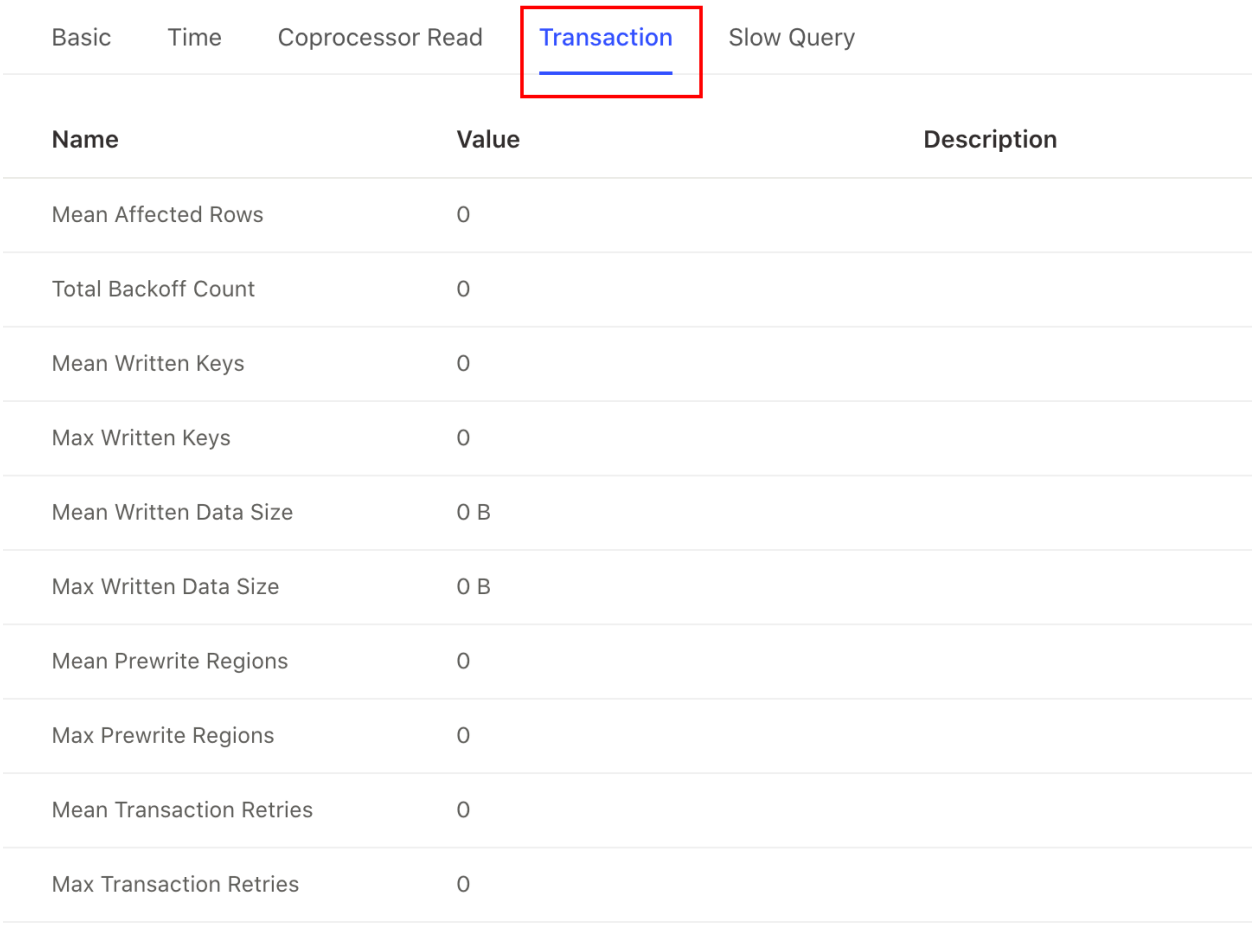


图 553: 事务

慢查询

如果该执行计划执行过慢，则在慢查询标签页下可以看到其关联的慢查询记录。

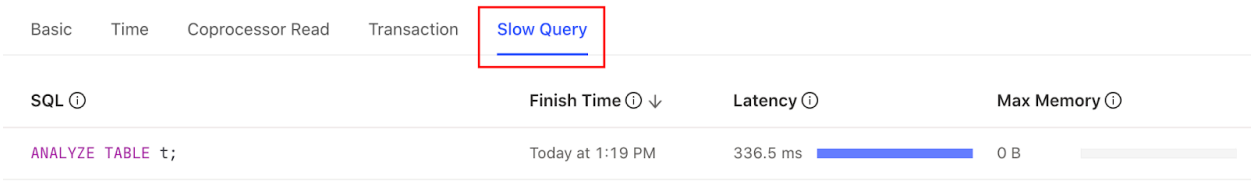


图 554: 慢查询

该区域显示的内容结构与慢查询页面一致，详见慢查询页面。

14.12.1.10 TiDB Dashboard 慢查询页面

该页面上能检索和查看集群中所有慢查询。

默认情况下，执行时间超过 300ms 的 SQL 查询就会被视为慢查询，被记录到慢查询日志中，并可通过本功能对记录到的慢查询进行查询。可调整 tidb_slow_log_threshold SESSION 变量或 TiDB slow-threshold 参数调整慢查询阈值。

> **注意：**
>
> 若关闭了慢查询日志，则本功能不可用。慢查询日志默认开启，可通过修改 TiDB 系统变量 tidb_enable_slow_log 开启或禁用。

14.12.1.10.1 访问列表页面

可以通过以下两种方法访问慢查询页面：

- 登录后，左侧导航条点击慢查询 (Slow Queries)：

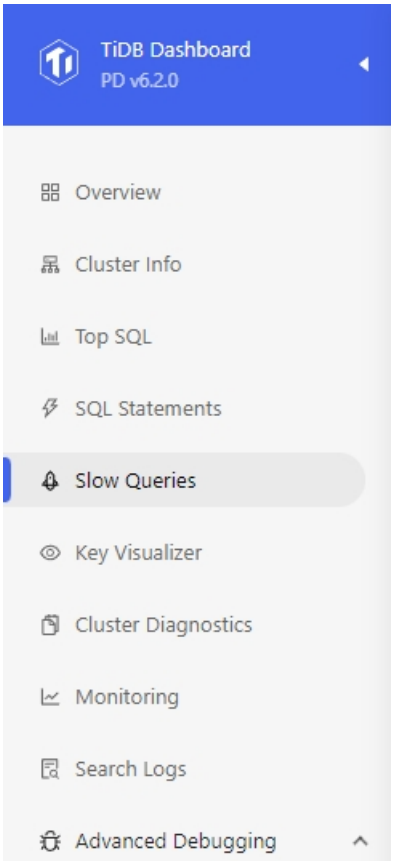


图 555: access 访问页面

- 在浏览器中访问 http://127.0.0.1:2379/dashboard/#/slow_query（将 127.0.0.1:2379 替换为任意实际 PD 地址和端口）。

慢查询页面所展示的所有数据都来自于 TiDB 慢查询系统表及慢查询日志，参见慢查询日志文档了解详细情况。

修改列表过滤条件

可按时间范围、慢查询语句关联的数据库、SQL 关键字、SQL 类型、显示的慢查询语句数量等条件过滤，筛选慢查询句。如下所示，默认显示 30 分钟内最近 100 条慢查询。

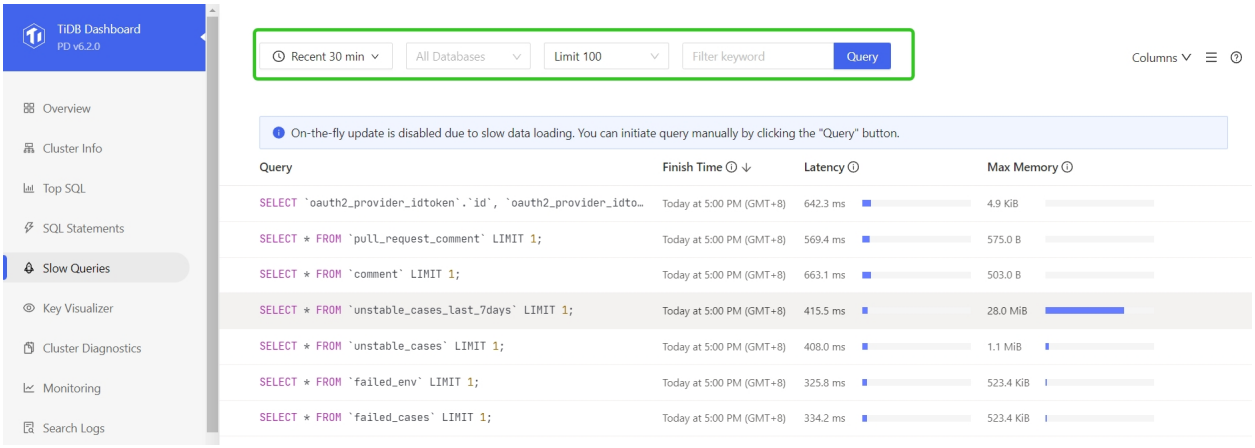


图 556: 修改列表过滤条件

显示更多列信息

页面顶部选择列 (Columns) 选项中可选择显示更多列，可将鼠标移动到列名右侧的 (i) 图标处查看列的说明：

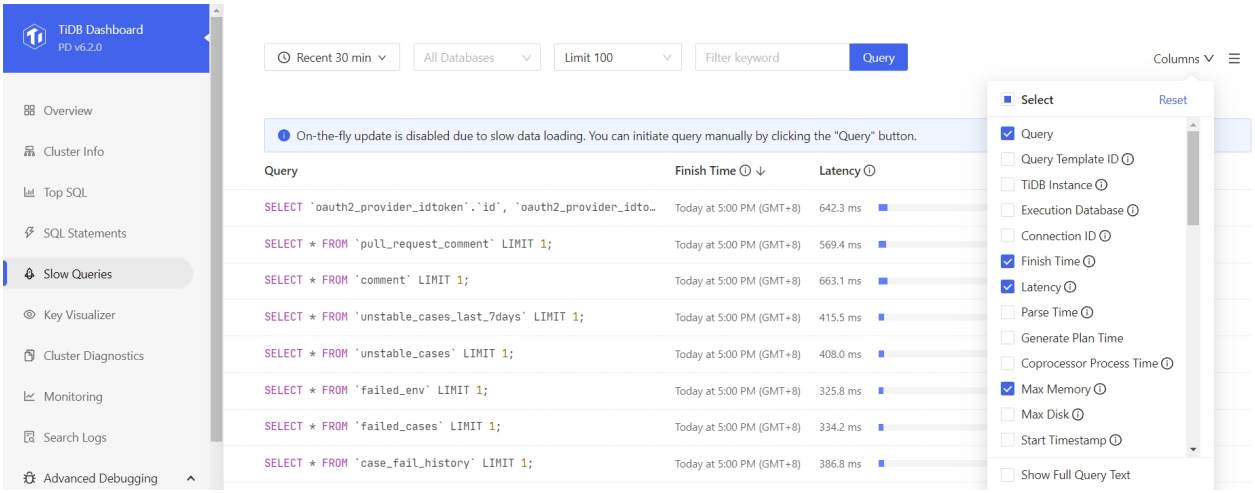


图 557: 显示更多列信息

导出慢查询到本地

点击页面右上角 ▯ (更多) 可以显示导出 (Export) 选项。点击导出 (Export) 后，TiDB Dashboard 会将当前列表中的慢查询以 CSV 文件的格式进行导出。

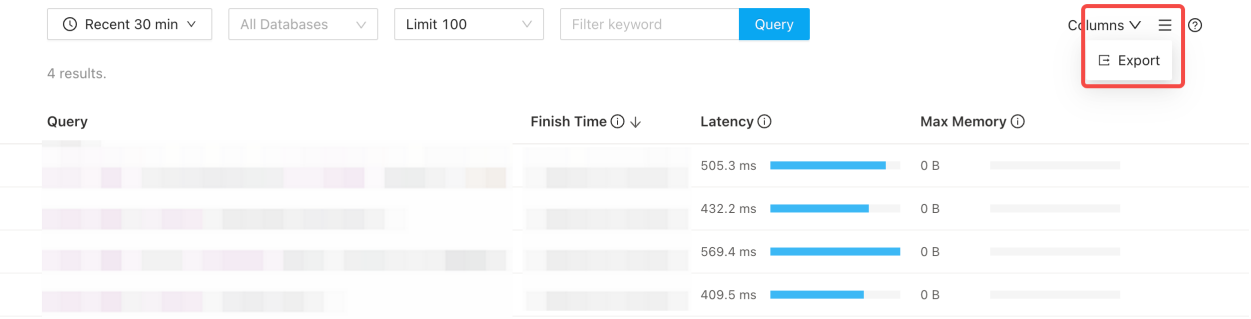


图 558: 导出慢查询到本地

修改列表排序依据

列表默认以结束运行时间 (Finish Time) 逆序排序，点击不同的列标题可以修改排序依据或切换排序顺序：

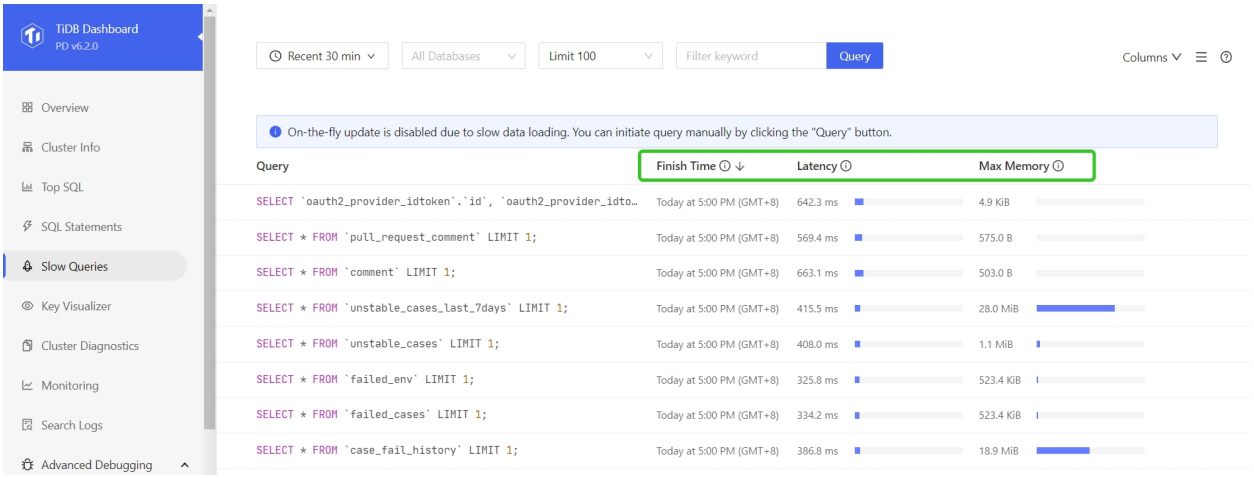


图 559: 修改列表排序依据

14.12.1.10.2 查看执行详情

在列表中点击任意一行可以显示该慢查询的详细执行信息，包含：

- SQL：慢查询 SQL 文本（下图中区域 1）
- 执行计划：慢查询的执行计划（下图中区域 2）
- 其他分类好的 SQL 执行信息（下图中区域 3）

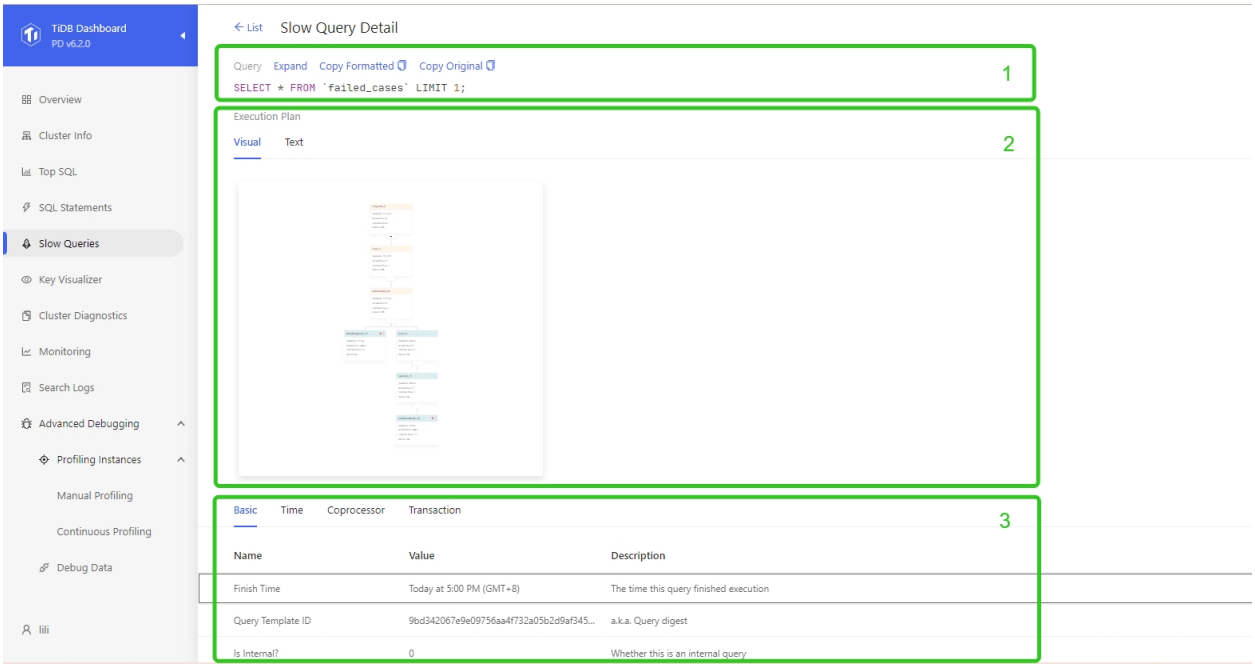


图 560: 查看执行详情

SQL

点击展开 (Expand) 可以展开相应项的完整内容，点击复制 (Copy) 可以复制内容到剪贴板。

执行计划

TiDB Dashboard 提供两种方式查看执行计划：图形和文本。通过图形化的执行计划，你可以更清晰地了解一个语句的具体算子和对应的内容。参阅理解 TiDB 执行计划。

图形化执行计划介绍

下图为一个执行计划的图形化展示。

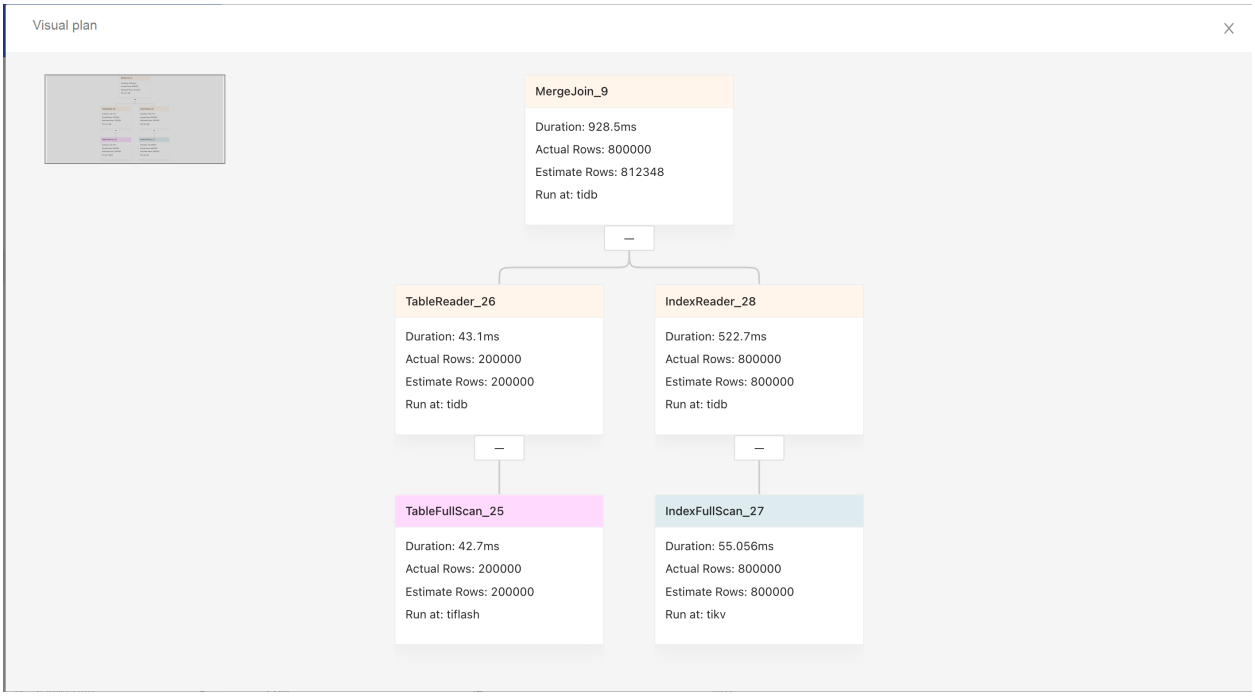


图 561: 可视化执行计划弹窗

- 执行计划的展示顺序是从左到右，从上到下。
- 上面的节点是父算子，下面的节点是子算子。
- 节点顶栏的颜色代表算子执行的组件：黄色代表 TiDB，蓝色代表 TiKV，粉色代表 TiFlash。
- 节点的顶栏为算子的名称，正文为算子的基本信息。

点击节点区域，右侧将弹出算子的详细信息。

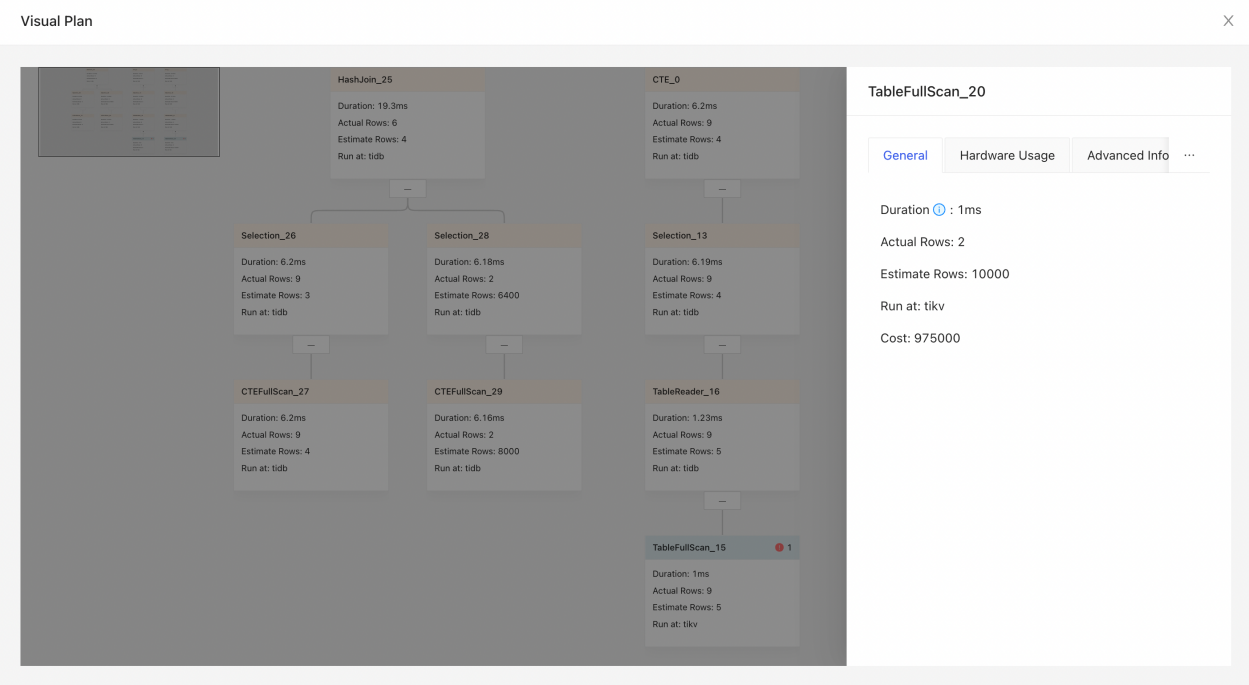


图 562: 可视化执行计划弹窗 -侧栏

SQL 执行相关信息

点击标签页标题可切换显示不同分类的 SQL 执行信息：

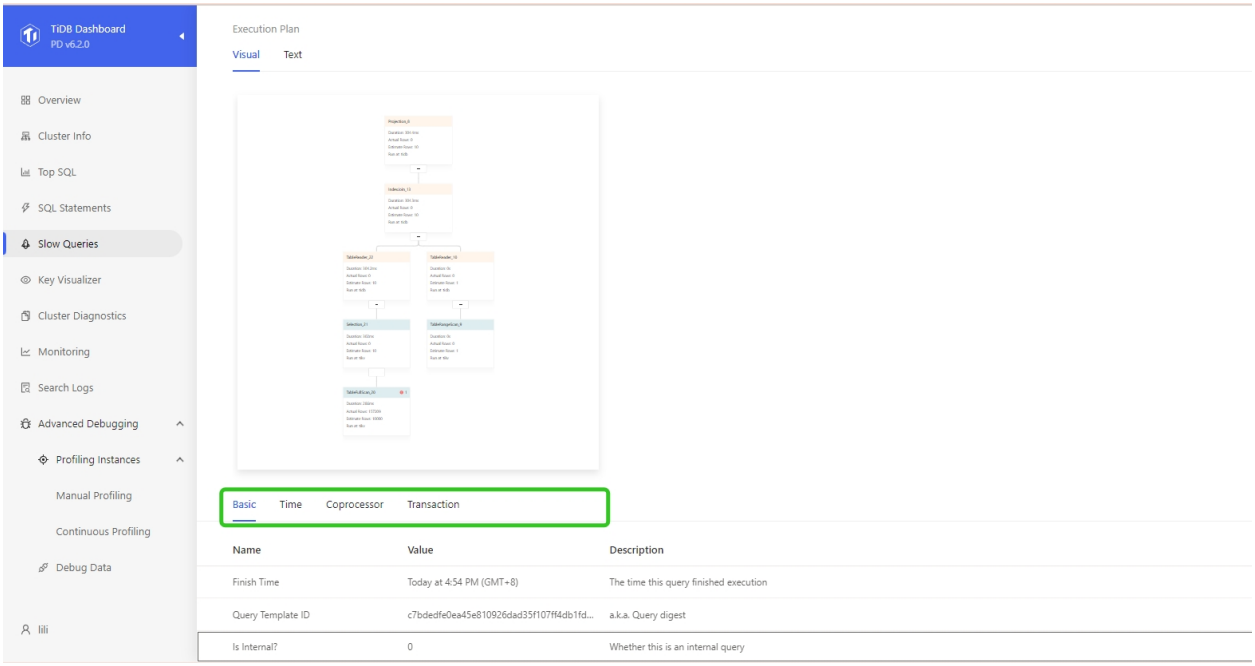


图 563: 显示不同分类执行信息

14.12.1.11 集群诊断页面

14.12.1.11.1 TiDB Dashboard 集群诊断页面

集群诊断是在指定的时间范围内，对集群可能存在的问题进行诊断，并将诊断结果和一些集群相关的负载监控信息汇总成一个诊断报告。诊断报告是网页形式，通过浏览器保存后可离线浏览和传阅。

> **注意：**
>
> 集群诊断功能依赖于集群中部署有 Prometheus 监控组件，参见TiUP 部署文档了解如何部署监控组件。若集群中没有部署监控组件，生成的诊断报告中将提示生成失败。

访问

可以通过以下两种方法访问集群诊断页面：

- 登录后，左侧导航条点击集群诊断 (Cluster Diagnostics)：

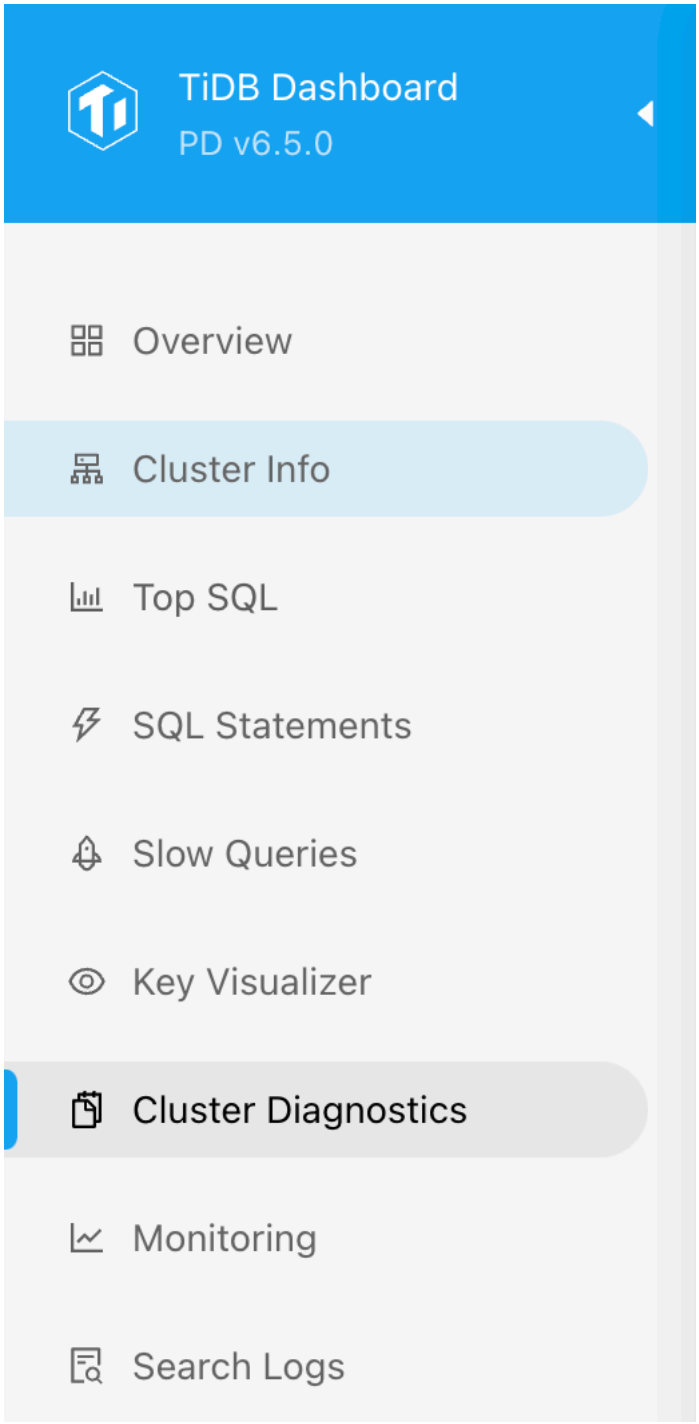


图 564: 访问

- 在浏览器中访问 http://127.0.0.1:2379/dashboard/#/diagnose（将 `127.0.0.1:2379` 替换为任意实际 PD 地址和端口）。

生成诊断报告

如果想对一个时间范围内的集群进行诊断，查看集群的负载等情况，可以使用以下步骤来生成一段时间范围

的诊断报告：

1. 设置区间的开始时间，例如 2022-05-21 14:40:00。
2. 设置区间长度。例如 10 min。
3. 点击开始 (Start)。

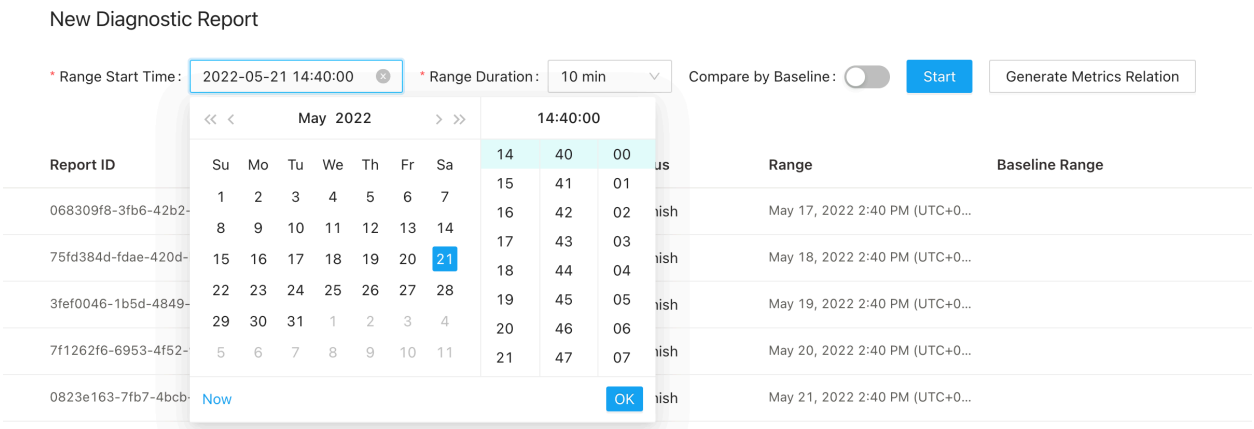


图 565: 生成单个时间段的诊断报告

> **注意：**
>
> 建议生成报告的时间范围在 1 min ~ 60 min 内，目前不建议生成超过 1 小时范围的报告。

以上操作会生成 2022-05-21 14:40:00 至 2022-05-21 14:50:00 时间范围的诊断报告。点击开始 (start) 后，会看到以下界面，生成进度 (Progress) 是生成报告的进度条，生成报告完成后，点击查看报告 (View Full Report) 即可。

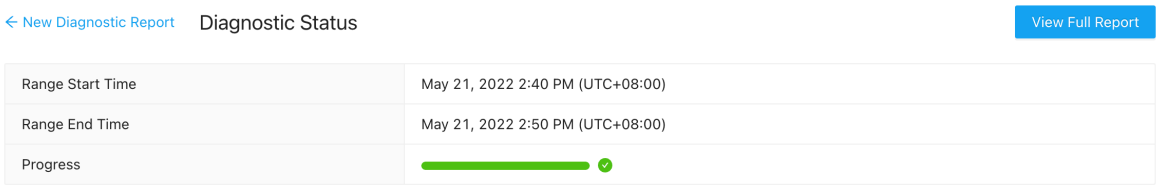


图 566: 生成报告的进度

生成对比诊断报告

如果系统在某个时间点发生异常，如 QPS 抖动或者延迟变高，可以生成一份异常时间范围和正常时间范围的对比报告，例如：

- 系统异常时间段：2022-05-21 14:40:00 ~ 2022-05-21 14:45:00，系统异常时间。
- 系统正常时间段：2022-05-21 14:30:00 ~ 2022-05-21 14:35:00，系统正常时间。

生成以上两个时间范围的对比报告的步骤如下：

1. 设置区间的开始时间，即异常时间段的开始时间，如 2022-05-21 14:40:00。
2. 设置区间长度 (Range Duration)。一般指系统异常的持续时间，例如 5 min。
3. 开启与基线区间对比开关 (Compare by Baseline)。
4. 设置基线开始时间 (Baseline Range Start Time)，即想要对比的系统正常时段的开始时间，如 2022-05-21 14:30:00。
5. 点击开始 (Start)。

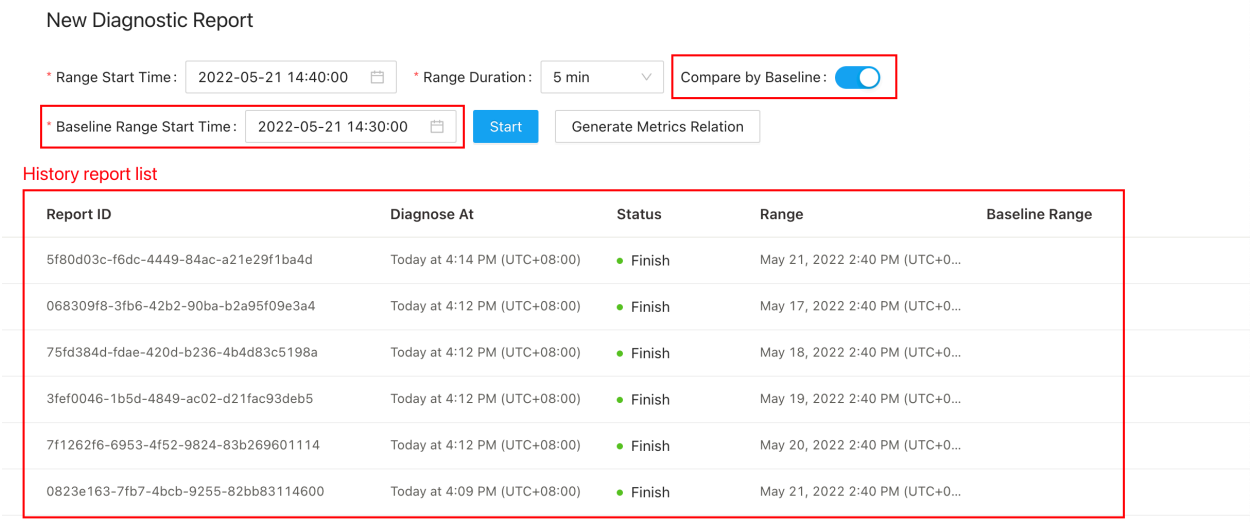


图 567: 生成对比报告

然后同样等报告生成完成后点击查看报告 (View Full Report) 即可。

另外，已生成的诊断报告会显式在诊断报告主页的列表里面，可以点击查看之前生成的报告，不用重复生成。

14.12.1.11.2 TiDB Dashboard 诊断报告

本文档主要介绍诊断报告的内容以及查看技巧，访问集群诊断和生成报告请参考诊断报告访问文档。

查看报告

诊断报告由以下几部分组成：

- 基本信息：包括生成报告的时间范围，集群的硬件信息，集群的拓扑版本信息。
- 诊断信息：显示自动诊断的结果。
- 负载信息：包括服务器，TIDB/PD/TiKV 相关的 CPU、内存等负载信息。
- 概览信息：包括 TiDB/PD/TiKV 的各个模块的耗时信息和错误信息。

- TiDB/PD/TiKV 监控信息：包括各个组件的监控信息。
- 配置信息：包括各个组件的配置信息。

报告中报表示例如下：

Total Time Consume

The table contain the event time consume in TiDB/TiKV/PD. METRIC_NAME is the event name; LABEL is the event label, such as instance, event type ...; TIME_RATIO is the TOTAL_TIME of this event devide by the TOTAL_TIME of upper event which TIME_RATIO is 1; TOTAL_TIME is the total time cost of this event; TOTAL_COUNT is the total count of this event; P999 is the max time of 0.999 quantile; P99 is the max time of 0.99 quantile; P90 is the max time of 0.90 quantile; P80 is the max time of 0.80 quantile;

METRIC_NAME	LABEL	TIME_RATIO	TOTAL_TIME	TOTAL_COUNT	P999	P99	P90	P80
tidb_query expand		1	23223.86	459625	63.54	45.55	0.1	0.06
tidb_get_token fold		0.000001	0.03	458888	0.00004	0.000001	0.000001	0.000001
|-- tidb_get_token	10.0.1.13:10080	0.000001	0.03	458888	0.00004	0.000001	0.000001	0.000001

图 568: 示例报表

上图中，最上面蓝框内的 Total Time Consume 是报表名。下方红框内的内容是对该报表意义的解释，以及报表中各个字段的含义。

报表中小按钮图标的解释如下：

- i 图标：鼠标移动到 i 图标处会显示该行的说明注释。
- expand：点击 expand 会看到这项监控更加详细的信息。如是上图中 `tidb_get_token` 的详细信息包括各个 TiDB 实例的延迟监控信息。
- fold：和 expand 相反，用于把监控的详细信息折叠起来。

所有监控基本上和 TiDB Grafna 监控面板上的监控内容相对应，发现某个模块异常后，可以在 TiDB Grafna 监控面板上查看更多详细的监控信息。

另外，报表中统计的 `TOTAL_TIME` 和 `TOTAL_COUNT` 由于是从 Prometheus 读取的监控数据，其统计会有一些计算上的精度误差。

以下介绍诊断报告的各部分内容。

基本信息

Report Time Range

Report Time Range 表显示生成报告的时间范围，包括开始时间和结束时间。

Report Time Range

START_TIME	END_TIME
2020-05-21 06:40:00	2020-05-21 06:50:00

图 569: report time range 报表

Cluster Hardware

Cluster Hardware 表显示集群中各服务器的硬件信息，包括 CPU、Memory、磁盘等信息。

Cluster Hardware

HOST	INSTANCE	CPU_CORES	MEMORY (GB)	DISK (GB)	UPTIME (DAY)
10.0.1.14	tiflash*1	20/40	125.64	nvme01: 498.638	2.9132028513567314
10.0.1.11	pd*1	20/40	125.64	sda1: 965.834	2.921851028772416
10.0.1.12	tikv*1	20/40	125.64	nvme0n1: 122.78 sda1: 965.834	2.9197623066052247
10.0.1.13	tidb*1	20/40	125.64	sda1: 498.638	2.917983893669314

图 570: Cluster Hardware 报表

上表中各个字段含义如下：

- HOST：服务器的 IP 地址。
- INSTANCE：该服务器部署的实例数量，如 pd * 1 代表这台服务器部署了 1 个 PD 实例。如 tidb * 2 pd * 1 表示这台服务器部署了 2 个 TiDB 实例和 1 个 PD 实例。
- CPU_CORES：表示服务器 CPU 的核心数，物理核心/逻辑核心。
- MEMORY：表示服务器的内存大小，单位是 GB。
- DISK：表示服务器磁盘大小，单位是 GB。
- UPTIME：服务器的启动时间，单位是 DAY。

Cluster Info

Cluster Info 为集群拓扑信息。表中信息来自 TiDB 的information_schema.cluster_info 系统表。

Cluster Info

TYPE	INSTANCE	STATUS_ADDRESS	VERSION	GIT_HASH	START_TIME	UPTIME
pd	10.0.1.11:2379	10.0.1.11:2379	4.1.0-alpha	a80b99ef0d70807d106bdc1b7c6e97a118679c7e	2020-05-18T13:47:01Z	65h40m54.520385474s
tidb	10.0.1.13:4000	10.0.1.13:10080	4.0.0-beta.2	ac30f5322e253125002663ad7c789807acfc9305	2020-05-18T13:47:10Z	65h40m45.520383242s
tiflash	10.0.1.14:3930	10.0.1.14:20292	v4.1.0-alpha-37-gacf3f673d	acf3f673dfbc3e6ffff0dda575760670993fefa6	2020-05-18T13:47:18Z	65h40m37.520390633s
tikv	10.0.1.12:20160	10.0.1.12:20180	4.1.0-alpha	6b09cadbf55311ac07fc51e1b43e3b57b54e71f2	2020-05-18T13:47:04Z	65h40m51.520389948s

图 571: Cluster Info 报表

上表中各个字段含义如下：

- TYPE：节点类型。
- INSTANCE：实例地址，为 IP:PORT 格式的字符串。
- STATUS_ADDRESS：HTTP API 的服务地址。
- VERSION：对应节点的语义版本号。
- GIT_HASH：编译节点版本时的 Git Commit Hash，用于识别两个节点是否是绝对一致的版本。
- START_TIME：对应节点的启动时间。
- UPTIME：对应节点已经运行的时间。

诊断信息

TiDB 内置自动诊断的结果，具体各字段含义以及介绍可以参考information_schema.inspection_result 系统表的内容。

负载信息

Node Load Info

Node Load Info 表显示服务器节点的负载信息，包括时间范围内，服务器以下指标的平均值 (AVG)、最大值 (MAX)、最小值 (MIN)：

- CPU 使用率，最大值是 100%
- 内存使用率
- 磁盘 I/O 使用率
- 磁盘写延迟
- 磁盘读延迟
- 磁盘每秒的读取字节数
- 磁盘每秒的写入字节数
- 节点网络每分钟收到的字节数
- 节点网络每分钟发送的字节数
- 节点正在使用的 TCP 连接数
- 节点所有的 TCP 连接数

Node Load Info

METRIC_NAME	instance	AVG	MAX	MIN
node_cpu_usage expand		40.42%	99.6%	6.98%
node_mem_usage expand		66.9%	96.28%	33.65%
node_disk_io_utilization expand		16%	97%	0%
Disk write latency expand		4000 us	50 ms	0 us
Disk read latency expand		2000 us	6000 us	0 us
tikv_disk_read_bytes expand		605.980 KB	15.842 MB	0.000 KB
tikv_disk_write_bytes expand		1.129 MB	12.235 MB	0.000 KB
node_network_in_traffic expand		225.706 KB	1.427 MB	0.000 KB
node_network_out_traffic expand		165.807 KB	1.659 MB	0.000 KB
node_tcp_in_use expand		19	47	5
node_tcp_connections expand		30	61	6

图 572: Node Load Info 报表

Instance CPU Usage

Instance CPU Usage 表显示各个 TiDB/PD/TiKV 进程的 CPU 使用率的平均值 (AVG)，最大值 (MAX)，最小值 (MIN)，这里进程 CPU 使用率最大值是 100% * CPU 逻辑核心数。

INSTANCE	JOB	AVG	MAX	MIN
172.16.5.40:10089	tidb	1891%	1906%	1878%
172.16.5.40:22151	tikv	517%	571%	459%
172.16.5.40:20151	tikv	498%	562%	444%
172.16.5.40:23151	tikv	352%	399%	311%
172.16.5.40:24799	pd	39%	45%	36%

图 573: Instance CPU Usage 报表

Instance Memory Usage

Instance Memory Usage 表显示各个 TiDB/PD/TiKV 进程占用内存字节数的平均值 (AVG)，最大值 (MAX)，最小值 (MIN)。

INSTANCE	JOB	AVG	MAX	MIN
172.16.5.40:20151	tikv	9.562 GB	9.605 GB	9.523 GB
172.16.5.40:23151	tikv	9.528 GB	9.574 GB	9.496 GB
172.16.5.40:22151	tikv	9.433 GB	9.498 GB	9.345 GB
172.16.5.40:10089	tidb	904.500 MB	911.660 MB	894.336 MB
172.16.5.40:24799	pd	61.089 MB	61.270 MB	60.980 MB

图 574: Instance Memory Usage 报表

TiKV Thread CPU Usage

TiKV Thread CPU Usage 表显示 TiKV 内部各个模块线程的 CPU 使用率的平均值 (AVG)、最大值 (MAX)、和最小值 (MIN)。这里进程 CPU 使用率最大值为 100% * 对应配置的线程数量。

TiKV Thread CPU Usage

METRIC_NAME	INSTANCE	AVG	MAX	MIN	CONFIG_KEY	CURRENT_CONFIG_VALUE
grpc expand		14.83%	16%	12.96%		
schedule worker fold		8.76%	9.47%	7.56%		
|-- sched_worker	10.0.1.12:20180	8.76%	9.47%	7.56%	storage.scheduler-worker-pool-size	4
storage read pool expand		3.24%	3.64%	2.67%		
storage read pool normal expand		3.22%	3.64%	2.67%		
Async apply fold		3.14%	3.4%	2.71%		
|-- Async apply	10.0.1.12:20180	3.14%	3.4%	2.71%	raftstore.apply-pool-size	2
raftstore expand		2.7%	2.93%	2.38%		
rocksdb expand		0.54%	2.13%	0%		
unified read pool expand		0.45%	4.24%	0%		
split_check expand		0.26%	0.84%	0%		
gc expand		0.04%	0.4%	0%		
storage read pool high expand		0.02%	0.04%	0%		
snapshot expand		0.004%	0.02%	0%		
cop normal						
cop high						
cop low						
storage read pool low expand		0%	0%	0%		
cop						

图 575: TiKV Thread CPU Usage 报表

上表中的字段解释如下：

- CONFIG_KEY：表示对应模块的相关线程数配置。
- CURRENT_CONFIG_VALUE：表示配置在生成报表时刻的当前值。

> **注意：**
>
> `CURRENT_CONFIG_VALUE` 是生成报告时的值，并不是报告时间范围内的值。目前不能获取历史时间某些配置的值。

TiDB/PD Goroutines Count

TiDB/PD Goroutines Count 表显示 TiDB/PD goroutines 数量的平均值 (AVG)，最大值 (MAX)，和最小值 (MIN)。如果 goroutines 数量超过 2000，说明该进程并发太高，会对整体请求的延迟有影响。

INSTANCE	JOB	AVG	MAX	MIN
172.16.5.40:10089	tidb	1434.33	1473	1394
172.16.5.40:24799	pd	157	157	157

图 576: TiDB/PD goroutines count 报表

概览信息

Time Consumed by Each Component

Time Consumed by Each Component 显示包括集群中 TiDB、PD、TiKV 各个模块的监控耗时以及各项耗时的占比。默认时间单位是秒。用户可以用该表快速定位哪些模块的耗时较多。

METRIC_NAME	LABEL	TIME_RATIO	TOTAL_TIME	TOTAL_COUNT	P999	P99	P90	P80
tidb_query expand		1	23223.86	459625	63.54	45.55	0.1	0.06
tidb_get_token expand		0.000001	0.03	458888	0.00004	0.000001	0.000001	0.000001
tidb_parse expand		0.00001	0.29	4603	0.0006	0.0004	0.0002	0.0001
tidb_compile expand		0.0008	17.82	463488	0.001	0.0009	0.0003	0.0002
tidb_execute expand		1	23237.39	463491	90.81	0.35	0.1	0.08
tidb_distsql_execution expand		0.0002	4.53	1064	0.13	0.1	0.02	0.01
tidb_cop expand		0.0005	12.39	1075	2.03	1.84	0.02	0.01
Transaction expand		1	23158.73	460023	8.16	7.91	5.38	0.06
tidb_transaction_local_latch_wait		0	0	0				
tidb_kv_backoff expand		0.00003	0.58	291	0.002	0.002	0.002	0.002
KV request expand		1.65	38325.58	2300949	2.03	1.84	0.04	0.03
Slow query expand		0.01	323.65	280	65.5	65.21	62.26	58.98
tidb_slow_query_cop_process		0	0	280	0.001	0.001	0.0009	0.0008
tidb_slow_query_cop_wait		0	0	280	0.001	0.001	0.0009	0.0008
DDL job		0	0	0				
tidb_ddl_worker		0	0	0				
tidb_ddl_update_self_version		0	0	0				
tidb_owner_handle_syncer		0	0	0				
Batch add index								
tidb_ddl_deploy_syncer		0	0	0				
Schema load expand		0.000008	0.19	27	0.06	0.06	0.06	0.05
TiDB meta operation expand		0.0001	3.21	1200	0.13	0.1	0.01	0.005
TiDB auto id request		0	0	0				
TiDB auto analyze expand		0.003	78.35	1	81.88	81.51	77.82	73.73
TiDB GC expand		0.000000001	0.00002	3	1	0.99	0.9	0.8
tidb_gc_push_task		0	0	0				
tidb_batch_client_unavailable		0	0	0				
tidb_batch_client_wait		0	0	0				
pd_tso_rpc expand		0.005	123.45	108540	0.02	0.008	0.003	0.002
pd_tso_wait expand		0.07	1662.29	920545	0.03	0.01	0.004	0.003
PD client cmd expand		0.15	3370.87	2761712	0.03	0.01	0.004	0.003
pd_client_request_rpc expand		0.005	123.45	108540	0.02	0.008	0.003	0.002
pd_grpc_completed_commands expand		0.00005	1.24	5255	0.01	0.01	0.006	0.005
pd_operator_finish								
pd_operator_step_finish								
Etcd transactions expand		0.000008	0.17	207	0.008	0.008	0.006	0.002
pd_region_heartbeat expand		0.00006	1.33	195	0	0	0	0
etcd_wal_fsync expand		0.00001	0.33	288	0.02	0.01	0.006	0.004
pd_peer_round_trip								

图 577: Total Time Consume 报表

上表各列的字段含义如下：

- METRIC_NAME：监控项的名称。
- Label：监控的 label 信息，点击 expand 后可以查看该项监控更加详细的各项 label 的监控信息。
- TIME_RATIO：该项为 TIME_RATIO 为 1 的监控行总时间与监控消耗的总时间的比例。如 kv_request 的总耗时占 tidb_query 总耗时的 1.65 = 38325.58/23223.86。因为 KV 请求会并行执行，所以所有 KV 请求的总时间有可能超过总查询 (tidb_query) 的执行时间。
- TOTAL_TIME：该项监控的总耗时。

- TOTAL_COUNT：该项监控执行的总次数。
- P999：该项监控的 P999 最大时间。
- P99：该项监控的 P99 最大时间。
- P90：该项监控的 P90 最大时间。
- P80：该项监控的 P80 最大时间。

以上监控中相关模块的耗时关系如下所示：

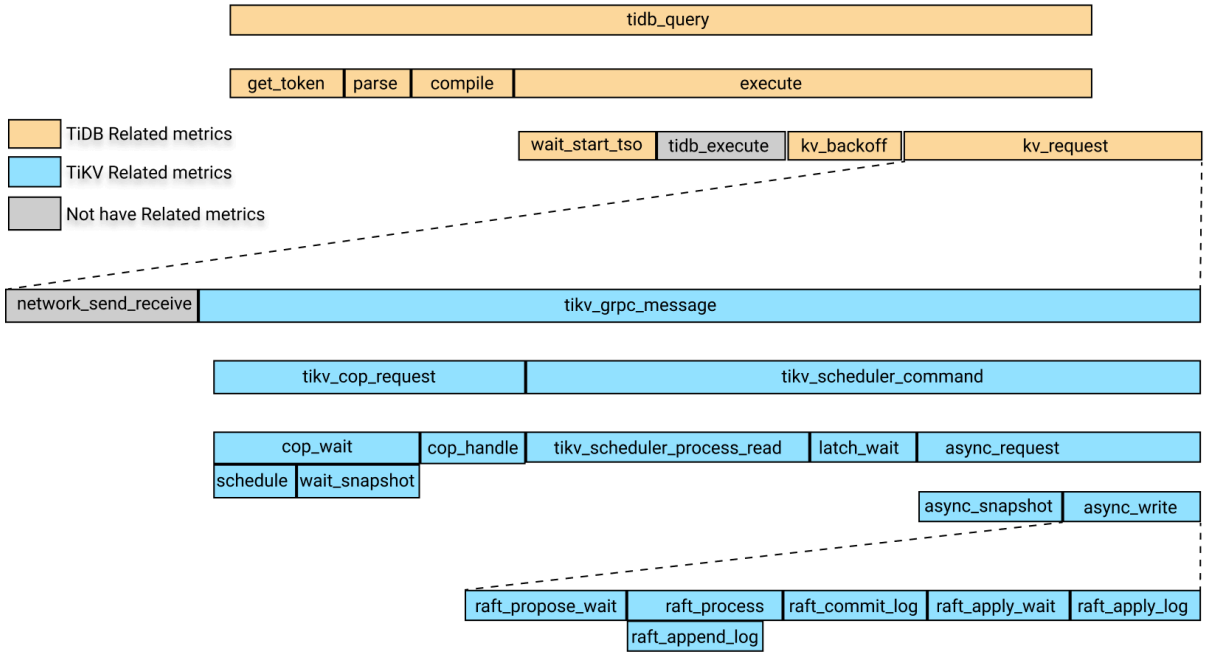


图 578: 各个模块耗时关系图

上图中，黄色部分是 TiDB 相关的监控，蓝色部分是 TiKV 相关的监控，灰色部分暂时没有具体对应的监控项。

上图中，tidb_query 的耗时包括以下部分的耗时：

- get_token
- parse
- compile
- execute

其中 execute 的耗时包括以下部分：

- wait_start_tso
- TiDB 层的执行时间，目前暂无监控
- KV 请求的时间
- tidb_kv_backoff 的时间，这是 KV 请求失败后进行 backoff 的时间

其中，KV 请求时间包含以下部分：

- 请求的网络发送以及接收耗时，目前该项暂无监控，可以大致用 KV 请求时间减去 `tikv_grpc_message` 的时间
- `tikv_grpc_message` 的耗时

其中，`tikv_grpc_message` 耗时包含以下部分：

- Coprocessor request 耗时，指用于处理 COP 类型的请求，该耗时包括以下部分：
  - `tikv_cop_wait`，请求排队等待的耗时
  - `Coprocessor handling request`，处理 COP 请求的耗时
- `tikv_scheduler_command` 耗时，该耗时包含以下部分：
  - `tikv_scheduler_processing_read`，处理读请求的耗时
  - `tikv_storage_async_request` 中获取 snapshot 的耗时（snapshot 是该项监控的 label）
  - 处理写请求的耗时，该耗时包括以下部分：
    * `tikv_scheduler_latch_wait`，等待 latch 的耗时
    * `tikv_storage_async_request` 中 write 的耗时（write 是该监控的 label）

其中，`tikv_storage_async_request` 中的 write 耗时是指 raft kv 写入的耗时，包括以下部分：

- `tikv_raft_propose_wait`
- `tikv_raft_process`，该耗时主要时间包括：
  - `tikv_raft_append_log`
  - `tikv_raft_commit_log`
  - `tikv_raft_apply_wait`
  - `tikv_raft_apply_log`

用户可以根据上述耗时之间的关系，利用 `TOTAL_TIME` 以及 P999，P99 的时间大致定位哪些模块耗时比较长，然后再看相关的监控。

> 注意：
>
> 由于 Raft KV 可能会将多个请求作为一个 batch 来写入，所以 `TOTAL_TIME` 不适用于来衡量 Raft KV 的写入相关监控项的耗时，这些监控项具体是 `tikv_raft_process`、`tikv_raft_append_log`、`tikv_raft_commit_log`、`tikv_raft_apply_wait`、`tikv_raft_apply_log`。此时用 P999 和 P99 的时间来对比各个模块的耗时更加合理。
>
> 原因是，假如有 10 个 async write 请求，Raft KV 内部将 10 个请求打包成一个 batch 执行，执行时间为 1 秒，所以每个请求的执行时间为 1 秒，10 个请求的总时间是 10 秒。但是 Raft KV 处理的总时间是 1 秒。如果用 `TOTAL_TIME` 来衡量，用户可能不明白剩余的 9 秒耗时在哪些模块下。这里从总请求数 (`TOTAL_COUNT`) 也能看出 Raft KV 的监控和其他监控的差异。

Errors Occurred in Each Component

Errors Occurred in Each Component 表显示包括 TiDB 和 TiKV 出现错误的总数。例如写 binlog 失败、`tikv server is busy`、`TiKV channel full`、`tikv write stall` 等错误，具体各项错误含义可以看行注释。

METRIC_NAME	LABEL	TOTAL_COUNT
tidb_binlog_error_total_count		0
tidb_handshake_error_total_count		0
tidb_transaction_retry_error_total_count		
tidb_kv_region_error_total_count expand		903
tidb_schema_lease_error_total_count		0
tikv_grpc_error_total_count		0
tikv_critical_error_total_count		
tikv_scheduler_is_busy_total_count		
tikv_channel_full_total_count		
tikv_coprocessor_request_error_total_count expand		1
tikv_engine_write_stall		0
tikv_server_report_failures_total_count expand		1396
tikv_storage_async_request_error expand		1176
tikv_lock_manager_detect_error_total_count		
tikv_backup_errors_total_count		
node_network_in_errors_total_count		0
node_network_out_errors_total_count		0

图 579: Errors Occurred in Each Component 报表

TiDB/PD/TiKV 的具体监控信息

这部分包括了 TiDB/PD/TiKV 更多的具体的监控信息。

TiDB 相关监控信息

Time Consumed by TiDB Component

Time Consumed by TiDB Component 表显示 TiDB 的各项监控耗时以及各项耗时的占比。和Time Consumed by Each Component 表类似，但是这个表的 label 信息会更丰富，细节更多。

TiDB Server Connections

TiDB Server Connections 表显示 TiDB 各个实例的客户端连接数。

TiDB Transaction

TiDB Transaction 表显示 TiDB 事务相关的监控。

METRIC_NAME	LABEL	TOTAL_VALUE	TOTAL_COUNT	P999	P99	P90	P80
tidb_transaction_retry_num		0	0	0	0	0	0
tidb_transaction_statement_num expand		6854588	6852371	4	4	3	2
tidb_txn_region_num expand		1070770	706887	3	2	2	2
tidb_txn_kv_write_num expand		513386	181312	234	2	2	2
tidb_txn_kv_write_size expand		266.772 MB	181296	116.913 KB	1.996 KB	1.905 KB	1.805 KB
tidb_load_safepoint_total_num expand		16					
tidb_lock_resolver_total_num expand		420514					

图 580: Transaction 报表

- TOTAL_VALUE：该项监控在报告时间段内所有值的和 (SUM)。
- TOTAL_COUNT：该项监控出现的总次数。
- P999: 该项监控的 P999 最大值。
- P99: 该项监控的 P99 最大值。
- P90: 该项监控的 P90 最大值。
- P80: 该项监控的 P80 最大值。

示例：

上表中，在报告时间范围的 tidb_txn_kv_write_size 表示一共约有 181296 次事务的 KV 写入，总 kv 写入大小是 266.772 MB，其中单次事务的 KV 写入的 P999、 P99、 P90、 P80 的最大值分别为 116.913 KB、 1.996 KB、 1.905 KB、 1.805 KB。

DDL Owner

MIN_TIME	DDL OWNER
2020-05-21 14:40:00	10.0.1.13:10080

图 581: TiDB DDL Owner 报表

上表表示从 2020-05-21 14:40:00 开始，集群的 DDL owner 在 10.0.1.13:10080 节点。如果 owner 发生变更，上表会有多行数据，其中 MinTime 列表示已知对应 Owner 的最小时间。

> 注意：
>
> 如果 owner 信息为空，不代表这个时间段内一定没有 owner，因为这里是依靠 ddl_worker 的监控信息来判断 DDL owner 的。也可能是这个时间段内 ddl_worker 没有做任何 DDL job 导致 owner 信息为空。

TiDB 中其他部分监控表如下：

- Statistics Info：TiDB 统计信息的相关监控。
- Top 10 Slow Query：报表时间范围内 Top 10 的慢查询信息。
- Top 10 Slow Query Group By Digest：报表时间范围内 Top 10 的慢查询信息，并按照 SQL 指纹聚合。
- Slow Query With Diff Plan：报表时间范围内执行计划发生变更的 SQL 语句。

PD 相关监控信息

PD 模块相关监控的报表如下：

- Time Consumed by PD Component：PD 中相关模块的耗时监控
- Scheduled Leader/Region：报表时间范围内集群发生的 `balance-region` 和 `balance leader` 监控，比如从 `tikv_note_1` 上调度走了多少个 leader，调度进了多少个 leader。
- Cluster Status：集群的状态信息，包括总 TiKV 数量、总集群存储容量、Region 数量、离线 TiKV 的数量等信息。
- Store Status：记录各个 TiKV 节点的状态信息，包括 Region score、 leader score、 Region/leader 的数量。
- etcd Status：PD 内部的 etcd 相关信息。

TiKV 相关监控信息

TIKV 模块的相关监控报表如下：

- Time Consumed by TiKV Component：TiKV 中相关模块的耗时监控。
- Time Consumed by RocksDB：TiKV 中 RocksDB 的耗时监控。
- TiKV Error：TiKV 中各个模块相关的 error 信息。
- TiKV Engine Size：TiKV 中各个节点 column family 的存储数据大小。
- Coprocessor Info：TiKV 中 Coprocessor 模块相关的监控。
- Raft Info：TiKV 中 Raft 模块的相关监控信息。
- Snapshot Info：TiKV 中 snapshot 相关监控信息。
- GC Info：TiKV 中 GC 相关的监控信息。
- Cache Hit：TiKV 中 Rocksdb 的各个缓存的命中率监控信息。

配置信息

配置信息中，部分模块的配置信息可以显示报告时间范围内的配置值，有部分配置则因为无法获取到历史的配置值，所以是生成报告时刻的当前配置值。

在报告时间范围内，以下表包括部分配置的在报告时间范围的开始时间的值：

- Scheduler Initial Config：PD 调度相关配置在报告开始时间的初始值。
- TiDB GC Initial Config：TiDB GC 相关配置在报告开始时间的初始值。
- TiKV RocksDB Initial Config：TiKV RocksDB 相关配置在报告开始时间的初始值。
- TiKV RaftStore Initial Config：TiKV RaftStore 相关配置在报告开始时间的初始值。

在报表时间范围内，如若有些配置被修改过，以下表包括部分配置被修改的记录：

- Scheduler Config Change History
- TiDB GC Config Change History

- TiKV RocksDB Config Change History
- TiKV RaftStore Config Change History

示例：

APPROXIMATE_CHANGE_TIME	CONFIG_ITEM	VALUE
2020-05-22T20:00:00+08:00	leader-schedule-limit	4
2020-05-22T20:07:00+08:00	leader-schedule-limit	8

图 582: Scheduler Config Change History 报表

上面报表显示，`leader-schedule-limit` 配置参数在报告时间范围内有被修改过：

- `2020-05-22T20:00:00+08:00`，即报告的开始时间 `leader-schedule-limit` 的配置值为 `4`，这里并不是指该配置被修改了，只是说明在报告时间范围的开始时间其配置值是 `4`。
- `2020-05-22T20:07:00+08:00`，`leader-schedule-limit` 的配置值为 `8`，说明在 `2020-05-22T20:07:00+08:00` 左右，该配置的值被修改了。

下面的报表是生成报告时，TiDB、PD、TiKV 的在生成报告时刻的当前配置：

- TiDB’s Current Config
- PD’s Current Config
- TiKV’s Current Config

对比报告

生成两个时间段的对比报告，其内容和单个时间段的报告是一样的，只是加入了对比列显示两个时间段的差别。下面主要介绍对比报告中的一些特有表以及如何查看对比报表。

首先在基本信息中的 Compare Report Time Range 报表会显示出对比的两个时间段：

Compare Report Time Range

T1.START_TIME	T1.END_TIME	T2.START_TIME	T2.END_TIME
2020-05-21 06:30:00	2020-05-21 06:35:00	2020-05-21 06:40:00	2020-05-21 06:45:00

图 583: Compare Report Time Range 报表

其中 `t1` 是正常时间段，或者叫参考时间段，`t2` 是异常时间段。

下面是一些慢查询相关的报表：

- Slow Queries In Time Range t2：仅出现在 `t2` 时间段但没有出现在 `t1` 时间段的慢查询。
- Top 10 slow query in time range t1：`t1` 时间段的 Top10 慢查询。
- Top 10 slow query in time range t2：`t2` 时间段的 Top10 慢查询。

DIFF_RATIO 介绍

本部分以 Instance CPU Usage 为例介绍 DIFF_RATIO。

Instance CPU Usage

INSTANCE	JOB	t1.AVG	t1.MAX	t1.MIN	t2.AVG	t2.MAX	t2.MIN	AVG_DIFF_RATIO	MAX_DIFF_RATIO	MIN_DIFF_RATIO
172.16.5.40:10089	tidb	410%	410%	410%	1240%	1240%	1240%	2.02	2.02	2.02
172.16.5.40:20151	tikv	221%	221%	221%	617%	617%	617%	1.79	1.79	1.79
172.16.5.40:20379	pd	82%	82%	82%	78%	78%	78%	-0.05	-0.05	-0.05

图 584: Compare Instance CPU Usage 报表

- t1.AVG、t1.MAX、t1.Min 分别是 t1 时间段内 CPU 使用率的平均值、最大值、最小值。
- t2.AVG、t2.MAX、t2.Min 分别是 t2 时间段内 CPU 使用率的平均值、最大值、最小值。
- AVG_DIFF_RATIO 表示 t1 和 t2 时间段平均值的 DIFF_RATIO。
- MAX_DIFF_RATIO 表示 t1 和 t2 时间段最大值的 DIFF_RATIO。
- MIN_DIFF_RATIO 表示 t1 和 t2 时间段最小值的 DIFF_RATIO。

DIFF_RATIO 表示两个时间段的差异大小，有以下几个取值方式：

- 如果该监控仅在 t2 时间内才有值，t1 时间段没有，则 DIFF_RATIO 取值为 1。
- 如果监控项仅在 t1 时间内才有值，t1 时间段没有，则 DIFF_RATIO 取值为 -1。
- 如果 t2 时间段的值比 t1 时间段的值大，则 DIFF_RATIO = (t2.value / t1.value)- 1。
- 如果 t2 时间段的值比 t1 时间段的值小，则 DIFF_RATIO = 1 - (t1.value / t2.value)。

例如上表中，tidb 节点的平均 CPU 使用率在 t2 时间段比 t1 时间段高 2.02 倍，2.02 = 1240/410 - 1。

Maximum Different Item 报表介绍

Maximum Different Item 的报表是对比两个时间段的监控项后，按照监控项的差异大小排序，通过这个表可以很快发现两个时间段哪些监控的差异最大。示例如下：

Maximum Different Item

The maximum different metrics between two time ranges.

TABLE	METRIC_NAME	LABEL	MAX_DIFF	t1.VALUE	t2.VALUE	VALUE_TYPE
TiKV, coprocessor_info Expand	TiKV Coprocessor scan	172.16.5.40:20151,select,get,lock	28916.85	61.33	1773532	TOTAL_VALUE
TiDB, statistics_info	store_query_feedback_total_count	172.16.5.40:10089,ok	7219.00		7219	TOTAL_COUNT
overview, total_time_consume	tidb_parse	general	1201.00		1201	TOTAL_COUNT
TiDB, tidb_time_consume	tidb_slow_query_cop_wait	172.16.5.40:10089	800.00		800	TOTAL_COUNT
overview, total_time_consume	tidb_slow_query_cop_process	172.16.5.40:10089	800.00		800	TOTAL_COUNT
overview, total_time_consume	Slow query	172.16.5.40:10089	799.00	1	800	TOTAL_COUNT
TiKV, coprocessor_info	TiKV Coprocessor response	172.16.5.40:20151	534.43	15.822 MB	8.273 GB	TOTAL_VALUE
overview, total_time_consume	tidb_distsql_execution	dag	470.06	0.68	320.32	TOTAL_TIME
load, tikv_thread_cpu_usage	unified read pool	172.16.5.40:20151	412.29	0.07%	28.93%	MIN
TiKV, coprocessor_info Expand	TiKV Coprocessor scan keys	172.16.5.40:20151,select	198.39	1067055.38	212755381.33	TOTAL_VALUE
overview, total_time_consume Expand	tidb_ddl_worker	drop view	139.00		139	TOTAL_COUNT
overview, total_error Expand	tikv_storage_async_request_error	snapshot	89.00		89	TOTAL_COUNT
overview, total_time_consume Expand	Coprocessor handling request	index	78.99	0.67	53.59	TOTAL_TIME
overview, total_time_consume	tidb_cop	172.16.5.40:10089	75.11	5.93	451.31	TOTAL_TIME
overview, total_time_consume	KV request	Cop	75.10	5.93	451.26	TOTAL_TIME
overview, total_time_consume Expand	Coprocessor request	select	73.12	4.73	350.6	TOTAL_TIME
PD, pd_time_consume Expand	pd_region_heartbeat	172.16.5.40:20150,1	-73.00	73		TOTAL_COUNT
overview, total_time_consume Expand	tikv_grpc_messge	coprocessor	71.75	5.56	404.49	TOTAL_TIME
TiDB, tidb_time_consume	tidb_ddl_update_self_version	172.16.5.40:10089,ok	71.00		71	TOTAL_COUNT
overview, total_time_consume Expand	TiDB meta operation	update_ddl_job	71.00		71	TOTAL_COUNT
overview, total_time_consume Expand	tidb_owner_handle_syncer	update_global_version	71.00		71	TOTAL_COUNT
TiDB, tidb_time_consume Expand	tidb_query	172.16.5.40:10089,internal	-62.84	15.96	0.25	P999
TiKV, scheduler_info Expand	Scheduler stage	172.16.5.40:20151,scan_lock,snapshot_ok	55.45		55.45	TOTAL_VALUE
overview, total_time_consume Expand	tikv_grpc_messge	kv_scan_lock	55.00		55	TOTAL_COUNT
TiKV, tikv_time_consume Expand	tikv_scheduler_command	172.16.5.40:20151,scan_lock	55.00		55	TOTAL_COUNT
overview, total_time_consume Expand	PD client cmd	get_region	44.00		44	TOTAL_COUNT
overview, total_time_consume Expand	tikv_cop_wait	select	39.40	171	6908	TOTAL_COUNT
TiDB, tidb_time_consume	Schema load	172.16.5.40:10089	37.50	0.02	0.77	TOTAL_TIME
overview, total_time_consume	Schema load	172.16.5.40:10089	30.25	0.008	0.25	P999

图 585: Maximum Different Item 报表

- `Table`：表示这个监控项来自于对比报告中报表，如 `TiKV, coprocessor_info` 表示是 TiKV 组件下的 `coprocessor_info` 报表。
- `METRIC_NAME`：监控项名，点击 expand 可以查看该监控的不同 label 的差异对比。
- `LABEL`：监控项对应的 label。比如 `TiKV Coprocessor scan` 监控项有两个 label，分别是 instance、req、tag、sql_type，分别表示为 TiKV 地址、请求类型、操作类型和操作的 column family。
- `MAX_DIFF`：差异大小，取值为 `t1.VALUE` 和 `t2.VALUE` 的 `DIFF_RATIO` 计算。

可以从上表中发现，`t2` 时间段比 `t1` 时间段多出了大量的 Coprocessor 请求，TiDB 的解析 SQL (parse) 时间也多了很多。

14.12.1.11.3 使用 TiDB Dashboard 诊断报告定位问题

本文介绍如何使用 TiDB Dashboard 诊断报告定位问题。

对比诊断功能示例

对比报告中对比诊断的功能，通过对比两个时间段的监控项差异来尝试帮助 DBA 定位问题。先看以下示例。

大查询/写入导致 QPS 抖动或延迟上升诊断

示例 1

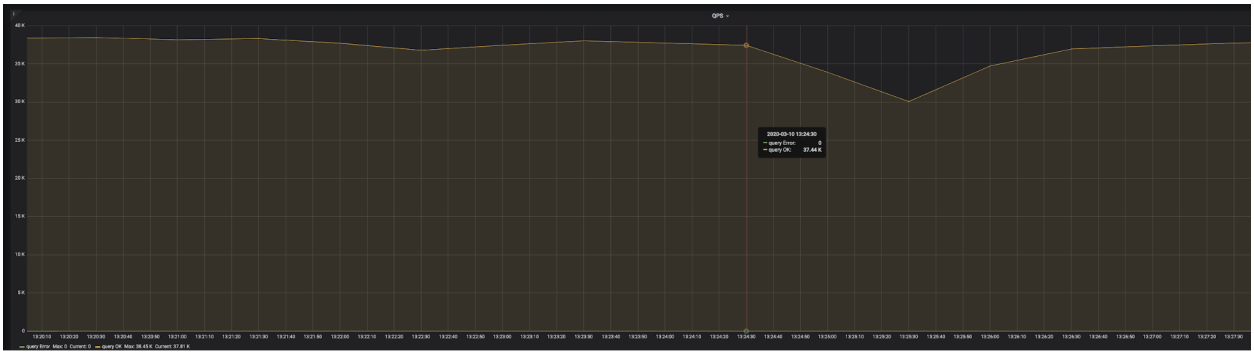


图 586: QPS 图

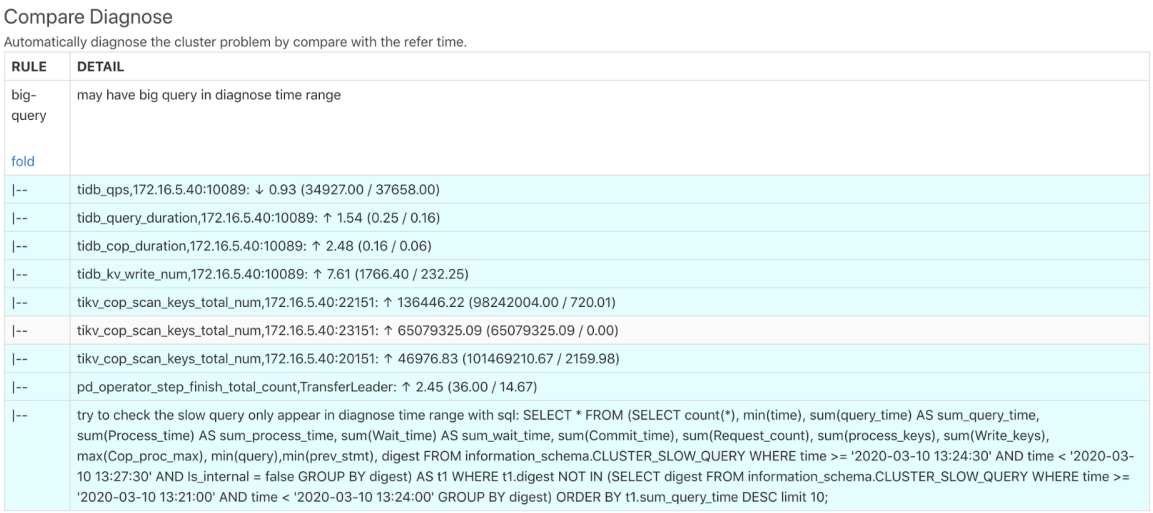
上图是 go-ycsb 压测的监控。可以发现，在 2020-03-10 13:24:30 时，QPS 突然开始下降，3 分钟后，QPS 开始恢复正常。

生成以下两个时间范围的对比报告：

- t1: 2020-03-10 13:21:00 - 2020-03-10 13:23:00，正常时间段，又叫参考时间段。
- t2: 2020-03-10 13:24:30 - 2020-03-10 13:27:30，QPS 开始下降的异常时间段。

这里两个时间间隔都是 3 分钟，因为抖动的影响范围为 3 分钟。因为诊断时会用一些监控的平均值做对比，所有间隔时间太长会导致平均值差异不明显，无法准确定位问题。

生成报告后查看 Compare Diagnose 报表内容如下：



Compare Diagnose

Automatically diagnose the cluster problem by compare with the refer time.

RULE	DETAIL
big-query fold	may have big query in diagnose time range
|--	tidb_qps,172.16.5.40:10089: ↓ 0.93 (34927.00 / 37658.00)
|--	tidb_query_duration,172.16.5.40:10089: ↑ 1.54 (0.25 / 0.16)
|--	tidb_cop_duration,172.16.5.40:10089: ↑ 2.48 (0.16 / 0.06)
|--	tidb_kv_write_num,172.16.5.40:10089: ↑ 7.61 (1766.40 / 232.25)
|--	tikv_cop_scan_keys_total_num,172.16.5.40:22151: ↑ 136446.22 (98242004.00 / 720.01)
|--	tikv_cop_scan_keys_total_num,172.16.5.40:23151: ↑ 65079325.09 (65079325.09 / 0.00)
|--	tikv_cop_scan_keys_total_num,172.16.5.40:20151: ↑ 46976.83 (101469210.67 / 2159.98)
|--	pd_operator_step_finish_total_count,TransferLeader: ↑ 2.45 (36.00 / 14.67)
|--	try to check the slow query only appear in diagnose time range with sql: SELECT * FROM (SELECT count(*), min(time), sum(query_time) AS sum_query_time, sum(Process_time) AS sum_process_time, sum(Wait_time) AS sum_wait_time, sum(Commit_time), sum(Request_count), sum(process_keys), sum(Write_keys), max(Cop_proc_max), min(query),min(prev_stmt), digest FROM information_schema.CLUSTER_SLOW_QUERY WHERE time >= '2020-03-10 13:24:30' AND time < '2020-03-10 13:27:30' AND Is_internal = false GROUP BY digest) AS t1 WHERE t1.digest NOT IN (SELECT digest FROM information_schema.CLUSTER_SLOW_QUERY WHERE time >= '2020-03-10 13:21:00' AND time < '2020-03-10 13:24:00' GROUP BY digest) ORDER BY t1.sum_query_time DESC limit 10;

图 587: 对比诊断结果

上面诊断结果显示，在诊断的时间内可能有大查询，下面的每一行的含义是：

- `tidb_qps`：QPS 下降 0.93 倍。
- `tidb_query_duration`：P999 的查询延迟上升 1.54 倍。
- `tidb_cop_duration`：P999 的 COP 请求的处理延迟上升 2.48 倍。
- `tidb_kv_write_num`：P999 的 tidb 的事务写入 kv 数量上升 7.61 倍。
- `tikv_cop_scan_keys_total_nun`：TiKV 的 coprocessor 扫描 key/value 的数量分别在 3 台 TiKV 上有很大的提升。
- `pd_operator_step_finish_total_count` 中，transfer leader 的数量上升 2.45 倍，说明异常时间段的调度比正常时间段要高。
- 提示可能有慢查询，并提示用 SQL 查询 TiDB 的慢日志。在 TiDB 中执行结果如下：

```
SELECT * FROM (SELECT count(*), min(time), sum(query_time) AS sum_query_time, sum(Process_time)
    ↪ AS sum_process_time, sum(Wait_time) AS sum_wait_time, sum(Commit_time), sum(Request_count
    ↪ ), sum(process_keys), sum(Write_keys), max(Cop_proc_max), min(query),min(prev_stmt),
    ↪ digest FROM information_schema.CLUSTER_SLOW_QUERY WHERE time >= '2020-03-10 13:24:30' AND
    ↪  time < '2020-03-10 13:27:30' AND Is_internal = false GROUP BY digest) AS t1 WHERE t1.
    ↪ digest NOT IN (SELECT digest FROM information_schema.CLUSTER_SLOW_QUERY WHERE time >= '
    ↪ 2020-03-10 13:21:00' AND time < '2020-03-10 13:24:00' GROUP BY digest) ORDER BY t1.
    ↪ sum_query_time DESC limit 10\G
***************************[ 1. row ]***************************
count(*)            | 196
min(time)           | 2020-03-10 13:24:30.204326
sum_query_time      | 46.878509117
sum_process_time    | 265.924
sum_wait_time       | 8.308
sum(Commit_time)    | 0.926820886
sum(Request_count) | 6035
```

```
sum(process_keys)   | 201453000
sum(Write_keys)     | 274500
max(Cop_proc_max)   | 0.263
min(query)          | delete from test.tcs2 limit 5000;
min(prev_stmt)      |
digest              | 24bd6d8a9b238086c9b8c3d240ad4ef32f79ce94cf5a468c0b8fe1eb5f8d03df
```

可以发现，从 13:24:30 开始有一个批量删除的大写入，一共执行了 196 次，每次删除 5000 行数据，总共耗时 46.8 秒。

示例 2

如果大查询一直没执行完，就不会记录慢日志，但仍可以进行诊断，示例如下：



图 588: QPS 图

上图中，也是在跑 go-ycsb 的压测。可以发现，在 2020-03-08 01:46:30 时，QPS 突然开始下降，并且一直没有恢复。

生成以下两个时间范围的对比报告：

- t1: 2020-03-08 01:36:00 - 2020-03-08 01:41:00，正常时间段，又叫参考时间段。
- t2: 2020-03-08 01:46:30 - 2020-03-08 01:51:30，QPS 开始下降的异常时间段。

生成报告后看 Compare Diagnose 报表的内容如下：

Compare Diagnose

Automatically diagnose the cluster problem by compare with the refer time.

RULE	DETAIL
big-query fold	may have big query in diagnose time range
|--	tidb_qps,172.16.5.40:10089: ↓ 0.85 (31881.00 / 37674.00)
|--	tidb_query_duration,172.16.5.40:10089: ↑ 1.35 (0.06 / 0.04)
|--	tidb_cop_duration,172.16.5.40:10089: ↑ 3.78 (0.08 / 0.02)
|--	tidb_kv_write_num,172.16.5.40:10089: ↑ 511.74 (511.74 / 0.00)
|--	tikv_cop_scan_keys_total_num,172.16.5.40:20151: ↑ 1277.21 (6270285.33 / 4909.36)
|--	try to check the slow query only appear in diagnose time range with sql: SELECT * FROM information_schema.cluster_log WHERE type='tidb' AND time >= '2020-03-08 01:46:30' AND time < '2020-03-08 01:51:30' AND level = 'warn' AND message LIKE '%expensive_query%'

图 589: 对比诊断结果

上面诊断结果的最后一行显示可能有慢查询，并提示用 SQL 查询 TiDB 日志中的 expensive query。在 TiDB 中执行结果如下：

```
> SELECT * FROM information_schema.cluster_log WHERE type='tidb' AND time >= '2020-03-08 01:46:30
    ↪ ' AND time < '2020-03-08 01:51:30' AND level = 'warn' AND message LIKE '%expensive_query%
    ↪ '\G
TIME     | 2020/03/08 01:47:35.846
TYPE     | tidb
INSTANCE | 172.16.5.40:4009
LEVEL    | WARN
MESSAGE  | [expensivequery.go:167] [expensive_query] [cost_time=60.085949605s] [process_time=2.52
    ↪ s] [wait_time=2.52s] [request_count=9] [total_keys=996009] [process_keys=996000] [
    ↪ num_cop_tasks=9] [process_avg_time=0.28s] [process_p90_time=0.344s] [process_max_time
    ↪ =0.344s] [process_max_addr=172.16.5.40:20150] [wait_avg_time=0.000777777s] [wait_p90_time
    ↪ =0.003s] [wait_max_time=0.003s] [wait_max_addr=172.16.5.40:20150] [stats=t_wide:pseudo] [
    ↪ conn_id=19717] [user=root] [database=test] [table_ids="[80,80]"] [txn_start_ts
    ↪ =415132076148785201] [mem_max="23583169 Bytes (22.490662574768066 MB)"] [sql="select
    ↪ count(*) from t_wide as t1 join t_wide as t2 where t1.c0>t2.c1 and t1.c2>0"]
```

以上查询结果显示，在 172.16.5.40:4009 这台 TiDB 上，2020/03/08 01:47:35.846 有一个已经执行了 60s 但还没有执行完的 expensive_query。

用对比报告定位问题

诊断有可能是误诊，使用对比报告或许可以帮助 DBA 更快速的定位问题。参考以下示例。

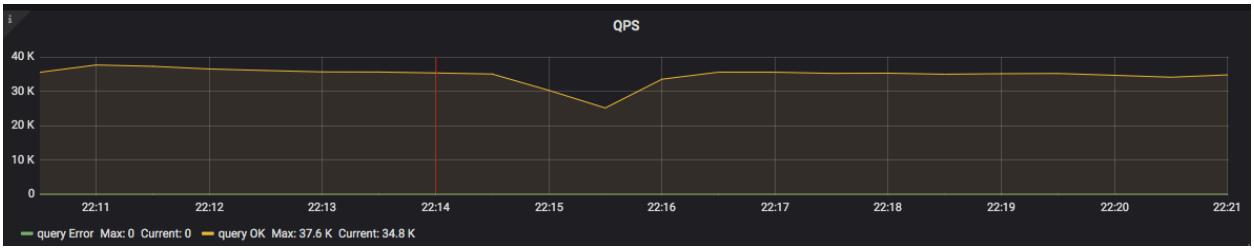


图 590: QPS 图

上图中，也是在跑 go-ycsb 的压测，可以发现，在 2020-05-22 22:14:00 时，QPS 突然开始下降，大概在持续 3 分钟后恢复。

生成以下 2 个时间范围的对比报告：

- t1: 2020-05-22 22:11:00 - 2020-05-22 22:14:00，正常时间段。
- t2: 2020-05-22 22:14:00 - 2020-05-22 22:17:00，QPS 开始下降的异常时间段。

生成对比报告后，查看 Max diff item 报表，该报表对比两个时间段的监控项后，按照监控项的差异大小排序，这个表的结果如下：

Maximum Different Item

The maximum different metrics between two time ranges.

TABLE	METRIC_NAME	LABEL	MAX_DIFF	t1.VALUE	t2.VALUE	VALUE_TYPE
TiDB, transaction	Transaction KV write count	172.16.5.40:10089	-21616.00	43234	2	P999
TiKV, coprocessor_info Expand	TiKV Coprocessor scan	172.16.5.40:20151,select,get,lock	10499.24	64	672015.48	TOTAL_VALUE
TiKV, coprocessor_info Expand	TiKV Coprocessor scan keys	172.16.5.40:20151,select	4264.45	15431.85	65823838.67	TOTAL_VALUE
TiDB, transaction	Transaction KV write size	172.16.5.40:10089	-2702.69	5.278 MB	1.999 KB	P999
overview, total_time_consume	Coprocessor handling request	select	2132.50	0.06	128.01	TOTAL_TIME
TiKV, coprocessor_info	TiKV Coprocessor response	172.16.5.40:20151	1535.00	1.792 MB	2.688 GB	TOTAL_VALUE
TiDB, statistics_info	store_query_feedback_total_count	172.16.5.40:10089,ok	1447.00		1447	TOTAL_COUNT
TiKV, scheduler_info Expand	tikv_scheduler_keys_written	172.16.5.40:20151,commit	-860.00	861	1	P99
TiKV, tikv_time_consume	Coprocessor request	172.16.5.40:20151,select	853.87	0.15	128.23	TOTAL_TIME
load, tikv_thread_cpu_usage	unified read pool	172.16.5.40:20151	667.27	0.11%	73.51%	AVG
TiKV, tikv_time_consume	Coprocessor handling request	172.16.5.40:20151,index	444.00	0.002	0.89	P999
TiKV, tikv_time_consume Expand	tikv_grpc_messge	172.16.5.40:20151,coprocessor	419.97	0.33	138.92	TOTAL_TIME
load, node_load_info	tikv_disk_read_bytes	172.16.5.40:19110,sda	-273.41	0.267 KB	0.000 KB	MAX
overview, total_time_consume Expand	KV request	Cop	244.29	0.66	161.89	TOTAL_TIME

图 591: 对比结果

从上面结果可以看出 t2 时间段新增了很多 Coprocessor 请求，可以猜测可能是 t2 时间段出现了一些大查询，或者是查询较多的负载。

实际上，在 t1 - t2 整个时间段内都在进行 go-ycsb 的压测，然后在 t2 时间段跑了 20 个 tpch 的查询，所以是因为 tpch 大查询导致了出现很多的 cop 请求。

这种大查询执行时间超过慢日志的阈值后也会记录在慢日志里面，可以继续查看 Slow Queries In Time ↪ Range t2 报表查看是否有一些慢查询。一些在 t1 时间段存在的查询，可能在 t2 时间段内就变成了慢查询，是因为 t2 时间段的其他负载影响导致该查询的执行变慢。

14.12.1.12 TiDB Dashboard 监控页面

从 TiDB Dashboard 监控页面，你可以查看性能分析和优化工具 Performance Overview 面板。借助 Performance Overview 面板，你可以高效地进行性能分析，确认用户响应时间的瓶颈是否在数据库中。如果数据库是整个系统的瓶颈，你可以通过数据库时间概览和 SQL 延迟的分解，定位数据库内部的瓶颈点。详情请参考TiDB 性能分析和优化方法。

14.12.1.12.1 访问页面

登录 Dashboard 后点击左侧导航的 Monitoring（监控）可以进入此功能页面：

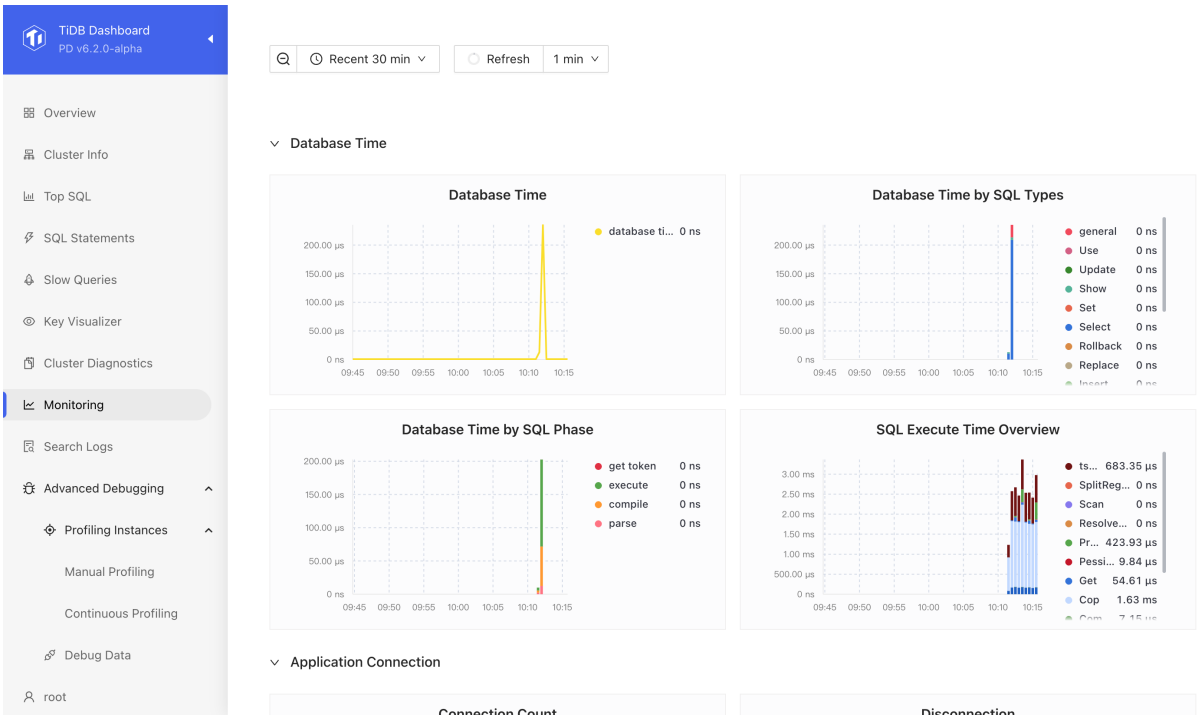


图 592: Monitoring page

如果你使用 TiUP 部署 TiDB，你也可以在 Grafana 上查看 Performance Overview 面板。监控架构参见TiDB 监控框架概述。



图 593: performance overview

14.12.1.12.2 面板监控指标详解

Performance Overview 面板按总分结构对 TiDB、TiKV 和 PD 的性能指标进行了编排组织，包含以下三部分内容：

- 总的概览：数据库时间和 SQL 执行时间概览。通过颜色优化法，你可以快速识别数据库负载特征和性能瓶颈。
- 资源负载：关键指标和资源利用率，包含数据库 QPS、应用和数据库的连接信息和请求命令类型、数据库内部 TSO 和 KV 请求 OPS、TiDB 和 TiKV 的资源使用概况。
- 自上而下的延迟分解：Query 延迟和连接空闲时间对比、Query 延迟分解、execute 阶段 TSO 请求和 KV 请求的延迟、TiKV 内部写延迟的分解等。

以下为 Performance Overview 面板监控说明：

Database Time by SQL Type

- database time: 每秒的总数据库时间
- sql_type: 每种 SQL 语句每秒消耗的数据库时间

Database Time by SQL Phase

- database time: 每秒的总数据库时间
- get token/parse/compile/execute: 4 个 SQL 处理阶段每秒消耗的数据库时间

execute 执行阶段为绿色，其他三个阶段偏红色系，如果非绿色的颜色占比明显，意味着在执行阶段之外数据库消耗了过多时间，需要进一步分析根源。

SQL Execute Time Overview

- execute time: execute 阶段每秒消耗的数据库时间
- tso_wait: execute 阶段每秒同步等待 TSO 的时间
- kv request type: execute 阶段每秒等待每种 KV 请求类型的时间，总的 KV request 等待时间可能超过 execute time，因为 KV request 是并发的。

绿色系标识代表常规的写 KV 请求（例如 Prewrite 和 Commit），蓝色系标识代表常规的读 KV 请求，其他色系标识需要注意的问题。例如，悲观锁加锁请求为红色，TSO 等待为深褐色。如果非蓝色系或者非绿色系占比明显，意味着执行阶段存在异常的瓶颈。例如，当发生严重锁冲突时，红色的悲观锁时间会占比明显；当负载中 TSO 等待的消耗时间过长时，深褐色会占比明显。

QPS

QPS：按 SELECT、INSERT、UPDATE 等类型统计所有 TiDB 实例上每秒执行的 SQL 语句数量

CPS By Type

CPS By Type：按照类型统计所有 TiDB 实例每秒处理的命令数（Command Per Second）

Queries Using Plan Cache OPS

Queries Using Plan Cache OPS：所有 TiDB 实例每秒使用 Plan Cache 的查询数量

KV/TSO Request OPS

- kv request total: 所有 TiDB 实例每秒总的 KV 请求数量
- kv request by type: 按 Get、Prewrite、Commit 等类型统计在所有 TiDB 实例每秒的请求数据

- tso - cmd：在所有 TiDB 实例每秒 tso cmd 的请求数量
- tso - request：在所有 TiDB 实例每秒 tso request 的请求数量

通常 tso - cmd 除以 tso - request 等于平均请求的 batch 大小。

Connection Count

- total：所有 TiDB 的连接数
- active connections：所有 TiDB 总的活跃连接数
- 各个 TiDB 的连接数

TiDB CPU

- avg：所有 TiDB 实例平均 CPU 利用率
- delta：所有 TiDB 实例中最大 CPU 利用率减去所有 TiDB 实例中最小 CPU 利用率
- max：所有 TiDB 实例中最大 CPU 利用率

TiKV CPU/IO MBps

- CPU-Avg：所有 TiKV 实例平均 CPU 利用率
- CPU-Delta：所有 TiKV 实例中最大 CPU 利用率减去所有 TiKV 实例中最小 CPU 利用率
- CPU-MAX：所有 TiKV 实例中最大 CPU 利用率
- IO-Avg：所有 TiKV 实例平均 MBps
- IO-Delta：所有 TiKV 实例中最大 MBps 减去所有 TiKV 实例中最小 MBps
- IO-MAX：所有 TiKV 实例中最大 MBps

Duration

- Duration：执行时间解释
  - 从客户端网络请求发送到 TiDB，到 TiDB 执行结束后返回给客户端的时间。一般情况下，客户端请求都是以 SQL 语句的形式发送，但也可以包含 COM_PING、COM_SLEEP、COM_STMT_FETCH、COM_SEND_LONG_DATA 之类的命令执行时间。
  - 由于 TiDB 支持 Multi-Query，因此，客户端可以一次性发送多条 SQL 语句，如 `select 1; select 1; select 1;`。此时的执行时间是所有 SQL 语句执行完成的总时间。
- avg：所有请求命令的平均执行时间
- 99：所有请求命令的 P99 执行时间
- avg by type：按 SELECT、 INSERT、 UPDATE 类型统计所有 TiDB 实例上所有请求命令的平均执行时间

Connection Idle Duration

Connection Idle Duration 指空闲连接的持续时间。

- avg-in-txn：处于事务中，空闲连接的平均持续时间

- avg-not-in-txn：没有处于事务中，空闲连接的平均持续时间
- 99-in-txn：处于事务中，空闲连接的 P99 持续时间
- 99-not-in-txn：没有处于事务中，空闲连接的 P99 持续时间

Parse Duration、 Compile Duration 和 Execute Duration

- Parse Duration：SQL 语句解析耗时统计
- Compile Duration：将解析后的 SQL AST 编译成执行计划的耗时
- Execution Duration：执行 SQL 语句执行计划耗时

这三个时间指标均包含均所有 TiDB 实例的平均值和 P99 值。

Avg TiDB KV Request Duration

按 `Get`、 `Prewrite`、 `Commit` 等类型统计在所有 TiDB 实例 KV 请求的平均执行时间。

Avg TiKV GRPC Duration

按 `get`、 `kv_prewrite`、 `kv_commit` 等类型统计所有 TiKV 实例对 gRPC 请求的平均执行时间。

PD TSO Wait/RPC Duration

- wait - avg：所有 TiDB 实例等待从 PD 返回 TSO 的平均时间
- rpc - avg：所有 TiDB 实例从向 PD 发送获取 TSO 的请求到接收到 TSO 的平均耗时
- wait - 99：所有 TiDB 实例等待从 PD 返回 TSO 的 P99 时间
- rpc - 99：所有 TiDB 实例从向 PD 发送获取 TSO 的请求到接收到 TSO 的 P99 耗时

Storage Async Write Duration、 Store Duration 和 Apply Duration

- Storage Async Write Duration：异步写所花费的时间
- Store Duration：异步写 Store 步骤所花费的时间
- Apply Duration：异步写 Apply 步骤所花费的时间

这三个时间指标都包含所有 TiKV 实例的平均值和 P99 值

平均 Storage async write duration = 平均 Store Duration + 平均 Apply Duration

Append Log Duration、 Commit Log Duration 和 Apply Log Duration

- Append Log Duration： Raft append 日志所花费的时间
- Commit Log Duration： Raft commit 日志所花费的时间
- Apply Log Duration： Raft apply 日志所花费的时间

这三个时间指标均包含所有 TiKV 实例的平均值和 P99 值。

14.12.1.13 TiDB Dashboard 日志搜索页面

该页面上允许用户在集群中搜索所有节点上的日志，在页面上预览搜索结果和下载日志。

14.12.1.13.1 访问

登录 Dashboard 后点击左侧导航的日志搜索可以进入此功能页面：

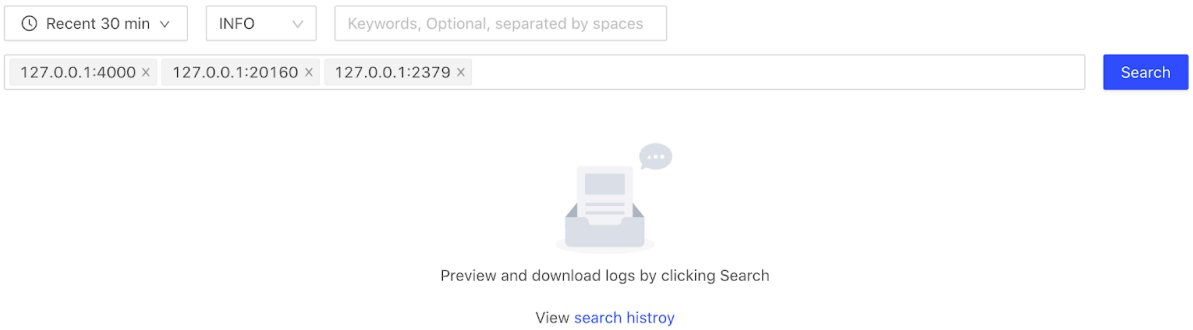


图 594: 日志搜索首页

该页面提供 4 个搜索参数，包括：

- 时间范围：限定搜索时间范围内的日志，默认值为最近 30 分钟
- 日志等级：限定最低日志等级，搜索该日志等级以上的所有日志，默认值为 INFO 等级
- 关键词：任何合法的任何字符串，可选，关键词之间以空格分割，支持正则表达式（大小写不敏感）
- 组件：选择要搜索的集群组件，多选，非空，默认选中所有组件

点击搜索按钮后，会进入搜索结果的详情页。

14.12.1.13.2 搜索结果详情

搜索结果详情页面如下图所示：



图 595: 搜索结果

整个页面分为三个部分：

- 搜索参数选项：与搜索页面的参数选项相同，可以在表单中重新选择参数并开始一次新的搜索（对应上图中框选的区域 1）
- 搜索进度：在页面右侧，展示当前搜索的进度，包括每个节点日志搜索状态和统计等（对应上图中框选的区域 2）
- 搜索结果（对应上图中框选的区域 3）：
  - 时间：日志产生的时间，时区与前端用户所处时区相同
  - 日志等级：日志等级
  - 组件类型：显示组件名及其地址
  - 日志：每条日志记录的正文部分，不包含日志的时间和日志等级。过长的日志会自动截断，鼠标单击该行可以展开查看完整内容，完整日志最长显示 512 个字符

> 注意：
>
> 在该页面上最多只显示 500 条搜索结果，完整的搜索结果可以通过下载得到。

搜索进度

在搜索进度区域中，对一个节点的搜索称为一次搜索任务，对搜索任务的状态分为：

- 运行中：开始搜索后，所有任务会进入运行中状态

- 成功：任务完成后自动转到成功，此时日志已缓存在 Dashboard 后端所在的本地磁盘中，可以提供给前端下载
- 失败：用户主动取消，或者某种原因报错退出的任务进入失败状态。任务失败时会自动清理本地临时文件

搜索进度区域包含三个控制按钮：

- 下载选中日志：下载被勾选组件的日志（只有已完成的才能被勾选），返回 tar 文件，解压得到一个或多个 zip 文件（每个组件对应一个 zip 文件），解压 zip 得到 log 文本文件
- 取消：取消所有正在运行的任务，只能在有运行中的任务时才能点击
- 重试：重试所有失败的任务，只有在有失败任务且无正在运行的任务时才能点击

14.12.1.13.3 搜索历史列表

在日志搜索首页点击查看搜索历史链接，进入搜索历史列表页面：

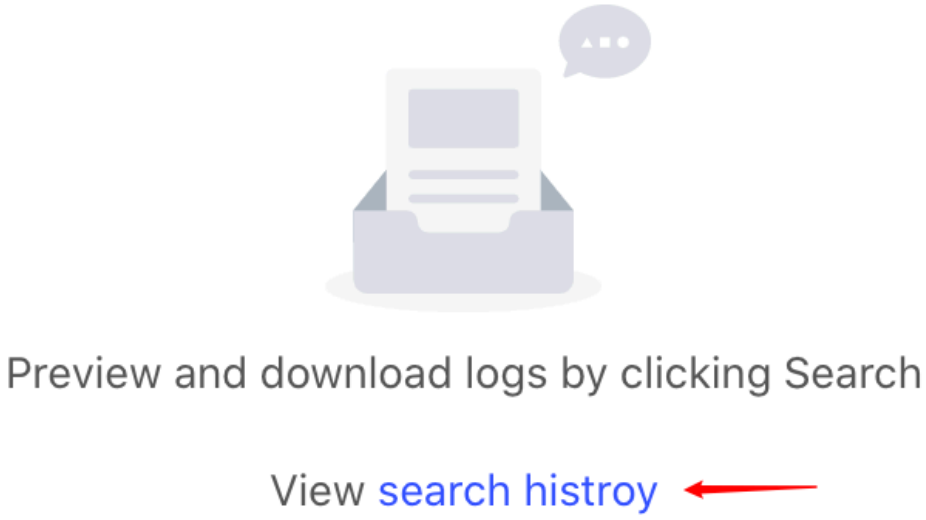


图 596: 搜索历史入口

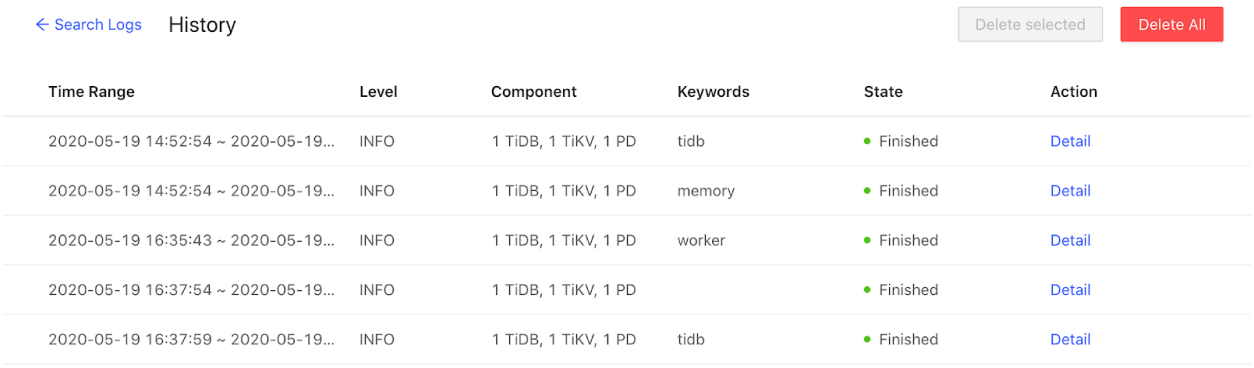


图 597: 搜索历史列表

该列表显示每次搜索的日志的时间范围、日志等级、组件、关键字以及搜索状态等信息。点击操作列的查看详情链接将跳转到此次搜索的结果详情页面。

可以对不再需要的搜索历史执行删除操作，点击右上角的删除全部任务，或者先选中需要删除的某些行后再点击删除选中的任务进行删除：

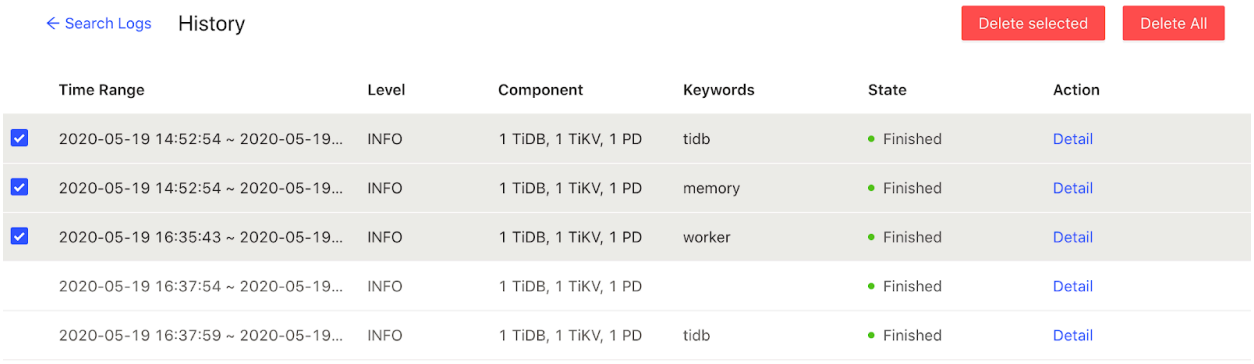


图 598: 删除搜索历史

14.12.1.14 实例性能分析

14.12.1.14.1 TiDB Dashboard 实例性能分析 - 手动分析页面

> **注意：**
>
> 该功能面向数据库技术专家，建议非专家用户在 PingCAP 专业技术人员的指导下使用该功能。

该页面允许用户按需地一键收集 TiDB、 TiKV、 PD、 TiFlash 各个实例的当前性能数据。收集到的性能数据可显示为火焰图或有向无环图形式。

通过这些性能数据，技术专家可以分析实例当前的 CPU、内存等资源消耗细节，帮助解决诸如当前 CPU 开销较高、内存占用较大、进程不明原因卡死等复杂问题。

启动分析后，TiDB Dashboard 将会收集当前一小段时间（默认 30 秒）的性能数据，因而只能用于分析集群当前正在面临的问题，对于当前已经不再复现的问题没有显著效果。若你想要收集或分析过去任意时刻性能数据，不希望每次都人工介入按需分析，请使用持续性能分析功能。

支持的性能数据

目前支持收集并分析以下性能数据：

- CPU：TiDB、 TiKV、 PD、 TiFlash 实例上各个内部函数的 CPU 开销情况

> ARM 环境中暂不支持对 TiKV 和 TiFlash 的 CPU 开销情况进行分析。

- Heap：TiDB、 PD 实例上各个内部函数的内存占用开销情况
- Mutex：TiDB、 PD 实例上各个处于等待状态的 Mutex 情况
- Goroutine：TiDB、 PD 实例上各个 Goroutine 的运行状态及调用栈情况

访问页面

可以通过以下两种方法访问实例性能分析页面：

- 登录后，左侧导航条点击高级调试 (Advanced Debugging) > 实例性能分析 (Profile Instances) > 手动分析 (Manual Profiling)：

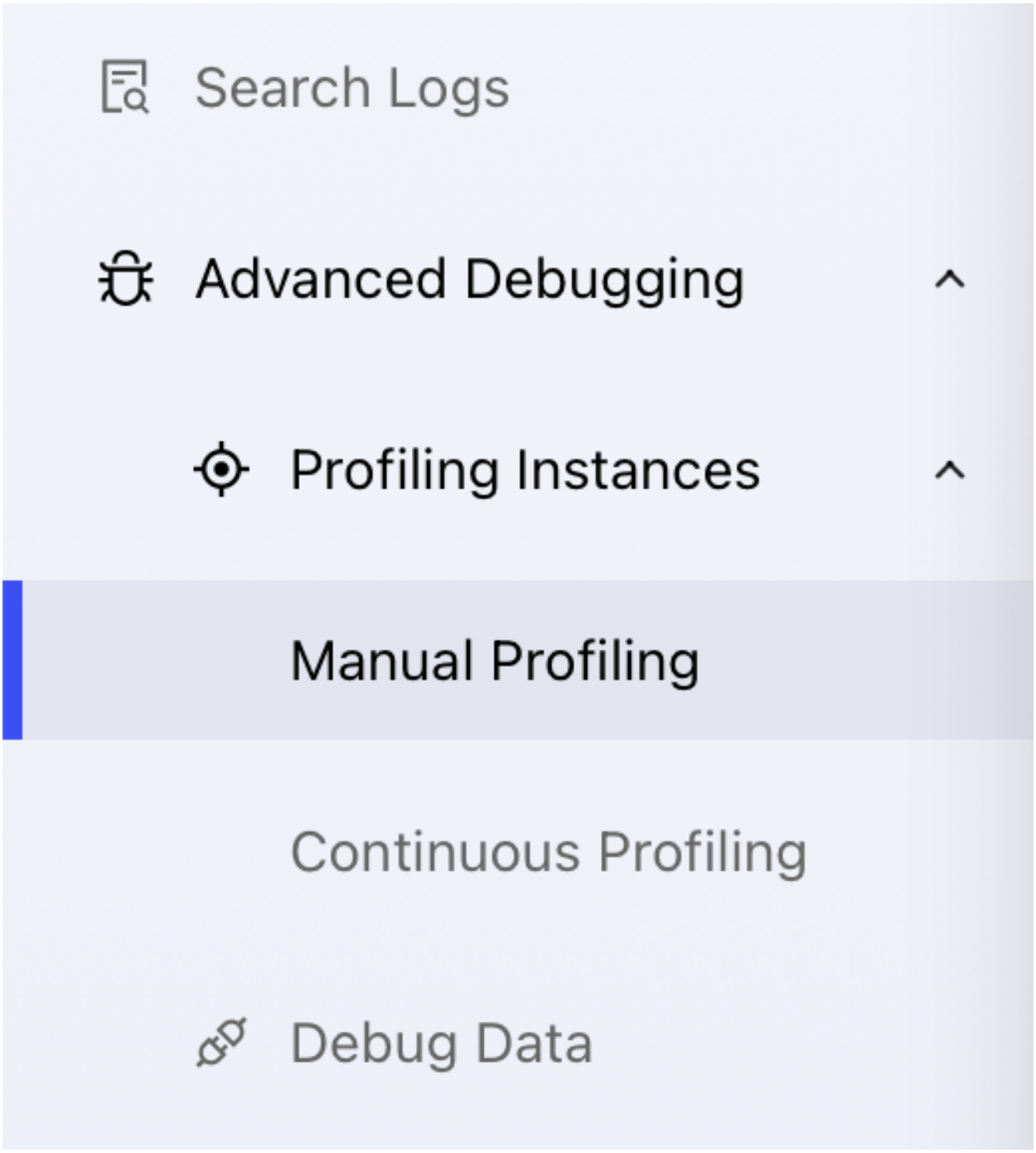


图 599: 访问页面

- 在浏览器中访问 http://127.0.0.1:2379/dashboard/#/instance_profiling (将 `127.0.0.1:2379` 替换为实际 PD 实例地址和端口)。

开始性能分析

在实例性能分析页面，选择至少一个目标实例和分析类型，确定性能分析时长 (默认为 30 秒)。点击开始分

析 (Start Profiling)：

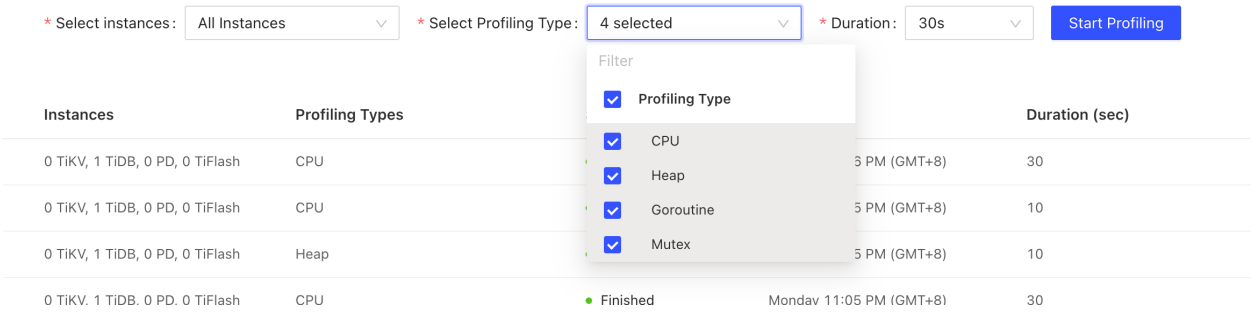


图 600: 开始分析

在已经启用了持续性能分析的集群上，手动性能分析不再可用。若要在启用了持续性能分析功能的集群上获取当前时刻的集群性能数据，请查看持续性能分析页面上最近一次的分析结果。

查看性能分析状态

开始性能分析后，可以看到实时性能分析状态和进度：

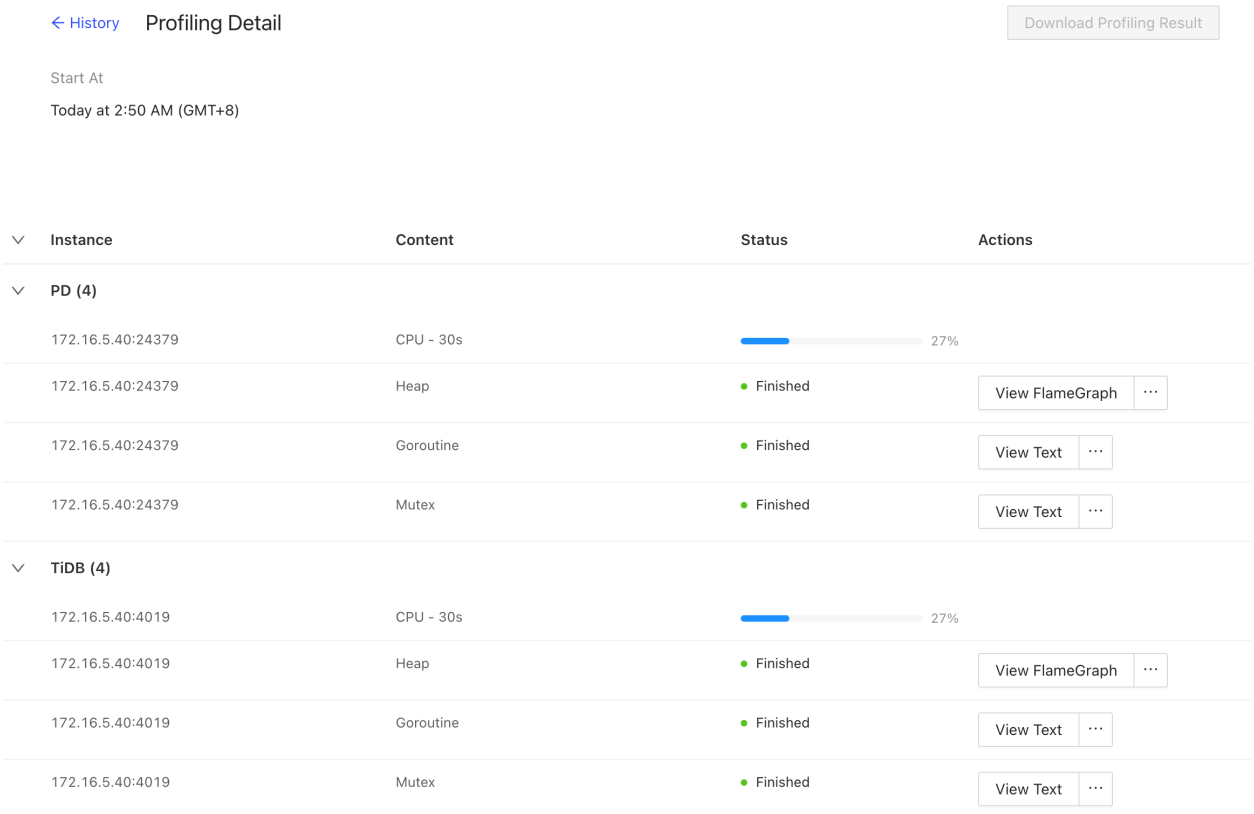


图 601: 实时状态

性能分析会在后台运行，刷新或退出当前页面不会终止已经运行的性能分析任务。

下载性能数据

所有实例的性能分析都完成后，可点击右上角下载按钮 (Download Profiling Result) 打包下载所有性能数据：

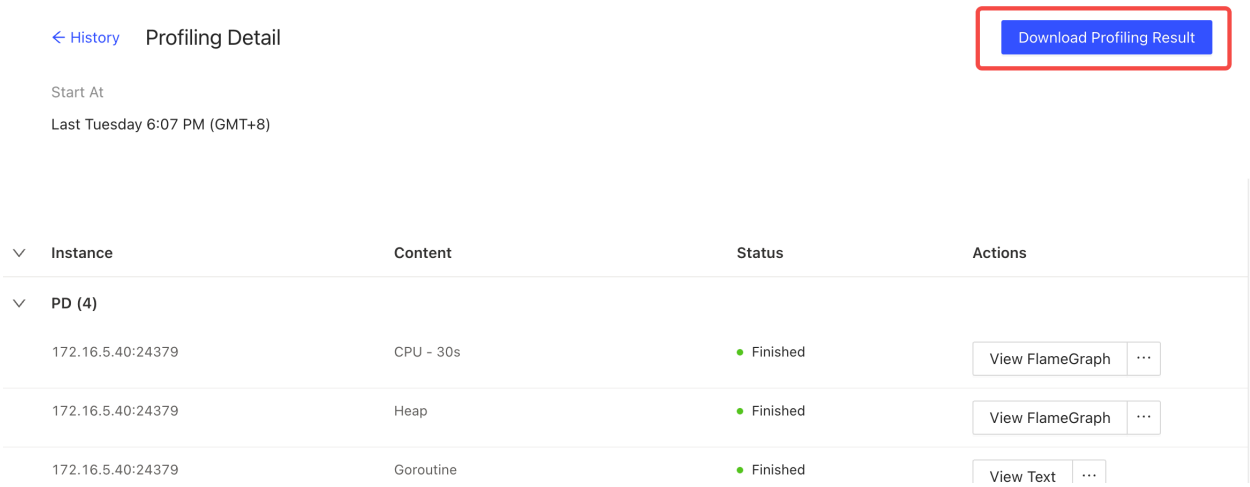


图 602: 下载分析结果

也可以点击列表中的单个实例查看其性能数据，或者悬浮到 ··· 按钮上下载原始数据：

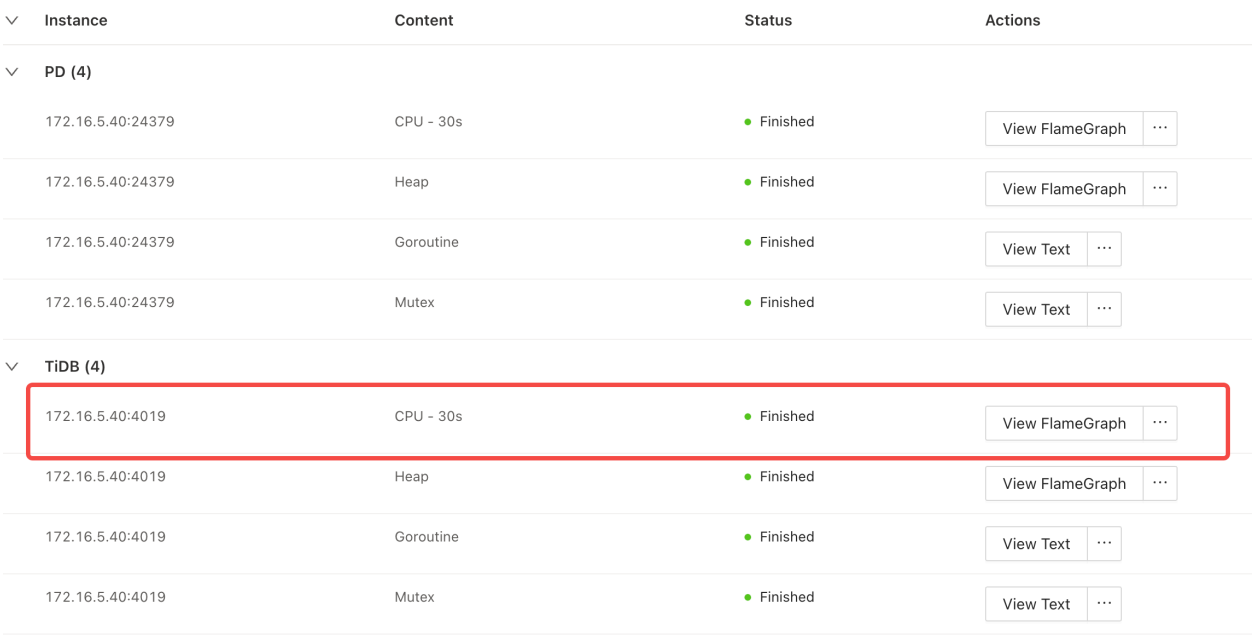
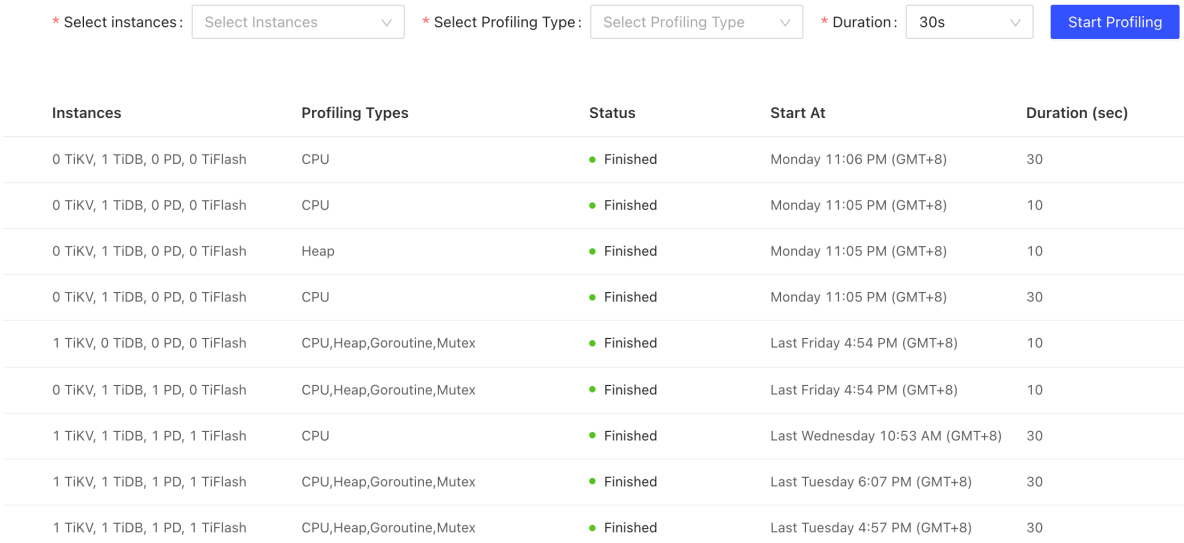


图 603: 在线查看分析结果

查看历史性能数据

在性能分析页面下方，列出了你手动发起的性能分析历史。点击任意一行，即可查看其状态详情：



Instances	Profiling Types	Status	Start At	Duration (sec)
0 TiKV, 1 TiDB, 0 PD, 0 TiFlash	CPU	Finished	Monday 11:06 PM (GMT+8)	30
0 TiKV, 1 TiDB, 0 PD, 0 TiFlash	CPU	Finished	Monday 11:05 PM (GMT+8)	10
0 TiKV, 1 TiDB, 0 PD, 0 TiFlash	Heap	Finished	Monday 11:05 PM (GMT+8)	10
0 TiKV, 1 TiDB, 0 PD, 0 TiFlash	CPU	Finished	Monday 11:05 PM (GMT+8)	30
1 TiKV, 0 TiDB, 0 PD, 0 TiFlash	CPU,Heap,Goroutine,Mutex	Finished	Last Friday 4:54 PM (GMT+8)	10
0 TiKV, 1 TiDB, 1 PD, 0 TiFlash	CPU,Heap,Goroutine,Mutex	Finished	Last Friday 4:54 PM (GMT+8)	10
1 TiKV, 1 TiDB, 1 PD, 1 TiFlash	CPU	Finished	Last Wednesday 10:53 AM (GMT+8)	30
1 TiKV, 1 TiDB, 1 PD, 1 TiFlash	CPU,Heap,Goroutine,Mutex	Finished	Last Tuesday 6:07 PM (GMT+8)	30
1 TiKV, 1 TiDB, 1 PD, 1 TiFlash	CPU,Heap,Goroutine,Mutex	Finished	Last Tuesday 4:57 PM (GMT+8)	30

图 604: 历史列表

关于状态详情页的操作，参见查看性能分析状态章节。

14.12.1.14.2 TiDB Dashboard 实例性能分析 - 持续分析页面

> **注意：**
>
> 该功能面向数据库技术专家，建议非专家用户在 PingCAP 专业技术人员的指导下使用该功能。

持续性能分析功能可以持续地收集 TiDB、TiKV、PD 各个实例的性能数据。收集到的性能数据可显示为火焰图或有向无环图形式。

通过这些性能数据，技术专家可以分析实例的 CPU、内存等资源消耗细节，帮助解决诸如某一时刻 CPU 开销较高、内存占用较大、进程不明原因卡死等复杂问题。即使这类问题无法得到复现，技术专家也可以通过查看当时记录下的历史性能数据，还原问题现场，对当时的问题进行细致分析，从而有效缩短平均故障恢复时间。

与手动性能分析的区别

持续性能分析是手动性能分析的强化功能，它们都能用于收集和分析各个实例不同维度的性能数据，主要区别如下：

- 手动性能分析仅在用户发起分析的时刻收集一小段时间（如 30 秒）的性能数据，持续性能分析开启后会持续收集性能数据。

- 手动性能分析只能用于分析集群当前的问题，持续性能分析既可以用于分析集群当前问题，也可以分析集群历史问题。
- 手动性能分析允许仅收集特定实例的特定性能数据，持续性能分析会收集所有实例的所有性能数据。
- 由于持续性能分析存储了更多性能数据，因此会占用更大的磁盘空间。

支持的性能数据

所有手动性能分析中的性能数据都会在该功能中收集：

- CPU：TiDB、TiKV、TiFlash、PD 实例上各个内部函数的 CPU 开销情况
- Heap：TiDB、PD 实例上各个内部函数的内存占用开销情况
- Mutex：TiDB、PD 实例上各个处于等待状态的 Mutex 情况
- Goroutine：TiDB、PD 实例上各个 Goroutine 的运行状态及调用栈情况

访问页面

你可以通过以下任一方式访问持续性能分析页面：

- 登录后，在左侧导航栏中点击高级调试 (Advanced Debugging) > 实例性能分析 (Profiling Instances) > 持续分析 (Continuous Profiling)。

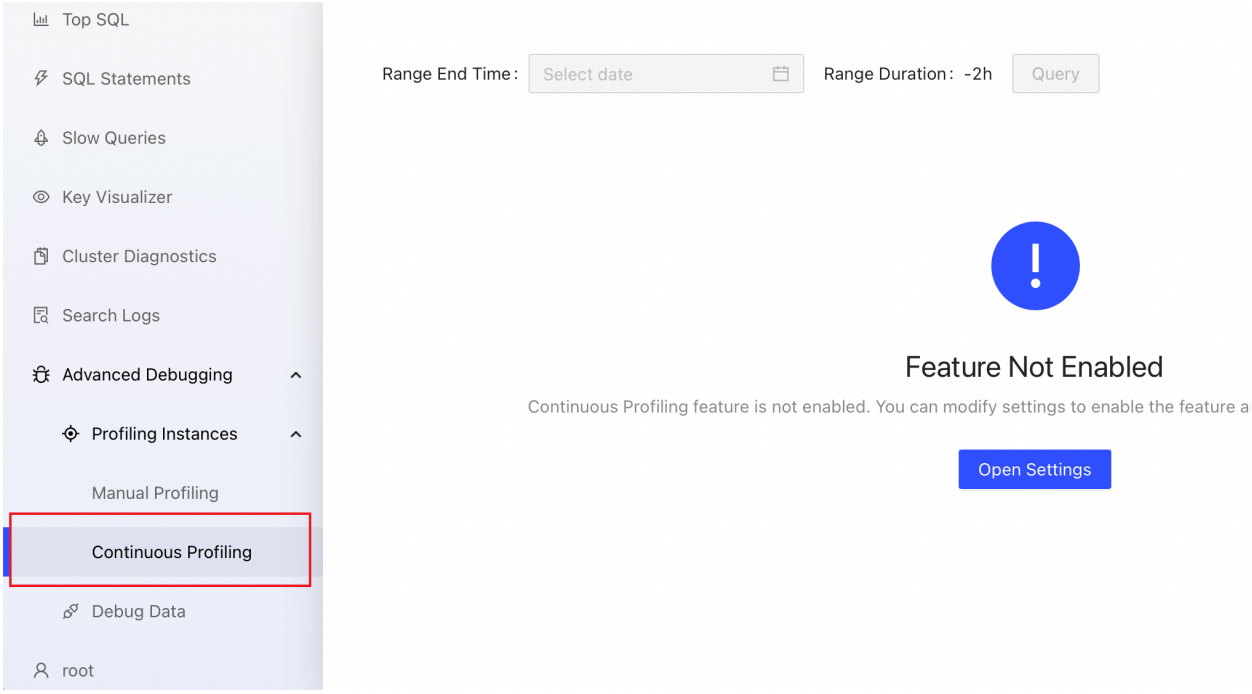


图 605: 访问页面

- 在浏览器中访问 http://127.0.0.1:2379/dashboard/#/continuous_profiling（将 `127.0.0.1:2379` 替换为实际 PD 实例地址和端口）。

启用持续性能分析

> 注意：
>
> 要使用持续性能分析，你需要使用 TiUP（v1.9.0 及以上版本）或 TiDB Operator（v1.3.0 及以上版本）部署或升级集群。如果你已经使用旧版本 TiUP 或 TiDB Operator 进行了集群升级，请参见FAQ 进行处理。

该功能启用后，TiDB Dashboard 会在后台持续收集性能数据，用户无需保持网页处于打开状态。后台收集的性能数据可设置保留时长，超过保留时长的性能数据将会被自动清理。

你可以通过以下步骤启用该功能：

1. 访问持续性能分析页面。
2. 点击打开设置 (Open Settings)。在右侧设置 (Settings) 页面，将启用特性 (Enable Feature) 下方的开关打开。设置保留时间 (Retention Period)，默认值为 3 天。
3. 点击保存 (Save)。

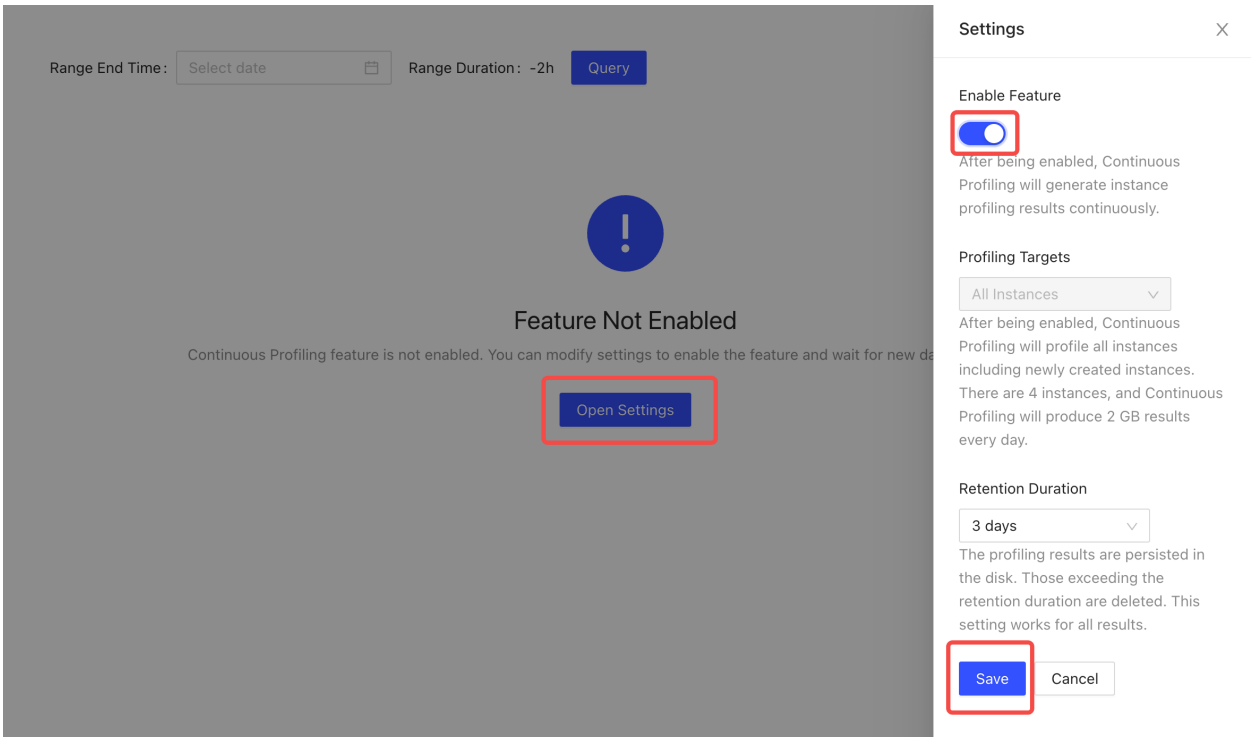


图 606: 启用功能

查看集群当前性能数据

在已经启用了持续性能分析的集群上，手动性能分析不再可用。若要查看当前时刻集群性能数据，请点击最近一次分析结果。

查看集群历史性能数据

你可以在列表中看到自启用该功能以来所有自动收集的性能数据：

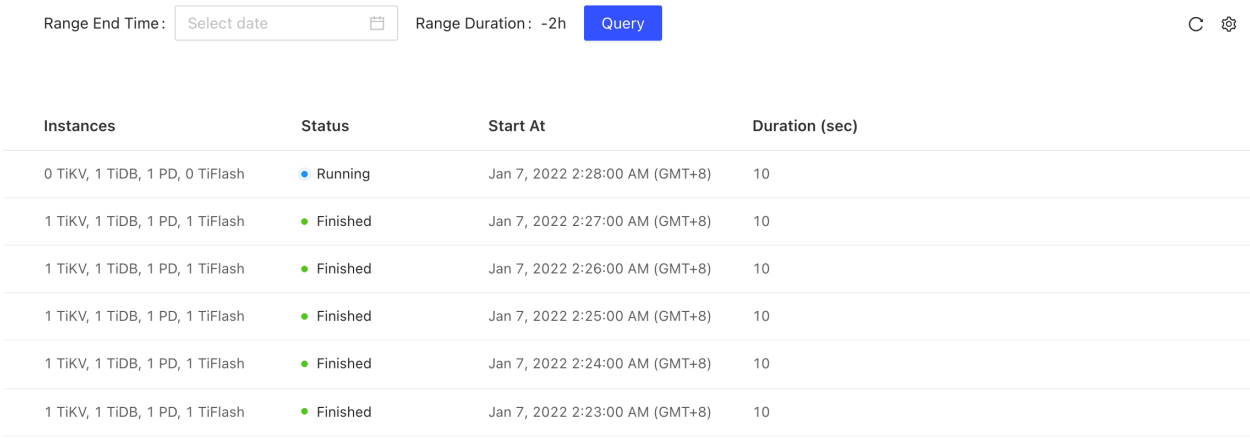


图 607: 历史结果

下载性能数据

进入某次分析结果后，可点击右上角下载按钮 (Download Profiling Result) 打包下载所有性能数据：

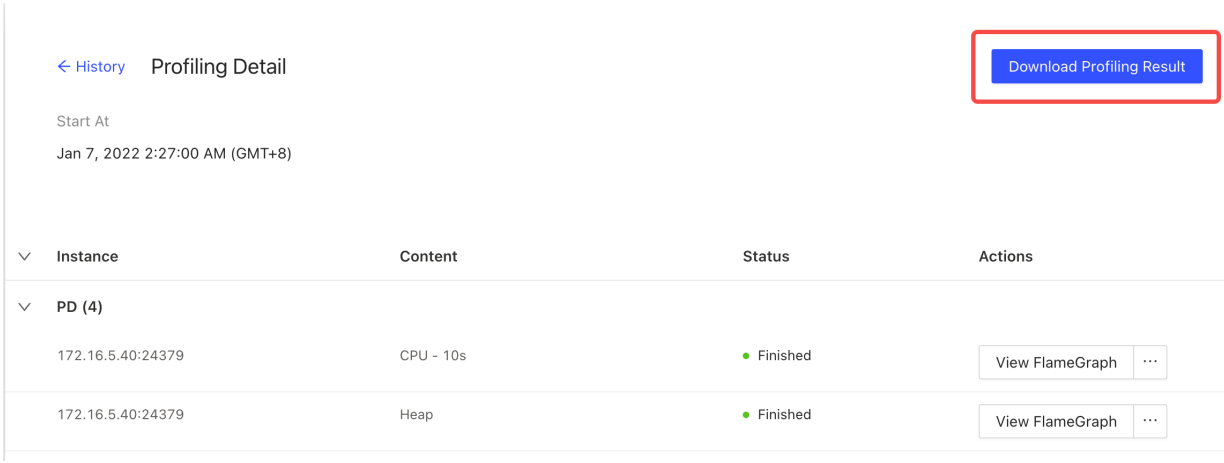


图 608: 下载某次分析结果

也可以点击列表中的单个实例查看其性能数据，或者悬浮在 ··· 按钮上下载原始数据：

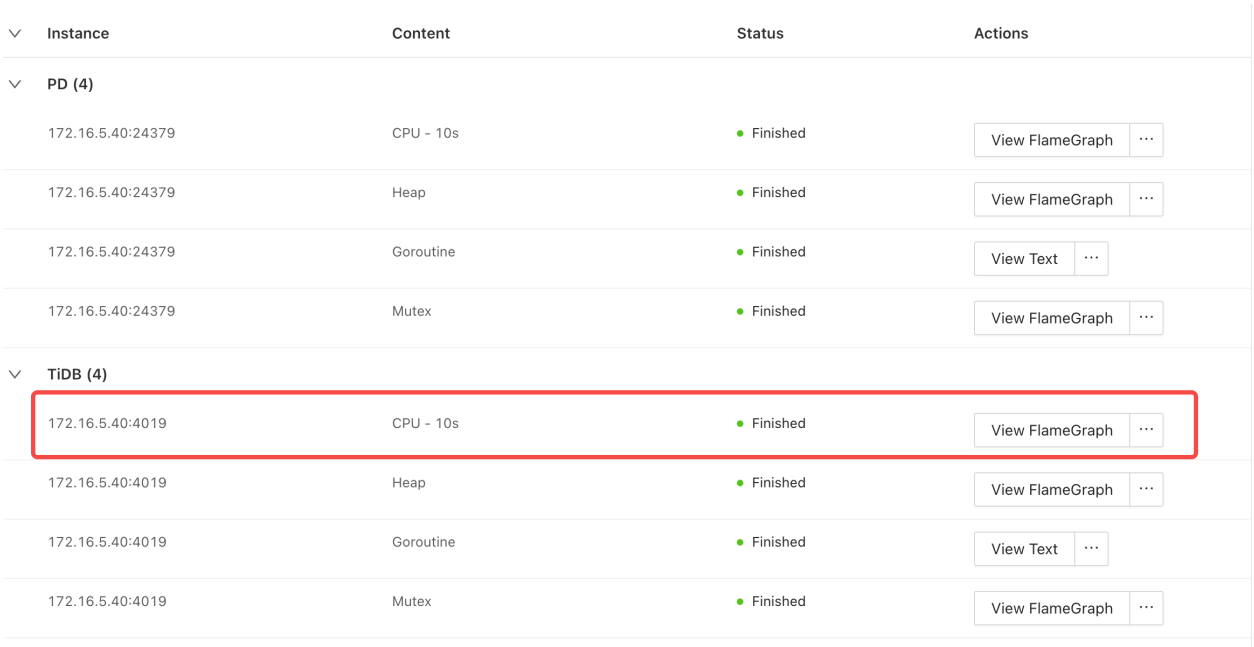


图 609: 查看单个实例分析结果

停用持续性能分析

你可以通过以下步骤停用该功能：

1. 访问持续性能分析页面。
2. 点击右上角齿轮按钮打开设置界面，将启用特性 (Enable Feature) 下方的开关关闭。
3. 点击保存 (Save)。
4. 在弹出的关闭持续分析功能 (Disable Continuous Profiling Feature) 对话框中，点击确认 (Disable)。

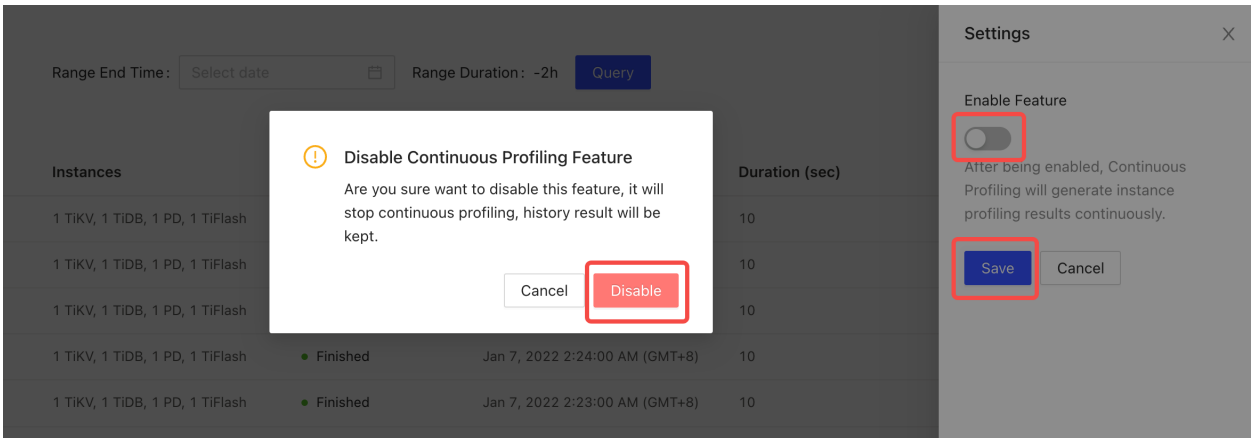


图 610: 停用功能

常见问题

1. 界面上提示“集群中未启动必要组件 NgMonitoring” 无法启用功能

请参见TiDB Dashboard FAQ。

2. 该功能开启后对集群是否有性能影响？

依据测试结果，该功能开启后对集群的平均性能影响小于 1%。

3. 该功能目前是什么状态？

该功能是正式特性，在生产环境中可用。

14.12.1.15 会话管理与配置

14.12.1.15.1 分享 TiDB Dashboard 会话

你可以将当前的 TiDB Dashboard 会话安全地分享给其他用户访问，这样其他用户无需要知道登录账号密码即可访问 TiDB Dashboard 并进行操作。

分享者操作步骤

1. 登录 TiDB Dashboard。
2. 点击边栏左下角的用户名访问配置界面。
3. 点击分享当前会话 (Share Current Session)。

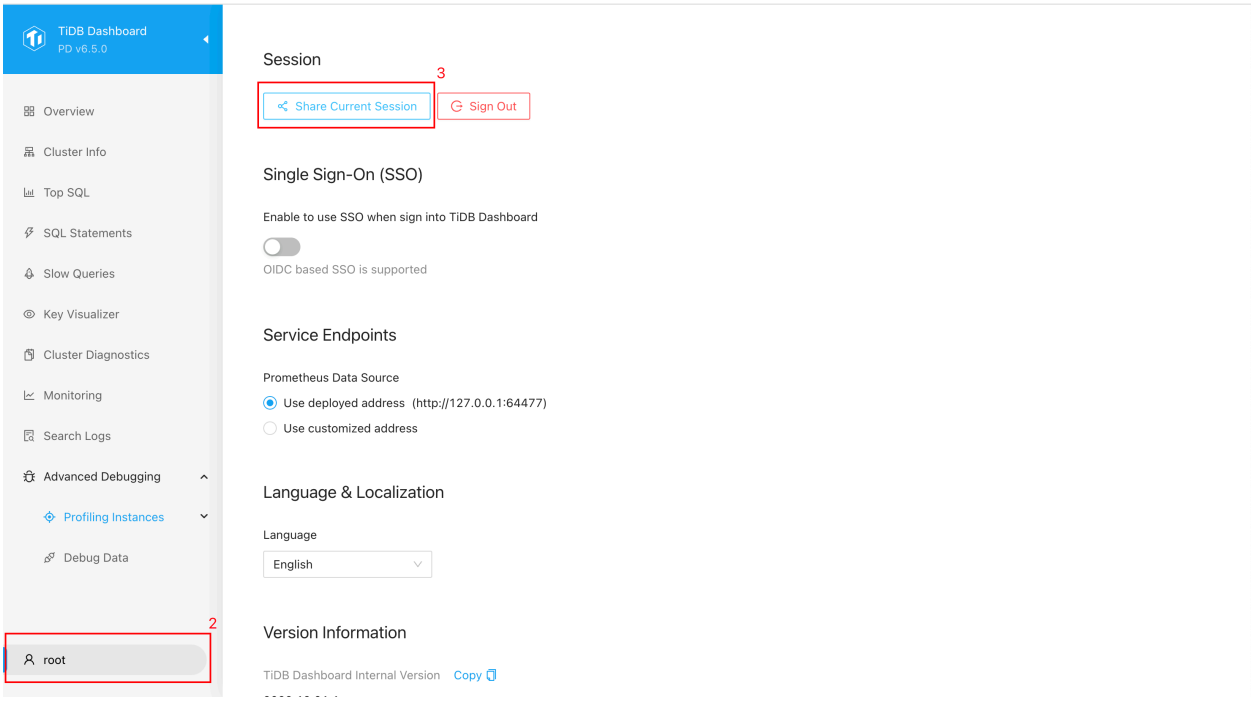


图 611: 操作示例

> 注意：
> 出于安全考虑，已分享的会话中不能使用分享功能将该会话再次分享给其他人。

4. 在弹出的对话框中，对分享进行细节配置：

- 有效时间：分享的会话在多少时间内有效。登出当前会话不影响已分享会话的有效时间。
- 以只读权限分享：分享的会话为只读，例如不允许进行配置修改等操作。

5. 点击生成授权码 (Generate Authorization Code)。

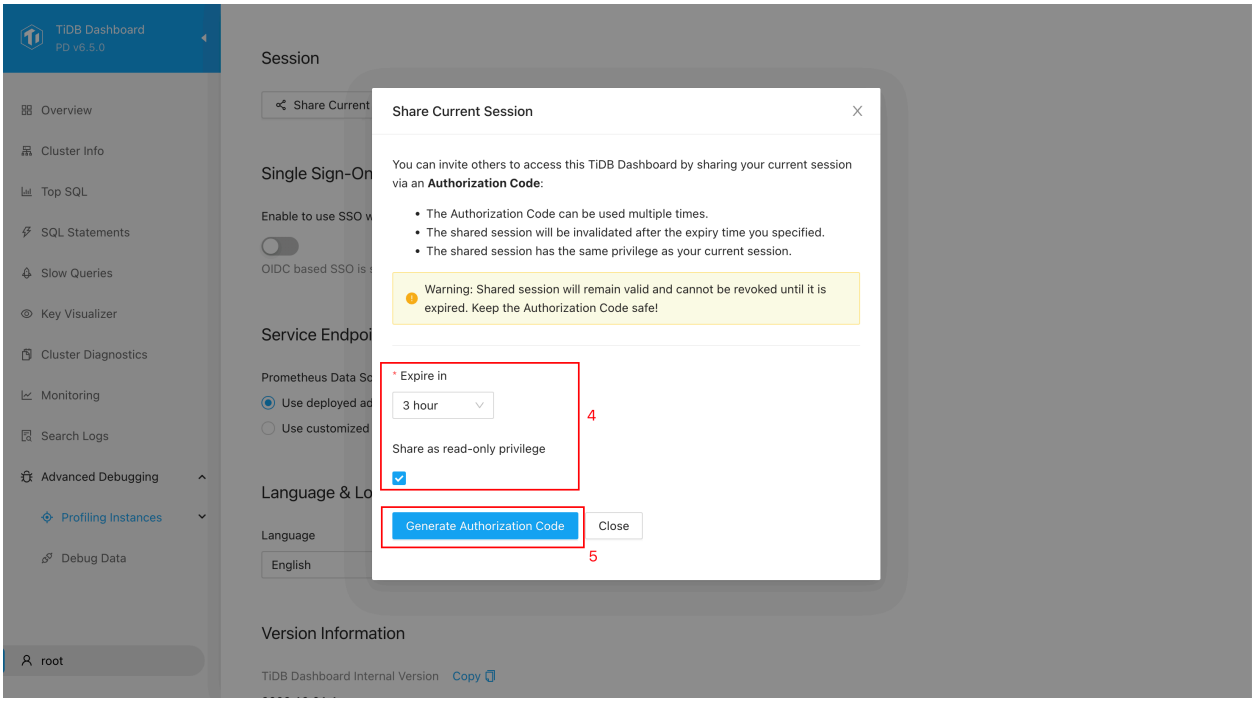


图 612: 操作示例

6. 将生成出来的授权码提供给要分享的用户。

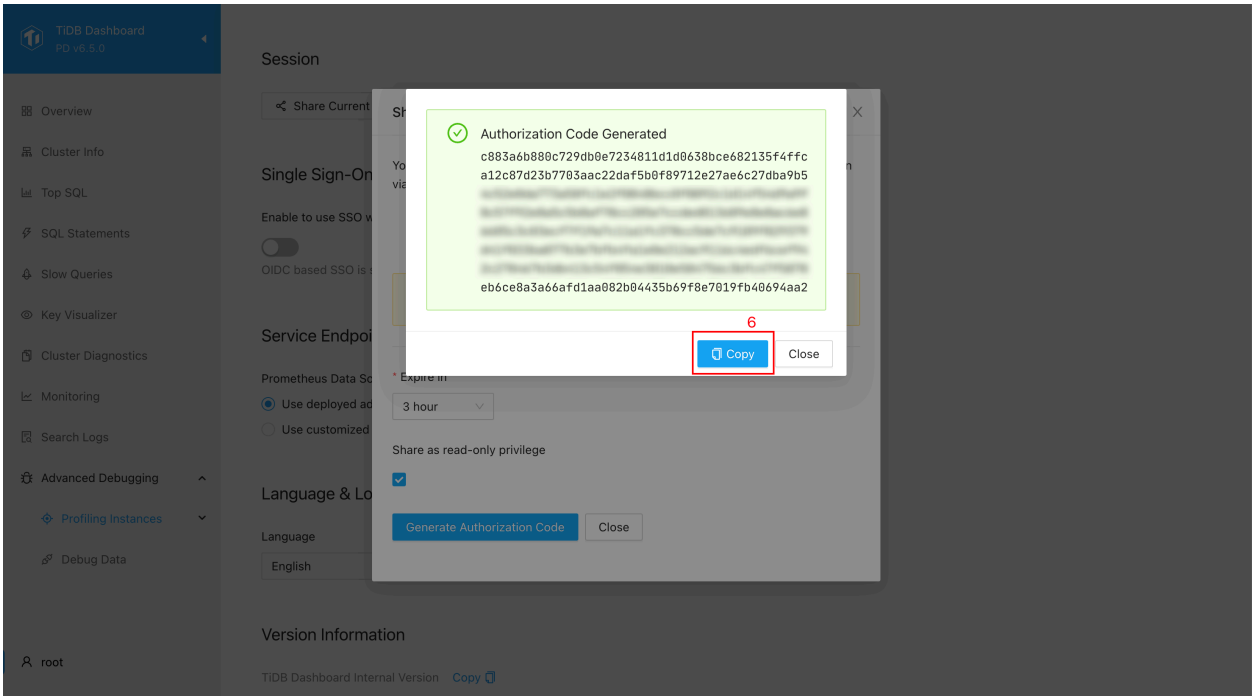


图 613: 操作示例

> **警告：**
>
> 请妥善地保管授权码。不要将授权码分发给不受信任的用户，否则他们也将具备访问和操作 TiDB Dashboard 的能力。

受邀请者操作步骤

1. 在 TiDB Dashboard 登录界面上，点击使用其他登录方式 (Use Alternative Authentication)。

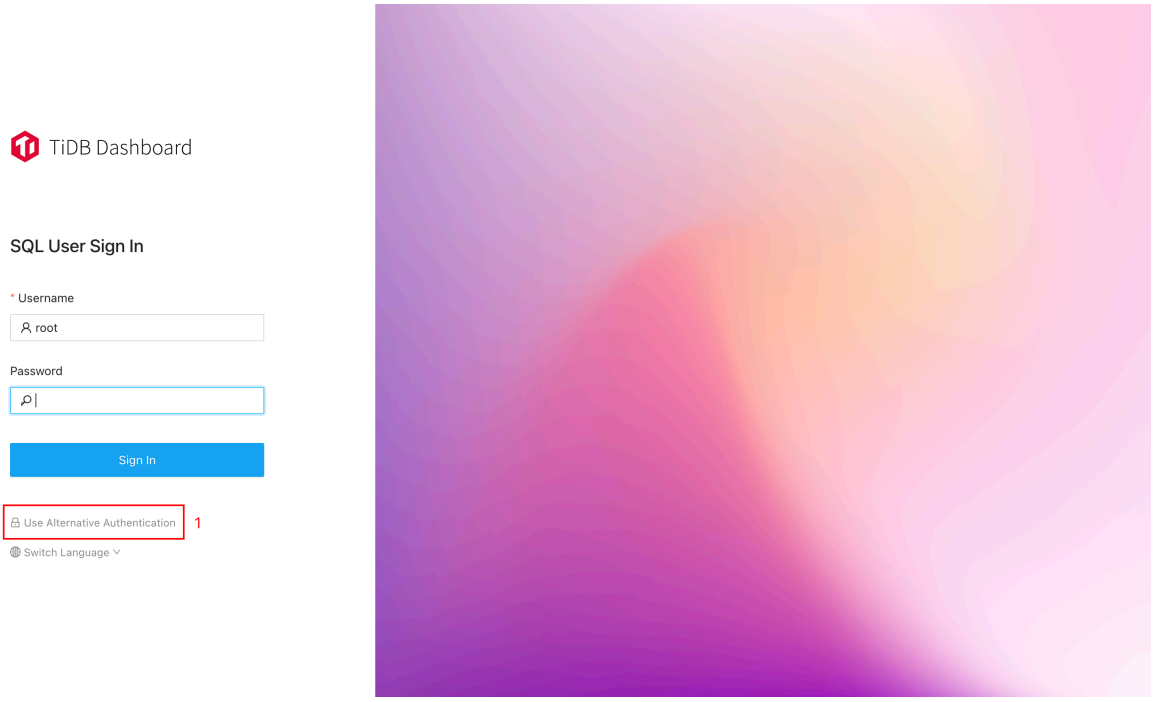


图 614: 操作示例

2. 选择使用授权码 (Authorization Code) 登录。

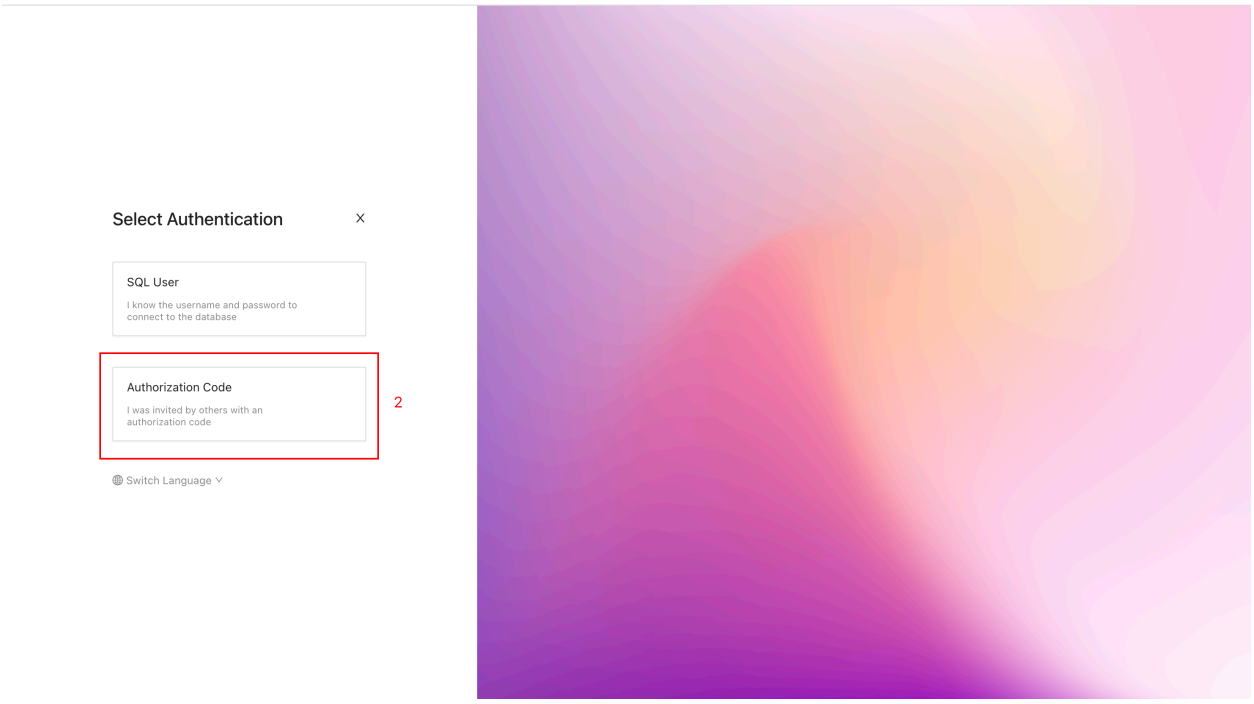


图 615: 操作示例

3. 输入从分享者取得的授权码。
4. 点击登录 (Sign In)。

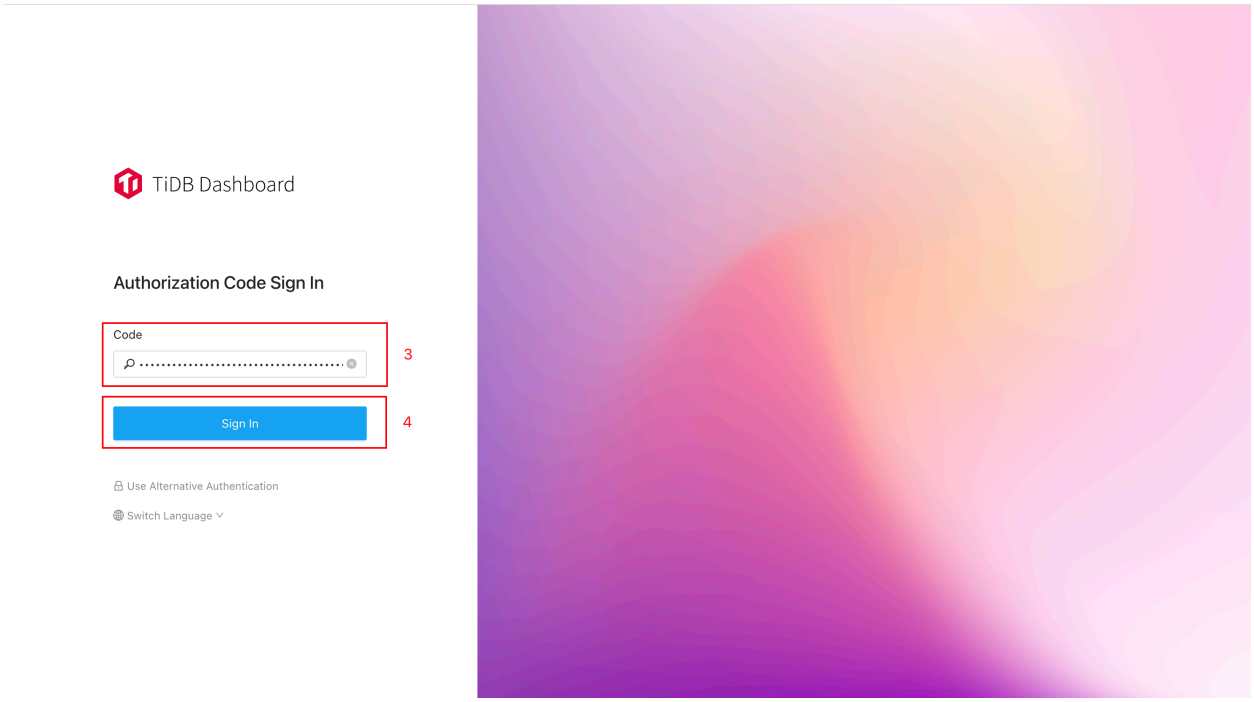


图 616: 操作示例

14.12.1.15.2 配置 TiDB Dashboard 使用 SSO 登录

TiDB Dashboard 支持基于 OIDC 协议的单点登录 (Single Sign-On)。配置 TiDB Dashboard 启用 SSO 登录后，你可以通过配置的 SSO 服务进行登录鉴权，无需输入 SQL 用户名和密码即可登录到 TiDB Dashboard。

配置 OIDC SSO

启用 SSO

1. 登录 TiDB Dashboard。
2. 点击边栏左下角的用户名访问配置界面。
3. 在单点登录 (Single Sign-On) 区域下，开启允许使用 SSO 登录到 TiDB Dashboard (Enable to use SSO when sign into TiDB Dashboard)。
4. 在表单中填写 OIDC Client ID 和 OIDC Discovery URL 字段。

   一般可以从 SSO 服务的提供商处获取到这两个字段信息：

   - OIDC Client ID 有时也被称为 OIDC Token Issuer
   - OIDC Discovery URL 有时也被称为 OIDC Token Audience。

5. 将 SQL 登录密码录入到 TiDB Dashboard 中，以便在 SSO 鉴权通过后完成登录。点击授权登录为该用户 (Authorize Impersonation) 录入密码。

   这是因为 TiDB Dashboard SSO 的原理是在 SSO 成功鉴权后，采用 TiDB Dashboard 内加密存储的 SQL 登录密码进行替代登录。

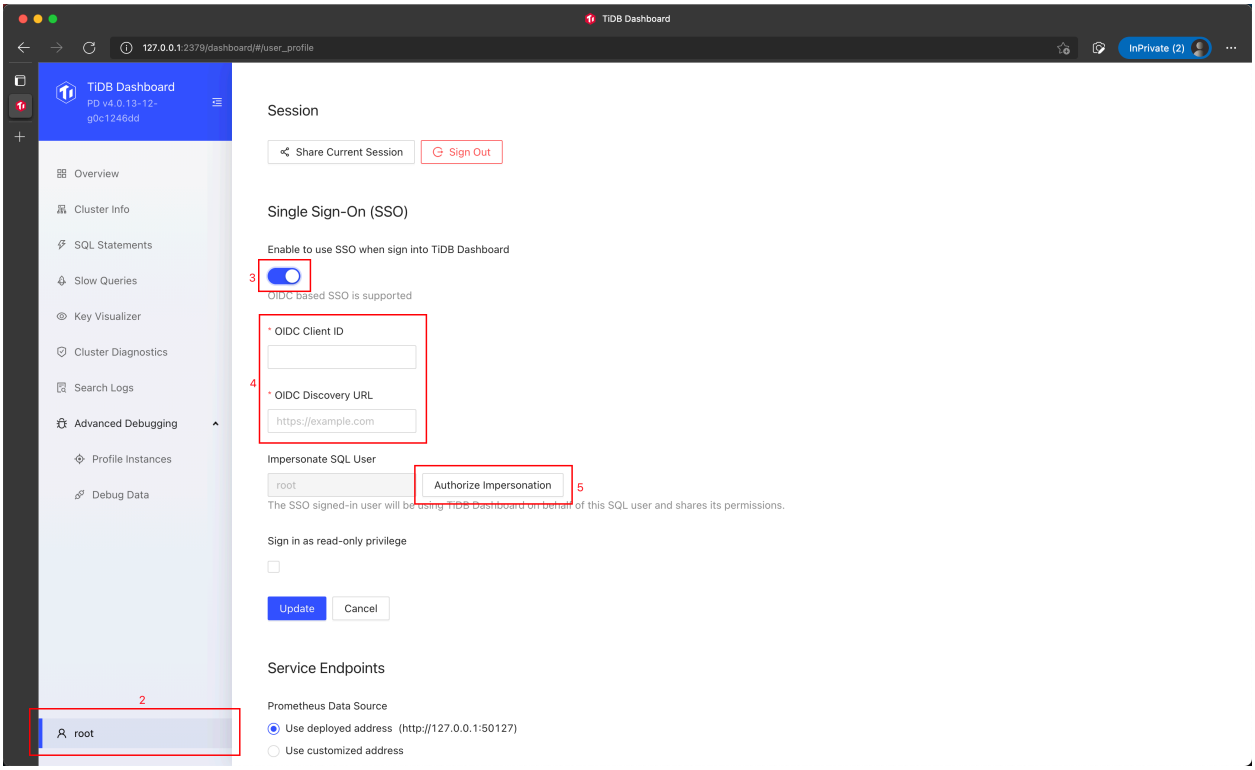


图 617: 操作示例

> **注意：**
>
> 你录入的密码将被加密存储。若 SQL 用户密码后续发生了变更，将导致 SSO 登录失败。这时可以重新录入密码使 SSO 登录恢复正常。

6. 在对话框中填写完密码后，点击授权并保存 (Authorize and Save)。

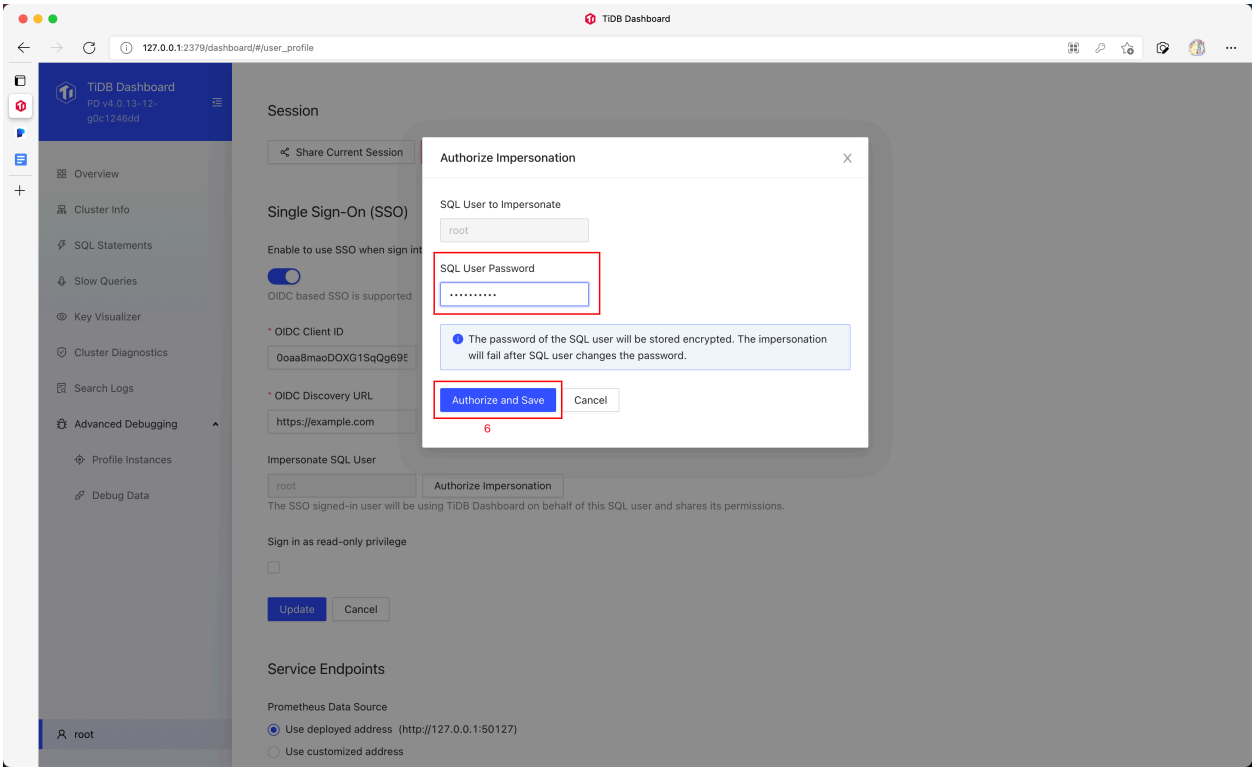


图 618: 操作示例

7. 点击更新 (Update) 保存配置。

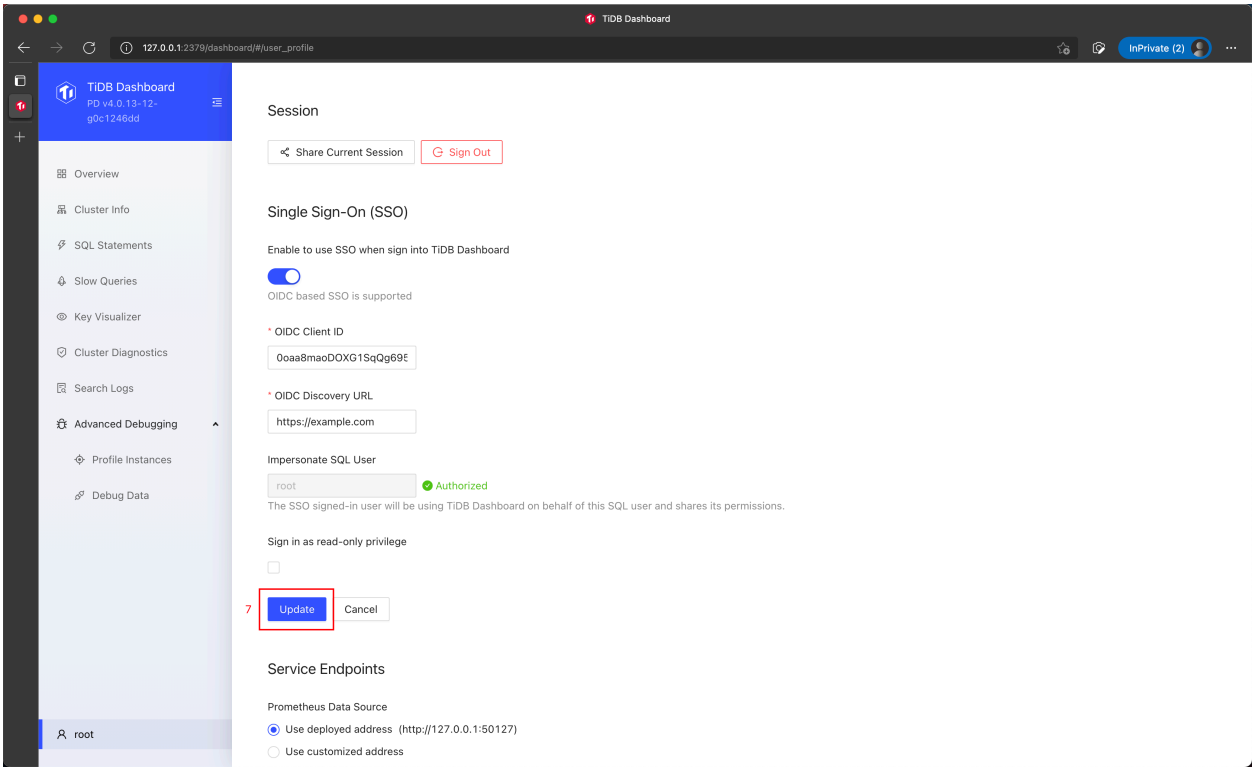


图 619: 操作示例

至此 TiDB Dashboard 中已经成功开启了 SSO 登录。

> **注意：**
>
> 出于安全原因，部分 SSO 服务还需要你进一步在 SSO 服务中配置受信任的登录和登出跳转地址，请参见 SSO 服务的具体帮助完成配置。

禁用 SSO

你可以随时禁用 SSO。禁用后，之前已录入并存储在本地的替代登录 SQL 密码将被彻底清除。禁用步骤如下：

1. 登录 TiDB Dashboard。
2. 点击边栏左下角用户名访问配置界面。
3. 在单点登录 (Single Sign-On) 区域下，关闭允许使用 SSO 登录到 TiDB Dashboard (Enable to use SSO when sign into TiDB Dashboard)。
4. 点击更新 (Update) 保存配置。

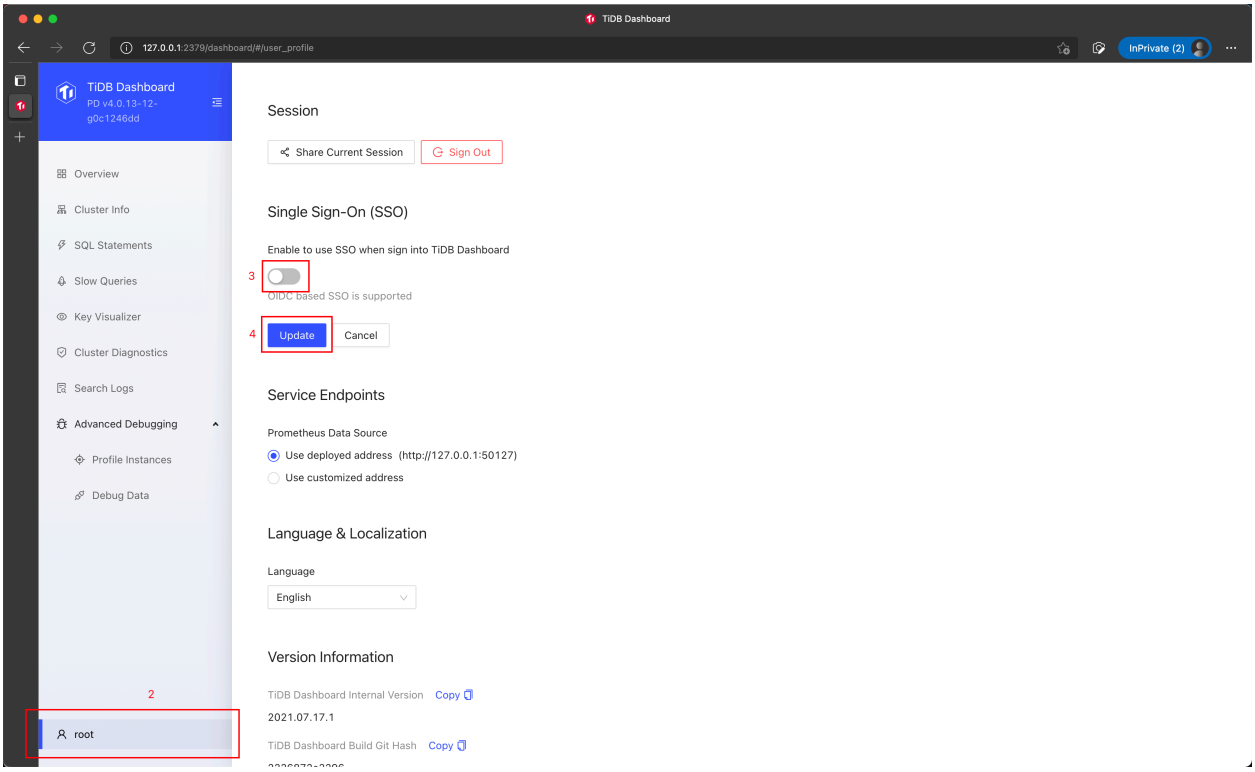


图 620: 操作示例

密码发生变更后重新录入密码

若替代登录的 SQL 用户密码发生了变更，则 SSO 登录将会失败。此时，你可以将新的登录密码录入到 TiDB Dashboard 中以恢复正常 SSO 登录功能，步骤如下：

1. 登录 TiDB Dashboard。
2. 点击边栏左下角用户名访问配置界面。
3. 在单点登录 (Single Sign-On) 区域下，点击授权登录为该用户 (Authorize Impersonation) 来录入新的密码。

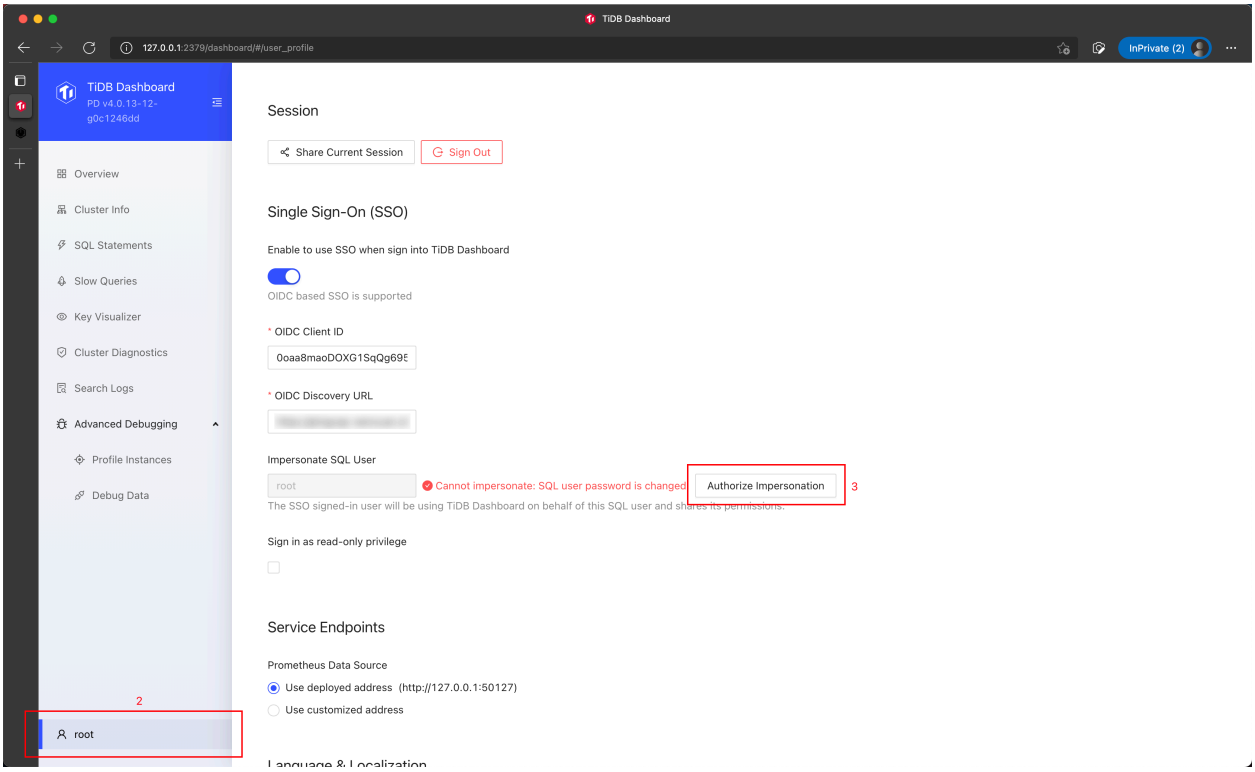


图 621: 操作示例

4. 在对话框中填写完毕密码后，点击授权并保存 (Authorize and Save)。

使用 SSO 登录

若 TiDB Dashboard 已经完成了 SSO 的配置，你可使用以下步骤完成登录：

1. 在 TiDB Dashboard 登录界面上，点击使用公司账号 SSO 登录 (Sign in via Company Account)。

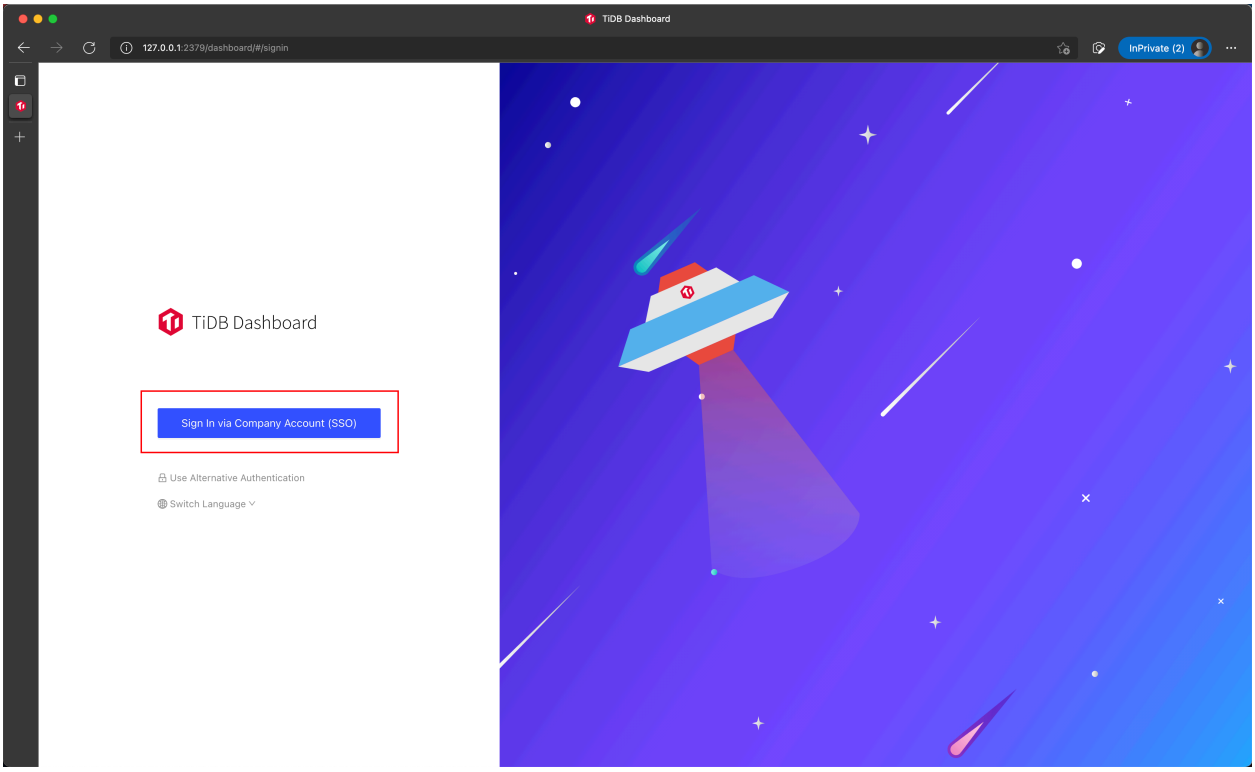


图 622: 操作示例

2. 在配置 SSO 的系统中进行登录。
3. 你将被重定向回 TiDB Dashboard 完成登录。

示例一：使用 Okta 进行 TiDB Dashboard SSO 登录认证

Okta 是一个提供 OIDC SSO 的身份认证服务。以下步骤展示了如何配置 Okta 及 TiDB Dashboard，使得 TiDB Dashboard 可以通过 Okta 进行 SSO 登录。

步骤一：配置 Okta

首先需要在 Okta 中创建一个用于集成 SSO 的 Application Integration。

1. 访问 Okta 管理后台。
2. 点击左侧边栏的 Applications > Applications。
3. 点击 Create App Integration。

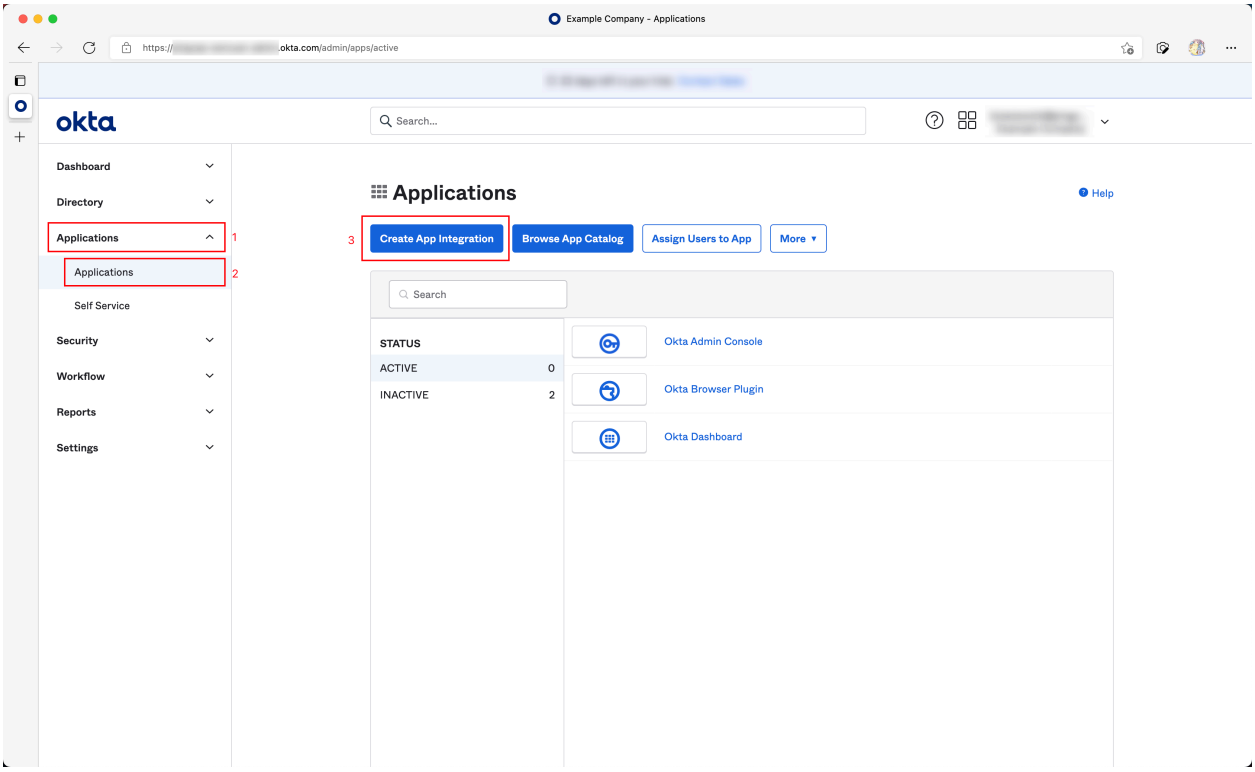


图 623: 操作示例

4. 在弹出的对话框中，Sign-in method 字段选择 OIDC - OpenID Connect。
5. Application Type 字段选择 Single-Page Application。
6. 对话框中点击 Next 按钮。

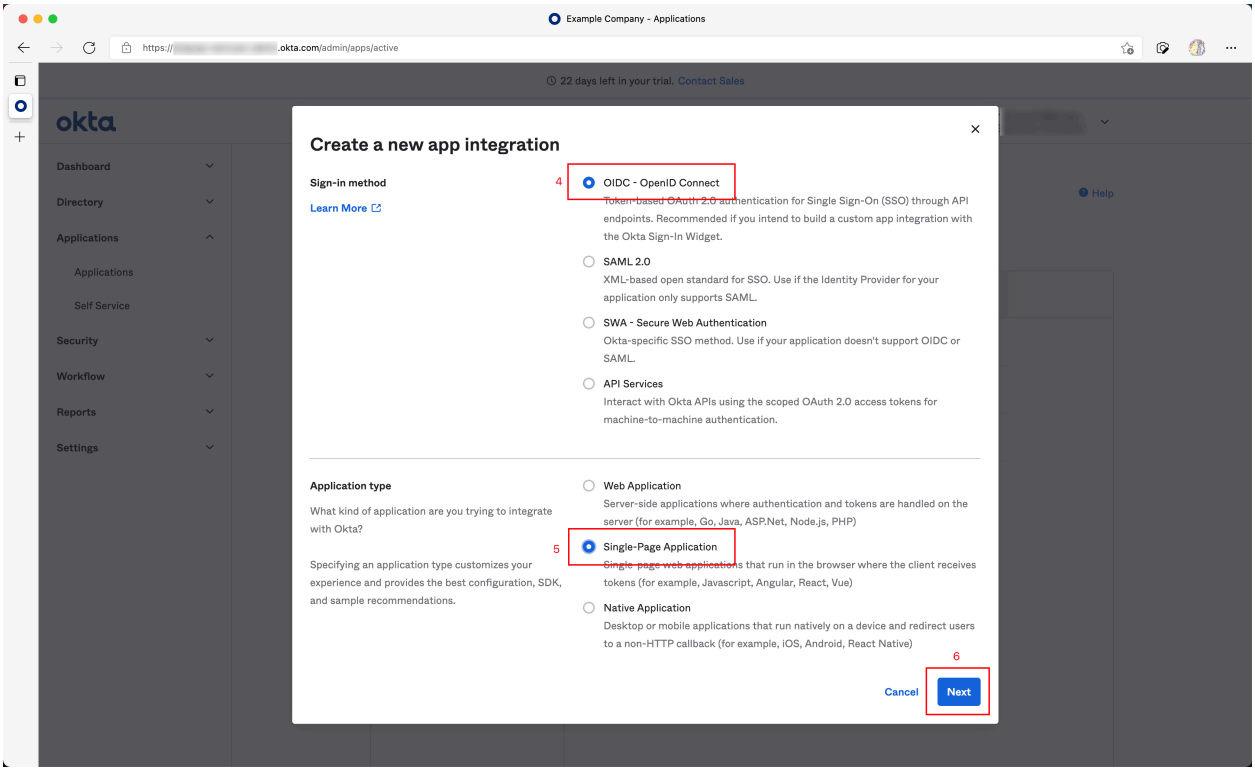


图 624: 操作示例

7. Sign-in redirect URIs 字段填写如下内容：

```
http://DASHBOARD_IP:PORT/dashboard/?sso_callback=1
```

以上内容中，将 DASHBOARD_IP:PORT 替换为你在浏览器中实际访问 TiDB Dashboard 所使用的域名（或 IP）及端口。

8. Sign-out redirect URIs 字段填写如下内容：

```
http://DASHBOARD_IP:PORT/dashboard/
```

类似地，将 DASHBOARD_IP:PORT 替换为实际的域名（或 IP）及端口。

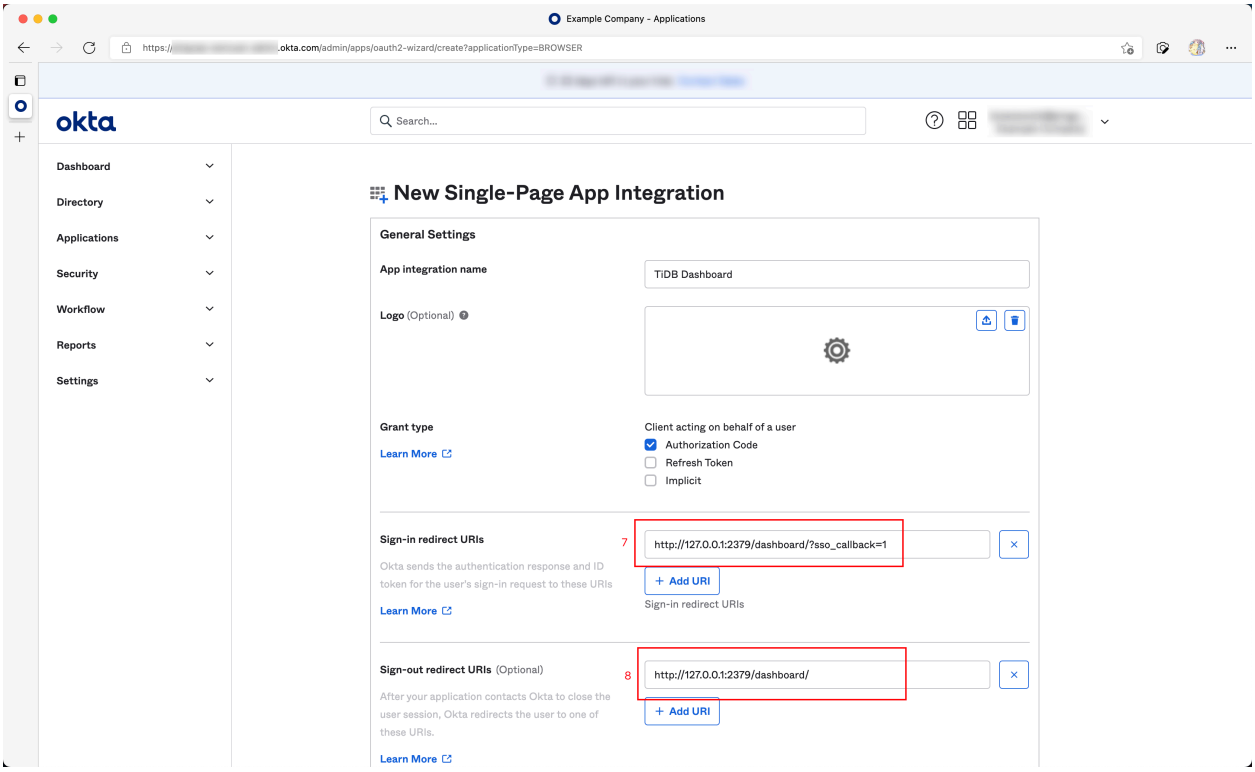


图 625: 操作示例

9. 在 Assignments 中按你的实际需求配置组织中哪些用户可以通过这个 SSO 登录 TiDB Dashboard，然后点击 Save 保存配置。

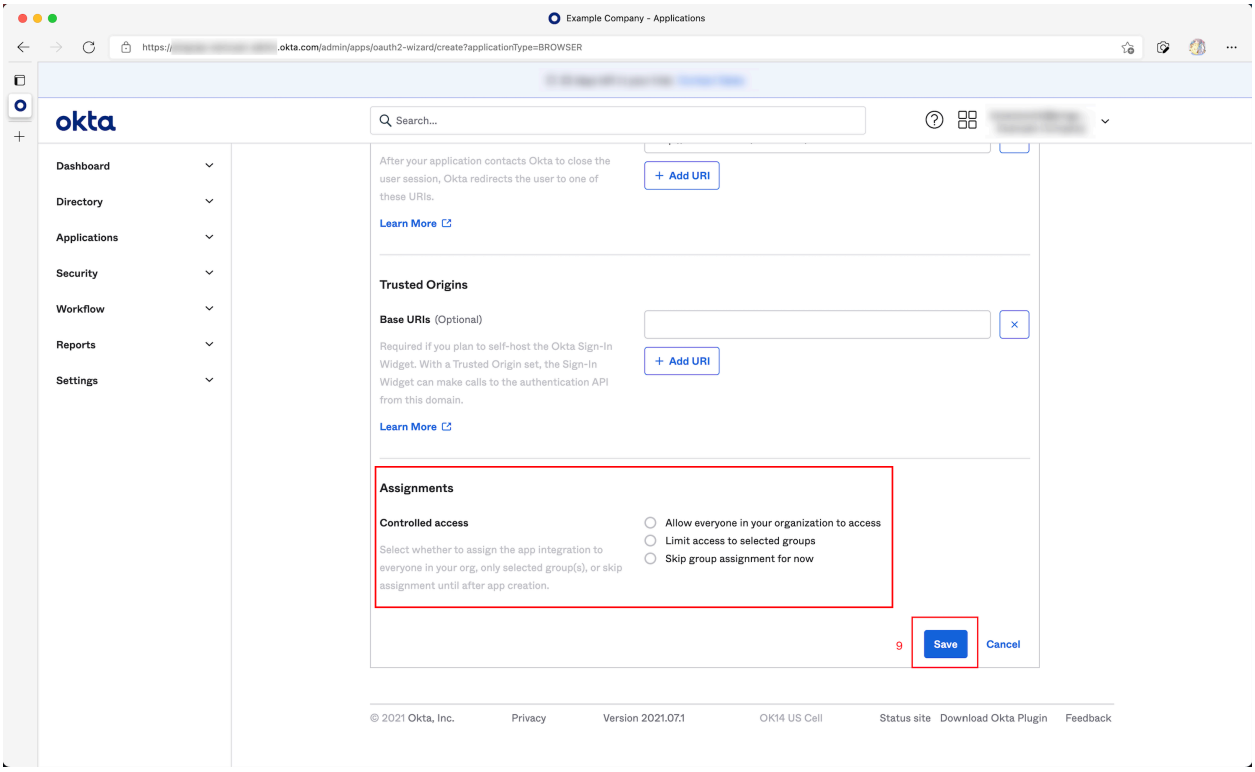


图 626: 操作示例

步骤二：获取 TiDB Dashboard 所需的配置参数并填入 TiDB Dashboard

1. 在 Okta 创建的 App Integration 中，点击 Sign On。

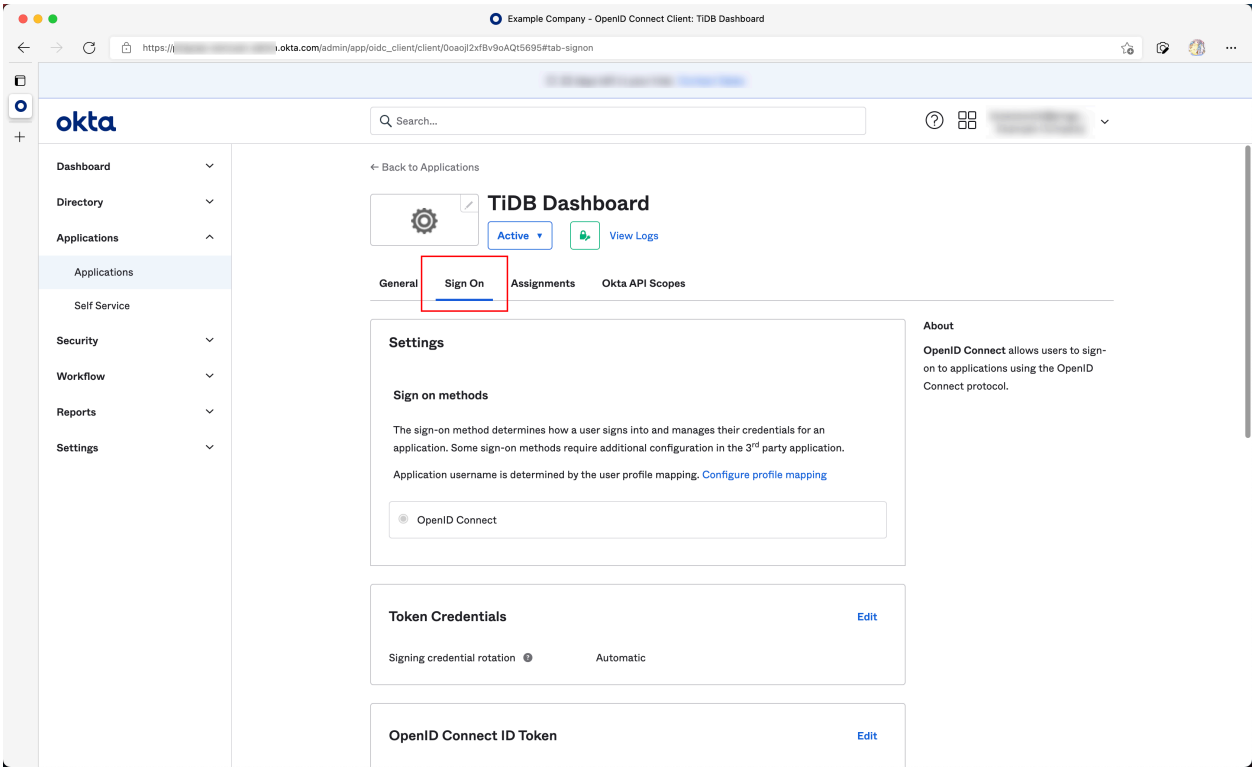


图 627: 操作示例 1

2. OpenID Connect ID Token 区域中有 Issuer 和 Audience 字段，复制这两个字段的值。

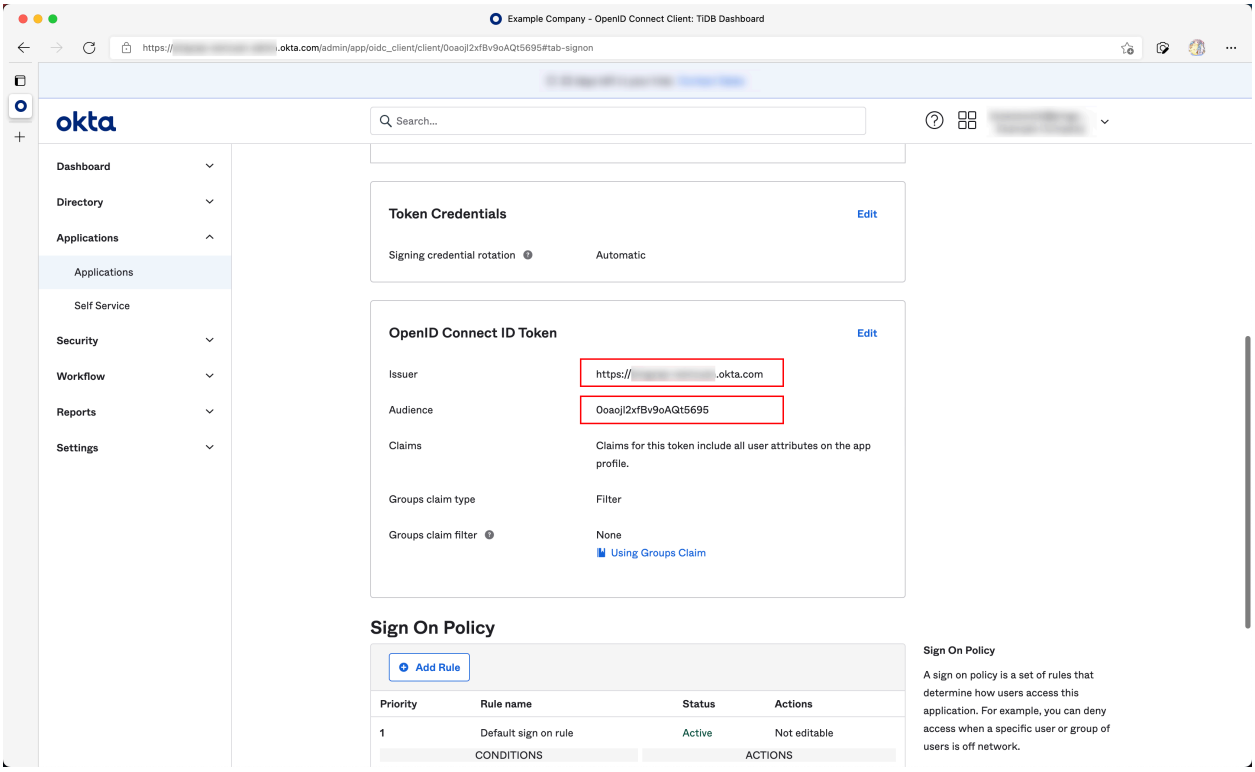


图 628: 操作示例 2

3. 打开 TiDB Dashboard 配置界面，将上一步获取到的 Issuer 填入 OIDC Client ID，将 Audience 填入 OIDC Discovery URL 后，完成授权并保存配置。示例如下：

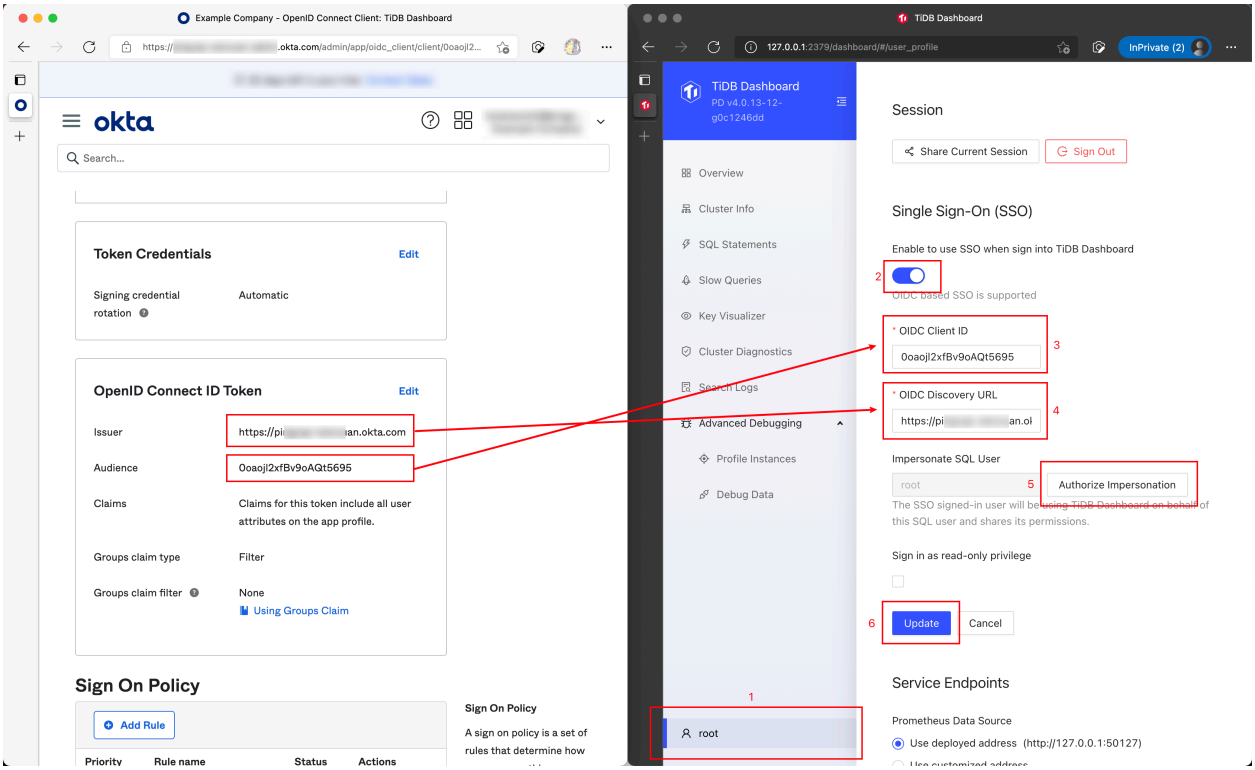


图 629: 操作示例 3

至此，TiDB Dashboard 已被配置为使用 Okta 进行 SSO 登录。

示例二：使用 Auth0 进行 TiDB Dashboard SSO 登录认证

和 Okta 类似，Auth0 也可以提供 OIDC SSO 的身份认证服务。

步骤一：配置 Auth0

1. 访问 Auth0 的管理后台。
2. 点击左侧边栏的 Applications > Applications。
3. 点击 Create Application，在弹出窗口中输入 Name，例如 “TiDB Dashboard”。在 Choose an application type 下选择 “Single Page Web Application”。

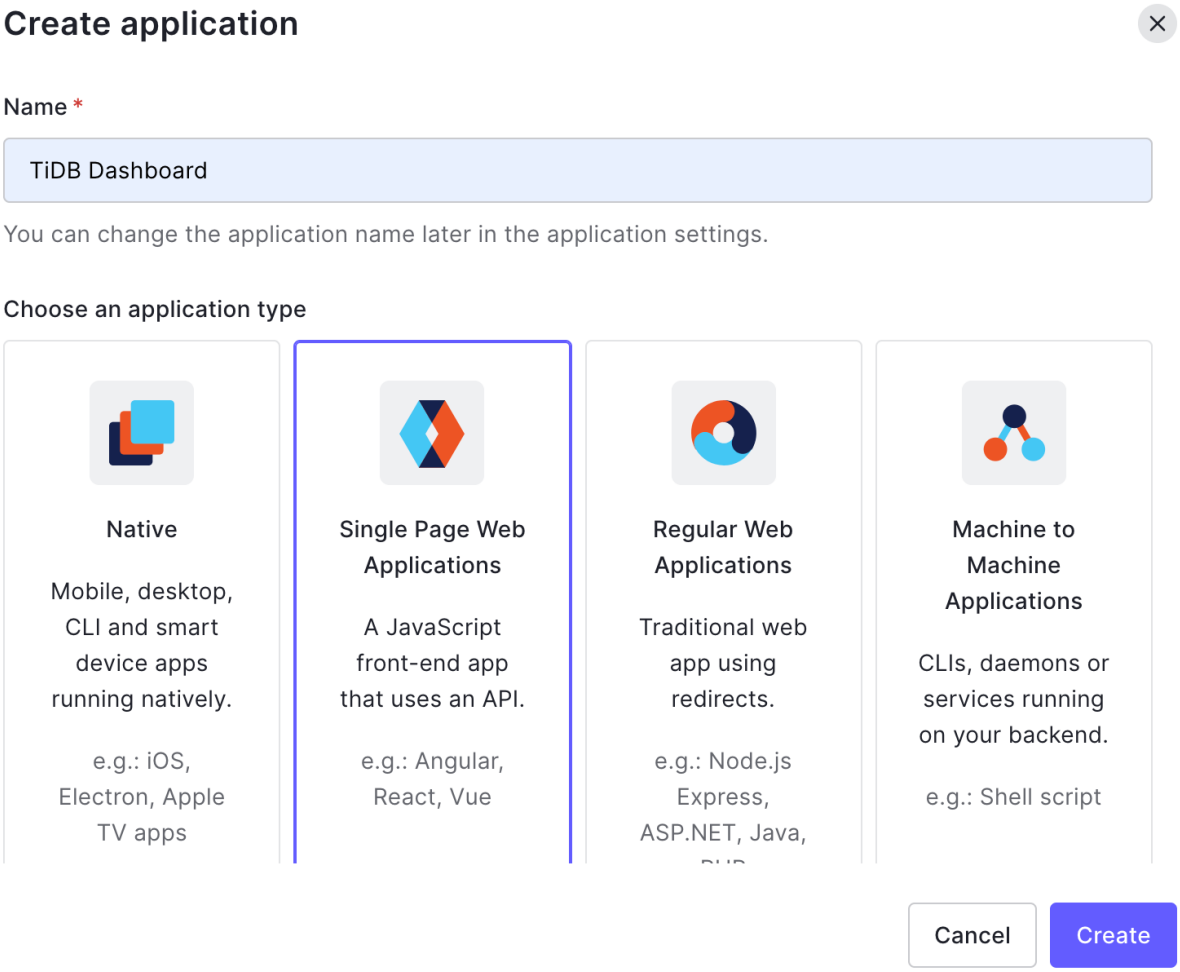


图 630: Create Application

4. 点击 Settings 栏。

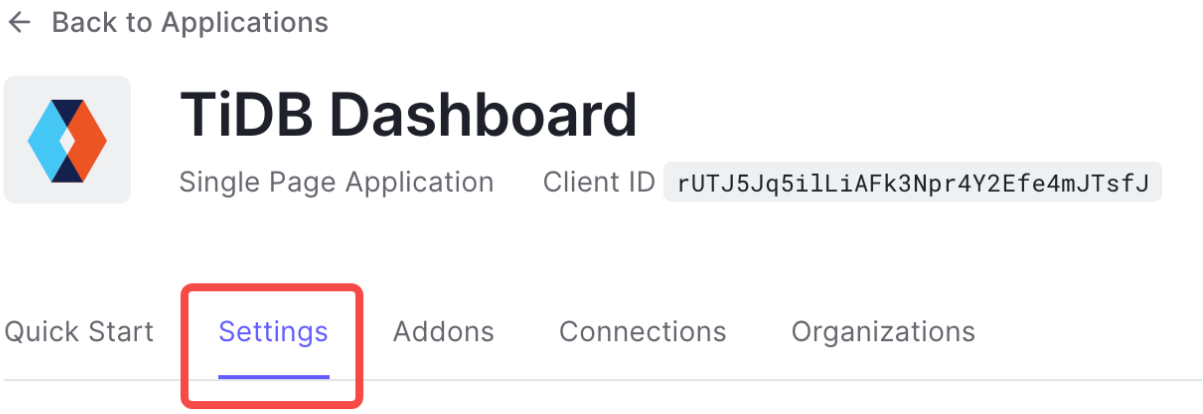


图 631: Settings

5. 在 Allowed Callback URLs 字段中填写如下内容：

```
http://DASHBOARD_IP:PORT/dashboard/?sso_callback=1
```

在以上内容中，将 DASHBOARD_IP:PORT 替换为你在浏览器中实际访问 TiDB Dashboard 所使用的域名（或 IP）及端口。

6. 在 Allowed Logout URLs 字段中填写如下内容：

```
http://DASHBOARD_IP:PORT/dashboard/
```

类似地，将 DASHBOARD_IP:PORT 替换为实际的域名（或 IP）及端口。

Allowed Callback URLs

http://localhost:3001/dashboard/?sso_callback=1,
http://127.0.0.1:2379/dashboard/?sso_callback

After the user authenticates we will only call back to any of these URLs. You can specify multiple valid URLs by comma-separating them (typically to handle different environments like QA or testing). Make sure to specify the protocol (`https://`) otherwise the callback may fail in some cases. With the exception of custom URI schemes for native clients, all callbacks should use protocol `https://`. You can use Organization URL parameters in these URLs.

