TiDB 中文手册

PingCAP Inc.

20231128

目录

1 文档中心

2 关于 TiDB

2.1 TiDB 简介

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

2.1.1 五大核心特性

- 一键水平扩容或者缩容

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

- 金融级高可用

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

- 实时 HTAP

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

- 云原生的分布式数据库

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

- 兼容 MySQL 5.7 协议和 MySQL 生态

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

2.1.2 四大核心应用场景

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

  众所周知，金融行业对数据一致性及高可靠、系统高可用、可扩展性、容灾要求较高。传统的解决方案是同城两个机房提供服务、异地一个机房提供数据容灾能力但不提供服务，此解决方案存在以下缺点：资源利用率低、维护成本高、RTO (Recovery Time Objective) 及 RPO (Recovery Point Objective) 无法真实达到企业所期望的值。TiDB 采用多副本 + Multi-Raft 协议的方式将数据调度到不同的机房、机架、机器，当部分机器出现故障时系统可自动进行切换，确保系统的 RTO <= 30s 及 RPO = 0。

- 对存储容量、可扩展性、并发要求较高的海量数据及高并发的 OLTP 场景

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

- Real-time HTAP 场景

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

- 数据汇聚、二次加工处理的场景

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

2.1.3 另请参阅

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

2.2 What’s New in TiDB 4.0

随着 TiDB v4.0 于 2020 年 5 月 28 日正式发布，TiDB v4.0 在稳定性、易用性、性能、安全和功能方面进行了大量的改进。本文概括性地介绍改进的内容，用户可根据实际情况决定升级到 TiDB v4.0。本文仅总结 v4.0 中面向用户且最重要的改进事项，功能及错误修复的完整列表，请参阅Release Notes。

2.2.1 调度功能

- 热点调度支持更多维度。热点调度在决策时，除了根据写入/读取流量作为调度依据外，新引入 key 的维度。可以很大程度改善原有单一维度决策造成的 CPU 资源利用率不均衡的问题。详情参阅：调度。

2.2.2 存储引擎

- TiFlash 是 TiDB 为完善 Realtime HTAP 形态引入的关键组件，TiFlash 通过 Multi-Raft Learner 协议实时从 TiKV 复制数据，确保行存储引擎 TiKV 和列存储引擎 TiFlash 之间的数据强一致。TiKV、TiFlash 可按需部署在不同的机器，解决 HTAP 资源隔离的问题。详情参阅：TiFlash。
- 4.0 版本中 TiKV 提供新的存储格式，提升宽表场景下编解码数据的效率。

2.2.3 TiDB Dashboard

- DBA 通过TiDB Dashboard UI 可以快速了解集群的集群拓扑、配置信息、日志信息、硬件信息、操作系统信息、慢查询信息、SQL 访问信息、诊断报告信息等，帮助 DBA 通过 SQL 快速了解、分析系统的各项指标，具体信息如下：
  - Cluster Info，提供集群中所有组件，包括：TiDB、TiKV、PD、TiFlash 运行状态及其所在主机的运行状态。
  - Key Visualizer，系统可视化输出 TiDB 集群一段时间内的流量情况，用于 DBA 分析 TiDB 集群的使用模式和排查流量热点。
  - SQL Statements，记录当系统执行的所有 SQL 以及 SQL 相关的统计信息，包括：执行次数、执行时间汇总等，帮助用户快速分析系统的 SQL 执行状况，判断系统中有哪些热点 SQL 语句等。
  - Slow Queries，汇总集群中所有的慢查询语句，帮助用户快速定位慢查询语句。
  - Diagnostic Report，系统会周期性的自动对集群可能存在的问题进行诊断，并将诊断结果和一些集群相关的负载监控信息汇总成一个诊断报告。诊断报告是网页形式，通过浏览器保存后可离线浏览和传阅。
  - Log Search & Download，可视化搜索、查询集群的日志信息，帮忙 DBA 分析系统的问题，提升 DBA 运维的效率。

2.2.4 部署运维工具

TiUP 是 4.0 版本中新推出的包管理器的工具，主要用于管理 TiDB 生态内的所有的包，提供组件管理、Playground、Cluster、TUF、离线部署等功能，将安装、部署、运维 TiDB 工具化，提升 DBA 部署、运维 TiDB 的效率。详情参阅：TiUP，具体的功能如下：

- 组件管理功能，提供一键式组件信息查询、安装、升级、卸载等功能，方便 DBA 管理 TiDB 的所有组件。
- 集群管理功能 (Cluster)：提供一键式 TiDB 集群的部署、运维 TiDB 功能，包括：安装、部署、扩容、缩容、升级、配置变更、启动、停止、重启，查询集群状信息等，支持管理多个 TiDB 集群。
- 本地部署功能 (Playground): 提供快速在本地部署一个 TiDB 集群，快速体验、了解 TiDB 的基本功能，注意：此功能仅用于快速了解 TiDB，不适合上生产。
- 私有镜像管理 (Mirror): 当无法通过公网访问 TiUP 官方镜像时，TiUP 提供构建私有镜像的方案，帮助用户构建私有镜像及提供离线部署部署的功能。
- 性能测试功能 (Benchmark): 提供一键部署性能测试工具的功能，主要提供 TPC-C、TPC-H 两种性能测试的 workload 。

2.2.5 事务

- 悲观事务正式 GA 并作为默认事务模式提供，支持 Read Committed 隔离级别以及 `SELECT FOR UPDATE NOWAIT` 语法。详情参阅：悲观事务模型。
- 支持大事务，最大事务限制由 100 MB 提升到了 10 GB，同时支持乐观事务和悲观事务。详情参阅：事务限制。

2.2.6 SQL 功能

- 在 SQL Plan Management 中引入了 SQL Bind 的自动捕获和演进，提升易用性和执行计划稳定性。详情参阅：绑定执行计划。

- 新增 15 种 SQL Hint 用于控制优化器生成执行计划，和执行引擎执行查询时的行为。详情参阅：SQL Hint。
- 支持 `SELECT INTO outfile` 语句，该语句用来将表数据导出到指定的文本文件中，配合上 `LOAD DATA`，可以方便的在数据库之间导入/导出数据。
- 支持自定义序列对象 Sequence，提供 `CACHE/NO_CACHE`、`CYCLE/NO_CYCLE` 选项定义序列的不同特性，满足序列生成的各种需求，用户可以通过 Sequence 替代第三方 ID 生成服务。详情参阅：Sequence。
- 新增 `Flashback` 命令，支持恢复被 `Truncate` 的表。详情参阅：`Flashback`。
- 新增查询数据时将 Join、Sort 中间结果写入本地磁盘，防止查询语句占用内存过多导致系统 OOM 的问题，提升系统的稳定性。
- 优化 `EXPLAIN` 和 `EXPLAIN ANALYZE` 的输出结果，显示更多的信息，提升排查问题的效率。详情参阅：Explain Analyze，Explain。
- 支持 Index Merge 功能，Index Merge 是一种新的表访问方式，当查询只涉及到单张表时，优化器会自动根据查询条件读取多个索引数据并对结果求并集，提升查询单张表时的性能。详情参阅：Index Merge。
- 支持 AutoRandom Key 作为 TiDB 在列属性上的扩展语法，AutoRandom 被设计用于解决自增主键列的写热点问题，为使用自增主键列的用户提供最低成本的 MySQL 迁移方案。详情参阅：AutoRandom Key。
- 新增集群拓扑、配置信息、日志信息、硬件信息、操作系统信息、慢查询信息等系统表等，帮助 DBA 通过 SQL 快速了解、分析系统的各项指标，详情参阅：information_schema，具体信息如下：
  - 新增集群拓扑、配置、日志、硬件、操作系统等信息表，帮助 DBA 快速了集群配置、状态信息：
    * `cluster_info` 表，用于保存集群的拓扑信息。
    * `cluster_log` 表，用于保存系统的日志信息。
    * `cluster_hardware`，`cluster_systeminfo`，用于保存系统中服务器的硬件系统，操作系统信息等。
  - 新增慢查询、诊断结果、性能监控等系统表，帮助 DBA 快速分析系统的性能瓶颈：
    * `cluster_slow_query` 表，用于记录保存全局的慢查询信息。
    * `cluster_processlist` 表，用于记录保存全局的 processlist。
    * `inspection_result` 表，4.0 版本新增自动性能诊断的功能，帮助 DBA 自动分析系统的性能瓶颈并自动输出相关的性能分析报告，方便 DBA 定位常见的问题和异常项，提升 DBA 运维的效率。
    * `metrics_summary` 和 `metric_summary_by_label` 表，用于记录保存系统中的所有监控指标信息，DBA 可以通过 SQL 访问所有的监控指标并可以与历史的监控指标进行对比，方便 DBA 定位、分析异常现象。
    * `inspection_summary` 表，用于记录保存不同的数据链路或者访问链路上各种关键的监控指标，方便 DBA 定位、分析常见数据链路或者访问链路中的异常现象，例如：读数据、写数据链路。

2.2.7 字符集及排序规则

在 TiDB 4.0 的新集群中，支持大小写和口音不敏感的排序规则 `utf8mb4_general_ci` 及 `utf8_general_ci`，详情参阅: 字符集及排序规则。

2.2.8 安全

- 完善客户端与服务端，组件与组件之间的加密通信，确保连接安全性，保护接收与发送的任何数据不会被网络犯罪分子读取和修改。主要支持基于证书的登录认证、在线更新证书、校验 TLS 证书的 CommonName 属性等功能。详情参阅：开启加密传输。
- 透明数据加密 (Transparent Data Encryption)，简称 TDE，是 TiDB 推出的一个新特性，用来对整个数据库提供保护。数据库开启 TDE 加密功能后，对于连接到数据库的应用程序来说是完全透明的，它不需要对现有应用程序做任何改变。因为 TDE 的加密特性是基本于文件级别的，系统会在将数据写到磁盘之前加密，在读取到内存之前解密，确保数据的安全性。目前主要支持 AES128-CTR、AES192-CTR、AES256-CTR 三种加密算法，支持通过 AWS KMS 管理密钥等功能。详情参阅静态加密。

2.2.9 备份与恢复

快速备份恢复功能，用来快速的备份与恢复单个 TiDB 集群的数据，确保数据的可靠性，符合企业备份与恢复或者等保的要求。主要支持快速的全量备份与恢复、支持按照数据排序后区间范围备份与恢复数据。详情参阅：快速备份与恢复工具。

2.2.10 服务级别功能

- 支持缓存 `Prepare/Execute` 请求的执行计划，提升 SQL 的执行效率。详情参阅：缓存执行计划。
- 支持自适应线程池功能，精简线程池数量，优化请求处理调度方式，提升产品易用性，提升产品的性能。
- Follower Read 功能是指在强一致性读的前提下使用 Region 的 follower 副本来承载数据读取的任务，从而提升 TiDB 集群的吞吐能力并降低 leader 负载。Follower Read 包含一系列将 TiKV 读取负载从 Region 的 leader 副本上 offload 到 follower 副本的负载均衡机制。TiKV 的 Follower Read 可以保证数据读取的一致性，可以为用户提供强一致的数据读取能力。详情参阅：Follower Read。

2.2.11 TiCDC

TiCDC 支持通过拉取 TiKV 变更日志实现 TiDB 集群之间数据同步，支持数据的高可靠、服务的高可用能力，确保数据不会丢失。用户可以通过订阅的方式订阅数据的变更信息，系统会自动将数据推送到下游系统，当前仅支持 MySQL 协议的数据库（例如：MySQL、TiDB），Kafka 及 Pulsar 作为 TiCDC 的下游，同时用户也可以通过 TiCDC 提供的开放数据协议自行扩展支持的下游系统。详情参阅：TiCDC。

2.3 TiDB 4.0 实验特性

本文档介绍 TiDB 4.0 版本中的实验特性。不建议在生产环境中使用实验特性。

2.3.1 调度功能

- Cascading Placement Rules 是一套副本规则系统，用于指导 PD 针对不同类型的数据生成对应的调度。通过组合不同的调度规则，用户可以精细地控制任何一段连续数据的副本数量、存放位置、主机类型、是否参与 Raft 投票、是否可以担任 Raft leader 等属性。详情参阅：Cascading Placement Rules。

- 弹性调度功能，结合 Kubernetes，可根据实时负载状态，动态扩缩节点，能够有效地缓解业务高峰的压力并且节约不必要的成本开销。详情参阅：启用 TidbCluster 弹性伸缩。
- 随机采样约 10000 行数据来快速构建统计信息（v3.0 实验特性）

2.3.2 SQL 功能

- 支持表达式索引 (Expression Index) 功能，表达式索引也叫函数索引，在创建索引时索引的字段不一定要是一个具体的列，而可以由一个或者多个列计算出来的表达式。对于快速访问那些基于计算结果的表非常有用。详情参阅：表达式索引。
- 执行计划缓存（v4.0 实验特性）。
- Cascades Planner：基于 Cascades 框架的自顶向下查询优化器。（v3.0 实验特性）
- 表级锁 (Table Lock)（v4.0.0 实验特性）

2.3.3 服务级别功能

- TiDB 实例支持以 Region 为单位缓存算子下推到 TiKV 的的返回结果，提升 SQL 语句完全一致、SQL 语句包含一个变化条件且仅限于表主键或分区主键，其他部分一致和 SQL 语句包含多个变化的条件且条件完全匹配一个复合索引列，其他部分一致场景时 SQL 执行的效率。详情参阅：缓存查询结果。
- 支持将配置参数持久化存储到 PD 中，支持动态修改配置项功能，提升产品易用性。

2.4 TiDB 基本功能

本文详细介绍 TiDB 具备的基本功能。

2.4.1 数据类型

- 数值类型：BIT、BOOL|BOOLEAN、SMALLINT、MEDIUMINT、INT|INTEGER、BIGINT、FLOAT、DOUBLE、DECIMAL。
- 日期和时间类型：DATE、TIME、DATETIME、TIMESTAMP、YEAR。
- 字符串类型：CHAR、VARCHAR、TEXT、TINYTEXT、MEDIUMTEXT、LONGTEXT、BINARY、VARBINARY、BLOB、TINYBLOB、MEDIUMBLOB、LONGBLOB、ENUM、SET。
- JSON 类型。

2.4.2 运算符

- 算术运算符、位运算符、比较运算符、逻辑运算符、日期和时间运算符等。

2.4.3 字符集及排序规则

- 字符集：UTF8、UTF8MB4、BINARY、ASCII、LATIN1。
- 排序规则：UTF8MB4_GENERAL_CI、UTF8MB4_GENERAL_BIN、UTF8_GENERAL_CI、UTF8_GENERAL_BIN、BINARY。

2.4.4 函数

- 控制流函数、字符串函数、日期和时间函数、位函数、数据类型转换函数、数据加解密函数、压缩和解压函数、信息函数、JSON 函数、聚合函数、窗口函数等。

2.4.5 SQL 语句

- 完全支持标准的 Data Definition Language (DDL) 语句，例如：CREATE、 DROP、 ALTER、 RENAME、 TRUNCATE 等。
- 完全支持标准的 Data Manipulation Language (DML) 语句，例如：INSERT、REPLACE、SELECT、Subqueries、UPDATE、LOAD DATA 等。
- 完全支持标准的 Transactional and Locking 语句，例如：START TRANSACTION、COMMIT、ROLLBACK、SET TRANSACTION 等。
- 完全支持标准的 Database Administration 语句，例如：SHOW、 SET 等。
- 完全支持标准的 Utility 语句，例如：DESCRIBE、 EXPLAIN、 USE 等。
- 完全支持 SQL GROUP BY 和 ORDER BY 子语句。
- 完全支持标准 SQL 语法的 LEFT OUTER JOIN 和 RIGHT OUTER JOIN。
- 完全支持标准 SQL 要求的表和列别名。

2.4.6 分区表

- 支持 Range 分区。
- 支持 Hash 分区。

2.4.7 视图

- 支持普通视图。

2.4.8 约束

- 支持非空约束。
- 支持主键约束。
- 支持唯一约束。

2.4.9 安全

- 支持基于 RBAC (role-based access control) 的权限管理。
- 支持密码管理。
- 支持通信、数据加密。
- 支持 IP 白名单。
- 支持审计功能。

2.4.10 工具

- 支持快速备份功能。
- 支持通过工具从 MySQL 迁移数据到 TiDB。
- 支持通过工具部署、运维 TiDB。

2.5 性能测试报告

2.5.1 TiDB Sysbench 性能对比测试报告 - v4.0 对比 v3.0

2.5.1.1 测试目的

测试对比 TiDB v4.0 和 v3.0 在 OLTP 场景下的性能。

2.5.1.2 测试环境 (AWS EC2)

2.5.1.2.1 硬件配置

服务类型	EC2 类型	实例数
PD	m5.xlarge	3
TiKV	i3.4xlarge	3
TiDB	c5.4xlarge	3
Sysbench	m5.4xlarge	1

2.5.1.2.2 软件版本

服务类型	软件版本
PD	3.0、4.0
TiDB	3.0、4.0
TiKV	3.0、4.0
Sysbench	1.0.20

2.5.1.2.3 参数配置

TiDB v3.0 参数配置

```
log.level: “error”
performance.max-procs: 20
prepared-plan-cache.enabled: true
tikv-client.max-batch-wait-time: 2000000
```

TiKV v3.0 参数配置

```
storage.scheduler-worker-pool-size: 5
raftstore.store-pool-size: 3
raftstore.apply-pool-size: 3
rocksdb.max-background-jobs: 3
raftdb.max-background-jobs: 3
raftdb.allow-concurrent-memtable-write: true
server.grpc-concurrency: 6
readpool.storage.normal-concurrency: 10
readpool.coprocessor.normal-concurrency: 5
```

TiDB v4.0 参数配置

```
log.level: “error”
performance.max-procs: 20
prepared-plan-cache.enabled: true
tikv-client.max-batch-wait-time: 2000000
```

TiKV v4.0 参数配置

```
storage.scheduler-worker-pool-size: 5
raftstore.store-pool-size: 3
raftstore.apply-pool-size: 3
rocksdb.max-background-jobs: 3
raftdb.max-background-jobs: 3
raftdb.allow-concurrent-memtable-write: true
server.request-batch-enable-cross-command: false
server.grpc-concurrency: 6
readpool.unified.min-thread-count: 5
readpool.unified.max-thread-count: 20
readpool.storage.normal-concurrency: 10
pessimistic-txn.pipelined: true
```

全局变量配置

```
set global tidb_hashagg_final_concurrency=1;
set global tidb_hashagg_partial_concurrency=1;
set global tidb_disable_txn_auto_retry=0;
```

2.5.1.3 测试方案

1. 通过 TiUP 部署 TiDB v4.0 和 v3.0。
2. 通过 Sysbench 导入 16 张表，每张表有 1000 万行数据。
3. 分别对每个表执行 `analyze table` 命令。
4. 备份数据，用于不同并发测试前进行数据恢复，以保证每次数据一致。
5. 启动 Sysbench 客户端，进行 `point_select`、`read_write`、`update_index` 和 `update_non_index` 测试。通过 AWS NLB 向 TiDB 加压，单轮预热 1 分钟，测试 5 分钟。
6. 每轮完成后停止集群，使用之前的备份的数据覆盖，再启动集群。

2.5.1.3.1 准备测试数据

执行以下命令来准备测试数据：

```
sysbench oltp_common \
    --threads=16 \
    --rand-type=uniform \
    --db-driver=mysql \
    --mysql-db=sbtest \
    --mysql-host=$aws_nlb_host \
    --mysql-port=$aws_nlb_port \
    --mysql-user=root \
    --mysql-password=password \
    prepare --tables=16 --table-size=10000000
```

2.5.1.3.2 执行测试命令

执行以下命令来执行测试：

```
sysbench $testname \
    --threads=$threads \
    --time=300 \
    --report-interval=1 \
    --rand-type=uniform \
    --db-driver=mysql \
    --mysql-db=sbtest \
    --mysql-host=$aws_nlb_host \
    --mysql-port=$aws_nlb_port \
    run --tables=16 --table-size=10000000
```

2.5.1.4 测试结果

2.5.1.4.1 Point Select 性能

Threads	v3.0 QPS	v3.0 95% latency (ms)	v4.0 QPS	v4.0 95% latency (ms)	QPS 提升
150	117085.701	1.667	118165.1357	1.608	0.92%
300	200621.4471	2.615	207774.0859	2.032	3.57%
600	283928.9323	4.569	320673.342	3.304	12.94%
900	343218.2624	6.686	383913.3855	4.652	11.86%
1200	347200.2366	8.092	408929.4372	6.318	17.78%
1500	366406.2767	10.562	418268.8856	7.985	14.15%

v4.0 对比 v3.0，Point Select 性能提升了 14%。

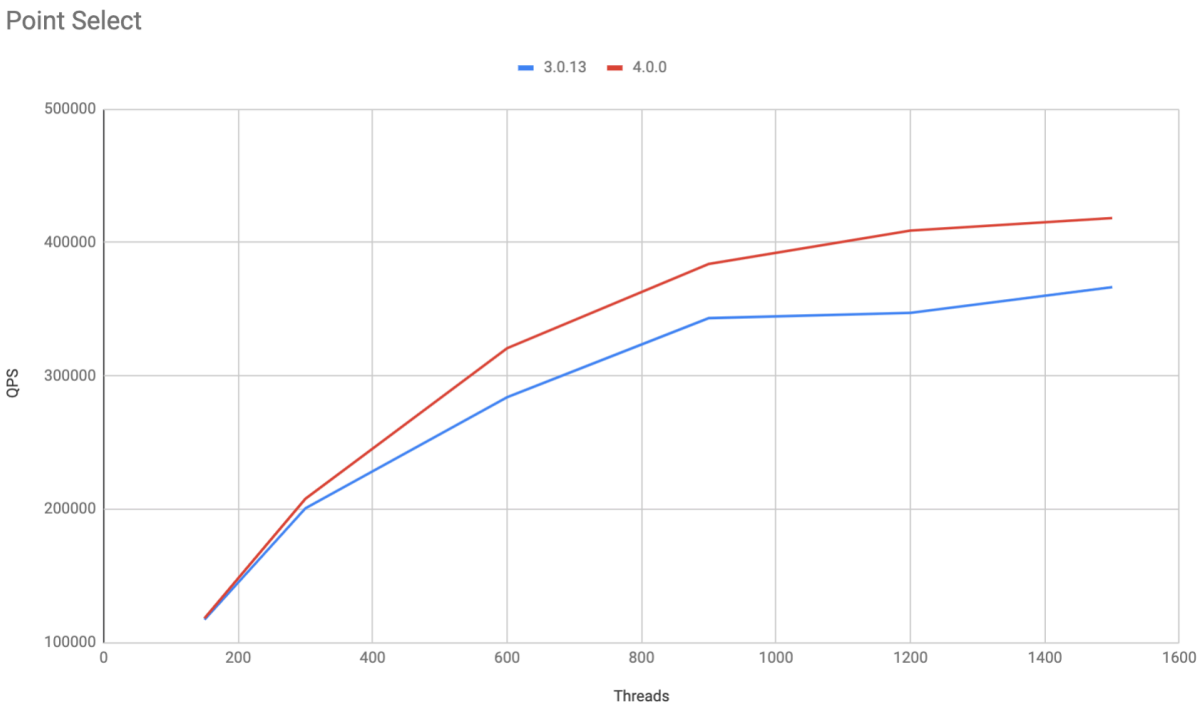


图 1: Point Select

2.5.1.4.2 Update Non-index 性能

Threads	v3.0 QPS	v3.0 95% latency (ms)	v4.0 QPS	v4.0 95% latency (ms)	QPS 提升
150	15446.41024	11.446	16954.39971	10.844	9.76%
300	22276.15572	17.319	24364.44689	16.706	9.37%
600	28784.88353	29.194	31635.70833	28.162	9.90%
900	32194.15548	42.611	35787.66078	38.942	11.16%
1200	33954.69114	58.923	38552.63158	51.018	13.54%
1500	35412.0032	74.464	40859.63755	62.193	15.38%

v4.0 对比 v3.0，Update Non-index 性能提升了 15%。

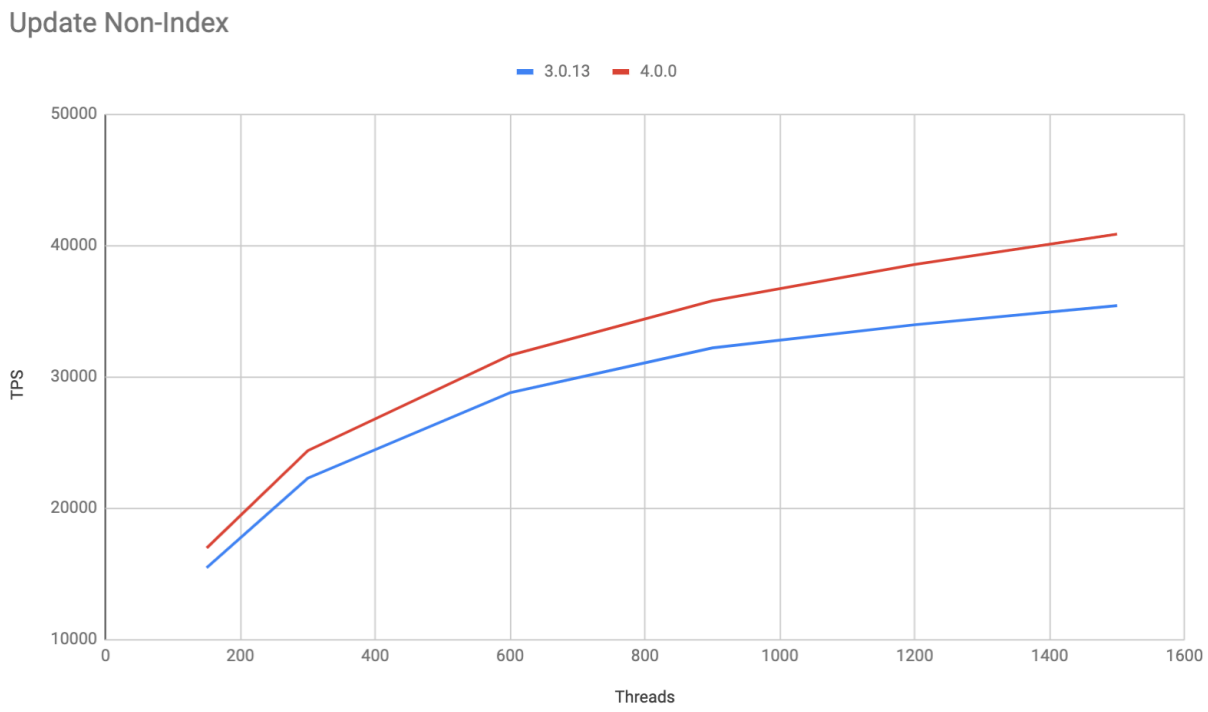


图 2: Update Non-index

2.5.1.4.3 Update Index 性能

Threads	v3.0 QPS	v3.0 95% latency (ms)	v4.0 QPS	v4.0 95% latency (ms)	QPS 提升
150	11164.40571	16.706	11954.73635	16.408	7.08%
300	14460.98057	28.162	15243.40899	28.162	5.41%
600	17112.73036	53.85	18535.07515	50.107	8.31%
900	18233.83426	86.002	20339.6901	70.548	11.55%
1200	18622.50283	127.805	21390.25122	94.104	14.86%
1500	18980.34447	170.479	22359.996	114.717	17.81%

v4.0 对比 v3.0，Update Index 性能提升了 17%。

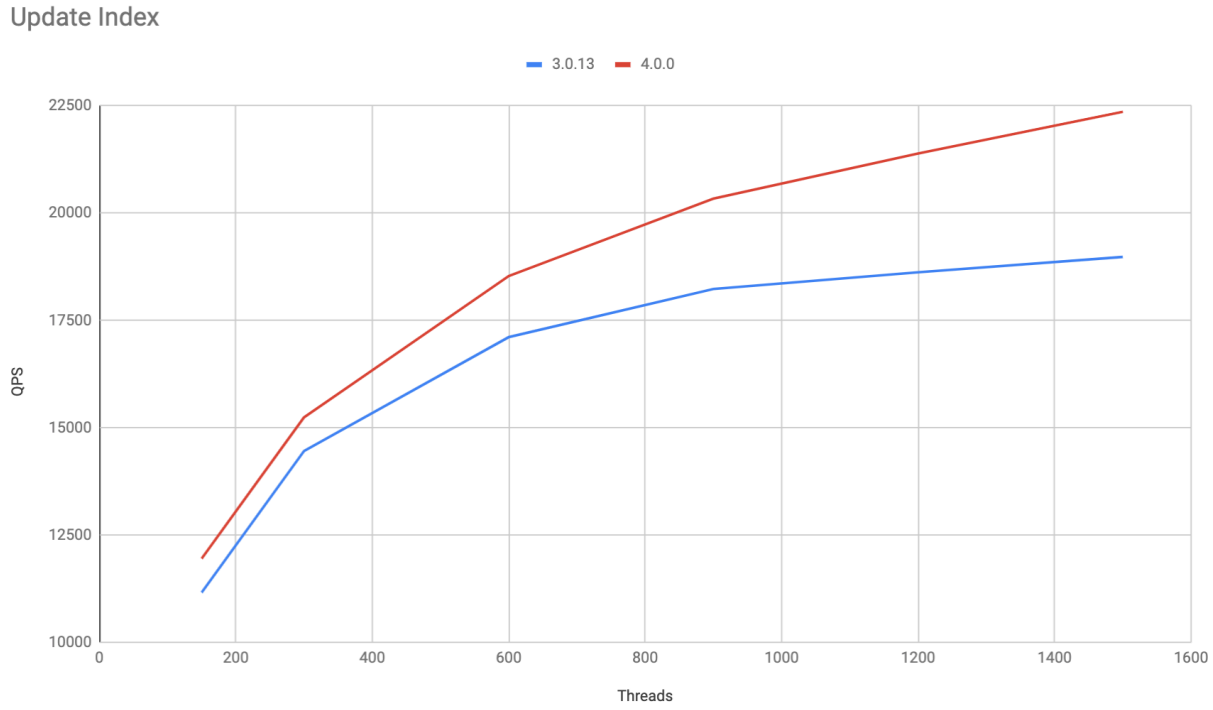


图 3: Update Index

2.5.1.4.4 Read Write 性能

Threads	v3.0 QPS	v3.0 95% latency (ms)	v4.0 QPS	v4.0 95% latency (ms)	QPS 提升
150	43768.33633	71.83	53912.63705	59.993	23.18%
300	55655.63589	121.085	71327.21336	97.555	28.16%
600	64642.96992	223.344	84487.75483	176.731	30.70%
900	68947.25293	325.984	90177.94612	257.95	30.79%
1200	71334.80099	434.829	92779.71507	344.078	30.06%
1500	72069.9115	580.017	95088.50812	434.829	31.94%

v4.0 对比 v3.0，Read Write 性能提升了 31%。

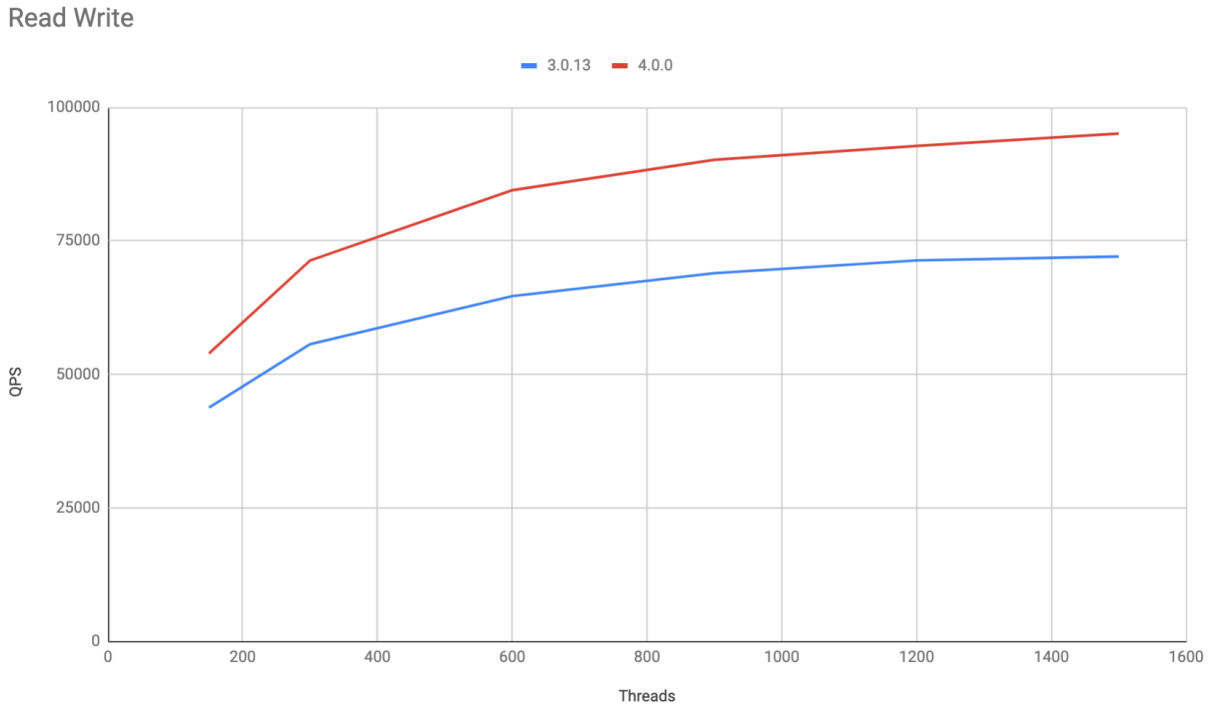


图 4: Read Write

2.5.2 TiDB TPC-H 性能对比测试报告 - v4.0 对比 v3.0

2.5.2.1 测试目的

对比 TiDB v4.0 和 v3.0 OLAP 场景下的性能。

因为 TiDB v4.0 中新引入了TiFlash 组件增强 TiDB HTAP 形态，本文的测试对象如下：

- v3.0 仅从 TiKV 读取数据。
- v4.0 仅从 TiKV 读取数据。
- v4.0 通过智能选择混合读取 TiKV、 TiFlash 的数据。

2.5.2.2 测试环境 (AWS EC2)

2.5.2.2.1 硬件配置

服务类型	EC2 类型	实例数
PD	m5.xlarge	3
TiDB	c5.4xlarge	2
TiKV & TiFlash	i3.4xlarge	3
TPC-H	m5.xlarge	1

2.5.2.2.2 软件版本

服务类型	软件版本
PD	3.0、4.0
TiDB	3.0、4.0
TiKV	3.0、4.0
TiFlash	4.0
tiup-bench	0.2

2.5.2.2.3 配置参数

v3.0

v3.0 的 TiDB 和 TiKV 均为默认参数配置。

变量配置

```
set global tidb_distsql_scan_concurrency = 30;
set global tidb_projection_concurrency = 16;
set global tidb_hashagg_partial_concurrency = 16;
set global tidb_hashagg_final_concurrency = 16;
set global tidb_hash_join_concurrency = 16;
set global tidb_index_lookup_concurrency = 16;
set global tidb_index_lookup_join_concurrency = 16;
```

v4.0

v4.0 的 TiDB 为默认参数配置。

TiKV 配置

```
readpool.storage.use-unified-pool: false
readpool.coprocessor.use-unified-pool: true
```

PD 配置

```
replication.enable-placement-rules: true
```

TiFlash 配置

```
logger.level: "info"
learner_config.log-level: "info"
```

变量配置

> **注意：**
>
> 部分参数为 SESSION 变量。建议所有查询都在当前 SESSION 中执行。

```
set global tidb_allow_batch_cop = 1;
set session tidb_opt_distinct_agg_push_down = 1;
set global tidb_distsql_scan_concurrency = 30;
set global tidb_projection_concurrency = 16;
set global tidb_hashagg_partial_concurrency = 16;
set global tidb_hashagg_final_concurrency = 16;
set global tidb_hash_join_concurrency = 16;
set global tidb_index_lookup_concurrency = 16;
set global tidb_index_lookup_join_concurrency = 16;
```

2.5.2.2.4 测试方案

硬件准备

为了避免 TiKV 和 TiFlash 争抢磁盘和 I/O 资源，把 EC2 配置的两个 NVMe SSD 盘分别挂载为 `/data1` 及 `/data2`，把 TiKV 的部署至 `/data1`，TiFlash 部署至 `/data2`。

测试过程

1. 通过 TiUP 部署 TiDB v4.0 和 v3.0。
2. 通过 TiUP 的 bench 工具导入 TPC-H 10G 数据。
    - 执行以下命令将数据导入 v3.0：

    ```
    tiup bench tpch prepare \
    --host ${tidb_v3_host} --port ${tidb_v3_port} --db tpch_10 \
    --sf 10 \
    --analyze --tidb_build_stats_concurrency 8 --tidb_distsql_scan_concurrency 30
    ```

    - 执行以下命令将数据导入 v4.0：

    ```
    tiup bench tpch prepare \
      --host ${tidb_v4_host} --port ${tidb_v4_port} --db tpch_10 --password ${password} \
      --sf 10 \
      --tiflash \
      --analyze --tidb_build_stats_concurrency 8 --tidb_distsql_scan_concurrency 30
    ```

3. 运行 TPC-H 的查询。
    1. 下载 TPC-H 的 SQL 查询文件：

    ```
    git clone https://github.com/pingcap/tidb-bench.git && cd tpch/queries
    ```

    2. 查询并记录耗时。
        - 对于 TiDB v3.0，使用 MySQL 客户端连接到 TiDB，然后执行查询，记录 v3.0 查询耗时。
        - 对于 TiDB v4.0，使用 MySQL 客户端连接到 TiDB，再根据测试的形态，选择其中一种操作：

- 设置 `set @@session.tidb_isolation_read_engines = 'tikv,tidb';` 后，再执行查询，记录 v4.0 仅从 TiKV 读取数据的查询耗时。
- 设置 `set @@session.tidb_isolation_read_engines = 'tikv,tiflash,tidb';` 后，再执行查询，记录 v4.0 通过智能选择从 TiKV 和 TiFlash 混合读取数据的查询耗时。