Allowed Logout URLs

http://localhost:3001/dashboard/,
http://127.0.0.1:2379/dashboard/

A set of URLs that are valid to redirect to after logout from Auth0. After a user logs out from Auth0 you can redirect them with the `returnTo` query parameter. The URL that you use in `returnTo` must be listed here. You can specify multiple valid URLs by comma-separating them. You can use the star symbol as a wildcard for subdomains (`*.google.com`). Query strings and hash information are not taken into account when validating these URLs. Read more about this at https://auth0.com/docs/login/logout

图 632: Settings

7. 其它设置保持默认，点击 Save Changes 保存。

步骤二：获取 TiDB Dashboard 所需的配置参数并填入 TiDB Dashboard

1. 将 Auth0 Settings 栏 Basic Information 项的 Client ID 字段的值填入 TiDB Dashboard 的 OIDC Client ID，将 Domain 字段的值，加上 `https://` 前缀和 `/` 后缀后填入 OIDC Discovery URL 中，例如 `https://example.us.auth0.com/`。完成授权并保存配置即可。

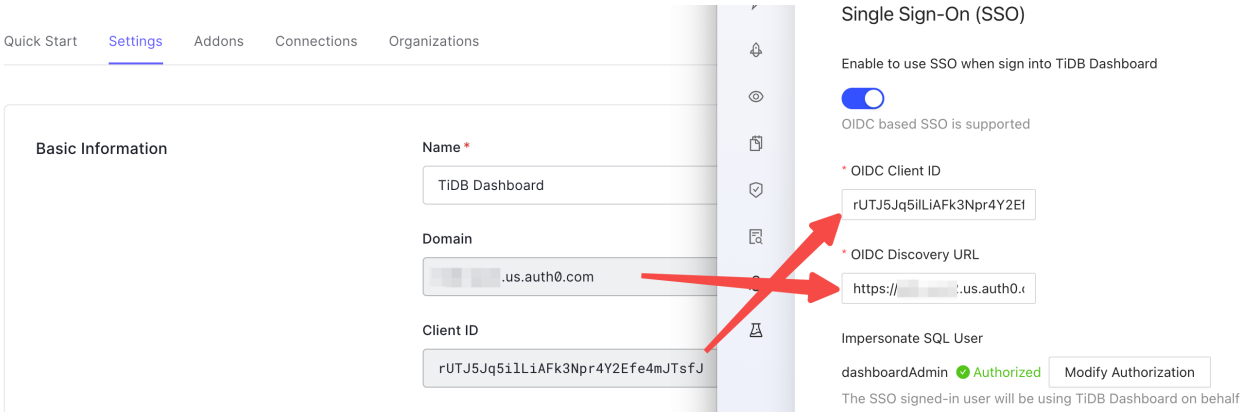


图 633: Settings

至此，TiDB Dashboard 已被配置为使用 Auth0 进行 SSO 登录。

示例三：使用 Casdoor 进行 TiDB Dashboard SSO 登录认证

Casdoor 是一个开源的、可以部署在私人服务器上的 SSO 平台。它与 TiDB Dashboard 的 SSO 功能兼容。以下步骤展示了如何配置 Casdoor 及 TiDB Dashboard，使得 TiDB Dashboard 可以通过 Casdoor 进行 SSO 登录。

步骤一：配置 Casdoor

1. 部署并访问 Casdoor 的管理后台。
2. 点击上方菜单栏的应用。
3. 点击添加按钮。

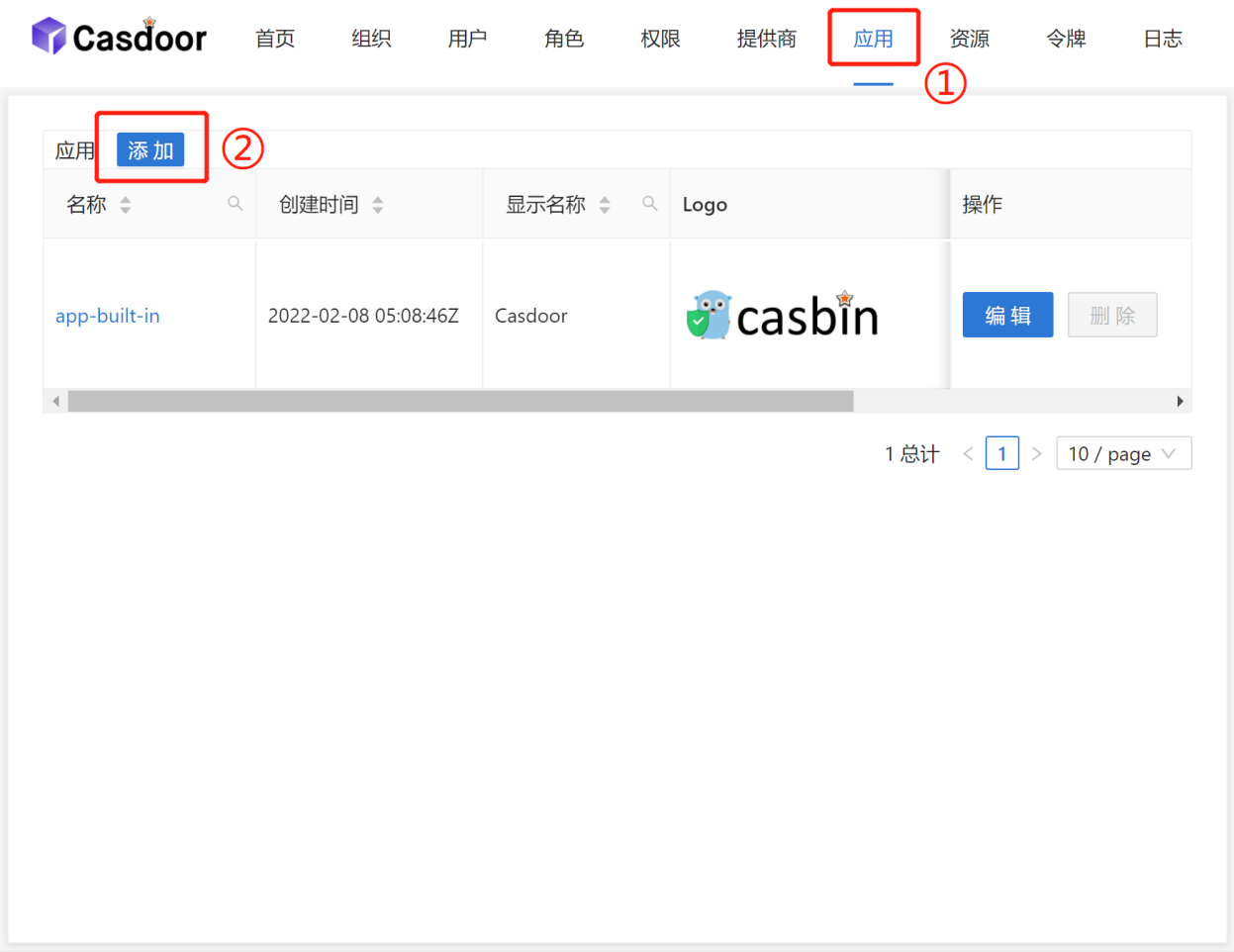


图 634: Settings

4. 填写名称和显示名称，比如：TiDB Dashboard。
5. 在回调 URLs 中添加如下内容：

```
http://DASHBOARD_IP:PORT/dashboard/?sso_callback=1
```

在以上内容中，将 DASHBOARD_IP:PORT 替换为你在浏览器中实际访问 TiDB Dashboard 所使用的域名（或 IP）及端口。

回调URLs ⓘ：

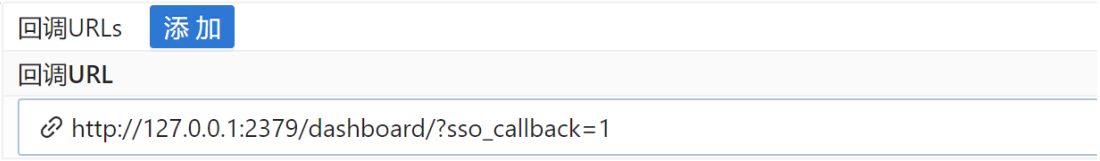


图 635: Settings

6. 其他设置保留默认值，点击保存 & 退出。
7. 保存页面上的客户端 ID。

步骤二：获取 TiDB Dashboard 所需的配置参数并填入 TiDB Dashboard

1. 将上一步获得的客户端 ID 字段的值填入 TiDB Dashboard 的 OIDC Client ID。
2. 将 Casdoor 部署地址加上 `https://` 前缀和 `/` 后缀后填入 OIDC Discovery URL 中，例如 `https://casdoor.example.com/`。完成授权并保存配置即可。

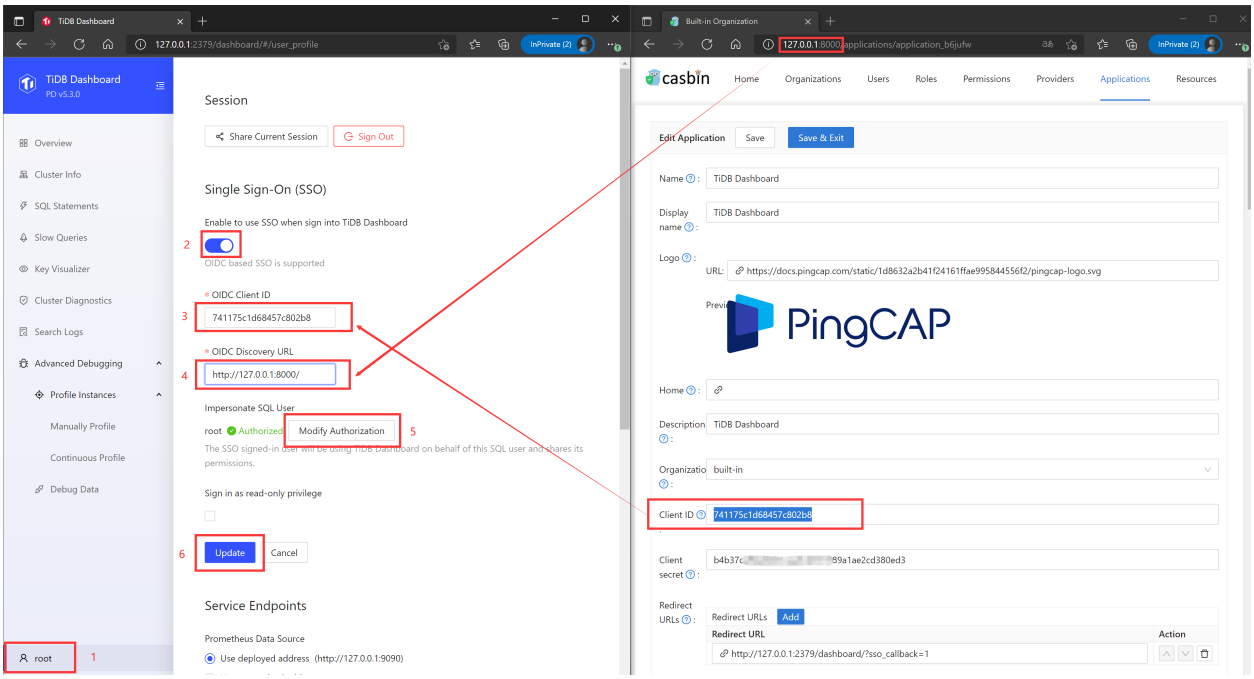


图 636: Settings

至此，TiDB Dashboard 已被配置为使用 Casdoor 进行 SSO 登录。

14.12.1.16 TiDB Dashboard 常见问题

本文汇总了使用 TiDB Dashboard 过程中的常见问题与解决办法。若无法找到对应问题，或者根据指引操作后问题仍然存在，请从 PingCAP 官方或 TiDB 社区获取支持。

14.12.1.16.1 访问

已配置防火墙或反向代理，但访问后被跳转到一个内部地址无法访问 TiDB Dashboard

集群部署有多个 PD 实例的情况下，只有其中某一个 PD 实例会真正运行 TiDB Dashboard 服务，访问其他 PD 实例时会发生浏览器端重定向。若防火墙或反向代理没有为此进行正确配置，就可能出现访问后被重定向到一个被防火墙或反向代理保护的内部地址的情况。

- 参阅TiDB Dashboard 多 PD 实例部署章节了解多 PD 实例下 TiDB Dashboard 的工作原理。
- 参阅通过反向代理使用 TiDB Dashboard 章节了解如何正确配置反向代理。
- 参阅提高 TiDB Dashboard 安全性章节了解如何正确配置防火墙。

双网卡部署时无法通过另一个网卡访问 TiDB Dashboard

PD 中的 TiDB Dashboard 出于安全考虑仅监听部署时所指定的 IP 地址（即只监听在一个网卡上），而非 `0.0.0.0`，因此当主机上安装了多个网卡时，通过另一个网卡将无法访问。

当你使用 `tiup cluster` 或 `tiup playground` 命令部署时，目前尚没有方法改变该行为。推荐使用反向代理将 TiDB Dashboard 安全地暴露给另一个网卡，具体参见通过反向代理使用 TiDB Dashboard 章节。

14.12.1.16.2 界面功能

概况页面中 QPS 及 Latency 显示 `prometheus_not_found` 错误

QPS 及 Latency 监控依赖于集群中已正常部署 Prometheus 监控实例，没有部署的情况下就会显示为错误。向集群中新部署 Prometheus 实例即可解决该问题。

若已经部署 Prometheus 监控实例但仍然显示为错误，可能的原因是您使用的部署工具（TiUP 或 TiDB Operator）版本比较旧，没有自动汇报监控地址，导致 TiDB Dashboard 无法感知并查询监控数据。可以升级到最新的部署工具并重试。

以下给出 TiUP 部署工具的操作方法，对于其他部署工具，请参阅工具对应文档。

1. 升级 TiUP、TiUP Cluster：

```
bash  tiup update --self tiup update cluster --force
```

2. 升级后，部署包含监控节点的新集群时，应当能正常显示监控。
3. 升级后，对于现有集群，可通过再次启动集群的方法汇报监控地址（将 CLUSTER_NAME 替换为实际集群名称）：

```
bash  tiup cluster start CLUSTER_NAME
```

即使集群已经启动，请仍然执行该命令。该命令不会影响集群上正常的业务，但会刷新并上报监控地址，从而能让监控在 TiDB Dashboard 中正常显示。

慢查询页面显示 `invalid connection` 错误

可能的原因是你开启了 TiDB 的 `prepared-plan-cache` 功能。`prepared-plan-cache` 是实验性功能，在某些版本的 TiDB 中可能无法正常运行，开启后可能会导致 TiDB Dashboard（及其他应用）出现该问题。你可以通过系统变量`tidb_enable_prepared_plan_cache` 关闭这项功能。

界面提示 集群中未启动必要组件 NgMonitoring

NgMonitoring 是 TiDB v5.4.0 及以上集群中内置的高级监控组件，用于支撑 TiDB Dashboard 的 持续性能分析和 Top SQL 等功能。使用较新版本 TiUP 部署或升级集群时，NgMonitoring 会自动部署；使用 TiDB Operator 部署集群时，需要依据启用持续性能分析手动部署 NgMonitoring。

如果界面提示 集群中未启动必要组件 NgMonitoring，可按以下方式排查部署问题。

使用 TiUP 部署的集群

第 1 步：检查 TiUP Cluster 版本

1. 检查 TiUP Cluster 版本，NgMonitoring 组件需要较高版本的部署工具支持（TiUP v1.9.0 及以上）：

```shell
tiup cluster --version
```

上述命令可查看 TiUP Cluster 的具体版本。例如：

```
tiup version 1.9.0 tiup
Go Version: go1.17.2
Git Ref: v1.9.0
```

2. 如果 TiUP 版本低于 v1.9.0，升级 TiUP 和 TiUP Cluster 版本至最新。

```shell
tiup update --all
```

第 2 步：在中控机上，通过 TiUP 添加 ng_port 配置项，然后重启 Prometheus 节点。

1. 以编辑模式打开集群的配置文件：

```shell
tiup cluster edit-config ${cluster-name}
```

2. 在 monitoring_servers 下面增加 ng_port:12020 参数：

```
monitoring_servers:
- host: 172.16.6.6
  ng_port: 12020
```

3. 重启 Prometheus 节点：

```shell
tiup cluster reload ${cluster-name} --role prometheus
```

如果执行完上述步骤后依然提示 NgMonitoring 未启动，请从 PingCAP 官方或 TiDB 社区获取支持。

使用 TiDB Operator 部署的集群

请参见 TiDB Operator 文档中启用持续性能分析的步骤部署 NgMonitoring 组件。

使用 TiUP Playground 启动的集群

TiUP Playground (>= v1.8.0) 在启动集群时会自动启动 NgMonitoring 组件。可使用以下命令更新 TiUP Playground 到最新版：

```
tiup update --self
tiup update playground
```

慢查询页面显示 unknown `field` 错误

集群升级后，如果慢查询页面出现 unknown `field` 错误，是由于升级后新版本 TiDB Dashboard 字段与浏览器缓存内的用户偏好设置的字段不兼容导致的。该问题已修复。如果你的集群版本低于 v5.0.3 或 v4.0.14，需要执行以下步骤清理浏览器缓存：

1. 打开 TiDB Dashboard 页面。
2. 打开浏览器的开发者工具。各浏览器的打开方式不同。

    - Firefox：菜单 > Web 开发者 > 切换工具箱（译者注：此处修改为最新的 Firefox Quantum），或者工具栏 > 切换工具箱。
    - Chrome：菜单 > 更多工具 > 开发者工具。
    - Safari：Develop > Show Web Inspector。如果你看不到 Develop 菜单，选择 Preferences > Advanced，然后点击 Show Develop menu in menu bar 复选框。

    以 Chrome 为例：

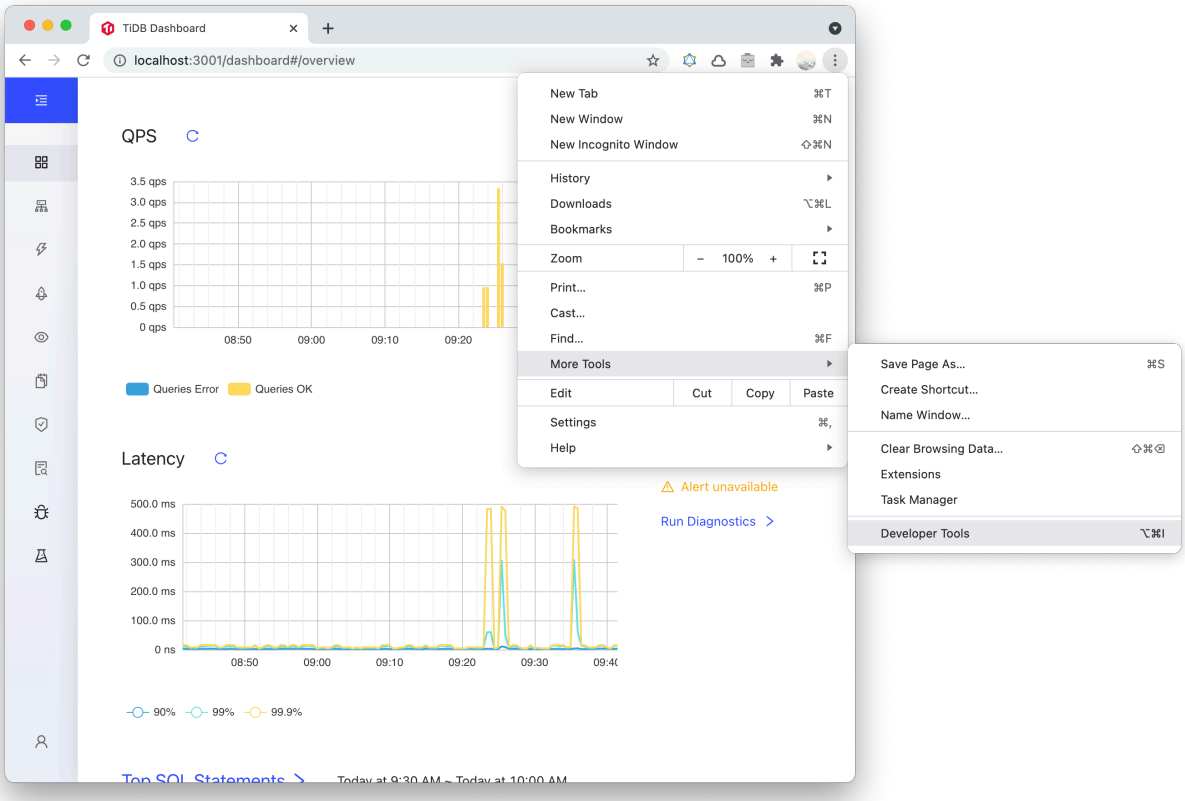


图 637: 打开开发者工具

3. 选中 Application 面板，展开 Local Storage 菜单并选中 TiDB Dashboard 页面的域名，点击 Clear All。

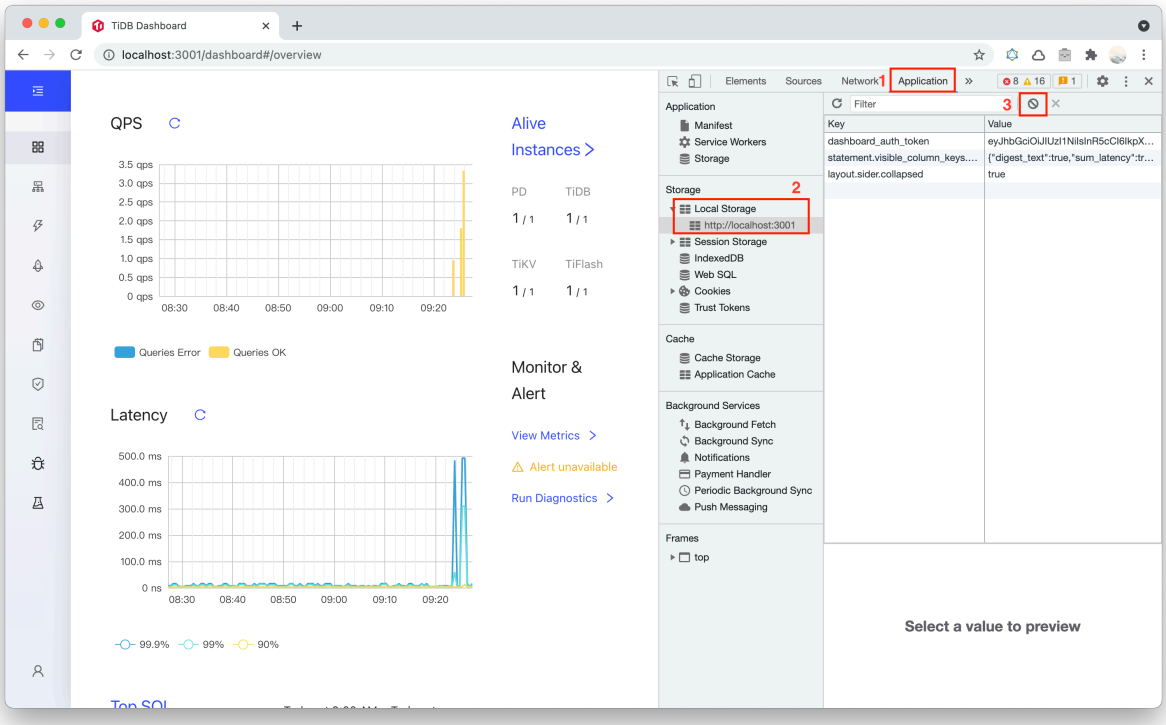


图 638: 清理 Local Storage

14.13 遥测

开启遥测后，TiDB、TiUP 及 TiDB Dashboard 会收集使用情况信息，并将这些信息分享给 PingCAP 用于改善产品，例如，通过这些使用信息，PingCAP 可以了解常见的 TiDB 集群操作，从而确定新功能优先级。

注意：

- 自 2023 年 2 月 20 日起，新发布的 TiDB 和 TiDB Dashboard 版本（含 v6.6.0），默认关闭遥测功能，即默认不再收集使用情况信息分享给 PingCAP。如果集群在升级至这些版本前使用默认的遥测配置，则升级后遥测功能处于关闭状态。具体的版本可参考TiDB 版本发布时间线。
- 从 v1.11.3 起，新部署的 TiUP 默认关闭遥测功能，即默认不再收集使用情况信息。如果从 v1.11.3 之前的 TiUP 版本升级至 v1.11.3 或更高 TiUP 版本，遥测保持升级前的开启或关闭状态。

14.13.1 哪些使用情况信息会被收集？

以下章节具体描述了各个组件收集并分享的使用情况信息。若收集的使用情况信息有变化，将在版本更新说明中告知。

> **注意：**
>
> 在任何情况下，集群中用户存储的数据都不会被收集。另请参阅 PingCAP 隐私声明。

14.13.1.1 TiDB

当 TiDB 遥测功能开启时，TiDB 集群将会以 6 小时为周期收集使用情况信息并分享给 PingCAP，包括（但不限于）：

- 随机生成的遥测标示符
- 集群的部署情况，包括各个组件所在的硬件信息（CPU、内存、磁盘）、组件版本号、操作系统版本号等
- 系统的查询请求状态，例如查询请求次数、持续时长等
- 系统组件的使用情况，例如 Async Commit 功能是否有被使用
- 去识别化处理后的产品遥测数据发送端的 IP 地址

可以通过执行以下 SQL 语句查看 TiDB 收集的使用情况信息内容：

```
ADMIN SHOW TELEMETRY;
```

14.13.1.2 TiDB Dashboard

当 TiDB Dashboard 遥测功能开启时，用户在 TiDB Dashboard 网页界面上进行操作时会将使用情况信息分享给 PingCAP，包括（但不限于）：

- 随机生成的遥测标示符
- 界面访问情况，如访问的 TiDB Dashboard 功能页面名称
- 用户浏览器及操作系统信息，如浏览器名称和版本号、操作系统名称、屏幕分辨率等

可以使用 Chrome 开发者工具的网络功能或 Firefox 开发者工具的网络监视器功能查看 TiDB Dashboard 发送的使用情况信息内容。

14.13.1.3 TiUP

当 TiUP 遥测功能开启时，执行 TiUP 命令时会将使用情况信息分享给 PingCAP，包括（但不限于）：

- 随机生成的遥测标示符
- TiUP 命令的执行情况，如命令执行是否成功、命令执行耗时等
- 使用 TiUP 进行部署的情况，如部署的目标机器硬件信息、组件版本号、修改过的部署配置名称等

使用 TiUP 时，可通过设置 TIUP_CLUSTER_DEBUG=enable 环境变量输出执行命令时收集的使用情况信息，例如：

```
TIUP_CLUSTER_DEBUG=enable tiup cluster list
```

14.13.1.4 TiSpark

> 注意：
>
> 自 TiSpark v3.3 起，默认关闭遥测功能，即 TiSpark 默认不收集使用情况信息，不将这些信息分享给 PingCAP 用于改善产品。

当 TiSpark 遥测功能开启时，Spark 在使用 TiSpark 时会发送会将使用情况信息分享给 PingCAP，包括（但不限于）：

- 随机生成的遥测标示符
- TiSpark 的部分配置信息，如读取引擎、是否开启流式读取等
- 用户集群部署情况，包括 TiSpark 所在节点的机器硬件信息、操作系统信息和组件版本号等

使用 TiSpark 时，可以通过查看 Spark 日志来了解 TiSpark 收集的使用情况，可将 Spark 日志级别调至 INFO 或更低，例如：

```
cat {spark.log} | grep Telemetry report | tail -n 1
```

14.13.2 开启遥测功能

14.13.2.1 开启 TiDB 遥测

14.13.2.1.1 第 1 步：修改配置文件中的遥测配置

在已有 TiDB 集群上开启遥测功能，需要为集群设置enable-telemetry = true，但需要重启集群后才能生效。

以下是在各个部署工具中修改遥测配置的具体步骤。

通过二进制手工部署

创建配置文件 tidb_config.toml 包含如下内容：

```
enable-telemetry = true
```

启动 TiDB 时指定命令行参数 --config=tidb_config.toml 使得该配置生效。

详情参见TiDB 配置参数、TiDB 配置文件描述。

通过 TiUP Playground 试用

创建配置文件 tidb_config.toml 包含如下内容：

```
enable-telemetry = true
```

启动 TiUP Playground 时，指定命令行参数 `--db.config tidb_config.toml` 使得该配置生效，如：

```
tiup playground --db.config tidb_config.toml
```

详情参见TiUP - 本地快速部署 TiDB 集群。

通过 TiUP Cluster 部署

修改部署拓扑文件 `topology.yaml`，新增（或在现有项中添加）以下内容：

```
server_configs:
  tidb:
    enable-telemetry: true
```

通过 TiDB Operator 在 Kubernetes 上部署

在 `tidb-cluster.yaml` 中或者 TidbCluster Custom Resource 中配置 `spec.tidb.config.enable-telemetry: true`。

详情参见在标准 Kubernetes 上部署 TiDB 集群。

> 注意：
>
> 该配置需使用 TiDB Operator v1.1.3 或更高版本才能生效。

14.13.2.1.2 第 2 步：修改系统全局变量的遥测配置

在步骤 1 完成后，还需要修改系统全局变量`tidb_enable_telemetry`，才能打开 TiDB 遥测功能：

```
SET GLOBAL tidb_enable_telemetry = 1;
```

配置文件和全局变量都开启后，遥测功能才能处于开启状态。

14.13.2.2 开启 TiDB Dashboard 遥测

可以修改 PD 配置中`dashboard.enable-telemetry = true` 启动 TiDB Dashboard 遥测功能。对于已启动的集群，该配置需要重启后才能生效。

以下列出在各个部署工具中修改遥测配置的具体步骤。

通过二进制手工部署

创建配置文件 `pd_config.toml` 包含如下内容：

```
[dashboard]
enable-telemetry = true
```

启动 PD 时指定命令行参数 `--config=pd_config.toml` 使得该配置生效。

详情参见PD 配置参数、PD 配置文件描述。

通过 TiUP Playground 试用

创建配置文件 `pd_config.toml` 包含如下内容：

```
[dashboard]
enable-telemetry = true
```

启动 TiUP Playground 时，指定命令行参数 `--pd.config pd_config.toml` 使得该配置生效，如：

```
tiup playground --pd.config pd_config.toml
```

详情参见TiUP - 本地快速部署 TiDB 集群。

通过 TiUP Cluster 部署

修改部署拓扑文件 `topology.yaml`，新增（或在现有项中添加）以下内容：

```
server_configs:
  pd:
    dashboard.enable-telemetry: true
```

通过 TiDB Operator 在 Kubernetes 上部署

在 `tidb-cluster.yaml` 中或者 TidbCluster Custom Resource 中配置 `spec.pd.config.dashboard.enable-telemetry: true`。

详情参见在标准 Kubernetes 上部署 TiDB 集群。

> **注意：**
>
> 该配置需使用 TiDB Operator v1.1.3 或更高版本才能生效。

14.13.2.3 开启 TiUP 遥测

可通过执行以下命令开启 TiUP 遥测功能：

```
tiup telemetry enable
```

14.13.2.4 开启 TiSpark 遥测

可以通过在 Spark 配置文件设置 `spark.tispark.telemetry.enable = true` 来开启 TiSpark 的遥测功能。

14.13.3 禁用遥测功能

14.13.3.1 部署 TiDB 时禁用 TiDB 遥测

在已有 TiDB 集群上开启遥测后，可以为每个集群设置`enable-telemetry = false` 以禁用 TiDB 遥测功能，但需要重启集群后才能生效。

以下是在各个部署工具中修改遥测配置的具体步骤。

通过二进制手工部署

创建配置文件 `tidb_config.toml` 包含如下内容：

```
enable-telemetry = false
```

启动 TiDB 时指定命令行参数 `--config=tidb_config.toml` 使得该配置生效。

详情参见TiDB 配置参数、TiDB 配置文件描述。

通过 TiUP Playground 试用

创建配置文件 `tidb_config.toml` 包含如下内容：

```
enable-telemetry = false
```

启动 TiUP Playground 时，指定命令行参数 `--db.config tidb_config.toml` 使得该配置生效，如：

```
tiup playground --db.config tidb_config.toml
```

详情参见TiUP - 本地快速部署 TiDB 集群。

通过 TiUP Cluster 部署

修改部署拓扑文件 `topology.yaml`，新增（或在现有项中添加）以下内容：

```
server_configs:
  tidb:
    enable-telemetry: false
```

通过 TiDB Operator 在 Kubernetes 上部署

在 `tidb-cluster.yaml` 中或者 TidbCluster Custom Resource 中配置 `spec.tidb.config.enable-telemetry: false`。

详情参见在标准 Kubernetes 上部署 TiDB 集群。

> 注意：
>
> 该配置需使用 TiDB Operator v1.1.3 或更高版本才能生效。

14.13.3.2 动态禁用 TiDB 遥测

对于已部署的 TiDB 集群，还可以修改系统全局变量`tidb_enable_telemetry` 动态禁用 TiDB 遥测功能：

```
SET GLOBAL tidb_enable_telemetry = 0;
```

配置文件的禁用优先级高于全局变量。若通过配置文件禁用了遥测功能，则全局变量的配置将不起作用，遥测功能总是处于关闭状态。

14.13.3.3 禁用 TiDB Dashboard 遥测

可以修改 PD 配置中`dashboard.enable-telemetry = false` 禁用 TiDB Dashboard 遥测功能。对于已启动的集群，该配置需要重启后才能生效。

以下列出在各个部署工具中修改遥测配置的具体步骤。

通过二进制手工部署

创建配置文件 `pd_config.toml` 包含如下内容：

```
[dashboard]
enable-telemetry = false
```

启动 PD 时指定命令行参数 `--config=pd_config.toml` 使得该配置生效。

详情参见PD 配置参数、PD 配置文件描述。

通过 TiUP Playground 试用

创建配置文件 `pd_config.toml` 包含如下内容：

```
[dashboard]
enable-telemetry = false
```

启动 TiUP Playground 时，指定命令行参数 `--pd.config pd_config.toml` 使得该配置生效，如：

```
tiup playground --pd.config pd_config.toml
```

详情参见TiUP - 本地快速部署 TiDB 集群。

通过 TiUP Cluster 部署

修改部署拓扑文件 `topology.yaml`，新增（或在现有项中添加）以下内容：

```
server_configs:
  pd:
    dashboard.enable-telemetry: false
```

通过 TiDB Operator 在 Kubernetes 上部署

在 `tidb-cluster.yaml` 中或者 TidbCluster Custom Resource 中配置 `spec.pd.config.dashboard.enable-telemetry: false`。

详情参见在标准 Kubernetes 上部署 TiDB 集群。

> **注意：**
>
> 该配置需使用 TiDB Operator v1.1.3 或更高版本才能生效。

14.13.3.4 禁用 TiUP 遥测

可通过执行以下命令禁用 TiUP 遥测功能：

```
tiup telemetry disable
```

14.13.3.5 禁用 TiSpark 遥测

可以通过在 Spark 配置文件设置 `spark.tispark.telemetry.enable = false` 来禁用 TiSpark 的遥测功能。

14.13.4 查看遥测启用状态

对于 TiDB 遥测，可通过执行以下 SQL 语句查看遥测状态：

```
ADMIN SHOW TELEMETRY;
```

若 DATA_PREVIEW 列为空，说明遥测没有开启，否则说明遥测已开启。还可以从 LAST_STATUS 列了解上次分享使用情况信息的时间、是否成功等。

对于 TiUP 遥测，可通过执行以下命令查看遥测状态：

```
tiup telemetry status
```

14.13.5 使用情况信息合规性

为了满足不同国家或地区对于此类信息的合规性要求，使用情况信息会按照不同的操作者 IP 地址发送到位于不同国家的服务器，具体如下：

- 若为中国大陆 IP 地址，使用情况信息将会发送并存储于中国大陆境内的公有云服务器。
- 若为中国大陆以外 IP 地址，使用情况信息将会发送并存储于美国的公有云服务器。

可参阅 PingCAP 隐私声明了解详情。

14.14 错误码与故障诊断

本篇文档描述在使用 TiDB 过程中会遇到的问题以及解决方法。

14.14.1 错误码

TiDB 兼容 MySQL 的错误码，在大多数情况下，返回和 MySQL 一样的错误码。关于 MySQL 的错误码列表，详见 [MySQL 5.7 Error Message Reference。另外还有一些 TiDB 特有的错误码：

> **注意：**
>
> 有一部分错误码属于内部错误，正常情况下 TiDB 会自行处理不会直接返回给用户，故没有在此列出。
>
> 如果您遇到了这里没有列出的错误码，请向 PingCAP 工程师或通过官方论坛寻求帮助。

- Error Number: 8001

  请求使用的内存超过 TiDB 内存使用的阈值限制。出现这种错误，可以通过调整系统变量`tidb_mem_quota_query`来增大单个 SQL 使用的内存上限。

- Error Number: 8002

  带有 `SELECT FOR UPDATE` 语句的事务，在遇到写入冲突时，为保证一致性无法进行重试，事务将进行回滚并返回该错误。出现这种错误，应用程序可以安全地重新执行整个事务。

- Error Number: 8003

  `ADMIN CHECK TABLE` 命令在遇到行数据跟索引不一致的时候返回该错误，在检查表中数据是否有损坏时常出现。出现该错误时，请向 PingCAP 工程师或通过官方论坛寻求帮助。

- Error Number: 8004

  单个事务过大，原因及解决方法请参考这里

- Error Number: 8005

  完整的报错信息为 `ERROR 8005 (HY000): Write Conflict, txnStartTS is stale`。

  事务在 TiDB 中遇到了写入冲突。请检查业务逻辑，重试写入操作。

- Error Number: 8018

  当执行重新载入插件时，如果之前插件没有载入过，则会出现该错误。出现该错误，进行插件首次载入即可。

- Error Number: 8019

  重新载入的插件版本与之前的插件版本不一致，无法重新载入插件并报告该错误。可重新载入插件，确保插件的版本与之前载入的插件版本一致。

- Error Number: 8020

  当表被加锁时，如果对该表执行写入操作，将出现该错误。请将表解锁后，再进行尝试写入。

- Error Number: 8021

  当向 TiKV 读取的 key 不存在时将出现该错误，该错误用于内部使用，对外表现为读到的结果为空。

- Error Number: 8022

  事务提交失败，已经回滚，应用程序可以安全的重新执行整个事务。

- Error Number: 8023

  在事务内，写入事务缓存时，设置了空值，将返回该错误。这是一个内部使用的错误，将由内部进行处理，不会返回给应用程序。

- Error Number: 8024

  非法的事务。当事务执行时，发现没有获取事务的 ID (Start Timestamp)，代表正在执行的事务是一个非法的事务，将返回该错误。通常情况下不会出现该问题，当发生时，请向 PingCAP 工程师或通过官方论坛寻求帮助。

- Error Number: 8025

  写入的单条键值对过大。TiDB 默认支持最大 6MB 的单个键值对，超过该限制可适当调整`txn-entry-size-limit`配置项以放宽限制。

- Error Number: 8026

  使用了没有实现的接口函数。该错误仅用于数据库内部，应用程序不会收到这个错误。

- Error Number: 8027

  表结构版本过期。TiDB 采用在线变更表结构的方法。当 TiDB server 表结构版本落后于整个系统的时，执行 SQL 将遇到该错误。遇到该错误，请检查该 TiDB server 与 PD leader 之间的网络。

- Error Number: 8028

  TiDB v6.3.0 引入了元数据锁特性。在关闭元数据锁的情况下，当事务执行时，事务无法感知到 TiDB 的表结构发生了变化。因此，TiDB 在事务提交时，会对事务涉及表的结构进行检查。如果事务执行中表结构发生了变化，则事务将提交失败，并返回该错误。遇到该错误，应用程序可以安全地重新执行整个事务。

  在打开元数据锁的情况下，非 RC 隔离级别中，如果从事务开始到初次访问一个表之间，该表进行了有损的列类型变更操作（例如 `INT` 类型变成 `CHAR` 类型是有损的，`TINYINT` 类型变成 `INT` 类型这种不需要重写数据的则是无损的），则访问该表的语句报错，事务不会自动回滚。用户可以继续执行其他语句，并决定是否回滚或者提交事务。

- Error Number: 8029

  当数据库内部进行数值转换发生错误时，将会出现该错误，该错误仅在内部使用，对外部应用将转换为具体类型的错误。

- Error Number: 8030

  将值转变为带符号正整数时发生了越界，将结果显示为负数。多在告警信息里出现。

- Error Number: 8031

  将负数转变为无符号数时，将负数转变为了正数。多在告警信息里出现。

- Error Number: 8032

  使用了非法的 year 格式。year 只允许 1 位、2 位和 4 位数。

- Error Number: 8033

  使用了非法的 year 值。year 的合法范围是 (1901, 2155)。

- Error Number: 8037

  week 函数中使用了非法的 mode 格式。mode 必须是一位数字，范围 [0, 7]。

- Error Number: 8038

  字段无法获取到默认值。一般作为内部错误使用，转换成其他具体错误类型后，返回给应用程序。

- Error Number: 8040

  尝试进行不支持的操作，比如在 View 和 Sequence 上进行 lock table。

- Error Number: 8047

  设置了不支持的系统变量值，通常在用户设置了数据库不支持的变量值后的告警信息里出现。

- Error Number: 8048

  设置了不支持的隔离级别，如果是使用第三方工具或框架等无法修改代码进行适配的情况，可以考虑通过`tidb_skip_isolation_level_check` 来绕过这一检查。

```
set @@tidb_skip_isolation_level_check = 1;
```

- Error Number: 8050

  设置了不支持的权限类型，遇到该错误请参考TiDB 权限说明进行调整。

- Error Number: 8051

  TiDB 在解析客户端发送的 Exec 参数列表时遇到了未知的数据类型。如果遇到这个错误，请检查客户端是否正常，如果客户端正常请向 PingCAP 工程师或通过官方论坛寻求帮助。

- Error Number: 8052

  来自客户端的数据包的序列号错误。如果遇到这个错误，请检查客户端是否正常，如果客户端正常请向 PingCAP 工程师或通过官方论坛寻求帮助。

- Error Number: 8055

  当前快照过旧，数据可能已经被 GC。可以调大`tidb_gc_life_time` 的值来避免该问题。从 TiDB v4.0.8 版本起，TiDB 会自动为长时间运行的事务保留数据，一般不会遇到该错误。

  有关 GC 的介绍和配置可以参考GC 机制简介和GC 配置文档。

- Error Number: 8059

  自动随机量可用次数用尽无法进行分配。当前没有恢复这类错误的方法。建议在使用 auto random 功能时使用 bigint 以获取最大的可分配次数，并尽量避免手动给 auto random 列赋值。相关的介绍和使用建议可以参考auto random 功能文档。

- Error Number: 8060

  非法的自增列偏移量。请检查 `auto_increment_increment` 和 `auto_increment_offset` 的取值是否符合要求。

- Error Number: 8061

  不支持的 SQL Hint。请参考Optimizer Hints 检查和修正 SQL Hint。

- Error Number: 8062

  SQL Hint 中使用了非法的 token，与 Hint 的保留字冲突。请参考Optimizer Hints 检查和修正 SQL Hint。

- Error Number: 8063

  SQL Hint 中限制内存使用量超过系统设置的上限，设置被忽略。请参考Optimizer Hints 检查和修正 SQL Hint。

- Error Number: 8064

  解析 SQL Hint 失败。请参考Optimizer Hints 检查和修正 SQL Hint。

- Error Number: 8065

  SQL Hint 中使用了非法的整数。请参考Optimizer Hints 检查和修正 SQL Hint。

- Error Number: 8066

  JSON_OBJECTAGG 函数的第二个参数是非法参数。

- Error Number: 8101

  插件 ID 格式错误，正确的格式是 `[name]-[version]` 并且 name 和 version 中不能带有 '-'。

- Error Number: 8102

  无法读取插件定义信息。请检查插件相关的配置。

- Error Number: 8103

  插件名称错误，请检查插件的配置。

- Error Number: 8104

  插件版本不匹配，请检查插件的配置。

- Error Number: 8105

  插件被重复载入。

- Error Number: 8106

  插件定义的系统变量名称没有以插件名作为开头，请联系插件的开发者进行修复。

- Error Number: 8107

  载入的插件未指定版本或指定的版本过低，请检查插件的配置。

- Error Number: 8108

  不支持的执行计划类型。该错误为内部处理的错误，如果遇到该报错请向 PingCAP 工程师或通过官方论坛寻求帮助。

- Error Number: 8109

  analyze 索引时找不到指定的索引。

- Error Number: 8110

  不能进行笛卡尔积运算，需要将配置文件里的 `cross-join` 设置为 `true`。

- Error Number: 8111

  execute 语句执行时找不到对应的 prepare 语句。

- Error Number: 8112

  execute 语句的参数个数与 prepare 语句不符合。

- Error Number: 8113

  execute 语句涉及的表结构在 prepare 语句执行后发生了变化。

- Error Number: 8115

  不支持 prepare 多行语句。

- Error Number: 8116

  不支持 prepare DDL 语句。

- Error Number: 8120

  获取不到事务的 start tso，请检查 PD Server 状态/监控/日志以及 TiDB Server 与 PD Server 之间的网络。

- Error Number: 8121

  权限检查失败，请检查数据库的权限配置。

- Error Number: 8122

  指定了通配符，但是找不到对应的表名。

- Error Number: 8123

  带聚合函数的 SQL 中返回非聚合的列，违反了 `only_full_group_by` 模式。请修改 SQL 或者考虑关闭 `only_full_group_by` 模式。

- Error Number: 8129

  TiDB 尚不支持键长度 >= 65536 的 JSON 对象。

- Error Number: 8130

  完整的报错信息为 `ERROR 8130 (HY000): client has multi-statement capability disabled`。

  从早期版本的 TiDB 升级后，可能会出现该问题。为了减少 SQL 注入攻击的影响，TiDB 目前默认不允许在同一 COM_QUERY 调用中执行多个查询。

  可通过系统变量`tidb_multi_statement_mode` 控制是否在同一 COM_QUERY 调用中执行多个查询。

- Error Number: 8138

  事务试图写入的行值有误，请参考数据索引不一致报错。

- Error Number: 8139

  事务试图写入的行和索引的 handle 值不一致，请参考数据索引不一致报错。

- Error Number: 8140

  事务试图写入的行和索引的值不一致，请参考数据索引不一致报错。

- Error Number: 8141

  事务写入时，对 key 的存在性断言报错，请参考数据索引不一致报错。

- Error Number: 8143

  非事务 DML 语句的一个 batch 报错，语句中止，请参考非事务 DML 语句

- Error Number: 8147

当`tidb_constraint_check_in_place_pessimistic` 设置为 OFF 时，为保证事务的正确性，SQL 语句执行时产生的任何错误都可能导致 TiDB 返回 8147 报错并中止当前事务。具体的错误原因，请参考对应的报错信息。详见约束。

- Error Number: 8154

  目前 LOAD DATA 不支持从 TiDB 服务器本地导入数据，可以指定 LOCAL 从客户端导入，或者将数据上传到 S3/GCS 再进行导入。请参考LOAD DATA。

- Error Number: 8156

  LOAD DATA 语句的文件路径不能为空。需要设置正确的路径再进行导入。请参考LOAD DATA。

- Error Number: 8158

  传入的 S3/GCS 路径无效。请参考外部存储设置有效的路径。

- Error Number: 8159

  TiDB 无法访问 LOAD DATA 语句中传入的 S3/GCS 路径。请确保填入的 S3/GCS bucket 存在，且你输入了正确的 access key 和 secret access key 以让 TiDB 服务器有权限访问 S3/GCS 对应的 bucket。

- Error Number: 8160

  LOAD DATA 读取数据文件失败。请根据具体的错误提示进行处理。

- Error Number: 8162

  LOAD DATA 语句存在错误。请参考LOAD DATA 查看已支持的功能。

- Error Number: 8200

  尚不支持的 DDL 语法。请参考与 MySQL DDL 的兼容性。

- Error Number: 8214

  DDL 操作被 admin cancel 操作终止。

- Error Number: 8215

  Admin Repair 表失败，如果遇到该报错请向 PingCAP 工程师或通过官方论坛寻求帮助。

- Error Number: 8216

  自动随机列使用的方法不正确，请参考auto random 功能文档进行修改。

- Error Number: 8223

  检测出数据与索引不一致的错误，如果遇到该报错请向 PingCAP 工程师或通过官方论坛寻求帮助。

- Error Number: 8224

  找不到 DDL job，请检查 restore 操作指定的 job id 是否存在。

- Error Number: 8225

  DDL 已经完成，无法被取消。

- Error Number: 8226

  DDL 几乎要完成了，无法被取消。

- Error Number: 8227

  创建 Sequence 时使用了不支持的选项，支持的选项的列表可以参考Sequence 使用文档。

- Error Number: 8228

  在 Sequence 上使用 `setval` 时指定了不支持的类型，该函数的示例可以在Sequence 使用文档中找到。

- Error Number: 8229

  事务超过存活时间，遇到该问题可以提交或者回滚当前事务，开启一个新事务。

- Error Number: 8230

  TiDB 目前不支持在新添加的列上使用 Sequence 作为默认值，如果尝试进行这类操作会返回该错误。

- Error Number: 8248

  资源组已存在。在重复创建资源组时返回该错误。

- Error Number: 8249

  资源组不存在。在修改或绑定不存在的资源组时返回该错误。请参考创建资源组。

- Error Number: 8250

  完整的报错信息如下：

  ```
  ERROR 8250 (HY000): Resource control feature is disabled. Run "SET GLOBAL tidb_enable_resource_control
  ↪ ='on'" to enable the feature
  ```

  资源控制的功能没有打开时，使用资源管控 (Resource Control) 相关功能会返回该错误。你可以开启全局变量`tidb_enable_resource_control` 启用资源管控。

- Error Number: 8251

  `Resource Control` 组件在 TiDB 启动时进行初始化，相关配置会从 `Resource Control` 的服务端 `Resource Manager` 上获取，如果此过程中出错，则会返回此错误。

- Error Number: 8252

  完整的报错信息如下：

  ```
  ERROR 8252 (HY000): Exceeded resource group quota limitation
  ```

  在尝试消耗超过资源组的限制时返回该错误。一般出现该错误，是由于单次事务太大或者并发太多导致，需调整事务大小或减少客户端并发数。

- Error Number: 9001

  完整的报错信息为 `ERROR 9001 (HY000): PD Server Timeout`。

  请求 PD 超时，请检查 PD Server 状态/监控/日志以及 TiDB Server 与 PD Server 之间的网络。

- Error Number: 9002

  完整的报错信息为 `ERROR 9002 (HY000): TiKV Server Timeout`。

  请求 TiKV 超时，请检查 TiKV Server 状态/监控/日志以及 TiDB Server 与 TiKV Server 之间的网络。

- Error Number: 9003

  完整的报错信息为 `ERROR 9003 (HY000): TiKV Server is Busy`。

  TiKV 操作繁忙，一般出现在数据库负载比较高时，请检查 TiKV Server 状态/监控/日志。

- Error Number: 9004

  完整的报错信息为 `ERROR 9004 (HY000): Resolve Lock Timeout`。

  清理锁超时，当数据库上承载的业务存在大量的事务冲突时，会遇到这种错误，请检查业务代码是否有锁争用。

- Error Number: 9005

  完整的报错信息为 `ERROR 9005 (HY000): Region is unavailable`。

  访问的 Region 不可用，某个 Raft Group 不可用，如副本数目不足，出现在 TiKV 比较繁忙或者是 TiKV 节点停机的时候，请检查 TiKV Server 状态/监控/日志。

- Error Number: 9006

  完整的报错信息为 `ERROR 9006 (HY000): GC life time is shorter than transaction duration`。

  GC Life Time 间隔时间过短，长事务本应读到的数据可能被清理了。你可以使用如下命令修改`tidb_gc_life_time` 的值：

  ```
  SET GLOBAL tidb_gc_life_time = '30m';
  ```

  其中 30m 代表仅清理 30 分钟前的数据，这可能会额外占用一定的存储空间。

- Error Number: 9007

  报错信息以 `ERROR 9007 (HY000): Write conflict` 开头。

  如果报错信息中含有“reason=LazyUniquenessCheck”，说明是悲观事务并且设置了`@@tidb_constraint_check_in_place_pes`
  `=OFF`，业务中存在唯一索引上的写冲突，此时悲观事务不能保证执行成功。可以在应用测重试事务，或将该变量设置成 ON 绕过。详见约束。

- Error Number: 9008

  同时向 TiKV 发送的请求过多，超过了限制。请调大 `tidb_store_limit` 或将其设置为 0 来取消对请求流量的限制。

- Error Number: 9010

  TiKV 无法处理这条 raft log，请检查 TiKV Server 状态/监控/日志。

- Error Number: 9012

  请求 TiFlash 超时。请检查 TiFlash Server 状态/监控/日志以及 TiDB Server 与 TiFlash Server 之间的网络。

- Error Number: 9013

  TiFlash 操作繁忙。该错误一般出现在数据库负载比较高时。请检查 TiFlash Server 的状态/监控/日志。

14.14.1.1 MySQL 原生报错汇总

- Error Number: 2013 (HY000)

  完整的报错信息为 `ERROR 2013 (HY000): Lost connection to MySQL server during query`。

  排查方法如下：

  - log 中是否有 panic

  - dmesg 中是否有 oom，命令：`dmesg -T | grep -i oom`
  - 长时间没有访问，也会收到这个报错，一般是 tcp 超时导致的，tcp 长时间不用，会被操作系统 kill。

- Error Number: 1105 (HY000)

  完整的报错信息为 `ERROR 1105 (HY000): other error: unknown error Wire Error(InvalidEnumValue (4004))`

  这类问题一般是 TiDB 和 TiKV 版本不匹配，在升级过程尽量一起升级，避免版本 mismatch。

- Error Number: 1148 (42000)

  完整的报错信息为 `ERROR 1148 (42000): the used command is not allowed with this TiDB version`。

  这个问题是因为在执行 `LOAD DATA LOCAL` 语句的时候，MySQL 客户端不允许执行此语句（即 `local_infile` 选项为 0）。解决方法是在启动 MySQL 客户端时，用 `--local-infile=1` 选项。具体启动指令类似：`mysql --local-infile=1 -u root -h 127.0.0.1 -P 4000`。有些 MySQL 客户端需要设置而有些不需要设置，原因是不同版本的 MySQL 客户端对 `local-infile` 的默认值不同。

- Error Number: 9001 (HY000)

  完整的报错信息为 `ERROR 9001 (HY000): PD server timeout start timestamp may fall behind safe point`

  这个报错一般是 TiDB 访问 PD 出了问题，TiDB 后台有个 worker 会不断地从 PD 查询 safepoint，如果超过 100s 查不成功就会报这个错。一般是因为 PD 磁盘操作过忙、反应过慢，或者 TiDB 和 PD 之间的网络有问题。TiDB 常见错误码请参考错误码与故障诊断。

- TiDB 日志中的报错信息：EOF

  当客户端或者 proxy 断开连接时，TiDB 不会立刻察觉连接已断开，而是等到开始往连接返回数据时，才发现连接已断开，此时日志会打印 EOF 错误。

14.14.2 故障诊断

参见故障诊断文档。

14.15 通过拓扑 label 进行副本调度

> **注意：**
>
> TiDB 在 v5.3.0 中引入了Placement Rules in SQL。使用该功能，你可以更方便地配置表和分区的位置。在未来版本中，Placement Rules in SQL 可能取代通过 PD 配置放置规则的功能。

为了提升 TiDB 集群的高可用性和数据容灾能力，我们推荐让 TiKV 节点尽可能在物理层面上分散，例如让 TiKV 节点分布在不同的机架甚至不同的机房。PD 调度器根据 TiKV 的拓扑信息，会自动在后台通过调度使得 Region 的各个副本尽可能隔离，从而使得数据容灾能力最大化。

要让这个机制生效，需要在部署时进行合理配置，把集群的拓扑信息（特别是 TiKV 的位置）上报给 PD。阅读本章前，请先确保阅读TiUP 部署方案。

14.15.1 根据集群拓扑配置 labels

14.15.1.1 设置 TiKV 和 TiFlash 的 labels

TiKV 和 TiFlash 支持在命令行参数或者配置文件中以键值对的形式绑定一些属性，我们把这些属性叫做标签 (label)。TiKV 和 TiFlash 在启动后，会将自身的标签上报给 PD，因此可以使用标签来标识 TiKV 和 TiFlash 节点的地理位置。

比如集群的拓扑结构分成四层：机房 (zone) -> 数据中心 (dc) -> 机架 (rack) -> 主机 (host)，就可以使用这 4 个标签来设置 TiKV 和 TiFlash 的位置。

使用命令行参数的方式启动一个 TiKV 实例：

```
tikv-server --labels zone=<zone>,dc=<dc>,rack=<rack>,host=<host>
```

使用配置文件的方式：

```
[server]
[server.labels]
zone = "<zone>"
dc = "<dc>"
rack = "<rack>"
host = "<host>"
```

TiFlash 支持通过 tiflash-learner.toml （ tiflash-proxy 的配置文件 ）的方式设置 labels：

```
[server]
[server.labels]
zone = "<zone>"
dc = "<dc>"
rack = "<rack>"
host = "<host>"
```

14.15.1.2 设置 TiDB 的 labels（可选）

如果需要使用Follower Read 的优先读同一区域副本的功能，需要为 TiDB 节点配置相关的 labels。

TiDB 支持使用配置文件的方式设置 labels：

```
[labels]
zone = "<zone>"
dc = "<dc>"
rack = "<rack>"
host = "<host>"
```

注意：

> 目前，TiDB 依赖 zone 标签匹配选择同一区域的副本。如果需要使用此功能，需要在 PD location-labels 配置中包含 zone，并在 TiDB、TiKV 和 TiFlash 设置的 labels 中包含 zone。关于如何设置 TiKV 和 TiFlash 的 labels，可参考设置 TiKV 和 TiFlash 的 labels。

14.15.1.3 设置 PD 的 location-labels 配置

根据前面的描述，标签可以是用来描述 TiKV 属性的任意键值对，但 PD 无从得知哪些标签是用来标识地理位置的，而且也无从得知这些标签的层次关系。因此，PD 也需要一些配置来使得 PD 理解 TiKV 节点拓扑。

PD 上的配置叫做 location-labels，是一个字符串数组。该配置的每一项与 TiKV labels 的 key 是对应的，而且其中每个 key 的顺序代表不同标签的级别关系（从左到右，隔离级别依次递减）。

location-labels 没有默认值，你可以根据具体需求来设置该值，包括 zone、rack、host 等等。同时，location-labels 对标签级别的数量也没有限制（即不限定于 3 个），只要其级别与 TiKV 服务器的标签匹配，则可以配置成功。

> 注意：
>
> - 必须同时配置 PD 的 location-labels 和 TiKV 的 labels 参数，否则 PD 不会根据拓扑结构进行调度。
> - 如果你使用 Placement Rules in SQL，只需要配置 TiKV 的 labels 即可。Placement Rules in SQL 目前不兼容 PD location-labels 设置，会忽略该设置。不建议 location-labels 与 Placement Rules in SQL 混用，否则可能产生非预期的结果。

你可以根据集群状态来选择不同的配置方式：

- 在集群初始化之前，可以通过 PD 的配置文件进行配置：

```
[replication]
location-labels = ["zone", "rack", "host"]
```

- 如果需要在 PD 集群初始化完成后进行配置，则需要使用 pd-ctl 工具进行在线更改：

```
pd-ctl config set location-labels zone,rack,host
```

14.15.1.4 设置 PD 的 isolation-level 配置

在配置了 location-labels 的前提下，用户可以还通过 isolation-level 配置来进一步加强对 TiKV 集群的拓扑隔离要求。假设按照上面的说明通过 location-labels 将集群的拓扑结构分成三层：机房 (zone) -> 机架 (rack) -> 主机 (host)，并对 isolation-level 作如下配置：

```
[replication]
isolation-level = "zone"
```

当 PD 集群初始化完成后，需要使用 pd-ctl 工具进行在线更改：

```
pd-ctl config set isolation-level zone
```

其中，isolation-level 配置是一个字符串，需要与 location-labels 的其中一个 key 对应。该参数限制 TiKV 拓扑集群的最小且强制隔离级别要求。

> **注意：**
>
> isolation-level 默认情况下为空，即不进行强制隔离级别限制，若要对其进行设置，必须先配置 PD 的 location-labels 参数，同时保证 isolation-level 的值一定为 location-labels 中的一个。

14.15.1.5 使用 TiUP 进行配置（推荐）

如果使用 TiUP 部署集群，可以在初始化配置文件中统一进行 location 相关配置。TiUP 会负责在部署时生成对应的 TiKV、PD 和 TiFlash 配置文件。

下面的例子定义了 zone 和 host 两层拓扑结构。集群的 TiKV 和 TiFlash 分布在三个 zone，z1、z2 和 z3。每个 zone 内有四台主机，z1 两台主机分别部署两个 TiKV 实例，另外两台分别部署一个 TiFlash 实例，z2 和 z3 其中两台主机分别部署一个 TiKV 实例，另外两台分别部署一个 TiFlash 实例。以下例子中 tikv-n 代表第 n 个 TiKV 节点的 IP 地址，tiflash-n 代表第 n 个 TiFlash 节点的 IP 地址。

```
server_configs:
  pd:
    replication.location-labels: ["zone", "host"]

tikv_servers:
## z1
  - host: tikv-1
    config:
      server.labels:
        zone: z1
        host: h1
   - host: tikv-2
    config:
      server.labels:
        zone: z1
        host: h1
  - host: tikv-3
    config:
      server.labels:
        zone: z1
        host: h2
  - host: tikv-4
```

```
    config:
      server.labels:
        zone: z1
        host: h2
## z2
  - host: tikv-5
    config:
      server.labels:
        zone: z2
        host: h1
   - host: tikv-6
    config:
      server.labels:
        zone: z2
        host: h2
## z3
  - host: tikv-7
    config:
      server.labels:
        zone: z3
        host: h1
  - host: tikv-8
    config:
      server.labels:
        zone: z3
        host: h2

tiflash_servers:
## z1
  - host: tiflash-1
    learner_config:
      server.labels:
        zone: z1
        host: h3
   - host: tiflash-2
    learner_config:
      server.labels:
        zone: z1
        host: h4
## z2
  - host: tiflash-3
    learner_config:
      server.labels:
        zone: z2
        host: h3
```

```
   - host: tiflash-4
    learner_config:
      server.labels:
        zone: z2
        host: h4
## z3
  - host: tiflash-5
    learner_config:
      server.labels:
        zone: z3
        host: h3
  - host: tiflash-6
    learner_config:
      server.labels:
        zone: z3
        host: h4
```

详情参阅TiUP 跨数据中心部署拓扑。

> **注意：**
>
> 如果你未在配置文件中配置 `replication.location-labels` 项，使用该拓扑配置文件部署集群时可能会报错。建议在部署集群前，确认 `replication.location-labels` 已配置。

14.15.2 基于拓扑 label 的 PD 调度策略

PD 在副本调度时，会按照 label 层级，保证同一份数据的不同副本尽可能分散。

下面以上一节的拓扑结构为例分析。

假设集群副本数设置为 3 (`max-replicas=3`)，因为总共有 3 个 zone，PD 会保证每个 Region 的 3 个副本分别放置在 z1/z2/z3，这样当任何一个数据中心发生故障时，TiDB 集群依然是可用的。

假如集群副本数设置为 5 (`max-replicas=5`)，因为总共只有 3 个 zone，在这一层级 PD 无法保证各个副本的隔离，此时 PD 调度器会退而求其次，保证在 host 这一层的隔离。也就是说，会出现一个 Region 的多个副本分布在同一个 zone 的情况，但是不会出现多个副本分布在同一台主机。

在 5 副本配置的前提下，如果 z3 出现了整体故障或隔离，并且 z3 在一段时间后仍然不能恢复（由 `max-store-down-time` 控制），PD 会通过调度补齐 5 副本，此时可用的主机只有 4 个了，故而无法保证 host 级别的隔离，于是可能出现多个副本被调度到同一台主机的情况。

但假如 `isolation-level` 设置不为空，值为 zone，这样就规定了 Region 副本在物理层面上的最低隔离要求，也就是说 PD 一定会保证同一 Region 的副本分散于不同的 zone 之上。即便遵循此隔离限制会无法满足 `max-replicas` 的多副本要求，PD 也不会进行相应的调度。例如，当前存在 TiKV 集群的三个机房 z1/z2/z3，在三副本的设置下，PD 会将同一 Region 的三个副本分别分散调度至这三个机房。若此时 z1 整个机房发生了停电事故并在一段时间后（由`max-store-down-time` 控制，默认为 30 分钟）仍然不能恢复，PD 会认为 z1 上的

Region 副本不再可用。但由于 `isolation-level` 设置为了 `zone`，PD 需要严格保证不同的 Region 副本不会落到同一 zone 上。此时的 z2 和 z3 均已存在副本，则 PD 在 `isolation-level` 的最小强制隔离级别限制下便不会进行任何调度，即使此时仅存在两个副本。

类似地，`isolation-level` 为 `rack` 时，最小隔离级别便为同一机房的不同 rack。在此设置下，如果能在 zone 级别保证隔离，会首先保证 zone 级别的隔离。只有在 zone 级别隔离无法完成时，才会考虑避免出现在同一 zone 同一 rack 的调度，并以此类推。

总的来说，PD 能够根据当前的拓扑结构使得集群容灾能力最大化。所以如果用户希望达到某个级别的容灾能力，就需要根据拓扑结构在对应级别提供多于副本数 (`max-replicas`) 的机器。同时 TiDB 也提供了诸如 `isolation-level` 这样的强制隔离级别设置，以便更灵活地根据场景来控制对数据的拓扑隔离级别。

15 常见问题解答 (FAQ)

15.1 TiDB 产品常见问题解答汇总

本文整合汇总了 TiDB 使用过程中常见问题解答的 (FAQ) 文档。

分类	相关文档
产品架构和原理	产品架构常见问题

| 安装部署 |

安装部署常见问题

TiUP 常见问题

Kubernetes 上的 TiDB 集群常见问题

| | 数据迁移 |

数据迁移常见问题

数据导入

TiDB Lightning 常见问题

TiDB Data Migration 常见问题

增量数据同步

TiCDC 常见问题解答

TiDB Binlog 常见问题

| | 数据备份与恢复 | 备份与恢复常见问题 | | SQL 使用 | SQL 操作常见问题 | | 集群升级 | TiDB 集群升级常见问题 | | 集群管理 |

TiDB 集群管理常见问题

TiUniManager 常见问题

| | 监控报警 |

TiDB 监控常见问题

TiDB Dashboard 常见问题

TiDB 集群报警规则

| | 高可用和高可靠 |

TiDB 高可用常见问题

TiDB 高可靠常见问题

| | 常见错误码 | 错误码与故障诊断 |

15.2 TiDB 产品常见问题

15.2.1 1.1 TiDB 介绍及整体架构

15.2.1.1 1.1.1 TiDB 是什么？

TiDB 是 PingCAP 公司自主设计、研发的开源分布式关系型数据库，是一款同时支持在线事务处理与在线分析处理 (Hybrid Transactional and Analytical Processing, HTAP) 的融合型分布式数据库产品，具备水平扩容或者缩容、金融级高可用、实时 HTAP、云原生的分布式数据库、兼容 MySQL 5.7 协议和 MySQL 生态等重要特性。目标是为用户提供一站式 OLTP (Online Transactional Processing)、OLAP (Online Analytical Processing)、HTAP 解决方案。TiDB 适合高可用、强一致要求较高、数据规模较大等各种应用场景。更多详细信息，请参阅TiDB 简介。

15.2.1.2 1.1.2 TiDB 整体架构

参见TiDB 整体架构，以及 TiDB 数据库的存储、计算与调度。

15.2.1.3 1.1.3 TiDB 是基于 MySQL 开发的吗？

不是。虽然 TiDB 支持 MySQL 语法和协议，但是 TiDB 是由 PingCAP 团队完全自主开发的产品。

15.2.1.4 1.1.4 TiDB、TiKV、Placement Driver (PD) 主要作用？

- TiDB 是 Server 计算层，主要负责 SQL 的解析、制定查询计划、生成执行器。
- TiKV 是分布式 Key-Value 存储引擎，用来存储真正的数据，简而言之，TiKV 是 TiDB 的存储引擎。
- PD 是 TiDB 集群的管理组件，负责存储 TiKV 的元数据，同时也负责分配时间戳以及对 TiKV 做负载均衡调度。

15.2.1.5 1.1.5 TiDB 易用性如何？

TiDB 使用起来很简单，可以将 TiDB 集群当成 MySQL 来用。你可以将 TiDB 用在任何以 MySQL 作为后台存储服务的应用中，并且基本上不需要修改应用代码，同时你可以用大部分流行的 MySQL 管理工具来管理 TiDB。

15.2.1.6 1.1.6 TiDB 和 MySQL 兼容性如何？

TiDB 支持绝大部分 MySQL 5.7 的语法，但目前还不支持触发器、存储过程、自定义函数等。详情参见与 MySQL 兼容性对比。

15.2.1.7 1.1.7 TiDB 支持分布式事务吗？

支持。无论是一个地方的几个节点，还是跨多个数据中心的多个节点，TiDB 均支持 ACID 分布式事务。

TiDB 事务模型灵感源自 Google Percolator 模型，主体是一个两阶段提交协议，并进行了一些实用的优化。该模型依赖于一个时间戳分配器，为每个事务分配单调递增的时间戳，这样就检测到事务冲突。在 TiDB 集群中，PD 承担时间戳分配器的角色。

15.2.1.8 1.1.8 TiDB 支持哪些编程语言？

只要支持 MySQL Client/Driver 的编程语言，都可以直接使用 TiDB。

15.2.1.9 1.1.9 TiDB 是否支持其他存储引擎？

是的，除了 TiKV 之外，TiDB 还支持一些单机存储引擎，比如 UniStore 和 MockTiKV。注意，TiDB 后续版本可能不再支持 MockTiKV。

要查看 TiDB 支持的存储引擎，可使用以下命令：

```
./bin/tidb-server -h
```

返回结果如下：

```
Usage of ./bin/tidb-server:
  -L string
        log level: info, debug, warn, error, fatal (default "info")
  -P string
        tidb server port (default "4000")
  -V    print version information and exit (default false)
.........
  -store string
        registered store name, [tikv, mocktikv, unistore] (default "unistore")
  ......
```

15.2.1.10 1.1.10 除了官方文档，有没有其他 TiDB 知识获取途径？

- 官方文档：获取 TiDB 相关知识最主要、最及时的途径。
- 官方博客：了解产品技术解读、观点洞察、案例实践。
- AskTUG 社区论坛：与社区用户、技术专家互动交流。
- PingCAP Education：学习线上课程，获得数据库能力认证。

15.2.1.11 1.1.11 TiDB 用户名长度限制？

在 TiDB 中，用户名最长为 32 个字符。

15.2.1.12 1.1.12 TiDB 中列数和行大小的限制是多少？

- TiDB 中默认的最大列数为 1017。你可以调整这个限制，最大可调整到 4096 列。
- TiDB 中默认单行大小不超过 6 MB。你可以调整这个限制，最大可调整到 120 MB。

更多信息，请参考TiDB 限制。

15.2.1.13 1.1.13 TiDB 是否支持 XA？

虽然 TiDB 的 JDBC 驱动用的就是 MySQL Connector/J，但是当使用 Atomikos 的时候，数据源要配置成类似这样的配置：`type="com.mysql.jdbc.jdbc2.optional.MysqlXADataSource"`。MySQL JDBC XADataSource 连接 TiDB 的模式目前是不支持的。MySQL JDBC 中配置好的 XADataSource 模式，只对 MySQL 数据库起作用（DML 去修改 redo 等）。

Atomikos 配好两个数据源后，JDBC 驱动都要设置成 XA 模式，然后 Atomikos 在操作 TM 和 RM (DB) 的时候，会通过数据源的配置，发起带有 XA 指令到 JDBC 层。JDBC 层 XA 模式启用的情况下，会对 InnoDB（如果是 MySQL 的话）下发操作一连串 XA 逻辑的动作，包括 DML 去变更 redo log 等，就是两阶段递交的那些操作。TiDB 目前的引擎版本中，没有对上层应用层 JTA/XA 的支持，不解析这些 Atomikos 发过来的 XA 类型的操作。

MySQL 是单机数据库，只能通过 XA 来满足跨数据库事务，而 TiDB 本身就通过 Google 的 Percolator 事务模型支持分布式事务，性能稳定性比 XA 要高出很多，所以不会也不需要支持 XA。

15.2.1.14 1.1.14 TiDB 如何在不影响性能的情况下支持对列存储引擎 (TiFlash) 的高并发 INSERT 或 UPDATE 操作？

- TiFlash 引入了 DeltaTree 这种特殊结构来处理列存引擎的修改。
- TiFlash 作为 Raft Group 中的 Learner 角色，不参与 log commit 选举，也不会写入数据。这意味着 DML 操作不需要等待 TiFlash 的确认，所以 TiFlash 不会影响 OLTP 的性能。另外，TiFlash 和 TiKV 分开部署在不同的实例上，不会相互影响。

15.2.1.15 1.1.15 TiFlash 是否具有最终一致性？

是的，TiFlash 默认保持数据强一致性。

15.2.2 1.2 TiDB 原理

15.2.2.1 1.2.1 存储 TiKV 详细解读

三篇文章了解 TiDB 技术内幕 - 说存储

15.2.2.2 1.2.2 计算 TiDB 详细解读

三篇文章了解 TiDB 技术内幕 - 说计算

15.2.2.3 1.2.3 调度 PD 详细解读

三篇文章了解 TiDB 技术内幕 - 谈调度

15.3 SQL 操作常见问题

本文档介绍 TiDB 中常见的 SQL 操作问题。

15.3.1 TiDB 是否支持二级键？

支持。你可以在具有唯一二级索引的非主键列上设置NOT NULL 约束。在这种情况下，该列用作二级键。

15.3.2 TiDB 在对大表执行 DDL 操作时，性能表现如何？

TiDB 在对大表执行 DDL 操作时，一般不会有什么问题。TiDB 支持在线 DDL 操作，且这些 DDL 操作不会阻塞 DML 操作。

对于添加列、删除列或删除索引等 DDL 操作，TiDB 可以快速完成这些操作。

对于添加索引等 DDL 操作，TiDB 需要进行回填 (backfill) 操作，这个过程需要较长的时间（取决于表的大小）和额外的资源消耗。对在线业务的影响可调节。TiDB 可以通过多线程进行 backfill，资源消耗可通过以下系统变量进行设置：

- tidb_ddl_reorg_worker_cnt
- tidb_ddl_reorg_priority
- tidb_ddl_error_count_limit
- tidb_ddl_reorg_batch_size

15.3.3 如何选择正确的查询计划？是否需要使用优化器提示？还是可以使用提示？

TiDB 包含一个基于成本的优化器。在大多数情况下，优化器会为你选择最优的查询计划。如果优化器工作欠佳，你可以使用优化器提示来干预优化器。

另外，你还可以使用执行计划绑定来为特定的 SQL 语句固定查询计划。

15.3.4 如何阻止特定的 SQL 语句执行（或者将某个 SQL 语句加入黑名单）?

你可以使用MAX_EXECUTION_TIME Hint 来创建SQL 绑定，将特定语句的执行时间限制为一个较小的值（例如 1ms）。这样，语句就会在超过限制时自动终止。

例如，要阻止执行 `SELECT * FROM t1, t2 WHERE t1.id = t2.id`，可以使用以下 SQL 绑定将语句的执行时间限制为 1ms：

```
CREATE GLOBAL BINDING for
    SELECT * FROM t1, t2 WHERE t1.id = t2.id
USING
    SELECT /*+ MAX_EXECUTION_TIME(1) */ * FROM t1, t2 WHERE t1.id = t2.id;
```

> 注意：
>
> MAX_EXECUTION_TIME 的精度大约为 100ms。在 TiDB 终止 SQL 语句之前，TiKV 中的任务可能已经开始执行。为了减少这种情况下 TiKV 的资源消耗，建议将系统变量tidb_enable_paging 的值设置为 ON。

删除该 SQL 绑定可以移除限制。

```
DROP GLOBAL BINDING for
    SELECT * FROM t1, t2 WHERE t1.id = t2.id;
```

15.3.5 TiDB 对哪些 MySQL variables 兼容？

详细可参考系统变量。

15.3.6 省略 ORDER BY 条件时 TiDB 中返回结果的顺序与 MySQL 中的不一致

这不是 bug。返回结果的顺序视不同情况而定，不保证顺序统一。

MySQL 中，返回结果的顺序可能较为固定，因为查询是通过单线程执行的。但升级到新版本后，查询计划也可能变化。无论是否期待返回结果有序，都推荐使用 ORDER BY 条件。

ISO/IEC 9075:1992, Database Language SQL- July 30, 1992 对此有如下表述：

> If an <order by clause> is not specified, then the table specified by the <cursor specification> is T and the ordering of rows in T is implementation-dependent. (如果未指定 <order by 条件>，通过 <cursor specification> 指定的表为 T，那么 T 表中的行顺序视执行情况而定。)

以下两条查询的结果都是合法的：

```
> select * from t;
+------+------+
| a    | b    |
+------+------+
|    1 |    1 |
|    2 |    2 |
+------+------+
2 rows in set (0.00 sec)
```

```
> select * from t; -- 不确定返回结果的顺序
+------+------+
| a    | b    |
```

```
+------+------+
|    2 |    2 |
|    1 |    1 |
+------+------+
2 rows in set (0.00 sec)
```

如果 ORDER BY 中使用的列不是唯一列，就无法确定该语句返回结果的顺序。在以下示例中，a 列有重复值，因此只有 ORDER BY a, b 能确定返回结果的顺序。

```
> select * from t order by a;
+------+------+
| a    | b    |
+------+------+
|    1 |    1 |
|    2 |    1 |
|    2 |    2 |
+------+------+
3 rows in set (0.00 sec)
```

在以下示例中，order by a 能确定 a 列的顺序，但不能确定 b 列的顺序。

```
> select * from t order by a;
+------+------+
| a    | b    |
+------+------+
|    1 |    1 |
|    2 |    2 |
|    2 |    1 |
+------+------+
3 rows in set (0.00 sec)
```

在 TiDB 中，你还可以使用系统变量tidb_enable_ordered_result_mode 来指定是否对最终的输出结果进行自动排序。

15.3.7 TiDB 是否支持 SELECT FOR UPDATE？

支持。当 TiDB 使用悲观锁（自 TiDB v3.0.8 起默认使用）时，TiDB 中 SELECT FOR UPDATE 的行为与 MySQL 中的基本一致。

当 TiDB 使用乐观锁时，SELECT FOR UPDATE 不会在事务启动时对数据加锁，而是在提交事务时检查冲突。如果检查出冲突，会回滚待提交的事务。

详情参考SELECT 语句语法元素说明。

15.3.8 TiDB 的 codec 能保证 UTF8 的字符串是 memcomparable 的吗？我们的 key 需要支持 UTF8，有什么编码建议吗？

TiDB 字符集默认就是 UTF8 而且目前只支持 UTF8，字符串就是 memcomparable 格式的。

15.3.9 一个事务中的语句数量最大是多少？

一个事务中的语句数量，默认限制最大为 5000 条。

在使用乐观事务并开启事务重试的情况下，默认限制 5000，可通过stmt-count-limit 调整。

15.3.10 TiDB 中，为什么出现后插入数据的自增 ID 反而小？

TiDB 的自增 ID (AUTO_INCREMENT) 只保证自增且唯一，并不保证连续分配。TiDB 目前采用批量分配的方式，所以如果在多台 TiDB server 上同时插入数据，分配的自增 ID 会不连续。当多个线程并发往不同的 TiDB server 插入数据的时候，有可能会出现后插入的数据自增 ID 小的情况。此外，TiDB 允许给整型类型的字段指定 AUTO_INCREMENT，且一个表只允许一个属性为 AUTO_INCREMENT 的字段。详情可参考自增 ID和AUTO_INCREMENT。

15.3.11 如何在 TiDB 中修改 sql_mode？

TiDB 支持在会话或全局作用域上修改sql_mode 系统变量。

- 对全局作用域变量的修改，设置后将作用于集群中的其它服务器，并且重启后更改依然有效。因此，你无需在每台 TiDB 服务器上都更改 sql_mode 的值。
- 对会话作用域变量的修改，设置后只影响当前会话，重启后更改消失。

15.3.12 用 Sqoop 批量写入 TiDB 数据，虽然配置了 --batch 选项，但还是会遇到 java.sql.BatchUpdateExecption: statement count 5001 exceeds the transaction limitation 的错误，该如何解决？

问题原因：在 Sqoop 中，--batch 是指每个批次提交 100 条 statement，但是默认每个 statement 包含 100 条 SQL 语句，所以此时 100 * 100 = 10000 条 SQL 语句，超出了 TiDB 的事务限制 5000 条。

解决办法：

- 增加选项 -Dsqoop.export.records.per.statement=10，完整的用法如下：

```
sqoop export \
    -Dsqoop.export.records.per.statement=10 \
    --connect jdbc:mysql://mysql.example.com/sqoop \
    --username sqoop ${user} \
    --password ${passwd} \
    --table ${tab_name} \
    --export-dir ${dir} \
    --batch
```

- 也可以选择增大 TiDB 的单个事物语句数量限制，不过此操作会导致内存增加。详情参见SQL 语句的限制。

15.3.13 TiDB 有像 Oracle 那样的 Flashback Query 功能么，DDL 支持么？

有，也支持 DDL。详细参考使用 AS OF TIMESTAMP 语法读取历史数据。

15.3.14 TiDB 中删除数据后会立即释放空间吗？

在 TiDB 中使用 DELETE，TRUNCATE 和 DROP 语句删除数据都不会立即释放空间。对于 TRUNCATE 和 DROP 操作，在达到 TiDB 的 GC (garbage collection) 时间后（默认 10 分钟），TiDB 的 GC 机制会删除数据并释放空间。对于 DELETE 操作，TiDB 的 GC 机制会删除数据，但不会立即释放空间，而是等到后续进行 compaction 时释放空间。

15.3.15 删除数据后查询速度为何会变慢？

删除大量数据后，会有很多无用的 key 存在，影响查询效率。要解决该问题，可以尝试开启Region Merge 功能，具体可参考最佳实践中的删除数据部分。

15.3.16 对数据做删除操作之后，空间回收比较慢，如何处理？

TiDB 采用了多版本并发控制 (MVCC) 机制，当新写入的数据覆盖旧的数据时，旧的数据不会被替换掉，而是与新写入的数据同时保留，并以时间戳来区分版本。为了使并发事务能查看到早期版本的数据，删除数据时 TiDB 不会立即回收空间，而是等待一段时间后再进行垃圾回收 (GC)。要配置历史数据的保留时限，你可以修改系统变量`tidb_gc_life_time`的值（默认值为 `10m0s`）。

15.3.17 SHOW PROCESSLIST 是否显示系统进程号？

TiDB 中的 SHOW PROCESSLIST 与 MySQL 中的 SHOW PROCESSLIST 显示内容基本一致，不会显示系统进程号。而返回结果中的 ID 表示当前的 session ID。其中 TiDB 的 SHOW PROCESSLIST 和 MySQL 的 SHOW PROCESSLIST 区别如下：

- 由于 TiDB 是分布式数据库，TiDB server 实例是无状态的 SQL 解析和执行引擎（详情可参考TiDB 整体架构），用户使用 MySQL 客户端登录的是哪个 TiDB server，SHOW PROCESSLIST 就会显示当前连接的这个 TiDB server 中执行的 session 列表，不是整个集群中运行的全部 session 列表；而 MySQL 是单机数据库，SHOW PROCESSLIST 列出的是当前整个 MySQL 数据库的全部执行 SQL 列表。
- 在查询执行期间，TiDB 中的 `State` 列不会持续更新。由于 TiDB 支持并行查询，每个语句可能同时处于多个状态，因此很难显示为某一种状态。

15.3.18 在 TiDB 中如何控制或改变 SQL 提交的执行优先级？

TiDB 支持改变全局或单个语句的优先级。优先级包括：

- `HIGH_PRIORITY`：该语句为高优先级语句，TiDB 在执行阶段会优先处理这条语句
- `LOW_PRIORITY`：该语句为低优先级语句，TiDB 在执行阶段会降低这条语句的优先级
- `DELAYED`：该语句为正常优先级语句，TiDB 不强制改变这条语句的优先级，与 `tidb_force_priority` 设置为 `NO_PRIORITY` 相同

以上两种参数可以结合 TiDB 的 DML 语言进行使用，使用方法举例如下：

1. 通过在数据库中写 SQL 的方式来调整优先级：

```
SELECT HIGH_PRIORITY | LOW_PRIORITY | DELAYED COUNT(*) FROM table_name;
INSERT HIGH_PRIORITY | LOW_PRIORITY | DELAYED INTO table_name insert_values;
DELETE HIGH_PRIORITY | LOW_PRIORITY | DELAYED FROM table_name;
UPDATE HIGH_PRIORITY | LOW_PRIORITY | DELAYED table_reference SET assignment_list WHERE
    ↪ where_condition;
REPLACE HIGH_PRIORITY | LOW_PRIORITY | DELAYED INTO table_name;
```

2. 全表扫会自动调整为低优先级，ANALYZE 也是默认低优先级。

15.3.19 在 TiDB 中 auto analyze 的触发策略是怎样的？

触发策略：如果一张新表达到 1000 条记录，并且在 1 分钟内没有写入，会自动触发。

当表的（修改数/当前总行数）比例大于 tidb_auto_analyze_ratio 的时候，会自动触发 analyze 语句。tidb_auto_analyze_ratio 的默认值为 0.5，即默认开启触发 auto analyze。为了保险起见，在开启 auto analyze 的时候，tidb_auto_analyze_ratio 的最小值为 0.3。但是该变量值不能大于等于 pseudo-estimate-ratio（默认值为 0.8），否则会有一段时间使用 pseudo 统计信息，建议设置值为 0.5。

你可以用系统变量tidb_enable_auto_analyze 关闭 auto analyze。

15.3.20 可以使用 Optimizer Hints 控制优化器行为吗？

在 TiDB 中，你可以用多种方法控制查询优化器的默认行为，包括使用Optimizer Hints 和SQL 执行计划管理 (SPM)。基本用法同 MySQL 中的一致，还包含若干 TiDB 特有的用法，例如：select column_name from table_name use ↪ index（index_name）where where_condition;。

15.3.21 DDL 执行

本节列出了 DDL 语句执行的相关问题。DDL 执行原理的详细说明，参见TiDB 中 DDL 执行原理及最佳实践。

15.3.21.1 各类 DDL 操作的预估耗时是多长？

假设 DDL 操作没有被阻塞，各个 TiDB server 能够正常更新 Schema 版本，DDL Owner 节点正常运行。在此情况下，各类 DDL 操作的预估耗时如下：

DDL 操作类型	预估耗时
Reorg DDL：例如 ADD INDEX、MODIFY COLUMN（Reorg 类型的数据更改）	取决于数据量、系统负载以及 DDL 参数的设置
General DDL（除 Reorg DDL 外的 DDL 类型）：例如 CREATE DATABASE、CREATE TABLE、DROP DATABASE、DROP TABLE、TRUNCATE TABLE、ALTER TABLE ADD、ALTER TABLE DROP、MODIFY COLUMN（只更改元数据）、DROP INDEX	1 秒左右

> **注意：**
>
> 以上为各类操作的预估耗时，请以实际操作耗时为准。

15.3.22 执行 DDL 会慢的可能原因

- 在一个用户会话中，DDL 语句之前有非 auto-commit 的 DML 语句，并且该 DML 语句的提交操作比较慢，会导致 DDL 语句执行慢。即执行 DDL 语句前，会先提交之前没有提交的 DML 语句。
- 多个 DDL 语句一起执行的时候，后面的几个 DDL 语句可能会比较慢，因为可能需要排队等待。排队场景包括：
  - 同一类型 DDL 语句需要排队（例如 CREATE TABLE 和 CREATE DATABASE 都是 General DDL，两个操作同时执行时，需要排队）。自 TiDB v6.2.0 起，支持并行 DDL 语句，但为了避免 DDL 使用过多 TiDB 的计算资源，也有并发度限制，因此会有一定的排队情况。
  - 对同一张表上执行的 DDL 操作存在依赖关系，后面的 DDL 语句需要等待前面的 DDL 操作完成。
- 在集群正常启动后，第一个 DDL 操作的执行时间可能会比较久，可能是因为 DDL 模块在进行 DDL Owner 的选举。
- 终止 TiDB 时，TiDB 不能与 PD 正常通信（包括停电的情况），或者用 `kill -9` 命令终止 TiDB 导致 TiDB 没有及时从 PD 清理注册数据。
- 集群中某个 TiDB 与 PD 或者 TiKV 之间发生通信问题，即 TiDB 不能及时获取最新版本信息。

15.3.22.1 触发 Information schema is changed 错误的原因？

TiDB 在执行 SQL 语句时，会根据隔离级别确定一个对象的 schema 版本来处理该 SQL 语句，而且 TiDB 支持在线异步变更 DDL。那么，在执行 DML 的时候可能有 DDL 语句也在执行，而你需要确保每个 SQL 语句在同一个 schema 上执行。所以当执行 DML 时，如果遇到正在执行中的 DDL 操作，TiDB 可能会报 `Information schema is changed` 的错误。

从 v6.4.0 开始，TiDB 实现了元数据锁机制，可以让 DML 语句的执行和 DDL Schema 变更协同进行，可以避免大部分 `Information schema is changed` 错误的发生。

报错的可能原因如下：

- 原因 1：正在执行的 DML 所涉及的表和集群中正在执行的 DDL 的表有相同的，那么这个 DML 语句就会报此错。可以通过命令 `admin show ddl job` 查看正在执行的 DDL 操作。
- 原因 2：这个 DML 执行时间很久，而这段时间内执行了很多 DDL 语句，导致中间 schema 版本变更次数超过 1024（此为默认值，可以通过 `tidb_max_delta_schema_count` 变量修改）。
- 原因 3：接受 DML 请求的 TiDB 长时间不能加载到 `schema information`（TiDB 与 PD 或 TiKV 之间的网络连接故障等会导致此问题），而这段时间内执行了很多 DDL 语句，导致中间 schema 版本变更次数超过 100。
- 原因 4：TiDB 重启后执行第一个 DDL 操作前，执行 DML 操作，并且在执行过程中遇到了第 1 个 DDL 操作（即在执行第 1 个 DDL 操作前，启动该 DML 对应的事务，且在该 DDL 变更第一个 schema 版本后，提交该 DML 对应的事务），那么这个 DML 会报此错。

以上原因中，只有原因 1 与表有关。原因 1 和原因 2 都不会导致业务问题，相应的 DML 会在失败后重试。对于原因 3，需要检查 TiDB 实例和 PD 及 TiKV 的网络情况。

> 注意：
>
> - 目前 TiDB 未缓存所有的 schema 版本信息。
> - 对于每个 DDL 操作，schema 版本变更的数量与对应 schema state 变更的次数一致。
> - 不同的 DDL 操作版本变更次数不一样。例如，create table 操作会有 1 次 schema 版本变更；add column 操作有 4 次 schema 版本变更。

15.3.22.2 触发 Information schema is out of date 错误的原因？

在 v6.5.0 之前，当执行 DML 时，TiDB 超过一个 DDL lease 时间（默认 45s）没能加载到最新的 schema 就可能会报 Information schema is out of date 的错误。遇到此错的可能原因如下：

- 执行此 DML 的 TiDB 被 kill 后准备退出，且此 DML 对应的事务执行时间超过一个 DDL lease，在事务提交时会报这个错误。
- TiDB 在执行此 DML 时，有一段时间内连不上 PD 或者 TiKV，导致 TiDB 超过一个 DDL lease 时间没有 load schema，或者导致 TiDB 断开与 PD 之间带 keep alive 设置的连接。

15.3.22.3 高并发情况下执行 DDL 时报错的原因？

高并发场景下执行 DDL 语句（比如批量建表）时，极少部分的 DDL 语句可能会由于并发执行时 key 冲突而执行失败。

并发执行 DDL 语句时，建议将 DDL 语句数量保持在 20 以下，否则你需要在应用端重试失败的 DDL 语句。

15.3.22.4 DDL 执行被阻塞的原因

在 TiDB v6.2.0 前，TiDB 按照 DDL 语句类型将 DDL 分配到两个先入先出的队列中，即 Reorg DDL 进入 Reorg 队列中，General DDL 进入 general 队列中。由于先入先出以及同一张表上的 DDL 语句需要串行执行，多个 DDL 语句在执行过程中可能会出现阻塞的问题。

例如对于以下 DDL 语句：

- DDL 1：CREATE INDEX idx on t(a int);
- DDL 2：ALTER TABLE t ADD COLUMN b int;
- DDL 3：CREATE TABLE t1(a int);

由于队列先入先出的限制，DDL 3 需要等待 DDL 2 执行。同时又因为同一张表上的 DDL 语句需要串行执行，DDL 2 需要等待 DDL 1 执行。因此，DDL 3 需要等待 DDL 1 先执行完，即使它们操作在不同的表上。

在 TiDB v6.2.0 及之后的版本中，TiDB DDL 模块采用了并发框架。在并发的框架下，不再有同一个队列先进先出的问题，而是从所有 DDL 任务中选出可以执行的 DDL 来执行。并且对 Reorg worker 的数量进行了扩充，大概为节点 CPU/4，这使得在并发框架中 TiDB 可以同时为多张表建索引。

无论是新集群还是从旧版本升级的集群，在 TiDB v6.2 及以上版本中，TiDB 都会自动使用并发框架，用户无需进行调整。

15.3.22.5 定位 DDL 执行卡住的问题

1. 先排除 DDL 语句通常执行慢的可能原因。
2. 使用以下任一方法找出 DDL owner 节点：
    - 通过 `curl http://{TiDBIP}:10080/info/all` 获取当前集群的 Owner
    - 通过监控 DDL > DDL META OPM 查看某个时间段的 Owner

- 如果 Owner 不存在，尝试手动触发 Owner 选举：`curl -X POST http://{TiDBIP}:10080/ddl/owner/resign`。
- 如果 Owner 存在，导出 Goroutine 堆栈并检查可能卡住的地方。

15.3.23 SQL 优化

15.3.23.1 TiDB 执行计划解读

详细解读理解 TiDB 执行计划。

15.3.23.2 统计信息收集

详细解读统计信息。

15.3.23.3 Count 如何加速？

Count 就是暴力扫表，提高并发度能显著提升扫表速度。如要调整并发度，可以使用 `tidb_distsql_scan_concurrency` ↪ 变量，但调整并发度需要同时考虑 CPU 和 I/O 资源。TiDB 每次执行查询时，都要访问 TiKV。在数据量小的情况下，MySQL 的数据都在内存里，而 TiDB 还需要进行一次网络访问。

加速建议：

- 提升硬件配置，可以参考部署建议。
- 提升并发度，默认是 10，可以尝试提升到 50，但是一般提升幅度在 2-4 倍之间。
- 测试大数据量的 count。
- 调优 TiKV 配置，可以参考性能调优。
- 参考下推计算结果缓存。

15.3.23.4 查看当前 DDL 的进度？

通过 `ADMIN SHOW DDL` 语句查看当前 job 进度。操作如下：

```
ADMIN SHOW DDL;
```

```
*************************** 1. row ***************************
  SCHEMA_VER: 140
       OWNER: 1a1c4174-0fcd-4ba0-add9-12d08c4077dc
RUNNING_JOBS: ID:121, Type:add index, State:running, SchemaState:write reorganization, SchemaID
    ↪ :1, TableID:118, RowCount:77312, ArgLen:0, start time: 2018-12-05 16:26:10.652 +0800 CST,
    ↪  Err:<nil>, ErrCount:0, SnapshotVersion:404749908941733890
     SELF_ID: 1a1c4174-0fcd-4ba0-add9-12d08c4077dc
```

从以上返回结果可知，当前正在处理的是 ADD INDEX 操作。且从 RUNNING_JOBS 列的 RowCount 字段可以知道当前 ADD INDEX 操作已经添加了 77312 行索引。

15.3.23.5 如何查看 DDL job？

可以使用 ADMIN SHOW DDL 语句查看正在运行的 DDL 作业。

- ADMIN SHOW DDL JOBS：用于查看当前 DDL 作业队列中的所有结果（包括正在运行以及等待运行的任务）以及已执行完成的 DDL 作业队列中的最近十条结果。
- ADMIN SHOW DDL JOBS QUERIES 'job_id' [, 'job_id'] ...：用于显示 job_id 对应的 DDL 任务的原始 SQL 语句。此 job_id 只搜索正在执行中的任务以及 DDL 历史作业队列中的最近十条结果。

15.3.23.6 TiDB 是否支持基于 COST 的优化 (CBO)？如果支持，实现到什么程度？

是的，TiDB 基于成本的优化器 (CBO) 对代价模型、统计信息进行持续优化。除此之外，TiDB 还支持 hash join、sort-merge join 等 join 算法。

15.3.23.7 如何确定某张表是否需要做 analyze ？

可以通过 SHOW STATS_HEALTHY 来查看 Healthy 字段，一般该字段值小于等于 60 的表需要做 analyze。

15.3.23.8 SQL 的执行计划展开成了树，ID 的序号有什么规律吗？这棵树的执行顺序会是怎么样的？

ID 没什么规律，只要是唯一就行。不过在生成执行计划时，有一个计数器，生成一个计划 ID 后序号就加 1，执行的顺序和序号无关。整个执行计划是一颗树，执行时从根节点开始，不断地向上返回数据。要理解执行计划，请参考理解 TiDB 执行计划。

15.3.23.9 TiDB 执行计划中，task cop 在一个 root 下，这个是并行的吗？

目前 TiDB 的计算任务隶属于两种不同的 task：cop task 和 root task。cop task 是指被下推到 KV 端分布式执行的计算任务，root task 是指在 TiDB 端单点执行的计算任务。

一般来讲 root task 的输入数据是来自于 cop task 的，但是 root task 在处理数据的时候，TiKV 上的 cop task 也可以同时处理数据，等待 TiDB 的 root task 拉取。所以从这个过程来看，root task 和 cop task 是并行的，同时存在数据上下游关系。

在执行的过程中，某些时间段也可能是并行的，第一个 cop task 在处理 [100, 200] 的数据，第二个 cop task 在处理 [1, 100] 的数据。执行计划的理解，请参考理解 TiDB 执行计划。

15.3.24 数据库优化

15.3.24.1 TiDB 参数及调整

详情参考TiDB 配置参数。

15.3.24.2 如何避免热点问题并实现负载均衡？TiDB 中是否有热分区或热范围问题？

要了解热点问题的场景，请参考常见热点问题。TiDB 的以下特性旨在帮助解决热点问题：

- SHARD_ROW_ID_BITS 属性。设置该属性后，行 ID 会被打散并写入多个 Region，以缓解写入热点问题。
- AUTO_RANDOM 属性，用于解决自增主键带来的热点问题。
- Coprocessor Cache，针对小表的读热点问题。
- Load Base Split，针对因 Region 访问不均衡（例如小表全表扫）而导致的热点问题。
- 缓存表，针对被频繁访问但更新较少的小热点表。

如果你遇到因热点引起的性能问题，可参考处理热点问题。

15.3.24.3 TiKV 性能参数调优

详情参考TiKV 性能参数调优。

15.4 TiDB 安装部署常见问题

本文介绍 TiDB 集群安装部署的常见问题、原因及解决方法。

15.4.1 软硬件要求 FAQ

15.4.1.1 TiDB 支持哪些操作系统？

关于 TiDB 支持的操作系统，参见TiDB 软件和硬件环境建议配置。

15.4.1.2 TiDB 对开发、测试、生产环境的服务器硬件配置有什么要求？

TiDB 支持部署和运行在 Intel x86-64 架构的 64 位通用硬件服务器平台。对于开发、测试、生产环境的服务器硬件配置，参见TiDB 软件和硬件环境建议配置 - 服务器建议配置。

15.4.1.3 两块网卡的目的是？万兆的目的是？

作为一个分布式集群，TiDB 对时间的要求还是比较高的，尤其是 PD 需要分发唯一的时间戳，如果 PD 时间不统一，如果有 PD 切换，将会等待更长的时间。两块网卡可以做 bond，保证数据传输的稳定，万兆可以保证数据传输的速度，千兆网卡容易出现瓶颈，我们强烈建议使用万兆网卡。

15.4.1.4 SSD 不做 RAID 是否可行？

资源可接受的话，我们建议做 RAID 10，如果资源有限，也可以不做 RAID。

15.4.1.5 TiDB 集群各个组件的配置推荐？

- TiDB 需要 CPU 和内存比较好的机器，参考官网配置要求，如果后期需要开启 TiDB Binlog，根据业务量的评估和 GC 时间的要求，也需要本地磁盘大一点，不要求 SSD 磁盘；
- PD 里面存了集群元信息，会有频繁的读写请求，对磁盘 I/O 要求相对比较高，磁盘太差会影响整个集群性能，推荐 SSD 磁盘，空间不用太大。另外集群 Region 数量越多对 CPU、内存的要求越高；
- TiKV 对 CPU、内存、磁盘要求都比较高，一定要用 SSD 磁盘。

详情可参考TiDB 软硬件环境需求。

15.4.2 安装部署 FAQ

如果用于生产环境，推荐使用 TiUP 部署 TiDB 集群。

15.4.2.1 为什么修改了 TiKV/PD 的 toml 配置文件，却没有生效？

这种情况一般是因为没有使用 `--config` 参数来指定配置文件（目前只会出现在 binary 部署的场景），TiKV/PD 会按默认值来设置。如果要使用配置文件，请设置 TiKV/PD 的 `--config` 参数。对于 TiKV 组件，修改配置后重启服务即可；对于 PD 组件，只会在第一次启动时读取配置文件，之后可以使用 pd-ctl 的方式来修改配置，详情可参考PD 配置参数。

15.4.2.2 TiDB 监控框架 Prometheus + Grafana 监控机器建议单独还是多台部署？

监控机建议单独部署。建议 CPU 8 core，内存 16 GB 以上，硬盘 500 GB 以上。

15.4.2.3 有一部分监控信息显示不出来？

查看访问监控的机器时间跟集群内机器的时间差，如果比较大，更正时间后即可显示正常。

15.4.2.4 supervise/svc/svstat 服务具体起什么作用？

- supervise 守护进程
- svc 启停服务
- svstat 查看进程状态

15.4.2.5 inventory.ini 变量参数解读

变量	含义
cluster_name	集群名称，可调整
tidb_version	TiDB 版本

变量	含义
deployment_method	部署方式，默认为 binary，可选 docker
process_supervision	进程监管方式，默认为 systemd，可选 supervise
timezone	修改部署目标机器时区，默认为 Asia/Shanghai，可调整，与 set_timezone 变量结合使用

变量	含义
set_timezone	默认为 True，即修改部署目标机器时区，关闭可修改为 False
enable_elk	目前不支持，请忽略
enable_firewalld	开启防火墙，默认不开启
enable_ntpd	检测部署目标机器 NTP 服务，默认为 True，请勿关闭

变量	含义
machine_benchmark	检测部署目标机器磁盘 IOPS，默认为 True，请勿关闭
set_hostname_as_ip	根据 IP 修改部署目标机器主机名，默认为 False
enable_binlog	是否在部署 pump 并开启 bin-log，默认为 False，依赖 Kafka 集群，参见 zookeeper_addrs 变量
zookeeper_addrs	Kafka 集群的 ZooKeeper 地址

变量	含义
enable_slow_query_log	TiDB 慢查询日志记录到单独文件 ({{ deploy_dir }}/log/tidb_slow_query.log)，默认为 False，记录到 tidb 日志
deploy_without_tidb	KV 模式，不部署 TiDB 服务，仅部署 PD、TiKV 及监控服务，请将 inventory.ini 文件中 tidb_servers 主机组 IP 设置为空。

15.4.2.6 如何单独记录 TiDB 中的慢查询日志，如何定位慢查询 SQL？

1. TiDB 中，对慢查询的定义在 TiDB 的配置文件中。`slow-threshold: 300`，这个参数是配置慢查询记录阈值的，单位是 ms。
2. 如果出现了慢查询，可以从 Grafana 监控定位到出现慢查询的 tidb-server 以及时间点，然后在对应节点查找日志中记录的 SQL 信息。
3. 除了日志，还可以通过 `ADMIN SHOW SLOW` 命令查看，详情可参考ADMIN SHOW SLOW 命令。

15.4.2.7 首次部署 TiDB 集群时，没有配置 tikv 的 Label 信息，在后续如何添加配置 Label？

TiDB 的 Label 设置是与集群的部署架构相关的，是集群部署中的重要内容，是 PD 进行全局管理和调度的依据。如果集群在初期部署过程中没有设置 Label，需要在后期对部署结构进行调整，就需要手动通过 PD 的管理工具 pd-ctl 来添加 location-labels 信息，例如：`config set location-labels "zone,rack,host"`（根据实际的 label 层级名字配置）。

pd-ctl 的使用参考PD Control 使用说明。

15.4.2.8 为什么测试磁盘的 dd 命令用 `oflag=direct` 这个选项？

Direct 模式就是把写入请求直接封装成 I/O 指令发到磁盘，这样是为了绕开文件系统的缓存，可以直接测试磁盘的真实的 I/O 读写能力。

15.4.2.9 如何用 fio 命令测试 TiKV 实例的磁盘性能？

- 随机读测试：

```
./fio -ioengine=psync -bs=32k -fdatasync=1 -thread -rw=randread -size=10G -filename=
    ↪ fio_randread_test.txt -name='fio randread test' -iodepth=4 -runtime=60 -numjobs=4 -
    ↪ group_reporting --output-format=json --output=fio_randread_result.json
```

- 顺序写和随机读混合测试：

```
./fio -ioengine=psync -bs=32k -fdatasync=1 -thread -rw=randrw -percentage_random=100,0 -size
    ↪ =10G -filename=fio_randread_write_test.txt -name='fio mixed randread and sequential
    ↪ write test' -iodepth=4 -runtime=60 -numjobs=4 -group_reporting --output-format=json
    ↪ --output=fio_randread_write_test.json
```

15.4.3 TiDB 支持在公有云上部署吗？

TiDB 支持在 Google GKE、AWS EKS 和阿里云 ACK 上部署使用。

此外，TiDB 云上部署也已在京东云、UCloud 上线。

15.5 迁移常见问题

本文介绍 TiDB 数据迁移中的常见问题。

如果要查看迁移相关工具的常见问题，请参考以下链接：

- 备份与恢复常见问题
- TiDB Binlog 常见问题
- TiDB Lightning 常见问题
- Data Migration 常见问题
- TiCDC 常见问题

15.5.1 全量数据导出导入

15.5.1.1 如何将一个运行在 MySQL 上的应用迁移到 TiDB 上？

TiDB 支持绝大多数 MySQL 语法，一般不需要修改代码。

15.5.1.2 导入导出速度慢，各组件日志中出现大量重试、EOF 错误并且没有其他错误

在没有其他逻辑出错的情况下，重试、EOF 可能是由网络问题引起的，建议首先使用相关工具排查网络连通状况。以下示例使用 iperf 进行排查：

- 在出现重试、EOF 错误的服务器端节点执行以下命令：

```
iperf3 -s
```

- 在出现重试、EOF 错误的客户端节点执行以下命令：

```
iperf3 -c <server-IP>
```

下面是一个网络连接良好的客户端节点的输出：

```
$ iperf3 -c 192.168.196.58
Connecting to host 192.168.196.58, port 5201
[  5] local 192.168.196.150 port 55397 connected to 192.168.196.58 port 5201
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-1.00   sec  18.0 MBytes   150 Mbits/sec
[  5]   1.00-2.00   sec  20.8 MBytes   175 Mbits/sec
[  5]   2.00-3.00   sec  18.2 MBytes   153 Mbits/sec
[  5]   3.00-4.00   sec  22.5 MBytes   188 Mbits/sec
[  5]   4.00-5.00   sec  22.4 MBytes   188 Mbits/sec
[  5]   5.00-6.00   sec  22.8 MBytes   191 Mbits/sec
[  5]   6.00-7.00   sec  20.8 MBytes   174 Mbits/sec
[  5]   7.00-8.00   sec  20.1 MBytes   168 Mbits/sec
[  5]   8.00-9.00   sec  20.8 MBytes   175 Mbits/sec
[  5]   9.00-10.00  sec  21.8 MBytes   183 Mbits/sec
```

```
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-10.00  sec   208 MBytes   175 Mbits/sec                  sender
[  5]   0.00-10.00  sec   208 MBytes   174 Mbits/sec                  receiver

iperf Done.
```

如果输出显示网络带宽较低、带宽波动大，各组件日志中就可能出现大量重试、EOF 错误。此时你需要咨询网络服务供应商以提升网络质量。

如果输出的各指标良好，请尝试更新各组件版本。如果更新后仍无法解决问题，请移步 AskTUG 论坛寻求帮助。

15.5.1.3 不小心把 MySQL 的 user 表导入到 TiDB 了，或者忘记密码，无法登录，如何处理？

重启 TiDB 服务，配置文件中增加 `-skip-grant-table=true` 参数，无密码登录集群后，可以根据情况重建用户，或者重建 mysql.user 表，具体表结构搜索官网。

15.5.1.4 如何导出 TiDB 数据？

你可以通过以下方式导出 TiDB 数据：

- 参考 MySQL 使用 mysqldump 导出某个表的部分数据，使用 mysqldump 加 where 条件导出。
- 使用 MySQL client 将 select 的结果输出到一个文件。

15.5.1.5 如何从 DB2、Oracle 数据库迁移到 TiDB？

DB2、Oracle 到 TiDB 数据迁移（增量 + 全量），通常做法有：

- 使用 Oracle 官方迁移工具，如 OGG、Gateway（透明网关）、CDC (Change Data Capture)。
- 自研数据导出导入程序实现。
- 导出 (Spool) 成文本文件，然后通过 Load infile 进行导入。
- 使用第三方数据迁移工具。

目前看来 OGG 最为合适。

15.5.1.6 用 Sqoop 批量写入 TiDB 数据，虽然配置了 `--batch` 选项，但还是会遇到 `java.sql.BatchUpdateExecption:statement count 5001 exceeds the transaction limitation` 的错误，该如何解决？

- 在 Sqoop 中，`--batch` 是指每个批次提交 100 条 statement，但是默认每个 statement 包含 100 条 SQL 语句，所以此时 100 * 100 = 10000 条 SQL 语句，超出了 TiDB 的事务限制 5000 条，可以增加选项 `-Dsqoop.export.records.per.statement=10` 来解决这个问题，完整的用法如下：

```
sqoop export \
    -Dsqoop.export.records.per.statement=10 \
    --connect jdbc:mysql://mysql.example.com/sqoop \
    --username sqoop ${user} \
```

```
    --password ${passwd} \
    --table ${tab_name} \
    --export-dir ${dir} \
    --batch
```

- 也可以选择增大 tidb 的单个事物语句数量限制，不过这个会导致内存上涨。

15.5.1.7 Dumpling 导出时引发上游数据库 OOM 或报错 “磁盘空间不足”

该问题可能有如下原因：

- 数据库主键分布不均匀，例如启用了SHARD_ROW_ID_BITS
- 上游数据库为 TiDB，导出表是分区表

在上述情况下，Dumpling 划分导出子范围时，会划分出过大的子范围，从而向上游发送结果过大的查询。请联系 AskTUG 社区专家获取实验版本的 Dumpling。

15.5.1.8 TiDB 有像 Oracle 那样的 Flashback Query 功能么，DDL 支持么？

有，也支持 DDL。详细参考TiDB 历史数据回溯。

15.5.2 在线数据同步

15.5.2.1 有没有现成的同步方案，可以将数据同步到 Hbase、 Elasticsearh 等其他存储？

没有，目前依赖程序自行实现。

15.5.3 业务流量迁入

15.5.3.1 如何快速迁移业务流量？

我们建议通过TiDB Data Migration 进行 MySQL -> TiDB 的业务数据的迁移；业务读写可以按照需求分阶段通过修改网络配置进行流量迁移，建议 DB 上层部署一个稳定的网络 LB（ HAproxy、 LVS、 F5、 DNS 等 ），这样直接修改网络配置就能实现无缝流量迁移。

15.5.3.2 TiDB 总读写流量有限制吗？

TiDB 读流量可以通过增加 TiDB server 进行扩展，总读容量无限制，写流量可以通过增加 TiKV 节点进行扩容，基本上写容量也没有限制。

15.5.3.3 Transaction too large 是什么原因，怎么解决？

TiDB 限制了单条 KV entry 不超过 6MB，可以修改配置文件中的`txn-entry-size-limit` 配置项进行调整，最大可以修改到 120MB。

分布式事务要做两阶段提交，而且底层还需要做 Raft 复制。如果一个事务非常大，提交过程会非常慢，事务写冲突概率会增加，而且事务失败后回滚会导致不必要的性能开销。所以我们设置了 key-value entry 的总大小

默认不超过 100MB。如果业务需要使用大事务，可以修改配置文件中的 `txn-total-size-limit` 配置项进行调整，最大可以修改到 10G。实际的大小限制还受机器的物理内存影响。

在 Google 的 Cloud Spanner 上面，也有类似的限制。

15.5.3.4 如何批量导入？

导入数据的时候，可以分批插入，每批最好不要超过 1w 行。

15.5.3.5 TiDB 中删除数据后会立即释放空间吗？

DELETE，TRUNCATE 和 DROP 都不会立即释放空间。对于 TRUNCATE 和 DROP 操作，在达到 TiDB 的 GC (garbage collection) 时间后（默认 10 分钟），TiDB 的 GC 机制会删除数据并释放空间。对于 DELETE 操作 TiDB 的 GC 机制会删除数据，但不会释放空间，而是当后续数据写入 RocksDB 且进行 compact 时对空间重新利用。

15.5.3.6 Load 数据时可以对目标表执行 DDL 操作吗？

不可以，加载数据期间不能对目标表执行任何 DDL 操作，这会导致数据加载失败。

15.5.3.7 TiDB 是否支持 replace into 语法？

支持。

15.5.3.8 删除数据后查询速度为何会变慢？

删除大量数据后，会有很多无用的 key 存在，影响查询效率。要解决该问题，可以尝试开启Region Merge 功能，具体可参考最佳实践中的删除数据部分。

15.5.3.9 数据删除最高效最快的方式？

在删除大量数据的时候，建议使用 `Delete from t where xx limit 5000`（xx 建议在满足业务过滤逻辑下，尽量加上强过滤索引列或者直接使用主键选定范围，如 `id >= 5000*n+m and id <= 5000*(n+1)+m` 这样的方案，通过循环来删除，用 `Affected Rows == 0` 作为循环结束条件，这样避免遇到事务大小的限制。如果一次删除的数据量非常大，这种循环的方式会越来越慢，因为每次删除都是从前向后遍历，前面的删除之后，短时间内会残留不少删除标记（后续会被 GC 掉），影响后面的 Delete 语句。如果有可能，建议把 Where 条件细化。可以参考官网最佳实践。

15.5.3.10 TiDB 如何提高数据加载速度？

主要有两个方面：

- 目前已开发分布式导入工具TiDB Lightning，需要注意的是数据导入过程中为了性能考虑，不会执行完整的事务流程，所以没办法保证导入过程中正在导入的数据的 ACID 约束，只能保证整个导入过程结束以后导入数据的 ACID 约束。因此适用场景主要为新数据的导入（比如新的表或者新的索引），或者是全量的备份恢复（先 Truncate 原表再导入）。
- TiDB 的数据加载与磁盘以及整体集群状态相关，加载数据时应关注该主机的磁盘利用率，TiClient Error/Backoff/Thread CPU 等相关 metric，可以分析相应瓶颈。

15.6 升级与升级后常见问题

本文介绍 TiDB 升级与升级后的常见问题与解决办法。

15.6.1 升级常见问题

本小节列出了 TiDB 升级相关的常见问题与解决办法。

15.6.1.1 滚动升级有那些影响？

滚动升级 TiDB 期间，业务运行会受到一定影响。因此，不建议在业务高峰期进行滚动升级。需要配置最小集群拓扑 (TiDB * 2、PD * 3、TiKV * 3)，如果集群环境中有 Pump 和 Drainer 服务，建议先停止 Drainer，然后滚动升级（升级 TiDB 时会升级 Pump）。

15.6.1.2 集群在执行 DDL 请求期间可以进行升级操作吗？

集群中有 DDL 语句正在被执行时（通常为 ADD INDEX 和列类型变更等耗时较久的 DDL 语句），请勿进行升级操作。在升级前，建议使用`ADMIN SHOW DDL` 命令查看集群中是否有正在进行的 DDL Job。如需升级，请等待 DDL 执行完成或使用`ADMIN CANCEL DDL` 命令取消该 DDL Job 后再进行升级。

另外，在升级 TiDB 集群的过程中，请勿执行 DDL 语句，否则可能会出现行为未定义的问题。

15.6.1.3 Binary 如何升级？

不推荐使用 Binary 来升级集群。建议使用 TiUP 升级 TiDB 进行升级，确保分布式系统版本一致性和兼容性。

15.6.2 升级后常见问题

本小节列出了一些升级后可能会遇到的问题与解决办法。

15.6.2.1 执行 DDL 操作时遇到的字符集 (charset) 问题

TiDB 在 v2.1.0 以及之前版本（包括 v2.0 所有版本）中，默认字符集是 UTF8。从 v2.1.1 开始，默认字符集变更为 UTF8MB4。如果在 v2.1.0 及之前版本中，建表时显式指定了 table 的 charset 为 UTF8，那么升级到 v2.1.1 之后，执行 DDL 操作可能会失败。

要避免该问题，需注意以下两个要点：

- 在 v2.1.3 之前，TiDB 不支持修改 column 的 charset。所以，执行 DDL 操作时，新 column 的 charset 需要和旧 column 的 charset 保持一致。
- 在 v2.1.3 之前，即使 column 的 charset 和 table 的 charset 不一样，`show create table` 也不会显示 column 的 charset，但可以通过 HTTP API 获取 table 的元信息来查看 column 的 charset，下文提供了示例。

15.6.2.1.1 unsupported modify column charset utf8mb4 not match origin utf8

- 升级前：v2.1.0 及之前版本

```
create table t(a varchar(10)) charset=utf8;
```

```
Query OK, 0 rows affected
Time: 0.106s
```

```
show create table t
```

```
+-------+-------------------------------------------------------+
| Table | Create Table                                          |
+-------+-------------------------------------------------------+
| t     | CREATE TABLE `t` (                                    |
|       |   `a` varchar(10) DEFAULT NULL                        |
|       | ) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_bin |
+-------+-------------------------------------------------------+
1 row in set
Time: 0.006s
```

- 升级后：v2.1.1、v2.1.2 会出现下面的问题，v2.1.3 以及之后版本不会出现下面的问题。

```
alter table t change column a a varchar(20);
```

```
ERROR 1105 (HY000): unsupported modify column charset utf8mb4 not match origin utf8
```

解决方案：显式指定 column charset，保持和原来的 charset 一致即可。

```
alter table t change column a a varchar(22) character set utf8;
```

- 根据要点 1，此处如果不指定 column 的 charset，会用默认的 UTF8MB4，所以需要指定 column charset 保持和原来一致。
- 根据要点 2，用 HTTP API 获取 table 元信息，然后根据 column 名字和 Charset 关键字搜索即可找到 column 的 charset。

```
curl "http://$IP:10080/schema/test/t" | python -m json.tool
```

这里用了 python 的格式化 json 的工具，也可以不加，此处只是为了方便注释。

```
{
    "ShardRowIDBits": 0,
    "auto_inc_id": 0,
    "charset": "utf8",   # table 的 charset
    "collate": "",
```

```
    "cols": [            # 从这里开始列举 column 的相关信息
        {
            ...
            "id": 1,
            "name": {
                "L": "a",
                "O": "a"   # column 的名字
            },
            "offset": 0,
            "origin_default": null,
            "state": 5,
            "type": {
                "Charset": "utf8",   # column a 的 charset
                "Collate": "utf8_bin",
                "Decimal": 0,
                "Elems": null,
                "Flag": 0,
                "Flen": 10,
                "Tp": 15
            }
        }
    ],
    ...
}
```

15.6.2.1.2 unsupported modify charset from utf8mb4 to utf8

- 升级前：v2.1.1，v2.1.2

```
create table t(a varchar(10)) charset=utf8;
```

```
Query OK, 0 rows affected
Time: 0.109s
```

```
show create table t;
```

```
+-------+-------------------------------------------------------+
| Table | Create Table                                          |
+-------+-------------------------------------------------------+
| t     | CREATE TABLE `t` (                                    |
|       |   `a` varchar(10) DEFAULT NULL                        |
|       | ) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_bin |
+-------+-------------------------------------------------------+
```

上面 show create table 只显示出了 table 的 charset，但其实 column 的 charset 是 UTF8MB4，这可以通过 HTTP API 获取 schema 来确认。这是一个 bug，即此处建表时 column 的 charset 应该要和 table 保持一致为 UTF8，该问题在 v2.1.3 中已经修复。

- 升级后：v2.1.3 及之后版本

```
show create table t;
```

```
+-------+-------------------------------------------------------------------+
| Table | Create Table                                                      |
+-------+-------------------------------------------------------------------+
| t     | CREATE TABLE `t` (                                                |
|       |   `a` varchar(10) CHARSET utf8mb4 COLLATE utf8mb4_bin DEFAULT NULL |
|       | ) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_bin             |
+-------+-------------------------------------------------------------------+
1 row in set
Time: 0.007s
```

```
alter table t change column a a varchar(20);
```

```
ERROR 1105 (HY000): unsupported modify charset from utf8mb4 to utf8
```

解决方案：

- 因为在 v2.1.3 之后，TiDB 支持修改 column 和 table 的 charset，所以这里推荐修改 table 的 charset 为 UTF8MB4。

```
alter table t convert to character set utf8mb4;
```

- 也可以像问题 1 一样指定 column 的 charset，保持和 column 原来的 charset (UTF8MB4) 一致即可。

```
alter table t change column a a varchar(20) character set utf8mb4;
```

15.6.2.1.3 ERROR 1366 (HY000): incorrect utf8 value f09f8c80( ) for column a

TiDB 在 v2.1.1 及之前版本中，如果 charset 是 UTF8，没有对 4-byte 的插入数据进行 UTF8 Unicode encoding 检查。在 v2.1.2 及之后版本中，添加了该检查。

- 升级前：v2.1.1 及之前版本

```
create table t(a varchar(100) charset utf8);
```

```
Query OK, 0 rows affected
```

```
insert t values (unhex('f09f8c80'));
```

```
Query OK, 1 row affected
```

- 升级后：v2.1.2 及之后版本

```
insert t values (unhex('f09f8c80'));
```

```
ERROR 1366 (HY000): incorrect utf8 value f09f8c80( ) for column a
```

解决方案：

- v2.1.2 版本：该版本不支持修改 column charset，所以只能跳过 UTF8 的检查。

```
set @@session.tidb_skip_utf8_check=1;
```

```
Query OK, 0 rows affected
```

```
insert t values (unhex('f09f8c80'));
```

```
Query OK, 1 row affected
```

- v2.1.3 及之后版本：建议修改 column 的 charset 为 UTF8MB4。或者也可以设置 tidb_skip_utf8_check 变量跳过 UTF8 的检查。如果跳过 UTF8 的检查，在需要将数据从 TiDB 同步回 MySQL 的时候，可能会失败，因为 MySQL 会执行该检查。

```
alter table t change column a a varchar(100) character set utf8mb4;
```

```
Query OK, 0 rows affected
```

```
insert t values (unhex('f09f8c80'));
```

```
Query OK, 1 row affected
```

关于 tidb_skip_utf8_check 变量，具体来说是指跳过 UTF8 和 UTF8MB4 类型对数据的合法性检查。如果跳过这个检查，在需要将数据从 TiDB 同步回 MySQL 的时候，可能会失败，因为 MySQL 执行该检查。如果只想跳过 UTF8 类型的检查，可以设置 tidb_check_mb4_value_in_utf8 变量。

tidb_check_mb4_value_in_utf8 在 v2.1.3 版本加入 config.toml 文件，可以修改配置文件里面的 check-mb4-value-in-utf8 后重启集群生效。

tidb_check_mb4_value_in_utf8 在 v2.1.5 版本开始可以用 HTTP API 来设置，也可以用 session 变量来设置。

  - HTTP API（HTTP API 只在单台服务器上生效）
    * 执行下列命令启用 HTTP API：

      ```
      curl -X POST -d "check_mb4_value_in_utf8=1" http://{TiDBIP}:10080/settings
      ```

    * 执行下列命令禁用 HTTP API：

      ```
      curl -X POST -d "check_mb4_value_in_utf8=0" http://{TiDBIP}:10080/settings
      ```

- Session 变量
  * 执行下列命令启用 Session 变量：

```
set @@session.tidb_check_mb4_value_in_utf8 = 1;
```

  * 执行下列命令禁用 Session 变量：

```
set @@session.tidb_check_mb4_value_in_utf8 = 0;
```

- v2.1.7 及之后版本，如果对表和 column 的字符集没有严格要求为 UTF8，也不想修改客户端代码去跳过 UTF8 检查或者手动修改 column 的 charset，可以在配置文件中把 `treat-old-version-utf8-as-utf8mb4` 打开。该配置的作用是自动把 v2.1.7 版本之前创建的旧版本的表和 column 的 UTF8 字符集转成 UTF8MB4。这个转换是在 TiDB load schema 时在内存中将 UTF8 转成 UTF8MB4，不会对实际存储的数据做任何修改。在配置文件中关闭 `treat-old-version-utf8-as-utf8mb4` 并重启 TiDB 后，以前字符集为 UTF8 的表和 column 的字符集仍然还是 UTF8。

> **注意：**
>
> `treat-old-version-utf8-as-utf8mb4` 参数默认打开，如果客户端强制需要用 UTF8 而不用 UTF8MB4，需要在配置文件中关闭。

15.7 TiDB 监控常见问题

本文介绍在监控 TiDB 集群时的常见问题、原因及解决方法。

- Prometheus 监控框架详情可见TiDB 监控框架概述。
- 监控指标解读详细参考重要监控指标详解。

15.7.1 目前的监控使用方式及主要监控指标，有没有更好看的监控？

TiDB 使用 Prometheus + Grafana 组成 TiDB 数据库系统的监控系统。用户在 Grafana 上通过 dashboard 可以监控到 TiDB 的各类运行指标，包括

- 系统资源的监控指标
- 客户端连接与 SQL 运行的指标
- 内部通信和 Region 调度的指标

通过这些指标，可以让数据库管理员更好的了解到系统的运行状态，运行瓶颈等内容。在监控指标的过程中，我们按照 TiDB 不同的模块，分别列出了各个模块重要的指标项，一般用户只需要关注这些常见的指标项。具体指标请参见官方文档。

15.7.2 Prometheus 监控数据默认 15 天自动清除一次，可以自己设定成 2 个月或者手动删除吗？

可以的，在 Prometheus 启动的机器上，找到启动脚本，然后修改启动参数，然后重启 Prometheus 生效。

```
--storage.tsdb.retention="60d"
```

15.7.3 Region Health 监控项

TiDB-2.0 版本中，PD metric 监控页面中，对 Region 健康度进行了监控，其中 Region Health 监控项是对所有 Region 副本状况的一些统计。其中 miss 是缺副本，extra 是多副本。同时也增加了按 Label 统计的隔离级别，level-1 表示这些 Region 的副本在第一级 Label 下是物理隔离的，没有配置 location label 时所有 Region 都在 level-0。

15.7.4 Statement Count 监控项中的 selectsimplefull 是什么意思？

代表全表扫，但是可能是很小的系统表。

15.7.5 监控上的 QPS 和 Statement OPS 有什么区别？

QPS 会统计执行的所有 SQL 命令，包括 use database、 load data、 begin、 commit、 set、 show、 insert、 select 等。

Statement OPS 只统计 select、 update、 insert 等业务相关的，所以 Statement OPS 的统计和业务比较相符。

15.8 TiDB 集群管理常见问题

本文介绍管理 TiDB 集群时的常见问题、原因及解决方法。

15.8.1 集群日常管理

本小节介绍集群日程管理中的常见问题、原因及解决方法。

15.8.1.1 TiDB 如何登录？

和 MySQL 登录方式一样，可以按照下面例子进行登录。

```
mysql -h 127.0.0.1 -uroot -P4000
```

15.8.1.2 TiDB 如何修改数据库系统变量？

和 MySQL 一样，TiDB 也分为静态参数和固态参数，静态参数可以直接通过 `SET GLOBAL xxx = n` 的方式进行修改，不过新参数值只限于该实例生命周期有效。

15.8.1.3 TiDB (TiKV) 有哪些数据目录？

默认在`--data-dir` 目录下，其中包括 backup、db、raft、snap 四个目录，分别存储备份、数据、raft 数据及镜像数据。

15.8.1.4 TiDB 有哪些系统表？

和 MySQL 类似，TiDB 中也有系统表，用于存放数据库运行时所需信息，具体信息参考TiDB 系统表文档。

15.8.1.5 TiDB 各节点服务器下是否有日志文件，如何管理？

默认情况下各节点服务器会在日志中输出标准错误，如果启动的时候通过 `--log-file` 参数指定了日志文件，那么日志会输出到指定的文件中，并且按天做 rotation。

15.8.1.6 TiDB、TiKV、PD 节点的各类文件存放在哪里？

如需快速了解 TiDB 节点、TiKV 节点、PD 节点的配置文件、数据文件及日志文件的相关介绍与其存放位置，建议观看下面的培训视频（时长 9 分钟）。

15.8.1.7 如何规范停止 TiDB？

- 若使用了 load balancer（推荐）：先停止 load balancer，然后执行 SHUTDOWN 语句。此时 TiDB 会根据graceful-wait-before-shutdown 设置值等待所有会话断开，然后停止运行。
- 若未使用 load balancer：执行 SHUTDOWN 语句，TiDB 组件会做 graceful shutdown。

15.8.1.8 TiDB 里面可以执行 kill 命令吗？

- Kill DML 语句：

  查询 `information_schema.cluster_processlist`，获取正在执行 DML 语句的 TiDB 实例地址和 session ID，然后执行 kill 命令。

  TiDB 从 v6.1.0 起新增 Global Kill 功能（由enable-global-kill 配置项控制，默认启用）。启用 Global Kill 功能时，直接执行 `kill session_id` 即可。

  对于 TiDB v6.1.0 之前的版本，或未启用 Global Kill 功能时，`kill session_id` 默认不生效，客户端需要连接到正在执行 DML 语句的 TiDB 实例，然后执行 `kill tidb session_id` 才能 kill DML 语句。如果客户端连接到其他 TiDB 实例或者客户端和 TiDB 集群之间有代理，`kill tidb session_id` 可能会被路由到其他的 TiDB 实例，从而错误地终止其他会话。具体可以参考KILL。

- Kill DDL 语句：执行 `admin show ddl jobs`，查找需要 kill 的 DDL job ID，然后执行 `admin cancel ddl jobs 'job_id' [, 'job_id'] ...`。具体可以参考ADMIN。

15.8.1.9 TiDB 是否支持会话超时？

TiDB 目前支持wait_timeout 和interactive_timeout 两种超时。

15.8.1.10 TiDB 的版本管理策略是怎么样的？

关于 TiDB 版本的管理策略，可以参考TiDB 版本规则。

15.8.1.11 部署和维护 TiDB 集群的运营成本如何？

TiDB 提供了一些特性和工具，可以帮助你以低成本管理集群：

- 在运维方面，TiUP 作为包管理器，简化了部署、扩缩容、升级和其他运维任务。

- 在监控方面，TiDB 监控框架使用 Prometheus 存储监控和性能指标，使用 Grafana 可视化这些指标。Grafana 内置了数十个面板，覆盖了数百个指标。
- 在故障诊断方面，TiDB 集群问题导图汇总了 TiDB 服务器和其他组件的常见问题。你可以使用这个导图来诊断和解决遇到的相关问题。

15.8.1.12 分不清 TiDB master 版本之间的区别，应该怎么办？

TiDB 目前社区非常活跃，同时，我们还在不断的优化和修改 BUG，因此 TiDB 的版本更新周期比较快，会不定期有新版本发布，请关注我们的版本发布时间线。此外 TiDB 安装推荐使用 TiUP 进行安装或使用 TiDB Operator 进行安装。TiDB 的版本号目前实现了统一管理，你可以通过如下任意方式查看 TiDB 的版本号：

- 通过 `select tidb_version()` 进行查看
- 通过执行 `tidb-server -V` 进行查看

15.8.1.13 有没有图形化部署 TiDB 的工具？

有。你可以使用TiUniManager，它是一款为分布式数据库 TiDB 打造的管控平台软件和数据库运维管理平台，为 TiDB 提供数据库集群管理功能、主机管理功能和平台管理功能，涵盖了数据库运维人员 (DBA) 在 TiDB 上进行的常用运维操作，帮助 DBA 对 TiDB 进行自动化、自助化和可视化管理。

15.8.1.14 如何扩容 TiDB 集群？

可以在不影响线上服务的情况下，对 TiDB 集群进行扩容。

- 如果是使用TiUP 部署的集群，可以参考使用 TiUP 扩容 TiDB 集群。
- 如果是使用TiDB Operator 在 Kubernetes 上部署的集群，可以参考在 Kubernetes 中手动扩容 TiDB 集群。

15.8.1.15 TiDB 如何进行水平扩展？

当您的业务不断增长时，数据库可能会面临三方面瓶颈，第一是存储空间，第二是计算资源，第三是读写容量，这时可以对 TiDB 集群做水平扩展。

- 如果是存储资源不够，可以通过添加 TiKV Server 节点来解决，新节点启动后，PD 会自动将其他节点的部分数据迁移过去，无需人工介入。
- 如果是计算资源不够，可以查看 TiDB Server 和 TiKV Server 节点的 CPU 消耗情况，再考虑添加 TiDB Server 节点或者是 TiKV Server 节点来解决，如添加 TiDB Server 节点，将其添加到前端 Load Balancer 配置之中即可。
- 如果是容量跟不上，一般可以考虑同时增加 TiDB Server 和 TiKV Server 节点。

15.8.1.16 Percolator 用了分布式锁，crash 的客户端会保持锁，会造成锁没有 release？

详细可参考 Percolator 和 TiDB 事务算法。

15.8.1.17 TiDB 为什么选用 gRPC 而不选用 Thrift，是因为 Google 在用吗？

不只是因为 Google 在用，有一些比较好的特性我们需要，比如流控、加密还有 Streaming。

15.8.1.18 like(bindo.customers.name, jason%, 92) 这个 92 代表什么？

那个是转义字符，默认是 (ASCII 92)。

15.8.1.19 为什么 information_schema.tables.data_length 记录的大小和 TiKV 监控面板上的 store size 不一样？

这是因为两者计算的角度不一样。information_schema.tables.data_length 是通过统计信息（平均每行的大小）得到的估算值。TiKV 监控面板上的 store size 是单个 TiKV 实例的数据文件（RocksDB 的 SST 文件）的大小总和。由于多版本和 TiKV 会压缩数据，所以两者显示的大小不一样。

15.8.1.20 为什么事务没有使用异步提交或一阶段提交？

在以下情况中，即使通过系统变量开启了异步提交和一阶段提交，TiDB 也不会使用这些特性：

- 如果开启了 TiDB Binlog，受 TiDB Binlog 的实现原理限制，TiDB 不会使用异步提交或一阶段提交特性。
- TiDB 只在事务写入不超过 256 个键值对，以及所有键值对里键的总大小不超过 4 KB 时，才会使用异步提交或一阶段提交特性。这是因为对于写入量大的事务，异步提交不能明显提升执行性能。

15.8.2 PD 管理

本小节介绍 PD 管理中的常见问题、原因及解决方法。

15.8.2.1 访问 PD 报错：TiKV cluster is not bootstrapped

PD 的大部分 API 需要在初始化 TiKV 集群以后才能使用，如果在部署新集群的时候只启动了 PD，还没有启动 TiKV，这时候访问 PD 就会报这个错误。遇到这个错误应该先把要部署的 TiKV 启动起来，TiKV 会自动完成初始化工作，然后就可以正常访问 PD。

15.8.2.2 PD 启动报错：etcd cluster ID mismatch

PD 启动参数中的 --initial-cluster 包含了某个不属于该集群的成员。遇到这个错误时请检查各个成员的所属集群，剔除错误的成员后即可正常启动。

15.8.2.3 PD 能容忍的时间同步误差是多少？

理论上，时间同步误差越小越好。PD 可容忍任意时长的误差，但是，时间同步误差越大意味着 PD 分配的时间戳与真实的物理时间相差越大，这个差距会影响读历史版本等功能。

15.8.2.4 Client 连接是如何寻找 PD 的？

Client 连接只能通过 TiDB 访问集群，TiDB 负责连接 PD 与 TiKV，PD 与 TiKV 对 Client 透明。当 TiDB 连接任意一台 PD 的时候，PD 会告知 TiDB 当前的 leader 是谁，如果此台 PD 不是 leader，TiDB 将会重新连接至 leader PD。

15.8.2.5 TiKV 节点 (Store) 各状态 (Up, Disconnect, Offline, Down, Tombstone) 之间的关系是什么？

使用 pd-ctl 可以查看 TiKV 节点的状态信息。如需查看各个状态之间的关系，请参考TiKV Store 状态之间的关系。

15.8.2.6 PD 参数中 leader-schedule-limit 和 region-schedule-limit 调度有什么区别？

- leader-schedule-limit 调度是用来均衡不同 TiKV 的 leader 数，影响处理查询的负载。
- region-schedule-limit 调度是均衡不同 TiKV 的副本数，影响不同节点的数据量。

15.8.2.7 每个 region 的 replica 数量可配置吗？调整的方法是？

可以，目前只能调整全局的 replica 数量。首次启动时 PD 会读配置文件 (conf/pd.yml)，使用其中的 max-replicas 配置，之后修改需要使用 pd-ctl 配置命令 `config set max-replicas $num`，配置后可通过 `config show all` 来查看已生效的配置。调整的时候，不会影响业务，会在后台添加，注意总 TiKV 实例数总是要大于等于设置的副本数，例如 3 副本需要至少 3 个 TiKV。增加副本数量之前需要预估额外的存储需求。pd-ctl 的详细用法可参考PD Control 使用说明。

15.8.2.8 缺少命令行集群管理工具，整个集群的健康度当前是否正常，不好确认？

可以通过 pd-ctl 等工具来判断集群大概的状态，详细的集群状态还是需要通过监控来确认。

15.8.2.9 集群下线节点后，怎么删除老集群节点监控信息？

下线节点一般指 TiKV 节点通过 pd-ctl 或者监控判断节点是否下线完成。节点下线完成后，手动停止下线节点上相关的服务。从 Prometheus 配置文件中删除对应节点的 node_exporter 信息。

15.8.3 TiDB server 管理

本小节介绍 TiDB server 管理中的常见问题、原因及解决方法。

15.8.3.1 TiDB 的 lease 参数应该如何设置？

启动 TiDB Server 时，需要通过命令行参数设置 lease 参数 (`--lease=60`)，其值会影响 DDL 的速度（只会影响当前执行 DDL 的 session，其他的 session 不会受影响）。在测试阶段，lease 的值可以设为 1s，加快测试进度；在生产环境下，我们推荐这个值设为分钟级（一般可以设为 60），这样可以保证 DDL 操作的安全。

15.8.3.2 DDL 在正常情况下的耗时是多少？

一般情况下处理一个 DDL 操作（之前没有其他 DDL 操作在处理）的耗时基本可以分如下为三种：

- add index 操作，且此操作对应表数据行数比较少，耗时约为 3s。
- add index 操作，且此操作对应表数据行数比较多，耗时具体由表中数据行数和当时 QPS 情况定（add index 操作优先级比一般 SQL 低）。
- 其他 DDL 操作耗时约为 1s。

此外，如果接收 DDL 请求的 TiDB 和 DDL owner 所处的 TiDB 是一台，那么上面列举的第一和第三种可能的耗时应该在几十到几百毫秒。

15.8.3.3 为什么有的时候执行 DDL 会很慢？

可能原因如下：

- 多个 DDL 语句一起执行的时候，后面的几个 DDL 语句会比较慢。原因是当前 TiDB 集群中 DDL 操作是串行执行的。
- 在正常集群启动后，第一个 DDL 操作的执行时间可能会比较久，一般在 30s 左右，这个原因是刚启动时 TiDB 在竞选处理 DDL 的 leader。
- 由于停 TiDB 时不能与 PD 正常通信（包括停电情况）或者用 `kill -9` 指令停 TiDB 导致 TiDB 没有及时从 PD 清理注册数据，那么会影响 TiDB 启动后 10min 内的 DDL 语句处理时间。这段时间内运行 DDL 语句时，每个 DDL 状态变化都需要等待 2 * lease（默认 lease = 45s）。
- 当集群中某个 TiDB 与 PD 之间发生通信问题，即 TiDB 不能从 PD 及时获取或更新版本信息，那么这时候 DDL 操作的每个状态处理需要等待 2 * lease。

15.8.3.4 TiDB 可以使用 S3 作为后端存储吗？

不可以，目前 TiDB 只支持分布式存储引擎和 GolevelDB/RocksDB/BoltDB 引擎。

15.8.3.5 Information_schema 能否支持更多真实信息？

Information_schema 库里面的表主要是为了兼容 MySQL 而存在，有些第三方软件会查询里面的信息。在目前 TiDB 的实现中，里面大部分只是一些空表。后续随着 TiDB 的升级，会提供更多的参数信息。当前 TiDB 支持的 Information_schema 请参考TiDB 系统数据库说明文档。

15.8.3.6 TiDB Backoff type 主要原因？

TiDB-server 与 TiKV-server 随时进行通信，在进行大量数据操作过程中，会出现 `Server is busy` 或者 `backoff.maxsleep 20000ms` 的日志提示信息，这是由于 TiKV-server 在处理过程中系统比较忙而出现的提示信息，通常这时候可以通过系统资源监控到 TiKV 主机系统资源使用率比较高的情况出现。如果这种情况出现，可以根据资源使用情况进行相应的扩容操作。

15.8.3.7 TiDB TiClient type 主要原因？

TiClient Region Error 该指标描述的是在 TiDB-server 作为客户端通过 KV 接口访问 TiKV-server 进行数据操作过程中，TiDB-server 操作 TiKV-server 中的 Region 数据出现的错误类型与 metric 指标，错误类型包括 not_leader、stale_epoch。出现这些错误的情况是当 TiDB-server 根据自己的缓存信息去操作 Region leader 数据的时候，Region leader 发生了迁移或者 TiKV 当前的 Region 信息与 TiDB 缓存的路由信息不一致而出现的错误提示。一般这种情况下，TiDB-server 都会自动重新从 PD 获取最新的路由数据，重做之前的操作。

15.8.3.8 TiDB 同时支持的最大并发连接数？

默认情况下，每个 TiDB 服务器的最大连接数没有限制。如有需要，可以在 `config.toml` 文件中设置 `instance.max_connections`（或者系统变量 `max_connections`）来限制最大连接数。如果并发量过大导致响应时间增加，建议通过添加 TiDB 节点进行扩容。

15.8.3.9 如何查看某张表创建的时间？

information_schema 库中的 tables 表里的 create_time 即为表的真实创建时间。

15.8.3.10 TiDB 的日志中 EXPENSIVE_QUERY 是什么意思？

TiDB 在执行 SQL 时，预估出来每个 operator 处理了超过 10000 条数据就认为这条 query 是 expensive query。可以通过修改 tidb-server 配置参数来对这个门限值进行调整，调整后需要重新启动 tidb-server。

15.8.3.11 如何预估 TiDB 中一张表的大小？

要预估 TiDB 中一张表的大小，你可以参考使用以下查询语句：

```
SELECT
    db_name,
    table_name,
    ROUND(SUM(total_size / cnt), 2) Approximate_Size,
    ROUND(SUM(total_size / cnt / (SELECT
                    ROUND(AVG(value), 2)
                FROM
                    METRICS_SCHEMA.store_size_amplification
                WHERE
                    value > 0)),
            2) Disk_Size
FROM
    (SELECT
        db_name,
            table_name,
            region_id,
            SUM(Approximate_Size) total_size,
            COUNT(*) cnt
    FROM
        information_schema.TIKV_REGION_STATUS
    WHERE
        db_name = @dbname
            AND table_name IN (@table_name)
    GROUP BY db_name , table_name , region_id) tabinfo
GROUP BY db_name , table_name;
```

在使用以上语句时，你需要根据实际情况填写并替换掉语句里的以下字段：

- `@dbname`：数据库名称。
- `@table_name`：目标表的名称。

此外，以上语句中：

- `store_size_amplification` 表示集群压缩比的平均值。除了使用 `SELECT * FROM METRICS_SCHEMA.store_size_amplification;` 语句进行查询以外，你还可以查看 Grafana 监控 PD - statistics balance 面板下各节点的 `Size amplification` 指标来获取该信息，集群压缩比的平均值即为所有节点的 `Size amplification` 平均值。
- `Approximate_Size` 表示压缩前表的单副本大小，该值为估算值，并非准确值。
- `Disk_Size` 表示压缩后表的大小，可根据 `Approximate_Size` 和 `store_size_amplification` 得出估算值。

15.8.4 TiKV 管理

本小节介绍 TiKV 管理中的常见问题、原因及解决方法。

15.8.4.1 如何为合规性或多租户应用程序指定数据位置？

可以使用放置规则 (Placement Rules) 为合规性或多租户应用程序指定数据位置。

Placement Rules in SQL 用于控制任何连续数据范围的属性，例如副本数量、Raft 角色、放置位置以及规则生效的键范围。

15.8.4.2 TiKV 集群副本建议配置数量是多少，是不是最小高可用配置（3 个）最好？

如果是测试环境 3 副本足够；在生产环境中，不可让集群副本数低于 3，需根据架构特点、业务系统及恢复能力的需求，适当增加副本数。值得注意的是，副本升高，性能会有下降，但是安全性更高。

15.8.4.3 TiKV 启动报错：cluster ID mismatch

TiKV 本地存储的 cluster ID 和指定的 PD 的 cluster ID 不一致。在部署新的 PD 集群的时候，PD 会随机生成一个 cluster ID，TiKV 第一次初始化的时候会从 PD 获取 cluster ID 存储在本地，下次启动的时候会检查本地的 cluster ID 与 PD 的 cluster ID 是否一致，如果不一致则会报错并退出。出现这个错误一个常见的原因是，用户原先部署了一个集群，后来把 PD 的数据删除了并且重新部署了新的 PD，但是 TiKV 还是使用旧的数据重启连到新的 PD 上，就会报这个错误。

15.8.4.4 TiKV 启动报错：duplicated store address

启动参数中的地址已经被其他的 TiKV 注册在 PD 集群中了。造成该错误的常见情况：TiKV `--data-dir` 指定的路径下没有数据文件夹（删除或移动后没有更新 –data-dir），用之前参数重新启动该 TiKV。请尝试用 pd-ctl 的 store delete 功能，删除之前的 store，然后重新启动 TiKV 即可。

15.8.4.5 TiKV master 和 slave 用的是一样的压缩算法，为什么效果不一样？

目前来看 master 有些文件的压缩率会高一些，这个取决于底层数据的分布和 RocksDB 的实现，数据大小偶尔有些波动是正常的，底层存储引擎会根据需要调整数据。

15.8.4.6 TiKV block cache 有哪些特性？

TiKV 使用了 RocksDB 的 Column Family (CF) 特性，KV 数据最终存储在默认 RocksDB 内部的 default、write、lock 3 个 CF 内。

- default CF 存储的是真正的数据，与其对应的参数位于 `[rocksdb.defaultcf]` 项中。
- write CF 存储的是数据的版本信息 (MVCC)、索引、小表相关的数据，相关的参数位于 `[rocksdb.writecf]` 项中。
- lock CF 存储的是锁信息，系统使用默认参数。
- Raft RocksDB 实例存储 Raft log。default CF 主要存储的是 Raft log，与其对应的参数位于 `[raftdb.defaultcf]` 项中。

- 所有 CF 共享一个 Block-cache，用于缓存数据块，加速 RocksDB 的读取速度。Block-cache 的大小通过参数 `block-cache-size` 控制，`block-cache-size` 越大，能够缓存的热点数据越多，对读取操作越有利，同时占用的系统内存也会越多。
- 每个 CF 有各自的 Write-buffer，大小通过 `write-buffer-size` 控制。

15.8.4.7 TiKV channel full 是什么原因？

- Raftstore 线程太忙，或者因 I/O 而卡住。可以看一下 Raftstore 的 CPU 使用情况。
- TiKV 过忙（CPU、磁盘 I/O 等），请求处理不过来。

15.8.4.8 TiKV 频繁切换 Region leader 是什么原因？

- 网络问题导致节点间通信卡了，查看 Report failures 监控。
- 原主 Leader 的节点卡了，导致没有及时给 Follower 发送消息。
- Raftstore 线程卡了。

15.8.4.9 如果一个节点挂了会影响服务吗？影响会持续多久？

TiDB 使用 Raft 在多个副本之间做数据同步（默认为每个 Region 3 个副本）。当一份备份出现问题时，其他的副本能保证数据的安全。根据 Raft 协议，当某个节点挂掉导致该节点里的 Leader 失效时，在最大 2 * lease time（leasetime 是 10 秒）时间后，通过 Raft 协议会很快将一个另外一个节点里的 Follower 选为新的 Region Leader 来提供服务。

15.8.4.10 TiKV 在分别在哪些场景下占用大量 IO、内存、CPU（超过参数配置的多倍）?

在大量写入、读取的场景中会占用大量的磁盘 IO、内存、CPU。在执行很复杂的查询，比如会产生很大中间结果集的情况下，会消耗很多的内存和 CPU 资源。

15.8.4.11 TiKV 是否可以使用 SAS/SATA 盘或者进行 SSD/SAS 混合部署？

不可以使用。TiDB 在进行 OLTP 场景中，数据访问和操作需要高 IO 磁盘的支持。TiDB 作为强一致的分布式数据库，存在一定的写放大，如副本复制、存储底层 Compaction，因此，TiDB 部署的最佳实践中推荐用户使用 NVMe SSD 磁盘作为数据存储磁盘。另外，TiKV 与 PD 不能混合部署。

15.8.4.12 数据表 Key 的 Range 范围划分是在数据接入之前就已经划分好了吗？

不是的，这个和 MySQL 分表规则不一样，需要提前设置好，TiKV 是根据 Region 的大小动态分裂的。

15.8.4.13 Region 是如何进行分裂的？

Region 不是前期划分好的，但确实有 Region 分裂机制。当 Region 的大小超过参数 `region-max-size` 或 `region-max-keys` 的值时，就会触发分裂，分裂后的信息会汇报给 PD。

15.8.4.14 TiKV 是否有类似 MySQL 的 `innodb_flush_log_trx_commit` 参数，来保证提交数据不丢失？

是的。TiKV 单机的存储引擎目前使用两个 RocksDB 实例，其中一个存储 raft-log。TiKV 有个 sync-log 参数，在 true 的情况下，每次提交都会强制刷盘到 raft-log，如果发生 crash 后，通过 raft-log 进行 KV 数据的恢复。

15.8.4.15 对 WAL 存储有什么推荐的硬件配置，例如 SSD，RAID 级别，RAID 卡 cache 策略，NUMA 设置，文件系统选择，操作系统的 IO 调度策略等？

WAL 属于顺序写，目前我们并没有单独对他进行配置，建议 SSD。RAID 如果允许的话，最好是 RAID 10，RAID 卡 cache、操作系统 I/O 调度目前没有针对性的最佳实践，Linux 7 以上默认配置即可。NUMA 没有特别建议，NUMA 内存分配策略可以尝试使用 `interleave = all`，文件系统建议 ext4。

15.8.4.16 在最严格的 `sync-log = true` 数据可用模式下，写入性能如何？

一般来说，开启 `sync-log` 会让性能损耗 30% 左右。关闭 `sync-log` 时的性能表现，请参见 TiDB Sysbench 性能测试报告。

15.8.4.17 是否可以利用 TiKV 的 Raft + 多副本达到完全的数据可靠，单机存储引擎是否需要最严格模式？

通过使用 Raft 一致性算法，数据在各 TiKV 节点间复制为多副本，以确保某个节点挂掉时数据的安全性。只有当数据已写入超过 50% 的副本时，应用才返回 ACK（三副本中的两副本）。但理论上两个节点也可能同时发生故障，所以除非是对性能要求高于数据安全的场景，一般都强烈推荐开启 `sync-log`。

另外，还有一种 `sync-log` 的替代方案，即在 Raft group 中用五个副本而非三个。这将允许两个副本同时发生故障，而仍然能保证数据安全性。

对于单机存储引擎也同样推荐打开 `sync-log` 模式。否则如果节点宕机可能会丢失最后一次写入数据。

15.8.4.18 使用 Raft 协议，数据写入会有多次网络的 roundtrip，实际写入延迟如何？

理论上，和单机数据库相比，数据写入会多四个网络延迟。

15.8.4.19 有没有类似 MySQL 的 InnoDB Memcached plugin，可以直接使用 KV 接口，可以不需要独立的 Cache？

TiKV 支持单独进行接口调用，理论上也可以起个实例做为 Cache，但 TiDB 最大的价值是分布式关系型数据库，我们原则上不对 TiKV 单独进行支持。

15.8.4.20 Coprocessor 组件的主要作用？

- 减少 TiDB 与 TiKV 之间的数据传输。
- 计算下推，充分利用 TiKV 的分布式计算资源。

15.8.4.21 IO error: No space left on device While appending to file

这是磁盘空间不足导致的，需要加节点或者扩大磁盘空间。

15.8.4.22 为什么 TiKV 容易出现 OOM？

TiKV 的内存占用主要来自于 RocksDB 的 block-cache，默认为系统总内存的 40%。当 TiKV 容易出现 OOM 时，检查 `block-cache-size` 配置是否过高。还需要注意，当单机部署了多个 TiKV 实例时，需要显式地配置该参数，以防止多个实例占用过多系统内存导致 OOM。

15.8.4.23 TiDB 数据和 RawKV 数据可存储于同一个 TiKV 集群里吗？

不可以。TiDB 数据（或使用其他事务 API 生成的数据）依赖于一种特殊的键值格式，和 RawKV API 数据（或其他基于 RawKV 的服务生成的数据）并不兼容。

15.8.5 TiDB 测试

本小节介绍 TiDB 测试中的常见问题、原因及解决方法。

15.8.5.1 TiDB Sysbench 基准测试结果如何？

很多用户在接触 TiDB 都习惯做一个基准测试或者 TiDB 与 MySQL 的对比测试，官方也做了一个类似测试。我们汇总很多测试结果后，发现虽然测试的数据有一定的偏差，但结论或者方向基本一致，由于 TiDB 与 MySQL 由于架构上的差别非常大，很多方面是很难找到一个基准点，所以官方的建议两点：

- 大家不要用过多精力纠结这类基准测试上，应该更多关注 TiDB 的场景上的区别。
- 大家可以直接参考 TiDB Sysbench 性能测试报告。

15.8.5.2 TiDB 集群容量 QPS 与节点数之间关系如何，和 MySQL 对比如何？

- 在 10 节点内，TiDB 写入能力 (Insert TPS) 和节点数量基本成 40% 线性递增，MySQL 由于是单节点写入，所以不具备写入扩展能力。
- MySQL 读扩容可以通过添加从库进行扩展，但写流量无法扩展，只能通过分库分表，而分库分表有很多问题，具体参考方案虽好，成本先行：数据库 Sharding+Proxy 实践解析。
- TiDB 不管是读流量、还是写流量都可以通过添加节点快速方便的进行扩展。

15.8.5.3 我们的 DBA 测试过 MySQL 性能，单台 TiDB 的性能没有 MySQL 性能那么好？

TiDB 设计的目标就是针对 MySQL 单台容量限制而被迫做的分库分表的场景，或者需要强一致性和完整分布式事务的场景。它的优势是通过尽量下推到存储节点进行并行计算。对于小表（比如千万级以下），不适合 TiDB，因为数据量少，Region 有限，发挥不了并行的优势。其中最极端的例子就是计数器表，几行记录高频更新，这几行在 TiDB 里，会变成存储引擎上的几个 KV，然后只落在一个 Region 里，而这个 Region 只落在一个节点上。加上后台强一致性复制的开销，TiDB 引擎到 TiKV 引擎的开销，最后表现出来的就是没有单个 MySQL 好。

15.8.6 TiDB 备份恢复

本小节介绍 TiDB 备份恢复中的常见问题、原因及解决方法。

15.8.6.1 TiDB 主要备份方式？

目前，数据量大时（大于 1 TB）推荐使用Backup & Restore (BR) 进行备份。其他场景推荐使用Dumpling 进行备份。

尽管 TiDB 也支持使用 MySQL 官方工具 `mysqldump` 进行数据备份和恢复，但其性能低于Dumpling，并且 `mysqldump` 备份和恢复大量数据的耗费更长。

其他备份恢复相关问题，可以参考备份与恢复常见问题。

15.8.6.2 备份和恢复的速度如何？

使用BR 进行备份和恢复时，备份速度大约为每个 TiKV 实例 40 MB/s，恢复速度大约为每个 TiKV 实例 100 MB/s。

15.9 高可用常见问题

本文档介绍高可用相关的常见问题。

15.9.1 TiDB 数据是强一致的吗？

通过使用 Raft 一致性算法，数据在各 TiKV 节点间复制为多副本，以确保某个节点宕机时数据的安全性。

在底层，TiKV 使用复制日志 + 状态机 (State Machine) 的模型来复制数据。对于写入请求，数据被写入 Raft Leader，然后 Leader 以日志的形式将命令复制到它的 Follower 中。当集群中的大多数节点收到此日志时，日志会被提交，状态机会相应作出变更，以此来实现强一致。

15.9.2 TiDB 是否提供三中心跨机房多活部署的推荐方案？

从架构来看，TiDB 支持真正意义上的跨中心异地多活。从实现层面看，多地部署方案依赖数据中心之间的网络延迟和稳定性，一般建议延迟在 5ms 以下。目前 TiDB 已经有相似的客户部署方案，参见两地三中心部署方案。

15.10 高可靠常见问题

本文档介绍高可靠相关的常见问题。

15.10.1 TiDB 是否支持数据加密？

支持。要加密传输中的数据，可以在 TiDB 客户端和服务器之间启用 TLS。要加密存储引擎中的数据，可以启用透明数据加密 (TDE)。

15.10.2 我们的安全漏洞扫描工具对 MySQL version 有要求，TiDB 是否支持修改 server 版本号呢？

TiDB 在 v3.0.8 后支持通过 TiDB 配置文件中的`server-version` 配置项来修改 server 版本号。

对于 v4.0 及以上版本的集群，如果使用 TiUP 部署集群，可以通过 `tiup cluster edit-config <cluster-name>` 修改配置文件中以下部分来设置合适的版本号：

```
server_configs:
  tidb:
    server-version: 'YOUR_VERSION_STRING'
```

修改完成后，使用 `tiup cluster reload <cluster-name> -R tidb` 命令使得以上修改生效，以避免出现安全漏洞扫描不通过的问题。

15.10.3 TiDB 支持哪些认证协议？过程是怎样的？

TiDB 和 MySQL 一样，在用户登录认证时使用 SASL 认证协议对密码进行处理。

客户端连接 TiDB 的时候，使用 challenge-response（挑战 -应答）的认证模式，过程如下：

1. 客户端连接服务器。
2. 服务器发送随机字符串 `challenge` 给客户端。
3. 客户端发送 `username` + `response` 给服务器。
4. 服务器验证 `response`。

15.10.4 如何修改用户名密码和权限？

因为 TiDB 是分布式数据库，想要在 TiDB 中修改用户密码，建议使用 `ALTER USER` 的方法，例如 `ALTER USER 'test'@'localhost' IDENTIFIED BY 'mypass';`。

不推荐使用 `UPDATE mysql.user` 的方法，因为这种方法可能会造成其它节点刷新不及时的情况。修改权限也一样，建议参考TiDB 用户账户管理文档中的方法。

15.11 备份与恢复常见问题

本文列出了在使用 Backup & Restore (BR) 完成备份与恢复任务时，可能会遇到的问题及相应的解决方法。

如果遇到未包含在此文档且无法解决的问题，可以在 AskTUG 社区中提问。

15.11.1 当误删除或误更新数据后，如何原地快速恢复？

从 TiDB v6.4.0 引入了完整的 Flashback 功能，可以支持原地快速恢复 GC 时间内的数据到指定时间点。在误操作场景下，推荐使用 Flashback 来恢复数据，具体可以参考Flashback 集群 和Flashback 数据库语法。

15.11.2 在 TiDB v5.4.0 及后续版本中，当在有负载的集群进行备份时，备份速度为什么会变得很慢？

从 TiDB v5.4.0 起，TiKV 的备份新增了自动调节功能。对于 v5.4.0 及以上版本，该功能会默认开启。当集群负载较高时，该功能会自动限制备份任务使用的资源，从而减少备份对在线集群的性能造成的影响。如需了解关于自动调节功能的更多信息，请参见自动调节。

TiKV 支持动态配置自动调节功能。因此，在开启或关闭该功能时，你不需要重启集群。以下为该功能的具体使用方法：

- 关闭功能：把 TiKV 配置项`backup.enable-auto-tune` 设置为 `false`。
- 开启功能：把 `backup.enable-auto-tune` 设置为 `true`。对于 v5.3.x 版本的集群，当 TiDB 升级到 v5.4.0 及以上版本后，自动调节功能会默认关闭。这时，你可以通过该方式手动开启此功能。

如需了解通过 `tikv-ctl` 在线修改自动调节功能的命令行，请参阅自动调节功能的使用方法。

另外，自动调节功能减少了进行备份任务时默认使用的工作线程数量（详见`backup.num-threads`）。因此，你通过 Grafana 监控面板看到的备份速度、CPU 使用率、I/O 资源利用率都会小于 v5.4 之前的版本。在 v5.4.0 之

前的版本中，backup.num-threads 的默认值为 CPU * 0.75，即处理备份任务的工作线程数量占了 75% 的逻辑 CPU，最大值为 32；在 v5.4.0 及之后的版本中，该配置项的默认值为 CPU * 0.5，最大值为 8。

在离线备份场景中，你也可以使用 tikv-ctl 把 backup.num-threads 修改为更大的数字，从而提升备份任务的速度。

15.11.3 PITR 问题

15.11.3.1 PITR 功能和flashback 集群有什么区别?

从使用场景角度来看，PITR 通常用于在集群完全停止服务或数据损坏且无法使用其他方案恢复时，将集群的数据恢复到指定的时间点。使用 PITR 时，你需要通过一个新的集群来完成数据恢复。而 flashback 集群则通常用于发生误操作或其他因素导致的数据错误时，将集群的数据恢复到数据错误发生前的最近时间点。

在大多数情况下，因为 flashback 具有更短的 RPO（接近零）和 RTO，flashback 比 PITR 更适合由于人为错误导致的数据错误场景。但当集群完全不可用时，flashback 集群功能也无法运行，此时 PITR 是恢复集群的唯一方案。因此，相比于 flashback，虽然 PITR 的 RPO（最长 5 分钟）和 RTO 时间较长，但 PITR 是制定数据库灾难恢复策略时必须考虑的数据安全基础。

15.11.3.2 上游数据库使用 TiDB Lightning Physical 方式导入数据时，为什么无法使用日志备份功能？

目前日志备份功能还没有完全适配 TiDB Lightning，导致 TiDB Lightning Physical 方式导入的数据无法备份到日志中。

在创建日志备份任务的上游集群中，请尽量避免使用 TiDB Lightning Physical 方式导入数据。可以选择使用 TiDB Lightning Logical 方式导入数据。若确实需要使用 Physical 导入方式，可在导入完成之后做一次快照备份操作，这样，PITR 就可以恢复到快照备份之后的时间点。

15.11.3.3 索引加速功能为什么与 PITR 功能不兼容？

Issue 链接：#38045

当前通过索引加速功能创建的索引数据无法被 PITR 备份。

因此，在 PITR 恢复完成后，BR 会将通过索引加速功能创建的索引数据删除，再重新创建。如果在日志备份期间通过索引加速功能创建的索引很多或索引数据很大，建议在创建索引后进行一次全量备份。

15.11.3.4 集群已经恢复了网络分区故障，为什么日志备份任务进度 checkpoint 仍然不推进？

Issue 链接：#13126

在集群出现网络分区故障后，备份任务难以继续备份日志，并且在超过一定的重试时间后，任务会被置为 ERROR 状态。此时备份任务已经停止，需要手动执行 br log resume 命令来恢复日志备份任务。

15.11.3.5 执行 PITR 恢复时遇到 execute over region id 报错，该如何处理？

Issue 链接：#37207

该场景发生在全量数据导入时开启了日志备份，并使用 PITR 恢复全量导入时间段的日志。经过测试发现，当存在长时间（24 小时）大量热点写入，且平均单台 TiKV 节点写入 OPS > 50k/s（可以通过 Grafana 中 TiKV-Details -> Backup Log -> Handle Event Rate 确认该数值），那么有几率会遇到这个情况。

当前版本中建议在集群初始化后，进行一次有效快照备份，并且以此作为基础进行 PITR 恢复。

15.11.3.6 在使用 br restore point 命令恢复下游集群后，TiFlash 引擎数据没有恢复？

PITR 目前不支持在恢复阶段直接将数据写入 TiFlash，在数据恢复完成后，br 会执行 ALTER TABLE table_name ↪ SET TIFLASH REPLICA ***，因此 TiFlash 副本在 PITR 完成恢复之后并不能马上可用，而是需要等待一段时间从 TiKV 节点同步数据。要查看同步进度，可以查询 INFORMATION_SCHEMA.tiflash_replica 表中的 progress 信息。

15.11.3.7 日志备份任务的 status 变为 ERROR，该如何处理？

在运行日志备份过程中，遇到错误后经过重试也无法恢复的故障场景，任务会被设置为 ERROR 状态，如：

```
br log status --pd x.x.x.x:2379

• Total 1 Tasks.
> #1 <
                    name: task1
                  status: ○ ERROR
                   start: 2022-07-25 13:49:02.868 +0000
                     end: 2090-11-18 14:07:45.624 +0000
                 storage: s3://tmp/br-log-backup0ef49055-5198-4be3-beab-d382a2189efb/Log
            speed(est.): 0.00 ops/s
      checkpoint[global]: 2022-07-25 14:46:50.118 +0000; gap=11h31m29s
          error[store=1]: KV:LogBackup:RaftReq
error-happen-at[store=1]: 2022-07-25 14:54:44.467 +0000; gap=11h23m35s
  error-message[store=1]: retry time exceeds: and error failed to get initial snapshot: failed to
      ↪  get the snapshot (region_id = 94812): Error during requesting raftstore: message: "
      ↪ read index not ready, reason can not read index due to merge, region 94812"
      ↪ read_index_not_ready { reason: "can not read index due to merge" region_id: 94812 }:
      ↪ failed to get initial snapshot: failed to get the snapshot (region_id = 94812): Error
      ↪ during requesting raftstore: message: "read index not ready, reason can not read index
      ↪ due to merge, region 94812" read_index_not_ready { reason: "can not read index due to
      ↪ merge" region_id: 94812 }: failed to get initial snapshot: failed to get the snapshot (
      ↪ region_id = 94812): Error during requesting raftstore: message: "read index not ready,
      ↪ reason can not read index due to merge, region 94812" read_index_not_ready { reason: "
      ↪ can not read index due to merge" region_id: 94812 }
```

此时，你需要根据错误提示来确认故障的原因并恢复故障。确认故障恢复之后，可执行下面的命令来恢复备份任务：

```
br log resume --task-name=task1 --pd x.x.x.x:2379
```

备份任务恢复后，再次查询 br log status，任务状态变为正常，备份任务继续执行。

```
• Total 1 Tasks.
> #1 <
```

```
              name: task1
            status: ● NORMAL
             start: 2022-07-25 13:49:02.868 +0000
               end: 2090-11-18 14:07:45.624 +0000
           storage: s3://tmp/br-log-backup0ef49055-5198-4be3-beab-d382a2189efb/Log
      speed(est.): 15509.75 ops/s
checkpoint[global]: 2022-07-25 14:46:50.118 +0000; gap=6m28s
```

> **注意：**
>
> 由于此功能会备份集群的多版本数据，当任务发生错误且状态变更为 ERROR 时，同时会将当前任务的备份进度点的数据设为一个 safe point，safe point 的数据将保证 24 小时不被 GC 掉。所以，当任务恢复之后，会从上一个备份点继续备份日志。如果任务失败时间超过 24 小时，前一次备份进度点的数据就已经 GC，此时恢复任务操作会提示失败。这种场景下，只能执行 br log stop 命令先停止本次任务，然后重新开启新的备份任务。

15.11.3.8 执行 br log resume 命令恢复处于暂停状态的任务时报 ErrBackupGCSafepointExceeded 错误，该如何处理？

```
Error: failed to check gc safePoint, checkpoint ts 433177834291200000: GC safepoint
    ↪ 433193092308795392 exceed TS 433177834291200000: [BR:Backup:ErrBackupGCSafepointExceeded]
    ↪ backup GC safepoint exceeded
```

暂停日志备份任务后，备份程序为了防止生成变更日志的 MVCC 数据被 GC，暂停任务程序会自动将当前备份点 checkpoint 设置为 service safepoint，允许保留最近 24 小时内的 MVCC 数据。当超过 24 小时后，备份点 checkpoint 的 MVCC 数据已经被 GC，此时程序会拒绝恢复备份任务。

此场景的处理办法是：先执行 br log stop 命令来删除当前的任务，然后执行 br log start 重新创建新的日志备份任务，同时做一个全量备份，便于后续做 PITR 恢复操作。

15.11.4 功能兼容性问题

15.11.4.1 为什么 BR 恢复的数据无法同步到 TiCDC / Drainer 的上游集群？

- BR 恢复的数据无法被同步到下游，因为恢复时 BR 直接导入 SST 文件，而下游集群目前没有办法获得上游的 SST 文件。
- 在 4.0.3 版本之前，恢复时产生的 DDL jobs 还可能会让 TiCDC / Drainer 执行异常的 DDL。所以，如果一定要在 TiCDC / Drainer 的上游集群执行恢复，请将 BR 恢复的所有表加入 TiCDC / Drainer 的阻止名单。

TiCDC 可以通过配置项中的 filter.rules 项完成，Drainer 则可以通过syncer.ignore-table 完成。

15.11.4.2 恢复时为什么会报 new_collations_enabled_on_first_bootstrap 不匹配？

从 TiDB v6.0.0 版本开始，new_collations_enabled_on_first_bootstrap 配置项的默认值由 false 改为 true。BR 会备份上游集群的 new_collations_enabled_on_first_bootstrap 配置项。当上下游集群的此项配置相同时，BR 才会将上游集群的备份数据安全地恢复到下游集群中。若上下游的该配置不相同，BR 会拒绝恢复，并报此配置项不匹配的错误。

如果需要将旧版本的备份数据恢复到 TiDB v6.0.0 或更新版本的 TiDB 集群中，你需要检查上下游集群中的该配置项是否相同：

- 若该配置项相同，则可在恢复命令中添加 --check-requirements=false 以跳过此项配置检查。
- 若该配置项不相同，且进行强行恢复，BR 会报数据校验错误。

15.11.4.3 恢复 Placement Rule 到集群时为什么会报错？

BR 在 v6.0.0 之前不支持放置规则，在 v6.0.0 及以上版本开始支持并提供了命令行选项 --with-tidb-placement-mode=strict/ignore 来控制放置规则的导入模式。默认值为 strict，代表导入并检查放置规则，当--with-tidb-placement-mode 设置为 ignore 时，BR 会忽略所有的放置规则。

15.11.5 进行数据恢复的问题

15.11.5.1 使用 BR 时遇到错误信息 Io(Os...)，该如何处理？

这类问题几乎都是 TiKV 在写盘的时候遇到的系统调用错误。例如遇到 Io(Os { code: 13, kind: PermissionDenied...}) 或者 Io(Os { code: 2, kind: NotFound...})。

首先检查备份目录的挂载方式和文件系统，尝试备份到其它文件夹或者硬盘。

目前已知备份到 samba 搭建的网盘时可能会遇到 Code: 22(invalid argument) 错误。

15.11.5.2 恢复时遇到错误信息 rpc error: code = Unavailable desc =...，该如何处理？

该问题一般是因为恢复数据的时候，恢复集群的性能不足导致的。可以从恢复集群的监控或者 TiKV 日志来辅助确认。

要解决这类问题，可以尝试扩大集群资源，以及调小恢复时的并发度 (concurrency)，打开限速 (ratelimit) 设置。

15.11.5.3 恢复备份数据时，BR 会报错 entry too large, the max entry size is 6291456, the size of data is 7690800

你可以尝试降低并发批量建表的大小，将 --ddl-batch-size 设置为 128 或者更小的值。

在--ddl-batch-size 的值大于 1 的情况下，使用 BR 恢复数据时，TiDB 会把执行创建表任务的 DDL job 队列写到 TiKV 上。由于 TiDB 能够一次性发送的 job message 的最大值默认为 6 MB（不建议修改此值，具体内容，参考txn-entry-size-limit 和raft-entry-max-size），TiDB 单次发送的所有表的 schema 大小总和也不应该超过 6 MB。因此，如果你设置的 --ddl-batch-size 的值过大，TiDB 单次发送的批量表的 schema 大小就会超出规定值，从而导致 BR 报 entry too large, the max entry size is 6291456, the size of data is 7690800 错误。

15.11.5.4 使用 local storage 的时候，备份的文件会存在哪里？

> **注意：**
>
> 如果没有挂载 NFS 到 br 工具或 TiKV 节点，或者使用了支持 S3、GCS 或 Azure Blob Storage 协议的远端存储，那么 br 工具备份的数据会在各个 TiKV 节点生成。注意这不是推荐的备份和恢复使用方式，因为备份数据会分散在各个节点的本地文件系统中，聚集这些备份数据可能会造成数据冗余和运维上的麻烦，而且在不聚集这些数据便直接恢复的时候会遇到 `SST file not found` 报错。

在使用 local storage 的时候，会在运行 br 命令行工具的节点生成 `backupmeta`，在各个 Region 的 Leader 所在 TiKV 节点生成备份文件。

15.11.5.5 恢复的时候，报错 `could not read local://...:download sst failed`，该如何处理？

在恢复的时候，每个节点都必须能够访问到所有的备份文件 (SST files)。默认情况下，假如使用 local storage，备份文件会分散在各个节点中，此时是无法直接恢复的，必须将每个 TiKV 节点的备份文件拷贝到其它所有 TiKV 节点才能恢复。所以建议使用 S3/GCS/Azure Blob Storage/NFS 作为备份存储。

15.11.5.6 遇到 Permission denied 或者 No such file or directory 错误，即使用 root 运行 br 命令行工具也无法解决，该如何处理？

需要确认 TiKV 是否有访问备份目录的权限。如果是备份，确认是否有写权限；如果是恢复，确认是否有读权限。

在进行备份操作时，如果使用本地磁盘或 NFS 作为存储介质，请确保执行 br 命令行工具的用户和启动 TiKV 的用户相同（如果 br 工具和 TiKV 位于不同的机器，则需要用户的 UID 相同），否则备份可能会出现该问题。

使用 root 运行 br 命令行工具仍旧有可能会因为磁盘权限而失败，因为备份文件 (SST) 的保存是由 TiKV 执行的。

> **注意：**
>
> 在恢复的时候也可能遇到同样的问题。
>
> 使用 br 命令行工具进行数据的恢复时，检验读权限的时机是在第一次读取 SST 文件时，考虑到执行 DDL 的耗时，这个时刻可能会离开始运行 br 命令行工具的时间很远。这样可能会出现等了很长时间之后遇到 Permission denied 错误失败的情况。
>
> 因此，最好在恢复前提前检查权限。

你可以按照如下步骤进行权限检查：

1. 执行 Linux 原生的进程查询命令

```
ps aux | grep tikv-server
```

命令输出示例如下:

```
tidb_ouo  9235 10.9  3.8 2019248 622776 ?      Ssl  08:28   1:12 bin/tikv-server --addr
    ↪ 0.0.0.0:20162 --advertise-addr 172.16.6.118:20162 --status-addr 0.0.0.0:20188 --
    ↪ advertise-status-addr 172.16.6.118:20188 --pd 172.16.6.118:2379 --data-dir /home/
    ↪ user1/tidb-data/tikv-20162 --config conf/tikv.toml --log-file /home/user1/tidb-
    ↪ deploy/tikv-20162/log/tikv.log
tidb_ouo  9236  9.8  3.8 2048940 631136 ?      Ssl  08:28   1:05 bin/tikv-server --addr
    ↪ 0.0.0.0:20161 --advertise-addr 172.16.6.118:20161 --status-addr 0.0.0.0:20189 --
    ↪ advertise-status-addr 172.16.6.118:20189 --pd 172.16.6.118:2379 --data-dir /home/
    ↪ user1/tidb-data/tikv-20161 --config conf/tikv.toml --log-file /home/user1/tidb-
    ↪ deploy/tikv-20161/log/tikv.log
```

或者执行以下命令：

```
ps aux | grep tikv-server | awk '{print $1}'
```

命令输出示例如下:

```
tidb_ouo
tidb_ouo
```

2. 使用 TiUP 命令查询集群的启动信息

```
tiup cluster list
```

命令输出示例如下：

```
[root@Copy-of-VM-EE-CentOS76-v1 br]# tiup cluster list
Starting component `cluster`: /root/.tiup/components/cluster/v1.5.2/tiup-cluster list
Name          User      Version  Path
    ↪ PrivateKey
----          ----      -------  ----
    ↪ ----------
tidb_cluster  tidb_ouo  v5.0.2   /root/.tiup/storage/cluster/clusters/tidb_cluster  /root/.
    ↪ tiup/storage/cluster/clusters/tidb_cluster/ssh/id_rsa
```

3. 检查备份目录的权限，例如 backup 目录是备份数据存储目录。命令示例如下：

```
ls -al backup
```

命令输出示例如下：

```
[root@Copy-of-VM-EE-CentOS76-v1 user1]# ls -al backup
total 0
drwxr-xr-x  2 root root   6 Jun 28 17:48 .
drwxr-xr-x 11 root root 310 Jul  4 10:35 ..
```

由以上命令输出结果可知，tikv-server 实例由用户 tidb_ouo 启动，但该账号没有 backup 目录的写入权限，所以备份失败。

15.11.5.7 恢复集群的时候，在 MySQL 下的业务表为什么没有恢复？

自 br v5.1.0 开始，全量备份会备份 mysql schema 下的表。br v6.2.0 以前的版本，在默认设置不会恢复 mysql schema 下的表。

如果需要恢复 `mysql` 下的用户创建的表（非系统表），可以通过table filter 来显式地包含目标表。以下示例中命令会在执行正常的恢复的同时恢复 `mysql.usertable`。

```
br restore full -f '*.*' -f '!mysql.*' -f 'mysql.usertable' -s $external_storage_url --with-sys-
    ↪ table
```

在上面的命令中，

- `-f '*.*'` 用于覆盖掉默认的规则。
- `-f '!mysql.*'` 指示 BR 不要恢复 `mysql` 中的表，除非另有指定。
- `-f 'mysql.usertable'` 则指定需要恢复 `mysql.usertable`。

如果只需要恢复 `mysql.usertable`，而无需恢复其他表，可以使用以下命令：

```
br restore full -f 'mysql.usertable' -s $external_storage_url --with-sys-table
```

此外请注意，即使设置了table filter，BR 也不会恢复以下系统表：

- 统计信息表（`mysql.stat_*`）
- 系统变量表（`mysql.tidb`、`mysql.global_variables`）
- 其他系统表

15.11.6 备份恢复功能相关知识

15.11.6.1 备份数据有多大，备份会有副本吗？

备份的时候仅仅在每个 Region 的 Leader 处生成该 Region 的备份文件。因此备份的大小等于数据大小，不会有多余的副本数据。所以最终的总大小大约是 TiKV 数据总量除以副本数。

但是假如想要从本地恢复数据，因为每个 TiKV 都必须要能访问到所有备份文件，在最终恢复的时候会有等同于恢复时 TiKV 节点数量的副本。

15.11.6.2 BR 备份恢复后，为什么监控显示磁盘使用空间不一致？

这个情况多数是因为备份时集群的数据压缩比率和恢复时的默认值不一致导致的，只要恢复的 checksum 阶段顺利通过，可以忽略这个问题，不影响正常使用。

15.11.6.3 使用 BR 恢复数据后是否需要对表执行 ANALYZE 以更新 TiDB 在表和索引上留下的统计信息？

BR 不会备份统计信息（v4.0.9 除外）。所以在恢复存档后需要手动执行 `ANALYZE TABLE` 或等待 TiDB 自动进行 ANALYZE。

BR v4.0.9 备份统计信息使 br 工具消耗过多内存，为保证备份过程正常，从 v4.0.10 开始默认关闭备份统计信息的功能。

如果不对表执行 ANALYZE，TiDB 会因统计信息不准确而选不中最优化的执行计划。如果查询性能不是重点关注项，可以忽略 ANALYZE。

15.11.6.4 可以同时启动多个恢复任务对单个集群进行恢复吗？

强烈不建议在单个集群中同时启动多个恢复任务进行数据恢复，原因如下：

- BR 在恢复数据时，会修改 PD 的一些全局配置。如果同时使用多个 BR 命令进行恢复，这些配置可能会被错误地覆写，导致集群状态异常。
- 因为 BR 在恢复数据的时候会占用大量集群资源，事实上并行恢复能获得的速度提升也非常有限。
- 多个恢复任务同时进行的场景没有经过测试，无法保证成功。

15.11.6.5 BR 会备份表的 SHARD_ROW_ID_BITS 和 PRE_SPLIT_REGIONS 信息吗？恢复出来的表会有多个 Region 吗？

会，BR 会备份表的SHARD_ROW_ID_BITS 和 PRE_SPLIT_REGIONS 信息，并恢复成多个 Region。

16 版本发布历史

16.1 TiDB 版本发布历史

TiDB 历史版本发布声明如下：

16.1.1 7.0

- 7.0.0-DMR: 2023-03-30

16.1.2 6.6

- 6.6.0-DMR: 2023-02-20

16.1.3 6.5

- 6.5.3: 2023-06-14
- 6.5.2: 2023-04-21
- 6.5.1: 2023-03-10
- 6.5.0: 2022-12-29

16.1.4 6.4

- 6.4.0-DMR: 2022-11-17

16.1.5 6.3

- 6.3.0-DMR: 2022-09-30

16.1.6 6.2

- 6.2.0-DMR: 2022-08-23

16.1.7 6.1

- 6.1.7: 2023-07-12
- 6.1.6: 2023-04-12
- 6.1.5: 2023-02-28
- 6.1.4: 2023-02-08
- 6.1.3: 2022-12-05
- 6.1.2: 2022-10-24
- 6.1.1: 2022-09-01
- 6.1.0: 2022-06-13

16.1.8 6.0

- 6.0.0-DMR: 2022-04-07

16.1.9 5.4

- 5.4.3: 2022-10-13
- 5.4.2: 2022-07-08
- 5.4.1: 2022-05-13
- 5.4.0: 2022-02-15

16.1.10 5.3

- 5.3.4: 2022-11-24
- 5.3.3: 2022-09-14
- 5.3.2: 2022-06-29
- 5.3.1: 2022-03-03
- 5.3.0: 2021-11-30

16.1.11 5.2

- 5.2.4: 2022-04-26
- 5.2.3: 2021-12-03
- 5.2.2: 2021-10-29
- 5.2.1: 2021-09-09
- 5.2.0: 2021-08-27

16.1.12 5.1

- 5.1.5: 2022-12-28
- 5.1.4: 2022-02-22
- 5.1.3: 2021-12-03
- 5.1.2: 2021-09-27
- 5.1.1: 2021-07-30
- 5.1.0: 2021-06-24

16.1.13 5.0

- 5.0.6: 2021-12-31
- 5.0.5: 2021-12-03
- 5.0.4: 2021-09-27
- 5.0.3: 2021-07-02
- 5.0.2: 2021-06-10
- 5.0.1: 2021-04-24
- 5.0.0: 2021-04-07
- 5.0.0-rc: 2021-01-12

16.1.14 4.0

- 4.0.16: 2021-12-17
- 4.0.15: 2021-09-27
- 4.0.14: 2021-07-27
- 4.0.13: 2021-05-28
- 4.0.12: 2021-04-02
- 4.0.11: 2021-02-26
- 4.0.10: 2021-01-15
- 4.0.9: 2020-12-21
- 4.0.8: 2020-10-30
- 4.0.7: 2020-09-29
- 4.0.6: 2020-09-15
- 4.0.5: 2020-08-31
- 4.0.4: 2020-07-31
- 4.0.3: 2020-07-24
- 4.0.2: 2020-07-01
- 4.0.1: 2020-06-12
- 4.0.0: 2020-05-28
- 4.0.0-rc.2: 2020-05-15
- 4.0.0-rc.1: 2020-04-28
- 4.0.0-rc: 2020-04-08
- 4.0.0-beta.2: 2020-03-18
- 4.0.0-beta.1: 2020-02-28
- 4.0.0-beta: 2020-01-17

16.1.15 3.1

- 3.1.2: 2020-06-04
- 3.1.1: 2020-04-30
- 3.1.0: 2020-04-16
- 3.1.0-rc: 2020-04-02
- 3.1.0-beta.2: 2020-03-09
- 3.1.0-beta.1: 2020-01-10
- 3.1.0-beta: 2019-12-20

16.1.16 3.0

- 3.0.20: 2020-12-25
- 3.0.19: 2020-09-25
- 3.0.18: 2020-08-21
- 3.0.17: 2020-08-03
- 3.0.16: 2020-07-03
- 3.0.15: 2020-06-05
- 3.0.14: 2020-05-09
- 3.0.13: 2020-04-22
- 3.0.12: 2020-03-16
- 3.0.11: 2020-03-04
- 3.0.10: 2020-02-20
- 3.0.9: 2020-01-14
- 3.0.8: 2019-12-31
- 3.0.7: 2019-12-04
- 3.0.6: 2019-11-28
- 3.0.5: 2019-10-25
- 3.0.4: 2019-10-08
- 3.0.3: 2019-08-29
- 3.0.2: 2019-08-07
- 3.0.1: 2019-07-16
- 3.0.0: 2019-06-28
- 3.0.0-rc.3: 2019-06-21
- 3.0.0-rc.2: 2019-05-28
- 3.0.0-rc.1: 2019-05-10
- 3.0.0-beta.1: 2019-03-26
- 3.0.0-beta: 2019-01-19

16.1.17 2.1

- 2.1.19: 2019-12-27
- 2.1.18: 2019-11-04
- 2.1.17: 2019-09-11
- 2.1.16: 2019-08-15

- 2.1.15: 2019-07-16
- 2.1.14: 2019-07-04
- 2.1.13: 2019-06-21
- 2.1.12: 2019-06-13
- 2.1.11: 2019-06-03
- 2.1.10: 2019-05-22
- 2.1.9: 2019-05-06
- 2.1.8: 2019-04-12
- 2.1.7: 2019-03-28
- 2.1.6: 2019-03-15
- 2.1.5: 2019-02-28
- 2.1.4: 2019-02-15
- 2.1.3: 2019-01-28
- 2.1.2: 2018-12-22
- 2.1.1: 2018-12-12
- 2.1.0: 2018-11-30
- 2.1.0-rc.5: 2018-11-12
- 2.1.0-rc.4: 2018-10-23
- 2.1.0-rc.3: 2018-09-29
- 2.1.0-rc.2: 2018-09-14
- 2.1.0-rc.1: 2018-08-24
- 2.1.0-beta: 2018-06-29

16.1.18 2.0

- 2.0.11: 2019-01-03
- 2.0.10: 2018-12-18
- 2.0.9: 2018-11-19
- 2.0.8: 2018-10-16
- 2.0.7: 2018-09-07
- 2.0.6: 2018-08-06
- 2.0.5: 2018-07-06
- 2.0.4: 2018-06-15
- 2.0.3: 2018-06-01
- 2.0.2: 2018-05-21
- 2.0.1: 2018-05-16
- 2.0.0: 2018-04-27
- 2.0.0-rc.5: 2018-04-17
- 2.0.0-rc.4: 2018-03-30
- 2.0.0-rc.3: 2018-03-23
- 2.0.0-rc.1: 2018-03-09
- 1.1.0-beta: 2018-02-24
- 1.1.0-alpha: 2018-01-19

16.1.19 1.0

- 1.0.8: 2018-02-11
- 1.0.7: 2018-01-22
- 1.0.6: 2018-01-08
- 1.0.5: 2017-12-26
- 1.0.4: 2017-12-11
- 1.0.3: 2017-11-28
- 1.0.2: 2017-11-13
- 1.0.1: 2017-11-01
- 1.0.0: 2017-10-16
- Pre-GA: 2017-08-30
- rc4: 2017-08-04
- rc3: 2017-06-16
- rc2: 2017-03-01
- rc1: 2016-12-23

16.2 TiDB 版本发布时间线

本文列出了所有已发布的 TiDB 版本，按发布时间倒序呈现。

版本	发布日期
6.1.7	2023-07-12
6.5.3	2023-06-14
6.5.2	2023-04-21
6.1.6	2023-04-12
7.0.0-DMR	2023-03-30
6.5.1	2023-03-10
6.1.5	2023-02-28
6.6.0-DMR	2023-02-20
6.1.4	2023-02-08
6.5.0	2022-12-29
5.1.5	2022-12-28
6.1.3	2022-12-05
5.3.4	2022-11-24
6.4.0-DMR	2022-11-17
6.1.2	2022-10-24
5.4.3	2022-10-13
6.3.0-DMR	2022-09-30
5.3.3	2022-09-14
6.1.1	2022-09-01
6.2.0-DMR	2022-08-23
5.4.2	2022-07-08
5.3.2	2022-06-29
6.1.0	2022-06-13

版本	发布日期
5.4.1	2022-05-13
5.2.4	2022-04-26
6.0.0-DMR	2022-04-07
5.3.1	2022-03-03
5.1.4	2022-02-22
5.4.0	2022-02-15
5.0.6	2021-12-31
4.0.16	2021-12-17
5.1.3	2021-12-03
5.0.5	2021-12-03
5.2.3	2021-12-03
5.3.0	2021-11-30
5.2.2	2021-10-29
5.1.2	2021-09-27
5.0.4	2021-09-27
4.0.15	2021-09-27
5.2.1	2021-09-09
5.2.0	2021-08-27
5.1.1	2021-07-30
4.0.14	2021-07-27
5.0.3	2021-07-02
5.1.0	2021-06-24
5.0.2	2021-06-10
4.0.13	2021-05-28
5.0.1	2021-04-24
5.0.0	2021-04-07
4.0.12	2021-04-02
4.0.11	2021-02-26
4.0.10	2021-01-15
5.0.0-rc	2021-01-12
3.0.20	2020-12-25
4.0.9	2020-12-21
4.0.8	2020-10-30
4.0.7	2020-09-29
3.0.19	2020-09-25
4.0.6	2020-09-15
4.0.5	2020-08-31
3.0.18	2020-08-21
3.0.17	2020-08-03
4.0.4	2020-07-31
4.0.3	2020-07-24
3.0.16	2020-07-03
4.0.2	2020-07-01

版本	发布日期
4.0.1	2020-06-12
3.0.15	2020-06-05
3.1.2	2020-06-04
4.0.0	2020-05-28
4.0.0-rc.2	2020-05-15
3.0.14	2020-05-09
3.1.1	2020-04-30
4.0.0-rc.1	2020-04-28
3.0.13	2020-04-22
3.1.0	2020-04-16
4.0.0-rc	2020-04-08
3.1.0-rc	2020-04-02
4.0.0-beta.2	2020-03-18
3.0.12	2020-03-16
3.1.0-beta.2	2020-03-09
3.0.11	2020-03-04
4.0.0-beta.1	2020-02-28
3.0.10	2020-02-20
4.0.0-beta	2020-01-17
3.0.9	2020-01-14
3.1.0-beta.1	2020-01-10
3.0.8	2019-12-31
2.1.19	2019-12-27
3.1.0-beta	2019-12-20
3.0.7	2019-12-04
3.0.6	2019-11-28
2.1.18	2019-11-04
3.0.5	2019-10-25
3.0.4	2019-10-08
2.1.17	2019-09-11
3.0.3	2019-08-29
2.1.16	2019-08-15
3.0.2	2019-08-07
3.0.1	2019-07-16
2.1.15	2019-07-16
2.1.14	2019-07-04
3.0.0	2019-06-28
3.0.0-rc.3	2019-06-21
2.1.13	2019-06-21
2.1.12	2019-06-13
2.1.11	2019-06-03
3.0.0-rc.2	2019-05-28
2.1.10	2019-05-22

版本	发布日期
3.0.0-rc.1	2019-05-10
2.1.9	2019-05-06
2.1.8	2019-04-12
2.1.7	2019-03-28
3.0.0-beta.1	2019-03-26
2.1.6	2019-03-15
2.1.5	2019-02-28
2.1.4	2019-02-15
2.1.3	2019-01-28
3.0.0-beta	2019-01-19
2.0.11	2019-01-03
2.1.2	2018-12-22
2.0.10	2018-12-18
2.1.1	2018-12-12
2.1.0	2018-11-30
2.0.9	2018-11-19
2.1.0-rc.5	2018-11-12
2.1.0-rc.4	2018-10-23
2.0.8	2018-10-16
2.1.0-rc.3	2018-09-29
2.1.0-rc.2	2018-09-14
2.0.7	2018-09-07
2.1.0-rc.1	2018-08-24
2.0.6	2018-08-06
2.0.5	2018-07-06
2.1.0-beta	2018-06-29
2.0.4	2018-06-15
2.0.3	2018-06-01
2.0.2	2018-05-21
2.0.1	2018-05-16
2.0.0	2018-04-27
2.0.0-rc.5	2018-04-17
2.0.0-rc.4	2018-03-30
2.0.0-rc.3	2018-03-23
2.0.0-rc.1	2018-03-09
1.1.0-beta	2018-02-24
1.0.8	2018-02-11
1.0.7	2018-01-22
1.1.0-alpha	2018-01-19
1.0.6	2018-01-08
1.0.5	2017-12-26
1.0.4	2017-12-11
1.0.3	2017-11-28

版本	发布日期
1.0.2	2017-11-13
1.0.1	2017-11-01
1.0.0	2017-10-16
Pre-GA	2017-08-30
rc4	2017-08-04
rc3	2017-06-16
rc2	2017-03-01
rc1	2016-12-23

16.3 TiDB 版本规则

建议始终升级到当前系列的最新补丁版本。

TiDB 提供两个版本系列：

- 长期支持版本
- 开发里程碑版本（自 TiDB v6.0.0 起引入）

关于对 TiDB 版本提供支持服务的标准和规则，参见 TiDB 版本周期支持策略。

16.3.1 版本命名

TiDB 版本的命名方式为 X.Y.Z。X.Y 代表一个版本系列。

- 从 TiDB 1.0 起，X 每年依次递增，X 的递增代表引入新的功能和改进。
- Y 从 0 开始依次递增，Y 的递增代表引入新的功能和改进。
- 一个版本系列首次发版时 Z 默认为 0，后续发补丁版本时 Z 从 1 开始依次递增。

TiDB v5.0.0 及其之前的版本命名规则，请查看历史版本。

16.3.2 长期支持版本

长期支持版本 (Long-Term Support Releases, LTS) 约每六个月发布一次，会引入新功能、改进、缺陷修复和安全漏洞修复。

LTS 命名方式为 X.Y.Z，Z 默认为 0。

示例版本:

- 6.1.0
- 5.4.0

在 LTS 生命周期内会按需发布补丁版本 (Patch Release)。补丁版本主要包含 bug 修复、安全漏洞修复，不会包含新的功能。

补丁版本命名方式为 X.Y.Z。其中，系列版本号 X.Y 与对应的 LTS 保持一致，补丁版本号 Z 从 1 开始依次递增。

示例版本:

- 6.1.1

v5.1.0、v5.2.0、v5.3.0、v5.4.0 发布周期仅间隔两个月，但均为 LTS，提供对应补丁版本。

16.3.3 开发里程碑版本

开发里程碑版本 (Development Milestone Releases, DMR) 约每两个月发布一次。如遇 LTS 发版，DMR 发版时间顺延两个月。DMR 会引入新的功能、改进和修复。但 TiDB 不提供基于 DMR 的补丁版本，如有相关 bug 会在后续版本系列中陆续修复。

DMR 命名方式为 X.Y.Z，Z 默认为 0，并添加后缀 -DMR。

示例版本:

- 6.0.0-DMR

16.3.4 TiDB 工具版本

一部分 TiDB 工具与 TiDB server 共同发布，使用相同的版本号体系，例如 TiDB Lightning。一部分 TiDB 工具与 TiDB server 分开发布，并使用独立的版本号体系，例如 TiUP 和 TiDB Operator。

16.3.5 不再沿用的历史版本号

16.3.5.1 正式发布版本

正式发布版本 (General Availability Releases, GA) 是 TiDB 当前系列版本的稳定版本，在候选发布版本 (Release Candidate Releases, RC) 之后发布，能够用于生产部署。

示例版本：

- 1.0
- 2.1 GA
- 5.0 GA

16.3.5.2 候选发布版本

候选发布版本 (Release Candidate Releases, RC) 会引入新的功能和改进。RC 版本可用于早期测试，较公开测试版本的稳定性有较大改善，其稳定性足以开始测试，但不适合用于生产部署。

示例版本：

- RC1
- 2.0-RC1
- 3.0.0-rc.1

16.3.5.3 公开测试版本

公开测试版本 (Beta Releases) 会引入新的功能和改进，相对于内部测试版本已有了很大的改进，消除了严重的错误，但还是存在着一些 bug，提供给尝鲜用户，可以用于测试最新的功能。

示例版本：

- 1.1 Beta
- 2.1 Beta
- 4.0.0-beta.1

16.3.5.4 内部测试版本

内部测试版本 (Alpha Releases) 是内部测试版，会引入新的功能和改进。Alpha 版是当前系列版本的最初版本。Alpha 版可能存在一些 bug，提供给尝鲜用户，可以用于测试最新的功能。

示例版本：

- 1.1 Alpha

16.4 TiDB 离线包

在使用 TiUP 离线部署 TiDB 前，你需要在官方下载页面选择对应版本的 TiDB server 离线镜像包（包含 TiUP 离线组件包）。

TiDB 提供了 amd64 和 arm64 两种架构的离线包。对于每种架构，TiDB 提供了两个二进制离线包：`TiDB-community-server` 软件包和 `TiDB-community-toolkit` 软件包。

`TiDB-community-server` 软件包中包含以下内容：

内容	变更说明
tidb-{version}-linux-{arch}.tar.gz	
tikv-{version}-linux-{arch}.tar.gz	
tiflash-{version}-linux-{arch}.tar.gz	
pd-{version}-linux-{arch}.tar.gz	
ctl-{version}-linux-{arch}.tar.gz	
grafana-{version}-linux-{arch}.tar.gz	
alertmanager-{version}-linux-{arch}.tar.gz	
blackbox_exporter-{version}-linux-{arch}.tar.gz	
prometheus-{version}-linux-{arch}.tar.gz	
node_exporter-{version}-linux-{arch}.tar.gz	
tiup-linux-{arch}.tar.gz	
tiup-{version}-linux-{arch}.tar.gz	
local_install.sh	
cluster-{version}-linux-{arch}.tar.gz	
insight-{version}-linux-{arch}.tar.gz	
diag-{version}-linux-{arch}.tar.gz	从 v6.0.0 起新增
influxdb-{version}-linux-{arch}.tar.gz	
playground-{version}-linux-{arch}.tar.gz	

> **注意：**
>
> 以上离线包名称中，`{version}` 取决于离线包中内容的版本号，`{arch}` 取决于离线包对应的架构（amd64 或 arm64）。

`TiDB-community-toolkit` 软件包中包含以下内容：

内容	变更说明
tikv-importer-{version}-linux-{arch}.tar.gz	
pd-recover-{version}-linux-{arch}.tar.gz	
etcdctl	从 v6.0.0 起新增
tiup-linux-{arch}.tar.gz	
tiup-{version}-linux-{arch}.tar.gz	
tidb-lightning-{version}-linux-{arch}.tar.gz	
tidb-lightning-ctl	
dumpling-{version}-linux-{arch}.tar.gz	
cdc-{version}-linux-{arch}.tar.gz	
dm-{version}-linux-{arch}.tar.gz	
dm-worker-{version}-linux-{arch}.tar.gz	
dm-master-{version}-linux-{arch}.tar.gz	
dmctl-{version}-linux-{arch}.tar.gz	
br-{version}-linux-{arch}.tar.gz	
spark-{version}-any-any.tar.gz	
tispark-{version}-any-any.tar.gz	
package-{version}-linux-{arch}.tar.gz	
bench-{version}-linux-{arch}.tar.gz	
errdoc-{version}-linux-{arch}.tar.gz	
dba-{version}-linux-{arch}.tar.gz	
PCC-{version}-linux-{arch}.tar.gz	
pump-{version}-linux-{arch}.tar.gz	
drainer-{version}-linux-{arch}.tar.gz	
binlogctl	从 v6.0.0 起新增
sync_diff_inspector	
reparo	
arbiter	
mydumper	从 v6.0.0 起新增
server-{version}-linux-{arch}.tar.gz	从 v6.2.0 起新增
grafana-{version}-linux-{arch}.tar.gz	从 v6.2.0 起新增
alertmanager-{version}-linux-{arch}.tar.gz	从 v6.2.0 起新增
prometheus-{version}-linux-{arch}.tar.gz	从 v6.2.0 起新增
blackbox_exporter-{version}-linux-{arch}.tar.gz	从 v6.2.0 起新增
node_exporter-{version}-linux-{arch}.tar.gz	从 v6.2.0 起新增

> **注意：**
>
> 以上离线包名称中，{version} 取决于离线包中工具的版本号，{arch} 取决于离线包对应的架构（amd64 或 arm64）。

16.4.1 延伸阅读

离线部署 TiDB 集群

16.5 v7.0

16.5.1 TiDB 7.0.0 Release Notes

发版日期：2023 年 3 月 30 日

TiDB 版本：7.0.0

试用链接：快速体验 | 下载离线包

在 7.0.0 版本中，你可以获得以下关键特性：

```
<th>分类</th>
<th>功能</th>
<th>描述</th>
```

```
<td rowspan="2">可扩展性与性能</td>
<td>会话级别内<a href="https://docs.pingcap.com/zh/tidb/v7.0/sql-non-prepared-plan-cache" target
    ↪ ="_blank">无需手动准备 SQL 执行计划缓存</a>（实验特性）</td>
<td>支持在会话级别自动重用执行计划缓存，可以减少编译并缩短相同 SQL 查询的时间，而无需事先手动准备
    ↪  Prepare Statement 语句。</td>
```

```
<td>TiFlash 支持<a href="https://docs.pingcap.com/zh/tidb/v7.0/tiflash-disaggregated-and-s3"
    ↪ target="_blank">存储计算分离和 S3 共享存储</a>（实验特性）</td>
<td>TiFlash 增加云原生架构的支持作为可选项：
  <ul>
    <li>支持存算分离架构，提升 HTAP 资源的弹性能力。</li>
    <li>支持基于 S3 的存储引擎，以更低的成本提供共享存储。</li>
  </ul>
</td>
```

```
<td rowspan="2">稳定性与高可用</td>
<td><a href="https://docs.pingcap.com/zh/tidb/v7.0/tidb-resource-control" target="_blank">
    ↪ 增强了资源管控</a>（实验特性）</td>
```

```
<td>支持使用资源组来为一个集群中的不同应用或工作负载分配和隔离资源。在这个版本中，TiDB
    ↪ 增加了对不同资源的绑定模式（用户级、会话级、语句级）和用户定义的优先级的支持，
    ↪ 你还可以使用命令来对集群整体资源量进行预估。</td>
```

```
<td>TiFlash 支持<a href="https://docs.pingcap.com/zh/tidb/v7.0/tiflash-spill-disk" target="_blank
    ↪ ">数据落盘</a></td>
<td>TiFlash 支持将中间结果落盘，以缓解数据密集型操作（如聚合、排序和 Hash Join）中的 OOM 问题。</
    ↪ td>
```

```
<td rowspan="2">SQL</td>
<td><a href="https://docs.pingcap.com/zh/tidb/v7.0/time-to-live" target="_blank">行级 TTL</a> (GA
    ↪ )</td>
<td>支持通过后台任务自动删除超过生命周期（Time to live）的数据，
    ↪ 并以此来自动管理数据规模并提高性能。</td>
```

```
<td><a href="https://docs.pingcap.com/zh/tidb/v7.0/partitioned-table#重组分区" target="_blank">
    ↪ 支持 <code>REORGANIZE PARTITION</code> 语法（List/Range 分区表）</a></td>
<td><code>REORGANIZE PARTITION</code> 语句可用于合并相邻分区，或将一个分区拆分为多个分区，
    ↪ 从而提升分区表的易用性。</td>
```

```
<td rowspan="2">数据库管理与可观测性<br/></td>
<td><a href="https://docs.pingcap.com/zh/tidb/v7.0/sql-statement-load-data" target="_blank"><code
    ↪ >LOAD DATA</code> 语句功能增强</a>（实验特性）</td>
<td><code>LOAD DATA</code> 语句功能增强，例如支持从 S3/GCS 导入数据。</td>
```

```
<td>TiCDC 支持<a href="https://docs.pingcap.com/zh/tidb/v7.0/ticdc-sink-to-cloud-storage" target
    ↪ ="_blank">对象存储 Sink</a> (GA)</td>
<td>TiCDC 支持将行变更事件同步到对象存储服务，包括 Amazon S3、GCS、Azure Blob Storage 和 NFS 等
    ↪ 。</td>
```

16.5.1.1 功能详情

16.5.1.1.1 可扩展性

- TiFlash 支持存算分离和对象存储（实验特性）#6882 @flowbehappy @JaySon-Huang @breezewish @JinheLin @lidezhu @CalvinNeo

  在 v7.0.0 之前的版本中，TiFlash 为存算一体架构。在此架构下，TiFlash 节点既是存储节点，也是计算节点，TiFlash 的计算能力和存储能力无法独立扩展；同时，TiFlash 节点只能使用本地存储。

  从 v7.0.0 起，TiFlash 新增存算分离架构。在此架构下，TiFlash 节点分为 Compute Node（计算节点）和 Write Node（写入节点）两种类型，并支持兼容 S3 API 的对象存储。这两种节点都可以单独扩缩容，独立调整计算或数据存储能力。TiFlash 的存算分离架构和存算一体架构不能混合使用、相互转换，需要在部署 TiFlash 时进行相应的配置指定使用其中的一种架构。

  更多信息，请参考用户文档。

16.5.1.1.2 性能

- 实现 Fast Online DDL 和 PITR 的兼容 #38045 @Leavrth

  TiDB v6.5.0 中Fast Online DDL 功能和PITR 未完全兼容。在使用 TiDB v6.5.0 时，建议先停止 PITR 后台备份任务，以 Fast Online DDL 方式快速添加索引，然后再启动 PITR 备份任务实现全量数据备份。

  从 TiDB v7.0.0 开始，Fast Online DDL 和 PITR 已经完全兼容。通过 PITR 恢复集群数据时，系统将自动回放日志备份期间记录的通过 Fast Online DDL 方式添加的索引操作，达到兼容效果。

  更多信息，请参考用户文档。

- TiFlash 引擎支持 Null-Aware Semi Join 和 Null-Aware Anti Semi Join 算子 #6674 @gengliqi

  当使用 `IN`、`NOT IN`、`= ANY` 或 `!= ALL` 算子引导的关联子查询时，TiDB 会将其转化为 Semi Join 或 Anti Semi Join，从而提升计算性能。如果转换后的 Join key 列可能为 `NULL`，则需要具有 Null-Aware 特性的 Join 算法，即Null-Aware Semi Join 和Null-Aware Anti Semi Join 算子。

  在 v7.0.0 之前的版本中，TiFlash 引擎不支持 Null-Aware Semi Join 和 Null-Aware Anti Semi Join 算子，因此无法将这些子查询直接下推至 TiFlash 引擎进行计算。从 TiDB v7.0.0 开始，TiFlash 引擎支持了 Null-Aware Semi Join 和 Null-Aware Anti Semi Join 算子。如果 SQL 包含这几种关联子查询，查询的表包含 TiFlash 副本，并且启用了MPP 模式，优化器将自动判断是否将 Null-Aware Semi Join 和 Null-Aware Anti Semi Join 算子下推至 TiFlash 引擎进行计算以提升整体性能。

  更多信息，请参考用户文档。

- TiFlash 引擎支持 FastScan 功能 (GA) #5252 @hongyunyan

  从 v6.3.0 开始，TiFlash 引擎引入了快速扫描功能 (FastScan) 作为实验特性。在 v7.0.0 中，该功能正式 GA。你可以使用系统变量`tiflash_fastscan` 启用快速扫描功能。通过牺牲强一致性保证，该功能可以大幅提升扫表性能。如果对应的表仅进行 `INSERT` 操作，而没有 `UPDATE`/`DELETE` 操作，则快速扫描功能在提升扫表性能的同时不会损失强一致性。

  更多信息，请参考用户文档。

- TiFlash 查询支持延迟物化功能（实验特性）#5829 @Lloyd-Pottiger

  当 `SELECT` 语句中包含过滤条件（`WHERE` 子句）时，TiFlash 默认会先读取该查询所需列的全部数据，然后再根据查询条件对数据进行过滤、聚合等计算任务。延迟物化是一种优化方式，它支持下推部分过滤条件到 TableScan 算子，即先扫描过滤条件相关的列数据，过滤得到符合条件的行后，再扫描这些行的其他列数据，继续后续计算，从而减少 IO 扫描和数据处理的计算量。

  TiFlash 延迟物化默认关闭，可以通过将系统变量`tidb_opt_enable_late_materialization` 设置为 `ON` 开启。开启后，TiDB 优化器会根据统计信息和查询的过滤条件，决定哪些过滤条件会被下推到 TableScan 算子。

  更多信息，请参考用户文档。

- 支持缓存非 Prepare 语句的执行计划（实验特性）#36598 @qw4990

  执行计划缓存是提高并发 OLTP 负载能力的重要手段，TiDB 已经支持Prepare 语句的计划缓存。在 v7.0.0 中，非 Prepare 语句的执行计划也能够被缓存，使执行计划缓存能够应用于更广泛的场景，从而提升 TiDB 的并发处理能力。

  这个功能目前默认关闭，你可以通过系统变量`tidb_enable_non_prepared_plan_cache` 打开。出于稳定性考虑，在当前版本中，TiDB 开辟了一块新的区域用于缓存非 Prepare 语句的执行计划，你可以通过系

统变量tidb_non_prepared_plan_cache_size 设置缓存大小。此外，该功能对 SQL 的模式也有一定的限制，具体参见使用限制。

更多信息，请参考用户文档。

- 解除执行计划缓存对子查询的限制 #40219 @fzzf678

  TiDB v7.0.0 移除了计划缓存对子查询的限制。带有子查询的 SQL 语句的执行计划可以被缓存，例如 `SELECT * FROM t WHERE a > (SELECT ...)`。这进一步扩大了执行计划缓存的应用范围，提高了 SQL 的执行效率。

  更多信息，请参考用户文档。

- TiKV 支持自动生成空的日志文件用于日志回收 #14371 @LykxSassinator

  TiKV 在 v6.3.0 中引入了Raft 日志回收特性，用以减少写负载的长尾延迟。但是，日志回收需要在 Raft 日志文件数量达到一定阈值后才能生效，使得用户无法直观感受到该特性对写负载吞吐的提升。

  为了改善用户体验，v7.0.0 正式引入了`raft-engine.prefill-for-recycle` 配置项，用以控制 TiKV 是否在进程启动时自动生成空的日志文件用于日志回收。该配置项启用时，TiKV 将在初始化时自动填充一批空日志文件用于日志回收，保证日志回收在初始化后立即生效。

  更多信息，请参考用户文档。

- 支持从窗口函数中推导 TopN 或 Limit 的优化规则，提升窗口函数的性能 #13936 @windtalker

  该功能默认关闭，需要将 session 变量tidb_opt_derive_topn 设置为 ON 开启。

  更多信息，请参考用户文档。

- 支持通过 Fast Online DDL 创建唯一索引 #40730 @tangenta

  TiDB v6.5.0 支持通过Fast Online DDL 方式创建普通的二级索引。从 v7.0.0 开始，TiDB 支持通过 Fast Online DDL 方式创建唯一索引。相比于 TiDB v6.1.0，大表添加唯一索引的性能预期将提升数倍。

  更多信息，请参考用户文档。

16.5.1.1.3 稳定性

- 增强了资源管控特性 (实验特性) #38825 @nolouch @BornChanger @glorv @tiancaiamao @Connor1996 @JmPotato @hnes @CabinfeverB @HuSharp

  TiDB 优化了基于资源组的资源管控特性。该特性将会极大地提升 TiDB 集群的资源利用效率和性能表现。资源管控特性的引入对 TiDB 具有里程碑的意义，你可以将一个分布式数据库集群划分成多个逻辑单元，将不同的数据库用户映射到对应的资源组中，并根据需要设置每个资源组的配额。当集群资源紧张时，来自同一个资源组的会话所使用的全部资源将被限制在配额内，避免其中一个资源组过度消耗，从而影响其他资源组中的会话正常运行。

  该特性也可以将多个来自不同系统的中小型应用合入一个 TiDB 集群中，个别应用的负载提升，不会影响其他应用的正常运行。而在系统负载较低的时候，繁忙的应用即使超过设定的读写配额，也仍然可以被分配到所需的系统资源，达到资源的最大化利用。此外，合理利用资源管控特性可以减少集群数量，降低运维难度及管理成本。

  该特性不仅提供了 Grafana 内置的 Resource Control Dashboard 展示资源的实际使用情况，协助你更合理地配置资源，还支持基于会话和语句级别（Hint）的动态资源管控能力。这些功能的引入将帮助你更精确地掌控 TiDB 集群的资源使用情况，并根据实际需要动态调整配额。

在 v7.0.0 中，支持为资源组设置绝对的调度优先级 (PRIORITY)，保障重要业务能够获取到资源，同时也扩展了资源组的设置方式。

你可以通过以下方式使用资源组：

- 用户级别。通过`CREATE USER`或`ALTER USER`语句将用户绑定到特定的资源组。绑定后，对应的用户新创建的会话会自动绑定对应的资源组。
- 会话级别。通过`SET RESOURCE GROUP`设置当前会话的资源组。
- 语句级别。通过`RESOURCE_GROUP()`设置当前语句的资源组。

更多信息，请参考用户文档。

- 支持 Fast Online DDL 的检查点机制，提升容错性和自动恢复能力 #42164 @tangenta

  TiDB v7.0.0 引入Fast Online DDL 的检查点机制，大幅提升 Fast Online DDL 的容错性和自动恢复能力。通过周期性记录并同步 DDL 进度，即使在 TiDB DDL Owner 故障重启或切换时，正在执行中的 DDL 仍能以 Fast Online DDL 方式继续执行，从而让 DDL 执行更加稳定高效。

  更多信息，请参考用户文档。

- TiFlash 引擎支持数据落盘 (Spill to disk) #6528 @windtalker

  为了提高执行性能，TiFlash 引擎尽可能将数据全部放在内存中运行。当数据量超过内存总大小时，TiFlash 会终止查询，避免内存耗尽导致系统崩溃。因此，TiFlash 可处理的数据量受限于可用的内存大小。

  从 v7.0.0 开始，TiFlash 引擎支持数据落盘功能，通过调整算子内存使用阈值`tidb_max_bytes_before_tiflash_external_grou`
  ↪ 、`tidb_max_bytes_before_tiflash_external_sort`、`tidb_max_bytes_before_tiflash_external_join`，控制对应算子的最大内存使用量。当算子使用内存超过一定阈值时，会自动将数据落盘，牺牲一定的性能，从而处理更多数据。

  更多信息，请参考用户文档。

- 提升统计信息的收集效率 #41930 @xuyifangreeneyes

  在 v7.0.0 中，TiDB 进一步优化了统计信息收集的逻辑，收集时间降低了约 25%。这一优化提升了中大型数据库集群的运行效率和稳定性，减少了统计信息收集对集群性能的影响。

- 新增优化器 Hint 对 MPP 进行干预 #39710 @Reminiscent

  TiDB 在 v7.0.0 中增加了一系列优化器 Hint，来影响 MPP 操作执行计划的生成。

  - `SHUFFLE_JOIN()`：针对 MPP 生效。提示优化器对指定表使用 Shuffle Join 算法。
  - `BROADCAST_JOIN()`：针对 MPP 生效。提示优化器对指定表使用 Broadcast Join 算法。
  - `MPP_1PHASE_AGG()`：针对 MPP 生效。提示优化器对指定查询块中所有聚合函数使用一阶段聚合算法。
  - `MPP_2PHASE_AGG()`：针对 MPP 生效。提示优化器对指定查询块中所有聚合函数使用二阶段聚合算法。

MPP 优化器 Hint 能够协助你干预 HTAP 查询，提升 HTAP 负载下的性能和稳定性。

更多信息，请参考用户文档。

- 优化器 Hint 可兼容指定连接方式与连接顺序 #36600 @Reminiscent

在 v7.0.0 中，优化器 Hint LEADING() 能够和影响连接方式的 Hint 配合使用，两者行为兼容。在多表连接的情况下，可以有效指定最佳的连接方式和连接顺序，提升优化器 Hint 对执行计划的控制能力。

新的 Hint 行为会有微小的变化。为确保向前兼容，TiDB 引入了变量tidb_opt_advanced_join_hint。当此变量为 OFF 时，行为向前兼容。从旧版本升级到 v7.0.0 及之后版本的集群，该变量会被设置成 OFF。为了获取更灵活的 Hint 行为，强烈建议在确保无性能回退的情况下，将该变量切换为 ON。

更多信息，请参考用户文档。

16.5.1.1.4 高可用

- TiDB 支持 prefer-leader 选项，在网络不稳定的情况下提供更高的读可用性，降低响应延迟 #40905 @LykxSassinator

  你可以通过系统变量tidb_replica_read 控制 TiDB 的数据读取行为。在 v7.0.0 中，该变量新增 prefer-↪ leader 选项。当设置为 prefer-leader 时，TiDB 会优先选择 leader 副本执行读取操作。当 leader 副本的处理速度明显变慢时，例如由于磁盘或网络性能抖动，TiDB 将选择其他可用的 follower 副本来执行读取操作，从而提高可用性并降低响应延迟。

  更多信息，请参考用户文档。

16.5.1.1.5 SQL 功能

- Time to live (TTL) 已基本可用 (GA) #39262 @lcwangchao @YangKeao

  TTL 提供了行级别的生命周期控制策略。在 TiDB 中，设置了 TTL 属性的表会根据配置自动检查并删除过期的行数据。TTL 的目标是帮助用户周期性且及时地清理不需要的数据，并尽量减少对用户负载的影响。

  更多信息，请参考用户文档。

- 支持 ALTER TABLE ... REORGANIZE PARTITION #15000 @mjonss

  TiDB 支持 ALTER TABLE... REORGANIZE PARTITION 语法。通过该语法，你可以对表的部分或所有分区进行重新组织，包括合并、拆分、或者其他修改，并且不丢失数据。

  更多信息，请参考用户文档。

- 支持 Key 分区 #41364 @TonsnakeLin

  TiDB 支持 Key 分区。 Key 分区与 Hash 分区都可以保证将数据均匀地分散到一定数量的分区里面，区别是 Hash 分区只能根据一个指定的整数表达式或字段进行分区，而 Key 分区可以根据字段列表进行分区，且 Key 分区的分区字段不局限于整数类型。

  更多信息，请参考用户文档。

16.5.1.1.6 数据库管理

- TiCDC 支持同步变更数据至存储服务 (GA) #6797 @zhaoxinyu

  TiCDC 支持将变更数据同步到兼容 Amazon S3 协议的存储服务、GCS、Azure Blob Storage 以及 NFS 中。存储服务价格便宜，使用方便。对于不使用 Kafka 的用户，可以选择同步变更数据到存储服务。使用该功能，TiCDC 会将变更数据保存到文件，发送到存储服务中。用户自研的消费程序可以定时从存储服务读取新产生的变更数据进行数据处理。目前，TiCDC 支持将格式为 canal-json 和 CSV 的变更数据同步至存储服务。

  更多信息，请参考用户文档。

- TiCDC OpenAPI v2 #8019 @sdojjy

  TiCDC 提供 OpenAPI v2。相比 OpenAPI v1，OpenAPI v2 提供了更完整的同步任务支持。OpenAPI 提供的功能是cdc cli 工具的一个子集。你可以通过 OpenAPI v2 对 TiCDC 集群进行查询和运维操作，如获取 TiCDC 节点状态、检查集群健康状态、管理同步任务等。

  更多信息，请参考用户文档。

- DBeaver v23.0.1 默认支持 TiDB #17396 @Icemap

  - 提供独立的 TiDB 模块、Icon 和标识。
  - 默认配置支持 TiDB Serverless，你可以更方便地连接 TiDB Serverless。
  - 支持识别 TiDB 版本，从而显示或隐藏外键 Tab。
  - 支持 Explain SQL 计划显示。
  - 支持 TiDB 语法高亮，如 PESSIMISTIC、OPTIMISTIC、AUTO_RANDOM、PLACEMENT、POLICY、REORGANIZE、EXCHANGE、CACHE、NONCLUSTERED、CLUSTERED 等。
  - 支持 TiDB 函数高亮，如 TIDB_BOUNDED_STALENESS、TIDB_DECODE_KEY、TIDB_DECODE_PLAN、TIDB_IS_DDL_OWNER、TIDB_PARSE_TSO、TIDB_VERSION、TIDB_DECODE_SQL_DIGESTS、TIDB_SHARD 等。

更多信息，请参考 DBeaver 用户文档。

16.5.1.1.7 数据迁移

- 增强 LOAD DATA 语句功能，支持导入云存储中的数据（实验特性）#40499 @lance6716

  之前，LOAD DATA 语句只能用于导入客户端的数据文件，如果你需要从云存储导入数据，不得不借助 TiDB Lightning 来实现。但是单独部署 TiDB Lightning 又会带来额外的部署成本和管理成本。现在可直接通过LOAD DATA 语句导入云存储中的数据。功能举例说明如下：

  - 支持从 Amazon S3 和 Google Cloud Storage 导入数据到 TiDB，且支持使用通配符一次性匹配多个源文件导入到 TiDB。
  - 支持 DEFINED NULL BY 来定义 null。
  - 支持 CSV、TSV 格式的源文件。

更多信息，请参考用户文档。

- TiDB Lightning 向 TiKV 传输键值对时支持启用压缩传输 #41163 @gozssky

  自 v7.0.0 起，TiDB Lightning 正式支持将本地编码排序后的键值对在网络传输时进行压缩再发送到 TiKV，从而减少网络传输的数据量，降低 50% ~ 80% 网络带宽开销。在 v6.6.0 版本之前不支持该功能，在数据量较大的情况下，TiDB Lightning 对网络带宽要求相对较高，且会产生较高的流量费。相比 v6.6.0, v7.0.0 优化了压缩算法，能以更快的速度将数据导入到 TiKV 中。同样是开启压缩选项，v6.6.0 会增加 2 倍导入时间，而 v7.0.0 只会增加大约 60% 时间。以上的压缩率和导入时长仅供参考，不同的使用场景会有差异。

  该功能默认关闭，你可以通过将 TiDB Lightning 配置项 compress-kv-pairs 设置为 "gzip" 或者 "gz" 开启此功能。

  更多信息，请参考用户文档。

16.5.1.2 兼容性变更

> **注意：**
>
> 以下为从 v6.6.0 升级至当前版本 (v7.0.0) 所需兼容性变更信息。如果从 v6.5.0 或之前版本升级到当前版本，可能也需要考虑和查看中间版本 release notes 中提到的兼容性变更信息。

16.5.1.2.1 MySQL 兼容性

- TiDB 移除了自增列必须是索引的约束 #40580 @tiancaiamao

  在 v7.0.0 之前，TiDB 自增列的行为与 MySQL 一致，要求自增列必须是索引或索引前缀。自 TiDB v7.0.0 起，TiDB 移除了自增列必须是索引或索引前缀的限制。这意味着用户现在可以更灵活地定义表的主键，并方便地使用自增列实现排序和分页，同时避免自增列带来的写入热点问题，并通过使用聚簇索引表提升查询性能。现在，你可以使用以下语法创建表：

  ```
  CREATE TABLE test1 (
      `id` int(11) NOT NULL AUTO_INCREMENT,
      `k` int(11) NOT NULL DEFAULT '0',
      `c` char(120) COLLATE utf8mb4_unicode_ci NOT NULL DEFAULT '',
      PRIMARY KEY(`k`, `id`)
  );
  ```

  此功能不影响 TiCDC 同步数据。

  更多信息，请参考用户文档。

- TiDB 支持 Key 分区类型，如下所示。

  ```
  CREATE TABLE employees (
  id INT NOT NULL,
  fname VARCHAR(30),
  lname VARCHAR(30),
  hired DATE NOT NULL DEFAULT '1970-01-01',
  separated DATE DEFAULT '9999-12-31',
  job_code INT,
  store_id INT) PARTITION BY KEY(store_id) PARTITIONS 4;
  ```

  自 TiDB v7.0.0 起，TiDB 支持 Key 分区，并支持解析 MySQL 的 `PARTITION BY LINEAR KEY` 语法，但会忽略其中的 `LINEAR` 关键字，只采用非线性 Hash 算法。Key 分区类型暂不支持分区字段列表为空的语句。

  更多信息，请参考用户文档。

16.5.1.2.2 行为变更

- TiCDC 修复了 Avro 编码 FLOAT 类型数据错误的问题 #8490 @3AceShowHand

  在升级 TiCDC 集群到 v7.0.0 时，如果使用 Avro 同步的表包含 FLOAT 类型数据，请在升级前手动调整 Confluent Schema Registry 的兼容性策略为 None，使 changefeed 能够成功更新 schema。否则，在升级之后 changefeed 将无法更新 schema 并进入错误状态。

- 自 v7.0.0 起，`tidb_dml_batch_size` 对`LOAD DATA` 语句不再生效。

16.5.1.2.3 系统变量

变量名	修改类型	描述
tidb_pessimistic_txn_aggressive_locking ↪	删除	更名为tidb_pessimistic_txn_fair_locking ↪ 。
tidb_enable_non_prepared_plan_cache ↪	修改	该变量从 v7.0.0 开始生效，用来控制是否开启非 Prepare 语句执行计划缓存。

变量名	修改类型	描述
tidb_enable_null_aware_anti_join ↪	修改	经进一步的测试后，该变量默认值从 OFF 修改为 ON，表示 TiDB 默认开启对特殊集合算子 NOT IN 和 != ALL 引导的子查询产生的 Anti Join 采用 Null-Aware Hash Join 的执行方式。

变量名	修改类型	描述
tidb_enable_resource_control ↪	修改	该变量默认值从 OFF 修改为 ON，表示默认开启集群按照资源组做资源隔离。资源管控功能在 v7.0.0 默认开启，便于你随时使用此功能。
tidb_non_prepared_plan_cache_size ↪	修改	该变量从 v7.0.0 开始生效，用来控制非 Prepare 语句执行计划缓存最多能够缓存的计划数量。

变量名	修改类型	描述
tidb_rc_read_check_ts	修改	从 v7.0.0 开始，该变量对于使用 Prepared Statement 协议下 cursor fetch read 游标模式不再生效。
tidb_enable_inl_join_inner_multi_pattern	新增	该变量用于控制当内表上有 Selection /Projection 算子时是否支持 Index Join。
tidb_enable_plan_cache_for_subquery	新增	该变量用于控制 Prepared Plan Cache 是否缓存包含子查询的查询。

变量名	修改类型	描述
tidb_enable_plan_replayer_continuous_capture ↪	新增	这个变量用来控制是否开启PLAN ↪ REPLAYER ↪ ↪ CONTINUOUS ↪ ↪ CAPTURE ↪ 功能。默认值 OFF 代表关闭功能。
tidb_load_based_replica_read_threshold ↪	新增	该变量用于设置基于负载的 replica read 的触发阈值。在 v7.0.0，该变量控制的功能尚未完全生效，请保留默认值。

变量名	修改类型	描述
tidb_opt_advanced_join_hint ↪	新增	这个变量用来控制用于控制连接算法的 Join Method Hint 是否会影响 Join Reorder 的优化过程。默认值为 ON，即采用新的兼容控制模式；OFF 则与 v7.0.0 以前的行为保持一致。为了向前兼容，从旧版本升级到 v7.0.0 及之后版本的集群，该变量会被设置成 OFF。

变量名	修改类型	描述
tidb_opt_derive_topn ↪	新增	这个变量用来控制是否开启从窗口函数中推导 TopN 或 Limit 的优化规则。默认值为 OFF，即未开启该优化规则。
tidb_opt_enable_late_materialization ↪	新增	这个变量用来控制是否启用 TiFlash 延迟物化功能。默认值为 OFF，即未开启 TiFlash 延迟物化功能。

变量名	修改类型	描述
tidb_opt_ordering_index_selectivity_threshold ↪	新增	该变量用于当 SQL 中存在 ORDER ↪ BY 和 LIMIT 子句且带有过滤条件时，控制优化器选择索引的行为。
tidb_pessimistic_txn_fair_locking ↪	新增	是否对悲观锁启用加强的悲观锁唤醒模型，以降低单行冲突场景下事务的尾延迟。默认值为 ON，从旧版本升级到 v7.0.0 或之后版本，该变量会被设置成 OFF。

变量名	修改类型	描述
tidb_ttl_running_tasks ↪	新增	这个变量用于限制整个集群内 TTL 任务的并发量。默认值 -1 表示与 TiKV 节点的数量相同。

16.5.1.2.4 配置文件参数

配置文件	配置项	修改类型	描述
TiKV	server. ↪ snap- ↪ max- ↪ write- ↪ bytes- ↪ per- ↪ sec	删除	更名为server. ↪ snap- ↪ io-max ↪ -bytes ↪ -per- ↪ sec。
TiKV	raft- ↪ engine ↪ . ↪ enable ↪ -log- ↪ recycle ↪	修改	默认值由 false 变更为 true。

配置文件	配置项	修改类型	描述
TiKV	resolved-ts.advance-ts-interval	修改	默认值由 "1s" 变更为 "20s"。该修改可以延长定期推进 Resolved TS 的时间间隔，从而减少 TiKV 节点之间的流量消耗。
TiKV	resource-control.enabled	修改	默认值由 false 变更为 true。
TiKV	raft-engine.prefill-for-recycle	新增	控制 Raft Engine 是否自动生成空的日志文件用于日志回收。默认值为 false。
PD	degraded-mode-wait-duration	新增	PD 中内置的Resource Control 相关配置项。用于配置触发降级模式需要等待的时间。默认值为 "0s"。

配置文件	配置项	修改类型	描述
PD	read-base-cost	新增	PD 中内置的 Resource Control 相关配置项。用于设置每次读请求转换成 RU 的基准系数。默认值为 0.25。
PD	read-cost-per-byte	新增	PD 中内置的 Resource Control 相关配置项。用于设置读流量转换成 RU 的基准系数。默认值为 1/(64 * 1024)。
PD	read-cpu-ms-cost	新增	PD 中内置的 Resource Control 相关配置项。用于设置 CPU 转换成 RU 的基准系数。默认值为 1/3。
PD	write-base-cost	新增	PD 中内置的 Resource Control 相关配置项。用于设置每次写请求转换成 RU 的基准系数。默认值为 1。

配置文件	配置项	修改类型	描述
PD	write- ↪ cost- ↪ per- ↪ byte	新增	PD 中内置的Resource Control 相关配置项。用于设置写流量转换成 RU 的基准系数。默认值为 1/1024。
TiFlash	flash. ↪ disaggregated_mode ↪	新增	在 TiFlash 的存算分离架构中，表示此 TiFlash 节点是 Write Node 还是 Compute Node。可选值为 tiflash_write ↪ 或者 tiflash_compute ↪ 。
TiFlash	storage. ↪ s3. ↪ endpoint ↪	新增	S3 的 endpoint 地址。
TiFlash	storage. ↪ s3. ↪ bucket	新增	TiFlash 的所有数据存储的 bucket。
TiFlash	storage. ↪ s3. ↪ root	新增	S3 bucket 中数据存储的根目录。
TiFlash	storage. ↪ s3. ↪ access_key_id ↪	新增	访问 S3 的 AC-CESS_KEY_ID。

配置文件	配置项	修改类型	描述
TiFlash	storage. ↪ s3. ↪ secret_access_key ↪	新增	访问 S3 的 SECRET_ACCESS_KEY。
TiFlash	storage. ↪ remote ↪ .cache ↪ .dir	新增	TiFlash Compute Node 的本地数据缓存目录。
TiFlash	storage. ↪ remote ↪ .cache ↪ . ↪ capacity ↪	新增	TiFlash Compute Node 的本地数据缓存目录的大小。

配置文件	配置项	修改类型	描述
TiDB Lightning	`add-index-by-sql`	新增	控制 Physical Import Mode 是否通过 SQL 方式添加索引。默认为 `false`，表示 TiDB Lightning 会将行数据以及索引数据都编码成 KV pairs 后一同导入 TiKV，实现机制和历史版本保持一致。通过 SQL 方式添加索引的优点是将导入数据与导入索引分开，可以快速导入数据，即使导入数据后，索引添加失败，也不会影响数据的一致性。

配置文件	配置项	修改类型	描述
TiCDC	enable-table-across-nodes	新增	将表按 Region 个数划分成多个同步范围，这些范围可由多个 TiCDC 节点同步。
TiCDC	region-threshold	新增	开启了 enable-table-across-nodes 后，该功能只对 Region 个数大于 region-threshold 值的表生效。
DM	analyze	新增	配置是否在 CHECKSUM 结束后对所有表逐个执行 ANALYZE TABLE <table> 操作，可配置 "required" "/" optional "/"off" 。默认为 "optional "。

配置文件	配置项	修改类型	描述
DM	range-concurrency	新增	配置 dm-worker 向 TiKV 写入 KV 数据的并发数。
DM	compress-kv-pairs	新增	配置 dm-worker 向 TiKV 发送 KV 数据时是否启用压缩，可配置 "gzip"，默认为空表示不压缩。
DM	pd-addr	新增	配置 Physical Import 时连接下游 PD server 的地址，填一个或多个均可。配置项为空时，默认使用 TiDB 中查询到的 PD 地址信息。

16.5.1.3 改进提升

- TiDB
  - 引入 EXPAND 算子辅助优化单个 SELECT 中包含多个 DISTINCT 的 SQL 性能 #16581 @AilinKid
  - Index Join 支持更多的 SQL 格式 #40505 @Yisaer
  - 避免某些情况下分区表数据需要在 TiDB 全局排序 #26166 @Defined2014
  - 支持同时使用 `fair lock mode` 和 `lock only if exists` 功能 #42068 @MyonKeminta
  - 支持打印事务慢日志以及相关事务内部事件 #41863 @ekexium
  - 支持 ILIKE 操作符 #40943 @xzhangxian1008

- PD
  - 新增监控指标以展示 store limit 限制导致的调度生成失败 #6043 @nolouch
- TiFlash
  - 减少 TiFlash 在写路径上的内存使用量 #7144 @hongyunyan
  - 减少 TiFlash 在有较多表的情况下的重启时间 #7146 @hongyunyan
  - 支持下推 `ILIKE` 操作符 #6740 @xzhangxian1008
- Tools
  - TiCDC
    * 支持在 Kafka 为下游的场景中将单个大表的数据变更分布到多个 TiCDC 节点，从而解决大规模 TiDB 集群的数据集成场景下单表扩展性的问题 #8247 @overvenus
      你可以通过设置 TiCDC 配置项 `enable_table_across_nodes` 为 `true` 来启用这个功能，并通过设置 `region_threshold` 来指定当一张表的 region 个数超过阈值时，TiCDC 开始将对应的表上的数据变更分布到多个 TiCDC 节点。
    * 支持在 redo applier 中拆分事务以提升 apply 吞吐，降低灾难场景的 RTO #8318 @CharlesCheung96
    * 改进表的调度策略，可以将单个表更均匀地拆分到各个 TiCDC 节点上 #8247 @overvenus
    * 在 MQ sink 中添加了 Large Row 监控指标 #8286 @hi-rustin
    * 在一个 Region 包含多个表的数据的场景下，减少了 TiKV 与 TiCDC 节点间的网络流量 #6346 @overvenus
    * 将 Checkpoint TS 和 Resolved TS 的 P99 指标的面板移动到 Lag analyze 面板 #8524 @hi-rustin
    * 支持在 redo log 里 apply DDL 事件 #8361 @CharlesCheung96
    * 支持根据上游写入吞吐来拆分调度表到 TiCDC 节点 #7720 @overvenus
  - TiDB Lightning
    * TiDB Lightning 的物理导入模式 (Physical Import Mode) 支持导入数据和索引分离导入，提升导入速度和稳定性 #42132 @gozssky
      TiDB Lightning 增加 `add-index-by-sql` 参数。默认值为 `false`，表示仍然会用 TiDB Lightning 将行数据以及索引数据编码成 KV pairs 后再一同导入到 TiKV。如果设置为 `true`，表示在物理导入模式 (Physical Import Mode) 下，会在导入数据完成后，通过 `ADD INDEX` 的 SQL 语句帮你建索引，提升导入数据的速度和稳定性。
    * TiDB Lightning 增加 `tikv-importer.keyspace-name` 参数。默认值为空字符串，表示 TiDB Lightning 会去自动获取这次导入对应的 keyspace 名字。如果指定了值，那么使用指定的 keyspace 名字来导入。这个参数使得 TiDB Lightning 导入多租户的 TiDB 集群场景下可以进行灵活配置。#41915 @lichunzhu

16.5.1.4 错误修复

- TiDB
  - 修复 TiDB 从 v6.5.1 升级到新版本时遗漏部分更新的问题 #41502 @chrysan
  - 修复部分系统变量升级后默认值没有修改的问题 #41423 @crazycs520
  - 修复加索引相关 Coprocessor 请求类型显示为 unknown 的问题 #41400 @tangenta

  - 修复加索引时报 PessimisticLockNotFound 错误的问题 #41515 @tangenta
  - 修复加唯一索引时误报 found duplicate key 错误的问题 #41630 @tangenta
  - 修复加索引时 panic 的问题 #41880 @tangenta
  - 修复 TiFlash 执行中遇到生成列会报错的问题 #40663 @guo-shaoge
  - 修复当存在时间类型时，TiDB 可能无法正确获取统计信息的问题 #41938 @xuyifangreeneyes
  - 修复在开启 Prepared Plan Cache 的情况下，索引全表扫可能会报错的问题 #42150 @fzzf678
  - 修复 IFNULL(NOT NULL COLUMN, ...) 可能返回错误结果的问题 #41734 @LittleFall
  - 修复当分区表的所有数据落在单个 Region 时，TiDB 可能执行得到错误结果的问题 #41801 @Defined2014
  - 修复当同一个 SQL 中出现多个不同的分区表时，TiDB 可能执行得到错误结果的问题 #42135 @mjonss
  - 修复在为分区表添加新的索引之后，该分区表可能无法正确触发统计信息的自动收集的问题 #41638 @xuyifangreeneyes
  - 修复在连续两次收集统计信息后，TiDB 可能读取到错误的列统计信息的问题 #42073 @xuyifangreeneyes
  - 修复在开启 Prepared Plan Cache 时 Index Merge 可能得到错误结果的问题 #41828 @qw4990
  - 修复 IndexMerge 中 goroutine 泄露的问题 #41605 @guo-shaoge
  - 修复非 BIGINT 类型的无符号整数与 STRING/DECIMAL 比较时可能出现错误结果的问题 #41736 @LittleFall
  - 修复 ANALYZE 语句可能会因为当前 session 的前一个 ANALYZE 语句因为内存超限被 kill 导致当前 ANALYZE 语句也被 kill 的问题 #41825 @XuHuaiyu
  - 修复 batch coprocessor 搜集信息过程中存在的数据竞争问题 #41412 @you06
  - 修复 assertion 错误导致无法为分区表打印 MVCC 信息的问题 #40629 @ekexium
  - 修复 fair lock mode 对不存在的 key 加锁处理的问题 #41527 @ekexium
  - 修复 INSERT IGNORE 和 REPLACE 语句对不修改 value 的 key 没有加锁的问题 #42121 @zyguan

- PD

  - 修复了 Scatter Region 操作后 leader 分布不均衡的问题 #6017 @HunDunDM
  - 修复了启动过程中获取 PD 成员时的数据竞争问题 #6069 @rleungx
  - 修复了热点统计信息中的数据竞争问题 #6069 @lhy1024
  - 修复了切换 Placement Rule 时可能存在的 leader 分布不均衡的问题 #6195 @bufferflies

- TiFlash

  - 修复了 Decimal 除法在某些情况下最后一位未进位的问题 #7022 @LittleFall
  - 修复了 Decimal 转换在某些情况下进位错误的问题 #6994 @windtalker
  - 修复了开启 new collation 后 TopN/Sort 算子结果可能出错的问题 #6807 @xzhangxian1008
  - 修复了单个 TiFlash 节点聚合结果集过大（超过 1200 万）时可能会导致 TiFlash 报错的问题 #6993 @windtalker

- Tools

  - Backup & Restore (BR)

    * 修复了在 PITR 恢复过程中等待 split Region 重试的时间不足的问题 #42001 @joccau
    * 修复了在 PITR 恢复过程遇到 memory is limited 错误导致恢复失败的问题 #41983 @joccau
    * 修复了 PD 节点宕机可能导致 PITR 日志备份进度不推进的问题 #14184 @YuJuncen
    * 缓解了 Region leadership 迁移导致 PITR 日志备份进度延迟变高的问题 #13638 @YuJuncen

- TiCDC
  - 修复了重启 changefeed 可能导致数据丢失或者 checkpoint 无法推进的问题 #8242 @overvenus
  - 修复了 DDL sink 出现的数据竞争问题 #8238 @3AceShowHand
  - 修复了状态为 `stopped` 的 changefeed 可能会自动重启的问题 #8330 @sdojjy
  - 修复了当所有 Kafka server 不可访问时会导致 TiCDC server panic 的问题 #8523 @3AceShowHand
  - 修复了下游为 MySQL 且执行语句和 TiDB 行为不兼容时可能导致数据丢失的问题 #8453 @asddongmen
  - 修复了 rolling upgrade 可能导致 TiCDC OOM 或者 checkpoint 卡住的问题 #8329 @overvenus
  - 修复了 Kubernetes 上不能平滑升级 (graceful upgrade) TiCDC 集群的问题 #8484 @overvenus
- TiDB Data Migration (DM)
  - 修复了 DM worker 节点使用 GCP Cloud Storage 时，由于断点续传信息记录过于频繁，达到了 GCP Cloud Storage 的请求频次上限，导致 DM worker 无法把数据写入 GCP Cloud Storage 中，从而导致全量数据加载失败的问题 #8482 @maxshuang
  - 修复了在多个导入任务同时同步同一个下游的数据，并且都使用了下游元数据表来记录断点续传信息时，所有任务的断点续传信息被写入了同一张元数据表，并且使用了相同的任务 ID 的问题 #8500 @maxshuang
- TiDB Lightning
  - 修复了当使用 Physical Import Mode 导入数据时，如果目标表的复合主键中存在 `auto_random` 列，但源数据中没有指定该列的值，TiDB Lightning 不能为 `auto_random` 列自动生成数据的问题 #41454 @D3Hunter
  - 修复了当使用 TiDB Lightning 的 Logical Import Mode 导入数据时，由于目标集群用户没有 `CONFIG` 权限导致导入失败的问题 #41915 @lichunzhu

16.5.1.5 贡献者

感谢来自 TiDB 社区的贡献者们：

- AntiTopQuark
- blacktear23
- BornChanger
- Dousir9
- erwadba
- HappyUncle
- jiyfhust
- L-maple
- liumengya94
- woofyzhao
- xiaguan

16.6 v6.6

16.6.1 TiDB 6.6.0 Release Notes

发版日期：2023 年 2 月 20 日

TiDB 版本：6.6.0-DMR

> **注意：**
>
> TiDB 6.6.0-DMR 的用户文档已归档。如无特殊需求，建议使用 TiDB 数据库的最新 LTS 版本。

试用链接：快速体验 | 下载离线包

在 6.6.0 版本中，你可以获得以下关键特性：

```
<th>分类</th>
<th>功能</th>
<th>描述</th>
```

```
<td rowspan="3">可扩展性与性能<br /></td>
  <td>TiKV 支持 <a href="https://docs.pingcap.com/zh/tidb/v6.6/partitioned-raft-kv" target="
      ↪ _blank">Partitioned Raft KV 存储引擎</a>（实验特性）</td>
  <td>TiKV 引入下一代存储引擎 Partitioned Raft KV，通过每个数据 Region 独享 RocksDB 实例，
      ↪ 可将集群的存储能力从 TB 级扩展到 PB 级，并提供更稳定的写入延迟和更强大的扩容能力。</td>
```

```
  <td>TiKV 支持<a href="https://docs.pingcap.com/zh/tidb/v6.6/system-variables#
      ↪ tidb_store_batch_size" target="_blank">批量聚合数据请求</a></td>
  <td>TiDB 支持将发送到相同 TiKV 实例的数据请求部分合并，减少子任务的数量和 RPC 请求的开销。
      ↪ 在数据离散分布且 gRPC 线程池资源紧张的情况下，批量化请求能够提升性能超 50%。</td>
```

```
<td>TiFlash 支持 <a href="https://docs.pingcap.com/zh/tidb/v6.6/stale-read" target="_blank">Stale
    ↪  Read</a> 和<a href="https://docs.pingcap.com/zh/tidb/v6.6/explain-mpp#mpp-version-和-
    ↪ exchange-数据压缩" target="_blank">压缩数据交换</a></td>
<td>TiFlash 支持 Stale Read 功能，在非实时性要求的场景可提升查询性能。TiFlash
    ↪ 默认支持带压缩的数据交换，可提升并行处理的数据交换效率，TPC-H 总体性能提升约 10%，
    ↪ 流量节省超 50%。</td>
```

```
<td rowspan="2">稳定性与高可用<br /></td>
<td><a href="https://docs.pingcap.com/zh/tidb/v6.6/tidb-resource-control" target="_blank">
    ↪ 资源管控</a>（实验特性）</td>
<td>引入资源管控框架，支持将资源配额映射到用户定义的资源组，在集群资源发生争用时，
    ↪ 对资源组内用户的资源使用进行限制。</td>
```

```
<td><a href="https://docs.pingcap.com/zh/tidb/v6.6/sql-plan-management#根据历史执行计划创建绑定"
    ↪ target="_blank">绑定历史执行计划</a></td>
<td>支持绑定历史执行计划，支持通过 TiDB Dashboard 快速绑定执行计划。</td>
```

```
<td rowspan="2">功能特性与兼容性<br /></td>
<td><a href="https://docs.pingcap.com/zh/tidb/v6.6/foreign-key" target="_blank">外键约束</a></td>
<td>支持 MySQL 兼容的外键约束，帮助保持数据一致性和提升数据质量。</td>
```

```
<td><a href="https://docs.pingcap.com/zh/tidb/v6.6/sql-statement-create-index/#多值索引" target="
    ↪ _blank">多值索引</a>（实验特性）</td>
<td>引入 MySQL 兼容的多值索引，增强 JSON 类型，提升 TiDB 对 MySQL 8.0 的兼容性。</td>
```

```
<td>数据库管理与可观测性<br /></td>
<td><a href="https://docs.pingcap.com/zh/tidb/v6.6/dm-precheck/#physical-import-检查项" target="
    ↪ _blank">DM 集成物理导入模式</a>（实验特性）</td>
<td>TiDB Data Migration (DM) 集成 TiDB Lightning 的 Physical Import 模式，提升 DM
    ↪ 全量数据迁移时的性能，大数据量场景下的迁移时间最多可提升 10 倍。</td>
```

16.6.1.1 功能详情

16.6.1.1.1 可扩展性

- 支持下一代 Partitioned Raft KV 存储引擎（实验特性）#11515 #12842 @busyjay @tonyxuqqi @tabokie @bufferflies @5kbpers @SpadeA-Tang @nolouch

  TiDB v6.6.0 之前，TiKV 基于 Raft 的存储引擎使用一个单一的 RocksDB 实例存储该 TiKV 实例所有 Region 的数据。为了更平稳地支持更大的集群，从 TiDB v6.6.0 开始，引入了一个全新的 TiKV 存储引擎，该引擎使用多个 RocksDB 实例来存储 TiKV 的 Region 数据，每个 Region 的数据都独立存储在单独的 RocksDB 实例中。新引擎能够更好地控制 RocksDB 实例的文件数和层级，并实现 Region 间数据操作的物理隔离，避免互相影响，还支持平稳管理更多的数据。可以理解为 TiKV 通过分区管理多个 RocksDB 实例，这也是该特性 Partitioned Raft KV 名字的由来。该功能的主要优势在于更好的写入性能，更快的扩缩容，相同硬件下可以支持更大的数据，也能支持更大的集群规模。

  该功能目前是实验特性，不推荐在生产环境中使用。

  更多信息，请参考用户文档。

- 支持 DDL 分布式并行执行框架（实验特性）#37125 @zimulala

  在过去的版本中，整个 TiDB 集群中仅允许一个 TiDB 实例作为 DDL Owner 处理 Schema 变更任务。为了进一步提升 DDL 的并发性，TiDB v6.6.0 版本引入了 DDL 分布式并行执行框架，支持集群中所有的 TiDB 实例并发执行同一个任务的 `StateWriteReorganization` 阶段，加速 DDL 的执行。该功能由系统变量`tidb_ddl_distribute_reorg` 控制是否开启，目前只支持 `Add Index` 操作。

16.6.1.1.2 性能

- 支持悲观锁队列的稳定唤醒模型 #13298 @MyonKeminta

  如果业务场景存在单点悲观锁冲突频繁的情况，原有的唤醒机制无法保证事务获取锁的时间，造成长尾延迟高，甚至获取锁超时。自 v6.6.0 起，你可以通过设置系统变量 `tidb_pessimistic_txn_aggressive_locking` ↪ 为 `ON` 开启悲观锁的稳定唤醒模型。在该唤醒模型下，队列的唤醒顺序可被严格控制，避免无效唤醒造成的资源浪费。在锁冲突严重的场景中，能够减少长尾延时，降低 P99 响应时间。

  更多信息，请参考用户文档。

- 批量聚合数据请求 #39361 @cfzjywxk @you06

  TiDB 向 TiKV 发送数据请求时，会根据数据所在的 Region 将请求编入不同的子任务，每个子任务只处理单个 Region 的请求。当访问的数据离散度很高时，即使数据量不大，也会生成众多的子任务，进而产生大量 RPC 请求，消耗额外的时间。自 v6.6.0 起，TiDB 支持将发送到相同 TiKV 实例的数据请求部分合并，减少子任务的数量和 RPC 请求的开销。在数据离散度高且 gRPC 线程池资源紧张的情况下，批量化请求能够将性能提升 50% 以上。

  此特性默认打开，可通过系统变量`tidb_store_batch_size`设置批量请求的大小。

- 解除执行计划缓存对 `LIMIT` 子句的限制 #40219 @fzzf678

  TiDB v6.6.0 移除了执行计划缓存的限制，带有变量的 `LIMIT` 子句可以进入执行计划缓存，如 `LIMIT ?` 或者 `LIMIT 10, ?`。这使得更多的 SQL 能够从计划缓存中获益，提升执行效率。目前出于安全考虑，仅支持缓存 `?` 参数值不大于 10000 的执行计划。

  更多信息，请参考用户文档。

- TiFlash 引擎支持带压缩的数据交换 #6620 @solotzg

  为了协同多节点进行计算，TiFlash 引擎需要在不同节点中进行数据交换。当需要交换的数据量非常大时，数据交换的性能可能影响整体计算效率。在 v6.6.0 版本中，TiFlash 引擎引入压缩机制，在必要时对需要交换的数据进行压缩，然后进行交换，从而提升数据交换效率。

  更多信息，请参考用户文档。

- TiFlash 支持 Stale Read 功能 #4483 @hehechen

  Stale Read 功能是从 TiDB v5.1.1 开始正式引入的，支持读取指定时间点或时间范围内的历史数据。Stale Read 允许直接读取 TiKV 本地副本数据，可以降低读取延迟，提升查询性能。在 v6.6.0 之前的版本中，TiFlash 并不支持 Stale Read 功能，即使 Stale Read 查询的表包含 TiFlash 副本，TiDB 也只能使用 TiKV 副本进行查询。

  在 v6.6.0 中，TiFlash 开始支持 Stale Read 功能。当使用`AS OF TIMESTAMP` 语法或`tidb_read_staleness` 系统变量等方式查询历史数据时，如果查询的表包含 TiFlash 副本，优化器可以选择 TiFlash 副本读取对应的数据，从而进一步提高查询性能。

  更多信息，请参考用户文档。

- 支持下推字符串函数 `regexp_replace` 至 TiFlash #6115 @xzhangxian1008

16.6.1.1.3 稳定性

- 支持基于资源组的资源管控（实验特性）#38825 @nolouch @BornChanger @glorv @tiancaiamao @Connor1996 @JmPotato @hnes @CabinfeverB @HuSharp

  你可以为 TiDB 集群创建资源组，将不同的数据库用户映射到对应的资源组中，根据需要设置每个资源组的配额。当集群资源紧张时，来自同一个资源组的会话所使用的全部资源将被限制在配额内，避免其中一个资源组过度消耗，从而影响其他资源组中的会话正常运行。TiDB 在 Grafana 上提供了内置视图展示资源的实际使用情况，协助你更合理地配置资源。

  资源管控特性的引入对 TiDB 具有里程碑的意义。它能够将一个分布式数据库集群划分成多个逻辑单元，即使个别单元对资源过度使用，也不会挤占其他单元所需的资源。利用该特性：

  - 你可以将多个来自不同系统的中小型应用合入一个 TiDB 集群中，个别应用的负载提升，不会影响其他应用的正常运行。而在系统负载较低的时候，繁忙的应用即使超过设定的读写配额，也仍然可以被分配到所需的系统资源，达到资源的最大化利用。

- 你可以选择将所有测试环境合入一个集群，或者将消耗较大的批量任务编入一个单独的资源组，在保证重要应用获得必要资源的同时，提升硬件利用率，降低运行成本。

此外，合理利用资源管控特性可以减少集群数量，降低运维难度及管理成本。

在 v6.6.0 中，启用资源管控特性需要同时打开 TiDB 的全局变量`tidb_enable_resource_control` 及 TiKV 的配置项`resource-control.enabled`。当前支持的限额方式基于“用量”（Request Unit，即 RU），RU 是 TiDB 对 CPU、I/O 等系统资源的统一抽象单位。

更多信息，请参考用户文档。

- 绑定历史执行计划 GA #39199 @fzzf678

  在 v6.5.0 中，TiDB 扩展了`CREATE [GLOBAL | SESSION] BINDING` 语句中的绑定对象，支持根据历史执行计划创建绑定。在 v6.6.0 中该功能 GA，执行计划的选择不仅限于当前 TiDB 节点，任意 TiDB 节点产生的历史执行计划都可以被选为SQL Binding 的目标，进一步提升了功能的易用性。

  更多信息，请参考用户文档。

- 新增若干优化器 Hint #39964 @Reminiscent

  TiDB 在 v6.6.0 中增加了若干优化器 Hint，用来控制 `LIMIT` 操作的执行计划选择：

  - `ORDER_INDEX()`：提示优化器使用指定的索引，读取数据时保持索引的顺序，生成类似 `Limit + IndexScan(keep order: true)` 的计划。
  - `NO_ORDER_INDEX()`：提示优化器使用指定的索引，读取数据时不保持顺序，生成类似 `TopN + IndexScan(keep order: false)` 的计划。

持续引入优化器 Hint 为用户提供了更多的干预手段，有助于解决 SQL 性能问题，并提升了整体性能的稳定性。

- 支持 DDL 动态资源管控（实验特性）#38025 @hawkingrei

  TiDB v6.6.0 版本引入了 DDL 动态资源管控，通过自动控制 DDL 的 CPU 使用量，尽量降低 DDL 变更任务对线上业务的影响。该功能仅在开启DDL 分布式并行执行框架后生效。

16.6.1.1.4 高可用

- Placement Rules in SQL 支持指定 SURVIVAL_PREFERENCE #38605 @nolouch

  SURVIVAL_PREFERENCES 为数据提供了生存偏好设置，从而提高数据的容灾生存能力。通过指定 SURVIVAL_PREFERENCE，你可以控制：

  - 对于跨云区域部署的 TiDB 集群，当某个云区域产生故障时，指定数据库或表能在另一个云区域继续提供服务。
  - 对于单个云区域内部署的 TiDB 集群，当某个可用区产生故障时，指定数据库或表能在另一个可用区继续提供服务。

更多信息，请参考用户文档。

- 支持通过 FLASHBACK CLUSTER TO TIMESTAMP 命令闪回 DDL 操作 #14088 @Defined2014 @JmPotato

  FLASHBACK CLUSTER TO TIMESTAMP 语句支持在 Garbage Collection (GC) life time 内快速回退整个集群到指定的时间点。在 TiDB v6.6.0 版本中，该功能新支持撤销 DDL 操作，适用于快速撤消集群的 DML 或 DDL 误操作、支持分钟级别的快速回退集群、支持在时间线上多次回退以确定特定数据更改发生的时间。

  更多信息，请参考用户文档。

16.6.1.1.5 SQL 功能

- 支持兼容 MySQL 语法的外键约束 #18209 @crazycs520

  TiDB v6.6.0 引入了兼容 MySQL 语法的外键约束功能，支持在表内、表间关联数据并进行约束校验，并且支持级联操作。该特性有助于将 MySQL 上的应用迁移到 TiDB、保持数据一致性、提升数据质量并且方便数据建模。

  更多信息，请参考用户文档。

- 支持 MySQL 兼容的多值索引 (Multi-Valued Indexes)（实验特性）#39592 @xiongjiwei @qw4990

  TiDB 在 v6.6.0 引入了 MySQL 兼容的多值索引 (Multi-Valued Indexes)。过滤 JSON 列中某个数组的值是常见的操作，但普通索引对这类操作起不到加速作用。在数组上创建多值索引能够大幅提升过滤的性能。如果 JSON 列中的某个数组上存在多值索引，那么可以利用多值索引过滤带有 MEMBER OF()、JSON_CONTAINS()、JSON_OVERLAPS() 函数的检索条件，从而减少大量的 I/O 消耗，提升运行速度。

  多值索引的引入，进一步增强了 TiDB 对 JSON 类型的支持，同时也提升了 TiDB 对 MySQL 8.0 的兼容性。

  更多信息，请参考用户文档。

16.6.1.1.6 数据库管理

- 支持配置只读存储节点来执行资源消耗型任务 @v01dstar

  在生产环境中，可能有部分只读操作定期消耗大量资源，对整个集群的性能产生影响，比如备份和大规模数据读取分析等。TiDB v6.6.0 支持配置只读存储节点，用来执行重度资源消耗的只读任务，避免对线上业务的影响。目前支持 TiDB、TiSpark 和 BR 读取只读节点上的数据。你可以按照操作步骤配置只读存储节点，并通过 TiDB 系统变量 tidb_replica_read、TiSpark 配置项 spark.tispark.replica_read 或 br 命令行参数 --replica-read-label 指定数据读取位置，以保证集群性能稳定。

  更多信息，请参考用户文档。

- 支持动态修改参数 store-io-pool-size #13964 @LykxSassinator

  TiKV 中的raftstore.store-io-pool-size 用于设定处理 Raft I/O 任务的线程池中线程的数量，需要在 TiKV 性能调优时进行调整。在 v6.6.0 之前，这个参数无法动态修改。v6.6.0 支持动态修改该参数，提高了 TiKV 性能调优的灵活性。

  更多信息，请参考用户文档。

- 支持指定集群初次启动时的初始化 SQL 脚本 #35624 @morgo

  TiDB 集群初次启动时，可通过命令行参数 --initialize-sql-file 指定执行的 SQL 脚本。该功能可用于修改系统变量的值、创建用户或分配权限等。

  更多信息，请参考用户文档。

- TiDB Data Migration (DM) 集成了 TiDB Lightning 的 Physical Import Mode（实验特性）@lance6716

  在 v6.6.0 版本中，DM 的全量迁移能力集成了 TiDB Lightning 的 Physical Import Mode，使得 DM 全量数据迁移的性能最高可提升 10 倍，大大缩短了大数据量场景下的迁移时间。在 v6.6.0 以前，数据量较多的场景下，需要单独配置 TiDB Lightning 的 Physical Import Mode 任务来进行快速的全量数据迁移，再用 DM 来进行增量数据迁移，配置较为复杂。从 v6.6.0 起，在迁移大数据量的场景，无需再配置 TiDB Lightning 的任务，使用一个 DM 任务即可完成。

  更多信息，请参考用户文档。

- TiDB Lightning 新增配置文件参数 `"header-schema-match"` 用于解决源文件里的列名和目标表的列名不匹配的问题 @dsdashun

  在 v6.6.0 版本中，TiDB Lightning 新增配置文件参数 `"header-schema-match"`，默认取值为 `true`，表示源 CSV 文件第一行有表的列名信息，且和目标表列名保持一致。如果 CSV 表头中的字段名和目标表的列名不匹配，此时可以将该配置设置为 `false`，TiDB Lightning 将忽略不匹配的问题，继续按目标表中的列顺序导入数据。

  更多信息，请参考用户文档。

- TiDB Lightning 向 TiKV 传输键值对时支持启用压缩传输 #41163 @gozssky

  自 v6.6.0 起，TiDB Lightning 支持将本地编码排序后的键值对在网络传输时进行压缩再发送到 TiKV，从而减少网络传输的数据量，降低网络带宽开销。之前版本不支持该功能，在数据量较大的情况下，TiDB Lightning 对网络带宽要求相对较高，且会产生较高的流量费。

  该功能默认关闭，你可以通过将 TiDB Lightning 配置项 `compress-kv-pairs` 设置为 `"gzip"` 或者 `"gz"` 开启此功能。

  更多信息，请参考用户文档。

- TiKV-CDC 工具 GA，支持订阅 RawKV 的数据变更 #48 @zeminzhou @haojinming @pingyu

  TiKV-CDC 是一个 TiKV 集群的 CDC (Change Data Capture) 工具。TiKV 可以独立于 TiDB，与 PD 构成 KV 数据库，此时的产品形态为 RawKV。TiKV-CDC 支持订阅 RawKV 的数据变更，并实时同步到下游 TiKV 集群，从而实现 RawKV 的跨集群复制。

  更多信息，请参考用户文档。

- TiCDC 支持在 Kafka 同步任务上开启单表的横向扩展功能，将单表的同步任务下发到多个 TiCDC 节点上执行（实验特性）#7720 @overvenus

  在 v6.6.0 之前，当上游单表写入量较大时，单表的复制能力无法横向扩展导致同步延迟增加。自 TiCDC v6.6.0 起，下游为 Kafka 的同步任务可以将上游单表的同步任务下发到多个 TiCDC 节点上执行，实现单表同步性能的横向扩展。

  更多信息，请参考用户文档。

- GORM 已添加 TiDB 集成测试。目前 TiDB 已成为 GORM 默认支持的数据库 #6014 @Icemap

  - GORM MySQL driver 在 `v1.4.6` 中，新增 TiDB `AUTO_RANDOM` 特性适配 #104
  - GORM MySQL driver 在 `v1.4.6` 中，修复了连接 TiDB 时，在 `AutoMigrate` 期间无法更改 `Unique` 字段的非 `Unique` 属性的问题 #105
  - GORM 文档提及 TiDB 作为默认数据库 #638

更多信息，请参考 GORM 用户文档。

16.6.1.1.7 可观测性

- 支持在 TiDB Dashboard 中快速绑定执行计划 #781 @YiniXu9506

  TiDB v6.6.0 中引入了执行计划快速绑定功能，你可以在 TiDB Dashboard 中快速完成 SQL 语句与特定计划的绑定。

  通过提供友好的界面，简化了在 TiDB Dashboard 上绑定 SQL 执行计划的过程，提高绑定过程的效率和用户体验。

  更多信息，请参考用户文档。

- 为执行计划缓存增加告警 @qw4990

  当执行计划无法被缓存时，TiDB 会通过告警的方式提示该计划无法被缓存的原因，降低诊断的难度。例如：

```
mysql> PREPARE st FROM 'SELECT * FROM t WHERE a<?';
Query OK, 0 rows affected (0.00 sec)

mysql> SET @a='1';
Query OK, 0 rows affected (0.00 sec)

mysql> EXECUTE st USING @a;
Empty set, 1 warning (0.01 sec)

mysql> SHOW WARNINGS;
+---------+------+-------------------------------------------------+
| Level   | Code | Message                                         |
+---------+------+-------------------------------------------------+
| Warning | 1105 | skip plan-cache: '1' may be converted to INT |
+---------+------+-------------------------------------------------+
```

  在以上例子中，优化器进行了非 INT 类型到 INT 类型的转换，产生的执行计划可能随着参数变化而存在风险，因此 TiDB 不缓存该计划。

  更多信息，请参考用户文档。

- 在慢查询中增加告警字段 #39893 @time-and-fate

  TiDB v6.6.0 慢查询日志中新增了 `Warnings` 字段，以帮助诊断查询性能问题。该字段以 JSON 格式记录慢查询语句在执行过程中产生的警告信息。你也可以在 TiDB Dashboard 的慢查询页面中查看 Warnings 信息。

  更多信息，请参考用户文档。

- 自动捕获执行计划的生成 #38779 @Yisaer

  在执行计划问题的排查过程中，PLAN REPLAYER 能够协助保存现场，提升诊断的效率。但在个别场景中，一些执行计划的生成无法任意重现，给诊断工作增加了难度。

  针对这类场景，在 TiDB v6.6.0 中，PLAN REPLAYER 扩展了自动捕获的能力。通过 PLAN REPLAYER CAPTURE 命令，你可以提前注册目标 SQL 语句，也可以同时指定目标执行计划。当 TiDB 检测到执行的 SQL 语句或执行计划与注册目标匹配时，会自动生成并打包 PLAN REPLAYER 的信息，提升执行计划不稳定时的诊断效率。

要启用该功能，需要将系统变量`tidb_enable_plan_replayer_capture`的值设为`ON`。

更多信息，请参考用户文档。

- 持久化 statements summary（实验特性）#40812 @mornyx

  在 TiDB v6.6.0 之前的版本中，statements summary 数据维护在内存中，一旦 TiDB 发生重启，数据便会全部丢失。开启 statements summary 持久化特性后，历史数据将定期被写入磁盘，相关系统表的查询数据源也将由内存变为磁盘。此时，TiDB 重启后，历史数据仍然会保留。

  更多信息，请参考用户文档。

16.6.1.1.8 安全

- TiFlash 支持 TLS 证书自动轮换 #5503 @ywqzzy

  TiDB v6.6.0 引入了 TiFlash TLS 证书自动轮换功能。在开启组件间加密传输的 TiDB 集群上，当 TiFlash 的 TLS 证书过期需要重新签发时，支持自动加载新的 TiFlash TLS 证书，无需重启 TiDB 集群。而且，TiDB 集群内部组件之间 TLS 过期轮换不影响 TiDB 集群的正常使用，保障了 TiDB 集群的高可用。

  更多信息，请参考用户文档。

- TiDB Lightning 支持通过 AWS IAM 角色的密钥以及会话令牌来访问 S3 数据 #4075 @okJiang

  在 v6.6.0 之前，TiDB Lightning 仅支持通过 AWS IAM 用户密钥访问 S3 的数据，无法使用临时会话令牌。自 v6.6.0 起，TiDB Lightning 支持通过 AWS IAM 角色密钥 + 会话令牌的方式来访问 S3 数据，以提高安全性。

  更多信息，请参考用户文档。

16.6.1.1.9 遥测

- 自 2023 年 2 月 20 日起，新发布的 TiDB 和 TiDB Dashboard 版本（含 v6.6.0），默认关闭遥测功能，即默认不再收集使用情况信息分享给 PingCAP。如果升级至这些版本前使用默认的遥测配置，则升级后遥测功能处于关闭状态。具体的版本可参考TiDB 版本发布时间线。
- 从 v1.11.3 起，新部署的 TiUP 默认关闭遥测功能，即默认不再收集使用情况信息。如果从 v1.11.3 之前的 TiUP 版本升级至 v1.11.3 或更高 TiUP 版本，遥测保持升级前的开启或关闭状态。

16.6.1.2 兼容性变更

> **注意：**
>
> 以下为从 v6.5.0 升级至当前版本 (v6.6.0) 所需兼容性变更信息。如果从 v6.4.0 或之前版本升级到当前版本，可能也需要考虑和查看中间版本 release notes 中提到的兼容性变更信息。

16.6.1.2.1 MySQL 兼容性

- 支持兼容 MySQL 语法的外键约束 #18209 @crazycs520
  更多信息，请参考本文的SQL 部分以及用户文档。
- 支持兼容 MySQL 语法的多值索引（实验特性）#39592 @xiongjiwei @qw4990
  更多信息，请参考本文的SQL 部分以及用户文档。

16.6.1.2.2 系统变量

变量名	修改类型	描述
tidb_enable_amend_pessimistic_txn ↪	删除	在 v6.5.0 中该变量被废弃。自 v6.6.0 起，该变量以及 AMEND ↪ TRANSACTION ↪ 特性被删除。TiDB 会使用元数据锁机制解决 Information ↪ ↪ schema ↪ is ↪ changed ↪ 报错的问题。

变量名	修改类型	描述
tidb_enable_concurrent_ddl	删除	这个变量用于控制是否让 TiDB 使用并发 DDL 语句。关闭该变量时 TiDB 采用旧的 DDL 执行框架，对 DDL 的并发支持有限。自 v6.6.0 起，该变量被删除，即不再支持旧的 DDL 执行框架。

变量名	修改类型	描述
tidb_ttl_job_interval ↪	删除	这个变量用于控制 TTL 后台清理任务的调度周期。自 v6.6.0 起删除该变量，因为自 v6.6.0 起 TiDB 为每张表提供了属性 TTL_JOB_INTERVAL ↪ 用于配置 TTL 运行的间隔，允许用户为每张表设置不同的运行间隔，比系统变量更加灵活。
foreign_key_checks ↪	修改	用于控制是否开启外键约束检查。默认值由 OFF 修改为 ON，表示默认开启外键检查。

变量名	修改类型	描述
tidb_enable_foreign_key ↪	修改	用于控制是否开启外键功能。默认值由 OFF 修改为 ON，表示默认开启外键功能。
tidb_enable_general_plan_cache ↪	修改	这个变量用来控制是否开启 General Plan Cache。自 v6.6.0 起，该变量更名为tidb_enable_non_prepared_plan_cache ↪ 。
tidb_enable_historical_stats ↪	修改	这个变量用来控制是否开启历史统计信息。默认值由 OFF 修改为 ON，表示默认开启历史统计信息。

变量名	修改类型	描述
tidb_enable_telemetry ↪	修改	默认值由 ON 修改为 OFF，表示默认关闭 TiDB 的遥测功能。
tidb_general_plan_cache_size ↪	修改	这个变量用来控制 General Plan Cache 最多能够缓存的计划数量。自 v6.6.0 起，该变量更名为 tidb_non_prepared_plan_cache_size ↪ 。
tidb_replica_read ↪	修改	新增选项 learner，指定 TiDB 从只读节点中读取数据的 learner 副本。

变量名	修改类型	描述
tidb_replica_read ↪	修改	新增选项 prefer-↪ leader ↪ ，以提高 TiDB 集群整体的读可用性。该选项被启用时，TiDB 会优先选择 Leader 副本进行读取操作；当 Leader 副本的处理性能显著下降时，TiDB 会自动将读操作转发给 Follower 副本。

变量名	修改类型	描述
tidb_store_batch_size	修改	该变量设置 IndexLookUp 算子回表时多个 Coprocessor Task 的 batch 大小。0 代表不使用 batch。自 v6.6.0 起，默认值由 0 调整为 4，即每批请求会有 4 个 Coprocessor Task 被 batch 到一个 task 中。

变量名	修改类型	描述
mpp_exchange_compression_mode ↪	新增	该变量用于选择 MPP Exchange 算子的数据压缩模式，当 TiDB 选择版本号为 1 的 MPP 执行计划时生效。默认值为 UNSPECIFIED ↪ ，表示 TiDB 自动选择 FAST 压缩模式。
mpp_version ↪	新增	该变量用于指定不同版本的 MPP 执行计划。指定后，TiDB 会选择指定版本的 MPP 执行计划。默认值为 UNSPECIFIED ↪ ，表示 TiDB 自动选择最新版本 1。

变量名	修改类型	描述
tidb_ddl_distribute_reorg	新增	这个变量用来控制是否开启分布式执行 DDL reorg 阶段，来提升此阶段的速度。默认值为 OFF，表示默认不开启分布式执行 DDL reorg 阶段。目前此开关只对 ADD INDEX 语句有效。

变量名	修改类型	描述
tidb_enable_historical_stats_for_capture	新增	这个变量用来控制 PLAN REPLAYER CAPTURE 抓取的内容是否默认带历史统计信息。默认值为 OFF，表示默认不带历史统计信息。

变量名	修改类型	描述
tidb_enable_pa ↪	新增	cache_for_param_limit 这个变量用来控制 Prepared Plan Cache 是否缓存 Limit 后带有 COUNT 的执行计划。默认值为 ON，表示默认缓存这样的执行计划。目前不支持缓存 Limit 后面的 COUNT 具体参数值大于 10000 的执行计划。

变量名	修改类型	描述
tidb_enable_plan_replayer_capture	新增	这个变量用来控制是否开启PLAN REPLAYER CAPTURE。默认值OFF，代表默认关闭PLAN REPLAYER CAPTURE。
tidb_enable_resource_control	新增	该变量是资源管控特性的开关。默认值为OFF。该变量设置为ON后，集群支持应用按照资源组做资源隔离。

变量名	修改类型	描述
tidb_historical_stats_duration	新增	这个变量用来控制历史统计信息在存储中的保留时间，默认值为 7 天。
tidb_index_join_double_read_penalty_cost_rate	新增	用于控制是否给 index join 增加一些惩罚性代价。默认值为 0，即不开启该功能。
tidb_pessimistic_txn_aggressive_locking	新增	是否对悲观锁启用加强的悲观锁唤醒模型。默认值为 OFF，表示默认不对悲观锁启用加强的悲观锁唤醒模型。

变量名	修改类型	描述
tidb_stmt_summary_enable_persistent ↪	新增	只读变量。表示是否开启statement summary tables 持久化。该变量的值与配置文件中tidb_stmt_summary_enable_persistent ↪ 的取值相同。
tidb_stmt_summary_filename ↪	新增	只读变量。表示当开启statement summary tables 持久化后持久化数据所写入的文件。该变量的值与配置文件中tidb_stmt_summary_filename ↪ 的取值相同。

变量名	修改类型	描述
tidb_stmt_summary_file_max_backups	新增	只读变量。表示当开启statement summary tables 持久化后持久化数据文件的最大数量限制。该变量的值与配置文件中tidb_stmt_summary_file_max_backups 的取值相同。
tidb_stmt_summary_file_max_days	新增	只读变量。表示当开启statement summary tables 持久化后持久化数据文件所保留的最大天数。该变量的值与配置文件中tidb_stmt_summary_file_max_days 的取值相同。

变量名	修改类型	描述
tidb_stmt_summary_file_max_size ↪	新增	只读变量。表示当开启statement summary tables 持久化后持久化数据单个文件的大小限制。该变量的值与配置文件中tidb_stmt_summary_file_max_size ↪ 的取值相同。

16.6.1.2.3 配置文件参数

配置文件	配置项	修改类型	描述
TiKV	rocksdb. ↪ enable ↪ - ↪ statistics ↪	删除	该配置项指定是否开启 RocksDB 的统计信息收集功能。从 v6.6.0 起，删除该配置项。所有集群默认开启统计信息收集，以便于故障排查。详情参见 #13942。

配置文件	配置项	修改类型	描述
TiKV	`raftdb.` ↪ `enable` ↪ `-` ↪ `statistics` ↪	删除	该配置项指定是否开启 Raft RocksDB 的统计信息收集功能。从 v6.6.0 起，删除该配置项。所有集群默认开启统计信息收集，以便于故障排查。详情参见 #13942。
TiKV	`storage.` ↪ `block-` ↪ `cache.` ↪ `shared`	删除	从 v6.6.0 起删除该配置项，默认开启 block cache 且无法关闭，详情参见 #12936。

配置文件	配置项	修改类型	描述
DM	on-duplicate	删除	该配置项控制全量导入阶段针对冲突数据的解决方式。自 v6.6.0 起，引入新的配置项 on-duplicate-logical 和 on-duplicate-physical，取代 on-duplicate。
TiDB	enable-telemetry	修改	自 v6.6.0 起，该配置项默认值由 true 改为 false，表示默认关闭 TiDB 的遥测功能。
TiKV	rocksdb.defaultcf.block-size 和rocksdb.writecf.block-size	修改	默认值由 64K 调整为 32K。

配置文件	配置项	修改类型	描述
TiKV	rocksdb. ↪ defaultcf ↪ .block ↪ -cache ↪ -size, rocksdb. ↪ writecf ↪ .block ↪ -cache ↪ -size, rocksdb. ↪ lockcf ↪ .block ↪ -cache ↪ -size	修改	从 v6.6.0 起，这三个配置项被废弃。详情参见 #12936。
PD	enable- ↪ telemetry ↪	修改	从 v6.6.0 起，该配置项的默认值由 `true` 改为 `false`，表示默认关闭 TiDB Dasboard 的遥测功能。
TiFlash	profile. ↪ default ↪ . ↪ max_memory_usage_for_all_queries ↪	修改	表示所有查询过程中，节点对中间数据的内存限制。自 v6.6.0 起默认值由 `0` 改为 `0.8`，表示节点占总内存的 80%。

配置文件	配置项	修改类型	描述
TiCDC	consistent ↪ . ↪ storage ↪	修改	redo log 备份文件的地址，除了 NFS，支持的 scheme 新增了 GCS 和 Azure。
DM	import- ↪ mode	修改	该配置项的可选值由 "sql" 和 "loader" 变更为 "logical" 和 "physical ↪ "。默认值为 "logical"，即使用 TiDB Lightning 的 Logical Import Mode 进行导入。
TiDB	initialize ↪ -sql- ↪ file	新增	用于指定 TiDB 集群初次启动时执行的 SQL 脚本。默认值为空。
TiDB	tidb_stmt_summary_enable_persistent ↪	新增	用于控制是否开启 statements summary 持久化。默认值为 false，即不开启该功能。

配置文件	配置项	修改类型	描述
TiDB	tidb_stmt_summary_file_max_backups	新增	当开启了 statements summary 持久化时，该配置用于限制持久化数据文件最大数量，默认值为 0，表示不限制文件数量。
TiDB	tidb_stmt_summary_file_max_days	新增	当开启了 statements summary 持久化时，该配置用于指定持久化数据文件所保留的最大天数，默认为 3 天。
TiDB	tidb_stmt_summary_file_max_size	新增	当开启了 statements summary 持久化时，该配置用于限制持久化数据单个文件的大小 (MiB)，默认值为 64。

配置文件	配置项	修改类型	描述
TiDB	tidb_stmt_summary_filename	新增	当开启了 statements summary 持久化时，该配置用于指定持久化数据所写入的文件名称，默认为 tidb-statements.log。
TiKV	resource-control.enabled	新增	控制是否支持对用户前台的读写请求按照对应的资源组配额做优先级调度。默认为 false，即关闭按照资源组配额调度。
TiKV	storage.engine	新增	用于设置存储引擎类型，可选值有 "raft-kv" 和 "partitioned-raft-kv"。该配置只能在创建新集群时指定，且后续无法更改。

配置文件	配置项	修改类型	描述
TiKV	rocksdb. ↪ write- ↪ buffer ↪ -flush ↪ - ↪ oldest ↪ -first	新增	用于设置当 RocksDB 当前 memtable 内存占用达到阈值之后的 Flush 策略。
TiKV	rocksdb. ↪ write- ↪ buffer ↪ -limit	新增	用于设置单个 TiKV 中所有 RocksDB 实例使用的 memtable 的总内存上限，默认值为本机内存的 25%。
PD	pd-server ↪ . ↪ enable ↪ -gogc- ↪ tuner	新增	控制是否开启 GOGC Tuner。默认关闭。
PD	pd-server ↪ .gc- ↪ tuner- ↪ threshold ↪	新增	GOGC Tuner 自动调节的最大内存阈值比例。默认值为 0.6。
PD	pd-server ↪ . ↪ server ↪ - ↪ memory ↪ -limit	新增	PD 实例的内存限制比例。0 表示不设内存限制。

配置文件	配置项	修改类型	描述
PD	pd-server ↪ . ↪ server ↪ - ↪ memory ↪ -limit ↪ -gc- ↪ trigger ↪	新增	PD 尝试触发 GC 的阈值比例。默认值为 0.7。
TiCDC	scheduler ↪ . ↪ region ↪ -per- ↪ span	新增	该配置项用于将表按 Region 个数划分成多个同步范围，这些范围可由多个 TiCDC 节点同步，默认值为 50000。
TiDB Lightning	compress- ↪ kv- ↪ pairs	新增	该配置项控制 Physical Import Mode 向 TiKV 发送 KV 时是否启用压缩，默认值为空，表示不启用压缩。

配置文件	配置项	修改类型	描述
DM	checksum-physical	新增	该配置项控制 Physical Import 在导入完成一张表后，对每一个表执行 ADMIN CHECKSUM TABLE <table> 进行数据校验。默认值为 "required"，表示导入完成后进行数据校验，如果校验失败会暂停任务，需要你手动处理。
DM	disk-quota-physical	新增	该配置项设置了磁盘的空间限制，对应 TiDB Lightning 的disk-quota 配置。

配置文件	配置项	修改类型	描述
DM	on-duplicate-logical	新增	该配置项控制 Logical Import 针对冲突数据的解决方式。默认值为 "replace"，表示用最新数据替代已有数据。
DM	on-duplicate-physical	新增	该配置项控制 Physical Import 针对冲突数据的解决方式。默认值为 "none"，表示遇到冲突数据时不进行处理。该模式性能最佳，但下游数据库会出现数据索引不一致的问题。
DM	sorting-dir-physical	新增	该配置项控制 Physical Import 用作本地排序的目录位置，该选项的默认值与 dir 配置项一致。

配置文件	配置项	修改类型	描述
sync-diff-inspector	skip-non-existing-table	新增	当下游数据库的表在上游不存在时，该配置项决定是否跳过对上下游数据库表数量不一致场景的校验。
TiSpark	spark.tispark.replica_read	新增	控制读取副本的类型，可选值为 leader、follower、learner。
TiSpark	spark.tispark.replica_read.label	新增	设置目标 TiKV 节点的标签。

16.6.1.2.4 其他

- 支持动态修改参数store-io-pool-size，增加了 TiKV 性能调优的灵活性。
- 解除了执行计划缓存对 LIMIT 子句的限制，提升了执行效率。
- v6.6.0 及以上的 BR 版本不支持恢复数据到 v6.1.0 及以下版本的集群。
- 自 v6.6.0 起，由于可能存在正确性问题，分区表目前不再支持修改列类型。

16.6.1.3 改进提升

- TiDB
  - 改进了 TTL 后台清理任务的调度机制，允许将单个表的清理任务拆分成若干子任务并调度到多个 TiDB 节点同时运行 #40361 @YangKeao
  - 优化了在设置了非默认的 delimiter 后多语句返回结果的列名显示 #39662 @mjonss
  - 优化了生成警告信息后的语句执行效率 #39702 @tiancaiamao
  - 为 ADD INDEX 支持分布式数据回填（实验特性）#37119 @zimulala
  - 允许使用 CURDATE() 作为列的默认值 #38356 @CbcWestwolf
  - 增加了 partial order prop push down 对 LIST 类型分区表的支持 #40273 @winoros

  - 为 Optimizer Hints 和执行计划绑定的冲突新增报错信息 #40910 @Reminiscent
  - 优化了 Plan Cache 策略，避免在一些场景使用 Plan Cache 时产生不优计划 #40312 #40218 #40280 #41136 #40686 @qw4990
  - 定期清理过期的 Region 缓存，避免内存泄漏和性能下降问题 #40461 @sticnarf
  - 不支持在分区表上执行 `MODIFY COLUMN` #39915 @wjhuang2016
  - 禁止重命名分区表所依赖的列 #40150 @mjonss
  - 优化了删除分区表所依赖的列的错误提示 #38739 @jiyfhust
  - 增加了 `FLASHBACK CLUSTER` 在检查 `min-resolved-ts` 失败后的重试机制 #39836 @Defined2014

- TiKV

  - 在 partitioned-raft-kv 模式下优化了一些参数的默认值：TiKV 配置项 `storage.block-cache.capacity` 的默认值由 45% 调整为 30%，`region-split-size` 的默认值由 96MiB 调整为 10GiB。当沿用 raft-kv 模式且 `enable-region-bucket` 为 `true` 时，`region-split-size` 默认调整为 1GiB #12842 @tonyxuqqi
  - 支持在 Raftstore 异步写入中进行优先级调度 #13730 @Connor1996
  - 支持在小于 1 core 的 CPU 下启动 TiKV #13586 #13752 #14017 @andreid-db
  - 优化 Raftstore slow score 探测的新机制，新增 `evict-slow-trend-scheduler` #14131 @innerr
  - 强制 RocksDB 的 block cache 为共享的，不再支持按照 CF 单独设置 block cache #12936 @busyjay

- PD

  - 增加全局内存阈值管理功能以缓解 OOM 问题（实验特性）#5827 @hnes
  - 增加 GC Tuner 功能以缓解 GC 压力（实验特性）#5827 @hnes
  - 新增 `evict-slow-trend-scheduler` 调度器用于检测和调度异常节点 #5808 @innerr
  - 新增 keyspace manager，支持对 keyspace 的管理 #5293 @AmoebaProtozoa

- TiFlash

  - 提供独立的 MVCC 位图过滤器，解耦 TiFlash 整体数据扫描流程中的 MVCC 过滤操作，为后续优化数据扫描流程提供基础 #6296 @JinheLin
  - 减少 TiFlash 在没有查询的情况下的内存使用，最高减少 30% #6589 @hongyunyan

- Tools

  - Backup & Restore (BR)

    * 优化 TiKV 端下载日志备份文件的并发度，提升常规场景下 PITR 恢复的性能 #14206 @YuJuncen

  - TiCDC

    * 支持 Batch UPDATE DML 语句，提升 TiCDC 的同步性能 #8084 @amyangfei
    * 采用异步模式实现 MQ sink 和 MySQL sink，提升 sink 的吞吐能力 #5928 @hicqu @hi-rustin

  - TiDB Data Migration (DM)

    * 优化了 DM 的告警规则和内容 #7376 @D3Hunter
      之前 “DM_XXX_process_exits_with_error” 类告警是遇到相关错误即报警，但有些告警是由于数据库连接长时间 idle 导致的，重连后即可恢复。为了减少此类报警，DM 将错误分为了可自动恢复错误和不可自动恢复错误：
      · 对于可自动恢复的错误，只有在 2 分钟内发生超过 3 次时才报警。
      · 对于不可自动恢复的错误，维持原有行为，立即报警。

    * 新增 async/batch relay writer 以优化 relay 性能 #4287 @GMHDBJD

  - TiDB Lightning

    * Physical Import Mode 支持 Keyspace #40531 @iosmanthus
    * 支持通过 `lightning.max-error` 设置最大冲突个数 #40743 @dsdashun
    * 支持导入带有 BOM header 的 CSV 数据文件 #40744 @dsdashun
    * 优化遇到 TiKV 限流错误时的处理逻辑，改为尝试其他空闲的 Region #40205 @lance6716
    * 导入时关闭对表外键的检查 #40027 @gozssky

  - Dumpling

    * 支持导出外键相关设置 #39913 @lichunzhu

  - sync-diff-inspector

    * 新增 `skip-non-existing-table` 参数，当下游数据库的表在上游不存在时，可配置该参数为 `true` 跳过对上下游数据库表数量不一致场景的校验 #692 @lichunzhu @liumengya94

16.6.1.4 错误修复

- TiDB

  - 修复了收集统计信息任务因为错误的 `datetime` 值而失败的问题 #39336 @xuyifangreeneyes
  - 修复了创建表之后 `stats_meta` 未创建的问题 #38189 @xuyifangreeneyes
  - 修复了 DDL 回填数据时频繁出现的事务写冲突问题 #24427 @mjonss
  - 修复了部分情况下空表无法使用 ingest 模式添加索引的问题 #39641 @tangenta
  - 修复了同一事务中不同 SQL 的慢日志 `wait_ts` 相同的问题 #39713 @TonsnakeLin
  - 修复了在添加列的过程中删除行记录时报 `Assertion Failed` 错误的问题 #39570 @wjhuang2016
  - 修复了修改列类型时报 `not a DDL owner` 错误的问题 #39643 @zimulala
  - 修复了在 AUTO_INCREMENT 列自动分配值耗尽后插入一行不报错的问题 #38950 @Dousir9
  - 修复了创建表达式索引时报 `Unknown column` 错误的问题 #39784 @Defined2014
  - 修复了当生成的列表达式包含表名时，重命名表后无法插入数据的问题 #39826 @Defined2014
  - 修复了当列处于 write-only 状态时 `INSERT ignore` 语句无法正确填充默认值的问题 #40192 @YangKeao
  - 修复了关闭资源管控模块时未能释放资源的问题 #40546 @zimulala
  - 修复了 TTL 任务不能及时触发统计信息更新的问题 #40109 @YangKeao
  - 修复了 TiDB 构造 key 范围时对 NULL 值处理不当，导致读取非预期数据的问题 #40158 @tiancaiamao
  - 修复了 `MODIFY COLUMN` 同时修改列默认值导致写入非法值的问题 #40164 @wjhuang2016
  - 修复了表 Region 比较多时因 Region 缓存失效导致加索引效率低下的问题 #38436 @tangenta
  - 修复了分配自增 ID 时的数据竞争问题 #40584 @Dousir9
  - 修复了 JSON 的 not 表达式实现与 MySQL 实现不兼容的问题 #40683 @YangKeao
  - 修复了并发视图模式可能会造成 DDL 操作卡住的问题 #40352 @zeminzhou
  - 修复了并发对分区表执行 `MODIFY COLUMN` 的 DDL 操作时有可能会造成数据不一致的问题 #40620 @mjonss @mjonss
  - 修复了使用 `caching_sha2_password` 方式进行认证但不指定密码导致的 “Malformed packet” 问题 #40831 @dveeden
  - 修复了在执行 TTL 任务时，如果表的主键包含 ENUM 类型的列任务会失败的问题 #40456 @lcwangchao
  - 修复了某些被 MDL 阻塞的 DDL 操作无法在 `mysql.tidb_mdl_view` 中查询到的问题 #40838 @YangKeao
  - 修复了 DDL 在 ingest 过程中可能会发生数据竞争的问题 #40970 @tangenta

- 修复了在改变时区后 TTL 任务可能会错误删除某些数据的问题 #41043 @lcwangchao
- 修复了 JSON_OBJECT 在某些情况下会报错的问题 #39806 @YangKeao
- 修复了 TiDB 在初始化时有可能死锁的问题 #40408 @Defined2014
- 修复了内存重用导致的在某些情况下系统变量的值会被错误修改的问题 #40979 @lcwangchao
- 修复了 ingest 模式下创建唯一索引可能会导致数据和索引不一致的问题 #40464 @tangenta
- 修复了并发 truncate 同一张表时，部分 truncate 操作无法被 MDL 阻塞的问题 #40484 @wjhuang2016
- 修复了 SHOW PRIVILEGES 命令显示的权限列表不完整的问题 #40591 @CbcWestwolf
- 修复了在添加唯一索引时有可能会 panic 的问题 #40592 @tangenta
- 修复了执行 ADMIN RECOVER 语句可能会造成索引数据损坏的问题 #40430 @xiongjiwei
- 修复了表达式索引中含有 CAST 时对表进行查询可能出错的问题 #40130 @xiongjiwei
- 修复了某些情况下唯一索引仍然可能产生重复数据的问题 #40217 @tangenta
- 修复了使用 Prepare 或 Execute 查询某些虚拟表时无法将表 ID 下推，导致在大量 Region 的情况下 PD OOM 的问题 #39605 @djshow832
- 修复了添加索引时可能导致数据竞争的问题 #40879 @tangenta
- 修复了由虚拟列引发的 can't find proper physical plan 问题 #41014 @AilinKid
- 修复了当动态裁剪模式下的分区表有 global binding 时，TiDB 重启失败的问题 #40368 @Yisaer
- 修复了 auto analyze 导致 graceful shutdown 耗时长的问题 #40038 @xuyifangreeneyes
- 修复了 IndexMerge 算子在触发内存限制行为时可能导致 TiDB server 崩溃的问题 #41036 @guo-shaoge
- 修复了在分区表上执行 SELECT * FROM table_name LIMIT 1 查询时，执行速度慢的问题 #40741 @solotzg

- TiKV

  - 修复转换 const Enum 类型到其他类型时报错的问题 #14156 @wshwsh12
  - 修复 Resolved TS 导致网络流量升高的问题 #14092 @overvenus
  - 修复 TiDB 中事务在执行悲观 DML 失败后，再执行其他 DML 时，如果 TiDB 和 TiKV 之间存在网络故障，可能会造成数据不一致的问题 #14038 @MyonKeminta

- PD

  - 修复 Region Scatter 任务会生成非预期的多余副本的问题 #5909 @HundunDM
  - 修复 Online Unsafe Recovery 功能在 auto-detect 模式下卡住并超时的问题 #5753 @Connor1996
  - 修复 replace-down-peer 在特定条件下执行变慢的问题 #5788 @HundunDM
  - 修复调用 ReportMinResolvedTS 过于频繁导致 PD OOM 的问题 #5965 @HundunDM

- TiFlash

  - 修复查询 TiFlash 相关的系统表可能会卡住的问题 #6745 @lidezhu
  - 修复半连接在计算笛卡尔积时，使用内存过量的问题 #6730 @gengliqi
  - 修复对 DECIMAL 数据类型进行除法运算时结果不舍入的问题 #6393 @LittleFall
  - 修复了 TiFlash 查询中由于 start_ts 无法唯一标识一个 MPP query 导致 MPP query 可能会被误取消的问题 #43426 @hehechen

- Tools

  - Backup & Restore (BR)

    * 修复恢复日志备份时热点 Region 导致恢复失败的问题 #37207 @Leavrth
    * 修复恢复数据到正在运行日志备份的集群，导致日志备份文件无法恢复的问题 #40797 @Leavrth

    * 修复 PITR 功能不支持 CA-bundle 认证的问题 #38775 @YuJuncen
    * 修复恢复时重复的临时表导致的 panic 问题 #40797 @joccau
    * 修复 PITR 不支持 PD 集群配置变更的问题 #14165 @YuJuncen
    * 修复 PD 与 TiDB server 的连接故障导致 PITR 备份进度不推进的问题 #41082 @YuJuncen
    * 修复 PD 与 TiKV 的连接故障导致 TiKV 不能监听 PITR 任务的问题 #14159 @YuJuncen
    * 修复当 TiDB 集群不存在 PITR 备份任务时，`resolve lock` 频率过高的问题 #40759 @joccau
    * 修复 PITR 备份任务被删除时，存在备份信息残留导致新任务出现数据不一致的问题 #40403 @joccau

  - TiCDC

    * 修复不能通过配置文件修改 `transaction_atomicity` 和 `protocol` 参数的问题 #7935 @CharlesCheung96
    * 修复 redo log 存储路径没做权限预检查的问题 #6335 @CharlesCheung96
    * 修复 redo log 容忍 S3 存储故障的时间过短的问题 #8089 @CharlesCheung96
    * 修复 changefeed 在 TiKV、TiCDC 节点扩缩容等特殊场景下卡住的问题 #8174 @hicqu
    * 修复 TiKV 节点之间流量过大的问题 #14092 @overvenus
    * 优化 pull-based sink 打开时 TiCDC 在 CPU 利用率、内存控制、吞吐等方面若干性能问题 #8142 #8157 #8001 #5928 @hicqu @hi-rustin

  - TiDB Data Migration (DM)

    * 修复 `binlog-schema delete` 命令执行失败的问题 #7373 @liumengya94
    * 修复当最后一个 binlog 是被 skip 的 DDL 时，checkpoint 不推进的问题 #8175 @D3Hunter
    * 修复当在某个表上同时指定 UPDATE 和非 UPDATE 类型的表达式过滤规则 `expression-filter` 时，所有 UPDATE 操作被跳过的问题 #7831 @lance6716
    * 修复当某个表上仅指定 `update-old-value-expr` 或 `update-new-value-expr` 时，过滤规则不生效或 DM 发生 panic 的问题 #7774 @lance6716

  - TiDB Lightning

    * 修复了部分场景下 TiDB 重启导致 TiDB Lightning 卡住的问题 #33714 @lichunzhu
    * 修复在并行导入时，当除最后一个 TiDB Lightning 实例外的其他实例都遇到本地重复记录时，TiDB Lightning 可能会错误地跳过冲突处理的问题 #40923 @lichunzhu
    * 修复 precheck 无法准确检测目标集群是否存在运行中的 TiCDC 的问题 #41040 @lance6716
    * 修复 TiDB Lightning 在 split-region 阶段发生 panic 的问题 #40934 @lance6716
    * 修复冲突处理逻辑 (`duplicate-resolution`) 可能导致 checksum 不一致的问题 #40657 @gozssky
    * 修复当数据文件中存在未闭合的 delimiter 时可能 OOM 的问题 #40400 @buchuitoudegou
    * 修复报错中的文件 offset 超过文件大小的问题 #40034 @buchuitoudegou
    * 修复新版 PD client 可能导致并行导入失败的问题 #40493 @AmoebaProtozoa
    * 修复 precheck 检查项有时无法监测到之前的导入失败遗留的脏数据的问题 #39477 @dsdashun

16.6.1.5 贡献者

感谢来自 TiDB 社区的贡献者们：

- morgo
- jiyfhust
- b41sh
- sourcelliu

- songzhibin97
- mamil
- Dousir9
- hihihuhu
- mychoxin
- xuning97
- andreid-db

16.7 v6.5

16.7.1 TiDB 6.5.3 Release Notes

发版日期：2023 年 6 月 14 日

TiDB 版本：6.5.3

试用链接：快速体验 | 生产部署 | 下载离线包

16.7.1.1 改进提升

- TiDB
  - 提升了对带有 Placement Rules 的分区表的 TRUNCATE 操作速度 #43070 @Lloyd-Pottiger
  - 在 resolve lock 之后避免无效的 Stale Read 重试 #43659 @you06
  - 在 Stale Read 遇到 DataIsNotReady 报错时使用 leader read 来降低延迟 #765 @Tema
  - 为 Stale Read 增加 `Stale Read OPS` 和 `Stale Read MBps` 指标，用于监控命中率和流量 #43325 @you06
- TiKV
  - 使用 gzip 压缩 `check_leader` 请求以减少流量 #14839 @cfzjywxk
- PD
  - PD Leader 选举使用单独的 gRPC 链接，防止受到其他请求的影响 #6403 @rleungx
- Tools
  - TiCDC
    * 优化 TiCDC 对 DDL 的处理方式，使 DDL 不阻塞其他无关的 DML Event 的使用，同时减少内存使用 #8106 @asddongmen
    * 调整 Decoder 接口，增加了新方法 AddKeyValue #8861 @3AceShowHand
    * 优化同步数据到对象存储的场景下发生 DDL 事件时的目录结构 #8890 @CharlesCheung96
    * 支持同步到 Kafka-on-Pulsar 下游 #8892 @hi-rustin
    * 当同步数据到 Kafka 时，支持 OAuth 协议验证方式 #8865 @hi-rustin
    * 优化采用 Avro 或 CSV 协议同步数据时 TiCDC 对 UPDATE 语句的处理方式，即将其拆分为 DELETE 和 INSERT 语句，这样用户从 DELETE 语句中即可获取修改前的 old value #9086 @3AceShowHand
    * 增加一个配置项 `insecure-skip-verify`，控制在同步数据到 Kafka 的场景下启用 TLS 时是否设置认证算法 #8867 @hi-rustin

    * TiCDC 优化 DDL 同步操作，减轻 DDL 操作对下游延迟的影响 #8686 @hi-rustin
    * 优化 TiCDC 在同步任务失败时对上游 GC TLS 的设置方法 #8403 @charleszheng44

  - TiDB Lightning

    * 在导入数据期间遇到 unknown RPC 错误时，增加了重试机制 #43291 @D3Hunter

  - TiDB Binlog

    * 优化表信息的获取方式，降低 Drainer 的初始化时间和内存占用 #1137 @lichunzhu

16.7.1.2 错误修复

- TiDB

  - 修复 `min, max` 查询结果出错的问题 #43805 @wshwsh12
  - 修复窗口函数计算下推到 TiFlash 时执行计划构造错误的问题 #43922 @gengliqi
  - 修复使用 CTE 的查询导致 TiDB 卡住的问题 #43749 #36896 @guo-shaoge
  - 修复在使用 AES_DECRYPT 表达式时，SQL 报错 `runtime error: index out of range` 的问题 #43063 @lcwangchao
  - 修复 `SHOW PROCESSLIST` 语句无法显示子查询时间较长语句的事务的 TxnStart 的问题 #40851 @crazycs520
  - 修复 PD 隔离可能会导致运行的 DDL 阻塞的问题 #44014 #43755 #44267 @wjhuang2016
  - 修复使用 UNION 查询联合视图和临时表时 TiDB panic 的问题 #42563 @lcwangchao
  - 修复 Placement Rule 在分区表下的行为问题，使得删除的分区 Placement Rule 可以被正确设置并回收 #44116 @lcwangchao
  - 修复在 TRUNCATE 分区表的某个分区时可能造成分区的 Placement Rule 失效的问题 #44031 @lcwangchao
  - 修复在重命名表期间 TiCDC 可能丢失部分行变更的问题 #43338 @tangenta
  - 修复使用 BR 导入表后 DDL 作业历史记录丢失的问题 #43725 @tangenta
  - 修复了 JSON_OBJECT 在某些情况下会报错的问题 #39806 @YangKeao
  - 修复 IPv6 环境下的集群无法查询部分系统视图的问题 #43286 @Defined2014 @nexustar
  - 修复当 PD 成员地址发生变化时，为 AUTO_INCREMENT 列分配 ID 会被长时间阻塞的问题 #42643 @tiancaiamao
  - 修复回收放置规则时，TiDB 向 PD 发送重复请求造成 PD 日志中出现大量 `full config reset` 的问题 #33069 @tiancaiamao
  - 修复了 `SHOW PRIVILEGES` 命令显示的权限列表不完整的问题 #40591 @CbcWestwolf
  - 修复 `ADMIN SHOW DDL JOBS LIMIT` 返回错误结果的问题 #42298 @CbcWestwolf
  - 修复在开启密码强度校验时对 `tidb_auth_token` 用户进行校验导致用户创建失败的问题 #44098 @CbcWestwolf
  - 修复动态裁剪模式下内连接表时找不到分区的问题 #43686 @mjonss
  - 修复在分区表上执行 `MODIFY COLUMN` 时输出 `Data Truncated` 相关报错的问题 #41118 @mjonss
  - 修复 IPv6 环境下显示错误的 TiDB 地址的问题 #43260 @nexustar
  - 修复在谓词下推的情况下 CTE 结果错误的问题 #43645 @winoros
  - 修复在带有非关联子查询的语句中使用公共表表达式 (CTE) 可能导致结果错误的问题 #44051 @winoros
  - 修复 Join Reorder 可能会造成 Outer Join 结果错误的问题 #44314 @AilinKid
  - 修复在一些极端情况下，悲观事务的第一条语句发生重试时，对该事务进行 resolve lock 可能影响事务正确性的问题 #42937 @MyonKeminta

  - 修复在一些罕见的情况下，悲观事务的残留悲观锁在 GC resolve lock 时可能影响数据正确性的问题 #43243 @MyonKeminta
  - 修复了 `batch cop` 在执行过程中的 scan detail 信息不准确的问题 #41582 @you06
  - 修复在同时使用 Stale Read 和 PREPARE 语句时 TiDB 无法读取到数据更新的问题 #43044 @you06
  - 修复执行 `LOAD DATA` 语句可能误报 `assertion failed` 的问题 #43849 @you06
  - 修复使用 Stale Read 过程中，当 coprocessor 遇到 `region data not ready` 情况时无法 fallback 到 leader 的问题 #43365 @you06

- TiKV

  - 修复 Continuous Profiling 中的文件句柄泄露的问题 #14224 @tabokie
  - 修复 PD 宕机可能造成 PITR 无法推进的问题 #14184 @YuJuncen
  - 修复加密 Key ID 冲突会导致旧 Key 被删除的问题 #14585 @tabokie
  - 修复 autocommit 和 point get replica read 可能破坏线性一致性的问题 #14715 @cfzjywxk
  - 修复集群从较低版本升级到 v6.5 或更高版本时，由于累计的 Lock 记录可能导致性能下降到问题 #14780 @MyonKeminta
  - 修复 TiDB Lightning 可能导致 SST 文件泄露的问题 #14745 @YuJuncen
  - 修复加密密钥和 raft log 文件删除之间的潜在冲突导致 TiKV 无法启动的问题 #14761 @Connor1996

- TiFlash

  - 修复分区表 TableScan 算子在 Region 迁移时性能劣化的问题 #7519 @Lloyd-Pottiger
  - 修复在 GENERATED 类型表字段与 TIMESTAMP 或 TIME 类型同时存在的情况下，查询 TiFlash 可能会报错的问题 #7468 @Lloyd-Pottiger
  - 修复大的更新事务可能会导致 TiFlash 反复报错重启的问题 #7316 @JaySon-Huang
  - 修复 `INSERT SELECT` 语句从 TiFlash 读取数据时报错 “Truncate error cast decimal as decimal” 的问题 #7348 @windtalker
  - 修复查询在 Join build 侧数据非常大，且包含许多小型字符串类型列时，消耗的内存可能会超过实际需要的问题 #7416 @yibin87

- Tools

  - Backup & Restore (BR)

    * 修复备份失败时 BR 的报错信息 “resolve lock timeout” 具有误导性，掩盖了实际错误的问题 #43236 @YuJuncen

  - TiCDC

    * 修复在表数量多达 50000 个时可能出现 OOM 的问题 #7872 @sdojjy
    * 修复 TiCDC 在上游 TiDB 发生 OOM 时卡住的问题 #8561 @overvenus
    * 修复 PD 出现网络隔离或 PD Owner 节点重启等故障时 TiCDC 卡住问题 #8808 #8812 #8877 @asddongmen
    * 修复 TiCDC 的时区设置问题 #8798 @hi-rustin
    * 修复上游 TiKV 节点 crash 时 checkpoint lag 上升的问题 #8858 @hicqu
    * 修复在同步数据到下游 MySQL 的场景中，当上游 TiDB 执行 `FLASHBACK CLUSTER TO TIMESTAMP` 语句后同步出错的问题 #8040 @asddongmen
    * 修复当同步数据到对象存储时上游的 `EXCHANGE PARTITION` 操作没有正常同步到下游的问题 #8914 @CharlesCheung96

    * 修复在某些特殊场景下 sorter 组件内存使用过多导致 OOM 的问题 #8974 @hicqu
    * 修复当下游为 Kafka 时，TiCDC 查询下游的元信息频率过高导致下游负载过大的问题 #8957 #8959 @hi-rustin
    * 修复当 Kafka 消息过大导致同步出错时，在 Log 中记录了消息体的问题 #9031 @darraes
    * 修复下游 Kafka 滚动重启时 TiCDC 节点发生 panic 的问题 #9023 @asddongmen
    * 修复同步数据到存储服务时，下游 DDL 语句对应的 JSON 文件中没有记录表中字段默认值的问题 #9066 @CharlesCheung96

  - TiDB Lightning

    * 修复宽表导入时可能出现 OOM 的问题 #43728 @D3Hunter
    * 修复大数据量导入时报 `write to tikv with no leader returned` 错误的问题 #43055 @lance6716
    * 修复当数据文件中存在未闭合的 delimiter 时可能 OOM 的问题 #40400 @buchuitoudegou @lance6716

  - TiDB Binlog

    * 修复遇到状态为 CANCELED 的 DDL 时 TiDB Binlog 报错的问题 #1228 @okJiang

16.7.2 TiDB 6.5.2 Release Notes

发版日期：2023 年 4 月 21 日

TiDB 版本：6.5.2

试用链接：快速体验 | 生产部署 | 下载离线包

16.7.2.1 兼容性变更

- TiCDC 修复了 Avro 编码 FLOAT 类型数据错误的问题 #8490 @3AceShowHand

  在升级 TiCDC 集群到 v6.5.2 或更高的 v6.5.x 版本时，如果使用 Avro 同步的表包含 FLOAT 类型数据，请在升级前手动调整 Confluent Schema Registry 的兼容性策略为 None，使 changefeed 能够成功更新 schema。否则，在升级之后 changefeed 将无法更新 schema 并进入错误状态。

- 为了解决同步分区表到存储服务时可能丢数据的问题，TiCDC 配置项`sink.enable-partition-separator`默认值从 `false` 修改为 `true`，代表默认会将表中各个分区的数据分不同的目录来存储。建议保持该配置项为 `true` 以避免该问题。#8724 @CharlesCheung96

16.7.2.2 改进提升

- TiDB

  - Prepared Plan Cache 支持缓存 BatchPointGet 计划 #42125 @qw4990
  - Index Join 支持更多的 SQL 格式 #40505 @Yisaer
  - 将 Index Merge Reader 中的一些 Log 等级从 "info" 降低为 "debug" #41949 @yibin87
  - 优化带 Limit 的 Range 类型分区表的 `distsql_concurrency` 设置以降低查询延迟 #41480 @you06

- TiFlash

- 减少了 TiFlash 在读取过程中的任务调度对 CPU 的消耗 #6495 @JinheLin
- 提升默认参数下 BR 和 TiDB Lightning 向 TiFlash 导入数据的性能 #7272 @breezewish

- Tools
  - TiCDC
    - 发布 TiCDC Open API v2.0 #8743 @sdojjy
    - 引入 `gomemlimit` 以防止 TiCDC 出现 OOM 问题 #8675 @amyangfei
    - 采用 multi-statement 的方式优化批量执行 `UPDATE` 场景下的同步性能 #8057 @amyangfei
    - 支持在 redo applier 中拆分事务以提升 apply 吞吐，降低灾难场景的 RTO #8318 @CharlesCheung96
    - 支持在 redo log 里 apply DDL 事件 #8361 @CharlesCheung96
  - TiDB Lightning
    - 支持导入带有 BOM header 的 CSV 数据文件 #40744 @dsdashun

16.7.2.3 错误修复

- TiDB
  - 修复缓存表执行新增列操作后，新增列值为 `NULL` 而非列的默认值的问题 #42928 @lqs
  - 修复分区特别多并且带有 TiFlash 副本的分区表在执行 `TRUNCATE TABLE` 时，出现写冲突导致 DDL 重试的问题 #42940 @mjonss
  - 修复对于执行中的 `DROP TABLE` 操作，`ADMIN SHOW DDL JOBS` 的结果中缺少表名的问题 #42268 @tiancaiamao
  - 修复读取 cgroup 信息出错导致 TiDB Server 无法启动的问题，报错信息为 “can’t read file memory.stat from cgroup v1: open /sys/memory.stat no such file or directory” #42659 @hawkingrei
  - 修复在分区表上执行修改列操作时，数据截断没有正确发出警告的问题 #24427 @mjonss
  - 修复了生成执行计划过程中，因为获取的 InfoSchema 不一致而导致的 TiDB panic 的问题 #41622 @tiancaiamao
  - 修复了使用 DDL 修改浮点类型时，保持长度不变且减少小数位后，旧数据仍然保持原样的问题 #41281 @zimulala
  - 修复事务内执行 PointUpdate 之后，`SELECT` 结果不正确的问题 #28011 @zyguan
  - 修复在使用 Cursor Fetch 且在 Execute、Fetch、Close 之间运行其它语句后，Fetch 与 Close 命令可能会返回错误结果或造成 TiDB Panic 的问题 #40094 @YangKeao
  - 修复 `INSERT IGNORE` 和 `REPLACE` 语句对不修改 value 的 key 没有加锁的问题 #42121 @zyguan
  - 修复 TiFlash 执行中遇到生成列会报错的问题 #40663 @guo-shaoge
  - 修复当同一个 SQL 中出现多个不同的分区表时，TiDB 可能执行得到错误结果的问题 #42135 @mjonss
  - 修复在开启 Prepared Plan Cache 的情况下，索引全表扫可能会报错的问题 #42150 @fzzf678
  - 修复在开启 Prepared Plan Cache 时 Index Merge 可能得到错误结果的问题 #41828 @qw4990
  - 修复了设置 `max_prepared_stmt_count` 不生效的问题 #39735 @xuyifangreeneyes
  - 修复全局内存控制可能错误地 Kill 内存使用量小于 `tidb_server_memory_limit_sess_min_size` 的 SQL 的问题 #42662 @XuHuaiyu
  - 修复分区表动态裁剪模式下 Index Join 可能导致 panic 的问题 #40596 @tiancaiamao

- TiKV
  - 修复 TiKV 解析 cgroup path 没有正确解析：符号的问题 #14538 @SpadeA-Tang

- PD
  - 修复 PD 可能会非预期地向 Region 添加多个 Learner 的问题 #5786 @HunDunDM
  - 修复了切换 Placement Rule 时可能存在的 leader 分布不均衡的问题 #6195 @bufferflies
- TiFlash
  - 修复 TiFlash 无法识别生成列的问题 #6801 @guo-shaoge
  - 修复了 Decimal 除法在某些情况下最后一位未进位的问题 #7022 @LittleFall
  - 修复了 Decimal 转换在某些情况下进位错误的问题 #6994 @windtalker
  - 修复了开启 new collation 后 TopN/Sort 算子结果可能出错的问题 #6807 @xzhangxian1008
  - 修复由于不兼容 TiCDC 导致 TiFlash 进程失败的问题 #7212 @hongyunyan
- Tools
  - Backup & Restore (BR)
    * 修复当 TiDB 集群不存在 PITR 备份任务时，`resolve lock` 频率过高的问题 #40759 @joccau
    * 修复了在 PITR 恢复过程中等待 split Region 重试的时间不足的问题 #42001 @joccau
  - TiCDC
    * 修复同步到对象存储时，partition 分隔符不生效问题 #8581 @CharlesCheung96 @hi-rustin
    * 修复同步到对象存储时，表调度可能导致数据丢失的问题 #8256 @zhaoxinyu
    * 修复不可重入的 DDL 导致同步卡住的问题 #8662 @hicqu
    * 修复同步到对象存储时，TiCDC 扩缩容可能导致数据丢失的问题 #8666 @CharlesCheung96
    * 修复在部分场景中 `cgroup` 的内存限制不生效问题 #8588 @amyangfei
    * 修复 Redo log 在 apply 时，特殊情况下出现数据丢失的问题 #8591 @CharlesCheung96
    * 修复 `db sorter` 使用内存时未受 `cgroup memory limit` 限制的问题 #8588 @amyangfei
    * 修复同步数据时由于 `UPDATE` 和 `INSERT` 语句乱序可能导致 `Duplicate entry` 错误的问题 #8597 @sdojjy
    * 修复由于 PD 和 TiCDC 之间的网络隔离引起 TiCDC 程序异常退出的问题 #8562 @overvenus
    * 修复了 Kubernetes 上不能平滑升级 (graceful upgrade) TiCDC 集群的问题 #8484 @overvenus
    * 修复了当所有 Kafka server 不可访问时会导致 TiCDC server panic 的问题 #8523 @3AceShowHand
    * 修复了重启 changefeed 可能导致数据丢失或者 checkpoint 无法推进的问题 #8242 @overvenus

16.7.3 TiDB 6.5.1 Release Notes

发版日期：2023 年 3 月 10 日

TiDB 版本：6.5.1

试用链接：快速体验 | 生产部署 | 下载离线包

16.7.3.1 兼容性变更

- 自 2023 年 2 月 20 日起，新发布的 TiDB 和 TiDB Dashboard 版本（包含 6.5.1），默认关闭遥测功能，即默认不再收集使用情况信息分享给 PingCAP。如果升级至这些版本前使用默认的遥测配置，则升级后遥测功能处于关闭状态。具体的版本可参考TiDB 版本发布时间线。

  - 系统变量tidb_enable_telemetry 默认值由 ON 修改为 OFF。
  - TiDB 配置项enable-telemetry 默认值由 true 改为 false。
  - PD 配置项enable-telemetry 默认值由 true 改为 false。

- 从 v1.11.3 起，新部署的 TiUP 默认关闭遥测功能，即默认不再收集使用情况信息。如果从 v1.11.3 之前的 TiUP 版本升级至 v1.11.3 或更高 TiUP 版本，遥测保持升级前的开启或关闭状态。
- 由于可能存在正确性问题，分区表目前不再支持修改列类型 #40620 @mjonss
- TiKV 配置项advance-ts-interval 默认值由 1s 修改为 20s。你可以通过调整该配置项提高 Stale Read 数据的时效性（即减少延时），详情参见减少 Stale Read 延时。

16.7.3.2 改进提升

- TiDB
  - 从 v6.5.1 起，由 v1.4.3 或以上版本的 TiDB Operator 部署的 TiDB 全栈支持 IPv6 地址，这意味着 TiDB 可以支持更大的地址空间，并提供更好的安全性和网络性能。
    * 完全支持 IPv6 寻址：TiDB 支持使用 IPv6 地址进行所有网络连接，包括客户端连接、节点之间的内部通信以及与外部系统的通信。
    * 双栈支持：如果你尚未准备好完全切换到 IPv6，TiDB 也支持双栈网络。这意味着你可以在同一个 TiDB 集群中使用 IPv4 和 IPv6 地址，可以通过配置 IPv6 优先的方式来选择网络部署模式。

  IPv6 部署相关信息，请参考 TiDB on Kubernetes 用户文档。
  - 支持指定集群初次启动时的初始化 SQL 脚本 #35624 @morgo

    TiDB v6.5.1 新增 initialize-sql-file 配置项。集群初次启动时，你可通过命令行参数 --initialize-sql-file 指定执行的 SQL 脚本。该功能可用于修改系统变量的值、创建用户或分配权限等。更多信息，请参考用户文档。
  - 定期清理过期的 Region 缓存，避免内存泄漏和性能下降问题 #40461 @sticnarf
  - 新增 --proxy-protocol-fallbackable 配置项，控制是否启用 PROXY 协议回退模式。如果设置为 true，TiDB 可以接受非 PROXY 协议规范或者没有发送 PROXY 协议头的客户端连接 #41409 @blacktear23
  - 提升了 Memory Tracker 的准确度 #40900 #40500 @wshwsh12
  - 当执行计划缓存无法生效时，系统会通过 Warning 返回原因 #40210 @qw4990
  - 改进了条件优化器在进行越界估算时的策略 #39008 @time-and-fate
- TiKV
  - 支持在小于 1 core 的 CPU 下启动 TiKV #13586 #13752 #14017 @andreid-db
  - 将 Unified Read Pool 的线程上限 (readpool.unified.max-thread-count) 提高至 CPU 配额的 10 倍 #13690 @v01dstar
  - 为了节省跨域流量，resolved-ts.advance-ts-interval 的默认值从 "1s" 修改为 "20s" #14100 @overvenus
- TiFlash
  - 显著提升 TiFlash 在大数据量下的启动速度 #6395 @hehechen

- Tools
  - Backup & Restore (BR)
    * 优化 TiKV 端下载日志备份文件的并发度，提升常规场景下 PITR 恢复的性能 #14206 @YuJuncen
  - TiCDC
    * 默认打开 pull-based sink 功能提升系统的吞吐 #8232 @hi-rustin
    * 支持将 redo log 存储至兼容 GCS 或 Azure 协议的对象存储 #7987 @CharlesCheung96
    * 采用异步模式实现 MQ sink 和 MySQL sink，提升 sink 的吞吐能力 #5928 @amyangfei @CharlesCheung96

16.7.3.3 错误修复

- TiDB
  - 修复`pessimistic-auto-commit`配置项对 “Point Get” 查询不生效的问题 #39928 @zyguan
  - 修复 `INSERT` 或 `REPLACE` 语句在长会话连接中执行可能造成 Panic 的问题 #40351 @fanrenhoo
  - 修复了 `auto analyze` 导致 graceful shutdown 耗时长的问题 #40038 @xuyifangreeneyes
  - 修复了 DDL 在 ingest 过程中可能会发生数据竞争的问题 #40970 @tangenta
  - 修复了添加索引时可能导致数据竞争的问题 #40879 @tangenta
  - 修复了表 Region 比较多时因 Region 缓存失效导致加索引效率低下的问题 #38436 @tangenta
  - 修复了 TiDB 在初始化时有可能死锁的问题 #40408 @Defined2014
  - 修复了 TiDB 构造 key 范围时对 `NULL` 值处理不当，导致读取非预期数据的问题 #40158 @tiancaiamao
  - 修复了内存重用导致的在某些情况下系统变量的值会被错误修改的问题 #40979 @lcwangchao
  - 修复了在执行 TTL 任务时，如果表的主键包含 `ENUM` 类型的列任务会失败的问题 #40456 @lcwangchao
  - 修复了在添加唯一索引时有可能会 panic 的问题 #40592 @tangenta
  - 修复了并发 truncate 同一张表时，部分 truncate 操作无法被 MDL 阻塞的问题 #40484 @wjhuang2016
  - 修复了当动态裁剪模式下的分区表有 global binding 时，TiDB 重启失败的问题 #40368 @Yisaer
  - 修复使用 “Cursor Read” 方式读取数据时可能因为 GC 而报错的问题 #39447 @zyguan
  - 修复 `SHOW PROCESSLIST` 信息中未显示 `EXECUTE` 语句的问题 #41156 @YangKeao
  - 修复了 `globalMemoryControl` 在终止查询时可能会遇上 `KILL` 不结束的问题 #41057 @wshwsh12
  - 修复了 `indexMerge` 遇到错误后可能会导致 TiDB 崩溃的问题 #41047 #40877 @guo-shaoge @windtalker
  - 修复 `ANALYZE` 语句可能会被 `KILL` 终止的问题 #41825 @XuHuaiyu
  - 修复了 `indexMerge` 中可能会出现 goroutine 泄露的问题 #41545 #41605 @guo-shaoge
  - 修复了无符号的 `TINYINT/SMALLINT/INT` 和小于 0 的 `DECIMAL/FLOAT/DOUBLE` 类型比较时，结果可能出错的问题 #41736 @LittleFall
  - 修复了开启系统变量 `tidb_enable_reuse_chunk` 后可能会出现内存泄露的问题 #40987 @guo-shaoge
  - 修复了时区中的数据争用可能导致数据和索引不一致问题 #40710 @wjhuang2016
  - 修复了 `batch cop` 在执行过程中的 scan detail 信息不准确的问题 #41582 @you06
  - 修复 `cop` 并发度上限不受限制的问题 #41134 @you06
  - 修复 `cursor read` 中 `statement context` 被错误缓存的问题 #39998 @zyguan
  - 周期性清理过时的 Region 缓存以避免内存泄露和性能倒退 #40355 @sticnarf
  - 修复对包含 `year <cmp> const` 的查询使用 Plan Cache 时结果可能出错的问题 #41628 @qw4990
  - 修复查询区间太多且数据改动量大时估算误差可能较大的问题 #39593 @time-and-fate
  - 修复使用 Plan Cache 时部分条件无法被下推通过 Join 算子的问题 #40093 #38205 @qw4990
  - 修复 IndexMerge 计划在 `SET` 类型列上可能生成错误区间的问题 #41273 #41293 @time-and-fate

  - 修复 Plan Cache 处理 `int_col <cmp> decimal` 条件时可能缓存 FullScan 计划的问题 #40679 #41032 @qw4990
  - 修复 Plan Cache 处理 `int_col in (decimal...)` 条件时可能缓存 FullScan 计划的问题 #40224 @qw4990
  - 修复 `ignore_plan_cache` hint 对 INSERT 语句可能不生效的问题 #40079 #39717 @qw4990
  - 修复 Auto Analyze 可能阻碍 TiDB 退出的问题 #40038 @xuyifangreeneyes
  - 修复在分区表的 Unsigned Primary Key 上可能构造错误访问区间的问题 #40309 @winoros
  - 修复 Plan Cache 可能缓存 Shuffle 算子导致返回错误结果的问题 #38335 @qw4990
  - 修复在分区表上创建 Global Binding 后可能导致 TiDB 启动错误的问题 #40368 @Yisaer
  - 修复慢日志中查询计划算子可能缺失的问题 #41458 @time-and-fate
  - 修复错误下推包含虚拟列的 TopN 算子到 TiKV 或 TiFlash 导致结果错误的问题 #41355 @Dousir9
  - 修复添加索引时数据不一致的问题 #40698 #40730 #41459 #40464 #40217 @tangenta
  - 修复添加索引时出现 `PessimisticLockNotFound` 的报错问题 #41515 @tangenta
  - 修复添加唯一索引时误报重复键的问题 #41630 @tangenta
  - 修复 TiDB 使用 `paging` 时性能下降的问题 #40741 @solotzg

- TiKV

  - 修复 Resolved TS 导致网络流量升高的问题 #14092 @overvenus
  - 修复 TiDB 中事务在执行悲观 DML 失败后，再执行其他 DML 时，如果 TiDB 和 TiKV 之间存在网络故障，可能会造成数据不一致的问题 #14038 @MyonKeminta
  - 修复转换 `const Enum` 类型到其他类型时报错的问题 #14156 @wshwsh12
  - 修复 cop task 分页计算错误的问题 #14254 @you06
  - 修复 `batch cop` 模式下 `scan_detail` 不准确问题 #14109 @you06
  - 修复 Raft Engine 中的一个潜在错误，该错误可能导致 TiKV 检测到 Raft 数据损坏而无法重启 #14338 @tonyxuqqi

- PD

  - 修复 `replace-down-peer` 在特定条件下执行变慢的问题 #5788 @HundunDM
  - 修复 PD 可能会非预期地向 Region 添加多个 Learner 的问题 #5786 @HunDunDM
  - 修复 Region Scatter 任务会生成非预期的多余副本的问题 #5909 @HundunDM
  - 修复调用 `ReportMinResolvedTS` 过于频繁导致 PD OOM 的问题 #5965 @HundunDM
  - 修复 Region Scatter 接口可能导致 leader 分布不均匀的问题 #6017 @HunDunDM

- TiFlash

  - 修复半连接在计算笛卡尔积时，使用内存过量的问题 #6730 @gengliqi
  - 修复 TiFlash 日志搜索过慢的问题 #6829 @hehechen
  - 修复 TiFlash 在反复重启后由于误删文件而无法启动的问题 #6486 @JaySon-Huang
  - 修复 TiFlash 在添加新列后查询可能报错的问题 #6726 @JaySon-Huang
  - 修复 TiFlash 配置不支持 IPv6 的问题 #6734 @ywqzzy

- Tools

  - Backup & Restore (BR)

    * 修复 PD 与 TiDB server 的连接故障导致 PITR 备份进度不推进的问题 #41082 @YuJuncen
    * 修复 PD 与 TiKV 的连接故障导致 TiKV 不能监听 PITR 任务的问题 #14159 @YuJuncen
    * 修复 PITR 不支持 PD 集群配置变更的问题 #14165 @YuJuncen

    * 修复 PITR 功能不支持 CA-bundle 认证的问题 #38775 @3pointer
    * 修复 PITR 备份任务被删除时，存在备份信息残留导致新任务出现数据不一致的问题 #40403 @joccau
    * 修复使用 br debug 命令解析 backupmeta 文件导致的 panic 的问题 #40878 @MoCuishle28
    * 修复在某些情况下因无法获取 Region 大小导致恢复失败的问题 #36053 @YuJuncen
    * 修复当 TiDB 集群不存在 PITR 备份任务时，`resolve lock` 频率过高的问题 #40759 @joccau
    * 修复恢复数据到正在运行日志备份的集群，导致日志备份文件无法恢复的问题 #40797 @Leavrth
    * 修复全量备份失败后，从断点重启备份时 BR 会 panic 的问题 #40704 @Leavrth
    * 修复 PITR 错误被覆盖的问题 #40576@Leavrth
    * 修复 PITR 备份任务在 advance owner 与 gc owner 不同时 checkpoint 不推进的问题 #41806 @joccau

- TiCDC

    * 修复 changefeed 在 TiKV、TiCDC 节点扩缩容等特殊场景下卡住的问题 #8174 @hicqu
    * 修复 redo log 存储路径没做权限预检查的问题 #6335 @CharlesCheung96
    * 修复 redo log 容忍 S3 存储故障的时间过短的问题 #8089 @CharlesCheung96
    * 修复不能通过配置文件修改 `transaction_atomicity` 和 `protocol` 参数的问题 #7935 @CharlesCheung96
    * 修复在同步大量表时 checkpoint 不推进问题 #8004 @overvenus
    * 修复当同步的延迟过大时 apply redo log 可能会出现 OOM 的问题 #8085 @CharlesCheung96
    * 修复当开启 redo log 写 meta 时性能下降的问题 #8074 @CharlesCheung96
    * 修复没有配置大事务拆分时，同步数据超过 context deadline 的问题 #7982 @hi-rustin
    * 修复在 PD 异常时，暂停一个 changefeed 会错误设置状态的问题 #8330 @sdojjy
    * 修复下游为 TiDB 或 MySQL 时，无主键且非空唯一索引所在列指定了 CHARACTER SET 同步时可能会出现数据不一致的问题 #8420 @asddongmen
    * 修复 table scheduling 或 blackhole sink 存在 panic 的问题 #8024 #8142 @hicqu

- TiDB Data Migration (DM)

    * 修复 `binlog-schema delete` 命令执行失败的问题 #7373 @liumengya94
    * 修复当最后一个 binlog 是被 skip 的 DDL 时，checkpoint 不推进的问题 #8175 @D3Hunter
    * 修复当在某个表上同时指定 UPDATE 和非 UPDATE 类型的表达式过滤规则 `expression-filter` 时，所有 UPDATE 操作被跳过的问题 #7831 @lance6716

- TiDB Lightning

    * 修复 precheck 检查项有时无法监测到之前的导入失败遗留的脏数据的问题 #39477 @dsdashun
    * 修复 TiDB Lightning 在 split-region 阶段发生 panic 的问题 #40934 @lance6716
    * 修复冲突处理逻辑 (`duplicate-resolution`) 可能导致 checksum 不一致的问题 #40657 @gozssky
    * 修复在并行导入时，当除最后一个 TiDB Lightning 实例外的其他实例都遇到本地重复记录时，TiDB Lightning 可能会错误地跳过冲突处理的问题 #40923 @lichunzhu
    * 修复了在使用 Local Backend 模式导入数据时，当导入目标表的复合主键中存在 `auto_random` 列，且源数据中没有指定该列的值时，相关列没有自动生成数据的问题 #41454 @D3Hunter

16.7.4 TiDB 6.5.0 Release Notes

发版日期：2022 年 12 月 29 日

TiDB 版本：6.5.0

试用链接：快速体验 | 生产部署 | 下载离线包

TiDB 6.5.0 为长期支持版本 (Long-Term Support Release, LTS)。

与 TiDB 6.4.0-DMR 相比，TiDB 6.5.0 引入了以下关键特性：

> Tip:
>
> 与前一个 LTS（即 6.1.0 版本）相比，TiDB 6.5.0 还引入了6.2.0-DMR、6.3.0-DMR、6.4.0-DMR 中已发布的新功能、提升改进和错误修复。
>
> - 要了解 6.1.0 LTS 和 6.5.0 LTS 之间的完整变更，除了参阅当前页面的 release notes，还需参阅6.2.0-DMR release notes、6.3.0-DMR release notes、6.4.0-DMR release notes。
> - 要快速对比 6.1.0 LTS 和 6.5.0 LTS 的关键特性，可以查看TiDB 功能概览中的 `v6.1` 和 `v6.5` 列。

- 添加索引加速特性 GA，添加索引的性能约提升为 v6.1.0 的 10 倍。
- TiDB 全局内存控制特性 GA，通过`tidb_server_memory_limit` 即可管理全局内存阈值。
- 支持高性能、全局单调递增的`AUTO_INCREMENT` 列属性 GA，兼容 MySQL。
- `FLASHBACK CLUSTER TO TIMESTAMP` 特性新增对 TiCDC 和 PITR 的兼容性支持，该特性已 GA。
- 优化器引入的更精准的代价模型Cost Model Version 2 GA，同时优化器增强索引合并INDEX MERGE 功能对 AND 连接的表达式的支持。
- 支持下推 JSON_EXTRACT() 函数至 TiFlash。
- 支持密码管理策略，满足密码合规审计需求。
- TiDB Lightning 和 Dumpling 支持导入和导出压缩格式文件。
- TiDB Data Migration (DM) 的增量数据校验特性 GA。
- TiDB 快照备份支持断点续传，此外PITR 的恢复性能提升了 50%，通用场景下 RPO 降低到 5 分钟。
- TiCDC 同步数据到 Kafka，吞吐从 4000 行每秒提升到 35000 行每秒，复制延迟降低到 2 秒。
- 提供行级别Time to live (TTL) 管理数据生命周期（实验特性）。
- TiCDC 支持 Amazon S3、Azure Blob Storage、NFS 的对象存储（实验特性）。

16.7.4.1 新功能

16.7.4.1.1 SQL

- TiDB 添加索引的性能约提升为原来的 10 倍 (GA) #35983 @benjamin2037 @tangenta

  TiDB v6.3.0 引入了添加索引加速作为实验特性，提升了添加索引回填过程的速度。该功能在 v6.5.0 正式 GA 并默认打开，预期大表添加索引的性能约提升为 v6.1.0 的 10 倍。添加索引加速适用于单条 SQL 语句串行添加索引的场景，在多条 SQL 并行添加索引时仅对其中一条添加索引的 SQL 语句生效。

- 提供轻量级元数据锁，提升 DDL 变更过程 DML 的成功率 (GA) #37275 @wjhuang2016

  TiDB v6.3.0 引入了元数据锁作为实验特性，通过协调表元数据变更过程中 DML 语句和 DDL 语句的优先级，让执行中的 DDL 语句等待持有旧版本元数据的 DML 语句提交，尽可能避免 DML 语句的 `Information schema is changed` 错误。该功能在 v6.5.0 正式 GA 并默认打开，适用于各类 DDL 变更场景。当集群从 v6.5.0 之前的版本升级到 v6.5.0 及之后的版本时，TiDB 默认自动开启该功能。如果需要关闭该功能，你可以将系统变量`tidb_enable_metadata_lock` 设置为 OFF。

  更多信息，请参考用户文档。

- 支持通过 FLASHBACK CLUSTER TO TIMESTAMP 命令将集群快速回退到特定的时间点 (GA) #37197 #13303 @Defined2014 @bb7133 @JmPotato @Connor1996 @HuSharp @CalvinNeo

  TiDB v6.4.0 引入了FLASHBACK CLUSTER TO TIMESTAMP 语句作为实验特性，支持在 Garbage Collection (GC) life time 内快速回退整个集群到指定的时间点。该功能在 v6.5.0 新增对 TiCDC 和 PITR 的兼容性支持并正式 GA，适用于快速撤消 DML 误操作、支持集群分钟级别的快速回退、支持在时间线上多次回退以确定特定数据更改发生的时间。

  更多信息，请参考用户文档。

- 完整支持包含 INSERT、REPLACE、UPDATE 和 DELETE 的非事务 DML 语句 #33485 @ekexium

  在大批量的数据处理场景，单一大事务 SQL 处理可能对集群稳定性和性能造成影响。非事务 DML 语句将一个 DML 语句拆成多个 SQL 语句在内部执行。拆分后的语句将牺牲事务原子性和隔离性，但是对于集群的稳定性有很大提升。TiDB 从 v6.1.0 开始支持非事务 DELETE 语句，v6.5.0 新增对非事务 INSERT、REPLACE 和 UPDATE 语句的支持。

  更多信息，请参考非事务 DML 语句 和BATCH 语句。

- 支持 Time to live (TTL)（实验特性）#39262 @lcwangchao

  TTL 提供了行级别的生命周期控制策略。在 TiDB 中，设置了 TTL 属性的表会根据配置自动检查并删除过期的行数据。TTL 设计的目标是在不影响在线读写负载的前提下，帮助用户周期性且及时地清理不需要的数据。

  更多信息，请参考用户文档。

- 支持通过 INSERT INTO SELECT 语句保存 TiFlash 查询结果（实验特性）#37515 @gengliqi

  从 v6.5.0 起，TiDB 支持下推 INSERT INTO SELECT 语句中的 SELECT 子句（分析查询）到 TiFlash，你可以将 TiFlash 的查询结果方便地保存到 INSERT INTO 指定的 TiDB 表中供后续分析使用，起到了结果缓存（即结果物化）的效果。例如：

  ```
  INSERT INTO t2 SELECT Mod(x,y) FROM t1;
  ```

  在实验特性阶段，该功能默认关闭。要开启此功能，请设置系统变量tidb_enable_tiflash_read_for_write_stmt ↪ 为 ON。使用该特性时，INSERT INTO 指定的结果表没有特殊限制，你可以自由选择是否为该表添加 TiFlash 副本。该特性典型的使用场景包括：

  - 使用 TiFlash 做复杂分析
  - 需重复使用 TiFlash 查询结果或响应高并发的在线请求
  - 与查询的输入数据相比，所需的结果集比较小，推荐 100 MiB 以内

更多信息，请参考用户文档。

- 支持绑定历史执行计划（实验特性）#39199 @fzzf678

  受 SQL 语句执行时各种因素的影响，之前最优的执行计划偶尔会被新的执行计划替代，进而影响 SQL 性能。在这种场景下，最优的执行计划可能仍旧在 SQL 执行历史中，还没有被清除。

  在 v6.5.0 中，TiDB 扩展了CREATE [GLOBAL | SESSION] BINDING 语句中的绑定对象，支持根据历史执行计划创建绑定。当 SQL 语句的执行计划发生改变时，只要原来的执行计划仍然在 SQL 执行历史内存表（例如，statements_summary）中，就可以在 CREATE [GLOBAL | SESSION] BINDING 语句中通过指定 plan_digest 绑定原来的执行计划，快速恢复 SQL 性能。此方式可以简化执行计划突变问题的处理，提升运维效率。

  更多信息，请参考用户文档。

16.7.4.1.2 安全

- 支持密码复杂度策略 #38928 @CbcWestwolf

  TiDB 启用密码复杂度策略功能后，在用户设置密码时，TiDB 会检查密码长度、大写和小写字符个数、数字字符个数、特殊字符个数、密码字典匹配、是否与用户名相同等，以此确保用户设置了安全的密码。

  TiDB 支持密码强度检查函数 `VALIDATE_PASSWORD_STRENGTH()`，用于判定一个给定密码的强度。

  更多信息，请参考用户文档。

- 支持密码过期策略 #38936 @CbcWestwolf

  TiDB 支持密码过期策略，包括手动密码过期、全局级别自动密码过期、账户级别自动密码过期。启用密码过期策略功能后，用户必须定期修改密码，防止密码长期使用带来的泄露风险，提高密码安全性。

  更多信息，请参考用户文档。

- 支持密码重用策略 #38937 @keeplearning20221

  TiDB 支持密码重用策略，包括全局级别密码重用策略、账户级别密码重用策略。启用密码重用策略功能后，用户不能使用最近一段时间使用过的密码或最近几次使用过的密码，以此降低密码的重复使用带来的泄漏风险，提高密码安全性。

  更多信息，请参考用户文档。

- 支持密码连续错误限制登录策略 #38938 @lastincisor

  TiDB 启用密码连续错误限制登录策略功能后，当用户登录时，如果连续多次密码错误，账户将被临时锁定，达到锁定时间后将自动解锁。

  更多信息，请参考用户文档。

16.7.4.1.3 可观测性

- TiDB Dashboard 在 Kubernetes 环境支持独立 Pod 部署 #1447 @SabaPing

  TiDB v6.5.0 且 TiDB Operator v1.4.0 之后，在 Kubernetes 上支持将 TiDB Dashboard 作为独立的 Pod 部署。在 TiDB Operator 环境，可直接访问该 Pod 的 IP 来打开 TiDB Dashboard。

  独立部署 TiDB Dashboard，可以获得以下收益：

  - TiDB Dashboard 的计算将不会再对 PD 节点有压力，可以更好的保障集群运行。
  - 如果 PD 节点因异常不可访问，也还可以继续使用 TiDB Dashboard 进行集群诊断。
  - 在开放 TiDB Dashboard 到外网时，不用担心 PD 中的特权端口的权限问题，降低集群的安全风险。

更多信息，请参考 TiDB Operator 部署独立的 TiDB Dashboard。

- Performance Overview 面板中新增 TiFlash 和 CDC (Change Data Capture) 面板 #39230 @dbsid

  TiDB 从 v6.1.0 起在 Grafana 中引入了 Performance Overview 面板，为 TiDB、TiKV、PD 提供了系统级别的总体性能诊断入口。在 v6.5.0 中，Performance Overview 面板中新增了 TiFlash 和 CDC 面板。通过此次新增，从 v6.5.0 起，使用单个 Performance Overview 面板即可分析 TiDB 集群中所有组件的性能。

  TiFlash 和 CDC 面板重新编排了 TiFlash 和 TiCDC 相关的监控信息，可以帮助你大幅提高 TiFlash 和 TiCDC 的性能分析和故障诊断效率：

  - 通过TiFlash 面板，你可以直观地了解 TiFlash 集群的请求类型、延迟分析和资源使用概览。
  - 通过CDC 面板，你可以直观地了解 TiCDC 集群的健康状况、同步延迟、数据流和下游写入延迟等信息。

更多信息，请参考用户文档。

16.7.4.1.4 性能

- 索引合并INDEX MERGE 功能支持 AND 连接的表达式 #39333 @guo-shaoge @time-and-fate @hailanwhu

  在 v6.5.0 前，TiDB 只支持对 OR 连接词的过滤条件使用索引合并特性。自 v6.5.0 起，TiDB 支持对于在 WHERE 子句中使用 AND 连接的过滤条件使用索引合并特性。TiDB 的索引合并至此可以覆盖更多普遍的查询过滤条件组合，不再限定于并集 (OR) 关系。v6.5.0 仅支持优化器自动选择 OR 条件下的索引合并。要开启对于 AND 连接的索引合并，你需要使用USE_INDEX_MERGE Hint。

  关于索引合并功能的更多信息，请参阅v5.4.0 Release Notes，以及优化器相关的用户文档。

- 新增支持下推以下 JSON 函数至 TiFlash #39458 @yibin87

  - `->`
  - `->>`
  - `JSON_EXTRACT()`

JSON 格式为应用设计提供了灵活的建模方式，目前越来越多的应用采用 JSON 格式进行数据交换和数据存储。通过将 JSON 函数下推至 TiFlash，你可以提高 JSON 类型数据的分析效率，拓展 TiDB 实时分析的应用场景。

- 新增支持下推以下字符串函数至 TiFlash #6115 @xzhangxian1008

  - `regexp_like`
  - `regexp_instr`
  - `regexp_substr`

- 新增全局 Hint 干预视图内查询计划的生成 #37887 @Reminiscent

  部分视图访问的场景需要用 Hint 对视图内查询的执行计划进行干预，以获得最佳性能。在 v6.5.0 中，TiDB 允许针对视图内的查询块添加全局 Hint，使查询中定义的 Hint 能够在视图内部生效。该特性为包含复杂视图嵌套的 SQL 提供 Hint 的注入手段，增强了执行计划控制能力，进而稳定复杂 SQL 的执行性能。全局 Hint 通过查询块命名和Hint 引用来开启。

  更多信息，请参考用户文档。

- 支持将分区表的排序操作下推至 TiKV #26166 @winoros

  分区表特性在 v6.1.0 正式 GA 后，TiDB 仍然在持续提升分区表相关的性能。在 v6.5.0 中，TiDB 支持将 ORDER BY 和 LIMIT 等排序操作下推至 TiKV 进行计算和过滤，降低网络 I/O 的开销，提升了使用分区表时 SQL 的性能。

- 优化器引入更精准的代价模型 Cost Model Version 2 (GA) #35240 @qw4990

  TiDB v6.2.0 引入了代价模型Cost Model Version 2 作为实验特性，通过更准确的代价估算方式，有利于最优执行计划的选择。尤其在部署了 TiFlash 的情况下，Cost Model Version 2 自动选择合理的存储引擎，避免过

多的人工介入。经过一段时间真实场景的测试，这个模型在 v6.5.0 正式 GA。新创建的集群将默认使用 Cost Model Version 2。对于升级到 v6.5.0 的集群，由于 Cost Model Version 2 可能会改变原有的执行计划，在经过充分的性能测试之后，你可以通过设置变量`tidb_cost_model_version = 2`使用新的代价模型。

Cost Model Version 2 成为正式功能大幅提升了 TiDB 优化器的整体能力，并使 TiDB 切实地向更加强大的 HTAP 数据库演进。

更多信息，请参考用户文档。

- TiFlash 对获取表行数的操作进行优化 #37165 @elsa0520

  在数据分析的场景中，通过无过滤条件的 `COUNT(*)` 获取表的实际行数是一个常见操作。TiFlash 在 v6.5.0 中优化了 `COUNT(*)` 的改写，自动选择带有“非空”属性且列定义最短的列进行计数，这样可以有效降低 TiFlash 上发生的 I/O 数量，进而提升获取表行数的执行效率。

16.7.4.1.5 稳定性

- TiDB 全局内存控制成为正式功能 (GA) #37816 @wshwsh12

  TiDB v6.4.0 引入了全局内存控制作为实验特性。自 v6.5.0 起，全局内存控制成为正式功能，能够跟踪到 TiDB 中主要的内存消耗。当全局内存消耗达到`tidb_server_memory_limit` 所定义的阈值时，TiDB 会尝试 GC 或取消 SQL 操作等方法限制内存使用，保证 TiDB 的稳定性。

  需要注意的是，会话中事务所消耗的内存（由配置项`txn-total-size-limit` 设置最大值）如今被内存管理模块跟踪：当单个会话的内存消耗达到系统变量`tidb_mem_quota_query` 所定义的阈值时，将会触发系统变量`tidb_mem_oom_action` 所定义的行为（默认为 `CANCEL`，即取消操作）。为了保证向前兼容，当配置`txn-total-size-limit` 为非默认值时，TiDB 仍旧会保证事务可以使用到 `txn-total-size-limit` 所设定的内存量而不被取消。

  在使用 v6.5.0 及以上版本时，建议移除配置项`txn-total-size-limit`，取消对事务内存做单独的限制，转而使用系统变量`tidb_mem_quota_query` 和`tidb_server_memory_limit` 对全局内存进行管理，从而提高内存的使用效率。

  更多信息，请参考用户文档。

16.7.4.1.6 易用性

- 完善 `EXPLAIN ANALYZE` 输出结果中 TiFlash 的 TableFullScan 算子的执行信息 #5926 @hongyunyan

  `EXPLAIN ANALYZE` 语句可以输出执行计划及运行时的统计信息。在 v6.5.0 中，TiFlash 对 TableFullScan 算子的执行信息进行了完善，补充了 DMFile 相关的执行信息。你可以更加直观地查看 TiFlash 的数据扫描状态信息，方便进行性能分析。

  更多信息，请参考用户文档。

- 支持将执行计划打印为 JSON 格式 #39261 @fzzf678

  在 v6.5.0 中，TiDB 扩展了执行计划的打印格式。通过在 `EXPLAIN` 语句中指定 `FORMAT = "tidb_json"` 能够将 SQL 的执行计划以 JSON 格式输出。借助这个能力，SQL 调试工具和诊断工具能够更方便准确地解读执行计划，进而提升 SQL 诊断调优的易用性。

  更多信息，请参考用户文档。

16.7.4.1.7 MySQL 兼容性

- 支持高性能、全局单调递增的 AUTO_INCREMENT 列属性 (GA) #38442 @tiancaiamao

  TiDB v6.4.0 引入了 AUTO_INCREMENT 的 MySQL 兼容模式作为实验特性，通过中心化分配自增 ID，实现了自增 ID 在所有 TiDB 实例上单调递增。使用该特性能够更容易地实现查询结果按自增 ID 排序。该功能在 v6.5.0 正式 GA。使用该功能的单表写入 TPS 预期超过 2 万，并支持通过弹性扩容提升单表和整个集群的写入吞吐。要使用 MySQL 兼容模式，你需要在建表时将 AUTO_ID_CACHE 设置为 1。

  ```
  CREATE TABLE t(a int AUTO_INCREMENT key) AUTO_ID_CACHE 1;
  ```

  更多信息，请参考用户文档。

16.7.4.1.8 数据迁移

- 支持导出和导入 gzip、snappy、zstd 三种压缩格式的 SQL、CSV 文件 #38514 @lichunzhu

  Dumpling 支持将数据导出为 gzip、snappy、zstd 三种压缩格式的 SQL、CSV 的压缩文件。TiDB Lightning 也支持导入这些格式的压缩文件。

  有这个功能之前，导出数据或者导入数据都需要较大的存储空间，用于存储已经导出或即将导入的 CSV 和 SQL 文件，需要较高的存储成本。该功能发布后，通过压缩数据文件，可以大幅降低存储成本。

  更多信息，请参考用户文档。

- 优化了 binlog 解析能力 #924 @gmhdbjd

  该功能允许过滤掉不在迁移任务里的库和表对象的 binlog event，不做解析，从而提升解析效率和稳定性。该策略在 v6.5.0 版本默认生效，无需额外操作。

  有这个功能之前，即使仅迁移几张表，也需要解析上游整个 binlog 文件，即仍要解析该 binlog 文件中不需要迁移的表的 binlog event，效率较低。同时，如果不在迁移任务里的库表的 binlog event 不支持解析，还会导致任务失败。推出该功能后，通过只解析在迁移任务里的库表对象的 binlog event，可以大大提升 binlog 解析效率，提升任务稳定性。

- Disk quota 功能 GA #446 @buchuitoudegou

  你可以为 TiDB Lightning 配置磁盘配额 (disk quota)。当磁盘配额不足时，TiDB Lightning 会暂停读取源数据以及写入临时文件，而是优先将已经完成排序的 key-value 写入 TiKV。TiDB Lightning 删除本地临时文件后，再继续导入过程。

  有这个功能之前，TiDB Lightning 在使用物理模式导入数据时，会在本地磁盘创建大量的临时文件，用来对原始数据进行编码、排序、分割。当用户本地磁盘空间不足时，TiDB Lightning 会由于写入文件失败而报错退出。推出该功能后，可避免 TiDB Lightning 任务写满本地磁盘。

  更多信息，请参考用户文档。

- DM 增量数据校验的功能 GA #4426 @D3Hunter

  在将增量数据从上游迁移到下游数据库的过程中，数据的流转有小概率导致错误或者丢失的情况。对于需要依赖强数据一致的场景，如信贷、证券等业务，你可以在数据迁移完成之后再对数据进行全量校验，确保数据的一致性。然而，在某些增量复制的业务场景下，上游和下游的写入是持续的、不会中断的。由于上下游的数据在不断变化，导致用户难以对表里的全部数据进行一致性校验。

  过去，需要中断业务才能进行全量数据校验，会影响业务。推出该功能后，你无需中断业务即可实现增量数据校验。

  更多信息，请参考用户文档。

16.7.4.1.9 数据共享与订阅

- TiCDC 支持输出变更数据至 storage sink（实验特性）#6797 @zhaoxinyu

  TiCDC 支持将 changed log 输出到 Amazon S3、Azure Blob Storage、NFS，以及兼容 Amazon S3 协议的存储服务中。Cloud storage 价格便宜，使用方便。对于不使用 Kafka 的用户，可以选择使用 storage sink。使用该功能，TiCDC 会将 changed log 保存到文件，发送到存储系统中。用户自研的消费程序定时从存储系统读取新产生的 changed log 进行数据处理。

  Storage sink 支持格式为 canal-json 和 csv 的 changed log。更多信息，请参考用户文档。

- TiCDC 支持在两个 TiDB 集群之间进行双向复制 #38587 @xiongjiwei @asddongmen

  TiCDC 支持在两个 TiDB 集群之间进行双向复制。如果业务上需要构建异地多活的 TiDB 集群架构，可以使用该功能作为 TiDB 多活的解决方案。只要为 TiDB 集群到另一个 TiDB 集群的 TiCDC 同步任务配置 `bdr-mode = true` 参数，就可以实现两个 TiDB 集群之间的数据相互复制。

  更多信息，请参考用户文档。

- TiCDC 支持在线更新 TLS 证书 tiflow#7908 @CharlesCheung96

  为确保系统数据安全，用户会对系统使用的证书设置相应的过期策略，经过固定的时间后会将系统使用的证书更换成新证书。TiCDC v6.5.0 支持在线更新 TLS 证书，在不影响同步任务的前提下，TiCDC 会自动检测和更新证书，无需用户手动操作，满足用户对证书更新的需求。

- TiCDC 性能提升 #7540 #7478 #7532 @sdojjy [@3AceShowHand](https://github.com/3AceShowHand)

  在 TiDB 场景测试验证中，TiCDC 的性能得到了比较大的提升。

  在同步到 Kafka 的场景中，单台 TiCDC 节点能处理的最大行变更吞吐可以达到 30K rows/s，同步延迟降低到 10s。即使在常规的 TiKV/TiCDC 滚动升级场景，同步延迟也小于 30s。

  在容灾场景测试中，打开 TiCDC redo log 和 Syncpoint 后，吞吐从 4000 行每秒提升到 35000 行每秒，容灾复制延迟可以保持在 2s。

16.7.4.1.10 备份和恢复

- TiDB 快照备份支持断点续传 #38647 @Leavrth

  TiDB 快照备份功能支持断点续传。当 BR 遇到可恢复的错误时会进行重试，但是超过固定重试次数之后会备份退出。断点续传功能允许对持续更长时间的可恢复故障进行重试恢复，比如几十分钟的网络故障。

  需要注意的是，如果你没有在 BR 退出后一个小时内完成故障恢复，那么还未备份的快照数据可能会被 GC 机制回收，从而造成备份失败。更多信息，请参考用户文档。

- PITR 性能大幅提升 [@joccau](https://github.com/joccau)

PITR 恢复的日志恢复阶段，单台 TiKV 的恢复速度可以达到 9 MiB/s，提升了 50%，并且恢复速度可扩展，有效地降低容灾场景的 RTO 指标；容灾场景的 RPO 优化到 5 分钟，在常规的集群运维，如滚动升级，单 TiKV 故障等场景下，可以达到 RPO = 5 min 的目标。

- TiKV-BR 工具 GA，支持 RawKV 的备份和恢复 #67 @pingyu @haojinming

  TiKV-BR 是一个 TiKV 集群的备份和恢复工具。TiKV 可以独立于 TiDB，与 PD 构成 KV 数据库，此时的产品形态为 RawKV。TiKV-BR 工具支持对使用 RawKV 的产品进行备份和恢复，也支持将 TiKV 集群中的数据从 API V1 备份为 API V2 数据，以实现 TiKV 集群api-version 的升级。

  更多信息，请参考用户文档。

16.7.4.2 兼容性变更

16.7.4.2.1 系统变量

变量名	修改类型	描述
tidb_enable_~~async~~_pessimistic_txn	~~废弃~~	从 v6.5.0 起，该变量被废弃，TiDB 会默认使用元数据锁机制解决 Information schema is changed 报错的问题。
tidb_enable_~~outer~~_join_reorder	修改	经进一步的测试后，该变量默认值从 OFF 修改为 ON，表示默认启用 Outer Join 的Join Reorder 算法。

变量名	修改类型	描述
tidb_cost_model_version	修改	经进一步的测试后，该变量默认值从 1 修改为 2，表示默认使用 Cost Model Version 2 进行索引选择和算子选择。

变量名	修改类型	描述
tidb_enable_gc_aware_memory_track ↪	修改	该变量默认值由 ON 修改为 OFF。由于在测试中发现 GC-Aware memory track 不准确，导致 Analyze 追踪到的内存过大的情况，因此先关闭内存追踪。在 Golang 1.19 下，GC-Aware memory track 追踪的内存对整体内存的影响变小。
tidb_enable_metadata_lock ↪	修改	经进一步的测试后，该变量默认值从 OFF 修改为 ON，表示默认开启元数据锁。

变量名	修改类型	描述
tidb_enable_tiflash_read_for_write_stmt ↪	修改	该变量从 v6.5.0 开始生效，默认值为 OFF，用来控制包含增删改的 SQL 语句中的读取操作能否下推到 TiFlash。
tidb_ddl_enable_fast_reorg ↪	修改	经进一步的测试后，该变量默认值从 OFF 修改为 ON，表示默认开启创建索引加速功能。

变量名	修改类型	描述
tidb_mem_quota_query	修改	在 v6.5.0 之前的版本中，该变量用来设置单条查询的内存使用限制。在 v6.5.0 及之后的版本中，为了对 DML 语句的内存进行更准确地控制，该变量用来设置单个会话整体的内存使用限制。

变量名	修改类型	描述
tidb_replica_read	修改	从 v6.5.0 起，为了优化各个 TiDB 节点的负载均衡，当该变量的值为 closest-adaptive 时，如果一个读请求的预估返回结果大于或等于tidb_adaptive_closest_read_threshold，在每个可用区中 closest-adaptive 配置实际生效的 TiDB 节点数总是与包含 TiDB 节点最少的可用区中的 TiDB 节点数相同。对于生效的节点，TiDB 会优先选择分布在同一可用区的副本执行

变量名	修改类型	描述
tidb_server_memory_limit	修改	该变量默认值由 0 修改为 80%，因为 TiDB 全局内存控制特性 GA，该调整默认开启 TiDB 实例的内存限制，并将默认的内存限制设为总内存的 80%。
default_password_lifetime	新增	用于设置全局自动密码过期策略，要求用户定期修改密码。默认值为 0，表示禁用全局自动密码过期策略。

变量名	修改类型	描述
disconnect_on_expired_password	新增	该变量是一个只读变量，用来显示 TiDB 是否会直接断开密码已过期用户的连接。
password_history	新增	基于密码更改次数的密码重用策略，不允许用户重复使用最近设置次数内使用过的密码。默认值为 0，表示禁用基于密码更改次数的密码重用策略。

变量名	修改类型	描述
password_reuse_interval	新增	基于经过时间限制的密码重用策略，不允许用户重复使用最近设置天数内使用过的密码。默认值为 0，表示禁用基于密码更改时间内的密码重用策略。
tidb_auto_build_stats_concurrency	新增	该变量用于设置执行统计信息自动更新的并发度，默认值为 1。

变量名	修改类型	描述
tidb_cdc_write_source	新增	当变量非 0 时，该 SESSION 写入的数据将被视为是由 TiCDC 写入的。这个变量仅由 TiCDC 设置，任何时候都不应该手动调整该变量。
tidb_index_merge_intersection_concurrency	新增	这个变量用来设置索引合并进行交集操作时的最大并发度，仅在以动态裁剪模式访问分区表时有效。
tidb_source_id	新增	设置在双向复制系统内不同集群的 ID。

变量名	修改类型	描述
tidb_sysproc_scan_concurrency ↪	新增	该变量用于设置 TiDB 执行内部 SQL 语句（例如统计信息自动更新）时 scan 操作的并发度，默认值为 1。
tidb_ttl_delete_batch_size ↪	新增	这个变量用于设置 TTL 任务中单个删除事务中允许删除的最大行数。
tidb_ttl_delete_rate_limit ↪	新增	这个变量用于限制在 TTL 任务中单个节点每秒允许 DELETE 语句执行的最大次数。当此变量设置为 0 时，则表示不做限制。

变量名	修改类型	描述
tidb_ttl_delete_worker_count	新增	这个变量用于设置每个 TiDB 节点上 TTL 删除任务的最大并发数。
tidb_ttl_job_enable	新增	这个变量用于控制是否启动 TTL 后台清理任务。如果设置为 OFF，所有具有 TTL 属性的表会自动停止清理过期数据。

变量名	修改类型	描述
tidb_ttl_job_interval	新增	这个变量用于控制 TTL 后台清理任务的调度周期。比如，如果当前值设置成了 1h0m0s，则代表每张设置了 TTL 属性的表会每小时清理一次过期数据。
tidb_ttl_job_schedule_window_start_time	新增	这个变量用于控制 TTL 后台清理任务的调度窗口的起始时间。请谨慎调整此参数，过小的窗口有可能会造成过期数据的清理无法完成。

变量名	修改类型	描述
tidb_ttl_job_schedule_window_end_time ↪	新增	这个变量用于控制 TTL 后台清理任务的调度窗口的结束时间。请谨慎调整此参数，过小的窗口有可能会造成过期数据的清理无法完成。
tidb_ttl_scan_batch_size ↪	新增	这个变量用于设置 TTL 任务中用来扫描过期数据的每个 SELECT 语句的 LIMIT 的值。
tidb_ttl_scan_worker_count ↪	新增	这个变量用于设置每个 TiDB 节点 TTL 扫描任务的最大并发数。

变量名	修改类型	描述
validate_password ↪ . ↪ check_user_name ↪	新增	密码复杂度策略检查项，设置的用户密码不允许密码与当前会话账户的用户名部分相同。只有validate_password ↪ . ↪ enable ↪ 开启时，该变量才生效。默认值为 ON。

变量名	修改类型	描述
validate_password.dictionary	新增	密码复杂度策略检查项，密码字典功能，设置的用户密码不允许包含字典中的单词。只有validate_password.enable 开启且validate_password.policy设置为 2 (STRONG)时，该变量才生效。默认值为空。
validate_password.enable	新增	密码复杂度策略检查的开关，设置为 ON 后，TiDB 才进行密码复杂度检查。默认值为 OFF。

变量名	修改类型	描述
validate_password.length	新增	密码复杂度策略检查项，限定了用户密码最小长度。只有validate_password.enable 开启时，该变量才生效。默认值为 8。
validate_password.mixed_case_count	新增	密码复杂度策略检查项，限定了用户密码中大写字符和小写字符的最小数量。只有validate_password.enable 开启且validate_password.policy 大于或等于 1 (MEDIUM) 时，该变量才生效。默认值为 1。

变量名	修改类型	描述
validate_password.number_count	新增	密码复杂度策略检查项，限定了用户密码中数字字符的最小数量。只有validate_password.enable 开启且validate_password.policy大于或等于 1 (MEDIUM)时，该变量才生效。默认值为 1。
validate_password.policy	新增	密码复杂度策略检查的强度，强度等级分为 [0, 1, 2]。只有validate_password.enable 开启时，该变量才生效。默认值为 1。

变量名	修改类型	描述
validate_password.special_char_count	新增	密码复杂度策略检查项，限定了用户密码中特殊字符的最小数量。只有validate_password.enable 开启且validate_password.policy 大于或等于 1 (MEDIUM) 时，该变量才生效。默认值为 1。

16.7.4.2.2 配置文件参数

配置文件	配置项	修改类型	描述
TiDB	server-memory-quota	废弃	自 v6.5.0 起，该配置项被废弃。请使用tidb_server_memory_limit 系统变量进行设置。

配置文件	配置项	修改类型	描述
TiDB	disconnect-on-expired-password	新增	该配置用于控制 TiDB 服务端是否直接断开密码已过期用户的连接，默认值为 true，表示 TiDB 服务端将直接断开密码已过期用户的连接。
TiKV	raw-min-ts-outlier-threshold	删除	从 v6.4.0 起，该配置项被废弃。从 v6.5.0 起，该配置项被删除。
TiKV	cdc.min-ts-interval	修改	为了降低 CDC 延迟，该配置的默认值从 1s 修改为 200ms。
TiKV	memory-use-ratio	新增	表示 PITR 日志恢复功能中可用内存与系统总内存的占比。

配置文件	配置项	修改类型	描述
TiCDC	sink. ↪ terminator ↪	新增	换行符，用来分隔两个数据变更事件。默认值为空，表示使用 \r\n 作为换行符。
TiCDC	sink.date ↪ - ↪ separator ↪	新增	文件路径的日期分隔类型。可选类型有 none、year、month、day。默认值为 none，即不使用日期分隔。
TiCDC	sink. ↪ enable ↪ - ↪ partition ↪ - ↪ separator ↪	新增	是否使用 partition 作为分隔字符串，默认值为 false，即一张表中各个 partition 的数据不会分不同的目录来存储。
TiCDC	sink.csv. ↪ delimiter ↪	新增	字段之间的分隔符。必须为 ASCII 字符，默认值为 ,。

配置文件	配置项	修改类型	描述
TiCDC	sink.csv. ↪ quote	新增	用于包裹字段的引号字符。空值代表不使用引号字符。默认值为 "。
TiCDC	sink.csv. ↪ null	新增	用于确定 CSV 列为 null 时将以什么字符来表示。默认值为 \N。
TiCDC	sink.csv. ↪ include ↪ - ↪ commit ↪ -ts	新增	是否在 CSV 行中包含 commit-ts。默认值为 false。

16.7.4.2.3 其他

- 从 v6.5.0 起，mysql.user 表新增 Password_reuse_history 和 Password_reuse_time 两个字段。
- 从 v6.5.0 起，添加索引加速功能默认开启。该功能和单条 ALTER TABLE 语句增删改多个列或索引功能未完全兼容，在使用索引加速功能添加唯一索引时，请避免在单条语句添加唯一索引的同时操作其他列或者索引对象。同时，该功能与PITR (Point-in-time recovery) 不兼容。在使用索引加速功能时，需要确保后台没有启动 PITR 备份任务，否则可能会出现非预期结果。详情请参考tidb_ddl_enable_fast_reorg。

16.7.4.3 废弃功能

从 v6.5.0 起，废弃 v4.0.7 版本引入的 AMEND TRANSACTION 机制，并使用元数据锁替代。

16.7.4.4 改进提升

- TiDB
  - 对于 BIT 和 CHAR 类型的列，使 INFORMATION_SCHEMA.COLUMNS 的显示结果与 MySQL 一致 #25472 @hawkingrei
  - 优化 TiDB 在 TiFlash MPP 模式下对 TiFlash 节点的探活机制，缓解节点异常时对性能的影响 #39686 @hackersean
- TiKV

  - 当剩余空间不足时停止 Raft Engine 的写入以避免硬盘空间耗尽 #13642 @jiayang-zheng
  - 支持将 `json_valid` 函数下推至 TiKV #13571 @lizhenhuan
  - 支持在一个备份请求中同时备份多个范围的数据 #13701 @Leavrth
  - 更新 rusoto 库以支持备份到 AWS 的 Asia Pacific (Jakarta) 区域 (ap-southeast-3) #13751 @3pointer
  - 减少悲观事务冲突 #13298 @MyonKeminta
  - 缓存外部存储对象以提升恢复性能 #13798 @YuJuncen
  - 在专用线程中运行 CheckLeader 以缩短 TiCDC 的复制延迟 #13774 @overvenus
  - Checkpoint 支持拉取模式 #13824 @YuJuncen
  - 升级 crossbeam-channel 以优化发送端的自旋问题 #13815 @sticnarf
  - TiKV 支持批量处理 Coprocessor 任务 #13849 @cfzjywxk
  - 故障恢复时通知 TiKV 唤醒休眠的 Region 以减少等待时间 #13648 @LykxSassinator
  - 通过代码优化减少内存申请的大小 #13827 @BusyJay
  - 引入 Raft extension 以提升代码可扩展性 #13827 @BusyJay
  - tikv-ctl 支持查询某个 key 范围中包含哪些 Region #13760 [@HuSharp](https://github.com/HuSharp)
  - 改进持续对特定行只加锁但不更新的情况下的读写性能 #13694 [@sticnarf](https://github.com/sticnarf)

- PD

  - 优化锁的粒度以减少锁争用，提升高并发下心跳的处理能力 #5586 @rleungx
  - 优化调度器在大规模集群下的性能，提升调度策略生产速度 #5473 @bufferflies
  - 提高 PD 加载 Region 的速度 #5606 @rleungx
  - 优化心跳处理过程，减少不必要的开销 #5648 @rleungx
  - 增加了自动清理 tombstone store 的功能 #5348 @nolouch

- TiFlash

  - 提升了 TiFlash 在 SQL 端没有攒批场景时的写入性能 #6404 @lidezhu
  - 在 `explain analyze` 结果中增加更多关于 TableFullScan 算子的信息 #5926 @hongyunyan

- Tools

  - TiDB Dashboard

    * 在慢查询页面新增以下三个字段：是否由 `prepare` 语句生成、查询计划是否来自缓存、查询计划是否来自绑定 #1451 @shhdgit

  - Backup & Restore (BR)

    * 优化清理备份日志数据时 BR 的内存使用 #38869 @Leavrth
    * 修复恢复过程中由于 PD leader 切换导致恢复失败的问题 #36910 @MoCuishle28
    * 日志备份的 TLS 功能使用 OpenSSL 协议，提升 TLS 的兼容性 #13867 @YuJuncen

  - TiCDC

    * 提升 Kafka 相关协议的编码性能 #7540 #7532 #7543 @3AceShowHand @sdojjy

  - TiDB Data Migration (DM)

    * 通过不再解析黑名单中表的数据提升 DM 同步数据的性能 #7622 @GMHDBJD
    * 通过异步写与批量写的方式提升 DM relay 的写数据效率 #7580 @GMHDBJD
    * 改进 DM 前置检查的错误提示信息 #7621 @buchuitoudegou
    * 改进 DM 针对老版本 MySQL 使用 `SHOW SLAVE HOSTS` 获取结果时的兼容性 #5017 @lyzx2001

16.7.4.5 错误修复

- TiDB
  - 修复 chunk 复用功能在部分情况下内存 chunk 被错误使用的问题 #38917 @keeplearning20221
  - 修复 `tidb_constraint_check_in_place_pessimistic` 可能被全局设置影响内部 session 的问题 #38766 @ekexium
  - 修复了 `AUTO_INCREMENT` 列无法和 `CHECK` 约束一起使用的问题 #38894 @YangKeao
  - 修复使用 `INSERT IGNORE INTO` 往 `SMALLINT` 类型的自增列插入 `STRING` 类型的数据会报错的问题 #38483 @hawkingrei
  - 修复了重命名分区表的分区列操作出现空指针报错的问题 #38932 @mjonss
  - 修复了修改分区表的分区列导致 DDL 卡死的问题 #38530 @mjonss
  - 修复了从 v4.0.16 升级到 v6.4.0 后 `ADMIN SHOW JOB` 操作崩溃的问题 #38980 @tangenta
  - 修复了 `tidb_decode_key` 函数未正确处理分区表编码的问题 #39304 @Defined2014
  - 修复了日志轮转时，gRPC 的错误日志信息未被重定向到正确的日志文件的问题 #38941 @xhebox
  - 修复了 `BEGIN; SELECT... FOR UPDATE;` 点查在读数据存储引擎 (`tidb_isolation_read_engines`) 未配置 TiKV 时生成非预期执行计划的问题 #39344 @Yisaer
  - 修复了错误地下推 `StreamAgg` 到 TiFlash 导致结果错误的问题 #39266 @fixdb
- TiKV
  - 修复 Raft Engine ctl 中的错误 #11119 @tabokie
  - 修复 tikv-ctl 执行 `compact raft` 命令时报错的问题 #13515 @guoxiangCN
  - 修复当启用 TLS 时日志备份无法使用的问题 #13867 @YuJuncen
  - 修复对 Geometry 字段类型的支持问题 #13651 @dveeden
  - 修复当未启用 new collation 时 `LIKE` 操作符中的 _ 无法匹配非 ASCII 字符的问题 #13769 @YangKeao
  - 修复 tikv-ctl 执行 `reset-to-version` 命令时被终止的问题 #13829 @tabokie
- PD
  - 修复热点调度配置在没有修改的情况下不持久化的问题 #5701 @HunDunDM
  - 修复 `rank-formula-version` 在升级过程中没有保持升级前的配置的问题 #5698 @HunDunDM
- TiFlash
  - 修复 TiFlash 重启后 delta 层的小文件无法合并 (compact) 的问题 #6159 @lidezhu
  - 修复 TiFlash File Open OPS 过高的问题 #6345 @JaySon-Huang
- Tools
  - Backup & Restore (BR)
    * 修复 BR 删除日志备份数据时，会删除不应被删除的数据的问题 #38939 @Leavrth
    * 修复数据库或数据表中使用旧的排序规则框架时，数据恢复失败的问题#39150 @MoCuishle28
    * 修复阿里云和华为云与 S3 存储不完全兼容导致的备份失败问题 #39545 @3pointer
  - TiCDC
    * 修复 PD leader crash 时 TiCDC 卡住的问题 #7470 @zeminzhou
    * 修复在执行 DDL 后，暂停然后恢复 changefeed 会导致数据丢失的问题 #7682 @asddongmen

    * 修复存在高版本 TiFlash 时，TiCDC 会误报错的问题 #7744 @overvenus
    * 修复下游网络发生异常时 sink 模块卡住的问题 #7706 @hicqu
    * 修复用户快速删除、创建同名同步任务可能导致的数据丢失问题 #7657 @overvenus

  - TiDB Data Migration (DM)

    * 修复在上游开启 GTID mode 且无数据时，无法启动 all mode 任务的问题 #7037 @liumengya94
    * 修复当 DM worker 即将退出时新 worker 调度过快导致数据被重复同步的问题 #7658 @GMHDBJD
    * 修复上游数据库使用正则匹配授权时 DM 前置检查不通过的错误 #7645 @lance6716

  - TiDB Lightning

    * 修复 TiDB Lightning 导入巨大数据源文件时的内存泄漏问题 #39331 @dsdashun
    * 修复 TiDB Lightning 在并行导入冲突检测时无法正确检测的问题 #39476 @dsdashun

16.7.4.6 贡献者

感谢来自 TiDB 社区的贡献者们：

- e1ijah1
- guoxiangCN（首次贡献者）
- jiayang-zheng
- jiyfhust
- mikechengwei
- pingandb
- sashashura
- sourcelliu
- wxbty

16.8 v6.4

16.8.1 TiDB 6.4.0 Release Notes

发版日期：2022 年 11 月 17 日

TiDB 版本：6.4.0-DMR

> **注意：**
>
> TiDB 6.4.0-DMR 的用户文档已归档。如无特殊需求，建议使用 TiDB 数据库的最新 LTS 版本。

试用链接：快速体验 | 下载离线包

在 6.4.0-DMR 版本中，你可以获得以下关键特性：

- 支持通过`FLASHBACK CLUSTER TO TIMESTAMP`命令将集群快速回退到特定的时间点 (实验特性)。
- 支持对 TiDB 实例的全局内存使用进行追踪（实验特性)。

- TiDB 分区表兼容 LINEAR HASH 分区语法。
- 支持高性能、全局单调递增的AUTO_INCREMENT 列属性（实验特性）。
- 支持对JSON 类型中的 Array 数据做范围选择。
- 实现磁盘故障、I/O 无响应等极端情况下的故障恢复加速。
- 新增动态规划算法来决定表的连接顺序。
- 引入新的优化器提示 NO_DECORRELATE 来控制关联优化的解除。
- 集群诊断功能 GA。
- TiFlash 静态加密支持国密算法 SM4。
- 支持通过 SQL 语句立即对指定分区的 TiFlash 副本进行物理数据整理 (Compaction)。
- 支持基于 AWS EBS snapshot 的集群备份和恢复。
- 支持在分库分表合并迁移场景中标记下游表中的数据来自上游哪个分库、分表和数据源。

16.8.1.1 新功能

16.8.1.1.1 SQL

- 支持通过 SQL 语句立即对指定分区的 TiFlash 副本进行物理数据整理 (Compaction) #5315 @hehechen

  TiDB v6.2.0 发布了针对全表的 TiFlash 副本立即进行物理数据整理 (Compaction) 的功能，支持用户自行选择合适的时机，手动执行 SQL 语句立即对 TiFlash 中的物理数据进行整理，从而减少存储空间占用，并提升查询性能。v6.4.0 细化了 TiFlash 副本数据整理的粒度，支持对表中指定分区的 TiFlash 副本立即进行数据整理。

  通过 SQL 语句 `ALTER TABLE table_name COMPACT [PARTITION PartitionNameList] [engine_type ↪ REPLICA]`，你可以立即对指定分区的 TiFlash 副本进行数据整理。

  更多信息，请参考用户文档。

- 支持通过 `FLASHBACK CLUSTER TO TIMESTAMP` 命令将集群快速回退到特定的时间点（实验特性）#37197 #13303 @Defined2014 @bb7133 @JmPotato @Connor1996 @HuSharp @CalvinNeo

  `FLASHBACK CLUSTER TO TIMESTAMP` 支持在 Garbage Collection (GC) life time 内快速回退整个集群到指定的时间点。使用该特性可以快速撤消 DML 误操作。例如，在误执行了没有 `WHERE` 子句的 `DELETE` 后，使用 `FLASHBACK CLUSTER TO TIMESTAMP` 能够在几分钟内将集群数据恢复到指定的时间点。该特性不依赖于数据库备份，并支持在时间线上多次回退以确定特定数据更改发生的时间。需要注意的是，`FLASHBACK CLUSTER TO TIMESTAMP` 不能替代数据库备份。

  在执行 `FLASHBACK CLUSTER TO TIMESTAMP` 之前，需要暂停 PITR 和 TiCDC 等工具上运行的同步任务，待 FLASHBACK 执行完成后再启动，否则会造成同步失败等问题。

  更多信息，请参考用户文档。

- 支持通过 `FLASHBACK DATABASE` 命令来恢复被删除的数据库 #20463 @erwadba

  `FLASHBACK DATABASE` 支持在 Garbage Collection (GC) life time 时间内恢复被 `DROP` 删除的数据库以及数据。该特性不依赖任何外部工具，可以轻松快速地通过 SQL 语句进行数据和元信息的恢复。

  更多信息，请参考用户文档。

16.8.1.1.2 安全

- TiFlash 静态加密支持国密算法 SM4 #5953 @lidezhu

  TiFlash 的静态加密新增 SM4 算法，你可以将配置文件 `tiflash-learner.toml` 中的 `data-encryption-method` 参数的值设置为 `sm4-ctr`，以启用基于国密算法 SM4 的静态加密能力。

  更多信息，请参考用户文档。

16.8.1.1.3 可观测性

- 集群诊断功能 GA #1438 @Hawkson-jee

  集群诊断功能是在指定的时间范围内，对集群可能存在的问题进行诊断，并将诊断结果和一些集群相关的负载监控信息汇总成一个诊断报告。诊断报告是网页形式，通过浏览器保存后可离线浏览和传阅。

  你可以通过该报告快速了解集群内的基本诊断信息，包括负载、组件、耗时和配置信息。若集群存在一些常见问题，在诊断信息部分可以了解 TiDB 内置自动诊断的结果。

16.8.1.1.4 性能

- 引入 Coprocessor Task 并发度自适应机制 #37724 @you06

  随着 Coprocessor Task 任务数增加，TiDB 将结合 TiKV 处理速度自动增加任务并发度（调整`tidb_distsql_scan_concurrency`），减少 Coprocessor Task 任务排队，降低延迟。