4. 提取整理耗时数据。

2.5.2.2.5 测试结果

> **注意：**
>
> 本测试所执行 SQL 语句对应的表只有主键，没有建立二级索引。因此以下测试结果为无索引结果。

Query ID	v3.0	v4.0 TiKV Only	v4.0 TiKV / TiFlash Automatically
1	7.78s	7.45s	2.09s
2	3.15s	1.71s	1.71s
3	6.61s	4.10s	4.05s
4	2.98s	2.56s	1.87s
5	20.35s	5.71s	8.53s
6	4.75s	2.44s	0.39s
7	7.97s	3.72s	3.59s
8	5.89s	3.22s	8.59s
9	34.08s	11.87s	15.41s
10	4.83s	2.75s	3.35s
11	3.98s	1.60s	1.59s
12	5.63s	3.40s	1.03s
13	5.41s	4.56s	4.02s
14	5.19s	3.10s	0.78s
15	10.25s	1.82s	1.26s
16	2.46s	1.51s	1.58s
17	23.76s	12.38s	8.52s
18	17.14s	16.38s	16.06s
19	5.70s	4.59s	3.20s
20	4.98s	1.89s	1.29s
21	11.12s	6.23s	6.26s
22	4.49s	3.05s	2.31s

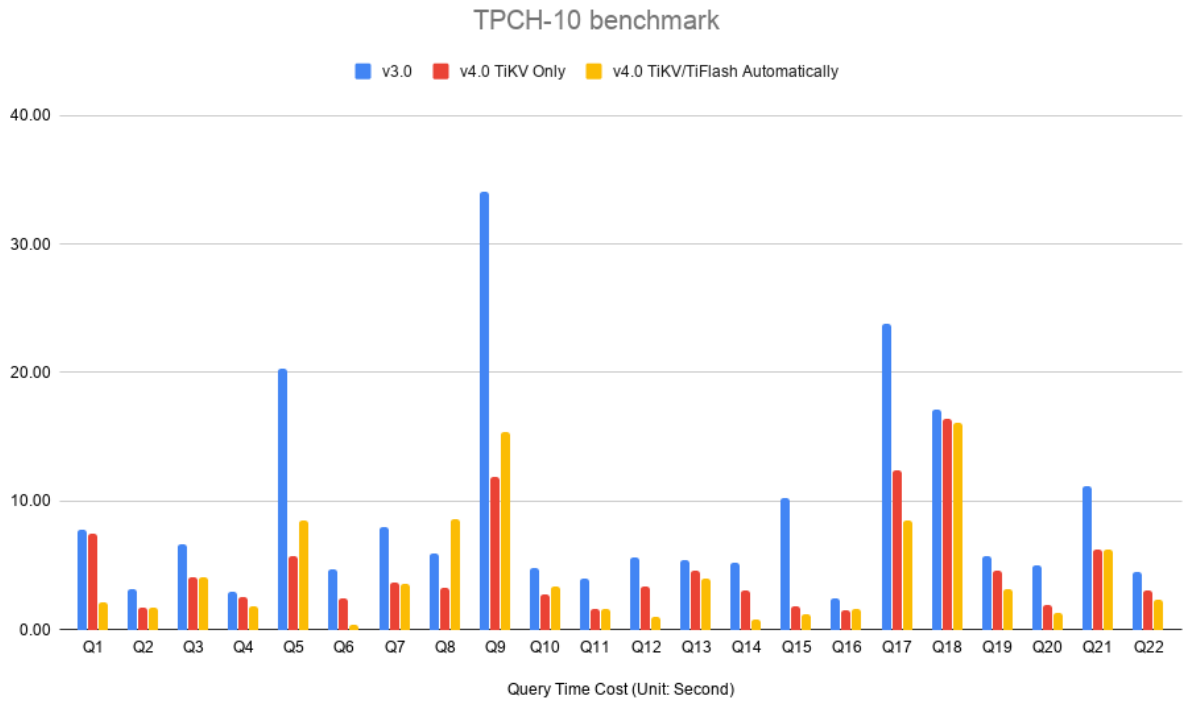


图 5: TPC-H

以上性能图中蓝色为 v3.0，红色为 v4.0（仅从 TiKV 读），黄色为 v4.0（从 TiKV、TiFlash 智能选取），纵坐标是查询的处理时间。纵坐标越低，表示性能越好。

- v4.0（仅从 TiKV 读取数据），即 TiDB 仅会从 TiKV 中读取数据。将该结果与 v3.0 的结果对比可得知，TiDB、TiKV 升级至 4.0 版本后，TPC-H 性能得到的提升幅度。
- v4.0（从 TiKV、TiFlash 智能选取），即 TiDB 优化器会自动根据代价估算选择是否使用 TiFlash 副本。将该结果与 v3.0 的结果对比可得知，在 v4.0 完整的 HTAP 形态下，TPC-H 性能得到的提升幅度。

2.5.3 TiDB TPC-C 性能对比测试报告 - v4.0 对比 v3.0

2.5.3.1 测试目的

测试对比 TiDB v4.0 和 v3.0 OLTP 场景下的性能。

2.5.3.2 测试环境 (AWS EC2)

2.5.3.2.1 硬件配置

服务类型	EC2 类型	实例数
PD	m5.xlarge	3

服务类型	EC2 类型	实例数
TiKV	i3.4xlarge	3
TiDB	c5.4xlarge	3
TPC-C	m5.4xlarge	1

2.5.3.2.2 软件版本

服务类型	软件版本
PD	3.0、4.0
TiDB	3.0、4.0
TiKV	3.0、4.0
BenchmarkSQL	无

2.5.3.2.3 配置参数

TiDB v3.0 参数配置

```
log.level: "error"
performance.max-procs: 20
prepared-plan-cache.enabled: true
tikv-client.max-batch-wait-time: 2000000
```

TiKV v3.0 参数配置

```
storage.scheduler-worker-pool-size: 5
raftstore.store-pool-size: 3
raftstore.apply-pool-size: 3
rocksdb.max-background-jobs: 3
raftdb.max-background-jobs: 3
raftdb.allow-concurrent-memtable-write: true
server.grpc-concurrency: 6
readpool.storage.normal-concurrency: 10
readpool.coprocessor.normal-concurrency: 5
```

TiDB v4.0 参数配置

```
log.level: "error"
performance.max-procs: 20
prepared-plan-cache.enabled: true
tikv-client.max-batch-wait-time: 2000000
```

TiKV v4.0 参数配置

```
storage.scheduler-worker-pool-size: 5
raftstore.store-pool-size: 3
```

```
raftstore.apply-pool-size: 3
rocksdb.max-background-jobs: 3
raftdb.max-background-jobs: 3
raftdb.allow-concurrent-memtable-write: true
server.request-batch-enable-cross-command: false
server.grpc-concurrency: 6
readpool.unified.min-thread-count: 5
readpool.unified.max-thread-count: 20
readpool.storage.normal-concurrency: 10
pessimistic-txn.pipelined: true
```

全局变量配置

```
set global tidb_hashagg_final_concurrency=1;
set global tidb_hashagg_partial_concurrency=1;
set global tidb_disable_txn_auto_retry=0;
```

2.5.3.2.4 测试方案

1. 通过 TiUP 部署 TiDB v4.0 和 v3.0。
2. 通过 BenchmarkSQL 导入 TPC-C 5000 Warehouse 数据。
    1. 编译 BenchmarkSQL：

        ```
        git clone https://github.com/pingcap/benchmarksql && cd benchmarksql && ant
        ```

    2. 进入 run 目录，根据实际情况编辑 `props.mysql` 文件，调整 conn、warehouses、loadWorkers、terminals、runMins 配置项。
    3. 运行 `runSQL.sh ./props.mysql sql.mysql/tableCreates.sql` 命令。
    4. 运行 `runSQL.sh ./props.mysql sql.mysql/indexCreates.sql` 命令。
    5. 运行 MySQL client 并对每个表执行 `analyze table` 语句。
3. 运行 `runBenchmark.sh ./props.mysql` 命令。
4. 从结果中提取 New Order 的 tpmC 的数据。

2.5.3.2.5 测试结果

v4.0 比 v3.0 在 TPC-C 性能上提升了 50%。

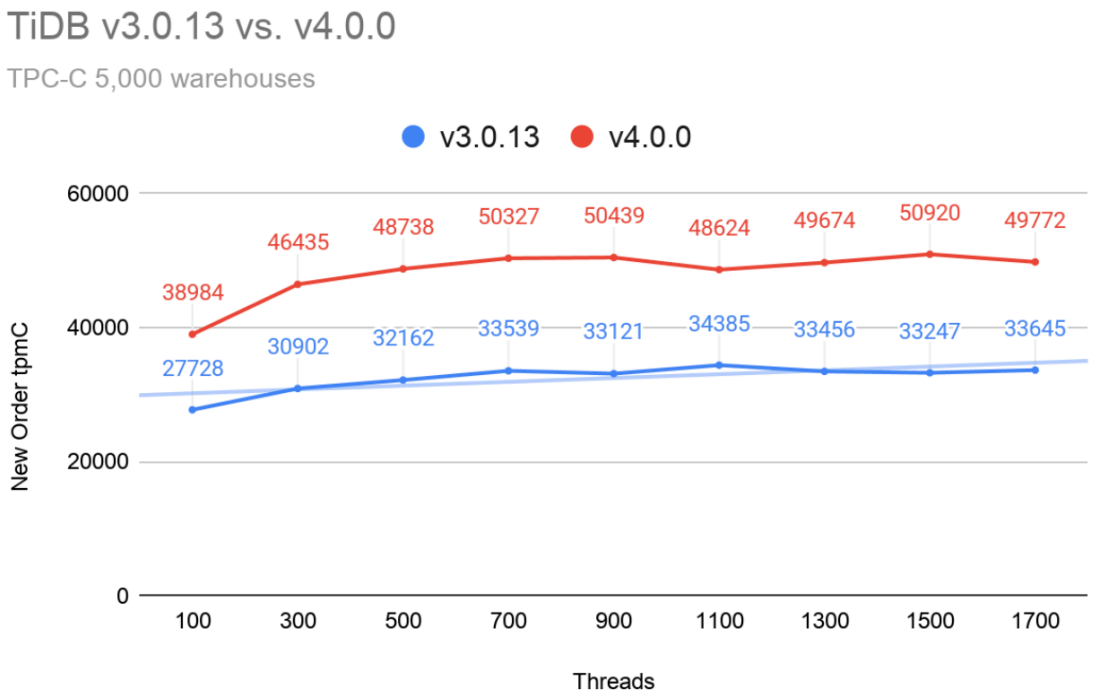


图 6: TPC-C

2.5.4 线上负载与 ADD INDEX 相互影响测试

2.5.4.1 测试目的

测试 OLTP 场景下，ADD INDEX 与线上负载的相互影响。

2.5.4.2 测试版本、时间、地点

TiDB 版本：v3.0.1

时间：2019 年 7 月

地点：北京

2.5.4.3 测试环境

测试在 Kubernetes 集群上进行，部署了 3 个 TiDB 实例，3 个 TiKV 实例和 3 个 PD 实例。

2.5.4.3.1 版本信息

组件	GitHash
TiDB	9e4e8da3c58c65123db5f26409759fe1847529f8
TiKV	4151dc8878985df191b47851d67ca21365396133

组件	GitHash
PD	811ce0b9a1335d1b2a049fd97ef9e186f1c9efc1

Sysbench 版本：1.0.17

2.5.4.3.2 TiDB 参数配置

TiDB、 TiKV 和 PD 均使用 TiDB Operator 默认配置。

2.5.4.3.3 集群拓扑

机器 IP	部署实例
172.31.8.8	Sysbench
172.31.7.69, 172.31.5.152, 172.31.11.133	PD
172.31.4.172, 172.31.1.155, 172.31.9.210	TiKV
172.31.7.80, 172.31.5.163, 172.31.11.123	TiDB

2.5.4.3.4 使用 Sysbench 模拟线上负载

使用 Sysbench 向集群导入 1 张表，200 万行数据。

执行如下命令导入数据：

```
sysbench oltp_common \
    --threads=16 \
    --rand-type=uniform \
    --db-driver=mysql \
    --mysql-db=sbtest \
    --mysql-host=$tidb_host \
    --mysql-port=$tidb_port \
    --mysql-user=root \
    prepare --tables=1 --table-size=2000000
```

执行如下命令测试数据：

```
sysbench $testname \
    --threads=$threads \
    --time=300000 \
    --report-interval=15 \
    --rand-type=uniform \
    --rand-seed=$RANDOM \
    --db-driver=mysql \
    --mysql-db=sbtest \
    --mysql-host=$tidb_host \
    --mysql-port=$tidb_port \
```

```
    --mysql-user=root \
    run --tables=1 --table-size=2000000
```

2.5.4.4 测试方案 1：ADD INDEX 目标列被频繁 Update

1. 开始 oltp_read_write 测试。
2. 与步骤 1 同时，使用 alter table sbtest1 add index c_idx(c) 添加索引。
3. 在步骤 2 结束，即索引添加完成时，停止步骤 1 的测试。
4. 获取 alter table ... add index 的运行时间、sysbench 在该时间段内的平均 TPS 和 QPS 作为指标。
5. 逐渐增大 tidb_ddl_reorg_worker_cnt 和 tidb_ddl_reorg_batch_size 两个参数的值，重复步骤 1-4。

2.5.4.4.1 测试结果

无 ADD INDEX 时 oltp_read_write 的结果

sysbench TPS	sysbench QPS
350.31	6806

tidb_ddl_reorg_batch_size = 32

tidb_ddl_reorg_worker_cnt	add_index_durations(s)	sysbench TPS	sysbench QPS
1	402	338.4	6776
2	266	330.3	6001
4	174	288.5	5769
8	129	280.6	5612
16	90	263.5	5273
32	54	229.2	4583
48	57	230.1	4601

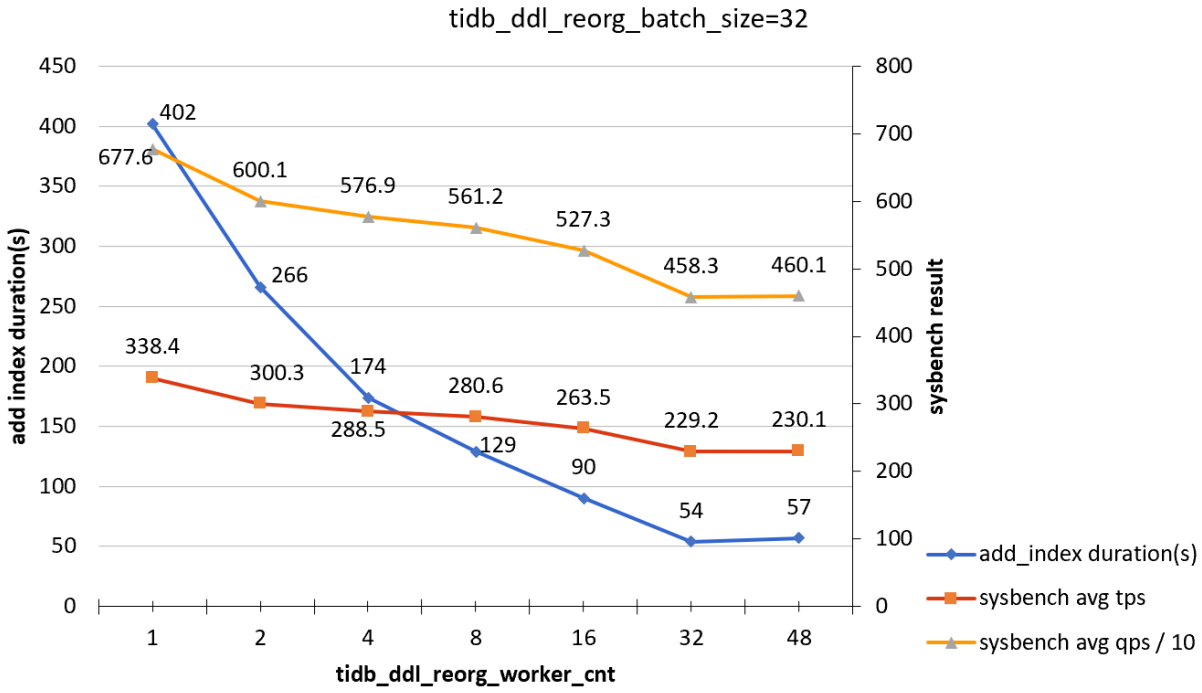


图 7: add-index-load-1-b32

```
tidb_ddl_reorg_batch_size = 64
```

tidb_ddl_reorg_worker_cnt	add_index_durations(s)	sysbench TPS	sysbench QPS
1	264	269.4	5388
2	163	266.2	5324
4	105	272.5	5430
8	78	262.5	5228
16	57	215.5	4308
32	42	185.2	3715
48	45	189.2	3794

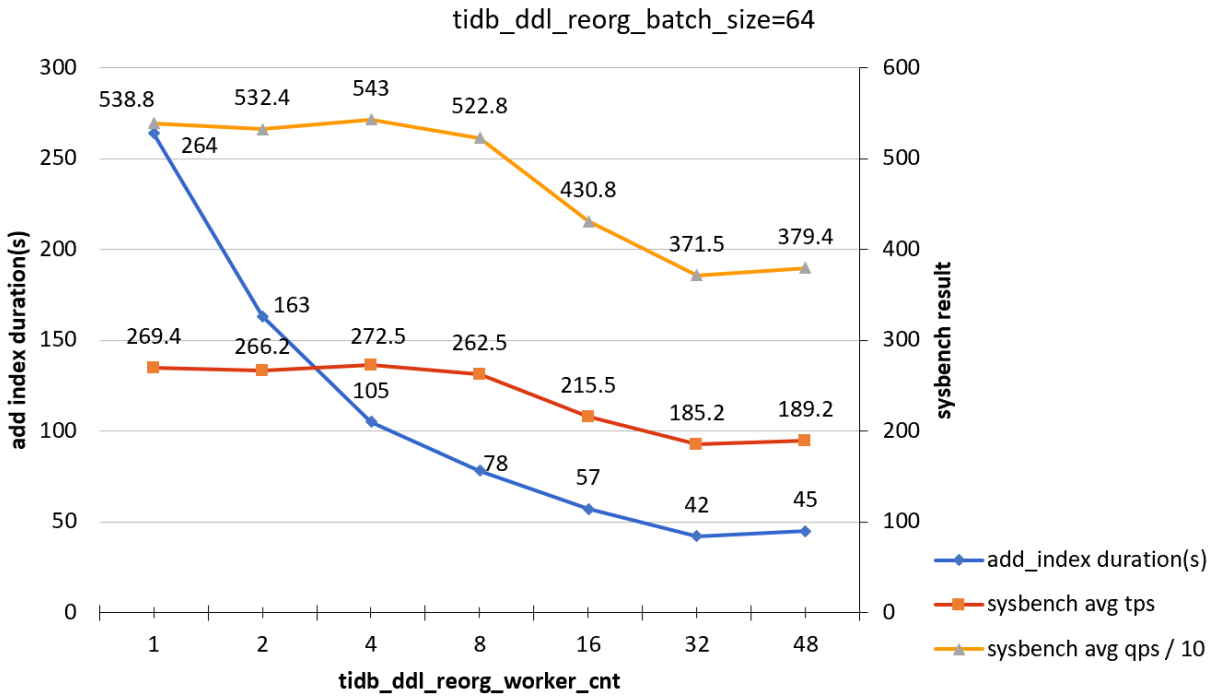


图 8: add-index-load-1-b64

`tidb_ddl_reorg_batch_size = 128`

tidb_ddl_reorg_worker_cnt	add_index_durations(s)	sysbench TPS	sysbench QPS
1	171	289.1	5779
2	110	274.2	5485
4	79	250.6	5011
8	51	246.1	4922
16	39	171.1	3431
32	35	130.8	2629
48	35	120.5	2425

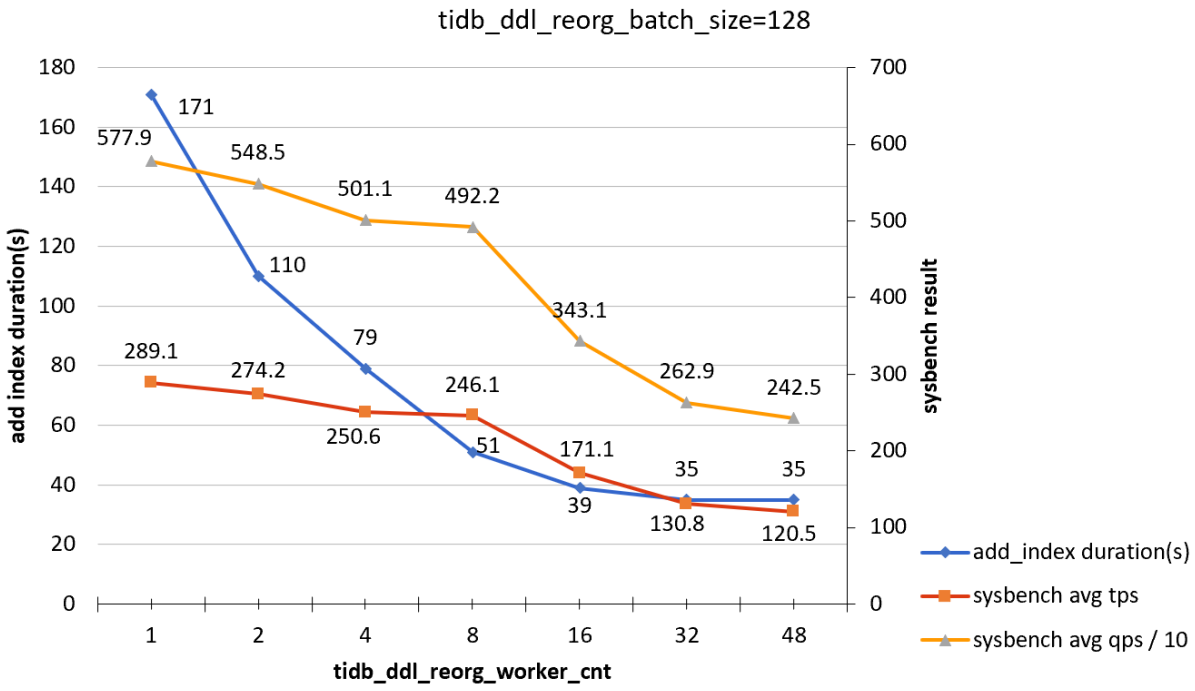


图 9: add-index-load-1-b128

```
tidb_ddl_reorg_batch_size = 256
```

tidb_ddl_reorg_worker_cnt	add_index_durations(s)	sysbench TPS	sysbench QPS
1	145	283.0	5659
2	96	282.2	5593
4	56	236.5	4735
8	45	194.2	3882
16	39	149.3	2893
32	36	113.5	2268
48	33	86.2	1715

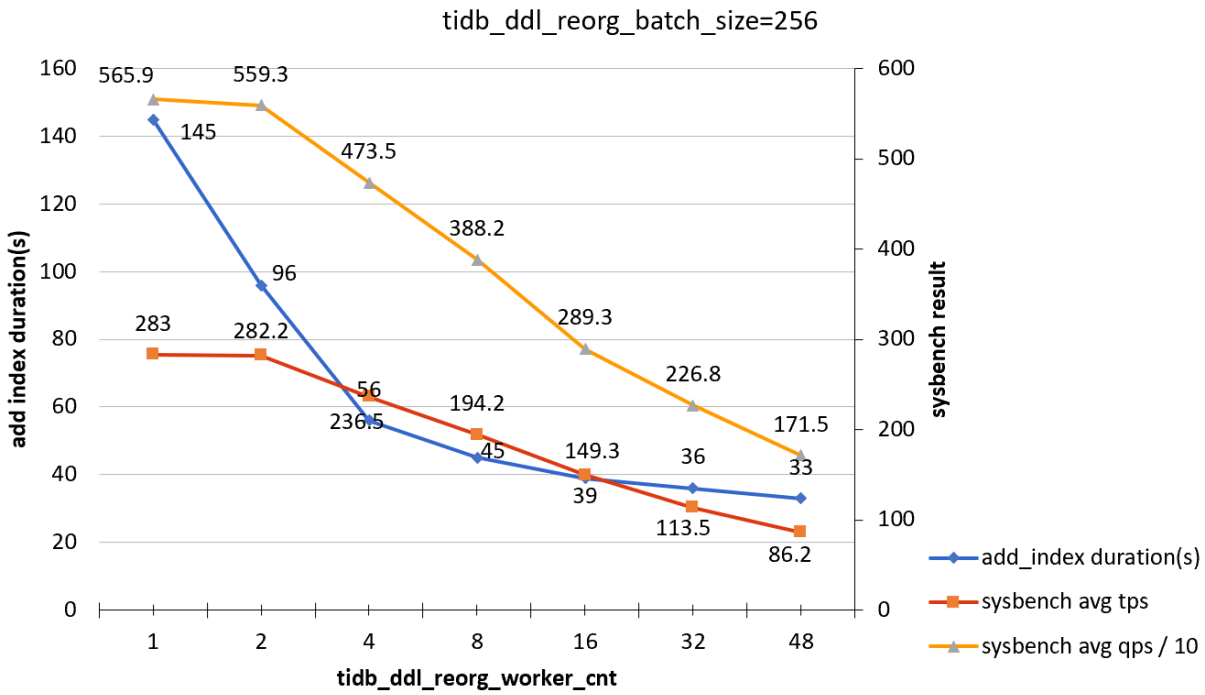


图 10: add-index-load-1-b256

`tidb_ddl_reorg_batch_size = 512`

tidb_ddl_reorg_worker_cnt	add_index_durations(s)	sysbench TPS	sysbench QPS
1	135	257.8	5147
2	78	252.8	5053
4	49	222.7	4478
8	36	145.4	2904
16	33	109	2190
32	33	72.5	1503
48	33	54.2	1318

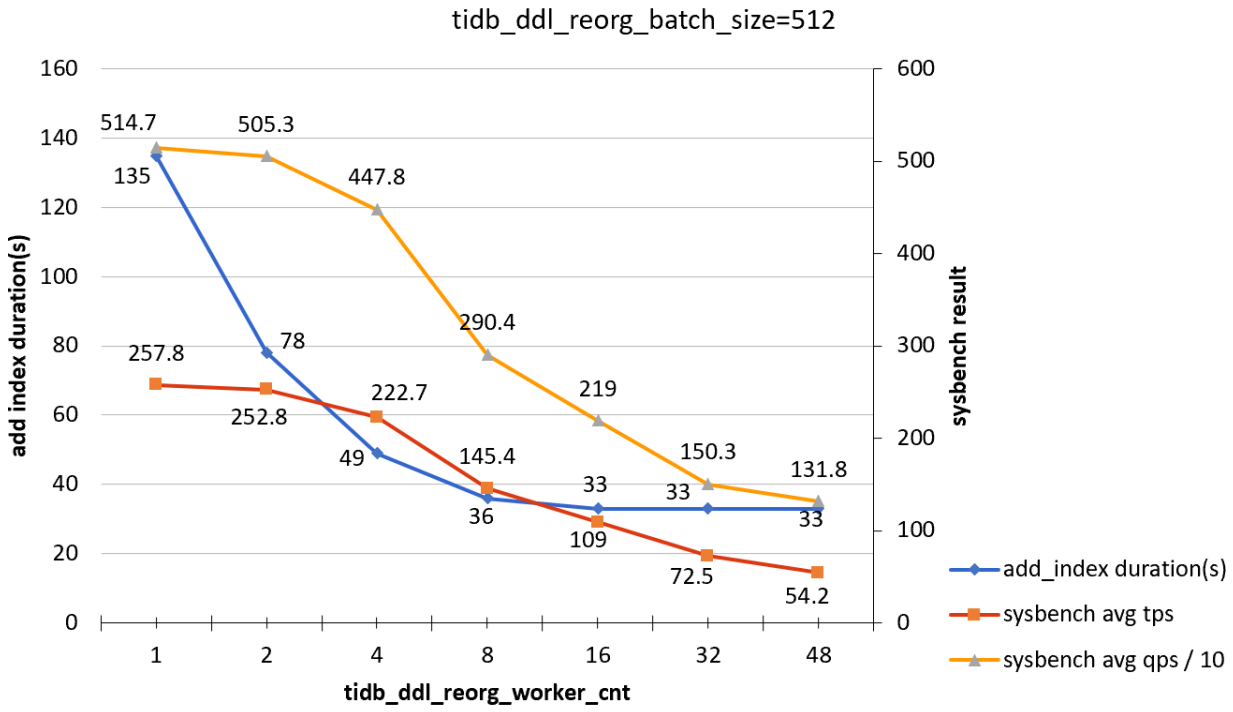


图 11: add-index-load-1-b512

`tidb_ddl_reorg_batch_size = 1024`

tidb_ddl_reorg_worker_cnt	add_index_durations(s)	sysbench TPS	sysbench QPS
1	111	244.3	4885
2	78	228.4	4573
4	54	168.8	3320
8	39	123.8	2475
16	36	59.6	1213
32	42	93.2	1835
48	51	115.7	2261

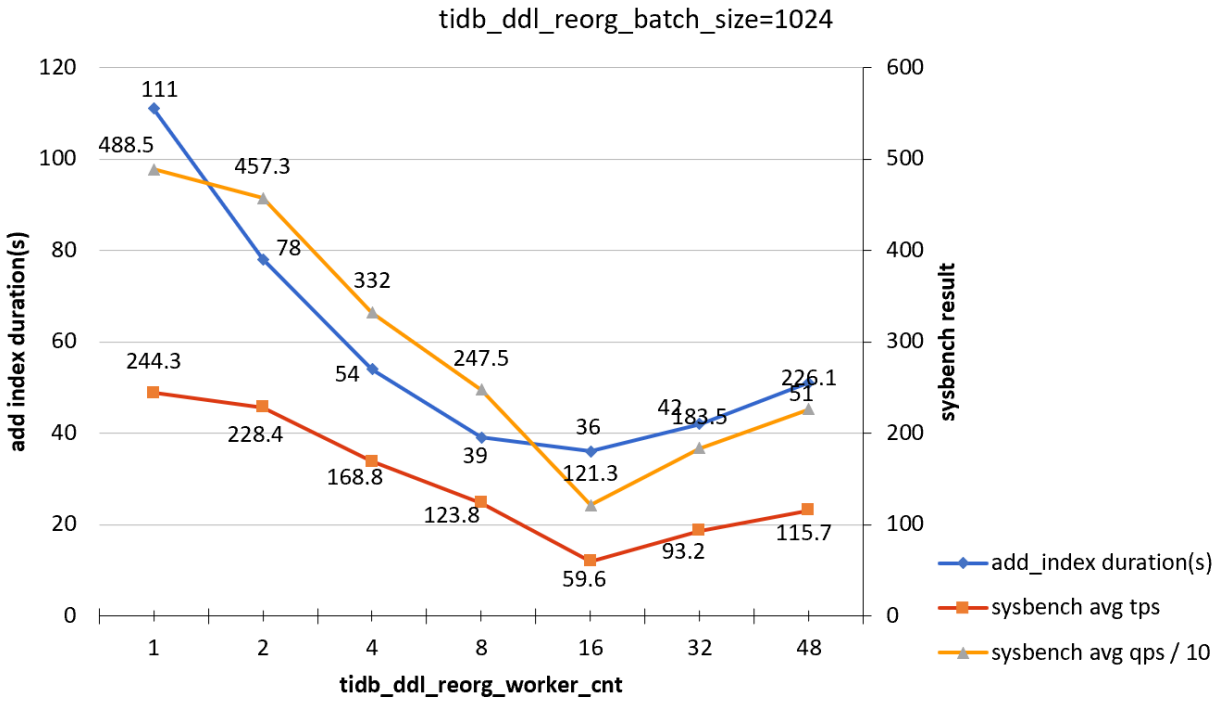


图 12: add-index-load-1-b1024

```
tidb_ddl_reorg_batch_size = 2048
```

tidb_ddl_reorg_worker_cnt	add_index_durations(s)	sysbench TPS	sysbench QPS
1	918	243.3	4855
2	1160	209.9	4194
4	342	185.4	3707
8	1316	151.0	3027
16	795	30.5	679
32	1130	26.69	547
48	893	27.5	552

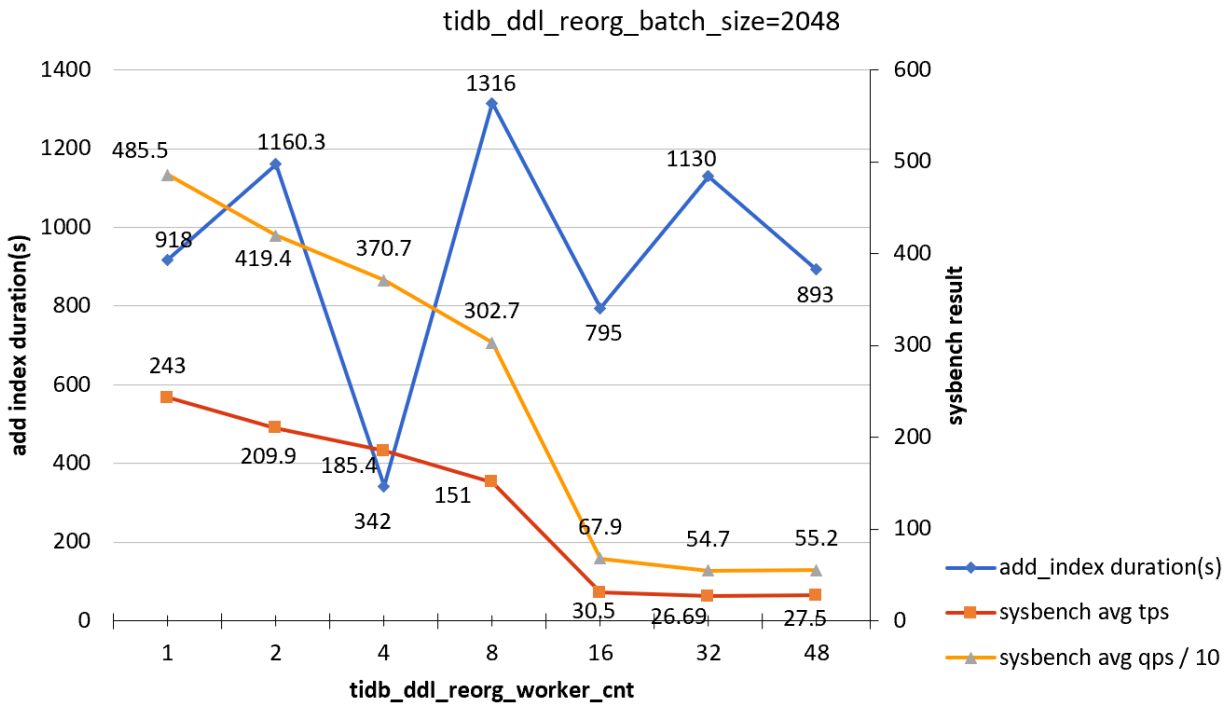


图 13: add-index-load-1-b2048

`tidb_ddl_reorg_batch_size = 4096`

tidb_ddl_reorg_worker_cnt	add_index_durations(s)	sysbench TPS	sysbench QPS
1	3042	200.0	4001
2	3022	203.8	4076
4	858	195.5	3971
8	3015	177.1	3522
16	837	143.8	2875
32	942	114	2267
48	187	54.2	1416

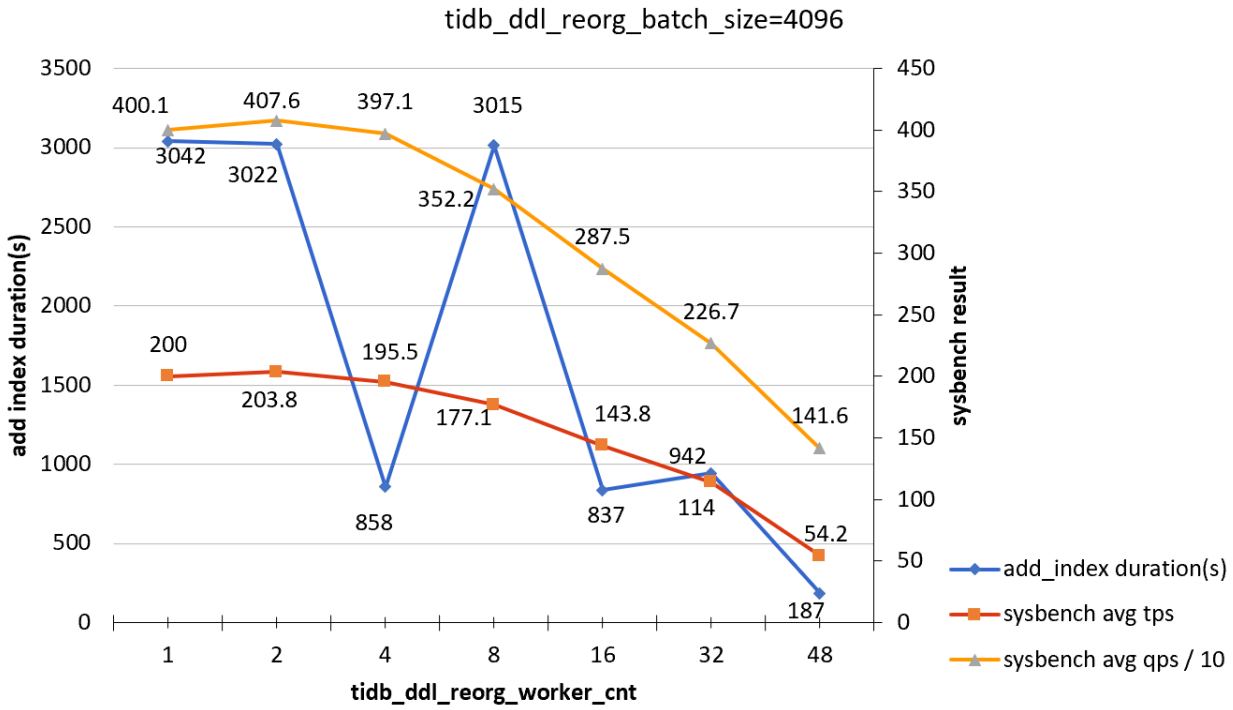


图 14: add-index-load-1-b4096

2.5.4.4.2 测试结论

若 ADD INDEX 的目标列正在进行较为频繁的写操作（本测试涉及列的 UPDATE、INSERT 和 DELETE），默认 ADD INDEX 配置对系统的线上负载有比较明显的影响，该影响主要来源于 ADD INDEX 与 Column Update 并发进行造成的写冲突，系统的表现反应在：