- 新增动态规划算法来决定表的连接顺序 #37825 @winoros

  在之前的版本中，TiDB 采用贪心算法来决定表的连接顺序。在 v6.4.0 中，优化器引入了动态规划算法。相比贪心算法，动态规划算法会枚举更多可能的连接顺序，进而有机会发现更好的执行计划，提升部分场景下 SQL 执行效率。

  由于动态规划算法的枚举过程可能消耗更多的时间，目前 Join Reorder 算法由变量`tidb_opt_join_reorder_threshold`控制，当参与 Join Reorder 的节点个数大于该阈值时选择贪心算法，反之选择动态规划算法。

  更多信息，请参考用户文档。

- 前缀索引支持对空值进行过滤 #21145 @xuyifangreeneyes

  该特性是对前缀索引使用上的优化。当表中某列存在前缀索引，那么 SQL 语句中对该列的 `IS NULL` 或 `IS NOT NULL` 条件可以直接利用前缀进行过滤，避免了这种情况下的回表，提升了 SQL 语句的执行性能。

  更多信息，请参考用户文档。

- 增强 TiDB Chunk 复用机制 #38606 @keeplearning20221

  在之前的版本中，TiDB 只在 `writechunk` 函数中复用 Chunk。TiDB v6.4.0 扩展 Chunk 复用机制到 Executor 的算子中，通过复用 Chunk 减少 TiDB 申请释放内存频率，进而提升部分场景下的 SQL 查询执行效率。你可以通过系统变量`tidb_enable_reuse_chunk` 来控制是否启用 Chunk 对象复用，默认为开启。

- 引入新的优化器提示 `NO_DECORRELATE` 来控制关联优化的解除 #37789 @time-and-fate

  默认情况下，TiDB 总是会尝试重写关联子查询以解除关联，这通常会提高执行效率。但是在一部分场景下，解除关联反而会降低执行效率。TiDB 在 v6.4.0 版本中引入了 hint `NO_DECORRELATE`，用来提示优化器不要对指定的查询块解除关联，以提升部分场景下的查询性能。

  更多信息，请参考用户文档。

- 提升了分区表统计信息收集的性能 #37977 @Yisaer

  在 v6.4.0 版本中，TiDB 优化了分区表统计信息的收集策略。你可以通过系统变量 tidb_auto_analyze_partition_batch_size 定义并发度，用并行的方式同时收集多个分区的统计信息，从而加快统计信息收集的速度，减少 analyze 所需的时间。

16.8.1.1.5 稳定性

- 磁盘故障、I/O 无响应等极端情况下的故障恢复加速 #13648 @LykxSassinator

数据库的可用性是企业用户最为关注的指标之一，但是在复杂的硬件环境下，如何快速检测故障并恢复一直是数据库面临的挑战之一。TiDB v6.4.0 全面优化了 TiKV 节点的状态检测机制。即使在磁盘故障或 I/O 无响应等极端情况下，TiDB 依然可以快速上报节点状态，同时搭配主动唤醒机制，提前发起 Leader 选举，加速集群自愈。通过这次优化，TiDB 在磁盘故障场景下，集群恢复时间可以缩短 50% 左右。

- TiDB 全局内存控制（实验特性）#37816 @wshwsh12

  v6.4.0 引入了全局内存控制，对 TiDB 实例的全局内存使用进行追踪。你可以通过系统变量 tidb_server_memory_limit 设置全局内存的使用上限。当内存使用量接近预设的上限时，TiDB 会尝试对内存进行回收，释放更多的可用内存；当内存使用量超出预设的上限时，TiDB 会识别出当前内存使用量最大的 SQL 操作，并取消这个操作，避免因为内存使用过度而产生的系统性问题。

  当 TiDB 实例的内存消耗存在潜在风险时，TiDB 会预先收集诊断信息并写入指定目录，方便对问题的诊断。同时，TiDB 提供了系统表视图 INFORMATION_SCHEMA.MEMORY_USAGE 和 INFORMATION_SCHEMA.MEMORY_USAGE_OPS_HISTORY 用来展示内存使用情况及历史操作，帮助用户清晰了解内存使用状况。

  全局内存控制是 TiDB 内存管理的重要一步，对实例采用全局视角，引入系统性方法对内存用量进行管理，这可以极大提升数据库的稳定性，提高服务的可用性，支持 TiDB 在更多重要场景平稳运行。

  更多信息，请参考用户文档。

- 控制优化器在构造范围时的内存占用 #37176 @xuyifangreeneyes

  v6.4.0 引入了系统变量 tidb_opt_range_max_size 来限制优化器在构造范围时消耗的内存上限。当内存使用超出这个限制，则放弃构造精确的范围，转而构建更粗粒度的范围，以此降低内存消耗。当 SQL 语句中的 IN 条件较多时，这个优化可以显著降低编译时的内存使用量，保证系统的稳定性。

  更多信息，请参考用户文档。

- 支持统计信息的同步加载（GA）#37434 @chrysan

  TiDB v6.4.0 起，正式开启了统计信息同步加载的特性（默认开启），支持在执行当前 SQL 语句时将直方图、TopN、CMSketch 等占用空间较大的统计信息同步加载到内存，提高优化该 SQL 语句时统计信息的完整性。

  更多信息，请参考用户文档。

- 降低批量写入请求对轻量级事务写入的响应时间的影响 #13313 @glorv

  定时批量 DML 任务存在于一部分系统的业务逻辑中。在此场景下，处理这些批量写入任务会增加在线交易的时延。在 v6.3.0 中，TiKV 对混合负载场景下读请求的优先级进行了优化，你可以通过 readpool.unified.auto-adjust-pool-size 配置项开启 TiKV 对统一处理读请求的线程池 (UnifyReadPool) 大小的自动调整。在 v6.4.0 中，TiKV 对写入请求也进行了动态识别和优先级调整，控制 Apply 线程每一轮处理单个状态机写入的最大数据量，从而降低批量写入对交易事务写入的响应时间的影响。

16.8.1.1.6 易用性

- TiKV API V2 成为正式功能 #11745 @pingyu

  在 v6.1.0 之前，TiKV 的 RawKV 接口仅存储客户端传入的原始数据，因此只提供基本的 Key-Value 读写能力。此外，由于编码方式不同、数据范围没有隔离等，同一个 TiKV 集群中，TiDB、事务 KV、RawKV 无法同时使用，因此对于不同使用方式并存的场景，必须部署多个集群，增加了机器和部署成本。

  TiKV API V2 提供了新的存储格式，功能亮点如下：

  - RawKV 数据以 MVCC 方式存储，记录数据的变更时间戳，并在此基础上提供 Change Data Capture 能力（实验特性，见 TiKV-CDC）。
  - 数据根据使用方式划分范围，支持单一集群 TiDB、事务 KV、RawKV 应用共存。
  - 预留 Key Space 字段，为多租户等特性提供支持。

你可以通过在 TiKV 的 `[storage]` 配置中设置 `api-version = 2` 来启用 TiKV API V2。

更多信息，请参考用户文档。

- 优化 TiFlash 数据同步进度的准确性 #4902 @hehechen

  TiDB 的 `INFORMATION_SCHEMA.TIFLASH_REPLICA` 表中的 `PROGRESS` 字段表示 TiFlash 副本与 TiKV 中对应表数据的同步进度。在之前的版本中，`PROCESS` 字段只显示 TiFlash 副本创建过程中的数据同步进度。在 TiFlash 副本创建完后，当在 TiKV 相应的表中导入新的数据时，该值不会更新数据的同步进度。

  v6.4.0 版本改进了 TiFlash 副本数据同步进度更新机制。在创建 TiFlash 副本后，进行数据导入等操作，TiFlash 副本需要和 TiKV 数据进行同步时，`INFORMATION_SCHEMA.TIFLASH_REPLICA` 表中的 `PROGRESS` 值将会更新，显示实际的数据同步进度。通过此优化，你可以方便地查看 TiFlash 数据同步的实际进度。

  更多信息，请参考用户文档。

16.8.1.1.7 MySQL 兼容性

- TiDB 分区表兼容 Linear Hash 分区语法 #38450 @mjonss

  TiDB 现有的分区方式支持 Hash、Range、List 分区。TiDB v6.4.0 增加了对 MySQL LINEAR HASH 分区语法的兼容。

  在 TiDB 上，你可以直接执行原有的 MySQL Linear Hash 分区的 DDL 语句，TiDB 将创建一个常规的非线性 Hash 分区表（注意 TiDB 内部实际不存在 LINEAR HASH 分区）。你也可以直接执行原有的 MySQL LINEAR HASH 分区的 DML 语句，TiDB 将正常返回对应的 TiDB Hash 分区的查询结果。此功能保证了 TiDB 对 MySQL LINEAR HASH 分区的语法兼容，方便基于 MySQL 的应用无缝迁移到 TiDB。

当分区数是 2 的幂时，TiDB Hash 分区表中行的分布情况与 MySQL Linear Hash 分区表相同，但当分区数不是 2 的幂时，TiDB Hash 分区表中行的分布情况与 MySQL Linear Hash 分区表会有所区别。

更多信息，请参考[用户文档](#tidb-对-linear-hash-分区的处理)。

- 支持高性能、全局单调递增的 AUTO_INCREMENT 列属性（实验特性）#38442 @tiancaiamao

  TiDB v6.4.0 引入了 AUTO_INCREMENT 的 MySQL 兼容模式，通过中心化分配自增 ID，实现了自增 ID 在所有 TiDB 实例上单调递增。使用该特性能够更容易地实现查询结果按自增 ID 排序。要使用 MySQL 兼容模式，你需要在建表时将 AUTO_ID_CACHE 设置为 1。

```
CREATE TABLE t(a int AUTO_INCREMENT key) AUTO_ID_CACHE 1;
```

更多信息，请参考用户文档。

- 支持对 JSON 类型中的 Array 数据做范围选择 #13644 @YangKeao

  从 v6.4.0 起，TiDB 支持 MySQL 兼容的范围选择语法。

  - 通过关键字 `to`，你可以指定元素起始和结束的位置，并选择 Array 中连续范围的元素，起始位置记为 0。例如，使用 `$[0 to 2]` 可以选择 Array 中的前三个元素。
  - 通过关键字 `last`，你可以指定 Array 中最后一个元素的位置，实现从右到左的位置设定。例如，使用 `$[last-2 to last]` 可以选择 Array 中的最后三个元素。

  该特性简化了 SQL 的编写过程，进一步提升了 JSON 类型的兼容能力，降低了 MySQL 应用向 TiDB 迁移的难度。

- 支持对数据库用户增加额外说明 #38172 @CbcWestwolf

  在 TiDB v6.4.0 中，你可以通过`CREATE USER` 或`ALTER USER` 语句为数据库用户添加额外的说明信息。TiDB 提供了两种说明格式，你可以通过 `COMMENT` 添加一段文本注释，也可以通过 `ATTRIBUTE` 添加一组 JSON 格式的结构化属性。

  此外，TiDB v6.4.0 新增了`USER_ATTRIBUTES` 表。你可以在该表中查看用户的注释和属性信息。

```
CREATE USER 'newuser1'@'%' COMMENT 'This user is created only for test';
CREATE USER 'newuser2'@'%' ATTRIBUTE '{"email": "user@pingcap.com"}';
SELECT * FROM INFORMATION_SCHAME.USER_ATTRIBUTES;
```

```
+-----------+------+-----------------------------------------------------+
| USER      | HOST | ATTRIBUTE                                           |
+-----------+------+-----------------------------------------------------+
| newuser1  | %    | {"comment": "This user is created only for test"} |
| newuser1  | %    | {"email": "user@pingcap.com"}                       |
+-----------+------+-----------------------------------------------------+
2 rows in set (0.00 sec)
```

这个特性提升了 TiDB 对 MySQL 的语法的兼容性，使得 TiDB 更容易融入 MySQL 生态的工具或平台。

16.8.1.1.8 备份和恢复

- 基于 AWS EBS snapshot 的集群备份和恢复 #33849 @fengou1

  如果你的 TiDB 集群部署在 EKS 上，使用了 AWS EBS 卷，并且对数据备份有以下要求，可考虑使用 TiDB Operator 将 TiDB 集群数据以卷快照以及元数据的方式备份至 Amazon S3：

  - 备份的影响降到最小，如备份对 QPS 和事务耗时影响小于 5%，不占用集群 CPU 以及内存。
  - 快速备份和恢复，比如 1 小时内完成备份，2 小时内完成恢复。

更多信息，请参考用户文档。

16.8.1.1.9 数据迁移

- 支持将上游数据源信息以扩展列形式写入下游合表 #37797 @lichunzhu

  在上游分库分表合并到 TiDB 的场景，你可以在目标表中手动额外增加几个字段（扩展列），并在配置 DM 任务时，对这几个扩展列赋值。例如，当赋予上游分库分表的名称时，通过 DM 写入到下游的记录会包含上游分库分表的名称。在一些数据异常的场景，你可以通过该功能快速定位目标表的问题数据源信息，如该数据来自上游哪个分库，哪个分表。

  更多信息，请参考提取分库分表数据源信息写入合表。

- 优化 DM 的前置检查项，将部分必须通过项改为非必须通过项 #7333 @lichunzhu

  为了使数据迁移任务顺利进行，DM 在启动迁移任务时会自动触发任务前置检查，并返回检查结果。只有当前置检查通过后，DM 才开始执行迁移任务。

  在 v6.4.0，DM 将如下三个检查项由必须通过项改为非必须通过项，提升了前置检查通过率：

  - 检查字符集是否存在兼容性差异
  - 检查上游表中是否存在主键或唯一键约束
  - 数据库主从配置，上游数据库必须设置数据库 ID `server_id`

- 增量迁移任务支持 binlog position 和 GTID 作为选配参数 #7393 @GMHDBJD

  v6.4.0 之前，只配置增量迁移任务时，需要传入 binlog position 或者 GTID 才能启动任务，配置复杂，用户理解成本高。自 v6.4.0 起，如果只需要执行增量迁移任务，则可以不指定 binlog position 或者 GTID 的参数取值，DM 将默认按任务的启动时间从上游获取该时间之后的 binlog file，并将这些增量数据迁移到下游，降低了使用时的理解成本和配置复杂度。

  更多信息，请参考DM 任务完整配置文件介绍。

- DM 任务增加一些状态信息的展示 #7343 @okJiang

  在 v6.4.0，DM 数据迁移任务新增了一些性能指标和进度指标，方便用户更直观地了解迁移性能和进度，同时为问题排查提供参考信息：

  - 增加了 DM 任务当前数据导出、数据导入的性能指标，单位 bytes/s。
  - 将当前 DM 写入目标库的性能指标命名从 TPS 改为 RPS (rows/second)。
  - 新增了 DM 全量任务数据导出的进度展示。

关于这些指标的详细介绍，请参考TiDB Data Migration 查询状态。

16.8.1.1.10 数据共享与订阅

- TiCDC 支持同步数据到 `3.2.0` 版本的 Kafka #7191 @3AceShowHand

  TiCDC 下游支持的 Kafka 最高版本从 `3.1.0` 变为 `3.2.0`。你可以通过 TiCDC 将数据同步到不高于 `3.2.0` 版本的 Kafka。

16.8.1.2 兼容性变更

16.8.1.2.1 系统变量

变量名	修改类型	描述
tidb_constraint_check_in_place_pessimistic	修改	该变量用于控制悲观事务中唯一约束检查的时间点。v6.4.0 去掉了它的 GLOBAL 作用域并支持通过配置项 pessimistic-txn.constraint-check-in-place-pessimistic 控制它的默认值。
tidb_ddl_flashback_concurrency	修改	该变量从 v6.4.0 开始生效，用来控制 FLASHBACK CLUSTER TO TIMESTAMP 的并发数。默认值为 64。

变量名	修改类型	描述
tidb_enable_clustered_index ↪	修改	该变量默认值从 INT_ONLY ↪ 修改为 ON，表示表的主键默认使用聚簇索引。
tidb_enable_paging ↪	修改	该变量默认值 OFF 修改为 ON，表示默认使用分页 (paging) 方式发送 Coprocessor 请求。
tidb_enable_prepared_plan_cache ↪	修改	该变量用来控制是否开启Prepared Plan Cache。v6.4.0 新增了 SESSION 作用域。

变量名	修改类型	描述
tidb_memory_usage_alarm_ratio ↪	修改	该变量用于设置触发 tidb-server 内存告警的内存使用比率，默认值从 0.8 修改为 0.7。
tidb_opt_agg_push_down ↪	修改	该变量用来设置优化器是否执行聚合函数下推到 Join，Projection 和 UnionAll 之前的优化操作。v6.4.0 新增了 GLOBAL 的作用域。

变量名	修改类型	描述
tidb_prepared_plan_cache_size	修改	该变量用来控制单个 session 的 Prepared Plan Cache 最多能够缓存的计划数量。v6.4.0 新增了 SESSION 作用域。
tidb_stats_load_sync_wait	修改	该变量默认值从 0 修改为 100，代表 SQL 执行同步加载完整统计信息默认等待 100 毫秒后会超时。

变量名	修改类型	描述
tidb_stats_load_pseudo_timeout ↪	修改	该变量默认值从 OFF 修改为 ON，代表统计信息同步加载超时后，SQL 会退回使用 pseudo 的统计信息。
last_sql_use_alloc ↪	新增	该变量是一个只读变量，用来显示上一个语句是否使用了缓存的 Chunk 对象 (Chunk allocation)。默认值为 OFF。

变量名	修改类型	描述
tidb_auto_analyze_partition_batch_size ↪	新增	该变量用于设置 TiDB 自动 analyze 分区表（即自动收集分区表上的统计信息）时，每次同时 analyze 分区的个数。默认值为 1。
tidb_enable_external_ts_read ↪	新增	该变量用于控制 TiDB 是否会读取tidb_external_ts ↪ 指定的时间戳前的历史数据。默认值为 OFF。
tidb_enable_gogc_tuner ↪	新增	该变量来用控制是否开启 GOGC Tuner，默认为 ON。

变量名	修改类型	描述
tidb_enable_reuse_chunk	新增	该变量用于控制 TiDB 是否启用 Chunk 对象缓存，默认为 ON，代表 TiDB 优先使用缓存中的 Chunk 对象，缓存中找不到申请的对象时才会从系统内存中申请。如果为 OFF，则直接从系统内存中申请 Chunk 对象。
tidb_enable_prepared_plan_cache_memory_monitor	新增	该变量用来控制是否统计 Prepared Plan Cache 中所缓存的执行计划占用的内存，默认为 ON。

变量名	修改类型	描述
tidb_external_ts ↪	新增	默认值为 0。当tidb_enable_external_ts_read ↪ 设置为 ON 时，TiDB 会依据该变量指定的时间戳读取历史数据。
tidb_gogc_tuner_threshold ↪	新增	该变量用来控制 GOGC Tuner 自动调节的最大内存阈值，超过阈值后 GOGC Tuner 会停止工作。默认值为 0.6。

变量名	修改类型	描述
tidb_memory_usage_alarm_keep_record_num ↪	新增	当 tidb-server 内存占用超过内存报警阈值并触发报警时，TiDB 默认只保留最近 5 次报警时所生成的状态文件。通过该变量可以调整该次数。
tidb_opt_prefix_index_single_scan ↪	新增	该变量用于控制 TiDB 优化器是否将某些过滤条件下推到前缀索引，尽量避免不必要的回表，从而提高查询性能。默认为 ON。

变量名	修改类型	描述
tidb_opt_range_max_size ↪	新增	该变量用于指定优化器构造扫描范围的内存用量上限。默认值为 67108864 ↪ （即 64 MiB ）。
tidb_server_memory_limit ↪	新增	该变量用于指定 TiDB 实例的内存限制（实验特性）。默认值为 0，表示不设内存限制。
tidb_server_memory_limit_gc_trigger ↪	新增	该变量用于控制 TiDB 尝试触发 GC 的阈值（实验特性）。默认值为 70%。

变量名	修改类型	描述
tidb_server_memory_limit_sess_min_size ↪	新增	开启内存限制后，TiDB 会终止当前实例上内存用量最高的 SQL 语句。本变量指定此情况下 SQL 语句被终止的最小内存用量（实验特性），默认值为 134217728 ↪ （即 128 MiB）。

16.8.1.2.2 配置文件参数

配置文件	配置项	修改类型	描述
TiDB	tidb_memory_usage_alarm_ratio ↪	废弃	该配置不再生效。
TiDB	memory-usage-alarm-ratio	废弃	该配置项被系统变量 tidb_memory_usage_alarm_ratio ↪ 所取代。如果在升级前设置过该配置项，升级后原配置将不再生效。

配置文件	配置项	修改类型	描述
TiDB	pessimistic-txn.constraint-check-in-place-pessimistic	新增	用于控制系统变量tidb_constraint_check_in_place_pessimistic 的默认值，默认值为 true。
TiDB	tidb-max-reuse-chunk	新增	用于控制每个连接最多缓存的 Chunk 对象数，默认值为 64。
TiDB	tidb-max-reuse-column	新增	用于控制每个连接最多缓存的 column 对象数，默认值为 256。
TiKV	cdc.raw-min-ts-outlier-threshold	废弃	该配置不再生效。
TiKV	causal-ts.alloc-ahead-buffer	新增	预分配给 TSO 的缓存大小（以时长计算），默认值为 3s。
TiKV	causal-ts.renew-batch-max-size	新增	单次时间戳请求的最大数量，默认值为 8192。

配置文件	配置项	修改类型	描述
TiKV	raftstore ↪ .apply ↪ -yield ↪ -write ↪ -size	新增	Apply 线程每一轮处理单个状态机写入的最大数据量，默认值为 32KiB。这是个软限制。
PD	tso- ↪ update ↪ - ↪ physical ↪ - ↪ interval ↪	新增	这个配置项从 v6.4.0 开始生效，用来控制 TSO 物理时钟更新周期，默认值为 50ms。
TiFlash	data- ↪ encryption ↪ - ↪ method	修改	扩展可选值范围：增加 sm4-ctr。设置为 sm4-ctr 时，数据将采用国密算法 SM4 加密后进行存储。

配置文件	配置项	修改类型	描述
DM	routes. ↪ route- ↪ rule ↪ -1. ↪ extract ↪ -table	新增	可选配置。用于提取分库分表场景中分表的源信息，提取的信息写入下游合表，用于标识数据来源。如果配置该项，需要提前在下游手动创建合表。
DM	routes. ↪ route- ↪ rule ↪ -1. ↪ extract ↪ - ↪ schema	新增	可选配置。用于提取分库分表场景中分库的源信息，提取的信息写入下游合表，用于标识数据来源。如果配置该项，需要提前在下游手动创建合表。

配置文件	配置项	修改类型	描述
DM	routes. ↪ route- ↪ rule ↪ -1. ↪ extract ↪ - ↪ source	新增	可选配置。用于提取分库分表场景中的源信息，提取的信息写入下游合表，用于标识数据来源。如果配置该项，需要提前在下游手动创建合表。
TiCDC	transaction ↪ - ↪ atomicity ↪	修改	默认值由 table 改为 none。该修改可降低同步延迟，减少系统出现 OOM 的风险。同时，修改默认值后，系统只拆分少量的事务（即超过 1024 行的事务），而不是拆分所有事务。

16.8.1.2.3 其他

- 从 v6.4.0 开始，mysql.user 表新增 User_attributes 和 Token_issuer 两个字段。如果从 v6.4.0 之前版本的备份数据恢复 mysql schema 下的系统表到 v6.4.0 集群，BR 将返回 mysql.user 表的 column count mismatch 错误。如果你未选择恢复 mysql schema 下的系统表，则不会报错。
- 针对命名规则符合 Dumpling 输出文件格式但后缀名并非 gzip 压缩格式的文件（例如 test-schema-create ↪ .sql.origin 和 test.table-schema.sql.origin），Lightning 的处理方式发生了变化。在 v6.4.0 之前的

版本中，如果待导入的文件中包含这类文件，Lightning 将跳过对这类文件的导入。从 v6.4.0 起，Lightning 将认为这些文件使用了不支持的压缩格式，导致导入失败。

- 从 v6.4.0 开始，TiCDC 使用 Syncpoint 功能需要同步任务拥有下游集群的 SYSTEM_VARIABLES_ADMIN 或者 SUPER 权限。

16.8.1.3 改进提升

- TiDB
  - 允许修改 noop 系统变量 `lc_messages` #38231 @djshow832
  - 允许 AUTO_RANDOM 列作为聚簇复合索引中的第一列 #38572 @tangenta
  - 内部事务重试使用悲观模式避免重试失败，降低耗时 #38136 @jackysp
- TiKV
  - 新增 `apply-yield-write-size` 配置项，以限制 Apply 线程每一轮处理单个状态机写入的数据大小，缓解 Raftstore 线程在 Apply 写入过大时的阻塞现象 #13313 @glorv
  - 在 Region 的 Leader 迁移前增加缓存预热阶段，缓解 Leader 迁移时造成的 QPS 剧烈抖动 #13060 @cosven
  - 支持将 `json_contains` 算子下推至 Coprocessor #13592 @lizhenhuan
  - 新增 `CausalTsProvider` 的异步实现，提升某些场景下刷盘的性能 #13428 @zeminzhou
- PD
  - 热点均衡调度器 v2 版本算法成为正式功能，在特定场景下 v2 版本算法可以在配置的两个维度均取得更好的均衡效果，并减少无效调度 #5021 @HundunDM
  - 改进 Operator step 超时机制，防止过早超时 #5596 @bufferflies
  - 优化调度器在大集群下的性能 #5473 @bufferflies
  - 支持使用非 PD 提供的外部时间戳 #5637 @lhy1024
- TiFlash
  - 重构了 TiFlash MPP 的错误处理流程，进一步提升了 MPP 的稳定性 #5095 @windtalker
  - 优化了 TiFlash 计算过程中的排序操作，以及对 Join 和 Aggregation 的 Key 的处理 #5294 @solotzg
  - 优化了 TiFlash 编解码的内存使用，去除了冗余传输列以提升 Join 性能 #6157 @yibin87
- Tools
  - TiDB Dashboard
    - 支持在 Monitoring 页面展示 TiFlash 相关指标，并且优化了该页面指标的展示方式 #1440 @YiniXu9506
    - 支持在 Slow Queries 列表和 SQL Statements 列表展示结果行数 #1443 @baurine
    - 当集群没有 Alertmanager 时不显示报错信息 #1444 @baurine
  - Backup & Restore (BR)
    - 改进加载元数据的机制，仅在需要时才将元数据加载到内存中，显著减少 PITR 过程中的内存压力 #38404 @YuJuncen
  - TiCDC

    * 支持同步 Exchange Partition 的 DDL 语句 #639 @asddongmen
    * 提升 MQ sink 模块非攒批发送的性能 #7353 @hi-rustin
    * 提升单表大量 Region 场景下 TiCDC puller 的性能 #7078 #7281 @sdojjy
    * 支持在 Syncpoint 功能开启时在下游 TiDB 集群使用 `tidb_enable_external_ts_read` 来读取历史数据 #7419 @asddongmen
    * 默认情况下关闭 safeMode 并开启大事务拆分功能，提升同步的稳定性 #7505 @asddongmen

  - TiDB Data Migration (DM)

    * 移除 dmctl 中无用的 `operate-source update` 指令 #7246 @buchuitoudegou
    * 解决了 TiDB 不兼容上游数据库的建表 SQL 导致 DM 全量迁移报错的问题，当上游的建表 SQL TiDB 不兼容时，用户可以提前在 TiDB 手动创建好目标表，让全量迁移任务继续运行 #37984 @lance6716

  - TiDB Lightning

    * 优化文件扫描逻辑，提升 Schema 类型文件的扫描速度 #38598 @dsdashun

16.8.1.4 错误修复

- TiDB

  - 修复新建索引之后有可能导致数据索引不一致的问题 #38165 @tangenta
  - 修复 `INFORMATION_SCHEMA.TIKV_REGION_STATUS` 表的权限问题 #38407 @CbcWestwolf
  - 修复 `mysql.tables_priv` 表中 `grantor` 字段缺失的问题 #38293 @CbcWestwolf
  - 修复公共表表达式在 join 时可能得到错误结果的问题 #38170 @wjhuang2016
  - 修复公共表表达式在 union 时可能得到错误结果的问题 #37928 @YangKeao
  - 修复监控面板 transaction region num 信息不准确的问题 #38139 @jackysp
  - 修复`tidb_constraint_check_in_place_pessimistic` 可能影响内部事务问题，修改该变量作用域为 SESSION #38766 @ekexium
  - 修复条件在某些场景下被错误下推至 projection 的问题 #35623 @Reminiscent
  - 修复 AND 和 OR 条件的 `isNullRejected` 检查错误导致查询结果错误的问题 #38304 @Yisaer
  - 修复外连接消除时没有考虑 `GROUP_CONCAT` 内部的 `ORDER BY` 导致查询出错的问题 #18216 @winoros
  - 修复错误下推的条件被 Join Reorder 丢弃后导致查询结果错误的问题 #38736 @winoros

- TiKV

  - 修复 Gitpod 环境中存在多个 `cgroup` 和 `mountinfo` 时 TiDB 启动异常的问题 #13660 @tabokie
  - 修复 TiKV 监控 `tikv_gc_compaction_filtered` 表达式错误的问题 #13537 @Defined2014
  - 修复 `delete_files_in_range` 存在异常导致的性能问题 #13534 @tabokie
  - 修复获取 Snapshot 时 Lease 过期引发的异常竞争问题 #13553 @SpadeA-Tang
  - 修复第一次 FLASHBACK 失败时存在异常的问题 #13672 #13704 #13723 @HuSharp

- PD

  - 修复 Stream 超时问题，提高 Leader 切换的速度 #5207 @CabinfeverB

- TiFlash

  - 修复由于 PageStorage GC 未能正确清除 Page 删除标记导致 WAL 文件过大从而导致 TiFlash OOM 的问题 #6163 @JaySon-Huang

- Tools

  - TiDB Dashboard

    * 修复查询某些复杂 SQL 语句的执行计划时 TiDB OOM 的问题 #1386 @baurine
    * 修复 NgMonitoring 丢失对 PD 节点的连接时可能导致 Top SQL 开关无效的问题 #164 @zhongzc

  - Backup & Restore (BR)

    * 修复恢复过程中由于 PD leader 切换导致恢复失败的问题 #36910 @MoCuishle28
    * 修复了无法暂停日志备份任务的问题 #38250 @joccau
    * 修复 BR 删除日志备份数据时，会删除不应被删除的数据的问题 #38939 @Leavrth
    * 修复 BR 首次删除存储在 Azure Blob Storage 或 Google Cloud Storage 的日志备份数据时执行失败的问题 #38229 @Leavrth

  - TiCDC

    * 修复 `changefeed query` 的输出中 `sasl-password` 显示为明文的问题 #7182 @dveeden
    * 修复在一个 etcd 事务中提交太多数据导致 TiCDC 服务不可用问题 #7131 @asddongmen
    * 修复 redo log 文件可能被错误删除的问题 #6413 @asddongmen
    * 修复 Kafka Sink V2 协议在同步宽表时性能回退的问题 #7344 @hi-rustin
    * 修复 checkpoint ts 可能被提前推进的问题 #7274 @hi-rustin
    * 修复 mounter 模块的日志级别设置不当导致 log 打印太多的问题 #7235 @hi-rustin
    * 修复一个 TiCDC 集群可能存在两个 owner 的问题 #4051 @asddongmen

  - TiDB Data Migration (DM)

    * 修复 DM WebUI 产生错误 `allow-list` 参数的问题 #7096 @zoubingwu
    * 修复 DM-worker 在启动、停止时有一定概率触发 data race 的问题 #6401 @liumengya94
    * 修复当同步 `UPDATE`、`DELETE` 语句且下游行数据不存在时，DM 静默忽略的问题 #6383 @GMHDBJD
    * 修复运行 `query-status` 命令后未显示 `secondsBehindMaster` 字段的问题 #7189 @GMHDBJD
    * 修复更新 Checkpoint 时可能触发大事务的问题 #5010 @lance6716
    * 修复在全量任务模式下，任务进入 sync 阶段且立刻失败时，DM 可能丢失上游表结构信息的问题 #7159 @lance6716
    * 修复开启一致性校验时可能触发死锁的问题 #7241 @buchuitoudegou
    * 修复任务预检查对 `INFORMATION_SCHEMA` 表需要 `SELECT` 权限的问题 #7317 @lance6716
    * 修复空的 TLS 配置导致报错的问题 #7384 @liumengya94

  - TiDB Lightning

    * 修复当导入 Apache Parquet 格式的数据时，如果目标表存在 binary 编码格式的字符串类型列，导入性能下降的问题 #38351 @dsdashun

  - TiDB Dumpling

    * 修复导出大量表时可能导致超时的问题 #36549 @lance6716
    * 修复加锁模式下，上游不存在对应表时导致加锁报错的问题 #38683 @lance6716

16.8.1.5 贡献者

感谢来自 TiDB 社区的贡献者们：

- 645775992
- An-DJ
- AndrewDi
- erwadba
- fuzhe1989
- goldwind-ting（首次贡献者）
- h3n4l
- igxlin（首次贡献者）
- ihcsim
- JigaoLuo
- morgo
- Ranxy
- shenqidebaozi（首次贡献者）
- taofengliu（首次贡献者）
- TszKitLo40
- wxbty（首次贡献者）
- zgcbj

16.9 v6.3

16.9.1 TiDB 6.3.0 Release Notes

发版日期：2022 年 9 月 30 日

TiDB 版本：6.3.0-DMR

> **注意：**
>
> TiDB 6.3.0-DMR 的用户文档已归档。如无特殊需求，建议使用 TiDB 数据库的最新 LTS 版本。

试用链接：快速体验 | 下载离线包

在 6.3.0-DMR 版本中，你可以获得以下关键特性：

- TiKV 静态加密支持国密算法 SM4。
- TiDB 支持基于国密算法 SM3 插件的身份验证。
- SQL 语句 `CREATE USER` 和 `ALTER USER` 支持 `ACCOUNT LOCK/UNLOCK` 选项。
- JSON 数据类型和 JSON 函数 GA。
- TiDB 支持 Null-Aware Anti Join。
- TiDB 提供 SQL 查询执行时间的细粒度指标。
- 分区表新增简化 Range 分区的语法糖。
- Range COLUMNS 分区方式支持定义多列。
- TiDB 添加索引的速度提升为原来的 3 倍。
- 降低资源消耗型查询对轻量查询响应时间的影响超 50%。

16.9.1.1 新功能

16.9.1.1.1 SQL

- 新增简化 Range 分区定义的语法糖 (syntactic sugar) Range INTERVAL 分区特性（实验特性）#35683 @mjonss

  提供了新的定义 Range 分区的方式Range INTERVAL 分区，不需要枚举所有分区，可大幅度缩短现有 Range 分区表定义语句冗长的书写方式。语义与原有 Range 分区等价。

- Range COLUMNS 分区方式支持定义多列 #36636 @mjonss

  支持Range COLUMNS 分区，`column_list` 不再限定为单一列，基本功能与 MySQL 等同。

- 分区表EXCHANGE PARTITION 功能 GA #35996 @ymkzpx

- 新增支持下推两个窗口函数 至 TiFlash #5579 @SeaRise

  - `LEAD()`
  - `LAG()`

- 提供轻量级元数据锁，提升 DDL 变更过程 DML 的成功率（实验特性）#37275 @wjhuang2016

  在 TiDB 中，对元数据对象的更改采用的是在线异步变更算法。事务在执行时会获取开始时对应的元数据快照。如果事务执行过程中相关表上发生了元数据的更改，为了保证数据的一致性，TiDB 会返回 `Information schema is changed` 的错误，导致用户事务提交失败。为了解决这个问题，在 TiDB v6.3.0 中，online DDL 算法中引入了元数据锁特性。通过协调表元数据变更过程中 DML 语句和 DDL 语句的优先级，让执行中的 DDL 语句等待持有旧版本元数据的 DML 语句提交，尽可能避免 DML 语句报错。

- 提升添加索引的性能，减少对 DML 事务的影响（实验特性）#35983 @benjamin2037

  TiDB v6.3.0 支持开启添加索引加速功能，提升了创建索引回填过程的速度。开启该功能后，TiDB 添加索引的性能提升约为原来的 3 倍。

16.9.1.1.2 安全

- TiKV 静态加密支持国密算法 SM4 #13041 @jiayang-zheng

  TiKV 的静态加密新增SM4 算法，用户在配置静态加密时，可将 `data-encryption-method` 参数设为 `sm4-ctr`，以启用基于国密算法 SM4 的静态加密能力。

- TiDB 支持国密算法 SM3 的身份验证 #36192 @CbcWestwolf

  TiDB 身份验证新增基于国密算法 SM3 的插件`tidb_sm3_password`，启用此插件后，用户密码将通过 SM3 进行加密存储和验证。

- TiDB-JDBC 支持国密算法 SM3 的身份验证 #25 @lastincisor

  用户密码的身份验证需要客户端的支持，现在TiDB-JDBC 支持国密算法 SM3，你可以使用国密算法 SM3 的身份验证能力通过 TiDB-JDBC 连接到 TiDB。

16.9.1.1.3 可观测性

- 提供 TiDB SQL 查询执行时间的细粒度指标 #34106 @cfzjywxk

  TiDB v6.3.0 提供了细粒度的数据指标，用于对执行时间进行细化观测。通过完整且细分的指标数据，可以清晰地了解 SQL 查询主要的时间消耗，从而快速发现关键问题，节省故障诊断的时间。

- 增强慢日志和 TRACE 语句的输出 #34106 @cfzjywxk

  TiDB v6.3.0 增强了慢日志的内容和 TRACE 的输出。你可以观测到 SQL 语句执行过程中，从 TiDB 解析到 KV RocksDB 落盘全链路的延迟数据，进一步增强 TiDB 的诊断能力。

- TiDB Dashboard 中显示死锁的历史记录 #34106 @cfzjywxk

  从 v6.3.0 起，死锁的历史记录将添加到 TiDB Dashboard。当你通过 TiDB Dashboard 的慢日志等手段发现某些 SQL 等待锁的时间较长时，可以分析 TiDB Dashboard 上的死锁历史记录来定位问题，提升了诊断的易用性。

16.9.1.1.4 性能

- TiFlash 调整 FastScan 功能使用方式（实验特性）#5252 @hongyunyan

  TiFlash 从 v6.2.0 版本开始引入的快速扫描功能 (FastScan)，性能上符合预期，但是使用方式上不够灵活。因此，TiFlash 在 v6.3.0 版本调整 FastScan 功能的使用方式：废弃了 `ALTER TABLE ...SET TiFLASH MODE ...` 语法启用方式，改为使用系统变量 `tiflash_fastscan` 进行控制。

  从 v6.2.0 版本升级到 v6.3.0 版本时，在 v6.2.0 版本的 FastScan 设置将失效，但不影响数据的正常读取。你需要重新使用变量方式设置 FastScan。从 v6.2.0 及更早版本升级到 v6.3.0 时，所有会话默认不开启 FastScan 功能，而是保持一致性的数据扫描功能。

- TiFlash 优化提升多并发场景下的数据扫描性能 #5376 @JinheLin

  TiFlash 通过合并相同数据的读取操作，减少了对相同数据的重复读取，优化了多并发任务情况下的资源开销，提升多并发下的数据扫描性能。避免了以往在多并发任务下，如果涉及相同数据，同一份数据需要在每个任务中分别进行读取的情况，以及可能出现在同一时间内对同一份数据进行多次读取的情况。

  该功能在 v6.2.0 版本以实验特性发布，在 v6.3.0 版本作为正式功能发布。

- TiFlash 副本同步性能优化 #5237 @breezewish

  TiFlash 使用 Raft 协议与 TiKV 进行副本数据同步。在 v6.3.0 版本之前，同步大量副本数据往往耗时较长。v6.3.0 版本优化了 TiFlash 副本同步机制，大幅提升了副本同步速度。因此，使用 BR 恢复数据、使用 TiDB Lightning 导入数据、或全新增加 TiFlash 副本时，副本将迅速地完成同步，你可以及时地使用 TiFlash 进行查询。此外，在 TiFlash 扩容、缩容、或修改 TiFlash 副本数时，TiFlash 副本也将更快地达到安全、均衡的状态。

- TiFlash 针对单个 `COUNT(DISTINCT)` 进行三阶段聚合 #37202 @fixdb

  TiFlash 将有且仅有单个 `COUNT(DISTINCT)` 聚合的查询改写为三阶段分布式执行的聚合，从而提高并发度，并提升性能。

- TiKV 支持日志回收 #214 @LykxSassinator

  TiKV Raft Engine 支持日志回收功能。该特性能够显著降低网络磁盘上 Raft 日志追加过程中的长尾延迟，提升了 TiKV 写入负载下的性能。

- TiDB 支持 Null-Aware Anti Join #37525 @Arenatlx

  TiDB v6.3.0 引入了新的连接类型Null-Aware Anti Join (NAAJ)。NAAJ 在处理集合操作时能够感知集合是否为空，或是否有空值，优化了 IN 和 = ANY 等操作的执行效率，提升 SQL 性能。

- 增加优化器 hint 控制哈希连接的驱动端 #35439 @Reminiscent

  在 v6.3.0 版本中，优化器引入了两个新的 hint，HASH_JOIN_BUILD() 和 HASH_JOIN_PROBE()，用于隐式地指定哈希连接的行为，同时分别指定哈希连接的构建端和探测端。如果优化器未选到最优执行计划，可以使用这两个 hint 来干预执行计划。

- 允许在会话级别内联展开公共表表达式 (CTE) #36514 @elsa0520

  在 v6.2.0 中，引入了优化器提示 MERGE，允许对 CTE 内联进行展开，使得 CTE 查询结果的消费者能够在 TiFlash 内并行执行。在 v6.3.0 中，又进一步添加了会话级变量tidb_opt_force_inline_cte，允许在会话中修改这个行为，提升了易用性。

16.9.1.1.5 事务

- 悲观事务可以延迟唯一约束检查 #36579 @ekexium

  你可以通过系统变量tidb_constraint_check_in_place_pessimistic 来控制悲观事务中唯一约束检查的时间点。该变量默认关闭。当开启该变量时（设为 ON），TiDB 会将悲观事务中的加锁操作和唯一约束检测推迟到必要的时候进行，以此提升批量 DML 操作的性能。

- 优化 Read-Committed 隔离级别中对 TSO 的获取 #36812 @TonsnakeLin

  在 Read-Committed 隔离级别中，引入新的系统变量tidb_rc_write_check_ts 控制语句对 TSO 的获取方式。在 Plan Cache 命中的情况下，通过降低对 TSO 的获取频率，提升批量 DML 的执行效率，降低跑批类任务的执行时间。

16.9.1.1.6 稳定性

- 降低资源消耗型查询对轻量查询响应时间的影响 #13313 @glorv

当资源消耗型查询与轻量查询同时运行时，轻量查询的响应时间会被严重影响。在这种情况下，通常希望优先快速处理轻量查询，以保证交易类负载的服务质量。因此 TiKV 在 v6.3.0 中优化了读请求的调度机制，使资源消耗型的查询在每一轮执行的时间更符合预期。这个特性大幅降低了资源消耗型查询对轻量查询的影响，使混合工作负载的 P99 延迟降低了 50% 以上。

- 修改优化器统计信息过期时的默认统计信息使用策略 #27601 @xuyifangreeneyes

  在 v5.3.0 版本，TiDB 引入系统变量tidb_enable_pseudo_for_outdated_stats 控制优化器在统计信息过期时的行为，默认为 ON，即保持旧版本行为不变：当 SQL 涉及的对象的统计信息过期时，优化器认为该表上除总行数以外的统计信息不再可靠，转而使用 pseudo 统计信息。经过一系列测试和用户实际场景分析，TiDB 在新版本中将 tidb_enable_pseudo_for_outdated_stats 的默认值改为 OFF，即使统计信息过期，优化器也仍会使用该表上的统计信息，这有利于执行计划的稳定性。

- TiKV 正式支持关闭 Titan 引擎 @tabokie

  正式支持对在线 TiKV 节点关闭 Titan 引擎。

- 缺少 GlobalStats 时自动选择分区静态剪裁 #37535 @Yisaer

  当启用分区动态剪裁时，优化器依赖GlobalStats 进行执行计划的选择。在GlobalStats 收集完成前，使用 pseudo 统计信息可能会造成性能回退。在 v6.3.0 版本中，如果在GlobalStats 收集未完成的情况下打开动态分区裁剪开关，TiDB 会维持静态分区剪裁的状态，直到 GlobalStats 收集完成。该方式确保在切换分区剪裁策略时系统性能保持稳定。

16.9.1.1.7 易用性

- 解决基于 SQL 的数据放置规则功能和构建 TiFlash 副本功能的冲突 #37171 @lcwangchao

  TiDB 在 v6.0.0 版本提供基于 SQL 的数据放置规则功能，但是由于实现机制问题，该功能和构建 TiFlash 副本功能有冲突。v6.3.0 版本进行改进优化，解决了这两个功能的冲突，使这两个功能更加方便易用。

16.9.1.1.8 MySQL 兼容性

- 新增支持 `REGEXP_INSTR()`、`REGEXP_LIKE()`、`REGEXP_REPLACE()` 和 `REGEXP_SUBSTR()` 4 个正则表达式函数，提升 TiDB 与 MySQL 8.0 的兼容性 #23881 @windtalker

  这些函数与 MySQL 的兼容性可参考正则函数与 MySQL 的兼容性。

- `CREATE USER` 和 `ALTER USER` 支持 `ACCOUNT LOCK/UNLOCK` 选项 #37051 @CbcWestwolf

  在执行`CREATE USER` 创建用户时，允许使用 `ACCOUNT LOCK/UNLOCK` 选项，限定被创建的用户是否被锁定。锁定后的用户不能正常登录数据库。

  对于已存在的用户，可以通过`ALTER USER` 使用 `ACCOUNT LOCK/UNLOCK` 选项，修改用户的锁定状态。

- JSON 数据类型和 JSON 函数 GA #36993 @xiongjiwei

  JSON 是一种流行的数据格式，被大量的程序设计所采用。TiDB 在早期版本就引入了JSON 支持，兼容 MySQL 的 JSON 数据类型和一部分 JSON 函数。在 v6.3.0 版本中，这些功能正式 GA，为 TiDB 提供了更丰富的数据类型支持，同时表达式索引和生成列也增加了对 JSON 函数的支持，进一步提升了 TiDB 对 MySQL 的兼容能力。

16.9.1.1.9 备份恢复

- PITR 支持GCS 和 Azure Blob Storage 作为备份存储 @joccau

  部署在 GCP 或者 Azure 上的用户，将 TiDB 集群升级至 v6.3.0 就可以使用 PITR 功能。

- BR 支持 AWS S3 Object Lock #13442 @3pointer

  你可以在 AWS 开启 S3 Object Lock 功能来防止备份数据写入后被修改或者删除。

16.9.1.1.10 数据迁移

- TiDB Lightning 支持将 Apache Hive 导出的 Parquet 文件导入到 TiDB #37536 @buchuitoudegou

- DM 新增配置项 `safe-mode-duration` #6224 @okJiang

  DM 任务配置文件中新增一个配置项`safe-mode-duration`，用户可以自行调节 DM 异常重启后进入安全模式的持续时间，默认值 60 秒。当设置为 “0s” 时，表示 DM 异常重启后尝试进入安全模式会报错。

16.9.1.1.11 数据共享与订阅

- TiCDC 支持对多个异地目标数据源进行数据复制 #5301 @sdojjy

  为了支持从一个 TiDB 集群复制数据到多个不同的异地数据系统，自 v6.3.0 开始，TiCDC 节点可以部署到多个不同的异地的机房中，用于分别负责对应机房的数据复制任务，以支撑各种复杂的异地数据复制使用场景和部署形态。

- TiCDC 支持维护上下游数据一致性快照 (Sync point) #6977 @asddongmen

  在灾备复制场景下，TiCDC 支持周期性地维护一个下游数据快照，使得该下游快照能与上游数据的快照保持一致。借助此能力，TiCDC 能更好地匹配读写分离应用场景，帮助用户降本增效。

- TiCDC 支持平滑升级 #4757 @overvenus @3AceShowHand

  用户使用TiUP (>=v1.11.0) 和 TiDB Operator (>=v1.3.8) 可以平滑滚动升级 TiCDC 集群。升级期间数据同步延时保持在 30 秒内，提高了稳定性，让 TiCDC 能更好地支持延时敏感型业务。

16.9.1.2 兼容性变更

16.9.1.2.1 系统变量

变量名	修改类型	描述
default_authentication_plugin ↪	修改	扩展可选值范围：增加 tidb_sm3_password ↪ 。设置为 tidb_sm3_password ↪ 时，用户密码验证的加密算法为国密算法 SM3。

变量名	修改类型	描述
sql_require_primary_key	新增	用于控制表是否必须有主键。启用该变量后，如果在没有主键的情况下创建或修改表，将返回错误。
tidb_adaptive_closest_read_threshold	新增	用于控制当 tidb_replica_read 设置为 closest-adaptive 时，优先将读请求发送至 TiDB server 所在区域副本的阈值。
tidb_constraint_check_in_place_pessimistic	新增	用于控制悲观事务中唯一约束检查的时间点。

变量名	修改类型	描述
tidb_ddl_disk_quota	新增	用于设置创建索引的回填过程中本地存储空间的使用限制。仅在 tidb_ddl_enable_fast_reorg 开启的情况下生效。
tidb_ddl_enable_fast_reorg	新增	用于控制是否开启添加索引加速功能，来提升创建索引回填过程的速度。
tidb_ddl_flashback_concurrency	新增	用于控制 flashback cluster 的并发数。在 v6.3.0，该变量控制的功能尚未完全生效，请保留默认值。

变量名	修改类型	描述
tidb_enable_exchange_partition	废弃	用于设置是否启用exchange partitions with tables 特性。自 v6.3.0 开始，其取值将固定为默认值 ON，即默认开启 exchange partitions with tables 。

变量名	修改类型	描述
tidb_enable_foreign_key	新增	用于控制是否开启 FOREIGN KEY 特性。在 v6.3.0，该变量控制的功能尚未完全生效，请保留默认值。
tidb_enable_general_plan_cache	新增	用于控制是否开启 General Plan Cache。在 v6.3.0，该变量控制的功能尚未完全生效，请保留默认值。
tidb_enable_metadata_lock	新增	用于设置是否开启元数据锁特性。

变量名	修改类型	描述
tidb_enable_null_aware_anti_join ↪	新增	用于控制 TiDB 对特殊集合算子 `NOT IN` 和 `!= ALL` 引导的子查询产生的 Anti Join 是否采用 Null-Aware Hash Join 的执行方式。
tidb_enable_pseudo_for_outdated_stats ↪	修改	用于控制优化器在统计信息过期时的行为。默认值由 `ON` 改为 `OFF`，即使统计信息过期，优化器也仍会使用该表上的统计信息。

变量名	修改类型	描述
tidb_enable_rate_limit_action ↪	修改	用于控制是否为读数据的算子开启动态内存控制功能。打开该变量可能会导致内存不受 tidb_mem_quota_query ↪ 控制，而加剧 OOM 风险，故将默认值由 ON 调整为 OFF。
tidb_enable_tiflash_read_for_write_stmt ↪	新增	用于控制写 SQL 中的读取是否会下推到 TiFlash。在 v6.3.0，该变量控制的功能尚未完全生效，请保留默认值。

变量名	修改类型	描述
tidb_enable_unsafe_substitute ↪	新增	用于控制是否对生成列中表达式替换使用不安全的替换方式。
tidb_general_plan_cache_size ↪	新增	用于控制 General Plan Cache 最多能够缓存的计划数量。在 v6.3.0，该变量控制的功能尚未完全生效，请保留默认值。
tidb_last_plan_replayer_token ↪	新增	只读变量，用于获取当前会话中最后一个 PLAN ↪ REPLAYER ↪ ↪ DUMP 的结果。

变量名	修改类型	描述
tidb_max_paging_size ↪	新增	用来设置 coprocessor 协议中 paging size 的最大的行数。
tidb_opt_force_inline_cte ↪	新增	用于控制是否强制开启 inline CTE。默认值为 OFF，即默认不强制开启 inline CTE。
tidb_opt_three_stage_distinct_agg ↪	新增	用于控制在 MPP 模式下是否将 COUNT(↪ DISTINCT ↪) 聚合改写为三阶段分布式执行的聚合。默认为 ON。

变量名	修改类型	描述
tidb_partition_prune_mode ↪	修改	用于设置是否开启分区表动态裁剪模式。自 v6.3.0 起，该变量默认值修改为 dynamic。

变量名	修改类型	描述
tidb_rc_read_check_ts ↪	修改	用于优化读语句时间戳的获取，适用于悲观事务 RC 隔离级别下读写冲突较少的场景。由于这个行为只针对特定业务负载，而对其他类型的负载可能造成性能回退。自 v6.3.0 起，该变量的作用域由 GLOBAL 或 SESSION 修改为 INSTANCE 级别，允许只针对部分 TiDB 实例打开。

变量名	修改类型	描述
tidb_rc_write_check_ts ↪	新增	用于优化时间戳的获取，适用于悲观事务 RC 隔离级别下点写冲突较少的场景，开启此变量可以避免点写语句获取全局 times-tamp 带来的延迟和开销。

变量名	修改类型	描述
tiflash_fastscan ↪	新增	用于控制是否启用 FastScan 功能。如果开启 FastScan 功能（设置为 ON 时），TiFlash 可以提供更高效的查询性能，但不保证查询结果的精度和数据一致性。

16.9.1.2.2 配置文件参数

配置文件	配置项	修改类型	描述
TiDB	temp-dir	新增	TiDB 用于存放临时数据的路径。需要使用 TiDB 节点本地存储的功能会将数据临时存放在这个目录下。默认值为 /tmp/tidb。

配置文件	配置项	修改类型	描述
TiKV	auto-adjust-pool-size	新增	控制是否开启自动调整线程池的大小。开启此配置可以基于当前的 CPU 使用情况，自动调整统一处理读请求的线程池 (UnifyReadPool) 的大小，优化 TiKV 的读性能。
TiKV	data-encryption-method	修改	扩展可选值范围：增加 `sm4-ctr`。设置为 `sm4-ctr` 时，数据将采用国密算法 SM4 加密后进行存储。

配置文件	配置项	修改类型	描述
TiKV	enable-log-recycle	新增	控制 Raft Engine 是否回收过期的日志文件。该配置项启用时，Raft Engine 将保留逻辑上被清除的日志文件，用于日志回收，减少写负载的长尾延迟。仅在 format-version 的值大于等于 2 时，该配置项才生效。
TiKV	format-version	新增	指定 Raft Engine 的日志文件格式版本。v6.3.0 以前的默认值为 1。v6.1.0 及以后版本的 TiKV 可以读取该格式。v6.3.0 及以后版本，该配置项默认值为 2，TiKV 可以读取该格式。

配置文件	配置项	修改类型	描述
TiKV	log-backup.enable	修改	默认值在 v6.3.0 以前是 false，v6.3.0 开始设为 true。
TiKV	log-backup.max-flush-interval	修改	默认值在 v6.3.0 以前是 5min，v6.3.0 开始设为 3min。
PD	enable-diagnostic	新增	控制是否开启诊断功能。默认值为 false。
TiFlash	dt_enable_read_thread	废弃	该参数从 v6.3.0 开始废弃，默认开启此功能且不能关闭。
DM	safe-mode-duration	新增	自动安全模式的持续时间。
TiCDC	enable-sync-point	新增	控制是否开启 sync point 功能。
TiCDC	sync-point-interval	新增	控制 sync point 功能对齐上下游 snapshot 的时间间隔。

配置文件	配置项	修改类型	描述
TiCDC	sync-point-retention	新增	sync point 功能在下游表中保存的数据的时长，超过这个时间的数据会被清理。
TiCDC	sink-uri.memory	废弃	废弃 memory 排序方式，不建议在任何情况下使用。

16.9.1.2.3 其他

- 日志备份支持 GCS 和 Azure Blob Storage 作为备份存储。
- 日志备份功能兼容分区交换 (Exchange Partition) DDL。
- 不再支持通过 `ALTER TABLE ...SET TiFLASH MODE ...` 语法启用或禁用FastScan 功能。从 v6.2.0 版本升级到 v6.3.0 版本时，在 v6.2.0 版本的 FastScan 设置将失效，但不影响数据的正常读取。你需要重新使用变量方式设置 FastScan。从 v6.2.0 及更早版本升级到 v6.3.0 时，所有会话默认不开启 FastScan 功能，而是保持一致性的数据扫描功能。
- 在 Linux AMD64 架构的硬件平台部署 TiFlash 时，CPU 必须支持 AVX2 指令集。确保命令 `cat /proc/cpuinfo | grep avx2` 有输出。而在 Linux ARM64 架构的硬件平台部署 TiFlash 时，CPU 必须支持 ARMv8 架构。确保命令 `cat /proc/cpuinfo | grep 'crc32' | grep 'asimd'` 有输出。通过使用向量扩展指令集，TiFlash 的向量化引擎能提供更好的性能。
- TiDB 支持的最小 HAProxy 版本为 v1.5。使用 v1.5 到 v2.1 之间的 HAProxy 时，需要在 `mysql-check` 中配置 `post-41`。建议使用 HAProxy v2.2 或更高版本。

16.9.1.3 废弃功能

自 v6.3.0 起，TiCDC 不再支持配置 Pulsar Sink。建议使用 StreamNative 官方维护的 kop 作为替代方案。

16.9.1.4 改进提升

- TiDB
  - 改进表名检查方式，由大小写敏感变为大小写不敏感 #34610 @tiancaiamao
  - 为 `init_connect` 值增加解析器检查流程，提升了与 MySQL 的兼容性 #35324 @CbcWestwolf
  - 改进新连接产生时的警告日志 #34964 @xiongjiwei
  - 优化用于查询 DDL 历史任务的 HTTP API，提供对 `start_job_id` 参数的支持 #35838 @tiancaiamao

  - 在 JSON 路径的语法有误时报错 #22525 #34959 @xiongjiwei
  - 修复一个伪共享问题，提升 Join 操作的性能 #37641 @gengliqi
  - PLAN REPLAYER 命令支持一次导出多条 SQL 语句的执行计划信息，提升了问题排查效率 #37798 @Yisaer

- TiKV

  - 支持配置 `unreachable_backoff` 避免 Raftstore 发现某个 Peer 无法连接时广播过多消息 #13054 @5kbpers
  - 提升 TSO 服务不可用的容忍度 #12794 @pingyu
  - 支持动态调整 RocksDB 进行 subcompaction 的并发个数 (`rocksdb.max-sub-compactions`) #13145 @ethercflow
  - 优化合并空 Region 的性能 #12421 @tabokie
  - 支持更多正则表达式函数 #13483 @gengliqi
  - 支持根据 CPU 使用情况自动调整 UnifyReadPool 线程池的大小 #13313 @glorv

- PD

  - 更新 TiDB Dashboard 中 TiKV IO MBps 指标的计算方式 #5366 @YiniXu9506
  - 将 TiDB Dashboard 中 `metrics` 更改为 `monitoring` #5366 @YiniXu9506

- TiFlash

  - 支持下推 `elt` 到 TiFlash #5104 @Willendless
  - 支持下推 `leftShift` 到 TiFlash #5099 @AnnieoftheStars
  - 支持下推 `castTimeAsDuration` 到 TiFlash #5306 @AntiTopQuark
  - 支持下推 `HexIntArg/HexStrArg` 到 TiFlash #5107 @YangKeao
  - 重构改进 TiFlash 的执行流解释器，支持新的解释器 Planner #4739 @SeaRise
  - 改进了 TiFlash memory tracker 的准确度 #5609 @bestwoody
  - 提升了 UTF8_BIN/ASCII_BIN/LATIN1_BIN/UTF8MB4_BIN collation 的列相关运算的速度 #5294 @solotzg
  - 在后台计算 ReadLimiter 的 I/O 吞吐量 #5401 #5091 @Lloyd-Pottiger

- Tools

  - Backup & Restore (BR)

    * PITR 可以聚合日志备份中生成的小日志文件，大幅减小文件数量 #13232 @Leavrth
    * PITR 支持在恢复完成后自动按照上游集群配置设置 TiFlash 副本 #37208 @YuJuncen

  - TiCDC

    * 提升上游 TiDB 引入并行 DDL 框架后 TiCDC 的兼容性 #6506 @lance6716
    * 支持在 MySL sink 出错时将 DML 语句的 `start ts` 输出到日志文件 #6460 @overvenus
    * 优化 API `api/v1/health`，使其返回的 TiCDC 集群健康状态更准确 #4757 @overvenus
    * 采用异步的模式实现 MQ sink 和 MySQL sink，提升 sink 的吞吐能力 #5928 @hicqu @hi-rustin
    * 删除已经废弃了的 pulsar sink #7087 @hi-rustin
    * 忽略掉与 changefeed 不相关的 DDL 语句以提升同步性能 #6447 @asddongmen

  - TiDB Data Migration (DM)

    * 提升与 MySQL 8.0 数据源的兼容性 #6448 @lance6716

    * 优化 DDL 执行逻辑，当执行 DDL 超时时，转为异步查询 DDL 执行结果 #4689 @lyzx2001

  - TiDB Lightning

    * 在 S3 URL 中添加新参数 `role-arn` 和 `external-id`，支持从其他账户访问 S3 数据 #36891 @dsdashun

16.9.1.5 错误修复

- TiDB

  - 修复权限检查跳过 PREAPRE 语句的问题 #35784 @lcwangchao
  - 修复系统变量 `tidb_enable_noop_variable` 能够设置为 WARN 的问题 #36647 @lcwangchao
  - 修复存在表达式索引定义时，`INFORMAITON_SCHEMA.COLUMNS` 表的 `ORDINAL_POSITION` 列可能不正确的问题 #31200 @bb7133
  - 修复时间戳大于 `MAXINT32` 时 TiDB 不报错的问题 #31585 @bb7133
  - 修复使用企业版插件时无法启动 TiDB server 的问题 #37319 @xhebox
  - 修复 `SHOW CREATE PLACEMENT POLICY` 输出结果不正确的问题 #37526 @xhebox
  - 修复临时表能用于交换分区的问题 #37201 @lcwangchao
  - 修复查询 `INFORMATION_SCHEMA.TIKV_REGION_STATUS` 返回不正确结果的问题 @zimulala
  - 修复使用 `EXPLAIN` 查询视图时不进行权限检查的问题 #34326 @hawkingrei
  - 修复 JSON `null` 不能被更新为 `NULL` 的问题 #37852 @YangKeao
  - 修复 DDL 任务的 `row_count` 不准确的问题 #25968 @Defined2014
  - 修复 `FLASHBACK TABLE` 运行不正常的问题 #37386 @tiancaiamao
  - 修复无法处理典型 MySQL 协议中 `prepared` 语句 flag 的问题 #36731 @dveeden
  - 修复在极端情况下，启动 TiDB 可能进入错误状态的问题 #36791 @xhebox
  - 修复 `INFORMATION_SCHEMA.VARIABLES_INFO` 不遵循安全增强模式 (SEM) 的问题 #37586 @CbcWestwolf
  - 修复带 UNION 算子的查询中转换字符串出错的问题 #31678 @cbcwestwolf
  - 修复在 TiFlash 中为分区表开启动态模式时结果出错的问题 #37254 @wshwsh12
  - 修复 TiDB 中二进制字符串和 JSON 之间的转换和对比行为与 MySQL 不兼容的问题 #31918 #25053 @YangKeao
  - 修复 TiDB 中的 `JSON_OBJECTAGG` 和 `JSON_ARRAYAGG` 在二进制值上与 MySQL 不兼容的问题 #25053 @YangKeao
  - 修复比较 JSON opaque 值时造成崩溃的问题 #37315 @YangKeao
  - 修复 JSON 聚合函数不能使用单精度浮点数的问题 #37287 @YangKeao
  - 修复 UNION 运算符可能会非预期地返回空结果的问题 #36903 @tiancaiamao
  - 修复 `castRealAsTime` 表达式的结果和 MySQL 不兼容的问题 #37462 @mengxin9014
  - 修复悲观 DML 操作锁定非唯一索引键的问题 #36235 @ekexium
  - 修复与自动提交 (auto-commit) 模式更改相关的事务提交行为 #36581 @cfzjywxk
  - 修复带 DML 算子的 `EXPLAIN ANALYZE` 语句可能在事务提交完成前返回结果的问题 #37373 @cfzjywxk
  - 修复在某些情况下 UPDATE 语句错误地消除了 projection 导致 `Can't find column` 报错的问题 #37568 @AilinKid
  - 修复执行 Join Reorder 操作时，会错误地下推 Outer Join 条件的问题 #37238 @AilinKid
  - 修复 IN 和 `NOT IN` 子查询在某些 pattern 下会报 `Can't find column` 的问题 #37032 @AilinKid
  - 修复 UPDATE 语句中带公共表表达式 (CTE) 的情况下会报 `Can't find column` 的问题 #35758 @AilinKid
  - 修复错误的 PromQL #35856 @Defined2014

- TiKV

  - 修复 PD Region heartbeat 连接异常中断后未重新连接的问题 #12934 @bufferflies
  - 修复 Raftstore 线程繁忙时，可能会出现 Region 重叠的问题 #13160 @5kbpers
  - 修复 PD 客户端可能会出现死锁的问题 #13191 @bufferflies #12933 @BurtonQin
  - 修复关闭加密时 TiKV panic 的问题 #13081 @jiayang-zheng
  - 修复 Dashboard 中 Unified Read Pool CPU 表达式错误的问题 #13086 @glorv
  - 修复当有一个 TiKV 实例出现网络隔离时，一段时间内服务不可用问题 #12966 @cosven
  - 修复误报 `PessimisticLockNotFound` 的问题 #13425 @sticnarf
  - 修复某些场景下 PITR 可能丢失数据的问题 #13281 @YuJuncen
  - 修复存在时间较长的悲观事务时，checkpoint 没有推进的问题 #13304 @YuJuncen
  - 修复 TiKV 无法区分 JSON 中时间类型（`DATETIME`、`DATE`、`TIMESTAMP` 和 `TIME`）和 `STRING` 类型的问题 #13417 @YangKeao
  - 修复 JSON bool 和其他 JSON value 的比较行为与 MySQL 不兼容的问题 #13386 #37481 @YangKeao

- PD

  - 修复开启 `enable-forwarding` 时，gRPC 处理返回错误不当导致 PD panic 的问题 #5373 @bufferflies
  - 修复不健康的 Region 可能导致 PD panic 的问题 #5491 @nolouch
  - 修复 PD 可能没创建 TiFlash Learner 副本的问题 #5401 @HunDunDM

- TiFlash

  - 修复取消查询时 window function 可能会导致 TiFlash 崩溃的问题 #5814 @SeaRise
  - 修复 `CAST(value AS datetime)` 输入数据无法转成 `DATETIME` 时会导致 TiFlash sys CPU 异常高的问题 #5097 @xzhangxian1008
  - 修复 `CAST(Real/Decimal AS time)` 函数执行结果与 MySQL 不一致的问题 #3779 @mengxin9014
  - 修复存储中的过时数据可能无法删除的问题 #5570 @JaySon-Huang
  - 修复 page GC 可能导致无法创建表格的问题 #5697 @JaySon-Huang
  - 修复使用包含 `NULL` 值的列创建主键时导致崩溃的问题 #5859 @JaySon-Huang

- Tools

  - Backup & Restore (BR)

    * 修复导致 PITR 的 checkpoint 信息过时的问题 #36423 @YuJuncen
    * 修复在恢复时配置过高的 concurrency 会导致 Region 不均衡的问题 #37549 @3pointer
    * 修复在 TiCDC 存在时，PITR 进度无法推进的问题 #37822 @YuJuncen
    * 修复当外部存储的鉴权 Key 中存在某些特殊符号时，会导致备份恢复失败的问题 #37469 @MoCuishle28

  - TiCDC

    * 修复 TiCDC 对含有 grpc 服务的非法 PD 地址报错不准确的问题 #6458 @crelax
    * 修复 `cdc cli changefeed list` 命令不返回 failed changefeed 的问题 #6334 @asddongmen
    * 修复如果 changefeed 初始化失败会导致 TiCDC 不可用的问题 #6859 @asddongmen

  - TiDB Binlog

    * 修复 compressor 设为 `gzip` 时 Drainer 无法正确发送请求至 Pump 的问题 #1152 @lichunzhu

  - TiDB Data Migration (DM)

    * 修复 DM 报错 `Specified key was too long` 的问题 #5315 @lance6716

    * 修复 relay 报错时可能导致 goroutine 泄露问题 #6193 @lance6716
    * 修复当 `collation_compatible` 设置为 `"strict"` 时，DM 可能生成有重复排序规则的 SQL 语句的问题 #6832 @lance6716
    * 修复 DM-worker 日志中出现过多警告信息 “found error when get timezone from binlog status_vars” 的问题 #6628 @lyzx2001
    * 修复数据同步过程中，latin1 字符集数据可能损坏的问题 #7028 @lance6716
  - TiDB Lightning
    * 修复 TiDB Lightning 不支持 Parquet 文件中以斜线 (/)、数字、非 ASCII 字符开头的特殊列名的问题 #36980 @D3Hunter

16.9.1.6 贡献者

感谢来自 TiDB 社区的贡献者们：

- @An-DJ
- @AnnieoftheStars
- @AntiTopQuark
- @blacktear23
- @BurtonQin（首次贡献者）
- @crelax
- @eltociear
- @fuzhe1989
- @erwadba
- @jianzhiyao
- @joycse06
- @morgo
- @onlyacat
- @peakji
- @rzrymiak
- @tisonkun
- @whitekeepwork
- @Ziy1-Tan

16.10 v6.2

16.10.1 TiDB 6.2.0 Release Notes

发版日期：2022 年 8 月 23 日

TiDB 版本：6.2.0-DMR

> **注意：**
>
> TiDB 6.2.0-DMR 的用户文档已归档。如无特殊需求，建议使用 TiDB 数据库的最新 LTS 版本。

在 6.2.0-DMR 版本中，你可以获得以下关键特性：

- TiDB Dashboard 支持可视化执行计划，查询计划展示更直观。
- TiDB Dashboard 新增Monitoring 页面用于性能分析和优化。
- TiDB 锁视图支持乐观事务被阻塞的信息，方便快速定位锁冲突。
- TiFlash 引入新的存储格式 PageStorage V3，提升稳定性和性能。
- 实现细粒度数据交换 (shuffle) 使窗口函数 (Window function) 可以利用多线程并行计算。
- 引入新的 DDL 并行执行框架，减少 DDL 阻塞，大幅提升执行效率。
- TiKV 支持自适应调整 CPU 使用率，确保数据库稳定高效运行。
- 支持point-in-time recovery (PITR)，允许恢复备份集群的历史任意时间点的快照。
- TiDB Lightning 使用 Physical Import Mode 导入时限制调度范围从集群降低到表级别。
- Backup & Restore (BR) 支持恢复用户和权限数据，备份恢复体验更平滑。
- TiCDC 支持过滤指定类型的 DDL 事件，解锁更多数据同步场景。
- 事务中支持SAVEPOINT 机制，可以灵活地控制事务内的回退节点。
- 单 ALTER TABLE 语句增删改多个列或索引，方便实现 SQL 功能，提升产品易用性。
- 支持RawKV 跨集群复制。

16.10.1.1 新功能

16.10.1.1.1 SQL

- 正式支持通过 SQL 语句对 TiFlash 副本立即触发物理数据整理 (Compaction)

  TiFlash 后台会依据特定条件、自动对物理数据进行整理 (Compaction)，减少无用数据的积压，并优化数据存储结构。在数据整理被自动触发前，TiFlash 数据表中往往存在一定量的无用数据。该特性支持用户自行选择合适的时机、手动执行 SQL 语句来对 TiFlash 中的物理数据立即进行整理，从而减少存储空间占用，并提升查询性能。此功能在 TiDB v6.1 作为一项实验功能引入，在 v6.2.0 版本正式发布。

  用户文档 #4145 @breezewish

16.10.1.1.2 可观测性

- TiDB Dashboard 从 PD 拆离

  将 TiDB Dashboard 拆分到 Monitoring 节点，减少监控组件对 PD 节点的影响，让 PD 节点更加稳定。

  @Hawkson-jee

- TiDB Dashboard 新增 Monitoring 页面

  TiDB Dashboard 新增 Monitoring 页面，展示在业务性能调优中所需的核心指标。用户可基于数据库时间的系统优化方法，利用该页面进行性能分析和优化。用户可以从全局、自顶向下的角度分析用户响应时间和数据库时间，确认用户响应时间的瓶颈是否在数据库中。如果瓶颈在数据库中，你可以通过数据库时间概览和 SQL 延迟的分解，定位数据库内部的瓶颈点，并进行针对性的优化。

  用户文档 #1381 @YiniXu9506

- TiDB Dashboard 支持可视化执行计划

  在 Statements 和 Slow Query 中提供可视化执行计划和基础问题诊断的能力。这是一种全新的查询计划的展示方式，目标是通过图形化的手段展示 Query 查询计划的每个步骤，从而使得用户能够更加直观方便地了解查询执行计划的细节。对于复杂的大型查询语句，可视化的展示方式对于深入理解其执行过程大有裨益。另外，系统会自动对每一个 Query 的执行计划进行分析，发现执行计划中潜在问题，提供优化方向。用户可以采用这些优化建议，降低特定 Query 的查询时长。

  用户文档 #1224 @time-and-fate

- 锁视图支持乐观事务被阻塞的信息

  大量锁冲突往往会造成严重的性能问题，而定位锁冲突是这类性能问题排查的必要手段之一。TiDB v6.2.0 之前，支持通过系统视图 `INFORMATION_SCHEMA.DATA_LOCK_WAITS` 查看锁冲突的关系，但是不支持乐观事务被悲观锁阻塞的情况。TiDB v6.2.0 扩展`DATA_LOCK_WAITS` 视图，提供乐观事务被悲观锁阻塞情况下的冲突关系，帮助用户快速定位锁冲突，同时为业务改进提供依据，从而减少这类锁冲突的发生频率，提升系统整体性能。

  用户文档 #34609 @longfangsong

16.10.1.1.3 性能

- 优化器增加对外连接顺序的提示

  在 v6.1.0 中引入的优化器提示 `LEADING` 可干预表的连接顺序，但是这个提示并不能应用在包含了外连接的查询中，见`LEADING` 文档。在 v6.2.0 中 TiDB 解除了这个限制，`LEADING` 提示对外连接同样生效。在包含外连接的查询中，你同样可以利用 `LEADING` 指定表的连接顺序，获得更好的 SQL 执行性能，并避免执行计划的突变。

  用户文档 #29932 @Reminiscent

- 引入新的优化器提示 `SEMI_JOIN_REWRITE` 改善 `EXISTS` 查询性能

  在部分场景下，带有 `EXISTS` 的查询并不能选择最优的执行计划，导致查询运行时间过长。在 v6.2.0 版本中，优化器增加了新的改写规则，用于优化此类场景下执行计划的选择，并通过 `SEMI_JOIN_REWRITE` 提示强制优化器进行改写，从而得到更好的查询性能。

  用户文档 #35323 @winoros

- 引入一个优化器提示 `MERGE`，提升分析类查询性能

  公共表表达式 (CTE) 是简化查询逻辑的有效方法，广泛应用于复杂查询的编写。在 v6.2.0 前，CTE 在 TiFlash 环境中还不能自动展开，这在一定程度上限制了 MPP 的运行效率。在 v6.2.0 中，TiDB 引入一个 MySQL 兼容的优化器提示 `MERGE`。利用这个提示，优化器允许对 CTE 内联进行展开，使得 CTE 查询结果的消费者能够在 TiFlash 内并行执行，从而提升此类分析查询的性能。

  用户文档 #36122 @dayicklp

- 优化了个别分析场景的聚合操作性能

  在使用 TiFlash 的 OLAP 场景下对列进行聚合操作时，如果被聚合列因分布不均而存在严重数据倾斜，并且被聚合列有大量不同的取值，那么该列的 `COUNT(DISTINCT)` 查询效率偏低。在 v6.2.0 版本中，引入了新的改写规则，针对单列的 `COUNT(DISTINCT)` 查询进行了优化，提升此类场景下的查询性能。

  用户文档 #36169 @fixdb

- TiDB 支持 DDL 并发执行

  TiDB v6.2.0 引入新的 DDL 并行执行框架，在不同表对象上的 DDL 可以并发执行，解决了之前不同表之间 DDL 相互阻塞的问题。同时在不同表对象的追加索引、列类型变更等场景下支持并行执行，大幅提升执行效率。

  #32031 @wjhuang2016

- 优化器增强了对字符串条件匹配的估算方式

  在字符串条件匹配场景下，行数估算偏差有可能影响最优执行计划的生成，例如条件 `like '%xyz'` 或者借助 `regex()` 的正则表达式匹配。为了提升此类场景下估算的准确度，TiDB v6.2.0 增强了这个场景下的估算方式。新的方法会结合统计信息的 TopN 信息以及系统变量，在提升估算准确率的同时，还引入了手动调整的方法，进而提升此场景下的 SQL 性能。

  用户文档 #36209 @time-and-fate

- 支持窗口函数下推到 TiFlash 进行多线程并行计算

  通过实现执行过程中的细粒度的数据交换 (shuffle) 能力，窗口函数的计算由单线程变为多线程并行计算，成倍降低查询响应时间。此性能改进不改变用户使用行为。你可以通过控制变量`tiflash_fine_grained_shuffle_batch_size` 来控制 shuffle 的粒度。

  用户文档 #4631 @guo-shaoge

- TiFlash 支持新版本的存储格式

  TiFlash 通过改进存储格式，大幅减轻了在高并发、高负载场景下 GC 造成 CPU 占用高的问题，可以有效减少后台任务 IO 流量，提升高并发、高负载下的稳定性。同时新版本存储格式可以显著降低空间放大，减少磁盘空间浪费。

  v6.2.0 默认以新版本存储格式保存数据。从更低版本升级到 6.2.0 版本后，不支持原地降级，否则更低版本的 TiFlash 无法识别新版本的数据格式。

  建议用户在升级前阅读TiFlash v6.2.0 升级帮助。

  用户文档 #3594 @JaySon-Huang @lidezhu @jiaqizho

- TiFlash 优化提升多并发场景下的数据扫描性能（实验特性）

  在 v6.2.0，TiFlash 通过合并相同数据的读取操作，减少对于相同数据的重复读取，优化了多并发任务情况下的资源开销，提升多并发下的数据扫描性能。避免了以往在多并发任务下，如果涉及相同数据，同一份数据需要在每个任务中分别进行读取的情况，以及可能出现在同一时间内对相同数据进行多次读取的情况。

  用户文档 #5376 @JinheLin

- TiFlash 新增 FastScan 功能，降低一致性保证，提高读写速度（实验特性）

  TiDB 从 v6.2.0 版本引入快速扫描功能 (FastScan)，支持跳过一致性检测以大幅提高速度，适用于离线分析任务等对于数据的精度和一致性要求不高的场景。以往，为了保证数据一致性，TiFlash 在数据扫描过程中需要对数据进行一致性检查，从多个不同版本的数据中找到符合要求的数据。

  从更低的版本升级到 v6.2.0 版本时，所有的表默认不开启 FastScan 功能，而是保持一致性的数据扫描功能。你可以为每一张表独立开启 FastScan 功能。如果在 v6.2.0 版本设定表开启 FastScan 功能后，当降级到更低版本时 FastScan 功能设置将失效，但不影响数据的正常读取。这种情况等同于强一致性的数据扫描功能。

  用户文档 #5252 @hongyunyan

16.10.1.1.4 稳定性

- TiKV 支持自适应调整 CPU 使用率（实验特性）

  数据库通常会使用后台进程来执行一些内部操作，通过采集各种统计信息，帮助用户定位性能问题，生成更优的执行计划，从而提升数据库的稳定性和性能。然而如何平衡后台操作和前台操作的资源开销，在不影响用户日常数据库使用的基础上如何更高效地采集信息，一直是数据库领域最为头疼的问题之一。

  从 v6.2.0 开始，TiDB 支持通过 TiKV 配置文件设置后台请求的 CPU 使用率，进而限制自动统计信息收集等后台操作在 TiKV 的 CPU 使用比例，避免极端情况下后台操作抢占对用户操作的资源，确保数据库稳定高效运行。

  同时，TiDB 还支持 CPU 使用率自动调节的功能，这时 TiKV 会根据实例的 CPU 占用情况，自适应地对后台请求占用的 CPU 资源进行动态调整。该功能默认关闭。

  用户文档 #12503 @BornChanger

16.10.1.1.5 易用性

- TiKV 支持通过命令行参数提供更详细的配置信息

  TiKV 配置文件可以实现对 TiKV 实例的管理。但是对运行时间长且多人管理的 TiKV 实例，用户修改了哪些配置文件，配置的默认值是什么，难以进行方便地比对。这在集群升级、迁移时容易造成困扰。从 TiDB v6.2.0 开始，tikv-server 新增命令行参数`—-config-info`，支持输出 TiKV 所有配置项的默认值和当前值，帮助用户快速验证 TiKV 进程的启动参数，提升易用性。

  用户文档 #12492 @glorv

16.10.1.1.6 MySQL 兼容性

- TiDB 支持使用一个 ALTER TABLE 语句增删改多个列或索引

  在 v6.2.0 之前，TiDB 仅支持单一 DDL 变更，导致用户在迁移异构数据库时经常会遇见 DDL 操作不兼容的情况，需要耗费额外的精力将复杂的 DDL 修改成 TiDB 支持的多个简单 DDL。同时还有一些用户依赖 ORM 框架，实现 SQL 组装，最终出现了 SQL 不兼容等问题。TiDB 从 v6.2.0 开始，支持使用 ALTER TABLE 语句修改一个表的多个模式对象，方便了用户 SQL 实现，也提升了产品易用性。

  用户文档 #14766 @tangenta

- 事务中支持 SAVEPOINT

  事务是数据库保证 ACID 特性的一系列连续操作的逻辑集合。在一些复杂业务场景下，你可能需要管理一个事务的大量操作，有时候需要在事务内实现部分操作的回退能力。Savepoint 就是针对事务内部实现的可命名保存点机制，通过这个机制，你可以灵活地控制事务内的回退节点，从而实现更复杂的事务管理能力，实现更为多样的业务设计。

  用户文档 #6840 @crazycs520

16.10.1.1.7 数据迁移

- BR 支持恢复用户和权限数据

  BR 快照备份和恢复支持恢复用户和权限数据，用户不再需要额外的方案恢复用户和权限数据，只需要在使用 BR 恢复数据时指定参数 `--with-sys-table`。

  用户文档 #35395 @D3Hunter

- 支持基于变更日志的备份和恢复实现 Point-in-time recovery

  基于变更日志和快照数据的备份恢复实现 PITR (point-in-time recovery) 功能，允许用户在新集群上恢复备份集群的历史任意时间点的快照。该功能可以满足以下的用户需求：

  - 降低灾备场景下的 RPO，如实现十几分钟的 RPO；
  - 用于处理业务数据写错的案例，如回滚业务数据到出错事件前；
  - 业务历史数据审计，满足司法审查的需求。
  - 该功能初版存在着一些使用限制，详细情况请参考功能使用文档。

  用户文档 #29501 @joccau

- DM 支持增量持续数据校验（实验特性）

  增量持续数据校验用于在数据同步过程中，持续对比上游 binlog 与下游实际写入记录是否存在异常，例如错误同步、记录缺失等。此特性用以解决常见的全量数据校验方案的滞后性，资源消耗过重等问题。

  用户文档 #4426 @D3Hunter @buchuitoudegou

- 自动识别 Amazon S3 bucket 所在的区域

  数据迁移任务可自动判断 S3 bucket 所在区域，不再需要显式传递区域参数。

  #34275 @WangLe1321

- 支持为 TiDB Lightning 配置磁盘资源配额（实验特性）

  当 TiDB Lightning 使用物理导入模式 (backend='local') 进行导入时，sorted-kv-dir 需要具备足以容纳数据源总量的空间，当空间不足时可能致使导入任务失败。新增的 `disk_quota` 配置项可以用于限定 TiDB Lightning 的磁盘空间使用总量，当 sorted-kv-dir 存储空间较少时也可以正常完成导入任务。

  用户文档 #446 @buchuitoudegou

- TiDB Lightning 支持使用 Physical Import Mode 导入数据到生产集群

  TiDB Lightning 原有的物理导入模式 (backend='local') 对目标集群影响较大，例如导入过程将停止 PD 调度等，因此仅适用于目标集群初次导入数据。

  TiDB Lightning 在现有基础上做了改进，导入时可以暂停对应表的调度，从而将影响范围由集群级别降低到表级别，即非导入的表仍可进行读写操作。

  此特性无需手动配置，目标 TiDB 集群版本在 v6.1.0 及以上且 TiDB Lightning 在 v6.2.0 及以上时自动生效。

  用户文档 #35148 @gozssky

- 调整TiDB Lightning 在线文档，使其目录结构更加合理和清晰。同时对文档中关于“后端模式”的描述进行了修改，使用 Physical Import Mode 替代原有 local backend，使用 Logical Import Mode 替代原有 tidb backend，以降低新用户的理解难度。

16.10.1.1.8 数据共享与订阅

- 支持 RawKV 跨集群复制（实验特性）

  支持订阅 RawKV 的数据变更，并通过新的 TiKV-CDC 组件将变更实时同步到下游 TiKV 集群，从而实现 RawKV 的跨集群复制能力。

  用户文档 #11965 @pingyu

- 支持过滤 DDL 和 DML 事件

  在一些特殊的场景下，用户可能希望对 TiDB 增量数据变更日志进行一定规则的过滤，例如过滤 Drop Table 等高风险 DDL。自 v6.2.0 起，TiCDC 支持过滤指定类型的 DDL 事件，支持基于 SQL 表达式过滤 DML 事件，从而适应更多的数据同步场景。

  用户文档 #6160 @asddongmen

16.10.1.2 兼容性变更

16.10.1.2.1 系统变量

变量名	修改类型	描述
tidb_enable_new_cost_interface	新增	控制是否使用重构后的代价模型Cost Model Version 2。

变量名	修改类型	描述
tidb_cost_model_version	新增	TiDB 在进行物理优化时会使用代价模型来进行索引选择和算子选择，该变量用于选择代价模型的版本。TiDB v6.2.0 引入了代价模型 Cost Model Version 2，在内部测试中比此前版本的代价模型更加准确。
tiflash_fine_grained_shuffle_stream_count	新增	当窗口函数下推到 TiFlash 执行时，可以通过该变量控制窗口函数执行的并行度。

变量名	修改类型	描述
tiflash_fine_grained_shuffle_batch_size	新增	细粒度 shuffle 功能开启时，该变量控制发送端发送数据的攒批大小，即发送端累计行数超过该值就会进行一次数据发送。
tidb_default_string_match_selectivity	新增	设置过滤条件中的 `like`、`rlike`、`regexp` 函数在行数估算时的默认选择率，以及是否对这些函数启用 TopN 辅助估算。
tidb_enable_array_type_storage_snapshot	新增	控制 `ANALYZE` 读取历史时刻的数据还是读取最新的数据。

变量名	修改类型	描述
tidb_generate_binary_plan	新增	用于指定是否在 slow log 和 statement summary 里包含以二进制格式编码的执行计划。

变量名	修改类型	描述
tidb_opt_skew_distinct_agg	新增	用于设置优化器是否将带有 DISTINCT 的聚合函数（例如 SELECT b, COUNT (DISTINCT a) FROM t GROUP BY b ）改写为两层聚合函数（例如 SELECT b, COUNT (a) FROM (SELECT b, a FROM t GROUP BY b, a)t GROUP BY b ）。

变量名	修改类型	描述
tidb_enable_noop_variables	新增	用于设置 SHOW [GLOBAL] VARIABLES 是否显示 noop 变量。
tidb_enable_concurrent_ddl	新增	用于控制是否让 TiDB 使用并发 DDL 语句。
tidb_min_paging_size	新增	用来设置 coprocessor 协议中 paging size 的最小的行数。
tidb_txn_commit_batch_size	新增	用于控制 TiDB 向 TiKV 发送的事务提交请求的批量大小。
tidb_enable_change_multi_schema	删除	支持使用一个 ALTER TABLE 语句增删改多个列或索引。

变量名	修改类型	描述
tidb_enable_outer_join_reorder	修改	用来控制 TiDB 的 join reorder 是否支持 outer join，在 v6.1.0 中为 ON，即默认开启。自 v6.2.0 起，该变量默认为 OFF，即默认关闭。

16.10.1.2.2 配置文件参数

配置文件	配置项	修改类型	描述
TiDB	feedback-probability	删除	该配置不再生效，不推荐使用。
TiDB	query-feedback-limit	删除	该配置不再生效，不推荐使用。
TiKV	server.simplify-metrics	新增	控制是否开启精简返回的 Metrics 数据。
TiKV	quota.background-cpu-time	新增	限制处理 TiKV 后台读写请求所使用的 CPU 资源使用量。

配置文件	配置项	修改类型	描述
TiKV	quota.background-write-bandwidth	新增	限制后台事务写入的带宽（暂未生效）。
TiKV	quota.background-read-bandwidth	新增	限制后台事务读取数据和 Coprocessor 读取数据的带宽（暂未生效）。
TiKV	quota.enable-auto-tune	新增	是否支持 quota 动态调整。如果打开该配置项，TiKV 会根据 TiKV 实例的负载情况动态调整对后台请求的限制 quota。
TiKV	rocksdb.defaultcf.format-version	新增	设置 SST 文件的格式版本。
TiKV	rocksdb.lockcf.format-version	新增	设置 SST 文件的格式版本。
TiKV	rocksdb.writecf.format-version	新增	设置 SST 文件的格式版本。
TiKV	rocksdb.enable-pipelined-commit	删除	该配置不再生效。
TiKV	gc-merge-rewrite	删除	该配置不再生效。
TiKV	log-backup.enable	新增	TiKV 是否开启日志备份功能。

配置文件	配置项	修改类型	描述
TiKV	log-backup.file-size-limit	新增	日志备份任务备份的数据达到一定大小时，自动 flush 到外部存储中。
TiKV	log-backup.initial-scan-pending-memory-quota	新增	增量扫描数据时，用于存放扫描数据的缓存大小。
TiKV	log-backup.max-flush-interval	新增	日志备份任务将备份数据写入到 External Storage 的最大间隔时间。
TiKV	log-backup.initial-scan-rate-limit	新增	增量扫描数据时，用于扫描时吞吐限流参数。
TiKV	log-backup.num-threads	新增	日志备份功能使用的线程数。
TiKV	log-backup.temp-path	新增	临时目录路径，TiKV 备份日志预先先写入临时目录，然后 flush 到外部存储中。

配置文件	配置项	修改类型	描述
PD	replication-mode.dr-auto-sync.wait-async-timeout	删除	废弃未生效的配置项。
PD	replication-mode.dr-auto-sync.wait-sync-timeout	删除	废弃未生效的配置项。
TiFlash	`storage.format_version`	修改	`format_version` 默认值变更为 4，v6.2.0 及以后版本的默认文件格式，优化了写放大问题，同时减少了后台线程消耗。
TiFlash	profiles.default.enable_read_thread	新增	控制存储引擎是否使用线程池读取数据。默认值为 false，不使用线程池读取数据。
TiFlash	profiles.default.dt_page_gc_threshold	新增	表示 PageStorage 单个数据文件中有效数据的最低比例。

配置文件	配置项	修改类型	描述
TiCDC	–overwrite-checkpoint-ts	新增	在 cdc cli changefeed resume 子命令下新增的参数。
TiCDC	–no-confirm	新增	在 cdc cli changefeed resume 子命令下新增的参数。
DM	mode	新增	Validator 参数，取值可以是 full、fast，默认是 none，即不开启校验。
DM	worker-count	新增	Validator 参数，后台校验的 validation worker 数量，默认是 4 个。
DM	row-error-delay	新增	Validator 参数，某一行多久没有验证通过会报错，默认是 30 min。
TiDB Lightning	tikv-importer.store-write-bwlimit	新增	限制 TiDB Lightning 向每个 TiKV Store 写入带宽带宽大小，默认为 0，表示不限制。

配置文件	配置项	修改类型	描述
TiDB Lightning	tikv-importer.disk-quota	新增	用于设置 TiDB Lightning 可以使用的磁盘配额。

16.10.1.2.3 其他

- TiFlash 的存储格式 (format_version) 不能直接从 4 降级到 3，详情请参考TiFlash v6.2.0 升级帮助。
- 在 v6.2.0 以及后续版本，强烈建议保留 dt_enable_logical_split 的默认值 false，不要将其修改为 true。具体请参考已知问题 #5576。
- 如果备份集群包含 TiFlash，执行 PITR 后恢复集群的数据不包含 TiFlash 副本，需要手动恢复 TiFlash 副本；执行 exchange partition DDL 会导致 PITR restore 出错；上游数据库使用 TiDB Lightning Physical 方式导入的数据，无法作为数据日志备份下来，数据导入后需要执行一次全量备份。关于 PITR 功能使用的其他事项，请参考PITR 使用限制。
- 从 v6.2.0 开始，BR 支持通过手动指定参数 --with-sys-table=true 来恢复 mysql schema 下的表。
- 使用 ALTER TABLE 增删改多个列或索引时，TiDB 会根据执行前的 schema 结构来验证一致性，而不管同一 DDL 语句中的更改。同时，语句的执行顺序上 TiDB 和 MySQL 在某些场景不兼容。
- 在集群中，如果 TiDB 组件的版本为 v6.2.0 及以上，则 TiKV 组件版本不得低于 v6.2.0。
- TiKV 新增配置项 split.region-cpu-overload-threshold-ratio 支持在线修改。
- 慢查询日志以及 INFORMATION_SCHEMA 中的系统表 statements_summary 和 slow_query 新增输出 binary_plan，即以二进制格式编码的执行计划。
- SHOW TABLE ... REGIONS 返回的结果中新增两列：SCHEDULING_CONSTRAINTS 以及 SCHEDULING_STATE，表示对应 Region 在 Placement In SQL 中设置的调度规则以及当前的调度状态。
- 从 v6.2.0 开始，你可以通过 TiKV-CDC 组件实现 RawKV 的 Change Data Capture (CDC)。
- 使用 ROLLBACK TO SAVEPOINT 语句将事务回滚到指定保存点时，MySQL 会释放该保存点之后才持有的锁，但在 TiDB 悲观事务中，不会立即释放该保存点之后才持有的锁，而是等到事务提交或者回滚时，才释放全部持有的锁。
- 从 v6.2.0 开始, 执行 SELECT tidb_version() 返回的信息中会包含 Store 类型（tikv 或者 unistore）
- TiDB 不再有隐藏的系统变量。
- 新增两个系统表：
  - INFORMATION_SCHEMA.VARIABLES_INFO：用于查看 TiDB 系统变量相关的信息。
  - PERFORMANCE_SCHEMA.SESSION_VARIABLES：用于查看 TiDB session 系统变量相关的信息。

16.10.1.3 废弃功能

从 TiDB v6.2.0 起，使用 BR 备份和恢复 RawKV 的功能废弃。

16.10.1.4 改进提升

- TiDB
  - 支持了 `SHOW COUNT(*)WARNINGS` 以及 `SHOW COUNT(*)ERRORS` #25068 @likzn
  - 对某些系统变量增加一些合法性验证 #35048 @morgo
  - 优化了一些类型转换的错误提示 #32447 @fanrenhoo
  - `KILL` 命令增加了对 DDL 操作的支持 #24144 @morgo
  - 提升了 `SHOW TABLES/DATABASES LIKE …` 命令的输出与 MySQL 的兼容性，输出的列名称中会包含 `LIKE` 的值 #35116 @likzn
  - 提升了 JSON 相关函数的性能 #35859 @wjhuang2016
  - 提升了使用 SHA-2 时登录密码的验证速度 #35998 @virusdefender
  - 精简了一些日志的输出 #36011 @dveeden
  - 优化了 coprocessor 通信协议，大幅度降低读取数据时 TiDB 进程的内存消耗，进而可进一步缓解扫全表场景及 Dumpling 导出数据场景下的 OOM 问题。该通信协议是否开启由 `tidb_enable_paging` 系统变量控制（作用域为 SESSION 或 GLOBAL），当前默认关闭。可将该变量值设为 `true` 进行开启 #35633 @tiancaiama @wshwsh12
  - 进一步优化了部分算子（HashJoin、HashAgg、Update、Delete）内存追踪的准确度 (#35634, #35631, #35635, @wshwsh12) (#34096, @ekexium)
  - 系统表 `INFORMATION_SCHEMA.DATA_LOCK_WAIT` 支持记录乐观事务的锁信息 #34609 @longfangson
  - 新增部分事务相关的监控指标 #34456 @longfangsong
- TiKV
  - 支持通过 gzip 压缩 metrics 响应减少 HTTP body 大小 #12355 @glorv
  - 优化了 TiKV Grafana Dashboard 的可读性 #12007 @kevin-xianliu
  - 优化 Apply 算子的写入性能 #12898 @ethercflow
  - 支持动态调整 RocksDB 进行 subcompaction 的并发个数 (`rocksdb.max-sub-compactions`) #13145 @ethercflow
- PD
  - 支持 Region CPU 维度统计信息并增强 Load Base Split 的覆盖场景 #12063 @Jmpotato
- TiFlash
  - 优化了 TiFlash MPP 的错误处理流程，增强了稳定性 #5095 @windtalker @yibin87
  - 优化了 UTF8_BIN/UTF8MB4_BIN Collation 的比较和排序操作 #5294 @solotzg
- Tools
  - Backup & Restore (BR)
    * 优化了全量备份数据组织形式，解决大规模集群备份时遇到的 S3 限流问题 #30087 @MoCuishle28
  - TiCDC

    * 优化了多 Region 场景下，runtime 上下文切换带来过多性能开销的问题 #5610 @hicqu
    * 优化了 redo log 的性能、修复 meta 和数据不一致的问题 (#6011 @CharlesCheung96) (#5924 @zhaoxinyu) (#6277 @hicqu)

  - TiDB Lightning

    * 增加更多可重试错误，包括 EOF、Read index not ready、Coprocessor 超时等 #36674, #36566 @D3Hunter

  - TiUP

    * 使用 TiUP 新部署集群时，Node-exporter 组件将使用 1.3.1 版本，Blackbox-exporter 组件将使用 0.21.1 版本，使用新版本组件可以确保在不同系统环境下成功部署

16.10.1.5 错误修复

- TiDB

  - 修复了在查询分区表中如果查询条件中有分区键且两者使用了不同的 COLLATE 时会错误的进行分区裁剪的问题 #32749 @mjonss
  - 修复了 SET ROLE 中如果 host 中有大写字母无法匹配到已经 GRANT 的 ROLE 的问题 #33061 @morgo
  - 修复了无法 DROP AUTO_INCREMENT 的列的问题 #34891 @Defined2014
  - 修复了 SHOW CONFIG 会显示一些已经移除掉的配置项的问题 #34867 @morgo
  - 修复了 SHOW DATABASES LIKE … 大小写敏感的问题 #34766 @e1ijah1
  - 修复了 SHOW TABLE STATUS LIKE … 大小写敏感的问题 #7518 @likzn
  - 修复了 max-index-length 检查在非严格模式下仍然报错的问题 #34931 @e1ijah1
  - 修复了 ALTER COLUMN ... DROP DEFAULT 不起作用的问题 #35018 @Defined2014
  - 修复了创建表时列的默认值和列类型不一致没有自动修正的问题 #34881 @Lloyd-Pottiger
  - 修复了在 DROP USER 之后 mysql.columns_priv 表中相关的数据没有被同步删除的问题 #35059 @lcwangchao
  - 通过禁止在一些系统的 schema 内创建表，修复了由此导致的 DDL 卡住的问题 #35205 @tangenta
  - 修复了某些情况下查询分区表可能导致 “index-out-of-range” 和 “non used index” 的问题 #35181 @mjonss
  - 通过支持 INTERVAL expr unit + expr 形式的语法，修复了该语句会报错的问题 #30253 @mjonss
  - 修复了在事务中创建的本地临时表无法找到的问题 #35644 @djshow832
  - 修复了给 ENUM 列设置 COLLATE 导致 panic 的问题 #31637 @wjhuang2016
  - 修复了当某台 PD 宕机时，由于没有重试其他 PD 节点，导致查询表 INFORMATION_SCHEMA.TIKV_REGION_STATUS 时请求失败的问题 #35708 @tangenta
  - 修复在 SET character_set_results = GBK 后 SHOW CREATE TABLE … 不能正确显示 SET 和 ENUM 列的问题 #31338 @tangenta
  - 修复了系统变量 tidb_log_file_max_days 和 tidb_config 的作用域不正确的问题 #35190 @morgo
  - 修复了类型是 ENUM 或者 SET 的列在 SHOW CREATE TABLE 的输出与 MySQL 不兼容的问题 #36317 @Defined2014
  - 修复了创建表时指定类型为 LONG BYTE 列的行为与 MySQL 不兼容的问题 #36239 @Defined2014
  - 修复了设置 auto_increment = x 对临时表无效的问题 #36224 @djshow832
  - 修复了在并发修改列的情况下可能导致 DEFAULT VALUE 不正确的问题 #35846 @wjhuang2016
  - 避免向非健康状态的 TiKV 节点发送请求，以提升可用性 #34906 @sticnarf
  - 修复了 LOAD DATA 语句中列的列表不生效的问题 #35198 @SpadeA-Tang
  - 修复部分场景非唯一二级索引被误加悲观锁的问题 #36235 @ekexium

- TiKV
  - 避免在悲观事务中报出 `Write Conflict` 错误 #11612 @sticnarf
  - 修复了在悲观事务中使用 Async Commit 导致重复提交记录的问题 #12615 @sticnarf
  - 修复了 TiKV API 从 `storage.api-version = 1` 升级为 `storage.api-version = 2` 时 panic 的问题 #12600 @pingyu
  - 修复了 TiKV 和 PD 配置文件中 Region size 不一致的问题 #12518 @5kbpers
  - 修复了 TiKV 持续重连 PD 的问题 #12506, #12827 @Connor1996
  - 修复了对空字符串进行类型转换导致 TiKV panic 的问题 #12673 @wshwsh12
  - 修复了 `DATETIME` 类型的数据包含小数部分和 `Z` 后缀导致检查报错的问题 #12739 @gengliqi
  - 修复 Apply 写入 TiKV RocksDB 的 perf context 粒度过大的问题 #11044 @LykxSassinator
  - 修复当backup/import/cdc 配置项设置错误时 TiKV 无法启动的问题 #12771 @3pointer
  - 修复同时分裂和销毁一个 peer 时可能导致 panic 的问题 #12825 @BusyJay
  - 修复在 Region merge 时 source peer 通过 snapshot 追日志时可能导致 panic 的问题 #12663 @BusyJay
  - 修复 `max_sample_size` 为 0 时 ANALYZE 可能导致 panic 的问题 #11192 @LykxSassinator
  - 修复了启用 Raft Engine 时未清理加密密钥的问题 #12890 @tabokie
  - 修复了 `get_valid_int_prefix` 函数与 TiDB 不兼容的问题，例如 `FLOAT` 类型被错误地转换成 `INT` #13045 @guo-shaoge
  - 修复了新创建的 Region Commit Log Duration 较高导致 QPS 下降的问题 #13077 @Connor1996
  - 修复 PD Region heartbeat 连接异常中断后未重新连接的问题 #12934 @bufferflies
- Tools
  - Backup & Restore (BR)

    修复了限速备份后，BR 没有重置速度限制的问题 #31722 @MoCuishle28

16.10.1.6 贡献者

感谢来自 TiDB 社区的贡献者们：

- e1ijah1
- PrajwalBorkar
- likzn
- rahulk789
- virusdefender
- joycse06
- morgo
- ixuh12
- blacktear23
- johnhaxx7
- GoGim1
- renbaoshuo
- Zheaoli
- fanrenhoo
- njuwelkin
- wirybeaver

- hey-kong
- fatelei
- eastfisher：首次贡献者
- Juneezee：首次贡献者

16.11 v6.1

16.11.1 TiDB 6.1.7 Release Notes

发版日期：2023 年 7 月 12 日

TiDB 版本：6.1.7

试用链接：快速体验 | 生产部署 | 下载离线包

16.11.1.1 改进提升

- TiDB
  - 内部事务重试使用悲观模式避免重试失败，降低耗时 #38136 @jackysp
- Tools
  - TiCDC
    * 支持 Batch UPDATE DML 语句，提升 TiCDC 的同步性能 #8084 @amyangfei
  - TiDB Lightning
    * 数据导入完成后使用 SQL 方式校验 checksum，提升数据校验的稳定性 #41941 @GMHDBJD

16.11.1.2 错误修复

- TiDB
  - 修复 `processInfo` 为空导致的 panic 问题 #43829 @zimulala
  - 修复 resolve lock 在 PD 时间跳变的情况下可能 hang 住的问题 #44822 @zyguan
  - 修复包含 CTE 的查询可能导致磁盘可用空间不足的问题 #44477 @guo-shaoge
  - 修复同时使用 CTE 和关联子查询可能导致查询结果出错或者 panic 的问题 #44649 #38170 #44774 @winoros @guo-shaoge
  - 修复 `SELECT CAST(n AS CHAR)` 语句中的 n 为负数时，查询结果出错的问题 #44786 @xhebox
  - 修复某些情况下 TiDB 查询 panic 的问题 #40857 @Dousir9
  - 修复 SQL compile 报错的日志未脱敏的问题 #41831 @lance6716
  - 修复当分区表定义中使用了 `FLOOR()` 函数对分区列进行取整时 `SELECT` 语句返回错误的问题 #42323 @jiyfhust
  - 修复 Region 分裂时查询分区表可能报错的问题 #43144 @lcwangchao
  - 修复读取统计信息时不必要的内存占用问题 #42052 @xuyifangreeneyes
  - 修复创建大量空分区表后内存占用过多的问题 #44308 @hawkingrei

  - 修复开启 tidb_opt_agg_push_down 时查询可能返回错误结果的问题 #44795 @AilinKid
  - 修复公共表表达式在 join 时可能得到错误结果的问题 #38170 @wjhuang2016
  - 修复在一些罕见的情况下，悲观事务的残留悲观锁在 GC resolve lock 时可能影响数据正确性的问题 #43243 @MyonKeminta
  - 修复缓存表执行新增列操作后，新增列值为 NULL 而非列的默认值的问题 #42928 @lqs
  - 修复分区表在 Index Join 的 probe 阶段找不到对应行而报错的问题 #43686 @AilinKid @mjonss
  - 修复删除数据库导致 GC 推进慢的问题 #33069 @tiancaiamao
  - 修复 ON UPDATE 语句没有正确更新主键导致数据索引不一致的问题 #44565 @zyguan
  - 修复在重命名表期间 TiCDC 可能丢失部分行变更的问题 #43338 @tangenta
  - 修复 Placement Rule 在分区表下的行为问题，使得删除的分区 Placement Rule 可以被正确设置并回收 #44116 @lcwangchao
  - 修复当 tidb_scatter_region 变量设置为开启时，对某个分区进行 TRUNCATE 操作后没有自动分裂 Region 的问题 #43174 #43028 @jiyfhust
  - 修复分区特别多并且带有 TiFlash 副本的分区表在执行 TRUNCATE TABLE 时，出现写冲突导致 DDL 重试的问题 #42940 @mjonss
  - 修复窗口函数计算下推到 TiFlash 时执行计划构造错误的问题 #43922 @gengliqi
  - 修复在带有非关联子查询的语句中使用公共表表达式 (CTE) 可能导致结果错误的问题 #44051 @winoros
  - 修复 memTracker 配合 cursor fetch 使用导致内存泄漏的问题 #44254 @YangKeao
  - 修复 INFORMATION_SCHEMA.DDL_JOBS 表中 QUERY 列的数据长度可能超出列定义的问题 #42440 @tiancaiamao
  - 修复 min, max 查询结果出错的问题 #43805 @wshwsh12
  - 修复 analyze 表时报语法错误的问题 #43392 @guo-shaoge
  - 修复 SHOW PROCESSLIST 语句无法显示子查询时间较长语句的事务的 TxnStart 的问题 #40851 @crazycs520
  - 修复对于执行中的 DROP TABLE 操作，ADMIN SHOW DDL JOBS 的结果中缺少表名的问题 #42268 @tiancaiamao
  - 修复 IPv6 环境下显示错误的 TiDB 地址的问题 #43260 @nexustar
  - 修复在使用 AES_DECRYPT 表达式时，SQL 报错 runtime error: index out of range 的问题 #43063 @lcwangchao
  - 修复在创建分区表时使用 SUBPARTITION 没有警告提醒的问题 #41198 #41200 @mjonss
  - 修复使用 CTE 的查询导致 TiDB 卡住的问题 #43749 #36896 @guo-shaoge
  - 修复在 TRUNCATE 分区表的某个分区时可能造成分区的 Placement Rule 失效的问题 #44031 @lcwangchao
  - 修复在谓词下推的情况下 CTE 结果错误的问题 #43645 @winoros
  - 修复与自动提交 (auto-commit) 模式更改相关的事务提交行为 #36581 @cfzjywxk

- TiKV

  - 修复 TiDB Lightning 可能导致 SST 文件泄露的问题 #14745 @YuJuncen
  - 修复加密 Key ID 冲突会导致旧 Key 被删除的问题 #14585 @tabokie
  - 修复 Continuous Profiling 中的文件句柄泄露的问题 #14224 @tabokie

- PD

  - 修复 gRPC 返回非预期格式错误的问题 #5161 @HuSharp

- Tools

- Backup & Restore (BR)
  - 修复某些情况下误报 `resolved lock timeout` 的问题 #43236 @YuJuncen
  - 修复集群中 TiKV 出现宕机导致备份速度降低的问题 #42973 @YuJuncen
- TiCDC
  - 修复 TiCDC 同步到 Kafka-on-Pulsar 时不能正确建立连接的问题 #8892 @hi-rustin
  - 修复 PD 地址或 leader 出现故障时 TiCDC 不能自动恢复的问题 #8812 #8877 @asddongmen
  - 修复当下游为 Kafka 时，TiCDC 查询下游的元信息频率过高导致下游负载过大的问题 #8957 #8959 @hi-rustin
  - 修复 PD 出现网络隔离或 PD Owner 节点重启等故障时 TiCDC 卡住问题 #8808 #8812 #8877 @asddongmen
- TiDB Lightning
  - 修复逻辑导入模式下，导入期间下游删除表可能导致 TiDB Lightning 元信息未及时更新的问题 #44614 @dsdashun
  - 修复竞态条件可能导致磁盘配额 (disk quota) 不准确的问题 #44867 @D3Hunter
  - 修复 NotLeader 错误未正确重试的问题 #43055 @lance6716
  - 修复 Lightning 内部 Buf 存在泄露可能导致 OOM 的问题 #40400 @buchuitoudegou
  - 修复宽表导入时可能出现 OOM 的问题 #43728 @D3Hunter
- TiDB Binlog
  - 修复 etcd client 初始化时没有自动同步最新节点信息的问题 #1236 @lichunzhu
  - 更新 TiKV Client 版本，解决 Drainer 因为 TiKV Client 版本过老导致 panic 的问题 #1170 @lichunzhu
  - 修复未过滤失败状态的 DDL 导致任务报错的问题 #1228 @lichunzhu

16.11.2 TiDB 6.1.6 Release Notes

发版日期：2023 年 4 月 12 日

TiDB 版本：6.1.6

试用链接：快速体验 | 生产部署 | 下载离线包

16.11.2.1 兼容性变更

- TiCDC 修复了 Avro 编码 FLOAT 类型数据错误的问题 #8490 @3AceShowHand

  在升级 TiCDC 集群到 v6.1.6 或更高的 v6.1.x 版本时，如果使用 Avro 同步的表包含 FLOAT 类型数据，请在升级前手动调整 Confluent Schema Registry 的兼容性策略为 None，使 changefeed 能够成功更新 schema。否则，在升级之后 changefeed 将无法更新 schema 并进入错误状态。

16.11.2.2 提升改进

- TiDB
  - Prepared Plan Cache 支持缓存 BatchPointGet 计划 #42125 @qw4990
  - Index Join 支持更多的 SQL 格式 #40505 @Yisaer
- TiKV
  - 支持在小于 1 core 的 CPU 下启动 TiKV #13586 #13752 #14017 @andreid-db

16.11.2.3 Bug 修复

- TiDB
  - 修复 `ignore_plan_cache` hint 对 INSERT 语句可能不生效的问题 #40079 #39717 @qw4990
  - 修复了 `indexMerge` 遇到错误后可能会导致 TiDB 崩溃的问题 #41047 #40877 @guo-shaoge @windtalker
  - 修复错误下推包含虚拟列的 TopN 算子到 TiKV 或 TiFlash 导致结果错误的问题 #41355 @Dousir9
  - 修复了使用 `Prepare` 或 `Execute` 查询某些虚拟表时无法将表 ID 下推，导致在大量 Region 的情况下 PD OOM 的问题 #39605 @djshow832
  - 修复 Plan Cache 处理 `int_col in (decimal...)` 条件时可能缓存 FullScan 计划的问题 #40224 @qw4990
  - 修复 IndexMerge 计划在 SET 类型列上可能生成错误区间的问题 #41273 #41293 @time-and-fate
  - 修复了无符号的 `TINYINT/SMALLINT/INT` 和小于 0 的 `DECIMAL/FLOAT/DOUBLE` 类型比较时，结果可能出错的问题 #41736 @LittleFall
  - 修复了查询 `INFORMATION_SCHEMA.CLUSTER_SLOW_QUERY` 表导致 TiDB 服务器 OOM 的问题，在 Grafana dashboard 中查看慢查询记录的时候可能会触发该问题 #33893 @crazycs520
  - 修复 Range 分区允许多个 `MAXVALUE` 分区的问题 #36329 @u5surf
  - 修复 Plan Cache 可能缓存 Shuffle 算子导致返回错误结果的问题 #38335 @qw4990
  - 修复了时区中的数据争用可能导致数据和索引不一致问题 #40710 @wjhuang2016
  - 修复了 `indexMerge` 中可能会出现 goroutine 泄露的问题 #41545 #41605 @guo-shaoge
  - 修复在使用 Cursor Fetch 且在 Execute、Fetch、Close 之间运行其它语句后，Fetch 与 Close 命令可能会返回错误结果或造成 TiDB Panic 的问题 #40094 [@YangKeao](https://github.com/YangKeao)
  - 修复了使用 DDL 修改浮点类型时，保持长度不变且减少小数位后，旧数据仍然保持原样的问题 #41281 [@zimulala](https://github.com/zimulala)
  - 修复了 Join `information_schema.columns` 表会造成 TiDB panic 的问题 #32459 [@tangenta](https://github.com/tangenta)
  - 修复了生成执行计划过程中，因为获取的 InfoSchema 不一致而导致的 TiDB panic 的问题 #41622 [@tiancaiamao](https://github.com/tiancaiamao)
  - 修复 TiFlash 执行中遇到生成列会报错的问题 #40663 @guo-shaoge
  - 修复当同一个 SQL 中出现多个不同的分区表时，TiDB 可能执行得到错误结果的问题 #42135 @mjonss
  - 修复 Plan Cache 可能缓存 Shuffle 算子导致返回错误结果的问题 #38335 @qw4990 @fzzf678
  - 修复使用 Index Merge 的方式读取包含 SET 类型列的表时，结果可能出错的问题 #41293 @time-and-fate
  - 修复在开启 Prepared Plan Cache 的情况下，索引全表扫可能会报错的问题 #42150 @fzzf678
  - 修复在 DDL 执行过程中，使用 PointGet 读取表的 SQL 语句可能会在执行时抛出 panic 的问题 #41622 @tiancaiamao
  - 修复事务内执行 PointUpdate 之后，`SELECT` 结果不正确的问题 #28011 @zyguan
  - 定期清理过期的 Region 缓存，避免内存泄漏和性能下降问题 #40461 @sticnarf @zyguan
  - 修复 `INSERT IGNORE` 和 `REPLACE` 语句对不修改 value 的 key 没有加锁的问题 #42121 @zyguan
- TiKV
  - 修复转换 `const Enum` 类型到其他类型时报错的问题 #14156 @wshwsh12
  - 修复 CPU 配额限制的问题 13084 @BornChanger
  - 修复 Snapshot Last Index 不正确的问题 12618 @LintianShi
- PD
  - 修复 Region Scatter 接口可能导致 leader 分布不均匀的问题 #6017 @HunDunDM
  - 修复 Online Unsafe Recovery 超时机制不生效的问题 #6107 @v01dstar

- TiFlash
  - 修复半连接在计算笛卡尔积时，使用内存过量的问题 #6730 @gengliqi
  - 修复 TiFlash 日志搜索过慢的问题 #6829 @hehechen
  - 修复了开启 new collation 后 TopN/Sort 算子结果可能出错的问题 #6807 @xzhangxian1008
  - 修复了 Decimal 转换在某些情况下进位错误的问题 #6994 @windtalker
  - 修复 TiFlash 无法识别生成列的问题 #6801 @guo-shaoge
  - 修复了 Decimal 除法在某些情况下最后一位未进位的问题 #7022 @LittleFall
- Tools
  - TiCDC
    * 修复同步数据时由于 UPDATE 和 INSERT 语句乱序可能导致 Duplicate entry 错误的问题 #8597 @sdojjy
    * 修复由于 PD 和 TiCDC 之间的网络隔离引起 TiCDC 程序异常退出的问题 #8562 @overvenus
    * 修复下游为 TiDB 或 MySQL 时，无主键且非空唯一索引所在列指定了 CHARACTER SET 同步时可能会出现数据不一致的问题 #8420 @zhaoxinyu
    * 修复 db sorter 使用内存时未受 cgroup memory limit 限制的问题 #8588 @amyangfei
    * 优化 cdc cli 在遇到非法输入时的错误提示 #7903 @CharlesCheung96
    * 修复 redo log 容忍 S3 存储故障的时间过短的问题 #8089 @CharlesCheung96
    * 修复在 PD 异常时，暂停一个 changefeed 会错误设置状态的问题 #8330 @sdojjy
  - TiDB Lightning
    * 修复冲突处理逻辑 (duplicate-resolution) 可能导致 checksum 不一致的问题 #40657 @gozssky
    * 修复 TiDB Lightning 在 split-region 阶段发生 panic 的问题 #40934 @lance6716
    * 修复了在使用 Local Backend 模式导入数据时，当导入目标表的复合主键中存在 auto_random 列，且源数据中没有指定该列的值时，相关列没有自动生成数据的问题 #41454 @D3Hunter
    * 修复在并行导入时，当除最后一个 TiDB Lightning 实例外的其他实例都遇到本地重复记录时，TiDB Lightning 可能会错误地跳过冲突处理的问题 #40923 @lichunzhu

16.11.3 TiDB 6.1.5 Release Notes

发版日期：2023 年 2 月 28 日

TiDB 版本：6.1.5

试用链接：快速体验 | 生产部署 | 下载离线包

16.11.3.1 兼容性变更

- 自 2023 年 2 月 20 日起，新发布的 TiDB 和 TiDB Dashboard 版本（包含 6.1.5），默认关闭遥测功能，即默认不再收集使用情况信息分享给 PingCAP。如果升级至这些版本前使用默认的遥测配置，则升级后遥测功能处于关闭状态。具体的版本可参考TiDB 版本发布时间线。
  - 系统变量tidb_enable_telemetry 默认值由 ON 修改为 OFF。
  - TiDB 配置项enable-telemetry 默认值由 true 改为 false。
  - PD 配置项enable-telemetry 默认值由 true 改为 false。

- 从 v1.11.3 起，新部署的 TiUP 默认关闭遥测功能，即默认不再收集使用情况信息。如果从 v1.11.3 之前的 TiUP 版本升级至 v1.11.3 或更高 TiUP 版本，遥测保持升级前的开启或关闭状态。

16.11.3.2 提升改进

- TiDB
  - 允许 AUTO_RANDOM 列作为聚簇复合索引中的第一列 #38572 @tangenta

16.11.3.3 Bug 修复

- TiDB
  - 修复 data race 可能导致 TiDB 重启的问题 #27725 @XuHuaiyu
  - 修复当使用 Read Committed 隔离级别时 UPDATE 语句可能读不到最新数据的问题 #41581 @cfzjywxk
- PD
  - 修复调用 ReportMinResolvedTS 过于频繁导致 PD OOM 的问题 #5965 @HundunDM
- Tools
  - TiCDC
    * 修复当同步的延迟过大时 apply redo log 可能会出现 OOM 的问题 #8085 @CharlesCheung96
    * 修复当开启 redo log 写 meta 时性能下降的问题 #8074 @CharlesCheung96
  - TiDB Data Migration (DM)
    * 修复 binlog-schema delete 命令执行失败的问题 #7373 @liumengya94
    * 修复当最后一个 binlog 是被 skip 的 DDL 时，checkpoint 不推进的问题 #8175 @D3Hunter

16.11.4 TiDB 6.1.4 Release Notes

发版日期：2023 年 2 月 8 日

TiDB 版本：6.1.4

试用链接：快速体验 | 生产部署 | 下载离线包

16.11.4.1 兼容性变更

- TiDB
  - 由于可能存在正确性问题，分区表目前不再支持修改列类型 #40620 @mjonss

16.11.4.2 提升改进

- TiFlash
  - 减少了高强度更新负载下的 TiFlash 实例的 IOPS 和写放大倍数，IOPS 最高减少 95%，写放大最高减少 65% #6460 @flowbehappy
- Tools
  - TiCDC
    * 增加了 DML batch 操作功能，提升了批量生成 SQL 语句场景下的吞吐 #7653 @asddongmen
    * 支持将 redo log 存储至兼容 GCS 或 Azure 协议的对象存储 #7987 @CharlesCheung96
  - TiDB Lightning
    * 将 precheck 检查项 `clusterResourceCheckItem` 和 `emptyRegionCheckItem` 的严重级别从 `Critical` 改为 `Warning` #37654 @niubell

16.11.4.3 Bug 修复

- TiDB
  - 修复了创建表时列的默认值和列类型不一致且没有自动修正的问题 #34881 @Lloyd-Pottiger @mjonss
  - 修复 `LazyTxn.LockKeys` 函数中的数据争用问题 #40355 @HuSharp
  - 修复 `INSERT` 或 `REPLACE` 语句在长会话连接中执行可能造成 Panic 的问题 #40351 @fanrenhoo
  - 修复使用 “Cursor Read” 方式读取数据时可能因为 GC 而报错的问题 #39447 @zyguan
  - 修复`pessimistic-auto-commit` 配置项对 “Point Get” 查询不生效的问题 #39928 @zyguan
  - 修复查询 `INFORMATION_SCHEMA.TIKV_REGION_STATUS` 返回不正确结果的问题 #37436 @zimulala
  - 修复使用 `IN` 或 `NOT IN` 的子查询在某些情况下会报错 `Can't find column` 的问题 #37032 @AilinKid @lance6716
- PD
  - 修复 PD 可能会非预期地向 Region 添加多个 Learner 的问题 #5786 @HunDunDM
- TiKV
  - 修复 Gitpod 环境中存在多个 `cgroup` 和 `mountinfo` 时 TiDB 启动异常的问题 #13660 @tabokie
  - 修复 tikv-ctl 执行 `reset-to-version` 命令时被终止的问题 #13829 @tabokie
  - 修复误报 `PessimisticLockNotFound` 的问题 #13425 @sticnarf
  - 修复单次写入超过 2 GiB 时 TiKV panic 的问题 #13848 @YuJuncen
  - 修复 TiDB 中事务在执行悲观 DML 失败后，再执行其他 DML 时，如果 TiDB 和 TiKV 之间存在网络故障，可能会造成数据不一致的问题 #14038 @MyonKeminta
  - 修复当未启用 new collation 时 `LIKE` 操作符中的 _ 无法匹配非 ASCII 字符的问题 #13769 @YangKeao
- TiFlash
  - 修复小概率情况下出现的 TiFlash 全局锁被长时间阻塞的问题 #6418 @SeaRise
  - 修复高更新负载可能导致 OOM 的问题 #6407 @JaySon-Huang

- Tools
  - Backup & Restore (BR)
    * 修复在某些情况下因无法获取 Region 大小导致恢复失败的问题 #36053 @YuJuncen
    * 修复使用 br debug 命令解析 backupmeta 文件导致的 panic 的问题 #40878 @MoCuishle28
  - TiCDC
    * 修复在同步大量表时 checkpoint 不推进问题 #8004 @asddongmen
    * 修复不能通过配置文件修改 transaction_atomicity 和 protocol 参数的问题 #7935 @CharlesCheung96
    * 修复当 TiFlash 的版本高于 TiCDC 时，TiCDC 会误报错的问题 #7744 @overvenus
    * 修复同步大事务时会出现 OOM 的问题 #7913 @overvenus
    * 修复没有配置大事务拆分时，同步数据超过 context deadline 的问题 #7982 @hi-rustin
    * 修复 changefeed query 的输出中 sasl-password 显示为明文的问题 #7182 @dveeden
    * 修复用户快速删除、创建同名同步任务可能导致的数据丢失问题 #7657 @overvenus
  - TiDB Data Migration (DM)
    * 修复当 SHOW GRANTS 结果中的下游数据库名包含通配符 * 时 precheck 报错的问题 #7645 @lance6716
    * 修复当 binlog 中 query event 为 COMMIT 时 DM 打印过多日志的问题 #7525 @liumengya94
    * 修复当 SSL 参数仅指定 ssl-ca 时 DM 任务无法启动的问题 #7941 @liumengya94
    * 修复当在某个表上同时指定 UPDATE 和非 UPDATE 类型的表达式过滤规则 expression-filter 时，所有 UPDATE 操作被跳过的问题 #7831 @lance6716
    * 修复当某个表上仅指定 update-old-value-expr 或 update-new-value-expr 时，过滤规则不生效或 DM 发生 panic 的问题 #7774 @lance6716
  - TiDB Lightning
    * 修复 TiDB Lightning 导入巨大数据源文件时的内存泄漏问题 #39331 @dsdashun
    * 修复 precheck 检查项有时无法监测到之前的导入失败遗留的脏数据的问题 #39477 @dsdashun

16.11.5 TiDB 6.1.3 Release Notes

发版日期：2022 年 12 月 5 日

TiDB 版本：6.1.3

试用链接：快速体验 | 生产部署 | 下载离线包

16.11.5.1 兼容性变更

- Tools
  - TiCDC
    * 将transaction-atomicity 的默认值从 table 修改为 none，降低同步延迟，减少系统出现 OOM 的风险。同时，系统只拆分少量的事务（即超过 1024 行的事务），而不再拆分所有事务 #7505 #5231 @asddongmen

16.11.5.2 提升改进

- PD
  - 优化锁的粒度以减少锁争用，提升高并发下心跳的处理能力 #5586 @rleungx
- Tools
  - TiCDC
    * 默认情况下关闭 safeMode 并开启大事务拆分功能，提升同步的稳定性 #7505 @asddongmen
    * 提升 Kafka 相关协议的编码性能 #7540 #7532 #7543 @sdojjy @3AceShowHand
- 其他
  - 为了提升 TiDB 稳定性，缓解 OOM 问题，TiDB 的 Go 编译器版本从 go1.18 升级到了 go1.19。通过设置 Go 环境变量 `GOMEMLIMIT`，可以将 TiDB 的内存使用维持在预定的水位线以下，缓解大部分 OOM 问题。更多信息，详见设置环境变量 `GOMEMLIMIT` 缓解 OOM 问题。

16.11.5.3 Bug 修复

- TiDB
  - 修复 `mysql.tables_priv` 表中 `grantor` 字段缺失的问题 #38293 @CbcWestwolf
  - 修复错误下推的条件被 Join Reorder 丢弃后导致查询结果错误的问题 #38736 @winoros
  - 修复使用 `get_lock()` 获得的锁无法持续 10 分钟以上的问题 #38706 @tangenta
  - 修复自增列不能和检查约束一起使用的问题 #38894 @YangKeao
  - 修复了 gRPC 日志导出到错误文件的问题 #38941 @xhebox
  - 修复当表被截断或删除时 TiFlash 同步状态未从 etcd 中删除的问题 #37168 @CalvinNeo
  - 修复通过数据源名称注入可读取任意文件的问题 (CVE-2022-3023) #38541 @lance6716
  - 修复函数 `str_to_date` 在 `NO_ZERO_DATE` SQL 模式下返回结果不正确的问题 #39146 @mengxin9014
  - 修复后台统计信息任务可能崩溃的问题 #35421 @lilinghai
  - 修复部分场景非唯一二级索引被误加悲观锁的问题 #36235 @ekexium
- PD
  - 修复 Stream 超时问题，提高 Leader 切换的速度 #5207 @CabinfeverB
- TiKV
  - 修复获取 Snapshot 时 Lease 过期引发的异常竞争问题 #13553 @SpadeA-Tang
- TiFlash
  - 修复逻辑运算符在 `UInt8` 类型下查询结果出错的问题 #6127 @xzhangxian1008
  - 修复 `CAST(value AS datetime)` 输入数据无法转成 `DATETIME` 时会导致 TiFlash sys CPU 异常高的问题 #5097 @xzhangxian1008
  - 修复高压力写入可能产生太多 delta 层小文件的问题 #6361 @lidezhu
  - 修复 TiFlash 重启后 delta 层的小文件无法合并 (compact) 的问题 #6159 @lidezhu

- Tools
  - Backup & Restore (BR)
    * 修复数据库或数据表中使用旧的排序规则框架时数据恢复失败的问题 #39150 @MoCuishle28
  - TiCDC
    * 修复在执行 DDL 后，暂停然后恢复 changefeed 会导致数据丢失的问题 #7682 @asddongmen
    * 修复下游网络发生异常时 sink 模块不能正确处理导致卡住的问题 #7706 @hicqu
  - TiDB Data Migration (DM)
    * 修复当 `collation_compatible` 设置为 `"strict"` 时，DM 可能生成有重复排序规则的 SQL 语句的问题 #6832 @lance6716
    * 修复 DM 可能由于 `Unknown placement policy` 错误导致任务停止的问题 #7493 @lance6716
    * 修复在某些场景下 relay log 文件会从上游重新拉取的问题 #7525 @liumengya94
    * 修复当 DM worker 即将退出时新 worker 调度过快导致数据被重复同步的问题 #7658 @GMHDBJD

16.11.6 TiDB 6.1.2 Release Notes

发版日期：2022 年 10 月 24 日

TiDB 版本：6.1.2

试用链接：快速体验 | 生产部署 | 下载离线包

16.11.6.1 提升改进

- TiDB
  - 允许在一张表上同时设置数据放置规则和 TiFlash 副本 #37171 @lcwangchao
- TiKV
  - 支持配置 `unreachable_backoff` 避免 Raftstore 发现某个 Peer 无法连接时广播过多消息 #13054 @5kbpers
  - 支持将 RocksDB write stall 参数设置为比 flow control 流控阈值更小的值 #13467 @tabokie
- Tools
  - TiDB Lightning
    * 增加 checksum 阶段可重试错误 (retryable error)，提升鲁棒性 #37690 @D3Hunter
  - TiCDC
    * 采用批处理 resolved ts 的模式，提升 region worker 的性能 #7078 @sdojjy

16.11.6.2 Bug 修复

- TiDB
  - 修复数据库级别的权限清理不正确的问题 #38363 @dveeden
  - 修复 SHOW CREATE PLACEMENT POLICY 输出结果不正确的问题 #37526 @xhebox
  - 修复了当某台 PD 宕机时，由于没有重试其他 PD 节点，导致查询表 INFORMATION_SCHEMA.TIKV_REGION_STATUS 时请求失败的问题 #35708 @tangenta
  - 修复 UNION 运算符可能会非预期地返回空结果的问题 #36903 @tiancaiamao
  - 修复在 TiFlash 中为分区表开启动态模式时结果出错的问题 #37254 @wshwsh12
  - 修复 Region 合并情况下 Region cache 没有及时被清理的问题 #37141 @sticnarf
  - 修复 KV 客户端发送不必要 ping 消息的问题 #36861 @jackysp
  - 修复带 DML 算子的 EXPLAIN ANALYZE 语句可能在事务提交完成前返回结果的问题 #37373 @cfzjywxk
  - 修复当 ORDER BY 子句里包含关联子查询时与 GROUP CONCAT 一起执行可能会导致出错的问题 #18216 @winoros
  - 修复 UPDATE 语句中带公共表表达式 (CTE) 的情况下会报 Can't find column 的问题 #35758 @AilinKid
  - 修复某些情况下，EXECUTE 语句可能抛出非预期异常的问题 #37187 @Reminiscent
- TiKV
  - 修复因引入跨 Region 批量 snapshot 导致 snapshot 数据不完整的问题 #13553 @SpadeA-Tang
  - 修复开启流量控制且显式设置 level0_slowdown_trigger 时出现 QPS 下降的问题 #11424 @Connor1996
  - 修复 Web 身份提供程序 (web identity provider) 报错并失效后，自动恢复为默认提供程序 (default provider) 时出现权限拒绝的问题 #13122 @3pointer
  - 修复当有一个 TiKV 实例出现网络隔离时，一段时间内服务不可用问题 #12966 @cosven
- PD
  - 修复 Region tree 统计可能不准确的问题 #5318 @rleungx
  - 修复 PD 可能没创建 TiFlash Learner 副本的问题 #5401 @HunDunDM
  - 修复 PD 无法正确处理 dashboard 代理请求的问题 #5321 @HunDunDM
  - 修复不健康的 Region 可能导致 PD panic 的问题 #5491 @nolouch
- TiFlash
  - 修复在大批量写入之后，I/O Limiter 可能错误地限制了读请求的 I/O 吞吐量，从而降低查询性能的问题 #5801 @JinheLin
  - 修复取消查询时 window function 可能会导致 TiFlash 崩溃的问题 #5814 @SeaRise
  - 修复使用包含 NULL 值的列创建主键时导致崩溃的问题 #5859 @JaySon-Huang
- Tools
  - TiDB Lightning
    * 修复非法 metric 数值可能导致 TiDB Lightning panic 的问题 #37338 @D3Hunter
  - TiDB Data Migration (DM)
    * 修复某些情况下，进入 sync unit 的任务中断导致的上游表结构信息丢失问题 #7159 @lance6716

    * 通过在保存 checkpoint 时拆分 SQL 语句解决大事务问题 #5010 @lance6716
    * 解决 Pre-Check 阶段 INFORMATION_SCHEMA 表需要 SELECT 权限的问题 #7317 @lance6716
    * 修复开启 fast/full validator 时 DM-worker 可能触发死锁问题 #7241 @buchuitoudegou
    * 修复 DM 报错 Specified key was too long 的问题 #5315 @lance6716
    * 修复数据同步过程中，latin1 字符集数据可能损坏的问题 #7028 @lance6716
  - TiCDC
    * 修复 cdc server 在尚未启动成功前收到 HTTP 请求导致 panic 的问题 #6838 @asddongmen
    * 修复升级时日志数量过多的问题 #7235 @hi-rustin
    * 修复 redo log 中错误清理非当前 changefeed 日志文件的问题 #6413 @hi-rustin
    * 修复在一个 etcd 事务中提交太多数据导致 TiCDC 服务不可用问题 #7131 @hi-rustin
    * 修复 redo log 中不可重入 DDL 重复执行可能导致的数据不一致性问题 #6927 @hicqu
  - Backup & Restore (BR)
    * 修复在恢复时配置过高的 concurrency 会导致 Region 不均衡的问题 #37549 @3pointer
    * 修复当外部存储的鉴权 Key 中存在某些特殊符号时，会导致备份恢复失败的问题 #37469 @MoCuishle28

16.11.7 TiDB 6.1.1 Release Notes

发版日期：2022 年 9 月 1 日

TiDB 版本：6.1.1

试用链接：快速体验 | 生产部署 | 下载离线包

16.11.7.1 兼容性变更

- TiDB
  - SHOW DATABASES LIKE … 语句不再大小写敏感 #34766 @e1ijah1
  - 将tidb_enable_outer_join_reorder 的默认值由 1 改为 0，即默认关闭 Join Reorder 对外连接的支持。
- Diagnosis
  - 默认关闭持续性能分析 (Continuous Profiling) 特性，以避免开启该特性后 TiFlash 可能会崩溃的问题，详情参见 #5687 @mornyx

16.11.7.2 其他变更

- 在 TiDB-community-toolkit 二进制包中新增了以下内容，详情参见TiDB 离线包。
  - server-{version}-linux-amd64.tar.gz
  - grafana-{version}-linux-amd64.tar.gz
  - alertmanager-{version}-linux-amd64.tar.gz
  - prometheus-{version}-linux-amd64.tar.gz
  - blackbox_exporter-{version}-linux-amd64.tar.gz
  - node_exporter-{version}-linux-amd64.tar.gz
- 针对不同操作系统和 CPU 架构的组合，引入不同级别质量标准的支持，见操作系统及平台要求。

16.11.7.3 提升改进

- TiDB
  - 引入新的优化器提示 `SEMI_JOIN_REWRITE` 改善 `EXISTS` 查询性能 #35323 @winoros
- TiKV
  - 支持通过 gzip 压缩 metrics 响应减少 HTTP body 大小 #12355 @winoros
  - 支持使用 `server.simplify-metrics` 配置项过滤部分 Metrics 采样数据以减少每次请求返回的 Metrics 数据量 #12355 @glorv
  - 支持动态调整 RocksDB 进行 subcompaction 的并发个数 (`rocksdb.max-sub-compactions`) #13145 @ethercflow
- PD
  - 提升 Balance Region 在特定阶段的调度速度 #4990 @bufferflies
- Tools
  - TiDB Lightning
    * 针对 `stale command` 等错误增加自动重试机制，提升导入成功率 #36877 @D3Hunter
  - TiDB Data Migration (DM)
    * 用户可手动设置 Lightning Loader 并发数 #5505 @buchuitoudegou
  - TiCDC
    * 在 changefeed 的配置中增加参数 `transaction-atomicity` 来控制是否拆分大事务，从而大幅减少大事务的延时和内存消耗 #5231 @CharlesCheung96
    * 优化了多 Region 场景下，runtime 上下文切换带来过多性能开销的问题 #5610 @hicqu
    * 优化 MySQL sink，实现自动关闭 safe mode #5611 @overvenus

16.11.7.4 Bug 修复

- TiDB
  - 修复 `INL_HASH_JOIN` 和 `LIMIT` 一起使用时可能会卡住的问题 #35638 @guo-shaoge
  - 修复 TiDB 在执行 `UPDATE` 语句时可能会 panic 的问题 #32311 @Yisaer
  - 修复 TiDB 在执行 `SHOW COLUMNS` 时会发出协处理器请求的问题 #36496 @tangenta
  - 修复执行 `SHOW WARNINGS` 时可能会报 `invalid memory address or nil pointer dereference` 的问题 #31569 @zyguan
  - 修复 Static Partition Prune 模式下带聚合条件的 SQL 语句在表为空时结果错误的问题 #35295 @tiancaiamao
  - 修复执行 Join Reorder 操作时会错误地下推 Outer Join 条件的问题 #37238 @winoros
  - 修复了 CTE 被引用多次时 schema hash code 被错误克隆导致的 `Can't find column ... in schema ↪ ...` 错误 #35404 @AilinKid
  - 修复了某些 Right Outer Join 场景下 Join Reorder 错误导致查询结果错误的问题 #36912 @winoros

  - 修复了执行计划在 EqualAll 的情况下，把 TiFlash 的 `firstrow` 聚合函数的 null flag 设错的问题 #34584 @fixdb
  - 修复了当查询创建了带 `IGNORE_PLAN_CACHE` hint 的 binding 后，无法再使用 Plan Cache 的问题 #34596 @fzzf678
  - 修复了 hash-partition window 和 single-partition window 之间缺少 `EXCHANGE` 算子的问题 #35990 @LittleFall
  - 修复某些情况下分区表无法充分利用索引来扫描数据的问题 #33966 @mjonss
  - 修复了聚合运算下推后为 partial aggregation 设置了错误的默认值导致结果错误的问题 #35295 @tiancaiamao
  - 修复了在某些情况下查询分区表可能返回 `index-out-of-range` 错误的问题 #35181 @mjonss
  - 修复了在查询分区表中如果查询条件中有分区键且两者使用了不同的 `COLLATE` 时会错误的进行分区裁剪的问题 #32749 @mjonss
  - 修复了在开启 TiDB Binlog 时，TiDB 执行 `ALTER SEQUENCE` 会产生错误的元信息版本号，进而导致 Drainer 报错退出的问题 #36276 @AilinKid
  - 修复了在极端情况下，启动 TiDB 可能进入错误状态的问题 #36791 @xhebox
  - 修复了在 TiDB Dashboard 中查询分区表的执行计划时，有可能出现 `UnkownPlanID` 的问题 #35153 @time-and-fate
  - 修复了 `LOAD DATA` 语句中列的列表不生效的问题 #35198 @SpadeA-Tang
  - 修复开启 TiDB Binlog 后插入重复数据导致 data and columnID count not match 错误的问题 #33608 @zyguan
  - 去除 `tidb_gc_life_time` 设置时间检查限制 #35392 @TonsnakeLin
  - 修复空分隔符使用情况下，`LOAD DATA` 出现死循环的问题 #33298 @zyguan
  - 避免向非健康状态的 TiKV 节点发送请求，以提升可用性 #34906 @sticnarf

- TiKV

  - 修复 Raftstore 线程繁忙时，可能会出现 Region 重叠的问题 #13160 @5kbpers
  - 修复 PD Region heartbeat 连接异常中断后未重新连接的问题 #12934 @bufferflies
  - 修复了对空字符串进行类型转换导致 TiKV panic 的问题 #12673 @wshwsh12
  - 修复了 TiKV 和 PD 配置文件中 Region size 不一致的问题 #12518 @5kbpers
  - 修复了启用 Raft Engine 时未清理加密密钥的问题 #12890 @tabokie
  - 修复同时分裂和销毁一个 peer 时可能导致 panic 的问题 #12825 @BusyJay
  - 修复在 Region merge 时 source peer 通过 snapshot 追日志时可能导致 panic 的问题 #12663 @BusyJay
  - 修复了 PD 客户端遇到报错时频繁重连的问题 #12345 @Connor1996
  - 修复了开启 Raft Engine 并发恢复时 TiKV 可能会 panic 的问题 #13123 @tabokie
  - 修复了新创建的 Region Commit Log Duration 较高导致 QPS 下降的问题 #13077 @Connor1996
  - 修复启用 Raft Engine 后特殊情况下 TiKV 会 panic 的问题 #12698 @tabokie
  - 修复无法找到 proc filesystem (procfs) 时警告级别日志过多的问题 #13116 @tabokie
  - 修复 Dashboard 中 Unified Read Pool CPU 表达式错误的问题 #13086 @glorv
  - 修复 Region 较大时，默认`region-split-check-diff` 可能会大于 bucket 大小的问题 #12598 @tonyxuqqi
  - 修复启用 Raft Engine 后，中止 Apply Snapshot 时可能会 panic 的问题 #12470 @tabokie
  - 修复 PD 客户端可能会出现死锁的问题 #13191 @bufferflies #12933 @BurtonQin

- PD

  - 修复当集群中节点的 label 设置异常时，store 上线进度评估不准确的问题 #5234 @rleungx
  - 修复开启 `enable-forwarding` 时 gRPC 处理返回错误不恰当导致 PD panic 的问题 #5373 @bufferflies
  - 修复 `/regions/replicated` 返回状态错误的问题 #5095 @rleungx

- TiFlash
  - 修复在 clustered index 表删除列导致 TiFlash 崩溃的问题 #5154 @hongyunyan
  - 修复 `format` 函数可能会报 `Data truncated` 错误的问题 #4891 @xzhangxian1008
  - 修复存储中残留过期数据且无法删除的问题 #5659 @lidezhu
  - 修复个别场景消耗不必要 CPU 的问题 #5409 @breezewish
  - 修复 TiFlash 无法在使用 IPv6 的集群运行的问题 #5247 @solotzg
  - 修复并行聚合出错时可能导致 TiFlash crash 的问题 #5356 @gengliqi
  - 修复 `MinTSOScheduler` 在查询出错时可能会泄露线程资源问题 #5556 @windtalker
- Tools
  - TiDB Lightning
    * 修复了使用 IPv6 host 时无法连接到 TiDB 的问题 #35880 @D3Hunter
    * 修复 `read index not ready` 问题，增加重试机制 #36566 @D3Hunter
    * 修复服务器模式下日志敏感信息被打印的问题 #36374 @lichunzhu
    * 修复 TiDB Lightning 不支持 Parquet 文件中以斜线 (`/`)、数字、非 ASCII 字符开头的特殊列名的问题 #36980 @D3Hunter
    * 修复极端情况下去重可能会导致 TiDB Lightning panic 的问题 #34163 @ForwardStar
  - TiDB Data Migration (DM)
    * 修复 `txn-entry-size-limit` 在 DM 不生效的问题 #6161 @ForwardStar
    * 修复 `check-task` 命令不能处理特殊编码的问题 #5895 @Ehco1996
    * 修复 `query-status` 内可能存在 data race 的问题 #4811 @lyzx2001
    * 修复 `operate-schema` 显示不一致问题 #5688 @ForwardStar
    * 修复 relay 报错时可能导致 goroutine 泄露问题 #6193 @lance6716
    * 修复 DM Worker 因 DB Conn 获取可能卡住的问题 #3733 @lance6716
    * 修复 DM IPv6 支持问题 #6249 @D3Hunter
  - TiCDC
    * 修复最大兼容版本错误的问题 #6039 @hi-rustin
    * 修复 cdc server 启动未完成接受请求时出现 panic 的问题 #5639 @asddongmen
    * 修复打开 sync-point 时 ddl sink 可能出现 panic 的问题 #4934 @asddongmen
    * 修复打开 sync-point 功能在某些特殊场景下出现卡住 changefeed 的问题 #6827 @hicqu
    * 修复 cdc server 重启时 API 工作不正常的问题 #5837 @asddongmen
    * 修复 black hole sink 场景下出现 data race 问题 #6206 @asddongmen
    * 修复 `enable-old-value = false` 时可能出现的 cdc panic 问题 #6198 @hi-rustin
    * 修复在开启 redo 功能时可能出现数据不一致问题 #6189 #6368 #6277 #6456 #6695 #6764 #6859 @asddongmen
    * 修复了 redo log 的性能问题，采取异步写的方式提升 redo 吞吐 #6011 @CharlesCheung96
    * 修复 MySQL sink 无法连接 IPv6 地址的问题 #6135 @hi-rustin
  - Backup & Restore (BR)
    * 修复了 RawKV 模式下 BR 报 `ErrRestoreTableIDMismatch` 错误的问题 #35279 @3pointer
    * 优化了全量备份数据组织形式，解决大规模集群备份时遇到的 S3 限流问题 #30087 @MoCuishle28
    * 修复日志中统计备份耗时不正确的问题 #35553 @ixuh12
  - Dumpling

    * 修复 GetDSN 方法不支持 IPv6 的问题 #36112 @D3Hunter

- TiDB Binlog

    * 修复 `compressor` 设为 `gzip` 时 Drainer 无法正确发送请求至 Pump 的问题 #1152 @lichunzhu

16.11.8 TiDB 6.1.0 Release Notes

发版日期：2022 年 6 月 13 日

TiDB 版本：6.1.0

试用链接：快速体验 | 生产部署 | 下载离线包

在 6.1.0 版本中，你可以获得以下关键特性：

- List 和 List COLUMNS 分区方式 GA，与 MySQL 5.7 兼容
- TiFlash 分区表动态裁剪 GA
- 支持兼容 MySQL 的用户级别锁管理
- 支持非事务性 DML 语法（目前仅支持 DELETE）
- TiFlash 支持按需触发物理数据整理（Compaction）
- MPP 实现窗口函数框架
- TiCDC 支持将 changelogs 以 Avro 协议输出到 Kafka
- TiCDC 支持在数据复制过程中拆分大事务，能够有效降低大事务带来的复制延迟
- DM 合库合表迁移场景的乐观 DDL 协调模式 GA

16.11.8.1 新功能

16.11.8.1.1 SQL

- List 和 List COLUMNS 分区方式正式 GA，与 MySQL 5.7 兼容。

  用户文档：List 分区，List COLUMNS 分区

- 支持通过 SQL 语句对 TiFlash 副本立即触发物理数据整理 (Compaction)（实验特性）

  在当前 TiFlash 后台自动整理物理数据（Compaction）机制基础上，新增 compact 命令，帮助刷新旧格式数据，提升读写性能。推荐在升级至 v6.1.0 之后，执行该语句以清理数据。此语句是对标准 SQL 语法的扩展，对 MySQL 客户端保持兼容。升级之外场景一般不需要特别关注。

  用户文档，#4145

- 实现窗口函数框架，支持以下分析函数：

    - `RANK()`
    - `DENSE_RANK()`
    - `ROW_NUMBER()`

用户文档，#33072

16.11.8.1.2 可观测性

- 持续性能分析支持 ARM 架构和 TiFlash 组件。

  用户文档

- Grafana 中新增 Performance Overview 面板，提供系统级别的总体性能诊断入口。

  Performance Overview 是 TiDB 监控 Grafana 可视化组件中的一个新增面板，基于数据库时间分析法和颜色优化法，按照自上而下的性能分析方法论对 TiDB 的性能指标做了重新梳理，为 TiDB 用户提供一个系统级别的总体性能诊断入口。通过 Performance Overview 面板，你可以直观地看到整个系统的性能瓶颈在哪里，数量级地缩短了性能诊断时间并降低了性能分析和诊断难度。

  用户文档

16.11.8.1.3 性能

- 支持自定义 Region 大小（实验特性）

  设置更大的 Region 可以有效减少 Region 数量，降低 Region 管理成本，提升集群性能和稳定性。该特性引入 bucket 概念，即将每个 Region 划分为更小的区间 bucket。使用 bucket 作为并发查询单位能够优化 Region 调大时的查询性能，动态调整热点 Region 的大小来保证热点调度效率和负载均衡。该特性目前属于实验特性，不建议在生产环境使用。

  用户文档，#11515

- Raft Engine 存储引擎 GA

  TiDB 从 v6.1.0 开始默认使用引擎 Raft Engine 作为 TiKV 的日志存储引擎。新引擎与 RocksDB 相比，可以减少至多 40% 的 TiKV I/O 写流量和 10% 的 CPU 使用，同时在特定负载下提升 5% 左右前台吞吐，减少 20% 长尾延迟。

  用户文档，#95

- 支持 join order hint 语法

  - LEADING hint：提示优化器使用指定的顺序作为连接前缀，好的连接前缀顺序可以在连接初期快速地降低数据量，提升查询性能。
  - STRAIGHT_JOIN hint：提示优化器按照表在 FROM 子句中的出现顺序进行连接。

  该特性提供更多的手段帮助用户固定连接顺序，合理运用 hint 语法，可以有效提升 SQL 性能和集群稳定性。

用户文档：LEADING，STRAIGHT_JOIN，#29932

- TiFlash 新增对以下函数的支持：

  - FROM_DAYS
  - TO_DAYS
  - TO_SECONDS
  - WEEKOFYEAR

用户文档，#4679，#4678，#4677

- 支持分区表动态裁剪

  支持分区表动态裁剪功能，以提升数据分析场景下的性能。v6.0.0 以前版本的用户升级完成后建议及时手动刷新既存分区表的统计信息，以达到最好的性能表现（全新安装，或在 v6.1.0 升级完成后新创建的分区表无需此动作）。

  用户文档：MPP 模式访问分区表，动态裁剪模式，#3873

16.11.8.1.4 稳定性

- SST 故障自动恢复

  当 RocksDB 后台检测到故障 SST 文件后，TiKV 会尝试通过调度所涉及的故障 Peer，并利用其它副本恢复该节点数据，用户可以通过参数 `background-error-recovery-window` 设置允许的最长恢复时间。如果恢复操作未能在设置的时间窗口内完成，TiKV 将会崩溃。该特性对于可恢复存储故障进行自动检测和恢复，提升了集群稳定性。

  用户文档，#10578

- 支持非事务性 DML 语法

  在大批量的数据处理场景，单一大事务 SQL 处理有可能对集群稳定性和性能造成影响。TiDB 从 v6.1.0 支持对 DELETE 语句提供拆分后批量处理的语法格式，拆分后的语句将牺牲事务原子性和隔离性，但是对于集群的稳定性有很大提升，详细语法请参考BATCH。

  用户文档

- TiDB 支持设置最大 GC 等待时间

  TiDB 的事务的实现采用了 MVCC（多版本并发控制）机制，当新写入的数据覆盖旧的数据时，旧的数据不会被替换掉，而是与新写入的数据同时保留，并通过 Garbage Collection (GC) 的任务定期清理不再需要的旧数据。定期 GC 清理有助于回收存储空间，提升集群性能和稳定性，TiDB 默认每 10 分钟自动执行 GC 清理。为了保证长时间执行的事务可以访问到对应的历史数据，当有执行中的事务时，GC 清理会被推迟。为了保证 GC 清理不会被无限制推迟，TiDB 从 v6.1.0 引入了系统变量`tidb_gc_max_wait_time` 来控制 GC 推迟的最大等待时间，超过该参数允许的时间，GC 将会被强制执行。该参数默认为 24 小时。通过该系统变量，用户可以有效地控制 GC 等待时长与长事务的关系，提升集群的稳定性。

  用户文档

- 支持设置统计信息自动采集任务的最长执行时间

  通过采集统计信息，数据库可以有效掌握数据的分布情况，从而生成合理的执行计划，提升 SQL 的执行效率。TiDB 在后台会定期对频繁变更的数据对象进行统计信息采集，但在业务高峰期时进行统计信息采集可能会对集群资源造成挤压，影响业务的稳定运行。TiDB 从 v6.1.0 开始提供系统变量`tidb_max_auto_analyze_time` 用来控制后台统计信息采集的最长执行时间，默认为 12 小时。当业务没有遇到资源瓶颈的情况时，建议不要修改该参数，确保数据对象的统计信息及时采集。但是当业务压力大，资源不足的时候，可以通过该变量减少统计信息采集的时长，避免统计信息采集对核心业务造成资源争抢。

  用户文档

16.11.8.1.5 易用性

- 支持集群在 Region 多副本丢失状态下在线一键式恢复数据

  在 TiDB v6.1.0 之前，当出现多个 TiKV 因物理机故障导致 Region 的多副本丢失时，用户需要停机所有 TiKV，使用 TiKV Control 逐一对 TiKV 进行恢复。从 TiDB v6.1.0 起，该过程被完全自动化且不需要停机，恢复过程中不影响其他正常的在线业务。通过 PD Control 触发在线有损恢复数据，大幅简化了恢复步骤，缩短了恢复所需时间，提供了更加友好的恢复摘要信息。

  用户文档，#10483

- 历史统计信息采集持久化

  支持通过 `SHOW ANALYZE STATUS` 语句查询集群级别的统计信息收集任务。在 TiDB v6.1.0 之前，`SHOW ANALYZE STATUS` 语句仅显示实例级别的统计信息收集任务，且 TiDB 重启后历史任务记录会被清空，因此用户无法查看历史统计信息的采集时间和相关细节。从 TiDB v6.1.0 起，该信息被持久化保存，集群重启后依然可查询，可以为统计信息异常引起的查询性能问题提供排查依据。

  用户文档

- TiDB/TiKV/TiFlash 参数支持在线修改

在 v6.1.0 之前的版本中，配置变更后，必须重启 TiDB 集群，配置才会生效，这对在线业务会造成一定的影响。TiDB v6.1.0 引入了在线修改配置功能，参数修改后，无需重启，即可生效。具体优化如下：

```
* TiDB 将部分配置项转化为系统变量，所有变量支持在线变更，并支持持久化。请注意，转化后，
    ↪ 原有配置项将被废弃。详细变更列表请查看[配置文件参数](#配置文件参数)。
* TiKV 支持部分参数在线变更。详细变更列表请查看[其他](#其他)。
* TiFlash 配置项 `max_threads` 转化为系统变量 `tidb_max_tiflash_threads`，
    ↪ 从而支持配置项在线变更和持久化。转化后，原有配置项不会废弃。
```

对于从 v6.1.0 之前版本升级到 v6.1.0 的集群（包括滚动升级和停机升级），请注意：

```
* 若升级前集群指定的配置文件中，存在已经配置的项，则升级过程中 TiDB
    ↪ 将会将配置项的值自动更新为对应系统变量的值，以保证升级后，
    ↪ 系统的行为不会因为参数的优化发生变化。
* 上述自动更新仅在升级过程中发生一次，升级完成之后，被废弃的配置项不再有任何效果。
```

通过本次优化，客户可以在线修改参数生效并持久化，方便客户进行日常运维，避免重启生效对在线业务造成的影响。

用户文档

- 支持全局 kill 查询或连接

  支持通过 `enable-global-kill` 配置项（默认开启）设置全局 kill 开关。

  在 TiDB v6.1.0 之前，当某个特定操作占用大量资源引发集群稳定性问题时，你需要先登陆到对应的 TiDB 节点，然后运行 `kill [TiDB] id` 命令终止对应的连接及操作。在 TiDB 节点多的情况下，这种方式使用不便，并且容易误操作。从 v6.1.0 起，当开启 `enable-global-kill` 配置项时，你可以在任意 TiDB 节点运行 kill 命令终止指定的连接及操作，而无需担心客户端和 TiDB 中间有代理时错误地终止其他查询或会话。目前 TiDB 暂时不支持用 Ctrl+C 终止查询或会话。

  用户文档，#8854

- TiKV API V2（实验特性）

  在 v6.1.0 之前，TiKV 作为 Raw Key Value 存储时，由于仅存储了客户端传入的原始数据，因此只提供基本的 Key Value 读写能力。

  TiKV API V2 提供了新的 Raw Key Value 存储格式与访问接口，包括：

  - 数据以 MVCC 方式存储，并记录了数据的变更时间戳。这个特性将为实现 Change Data Capture、增量备份与恢复等打下基础。
  - 数据根据不同的使用方式划分范围，支持单一集群 TiDB、事务 KV、RawKV 应用共存。

由于底层存储格式发生了重大变化，启用 API V2 后，不能将 TiKV 集群回退到 v6.1.0 之前的版本，否则可能导致数据损坏。

```
[用户文档](#api-version-从-v610-版本开始引入)，[#11745](https://github.com/tikv/tikv/issues
    ↪ /11745)
```

16.11.8.1.6 MySQL 兼容性

- 支持兼容 MySQL 的用户级别锁管理

  用户级别锁是 MySQL 通过内置函数提供的用户命名锁管理系统。它们可以提供锁阻塞、等待、等锁管理能力。用户级别锁在 ORM 框架中也有较为广泛的应用，例如 Rails、Elixir 和 Ecto 等。TiDB 从 v6.1.0 版本开始支持兼容 MySQL 的用户级别锁管理，支持 GET_LOCK、RELEASE_LOCK、RELEASE_ALL_LOCKS 函数。

  用户文档，#14994

16.11.8.1.7 数据迁移

- DM 合库合表迁移场景的乐观 DDL 协调模式 GA

  对于合库合表迁移任务，DM 在现有乐观 DDL 协调策略的基础上增加了大量场景测试，足以覆盖 90% 日常使用场景。相比悲观协调策略，乐观协调在使用上更为简单、高效，在仔细阅读注意事项后可优先使用。

  用户文档

- DM 的 WebUI 支持根据指定参数条件启动任务

  开始一个迁移任务时，允许指定“开始时间”和“safe-mode 持续时间”。这在创建具有大量 source 的增量迁移任务时尤其有用，无需再为每个 source 精确指定 binlog 起始同步位置。

  用户文档，#5442

16.11.8.1.8 数据共享订阅

- 支持与更丰富的第三方数据生态系统进行数据共享

  - TiCDC 支持将 TiDB 数据库的增量数据以 Avro 格式发送到 Kafka，通过 Confluent 与 KSQL、Snowflake 等第三方系统进行数据共享。

    用户文档，#5338

  - TiCDC 支持将 TiDB 数据库的增量数据按表分发到不同的 Kafka Topic 中，结合 Canal-json 格式可以将数据直接与 Flink 共享。

    用户文档，#4423
  - TiCDC 支持 SASL GSSAPI 认证类型。增加了使用 Kafka 的 SASL 认证示例。

    用户文档，#4423

- TiCDC 支持同步使用 GBK 编码的上游表。

  用户文档，#4806

16.11.8.2 兼容性变更

16.11.8.2.1 系统变量

变量名	修改类型	描述
tidb_enable_list_partition ↪	修改	默认值从 OFF 改为 ON。
tidb_mem_quota_query ↪	修改	增加 GLOBAL 作用域，变量值可以持久化到集群。
tidb_query_log_max_len ↪	修改	变量作用域由 INSTANCE 修改为 GLOBAL，变量值可以持久化到集群。取值范围修改为 [0, ↪ 1073741824] ↪ 。

变量名	修改类型	描述
require_secure_transport	新增	由 TiDB 配置项 require - secure - transport 转化而来。
tidb_committer_concurrency	新增	由 TiDB 配置项 committer - concurrency 转化而来。
tidb_enable_auto_analyze	新增	由 TiDB 配置项 run-auto - analyze 转化而来。
tidb_enable_new_only_full_group_by_check	新增	控制 TiDB 执行 ONLY_FULL_GROUP_BY 检查时的行为。

变量名	修改类型	描述
tidb_enable_outer_join_reorder ↪	新增	控制 TiDB 的 Join Reorder 算法支持 Outer Join，默认开启。对于从旧版本升级上来的集群，该变量的默认值也会是 TRUE。
tidb_enable_prepared_plan_cache ↪	新增	由 TiDB 配置项 prepared ↪ - ↪ plan ↪ - ↪ cache ↪ . ↪ enabled ↪ 转化而来。
tidb_gc_max_wait_time ↪	新增	用于指定活跃事务阻碍 GC safe point 推进的最大时间。

变量名	修改类型	描述
tidb_max_auto_analyze_time ↪	新增	用于指定自动 ANALYZE 的最大执行时间。
tidb_max_tiflash_threads ↪	新增	由 TiFlash 配置项 max_threads ↪ 转化而来，表示 TiFlash 中 request 执行的最大并发度。
tidb_mem_oom_action ↪	新增	由 TiDB 配置项 oom-action ↪ 转化而来。

变量名	修改类型	描述
tidb_mem_quota_analyze	新增	控制 TiDB 更新统计信息时总的内存占用，包括用户执行的 ANALYZE TABLE 和 TiDB 后台自动执行的统计信息更新任务。
tidb_nontransactional_ignore_error	新增	设置是否在非事务语句中立刻返回错误。
tidb_prepared_plan_cache_memory_guard_ratio	新增	由 TiDB 配置项 prepared-plan-cache.memory-guard-ratio 转化而来。

变量名	修改类型	描述
tidb_prepared_plan_cache_size ↪	新增	由 TiDB 配置项 prepared ↪ - ↪ plan ↪ - ↪ cache ↪ . ↪ capacity ↪ 转化而来。
tidb_stats_cache_mem_quota ↪	新增	控制 TiDB 内部统计信息缓存使用内存的上限。

16.11.8.2.2 配置文件参数

配置文件	配置项	修改类型	描述
TiDB	committer ↪ - ↪ concurrency ↪	删除	转化为系统变量 tidb_committer_concurrency ↪ 。该配置项不再生效，如需修改，需修改对应的系统变量。

配置文件	配置项	修改类型	描述
TiDB	lower-case-table-names	删除	TiDB 目前只支持 lower_case_table_name =2，如果升级前设置了其他值，升级到 v6.1.0 后该值会丢失。
TiDB	mem-quota-query	删除	转化为系统变量 tidb_mem_quota_query 。该配置项不再生效，如需修改，需修改对应的系统变量。
TiDB	oom-action	删除	转化为系统变量 tidb_mem_oom_action 。该配置项不再生效，如需修改，需修改对应的系统变量。

配置文件	配置项	修改类型	描述
TiDB	prepared-plan-cache.capacity	删除	转化为系统变量 tidb_prepared_plan_cache_size。该配置项不再生效，如需修改，需修改对应的系统变量。
TiDB	prepared-plan-cache.enabled	删除	转化为系统变量 tidb_enable_prepared_plan_cache。该配置项不再生效，如需修改，需修改对应的系统变量。
TiDB	query-log.max-len	删除	转化为系统变量 tidb_query_log_max_len。该配置项不再生效，如需修改，需修改对应的系统变量。

配置文件	配置项	修改类型	描述
TiDB	require-secure-transport	删除	转化为系统变量 require_secure_transport。该配置项不再生效，如需修改，需修改对应的系统变量。
TiDB	run-auto-analyze	删除	转化为系统变量 tidb_enable_auto_analyze。该配置项不再生效，如需修改，需修改对应的系统变量。

配置文件	配置项	修改类型	描述
TiDB	enable-global-kill	新增	当该配置项值默认为 true，KILL 语句和 KILL TIDB 语句均能跨节点终止查询或连接，无需担心错误地终止其他查询或连接。
TiDB	enable-stats-cache-mem-quota	新增	控制 TiDB 是否开启统计信息缓存的内存上限。
TiKV	raft-engine.enable	修改	默认值从 false 修改为 true。

配置文件	配置项	修改类型	描述
TiKV	region-max-keys	修改	默认值从 1440000 修改为 region-split-keys / 2 * 3。
TiKV	region-max-size	修改	默认值从 144 MB 修改为 region-split-size / 2 * 3。
TiKV	coprocessor.enable-region-bucket	新增	是否将 Region 划分为更小的区间 bucket。
TiKV	coprocessor.region-bucket-size	新增	设置 enable-region-bucket 启用时 bucket 的预期大小。

配置文件	配置项	修改类型	描述
TiKV	causal-ts.renew-batch-min-size	新增	时间戳缓存的最小数量。
TiKV	causal-ts.renew-interval	新增	刷新本地缓存时间戳的周期。
TiKV	max-snapshot-file-raw-size	新增	当 snapshot 文件大于该配置项指定的大小时，snapshot 文件会被切割为多个文件。
TiKV	raft-engine.memory-limit	新增	指定 Raft Engine 使用内存的上限。

配置文件	配置项	修改类型	描述
TiKV	storage ↪ . ↪ background ↪ - ↪ error ↪ - ↪ recovery ↪ - ↪ window ↪	新增	RocksDB 检测到可恢复的后台错误后，所允许的最长恢复时间。
TiKV	storage ↪ . ↪ api ↪ - ↪ version ↪	新增	TiKV 作为 Raw Key Value 存储数据时使用的存储格式与接口版本。
PD	schedule ↪ . ↪ max ↪ - ↪ store ↪ - ↪ preparing ↪ - ↪ time ↪	新增	控制 store 上线阶段的最长等待时间。
TiCDC	enable ↪ - ↪ tls ↪	新增	控制是否使用 TLS 连接 Kafka。

配置文件	配置项	修改类型	描述
TiCDC	sasl-↪ gssapi↪ -↪ user↪ sasl↪ -↪ gssapi↪ -↪ password↪ sasl↪ -↪ gssapi↪ -↪ auth↪ -↪ type↪ sasl↪ -↪ gssapi↪ -↪ service↪ -↪ name↪ sasl↪ -↪ gssapi↪ -↪ realm↪ sasl↪ -↪ gssapi↪ -↪ key↪ -↪ tab↪ -↪ path↪ sasl↪ -↪ gssapi↪ -↪ kerberos↪ -↪ config↪ - 3811↪ path↪	新增	支持 Kafka SASL/GSS-API 认证所需要的参数。详情见Sink URI 配置 kafka。

配置文件	配置项	修改类型	描述
TiCDC	avro-decimal-handling-mode	新增	控制 Avro 格式的输出细节。
TiCDC	dispatchers.topic	新增	控制 TiCDC 将增量数据分发到不同 Kafka Topic 的策略
TiCDC	dispatchers.partition	新增	dispatchers.partition 是原 dispatchers.dispatcher 配置项的别名，用于控制增量数据的 Kafka Partition 分发策略。

配置文件	配置项	修改类型	描述
TiCDC	schema-registry	新增	用于指定存储 Avro Schema 的 Schema Registry Endpoint。
DM	dmctl start-relay 命令中的 worker 参数	删除	不推荐使用的方式，将通过更为简单的实现替代。
DM	source 配置中的 relay-dir	删除	由 worker 配置文件中的同名配置项替代。
DM	task 配置中的 is-sharding	删除	由 shard-mode 配置项替代。
DM	task 配置中的 auto-fix-gtid	删除	该配置已在 5.x 版本废弃，v6.1.0 正式移除。

配置文件	配置项	修改类型	描述
DM	source 配置中的 meta-dir、charset	删除	该配置已在 5.x 版本废弃，v6.1.0 正式移除。

16.11.8.2.3 其他

- 默认开启 Prepared Plan Cache

  在新集群默认打开 Prepared Plan Cache 的开关，对 Prepare / Execute 请求的执行计划进行缓存，以便在后续执行时跳过查询计划优化这个步骤，获得性能上的提升。升级上来的集群会继承配置文件内的配置。新集群会使用新的默认值，即默认打开该特性，并且每个 Session 最大缓存计划数为 100（capacity=100）。开启后会有一定的内存消耗，详情见Prepared Plan Cache 的内存管理。

- 在 TiDB v6.1.0 之前，SHOW ANALYZE STATUS 显示实例级别的任务，且 TiDB 重启后任务记录会被清空。从 TiDB v6.1.0 起，SHOW ANALYZE STATUS 显示集群级别的任务，且 TiDB 重启后仍能看到重启之前的任务记录。当 tidb_analyze_version = 2 时 Job_info 列增加了 analyze option 的信息。

- TiKV 中损坏的 SST 文件会导致 TiKV 进程崩溃。在 TiDB v6.1.0 之前，损坏的 SST 文件会导致 TiKV 进程立即崩溃，从 TiDB v6.1.0 起，TiKV 进程会在 SST 文件损坏的 1 小时之后崩溃。

- TiKV 中以下配置项支持在线修改：

  - raftstore.raft-entry-max-size
  - quota.foreground-cpu-time
  - quota.foreground-write-bandwidth
  - quota.foreground-read-bandwidth
  - quota.max-delay-duration
  - server.grpc-memory-pool-quota
  - server.max-grpc-send-msg-len
  - server.raft-msg-max-batch-size

- v6.1.0 中，部分配置项转化为系统变量。对于从 v6.1.0 之前版本升级到 v6.1.0 的集群（包括滚动升级和停机升级），请注意：

  - 若升级前集群指定的配置文件中，存在已经配置的项，则升级过程中 TiDB 将会将配置项的值自动更新为对应系统变量的值，以保证升级后，系统的行为不会因为参数的优化发生变化。
  - 上述自动更新仅在升级过程中发生一次，升级完成之后，被废弃的配置项不再有任何效果。

- DM WebUI 移除 Dashboard 页面。

- TiCDC 启用 dispatchers.topic、dispatchers.partition 配置后不支持降级到 v6.1.0 以前的版本。

- TiCDC 使用 Avro 协议的 Changefeed 不支持降级到 v6.1.0 以前的版本。

16.11.8.3 改进提升

- TiDB
  - 提升 UnionScanRead 算子的性能 #32433
  - 优化 Explain 输出中 task type 的显示（增加 MPP task type）#33332
  - 支持使用 rand() 作为列的默认值 #10377
  - 支持使用 uuid() 作为列的默认值 #33870
  - 支持将表或列的字符集从 latin1 修改为 utf8/utf8mb4 #34008
- TiKV
  - 提升引入内存悲观锁后 CDC 旧数据的命中率 #12279
  - 健康检查可以检测到无法正常工作的 Raftstore，使得 TiKV client 可以及时更新 Region Cache #12398
  - 支持设置 Raft Engine 的内存限制 #12255
  - TiKV 自动检测和删除损坏的 SST 文件，提高产品可服务性 #10578
  - CDC 支持 RawKV #11965
  - 支持将较大的 snapshot 切割为多个文件 #11595
  - 将 snapshot 垃圾回收 (GC) 从 Raftstore 中迁移到后台线程，防止 snapshot GC 阻塞 Raftstore 消息循环 #11966
  - 支持动态设置 gRPC 可发送的最大消息长度 (max-grpc-send-msg-len) 和单个 gPRC 消息可包含的最大 Raft 消息个数 (raft-msg-max-batch-size) #12334
  - 支持通过 Raft 执行在线有损恢复 #10483
- PD
  - 支持设置 Region Label 的 Time to live (TTL) #4694
  - 支持 Region Buckets #4668
  - 默认关闭编译 swagger server #4932
- TiFlash
  - 优化聚合算子的内存统计，从而能在 Merge 阶段选择更高效的算法 #4451
- Tools
  - Backup & Restore (BR)
    - 支持备份恢复空库 #33866
  - TiDB Lightning
    - 优化 Scatter Region 为批量模式，提升 Scatter Region 过程的稳定性 #33618
  - TiCDC
    - TiCDC 支持在数据复制过程中拆分大事务，能够有效降低大事务带来的复制延迟 #5280

16.11.8.4 错误修复

- TiDB
  - 修复 IN 函数处理 BIT 数据类型时可能会导致 TiDB panic 的问题 #33070
  - 修复 UnionScan 无法保序导致的查询结果不正确的问题 #33175
  - 修复特定情况下 Merge Join 执行结果错误的问题 #33042
  - 修复动态裁减模式下 `index join` 的结果可能会错误的问题 #33231
  - 修复分区表的一些分区被 DROP 删除后数据可能不被 GC 垃圾回收的问题 #33620
  - 修复集群的 PD 节点被替换后一些 DDL 语句会卡住一段时间的问题 #33908
  - 修复了查询 `INFORMATION_SCHEMA.CLUSTER_SLOW_QUERY` 表导致 TiDB 服务器 OOM 的问题，在 Grafana dashboard 中查看慢查询记录的时候可能会触发该问题 #33893
  - 修复系统变量 `max_allowed_packet` 不生效的问题 #31422
  - 修复 TopSQL 模块的内存泄露问题 #34525，#34502
  - 修复 Plan Cache 对于 PointGet 计划有时候会出错的问题 #32371
  - 修复在 RC 隔离情况下 Plan Cache 启用时可能导致查询结果错误的问题 #34447
- TiKV
  - 修复下线一个 TiKV 实例导致 Raft log lag 越来越大的问题 #12161
  - 修复待 merge 的 Region 无效会导致 TiKV panic 且非预期地销毁 peer 的问题 #12232
  - 修复从 v5.3.1、v5.4.0 升级到 v6.0.0 及以上版本时 TiKV 报 `failed to load_latest_options` 错误的问题 #12269
  - 修复内存资源不足时 append Raft log 导致 OOM 的问题 #11379
  - 修复销毁 peer 和批量分裂 Region 之间的竞争导致的 TiKV panic #12368
  - 修复 `stats_monitor` 线程陷入死循环导致短期内 TiKV 内存占用陡增的问题 #12416
  - 修复进行 Follower Read 时，可能会报 `invalid store ID 0` 错误的问题 #12478
- PD
  - 修复 `not leader` 的 status code 有误的问题 #4797
  - 修复在某些特殊情况下 TSO fallback 的问题 #4884
  - 修复已清除的 `tombstone store` 信息在切换 PD leader 后再次出现的问题 #4941
  - 修复 PD leader 转移后调度不能立即启动的问题 #4769
- TiDB Dashboard
  - 修复 Top SQL 功能无法统计到其开启时刻正在运行的 SQL 的问题 #33859
- TiFlash
  - 修复大量 INSERT 和 DELETE 操作后可能导致 TiFlash 数据不一致的问题 #4956
- Tools
  - TiCDC
    * 优化了 ddl schema 缓存方式，降低了内存消耗 #1386
    * 修复了增量扫描特殊场景下的数据丢失问题 #5468

- TiDB Data Migration (DM)
  - 修复 `start-time` 时区问题，从使用下游时区改为使用上游时区 #5471
  - 修复任务自动恢复后，DM 会占用更多磁盘空间的问题 #3734，#5344
  - 修复 checkpoint flush 可能导致失败行数据被跳过的问题 #5279
  - 修复了某些情况下，过滤 DDL 并在下游手动执行会导致同步任务不能自动重试恢复的问题 #5272
  - 修复在未设置 `case-sensitive: true` 时无法同步大写表的问题 #5255
  - 修复了在 `SHOW CREATE TABLE` 语句返回的索引中，主键没有排在第一位导致的 DM worker panic 的问题 #5159
  - 修复了当开启 GTID 模式或者任务自动恢复时，可能出现一段时间 CPU 占用高并打印大量日志的问题 #5063
  - 修复 DM Web UI offline 选项及其他使用问题 #4993
  - 修复上游 GTID 配置为空时，增量任务启动失败的问题 #3731
  - 修复空配置可能导致 dm-master panic 的问题 #3732
- TiDB Lightning
  - 修复前置检查中没有检查本地磁盘空间以及集群是否可用的问题 #34213
  - 修复 schema 路由错误的问题 #33381
  - 修复 TiDB Lightning panic 时 PD 配置未正确恢复的问题 #31733
  - 修复由 `auto_increment` 列的数据越界导致 local 模式导入失败的问题 #29737
  - 修复 `auto_random`、`auto_increment` 列为空时 local 模式导入失败的问题 #34208

16.12 v6.0

16.12.1 TiDB 6.0.0 Release Notes

发版日期：2022 年 4 月 7 日

TiDB 版本：6.0.0-DMR

> **注意：**
>
> TiDB 6.0.0-DMR 的用户文档已归档。如无特殊需求，建议使用 TiDB 数据库的最新 LTS 版本。

在 6.0.0-DMR 版本中，你可以获得以下关键特性：

- 基于 SQL 的数据放置规则，提供更灵活的数据放置管理能力。
- 内核层面的数据索引一致性检查，通过极低的资源开销提升系统稳定性和健壮性。
- 面向非专家的性能诊断功能 Top SQL，提供一体化、自助的数据库性能观测及诊断能力。
- 支持持续性能分析，持续记录集群的故障现场性能数据，缩短技术专家故障诊断时间。
- 热点小表缓存，大幅提高访问性能，提升吞吐，降低访问延迟。
- 内存悲观锁优化，在悲观锁性能瓶颈下，可以有效降低 10% 延迟，提升 10% QPS。
- 增强 Prepared Statement 执行计划共享，降低 CPU 资源消耗，提升 SQL 执行效率。

- 提升 MPP 引擎计算性能，支持更多表达式下推，正式引入弹性线程池。
- 新增 DM WebUI，方便地通过图形化的方式管理大量迁移任务。
- 提升 TiCDC 在大规模集群下同步数据的稳定性和资源利用效率，支持高达 10 万张表的同时同步。
- TiKV 节点重启后 leader 平衡加速，提升业务恢复速度。
- 支持手动取消统计信息的自动更新，减少资源争抢，降低对业务 SQL 性能的影响。
- PingCAP Clinic 自动诊断服务（Technical Preview 版本）
- TiDB Enterprise Manager 企业级数据库管理平台

另：作为 TiDB HTAP 方案的核心组件，TiFlashTM 于本次发布同时正式开放源码。详见 GitHub。

16.12.1.1 版本策略变更

从 TiDB v6.0.0 开始，TiDB 的发版会有两个系列：

- 长期支持版本 (Long-Term Support Releases)

  长期支持版本约每六个月发布一次，会引入新的功能和改进，并会按需在版本生命周期内发布 Bug 修订版本。例如：v6.1.0。

- 开发里程碑版 (Development Milestone Releases, DMR)

  DMR 版本约每两个月发布一次，会引入新的功能和改进。TiDB 不提供基于 DMR 的 Bug 修订版本，不推荐在生产环境使用。例如：v6.0.0-DMR。

v6.0.0 是 DMR 版本，版本名称为 6.0.0-DMR。

16.12.1.2 新功能

16.12.1.2.1 SQL

- 基于 SQL 的数据放置规则

  TiDB 是具有优秀扩展能力的分布式数据库，通常数据横跨多个服务器甚至多数据中心部署，数据调度管理是 TiDB 最重要的基础能力之一。大多数情况下用户无需关心数据如何调度管理，但是随着业务复杂度的提升，因隔离性和访问延迟导致的数据部署变更是 TiDB 面对的新的挑战。TiDB 从 6.0.0 版本开始正式提供基于 SQL 接口的数据调度管理能力，支持针对任意数据提供副本数、角色类型、放置位置等维度的灵活调度管理能力，在多业务共享集群、跨 AZ 部署下提供更灵活的数据放置管理能力。

  用户文档

- 新增按库构建 TiFlash 副本功能。用户仅需使用一条 SQL 即可对某一个数据库中所有的表添加 TiFlash 副本，极大地节约了运维成本。

  用户文档

16.12.1.2.2 事务

- 内核层面增加数据索引一致性检查

  在事务执行过中增加数据索引一致性检查，通过极低的资源开销提升系统稳定性和健壮性。你可以通过 `tidb_enable_mutation_checker` 和 `tidb_txn_assertion_level` 参数控制检查行为。默认配置下，大多数场景下 QPS 下降控制在 2% 以内。关于数据索引一致性检查的报错说明，请参考用户文档。

16.12.1.2.3 可观测性

- Top SQL：面向非专家的 SQL 性能诊断功能

  Top SQL 是一个面向运维人员及应用开发者的一体化、自助的数据库性能观测和诊断功能，集成于 TiDB Dashboard 图形化界面，在 TiDB v6.0.0 正式发布。

  与现有 TiDB Dashboard 中各个面向数据库专家的诊断功能不同的是，Top SQL 完全面向非专家：你不需要观察几千张监控图表寻找相关性，也不需要理解诸如 Raft Snapsnot、RocksDB、MVCC、TSO 等 TiDB 内部机制，仅需要知道常见的数据库概念，如索引、锁冲突、执行计划等，就可以通过 Top SQL 快速分析数据库负载情况，并提升应用程序的性能。

  Top SQL 功能功能默认关闭。启用后，通过 Top SQL 提供的各个 TiDB 或 TiKV 节点实时 CPU 负载情况，你可以直观了解各节点的高 CPU 负载来自哪些 SQL 语句，从而快速分析诸如数据库热点和负载陡升等问题。例如，你可以通过 Top SQL 分析某个 TiKV 节点上正在消耗 90% CPU 负载的 SQL 查询语句的具体内容及执行情况。

  用户文档

- 持续性能分析

  持续性能分析 (Continuous Profiling) 功能集成于 TiDB Dashboard，在 TiDB v6.0.0 中正式发布。该功能默认关闭，启用该功能后，集群将以极低的开销自动收集各 TiDB、TiKV 及 PD 实例每时每刻的性能数据。通过这些历史性能数据，技术专家可以在事后回溯、分析该集群任意时刻（如曾经出现过高内存占用）的问题根因，无需等待问题复现，从而有助于缩短故障诊断时间。

  用户文档

16.12.1.2.4 性能

- 热点小表缓存

  用户业务遇到热点小表访问场景下，支持显式将热点表缓存于内存中，大幅提高访问性能，提升吞吐，降低访问延迟。该方案可以有效避免引入三方缓存中间件，降低架构复杂性，减少运维管理成本，适用于高频访问低频更新的小表场景，例如配置表，汇率表等。

  用户文档，#25293

- 内存悲观锁优化

  TiDB 从 v6.0.0 开始默认开启内存悲观锁功能。开启后，悲观事务锁管理将在内存中完成，避免悲观锁持久化，也避免了锁信息的 Raft 复制，大大降低悲观事务锁管理的开销。在悲观锁性能瓶颈下，通过悲观锁内存优化，可以有效降低 10% 延迟，提升 10% QPS。

  用户文档，#11452

- RC 隔离级别下优化 TSO 获取开销

  在RC 隔离级别下，增加 `tidb_rc_read_check_ts` 变量，用于在读写冲突较少情况下，减少不必要 TSO 获取，从而降低查询延迟。该参数默认关闭，开启后，在没有读写冲突的场景下，该优化几乎可以避免重复 TSO 获取，降低延迟。但是在高读写冲突场景下，开启该参数有可能造成性能回退，请验证后使用。

  用户文档，#33159

- 增强 Prepared Statement 执行计划共享

  SQL 执行计划复用可以有效减少 SQL 解析时间，降低 CPU 资源消耗，提升 SQL 执行效率。有效复用 SQL 执行计划是 SQL 调优的重要手段之一。TiDB 已经支持 Prepared Statement 下的计划共享。但是在 Prepared Statement close 时，TiDB 会主动清空对应的 Plan Cache。这会对重复执行的 SQL 造成不必要的解析，影响语句的执行效率。TiDB 从 v6.0.0 开始支持通过 `tidb_ignore_prepared_cache_close_stmt` 参数控制是否忽视 `COM_STMT_CLOSE` 指令，该参数默认关闭。开启该参数后，TiDB 可以忽视 Prepared Statement 的 close 指令，并在缓存中保留对应的执行计划，从而提升执行计划的复用率。

  用户文档，#31056

- 增强查询的下推功能

  TiDB 是原生计算存储分离架构，算子下推可以在存储层过滤无效数据，大大减少 TiDB 与 TiKV 的数据传输，提升查询效率。TiDB 在 v6.0.0 支持更多的表达式和 BIT 数据类型下推至 TiKV，以提升运算该类内容时的查询效率。

  用户文档，#30738

- 热点索引优化

  在二级索引上批量写入单调递增的值会形成索引热点，影响整体写入吞吐。自 v6.0.0 起，TiDB 支持通过 `tidb_shard` 函数将热点索引打散，以提升写入性能。目前 `tidb_shard` 只能打散二级唯一索引的热点。该方案不需要用户修改原有查询条件，对业务非常友好，适用于高吞吐写入、点查询、批量点查询场景。请注意如果业务中使用范围查询打散后的数据，可能造成性能回退，请验证后使用。

  用户文档，#31040

- TiFlash MPP 引擎支持分区表的动态裁剪模式（实验特性）

  在该模式下，TiDB 也可以使用 TiFlash MPP 引擎读取和计算分区表的数据，从而大大提升分区表的查询性能。

  用户文档

- 持续提升 MPP 引擎计算性能

  - 支持更多函数和算子下推至 MPP 引擎

    * 逻辑函数：`IS`，`IS NOT`
    * 字符串函数：`REGEXP()`，`NOT REGEXP()`
    * 数学函数：`GREATEST(int/real)`，`LEAST(int/real)`
    * 日期函数：`DAYNAME()`，`DAYOFMONTH()`，`DAYOFWEEK()`，`DAYOFYEAR()`，`LAST_DAY()`，`MONTHNAME()`
    * 算子：Anti Left Outer Semi Join, Left Outer Semi Join

    用户文档

  - 正式引入弹性线程池，提升 CPU 利用率，默认开启此功能

    用户文档

16.12.1.2.5 稳定性

- 执行计划自动捕获增强

  增强了执行计划自动捕获的易用性，增加了黑名单功能，支持表名、频率和用户名三个维度的黑名单设置。引入新的算法来优化绑定缓存的内存管理机制。开启自动捕获后，可以为绝大多数 OLTP 类查询

自动创建绑定，从而固定被绑定语句的执行计划，避免因执行计划变动导致的性能问题。通常用于大版本升级和集群迁移等场景，可以有效减少因计划回退造成的性能问题。

用户文档，#32466

- TiKV 过载资源保护增强 (实验特性)

  当 TiKV 部署的机型资源受限时，如果前台处理的读写请求量过大，会导致后台处理请求的 CPU 资源被前台占用，最终影响 TiKV 性能的稳定性。TiDB v6.0.0 支持手动限制 TiKV 前台各类请求的资源用量，包括 CPU、读写带宽等，以提升集群在长期高负载压力下的稳定性。

  用户文档，#12131

- TiFlash 新增支持 zstd 压缩算法

  新增 `profiles.default.dt_compression_method` 和 `profiles.default.dt_compression_level` 两个参数，用户可根据对性能和容量的平衡，选择不同的压缩算法。

  用户文档

- TiFlash 默认开启支持所有 I/O 的校验 (Checksum)。

  此项功能曾作为实验特性在 v5.4 释出。除增强了数据的正确性安全性外，对用户使用不产生明显的直接影响。

  警告：新版本数据格式将不支持原地降级为早于 v5.4 的版本，需要在降级处理时删除 TiFlash Replica 待降级完成后重新同步；或使用离线工具进行数据版本降级。

  用户文档

- TiFlash 引入异步 gRPC 和 Min-TSO 调度机制，更好的管理线程使用，防止线程数过高导致的系统崩溃。

  用户文档

16.12.1.2.6 数据迁移

DM

- WebUI (实验特性)

  新增 WebUI 方便地通过图形化的方式管理大量迁移任务。已支持的功能有：

  - 在 Dashboard 上显示迁移任务信息
  - 管理迁移任务
  - 配置上游参数
  - 查询同步状态
  - 查看 Master 和 Worker 信息

  此特性为实验性，尚存在不完善之处。建议仅用于体验，且已知与 dmctl 操作同一任务可能存在问题，此现象将于后续版本改进。

  用户文档

- 增强错误处理机制

  当迁移任务中断时，增加更多命令以方便的解决遇到的错误。例如：

  - 遇到 schema 错误时，可以通过 `binlog-schema update` 命令的 `--from-source/--from-target` 参数直接更新，无需单独编写 schema 文件。
  - 可以指定某个 binlog position，inject/replace/skip/revert DDL 语句。

  用户文档

- 支持存储全量数据至 Amazon S3

  DM 执行 all/full 类型的迁移任务时，需要足够的硬盘空间存放上游的全量数据。相比 EBS，Amazon S3 具有更低的成本和近似无限的容量。DM 现在可以将 dump 目录配置为 Amazon S3 路径，执行 all/full 类型的迁移任务时则直接使用 Amazon S3 存放全量数据。

  用户文档

- 支持从指定的时间点启动迁移任务

  启动任务时新增了参数 `--start-time`，支持 `'2021-10-21 00:01:00'` 或 `'2021-10-21T00:01:00'` 格式的自定义时间。

  此特性在多个 MySQL 实例合并增量迁移场景尤其有用，无需为每个上游增量同步设置 binlog 起始位置，而是通过 `--start-time` 参数配合 safe-mode 即可更快的完成增量任务配置。

  用户文档

TiDB Lightning

- 最大可容忍错误

  增加 `lightning.max-error` 配置项，默认值为 0 以保持原有行为。当值 > 0 时，表示 max-error 功能开启。Lightning 在进行编码时，如果出现报错，则会在目标 TiDB 的 `lightning_task_info.type_error_v1` 表中增加一条记录包含此报错行的信息，并忽略此行；当出现的报错行数超过配置值时 Lightning 将立即退出。

  与之相匹配的配置项 `lightning.task-info-schema-name` 用于定义保存出错数据记录的库名。

  此特性尚不能涵盖所有类型的错误，例如 syntax error。

  用户文档

16.12.1.2.7 TiDB 数据共享订阅

- 支持高达 10 万张表的同时同步

  TiCDC 针对数据处理流程进行了优化，降低了处理每张表增量数据时所需要的资源，极大地提升了 TiCDC 在大规模集群下同步数据的稳定性和资源利用效率。在测试中，TiCDC 可以稳定支持 10 万张表的增量数据的同时同步。

16.12.1.2.8 部署及运维

- 默认采用新 Collation 规则

  TiDB 从 v4.0 开始支持新 collation 规则，在大小写不敏感、口音不敏感、padding 规则上与 MySQL 行为保持一致。新 Collation 规则可以通过 new_collations_enabled_on_first_bootstrap 参数控制，默认关闭。从 v6.0.0 开始，TiDB 默认开启新 Collation 规则，请注意该配置仅在集群初始化时生效。

  用户文档

- TiKV 节点重启后 leader 平衡加速

  TiKV 节点重启后，需要将分布不均匀的 leader 重分配以达到负载均衡的效果。在大规模集群下，leader 平衡时间与 Region 数量正相关。例如，在 100K Region 下，leader 平衡耗时可能达到 20-30 分钟，容易引发负载不均导致的性能问题，造成稳定性风险。TiDB v6.0.0 提供了 leader 平衡的并发度参数控制，并调整默认值为原来的 4 倍，大幅缩短 leader 重平衡的时间，提升 TiKV 节点重启后的业务恢复速度。

  用户文档，#4610

- 支持手动取消统计信息的自动更新

  统计信息是影响 SQL 性能的最重要基础数据之一，为了保证统计信息的完整性和及时性，TiDB 会在后台定期自动更新对象的统计信息。但是，统计信息自动更新可能造成资源争抢，影响业务 SQL 性能。为了避免这个问题，TiDB v6.0.0 支持手动取消统计信息的自动更新。

  用户文档

- PingCAP Clinic 诊断服务 Technical Preview 版本上线

  PingCAP Clinic 为 TiDB 集群提供诊断服务，支持远程定位集群问题和本地快速检查集群状态，用于从全生命周期确保 TiDB 集群稳定运行、预测可出现的集群问题、降低问题出现概率、快速定位并修复问题。

  当 TiDB 集群出现问题，需要邀请 PingCAP 技术支持人员协助定位问题时，你可以通过 PingCAP Clinic 服务采集并上传诊断数据，从而大大提高定位问题的速度。

  用户文档

- 企业级数据库管理平台 TiDB Enterprise Manager

  TiDB Enterprise Manager 是一款以 TiDB 数据库为核心的企业级数据库管理平台，帮助用户在本地部署环境或公有云环境中管理 TiDB 集群。

  TiDB Enterprise Manager 不仅为 TiDB 集群提供全生命周期的可视化管理，也同时一站式提供 TiDB 数据库的参数管理、数据库版本升级、克隆集群、主备集群切换、数据导入导出、数据同步、数据备份恢复服务，能有效提高 TiDB 集群运维效率，降低企业运维成本。

  TiDB Enterprise Manager 当前为企业版特性。要获取 TiDB Enterprise Manager 及其文档，请在 TiDB 产品页面企业版下点击立即咨询与 PingCAP 取得联系。

- 支持监控组件的自定义配置

  使用 TiUP 部署 TiDB 集群时，TiUP 会同时自动部署 Prometheus、Grafana 和 Alertmanager 等监控组件，并且在集群扩容中自动为新增节点添加监控配置。通过在 `topology.yaml` 文件中添加对应的配置项，你可以对监控组件进行自定义配置。

  用户文档

16.12.1.3 兼容性变化

> 注意：
>
> 当从一个早期的 TiDB 版本升级到 TiDB v6.0.0 时，如需了解所有中间版本对应的兼容性更改说明，请查看对应版本的Release Notes。

16.12.1.3.1 系统变量

变量名	修改类型	描述
placement_checks	删除	该变量用于控制 DDL 语句是否验证通过 Placement Rules in SQL 指定的放置规则。已被 tidb_placement_mode 替代。
tidb_enable_alter_placement	删除	该变量用于开启 Placement Rules in SQL。

变量名	修改类型	描述
tidb_mem_quota_hashjoin ↪ tidb_mem_quota_indexlookupjoin ↪ tidb_mem_quota_indexlookupreader ↪ tidb_mem_quota_mergejoin ↪ tidb_mem_quota_sort ↪ tidb_mem_quota_topn ↪ tidb_mem_quota_query ↪	删除	从 TiDB v5.0 起，这几个变量被 tidb_mem_quota_query ↪ 取代并从系统变量文档中移除，为了保证兼容性代码中还保留。从 TiDB v6.0.0 起，代码中也正式移除这些变量。

变量名	修改类型	描述
tidb_enable_mutation_checker ↪	新增	设置是否开启 mutation checker，默认开启。如果从低于 v6.0.0 的版本升级到 v6.0.0，升级后默认关闭。
tidb_ignore_prepared_cache_close_stmt ↪	新增	设置是否忽略关闭 Prepared Statement 的指令，默认值为 OFF。

变量名	修改类型	描述
tidb_memory_usage_binding_cache_ratio	新增	设置存放 binding 的缓存的内存使用阈值，默认值为 67108864 (64 MiB)。
tidb_placement_mode	新增	控制 DDL 语句是否忽略 Placement Rules in SQL 指定的放置规则。默认值为 strict，表示不忽略。

| tidb_rc_read_check_ts | 新增 |

优化事务内读语句延迟。如果读写冲突较为严重，开启此变量会增加额外开销和延迟，造成性能回退。默认关闭。

该变量与 replica-read 尚不兼容，开启 tidb_rc_read_check_ts 的读请求无法使用 replica-read，请勿同时开启两个变量。

| | tidb_sysdate_is_now | 新增 | 控制 SYSDATE 函数是否替换为 NOW 函数，效果与 MySQL 中的 sysdate-is-now 一致。默认值为 OFF。 | | tidb_table_cache_lease | 新增 | 用来控制缓存表（新增 feature）的 lease 时间，默认值是 3 秒。 | | tidb_top_sql_max_meta_count | 新增 | 用于控制 Top SQL 每分钟最多收集 SQL 语句类型的数量，默认值为 5000。 | | tidb_top_sql_max_time_series_count | 新增 | 用于控制 Top SQL 每分钟保留消耗负载最大

的前多少条 SQL（即 Top N）的数据，默认值为 100。| | `tidb_txn_assertion_level` | 新增 | 设置 assertion 级别，assertion 是一项在事务提交过程中进行的数据索引一致性校验。默认仅开启对性能影响微小的检查，包含大部分检查效果。如果从低于 v6.0.0 的版本升级到 v6.0.0，升级后默认关闭检查。 |

16.12.1.3.2 配置文件参数

配置文件	配置项	修改类型	描述
配置文件	配置项	修改类型	描述
TiDB	stmt-summary.enable	删除	系统表 statement_summary_tables 的相关配置，所有配置项现已移除，统一改成用 SQL variable 控制。
	stmt-summary.history-size		
	stmt-summary.max-sql-length		
	stmt-summary.max-stmt-count		
	stmt-summary.refresh-interval		

配置文件	配置项	修改类型	描述
TiDB	new_collations_enabled_on_first_bootstrap ↪	修改	用于开启新的 collation 支持。自 v6.0.0 起默认值从 false 改为 true。该配置项只有在初次初始化集群时生效，初始化集群后，无法通过更改该配置项打开或关闭新的 collation 框架。

配置文件	配置项	修改类型	描述
TiKV	backup ↪ . ↪ num ↪ - ↪ threads ↪	修改	修改可调整范围为 [1, ↪ CPU ↪]。
TiKV	raftstore ↪ . ↪ apply ↪ - ↪ max ↪ - ↪ batch ↪ - ↪ size ↪	修改	添加最大值为 10240。

| TiKV | raftstore.raft-max-size-per-msg | 修改 |

修改最小值（由 0 修改为大于 0）

添加最大值为 3GB

添加单位（由 MB 增加为 KB|MB|GB）

| | TiKV | raftstore.store-max-batch-size | 修改 | 添加最大值为 10240。| | TiKV | readpool.unified.max-thread ↪ -count | 修改 | 修改可调整范围为 [min-thread-count, MAX(4, CPU)]。| | TiKV | rocksdb.enable-pipelined ↪ -write | 修改 | 修改默认值为 false。开启时会使用旧的 Pipelined Write，关闭时会使用新的 Pipelined Commit 机制。| | TiKV | rocksdb.max-background-flushes | 修改 | 在 CPU 核数为 10 时修改默认值为 3，在 CPU 核数量为 8 时默认为 2。| | TiKV | rocksdb.max-background-jobs | 修改 | 在 CPU 核数为 10 时修改默认值为 9，在 CPU 核数量为 8 时默认为 7。| | TiFlash | profiles.default.dt_enable_logical_split | 修改 | 存储引擎的 segment 分裂是否使用逻辑分裂。自 v6.0.0 起默认值从 true 改为 false。| | TiFlash | profiles.default.enable_elastic_threadpool | 修改 | 是否启用可自动扩展的线程池。自 v6.0.0 起默认值从 false 改为 true。| | TiFlash | storage.format_version | 修改 | 该配置项控制 TiFlash 存储引擎的校验功能，自 v6.0.0 起默认值从 2 改为 3。format_version 设置为 3 时，支持对 TiFlash 的所有数据的读操作进行一致性校验，避免由于硬件故障而读到错误的数据。注意：新版本数据格式不支持原地降级为早于 5.4 的版本。 | | TiDB | pessimistic-txn.pessimistic-auto-commit | 新增 | 用来控制开启全局悲观事务模式下 (tidb_txn_mode='pessimistic') 时，自动提交的事务使用的事务模式。| | TiKV | pessimistic-txn.in-memory | 新增 | 开启内存悲观锁功能。开启该功能后，悲观事务会尽可能在 TiKV 内存中存储悲观锁，而不将悲观锁写入磁盘，也不将悲观锁同步给其他副本，从而提升悲观事务的性能。但有较低概率出现悲观锁丢失的情况，可能会导致悲观事务提交失败。该参数默认值为 true。| | TiKV | quota | 新增 | 新增前台限流相关的配置项，可以限制前台各类请求所占用的资源。前台限流功能为实验特性，默认关闭。新增的相关配置项为 foreground-cpu-time、foreground-write-bandwidth、foreground-read-bandwidth、max-delay- ↪ duration。| | TiFlash | profiles.default.dt_compression_method | 新增 | TiFlash 存储引擎的压缩算法，支持

LZ4、zstd 和 LZ4HC，大小写不敏感。默认使用 LZ4 算法。| | TiFlash | profiles.default.dt_compression_level | 新增 | TiFlash 存储引擎的压缩级别，默认值 1。| | DM | loaders.<name>.import-mode | 新增 | 该配置项控制全量阶段数据导入的模式。自 v6.0.0 起全量阶段默认使用 TiDB Lightning 的 TiDB-backend 方式导入，替换原来的 Loader 组件。此变动为内部组件替换，对日常使用没有明显影响。默认值 sql 表示启用 tidb-backend 组件，可能在极少数场景下存在未能完全兼容的情况，可以通过配置为 “loader” 回退。| | DM | loaders.<name>.on-duplicate | 新增 | 该配置项控制全量导入阶段出现的冲突数据的解决方式。默认值为 replace，覆盖重复数据。| | TiCDC | dial-timeout | 新增 | 和下游 Kafka 建立连接的超时时长，默认值为 10s | | TiCDC | read-timeout | 新增 | 读取下游 Kafka 返回的 response 的超时时长，默认值 10s | | TiCDC | write-timeout | 新增 | 向下游 Kafka 发送 request 的超时时长，默认值为 10s |

16.12.1.3.3 其他

- 数据放置策略的兼容性变更：
  - 不支持绑定，并从语法中删除直接放置 (direct placement) 选项。
  - CREATE PLACEMENT POLICY 和 ALTER PLACEMENT POLICY 语句不再支持 VOTERS 和 VOTER_CONSTRAINTS 放置选项。
  - TiDB 数据迁移工具（TiDB Binlog、TiCDC、BR）现在兼容 placement rules。Placement 选项移到 binlog 的特殊注释中。
  - 系统表 information_schema.placement_rules 重命名为 information_schema.placement_policies。此表现在只展示放置策略的信息。
  - 系统变量 placement_checks 被 tidb_placement_mode 替代。
  - 禁止在有 TiFlash 副本的表上添加带放置规则的分区。
  - 将 TIDB_DIRECT_PLACEMENT 列从 INFORMATION_SCHEMA 表中删除。
- 执行计划管理（SPM）绑定的 status 值变更：
  - 删除 using。
  - 新增 enabled（可用），取代之前版本的 using 状态。
  - 新增 disabled（不可用）。
- DM 修改 OpenAPI 接口
  - 由于内部机制变更，任务管理相关接口与之前的实验特性版本无法保持兼容，需要参阅新的OpenAPI 文档进行适配。
- DM 全量数据冲突处理方式变化
  - 新增 loader.<name>.on-duplicate 参数，默认值为 replace，表示覆盖冲突数据。若希望保持以前版本的行为，可以改为 error。此参数仅影响全量数据导入阶段的行为。
- DM 需使用对应版本的 dmctl 工具
  - 由于内部机制变更，升级 DM 集群版本至 v6.0.0 后，也必须升级 dmctl 与之匹配。
- 在 v5.4（仅 v5.4）中，TiDB 允许将一些 noop 系统变量设置为不正确的值。从 v6.0.0 起，TiDB 不再允许将系统变量设置为不正确的值 #31538

16.12.1.4 离线包变更

TiDB 提供两个离线包下载：`v6.0.0 TiDB-community-server` 软件包和 `v6.0.0 TiDB-community-toolkit` 软件包。

在 6.0.0-DMR 版本中，两个离线包的内容物做了一些调整。在 `v6.0.0 TiDB-community-server` 软件包中，离线包的内容物包含：

- tidb-{version}-linux-amd64.tar.gz
- tikv-{version}-linux-amd64.tar.gz
- tiflash-{version}-linux-amd64.tar.gz
- pd-{version}-linux-amd64.tar.gz
- ctl-{version}-linux-amd64.tar.gz
- grafana-{version}-linux-amd64.tar.gz
- alertmanager-{version}-linux-amd64.tar.gz
- blackbox_exporter-{version}-linux-amd64.tar.gz
- prometheus-{version}-linux-amd64.tar.gz
- node_exporter-{version}-linux-amd64.tar.gz
- tiup-linux-amd64.tar.gz
- tiup-{version}-linux-amd64.tar.gz
- local_install.sh
- cluster-{version}-linux-amd64.tar.gz
- insight-{version}-linux-amd64.tar.gz
- diag-{version}-linux-amd64.tar.gz
- influxdb-{version}-linux-amd64.tar.gz
- playground-{version}-linux-amd64.tar.gz

在 `v6.0.0 TiDB-community-toolkit` 软件包中，离线包的内容物包含：

- tikv-importer-{version}-linux-amd64.tar.gz
- pd-recover-{version}-linux-amd64.tar.gz
- etcdctl
- tiup-linux-amd64.tar.gz
- tiup-{version}-linux-amd64.tar.gz
- tidb-lightning-{version}-linux-amd64.tar.gz
- tidb-lightning-ctl
- dumpling-{version}-linux-amd64.tar.gz
- cdc-{version}-linux-amd64.tar.gz
- dm-{version}-linux-amd64.tar.gz
- dm-worker-{version}-linux-amd64.tar.gz
- dm-master-{version}-linux-amd64.tar.gz
- dmctl-{version}-linux-amd64.tar.gz
- br-{version}-linux-amd64.tar.gz
- spark-{version}-any-any.tar.gz
- tispark-{version}-any-any.tar.gz
- package-{version}-linux-amd64.tar.gz
- bench-{version}-linux-amd64.tar.gz

- errdoc-{version}-linux-amd64.tar.gz
- dba-{version}-linux-amd64.tar.gz
- PCC-{version}-linux-amd64.tar.gz
- pump-{version}-linux-amd64.tar.gz
- drainer-{version}-linux-amd64.tar.gz
- binlogctl
- sync_diff_inspector
- reparo
- arbiter
- mydumper

16.12.1.5 提升改进

- TiDB
  - 当通过 FLASHBACK 或 RECOVER 语句恢复一张表之后，自动清除该表的放置规则信息 #31668
  - 新增一个性能概览监控面板，展示典型关键路径上的核心性能指标，使 TiDB 上的指标分析更加容易 #31676
  - 支持在 LOAD DATA LOCAL INFILE 语句中使用 REPLACE 关键字 #24515
  - 支持在 Range 类型分区表中对 IN 表达式进行分区裁剪 #26739
  - 消除 MPP 聚合查询中可能冗余的 Exchange 操作，提高查询效率 #31762
  - 允许在 TRUNCATE PARTITION 和 DROP PARTITION 语句中使用重复的分区名，提高与 MySQL 的兼容性 #31681
  - 支持在 ADMIN SHOW DDL JOBS 语句的执行结果中显示 CREATE_TIME 信息 #23494
  - 新增内置函数 CHARSET() #3931
  - 支持通过用户名过滤自动捕获的黑名单 #32558
  - 支持在自动捕获的黑名单中使用通配符 #32714
  - 优化 ADMIN SHOW DDL JOBS 和 SHOW TABLE STATUS 语句的执行结果，支持根据当前的 time_zone 显示结果中的时间 #26642
  - 支持下推 DAYNAME()和 MONTHNAME() 函数到 TiFlash #32594
  - 支持下推 REGEXP 函数到 TiFlash #32637
  - 支持下推 DAYOFMONTH()，LAST_DAY() 函数到 TiFlash #33012
  - 支持下推 DAYOFWEEK() 和 DAYOFYEAR() 函数到 TiFlash #33130
  - 支持下推 IS_TRUE、IS_FALSE、IS_TRUE_WITH_NULL 函数到 TiFlash #33047
  - 支持下推 GREATEST 和 LEAST 函数到 TiFlash #32787
  - 支持追踪 UnionScan 算子的执行情况 #32631
  - 支持读取 _tidb_rowid 列的查询能够使用 PointGet 计划 #31543
  - 支持在 EXPLAIN 语句的输出中显示原有的分区名而不转换为小写 #32719
  - 支持对 RANGE COLUMNS 分区表在 IN 条件和字符串类型上进行分区裁剪 #32626
  - 当设置系统变量为 NULL 时提供错误提示 #32850
  - 移除非 MPP 模式的 Broadcast Join #31465
  - 支持在动态裁剪模式的分区表上执行 MPP 计划 #32347
  - 支持对公共表表达式 (CTE) 进行谓词下推 #28163
  - 简化 Statement Summary 和 Capture Plan Baselines 的配置为只在全局基础上可用 #30557
  - 更新 gopsutil 的版本到 v3.21.12，避免在 macOS 12 上构建二进制时出现警告 #31607

- TiKV
  - 提升 Raftstore 对含有较多 key ranges 的 batch 的采样准确度 #12327
  - 为 debug/pprof/profile 添加正确的 Content-Type，使 Profile 更容易被识别 #11521
  - 当 Raftstore 在心跳或处理读请求时，通过更新其租约时间来无限延长 leader 的租约时间，减少 leader 切换导致的延迟抖动 #11579
  - 切换 leader 时以选择代价最小的 store 为目标，提升性能稳定性 #10602
  - 异步获取 Raft log，减少阻塞 Raftstore 带来的性能抖动 #11320
  - 向量计算支持 QUARTER 函数 #5751
  - 支持 BIT 数据类型下推至 TiKV #30738
  - 支持 MOD 函数和 SYSDATE 函数下推至 TiKV #11916
  - 通过减少需要进行清理锁 (Resolve Locks) 步骤的 Region 数量来减少 TiCDC 恢复时间 #11993
  - 支持动态修改 raftstore.raft-max-inflight-msgs #11865
  - 支持 EXTRA_PHYSICAL_TABLE_ID_COL_ID，以实现动态裁剪模式 #11888
  - 支持以 buckets 为单位进行计算 #11759
  - 将 RawKV API V2 的 Key 编码为 user-key + memcomparable-padding + timestamp #11965
  - 将 RawKV API V2 的 Value 编码为 user-value + ttl + ValueMeta，并且将 delete 编码在 ValueMeta 中 #11965
  - 支持动态修改 raftstore.raft-max-size-per-msg #12017
  - 使 Grafana 支持 multi-k8s 的监控 #12104
  - 通过将 leader 转让给 CDC observer 减少延迟抖动 #12111
  - 支持动态修改 raftstore.apply_max_batch_size 和 raftstore.store_max_batch_size #11982
  - 支持 RawKV API V2 在收到 raw_get 或 raw_scan 请求时会返回最新的版本 #11965
  - 支持 RCCheckTS 的一致性读 #12097
  - 支持动态修改 storage.scheduler-worker-pool-size（Scheduler 线程池中线程的数量）#12067
  - 通过全局的前台限流器来控制 CPU 与带宽的使用，增加 TiKV 性能的稳定性 #11855
  - 支持动态修改 readpool.unified.max-thread-count（UnifyReadPool 线程池中线程的数量）#11781
  - 使用 TiKV 内部的 pipeline 替代 RocksDB pipeline，废弃 rocksdb.enable-multibatch-write 参数 #12059
- PD
  - 支持驱逐 leader 时自动选择最快的目标进行迁移，加速驱逐过程 #4229
  - 禁止直接从有 2 副本的 Raft Group 中删除一个 Voter，防止 Region 不可用 #4564
  - 提升 Balance Leader 的调度速度 #4652
- TiFlash
  - 禁止了 TiFlash 文件的逻辑分裂（默认参数调整为 profiles.default.dt_enable_logical_split = ↪ false，详见用户文档），优化了 TiFlash 列存储的空间使用效率，使得同一个表在同步到 TiFlash 后所占用空间与 TiKV 相近
  - TiFlash 优化了集群管理和 replica 数据同步机制。将原有的集群管理模块迁移整合进了 TiDB，并提高了为小表创建 TiFlash replica 的速度 #29924
- Tools
  - Backup & Restore (BR)
    * 默认开启批量建表功能，备份数据恢复速度提升。在模拟测试中恢复 16 TB 的备份数据到 15 个节点的 TiKV 集群（每个节点有 16 个 CPU 核心），恢复吞吐可以达到 2.66 GiB/s #27036

    * 支持导入与导出放置规则 (Placement Rule)。增加参数 `--with-tidb-placement-mode` 来控制导入时是否忽略放置规则 #32290

  - TiCDC

    * 在 Grafana 中添加 `Lag analyze` 监控面板 #4891
    * 支持放置规则 (placement rules) #4846)
    * 同步处理 HTTP API #1710
    * 为 changefeed 重启操作添加指数退避机制 #3329
    * 设置 MySQL sink 的默认隔离级别为 Read Committed，以减少 MySQL 中的死锁 #3589
    * 在创建 changefeed 时验证参数合法，优化报错信息 #1716 #1718 #1719 #4472
    * 暴露 Kafka producer 配置参数，使之在 TiCDC 中可配置 #4385

  - TiDB Data Migration (DM)

    * 支持在 “乐观协调” 模式（optimistic）下，上游表结构不一致的情况下仍能启动任务 #3629 #3708 #3786
    * 支持在 `stopped` 状态下创建任务 #4484
    * 支持 Syncer 使用 DM-worker 的工作目录写内部文件，不再使用 /tmp 目录。任务停止后会清理掉该目录 #4107
    * 优化了 Precheck 功能。不再允许跳过某些重要的检查 #3608

  - TiDB Lightning

    * 增加了更多重试错误类型 #31376
    * 支持 base64 格式的密码字符串 #31194
    * 标准化错误码和错误输出 #32239

16.12.1.6 Bug 修复

- TiDB

  - 修复了当 `SCHEDULE = majority_in_primary`，且 `PrimaryRegion` 和 `Regions` 的值相同时创建 placement rule 会报错的问题 #31271
  - 修复查询时用到 index lookup join 导致 `invalid transaction` 报错的问题 #30468
  - 修复了当授予大于等于 2 个权限时 `show grants` 返回不正确的结果的问题 #30855
  - 修复了在默认值为 `CURRENT_TIMESTAMP` 的字段执行 `INSERT INTO t1 SET tsCol = DEFAULT` 语句时插入零值的问题 #29926
  - 通过避免编码字符串类型的最大值和最小非空值，修复读取结果时的报错问题 #31721
  - 修复 LOAD DATA 语句处理转义字符时可能 panic 的问题 #31589
  - 修复带有 collation 的 `greatest` 或 `least` 函数结果出错的问题 #31789
  - 修复 date_add 和 date_sub 函数可能返回错误数据类型的问题 #31809
  - 修复使用 insert 语句插入数据到虚拟生成列时可能出现 panic 的问题 #31735
  - 修复创建 list column 分区表时出现重复列不报错的问题 #31784
  - 修复 `select for update union select` 语句使用错误快照导致结果可能错误的问题 #31530
  - 修复当恢复完成后，Region 有可能分布不均的问题 #31034
  - 修复 `json` 类型 Coercibility 值不正确的问题 #31541
  - 修复了 `json` 类型在 builtin-func 中推导 collation 错误的问题 #31320
  - 修复当设置 TiFlash 副本数为 0 时 PD 规则没有被删除的问题 #32190

  - 修复 `alter column set default` 错误地修改表定义的问题 #31074
  - 修复 date_format 对 `'\n'` 的处理与 MySQL 不兼容的问题 #32232
  - 修复使用 join 更新分区表时可能报错的问题 #31629
  - 修复 Nulleq 函数作用在 Enum 类型上可能出现结果错误的问题 #32428
  - 修复 upper 和 lower 函数可能造成 panic 的问题 #32488
  - 修复了将其他类型列更改为时间戳类型列时可能遇到的时区问题 #29585
  - 修复使用 ChunkRPC 导出数据时可能造成 TiDB OOM 的问题 #31981 #30880
  - 修复动态分区裁剪模式下访问分区表时 Limit 在子查询中不能生效的问题 #32516
  - 修复 `INFORMATION_SCHEMA.COLUMNS` 表中 `bit` 类型默认值格式错误或或不一致问题 #32655
  - 修复重启实例后 list 分区表的分区裁剪可能不生效的问题 #32416
  - 修复了在执行 `SET timestamp` 语句后，`add column` 语句可能会使用错误的默认时间戳的问题 #31968
  - 修复使用 MySQL 5.5/5.6 客户端连接 TiDB 无密码用户时可能失败的问题 #32334
  - 修复在事务中使用动态模式读取分区表时结果不正确的问题 #29851
  - 修复 TiDB 可能向 TiFlash 发送重复任务的问题 #32814
  - 修复 `timdiff` 函数的输入包含毫秒时可能出现结果错误的问题 #31680
  - 修复显式读取分区并使用 IndexJoin 计划时可能出现结果错误的问题 #32007
  - 修复重命名列时并发修改列类型会导致重命名错误的问题 #31075
  - 修复 TiFlash 执行计划网络成本计算公式未与 TiKV 对齐的问题 #30103
  - 修复 `KILL TIDB` 在空闲链接上无法立即生效的问题 #24031
  - 修复读取生成列时可能出现结果错误的问题 #33038
  - 修复使用 left join 同时删除多张表数据时可能出现错误结果的问题 #31321
  - 修复 `subtime` 函数在出现 Overflow 时可能返回错误结果的问题 #31868
  - 修复当聚合查询包含 having 条件时 selection 算子不能被下推的问题 #33166
  - 修复查询报错时可能阻塞 CTE 的问题 #31302
  - 修复了在非严格模式下，创建表时 varbinary 或 varchar 类型列的长度太大导致报错的问题 #30328
  - 修复未指定 follower 时 `information_schema.placement_policies` 中显示的 follower 值有误的问题 #31702
  - 修复创建索引时允许指定列前缀长度为 0 的问题 #31972
  - 修复允许分区名以空格结尾的问题 #31535
  - 修正 `RENAME TABLE` 语句的报错信息 #29893

- TiKV

  - 修复 Peer 状态为 Applying 时快照文件被删除会造成 panic 的问题 #11746
  - 修复开启流量控制且显式设置 `level0_slowdown_trigger` 时出现 QPS 下降的问题 #11424
  - 修复删除 Peer 可能造成高延迟的问题 #10210
  - 修复 GC worker 繁忙后无法执行范围删除（即执行内部命令 `unsafe_destroy_range`）的问题 #11903
  - 修复在某些边界场景中 `StoreMeta` 内数据被意外删除会引发 TiKV panic 的问题 #11852
  - 修复在 ARM 平台上进行性能分析造成 TiKV panic 的问题 #10658
  - 修复 TiKV 运行 2 年以上可能 panic 的问题 #11940
  - 修复因缺少 SSE 指令集导致的 ARM64 架构下的编译问题 #12034
  - 修复删除未初始化的副本可能会造成旧副本被重新创建的问题 #10533
  - 修复旧信息造成 TiKV panic 的问题 #12023
  - 修复 TsSet 转换可能发生未定义行为 (UB) 的问题 #12070
  - 修复 Replica Read 可能违反线性一致性的问题 #12109
  - 修复在 Ubuntu 18.04 下进行性能分析会造成 TiKV panic 的问题 #9765

  - 修复 tikv-ctl 对 `bad-ssts` 结果字符串进行错误匹配的问题 #12329
  - 修复因内存统计指标溢出而造成的间歇性丢包和内存不足 (OOM) 的问题 #12160
  - 修复 TiKV 在退出时可能误报 panic 的问题 #12231

- PD

  - 修复 PD 生成带有无意义的 Joint Consensus 步骤的 Operator 的问题 #4362
  - 修复关闭 PD Client 时撤销 TSO 的流程可能会卡住的问题 #4549
  - 修复 Region Scatterer 生成的调度缺失部分 Peer 的问题 #4565
  - 修复不能动态设置 `dr-autosync` 的 `Duration` 字段的问题 #4651

- TiFlash

  - 修复启用内存限制时 TiFlash 崩溃的问题 #3902
  - 修复过期数据回收缓慢的问题 #4146
  - 修复并发执行多个 DDL 操作和 Apply Snapshot 操作时 TiFlash 可能会崩溃问题 #4072
  - 修复在读取工作量大时添加列后可能出现的查询错误 #3967
  - 修复 `SQRT` 函数中传入负值后返回 `NaN` 而不是 `Null` 的问题 #3598
  - 修复将 `INT` 类型转换为 `DECIMAL` 类型可能造成溢出的问题 #3920
  - 修复 `IN` 函数的结果在多值表达式中不正确的问题 #4016
  - 修复日期格式将 `'\n'` 处理为非法分隔符的问题 #4036
  - 修复在高并发场景下 Learner 读过程时间过长的问题 #3555
  - 修复将 `DATETIME` 转换为 `DECIMAL` 时结果错误的问题 #4151
  - 修复查询被取消时出现的内存泄露问题 #4098
  - 修复开启弹性线程池可能导致内存泄漏的问题 #4098
  - 修复启用本地隧道时取消 MPP 查询可能导致任务永远挂起的问题 #4229
  - 修复 HashJoin 构建端失败可能导致 MPP 查询永远挂起的问题 #4195
  - 修复 MPP 任务可能永远泄漏线程的问题 #4238

- Tools

  - Backup & Restore (BR)

    * 修复了当恢复操作遇到一些无法恢复的错误时，BR 被卡住的问题 #33200
    * 修复了在备份重试过程中加密信息丢失导致的恢复操作失败的问题 #32423

  - TiCDC

    * 修复 MySQL sink 在禁用 `batch-replace-enable` 参数时生成重复 `replace` SQL 语句的错误 #4501
    * 修复了 TiCDC 进程在 PD leader 被杀死时的异常退出问题 #4248
    * 修复使用某些版本 MySQL sink 时可能遇到 `Unknown system variable 'transaction_isolation'` 报错的问题 #4504
    * 修复 `Canal-JSON` 错误处理 `string` 格式可能导致的 TiCDC panic 问题 #4635
    * 修复某些情况下序列对象被错误同步的问题 #4552
    * 修复 `Canal-JSON` 不支持 nil 可能导致的 TiCDC panic 问题 #4736
    * 修复对 Enum/Set 和 TinyText/MediumText/Text/LongText 类型 avro 编码的数据映射错误 #4454
    * 修复 Avro 把 `NOT NULL` 列转换成 `nullable` 字段的错误 #4818
    * 修复 TiCDC 无法退出的问题 #4699

  - TiDB Data Migration (DM)

    * 修复部分 syncer metrics 只有在查询状态时才得以更新的问题 #4281
    * 修复了 UPDATE 语句在安全模式下执行错误会导致 DM 进程挂掉的问题 #4317
    * 修复了 varchar 类型值长度过长时的 `Column length too big` 错误 #4637
    * 修复了多个 DM-worker 写入来自同一上游的数据导致的冲突问题 #3737
    * 修复了日志中出现数百条 “checkpoint has no change, skip sync flush checkpoint” 以及迁移性能下降的问题 #4619
    * 修复了悲观模式下对上游增量数据进行分库分表合并迁移时有可能会丢 DML 的问题 #5002
  - TiDB Lightning
    * 修复在某些导入操作没有包含源文件时，TiDB Lightning 不会删除 metadata schema 的问题 #28144
    * 修复了源文件和目标集群中的表格名称不一致导致数据迁移失败的问题 #31771
    * 修复了 checksum 报错 “GC life time is shorter than transaction duration” #32733
    * 修复了检查空表失败导致 TiDB Lightning 卡住的问题 #31797
  - Dumpling
    * 修复了执行 `dumpling --sql $query` 进度显示不准确的问题 #30532
    * 修复了 Amazon S3 无法正确计算压缩数据大小的问题 #30534
  - TiDB Binlog
    * 修复了上游写大事务向 Kafka 同步时可能会导致 TiDB Binlog 被跳过的问题 #1136

如果你在使用 TiDB v6.0.0 的过程中遇到问题，可以到 AskTUG 论坛浏览、搜索或反馈问题。

16.13 v5.4

16.13.1 TiDB 5.4.3 Release Notes

发版日期：2022 年 10 月 13 日

TiDB 版本：5.4.3

16.13.1.1 提升改进

- TiKV
  - 支持将 RocksDB write stall 参数设置为比 flow control 流控阈值更小的值 #13467
  - 支持配置 `unreachable_backoff` 避免 Raftstore 发现某个 Peer 无法连接时广播过多消息 #13054
- Tools
  - TiDB Lightning
    * 优化 Scatter Region 为批量模式，提升 Scatter Region 过程的稳定性 #33618
  - TiCDC
    * 优化了多 Region 场景下，runtime 上下文切换带来过多性能开销的问题 #5610

16.13.1.2 Bug 修复

- TiDB
  - 修复 `SHOW CREATE PLACEMENT POLICY` 输出结果不正确的问题 #37526
  - 修复集群的 PD 节点被替换后一些 DDL 语句会卡住一段时间的问题 #33908
  - 修复 `KILL TIDB` 在空闲连接上无法立即生效的问题 #24031
  - 修复在 TiDB 上查询 `INFORMATION_SCHEMA.COLUMNS` 系统表得到的 `DATA_TYPE` 和 `COLUMN_TYPE` 列结果不正确的问题 #36496
  - 修复在开启 TiDB Binlog 时，TiDB 执行 `ALTER SEQUENCE` 会产生错误的元信息版本号，进而导致 Drainer 报错退出的问题 #36276
  - 修复 `UNION` 运算符可能会非预期地返回空结果的问题 #36903
  - 修复在 TiFlash 中为分区表开启动态模式时结果出错的问题 #37254
  - 修复 `INL_HASH_JOIN` 和 `LIMIT` 一起使用时可能会卡住的问题 #35638
  - 修复执行 `SHOW WARNINGS` 时可能会报 `invalid memory address or nil pointer dereference` 的问题 #31569
  - 修复在读已提交隔离级别 (RC isolation level) 执行 Stale Read 报 `invalid transaction` 错误的问题 #30872
  - 修复带 DML 算子的 `EXPLAIN ANALYZE` 语句可能在事务提交完成前返回结果的问题 #37373
  - 修复开启 TiDB Binlog 后插入重复数据导致 data and columnID count not match 错误的问题 #33608
  - 修复 Static Partition Prune 模式下带聚合条件的 SQL 语句在表为空时结果错误的问题 #35295
  - 修复 TiDB 在执行 `UPDATE` 语句时可能会 panic 的问题 #32311
  - 修复 UnionScan 无法保序导致的查询结果不正确的问题 #33175
  - 修复在某些情况下 `UPDATE` 语句错误地消除了 projection 导致 `Can't find column` 报错的问题 #37568
  - 修复某些情况下分区表无法充分利用索引来扫描数据的问题 #33966
  - 修复某些情况下，`EXECUTE` 语句可能抛出非预期异常的问题 #37187
  - 修复开启 Prepared Plan Cache 后，使用 `BIT` 类型的索引可能会导致查询结果错误的问题 #33067
- TiKV
  - 修复了 PD leader 发生切换或重启 PD 后，在集群中执行 SQL 语句会出现持续报错的问题 #12934
    - 问题原因：该问题是由于 TiKV 存在 bug，TiKV 向 PD client 发送心跳请求失败后不会重试，只能等待与 PD client 重连。这样，故障 TiKV 节点上的 Region 的信息会逐步变旧，使得 TiDB 无法获取最新的 Region 信息，导致 SQL 执行出错。
    - 影响版本：v5.3.2 和 v5.4.2。目前该问题已在 v5.3.3 和 v5.4.3 上修复。如果你使用 v5.4.2 的 TiDB 集群，可以升级至 v5.4.3。
    - 规避方法：除升级外，你还可以重启无法向 PD 发送 Region 心跳的 TiKV 节点，直至不再有待发送的 Region 心跳为止。
  - 修复 Web 身份提供程序 (web identity provider) 报错并失效后，自动恢复为默认提供程序 (default provider) 时出现权限拒绝的问题 #13122
  - 修复 PD 客户端可能会出现死锁的问题 #13191
  - 修复 Raftstore 线程繁忙时，可能会出现 Region 重叠的问题 #13160
- PD
  - 修复 PD 无法正确处理 dashboard 代理请求的问题 #5321
  - 修复已清除的 `tombstone store` 信息在切换 PD leader 后再次出现的问题 #4941

  - 修复 PD 可能没创建 TiFlash Learner 副本的问题 #5401

- TiFlash

  - 修复 `format` 函数可能会报 `Data truncated` 错误的问题 #4891
  - 修复并行聚合出错时可能导致 TiFlash crash 的问题 #5356
  - 修复使用包含 NULL 值的列创建主键时导致崩溃的问题 #5859

- Tools

  - TiDB Lightning

    * 修复 BIGINT 类型自增列可能越界的问题 #27397
    * 修复极端情况下去重可能会导致 TiDB Lightning panic 的问题 #34163
    * 修复 TiDB Lightning 不支持 Parquet 文件中以斜线 (/)、数字、非 ASCII 字符开头的特殊列名的问题 #36980
    * 修复了使用 IPv6 host 时无法连接到 TiDB 的问题 #35880

  - TiDB Data Migration (DM)

    * 修复 DM Worker 获取 DB 连接时可能卡住的问题 #3733
    * 修复 DM 报错 `Specified key was too long` 的问题 #5315
    * 修复数据同步过程中，latin1 字符集数据可能损坏的问题 #7028
    * 修复 DM IPv6 支持问题 #6249
    * 修复 `query-status` 内可能存在 data race 的问题 #4811
    * 修复 relay 报错时可能导致 goroutine 泄露问题 #6193

  - TiCDC

    * 修复 `enable-old-value = false` 时可能出现的 cdc panic 问题 #6198

  - Backup & Restore (BR)

    * 修复当外部存储的鉴权 Key 中存在某些特殊符号时，会导致备份恢复失败的问题 #37469
    * 修复在恢复时配置过高的 concurrency 会导致 Region 不均衡的问题 #37549

  - Dumpling

    * 修复 GetDSN 方法不支持 IPv6 的问题 #36112

16.13.2 TiDB 5.4.2 Release Notes

发版日期：2022 年 7 月 8 日

TiDB 版本：5.4.2

> **警告：**
>
> 不建议使用 v5.4.2，因为该版本已知存在 bug，详情参见 #12934。该 bug 已在 v5.4.3 中修复，建议升级至v5.4.3。

16.13.2.1 提升改进

- TiDB
  - 避免向非健康状态的 TiKV 节点发送请求，以提升可用性 #34906
- TiKV
  - 当 TLS 证书更新时自动重新加载，以提升可用性 #12546
  - 健康检查可以检测到无法正常工作的 Raftstore，使得 TiKV client 可以及时更新 Region Cache #12398
  - 通过将 leader 转让给 CDC observer 减少延迟抖动 #12111
- PD
  - 默认关闭编译 swagger server #4932
- Tools
  - TiDB Lightning
    * 优化 Scatter Region 为批量模式，提升 Scatter Region 过程的稳定性 #33618

16.13.2.2 Bug 修复

- TiDB
  - 修复在 binary protocol 下 Plan Cache 有时会缓存错误的 TableDual 计划的问题 #34690 #34678
  - 修复了执行计划在 EqualAll 的情况下，把 TiFlash 的 `firstrow` 聚合函数的 null flag 设错的问题 #34584
  - 修复了执行处理器为 TiFlash 生成错误的两阶段 aggregate 计划的问题 #34682
  - 修复了 `tidb_opt_agg_push_down` 和 `tidb_enforce_mpp` 启用时执行处理器的错误行为 #34465
  - 修复了 Plan Cache 在被 evict 时使用了错误的 memory usage 指标的问题 #34613
  - 修复了 `LOAD DATA` 语句中列的列表不生效的问题 #35198
  - 避免在悲观事务中报出 `Write Conflict` 错误 #11612
  - 修复了在遇到 Region error 和网络问题时 prewrite 请求不幂等的问题 #34875
  - 修复了回滚 async commit 事务可能导致事务不满足原子性的问题 #33641
  - 如果发生网络连接问题，TiDB 并不总是能正确释放已断开会话所占有的资源。该修复可以确保 TiDB 回滚打开的事务以及释放其他相关资源。#34722
  - 修复在查询包含 CTE 的视图时，可能误报 `references invalid table` 的问题 #33965
  - 修复 TiDB 由于 `fatal error: concurrent map read and map write` 发生崩溃的问题 #35340
- TiKV
  - 修复 `max_sample_size` 为 0 时 ANALYZE 可能导致 panic 的问题 #11192
  - 修复 TiKV 在退出时可能误报 panic 的问题 #12231
  - 修复在 Region merge 时 source peer 通过 snapshot 追日志时可能导致 panic 的问题 #12663
  - 修复同时分裂和销毁一个 peer 时可能导致 panic 的问题 #12825
  - 修复了 PD 客户端遇到报错时频繁重连的问题 #12345
  - 修复了 `DATETIME` 类型的数据包含小数部分和 `Z` 后缀导致检查报错的问题 #12739
  - 修复了对空字符串进行类型转换导致 TiKV panic 的问题 #12673

  - 修复了在悲观事务中使用 Async Commit 导致重复提交记录的问题 #12615
  - 修复进行 Follower Read 时，可能会报 `invalid store ID 0` 错误的问题 #12478
  - 修复销毁 peer 和批量分裂 Region 之间的竞争导致的 TiKV panic #12368
  - 修复 tikv-ctl 对 `bad-ssts` 结果字符串进行错误匹配的问题 #12329
  - 修复在 aufs 上启动 TiKV 报错的问题 #12543

- PD

  - 修复 `not leader` 的 status code 有误的问题 #4797
  - 修复由于 Hot Region 没有 leader 导致 PD Panic 的问题 #5005
  - 修复 PD leader 转移后调度不能立即启动的问题 #4769
  - 修复在某些特殊情况下 TSO fallback 的问题 #4884

- TiFlash

  - 修复在 clustered index 表删除列导致 TiFlash 崩溃的问题 #5154
  - 修复大量 INSERT 和 DELETE 操作后可能导致 TiFlash 数据不一致的问题 #4956
  - 修复极端情况下 decimal 比较结果可能有误的问题 #4512

- Tools

  - Backup & Restore (BR)

    * 修复了 RawKV 模式下 BR 报 `ErrRestoreTableIDMismatch` 错误的问题 #35279
    * 修复了出现保存文件错误时 BR 没有重试的问题 #34865
    * 修复了 BR 运行时 panic 的问题 #34956
    * 修复 BR 无法处理 S3 内部错误的问题 #34350
    * 修复了当恢复操作遇到一些无法恢复的错误时，BR 被卡住的问题 #33200

  - TiCDC

    * 修复了增量扫描特殊场景下的数据丢失问题 #5468
    * 修复 redo log manager 提前 flush log 的问题 #5486
    * 修复当一部分表没有被 redo writer 管理时 resolved ts 提前推进的问题 #5486
    * 添加 UUID 作为 redo log file 的后缀以解决文件名冲突引起的数据丢失问题 #5486
    * 修复了 Region Leader 丢失时重试超过次数上限后同步中断的问题 #5230
    * 修复 MySQL Sink 可能会保存错误的 checkpointTs 的问题 #5107
    * 修复了 HTTP server 中潜在的 goroutine 泄露问题 #5303
    * 修复了元信息所在 Region 发生变化时可能导致延迟上升的问题 #4756 #4762

  - TiDB Data Migration (DM)

    * 修复了任务自动恢复后 DM 会占用更大的磁盘空间的问题 #5344
    * 修复在未设置 `case-sensitive: true` 时无法同步大写表的问题 #5255

16.13.3 TiDB 5.4.1 Release Notes

发版日期：2022 年 5 月 13 日

TiDB 版本：5.4.1

16.13.3.1 兼容性更改

TiDB v5.4.1 未引入产品设计上的兼容性变化，但请注意该版本中的 Bug 修复也可能带来兼容性变更，请参考Bug 修复部分。

16.13.3.2 提升改进

- TiDB
  - 支持读取 `_tidb_rowid` 列的查询能够使用 PointGet 计划 #31543
  - 为 `Apply` 算子添加更多日志和监控指标，方便用户了解该算子的并行状态 #33887
  - 对于统计信息使用的 Analyze Version 2，优化了 TopN 裁剪的逻辑 #34256
  - 在 Grafana 的仪表盘中支持显示多个 Kubernetes 集群 #32593
- TiKV
  - 在 Grafana 的仪表盘中支持显示多个 Kubernetes 集群 #12104
- PD
  - 在 Grafana 的仪表盘中支持显示多个 Kubernetes 集群 #4673
- TiFlash
  - 在 Grafana 的仪表盘中支持显示多个 Kubernetes 集群 #4129
- Tools
  - TiCDC
    * 在 Grafana 监控面板中支持多个 Kubernetes 集群 #4665
    * 暴露 Kafka producer 配置参数，使之在 TiCDC 中可配置 #4385
  - TiDB Data Migration (DM)
    * 支持 Syncer 使用 DM-worker 的工作目录写内部文件，不再使用 /tmp 目录。任务停止后会清理掉该目录 #4107

16.13.3.3 Bug 修复

- TiDB
  - 修复 date_format 对 `'\n'` 的处理与 MySQL 不兼容的问题 #32232
  - 修复 `ENUM` 或 `SET` 类型的列因为编码错误导致写入数据错误的问题 #32302
  - 修复特定情况下 Merge Join 执行结果错误的问题 #33042
  - 修复关联子查询返回结果中有常量时导致执行结果出错的问题 #32089
  - 修复在 TiFlash 不支持使用空范围读表的情况，依然选择 TiFlash 导致查询结果错误的问题 #33083
  - 修复开启 New Collation 时，作用在 `ENUM` 或`SET` 列上的 `MAX` 或 `MIN` 函数结果出错的问题 #31638
  - 修复查询报错时可能阻塞 `CTE` 的问题 #31302
  - 修复 Nulleq 函数作用在 Enum 类型上可能出现结果错误的问题 #32428

- 修复使用 ChunkRPC 导出数据时可能造成 TiDB OOM 的问题 #31981 #30880
- 修复开启 `tidb_restricted_read_only` 后 `tidb_super_read_only` 没有自动开启的问题 #31745
- 修复带有 collation 的 `greatest` 或 `least` 函数结果出错的问题 #31789
- 修复 LOAD DATA 语句处理转义字符时可能 panic 的问题 #31589
- 修复查询时用到 index lookup join 导致 `invalid transaction` 报错的问题 #30468
- 修复使用 left join 同时删除多张表数据时可能出现错误结果的问题 #31321
- 修复 TiDB 可能向 TiFlash 发送重复任务的问题 #32814
- 修复了集群从 4.0 版本升级后，为用户授予 `all` 权限时报错的问题 #33588
- 修复了在 MySQL binary 协议下，当 schema 变更后，执行 prepared statement 会导致会话崩溃的问题 #33509
- 修复了 `compress()` 表达式在 `tidb_enable_vectorized_expression` 开启时，执行会报错的问题 #33397
- 修复了 `reArrangeFallback()` 函数使用 CPU 资源过多的问题 #30353
- 修复对于新加入的分区，表属性 (table attributes) 无法被检索到，以及分区更新后，表的 range 信息不会被更新的问题 #33929
- 修复了表的 TopN 统计信息在初始化时未正确排序的问题 #34216
- 修复了读取 `INFORMATION_SCHEMA.ATTRIBUTES` 表报错的问题，对于无法识别的 attributes 会做跳过处理 #33665
- 修复了当查询要求结果有序的情况下，即使设置了 `@@tidb_enable_parallel_apply`，Apply 算子依然不使用并行模式执行的问题 #34237
- 修复了在 sql_mode 为 `NO_ZERO_DATE` 的限制下，用户依然可以插入数据 `'0000-00-00 00:00:00'` 到 `datetime` 列的问题 #34099
- 修复了查询 `INFORMATION_SCHEMA.CLUSTER_SLOW_QUERY` 表导致 TiDB 服务器 OOM 的问题，在 Grafana dashboard 中查看慢查询记录的时候可能会触发该问题 #33893
- 修复了在 `NOWAIT` 语句中，事务执行遇到了锁后，并不会立刻返回的问题 #32754
- 修复创建字符集为 GBK 且 collation 为 `gbk_bin` 的表失败的问题 #31308
- 修复启用配置 `enable-new-charset` 后，当字符集为 GBK 且指定 collation 时，建表报错 “Unknown character set” 的问题 #31297

- TiKV

  - 修复待 merge 的 Region 无效会导致 TiKV panic 且非预期地销毁 peer 的问题 #12232
  - 修复旧信息造成 TiKV panic 的问题 #12023
  - 修复因内存统计指标溢出而造成的间歇性丢包和内存不足 (OOM) 的问题 #12160
  - 修复在 Ubuntu 18.04 下进行性能分析会造成 TiKV panic 的问题 #9765
  - 修复 replica read 可能违反线性一致性的问题 #12109
  - 修复合并 Region 时因 target peer 被一个未进行初始化就被销毁的 peer 所替换，从而引起 TiKV panic 的问题 #12048
  - 修复 TiKV 运行 2 年以上可能 panic 的问题 #11940
  - 通过减少需要进行清理锁 (Resolve Locks) 步骤的 Region 数量来减少 TiCDC 恢复时间 #11993
  - 修复 Peer 状态为 Applying 时快照文件被删除会造成 panic 的问题 #11746
  - 修复删除 Peer 可能造成高延迟的问题 #10210
  - 修复资源管理不正确断言导致 panic 的问题 #12234
  - 修复部分情况下 slow score 计算不准确的问题 #12254
  - 修复 `resolved_ts` 模块内存管理不合理导致的 OOM 问题，增加更多监控指标 #12159
  - 修复网络出现问题的情况下，已成功提交的乐观事务可能报 `Write Conflict` 错误的问题 #34066

  - 修复启用 replica read 时，在网络出现问题的情况下 TiKV 可能 panic 的问题 #12046

- PD

  - 修复不能动态设置 `dr-autosync` 的 `Duration` 字段的问题 #4651
  - 修复存在较大空间 Store 时（例如 2T），无法检测满的小空间 Store，从而无法进行平衡调度的问题 #4805
  - 修复监控信息中已删除 label 的残留问题 #4825

- TiFlash

  - 修复启用 TLS 时可能导致的崩溃 #4196
  - 修复在滞后的 Region peer 上执行 Region Merge 导致的元数据损坏问题 #4437
  - 修复在执行带有 JOIN 的查询遇到错误时可能被挂起的问题 #4195
  - 修复 MPP 任务可能永远泄漏线程的问题 #4238
  - 修复将 FLOAT 类型转换为 DECIMAL 类型可能造成溢出的问题 #3998
  - 修复过期数据回收缓慢的问题 #4146
  - 修复启用本地隧道时取消 MPP 查询可能导致任务永远挂起的问题 #4229
  - 修复查询被取消时出现的内存泄露问题 #4098
  - 修复将 DATETIME 转换为 DECIMAL 时结果错误的问题 #4151
  - 修复并发执行多个 DDL 操作和 Apply Snapshot 操作时 TiFlash 可能会崩溃问题 #4072
  - 修复错误地配置存储目录会导致非预期行为的问题 #4093
  - 修复一些异常没有被正确地处理的问题 #4101
  - 修复将 INT 类型转换为 DECIMAL 类型可能造成溢出的问题 #3920
  - 修复 IN 函数的结果在多值表达式中不正确的问题 #4016
  - 修复日期格式将 `'\n'` 处理为非法分隔符的问题 #4036
  - 修复在读取工作量大时添加列后可能出现的查询错误 #3967
  - 修复启用内存限制后 TiFlash 崩溃的问题 #3902
  - 修复 DTFile 中可能出现的数据损坏问题 #4778
  - 修复查询存在大量 delete 操作的表时可能报错的问题 #4747
  - 修复 TiFlash 随机报错 “Keepalive watchdog fired” 的问题 #4192
  - 修复 TiFlash 节点上遗留了与 Region range 不匹配的数据的问题 #4414
  - 修复空 segments 在 GC 后无法合并的问题 #4511

- Tools

  - Backup & Restore (BR)

    * 修复了在备份重试过程中加密信息丢失导致的恢复操作失败的问题 #32423
    * 修复 BR 无法备份 RawKV 的问题 #32607
    * 修复增量恢复后在表中插入记录时遇到的重复主键问题 #33596
    * 修复增量恢复期间，由于 DDL 查询任务为空导致的报错 #33322
    * 修复当恢复完成后，Region 有可能分布不均的问题 #31034
    * 修复恢复过程中 Region 不一致时 BR 重试次数不足的问题 #33419
    * 修复在某些情况下，恢复过程中开启 merge 小文件功能导致 BR panic 的问题 #33801
    * 修复了 BR 或 TiDB Lightning 在异常退出的时候，scheduler 没有重置的问题 #33546

  - TiCDC

    * 修复切换 owner 会导致其 metrics 数据不正确的问题 #4774
    * 修复 Canal-JSON 不支持 nil 可能导致的 TiCDC panic 问题 #4736
    * 修复 Unified Sorter 的 workerpool 稳定性问题 #4447
    * 修复某些情况下序列对象被错误同步的问题 #4552
    * 修复 Canal-JSON 错误处理 `string` 格式可能导致的 TiCDC panic 问题 #4635
    * 修复了 TiCDC 进程在 PD leader 被杀死时的异常退出问题 #4248
    * 修复 MySQL sink 在禁用 `batch-replace-enable` 参数时生成重复 `replace` SQL 语句的错误 #4501
    * 修复了 `rename tables` DDL 导致的生成 DML 错误的问题 #5059
    * 修复了在某些极端情况下，开启 new scheduler（默认关闭）时更新 owner 导致同步卡住的问题 #4963
    * 修复了在开启 new scheduler（默认关闭）时出现报错 (ErrProcessorDuplicateOperations) 的问题 #4769
    * 修复了在开启 TLS 后，`--pd` 中设置的第一个 PD 不可用导致 TiCDC 无法启动的问题 #4777
    * 修复了在表被调度时 checkpoint 监控缺失的问题 #4714

  - TiDB Lightning

    * 修复了 checksum 报错 “GC life time is shorter than transaction duration” #32733
    * 修复了检查空表失败导致 TiDB Lightning 卡住的问题 #31797
    * 修复在某些导入操作没有包含源文件时，TiDB Lightning 不会删除 metadata schema 的问题 #28144
    * 修复前置检查中没有检查本地磁盘空间以及集群是否可用的问题 #34213

  - TiDB Data Migration (DM)

    * 修复了日志中出现数百条 “checkpoint has no change, skip sync flush checkpoint” 以及迁移性能下降的问题 #4619
    * 修复了 varchar 类型值长度过长时的 `Column length too big` 错误 #4637
    * 修复了 UPDATE 语句在安全模式下执行错误会导致 DM 进程挂掉的问题 #4317
    * 修复了某些情况下，过滤 DDL 并在下游手动执行会导致同步任务不能自动重试恢复的问题 #5272
    * 修复了在上游没有开启 binlog 时执行 `query-status` 命令返回结果为空的问题 #5121
    * 修复了在 `SHOW CREATE TABLE` 语句返回的索引中，主键没有排在第一位导致的 DM worker panic 的问题 #5159
    * 修复了当开启 GTID 模式或者任务自动恢复时，可能出现一段时间 CPU 占用高并打印大量日志的问题 #5063

16.13.4 TiDB 5.4 Release Notes

发版日期：2022 年 2 月 15 日

TiDB 版本：5.4.0

在 v5.4.0 版本中，你可以获得以下关键特性：

- 支持 GBK 字符集
- 支持索引合并 (Index Merge) 数据访问方法，能够合并多个列上索引的条件过滤结果
- 支持通过 session 变量实现有界限过期数据读取
- 支持统计信息采集配置持久化
- 支持使用 Raft Engine 作为 TiKV 的日志存储引擎（实验特性）

- 优化备份对集群的影响
- 支持 Azure Blob Storage 作为备份目标存储
- 持续提升 TiFlash 列式存储引擎和 MPP 计算引擎的稳定性和性能
- 为 TiDB Lightning 增加已存在数据表是否允许导入的开关
- 优化持续性能分析（实验特性）
- TiSpark 支持用户认证与鉴权

16.13.4.1 兼容性变化

> **注意：**
>
> 当从一个早期的 TiDB 版本升级到 TiDB v5.4.0 时，如需了解所有中间版本对应的兼容性更改说明，请查看对应版本的Release Notes。

16.13.4.1.1 系统变量

变量名	修改类型	描述
tidb_enable_column_tracking ↪	新增	用于控制是否开启 TiDB 对 PREDICATE ↪ COLUMNS 的收集，默认值为 OFF。

变量名	修改类型	描述
tidb_enable_paging ↪	新增	此变量用于控制 IndexLookUp 算子是否使用分页 (paging) 方式发送 Coprocessor 请求，默认值为 OFF。对于使用 IndexLookUp 和 Limit 并且 Limit 无法下推到 IndexScan 上的读请求，可能会出现读请求的延迟高、TiKV 的 Unified read pool CPU 使用率高的情况。在这种情况下，由于 Limit 算子只需要少部分数据，开启 tidb_enable_paging ↪ ，能够减少处理数据的数量，从而降低延迟、减少资源消耗。
tidb_enable_top_ ↪	新增	用于控制是否开启 Top SQL 特性，默认值为 OFF。
tidb_persist_ana _options ↪	新增	用于控制是否开启ANALYZE 配置持久化特性，默认值为 ON。
tidb_read_stalen ↪	新增	用于设置当前会话允许读取的历史数据范围，默认值为 0。
tidb_regard_null point ↪	新增	用于控制优化器是否可以将包含 null 的等值条件作为前缀条件来访问索引。

变量名	修改类型	描述
tidb_stats_load_sync_wait	新增	这个变量用于控制是否开启统计信息的同步加载模式（默认为 0 代表不开启，即为异步加载模式），以及开启的情况下，SQL 执行同步加载完整统计信息等待多久后会超时。
tidb_stats_load_pseudo_timeout	新增	用于控制统计信息同步加载超时后，SQL 是执行失败 (OFF) 还是退回使用 pseudo 的统计信息 (ON)，默认值为 OFF。
tidb_backoff_lock_fast	修改	默认值由 100 修改为 10。
tidb_enable_index_merge	修改	默认值由 OFF 改为 ON。如果从低于 v4.0.0 版本升级到 v5.4.0 及以上版本的集群，该变量值默认保持 OFF。如果从 v4.0.0 及以上版本升级到 v5.4.0 及以上版本的集群，该变量开关保持升级前的状态。对于 v5.4.0 及以上版本的新建集群，该变量开关默认保持 ON。
tidb_store_limit	修改	v5.4.0 前支持实例级别及集群级别的设置，现在只支持集群级别的设置。

16.13.4.1.2 配置文件参数

配置文件	配置项	修改类型	描述
TiDB	stats-load-concurrency	新增	用于设置 TiDB 统计信息同步加载功能可以并发处理的最大列数，默认值为 5。
TiDB	stats-load-queue-size	新增	用于设置 TiDB 统计信息同步加载功能最多可以缓存多少列的请求，默认值为 1000。
TiKV	snap-generator-pool-size	新增	配置 snap-generator 线程池大小，默认值为 2。
TiKV	log.file.max-size、log.file.max-days、log.file.max-backups	新增	参数说明见TiKV 配置文件 - log.file。

配置文件	配置项	修改类型	描述
TiKV	raft-engine	新增	包含 enable、dir、batch-compression-threshold、bytes-per-sync、target-file-size、purge-threshold、recovery-mode、recovery-read-block-size、recovery-read-block-size、recovery-threads，详情参见TiKV 配置文件：raft-engine。
TiKV	backup.enable-auto-tune	修改	在 v5.3.0 中默认值为 false，自 v5.4.0 起默认值改为 true。表示在集群资源占用率较高的情况下，是否允许 BR 自动限制备份使用的资源，减少对集群的影响。在默认配置下，备份速度可能下降。

配置文件	配置项	修改类型	描述
TiKV	log-level、log-format、log-file、log-rotation-↪ size	修改	将 TiKV log 参数名替换为与 TiDB log 参数一致的参数名，即 log.level、log.format、log.file.↪ filename、log.enable-↪ timestamp。如果只设置了原参数、且把其值设为非默认值，原参数与新参数会保持兼容；如果同时设置了原参数和新参数，则会使用新参数。详情参见TiKV 配置文件 - log。
TiKV	log-rotation-↪ timespan	删除	轮换日志的时间跨度。当超过该时间跨度，日志文件会被轮换，即在当前日志文件的文件名后附加一个时间戳，并创建一个新文件。
TiKV	allow-remove-↪ leader	删除	决定是否允许删除主开关。
TiKV	raft-msg-↪ flush-↪ interval	删除	Raft 消息攒批发出的间隔时间。每隔该配置项指定的间隔，Raft 消息会攒批发出。

配置文件	配置项	修改类型	描述
PD	log.level	修改	默认值由“INFO”改为“info”，保证大小写不敏感。
TiFlash	profile.↪ default.↪ enable_elastic_threadpool↪	新增	表示是否启用可自动扩展的线程池。打开该配置项可以显著提高 TiFlash 在高并发场景的 CPU 利用率。默认值为 false。
TiFlash	storage.↪ format_version↪	新增	表示 DTFile 储存文件格式，默认值为 2，该格式在数据文件中内嵌哈希值。也可以设置为 3，该格式包含元数据，标记数据校验，支持多种哈希算法。
TiFlash	logger.count	修改	默认值修改为 10。
TiFlash	status.↪ metrics_port↪	修改	默认值修改为 8234。
TiFlash	raftstore.↪ apply-pool↪ -size	新增	处理 Raft 数据落盘的线程池中线程的数量，默认值为 4。
TiFlash	raftstore.↪ store-pool↪ -size	新增	处理 Raft 的线程池中线程的数量，即 Raftstore 线程池的大小，默认值为 4。

配置文件	配置项	修改类型	描述
TiDB Data Migration (DM)	collation_compat ↪	新增	同步 CREATE 语句中缺省 Collation 的方式，可选“loose”和“strict”，默认为“loose”。
TiCDC	max-message- ↪ bytes	修改	将 Kafka sink 模块的 max-message- ↪ bytes 默认值设置为 104857601 (10MB)。
TiCDC	partition-num	修改	将 Kafka Sink partition-num 的默认值改由 4 为 3，使 TiCDC 更加平均地分发消息到各个 Kafka partition。
TiDB Lightning	meta-schema- ↪ name	修改	此配置项控制 TiDB Lightning 在目标 TiDB 中保存 metadata 对应的 schema name。从 v5.4.0 开始，只在开启了并行导入功能时 (对应配置为 tikv- ↪ importer. ↪ incremental ↪ -import = ↪ true)，才会在目标 TiDB 中创建此库。

配置文件	配置项	修改类型	描述
TiDB Lightning	`task-info-schema-name`	新增	用于配置当 TiDB Lightning 检测到冲突数据时，对应冲突数据存储的库名，默认值为“lightning_task_info”。如果没有开启冲突检测功能，则无需配置此参数。
TiDB Lightning	`incremental-import`	新增	是否允许向已存在数据的表导入数据。默认值为 false。

16.13.4.1.3 其他

- 为 TiDB 和 PD 之间新增接口。使用 `information_schema.TIDB_HOT_REGIONS_HISTORY` 系统表时，TiDB 需要使用匹配的 PD 版本。
- 对 log 相关参数，TiDB Server、PD Server 和 TiKV Server 将采用统一的参数命名方式来管理日志命名、输出格式、轮转和过期的规则。参见TiKV 配置文件 - log。
- 自 v5.4.0 起，对于通过 Plan Cache 已经缓存的执行计划，如果为其创建绑定 (Binding)，会使得对应查询已经缓存的计划失效。v5.4.0 前已经缓存的计划不受新 Binding 的影响。
- 在 v5.3 及更早版本中，TiDB Data Migration (DM) 文档独立于 TiDB 文档。自 v5.4 起，TiDB Data Migration 的文档已合并入相同版本的 TiDB 文档，无需跳转到 DM 文档站，你可以直接在 TiDB 文档站阅读DM 文档。
- 移除 cdclog。自 v5.4.0 起，不再支持 cdclog。
- 改进了给系统变量设置为字符串 “DEFAULT” 的行为，以便与 MySQL 更兼容 #29680
- 将系统变量 `lc_time_names` 改成只读 #30084
- 将系统变量 `tidb_store_limit` 的作用域从 INSTANCE 或 GLOBAL 更改为 GLOBAL #30756
- 当列数据中有零时，禁止列从整型类型转成时间类型 #25728
- 修复插入浮点值时对 `Inf` 和 NaN 值不报错问题 #30148
- 修复了当 Auto ID 超出范围时，REPLACE 语句更改其他行 ID 值的问题 #30301

16.13.4.2 新功能

16.13.4.2.1 SQL

- TiDB 从 v5.4.0 起支持 GBK 字符集

  在 v5.4.0 前，TiDB 支持 `ascii`、`binary`、`latin1`、`utf8` 和 `utf8mb4` 字符集。

为了更好的支持中文用户，TiDB 从 v5.4.0 起支持 GBK 字符集。在初次初始化 TiDB 集群时开启 TiDB 配置项new_collations_enabled_on_first_bootstrap 后，TiDB GBK 字符集同时支持 gbk_bin 和 gbk_chinese_ci 这两种排序规则。

在使用 GBK 字符集时，需要注意兼容性限制。

用户文档

16.13.4.2.2 安全

- TiSpark 支持用户认证与鉴权

  TiSpark 提供数据库和表级别的读写授权验证以及数据库用户认证验证。开启该功能后，能避免业务侧未经授权运行抽数等批量任务获取数据，提高线上集群的稳定性和数据安全性。从 TiSpark v2.5.0 起开始支持。

  该功能默认关闭。开启后，如果用户没有对应的权限，通过 TiSpark 操作会抛出对应的异常。

  用户文档

- TiUP 部署方式支持为 root 用户生成初始密码

  集群启动命令增加了 `--init` 参数，有了该参数，在 TiUP 部署场景，TiUP 会为数据库 root 用户生成一个初始的强密码，避免 root 用户使用空密码所带来的安全风险，增强数据库的安全性。

  用户文档

16.13.4.2.3 性能

- 持续提升 TiFlash 列式存储引擎和 MPP 计算引擎的稳定性和性能

  - 支持将更多函数下推至 MPP 引擎
    * 字符串函数：`LPAD()`、`RPAD()`、`STRCMP()`
    * 日期时间函数：`ADDDATE(string, real)`、`DATE_ADD(string, real)`、`DATE_SUB(string, real)`、`SUBDATE(string, real)`、`QUARTER()`
  - 引入可扩缩容弹性线程池，提升资源利用率（实验特性）
  - 提升从 TiKV 同步数据时，由行存格式到列存格式的数据转换效率，整体的数据同步性能提升 50%
  - 调整一些配置项的默认值，提升 TiFlash 的性能和稳定性。HTAP 混合负载下，单表简单查询的性能最高提升 20%

  用户文档：TiFlash 支持的计算下推，TiFlash 配置文件

- 通过 session 变量实现有界限过期数据读取

  TiDB 是基于 Raft 协议的多副本分布式数据库。面对高并发，高吞吐业务场景，可以通过 follower 节点实现读性能扩展，构建读写分离架构。

  针对不同的业务场景，follower 提供强一致读和弱一致过期读两种读模式。强一致读模式可以满足数据实时性要求严格的业务场景。然而，当采用该模式时，特别是在跨机房架构下，由于会出现 leader 和 follower 的数据同步延迟、吞吐量降低等情况，会出现延迟问题。

  在对数据实时性要求不高的业务场景下，可以选择过期读模式。使用该模式可以降低延迟和提升吞吐。TiDB 目前支持通过以下方式实现过期读：使用 SQL 语句读取一个基于历史时间的数据和开启基于历史

时间的只读事务。通过这两种方式，你均可以从指定时间点或时间范围内读取对应的历史数据。具体用法，请参考使用 AS OF TIMESTAMP 语法读取历史数据。

从 v5.4.0 版本开始 TiDB 支持通过 session 变量设置有界限过期读，进一步提升易用性，满足准实时场景下低延迟高吞吐数据访问的业务诉求。具体设置示例如下：

```
set @@tidb_replica_read=leader_and_follower
set @@tidb_read_staleness="-5"
```

通过该设置，TiDB 可以实现就近选取 leader 或 follower 节点，并读取 5 秒钟内的最新过期数据。

用户文档

- TiDB 正式发布索引合并功能

  索引合并 (Index Merge) 是在 TiDB v4.0 版本中作为实验特性引入的一种查询执行方式的优化，可以大幅提高查询在扫描多列数据时条件过滤的效率。例如对以下的查询，若 WHERE 子句中两个 OR 连接的过滤条件在各自包含的 key1 与 key2 两个列上都存在索引，则索引合并可以同时利用 key1 与 key2 上的索引分别进行过滤，然后合并出最终的结果。

  ```
  SELECT * FROM table WHERE key1 <= 100 OR key2 = 200;
  ```

  以往 TiDB 在一个表上的查询只能使用一个索引，无法同时使用多个索引进行条件过滤。相较以往，索引合并避免了此情况下可能不必要的大量数据扫描，也可以使得需要灵活查询不特定多列数据组合的用户利用单列上的索引达到高效稳定的查询，无需大量构建多列复合索引。

  本版本正式发布了索引合并特性，但仍存在以下的使用条件和限制：

  - 目前 TiDB 的索引合并优化只限于析取范式 (X1 □ X2 □ ⋯Xn)，即 WHERE 子句中过滤条件连接词为 OR。
  - 如果全新部署的集群版本为 v5.4.0 或以上，此特性默认开启。如果从 v5.4.0 以前的版本升级到 v5.4.0 或以上，默认保持升级前此特性的开关状态（v4.0.0 之前无此项特性的版本默认关闭），由用户决定是否开启。

  用户文档

- 新增 Raft Engine（实验特性）

  支持使用 Raft Engine 作为 TiKV 的日志存储引擎。与使用 RocksDB 相比，Raft Engine 可以减少至多 40% 的 TiKV I/O 写流量和 10% 的 CPU 使用，同时在特定负载下提升 5% 左右前台吞吐，减少 20% 尾延迟。此外，Raft Engine 提升了日志回收效率，修复了极端条件下日志堆积的问题。

  Raft Engine 目前仍属于实验特性，并默认关闭。另外请注意 v5.4.0 版本的 Raft Engine 数据格式与之前版本不兼容，对集群做升级操作之前，需要确保所有 TiKV 节点上的 Raft Engine 已被关闭。只建议在 v5.4.0 及以后的版本使用 Raft Engine。

  用户文档

- 支持收集 PREDICATE COLUMNS 的统计信息（实验特性）

  执行 SQL 语句时，优化器在大多数情况下只会用到部分列（例如，WHERE、JOIN、ORDER BY、GROUP BY 子句中出现的列）的统计信息，这些用到的列称为 PREDICATE COLUMNS。

  从 v5.4.0 开始，你可以设置系统变量tidb_enable_column_tracking 的值为 ON 开启 TiDB 对 PREDICATE COLUMNS 的收集。

开启后，TiDB 将每隔 100 * stats-lease 时间将 PREDICATE COLUMNS 信息写入系统表 mysql.column_stats_usage。等到业务的查询模式稳定以后，使用 ANALYZE TABLE TableName PREDICATE COLUMNS 语法收集 PREDICATE COLUMNS 列的统计信息，可以极大地降低收集统计信息的开销。

用户文档

- 支持统计信息的同步加载（实验特性）

  从 v5.4.0 开始，TiDB 引入了统计信息同步加载的特性（默认关闭），支持执行当前 SQL 语句时将直方图、TopN、CMSketch 等占用空间较大的统计信息同步加载到内存，提高该 SQL 语句优化时统计信息的完整性。

  用户文档

16.13.4.2.4 稳定性

- 支持统计信息采集配置持久化

  统计信息是优化器生成执行计划时所参考的基础信息之一，统计信息的准确性直接影响生成的执行计划是否合理。为了保证统计信息的准确性，有时候需要针对不同的表、分区、索引设置不同的采集配置项。

  TiDB 从 v5.4.0 版本开始支持 ANALYZE 配置持久化功能，方便后续收集统计信息时沿用已有配置项。

  ANALYZE 配置持久化功能默认开启（系统变量 tidb_analyze_version 为默认值 2，tidb_persist_analyze_options 为默认值 ON），用于记录手动执行 ANALYZE 语句时指定的持久化配置项。记录后，当 TiDB 下一次自动更新统计信息或者你手动收集统计信息但未指定配置项时，TiDB 会按照记录的配置项收集统计信息。

  用户文档

16.13.4.2.5 高可用和容灾

- 优化备份对集群的影响

  Backup & Restore (BR) 增加了备份线程自动调节功能（默认开启）。该功能通过监控集群资源的使用率自动调节备份的线程数的方式，降低备份过程对集群的影响。在某些 Case 验证中，通过增加集群用于备份的资源和开启备份线程自动调节功能，备份的影响可以降低到 10% 以下。

  用户文档

- 支持 Azure Blob Storage 作为备份目标存储

  Backup & Restore (BR) 支持 Azure Blob Storage 作为备份的远端目标存储。在 Azure Cloud 环境部署 TiDB 的用户，可以支持使用该功能将集群数据备份到 Azure Blob Storage 服务中。

  用户文档

16.13.4.2.6 数据迁移

- 为 TiDB Lightning 增加已存在数据表是否允许导入的开关

  为 TiDB Lightning 增加 incremental-import 开关。默认值为 false，表明目标表已存在数据时将不会执行导入。将默认值改为 true 则继续导入。注意，当使用并行导入特性时，需要将该配置项设为 true。

  用户文档

- 增加新配置允许自定义用于保存 TiDB Lightning 并行导入特性的元信息的 schema 名称

  为 TiDB Lightning 增加 `meta-schema-name` 配置。在并行导入模式下，该参数用于在目标集群保存各个 TiDB Lightning 实例元信息的 schema 名称，默认值为 “lightning_metadata”。对于参与同一批并行导入的每个 TiDB Lightning 实例，必须将此配置项设置为相同的值，否则将无法确保导入数据的正确性。

  用户文档

- 在 TiDB Lightning 中添加重复数据的检测

  在 `backend=local` 模式下，数据导入完成之前 TiDB Lightning 会输出冲突数据，然后从数据库中删除这些冲突数据。用户可以在导入完成后解析冲突数据，并根据业务规则选择适合的数据进行插入。建议根据冲突数据清洗上游数据源，避免在后续增量数据迁移阶段遇到冲突数据而造成数据不一致。

  用户文档

- 在 TiDB Data Migration (DM) 中优化 relay log 的使用方式

  - 恢复 `source` 配置中 `enable-relay` 开关。
  - 增加通过 `start-relay` 和 `stop-relay` 命令动态开启和关闭 relay log 的功能。
  - relay log 的开启状态与 `source` 绑定，source 迁移到任意 DM-worker 均保持原有开启或关闭状态。
  - relay log 的存放路径移至 DM-worker 配置文件。

  用户文档

- 在 DM 中优化排序规则的处理方式

  增加 `collation_compatible` 开关，支持 strict 和 loose（默认）两种模式：

  - 如果对排序规则要求不严格，允许上下游查询结果排序规则不一致，使用默认的 loose 模式可以避免报错。
  - 如果对排序规则要求严格，业务要求排序规则必须一致，则应当使用 strict 模式。但如果下游不支持上游缺省的 collation，同步可能会报错。

  用户文档

- 在 DM 中优化 `transfer source`，支持平滑执行同步任务

  当 DM-worker 所在各节点负载不均衡时，`transfer source` 命令可用于手动将某 `source` 配置迁移到其他节点。优化后的 `transfer source` 简化了用户操作步骤，不再要求先暂停所有关联 task 而是直接执行平滑迁移，DM 将在内部完成所需操作。

- DM OpenAPI 特性 GA

  DM 支持通过 API 的方式进行日常管理，包括增加数据源、管理任务等。本次更新 OpenAPI 从实验特性转为正式特性。

  用户文档

16.13.4.2.7 问题诊断效率

- Top SQL（实验特性）

  新推出实验性特性 Top SQL（默认关闭），帮助用户轻松找到节点中消耗负载较大的查询。

  用户文档

16.13.4.2.8 TiDB 数据共享订阅

- 优化 TiCDC 对集群的影响

  大幅降低了 TiCDC 启用后，对 TiDB 集群的性能影响。在实验室环境中，TiCDC 对 TiDB 的性能影响可以降低到 5% 以下。

16.13.4.2.9 部署及运维

- 增强持续性能分析（实验特性）
  - 支持更多组件：除了 TiDB、PD 和 TiKV 外，v5.4.0 版本中还支持查看 TiFlash CPU Profiling。
  - 支持更方便的查看形式：支持以火焰图形式查看 CPU Profiling 和 Goroutine 结果。
  - 支持更多部署环境：支持在 TiDB Operator 部署环境下启用持续性能分析功能。

  该功能默认关闭，需进入 TiDB Dashboard 持续性能分析页面开启。

  持续性能分析仅支持由 v1.9.0 及以上版本 TiUP 或 v1.3.0 及以上版本 TiDB Operator 升级或安装的集群。

  用户文档

16.13.4.3 提升改进

- TiDB
  - 支持 `ADMIN {SESSION | INSTANCE | GLOBAL} PLAN_CACHE` 语法，用于清空缓存的查询计划 #30370
- TiKV
  - Coprocessor 支持分页 API 进行流式处理 #11448
  - 支持 `read-through-lock`，使读操作不需要等待清理 secondary lock #11402
  - 增加了磁盘保护机制，尽量避免磁盘空间耗尽导致 panic #10537
  - 日志支持存档和轮替 #11651
  - 减少 Raft 客户端的系统调用并提高 CPU 效率 #11309
  - Coprocessor 支持下推 substring 到 TiKV #11495
  - 通过跳过读锁提高在 RC 隔离级别中扫描的性能 #11485
  - 减少备份使用的默认线程池大小，并在压力大时限制其使用 #11000
  - 支持动态调整 Apply 和 Store 线程池大小 #11159
  - 支持配置 `snap-generator` 线程池大小 #11247
  - 优化在文件数较多且读写频繁的场景下 RocksDB 的全局锁争用问题 #250
- PD
  - 默认开启历史热点记录功能 #25281
  - 新增 HTTP Component 的签名，用于标识请求来源 #4490
  - TiDB Dashboard 更新至 v2021.12.31 #4257
- TiFlash
  - 优化本地算子通讯

  - 调高 gRPC 非临时线程数，避免频繁创建/销毁线程

- Tools

  - Backup & Restore (BR)

    * 增加 BR 加密备份时，对密钥的合法性检查 #29794

  - TiCDC

    * 减少 “EventFeed retry rate limited” 日志的数量 #4006
    * 降低在同步大量表时的同步延时 #3900
    * 减少 TiKV 节点宕机后 KV client 恢复的时间 #3191

  - TiDB Data Migration (DM)

    * 降低开启 relay 时的 CPU 使用率 #2214

  - TiDB Lightning

    * 在 TiDB-backend 模式下，默认改用乐观事务进行写入来提升性能 #30953

  - Dumpling

    * 提升 Dumpling 检查数据库版本时的兼容性 #29500
    * 在导出 `CREATE DATABASE` 和 `CREATE TABLE` 时添加默认的 collation #3420

16.13.4.4 Bug 修复

- TiDB

  - 修复当从 v4.x 版本升级到 v5.x 版本后 `tidb_analyze_version` 的值变化的问题 #25422
  - 修复在子查询中使用不同的 collation 导致查询结果错误的问题 #30748
  - 修复 `concat(ifnull(time(3))` 的结果与 MySQL 不一致的问题 #29498
  - 修复乐观事务下数据索引可能不一致的问题 #30410
  - 修复当表达式不能下推给 TiKV 时 IndexMerge 查询计划错误的问题 #30200
  - 修复并发的列类型变更导致 schema 与数据不一致的问题 #31048
  - 修复当有子查询时 IndexMerge 的查询结果错误的问题 #30913
  - 修复当客户端设置过大的 FetchSize 后 TiDB 执行会 panic 的问题 #30896
  - 修复 LEFT JOIN 有时会被错误地转换成 INNER JOIN 的问题 #20510
  - 修复当 `CASE-WHEN` 表达式与 collation 同时使用时可能 panic 的问题 #30245
  - 修复当 `IN` 的值中带有二进制常量时查询结果错误的问题 #31261
  - 修复当 CTE 中带有子查询时查询结果错误的问题 #31255
  - 修复 `INSERT ... SELECT ... ON DUPLICATE KEY UPDATE` 语句 panic 的问题 #28078
  - 修复 INDEX HASH JOIN 报 `send on closed channel` 的问题 #31129

- TiKV

  - 修复 MVCC 删除记录可能不会被 GC 删除的问题 #11217
  - 修复悲观事务中 prewrite 请求重试在极少数情况下影响数据一致性的风险 #11187
  - 修复 GC 扫描导致的内存溢出 #11410
  - 修复当达到磁盘容量满时 RocksDB flush 或 compaction 导致的 panic #11224

- PD
  - 修复 Region 统计不受 `flow-round-by-digit` 影响的问题 #4295
  - 修复调度 Operator 因为目标 Store 处于 Down 的状态而无法快速失败的问题 #3353
  - 修复不能 Merge 在 Offline Store 上面的 Region 的问题 #4119
  - 修复热点统计中无法剔除冷热点数据的问题 #4390
- TiFlash
  - 修复当 MPP 查询被终止时，TiFlash 偶发的崩溃问题
  - 修复 `where <string>` 查询结果出错的问题
  - 修复整数类型主键的列类型设置为较大范围后数据可能不一致的问题
  - 修复输入早于 1970-01-01 00:00:01 UTC 时，`unix_timestamp` 行为与 TiDB/MySQL 不一致的问题
  - 修复 TiFlash 重启时偶发的 `EstablishMPPConnection` 错误
  - 修复在 TiFlash 与 TiDB/TiKV 之间 `CastStringAsDecimal` 行为不一致的问题
  - 修复查询报错 `DB::Exception: Encode type of coprocessor response is not CHBlock`
  - 修复在 TiFlash 与 TiDB/TiKV 之间 `castStringAsReal` 行为不一致的问题
  - 修复 TiFlash 的 `date_add_string_xxx` 函数返回值与 MySQL 不一致的问题
- Tools
  - Backup & Restore (BR)
    * 修复当恢复完成后，Region 有可能分布不均的问题 #30425
    * 修复当使用 `minio` 作为备份存储时，不能在 endpoint 中指定 `'/'` 的问题 #30104
    * 修复因为并发备份系统表，导致表名更新失败，无法恢复系统表的问题 #29710
  - TiCDC
    * 修复当 `min.insync.replicas` 小于 `replication-factor` 时不能同步的问题 #3994
    * 修复 `cached region` 监控指标为负数的问题 #4300
    * 修复 `mq sink write row` 没有监控数据的问题 #3431
    * 修复 `sql mode` 兼容性问题 #3810
    * 修复在移除同步任务后潜在的 panic 问题 #3128
    * 修复输出默认列值时的 panic 问题和数据不一致的问题 #3929
    * 修复不支持同步默认值的问题 #3793
    * 修复潜在的同步流控死锁问题 #4055
    * 修复在磁盘写满时无日志输出的问题 #3362
    * 修复 DDL 特殊注释导致的同步停止的问题 #3755
    * 修复在某些 RHEL 发行版上因时区问题导致服务无法启动的问题 #3584
    * 修复因 checkpoint 不准确导致的潜在的数据丢失问题 #3545
    * 修复在容器环境中 OOM 的问题 #1798
    * 修复 `config.Metadata.Timeout` 没有正确配置而导致的同步停止问题 #3352
  - TiDB Data Migration (DM)
    * 修复 `CREATE VIEW` 语句中断复制任务的问题 #4173
    * 修复 skip DDL 后需要重置 Schema 的问题 #4177
    * 修复 skip DDL 后未及时更新表检查点的问题 #4184
    * 修复 TiDB 和 Parser 版本兼容问题 #4298

    * 修复部分 syncer metrics 只有在查询状态时才得以更新的问题 #4281

  - TiDB Lightning

    * 修复当 TiDB Lightning 没有权限访问 `mysql.tidb` 表时，导入的结果不正确的问题 #31088
    * 修复 TiDB Lightning 重启时，跳过某些检查的问题 #30772
    * 修复当 S3 路径不存在时，TiDB Lightning 没有及时报错的问题 #30674

  - TiDB Binlog

    * 修复 Drainer 不兼容 `CREATE PLACEMENT POLICY` 语句导致处理失败的问题 #1118

16.14 v5.3

16.14.1 TiDB 5.3.4 Release Notes

发版日期：2022 年 11 月 24 日

TiDB 版本：5.3.4

16.14.1.1 提升改进

- TiKV

  - 当 TLS 证书更新时自动重新加载，以提升可用性 #12546

16.14.1.2 Bug 修复

- TiDB

  - 修复 Region 合并情况下 Region cache 没有及时被清理的问题 #37141
  - 修复 `ENUM` 或 `SET` 类型的列因为编码错误导致写入数据错误的问题 #32302
  - 修复数据库级别的权限清理不正确的问题 #38363
  - 修复 `mysql.tables_priv` 表中 `grantor` 字段缺失的问题 #38293
  - 修复 `KILL TIDB` 在空闲链接上无法立即生效的问题 #24031
  - 修复 `date_add` 和 `date_sub` 函数返回类型的行为与 MySQL 不一致的问题 #36394, #27573
  - 修复 Parser 恢复 `table option` 中 `INSERT_METHOD` 字段错误的问题 #38368
  - 修复 MySQL 5.1 及之前版本客户端连接 TiDB Server 时鉴权失败的问题 #29725
  - 修复当 `GREATEST` 和 `LEAST` 函数传入无符号整型值时，计算结果出错的问题 #30101
  - 修复 `concat(ifnull(time(3))` 的结果与 MySQL 不一致的问题 #29498
  - 修复当从 TiFlash 查询 `avg()` 函数时，返回错误 `ERROR 1105 (HY000): other error for mpp stream ↪ : Could not convert to the target type - -value is out of range.` 的问题 #29952
  - 修复查询 HashJoin 时，返回错误 `ERROR 1105 (HY000): close of nil channel` 的问题 #30289
  - 修复 TiKV 和 TiFlash 在进行逻辑运算时结果不一致的问题 #37258
  - 修复带 DML 算子的 `EXPLAIN ANALYZE` 语句可能在事务提交完成前返回结果的问题 #37373
  - 修复合并多个 Region 后 Region cache 没有正确清理的问题 #37174
  - 修复某些情况下，`EXECUTE` 语句可能抛出非预期异常的问题 #37187
  - 修复当 `ORDER BY` 子句里包含关联子查询时与 `GROUP CONCAT` 一起执行可能会导致出错的问题 #18216

  - 修复使用 Plan Cache 时，由于 Decimal 和 Real 的 length 和 width 设置错误而导致的结果出错问题 #29565

- PD

  - 修复 PD 无法正确处理 dashboard 代理请求的问题 #5321
  - 修复 PD 在特定条件下不会创建 TiFlash Learner 副本的问题 #5401
  - 修复 Stream 超时问题，提高 Leader 切换的速度 #5207

- TiFlash

  - 修复逻辑运算符在 UInt8 类型下查询结果出错的问题 #6127
  - 修复由于使用 0.0 作为整数类型的默认值导致 TiFlash 节点失败的问题，比如`i` int(11)NOT NULL
    ↪ DEFAULT '0.0' #3157

- Tools

  - Dumpling

    * 修复 Dumpling 同时指定 --compress 配置和 S3 导出目录时无法导出数据的问题 #30534

  - TiCDC

    * 修复由于没有及时上报 MySQL 相关错误导致同步任务状态不正确的问题 #6698

16.14.2 TiDB 5.3.3 Release Note

发版日期：2022 年 9 月 14 日

TiDB 版本：5.3.3

16.14.2.1 Bug 修复

- TiKV

  - 修复了 PD leader 发生切换或重启 PD 后，在集群中执行 SQL 语句会出现持续报错的问题。

    * 问题原因：该问题是由于 TiKV 存在 bug，TiKV 向 PD client 发送心跳请求失败后不会重试，只能等待与 PD client 重连。这样，故障 TiKV 节点上的 Region 的信息会逐步变旧，使得 TiDB 无法获取最新的 Region 信息，导致 SQL 执行出错。
    * 影响版本：v5.3.2 和 v5.4.2。目前该问题已在 v5.3.3 上修复。如果你使用 v5.3.2 的 TiDB 集群，可以升级至 v5.3.3。
    * 规避方法：除升级外，你还可以重启无法向 PD 发送 Region 心跳的 TiKV 节点，直至不再有待发送的 Region 心跳为止。

    Bug 详情参见 #12934。

16.14.3 TiDB 5.3.2 Release Notes

发版日期：2022 年 6 月 29 日

TiDB 版本：5.3.2

> **警告：**
>
> 不建议使用 v5.3.2，因为该版本已知存在 bug，详情参见 #12934。该 bug 已在 v5.3.3 中修复，建议升级至v5.3.3。

16.14.3.1 兼容性变更

- TiDB
  - 修复当 auto ID 超出范围时，REPLACE 语句错误地修改了其它行的问题 #29483
- PD
  - 默认关闭编译 swagger server #4932

16.14.3.2 提升改进

- TiKV
  - 减少 Raft 客户端的系统调用并提高 CPU 效率 #11309
  - 健康检查可以检测到无法正常工作的 Raftstore，使得 TiKV client 可以及时更新 Region Cache #12398
  - 通过将 leader 转让给 CDC observer 减少延迟抖动 #12111
  - 在 Raft 日志垃圾回收模块中添加了更多监控指标，从而定位该模块中出现的性能问题 #11374
- Tools
  - TiDB Data Migration (DM)
    - 支持 Syncer 使用 DM-worker 的工作目录写内部文件，不再使用 `/tmp` 目录。任务停止后会清理掉该目录 #4107
  - TiDB Lightning
    - 优化 Scatter Region 为批量模式，提升 Scatter Region 过程的稳定性 #33618

16.14.3.3 Bug 修复

- TiDB
  - 修复了 Amazon S3 无法正确计算压缩数据大小的问题 #30534
  - 修复乐观事务下数据索引可能不一致的问题 #30410
  - 修复当 SQL 语句中存在 JSON 类型列与 CHAR 类型列连接时，SQL 出错的问题 #29401
  - 如果发生网络连接问题，TiDB 并不总是能正确释放已断开会话所占有的资源。该修复可以确保 TiDB 回滚打开的事务以及释放其他相关资源。#34722
  - 修复开启 TiDB Binlog 后插入重复数据导致 `data and columnID count not match` 错误的问题 #33608
  - 修复在 RC 隔离情况下 Plan Cache 启用时可能导致查询结果错误的问题 #34447

  - 修复了在 MySQL binary 协议下，当 schema 变更后，执行 prepared statement 会导致会话崩溃的问题 #33509
  - 修复对于新加入的分区，表属性 (table attributes) 无法被检索到，以及分区更新后，表的 range 信息不会被更新的问题 #33929
  - 修复了查询 `INFORMATION_SCHEMA.CLUSTER_SLOW_QUERY` 表导致 TiDB 服务器 OOM 的问题，在 Grafana dashboard 中查看慢查询记录的时候可能会触发该问题 #33893
  - 修复集群的 PD 节点被替换后一些 DDL 语句会卡住一段时间的问题 #33908
  - 修复了集群从 4.0 版本升级后，为用户授予 `all` 权限时报错的问题 #33588
  - 修复使用 left join 同时删除多张表数据时可能出现错误结果的问题 #31321
  - 修复 TiDB 可能向 TiFlash 发送重复任务的问题 #32814
  - 修复 TiDB 的后台 HTTP 服务可能没有正确关闭导致集群状态异常的问题 #30571
  - 修复 TiDB 由于 `fatal error: concurrent map read and map write` 发生崩溃的问题 #35340

- TiKV

  - 修复了 PD 客户端遇到报错时频繁重连的问题 #12345
  - 修复了 `DATETIME` 类型的数据包含小数部分和 `Z` 后缀导致检查报错的问题 #12739
  - 修复了对空字符串进行类型转换导致 TiKV panic 的问题 #12673
  - 修复了在悲观事务中使用 Async Commit 导致重复提交记录的问题 #12615
  - 修复进行 Follower Read 时，可能会报 `invalid store ID 0` 错误的问题 #12478
  - 修复销毁 peer 和批量分裂 Region 之间的竞争导致的 TiKV panic #12368
  - 修复网络出现问题的情况下，已成功提交的乐观事务可能报 `Write Conflict` 错误的问题 #34066
  - 修复待 merge 的 Region 无效会导致 TiKV panic 且非预期地销毁 peer 的问题 #12232
  - 修复旧信息造成 TiKV panic 的问题 #12023
  - 修复因内存统计指标溢出而造成的间歇性丢包和内存不足 (OOM) 的问题 #12160
  - 修复在 Ubuntu 18.04 下进行性能分析会造成 TiKV panic 的问题 #9765
  - 修复 tikv-ctl 对 `bad-ssts` 结果字符串进行错误匹配的问题 #12329
  - 修复 replica read 可能违反线性一致性的问题 #12109
  - 修复合并 Region 时因 target peer 被一个未进行初始化就被销毁的 peer 所替换，从而引起 TiKV panic 的问题 #12048
  - 修复 TiKV 运行 2 年以上可能 panic 的问题 #11940

- PD

  - 修复由于 Hot Region 没有 leader 导致 PD Panic 的问题 #5005
  - 修复 PD leader 转移后调度不能立即启动的问题 #4769
  - 修复已清除的 `tombstone store` 信息在切换 PD leader 后再次出现的问题 #4941
  - 修复在某些特殊情况下 TSO fallback 的问题 #4884
  - 修复存在较大空间 Store 时（例如 2T），无法检测满的小空间 Store，从而无法进行平衡调度的问题 #4805
  - 修复 `SchedulerMaxWaitingOperator` 设置为 1 时不产生调度的问题 #4946
  - 修复监控信息中已删除 label 的残留问题 #4825

- TiFlash

  - 修复错误地配置存储目录会导致非预期行为的问题 #4093
  - 修复 TiFlash 节点上遗留了与 Region range 不匹配的数据的问题 #4414

  - 修复在添加一些 NOT NULL 的列时报 TiFlash_schema_error 的问题 #4596
  - 修复由于 commit state jump backward 错误导致 TiFlash 反复崩溃的问题 #2576
  - 修复大量 INSERT 和 DELETE 操作后可能导致 TiFlash 数据不一致的问题 #4956
  - 修复启用本地隧道时取消 MPP 查询可能导致任务永远挂起的问题 #4229
  - 修复 TiFlash 使用远程读时可能会误报集群 TiFlash 版本不一致的问题 #3713
  - 修复 MPP query 会随机碰到 gRPC keepalive timeout 导致 query 失败的问题 #4662
  - 修复 MPP exchange receiver 如果出现大量重试可能会导致 query hang 的问题 #3444
  - 修复将 DATETIME 转换为 DECIMAL 时结果错误的问题 #4151
  - 修复将 FLOAT 类型转换为 DECIMAL 类型可能造成溢出的问题 #3998
  - 修复 json_length 对空字符串可能会报 index out of bounds 错误的问题 #2705
  - 修复极端情况下 decimal 比较结果可能有误的问题 #4512
  - 修复在执行带有 JOIN 的查询遇到错误时可能被挂起的问题 #4195
  - 修复查询语句包含 where <string> 时查询结果出错的问题 #3447
  - 修复 CastStringAsReal 在 TiFlash 的行为与在 TiDB、TiKV 的行为不一致的问题 #3475
  - 修复转换 string 类型为 datetime 类型时，microsecond 结果可能不对的问题 #3556
  - 修复查询存在大量 delete 操作的表时可能报错的问题 #4747
  - 修复 TiFlash 随机报错 “Keepalive watchdog fired” 的问题 #4192
  - 修复 TiFlash 节点上遗留了与 Region range 不匹配的数据的问题 #4414
  - 修复 MPP 任务可能永远泄漏线程的问题 #4238
  - 修复空 segments 在 GC 后无法合并的问题 #4511
  - 修复启用 TLS 时可能导致的崩溃 #4196
  - 修复过期数据回收缓慢的问题 #4146
  - 修复错误地配置存储目录会导致非预期行为的问题 #4093
  - 修复一些异常没有被正确地处理的问题 #4101
  - 修复在读取工作量大时添加列后可能出现的查询错误 #3967
  - 修复 STR_TO_DATE() 函数对微秒前导零的错误解析 #3557
  - 修复 TiFlash 重启时偶发的 EstablishMPPConnection 错误 #3615

- Tools

  - Backup & Restore (BR)

    * 修复增量恢复后在表中插入记录时遇到的重复主键问题 #33596
    * 修复了 BR 或 TiDB Lightning 在异常退出的时候，scheduler 没有重置的问题 #33546
    * 修复增量恢复期间，由于 DDL 查询任务为空导致的报错 #33322
    * 修复恢复过程中 Region 不一致时 BR 重试次数不足的问题 #33419
    * 修复了当恢复操作遇到一些无法恢复的错误时，BR 被卡住的问题 #33200
    * 修复 BR 无法备份 RawKV 的问题 #32607
    * 修复 BR 无法处理 S3 内部错误的问题 #34350

  - TiCDC

    * 修复切换 owner 会导致其 metrics 数据不正确的问题 #4774
    * 修复 redo log manager 提前 flush log 的问题 #5486
    * 修复当一部分表没有被 redo writer 管理时 resolved ts 提前推进的问题 #5486
    * 添加 UUID 作为 redo log file 的后缀以解决文件名冲突引起的数据丢失问题 #5486
    * 修复 MySQL Sink 可能会保存错误的 checkpointTs 的问题 #5107
    * 修复 TiCDC 集群升级后可能会 panic 的问题 #5266

    * 修复在同一节点反复调入调出一张表可能会导致同步任务 (changefeed) 被卡住的问题 #4464
    * 修复了在开启 TLS 后，`--pd` 中设置的第一个 PD 不可用导致 TiCDC 无法启动的问题 #4777
    * 修复当 PD 状态不正常时 OpenAPI 可能会卡住的问题 #4778
    * 修复 Unified Sorter 的 workerpool 稳定性问题 #4447
    * 修复某些情况下序列对象被错误同步的问题 #4552

  - TiDB Data Migration (DM)

    * 修复任务自动恢复后，DM 会占用更多磁盘空间的问题 #3734，#5344
    * 修复在未设置 `case-sensitive: true` 时无法同步大写表的问题 #5255
    * 修复了某些情况下，过滤 DDL 并在下游手动执行会导致同步任务不能自动重试恢复的问题 #5272
    * 修复了在 `SHOW CREATE TABLE` 语句返回的索引中，主键没有排在第一位导致的 DM worker panic 的问题 #5159
    * 修复了当开启 GTID 模式或者任务自动恢复时，可能出现一段时间 CPU 占用高并打印大量日志的问题 #5063
    * 修复重启 DM-master 后 relay log 可能会被关闭的问题 #4803

  - TiDB Lightning

    * 修复由 `auto_increment` 列的数据越界导致 local 模式导入失败的问题 #29737
    * 修复前置检查中没有检查本地磁盘空间以及集群是否可用的问题 #34213
    * 修复了 checksum 报错“GC life time is shorter than transaction duration” #32733

16.14.4 TiDB 5.3.1 Release Notes

发版日期：2022 年 3 月 3 日

TiDB 版本：5.3.1

16.14.4.1 兼容性更改

- Tools

  - TiDB Lightning

    * 将 `regionMaxKeyCount` 的默认值从 1_440_000 调整为 1_280_000，以避免数据导入后出现过多空 Region #30018

16.14.4.2 提升改进

- TiDB

  - 优化用户登录模式匹配的逻辑，增强与 MySQL 的兼容性 #30450

- TiKV

  - 通过减少需要进行清理锁 (Resolve Locks) 步骤的 Region 数量来减少 TiCDC 恢复时间 #11993
  - 通过增加对 Raft log 进行垃圾回收 (GC) 时的 write batch 大小来加快 GC 速度 #11404

  - 将 proc filesystem (procfs) 升级至 0.12.0 版本 #11702

- PD

  - 优化 DR_STATE 文件内容的格式 #4341

- Tools

  - TiCDC

    * 暴露 Kafka producer 配置参数，使之在 TiCDC 中可配置 #4385
    * 当指定 S3 为存储后端时，在 TiCDC 启动前增加预清理逻辑 #3878
    * TiCDC 客户端能在未指定证书名的情况下工作 #3627
    * 修复因 checkpoint 不准确导致的潜在的数据丢失问题 #3545
    * 为 changefeed 重启操作添加指数退避机制 #3329
    * 将 Kafka Sink `partition-num` 的默认值改为 3，使 TiCDC 更加平均地分发消息到各个 Kafka partition #3337
    * 减少 “EventFeed retry rate limited” 日志的数量 #4006
    * 将 `max-message-bytes` 默认值设置为 10M #4041
    * 增加更多 Prometheus 和 Grafana 监控告警参数，包括 `no owner alert`、`mounter row`、`table sink total row` 和 `buffer sink total row` #4054 #1606
    * 减少 TiKV 节点宕机后 KV client 恢复的时间 #3191

  - TiDB Lightning

    * 优化了本地磁盘空间检查失败时前置检查的提示信息 #30395

16.14.4.3 Bug 修复

- TiDB

  - 修复 date_format 对 `'\n'` 的处理与 MySQL 不兼容的问题 #32232
  - 修复 `alter column set default` 错误地修改表定义的问题 #31074
  - 修复开启 `tidb_restricted_read_only` 后 `tidb_super_read_only` 没有自动开启的问题 #31745
  - 修复带有 collation 的 `greatest` 或 `least` 函数结果出错的问题 #31789
  - 修复执行查询时报 MPP task list 为空错误的问题 #31636
  - 修复 innerWorker panic 导致的 index join 结果错误的问题 #31494
  - 修复将 FLOAT 列改为 DOUBLE 列后查询结果有误的问题 #31372
  - 修复查询时用到 index lookup join 导致 `invalid transaction` 报错的问题 #30468
  - 修复针对 `Order By` 的优化导致查询结果有误的问题 #30271
  - 修复 MaxDays 和 MaxBackups 配置项对慢日志不生效的问题 #25716
  - 修复 `INSERT ... SELECT ... ON DUPLICATE KEY UPDATE` 语句 panic 的问题 #28078

- TiKV

  - 修复 Peer 状态为 Applying 时快照文件被删除会造成 Panic 的问题 #11746
  - 修复开启流量控制且显式设置 `level0_slowdown_trigger` 时出现 QPS 下降的问题 #11424
  - 修复 cgroup controller 没有被挂载会造成 Panic 的问题 #11569
  - 修复 TiKV 停止后 Resolved TS 延迟会增加的问题 #11351

  - 修复 GC worker 繁忙后无法执行范围删除（即执行 unsafe_destroy_range 参数）的问题 #11903
  - 修复删除 Peer 可能造成高延迟的问题 #10210
  - 修复 Region 没有数据时 any_value 函数结果错误的问题 #11735
  - 修复删除未初始化的副本可能会造成旧副本被重新创建的问题 #10533
  - 修复在已完成重新选举但没有通知被隔离的 Peer 的情况下执行 Prepare Merge 会导致元数据损坏的问题 #11526
  - 修复协程的执行速度太快时偶尔出现的死锁问题 #11549
  - 修复某个 TiKV 节点停机会导致 Resolved Timestamp 进度落后的问题 #11351
  - 修复 Raft client 中 batch 消息过大的问题 #9714
  - 修复在极端情况下同时进行 Region Merge、ConfChange 和 Snapshot 时，TiKV 会出现 Panic 的问题 #11475
  - 修复逆序扫表时 TiKV 无法正确读到内存锁的问题 #11440
  - 修复当达到磁盘容量满时 RocksDB flush 或 compaction 导致的 panic #11224
  - 修复 tikv-ctl 无法正确输出 Region 相关信息的问题 #11393
  - 修复 TiKV 监控项中实例级别 gRPC 的平均延迟时间不准确的问题 #11299

- PD

  - 修复特定情况下调度带有不需要的 JointConsensus 步骤的问题 #4362
  - 修复对单个 Voter 直接进行降级时无法执行调度的问题 #4444
  - 修复更新副本同步模式的配置时出现的数据竞争问题 #4325
  - 修复特定情况下读锁不释放的问题 #4354
  - 修复当 Region 心跳低于 60 秒时热点 Cache 不能清空的问题 #4390

- TiFlash

  - 修复 cast(arg as decimal(x,y)) 在入参 arg 大于 decimal(x,y) 的表示范围时，计算结果出错的问题
  - 修复开启 max_memory_usage 和 max_memory_usage_for_all_queries 配置项时，TiFlash 崩溃的问题
  - 修复 cast(string as real) 在部分场景下计算结果出错的问题
  - 修复 cast(string as decimal) 在部分场景下计算结果出错的问题
  - 修复在把主键列类型修改为一个范围更大的整型类型时，数据索引可能不一致的问题
  - 修复 in 表达式在形如 (arg0, arg1)in (x,y) 的多个参数的情况下计算结果出错的问题
  - 修复当 MPP 查询被终止时，TiFlash 偶发的崩溃问题
  - 修复 str_to_date 函数在入参以 0 开头时，计算结果出错的问题
  - 修复当查询的过滤条件形如 where <string> 时，计算结果出错的问题
  - 修复 cast(string as datetime) 在入参形如 %Y-%m-%d\n%H:%i:%s 时，查询结果出错的问题

- Tools

  - Backup & Restore (BR)

    * 修复当恢复完成后，Region 有可能分布不均的问题 #31034

  - TiCDC

    * 修复了 varchar 类型值长度过长时的 Column length too big 错误 #4637
    * 修复了 TiCDC 进程在 PD leader 被杀死时的异常退出问题 #4248
    * 修复了 UPDATE 语句在安全模式下执行错误会导致 DM 进程挂掉的问题 #4317
    * 修复 cached region 监控指标为负数的问题 #4300

    * 修复了 HTTP API 在查询的组件不存在时导致 CDC 挂掉的问题 #3840
    * 修复了移除一个暂停的同步时，CDC Redo Log 无法被正确清理的问题 #4740
    * 修复在容器环境中 OOM 的问题 #1798
    * 修复了停止加载中的任务会导致它被意外调度的问题 #3771
    * 纠正了在 Loader 上使用 `query-status` 命令查询到错误的进度的问题 #3252
    * 修复了 HTTP API 在集群中存在不同版本 TiCDC 节点时无法正常工作的问题 #3483
    * 修复了当 TiCDC Redo Log 配置在 S3 存储上时 TiCDC 异常退出问题 #3523
    * 修复不支持同步默认值的问题 #3793
    * 修复 MySQL sink 在禁用 `batch-replace-enable` 参数时生成重复 `replace` SQL 语句的错误 #4501
    * 修复部分 syncer metrics 只有在查询状态时才得以更新的问题 #4281
    * 修复当 Kafka 为下游时 `txn_batch_size` 监控指标数据不准确的问题 #3431
    * 修复当 `min.insync.replicas` 小于 `replication-factor` 时不能同步的问题 #3994
    * 修复当 Kafka 为下游时 `txn_batch_size` 监控指标数据不准确的问题 #3431
    * 修复在移除同步任务后潜在的 panic 问题 #3128
    * 修复潜在的同步流控死锁问题 #4055
    * 修复人为删除 etcd 任务的状态时导致 TiCDC panic 的问题 #2980
    * 修复 DDL 特殊注释导致的同步停止的问题 #3755
    * Kafka sink 模块添加默认的元数据获取超时时间配置 (`config.Metadata.Timeout`) #3352
    * 修复 `cdc server` 命令在 Red Hat Enterprise Linux 系统的部分版本（如 6.8、6.9 等）上运行时出现时区错误的问题 #3584
    * 修复集群升级后 `stopped` 状态的 changefeed 自动恢复的问题 #3473
    * 修复不支持同步默认值的问题 #3793
    * 修复 MySQL sink 模块出现死锁时告警过于频繁的问题 #2706
    * 修复 Canal 和 Maxwell 协议下 TiCDC 没有自动开启 `enable-old-value` 选项的问题 #3676
    * 修复 Avro sink 模块不支持解析 JSON 类型列的问题 #3624
    * 修复监控 checkpoint lag 出现负值的问题 #3010

- TiDB Data Migration (DM)

    * 修复了 DM 的 master/worker 线程以特定顺序重启后中继状态错误的问题 #3478
    * 修复了 DM worker 在重启后无法完成初始化的问题 #3344
    * 修复了 DM 任务在分区表相关 DDL 执行时间过长时失败的问题 #3854
    * 修复了 DM 在上游是 MySQL 8.0 时报错 “invalid sequence” 的问题 #3847
    * 修复了 DM 采用细粒度失败重试策略导致数据丢失问题 #3487
    * 修复 `CREATE VIEW` 语句中断复制任务的问题 #4173
    * 修复 skip DDL 后需要重置 Schema 的问题 #4177

- TiDB Lightning

    * 修复在某些导入操作没有包含源文件时，TiDB Lightning 不会删除 metadata schema 的问题 #28144
    * 修复存储 URL 前缀为 “gs://xxx” 而不是 “gcs://xxx” 时，TiDB Lightning 报错的问题 #32742
    * 修复设置 –log-file= “-” 时，没有 log 输出到 stdout 的问题 #29876
    * 修复 S3 存储路径不存在时 TiDB Lightning 不报错的问题 #30709

16.14.5 TiDB 5.3 Release Notes

发版日期：2021 年 11 月 30 日

TiDB 版本：5.3.0

在 v5.3.0 版本中，你可以获得以下关键特性：

- 引入临时表，简化业务逻辑并提升性能
- 支持设置表和分区的表属性
- 支持为 TiDB Dashboard 创建最小权限用户，提高系统安全性
- 优化 TiDB 时间戳处理流程，提升系统的整体性能
- 提高 DM 同步性能，实现以更低的延迟将数据从 MySQL 同步数据到 TiDB
- 支持 TiDB Lightning 分布式并行导入，提升全量数据迁移效率
- 支持“一键”保存和恢复现场问题的相关信息，提升查询计划问题诊断的效率
- 支持持续性能分析 (Continuous Profiling) 实验特性，提高数据库性能的可观测性
- 持续优化存储和计算引擎，提升系统性能和稳定性
- 降低 TiKV 写入延迟，从 Raftstore 线程池中分离出 IO 线程池（默认不开启）

16.14.5.1 兼容性变化

> **注意：**
>
> 当从一个早期的 TiDB 版本升级到 TiDB v5.3.0 时，如需了解所有中间版本对应的兼容性更改说明，请查看对应版本的Release Notes。

16.14.5.1.1 系统变量

变量名	修改类型	描述
tidb_enable_noop_functions	修改	由于 TiDB v5.3.0 支持临时表，此变量的控制范围不再包括 `CREATE TEMPORARY TABLE` 和 `DROP TEMPORARY TABLE` 行为。

变量名	修改类型	描述
tidb_enable_pseudo_for_outdated_stats ↪	新增	此变量用于控制优化器在一张表上的统计信息过期时的行为。默认值为 ON，当表数据被修改的行数大于该表总行数的 80%（该比例可通过pseudo ↪ -estimate- ↪ ratio 配置项调整）时，优化器认为该表上除总行数以外的统计信息不再可靠，转而使用 pseudo 统计信息。将该变量值设为 OFF 后，即使统计信息过期，优化器也仍会使用该表上的统计信息。
tidb_enable_tso_follower_proxy ↪	新增	此变量用于开启或关闭 TSO Follower Proxy 特性。默认值为 OFF，代表关闭 TSO Follower Proxy 特性。此时，TiDB 仅会从 PD leader 获取 TSO。当开启该特性之后，TiDB 在获取 TSO 时会将请求均匀地发送到所有 PD 节点上，通过 PD follower 转发 TSO 请求，从而降低 PD leader 的 CPU 压力。

变量名	修改类型	描述
tidb_tso_client_batch_max_wait_time ↪	新增	此变量用于设置 TiDB 向 PD 请求 TSO 时进行一次攒批操作的最大等待时长。默认值为 0，即不进行额外的等待。
tidb_tmp_table_max_size	新增	此变量用于限制单个临时表的最大大小，临时表超出该大小后报错。

16.14.5.1.2 配置文件参数

配置文件	配置项	修改类型	描述
TiDB	prepared-plan-cache. ↪ capacity	修改	此配置项用于控制缓存语句的数量。默认值从 100 修改为 1000。
TiKV	storage. ↪ reserve-space	修改	此配置项用于控制 TiKV 启动时用于保护磁盘的预留空间。从 v5.3.0 起，预留空间的 80% 用作磁盘空间不足时运维操作所需要的额外磁盘空间，剩余的 20% 为磁盘临时文件。

配置文件	配置项	修改类型	描述
TiKV	memory-usage-limit	修改	以前的版本没有 memory-usage-limit 参数，升级后该参数值根据 storage.block-cache.capacity 来计算。
TiKV	raftstore.store-io-pool-size	新增	表示处理 Raft I/O 任务的线程池中线程的数量，即 StoreWriter 线程池的大小。
TiKV	raftstore.raft-write-size-limit	新增	触发 Raft 数据写入的阈值。当数据大小超过该配置项值，数据会被写入磁盘。当 raftstore.store-io-pool-size 的值为 0 时，该配置项不生效。
TiKV	raftstore.raft-msg-flush-interval	新增	Raft 消息攒批发出的间隔时间。每隔该配置项指定的间隔，Raft 消息会攒批发出。当 raftstore.store-io-pool-size 的值为 0 时，该配置项不生效。

配置文件	配置项	修改类型	描述
TiKV	raftstore.raft-reject-transfer-leader-duration	删除	控制迁移 leader 到新加节点的最小时间。
PD	log.file.max-days	修改	此配置项用于控制日志保留的最长天数。默认值从 1 修改为 0。
PD	log.file.max-backups	修改	此配置项用于控制日志文件保留的最大个数。默认值从 7 修改为 0。
PD	patrol-region-interval	修改	此配置项用于控制 replicaChecker 检查 Region 健康状态的运行频率，越短则运行越快，通常状况不需要调整。默认值从 100ms 修改为 10ms。
PD	max-snapshot-count	修改	此配置项用于控制单个 store 最多同时接收或发送的 snapshot 数量，调度受制于这个配置来防止抢占正常业务的资源。默认值从 3 修改为 64。

配置文件	配置项	修改类型	描述
PD	max-pending-peer-count	修改	此配置项用于控制单个 store 的 pending peer 上限，调度受制于这个配置来防止在部分节点产生大量日志落后的 Region。默认值从 16 修改为 64。
TiD Lightning	meta-schema-name	新增	在目标集群保存各个 TiDB Lightning 实例元信息的 schema 名字，默认值为 “lightning_metadata”。

16.14.5.1.3 其他

- 临时表：
  - 对于本地临时表，如果在 v5.3.0 升级前创建了本地临时表，这些临时表实际为普通表，在升级至 v5.3.0 或更高版本后，也会被 TiDB 当成普通表处理。对于全局临时表，如果在 v5.3.0 上创建了全局临时表，当 TiDB 降级至 v5.3.0 以前版本后，这些临时表会被当作普通表处理，导致数据错误。
  - TiCDC 和 BR 从 v5.3.0 开始支持全局临时表。如果使用 v5.3.0 以下版本同步全局临时表到下游，会导致表定义错误。
  - 通过 TiDB 数据迁移工具导入的集群、恢复后的集群、同步的下游集群必须是 TiDB v5.3.0 及以上版本，否则创建全局临时表时报错。
  - 关于临时表的更多兼容性信息，请参考与 MySQL 临时表的兼容性和与其他 TiDB 功能的兼容性限制。
- 对于 v5.3.0 之前的版本，当系统变量设置为非法值时，TiDB 会报错。从 v5.3.0 起，当系统变量设置为非法值时，TiDB 会返回成功，并报类似 `|Warning | 1292 | Truncated incorrect xxx: 'xx'` 的警告。
- 修复 `SHOW CREATE VIEW` 不需要 `SHOW VIEW` 权限的问题，现在用户必须具有 `SHOW VIEW` 权限才允许执行 `SHOW CREATE VIEW` 语句。
- 系统变量 `sql_auto_is_null` 被加入 Noop Function 中，当 `tidb_enable_noop_functions = 0/OFF` 时，修改该变量会报错。
- 不再允许执行 `GRANT ALL ON performance_schema.*` 语法，在 TiDB 上执行该语句会报错。

- 修复 v5.3.0 之前的版本中新增索引会导致在规定时间外触发 auto-analyze 的问题。在 v5.3.0 中，用户通过 `tidb_auto_analyze_start_time` 和 `tidb_auto_analyze_end_time` 设定时间段后，只会在该时间段内触发 auto-analyze。
- plugin 默认存放目录从 `""` 改为 `/data/deploy/plugin`。
- DM 代码迁移至 TiCDC 代码仓库的 dm 文件夹。从 DM v5.3.0 起，DM 采用与 TiDB 相同的版本号。DM v2.0 的下一个主版本为 DM v5.3。DM v2.0 到 v5.3 无兼容性变更，升级过程与正常升级无差异。
- 默认部署 Prometheus v2.27.1 ，它是 2021 年 5 月发布的版本，提供了更多的功能并解决了安全风险。相对于 5.3.0 之前版本 TiDB 默认使用的 Prometheus v2.8.1，v2.27.1 存在 Alert 时间格式变化，详情见 Prometheus commit。

16.14.5.2 新功能

16.14.5.2.1 SQL

- 使用 SQL 接口设置数据放置规则（实验特性）

  新增对 `[CREATE | ALTER] PLACEMENT POLICY` 语句的支持，提供 SQL 接口设置数据放置规则。通过该功能，用户可以指定任意连续数据按照不同地域、机房、机柜、主机、硬件、副本数规则进行部署，满足低成本、高可用、灵活多变的业务诉求。该功能可以实现以下业务场景：

  - 合并多个不同业务的数据库，大幅减少数据库常规运维管理的成本，并通过规则配置实现业务资源隔离。
  - 增加重要数据的副本数，提高业务可用性和数据可靠性。
  - 将最新数据存入 SSD，历史数据存入 HDD，降低归档数据存储成本。
  - 把热点数据的 leader 放到高性能的 TiKV 实例上。
  - 将冷数据分离到不同的存储中以提高可用性。

  用户文档，#18030

- 临时表

  新增对 `CREATE [GLOBAL] TEMPORARY TABLE` 语句的支持。支持创建临时表，方便管理业务中间计算的临时数据。临时表中的数据均保存在内存中，用户可通过 `tidb_tmp_table_max_size` 变量限制临时表的内存大小。TiDB 支持以下两种临时表：

  - Global 临时表
    * 对集群内所有 session 可见，表结构持久化。
    * 提供事务级别的数据隔离，数据只在事务内有效，事务结束后自动删除数据。
  - Local 临时表
    * 只对当前 session 可见，表结构不持久化。
    * 支持重名，用户无需为业务设计复杂的表命名规则。
    * 提供会话级别的数据隔离，降低业务设计复杂度，会话结束后删除临时表。

    用户文档，#24169

- 支持 FOR UPDATE OF TABLES 语法

  对于存在多表 join 的语句，支持只对 OF TABLES 中包含的表关联的行进行悲观锁加锁操作。

  用户文档，#28689

- 设置表属性

  增加 ALTER TABLE [PARTITION] ATTRIBUTES 语句支持，允许用户为表和分区设置属性。目前 TiDB 仅支持设置 merge_option 属性。通过为表或分区添加 merge_option 属性，用户可以显式控制 Region 是否合并。

  应用场景：当用户 SPLIT TABLE 之后，如果超过一定时间后（由 PD 参数split-merge-interval 控制）没有插入数据，空 Region 默认会被自动合并。此时，可以通过该功能设置表属性为 merge_option=deny，避免 Region 的自动合并。

  用户文档，#3839

16.14.5.2.2 安全

- 支持为 TiDB Dashboard 创建最小权限用户

  TiDB Dashboard 的账号体系与 TiDB SQL 用户一致，并基于 TiDB SQL 用户的权限进行 TiDB Dashboard 授权验证。TiDB Dashboard 所需的权限较少，甚至可以只有只读权限。可以基于最小权限原则配置合适的用户访问 TiDB Dashboard，减少高权限用户的使用场景。

  建议为访问 TiDB Dashboard 创建一个最小权限的 SQL 用户，并用该用户登录 TiDB Dashboard，避免使用高权限用户，提升安全性。

  用户文档

16.14.5.2.3 性能

- 优化 PD 时间戳处理流程

  优化 TiDB 时间戳处理流程，支持通过开启 PD Follower Proxy 和调整 PD client 批量请求 TSO 时所需的 batch 等待时间的方式来降低 PD leader 时间戳处理负载，提升系统整体可扩展性。

  - 支持通过系统变量tidb_enable_tso_follower_proxy 设置 PD Follower Proxy 功能开关。在 PD 时间戳请求负载过高的情况下，通过开启 PD Follower Proxy，可以将 follower 上请求周期内收集到的 TSO request 批量转发到 leader 节点，从而有效减少 client 与 leader 的直接交互次数，降低 leader 的负载，提升 TiDB 整体性能。

    > 注意：
    > 在 client 数较少、PD leader 负载不高的情况下，不建议开启 PD Follower Proxy 功能。

  - 支持通过tidb_tso_client_batch_max_wait_time 系统变量设置 PD client 批量请求 TSO 时所需的最大 batch 等待时间，单位为毫秒。在 PD TSO 请求负载过高的情况下，可以通过调大等待时间获得更大的 batch size，从而降低 PD 负载，提升吞吐。

    > 注意：
    > 在 TSO 请求负载不高的情况下，不建议调整该参数。

  用户文档，#3149

16.14.5.2.4 稳定性

- 支持多节点永久损坏后的在线有损恢复（实验特性）

  新增对 `pd-ctl unsafe remove-failed-stores` 命令的支持，实现在线数据有损恢复。当多数副本发生了永久性损坏（如磁盘损坏）等问题，导致无法在业务层读写一段数据时，PD 可以执行在线数据恢复，使该数据恢复至可读写状态。

  进行该功能相关操作时，建议在 TiDB 团队支持下完成。

  用户文档，#10483

16.14.5.2.5 数据迁移

- 提高 DM 同步性能

  支持以下功能，实现以更低的延迟将数据从 MySQL 同步数据到 TiDB。

  - 合并单行数据的多次变更 (Compact multiple updates on a single row into one statement)
  - 点查更新合并为批量操作 (Merge batch updates of multiple rows into one statement)

- 增加 DM 的 OpenAPI 以更方便地管理集群（实验特性）

  DM 提供 OpenAPI 功能，用户可通过 OpenAPI 对 DM 集群进行查询和运维操作。OpenAPI 的总体功能和dmctl工具类似。

  当前 OpenAPI 功能为实验特性，默认关闭，不建议在生产环境中使用。

  用户文档

- TiDB Lightning 并行导入

  TiDB Lightning 支持用户同时部署多个 Lightning，并行地将单表或者多表数据迁移 TiDB。该功能无需特别的配置，在不改变用户使用习惯的同时，极大提高了用户的数据迁移能力，助力大数据量业务架构升级，在生产环境使用 TiDB。

  经测试，使用 10 台 TiDB Lightning，20 TB 规模的 MySQL 数据可以在 8 小时内导入到 TiDB，单台 TiDB Lightning 可以支持 250 GiB/h 的导入速度，整体效率提升了 8 倍。

  用户文档

- TiDB Lightning 执行任务前的检查项

  TiDB Lightning 增加了执行前检查配置的功能。默认开启。该功能会自动进行一些磁盘空间和执行配置的常规检查，主要目的是确保后续的整个导入过程顺利。

  用户文档

- TiDB Lightning 支持导入 GBK 编码的文件

  通过指定源数据文件的字符集，TiDB Lightning 会在导入过程中将源文件从指定的字符集转换为 UTF-8 编码。

  用户文档

- Sync-diff-inspector 优化

  - 大幅提升了对比速度，由原来的 375 MB/s 提升至 700 MB/s。

- 对比过程中对 TiDB 节点的内存消耗降低近一半。
- 优化了用户交互界面，在对比过程中可以显示进度。

用户文档

16.14.5.2.6 问题诊断效率

- 保存和恢复集群现场信息

  在定位排查 TiDB 集群问题时，用户经常需要提供系统和查询计划相关的信息。为了帮助用户更方便地获取相关信息，更高效地排查集群问题，TiDB 在 v5.3.0 中引入了 PLAN REPLAYER 命令，用于“一键”保存和恢复现场问题的相关信息，提升查询计划问题诊断的效率，同时方便将问题归档管理。

  PLAN REPLAYER 主要功能如下：

  - 导出排查现场 TiDB 集群的相关信息，导出为 ZIP 格式的文件用于保存。
  - 在任意 TiDB 集群上导入另一 TiDB 集群现场信息的 ZIP 文件。

  用户文档，#26325

16.14.5.2.7 TiDB 数据共享订阅

- TiCDC 支持灾备场景下的最终一致性复制

  在主从灾备架构下，当生产集群（即 TiCDC 同步的上游集群）发生灾难、且短时间内无法恢复对外提供服务时，TiCDC 具备保证从集群数据一致性的能力，并允许业务快速的将流量切换至从集群，避免数据库长时间不可用而对业务造成影响。

  该功能支持 TiCDC 将 TiDB 集群的增量数据复制到备用关系型数据库 TiDB/Aurora/MySQL/MariaDB，在 TiCDC 正常同步没有延迟的情况下，上游发生灾难后，可以在 5 分钟内将下游集群恢复到上游的某个 snapshot 状态，并且允许丢失的数据小于 30 分钟。即 RTO <= 5min，RPO <= 30min。

  用户文档

- TiCDC 支持 HTTP 协议 OpenAPI 对 TiCDC 任务进行管理

  从 TiDB v5.3.0 起，TiCDC 提供的 OpenAPI 功能成为正式特性，用户可通过 OpenAPI 对 TiCDC 集群进行查询和运维操作。

  用户文档

16.14.5.2.8 部署及运维

- 持续性能分析（实验特性）

  TiDB Dashboard 引入持续性能分析功能，提供在集群运行状态时自动保存实例性能分析结果的能力，通过火焰图的形式提高了 TiDB 集群性能的可观测性，有助于缩短故障诊断时间。

  该功能默认关闭，需进入 TiDB Dashboard 持续性能分析页面开启。

  持续性能分析功能必须使用 TiUP 1.7.0 及以上版本升级或安装的集群才可使用。

  用户文档

16.14.5.3 遥测

TiDB 在遥测中新增收集 TEMPORARY TABLE 功能的开启情况。收集的数据中不包含任何实际业务的表名或表数据。

关于所收集的信息详情及如何禁用该行为，参见遥测文档。

16.14.5.4 移除功能

从 TiCDC v5.3.0 版本开始，TiDB 集群之间的环形同步功能（v5.0 实验特性）被移除。如果在升级 TiCDC 前已经使用过环形同步功能复制数据，升级后此部分数据的环形同步不受影响。

16.14.5.5 提升改进

- TiDB
  - 当 coprocessor 遇到锁时，在调试日志中显示受影响的 SQL 语句帮助诊断问题 #27718
  - 在 SQL 逻辑层备份和恢复数据时，支持显示备份和恢复数据的大小 #27247
  - 改进 `tidb_analyze_version=2` 时 ANALYZE 默认的收集逻辑，提高收集速度并且降低资源开销
  - 引入语法 `ANALYZE TABLE table_name COLUMNS col_1, col_2, ..., col_n`，为宽表提供只收集一部分列统计信息的方案，提高宽表收集统计信息的速度
- TiKV
  - 增强磁盘空间防护能力，提升存储稳定性。

    针对 TiKV 遇到磁盘写满错误时可能 Panic 的问题，为磁盘剩余空间引入两级阈值防御机制，避免超额流量耗尽磁盘空间。同时，提供阈值触发时的空间回收能力。当剩余空间触发阈值时，部分写操作会失败，并返回 disk full 错误和盘满节点列表。此时，可以通过 `Drop/Truncate Table` 或者扩容等方式来回收空间，恢复服务。
  - 简化 L0 层流控算法 #10879
  - 优化 raft client 错误日志的收集 #10944
  - 优化日志线程以避免其成为性能瓶颈 #10841
  - 添加更多的写入查询统计类型 #10507
  - 降低写入延迟，从 Raftstore 线程池中分离出 IO 线程池（默认不开启）。具体调优操作，请参考TiKV 线程池性能调优 #10540
- PD
  - 热点调度器的 QPS 维度支持更多的写请求类型 #3869
  - 通过动态调整 Balance Region 调度器的重试上限，优化该调度器的性能 #3744
  - 将 TiDB Dashboard 升级至 v2021.10.08.1 #4070
  - 允许 Evict Leader 调度器调度拥有不健康副本的 Region #4093
  - 优化调度器退出的速度 #4146
- TiFlash
  - 显著优化了 TableScan 算子的执行效率

- 优化了 Exchange 算子的执行效率
- 减少了存储引擎的 GC 过程中的写放大和内存使用（实验功能）
- 改进了 TiFlash 重启时的稳定性和可用性，减少了重启结束后短时间内查询可能失败的情况
- 增加支持下推多个新的字符串，时间等函数到 MPP 引擎
  * 字符串函数：LIKE pattern，FORMAT()，LOWER()，LTRIM()，RTRIM()，SUBSTRING_INDEX()，TRIM()，UCASE()，UPPER()
  * 数学函数：ROUND(decimal, int)
  * 日期时间函数：HOUR()，MICROSECOND()，MINUTE()，SECOND()，SYSDATE()
  * 类型转换函数：CAST(time, real)
  * 聚合函数：GROUP_CONCAT()，SUM(enum)
- 提供了 512 位 SIMD 支持
- 增强了对过期的数据版本的清理算法，减少磁盘使用量及提高读文件性能
- 解决了用户在某些非 Linux 平台系统上查看 dashboard 时，无法获取内存或 CPU 等相关信息的问题
- 统一 TiFlash 日志文件的命名风格（与 TiKV 保持一致），并支持动态修改 logger.count、 logger.size
- 完善了列存文件的数据校验能力（checksums，实验功能）

- Tools
  - TiCDC
    * 通过修改 Kafka sink 配置项 MaxMessageBytes 的默认值，由 64 MB 减小为 1 MB，以修复消息过大会被 Kafka Broker 拒收的问题 #3104
    * 减少同步链路中的内存占用 #2553 #3037 #2726
    * 优化监控项和告警规则，提升了同步链路、内存 GC、存量数据扫描过程的可观测性 #2735 #1606 #3000 #2985 #2156
    * 当同步任务状态正常时，不再显示历史错误信息，避免误导用户 #2242

16.14.5.6 Bug 修复

- TiDB
  - 修复在分区中下推聚合算子时，因浅拷贝 schema 列导致执行计划出错，进而导致执行时报错的问题 #27797 #26554
  - 修复 `plan cache` 无法感知 unsigned 标志变化的问题 #28254
  - 修复当分区功能出现 `out of range` 时 `partition pruning` 出错的问题 #28233
  - 修复在某些情况下 Planner 可能缓存无效 `join` 计划的问题 #28087
  - 修复 hash 列为 enum 时构建错误 `IndexLookUpJoin` 的问题 #27893
  - 修复批处理客户端在某些罕见情况下回收空闲连接可能会阻塞发送请求的问题 #27688
  - 修复当 TiDB Lightning 在目标集群上执行校验失败时 panic 的问题 #27686
  - 修复某些情况下 `date_add` 和 `date_sub` 函数执行结果错误的问题 #27232
  - 修复 hour 函数在向量化表达式中执行结果错误的问题 #28643
  - 修复连接到 MySQL 5.1 或更早的客户端时存在的认证问题 #27855
  - 修复当新增索引时自动分析可能会在指定时间之外触发的问题 #28698
  - 修复设置任何会话变量都会使 `tidb_snapshot` 失效的问题 #28683

- 修复在有大量 `miss-peer region` 的集群中 BR 不可用的问题 #27534
- 修复当不支持的 `cast` 被下推到 TiFlash 时出现的非预期错误，例如 `tidb_cast to Int32 is not supported` #23907
- 修复 `%s value is out of range in '%s'` 报错中缺失 `DECIMAL overflow` 信息的问题 #27964
- 修复 MPP 节点的可用性检测在某些边界场景中无法工作的问题 #3118
- 修复分配 `MPP task ID` 时出现 `DATA RACE` 的问题 #27952
- 修复删除空的 `dual table` 后 MPP 查询出现 `index out of range` 报错的问题 #28250
- 修复运行 MPP 查询时出现 `invalid cop task execution summaries length` 相关日志的问题 #1791
- 修复运行 MPP 查询时出现 `cannot found column in Schema column` 报错的问题 #28149
- 修复 TiDB 在 TiFlash 关闭时可能出现 panic 的问题 #28096
- 移除对基于 3DES (三重数据加密算法) 不安全的 TLS 加密套件的支持 #27859
- 修复因 Lightning 前置检查会连接已下线的 TiKV 节点而导致导入失败的问题 #27826
- 修复在导入太多文件到表时前置检查花费太多时间的问题 #27605
- 修复表达式重写时 `between` 推断出错误排序规则的问题 #27146
- 修复 `group_concat` 函数没有考虑排序规则的问题 #27429
- 修复 `extract` 函数处理负值时的问题 #27236
- 修复当设置 `NO_UNSIGNED_SUBTRACTION` 时创建分区失败的问题 #26765
- 避免在列修剪和聚合下推中使用有副作用的表达式 #27106
- 删除无用的 gRPC 日志 #24190
- 限制有效的小数点长度以修复精度相关的问题 #3091
- 修复 `plus` 表达式中检查溢出方法出错的问题 #26977
- 修复当导出带有 `new collation` 数据的表的统计信息时报 `data too long` 错误的问题 #27024
- 修复 `TIDB_TRX` 中不包含重试事务的问题 #28670
- 修复配置项 `plugin_dir` 的默认值错误问题 #28084
- 修复 `CONVERT_TZ` 函数在指定时区和 UTC 偏移量时返回 `NULL` 的问题 #8311
- 修复如果 `character_set_server` 和 `collation_server` 指定的字符集未在 `CREATE SCHEMA` 语句中指定时，那么创建的新表结构不使用 `character_set_server` 和 `collation_server` 指定的字符集的问题 #27214

- TiKV

  - 修复 Region 迁移时 Raftstore 模块出现死锁导致 TiKV 不可用的问题。用户可通过关闭调度并重启出问题的 TiKV 来临时应对。 #10909
  - 修复因 Congest 错误而导致的 CDC 频繁增加 scan 重试的问题 #11082
  - 修复因 channel 打满而导致的 Raft 断连情况 #11047
  - 修复 Raft client 中 batch 消息过大的问题 #9714
  - 修复 `resolved_ts` 中协程泄漏的问题 #10965
  - 修复当 response 大小超过 4 GiB 时 Coprocessor panic 的问题 #9012
  - 修复当一个 snapshot 文件无法被垃圾清理 (GC) 时 snapshot GC 会缺失 GC snapshot 文件的问题 #10813
  - 修复当处理 Coprocessor 请求时因超时而导致 panic 的问题 #10852
  - 修复因统计线程监控数据导致的内存泄漏 #11195
  - 修复在某些平台获取 cgroup 信息导致 panic 的问题 #10980
  - 修复 Compaction Filter GC 无法清除 MVCC Deletion 版本导致 scan 性能下降的问题 #11248

- PD

  - 修复因超过副本配置数量而导致错误删除带有数据且处于 pending 状态的副本的问题 #4045

  - 修复 PD 未能及时修复 Down Peer 副本的问题 #4077
  - 修复 Scatter Range 调度器无法对空 Region 进行调度的问题 #4118
  - 修复 key manager 占用过多 CPU 的问题 #4071
  - 修复热点调度器变更配置的过程中可能会存在的数据竞争问题 #4159
  - 修复因 Region syncer 卡住而导致 leader 选举慢的问题 #3936

- TiFlash

  - 修复 TiFlash Store Size 统计结果不准确的问题
  - 修复 TiFlash 在部分平台上由于缺失 `nsl` 库而无法启动的问题
  - 阻止 wait index 无限等待，防止写入压力较重时 TiFlash 长时间等待数据同步而无法提供服务的问题（新增默认超时为 5 分钟）
  - 解决了当日志体量很大时，用户搜索日志很慢或搜索不出的问题
  - 解决了搜索比较久远的历史日志时，只能搜索出最近的一部分日志的问题
  - 修复在打开 new collation 的情况下可能出现的结果错误
  - 修复 SQL 语句中含有极长嵌套表达式时可能出现的解析错误
  - 修复 Exchange 算子中可能出现的 `Block schema mismatch` 错误
  - 修复 Decimal 类型比较时可能出现的 `Can't compare` 错误
  - 修复 `left/substring` 函数中的 `3rd arguments of function substringUTF8 must be constants` 错误

- Tools

  - TiCDC

    * 修复当上游 TiDB 实例意外退出时，TiCDC 同步任务推进可能停滞的问题 #3061
    * 修复当 TiKV 向同一 Region 发送重复请求时，TiCDC 进程 panic 的问题 #2386
    * 修复在验证下游 TiDB/MySQL 可用性时产生的不必要的 CPU 消耗 #3073
    * 修复 TiCDC 产生的 Kafka 消息体积不受 `max-message-size` 约束的问题 #2962
    * 修复当写入 Kafka 消息发生错误时，TiCDC 同步任务推进可能停滞的问题 #2978
    * 修复当开启 `force-replicate` 时，可能某些没有有效索引的分区表被忽略的问题 #2834
    * 修复当扫描存量数据耗时过长时，可能由于 TiKV 进行 GC 而导致存量数据扫描失败的问题 #2470
    * 修复在将某些类型的列编码为 Open Protocol 格式时，TiCDC 进程可能 panic 的问题 #2758
    * 修复在将某些类型的列编码为 Avro 格式时，TiCDC 进程可能 panic 的问题 #2648

  - TiDB Binlog

    * 修复当大部分表被过滤掉时，在某些特殊的负载下，checkpoint 不更新的问题 #1075

16.15 v5.2

16.15.1 TiDB 5.2.4 Release Notes

发版日期：2022 年 4 月 26 日

TiDB 版本：5.2.4

16.15.1.1 兼容性更改

- TiDB
  - 将系统变量`tidb_analyze_version`的默认值从 2 修改为 1 #31748
- TiKV
  - 新增 `raft-log-compact-sync-interval` 配置项，用于压缩非必要的 Raft 日志的时间间隔，默认值为 `"2s"` #11404
  - 将`raft-log-gc-tick-interval`的默认值从 `"10s"` 修改为 `"3s"` #11404
  - 当`storage.flow-control.enable`的值为 `true` 时，`storage.flow-control.hard-pending-compaction-bytes-limit` 的配置会覆盖`rocksdb.(defaultcf|writecf|lockcf).hard-pending-compaction-bytes-limit`的配置 #11424
- Tools
  - TiDB Lightning
    * 将 `regionMaxKeyCount` 的默认值从 1_440_000 调整为 1_280_000，以避免数据导入后出现过多空 Region #30018

16.15.1.2 提升改进

- TiKV
  - 通过将 leader 转让给 CDC observer 减少延迟抖动 #12111
  - 通过减少需要进行清理锁 (Resolve Locks) 步骤的 Region 数量来减少 TiCDC 恢复时间 #11993
  - 将 proc filesystem (procfs) 升级至 0.12.0 版本 #11702
  - 通过增加对 Raft log 进行垃圾回收 (GC) 时的 write batch 大小来加快 GC 速度 #11404
  - 将插入 SST 文件时的校验操作从 Apply 线程池移动到 Import 线程池，从而提高 SST 文件的插入速度 #11239
- Tools
  - TiCDC
    * 将 Kafka Sink `partition-num` 的默认值改为 3，使 TiCDC 更加平均地分发消息到各个 Kafka partition #3337
    * 减少 TiKV 节点宕机后 KV client 恢复的时间 #3191
    * 在 Grafana 中添加 `Lag analyze` 监控面板 #4891
    * 暴露 Kafka producer 配置参数，使之在 TiCDC 中可配置 #4385
    * 为 changefeed 重启操作添加指数退避机制 #3329
    * 减少 “EventFeed retry rate limited” 日志的数量 #4006
    * 将 `max-message-bytes` 默认值设置为 10M #4041
    * 增加更多 Prometheus 和 Grafana 监控告警参数，包括 `no owner alert`、`mounter row`、`table sink total row` 和 `buffer sink total row` #4054 #1606
    * 在 Grafana 监控面板中支持多个 Kubernetes 集群 #4665
    * 在 `changefeed checkpoint` 监控项中添加监控指标预计追加时间 (catch-up ETA) #5232

16.15.1.3 Bug 修复

- TiDB
  - 修复 Nulleq 函数作用在 Enum 类型上可能出现结果错误的问题 #32428
  - 修复 INDEX HASH JOIN 报 `send on closed channel` 的问题 #31129
  - 修复并发的列类型变更导致 schema 与数据不一致的问题 #31048
  - 修复乐观事务下数据索引可能不一致的问题 #30410
  - 修复当 SQL 语句中存在 JSON 类型列与 CHAR 类型列连接时，SQL 出错的问题 #29401
  - 修复窗口函数在使用事务时，计算结果与不使用事务时的计算结果不一致的问题 #29947
  - 修复 SQL 语句中带有 NATURAL JOIN 时，意外报错 `Column 'col_name' in field list is ambiguous` 的问题 #25041
  - 修复将 `Decimal` 转为 `String` 时长度信息错误的问题 #29417
  - 修复由于 `tidb_enable_vectorized_expression` 设置的值不同（on 或 `off`）导致 GREATEST 函数返回结果不一致的问题 #29434
  - 修复使用 left join 同时删除多张表数据时可能出现错误结果的问题 #31321
  - 修复 TiDB 可能向 TiFlash 发送重复任务的问题 #32814
  - 修复执行查询时报 MPP task list 为空错误的问题 #31636
  - 修复 innerWorker panic 导致的 index join 结果错误的问题 #31494
  - 修复 `INSERT ... SELECT ... ON DUPLICATE KEY UPDATE` 语句 panic 的问题 #28078
  - 修复针对 `Order By` 的优化导致查询结果有误的问题 #30271
  - 修复使用 ENUM 类型的列进行 Join 时结果可能不正确的问题 #27831
  - 修复当 `CASE WHEN` 函数和 ENUM 类型一起使用时的崩溃问题 #29357
  - 修复 `microsecond` 函数的向量化表达式版本结果不正确的问题 #29244
  - 修复窗口函数执行时本应报错但是让 TiDB 崩溃的问题 #30326
  - 修复特定情况下 Merge Join 执行结果错误的问题 #33042
  - 修复关联子查询返回结果中有常量时导致执行结果出错的问题 #32089
  - 修复 ENUM 或 SET 类型的列因为编码错误导致写入数据错误的问题 #32302
  - 修复开启 New Collation 时，作用在 ENUM 或SET 列上的 MAX 或 MIN 函数结果出错的问题 #31638
  - 修复某些情况下 IndexHashJoin 算子没有正常退出的问题 #31062
  - 修复有虚拟列时可能导致 TiDB 读到错误数据的问题 #30965
  - 修复日志级别的设置没有对慢查询日志生效的问题 #30309
  - 修复某些情况下分区表无法充分利用索引来扫描数据的问题 #33966
  - 修复 TiDB 的后台 HTTP 服务可能没有正确关闭导致集群状态异常的问题 #30571
  - 修复 TiDB 会非预期地打印很多鉴权失败相关日志的问题 #29709
  - 修复系统变量 `max_allowed_packet` 不生效的问题 #31422
  - 修复当 auto ID 超出范围时，REPLACE 语句错误地修改了其它行的问题 #29483
  - 修复慢查询日志无法正常输出而且可能消耗大量内存的问题 #32656
  - 修复 NATRUAL JOIN 可能输出多余列的问题 #24981
  - 修复使用前缀列索引时，ORDER + LIMIT 语句可能导致结果出错的问题 #29711
  - 修复乐观事务重试时，DOUBLE 类型的自增列可能在重试时值发生改变的问题 #29892
  - 修复 STR_TO_DATE 函数无法正确处理微秒部分的前导 0 的问题 #30078
  - 修复在 TiFlash 不支持使用空范围读表的情况，依然选择 TiFlash 导致查询结果错误的问题 #33083
- TiKV
  - 修复旧信息造成 TiKV panic 的问题 #12023

  - 修复因内存统计指标溢出而造成的间歇性丢包和内存不足 (OOM) 的问题 #12160
  - 修复在 Ubuntu 18.04 下进行性能分析会造成 TiKV panic 的问题 #9765
  - 修复 tikv-ctl 对 `bad-ssts` 结果字符串进行错误匹配的问题 #12329
  - 修复 Replica Read 可能违反线性一致性的问题 #12109
  - 修复 TiKV 运行 2 年以上可能 panic 的问题 #11940
  - 修复开启流量控制且显式设置 `level0_slowdown_trigger` 时出现 QPS 下降的问题 #11424
  - 修复 cgroup controller 没有被挂载会造成 panic 的问题 #11569
  - 修复在滞后的 Region peer 上执行 Region Merge 导致的元数据损坏问题 #11526
  - 修复 TiKV 停止后 Resolved TS 延迟会增加的问题 #11351
  - 修复在极端情况下同时进行 Region Merge、ConfChange 和 snapshot 时，TiKV 会出现 panic 的问题 #11475
  - 修复 tikv-ctl 无法正确输出 Region 相关信息的问题 #11393
  - 修复 Decimal 除法计算的结果为 0 时符号为负的问题 #29586
  - 修复悲观事务中 prewrite 请求重试在极少数情况下影响数据一致性的风险 #11187
  - 修复因统计线程监控数据导致的内存泄漏 #11195
  - 修复 TiKV 监控项中实例级别 gRPC 的平均延迟时间不准确的问题 #11299
  - 修复 Peer 状态为 Applying 时快照文件被删除会造成 panic 的问题 #11746
  - 修复 GC worker 繁忙后无法执行范围删除（即执行内部命令 `unsafe_destroy_range`）的问题 #11903
  - 修复删除未初始化的副本可能会造成旧副本被重新创建的问题 #10533
  - 修复逆序扫表时 TiKV 无法正确读到内存锁的问题 #11440
  - 修复协程的执行速度太快时偶尔出现的死锁问题 #11549
  - 修复删除 Peer 可能造成高延迟的问题 #10210
  - 修复待 merge 的 Region 无效会导致 TiKV panic 且非预期地销毁 peer 的问题 #12232
  - 修复合并 Region 时因 target peer 被一个未进行初始化就被销毁的 peer 所替换，从而引起 TiKV panic 的问题 #12048
  - 修复 apply snapshot 被暂停时会引起 TiKV panic 的问题 #11618
  - 修复了在 operator 执行失败时，TiKV 无法正确地计算正在发送的 snapshot 数量的问题 #11341

- PD

  - 修复 Region Scatterer 生成的调度缺失部分 Peer 的问题 #4565
  - 修复热点统计中无法剔除冷热点数据的问题 #4390

- TiFlash

  - 修复 MPP 任务可能永远泄漏线程的问题 #4238
  - 修复 `IN` 函数的结果在多值表达式中不正确的问题 #4016
  - 修复日期格式将 `'\n'` 处理为非法分隔符的问题 #4036
  - 修复在读取工作量大时添加列后可能出现的查询错误 #3967
  - 修复错误地配置存储目录会导致非预期行为的问题 #4093
  - 修复一些异常没有被正确地处理的问题 #4101
  - 修复 `STR_TO_DATE()` 函数对微秒前导零的错误解析 #3557
  - 修复将 `INT` 类型转换为 `DECIMAL` 类型可能造成溢出的问题 #3920
  - 修复将 `DATETIME` 转换为 `DECIMAL` 时结果错误的问题 #4151
  - 修复将 `FLOAT` 类型转换为 `DECIMAL` 类型可能造成溢出的问题 #3998
  - 修复 `CastStringAsReal` 在 TiFlash 的行为与在 TiDB、TiKV 的行为不一致的问题 #3475
  - 修复 `CastStringAsDecimal` 在 TiFlash 的行为与在 TiDB、TiKV 的行为不一致的问题 #3619
  - 修复 TiFlash 重启时偶发的 `EstablishMPPConnection` 错误 #3615

  - 修复当设置 TiFlash 副本数为 0（即删除数据）后数据无法回收的问题 #3659
  - 修复当主键为 handle 时，扩宽主键列可能导致的数据不一致问题 #3569
  - 修复 SQL 语句中含有极长嵌套表达式时可能出现的解析错误 #3354
  - 修复查询语句包含 `where <string>` 时查询结果出错的问题 #3447
  - 修复 `new_collations_enabled_on_first_bootstrap` 开启后查询结果可能出错的问题 #3388, #3391
  - 修复启用 TLS 时可能导致的崩溃 #4196
  - 修复启用内存限制后 TiFlash 崩溃的问题 #3902
  - 修复当 MPP 查询被终止时，TiFlash 偶发的崩溃问题 #3401
  - 修复非预期的 `Unexpected type of column: Nullable(Nothing)` 错误 #3351
  - 修复在滞后的 Region peer 上执行 Region Merge 导致的元数据损坏问题 #4437
  - 修复在执行带有 JOIN 的查询遇到错误时可能被挂起的问题 #4195
  - 修复不正确的执行计划可能导致 MPP 查询出错的问题 #3389

- Tools

  - Backup & Restore (BR)

    * 修复 BR 无法备份 RawKV 的问题 #32607

  - TiCDC

    * 修复不支持同步默认值的问题 #3793
    * 修复某些情况下序列对象被错误同步的问题 #4552
    * 修复了 TiCDC 进程在 PD leader 被杀死时的异常退出问题 #4248
    * 修复 MySQL sink 在禁用 `batch-replace-enable` 参数时生成重复 `replace` SQL 语句的错误 #4501
    * 修复输出默认列值时的 panic 问题和数据不一致的问题 #3929
    * 修复 `mq sink write row` 没有监控数据的问题 #3431
    * 修复当 `min.insync.replicas` 小于 `replication-factor` 时不能同步的问题 #3994
    * 修复在移除同步任务后潜在的 panic 问题 #3128
    * 修复了 HTTP API 在查询的组件不存在时导致 CDC 挂掉的问题 #3840
    * 修复因 checkpoint 不准确导致的潜在的数据丢失问题 #3545
    * 修复潜在的同步流控死锁问题 #4055
    * 修复人为删除 etcd 任务的状态时导致 TiCDC panic 的问题 #2980
    * 修复 DDL 特殊注释导致的同步停止的问题 #3755
    * 修复 `config.Metadata.Timeout` 没有正确配置而导致的同步停止问题 #3352
    * 修复在某些 RHEL 发行版上因时区问题导致服务无法启动的问题 #3584
    * 修复集群升级后 `stopped` 状态的 changefeed 自动恢复的问题 #3473
    * 修复 MySQL sink 模块出现死锁时告警过于频繁的问题 #2706
    * 修复 Canal 和 Maxwell 协议下 TiCDC 没有自动开启 `enable-old-value` 选项的问题 #3676
    * 修复 Avro sink 模块不支持解析 JSON 类型列的问题 #3624
    * 修复监控 checkpoint lag 出现负值的问题 #3010
    * 修复在容器环境中 OOM 的问题 #1798
    * 修复执行 DDL 后的内存泄漏的问题 #3174
    * 修复在同一节点反复调入调出一张表可能会导致同步任务 (changefeed) 被卡住的问题 #4464
    * 修复当 PD 状态不正常时 OpenAPI 可能会卡住的问题 #4778
    * 修复切换 owner 会导致其 metrics 数据不正确的问题 #4774
    * 修复 Unified Sorter 的 workerpool 稳定性问题 #4447
    * 修复 `cached region` 监控指标为负数的问题 #4300

- TiDB Lightning
  * 修复当 TiDB Lightning 没有权限访问 `mysql.tidb` 表时，导入的结果不正确的问题 #31088
  * 修复了 checksum 报错“GC life time is shorter than transaction duration” #32733
  * 修复在某些导入操作没有包含源文件时，TiDB Lightning 不会删除 metadata schema 的问题 #28144
  * 修复 S3 存储路径不存在时 TiDB Lightning 不报错的问题 #28031 #30709
  * 修复在 GCS 上遍历超过 1000 个 key 时会出错的问题 #30377

16.15.2 TiDB 5.2.3 Release Note

发版日期：2021 年 12 月 3 日

TiDB 版本：5.2.3

16.15.2.1 Bug 修复

- TiKV
  - 修复 `GcKeys` 任务被多个键调用时无法正常进行，导致 Compaction Filter GC 可能不删除 MVCC deletion 信息的问题 #11217

16.15.3 TiDB 5.2.2 Release Notes

发版日期：2021 年 10 月 29 日

TiDB 版本：5.2.2

16.15.3.1 提升改进

- TiDB
  - 当 coprocessor 遇到锁时，在调试日志中显示受影响的 SQL 语句帮助诊断问题 #27718
  - 在 SQL 逻辑层备份和恢复数据时，支持显示备份和恢复数据的大小 #27247
- TiKV
  - 简化 L0 层流控算法 #10879
  - 优化 raft client 错误日志的收集 #10983
  - 优化日志线程以避免其成为性能瓶颈 #10841
  - 添加更多的写入查询统计类型 #10507
- PD
  - 热点调度器的 QPS 维度支持更多的写请求类型 #3869
  - 通过动态调整 Balance Region 调度器的重试上限，优化该调度器的性能 #3744
  - 将 TiDB Dashboard 升级至 v2021.10.08.1 #4070
  - 允许 Evict Leader 调度器调度拥有不健康副本的 Region #4093
  - 优化调度器退出的速度 #4146

- Tools
  - TiCDC
    * 通过修改 Kafka sink 配置项 MaxMessageBytes 的默认值，由 64 MB 减小为 1 MB，以修复消息过大会被 Kafka Broker 拒收的问题 #3104
    * 减少同步链路中的内存占用 #2553#3037 #2726
    * 优化监控项和告警规则，提升了同步链路、内存 GC、存量数据扫描过程的可观测性 #2735 #1606 #3000 #2985 #2156
    * 当同步任务状态正常时，不再显示历史错误信息，避免误导用户 #2242

16.15.3.2 Bug 修复

- TiDB
  - 修复 `plan cache` 无法感知 unsigned 标志变化的问题 #28254
  - 修复当分区功能出现 `out of range` 时 `partition pruning` 出错的问题 #28233
  - 修复在某些情况下 Planner 可能缓存无效 `join` 计划的问题 #28087
  - 修复 hash 列为 enum 时构建错误 `IndexLookUpJoin` 的问题 #27893
  - 修复批处理客户端在某些罕见情况下回收空闲连接可能会阻塞发送请求的问题 #27688
  - 修复当 TiDB Lightning 在目标集群上执行校验失败时 panic 的问题 #27686
  - 修复某些情况下 `date_add` 和 `date_sub` 函数执行结果错误的问题 #27232
  - 修复 hour 函数在向量化表达式中执行结果错误的问题 #28643
  - 修复连接到 MySQL 5.1 或更早的客户端时存在的认证问题 #27855
  - 修复当新增索引时自动分析可能会在指定时间之外触发的问题 #28698
  - 修复设置任何会话变量都会使 `tidb_snapshot` 失效的问题 #28683
  - 修复在有大量 `miss-peer region` 的集群中 BR 不可用的问题 #27534
  - 修复当不支持的 `cast` 被下推到 TiFlash 时出现的非预期错误，例如 `tidb_cast to Int32 is not supported` #23907
  - 修复 `%s value is out of range in '%s'` 报错中缺失 `DECIMAL overflow` 信息的问题 #27964
  - 修复 MPP 节点的可用性检测在某些边界场景中无法工作的问题 #3118
  - 修复分配 `MPP task ID` 时出现 `DATA RACE` 的问题 #27952
  - 修复删除空的 `dual table` 后 MPP 查询出现 `index out of range` 报错的问题 #28250
  - 修复运行 MPP 查询时出现 `invalid cop task execution summaries length` 相关日志的问题 #1791
  - 修复运行 MPP 查询时出现 `cannot found column in Schema column` 报错的问题 #28149
  - 修复 TiDB 在 TiFlash 关闭时可能出现 panic 的问题 #28096
  - 移除对基于 3DES (三重数据加密算法) 不安全的 TLS 加密套件的支持 #27859
  - 修复因 Lightning 前置检查会连接已下线的 TiKV 节点而导致导入失败的问题 #27826
  - 修复在导入太多文件到表时前置检查花费太多时间的问题 #27605
  - 修复表达式重写时 between 推断出错误排序规则的问题 #27146
  - 修复 `group_concat` 函数没有考虑排序规则的问题 #27429
  - 修复 `extract` 函数处理负值时的问题 #27236
  - 修复当设置 `NO_UNSIGNED_SUBTRACTION` 时创建分区失败的问题 #26765
  - 避免在列修剪和聚合下推中使用有副作用的表达式 #27106
  - 删除无用的 gRPC 日志 #24190
  - 限制有效的小数点长度以修复精度相关的问题 #3091

  - 修复 `plus` 表达式中检查溢出方法出错的问题 #26977
  - 修复当导出带有 `new collation` 数据的表的统计信息时报 `data too long` 错误的问题 #27024
  - 修复 `TIDB_TRX` 中不包含重试事务的问题 #28670

- TiKV

  - 修复因 Congest 错误而导致的 CDC 频繁增加 scan 重试的问题 #11082
  - 修复因 channel 打满而导致的 Raft 断连情况 #11047
  - 修复 Raft client 中 batch 消息过大的问题 #9714
  - 修复 `resolved_ts` 中协程泄漏的问题 #10965
  - 修复当 response 大小超过 4 GiB 时 Coprocessor panic 的问题 #9012
  - 修复当一个 snapshot 文件无法被垃圾清理 (GC) 时 snapshot GC 会缺失 GC snapshot 文件的问题 #10813
  - 修复当处理 Coprocessor 请求时因超时而导致 panic 的问题 #10852

- PD

  - 修复因超过副本配置数量而导致错误删除带有数据且处于 pending 状态的副本的问题 #4045
  - 修复 PD 未能及时修复 Down Peer 副本的问题 #4077
  - 修复 Scatter Range 调度器无法对空 Region 进行调度的问题 #4118
  - 修复 key manager 占用过多 CPU 的问题 #4071
  - 修复热点调度器变更配置的过程中可能会存在的数据竞争问题 #4159
  - 修复因 Region syncer 卡住而导致 leader 选举慢的问题 #3936

- TiFlash

  - 修复 TiFlash 在部分平台上由于缺失 `nsl` 库而无法启动的问题

- Tools

  - TiCDC

    * 修复当上游 TiDB 实例意外退出时，TiCDC 同步任务推进可能停滞的问题 #3061
    * 修复当 TiKV 向同一 Region 发送重复请求时，TiCDC 进程 panic 的问题 #2386
    * 修复在验证下游 TiDB/MySQL 可用性时产生的不必要的 CPU 消耗 #3073
    * 修复 TiCDC 产生的 Kafka 消息体积不受 `max-message-size` 约束的问题 #2962
    * 修复当写入 Kafka 消息发生错误时，TiCDC 同步任务推进可能停滞的问题 #2978
    * 修复当开启 `force-replicate` 时，可能某些没有有效索引的分区表被忽略的问题 #2834
    * 修复当扫描存量数据耗时过长时，可能由于 TiKV 进行 GC 而导致存量数据扫描失败的问题 #2470
    * 修复在将某些类型的列编码为 Open Protocol 格式时，TiCDC 进程可能 panic 的问题 #2758
    * 修复在将某些类型的列编码为 Avro 格式时，TiCDC 进程可能 panic 的问题 #2648

  - TiDB Binlog

    * 修复当大部分表被过滤掉时，在某些特殊的负载下，checkpoint 不更新的问题 #1075

16.15.4 TiDB 5.2.1 Release Notes

发版日期：2021 年 9 月 9 日

TiDB 版本：5.2.1

16.15.4.1 Bug 修复

- TiDB
  - 修复在分区中下推聚合算子时，因浅拷贝 schema 列导致执行计划出错，进而导致执行时报错的问题 #27797 #26554
- TiKV
  - 修复 Region 迁移时 Raftstore 模块出现死锁导致 TiKV 不可用的问题。用户可通过关闭调度并重启出问题的 TiKV 来临时应对。 #10909

16.15.5 TiDB 5.2 Release Notes

发版日期：2021 年 8 月 27 日

TiDB 版本：5.2.0

> **警告：**
>
> 该版本存在一些已知问题，已在新版本中修复，建议使用 5.2.x 的最新版本。

在 5.2 版本中，你可以获得以下关键特性：

- 支持基于部分函数创建表达式索引 (Expression index)，极大提升查询的性能。
- 提升优化器的估算准确度 (Cardinality Estimation)，有助于选中最优的执行计划。
- 锁视图 (Lock View) 成为 GA 特性，提供更直观方便的方式观察事务加锁情况以及排查死锁问题。
- 新增 TiFlash I/O 限流功能，提升 TiFlash 读写稳定性。
- 为 TiKV 引入新的流控机制代替之前的 RocksDB write stall 流控机制，提升 TiKV 流控稳定性。
- 简化 Data Migration (DM) 工具运维，降低运维管理的成本。
- TiCDC 支持 HTTP 协议 OpenAPI 对 TiCDC 任务进行管理，在 Kubernetes 以及本地部署环境下提供更友好的运维方式。(实验特性)

16.15.5.1 兼容性更改

> **注意：**
>
> 当从一个早期的 TiDB 版本升级到 TiDB 5.2 时，如需了解所有中间版本对应的兼容性更改说明，请查看对应版本的Release Note。

16.15.5.1.1 系统变量

变量名	修改类型	描述
default_authentication_plugin	新增	设置服务器对外通告的默认身份验证方式，默认值为 mysql_native_password。
tidb_enable_auto_increment_in_generated	新增	控制是否允许在创建生成列或者表达式索引时引用自增列，默认值为OFF。
tidb_opt_enable_correlation_adjustment	新增	控制优化器是否开启交叉估算，默认值为ON。
tidb_opt_limit_push_down_threshold	新增	设置将 Limit 和 TopN 算子下推到 TiKV 的阈值，默认值为100。
tidb_stmt_summary_max_stmt_count	修改	表示 statement summary 在内存中保存的语句的最大数量。默认值从 200 修改为 3000。
tidb_enable_streaming	废弃	系统变量 enable-streaming已废弃，不建议再使用。

16.15.5.1.2 配置文件参数

配置文件	配置项	修改类型	描述
TiDB 配置文件	pessimistic-txn.deadlock-history-collect-retryable	新增	控制INFORMATION_SCHEMA.DEADLOCKS表中是否收集可重试的死锁错误信息。

配置文件	配置项	修改类型	描述
TiDB 配置文件	security.auto-tls	新增	控制 TiDB 启动时是否自动生成 TLS 证书，默认值为 false。
TiDB 配置文件	stmt-summary.max-stmt-count	修改	表示 statement summary tables 中保存的 SQL 种类的最大数量。默认值从 200 修改为 3000。
TiDB 配置文件	experimental.allow-expression-index	废弃	废弃 TiDB 配置文件中allow-expression-index 配置项
TiKV 配置文件	raftstore.cmd-batch	新增	对请求进行攒批的控制开关，开启后可显著提升写入性能。默认值为 true。
TiKV 配置文件	raftstore.inspect-interval	新增	TiKV 每隔一段时间会检测 Raftstore 组件的延迟情况，该配置项设置检测的时间间隔。当检测的延迟超过该时间，该检测会被记为超时。默认值为 500ms。
TiKV 配置文件	raftstore.max-peer-down-duration	修改	表示副本允许的最长未响应时间，超过将被标记为 down，后续 PD 会尝试将其删掉。默认值从 5m 修改为 10m。

配置文件	配置项	修改类型	描述
TiKV 配置文件	server.raft-client-queue-size	新增	指定 TiKV 中发送 Raft 消息的缓冲区大小。默认值为 8192。
TiKV 配置文件	storage.flow-control.enable	新增	表示是否开启 TiKV 流量控制机制。默认值为 true。
TiKV 配置文件	storage.flow-control.memtables-threshold	新增	当 KvDB 的 memtable 的个数达到该阈值时，流控机制开始工作。默认值为 5。
TiKV 配置文件	storage.flow-control.l0-files-threshold	新增	当 KvDB 的 L0 文件个数达到该阈值时，流控机制开始工作。默认值为 9。
TiKV 配置文件	storage.flow-control.soft-pending-compaction-bytes-limit	新增	当 KvDB 的 pending compaction bytes 达到该阈值时，流控机制开始拒绝部分写入请求并报错。默认值为“192GB”。
TiKV 配置文件	storage.flow-control.hard-pending-compaction-bytes-limit	新增	当 KvDB 的 pending compaction bytes 达到该阈值时，流控机制开始拒绝所有写入请求并报错。默认值为“1024GB”。

16.15.5.1.3 其他

- 升级前，请检查系统变量tidb_evolve_plan_baselines 的值是否为 ON。如果为 ON，需要将其改成 OFF，否则会导致升级失败。

- v4.0 集群升级到 v5.2 集群后，`tidb_multi_statement_mode` 变量的默认值由 WARN 变为 OFF。
- 升级前，请检查 TiDB 配置项 `feedback-probability` 的值。如果不为 0，升级后会触发 “panic in the recoverable goroutine” 报错，但不影响升级。
- 兼容 MySQL 5.7 的 noop 变量 `innodb_default_row_format`，配置此变量无实际效果 #23541。
- 从 TiDB 5.2 起，为了提高系统安全性，建议（但不要求）对来自客户端的连接进行传输层加密，TiDB 提供 Auto TLS 功能在 TiDB 服务器端自动配置并开启加密。要使用 Auto TLS 功能，请在 TiDB 升级前将 TiDB 配置文件中的`security.auto-tls` 设置为 `true`。
- 支持 `caching_sha2_password` 身份验证方式，简化了从 MySQL 8.0 的迁移操作，并提升了安全性。

16.15.5.2 新功能

16.15.5.2.1 SQL

- 支持基于部分函数创建表达式索引 (Expression index)

  表达式索引是一种特殊的索引，能将索引建立于表达式上。创建了表达式索引后，TiDB 支持基于表达式的查询，极大提升查询的性能。

  用户文档，#25150

- 支持 Oracle 中的 `translate`函数

  `translate` 函数可以将字符串中出现的所有指定字符替换为其它字符，TiDB 中的 `translate` 函数不会像 Oracle 一样将空字符串视为NULL。

  用户文档

- 支持 Spilling HashAgg

  支持 HashAgg 的落盘。当包含 HashAgg 算子的 SQL 语句引起 OOM 时，可以尝试设置算子的并发度为 1 来触发落盘，缓解 TiDB 内存压力。

  用户文档，#25882

- 提升优化器的估算准确度 (Cardinality Estimation)

  - 提升 TiDB 对 TopN/Limit 估算的准确度。例如，对于包含 `order by col limit x` 的大表分页查询，TiDB 可以更容易地选对索引，降低查询响应时间。
  - 提升对越界估算的准确度。例如，在当天统计信息尚未更新的情况下，对于包含 `where date=Now()` 的查询，TiDB 也能准确地选中对应索引。
  - 引入变量 `tidb_opt_limit_push_down_threshold` 控制优化器对 Limit/TopN 的下推行为，可以解决部分情况下因为估算误差导致 Limit/TopN 不能下推的问题。

  用户文档，#26085

- 提升优化器的索引过滤规则 (Index Selection)

  新增加了一些索引选择的裁剪规则，在通过统计信息进行对比之前，通过规则进一步对可能的选择的索引范围进行缩小。从而减小各种情况下选到不优索引的概率。

  用户文档

16.15.5.2.2 事务

- 锁视图 (Lock View) 成为 GA 特性

  Lock View 用于提供关于悲观锁的锁冲突和锁等待的更多信息，方便 DBA 通过锁视图功能来观察事务加锁情况以及排查死锁问题等。

  在 5.2 中，Lock View 新增以下特性：

  - 对于 Lock View 所属的各张表中的 SQL Digest 列，v5.2 额外增加了一列显示对应的归一化的 SQL 语句文本，无需手动查询 SQL Digest 对应的语句。
  - 增加了 TIDB_DECODE_SQL_DIGESTS 函数用于在集群中查询一组 SQL Digest 所对应的 SQL 语句的归一化形式（即去除格式和参数后的形式），简化了查询某一事务历史执行过的语句的操作
  - 在 DATA_LOCK_WAITS 和 DEADLOCKS 系统表中，增加一列显示 key 中解出的表名、row id、索引值等信息，简化了定位 key 所属的表、解读 key 的内容等信息的操作。
  - 支持在 DEADLOCKS 表中收集可重试的死锁错误的信息，以便于排查因可重试的死锁引发的问题。默认不收集，可通过配置选项 pessimistic-txn.deadlock-history-collect-retryable 启用。
  - TIDB_TRX 系统表支持区分正在执行查询的事务和闲置中的事务，即将原来的 Normal 状态拆分成 Running 和 Idle 状态。

  用户文档：

  - 查看集群中所有 TiKV 节点上当前正在发生的悲观锁等锁：DATA_LOCK_WAITS
  - 查看 TiDB 节点上最近发生的若干次死锁错误：DEADLOCKS
  - 查看 TiDB 节点上正在执行的事务的信息：TIDB_TRX

- 对带有 AUTO_RANDOM 或者 SHARD_ROW_ID_BITS 属性的表，优化其大部分添加索引操作的场景。

16.15.5.2.3 稳定性

- 新增 TiFlash I/O 限流功能

  TiFlash I/O 限流功能主要针对磁盘带宽较小且明确知道磁盘带宽大小的云盘场景，默认关闭。

  TiFlash I/O Rate Limiter 提供了一个新的防止“读/写”任务之间过度竞争系统 IO 资源的机制，可以平衡系统对“读”和“写”任务的响应，并且可以根据读/写负载的情况自动限速。

  用户文档

- 提升 TiKV 流控稳定性

  TiKV 引入了新的流控机制代替之前的 RocksDB write stall 流控机制。相比于 write stall 机制，新的流控机制通过以下改进减少了流控对前台写入稳定性的影响：

  - 当 RocksDB compaction 压力堆积时，通过在 TiKV scheduler 层进行流控而不是在 RocksDB 层进行流控，避免 RocksDB write stall 造成的 raftstore 卡顿并造成 Raft 选举超时导致发生节点 leader 迁移的问题。
  - 改善流控算法，有效降低大写入压力下导致 QPS 下降的问题

  用户文档，#10137

- 自动检测并恢复集群中单个 TiKV 变慢带来的影响

  在 TiKV 中引入了慢节点检测机制，通过检测 TiKV Raftstore 的快慢来计算出一个分数，并通过 Store Heartbeat 上报给 PD。并且在 PD 上增加了 `evict-slow-store-scheduler` 调度器，能够自动驱逐单个变慢的 TiKV 节点上的 Leader，以降低其对整个集群性能的影响。同时，还增加了慢节点相关的报警项，帮助用户快速发现并处理问题。

  用户文档，#10539

16.15.5.2.4 数据迁移

- 简化 Data Migration (DM) 工具运维

  DM v2.0.6 支持自动识别使用 VIP 的数据源实例切换事件（failover/计划切换），自动连接上新的数据源实例，减少数据复制的延迟和减少运维操作步骤

- TiDB Lightning 支持自定义 CSV 数据的终止符，兼容 MySQL LOAD DATA CSV 数据格式。使得 TiDB Lightning 可以直接使用在用户数据流转架构体系中。#1297

16.15.5.2.5 TiDB 数据共享订阅

TiCDC 支持 HTTP 协议 OpenAPI 对 TiCDC 任务进行管理，在 Kubernetes 以及本地部署环境下提供更友好的运维方式。(实验特性）

#2411

16.15.5.2.6 部署及运维

支持在使用 Apple M1 芯片的本地 Mac 机器上使用 `tiup playground` 命令。

16.15.5.3 功能增强

- Tools
  - TiCDC
    * 新增专为 TiDB 设计的比基于 JSON 的开放协议更紧凑的二进制 MQ 格式 #1621
    * 移除对 file sorter 的支持 #2114
    * 支持日志轮替配置 #2182
  - TiDB Lightning
    * 支持 CSV 文件中除 `\r` 和 `\n` 之外的自定义行尾 #1297
    * 支持表达式索引和依赖于虚拟生成列的索引 #1407
  - Dumpling
    * 支持备份兼容 MySQL 但不支持 `START TRANSACTION ... WITH CONSISTENT SNAPSHOT` 和 `SHOW CREATE TABLE` 语句的数据库 #311

16.15.5.4 提升改进

- TiDB
  - 支持将内置函数 `json_unquote()` 下推到 TiKV #24415
  - 支持在 Dual 表上移除 Union 算子的优化 #25614
  - 优化聚合算子的代价常数 #25241
  - 允许 MPP outer join 根据表行数选择构建表 #25142
  - 支持 MPP 查询任务按 Region 均衡到不同 TiFlash 节点上 #24724
  - 支持执行 MPP 查询后将缓存中过时的 Region 无效化 #24432
  - 提升内置函数 `str_to_date` 在格式指定器中 `%b/%M/%r/%T` 的 MySQL 兼容性 #25767
  - 修复因对同一条查询重复创建不同 binding 可能导致的多个 TiDB 上 binding cache 不一致的问题 #26015
  - 修复升级可能会导致的 binding 无法被加载到缓存的问题 #23295
  - 对 `SHOW BINDINGS` 结果按照 (original_sql, update_time) 有序输出 #26139
  - 改进使用 binding 优化查询的逻辑，减少对查询的优化次数 #26141
  - 支持标记为删除状态的 binding 进行自动垃圾回收 #26206
  - 在 `EXPLAIN VERBOSE` 的结果中显示查询优化是否使用了某个 binding #26930
  - 增加新的状态变量 `last_plan_binding_update_time` 用于查看当前 TiDB 实例中 binding cache 对应的时间戳 #26340
  - 在打开 binding 演进或者执行 `admin evolve bindings` 时提供报错，避免自动演进绑定（目前为试验特性，已在当前 TiDB 版本关闭）影响到其他功能 #26333
- PD
  - 热点调度增加对 QPS 维度的支持，同时支持调整维度的优先级顺序#3869
  - 热点调度支持对 TiFlash 的写热点进行调度 #3900
- TiFlash
  - 新增若干运算符的支持：`MOD / %`, `LIKE`
  - 新增若干字符串函数的支持：`ASCII()`, `COALESCE()`, `LENGTH()`, `POSITION()`, `TRIM()`
  - 新增若干数学函数的支持：`CONV()`, `CRC32()`, `DEGREES()`, `EXP()`, `LN()`, `LOG()`, `LOG10()`, `LOG2()`, `POW()`, `RADIANS()`, `ROUND(decimal)`, `SIN()`, `MOD()`
  - 新增若干日期函数的支持：`ADDDATE(string, real)`, `DATE_ADD(string, real)`, `DATE()`
  - 新增更多的函数支持：`INET_NTOA()`, `INET_ATON()`, `INET6_ATON`, `INET6_NTOA()`
  - 当 new collation 打开时，支持 MPP 模式下的 Shuffled Hash Join 和 Shuffled Hash Aggregation 运算
  - 优化基础代码提升 MPP 性能
  - 支持将 `STRING` 类型转换为 `DOUBLE` 类型
  - 通过多线程优化右外连接中的非连接数据
  - 支持在 MPP 查询中自动清理过期的 Region 信息
- Tools
  - TiCDC
    * 为 kv client 增量扫添加并发限制 #1899
    * 始终在 TiCDC 内部拉取 old value #2271
    * 当遇到不可恢复的 DML 错误，TiCDC 快速失败并退出 #1928

    * 在 Region 初始化后不立即执行 resolve lock #2235
    * 优化 workerpool 以降低在高并发情况下 goroutine 的数量 #2201
  - Dumpling
    * 通过 `tidb_rowid` 对 TiDB v3.x 的表进行数据划分以节省 TiDB 的内存 #301
    * 减少 Dumpling 对 `information_schema` 库的访问以提高稳定性 #305

16.15.5.5 Bug 修复

- TiDB
  - 修复在 SET 类型列上 Merge Join 结果不正确的问题 #25669
  - 修复 IN 表达式参数的数据腐蚀问题 #25591
  - 避免 GC 的 session 受全局变量的影响 #24976
  - 修复了在窗口函数查询中使用 `Limit` 时出现 panic 问题 #25344
  - 修复查询分区表时使用 `Limit` 返回错误值的问题 #24636
  - 修复了 IFNULL 在 ENUM 或 SET 类型上不能正确生效的问题 #24944
  - 修复了 Join 子查询中的 count 被改写为 `first_row` 导致结果不正确的问题 #24865
  - 修复了 TopN 算子下使用 `ParallelApply` 查询时卡住的问题 #24930
  - 修复了使用含有多列的前缀索引查询时出现多余结果的问题 #24356
  - 修复了操作符 <=> 不能正确生效的问题 #24477
  - 修复并行 Apply 算子的数据竞争问题 #23280
  - 修复对 PartitionUnion 算子的 IndexMerge 结果排序时出现 `index out of range` 错误 #23919
  - 修复 `tidb_snapshot` 被允许设置为非预期的过大值，而可能造成事务隔离性被破坏的问题 #25680
  - 修复 ODBC 类常数（例如 `{d '2020-01-01'}`）不能被用作表达式的问题 #25531
  - 修复 SELECT DISTINCT 被转化为 Batch Get 而导致结果不正确的问题 #25320
  - 修复无法触发将查询从 TiFlash 回退到 TiKV 的问题 #23665 #24421
  - 修复在检查 `only_full_group_by` 时的 `index-out-of-range` 错误 #23839
  - 修复关联子查询中 Index Join 的结果不正确问题 #25799
- TiKV
  - 修复错误的 `tikv_raftstore_hibernated_peer_state` 监控指标 #10330
  - 修复 coprocessor 中 `json_unquote()` 函数错误的参数类型 #10176
  - 正常关机时跳过清理 Raftstore 的回调从而避免在某些情况下破坏事务的 ACID #10353 #10307
  - 修复在 Leader 上 Replica Read 共享 Read Index 的问题 #10347
  - 修复 coprocessor 转换 DOUBLE 到 DOUBLE 的错误函数 #25200
- PD
  - 修复多个调度器之间存在调度冲突时无法产生预期调度的问题 #3807 #3778
- TiFlash
  - 修复因 split 失败而不断重启的问题
  - 修复无法删除 Delta 历史数据的潜在问题
  - 修复在 CAST 函数中为非二进制字符串填充错误数据的问题
  - 修复处理包含复杂 GROUP BY 列的聚合查询时结果不正确的问题

- 修复写入压力过大时出现进程崩溃的问题
- 修复右连接键不为空且左连接键可为空时进程崩溃的问题
- 修复 `read-index` 请求耗时长的潜在问题
- 修复读负载高的情况下进程崩溃的问题
- 修复 `Date_Format` 函数在参数类型为 `STRING` 且包含 `NULL` 值时可能导致 TiFlash server 崩溃的问题

- Tools
  - TiCDC
    - 修复 TiCDC owner 在刷新 checkpoint 时异常退出的问题 #1902
    - 修复 changefeed 创建成功后立即失败的问题 #2113
    - 修复不合法格式的 rules filter 导致 changefeed 失败的问题 #1625
    - 修复 TiCDC Owner 崩溃时潜在的 DDL 丢失问题 #1260
    - 修复 CLI 在默认的 sort-engine 选项上与 4.0.x 集群的兼容性问题 #2373
    - 修复 TiCDC 遇到 `ErrSchemaStorageTableMiss` 错误时可能导致 changefeed 被意外重置的问题 #2422
    - 修复 TiCDC 遇到 `ErrGCTTLExceeded` 错误时 changefeed 不能被 remove 的问题 #2391
    - 修复 TiCDC 同步大表到 cdclog 失败的问题 #1259 #2424
    - 修复 TiCDC 在重新调度 table 时多个 processors 可能向同一个 table 写数据的问题 #2230
  - Backup & Restore (BR)
    - 修复 BR 恢复中忽略了所有系统表的问题 #1197 #1201
    - 修复 BR 恢复 cdclog 时漏掉 DDL 操作的问题 #870
  - TiDB Lightning
    - 修复 TiDB Lightning 解析 Parquet 文件中 `DECIMAL` 类型数据失败的问题 #1272
    - 修复 TiDB Lightning 恢复 table schema 时报错“Error 9007: Write conflict”的问题 #1290
    - 修复 TiDB Lightning 因 int handle 溢出导致导入数据失败的问题 #1291
    - 修复 TiDB Lightning 在 local backend 模式下因数据丢失可能遇到 checksum 不匹配的问题 #1403
    - 修复 TiDB Lightning 恢复 table schema 时与 clustered index 不兼容的问题 #1362
  - Dumpling
    - 修复 Dumpling GC safepoint 设置过晚导致数据导出失败的问题 #290
    - 修复 Dumpling 在特定 MySQL 版本下获取上游表名时卡住的问题 #322

16.16 v5.1

16.16.1 TiDB 5.1.5 Release Notes

发版日期：2022 年 12 月 28 日

TiDB 版本：5.1.5

试用链接：快速体验 | 生产部署 | 下载离线包

16.16.1.1 兼容性变更

- PD
  - 默认关闭编译 swagger server #4932

16.16.1.2 Bug 修复

- TiDB
  - 修复窗口函数执行时本应报错但是让 TiDB 崩溃的问题 #30326
  - 修复在 TiFlash 中为分区表开启动态模式时结果出错的问题 #37254
  - 修复当 GREATEST 和 LEAST 函数传入无符号整型值时，计算结果出错的问题 #30101
  - 修复使用 left join 同时删除多张表数据时可能出现错误结果的问题 #31321
  - 修复 `concat(ifnull(time(3)))` 的结果与 MySQL 不一致的问题 #29498
  - 修复 SQL 语句中包含 `cast(integer as char)union string` 时计算结果出错的问题 #29513
  - 修复 `INL_HASH_JOIN` 和 `LIMIT` 一起使用时可能会卡住的问题 #35638
  - 修复当有 Region 返回空数据时 ANY_VALUE 结果不正确的问题 #30923
  - 修复 innerWorker panic 导致的 index join 结果错误的问题 #31494
  - 修复当 SQL 语句中存在 JSON 类型列与 CHAR 类型列连接时，SQL 出错的问题 #29401
  - 修复 TiDB 的后台 HTTP 服务可能没有正确关闭导致集群状态异常的问题 #30571
  - 修复并发的列类型变更导致 schema 与数据不一致的问题 #31048
  - 修复 `KILL TIDB` 在空闲链接上无法立即生效的问题 #24031
  - 修复设置 SESSION 变量会导致 `tidb_snapshot` 不工作的问题 #35515
  - 修复 Region 合并情况下 Region cache 没有及时被清理的问题 #37141
  - 修复因为 KV client 中连接数据争用导致的 panic 问题 #33773
  - 修复在开启 TiDB Binlog 时，TiDB 执行 `ALTER SEQUENCE` 会产生错误的元信息版本号，进而导致 Drainer 报错退出的问题 #36276
  - 修复 TiDB 由于 `fatal error: concurrent map read and map write` 发生崩溃的问题 #35340
  - 修复在 TiFlash 不支持使用空范围读表的情况，依然选择 TiFlash 导致查询结果错误的问题 #33083
  - 修复当从 TiFlash 查询 `avg()` 函数时，返回错误 `ERROR 1105 (HY000): other error for mpp stream ↪ : Could not convert to the target type - -value is out of range.` 的问题 #29952
  - 修复查询 HashJoin 时，返回错误 `ERROR 1105 (HY000): close of nil channel` 的问题 #30289
  - 修复 TiKV 和 TiFlash 在进行逻辑运算时结果不一致的问题 #37258
  - 修复某些情况下，`EXECUTE` 语句可能抛出非预期异常的问题 #37187
  - 修复了 `tidb_opt_agg_push_down` 和 `tidb_enforce_mpp` 启用时执行处理器的错误行为 #34465
  - 修复 TiDB 在执行 `SHOW COLUMNS` 时会发出协处理器请求的问题 #36496
  - 当 `enable-table-lock` 参数未开启时为 `lock tables` 和 `unlock tables` 新增警告 #28967
  - 修复 Range 分区允许多个 MAXVALUE 分区的问题 #36329
- TiKV
  - 修复了 DATETIME 类型的数据包含小数部分和 Z 后缀导致检查报错的问题 #12739
  - 修复 replica read 可能违反线性一致性的问题 #12109
  - 修复 Raftstore 线程繁忙时，可能会出现 Region 重叠的问题 #13160
  - 修复 apply snapshot 被暂停时会引起 TiKV panic 的问题 #11618
  - 修复 TiKV 运行 2 年以上可能 panic 的问题 #11940
  - 修复在 Region merge 时 source peer 通过 snapshot 追日志时可能导致 panic 的问题 #12663
  - 修复了对空字符串进行类型转换导致 TiKV panic 的问题 #12673
  - 修复旧信息造成 TiKV panic 的问题 #12023
  - 修复同时分裂和销毁一个 peer 时可能导致 panic 的问题 #12825
  - 修复合并 Region 时因 target peer 被一个未进行初始化就被销毁的 peer 所替换，从而引起 TiKV panic 的问题 #12048

  - 修复进行 Follower Read 时，可能会报 `invalid store ID 0` 错误的问题 #12478
  - 修复了在悲观事务中使用 Async Commit 导致重复提交记录的问题 #12615
  - 支持配置 `unreachable_backoff` 避免 Raftstore 发现某个 Peer 无法连接时广播过多消息 #13054
  - 修复网络出现问题的情况下，已成功提交的乐观事务可能报 `Write Conflict` 错误的问题 #34066
  - 修复 Dashboard 中 Unified Read Pool CPU 表达式错误的问题 #13086

- PD

  - 修复已清除的 `tombstone store` 信息在切换 PD leader 后再次出现的问题 #4941
  - 修复 PD leader 转移后调度不能立即启动的问题 #4769
  - 修复 `not leader` 的 status code 有误的问题 #4797
  - 修复 PD 无法正确处理 dashboard 代理请求的问题 #5321
  - 修复在某些特殊情况下 TSO fallback 的问题 #4884
  - 修复 PD 在特定条件下不会创建 TiFlash Learner 副本的问题 #5401
  - 修复监控信息中已删除 label 的残留问题 #4825
  - 修复存在较大空间 Store 时（例如 2T），无法检测满的小空间 Store，从而无法进行平衡调度的问题 #4805
  - 修复 `SchedulerMaxWaitingOperator` 设置为 1 时不产生调度的问题 #4946

- TiFlash

  - 修复转换 string 类型为 datetime 类型时，`microsecond` 结果可能不正确的问题 #3556
  - 修复启用 TLS 时可能导致的崩溃 #4196
  - 修复并行聚合出错时可能导致 TiFlash crash 的问题 #5356
  - 修复在执行带有 JOIN 的查询遇到错误时可能被挂起的问题 #4195
  - 修复 OR 函数计算结果错误的问题 #5849
  - 修复错误地配置存储目录会导致非预期行为的问题 #4093
  - 修复大量 INSERT 和 DELETE 操作后可能导致 TiFlash 数据不一致的问题 #4956
  - 修复 TiFlash 节点上遗留了与 Region range 不匹配的数据的问题 #4414
  - 修复在读取工作量大时添加列后可能出现的查询错误 #3967
  - 修复由于 `commit state jump backward` 错误导致 TiFlash 反复崩溃的问题 #2576
  - 修复查询存在大量 delete 操作的表时可能报错的问题 #4747
  - 修复日期格式将 `''` 处理为非法分隔符的问题 #4036
  - 修复将 DATETIME 转换为 DECIMAL 时结果错误的问题 #4151
  - 修复一些异常没有被正确地处理的问题 #4101
  - 修复 `Prepare Merge` 可能导致 raft 状态机元数据损坏，从而引起 TiFlash 重启的问题 #3435
  - 修复 MPP query 会随机碰到 gRPC keepalive timeout 导致 query 失败的问题 #4662
  - 修复 IN 函数的结果在多值表达式中不正确的问题 #4016
  - 修复 MPP 任务可能永远泄漏线程的问题 #4238
  - 修复过期数据回收缓慢的问题 #4146
  - 修复将 FLOAT 类型转换为 DECIMAL 类型可能造成溢出的问题 #3998
  - 修复逻辑运算符在 UInt8 类型下查询结果出错的问题 #6127
  - 修复 `json_length` 对空字符串可能会报 `index out of bounds` 错误的问题 #2705
  - 修复极端情况下 decimal 比较结果可能有误的问题 #4512
  - 修复在添加一些 NOT NULL 的列时报 `TiFlash_schema_error` 的问题 #4596
  - 修复由于使用 `0.0` 作为整数类型的默认值导致 TiFlash 节点失败的问题，比如`` `i` int(11)NOT NULL DEFAULT '0.0' `` #3157

- Tools
  - TiDB Binlog
    * 修复 `compressor` 设为 `gzip` 时 Drainer 无法正确发送请求至 Pump 的问题 #1152
  - Backup & Restore (BR)
    * 修复因为并发备份系统表，导致表名更新失败，无法恢复系统表的问题 #29710
  - TiCDC
    * 修复了增量扫描特殊场景下的数据丢失问题 #5468
    * 修复 Sorter 组件缺失监控数据的问题 #5690
    * 优化了 ddl schema 缓存方式，降低了内存消耗 #1386

16.16.2 TiDB 5.1.4 Release Notes

发版日期：2022 年 2 月 22 日

TiDB 版本：5.1.4

16.16.2.1 兼容性更改

- TiDB
  - 将系统变量`tidb_analyze_version` 的默认值从 2 修改为 1 #31748
  - 自 v5.1.4 起，TiKV 在开启 `storage.enable-ttl` 后会拒绝 TiDB 的请求，因为 TiKV 的 TTL 功能仅支持 RawKV 模式 #27303
- Tools
  - TiCDC
    * 将 `max-message-bytes` 默认值设置为 10M #4041

16.16.2.2 提升改进

- TiDB
  - 支持在 Range 类型分区表中对 `IN` 表达式进行分区裁剪 #26739
  - 提高在 `IndexJoin` 执行过程中追踪内存占用的准确度 #28650
- TiKV
  - 将 proc filesystem (procfs) 升级至 0.12.0 版本 #11702
  - 优化 Raft client 错误日志的收集 #11959
  - 将插入 SST 文件时的校验操作从 Apply 线程池移动到 Import 线程池，从而提高 SST 文件的插入速度 #11239
- PD

- 优化调度器退出的速度 #4146

- TiFlash
  - 添加 ADDDATE() 和 DATE_ADD() 到 TiFlash 的下推支持
  - 添加 INET6_ATON() 和 INET6_NTOA() 到 TiFlash 的下推支持
  - 添加 INET_ATON() 和 INET_NTOA() 到 TiFlash 的下推支持
  - 把 DAG Request 中表达式或者执行计划树的最大深度限制从 100 提升到 200

- Tools
  - TiCDC
    * 为 changefeed 重启操作添加指数退避机制 #3329
    * 降低在同步大量表时的同步延时 #3900
    * 增加观察 incremental scan 剩余时间的指标 #2985
    * 减少 “EventFeed retry rate limited” 日志的数量 #4006
    * 增加更多 Prometheus 和 Grafana 监控告警参数，包括 no owner alert、mounter row、table sink total row 和 buffer sink total row #4054 #1606
    * 优化 TiKV 重新加载时的速率限制控制，缓解 changefeed 初始化时 gPRC 的拥堵问题 #3110
    * 减少 TiKV 节点宕机后 KV client 恢复的时间 #3191

16.16.2.3 Bug 修复

- TiDB
  - 修复当系统变量 @@tidb_analyze_version = 2 时出现的内存泄露问题 #32499
  - 修复 MaxDays 和 MaxBackups 配置项对慢日志不生效的问题 #25716
  - 修复 INSERT ... SELECT ... ON DUPLICATE KEY UPDATE 语句 panic 的问题 #28078
  - 修复使用 ENUM 类型的列进行 Join 时结果可能不正确的问题 #27831
  - 修复 INDEX HASH JOIN 报 send on closed channel 的问题 #31129
  - 修复使用BatchCommands API 时，少数情况下 TiDB 数据请求无法及时发送到 TiKV 的问题 #32500
  - 修复乐观事务下数据索引可能不一致的问题 #30410
  - 修复窗口函数在使用事务时，计算结果与不使用事务时的计算结果不一致的问题 #29947
  - 修复将 Decimal 转为 String 时长度信息错误的问题 #29417
  - 修复将向量化表达式 tidb_enable_vectorized_expression 设置为 off 时，GREATEST 函数返回结果可能不正确的问题 #29434
  - 修复在某些情况下优化器可能缓存无效 join 计划的问题 #28087
  - 修复向量化表达式设置为 on 时，microsecond 和 hour 函数返回结果可能不正确的问题 #29244 #28643
  - 修复在某些场景下执行 ALTER TABLE.. ADD INDEX 语句时 TiDB panic 的问题 #27687
  - 修复 MPP 节点的可用性检测在某些边界场景中无法工作的问题 #3118
  - 修复分配 MPP task ID 时出现 DATA RACE 的问题 #27952
  - 修复删除空的 dual table 后 MPP 查询出现 index out of range 报错的问题 #28250
  - 修复运行 MPP 查询时出现 invalid cop task execution summaries length 相关日志的问题 #1791
  - 修复 SET GLOBAL tidb_skip_isolation_level_check=1 无法在新会话中生效的问题 #27897
  - 修复 tiup bench 命令运行时间过长导致的 index out of range 问题 #26832

- TiKV
  - 修复 GC worker 繁忙后无法执行范围删除（即执行 `unsafe_destroy_range` 参数）的问题 #11903
  - 修复删除 Peer 可能造成高延迟的问题 #10210
  - 修复 Region 没有数据时 `any_value` 函数结果错误的问题 #11735
  - 修复删除未初始化的副本可能会造成旧副本被重新创建的问题 #10533
  - 修复在已完成重新选举但没有通知被隔离的 Peer 的情况下执行 `Prepare Merge` 会导致元数据损坏的问题 #11526
  - 修复协程的执行速度太快时偶尔出现的死锁问题 #11549
  - 修复分析火焰图时潜在的死锁和内存泄漏的问题 #11108
  - 修复悲观事务中 prewrite 请求重试在极少数情况下影响数据一致性的风险 #11187
  - 修复 `resource-metering.enabled` 配置不生效的问题 #11235
  - 修复 `resolved_ts` 中协程泄漏的问题 #10965
  - 修复在低写入流量下误报 “GC can not work” 错误的问题 #9910
  - 修复 tikv-ctl 无法正确输出 Region 相关信息的问题 #11393
  - 修复某个 TiKV 节点停机会导致 Resolved Timestamp 进度落后的问题 #11351
  - 修复在极端情况下同时进行 Region Merge、ConfChange 和 Snapshot 时，TiKV 会出现 Panic 的问题 #11475
  - 修复逆序扫表时 TiKV 无法正确读到内存锁的问题 #11440
  - 修复 Decimal 除法计算的结果为 0 时符号为负的问题 #29586
  - 修复因统计线程监控数据导致的内存泄漏 #11195
  - 修复在缺失下游数据库时出现 TiCDC Panic 的问题 #11123
  - 修复因 Congest 错误而导致的 TiCDC 频繁增加 scan 重试的问题 #11082
  - 修复 Raft client 中 batch 消息过大的问题 #9714
  - 折叠了 Grafana Dashboard 中与 Storage 相关的不常用的监控指标 #11681
- PD
  - 修复了 region scatter 生成的调度可能导致 peer 数量减少的问题 #4565
  - 修复 Region 统计不受 `flow-round-by-digit` 影响的问题 #4295
  - 修复因 Region syncer 卡住而导致 leader 选举慢的问题 #3936
  - 允许 Evict Leader 调度器调度拥有不健康副本的 Region #4093
  - 修复热点统计中无法剔除冷热点数据的问题 #4390
  - 修复 TiKV 节点缩容后可能导致 Panic 的问题 #4344
  - 修复调度 Operator 因为目标 Store 处于 Down 的状态而无法快速失败的问题 #3353
- TiFlash
  - 修复 `str_to_date()` 函数对微秒前导零的错误解析
  - 修复启用内存限制后 TiFlash 崩溃的问题
  - 修复输入早于 1970-01-01 00:00:01 UTC 时，`unix_timestamp` 行为与 TiDB/MySQL 不一致的问题
  - 修复当主键为 handle 时，扩宽主键列可能导致的数据不一致问题
  - 修复 Decimal 类型比较时可能出现的数据溢出问题和 `Can't compare` 报错
  - 修复非预期的 `3rd arguments of function substringUTF8 must be constants.` 报错
  - 修复在没有 `nsl` 库的平台上 TiFlash 无法启动的问题
  - 修复 Decimal 类型转换时的数据溢出问题
  - 修复在 TiFlash 与 TiDB/TiKV 之间 `castStringAsReal` 行为不一致的问题
  - 修复 TiFlash 重启时偶发的 `EstablishMPPConnection` 错误

- 修复当设置 TiFlash 副本数为 0（即删除数据）后数据无法回收的问题
- 修复在 TiFlash 与 TiDB/TiKV 之间 `CastStringAsDecimal` 行为不一致的问题
- 修复 `where <string>` 查询结果出错的问题
- 修复当 MPP 查询被终止时，TiFlash 偶发的崩溃问题
- 修复非预期的 `Unexpected type of column: Nullable(Nothing)` 报错

- Tools

  - TiCDC

    * 修复 MySQL sink 在禁用 `batch-replace-enable` 参数时生成重复 `replace` SQL 语句的错误 #4501
    * 修复 `cached region` 监控指标为负数的问题 #4300
    * 修复当 `min.insync.replicas` 小于 `replication-factor` 时不能同步的问题 #3994
    * 修复在移除同步任务后潜在的 panic 问题 #3128
    * 修复因 checkpoint 不准确导致的潜在的数据丢失问题 #3545
    * 修复潜在的同步流控死锁问题 #4055
    * 修复 DDL 特殊注释导致的同步停止的问题 #3755
    * 修复 EtcdWorker 可能阻塞 owner 和 processor 的问题 #3750
    * 修复集群升级后 `stopped` 状态的 changefeed 自动恢复的问题 #3473
    * 修复不支持同步默认值的问题 #3793
    * 修复 TiCDC 默认值填充异常导致的数据不一致问题 #3918 #3929
    * 修复当 PD leader 故障后转移到其他节点时 owner 卡住的问题 #3615
    * 修复人为删除 etcd 任务的状态时导致 TiCDC panic 的问题 #2980
    * 修复在某些 RHEL 发行版上因时区问题导致服务无法启动的问题 #3584
    * 修复 MySQL sink 模块出现死锁时告警过于频繁的问题 #2706
    * 修复 Canal 和 Maxwell 协议下 TiCDC 没有自动开启 `enable-old-value` 选项的问题 #3676
    * 修复 Avro sink 模块不支持解析 JSON 类型列的问题 #3624
    * 修复监控 checkpoint lag 出现负值的问题 #3010
    * 修复在容器环境中 OOM 的问题 #1798
    * 修复在多个 TiKV 崩溃或强制重启时可能遇到复制中断的问题 #3288
    * 修复执行 DDL 后的内存泄漏的问题 #3174
    * 修复当发生 ErrGCTTLExceeded 错误时，changefeed 不快速失败的问题 #3111
    * 修复当上游 TiDB 实例意外退出时，TiCDC 同步任务推进可能停滞的问题 #3061
    * 修复当 TiKV 向同一 Region 发送重复请求时，TiCDC 进程 Panic 的问题 #2386
    * 将参数 `max-message-bytes` 的默认值设置为 `10M`，减少 Kafka 发送过大消息的概率 #3081
    * 修复当写入 Kafka 消息发生错误时，TiCDC 同步任务推进可能停滞的问题 #2978

  - Backup & Restore (BR)

    * 修复当恢复完成后，Region 有可能分布不均的问题 #30425 #31034

  - TiDB Binlog

    修复 `strict-format` 为 `true` 且 CSV 文件大小为 256 MB 时，CSV 文件导入报 `InvalidRange` 的问题 #27763

  - TiDB Lightning

    * 修复 S3 存储路径不存在时 TiDB Lightning 不报错的问题 #28031 #30709

16.16.3 TiDB 5.1.3 Release Note

发版日期：2021 年 12 月 3 日

TiDB 版本：5.1.3

16.16.3.1 Bug 修复

- TiKV
  - 修复 GcKeys 任务被多个键调用时无法正常进行，导致 Compaction Filter GC 可能不删除 MVCC deletion 信息的问题 #11217

16.16.4 TiDB 5.1.2 Release Notes

发版日期：2021 年 9 月 27 日

TiDB 版本：5.1.2

16.16.4.1 兼容性更改

- TiDB
  - 以下 Bug 修复涉及执行结果变化，可能引起兼容性变化：
    * 修复了 `greatest(datetime)union null` 返回空字符串的问题 #26532
    * 修复了 `having` 子句可能执行错误的问题 #26496
    * 修复了当 `between` 表达式两边的 collation 不一致会导致查询结果错误的问题 #27146
    * 修复了当 `group_concat` 函数包含非 `bin` 的 collation 时查询结果错误的问题 #27429
    * 修复了当开启 New Collation 时，在多列上执行 `count(distinct)` 表达式结果错误的问题 #27091
    * 修复了 `extract` 函数的参数是负数时查询结果错误的问题 #27236
    * 修复了当 `SQL_MODE` 为 'STRICT_TRANS_TABLES' 时，插入非法时间不报错的问题 #26762
    * 修复了当 `SQL_MODE` 为 'NO_ZERO_IN_DATE' 时，使用非法的默认时间不报错的问题 #26766
- Tools
  - TiCDC
    * 将兼容版本从 `5.1.0-alpha` 改为 `5.2.0-alpha` #2659

16.16.4.2 改进提升

- TiDB
  - 根据直方图行数来触 auto-analyze，提升 auto-analyze 触发的准确度 #24237
- TiKV
  - 支持动态更改 TiCDC 配置项 #10645

  - 减少 Resolved TS 消息的大小以节省网络带宽 #2448
  - 支持限制每个 Store 发送的心跳包信息中 peer stats 的数量 #10621

- PD

  - 存在 scatter range 调度器情况下，允许空 Region 被调度，并可以在该调度器中使用单独的 `tolerance` 配置项 #4117
  - 提升 PD 之间同步 Region 信息的性能 #3933
  - 支持根据 Operator 的生成情况动态调整 Store 的重试次数 #3744

- TiFlash

  - 支持 `DATE()` 函数
  - 为 Grafana 面板增加每个实例的写入吞吐
  - 优化 `leader-read` 流程的性能
  - 加速 MPP 任务取消的流程

- Tools

  - TiCDC

    * 当统一分类器使用内存进行分类时，优化内存管理 #2553
    * 当并发性高时，优化 workerpool 以减少 goroutines 的数量 #2488
    * 当一个表的区域从一个 TiKV 节点转移出去时，减少 goroutine 的使用 #2284
    * 添加一个全局 gRPC 连接池并在 KV 客户端之间共享 gRPC 连接 #2534
    * 禁止跨主要和次要版本操作 TiCDC 集群 #2599

  - Dumpling

    * 支持备份一些无法执行 `START TRANSACTION ... WITH CONSISTENT SNAPSHOT` 和 `SHOW CREATE TABLE` 语句的 MySQL 兼容数据库 #309

16.16.4.3 Bug 修复

- TiDB

  - 修复 hash 列为 `ENUM` 类型时 index hash join 的结果可能出错的问题 #27893
  - 修复极少数情况下 batch client 复用空闲连接可能阻塞请求发送的问题 #27678
  - 修复了 `FLOAT64` 类型的溢出检查与 MySQL 不一致的问题 #23897
  - 修复 TiDB 把 `pd is timeout` 错误返回成 unkonwn 的问题 #26147
  - 修复了 `case when` 表达式的字符集和排序规则错误的问题 #26662
  - 修复 MPP 查询可能返回 `can not found column in Schema column` 错误的问题 #28148
  - 修复 TiFlash 宕机可能导致 TiDB Panic 的问题 #28096
  - 修复使用谓词 `enum like 'x%'` 可能导致错误结果的问题 #27130
  - 修复当使用 IndexLookupJoin 时公共表达式 (CTE) 死锁的问题 #27410
  - 修复死锁重试被错误记录到 `INFORMATION_SCHEMA.DEADLOCKS` 表的问题 #27400
  - 修复分区表上的 `TABLESAMPLE` 查询结果排序不生效的问题 #27349
  - 移除未使用的 `/debug/sub-optimal-plan` HTTP 接口相关逻辑 #27265
  - 修复 hash 分区表处理无符号数据时查询返回错误结果的问题 #26569