- 随着两个参数的逐渐增大，TiKV_prewrite_latch_wait_duration 有明显的升高，造成写入变慢。
- tidb_ddl_reorg_worker_cnt 与 tidb_ddl_reorg_batch_size 非常大时，admin show ddl 命令可以看到 DDL job 的多次重试（例如 Write conflict, txnStartTS 410327455965380624 is stale [try again later], ErrCount:38, SnapshotVersion:410327228136030220），此时 ADD INDEX 会持续非常久才能完成。

2.5.4.5 测试方案 2：ADD INDEX 目标列不涉及写入（仅查询）

1. 开始 oltp_read_only 测试。
2. 与步骤 1 同时，使用 alter table sbtest1 add index c_idx(c) 添加索引。
3. 在步骤 2 结束，即索引添加完成时，停止步骤 1。
4. 获取 alter table ... add index 的运行时间、sysbench 在该时间段内的平均 TPS 和 QPS 作为指标。
5. 逐渐增大 tidb_ddl_reorg_worker_cnt 和 tidb_ddl_reorg_batch_size 两个参数，重复步骤 1-4。

2.5.4.5.1 测试结果

无 ADD INDEX 时 oltp_read_only 结果

sysbench TPS	sysbench QPS
550.9	8812.8

`tidb_ddl_reorg_batch_size = 32`

tidb_ddl_reorg_worker_cnt	add_index_durations(s)	sysbench TPS	sysbench QPS
1	376	548.9	8780
2	212	541.5	8523
4	135	538.6	8549
8	114	536.7	8393
16	77	533.9	8292
32	46	533.4	8103
48	46	532.2	8074

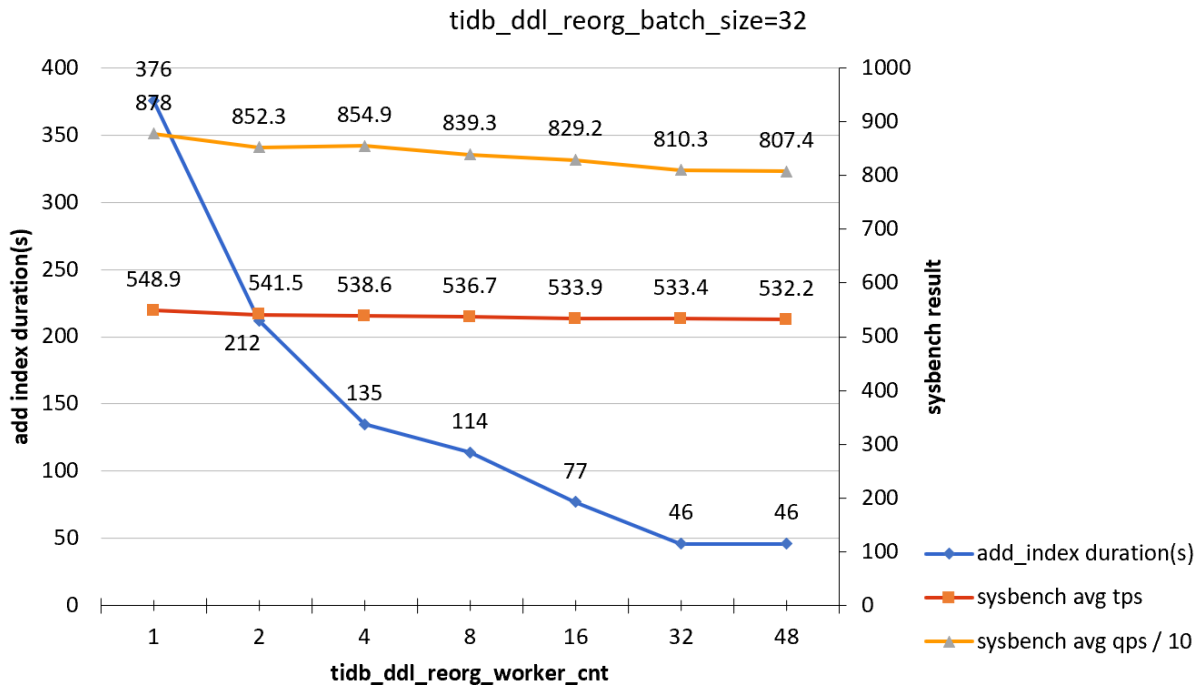


图 15: add-index-load-2-b32

`tidb_ddl_reorg_batch_size = 1024`

tidb_ddl_reorg_worker_cnt	add_index_durations(s)	sysbench TPS	sysbench QPS
1	91	536.8	8316
2	52	533.9	8165
4	40	522.4	7947
8	36	510	7860

tidb_ddl_reorg_worker_cnt	add_index_durations(s)	sysbench TPS	sysbench QPS
16	33	485.5	7704
32	31	467.5	7516
48	30	562.1	7442

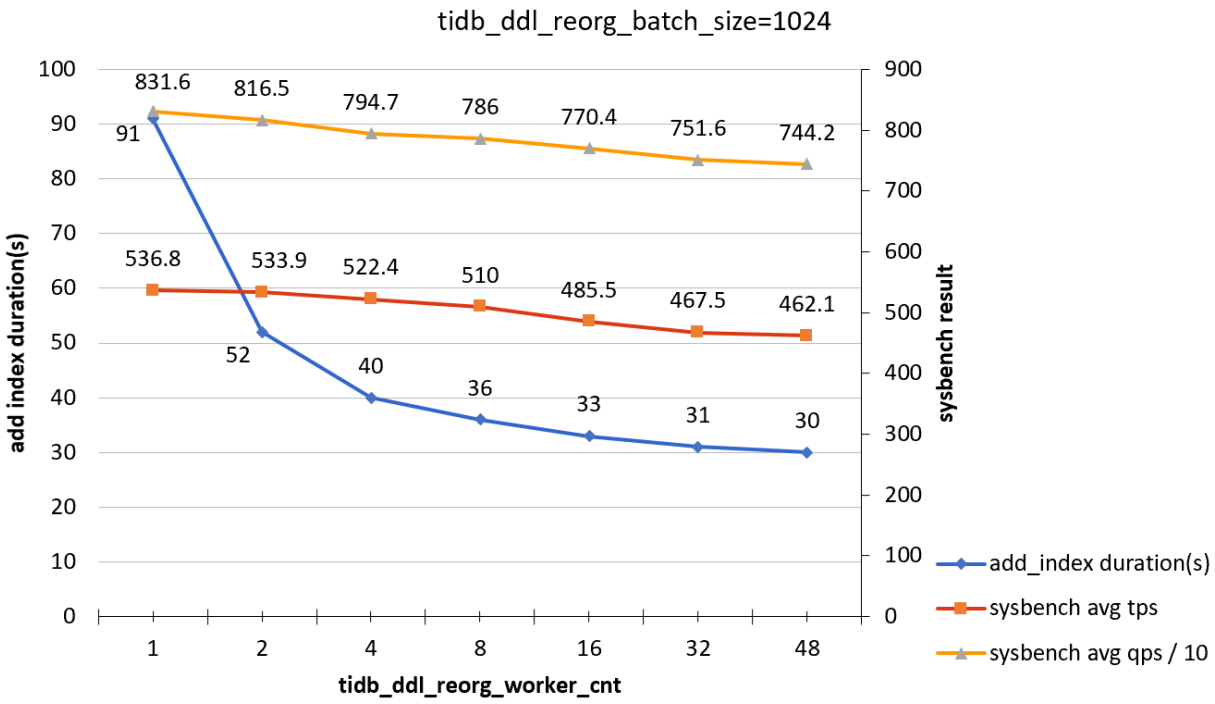


图 16: add-index-load-2-b1024

```
tidb_ddl_reorg_batch_size = 4096
```

tidb_ddl_reorg_worker_cnt	add_index_durations(s)	sysbench TPS	sysbench QPS
1	103	502.2	7823
2	63	486.5	7672
4	52	467.4	7516
8	39	452.5	7302
16	35	447.2	7206
32	30	441.9	7057
48	30	440.1	7004

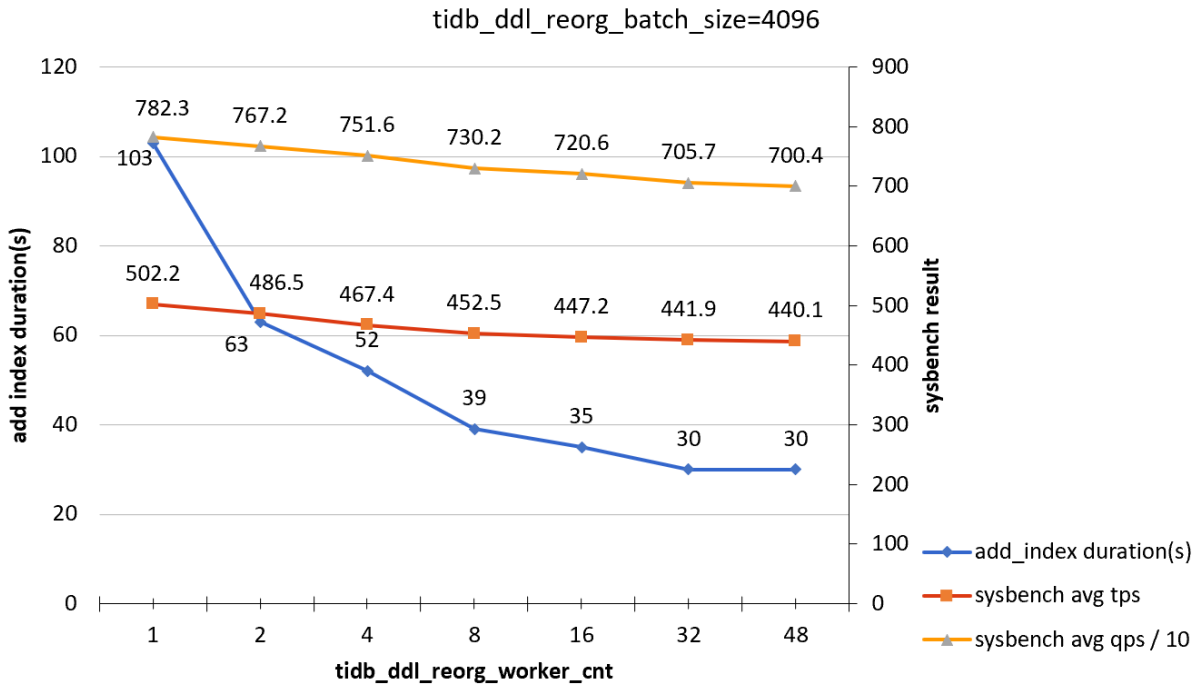


图 17: add-index-load-2-b4096

2.5.4.5.2 测试结论

ADD INDEX 的目标列仅有查询负载时，ADD INDEX 对负载的影响不明显。

2.5.4.6 测试方案 3：集群负载不涉及 ADD INDEX 目标列

1. 开始 oltp_read_write 测试。
2. 与步骤 1 同时，使用 alter table test add index pad_idx(pad) 添加索引。
3. 在步骤 2 结束，即索引添加完成时，停止步骤 1 的测试。
4. 获取 alter table ... add index 的运行时间、sysbench 在该时间段内的平均 TPS 和 QPS 作为指标。
5. 逐渐增大 tidb_ddl_reorg_worker_cnt 和 tidb_ddl_reorg_batch_size 两个参数，重复步骤 1-4。

2.5.4.6.1 测试结果

无 ADD INDEX 时 oltp_read_write 的结果

sysbench TPS	sysbench QPS
350.31	6806

tidb_ddl_reorg_batch_size = 32

tidb_ddl_reorg_worker_cnt	add_index_durations(s)	sysbench TPS	sysbench QPS
1	372	350.4	6892
2	207	344.2	6700
4	140	343.1	6672
8	121	339.1	6579
16	76	340	6607
32	42	343.1	6695
48	42	333.4	6454

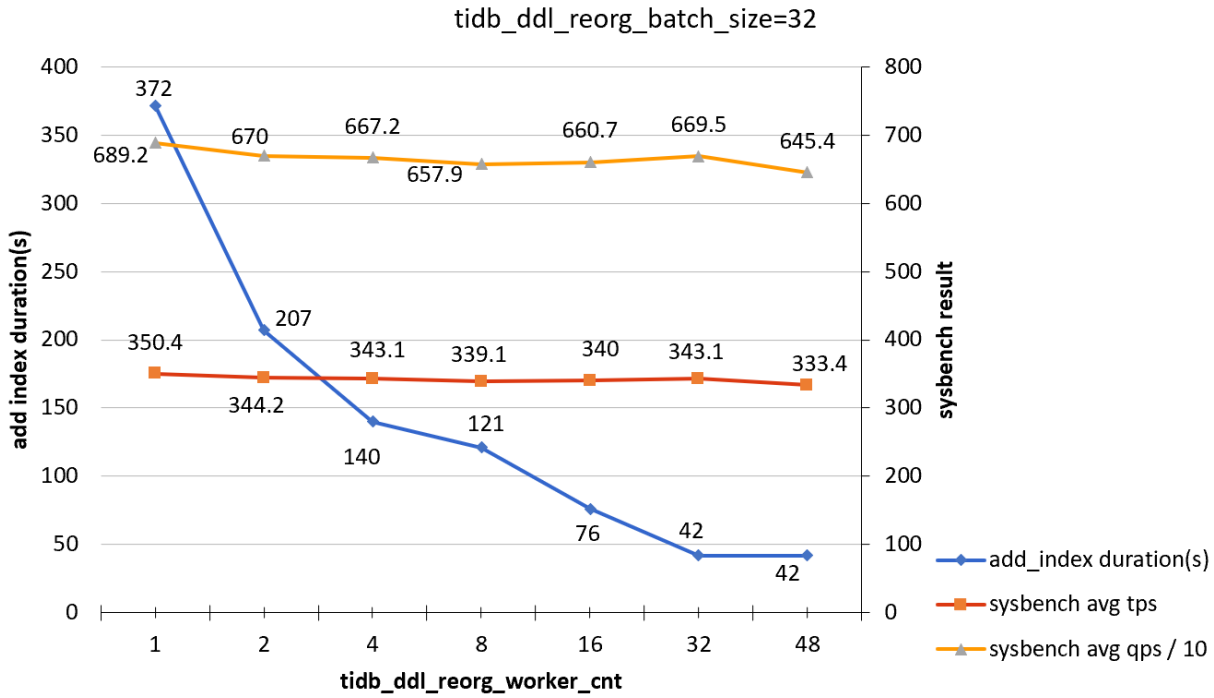


图 18: add-index-load-3-b32

```
tidb_ddl_reorg_batch_size = 1024
```

tidb_ddl_reorg_worker_cnt	add_index_durations(s)	sysbench TPS	sysbench QPS
1	94	352.4	6794
2	50	332	6493
4	45	330	6456
8	36	325.5	6324
16	32	312.5	6294
32	32	300.6	6017
48	31	279.5	5612

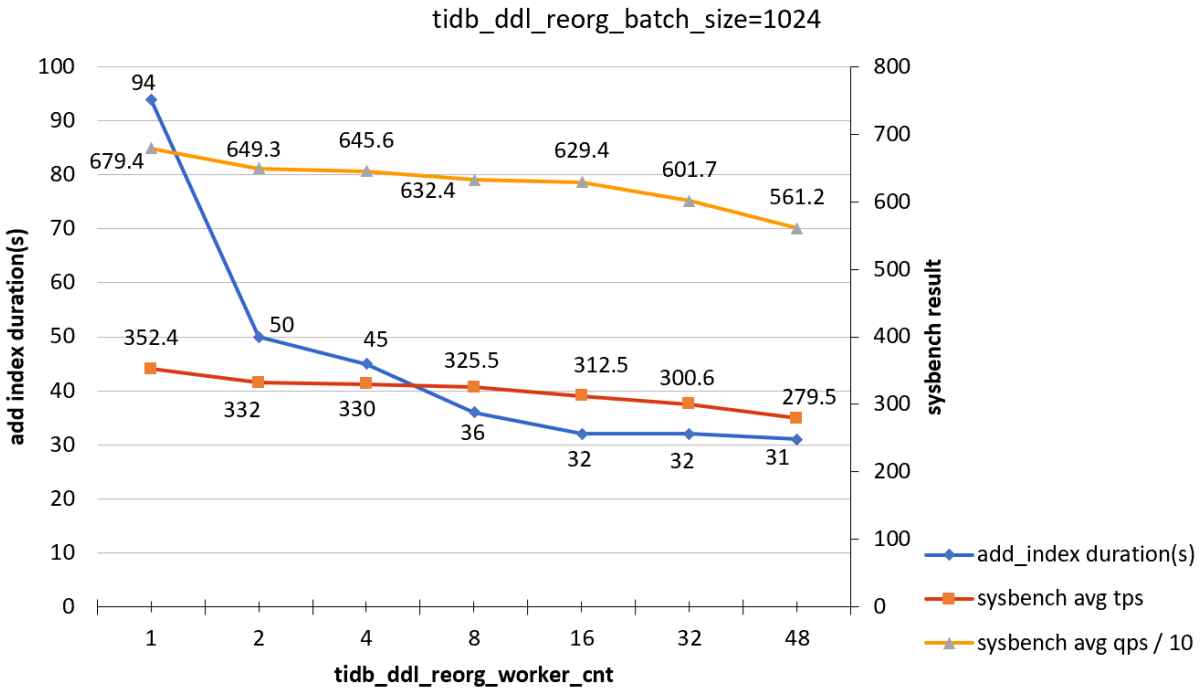


图 19: add-index-load-3-b1024

`tidb_ddl_reorg_batch_size = 4096`

tidb_ddl_reorg_worker_cnt	add_index_durations(s)	sysbench TPS	sysbench QPS
1	116	325.5	6324
2	65	312.5	6290
4	50	300.6	6017
8	37	279.5	5612
16	34	250.4	5365
32	32	220.2	4924
48	33	214.8	4544

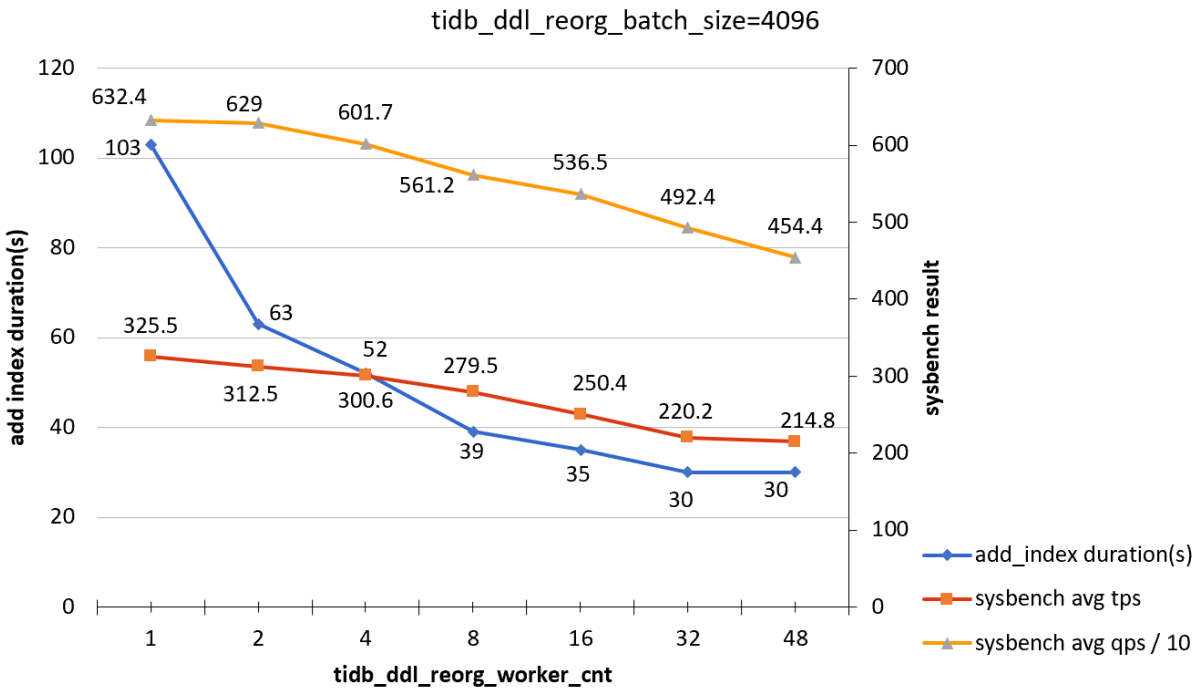


图 20: add-index-load-3-b4096

2.5.4.6.2 测试结论

ADD INDEX 的目标列与负载无关时，ADD INDEX 对负载的影响不明显。

2.5.4.7 总结

- 当 ADD INDEX 的目标列被频繁更新（包含 UPDATE、INSERT 和 DELETE）时，默认配置会造成较为频繁的写冲突，使得在线负载较大；同时 ADD INDEX 也可能由于不断地重试，需要很长的时间才能完成。在本次测试中，将 tidb_ddl_reorg_worker_cnt 和 tidb_ddl_reorg_batch_size 的乘积调整为默认值的 1/32（例如 tidb_ddl_reorg_worker_cnt = 4，tidb_ddl_reorg_batch_size = 256）可以取得较好的效果。
- 当 ADD INDEX 的目标列仅涉及查询负载，或者与线上负载不直接相关时，可以直接使用默认配置。

2.6 与 MySQL 兼容性对比

- TiDB 高度兼容 MySQL 5.7 协议、MySQL 5.7 常用的功能及语法。MySQL 5.7 生态中的系统工具（PHPMyAdmin、Navicat、MySQL Workbench、mysqldump、Mydumper/Myloader）、客户端等均适用于 TiDB。
- 但 TiDB 尚未支持一些 MySQL 功能，可能的原因如下：
  - 有更好的解决方案，例如 JSON 取代 XML 函数。
  - 目前对这些功能的需求度不高，例如存储流程和函数。
  - 一些功能在分布式系统上的实现难度较大。
- 除此以外，TiDB 不支持 MySQL 复制协议，但提供了专用工具用于与 MySQL 复制数据

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

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

2.6.1 不支持的功能特性

- 存储过程与函数
- 触发器
- 事件
- 自定义函数
- 外键约束 #18209
- 临时表 #1248
- 全文语法与索引 #1793
- 空间类型的函数（即 GIS/GEOMETRY）、数据类型和索引 #6347
- 非 ascii/latin1/binary/utf8/utf8mb4 的字符集
- SYS schema
- MySQL 追踪优化器
- XML 函数
- X-Protocol #1109
- Savepoints #6840
- 列级权限 #9766
- XA 语法（TiDB 内部使用两阶段提交，但并没有通过 SQL 接口公开）
- CREATE TABLE tblName AS SELECT stmt 语法 #4754
- CHECK TABLE 语法 #4673
- CHECKSUM TABLE 语法 #1895
- GET_LOCK 和 RELEASE_LOCK 函数 #14994

2.6.2 与 MySQL 有差异的特性详细说明

2.6.2.1 自增 ID

- TiDB 的自增列仅保证唯一，也能保证在单个 TiDB server 中自增，但不保证多个 TiDB server 中自增，不保证自动分配的值的连续性，建议不要将缺省值和自定义值混用，若混用可能会收 Duplicated Error 的错误信息。
- TiDB 可通过 tidb_allow_remove_auto_inc 系统变量开启或者关闭允许移除列的 AUTO_INCREMENT 属性。删除列属性的语法是：ALTER TABLE MODIFY 或 ALTER TABLE CHANGE。

- TiDB 不支持添加列的 AUTO_INCREMENT 属性，移除该属性后不可恢复。

自增 ID 详情可参阅AUTO_INCREMENT。

> 注意：
>
> - tidb_allow_remove_auto_inc 要求版本号 >= v2.1.18 或者 >= v3.0.4。
> - 表的 AUTO_ID_CACHE 属性要求版本号 >= v3.0.14 或者 >= v3.1.2 或者 >= v4.0.0-rc.2。
> - 若创建表时没有指定主键时，TiDB 会使用 _tidb_rowid 来标识行，该数值的分配会和自增列（如果存在的话）共用一个分配器。如果指定了自增列为主键，则 TiDB 会用该列来标识行。因此会有以下的示例情况：

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

mysql> INSERT INTO t VALUES(),(),();
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0

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

2.6.2.2 Performance schema

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

2.6.2.3 查询计划

EXPLAIN/EXPLAIN FOR 输出格式、内容、权限设置与 MySQL 有比较大的差别，参见理解 TiDB 执行计划。

2.6.2.4 内建函数

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

2.6.2.5 DDL 的限制

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

- 不能在单条 ALTER TABLE 语句中完成多个操作。例如，不能在单个语句中添加多个列或索引，否则，可能会输出 Unsupported multi schema change 的错误。
- 不支持不同类型的索引 (HASH|BTREE|RTREE|FULLTEXT)。若指定了不同类型的索引，TiDB 会解析并忽略这些索引。
- 不支持添加/删除主键，除非开启了alter-primary-key 配置项。
- 不支持将字段类型修改为其超集，例如不支持从 INTEGER 修改为 VARCHAR，或者从 TIMESTAMP 修改为 DATETIME，否则可能输出的错误信息 Unsupported modify column: type %d not match origin %d。
- 更改/修改数据类型时，尚未支持“有损更改”，例如不支持从 BIGINT 更改为 INT。
- 更改/修改 DECIMAL 类型时，不支持更改精度。
- 更改/修改整数列时，不允许更改 UNSIGNED 属性。
- TiDB 中，ALGORITHM={INSTANT,INPLACE,COPY} 语法只作为一种指定，并不更改 ALTER 算法，详情参阅ALTER TABLE。
- 分区表支持 HASH 和 RANGE 分区类型。对于不支持的分区类型，TiDB 会返回 Warning: Unsupported partition type, treat as normal table。
- 分区表支持 ADD、DROP、TRUNCATE 操作，其他分区操作会被忽略。TiDB 不支持以下分区表语法：
  - PARTITION BY LIST
  - PARTITION BY KEY
  - SUBPARTITION
  - {CHECK|EXCHANGE|OPTIMIZE|REPAIR|IMPORT|DISCARD|REBUILD|REORGANIZE|COALESCE} PARTITION

2.6.2.6 ANALYZE TABLE

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

更多信息统计的差异请参阅ANALYZE TABLE。

2.6.2.7 SELECT 的限制

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

详情参见SELECT。

2.6.2.8 UPDATE 语句

详情参见UPDATE。

2.6.2.9 视图

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

2.6.2.10 存储引擎

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

2.6.2.11 SQL 模式

TiDB 支持大部分SQL 模式。不支持的 SQL 模式如下：

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

2.6.2.12 默认设置

- 字符集：
  - TiDB 默认：`utf8mb4`。
  - MySQL 5.7 默认：`latin1`。
  - MySQL 8.0 默认：`utf8mb4`。
- 排序规则：
  - TiDB 中 `utf8mb4` 字符集默认：`utf8mb4_bin`。
  - MySQL 5.7 中 `utf8mb4` 字符集默认：`utf8mb4_general_ci`。
  - MySQL 8.0 中 `utf8mb4` 字符集默认：`utf8mb4_0900_ai_ci`。
- `foreign_key_checks`：
  - TiDB 默认：`OFF`，且仅支持设置该值为 `OFF`。
  - MySQL 5.7 默认：`ON`。
- SQL mode：
  - TiDB 默认：`ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION`。
  - MySQL 5.7 默认与 TiDB 相同。
  - MySQL 8.0 默认 `ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO,NO_ENGINE_SUBSTITUTION`。
- `lower_case_table_names`：
  - TiDB 默认：`2`，且仅支持设置该值为 `2`。
  - MySQL 默认如下：
    * Linux 系统中该值为 0
    * Windows 系统中该值为 1
    * macOS 系统中该值为 2
- `explicit_defaults_for_timestamp`：
  - TiDB 默认：`ON`，且仅支持设置该值为 `ON`。
  - MySQL 5.7 默认：`OFF`。
  - MySQL 8.0 默认：`ON`。

2.6.2.13 日期时间处理的区别

2.6.2.13.1 时区

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

2.6.2.13.2 零月和零日

- 与 MySQL 一样，TiDB 默认启用了 `NO_ZERO_DATE` 和 `NO_ZERO_IN_DATE` 模式，但是 TiDB 与 MySQL 在处理这两个 SQL 模式有以下不同：
  - TiDB 在非严格模式下启用以上两个 SQL 模式，插入零月/零日/零日期不会给出警告，MySQL 则会给出对应的警告。
  - TiDB 在严格模式下，启用了 `NO_ZERO_DATE`，仍然能够插入零日期；如果启用了 `NO_ZERO_IN_DATE` 则无法插入零月/零日日期。MySQL 在严格模式下则都无法插入两种类型的日期。

2.6.2.14 类型系统

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

2.6.2.15 MySQL 弃用功能导致的不兼容问题

TiDB 不支持 MySQL 中标记为弃用的功能，包括：

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

2.7 使用限制

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

2.7.1 标识符长度限制

标识符类型	最大长度（字符）
Database	64
Table	64

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

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

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

2.7.3 单个 Database 的限制

类型	最大限制
Tables	unlimited

2.7.4 单个 Table 的限制

类型	最大限制
Columns	512
Indexs	64
Rows	unlimited
Size	unlimited
Partitions	1024

2.7.5 单行的限制

类型	最大限制
Size	默认为 6MB，可通过`txn-entry-size-limit`配置项调整

2.7.6 字符串类型限制

类型	最大限制
CHAR	256 字符

类型	最大限制
BINARY	256 字节
VARBINARY	65535 字节
VARCHAR	16383 字符
TEXT	6MB 字节
BLOB	6MB 字节

2.7.7 SQL Statements 的限制

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

2.8 TiDB 社区荣誉列表

TiDB 开发者社区以 SIG (Special Interest Group) 为单位管理组织开发者。每个模块都有其固定的 SIG 负责新功能开发、性能优化、稳定性保障等。如果你也想成为 TiDB 的开发者，加入感兴趣的 SIG、与一线工程师面对面讨论无疑是最好的方式。以下是截至 TiDB 4.0 GA，TiDB 社区贡献者列表及对应角色：

2.8.1 Committers

SIG name	GitHub ID
planner	winoros
transaction	jackysp
k8s	cofyc
scheduling	rleungx
Dashboard	breeswish
docs	lilin90
execution	qw4990
migrate	kennytm
tiup	lonng
web	wd0517
ddl	zimulala
transaction	bobotu
transaction	cfzjywxk
transaction	lysu
transaction	tiancaiamao
transaction	youjiali1995
transaction	coocood
transaction	imtbkcat
transaction	MyonKeminta
transaction	nrc

SIG name	GitHub ID
k8s	AstroProfundis
k8s	aylei
k8s	DanielZhangQD
k8s	gregwebs
k8s	jlerche
k8s	LinuxGit
k8s	onlymellb
k8s	qiffang
k8s	sdojjy
k8s	shuijing198799
k8s	tennix
k8s	weekface
scheduling	disksing
scheduling	hundundm
scheduling	lhy1024
scheduling	nolouch
scheduling	shafreeck
Dashboard	crazycs520
Dashboard	Deardrops
Dashboard	mapleFU
Dashboard	reafans
Dashboard	rleungx
docs	CaitinChen
docs	cofyc
docs	DanielZhangQD
docs	dcalvin
docs	jackysp
docs	ran-huang
docs	TomShawn
docs	toutdesuite
docs	WangXiangUSTC
migrate	3pointer
migrate	5kbpers
migrate	amyangfei
migrate	gmhdbjd
migrate	GregoryIan
migrate	july2993
migrate	leoppro
migrate	lichunzhu
migrate	overvenus
migrate	YuJuncen
tiup	AstroProfundis
tiup	july2993

SIG name	GitHub ID
tiup	nrc
tiup	birdstorm
tiup	breeswish
execution	Reminiscent
execution	wshwsh12
execution	zz-jason
web	g1eny0ung
web	YiniXu9506
ddl	AilinKid
ddl	bb7133
ddl	crazycs520
ddl	djshow832
ddl	lonng
ddl	winkyao
ddl	wjhuang2016
planner	eurekaka
planner	francis0407
planner	lzmhhh123
migrate	csuzhangxc

2.8.2 Reviewers

SIG name	GitHub ID
docs	anotherrachel
docs	aylei
docs	crazycs520
docs	ericsyh
docs	juliezhang1112
docs	morgo
docs	weekface
docs	YiniXu9506
execution	b41sh
execution	js00070
execution	mmyj
execution	shihongzhi
execution	tangwz
execution	tsthght
ddl	Deardrops
ddl	lysu
planner	imtbkcat
planner	lamxTyler
planner	SunRunAway

SIG name	GitHub ID
planner	wjhuang2016
planner	XuHuaiyu
planner	zz-jason

2.8.3 Active Contributors

SIG name	GitHub ID
k8s	cwen0
k8s	mikechengwei
k8s	shonge
k8s	xiaojingchen
k8s	shinnosuke-okada
scheduling	mantuliu
docs	3pointer
docs	amyangfei
docs	csuzhangxc
docs	Deardrops
docs	gmhdbjd
docs	huangxiuyan
docs	IzabelWang
docs	july2993
docs	kissmydb
docs	kolbe
docs	lamxTyler
docs	lance6716
docs	lichunzhu
docs	liubo0127
docs	lysu
docs	superlzs0476
docs	tangenta
docs	tennix
docs	tiancaiamao
docs	xiaojingchen
docs	Yisaer
docs	zhouqiang-cl
docs	zimulala
tiup	c4pt0r
tiup	YangKeao
tiup	qinzuoyan
execution	AerysNan
execution	AndrewDi
execution	ekalinin

SIG name	GitHub ID
execution	erjiaqing
execution	hey-kong
execution	jacklightChen
execution	k-ye
execution	pingyu
execution	Rustin-Liu
execution	spongedu
execution	TennyZhuang
execution	xiekeyi98
ddl	reafans
ddl	Rustin-Liu
ddl	spongedu
planner	Deardrops
planner	foreyes
planner	lonng
planner	SeaRise
planner	tiancaiamao
planner	wshwsh12

3 快速上手

3.1 TiDB 数据库快速上手指南

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

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

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

3.1.1 部署本地测试集群

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

> **注意：**
>
> 由于部分 TiDB 组件尚未发布支持 Apple M1 芯片的版本，暂不支持在使用 Apple M1 芯片的本地 Mac 机器上使用 `tiup playground` 命令。

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

1. 下载并安装 TiUP。

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

2. 声明全局环境变量。

> **注意：**
>
> TiUP 安装完成后会提示对应 `profile` 文件的绝对路径。在执行以下 `source` 命令前，需要根据 `profile` 文件的实际位置修改命令。

```
source .bash_profile
```

3. 在当前 session 执行以下命令启动集群。

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

    ```
    tiup playground
    ```

    - 也可以指定 TiDB 版本以及各组件实例个数，命令类似于：

    ```
    tiup playground v4.0.16 --db 2 --pd 3 --kv 3 --monitor
    ```

    上述命令会在本地下载并启动某个版本的集群（例如 v4.0.16），`--monitor` 表示同时部署监控组件。最新版本可以通过执行 `tiup list tidb` 来查看。运行结果将显示集群的访问方式：

    ```
    CLUSTER START SUCCESSFULLY, Enjoy it ^-^
    To connect TiDB: mysql --host 127.0.0.1 --port 4000 -u root
    To connect TiDB: mysql --host 127.0.0.1 --port 4001 -u root
    To view the dashboard: http://127.0.0.1:2379/dashboard
    To view the monitor: http://127.0.0.1:9090
    ```

> **注意：**

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

4. 新开启一个 session 以访问 TiDB 数据库。

    - 使用 TiUP `client` 连接 TiDB：

    ```
    tiup client
    ```

    - 也可使用 MySQL 客户端连接 TiDB：

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

5. 通过 http://127.0.0.1:9090 访问 TiDB 的 Prometheus 管理界面。
6. 通过 http://127.0.0.1:2379/dashboard 访问TiDB Dashboard 页面，默认用户名为 `root`，密码为空。
7. （可选）将数据加载到 TiFlash 进行分析。
8. 测试完成之后，可以通过执行以下步骤来清理集群：

    1. 通过按下 ctrl + c 键停掉进程。
    2. 执行以下命令：

    ```
    tiup clean --all
    ```

注意：

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

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

1. 下载并安装 TiUP。

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

2. 声明全局环境变量。

> **注意：**
>
> TiUP 安装完成后会提示对应 `profile` 文件的绝对路径。在执行以下 `source` 命令前，需要根据 `profile` 文件的实际位置修改命令。

```
source .bash_profile
```

3. 在当前 session 执行以下命令启动集群。

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

    ```
    tiup playground
    ```

    - 也可以指定 TiDB 版本以及各组件实例个数，命令类似于：

    ```
    tiup playground v4.0.16 --db 2 --pd 3 --kv 3 --monitor
    ```

    上述命令会在本地下载并启动某个版本的集群（例如 v4.0.16），`--monitor` 表示同时部署监控组件。最新版本可以通过执行 `tiup list tidb` 来查看。运行结果将显示集群的访问方式：

    ```
    CLUSTER START SUCCESSFULLY, Enjoy it ^-^
    To connect TiDB: mysql --host 127.0.0.1 --port 4000 -u root
    To connect TiDB: mysql --host 127.0.0.1 --port 4001 -u root
    To view the dashboard: http://127.0.0.1:2379/dashboard
    To view the monitor: http://127.0.0.1:9090
    ```