  - 修复当 NO_UNSIGNED_SUBTRACTION 被设置时创建分区出错的问题 #26765
  - 修复 Apply 转换为 Join 时缺失 distinct 的问题 #26958
  - 为处于恢复状态的 TiFlash 节点设置一段屏蔽时间，避免在此时间内阻塞查询 #26897
  - 修复 CTE 被引用多次时可能出现的 bug 的问题 #26212
  - 修复使用 MergeJoin 时可能造成 CTE 出现 bug 的问题 #25474
  - 修复当 Join 分区表和普通表时 select for update 不能正确上锁的问题 #26251
  - 修复当 Join 分区表和普通表时 select for update 语句结果报错的问题 #26250
  - 修复 PointGet 不能使用轻量清锁的问题 #26562

- TiKV

  - 修复当 TiKV 从 v3.x 升级至较高版本后出现 Panic 的问题 #10902
  - 修复损坏的快照文件可能会造成磁盘空间无法回收的问题 #10813
  - 当 slogger 线程过载且队列已满时，删除日志而不是阻塞线程 #10841
  - 使 TiKV Coprocessor 慢日志只考虑处理请求所花费的时间 #10841
  - 修复当处理 Coprocessor 请求时因超时而导致 Panic 的问题 #10852
  - 修复 TiKV 在启用 Titan 并从 v5.0 以前的版本升级时出现 Panic 的问题 #10842
  - 修复高版本的 TiKV 无法回滚到 v5.0.x 的问题 #10842
  - 修复 TiKV 可能会在 RocksDB 读取文件之前删除文件的错误 #10438
  - 修复遗留的悲观锁导致的解析失败的问题 #26404

- PD

  - 修复 PD 未能及时修复 Down Peer 副本的问题 #4077
  - 修复 replication.max-replicas 更新后默认的 Placement Rules 副本数量不变的问题 #3886
  - 修复 PD 在扩容 TiKV 时可能会 Panic 的问题 #3868
  - 修复当集群中存在 evict leader 调度器时，PD 热点调度无法工作的问题 #3697

- TiFlash

  - 修复无法建立 MPP 连接时出现异常结果的问题
  - 修复多盘部署时数据不一致的潜在问题
  - 修复高负载时出现 MPP 查询结果错误的问题
  - 修复 MPP 查询卡死的潜在问题
  - 修复并发执行节点初始化与 DDL 操作时出现异常的问题
  - 修复当查询过滤条件包含诸如 CONSTANT、<、<=、>、>= 或 COLUMN 时出现错误结果的问题
  - 修复并发执行多个 DDL 操作和 Apply Snapshot 操作时出现异常的潜在问题
  - 修复写入压力大时 metrics 中 store size 不准确的问题
  - 修复 TiFlash 长时间运行后无法回收 Delta 历史数据的潜在问题
  - 修复开启 New Collation 时查询结果错误的问题
  - 修复解锁出现异常的潜在问题
  - 修复 metrics 显示错误值的问题

- Tools

  - Backup & Restore (BR)

    * 修复了备份数据和恢复数据时显示的平均速度数值不准确的问题 #1405

- Dumpling
  * 修复特定 MySQL 版本（8.0.3，8.0.23）下，`show table status` 返回结果不正确导致 dump 阶段卡死的问题 #333
  * 修复 CLI 在默认 `sort-engine` 选项上与 4.0.x 集群的兼容性问题 #2373
- TiCDC
  * 修复未充分考虑字符串类型的值可能是 `string` 或 `[]byte` 时，转换为 JSON 格式可能导致 TiCDC 进程崩溃的问题 #2758
  * 降低 gRPC 的 window size 以避免出现 OOM 的情况 #2673
  * 修复内存压力较高时 gRPC 的 `keepalive` 错误 #2202
  * 修复 `unsigned tinyint` 导致 TiCDC 崩溃的问题 #2648
  * 修复 TiCDC Open Protocol 下输出空值的问题。修复后，在开放协议下，未包含变更的事务 TiCDC 处理时不再输出空值 #2612
  * 修复手动重启 TiCDC 时 DDL 处理存在的问题 #2603
  * 修复操作元数据时，`EtcdWorker` 的快照隔离可能被破坏的问题 #2559
  * 修复 TiCDC 在重新调度 table 时多个 processors 可能向同一个 table 写数据的问题 #2230
  * 修复 TiCDC 遇到 `ErrSchemaStorageTableMiss` 错误时可能导致 changefeed 被意外重置的问题 #2422
  * 修复 TiCDC 遇到 `ErrGCTTLExceeded` 错误时 changefeed 不能被移除的问题 #2391
  * 修复 TiCDC 同步大表到 cdclog 失败的问题 #1259#2424

16.16.5 TiDB 5.1.1 Release Notes

发版日期：2021 年 7 月 30 日

TiDB 版本：5.1.1

16.16.5.1 兼容性更改

- TiDB
  - 对于从 v4.0 升级至 v5.1 的集群，`tidb_multi_statement_mode` 的默认值为 `OFF`。建议使用客户端库的多语句功能，参考`tidb_multi_statement_mode` 文档 #25751
  - 将系统变量 `tidb_stmt_summary_max_stmt_count` 的默认值从 200 修改为 3000 #25874
  - 访问 `table_storage_stats` 表需要 `SUPER` 权限 #26352
  - 访问 `information_schema.user_privileges` 表需要 `mysql.user` 上的 `SELECT` 权限来显示其他人的权限 #26311
  - 访问 `information_schema.cluster_hardware` 需要 `CONFIG` 权限 #26297
  - 访问 `information_schema.cluster_info` 表需要 `PROCESS` 权限 #26297
  - 访问 `information_schema.cluster_load` 表需要 `PROCESS` 权限 #26297
  - 访问 `information_schema.cluster_systeminfo` 表需要 `PROCESS` 权限 #26297
  - 访问 `information_schema.cluster_log` 表需要 `PROCESS` 权限 #26297
  - 访问 `information_schema.cluster_config` 表需要 `CONFIG` 权限 #26150

16.16.5.2 功能增强

- TiDB Dashboard
  - 新增 OIDC SSO 支持。通过设置兼容 OIDC 标准的 SSO 服务（例如 Okta、Auth0 等），用户可以在不输入 SQL 密码的情况下登录 TiDB Dashboard #3883
- TiFlash
  - 支持 DAG 请求中的 HAVING() 函数

16.16.5.3 改进提升

- TiDB
  - Stale Read 成为正式功能 (GA)
  - 避免对 paramMarker 的分配以加快数据插入速度 #26076
  - 支持稳定结果模式，使查询结果更稳定 #25995
  - 支持将函数 json_unquote() 下推到 TiKV #26265
  - 支持 MPP 查询的重试 #26480
  - 对于 point get 或 batch point get 算子，在唯一索引写入过程中，将悲观锁 LOCK 记录转化为 PUT 记录 #26225
  - 禁止使用 Stale 查询来进行创建视图 #26225
  - 在 MPP 模式下彻底下推 COUNT(DISTINCT) 聚合函数 #26194
  - 在发起 MPP 查询之前检查 TiFlash 的可用性 #26192
  - 不允许将读时间戳设置为将来的时间 #25763
  - 当聚合函数在 EXPLAIN 语句中不能被下推时打印警告日志 #25737
  - 增加 statements_summary_evicted 表来记录集群的驱逐数量信息 #25587
  - 提升内置函数 str_to_date 在格式指定器中 %b/%M/%r/%T 的 MySQL 兼容性 #25768
- TiKV
  - 提升 prewrite 请求的幂等性以减少不确定性错误的概率 #10586
  - 预防处理多个过期命令时出现栈溢出的风险 #10502
  - 不使用 Stale Read 请求的 start_ts 更新 max_ts 以避免过多不必要的 commit 请求重试 #10451
  - 分离处理读写的 ready 状态以减少读延迟 #10592
  - 降低 I/O 限流器开启后对数据导入速度的影响 #10390
  - 提升 Raft gRPC 连接的负载均衡 #10495
- Tools
  - TiCDC
    - 移除 file sorter 文件排序器 #2327
    - 优化连接 PD 时缺少证书情况下的报错提示 #1973
  - TiDB Lightning
    - 为恢复 schema 添加重试机制 #1294
  - Dumpling
    - 上游是 TiDB v3.x 集群时，使用 _tidb_rowid 来切分表以减少 TiDB 的内存使用 #295
    - 减少访问数据库元信息的频率以提升性能和稳定性 #315

16.16.5.4 Bug 修复

- TiDB
  - 修复了 `tidb_enable_amend_pessimistic_txn=on` 下更改列类型可能出现数据丢失的问题 #26203
  - 修复了 `last_day` 函数的行为在 SQL 模式下不兼容的问题 #26001
  - 修复 `LIMIT` 位于窗口函数之上时可能出现的 panic 问题 #25344
  - 修复了提交悲观事务可能会导致写冲突的问题 #25964
  - 修复关联子查询中 Index Join 的结果不正确问题 #25799
  - 修复了成功提交的悲观事务可能会报提交失败的问题 #10468
  - 修复在 SET 类型列上 Merge Join 结果不正确的问题 #25669
  - 修复了在悲观事务中索引键值可能会被重复提交的问题 #26359
  - 修复了优化器在定位分区时存在整数溢出的风险 #26227
  - 修复了将 DATE 类型转换成时间戳时可能会写入无效值的问题 #26292
  - 修复了 Coprocessor Cache 监控项未在 Grafana 中显示的问题 #26338
  - 修复了遥测引起的干扰日志 #25760 #25785
  - 修复了索引前缀的查询范围问题 #26029
  - 修复了并发 truncate 同一个 partition 会导致 DDL 执行卡住的问题 #26229
  - 修复了 EMUN 元素重复的问题 #25955
  - 修复了 CTE 迭代器没有正确关闭的问题 #26112
  - 修复 `LOAD DATA` 语句可能不正常导入非 utf8 数据的问题 #25979
  - 修复在无符号整数列上使用窗口函数可能导致崩溃的问题 #25956
  - 修复了清除 Async Commit 锁时可能会导致 TiDB panic 的问题 #25778
  - 修复了 Stale Read 不完全兼容 PREPARE 语句的问题 #25800
  - 修复 ODBC 类常数（例如 `{d '2020-01-01'}`）不能被用作表达式的问题 #25531
  - 修复了单独运行 TiDB 时出现的一个错误 #25555
- TiKV
  - 修复特定平台上的 duration 计算可能崩溃的问题 #10569
  - 修复 Load Base Split 误用 `batch_get_command` 中未编码键的问题 #10542
  - 修复在线变更 `resolved-ts.advance-ts-interval` 配置无法立即生效的问题 #10426
  - 修复在超过 4 副本的罕见场景下 Follower 元信息损坏的问题 #10225
  - 修复开启加密后再次生成同样的 snapshot 会出现 panic 的问题 #9786 #10407
  - 修正 `tikv_raftstore_hibernated_peer_state` 监控指标项 #10330
  - 修复 coprocessor 中 `json_unquote()` 函数错误的参数类型 #10176
  - 修复悲观事务中索引键被重复 commit 的问题 #10468
  - 修复 `ReadIndex` 请求在 leader 迁移后返回过期数据的问题 #9351
- PD
  - 修复多个调度器产生调度冲突时无法生产预期调度的问题 #3807 #3778
  - 修复当调度器被删除后，可能会再度运行的问题 #2572
- TiFlash
  - 修复执行扫表任务时出现进程崩溃的潜在问题

  - 修复处理 DAG 请求时出现 `duplicated region` 报错的问题
  - 修复读负载高的情况下进程崩溃的问题
  - 修复执行 `DateFormat` 函数时出现进程崩溃的潜在问题
  - 修复执行 MPP 任务时出现内存泄漏的潜在问题
  - 修复执行 `COUNT` 或 `COUNT DISTINCT` 函数时出现非预期结果的问题
  - 修复多盘部署时出现数据无法恢复的潜在问题
  - 修复 TiDB Dashboard 无法正确显示 TiFlash 磁盘信息的问题
  - 修复析构 `SharedQueryBlockInputStream` 时出现进程崩溃的潜在问题
  - 修复析构 `MPPTask` 时出现进程崩溃的潜在问题
  - 修复通过快照同步数据后可能出现的数据不一致的问题

- Tools

  - TiCDC

    * 修复对 New Collation 的支持 #2301
    * 修复了运行时因非同步访问共享 map 可能导致 panic 的问题 #2300
    * 修复了 DDL 语句执行时 owner 崩溃可能导致的 DDL event 遗漏的问题 #2290
    * 修复了试图过早在 TiDB 中解锁的问题 #2188
    * 修复了表迁移后节点崩溃可能导致数据丢失的问题 #2033
    * 修复了 `changefeed update` 对 `--sort-dir` and `--start-ts` 的处理逻辑 #1921

  - Backup & Restore (BR)

    * 修复了错误计算待恢复数据的大小的问题 #1270
    * 修复了从 cdclog 恢复数据时会遗漏 DDL event 的问题 #870

  - TiDB Lightning

    * 修复 TiDB Lightning 解析 Parquet 文件中 `DECIMAL` 类型数据失败的问题 #1275
    * 修复了计算 key 区间时出现整数型溢出的问题 #1291 #1290

16.16.6 TiDB 5.1 Release Notes

发版日期：2021 年 6 月 24 日

TiDB 版本：5.1

在 5.1 版本中，你可以获得以下关键特性：

- 支持 MySQL 8 中的公共表表达式 (Common Table Expression)，提高了 SQL 语句的可读性与执行效率。
- 支持对数据表列类型的在线变更，提高了业务开发的灵活性。
- 引入一种新的统计信息类型，默认作为实验特性启用，提升查询稳定性。
- 支持 MySQL 8 中的动态权限 (Dynamic Privileges) 配置，实现对某些操作更细粒度的控制。
- 支持通过 Stale Read 功能直接读取本地副本数据，降低读取延迟，提升查询性能（实验特性）。
- 新增锁视图 (Lock View) 功能方便 DBA 观察事务加锁情况以及排查死锁问题（实验特性）。
- 新增 TiKV 后台任务写入限制 (TiKV Write Rate Limiter)，保证读写请求的延迟稳定性。

16.16.6.1 兼容性更改

> **注意：**
>
> 当从一个早期的 TiDB 版本升级到 TiDB 5.1 时，如需了解所有中间版本对应的兼容性更改说明，请查看对应版本的Release Note。

16.16.6.1.1 系统变量

变量名	修改类型	描述
cte_max_recursion_depth	新增	用于控制公共表表达式最大递归深度。
init_connect	新增	用于控制初始连接。
tidb_analyze_version	新增	用于控制所收集到的统计信息。默认值为 2，默认作为实验特性启用。
tidb_enable_enhanced_security	新增	表示所连接的 TiDB 服务器是否启用了安全增强模式（SEM），在不重新启动 TiDB 服务器的情况下不能改变该变量。
tidb_enforce_mpp	新增	用于忽略优化器代价估算，强制使用 MPP 模式。BOOL 类型，默认值为 `false`。
tidb_partition_prune_mode	新增	用于设置是否开启分区表动态裁剪模式（实验特性）。默认值为 `static`，即默认不启用分区表动态裁剪模式。

16.16.6.1.2 配置文件参数

配置文件	配置项	修改类型	描述
TiDB 配置文件	security.enable-sem	新增	控制是否启用安全增强模式 (SEM)。默认值为 false，代表未启用。
TiDB 配置文件	performance.committer-concurrency	修改	在单个事务的提交阶段，控制用于执行提交操作相关请求的并发数。默认值从 16 修改为 128。
TiDB 配置文件	performance.tcp-no-delay	新增	控制 TiDB 是否在 TCP 层开启 TCP_NODELAY。默认值为 true，代表开启。
TiDB 配置文件	performance.enforce-mpp	新增	用于在实例级别控制 TiDB 是否忽略优化器代价估算，强制使用 MPP 模式，默认值为 false。该配置项可以控制系统变量tidb_enforce_mpp 的初始值。
TiDB 配置文件	pessimistic-txn.deadlock-history-capacity	新增	控制单个 TiDB 节点的INFORMATION_SCHEMA.DEADLOCKS 表最多可记录的死锁事件个数，默认值为“10”。

配置文件	配置项	修改类型	描述
TiKV 配置文件	abort-on-panic	新增	设置 TiKV panic 时 abort 进程是否允许系统生成 core dump 文件。默认值为 false，代表不允许生成 core dump 文件。
TiKV 配置文件	hibernate-regions	修改	默认值从 false 修改为 true。如果 Region 长时间处于非活跃状态，即被自动设置为静默状态。
TiKV 配置文件	old-value-cache-memory-quota	新增	设置缓存在内存中的 TiCDC Old Value 的条目占用内存的上限。默认值为 512MB。
TiKV 配置文件	sink-memory-quota	新增	设置缓存在内存中的 TiCDC 数据变更事件占用内存的上限。默认值为 512MB。
TiKV 配置文件	incremental-scan-threads	新增	控制增量扫描历史数据任务的线程个数。默认值为 4，代表 4 个线程。
TiKV 配置文件	incremental-scan-concurrency	新增	控制增量扫描历史数据任务的最大并发执行个数。默认值为 6，代表最多并发执行 6 个任务。

配置文件	配置项	修改类型	描述
TiKV 配置文件	soft-pending-compaction-bytes-limit	修改	pending compaction bytes 的软限制，默认值从“64GB”修改为“192GB”。
TiKV 配置文件	storage.io-rate-limit	新增	控制 TiKV 写入的 IO 速率。storage.io-rate-limit.max-bytes-per-sec 默认值为“0MB”。
TiKV 配置文件	resolved-ts.enable	新增	为所有 Region leader 维护 resolved-ts，默认值为 true。
TiKV 配置文件	resolved-ts.advance-ts-interval	新增	推进 resolved-ts 的间隔，默认为“1s”，支持动态更改。
TiKV 配置文件	resolved-ts.scan-lock-pool-size	新增	用于初始化 resolved-ts 时扫锁的线程数，默认值为 2。

16.16.6.1.3 其他

- 升级前，请检查 TiDB 配置项 feedback-probability 的值。如果不为 0，升级后会触发“panic in the recoverable goroutine”报错，但不影响升级。
- 为了提升 TiDB 性能，TiDB 的 Go 编译器版本从 go1.13.7 升级到了 go1.16.4。如果你是 TiDB 的开发者，为了能保证顺利编译，请对应升级你的 Go 编译器版本。
- 请避免在对使用 TiDB Binlog 的集群进行滚动升级的过程中新创建聚簇索引表。
- 请避免在 TiDB 滚动升级时执行 alter table ... modify column 或 alter table ... change column。
- 当按表构建 TiFlash 副本时，v5.1 版本及后续版本将不再支持设置系统表的 replica。在集群升级前，需要清除相关系统表的 replica，否则会导致升级失败。
- 在 TiCDC 的 cdc cli changefeed 命令中废弃 --sort-dir 参数，用户可在 cdc server 命令中设定 --sort-dir。#1795
- 升级到 TiDB 5.1 之后，如果遇到“function READ ONLY has only noop implementation”错误，可以将系统变量tidb_enable_noop_functions 的值设置为 ON 以忽略此报错。因为 MySQL 的‘read_only’变量在 TiDB 中

尚不生效（属于 ‘noop’ 行为），即使在 TiDB 中设置了此变量，集群仍然是可写的。

16.16.6.2 新功能

16.16.6.2.1 SQL

- 新增 MySQL 8 中的公共表表达式（Common Table Expression，简称 CTE）。

  CTE 为 TiDB 带来递归或非递归查询层次结构数据的能力，满足了人力资源、制造业、金融市场和教育在内的多种应用领域需要使用树形查询实现业务逻辑的需求。

  在 TiDB 中，你可以通过 WITH 语句使用公共表表达式。用户文档，#17472

- 新增 MySQL 8 中的动态权限 (Dynamic Privileges)。

  动态权限用于限制 SUPER 权限，为 TiDB 提供更灵活的权限配置，实现对某些操作更细粒度的控制。例如，你可以使用动态权限来创建一个只能执行 BACKUP 和 RESTORE 操作的用户帐户。

  支持的动态权限包括：

  - BACKUP_ADMIN
  - RESTORE_ADMIN
  - ROLE_ADMIN
  - CONNECTION_ADMIN
  - SYSTEM_VARIABLES_ADMIN

  你也可以使用插件来添加新的权限。若要查看全部的动态权限，请执行 SHOW PRIVILEGES 语句。用户文档

- 新增安全增强模式 (Security Enhanced Mode) 配置项，用于对 TiDB 管理员进行更细粒度的权限划分。

  安全增强模式默认关闭，如需开启，请参考用户文档。

- 全面加强列类型的在线变更能力，支持通过 ALTER TABLE 语句进行列的在线类型修改，包括但不限于：

  - 从 VARCHAR 转换为 BIGINT
  - DECIMAL 精度修改
  - 从 VARCHAR(10) 到 VARCHAR(5) 的长度压缩

  用户文档

- 引入新的语法 AS OF TIMESTAMP，支持通过 Stale Read 功能从指定的时间点或时间范围内读取历史数据（实验特性）。

  用户文档，#21094

  AS OF TIMESTAMP 语法示例如下：

```
SELECT * FROM t AS OF TIMESTAMP  '2020-09-06 00:00:00';
START TRANSACTION READ ONLY AS OF TIMESTAMP '2020-09-06 00:00:00';
SET TRANSACTION READ ONLY as of timestamp '2020-09-06 00:00:00';
```

- 引入一种新的统计信息类型 `tidb_analyze_version = 2`（实验特性）。

  `tidb_analyze_version = 2` 默认启用，避免了 Version 1 中因为哈希冲突导致的在较大的数据量中可能产生的较大误差，并保持了大多数场景中的估算精度。

  用户文档

16.16.6.2.2 事务

- 新增锁视图 (Lock View)（实验特性）

  Lock View 用于提供关于悲观锁的锁冲突和锁等待的更多信息，方便 DBA 通过锁视图功能来观察事务加锁情况以及排查死锁问题等 #24199

  用户文档：

  - 查看集群中所有 TiKV 节点上当前正在发生的悲观锁等锁：DATA_LOCK_WAITS
  - 查看 TiDB 节点上最近发生的若干次死锁错误：DEADLOCKS
  - 查看 TiDB 节点上正在执行的事务的信息：TIDB_TRX

16.16.6.2.3 性能

- 数据副本非一致性读 (Stale Read)（实验特性）

  直接读取本地副本数据，降低读取延迟，提升查询性能

  用户文档，#21094

- 默认开启 Hibernate Region 特性。

  如果 Region 长时间处于非活跃状态，即被自动设置为静默状态，可以降低 Leader 和 Follower 之间心跳信息的系统开销。

  用户文档，#10266

16.16.6.2.4 稳定性

- TiCDC 复制稳定性问题解决
  - 改善 TiCDC 内存使用，避免在以下场景出现 OOM
    * 同步中断期间积累大量数据，超过 1TB，重新同步出现 OOM 问题
    * 大量数据写入造成 TiCDC 出现 OOM 问题
  - 改善 TiCDC 同步中断问题，缓解以下场景的问题 project#11
    * 网络不稳定情况下出现的同步中断问题
    * 在部分 TiKV/PD/TiCDC 节点宕机情况下出现的同步中断问题
- TiFlash 存储内存控制

  优化了 Region 快照生成的速度和内存使用量，减少了 OOM 的可能性

- 新增 TiKV 后台任务写入限制 (TiKV Write Rate Limiter)

  TiKV Write Rate Limiter 通过平滑 TiKV 后台任务如 GC，Compaction 等的写入流量，保证读写请求的延迟稳定性。TiKV 后台任务写入限制默认值为 “0MB”，建议将此限制设置为磁盘的最佳 I/O 带宽，例如云盘厂商指定的最大 I/O 带宽。

  用户文档，#9156

- 解决多个扩缩容时的调度稳定性问题

16.16.6.2.5 遥测

TiDB 在遥测中新增收集集群请求的运行状态，包括执行情况、失败情况等。

若要了解所收集的信息详情及如何禁用该行为，请参见遥测文档。

16.16.6.3 提升改进

- TiDB
  - 支持 `VITESS_HASH()` 函数 #23915
  - 支持枚举类型下推到 TiKV，提升 WHERE 子句中使用枚举类型时的性能 #23619
  - 支持 `RENAME USER` 语法 #23648
  - 优化 Window Function 计算过程，解决了使用 ROW_NUMBER() 对数据分页时 TiDB OOM 的问题 #23807
  - 优化 UNION ALL 的计算过程，解决了使用 UNION ALL 连接大量 SELECT 语句时 TiDB OOM 的问题 #21441
  - 优化分区表动态裁剪模式，提升其性能和稳定性 #24150
  - 解决多种情况下出现的 `Region is Unavailable` 问题 project#62
    - 修复频繁调度情况下可能出现的多个 `Region is Unavailable` 问题
    - 解决部分高压力写入情况下可能出现的 `Region is Unavailable` 问题
  - 当内存中的统计信息缓存是最新的时，避免后台作业频繁读取 `mysql.stats_histograms` 表造成高 CPU 使用率 #24317
- TiKV
  - 使用 `zstd` 压缩 Region Snapshot，防止大量调度或扩缩容情况下出现各节点之间空间差异比较大的问题 #10005
  - 解决多种情况下的 OOM 问题 #10183
    - 增加各模块内存使用情况追踪
    - 解决 Raft entries cache 过大导致的 OOM 问题
    - 解决 GC tasks 堆积导致的 OOM 问题
    - 解决一次性从 Raft log 取太多 Raft entries 到内存导致 OOM 问题
  - 让 Region 分裂更均匀，缓解有写入热点时 Region 大小的增长速度超过分裂速度的问题 #9785
- TiFlash
  - 新增对 `Union All`、`TopN`、`Limit` 函数的支持

  - 新增 MPP 模式下对笛卡尔积 left outer join 和 semi anti join 的支持
  - 优化锁操作以避免 DDL 语句和读数据相互阻塞
  - 优化 TiFlash 对过期数据的清理
  - 新增支持对 `timestamp` 列的查询过滤条件在 TiFlash 存储层进一步过滤
  - 在集群中有大量表时，优化 TiFlash 的启动速度及扩容速度
  - 提升 TiFlash 在未知 CPU 上运行的兼容性

- PD

  - 避免在添加 `scatter region` 调度器后出现的非预期统计行为 #3602
  - 解决扩缩容过程中出现的多个调度问题
    * 优化副本 snapshot 生成流程，解决扩缩容调度慢问题：#3563 #10059 #10001
    * 解决由于流量变化引带来的心跳压力引起的调度慢问题 #3693 #3739 #3728 #3751
    * 减少大集群由于调度产生的空间差异问题，并优化调度公式防止由于压缩率差异大引发的类似异构空间集群的爆盘问题 #3592 #10005

- Tools

  - Backup & Restore (BR)
    * 支持备份和恢复 `mysql` schema 下的用户数据表 #1143 #1078
    * BR 支持 S3 兼容的存储（基于 virtual-host 寻址模式）#10243
    * BR 改进 backupmeta 格式，减少内存占用 #1171
  - TiCDC
    * 改进了部分日志信息的描述使其更加明确清晰，对诊断问题更有帮助 #1759
    * 为 TiCDC 扫描的速度添加感知下游处理能力的 (back pressure) 功能 #10151
    * 减少 TiCDC 进行初次扫描的内存使用量 #10133
    * 提升了悲观事务中 TiCDC Old Value 的缓存命中率 #10089
  - Dumpling
    * 改善从 TiDB v4.0 导出数据的逻辑避免 TiDB OOM #273
    * 修复备份失败却没有错误输出的问题 #280
  - TiDB Lightning
    * 提升导入速度。优化结果显示，导入 TPC-C 数据速度提升在 30% 左右，导入索引比较多（5 个索引）的大表 (2TB+) 速度提升超过 50% #753
    * 导入前对导入数据和目标集群进行检查，如果不符合导入要求，则报错拒绝导入程序的运行 #999
    * 优化 Local 后端更新 checkpoint 的时机，提升断点重启时的性能 #1080

16.16.6.4 Bug 修复

- TiDB

  - 修复投影消除在投影结果为空时执行结果可能错误的问题 #23887
  - 修复列包含 NULL 值时查询结果在某些情况下可能错误的问题 #23891
  - 当有虚拟列参与扫描时不允许生成 MPP 计划 #23886

- 修复 Plan Cache 中对 PointGet 和 TableDual 错误的重复使用 #23187 #23144 #23304 #23290
- 修复优化器在为聚簇索引构建 IndexMerge 执行计划时出现的错误 #23906
- 修复 BIT 类型相关错误的类型推导 #23832
- 修复某些优化器 Hint 在 PointGet 算子存在时无法生效的问题 #23570
- 修复 DDL 遇到错误回滚时可能失败的问题 #23893
- 修复二进制字面值常量的索引范围构造错误的问题 #23672
- 修复某些情况下 IN 语句的执行结果可能错误的问题 #23889
- 修复某些字符串函数的返回结果错误的问题 #23759
- 执行 REPLACE 语句需要用户同时拥有 INSERT 和 DELETE 权限 #23909
- 修复点查时出现的性能回退 #24070
- 修复因错误比较二进制与字节而导致的 TableDual 计划错误的问题 #23846
- 修复了在某些情况下，使用前缀索引和 Index Join 导致的 panic 的问题 #24547 #24716 #24717
- 修复了 point get 的 prepare plan cache 被事务中的 point get 语句不正确使用的问题 #24741
- 修复了当排序规则为 ascii_bin 或 latin1_bin 时，写入错误的前缀索引值的问题 #24569
- 修复了正在执行的事务被 GC worker 中断的问题 #24591
- 修复了当 new-collation 开启且 new-row-format 关闭的情况下，点查在聚簇索引下可能出错的问题 #24541
- 为 Shuffle Hash Join 重构分区键的转换功能 #24490
- 修复了当查询包含 HAVING 子句时，在构建计划的过程中 panic 的问题 #24045
- 修复了列裁剪优化导致 Apply 算子和 Join 算子执行结果错误的问题 #23887
- 修复了从 Async Commit 回退的主锁无法被清除的问题 #24384
- 修复了一个统计信息 GC 的问题，该问题可能导致重复的 fm-sketch 记录 #24357
- 当悲观锁事务收到 ErrKeyExists 错误时，避免不必要的悲观事务回滚 #23799
- 修复了当 sql_mode 包含 ANSI_QUOTES 时，数值字面值无法被识别的问题 #25015
- 禁止如 INSERT INTO table PARTITION (<partitions>)... ON DUPLICATE KEY UPDATE 的语句从 non-listed partitions 读取数据 #24746
- 修复了当 SQL 语句包含 GROUP BY 以及 UNION 时，可能会出现的 index out of range 的问题 #24281
- 修复了 CONCAT 函数错误处理排序规则的问题 #24296
- 修复了全局变量 collation_server 对新会话无法生效的问题 #24156

- TiKV

  - 修复了 Coprocessor 未正确处理 IN 表达式有符号整数或无符号整数类型数据的问题 #9821
  - 修复了在批量 ingest SST 文件后产生大量空 Region 的问题 #964
  - 修复了 file dictionary 文件损坏之后 TiKV 无法启动的问题 #9886
  - 修复了由于读取旧值而导致的 TiCDC OOM 问题 #9996 #9981
  - 修复了聚簇主键列在次级索引上的 latin1_bin 字符集出现空值的问题 #24548
  - 新增 abort-on-panic 配置，允许 TiKV 在 panic 时生成 core dump 文件。用户仍需正确配置环境以开启 core dump。 #10216
  - 修复了 TiKV 不繁忙时 point get 查询性能回退的问题 #10046

- PD

  - 修复在 store 数量多的情况下，切换 PD Leader 慢的问题 #3697
  - 修复删除不存在的 evict leader 调度器时出现 panic 的问题 #3660
  - 修复 offline peer 在合并完后未更新统计的问题 #3611

- TiFlash
  - 修复 `TIME` 类型转换为 `INT` 类型时产生错误结果的问题
  - 修复 `receiver` 可能无法在 10 秒内找到对应任务的问题
  - 修复 `cancelMPPQuery` 中可能存在无效迭代器的问题
  - 修复 `bitwise` 算子和 TiDB 行为不一致的问题
  - 修复当使用 `prefix key` 时出现范围重叠报错的问题
  - 修复字符串转换为 `INT` 时产生错误结果的问题
  - 修复连续快速写入可能导致 TiFlash 内存溢出的问题
  - 修复 Table GC 时会引发空指针的问题
  - 修复向已被删除的表写数据时 TiFlash 进程崩溃的问题
  - 修复当使用 BR 恢复数据时 TiFlash 进程可能崩溃的问题
  - 修复并发复制共享 Delta 索引导致结果错误的问题
  - 修复 TiFlash 在 Compaction Filter 特性开启时可能崩溃的问题
  - 修复了从 Async Commit 回退的锁无法被 TiFlash 清除的问题
  - 修复当 `TIMEZONE` 类型的转换结果包含 `TIMESTAMP` 类型时返回错误结果的问题
  - 修复 TiFlash 在 Segment Split 期间异常退出的问题
- Tools
  - TiDB Lightning
    * 修复在生成 KV 数据时可能发生的 panic 问题 #1127
    * 修复数据导入期间 Batch Split Region 因键的总大小超过 Raft 条目限制而可能失败的问题 #969
    * 修复在导入 CSV 文件时，如果文件的最后一行未包含换行符 (`\r\n`) 会导入报错的问题 #1133
    * 修复待导入的目标表中包含 double 类型的自增列时会导致表的 auto_Increment 值异常的问题 #1178
  - Backup & Restore (BR)
    * 修复备份期间少数 TiKV 节点不可用导致的备份中断问题 #980
  - TiCDC
    * 修复 Unified Sorter 中的并发问题并过滤无用的错误消息 #1678
    * 修复同步到 MinIO 时，重复创建目录会导致同步中断的问题 #1463
    * 默认开启会话变量 `explicit_defaults_for_timestamp`，使得下游 MySQL 5.7 和上游 TiDB 的行为保持一致 #1585
    * 修复错误地处理 `io.EOF` 可能导致同步中断的问题 #1633
    * 修正 TiCDC 面板中的 TiKV CDC endpoint CPU 统计信息 #1645
    * 增加 `defaultBufferChanSize` 来避免某些情况下同步阻塞的问题 #1259
    * 修复 Avro 输出中丢失时区信息的问题 #1712
    * 支持清理 Unified Sorter 过期的文件并禁止共享 `sort-dir` 目录 #1742
    * 修复存在大量过期 Region 信息时 KV 客户端可能锁死的问题 #1599
    * 修复 `--cert-allowed-cn` 参数中错误的帮助消息 #1697
    * 修复因更新 `explicit_defaults_for_timestamp` 而需要 MySQL SUPER 权限的问题 #1750
    * 添加 sink 流控以降低内存溢出的风险 #1840
    * 修复调度数据表时可能发生的同步终止问题 #1828
    * 修复 TiCDC changefeed 断点卡住导致 TiKV GC safe point 不推进的问题 #1759

16.17 v5.0

16.17.1 TiDB 5.0.6 Release Notes

发版日期：2021 年 12 月 31 日

TiDB 版本：5.0.6

16.17.1.1 兼容性更改

- Tools
  - TiCDC
    * 将 cdc server 命令的错误输出从标准输出 (stdout) 改为标准错误 (stderr) #3133
    * 将 Kafka sink 模块的 `max-message-bytes` 默认值设置为 10M #3081
    * 将 Kafka Sink `partition-num` 的默认值改为 3，使 TiCDC 更加平均地分发消息到各个 Kafka partition #3337

16.17.1.2 提升改进

- TiDB
  - 当 coprocessor 遇到锁时，在调试日志中显示受影响的 SQL 语句帮助诊断问题 #27718
- TiKV
  - 将插入 SST 文件时的校验操作从 Apply 线程池移动到 Import 线程池，从而提高 SST 文件的插入速度 #11239
  - 在 Raft 日志垃圾回收模块中添加了更多监控指标，从而定位该模块中出现的性能问题 #11374
  - 折叠了 Grafana Dashboard 中与 Storage 相关的不常用的监控指标 #11681
- PD
  - 优化调度器退出的速度 #4146
  - 通过允许 `scatter-range-scheduler` 调度器调度空 Region 和修复该调度器的配置，使该调度器的调度结果更加均匀 #4497
  - 允许 Evict Leader 调度器调度拥有不健康副本的 Region #4093
- Tools
  - TiCDC
    * 优化 TiKV 重新加载时的速率限制控制，缓解 changefeed 初始化时 gPRC 的拥堵问题 #3110
    * 为 EtcdWorker 添加 tick 频率限制，防止 PD 的 etcd 写入次数过于频繁影响 PD 服务 #3112
    * Kafka sink 模块添加默认的元数据获取超时时间配置 (`config.Metadata.Timeout`) #3352
    * 将参数 `max-message-bytes` 的默认值设置为 10M，减少 Kafka 不能发送消息的概率 #3081
    * 增加更多 Prometheus 和 Grafana 监控告警参数，包括 `no owner alert`、`mounter row`、`table sink total row` 和 `buffer sink total row` #4054 #1606

- Backup & Restore (BR)
    * 遇到 PD 请求错误或 TiKV I/O 超时错误时重试 BR 任务 #27787
    * 增强恢复的鲁棒性 #27421
- TiDB Lightning
    * 支持导入数据到带有表达式索引或带有基于虚拟生成列的索引的表中 #1404

16.17.1.3 Bug 修复

- TiDB
    - 修复乐观事务冲突可能导致事务相互阻塞的问题 #11148
    - 修复运行 MPP 查询时出现 `invalid cop task execution summaries length` 相关日志的问题 #1791
    - 修复 DML 和 DDL 语句并发执行时可能发生 panic 的问题 #30940
    - 修复 `grant` 和 `revoke` 操作在授予和撤销全局权限时，报 `privilege check fail` 错误的问题 #29675
    - 修复在某些场景下执行 `ALTER TABLE.. ADD INDEX` 语句时 TiDB panic 的问题 #27687
    - 修复配置项 `enforce-mpp` 在 v5.0.4 中不生效的问题 #29252
    - 修复当 `CASE WHEN` 函数和 `ENUM` 类型一起使用时的崩溃问题 #29357
    - 修复 `microsecond` 函数的向量化表达式版本结果不正确的问题 #29244
    - 修复 `auto analyze` 输出的日志信息不完整的问题 #29188
    - 修复 hour 函数在向量化表达式中执行结果错误的问题 #28643
    - 修复当不支持的 `cast` 被下推到 TiFlash 时出现的非预期错误，例如 `tidb_cast to Int32 is not supported` #23907
    - 修复 MPP 节点的可用性检测在某些边界场景中无法工作的问题 #3118
    - 修复分配 `MPP task ID` 时出现 `DATA RACE` 的问题 #27952
    - 修复删除空的 `dual table` 后 MPP 查询出现 `index out of range` 报错的问题 #28250
    - 修复并行插入无效日期类型值时，TiDB panic 的问题 #25393
    - 修复在 MPP 模式下查询时，报 `can not found column in Schema column` 错误的问题 #30980
    - 修复 TiDB 在 TiFlash 关闭时可能出现 panic 的问题 #28096
    - 修复优化器在进行 join reorder 优化时，报 `index out of range` 错误的问题 #24095
    - 修复当 `ENUM` 类型作为 `IF` 或 `CASE WHEN` 等控制函数的参数时，返回结果不正确的问题 #23114
    - 修复 `CONCAT(IFNULL(TIME(3))` 计算结果出错的问题 #29498
    - 修复当 `GREATEST` 和 `LEAST` 函数传入无符号整型值时，计算结果出错的问题 #30101
    - 修复当 SQL 语句中存在 JSON 类型列与 CHAR 类型列连接时，SQL 出错的问题 #29401
    - 修复错误使用 lazy existence 检查和 untouched key optimization 可能导致数据索引不一致的问题 #30410
    - 修复窗口函数在使用事务时，计算结果与不使用事务时的计算结果不一致的问题 #29947
    - 修复 SQL 语句中包含 `cast(integer as char)union string` 时计算结果出错的问题 #29513
    - 修复将 `Decimal` 转为 `String` 时长度信息错误的问题 #29417
    - 修复 SQL 语句中带有 NATURAL JOIN 时，意外报错 `Column 'col_name' in field list is ambiguous` 的问题 #25041
    - 修复由于 `tidb_enable_vectorized_expression` 设置的值不同（on 或 `off`）导致 `GREATEST` 函数返回结果不一致的问题 #29434
    - 修复在某些情况下 Planner 可能缓存无效 `join` 计划的问题 #28087
    - 修复在某些情况下 SQL 语句在 join 结果上计算聚合函数时，报错 `index out of range [1] with length 1` 的问题 #1978

- TiKV
  - 修复某个 TiKV 节点停机会导致 Resolved Timestamp 进度落后的问题 #11351
  - 修复 Raft client 中 batch 消息过大的问题 #9714
  - 修复在极端情况下同时进行 Region Merge、ConfChange 和 Snapshot 时，TiKV 会出现 Panic 的问题 #11475
  - 修复逆序扫表时 TiKV 无法正确读到内存锁的问题 #11440
  - 修复 Decimal 除法计算的结果为 0 时符号为负的问题 #29586
  - 修复垃圾回收任务的堆积可能会导致 TiKV 内存耗尽的问题 #11410
  - 修复 TiKV 监控项中实例级别 gRPC 的平均延迟时间不准确的问题 #11299
  - 修复因统计线程监控数据导致的内存泄漏 #11195
  - 修复在缺失下游数据库时出现 TiCDC Panic 的问题 #11123
  - 修复因 Congest 错误而导致的 TiCDC 频繁增加 scan 重试的问题 #11082
  - 修复因 channel 打满而导致的 Raft 断连的问题 #11047
  - 修复在 TiDB Lightning 导入数据时 TiKV 会因为文件不存在而出现 Panic 的问题 #10438
  - 修复由于无法在 `Max/Min` 函数中正确识别 Int64 是否为有符号整数，导致 `Max/Min` 函数的计算结果不正确的问题 #10158
  - 修复当 TiKV 副本节点获取了快照后，TiKV 会因为无法准确地修改元信息而导致该副本节点宕机的问题 #10225
  - 修复 backup 线程池泄漏的问题 #10287
  - 修复将不合法字符串转化为浮点数的问题 #23322
- PD
  - 修复 TiKV 节点缩容后可能导致 Panic 的问题 #4344
  - 修复 Operator 被停止服务的节点阻塞的问题 #3353
  - 修复因 Region syncer 卡住而导致 leader 选举慢的问题 #3936
  - 修复当对宕机的节点进行修复时删除副本的速度会受限的问题 #4090
  - 修复当 Region 心跳低于 60 秒时热点 Cache 不能清空的问题 #4390
- TiFlash
  - 修复整数类型主键的列类型设置为较大范围后数据可能不一致的问题
  - 修复在某些平台 TiFlash 由于找不到 `libnsl.so` 库而无法启动的问题，比如某些 ARM 平台
  - 修复 TiFlash 的 Store size 统计信息与实际容量不一致的问题
  - 修复 TiFlash 由于 `Cannot open file` 错误而导致的进程失败
  - 修复当 MPP 查询被终止时，TiFlash 偶发的崩溃问题
  - 修复查询报错 `3rd arguments of function substringUTF8 must be constants`
  - 修复由于过多 OR 条件导致的查询报错
  - 修复 `where <string>` 查询结果出错的问题
  - 修复在 TiFlash 与 TiDB/TiKV 之间 `CastStringAsDecimal` 行为不一致的问题。
  - 修复由于 “different types: expected Nullable(Int64), got Int64” 错误而导致查询报错的问题
  - 修复由于 “Unexpected type of column: Nullable(Nothing)” 错误而导致的查询报错
  - 修复 `Decimal` 类型比较时，可能导致数据溢出并导致查询失败的问题
- Tools
  - TiCDC

    * 修复当开启 `force-replicate` 时，可能某些没有有效索引的分区表被忽略的问题 #2834
    * 修复 cdc cli 接收到非预期参数时截断用户参数，导致用户输入数据丢失问题 #2303
    * 修复当写入 Kafka 消息发生错误时，TiCDC 同步任务推进可能停滞的问题 #2978
    * 修复 MQ sink 模块不支持非 binary json 类型列解析 #2758
    * 将参数 `max-message-bytes` 的默认值设置为 `10M`，减少 Kafka 发送过大消息的概率 #3081
    * 修复当上游 TiDB 实例意外退出时，TiCDC 同步任务推进可能停滞的问题 #3061
    * 修复当 TiKV 向同一 Region 发送重复请求时，TiCDC 进程 Panic 的问题 #2386
    * 修复在多个 TiKV 崩溃或强制重启时可能遇到复制中断的问题 #3288
    * 修复监控 checkpoint lag 出现负值的问题 #3010
    * 修复 MySQL sink 模块出现死锁时告警过于频繁的问题 #2706
    * 修复 Avro sink 模块不支持解析 JSON 类型列的问题 #3624
    * 修复 TiKV owner 重启时 TiCDC 从 TiKV 读取到错误的元数据快照的问题 #2603
    * 修复执行 DDL 后的内存泄漏的问题 #3174
    * 修复 Canal 和 Maxwell 协议下 TiCDC 没有自动开启 `enable-old-value` 选项的问题 #3676
    * 修复 `cdc server` 命令在 Red Hat Enterprise Linux 系统的部分版本（如 6.8、6.9 等）上运行时出现时区错误的问题 #3584
    * 修复当 Kafka 为下游时 `txn_batch_size` 监控指标数据不准确的问题 #3431
    * 修复 tikv_cdc_min_resolved_ts_no_change_for_1m 监控在没有 changefeed 的情况下持续更新的问题 #11017
    * 修复人为删除 etcd 任务的状态时导致 TiCDC panic 的问题 #2980
    * 修复当发生 ErrGCTTLExceeded 错误时，changefeed 不快速失败的问题 #3111
    * 修复当扫描存量数据耗时过长时，可能由于 TiKV 进行 GC 而导致存量数据扫描失败的问题 #2470
    * 修复在容器环境中 OOM 的问题 #1798
  - Backup & Restore (BR)
    * 修复 BR 中平均速度 (average-speed) 不准确的问题 #1405
  - Dumpling
    * 修复 Dumpling 在导出包含复合主键或唯一约束的表时导出速度过慢的问题 #29386

16.17.2 TiDB 5.0.5 Release Note

发版日期：2021 年 12 月 3 日

TiDB 版本：5.0.5

16.17.2.1 Bug 修复

- TiKV
  - 修复 GcKeys 任务被多个键调用时无法正常进行，导致 Compaction Filter GC 可能不删除 MVCC deletion 信息的问题 #11217

16.17.3 TiDB 5.0.4 Release Notes

发版日期：2021 年 9 月 27 日

TiDB 版本：5.0.4

16.17.3.1 兼容性更改

- TiDB
  - 修复在新会话中执行 `SHOW VARIABLES` 速度较慢的问题。该修复回退了 #19341 中的部分更改，可能会引起兼容性问题。#24326
  - 将系统变量 `tidb_stmt_summary_max_stmt_count` 的默认值从 200 修改为 3000 #25873
  - 以下 Bug 修复涉及执行结果变化，可能引起兼容性变化：
    * 修复了当 `UNION` 的子计划包含 `NULL` 值时 TiDB 返回错误结果的问题 #26559
    * 修复了 `greatest(datetime)union null` 返回空字符串的问题 #26532
    * 修复了 `last_day` 函数的行为在 SQL 模式下不兼容的问题 #26000
    * 修复了 `having` 子句可能执行错误的问题 #26496
    * 修复了当 `between` 表达式两边的 collation 不一致会导致查询结果错误的问题 #27146
    * 修复了当 `group_concat` 函数包含非 `bin` 的 collation 时查询结果错误的问题 #27429
    * 修复了当开启 New Collation 时，在多列上执行 `count(distinct)` 表达式结果错误的问题 #27091
    * 修复了 `extract` 函数的参数是负数时查询结果错误的问题 #27236
    * 修复了当 `SQL_MODE` 为 ‘STRICT_TRANS_TABLES’ 时，插入非法时间不报错的问题 #26762
    * 修复了当 `SQL_MODE` 为 ‘NO_ZERO_IN_DATE’ 时，使用非法的默认时间不报错的问题 #26766
    * 修复了索引前缀的查询范围问题 #26029
    * 修复了 `LOAD DATA` 语句可能会异常导入非 utf8 数据的问题 #25979
    * 修复了当二级索引包含主键中的列时，`insert ignore on duplicate update` 可能插入错误数据的问题 #25809
    * 修复了当分区表有聚簇索引时，`insert ignore duplicate update` 可能插入错误数据的问题 #25846
    * 修复了当 point get 或 batch point get 的查找键是 `ENUM` 类型时，查询结果可能错误的问题 #24562
    * 修复了 `BIT` 类型值做除法时查询结果不正确的问题 #23479
    * 修复了 `prepared` 语句和直接查询的结果可能不一致的问题 #22949
    * 修复了当 `YEAR` 类型与字符串或整数类型比较时，结果可能错误的问题 #23262

16.17.3.2 功能增强

- TiDB
  - 支持将系统变量 `tidb_enforce_mpp` 的值设为 1 以忽略优化器代价估算，强制使用 MPP 模式 #26382
- TiKV
  - 支持动态修改 TiCDC 配置 #10645
- PD

  - 为 TiDB Dashboard 增加基于 OIDC 的 SSO 支持 #3884

- TiFlash

  - 支持在 DAG 请求中使用 HAVING() 函数
  - 支持 DATE() 函数
  - 为 Grafana 面板增加每个实例的写入吞吐

16.17.3.3 提升改进

- TiDB

  - 基于直方图的 row count 来触发 auto-analyze #24237
  - 当一个 TiFlash 节点此前因宕机重启过，TiDB 一段时间内不给该节点发送请求 #26757
  - 增大 `split region` 的速度上限，使 `split table` 和 `presplit` 更稳定 #22969
  - 支持 MPP 查询的重试 #26483
  - 在发起 MPP 查询之前检查 TiFlash 的可用性 #1807
  - 支持稳定结果模式，使查询结果更稳定 #26084
  - 支持 MySQL 系统变量 `init_connect` 及相关功能 #18894
  - 在 MPP 模式中彻底下推 COUNT(DISTINCT) 聚合函数 #25861
  - 当聚合函数在 EXPLAIN 语句中不能被下推时打印警告日志 #25736
  - 在 Grafana 监控中为 TiFlashQueryTotalCounter 加上错误标签 #25327
  - 支持在 HTTP API 中通过二级索引查询聚簇索引表的 MVCC 数据 #24209
  - 优化 `prepared` 语句在解析器的内存分配 #24371

- TiKV

  - 分离处理读写的 ready 状态以减少读延迟 #10475
  - 减少 Resolved TS 消息的大小以节省网络带宽 #2448
  - 当 slogger 线程过载且队列已满时，删除日志而不是阻塞线程 #10841
  - 使 TiKV Coprocessor 慢日志只考虑处理请求所花费的时间 #10841
  - 使预写尽可能具有幂等性，以减少出现未确定错误的可能 #10587
  - 避免在低写入流量下误报 “GC can not work” 错误 #10662
  - 使数据库在备份时总是与原始集群大小相匹配 #10643
  - 确保 Panic 信息刷新到日志 #9955

- PD

  - 提升了 PD 之间同步 Region 信息的性能 #3993

- Tools

  - Dumpling

    * 支持对 MySQL 兼容的特定数据库进行备份，这些数据库不支持 `START TRANSACTION ... WITH CONSISTENT SNAPSHOT` 和 `SHOW CREATE TABLE` 语法 #309

  - TiCDC

    * 优化 Unified Sorter 使用内存排序时的内存管理 #2553

* 禁止使用不同的 major 和 minor 版本启动 TiCDC 节点 #2598
* 当某张表的 Region 从某个 TiKV 节点全部迁移走时，减少 goroutine 资源的使用 #2284
* 移除 `file sorter` 文件排序器 #2326
* 总是从 TiKV 获取行变更的历史值 (old value)，输出会根据 `enable-old-value` 配置进行调整 #2301
* 优化连接 PD 时缺少证书情况下的报错提示 #1973
* 在高并发下减少 workerpool 中创建的 goroutine 数量 #2211
* 为所有 KV 客户端创建全局共享的 gRPC 连接池 #2533

16.17.3.4 Bug 修复

- TiDB
  - 修复了当查询分区表且分区键带有 `IS NULL` 条件时，TiDB 可能 panic 的问题 #23802
  - 修复了 FLOAT64 类型的溢出检查与 MySQL 不一致的问题 #23897
  - 修复了 `case when` 表达式的字符集和排序规则错误的问题 #26662
  - 修复了提交悲观事务可能会导致写冲突的问题 #25964
  - 修复了在悲观事务中索引键值可能会被重复提交的问题 #26359 #10600
  - 修复了清除 Async Commit 锁时可能会导致 TiDB panic 的问题 #25778
  - 修复了使用 `INDEX MERGE` 时可能找不到列的问题 #25045
  - 修复了使用 `ALTER USER REQUIRE SSL` 会清空用户的 `authentication_string` 的问题 #25225
  - 修复了新集群上系统变量 `tidb_gc_scan_lock_mode` 的值显示为 “PHYSICAL” 而实际是 “LEGACY” 的问题 #25100
  - 修复了系统表 `TIKV_REGION_PEERS` 没有正确显示 DOWN 状态的问题 #24879
  - 修复了使用 HTTP API 时导致内存泄漏的问题 #24649
  - 修复了视图不支持 DEFINER 的问题 #24414
  - 修复了 `tidb-server --help` 退出状态是 2 的问题 #24046
  - 修复了设置全局系统变量 `dml_batch_size` 不生效的问题 #24709
  - 修复了同时使用 `read_from_storage` 和分区表会报错的问题 #20372
  - 修复了 TiDB 在执行投影算子时会 Panic 的问题 #24264
  - 修复了统计信息可能导致查询 Panic 的问题 #24061
  - 修复了在 BIT 类型的列上使用 `approx_percentile` 函数可能 Panic 的问题 #23662
  - 修复了 Grafana 上 Coprocessor Cache 面板的数据显示不正确的问题 #26338
  - 修复了并发 truncate 同一个分区会导致 DDL 语句执行卡住的问题 #26229
  - 修复当会话变量用作 `GROUP BY` 项时查询结果出错的问题 #27106
  - 修复连接表时 VARCHAR 类型与时间戳之间错误的隐式转换 #25902
  - 修复相关子查询语句中的错误结果 #27233
- TiKV
  - 修复损坏的快照文件可能会造成磁盘空间无法回收的问题 #10813
  - 修复了 TiKV 在启用 Titan 并从 5.0 以前的版本升级时出现 Panic 的问题 #10843
  - 修复了高版本的 TiKV 无法回滚到 v5.0.x 的问题 #10843
  - 修复了启用 Titan 并从 v5.0 以前的版本升级到 v5.0 及以后的版本时 TiKV 崩溃的问题（例如，如果集群从 TiKV v3.x 升级并在升级之前启用了 Titan，则该集群可能会遇到该问题）#10774
  - 修复了遗留的悲观锁导致的解析失败的问题 #26404
  - 修复在某些平台上计算时间间隔出现 Panic 的问题 #10571

  - 修复 Load Base Split 中 `batch_get_command` 的键值未编码问题 #10542

- PD

  - 修复 PD 未能及时修复 Down Peer 副本的问题 #4077
  - 修复了 `replication.max-replicas` 更新后默认的 Placement Rules 副本数量不变的问题 #3886
  - 修复了 PD 在扩容 TiKV 时可能会 Panic 的问题 #3868
  - 修复多个调度器运行时会造成调度冲突的问题 #3807
  - 修复了删除调度器后调度器可能再次出现的问题 #2572

- TiFlash

  - 修复执行扫表任务时潜在的进程崩溃问题
  - 修复执行 MPP 任务时潜在的内存泄漏问题
  - 修复处理 DAG 请求时出现 `duplicated region` 报错的问题
  - 修复执行 COUNT 或 `COUNT DISTINCT` 函数时出现非预期结果的问题
  - 修复执行 MPP 任务时潜在的进程崩溃问题
  - 修复 TiFlash 多盘部署时无法恢复数据的潜在问题
  - 修复析构 `SharedQueryBlockInputStream` 时出现进程崩溃的潜在问题
  - 修复析构 `MPPTask` 时出现进程崩溃的潜在问题
  - 修复 TiFlash 无法建立 MPP 连接时出现非预期结果的问题
  - 修复解锁时潜在的进程崩溃问题
  - 修复写入压力大时 metrics 中 store size 不准确的问题
  - 修复当查询过滤条件包含诸如 CONSTANT、<、<=、>、>= 或 COLUMN 时出现错误结果的问题
  - 修复 TiFlash 长时间运行后无法回收 Delta 数据的潜在问题
  - 修复 metrics 显示错误数值的潜在问题
  - 修复多盘部署时数据不一致的潜在问题

- Tools

  - Dumpling

    * 修复在 MySQL 8.0.3 或更高版本执行 `show table status` 语句卡住的问题 #322

  - TiCDC

    * 修复将 `mysql.TypeString, mysql.TypeVarString, mysql.TypeVarchar` 等类型的数据编码为 JSON 时进程崩溃的问题 #2758
    * 修复重新调度一张表时多个处理器将数据写入同一张表引发的数据不一致的问题 #2417
    * 减小 gRPC 窗口来避免 Region 数量过多时触发内存溢出 #2724
    * 修复内存压力大时 gRPC 连接频繁断开的错误 #2202
    * 修复 TiCDC 在处理无符号 `TINYINT` 类型时崩溃的问题 #2648
    * 修复 TiCDC Open Protocol 在上游插入事务并删除同一行数据的情况下输出空值的问题 #2612
    * 修复同步任务从一个表结构变更的 finish TS 开始时 DDL 处理失败的问题 #2603
    * 修复无响应的下游中断 old owner 中的同步任务直到该任务超时的问题 #2295
    * 修复元信息管理问题 #2558
    * 修复 TiCDC 切换 owner 后出现数据不一致的问题 #2230
    * 修复 `capture list` 命令输出中出现已过期 capture 的问题 #2388
    * 修复集成测试中遇到的由于 DDL Job 重复导致的 `ErrSchemaStorageTableMiss` 错误 #2422

    * 修复遇到 ErrGCTTLExceeded 错误时 changefeed 无法被删除的问题 #2391
    * 修复同步数据量大的表到 cdclog 失败的问题 #1259 #2424
    * 修复客户端向后兼容性问题 #2373
    * 修复 SinkManager 中对 map 的不安全并发访问 #2299
    * 修复 owner 在执行 DDL 语句时崩溃可能导致 DDL 任务丢失的问题 #1260
    * 修复在 Region 初始化时立刻执行清锁的问题 #2188
    * 修复创建新的分区表时部分分区被重复分发的问题 #2263
    * 修复同步任务已删除但 TiCDC 持续报警的问题 #2156

16.17.4 TiDB 5.0.3 Release Notes

发版日期：2021 年 7 月 2 日

TiDB 版本：5.0.3

16.17.4.1 兼容性更改

- TiDB
    - v4.0 集群升级到 v5.0 或更高版本（dev 和 v5.1）的集群后，tidb_multi_statement_mode 变量的默认值由 WARN 变为 OFF
    - 兼容 MySQL 5.7 的 noop 变量 innodb_default_row_format，配置此变量无实际效果 #23541

16.17.4.2 功能增强

- Tools
    - TiCDC
        * 增加 HTTP API 获取 TiCDC changefeed 信息和节点健康信息 #1955
        * 为 kafka 下游增加 SASL/SCRAM 支持 #1942
        * 使 TiCDC 在 server 级别支持 --data-dir 配置 #2070

16.17.4.3 提升改进

- TiDB
    - 支持将 TopN 算子下推到 TiFlash #25162
    - 支持将内置函数 json_unquote() 下推到 TiKV #24415
    - 支持在 Dual 表上移除 Union 算子的优化 #25614
    - 支持将内置函数 replace() 下推到 TiFlash #25565
    - 支持将内置函数 unix_timestamp()、concat()、year()、day()、datediff()、datesub()、concat_ws() 下推到 TiFlash #25564
    - 优化聚合算子的代价常数 #25241
    - 支持将 Limit 算子下推到 TiFlash #25159
    - 支持将内置函数 str_to_date() 下推到 TiFlash #25148

  - 允许 MPP outer join 根据表行数选择构建表 #25142
  - 支持将内置函数 `left()`、`right()`、`abs()` 下推到 TiFlash #25133
  - 支持将 Broadcast Cartesian Join 下推到 TiFlash #25106
  - 支持将 `Union All` 算子下推到 TiFlash #25051
  - 支持 MPP 查询任务按 Region 均衡到不同 TiFlash 节点上 #24724
  - 支持执行 MPP 查询后将缓存中过时的 Region 无效化 #24432
  - 提升内置函数 `str_to_date` 在格式指定器中 `%b/%M/%r/%T` 的 MySQL 兼容性 #25767

- TiKV

  - 限制 TiCDC sink 的内存消耗 #10305
  - 为 TiCDC old value 缓存增加基于内存使用量的上限 #10313

- PD

  - 将 TiDB Dashboard 升级至 v2021.06.15.1 #3798

- TiFlash

  - 支持将 `STRING` 类型转换为 `DOUBLE` 类型
  - 支持 `STR_TO_DATE()` 函数
  - 通过多线程优化右外连接中的非连接数据
  - 支持笛卡尔积 Join
  - 支持 `LEFT()` 和 `RIGHT()` 函数
  - 支持在 MPP 查询中自动清理过期的 Region 信息
  - 支持 `ABS()` 函数

- Tools

  - TiCDC

    * 优化 gRPC 的重连逻辑，提升 KV client 的吞吐 #1586 #1501 #1682 #1393 #1847 #1905 #1904
    * 优化 sorter I/O 报错信息

16.17.4.4 Bug 修复

- TiDB

  - 修复在 `SET` 类型列上 Merge Join 结果不正确的问题 #25669
  - 修复 `IN` 表达式参数的数据腐蚀问题 #25591
  - 避免 GC 的 session 受全局变量的影响 #24976
  - 修复了在窗口函数查询中使用 `Limit` 时出现 panic 问题 #25344
  - 修复查询分区表时使用 `Limit` 返回错误值的问题 #24636
  - 修复了 `IFNULL` 在 `ENUM` 或 `SET` 类型上不能正确生效的问题 #24944
  - 修复了 Join 子查询中的 `count` 被改写为 `first_row` 导致结果不正确的问题 #24865
  - 修复了 TopN 算子下使用 `ParallelApply` 查询时卡住的问题 #24930
  - 修复了使用含有多列的前缀索引查询时出现多余结果的问题 #24356
  - 修复了操作符 `<=>` 不能正确生效的问题 #24477

  - 修复并行 Apply 算子的数据竞争问题 #23280
  - 修复对 PartitionUnion 算子的 IndexMerge 结果排序时出现 `index out of range` 错误 #23919
  - 修复 `tidb_snapshot` 被允许设置为非预期的过大值，而可能造成事务隔离性被破坏的问题 #25680
  - 修复 ODBC 类常数（例如 `{d '2020-01-01'}`）不能被用作表达式的问题 #25531
  - 修复 `SELECT DISTINCT` 被转化为 Batch Get 而导致结果不正确的问题 #25320
  - 修复无法触发将查询从 TiFlash 回退到 TiKV 的问题 #23665 #24421
  - 修复在检查 `only_full_group_by` 时的 `index-out-of-range` 错误 #23839
  - 修复关联子查询中 Index Join 的结果不正确问题 #25799

- TiKV

  - 修复错误的 `tikv_raftstore_hibernated_peer_state` 监控指标 #10330
  - 修复 coprocessor 中 `json_unquote()` 函数错误的参数类型 #10176
  - 正常关机时跳过清理 Raftstore 的回调从而避免在某些情况下破坏事务的 ACID #10353 #10307
  - 修复在 Leader 上 Replica Read 共享 Read Index 的问题 #10347
  - 修复 coprocessor 转换 `DOUBLE` 到 `DOUBLE` 的错误函数 #25200

- PD

  - 修复在 scheduler 启动之后，加载 TTL 配置产生的数据竞争问题 #3771
  - 修复 `is_learner` 字段在 TiDB 的 `TIKV_REGION_PEERS` 表中显示异常的问题 #3372 #24293
  - 修复在一个 zone 内所有 TiKV 节点下线或宕机的情况下，PD 不往其他 zone 调度数据的问题 #3705
  - 修复在添加 scatter range 调度器后导致 PD 挂掉的问题 #3762

- TiFlash

  - 修复因 split 失败而不断重启的问题
  - 修复无法删除 Delta 历史数据的潜在问题
  - 修复在 `CAST` 函数中为非二进制字符串填充错误数据的问题
  - 修复处理包含复杂 `GROUP BY` 列的聚合查询时结果不正确的问题
  - 修复写入压力过大时出现进程崩溃的问题
  - 修复右连接键不为空且左连接键可为空时进程崩溃的问题
  - 修复 `read-index` 请求耗时长的潜在问题
  - 修复读负载高的情况下进程崩溃的问题
  - 修复 `Date_Format` 函数在参数类型为 `STRING` 且包含 `NULL` 值时可能导致 TiFlash server 崩溃的问题

- Tools

  - TiCDC

    * 修复 TiCDC owner 在刷新 checkpoint 时异常退出的问题 #1902
    * 修复写 MySQL 下游出错暂停时 MySQL 连接泄漏的问题 #1946
    * 修复 TiCDC 读取 `/proc/meminfo` 失败时出现的 panic 问题 #2024
    * 减少 TiCDC 运行时的内存使用 #2012 #1958
    * 修复 resolved ts 计算慢导致 TiCDC panic 的问题 #1576
    * 修复 processor 潜在的死锁问题 #2142

  - Backup & Restore (BR)

    * 修复 BR 恢复中忽略了所有系统表的问题 #1197 #1201

    * 修复在 Backup & Restore 数据恢复期间开启 TDE 会报出文件已存在的错误 #1179

  - TiDB Lightning

    * 修复 TiDB Lightning 在特殊数据下 panic 的问题 #1213
    * 修复 TiDB Lightning 导入大文件拆分时遇到的 EOF 报错问题 #1133
    * 修复 TiDB Lightning 导入含 `auto_increment` 的 DOUBLE 或 FLOAT 类型列的表时生成极大 base 值的问题 #1186
    * 修复 TiDB Lightning 解析 Parquet 文件中 DECIMAL 类型数据失败的问题 #1277

16.17.5 TiDB 5.0.2 Release Notes

发版日期：2021 年 6 月 10 日

TiDB 版本：5.0.2

16.17.5.1 兼容性更改

- Tools

  - TiCDC

    * 在 `cdc cli changefeed` 命令中废弃 `--sort-dir` 参数，用户可在 `cdc server` 命令中设定 `--sort-dir` #1795

16.17.5.2 新功能

- TiKV

  - 默认开启 Hibernate Region 特性 #10266

16.17.5.3 提升改进

- TiDB

  - 当内存中的统计信息缓存是最新的时，避免后台作业频繁读取 `mysql.stats_histograms` 表造成高 CPU 使用率 #24317

- TiKV

  - BR 支持 S3 兼容的存储（基于 virtual-host 寻址模式）#10243
  - 为 TiCDC 扫描的速度添加背压 (back pressure) 功能 #10151
  - 减少 TiCDC 进行初次扫描的内存使用量 #10133
  - 提升了悲观事务中 TiCDC Old Value 的缓存命中率 #10089
  - 让 Region 分裂更均匀，缓解有写入热点时 Region 大小的增长速度超过分裂速度的问题 #9785

- TiFlash

  - 优化锁操作以避免 DDL 语句和读数据相互阻塞

  - 支持 INTEGER 和 REAL 类型转化为 REAL 类型

- Tools

  - TiCDC

    * 添加关于数据表内存使用情况的监控 #1885
    * 优化排序阶段的内存和 CPU 使用 #1863
    * 删除了一些可能让用户困惑的无用日志信息 #1759

  - Backup & Restore (BR)

    * 优化了一些含糊的报错信息 #1132
    * 支持检查备份的版本信息 #1091
    * 支持备份和恢复 mysql schema 下的系统表 #1143 #1078

  - Dumpling

    * 修复备份失败却没有错误输出的问题 #280

16.17.5.4 Bug 修复

- TiDB

  - 修复了在某些情况下，使用前缀索引和 Index Join 导致的 panic 的问题 #24547 #24716 #24717
  - 修复了 point get 的 prepare plan cache 被事务中的 point get 语句不正确使用的问题 #24741
  - 修复了当排序规则为 ascii_bin 或 latin1_bin 时，写入错误的前缀索引值的问题 #24569
  - 修复了正在执行的事务被 GC worker 中断的问题 #24591
  - 修复了当 new-collation 开启且 new-row-format 关闭的情况下，点查在聚簇索引下可能出错的问题 #24541
  - 为 Shuffle Hash Join 重构分区键的转换功能 #24490
  - 修复了当查询包含 HAVING 子句时，在构建计划的过程中 panic 的问题 #24045
  - 修复了列裁剪优化导致 Apply 算子和 Join 算子执行结果错误的问题 #23887
  - 修复了从 Async Commit 回退的主锁无法被清除的问题 #24384
  - 修复了一个统计信息 GC 的问题，该问题可能导致重复的 fm-sketch 记录 #24357
  - 当悲观锁事务收到 ErrKeyExists 错误时，避免不必要的悲观事务回滚 #23799
  - 修复了当 sql_mode 包含 ANSI_QUOTES 时，数值字面值无法被识别的问题 #24429
  - 禁止如 INSERT INTO table PARTITION (<partitions>)... ON DUPLICATE KEY UPDATE 的语句从 non-listed partitions 读取数据 #24746
  - 修复了当 SQL 语句包含 GROUP BY 以及 UNION 时，可能会出现的 index out of range 的问题 #24281
  - 修复了 CONCAT 函数错误处理排序规则的问题 #24296
  - 修复了全局变量 collation_server 对新会话无法生效的问题 #24156

- TiKV

  - 修复了由于读取旧值而导致的 TiCDC OOM 问题 #9996 #9981
  - 修复了聚簇主键列在次级索引上的 latin1_bin 字符集出现空值的问题 #24548
  - 新增 abort-on-panic 配置，允许 TiKV 在 panic 时生成 core dump 文件。用户仍需正确配置环境以开启 core dump。 #10216
  - 修复了 TiKV 不繁忙时 point get 查询性能回退的问题 #10046

- PD
  - 修复在 store 数量多的情况下，切换 PD Leader 慢的问题 #3697
  - 修复删除不存在的 evict leader 调度器时出现 panic 的问题 #3660
  - 修复 offline peer 在合并完后未更新统计的问题 #3611
- TiFlash
  - 修复并发复制共享 Delta 索引导致结果错误的问题
  - 修复当存在数据缺失的情况下 TiFlash 无法启动的问题
  - 修复旧的 dm 文件无法被自动清理的问题
  - 修复 TiFlash 在 Compaction Filter 特性开启时可能崩溃的问题
  - 修复 ExchangeSender 可能传输重复数据的问题
  - 修复了从 Async Commit 回退的锁无法被 TiFlash 清除的问题
  - 修复当 TIMEZONE 类型的转换结果包含 TIMESTAMP 类型时返回错误结果的问题
  - 修复 TiFlash 在 Segment Split 期间异常退出的问题
  - 修复非根节点 MPP 任务的执行信息显示不正确的问题
- Tools
  - TiCDC
    - 修复 Avro 输出中丢失时区信息的问题 #1712
    - 支持清理 Unified Sorter 过期的文件并禁止共享 `sort-dir` 目录 #1742
    - 修复存在大量过期 Region 信息时 KV 客户端可能锁死的问题 #1599
    - 修复 `--cert-allowed-cn` 参数中错误的帮助消息 #1697
    - 修复因更新 `explicit_defaults_for_timestamp` 而需要 MySQL SUPER 权限的问题 #1750
    - 添加 sink 流控以降低内存溢出的风险 #1840
    - 修复调度数据表时可能发生的同步终止问题 #1828
    - 修复 TiCDC changefeed 断点卡住导致 TiKV GC safe point 不推进的问题 #1759
  - Backup & Restore (BR)
    - 修复 log restore 时丢失删除事件的问题 #1063
    - 修复 BR 发送过多无用 RPC 请求到 TiKV 的问题 #1037
    - 修复备份失败却没有错误输出的问题 #1043
  - TiDB Lightning
    - 修复在生成 KV 数据时可能发生的 panic 问题 #1127
    - 修复 TiDB-backend 模式下因没有开启 autocommit 而无法加载数据的问题 #1104
    - 修复数据导入期间 Batch Split Region 因键的总大小超过 Raft 条目限制而可能失败的问题 #969

16.17.6 TiDB 5.0.1 Release Notes

发版日期：2021 年 4 月 24 日

TiDB 版本：5.0.1

16.17.6.1 兼容性更改

- TiDB 配置文件
  - `committer-concurrency` 配置项的默认值由 16 改成 128。

16.17.6.2 改进提升

- TiDB
  - 支持 `VITESS_HASH()` 函数 #23915
- TiKV
  - 使用 `zstd` 压缩 Region Snapshot #10005
- PD
  - 调整 Region 分数公式使其更适用于异构集群 #3605
  - 避免在添加 `scatter region` 调度器后出现的非预期统计行为 #3602
- Tools
  - Backup & Restore (BR)
    * 删除 Summary 日志中一些容易被误解的信息 #1009

16.17.6.3 Bug 修复

- TiDB
  - 修复投影消除在投影结果为空时执行结果可能错误的问题 #24093
  - 修复列包含 `NULL` 值时查询结果在某些情况下可能错误的问题 #24063
  - 当有虚拟列参与扫描时不允许生成 MPP 计划 #24058
  - 修复 Plan Cache 中对 `PointGet` 和 `TableDual` 错误的重复使用 #24043
  - 修复优化器在为聚簇索引构建 `IndexMerge` 执行计划时出现的错误 #24042
  - 修复 BIT 类型相关错误的类型推导 #24027
  - 修复某些优化器 Hint 在 `PointGet` 算子存在时无法生效的问题 #23685
  - 修复 DDL 遇到错误回滚时可能失败的问题 #24080
  - 修复二进制字面值常量的索引范围构造错误的问题 #24041
  - 修复某些情况下 `IN` 语句的执行结果可能错误的问题 #24023
  - 修复某些字符串函数的返回结果错误的问题 #23879
  - 执行 `REPLACE` 语句需要用户同时拥有 `INSERT` 和 `DELETE` 权限 #23939
  - 修复点查时出现的性能回退 #24070
  - 修复因错误比较二进制与字节而导致的 `TableDual` 计划错误的问题 #23918
- TiKV
  - 修复了 Coprocessor 未正确处理 `IN` 表达式有符号整数或无符号整数类型数据的问题 #10018

  - 修复了在批量 ingest SST 文件后产生大量空 Region 的问题 #10015
  - 修复了在 `cast_string_as_time` 中输入非法的 UTF-8 字节后导致崩溃的问题 #9995
  - 修复了 file dictionary 文件损坏之后 TiKV 无法启动的问题 #9992

- TiFlash

  - 修复存储引擎无法删除某些范围数据的问题
  - 修复 `TIME` 类型转换为 `INT` 类型时产生错误结果的问题
  - 修复 `receiver` 可能无法在 10 秒内找到对应任务的问题
  - 修复 `cancelMPPQuery` 中可能存在无效迭代器的问题
  - 修复 `bitwise` 算子和 TiDB 行为不一致的问题
  - 修复当使用 `prefix key` 时出现范围重叠报错的问题
  - 修复字符串转换为 `INT` 时产生错误结果的问题
  - 修复连续快速写入可能导致 TiFlash 内存溢出的问题
  - 修复列名重复会引发报错的问题
  - 修复 MPP 执行计划无法被解析的问题
  - 修复 Table GC 时会引发空指针的问题
  - 修复向已被删除的表写数据时 TiFlash 进程崩溃的问题
  - 修复当使用 BR 恢复数据时 TiFlash 进程可能崩溃的问题

- Tools

  - TiDB Lightning

    * 修复导入过程中进度日志中的表数量不准确的问题 #1005

  - Backup & Restore (BR)

    * 修复实际的备份速度超过 `--ratelimit` 限制的问题 #1026
    * 修复备份期间少数 TiKV 节点不可用导致的备份中断问题 #1019
    * 修复 TiDB Lightning 在导入过程中进度日志中的表数量不准确的问题 #1005

  - TiCDC

    * 修复 Unified Sorter 中的并发问题并过滤无用的错误消息 #1678
    * 修复同步到 MinIO 时，重复创建目录会导致同步中断的问题 #1672
    * 默认开启会话变量 `explicit_defaults_for_timestamp`，使得下游 MySQL 5.7 和上游 TiDB 的行为保持一致 #1659
    * 修复错误地处理 `io.EOF` 可能导致同步中断的问题 #1648
    * 修正 TiCDC 面板中的 TiKV CDC endpoint CPU 统计信息 #1645
    * 增加 `defaultBufferChanSize` 来避免某些情况下同步阻塞的问题 #1632

16.17.7 What’s New in TiDB 5.0

发版日期：2021 年 04 月 07 日

TiDB 版本：5.0.0

5.0 版本中，我们专注于帮助企业基于 TiDB 数据库快速构建应用程序，使企业在构建过程中无需担心数据库的性能、性能抖动、安全、高可用、容灾、SQL 语句的性能问题排查等问题。

在 5.0 版本中，你可以获得以下关键特性：

- TiDB 通过 TiFlash 节点引入了 MPP 架构。这使得大型表连接类查询可以由不同 TiFlash 节点共同分担完成。当 MPP 模式开启后，TiDB 将会根据代价决定是否应该交由 MPP 框架进行计算。MPP 模式下，表连接将通过对 JOIN Key 进行数据计算时重分布（Exchange 操作）的方式把计算压力分摊到各个 TiFlash 执行节点，从而达到加速计算的目的。经测试，TiDB 5.0 在同等资源下，MPP 引擎的总体性能是 Greenplum 6.15.0 与 Apache Spark 3.1.1 两到三倍之间，部分查询可达 8 倍性能差异。
- 引入聚簇索引功能，提升数据库的性能。例如，TPC-C tpmC 的性能提升了 39%。
- 开启异步提交事务功能，降低写入数据的延迟。例如：Sysbench 设置 64 线程测试 Update index 时，平均延迟由 12.04 ms 降低到 7.01ms，降低了 41.7%。
- 通过提升优化器的稳定性及限制系统任务对 I/O、网络、CPU、内存等资源的占用，降低系统的抖动。例如：测试 8 小时，TPC-C 测试中 tpmC 抖动标准差的值小于等于 2%。
- 通过完善调度功能及保证执行计划在最大程度上保持不变，提升系统的稳定性。
- 引入 Raft Joint Consensus 算法，确保 Region 成员变更时系统的可用性。
- 优化 EXPLAIN 功能、引入不可见索引等功能帮助提升 DBA 调试及 SQL 语句执行的效率。
- 通过从 TiDB 备份文件到 Amazon S3、Google Cloud GCS，或者从 Amazon S3、Google Cloud GCS 恢复文件到 TiDB，确保企业数据的可靠性。
- 提升从 Amazon S3 或者 TiDB/MySQL 导入导出数据的性能，帮忙企业在云上快速构建应用。例如：导入 1TiB TPC-C 数据性能提升了 40%，由 254 GiB/h 提升到 366 GiB/h。

16.17.7.1 兼容性变化

16.17.7.1.1 系统变量

- 新增系统变量`tidb_executor_concurrency`，用于统一控制算子并发度。原有的 tidb_*_concurrency（例如 `tidb_projection_concurrency`）设置仍然生效，使用过程中会提示已废弃警告。
- 新增系统变量`tidb_skip_ascii_check`，用于决定在写入 ASCII 字符集的列时，是否对字符的合法性进行检查，默认为 OFF。
- 新增系统变量`tidb_enable_strict_double_type_check`，用于决定类似 “double(N)” 语法是否允许被定义在表结构中，默认为 OFF。
- 系统变量`tidb_dml_batch_size` 的默认值由 20000 修改为 0，即在 “LOAD/INSERT INTO SELECT …” 等语法中，不再默认使用 Batch DML，而是通过大事务以满足严格的 ACID 语义。

> **注意：**
>
> 该变量作用域从 session 改变为 global，且默认值从 20000 修改为 0，如果应用依赖于原始默认值，需要在升级之后使用 `set global` 语句修改该变量值为原始值。

- 临时表的语法兼容性受到`tidb_enable_noop_functions` 系统变量的控制：当 `tidb_enable_noop_functions` ↪ 为 OFF 时，`CREATE TEMPORARY TABLE` 语法将会报错。
- 新增`tidb_gc_concurrency`、`tidb_gc_enable`、`tidb_gc_life_time`、`tidb_gc_run_interval`、`tidb_gc_scan_lock_mode` ↪ 系统变量，用于直接通过系统变量调整垃圾回收相关参数。
- 系统变量`enable-joint-consensus` 默认值由 `false` 改成 `true`，默认开启 Joint consensus 功能。

- 系统变量 tidb_enable_amend_pessimistic_txn 的值由数字 0 或者 1 变更成 ON 或者 OFF。
- 系统变量tidb_enable_clustered_index 默认值由 OFF 改成 INT_ONLY 且含义有如下变化：
  - ON：开启聚簇索引，支持添加或者删除非聚簇索引。
  - OFF：关闭聚簇索引，支持添加或者删除非聚簇索引。
  - INT_ONLY：默认值，行为与 v5.0 以下版本保持一致，与 alter-primary-key = false 一起使用可控制 INT 类型是否开启聚簇索引。

> 注意：
>
> 5.0 GA 中 tidb_enable_clustered_index 的 INT_ONLY 值和 5.0 RC 中的 OFF 值含义一致，从已设置 OFF 的 5.0 RC 集群升级至 5.0 GA 后，将展示为 INT_ONLY。

16.17.7.1.2 配置文件参数

- 新增index-limit 配置项，默认值为 64，取值范围是 [64, 512]。MySQL 一张表最多支持 64 个索引，如果该配置超过默认值并为某张表创建超过 64 个索引，该表结构再次导入 MySQL 将会报错。
- 新增enable-enum-length-limit 配置项，用于兼容 MySQL ENUM/SET 元素长度并保持一致（ENUM 长度 < 255），默认值为 true。
- 删除 pessimistic-txn.enable 配置项，通过环境变量tidb_txn_mode 替代。
- 删除 performance.max-memory 配置项，通过performance.server-memory-quota 替代。
- 删除 tikv-client.copr-cache.enable 配置项，通过tikv-client.copr-cache.capacity-mb 替代，如果配置项的值为 0.0 代表关闭此功能，大于 0.0 代表开启此功能，默认：1000.0。
- 删除 rocksdb.auto-tuned 配置项，通过rocksdb.rate-limiter-auto-tuned 替代。
- 删除 raftstore.sync-log 配置项，默认会写入数据强制落盘，之前显式关闭 raftstore.sync-log，成功升级 v5.0 版本后，会强制改为 true。
- gc.enable-compaction-filter 配置项的默认值由 false 改成 true。
- enable-cross-table-merge 配置项的默认值由 false 改成 true。
- rate-limiter-auto-tuned 配置项的默认值由 false 改成 true。

16.17.7.1.3 其他

- 升级前，请检查 TiDB 配置项 feedback-probability 的值。如果不为 0，升级后会触发 “panic in the recoverable goroutine” 报错，但不影响升级。
- 为了避免造成数据正确性问题，列类型变更不再允许 VARCHAR 类型和 CHAR 类型的互相转换。

16.17.7.2 新功能

16.17.7.2.1 SQL

List 分区表 (List Partition)（实验特性）

用户文档

采用 List 分区表后，你可以高效地查询、维护有大量数据的表。

List 分区表会按照 PARTITION BY LIST(expr)PARTITION part_name VALUES IN (...) 表达式来定义分区，定义如何将数据划分到不同的分区中。分区表的数据集合最多支持 1024 个值，值的类型只支持整数型，不能有重复的值。可通过 PARTITION ... VALUES IN (...) 子句对值进行定义。

你可以设置 session 变量tidb_enable_list_partition 的值为 ON，开启 List 分区表功能。

List COLUMNS 分区表 (List COLUMNS Partition)（实验特性）

用户文档

List COLUMNS 分区表是 List 分区表的变体，主要的区别是分区键可以由多个列组成，列的类型不再局限于整数类型，也可以是字符串、DATE 和 DATETIME 等类型。

你可以设置 session 变量tidb_enable_list_partition 的值为 ON，开启 List COLUMNS 分区表功能。

不可见索引 (Invisible Indexes)

用户文档，#9246

DBA 调试和选择相对最优的索引时，可以通过 SQL 语句将某个索引设置成 Visible 或者 Invisible，避免执行消耗资源较多的操作，如 DROP INDEX 或 ADD INDEX。

DBA 通过 ALTER INDEX 语句可以修改某个索引的可见性。修改后，查询优化器会根据索引的可见性决定是否将此索引加入到索引列表中。

EXCEPT 和 INTERSECT 操作符

用户文档，#18031

INTERSECT 操作符是一个集合操作符，返回两个或者多个查询结果集的交集。一定程度上可以替代 Inner Join 操作符。

EXCEPT 操作符是一个集合操作符，返回两个查询结果集的差集，即在第一个查询结果中存在但在第二个查询结果中不存在的结果集。

16.17.7.2.2 事务

#18005

悲观事务模式下，如果事务所涉及到的表存在并发的 DDL 操作或者 SCHEMA VERSION 变更，系统自动将该事务的 SCHEMA VERSION 更新到最新版本，以此确保事务会提交成功，避免事务因并发的 DDL 操作或者 SCHEMA VERSION 变更而中断时客户端收到 Information schema is changed 的错误信息。

系统默认关闭此功能，你可以通过修改 tidb_enable_amend_pessimistic_txn 系统变量开启此功能，此功能从 4.0.7 版本开始提供，5.0 版本主要修复了以下问题：

- TiDB Binlog 在执行 Add column 操作的兼容性问题
- 与唯一索引一起使用时存在的数据不一致性的问题
- 与添加索引一起使用时存在的数据不一致性的问题

当前此功能存在以下不兼容性问题：

- 并发事务场景下事务的语义可能发生变化的问题
- 与 TiDB Binlog 一起使用时，存在已知的兼容性问题 #20996
- 与 change column 功能不兼容 #21470

16.17.7.2.3 字符集和排序规则

- 支持 utf8mb4_unicode_ci 和 utf8_unicode_ci 排序规则。用户文档，#17596
- 支持字符集比较排序时不区分大小写。

16.17.7.2.4 安全

用户文档，#18566

为满足各种安全合规（如《通用数据保护条例》(GDPR)）的要求，系统在输出错误信息和日志信息时，支持对敏感信息（例如，身份证信息、信用卡号）进行脱敏处理，避免敏感信息泄露。

TiDB 支持对输出的日志信息进行脱敏处理，你可以通过以下开关开启此功能：

- 全局系统变量tidb_redact_log：默认值为 0，即关闭脱敏。设置变量值为 1 开启 tidb-server 的日志脱敏功能。
- 配置项 security.redact-info-log：默认值为 false，即关闭脱敏。设置配置项值为 true 开启 tikv-server 的日志脱敏功能。#2852
- 配置项 security.redact-info-log：默认值为 false，即关闭脱敏。设置配置项值为 true 开启 pd-server 的日志脱敏功能。
- 配置项 security.redact_info_log（对于 tiflash-server）和配置项 security.redact-info-log（对于 tiflash-learner）：两个配置项的默认值均为 false，即关闭脱敏。设置配置项值为 true 开启 tiflash-server 及 tiflash-learner 的日志脱敏功能。

此功能从 5.0 版本中开始提供，使用过程中必须开启以上所有系统变量及配置项。

16.17.7.3 性能优化

16.17.7.3.1 MPP 架构

用户文档

TiDB 通过 TiFlash 节点引入了 MPP 架构。这使得大型表连接类查询可以由不同 TiFlash 节点分担共同完成。

当 MPP 模式开启后，TiDB 会通过代价决策是否应该交由 MPP 框架进行计算。MPP 模式下，表连接将通过对 JOIN Key 进行数据计算时重分布（Exchange 操作）的方式把计算压力分摊到各个 TiFlash 执行节点，从而达到加速计算的目的。更进一步，加上之前 TiFlash 已经支持的聚合计算，MPP 模式下 TiDB 可以将一个查询的计算都下推到 TiFlash MPP 集群，从而借助分布式环境加速整个执行过程，大幅度提升分析查询速度。

经过 Benchmark 测试，在 TPC-H 100 的规模下，TiFlash MPP 提供了显著超越 Greenplum，Apache Spark 等传统分析数据库或数据湖上分析引擎的速度。借助这套架构，用户可以直接针对最新的交易数据进行大规模分析查询，且性能超越传统离线分析方案。经测试，TiDB 5.0 在同等资源下，MPP 引擎的总体性能是 Greenplum 6.15.0 与 Apache Spark 3.1.1 两到三倍之间，部分查询可达 8 倍性能差异。

当前 MPP 模式不支持的主要功能如下（详细信息请参阅用户文档）：

- 分区表
- Window Function

- Collation
- 部分内置函数
- 读取 TiKV 数据
- OOM Spill
- Union
- Full Outer Join

16.17.7.3.2 聚簇索引

用户文档，#4841

DBA、数据库应用开发者在设计表结构时或者分析业务数据的行为时，如果发现有部分列经常分组排序、返回某范围的数据、返回少量不同的值的数据、有主键列及业务数据不会有读写热点时，建议选择聚簇索引。

聚簇索引 (Clustered Index)，部分数据库管理系统也叫索引组织表，是一种和表的数据相关联的存储结构。创建聚簇索引时可指定包含表中的一列或多列作为索引的键值。这些键存储在一个结构中，使 TiDB 能够快速有效地找到与键值相关联的行，提升查询和写入数据的性能。

开启聚簇索引功能后，TiDB 性能在一些场景下会有较大幅度的提升。例如，TPC-C tpmC 的性能提升了 39%。聚簇索引主要在以下场景会有性能提升：

- 插入数据时会减少一次从网络写入索引数据。
- 等值条件查询仅涉及主键时会减少一次从网络读取数据。
- 范围条件查询仅涉及主键时会减少多次从网络读取数据。
- 等值或范围条件查询涉及主键的前缀时会减少多次从网络读取数据。

每张表既可以采用聚簇索引排序存储数据，也可以采用非聚簇索引，两者区别如下：

- 创建聚簇索引时，可指定包含表中的一列或多列作为索引的键值，聚簇索引根据键值对表的数据进行排序和存储，每张表只能有一个聚簇索引，当某张表有聚簇索引时，该表称为聚簇索引表。相反如果该表没有聚簇索引，称为非聚簇索引表。
- 创建非聚簇索引时，表中的数据存储在无序结构中，用户无需显式指定非聚簇索引的键值，系统会自动为每一行数据分配唯一的 ROWID，标识一行数据的位置信息，查询数据时会用 ROWID 定位一行数据。查询或者写入数据时至少会有两次网络 I/O，因此查询或者写入数据的性能相比聚簇索引会有所下降。

当修改表的数据时，数据库系统会自动维护聚簇索引和非聚簇索引，用户无需参与。

系统默认采用非聚簇索引，用户可以通过以下两种方式选择使用聚簇索引或非聚簇索引：

- 创建表时在语句上指定 `CLUSTERED | NONCLUSTERED`，指定后系统将按照指定的方式创建表。具体语法如下：

```
CREATE TABLE `t` (`a` VARCHAR(255), `b` INT, PRIMARY KEY (`a`, `b`) CLUSTERED);
```

或者：

```
CREATE TABLE `t` (`a` VARCHAR(255) PRIMARY KEY CLUSTERED, `b` INT);
```

通过 `SHOW INDEX FROM tbl-name` 语句可查询表是否有聚簇索引。

- 设置 tidb_enable_clustered_index 控制聚簇索引功能，取值：ON|OFF|INT_ONLY
  - ON：开启聚簇索引，支持添加或者删除非聚簇索引。
  - OFF：关闭聚簇索引，支持添加或者删除非聚簇索引。
  - INT_ONLY：默认值，行为与 5.0 以下版本保持一致，与 alter-primary-key = false 一起使用可控制 INT 类型是否开启聚簇索引。

优先级方面，建表时有指定 CLUSTERED | NONCLUSTERED 时，优先级高于系统变量和配置项。

推荐创建表时在语句上指定 CLUSTERED | NONCLUSTERED 的方式使用聚簇索引和非聚簇索引，此方式对业务更加灵活，业务可以根据需求在同一个系统同时使用所有数据类型的聚簇索引和非聚簇索引。

不推荐使用 tidb_enable_clustered_index = INT_ONLY，原因是 INT_ONLY 是满足兼容性而临时设置的值，不推荐使用，未来会废弃。

聚簇索引功能有如下限制：

- 不支持聚簇索引和非聚簇索引相互转换。
- 不支持删除聚簇索引。
- 不支持通过 ALTER TABLE SQL 语句增加、删除、修改聚簇索引。
- 不支持重组织和重建聚簇索引。
- 不支持启用、禁用索引，也就是不可见索引功能对聚簇索引不生效。
- 不支持 UNIQUE KEY 作为聚簇索引。
- 不支持与 TiDB Binlog 一起使用。开启 TiDB Binlog 后 TiDB 只允许创建单个整数列作为主键的聚簇索引；已创建的聚簇索引表的数据插入、删除和更新动作不会通过 TiDB Binlog 同步到下游。
- 不支持与 SHARD_ROW_ID_BITS 和 PRE_SPLIT_REGIONS 属性一起使用。
- 集群升级回滚时，存量的表不受影响，新增表可以通过导入、导出数据的方式降级。

16.17.7.3.3 异步提交事务 (Async Commit)

用户文档，#8316

数据库的客户端会同步等待数据库系统通过两阶段 (2PC) 完成事务的提交，事务在第一阶段提交成功后就会返回结果给客户端，系统会在后台异步执行第二阶段提交操作，降低事务提交的延迟。如果事务的写入只涉及一个 Region，则第二阶段可以直接被省略，变成一阶段提交。

开启异步提交事务特性后，在硬件、配置完全相同的情况下，Sysbench 设置 64 线程测试 Update index 时，平均延迟由 12.04 ms 降低到 7.01ms ，降低了 41.7%。

开启异步提交事务特性时，数据库应用开发人员可以考虑将事务的一致性从线性一致性降低到因果一致性，减少 1 次网络交互降低延迟，提升数据写入的性能。开启因果一致性的 SQL 语句为 START TRANSACTION WITH CAUSAL CONSISTENCY。

开启因果一致性后，在硬件和配置完全相同的情况下，Sysbench 设置 64 线程测试 oltp_write_only 时，平均延迟由 11.86ms 降低到 11.19ms，降低了 5.6%。

事务的一致性从线性一致性降低到因果一致性后，如果应用程序中多个事务之间没有相互依赖关系时，事务没有全局一致的顺序。

新创建的 5.0 集群默认开启异步提交事务功能。

从旧版本升级到 5.0 的集群，默认不开启该功能，你可以执行 `set global tidb_enable_async_commit = ON;` 和 `set global tidb_enable_1pc = ON;` 语句开启该功能。

异步提交事务功能有如下限制：

- 不支持直接降级

16.17.7.3.4 默认开启 Coprocessor cache 功能

用户文档

5.0 GA 默认开启 Coprocessor cache 功能。开启此功能后，TiDB 会在 tidb-server 中缓存算子下推到 tikv-server 计算后的结果，降低读取数据的延时。

要关闭 Coprocessor cache 功能，你可以修改 `tikv-client.copr-cache` 的 `capacity-mb` 配置项为 0.0。

16.17.7.3.5 提升 `delete from table where id < ? limit ?` 语句执行的性能

#18028

`delete from table where id < ? limit ?` 语句执行的 p99 性能提升了 4 倍。

16.17.7.3.6 优化 load base 切分策略，解决部分小表热点读场景数据无法切分的性能问题

16.17.7.4 稳定性提升

16.17.7.4.1 优化因调度功能不完善引起的性能抖动问题

#18005

TiDB 调度过程中会占用 I/O、网络、CPU、内存等资源，若不对调度的任务进行控制，QPS 和延时会因为资源被抢占而出现性能抖动问题。

通过以下几项的优化，测试 8 小时，TPC-C 测试中 tpm-C 抖动标准差的值小于等于 2%。

引入新的调度算分公式，减少不必要的调度，减少因调度引起的性能抖动问题

当节点的总容量总是在系统设置的水位线附近波动或者 `store-limit` 配置项设置过大时，为满足容量负载的设计，系统会频繁地将 Region 调度到其他节点，甚至还会调度到原来的节点，调度过程中会占用 I/O、网络、CPU、内存等资源，引起性能抖动问题，但这类调度其实意义不大。

为缓解此问题，PD 引入了一套新的调度算分公式并默认开启，可通过 `region-score-formula-version = v1` 配置项切换回之前的调度算分公式。

默认开启跨表合并 Region 功能

用户文档

在 5.0 之前，TiDB 默认关闭跨表合并 Region 的功能。从 5.0 起，TiDB 默认开启跨表合并 Region 功能，减少空 Region 的数量，降低系统的网络、内存、CPU 的开销。你可以通过修改 `schedule.enable-cross-table-merge` 配置项关闭此功能。

默认开启自动调整 Compaction 压缩的速度，平衡后台任务与前端的数据读写对 I/O 资源的争抢

用户文档

在 5.0 之前，为了平衡后台任务与前端的数据读写对 I/O 资源的争抢，自动调整 Compaction 的速度这个功能默认是关闭的；从 5.0 起，TiDB 默认开启此功能并优化调整算法，开启之后延迟抖动比未开启此功能时的抖动大幅减少。

你可以通过修改 `rate-limiter-auto-tuned` 配置项关闭此功能。

默认开启 GC in Compaction filter 功能，减少 GC 对 CPU、I/O 资源的占用

用户文档，#18009

TiDB 在进行垃圾回收和数据 Compaction 时，分区会占用 CPU、I/O 资源，系统执行这两个任务过程中存在数据重叠。

GC Compaction Filter 特性将这两个任务合并在同一个任务中完成，减少对 CPU、I/O 资源的占用。系统默认开启此功能，你可以通过设置 `gc.enable-compaction-filter = false` 关闭此功能。

TiFlash 限制压缩或整理数据占用 I/O 资源（实验特性）

该特性能缓解后台任务与前端的数据读写对 I/O 资源的争抢。

系统默认关闭该特性，你可以通过 `bg_task_io_rate_limit` 配置项开启限制压缩或整理数据 I/O 资源。

增强检查调度约束的性能，提升大集群中修复不健康 Region 的性能

16.17.7.4.2 保证执行计划在最大程度保持不变，避免性能抖动

用户文档

SQL BINDING 支持 `INSERT`、`REPLACE`、`UPDATE`、`DELETE` 语句

在数据库性能调优或者运维过程中，如果发现因为执行计划不稳定导致系统性能不稳定时，你可以根据自身的经验或者通过 `EXPLAIN ANALYZE` 测试选择一条人为优化过的 SQL 语句，通过 SQL BINDING 将优化过的 SQL 语句与业务代码执行的 SQL 语句绑定，确保性能的稳定性。

通过 SQL BINDING 语句手动的绑定 SQL 语句时，你需要确保优化过的 SQL 语句的语法与原来 SQL 语句的语法保持一致。

你可以通过 `SHOW {GLOBAL | SESSION} BINDINGS` 命令查看手工、系统自动绑定的执行计划信息。输出信息基本跟 5.0 之前的版本保持一致。

自动捕获、绑定执行计划

在升级 TiDB 时，为避免性能抖动问题，你可以开启自动捕获并绑定执行计划的功能，由系统自动捕获并绑定最近一次执行计划然后存储在系统表中。升级完成后，你可以通过 `SHOW GLOBAL BINDINGS` 导出绑定的执行计划，自行分析并决策是否要删除绑定的执行计划。

系统默认关闭自动捕获并绑定执行计划的功能，你可以通过修改 Server 或者设置全局系统变量 `tidb_capture_plan_baselines = ON` 开启此功能。开启此功能后，系统每隔 `bind-info-lease`（默认 3 秒）从 Statement Summary 抓取出现过至少 2 次的 SQL 语句并自动捕获、绑定。

16.17.7.4.3 TiFlash 查询稳定性提升

新增系统变量`tidb_allow_fallback_to_tikv`，用于决定在 TiFlash 查询失败时，自动将查询回退到 TiKV 尝试执行，默认为 OFF。

16.17.7.4.4 TiCDC 稳定性提升，缓解同步过多增量变更数据的 OOM 问题

用户文档，#1150

自 v4.0.9 版本起，TiCDC 引入变更数据本地排序功能 Unified Sorter。在 5.0 版本，默认开启此功能以缓解类似场景下的 OOM 问题：

- 场景一：TiCDC 数据订阅任务暂停中断时间长，其间积累了大量的增量更新数据需要同步。
- 场景二：从较早的时间点启动数据订阅任务，业务写入量大，积累了大量的更新数据需要同步。

Unified Sorter 整合了老版本提供的 memory、 file sort-engine 配置选择，不需要用户手动配置变更的运维操作。

限制与约束：

- 用户需要根据业务数据更新量提供充足的磁盘空间，推荐使用大于 128G 的 SSD 磁盘。

16.17.7.5 高可用和容灾

16.17.7.5.1 提升 Region 成员变更时的可用性

用户文档，#18079，#7587，#2860

Region 在完成成员变更时，由于“添加”和“删除”成员操作分成两步，如果两步操作之间有故障发生会引起 Region 不可用并且会返回前端业务的错误信息。

TiDB 引入的 Raft Joint Consensus 算法将成员变更操作中的“添加”和“删除”合并为一个操作，并发送给所有成员，提升了 Region 成员变更时的可用性。在变更过程中，Region 处于中间的状态，如果任何被修改的成员失败，系统仍然可以使用。

系统默认开启此功能，你可以通过 `pd-ctl config set enable-joint-consensus` 命令设置选项值为 false 关闭此功能。

16.17.7.5.2 优化内存管理模块，降低系统 OOM 的风险

跟踪统计聚合函数的内存使用情况，系统默认开启该功能，开启后带有聚合函数的 SQL 语句在执行时，如果当前查询内存总的使用量超过 `mem-quota-query` 阈值时，系统自动采用 `oom-action` 定义的相应操作。

16.17.7.5.3 提升系统在发生网络分区时的可用性

16.17.7.6 数据迁移

16.17.7.6.1 从 S3/Aurora 数据迁移到 TiDB

数据迁移类工具支持 Amazon S3（也包含支持 S3 协议的其他存储服务）作为数据迁移的中间转存介质，同时支持将 Aurora 快照数据直接初始化 TiDB 中，丰富了数据从 Amazon S3/Aurora 迁移到 TiDB 的选择。

该功能使用方法可以参照以下文档：

- 将 MySQL/Aurora 数据导出到 Amazon S3，#8
- 从 Amazon S3 将 Aurora Snapshot 数据初始化到 TiDB，#266

16.17.7.6.2 TiDB Cloud 数据导入性能优化

数据导入工具 TiDB Lightning 针对 TiDB Cloud AWS T1.standard 配置（及其等同配置）的 TiDB 集群进行了数据导入性能优化，测试结果显式使用 TiDB Lightning 导入 1TB TPC-C 数据到 TiDB，性能提升了 40%，由 254 GiB/h 提升到了 366 GiB/h。

16.17.7.7 TiDB 数据共享订阅

16.17.7.7.1 TiCDC 集成第三方生态 Kafka Connect (Confluent Platform)（实验特性）

用户文档，#660

为满足将 TiDB 的数据流转到其他系统以支持相关的业务需求，该功能可以把 TiDB 数据流转到 Kafka、Hadoop、Oracle 等系统。

Confluent 平台提供的 kafka connectors 协议支持向不同协议关系型或非关系型数据库传输数据，在社区被广泛使用。TiDB 通过 TiCDC 集成到 Confluent 平台的 Kafka Connect，扩展了 TiDB 数据流转到其他异构数据库或者系统的能力。

16.17.7.8 问题诊断

用户文档

在排查 SQL 语句性能问题时，需要详细的信息来判断引起性能问题的原因。5.0 版本之前，EXPLAIN 收集的信息不够完善，DBA 只能通过日志信息、监控信息或者盲猜的方式来判断问题的原因，效率比较低。

5.0 版本中，通过以下几项优化提升排查问题的效率：

- 支持对所有 DML 语句使用 EXPLAIN ANALYZE 语句以查看实际的执行计划及各个算子的执行详情。#18056
- 支持对正在执行的 SQL 语句使用 EXPLAIN FOR CONNECTION 语句以查看实时执行状态，例如各个算子的执行时间、已处理的数据行数等。#18233
- EXPLAIN ANALYZE 语句显示的算子执行详情中新增算子发送的 RPC 请求数、处理锁冲突耗时、网络延迟、RocksDB 已删除数据的扫描量、RocksDB 缓存命中情况等。#18663
- 慢查询日志中自动记录 SQL 语句执行时的详细执行状态，输出的信息与 EXPLAIN ANALYZE 语句输出信息保持一致，例如各个算子消耗的时间、处理数据行数、发送的 RPC 请求数等。#15009

16.17.7.9 部署及运维

16.17.7.9.1 优化集群部署操作逻辑，帮助 DBA 更快地部署一套标准的 TiDB 生产集群

用户文档

DBA 在使用 TiUP 部署 TiDB 集群过程发现环境初始化比较复杂、校验配置过多，集群拓扑文件比较难编辑等问题，导致 DBA 的部署效率比较低。5.0 版本通过以下几个事项提升 DBA 部署 TiDB 的效率：

- TiUP Cluster 支持使用 `check topo.yaml` 命令，进行更全面一键式环境检查并给出修复建议。
- TiUP Cluster 支持使用 `check topo.yaml --apply` 命令，自动修复检查过程中发现的环境问题。
- TiUP Cluster 支持 `template` 命令，获取集群拓扑模板文件，供 DBA 编辑且支持修改全局的节点参数。

- TiUP 支持使用 `edit-config` 命令编辑 `remote_config` 参数配置远程 Prometheus。
- TiUP 支持使用 `edit-config` 命令编辑 `external_alertmanagers` 参数配置不同的 AlertManager。
- 在 tiup-cluster 中使用 `edit-config` 子命令编辑拓扑文件时允许改变配置项值的数据类型。

16.17.7.9.2 提升升级稳定性

TiUP v1.4.0 版本以前，DBA 使用 tiup-cluster 升级 TiDB 集群时会导致 SQL 响应持续长时间抖动，PD 在线滚动升级期间集群 QPS 抖动时间维持在 10~30s。

TiUP v1.4.0 版本调整了逻辑，优化如下：

- 升级 PD 时，会主动判断被重启的 PD 节点状态，确认就绪后再滚动升级下一个 PD 节点。
- 主动识别 PD 角色，先升级 follower 角色 PD 节点，最后再升级 PD Leader 节点。

16.17.7.9.3 优化升级时长

TiUP v1.4.0 版本以前，DBA 使用 tiup-cluster 升级 TiDB 集群时，对于节点数比较多的集群，整个升级的时间会持续很长，不能满足部分有升级时间窗口要求的用户。

从 v1.4.0 版本起，TiUP 进行了以下几处优化：

- 新版本 TiUP 支持使用 `tiup cluster upgrade --offline` 子命令实现快速的离线升级。
- 对于使用滚动升级的用户，新版本 TiUP 默认会加速升级期间 Region Leader 的搬迁速度以减少滚动升级 TiKV 消耗的时间。
- 运行滚动升级前使用 `check` 子命令，对 Region 监控状态的检查，确保集群升级前状态正常以减少升级失败的概率。

16.17.7.9.4 支持断点功能

TiUP v1.4.0 版本以前，DBA 使用 tiup-cluster 升级 TiDB 集群时，如果命令执行中断，那么整个升级操作都需重新开始。

新版本 TiUP 支持使用 tiup-cluster `replay` 子命令从断点处重试失败的操作，以避免升级中断后所有操作重新执行。

16.17.7.9.5 增强运维功能

新版本 TiUP 进一步强化了 TiDB 集群运维的功能：

- 支持对已停机的 TiDB 和 DM 集群进行升级或 patch 操作，以适应更多用户的使用场景。
- 为 tiup-cluster 的 `display` 子命令添加 `--version` 参数用于获取集群版本。
- 在被缩容的节点中仅包含 Prometheus 时不执行更新监控配置的操作，以避免因 Prometheus 节点不存在而缩容失败
- 在使用 TiUP 命令输入结果不正确时将用户输入的内容添加到错误信息中，以便用户更快定位问题原因。

16.17.7.10 遥测

TiDB 在遥测中新增收集集群的使用指标，包括数据表数量、查询次数、新特性是否启用等。

若要了解所收集的信息详情及如何禁用该行为，请参见遥测文档。

16.17.8 TiDB 5.0 RC Release Notes

发版日期：2021 年 1 月 12 日

TiDB 版本：5.0.0-rc

TiDB 5.0.0-rc 版本是 5.0 版本的前序版本。在 5.0 版本中，我们专注于帮助企业基于 TiDB 数据库快速构建应用程序，使企业在构建过程中无需担心数据库的性能、性能抖动、安全、高可用、容灾、SQL 语句的性能问题排查等问题。

在 TiDB 5.0 版本中，你可以获得以下关键特性：

- 开启聚簇索引功能，提升数据库的性能。例如：TPC-C tpmC 测试下的性能提升了 39%。
- 开启异步提交事务功能，降低写入数据的延迟。例如：Sysbench oltp-insert 测试中延迟降低了 37.3%。
- 通过提升优化器的稳定性及限制系统任务对 I/O、网络、CPU、内存等资源的占用，降低系统的抖动。例如：长期测试 72 小时，衡量 Sysbench TPS 抖动标准差的值从 11.09% 降低到 3.36%。
- 引入 Raft Joint Consensus 算法，确保 Region 成员变更时系统的可用性。
- 优化 EXPLAIN 功能、引入不可见索引等功能帮助提升 DBA 调试及 SQL 语句的效率。
- 通过备份文件到 AWS S3、Google Cloud GCS 或者从 AWS S3、Google Cloud GCS 恢复到 TiDB，确保企业数据的可靠性。
- 提升从 AWS S3 或者 TiDB/MySQL 导入导出数据的性能，帮忙企业在云上快速构建应用。例如：导入 1TiB TPC-C 数据性能提升了 40%，由 254 GiB/h 提升到 366 GiB/h。

16.17.8.1 SQL

16.17.8.1.1 支持聚簇索引（实验特性）

开启聚簇索引功能后，TiDB 性能在以下条件下会有较大幅度的提升，例如：TPC-C tpmC 的性能提升了 39%。聚簇索引主要在以下条件时会有性能提升：

- 插入数据时会减少一次从网络写入索引数据。
- 等值条件查询仅涉及主键时会减少一次从网络读取数据。
- 范围条件查询仅涉及主键时会减少多次从网络读取数据。
- 等值或范围条件查询涉及主键的前缀时会减少多次从网络读取数据。

聚簇索引定义了数据在表中的物理存储顺序，表的数据只能按照聚簇索引的定义进行排序，每个表只能有一个聚簇索引。

用户可通过修改 `tidb_enable_clustered_index` 变量的方式开启聚簇索引功能。开启后仅在创建新表时生效，适用于主键是多个列或者单个列的非整数类型。如果主键是单列整数类型或者表没有主键，系统会按照原有的方式进行数据排序，不受聚簇索引的影响。

例如，可通过 `select tidb_pk_type from information_schema.tables where table_name = '{tbl_name}'` 语名可查询 `tbl_name` 是否有聚簇索引。

- 用户文档
- 相关 issue：#4841

16.17.8.1.2 支持不可见索引

DBA 调试和选择相对最优的索引时，可以通过 SQL 语句将某个索引设置成 `Visible` 或者 `Invisible`，避免执行消耗资源较多的操作，例如：`DROP INDEX` 或 `ADD INDEX`。

DBA 通过 `ALTER INDEX` 语句来修改某个索引的可见性。修改后优化器会根据索引的可见性决定是否将此索引加入到索引列表中。

- 用户文档
- 相关 issue：#9246

16.17.8.1.3 支持 EXCEPT/INTERSECT 操作符

`INTERSECT` 操作符是一个集合操作符，返回两个或者多个查询结果集的交集。一定程度上可以替代 `Inner Join` 操作符。

`EXCEPT` 操作符是一个集合操作符，将两个查询语句的结果合并在一起，并返回在第一个查询语句中有但在第二个查询句中不存在的结果集。

- 用户文档
- 相关 issue：#18031

16.17.8.2 事务

16.17.8.2.1 提升悲观事务执行成功的概率

悲观事务模式下，如果事务所涉及到的表存在并发 DDL 操作和 `SCHEMA VERSION` 变更，系统会自动将该事务的 `SCHEMA VERSION` 更新到最新版本，确保事务会提交成功，避免事务因 DDL 操作而中断。事务中断时客户端会收到 `Information schema is changed` 的错误信息。

- 相关 issue：#18005

16.17.8.3 字符集和排序规则

使用 `utf8mb4_unicode_ci` 和 `utf8_unicode_ci` 排序规则和字符集比较排序时不区分大小写。

- 用户文档
- 相关 issue：#17596

16.17.8.4 安全

16.17.8.4.1 错误信息和日志信息的脱敏

系统在输出错误信息和日志信息时，支持对敏感信息进行脱敏处理，避免敏感信息泄露。敏感信息可能是身份证信息、信用卡号等。

- 通过 SQL 语句修改 `tidb_redact_log=1` 开启 tidb-server 的错误信息和日志信息脱敏功能

- 通过修改 tikv-server 的 `security.redact-info-log = true` 配置项开启错误信息和日志信息脱敏功能
- 通过修改 pd-server 的 `security.redact-info-log = true` 配置项开启错误信息和日志信息脱敏功能 #2852 #3011
- 通过修改 tiflash-server 的 `security.redact_info_log = true` 以及 tiflash-learner 的 `security.redact-info-log = true` 配置项开启错误信息和日志信息脱敏功能

用户文档

相关 issue：#18566

16.17.8.5 性能提升

16.17.8.5.1 支持异步提交事务（实验特性）

开启异步提交事务可使延迟有较大幅度的降低，例如：Sysbench oltp-insert 测试中开启异步提交事务的延迟与不开启时相比降低了 37.3%。

数据库的客户端会同步等待数据库通过两阶段 (2PC) 完成事务的提交。开启 Async Commit 特性后事务两阶段提交在第一阶段提交成功后就会返回结果给客户端，第二阶段会在后台异步执行。通过事务两阶段异步提交的方式降低事务提交的延迟。

此特性只能显式地修改 `tidb_guarantee_external_consistency = ON` 变量后才能保证事务的外部一致性。开启后性能有较大幅度的下降。

用户可通过修改 `tidb_enable_async_commit = ON` 全局变量开启此功能。

- 用户文档
- 相关 issue：#8316

16.17.8.5.2 提升优化器选择索引的稳定性（实验特性）

优化器若无法长期稳定地选择相对合适的索引，会在很大程度上决定着查询语句的延迟是否有抖动。为确保相同的 SQL 语句不会因为统计信息缺失、不准确等因素导致优化器每次都从多个候选索引选持不同的索引，我们对统计信息模块进行了完善和重构。主要完善如下：

- 扩展统计信息功能，收集多列 NDV、多列顺序依赖性、多列函数依赖性等信息，帮助优化器选择相对较优的索引。
- 重构统计信息模块，帮助优化器选择相对较优的索引。
  - 从 CMSKetch 中删除 TopN 值。
  - 重构 TopN 搜索逻辑。
  - 从直方图中删除 TopN 信息，建立直方图的索引，方便维护 Bucket NDV。

相关 issue：#18065

16.17.8.5.3 优化因调度功能不完善或者 I/O 限流不完善引起的性能抖动问题

TiDB 调度过程中会占用 I/O、Network、CPU、Memory 等资源，若不对调度的任务进行控制，QPS 和延时会因为资源被抢占而出现性能抖动问题。通过以下几项的优化，长期测试 72 小时，衡量 Sysbench TPS 抖动标准差的值从 11.09% 降低到 3.36%。

- 减少节点的容量总是在水位线附近波动引起的调度及 PD 的 `store-limit` 配置项设置过大引起的调度，引入一套新的调度算分公式并通过 `region-score-formula-version = v2` 配置项启用新的调度算分公式 #3269
- 通过修改 `enable-cross-table-merge = true` 开启跨 Region 合并功能，减少空 Region 的数量 #3129
- TiKV 后台压缩数据会占用大量 I/O 资源，系统通过自动调整压缩的速度来平衡后台任务与前端的数据读写对 I/O 资源的争抢，通过 `rate-limiter-auto-tuned` 配置项开启此功能后，延迟抖动比未开启此功能时的抖动大幅减少 #18011
- TiKV 在进行垃圾数据回收和数据压缩时，分区会占用 CPU、I/O 资源，系统执行这两个任务过程中存在数据重叠。GC Compaction Filter 特性将这两个任务合二为一在同一个任务中完成，减 I/O 的占用。此特性为实验性特性，通过 `gc.enable-compaction-filter = true` 开启 #18009
- TiFlash 压缩或者整理数据会占用大量 I/O 资源，系统通过限制压缩或整理数据占用的 I/O 量缓解资源争抢。此特性为实验性特性，通过 `bg_task_io_rate_limit` 配置项开启限制压缩或整理数据 I/O 资源。

相关 issue：#18005

16.17.8.5.4 提升 Real-time BI / Data Warehousing 场景下 TiFlash 的稳定性

- 限制 DeltaIndex 的内存使用量，避免大数据量下内存使用过多导致系统 OOM。
- 限制后台数据整理任务使用的 I/O 写流量，降低对前台任务的影响。
- 新增加线程池，排队处理 coprocessor 任务，避免高并发处理 coprocessor 时内存占用过多导致系统 OOM。

16.17.8.5.5 其他性能优化

- 提升 `delete from table where id < ?` 语句执行的性能，p99 性能提升了 4 倍 #18028
- TiFlash 支持同时向本地多块磁盘并发读、写数据，充分利用本地多块磁盘并发的读、写数据的能力，提升性能

16.17.8.6 高可用和容灾

16.17.8.6.1 提升 Region 成员变更时的可用性（实验特性）

Region 在完成成员变更时，由于“添加”和“删除”成员操作分成两步，如果此时有故障发生会引起 Region 不可用并且会返回前端业务的错误信息。引入的 Raft Joint Consensus 算法，可提升 Region 成员变更时的可用性，将成员变更操作中的“添加”和“删除”合并为一个操作，并发送给所有成员。在变更过程中，Region 处于中间的状态，如果任何被修改的成员失败，系统仍然可以使用。用户可通过 `pd-ctl config set enable-joint-consensus true` 修改成员变量的方式开启此功能。

- 用户文档
- 相关 issue：#18079 #7587 #2860

16.17.8.6.2 优化内存管理模块，降低系统内存溢出的风险

- 减少缓存统计信息的内存消耗。
- 减少使用 Dumpling 工具导出数据时的内存消耗。
- 通过将数据加密码的中间结果存储到磁盘，减少内存消耗。

16.17.8.7 备份与恢复

- BR 支持将数据备份到 AWS S3、 Google Cloud GCS（用户文档）
- BR 支持从 AWS S3、 Google Cloud GCS 恢复数据到 TiDB（用户文档）
- 相关 issue：#89

16.17.8.8 数据的导入和导出

- TiDB Lightning 支持从 AWS S3 将 Aurora Snapshot 数据导入 TiDB（相关 issue：#266）
- 使用 TiDB Lightning 在 DBaaS T1.standard 中导入 1TiB TPCC 数据，性能提升了 40%，由 254 GiB/h 提升到 366 GiB/h
- Dumpling 支持将 TiDB/MySQL 数据导出到 AWS S3（实验特性）（相关 issue：#8，用户文档）

16.17.8.9 问题诊断

16.17.8.9.1 优化 EXPLAIN 功能，收集更多的信息，方便 DBA 排查性能问题

DBA 在排查 SQL 语句性能问题时，需要比较详细的信息来判断引起性能问题的原因。之前版本中 EXPLAIN 收集的信息不够完善，DBA 只能通过日志信息、监控信息或者盲猜的方式来判断问题的原因，效率比较低。此版本通过以下几项优化事项提升排查问题效率：

- 支持对所有 DML 语句使用 EXPLAIN ANALYZE 语句以查看实际的执行计划及各个算子的执行详情 #18056
- 支持对正在执行的 SQL 语句使用 EXPLAIN FOR CONNECTION 语句以查看实时执行状态，如各个算子的执行时间、已处理的数据行数等 #18233
- EXPLAIN ANALYZE 语句显示的算子执行详情中新增算子发送的 RPC 请求数、处理锁冲突耗时、网络延迟、RocksDB 已删除数据的扫描量、RocksDB 缓存命中情况等 #18663
- 慢查询日志中自动记录 SQL 语句执行时的详细执行状态，输出的信息与 EXPLAIN ANALYZE 语句输出信息保持一致，例如各个算子消耗的时间、处理数据行数、发送的 RPC 请求数等 #15009

用户文档

16.17.8.10 部署及运维

- TiUP 支持将 TiDB Ansible 的配置信息导入到 TiUP。以前导入 Ansible 集群的时候 TiUP 会将用户的配置放在 `ansible-imported-configs` 目录下面。用户后续修改配置执行 `tiup cluster edit-config` 时，配置编辑界面中不显示导入的配置，会给用户造成困扰。现在导入 TiDB Ansible 配置信息的时候 TiUP 不仅会放一份到 ansible-imported-configs 目录下面，还会导入到 `tiup cluster edit` 的配置编辑界面，这样用户以后编辑集群配置时就能够看到导入的配置了。

- 增强 TiUP `mirror`命令的功能，支持将多个镜像合并成一个，支持在本地镜像发布组件，支持添加组件所有者到本地镜像 #814
  - 金融行业或者大型企业生产环境的变更是一项非常严肃的事情，若每个版本都采用光盘安装一次，用户使用起来不是很方便。TiUP 提升 `merge` 命令将多个安装包合并成一个，方便 DBA 安装部署。
  - 在 v4.0 中，用户发布自建的镜像时需要启动 tiup-server，使用起来不是很方便。在 v5.0 中，执行 `tiup mirror set` 将当前镜像设置成本地的镜像就可以方便发布自建镜像。

16.18 v4.0

16.18.1 TiDB 4.0.16 Release Notes

发版日期：2021 年 12 月 17 日

TiDB 版本：4.0.16

16.18.1.1 兼容性更改

- TiKV
  - 在 v4.0.16 以前，当把一个非法的 UTF-8 字符串转换为 Real 类型时会直接报错。自 v4.0.16 起，TiDB 会依照该字符串中的合法 UTF-8 前缀进行转换 #11466
- Tools
  - TiCDC
    * 将 Kafka Sink `max-message-bytes` 的默认值改为 1 MB，防止 TiCDC 发送过大消息到 Kafka 集群 #2962
    * 将 Kafka Sink `partition-num` 的默认值改为 3，使 TiCDC 更加平均地分发消息到各个 Kafka partition #3337

16.18.1.2 提升改进

- TiDB
  - 升级 Grafana 到 v7.5.11，规避老版本的安全漏洞
- TiKV
  - 当使用 Backup & Restore 恢复数据或使用 TiDB Lightning 的 Local-backend 导入数据时，采用 zstd 算法压缩 SST 文件，从而减小磁盘使用空间 #11469
- Tools
  - Backup & Restore (BR)
    * 增强恢复的鲁棒性 #27421
  - TiCDC
    * 为 EtcdWorker 添加 tick 频率限制，防止 PD 的 etcd 写入次数过于频繁影响 PD 服务 #3112
    * 优化 TiKV 重新加载时的速率限制控制，缓解 changefeed 初始化时 gPRC 的拥堵问题 #3110

16.18.1.3 Bug 修复

- TiDB
  - 修复在统计信息模块的估算代价中，当执行 range 转 points 时由于数值溢出而导致的查询崩溃 #23625
  - 修复当 ENUM 类型作为 IF 或 CASE WHEN 等控制函数的参数时，返回结果不正确的问题 #23114
  - 修复由于 tidb_enable_vectorized_expression 设置的值不同（on 或 off）导致 GREATEST 函数返回结果不一致的问题 #29434
  - 修复 Index Join 在使用前缀索引时某些情况下崩溃的问题 #24547
  - 修复在某些情况下 Planner 可能缓存无效 join 计划的问题 #28087
  - 修复当 sql_mode 为空时，TiDB 无法插入 null 到非 null 列的问题 #11648
  - 修正 GREATEST 和 LEAST 函数的返回值类型错误 #29019
  - 修复 grant 和 revoke 操作在授予和撤销全局权限时，报 privilege check fail 错误的问题 #29675
  - 修复当 CASE WHEN 函数和 ENUM 类型一起使用时的崩溃问题 #29357
  - 修复 microsecond 函数的向量化表达式版本结果不正确的问题 #29244
  - 修复 hour 函数在向量化表达式中执行结果错误的问题 #28643
  - 修复乐观事务冲突可能导致事务相互阻塞的问题 #11148
  - 修复 auto analyze 输出的日志信息不完整的问题 #29188
  - 修复当 SQL_MODE 为 'NO_ZERO_IN_DATE' 时，使用非法的默认时间不报错的问题 #26766
  - 修复 Grafana 上 Coprocessor Cache 监控面板不显示数据的问题。现在 Grafana 会显示 hits/miss/evict 的数据 #26338
  - 修复并发 truncate 同一个分区会导致 DDL 语句执行卡住的问题 #26229
  - 修复将 Decimal 转为 String 时长度信息错误的问题 #29417
  - 修复使用 NATUAL JOIN 连接多张表时，查询结果中多出一列的问题 #29481
  - 修复 IndexScan 使用前缀索引时，TopN 被错误下推至 indexPlan 的问题 #29711
  - 修复在 DOUBLE 类型的自增列上重试事务会导致数据错误的问题 #29892
- TiKV
  - 修复在极端情况下同时进行 Region Merge、ConfChange 和 Snapshot 时，TiKV 会出现 Panic 的问题 #11475
  - 修复 Decimal 除法计算的结果为 0 时符号为负的问题 #29586
  - 修复 TiKV 监控项中实例级别 gRPC 的平均延迟时间不准确的问题 #11299
  - 修复在缺失下游数据库时出现 TiCDC Panic 的问题 #11123
  - 修复因 channel 打满而导致的 Raft 断连的问题 #11047
  - 修复由于无法在 Max/Min 函数中正确识别 Int64 是否为有符号整数，导致 Max/Min 函数的计算结果不正确的问题 #10158
  - 修复因 Congest 错误而导致的 CDC 频繁增加 scan 重试的问题 #11082
- PD
  - 修复 TiKV 节点缩容后可能导致 Panic 的问题 #4344
  - 修复因 Region syncer 卡住而导致 leader 选举慢的问题 #3936
  - 允许 Evict Leader 调度器调度拥有不健康副本的 Region #4093
- TiFlash
  - 修复 TiFlash 在部分平台上由于缺失 nsl 库而无法启动的问题

- Tools
  - TiDB Binlog
    * 修复传输事务超过 1 GB 时 Drainer 会退出的问题 #28659
  - TiCDC
    * 修复监控 checkpoint lag 出现负值的问题 #3010
    * 修复在容器环境中 OOM 的问题 #1798
    * 修复在多个 TiKV 崩溃或强制重启时可能遇到复制中断的问题 #3288
    * 修复执行 DDL 后的内存泄漏的问题 #3174
    * 修复当发生 ErrGCTTLExceeded 错误时，changefeed 不快速失败的问题 #3111
    * 修复当上游 TiDB 实例意外退出时，TiCDC 同步任务推进可能停滞的问题 #3061
    * 修复当 TiKV 向同一 Region 发送重复请求时，TiCDC 进程 Panic 的问题 #2386
    * 修复 TiCDC 产生的 Kafka 消息体积不受 `max-message-size` 约束的问题 #2962
    * 修复 tikv_cdc_min_resolved_ts_no_change_for_1m 监控在没有 changefeed 的情况下持续更新的问题 #11017
    * 修复当写入 Kafka 消息发生错误时，TiCDC 同步任务推进可能停滞的问题 #2978
    * 修复当开启 `force-replicate` 时，可能某些没有有效索引的分区表被忽略的问题 #2834
    * 修复在创建新的 changefeed 时可能发生的内存泄漏问题 #2389
    * 修复可能因为 Sink 组件提前推进 resolved ts 导致数据不一致的问题 #3503
    * 修复当扫描存量数据耗时过长时，可能由于 TiKV 进行 GC 而导致存量数据扫描失败的问题 #2470
    * 修复 changefeed 更新命令无法识别全局命令行参数的问题 #2803

16.18.2 TiDB 4.0.15 Release Notes

发版日期：2021 年 9 月 27 日

TiDB 版本：4.0.15

16.18.2.1 兼容性更改

- TiDB
  - 修复在新会话中执行 `SHOW VARIABLES` 速度较慢的问题。该修复回退了 #21045 中的部分更改，可能会引起兼容性问题。#24326
  - 以下 Bug 修复涉及执行结果变化，可能引起兼容性变化：
    * 修复了 `greatest(datetime)union null` 返回空字符串的问题 #26532
    * 修复了 `having` 可能执行错误的问题 #26496
    * 修复了当 `between` 表达式两边的 collation 不一致会导致查询结果错误的问题 #27146
    * 修复了 `extract` 函数的参数是负数时查询结果错误的问题 #27236
    * 修复了当 `group_concat` 函数包含非 `bin` 的 collation 时查询结果错误的问题 #27429
    * 修复将 `Apply` 算子转为 `Join` 时漏掉列信息的问题 #27233
    * 修复将非法字符串转为 `DATE` 类型时的非预期行为 #26762
    * 修复开启 New Collation 时多列的 `count distinct` 返回结果错误的问题 #27091

16.18.2.2 功能增强

- TiKV
  - 支持动态修改 TiCDC 配置 #10645

16.18.2.3 提升改进

- TiDB
  - 基于直方图的 row count 来触发 auto-analyze #24237
- TiKV
  - 分离处理读写的 ready 状态以减少读延迟 #10475
  - TiKV Coprocessor 慢日志只考虑处理请求所花费的时间 #10841
  - 当 slogger 线程过载且队列已满时，删除日志而不是阻塞线程 #10841
  - 减少 Resolved TS 消息的大小以节省网络带宽 #2448
- PD
  - 提升了 PD 之间同步 Region 信息的性能 #3932
- Tools
  - Backup & Restore (BR)
    * 并发执行分裂和打散 Region 的操作，提升恢复速度 #1363
    * 遇到 PD 请求错误或 TiKV I/O 超时错误时重试 BR 任务 #27787
    * 恢复大量小表时减少空 Region 的产生，避免影响恢复后的集群运行 #1374
    * 创建表的时候自动执行 `rebase auto id` 操作，省去了单独的 `rebase auto id` DDL 操作，加快恢复速度 #1424
  - Dumpling
    * 获取表信息前过滤掉不需要导出的数据库，提升 `SHOW TABLE STATUS` 的过滤效率 #337
    * 使用 `SHOW FULL TABLES` 来获取需要导出的表，因为 `SHOW TABLE STATUS` 在某些 MySQL 版本上运行存在问题 #322
    * 支持对 MySQL 兼容的特定数据库进行备份，这些数据库不支持 `START TRANSACTION ... WITH CONSISTENT SNAPSHOT` 和 `SHOW CREATE TABLE` 语法 #309
    * 完善 Dumpling 的警告日志，避免让用户误以为导出失败 #340
  - TiDB Lightning
    * 支持导入数据到带有表达式索引或带有基于虚拟生成列的索引的表中 #1404
  - TiCDC
    * TiCDC 总是内部从 TiKV 拉取 old value，提升 TiCDC 易用性 #2397
    * 当某张表的 Region 从某个 TiKV 节点全部迁移走时，减少 goroutine 资源的使用 #2284
    * 在高并发下减少 workerpool 中创建的 goroutine 数量 #2211
    * 异步执行 DDL 语句，不阻塞其他 changefeed #2295

    * 为所有 KV 客户端创建全局共享的 gRPC 连接池 #2531
    * 遇到无法恢复的 DML 错误立即退出，不进行重试 #1724
    * 优化 Unified Sorter 使用内存排序时的内存管理 #2553
    * 为执行 DDL 语句新增 Prometheus 监控指标 #2595 #2669
    * 禁止使用不同的 major 和 minor 版本启动 TiCDC 节点 #2601
    * 移除 `file sorter` 文件排序器 #2325
    * 清理被删 changefeed 的监控数据和已退出处理节点的监控数据 #2156
    * 优化 Region 初始化后的清锁算法 #2188

16.18.2.4 Bug 修复

- TiDB
    - 修复构建 range 时未正确给二进制字面值设置排序规则的问题 #23672
    - 修复当查询包含 GROUP BY 和 UNION 时报错 “index out of range” 的问题 #26553
    - 修复当 TiKV 有 tombstone store 时 TiDB 发送请求失败的问题 #23676 #24648
    - 移除文档中未记录的 `/debug/sub-optimal-plan` HTTP API #27264
    - 修复 `case when` 表达式的字符集不正确的问题 #26662
- TiKV
    - 修复数据恢复期间启用 TDE 时 BR 报告文件已存在错误的问题 #1179
    - 修复损坏的快照文件可能会造成磁盘空间无法回收的问题 #10813
    - 修复 TiKV 过于频繁删除陈旧 Region 的问题 #10680
    - 修复 TiKV 频繁重新连接 PD 客户端的问题 #9690
    - 从加密文件字典中检查陈旧的文件信息 #9115
- PD
    - 修复 PD 未能及时修复 Down Peer 副本的问题 #4077
    - 修复了 PD 在扩容 TiKV 时可能会 Panic 的问题 #3868
- TiFlash
    - 修复多盘部署时数据不一致的潜在问题
    - 修复当查询过滤条件包含诸如 CONSTANT、<、<=、>、>= 或 COLUMN 时出现错误结果的问题
    - 修复写入压力大时 metrics 中 store size 不准确的问题
    - 修复 TiFlash 多盘部署时无法恢复数据的潜在问题
    - 修复 TiFlash 长时间运行后无法回收 Delta 历史数据的潜在问题
- Tools
    - Backup & Restore (BR)
        * 修复了备份和恢复中平均速度计算不准确的问题 #1405
    - TiCDC
        * 修复集成测试中遇到由于 DDL Job 重复导致的 `ErrSchemaStorageTableMiss` 错误 #2422
        * 修复遇到 `ErrGCTTLExceeded` 错误时 changefeed 无法被删除的问题 #2391

* 修复 `capture list` 命令输出中出现已过期 capture 的问题 #2388
* 修复 TiCDC processor 出现死锁的问题 #2017
* 修复重新调度一张表时多个处理器将数据写入同一张表引发的数据不一致的问题 #2230
* 修复元数据管理出现 `EtcdWorker` 快照隔离被破坏的问题 #2557
* 修复因为 DDL sink 错误导致 changefeed 不能被停止的问题 #2552
* 修复一个 TiCDC Open Protocol 的问题：当一个事务中没有任何数据写入时候，TiCDC 产生一个空消息 #2612
* 修复 TiCDC 在处理无符号 `TINYINT` 类型时崩溃的问题 #2648
* 减小 gRPC 窗口来避免 Region 数量过多时触发内存溢出 #2202
* 修复因 TiCDC capture 过多 Regions 出现的 OOM 问题 #2673
* 修复将 `mysql.TypeString, mysql.TypeVarString, mysql.TypeVarchar` 等类型的数据编码为 JSON 时进程崩溃的问题 #2758
* 修复在创建新的 changefeed 时可能发生的内存泄漏问题 #2389
* 修复同步任务从一个表结构变更的 finish TS 开始时 DDL 处理失败的问题 #2603
* 修复 owner 在执行 DDL 语句时崩溃可能导致 DDL 任务丢失的问题 #1260
* 修复 `SinkManager` 中对 map 的不安全并发访问 #2298

16.18.3 TiDB 4.0.14 Release Notes

发版日期：2021 年 7 月 27 日

TiDB 版本：4.0.14

16.18.3.1 兼容性更改

- TiDB
  - 在 v4.0 中将 `tidb_multi_statement_mode` 的默认值从 `WARN` 更改为 `OFF`。建议使用客户端库的多语句功能，参考`tidb_multi_statement_mode` 文档。#25749
  - 将 Grafana 从 v6.1.16 升级到 v7.5.7 以解决两个安全漏洞，参考 Grafana 博文。
  - 将系统变量 `tidb_stmt_summary_max_stmt_count` 的默认值从 200 修改为 3000 #25872
- TiKV
  - 将 `merge-check-tick-interval` 配置项的默认值从 10 修改为 2 以加快 Region 合并的速度 #9676

16.18.3.2 功能增强

- TiKV
  - 添加监控项 `pending` 用以监控 pending PD 心跳，帮助定位 PD 线程变慢的问题 #10008
  - 支持 virtual-host 风格的地址来让 BR 兼容类 S3 储存 #10242
- TiDB Dashboard
  - 新增 OIDC SSO 支持。通过设置兼容 OIDC 标准的 SSO 服务（例如 Okta、Auth0 等），用户可以在不输入 SQL 密码的情况下登录 TiDB Dashboard #960
  - 新增 Debug API 界面用于高级调试，通过该界面可以替代命令行方式来调用 TiDB 和 PD 的内部调试性 API #927

16.18.3.3 改进提升

- TiDB
  - 对于 `point get` 或 `batch point get` 算子，在唯一索引写入过程中，将悲观锁 LOCK 记录转化为 PUT 记录 #26223
  - 支持 MySQL 的系统变量 `init_connect` 及其相关功能 #26031
  - 支持稳定结果模式，使查询结果更稳定 #26003
  - 支持将函数 `json_unquote()` 下推到 TiKV #25721
  - 使 SQL 计划管理 (SPM) 不受字符集的影响 #23295
- TiKV
  - 关闭 TiKV 时，优先关闭 status server 来确保客户端可以正确检测关闭状态 #10504
  - 响应过期副本的消息，以确保过期副本被更快清除 #10400
  - 限制 TiCDC sink 的内存消耗 #10147
  - 当 Region 太大时，使用均匀分裂来加快分裂速度 #10275
- PD
  - 减少各调度器在同时工作时产生的冲突 #3854
- TiDB Dashboard
  - 更新 TiDB Dashboard 版本至 v2021.07.17.1 #3882
  - 支持将当前会话分享为只读的会话，禁止对分享的会话进行修改操作 #960
- Tools
  - Backup & Restore (BR)
    - 恢复数据时合并小文件以提升恢复速度 #655
  - Dumpling
    - 上游是 TiDB v3.x 集群时，使用 `_tidb_rowid` 来切分表以减少 TiDB 的内存使用 #306
  - TiCDC
    - 优化连接 PD 时缺少证书情况下的报错提示 #1973
    - 优化 sorter I/O 报错信息 #1976
    - 在 KV client 中新增 Region 增量扫描的并发度上限，减小 TiKV 的压力 #1926
    - 新增表内存使用量的监控项 #1884
    - 新增 TiCDC 服务端配置项 `capture-session-ttl` #2169

16.18.3.4 Bug 修复

- TiDB
  - 修复当连接一个带 `WHERE` 条件的子查询（值为 `false`）时 `SELECT` 的结果与 MySQL 不兼容的问题 #24865

  - 修复当参数是 ENUM 或 SET 类型时 `ifnull` 函数计算错误的问题 #24944
  - 修复某些情况下错误的聚合函数消除 #25202
  - 修复 Merge Join 运算中当列为 SET 类型时可能产生错误结果的问题 #25669
  - 修复 Cartesian Join 运算返回错误结果的问题 #25591
  - 修复 `SELECT ... FOR UPDATE` 语句进行连接运算且连接使用分区表时，可能产生异常退出情况的问题 #20028
  - 修复缓存的 `prepared` 计划被错误用于 `point get` 的问题 #24741
  - 修复 `LOAD DATA` 语句可以不正常导入非 utf8 数据的问题 #25979
  - 修复通过 HTTP API 访问统计信息时，可能导致内存泄露的问题 #24650
  - 修复执行 `ALTER USER` 语句时出现的安全性问题 #25225
  - 修复系统表 `TIKV_REGION_PEERS` 不能正确处理 DOWN 状态的问题 #24879
  - 修复解析 `DateTime` 时不截断非法字符串的问题 #22231
  - 修复 `select into outfile` 语句在列类型是 YEAR 时，可能无法产生结果的问题 #22159
  - 修复 UNION 子查询中出现 NULL 时可能导致查询结果出错的问题 #26532
  - 修复某些情况下投影算子在执行时可能造成 panic 的问题 #26534

- TiKV

  - 修复特定平台上的 duration 计算可能崩溃的问题 #related-issue
  - 修复将 DOUBLE 类型转换为 DOUBLE 的错误函数 #25200
  - 修复使用 async logger 时 panic 日志可能会丢失的问题 #8998
  - 修复开启加密后再次生成同样的 snapshot 会出现 panic 的问题 #9786 #10407
  - 修复 coprocessor 中 `json_unquote()` 函数错误的参数类型 #10176
  - 修复关机期间出现的可疑警告和来自 Raftstore 的非确定性响应 #10353 #10307
  - 修复备份线程泄漏的问题 #10287
  - 修复 Region split 过慢以及进行 Region merge 时，Region split 可能会损坏 metadata 的问题 #8456 #8783
  - 修复特定情况下 Region 心跳会导致 TiKV 不进行 split 的问题 #10111
  - 修复 TiKV 和 TiDB 间 CM Sketch 格式不一致导致统计信息错误问题 #25638
  - 修复 `apply wait duration` 指标的错误统计 #9893
  - 修复使用 Titan 时 `delete_files_in_range` 以后可能会产生 “Missing Blob” 报错的问题 #10232

- PD

  - 修复调度器在执行删除操作后可能再次出现的问题 #2572
  - 修复调度器在临时配置加载完毕前启动可能导致数据争用的问题 #3771
  - 修复打散 Region 操作可能导致 PD panic 的问题 #3761
  - 修复部分 Operator 未被正确设置优先级的问题 #3703
  - 修复从不存在的 Store 上删除 `evict-leader` 调度器时可能导致 PD panic 的问题 #3660
  - 修复了当集群内 Store 非常多时，PD 切换 Leader 慢的问题 #3697

- TiDB Dashboard

  - 修复实例性能分析界面无法获取全部 TiDB 实例信息的问题 #944
  - 修复 SQL 语句分析界面不显示执行 “计划数” 的问题 #939
  - 修复在升级集群后慢查询界面可能显示 “unknown field” 错误的问题 #902

- TiFlash

  - 修复编译 DAG 请求时出现进程崩溃的潜在问题
  - 修复读负载高的情况下进程崩溃的问题
  - 修复因列存中 split 失败导致 TiFlash 不断重启的问题
  - 修复无法删除 Delta 历史数据的潜在问题
  - 修复并发复制共享 Delta 索引导致结果错误的问题
  - 修复当数据缺失时 TiFlash 无法重启的问题
  - 修复旧的 dm 文件无法被自动清理的问题
  - 修复 SUBSTRING 函数包含特殊参数时引起进程崩溃的潜在问题
  - 修复将 INT 类型转换为 TIME 类型时产生错误结果的问题

- Tools

  - Backup & Restore (BR)

    * 修复不能恢复 mysql 库内的用户表的问题 #1142

  - TiDB Lightning

    * 修复 TiDB Lightning 解析 Parquet 文件中 DECIMAL 类型数据失败的问题 #1276
    * 修复 TiDB Lightning 导入大文件拆分时遇到的 EOF 报错问题 #1133
    * 修复 TiDB Lightning 导入含 auto_increment 的 DOUBLE 或 FLOAT 类型列的表时生成极大 base 值的问题 #1185
    * 修复在生成超过 4 GB 的 KV 数据时可能发生的 panic 问题 #1128

  - Dumpling

    * 使用 Dumpling 导出至 S3 存储时，不再要求 s3:ListBucket 权限覆盖整个 Bucket，只需要覆盖导出的前缀即可 #898

  - TiCDC

    * 修复分区表新增分区后的处理 #2205
    * 修复 TiCDC 无法读取 /proc/meminfo 导致崩溃的问题 #2023
    * 减少 TiCDC 运行时的内存使用 #2011 #1957
    * 修复 MySQL sink 遇到错误或暂停时，MySQL 连接会泄漏的问题 #1945
    * 修复当 start TS 小于 current TS 减去 GC TTL 时无法创建 TiCDC changefeed 的问题 #1839
    * 减少 sort heap 的内存 malloc，以降低 CPU 开销 #1853
    * 修复调度数据表时可能发生的同步终止问题 #1827

16.18.4 TiDB 4.0.13 Release Notes

发版日期：2021 年 5 月 28 日

TiDB 版本：4.0.13

16.18.4.1 新功能

- TiDB

  - 支持将列属性 AUTO_INCREMENT 变更为 AUTO_RANDOM #24608
  - 引入 infoschema.client_errors_summary 表，用以追踪返回给客户端的错误 #23267

16.18.4.2 提升改进

- TiDB
  - 当内存中的统计信息缓存是最新的时，避免后台作业频繁读取 `mysql.stats_histograms` 表造成高 CPU 使用率 #24352
- TiKV
  - 提高 `store used size` 计算过程的准确性 #9904
  - 在 EpochNotMatch 消息中返回更多的 Region 以降低 Region miss 的发生 #9731
  - 加快释放长期运行集群中堆积的内存 #10035
- PD
  - 优化 TSO 处理时间的统计指标，帮助用户判断 PD 侧的 TSO 处理时间是否过长 #3524
  - 更新 Dashboard 的版本至 v2021.03.12.1 #3469
- TiFlash
  - 自动清除过期历史数据以释放磁盘空间
- Tools
  - Backup & Restore (BR)
    * 支持备份恢复系统库 `mysql` 下的用户表 #1077
    * 更新 `checkVersion` 以检查集群版本和备份数据版本 #1090
    * 容忍在备份期间集群中出现少数 TiKV 节点宕机 #1062
  - TiCDC
    * 为内部处理单元增加流程控制，避免出现内存溢出问题 #1751
    * 增加 Unified Sorter 清理陈旧临时文件的功能，禁止多个 `cdc` 服务共享 `sort-dir` 目录 #1741
    * 给 Failpoint 增加 HTTP 接口调用 #1732

16.18.4.3 Bug 修复

- TiDB
  - 修复带有子查询的 UPDATE 语句更新生成列时会 panic 的问题 #24658
  - 修复使用多列索引读取数据时返回结果重复的问题 #24634
  - 修复在 DIV 表达式中使用 BIT 类型常量作为除数造成查询结果错误的问题 #24266
  - 修复 NO_ZERO_IN_DATE SQL 模式对 DDL 语句中设置的列默认值无效的问题 #24185
  - 修复 BIT 类型列与整型列进行 UNION 并集运算时，查询结果出错的问题 #24026
  - 修复 BINARY 类型与 CHAR 类型比较时，错误地生成了 TableDual 执行计划的问题 #23917
  - 修复 `insert ignore on duplicate` 非预期的删除表记录的问题 #23825
  - 修复 Audit 插件导致 TiDB panic 的问题 #23819
  - 修复 HashJoin 算子未正确处理排序规则的问题 #23812
  - 修复悲观事务中，`batch_point_get` 处理异常值出错导致连接断开的问题 #23778

  - 修复 `tidb_row_format_version` 配置项的值被设置为 1，且 `enable_new_collation` 的值被设置为 `true` 时，数据索引不一致的问题 #23772
  - 修复整型列与字符串类型常量比较时，查询结果出错的问题 #23705
  - 修复 `approx_percent` 函数中传入 `BIT` 类型列时出错的问题 #23702
  - 修复执行 TiFlash 批量请求时，TiDB 误报 `TiKV server timeout` 的问题 #23700
  - 修复 `IndexJoin` 在前缀列索引上计算结果出错的问题 #23691
  - 修复由于 `BINARY` 类型列上排序规则处理不当导致查询结果出错的问题 #23598
  - 修复当 `UPDATE` 语句中存在含 `HAVING` 子句的连接查询时，执行崩溃的问题 #23575
  - 修复比较类型表达式中使用 `NULL` 常量导致 TiFlash 计算结果出错的问题 #23474
  - 修复 `YEAR` 类型列与字符类型常量比较结果出错的问题 #23335
  - 修复 `session.group_concat_max_len` 被设置得过小时，`group_concat` 执行崩溃的问题 #23257
  - 修复 `TIME` 类型列上使用 `BETWEEN` 表达式计算结果出错的问题 #23233
  - 修复 `DELETE` 语句中出现的权限检查问题 #23215
  - 修复往 `DECIMAL` 列中插入非法字符串时不报错的问题 #23196
  - 修复往 `DECIMAL` 类型列插入数据时解析出错的问题 #23152
  - 修复 `USE_INDEX_MERGE` hint 无法生效的问题 #22924
  - 修复使用 `ENUM` 或 `SET` 类型列作为 `WHERE` 过滤条件时，查询结果出错的问题 #22814
  - 修复 Clustered Index 与 New Collation 同时使用时，查询结果出错的问题 #21408
  - 修复 `enable_new_collation` 开启时，`ANALYZE` 出错的问题 #21299
  - 修复视图处理默认 `ROLE` 时，未正确处理相关 `DEFINER` 的问题 #24531
  - 修复取消 DDL Job 时卡住的问题 #24445
  - 修复 `concat` 函数错误处理排序规则的问题 #24300
  - 修复当 `SELECT` 域中包含 `IN` 子查询且子查询外侧表含有空值元组时，查询结果出错的问题 #24022
  - 修复逆序扫表时，TiFlash 被优化器错误选用的问题 #23974
  - 修复点查的返回结果中，列名与 MySQL 不一致的问题 #23970
  - 修复在数据库名含有大写字母的库中执行 `show table status` 结果为空的问题 #23958
  - 修复不同时拥有 `INSERT` 及 `DELETE` 权限的用户可以执行 `REPLACE` 操作的问题 #23938
  - 修复由于未正确处理排序规则导致的 `concat/make_set/insert` 表达式计算结果出错的问题 #23878
  - 修复在含有 `RANGE` 分区的表上查询时，查询崩溃的问题 #23689
  - 修复如下问题：在旧版本的集群中，若 `tidb_enable_table_partition` 被设置为 `false`，含有分区的表会被当作普通表处理。此时由旧版本升级至新版本时，在该表上执行 `batch point get` 查询会导致连接崩溃 #23682
  - 修复配置了 TiDB 监听 TCP 连接及 UNIX 套接字时，TCP 连接中远程主机未被正确验证的问题 #23513
  - 修复由于非默认的排序规则导致查询结果出错的问题 #22923
  - 修复 Grafana 的 Coprocessor Cache 面板不显示数据的问题 #22617
  - 修复优化器访问统计信息缓存时出错的问题 #22565

- TiKV

  - 修复因磁盘写满后 `file_dict` 写入不完全导致 TiKV 无法重启的问题 #9963
  - 限制 TiCDC 默认的扫描速度为 128MB/s #9983
  - 减少 TiCDC 进行初次扫描的内存使用量 #10133
  - 为 TiCDC 扫描的速度添加背压 (back pressure) 功能 #10142
  - 通过避免不必要的读取来获取 TiCDC 旧值以解决潜在的 OOM 问题 #10031
  - 修复了由于读取旧值而导致的 TiCDC OOM 问题 #10197
  - 为 S3 存储添加超时机制以避免 S3 客户端没有任何响应地挂起 #10132

- TiFlash
  - 修复未向 Prometheus 报告 `delta-merge-tasks` 数量的问题
  - 修复 `Segment Split` 期间发生进程崩溃的问题
  - 修复 Grafana 中 `Region write Duration` 面板位置错误的问题
  - 修复了存储引擎无法删除数据的潜在问题
  - 修复 `TIME` 类型转换为 `INT` 类型时产生错误结果的问题
  - 修复 `bitwise` 算子和 TiDB 行为不一致的问题
  - 修复字符串转换为 `INT` 时产生错误结果的问题
  - 修复连续快速写入可能导致 TiFlash 内存溢出的问题
  - 修复 Table GC 时会引发空指针的问题
  - 修复向已被删除的表写数据时 TiFlash 进程崩溃的问题
  - 修复当使用 BR 恢复数据时 TiFlash 进程可能崩溃的问题
  - 修复当使用通用 CI 排序规则时字符权重错误的问题
  - 修复被逻辑删除的表可能丢失数据的问题
  - 修复比较包含空字符的字符串时产生错误结果的问题
  - 修复输入列包含空常量时逻辑函数返回错误结果的问题
  - 修复逻辑函数仅接受数字类型输入的问题
  - 修复时间戳值为 `1970-01-01` 且时区偏移为负时计算结果不正确的问题
  - 修复 `Decimal256` 的哈希值计算结果不稳定的问题
- Tools
  - TiCDC
    * 修复当 Sorter 的输入通道卡住时，流控导致的死锁问题 #1779
    * 修复 TiCDC changefeed 断点卡住导致 TiKV GC safe point 不推进的问题 #1756
    * 回滚 `explicit_defaults_for_timestamp` 的改动，确保不用 SUPER 权限也可以同步数据到 MySQL #1749
  - TiDB Lightning
    * 修复在 autocommit 关闭的情况下，TiDB Lightning TiDB-backend 无法导入数据的问题

16.18.5 TiDB 4.0.12 Release Notes

发版日期：2021 年 4 月 2 日

TiDB 版本：4.0.12

16.18.5.1 新功能

- TiFlash
  - 新增工具用于检测当前 `tiflash replica` 的状态

16.18.5.2 改进提升

- TiDB
  - 优化 EXPLAIN 语句在 batch cop 模式下的输出信息 #23164
  - 在 EXPLAIN 语句的输出中，为无法下推到存储层的表达式增加告警信息 #23020
  - 调整 DDL 包中部分 Execute/ExecRestricted 的使用为安全 API (2) #22935
  - 调整 DDL 包中部分 Execute/ExecRestricted 的使用为安全 API (1) #22929
  - 添加 optimization-time 和 wait-TS-time 到慢日志中 #22918
  - 支持从 infoschema.partitions 表中查询 partition_id #22489
  - 添加 last_plan_from_binding 以帮助用户了解 SQL 执行计划是否与 binding 的 hint 相匹配 #21430
  - 支持在没有 pre-split 选项时也能执行 TRUNCATE 表操作 #22872
  - 为 str_to_date 表达式添加三种新的格式限定符 #22812
  - 在 metrics 监控中记录 PREPARE 执行失败的问题为 Failed Query OPM #22672
  - 当设置了 tidb_snapshot 时，不对 PREPARE 语句的执行报错 #22641
- TiKV
  - 消除短时间内大量重连接的现象 #9879
  - 对多 Tombstones 场景下的写操作和 Batch Get 进行优化 #9729
  - 将 leader-transfer-max-log-lag 的默认值改为 128，以增加切换 leader 的成功率 #9605
- PD
  - 只有当 pending-peer 或 down-peer 变更时才更新 Region Cache，减少心跳更新压力 #3471
  - 防止 split-cache 中的 Region 成为合并的目标 #3459
- TiFlash
  - 优化配置文件并删除无用项
  - 减小 TiFlash 二进制文件大小
  - 使用自适应的 GC 策略以减少内存使用
- Tools
  - TiCDC
    - 若任务的暂停同步时间超过 1 天，再次启动该任务时需要二次确认 #1497
    - 为 Old Value 功能添加监控面板 #1571
  - Backup & Restore (BR)
    - 记录 HTTP_PROXY 和 HTTPS_PROXY 环境变量 #827
    - 提升多表场景下的备份性能 #745
    - 在 service safe point 检查失败时报告错误 #826
    - 在 backupmeta 中记录集群版本和 BR 版本 #803
    - 遇到外部存储错误时，重试备份以便提高备份成功率 #851
    - 减少 BR 备份的内存使用 #886
  - TiDB Lightning
    - 运行 TiDB Lightning 前检查 TiDB 集群版本以防止未知错误 #787

* 在遇到 cancel 错误时及时退出 #867
* 添加 `tikv-importer.engine-mem-cache-size` 和 `tikv-importer.local-writer-mem-cache-size` 参数以便调整内存占用和性能之间的平衡 #866
* Local-backend 并发运行 `batch split region` 以提高导入速度 #868
* 从 S3 存储导入数据时，TiDB Lightning 不再要求 `s3:ListBucket` 权限 #919
* 从 checkpoint 恢复时，TiDB Lightning 会继续使用之前的导入引擎 #924

16.18.5.3 Bug 修复

- TiDB
  - 修复当变量为十六进制字面量时，`get` 表达式出错的问题 #23372
  - 修复生成 Enum 和 Set 类型的快速执行计划时使用了错误 Collation 的问题 #23292
  - 修复 `nullif` 和 `is-null` 表达式一起使用时可能出现结果错误的问题 #23279
  - 修复自动搜集统计信息在规定时间窗口外被触发的问题 #23219
  - 修复 `point-get` 计划中 CAST 函数可能忽略错误的问题 #23211
  - 修复 `CurrentDB` 为空时 SPM 可能不生效的问题 #23209
  - 修复 IndexMerge 执行计划中可能出现错误过滤条件的问题 #23165
  - 修复 `NULL` 常量的返回类型中可能出现 `NotNullFlag` 的问题 #23135
  - 修复 Text 类型可能遗漏处理 Collation 的问题 #23092
  - 修复 Range 分区表处理 `IN` 表达式可能出错的问题 #23074
  - 修复将 TiKV 标记为 Tombstone 后，在相同地址和端口启动不同 StoreID 的新 TiKV 会持续返回 `StoreNotMatch` 的问题 #23071
  - INT 类型为 `NULL` 且和 `YEAR` 进行比较时不进行类型调整 #22844
  - 修复当表含有 `auto_random` 列 load data 时失去连接的问题 #22736
  - 修复取消 DDL 操作 panic 时可能阻塞其他 DDL 操作的问题 #23297
  - 修复进行 `NULL` 和 `YEAR` 比较时可能生成错误 key range 的问题 #23104
  - 修复创建视图成功但是使用时可能失败的问题 #23083
- TiKV
  - 修复 `IN` 表达式没有正确处理有符号和无符号整型数的问题 #9850
  - 修复 Ingest 操作不可重入问题 #9779
  - 修复 TiKV 在处理 JSON 向字符串转换时空格缺失的问题 #9666
- PD
  - 修复在 store 缺失 label 的情况下隔离级别错误的问题 #3474
- TiFlash
  - 修复当 `binary` 列的默认值前后包含 0 字节时查询结果错误的问题
  - 修复当数据库名称中包含特殊字符时无法同步数据的问题
  - 修复 `IN` 表达式中出现 `Decimal` 列时查询结果错误的问题
  - 修复 Grafana 中已打开文件数指标过高的问题
  - 修复当表达式中包含 `Timestamp` 类型时查询结果错误的问题
  - 修复处理 `FROM_UNIXTIME` 表达式时可能发生的无响应的问题

  - 修复字符串转换为整数结果不正确的问题
  - 修复 `like` 表达式可能返回错误结果的问题

- Tools

  - TiCDC

    * 修复 `resolved ts` 时间乱序的问题 #1464
    * 修复由于网络不稳定而导致的表调度出错引发的数据丢失问题 #1508
    * 修复终止 processor 时不能及时释放资源的问题 #1547
    * 修复因事务计数器未正确更新而导致下游数据库链接可能泄露的问题 #1524
    * 修复因 PD 抖动时多个 owner 共存可能导致数据表丢失的问题 #1540

  - Backup & Restore (BR)

    * 修复 `WalkDir` 在目标为 bucket name 的时候无返回值的问题 #733
    * 修复 `status` 端口无 TLS 的问题 #839

  - TiDB Lightning

    * 修复 TiKV Importer 可能忽略文件已存在的错误 #848
    * 修复 TiDB Lightning 可能使用错误的时间戳而读到错误数据的问题 #850
    * 修复 TiDB Lightning 非预期退出时可能造成 checkpoint 文件损坏的问题 #889
    * 修复由于忽略 `cancel` 错误而可能导致的数据错误的问题 #874

16.18.6 TiDB 4.0.11 Release Notes

发版日期：2021 年 2 月 26 日

TiDB 版本：4.0.11

16.18.6.1 新功能

- TiDB

  - 支持 `uft8_unicode_ci` 和 `utf8mb4_unicode_ci` 排序规则 #22558

- TiKV

  - 支持 `utf8mb4_unicode_ci` 排序规则 #9577
  - 支持 `cast_year_as_time` 排序规则 #9299

- TiFlash

  - 增加排队处理 Coprocessor 任务的线程池以降低内存溢出几率，并增加配置项 `cop_pool_size` 和 `batch_cop_pool_size`，默认值为 物理核数 `* 2`

16.18.6.2 改进提升

- TiDB
  - 重排由 `outer join` 简化的 `inner join` 顺序 #22402
  - Grafana 面板支持多集群 #22534
  - 为多语句问题提供替代解决方案 #22468
  - 将慢查询监控区分为 `internal` 和 `general` 两类 #22405
  - 为 `utf8_unicode_ci` 和 `utf8mb4_unicode_ci` 排序规则增加接口 #22099
- TiKV
  - 为 DBaaS 添加 server 信息的监控指标 #9591
  - Grafana dashboards 支持监控多个集群 #9572
  - 汇报 RocksDB 的监控指标到 TiDB #9316
  - 为 Coprocessor 任务记录暂停时间 #9277
  - 为 Load Base Split 添加 key 数量和大小的阀值 #9354
  - 在导入数据前检查文件是否存在 #9544
  - 改进 Fast Tune 面板 #9180
- PD
  - Grafana dashboards 支持监控多个集群 #3398
- TiFlash
  - 优化 `date_format` 函数的性能
  - 优化处理 ingest SST 时的内存开销
  - 优化 Batch Coprocessor 内部的重试逻辑以降低 Region error 的出现概率
- Tools
  - TiCDC
    - 在 `capture` 元信息中添加版本信息和在 `changefeed` 元信息中创建该 `changefeed` 的 CLI 版本 #1342
  - TiDB Lightning
    - 并行创建数据表以提升导入速度 #502
    - 跳过分裂小 Region 以提升导入速度 #524
    - 添加导入进度条并提升恢复进度的精确度 #506

16.18.6.3 Bug 修复

- TiDB
  - 修复异常的 `unicode_ci` 常数传递 #22614
  - 修复可能导致排序规则和 coercibility 错误的问题 #22602
  - 修复可能导致错误排序规则结果的问题 #22599

- 修复不同排序规则的常数替换问题 #22582
- 修复 like 函数使用排序规则时可能返回错误结果的问题 #22531
- 修复 least 和 greatest 函数 duration 类型推导错误问题 #22580
- 修复 like 函数处理 _ 宽字符后加 % 出错的问题 #22575
- 修复比较函数 least 和 greatest 类型推导错误的问题 #22562
- 修复使用 like 函数处理 Unicode 字符串错误的问题 #22529
- 修复点查请求无法取得 @@tidb_snapshot 变量中快照的问题 #22527
- 修复生成多个 join 相关 hint 可能 panic 的问题 #22518
- 修复转换字符串为 BIT 类型不准确的问题 #22420
- 修复插入 tidb_rowid 列时出现的 index out of range 报错问题 #22359
- 修复缓存计划被错误地使用的问题 #22353
- 修复 WEIGHT_STRING 函数处理过长字符串出现 panic 的问题 #22332
- 禁止参数数量不合法时使用生成列 #22174
- 在构造执行计划前正确地设置进程执行信息 #22148
- 修复 IndexLookUp 执行统计不准的问题 #22136
- 容器部署时为内存使用信息增加缓存 #22116
- 修复解码执行计划错误的问题 #22022
- 使用错误的窗口函数说明时提供报错 #21976
- 使用 PREPARE 语句嵌套 EXECUTE、DEALLOCATE 或 PREPARE 时报错 #21972
- 修复使用 INSERT IGNORE 到不存在的分区时不报错的问题 #21971
- 统一 EXPLAIN 和 slow log 中的执行计划编码 #21964
- 修复聚合算子下 join 出现未知列的问题 #21957
- 修复 ceiling 函数中类型推导错误的问题 #21936
- 修复 Double 列忽略精度的问题 #21916
- 修复关联聚合在子查询中被计算的问题 #21877
- 当 JSON 数据长度超过 65536 时提供报错 #21870
- 修复 dyname 函数和 MySQL 不兼容的问题 #21850
- 修复输入数据过长时 to_base64 函数返回 NULL 的问题 #21813
- 修复在子查询中比较多个字段失败的问题 #21808
- 修复 JSON 中比较浮点数的问题 #21785
- 修复 JSON 类型比较的问题 #21718
- 修复 cast 函数的 coercibility 值设置错误的问题 #21714
- 修复使用 IF 函数时可能出现 panic 的问题 #21711
- 修复 JSON 搜索返回 NULL 和 MySQL 不兼容的问题 #21700
- 修复 ORDER BY 和 HAVING 子句检查 only_full_group_by 模式的问题 #21697
- 修复 Day/Time 单位和 MySQL 不兼容的问题 #21676
- 修复 LEAD 和 LAG 函数默认值类型问题 #21665
- LOAD DATA 时执行检测以保证只能往基础表中导入数据 #21638
- 修复 addtime 和 subtime 函数处理非法参数的问题 #21635
- 将近似值的舍入规则更改为 “舍入到最接近的偶数” #21628
- 修复 WEEK() 在被明确读取前无法识别全局变量 default_week_format 的问题 #21623

- TiKV

  - 修复当设置 PROST=1 时构建 TiKV 失败的问题 #9604
  - 修复不匹配的内存诊断信息 #9589

  - 修复在恢复 RawKV 数据时部分 key range 的 end key 的包含性问题 #9583
  - 修复当 TiCDC 增量扫数据时读取一个被回滚的事务的某个 key 的旧值时 TiKV 可能会 panic 的问题 #9569
  - 修复使用不同配置的连接拉取同一个 Region 的变更时旧值配置不匹配的问题 #9565
  - 修复 TiVK 运行在网络接口缺少 MAC 地址的设备上会崩溃的问题（自 v4.0.9 引入）#9516
  - 修复 TiKV 在备份大 Region 时会内存溢出的问题 #9448
  - 修复 `region-split-check-diff` 无法自定义配置的问题 #9530
  - 修复系统时间回退时 TiKV 会 panic 的问题 #9542

- PD

  - 修复成员健康的监控显示不正确的问题 #3368
  - 禁止有副本的不正常 tombstone store 被清除 #3352
  - 修复 store limit 无法持久化的问题 #3403
  - 调整 `scatter range schedler` 的 limit 限制 #3401

- TiFlash

  - 修复 Decimal 类型的 `min/max` 计算结果错误的问题
  - 修复读取数据时有可能导致 crash 的问题
  - 修复 DDL 操作后写入的数据可能会在 compaction 后丢失的问题
  - 修复 Coprocessor 中错误解析 Decimal 常量的问题
  - 修复 Learner Read 过程中可能导致 crash 的问题
  - 修复 TiFlash 中除以 0 或 NULL 的行为与 TiDB 不一致的问题

- Tools

  - TiCDC

    * 修复 TiCDC 服务在同时发生 `ErrTaskStatusNotExists` 和 `capture` 会话关闭的情况下的非预期的退出的问题 #1240
    * 修复 `changefeed` 之间不同 Old Value 设置会互相影响的问题 #1347
    * 修复 TiCDC 服务在遇见错误的 `sort-engine` 参数时卡住的问题 #1309
    * 修复在非 Owner 节点上获取 debug 信息退出的问题 #1349
    * 修复 `ticdc_processor_num_of_tables` 和 `ticdc_processor_table_resolved_ts` 两个监控指标在增删数据表时没有被正确更新的问题 #1351
    * 修复 Processor 在添加同步数据表时退出而造成的潜在的数据丢失问题 #1363
    * 修复 Owner 在数据表迁移期间造成非正常状态的 TiCDC 服务退出的问题 #1352
    * 修复 TiCDC 服务在丢失 service GC safepoint 时没有及时退出的问题 #1367
    * 修复 KV client 可能跳过创建 event feed 的问题 #1336
    * 修复同步事务到下游时事务原子性被破坏的问题 #1375

  - Backup & Restore (BR)

    * 修复恢复备份后 TiKV 可能产生大 Region 的问题 #702
    * 修复在没有 Auto ID 的数据表上恢复 Auto ID 的问题 #720

  - TiDB Lightning

    * 修复使用 TiDB-backend 时可能触发 `column count mismatch` 的问题 #535

    * 修复 TiDB-backend 在导入数据源 column 个数和数据表 column 个数不匹配时非预期退出的问题 #528
    * 修复导入期间 TiKV 可能发生的非预期退出的问题 #554

16.18.7 TiDB 4.0.10 Release Notes

发版日期：2021 年 1 月 15 日

TiDB 版本：4.0.10

16.18.7.1 新功能

- PD
    - 添加了配置项 `enable-redact-log`，可以设置将日志中的用户数据脱敏 #3266
- TiFlash
    - 添加了配置项 `security.redact_info_log`，可以设置将日志中的用户数据脱敏

16.18.7.2 改进提升

- TiDB
    - 添加 `txn-entry-size-limit` 配置项，用于限制事务中单个 key-value 记录的大小 #21843
- PD
    - 优化了 `store-state-filter` 的监控，可以显示更加具体的原因 #3100
    - 更新 `go.etcd.io/bbolt` 依赖至 v1.3.5 #3331
- Tools
    - TiCDC
        * 为 `maxwell` 协议默认开启 old value 特性 #1144
        * 默认启用 unified sorter 特性 #1230
    - Dumpling
        * 支持检查未定义的参数，支持输出导出的进度 #228
    - TiDB Lightning
        * 支持重试读 S3 遇到的错误 #533

16.18.7.3 Bug 修复

- TiDB
  - 修复由于并发导致的 batch client 超时问题 #22336
  - 修复由于并发地自动捕获 SQL 绑定而导致的重复绑定问题 #22295
  - 当日志级别为 'debug' 时，让 SQL 语句绑定的自动捕获正确运行 #22293
  - 当 Region 合并正在发生时，正确地释放锁 #22267
  - 对 Datetime 类型的用户变量返回正确的值 #22143
  - 修复错误使用 Index Merge 访问方式的问题 #22124
  - 修复由于执行计划缓存导致 TiFlash 报 wrong precision 错误的问题 #21960
  - 修复由于 schema 变更导致的错误结果 #21596
  - 避免在 ALTER TABLE 中不必要地更改 column flag #21474
  - 让包含子查询块别名的 optimizer hint 生效 #21380
  - 为 IndexHashJoin 和 IndexMergeJoin 生成正确的 optimizer hint #21020
- TiKV
  - 修复了 peer 和 ready 之间的错误映射 #9409
  - 修复一些日志信息在 security.redact-info-log 设置为 true 时未脱敏的问题 #9314
- PD
  - 修复 ID 分配不是单调递增的问题 #3308 #3323
  - 修复 PD client 在某些情况下可能卡住的问题 #3285
- TiFlash
  - 修复了 TiFlash 解析老版本 TiDB 表结构失败导致 TiFlash 无法启动的问题
  - 修复了在 RedHat 系统中 TiFlash 会对 cpu_time 进行错误处理导致 TiFlash 无法启动的问题
  - 修复了将配置项 path_realtime_mode 设置为 true 时 TiFlash 无法启动的问题
  - 修复了当调用有三个参数的 substr 函数时，返回结果错误的问题
  - 修复了当 TiDB 对 Enum 枚举进行无损修改时，TiFlash 无法读取修改后的值的问题
- Tools
  - TiCDC
    - 修复 maxwell 协议的问题，包括 base64 数据输出和将 TSO 转换成 unix timestamp #1173
    - 修复过期的元数据可能引发新创建的 changefeed 异常的问题 #1184
    - 修复在关闭的 notifier 上创建 receiver 的问题#1199
    - 修复在 etcd 更新缓慢时导致内存访问量增长的问题 #1227
    - 修复 max-batch-size 不生效的问题 #1253
    - 修复清理过期任务信息的问题 #1280
    - 修复 MySQL sink 中由于没有调用 rollback 而导致回收 db conn 卡住的问题 #1285
  - Dumpling
    - 修改默认设置的tidb_mem_quota_query 的行为以避免 TiDB 内存溢出 #233
  - Backup & Restore (BR)

    * 修复 BR v4.0.9 无法恢复 BR v4.0.8 保存在 GCS 上的备份 #688
    * 修复在恢复 GCS 上的备份时可能发生的 panic 问题 #673
    * 默认禁用备份统计信息以避免 BR 内存溢出 #693

  - TiDB Binlog

    * 修复在启用 AMEND TRANSACTION 特性时，Drainer 可能会使用错误 schema 来生成 SQL 语句的问题 #1033

  - TiDB Lightning

    * 修复未正确编码 Region key 而导致分裂 Region 失败问题 #531
    * 修复可能丢失 CREATE TABLE 失败的错误 #530
    * 修复使用 TiDB-backend 时遇到的 `column count mismatch` 问题 #535

16.18.8 TiDB 4.0.9 Release Notes

发版日期：2020 年 12 月 21 日

TiDB 版本：4.0.9

16.18.8.1 兼容性更改

- TiDB

  - 废弃配置文件中 `enable-streaming` 配置项 #21055

- TiKV

  - 减少开启加密时的 I/O 开销和锁冲突。该修改向下不兼容。如果需要降级至 v4.0.9 以下，需要将 `security.encryption.enable-file-dictionary-log` 配置为 `false`，并在降级前重启 #9195

16.18.8.2 新功能

- TiFlash

  - 支持将存储引擎的新数据分布在多个硬盘上，分摊 I/O 压力（实验特性）

- TiDB Dashboard

  - SQL 语句分析功能的列表界面支持显示所有字段并排序 #749
  - 集群拓扑页面支持缩放 #772
  - SQL 语句分析及慢日志页面支持显示 SQL 语句的临时存储占用大小 #777
  - SQL 语句分析及慢日志列表界面支持导出列表数据 #778
  - 支持配置自定义 Prometheus 地址 #808
  - 新增集群实例统计摘要页面 #815
  - 慢日志详情页面新增更多时间字段 #810

16.18.8.3 优化提升

- TiDB
  - 在转换等值条件为其它条件时，通过使用启发式规则，避免生成 (index) merge join 以得到更好的执行计划 #21146
  - 区分用户变量的类型 #21107
  - 在配置文件中添加了 `performance.gogc` 配置项，用于设置 GOGC #20922
  - 提升 `Timestamp` 和 `Datetime` 类型的二进制输出结果与 MySQL 的兼容性 #21135
  - 优化用户使用 `LOCK IN SHARE MODE` SQL 语句时输出的报错信息 #21005
  - 优化在可剪切的表达式进行常量折叠时输出的错误信息，避免输出不必要的警告或错误信息 #21040
  - 优化 `LOAD DATA` 语句执行 `PREPARE` 时的报错信息 #21199
  - 修改整型列的类型时，忽略掉整型字段的零值填充的属性 #20986
  - 在 `EXPLAIN ANALYZE` 结果中输出 DML 语句执行器相关运行时的信息 #21066
  - 禁止在一条语句中对主键做出多次不同的修改 #21113
  - 添加连接空闲时间的监控项 #21301
  - 新增运行 `runtime/trace` 分析工具时系统自动临时开启慢日志的功能 #20578
- TiKV
  - 添加标记以跟踪 `split` 命令的来源 #8936
  - 支持动态修改 `pessimistic-txn.pipelined` 配置项 #9100
  - 减少运行 Backup & Restore 和 TiDB Lightning 时对系统的性能影响 #9098
  - 添加 Ingesting SST 报错的监控项 #9096
  - 阻止 Leader 在任意副本需要复制日志时进入休眠状态 #9093
  - 提高 Pipelined locking 的成功率 #9086
  - 调整配置项 `apply-max-batch-size` 和 `store-max-batch-size` 的默认值为 1024 #9020
  - 添加 `max-background-flushes` 配置 #8947
  - 默认关闭 RocksDB consistency check 以提高性能 #9029
  - 将 Region 大小的查询操作移出 `pd heartbeat worker` 以减轻其压力 #9185
- PD
  - TiKV store 转变为 `Tombstone` 状态时检查 TiKV 集群的版本号，防止用户降级和升级过程中的开启不兼容特性#3213
  - 禁止低版本的 TiKV 强制从 `Tombstone` 状态转为 Up #3206
- TiDB Dashboard
  - 对于 SQL 语句文本点击“展开”后支持保持展开状态 #775
  - 默认在新窗口打开 SQL 语句分析和慢日志详情 #816
  - 改进慢日志页面部分时间字段描述 #817
  - 改进错误信息提示，显示更完整的错误内容 #794
- TiFlash
  - 降低 Replica read 时的延迟

- 优化 TiFlash 的错误信息
- 优化在大数据量下，对缓存数据大小的限制
- 添加正在处理的 Coprocessor 请求数量的 metric

- Tools

  - Backup & Restore (BR)

    * BR 不再接受存在歧义的 `--checksum false`（不会正确关闭 checksum）命令行参数，正确用法为 `--checksum=false` #588
    * 支持暂时性地调整 PD 的参数，在 BR 意外退出后，PD 能自动恢复回正常参数 #596
    * 支持恢复数据表的统计信息 #622
    * 系统自动重试 `read index not ready` 和 `proposal in merging mode` 两种错误 #626

  - TiCDC

    * 添加对 TiKV 开启 Hibernate Region 的告警规则 #1120
    * 优化 schema storage 的内存使用 #1127
    * 增加 Unified Sorter 功能，可以在数据量较大的情况下提升增量扫阶段的同步速度（实验特性）#1122
    * 支持在 TiCDC Open Protocol 中配置单条 Kafka 消息的最大大小和包含的最大行变更数量（仅在 Kafka sink 生效）#1079

  - Dumpling

    * 对导出失败部分的数据进行重试 #182
    * 支持同时设置 `-F` 和 `-r` 两个参数 #177
    * 默认不导出系统表 #194
    * 在设置 `--transactional-consistency` 参数时支持重建 MySQL 链接 #199
    * 支持使用 `-c,--compress` 参数指定 Dumpling 使用的压缩算法，空字符串代表不压缩 #202

  - TiDB Lightning

    * 默认不恢复系统表 #459
    * 支持为 auto-random 的主键设置默认值 #457
    * 完善 Local 模式下分裂 Region 的精度 #422
    * 支持给 `tikv-importer.region-split-size`、`mydumper.read-block-size`、`mydumper.batch-size` 和 `mydumper.max-region-size` 设置可读的参数（比如 “2.5 GiB”）#471

  - TiDB Binlog

    * 在写下游出错时给 Drainer 设置非零退出码 #1012

16.18.8.4 Bug 修复

- TiDB

  - 修复了前缀索引和 OR 条件一起使用时结果不正确的问题 #21287
  - 修复了开启自动重试后可能出现的一处 panic #21285
  - 修复了根据列类型检查分区表定义时出现的一处问题 #21273
  - 修复了分区表对于列的类型检查的一处问题。分区表达式的值的类型和分区列的类型必须一致 #21136

  - 修复了哈希分区表对于分区名唯一性检查的问题 #21257
  - 修复非 INT 类型的值，插入到哈希分区表后结果不正确的问题 #21238
  - 修复了在部分写入类的场景中，使用了 index join 会遇到非预期报错的问题 #21249
  - 修复了在 CASE WHEN 中 BigInt 无符列的值被错误地转换成有符类型的问题 #21236
  - 修复了 index hash join 和 index merge join 没有考虑 collation 的问题 #21219
  - 修复了分区表在建表和查询时，没有考虑 collation 的问题 #21181
  - 修复了慢日志记录的查询结果可能不全的问题 #21211
  - 修复了一处数据库名大小写处理不当，导致的 DELETE 未正确删除数据的问题 #21206
  - 修复了执行 DML 语句导致 schema 的内存被覆盖的问题 #21050
  - 修复了使用 join 时，无法查询到合并后的列的问题 #21021
  - 修复了一些 semi-join 的查询结果不正确的问题 #21019
  - 修复了表锁对于 UPDATE 语句不生效的问题 #21002
  - 修复创建递归的视图出现栈溢出的问题 #21001
  - 修复了 index merge join 在执行外连接的时候，结果不符合预期的问题 #20954
  - 修复了一处事务问题，该场景下应该返回结果未知，但是却返回了执行失败 #20925
  - 修复 explain for connection 无法显示最后一次执行计划的问题 #21315
  - 修复在 Read Committed 隔离级别下，Index Merge 结果不正确的问题 #21253
  - 修复了由于事务写冲突重试导致的 auto-ID 分配失败 #21079
  - 修复了 JSON 数据无法通过 load data 无法正确导入到 TiDB 的问题 #21074
  - 修复新增加 Enum 类型列的默认值问题 #20998
  - 对于日期类型的数学计算，保留原始的数据类型信息，修复 adddate 函数插入非法值的问题 #21176
  - 修复了部分场景错误地生成了 PointGet 的执行计划，导致执行结果不正确 #21244
  - 在 ADD_DATE 函数中忽略夏令时的转换，以和 MySQL 兼容 #20888
  - 修复了插入尾部带有超出 varchar 和 char 长度限制的空白字符的字符串时报错的 bug #21282
  - 修复了对比 int 和 year 类型时没有将 [1, 69] 的整数转换为 [2001, 2060] 以及没有将 [70, 99] 的整数转换为 [1970, 1999] 的兼容性 bug #21283
  - 修复了 sum() 函数计算 Double 类型字段的结果溢出导致 panic 的问题 #21272
  - 修复了 DELETE 语句未能给 unique key 加悲观锁的问题 #20705
  - 修复了快照读能够命中 lock cache，返回错误结果的问题 #21539
  - 修复了在同一个事务中读取大量数据时可能发生的内存泄漏问题 #21129
  - 修复了在子查询中省略表别名时的语法解析错误问题 #20367
  - 修复了查询中 IN 函数的参数为 time 类型时可能返回错误结果的问题 #21290

- TiKV

  - 修复当列个数大于 255 时，下推返回错误结果集的问题 #9131
  - 修复网络隔离时 Region Merge 可能会导致数据丢失的问题 #9108
  - 修复使用 latin1 字符集时，ANALYZE 语句会导致 panic 的问题 #9082
  - 修复类型转换中将数字转成时间会得到错误结果的问题 #9031
  - 修复当开启加密时无法使用 TiDB Lightning 导入数据的问题 #8995
  - 修复使用 0.0.0.0 时 advertise-status-addr 异常的问题 #9036
  - 修复当事务删除 key 时却报 key 已存在的问题 #8930
  - 修复 RocksDB cache 映射错误导致的数据错误问题 #9029
  - 修复当 Leader 切换时 Follower Read 可能返回旧数据的问题 #9240
  - 修复悲观锁下可能读到旧值的问题 #9282
  - 修复 transfer leader 后 replica read 可能会读到旧值的问题 #9240

  - 修复 TiKV 在 profiling 结束后再收到 SIGPROF 会 panic 的问题 #9229

- PD
  - 修复在特殊情况下 Placement Rule 指定的 leader 绑定不生效的问题 #3208
  - 修复 `trace-region-flow` 在配置更新时被置为 `false` 的问题 #3120
  - 修复特殊情况下 safepoint 有无限 TTL 的问题 #3143

- TiDB Dashboard
  - 修复部分时间显示混杂中英文的问题 #755
  - 修复部分不兼容的浏览器中没有提示不兼容的问题 #776
  - 修复部分情况下事务时间戳显示不正确的问题 #793
  - 修复部分 SQL 文本格式化后成为无效 SQL 语句的问题 #805

- TiFlash
  - 修复 `INFORMATION_SCHEMA.CLUSTER_HARDWARE` 中可能包含未被使用的硬盘信息的问题
  - 修复 Delta cache 内存占用量估算偏少的问题
  - 修复由线程统计信息引起的内存泄露问题

- Tools
  - Backup & Restore (BR)
    * 修复因 S3 secret access keys 中存在特殊字符而导致失败的问题 #617
  - TiCDC
    * 修复某些异常情况下存在多个 Owner 的问题 #1104
    * 修复在 TiKV 节点意外退出或重启恢复情况下 TiCDC 不能正常同步的问题，该 bug 在 v4.0.8 引入 #1198
    * 修复在表初始化过程中会向 etcd 中重复写入元数据的问题 #1191
    * 修复 schema storage 缓存 TiDB 表信息的过程中因更新表信息延迟或过早 GC 导致同步中断的问题 #1114
    * 修复 schema storage 在 DDL 频繁的情况下会消耗过多内存的问题 #1127
    * 修复在同步任务暂停或取消之后会产生 goroutine 泄露的问题 #1075
    * 增加 Kafka producer 最大重试时间到 600s，避免在下游 Kafka 服务或网络抖动情况下同步中断 #1118
    * 修复 Kafka 消息所包含行变更数量不能正常生效的问题 #1112
    * 修复当 TiCDC 与 PD 间网络出现抖动，并且同时操作 TiCDC changefeed 暂停和恢复，可能会出现部分表数据没有被同步的问题 #1213
    * 修复 TiCDC 与 PD 网络不稳定情况下 TiCDC 可能出现进程非预期退出的问题 #1218
    * 在 TiCDC 内部使用全局 PD client，以及修复 PD client 被错误关闭导致同步阻塞的问题 #1217
    * 修复 TiCDC owner 节点可能在 etcd watch client 里消耗过多内存的问题 #1224
  - Dumpling
    * 修复在某些情况下 MySQL 链接关闭导致 Dumpling 卡住的问题 #190
  - TiDB Lightning
    * 修复使用错误信息编码 key 的问题 #437
    * 修复 GC life time TTL 不生效的问题 #448
    * 修复手动关闭时可能出现的 panic 问题 #484

16.18.9 TiDB 4.0.8 Release Notes

发版日期：2020 年 10 月 30 日

TiDB 版本：4.0.8

16.18.9.1 新功能

- TiDB
  - 支持聚合函数 APPROX_PERCENTILE #20197
- TiFlash
  - 支持 CAST 函数下推
- Tools
  - TiCDC
    * 支持快照级别一致性复制 #932

16.18.9.2 优化提升

- TiDB
  - 在挑选索引组合计算表达式选择率的贪心算法里优先使用选择率低的索引 #20154
  - 在 Coprocessor 运行状态中记录更多的 RPC 信息 #19264
  - 优化读取慢日志的效率，以提升慢查询性能 #20556
  - 在挑选执行计划时，优化器会在 Plan Binding 阶段等待超时的执行计划以记录更多的 Debug 信息 #20530
  - 在慢查询和慢日志中增加语句的重试时间 #20495 #20494
  - 增加系统表 `table_storage_stats` #20431
  - 为 INSERT/UPDATE/REPLACE 语句记录 RPC 相关的运行时信息 #20430
  - 在 `EXPLAIN FOR CONNECTION` 语句的结果中新增算子信息 #20384
  - 在 TiDB 日志中将客户端的连接建立/断开日志级别调整为 DEBUG #20321
  - 增加 Coprocessor Cache 的监控信息 #20293
  - 在运行时信息中记录更多的悲观锁相关参数 #20199
  - 在运行时信息和 Trace 功能中增加两个新的耗时信息 #20187
  - 在慢日志中添加事务提交的运行时信息 #20185
  - 关闭 Index Merge Join #20599
  - 为临时字符串常量增加 ISO 8601 和时区支持 #20670
- TiKV
  - 添加 Fast-Tune 监控页辅助性能诊断 #8804
  - 添加 `security.redact-info-log` 配置，用于从日志中删除用户数据 #8746
  - 修改 error code 的 metafile 格式 #8877

  - 开启动态修改 pessimistic-txn.pipelined 配置 #8853
  - 默认开启 memory profile 功能 #8801

- PD

  - 生成 error 的 metafile #3090
  - 为 operator 日志添加更多有用信息 #3009

- TiFlash

  - 添加关于 Raft log 的监控
  - 添加关于 cop 任务内存使用的监控
  - 在存在删除数据的情况下使 min/max 索引更加准确
  - 提高小批量数据下的查询性能
  - 添加 error.toml 文件以支持标准错误码

- Tools

  - Backup & Restore (BR)

    * 通过将 split 和 ingest 流水线来加速恢复 #427
    * 支持手动恢复 PD 的调度器 #530
    * 将移除 PD 调度器接口改为暂停调度器 #551

  - TiCDC

    * 在 MySQL sink 中定期输出统计信息 #1023

  - Dumpling

    * 支持直接导出数据到 S3 存储上 #155
    * 支持导出 View 视图 #158
    * 支持导出只包含生成列的数据表 #166

  - TiDB Lightning

    * 支持多字节的 CSV delimiter 和 separator #406
    * 通过禁止一些 PD 调度器加速导入 #408
    * 在 v4.0 集群上使用 GC-TTL 接口来防止 checksum 阶段的 GC 报错 #396

16.18.9.3 Bug 修复

- TiDB

  - 修复使用分区表时，可能遇到非预期 Panic 的问题 #20565
  - 修复外连接时，若外表有过滤条件，Index Merge Join 结果有时不正确的问题 #20427
  - 修复 BIT 类型进行转换时，由于类型长度溢出而错误地返回 NULL 的问题 #20363
  - 修复 ALTER TABLE ... 语法改变 BIT 类型的默认值，可能导致默认值错误的问题 #20340
  - 修复 BIT 类型转换为 INT64 时可能导致长度溢出错误的问题 #20312
  - 修复混合类型的列在进行条件传播优化时，可能导致结果错误的问题 #20297
  - 修复 Plan Cache 在存储过期执行计划时，可能 Panic 的问题 #20246
  - 修复 FROM_UNIXTIME 和 UNION ALL 一起使用时，返回结果会被错误地截断的问题 #20240

  - 修复 Enum 类型在转换为 Float 类型时可能导致错误结果的问题 #20235
  - 修复 RegionStore 在某些条件下会 Panic 的问题 #20210
  - 修复 BatchPointGet 请求对无符号整数的最大值进行排序时，结果错误的问题 #20205
  - 修复 Enum/Set 类型在混合 Collation 的类型判定时，结果 Collation 可能与 MySQL 不兼容的问题 #20364
  - 修复将其他类型的 0 转换为 YEAR 类型时，结果与 MySQL 不兼容的问题 #20292
  - 修复 KV Duration 监控指标中包含 store0 时，上报结果不正确的问题 #20260
  - 修复写入 Float 类型数据时，由于长度溢出提示 out of range 错误后仍然被错误地写入问题 #20252
  - 修复 NOT NULL 属性生成列允许在某些情况下写入 NULL 值的问题 #20216
  - 修复 YEAR 类型数据写入超过允许范围时，错误提示不准确的问题 #20170
  - 修复某些情况下悲观事务重试时，会报错 invalid auto-id 的问题 #20134
  - 修复 ALTER TABLE 更改 Enum/Set 类型时，未进行重复性约束检查的问题 #20046
  - 修复一些算子在并发执行时，记录的 Coprocessor Task 运行时信息错误的问题 #19947
  - 修复只读系统变量无法被作为 Session 级变量显式 SELECT 的问题 #19944
  - 修复重复 ORDER BY 条件有时会导致执行计划选择不是最优的问题 #20333
  - 修复生成 Metric Profile 时，由于字体超过允许的最大值导致失败的问题 #20637

- TiKV

  - 修复加密功能中锁冲突导致 pd-worker 处理心跳慢的问题 #8869
  - 修复错误生成 memory profile 的问题 #8790
  - 修复备份时指定 GCS 储存类别 (storage class) 报错的问题 #8763
  - 修复了重启或者新 Split 的 Learner 节点找不到 Leader 的问题 #8864

- PD

  - 修复了 TiDB Dashboard 在某些场景下引起 PD panic 的错误 #3096
  - 修复了某个 PD store 下线超过十分钟后可能引起 PD panic 的错误 #3069

- TiFlash

  - 修复了日志信息中时间戳错误的问题
  - 修复了使用多盘部署时错误的容量导致创建 TiFlash 副本失败的问题
  - 修复了 TiFlash 重启后可能提示数据文件损坏的问题
  - 修复了 TiFlash 崩溃后磁盘上可能残留损坏文件的问题
  - 修复了在写流量较小情况下，由于 Raft Learner 协议中的状态不能及时更新而导致 wait index duration 变长，造成查询慢的问题

- Tools

  - Backup & Restore (BR)

    * 修复 Restore 期间可能发生的 send on closed channel panic 问题 #559

  - TiCDC

    * 修复 owner 因更新 GC safepoint 失败而非预期退出的问题 #979
    * 修复非预期的任务信息更新 #1017
    * 修复非预期的空 Maxwell 消息 #978

- TiDB Lightning
    * 修复列信息错误的问题 #420
    * 修复 Local 模式下获取 Region 信息出现死循环的问题 #418

16.18.10 TiDB 4.0.7 Release Notes

发版日期：2020 年 9 月 29 日

TiDB 版本：4.0.7

16.18.10.1 新功能

- PD
    - PD 客户端中添加 `GetAllMembers` 函数，用于获取 PD 成员信息 #2980
- TiDB Dashboard
    - 支持生成统计指标关系图 #760

16.18.10.2 优化提升

- TiDB
    - 为 `join` 算子添加更多执行信息 #20093
    - 为 `EXPLAIN ANALYZE` 语句添加协处理器缓存命中率信息 #19972
    - 支持将 ROUND 函数下推至 TiFlash #19967
    - 在 `ANALYZE` 过程中为 `CMSketch` 添加默认值 #19927
    - 重新更改某些日志错误信息以使其脱敏 #20004
    - 支持接受来自 MySQL 8.0 客户端的连接 #19959
- TiKV
    - 支持日志输出为 JSON 格式 #8382
- PD
    - operator 统计计数器增加的时间点，从创建时改为执行完毕时 #2983
    - 将 `make-up-replica` operator 调整为高优先级 #2977
- TiFlash
    - 完善数据读取过程中遇到 Region meta 变更的错误处理
- Tools
    - TiCDC
        * 提升开启 Old Value 后 MySQL Sink 的同步效率 #955

  - Backup & Restore (BR)
    * 增加备份时链接中断重试 #508
  - TiDB Lightning
    * 增加动态设置 log 级别的 HTTP 接口 #393

16.18.10.3 Bug 修复

- TiDB
  - 修复向量化函数 and/or/COALESCE 因为提前计算导致的问题 #20092
  - 修复不同存储类型导致相同的执行计划摘要的问题 #20076
  - 修复错误函数 `!= any()` 的错误表现 #20062
  - 修复当慢日志文件不存在时输出慢日志报错的问题 #20051
  - 修复当上下文取消后 Region 请求不断重试的问题 #20031
  - 修复查询 `cluster_slow_query` 表的时间类型在 streaming 的请求下报错的问题 #19943
  - 修复 DML 语句使用 `case when` 函数时可能导致 schema 改变的问题 #20095
  - 修复 slow log 中 `prev_stmt` 的信息未脱敏的问题 #20048
  - 修复当 tidb-server 不正常退出时没有释放表锁的问题 #20020
  - 修复当插入 ENUM 和 SET 类型的字段产生不正确的错误信息的问题 #19950
  - 修复 `IsTrue` 函数在某些情况下的错误表现 #19903
  - 修复在 PD 扩容或缩容情况下 `CLUSTER_INFO` 系统表可能不正常运行的问题 #20026
  - 修复在控制表达式中某些情况下产生不必要的错误或报警信息 #19910
  - 改变更新统计信息的方式以避免造成 OOM 的情况 #20013
- TiKV
  - 修复 TLS 握手失败后会导致 Status API 不可用的问题 #8649
  - 修复一些平台上可能存在潜在未定义行为的问题 #7782
  - 修复执行 UnsafeDestroyRange 操作时生成快照可能导致 Panic 的问题 #8681
- PD
  - 修复当 `balance-region` 开启时，如果存在 Region 没有 Leader，可能会导致 PD panic 的问题 #2994
  - 修复 Region 合并后 Region 大小和 Region key 数量的统计偏差 #2985
  - 修复热点统计不正确的问题 #2991
  - 修复 `redirectSchedulerDelete` 中未进行 `nil` 判断的问题 #2974
- TiFlash
  - 修正 right outer join 结果错误的问题
- Tools
  - Backup & Restore (BR)
    * 修复了在恢复数据后导致 TiDB 配置变更的错误 #509
  - Dumpling
    * 修复了在某些变量为空的情况下 metadata 解析失败的问题 #150

16.18.11 TiDB 4.0.6 Release Notes

发版日期：2020 年 9 月 15 日

TiDB 版本：4.0.6

16.18.11.1 新功能

- TiFlash
  - 在 TiFlash 中支持在广播 Join 中使用外连接
- TiDB Dashboard
  - 添加 Query 编辑和执行页面（实验性功能）#713
  - 添加 Store 地理拓扑显示页面 #719
  - 添加集群配置调整页面（实验性功能）#733
  - 支持共享当前 session #741
  - 支持显示 SQL 语句分析中执行计划的数量 #746
- Tools
  - TiCDC（自 v4.0.6 起，TiCDC 成为正式功能，可用于生产环境）
    * 支持输出 `maxwell` 格式的数据 #869

16.18.11.2 优化提升

- TiDB
  - 使用标准错误替换 TiDB 中的错误码和错误信息 #19888
  - 提升分区表的写性能 #19649
  - 在 Cop Runtime 统计信息中记录更多的 RPC Runtime 信息 #19264
  - 禁止在 `metrics_schema` 和 `performance_schema` 中创建表 #19792
  - 支持调整 Union 执行算子的并发度 #19886
  - 支持在广播 Join 中使用外连接 #19664
  - 添加对 process list 的 digest #19829
  - 对于自动提交语句的重试转换到悲观锁模式 #19796
  - 在 `Str_to_date` 函数中支持 `%r` 和 `%T` 的数据格式 #19693
  - 使 `SELECT INTO OUTFILE` 需要文件权限 #19577
  - 支持 `stddev_pop` 函数 #19541
  - 添加 `TiDB-Runtime` 面板 #19396
  - 提升 `ALTER TABLE ALGORITHMS` 的兼容性 #19364
  - 在慢日志的计划字段中加入编码好的 `INSERT/DELETED/UPDATE` 计划 #19269
- TiKV
  - 优化 `DropTable/TruncateTable` 时导致的性能下降 #8627

  - 支持生成标准错误码的 meta 文件 #8619
  - scan detail 中增加 tombstone 个数的 metrics #8618
  - 添加 rocksdb perf context 到 Grafana 默认面板 #8467

- PD

  - 升级 Dashboard 到 v2020.09.08.1 #2928
  - 添加更多关于 store 和 Region 心跳的 metrics #2891
  - 回滚空间不足的阈值策略 #2875
  - 支持标准错误码
    * #2918 #2911 #2913 #2915 #2912
    * #2907 #2906 #2903 #2806 #2900 #2902

- TiFlash

  - 在 Grafana 中添加关于数据同步（`apply Region snapshots` 和 `ingest SST files`）的监控面板
  - 在 Grafana 中添加关于 `write stall` 的监控面板
  - 添加 `dt_segment_force_merge_delta_rows` 及 `dt_segment_force_merge_delta_deletes` 用于调整阈值以避免 `write stall` 发生
  - 支持在 TiFlash-Proxy 中把 `raftstore.snap-handle-pool-size` 设为 0 以禁用多线程同步 Region snapshot，可降低同步数据时内存消耗
  - 在 `https_port` 及 `metrics_port` 上支持 CN 检查

- Tools

  - TiCDC
    * 在初始化阶段跳过 resolved lock #910
    * 减少写 PD 的频率 #937
  - Backup & Restore (BR)
    * 在 Summary 中添加真实消耗的时间 #486
  - Dumpling
    * 支持输出带有列名的 `INSERT` 语句 #135
    * 将 `--filesize` 和 `--statement-size` 参数与 mydumper 保持统一 #142
  - TiDB Lightning
    * Split 的 Region 大小更加精确 #369
  - TiDB Binlog
    * 支持以 go time 的格式设置 GC 时间 #996

16.18.11.3 Bug 修复

- TiDB

  - 修复了在 Metric Profile 中 `tikv_cop_wait` time 的一个问题 #19881
  - 修复了 `SHOW GRANTS` 显示错误结果的问题 #19834

- 修复了使用 `!= ALL (subq)` 查询结果不正确的问题 #19831
- 修复了转换 ENUM 和 SET 类型的一个问题 #19778
- 增加了 `SHOW STATS_META`、`SHOW STATS_BUCKET` 的一个权限检查 #19760
- 修复了由 `builtinGreatestStringSig` 和 `builtinLeastStringSig` 引起的列长度不匹配问题 #19758
- 如果向量化计算抛出多余的 errors 或者 warnings，回退向量化执行到标量执行 #19749
- 修复了在相关列类型是 `Bit` 时 `Apply` 算子出现错误的问题 #19692
- 修复了在 MySQL 8.0 客户端中查询 processlist 和 cluster_log 时出现的问题 #19690
- 修复了相同类型的 plan 具有不同类型的 plan digest 的错误 #19684
- 禁止从 `Decimal` to `Int` 变更列类型 #19682
- 修复了 `SELECT ... INTO OUTFILE` 返回运行时错误的问题 #19672
- 修复了 `builtinRealIsFalseSig` 的不正确的实现 #19670
- 修复了分区表达式检查漏掉括号表达式的问题 #19614
- 修复了当在 `HashJoin` 上具有 `Apply` 算子时的查询错误 #19611
- 修复了向量化将 `Real` cast 成 `Time` 类型时的错误的结果 #19594
- 修复了 `SHOW GRANTS` 可以显示不存在用户的 grants 信息的错误 #19588
- 修复了当在 `IndexLookupJoin` 上具有 `Apply` 算子时的查询错误 #19566
- 修复了当在分区表上将 `Apply` 转化成 `HashJoin` 时的错误结果 #19546
- 修复了当在 `Apply` 的 inner 端具有 `IndexLookUp` 算子时的错误结果 #19508
- 修复了使用视图时非预期的 panic #19491
- 修复了 `anti-semi-join` 查询时的不正确结果 #19477
- 修复了删除统计信息时未删除 topN 的统计信息的错误 #19465
- 修复了因错误使用 batch point get 时产生的错误结果 #19460
- 修复了在带有虚拟生成列的 `IndexLookupJoin` 上无法找到列的错误 #19439
- 修复了在 `SELECT` 和 `UPDATE` 查询上的不同计划比较 datum 的错误 #19403
- 修复了 TiFlash 在 Region cache 上产生的 work index 数据争用 #19362
- 修复了 `logarithm` 函数不返回 warning 的错误 #19291
- 修复了当使用 TiDB 落盘时产生的非预期错误 #19272
- 支持在 Index Join 的 inner 端使用单个分区表 #19197
- 修复了对 decimal 产生的错误的 hash 键值 #19188
- 修复了当 table EndKey 和 Region EndKey 相同时 TiDB 会产生 no regions 的错误 #19895
- 修复了 `alter partition` 的非预期成功 #19891
- 修复了在下推表达式上，默认最大允许的包长的错误 #19876
- 修复了在 ENUM/SET 列上 `Max/Min` 函数的错误行为 #19869
- 修复了当部分 TiFlash 节点下线之后，`tiflash_segments` 和 `tiflash_tables` 系统表读取失败的问题 #19748
- 修复了 `Count()` 聚集函数的错误结果 #19628
- 修复了 `TRUNCATE` 操作的运行时错误 #19445
- 修复了 `PREPARE statement FROM @Var` 语句在 `Var` 包含大写字符时候会失败的错误 #19378
- 修复了在具有大写表名的表上修改 charset 会产生 panic 的错误 #19302
- 修复了当在包含 `tikv/tiflash` 信息时，`information_schema.statements_summary` 和 `explain` 计划的不一致性 #19159
- 修复了在测试中 `select into outfile` 出现文件不存在的错误 #19725
- 修复了 `INFORMATION_SCHEMA.CLUSTER_HARDWARE` 不含有 raid 设备信息的问题 #19457
- 修复一个问题，使具有 `case-when` 表达式生成列的索引添加操作在遇到 parse 错误时能够正常退出 #19395

  - 修复 DDL 长时间重试的错误 #19488
  - 修复错误，使 `alter table db.t1 add constraint fk foreign key (c2)references t2(c1)` 语句执行不需要先执行 `use db` #19471
  - 修复使日志文件中 dispatch errors 从 Error 形式转变为 Info 信息 #19454

- TiKV

  - 修复开启 collation 时对于非 index 列统计信息估算错误的问题 #8620
  - 修复当迁移 Region 时 Green GC 可能错过 lock 的问题 #8460
  - 修复 TiKV 在极端繁忙下 Raft 成员变更可能出现 panic 的问题 #8497
  - 修复 PD client 和其他线程发起 PD sync requests 可能导致死锁的问题 #8612
  - 升级 jemalloc 到 5.2.1 以解决 huge page 的内存分配问题 #8463
  - 修复 unified thread pool 可能停止工作的问题 #8427

- PD

  - 添加 `initial-cluster-token` 配置避免启动时 cluster 之间的通信 #2922
  - 修正自动模式下 store limit 的单位 #2826
  - 添加对于 scheduler 持久化时引发的错误的处理 #2818
  - 修复 scheduler 的 http 接口的返回结果可能为空的问题 #2871 #2874

- TiFlash

  - 修复在更早版本中修改主键列名后，升级到 v4.0.4/v4.0.5 时 TiFlash 启动失败的问题
  - 修复在修改列的 nullable 属性后访问数据可能抛异常的问题
  - 修复在计算表同步状态时导致的崩溃问题
  - 修复当用户进行一些不兼容的 DDL 操作后，读取 TiFlash 数据遇到异常的问题
  - 修复从 TiDB 同步到不支持的 collation 时，抛出异常的问题
  - 修复 Grafana 中 TiFlash coprocessor executor QPS 面板始终显示为 0 的问题
  - 修复 `FROM_UNIXTIME` 函数遇到 `NULL` 值时返回错误结果的问题

- Tools

  - TiCDC

    * 解决某些场景下内存泄露的问题 #942
    * 解决 Kafka sink 可能会出现的异常退出的问题 #912
    * 解决 CRTs 小于 Resolved Ts 而异常退出的问题 #927
    * 解决同步任务可能卡在 MySQL 上的问题 #936
    * 修复 TiCDC 不合理的 Resolved Ts 超时等待 #8573

  - Backup & Restore (BR)

    * 解决数据校验期间可能出现的异常退出的问题 #479
    * 解决 PD leader 切换后可能出现的异常退出的问题 #496

  - Dumpling

    * 解决 binary 类型的 `NULL` 值没有被正确处理的问题 #137

  - TiDB Lightning

    * 解决 write 和 ingest 失败后依旧显示成功的问题 #381
    * 解决写 checkpoint 不及时的问题 #386

16.18.12 TiDB 4.0.5 Release Notes

发版日期：2020 年 8 月 31 日

TiDB 版本：4.0.5

16.18.12.1 兼容性变化

- TiDB
  - 修改 `drop partition` 和 `truncate partition` 的参数 #18930
  - 为 `add partition` 操作添加状态检查 #18865

16.18.12.2 新功能

- TiKV
  - 为错误定义错误码 #8387
- TiFlash
  - 支持与 TiDB 统一的 log 格式
- Tools
  - TiCDC
    * 支持加密 Kafka 链接 #764
    * 支持输出 old value #708
    * 添加列的特征的标识 #796
    * 支持输出上一版本的 DDL 和表结构 #799

16.18.12.3 优化提升

- TiDB
  - 优化 `Union` 场景下 `DecodePlan` 的开销 #18941
  - 减少 GC 在遇到 `Region cache miss` 错误时扫描锁的次数 #18876
  - 减少统计信息 feedback 对集群性能的影响 #18772
  - 支持在 RPC 请求返回结果前取消操作 #18580
  - 支持使用 HTTP API 生成带有相关监控项名称的 profile #18531
  - 支持分区表的预打散功能 #17863
  - 在监控面板中显示每个实例的内存使用详情 #18679
  - 在 `EXPLAIN` 中显示 `BatchPointGet` 算子的详细运行信息 #18892
  - 在 `EXPLAIN` 中显示 `PointGet` 算子的详细运行信息 #18817
  - 解决 `MemTracker` 潜在的死锁问题 #18395
  - 提高字符串转换为整数类型和小数类型的兼容性，支持将 JSON 转换为时间日期类型 #18159
  - 支持限制 `TableReader` 算子内存使用 #18392

- 在 batch cop 请求重试时避免多次 backoff #18999
- 提升 ALTER TABLE 的兼容性 #19270
- 单个分区支持 IndexJoin #19151
- 支持在 log 中存在非法字符时搜索 log #18579

- PD

- 支持打散特殊存储引擎节点（例如 TiFlash）上的 Region #2706
- 支持通过 API 指定某范围内的 Region 优先进行调度 #2687
- 优化 Region 打散操作，使得 Leader 分布更均匀 #2684
- 针对 TSO 请求添加更多测试和日志 #2678
- 避免 Region Leader 变化时可能产生的不必要的缓存更新 #2672
- 增加选项允许 store.GetLimit 返回 tombstone 状态的 store #2743
- 支持 PD Leader 和 Follower 之间同步 Region Leader 变更 #2795
- 增加查询 GC safepoint 服务的命令 #2797
- 替换 filter 中的 region.Clone 调用，优化性能 #2801
- 增加关闭 Region 流量统计缓存更新的选项，用于提升大规模集群的性能 #2848

- TiFlash

- 添加更多的 Grafana 监控面板，比如 CPU、I/O、RAM 使用量，以及存储引擎的各项指标
- 通过优化 Raft logs 的处理逻辑，减少 I/O 操作
- 加快 add partition DDL 之后 Region 的调度速度
- 优化 DeltaTree 引擎中 delta 数据的整理，减少读写放大
- 通过使用多线程对 Region snapshot 进行预处理，优化从 TiKV 同步 Region 副本的性能
- 优化系统负载较低时打开文件描述符的数量，降低系统资源占用量
- 减少 TiFlash 重启时新创建的文件数量
- 支持数据存储的静态加密功能
- 支持数据传输的 TLS 功能

- Tools

- TiCDC
  * 减少了获取时间戳的频率 #801
- Backup & Restore (BR)
  * 优化了日志 #428
- Dumpling
  * 减少导出 MySQL 时持锁的时间 #121
- TiDB Lightning
  * 优化了日志 #352

16.18.12.4 Bug 修复

- TiDB

  - 修复 builtinCastRealAsDecimalSig 函数中未正确处理 ErrTruncate/Overflow 错误导致报 should ensure all columns have the same length 错误的问题 #18967
  - 修复 pre_split_regions 对分区表不生效的问题 #18837
  - 修复大事务提前终止的问题 #18813
  - 修复使用 collation 相关函数查询结果错误的问题 #18735
  - 修复 getAutoIncrementID() 函数逻辑错误导致导出工具报 table not exist 错误的问题 #18692
  - 修复 select a from t having t.a 报 unknown column error 的问题 #18434
  - 修复 Hash 分区表的分区键为整数类型时，写入 64 位无符号类型导致溢出 panic 的问题 #18186
  - 修复 char 函数行为错误的问题 #18122
  - 修复 ADMIN REPAIR TABLE 无法解析 range 分区表表达式中整数的问题 #17988
  - 修复 SET CHARSET 行为不正确的问题 #17289
  - 修复由于错误的设置 collation 导致 collation 函数返回错误结果的问题 #17231
  - 修复 STR_TO_DATE 和 MySQL 行为不一致的问题 #18727
  - 修复 cluster_info 表中，TiDB 版本和 PD/TiKV 不一致的问题 #18413
  - 修复悲观事务未能检查出重复数据导致可以重复写入冲突数据的问题 #19004
  - 修复 union select for update 存在并发竞态的问题 #19006
  - 修复自查询含有 PointGet 算子时返回结果错误的问题 #19046
  - 修复 IndexLookUp 作为 Apply 的内连接算子时查询结果不正确的问题 #19496
  - 修复 anti-semi-join 查询结果不正确的问题 #19472
  - 修复 BatchPointGet 查询结果不正确的问题 #19456
  - 修复 UnionScan 作为 Apply 的内连接算子时查询结果不正确的问题 #19496
  - 修复使用 EXECUTE 语句产生大查询日志造成 panic 的问题 #17419
  - 修复 IndexJoin 在使用 ENUM 或 SET 类型作为连接键报错的问题 #19235
  - 修复在索引值为 NULL 时无法构建出查询范围的问题 #19358
  - 修复更新全局配置导致的数据竞态问题 #17964
  - 修复修改 schema 字符集导致 panic 的问题 #19286
  - 修复修改文件夹对中间结果落盘功能的影响 #18970
  - 修复 decimal 类型哈希值不正确的问题 #19131
  - 修复 PointGet 和 BatchPointGet 在分区表场景下报错的问题 #19141
  - 修复共同使用 Apply 算子和 UnionScan 算子时查询结果不正确的问题 #19104
  - 修复生成列索引结果不正确的问题 #17989
  - 修复并发收集统计信息 panic 的问题 #18983

- TiKV

  - 修复开启 Hibernate Region 时，某些情况下 leader 选举慢的问题 #8292
  - 修复 Region 调度产生的一个内存泄露问题 #8357
  - 增加 hibernate-timeout 配置避免 leader 过快变为 Hibernate 状态 #8208

- PD

  - 修复 PD leader 切换时可能导致一段时间内 TSO 不可用的问题 #2666
  - 修复开启 Placement Rule 时，某些情况下 Region 无法调度至最佳状态的问题 #2720
  - 修复开启 Placement Rules 后，Balance Leader 不工作的问题 #2726
  - 修复不健康的 Store 未从负载统计信息中过滤的问题 #2805

- TiFlash

  - 修复 TiFlash 从旧版本升级到新版本的过程中，由于包含特殊字符而导致进程无法启动的问题
  - 修复 TiFlash 进程在初始化过程中，一旦出现任何异常就无法退出的问题

- Tools

  - Backup & Restore (BR)

    * 修复 total KV 和 total bytes 被计算两次的问题 #472
    * 修复切换模式不及时导致恢复缓慢的问题 #473

  - Dumpling

    * 修复 FTWRL 锁没有及时释放的问题 #128

  - TiCDC

    * 解决了同步任务不能被移除的问题 #782
    * 修正了错误的删除事件 #787
    * 解决了已停止的同步任务会卡住 GC 的问题 #797
    * 解决了网络阻塞导致同步任务不能退出的问题 #825
    * 修复在某些情况下无关数据被错误地到下游的问题 #743

  - TiDB Lightning

    * 解决了 TiDB backend 遇到空 binary/hex 的时候出现语法错误的问题 #357

16.18.13 TiDB 4.0.4 Release Notes

发版日期：2020 年 7 月 31 日

TiDB 版本：4.0.4

16.18.13.1 Bug 修复

- TiDB

  - 修复查询 `information_schema.columns` 卡死的问题 #18849
  - 修复 `PointGet` 和 `BatchPointGet` 在遇到 `in(null)` 条件时出错的问题 #18848
  - 修复 `BatchPointGet` 算子结果不正确的问题 #18815
  - 修复 `HashJoin` 算子在遇到 `set`、`enum` 类型时查询结果不正确的问题 #18859

16.18.14 TiDB 4.0.3 Release Notes

发版日期：2020 年 7 月 24 日

TiDB 版本：4.0.3

16.18.14.1 新功能

- TiDB Dashboard
  - 显示详细的 TiDB Dashboard 版本信息 #679
  - 显示不受支持的浏览器或过时的浏览器的兼容性通知 #654
  - 支持在 SQL 语句分析页面搜索 #658
- TiFlash
  - TiFlash proxy 支持文件加密功能
- Tools
  - Backup & Restore (BR)
    - 支持使用 zstd、 lz4、 snappy 算法压缩备份文件 #404
  - TiCDC
    - 支持 sink-uri 中配置 Kafka 客户端的 ID #706
    - 支持离线更新同步任务的配置 #699
    - 支持自定义同步任务的 ID #727
    - 支持使用 SSL 加密链接向 MySQL 输出数据 #347
    - 支持输出 Avro 格式的变更数据 #753
    - 支持向 Apache Pulsar 输出变更数据 #751
  - Dumpling
    - 支持自定义 CSV 文件的分隔符和换行符 #116
    - 支持自定义输出文件名格式 #122

16.18.14.2 改进提升

- TiDB
  - 增加全局变量 `tidb_log_desensitization` 来控制在日志中记录 SQL 时是否脱敏 #18581
  - 默认打开 `tidb_allow_batch_cop` #18552
  - 加速 `kill tidb sesesion_id` 的执行速度 #18505
  - 函数 `tidb_decode_plan` 的结果增加表头输出 #18501
  - 配置检查器可以兼容旧版本的配置文件 #18046
  - 默认打开执行信息的收集 #18518
  - 增加系统表 `tiflash_tables` 和 `tiflash_segments` #18536
  - `AUTO RANDOM` 被移出实验特性并正式 GA，有如下的改进和兼容性修改：
    - 在配置文件中，将 `experimental.allow-auto-random` 废弃，该无论该选项如何配置，都可以在列上定义 `AUTO_RANDOM` 属性 #18613 #18623
    - 为避免显式写入 `AUTO_RANDOM` 列造成非预期的 `AUTO_RANDOM_BASE` 的更新，新增 session 变量 `tidb_allow_auto_random_explicit_insert` 用于控制 `AUTO_RANDOM` 列的显式写入，该变量默认值为 `false` #18508

    * 为避免分配空间被快速消耗，AUTO_RANDOM 列现在仅允许在 BIGINT 和 UNSIGNED BIGINT 列上定义，并将最大的 Shard Bit 数量限制为 15 #18538
    * 当在 BIGINT 列上定义 AUTO_RANDOM 属性，并显示插入负值的整型主键时，将不会再触发 AUTO_RANDOM_BASE 的更新 #17987
    * 当在 UNSIGNED BIGINT 列上定义 AUTO_RANDOM 属性，分配 ID 时将利用整数的最高位以获得更大的分配空间 #18404
    * 在 SHOW CREATE TABLE 的结果中支持 AUTO_RANDOM_BASE 属性的更新 #18316

- TiKV

  - 添加了新的配置项 backup.num-threads 用语控制 backup 线程池的大小 #8199
  - 收取 snapshot 时不再发送 store heartbeat #8136
  - 支持动态调整 shared block cache 的大小 #8232

- PD

  - 支持 JSON 格式日志 #2565

- TiDB Dashboard

  - 优化 key Visualizer 中冷表的 bucket 合并 #674
  - 重命名配置项 disable-telemetry 以使遥测更一致 #684
  - 切换页面时显示进度条 #661
  - 保证慢日志查询和日志查询行为的一致性，即使在空格存在的情况 #682

- TiFlash

  - 将 Grafana DDL Jobs 面板中的单位修改为 operations per minute
  - 在 Grafana 中新增关于 TiFlash-Proxy 的详细监控指标面板
  - 降低 TiFlash Proxy 的 IOPS

- Tools

  - TiCDC

    * 将监控指标总的表 ID 替换为表名 #695

  - Backup & Restore (BR)

    * 支持输出 JSON 格式的日志 #336
    * 支持在运行 BR 期间动态开启 pprof #372
    * 加速恢复时 DDL 的执行速度 #377

  - TiDB Lightning

    * 使用一种更加简单易懂的表过滤机制替换原先的黑白名单机制 #332

16.18.14.3 Bug 修复

- TiDB

  - 当 IndexHashJoin 遇到执行中发生非内存相关的错误时，返回错误而不是空结果集 #18586

  - 修复 gRPC transportReader 导致的反复异常 #18562
  - 修复因为 Green GC 不会扫描已下线 store 上的锁而可能导致数据不完整的问题 #18550
  - 非只读语句不会使用 TiFlash 引擎 #18534
  - 当查询连接异常时返回真实的错误信息 #18500
  - 修复非 repair mode 的 TiDB 节点不会重新读取修复的表元信息的错误 #18323
  - 修复当锁住的 primary key 在当前事务被插入/删除时可能造成的结果不一致问题 #18291
  - 修复数据落盘为正确生效导致的内存溢出 #18288
  - 修复 `REPLACE INTO` 语句作用在包含生成列的表时会错误报错的问题 #17907
  - 当 `IndexHashJoin` 及 `IndexMergeJoin` 执行异常时抛出 `Out Of Memory Quota!` 错误 #18527
  - 修复当 `Index Join` 使用的索引包含整型主键时，特殊情况下执行结果可能出错的问题 #18565
  - 修复当开启 new collation 时，若在事务内的更新涉及了 new collation 列，并在该事务内通过唯一索引读取更新数据时，被更新的数据无法被读取到的问题 #18703

- TiKV

  - 修复 merge 期间可能读到过期数据的问题 #8113
  - 修复聚合函数 `min/max` 下推到 TiKV 时，collation 不能正确工作的问题 #8108

- PD

  - 修复如果服务器崩溃，创建 TSO 流可能会被阻塞一段时间的问题 #2648
  - 修复 `getSchedulers` 可能导致数据争用的问题 #2638
  - 修复删除 `scheduler` 时导致死锁的问题 #2637
  - 修复 `balance-leader-scheduler` 没有考虑 placement rule 的问题 #2636
  - 修复有时无法正确设置 `safepoint` 的问题，这可能会使 BR 和 dumpling 失败 #2635
  - 修复 `hot region scheduler` 中目标 store 选择错误的问题 #2627
  - 修复 PD Leader 切换时 TSO 请求可能花费太长时间的问题 #2622
  - 修复 PD Leader 切换后过期 `scheduler` 的问题 #2608
  - 修复了启用 placement rule 时，有时 Region 的副本可能无法调整到最佳位置的问题 #2605
  - 修复了存储的部署路径不会随着部署目录移动而更新的问题 #2600
  - 修复了 `store limit` 可能为零的问题 #2588

- TiDB Dashboard

  - 修复 TiDB 扩容时的 TiDB 连接错误 #689
  - 修复 TiFlash 实例未显示在日志搜索页面的问题 #680
  - 修复概况页面刷新之后 metrics 会重置的问题 #663
  - 修复某些 TLS 方案中的连接问题 #660
  - 修复在某些情况下无法正确显示语言的下拉列表 #677

- TiFlash

  - 修复更改主键列名后 TiFlash 崩溃的问题
  - 修复 Learner Read 与 Remove Region 并发时可能的死锁问题

- Tools

  - TiCDC

    * 解决了某些场景下可能发生的 OOM 问题 #704
    * 解决了某些特殊表名可能导致 SQL 语法出错的问题 #676
    * 解决了同步任务处理单元无法正常退出的问题 #693

  - Backup & Restore (BR)

    * 解决了备份汇总报告中时间为负数的问题 #405

  - Dumpling

    * 解决了 NULL 值在有 `-r` 参数时被忽略的问题 #119
    * 解决了导出数据时 flush table 没有正常工作的问题 #117

  - TiDB Lightning

    * 解决了 `--log-file` 参数不生效的问题 #345

  - TiDB Binlog

    * 修复开启 TLS 写下游时用来保存 checkpoint 的 DB 没有开启 TLS 导致 Drainer 无法启动的问题 #988

16.18.15 TiDB 4.0.2 Release Notes

发版日期：2020 年 7 月 1 日

TiDB 版本：4.0.2

16.18.15.1 兼容性

- TiDB

  - 移除慢查询日志和 statement summary 表中的敏感信息 #18130
  - 禁止在 sequence 缓存中出现负数 #18103
  - CLUSTER_INFO 表中不再显示 tombstone 状态的 TiKV 和 TiFlash 节点 #17953
  - 诊断规则 `current-load` 变更为 `node-check` #17660

- PD

  - 持久化 `store-limit` 配置项，弃用 `store-balance-rate` 配置 #2557

16.18.15.2 新更改

- TiDB 及 TiDB Dashboard 默认收集使用情况信息，并将这些信息分享给 PingCAP 用于改善产品 #18180。若要了解所收集的信息详情及如何禁用该行为，请参见遥测文档。

16.18.15.3 新功能

- TiDB

  - 支持在 `INSERT` 语句中使用 `MEMORY_QUOTA()` hint #18101
  - 支持基于 TLS 证书 SAN 属性的登录认证 #17698

  - REGEXP() 函数支持 collation #17581
  - 支持会话和全局变量 sql_select_limit #17604
  - 支持新增分区时自动分裂 Region 的功能 #17665
  - 支持函数 IF()/BITXOR()/BITNEG()/JSON_LENGTH() 下推到 TiFlash Coprocessor 上执行 #17651 #17592
  - 支持聚合函数 APPROX_COUNT_DISTINCT()，用于快速计算 COUNT(DISTINCT) 的近似值 #18120
  - TiFlash 支持了 collation，支持相应的函数下推 #17705
  - INFORMATION_SCHEMA.INSPECTION_RESULT 表新增 STATUS_ADDRESS 列，用于展示节点的 status 地址 #17695
  - MYSQL.BIND_INFO 表新增 SOURCE 列，用于展示 binding 的创建方式 #17587
  - PERFORMANCE_SCHEMA.EVENTS_STATEMENTS_SUMMARY_BY_DIGEST 表新增 PLAN_IN_CACHE 和 PLAN_CACHE_HITS 列，用于展示 plan cache 的使用情况 #17493
  - 新增配置项 enable-collect-execution-info 和会话级变量 tidb_enable_collect_execution_info 用于控制是否记录算子的执行信息并打印到慢查询日志中 #18073 #18072
  - 新增全局变量 tidb_slow_log_masking，用于控制是否脱敏慢查询日志中的查询 #17694
  - 增加对 TiKV 配置项 storage.block-cache.capacity 的诊断规则 #17671
  - 新增 SQL 语法 BACKUP/RESTORE 来进行数据备份恢复 #15274

- TiKV

  - TiKV Control 支持 encryption-meta 命令 #8103
  - 增加 RocksDB::WriteImpl 相关的 perf context 监控 #7991

- PD

  - 对 leader 执行 remove-peer 操作时，让这个 operator 不等待超时，立刻失败 #2551
  - 对 TiFlash 节点设置更合理的 store limit 配置默认值 #2559

- TiFlash

  - Coprocessor 支持新的聚合函数 APPROX_COUNT_DISTINCT
  - 存储引擎中的粗糙索引默认开启
  - 支持运行在 ARM 架构
  - Coprocessor 支持 JSON_LENGTH 函数下推

- TiCDC

  - 支持 Capture 节点扩容时迁移部分子任务到新加节点 #665
  - Cli 中添加清理 TiCDC GC TTL 的功能 #652
  - 在 MQ sink 中支持输出 Canal 协议 #649

16.18.15.4 改进提升

- TiDB

  - 降低当集群中 CM-Sketch 占用过多内存时，Golang 内存分配带来的查询延迟 #17545
  - 缩短 TiKV 故障恢复时集群 QPS 的恢复时间 #17681
  - 为 partition 表上的查询支持聚合函数下推到 TiKV 或者 TiFlash coprocessor #17655
  - 提升索引上等值条件的行数估算准确度 #17611

- TiKV
  - 优化 PD client panic 日志信息 #8093
  - 重新加回 `process_cpu_seconds_total` 和 `process_start_time_seconds` 监控 #8029
- TiFlash
  - 提升从旧版本升级时的兼容性
  - 降低 delta index 的内存使用量
  - 使用更高效的 delta index update 算法
- Tools
  - Backup & Restore (BR)
    * 提升多表场景下的恢复数据性能 #266

16.18.15.5 Bug 修复

- TiDB
  - 修复 `tidb_isolation_read_engines` 更改后从 plan cache 中获取的执行计划不正确的问题 #17570
  - 修复某些情况下 `EXPLAIN FOR CONNECTION` 返回运行时错误的问题 #18124
  - 修复某些情况下 `last_plan_from_cache` 结果不正确的问题 #18111
  - 修复执行 plan cache 中的 `UNIX_TIMESTAMP()` 时的运行时错误 #18002 #17673
  - 修复 `HashJoin` 算子的孩子节点返回 `NULL` 类型的结果时，计算过程中的运行时错误 #17937
  - 修复当在同一个数据库中并发执行 `DROP DATABASE` 语句和相关 DDL 语句时的运行错误 #17659
  - 修复当 `COERCIBILITY()` 的输入参数是用户变量时结果不正确的问题 #17890
  - 修复 `IndexMergeJoin` 算子偶尔卡住的问题 #18091
  - 修复 `IndexMergeJoin` 算子触发 oom-action 后被取消执行时卡住的问题 #17654
  - 修复 `Insert` 和 `Replace` 算子的内存统计过大的问题 #18062
  - 修复在执行 `DROP DATABASE` 的同时对同一个数据库中的表 `DROP TABLE` 时，数据不再向 TiFlash 同步的问题 #17901
  - 修复 TiDB 和对象存储服务之间 BACKUP/RESTORE 失败的问题 #17844
  - 修复权限检查失败时的错误信息 #17724
  - 修复 DELETE/UPDATE 语句的 feedback 被错误收集的问题 #17843
  - 禁止更改非 AUTO_RANDOM 表的 AUTO_RANDOM_BASE 值 #17828
  - 修复通过 `ALTER TABLE ... RENAME` 在数据库间移动表时，AUTO_RANDOM 列分配到错误结果的问题 #18243
  - 修复系统变量 `tidb_isolation_read_engines` 的值中没有 `tidb` 时某些系统表无法访问的问题 #17719
  - 修复 JSON 中大整数和浮点数比较的精度问题 #17717
  - 修复 `COUNT()` 函数的返回类型中 `DECIMAL` 不正确的问题 #17704
  - 修复 `HEX()` 函数的输入类型是二进制字符串时结果不正确的问题 #17620
  - 修复查询 `INFORMATION_SCHEMA.INSPECTION_SUMMARY` 表没有指定过滤条件时返回结果为空的问题 #17697
  - 修复 `ALTER USER` 语句使用哈希后的密码更新用户信息后，密码不符合预期的问题 #17646

  - 为 ENUM 和 SET 类型支持 collation #17701
  - 修复 CREATE TABLE 时预切分 Region 的超时机制不生效的问题 #17619
  - 修复某些情况下 DDL 后台作业重试时，schema 未正确更新导致的 DDL 原子性问题 #17608
  - 修复 FIELD() 函数的参数包含 column 时结果不正确的问题 #17562
  - 修复某些情况下 max_execution_time hint 不生效的问题 #17536
  - 修复某些情况下 EXPLAIN ANALYZE 的结果中并发信息被多次打印的问题 #17350
  - 修复对 STR_TO_DATE 函数的 %h 解析和 MySQL 不兼容问题 #17498
  - 修复 tidb_replica_read 设置成 follower，并且 Region 的 leader 和 follower/learner 之间出现网络分区后，TiDB 发送的 request 一直重试的问题 #17443
  - 修复某些情况下 TiDB 一直 ping PD 的 follower 的问题 #17947
  - 修复老版本的 range partition 表无法在 4.0 集群中加载的问题 #17983
  - 修复当多个 Region 的请求同时超时时整个 SQL 语句超时的问题 #17585
  - 修复解析日期类型的分隔符时和 MySQL 不兼容的问题 #17501
  - 修复少数情况下发给 TiKV 的请求错发给 TiFlash 的问题 #18105
  - 修复当前事务中主键被插入/删除但主键的锁却被另一事务清除可能造成结果不一致的问题 #18250

- TiKV

  - 修复 status server 的内存安全问题 #8101
  - 修复 json 数字比较的精度丢失问题 #8087
  - 修改错误的慢查询日志 #8050
  - 修复 merge 可能导致 peer 无法被移除的问题 #8048
  - 修复 tikv-ctl recover-mvcc 未清除无效的悲观锁 #8047
  - 修复一些遗漏的 Titan 监控 #7997
  - 修复向 TiCDC 返回 duplicated error 的问题 #7887

- PD

  - 验证 pd-server.dashboard-address 配置项的正确性 #2517
  - 修复设置 store-limit-mode 为 auto 时可能引起 PD panic 的问题 #2544
  - 修复某些情况下热点不能识别的问题 #2463
  - 修复某些情况下 Placement Rules 会使 store 状态变更为 tombstone 的进程被阻塞的问题 #2546
  - 修复某些情况下从低版本升级后，PD 无法正常启动的问题 #2564

- TiFlash

  - 修正 proxy 遇到 region not found 时可能的 panic 的问题
  - 修正 schema 同步遇到 I/O exception 时可能无法继续同步的问题

16.18.16 TiDB 4.0.1 Release Notes

发版日期：2020 年 6 月 12 日

TiDB 版本：4.0.1

16.18.16.1 新功能

- TiKV
  - 添加 `--advertise-status-addr` 启动参数 #8046
- PD
  - 为内嵌的 TiDB Dashboard 添加内部代理的支持 #2511
  - 添加对 PD 客户端自定义超时的设置 #2509
- TiFlash
  - 支持 TiDB `new collation framework` 排序规则框架
  - 支持函数 `If/BitAnd/BitOr/BitXor/BitNot/Json_length` 下推到 TiFlash
  - 支持 TiFlash 中对于大事务的 Resolve Lock 逻辑
- Tools
  - Backup & Restore (BR)
    * 增加启动时集群版本检查，避免 BR 和 TiDB 集群不兼容的问题 #311

16.18.16.2 Bug 修复

- TiKV
  - 修复日志中 `use-unified-pool` 配置打印不正确的问题 #7946
  - 修复 tikv-ctl 不支持相对路径的问题 #7963
  - 修复点查监控指标不准确的问题 #8033
  - 修复过时副本在网络隔离消除后不能销毁的问题 #8006
  - 修复 `read index` 可能过时的问题 #8043
  - 改善备份恢复文件操作的可靠性 #7917
- PD
  - 防止某些场景下对 Placement Rules 的错误配置 #2516
  - 修复删除 Placement Rules 可能引发 panic 的问题 #2515
  - 修复当 Store 的已用空间为零时无法获取 Store 信息的 bug #2474
- TiFlash
  - 修复 TiFlash 中 Bit 类型列的 Default Value 解析不正确的问题
  - 修复 TiFlash 对于 `1970-01-01 00:00:00 UTC` 在部分时区下计算错误的问题

16.18.17 TiDB 4.0 GA Release Notes

发版日期：2020 年 5 月 28 日

TiDB 版本：4.0.0

16.18.17.1 兼容性变化

- TiDB
  - 优化事务过大时系统的报错信息，方便排查问题 #17219
- TiCDC
  - 优化 Changefeed 配置文件的结构，提升易用性 #588
  - 新增 `ignore-txn-start-ts` 配置项，过滤事务时条件由原来的 `commit_ts` 改为 `start_ts` #589

16.18.17.2 重点修复的 Bug

- TiKV
  - 修复 BR 备份时出现 `DefaultNotFound` 错误的问题 #7937
  - 修复 `ReadIndex` 因响应的数据包乱序而导致系统 panic 的问题 #7930
  - 修复 TiKV 重启后由于 snapshot 文件被错误删除导致系统 panic 的问题 #7927
- TiFlash
  - 修复因 Raft Admin Command 处理逻辑不正确，系统 panic 导致数据可能会丢失的问题

16.18.17.3 新功能

- TiDB
  - 新增 `committer-concurrency` 配置项，用于控制 retry commit 阶段的 goroutine 数量 #16849
  - 支持 `show table partition regions` 语法 #17294
  - 新增 `tmp-storage-quota` 配置项，用于限制 tidb-server 使用的临时磁盘空间 #15700
  - 创建和更改表时新增检查分区表是否使用唯一前缀索引的功能 #17213
  - 支持 `insert/replace into tbl_name partition(partition_name_list)` 语句 #17313
  - Distinct 函数支持检查 collations 的值 #17240
  - 哈希分区裁剪时支持 `is null` 过滤条件 #17310
  - 分区表中支持 `admin check index`、`admin cleanup index` 和 `admin recover index` #17392 #17405 #17317
  - 支持 `in` 表达式的范围分区裁剪 #17320
- TiFlash
  - Learner 读取数据时通过 Lock CF 的 `min commit ts` 值过滤出符合条件的 TSO 对应的数据
  - 若 Timestamp 类型的值小于 `1970-01-01 00:00:00`，系统显式报错以避免计算结果出错
  - Search log 的正则表达式支持使用 flag 参数
- TiKV
  - 支持 `ascii_bin` 和 `latin1_bin` 编码的排序规则 #7919
- PD
  - 支持为内置的 TiDB Dashboard 指定反向代理资源前缀 #2457
  - PD client Region 相关接口支持返回 `pending peer` 和 `down peer` 的信息 #2443

  - 添加 `Direction of hotspot move leader`、`Direction of hotspot move peer` 和 `Hot cache read entry number` 等监控 #2448

- Tools

  - Backup & Restore (BR)

    * 支持备份与恢复 `Sequence` 和 `View` #242

  - TiCDC

    * 创建 Changefeed 时新增检查 Sink URI 的合法性 #561
    * 系统启动时检查 PD 和 TiKV 版本是否符合系统要求 #570
    * 支持同一个调度任务生成周期内可调度多张表 #572
    * HTTP API 中增加节点角色的信息 #591

16.18.17.4 Bug 修复

- TiDB

  - 修复收发消息有不符合预期的超时，禁止合并发向 TiFlash 的消息 #17307
  - 修复分区裁剪时未正确区分有符号/无符号整数的错误，并提高了性能 #17230
  - 修复 3.1.1 升级到 4.0 时由于 `mysql.user` 表不兼容导致升级失败的问题 #17300
  - 修复 `update` 语句分区选择不正确的问题 #17305
  - 修复从 TiKV 收到未知错误消息时系统 panic 的问题 #17380
  - 修复创建按 key 分区的分区表由于处理逻辑不正确导致系统 panic 的问题 #17242
  - 修复某些情况下优化器处理逻辑不正确导致错误地选择 `Index Merge Join` 的问题 #17365
  - 修复 Grafana 中 `SELECT` 语句的 `duration` 的监控指标不准确的问题 #16561
  - 修复当系统发生错误时，GC 线程卡住的问题 #16915
  - 修复当列的类型是 `Boolean` 时，由于 `UNIQUE` 约束比较的逻辑不正确导致输出结果不正确的问题 #17306
  - 修复 `tidb_opt_agg_push_down` 开启且聚合函数下推分区表信息时，由于逻辑处理不正确导致系统 panic 的问题 #17328
  - 修复某些情况下会访问已经发生故障的 TiKV 的问题 #17342
  - 修复 `tidb.toml` 中的配置项 `isolation-read` 不生效时的问题 #17322
  - 修复通过 `hint` 强制执行流式聚合时由于逻辑处理不正确导致输出的结果顺序不正确的问题 #17347
  - 修复不同 `SQL_MODE` 时 `insert` 处理 `DIV` 的行为 #17314

- TiFlash

  - 修复 Search log 功能正则表达式匹配的行为与其他组件不一致的问题
  - 默认关闭 Raft Compact Log Command 延迟处理的优化，避免节点大量写入数据时重启时间过长的问题
  - 修复部分场景因 TiDB 执行 `DROP DATABASE` 处理逻辑不正确导致系统启动不成功的问题
  - 修复采集 `Server_info` 中 CPU 信息方式与其他组件不一样的问题
  - 修复开启 `batch coprocessor` 功能时，执行 `Query` 报错 `Too Many Pings` 的问题
  - 修复 Dashboard 因未上报 `deploy path` 信息导致相关信息显示不正确的问题

- TiKV

  - 修复 BR 备份时出现 `DefaultNotFound` 错误的问题 #7937

  - 修复 ReadIndex 因响应的数据包乱序时导致系统 panic 问题 #7930
  - 修复读请求回调函数没有被调用，导致返回非预期错误的问题 #7921
  - 修复 TiKV 重启后由于 snapshot 文件被错误删除导致系统 panic 问题 #7927
  - 修复存储加密中因处理逻辑不正确导致 master key 无法轮转的问题 #7898
  - 修复开启存储加密后 snapshot 的 lock cf 文件在接收后未被加密的问题 #7922
- PD
  - 修复使用 pd-ctl 删除 `evict-leader-scheduler` 或者 `grant-leader-scheduler` 时报 404 错误的问题 #2446
  - 修复当存在 TiFlash 副本的时候，可能会导致 `presplit` 功能无法正常使用的问题 #2447
- Tools
  - BR
    * 修复从云存储恢复数据时因网络原因导致恢复失败的问题 #298
  - TiCDC
    * 修复若干因数据争用 (data race) 导致系统 panic 的问题 #565 #566
    * 修复若干因处理逻辑不正确导致资源泄露或系统阻塞的问题 #574 #586
    * 修复 CLI 因连接不上 PD 导致命令行阻塞的问题 #579

16.18.18 TiDB 4.0 RC.2 Release Notes

发版日期：2020 年 5 月 15 日

TiDB 版本：4.0.0-rc.2

16.18.18.1 兼容性变化

- TiDB
  - 去掉了特别为开启 Binlog 时定义的事务容量上限 (100 MB)，现在事务的容量上限统一为 10 GB，但若开启 Binlog 且下游是 Kafka，由于 Kafka 消息大小的限制是 1 GB，请根据情况调整 `txn-total-size-limit` 配置参数 #16941
  - 查询 `CLUSTER_LOG` 表时，如果未指定时间范围，由默认时间范围变更为返回错误且用户必须指定时间范围 #17003
  - `CREATE TABLE` 创建分区表时指定未支持的 sub-partition 或 linear hash 选项，将会创建非分区普通表，而不是选项未生效的分区表 #17197
- TiKV
  - 将加密相关的配置移到 security 分类下，即调整配置项 `[encryption]` 为 `[security.encryption]` #7810
- Tools
  - TiDB Lightning
    * 导入数据时将 SQL Mode 由默认改成 `ONLY_FULL_GROUP_BY,NO_AUTO_CREATE_USER`，提高兼容性 #316
    * 在 tidb-backend 模式下禁止访问 PD 或者 TiKV 端口 #312
    * 日志信息默认输出到 tmp 文件且在启动时输出 tmp 文件的路径 #313

16.18.18.2 重点修复的 Bug

- TiDB
  - 修复当 WHERE 语句只有一个等值条件时错误选择分区表分区的问题 #17054
  - 修复当 WHERE 语句只包含字符串列时构造错误的 Index Range 导致结果错误的问题 #16660
  - 修复事务中执行 DELETE 之后再进行唯一索引点查语句 Panic 的问题 #16991
  - 修复 GC worker 在有错误发生时可能死锁的问题 #16915
  - 避免 TiKV 未宕机仅响应慢情况下的无故 RegionMiss 重试 #16956
  - 修改客户端 MySQL 协议握手阶段日志级别为 DEBUG，以解决干扰日志输出的问题 #16881
  - 修复 TRUNCATE 后未按照表定义的 PRE_SPLIT_REGIONS 信息进行预切分 Region 的问题 #16776
  - 修复两阶段提交中第二阶段遇到 TiKV 不可用重试导致的 Goroutine 暴涨的问题 #16876
  - 修复部分表达式不能下推可能导致语句执行 Panic 的问题 #16869
  - 修复 IndexMerge 在分区表上执行结果错误的问题 #17124
  - 修复因 Memory Tracker 锁竞争引起的宽表性能回退问题 #17234
- TiFlash
  - 修复库名、表名含特殊字符，系统升级后无法正常启动的问题

16.18.18.3 新功能

- TiDB
  - 新增 BACKUP 和 RESTORE 语句进行备份与恢复 #16960
  - 支持在提交前预检查单个 Region 提交数据量，并在超过阈值时预切分 Region 后再进行提交 #16959
  - 新增 Session 作用域变量 LAST_PLAN_FROM_CACHE，用于指示上一条语句是否命中 Plan Cache #16830
  - 支持在慢日志和 SLOW_LOG 表中记录 Cop_time 信息 #16904
  - 支持在 Grafana 中展示更多 Go Runtime 内存监控指标 #16928
  - 支持在 General Log 中输出 forUpdateTS 和 Read Consistency 隔离级别信息 #16946
  - 支持对 TiKV Region Resolve Lock 请求进行去重 #16925
  - 支持 SET CONFIG 语句进行 PD/TiKV 节点配置修改 #16853
  - 支持在 CREATE TABLE 语句中指定 auto_random 选项 #16813
  - 通过为 DistSQL 请求分配 TaskID 让 TiKV 更好地调度处理请求 #17155
  - 支持在 MySQL 客户端登录后显示 TiDB server 版本信息 #17187
  - 支持在 GROUP_CONCAT 中指定 ORDER BY 子句 #16990
  - 支持在 Slow Log 中展示 Plan_from_cache 信息，用于指示语句是否命中 Plan Cache #17121
  - Dashboard 支持显示 TiFlash 多盘部署容量信息功能
  - Dashboard 支持通过 SQL 查询 TiFlash 日志的功能
- TiKV
  - 加密存储适配 tikv-ctl，适配后开启加密存储后通过 tikv-ctl 操作管理集群 #7698
  - 新增加密码 Snapshot 中的 lock column famliy 的功能 #7712
  - 修改 Raftstore latency 在 Grafana 面板显示方式，采用 heatmap 方便诊断性能抖动问题 #7717
  - 支持配置 gRPC 消息大小的上限 #7824
  - Grafana 面板中添加了 encryption 相关的监控 #7827

  - TiKV 支持 ALPN 协议 #7825
  - 添加了更多的关于 Titan 的统计信息 #7818
  - 统一线程池支持用客户端提供的 task ID 来区分任务，以避免一个请求被来自同一个事务的另一个请求降低优先级 #7814
  - 提升了 batch insert 请求的性能 #7718

- PD

  - 下线节点时放开 Remove peer 的速度 #2372

- TiFlash

  - 调整 Grafana 中 Read Index 的 Count 图表的名称为 Ops
  - 优化系统负载较低时打开文件描述符的数据，降低系统资源占用量
  - 新增 capacity 配置参数用于限制存储数据容量

- Tools

  - TiDB Lightning

    * tidb-lightning-ctl 新增 `fetch-mode` 子命令，输出 TiKV 集群模式 #287

  - TiCDC

    * 支持通过 `cdc cli` 来管理同步任务 (changefeed) #546

  - Backup & Restore (BR)

    * 支持备份时候自动调整 GC 时间 #257
    * 优化恢复数据时的 PD 参数，以加速恢复 #198

16.18.18.4 Bug 修复

- TiDB

  - 完善多个算子中判断是否使用向量化进行表达式执行的逻辑 #16383
  - 修复 IndexMerge Hint 未能正确检查数据库名称的问题 #16932
  - 修复 Sequence 可以被 TRUNCATE 的问题 #17037
  - 修复 Sequence 可以被 INSERT/UPDATE/ANALYZE/DELETE 的问题 #16957
  - 修复启动阶段执行的内部 SQL 在 Statement Summary 表中未能正确记录为内部 SQL 的问题 #17062
  - 修复因 TiFlash 支持但 TiKV 不支持的过滤条件仅被下推到 IndexLookupJoin 算子之下导致的语句报错问题 #17036
  - 修复开启 Collation 后，`LIKE` 表达式可能出现的并发问题 #16997
  - 修复开启 Collation 后，`LIKE` 函数无法正确构造 Range 查询索引的问题 #16783
  - 修复触发填充 Plan Cache 语句后执行 `@@LAST_PLAN_FROM_CACHE` 返回值错误的问题 #16831
  - 修复为 IndexMerge 计算候选路径时漏掉 Index 上的 `TableFilter` 的问题 #16947
  - 修复使用 MergeJoin Hint 并存在 TableDual 算子时无法产生物理查询计划的问题 #17016
  - 修复 Statement Summary 表的 `Stmt_Type` 列值大小写错误的问题 #17018
  - 修复因不同用户使用相同的 tmp-storage-path 导致服务无法启动报 `Permission Denied` 错误问题 #16996

  - 修复返回结果类型由多个输入列决定的表达式（例如：`CASE WHEN`）结果类型 `NotNullFlag` 标识推导不正确的问题 #16995
  - 修复 Green GC 在有 Dirty Store 的情况下可能遗留锁的问题 #16949
  - 修复 Green GC 在遇到单个 key 有多个不同锁的情况下会遗留下锁的问题 #16948
  - 修复 `INSERT VALUE` 中子查询引用父查询列导致插入值错误的问题 #16952
  - 修复对 Float 值进行 AND 操作结果不正的问题 #16666
  - 修复 Expensive Log 日志中 `WAIT_TIME` 字段信息错误的问题 #16907
  - 修复悲观事务模式执行语句 `SELECT FOR UPDATE` 不能被记录到 Slow Log 的问题 #16897
  - 修复在 Enum 或 Set 类型列上执行 `SELECT DISTINCT` 时结果错误的问题 #16892
  - 修复 `auto_random_base` 在 `SHOW CREATE TABLE` 的显示问题 #16864
  - 修复 `WHERE string_value` 结果不正确的问题 #16559
  - 修复 `GROUP BY` Window Function 错误消息和 MySQL 不一致问题 #16165
  - 修复 `FLASH TABLE` 语句在数据库名有大写字母时执行失败的问题 #17167
  - 修复 Projection 执行器内存消耗记录不准确的问题 #17118
  - 修复 `SLOW_QUERY` 表在不同时区下时间过滤不正确的问题 #17164
  - 修复 IndexMerge 和虚拟生成列一起使用时 Panic 的问题 #17126
  - 修复 `INSTR` 和 `LOCATE` 函数大小写问题 #17068
  - 修复开启 `tidb_allow_batch_cop` 配置后频繁出现 `tikv server timeout` 错误的问题 #17161
  - 修复 Float 类型进行 XOR 操作结果和 MySQL 8.0 不一致的问题 #16978
  - 修复 `ALTER TABLE REORGANIZE PARTITION` 不支持但执行未报错的问题 #17178
  - 修复 `EXPLAIN FORMAT="dot" FOR CONNECTION ID` 可能遇到不支持展示的 Plan 发生报错的问题 #17160
  - 修复 Prepared Statement 在 Statement Summary 表中 `EXEC_COUNT` 列的记录问题 #17086
  - 修复设置 Statement Summary 系统变量时未检查值是否合法的问题 #17129
  - 修复启用 Plan Cache 时使用越界值查询 `UNSIGNED BIGINT` 主键报错的问题 #17120
  - 修复 Grafana TiDB Summary 面板基于机器实例和请求类型展示 QPS 不正确的问题 #17105

- TiKV

  - 修复 restore 后生成大量空 Region 的问题 #7632
  - 修复 Raftstore 收到乱序 read index 响应时会引发 panic 的问题 #7370
  - 修复启用统一线程池时，不会验证 storage 或 `coprocessor read pool` 配置是否无效的问题 #7513
  - 修复 TiKV server 关闭时，join 可能 panic 的问题 #7713
  - 修复通过诊断 API 搜索慢日志无返回结果的问题 #7776
  - 修复节点长时间运行时，系统会产生较多内存碎片的问题 #7556
  - 修复部分情况下因存储无效日期导致 SQL 执行失败的问题 #7268
  - 修复从 GCS 进行恢复数据不能正确运行的问题 #7739
  - 修复存储加密时未进行 KMS Key Id 验证的问题 #7719
  - 修复 Coprocessor 在不同架构编译器下潜在正确性问题 #7714 #7730
  - 修复启用加密时出现 `snapshot ingestion` 错误的问题 #7815
  - 修复当改写配置文件时发生 `Invalid cross-device link` 的问题 #7817
  - 修复将配置文件写入到空文件中时会出现错误的 toml 格式的问题 #7817
  - 修复 Raftstore 中已销毁的 Peer 仍然可能处理请求的问题 #7836

- PD

  - 修复 pd-ctl 中使用 `region key` 命令时发生 404 错误的问题 #2399

  - 修复 Grafana 面板中缺失关于 TSO 和 ID 分配的监控的问题 #2405
  - 修复 Docker 镜像中不包含 `pd-recover` 的问题 #2406
  - 将数据目录的路径解析为绝对路径以解决 TiDB Dashboard 可能不能正确显示 PD 信息的问题 #2420
  - 修复在 pd-ctl 中使用 `scheduler config shuffle-region-scheduler` 命令时没有默认输出的问题 #2416

- TiFlash

  - 修复部分场景错误上报已使用空间信息的问题

- Tools

  - TiDB Binlog

    * 修复当下游为 Kafka 时对 `mediumint` 类型数据未处理的问题 #962
    * 修复当 DDL 中库名为关键字时 reparo 解析失败的问题 #961

  - TiCDC

    * 修复当环境变量 TZ 未设置时使用错误时区的问题 #512
    * 修复 owner 因为部分错误没有正确处理导致 server 退出时没有清理资源的问题 #528
    * 修复重连 TiKV 可能导致 TiCDC 阻塞的问题 #531
    * 优化初始化表结构时的内存使用 #534
    * 使用 watch 模式监听同步状态变更并进行准实时更新，减少同步延迟 #481

  - Backup & Restore (BR)

    * 修复 BR 恢复带有 `auto_random` 属性的表之后，插入数据有一定概率触发 duplicate entry 错误的问题 #241

16.18.19 TiDB 4.0 RC.1 Release Notes

发版日期：2020 年 4 月 28 日

TiDB 版本：4.0.0-rc.1

16.18.19.1 兼容性变化

- TiKV

  - 默认关闭 hibernate region #7618

- TiDB Binlog

  - Drainer 增加对 Sequence DDL 的支持 #950

16.18.19.2 重点修复的 Bug

- TiDB
  - 修复由于未检查 `MemBuffer`，事务内执行 `INSERT ... ON DUPLICATE KEY UPDATE` 语句插入多行重复数据可能出错的问题 #16689
  - 修复 lock 多行重复 keys 时导致数据索引不一致的问题 #16769
  - 修复回收空闲 gRPC 连接导致 TiDB panic 的问题 #16303
- TiKV
  - 修复 TiDB 探活请求触发的死锁问题 #7540
  - 修复事务的 min commit ts 可能溢出、影响数据正确性的问题 #7642
- TiFlash
  - 修正多数据路径配置下进行 rename table 会导致数据丢失的问题
  - 修正当 Region 处于 merge 状态时读取产生的数据错误
  - 修正当 Region 处于非 normal 状态时读取产生的数据错误
  - 修正 TiFlash 中表名的映射方式以正确支持 `recover table/flashback table`
  - 修正数据存储路径以解决 `rename table` 时潜在的数据丢失问题
  - 修正 Super batch 开启后，有一定概率 TiDB panic 的问题
  - 修正在线更新时的读模型以优化读性能
- TiCDC
  - 修复 TiCDC 内部维护的 schema 信息对于读写操作时序问题没有正确处理导致同步失败问题 #438 #450 #478 #496
  - 修复 TiKV client 遇到部分 TiKV 异常没有正确维护内部资源的 bug #499 #492
  - 修复节点异常残留元数据信息没有正确清理的 bug #488 #504
  - 修复 TiKV client 没有正确处理 prewrite 重复发送的问题 #446
  - 修复 TiKV client 没有正确处理在初始化前接收到冗余 prewrite 的问题 #448
- Backup & Restore (BR)
  - 修复关闭 checksum 情况下，仍然执行 checksum 的问题 #223
  - 修复 TiDB 开启 auto-random 或 alter-pk 时，增量备份失败的问题 #230 #231

16.18.19.3 新功能

- TiDB
  - 支持发送 `batch coprocessor` 请求给 TiFlash #16226
  - 默认打开 Coprocessor cache #16710
  - 在 SQL 语句的特殊注释中，只有被注册了语句片段才能被 parser 正常解析，否则将被忽略 #16157
  - 支持使用 `SHOW CONFIG` 语法显示 PD 和 TiKV 的配置 #16475
- TiKV

- 支持在备份到 S3 时使用用户提供的 KMS key 进行服务端加密 #7630
- 支持基于负载的 Region split #7623
- 支持 common name 验证 #7468
- 通过检查文件锁避免多个 TiKV 实例绑定相同的地址 #7447
- Encryption at rest 支持 AWS KMS #7465

- Placement Driver (PD)

  - 移除 `config manager` 以使其它组件自行控制它们的配置 #2349

- TiFlash

  - 添加 DeltaTree 引擎读写负载相关 metrics 上报
  - 缓存 `handle` 列和 `version` 列减小单次读请求的磁盘 I/O
  - 增加了 TiFlash 对于 `FromUnixTime` 和 `Date_format` 函数的支持
  - 根据第一块盘估算全局状态并上报
  - Grafana 添加 DeltaTree 引擎读写负载相关图表
  - 优化 TiFlash chunk encode decimal 的数据
  - 实现了 Diagnostics（SQL 诊断）的 gRPC API，以支持 `INFORMATION_SCHEMA.CLUSTER_INFO` 等系统表的查询

- TiCDC

  - 在 Kafka sink 模块提供发送批量消息支持 #426
  - 支持在 processor 内使用文件排序 #477
  - 增加自动 resolve lock 的支持 #459
  - 增加自动向 PD 设置 TiCDC 服务级别 safepoint 的功能 #487
  - 增加数据同步时的时区配置 #498

- Backup & Restore (BR)

  - 支持在存储的 URL 中配置 S3/GCS #246

16.18.19.4 Bug 修复

- TiDB

  - 修复系统表由于 unsigned 列定义导致无法正确显示负数的问题 #16004
  - 当使用 `use_index_merge` hint 包含不存在的索引时添加警告 #15960
  - 禁止多个 TiDB server 共享同一个临时目录 #16026
  - 修复打开 plan cache 时，执行 `explain for connection` 语句 panic 的问题 #16285
  - 修复显示 `tidb_capture_plan_baselines` 系统变量不正确的问题 #16048
  - 修复 `prepare` 语句中的 `group by` 语句解析错误的问题 #16377
  - 修复 `analyze primary key` 可能 panic 的问题 #16081
  - 修复 `cluster_info` 系统表中 TiFlash 节点信息错误的问题 #16024
  - 修复 index merge 可能 panic 的问题 #16360
  - 修复 index merge 遇到 generated column 时导致结果错误的问题 #16359
  - 修复 `show create table` 语句显示 sequence 默认值错误的问题 #16526

- 修复主键使用 sequence 作为默认值时出现 `not-null` 的错误的问题 #16510
- 修复当 TiKV 持续返回 `StaleCommand` 期间执行 SQL 卡住不报错的问题 #16530
- 修复 `CREATE DATABASE` 时仅指定 `COLLATE` 时会报错的问题，同时在 `SHOW CREATE DATABASE` 结果中添加缺失的 `COLLATE` 部分 #16540
- 修复打开 plan-cache 时，分区裁剪失败的问题 #16723
- 修复点查在 handle 溢出时返回错误结果的问题 #16755
- 修复等值时间查询 `slow_query` 系统表返回错误结果的问题 #16806

- TiKV

  - 解决 OpenSSL 的安全性问题：CVE-2020-1967 #7622
  - 避免将 BatchRollback 产生的 rollback 记录标为保护的记录以改善冲突较多场景下的性能 #7604
  - 修复锁竞争严重的场景下，不必要地唤醒事务导致多余的重试和性能下降的问题 #7551
  - 修复多次 merge 时，Region 可能卡住的问题 #7518
  - 修复删除 learner 时，learner 可能并未被删除的问题 #7518
  - 修复 follower read 可能使 raft-rs panic 的问题 #7408
  - 修复 SQL 可能因 “group by constant” 错误失败的问题 #7383
  - 修复当一个乐观锁对应的 primary lock 是悲观锁时，该乐观锁可能阻塞读的问题 #7328

- Placement Driver (PD)

  - 修复部分 API TLS 认证失败的问题 #2363
  - 修复 `config` API 不能识别带有前缀的配置项的问题 #2354
  - 修复找不到 scheduler 时会返回 500 错误的问题 #2328
  - 修复 `scheduler config balance-hot-region-scheduler list` 命令返回 404 错误的问题 #2321

- TiFlash

  - 禁用存储引擎的粗糙索引优化
  - 修正对 Region 进行 resolve lock 时遇到需要跳过的 lock 抛异常的问题
  - 修正 Coprocessor 统计信息收集的 NPE
  - 修正 Region meta 的检查以保证 Region Split/Region Merge 流程的正确性
  - 修正对 Coprocessor response 大小未做估算导致的消息大小超过 gRPC 限制的问题
  - 修正 TiFlash 对 `AdminCmdType::Split` 命令的处理

16.18.20 TiDB 4.0 RC Release Notes

发版日期：2020 年 4 月 8 日

TiDB 版本：4.0.0-rc

TiUP 版本：0.0.3

> **警告：**
>
> 该版本存在一些已知问题，已在新版本中修复，建议使用 4.0.x 的最新版本。

16.18.20.1 兼容性变化

- TiDB
  - 当 tidb-server 状态端口被占用时由原来打印一条告警日志改成拒绝启动 #15177
- TiKV
  - 悲观事务支持 pipelined 功能，TPC-C 性能提升 20%，风险：pipelined 功能可能会在执行阶段加锁不成功导致事务提交失败 #6984
  - 调整 unify-read-pool 配置项的方式，仅在新部署的集群时默认启用，旧集群保持原来的方式 #7059
- Tools
  - TiDB Binlog
    * 新增验证 Common Name 配置项目的功能 #934

16.18.20.2 重点修复的 Bug

- TiDB
  - 修复 DDL 采用 PREPARE 语句执行时，由于内部记录的 job query 不正确，导致上下游同步可能出错的问题 #15435
  - 修复 Read Committed 隔离级别下，子查询的输出结果可能不正确的问题 #15471
  - 修复 Inline Projection 优化所导致的结果错误问题 #15411
  - 修复某些情况下 SQL Hint `INL_MERGE_JOIN` 未正确执行的问题 #15515
  - 修复向 `AutoRandom` 列显式写入负数时，`AutoRandom` 列会 Rebase 的问题 #15397

16.18.20.3 新功能

- TiDB
  - 新增大小写不敏感的排序规则，用户可在新集群上启用 `utf8mb4_general_ci` 和 `utf8_general_ci` #33
  - 增强 `RECOVER TABLE` 语法，现在该语法支持恢复被 Truncate 的表 #15398
  - 当 tidb-server 状态端口被占用时由原来打印一条告警日志改成拒绝启动 #15177
  - 优化使用 Sequence 作为列的默认值时的写入性能 #15216
  - 新增 `DDLJobs` 系统表，用于查询 DDL 任务详细信息 #14837
  - 优化 aggFuncSum 的性能 #14887
  - 优化 `EXPLAIN` 的输出结果 #15507
- TiKV
  - 悲观事务支持 pipelined 功能，TPC-C 性能提升 20%，风险：pipelined 功能可能会在执行阶段加锁不成功导致事务提交失败 #6984
  - HTTP 端口支持 TLS #5393
  - 调整 unify-read-pool 配置项的方式，仅在新部署的集群时默认启用，旧集群保持原来的方式 #7059

- PD
  - 新增通过 HTTP 接口获取 PD 默认配置信息功能 #2258
- Tools
  - TiDB Binlog
    * 新增验证 Common Name 配置项目的功能 #934
  - TiDB Lightning
    * 优化 TiDB Lightning 的性能 #281 #275

16.18.20.4 Bug 修复

- TiDB
  - 修复 DDL 采用 PREPARE 语句执行时，由于内部记录的 job query 不正确，导致上下游同步可能出错的问题 #15435
  - 修复 Read Committed 隔离级别下，子查询的输出结果可能不正确的问题 #15471
  - 修复 `INSERT ... VALUES` 指定 `BIT(N)` 类型数据时可能报错的问题 #15350
  - 修复 DDL Job 内部重试时，`ErrorCount` 的值没有被正确累加导致未完全达到重试预期的问题 #15373
  - 修复 TiDB 连接 TiFlash 时，垃圾回收可能工作不正常的问题 #15505
  - 修复 Inline Projection 优化所导致的结果错误问题 #15411
  - 修复某些情况下 SQL Hint `INL_MERGE_JOIN` 未正确执行的问题 #15515
  - 修复向 AutoRandom 列显式写入负数时，AutoRandom 列会 Rebase 的问题 #15397
- TiKV
  - 修复启用 Follower Read 功能，由于 transfer leader 导致系统 Panic 的问题 #7101
- Tools
  - TiDB Lightning
    * 修复 backend 是 TiDB 时由于字符转换错误导致数据错误的问题 #283
  - TiCDC
    * 修复 MySQL sink 执行 DDL 时，若下游没有 test 库系统报错的问题 #353
    * CDC cli 新增实时交互模式功能 #351
    * 同步数据时增加对上游表是否可同步的检查 #368
    * 新增异步写入 Kafka 的功能 #344

16.18.21 TiDB 4.0.0 Beta.2 Release Notes

发版日期：2020 年 3 月 18 日

TiDB **版本：**4.0.0-beta.2

TiDB Ansible **版本：**4.0.0-beta.2

16.18.21.1 兼容性变化

- Tools
  - TiDB Binlog
    * 修复 Drainer 配置 `disable-dispatch`、`disable-causality` 时系统直接报错并退出的问题 #915

16.18.21.2 新功能

- TiKV
  - 支持将动态修改配置的结果持久化存储到硬盘 #6684
- PD
  - 支持将动态修改配置的结果持久化存储到硬盘 #2153
- Tools
  - TiDB Binlog
    * 新增 TiDB 集群之间数据双向复制功能 #879 #903
  - TiDB Lightning
    * 新增配置 TLS 功能 #40 #270
  - 新增 TiCDC 工具，提供以下功能：
    * 捕捉 TiKV 变化的数据，同步到下游 Kafka、MySQL 协议的数据库
    * 确保数据最终一致性，若下游是 Kafka，也可确保行级别的有序
    * 提供进程级别的高可用能力
  - BR
    * 开启增量备份、支持将备份文件存储在 AWS S3 等实验性功能 #175
- TiDB Ansible
  - 新增将节点信息注册到 etcd 的功能 #1196
  - 新增支持在 ARM 平台上部署 TiDB 服务的功能 #1204

16.18.21.3 Bug 修复

- TiKV
  - 修复 backup 在遇到空的 short value 时可能 panic 的问题 #6718
  - 修复 Hibernate Region 在某些特殊条件下未被正确唤醒的问题 #6772 #6648 #6376
- PD
  - 修复因 rule checker 在给 Region 分配 store 失败导致系统 panic 的问题 #2160
  - 修复启用动态修改配置功能后，配置可能在切换 leader 时有同步延迟的问题 #2154
- Tools
  - BR
    * 修复因 PD 无法处理过大消息导致在数据规模较大时恢复失败的问题 #167
    * 修复因 BR 与 TiDB 版本不兼容导致 BR 运行失败的问题 #186
    * 修复因 BR 与 TiFlash 不兼容导致 BR 运行失败的问题 #194

16.18.22 TiDB 4.0.0 Beta.1 Release Notes

发版日期：2020 年 2 月 28 日

TiDB **版本：**4.0.0-beta.1

TiDB Ansible **版本：**4.0.0-beta.1

16.18.22.1 兼容性变化

- TiDB
  - 修改配置项 `log.enable-slow-log` 的类型，由整数型改为布尔类型 #14864
  - 调整修改系统表 `mysql.user` 中 `password` 列名为 `authentication_string`，与 MySQL 5.7 保持一致（该变动会导致升级后不能回退）#14598
  - `txn-total-size-limit` 配置项的默认值由 1GB 调整为 100MB #14522
  - 新增动态修改、更新配置项的功能，配置项由 PD 持久化存储 #14750 #14303 #14830
- TiKV
  - 新增 `readpool.unify-read-pool` 配置项，默认值为 `True`，用于控制点查是否共用 Coprocessor 的处理线程 #6375 #6401 #6534 #6582 #6585 #6593 #6597 #6677
- PD
  - 优化 HTTP API 兼容以新的配置项管理方式 #2080
- TiDB Lightning
  - 优化配置项，部分配置项在没有设置的时候使用默认配置 #255
- TiDB Ansible
  - 将 `theflash` 更名为 `tiflash` #1130
  - 优化 TiFlash 配置文件中的默认值及相关配置 #1138

16.18.22.2 新功能

- TiDB
  - 慢日志系统表 `SLOW_QUERY / CLUSTER_SLOW_QUERY` 支持查询任意时间段的日志#14840 #14878
  - 支持 SQL 性能诊断功能
    * #14843 #14810 #14835 #14801 #14743
    * #14718 #14721 #14670 #14663 #14668
    * #14896
  - 新增 Sequence 功能 #14731 #14589 #14674 #14442
  - 新增动态修改、更新配置项的功能，配置项由 PD 持久化存储 #14750 #14303 #14830
  - 新增系统自动根据负载均衡策略从不同角色上读取数据的功能，且新增 `leader-and-follower` 系统变量用于控制开启此功能 #14761
  - 新增 `Coercibility` 函数 #14739
  - 支持在分区表上建立 TiFlash 副本 #14735 #14713 #14644

  - 完善 SLOW_QUERY 表的权限检查 #14451
  - 新增当 Join 时若内存不足时系统自动将中间结果写磁盘文件的功能 #14708 #14279
  - 新增通过查询 information_schema.PARTITIONS 系统表查看 partition 详细信息的功能 #14347
  - 新增 json_objectagg 聚合函数 #11154
  - 新增审计日志记录用户登录失败的功能 #14594
  - 新增 max-server-connections 配置项，默认值为 4096 ，用于控制单个服务器连接数 #14409
  - 支持隔离读在 Server 级别指定多个存储引擎 #14440
  - 优化 Apply 算子和 Sort 算子的代价估算模型，提升系统稳定性 #13550 #14708

- TiKV

  - 新增通过 HTTP API 从 status 端口获取配置项的功能 #6480
  - 提升 Coprocessor 的 Chunk Encoder 的性能 #6341

- PD

  - 新增通过 UI 访问集群热点数据分布功能 #2086
  - 新增收集集群组件的启动时间的功能 #2116
  - member API 返回信息新增部署路径和组件版本信息 #2130
  - pd-ctl 新增 component 命令用于修改、查看组件配置信息 (experimental) #2092

- TiDB Binlog

  - 同步链路新增 TLS 功能 #904 #894
  - Drainer 新增 kafka-client-id 配置项，支持连接 Kafka 客户端配置客户端 ID #902
  - Drainer 新增清理增量备份文件的功能 #885

- TiDB Ansible

  - 新增同一个集群中部署多个 Grafana/Prometheus/Alertmanager 的功能 #1142
  - TiFlash 配置文件新增 metrics_port 配置项，默认值为8234 #1145
  - TiFlash 配置文件新增 flash_proxy_status_port 配置项，默认值为 20292 #1141
  - 新增 TiFlash 监控 Dashboard #1147 #1151

16.18.22.3 Bug 修复

- TiDB

  - 修复创建视图时，列名超过 64 个字符时将报错的问题，报错改为重命名过长的列名 #14850
  - 修复因 create or replace view 语句处理逻辑不正确导致 information_schema.views 中有重复数据的问题 #14832
  - 修复开启 plan cache 之后，BatchPointGet 的获取到错误数据的问题 #14855
  - 修复按照时间分区的分区表，在修改时区后，因处理逻辑不正确导致数据插入到错误分区表的问题 #14370
  - 修复 IsTrue 函数的表达式因名称不正确，在执行外连化简利用非法函数名重建表达式导致系统 panic 的问题 #14515
  - 修复 show binding 语句权限检查不正确的问题 #14443

- TiKV

  - 修复 CAST 函数在 TiDB 和 TiKV 中行为不一致性的问题 #6463 #6461 #6459 #6474 #6492 #6569

- TiDB Lightning
  - 修复在非 server mode 模式下 web 界面无法打开的问题 #259

16.18.23 TiDB 4.0 Beta Release Notes

发版日期：2020 年 1 月 17 日

TiDB 版本：4.0.0-beta

TiDB Ansible 版本：4.0.0-beta

16.18.23.1 TiDB

- 当 `Insert/Replace/Delete/Update` 在执行过程中所使用的内存空间超过启动配置项 `MemQuotaQuery` 的限制时，输出日志或取消本次执行过程，具体行为取决于启动配置项 `OOMAction` #14179 #14289 #14299
- 估算 `Index Join` 的代价时由仅考虑驱动表的行数调整为考虑驱动表和被驱动表的行数，提升估算的准确性 #12085
- 新增 15 个 SQL hint，用于控制优化器行为，提升优化器稳定性
  - #11253 #11364 #11673 #11740 #11746
  - #11809 #11996 #12043 #12059 #12246
  - #12382
- 提升查询中所涉及到的列能被索引全覆盖时的性能 #12022
- 对表上的 OR 表达式过滤条件，使用多个索引组合进行表访问，提升查询性能 #10121 #10512 #11245 #12225 #12248 #12305 #12843
- 优化 Range 计算流程，缓存并去重索引计算的结果，减少 CPU 开销，提升 range 计算的性能 #12856
- Slow Log 日志的级别与普通日志的级别解耦 #12359
- 新增 `oom-use-tmp-storage` 参数，默认值为 `true`，用于控制当单条 SQL 执行过程中占用内存使用超过 `mem-quota-query` 且 SQL 中包含 `Hash Join` 时，系统会采用临时文件来缓存中间结果 #11832 #11937 #12116 #12067
- 支持使用 `create index/alter table add index` 语句创建表达式索引，使用 `drop index` 语句删除表达式索引 #14117
- `query-log-max-len` 参数默认值调大为 `4096`，减少输出被截断 SQL 的数量，此参数可通过 SQL 动态调整 #12491
- 支持在列属性上添加 `AutoRandom` 关键字，用于控制系统自动为主键分配随机整数，避免 `AutoIncrement` 自增主键带来的写入热点问题 #13127
- 支持表级锁 (Table Locks) #11038
- `ADMIN SHOW DDL JOBS` 支持 `LIKE` 或 `WHERE` 语法进行条件过滤 #12484
- `information_schema.tables` 表新增 `TIDB_ROW_ID_SHARDING_INFO` 列，输出列的 RowID 打散相关的信息（例如：表 A 指定 `SHARD_ROW_ID_BITS`，该列的值为 `"SHARD_BITS={bit_number}"`）#13418
- 优化 SQL 错误信息的错误码，避免出现多个错误信息的错误码都是 `ERROR 1105 (HY000)`（即类型为 `Unknown Error`）的情况
  - #14002 #13874 #13733 #13654 #13646
  - #13540 #13366 #13329 #13300 #13233
  - #13033 #12866 #14054

- 在估算行数时将离散类型的很窄的 range 转化为 `point set` 然后用 CM-Sketch 提升估算精度 #11524
- 支持普通 `Analyze` 得到的 CM-Sketch 维护 TopN 信息，将出现次数较多的值单独维护 #11409
- 支持动态调整 CM-Sketch 长宽和 TopN 数目 #11278
- 新增 SQL Binding 的自动捕获和自动演进功能 #13199 #12434
- 优化与 TiKV 之间通信息编码格式，采用 Chunk 格式编码，提升网络通信性能 #12023 #12536 #12613 #12621 #12899 #13060 #13349
- 支持新的行存储格式，提升宽表性能 #12634
- 优化 `Recover Binlog` 接口，确保等待当前正在提交的事务都提交完成再返回 #13740
- 新增通过 HTTP `info/all` 接口，查询集群中所有 TiDB server 开启 binlog 的状态 #13025
- 新增在事务模式是悲观事务时，支持使用 MySQL 兼容的 Read Committed 事务隔离级别 #14087
- 支持超大事务，事务大小的限制受限于物理内存大小
  - #11999 #11986 #11974 #11817 #11807
  - #12133 #12223 #12980 #13123 #13299
  - #13432 #13599
- 提升 `Kill` 稳定性 #10841
- `LOAD DATA` 支持十六进制和二进制表达式作为分隔符 #11029
- `IndexLookupJoin` 拆分为 `IndexHashJoin` 与 `IndexMergeJoin`，提升 `IndexLookupJoin` 的执行性能，减少执行过程中的内存消耗 #8861 #12139 #12349 #13238 #13451 #13714
- 修复 RBAC 若干问题 #13896 #13820 #13940 #14090 #13940 #13014
- 修复创建视图时，由于 `select` 语句包含 `union` 视图无法创建成功的问题 #12595
- 修复 CAST 函数若干问题
  - #12858 #11968 #11640 #11483 #11493
  - #11376 #11355 #11114 #14405 #14323
  - #13837 #13401 #13334 #12652 #12864
  - #12623 #11989
- Slow log 输出 TiKV RPC 的 `backoff` 具体信息，方便排查问题 #13770
- 优化统一 expensive log 中内存统计信息的格式 #12809
- 优化 `explain` 显式格式，支持输出算子占用内存和磁盘的信息 #13914 #13692 #13686 #11415 #13927 #13764 #13720
- 优化 `LOAD DATA` 重复值检查，按照事务粒度进行且事务大小可通过 `tidb_dml_batch_size` 配置 #11132
- 优化 `LOAD DATA` 性能，将数据读取处理和批量提交分离，且分派到不同的 Worker 处理 #11533 #11284

16.18.23.2 TiKV

- 升级 RocksDB 的版本到 6.4.6
- 系统启动时自动创建 2GB 大小的空文件，解决磁盘空间被写满时系统无法正常进行 Compaction 的问题 #6321
- 新增快速备份恢复功能
  - #6462 #6395 #6378 #6374 #6349
  - #6339 #6308 #6295 #6286 #6283
  - #6261 #6222 #6209 #6204 #6202
  - #6198 #6186 #6177 #6146 #6071
  - #6042 #5877 #5806 #5803 #5800

  - #5781 #5772 #5689 #5683

- 新增从 Follower 副本读取数据的功能

  - #5051 #5118 #5213 #5316 #5401
  - #5919 #5887 #6340 #6348 #6396

- 提升 TiDB 通过索引读取数据的性能 #5682
- 修复 CAST 函数在 TiDB 和 TiKV 中行为不一致性的问题

  - #6459 #6461 #6458 #6447 #6440
  - #6425 #6424 #6390 #5842 #5528
  - #5334 #5199 #5167 #5146 #5141
  - #4998 #5029 #5099 #5006 #5095
  - #5093 #5090 #4987 #5066 #5038
  - #4962 #4890 #4727 #6060 #5761
  - #5793 #5468 #5540 #5548 #5455
  - #5543 #5433 #5431 #5423 #5179
  - #5134 #4685 #4650 #6463

16.18.23.3 PD

- 新增根据存储节点负载信息优化热点调度的功能

  - #1870 #1982 #1998 #1843 #1750

- 新增 Placement Rules 功能，通过组合不同的调度规则，精细控制任意一段数据的副本的数量、存放位置、存储主机类型、角色等信息

  - #2051 #1999 #2042 #1917 #1904
  - #1897 #1894 #1865 #1855 #1834

- 支持插件功能 (experimental) #1799
- 新增调度器支持自定义配置功能，支持配置调度器的作用范围 (experimental) #1735 #1783 #1791
- 新增根据集群负载信息自动调整调度速度的功能（experimental，默认不打开）#1875 #1887 #1902

16.18.23.4 Tools

- TiDB Lightning

  - 命令行增加配置下游数据库密码的参数 #253

16.18.23.5 TiDB Ansible

- 下载包增加 Checksum 检查，防止下载到不完整的包 #1002
- 新增检查 systemd 版本功能，systemd 版本最低要求 `systemd-219-52` #1020 #1074
- 修复 TiDB Lightning 启动时未正确创建日志目录的问题 #1103
- 修复 TiDB Lightning 自定义端口不生效的问题 #1107
- 新增支持部署运维 TiFlash 的功能 #1119

16.19 v3.1

16.19.1 TiDB 3.1.2 Release Notes

发版日期：2020 年 6 月 4 日

TiDB 版本：3.1.2

16.19.1.1 Bug 修复

- TiKV
  - 修复 S3 和 GCS 备份恢复时的错误处理问题 #7965
  - 修复备份过程中的 `DefaultNotFound` 错误 #7838
- Tools
  - Backup & Restore (BR)
    * 提升备份恢复到 S3 和 GCS 存储的稳定性，在网络较差时会自动重试 #314 #7965
    * 修复恢复数据时因找不到 Region leader 出现的 `NotLeader` 错误，BR 会自动重试 #303
    * 修复恢复数据时 `rowID` 大于 `2^(63)` 的数据丢失问题 #323
    * 修复恢复数据时无法恢复空库空表的问题 #318
    * 增加备份恢复 S3 时的 AWS KMS 服务端加密 (SSE) 支持 #261

16.19.2 TiDB 3.1.1 Release Notes

发版日期：2020 年 4 月 30 日

TiDB 版本：3.1.1

TiDB Ansible 版本：3.1.1

16.19.2.1 新功能

- TiDB
  - 添加 `auto_rand_base` 的 table option #16812
  - 添加 `Feature ID` 注释：在 SQL 语句的特殊注释中，只有被注册了语句片段才能被 parser 正常解析，否则将被忽略 #16155
- TiFlash
  - 缓存 `handle` 列和 `version` 列减小单次读请求的磁盘 I/O
  - Grafana 添加 DeltaTree 引擎读写负载相关图表
  - 优化 TiFlash chunk encode decimal 数据的流程
  - TiFlash 低负载时，减少打开的文件描述符数量

16.19.2.2 Bug 修复

- TiDB
  - 修复实例级别的隔离读设置不生效的问题，以及 TiDB 升级后隔离读设置被不正确保留的问题 #16482 #16802
  - 修复 hash 分区表上面的分区选择语法，现在 `partition(P0)` 这样的语法不会报错 #16076
  - 修复若 update sql 中包含 view，但不会对 view 进行 update，update 语句仍然报错的问题#16789
  - 修复对查询最内层的 `not not` 消除而造成结果错误的问题 #16423
- TiFlash
  - 修复当 Region 处于非 normal 状态时读取产生的数据错误
  - 修复 TiFlash 中表名的映射方式以正确支持 `recover table/flashback table`
  - 修复数据存储路径以解决 `rename table` 时潜在的数据丢失问题
  - 修复在线更新时的读模型以优化读性能
  - 修复 database/table name 含特殊字符，升级后无法正常启动的问题
- Tools
  - Backup & Restore (BR)
    * 修复 BR 恢复带有 auto_random 属性的表之后，插入数据有一定概率触发 duplicate entry 错误的问题 #241

16.19.3 TiDB 3.1 GA Release Notes

发版日期：2020 年 4 月 16 日

TiDB 版本：3.1.0 GA

TiDB Ansible 版本：3.1.0 GA

16.19.3.1 兼容性变化

- TiDB
  - 支持 TiDB 在启动服务时，在开启 `report-status` 配置项情况下，如果发现 HTTP 监听端口不可用，则直接退出启动 #16291
- Tools
  - Backup & Restore (BR)
    * BR 不支持在 3.1 GA 版本之前的 TiKV 集群上进行恢复 #233

16.19.3.2 新功能

- TiDB
  - 支持在 `explain format = "dot"` 中展示 coprocessor 任务的信息 #16125
  - 通过 `disable-error-stack` 配置项减少日志的冗余 stack 信息 #16182
- Placement Driver (PD)
  - 优化热点 Region 调度 #2342
- TiFlash
  - 添加上报 DeltaTree 引擎读写负载相关 metrics 信息
  - 支持 `fromUnixTime` 和 `dateFormat` 函数下推
  - 默认禁用粗粒度索引过滤器
- TiDB Ansible
  - 新增 TiFlash 监控 #1253 #1257
  - 优化 TiFlash 配置参数 #1262 #1265 #1271
  - 优化 TiDB 启动脚本 #1268

16.19.3.3 Bug 修复

- TiDB
  - 修复 merge join 在某些场景下 panic 的问题 #15920
  - 修复在计算选择率时重复考虑某些表达式的问题 #16052
  - 修复极端情况下 load 统计信息可能出现的 panic 的问题 #15710
  - 修复 SQL query 中存在等价表达式在某些情况下无法识别导致报错的问题 #16015
  - 修复从一个数据库中查询另一个数据库的 `view` 时报错的问题 #15867
  - 修复 fast analyze handle 列时 panic 的问题 #16080
  - 修复 `current_role` 输出结果字符集不正确的问题 #16084
  - 完善 MySQL 连接握手错误相关日志 #15799
  - 修复加载审计插件后端口探测活动导致 panic 的问题 #16065
  - 修复因 `TypeNull` 类被错误识别为变长类型，导致 left join 上的 `sort` 算子 panic 的问题 #15739
  - 修复监控 session 重试错误计数不准确的问题 #16120
  - 修复在 `ALLOW_INVALID_DATES` 模式下，weekday 结果出错的问题 #16171
  - 修复在集群中存在 TiFlash 节点时，GC 可能不能正常工作的问题 #15761
  - 修复创建 hash 分区表时指定非常大的分区数量导致 TiDB OOM 的问题 #16219
  - 让 `union` 语句的行为和 `select` 语句保持相同，修复把 warnings 当 error 的问题 #16138
  - 修复 TopN 下推到 mocktikv 中的执行错误 #16200
  - 增大 `chunk.column.nullBitMap` 的初始化长度，以避免多余的 `runtime.growslice` 开销 #16142
- TiKV
  - 修复 replica read 导致 TiKV panic 的问题 #7418 #7369

  - 修复 restore 产生许多空 Region 的问题 #7419
  - 修复重复的 resolve lock 请求可能会破坏悲观事务原子性的问题 #7389

- TiFlash

  - 修复从 TiDB 同步 schema 时，进行 `rename table` 时潜在的问题
  - 修复多数据路径配置下进行 `rename table` 会导致数据丢失的问题
  - 修复某些场景下 TiFlash 存储空间上报错误的问题
  - 修复开启 Region Merge 情况下从 TiFlash 读取时潜在的问题

- Tools

  - TiDB Binlog

    * 修复因为 TiFlash 相关的 DDL job 导致 Drainer 同步中断的问题 #948 #942

  - BR

    * 修复关闭 checksum 情况下，仍然执行 checksum 的问题 #223
    * 修复 TiDB 开启 auto-random 或 alter-pk 时，增量备份失败的问题 #230 #231

16.19.4 TiDB 3.1 RC Release Notes

发版日期：2020 年 4 月 2 日

TiDB 版本：3.1.0-rc

TiDB Ansible 版本：3.1.0-rc

> **警告：**
>
> 该版本存在一些已知问题，已在新版本中修复，建议使用 3.1.x 的最新版本。

16.19.4.1 新功能

- TiDB

  - 采用的二分搜索实现分区裁剪，以来提升性能 #15678
  - 支持 RECOVER 语法恢复被 `truncate table` 删除的数据 #15460
  - 支持重用语句重试中已分配的 AUTO_RANDOM ID #15393
  - 支持 `recover table` 恢复 AUTO_RANDOM ID 分配器的状态 #15393
  - 支持 YEAR、MONTH、TO_DAY 函数作为 Hash partition table 的分区 key #15619
  - 只在读到数据，需要加锁的时候，才对表做 schema-change 的检查 #15708
  - 为 session 变量 `tidb_replica_read` 增加 `leader-and-follower` 值，实现读请求在 leader 和 follower 直接负载均衡 #15721
  - 支持 TiDB 在每次新建连接时动态更新 TLS 证书，实现不重启更新过期客户端证书 #15163
  - 通过更新 PD Client 支持每次新建连接是读取加载最新的证书 #15425

  - 如果配置了 `Cluster-SSL-*` 强制让 TiDB-PD 和 TiDB-TiDB 使用配置的证书进行 HTTPS 协议传输 #15430
  - 新增和 MySQL 兼容的 `--require-secure-transport` 启动项，配置时强制客户端使用 TLS #15442
  - 添加 `cluster-verify-cn` 配置，只有拥有特定 CN 属性值证书的访问者才能访问 TiDB Status Port 或建立 gRPC 连接 #15137

- TiKV

  - 支持通过 Raw KV API 备份数据 #7051
  - 状态服务支持 TLS #7142
  - KV server 支持 TLS #7305
  - 优化持有锁的时间以提升备份性能 #7202

- PD

  - `shuffle-region-scheduler` 支持调度 learner #2235
  - pd-ctl 增加配置 Placement Rules 的命令 #2306

- Tools

  - TiDB Binlog

    * 同步链路新增 TLS 功能 #931 #937 #939
    * Drainer 新增 `kafka-client-id` 配置项，支持连接 Kafka 客户端配置客户端 ID #929

  - TiDB Lightning

    * 优化 Lightning 的性能 #281 #275
    * 支持 TLS #270

  - BR

    * 优化日志输出信息，对用户更友好 #189

- TiDB Ansible

  - 优化 TiFlash 数据目录创建的方式 #1242
  - TiFlash 新增 `Write Amplification` 监控项 #1234
  - 优化 CPU epollexclusive 检查失败时提示信息，包括：通过升级内核版本解决，且提示支持的最小内核版本 #1243

16.19.4.2 Bug 修复

- TiDB

  - 修复由于 update tiflash replica 类型的 DDL 太频繁导致的 information schema changed 错误的问题 #14884
  - 修复在使用 AUTO_RANDOM 时，未正确生成的 `last_insert_id` 的问题 #15149
  - 修复更新 TiFlash replica 状态时可能导致 DDL 卡住的问题 #15161
  - 当存在谓词无法下推时，禁止聚合下推和 TopN 下推 #15141
  - 禁止相互嵌套地创建 `view` #15440
  - 修复 `set role all` 后执行 `select current_role` 报错的问题 #15570
  - 修复查询中指定列的 `view` 名时，报不识别 `view` 的问题 #15573

  - 修复预处理 DDL 语句在写 binlog 信息时可能出错的问题 #15444
  - 修复同时访问视图和分区表时导致 panic 的问题 #15560
  - 修复 `update duplicate key` 语句中 `bit(n)` 类型的 column 报错的问题 #15487
  - 修复 `max-execution-time` 部分场景下不生效的问题 #15616
  - 修复在生成 Index 计划时未判断当前的 ReadEngine 中是否包含 TiKV 的问题 #15773

- TiKV

  - 修复在关闭一致性检查参数时，事务中插入已存在的 Key 且立马删除导致冲突检测失效或数据索引不一致的问题 #7112
  - 修复 TopN 比较无符号整型时计算错误的问题 #7199
  - Raftstore 引入流控机制，解决没有流控可能导致追日志太慢可能导致集群卡住，以及事务大小太大会导致 TiKV 间连接频繁重连的问题 #7087 #7078
  - 修复发送到 replicas 的读请求可能被永久卡住的问题 #6543
  - 修复 replica read 会被 apply snapshot 阻塞的问题 #7249
  - 修复 read index 在 transfer leader 情况下可能导致 panic 的问题 #7240
  - 修复备份到 S3 时所有 SST 文件填充为零的问题 #6967
  - 修复备份时未记录 SST 文件大小的导致恢复后有很多空 Region 的问题 #6983
  - 备份支持 AWS IAM web identity #7297

- PD

  - 修复 PD 因处理 Region heartbeat 时的数据竞争导致 Region 信息不正确的问题 #2234
  - 修复 `random-merge-scheduler` 未遵守 location labels 和 Placement Rules 规则的问题 #2212
  - 修复 Placement Rule 被具有相同 `startKey` 和 endKey 的 Placement Rule 覆盖的问题 #2222
  - 修复 API 输出的版本号与 PD server 输出版本号不一致的问题 #2192

- Tools

  - TiDB Lightning

    * 修复 backend 是 TiDB 时由于字符转化错误导致数据错误的问题 #283

  - BR

    * 修复了在开启 TiFlash 集群中，无法使用 BR 恢复的问题 #194

16.19.5 TiDB 3.1 Beta.2 Release Notes

发版日期：2020 年 3 月 9 日

TiDB 版本：3.1.0-beta.2

TiDB Ansible 版本：3.1.0-beta.2

> **警告：**
>
> 该版本存在一些已知问题，已在新版本中修复，建议使用 3.1.x 的最新版本。

16.19.5.1 兼容性变化

- Tools
  - TiDB Lightning
    * 优化配置项，部分配置项在没有进行配置的时候使用TiDB Lightning 配置参数中的默认配置 #255
    * 新增 `--tidb-password` 命令行参数，用于设置 TiDB 的密码 #253

16.19.5.2 新功能

- TiDB
  - 支持在列属性上添加 `AutoRandom` 关键字，控制系统自动为主键分配随机整数，避免 `AUTO_INCREMENT` 自增主键带来的写入热点问题 #14555
  - 新增通过 DDL 语句为表创建、删除列存储副本的功能 #14537
  - 新增优化器可自主选择不同的存储引擎的功能 #14537
  - 新增 SQL Hint 支持不同的存储引擎的功能 #14537
  - 新增通过 `tidb_replic_read` 系统变量从 Follower 上读取数据的功能 #13464
- TiKV
  - Raftstore
    * 新增 `peer_address` 参数，为其他类型的服务提供通过不同端连接此 TiKV server 的能力 #6491
    * 新增 `read_index` 和 `read_index_resp` 监控项，用于监控 `ReadIndex` 请求数 #6610
  - PD Client
    * 新增将本地线程统计信息汇报给 PD 的功能 #6605
  - Backup
    * 用 Rust 的 `async-speed-limit` 流控库替代 `RocksIOLimiter` 流控库，避免备份时拷贝多次内存的问题 #6462
- PD
  - 新增 location label 的名字中允许使用斜杠 `/` 的功能 #2084
- TiFlash
  - 初始版本
- TiDB Ansible
  - 新增同一个集群中部署多个 Grafana/Prometheus/Alertmanager 的功能 #1143
  - 新增部署 TiFlash 组件的功能 #1148
  - 新增 TiFlash 组件相关的监控指标 #1152

16.19.5.3 Bug 修复

- TiKV
  - Raftstore
    * 修复静默 Region 读数据处理不当导致无法处理读请求的问题 #6450

        * 修复 ReadIndex 在 leader 切换时可能导致系统 panic 的问题 #6613
        * 修复 Hibernate Region 在某些特殊条件下未被正确唤醒的问题 #6730 #6737 #6972
    - Backup
        * 修复备份数据时备份了多余的数据，导致恢复数据时数据索引不一致的问题 #6659
        * 修复备份时因处理已被删除的值逻辑不正确导致系统 panic 的问题 #6726
- PD
    - 修复因 rule checker 给 Region 分配 store 失败导致系统 panic 的问题 #2161
- Tools
    - TiDB Lightning
        * 修复在非 Server mode 模式下 web 界面无法打开的问题 #259
    - BR
        * 修复在恢复数据过程中遇到不可恢复的错误时，程序无法及时退出的问题 #152
- TiDB Ansible
    - 修复在某些场景下获取不到 PD Leader 导致滚动升级命令执行失败的问题 #1122

16.19.6 TiDB 3.1 Beta.1 Release Notes

发版日期：2020 年 1 月 10 日

TiDB 版本：3.1.0-beta.1

TiDB Ansible 版本：3.1.0-beta.1

16.19.6.1 TiKV

- backup
    - 备份文件的名称由 `start_key` 改为 `start_key` 的 hash 值，减少文件名的长度，方便阅读 #6198
    - 关闭 RocksDB `force_consistency_checks` 检查功能，避免一致性检查误报的问题 #6249
    - 新增增量备份功能 #6286
- sst_importer
    - 修复恢复后 SST 文件没有 MVCC Properties 的问题 #6378
    - 新增 `tikv_import_download_duration`、`tikv_import_download_bytes`、`tikv_import_ingest_duration` ↪ 、`tikv_import_ingest_bytes`、`tikv_import_error_counter` 等监控项，用于观察 Download SST 和 Ingest SST 的开销 #6404
- raftstore
    - 修复因 Follower Read 在 leader 变更时读到旧数据的问题，导致事务的隔离性被破坏的问题 #6343

16.19.6.2 Tools

- BR (Backup and Restore)
  - 修复备份进度信息不准确的问题 #127
  - 提升 split Region 的性能 #122
  - 新增备份恢复分区表的功能 #137
  - 新增自动调度 PD schedulers 功能 #123
  - 修复非 PKIsHandle 表恢复后数据覆盖的问题 #139

16.19.6.3 TiDB Ansible

- 新增初始化阶段自动关闭操作系统 THP 的功能 #1086
- 新增 BR 组件的 Grafana 监控 #1093
- 优化 TiDB Lightning 部署，自动创建相关目录 #1104

16.19.7 TiDB 3.1 Beta Release Notes

发版日期：2019 年 12 月 20 日

TiDB 版本：3.1.0-beta

TiDB Ansible 版本：3.1.0-beta

16.19.7.1 TiDB

- SQL 优化器
  - 丰富 SQL hint #12192
- 新功能
  - TiDB 支持 Follower Read 功能 #12535

16.19.7.2 TiKV

- 支持分布式备份恢复功能 #5532
- TiKV 支持 Follower Read 功能 #5562

16.19.7.3 PD

- 支持分布式备份恢复功能 #1896

16.20 v3.0

16.20.1 TiDB 3.0.20 Release Notes

发版日期：2020 年 12 月 25 日

TiDB 版本：3.0.20

16.20.1.1 兼容性更改

- TiDB
  - 废弃配置文件中的 `enable-streaming` 配置项 #21054

16.20.1.2 改进提升

- TiDB
  - 优化 `LOAD DATA` 语句执行 `PREPARE` 时的报错信息 #21222
- TiKV
  - 增加 `end_point_slow_log_threshold` 配置 #9145

16.20.1.3 Bug 修复

- TiDB
  - 修复错误缓存悲观事务提交状态的问题 #21706
  - 修复当查询 `INFORMATION_SCHEMA.TIDB_HOT_REGIONS` 时，统计信息不准确的问题 #21319
  - 修复了一处数据库名大小写处理不当，导致的 `DELETE` 未正确删除数据的问题 #21205
  - 修复创建递归的视图出现栈溢出的问题 #21000
  - 修复 TiKV 客户端 goroutine 泄漏的问题 #20863
  - 修复 `year` 类型默认值为 `0` 的问题 #20828
  - 修复 Index Lookup Join 的 goroutine 泄漏问题 #20791
  - 修复在悲观事务中执行 `INSERT SELECT FOR UPDATE` 后客户端收到 malformed packet 的问题 #20681
  - 修复 `'posixrules'` 错误时区的问题 #20605
  - 修复将无符号整型转换为 bit 类型时出现的错误 #20362
  - 修复 bit 列类型默认值错误的问题 #20339
  - 修复当等值条件中有 `Enum` 和 `Set` 类型时结果可能错误的问题 #20296
  - 修复 `!= any()` 时的错误问题 #20061
  - 修复类型转换在 `BETWEEN...AND...` 中会遇到结果错误的问题 #21503
  - 修复 `ADDDATE` 函数兼容性的问题 #21008
  - 为新增的 `Enum` 列设置正确的默认值 #20999
  - 修复 `SELECT DATE_ADD('2007-03-28 22:08:28',INTERVAL "-2.-2" SECOND)` 这类 SQL 语句的结果问题，使之兼容 MySQL #20627
  - 修复当修改列属性时，默认类型设置错误的问题 #20532
  - 修复 `timestamp` 函数的参数为 `float` 和 `decimal` 时，结果错误的问题 #20469
  - 修复统计信息可能会死锁的问题 #20424
  - 修复溢出的 Float 类型数据被 `INSERT` 的问题 #20251
- TiKV
  - 修复当事务删除 key 时却报 key 已存在的问题 #8931
- PD
  - 修复当 stale Region 过多时，启动 PD 会打印过量日志的问题 #3064

16.20.2 TiDB 3.0.19 Release Notes

发版日期：2020 年 9 月 25 日

TiDB 版本：3.0.19

16.20.2.1 兼容性变化

- PD
  - 更改 PD 的导入路径 `pingcap/pd` 为 `tikv/pd` #2779
  - 更改 PD 的 copyright 信息 `PingCAP, Inc` 为 `TiKV Project Authors` #2777

16.20.2.2 提升改进

- TiDB
  - 缓解故障恢复对 QPS 的影响 #19764
  - 支持调整 `union` 运算符的并发数 #19885
- TiKV
  - 永久开启 `sync-log` #8636
- PD
  - 添加关于 PD 重启的告警规则 #2789

16.20.2.3 Bug 修复

- TiDB
  - 修复 `slow-log` 文件不存在导致查询出错的问题 #20050
  - 添加对 `SHOW STATS_META` 和 `SHOW STATS_BUCKET` 这两个命令的权限检查 #19759
  - 禁止将 Decimal 类型改成 Integer 类型 #19681
  - 修复更改 ENUM/SET 类型的列时没有检查限制的问题 #20045
  - 修复 tidb-server 在 panic 后没有释放 table lock 的问题 #20021
  - 修复 OR 运算符在 WHERE 子句中没有正确处理的问题 #19901
- TiKV
  - 修复 TiKV 的 status server 解析响应出错导致 panic 的问题 #8540
- Tools
  - TiDB Lightning
    * 修复了严格模式下 CSV 中遇到不合法 UTF 字符集没有及时退出进程的问题 #378

16.20.3 TiDB 3.0.18 Release Notes

发版日期：2020 年 8 月 21 日

TiDB 版本：3.0.18

16.20.3.1 提升改进

- Tools
  - TiDB Binlog
    * 支持更加细粒度的 Pump GC 时间 #996

16.20.3.2 Bug 修复

- TiDB
  - 修复 Hash 函数对 `Decimal` 类型的错误处理导致 HashJoin 结果错误的问题 #19185
  - 修复 Hash 函数对 `Set` 和 `Enum` 类型的错误处理导致 HashJoin 结果错误的问题 #19175
  - 修复 Duplicate Key 检测在悲观事务下失效的问题 #19236
  - 修复 `Apply` 算子和 `Union Scan` 算子执行导致结果错误的问题 #19297
  - 修复某些缓存的执行计划在事务中执行结果错误的问题 #19274
- TiKV
  - 将 GC 的失败日志从 Error 级别改成 Warning 级别 #8444
- Tools
  - TiDB Lightning
    * 修复命令行参数 `--log-file` 无法生效的问题 #345
    * 修复 TiDB-backend 遇到空的 binary/hex 报语法错误的问题 #357
    * 修复使用 TiDB backend 时非预期的 `switch-mode` 调用 #368

16.20.4 TiDB 3.0.17 Release Notes

发版日期：2020 年 8 月 3 日

TiDB 版本：3.0.17

16.20.4.1 Bug 修复

- TiDB
  - 当一个查询中含有 `IndexHashJoin` 或 `IndexMergeJoin` 算子，且该算子的子节点发生 panic 时，返回客户端 panic 的原因，而非返回空结果 #18498
  - 修复形如 `SELECT a FROM t HAVING t.a` 的查询返回 UnknowColumn 错误的问题 #18432

- 当一张表没有主键，或其主键为整型时，禁止在这张表上执行添加主键 #18342
- 对 EXPLAIN FORMAT="dot" FOR CONNECTION 始终返回空结果 #17157
- 修复 STR_TO_DATE 函数处理 '%r' 和 '%h' 的行为 #18725

- TiKV

  - 修复在 Region 合并过程中可能导致读到旧数据的问题 #8111
  - 修复调度时可能产生内存泄漏的问题 #8355

- TiDB Lightning

  - 解决 log-file 参数不生效的问题 #345

16.20.4.2 优化

- TiDB

  - 将配置项 query-feedback-limit 默认值从 1024 修改为 512, 并优化统计信息反馈机制，降低其对集群的性能影响 #18770
  - 限制单次 split 请求中的 Region 个数 #18694
  - 加速 HTTP API /tiflash/replica 在集群中存在大量历史 DDL 记录时的访问速度 #18386
  - 提升索引等值条件下的行数估算准确率 #17609
  - 加快 kill tidb conn_id 的响应速度 #18506

- TiKV

  - 新增 hibernate-timeout 配置支持推后 Region 休眠时间，减少 Region 休眠对滚动升级的影响 #8207

- TiDB Lightning

  - 废弃 [black-white-list] 参数，新增一种更加简单易用的过滤规则 #332

16.20.5 TiDB 3.0.16 Release Notes

发版日期：2020 年 7 月 3 日

TiDB 版本：3.0.16

16.20.5.1 优化

- TiDB

  - 在 hash partition pruning 中支持 is null 过滤条件 #17308
  - 为每个 Region 设置单独的 Backoffer 避免多个 Region 同时失败引起等待时间过长 #17583
  - 添加新 partition 更新已有 partition 的分裂信息 #17668
  - 丢弃来自 delete / update 语句的 feedbacks #17841
  - 调整 job.DecodeArgs 中 json.Unmarshal 的使用以兼容新的 Go 版本 #17887
  - 移除 slow log 和 statement summary 中一些敏感信息 #18128

- Datetime 解析的分隔符和 MySQL 兼容 #17499
- 解析日期的 %h 时限定在 1..12 范围内 #17496

- TiKV

  - 避免在收到 snapshot 之后发送心跳给 PD 以提高稳定性 #8145
  - 优化了 PD client 的日志 #8091

16.20.5.2 Bug 修复

- TiDB

  - 修复当锁住的 primary key 在当前事务被插入/删除时可能造成的结果不一致问题 #18248
  - 修复因字段含义不一致导致日志中出现大量 Got too many pings gRPC 错误的问题 #17944
  - 修复当 HashJoin 返回 Null 类型列可能造成的 panic 问题 #17935
  - 修复访问被拒绝时的错误信息 #17722
  - 修复 JSON 数据中 int 和 float 类型比较的问题 #17715
  - 修复 Failpoint 测试造成的 data race 问题 #17710
  - 修复 Region 预分裂超时在创建表时可能不生效的问题 #17617
  - 修复 BatchClient 中因为失败可能导致的主动 panic #17378
  - 修复 FLASHBACK TABLE 在某些情况下可能失败的问题 #17165
  - 修复只有 string 列时 range 范围计算可能不准确的问题 #16658
  - 修复 only_full_group_by 模式下的错误 #16620
  - 修复 case when 函数返回字段长度不准确的问题 #16562
  - 修复 count 聚合函数对 decimal 类型推断的问题 #17702

- TiKV

  - 修复了潜在的 ingest file 导致的读取结果错误的问题 #8039
  - 修复了多次 merge 过程中被隔离的节点上的副本无法被正确移除的问题 #8005

- PD

  - 修复一些情况下使用 PD Control 查询 Region 报 404 错误的问题 #2577

16.20.6 TiDB 3.0.15 Release Notes

发版日期：2020 年 6 月 5 日

TiDB 版本：3.0.15

16.20.6.1 新功能

- TiDB

  - 禁止分区表上的查询使用 plan cache 功能 #16759
  - 分区表支持 admin recover index、admin check index 语句 #17315 #17390

- Range 类型分区表支持按 in 查询条件进行分区裁剪 #17318
- 优化 SHOW CREATE TABLE 的输出结果，在分区名称上添加了引号 #16315
- GROUP_CONCAT 支持 ORDER BY 子句 #16988
- 优化统计信息 CMSketch 的内存分配机制，减少垃圾回收导致的性能影响 #17543

- PD

  - 新增按照 Leader 个数调度的策略 #2479

16.20.6.2 Bug 修复

- TiDB

  - Hash 聚合函数中，采用深拷贝的方式拷贝 enum 和 set 类型数据，且修复一处正确性问题 #16890
  - 修复点查因整数溢出处理逻辑不正确导致输出结果不正确的问题 #16753
  - 修复 CHAR() 函数作为查询的谓词条件时因处理逻辑不正确导致输出的结果不正确的问题 #16557
  - 修复 IsTrue 和 IsFalse 函数存储层和计算层计算结果不一致的问题#16627
  - 修复部分表达式（例如 case when）中，Not Null 标记设置不正确的问题 #16993
  - 修复部分场景中优化器无法为 TableDual 找到物理计划的问题 #17014
  - 修复 Hash 分区表中分区选择的语法没有正确生效的问题 #17051
  - 修复 XOR 作用于浮点数时，结果与 MySQL 不一致的问题 #16976
  - 修复 prepare 方式执行 DDL 语句出错的问题 #17415
  - 修复 ID 分配器中计算 Batch 大小的逻辑处理不正确的问题 #17548
  - 修复 MAX_EXEC_TIME 的 SQL Hint 在超过 expensive 阈值后不生效的问题 #17534

- TiKV

  - 修复长时间运行后由于处理逻辑不正确导致碎片整理不再有效的问题 #7790
  - 修复系统意外重启后错误地删除 snapshot 文件导致系统 panic 的问题 #7925
  - 修复因消息包过大导致 gRPC 连接断开的问题 #7822

16.20.7 TiDB 3.0.14 Release Notes

发版日期：2020 年 5 月 9 日

TiDB 版本：3.0.14

16.20.7.1 兼容性变化

- TiDB

  - performance_schema 和 metrics_schema 由读写改为只读 #15417

16.20.7.2 重点修复的 Bug

- TiDB
  - 修复 join 条件在 handle 列上存在多个等值条件时，index join 查询结果错误的问题 #15734
  - 修复 fast analyze handle 列 panic 的问题 #16079
  - 修复通过 prepare 方式执行 DDL 语句时，DDL job 结构中 query 字段错误的问题，该问题可能导致使用 Binlog 同步时，上下游数据产生不一致 #15443
- TiKV
  - 修复重复清锁请求可能破坏事务原子性的问题 #7388

16.20.7.3 新功能

- TiDB
  - `admin show ddl jobs` 查询结果中添加库名和表名列 #16428
  - `RECOVER TABLE` 支持恢复被 `TRUNCATE` 的表 #15458
  - 新增 `SHOW GRANTS` 语句权限检查的功能 #16168
  - 新增 `LOAD DATA` 语句权限检查 #16736
  - 提升时间日期相关函数作为 partition key 时，分区裁剪的性能 #15618
  - `dispatch error` 的日志级别从 `WARN` 调整为 `ERROR` #16232
  - 新增支持 `require-secure-transport` 启动项，以强制要求客户端必须使用 TLS #15415
  - 支持内部组件间 http 通信使用 TLS #15419
  - `information_schema.processlist` 表中添加显示当前事务 `start_ts` 信息 #16160
  - 新增自动重加载集群间通讯 TLS 证书信息的功能 #15162
  - 通过重构分区裁剪的实现，提升分区表的读操作的性能 #15628
  - 新增当使用 `floor(unix_timestamp(a))` 作为 `range` 分区表的分区表达式时，支持分区裁剪功能 #16521
  - 修改 `update` 语句中包含 `view` 且不对该 `view` 进行 update 时的行为，由不允许执行改为正常执行 #16787
  - 禁止创建嵌套 `view` #15424
  - 禁止 truncate `view` #16420
  - 当列处于非 public 状态时，禁止用 `update` 语句显式的更新此列的值 #15576
  - 当 status 端口被占用时，禁止启动 TiDB #15466
  - `current_role` 函数的字符集由 binary 调整为 utf8mb4 #16083
  - 通过在处理完每个 Region 后增加检查 `max-execution-time` 是否符合条件，提升系统处理 `max-execution-time` 的响应灵敏度 #15615
  - 新增语法 `ALTER TABLE ... AUTO_ID_CACHE` 用于显式设置 `auto_id` 的缓存步长 #16287
- TiKV
  - 提升乐观事务存在大量冲突及 `BatchRollback` 存在时的性能 #7605
  - 提升悲观事务冲突严重的场景下悲观锁 waiter 被频繁唤醒导致性能下降的问题 #7584
- Tools

- TiDB Lightning
    * tidb-lightning-ctl 新增 `fetch-mode` 子命令，输出 TiKV 集群模式 #287

16.20.7.4 Bug 修复

- TiDB
    - 修复 `WEEKEND` 函数在 SQL mode 为 `ALLOW_INVALID_DATES` 时结果与 MySQL 不兼容的问题 #16170
    - 修复当索引列上包含自增主键时，`DROP INDEX` 执行失败的问题 #16008
    - 修复 Statement Summary 中，`TABLE_NAMES` 列值有时会不正确的问题 #15231
    - 修复因 Plan Cache 启动后部分表达式计算结果错误的问题 #16184
    - 修复函数 `not/istrue /isfalse` 计算结果错误的问题 #15916
    - 修复带有冗余索引的表 MergeJoin 时 Panic 的问题 #15919
    - 修复谓词只跟外表有联接的情况下错误地化简外链接的问题 #16492
    - 修复 `SET ROLE` 导致的 `CURRENT_ROLE` 函数报错问题 #15569
    - 修复 `LOAD DATA` 在遇到 `\` 时，处理结果与 MySQL 不兼容的问题 #16633
    - 修复数据库可见性与 MySQL 不兼容的问题 #14939
    - 修复 `SET DEFAULT ROLE ALL` 语句的权限检查不正确的问题 #15585
    - 修复 plan cache 导致的分区裁剪失效问题 #15818
    - 修复因事务未对相关表进行加锁，该表存在并发的 DDL 操作且有阻塞时导致事务提交时报 `schema change` 的问题 #15707
    - 修复 `IF(not_int, *, *)` 行为不正确的问题 #15356
    - 修复 `CASE WHEN (not_int)` 行为不正确的问题 #15359
    - 修复在使用非当前 schema 中的视图时报 `Unknown column` 错误的问题 #15866
    - 修复解析时间字符串的结果与 MySQL 不兼容的问题 #16242
    - 修复 left join 右孩子节点有 `null` 列可能会导致 join 上的排序算子 panic 的问题 #15798
    - 修复当 TiKV 持续返回 `StaleCommand` 错误期间，执行 SQL 的流程被阻塞且不报错的问题 #16528
    - 修复启用审计插件后端口探活可能会导致 panic 的问题 #16064
    - 修复 `fast analyze` 作用于 index 时导致 panic 的问题 #15967
    - 修复某些情况下 `SELECT * FROM INFORMATION_SCHEMA.PROCESSLIST` 语句 panic 的问题 #16309
    - 修复哈希分区表在建表时由于分配内存之前未及时检查分区数量导致当指定非常大的分区数量（例如 9999999999999）时，导致 TiDB OOM 的问题 #16218
    - 修复 `information_schema.tidb_hot_table` 对于分区表信息不准确的问题 #16726
    - 修复分区选择算法在哈希分区表上不生效的问题 #16070
    - 修复 mvcc 系列的 HTTP API 不支持分区表的问题 #16191
    - 保持 `UNION` 语句和 `SELECT` 语句对于错误处理的行为一致 #16137
    - 修复当 `VALUES` 函数参数类型为 `bit(n)` 时行为不正确的问题 #15486
    - 修复 `view` 列名过长时处理逻辑与 MySQL 不一致的问题，当列名过长时，系统自动生成一个短的列名 #14873
    - 修复 `(not not col)` 被错误地优化为 `col` 的问题 #16094
    - 修复 index join 构造内表 range 错误的问题 #15753
    - 修复 `only_full_group_by` 对含括号的表达式检查错误的问题 #16012
    - 修复 `select view_name.col_name from view_name` 报错的问题 #15572
- TiKV

- 修复某些情况节点隔离恢复之后无法被正确删掉的问题 #7703
- 修复网络隔离时 Region Merge 可能导致数据丢失的问题 #7679
- 修复某些情况 learner 无法被正确移除的问题 #7598
- 修复扫描 raw kv 时可能乱序的问题 #7597
- 修复由于 Raft 消息 batch 过大时导致连接重连的问题 #7542
- 修复 empty request 造成 gRPC 线程死锁的问题 #7538
- 修复 merge 过程中 learner 重启的处理逻辑不正确的问题 #7457
- 修复重复清锁请求可能破坏事务原子性的问题 #7388

16.20.8 TiDB 3.0.13 Release Notes

发版日期：2020 年 04 月 22 日

TiDB 版本：3.0.13

16.20.8.1 Bug 修复

- TiDB
  - 修复由于未检查 `MemBuffer`，事务内执行 `INSERT ... ON DUPLICATE KEY UPDATE` 语句插入多行重复数据可能出错的问题 #16690
- TiKV
  - 修复重复多次执行 `Region Merge` 导致系统被阻塞的问题，阻塞期间服务不可用 #7612

16.20.9 TiDB 3.0.12 Release Notes

发版日期：2020 年 3 月 16 日

TiDB 版本：3.0.12

TiDB Ansible 版本：3.0.12

> **警告：**
>
> 该版本存在一些已知问题，已在新版本中修复，建议使用 3.0.x 的最新版本。

16.20.9.1 兼容性变化

- TiDB
  - 修复慢日志中记录 prewrite binlog 的时间部分计时不准确问题。原本计时的字段名是 `Binlog_prewrite_time`，这次修正后，名称更改为 `Wait_prewrite_binlog_time`。#15276

16.20.9.2 新功能

- TiDB
  - 支持通过 `alter instance` 语句动态加载已被替换的证书文件 #15080 #15292
  - 添加 `cluster-verify-cn` 配置项，配置后必须是对应 CN 证书才使用 status 服务 #15164
  - 在每个 TiDB server 中添加对 DDL 请求的一个限流的功能，从而降低 DDL 请求冲突报错频率 #15148
  - 支持在 binlog 写入失败时，TiDB 退出 #15339
- Tools
  - TiDB Binlog
    * Drainer 新增 `kafka-client-id` 配置项，支持连接 Kafka 客户端配置客户端 ID #929

16.20.9.3 Bug 修复

- TiDB
  - 使 GRANT/REVOKE 在对多个用户修改时，保证原子性 #15092
  - 修复在分区表上面悲观锁的加锁未能锁住正确的行的问题 #15114
  - 建索引长度超过限制时，使报错信息根据配置中 `max-index-length` 的值显示 #15130
  - 修复 `FROM_UNIXTIME` 函数小数点位数不正确的问题 #15270
  - 修复一个事务中删除自己写的记录导致冲突检测失效或数据索引不一致问题 #15176
- TiKV
  - 修复一个在关闭一致性检查参数时，在事务中插入一个已存在的 Key 然后立马删除，导致冲突检测失效或数据索引不一致的问题 #7054
  - Raftstore 引入流控机制，解决没有流控可能导致追日志太慢可能导致集群卡住，以及事务大小太大会导致 TiKV 间连接频繁重连的问题 #7072 #6993
- PD
  - 修复 PD 因处理 Region heartbeat 时的数据竞争导致 Region 信息不正确的问题 #2233
- TiDB Ansible
  - 支持一个集群部署多个 Grafana/Prometheus/Alertmanager #1198

16.20.10 TiDB 3.0.11 Release Notes

发版日期：2020 年 3 月 4 日

TiDB 版本：3.0.11

TiDB Ansible 版本：3.0.11

> **警告：**
>
> 该版本存在一些已知问题，已在新版本中修复，建议使用 3.0.x 的最新版本。

16.20.10.1 兼容性变化

- TiDB
  - 新增 `max-index-length` 配置项，用于控制索引支持的最大长度，用户可自由选择兼容 v3.0.7 之前版本或者兼容 MySQL #15057

16.20.10.2 新功能

- TiDB
  - 新增在 `information_schema.PARTITIONS` 表中显示分区表的分区元信息的功能 #14849
- TiDB Binlog
  - 新增 TiDB 集群之间数据双向复制功能 #884 #909
- TiDB Lightning
  - 新增配置 TLS 功能 #44 #270
- TiDB Ansible
  - 优化 `create_user.yml` 的逻辑，中控机使用的用户不必和 `ansible_user` 一致 #1184

16.20.10.3 Bug 修复

- TiDB
  - 修复由于涉及 Union 的查询没有标记为只读，在乐观事务开启重试时会导致 Goroutine 泄露的问题 #15076
  - 修复执行 `SET SESSION tidb_snapshot = 'xxx';` 语句后，由于执行时未正确使用 `tidb_snapshot` 变量的值，导致 `SHOW TABLE STATUS` 未正确输出快照时刻表状态的问题 #14391
  - 修复 `Sort Merge Join` 与 `ORDER BY DESC` 在同一条 SQL 语句中时，输出结果不正确的问题 #14664
  - 修复创建分区表时，由于使用不支持的表达式，导致 TiDB server panic 的问题，修复后返回 `This ↪ partition function is not allowed` 错误信息 #14769
  - 修复执行 `select max()from subquery` 语句且 Subquery 包含 Union 的子查询时，输出结果不正确的问题 #14944
  - 修复执行 `DROP BINDING` 语句解除执行计划绑定后，执行 `SHOW BINDINGS` 语句系统返回错误信息的问题 #14865
  - 修复查询语句中别名长度大于 256 时，由于在查询结果中未按照 MySQL 协议对别名截断，导致连接被断开的问题 #14940
  - 修复字符串类型被用作 `DIV` 中时，查询结果可能不正确的问题，例如：`select 1 / '2007' div 1` 现在可以被正确地执行 #14098
- TiKV
  - 优化日志输出，删除部分不必要的日志 #6657
  - 修复 peer 在高负载情况下若被删除可能导致 panic 的问题 #6704
  - 修复 Hibernate Region 在某些特殊条件下未被正确唤醒的问题 #6732 #6738
- TiDB Ansible
  - 修复 `tidb-ansible` 中失效、过期的文档链接 #1169
  - 修复 `wait for region replication complete` task 可能出现未定义变量的问题 #1173

16.20.11 TiDB 3.0.10 Release Notes

发版日期：2020 年 2 月 20 日

TiDB 版本：3.0.10

TiDB Ansible 版本：3.0.10

> **警告：**
>
> 该版本存在一些已知问题，已在新版本中修复，建议使用 3.0.x 的最新版本。

16.20.11.1 TiDB

- 修复 IndexLookUpJoin 在利用 OtherCondition 构造 InnerRange 时出现错误 Join 结果 #14599
- 删除 `tidb_pprof_sql_cpu` 配置项，新增 Server 级别的 `tidb_pprof_sql_cpu` 变量 #14416
- 修复用户只在具有全局权限时才能查询所有数据库的问题 #14386
- 修复执行 point-get 时由于事务超时导致数据的可见性不符合预期的问题 #14480
- 将悲观事务激活的时机改为延迟激活，与乐观事务模型保持一致 #14474
- 修复 unixtimestamp 表达式在计算分区表分区的时区不正确的问题 #14476
- 新增`tidb_session_statement_deadlock_detect_duration_seconds` 监控项，用于监控死锁检测时间 #14484
- 修复 GC worker 由于部分逻辑不正确导致系统 panic 的问题 #14439
- 修复 IsTrue 函数的表达式名称不正确的问题 #14516
- 修复部分内存使用统计不准确的问题 #14533
- 修复统计信息 CM-Sketch 初始化时由于处理逻辑不正确导致系统 panic 的问题 #14470
- 修复查询分区表时分区裁剪 (partition pruning) 不准确的问题 #14546
- 修复 SQL 绑定中 SQL 语句默认数据库名设置不正确的问题 #14548
- 修复 json_key 与 MySQL 不兼容的问题 #14561
- 新增分区表自动更新统计信息的功能 #14566
- 修复执行 point-get 时 plan id 会变化的问题，正常情况 plan id 始终是 1 #14595
- 修复 SQL 绑定不完全匹配时处理逻辑不正确导致系统 panic 的问题 #14263
- 新增 `tidb_session_statement_pessimistic_retry_count` 监控项，用于监控悲观事务加锁失败后重试次数 #14619
- 修复 `show binding` 语句权限检查不正确的问题 #14618
- 修复由于 backoff 的逻辑里没有检查 killed 标记，导致 kill 无法正确执行的问题 #14614
- 通过减少持有内部锁的时间来提高 statement summary 的性能 #14627
- 修复 TiDB 从字符串解析成时间与 MySQL 不兼容的问题 #14570
- 新增审计日志记录用户登录失败的功能 #14620
- 新增 `tidb_session_ statement_lock_keys_count` 监控项，用于监控悲观事务的 lock keys 的数量 #14634
- 修复 json 对 & < > 等字符输出转义不正确的问题 #14637
- 修复 hash-join 在建 hash-table 时由于内存使用过多导致系统 panic 的问题 #14642
- 修复 SQL 绑定处理不合法记录时处理逻辑不正确导致 panic 的问题 #14645
- 修复 Decimal 除法计算与 MySQL 不兼容的问题，Decimal 除法计算中增加 Truncated 错误检测 #14673
- 修复给用户授权不存在的表执行成功的问题 #14611

16.20.11.2 TiKV

- Raftstore
  - 修复由于 Region merge 失败导致系统 Panic #6460 或者数据丢失 #598 的问题 #6481
  - 支持 yield 优化调度公平性，支持预迁移 leader 优化 leader 调度的稳定性 #6563

16.20.11.3 PD

- 当系统流量有变化时，系统自动更新 Region 缓存信息，解决缓存失效的问题 #2103
- 采用 leader 租约时间确定 TSO 的有效时间 #2117

16.20.11.4 Tools

- TiDB Binlog
  - Drainer 支持 relay log #893
- TiDB Lightning
  - 优化配置项，部分配置项在没有设置的时候使用默认配置 #255
  - 修复在非 server mode 模式下 web 界面无法打开的问题 #259

16.20.11.5 TiDB Ansible

- 修复某些场景获取不到 PD Leader 导致命令执行失败的问题 #1121
- TiDB Dashboard 新增 `Deadlock Detect Duration` 监控项 #1127
- TiDB Dashboard 新增 `Statement Lock Keys Count` 监控项 #1132
- TiDB Dashboard 新增 `Statement Pessimistic Retry Count` 监控项 #1133

16.20.12 TiDB 3.0.9 Release Notes

发版日期：2020 年 1 月 14 日

TiDB 版本：3.0.9

TiDB Ansible 版本：3.0.9

> **警告：**
>
> 该版本存在一些已知问题，已在新版本中修复，建议使用 3.0.x 的最新版本。

16.20.12.1 TiDB

- Executor
  - 修复聚合函数作用于枚举和集合列时结果不正确的问题 #14364
- Server
  - 支持系统变量 `auto_increment_increment` 和 `auto_increment_offset` #14396
  - 新增 `tidb_tikvclient_ttl_lifetime_reach_total` 监控项，监控悲观事务 TTL 达到 10 分钟的数量 #14300
  - 执行 SQL 过程中当发生 panic 时输出导致 panic 的 SQL 信息 #14322
  - statement summary 系统表新增 `plan` 和 `plan_digest` 字段，记录当前正在执行的 `plan` 和 `plan` 的签名 #14285
  - 配置项 `stmt-summary.max-stmt-count` 的默认值从 100 调整至 200 #14285
  - slow query 表新增 `plan_digest` 字段，记录 `plan` 的签名 #14292
- DDL
  - 修复 `alter table ... add index` 语句创建匿名索引行为与 MySQL 不一致的问题 #14310
  - 修复 `drop table` 错误删除视图的问题 #14052
- Planner
  - 提升类似 `select max(a), min(a)from t` 语句的性能。如果 a 列表上有索引，该语句会被优化为 `select * from (select a from t order by a desc limit 1)as t1, (select a from t order by a limit 1)as t2` 以避免全表扫 #14410

16.20.12.2 TiKV

- Raftstore
  - 提升 Raft 成员变更的速度 #6421
- Transaction
  - 新增 `tikv_lock_manager_waiter_lifetime_duration`、`tikv_lock_manager_detect_duration`、`tikv_lock_manager_detect_duration` 监控项，用于监控 `waiter` 的生命周期、死锁检测耗费时间、`wait table` 的状态 #6392
  - 通过优化配置项 `wait-for-lock-time` 默认值从 3s 调整到 1s、`wake-up-delay-duration` 默认值从 100ms 调整为 20ms，以降低极端场景下 Region Leader 切换、切换死锁检测的 leader 导致的事务执行延迟 #6429
  - 修复 Region Merge 过程中可能导致死锁检测器 leader 角色误判的问题 #6431

16.20.12.3 PD

- 新增 location label 的名字中允许使用斜杠 / 的功能 #2083
- 修复因为不正确地统计了 tombstone 的标签，导致该统计信息不准的问题 #2060

16.20.12.4 Tools

- TiDB Binlog
  - Drainer 输出的 binlog 协议中新增 unique key 信息 #862
  - Drainer 支持使用加密后的数据库连接密码 #868

16.20.12.5 TiDB Ansible

- 优化 Lightning 部署，自动创建相关目录 #1105

16.20.13 TiDB 3.0.8 Release Notes

发版日期：2019 年 12 月 31 日

TiDB 版本：3.0.8

TiDB Ansible 版本：3.0.8

16.20.13.1 TiDB

- SQL 优化器
  - 修复 SQL Binding 因为 cache 更新不及时，导致绑定计划错误的问题 #13891
  - 修复当 SQL 包含符号列表（类似于 `?, ?, ?` 这样的占位符）时，SQL Binding 可能失效的问题 #14004
  - 修复 SQL Binding 由于原 SQL 以 ; 结尾而不能创建/删除的问题 #14113
  - 修复 PhysicalUnionScan 算子没有正确设置统计信息，导致查询计划可能选错的问题 #14133
  - 移除 `minAutoAnalyzeRatio` 约束使自动 analyze 更及时 #14015
- SQL 执行引擎
  - 修复 `INSERT/REPLACE/UPDATE ... SET ... = DEFAULT` 语法会报错的问题，修复 `DEFAULT` 表达式与虚拟生成列配合使用会报错的问题 #13682
  - 修复 `INSERT` 语句在进行字符串类型到浮点类型转换时，可能会报错的问题 #14011
  - 修复 HashAgg Executor 并发值未被正确初始化，导致聚合操作执行在一些情况下效率低的问题 #13811
  - 修复 group by item 被括号包含时执行报错的问题 #13658
  - 修复 TiDB 没有正确计算 group by item，导致某些情况下 OUTER JOIN 执行会报错的问题 #14014
  - 修复向 Range 分区表写入超过 Range 外的数据时，报错信息不准确的问题 #14107
  - 鉴于 MySQL 8 即将废弃 PadCharToFullLength，revert PR #10124 并撤销 PadCharToFullLength 的效果，以避免一些特殊情况下查询结果不符合预期 #14157
  - 修复 ExplainExec 中没有保证 `close()` 的调用而导致 `EXPLAIN ANALYZE` 时造成 goroutine 泄露的问题 #14226
- DDL
  - 优化 “change column”/“modify column” 的输出的报错信息，让人更容易理解 #13796
  - 新增 `SPLIT PARTITION TABLE` 语法，支持分区表切分 Region 功能 #13929
  - 修复创建索引时，没有正确检查长度，导致索引长度超过 3072 字节没有报错的问题 #13779

  - 修复由于分区表添加索引时若花费时间过长，可能导致输出 GC life time is shorter than transaction duration 报错信息的问题 #14132
  - 修复在 DROP COLUMN/MODIFY COLUMN/CHANGE COLUMN 时没有检查外键导致执行 SELECT * FROM information_schema.KEY_COLUMN_USAGE 语句时发生 panic 的问题 #14105

- Server
  - Statement Summary 功能改进：
    * 新增大量的 SQL 指标字段，便于对 SQL 进行更详细的统计分析 #14151，#14168
    * 新增 stmt-summary.refresh-interval 参数用于控制定期将 events_statements_summary_by_digest 表中过期的数据移到 events_statements_summary_by_digest_history 表，默认间隔时间：30min #14161
    * 新增 events_statements_summary_by_digest_history 表,保存从 events_statements_summary_by_digest 中过期的数据 #14166
  - 修复执行 RBAC 相关的内部 SQL 时，错误输出 binlog 的问题 #13890
  - 新增 server-version 配置项来控制修改 TiDB server 版本的功能 #13906
  - 新增通过 HTTP 接口恢复 TiDB binlog 写入功能 #13892
  - 将 GRANT roles TO user 所需要的权限由 GrantPriv 修改为 ROLE_ADMIN 或 SUPER，以与 MySQL 保持一致 #13932
  - 当 GRANT 语句未指定 database 名时，TiDB 行为由使用当前 database 改为报错 No database selected，与 MySQL 保持兼容 #13784
  - 修改 REVOKE 语句执行权限从 SuperPriv 改成用户只需要有对应 Schema 的权限，就可以执行 REVOKE 语句，与 MySQL 保持一致 #13306
  - 修复 GRANT ALL 语法在没有 WITH GRANT OPTION 时，错误地将 GrantPriv 授权给目标用户的问题 #13943
  - 修复 LoadDataInfo 中调用 addRecord 报错时，报错信息不包含导致 LOAD DATA 语句行为不正确信息的问题 #13980
  - 修复因查询中多个 SQL 语句共用同一个 StartTime 导致输出错误的慢查询信息的问题 #13898
  - 修复 batchClient 处理大事务时可能造成内存泄露的问题 #14032
  - 修复 system_time_zone 固定显示为 CST 的问题，现在 TiDB 的 system_time_zone 会从 mysql.tidb 表中的 systemTZ 获取 #14086
  - 修复 GRANT ALL 语法授予权限不完整（例如 Lock_tables_priv）的问题 #14092
  - 修复 Priv_create_user 权限不能 CREATE ROLE 和 DROP ROLE的问题 #14088
  - 将 ErrInvalidFieldSize 的错误码从 1105(Unknow Error) 改成 3013 #13737
  - 新增 SHUTDOWN 命令用于停止 TiDB Server，并新增 ShutdownPriv 权限 #14104
  - 修复 DROP ROLE 语句的原子性问题，避免语句执行失败时，一些 ROLE 仍然被非预期地删除 #14130
  - 修复 3.0 以下版本升级到 3.0 时，tidb_enable_window_function 在 SHOW VARIABLE 语句的查询结果错误输出 1 的问题，修复后输出 0 #14131
  - 修复 TiKV 节点下线时，由于 gcworker 持续重试导致可能出现 goroutine 泄露的问题 #14106
  - 在慢日志中记录 Binlog 的 Prewrite 的时间，提升问题追查的易用性 #14138
  - tidb_enable_table_partition 变量支持 GLOBAL SCOPE 作用域 #14091
  - 修复新增权限时未正确将新增的权限赋予对应的用户导致用户权限可能缺失或者被误添加的问题 #14178
  - 修复当 TiKV 链接断开时，由于 rpcClient 不会关闭而导致 CheckStreamTimeoutLoop goroutine 会泄露的问题 #14227
  - 支持基于证书的身份验证（使用文档）#13955

- Transaction
  - 创建新集群时，`tidb_txn_mode` 变量的默认值由 `""` 改为 `"pessimistic"` #14171
  - 修复悲观事务模式，事务重试时单条语句的等锁时间没有被重置导致等锁时间过长的问题 #13990
  - 修复悲观事务模式，因对没有修改的数据未加锁导致可能读到不正确数据的问题 #14050
  - 修复 mocktikv 中 prewrite 时，没有区分事务类型，导致重复的 insert value 约束检查 #14175
  - 修复 `session.TxnState` 状态为 `Invalid` 时，事务没有被正确处理导致 panic 的问题 #13988
  - 修复 mocktikv 中 `ErrConfclit` 结构未包含 `ConflictCommitTS` 的问题 #14080
  - 修复 TiDB 在 Resolve Lock 之后，没有正确处理锁超时检查导致事务卡住的问题 #14083
- Monitor
  - `LockKeys` 新增 `pessimistic_lock_keys_duration` 监控 #14194

16.20.13.2 TiKV

- Coprocessor
  - 修改 Coprocessor 遇到错误时输出日志的级别从 `error` 改成 `warn` #6051
  - 修改统计信息采样数据的更新行为从直接更行改成先删除再插入，更新行为与 tidb-server 保持一致 #6069
- Raftstore
  - 修复因重复向 `peerfsm` 发送 destory 消息，`peerfsm` 被多次销毁导致 panic 的问题 #6297
  - `split-region-on-table` 默认值由 `true` 改成 `false`，默认关闭按 table 切分 Region 的功能 #6253
- Engine
  - 修复极端条件下因 RocksDB 迭代器错误未正确处理导致可能返回空数据的问题 #6326
- 事务
  - 修复悲观锁因锁未被正确清理导致 Key 无法写入数据，且出现 GC 卡住的问题 #6354
  - 优化悲观锁等锁机制，提升锁冲突严重场景的性能 #6296
- 将内存分配库的默认值由 `tikv_alloc/default` 改成 `jemalloc` #6206

16.20.13.3 PD

- Client
  - 新增通过 `context` 创建新 client，创建新 client 时可设置超时时间 #1994
  - 新增创建 KeepAlive 连接功能 #2035
- 优化 `/api/v1/regions` API 的性能 #1986
- 修复删除 `tombstone` 状态的 Store 可能会导致 panic 的隐患 #2038
- 修复从磁盘加载 Region 信息时错误的将范围有重叠的 Region 删除的问题 #2011，#2040
- 将 etcd 版本从 3.4.0 升级到 3.4.3 稳定版本，注意升级后只能通过 pd-recover 工具降级 #2058

16.20.13.4 Tools

- TiDB Binlog
  - 修复 Pump 由于没有收到 DDL 的 commit binlog 导致 binlog 被忽略的问题 #853

16.20.13.5 TiDB Ansible

- 回滚被精简的配置项 #1053
- 优化滚动升级时 TiDB 版本检查的逻辑 #1056
- TiSpark 版本升级到 2.1.8 #1061
- 修复 Grafana 监控上 PD 页面 Role 监控项显示不正确的问题 #1065
- 优化 Grafana 监控上 TiKV Detail 页面上 `Thread Voluntary Context Switches` 和 `Thread Nonvoluntary Context Switches` 监控项 #1071

16.20.14 TiDB 3.0.7 Release Notes

发版日期：2019 年 12 月 4 日

TiDB 版本：3.0.7

TiDB Ansible 版本：3.0.7

16.20.14.1 TiDB

- 修复 TiDB server 本地时间落后于 TSO 时间时，可能造成锁的 TTL 过大的问题 #13868
- 修复从字符串解析日期时，由于使用本地时区 (`gotime.Local`) 而导致解析结果的时区不正确的问题 #13793
- 修复 `builtinIntervalRealSig` 的实现中，`binSearch` 方法不会返回 error，导致最终结果可能不正确的问题 #13767
- 修复整型数据被转换为无符号浮点/Decimal 类型时，精度可能丢失造成数据错误的问题 #13755
- 修复 Natural Outer Join 和 Outer Join 使用 `USING` 语法时，`not null` 标记没有被重置导致结果错误的问题 #13739
- 修复更新统计信息时可能存在数据竞争，导致统计信息不准确的问题 #13687

16.20.14.2 TiKV

- 判断死锁检测服务的第一个 Region 时，加上 Region 合法检测，防止信息不完整的 Region 导致误判 #6110
- 修复潜在的内存泄漏问题 #6128

16.20.15 TiDB 3.0.6 Release Notes

发版日期：2019 年 11 月 28 日

TiDB 版本：3.0.6

TiDB Ansible 版本：3.0.6

16.20.15.1 TiDB

- SQL 优化器
  - 修复窗口函数 AST Restore SQL 文本后结果不正确问题，`over w` 不应被 restore 成 `over (w)` #12933
  - 修复 stream aggregation 下推给 double read 的问题 #12690
  - 修复 SQL bind 中引号处理不正确的问题 #13117
  - 优化 `select max(_tidb_rowid)from t` 场景，避免全表扫 #13095
  - 修复当查询语句中包含变量赋值表达式时查询结果不正确的问题 #13231
  - 修复 UPDATE 语句中同时包含子查询和 generated column 时结果错误的问题；修复 UPDATE 语句中包含不同数据库的两个表名相同的表时，UPDATE 执行报错的问题 #13350
  - 支持用 `_tidb_rowid` 做点查 #13416
  - 修复分区表统计信息使用错误导致生成执行计划不正确的问题 #13628
- SQL 执行引擎
  - 修复 year 类型对于无效值的处理时和 MySQL 不兼容问题 #12745
  - 在 `INSERT ON DUPLICATE UPDATE` 语句中复用 Chunk 以降低内存开销 #12998
  - 添加内置函数 `JSON_VALID` 的支持 #13133
  - 支持在分区表上执行 `ADMIN CHECK TABLE` #13140
  - 修复对空表进行 `FAST ANALYZE` 时 panic 的问题 #13343
  - 修复在包含多列索引的空表上执行 Fast Analyze 时 panic 的问题 #13394
  - 修复当 `WHERE` 子句上有 UNIQUE KEY 的等值条件时，估算行数大于 1 的问题 #13382
  - 修复当 TiDB 开启 `Streaming` 后返回数据有可能重复的问题 #13254
  - 将 `CMSketch` 中出现次数最多的 N 个值抽取出来，提高估算准确度 #13429
- Server
  - 当 gRPC 请求超时时，提前让发往 TiKV 的请求失败 #12926
  - 添加以下虚拟表： #13009
    * `performance_schema.tidb_profile_allocs`
    * `performance_schema.tidb_profile_block`
    * `performance_schema.tidb_profile_cpu`
    * `performance_schema.tidb_profile_goroutines`
  - 修复 query 在等待悲观锁时，kill query 不生效的问题 #12989
  - 当悲观事务上锁失败，且事务只涉及一个 key 的修改时，不再异步回滚 #12707
  - 修复 split Region 请求的 response 为空时 panic 的问题 #13092
  - 悲观事务在其他事务先锁住导致上锁失败时，避免重复 backoff #13116
  - 修改 TiDB 检查配置时的行为，出现不能识别的配置选项时，打印警告日志 #13272
  - 支持通过 `/info/all` 接口获取所有 TiDB 节点的 binlog 状态 #13187
  - 修复 kill connection 时可能出现 goroutine 泄漏的问题 #13251
  - 让 `innodb_lock_wait_timeout` 参数在悲观事务中生效，用于控制悲观锁的等锁超时时间 #13165
  - 当悲观事务的 query 被 kill 后，停止更新悲观事务的 TTL，避免其他的事务做不必要的等待 #13046
- DDL
  - 修复 `SHOW CREATE VIEW` 结果与 MySQL 不一致的问题 #12912
  - 支持基于 UNION 创建 View，例如 `create view v as select * from t1 union select * from t2` #12955

  - 给 slow_query 表添加更多事务相关的字段：#13072
    * Prewrite_time
    * Commit_time
    * Get_commit_ts_time
    * Commit_backoff_time
    * Backoff_types
    * Resolve_lock_time
    * Local_latch_wait_time
    * Write_key
    * Write_size
    * Prewrite_region
    * Txn_retry
  - 创建表时如果表包含 collate 则列使用表的 collate 而不是系统默认的字符集 #13174
  - 创建表时限制索引名字的长度 #13310
  - 修复 rename table 时未检查表名长度的问题 #13346
  - 新增 alter-primary-key 配置来支持 TiDB add/drop primary key，该配置默认关闭 #13522

16.20.15.2 TiKV

- 修复 acquire_pessimistic_lock 接口返回错误 txn_size 的问题 #5740
- 限制 GC worker 每秒写入量，降低对性能的影响 #5735
- 优化 lock manager 的准确度 #5845
- 悲观锁支持 innodb_lock_wait_timeout #5848
- 添加 Titan 相关配置检测 #5720
- 支持用 tikv-ctl 动态修改 GC 限流配置：tikv-ctl --host=ip:port modify-tikv-config -m server -n gc
  ↪ .max_write_bytes_per_sec -v 10MB #5957
- 减少无用的 clean up 请求，降低死锁检测器的压力 #5965
- 悲观事务 prewrite 时避免 TTL 被缩短 #6056
- 修复 Titan 可能发生 missing blob file 的问题 #5968
- 修复 Titan 可能导致 RocksDBOptions 不生效的问题 #6009

16.20.15.3 PD

- 为每个过滤器添加一个名为 ActOn 的新维度，以指示每个 scheduler 和 checker 受过滤器的影响。删除两个未使用的过滤器：disconnectFilter 和 rejectLeaderFilter #1911
- 当 PD 生成时间戳的时间超过 5 毫秒时，将打印一条 warning 日志 #1867
- 当存在 endpoint 不可用时，降低 client 日志级别 #1856
- 修复 region_syncer 同步时 gRPC 消息包可能过大的问题 #1952

16.20.15.4 Tools

- TiDB Binlog
  - Drainer 配置 initial-commit-ts 为 -1 时，从 PD 处获取初始同步时间戳 #788

  - Drainer checkpoint 存储与下游解耦，支持选择配置 checkpoint 保存到 MySQL 或者本地文件 #790
  - 修复 Drainer 在配置同步库表过滤使用空值会导致 Panic 的问题 #801
  - 修复 Drainer 因为向下游应用 Binlog 失败而 Panic 后进程没有退出而是进入死锁状态的问题 #807
  - 修复 Pump 下线因为 gRPC 的 `GracefulStop` 流程而 hang 住的问题 #817
  - 修复 Drainer 在 TiDB 执行 DROP COLUMN DDL 期间收到缺少一列的 binlog 而同步出错的问题（要求 TiDB 3.0.6 以上）#827

- TiDB Lightning
  - TiDB Backend 模式新增 `max-allowed-packet` 配置项，默认值为 64M #248

16.20.16 TiDB 3.0.5 Release Notes

发版日期：2019 年 10 月 25 日

TiDB 版本：3.0.5

TiDB Ansible 版本：3.0.5

16.20.16.1 TiDB

- SQL 优化器
  - 支持对 Window Functions 进行边界检查 #12404
  - 修复 partition 表上的 `IndexJoin` 返回错误结果的问题 #12712
  - 修复外连接 Apply 算子上层的 `ifnull` 函数返回错误结果的问题 #12694
  - 修复当 UPDATE 的 `where` 条件中包含子查询时更新失败的问题 #12597
  - 修复当查询条件中包含 `cast` 函数时 outer join 被错误转化为 inner join 的问题 #12790
  - 修复 `AntiSemiJoin` 的 join 条件中错误的表达式传递 #12799
  - 修复初始化统计信息时由于浅拷贝造成的统计信息出错问题 #12817
  - 修复 TiDB 中 `str_to_date` 函数在日期字符串和格式化字符串不匹配的情况下，返回结果与 MySQL 不一致的问题 #12725
- SQL 执行引擎
  - 修复在 `from_unixtime` 函数处理 null 时发生 panic 的问题 #12551
  - 修复 Admin Cancel DDL jobs 时报 `invalid list index` 错的问题 #12671
  - 修复使用 Window Functions 时发生数组越界的问题 #12660
  - 改进 `AutoIncrement` 列隐式分配时的行为，与 MySQL 自增锁的默认模式（“consecutive” lock mode）保持一致：对于单行 Insert 语句的多个自增 `AutoIncrement` ID 的隐式分配，TiDB 保证分配值的连续性。该改进保证 JDBC `getGeneratedKeys()` 方法在任意场景下都能得到正确的结果。#12602
  - 修复当 HashAgg 作为 Apply 子节点时查询 hang 住的问题 #12766
  - 修复逻辑表达式 AND 或 OR 在涉及类型转换时返回错误结果的问题 #12811
- Server
  - 实现修改事务 TTL 的接口函数，以助后续支持大事务 #12397
  - 支持将事务的 TTL 按需延长（最长可到 10min），用于支持悲观事务 #12579
  - 将 TiDB 缓存 schema 变更及相关表信息的次数从 100 调整为 1024，且支持通过 `tidb_max_delta_schema_count` 系统变量修改 #12502

  - 更新了 kvrpc.Cleanup 协议的行为，不再清理未超时事务的锁 #12417
  - 支持将 Partition 表信息记录到 information_schema.tables 表 #12631
  - 支持通过 region-cache-ttl 配置修改 Region Cache 的 TTL #12683
  - 支持在慢日志中打印执行计划压缩编码后的信息，此功能默认开启，可以通过 slow-log-plan 配置或者 tidb_record_plan_in_slow_log 变量进行开关控制。另外支持 tidb_decode_plan 函数将慢日志中的执行计划列编码信息解析成执行计划信息。#12808
  - 在 information_schema.processlist 表中支持显示内存使用信息 #12801
  - 修复 TiKV Client 判断连接空闲时可能出错并出现非预期的告警的问题 #12846
  - 修复 tikvSnapshot 没有正确对 BatchGet() 的 KV 结果进行缓存，导致 INSERT IGNORE 语句性能有所下降的问题 #12872
  - 修复了因建立到部分 KV 服务的连接较慢最终导致 TiDB 响应速度相对变慢的情况 #12814

- DDL

  - 修复 Create Table 操作对 Set 列不能正确设置 Int 类型默认值的问题 #12267
  - 支持 Create Table 语句中建唯一索引时带多个 Unique #12463
  - 修复使用 Alter Table 添加 Bit 类型列时，对存在的行填充此列的默认值可能出错的问题 #12489
  - 修复 Range 分区表以 Date 或 Datetime 类型列作为分区键时，添加分区失败的问题 #12815
  - 对于 Date 或 Datetime 类型列作为分区键的 Range 分区表，在建表或者添加分区时，支持检查分区类型与分区键类型的统一性 #12792
  - 在创建 Range 分区表时，添加对 Unique Key 列集合需大于等于分区列集合的检查 #12718

- Monitor

  - 添加统计 Commit 与 Rollback 操作的监控到 Transaction OPS 面板 #12505
  - 添加统计 Add Index 操作进度的监控 #12390

16.20.16.2 TiKV

- Storage

  - 悲观事务新特性：事务 Cleanup 接口支持只清理 TTL 已经过期的锁 #5589
  - 修复事务 Primary key 的 Rollback 被折叠的问题 #5646，#5671
  - 修复悲观锁下点查可能返回历史旧版本的问题 #5634

- Raftstore

  - 减少 Raftstore 消息的 flush 操作，以提升性能，减少 CPU 占用 #5617
  - 优化获取 Region 的大小和 key 个数估计值的开销，减少心跳的开销，降低 CPU 占用 #5620
  - 修复 Raftstore 取到非法数据时打印错误日志并 panic 的问题 #5643

- Engine

  - 打开 RocksDB force_consistency_checks，提高数据安全性 #5662
  - 修复 Titan 并发 flush 情况下有可能造成数据丢失的问题 #5672
  - 更新 rust-rocksdb 版本以避开 intra-L0 compaction 导致 TiKV 崩溃重启的问题 #5710

16.20.16.3 PD

- 提高 Region 占用空间的精度 #1782

- 修复 `--help` 命令输出内容 #1763
- 修复 TLS 开启后 http 请求重定向失败的问题 #1777
- 修复 pd-ctl 使用 `store shows limit` 命令 panic 的问题 #1808
- 提高 label 监控可读性以及当 leader 发生切换后重置原 leader 的监控数据，防止误报 #1815

16.20.16.4 Tools

- TiDB Binlog
  - 修复 `ALTER DATABASE` 相关 DDL 会导致 Drainer 异常退出的问题 #769
  - 支持对 Commit binlog 查询事务状态信息，提升同步效率 #757
  - 修复当 Drainer 的 `start_ts` 大于 Pump 中最大的 `commit_ts` 时，有可能引起 Pump panic 的问题 #758
- TiDB Lightning
  - 整合 Loader 全量逻辑导入功能，支持配置 backend 模式 #221

16.20.16.5 TiDB Ansible

- 增加 TiDB 添加索引速度的监控 #986
- 精简配置文件内容，移除不需要用户配置的参数 #1043c，#998
- 修复 performance read 和 performance write 监控表达式错误的问题 #e90e7
- 更新 raftstore CPU 使用率的监控显示方式以及 raftstore CPU 使用率的告警规则 #992
- 更新 Overview 监控面板中 TiKV 的 CPU 监控项，过滤掉多余的监控内容 #1001

16.20.17 TiDB 3.0.4 Release Notes

发版日期：2019 年 10 月 8 日

TiDB 版本：3.0.4

TiDB Ansible 版本：3.0.4

- 新特性
  - 新增系统表 `performance_schema.events_statements_summary_by_digest`，用于排查 SQL 级别的性能问题
  - TiDB 的 `SHOW TABLE REGIONS` 语法新增 `WHERE` 条件子句
  - Reparo 新增 `worker-count` 和 `txn-batch` 配置项，用于控制恢复速率
- 改进提升
  - TiKV 支持批量 `Split` 和空的 `Split` 命令，使得 Split 可以批量进行
  - TiKV 添加 RocksDB 双向链表支持，提升逆序扫性能
  - Ansible 新增 `iosnoop` 和 `funcslower` 两个 perf 工具，方便诊断集群状态
  - TiDB 优化慢日志输出内容，删除冗余字段
- 行为变更

  - TiDB 修改 `txn-local-latches.enable` 默认值为 `false`，默认不启用本地事务冲突检测
  - TiDB 添加全局作用域系统变量 `tidb_txn_mode`，允许配置使用悲观锁，请注意默认情况下，TiDB 仍然使用乐观锁
  - TiDB 慢日志中的 `Index_ids` 字段替换为 `Index_names` 字段，提升慢日志易用性
  - TiDB 配置文件中添加 `split-region-max-num` 参数，用于调整 `SPLIT TABLE` 语法允许的最大 Region 数量
  - TiDB 修改 SQL 超出内存限制后的行为，从断开链接修改为返回 `Out Of Memory Quota` 错误
  - 为避免误操作，TiDB 默认不再允许删除列的 `AUTO_INCREMENT` 属性，当确实需要删除时，请更改系统变量 `tidb_allow_remove_auto_inc`

- 问题修复

  - TiDB 修复特殊语法 `PRE_SPLIT_REGIONS` 没有使用注释的方式向下游同步的问题
  - TiDB 修复使用游标获取 `PREPARE + EXECUTE` 执行结果时，慢日志不正确的问题
  - PD 修复相邻小 Region 无法 Merge 的问题
  - TiKV 修复空闲集群中文件描述符泄漏导致长期运行可能会引起 TiKV 进程异常退出的问题

- 社区贡献者

  感谢以下社区贡献者参与本次发版：

  - sduzh
  - lizhenda

16.20.17.1 TiDB

- SQL 优化器

  - 修复 `Feedback` 切分查询范围出错的问题 #12170
  - 修改当 `SHOW STATS_BUCKETS` 结果中包含无效 Key 时的行为，将返回错误修改为使用 16 进制显示 #12094
  - 修复查询中包含 `SLEEP` 函数时（例如 `select 1 from (select sleep(1))t;`），由于列裁剪导致查询中的 `sleep(1)` 失效的问题 #11953
  - 当查询只关心表的行数而不关心表数据时，使用索引扫描降低 IO #12112
  - 当 `use index()` 中没有指定索引时不去使用任何索引，和 MySQL 兼容（如 `explain select a from t use index();`）#12100
  - 严格限制统计信息 `CMSketch` 中 `TopN` 记录的数量，修复快速 `analyze` 因为超过事务大小限制而失败的问题 #11914
  - 修复 `Update` 语句包含子查询时，转换子查询出现的错误 #12483
  - 将 Limit 算子下推到 `IndexLookUpReader` 执行逻辑中优化 `select ... limit ... offset ...` 的执行性能 #12378

- SQL 执行引擎

  - `PREPARED` 语句执行错误时，在日志中打印 SQL 语句 #12191
  - 分区表使用 `UNIX_TIMESTAMP` 函数分区时，支持分区裁剪 #12169
  - 修复 `AUTO INCREMENT` 分配 `MAX int64` 和 `MAX uint64` 没有报错的问题 #12162
  - `SHOW TABLE ⋯ REGIONS` 和 `SHOW TABLE .. INDEX ⋯ REGIONS` 语法新增 `WHERE` 条件子句 #12123
  - 修改 SQL 超出内存限制后的行为，从断开链接修改为返回 `Out Of Memory Quota` 错误 #12127

- 修复 JSON_UNQUOTE 函数处理 JSON 文本结果不正确的问题 #11955
- 修复 INSERT 语句中，第一行中为 AUTO_INCREMENT 列赋值，LAST INSERT ID 不正确的问题（例如 insert into t (pk, c)values (1, 2), (NULL, 3)）#12002
- 修复 PREPARE 语句中，GroupBY 解析规则错误的问题 #12351
- 修复点查中权限检查不正确的问题 #12340
- 修复 PREPARE 语句类型没有记录在监控中的问题 #12331
- 支持点查中表名使用别名（例如 select * from t tmp where a = "aa"）#12282
- 修复向 BIT 类型列插入数值时，值没有作为无符号类型处理而导致插入负数报错的问题 #12423
- 修复时间取整不正确的问题（例如 2019-09-11 11:17:47.999999666 应该被取整到 2019-09-11 ↪ 11:17:48）#12258
- 调整表达式黑名单系统表的用法（例如 < 与 lt 等价）#11975
- 调整函数不存在的错误消息，添加数据库前缀（例如 [expression:1305]FUNCTION test.std_samp ↪ does not exist）#12111

- Server
  - 慢日志中添加 Prev_stmt 字段，用于最后一条语句是 COMMIT 时输出前一条语句 #12180
  - 优化慢日志输出内容，删除冗余字段 #12144
  - 修改 txn-local-latches.enable 默认值为 false，默认不启用本地事务冲突检测 #12095
  - 将慢日志中的 Index_ids 字段替换为 Index_names 字段，提升慢日志易用性 #12061
  - 添加全局作用域系统变量 tidb_txn_mode，允许配置使用悲观锁 #12049
  - 慢日志中添加 Backoff 字段，用来记录 2PC Commit 阶段的 Backoff 信息 #12335
  - 修复使用游标获取 PREPARE + EXECUTE 执行结果时，慢日志不正确的问题（例如 PREPARE stmt1FROM ↪ SELECT * FROM t WHERE a > ?; EXECUTE stmt1 USING @variable）#12392
  - 支持使用 tidb_enable_stmt_summary，开启后会对 SQL 语句进行统计，并可以使用系统表 performance_schema.events_statements_summary_by_digest 查询统计结果 #12308
  - 调整了 tikv-client 中部分日志级别（例如由于连接断开使得打印的 batchRecvLoop fails 日志级别由 ERROR 改为 INFO）#12383

- DDL
  - 新增变量 tidb_allow_remove_auto_inc，默认禁止删除列 AUTO_INCREMENT 属性 #12145
  - 修复 TiDB 特殊语法 PRE_SPLIT_REGIONS 没有使用注释的方式向下游同步，导致下游数据库报错的问题 #12120
  - 在配置文件中添加 split-region-max-num 参数，使得 SPLIT TABLE 语法允许的最大 Region 数量可调整，该参数默认值 10000 #12097
  - 支持将一个 Region 切分成多个 Region，并修复打散 Region 超时的问题 #12343
  - 修复当索引包含自增列，并且该自增列被两个索引引用时删除失败的问题 #12344

- Monitor
  - 增加监控指标 connection_transient_failure_count，用于统计 tikvclient 的 gRPC 连接错误数量 #12093

16.20.17.2 TiKV

- Raftstore
  - 修复 Raftstore 统计空 Region 中 key 个数不准确问题 #5414

  - 添加 RocksDB 双向链表支持，提升逆序扫性能 #5368
  - 支持 PD 批量 `Split` 和空的 `Split` 命令，使得 Split 可以批量进行，提高 Split 效率 #5470

- Server

  - 修复查看版本命令的输出格式与 2.X 格式不一致的问题 #5501
  - 更新 Titan 至 3.0 分支最新版本 #5517
  - 更新 grpcio 至 v0.4.5 版本 #5523
  - 修复 gRPC coredump 问题，支持内存共享，以避免此处引起 OOM #5524
  - 修复空闲集群中文件描述符泄漏导致长期运行可能会引起 TiKV 进程异常退出的问题 #5567

- Storage

  - 支持悲观锁事务心跳检测 API，以使得 TiDB 的悲观锁行为与 MySQL 尽量一致 #5507
  - 修复部分情况下点查性能较低的问题 #5495 #5463

16.20.17.3 PD

- 修复相邻小 Region 无法 Merge 的问题 #1726
- 修复 pd-ctl 的 TLS 启用参数失效问题 #1738
- 修复可能导致 PD operator 被意外移除的线程安全问题 #1734
- Region syncer 支持 TLS #1739

16.20.17.4 Tools

- TiDB Binlog

  - Reparo 新增 `worker-count` 和 `txn-batch` 配置项，用于控制恢复速率 #746
  - Drainer 优化内存使用，提升同步执行效率 #737

- TiDB Lightning

  - 修复从 checkpoint 点重新导入可能会导致 TiDB Lightning 崩溃的问题 #237
  - 修改计算 `AUTO_INCREMENT` 的算法，降低溢出的风险 #227

16.20.17.5 TiDB Ansible

- 更新 TiSpark 版本至 2.2.0 #926
- 更新 TiDB 配置项 `pessimistic_txn` 的默认值为 `true` #933
- 新增更多系统级别监控到 `node_exporter` #938
- 新增 `iosnoop` 和 `funcslower` 两个 perf 工具，方便诊断集群状态 #946
- Ansible 的 Raw 模块更新成 Shell 模块，解决密码过期等场景发生的长时间等待问题 #949
- 更新 TiDB 配置项 `txn_local_latches` 的默认值为 `false`
- 优化 Grafana dashboard 监控项和告警规则 #962 #963 #969
- 新增配置文件检查功能，在部署和升级之前检查配置文件是否正确 #934 #972

16.20.18 TiDB 3.0.3 Release Notes

发版日期：2019 年 8 月 29 日

TiDB 版本：3.0.3

TiDB Ansible 版本：3.0.3

16.20.18.1 TiDB

- SQL 优化器
  - 添加 `opt_rule_blacklist` 表，用于禁用一些逻辑优化规则，比如 `aggregation_eliminate`，`column_prune` 等 #11658
  - 修复 Index join 的 join key 中使用前缀索引或者使用 unsigned 的索引列等于负数时结果不正确的问题 #11759
  - 修复 `create ··· binding ...` 的 Select 语句中带有 ” 或者 \ 时解析报错的问题 #11726
- SQL 执行引擎
  - 修复 `Quote` 函数处理 null 值的返回值类型出错的问题 #11619
  - 修复 Max 和 Min 在推导类型时没有去除 NotNullFlag 导致 `ifnull` 结果错误的问题 #11641
  - 修复对字符形式的 Bit 类型数据比较出错的问题 #11660
  - 减少需要顺序读取数据的并发度，以降低 OOM 出现概率 #11679
  - 修复对应含有多个参数的内置函数（如 `if`、`coalesce` 等），在多个参数都为 unsigned 时类型推导不正确的问题 #11621
  - 修复 `Div` 函数处理 unsigned 的 decimal 类型时与 MySQL 行为不兼容的问题 #11813
  - 修复执行修改 Pump/Drainer 状态的 SQL 时会报 panic 的问题 #11827
  - 修复在 Autocommit = 1 且没有 begin 的时，`select ... for update` 出现 panic 的问题 #11736
  - 修复执行 `set default role` 语句时权限检查出错的问题 #11777
  - 修复执行 `create user` 和`drop user` 语句出现权限检查错误的问题 #11814
  - 修复 `select ... for update` 在构建为 PointGetExecutor 时会重试的问题 #11718
  - 修复 Window function 处理 Partition 时边界出错的问题 #11825
  - 修复 `time` 函数在处理错误格式参数时直接断链接的问题 #11893
  - 修复 Window function 没有检查传入参数的问题 #11705
  - 修复 Explain 查看的 Plan 结果跟真实执行的 Plan 结果不一致的问题 #11186
  - 修复 Window function 内存重复引用导致崩溃或结果不正确的问题 #11823
  - 修复 Slow log 里面 Succ 字段信息错误的问题 #11887
- Server
  - 重命名 `tidb_back_off_weight` 变量为 `tidb_backoff_weight` #11665
  - 更新与当前 TiDB 兼容的最低版本的 TiKV 为 v3.0.0 的信息 #11618
  - 支持 `make testSuite` 来确保测试中的 Suite 都被正确使用 #11685
- DDL
  - 禁止不支持的 Partition 相关的 DDL 的执行，其中包括修改 Partition 类型，同时删除多个 Partition 等 #11373
  - 禁止 Generated Column 的位置在它依赖的列前 #11686

- 修改添加索引操作中使用的 `tidb_ddl_reorg_worker_cnt` 和 `tidb_ddl_reorg_batch_size` 变量的默认值 #11874

- Monitor
  - Backoff 监控添加类型，且补充之前没有统计到的 Backoff，比如 commit 时遇到的 Backoff #11728

16.20.18.2 TiKV

- 修复 ReadIndex 请求可能由于重复 Context 而无法响应请求的问题 #5256
- 修复 `PutStore` 过早而引起一些调度造成抖动的问题 #5277
- 修复 Region Heartbeat 上报的时间戳不准的问题 #5296
- 剔除 share block cache 信息减少 coredump 文件大小 #5322
- 修复 Region merge 中会引起 TiKV panic 的问题 #5291
- 加快死锁检测器器的 leader 变更检查 #5317
- 使用 grpc env 创建 deadlock 的客户端 #5346
- 添加 `config-check` 检查配置是否正确 #5349
- 修复 ReadIndex 请求在没有 leader 情况下不返回的问题 #5351

16.20.18.3 PD

- `pdctl` 返回成功信息 #1685

16.20.18.4 Tools

- TiDB Binlog
  - 将 Drainer `defaultBinlogItemCount` 默认值从 65536 改为 512，减少 Drainer 启动时 OOM 的情况 #721
  - 优化 Pump server 下线处理逻辑，避免出现 Pump 下线阻塞的问题 #701
- TiDB Lightning
  - 导入时默认过滤系统库 `mysql`，`information_schema`，`performance_schema`，`sys` #225

16.20.18.5 TiDB Ansible

- 优化滚动升级 PD 的操作，提高稳定性 #894
- 移除当前 Grafana 版本不支持的 Grafana Collector 组件 #892
- 更新 TiKV 告警规则 #898
- 修复生成的 TiKV 配置遗漏 `pessimistic-txn` 参数的问题 #911
- 更新 Spark 版本为 2.4.3，同时更新 TiSpark 为兼容该 Spark 的 2.1.4 版本 #913 #918

16.20.19 TiDB 3.0.2 Release Notes

发版日期：2019 年 8 月 7 日

TiDB 版本：3.0.2

TiDB Ansible 版本：3.0.2

16.20.19.1 TiDB

- SQL 优化器
  - 修复当同一张表在查询里出现多次且逻辑上查询结果恒为空时报错“Can’t find column in schema”的问题 #11247
  - 修复了 TIDB_INLJ Hint 无法以指定表为 Inner 表构建 IndexJoin 时仍，会强制将其作为 Outer 表构建 IndexJoin，同时 Hint 可能会在不应生效的地方生效的错误，该错误是由于强制选取 IndexJoin 的判断逻辑有误，以及对表别名的处理有误导致的；该错误仅对包含 TIDB_INLJ 的查询产生影响 #11362
  - 修复某些情况下（例如 SELECT IF(1,c,c)FROM t），查询结果的列名称不正确的问题 #11379
  - 修复 LIKE 表达式某些情况下被隐式转换为 0，导致诸如 SELECT 0 LIKE 'a string' 返回结果为 TRUE 的问题 #11411
  - 支持在 SHOW 语句中使用子查询，现在可以支持诸如 SHOW COLUMNS FROM tbl WHERE FIELDS IN (↪ SELECT 'a') 的写法 #11459
  - 修复 outerJoinElimination 优化规则没有正确处理列的别名，导致找不到聚合函数的相关列而查询报错的问题；改进了优化过程中对别名的解析，以使得优化可以覆盖更多类型的查询 #11377
  - 修复 Window Function 中多个违反语义约束（例如 UNBOUNDED PRECEDING 不允许在 Frame 定义的最后）时没有报错的问题 #11543
  - 修 复 ERROR 3593 (HY000): You cannot use the window function FUNCTION_NAME in this ↪ context 报错信息中，FUNCTION_NAME 不为小写的问题，导致与 MySQL 不兼容 #11535
  - 修复 Window Function 中 IGNORE NULLS 语法尚未实现，但使用时没有报错的问题 #11593
  - 修复优化器对时间类型数据的等值条件代价估算不准确的问题 #11512
  - 支持根据反馈信息对统计信息 Top-N 进行更新 #11507
- SQL 执行引擎
  - 修复 INSERT 函数在参数中包含 NULL 时，返回值不为 NULL 的问题 #11248
  - 修复 ADMIN CHECKSUM 语句在检查分区表时计算结果不正确的问题 #11266
  - 修复 INDEX JOIN 在使用前缀索引时可能结果不正确的问题 #11246
  - 修复 DATE_ADD 函数在进行涉及微秒的日期减法时，没有正确地对日期的小数位数进行对齐导致结果不正确的问题 #11288
  - 修复 DATE_ADD 函数没有正确地对 INTERVAL 中的负数部分处理导致结果不正确的问题 #11325
  - 修复 Mod(%)、Multiple(*) 和 Minus(-) 返回结果为 0 时，在小数位数较多（例如 select 0.000 % ↪ 0.11234500000000000000）的情况下与 MySQL 位数不一致的问题 #11251
  - 修复 CONCAT 和 CONCAT_WS 函数在返回结果长度超过 max_allowed_packet 时，没有正确返回 NULL 和 Warning 的问题 #11275
  - 修复 SUBTIME 和 ADDTIME 函数在参数不合法时，没有正确返回 NULL 和 Warning 的问题 #11337
  - 修复 CONVERT_TZ 函数在参数不合法时，没有正确返回 NULL 的问题 #11359
  - EXPLAIN ANALYZE 结果中添加了 MEMORY 列，显示 QUERY 的内存使用 #11418
  - EXPLAIN 结果中，为笛卡尔积 Join 添加了 CARTESIAN 关键字 #11429
  - 修复类型为 FLOAT 和 DOUBLE 的自增列数据不正确的问题 #11385
  - 修复 Dump Pseudo Statistics 时，由于部分信息为 nil 导致 panic 的问题 #11460
  - 修复常量折叠优化导致 SELECT ··· CASE WHEN ··· ELSE NULL ··· 查询结果不正确的问题 #11441
  - 修复 floatStrToIntStr 对诸如 +999.9999e2 的输入没有正确解析的问题 #11473
  - 修复 DATE_ADD 和 DATE_SUB 函数结果超出合法范围时，某些情况下不会返回 NULL 的问题 #11476
  - 修复长字符串转换为整型时，若字符串包含不合法字符，转换结果与 MySQL 不一致的问题 #11469
  - 修复 REGEXP BINARY 函数对大小写敏感，导致与 MySQL 不兼容的问题 #11504

- 修复 GRANT ROLE 语句在接受 CURRENT_ROLE 时报错的问题；修复 REVOKE ROLE 语句没有能够正确收回 mysql.default_role 权限的问题 #11356
- 修复执行诸如 SELECT ADDDATE('2008-01-34', -1) 时，Incorrect datetime value Warning 信息的显示格式问题 #11447
- 修复将 JSON 数据中的 Float 类型字段转为 Int 类型溢出时，报错信息中应当提示 constant … overflows bigint 而不应当为 constant … overflows float 的问题 #11534
- 修复 DATE_ADD 函数接受 FLOAT、DOUBLE 和 DECIMAL 类型的列参数时，没有正确地进行类型转换而导致结果可能不正确的问题 #11527
- 修复 DATE_ADD 函数中，没有正确处理 INTERVAL 小数部分的符号而导致结果不正确的问题 #11615
- 修复 Ranger 没有正确处理前缀索引，导致 Index Lookup Join 中包含前缀索引时，查询结果不正确的问题 #11565
- 修复 NAME_CONST 函数第二个参数为负数时执行会报 Incorrect arguments to NAME_CONST 的问题 #11268
- 修复一条 SQL 语句在涉及当前时间计算时（例如 CURRENT_TIMSTAMP 或者 NOW），多次取当前时间值，结果与 MySQL 不兼容的问题：现在同一条 SQL 语句中取当前时间时，均使用相同值 #11394
- 修复了父 Executor Close 出现错误时，没有对 ChildExecutor 调用 Close 的问题，该问题可能导致 KILL 语句失效时，子 ChildExecutor 没有关闭而导致 Goroutine 泄露 #11576

- Server
  - 修复 LOAD DATA 处理 CSV 文件中缺失的 TIMESTAMP 字段时，自动补充的值是 0 不是当前时间戳的问题 #11250
  - 修复 SHOW CREATE USER 语句没有正确检查相关权限的问题，以及 SHOW CREATE USER CURRENT_USER() 结果中 USER、HOST 可能不正确的问题 #11229
  - 修复在 JDBC 中使用 executeBatch 可能返回结果不正确的问题 #11290
  - TiKV Server 在更换端口时，减少 Streaming Client 的报错信息的日志打印 #11370
  - 优化 Streaming Client 在重新与 TiKV Server 连接时的逻辑：现在 Streaming Client 不会长时间被 Block #11372
  - INFORMATION_SCHEMA.TIDB_HOT_REGIONS 中新增 REGION_ID#11350
  - 取消了从 PD API 获取 Region 相关信息时的超时时间，保证在 Region 数量较大时，调用 TiDB API http://{TiDBIP}:10080/regions/hot 不会因为 PD 超时而获取失败 #11383
  - 修复 HTTP API 中，与 Region 相关的请求没有返回分区表相关的 Region 问题 #11466
  - 做以下改动以降低用户手动验证悲观锁时，操作较慢导致锁超时的概率 #11521：
    * 悲观锁的默认 TTL 时间由 30 秒提升为 40 秒
    * 最大允许的 TTL 时间由 60 秒提升为 120 秒
    * 悲观锁的持续时间改为从第一次 LockKeys 请求时开始计算
  - 修改 TiKV Client 中的 SendRequest 函数逻辑：当连接无法建立时，由一直等待改为尽快尝试连接其他 Peer #11531
  - 优化 Region Cache：当一个 Store 下线，同时另一个 Store 以同样的地址上线时，将已下线的 Store 标记为失效以尽快在 Cache 中更新 Store 的信息 #11567
  - 为 http://{TiDB_ADDRESS:TIDB_IP}/mvcc/key/{db}/{table}/{handle} API 的返回结果添加 Region ID 信息 #11557
  - 修复 Scatter Table API 没有对 Range Key 进行转义导致 Scatter Table 不生效的问题 #11298
  - 优化 Region Cache：当 Region 所在的 Store 无法访问时，将对应的 Store 信息标记失效以避免对这些 Store 的访问造成查询性能下降 #11498
  - 修复了多次 DROP 同名 DATABASE 后，DATABASE 内的表结构仍然能够通过 HTTP API 获取到的错误

#11585

- DDL
  - 修复在非字符串类型且长度为 0 的列建立索引时出错的问题 #11214
  - 禁止对带有外键约束和全文索引的列进行修改（注意：TiDB 仍然仅在语法上支持外键约束和全文索引）#11274
  - 修复并发使用 ALTER TABLE 语句更改的位置和列的默认值时，可能导致列的索引 Offset 出错的问题 #11346
  - 修复解析 JSON 文本的两个问题：
    * ConvertJSONToFloat 中使用 int64 作为 uint64 的中间解析结果，导致精度溢出的问题 #11433
    * ConvertJSONToInt 中使用 int64 作为 uint64 的中间解析结果，导致精度溢出的问题 #11551
  - 禁止 DROP 自增列索引，修复因为 DROP 自增列上的索引导致自增列结果可能出错的问题 #11399
  - 修复以下问题 #11492：
    * 修复显式指定列的排序规则但没有指定字符集时，列的字符集与排序规则不一致的问题
    * 修复 ALTER TABLE ··· MODIFY COLUMN 指定的字符集和排序规则冲突时，没有正确报错的问题
    * 修复 ALTER TABLE ··· MODIFY COLUMN 指定多次字符集和排序规则时，行为与 MySQL 不兼容的问题
  - 为 TRACE 语句的结果添加子查询的 trace 细节信息 #11458
  - 优化 ADMIN CHECK TABLE 执行性能，大幅降低了语句的执行耗时 #11547
  - 为 SPLIT TABLE ··· REGIONS/INDEX 添加了返回结果，结果包含 TOTAL_SPLIT_REGION 和 SCATTER_FINISH_RATIO 展示在超时时间内，切分成功的 Region 数量 #11484
  - 修复 ON UPDATE CURRENT_TIMESTAMP 作为列的属性且指定浮点精度时，SHOW CREATE TABLE 等语句显示精度不完整的问题 #11591
  - 修复一个虚拟生成列的表达式中含有另一个虚拟生成列时，该列的索引结果不能正确被计算的问题 #11475
  - 修复 ALTER TABLE ··· ADD PARTITION ··· 语句中，VALUE LESS THAN 后不能出现负号的问题 #11581
- Monitor
  - 修复 TiKVTxnCmdCounter 监控指标没有注册导致数据没有被收集上报的问题 #11316
  - 为 Bind Info 添加了 BindUsageCounter、BindTotalGauge 和 BindMemoryUsage 监控指标 #11467

16.20.19.2 TiKV

- 修复由于 Raft Log 写入不及时可能导致 TiKV panic 的 bug #5160
- 修复 TiKV panic 后 panic 信息不会写入日志的 bug #5198
- 修复了悲观事务下 Insert 行为可能不正确的 bug #5203
- 降低一部分不需要人工干预的日志输出级别为 INFO #5193
- 提高存储引擎大小监控项的准确程度 #5200
- 提高 tikc-ctl 中 Region size 的准确程度 #5195
- 提高悲观锁死锁检测性能 #5192
- 提高 Titan 存储引擎 GC 性能 #5197

16.20.19.3 PD

- 修复 Scatter Region 调度器不能工作的 bug #1642

- 修复 pd-ctl 中不能进行 merge Region 操作的 bug #1653
- 修复 pd-ctl 中不能进行 remove-tombstone 操作的 bug #1651
- 修复 scan region 不能找到 key 范围相交的 Region 的问题 #1648
- 增加重试机制确保 PD 增加成员成功 #1643

16.20.19.4 Tools

TiDB Binlog

- 增加启动时配置项检查功能，遇到不合法配置项会退出运行并给出错误信息 #687
- Drainer 增加 `node-id` 配置，用于指定固定逻辑 Drainer #684

TiDB Lightning

- 修复 2 个 checksum 同时运行的情况下，`tikv_gc_life_time` 没有正常修改回原本值的问题 #218
- 增加启动时配置项检查功能，遇到不合法配置项会退出运行并给出错误信息 #217

16.20.19.5 TiDB Ansible

- 修复 Disk Performance 监控把 second 作为 ms 的单位错误的问题 #840
- Spark 新增 log4j 日志配置 #841
- 修复在开启了 Binlog 并且设置了 Kafka 或者 ZooKeeper 时导致生成的 Prometheus 配置文件格式错误的问题 #844
- 修复生成的 TiDB 配置文件中遗漏 `pessimistic-txn` 配置参数的问题 #850
- TiDB Dashboard 新增和优化 Metrics #853
- TiDB Dashboard 上每个监控项增加描述 #854
- 新增 TiDB Summary Dashboard，用于更好的查看集群状态和排查问题 #855
- TiKV Dashboard 更新 Allocator Stats 监控项 #857
- 修复 Node Exporter 的告警表达式单位错误的问题 #860
- 更新 tispark jar 包为 v2.1.2 版本 #862
- 更新 Ansible Task 功能描述 #867
- 兼容 TiDB 变更，TiDB Dashboard 更新 Local reader requests 监控项的表达式 #874
- Overview Dashboard 更新 TiKV Memory 监控项的表达式，修复监控显示错误的问题 #879
- 移除 Kafka 模式 Binlog 的支持 #878
- 修复执行 `rolling_update.yml` 操作时，切换 PD Leader 失效的 bug #887

16.20.20 TiDB 3.0.1 Release Notes

发版日期：2019 年 7 月 16 日

TiDB 版本：3.0.1

TiDB Ansible 版本：3.0.1

16.20.20.1 TiDB

- 新增对 MAX_EXECUTION_TIME 特性的支持 #11026
- 新增 tidb_wait_split_region_finish_backoff Session 变量，用于控制 Region 打散的 Backoff 时间 #11166
- 新增根据负载情况自动调整 Auto ID 分配的步长功能，步长自动调整范围最小 1000，最大 2000000 #11006
- 新增 ADMIN PLUGINS ENABLE/ADMIN PLUGINS DISABLE SQL 语句，管理 Plugin 的动态开启或关闭 #11157
- Audit Plugin 新增审计连接功能 #11013
- 修改 Region 打散时的默认行为为等待 PD 调度完成 #11166
- 禁止 Window Function 在 Prepare Plan Cache 中被缓存，避免某些情况下出现结果不正确的问题 #11048
- 禁止使用 Alter 语句修改 Stored Generated Column 的定义 #11068
- 禁止将 Virtual Generated Column 更改为 Stored Generated Column #11068
- 禁止改变带有索引的 Generated Column 的表达式 #11068
- 支持 TiDB 在 ARM64 架构下的编译 #11150
- 支持修改 Database/Table 的 Collate，条件限制为 Database/Table 字符集必须是 UTF8/UTF8MB4 #11086
- 修复 UPDATE ··· SELECT 语句中，SELECT 子查询没有解析到 UPDATE 表达式中的列而被误裁剪，导致报错的问题 #11252
- 修复点查时，某列被查询多次而且结果为 NULL 时会 Panic 的问题 #11226
- 修复 RAND 函数由于非线程安全的 rand.Rand 导致的 Data Race 问题 #11169
- 修复 oom-action="cancel" 时，某些情况下 SQL 内存使用超阈值没有被取消执行，返回结果不正确的问题 #11004
- 修复 MemTracker 未正确清理统计的内存使用值导致 SHOW PROCESSLIST 显示内存使用不为 0 的问题 #10970
- 修复某些情况下整数和非整数比较结果不正确的问题 #11194
- 修复在显式事务中查询对 Table Partition 的查询包含谓词时，查询结果不正确的问题 #11196
- 修复 DDL Job 由于 infoHandle 可能为 NULL 导致 Panic 的问题 #11022
- 修复嵌套聚合查询时，由于被查询列在子查询中没有引用而被误裁剪导致查询结果错误的问题 #11020
- 修复 Sleep 函数响应 Kill 命令不及时的问题 #11028
- 修复 SHOW PROCESSLIST 命令显示的 DB 和 INFO 列与 MySQL 不兼容的问题 #11003
- 修复 skip-grant-table=true 时，FLUSH PRIVILEGES 语句导致系统 Panic 的问题 #11027
- 修复表主键为 UNSIGNED 整数时，FAST ANALYZE 收集主键的统计信息不正确的问题 #11099
- 修复某些情况下 FAST ANALYZE 语句报 “invalid key” Error 的问题 #11098
- 修复 CURRENT_TIMESTAMP 作为列的默认值且指定浮点精度时，SHOW CREATE TABLE 等语句显示精度不完整的问题 #11088
- 修复窗口函数报错时函数名没有小写的问题，兼容 MySQL #11118
- 修复 TiKV Client Batch gRPC 的后台线程 panic 后导致 TiDB 无法正常连接 TiKV 进而无法提供服务的问题 #11101
- 修复 SetVar 方法由于字符串浅拷贝导致设置的变量不正确的问题 #11044
- 修复 INSERT ··· ON DUPLICATE 语句作用在 Table Partition 时执行失败报错的问题 #11231
- 悲观锁（实验性特性）
  - 修复悲观锁进行点查且数据为空时，由于行锁未生效导致结果不正确的问题 #10976
  - 修复使用悲观锁查询时由于没有使用 SELECT ··· FOR UPDATE 的 TSO 导致查询结果不正确的问题 #11015
  - 修改乐观锁与悲观锁同时使用时，乐观事务遇到悲观锁冲突时，检测行为由立即检测冲突修改为等待，防止锁冲突进一步恶化 #11051

16.20.20.2 TiKV

- 统计信息中新增对 Blob 文件大小的统计 #5060
- 修复由于进程退出未正确清理内存资源导致进程在退出时 core dump 问题 #5053
- 新增与 Titan 引擎相关的所有监控指标 #4772，#4836
- 统计打开文件句柄数量时，新增 Titan 引擎打开文件句柄数量，防止因文件句柄数统计不准确导致系统无文件句柄可用的问题 #5026
- 通过设置 `blob_run_mode` 来决定是否在某个 CF 上启动 Titan 引擎 #4991
- 修复读操作读不到悲观事务 commit 信息的问题 #5067
- 新增 `blob-run-mode` 配置参数控制 Titan 引擎的运行模式，取值：normal、 read-only、 fallback #4865
- 提升死锁检测的性能 #5089

16.20.20.3 PD

- 修复热点 Region 调度时，调度限制会自动调整为 0 的问题 #1552
- 新增 `enable-grpc-gateway` 的配置选项，用于开启 etcd 的 grpc gateway 功能 #1596
- 新增 `store-balance-rate`、 `hot-region-schedule-limit` 等与调度器配置相关的统计信息 #1601
- 优化热点 Region 调度策略，调度时跳过缺失副本的 Region，防止多个副本调度到同一个机房 #1609
- 优化 Region Merge 处理逻辑，优先 Merge Region Size 较小的 Region，提升 Region Merge 的速度 #1613
- 优化单次调度热点 Region 的限制值为 64，防止调度任务过多占用系统资源，影响性能 #1616
- 优化 Region 调度策略，新增优先调度 Pending 状态的 Region 功能 #1617
- 修复无法添加 `random-merge` 和 `admin-merge-region` operator 的问题 #1634
- 调整日志中输出 Region 中 Key 的格式为 16 进制，方便用户查看 #1639

16.20.20.4 Tools

TiDB Binlog

- 优化 Pump GC 策略，删除保证未被消费的 Binlog 不被清理的限制，确保资源不会长期占用 #646

TiDB Lightning

- 修正 SQL dump 指明的列名不是小写时导入错误的问题 #210

16.20.20.5 TiDB Ansible

- 新增 `ansible` 命令及其 `jmespath`、 `Jinja2` 依赖包的预检查功能 #803，#813
- Pump 新增 `stop-write-at-available-space` 参数，控制当磁盘剩余空间小于该值（默认 10 GiB）时，Pump 停止写入 Binlog #806
- 更新 TiKV 监控中的 IO 监控项，兼容新版本监控组件 #820
- 更新 PD 监控信息，并修复 Disk Performance Dashboard 中 Disk Latency 显示为空的异常 #817
- TiKV Details Dashboard 新增 Titan 监控项 #824

16.20.21 TiDB 3.0 GA Release Notes

发版日期：2019 年 6 月 28 日

TiDB 版本：3.0.0

TiDB Ansible 版本：3.0.0

16.20.21.1 Overview

2019 年 6 月 28 日，TiDB 发布 3.0 GA 版本，对应的 TiDB Ansible 版本为 3.0.0。相比于 V2.1，V3.0.0 版本在以下方面有重要改进：

- 稳定性方面，显著提升了大规模集群的稳定性，集群支持 150+ 存储节点，300+TB 存储容量长期稳定运行。
- 易用性方面有显著的提升，降低用户运维成本，例如：标准化慢查询日志，制定日志文件输出规范，新增 `EXPLAIN ANALYZE`，SQL Trace 功能方便排查问题等。
- 性能方面，与 2.1 相比，TPC-C 性能提升约 4.5 倍，Sysbench 性能提升约 1.5 倍。因支持 View，TPC-H 50G Q15 可正常运行。
- 新功能方面增加了窗口函数、视图（实验特性）、分区表、插件系统、悲观锁（实验特性）、`SQL Plan Management` 等特性。

16.20.21.2 TiDB

- 新功能
  - 新增 Window Function，支持所有 MySQL 8.0 中的窗口函数，包括 `NTILE`，`LEAD`，`LAG`、`PERCENT_RANK`、`NTH_VALUE`、`CUME_DIST`、`FIRST_VALUE`、`LAST_VALUE`、`RANK`、`DENSE_RANK`、`ROW_NUMBER` 函数
  - 新增 View 功能（实验特性）
  - 完善 Table Partition 功能：
    * Range Partition
    * Hash Partition
  - 新增插件系统，官方提供 `IP` 白名单（企业版特性），审计日志（企业版特性）等插件
  - 新增 `SQL Plan Management` 功能，通过绑定 SQL 执行计划确保查询的稳定性（实验特性）
- SQL 优化器
  - 优化`NOT EXISTS` 子查询，转化为 Anti Semi Join 提升性能
  - 优化 `Outer Join` 常量传播，新增 `Outer Join` 消除优化规则，避免无效计算，提升性能
  - 优化 `IN` 子查询，先聚合后执行 `Inner Join`，提升性能
  - 优化 Index Join，适应更多的场景，提升性能
  - 优化 Range Partition 的 Partition Pruning 优化规则，提升性能
  - 优化 `_tidb_rowid` 查询逻辑，避免全表扫描，提升性能
  - 当过滤条件中包含相关列时，在抽取复合索引的访问条件时尽可能多地匹配索引的前缀列，提升性能
  - 利用列之间的顺序相关性，提升代价估算准确度
  - 基于统计信息的贪心算法及动态规划算法改进了 `Join Order`，提升多表关联的执行速度
  - 新增 Skyline Pruning，利用规则防止执行计划过于依赖统计信息，提升查询的稳定性
  - 提升单列索引上值为 NULL 时行数估算准确度
  - 新增 `FAST ANALYZE`，通过在各个 Region 中随机采样避免全表扫描的方式提升统计信息收集性能
  - 新增单调递增的索引列增量 `Analyze` 功能，提升统计信息收集性能
  - 支持 `DO` 语句中使用子查询
  - 支持在事务中使用 Index Join
  - 优化 `prepare/execute`，支持不带参数的 DDL 语句
  - 修改变量 `stats-lease` 值为 0 时系统的行为，使其自动加载统计

  - 新增导出历史统计信息功能
  - 新增导入导出列的关联性信息功能

- SQL 执行引擎
  - 优化日志输出，EXECUTE 语句输出用户变量，COMMIT 语句输出慢查询日志，方便排查问题
  - 新增 EXPLAIN ANALYZE 功能，提升 SQL 调优易用性
  - 新增 admin show next_row_id 功能，方便获取下一行 ID
  - 新增 JSON_QUOTE、JSON_ARRAY_APPEND、JSON_MERGE_PRESERVE、BENCHMARK、COALESCE、NAME_CONST 6 个内建函数
  - 优化 Chunk 大小控制逻辑，根据查询上下文件动态调整，降低 SQL 执行时间和资源消耗，提升性能
  - 新增 TableReader、IndexReader 和 IndexLookupReader 算子内存追踪控制
  - 优化 Merge Join 算子，使其支持空的 ON 条件
  - 优化单个表列较多时写入性能，提升数倍性能
  - 通过支持逆序扫数据提升 admin show ddl jobs 的性能
  - 新增 split table region 语句，手动分裂表的 Region，缓解热点问题
  - 新增 split index region 语句，手动分裂索引的 Region，缓解热点问题
  - 新增黑名单禁止下推表达式到 Coprocessor 功能
  - 优化 Expensive Query 日志，在日志中打印执行时间或者使用内存超过阈值的 SQL 查询

- DDL
  - 支持字符集从 utf8 转换到 utf8mb4 的功能
  - 修改默认字符集从 utf8 变为 utf8mb4
  - 新增 alter schema 语句修改数据库 charset 和 collation 功能
  - 新增 ALTER ALGORITHM INPLACE/INSTANT 功能
  - 新增 SHOW CREATE VIEW 功能
  - 新增 SHOW CREATE USER 功能
  - 新增快速恢复误删除的表功能
  - 新增动态调整 ADD INDEX 的并发数功能
  - 新增 pre_split_regions 选项，在 CREATE TABLE 时预先分配 Region，缓解建表后大量写入造成的写热点问题
  - 新增通过 SQL 语句指定表的索引及范围分裂 Region，缓解热点问题
  - 新增 ddl_error_count_limit 全局变量，控制 DDL 任务重次数
  - 新增列属性包含 AUTO_INCREMENT 时利用 SHARD_ROW_ID_BITS 打散行 ID 功能，缓解热点问题
  - 优化无效 DDL 元信息存活时间，使集群升级后一段时间 DDL 操作比较慢的情况变短

- 事务
  - 新增悲观事务模式（实验特性）
  - 优化事务处理逻辑，适应更多场景，具体如下：
    * tidb_disable_txn_auto_retry 的默认值为 on，即不会重试非自动提交的事务
    * 新增 tidb_batch_commit 系统变量控制将事务拆分成多个事务并发执行
    * 新增 tidb_low_resolution_tso 系统变量控制批量获取 tso 个数，减少事务获取 tso 的次数以适应某些数据一致性要求较低的场景
    * 新增 tidb_skip_isolation_level_check 变量控制事务检查隔离级别设置为 SERIALIZABLE 时是否报错
    * 修改 tidb_disable_txn_auto_retry 系统变量的行为，修改为影响所有的可重试错误

- 权限管理 - 对 ANALYZE、USE、SET GLOBAL、SHOW PROCESSLIST 语句进行权限检查 - 新增基于角色的权限访问控制功能 (RBAC)（实验特性）
- Server
  - 优化慢查询日志，具体包括：
    * 重构慢查询日志格式
    * 优化慢查询日志内容
    * 优化查询慢查询日志的方法，通过内存表 INFORMATION_SCHEMA.SLOW_QUERY，ADMIN SHOW SLOW 语句查询慢查询日志
  - 制定日志格式规范，重构日志系统，方便工具收集分析
  - 新增 SQL 语句管理 TiDB Binlog 服务功能，包括查询状态，开启 TiDB Binlog，维护发送 TiDB Binlog 策略
  - 新增通过 unix_socket 方式连接数据库
  - 新增 SQL 语句 Trace 功能
  - 新增 /debug/zip HTTP 接口，获取 TiDB 实例的信息，方便排查问题
  - 优化监控项，方便排查问题，如下：
    * 新增 high_error_rate_feedback_total 监控项，监控真实数据量与统计信息估算数据量之间的差距
    * 新增 Database 维度的 QPS 监控项
  - 优化系统初始化流程，仅允许 DDL Owner 执行初始化操作，缩短初始化或升级过程中的启动时间
  - 优化 kill query 语句执行逻辑，提升性能，确保资源正确释放
  - 新增启动选项 config-check 检查配置文件合法性
  - 新增 tidb_back_off_weight 系统变量，控制内部出错重试的退避时间
  - 新增 wait_timeout、interactive_timeout 系统变量，控制连接空闲超过变量的值，系统自动断开连接。
  - 新增连接 TiKV 的连接池，减少连接创建时间
- 兼容性
  - 支持 ALLOW_INVALID_DATES SQL mode
  - 支持 MySQL 320 握手协议
  - 支持将 unsigned bigint 列声明为自增列
  - 支持 SHOW CREATE DATABASE IF NOT EXISTS 语法
  - 优化 load data 对 CSV 文件的容错
  - 过滤条件中包含用户变量时谓词不下推，兼容 MySQL Window Function 中使用用户变量行为

16.20.21.3 PD

- 新增从单个节点重建集群的功能
- 将 Region 元信息从 etcd 移到 go-leveldb 存储引擎，解决大规模集群 etcd 存储瓶颈问题
- API
  - 新增 remove-tombstone 接口，用于清理 Tombstone Store
  - 新增 ScanRegions 接口，用于批量查询 Region 信息
  - 新增 GetOperator 接口，用于查询运行中的 Operator
  - 优化 GetStores 接口的性能

- 配置
  - 优化配置检查逻辑，防止配置项错误
  - 新增 `enable-two-way-merge`，用于控制 Region merge 的方向
  - 新增 `hot-region-schedule-limit`，用于控制热点 Region 调度速度
  - 新增 `hot-region-cache-hits-threshold`，连续命中阈值用于判断热点
  - 新增 `store-balance-rate` 配置，用于控制每分钟产生 balance Region Operator 数量的上限
- 调度器优化
  - 添加 Store Limit 机制限制调度速度，使得速度限制适用于不同规模的集群
  - 添加 `waitingOperator` 队列，用于优化不同调度器之间资源竞争的问题
  - 支持调度限速功能，主动向 TiKV 下发调度操作，限制单节点同时执行调度任务的个数，提升调度速度
  - Region Scatter 调度不再受 limit 机制限制，提升调度的速度
  - 新增 `shuffle-hot-region` 调度器，解决稳定性测试易用性问题
- 模拟器
  - 新增数据导入场景模拟
  - 新增为 Store 设置不同的心跳间隔的功能
- 其他
  - 升级 etcd，解决输出日志格式不一致，prevote 时选举不出 Leader，Lease 死锁等问题
  - 制定日志格式规范，重构日志系统，方便工具收集分析
  - 新增调度参数，集群 Label 信息，PD 处理 TSO 请求的耗时，Store ID 与地址信息等监控指标

16.20.21.4 TiKV

- 新增分布式 GC 以及并行 Resolve Lock 功能，提升 GC 的性能
- 新增逆向 `raw_scan` 和 `raw_batch_scan` 功能
- 新增多线程 Raftstore 和 Apply 功能，提升单节点内可扩展性，提升单节点内并发处理能力，提升单节点的资源利用率，降低延时，同等压力情况下性能提升 70%
- 新增批量接收和发送 Raft 消息功能，写入密集的场景 TPS 提升 7%
- 新增 Apply snapshot 之前检查 RocksDB level 0 文件的优化，避免产生 Write stall
- 新增 Titan 存储引擎插件，提升 Value 超过 1KiB 时系统的性能，一定程度上缓解写放大问题（实验特性）
- 新增悲观事务模式（实验特性）
- 新增通过 HTTP 方式获取监控信息功能
- 修改 Insert 语义，仅在 Key 不存在的时候 Prewrite 才成功
- 制定日志格式规范，重构日志系统，方便工具收集分析
- 新增配置信息，Key 越界相关的性能监控指标

- RawKV 使用 Local Reader，提升性能
- Engine
  - 优化内存管理，减少 `Iterator Key Bound Option` 的内存分配和拷贝，提升性能
  - 支持多个 column family 共享 block cache，提升资源的利用率
- Server
  - 优化 `batch commands` 的上下文切换开销，提升性能
  - 删除 txn scheduler
  - 新增 read index，GC worker 相关监控项
- RaftStore
  - 新增 hibernate Regions 功能，优化 RaftStore CPU 的消耗（实验特性）
  - 删除 local reader 线程
- Coprocessor
  - 重构计算框架，实现向量化算子、向量化表达式计算、向量化聚合，提升性能
  - 支持为 TiDB `EXPLAIN ANALYZE` 语句提供算子执行详情
  - 改用 work-stealing 线程池模型，减少上下文切换

16.20.21.5 Tools

- TiDB Lightning
  - 支持数据表重定向同步功能
  - 新增导入 CSV 文件功能
  - 提升 SQL 转 KV 对的性能
  - 单表支持批量导入功能，提升单表导入的性能
  - 支持将大表的数据和索引分别导入，提升 `TiKV-Importer` 导入数据性能
  - 支持对新增文件中缺少 Column 数据时使用 row id 或者列的默认值填充缺少的 column 数据
  - `TiKV-Importer` 支持对 upload SST 到 TiKV 限速功能
- TiDB Binlog
  - Drainer 新增 `advertise-addr` 配置，支持容器环境中使用桥接模式
  - Pump 使用 TiKV GetMvccByKey 接口加快事务状态查询
  - 新增组件之间通讯数据压缩功能，减少网络资源消耗
  - 新增 Arbiter 工具支持从 Kafka 读取 binlog 并同步到 MySQL 功能
  - Reparo 支持过滤不需要被同步的文件的功能
  - 新增同步 Generated column 功能
  - 新增 syncer.sql-mode 配置项，支持采用不同的 SQL mode 解析 DDL
  - 新增 syncer.ignore-table 配置项，过滤不需要被同步的表
- sync-diff-inspector
  - 新增 checkpoint 功能，支持从断点继续校验的功能
  - 新增 only-use-checksum 配置项，控制仅通过计算 checksum 校验数据的一致性
  - 新增采用 TiDB 统计信息以及使用多个 Column 划分 Chunk 的功能，适应更多的场景