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

4. 新开启一个 session 以访问 TiDB 数据库。

    - 使用 TiUP `client` 连接 TiDB：

    ```
    tiup client
    ```

    - 也可使用 MySQL 客户端连接 TiDB：

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

5. 通过 http://127.0.0.1:9090 访问 TiDB 的 Prometheus 管理界面。

6. 通过 http://127.0.0.1:2379/dashboard 访问TiDB Dashboard 页面，默认用户名为 root，密码为空。

7. （可选）将数据加载到 TiFlash 进行分析。
8. 测试完成之后，可以通过执行以下步骤来清理集群：
    1. 通过按下 ctrl + c 键停掉进程。
    2. 执行以下命令：

```
tiup clean --all
```

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

3.1.2 在单机上模拟部署生产环境集群

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

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

3.1.2.1 准备环境

准备一台部署主机，确保其软件满足需求：

- 推荐安装 CentOS 7.3 及以上版本
- Linux 操作系统开放外网访问，用于下载 TiDB 及相关软件安装包

最小规模的 TiDB 集群拓扑：

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

实例	个数	IP	配置
TiKV	3	10.0.1.1 10.0.1.1 10.0.1.1	避免端口和目录冲突
TiDB	1	10.0.1.1	默认端口全局目录配置
PD	1	10.0.1.1	默认端口全局目录配置
TiFlash	1	10.0.1.1	默认端口全局目录配置
Monitor	1	10.0.1.1	默认端口全局目录配置

部署主机软件和环境要求：

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

3.1.2.2 实施部署

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

1. 下载并安装 TiUP：

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

2. 声明全局环境变量：

> 注意：
> TiUP 安装完成后会提示对应 `profile` 文件的绝对路径。在执行以下 `source` 命令前，需要根据 `profile` 文件的实际位置修改命令。

```
source .bash_profile
```

3. 安装 TiUP 的 cluster 组件：

```
tiup cluster
```

4. 如果机器已经安装 TiUP cluster，需要更新软件版本：

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

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

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

```
service sshd restart
```

6. 创建并启动集群

   按下面的配置模板，编辑配置文件，命名为 topo.yaml，其中：

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

   配置模板如下：

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

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

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

pd_servers:
 - host: 10.0.1.1

tidb_servers:
 - host: 10.0.1.1

tikv_servers:
 - host: 10.0.1.1
   port: 20160
   status_port: 20180
   config:
```

```
    server.labels: { host: "logic-host-1" }

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

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

tiflash_servers:
 - host: 10.0.1.1

monitoring_servers:
 - host: 10.0.1.1

grafana_servers:
 - host: 10.0.1.1
```

7. 执行集群部署命令：

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

- 参数 <cluster-name> 表示设置集群名称
- 参数 <tidb-version> 表示设置集群版本，可以通过 tiup list tidb 命令来查看当前支持部署的 TiDB 版本

按照引导，输入 y 及 root 密码，来完成部署：

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

8. 启动集群：

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

9. 访问集群：

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

```
yum -y install mysql
```

- 访问 TiDB 数据库，密码为空：

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

- 访问 TiDB 的 Grafana 监控：

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

- 访问 TiDB 的 Dashboard：

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

- 执行以下命令确认当前已经部署的集群列表：

```
tiup cluster list
```

- 执行以下命令查看集群的拓扑结构和状态：

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

3.1.3 探索更多

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

> 注意：
>
> TiDB、TiUP 及 TiDB Dashboard 默认会收集使用情况信息，并将这些信息分享给 PingCAP 用于改善产品。若要了解所收集的信息详情及如何禁用该行为，请参见遥测。

3.2 SQL 基本操作

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

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

3.2.1 分类

SQL 语言通常按照功能划分成以下的 4 个部分：

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

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

3.2.2 查看、创建和删除数据库

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

使用 `SHOW DATABASES` 语句查看系统中数据库列表：

```
SHOW DATABASES;
```

使用名为 `mysql` 的数据库：

```
USE mysql;
```

使用 `SHOW TABLES` 语句查看数据库中的所有表。例如：

```
SHOW TABLES FROM mysql;
```

使用 `CREATE DATABASE` 语句创建数据库。语法如下：

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

例如，要创建一个名为 `samp_db` 的数据库，可使用以下语句：

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

添加 `IF NOT EXISTS` 可防止发生错误。

使用 `DROP DATABASE` 语句删除数据库。例如：

```
DROP DATABASE samp_db;
```

3.2.3 创建、查看和删除表

使用 CREATE TABLE 语句创建表。语法如下：

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

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

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

使用 SHOW CREATE 语句查看建表语句，即 DDL。例如：

```
SHOW CREATE TABLE person;
```

使用 DROP TABLE 语句删除表。例如：

```
DROP TABLE person;
```

3.2.4 创建、查看和删除索引

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

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

或者：

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

对于值唯一的列，可以创建唯一索引。例如：

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

或者：

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

使用 SHOW INDEX 语句查看表内所有索引：

```
SHOW INDEX FROM person;
```

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

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

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

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

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

3.2.5 记录的增删改

使用 INSERT 语句向表内插入表记录。例如：

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

使用 INSERT 语句向表内插入包含部分字段数据的表记录。例如：

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

使用 UPDATE 语句向表内修改表记录的部分字段数据。例如：

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

使用 DELETE 语句向表内删除部分表记录。例如：

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

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

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

3.2.6 查询数据

使用 SELECT 语句检索表内数据。例如：

```
SELECT * FROM person;
```

在 SELECT 后面加上要查询的列名。例如：

```
SELECT name FROM person;
+------+
| name |
+------+
| tom  |
+------+
```

使用 WHERE 子句，对所有记录进行是否符合条件的筛选后再返回。例如：

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

常用的 DCL 功能是创建或删除用户，和对用户权限的管理。

3.2.7 创建、授权和删除用户

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

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

授权用户 tiuser 可检索数据库 samp_db 内的表：

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

查询用户 tiuser 的权限：

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

删除用户 tiuser：

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

4 部署集群

4.1 TiDB 软件和硬件环境建议配置

TiDB 作为一款开源分布式 NewSQL 数据库，可以很好的部署和运行在 Intel 架构服务器环境、ARM 架构的服务器环境及主流虚拟化环境，并支持绝大多数的主流硬件网络。作为一款高性能数据库系统，TiDB 支持主流的 Linux 操作系统环境。

4.1.1 Linux 操作系统版本要求

Linux 操作系统平台	版本
Red Hat Enterprise Linux	7.3 及以上的 7.x 版本
CentOS	7.3 及以上的 7.x 版本
Oracle Enterprise Linux	7.3 及以上的 7.x 版本
Ubuntu LTS	16.04 及以上的版本

注意：

- TiDB 只支持 Red Hat 兼容内核 (RHCK) 的 Oracle Enterprise Linux，不支持 Oracle Enterprise Linux 提供的 Unbreakable Enterprise Kernel。
- TiDB 在 CentOS 7.3 的环境下进行过大量的测试，同时社区也有很多该操作系统部署的最佳实践，因此，建议使用 CentOS 7.3 以上的 7.x Linux 操作系统来部署 TiDB。
- 以上 Linux 操作系统可运行在物理服务器以及 VMware、KVM 及 XEN 主流虚拟化环境上。

- 目前尚不支持 Red Hat Enterprise Linux 8.0、 CentOS 8 Stream 和 Oracle Enterprise Linux 8.0，因为目前对这些平台的测试还在进行中。
- 不计划支持 CentOS 8 Linux，因为 CentOS 的上游支持已于 2021 年 12 月 31 日终止。
- TiDB 将不再支持 Ubuntu 16.04。强烈建议升级到 Ubuntu 18.04 或更高版本。

其他 Linux 操作系统版本（例如 Debian Linux 和 Fedora Linux）也许可以运行 TiDB，但尚未得到 TiDB 官方支持。

4.1.2 软件配置要求

4.1.2.1 中控机软件配置

软件	版本
sshpass	1.06 及以上
TiUP	0.6.2 及以上

注意：

中控机需要部署TiUP 软件来完成 TiDB 集群运维管理。

4.1.2.2 目标主机建议配置软件

软件	版本
sshpass	1.06 及以上
numa	2.0.12 及以上
tar	任意

4.1.3 服务器建议配置

TiDB 支持部署和运行在 Intel x86-64 架构的 64 位通用硬件服务器平台或者 ARM 架构的硬件服务器平台。对于开发，测试，及生产环境的服务器硬件配置（不包含操作系统 OS 本身的占用）有以下要求和建议：

4.1.3.1 开发及测试环境

组件	CPU	内存	本地存储	网络	实例数量 (最低要求)
TiDB	8 核 +	16 GB+	无特殊要求	千兆网卡	1（可与 PD 同机器）
PD	4 核 +	8 GB+	SAS, 200 GB+	千兆网卡	1（可与 TiDB 同机器）
TiKV	8 核 +	32 GB+	SSD, 200 GB+	千兆网卡	3

组件	CPU	内存	本地存储	网络	实例数量 (最低要求)
TiFlash	32 核 +	64 GB+	SSD, 200 GB+	千兆网卡	1
TiCDC	8 核 +	16 GB+	SAS, 200 GB+	千兆网卡	1

注意：

- 验证测试环境中的 TiDB 和 PD 可以部署在同一台服务器上。
- 如进行性能相关的测试，避免采用低性能存储和网络硬件配置，防止对测试结果的正确性产生干扰。
- TiKV 的 SSD 盘推荐使用 NVME 接口以保证读写更快。
- 如果仅验证功能，建议使用TiDB 数据库快速上手指南进行单机功能测试。
- TiDB 对于磁盘的使用以存放日志为主，因此在测试环境中对于磁盘类型和容量并无特殊要求。

4.1.3.2 生产环境

组件	CPU	内存	硬盘类型	网络	实例数量 (最低要求)
TiDB	16 核 +	32 GB+	SAS	万兆网卡（2 块最佳）	2
PD	4 核 +	8 GB+	SSD	万兆网卡（2 块最佳）	3
TiKV	16 核 +	32 GB+	SSD	万兆网卡（2 块最佳）	3
TiFlash	48 核 +	128 GB+	1 or more SSDs	万兆网卡（2 块最佳）	2
TiCDC	16 核 +	64 GB+	SSD	万兆网卡（2 块最佳）	2
监控	8 核 +	16 GB+	SAS	千兆网卡	1

注意：

- 生产环境中的 TiDB 和 PD 可以部署和运行在同服务器上，如对性能和可靠性有更高的要求，应尽可能分开部署。
- 生产环境强烈推荐使用更高的配置。
- TiKV 硬盘大小配置建议 PCI-E SSD 不超过 2 TB，普通 SSD 不超过 1.5 TB。
- TiFlash 支持多盘部署。
- TiFlash 数据目录的第一块磁盘推荐用高性能 SSD 来缓冲 TiKV 同步数据的实时写入，该盘性能应不低于 TiKV 所使用的磁盘，比如 PCI-E SSD。并且该磁盘容量建议不小于总容量的 10%，否则它可能成为这个节点的能承载的数据量的瓶颈。而其他磁盘可以根据需求部署多块普通 SSD，当然更好的 PCI-E SSD 硬盘会带来更好的性能。
- TiFlash 推荐与 TiKV 部署在不同节点，如果条件所限必须将 TiFlash 与 TiKV 部署在相同节点，则需要适当增加 CPU 核数和内存，且尽量将 TiFlash 与 TiKV 部署在不同的磁盘，以免互相干扰。

- TiFlash 硬盘总容量大致为：整个 TiKV 集群的需同步数据容量 / TiKV 副本数 * TiFlash 副本数。例如整体 TiKV 的规划容量为 1 TB、TiKV 副本数为 3、TiFlash 副本数为 2，则 TiFlash 的推荐总容量为 1024 GB / 3 * 2。用户可以选择同步部分表数据而非全部，具体容量可以根据需要同步的表的数据量具体分析。
- TiCDC 硬盘配置建议 1 TB+ PCIE-SSD。

4.1.4 网络要求

TiDB 作为开源分布式 NewSQL 数据库，其正常运行需要网络环境提供如下的网络端口配置要求，管理员可根据实际环境中 TiDB 组件部署的方案，在网络侧和主机侧开放相关端口：

组件	默认端口	说明
TiDB	4000	应用及 DBA 工具访问通信端口
TiDB	10080	TiDB 状态信息上报通信端口
TiKV	20160	TiKV 通信端口
TiKV	20180	TiKV 状态信息上报通信端口
PD	2379	提供 TiDB 和 PD 通信端口
PD	2380	PD 集群节点间通信端口
TiFlash	9000	TiFlash TCP 服务端口
TiFlash	8123	TiFlash HTTP 服务端口
TiFlash	3930	TiFlash RAFT 服务和 Coprocessor 服务端口
TiFlash	20170	TiFlash Proxy 服务端口
TiFlash	20292	Prometheus 拉取 TiFlash Proxy metrics 端口
TiFlash	8234	Prometheus 拉取 TiFlash metrics 端口
Pump	8250	Pump 通信端口
Drainer	8249	Drainer 通信端口
CDC	8300	CDC 通信接口
Prometheus	9090	Prometheus 服务通信端口
Node_exporter	9100	TiDB 集群每个节点的系统信息上报通信端口
Blackbox_exporter	9115	Blackbox_exporter 通信端口，用于 TiDB 集群端口监控
Grafana	3000	Web 监控服务对外服务和客户端 (浏览器) 访问端口
Alertmanager	9093	告警 web 服务端口
Alertmanager	9094	告警通信端口

4.1.5 客户端 Web 浏览器要求

TiDB 提供了基于 Grafana 的技术平台，对数据库集群的各项指标进行可视化展现。采用支持 Javascript 的微软 IE、Google Chrome、 Mozilla Firefox 的较新版本即可访问监控入口。

4.2 TiDB 环境与系统配置检查

本文介绍部署 TiDB 前的环境检查操作，以下各项操作按优先级排序。

4.2.1 在 TiKV 部署目标机器上添加数据盘 EXT4 文件系统挂载参数

生产环境部署，建议使用 EXT4 类型文件系统的 NVME 类型的 SSD 磁盘存储 TiKV 数据文件。这个配置方案为最佳实施方案，其可靠性、安全性、稳定性已经在大量线上场景中得到证实。

使用 root 用户登录目标机器，将部署目标机器数据盘格式化成 ext4 文件系统，挂载时添加 nodelalloc 和 noatime 挂载参数。nodelalloc 是必选参数，否则 TiUP 安装时检测无法通过；noatime 是可选建议参数。

> 注意：
>
> 如果你的数据盘已经格式化成 ext4 并挂载了磁盘，可先执行 umount /dev/nvme0n1p1 命令卸载，从编辑 /etc/fstab 文件步骤开始执行，添加挂载参数重新挂载即可。

以 /dev/nvme0n1 数据盘为例，具体操作步骤如下：

1. 查看数据盘。

```
fdisk -l
```

```
Disk /dev/nvme0n1: 1000 GB
```

2. 创建分区。

```
parted -s -a optimal /dev/nvme0n1 mklabel gpt -- mkpart primary ext4 1 -1
```

> 注意：
>
> 使用 lsblk 命令查看分区的设备号：对于 nvme 磁盘，生成的分区设备号一般为 nvme0n1p1；对于普通磁盘（例如 /dev/sdb），生成的的分区设备号一般为 sdb1。

3. 格式化文件系统。

```
mkfs.ext4 /dev/nvme0n1p1
```

4. 查看数据盘分区 UUID。

   本例中 nvme0n1p1 的 UUID 为 c51eb23b-195c-4061-92a9-3fad812cc12f。

```
lsblk -f
```

```
NAME    FSTYPE LABEL UUID                                 MOUNTPOINT
sda
├─sda1  ext4         237b634b-a565-477b-8371-6dff0c41f5ab /boot
├─sda2  swap         f414c5c0-f823-4bb1-8fdf-e531173a72ed
└─sda3  ext4         547909c1-398d-4696-94c6-03e43e317b60 /
```

```
sr0
nvme0n1
└─nvme0n1p1 ext4         c51eb23b-195c-4061-92a9-3fad812cc12f
```

5. 编辑 /etc/fstab 文件，添加 nodelalloc 挂载参数。

```
vi /etc/fstab
```

```
UUID=c51eb23b-195c-4061-92a9-3fad812cc12f /data1 ext4 defaults,nodelalloc,noatime 0 2
```

6. 挂载数据盘。

```
mkdir /data1 && \
mount -a
```

7. 执行以下命令，如果文件系统为 ext4，并且挂载参数中包含 nodelalloc，则表示已生效。

```
mount -t ext4
```

```
/dev/nvme0n1p1 on /data1 type ext4 (rw,noatime,nodelalloc,data=ordered)
```

4.2.2 检测及关闭系统 swap

TiDB 运行需要有足够的内存。如果内存不足，不建议使用 swap 作为内存不足的缓冲，因为这会降低性能。建议永久关闭系统 swap。

要永久关闭 swap，可执行以如下命令:

```
echo "vm.swappiness = 0">> /etc/sysctl.conf
swapoff -a && swapon -a
sysctl -p
```

> 注意：
>
> - 一起执行 swapoff -a 和 swapon -a 命令是为了刷新 swap，将 swap 里的数据转储回内存，并清空 swap 里的数据。不可省略 swappiness 设置而只执行 swapoff -a；否则，重启后 swap 会再次自动打开，使得操作失效。
> - 执行 sysctl -p 命令是为了在不重启的情况下使配置生效。

4.2.3 检测及关闭目标部署机器的防火墙

本段介绍如何关闭目标主机防火墙配置，因为在 TiDB 集群中，需要将节点间的访问端口打通才可以保证读写请求、数据心跳等信息的正常的传输。在普遍线上场景中，数据库到业务服务和数据库节点的网络联通都是在安全域内完成数据交互。如果没有特殊安全的要求，建议将目标节点的防火墙进行关闭。否则建议按照端口使用规则，将端口信息配置到防火墙服务的白名单中。

1. 检查防火墙状态（以 CentOS Linux release 7.7.1908 (Core) 为例）

```
sudo firewall-cmd --state
sudo systemctl status firewalld.service
```

2. 关闭防火墙服务

```
sudo systemctl stop firewalld.service
```

3. 关闭防火墙自动启动服务

```
sudo systemctl disable firewalld.service
```

4. 检查防火墙状态

```
sudo systemctl status firewalld.service
```

4.2.4 检测及安装 NTP 服务

TiDB 是一套分布式数据库系统，需要节点间保证时间的同步，从而确保 ACID 模型的事务线性一致性。目前解决授时的普遍方案是采用 NTP 服务，可以通过互联网中的 pool.ntp.org 授时服务来保证节点的时间同步，也可以使用离线环境自己搭建的 NTP 服务来解决授时。

采用如下步骤检查是否安装 NTP 服务以及与 NTP 服务器正常同步：

1. 执行以下命令，如果输出 running 表示 NTP 服务正在运行：

```
sudo systemctl status ntpd.service
```

```
ntpd.service - Network Time Service
Loaded: loaded (/usr/lib/systemd/system/ntpd.service; disabled; vendor preset: disabled)
Active: active (running) since 一 2017-12-18 13:13:19 CST; 3s ago
```

   - 若返回报错信息 Unit ntpd.service could not be found.，请尝试执行以下命令，以查看与 NTP 进行时钟同步所使用的系统配置是 chronyd 还是 ntpd：

```
sudo systemctl status chronyd.service
```

```
chronyd.service - NTP client/server
Loaded: loaded (/usr/lib/systemd/system/chronyd.service; enabled; vendor preset:
    ↪ enabled)
Active: active (running) since Mon 2021-04-05 09:55:29 EDT; 3 days ago
```

若发现系统既没有配置 chronyd 也没有配置 ntpd，则表示系统尚未安装任一服务。此时，应先安装其中一个服务，并保证它可以自动启动，默认使用 ntpd。

如果你使用的系统配置是 `chronyd`，请直接执行步骤 3。

2. 执行 ntpstat 命令检测是否与 NTP 服务器同步：

> **注意：**
>
> Ubuntu 系统需安装 ntpstat 软件包。

```
ntpstat
```

- 如果输出 synchronised to NTP server，表示正在与 NTP 服务器正常同步：

```
synchronised to NTP server (85.199.214.101) at stratum 2
time correct to within 91 ms
polling server every 1024 s
```

- 以下情况表示 NTP 服务未正常同步：

```
unsynchronised
```

- 以下情况表示 NTP 服务未正常运行：

```
Unable to talk to NTP daemon. Is it running?
```

3. 执行 chronyc tracking 命令查看 Chrony 服务是否与 NTP 服务器同步。

> **注意：**
>
> 该操作仅适用于使用 Chrony 的系统，不适用于使用 NTPd 的系统。

```
chronyc tracking
```

- 如果该命令返回结果为 Leap status : Normal，则代表同步过程正常。

```
Reference ID    : 5EC69F0A (ntp1.time.nl)
Stratum         : 2
Ref time (UTC)  : Thu May 20 15:19:08 2021
System time     : 0.000022151 seconds slow of NTP time
Last offset     : -0.000041040 seconds
RMS offset      : 0.000053422 seconds
Frequency       : 2.286 ppm slow
Residual freq   : -0.000 ppm
Skew            : 0.012 ppm
Root delay      : 0.012706812 seconds
Root dispersion : 0.000430042 seconds
Update interval : 1029.8 seconds
Leap status     : Normal
```

- 如果该命令返回结果如下，则表示同步过程出错：

```
Leap status    : Not synchronised
```

- 如果该命令返回结果如下，则表示 Chrony 服务未正常运行：

```
506 Cannot talk to daemon
```

如果要使 NTP 服务尽快开始同步，执行以下命令。可以将 pool.ntp.org 替换为你的 NTP 服务器：

```
sudo systemctl stop ntpd.service && \
sudo ntpdate pool.ntp.org && \
sudo systemctl start ntpd.service
```

如果要在 CentOS 7 系统上手动安装 NTP 服务，可执行以下命令：

```
sudo yum install ntp ntpdate && \
sudo systemctl start ntpd.service && \
sudo systemctl enable ntpd.service
```

4.2.5 检查和配置操作系统优化参数

在生产系统的 TiDB 中，建议对操作系统进行如下的配置优化：

1. 关闭透明大页（即 Transparent Huge Pages，缩写为 THP）。数据库的内存访问模式往往是稀疏的而非连续的。当高阶内存碎片化比较严重时，分配 THP 页面会出现较高的延迟。
2. 将存储介质的 I/O 调度器设置为 noop。对于高速 SSD 存储介质，内核的 I/O 调度操作会导致性能损失。将调度器设置为 noop 后，内核不做任何操作，直接将 I/O 请求下发给硬件，以获取更好的性能。同时，noop 调度器也有较好的普适性。
3. 为调整 CPU 频率的 cpufreq 模块选用 performance 模式。将 CPU 频率固定在其支持的最高运行频率上，不进行动态调节，可获取最佳的性能。

采用如下步骤检查操作系统的当前配置，并配置系统优化参数：

1. 执行以下命令查看透明大页的开启状态。

```
cat /sys/kernel/mm/transparent_hugepage/enabled
```

```
[always] madvise never
```

> 注意：
>
> `[always] madvise never` 表示透明大页处于启用状态，需要关闭。

2. 执行以下命令查看数据目录所在磁盘的 I/O 调度器。假设在 sdb、sdc 两个磁盘上创建了数据目录。

```
cat /sys/block/sd[bc]/queue/scheduler
```

```
noop [deadline] cfq
noop [deadline] cfq
```

> 注意：
>
> `noop [deadline] cfq` 表示磁盘的 I/O 调度器使用 `deadline`，需要进行修改。

3. 执行以下命令查看磁盘的唯一标识 `ID_SERIAL`。

```
udevadm info --name=/dev/sdb | grep ID_SERIAL
```

```
E: ID_SERIAL=36d0946606d79f90025f3e09a0c1f9e81
E: ID_SERIAL_SHORT=6d0946606d79f90025f3e09a0c1f9e81
```

> 注意：
>
> 如果多个磁盘都分配了数据目录，需要多次执行以上命令，记录所有磁盘各自的唯一标识。

4. 执行以下命令查看 cpufreq 模块选用的节能策略。

```
cpupower frequency-info --policy
```

```
analyzing CPU 0:
current policy: frequency should be within 1.20 GHz and 3.10 GHz.
              The governor "powersave" may decide which speed to use within this range.
```

> **注意：**
>
> The governor "powersave" 表示 cpufreq 的节能策略使用 powersave，需要调整为 performance 策略。如果是虚拟机或者云主机，则不需要调整，命令输出通常为 Unable to determine current policy。

5. 配置系统优化参数。

    - 方法一：使用 tuned（推荐）

        1. 执行 tuned-adm list 命令查看当前操作系统的 tuned 策略。

```
tuned-adm list
```

```
Available profiles:
- balanced                    - General non-specialized tuned profile
- desktop                     - Optimize for the desktop use-case
- hpc-compute                 - Optimize for HPC compute workloads
- latency-performance         - Optimize for deterministic performance at the cost
    ↪ of increased power consumption
- network-latency             - Optimize for deterministic performance at the cost
    ↪ of increased power consumption, focused on low latency network performance
- network-throughput          - Optimize for streaming network throughput,
    ↪ generally only necessary on older CPUs or 40G+ networks
- powersave                   - Optimize for low power consumption
- throughput-performance      - Broadly applicable tuning that provides excellent
    ↪ performance across a variety of common server workloads
- virtual-guest               - Optimize for running inside a virtual guest
- virtual-host                - Optimize for running KVM guests
Current active profile: balanced
```

        Current active profile: balanced 表示当前操作系统的 tuned 策略使用 balanced，建议在当前策略的基础上添加操作系统优化配置。

        2. 创建新的 tuned 策略。

```
mkdir /etc/tuned/balanced-tidb-optimal/
vi /etc/tuned/balanced-tidb-optimal/tuned.conf
```

```
[main]
include=balanced

[cpu]
governor=performance

[vm]
transparent_hugepages=never
```

```
[disk]
devices_udev_regex=(ID_SERIAL=36d0946606d79f90025f3e09a0c1fc035)|(ID_SERIAL=36
    ↪ d0946606d79f90025f3e09a0c1f9e81)
elevator=noop
```

include=balanced 表示在现有的 balanced 策略基础上添加操作系统优化配置。

3. 应用新的 tuned 策略。

```
tuned-adm profile balanced-tidb-optimal
```

- 方法二：使用脚本方式。如果已经使用 tuned 方法，请跳过本方法。

1. 执行 grubby 命令查看默认内核版本。

> 注意：
> 需安装 grubby 软件包。

```
grubby --default-kernel
```

```
/boot/vmlinuz-3.10.0-957.el7.x86_64
```

2. 执行 grubby --update-kernel 命令修改内核配置。

```
grubby --args="transparent_hugepage=never" --update-kernel /boot/vmlinuz
    ↪ -3.10.0-957.el7.x86_64
```

> 注意：
> --update-kernel 后需要使用实际的默认内核版本。

3. 执行 grubby --info 命令查看修改后的默认内核配置。

```
grubby --info /boot/vmlinuz-3.10.0-957.el7.x86_64
```

> 注意：
> --info 后需要使用实际的默认内核版本。

```
index=0
kernel=/boot/vmlinuz-3.10.0-957.el7.x86_64
args="ro crashkernel=auto rd.lvm.lv=centos/root rd.lvm.lv=centos/swap rhgb quiet
    ↪ LANG=en_US.UTF-8 transparent_hugepage=never"
root=/dev/mapper/centos-root
initrd=/boot/initramfs-3.10.0-957.el7.x86_64.img
title=CentOS Linux (3.10.0-957.el7.x86_64) 7 (Core)
```

4. 修改当前的内核配置立即关闭透明大页。

```
echo never > /sys/kernel/mm/transparent_hugepage/enabled
echo never > /sys/kernel/mm/transparent_hugepage/defrag
```

5. 配置 udev 脚本应用 IO 调度器策略。

```
vi /etc/udev/rules.d/60-tidb-schedulers.rules
```

```
ACTION=="add|change", SUBSYSTEM=="block", ENV{ID_SERIAL}=="36
    ↪ d0946606d79f90025f3e09a0c1fc035", ATTR{queue/scheduler}="noop"
ACTION=="add|change", SUBSYSTEM=="block", ENV{ID_SERIAL}=="36
    ↪ d0946606d79f90025f3e09a0c1f9e81", ATTR{queue/scheduler}="noop"
```

6. 应用 udev 脚本。

```
udevadm control --reload-rules
udevadm trigger --type=devices --action=change
```

7. 创建 CPU 节能策略配置服务。

```
cat  >> /etc/systemd/system/cpupower.service << EOF
[Unit]
Description=CPU performance
[Service]
Type=oneshot
ExecStart=/usr/bin/cpupower frequency-set --governor performance
[Install]
WantedBy=multi-user.target
EOF
```

8. 应用 CPU 节能策略配置服务。

```
systemctl daemon-reload
systemctl enable cpupower.service
systemctl start cpupower.service
```

6. 执行以下命令验证透明大页的状态。

```
cat /sys/kernel/mm/transparent_hugepage/enabled
```

```
always madvise [never]
```

7. 执行以下命令验证数据目录所在磁盘的 I/O 调度器。

```
cat /sys/block/sd[bc]/queue/scheduler
```

```
[noop] deadline cfq
[noop] deadline cfq
```

8. 执行以下命令查看 cpufreq 模块选用的节能策略。

```
cpupower frequency-info --policy
  ```
```

analyzing CPU 0: current policy: frequency should be within 1.20 GHz and 3.10 GHz. The governor “performance” may decide which speed to use within this range. “ ‘

### 4.2.6 手动配置 SSH 互信及 sudo 免密码

对于有需求，通过手动配置中控机至目标节点互信的场景，可参考本段。通常推荐使用 TiUP 部署工具会自动配置 SSH 互信及免密登录，可忽略本段内容。

1. 以 root 用户依次登录到部署目标机器创建 tidb 用户并设置登录密码。

```
useradd tidb && \
passwd tidb
```

2. 执行以下命令，将 tidb ALL=(ALL)NOPASSWD: ALL 添加到文件末尾，即配置好 sudo 免密码。

```
visudo
```

```
tidb ALL=(ALL) NOPASSWD: ALL
```

3. 以 tidb 用户登录到中控机，执行以下命令。将 10.0.1.1 替换成你的部署目标机器 IP，按提示输入部署目标机器 tidb 用户密码，执行成功后即创建好 SSH 互信，其他机器同理。新建的 tidb 用户下没有 .ssh 目录，需要执行生成 rsa 密钥的命令来生成 .ssh 目录。如果要在中控机上部署 TiDB 组件，需要为中控机和中控机自身配置互信。

```
ssh-keygen -t rsa
ssh-copy-id -i ~/.ssh/id_rsa.pub 10.0.1.1
```

4. 以 tidb 用户登录中控机，通过 ssh 的方式登录目标机器 IP。如果不需要输入密码并登录成功，即表示 SSH 互信配置成功。

```
ssh 10.0.1.1
```

```
[tidb@10.0.1.1 ~]$
```

5. 以 tidb 用户登录到部署目标机器后，执行以下命令，不需要输入密码并切换到 root 用户，表示 tidb 用户 sudo 免密码配置成功。

```
sudo -su root
```

```
[root@10.0.1.1 tidb]#
```
```

4.2.7 安装 numactl 工具

本段主要介绍如果安装 NUMA 工具。在生产环境中，因为硬件机器配置往往高于需求，为了更合理规划资源，会考虑单机多实例部署 TiDB 或者 TiKV。NUMA 绑核工具的使用，主要为了防止 CPU 资源的争抢，引发性能衰退。

> 注意：
>
> - NUMA 绑核是用来隔离 CPU 资源的一种方法，适合高配置物理机环境部署多实例使用。
> - 通过 `tiup cluster deploy` 完成部署操作，就可以通过 `exec` 命令来进行集群级别管理工作。

1. 登录到目标节点进行安装（以 CentOS Linux release 7.7.1908 (Core) 为例）

```
sudo yum -y install numactl
```

2. 通过 TiUP 的 cluster 执行完 exec 命令来完成批量安装

```
tiup cluster exec --help
```

```
Run shell command on host in the tidb cluster

Usage:
cluster exec <cluster-name> [flags]

Flags:
    --command string   the command run on cluster host (default "ls")
-h, --help             help for exec
    --sudo             use root permissions (default false)
```

将 tidb-test 集群所有目标主机通过 sudo 权限执行安装命令

```
tiup cluster exec tidb-test --sudo --command "yum -y install numactl"
```

4.3 配置拓扑结构

4.3.1 最小拓扑架构

本文档介绍 TiDB 集群最小部署的拓扑架构。

4.3.1.1 拓扑信息

实例	个数	物理机配置	IP	配置
TiDB	3	16 VCore 32GB * 1	10.0.1.1 10.0.1.2 10.0.1.3	默认端口全局目录配置

实例	个数	物理机配置	IP	配置
PD	3	4 VCore 8GB * 1	10.0.1.4 10.0.1.5 10.0.1.6	默认端口全局目录配置
TiKV	3	16 VCore 32GB 2TB (nvme ssd) * 1	10.0.1.7 10.0.1.8 10.0.1.9	默认端口全局目录配置
Monitoring & Grafana	1	4 VCore 8GB * 1 500GB (ssd)	10.0.1.11	默认端口全局目录配置

4.3.1.1.1 拓扑模版

简单最小配置模板

详细最小配置模板

以上 TiDB 集群拓扑文件中，详细的配置项说明见通过 TiUP 部署 TiDB 集群的拓扑文件配置。

> 注意：
>
> - 无需手动创建配置文件中的 `tidb` 用户，TiUP cluster 组件会在目标主机上自动创建该用户。可以自定义用户，也可以和中控机的用户保持一致。
> - 如果部署目录配置为相对路径，会部署在用户家目录下。

4.3.2 TiFlash 部署拓扑

本文介绍在部署最小拓扑集群的基础上，部署TiFlash 的拓扑结构。TiFlash 是列式的存储引擎，已经成为集群拓扑的标配，适合 Real-Time HTAP 业务。

4.3.2.1 拓扑信息

实例	个数	物理机配置	IP	配置
TiDB	3	16 VCore 32GB * 1	10.0.1.1 10.0.1.2 10.0.1.3	默认端口全局目录配置
PD	3	4 VCore 8GB * 1	10.0.1.4 10.0.1.5 10.0.1.6	默认端口全局目录配置
TiKV	3	16 VCore 32GB 2TB (nvme ssd) * 1	10.0.1.1 10.0.1.2 10.0.1.3	默认端口全局目录配置
TiFlash	1	32 VCore 64 GB 2TB (nvme ssd) * 1	10.0.1.10	默认端口全局目录配置
Monitoring & Grafana	1	4 VCore 8GB * 1 500GB (ssd)	10.0.1.10	默认端口全局目录配置

4.3.2.1.1 拓扑模版

- 简单 TiFlash 配置模版
- 详细 TiFlash 配置模版

以上 TiDB 集群拓扑文件中，详细的配置项说明见通过 TiUP 部署 TiDB 集群的拓扑文件配置。

4.3.2.1.2 关键参数介绍

- 需要将配置模板中 `replication.enable-placement-rules` 设置为 `true`，以开启 PD 的Placement Rules 功能。
- `tiflash_servers` 实例级别配置 `"-host"` 目前只支持 IP，不支持域名。
- TiFlash 具体的参数配置介绍可参考TiFlash 参数配置。

> 注意：
>
> - 无需手动创建配置文件中的 `tidb` 用户，TiUP cluster 组件会在目标主机上自动创建该用户。可以自定义用户，也可以和中控机的用户保持一致。
> - 如果部署目录配置为相对路径，会部署在用户家目录下。

4.3.3 TiCDC 部署拓扑

> 注意：
>
> TiCDC 从 v4.0.6 起成为正式功能，可用于生产环境。

本文介绍TiCDC 部署的拓扑，以及如何在最小拓扑的基础上同时部署 TiCDC。TiCDC 是 4.0 版本开始支持的 TiDB 增量数据同步工具，支持多种下游 (TiDB/MySQL/MQ)。相比于 TiDB Binlog，TiCDC 有延迟更低、天然高可用等优点。

4.3.3.1 拓扑信息

实例	个数	物理机配置	IP	配置
TiDB	3	16 VCore 32GB * 1	10.0.1.1 10.0.1.2 10.0.1.3	默认端口全局目录配置
PD	3	4 VCore 8GB * 1	10.0.1.4 10.0.1.5 10.0.1.6	默认端口全局目录配置
TiKV	3	16 VCore 32GB 2TB (nvme ssd) * 1	10.0.1.7 10.0.1.8 10.0.1.9	默认端口全局目录配置
CDC	3	8 VCore 16GB * 1	10.0.1.11 10.0.1.12 10.0.1.13	默认端口全局目录配置
Monitoring & Grafana	1	4 VCore 8GB * 1 500GB (ssd)	10.0.1.11	默认端口全局目录配置

4.3.3.1.1 拓扑模版

简单 TiCDC 配置模板

详细 TiCDC 配置模板

以上 TiDB 集群拓扑文件中，详细的配置项说明见通过 TiUP 部署 TiDB 集群的拓扑文件配置。

注意：

- 无需手动创建配置文件中的 `tidb` 用户，TiUP cluster 组件会在目标主机上自动创建该用户。可以自定义用户，也可以和中控机的用户保持一致。
- 如果部署目录配置为相对路径，会部署在用户家目录下。

4.3.4 TiDB Binlog 部署拓扑

本文介绍在部署最小拓扑集群的基础上，同时部署TiDB Binlog。TiDB Binlog 是目前广泛使用的增量同步组件，可提供准实时备份和同步功能。

4.3.4.1 拓扑信息

实例	个数	物理机配置	IP	配置
TiDB	3	16 VCore 32 GB	10.0.1.1 10.0.1.2 10.0.1.3	默认端口配置；开启 enable_binlog；开启 ignore-error
PD	3	4 VCore 8 GB	10.0.1.4 10.0.1.5 10.0.1.6	默认端口配置

实例	个数	物理机配置	IP	配置
TiKV	3	16 VCore 32 GB	10.0.1.7 10.0.1.8 10.0.1.9	默认端口配置
Pump	3	8 VCore 16GB	10.0.1.1 10.0.1.7 10.0.1.8	默认端口配置；设置 GC 时间 7 天

实例	个数	物理机配置	IP	配置
Drainer	1	8 VCore 16GB	10.0.1.11	默认端口配置；设置默认初始化 commitTS -1 为最近的时间戳配置下游目标 TiDB 10.0.1.12:4000

4.3.4.1.1 拓扑模版

简单 TiDB Binlog 配置模板（下游为 MySQL）

简单 TiDB Binlog 配置模板（下游为 file）

详细 TiDB Binlog 配置模板

以上 TiDB 集群拓扑文件中，详细的配置项说明见通过 TiUP 部署 TiDB 集群的拓扑文件配置。

4.3.4.1.2 关键参数介绍

拓扑配置模版的关键参数如下：

- `server_configs.tidb.binlog.enable: true`

  开启 TiDB Binlog 服务，默认为 false。

- `server_configs.tidb.binlog.ignore-error: true`

  高可用场景建议开启，如果设置为 true，发生错误时，TiDB 会停止写入 TiDB Binlog，并且在监控项 `tidb_server_critical_error_total` 上计数加 1；如果设置为 false，一旦写入 TiDB Binlog 失败，会停止整个 TiDB 的服务。

- `drainer_servers.config.syncer.db-type`

  TiDB Binlog 的下游类型，目前支持 `mysql`、`tidb`、`kafka` 和 `file`。

- `drainer_servers.config.syncer.to`

  TiDB Binlog 的下游配置。根据 `db-type` 的不同，该选项可配置下游数据库的连接参数、Kafka 的连接参数、文件保存路径。详细说明可参见TiDB Binlog 配置说明。

> 注意：
>
> - 编辑配置文件模版时，如无需自定义端口或者目录，仅修改 IP 即可。
> - 无需手动创建配置文件中的 `tidb` 用户，TiUP cluster 组件会在目标主机上自动创建该用户。可以自定义用户，也可以和中控机的用户保持一致。
> - 如果部署目录配置为相对路径，会部署在用户家目录下。

4.3.5 TiSpark 部署拓扑

本文介绍 TiSpark 部署的拓扑，以及如何在最小拓扑的基础上同时部署 TiSpark。TiSpark 是 PingCAP 为解决用户复杂 OLAP 需求而推出的产品。它借助 Spark 平台，同时融合 TiKV 分布式集群的优势，和 TiDB 一起为用户一站式解决 HTAP (Hybrid Transactional/Analytical Processing) 的需求。

关于 TiSpark 的架构介绍与使用，参见TiSpark 用户指南。

> 警告：
>
> TiUP Cluster 的 TiSpark 支持目前为实验特性，不建议在生产环境中使用。

4.3.5.1 拓扑信息

实例	个数	物理机配置	IP	配置
TiDB	3	16 VCore 32GB * 1	10.0.1.1 10.0.1.2 10.0.1.3	默认端口全局目录配置
PD	3	4 VCore 8GB * 1	10.0.1.4 10.0.1.5 10.0.1.6	默认端口全局目录配置
TiKV	3	16 VCore 32GB 2TB (nvme ssd) * 1	10.0.1.7 10.0.1.8 10.0.1.9	默认端口全局目录配置

实例	个数	物理机配置	IP	配置
TiSpark	3	8 VCore 16GB * 1	10.0.1.2(master) 10.0.1.22(worker) 10.0.1.23(worker)	默认端口 全局目录配置
Monitoring & Grafana	1	4 VCore 8GB * 1 500GB (ssd)	10.0.1.11	默认端口 全局目录配置

4.3.5.1.1 拓扑模版

简单 TiSpark 配置模板

详细 TiSpark 配置模板

以上 TiDB 集群拓扑文件中，详细的配置项说明见通过 TiUP 部署 TiDB 集群的拓扑文件配置。

> **注意：**
>
> - 无需手动创建配置文件中的 `tidb` 用户，TiUP cluster 组件会在目标主机上自动创建该用户。可以自定义用户，也可以和中控机的用户保持一致。
> - 如果部署目录配置为相对路径，会部署在用户家目录下。

4.3.5.2 环境要求

由于 TiSpark 基于 Apache Spark 集群，在启动包含 TiSpark 组件的 TiDB 集群前，需要在部署了 TiSpark 组件的服务器上安装 Java 运行时环境 (JRE) 8，否则将无法启动相关组件。

TiUP 不提供自动安装 JRE 的支持，该操作需要用户自行完成。JRE 8 的安装方法可以参考 OpenJDK 的文档说明。

如果部署服务器上已经安装有 JRE 8，但不在系统的默认包管理工具路径中，可以通过在拓扑配置中设置 java_home 参数来指定要使用的 JRE 环境所在的路径。该参数对应系统环境变量 JAVA_HOME。

4.3.6 跨数据中心部署拓扑

本文以典型的两地三中心为例，介绍跨数据中心部署的拓扑以及关键参数。本文示例所涉及的城市是上海（即 sha）和北京（即 bja 和 bjb）。

4.3.6.1 拓扑信息

实例	个数	物理机配置	BJ IP	SH IP	配置
TiDB	5	16 VCore 32GB * 1	10.0.1.1 10.0.1.2 10.0.1.3 10.0.1.4	10.0.1.5	默认端口全局目录配置
PD	5	4 VCore 8GB * 1	10.0.1.6 10.0.1.7 10.0.1.8 10.0.1.9	10.0.1.10	默认端口全局目录配置

实例	个数	物理机配置	BJ IP	SH IP	配置
TiKV	5	16 VCore 32GB 2TB (nvme ssd) * 1	10.0.1.11 10.0.1.12 10.0.1.13 10.0.1.14	10.0.1.15	默认端口全局目录配置
Monitoring & Grafana	1	4 VCore 8GB * 1 500GB (ssd)	10.0.1.16		默认端口全局目录配置

4.3.6.1.1 拓扑模版

跨机房配置模板

以上 TiDB 集群拓扑文件中，详细的配置项说明见通过 TiUP 部署 TiDB 集群的拓扑文件配置。

4.3.6.1.2 关键参数配置

本节介绍跨数据中心部署 TiDB 集群的关键参数配置。

TiKV 参数

- 设置 gRPC 的压缩格式，默认为 none。为提高跨机房部署场景的目标节点间 gRPC 包的传输速度，建议设置为 gzip 格式。