16.20.21.6 TiDB Ansible

- 升级监控组件版本到安全的版本
  - Prometheus 从 2.2.1 升级到 2.8.1 版本
  - Pushgateway 从 0.4.0 升级到 0.7.0 版本
  - Node_exporter 从 0.15.2 升级到 0.17.0 版本
  - Alertmanager 从 0.14.0 升级到 0.17.0 版本
  - Grafana 从 4.6.3 升级到 6.1.6 版本
  - Ansible 从 2.5.14 升级到 2.7.11 版本
- 新增 TiKV summary 监控面板，方便查看集群状态
- 新增 TiKV trouble_shooting 监控面板，删除重复项，方便排查问题
- 新增 TiKV details 监控面板，方便调试排查问题
- 新增滚动升级并发检测版本是否一致功能，提升滚动升级性能
- 新增 lightning 部署运维功能
- 优化 `table-regions.py` 脚本，新增按表显示 leader 分布功能
- 优化 TiDB 监控，新增以 SQL 类别显示延迟的监控项
- 修改操作系统版本限制，仅支持 CentOS 7.0 及以上，Red Hat 7.0 及以上版本的操作系统
- 新增预测集群最大 QPS 的监控项，默认隐藏

16.20.22 TiDB 3.0.0-rc.3 Release Notes

发版日期：2019 年 6 月 21 日

TiDB 版本：3.0.0-rc.3

TiDB Ansible 版本：3.0.0-rc.3

16.20.22.1 Overview

2019 年 6 月 21 日，TiDB 发布 3.0.0-rc.3 版本，对应的 TiDB Ansible 版本为 3.0.0-rc.3。相比 3.0.0-rc.2 版本，该版本对系统稳定性、易用性、功能、优化器、统计信息以及执行引擎做了很多改进。

16.20.22.2 TiDB

- SQL 优化器
  - 删除收集虚拟生成列的统计信息功能 #10629
  - 修复点查时主键常量溢出的问题 #10699
  - 修复 `fast analyze` 因使用未初始化的信息导致 panic #10691
  - 修复 prepare `create view` 语句执行过程中因列信息错误导致执行失败的问题 #10713
  - 修复在处理 window function 时列信息未拷贝的问题 #10720
  - 修复 index join 中内表过滤条件在某些情况下的选择率估计错误的问题 #10854
  - 新增变量 `stats-lease` 值为 0 时系统自动加载统计数据功能 #10811
- 执行引擎
  - 修复在 `StreamAggExec` 调用 `Close` 函数资源未正确释放问题 #10636

  - 修复对分区表执行 `show create table` 结果中 `table_option` 与 `partition_options` 顺序不正确问题 #10689
  - 通过支持逆序扫数据提升 `admin show ddl jobs` 的性能 #10687
  - 修复 RBAC 中对 `show grants` 语句带 `current_user` 字段时结果与 MySQL 不兼容的问题 #10684
  - 修复 UUID 在多节点上可能生成重复值的问题 #10712
  - 修复 `explain` 没考虑 `show view` 权限的问题 #10635
  - 新增 `split table region` 语句，手动分裂表的 Region，缓解热点问题 #10765
  - 新增 `split index region` 语句，手动分裂索引的 region 缓解热点问题 #10764
  - 修复连续执行多个 `create user`、`grant` 或 `revoke` 等类似语句执行不正确的问题 #10737
  - 新增黑名单禁止下推表达式到 coprocessor 功能 #10791
  - 新增查询超出内存配置限制时打印 `expensive query` 日志的功能 #10849
  - 新增 `bind-info-lease` 配置项控制修改绑定执行计划的更新时间 #10727
  - 修复因持有 `execdetails.ExecDetails` 指针时 Coprocessor 的资源无法快速释放导致的在大并发场景下 OOM 的问题 #10832
  - 修复某些情况下 `kill` 语句导致的 panic 问题 #10876

- Server

  - 修复 GC 时可能发生的 goroutine 泄露问题 #10683
  - 支持 slow query 里面显示 `host` 信息 #10693
  - 支持循环利用与 TiKV 交互的空闲链接 #10632
  - 修复 RBAC 对开启 `skip-grant-table` 选项的支持问题 #10738
  - 修复 `pessimistic-txn` 配置失效的问题 #10825
  - 修复主动取消的 ticlient 请求还会被重试的问题 #10850
  - 提高在悲观事务和乐观事务冲突情况下的性能 #10881

- DDL

  - 修复在使用 `alter table` 修改 charset 时导致 `blob` 类型改变的问题 #10698
  - 新增列属性包含 `AUTO_INCREMENT` 时利用 `SHARD_ROW_ID_BITS` 打散行 ID 功能，缓解热点问题 #10794
  - 禁止通过 `alter table` 添加存储的生成列 #10808
  - 优化无效 DDL 元信息存活时间，使集群升级后一段时间 DDL 操作比较慢的情况变短 #10795

16.20.22.3 PD

- 新增 `enable-two-way-merge` 配置项，控制合并时仅允许单向合并 #1583
- 新增 `AddLightLearner` 和 `AddLightPeer` 的调度操作，Region Scatter 调度不受 limit 机制限 #1563
- 修复系统启动时因数据可能只进行一副本复制而导致可靠性不足的问题 #1581
- 优化配置检查逻辑，防止配置项错误 #1585
- 调整 `store-balance-rate` 配置的定义为每分钟产生 balance operator 数量的上限 #1591
- 修复 store 可能一直无法产生调度操作的问题 #1590

16.20.22.4 TiKV

- Engine

  - 修复因迭代器未检查状态导致系统生成残缺 snapshot 的问题 #4936

- 修复在机器异常掉电时由于接收 snapshot 未及时将数据刷新到磁盘导致丢数据的问题 #4937

- Server
  - 新增检查 `block-size` 配置的有效性功能 #4928
  - 新增 read index 相关监控项 #4830
  - 新增 GC worker 相关监控项 #4922

- Raftstore
  - 修复 local reader 的 cache 没有正确清理的问题 #4778
  - 修复进行 transfer leader 和 conf change 时可能导致请求延迟增加的问题 #4734
  - 修复误报 stale command 的问题 #4682
  - 修复 command 可能一直 pending 的问题 #4810
  - 修复 snapshot 文件未及时落盘而导致掉电后文件损坏的问题 #4807，#4850

- Coprocessor
  - 向量计算支持 Top-N #4827
  - 向量计算支持 Stream 聚合 #4786
  - 向量计算中支持 AVG 聚合函 #4777
  - 向量计算中支持 First 聚合函数 #4771
  - 向量计算中支持 SUM 聚合函数 #4797
  - 向量计算中支持 MAX/MIN 聚合函数 #4837
  - 向量计算中支持 Like 表达式 #4747
  - 向量计算中支持 MultiplyDecimal 表达式 #4849
  - 向量计算中支持 BitAnd/BitOr/BitXor 表达式 #4724
  - 向量计算中支持 UnaryNot 表达式 #4808

- Transaction
  - 修复悲观事务中非悲观锁冲突导致出错的问题 #4801，#4883
  - 减少在开启悲观事务后乐观事务的无用计算，提高性能 #4813
  - 新增单语句 rollback，保证在当前语句发生死锁时不需要 rollback 整个事务 #4848
  - 新增悲观事务相关监控项 #4852
  - 支持 `ResolveLockLite` 命令用于轻量级清锁以优化在冲突严重时的性能 #4882

- tikv-ctl
  - 新增 `bad-regions` 命令支持检测更多的异常情况 #4862
  - `tombstone` 命令新增强制执行功能 #4862

- Misc
  - 新增 `dist_release` 编译命令 #4841

16.20.22.5 Tools

- TiDB Binlog
  - 修复 Pump 因写入失败时未检查返回值导致偏移量错误的问题 #640
  - Drainer 新增 `advertise-addr` 配置，支持容器环境中使用桥接模式 #634
  - Pump 新增 `GetMvccByEncodeKey` 函数，加快事务状态查询 #632

16.20.22.6 TiDB Ansible

- 新增预测集群最大 QPS 的监控项（默认隐藏）#f5cfa4d

16.20.23 TiDB 3.0.0-rc.2 Release Notes

发版日期：2019 年 5 月 28 日

TiDB 版本：3.0.0-rc.2

TiDB Ansible 版本：3.0.0-rc.2

16.20.23.1 Overview

2019 年 5 月 28 日，TiDB 发布 3.0.0-rc.2 版本，对应的 TiDB Ansible 版本为 3.0.0-rc.2。相比 3.0.0-rc.1 版本，该版本对系统稳定性、易用性、功能、优化器、统计信息以及执行引擎做了很多改进。

16.20.23.2 TiDB

- SQL 优化器
  - 在更多的场景中支持 Index Join #10540
  - 支持导出历史统计信息 #10291
  - 支持对单调递增的索引列增量 `Analyze` #10355
  - 忽略 `Order By` 子句中的 NULL 值 #10488
  - 修复精简列信息时逻辑算子 `UnionAll` 的 Schema 信息的计算不正确的问题 #10384
  - 下推 `Not` 操作符时避免修改原表达式 #10363
  - 支持导入导出列的关联性信息 #10573
- 执行引擎
  - 有唯一索引的虚拟生成列可以在 `replace on duplicate key update/insert on duplicate key update` 语句中被正确地处理 #10370
  - 修复 CHAR 列上的扫描范围计算 #10124
  - 修复 `PointGet` 处理负数不正确问题 #10113
  - 合并具有相同窗口名的窗口函数，提高执行效率 #9866
  - 窗口函数中 Range Frame 可以无需 `Order By` 子句 #10496
- Server
  - 修复 TiKV 故障时，TiDB 不断创建与 TiKV 的新连接的问题 #10301
  - `tidb_disable_txn_auto_retry` 不再只影响写入冲突错误，而是影响所有的可重试错误 #10339
  - 不带参数的 DDL 语句可以通过 `prepare/execute` 来执行 #10144
  - 新增 `tidb_back_off_weight` 变量，控制 TiDB 内部 back off 时间的长短 #10266
  - `tidb_disable_txn_auto_retry` 的默认值改为 `on`，即默认情况下，TiDB 不会重试非自动提交的事务 #10266
  - 修复 RBAC 中对 role 的数据库权限的判断不正确的问题 #10261
  - 支持悲观事务模式（实验特性）#10297
  - 降低某些情况下处理锁冲突时的等待时间 #10006

  - 重构 Region cache，增加在 Region 故障时的轮询逻辑 #10256
  - 新增 `tidb_low_resolution_tso` 变量，控制批量获取 `tso` 个数，减少事务获取 `tso` 的次数，以适应某些数据一致性要求较低的场景 #10428

- DDL

  - 修复旧版本的 TiDB 存储的字符集名称大写的问题 #10272
  - 支持 table partition 预分裂 Region 功能，该选项可以在建表时预先分配 table Region，避免建表后大量写入造成的写热点 #10221
  - 修复某些情况下 TiDB 更新版本信息到 PD 不准确的问题 #10324
  - 支持通过 `alter schema` 语句修改数据库 charset 和 collation #10393
  - 支持通过语句按指定表的索引及范围分裂 Region，用于缓解热点问题 #10203
  - 禁止 `alter table` 语句修改 `decimal` 列的精度 #10433
  - 修复 hash partition 中对表达式和函数的约束 #10273
  - 修复某些情况下对含有 partition 的 table 添加索引时引发 TiDB panic 的问题 #10475
  - 添加对某些极端情况下导致 schema 出错的防护功能 #10464
  - 创建 range partition 若有单列或者创建 hash partition 时默认开启分区功能 #9936

16.20.23.3 PD

- 默认开启 Region storage 将 Region 元信息存储到 Region storage 中 #1524
- 修复热点调度受其他调度器抢占的问题 #1522
- 修复 Leader 优先级不生效的问题 #1533
- 新增 `ScanRegions` 的 gRPC 接口 #1535
- 主动下发 operator 加快调度速度 #1536
- 添加 store limit 机制，限制单个 store 的调度速度 #1474
- 修复 `config` 状态不一致的问题 #1476

16.20.23.4 TiKV

- Engine

  - 支持多个 column family 共享 block cache #4563

- Server

  - 移除 txn scheduler #4098
  - 支持悲观锁事务 #4698

- Raftstore

  - 新增 hibernate Regions 特性，减少 raftstore CPU 的消耗 #4591
  - 移除 local reader 线程 #4558
  - 修复 Leader 不回复 Learner `ReadIndex` 请求的问题 #4653
  - 修复在某些情况下 transfer leader 失败的问题 #4684
  - 修复在某些情况下可能发生的脏读问题 #4688
  - 修复在某些情况下 snapshot 少包含数据的问题 #4716

- Coprocessor

  - 新增更多的 RPN 函数
    * LogicalOr #4691
    * LTReal #4602
    * LEReal #4602
    * GTReal #4602
    * GEReal #4602
    * NEReal #4602
    * EQReal #4602
    * IsNull #4720
    * IsTrue #4720
    * IsFalse #4720
    * 支持 Int 比较运算 #4625
    * 支持 Decimal 比较运算 #4625
    * 支持 String 比较运算 #4625
    * 支持 Time 比较运算 #4625
    * 支持 Duration 比较运算 #4625
    * 支持 Json 比较运算 #4625
    * 支持 Int 加法运算 #4733
    * 支持 Real 加法运算 #4733
    * 支持 Decimal 加法运算 #4733
    * 支持 Int 求余函数 #4727
    * 支持 Real 求余函数 #4727
    * 支持 Decimal 求余函数 #4727
    * 支持 Int 减法运算 #4746
    * 支持 Real 减法运算 #4746
    * 支持 Decimal 减法运算 #4746

16.20.23.5 Tools

- TiDB Binlog
  - Drainer 增加下游同步延迟监控项 checkpoint_delay #594
- TiDB Lightning
  - 支持数据库合并，数据表合并同步功能 #95
  - 新增 KV 写入失败重试机制 #176
  - 配置项 table-concurrency 默认值修改为 6 #175
  - 减少必要的配置项，tidb.port 和 tidb..pd-addr 支持自动获取 #173

16.20.24 TiDB 3.0.0-rc.1 Release Notes

发版日期：2019 年 5 月 10 日

TiDB 版本：3.0.0-rc.1

TiDB Ansible 版本：3.0.0-rc.1

16.20.24.1 Overview

2019 年 5 月 10 日，TiDB 发布 3.0.0-rc.1 版，对应的 TiDB Ansible 版本为 3.0.0-rc.1。相比 3.0.0-beta.1 版本，该版本对系统稳定性、易用性、功能、优化器、统计信息以及执行引擎做了很多改进。

16.20.24.2 TiDB

- SQL 优化器
  - 利用列之间的顺序相关性提升代价估算准确度，并提供启发式参数 `tidb_opt_correlation_exp_factor` 用于控制在相关性无法被直接用于估算的场景下对索引扫描的偏好程度。#9839
  - 当过滤条件中包含相关列时，在抽取复合索引的访问条件时尽可能多地匹配索引的前缀列。#10053
  - 用动态规划决定连接的执行顺序，当参与连接的表数量不多于 `tidb_opt_join_reorder_threshold` 时启用。#8816
  - 在构造 Index Join 的的内表中，以复合索引作为访问条件时，尽可能多地匹配索引的前缀列。#8471
  - 提升对单列索引上值为 NULL 的行数估算准确度。#9474
  - 在逻辑优化阶段消除聚合函数时特殊处理 GROUP_CONCAT ，防止产生错误的执行结果。#9967
  - 当过滤条件为常量时，正确地将它下推到连接算子的子节点上。#9848
  - 在逻辑优化阶段列剪裁时特殊处理一些函数，例如 RAND() ，防止产生和 MySQL 不兼容的执行结果。#10064
  - 支持 FAST ANALYZE，通过`tidb_enable_fast_analyze` 变量控制。该特性通过用对 Region 进行采样取代扫描整个 region 的方式加速统计信息收集。#10258
  - 支持 SQL PLAN MANAGEMENT。该特性通过对 SQL 进行执行计划绑定，以确保执行稳定性。该特性目前处于测试阶段，仅支持对 SELECT 语句使用绑定的执行计划，不建议在生产场景中直接使用。#10284
- 执行引擎
  - 支持对 TableReader、 IndexReader 和 IndexLookupReader 算子进行内存追踪控制。#10003
  - 在慢日志中展示更多 COPROCESSOR 端执行任务相关细节。如 COPROCESSOR 任务数，平均/最长/90% 执行/等待时间，执行/等待时间最长的 TiKV 地址等。#10165
  - 支持 PREPARE 不含占位符的 DDL 语句。#10144
- Server
  - TiDB 启动时，只允许 DDL owner 执行 bootstrap #10029
  - 新增 `tidb_skip_isolation_level_check` 变量控制检查隔离级别设置为 SERIALIZABLE 时不报错 #10065
  - 在慢日志中，将隐式提交的时间与 SQL 执行时间融合在一起 #10294
  - RBAC 权限管理
    * 支持 SHOW GRANT #10016
    * 支持 SET DEFAULT ROLE #9949
    * 支持 GRANT ROLE #9721
  - 修正了插件退出时导致 TiDB 退出的问题 #9889
  - 修正只读语句被错误地放到事务历史中的问题 #9723
  - kill 语句可以更快的结束 SQL 的执行，并快速释放资源 #9844
  - 增加启动选项 `config-check` 来检查配置文件的合法性 #9855
  - 修正非严格模式下对于写入 NULL 字段的合法性检查 #10161

- DDL
  - 为 CREATE TABLE 添加了 pre_split_regions 选项，该选项可以在建表时预先分配 Table Region，避免建表后大量写入造成的写热点 #10138
  - 优化了部分 DDL 语句的执行性能 #10170
  - FULLTEXT KEY 新增不支持全文索引的 warning #9821
  - 修正了旧版本 TiDB 中，UTF8 和 UTF8MB4 编码的兼容性问题 #9820
  - 修正了一个表的 shard_row_id_bits 的潜在 BUG #9868
  - 修正了 ALTER TABLE Charset 后，Column Charset 不会跟随变化的 BUG #9790
  - 修正了使用 BINARY/BIT 作为 Column Default Value 时，SHOW COLUMN 可能出错的 BUG #9897
  - 修正了 SHOW FULL COLUMNS 语句中，CHARSET / COLLATION 显示的兼容性问题 #10007
  - 现在 SHOW COLLATIONS 语句只会列出 TiDB 所实际支持的 COLLATIONS #10186

16.20.24.3 PD

- 升级 ETCD 版本 #1452
  - 统一 etcd 的日志格式与 pd server 一致
  - 修复 prevote 可能无法选出 Leader 的问题
  - 快速 drop 掉会失败的 propose 和 read 请求，减少阻塞后面的请求时间
  - 修复 Lease 的死锁问题
- 修复 store 读热点的 keys 统计不正确问题 #1487
- 支持从单一 PD 节点强制重建 PD 集群 #1485
- 修复 Scatter Region 产生无效 Operator Step 的问题 #1482
- 修复 Region Merge Operator 超时时间过短的问题 #1495
- 热点调度使用高优先级 #1492
- 添加 PD server 端处理 TSO 请求的耗时 Metrics #1502
- 添加相对应的 Store ID 和 Address 到 store 相关的 Metrics #1506
- 支持 GetOperator 服务 #1477
- 修复 Heartbeat stream 下发送 error 找不到 store 的问题 #1521

16.20.24.4 TiKV

- Engine
  - 修复读流量统计不准确问题 #4436
  - 修复 prefix extractor panic 的问题 #4503
  - 优化内存管理，减少 `Iterator Key Bound Option` 的内存分配和拷贝 #4537
  - 修复 Merge Region 时未考虑 Learner log gap 造成的 panic 问题 #4559
  - 支持不同的 `column families` 共享 `block cache` #4612
- Server
  - 减少 `batch commands` 的上下文切换开销 #4473
  - 检查 seek iterator status 的合法性 #4470
- RaftStore

  - 可配置化 `properties index distance` #4517

- Coprocessor
  - 新增 batch index scan executor #4419
  - 新增向量化 evaluation 框架 #4322
  - 新增 batch 执行器统计框架 #4433
  - 构建 RPN expression 时检查 max column 以防止 evaluation 阶段 column offset 越界的问题 #4481
  - 实现 `BatchLimitExecutor` #4469
  - ReadPool 使用 `tokio-threadpool` 替换原本的 `futures-cpupool`，减少 context switch #4486
  - 新增 batch 聚合框架 #4533
  - 新增 `BatchSelectionExecutor` #4562
  - 实现 batch aggression function AVG #4570
  - 实现 RPN function `LogicalAnd` #4575

- Misc
  - 支持选用 tcmalloc 为内存分配器 #4370

16.20.24.5 Tools

- TiDB Binlog
  - 修复 unsigned int 类型的主键列的 binlog 数据为负数，造成同步出错中断的问题 #573
  - 删除下游是 pb 时的压缩选项，修改下游名字 pb 成 file #559
  - Pump 新增 storage.sync-log 配置项，支持 Pump 本地存储异步刷盘 #509
  - Pump 和 Drainer 之间通讯支持流量压缩 #495
  - Drainer 新增 syncer.sql-mode 配置项，支持使用不同 sql-mode 解析 DDL query #511
  - Drainer 新增 syncer.ignore-table 配置项，支持过滤不需要同步的表 #520

- Lightning
  - 使用 row id 或者列的默认值填充 dump 文件中缺少的 column 数据 #170
  - Importer 修复部分 SST 导入失败依然返回导入成功的 bug #4566
  - Importer 支持 upload SST 到 TiKV 限速 #4412
  - Lightning 优化导入表的顺序，按照表的数据大小顺序进行导入，减少导入过程中大表执行 checksum 和 Analyze 对集群的影响，并且提高 Checksum 和 Analyze 的成功率 #156
  - 提升 Lightning encode SQL 性能，性能提升 50%，直接解析数据源文件内容成 TiDB 的 types.Datum，省去 KV encoder 的多余解析工作 #145
  - 日志格式改为 Unified Log Format #162
  - 新增一些命令行选项，即使缺少配置文件也能使用。#157

- 数据同步对比工具 (sync-diff-inspector)
  - 支持 checkpoint，记录校验状态，重启后从上次进度继续校验 #224
  - 增加配置项 only-use-checksum，只通过计算 checksum 来检查数据是否一致 #215

16.20.24.6 TiDB Ansible

- TiKV 监控变更以及更新 Ansible、 Grafana、 Prometheus 版本 #727

  - summary 监控适用于用户查看集群状态
  - trouble_shooting 监控适用于 DBA 排查问题
  - details 监控适用于开发分析问题

- 修复下载 Kafka 版本 Binlog 失败的 BUG #730
- 修改操作系统版本限制，仅支持 CentOS 7.0 及以上，Red Hat 7.0 及以上版本的操作系统 #733
- 滚动升级时的版本检测改为多并发 #736
- 更新 README 中文档链接#740
- 移除重复的 TiKV 监控项，新增 trouble shooting 监控项 #735
- 优化 `table-regions.py` 脚本，按表显示 leader 分布 #739
- 更新 drainer 配置文件 #745
- 优化 TiDB 监控，新增以 SQL 类别显示延迟的监控项 #747
- 更新 Lightning 配置文件，新增 tidb_lightning_ctl 脚本 #1e946f8

16.20.25 TiDB 3.0.0 Beta.1 Release Notes

发版日期：2019 年 3 月 26 日

TiDB 版本：3.0.0-beta.1

TiDB Ansible 版本：3.0.0-beta.1

16.20.25.1 Overview

2019 年 03 月 26 日，TiDB 发布 3.0.0 Beta.1 版，对应的 TiDB Ansible 版本为 3.0.0 Beta.1。相比 3.0.0 Beta 版本，该版本对系统稳定性、易用性、功能、优化器、统计信息以及执行引擎做了很多改进。

16.20.25.2 TiDB

- SQL 优化器
  - 支持使用 Sort Merge Join 计算笛卡尔积 #9032
  - 支持 Skyline Pruning，用一些规则来防止执行计划过于依赖统计信息 #9337
  - 支持 Window Functions
    * NTILE #9682
    * LEAD 和 LAG #9672
    * PERCENT_RANK #9671
    * NTH_VALUE #9596
    * CUME_DIST #9619
    * FIRST_VALUE 和 LAST_VALUE #9560
    * RANK 和 DENSE_RANK #9500
    * RANGE FRAMED #9450
    * ROW FRAMED #9358
    * ROW NUMBER #9098
  - 增加了一类统计信息，表示列和 handle 列之间顺序的相关性 #9315
- SQL 执行引擎

    - 增加内建函数
        * JSON_QUOTE #7832
        * JSON_ARRAY_APPEND #9609
        * JSON_MERGE_PRESERVE #8931
        * BENCHMARK #9252
        * COALESCE #9087
        * NAME_CONST #9261
    - 根据查询上下文优化 Chunk 大小，降低 SQL 执行时间和集群的资源消耗 #6489
- 权限管理
    - 支持 SET ROLE 和 CURRENT_ROLE #9581
    - 支持 DROP ROLE #9616
    - 支持 CREATE ROLE #9461
- Server
    - 新增 /debug/zip HTTP 接口，获取当前 TiDB 实例的信息 #9651
    - 支持使用 show pump status/show drainer status 语句查看 Pump/Drainer 状态 #9456
    - 支持使用 SQL 语句在线修改 Pump/Drainer 状态 #9789
    - 支持给 SQL 文本加上 HASH 指纹，方便追查慢 SQL #9662
    - 新增 log_bin 系统变量，默认：0，管理 binlog 开启状态，当前仅支持查看状态 #9343
    - 支持通过配置文件管理发送 binlog 策略 #9864
    - 支持通过内存表 INFORMATION_SCHEMA.SLOW_QUERY 查询慢日志 #9290
    - 将 TiDB 显示的 MySQL Version 从 5.7.10 变更为 5.7.25 #9553
    - 统一日志格式规范，利于工具收集分析
    - 增加监控项 high_error_rate_feedback_total，记录实际数据量与统计信息估算数据量差距情况 #9209
    - 新增 Database 维度的 QPS 监控项, 可以通过配置项开启 #9151
- DDL
    - 增加ddl_error_count_limit全局变量，默认值：512，限制 DDL 任务重试次数，超过限制次数会取消出错的 DDL #9295
    - 支持 ALTER ALGORITHM INPLACE/INSTANT #8811
    - 支持 SHOW CREATE VIEW 语句 #9309
    - 支持 SHOW CREATE USER 语句 #9240

16.20.25.3 PD

- 统一日志格式规范，利于工具收集分析
- 模拟器
    - 支持不同 store 可采用不同的心跳间隔时间 #1418
    - 添加导入数据的场景 #1263
- 热点调度可配置化 #1412
- 增加 store 地址为维度的监控项，代替原有的 Store ID #1429
- 优化 GetStores 开销，加快 Region 巡检周期 #1410
- 新增删除 Tombstone Store 的接口 #1472

16.20.25.4 TiKV

- 优化 Coprocessor 计算执行框架，完成 TableScan 算子，单 TableScan 即扫表操作性能提升 5% ~ 30%
  - 实现行 `BatchRows` 和列 `BatchColumn` 的定义 #3660
  - 实现 `VectorLike` 使得编码和解码的数据能够用统一的方式访问 #4242
  - 定义 `BatchExecutor` 接口，实现将请求转化为 `BatchExecutor` 的方法 #4243
  - 实现将表达式树转化成 RPN 格式 #4329
  - TableScan 算子实现为 Batch 方式，通过向量化计算加速计算 #4351
- 统一日志格式规范，利于工具收集分析
- 支持 Raw Read 接口使用 Local Reader 进行读 #4222
- 新增配置信息的 Metrics #4206
- 新增 Key 越界的 Metrics #4255
- 新增碰到扫越界错误时 Panic 或者报错选项 #4254
- 增加 Insert 语义，只有在 Key 不存在的时候 Prewrite 才成功，消除 Batch Get #4085
- Batch System 使用更加公平的 batch 策略 #4200
- tikv-ctl 支持 Raw scan #3825

16.20.25.5 Tools

- TiDB Binlog
  - 新增 Arbiter 工具支持从 Kafka 读取 binlog 同步到 MySQL
  - Reparo 支持过滤不需要同步的文件
  - 支持同步 generated column
- Lightning
  - 支持禁用 TiKV periodic Level-1 compaction，当 TiKV 集群为 2.1.4 或更高时，在导入模式下会自动执行 Level-1 compaction #119，#4199
  - 根据 `table_concurrency` 配置项限制 import engines 数量，默认值：16，防止过多占用 importer 磁盘空间 #119
  - 支持保存中间状态的 SST 到磁盘，减少内存使用 #4369
  - 优化 TiKV-Importer 导入性能，支持将大表的数据和索引分离导入 #132
  - 支持 CSV 文件导入 #111
- 数据同步对比工具 (sync-diff-inspector)
  - 支持使用 TiDB 统计信息来划分对比的 chunk #197
  - 支持使用多个 column 来划分对比的 chunk #197

16.20.26 TiDB 3.0 Beta Release Notes

2019 年 1 月 19 日，TiDB 发布 3.0 Beta 版，TiDB Ansible 相应发布 3.0 Beta 版本。相比 2.1 版本，该版本对系统稳定性、优化器、统计信息以及执行引擎做了很多改进。

16.20.26.1 TiDB

- 新特性
  - 支持 View
  - 支持窗口函数
  - 支持 Range 分区
  - 支持 Hash 分区
- SQL 优化器
  - 重新支持聚合消除的优化规则 #7676
  - 优化 NOT EXISTS 子查询，将其转化为 Anti Semi Join #7842
  - 添加 tidb_enable_cascades_planner 变量以支持新的 Cascades 优化器。目前 Cascades 优化器尚未实现完全，默认关闭 #7879
  - 支持在事务中使用 Index Join #7877
  - 优化 Outer Join 上的常量传播，使得对 Join 结果里和 Outer 表相关的过滤条件能够下推过 Outer Join 到 Outer 表上，减少 Outer Join 的无用计算量，提升执行性能 #7794
  - 调整投影消除的优化规则到聚合消除之后，消除掉冗余的 Project 算子 #7909
  - 优化 IFNULL 函数，当输入参数具有非 NULL 的属性的时候，消除该函数 #7924
  - 支持对 _tidb_rowid 构造查询的 Range，避免全表扫，减轻集群压力 #8047
  - 优化 IN 子查询为先聚合后做 Inner Join 并，添加变量 tidb_opt_insubq_to_join_and_agg 以控制是否开启该优化规则并默认打开 #7531
  - 支持在 DO 语句中使用子查询 #8343
  - 添加 Outer Join 消除的优化规则，减少不必要的扫表和 Join 操作，提升执行性能 #8021
  - 修改 TIDB_INLJ 优化器 Hint 的行为，优化器将使用 Hint 中指定的表当做 Index Join 的 Inner 表 #8243
  - 更大范围的启用 PointGet，使得当 Prepare 语句的执行计划缓存生效时也能利用上它 #8108
  - 引入贪心的 Join Reorder 算法，优化多表 Join 时 Join 顺序选择的问题 #8394
  - 支持 View #8757
  - 支持 Window Function #8630
  - 当 TIDB_INLJ 未生效时，返回 warning 给客户端，增强易用性 #9037
  - 支持根据过滤条件和表的统计信息推导过滤后数据的统计信息的功能 #7921
  - 增强 Range Partition 的 Partition Pruning 优化规则 #8885
- SQL 执行引擎
  - 优化 Merge Join 算子，使其支持空的 ON 条件 #9037
  - 优化日志，打印执行 EXECUTE 语句时使用的用户变量 #7684
  - 优化日志，为 COMMIT 语句打印慢查询信息 #7951
  - 支持 EXPLAIN ANALYZE 功能，使得 SQL 调优过程更加简单 #7827
  - 优化列很多的宽表的写入性能 #7935
  - 支持 admin show next_row_id #8242
  - 添加变量 tidb_init_chunk_size 以控制执行引擎使用的初始 Chunk 大小 #8480
  - 完善 shard_row_id_bits，对自增 ID 做越界检查 #8936
- Prepare 语句
  - 对包含子查询的 Prepare 语句，禁止其添加到 Prepare 语句的执行计划缓存中，确保输入不同的用户变量时执行计划的正确性 #8064

  - 优化 Prepare 语句的执行计划缓存，使得当语句中包含非确定性函数的时候，该语句的执行计划也能被缓存 #8105
  - 优化 Prepare 语句的执行计划缓存，使得 DELETE/UPDATE/INSERT 的执行计划也能被缓存 #8107
  - 优化 Prepare 语句的执行计划缓存，当执行 DEALLOCATE 语句时从缓存中剔除对应的执行计划 #8332
  - 优化 Prepare 语句的执行计划缓存，通过控制其内存使用以避免缓存过多执行计划导致 TiDB OOM 的问题 #8339
  - 优化 Prepare 语句，使得 ORDER BY/GROUP BY/LIMIT 子句中可以使用 “?” 占位符 #8206
- 权限管理
  - 增加对 ANALYZE 语句的权限检查 #8486
  - 增加对 USE 语句的权限检查 #8414
  - 增加对 SET GLOBAL 语句的权限检查 #8837
  - 增加对 SHOW PROCESSLIST 语句的权限检查 #7858
- Server
  - 支持了对 SQL 语句的 Trace 功能 #9029
  - 支持了插件框架 #8788
  - 支持同时使用 unix_socket 和 TCP 两种方式连接数据库 #8836
  - 支持了系统变量 interactive_timeout #8573
  - 支持了系统变量 wait_timeout #8346
  - 提供了变量 tidb_batch_commit，可以按语句数将事务分解为多个事务 #8293
  - 支持 ADMIN SHOW SLOW 语句，方便查看慢日志 #7785
- 兼容性
  - 支持了 ALLOW_INVALID_DATES 这种 SQL mode #9027
  - 提升了 load data 对 CSV 文件的容错能力 #9005
  - 支持了 MySQL 320 握手协议 #8812
  - 支持将 unsigned bigint 列声明为自增列 #8181
  - 支持 SHOW CREATE DATABASE IF NOT EXISTS 语法 #8926
  - 当过滤条件中包含用户变量时不对其进行谓词下推的操作，更加兼容 MySQL 中使用用户变量模拟 Window Function 的行为 #8412
- DDL
  - 支持快速恢复误删除的表 #7937
  - 支持动态调整 ADD INDEX 的并发数 #8295
  - 支持更改表或者列的字符集到 utf8/utf8mb4 #8037
  - 默认字符集从 utf8 变为 utf8mb4 #7965
  - 支持 RANGE PARTITION #8011

16.20.26.2 Tools

- TiDB Lightning
  - 大幅优化 SQL 转 KV 的处理速度 #110
  - 对单表支持 batch 导入，提高导入性能和稳定性 #113

16.20.26.3 PD

- 增加 RegionStorage 单独存储 Region 元信息 #1237
- 增加 shuffle hot region 调度 #1361
- 增加调度参数相关 Metrics #1406
- 增加集群 Label 信息相关 Metrics #1402
- 增加导入数据场景模拟 #1263
- 修复 Leader 选举相关的 Watch 问题 #1396

16.20.26.4 TiKV

- 支持了分布式 GC #3179
- 在 Apply snapshot 之前检查 RocksDB level 0 文件，避免产生 Write stall #3606
- 支持了逆向 raw_scan 和 raw_batch_scan #3742
- 更好的夏令时支持 #3786
- 支持了使用 HTTP 方式获取监控信息 #3855
- 支持批量方式接收和发送 Raft 消息 #3931
- 引入了新的存储引擎 Titan #3985
- 升级 gRPC 到 v1.17.2 #4023
- 支持批量方式接收客户端请求和发送回复 #4043
- 多线程 Apply #4044
- 多线程 Raftstore #4066

16.21 v2.1

16.21.1 TiDB 2.1.19 Release Notes

发版日期：2019 年 12 月 27 日

TiDB 版本：2.1.19

TiDB Ansible 版本：2.1.19

16.21.1.1 TiDB

- SQL 优化器
  - 优化 select max(_tidb_rowid)from t 的场景，避免全表扫 #13294
  - 修复当查询语句中赋予用户变量错误的值且将谓词下推后导致错误的输出结果 #13230
  - 修复更新统计信息时可能存在数据竞争，导致统计信息不准确的问题 #13690
  - 修复 UPDATE 语句中同时包含子查询和 stored generated column 时结果错误的问题；修复 UPDATE 语句中包含不同数据库的两个表名相同时，UPDATE 执行报错的问题 #13357
  - 修复 PhysicalUnionScan 算子没有正确设置统计信息，导致查询计划可能选错的问题 #14134
  - 移除 minAutoAnalyzeRatio 约束使自动 ANALYZE 更及时 #14013
  - 当 WHERE 子句上有 UNIQUE KEY 的等值条件时，估算行数应该不大于 1 #13385

- SQL 执行引擎
  - 修复 ConvertJSONToInt 中使用 int64 作为 uint64 的中间解析结果，导致精度溢出的问题 #13036
  - 修复查询中包含 SLEEP 函数时（例如 select 1 from (select sleep(1))t;)），由于列裁剪导致查询中的 sleep(1) 失效的问题 #13039
  - 通过实现在 INSERT ON DUPLICATE UPDATE 语句中复用 Chunk 来降低内存开销 #12999
  - 给 slow_query 表添加事务相关的信息段 #13129，如下：
    * Prewrite_time
    * Commit_time
    * Get_commit_ts_time
    * Commit_backoff_time
    * Backoff_types
    * Resolve_lock_time
    * Local_latch_wait_time
    * Write_key
    * Write_size
    * Prewrite_region
    * Txn_retry
  - 修复 UPDATE 语句中包含子查询时转换子查询出现的错误和当 UPDATE 的 WHERE 条件中包含子查询时更新失败的问题 #13120
  - 支持在分区表上执行 ADMIN CHECK TABLE #13143
  - 修复 ON UPDATE CURRENT_TIMESTAMP 作为列的属性且指定浮点精度时，SHOW CREATE TABLE 等语句显示精度不完整的问题 #12462
  - 修复在 DROP/MODIFY/CHANGE COLUMN 时没有检查外键导致执行 SELECT * FROM information_schema
    ↪ .KEY_COLUMN_USAGE 语句时发生 panic 的问题 #14162
  - 修复 TiDB 开启 Streaming 后返回数据可能重复的问题 #13255
  - 修复夏令时导致的“无效时间格式”问题 #13624
  - 修复整型数据被转换为无符号 Real/Decimal 类型时，精度可能丢失的问题 #13756
  - 修复 Quote 函数处理 null 值时返回值类型出错的问题 #13681
  - 修复从字符串解析日期时，由于使用 golang time.Local 本地时区导致解析结果的时区不正确的问题 #13792
  - 修复 builtinIntervalRealSig 的实现中，由于 binSearch 方法不会返回 error，导致最终结果可能不正确的问题 #13768
  - 修复 INSERT 语句在进行字符串类型到浮点类型转换时，可能会报错的问题 #14009
  - 修复 sum(distinct) 函数输出结果不正确的问题 #13041
  - 修复由于对 jsonUnquoteFunction 函数的返回类型长度赋值不正确的值，导致在 union 中同位置数据上进行 cast 转换时会截断数据的问题 #13645
  - 修复由于权限检查过于严格导致设置密码失败的问题 #13805
- Server
  - 修复 KILL CONNECTION 可能出现 goroutine 泄漏的问题 #13252
  - 新增通过 HTTP API 的 info/all 接口获取所有 TiDB 节点的 binlog 状态功能 #13188
  - 修复在 Windows 上 build TiDB 项目失败的问题 #13650
  - 新增 server-version 配置项来控制修改 TiDB server 版本的功能 #13904
  - 修复通过 Go1.13 版本编译的二进制程序 plugin 不能正常运行的问题 #13527
- DDL

- 新增创建表时如果表包含 COLLATE 则列的 COLLATE 使用表的 COLLATE #13190
- 新增创建表时限制索引名字的长度的功能 #13311
- 修复 rename table 时未检查表名长度的问题 #13345
- 新增 BIT 列的宽度范围检查的功能 #13511
- 优化 change/modify column 的输出的错误信息，让人更容易理解 #13798
- 修复执行 drop column 操作且下游 Drainer 还没有执行此 drop column 操作时，下游可能会收到不带此列的 DML 的问题 #13974

16.21.1.2 TiKV

- Raftstore
  - 修复 Region merge 和应用 Compact log 过程中系统若有重启，当重启时由于未正确设置 is_merging 的值导致系统 panic 的问题 #5884
- Importer
  - 取消 gRPC 的消息长度限制 #5809

16.21.1.3 PD

- 提升获取 Region 列表的 HTTP API 性能 #1988
- 升级 etcd，修复 etcd PreVote 无法选出 leader 的问题（升级后无法降级）#2052

16.21.1.4 Tools

- TiDB Binlog
  - 优化通过 binlogctl 输出的节点状态信息 #777
  - 修复当 Drainer 过滤配置为 nil 时 panic 的问题 #802
  - 优化 Pump 的 Graceful 退出方式 #825
  - 新增 Pump 写 binlog 数据时更详细的监控指标 #830
  - 优化 Drainer 在执行 DDL 后刷新表结构信息的逻辑 #836
  - 修复 Pump 在没有收到 DDL 的 commit binlog 时该 binlog 被忽略的问题 #855

16.21.1.5 TiDB Ansible

- TiDB 服务 Uncommon Error OPM 监控项更名为 Write Binlog Error 并增加对应的告警 #1038
- 升级 TiSpark 版本为 2.1.8 #1063

16.21.2 TiDB 2.1.18 Release Notes

发版日期：2019 年 11 月 4 日

TiDB 版本：2.1.18

TiDB Ansible 版本：2.1.18

16.21.2.1 TiDB

- SQL 优化器
  - 修复 Feedback 切分查询范围出错的问题 #12172
  - 修复点查中权限检查不正确的问题 #12341
  - 将 Limit 算子下推到 IndexLookUpReader 执行逻辑中，优化 `select ... limit ... offset ...` 的执行性能 #12380
  - 支持在 `ORDER BY`、`GROUP BY` 和 `LIMIT OFFSET` 中使用参数 #12514
  - 修复 partition 表上的 IndexJoin 返回错误结果的问题 #12713
  - 修复 TiDB 中 `str_to_date` 函数在日期字符串和格式化字符串不匹配的情况下，返回结果与 MySQL 不一致的问题 #12757
  - 修复当查询条件中包含 cast 函数时 outer join 被错误转化为 inner join 的问题 #12791
  - 修复 AntiSemiJoin 的 join 条件中错误的表达式传递 #12800
- SQL 执行引擎
  - 修复时间取整不正确的问题，(如 2019-09-11 11:17:47.999999666 应该被取整到 2019-09-11 11:17:48) #12259
  - 修复 `PREPARE` 语句类型没有记录在监控中的问题 #12329
  - 修复 `FROM_UNIXTIME` 在检查 NULL 值时 panic 的错误 #12572
  - 修复 `YEAR` 类型数据插入非法年份时，结果为 `NULL` 而不是 `0000` 的兼容性问题 #12744
  - 改进 AutoIncrement 列隐式分配时的行为，与 MySQL 自增锁的默认模式 (“consecutive” lock mode) 保持一致：对于单行 Insert 语句的多个自增 AutoIncrement ID 的隐式分配，TiDB 保证分配值的连续性。该改进保证 JDBC `getGeneratedKeys()` 方法在任意场景下都能得到正确的结果。#12619
  - 修复当 HashAgg 作为 Apply 子节点时查询 hang 住的问题 #12769
  - 修复逻辑表达式 AND / OR 在涉及类型转换时返回错误结果的问题 #12813
- Server
  - 修复 `KILL TIDB QUERY` 语法对 `SLEEP()` 语句无效的问题 #12159
  - 修复 AUTO INCREMENT 分配 MAX int64 和 MAX uint64 没有报错的问题 #12210
  - 修复日志级别设置为 `ERROR` 时，慢日志不会被记录的问题 #12373
  - 将缓存 100 个 Schema 变更相关的表信息调整成 1024 个，且支持通过 `tidb_max_delta_schema_count` 系统变量修改 #12515
  - 将 SQL 的统计方式开始时间由“开始执行”改为“开始编译”，使得 SQL 性能统计更加准确 #12638
  - 在 TiDB 日志中添加 `set session autocommit` 的记录 #12568
  - 将 SQL 的开始时间记录在 `SessionVars` 中，避免计划执行时，该时间被重置 #12676
  - 在 `Order By/Group By/Limit Offset` 字句中支持 `?` 占位符 #12514
  - 慢日志中添加 `Prev_stmt` 字段，用于最后一条语句是 `COMMIT` 时输出前一条语句 #12724
  - 当一个显式提交的事务 `COMMIT` 时出错，在日志中记录 `COMMIT` 前一条语句 #12747
  - 优化在 TiDB Server 执行 SQL 时，对前一条语句的保存方式以提升性能 #12751
  - 修复 `skip-grant-table=true` 时，`FLUSH PRIVILEGES` 语句导致系统 Panic 的问题 #12816
  - 将 AutoID 的最小申请步长从 1000 增加为 30000，避免短时间大量写入时频繁请求 AutoID 造成性能瓶颈 #12891
  - 修复 Prepared 语句在 TiDB 发生 panic 时错误日志中未打印出错 SQL 的问题 #12954
  - 修复 COM_STMT_FETCH 慢日志时间记录和 MySQL 不一致问题 #12953
  - 当遇到写冲突时，在报错信息中添加错误码，以方便对冲突原因进行诊断 #12878

- DDL
  - 为避免误操作，TiDB 默认不再允许删除列的 AUTO_INCREMENT 属性，当确实需要删除时，请更改系统变量 tidb_allow_remove_auto_inc；相关文档请见系统变量和语法 #12146
  - 支持 Create Table 语句中建唯一索引时带多个 Unique #12469
  - 修复 CreateTable 语句中指定外键约束时，外键表在没有指定 Database 时未能使用主表的 Database 导致报错的问题 #12678
  - 修复 ADMIN CANCEL DDL JOBS 时报 invalid list index 错的问题 #12681
- Monitor
  - Backoff 监控添加类型，且补充之前没有统计到的 Backoff，比如 commit 时遇到的 Backoff #12326
  - 添加统计 Add Index 操作进度的监控 #12389

16.21.2.2 PD

- 修复 pd-ctl --help 命令输出内容 #1772

16.21.2.3 Tools

- TiDB Binlog
  - 修复 ALTER DATABASE 相关 DDL 会导致 Drainer 异常退出的问题 #770
  - 支持对 Commit binlog 查询事务状态信息，提升同步效率 #761
  - 修复当 Drainer 的 start_ts 大于 Pump 中最大的 commit_ts 时候有可能引起 Pump panic 的问题 #759

16.21.2.4 TiDB Ansible

- TiDB Binlog 增加 queue size 和 query histogram 监控项 #952
- 更新 TiDB 告警表达式 #961
- 新增配置文件检查功能，部署或者更新前会检查配置是否合理 #973
- TiDB 新增加索引速度监控项 #987
- 更新 TiDB Binlog 监控 Dashboard，兼容 4.6.3 版本的 Grafana #993

16.21.3 TiDB 2.1.17 Release Notes

发版日期：2019 年 9 月 11 日

TiDB 版本：2.1.17

TiDB Ansible 版本：2.1.17

- 新特性
  - TiDB 的 SHOW TABLE REGIONS 语法新增 WHERE 条件子句
  - TiKV、PD 新增 config-check 功能，用于配置项检查
  - pd-ctl 新增 remove-tombstone 命令，支持清理 tombstone store 记录
  - Reparo 新增 worker-count 和 txn-batch 配置项，用于控制恢复速率

- 改进提升
  - PD 优化调度流程，支持主动下发调度
  - TiKV 优化启动流程，减少重启节点带来的抖动
- 行为变更
  - TiDB 慢日志中的 `start ts` 由最后一次重试的时间改为第一次执行的时间
  - TiDB 慢日志中的 `Index_ids` 字段替换为 `Index_names` 字段，提升慢日志易用性
  - TiDB 配置文件中添加 `split-region-max-num` 参数，用于调整 `SPLIT TABLE` 语法允许的最大 Region 数量，默认配置下，允许的数量由 1,000 增加至 10,000

16.21.3.1 TiDB

- SQL 优化器
  - 修复 `EvalSubquery` 在构建 `Executor` 出现错误时，错误信息没有被正确返回的问题 #11811
  - 修复 Index Lookup Join 中，外表的行数大于一个 batch 时，查询的结果可能不正确的问题；扩大 Index Lookup Join 的作用范围：可以使用 `UnionScan` 作为 `IndexJoin` 的子节点 #11843
  - 针对统计信息的反馈过程可能产生失效 Key 的情况，`SHOW STAT_BUCKETS` 语句现在增加了失效 Key 的显示，例如：`invalid encoded key flag 252` #12098
- SQL 执行引擎
  - 修复 `CAST` 函数在进行数值类型转换时，首先将数值转换为 `UINT` 导致一些结果不正确的问题（例如，`select cast(13835058000000000000 as double)`）#11712
  - 修复 `DIV` 运算的被除数为 `DECIMAL` 类型且运算含有负数时，运算结果可能不正确的问题 #11812
  - 添加 `ConvertStrToIntStrict` 函数，修复执行 `SELECT/EXPLAIN` 语句时，一些字符串转换 `INT` 类型结果与 MySQL 不兼容的问题 #11892
  - 修复使用 `EXPLAIN ... FOR CONNECTION` 语法时，`stmtCtx` 没有正确设置导致 `Explain` 结果可能不正确的问题 #11978
  - 修复 `unaryMinus` 函数，当 Int 结果溢出时，返回结果类型没有为 Decimal 导致与 MySQL 不兼容的问题 #11990
  - 修复 `LOAD DATA` 语句执行时，计数顺序导致的 `last_insert_id()` 可能不正确的问题 #11994
  - 修复用户显式、隐式混合写入自增列数据时，`last_insert_id()` 可能不正确的问题 #12001
  - 修复一个 `JSON_UNQUOTE` 函数兼容性问题：只有在双引号 " 内的值需要 Unquote，例如 `SELECT JSON_UNQUOTE("\\\\")` 应当为 “\\”（不进行 Unquote）#12096
- Server
  - TiDB 事务重试时，记录在慢日志中的 `start ts` 由最后一次重试的时间改为第一次执行的时间 #11878
  - 在 `LockResolver` 中添加事务的 Key 数量：当 Key 数量较少时，可以避免对整个 Region 的 Scan 操作，减小清锁的代价 #11889
  - 修复慢日志中，`succ` 字段值可能不正确的问题 #11886
  - 将慢日志中的 `Index_ids` 字段替换为 `Index_names` 字段，提升慢日志易用性 #12063
  - 修复 `Duration` 内容中包含 - 时（例如 `select time('--')`），TiDB 解析为 EOF Error 导致连接断开的错误 #11910
  - 改进 `RegionCache`：当一个 Region 失效时，它将会更快地从 `RegionCache` 中移除，减少向该 Region 发送请求的个数 #11931

  - 修复 oom-action = "cancel" 时，当 Insert Into ⋯ Select 语句发生 OOM，OOM Panic 没有被正确处理而导致连接断开的问题 #12126

- DDL
  - 为 tikvSnapshot 添加逆序扫描接口用于高效地查询 DDL History Job，使用该接口后 ADMIN SHOW DDL JOBS 的执行时间有明显降低 #11789
  - 改进 CREATE TABLE ... PRE_SPLIT_REGION 的语义：当指定 PRE_SPLIT_REGION = N 时，将预切分的 Region 个数由 2^(N-1) 改为 2^N #11797
  - 根据线上负载与 Add Index 相互影响测试，调小 Add Index 后台工作线程的默认参数以避免对线上负载造成较大影响 #11875
  - 改进 SPLIT TABLE 语法的行为：当使用 SPLIT TABLE ... REGIONS N 对 Region 切分时，会生成 N 个数据 Region 和 1 个索引 Region #11929
  - 在配置文件中添加 split-region-max-num 参数，使得 SPLIT TABLE 语法允许的最大 Region 数量可调整，该参数默认值 10000 #12080
  - 修复写 binlog 时，CREATE TABLE 语句中 PRE_SPLIT_REGIONS 部分没有被注释，导致语句不能被下游 MySQL 解析的问题 #12121
  - SHOW TABLE ⋯ REGIONS 和 SHOW TABLE .. INDEX ⋯ REGIONS 语法新增 WHERE 条件子句 #12124

- Monitor
  - 增加监控指标 connection_transient_failure_count，用于统计 tikvclient 的 gRPC 连接错误数量 #12092

16.21.3.2 TiKV

- 解决某些情况下 Region 内 key 个数统计不准的问题 #5415
- TiKV 新增 config-check 选项，用于检查 TiKV 配置项是否合法 #5391
- 优化启动流程，减少重启节点带来的抖动 #5277
- 优化某些情况下解锁的流程，加速事务解锁 #5339
- 优化 get_txn_commit_info 的流程，加速事务提交 #5062
- 简化 Raft 相关的 log #5425
- 解决在某些情况下 TiKV 异常退出的问题 #5441

16.21.3.3 PD

- PD 新增 config-check 选项，用于检查 PD 配置项是否合法 #1725
- pd-ctl 新增 remove-tombstone 命令，支持清理 tombstone store 记录 #1705
- 支持主动下发 Operator，加快调度速度 #1686

16.21.3.4 Tools

- TiDB Binlog
  - Reparo 新增 worker-count 和 txn-batch 配置项，用于控制恢复速率 #746
  - Drainer 优化内存使用，提升同步执行效率 #735
  - Pump 修复有时候无法正常下线的 bug #739

  - Pump 优化 LevelDB 处理逻辑，提升 GC 执行效率 #720

- TiDB Lightning
  - 修复从 checkpoint 点重新导入可能会导致 tidb-lightning 崩溃的 bug #239

16.21.3.5 TiDB Ansible

- 更新 Spark 版本为 2.4.3，同时更新 TiSpark 为兼容该 Spark 的 2.2.0 版本 #914，#919
- 修复了当远程机器密码过期时长时间连接等待的问题 #937，#948

16.21.4 TiDB 2.1.16 Release Notes

发版日期：2019 年 8 月 15 日

TiDB 版本：2.1.16

TiDB Ansible 版本：2.1.16

16.21.4.1 TiDB

- SQL 优化器
  - 修复时间列上的等值条件 Row Count 估算不准确的问题 #11526
  - 修复 `TIDB_INLJ` Hint 不生效或者对非指定的表生效的问题 #11361
  - 将查询中的 NOT EXISTS 由 OUTER JOIN 实现方式改为 ANTI JOIN，便于找到更优执行计划 #11291
  - 支持在 SHOW 语句中使用子查询，现在可以支持诸如 `SHOW COLUMNS FROM tbl WHERE FIELDS IN (SELECT 'a')` 的写法 #11461
  - 修复常量折叠优化导致 `SELECT ··· CASE WHEN ··· ELSE NULL ...` 查询结果不正确的问题 #11441
- SQL 执行引擎
  - 修复函数 DATE_ADD 在 INTERVAL 为负的情况下结果错误的问题 #11616
  - 修复 DATE_ADD 函数接受 FLOAT、DOUBLE 和 DECIMAL 类型的参数时，没有正确地进行类型转换而导致结果可能不正确的问题 #11628
  - 修复 CAST(JSON AS SIGNED) 出现 OVERFLOW 时错误信息不准确的问题 #11562
  - 修复在关闭 Executor 的过程中，子节点关闭返回错误时其他子节点未关闭的问题 #11598
  - 支持 SPLIT TABLE 语句返回切分成功的 REGION 数量，并且当部分 REGION SCATTER 在超时未完成调度时，不再返回错误，而是返回完成调度的比例 #11487
  - 修复 `REGEXP BINARY` 函数对大小写敏感，与 MySQL 不兼容的问题 #11505
  - 修复 DATE_ADD / DATE_SUB 结果中 YEAR 小于 0 或大于 65535 时溢出导致结果没有正确返回 NULL 值的问题 #11477
  - 慢查询表中添加用于表示是否执行成功的 Succ 字段 #11412
  - 修复一条 SQL 语句在涉及当前时间计算时（例如 CURRENT_TIMESTAMP 或者 NOW），多次取当前时间值，结果与 MySQL 不兼容的问题：现在同一条 SQL 语句中取当前时间时，均使用相同值 #11392
  - 修复 AUTO INCREMENT 列未处理 FLOAT / DOUBLE 的问题 #11389
  - 修复 CONVERT_TZ 函数在参数不合法时，没有正确返回 NULL 的问题 #11357
  - 修复 PARTITION BY LIST 报错的问题（仅添加语法支持，TiDB 执行时候会作为普通表创建并提供提示信息）#11236

  - 修复 `Mod(%)`、`Multiple(*)` 和 `Minus(-)` 返回结果为 0 时，在小数位数较多（例如 `select 0.000 % 0.11234500000000000000`）的情况下与 MySQL 位数不一致的问题 #11353

- Server
  - 修复插件在 OnInit 回调中获取 Domain 为 NULL 的问题 #11426
  - 修复当 Schema 删除后，依然可以通过 HTTP 接口获取该 Schema 中表信息的问题 #11586

- DDL
  - 禁止 DROP 自增列索引，修复因为 DROP 自增列上的索引导致自增列结果可能出错的问题 #11402
  - 修复列和表使用不同的 CHARSET 和 COLLATE 创建表和修改表时，列的字符集不正确的问题 #11423
  - 修复并行执行 “alter table ··· set default···” 和其他修改此列信息的 DDL，可能导致此列的结构出错的问题 #11374
  - 修复当 Generated column A 依赖 Generated column B 时，使用 A 创建索引，数据回填失败的问题 #11538
  - 提升 ADMIN CHECK TABLE 的速度 #11538

16.21.4.2 TiKV

- 访问正在关闭的 TiKV Region 时返回 Close 错误 #4820
- 支持逆向 `raw_scan` 和逆向 `raw_batch_scan` 接口 #5148

16.21.4.3 Tools

- TiDB Binlog
  - Drainer 增加 `ignore-txn-commit-ts` 配置项，用于跳过执行某些事务语句 #697
  - 增加启动时配置项检查功能，遇到不合法配置项会退出运行并给出错误信息 #708
  - Drainer 增加 `node-id` 配置，用于指定固定逻辑 Drainer #706

- TiDB Lightning
  - 修复 2 个 checksum 同时运行的情况下，`tikv_gc_life_time` 没有正常修改回原本值的问题 #224

16.21.4.4 TiDB Ansible

- Spark 新增 log4j 日志配置 #842
- 更新 tispark jar 包为 v2.1.2 版本 #863
- 修复了 TiDB Binlog 使用 Kafka 或者 ZooKeeper 时导致生成的 Prometheus 配置文件格式错误的问题 #845
- 修复执行 `rolling_update.yml` 操作时，切换 PD Leader 失效的 Bug #888
- 优化滚动升级 PD 节点的逻辑，先升级 Follower 再升级 Leader，提高稳定性 #895

16.21.5 TiDB 2.1.15 Release Notes

发版日期：2019 年 7 月 18 日

TiDB 版本：2.1.15

TiDB Ansible 版本：2.1.15

16.21.5.1 TiDB

- 修复 DATE_ADD 函数处理微秒时由于没有对齐导致结果不正确的问题 #11289
- 修复 DELETE 语句中，字符串列中的空值与 FLOAT/INT 做比较时会报错的问题 #11279
- 修复 INSERT 函数参数有 NULL 值时，未正确返回 NULL 值的问题 #11249
- 修复在非字符串类型且长度为 0 的列建立索引时出错的问题 #11215
- 新增 SHOW TABLE REGIONS 的语句，支持通过 SQL 查询表的 Region 分布情况 #11238
- 修复 UPDATE ··· SELECT 语句因 SELECT 子查询中使用投影消除来优化规则所导致的报错 #11254
- 新增 ADMIN PLUGINS ENABLE/DISABLE SQL 语句，支持通过 SQL 动态开启/关闭 Plugin #11189
- Audit Plugin 新增审计连接功能 #11189
- 修复点查时，某列被查询多次而且结果为 NULL 时会 Panic 的问题 #11227
- 新增 tidb_scatter_region 配置项，控制创建表时是否开启自己打散 Record Region #11213
- 修复 RAND 函数由于非线程安全的 rand.Rand 导致的 Data Race 问题 #11170
- 修复某些情况下整数和非整数比较结果不正确的问题 #11191
- 支持修改 Database/Table 的 Collate，条件限制为 Database/Table 字符集必须是 UTF8/UTF8MB4 #11085
- 修复 CURRENT_TIMESTAMP 作为列的默认值且指定浮点精度时，SHOW CREATE TABLE 等语句显示精度不完整的问题 #11087

16.21.5.2 TiKV

- 统一日志格式 #5083
- 提高 region approximate size/key 在极端情况下的准确性，提升调度准确度 #5085

16.21.5.3 PD

- 统一日志格式 #1625

16.21.5.4 Tools

TiDB Binlog

- 优化 Pump GC 策略，去掉了未被在线 drainer 消费到的 binlog 保证不清理的限制 #663

TiDB Lightning

- 修复 SQL dump 指明的列名不是小写时导入错误的问题 #210

16.21.5.5 TiDB Ansible

- TiDB Dashboard 新增 parse duration 和 compile duration 监控项，用于监测 SQL 语句解析耗时和执行计划编译耗时 #815

16.21.6 TiDB 2.1.14 Release Notes

发版日期：2019 年 7 月 4 日

TiDB 版本：2.1.14

TiDB Ansible 版本：2.1.14

16.21.6.1 TiDB

- 修复某些情况下列裁剪导致查询结果不正确的问题 #11019
- 修复 `show processlist` 中 db 和 `info` 列信息显示有误的问题 #11000
- 修复 `MAX_EXECUTION_TIME` 作为 SQL hint 和全局变量在某些情况下不生效的问题 #10999
- 支持根据负载情况自动调整 Auto ID 分配的步长 #10997
- 修复 SQL 查询结束时 `MemTracker` 统计的 DistSQL 内存信息未正确清理的问题 #10971
- `information_schema.processlist` 表中新增 `MEM` 列用于描述 Query 的内存使用情况 #10896
- 新增全局系统变量 `max_execution_time`，用于控制查询的最大执行时间 #10940
- 修复使用未支持的聚合函数导致 TiDB panic 的问题 #10911
- 新增 `load data` 语句失败后自动回滚最后一个事务功能 #10862
- 修复 TiDB 超过内存配额的行为配置为 CANCEL 时，某些情况下 TiDB 返回结果不正确的问题 #11016
- 禁用 TRACE 语句，避免 TiDB panic 问题 #11039
- 新增 `mysql.expr_pushdown_blacklist` 系统表，控制动态开启/关闭 TiDB 下推到 Coprocessor 的函数 #10998
- 修复 `ANY_VALUE` 函数在 `ONLY_FULL_GROUP_BY` 模式下不生效的问题 #10994
- 修复给字符串类型的用户量赋值时因未深度拷贝导致赋值不正确的问题 #11043

16.21.6.2 TiKV

- 优化 Raftstore 消息处理中对空回调的处理流程，避免发送不必要的消息 #4682

16.21.6.3 PD

- 调整当读取到无效配置项时日志信息输出的级别，由 Error 调整为 Warning #1577

16.21.6.4 Tools

TiDB Binlog

- Reparo
  - 新增 `safe-mode` 配置项，开启后支持导入重复的数据 #662
- Pump
  - 新增 `stop-write-at-available-space` 配置项，限制 Binlog 空间保留的大小 #659
  - 修复 LevelDB L0 文件个数为 0 时 GC 有时不生效的问题 #648
  - 优化 log 文件删除的算法，加速释放空间 #648
- Drainer
  - 修复下游 TiDB BIT 类型列更新失败的问题 #655

16.21.6.5 TiDB Ansible

- 新增 ansible 命令及其 jmespath、Jinja2 依赖包的预检查功能 #807
- Pump 新增 stop-write-at-available-space 参数，当磁盘剩余空间小于该值（默认 10 GiB）时，Pump 停止写入 Binlog #807

16.21.7 TiDB 2.1.13 Release Notes

发版日期：2019 年 6 月 21 日

TiDB 版本：2.1.13

TiDB Ansible 版本：2.1.13

16.21.7.1 TiDB

- 新增列属性包含 AUTO_INCREMENT 时利用 SHARD_ROW_ID_BITS 打散行 ID 功能，缓解热点问题 #10788
- 优化无效 DDL 元信息存活时间，缩短集群升级后恢复 DDL 操作正常执行所需的时间 #10789
- 修复因持有 execdetails.ExecDetails 指针时 Coprocessor 的资源无法快速释放导致的在大并发场景下 OOM 的问题 #10833
- 新增 update-stats配置项，控制是否更新统计信息 #10772
- 新增 3 个 TiDB 特有语法，支持预先切分 Region，解决热点问题：
  - 新增 Table Option PRE_SPLIT_REGIONS 选项 #10863
  - 新增 SPLIT TABLE table_name INDEX index_name 语法 #10865
  - 新增 SPLIT TABLE [table_name] BETWEEN (min_value...)AND (max_value...)REGIONS [region_num ↪] 语法 #10882
- 修复某些情况下 KILL 语句导致的 panic 问题 #10879
- 增强 ADD_DATE 在某些情况下跟 MySQL 的兼容性 #10718
- 修复 index join 中内表过滤条件在某些情况下的选择率估计错误的问题 #10856

16.21.7.2 TiKV

- 修复因迭代器未检查状态导致系统生成残缺 snapshot 的问题 #4940
- 新增检查 block-size 配置的有效性功能 #4930

16.21.7.3 Tools

TiDB Binlog

- 修复 Pump 因写入失败时未检查返回值导致偏移量错误问题 #640
- Drainer 新增 advertise-addr 配置，支持容器环境中使用桥接模式 #634

16.21.8 TiDB 2.1.12 Release Notes

发版日期：2019 年 6 月 13 日

TiDB 版本：2.1.12

TiDB Ansible 版本：2.1.12

16.21.8.1 TiDB

- 修复在使用 feedback 时由于类型不匹配导致进程 panic 的问题 #10755
- 修复某些情况下改变字符集导致 BLOB 类型变成 TEXT 类型的问题 #10745
- 修复某些情况下在事务中的 GRANT 操作误报 “Duplicate Entry” 错误的问题 #10739
- 提升以下功能跟 MySQL 的兼容性：
  - DAYNAME 函数 #10732
  - MONTHNAME 函数 #10733
  - EXTRACT 函数在处理 MONTH 的时候支持零值 #10702
  - DECIMAL 类型转换成 TIMESTAMP 或者 DATETIME 类型 #10734
- 修改表的字符集时，同步修改列的字符集 #10714
- 修复某些情况下 DECIMAL 转换成浮点数的溢出问题 #10730
- 修复 TiDB 跟 TiKV 在 gRPC 最大封包设置不一致导致的某些超大封包报 “grpc: received message larger than max” 错误的问题 #10710
- 修复某些情况下 ORDER BY 没有过滤 NULL 值导致的 panic 问题 #10488
- 修复 UUID 函数的返回值，在多机器情况可能出现重复的问题 #10711
- `CAST(-num as datetime)` 的返回值由错误变更为 NULL 值 #10703
- 修复某些情况下 unsigned 列直方图遇到 signed 越界的问题 #10695
- 修复统计信息的 feedback 遇到 bigint unsigned 主键时处理不正确导致读数据时报错的问题 #10307
- 修复分区表某些情况下 `Show Create Table` 结果显示不正确的问题 #10690
- 修复在某些关联子查询上聚合函数 GROUP_CONCAT 计算不正确的问题 #10670
- 修复某些情况下 slow query 内存表在解析慢日志的时候导致的显示结果错乱的问题 #10776

16.21.8.2 PD

- 修复极端情况下 etcd Leader 选举阻塞的问题 #1576

16.21.8.3 TiKV

- 修复极端条件下 Leader 迁移过程中 Region 不可用的问题 #4799
- 修复在机器异常掉电时由于接收 snapshot 未及时将数据刷新到磁盘导致丢数据的问题 #4850

16.21.9 TiDB 2.1.11 Release Notes

发版日期：2019 年 6 月 03 日

TiDB 版本：2.1.11

TiDB Ansible 版本：2.1.11

16.21.9.1 TiDB

- 修复 delete 多表 join 的结果时使用错误 schema 的问题 #10595
- 修复 `CONVERT()` 函数返回错误的字段类型的问题 #10263
- 更新统计信息时合并不重叠的反馈信息 #10569
- 修复 `unix_timestamp()-unix_timestamp(now())` 计算错误的问题 #10491
- 修复 `period_diff` 与 MySQL 8.0 不兼容的问题 #10501
- 收集统计信息的时候，忽略 `Virtual Column`，避免异常报错 #10628
- 支持 `SHOW OPEN TABLES` 语句 #10374
- 修复某些情况下导致的 goroutine 泄露问题 #10656
- 修复某些情况下设置 `tidb_snapshot` 变量时间格式解析出错的问题 #10637

16.21.9.2 PD

- 修复因为 `balance-region` 可能会导致热点 Region 没有机会调度的问题 #1551
- 将热点相关调度的优先级改为高优先级 #1551
- 新增配置项 `hot-region-schedule-limit` 控制同时进行热点调度任务的数量及新增 `hot-region-cache-hits-threshold` 控制判断是否为热点 Region #1551

16.21.9.3 TiKV

- 修复在仅有一个 leader，learner 时，learner 读到空 index 的问题 #4751
- 将 `ScanLock` 和 `ResolveLock` 放在高优先级线程池中处理，减少对普通优先级命令的影响 #4791
- 同步所有收到的 snapshot 的文件 #4811

16.21.9.4 Tools

- TiDB Binlog
  - 新增 GC 删数据限速功能，避免因为删除数据导致 QPS 降低的问题 #620

16.21.9.5 TiDB Ansible

- 新增 Drainer 参数 #760

16.21.10 TiDB 2.1.10 Release Notes

发版日期：2019 年 5 月 22 日

TiDB 版本：2.1.10

TiDB Ansible 版本：2.1.10

16.21.10.1 TiDB

- 修复在使用 tidb_snapshot 读取历史数据的时候，某些异常情况导致的表结构不正确 #10359
- 修复 NOT 函数在某些情况下导致的读取结果错误的问题 #10363
- 修复 Generated Column 在 Replace 或者 Insert on duplicate update 语句中的错误行为 #10385
- 修复 BETWEEN 函数在 DATE/DATETIME 类型比较的一个 bug #10407
- 修复使用 SLOW_QUERY 表查询慢日志时，单行慢日志长度过长导致的报错 #10412
- 修复某些情况下 DATETIME 和 INTERVAL 相加的结果跟 MySQL 不一致的问题 #10416，#10418
- 增加闰年二月的非法时间的检查 #10417
- 内部的初始化操作限制只在 DDL Owner 中执行，避免了初始化集群的时候出现的大量冲突报错 #10426
- 修复 DESC 在输出时间戳列的默认值为 default current_timestamp on update current_timestamp 时跟 MySQL 不兼容的问题 #10337
- 修复 Update 语句中权限检查出错的问题 #10439
- 修复 CHAR 类型的列在某些情况下 RANGE 计算错误导致的错误结果的问题 #10455
- 避免 ALTER SHARD_ROW_ID_BITS 缩小 shard bits 位数在极低概率下，可能导致的数据错误 #9868
- 修复 ORDER BY RAND() 不返回随机数字的问题 #10064
- 禁止 ALTER 语句修改 DECIMAL 的精度 #10458
- 修复 TIME_FORMAT 函数与 MySQL 的兼容问题 #10474
- 检查 PERIOD_ADD 中参数的合法性 #10430
- 修复非法的 YEAR 字符串在 TiDB 中的表现跟 MySQL 不兼容的问题 #10493
- 支持 ALTER DATABASE 语法 #10503
- 修复 SLOW_QUERY 内存表在慢语句没有；的情况下报错的问题 #10536
- 修复某些情况下 Partitioned Table 的表 Add index 操作没有办法取消的问题 #10533
- 修复在某些情况下无法抓住内存使用太多导致 OOM 的问题 #10545
- 增强 DDL 操作改写表元信息的安全性 #10547

16.21.10.2 PD

- 修复 Leader 优先级不生效的问题 #1533

16.21.10.3 TiKV

- 拒绝在最近发生过成员变更的 Region 上执行 transfer leader，防止迁移失败 #4684
- Coprocessor metrics 上添加 priority 标签 #4643
- 修复 transfer leader 中可能发生的脏读问题 #4724
- 修复某些情况下 CommitMerge 导致 TiKV 不能重启的问题 #4615
- 修复 unknown 的日志 #4730

16.21.10.4 Tools

- TiDB Lightning
  - 新增 TiDB Lightning 发送数据到 importer 失败时进行重试 #176
- TiDB Binlog
  - 优化 Pump storage 组件 log，以利于排查问题 #607

16.21.10.5 TiDB Ansible

- 更新 TiDB Lightning 配置文件，新增 `tidb_lightning_ctl` 脚本 #d3a4a368

16.21.11 TiDB 2.1.9 Release Notes

发版日期：2019 年 5 月 6 日

TiDB 版本：2.1.9

TiDB Ansible 版本：2.1.9

16.21.11.1 TiDB

- 修复 `MAKETIME` 函数在 unsigned 类型溢出时的兼容性 #10089
- 修复常量折叠在某些情况下导致的栈溢出 #10189
- 修复 Update 在某些有别名的情况下权限检查的问题 #10157 #10326
- 追踪以及控制 DistSQL 中的内存使用 #10197
- 支持指定 collation 为 utf8mb4_0900_ai_ci #10201
- 修复主键为 Unsigned 类型的时候，MAX 函数结果错误的问题 #10209
- 修复在非 Strict SQL Mode 下可以插入 NULL 值到 NOT NULL 列的问题 #10254
- 修复 COUNT 函数在 DISTINCT 有多列的情况下结果错误的问题 #10270
- 修复 LOAD DATA 解析不规则的 CSV 文件时候 Panic 的问题 #10269
- 忽略 Index Lookup Join 中内外 join key 类型不一致的时候出现的 overflow 错误 #10244
- 修复某些情况下错误判定语句为 point-get 的问题 #10299
- 修复某些情况下时间类型未转换时区导致的结果错误问题 #10345
- 修复 TiDB 字符集在某些情况下大小写比较不一致的问题 #10354
- 支持控制算子返回的行数 #9166
  - Selection & Projection #10110
  - StreamAgg & HashAgg #10133
  - TableReader & IndexReader & IndexLookup #10169
- 慢日志改进
  - 增加 SQL Digest 用于区分同类 SQL #10093
  - 增加慢语句使用的统计信息的版本信息 #10220
  - 输出语句内存使用量 #10246
  - 调整 Coprocessor 相关信息的输出格式，让其能被 pt-query-digest 解析 #10300
  - 修复慢语句中带有 # 字符的问题 #10275
  - 增加一些信息的列到慢查询的内存表 #10317
  - 将事务提交时间算入慢语句执行时间 #10310
  - 修复某些时间格式无法被 pt-query-digest 解析的问题 #10323

16.21.11.2 PD

- 支持 GetOperator 服务 #1514

16.21.11.3 TiKV

- 修复在 transfer leader 时非预期的 quorum 变化 #4604

16.21.11.4 Tools

- TiDB Binlog
  - 修复 unsigned int 类型的主键列的 binlog 数据为负数，造成同步出错中断的问题 #574
  - 删除下游是 pb 时的压缩选项，修改下游名字 pb 成 `file` #597
  - 修复 2.1.7 引入的 Reparo 生成错误 update 语句的 bug #576
- TiDB Lightning
  - 修复 parser 解析 bit 类型的 column 数据错误的 bug #164
  - 使用 row id 或者列的默认值填充 dump 文件中缺少的 column 数据 #174
  - Importer 修复部分 SST 导入失败依然返回导入成功的 bug #4566
  - Importer 支持 upload SST 到 TiKV 限速 #4607
  - 修改 Importer RocksDB SST 压缩方法为 `lz4`，减少 CPU 消耗 #4624
- sync-diff-inspector
  - 支持 checkpoint #227

16.21.11.5 TiDB Ansible

- 更新 tidb-ansible 中的文档链接，兼容重构之后的文档 #740，#741
- 移除 `inventory.ini` 中的 `enable_slow_query_log` 参数，默认即将 slow log 输出到单独的日志文件中 #742

16.21.12 TiDB 2.1.8 Release Notes

发版日期：2019 年 4 月 12 日

TiDB 版本：2.1.8

TiDB Ansible 版本：2.1.8

16.21.12.1 TiDB

- 修复 `GROUP_CONCAT` 函数在参数存在 NULL 值情况下与 MySQL 处理逻辑不兼容的问题 #9930
- 修复在 Distinct 模式下 decimal 类型值之间相等比较的问题 #9931
- 修复 `SHOW FULL COLUMNS` 语句在 date，datetime，timestamp 类型的 Collation 的兼容性问题
  - #9938
  - #10114
- 修复过滤条件存在关联列的时候统计信息估算行数不准确的问题 #9937
- 修复 `DATE_ADD` 跟 `DATE_SUB` 函数的兼容性问题
  - #9963

  - #9966

- STR_TO_DATE 函数支持格式 %H，提升兼容性 #9964
- 修复 GROUP_CONCAT 函数在 group by 唯一索引的情况下结果错误的问题 #9969
- 当 Optimizer Hints 存在不匹配的表名的时候返回 warning #9970
- 统一日志格式规范，利于工具收集分析日志规范
- 修复大量 NULL 值导致统计信息估算不准确的问题 #9979
- 修复 TIMESTAMP 类型默认值为边界值的时候报错的问题 #9987
- 检查设置 `time_zone` 值的合法性 #10000
- 支持时间格式 `2019.01.01` #10001
- 修复某些情况下 EXPLAIN 结果中行数估计错误显示的问题 #10044
- 修复 `KILL TIDB [session id]` 某些情况下无法快速停止语句执行的问题 #9976
- 修复常量过滤条件在某些情况中谓词下推的问题 #10049
- 修复某些情况下 READ-ONLY 语句没有被当成 READ-ONLY 来处理的问题 #10048

16.21.12.2 PD

- 修复 Scatter Region 产生无效 Operator Step 的问题 #1482
- 修复 store 读热点的 key 统计不正确问题 #1487
- 修复 Region Merge Operator 超时时间过短的问题 #1495
- 添加 PD server 端处理 TSO 请求的耗时 metrics #1502

16.21.12.3 TiKV

- 修复读流量统计错误的问题 #4441
- 修复 Region 数过多的情况下 raftstore 的性能问题 #4484
- 调整当 level 0 SST 数量超过 `level_zero_slowdown_writes_trigger/2` 时不再继续 ingest file #4464

16.21.12.4 Tools

- Lightning 优化导入表的顺序，按照表的数据大小顺序进行导入，减少导入过程中大表执行 `Checksum` 和 `Analyze` 对集群的影响，并且提高 `Checksum` 和 `Analyze` 的成功率 #156
- 提升 Lightning encode SQL 性能，性能提升 50%，直接解析数据源文件内容成 TiDB 的 `types.Datum`，省去 KV encoder 的多余解析工作 #145
- TiDB Binlog Pump 新增 `storage.sync-log` 配置项，支持 Pump 本地存储异步刷盘 #529
- TiDB Binlog Pump 和 Drainer 之间通讯支持流量压缩 #530
- TiDB Binlog Drainer 新增 `syncer.sql-mode` 配置项，支持使用不同 `sql-mode` 解析 DDL query #513
- TiDB Binlog Drainer 新增 `syncer.ignore-table` 配置项，支持过滤不需要同步的表 #526

16.21.12.5 TiDB Ansible

- 修改操作系统版本限制，仅支持 CentOS 7.0 及以上，Red Hat 7.0 及以上版本的操作系统 #734
- 添加检测系统是否支持 `epollexclusive` #728
- 增加滚动升级版本限制，不允许从 2.0.1 及以下版本滚动升级到 2.1 及以上版本 #728

16.21.13 TiDB 2.1.7 Release Notes

发版日期：2019 年 3 月 28 日

TiDB 版本：2.1.7

TiDB Ansible 版本：2.1.7

16.21.13.1 TiDB

- 修复因 DDL 被取消导致升级程序时启动时间变长问题 #9768
- 修复配置项 `check-mb4-value-in-utf8` 在 `config.example.toml` 中位置错误的问题 #9852
- 提升内置函数 `str_to_date` 跟 MySQL 的兼容性 #9817
- 修复内置函数 `last_day` 的兼容性问题 #9750
- `infoschema.tables` 添加 `tidb_table_id` 列，方便通过 SQL 语句获取 `table_id`，新增 `tidb_indexes` 系统表管理 Table 与 Index 之间的关系 #9862
- 增加 Table Partition 定义为空的检查 #9663
- 将 `Truncate Table` 需要的权限由删除权限变为删表权限，与 MySQL 保持一致 #9876
- 支持在 DO 语句中使用子查询 #9877
- 修复变量 `default_week_format` 在 week 函数中不生效的问题 #9753
- 支持插件机制 #9880，#9888
- 支持使用系统变量 `log_bin` 查看 binlog 开启状况 #9634
- 支持使用 SQL 语句查看 Pump/Drainer 状态 #9896
- 修复升级时对 utf8 检查 mb4 字符的兼容性 #9887
- 修复某些情况下对 JSON 数据的聚合函数在计算过程中 Panic 的问题 #9927

16.21.13.2 PD

- 修改副本数为 1 时 balance-region 无法迁移 leader 问题 #1462

16.21.13.3 Tools

- 支持 binlog 同步 generated column #491

16.21.13.4 TiDB Ansible

- Prometheus 监控数据默认保留时间改成 30d

16.21.14 TiDB 2.1.6 Release Notes

2019 年 3 月 15 日，TiDB 发布 2.1.6 版，TiDB Ansible 相应发布 2.1.6 版本。相比 2.1.5 版本，该版本对系统稳定性、优化器、统计信息以及执行引擎做了很多改进。

16.21.14.1 TiDB

- 优化器/执行器
  - 当两个表在 `TIDB_INLJ` 的 Hint 中时，基于代价来选择外表 #9615
  - 修复在某些情况下，没有正确选择 IndexScan 的问题 #9587
  - 修复聚合函数在子查询里面的检查跟 MySQL 不兼容的行为 #9551
  - 使 `show stats_histograms` 语句只输出合法的列，避免 Panic #9502
- Server
  - 支持变量 `log_bin`，用于开启/关闭 Binlog #9634
  - 在事务中添加一个防御性检查，避免错误的事务提交 #9559
  - 修复设置变量导致的 Panic 的问题 #9539
- DDL
  - 修复 Create Table Like 语句在某些情况导致 Panic 的问题 #9652
  - 打开 etcd client 的 AutoSync 特性，防止某些情况下 TiDB 无法连接上 etcd 的问题 #9600

16.21.14.2 TiKV

- 修复在某些情况下解析 protobuf 失败导致 `StoreNotMatch` 错误的问题 #4303

16.21.14.3 Tools

- Lightning
  - importer 的默认的 region-split-size 变更为 512 MiB #4369
  - 保存原先在内存中的中间状态的 SST 到磁盘，减少内存使用 #4369
  - 限制 RocksDB 的内存使用 #4369
  - 修复 Region 还没有调度完成时进行 scatter 的问题 #4369
  - 将大表的数据和索引分离导入，在分批导入时能有效降低耗时 #132
  - 支援 CSV #111
  - 修复库名中含非英数字符时导入失败的错误 #9547

16.21.15 TiDB 2.1.5 Release Notes

2019 年 2 月 28 日，TiDB 发布 2.1.5 版，TiDB Ansible 相应发布 2.1.5 版本。相比 2.1.4 版本，该版本对系统稳定性、优化器、统计信息以及执行引擎做了很多改进。

16.21.15.1 TiDB

- 优化器/执行器
  - 当列的字符集信息和表的字符集信息相同时，`SHOW CREATE TABLE` 不再打印列的字符集信息，使其结果更加兼容 MySQL #9306

  - 将 Sort 算子中的表达计算抽取出来用一个 Project 算子完成，简化 Sort 算子的计算逻辑，修复某些情况下 Sort 算子结果不正确或者 panic 的问题 #9319
  - 移除 Sort 算子中的数值为常量的排序字段 #9335，#9440
  - 修复向无符号整数列插入数据时数据溢出的问题 #9339
  - 目标 binary 的长度超过 max_allowed_packet 时，将 cast_as_binary 设置为 NULL #9349
  - 优化 IF 和 IFNULL 的常量折叠过程 #9351
  - 使用 skyline pruning 优化 TiDB 的索引选择，增加简单查询的稳定性 #9356
  - 支持对析取范式计算选择率 #9405
  - 修复 !=ANY() 和 =ALL() 在某些情况下 SQL 查询结果不正确的问题 #9403
  - 修复执行 Merge Join 操作的两个表的 Join Key 类型不同时结果可能不正确或者 panic 的问题 #9438
  - 修复某些情况下 RAND() 函数结果和 MySQL 不兼容的问题 #9446
  - 重构 Semi Join 对 NULL 值和空结果集的处理逻辑，使其返回正确的结果，更加兼容 MySQL #9449
- Server
  - 增加系统变量 tidb_constraint_check_in_place，在 INSERT 语句执行时即检查数据的唯一性约束 #9401
  - 修复系统变量 tidb_force_priority 的值和配置文件中的设置的值不一致的问题 #9347
  - 在 general log 中增加 “current_db” 字段打印当前使用的数据库 #9346
  - 增加通过表 ID 来获取表信息的 HTTP API #9408
  - 修复 LOAD DATA 在某些情况下导入数据不正确的问题 #9414
  - 修复某些情况下，客户端建立连接慢的问题 #9451
- DDL
  - 修复撤销 DROP COLUMN 操作中的一些问题 #9352
  - 修复撤销 DROP/ADD 分区表操作中的一些问题 #9376
  - 修复某些情况下 ADMIN CHECK TABLE 误报数据索引不一致的问题 #9399
  - 修复 TIMESTAMP 类型的默认值在时区上的一些问题 #9108

16.21.15.2 PD

- GetAllStores 接口提供了 exclude_tombstone_stores 选项，将 Tombstone store 从返回结果中去除 #1444

16.21.15.3 TiKV

- 修复了某些情况下 Importer 导入失败的问题 #4223
- 修复了某些情况下 “key not in region” 错误 #4125
- 修复了某些情况下 Region merge 导致 panic 的问题 #4235
- 添加了详细的 StoreNotMatch 错误信息 #3885

16.21.15.4 Tools

- Lightning
  - 集群中有 Tombstone store 时 Lightning 不会再报错退出 #4223
- TiDB Binlog
  - 修正 DDL Binlog 同步方案，确保 DDL 同步的正确性 #9304

16.21.16 TiDB 2.1.4 Release Notes

2019 年 2 月 15 日，TiDB 发布 2.1.4 版，TiDB Ansible 相应发布 2.1.4 版本。相比 2.1.3 版本，该版本对系统稳定性、优化器、统计信息以及执行引擎做了很多改进。

16.21.16.1 TiDB

- 优化器/执行器
  - 修复 VALUES 函数未正确处理 FLOAT 类型的问题 #9223
  - 修复某些情况下 CAST 浮点数成字符串结果不正确的问题 #9227
  - 修复 FORMAT 函数在某些情况下结果不正确的问题 #9235
  - 修复某些情况下处理 Join 查询时 panic 的问题 #9264
  - 修复 VALUES 函数未正确处理 ENUM 类型的问题 #9280
  - 修复 DATE_ADD/DATE_SUB 在某些情况下结果不正确的问题 #9284
- Server
  - 优化 reload privilege success 日志，将其调整为 DEBUG 级别 #9274
- DDL
  - `tidb_ddl_reorg_worker_cnt` 和 `tidb_ddl_reorg_batch_size` 变成 GLOBAL 变量 #9134
  - 修复某些异常情况下，在 Generated column 增加索引导致的 Bug #9289

16.21.16.2 TiKV

- 修复在 TiKV 关闭时可能发生重复写的问题 #4146
- 修复某些情况下 event listener 结果处理异常的问题 #4132

16.21.16.3 Tools

- Lightning
  - 优化内存使用 #107，#108
  - 去掉 dump files 的 chunk 划分，减少对 dump files 的一次额外解析 #109
  - 限制读取 dump files 的 I/O 并发，避免过多的 cache miss 导致性能下降 #110
  - 对单个表实现 batch 导入，提高导入的稳定性 #110
  - TiKV 在 import 模式下开启 auto compactions #4199
  - 增加禁用 TiKV periodic Level-1 compaction 参数，因为当 TiKV 集群为 2.1.4 或更高版本时，在导入模式下会自动执行 Level-1 compaction #119
  - 限制 import engines 数量，避免过大占用 importer 磁盘空间 #119
- 数据同步对比统计 (sync-diff-inspector) 支持使用 TiDB 统计信息来划分 chunk #197

16.21.17 TiDB 2.1.3 Release Notes

2019 年 01 月 28 日，TiDB 发布 2.1.3 版，TiDB Ansible 相应发布 2.1.3 版本。相比 2.1.2 版本，该版本对系统稳定性、优化器、统计信息以及执行引擎做了很多改进。

16.21.17.1 TiDB

- 优化器/执行器
  - 修复某些情况下 Prepared Plan Cache panic 的问题 #8826
  - 修复在有前缀索引的某些情况下，Range 计算错误的问题 #8851
  - 当 SQL_MODE 不为 STRICT 时，`CAST(str AS TIME(N))` 在 `str` 为非法的 TIME 格式的字符串时返回 NULL #8966
  - 修复某些情况下 Generated Column 在 Update 中 Panic 的问题 #8980
  - 修复统计信息直方图某些情况下上界溢出的问题 #8989
  - 支持对 `_tidb_rowid` 构造查询的 Range，避免全表扫，减轻集群压力 #9059
  - `CAST(AS TIME)` 在精度太大的情况下返回一个错误 #9058
  - 允许把 `Sort Merge Join` 用于笛卡尔积 #9037
  - 修复统计信息的 worker 在某些情况下 panic 之后无法恢复的问题 #9085
  - 修复某些情况下 `Sort Merge Join` 结果不正确的问题 #9046
  - 支持在 CASE 子句返回 JSON 类型 #8355
- Server
  - 当语句中有非 TiDB hint 的注释时返回警告，而不是错误 #8766
  - 验证设置的 TIMEZONE 的合法性 #8879
  - 优化 Metrics 项 `QueryDurationHistogram`，展示更多语句的类型 #8875
  - 修复 bigint 某些情况下下界溢出的问题 #8544
  - 支持 ALLOW_INVALID_DATES SQL mode #9110
- DDL
  - 修复一个 RENAME TABLE 的兼容性问题，保持行为跟 MySQL 一致 #8808
  - 支持 `ADD INDEX` 的并发修改即时生效 #8786
  - 修复在 `ADD COLUMN` 的过程中，某些情况 Update 语句 panic 的问题 #8906
  - 修复某些情况下并发创建 Table Partition 的问题 #8902
  - 支持把 `utf8` 字符集转换为 `utf8mb4` 字符集 #8951 #9152
  - 处理 Shard Bits 溢出的问题 #8976
  - 支持 `SHOW CREATE TABLE` 输出列的字符集 #9053
  - 修复 varchar 最大支持字符数在 `utf8mb4` 下限制的问题 #8818
  - 支持 `ALTER TABLE TRUNCATE TABLE PARTITION` #9093
  - 修复创建表的时候缺省字符集推算的问题 #9147

16.21.17.2 PD

- 修复 Leader 选举相关的 Watch 问题 #1396

16.21.17.3 TiKV

- 支持了使用 HTTP 方式获取监控信息 #3855
- 修复 `data_format` 遇到 NULL 时的问题 #4075
- 添加验证 Scan 请求的边界合法性 #4124

16.21.17.4 Tools

- TiDB Binlog
  - 修复在启动或者重启时 `no available pump` 的问题 #157
  - 开启 Pump client log 输出 #165
  - 修复表只有 unique key 没有 primary key 的情况下，unique key 包含 NULL 值导致数据更新不一致的问题

16.21.18 TiDB 2.1.2 Release Notes

2018 年 12 月 22 日，TiDB 发布 2.1.2 版，TiDB Ansible 相应发布 2.1.2 版本。该版本在 2.1.1 版的基础上，对系统兼容性、稳定性做出了改进。

16.21.18.1 TiDB

- 兼容 Kafka 版本的 TiDB Binlog #8747
- 完善滚动升级下 TiDB 的退出机制 #8707
- 修复在某些情况下为 generated column 增加索引 panic 的问题 #8676
- 修复在某些情况下语句有 `TIDB_SMJ Hint` 的时候优化器无法找到正确执行计划的问题 #8729
- 修复在某些情况下 `AntiSemiJoin` 返回错误结果的问题 #8730
- 增强 `utf8` 字符集的有效字符检查 #8754
- 修复事务中先写后读的情况下时间类型字段可能返回错误结果的问题 #8746

16.21.18.2 PD

- 修复 Region Merge 相关的 Region 信息更新问题 #1377

16.21.18.3 TiKV

- 支持以日 (d) 为时间单位的配置格式，并解决配置兼容性问题 #3931
- 修复 Approximate Size Split 可能会 panic 的问题 #3942
- 修复两个 Region merge 相关问题 #3822，#3873

16.21.18.4 Tools

- TiDB Lightning
  - 支持最小 TiDB 集群版本为 2.1.0
  - 修复解析包含 JSON 类型数据的文件内容出错 #144
  - 修复使用 checkpoint 重启后 `Too many open engines` 错误
- TiDB Binlog
  - 消除了 Drainer 往 Kafka 写数据的一些瓶颈点
  - TiDB 支持写 Kafka 版本的 TiDB Binlog

16.21.19 TiDB 2.1.1 Release Notes

2018 年 12 月 12 日，TiDB 发布 2.1.1 版。相比 2.1.0 版本，该版本对系统稳定性、优化器、统计信息以及执行引擎做了很多改进。

16.21.19.1 TiDB

- 优化器/执行器
  - 修复时间为负值时的四舍五入错误 #8574
  - 修复 `uncompress` 函数未检查数据长度的问题 #8606
  - 在执行 `execute` 命令后重置 `prepare` 语句绑定的变量 #8652
  - 支持对分区表自动收集统计信息 #8649
  - 修复在下推 `abs` 函数时设置错误的整数类型 #8628
  - 修复 JSON 列的数据竞争问题 #8660
- Server
  - 修复在 PD 故障时获取错误 TSO 的问题 #8567
  - 修复不规范的语句导致启动失败的问题 #8576
  - 修复在事务重试时使用了错误的参数 #8638
- DDL
  - 将表的默认字符集和排序规则改为 `utf8mb4` 和 `utf8mb4_bin` #8590
  - 增加变量 `ddl_reorg_batch_size` 来控制添加索引的速度 #8614
  - DDL 中的 character set 和 collation 选项内容不再大小写敏感 #8611
  - 修复对于生成列添加索引的问题 #8655

16.21.19.2 PD

- 修复一些配置项无法在配置文件中设置为 0 的问题 #1334
- 启动时检查未定义的配置 #1362
- 避免 transfer leader 至新创建的 Peer，优化可能产生的延迟增加问题 #1339
- 修复 RaftCluster 在退出时可能的死锁问题 #1370

16.21.19.3 TiKV

- 避免 transfer leader 至新创建的 Peer，优化可能产生的延迟增加问题 #3878

16.21.19.4 Tools

- Lightning
  - 优化对导入表的 `analyze` 机制，提升了导入速度
  - 支持 checkpoint 信息储存在本地文件
- TiDB Binlog
  - 修复 pb files 输出 bug，表只有主键列则无法产生 pb event

16.21.20 TiDB 2.1 GA Release Notes

2018 年 11 月 30 日，TiDB 发布 2.1 GA 版。相比 2.0 版本，该版本对系统稳定性、性能、兼容性、易用性做了大量改进。

16.21.20.1 TiDB

- SQL 优化器
  - 优化 `Index Join` 选择范围，提升执行性能
  - 优化 `Index Join` 外表选择，使用估算的行数较少的表作为外表
  - 扩大 Join Hint `TIDB_SMJ` 的作用范围，在没有合适索引可用的情况下也可使用 Merge Join
  - 加强 Join Hint `TIDB_INLJ` 的能力，可以指定 Join 中的内表
  - 优化关联子查询，包括下推 Filter 和扩大索引选择范围，部分查询的效率有数量级的提升
  - 支持在 `UPDATE` 和 `DELETE` 语句中使用 Index Hint 和 Join Hint
  - 支持更多函数下推：`ABS/CEIL/FLOOR/IS TRUE/IS FALSE`
  - 优化内建函数 `IF` 和 `IFNULL` 的常量折叠算法
  - 优化 `EXPLAIN` 语句输出格式，使用层级结构表示算子之间的上下游关系
- SQL 执行引擎
  - 重构所有聚合函数，提升 `Stream` 和 `Hash` 聚合算子的执行效率
  - 实现并行 `Hash Aggregate` 算子，部分场景下有 350% 的性能提升
  - 实现并行 `Project` 算子，部分场景有 74% 的性能提升
  - 并发地读取 `Hash Join` 的 `Inner` 表和 `Outer` 表的数据，提升执行性能
  - 优化 `REPLACE INTO` 语句的执行速度，性能提升 10x
  - 优化时间类型的内存占用，时间类型数据的内存使用降低为原来的一半
  - 优化点查的查询性能，Sysbench 点查效率提升 60%
  - TiDB 插入和更新宽表，性能提升接近 20 倍
  - 支持在配置文件中设置单个查询的内存使用上限
  - 优化 `Hash Join` 的执行过程，当 Join 类型为 `Inner Join` 或者 `Semi Join` 时，如果内表为空，不再读取外表数据，快速返回结果
  - 支持 `EXPLAIN ANALYZE` 语句，用于查看 Query 执行过程中各个算子的运行时间，返回结果行数等运行时统计信息
- 统计信息
  - 支持只在一天中的某个时间段开启统计信息自动 `ANALYZE` 的功能
  - 支持根据查询的反馈自动更新表的统计信息
  - 支持通过 `ANALYZE TABLE WITH BUCKETS` 语句配置直方图中桶的个数
  - 优化等值查询和范围查询混合的情况下使用直方图估算 Row Count 的算法
- 表达式
  - 支持内建函数：
    * `json_contains`
    * `json_contains_path`

    * encode/decode

- Server
  - 支持在单个 tidb-server 实例内部对冲突事务排队，优化事务间冲突频繁的场景下的性能
  - 支持 Server Side Cursor
  - 新增 HTTP 管理接口
  - 打散 table 的 Regions 在 TiKV 集群中的分布
  - 控制是否打开 `general log`
  - 在线修改日志级别
  - 查询 TiDB 集群信息
  - 添加 `auto_analyze_ratio` 系统变量控制自动 Analyze 的阈值
  - 添加 `tidb_retry_limit` 系统变量控制事务自动重试的次数
  - 添加 `tidb_disable_txn_auto_retry` 系统变量控制事务是否自动重试
  - 支持使用 `admin show slow` 语句来获取慢查询语句
  - 增加环境变量 `tidb_slow_log_threshold` 动态设置 slow log 的阈值
  - 增加环境变量 `tidb_query_log_max_len` 动态设置日志中被截断的原始 SQL 语句的长度
- DDL
  - 支持 Add Index 语句与其他 DDL 语句并行执行，避免耗时的 Add Index 操作阻塞其他操作
  - 优化 `Add Index` 的速度，在某些场景下速度大幅提升
  - 支持 `select tidb_is_ddl_owner()` 语句，方便判断 TiDB 是否为 `DDL Owner`
  - 支持 `ALTER TABLE FORCE` 语法
  - 支持 `ALTER TABLE RENAME KEY TO` 语法
  - `Admin Show DDL Jobs` 输出结果中添加表名、库名等信息
  - 支持使用 `ddl/owner/resign` HTTP 接口释放 DDL Owner 并开启新一轮 DDL Owner 选举
- 兼容性
  - 支持更多 MySQL 语法
  - `BIT` 聚合函数支持 `ALL` 参数
  - 支持 `SHOW PRIVILEGES` 语句
  - 支持 `LOAD DATA` 语句的 `CHARACTER SET` 语法
  - 支持 `CREATE USER` 语句的 `IDENTIFIED WITH` 语法
  - 支持 `LOAD DATA IGNORE LINES` 语句
  - `Show ProcessList` 语句返回更准确信息

16.21.20.2 PD (Placement Driver)

- 可用性优化
  - 引入 TiKV 版本控制机制，支持集群滚动兼容升级
  - PD 节点间开启 `Raft PreVote`，避免网络隔离后恢复时产生的重新选举
  - 开启 `raft learner` 功能，降低调度时出现宕机导致数据不可用的风险
  - TSO 分配不再受系统时间回退影响
  - 支持 `Region merge` 功能，减少元数据带来的开销

- 调度器优化
  - 优化 Down Store 的处理流程，加快发生宕机后补副本的速度
  - 优化热点调度器，在流量统计信息抖动时适应性更好
  - 优化 Coordinator 的启动，减少重启 PD 时带来的不必要调度
  - 优化 Balance Scheduler 频繁调度小 Region 的问题
  - 优化 Region merge，调度时考虑 Region 中数据的行数
  - 新增一些控制调度策略的开关
  - 完善调度模拟器，添加调度场景模拟
- API 及运维工具
  - 新增 `GetPrevRegion` 接口，用于支持 TiDB reverse scan 功能
  - 新增 `BatchSplitRegion` 接口，用于支持 TiKV 快速 Region 分裂
  - 新增 `GCSafePoint` 接口，用于支持 TiDB 并发分布式 GC
  - 新增 `GetAllStores` 接口，用于支持 TiDB 并发分布式 GC
  - pd-ctl 新增：
    * 使用统计信息进行 Region split
    * 调用 `jq` 来格式化 JSON 输出
    * 查询指定 store 的 Region 信息
    * 查询按 version 排序的 topN 的 Region 列表
    * 查询按 size 排序的 topN 的 Region 列表
    * 更精确的 TSO 解码
  - pd-recover 不再需要提供 max-replica 参数
- 监控
  - 增加 `Filter` 相关的监控
  - 新增 etcd Raft 状态机相关监控
- 性能优化
  - 优化处理 Region heartbeat 的性能，减少 heartbeat 带来的内存开销
  - 优化 Region tree 性能
  - 优化计算热点统计的性能问题

16.21.20.3 TiKV

- Coprocessor
  - 新增支持大量内建函数
  - 新增 Coprocessor ReadPool，提高请求处理并发度
  - 修复时间函数解析以及时区相关问题
  - 优化下推聚合计算的内存使用
- Transaction

  - 优化 MVCC 读取逻辑以及内存使用效率，提高扫描操作的性能，Count 全表性能比 2.0 版本提升 1 倍
  - 折叠 MVCC 中连续的 Rollback 记录，保证记录的读取性能
  - 新增 UnsafeDestroyRange API 用于在 drop table/index 的情况下快速回收空间
  - GC 模块独立出来，减少对正常写入的影响
  - kv_scan 命令支持设置 upper bound

- Raftstore

  - 优化 snapshot 文件写入流程避免导致 RocksDB stall
  - 增加 LocalReader 线程专门处理读请求，降低读请求的延迟
  - 支持 BatchSplit 避免大量写入导致产生特别大的 Region
  - 支持按照统计信息进行 Region Split，减少 IO 开销
  - 支持按照 Key 的数量进行 Region Split，提高索引扫描的并发度
  - 优化部分 Raft 消息处理流程，避免 Region Split 带来不必要的延迟
  - 启用 PreVote 功能，减少网络隔离对服务的影响

- 存储引擎

  - 修复 RocksDB CompactFiles 的 bug，可能影响 Lightning 导入的数据
  - 升级 RocksDB 到 v5.15，解决 snapshot 文件可能会被写坏的问题
  - 优化 IngestExternalFile，避免 flush 卡住写入的问题

- tikv-ctl

  - 新增 ldb 命令，方便排查 RocksDB 相关问题
  - compact 命令支持指定是否 compact bottommost 层的数据

16.21.20.4 Tools

- 全量数据快速导入工具 TiDB Lightning
- 支持新版本 TiDB Binlog

16.21.20.5 升级兼容性说明

- 由于新版本存储引擎更新，不支持在升级后回退至 2.0.x 或更旧版本
- 从 2.0.6 之前的版本升级到 2.1 之前，最好确认集群中是否存在正在运行中的 DDL 操作，特别是耗时的 Add Index 操作，等 DDL 操作完成后再执行升级操作
- 因为 2.1 版本启用了并行 DDL，对于早于 2.0.1 版本的集群，无法滚动升级到 2.1，可以选择下面两种方案：

  - 停机升级，直接从早于 2.0.1 的 TiDB 版本升级到 2.1
  - 先滚动升级到 2.0.1 或者之后的 2.0.x 版本，再滚动升级到 2.1 版本

16.21.21 TiDB 2.1 RC5 Release Notes

2018 年 11 月 12 日，TiDB 发布 2.1 RC5 版。相比 2.1 RC4 版本，该版本对系统稳定性、优化器、统计信息以及执行引擎做了很多改进。

16.21.21.1 TiDB

- SQL 优化器
  - 修复 IndexReader 在某些情况下读取的 handle 不正确的问题 #8132
  - 修复 IndexScan Prepared 语句在使用 Plan Cache 的时候的问题 #8055
  - 修复 Union 语句结果不稳定的问题 #8165
- 执行器
  - 提升 TiDB 插入和更新宽表的性能 #8024
  - 内建函数 Truncate 支持 unsigned int 参数 #8068
  - 修复转换 JSON 数据到 decimal 类型出错的问题 #8109
  - 修复 float 类型在 Update 时出错的问题 #8170
- 统计信息
  - 修复点查在某些情况下，统计信息出现错误的问题 #8035
  - 修复统计信息某些情况下在 primary key 的选择率的问题 #8149
  - 修复被删除的表的统计信息长时间没有清理的问题 #8182
- Server
  - 提升日志的可读性，完善日志信息
    * #8063
    * #8053
    * #8224
  - 修复获取 infoschema.profiling 表数据出错的问题 #8096
  - 替换 unix socket，使用 pumps client 来写 binlog #8098
  - 增加环境变量 tidb_slow_log_threshold 动态设置 slow log 的阈值 #8094
  - 增加环境变量 tidb_query_log_max_len 动态设置日志中被截断的原始 SQL 语句的长度 #8200
  - 增加环境变量 tidb_opt_write_row_id 来控制是否允许写入 _tidb_rowid #8218
  - ticlient Scan 命令增加边界，解决数据扫出边界的问题 #8081，#8247
- DDL
  - 修复在事务中某些情况下执行 DDL 语句出错的问题 #8056
  - 修复 partition 分区表执行 truncate table 没有生效的问题 #8103
  - 修复某些情况下 DDL 操作在被 cancel 之后没有正确回滚的问题 #8057
  - 增加命令 admin show next_row_id，返回下一个可用的行 ID #8268

16.21.21.2 PD

- 修复 pd-ctl 读取 Region key 的相关问题
  - #1298
  - #1299
  - #1308
- 修复 regions/check API 输出错误的问题 #1311
- 修复 PD join 失败后无法重新 join 的问题 #1279
- 修复某些情况下 watch leader 会丢失事件的问题 #1317

16.21.21.3 TiKV

- 优化 `WriteConflict` 报错信息 #3750
- 增加 panic 标记文件 #3746
- 降级 grpcio，避免新版本 gRPC 导致的 segment fault 问题 #3650
- 增加 `kv_scan` 接口扫描上界的限制 #3749

16.21.21.4 Tools

- TiDB 支持 TiDB Binlog cluster，不兼容旧版本 TiDB Binlog #8093，使用文档

16.21.22 TiDB 2.1 RC4 Release Notes

2018 年 10 月 23 日，TiDB 发布 2.1 RC4 版。相比 2.1 RC3 版本，该版本对系统稳定性、优化器、统计信息以及执行引擎做了很多改进。

16.21.22.1 TiDB

- SQL 优化器
  - 修复某些情况下 `UnionAll` 的列裁剪不正确的问题 #7941
  - 修复某些情况下 `UnionAll` 算子结果不正确的问题 #8007
- SQL 执行引擎
  - 修复 AVG 函数的精度问题 #7874
  - 支持通过 `EXPLAIN ANALYZE` 语句查看 Query 执行过程中各个算子的运行时间，返回结果行数等运行时统计信息 #7925
  - 修复多次引用同一列时 `PointGet` 算子 panic 的问题 #7943
  - 修复当 `Limit` 子句中的值太大时 panic 的问题 #8002
  - 修复某些情况下 `AddDate/SubDate` 执行过程中 panic 的问题 #8009
- 统计信息
  - 修复将组合索引的直方图下边界前缀判断为越界的问题 #7856
  - 修复统计信息收集引发的内存泄漏问题 #7873
  - 修复直方图为空时 panic 的问题 #7928
  - 修复加载统计信息时直方图边界越界的问题 #7944
  - 限制统计信息采样过程中数值的最大长度 #7982
- Server
  - 重构 Latch，避免事务冲突误判，提升并发事务的执行性能 #7711
  - 修复某些情况下收集 Slow Query 导致的 panic 问题 #7874
  - 修复 `LOAD DATA` 语句中，`ESCAPED BY` 为空字符串时 panic 的问题 #8005
  - 完善 “coprocessor error” 日志信息 #8006
- 兼容性
  - 当 Query 为空时，将 `SHOW PROCESSLIST` 结果中的 Command 字段设置为 “Sleep” #7839

- 表达式
  - 修复 SYSDATE 函数被常量折叠的问题 #7895
  - 修复 SUBSTRING_INDEX 在某些情况下 panic 的问题 #7897
- DDL
  - 修复抛出 “invalid ddl job type” 的错误时导致栈溢出的问题 #7958
  - 修复某些情况下 ADMIN CHECK TABLE 结果不正确的问题 #7975

16.21.22.2 PD

- 修复下线后的 TiKV 没有从 Grafana 面板中移除的问题 #1261
- 修复 grpc-go 设置 status 时的 data race 问题#1265
- 修复 etcd 启动失败导致的服务挂起问题 #1267
- 修复 leader 切换过程中可能产生的 data race #1273
- 修复下线 TiKV 时可能输出多余 warning 日志的问题 #1280

16.21.22.3 TiKV

- 优化 apply snapshot 导致的 RocksDB Write stall 的问题 #3606
- 增加 raftstore tick 相关 metrics #3657
- 升级 RocksDB，修复写入卡死及 IngestExternalFile 时可能写坏源文件的问题 #3661
- 升级 grpcio，修复 “too many pings” 误报的问题 #3650

16.21.23 TiDB 2.1 RC3 Release Notes

2018 年 9 月 29 日，TiDB 发布 2.1 RC3 版。相比 2.1 RC2 版本，该版本对系统稳定性、兼容性、优化器以及执行引擎做了很多改进。

16.21.23.1 TiDB

- SQL 优化器
  - 修复语句内包含内嵌的 LEFT OUTER JOIN 时，结果不正确的问题 #7689
  - 增强 JOIN 语句上的 predicate pushdown 优化规则 #7645
  - 修复 UnionScan 算子的 predicate pushdown 优化规则 #7695
  - 修复 Union 算子的 unique key 属性设置不正确的问题 #7680
  - 增强常量折叠的优化规则 #7696
  - 把常量传播后的 filter 是 null 的 data source 优化成 table dual #7756
- SQL 执行引擎
  - 优化事务内读请求的性能 #7717
  - 优化部分执行器 Chunk 内存分配的开销 #7540
  - 修复点查全部为 NULL 的列导致数组越界的问题 #7790
- Server

  - 修复配置文件里内存配额选项不生效的问题 #7729
  - 添加 tidb_force_priority 系统变量用来整体设置语句执行的优先级 #7694
  - 支持使用 `admin show slow` 语句来获取 SLOW QUERY LOG #7785
- 兼容性
  - 修复 `information_schema.schemata` 里 `charset/collation` 结果不正确的问题 #7751
  - 修复 `hostname` 系统变量的值为空的问题 #7750
- 表达式
  - 内建函数 `AES_ENCRYPT/AES_DECRYPT` 支持 `init_vecter` 参数 #7425
  - 修复部分表达式 `Format` 结果不正确的问题 #7770
  - 支持内建函数 `JSON_LENGTH` #7739
  - 修复 unsigned integer 类型 cast 为 decimal 类型结果不正确的问题 #7792
- DML
  - 修复 `INSERT ⋯ ON DUPLICATE KEY UPDATE` 语句在 unique key 更新时结果不正确的问题 #7675
- DDL
  - 修复在新建的 timestamp 类型的列上新建索引时，索引值没有做时区转换的问题 #7724
  - 支持 enum 类型 append 新的值 #7767
  - 快速新建 etcd session，使网络隔离后，集群更快恢复可用 #7774

16.21.23.2 PD

- 新特性
  - 添加获取按大小逆序排序的 Region 列表 API (/size) #1254
- 功能改进
  - Region API 会返回更详细的信息 #1252
- Bug 修复
  - 修复 PD 切换 leader 以后 `adjacent-region-scheduler` 可能会导致 crash 的问题 #1250

16.21.23.3 TiKV

- 性能优化
  - 优化函数下推的并发支持 #3515
- 新特性
  - 添加对 Log 函数的支持 #3603
  - 添加对 sha1 函数的支持 #3612
  - 添加 `truncate_int` 函数的支持 #3532
  - 添加 `year` 函数的支持 #3622
  - 添加 `truncate_real` 函数的支持 #3633
- Bug 修复
  - 修正时间函数相关的报错行为 #3487 #3615
  - 修复字符串解析成时间与 TiDB 不一致的问题 #3589

16.21.24 TiDB 2.1 RC2 Release Notes

2018 年 9 月 14 日，TiDB 发布 2.1 RC2 版。相比 2.1 RC1 版本，该版本对系统稳定性、优化器、统计信息以及执行引擎做了很多改进。

16.21.24.1 TiDB

- SQL 优化器
  - 新版 Planner 设计方案 #7543
  - 提升常量传播优化规则 #7276
  - 增强 Range 的计算逻辑使其能够同时处理多个 IN 或者等值条件 #7577
  - 修复当 Range 为空时，TableScan 的估算结果不正确的问题 #7583
  - 为 UPDATE 语句支持 PointGet 算子 #7586
  - 修复 FirstRow 聚合函数某些情况下在执行过程中 panic 的问题 #7624
- SQL 执行引擎
  - 解决 HashJoin 算子在遇到错误的情况下潜在的 DataRace 问题 #7554
  - HashJoin 算子同时读取内表数据和构建 Hash 表 #7544
  - 优化 Hash 聚合算子性能 #7541
  - 优化 Join 算子性能 #7493、 #7433
  - 修复 UPDATE JOIN 在 Join 顺序改变后结果不正确的问题 #7571
  - 提升 Chunk 迭代器的性能 #7585
- 统计信息
  - 解决重复自动 Analyze 统计信息的问题 #7550
  - 解决统计信息无变化时更新统计信息遇到错误的问题 #7530
  - Analyze 执行时使用低优先级以及 RC 隔离级别 #7496
  - 支持只在一天中的某个时间段开启统计信息自动更新的功能 #7570
  - 修复统计信息写日志时发生的 panic #7588
  - 支持通过 ANALYZE TABLE WITH BUCKETS 语句配置直方图中桶的个数 #7619
  - 修复更新空的直方图时 panic 的问题 #7640
  - 使用统计信息更新 information_schema.tables.data_length #7657
- Server
  - 增加 Trace 相关的依赖库 #7532
  - 开启 Golang 的 mutex profile 功能 #7512
  - Admin 语句需要 Super_priv 权限 #7486
  - 禁止用户 Drop 关键的系统表 #7471
  - 从 juju/errors 切换到 pkg/errors #7151
  - 完成 SQL Tracing 功能原型 #7016
  - 删除 goroutine pool #7564
  - 支持使用 USER1 信号来查看 goroutine 信息 #7587
  - 将 TiDB 启动时的内部 SQL 设置为高优先级 #7616
  - 在监控中用不同的标签区分内部 SQL 和用户 SQL #7631
  - 缓存最近一周内最慢的 30 条慢查询日志在 TiDB Server 上 #7646

  - TiDB 集群设置时区的方案 #7656
  - 丰富 GC life time is shorter than transaction duration 错误信息 #7658
  - 在 TiDB 集群启动时设置集群时区信息 #7638

- 兼容性
  - Year 类型字段增加 unsigned flag #7542
  - 修复在 Prepare/Execute 模式下，Year 类型结果长度设置问题 #7525
  - 修复 Prepare/Execute 模式下时间 0 值的处理问题 #7506
  - 解决整数类型除法实现中的错误处理问题 #7492
  - 解决 ComStmtSendLongData 处理过程中的兼容性问题 #7485
  - 解决字符串转为整数类型过程中的错误处理问题 #7483
  - 优化 information_schema.columns_in_table 表中的值精度 #7463
  - 修复使用 MariaDB 客户端对字符串类型数据的写入和更新的兼容性问题 #7573
  - 修复返回值别名的兼容性问题 #7600
  - 修复 information_schema.COLUMNS 表中浮点数的 NUMERIC_SCALE 值不正确的问题 #7602
  - 解决单行注释内容为空 Parser 报错的问题 #7612
- 表达式
  - 在 insert 函数中检查 max_allowed_packet 的值 #7528
  - 支持内建函数 json_contains #7443
  - 支持内建函数 json_contains_path #7596
  - 支持内建函数 encode/decode #7622
  - 修复一些时间相关的函数在某些情况下和 MySQL 行为不兼容的问题 #7636
  - 解决从字符串中解析时间类型数据的兼容性问题 #7654
  - 解决计算 DateTime 类型数据的默认值时没有考虑时区的问题 #7655
- DML
  - InsertOnDuplicateUpdate 语句设置正确的 last_insert_id #7534
  - 减少需要更新 auto_increment_id 计数器的情况 #7515
  - 优化 Duplicate Key 错误的报错信息 #7495
  - 修复 insert...select...on duplicate key update 问题 #7406
  - 支持 LOAD DATA IGNORE LINES 语句 #7576
- DDL
  - 在监控中增加 DDL Job 的类型和当前 Schema 版本的信息 #7472
  - 完成 Admin Restore Table 功能方案设计 #7383
  - 解决 Bit 类型的默认值超过 128 的问题 #7249
  - 解决 Bit 类型默认值不能为 NULL 的问题 #7604
  - 减少 DDL 队列中检查 CREATE TABLE/DATABASE 任务的时间间隔 #7608
  - 使用 ddl/owner/resign HTTP 接口释放 DDL Owner 并开启新一轮 Owner 选举 #7649
- TiKV Go Client
  - 支持 Seek 操作只获取 Key #7419
- Table Partition（实验性）
  - 解决无法使用 Bigint 类型列作为 Partition Key 的问题 #7520
  - 支持 Partitioned Table 添加索引过程中遇到问题回滚操作 #7437

16.21.24.2 PD

- 新特性
  - 支持 `GetAllStores`的接口 #1228
  - Simulator 添加评估调度的统计信息 #1218
- 功能改进
  - 优化 Down Store 的处理流程，尽快地补副本 #1222
  - 优化 Coordinator 的启动，减少重启 PD 带来的不必要调度 #1225
  - 优化内存使用，减少 heartbeat 带来的内存开销 #1195
  - 优化错误处理，完善日志信息 #1227
  - pd-ctl 支持查询指定 store 的 Region 信息 #1231
  - pd-ctl 支持查询按 version 比对的 topN 的 Region 信息 #1233
  - pd-ctl 支持更精确的 TSO 解码 #1242
- Bug 修复
  - 修复 pd-ctl 使用 hot store 命令错误退出的问题 #1244

16.21.24.3 TiKV

- 性能优化
  - 支持基于统计估算进行 Region split，减少 I/O 开销 #3511
  - 减少部分组件的内存拷贝 #3530
- 功能改进
  - 增加大量内建函数下推支持
  - 增加 `leader-transfer-max-log-lag` 配置解决特定场景 leader 调度失败的问题 #3507
  - 增加 `max-open-engines` 配置限制 `tikv-importer` 同时打开的 engine 个数 #3496
  - 限制垃圾数据的清理速度，减少对 `snapshot apply` 的影响 #3547
  - 对关键 Raft 消息广播 commit 信息，避免不必要的延迟 #3592
- Bug 修复
  - 修复新分裂 Region 的 PreVote 消息被丢弃导致的 leader 选举问题 #3557
  - 修复 Region merge 以后 follower 的相关统计信息 #3573
  - 修复 local reader 使用过期 Region 信息的问题 #3565

16.21.25 TiDB 2.1 RC1 Release Notes

2018 年 8 月 24 日，TiDB 发布 2.1 RC1 版。相比 2.1 Beta 版本，该版本对系统稳定性、优化器、统计信息以及执行引擎做了很多改进。

16.21.25.1 TiDB

- SQL 优化器
  - 修复某些情况下关联子查询去关联后结果不正确的问题 #6972
  - 优化 Explain 输出结果 #7011#7041
  - 优化 IndexJoin 驱动表选择策略#7019
  - 去掉非 PREPARE 语句的 Plan Cache #7040
  - 修复某些情况下 INSERT 语句无法正常解析执行的问题 #7068
  - 修复某些情况下 IndexJoin 结果不正确的问题 #7150
  - 修复某些情况下使用唯一索引不能查询到 NULL 值的问题 #7163
  - 修复 UTF-8 编码情况下前缀索引的范围计算不正确的问题 #7194
  - 修复某些情况下 Project 算子消除导致的结果不正确的问题 #7257
  - 修复主键为整数类型时无法使用 USE INDEX(PRIMARY) 的问题 #7316
  - 修复某些情况下使用关联列无法计算索引范围的问题 #7357
- SQL 执行引擎
  - 修复某些情况下夏令时时间计算结果不正确的问题 #6823
  - 重构聚合函数框架，提升 Stream 和 Hash 聚合算子的执行效率 #6852
  - 修复某些情况下 Hash 聚合算子不能正常退出的问题 #6982
  - 修复 BIT_AND/BIT_OR/BIT_XOR 没有正确处理非整型数据的问题 #6994
  - 优化 REPLACE INTO 语句的执行速度，性能提升近 10 倍 #7027
  - 优化时间类型的内存占用，时间类型数据的内存使用降低为原来的一半 #7043
  - 修复 UNION 语句整合有符号和无符号型整数结果时与 MySQL 不兼容的问题 #7112
  - 修复 LPAD/RPAD/TO_BASE64/FROM_BASE64/REPEAT 因为申请过多内存导致 TiDB panic 的问题 #7171 #7266 #7409 #7431
  - 修复 MergeJoin/IndexJoin 在处理 NULL 值时结果不正确的问题 #7255
  - 修复某些情况下 Outer Join 结果不正确的问题 #7288
  - 增强 Data Truncated 的报错信息，便于定位出错的数据和表中对应的字段 #7401
  - 修复某些情况下 Decimal 计算结果不正确的问题 #7001 #7113 #7202 #7208
  - 优化点查的查询性能 #6937
  - 禁用 Read Committed 隔离级别，避免潜在的问题 #7211
  - 修复某些情况下 LTRIM/RTRIM/TRIM 结果不正确的问题 #7291
  - 修复 MaxOneRow 算子无法保证返回结果不超过 1 行的问题 #7375
  - 拆分 range 个数过多的 Coprocessor 请求 #7454
- 统计信息
  - 优化统计信息动态收集机制 #6796
  - 解决数据频繁更新场景下 Auto Analyze 不工作的问题 #7022
  - 减少统计信息动态更新过程中的写入冲突 #7124
  - 优化统计信息不准确情况下的代价估算 #7175
  - 优化 AccessPath 的代价估算策略 #7233
- Server
  - 修复加载权限信息时的 bug #6976
  - 修复 Kill 命令对权限的检查过严问题 #6954

  - 解决 Binary 协议中某些数值类型移除的问题 #6922
  - 精简日志输出 #7029
  - 处理 mismatchClusterID 问题 #7053
  - 增加 advertise-address 配置项 #7078
  - 增加 GrpcKeepAlive 选项 #7100
  - 增加连接或者 Token 时间监控 #7110
  - 优化数据解码性能 #7149
  - INFORMMATION_SCHEMA 中增加 PROCESSLIST 表 #7236
  - 解决权限验证时多条规则可以命中情况下的顺序问题 #7211
  - 将部分编码相关的系统变量默认值改为 UTF-8 #7198
  - 慢查询日志显示更详细的信息 #7302
  - 支持在 PD 注册 tidb-server 的相关信息并通过 HTTP API 获取 #7082
- 兼容性
  - 支持 Session 变量 warning_count 和 error_count #6945
  - 读取系统变量时增加 Scope 检查 #6958
  - 支持 MAX_EXECUTION_TIME 语法 #7012
  - 支持更多的 SET 语法 #7020
  - Set 系统变量值过程中增加合法性校验 #7117
  - 增加 Prepare 语句中 PlaceHolder 数量的校验 #7162
  - 支持 set character_set_results = null #7353
  - 支持 flush status 语法 #7369
  - 修复 SET 和 ENUM 类型在 information_schema 里的 column size #7347
  - 支持建表语句里的 NATIONAL CHARACTER 语法 #7378
  - 支持 LOAD DATA 语句的 CHARACTER SET 语法 #7391
  - 修复 SET 和 ENUM 类型的 column info #7417
  - 支持 CREATE USER 语句的 IDENTIFIED WITH 语法 #7402
  - 修复 TIMESTAMP 类型计算过程中丢失精度的问题 #7418
  - 支持更多 SYSTEM 变量的合法性验证 #7196
  - 修复 CHAR_LENGTH 函数在计算 binary string 时结果不正确的问题 #7410
  - 修复在包含 GROUP BY 的语句里 CONCAT 结果不正确的问题 #7448
  - 修复 DECIMAL 类型 CAST 到 STRING 类型时，类型长度不准确的问题 #7451
- DML
  - 解决 Load Data 语句的稳定性 #6927
  - 解决一些 Batch 操作情况下的内存使用问题 #7086
  - 提升 Replace Into 语句的性能 #7027
  - 修复写入 CURRENT_TIMESTAMP 时，精度不一致的问题 #7355
- DDL
  - 改进 DDL 判断 Schema 是否已经同步的方法，避免某些情况下的误判 #7319
  - 修复在 ADD INDEX 过程中的 SHOW CREATE TABLE 结果 #6993
  - 非严格 sql-mode 模式下，text/blob/json 的默认值可以为空 #7230
  - 修复某些特定场景下 ADD INDEX 的问题 #7142
  - 大幅度提升添加 UNIQUE-KEY 索引操作的速度 #7132
  - 修复 Prefix-index 在 UTF-8 字符集的场景下的截断问题 #7109

  - 增加环境变量 `tidb_ddl_reorg_priority` 来控制 add-index 操作的优先级 #7116
  - 修复 `information_schema.tables` 中 AUTO-INCREMENT 的显示问题 #7037
  - 支持 `admin show ddl jobs <number>` 命令，支持输出 number 个 DDL jobs #7028
  - 支持并行 DDL 任务执行 #6955
- Table Partition（实验性）
  - 支持一级分区
  - 支持 `Range Partition`

16.21.25.2 PD

- 新特性
  - 引入版本控制机制，支持集群滚动兼容升级
  - 开启 `Region merge` 功能
  - 支持 `GetPrevRegion` 接口
  - 支持批量 `split Region`
  - 支持存储 GC safepoint
- 功能改进
  - 优化系统时间回退影响 TSO 分配的问题
  - 优化处理 Region heartbeat 的性能
  - 优化 Region tree 性能
  - 优化计算热点统计的性能问题
  - 优化 API 接口错误码返回
  - 新增一些控制调度策略的开关
  - 禁止在 `label` 中使用特殊字符
  - 完善调度模拟器
  - pd-ctl 支持使用统计信息进行 Region split
  - pd-ctl 支持调用 `jq` 来格式化 JSON 输出
  - 新增 etcd Raft 状态机相关 metrics
- Bug 修复
  - 修复 leader 切换后 namespace 未重新加载的问题
  - 修复 namespace 调度超出 schedule limit 配置的问题
  - 修复热点调度超出 schedule limit 的问题
  - 修复 PD client 关闭时输出一些错误日志的问题
  - 修复 Region 心跳延迟统计有误的问题

16.21.25.3 TiKV

- 新特性
  - 支持 `batch split`，防止热点 Region 写入产生超大 Region
  - 支持设置根据数据行数 split Region，提升 index scan 效率
- 性能优化

  - 使用 LocalReader 将 Read 操作从 raftstore 线程分离，减少 Read 延迟
  - 重构 MVCC 框架，优化 memory 使用，提升 scan read 性能
  - 支持基于统计估算进行 Region split，减少 I/O 开销
  - 优化连续写入 Rollback 记录后影响读性能的问题
  - 减少下推聚合计算的内存开销

- 功能改进

  - 增加大量内建函数下推支持，更完善的 charset 支持
  - 优化 GC 流程，提升 GC 速度并降低 GC 对系统的影响
  - 开启 prevote，加快网络异常时的恢复服务速度
  - 增加 RocksDB 日志文件相关的配置项
  - 调整 scheduler latch 默认配置
  - 使用 tikv-ctl 手动 compact 时可设定是否 compact RocksDB 最底层数据
  - 增加启动时的环境变量检查
  - 支持基于已有数据动态设置 dynamic_level_bytes 参数
  - 支持自定义日志格式
  - tikv-ctl 整合 tikv-fail 工具
  - 增加 threads IO metrics

- Bug 修复

  - 修复 decimal 相关问题
  - 修复 gRPC max_send_message_len 设置有误的问题
  - 修复 region_size 配置不当时产生的问题

16.21.26 TiDB 2.1 Beta Release Notes

2018 年 6 月 29 日，TiDB 发布 2.1 Beta 版。相比 2.0 版本，该版本对系统稳定性、优化器、统计信息以及执行引擎做了很多改进。

16.21.26.1 TiDB

- SQL 优化器

  - 优化 Index Join 选择范围，提升执行性能
  - 优化关联子查询，下推 Filter 和扩大索引选择范围，部分查询的效率有数量级的提升
  - 在 UPDATE、DELETE 语句中支持 Index Hint 和 Join Hint
  - 优化器 Hint TIDM_SMJ 在没有索引可用的情况下可生效
  - 支持更多函数下推：ABS/CEIL/FLOOR/IS TRUE/IS FALSE
  - 在常量折叠过程中特殊处理函数 IF 和 IFNULL
  - 优化 EXPLAIN 语句输出格式

- SQL 执行引擎

  - 实现并行 Hash Aggregate 算子，部分场景下能提高 Hash Aggregate 计算性能 350%
  - 实现并行 Project 算子，部分场景下性能提升达 74%
  - 并发地读取 Hash Join 的 Inner 表和 Outer 表的数据，提升执行性能
  - 修复部分场景下 INSERT ⋯ ON DUPLICATE KEY UPDATE ⋯ 结果不正确的问题

  - 修复 CONCAT_WS/FLOOR/CEIL/DIV 内建函数的结果不正确的问题

- Server
  - 添加 HTTP API 打散 table 的 Regions 在 TiKV 集群中的分布
  - 添加 auto_analyze_ratio 系统变量控制自动 analyze 的阈值
  - 添加 HTTP API 控制是否打开 general log
  - 添加 HTTP API 在线修改日志级别
  - 在 general log 和 slow query log 中添加 user 信息
  - 支持 Server side cursor
- 兼容性
  - 支持更多 MySQL 语法
  - BIT 聚合函数支持 ALL 参数
  - 支持 SHOW PRIVILEGES 语句
- DML
  - 减少 INSERT INTO SELECT 语句的内存占用
  - 修复 Plan Cache 的性能问题
  - 添加 tidb_retry_limit 系统变量控制事务自动重试的次数
  - 添加 tidb_disable_txn_auto_retry 系统变量控制事务是否自动重试
  - 修复写入 time 类型的数据精度问题
  - 支持本地冲突事务排队，优化冲突事务性能
  - 修复 UPDATE 语句的 Affected Rows
  - 优化 insert ignore on duplicate key update 语句性能
  - 优化 Create Table 语句的执行速度
  - 优化 Add index 的速度，在某些场景下速度大幅提升
  - 修复 Alter table add column 增加列超过表的列数限制的问题
  - 修复在某些异常情况下 DDL 任务重试导致 TiKV 压力增加的问题
  - 修复在某些异常情况下 TiDB 不断重载 Schema 信息的问题
- DDL
  - Show Create Table 不再输出外键相关的内容
  - 支持 select tidb_is_ddl_owner() 语句，方便判断 TiDB 是否为 DDL Owner
  - 修复某些场景下 YEAR 类型删除索引的问题
  - 修复并发执行场景下的 Rename table 的问题
  - 支持 ALTER TABLE FORCE 语法
  - 支持 ALTER TABLE RENAME KEY TO 语法
  - admin show ddl jobs 输出信息中添加表名、库名等信息

16.21.26.2 PD

- PD 节点间开启 Raft PreVote，避免网络隔离后恢复时产生的重新选举
- 优化 Balance Scheduler 频繁调度小 Region 的问题
- 优化热点调度器，在流量统计信息抖动时适应性更好
- region merge 调度时跳过数据行数较多的 Region
- 默认开启 raft learner 功能，降低调度时出现宕机导致数据不可用的风险

- pd-recover 移除 max-replica 参数
- 增加 Filter 相关的 metrics
- 修复 tikv-ctl unsafe recovery 之后 Region 信息没更新的问题
- 修复某些场景下副本迁移导致 TiKV 磁盘空间耗尽的问题
- 兼容性提示
  - 由于新版本存储引擎更新，不支持在升级后回退至 2.0.x 或更旧版本
  - 新版本默认开启 raft learner 功能，如果从 1.x 版本集群升级至 2.1 版本，须停机升级或者先滚动升级 TiKV，完成后再滚动升级 PD

16.21.26.3 TiKV

- 升级 Rust 到 nightly-2018-06-14 版本
- 开启 PreVote，避免网络隔离后恢复时产生的重新选举
- 添加 metric，显示 RocksDB 内部每层的文件数和 ingest 相关的信息
- GC 运行时打印版本太多的 key
- 使用 static metric 优化多 label metric 性能（YCSB raw get 提升 3%）
- 去掉多个模块的 box，使用范型提升运行时性能（YCSB raw get 提升 3%）
- 使用 asynchronous log 提升写日志性能
- 增加收集线程状态的 metric
- 通过减少程序中 box 的使用来减少内存拷贝的次数，提升性能

16.22 v2.0

16.22.1 TiDB 2.0.11 Release Notes

2019 年 1 月 3 日，TiDB 发布 2.0.11 版，TiDB Ansible 相应发布 2.0.11 版本。该版本在 2.0.10 版的基础上，对系统兼容性、稳定性做出了改进。

16.22.1.1 TiDB

- 修复 PD 发生异常的情况下，Error 没有被正确处理的问题 #8764
- 修复 Rename 相同表的行为，跟 MySQL 保持一致 #8809
- 修复 ADMIN CHECK TABLE 在 ADD INDEX 过程中误报的问题 #8750
- 修复前缀索引在某些情况下，开闭范围区间错误的问题 #8877
- 修复在某些添加列的情况下，UPDATE 语句 panic 的问题 #8904

16.22.1.2 TiKV

- 修复了两个 Region merge 相关的问题 #4003 #4004

16.22.2 TiDB 2.0.10 Release Notes

2018 年 12 月 18 日，TiDB 发布 2.0.10 版，TiDB Ansible 相应发布 2.0.10 版本。该版本在 2.0.9 版的基础上，对系统兼容性、稳定性做出了改进。

16.22.2.1 TiDB

- 修复取消 DDL 任务的时候可能导致的问题 #8513
- 修复 ORDER BY，UNION 语句无法引用带表名的列的问题 #8514
- 修复 UNCOMPRESS 函数没有判断错误输入长度的问题 #8607
- 修复 ANSI_QUOTES SQL_MODE 在 TiDB 升级的时候遇到的问题 #8575
- 修复某些情况下 select 返回结果错误的问题 #8570
- 修复 TiDB 在收到退出信号的时候可能无法退出的问题 #8501
- 修复某些情况下 IndexLookUpJoin 返回错误结果的问题 #8508
- 避免下推有 GetVar 或 SetVar 的 filter #8454
- 修复某些情况下 UNION 语句结果长度错误的问题 #8491
- 修复 PREPARE FROM @var_name 的问题 #8488
- 修复某些情况下导出统计信息 panic 的问题 #8464
- 修复统计信息某些情况下对点查估算的问题 #8493
- 修复某些情况下返回 Enum 默认值为字符串导致的 panic #8476
- 修复在宽表场景下，占用太多内存的问题 #8467
- 修复 Parser 对取模操作错误格式化导致的问题 #8431
- 修复某些情况下添加外键约束导致的 panic 问题 #8421，#8410
- 修复 YEAR 类型错误转换零值的问题 #8396
- 修复 VALUES 函数在参数不为列的时候 panic 的问题 #8404
- 存在子查询的语句禁用 Plan Cache #8395

16.22.2.2 PD

- 修复 RaftCluster 在退出时可能的死锁问题 #1370

16.22.2.3 TiKV

- 修复迁移 Leader 到新节点时造成请求延时问题 #3929
- 修复多余的 Region 心跳 #3930

16.22.3 TiDB 2.0.9 Release Notes

2018 年 11 月 19 日，TiDB 发布 2.0.9 版。该版本在 2.0.8 版的基础上，对系统兼容性、稳定性做出了改进。

16.22.3.1 TiDB

- 修复统计信息直方图为空的时候导致的问题 #7927
- 修复 UNION ALL 语句在某些情况下 panic 的问题 #7942
- 修复错误的 DDL JOB 情况下导致的递归溢出问题 #7959
- 为 Commit 操作加上慢操作日志 #7983
- 修复 Limit 值太大的情况下导致的 panic 问题 #8004
- 支持 USING 子句指定 utf8mb4 字符集 #8048
- 内建函数 TRUNCATE 支持类型为 unsigned int 的参数 #8069

- 修复统计信息模块在某些情况下主键选择率估算的问题 #8150
- 增加 Session 变量来控制是否允许写入 _tidb_rowid #8126
- 修复 PhysicalProjection 在某些情况下 panic 的问题 #8154
- 修复 Union 语句在某些情况下结果不稳定的问题 #8168
- 修复在非插入语句下 values 没有返回 NULL 的问题 #8179
- 修复某些情况下统计信息模块无法清除过期统计数据的问题 #8184
- 让事务允许的最长运行时间变成一个可配置项 #8209
- 修复 expression rewriter 某些情况下错误的比较逻辑 #8288
- 消除 UNION ORDER BY 语句生成的多余列的问题 #8307
- 支持 admin show next_row_id 语句 #8274
- 修复 Show Create Table 语句中特殊字符转义的问题 #8321
- 修复 UNION 语句在某些情况下遇到非预期错误的问题 #8318
- 修复某些情况下取消 DDL 任务导致的 Schema 没有回滚的问题 #8312
- 把变量 tidb_max_chunk_size 变成全局环境变量 #8333
- ticlient Scan 命令增加边界，解决数据扫出边界的问题 #8309 #8310

16.22.3.2 PD

- 修复 etcd 启动失败导致的服务挂起问题 #1267
- 修复 pd-ctl 读取 Region key 的相关问题 #1298 #1299 #1308
- 修复 regions/check API 输出错误的问题 #1311
- 修复 PD join 失败后无法重新 join 的问题 #1279

16.22.3.3 TiKV

- 增加 kv_scan 接口扫描上界的限制 #3749
- 废弃配置 max-tasks-xxx 并新增 max-tasks-per-worker-xxx #3093
- 修复 RocksDB CompactFiles 的问题 #3789

16.22.4 TiDB 2.0.8 Release Notes

2018 年 10 月 16 日，TiDB 发布 2.0.8 版。该版本在 2.0.7 版的基础上，对系统兼容性、稳定性做出了改进。

16.22.4.1 TiDB

- 功能改进
  - 在 Update 没有更改相应 AUTO-INCREMENT 列情况下，防止 AUTO-ID 被消耗过快 #7846
- Bug 修复
  - 在 PD Leader 异常宕机的情况下，TiDB 快速创建 etcd Session 恢复服务 #7810
  - 修复 DateTime 类型使用默认值时候没有考虑时区的问题 #7672
  - 修复 duplicate key update 在某些情况下没有正确插入值的问题 #7685
  - 修复 UnionScan 中谓词条件没有下推的问题 #7726

- 修复增加 TIMESTAMP 索引没有正确处理时区的问题 #7812
- 修复某些情况下统计信息模块导致的内存泄露问题 #7864
- 修复在某些异常情况下，无法获得 ANALYZE 结果的问题 #7871
- 令 SYSDATE 不做表达式展开，以返回正确的结果 #7894
- 修复某些情况下，substring_index panic 的问题 #7896
- 修复某些情况下，错误将 OUTER JOIN 转为 INNER JOIN 的问题 #7899

16.22.4.2 TiKV

- Bug 修复
  - 修复节点宕机时 Raftstore EntryCache 占用内存持续上升的问题 #3529

16.22.5 TiDB 2.0.7 Release Notes

2018 年 9 月 7 日，TiDB 发布 2.0.7 版。该版本在 2.0.6 版的基础上，对系统兼容性、稳定性做出了改进。

16.22.5.1 TiDB

- New Feature
  - 在 information_schema 里添加 PROCESSLIST 表 #7286
- Improvements
  - 收集更多语句执行细节，并输出在 SLOW QUERY 日志里 #7364
  - SHOW CREATE TABLE 不再输出分区信息 #7388
  - 通过设置 RC 隔离级别和低优先级优化 ANALYZE 语句执行效率 #7500
  - 加速 ADD UNIQUE INDEX #7562
  - 增加控制 DDL 并发度的选项 #7563
- Bug Fixes
  - 修复 PRIMARY KEY 为整数的表，无法使用 USE INDEX(PRIMARY) 的问题 #7298
  - 修复 Merge Join 和 Index Join 在 inner row 为 NULL 时输出多余结果的问题 #7301
  - 修复 chunk size 设置过小时，Join 输出多余结果的问题 #7315
  - 修复建表语句中包含 range column 语法导致 panic 的问题 #7379
  - 修复 admin check table 对时间类型的列误报的问题 #7457
  - 修复以默认值 current_timestamp 插入的数据无法用 = 条件查询到的问题 #7467
  - 修复以 ComStmtSendLongData 命令插入空字符串参数被误解析为 NULL 的问题 #7508
  - 修复特定场景下 auto analyze 不断重复执行的问题 #7556
  - 修复 parser 无法解析以换行符结尾的单行注释的问题 #7635

16.22.5.2 TiKV

- Improvement
  - 空集群默认打开 dynamic-level-bytes 参数减少空间放大
- Bug Fix
  - 在 Region merge 之后更新 Region 的 approximate size 和 keys

16.22.6 TiDB 2.0.6 Release Notes

2018 年 8 月 6 日，TiDB 发布 2.0.6 版。该版本在 2.0.5 版的基础上，对系统兼容性、稳定性做出了改进。

16.22.6.1 TiDB

- Improvements
  - 精简 “set system variable” 日志的长度，减少日志文件体积 #7031
  - 在日志中记录 ADD INDEX 执行过程中的慢操作，便于定位问题 #7083
  - 减少更新统计信息操作中的事务冲突 #7138
  - 当待估算的值超过统计信息范围时，提高行数估计的准确度 #7185
  - 当使用 Index Join 时，选择行数估计较小的表作为驱动表，提高 Index Join 的执行效率 #7227
  - 为 ANALYZE TABLE 语句执行过程中发生的 panic 添加 recover 机制，避免收集统计信息过程中的异常行为导致 tidb-server 不可用 #7228
  - 当 RPAD/LPAD 的结果超过设置系统变量 max_allowed_packet 时，返回 NULL 和对应的 warning，兼容 MySQL #7244
  - 设置 PREPARE 语句中占位符数量上限为 65535，兼容 MySQL #7250
- Bug Fixes
  - 修复某些情况下，DROP USER 语句和 MySQL 行为不兼容的问题 #7014
  - 修复当 tidb_batch_insert 打开后，INSERT/LOAD DATA 等语句在某些场景下 OOM 的问题 #7092
  - 修复某个表的数据持续更新时，其统计信息自动更新失效的问题 #7093
  - 修复防火墙断掉不活跃的 gRPC 连接的问题 #7099
  - 修复某些场景下使用前缀索引结果不正确的问题 #7126
  - 修复某些场景下统计信息过时导致 panic 的问题 #7155
  - 修复某些场景下 ADD INDEX 后索引数据少一条的问题 #7156
  - 修复某些场景下查询唯一索引上的 NULL 值结果不正确的问题 #7172
  - 修复某些场景下 DECIMAL 的乘法结果出现乱码的问题 #7212
  - 修复某些场景下 DECIMAL 的取模运算结果不正确的问题 #7245
  - 修复某些特殊语句序列下在事务中执行 UPDATE/DELETE 语句后结果不正确的问题 #7219
  - 修复某些场景下 UNION ALL/UPDATE 语句在构造执行计划过程中 panic 的问题 #7225
  - 修复某些场景下前缀索引的索引范围计算错误的问题 #7231
  - 修复某些场景下 LOAD DATA 语句不写 binlog 的问题 #7242
  - 修复某些场景下在 ADD INDEX 过程中 SHOW CREATE TABLE 结果不正确的问题 #7243
  - 修复某些场景下 Index Join 因为没有初始化事务时间戳而 panic 的问题 #7246
  - 修复 ADMIN CHECK TABLE 因为误用 session 中的时区而导致误报的问题 #7258
  - 修复 ADMIN CLEANUP INDEX 在某些场景下索引没有清除干净的问题 #7265
  - 禁用 Read Committed 事务隔离级别 #7282

16.22.6.2 TiKV

- Improvements
  - 扩大默认 scheduler slots 值以减少假冲突现象
  - 减少回滚事务的连续标记以提升冲突极端严重下的读性能

  - 限制 RocksDB log 文件的大小和个数以减少长时间运行下不必要的磁盘占用

- Bug Fixes

  - 修复字符串转 Decimal 时出现的 crash

16.22.7 TiDB 2.0.5 Release Notes

2018 年 7 月 6 日，TiDB 发布 2.0.5 版。该版本在 2.0.4 版的基础上，对系统兼容性、稳定性做出了改进。

16.22.7.1 TiDB

- New Features

  - 增加一个系统变量 `tidb_disable_txn_auto_retry`，用于关闭事务自动重试 #6877

- Improvements

  - 调整计算 `Selection` 代价的方式，结果更准确 #6989
  - 查询条件能够完全匹配唯一索引或者主键时，直接选择作为查询路径 #6966
  - 启动服务失败时，做必要的清理工作 #6964
  - 在 `Load Data` 语句中，将 `\N` 处理为 NULL #6962
  - 优化 CBO 代码结构 #6953
  - 启动服务时，尽早上报监控数据 #6931
  - 对慢查询日志格式进行优化：去除 SQL 语句中的换行符，增加用户信息 #6920
  - 支持注释中存在多个星号的情况 #6858

- Bug Fixes

  - 修复 `KILL QUERY` 语句权限检查问题 #7003
  - 修复用户数量超过 1024 时可能造成无法登录的问题 #6986
  - 修复一个写入无符号类型 `float/double` 数据的问题 #6940
  - 修复 `COM_FIELD_LIST` 命令的兼容性，解决部分 MariaDB 客户端遇到 Panic 的问题 #6929
  - 修复 `CREATE TABLE IF NOT EXISTS LIKE` 行为 #6928
  - 修复一个 TopN 下推过程中的问题 #6923
  - 修复 `Add Index` 过程中遇到错误时当前处理的行 ID 记录问题 #6903

16.22.7.2 PD

- 修复某些场景下副本迁移导致 TiKV 磁盘空间耗尽的问题
- 修复 `AdjacentRegionScheduler` 导致的崩溃问题

16.22.7.3 TiKV

- 修复 decimal 运算中潜在的溢出问题
- 修复 merge 过程中可能发生的脏读问题

16.22.8 TiDB 2.0.4 Release Notes

2018 年 6 月 15 日，TiDB 发布 2.0.4 版。该版本在 2.0.3 版的基础上，对系统兼容性、稳定性做出了改进。

16.22.8.1 TiDB

- 支持 `ALTER TABLE t DROP COLUMN a CASCADE` 语法
- 支持设置 `tidb_snapshot` 变量的值为 TSO
- 优化监控项中语句类型展示
- 优化查询代价估计精度
- 设置 gRPC 的 `backoff max delay` 参数
- 支持通过配置文件设置单条语句的内存使用阈值
- 重构 Optimizer 的 error
- 解决 Cast Decimal 数据的副作用问题
- 解决特定场景下 `Merge Join` 算子结果错误的问题
- 解决转换 `Null` 对象到 String 的问题
- 解决 Cast JSON 数据为 JSON 类型的问题
- 解决 `Union + OrderBy` 情况下结果顺序和 MySQL 不一致的问题
- 解决 `Union` 语句中对 `Limit/OrderBy` 子句的合法性检查规则问题
- 解决 `Union All` 的结果兼容性问题
- 解决谓词下推中的一个 Bug
- 解决 `Union` 语句对 `For Update` 子句的兼容性问题
- 解决 `concat_ws` 函数对结果错误截断的问题

16.22.8.2 PD

- 改进 `max-pending-peer-count` 调度参数未设置时的行为，调整为不限制最大 `PendingPeer` 的数量

16.22.8.3 TiKV

- 新增 RocksDB `PerfContext` 接口用于调试
- 移除 `import-mode` 参数
- 为 `tikv-ctl` 添加 `region-properties` 命令
- 优化有大量 RocksDB tombstone 时 `reverse-seek` 过慢的问题
- 修复 `do_sub` 导致的崩溃问题
- 当 GC 遇到有太多版本的数据时记录日志

16.22.9 TiDB 2.0.3 Release Notes

2018 年 6 月 1 日，TiDB 发布 2.0.3 版。该版本在 2.0.2 版的基础上，对系统兼容性、稳定性做出了改进。

16.22.9.1 TiDB

- 支持在线更改日志级别
- 支持 COM_CHANGE_USER 命令
- 支持二进制协议情况下使用时间类型参数
- 优化带 BETWEEN 表达式的查询条件代价估算
- 在 SHOW CREATE TABLE 里不显示 FOREIGN KEY 信息
- 优化带 LIMIT 子句的查询代价估算
- 修复 YEAR 类型作为唯一索引的问题
- 修复在没有唯一索引的情况下 ON DUPLICATE KEY UPDATE 的问题
- 修复 CEIL 函数的兼容性问题
- 修复 DECIMAL 类型计算 DIV 的精度问题
- 修复 ADMIN CHECK TABLE 误报的问题
- 修复 MAX/MIN 在特定表达式参数下 panic 的问题
- 修复特殊情况下 JOIN 结果为空的问题
- 修复 IN 表达式构造查询 Range 的问题
- 修复使用 Prepare 方式进行查询且启用 Plan Cache 情况下的 Range 计算问题
- 修复异常情况下频繁加载 Schema 信息的问题

16.22.9.2 PD

- 修复在特定条件下收集 hot-cache metrics 会 panic 的问题
- 修复对旧的 Region 产生调度的问题

16.22.9.3 TiKV

- 修复 learner flag 错误上报给 PD 的 bug
- 在 do_div_mod 中 divisor/dividend 为 0 时返回错误

16.22.10 TiDB 2.0.2 Release Notes

2018 年 5 月 21 日，TiDB 发布 2.0.2 版。该版本在 2.0.1 版的基础上，对系统稳定性做出了改进。

16.22.10.1 TiDB

- 修复 Decimal 除法内置函数下推的问题
- 支持 Delete 语句中使用 USE INDEX 的语法
- 禁止在带有 Auto-Increment 的列中使用 shard_row_id_bits 特性
- 增加写入 Binlog 的超时机制

16.22.10.2 PD

- 使 balance leader scheduler 过滤失连节点

- 更改 transfer leader operator 的超时时间为 10 秒
- 修复 label scheduler 在集群 Regions 不健康状态下不调度的问题
- 修复 evict leader scheduler 调度不当的问题

16.22.10.3 TiKV

- 修复 Raft 日志没有打出来的问题
- 支持配置更多 gRPC 相关参数
- 支持配置选举超时的取值范围
- 修复过期 learner 没有删掉的问题
- 修复 snapshot 中间文件被误删的问题

16.22.11 TiDB 2.0.1 Release Notes

2018 年 5 月 16 日，TiDB 发布 2.0.1 版。该版本在 2.0.0 (GA) 版的基础上，对 MySQL 兼容性、系统稳定性做出了改进。

16.22.11.1 TiDB

- 实时更新 `Add Index` 的进度到 DDL 任务信息中
- 添加 Session 变量 `tidb_auto_analyze_ratio` 控制统计信息自动更新阈值
- 修复当事务提交失败时可能未清理所有的残留状态的问题
- 修复加索引在部分情况下的 Bug
- 修复 DDL 修改表面操作在某些并发场景下的正确性问题
- 修复某些情况下 `LIMIT` 结果不正确的问题
- 修复 `ADMIN CHECK INDEX` 语句索引名字区分大小写问题
- 修复 `UNION` 语句的兼容性问题
- 修复插入 `TIME` 类型数据的兼容性问题
- 修复某些情况下 `copIteratorTaskSender` 导致的 goroutine 泄漏问题
- 增加一个选项，用于设置 TiDB 在写 Binlog 失败的情况下的行为
- 优化 Coprocessor 慢请求日志格式，区分处理时间长与排队时间长的任务
- MySQL 协议握手阶段发生错误不打印日志，避免 KeepAlive 造成大量日志
- 优化 `Out of range value for column` 的错误信息
- 修复 `Update` 语句中遇到子查询导致结果错误的问题
- 调整 TiDB 进程处理 `SIGTERM` 的行为，不等待正在执行的 Query 完成

16.22.11.2 PD

- 添加 `Scatter Range` 调度，调度指定 Key Range 包含的 Region
- 优化 `Merge Region` 调度，使新分裂不久的 Region 不能被合并
- 添加 learner 相关的 metrics
- 修复重启误删 scheduler 的问题
- 修复解析配置文件出错问题
- 修复 etcd leader 和 PD leader 不同步的问题

- 修复关闭 learner 情况下还有 learner 出现的问题
- 修复读取包过大造成 load Regions 失败的问题

16.22.11.3 TiKV

- 修复 `SELECT FOR UPDATE` 阻止其他人读的问题
- 优化慢查询的日志
- 减少 `thread_yield` 的调用次数
- 修复生成 snapshot 会意外阻塞 raftstore 的 bug
- 修复特殊情况下开启 learner 无法选举成功的问题
- 修复极端情况下分裂可能导致的脏读问题
- 修正读线程池的配置默认值
- 修正删大数据表会影响写性能的问题

16.22.12 TiDB 2.0 Release Notes

2018 年 4 月 27 日，TiDB 发布 2.0 GA 版。相比 1.0 版本，该版本对 MySQL 兼容性、系统稳定性、优化器和执行器做了很多改进。

16.22.12.1 TiDB

- SQL 优化器
  - 精简统计信息数据结构，减小内存占用
  - 加快进程启动时加载统计信息速度
  - 支持统计信息动态更新experimental
  - 优化代价模型，对代价估算更精准
  - 使用 `Count-Min Sketch` 更精确地估算点查的代价
  - 支持分析更复杂的条件，尽可能充分的使用索引
  - 支持通过 `STRAIGHT_JOIN` 语法手动指定 Join 顺序
  - `GROUP BY`子句为空时使用 Stream Aggregation 算子，提升性能
  - 支持使用索引计算 `Max/Min` 函数
  - 优化关联子查询处理算法，支持将更多类型的关联子查询解关联并转化成 `Left Outer Join`
  - 扩大 `IndexLookupJoin` 的使用范围，索引前缀匹配的场景也可以使用该算法
- SQL 执行引擎
  - 使用 Chunk 结构重构所有执行器算子，提升分析型语句执行性能，减少内存占用，显著提升 TPC-H 结果
  - 支持 Streaming Aggregation 算子下推
  - 优化 `Insert Into Ignore` 语句性能，提升 10 倍以上
  - 优化 `Insert On Duplicate Key Update` 语句性能，提升 10 倍以上
  - 下推更多的数据类型和函数到 TiKV 计算
  - 优化 `Load Data` 性能，提升 10 倍以上
  - 支持对物理算子内存使用进行统计，通过配置文件以及系统变量指定超过阈值后的处理行为
  - 支持限制单条 SQL 语句使用内存的大小，减少程序 OOM 风险

  - 支持在 CRUD 操作中使用隐式的行 ID
  - 提升点查性能

- Server
  - 支持 Proxy Protocol
  - 添加大量监控项，优化日志
  - 支持配置文件的合法性检测
  - 支持 HTTP API 获取 TiDB 参数信息
  - 使用 Batch 方式 Resolve Lock，提升垃圾回收速度
  - 支持多线程垃圾回收
  - 支持 TLS

- 兼容性
  - 支持更多 MySQL 语法
  - 支持配置文件修改 `lower_case_table_names` 系统变量，用于支持 OGG 数据同步工具
  - 提升对 Navicat 的兼容性
  - 在 `Information_Schema` 中支持显示建表时间
  - 修复部分函数/表达式返回类型和 MySQL 不同的问题
  - 提升对 JDBC 兼容性
  - 支持更多的 `SQL_MODE`

- DDL
  - 优化 `Add Index` 的执行速度，部分场景下速度大幅度提升
  - `Add Index` 操作变更为低优先级，降低对线上业务影响
  - `Admin Show DDL Jobs` 输出更详细的 DDL 任务状态信息
  - 支持 `Admin Show DDL Job Queries JobID` 查询当前正在运行的 DDL 任务的原始语句
  - 支持 `Admin Recover Index` 命令，用于灾难恢复情况下修复索引数据
  - 支持通过 `Alter` 语句修改 Table Options

16.22.12.2 PD

- 增加 `Region Merge` 支持，合并数据删除后产生的空 Region experimental
- 增加 `Raft Learner` 支持experimental
- 调度器优化
  - 调度器适应不同的 Region size
  - 提升 TiKV 宕机时数据恢复的优先级和恢复速度
  - 提升下线 TiKV 节点搬迁数据的速度
  - 优化 TiKV 节点空间不足时的调度策略，尽可能防止空间不足时磁盘被写满
  - 提升 balance-leader scheduler 的调度效率
  - 减少 balance-region scheduler 调度开销
  - 优化 hot-region scheduler 的执行效率
- 运维接口及配置
  - 增加 TLS 支持
  - 支持设置 PD leader 优先级

  - 支持基于 label 配置属性
  - 支持配置特定 label 的节点不调度 Region leader
  - 支持手动 Split Region，可用于处理单 Region 热点的问题
  - 支持打散指定 Region，用于某些情况下手动调整热点 Region 分布
  - 增加配置参数检查规则，完善配置项的合法性较验
- 调试接口
  - 增加 `Drop Region` 调试接口
  - 增加枚举各个 PD health 状态的接口
- 统计相关
  - 添加异常 Region 的统计
  - 添加 Region 隔离级别的统计
  - 添加调度相关 metrics
- 性能优化
  - PD leader 尽量与 etcd leader 保持同步，提升写入性能
  - 优化 Region heartbeat 性能，现可支持超过 100 万 Region

16.22.12.3 TiKV

- 功能
  - 保护关键配置，防止错误修改
  - 支持 `Region Merge` experimental
  - 添加 `Raw DeleteRange` API
  - 添加 `GetMetric` API
  - 添加 `Raw Batch Put`，`Raw Batch Get`，`Raw Batch Delete` 和 `Raw Batch Scan`
  - 给 Raw KV API 增加 Column Family 参数，能对特定 Column Family 进行操作
  - Coprocessor 支持 streaming 模式，支持 streaming 聚合
  - 支持配置 Coprocessor 请求的超时时间
  - 心跳包携带时间戳
  - 支持在线修改 RocksDB 的一些参数，包括 `block-cache-size` 大小等
  - 支持配置 Coprocessor 遇到某些错误时的行为
  - 支持以导数据模式启动，减少导数据过程中的写放大
  - 支持手动对 region 进行对半 split
  - 完善数据修复工具 tikv-ctl
  - Coprocessor 返回更多的统计信息，以便指导 TiDB 的行为
  - 支持 ImportSST API，可以用于 SST 文件导入experimental
  - 新增 TiKV Importer 二进制，与 TiDB Lightning 集成用于快速导入数据experimental
- 性能
  - 使用 ReadPool 优化读性能，`raw_get/get/batch_get` 提升 30%
  - 提升 metrics 的性能
  - Raft snapshot 处理完之后立即通知 PD，加快调度速度
  - 解决 RocksDB 刷盘导致性能抖动问题

  - 提升在数据删除之后的空间回收
  - 加速启动过程中的垃圾清理过程
  - 使用 DeleteFilesInRanges 减少副本迁移时 I/O 开销
- 稳定性
  - 解决在 PD leader 发送切换的情况下 gRPC call 不返回问题
  - 解决由于 snapshot 导致下线节点慢的问题
  - 限制搬移副本临时占用的空间大小
  - 如果有 Region 长时间没有 Leader，进行上报
  - 根据 compaction 事件及时更新统计的 Region size
  - 限制单次 scan lock 请求的扫描的数据量，防止超时
  - 限制接收 snapshot 过程中的内存占用，防止 OOM
  - 提升 CI test 的速度
  - 解决由于 snapshot 太多导致的 OOM 问题
  - 配置 gRPC 的 keepalive 参数
  - 修复 Region 增多容易 OOM 的问题

16.22.12.4 TiSpark

TiSpark 使用独立的版本号，现为 1.0 GA。TiSpark 1.0 版本组件提供了针对 TiDB 上的数据使用 Apache Spark 进行分布式计算的能力。

- 提供了针对 TiKV 读取的 gRPC 通信框架
- 提供了对 TiKV 组件数据的和通信协议部分的编码解码
- 提供了计算下推功能，包含：
  - 聚合下推
  - 谓词下推
  - TopN 下推
  - Limit 下推
- 提供了索引相关的支持
  - 谓词转化聚簇索引范围
  - 谓词转化次级索引
  - Index Only 查询优化
  - 运行时索引退化扫表优化
- 提供了基于代价的优化
  - 统计信息支持
  - 索引选择
  - 广播表代价估算
- 多种 Spark Interface 的支持
  - Spark Shell 支持
  - ThriftServer/JDBC 支持
  - Spark-SQL 交互支持
  - PySpark Shell 支持
  - SparkR 支持

16.22.13 TiDB 2.0 RC5 Release Notes

2018 年 4 月 17 日，TiDB 发布 2.0 RC5 版。该版本在 RC4 版的基础上，对 MySQL 兼容性、系统稳定性和优化器做了很多改进。

16.22.13.1 TiDB

- 修复应用 Top-N 下推规则的问题
- 修复对包含 NULL 值的列的行数估算
- 修复 Binary 类型的 0 值
- 修复事务内的 BatchGet 问题
- 回滚 Add Index 操作的时候，清除清除已写入的数据，减少空间占用
- 优化 insert on duplicate key update 语句性能，提升 10 倍以上
- 修复 UNIX_TIMESTAMP 函数返回结果类型问题返回结果类型问题
- 修复在添加 NOT NULL 列的过程中，插入 NULL 值的问题
- Show Process List 语句支持显示执行语句的内存占用
- 修复极端情况下 Alter Table Modify Column 出错问题
- 支持通过 Alter 语句设置 table comment

16.22.13.2 PD

- 添加 Raft Learner 支持
- 优化 Balance Region Scheduler，减少调度开销
- 调整默认 schedule-limit 配置
- 修复频繁分配 ID 问题
- 修复添加调度兼容性问题

16.22.13.3 TiKV

- tikv-ctl 支持 compact 指定的 Region
- Raw KV 支持 Batch Put、 Batch Get、 Batch Delete 和 Batch Scan
- 解决太多 snapshot 导致的 OOM 问题
- Coprocessor 返回更详细的错误信息
- 支持通过 tikv-ctl 动态修改 TiKV 的 block-cache-size
- 进一步完善 importer 功能
- 简化 ImportSST::Upload 接口
- 设置 gRPC 的 keepalive 属性
- tikv-importer 作为独立的 binary 从 TiKV 中分离出来
- 统计 Coprocessor 每个 scan range 命令扫描了多少行数据
- 解决在 macOS 系统上的编译问题
- 优化 metric 相关的内容
- 解决 snapshot 相关的一个潜在 bug
- 解决误用了一个 RocksDB metric 的问题
- Coprocessor 支持 overflow as warning 选项

16.22.14 TiDB 2.0 RC4 Release Notes

2018 年 3 月 30 日，TiDB 发布 2.0 RC4 版。该版本在 2.0 RC3 版的基础上，对 MySQL 兼容性、系统稳定性和优化器做了很多改进。

16.22.14.1 TiDB

- 支持 `SHOW GRANTS FOR CURRENT_USER();`
- 修复 UnionScan 里的 `Expression` 没有 Clone 的问题
- 支持 `SET TRANSACTION` 语法
- 修复 `copIterator` 中潜在的 goroutine 泄露问题
- 修复 `admin check table` 对包含 null 的 unique index 误判的问题
- 支持用科学计数法显示浮点数
- 修复 binary literal 计算时的类型推导
- 修复解析 `CREATE VIEW` 语句的问题
- 修复语句中同时包含 `ORDER BY` 和 `LIMIT 0` 时 panic 的问题
- 提升 `DecodeBytes` 执行性能
- 优化 `LIMIT 0` 为 `TableDual`，避免无用的执行计划构建

16.22.14.2 PD

- 支持手动 split Region，可用于处理单 Region 热点的问题
- 修复 `pdctl` 运行 `config show all` 不显示 label property 的问题
- metrics 及代码结构相关的优化

16.22.14.3 TiKV

- 限制接收 snapshot 时的内存使用，解决极端情况下的 OOM
- 可以配置 Coprocessor 在遇到 warnings 时的行为
- TiKV 支持导数据模式
- 支持 Region 从正中间分裂
- 提升 CI test 的速度
- 使用 `crossbeam channel`
- 改善 TiKV 在被隔离的情况下由于 leader missing 输出太多日志的问题

16.22.15 TiDB 2.0 RC3 Release Notes

2018 年 3 月 23 日，TiDB 发布 2.0 RC3 版。该版本在 2.0 RC2 版的基础上，对 MySQL 兼容性、系统稳定性和优化器做了很多改进。

16.22.15.1 TiDB

- 修复部分场景下 `MAX/MIN` 结果不正确的问题

- 修复部分场景下 Sort Merge Join 结果未按照 Join Key 有序的问题
- 修复边界条件下 uint 和 int 比较的错误
- 完善浮点数类型的长度和精度检查，提升 MySQL 兼容性
- 完善时间类型解析报错日志，添加更多错误信息
- 完善内存控制，新增对 IndexLookupExecutor 的内存统计
- 优化 ADD INDEX 的执行速度，部分场景下速度大幅度提升
- GROUP BY 子句为空时使用 Stream Aggregation 算子，提升速度
- 支持通过 STRAIGHT_JOIN 来关闭优化器的 Join Reorder 优化
- ADMIN SHOW DDL JOBS 输出更详细的 DDL 任务状态信息
- 支持 ADMIN SHOW DDL JOB QUERIES 查询当前正在运行的 DDL 任务的原始语句
- 支持 ADMIN RECOVER INDEX 命令，用于灾难恢复情况下修复索引数据
- ADD INDEX 操作变更为低优先级，降低对线上业务影响
- 支持参数为 JSON 类型的 SUM/AVG 等聚合函数
- 支持配置文件修改 lower_case_table_names 系统变量，用于支持 OGG 数据同步工具
- 提升对 Navicat 管理工具的兼容性
- 支持在 CRUD 操作中使用隐式的行 ID

16.22.15.2 PD

- 支持 Region Merge，合并数据删除后产生的空 Region 或小 Region
- 添加副本时忽略有大量 pending peer 的节点，提升恢复副本及下线的速度
- 优化有大量空 Region 时产生的频繁调度问题
- 优化不同 label 中资源不均衡的场景中 leader balance 调度的速度
- 添加更多异常 Region 的统计

16.22.15.3 TiKV

- 支持 Region Merge
- Raft snapshot 流程完成之后立刻通知 PD，加速调度
- 增加 Raw DeleteRange API
- 增加 GetMetric API
- 减缓 RocksDB sync 文件造成的 I/O 波动
- 优化了对 delete 掉数据的空间回收机制
- 完善数据恢复工具 tikv-ctl
- 解决了由于 snapshot 导致下线节点慢的问题
- Coprocessor 支持 streaming
- 支持 Readpool，raw_get/get/batch_get 性能提升 30%
- 支持配置 Coprocessor 请求超时时间
- Coprocessor 支持 streaming aggregation
- 上报 Region heartbeat 时携带时间信息
- 限制 snapshot 文件的空间使用，防止占用过多磁盘空间
- 对长时间不能选出 leader 的 Region 进行记录上报
- 加速启动阶段的垃圾清理工作
- 根据 compaction 事件及时更新对应 Region 的 size 信息

- 对 `scan lock` 的大小进行限制，防止请求超时
- 使用 `DeleteRange` 加速 Region 删除
- 支持在线修改 RocksDB 的参数

16.22.16 TiDB 2.0 RC1 Release Notes

2018 年 3 月 9 日，TiDB 发布 2.0 RC1 版。该版本在上一版的基础上，对 MySQL 兼容性、系统稳定性和优化器做了很多改进。

16.22.16.1 TiDB

- 支持限制单条 SQL 语句使用内存的大小，减少程序 OOM 风险
- 支持下推流式聚合算子到 TiKV
- 支持配置文件的合法性检测
- 支持 HTTP API 获取 TiDB 参数信息
- Parser 兼容更多 MySQL 语法
- 提升对 Navicat 的兼容性
- 优化器提升，提取多个 OR 条件的公共表达式，选取更优执行计划
- 优化器提升，在更多场景下将子查询转换成 Join 算子，选取更优查询计划
- 使用 Batch 方式 Resolve Lock，提升垃圾回收速度
- 修复 Boolean 类型的字段长度，提升兼容性
- 优化 Add Index 操作，所有的读写操作采用低优先级，减小对在线业务的影响

16.22.16.2 PD

- 优化检查 Region 状态的代码逻辑，提升程序性能
- 优化异常情况下日志信息输出，便于调试
- 修复监控中关于 TiKV 节点磁盘空间不足情况的统计
- 修复开启 TLS 时健康检查接口误报的问题
- 修复同时添加副本数量可能超过配置阈值的问题，提升程序稳定性

16.22.16.3 TiKV

- 修复 PD leader 切换，gRPC call 没被 cancel 的问题
- 对重要配置进行保护，第一次设置之后不允许变更
- 增加获取 metrics 的 gRPC API
- 启动时候，检查是否使用 SSD
- 使用 ReadPool 优化读性能，`raw get` 测试性能提升 30%
- 完善 metrics，优化 metrics 的使用

16.22.17 TiDB 1.1 Beta Release Notes

2018 年 2 月 24 日，TiDB 发布 1.1 Beta 版。该版本在 1.1 Alpha 版的基础上，对 MySQL 兼容性、系统稳定性做了很多改进。

16.22.17.1 TiDB

- 添加更多监控项，优化日志
- 兼容更多 MySQL 语法
- 在 `information_schema` 中支持显示建表时间
- 提速包含 MaxOneRow 算子的查询
- 控制 Join 产生的中间结果集大小，进一步减少 Join 的内存使用
- 增加 `tidb_config` session 变量，输出当前 TiDB 配置
- 修复 Union 和 `Index Join` 算子中遇到的 panic 问题
- 修复 `Sort Merge Join` 算子在部分场景下结果错误的问题
- 修复 `Show Index` 语句显示正在添加过程中的索引的问题
- 修复 `Drop Stats` 语句失败的问题
- 优化 SQL 引擎查询性能，Sysbench 的 Select/OLTP 测试结果提升 10%
- 使用新的执行引擎提升优化器中的子查询计算速度；相比 1.0 版本，在 TPC-H 以及 TPC-DS 等测试中有显著提升

16.22.17.2 PD

- 增加 Drop Region 调试接口
- 支持设置 PD leader 优先级
- 支持配置特定 label 的节点不调度 Raft leader
- 增加枚举各个 PD health 状态的接口
- 添加更多 metrics
- PD leader 尽量与 etcd leader 保持同步
- 提高 TiKV 宕机时数据恢复优先级和恢复速度
- 完善 `data-dir` 配置项的合法性较验
- 优化 Region heartbeat 性能
- 修复热点调度破坏 label 约束的问题
- 其他稳定性问题修复

16.22.17.3 TiKV

- 使用 offset + limit 遍历 lock，消除潜在的 GC 问题
- 支持批量 resolve lock，提升 GC 速度
- 支持并行 GC，提升 GC 速度
- 使用 RocksDB compaction listener 更新 Region Size，让 PD 更精确的进行调度
- 使用 `DeleteFilesInRanges` 批量删除过期数据，提高 TiKV 启动速度
- 设置 Raft snapshot max size，防止遗留文件占用太多空间
- `tikv-ctl` 支持更多修复操作
- 优化有序流式聚合操作
- 完善 metrics，修复 bug

16.22.18 TiDB 1.1 Alpha Release Notes

2018 年 1 月 19 日，TiDB 发布 1.1 Alpha 版。该版本对 MySQL 兼容性、SQL 优化器、系统稳定性、性能做了大量的工作。

16.22.18.1 TiDB

- SQL parser
  - 兼容更多语法
- SQL 查询优化器
  - 统计信息减小内存占用
  - 优化统计信息启动时载入的时间
  - 更精确的代价估算
  - 使用 `Count-Min Sketch` 更精确地估算点查的代价
  - 支持更复杂的条件，更充分使用索引
- SQL 执行器
  - 使用 Chunk 结构重构所有执行器算子，提升分析型语句执行性能，减少内存占用
  - 优化 `INSERT IGNORE` 语句性能
  - 下推更多的类型和函数
  - 支持更多的 `SQL_MODE`
  - 优化 `Load Data` 性能，速度提升 10 倍
  - 优化 `Use Database` 性能
  - 支持对物理算子内存使用进行统计
- Server
  - 支持 PROXY protocol

16.22.18.2 PD

- 增加更多的 API
- 支持 TLS
- 给 Simulator 增加更多的 case
- 调度适应不同的 Region size
- Fix 了一些调度的 bug

16.22.18.3 TiKV

- 支持 Raft learner
- 优化 Raft Snapshot，减少 I/O 开销
- 支持 TLS
- 优化 RocksDB 配置，提升性能
- 优化 Coprocessor `count (*)` 和点查 unique index 的性能

- 增加更多的 Failpoint 以及稳定性测试 case
- 解决 PD 和 TiKV 之间重连的问题
- 增强数据恢复工具 `tikv-ctl` 的功能
- Region 支持按 table 进行分裂
- 支持 `Delete Range` 功能
- 支持设置 snapshot 导致的 I/O 上限
- 完善流控机制

16.23 v1.0

16.23.1 TiDB 1.0 Release Notes

2017 年 10 月 16 日，TiDB 发布 GA 版 (TiDB 1.0)。该版本对 MySQL 兼容性、SQL 优化器、系统稳定性、性能做了大量的工作。

16.23.1.1 TiDB

- SQL 查询优化器
  - 调整代价模型
  - Analyze 下推
  - 函数签名下推
- 优化内部数据格式，减小中间结果大小
- 提升 MySQL 兼容性
- 支持 NO_SQL_CACHE 语法，控制存储引擎对缓存的使用
- 重构 Hash Aggregator 算子，降低内存使用
- 支持 Stream Aggragator 算子

16.23.1.2 PD

- 支持基于读流量的热点调度
- 支持设置 Store 权重，以及基于权重的调度

16.23.1.3 TiKV

- Coprocessor 支持更多下推函数
- 支持取样操作下推
- 支持手动触发数据 Compact，用于快速回收空间
- 提升性能和稳定性
- 增加 Debug API，方便调试

16.23.1.4 TiSpark Beta Release

- 支持可配置框架
- 支持 ThriftSever/JDBC 和 Spark SQL 脚本入口

16.23.1.5 源码地址

源码地址

16.23.1.6 鸣谢

16.23.1.6.1 特别感谢参与项目的企业和团队

- Archon
- Mobike
- SpeedyCloud
- UCloud
- 腾讯云
- 韩国三星研究院

16.23.1.6.2 感谢以下组织/个人提供出色的开源软件/服务

- Asta Xie
- CNCF
- CoreOS
- Databricks
- Docker
- Github
- Grafana
- gRPC
- Jepsen
- Kubernetes
- Namazu
- Prometheus
- RedHat
- RocksDB Team
- Rust Team

16.23.1.6.3 感谢社区个人贡献者 TiDB Contributor

- 8cbx
- Akihiro Suda
- aliyx
- alston111111
- andelf
- Andy Librian
- Arthur Yang
- astaxie
- Bai, Yang

- bailaohe
- Bin Liu
- Blame cosmos
- Breezewish
- Carlos Ferreira
- Ce Gao
- Changjian Zhang
- Cheng Lian
- Cholerae Hu
- Chu Chao
- coldwater
- Cole R Lawrence
- cuiqiu
- cuiyuan
- Cwen
- Dagang
- David Chen
- David Ding
- dawxy
- dcadevil
- Deshi Xiao
- Di Tang
- disksing
- dongxu
- dreamquster
- Drogon
- Du Chuan
- Dylan Wen
- eBoyy
- Eric Romano
- Ewan Chou
- Fiisio
- follitude
- Fred Wang
- follitude
- fud
- fudali
- gaoyangxiaozhu
- Gogs
- goroutine
- Gregory Ian
- Guanqun Lu
- Guilherme Hübner Franco
- Haibin Xie
- Han Fei

- Hiroaki Nakamura
- hiwjd
- Hongyuan Wang
- Hu Ming
- Hu Ziming
- Huachao Huang
- HuaiyuXu
- Huxley Hu
- iamxy
- Ian
- insion
- iroi44
- Ivan.Yang
- Jack Yu
- jacky liu
- Jan Mercl
- Jason W
- Jay
- Jay Lee
- Jianfei Wang
- Jiaxing Liang
- Jie Zhou
- jinhelin
- Jonathan Boulle
- Karl Ostendorf
- knarfeh
- Kuiba
- leixuechun
- li
- Li Shihai
- Liao Qiang
- Light
- lijian
- Lilian Lee
- Liqueur Librazy
- Liu Cong
- Liu Shaohui
- liubo0127
- liyanan
- lkk2003rty
- Louis
- louishust
- luckcolors
- Lynn
- Mae Huang

- maiyang
- maxwell
- mengshangqi
- Michael Belenchenko
- mo2zie
- morefreeze
- MQ
- mxlxm
- Neil Shen
- netroby
- ngaut
- Nicole Nie
- nolouch
- onlymellb
- overvenus
- PaladinTyrion
- paulg
- Priya Seth
- qgxiaozhan
- qhsong
- Qiannan
- qiuyesuifeng
- queenypingcap
- qupeng
- Rain Li
- ranxiaolong
- Ray
- Rick Yu
- shady
- ShawnLi
- Shen Li
- Sheng Tang
- Shirly
- Shuai Li
- ShuNing
- ShuYu Wang
- siddontang
- silenceper
- Simon J Mudd
- Simon Xia
- skimmilk6877
- sllt
- soup
- Sphinx
- Steffen

- sumBug
- sunhao2017
- Tao Meng
- Tao Zhou
- tennix
- tiancaiamao
- TianGuangyu
- Tristan Su
- ueizhou
- UncP
- Unknwon
- v01dstar
- Van
- WangXiangUSTC
- wangyisong1996
- weekface
- wegel
- Wei Fu
- Wenbin Xiao
- Wenting Li
- Wenxuan Shi
- winkyao
- woodpenker
- wuxuelian
- Xiang Li
- xiaojian cai
- Xuanjia Yang
- Xuanwo
- XuHuaiyu
- Yang Zhexuan
- Yann Autissier
- Yanzhe Chen
- Yiding Cui
- Yim
- youyouhu
- Yu Jun
- Yuwen Shen
- Zejun Li
- Zhang Yuning
- zhangjinpeng1987
- ZHAO Yijun
- ZhengQian
- ZhengQianFang
- zhengwanbo
- Zhe-xuan Yang

- ZhiFeng Hu
- Zhiyuan Zheng
- Zhou Tao
- Zhoubirdblue
- zhouningnan
- Ziyi Yan
- zs634134578
- zyguan
- zz-jason
- qiukeren
- hawkingrei
- wangyanjun
- zxylvlp

16.23.2 TiDB Pre-GA Release Notes

2017 年 8 月 30 日，TiDB 发布 Pre-GA 版。该版本对 MySQL 兼容性、SQL 优化器、系统稳定性、性能做了大量的工作。

16.23.2.1 TiDB

- SQL 查询优化器
  - 调整代价模型
  - 优化索引选择，支持不同类型字段比较的索引选择
  - 支持基于贪心算法的 Join Reorder
- 大量 MySQL 兼容性相关功能
- 支持 Natural Join
- 完成 JSON 类型支持 (Experimental)，包括对 JSON 中的字段查询、更新、建索引
- 裁剪无用数据，减小执行器内存消耗
- 支持在 SQL 语句中设置优先级，并根据查询类型自动设置部分语句的优先级
- 完成表达式重构，执行速度提升 30% 左右

16.23.2.2 PD

- 支持手动切换 PD 集群 Leader

16.23.2.3 TiKV

- Raft Log 使用独立的 RocksDB 实例
- 使用 DeleteRange 加快删除副本速度
- Coprocessor 支持更多运算符下推
- 提升性能，提升稳定性

16.23.2.4 TiSpark Beta Release

- 支持谓词下推
- 支持聚合下推
- 支持范围裁剪
- 通过 TPC-H 测试（除去一个需要 View 的 Query）

16.23.3 TiDB RC4 Release Notes

2017 年 8 月 4 日，TiDB 正式发布 RC4 版。该版本对 MySQL 兼容性、SQL 优化器、系统稳定性、性能做了大量的工作。性能方面重点优化了写入速度，计算任务调度支持优先级，避免分析型大事务影响在线事务。SQL 优化器全新改版，查询代价估算更加准确，且能够自动选择 Join 物理算子。功能方面进一步 MySQL 兼容性。同时为了更好的支持 OLAP 业务，开源了 TiSpark 项目，可以通过 Spark 读取和分析 TiKV 中的数据。

16.23.3.1 TiDB

- SQL 查询优化器重构
  - 更好的支持 TopN 查询
  - 支持 Join 算子根据代价自动选择
  - 更完善的 Projection Elimination
- Schema 版本检查区分 Table，避免 DDL 干扰其他正在执行的事务
- 支持 BatchIndexJoin
- 完善 Explain 语句
- 提升 Index Scan 性能
- 大量 MySQL 兼容性相关功能
- 支持 Json 类型及其操作
- 支持查询优先级、隔离级别的设置

16.23.3.2 PD

- 支持通过 PD 设置 TiKV location labels
- 调度优化
  - 支持 PD 主动向 TiKV 下发调度命令
  - 加快 region heartbeat 响应速度
  - 优化 balance 算法
- 优化数据加载，加快 failover 速度

16.23.3.3 TiKV

- 支持查询优先级设置
- 支持 RC 隔离级别
- 完善 Jepsen，提升稳定性

- 支持 Document Store
- Coprocessor 支持更多下推函数
- 提升性能，提升稳定性

16.23.3.4 TiSpark Beta Release

- 支持谓词下推
- 支持聚合下推
- 支持范围裁剪
- 通过 TPC-H 测试（除去一个需要 View 的 Query）

16.23.4 TiDB RC3 Release Notes

2017 年 6 月 16 日，TiDB 正式发布 RC3 版。该版本对 MySQL 兼容性、SQL 优化器、系统稳定性、性能做了大量的工作。性能方面重点优化了负载均衡调度策略和流程。功能方面进一步完善权限管理功能，用户可以按照 MySQL 的权限管理方式控制数据访问权限。另外 DDL 的速度也得到显著的提升。同时为了简化运维工作，开源了 TiDB Ansible 项目，可以一键部署/升级/启停 TiDB 集群。

16.23.4.1 TiDB

- SQL 查询优化器
  - 统计信息收集和使用
  - 关联子查询优化
  - 优化 CBO 框架
  - 通过 Unique Key 信息消除聚合
  - 重构 Expression
  - Distinct 转换为 GroupBy
  - 支持 topn 操作下推
- 支持基本权限管理
- 新增大量 MySQL 内建函数
- 完善 Alter Table 语句，支持修改表名、默认值、注释
- 支持 Create Table Like 语句
- 支持 Show Warnings 语句
- 支持 Rename Table 语句
- 限制单个事务大小，避免大事务阻塞整个集群
- Load Data 过程中对数据进行自动拆分
- 优化 AddIndex、 Delete 语句性能
- 支持 “ANSI_QUOTES” sql_mode
- 完善监控
- 修复 Bug
- 修复内存泄漏问题

16.23.4.2 PD

- 支持 Label 对副本进行 Location 调度
- 基于 region 数量的快速调度
- pd-ctl 支持更多功能
  - 添加、删除 PD
  - 通过 Key 获取 Region 信息
  - 添加、删除 scheduler 和 operator
  - 获取集群 label 信息

16.23.4.3 TiKV

- 支持 Async Apply 提升整体写入性能
- 使用 prefix seek 提升 Write CF 的读取性能
- 使用 memory hint prefix 提升 Raft CF 插入性能
- 优化单行读事务性能
- 支持更多下推功能
- 加入更多统计
- 修复 Bug

16.23.5 TiDB RC2 Release Notes

2017 年 3 月 1 日，TiDB 正式发布 RC2 版。该版本对 MySQL 兼容性、SQL 优化器、系统稳定性、性能做了大量的工作。对于 OLTP 场景，读取性能提升 60%，写入性能提升 30%。另外提供了权限管理功能，用户可以按照 MySQL 的权限管理方式控制数据访问权限。

16.23.5.1 TiDB

- SQL 查询优化器
  - 统计信息收集和使用
  - 关联子查询优化
  - 优化 CBO 框架
  - 通过 Unique Key 信息消除聚合
  - 重构 Expression
  - Distinct 转换为 GroupBy
  - 支持 topn 操作下推
- 支持基本权限管理
- 新增大量 MySQL 内建函数
- 完善 Alter Table 语句，支持修改表名、默认值、注释
- 支持 Create Table Like 语句
- 支持 Show Warnings 语句
- 支持 Rename Table 语句
- 限制单个事务大小，避免大事务阻塞整个集群

- Load Data 过程中对数据进行自动拆分
- 优化 AddIndex、 Delete 语句性能
- 支持 “ANSI_QUOTES” sql_mode
- 完善监控
- 修复 Bug
- 修复内存泄漏问题

16.23.5.2 PD

- 支持 Label 对副本进行 Location 调度
- 基于 region 数量的快速调度
- pd-ctl 支持更多功能
  - 添加、 删除 PD
  - 通过 Key 获取 Region 信息
  - 添加、 删除 scheduler 和 operator
  - 获取集群 label 信息

16.23.5.3 TiKV

- 支持 Async Apply 提升整体写入性能
- 使用 prefix seek 提升 Write CF 的读取性能
- 使用 memory hint prefix 提升 Raft CF 插入性能
- 优化单行读事务性能
- 支持更多下推功能
- 加入更多统计
- 修复 Bug

16.23.6 TiDB RC1 Release Notes

2016 年 12 月 23 日，分布式关系型数据库 TiDB 正式发布 RC1。

16.23.6.1 TiKV

- 提升写入速度
- 降低磁盘空间占用
- 支持百 TB 级别数据
- 提升稳定性，集群规模支持 200 个节点
- 提供 Raw KV API，以及 Golang client

16.23.6.2 PD

- PD 调度策略框架优化，策略更加灵活合理
- 添加 label 支持，支持跨 DC 调度
- 提供 PD Controler，方便操作 PD 集群

16.23.6.3 TiDB

- SQL 查询优化器
  - 支持 eager aggregate
  - 更详细的 explain 信息
  - union 算子并行化
  - 子查询性能优化
  - 条件下推优化
  - 优化 CBO 框架
- 重构 time 相关类型的实现，提升和 MySQL 的兼容性
- 支持更多的 MySQL 内建函数
- Add Index 语句提速
- 支持用 change column 语句修改列名；支持使用 Alter table 的 modify column 和 change column 完成部分列类型转换

16.23.6.4 工具

- Loader：兼容 Percona 的 Mydumper 数据格式，提供多线程导入、出错重试、断点续传等功能，并且针对 TiDB 有优化
- 开发完成一键部署工具

17 术语表

17.1 A

17.1.1 ACID

ACID 是指数据库管理系统在写入或更新资料的过程中，为保证事务是正确可靠的，所必须具备的四个特性：原子性 (atomicity)、 一致性 (consistency)、 隔离性 (isolation) 以及持久性 (durability)。

- 原子性 (atomicity) 指一个事务中的所有操作，或者全部完成，或者全部不完成，不会结束在中间某个环节。 TiDB 通过 Primary Key 所在Region 的原子性来保证分布式事务的原子性。
- 一致性 (consistency) 指在事务开始之前和结束以后，数据库的完整性没有被破坏。 TiDB 在写入数据之前，会校验数据的一致性，校验通过才会写入内存并返回成功。
- 隔离性 (isolation) 指数据库允许多个并发事务同时对其数据进行读写和修改的能力。隔离性可以防止多个事务并发执行时由于交叉执行而导致数据的不一致，主要用于处理并发场景。 TiDB 目前只支持一种隔离级别，即可重复读。
- 持久性 (durability) 指事务处理结束后，对数据的修改就是永久的，即便系统故障也不会丢失。在 TiDB 中，事务一旦提交成功，数据全部持久化存储到 TiKV，此时即使 TiDB 服务器宕机也不会出现数据丢失。

17.2 B

17.2.1 BR

TiDB 备份恢复功能用户文档中的名词 BR 根据上下文不同有不同的解释，比较常见的指代用法：

- TiDB 备份恢复功能，包含 br CLI、 TiDB Operator、 TiDB Cloud 提供的备份和恢复功能集合。
- 架构中的 BR 功能组件。

名词 br 一般用来指代 br CLI 工具。

17.2.2 Batch Create Table

批量建表 (Batch Create Table) 是在 TiDB v6.0.0 中引入的新功能，此功能默认开启。当需要恢复的数据中带有大量的表（约 50000 张）时，批量建表功能显著提升数据恢复的速度。详情参见批量建表。

17.2.3 Baseline Capturing

自动捕获绑定 (Baseline Capturing) 会对符合捕获条件的查询进行捕获，为符合条件的查询生成相应的绑定。通常用于升级时的计划回退防护。

17.2.4 Bucket

一个Region 在逻辑上划分为多个小范围，称为 bucket。TiKV 按 bucket 收集查询统计数据，并将 bucket 的情况报告给 PD。详情参见 Bucket 设计文档。

17.3 C

17.3.1 Cached Table

缓存表 (Cached Table) 是指 TiDB 把整张表的数据加载到服务器的内存中，直接从内存中获取表数据，避免从 TiKV 获取表数据，从而提升读性能。详情参见缓存表。

17.3.2 Continuous Profiling

持续性能分析 (Continuous Profiling) 是从 TiDB v5.3 起引入的一种从系统调用层面解读资源开销的方法。引入该方法后，TiDB 可提供数据库源码级性能观测，通过火焰图的形式帮助研发、运维人员定位性能问题的根因。详情参见TiDB Dashboard 实例性能分析 - 持续分析页面。

17.4 D

17.4.1 Dynamic Pruning

动态裁剪 (Dynamic Pruning) 是 TiDB 访问分区表的两种模式之一。在动态裁剪模式下，TiDB 的每个算子都支持直接访问多个分区，省略 Union 操作，提高执行效率，还避免了 Union 并发管理的问题。

17.5 I

17.5.1 Index Merge

索引合并 (Index Merge) 是在 TiDB v4.0 版本中作为实验特性引入的一种查询执行方式的优化，可以大幅提高查询在扫描多列数据时条件过滤的效率。自 v5.4 版本起，Index Merge 成为正式功能，详情参见用 EXPLAIN 查看索引合并的 SQL 执行计划。

17.5.2 In-Memory Pessimistic Lock

内存悲观锁 (In-Memory Pessimistic Lock) 是在 TiDB v6.0.0 中引入的新功能。开启内存悲观锁功能后，悲观锁通常只会被存储在 Region leader 的内存中，而不会将锁持久化到磁盘，也不会通过 Raft 协议将锁同步到其他副本，因此可以大大降低悲观事务加锁的开销，提升悲观事务的吞吐并降低延迟。

17.6 L

17.6.1 Leader/Follower/Learner

它们分别对应Peer 的三种角色。其中 Leader 负责响应客户端的读写请求；Follower 被动地从 Leader 同步数据，当 Leader 失效时会进行选举产生新的 Leader；Learner 是一种特殊的角色，它只参与同步 raft log 而不参与投票，在目前的实现中只短暂存在于添加副本的中间步骤。

17.7 O

17.7.1 Old value

Old value 特指在 TiCDC 输出的增量变更日志中的“原始值”。可以通过配置来指定 TiCDC 输出的增量变更日志是否包含“原始值”。

17.7.2 Operator

Operator 是应用于一个 Region 的，服务于某个调度目的的一系列操作的集合。例如“将 Region 2 的 Leader 迁移至 Store 5”，“将 Region 2 的副本迁移到 Store 1, 4, 5”等。

Operator 可以是由 Scheduler 通过计算生成的，也可以是由外部 API 创建的。

17.7.3 Operator Step

Operator Step 是 Operator 执行过程的一个步骤，一个 Operator 常常会包含多个 Operator Step。

目前 PD 可生成的 Step 包括：

- `TransferLeader`：将 Region Leader 迁移至指定 Peer
- `AddPeer`：在指定 Store 添加 Follower
- `RemovePeer`：删除一个 Region Peer

- `AddLearner`：在指定 Store 添加 Region Learner
- `PromoteLearner`：将指定 Learner 提升为 Follower
- `SplitRegion`：将指定 Region 一分为二

17.8 P

17.8.1 Pending/Down

Pending 和 Down 是 Peer 可能出现的两种特殊状态。其中 Pending 表示 Follower 或 Learner 的 raft log 与 Leader 有较大差距，Pending 状态的 Follower 无法被选举成 Leader。Down 是指 Leader 长时间没有收到对应 Peer 的消息，通常意味着对应节点发生了宕机或者网络隔离。

17.8.2 Point get

点查 (point get) 是指通过主键或唯一索引直接读取一行的查询方式。点查的返回结果最多是一行数据。

17.8.3 Predicate columns

执行 SQL 语句时，优化器在大多数情况下只会用到部分列（例如，`WHERE`、`JOIN`、`ORDER BY`、`GROUP BY` 子句中出现的列）的统计信息，这些用到的列称为 `PREDICATE COLUMNS`。详情参见收集部分列的统计信息。

17.9 Q

17.9.1 Quota Limiter

前台限流 (Quota Limiter) 是在 TiDB v6.0.0 版本中作为实验特性引入的功能。当 TiKV 部署的机型资源有限（如 4v CPU，16 G 内存）时，如果 TiKV 前台处理的读写请求量过大，会占用 TiKV 后台处理请求所需的 CPU 资源，最终影响 TiKV 性能的稳定性。此时，开启前台限流相关的quota 相关配置项可以限制前台各类请求占用的 CPU 资源。

17.10 R

17.10.1 Raft Engine

一种内置的持久化存储引擎，有着日志结构的设计，为 TiKV 提供 multi-Raft 日志存储。从 v5.4 起，TiDB 支持使用 Raft Engine 作为 TiKV 的日志存储引擎。详情参见Raft Engine。

17.10.2 Region/Peer/Raft Group

每个 Region 负责维护集群的一段连续数据（默认配置下平均约 96 MiB），每份数据会在不同的 Store 存储多个副本（默认配置是 3 副本），每个副本称为 Peer。同一个 Region 的多个 Peer 通过 raft 协议进行数据同步，所以 Peer 也用来指代 raft 实例中的成员。TiKV 使用 multi-raft 模式来管理数据，即每个 Region 都对应一个独立运行的 raft 实例，我们也把这样的一个 raft 实例叫做一个 Raft Group。

17.10.3 Region Split

TiKV 集群中的 Region 不是一开始就划分好的，而是随着数据写入逐渐分裂生成的，分裂的过程被称为 Region Split。

其机制是集群初始化时构建一个初始 Region 覆盖整个 key space，随后在运行过程中每当 Region 数据达到一定量之后就通过 Split 产生新的 Region。

17.10.4 Restore

备份操作的逆过程，即利用保存的备份数据还原出原始数据的过程。

17.11 S

17.11.1 Scheduler

Scheduler（调度器）是 PD 中生成调度的组件。PD 中每个调度器是独立运行的，分别服务于不同的调度目的。常用的调度器及其调用目标有：

- `balance-leader-scheduler`：保持不同节点的 Leader 均衡。
- `balance-region-scheduler`：保持不同节点的 Peer 均衡。
- `hot-region-scheduler`：保持不同节点的读写热点 Region 均衡。
- `evict-leader-{store-id}`：驱逐某个节点的所有 Leader。（常用于滚动升级）

17.11.2 Store

PD 中的 Store 指的是集群中的存储节点，也就是 tikv-server 实例。Store 与 TiKV 实例是严格一一对应的，即使在同一主机甚至同一块磁盘部署多个 TiKV 实例，这些实例也对会对应不同的 Store。

17.12 T

17.12.1 Top SQL

Top SQL 用于找到一段时间内对某个 TiDB 或 TiKV 节点消耗负载较大的 SQL 查询。详情参见Top SQL 用户文档。

17.12.2 TSO

因为 TiKV 是一个分布式的储存系统，它需要一个全球性的授时服务 TSO (Timestamp Oracle)，来分配一个单调递增的时间戳。这样的功能在 TiKV 中是由 PD 提供的，在 Google 的 Spanner 中是由多个原子钟和 GPS 来提供的。

Time to Live (TTL) 提供了行级别的生命周期控制策略。通过为表设置 TTL 属性，TiDB 可以周期性地自动检查并清理表中的过期数据。

© 2023 PingCAP 公司保留所有权利。除非版权法允许，否则在未得到本公司事先给出的书面许可的情况下，严禁复制、改编或翻译本文。