```
server.grpc-compression-type: gzip
```

- label 配置

  由于采用跨机房部署 TiKV，为了避免物理机宕机导致 Raft Group 默认的 5 副本中丢失 3 副本，使集群不可用的问题，可以通过 label 来实现 PD 智能调度，保证同中心、同机柜、同机器 TiKV 实例不会出现 Raft Group 有 3 副本的情况。

- TiKV 配置

  相同物理机配置相同的 host 级别 label 信息：

```
config:
  server.labels:
    zone: bj
    dc: bja
    rack: rack1
    host: host2
```

- 防止异地 TiKV 节点发起不必要的 Raft 选举，需要将异地 TiKV 节点发起选举时经过最少的 tick 个数和最多经过的 tick 个数都调大，这两个参数默认设置均为 0。

```
raftstore.raft-min-election-timeout-ticks: 1000
raftstore.raft-max-election-timeout-ticks: 1020
```

PD 参数

- PD 元数据信息记录 TiKV 集群的拓扑信息，根据四个维度调度 Raft Group 副本。

```
replication.location-labels: ["zone","dc","rack","host"]
```

- 调整 Raft Group 的副本数据量为 5 ，保证集群的高可用性。

```
replication.max-replicas: 5
```

- 拒绝异地机房 TiKV 的 Raft 副本选举为 Leader。

```
label-property:
      reject-leader:
        - key: "dc"
          value: "sha"
```

> 注意：
>
> - 无需手动创建配置文件中的 tidb 用户，TiUP cluster 组件会在目标主机上自动创建该用户。可以自定义用户，也可以和中控机的用户保持一致。
> - 如果部署目录配置为相对路径，会部署在用户家目录下。

有关 Label 的使用和 Raft Group 副本数量，详见通过拓扑 label 进行副本调度。

4.3.7 混合部署拓扑

本文介绍 TiDB 集群的 TiKV 和 TiDB 混合部署拓扑以及主要参数。常见的场景为，部署机为多路 CPU 处理器，内存也充足，为提高物理机资源利用率，可单机多实例部署，即 TiDB、TiKV 通过 numa 绑核，隔离 CPU 资源。PD 和 Prometheus 混合部署，但两者的数据目录需要使用独立的文件系统。

4.3.7.1 拓扑信息

实例	个数	物理机配置	IP	配置
TiDB	6	32 VCore 64GB	10.0.1.1 10.0.1.2 10.0.1.3	配置 numa 绑核操作配置
PD	3	16 VCore 32 GB	10.0.1.4 10.0.1.5 10.0.1.6	loca-tion_lables 参数

实例	个数	物理机配置	IP	配置
TiKV	6	32 VCore 64GB	10.0.1.7 10.0.1.8 10.0.1.9	1. 区分实例级别的 port、status_port； 2. 配置全局参数 readpool、storage 以及 raftstore； 3. 配置实例级别 host 维度的 labels； 4. 配置 numa 绑核

实例	个数	物理机配置	IP	配置
Monitoring & Grafana	1	4 VCore 8GB * 1 500GB (ssd)	10.0.1.11	默认配置

4.3.7.1.1 拓扑模版

简单混部配置模板

详细混部配置模板

以上 TiDB 集群拓扑文件中，详细的配置项说明见通过 TiUP 部署 TiDB 集群的拓扑文件配置。

4.3.7.1.2 混合部署的关键参数介绍

本节介绍单机多实例的关键参数，主要用于 TiDB、TiKV 的单机多实例部署场景。你需要按照提供的计算公式，将结果填写至上一步的配置模板中。

- TiKV 进行配置优化
  - readpool 线程池自适应，配置 `readpool.unified.max-thread-count` 参数可以使 `readpool.storage` 和 `readpool.coprocessor` 共用统一线程池，同时要分别设置自适应开关。
    * 开启 `readpool.storage` 和 `readpool.coprocessor`：

```
readpool.storage.use-unified-pool: true
readpool.coprocessor.use-unified-pool: true
```

    * 计算公式如下：

```
readpool.unified.max-thread-count = cores * 0.8 / TiKV 数量
```

  - storage CF (all RocksDB column families) 内存自适应，配置 `storage.block-cache.capacity` 参数即可实现 CF 之间自动平衡内存使用。
    * `storage.block-cache` 默认开启 CF 自适应，无需修改。

```
storage.block-cache.shared: true
```

    * 计算公式如下：

```
storage.block-cache.capacity = (MEM_TOTAL * 0.5 / TiKV 实例数量)
```

- 如果多个 TiKV 实例部署在同一块物理磁盘上，需要在 tikv 配置中添加 capacity 参数：

```
raftstore.capacity = 磁盘总容量 / TiKV 实例数量
```

- label 调度配置

  由于采用单机多实例部署 TiKV，为了避免物理机宕机导致 Region Group 默认 3 副本的 2 副本丢失，导致集群不可用的问题，可以通过 label 来实现 PD 智能调度，保证同台机器的多 TiKV 实例不会出现 Region Group 只有 2 副本的情况。

  - TiKV 配置

    相同物理机配置相同的 host 级别 label 信息：

```
config:
  server.labels:
    host: tikv1
```

  - PD 配置

    PD 需要配置 labels 类型来识别并调度 Region：

```
pd:
  replication.location-labels: ["host"]
```

- numa_node 绑核

  - 在实例参数模块配置对应的 numa_node 参数，并添加对应的物理 CPU 的核数；
  - numa 绑核使用前，确认已经安装 numactl 工具，以及物理机对应的物理机 CPU 的信息后，再进行参数配置；
  - numa_node 这个配置参数与 `numactl --membind` 配置对应。

注意：

- 编辑配置文件模版时，注意修改必要参数、IP、端口及目录。
- 各个组件的 deploy_dir，默认会使用 global 中的 <deploy_dir>/<components_name>-<port>。例如 tidb 端口指定 4001，则 deploy_dir 默认为 /tidb-deploy/tidb-4001。因此，在多实例场景下指定非默认端口时，无需再次指定目录。
- 无需手动创建配置文件中的 tidb 用户，TiUP cluster 组件会在部署主机上自动创建该用户。可以自定义用户，也可以和中控机的用户保持一致。
- 如果部署目录配置为相对路径，会部署在用户家目录下。

4.4 安装与启动

4.4.1 Linux 环境

4.4.1.1 使用 TiUP 部署 TiDB 集群

TiUP 是 TiDB 4.0 版本引入的集群运维工具，TiUP cluster 是 TiUP 提供的使用 Golang 编写的集群管理组件，通过 TiUP cluster 组件就可以进行日常的运维工作，包括部署、启动、关闭、销毁、弹性扩缩容、升级 TiDB 集群，以及管理 TiDB 集群参数。

目前 TiUP 可以支持部署 TiDB、TiFlash、TiDB Binlog、TiCDC，以及监控系统。本文将介绍不同集群拓扑的具体部署步骤。

4.4.1.1.1 第 1 步：软硬件环境需求及前置检查

软硬件环境需求

环境与系统配置检查

4.4.1.1.2 第 2 步：在中控机上安装 TiUP 组件

使用普通用户登录中控机，以 tidb 用户为例，后续安装 TiUP 及集群管理操作均通过该用户完成：

1. 执行如下命令安装 TiUP 工具：

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

2. 按如下步骤设置 TiUP 环境变量：

   重新声明全局环境变量：

```
source .bash_profile
```

   确认 TiUP 工具是否安装：

```
which tiup
```

3. 安装 TiUP cluster 组件

```
tiup cluster
```

4. 如果已经安装，则更新 TiUP cluster 组件至最新版本：

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

   预期输出 “Update successfully!” 字样。

5. 验证当前 TiUP cluster 版本信息。执行如下命令查看 TiUP cluster 组件版本：

```
tiup --binary cluster
```

4.4.1.1.3 第 3 步：编辑初始化配置文件

请根据不同的集群拓扑，编辑 TiUP 所需的集群初始化配置文件。

这里举出常见的几种场景，请根据链接中的拓扑说明，以及给出的配置文件模板，新建一个配置文件 `topology.yaml`。如果有其他组合场景的需求，请根据多个模板自行调整。

- 最小拓扑架构

  最基本的集群拓扑，包括 tidb-server、 tikv-server、 pd-server，适合 OLTP 业务。

- 增加 TiFlash 拓扑架构

  包含最小拓扑的基础上，同时部署 TiFlash。TiFlash 是列式的存储引擎，已经逐步成为集群拓扑的标配。适合 Real-Time HTAP 业务。

- 增加 TiCDC 拓扑架构

  包含最小拓扑的基础上，同时部署 TiCDC。TiCDC 是 4.0 版本开始支持的 TiDB 增量数据同步工具，支持多种下游 (TiDB/MySQL/MQ)。相比于 TiDB Binlog，TiCDC 有延迟更低、天然高可用等优点。在部署完成后，需要启动 TiCDC，通过 `cdc cli` 创建同步任务。

- 增加 TiDB Binlog 拓扑架构

  包含最小拓扑的基础上，同时部署 TiDB Binlog。TiDB Binlog 是目前广泛使用的增量同步组件，可提供准实时备份和同步功能。

- 增加 TiSpark 拓扑架构

  包含最小拓扑的基础上，同时部署 TiSpark 组件。TiSpark 是 PingCAP 为解决用户复杂 OLAP 需求而推出的产品。TiUP cluster 组件对 TiSpark 的支持目前为实验性特性。

- 混合部署拓扑架构

  适用于单台机器，混合部署多个实例的情况，也包括单机多实例，需要额外增加目录、端口、资源配比、label 等配置。

- 跨机房部署拓扑架构

  以典型的 两地三中心 架构为例，介绍跨机房部署架构，以及需要注意的关键设置。

> 注意：
>
> - 对于需要全局生效的参数，请在配置文件中 `server_configs` 的对应组件下配置。
> - 对于需要某个节点生效的参数，请在具体节点的 `config` 中配置。
> - 配置的层次结构使用 `.` 表示。如：`log.slow-threshold`。更多格式参考 TiUP 配置参数模版。
> - 更多参数说明，请参考 TiDB `config.toml.example`、TiKV `config.toml.example`、PD `config.toml.example` 和TiFlash 配置参数。

4.4.1.1.4 第 4 步：执行部署命令

> 注意：
>
> 通过 TiUP 进行集群部署可以使用密钥或者交互密码方式来进行安全认证：
>
> - 如果是密钥方式，可以通过 `-i` 或者 `--identity_file` 来指定密钥的路径；
> - 如果是密码方式，可以通过 `-p` 进入密码交互窗口；
> - 如果已经配置免密登录目标机，则不需填写认证。
>
> 一般情况下 TiUP 会在目标机器上创建 `topology.yaml` 中约定的用户和组，以下情况例外：
>
> - `topology.yaml` 中设置的用户名在目标机器上已存在。
> - 在命令行上使用了参数 `--skip-create-user` 明确指定跳过创建用户的步骤。

```
tiup cluster deploy tidb-test v4.0.16 ./topology.yaml --user root [-p] [-i /home/root/.ssh/
    ↪ gcp_rsa]
```

以上部署命令中：

- 通过 TiUP cluster 部署的集群名称为 `tidb-test`
- 部署版本为 `v4.0.16`，最新版本可以通过执行 `tiup list tidb` 来查看 TiUP 支持的版本
- 初始化配置文件为 `topology.yaml`
- –user root：通过 root 用户登录到目标主机完成集群部署，该用户需要有 ssh 到目标机器的权限，并且在目标机器有 sudo 权限。也可以用其他有 ssh 和 sudo 权限的用户完成部署。
- [-i] 及 [-p]：非必选项，如果已经配置免密登录目标机，则不需填写。否则选择其一即可，[-i] 为可登录到目标机的 root 用户（或 –user 指定的其他用户）的私钥，也可使用 [-p] 交互式输入该用户的密码
- 如果需要指定在目标机创建的用户组名，可以参考这个例子。

预期日志结尾输出会有 `Deployed cluster `tidb-test` successfully` 关键词，表示部署成功。

4.4.1.1.5 第 5 步：查看 TiUP 管理的集群情况

```
tiup cluster list
```

TiUP 支持管理多个 TiDB 集群，该命令会输出当前通过 TiUP cluster 管理的所有集群信息，包括集群名称、部署用户、版本、密钥信息等：

```
Starting /home/tidb/.tiup/components/cluster/v1.0.0/cluster list
Name              User  Version        Path
    ↪                                                    PrivateKey
----              ----  -------        ----
    ↪                                                    ----------
tidb-test         tidb  v4.0.16      /home/tidb/.tiup/storage/cluster/clusters/tidb-test
    ↪ /home/tidb/.tiup/storage/cluster/clusters/tidb-test/ssh/id_rsa
```

4.4.1.1.6 第 6 步：检查部署的 TiDB 集群情况

例如，执行如下命令检查 tidb-test 集群情况：

```
tiup cluster display tidb-test
```

预期输出包括 tidb-test 集群中实例 ID、角色、主机、监听端口和状态（由于还未启动，所以状态为 Down/inactive）、目录信息。

4.4.1.1.7 第 7 步：启动集群

```
tiup cluster start tidb-test
```

预期结果输出 Started cluster `tidb-test` successfully 标志启动成功。

4.4.1.1.8 第 8 步：验证集群运行状态

- 通过 TiUP 检查集群状态

```
tiup cluster display tidb-test
```

预期结果输出，注意 Status 状态信息为 Up 说明集群状态正常

- 执行如下命令登录数据库：

```
mysql -u root -h 10.0.1.4 -P 4000
```

此外，也需要验证监控系统、TiDB Dashboard 的运行状态，以及简单命令的执行，验证方式可参考验证集群运行状态。

4.4.1.1.9 探索更多

如果你已同时部署了TiFlash，接下来可参阅以下文档：

- 使用 TiFlash
- TiFlash 集群运维
- TiFlash 报警规则与处理方法
- TiFlash 常见问题

如果你已同时部署了TiCDC，接下来可参阅以下文档：

- TiCDC 任务管理
- TiCDC 常见问题

> **注意：**
>
> TiDB、TiUP 及 TiDB Dashboard 默认会收集使用情况信息，并将这些信息分享给 PingCAP 用于改善产品。若要了解所收集的信息详情及如何禁用该行为，请参见遥测。

4.4.1.2 使用 TiUP 离线部署 TiDB 集群

本文介绍如何使用 TiUP 离线部署 TiDB 集群，具体的操作步骤如下。

4.4.1.2.1 1. 准备 TiUP 离线组件包

方式一：下载官方 TiUP 离线组件包

在 官方下载页面 选择对应版本的 TiDB server 离线镜像包（包含 TiUP 离线组件包）。

方式二：使用 `tiup mirror clone` 命令手动打包离线组件包

1. 在线环境中安装 TiUP 包管理器工具

    1. 执行如下命令安装 TiUP 工具：

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

    2. 重新声明全局环境变量：

    ```
    source .bash_profile
    ```

    3. 确认 TiUP 工具是否安装：

    ```
    which tiup
    ```

2. 使用 TiUP 制作离线镜像

    1. 在一台和外网相通的机器上拉取需要的组件：

    ```
    tiup mirror clone tidb-community-server-${version}-linux-amd64 ${version} --os=linux --
        ↪ arch=amd64
    ```

    该命令会在当前目录下创建一个名叫 `tidb-community-server-${version}-linux-amd64` 的目录，里面包含 TiUP 管理的组件包。

    2. 通过 tar 命令将该组件包打包然后发送到隔离环境的中控机：

    ```
    tar czvf tidb-community-server-${version}-linux-amd64.tar.gz tidb-community-server-${
        ↪ version}-linux-amd64
    ```

    此时，`tidb-community-server-${version}-linux-amd64.tar.gz` 就是一个独立的离线环境包。

4.4.1.2.2 2. 部署离线环境 TiUP 组件

将离线包发送到目标集群的中控机后，执行以下命令安装 TiUP 组件：

```
tar xzvf tidb-community-server-${version}-linux-amd64.tar.gz
sh tidb-community-server-${version}-linux-amd64/local_install.sh
source /home/tidb/.bash_profile
```

`local_install.sh` 脚本会自动执行 `tiup mirror set tidb-community-server-${version}-linux-amd64` 命令将当前镜像地址设置为 `tidb-community-server-${version}-linux-amd64`。

若需将镜像切换到其他目录，可以通过手动执行 `tiup mirror set <mirror-dir>` 进行切换。如果需要切换到在线环境，可执行 `tiup mirror set https://tiup-mirrors.pingcap.com`。

4.4.1.2.3 3. TiKV 数据盘挂载

> 注意：
>
> 推荐 TiKV 部署目标机器的数据目录使用 EXT4 文件系统格式。相比于 XFS 文件系统格式，EXT4 文件系统格式在 TiDB 集群部署案例较多，生产环境优先选择使用 EXT4 文件系统格式。

使用 `root` 用户登录目标机器，将部署目标机器数据盘格式化成 ext4 文件系统，挂载时添加 `nodelalloc` 和 `noatime` 挂载参数。`nodelalloc` 是必选参数，否则 TiUP 安装时检测无法通过；`noatime` 是可选建议参数。

> 注意：
>
> 如果你的数据盘已经格式化成 ext4 并挂载了磁盘，可先执行 `umount /dev/nvme0n1p1` 命令卸载，从编辑 `/etc/fstab` 文件步骤开始执行，添加挂载参数重新挂载即可。

以 `/dev/nvme0n1` 数据盘为例，具体操作步骤如下：

1. 查看数据盘。

```
fdisk -l
```

```
Disk /dev/nvme0n1: 1000 GB
```

2. 创建分区表。

```
parted -s -a optimal /dev/nvme0n1 mklabel gpt -- mkpart primary ext4 1 -1
```

> **注意：**
>
> 使用 `lsblk` 命令查看分区的设备号：对于 nvme 磁盘，生成的分区设备号一般为 `nvme0n1p1`；对于普通磁盘（例如 `/dev/sdb`），生成的的分区设备号一般为 `sdb1`。

3. 格式化文件系统。

```
mkfs.ext4 /dev/nvme0n1p1
```

4. 查看数据盘分区 UUID。

   本例中 `nvme0n1p1` 的 UUID 为 `c51eb23b-195c-4061-92a9-3fad812cc12f`。

```
lsblk -f
```

```
NAME    FSTYPE LABEL UUID                                 MOUNTPOINT
sda
├─sda1  ext4         237b634b-a565-477b-8371-6dff0c41f5ab /boot
├─sda2  swap         f414c5c0-f823-4bb1-8fdf-e531173a72ed
└─sda3  ext4         547909c1-398d-4696-94c6-03e43e317b60 /
sr0
nvme0n1
└─nvme0n1p1 ext4         c51eb23b-195c-4061-92a9-3fad812cc12f
```

5. 编辑 `/etc/fstab` 文件，添加 `nodelalloc` 挂载参数。

```
vi /etc/fstab
```

```
UUID=c51eb23b-195c-4061-92a9-3fad812cc12f /data1 ext4 defaults,nodelalloc,noatime 0 2
```

6. 挂载数据盘。

```
mkdir /data1 && \
mount -a
```

7. 执行以下命令，如果文件系统为 ext4，并且挂载参数中包含 `nodelalloc`，则表示已生效。

```
mount -t ext4
```

```
/dev/nvme0n1p1 on /data1 type ext4 (rw,noatime,nodelalloc,data=ordered)
```

4.4.1.2.4 4. 配置初始化参数文件 `topology.yaml`

集群初始化配置文件需要手动编写，完整的全配置参数模版可以参考 Github TiUP 项目配置参数模版。需要在中控机上面创建 YAML 格式配置文件，例如 `topology.yaml`:

```
cat topology.yaml
```

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

server_configs:
  pd:
    replication.enable-placement-rules: true

pd_servers:
  - host: 10.0.1.4
  - host: 10.0.1.5
  - host: 10.0.1.6
tidb_servers:
  - host: 10.0.1.7
  - host: 10.0.1.8
  - host: 10.0.1.9
tikv_servers:
  - host: 10.0.1.1
  - host: 10.0.1.2
  - host: 10.0.1.3
tiflash_servers:
  - host: 10.0.1.10
    data_dir: /data1/tiflash/data,/data2/tiflash/data
cdc_servers:
  - host: 10.0.1.6
  - host: 10.0.1.7
  - host: 10.0.1.8
monitoring_servers:
  - host: 10.0.1.4
grafana_servers:
  - host: 10.0.1.4
alertmanager_servers:
  - host: 10.0.1.4
```

4.4.1.2.5 5. 部署 TiDB 集群

执行以下命令来部署 TiDB 集群。

```
tiup cluster deploy tidb-test v4.0.16 topology.yaml --user tidb [-p] [-i /home/root/.ssh/gcp_rsa]
tiup cluster start tidb-test
```

参数说明：

- 通过 TiUP cluster 部署的集群名称为 `tidb-test`
- 部署版本为 `v4.0.16`，其他版本可以执行 `tiup list tidb` 获取
- 初始化配置文件为 `topology.yaml`
- –user tidb：通过 tidb 用户登录到目标主机完成集群部署，该用户需要有 ssh 到目标机器的权限，并且在目标机器有 sudo 权限。也可以用其他有 ssh 和 sudo 权限的用户完成部署。
- [-i] 及 [-p]：非必选项，如果已经配置免密登陆目标机，则不需填写。否则选择其一即可，[-i] 为可登录到部署机 root 用户（或 –user 指定的其他用户）的私钥，也可使用 [-p] 交互式输入该用户的密码

预期日志结尾输出会有 `Deployed cluster `tidb-test` successfully` 关键词，表示部署成功。

部署完成后，集群相关操作可参考cluster 命令。

注意：

TiDB 和 TiUP 默认会收集使用情况信息，并将这些信息分享给 PingCAP 用于改善产品。若要了解所收集的信息详情及如何禁用该行为，请参见遥测。

4.4.1.3 使用 TiDB Ansible 部署 TiDB 集群

警告：

对于生产环境，推荐使用 TiUP 部署 TiDB 集群。从 TiDB 4.0 版本开始，PingCAP 不再维护 TiDB Ansible 的部署方式。继续使用 TiDB Ansible 部署 TiDB 集群可能存在风险，因此不推荐该部署方式。TiUP 可直接支持之前使用 TiDB Ansible 方式部署的集群。

如果只是希望测试 TiDB 或体验 TiDB 特性，可参考TiDB 快速上手指南或者使用 Docker Compose 在单机上快速部署 TiDB 集群。

Ansible 是一款自动化运维工具，TiDB Ansible 是 PingCAP 基于 Ansible playbook 功能编写的集群部署工具。本文档介绍如何使用 TiDB Ansible 部署一个完整的 TiDB 集群。

本部署工具可以通过配置文件设置集群拓扑，完成以下各项运维工作：

- 初始化操作系统参数
- 部署 TiDB 集群（包括 PD、TiDB、TiKV 等组件和监控组件）
- 启动集群
- 关闭集群

- 变更组件配置
- 集群扩容缩容
- 升级组件版本
- 集群开启 binlog
- 清除集群数据
- 销毁集群

4.4.1.3.1 准备机器

1. 部署目标机器若干

    - 建议 4 台及以上，TiKV 至少 3 实例，且与 TiDB、PD 模块不位于同一主机，详见部署建议。
    - 目前支持在 x86_64 (AMD64) 和 ARM64 两种架构上部署 TiDB 集群。在 AMD64 架构下，建议使用 CentOS 7.3 及以上版本 Linux 操作系统；在 ARM 架构下，建议使用 CentOS 7.6 1810 版本 Linux 操作系统。
    - 机器之间内网互通。

> **注意：**
> 使用 TiDB Ansible 方式部署时，TiKV 及 PD 节点数据目录所在磁盘请使用 SSD 磁盘，否则无法通过检测。如果仅验证功能，建议使用 Docker Compose 部署方案单机进行测试。

2. 部署中控机一台

    - 中控机可以是部署目标机器中的某一台。
    - 推荐安装 CentOS 7.3 及以上版本 Linux 操作系统（默认包含 Python 2.7）。
    - 该机器需开放外网访问，用于下载 TiDB 及相关软件安装包。

4.4.1.3.2 第 1 步：在中控机上安装系统依赖包

以 root 用户登录中控机，然后根据操作系统类型执行相应的安装命令。

- 如果中控机使用的是 CentOS 7 系统，执行以下命令：

```
yum -y install epel-release git curl sshpass && \
yum -y install python2-pip
```

- 如果中控机使用的是 Ubuntu 系统，执行以下命令：

```
apt-get -y install git curl sshpass python-pip
```

4.4.1.3.3 第 2 步：在中控机上创建 tidb 用户，并生成 SSH key

以 root 用户登录中控机，执行以下步骤：

1. 创建 tidb 用户。

```
useradd -m -d /home/tidb tidb
```

2. 设置 tidb 用户密码。

```
passwd tidb
```

3. 配置 tidb 用户 sudo 免密码，将 tidb ALL=(ALL)NOPASSWD: ALL 添加到文件末尾即可。

```
visudo
```

```
tidb ALL=(ALL) NOPASSWD: ALL
```

4. 生成 SSH key。

   执行 su 命令，从 root 用户切换到 tidb 用户下。

```
su - tidb
```

   创建 tidb 用户 SSH key，提示 Enter passphrase 时直接回车即可。执行成功后，SSH 私钥文件为 /home/
   ↪ tidb/.ssh/id_rsa，SSH 公钥文件为 /home/tidb/.ssh/id_rsa.pub。

```
ssh-keygen -t rsa
```

```
Generating public/private rsa key pair.
Enter file in which to save the key (/home/tidb/.ssh/id_rsa):
Created directory '/home/tidb/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/tidb/.ssh/id_rsa.
Your public key has been saved in /home/tidb/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:eIBykszR1KyECA/h0d7PRKz4fhAeli7IrVphhte7/So tidb@172.16.10.49
The key's randomart image is:
+---[RSA 2048]----+
|=+o+.o.          |
|o=o+o.oo         |
| .O.=.=          |
| . B.B +         |
|o B * B S        |
| * + * +         |
|  o + .          |
| o  E+ .         |
|o   ..+o.        |
+----[SHA256]-----+
```

4.4.1.3.4 第 3 步：在中控机器上下载 TiDB Ansible

以 tidb 用户登录中控机并进入 /home/tidb 目录。使用以下命令从 TiDB Ansible 项目上下载 TiDB Ansible 4.0 相应 TAG 版本，默认的文件夹名称为 tidb-ansible。

```
git clone -b $tag https://github.com/pingcap/tidb-ansible.git
```

注意：

- $tag 替换为选定的 TAG 版本的值，例如 v4.0.0-beta.2。
- 部署和升级 TiDB 集群需使用对应的 tidb-ansible 版本，通过改 inventory.ini 文件中的版本来混用可能会产生一些错误。
- 请务必按文档操作，将 tidb-ansible 下载到 /home/tidb 目录下，权限为 tidb 用户，不要下载到 /root 下，否则会遇到权限问题。

4.4.1.3.5 第 4 步：在中控机器上安装 TiDB Ansible 及其依赖

以 tidb 用户登录中控机，请务必按以下方式通过 pip 安装 TiDB Ansible 及其相关依赖的指定版本，否则会有兼容问题。目前，TiDB Ansible release-4.0 版本兼容 Ansible 2.5 ~ 2.7.11 (2.5 ≤ Ansible ≤ 2.7.11)。

1. 在中控机器上安装 TiDB Ansible 及其依赖。

```
cd /home/tidb/tidb-ansible && \
sudo pip install -r ./requirements.txt
```

Ansible 及相关依赖的版本信息记录在 tidb-ansible/requirements.txt 文件中。

2. 查看 Ansible 的版本。

```
ansible --version
```

```
ansible 2.7.11
```

4.4.1.3.6 第 5 步：在中控机上配置部署机器 SSH 互信及 sudo 规则

以 tidb 用户登录中控机，然后执行以下步骤：

1. 将你的部署目标机器 IP 添加到 hosts.ini 文件的 [servers] 区块下。

```
cd /home/tidb/tidb-ansible && \
vi hosts.ini
```

```
[servers]
172.16.10.1
172.16.10.2
172.16.10.3
172.16.10.4
172.16.10.5
172.16.10.6

[all:vars]
username = tidb
ntp_server = pool.ntp.org
```

2. 执行以下命令，按提示输入部署目标机器的 root 用户密码。

```
ansible-playbook -i hosts.ini create_users.yml -u root -k
```

该步骤将在部署目标机器上创建 tidb 用户，并配置 sudo 规则，配置中控机与部署目标机器之间的 SSH 互信。

如果要手工配置 SSH 互信及 sudo 免密码，可参考如何手工配置 ssh 互信及 sudo 免密码。

4.4.1.3.7 第 6 步：在部署目标机器上安装 NTP 服务

> 注意：
>
> 如果你的部署目标机器时间、时区设置一致，已开启 NTP 服务且在正常同步时间，此步骤可忽略。可参考如何检测 NTP 服务是否正常。

以 tidb 用户登录中控机，执行以下命令：

```
cd /home/tidb/tidb-ansible && \
ansible-playbook -i hosts.ini deploy_ntp.yml -u tidb -b
```

该步骤将在部署目标机器上使用系统自带软件源联网安装并启动 NTP 服务，服务使用安装包默认的 NTP server 列表，见配置文件 /etc/ntp.conf 中 server 参数。如果使用默认的 NTP server，你的机器需要连接外网。

为了让 NTP 尽快开始同步，启动 NTP 服务前，系统会执行 ntpdate 命令，与用户在 hosts.ini 文件中指定的 ntp_server 同步日期与时间。默认的服务器为 pool.ntp.org，也可替换为你的 NTP server。

4.4.1.3.8 第 7 步：在部署目标机器上配置 CPUfreq 调节器模式

为了让 CPU 发挥最大性能，请将 CPUfreq 调节器模式设置为 performance 模式。如需了解 CPUfreq 的更多信息，可查看使用 CPUFREQ 调控器文档。

查看系统支持的调节器模式

执行以下 cpupower 命令，可查看系统支持的调节器模式：

```
cpupower frequency-info --governors
```

```
analyzing CPU 0:
  available cpufreq governors: performance powersave
```

> 注意：
>
> 本例中系统支持设置 performance 和 powersave 模式。如果返回 Not Available，表示当前系统不支持配置 CPUfreq，跳过该步骤即可。

```
cpupower frequency-info --governors
```

```
analyzing CPU 0:
  available cpufreq governors: Not Available
```

查看系统当前的 CPUfreq 调节器模式

执行以下 cpupower 命令，可查看系统当前的 CPUfreq 调节器模式：

```
cpupower frequency-info --policy
```

```
analyzing CPU 0:
  current policy: frequency should be within 1.20 GHz and 3.20 GHz.
                  The governor "powersave" may decide which speed to use
                  within this range.
```

如上述代码所示，本例中的当前配置是 powersave 模式。

修改调节器模式

你可以通过以下两种方法来修改调节器模式。本例中，当前调节器模式为 powersave，以下命令会将模式变更为 performance。

- 使用 cpupower frequency-set --governor 命令来修改。

  ```
  cpupower frequency-set --governor performance
  ```

- 使用以下命令在部署目标机器上批量设置。

  ```
  ansible -i hosts.ini all -m shell -a "cpupower frequency-set --governor performance" -u tidb
      ↪  -b
  ```

4.4.1.3.9 第 8 步：在部署目标机器上添加数据盘 ext4 文件系统挂载参数

使用 root 用户登录目标机器，将部署目标机器数据盘格式化成 ext4 文件系统，挂载时添加 nodelalloc 和 noatime 挂载参数。nodelalloc 是必选参数，否则 Ansible 安装时检测无法通过；noatime 是可选建议参数。

> **注意：**
>
> 如果你的数据盘已经格式化成 ext4 并挂载了磁盘，可先执行 umount /dev/nvme0n1p1 命令卸载，从编辑 /etc/fstab 文件步骤开始执行，添加挂载参数重新挂载即可。

以 /dev/nvme0n1 数据盘为例，具体操作步骤如下：

1. 查看数据盘。

```
fdisk -l
```

```
Disk /dev/nvme0n1: 1000 GB
```

2. 创建分区表。

```
parted -s -a optimal /dev/nvme0n1 mklabel gpt -- mkpart primary ext4 1 -1
```

> **注意：**
>
> 使用 lsblk 命令查看分区的设备号：对于 nvme 磁盘，生成的分区设备号一般为 nvme0n1p1；对于普通磁盘（例如 /dev/sdb），生成的的分区设备号一般为 sdb1。

3. 格式化文件系统。

```
mkfs.ext4 /dev/nvme0n1p1
```

4. 查看数据盘分区 UUID。

   本例中 nvme0n1p1 的 UUID 为 c51eb23b-195c-4061-92a9-3fad812cc12f。

```
lsblk -f
```

```
NAME    FSTYPE LABEL UUID                                 MOUNTPOINT
sda
├─sda1  ext4         237b634b-a565-477b-8371-6dff0c41f5ab /boot
├─sda2  swap         f414c5c0-f823-4bb1-8fdf-e531173a72ed
└─sda3  ext4         547909c1-398d-4696-94c6-03e43e317b60 /
sr0
nvme0n1
└─nvme0n1p1 ext4         c51eb23b-195c-4061-92a9-3fad812cc12f
```

5. 编辑 /etc/fstab 文件，添加 nodelalloc 挂载参数。

```
vi /etc/fstab
```

```
UUID=c51eb23b-195c-4061-92a9-3fad812cc12f /data1 ext4 defaults,nodelalloc,noatime 0 2
```

6. 挂载数据盘。

```
mkdir /data1 && \
mount -a
```

7. 执行以下命令，如果文件系统为 ext4，并且挂载参数中包含 nodelalloc，则表示已生效。

```
mount -t ext4
```

```
/dev/nvme0n1p1 on /data1 type ext4 (rw,noatime,nodelalloc,data=ordered)
```

4.4.1.3.10 第 9 步：编辑 inventory.ini 文件，分配机器资源

以 tidb 用户登录中控机，编辑 /home/tidb/tidb-ansible/inventory.ini 文件为 TiDB 集群分配机器资源。一个标准的 TiDB 集群需要 6 台机器：2 个 TiDB 实例，3 个 PD 实例，3 个 TiKV 实例。

- 至少需部署 3 个 TiKV 实例。
- 不要将 TiKV 实例与 TiDB 或 PD 实例混合部署在同一台机器上。
- 将第一台 TiDB 机器同时用作监控机。

> 注意：
>
> 请使用内网 IP 来部署集群，如果部署目标机器 SSH 端口非默认的 22 端口，需添加 ansible_port 变量，如 TiDB1 ansible_host=172.16.10.1 ansible_port=5555。
>
> 如果是 ARM 架构的机器，需要将 cpu_architecture 改为 arm64。

你可以根据实际场景从以下两种集群拓扑中选择一种：

- 单机单 TiKV 实例集群拓扑

  默认情况下，建议在每个 TiKV 节点上仅部署一个 TiKV 实例，以提高性能。但是，如果你的 TiKV 部署机器的 CPU 和内存配置是部署建议的两倍或以上，并且一个节点拥有两块 SSD 硬盘或者单块 SSD 硬盘的容量大于 2 TB，则可以考虑部署两实例，但不建议部署两个以上实例。

- 单机多 TiKV 实例集群拓扑

单机单 TiKV 实例集群拓扑

Name	Host IP	Services
node1	172.16.10.1	PD1, TiDB1
node2	172.16.10.2	PD2, TiDB2
node3	172.16.10.3	PD3
node4	172.16.10.4	TiKV1
node5	172.16.10.5	TiKV2
node6	172.16.10.6	TiKV3

```
[tidb_servers]
172.16.10.1
172.16.10.2

[pd_servers]
172.16.10.1
172.16.10.2
172.16.10.3

[tikv_servers]
172.16.10.4
172.16.10.5
172.16.10.6

[monitoring_servers]
172.16.10.1

[grafana_servers]
172.16.10.1

[monitored_servers]
172.16.10.1
172.16.10.2
172.16.10.3
172.16.10.4
172.16.10.5
172.16.10.6
```

单机多 TiKV 实例集群拓扑

以两实例为例：

Name	Host IP	Services
node1	172.16.10.1	PD1, TiDB1
node2	172.16.10.2	PD2, TiDB2
node3	172.16.10.3	PD3

Name	Host IP	Services
node4	172.16.10.4	TiKV1-1, TiKV1-2
node5	172.16.10.5	TiKV2-1, TiKV2-2
node6	172.16.10.6	TiKV3-1, TiKV3-2

```
[tidb_servers]
172.16.10.1
172.16.10.2

[pd_servers]
172.16.10.1
172.16.10.2
172.16.10.3

#### 注意：要使用 TiKV 的 labels，必须同时配置 PD 的 location_labels 参数，否则 labels 设置不生效
    ↪ 。

#### 多实例场景需要额外配置 status 端口，示例如下：
[tikv_servers]
TiKV1-1 ansible_host=172.16.10.4 deploy_dir=/data1/deploy tikv_port=20171 tikv_status_port=20181
    ↪ labels="host=tikv1"
TiKV1-2 ansible_host=172.16.10.4 deploy_dir=/data2/deploy tikv_port=20172 tikv_status_port=20182
    ↪ labels="host=tikv1"
TiKV2-1 ansible_host=172.16.10.5 deploy_dir=/data1/deploy tikv_port=20171 tikv_status_port=20181
    ↪ labels="host=tikv2"
TiKV2-2 ansible_host=172.16.10.5 deploy_dir=/data2/deploy tikv_port=20172 tikv_status_port=20182
    ↪ labels="host=tikv2"
TiKV3-1 ansible_host=172.16.10.6 deploy_dir=/data1/deploy tikv_port=20171 tikv_status_port=20181
    ↪ labels="host=tikv3"
TiKV3-2 ansible_host=172.16.10.6 deploy_dir=/data2/deploy tikv_port=20172 tikv_status_port=20182
    ↪ labels="host=tikv3"

[monitoring_servers]
172.16.10.1

[grafana_servers]
172.16.10.1

[monitored_servers]
172.16.10.1
172.16.10.2
172.16.10.3
172.16.10.4
172.16.10.5
```

```
172.16.10.6

#### 注意：为使 TiKV 的 labels 设置生效，部署集群时必须设置 PD 的 location_labels 参数。
[pd_servers:vars]
location_labels = ["host"]
```

- 服务配置文件参数调整

1. 多实例情况下，需要修改 tidb-ansible/conf/tikv.yml 中 block-cache-size 下面的 capacity 参数：

```
storage:
  block-cache:
    capacity: "1GB"
```

> 注意：
> TiKV 实例数量指每个服务器上 TiKV 的进程数量。
> 推荐设置：capacity = MEM_TOTAL * 0.5 / TiKV 实例数量

2. 多实例情况下，需要修改 tidb-ansible/conf/tikv.yml 中 high-concurrency、normal-concurrency 和 low-concurrency 三个参数：

```
readpool:
  coprocessor:
    # Notice: if CPU_NUM > 8, default thread pool size for coprocessors
    # will be set to CPU_NUM * 0.8.
    # high-concurrency: 8
    # normal-concurrency: 8
    # low-concurrency: 8
```

> 注意：
> 推荐设置：TiKV 实例数量 * 参数值 = CPU 核心数量 * 0.8

3. 如果多个 TiKV 实例部署在同一块物理磁盘上，需要修改 conf/tikv.yml 中的 capacity 参数：

```
raftstore:
  capacity: 0
```

> 注意：
> 推荐配置：capacity = 磁盘总容量 / TiKV 实例数量
> 例如：capacity: "100GB"

4.4.1.3.11 第 10 步：调整 inventory.ini 文件中的变量

本小节介绍如何编辑部署目录的变量和 inventory.ini 文件中的其它变量。

调整部署目录

部署目录通过 deploy_dir 变量控制，默认全局变量已设置为 /home/tidb/deploy，对所有服务生效。如数据盘挂载目录为 /data1，可设置为 /data1/deploy，样例如下：

```
##### Global variables
[all:vars]
deploy_dir = /data1/deploy
```

如为某一服务单独设置部署目录，可在配置服务主机列表时配置主机变量，以 TiKV 节点为例，其他服务类推，请务必添加第一列别名，以免服务混布时混淆。

```
TiKV1-1 ansible_host=172.16.10.4 deploy_dir=/data1/deploy
```

调整其它变量（可选）

> 注意：
> 以下控制变量开启请使用首字母大写 True，关闭请使用首字母大写 False。

变量	含义
cluster_name	集群名称，可调整
cpu_architecture	CPU 体系架构，默认为 amd64，可选 arm64
tidb_version	TiDB 版本，TiDB Ansible 各分支默认已配置
process_supervision	进程监管方式，默认为 systemd，可选 supervise
timezone	新安装 TiDB 集群第一次启动 bootstrap（初始化）时，将 TiDB 全局默认时区设置为该值。TiDB 使用的时区后续可通过 time_zone 全局变量和 session 变量来修改，参考时区支持。默认为 Asia/Shanghai，可选值参考 timzone 列表。
enable_firewalld	开启防火墙，默认不开启，如需开启，请将部署建议 -网络要求 中的端口加入白名单
enable_ntpd	检测部署目标机器 NTP 服务，默认为 True，请勿关闭
set_hostname	根据 IP 修改部署目标机器主机名，默认为 False
enable_binlog	是否部署 Pump 并开启 binlog，默认为 False，依赖 Kafka 集群，参见 zookeeper_addrs 变量
zookeeper_addrs	binlog Kafka 集群的 zookeeper 地址
deploy_without_tidb	KV 模式，不部署 TiDB 服务，仅部署 PD、TiKV 及监控服务，请将 inventory.ini 文件中 tidb_servers 主机组 IP 设置为空。
alertmanager_target	可选：如果你已单独部署 alertmanager，可配置该变量，格式：alertmanager_host:alertmanager_port
grafana_admin_user	Grafana 管理员帐号用户名，默认为 admin

变量	含义
grafana_admin_password	Grafana 管理员帐号密码，默认为 admin，用于 Ansible 导入 Dashboard 和创建 API Key，如后期通过 grafana web 修改了密码，请更新此变量
collect_log_recent_hou	采集日志时，采集最近几个小时的日志，默认为 2 小时
enable_bandwidth_limit	在中控机上从部署目标机器拉取诊断数据时，是否限速，默认为 True，与 collect_bandwidth_limit 变量结合使用
collect_bandwidth_limi	在中控机上从部署目标机器拉取诊断数据时限速多少，单位: Kbit/s，默认 10000，即 10Mb/s，如果是单机多 TiKV 实例部署方式，需除以单机实例个数
prometheus_storage_ret	Prometheus 监控数据的保留时间（默认为 30 天）；2.1.7、3.0 以及之后的 tidb-ansible 版本中，group_vars/monitoring_servers.yml 文件里新增的配置

4.4.1.3.12 第 11 步：部署 TiDB 集群

ansible-playbook 执行 Playbook 时，默认并发为 5。部署目标机器较多时，可添加 -f 参数指定并发数，例如 ansible-playbook deploy.yml -f 10。以下示例使用 tidb 用户作为服务运行用户：

1. 在 tidb-ansible/inventory.ini 文件中，确认 ansible_user = tidb。

```
## Connection
# ssh via normal user
ansible_user = tidb
```

> 注意：
> 不要将 ansible_user 设置为 root 用户，因为 tidb-ansible 限制了服务以普通用户运行。

执行以下命令，如果所有 server 均返回 tidb，表示 SSH 互信配置成功：

```
ansible -i inventory.ini all -m shell -a 'whoami'
```

执行以下命令，如果所有 server 均返回 root，表示 tidb 用户 sudo 免密码配置成功。

```
ansible -i inventory.ini all -m shell -a 'whoami' -b
```

2. 执行 local_prepare.yml playbook，联网下载 TiDB binary 至中控机。

```
ansible-playbook local_prepare.yml
```

3. 初始化系统环境，修改内核参数。

```
ansible-playbook bootstrap.yml
```

4. 部署 TiDB 集群软件。

```
ansible-playbook deploy.yml
```

> **注意：**
>
> Grafana Dashboard 上的 Report 按钮可用来生成 PDF 文件，此功能依赖 `fontconfig` 包和英文字体。如需使用该功能，登录 grafana_servers 机器，用以下命令安装：
>
> ```
> sudo yum install fontconfig open-sans-fonts
> ```

5. 启动 TiDB 集群。

```
ansible-playbook start.yml
```

4.4.1.3.13 测试集群

TiDB 兼容 MySQL，因此可使用 MySQL 客户端直接连接 TiDB。推荐配置负载均衡以提供统一的 SQL 接口。

1. 使用 MySQL 客户端连接 TiDB 集群。TiDB 服务的默认端口为 4000。

```
mysql -u root -h 172.16.10.1 -P 4000
```

2. 通过浏览器访问监控平台。

    - 地址：http://172.16.10.1:3000
    - 默认帐号与密码：`admin`；`admin`

> **注意：**
>
> TiDB 默认会定期收集使用情况信息，并将这些信息分享给 PingCAP 用于改善产品。若要了解所收集的信息详情及如何禁用该行为，请参见遥测。

4.4.1.3.14 常见部署问题

本小节介绍使用 TiDB Ansible 部署 TiDB 集群过程中的常见问题与解决方案。

如何自定义端口

修改 `inventory.ini` 文件，在相应服务 IP 后添加以下主机变量即可：

组件	端口变量	默认端口	说明
TiDB	tidb_port	4000	应用及 DBA 工具访问通信端口
TiDB	tidb_status_port	10080	TiDB 状态信息上报通信端口
TiKV	tikv_port	20160	TiKV 通信端口

组件	端口变量	默认端口	说明
TiKV	tikv_status_port	20180	上报 TiKV 状态的通信端口
PD	pd_client_port	2379	提供 TiDB 和 PD 通信端口
PD	pd_peer_port	2380	PD 集群节点间通信端口
Pump	pump_port	8250	Pump 通信端口
Prometheus	prometheus_port	9090	Prometheus 服务通信端口
Pushgateway	pushgateway_port	9091	TiDB，TiKV，PD 监控聚合和上报端口
Node_exporter	node_exporter_port	9100	TiDB 集群每个节点的系统信息上报通信端口
Blackbox_exporter	blackbox_exporter_port	9115	Blackbox_exporter 通信端口，用于 TiDB 集群端口监控
Grafana	grafana_port	3000	Web 监控服务对外服务和客户端 (浏览器) 访问端口
Kafka_exporter	kafka_exporter_port	9308	Kafka_exporter 通信端口，用于监控 binlog Kafka 集群

如何自定义部署目录

修改 `inventory.ini` 文件，在相应服务 IP 后添加以下主机变量即可：

组件	目录变量	默认目录	说明
全局	deploy_dir	/home/tidb/deploy	部署目录
TiDB	tidb_log_dir	{{ deploy_dir }}/log	日志目录
TiKV	tikv_log_dir	{{ deploy_dir }}/log	日志目录
TiKV	tikv_data_dir	{{ deploy_dir }}/data	数据目录
TiKV	wal_dir	“”	rocksdb write-ahead 日志目录，为空时与 TiKV 数据目录一致
TiKV	raftdb_path	“”	raftdb 目录，为空时为 tikv_data_dir/raft
PD	pd_log_dir	{{ deploy_dir }}/log	日志目录
PD	pd_data_dir	{{ deploy_dir }}/data.pd	数据目录
pump	pump_log_dir	{{ deploy_dir }}/log	日志目录
pump	pump_data_dir	{{ deploy_dir }}/data.pump	数据目录
prometheus	prometheus_log_dir	{{ deploy_dir }}/log	日志目录
prometheus	prometheus_data_dir	{{ deploy_dir }}/data.metrics	数据目录
pushgateway	pushgateway_log_dir	{{ deploy_dir }}/log	日志目录
node_exporter	node_exporter_log_dir	{{ deploy_dir }}/log	日志目录
grafana	grafana_log_dir	{{ deploy_dir }}/log	日志目录
grafana	grafana_data_dir	{{ deploy_dir }}/data.grafana	数据目录

如何检测 NTP 服务是否正常

1. 执行以下命令，如果输出 `running` 表示 NTP 服务正在运行：

```
sudo systemctl status ntpd.service
```

```
ntpd.service - Network Time Service
Loaded: loaded (/usr/lib/systemd/system/ntpd.service; disabled; vendor preset: disabled)
Active: active (running) since 一 2017-12-18 13:13:19 CST; 3s ago
```

2. 执行 ntpstat 命令，如果输出 synchronised to NTP server（正在与 NTP server 同步），表示在正常同步：

```
ntpstat
```

```
synchronised to NTP server (85.199.214.101) at stratum 2
time correct to within 91 ms
polling server every 1024 s
```

> **注意：**
>
> Ubuntu 系统需安装 ntpstat 软件包。

- 以下情况表示 NTP 服务未正常同步：

```
ntpstat
```

```
unsynchronised
```

- 以下情况表示 NTP 服务未正常运行：

```
ntpstat
```

```
Unable to talk to NTP daemon. Is it running?
```

- 如果要使 NTP 服务尽快开始同步，执行以下命令。可以将 pool.ntp.org 替换为你的 NTP server：

```
sudo systemctl stop ntpd.service && \
sudo ntpdate pool.ntp.org && \
sudo systemctl start ntpd.service
```

- 如果要在 CentOS 7 系统上手动安装 NTP 服务，可执行以下命令：

```
sudo yum install ntp ntpdate && \
sudo systemctl start ntpd.service && \
sudo systemctl enable ntpd.service
```

如何调整进程监管方式从 supervise 到 systemd

```
process supervision, [systemd, supervise]
```

```
process_supervision = systemd
```

TiDB Anisble 在 TiDB v1.0.4 版本之前进程监管方式默认为 supervise。之前安装的集群可保持不变，如需更新为 systemd，需关闭集群，按以下方式变更：

```
ansible-playbook stop.yml && \
ansible-playbook deploy.yml -D && \
ansible-playbook start.yml
```

如何手工配置 SSH 互信及 sudo 免密码

1. 以 root 用户依次登录到部署目标机器创建 tidb 用户并设置登录密码。

```
useradd tidb && \
passwd tidb
```

2. 执行以下命令，将 tidb ALL=(ALL)NOPASSWD: ALL 添加到文件末尾，即配置好 sudo 免密码。

```
visudo
```

```
tidb ALL=(ALL) NOPASSWD: ALL
```

3. 以 tidb 用户登录到中控机，执行以下命令。将 172.16.10.61 替换成你的部署目标机器 IP，按提示输入部署目标机器 tidb 用户密码，执行成功后即创建好 SSH 互信，其他机器同理。

```
ssh-copy-id -i ~/.ssh/id_rsa.pub 172.16.10.61
```

4. 以 tidb 用户登录中控机，通过 ssh 的方式登录目标机器 IP。如果不需要输入密码并登录成功，即表示 SSH 互信配置成功。

```
ssh 172.16.10.61
```

```
[tidb@172.16.10.61 ~]$
```

5. 以 tidb 用户登录到部署目标机器后，执行以下命令，不需要输入密码并切换到 root 用户，表示 tidb 用户 sudo 免密码配置成功。

```
sudo -su root
```

```
[root@172.16.10.61 tidb]#
```

You need to install jmespath prior to running json_query filter 报错

1. 请参照在中控机器上安装 TiDB Ansible 及其依赖 在中控机上通过 pip 安装 TiDB Ansible 及相关依赖的指定版本，默认会安装 jmespath。
2. 执行以下命令，验证 jmespath 是否安装成功：

```
pip show jmespath
```

```
Name: jmespath
Version: 0.9.0
```

3. 在中控机上 Python 交互窗口里执行 `import jmespath`。

    - 如果没有报错，表示依赖安装成功。
    - 如果有 `ImportError: No module named jmespath` 报错，表示未成功安装 Python `jmespath` 模块。

```
python
```

```
Python 2.7.5 (default, Nov  6 2016, 00:28:07)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-11)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
```

```
import jmespath
```

启动 Pump/Drainer 报 `zk: node does not exist` 错误

请检查 `inventory.ini` 里的 `zookeeper_addrs` 参数配置与 Kafka 集群内的配置是否相同、是否填写了命名空间。关于命名空间的配置说明如下：

```
#### ZooKeeper connection string (see ZooKeeper docs for details).
#### ZooKeeper address of Kafka cluster, example:
#### zookeeper_addrs = "192.168.0.11:2181,192.168.0.12:2181,192.168.0.13:2181"
#### You can also append an optional chroot string to the URLs to specify the root directory for
    ↪ all Kafka znodes. Example:
#### zookeeper_addrs = "192.168.0.11:2181,192.168.0.12:2181,192.168.0.13:2181/kafka/123"
```

4.4.1.4 离线 TiDB Ansible 部署方案

> **警告：**
>
> 对于生产环境，推荐使用 TiUP 离线部署 TiDB 集群。从 TiDB 4.0 版本开始，PingCAP 不再维护 TiDB Ansible 的部署方式。继续使用 TiDB Ansible 部署 TiDB 集群可能存在风险，因此不推荐该部署方式。TiUP 可直接支持之前使用 TiDB Ansible 方式部署的集群。
>
> 如果只是希望测试 TiDB 或体验 TiDB 特性，可参考TiDB 快速上手指南或者使用 Docker Compose 在单机上快速部署 TiDB 集群。

4.4.1.4.1 准备机器

1. 下载机一台

- 该机器需开放外网访问，用于下载 TiDB Ansible、 TiDB 及相关软件安装包。
- 推荐安装 CentOS 7.3 及以上版本 Linux 操作系统。

2. 部署目标机器若干及部署中控机一台

- 系统要求及配置参考准备机器。
- 可以无法访问外网。

4.4.1.4.2 在中控机上安装系统依赖包

1. 在下载机上下载系统依赖离线安装包，然后上传至中控机。该离线包仅支持 CentOS 7 系统，包含 pip 及 sshpass。

2. 在中控机上安装系统依赖包：

```
tar -xzvf ansible-system-rpms.el7.tar.gz &&
cd ansible-system-rpms.el7 &&
chmod u+x install_ansible_system_rpms.sh &&
./install_ansible_system_rpms.sh
```

3. 安装完成后，可通过 pip -V 验证 pip 是否安装成功：

```
pip -V
```

```
pip 8.1.2 from /usr/lib/python2.7/site-packages (python 2.7)
```

> 注意：
>
> 如果你的系统已安装 pip，请确认版本 >= 8.1.2，否则离线安装 TiDB Ansible 及其依赖时，会有兼容问题。

4.4.1.4.3 在中控机上创建 tidb 用户，并生成 ssh key

参考在中控机上创建 tidb 用户，并生成 ssh key 即可。

4.4.1.4.4 在中控机器上离线安装 TiDB Ansible 及其依赖

以下是 CentOS 7 系统 Ansible 离线安装方式：

建议使用 Ansible 2.4 至 2.7.11 版本，Ansible 及相关依赖版本记录在 tidb-ansible/requirements.txt 文件中。下面步骤以安装 Ansible 2.5 为例。

1. 在下载机上下载 Ansible 2.5 离线安装包，然后上传至中控机。

2. 离线安装 TiDB Ansible 及相关依赖：

```
tar -xzvf ansible-2.5.0-pip.tar.gz &&
cd ansible-2.5.0-pip/ &&
chmod u+x install_ansible.sh &&
./install_ansible.sh
```

3. 安装完成后，可通过 `ansible --version` 查看版本：

```
ansible --version
```

```
ansible 2.5.0
```

4.4.1.4.5 在下载机上下载 TiDB Ansible 及 TiDB 安装包

1. 在下载机上安装 TiDB Ansible：

   请按以下方式在 CentOS 7 系统的下载机上在线安装 TiDB Ansible。安装完成后，可通过 `ansible --version` 查看版本，请务必确认是 Ansible 2.5.0 版本，否则会有兼容问题。

```
yum install epel-release &&
yum install ansible curl &&
ansible --version
```

```
ansible 2.5.0
```

2. 下载 tidb-ansible：

   使用以下命令从 Github TiDB Ansible 项目上下载 TiDB Ansible 相应版本，默认的文件夹名称为 `tidb-ansible`。

```
git clone https://github.com/pingcap/tidb-ansible.git
```

> 注意：
> 部署和升级 TiDB 集群需使用对应的 tidb-ansible 版本，通过改 `inventory.ini` 文件中的版本来混用可能会产生一些错误。

3. 执行 `local_prepare.yml` playbook，联网下载 TiDB binary 到下载机：

```
cd tidb-ansible &&
ansible-playbook local_prepare.yml
```

4. 将执行完以上命令之后的 `tidb-ansible` 文件夹拷贝到中控机 `/home/tidb` 目录下，文件属主权限需是 `tidb` 用户。

4.4.1.4.6 在中控机上配置部署机器 SSH 互信及 sudo 规则

参考在中控机上配置部署机器 SSH 互信及 sudo 规则即可。

4.4.1.4.7 在部署目标机器上安装 NTP 服务

如果你的部署目标机器时间、时区设置一致，已开启 NTP 服务且在正常同步时间，此步骤可忽略，可参考如何检测 NTP 服务是否正常。

参考在部署目标机器上安装 NTP 服务即可。

4.4.1.4.8 在部署目标机器上配置 CPUfreq 调节器模式

参考在部署目标机器上配置 CPUfreq 调节器模式即可。

4.4.1.4.9 在部署目标机器上添加数据盘 ext4 文件系统挂载参数

参考在部署目标机器上添加数据盘 ext4 文件系统挂载参数即可。

4.4.1.4.10 分配机器资源，编辑 inventory.ini 文件

参考分配机器资源，编辑 inventory.ini 文件即可。

4.4.1.4.11 部署任务

1. `ansible-playbook local_prepare.yml` 该 playbook 不需要再执行。
2. 参考部署任务即可。

4.4.1.4.12 测试集群

参考测试集群即可。

> **注意：**
>
> TiDB 默认会定期收集使用情况信息，并将这些信息分享给 PingCAP 用于改善产品。若要了解所收集的信息详情及如何禁用该行为，请参见遥测。

4.4.1.5 在 Kubernetes 上部署 TiDB

你可以使用 TiDB Operator 在 Kubernetes 上部署 TiDB。TiDB Operator 是 Kubernetes 上的 TiDB 集群自动运维系统，提供包括部署、升级、扩缩容、备份恢复、配置变更的 TiDB 全生命周期管理。借助 TiDB Operator，TiDB 可以无缝运行在公有云或自托管的 Kubernetes 集群上。

TiDB Operator 的文档目前独立于 TiDB 文档。要查看如何在 Kubernetes 上部署 TiDB 的详细步骤，请参阅对应版本的 TiDB Operator 文档：

- TiDB in Kubernetes 用户文档

4.5 验证集群运行状态

本文档介绍如何通过TiDB Dashboard 和 Grafana 检查集群状态，以及登录数据库执行简单 DML、DDL 操作和查询 SQL 语句。

4.5.1 通过 TiDB Dashboard 和 Grafana 检查集群状态

本节介绍如何通过TiDB Dashboard 和 Grafana 检查集群状态。

4.5.1.1 查看 TiDB Dashboard 检查 TiDB 集群状态

1. 通过 `{pd-ip}:{pd-port}/dashboard` 登录 TiDB Dashboard，登录用户和口令为 TiDB 数据库 `root` 用户和口令。如果你修改过数据库的 `root` 密码，则以修改后的密码为准，默认密码为空。

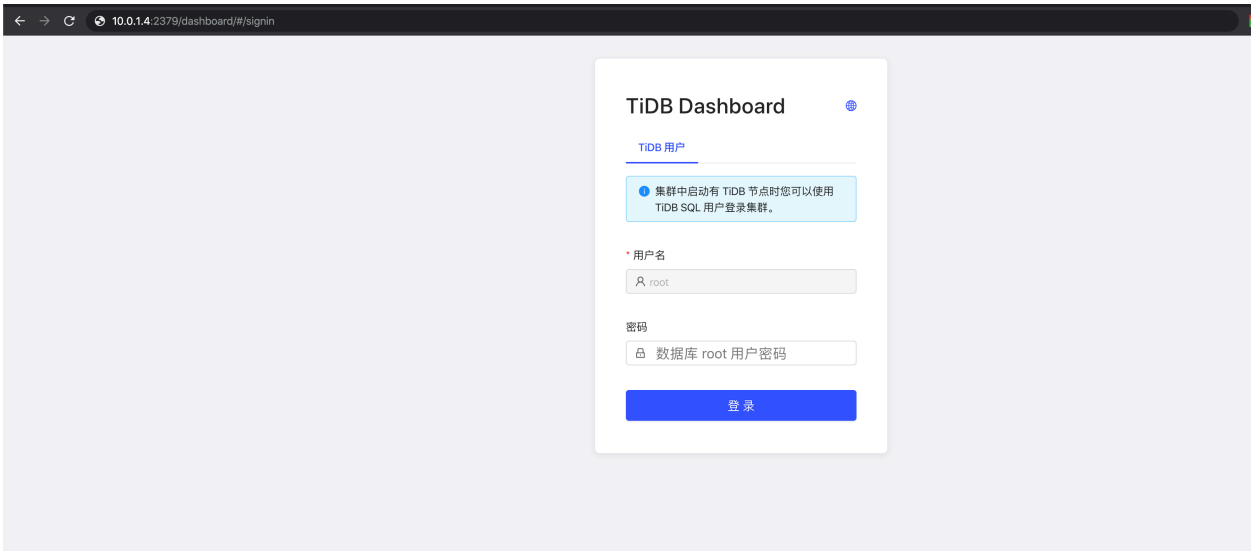


图 21: TiDB-Dashboard

2. 主页面显示 TiDB 集群中节点信息

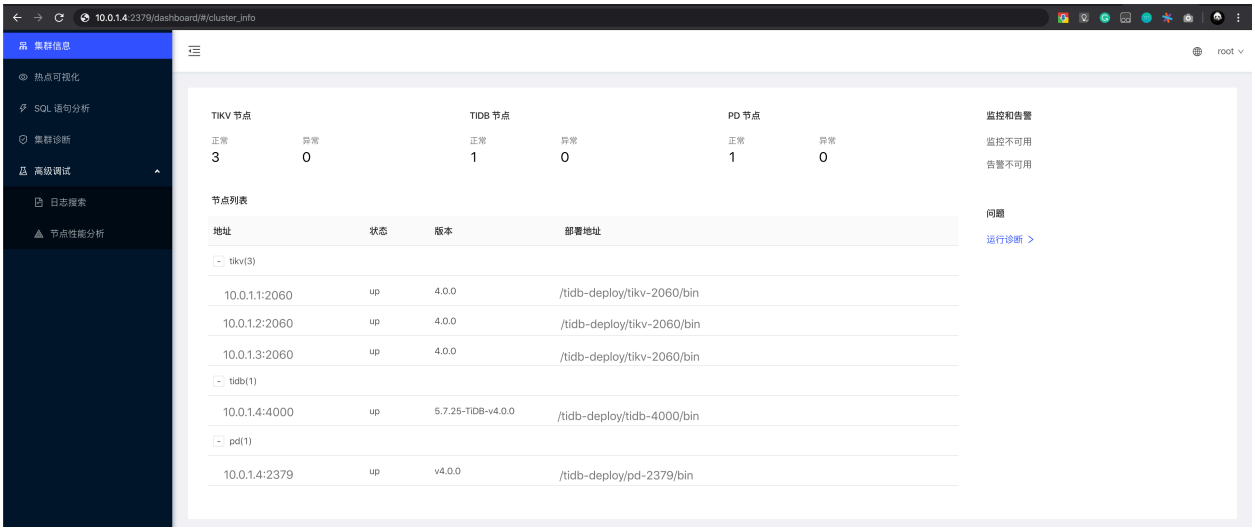


图 22: TiDB-Dashboard-status

4.5.1.2 查看 Grafana 监控 Overview 页面检查 TiDB 集群状态

- 通过 `{Grafana-ip}:3000` 登录 Grafana 监控，默认用户名及密码为 `admin/admin`。
- 点击 Overview 监控页面检查 TiDB 端口和负载监控信息。

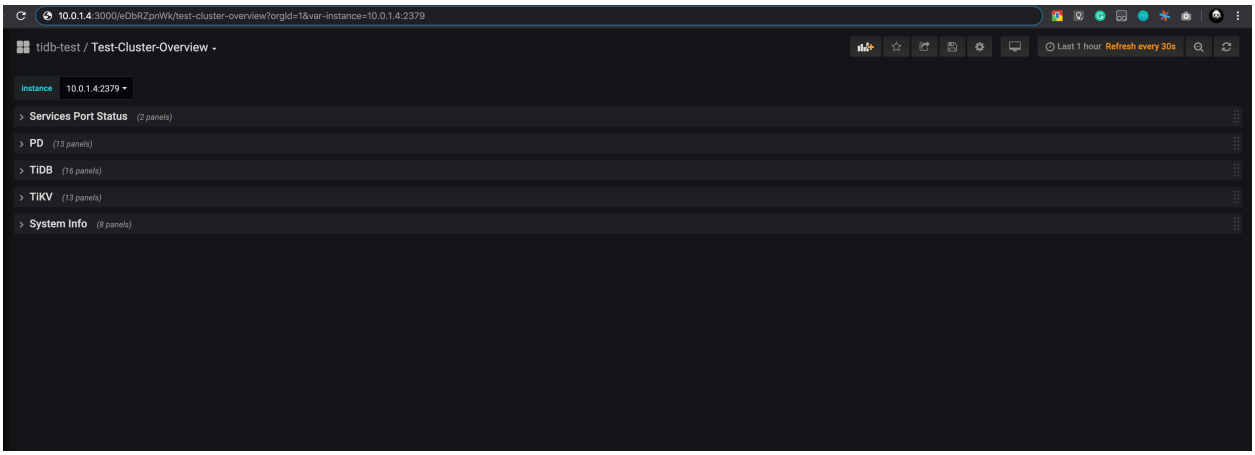


图 23: Grafana-overview

4.5.2 登录数据库执行简单 DML/DDL 操作和查询 SQL 语句

> **注意：**
>
> 登录数据库前，你需要安装 MySQL 客户端。

执行如下命令登录数据库：

```
mysql -u root -h 10.0.1.4 -P 4000
```

输出下列信息表示登录成功：

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3
Server version: 5.7.25-TiDB-v4.0.0 TiDB Server (Apache License 2.0) Community Edition, MySQL 5.7
    ↪ compatible

Copyright (c) 2000, 2015, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
```

4.5.2.1 数据库操作

- 检查 TiDB 版本

```
select tidb_version()\G
```

预期结果输出：

```
*************************** 1. row ***************************
tidb_version(): Release Version: v4.0.0
Edition: Community
Git Commit Hash: 689a6b6439ae7835947fcaccf329a3fc303986cb
Git Branch: HEAD
UTC Build Time: 2020-05-28 11:09:45
GoVersion: go1.13.4
Race Enabled: false
TiKV Min Version: v3.0.0-60965b006877ca7234adaced7890d7b029ed1306
Check Table Before Drop: false
1 row in set (0.00 sec)
```

- 创建 PingCAP database

```
create database pingcap;
```

```
Query OK, 0 rows affected (0.10 sec)
```

```
use pingcap;
```

预期输出

```
Database changed
```

- 创建 tab_tidb 表

```
CREATE TABLE `tab_tidb` (
`id` int(11) NOT NULL AUTO_INCREMENT,
`name` varchar(20) NOT NULL DEFAULT '',
`age` int(11) NOT NULL DEFAULT 0,
`version` varchar(20) NOT NULL DEFAULT '',
PRIMARY KEY (`id`),
KEY `idx_age` (`age`));
```

预期输出

```
Query OK, 0 rows affected (0.11 sec)
```

- 插入数据

```
insert into `tab_tidb` values (1,'TiDB',5,'TiDB-v4.0.0');
```

预期输出

```
Query OK, 1 row affected (0.03 sec)
```

- 查看 tab_tidb 结果

```
select * from tab_tidb;
```

预期输出

```
+----+------+-----+-------------+
| id | name | age | version     |
+----+------+-----+-------------+
|  1 | TiDB |   5 | TiDB-v4.0.0 |
+----+------+-----+-------------+
1 row in set (0.00 sec)
```

- 查看 TiKV store 状态、store_id、存储情况以及启动时间

```
select STORE_ID,ADDRESS,STORE_STATE,STORE_STATE_NAME,CAPACITY,AVAILABLE,UPTIME from
    ↪ INFORMATION_SCHEMA.TIKV_STORE_STATUS;
```

预期输出

```
+--
    ↪ --------+--------------------+-------------+-----------------+----------+-----------+-------------
    ↪
```

```
| STORE_ID | ADDRESS            | STORE_STATE | STORE_STATE_NAME | CAPACITY | AVAILABLE |
    ↪ UPTIME             |
+--
    ↪ --------+--------------------+-------------+------------------+----------+-----------+-------------
    ↪
|        1 | 10.0.1.1:20160 |           0 | Up               | 49.98GiB | 46.3GiB   | 5
    ↪ h21m52.474864026s |
|        4 | 10.0.1.2:20160 |           0 | Up               | 49.98GiB | 46.32GiB  | 5
    ↪ h21m52.522669177s |
|        5 | 10.0.1.3:20160 |           0 | Up               | 49.98GiB | 45.44GiB  | 5
    ↪ h21m52.713660541s |
+--
    ↪ --------+--------------------+-------------+------------------+----------+-----------+-------------
    ↪
3 rows in set (0.00 sec)
```

- 退出

```
exit
```

预期输出

```
Bye
```

4.6 性能测试方法

4.6.1 如何用 Sysbench 测试 TiDB

本次测试使用的是 TiDB 3.0 Beta 和 Sysbench 1.0.14。建议使用 Sysbench 1.0 或之后的更新版本，可在 Sysbench Release 1.0.14 页面下载。

4.6.1.1 测试环境

- 硬件要求
- 参考 TiDB 部署文档部署 TiDB 集群。在 3 台服务器的条件下，建议每台机器部署 1 个 TiDB，1 个 PD，和 1 个 TiKV 实例。关于磁盘，以 32 张表、每张表 10M 行数据为例，建议 TiKV 的数据目录所在的磁盘空间大于 512 GB。对于单个 TiDB 的并发连接数，建议控制在 500 以内，如需增加整个系统的并发压力，可以增加 TiDB 实例，具体增加的 TiDB 个数视测试压力而定。

IDC 机器：

类别	名称
OS	Linux (CentOS 7.3.1611)
CPU	40 vCPUs, Intel® Xeon® CPU E5-2630 v4 @ 2.20GHz

类别	名称
RAM	128GB
DISK	Intel Optane SSD P4800X 375G * 1
NIC	10Gb Ethernet

4.6.1.2 测试方案

4.6.1.2.1 TiDB 版本信息

组件	GitHash
TiDB	7a240818d19ae96e4165af9ea35df92466f59ce6
TiKV	e26ceadcdfe94fb6ff83b5abb614ea3115394bcd
PD	5e81548c3c1a1adab056d977e7767307a39ecb70

4.6.1.2.2 集群拓扑

机器 IP	部署实例
172.16.30.31	3*sysbench
172.16.30.33	1*tidb 1*pd 1*tikv
172.16.30.34	1*tidb 1*pd 1*tikv
172.16.30.35	1*tidb 1*pd 1*tikv

4.6.1.2.3 TiDB 配置

升高日志级别，可以减少打印日志数量，对 TiDB 的性能有积极影响。开启 TiDB 配置中的 `prepared plan cache`，以减少优化执行计划的开销。具体在 TiDB 配置文件中加入：

```
[log]
level = "error"
[prepared-plan-cache]
enabled = true
```

4.6.1.2.4 TiKV 配置

升高 TiKV 的日志级别同样有利于提高性能表现。

TiKV 集群存在多个 Column Family，包括 Default CF、Write CF 和 LockCF，主要用于存储不同类型的数据。对于 Sysbench 测试，需要关注 Default CF 和 Write CF，导入数据的 Column Family 在 TiDB 集群中的比例是固定的。这个比例是：

Default CF : Write CF = 4 : 1

在 TiKV 中需要根据机器内存大小配置 RocksDB 的 block cache，以充分利用内存。以 40 GB 内存的虚拟机部署一个 TiKV 为例，其 block cache 建议配置如下:

```
log-level = "error"
[rocksdb.defaultcf]
block-cache-size = "24GB"
[rocksdb.writecf]
block-cache-size = "6GB"
```

对于 3.0 及以后的版本，还可以使用共享 block cache 的方式进行设置：

```
log-level = "error"
[storage.block-cache]
capacity = "30GB"
```

更详细的 TiKV 参数调优请参考TiKV 内存参数性能调优。

4.6.1.3 测试过程

> **注意：**
>
> 此次测试并没有使用如 HAproxy 等负载均衡工具。在 TiDB 单一节点上进行 Sysbench 测试，并把结果相加。负载均衡工具和不同版本参数也会影响性能表现。

4.6.1.3.1 Sysbench 配置

以下为 Sysbench 配置文件样例：

```
mysql-host={TIDB_HOST}
mysql-port=4000
mysql-user=root
mysql-password=password
mysql-db=sbtest
time=600
threads={8, 16, 32, 64, 128, 256}
report-interval=10
db-driver=mysql
```

可根据实际需求调整其参数，其中 TIDB_HOST 为 TiDB server 的 IP 地址（配置文件中不能写多个地址），threads 为测试中的并发连接数，可在 “8, 16, 32, 64, 128, 256” 中调整，导入数据时，建议设置 threads = 8 或者 16。调整后，将该文件保存为名为 config 的文件。

配置文件参考示例如下：

```
mysql-host=172.16.30.33
mysql-port=4000
mysql-user=root
```

```
mysql-password=password
mysql-db=sbtest
time=600
threads=16
report-interval=10
db-driver=mysql
```

4.6.1.3.2 数据导入

> **注意：**
>
> 如果 TiDB 启用了乐观事务模型（默认为悲观锁模式），当发现并发冲突时，会回滚事务。将 `tidb_disable_txn_auto_retry` 设置为 `off` 会开启事务冲突后的自动重试机制，可以尽可能避免事务冲突报错导致 Sysbench 程序退出的问题。

在数据导入前，需要对 TiDB 进行简单设置。在 MySQL 客户端中执行如下命令：

```
set global tidb_disable_txn_auto_retry = off;
```

然后退出客户端。

重新启动 MySQL 客户端执行以下 SQL 语句，创建数据库 `sbtest`：

```
create database sbtest;
```

调整 Sysbench 脚本创建索引的顺序。Sysbench 按照“建表 -> 插入数据 -> 创建索引”的顺序导入数据。对于 TiDB 而言，该方式会花费更多的导入时间。你可以通过调整顺序来加速数据的导入。

假设使用的 Sysbench 版本为 1.0.14，可以通过以下两种方式来修改：

1. 直接下载为 TiDB 修改好的 oltp_common.lua 文件，覆盖 `/usr/share/sysbench/oltp_common.lua` 文件。
2. 将 `/usr/share/sysbench/oltp_common.lua` 的第 235 行到第 240 行移动到第 198 行以后。

> **注意：**
>
> 此操作为可选操作，仅节约了数据导入时间。

命令行输入以下命令，开始导入数据，config 文件为上一步中配置的文件：

```
sysbench --config-file=config oltp_point_select --tables=32 --table-size=10000000 prepare
```

4.6.1.3.3 数据预热与统计信息收集

数据预热可将磁盘中的数据载入内存的 block cache 中，预热后的数据对系统整体的性能有较大的改善，建议在每次重启集群后进行一次数据预热。

Sysbench 1.0.14 没有提供数据预热的功能，因此需要手动进行数据预热。如果使用更新的 Sysbench 版本，可以使用自带的预热功能。

以 Sysbench 中某张表 sbtest7 为例，执行如下 SQL 语句进行数据预热：

```
SELECT COUNT(pad) FROM sbtest7 USE INDEX (k_7);
```

统计信息收集有助于优化器选择更为准确的执行计划，可以通过 analyze 命令来收集表 sbtest 的统计信息，每个表都需要统计。

```
ANALYZE TABLE sbtest7;
```

4.6.1.3.4 Point select 测试命令

```
sysbench --config-file=config oltp_point_select --tables=32 --table-size=10000000 run
```

4.6.1.3.5 Update index 测试命令

```
sysbench --config-file=config oltp_update_index --tables=32 --table-size=10000000 run
```

4.6.1.3.6 Read-only 测试命令

```
sysbench --config-file=config oltp_read_only --tables=32 --table-size=10000000 run
```

4.6.1.4 测试结果

测试了数据 32 表，每表有 10M 数据。

对每个 tidb-server 进行了 Sysbench 测试，将结果相加，得出最终结果：

4.6.1.4.1 oltp_point_select

类型	Thread	TPS	QPS	avg.latency(ms)	.95.latency(ms)	max.latency(ms)
point_select	3*8	67502.55	67502.55	0.36	0.42	141.92
point_select	3*16	120141.84	120141.84	0.40	0.52	20.99
point_select	3*32	170142.92	170142.92	0.58	0.99	28.08
point_select	3*64	195218.54	195218.54	0.98	2.14	21.82
point_select	3*128	208189.53	208189.53	1.84	4.33	31.02

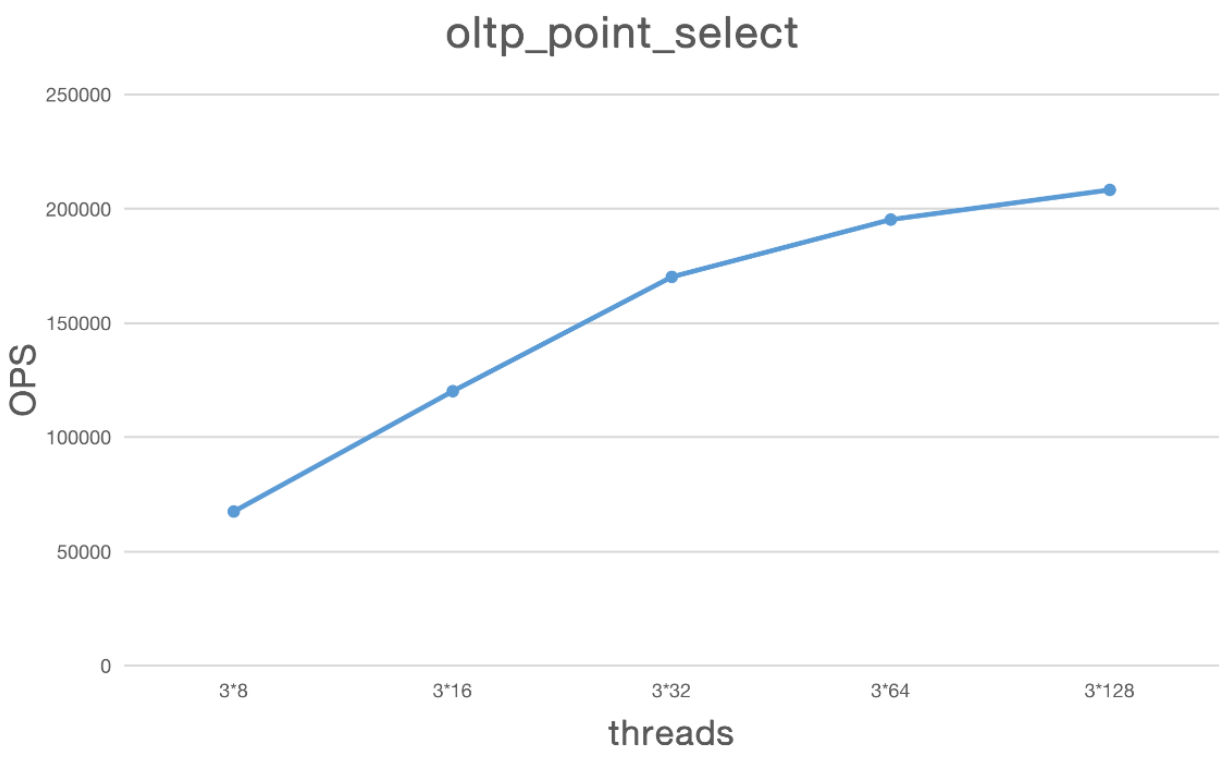


图 24: oltp_point_select

4.6.1.4.2 oltp_update_index

类型	Thread	TPS	QPS	avg.latency(ms)	.95.latency(ms)	max.latency(ms)
oltp_update_index	3*8	9668.98	9668.98	2.51	3.19	103.88
oltp_update_index	3*16	12834.99	12834.99	3.79	5.47	176.90
oltp_update_index	3*32	15955.77	15955.77	6.07	9.39	4787.14
oltp_update_index	3*64	18697.17	18697.17	10.34	17.63	4539.04
oltp_update_index	3*128	20446.81	20446.81	18.98	40.37	5394.75
oltp_update_index	3*256	23563.03	23563.03	32.86	78.60	5530.69

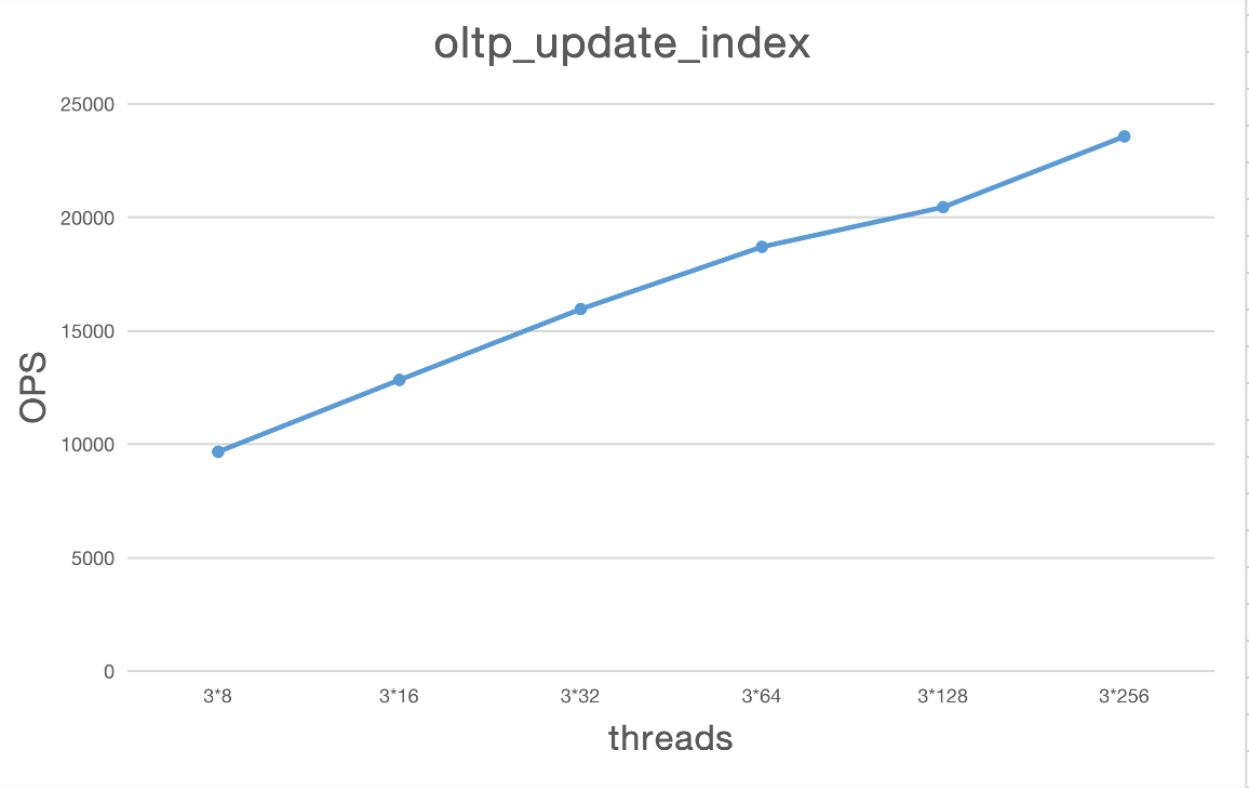


图 25: oltp_update_index

4.6.1.4.3 oltp_read_only

类型	Thread	TPS	QPS	avg.latency(ms)	.95.latency(ms)	max.latency(ms)
oltp_read_only	3*8	2411.00	38575.96	9.92	20.00	92.23
oltp_read_only	3*16	3873.53	61976.50	12.25	16.12	56.94
oltp_read_only	3*32	5066.88	81070.16	19.42	26.20	123.41
oltp_read_only	3*64	5466.36	87461.81	34.65	63.20	231.19
oltp_read_only	3*128	6684.16	106946.59	57.29	97.55	180.85

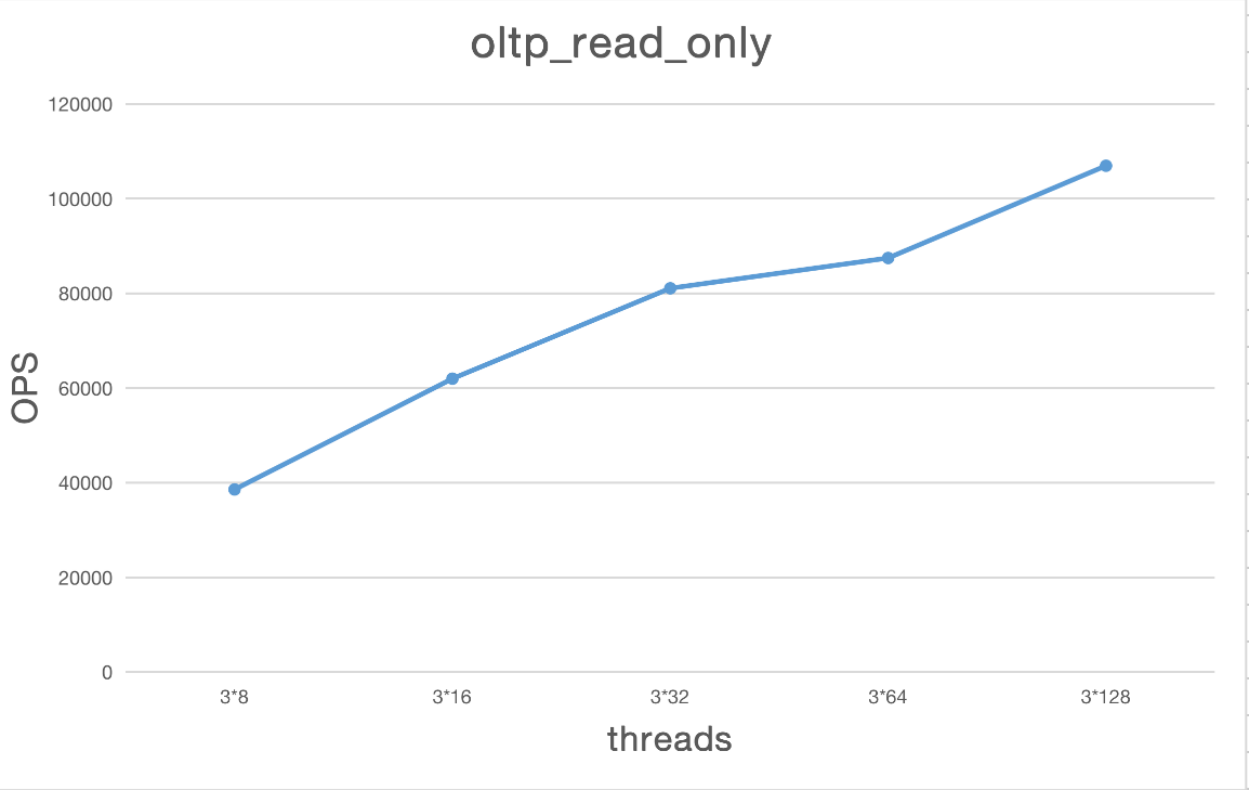


图 26: oltp_read_only

4.6.1.5 常见问题

4.6.1.5.1 在高并发压力下，TiDB、TiKV 的配置都合理，为什么整体性能还是偏低？

这种情况可能与使用了 proxy 有关。可以尝试直接对单个 TiDB 加压，将求和后的结果与使用 proxy 的情况进行对比。

以 HAproxy 为例。`nbproc` 参数可以增加其最大启动的进程数，较新版本的 HAproxy 还支持 `nbthread` 和 `cpu-map` 等。这些都可以减少对其性能的不利影响。

4.6.1.5.2 在高并发压力下，为什么 TiKV 的 CPU 利用率依然很低？

TiKV 虽然整体 CPU 偏低，但部分模块的 CPU 可能已经达到了很高的利用率。

TiKV 的其他模块，如 storage readpool、coprocessor 和 gRPC 的最大并发度限制是可以通过 TiKV 的配置文件进行调整的。

通过 Grafana 的 TiKV Thread CPU 监控面板可以观察到其实际使用率。如出现多线程模块瓶颈，可以通过增加该模块并发度进行调整。

4.6.1.5.3 在高并发压力下，TiKV 也未达到 CPU 使用瓶颈，为什么 TiDB 的 CPU 利用率依然很低？

在某些高端设备上，使用的是 NUMA 架构的 CPU，跨 CPU 访问远端内存将极大降低性能。TiDB 默认将使用服务器所有 CPU，goroutine 的调度不可避免地会出现跨 CPU 内存访问。

因此，建议在 NUMA 架构服务器上，部署 n 个 TiDB（n = NUMA CPU 的个数），同时将 TiDB 的 `max-procs` 变量的值设置为与 NUMA CPU 的核数相同。

4.6.2 如何对 TiDB 进行 TPC-C 测试

本文介绍如何对 TiDB 进行 TPC-C 测试。

4.6.2.1 准备测试程序

TPC-C 是一个对 OLTP（联机交易处理）系统进行测试的规范，使用一个商品销售模型对 OLTP 系统进行测试，其中包含五类事务：

- NewOrder – 新订单的生成
- Payment – 订单付款
- OrderStatus – 最近订单查询
- Delivery – 配送
- StockLevel – 库存缺货状态分析

在测试开始前，TPC-C Benchmark 规定了数据库的初始状态，也就是数据库中数据生成的规则，其中 ITEM 表中固定包含 10 万种商品，仓库的数量可进行调整，假设 WAREHOUSE 表中有 W 条记录，那么：

- STOCK 表中应有 W * 10 万条记录（每个仓库对应 10 万种商品的库存数据）
- DISTRICT 表中应有 W * 10 条记录（每个仓库为 10 个地区提供服务）
- CUSTOMER 表中应有 W * 10 * 3000 条记录（每个地区有 3000 个客户）
- HISTORY 表中应有 W * 10 * 3000 条记录（每个客户一条交易历史）
- ORDER 表中应有 W * 10 * 3000 条记录（每个地区 3000 个订单），并且最后生成的 900 个订单被添加到 NEW-ORDER 表中，每个订单随机生成 5 ~ 15 条 ORDER-LINE 记录。

我们将以 1000 WAREHOUSE 为例进行测试。

TPC-C 使用 tpmC 值（Transactions per Minute）来衡量系统最大有效吞吐量 (MQTh, Max Qualified Throughput)，其中 Transactions 以 NewOrder Transaction 为准，即最终衡量单位为每分钟处理的新订单数。

本文使用开源的 BenchmarkSQL 5.0 作为 TPC-C 测试实现并添加了对 MySQL 协议的支持，可以通过以下命令下载测试程序:

```
git clone -b 5.0-mysql-support-opt-2.1 https://github.com/pingcap/benchmarksql.git
```

安装 java 和 ant，以 CentOS 为例，可以执行以下命令进行安装

```
sudo yum install -y java ant
```

进入 benchmarksql 目录并执行 ant 构建

```
cd benchmarksql
ant
```

4.6.2.2 部署 TiDB 集群

对于 1000 WAREHOUSE，在 3 台服务器上部署集群。

在 3 台服务器的条件下，建议每台机器部署 1 个 TiDB，1 个 PD 和 1 个 TiKV 实例。

比如这里采用的机器硬件配置是：

类别	名称
OS	Linux (CentOS 7.3.1611)
CPU	40 vCPUs, Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz
RAM	128GB
DISK	Optane 500GB SSD

因为该型号 CPU 是 NUMA 架构，建议用 `numactl` 进行绑核。

1. 安装 numactl 工具。
2. 用 `lscpu` 查看 NUMA node，比如：

```
NUMA node0 CPU(s):     0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38
NUMA node1 CPU(s):     1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39
```

3. 通过修改 `{tidb_deploy_path}/scripts/run_tidb.sh` 启动脚本，加入 `numactl` 来启动 TiDB：

```
#!/bin/bash
set -e

ulimit -n 1000000

# WARNING: This file was auto-generated. Do not edit!
#          All your edit might be overwritten!
DEPLOY_DIR=/home/damon/deploy/tidb1-1

cd "${DEPLOY_DIR}" || exit 1

export TZ=Asia/Shanghai

# 同一台机器不同的 TiDB 实例需要指定不同的 cpunodebind 以及 membind；来绑定不同的 Numa node
exec numactl --cpunodebind=0  --membind=0 bin/tidb-server \
    -P 4111 \
    --status="10191" \
    --advertise-address="172.16.4.53" \
    --path="172.16.4.10:2490" \
    --config=conf/tidb.toml \
    --log-slow-query="/home/damon/deploy/tidb1-1/log/tidb_slow_query.log" \
    --log-file="/home/damon/deploy/tidb1-1/log/tidb.log" 2>> "/home/damon/deploy/tidb1-1/log
        ↪ /tidb_stderr.log"
```

> **注意：**
>
> 直接修改 `run_tidb.sh` 可能会被覆盖，因此在生产环境中，如有绑核需求，建议使用 TiUP 绑核。

4. 选择部署一个 HAproxy 来进行多个 TiDB node 的负载均衡，推荐配置 nbproc 为 CPU 核数。

4.6.2.3 配置调整

4.6.2.3.1 TiDB 配置

```
[log]
level = "error"

[performance]
### 根据 NUMA 配置，设置单个 TiDB 最大使用的 CPU 核数
max-procs = 20

[prepared-plan-cache]
### 开启 TiDB 配置中的 prepared plan cache，以减少优化执行计划的开销
enabled = true
```

4.6.2.3.2 TiKV 配置

开始可以使用基本的配置，压测运行后可以通过观察 Grafana 并参考TiKV 线程池调优说明进行调整。

4.6.2.3.3 BenchmarkSQL 配置

修改 `benchmarksql/run/props.mysql`：

```
conn=jdbc:mysql://{HAPROXY-HOST}:{HAPROXY-PORT}/tpcc?useSSL=false&useServerPrepStmts=true&
    ↪ useConfigs=maxPerformance
warehouses=1000 # 使用 1000 个 warehouse
terminals=500   # 使用 500 个终端
loadWorkers=32  # 导入数据的并发数
```

4.6.2.4 导入数据

导入数据通常是整个 TPC-C 测试中最耗时，也是最容易出问题的阶段。

首先用 MySQL 客户端连接到 TiDB-Server 并执行：

```
create database tpcc
```

之后在 shell 中运行 BenchmarkSQL 建表脚本：

```
cd run && \
./runSQL.sh props.mysql sql.mysql/tableCreates.sql && \
./runSQL.sh props.mysql sql.mysql/indexCreates.sql
```

4.6.2.4.1 直接使用 BenchmarkSQL 导入

运行导入数据脚本：

```
./runLoader.sh props.mysql
```

根据机器配置这个过程可能会持续几个小时。

4.6.2.4.2 通过 TiDB Lightning 导入

由于导入数据量随着 warehouse 的增加而增加，当需要导入 1000 warehouse 以上数据时，可以先用 BenchmarkSQL 生成 csv 文件，再将文件通过 TiDB Lightning 导入的方式来快速导入。生成的 csv 文件也可以多次复用，节省每次生成所需要的时间。

修改 BenchmarkSQL 的配置文件

1 warehouse 的 csv 文件需要 77 MB 磁盘空间，在生成之前要根据需要分配足够的磁盘空间来保存 csv 文件。可以在 `benchmarksql/run/props.mysql` 文件中增加一行：

```
fileLocation=/home/user/csv/  # 存储 csv 文件的目录绝对路径，需保证有足够的空间
```

因为最终要使用 TiDB Lightning 导入数据，所以 csv 文件名最好符合其要求，即 `{database}.{table}.csv` 的命名法。这里可以将以上配置改为：

```
fileLocation=/home/user/csv/tpcc.  # 存储 csv 文件的目录绝对路径 + 文件名前缀（database）
```

这样生成的 csv 文件名将会是类似 `tpcc.bmsql_warehouse.csv` 的样式，符合 TiDB Lightning 的要求。

生成 csv 文件

```
./runLoader.sh props.mysql
```

通过 TiDB Lightning 导入

通过 TiDB Lightning 导入数据请参考TiDB Lightning 部署执行章节。这里我们介绍下通过 TiDB Ansible 部署 TiDB Lightning 导入数据的方法。

修改 inventory.ini

这里最好手动指定清楚部署的 IP、端口、目录，避免各种冲突问题带来的异常，其中 import_dir 的磁盘空间参考Lightning 部署执行，data_source_dir 就是存储上一节 csv 数据的目录。

```
[importer_server]
IS1 ansible_host=172.16.5.34 deploy_dir=/data2/is1 tikv_importer_port=13323 import_dir=/data2/
    ↪ import
```

```
[lightning_server]
LS1 ansible_host=172.16.5.34 deploy_dir=/data2/ls1 tidb_lightning_pprof_port=23323
    ↪ data_source_dir=/home/user/csv
```

修改 conf/tidb-lightning.yml

```
mydumper:
    no-schema: true
    csv:
        separator: ','
        delimiter: ''
        header: false
        not-null: false
        'null': 'NULL'
        backslash-escape: true
        trim-last-separator: false
```

部署 TiDB Lightning 和 TiKV Importer

```
ansible-playbook deploy.yml --tags=lightning
```

启动

- 登录到部署 TiDB Lightning 和 TiKV Importer 的服务器
- 进入部署目录
- 在 TiKV Importer 目录下执行 `scripts/start_importer.sh`，启动 TiKV Importer
- 在 TiDB Lightning 目录下执行 `scripts/start_lightning.sh`，开始导入数据

由于是用 TiDB Ansible 进行部署的，可以在监控页面看到 TiDB Lightning 的导入进度，或者通过日志查看导入是否结束。

4.6.2.4.3 导入完成后

数据导入完成之后，可以运行 `sql.common/test.sql` 进行数据正确性验证，如果所有 SQL 语句都返回结果为空，即为数据导入正确。

4.6.2.5 运行测试

执行 BenchmarkSQL 测试脚本：

```
nohup ./runBenchmark.sh props.mysql &> test.log &
```

运行结束后通过 `test.log` 查看结果：

```
07:09:53,455 [Thread-351] INFO   jTPCC : Term-00, Measured tpmC (NewOrders) = 77373.25
07:09:53,455 [Thread-351] INFO   jTPCC : Term-00, Measured tpmTOTAL = 171959.88
07:09:53,455 [Thread-351] INFO   jTPCC : Term-00, Session Start     = 2019-03-21 07:07:52
07:09:53,456 [Thread-351] INFO   jTPCC : Term-00, Session End       = 2019-03-21 07:09:53
07:09:53,456 [Thread-351] INFO   jTPCC : Term-00, Transaction Count = 345240
```

tpmC 部分即为测试结果。

测试完成之后，也可以运行 `sql.common/test.sql` 进行数据正确性验证，如果所有 SQL 语句的返回结果都为空，即为数据测试过程正确。

5 数据迁移

5.1 数据迁移概述

本文档介绍支持从哪些路径将数据迁移到 TiDB，包括从 MySQL 迁移到 TiDB 和从 CSV/SQL 文件迁移到 TiDB。

5.1.1 从 Aurora 迁移到 TiDB

在云环境下，可以直接通过 Aurora Snapshot 导出的方式，将全量数据迁移至 TiDB。详细可参考使用 TiDB Lightning 从 Amazon Aurora MySQL 迁移全量数据。

5.1.2 从 MySQL 迁移到 TiDB

目前推荐使用以下两种方式将 MySQL 数据迁移到 TiDB。

- 使用 Dumpling 和 TiDB Lightning 迁移全量数据
- 使用 DM 迁移数据

5.1.2.1 使用 Dumpling 和 TiDB Lightning 迁移全量数据

5.1.2.1.1 适合场景

适合 MySQL 全量数据的大小大于 1TB 的场景。该方案只能迁移全量数据，如果需要继续同步增量数据，需要再使用 TiDB Data Migration (DM) 创建增量同步任务。

5.1.2.1.2 迁移方法

使用 Dumpling 导出 MySQL 的全量数据，再使用 TiDB Lightning 将全量数据导入到 TiDB，详细信息可参考使用 Dumpling 与 TiDB Lightning 进行全量迁移

5.1.2.2 使用 DM 迁移数据

5.1.2.2.1 适合场景

适合迁移 MySQL 全量数据并同步增量数据的场景，且全量数据的大小小于 1TB。如果全量数据的大小大于 1TB，建议使用 Dumpling 和 TiDB Lightning 导入全量数据后，再使用 DM 同步增量数据。

5.1.2.2.2 迁移方法

DM 支持将 MySQL 全量数据迁移到 TiDB，并同步 MySQL 的增量数据到 TiDB，详细信息可参考使用 DM 工具从 Amazon Aurora MySQL 迁移

5.1.3 从文件迁移到 TiDB

支持通过 CSV 和 SQL 两种格式文件将数据迁移到 TiDB。

5.1.3.1 从 CSV 文件迁移到 TiDB

5.1.3.1.1 适合场景

适合将不兼容 MySQL 协议的异构数据库的数据迁移到 TiDB。

5.1.3.1.2 迁移方法

将全量数据导出到 CSV 格式的文件中，将 CSV 文件导入到 TiDB 有以下两种方法：

- 使用 TiDB Lightning 将 CSV 格式的数据导入到 TiDB

  TiDB Lightning 导入速度快，适合 CSV 文件数据量较大的场景。详细信息可参考从 CSV 文件迁移至 TiDB。

- 使用 LOAD DATA 语句将 CSV 格式的数据导入到 TiDB

  在 TiDB 中执行 LOAD DATA SQL 语句导入 CSV 格式的数据，这种导入方法使用比较方便，但是如果在导入过程中出现错误或者中断，需要人工介入，检查数据的一致性和完整性，因此不建议在生产环境中使用。详细信息可参考LOAD DATA。

5.1.3.2 从 SQL 文件迁移到 TiDB

该部分内容与使用 Dumpling 和 TiDB Lightning 迁移全量数据相同。

5.2 从 MySQL 迁移至 TiDB

5.2.1 使用 TiDB Lightning 从 Amazon Aurora MySQL 迁移全量数据

本文介绍如何使用 TiDB Lightning 从 Amazon Aurora MySQL 迁移全量数据到 TiDB。

5.2.1.1 第一步：从 Aurora 导出全量数据至 Amazon S3

请参考 AWS 官方文档：将数据库快照数据导出到 Amazon S3 将 Aurora 数据的快照导出到 Amazon S3。

5.2.1.2 第二步：部署 TiDB Lightning

具体的部署方法见TiDB Lightning 部署。

5.2.1.3 第三步：配置 TiDB Lightning 的数据源

根据部署方式不同，按如下步骤编辑配置文件 `tidb-lighting.toml`。

1. 将配置文件中 `[mydumper]` 部分的 `data-source-dir` 设置为第一步导出的 S3 Bucket 路径。

```
[mydumper]
# 数据源目录
data-source-dir = "s3://bucket-name/data-path"
```

2. 增加目标集群 TiDB 的配置。

```
[tidb]
# 目标集群的信息。tidb-server 的地址，填一个即可
host = "172.16.31.1"
port = 4000
user = "root"
password = ""
# 集群的 PD 地址
pd-addr = "127.0.0.1:2379"
```

3. 设置后端模式。

```
[tikv-importer]
# 使用 Local-backend
backend = "local"
# 本地临时文件的存储路径。请确保对应的路径不存在或目录为空，
    ↪ 并且所在的磁盘空间须大于待导入数据集的大小。
sorted-kv-dir = "/path/to/local-temp-dir"
```

4. 设置文件路由。

```
[mydumper]
no-schema = true

[[mydumper.files]]
# 使用单引号字符串避免转义
pattern = '(?i)^(?:[^/]*/)*([a-z0-9_]+)\.([a-z0-9_]+)/(?:[^/]*/)*(?:[a-z0-9\-_.]+\.(parquet)
    ↪ )$'
schema = '$1'
table = '$2'
type = '$3'
```

> 注意：

- 本示例选择使用 Local-backend，可以提供最优性能。你也可根据实际需要使用 TiDB-backend 或 Importer-backend。三种后端模式的具体差别参见TiDB Lightning Backend。
- 由于从 Aurora 导出快照数据的路径与 TiDB Lightning 支持的默认文件命名格式不同，因此需要额外设置文件路由配置。
- 如果目标 TiDB 集群开启了 TLS，还需要进行 TLS 相关的设置。

其它配置参考TiDB Lightning 配置。

5.2.1.4 第四步：创建表结构

由于从 Aurora 导出至 S3 的快照数据没有包含用于创建库表的 SQL 语句文件，所以需要手动将对应库表的建表语句导出并导入至 TiDB。可以使用 Dumpling + TiDB Lightning 的方式创建所有表结构：

1. 使用 Dumpling 导出表结构文件：

```
./dumpling --host 127.0.0.1 --port 4000 --user root --password password --no-data --output
    ↪ ./schema --filter "mydb.*"
```

注意：

- 请根据实际情况设置数据源地址的相关参数和输出文件的路径。
- 如果需要导出所有库表，则不需要设置 `--filter` 相关参数。如果只需要导出部分库表，可参考 table-filter 进行设置。

2. 使用 TiDB Lightning 创建表结构

```
./tidb-lightning -config tidb-lightning.toml -d ./schema -no-schema=false
```

此次启动 TiDB Lightning 只用于创建表结构，通常能迅速执行完以上命令。在常规速度下，每秒可以执行约 10 条建表语句。

注意：

- 如果需要创建的库表较少，也可以直接手动在 TiDB 创建对应的库和表，或者使用 mysqldump 等其他工具导出 Schema 然后导入至 TiDB。

5.2.1.5 第五步：开启 TiDB Lightning 进行数据导入

运行 TiDB Lightning。如果直接在命令行中用 nohup 启动程序，可能会因为 SIGHUP 信号而退出，建议把 nohup 放到脚本里面，如：

```
### !/bin/bash
export AWS_ACCESS_KEY_ID=${AccessKey}
export AWS_SECRET_ACCESS_KEY=${SecretKey}
nohup ./tidb-lightning -config tidb-lightning.toml > nohup.out &
```

导入开始后，可以采用以下两种方式查看进度：

- 通过 grep 日志关键字 progress 查看进度，默认 5 分钟更新一次。
- 通过监控面板查看进度，具体参见TiDB-Lightning 监控。

5.2.2 使用 TiDB Lightning 从 MySQL SQL 文件迁移数据

本文介绍如何使用 TiDB Lightning 从 MySQL SQL 文件迁移数据到 TiDB。关于如何生成 MySQL SQL 文件，可以参考Dumpling 文档。

5.2.2.1 第 1 步：部署 TiDB Lightning

使用 TiDB Lightning 将数据导入 TiDB。TiDB Lightning 具体的部署方法见TiDB Lightning 部署。

> 注意：
>
> - 如果选用 Local-backend 来导入数据，导入期间集群无法提供正常的服务，速度更快，适用于导入大量的数据（TB 以上级别）。
> - 如果选用 TiDB-backend 来导入数据，导入期间集群可以正常提供服务，但相对导入速度较慢。
> - 二者的具体差别参见TiDB Lightning Backend。

5.2.2.2 第 2 步：配置 TiDB Lightning 的数据源

本文以选用 TiDB-backend 导入数据为例。增加 `tidb-lightning.toml` 配置文件，在文件中添加以下主要配置：

1. 将 `[mydumper]` 下的 `data-source-dir` 设置为 MySQL 的 SQL 文件路径。

```
[mydumper]
# 数据源目录
data-source-dir = "/data/export"
```

> 注意：
> 如果下游已经存在对应的 schema，那么可以设置 `no-schema=true` 来跳过 schema 创建的步骤。

2. 增加目标集群 TiDB 的配置。

```
[tidb]
# 目标集群的信息。tidb-server 的地址，填一个即可
host = "172.16.31.1"
port = 4000
user = "root"
password = ""
```

3. 增加 TiDB-backend 的必要参数。本文采用 TiDB-backend 模式。此处也可以根据实际应用场景设置为 “local” 或 “importer”。具体请参考后端模式。

```
[tikv-importer]
backend = "tidb"
```

4. 增加导入 TiDB 集群必要参数

```
[tidb]
host = "{{tidb-host}}"
port = {{tidb-port}}
user = "{{tidb-user}}"
password = "{{tidb-password}}"
```

其它配置参考TiDB Lightning 配置。

5.2.2.3 第 3 步：开启 TiDB Lightning 进行数据导入

运行 TiDB Lightning。如果直接在命令行中用 nohup 启动程序，可能会因为 SIGHUP 信号而退出，建议把 nohup 放到脚本里面，如：

```
### !/bin/bash
nohup ./tidb-lightning -config tidb-lightning.toml > nohup.out &
```

导入开始后，可以采用以下两种方式查看进度：

- 通过 grep 日志关键字 progress 查看进度，默认 5 分钟更新一次。
- 通过监控面板查看进度，具体参见TiDB Lightning 监控。

5.2.3 使用 DM 从 Amazon Aurora MySQL 迁移数据

若要从 MySQL 协议数据库迁移数据到 TiDB，可参阅 DM 文档从 Aurora 迁移到 TiDB。

5.3 从 CSV 文件迁移至 TiDB

5.3.1 CSV 支持与限制

本文介绍如何使用 TiDB Lightning 从 CSV 文件迁移数据到 TiDB。关于如何从 MySQL 生成 CSV 文件，可以参考使用 Dumpling 导出到 CSV 文件。

TiDB Lightning 支持读取 CSV（逗号分隔值）的数据源，以及其他定界符格式如 TSV（制表符分隔值）。

5.3.1.1 文件名

包含整张表的 CSV 文件需命名为 `db_name.table_name.csv`，该文件会被解析为数据库 `db_name` 里名为 `table_name` 的表。

如果一个表分布于多个 CSV 文件，这些 CSV 文件命名需加上文件编号的后缀，如 `db_name.table_name.003.csv`。数字部分不需要连续但必须递增，并用零填充。

文件扩展名必须为 `*.csv`，即使文件的内容并非逗号分隔。

5.3.1.2 表结构

CSV 文件是没有表结构的。要导入 TiDB，就必须为其提供表结构。可以通过以下任一方法实现：

- 创建包含 DDL 语句 `CREATE TABLE` 的文件 `db_name.table_name-schema.sql` 以及包含 `CREATE DATABASE` DDL 语句的文件 `db_name-schema-create.sql`。
- 首先在 TiDB 中直接创建空表，然后在 `tidb-lightning.toml` 中设置 `[mydumper] no-schema = true`。

5.3.1.3 配置

CSV 格式可在 `tidb-lightning.toml` 文件中 `[mydumper.csv]` 下配置。大部分设置项在 MySQL `LOAD DATA` 语句中都有对应的项目。

```
[mydumper.csv]
### 字段分隔符，必须为 ASCII 字符。
separator = ','
### 引用定界符，可以为 ASCII 字符或空字符。
delimiter = '"'
### CSV 文件是否包含表头。
### 如果为 true，首行将会被跳过。
header = true
### CSV 是否包含 NULL。
### 如果为 true，CSV 文件的任何列都不能解析为 NULL。
not-null = false
### 如果 `not-null` 为 false（即 CSV 可以包含 NULL），
### 为以下值的字段将会被解析为 NULL。
null = '\N'
### 是否解析字段内的反斜线转义符。
backslash-escape = true
```

```
### 是否移除以分隔符结束的行。
trim-last-separator = false
```

5.3.1.3.1 separator

- 指定字段分隔符。
- 必须为单个 ASCII 字符。
- 常用值：
  - CSV 用 ','
  - TSV 用 "\t"
- 对应 LOAD DATA 语句中的 FIELDS TERMINATED BY 项。

5.3.1.3.2 delimiter

- 指定引用定界符。
- 如果 delimiter 为空，所有字段都会被取消引用。
- 常用值：
  - '"' 使用双引号引用字段，和 RFC 4180 一致。
  - '' 不引用
- 对应 LOAD DATA 语句中的 FIELDS ENCLOSED BY 项。

5.3.1.3.3 header

- 是否所有 CSV 文件都包含表头行。
- 如为 true，第一行会被用作列名。如为 false，第一行并无特殊性，按普通的数据行处理。

5.3.1.3.4 not-null 和 null

- not-null 决定是否所有字段不能为空。
- 如果 not-null 为 false，设定了 null 的字符串会被转换为 SQL NULL 而非具体数值。
- 引用不影响字段是否为空。

  例如有如下 CSV 文件：

  ```
  A,B,C
  \N,"\N",
  ```

  在默认设置（not-null = false; null = '\N'）下，列 A and B 导入 TiDB 后都将会转换为 NULL。列 C 是空字符串 ''，但并不会解析为 NULL。

5.3.1.3.5 backslash-escape

- 是否解析字段内的反斜线转义符。
- 如果 backslash-escape 为 true，下列转义符会被识别并转换。

转义符	转换为
\0	空字符 (U+0000)
\b	退格 (U+0008)
\n	换行 (U+000A)
\r	回车 (U+000D)
\t	制表符 (U+0009)
\Z	Windows EOF (U+001A)

  其他情况下（如 \"）反斜线会被移除，仅在字段中保留其后面的字符（"），这种情况下，保留的字符仅作为普通字符，特殊功能（如界定符）都会失效。

- 引用不会影响反斜线转义符的解析与否。
- 对应 LOAD DATA 语句中的 FIELDS ESCAPED BY '\' 项。

5.3.1.3.6 trim-last-separator

- 将 separator 字段当作终止符，并移除尾部所有分隔符。

  例如有如下 CSV 文件：

```
A,,B,,
```

- 当 trim-last-separator = false，该文件会被解析为包含 5 个字段的行 ('A', '', 'B', '', '')。
- 当 trim-last-separator = true，该文件会被解析为包含 3 个字段的行 ('A', '', 'B')。

5.3.1.3.7 不可配置项

TiDB Lightning 并不完全支持 LOAD DATA 语句中的所有配置项。例如：

- 行终止符只能是 CR（\r），LF（\n）或 CRLF（\r\n），也就是说，无法自定义 LINES TERMINATED BY。
- 不可使用行前缀（LINES STARTING BY）。
- 不可跳过表头（IGNORE n LINES）。如有表头，必须是有效的列名。
- 定界符和分隔符只能为单个 ASCII 字符。

5.3.1.4 设置 strict-format 启用严格格式

导入文件的大小统一约为 256 MB 时，TiDB Lightning 可达到最佳工作状态。如果导入单个 CSV 大文件，TiDB Lightning 只能使用一个线程来处理，这会降低导入速度。

要解决此问题，可先将 CSV 文件分割为多个文件。对于通用格式的 CSV 文件，在没有读取整个文件的情况下无法快速确定行的开始和结束位置。因此，默认情况下 TiDB Lightning 不会自动分割 CSV 文件。但如果你确定待导入的 CSV 文件符合特定的限制要求，则可以启用 `strict-format` 设置。启用后，TiDB Lightning 会将单个 CSV 大文件分割为单个大小为 256 MB 的多个文件块进行并行处理。

```
[mydumper]
strict-format = true
```

严格格式的 CSV 文件中，每个字段仅占一行，即必须满足以下条件之一：

- 分隔符为空；
- 每个字段不包含 CR（`\r`）或 LF（`\n`）。

如果 CSV 文件不是严格格式但 `strict-format` 被误设为 `true`，跨多行的单个完整字段会被分割成两部分，导致解析失败，甚至不报错地导入已损坏的数据。

5.3.1.5 通用配置

5.3.1.5.1 CSV

默认设置已按照 RFC 4180 调整。

```
[mydumper.csv]
separator = ','
delimiter = '"'
header = true
not-null = false
null = '\N'
backslash-escape = true
trim-last-separator = false
```

示例内容：

```
ID,Region,Count
1,"East",32
2,"South",\N
3,"West",10
4,"North",39
```

5.3.1.5.2 TSV

```
[mydumper.csv]
separator = "\t"
delimiter = ''
header = true
not-null = false
```

```
null = 'NULL'
backslash-escape = false
trim-last-separator = false
```

示例内容：

```
ID      Region      Count
1       East        32
2       South       NULL
3       West        10
4       North       39
```

5.3.1.5.3 TPC-H DBGEN

```
[mydumper.csv]
separator = '|'
delimiter = ''
header = false
not-null = true
backslash-escape = false
trim-last-separator = true
```

示例内容：

```
1|East|32|
2|South|0|
3|West|10|
4|North|39|
```

5.3.2 LOAD DATA

`LOAD DATA` 语句用于将数据批量加载到 TiDB 表中。

5.3.2.1 语法图

```
LoadDataStmt ::=
    'LOAD' 'DATA' LocalOpt 'INFILE' stringLit DuplicateOpt 'INTO' 'TABLE' TableName CharsetOpt
        ↪ Fields Lines IgnoreLines ColumnNameOrUserVarListOptWithBrackets LoadDataSetSpecOpt
```

5.3.2.2 参数说明

用户可以使用 `LocalOpt` 参数来指定导入的数据文件位于客户端或者服务端。目前 TiDB 只支持从客户端进行数据导入，因此在导入数据时 `LocalOpt` 应设置成 `Local`。

用户可以使用 Fields 和 Lines 参数来指定如何处理数据格式，使用 FIELDS TERMINATED BY 来指定每个数据的分隔符号，使用 FIELDS ENCLOSED BY 来指定消除数据的包围符号。如果用户希望以某个字符为结尾切分每行数据，可以使用 LINES TERMINATED BY 来指定行的终止符。

例如对于以下格式的数据：

```
"bob","20","street 1"\r\n
"alice","33","street 1"\r\n
```

如果想分别提取 bob、20、street 1，可以指定数据的分隔符号为 ','，数据的包围符号为 '\"'。可以写成：

```
FIELDS TERMINATED BY ',' ENCLOSED BY '\"' LINES TERMINATED BY '\r\n'
```

如果不指定处理数据的参数，将会按以下参数处理

```
FIELDS TERMINATED BY '\t' ENCLOSED BY ''
LINES TERMINATED BY '\n'
```

用户可以通过 IGNORE number LINES 参数来忽略文件开始的 number 行，例如可以使用 IGNORE 1 LINES 来忽略文件的首行。

另外，TiDB 目前对参数 DuplicateOpt、CharsetOpt、LoadDataSetSpecOpt 仅支持语法解析。

5.3.2.3 示例

```
CREATE TABLE trips (
    trip_id bigint NOT NULL PRIMARY KEY AUTO_INCREMENT,
    duration integer not null,
    start_date datetime,
    end_date datetime,
    start_station_number integer,
    start_station varchar(255),
    end_station_number integer,
    end_station varchar(255),
    bike_number varchar(255),
    member_type varchar(255)
    );
```

```
Query OK, 0 rows affected (0.14 sec)
```

通过 LOAD DATA 导入数据，指定数据的分隔符为逗号，忽略包围数据的引号，并且忽略文件的第一行数据。

如果此时遇到 ERROR 1148 (42000): the used command is not allowed with this TiDB version 报错信息。可以参考以下文档解决：

ERROR 1148 (42000): the used command is not allowed with this TiDB version 问题的处理方法

```
LOAD DATA LOCAL INFILE '/mnt/evo970/data-sets/bikeshare-data/2017Q4-capitalbikeshare-tripdata.csv
    ↪ ' INTO TABLE trips FIELDS TERMINATED BY ',' ENCLOSED BY '\"' LINES TERMINATED BY '\r\n'
    ↪ IGNORE 1 LINES (duration, start_date, end_date, start_station_number, start_station,
    ↪ end_station_number, end_station, bike_number, member_type);
```

```
Query OK, 815264 rows affected (39.63 sec)
Records: 815264  Deleted: 0  Skipped: 0  Warnings: 0
```

LOAD DATA 也支持使用十六进制 ASCII 字符表达式或二进制 ASCII 字符表达式作为 FIELDS ENCLOSED BY 和 FIELDS TERMINATED BY 的参数。示例如下：

```
LOAD DATA LOCAL INFILE '/mnt/evo970/data-sets/bikeshare-data/2017Q4-capitalbikeshare-tripdata.csv
    ↪ ' INTO TABLE trips FIELDS TERMINATED BY x'2c' ENCLOSED BY b'100010' LINES TERMINATED BY '
    ↪ \r\n' IGNORE 1 LINES (duration, start_date, end_date, start_station_number, start_station
    ↪ , end_station_number, end_station, bike_number, member_type);
```

以上示例中 x'2c' 是字符 , 的十六进制表示，b'100010' 是字符 " 的二进制表示。

5.3.2.4 MySQL 兼容性

- 默认情况下，TiDB 每 20,000 行会进行一次提交。这类似于 MySQL NDB Cluster，但并非 InnoDB 存储引擎的默认配置。

> 注意：
>
> - 这种拆分事务提交的方式是以打破事务的原子性和隔离性为代价的，使用该特性时，使用者需要保证没有其他对正在处理的表的任何操作，并且在出现报错时，需要及时人工介入，检查数据的一致性和完整性。因此，不建议对读写频繁的表使用 LOAD DATA 语句。
> - 无论以多少行为一个事务提交，LOAD DATA 都不会被显式事务中的ROLLBACK 语句回滚。
> - LOAD DATA 语句始终以乐观事务模式执行，不受 TiDB 事务模式设置的影响。

5.3.2.5 另请参阅

- INSERT
- 乐观事务模型
- 悲观事务模式

5.4 使用 TiDB Lightning 从 MySQL SQL 文件迁移数据

本文介绍如何使用 TiDB Lightning 从 MySQL SQL 文件迁移数据到 TiDB。关于如何生成 MySQL SQL 文件，可以参考Dumpling 文档。

5.4.1 第 1 步：部署 TiDB Lightning

使用 TiDB Lightning 将数据导入 TiDB。TiDB Lightning 具体的部署方法见TiDB Lightning 部署。

注意：

- 如果选用 Local-backend 来导入数据，导入期间集群无法提供正常的服务，速度更快，适用于导入大量的数据（TB 以上级别）。
- 如果选用 TiDB-backend 来导入数据，导入期间集群可以正常提供服务，但相对导入速度较慢。
- 二者的具体差别参见TiDB Lightning Backend。

5.4.2 第 2 步：配置 TiDB Lightning 的数据源

本文以选用 TiDB-backend 导入数据为例。增加 `tidb-lightning.toml` 配置文件，在文件中添加以下主要配置：

1. 将 `[mydumper]` 下的 `data-source-dir` 设置为 MySQL 的 SQL 文件路径。

```
[mydumper]
# 数据源目录
data-source-dir = "/data/export"
```

注意：
如果下游已经存在对应的 schema，那么可以设置 `no-schema=true` 来跳过 schema 创建的步骤。

2. 增加目标集群 TiDB 的配置。

```
[tidb]
# 目标集群的信息。tidb-server 的地址，填一个即可
host = "172.16.31.1"
port = 4000
user = "root"
password = ""
```

3. 增加 TiDB-backend 的必要参数。本文采用 TiDB-backend 模式。此处也可以根据实际应用场景设置为“local”或“importer”。具体请参考后端模式。

```
[tikv-importer]
backend = "tidb"
```

4. 增加导入 TiDB 集群必要参数

```
[tidb]
host = "{{tidb-host}}"
port = {{tidb-port}}
```

```
user = "{{tidb-user}}"
password = "{{tidb-password}}"
```

其它配置参考TiDB Lightning 配置。

5.4.3 第 3 步：开启 TiDB Lightning 进行数据导入

运行 TiDB Lightning。如果直接在命令行中用 nohup 启动程序，可能会因为 SIGHUP 信号而退出，建议把 nohup 放到脚本里面，如：

```
## !/bin/bash
nohup ./tidb-lightning -config tidb-lightning.toml > nohup.out &
```

导入开始后，可以采用以下两种方式查看进度：

- 通过 grep 日志关键字 progress 查看进度，默认 5 分钟更新一次。
- 通过监控面板查看进度，具体参见TiDB Lightning 监控。

6 运维操作

6.1 升级 TiDB 版本

6.1.1 使用 TiUP 升级 TiDB

本文档适用于使用 TiUP 从 TiDB 3.0 或 3.1 版本升级至 TiDB 4.0 版本，以及从 4.0 版本升级至后续版本。

如果原集群使用 TiDB Ansible 部署，TiUP 也支持将 TiDB Ansible 配置导入，参考 `tiup cluster import`，并完成升级。

6.1.1.1 1. 升级兼容性说明

- 不支持在升级后回退至 3.0 或更旧版本。
- 3.0 之前的版本，需要先通过 TiDB Ansible 升级到 3.0 版本，然后按照本文档的说明，使用 TiUP 将 TiDB Ansible 配置导入，再升级到 4.0 版本。
- TiDB Ansible 配置导入到 TiUP 中管理后，不能再通过 TiDB Ansible 对集群进行操作，否则可能因元信息不一致造成冲突。
- 对于满足以下情况之一的 TiDB Ansible 部署的集群，暂不支持导入：
  - 启用了 TLS 加密功能的集群
  - 纯 KV 集群（没有 TiDB 实例的集群）
  - 启用了 Kafka 的集群
  - 启用了 Spark 的集群
  - 启用了 TiDB Lightning / TiKV Importer 的集群

  - 仍使用老版本 'push' 的方式收集监控指标（从 3.0 默认为 'pull' 模式，如果没有特意调整过则可以支持）
  - 在 inventory.ini 配置文件中单独为机器的 node_exporter / blackbox_exporter 通过 node_exporter_port / blackbox_exporter_port 设置了非默认端口（在 group_vars 目录中统一配置的可以兼容）或者单独为某一台机器的 node_exporter / blackbox_exporter 设置了和其他机器的 node_exporter / blackbox_exporter 不同的 deploy_dir

- 如果使用 TiDB Ansible 部署的集群中有部分节点未部署监控，应当先使用 TiDB Ansible 在 inventory.ini 文件的 monitored_servers 分组中补充对应节点的信息，并通过 deploy.yaml playbook 将补充的监控组件部署完整。否则在数据导入 TiUP 后进行其他运维操作时，可能会因监控组件缺失而出错。
- 支持 TiDB Binlog，TiCDC，TiFlash 等组件版本的升级。
- 从 2.0.6 之前的版本升级到 4.0 版本之前，需要确认集群中是否存在正在运行中的 DDL 操作，特别是耗时的 Add Index 操作，等 DDL 操作完成后再执行升级操作
- 2.1 及之后版本启用了并行 DDL，早于 2.0.1 版本的集群，无法滚动升级到 4.0 版本，可以选择下面两种方案：
  - 停机升级，直接从早于 2.0.1 的 TiDB 版本升级到 4.0 版本，然后使用 TiUP Cluster 提供的 tiup cluster import 命令导入配置并使用 TiUP 作为管理工具。
  - 滚动升级，先滚动升级到 2.0.1 或者之后的 2.0.x 版本，再滚动升级到 4.0 版本。然后使用 TiUP Cluster 提供的 tiup cluster import 命令导入配置并使用 TiUP 作为管理工具。

> **注意：**
>
> 在升级的过程中不要执行 DDL 请求，否则可能会出现行为未定义的问题。

6.1.1.2 2. 在中控机器上安装 TiUP

1. 在中控机上执行如下命令安装 TiUP：

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

2. 重新声明全局环境变量：

```
source .bash_profile
```

3. 确认 TiUP 工具是否安装：

```
which tiup
```

4. 安装 TiUP 的 cluster 工具：

```
tiup cluster
```

如果之前安装过 TiUP，使用如下命令更新至最新版本即可：

```
tiup update cluster
```

> **注意：**
>
> 如果 `tiup --version` 显示 `tiup` 版本低于 `v1.0.0`，请在执行 `tiup update cluster` 之前先执行 `tiup update --self` 命令更新 `tiup` 版本。

6.1.1.3 3. 将 TiDB Ansible 及 inventory.ini 配置导入到 TiUP

> **注意：**
>
> - 目前 TiUP 仅支持 `systemd` 的进程管理模式。如果此前使用 TiDB Ansible 部署时选择了 `supervise`，需要先按使用 TiDB Ansible 部署 TiDB 集群迁移到 `systemd`。
> - 如果原集群已经是 TiUP 部署，可以跳过此步骤。
> - 目前默认识别 `inventory.ini` 配置文件，如果你的配置为其他名称，请指定。
> - 你需要确保当前集群的状态与 `inventory.ini` 中的拓扑一致，并确保集群的组件运行正常，否则导入后会导致集群元信息异常。
> - 如果在一个 TiDB Ansible 目录中管理多个不同的 `inventory.ini` 配置文件和 TiDB 集群，将其中一个集群导入到 TiUP 时，需要指定 `--no-backup` 以避免将 Ansible 目录移动到 TiUP 管理目录下面。

6.1.1.3.1 3.1 将 TiDB Ansible 集群导入到 TiUP 中

1. 以 `/home/tidb/tidb-ansible` 为示例路径，使用如下命令将 TiDB Ansible 集群导入到 TiUP 中：

    ```
    tiup cluster import -d /home/tidb/tidb-ansible
    ```

2. 执行导入命令后，若集群 `Inventory` 信息解析成功，将出现如下提示：

    ```
    tiup cluster import -d /home/tidb/tidb-ansible/
    ```

    ```
    Found inventory file /home/tidb/tidb-ansible/inventory.ini, parsing...
    Found cluster "ansible-cluster" (v3.0.12), deployed with user tidb.
    Prepared to import TiDB v3.0.12 cluster ansible-cluster.
    Do you want to continue? [y/N]:
    ```

3. 核对解析得到的集群名和版本无误后，输入y 确认继续执行。

    - 若 `Inventory` 信息解析出错，导入过程将会终止，终止不会对原 Ansible 部署方式有任何影响，之后需根据错误提示调整并重试。

- 若 Ansible 中原集群名与 TiUP 中任一已有集群的名称重复，将会给出警示信息并提示输入一个新的集群名。因此，请注意不要重复对同一个集群执行导入，导致 TiUP 中同一个集群有多个名字

导入完成后，可以通过 `tiup cluster display <cluster-name>` 查看当前集群状态以验证导入结果。由于 `display` 命令会查询各节点的实时状态，所以命令执行可能需要等待少许时间。

6.1.1.3.2 3.2 编辑 TiUP 拓扑配置文件

> **注意：**
>
> 以下情况可跳过该步骤：
>
> - 原集群没有修改过配置参数。
> - 升级后希望使用 4.0 默认参数。

1. 进入 TiDB Ansible 的备份目录 `~/.tiup/storage/cluster/clusters/{cluster_name}/ansible-imported-configs`，确认配置模板中修改过的参数。
2. 进入拓扑文件的 vi 编辑模式：

```
tiup cluster edit-config <cluster-name>
```

3. 参考 topology 配置模板的格式，将原集群修改过的参数填到拓扑文件的 `server_configs` 下面。

修改完成后 wq 保存并退出编辑模式，输入 Y 确认变更。

> **注意：**
>
> 升级到 4.0 版本前，请确认已在 3.0 修改的参数在 4.0 版本中是兼容的，可参考配置模板。
>
> TiUP 版本 <= v1.0.8 可能无法正确获取 TiFlash 的数据目录，需要确认 `data_dir` 与 TiFlash 配置的 `path` 值是否一致。若不一致需要进行如下操作把 TiFlash 的 `data_dir` 改成与 `path` 一致的值：
>
> 1. 执行 `tiup cluster edit-config <cluster-name>` 命令修改配置文件。
> 2. 修改对应 TiFlash 的 `data_dir` 配置：
>
> ```
> yaml    tiflash_servers:   - host: 10.0.1.14    data_dir: /data/tiflash-11315 #
> ↪ 修改为 TiFlash 配置文件的 `path` 值
> ```

6.1.1.4 4. 滚动升级 TiDB 集群

本部分介绍如何滚动升级 TiDB 集群以及升级后的验证。

6.1.1.4.1 4.1 将集群升级到指定版本

```
tiup cluster upgrade <cluster-name> <version>
```

以升级到 v4.0.16 版本为例：

```
tiup cluster upgrade <cluster-name> v4.0.16
```

滚动升级会逐个升级所有的组件。升级 TiKV 期间，会逐个将 TiKV 上的所有 leader 切走再停止该 TiKV 实例。默认超时时间为 5 分钟，超过后会直接停止实例。

如果不希望驱逐 leader，而希望立刻升级，可以在上述命令中指定 `--force`，该方式会造成性能抖动，不会造成数据损失。

如果希望保持性能稳定，则需要保证 TiKV 上的所有 leader 驱逐完成后再停止该 TiKV 实例，可以指定 `--transfer-timeout` 为一个超大值，如 `--transfer-timeout 100000000`，单位为 s。

6.1.1.4.2 4.2 升级后验证

执行 `display` 命令来查看最新的集群版本 `TiDB Version`：

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

```
Starting /home/tidblk/.tiup/components/cluster/v1.0.0/cluster display <cluster-name>
TiDB Cluster: <cluster-name>
TiDB Version: v4.0.16
```

> 注意：
>
> TiUP 及 TiDB（v4.0.2 起）默认会收集使用情况信息，并将这些信息分享给 PingCAP 用于改善产品。若要了解所收集的信息详情及如何禁用该行为，请参见遥测。

6.1.1.5 5. 升级 FAQ

本部分介绍使用 TiUP 升级 TiDB 集群遇到的常见问题。

6.1.1.5.1 5.1 升级时报错中断，处理完报错后，如何继续升级

重新执行 `tiup cluster upgrade` 命令进行升级，升级操作会重启之前已经升级完成的节点。TiDB 4.0 后续版本将支持从中断的位置继续升级。

6.1.1.5.2 5.2 升级过程中 evict leader 等待时间过长，如何跳过该步骤快速升级

可以指定 `--force`，升级时会跳过 `PD transfer leader` 和 `TiKV evict leader` 过程，直接重启并升级版本，对线上运行的集群影响较大。命令如下：

```
tiup cluster upgrade <cluster-name> v4.0.16 --force
```

6.1.1.5.3 5.3 升级完成后，如何更新 pd-ctl 等周边工具版本

目前 TiUP 没有对周边工具的版本进行管理更新，如需下载最新版本的工具包，直接下载 TiDB 安装包即可，将 `{version}` 替换为对应的版本如 `v4.0.16`，下载地址如下：

```
https://download.pingcap.org/tidb-{version}-linux-amd64.tar.gz
```

6.1.1.5.4 5.4 升级过程中 TiFlash 组件升级失败

TiFlash 在 `v4.0.0-rc.2` 之前的版本可能有一些不兼容的问题。因此，如果将包含 TiFlash 组件的集群由此前版本升级至 `v4.0.0-rc.2` 之后版本的过程中遇到问题，可在 ASK TUG 反馈，寻求研发人员支持。

6.1.1.6 6. TiDB 4.0 兼容性变化

- `oom-action` 参数设置为 `cancel` 时，当查询语句触发 OOM 阈值后会被 kill 掉，升级到 4.0 版本后除了 `select` 语句，还可能 kill 掉 `insert/update/delete` 等 DML 语句。
- 4.0 版本增加了 `rename` 时对表名长度的检查，长度限制为 64 个字符。升级后 `rename` 后的表名长度超过这个限制会报错，3.0 及之前的版本则不会报错。
- 4.0 版本增加了对分区表的分区名长度的检查，长度限制为 64 个字符。升级后，当你创建和修改分区表时，如果分区名长度超过这个限制会报错，3.0 及之前的版本则不会报错。
- 4.0 版本对 `explain` 执行计划的输出格式做了改进，需要注意是否有针对 `explain` 制订了自动化的分析程序。
- 4.0 版本支持Read Committed 隔离级别。升级到 4.0 后，在悲观事务里隔离级别设置为 `READ-COMMITTED` 会生效，3.0 及之前的版本则不会生效。
- 4.0 版本执行 `alter reorganize partition` 会报错，之前的版本则不会报错，只是语法上支持没有实际效果。
- 4.0 版本创建 `linear hash partition` 和 subpartition 分区表时实际不生效，会转换为普通表，之前的版本则转换为普通分区表。

6.1.2 使用 TiUP 离线镜像升级 TiDB

本文档适用于通过升级 TiUP 离线镜像升级 TiDB 集群，升级步骤如下。

6.1.2.1 1. 更新 TiUP 离线镜像

如果用户希望升级更新本地的 TiUP 离线镜像，可以参考使用 TiUP 离线部署 TiDB 集群的步骤 1 与步骤 2 下载部署新版本的 TiUP 离线镜像。在执行 `local_install.sh` 后，TiUP 会完成覆盖升级。

此时离线镜像已经更新成功。如果覆盖后发现 TiUP 运行报错，可能是 manifest 未更新导致，可尝试 `rm -rf ~/.tiup/manifests` 后再使用 TiUP。

6.1.2.2 2. 升级 TiDB 集群

在更新本地镜像后，可参考使用 TiUP 升级 TiDB升级 TiDB 集群。

> **注意：**
>
> TiUP 及 TiDB（v4.0.2 起）默认会收集使用情况信息，并将这些信息分享给 PingCAP 用于改善产品。若要了解所收集的信息详情及如何禁用该行为，请参见遥测。

6.1.3 使用 TiDB Operator

6.1.4 使用 TiDB Ansible 升级 TiDB

> **警告：**
>
> 对于生产环境，推荐使用 TiUP 升级 TiDB 集群。自 v4.0 版本起，PingCAP 不再提供 TiDB Ansible 的升级支持（废弃）。继续使用 TiDB Ansible 升级 TiDB 集群可能存在风险，因此不推荐该升级方式。

本文档适用于从 TiDB 2.0、2.1、3.0、3.1 版本升级至 TiDB 4.0 版本以及 TiDB 4.0 的低版本升级至 TiDB 4.0 高版本。目前，TiDB 4.0 版本兼容TiDB Binlog Cluster 版本。

6.1.4.1 升级兼容性说明

- 不支持在升级后回退至 3.1.x 或更旧版本
- 从 2.0.6 之前的版本升级到 4.0 之前，需要确认集群中是否存在正在运行中的 DDL 操作，特别是耗时的 Add Index 操作，等 DDL 操作完成后再执行升级操作
- 2.1 及之后版本启用了并行 DDL，早于 2.0.1 版本的集群，无法滚动升级到 4.0 版本，可以选择下面两种方案：
  - 停机升级，直接从早于 2.0.1 的 TiDB 版本升级到 4.0 版本
  - 先滚动升级到 2.0.1 或者之后的 2.0.x 版本，再滚动升级到 4.0 版本

> **注意：**
>
> 在升级的过程中不要执行 DDL 请求，否则可能会出现行为未定义的问题。

6.1.4.2 在中控机器上安装 TiDB Ansible 及其依赖

> **注意：**
>
> 如果已经安装了 TiDB Ansible 及其依赖，可跳过该步骤。

TiDB Ansible 最新开发版依赖 2.5.0 及以上但不高于 2.7.11 的 Ansible 版本（`2.5.0 ≦ ansible ≦ 2.7.11`，建议 2.7.11 版本），另依赖 Python 模块：`jinja2 ≧ 2.9.6` 和 `jmespath ≧ 0.9.0`。为方便管理依赖，建议使用 `pip` 安装 TiDB Ansible 及其依赖，可参照在中控机器上安装 TiDB Ansible 及其依赖 进行安装。离线环境参照在中控机器上离线安装 TiDB Ansible 及其依赖。

安装完成后，可通过以下命令查看版本：

```
ansible --version
```

```
ansible 2.7.11
```

```
pip show jinja2
```

```
Name: Jinja2
Version: 2.10
```

```
pip show jmespath
```

```
Name: jmespath
Version: 0.9.0
```

> **注意：**
>
> 请务必按以上文档安装 TiDB Ansible 及其依赖。确认 Jinja2 版本是否正确，否则启动 Grafana 时会报错。确认 jmespath 版本是否正确，否则滚动升级 TiKV 时会报错。

6.1.4.3 在中控机器上下载 TiDB Ansible

以 `tidb` 用户登录中控机并进入 `/home/tidb` 目录，备份 TiDB 2.0、2.1、3.0、3.1 或其他低版本的 tidb-ansible 文件夹：

```
mv tidb-ansible tidb-ansible-bak
```

下载 TiDB 4.0 版本对应的 TiDB Ansible，默认的文件夹名称为 `tidb-ansible`。`$tag` 需替换为选定的 TAG 版本的值，例如 `v4.0.0-rc`。

```
git clone -b $tag https://github.com/pingcap/tidb-ansible.git
```

6.1.4.4 编辑 inventory.ini 文件和配置文件

以 `tidb` 用户登录中控机并进入 `/home/tidb/tidb-ansible` 目录。

6.1.4.4.1 编辑 inventory.ini 文件

编辑 inventory.ini 文件，IP 信息参照备份文件 /home/tidb/tidb-ansible-bak/inventory.ini。

以下变量配置，需要重点确认，变量含义可参考inventory.ini 变量调整。

1. 请确认 ansible_user 配置的是普通用户。为统一权限管理，不再支持使用 root 用户远程安装。默认配置中使用 tidb 用户作为 SSH 远程用户及程序运行用户。

```
## Connection
# ssh via normal user
ansible_user = tidb
```

   可参考如何配置 SSH 互信及 sudo 规则自动配置主机间互信。

2. process_supervision 变量请与之前版本保持一致，默认推荐使用 systemd。

```
# process supervision, [systemd, supervise]
process_supervision = systemd
```

   如需变更，可参考如何调整进程监管方式从 supervise 到 systemd，先使用备份 /home/tidb/tidb-ansible
   ↪ -bak/ 分支变更进程监管方式再升级。

6.1.4.4.2 编辑 TiDB 集群组件配置文件

如之前自定义过 TiDB 集群组件配置文件，请参照备份文件修改 /home/tidb/tidb-ansible/conf 下对应配置文件。

注意以下参数变更：

- TiKV 配置中 end-point-concurrency 变更为 high-concurrency、normal-concurrency 和 low-concurrency 三个参数：

```
readpool:
  coprocessor:
    # Notice: if CPU_NUM > 8, default thread pool size for coprocessors
    # will be set to CPU_NUM * 0.8.
    # high-concurrency: 8
    # normal-concurrency: 8
    # low-concurrency: 8
```

> 注意：
> 2.0 版本升级且单机多 TiKV 实例（进程）情况下，需要修改这三个参数。
> 推荐设置：TiKV 实例数量 * 参数值 = CPU 核心数量 * 0.8

- TiKV 配置中不同 CF 中的 block-cache-size 参数变更为 block-cache：

```
storage:
  block-cache:
    capacity: "1GB"
```

> **注意：**
>
> 单机多 TiKV 实例（进程）情况下，需要修改 capacity 参数。如果当前版本已经是新的配置，则不需要再修改。
>
> 推荐设置：capacity = (MEM_TOTAL * 0.5 / TiKV 实例数量)

- TiKV 配置中单机多实例场景需要额外配置 tikv_status_port 端口：

```
[tikv_servers]
TiKV1-1 ansible_host=172.16.10.4 deploy_dir=/data1/deploy tikv_port=20171 tikv_status_port
    ↪ =20181 labels="host=tikv1"
TiKV1-2 ansible_host=172.16.10.4 deploy_dir=/data2/deploy tikv_port=20172 tikv_status_port
    ↪ =20182 labels="host=tikv1"
TiKV2-1 ansible_host=172.16.10.5 deploy_dir=/data1/deploy tikv_port=20171 tikv_status_port
    ↪ =20181 labels="host=tikv2"
TiKV2-2 ansible_host=172.16.10.5 deploy_dir=/data2/deploy tikv_port=20172 tikv_status_port
    ↪ =20182 labels="host=tikv2"
TiKV3-1 ansible_host=172.16.10.6 deploy_dir=/data1/deploy tikv_port=20171 tikv_status_port
    ↪ =20181 labels="host=tikv3"
TiKV3-2 ansible_host=172.16.10.6 deploy_dir=/data2/deploy tikv_port=20172 tikv_status_port
    ↪ =20182 labels="host=tikv3"
```

> **注意：**
>
> 从 3.0 以前版本（不包括 3.0）升级到 4.0 版本，并且单机多 TiKV 实例（进程）情况下，需要添加 tikv_status_port 参数。
>
> 配置前，注意检查端口是否有冲突。

6.1.4.5 下载 TiDB latest binary 到中控机

确认 tidb-ansible/inventory.ini 文件中 tidb_version = v4.0.x，然后执行以下命令下载 TiDB 4.0 binary 到中控机。

```
ansible-playbook local_prepare.yml
```

6.1.4.6 滚动升级 TiDB 集群组件

- 如果 process_supervision 变量使用默认的 systemd 参数：

- 当前集群版本 < 3.0，则通过 `excessive_rolling_update.yml` 滚动升级 TiDB 集群。

```
ansible-playbook excessive_rolling_update.yml
```

- 当前集群版本 ≥ 3.0.0，滚动升级及日常滚动重启 TiDB 集群，使用 `rolling_update.yml`。

```
ansible-playbook rolling_update.yml
```

- 如果 `process_supervision` 变量使用的是 `supervise` 参数，无论当前集群为哪个版本，均通过 `rolling_update.yml` 来滚动升级 TiDB 集群。

```
ansible-playbook rolling_update.yml
```

6.1.4.7 滚动升级 TiDB 监控组件

```
ansible-playbook rolling_update_monitor.yml
```

> **注意：**
>
> TiDB（v4.0.2 起）默认会定期收集使用情况信息，并将这些信息分享给 PingCAP 用于改善产品。若要了解所收集的信息详情及如何禁用该行为，请参见遥测。

6.2 扩缩容

6.2.1 使用 TiUP 扩容缩容 TiDB 集群

TiDB 集群可以在不中断线上服务的情况下进行扩容和缩容。

本文介绍如何使用 TiUP 扩容缩容集群中的 TiDB、TiKV、PD、TiCDC 或者 TiFlash 节点。如未安装 TiUP，可参考升级文档中的步骤，将集群 Import 到 TiUP 环境中，再使用 TiUP 进行扩容缩容。

你可以通过 `tiup cluster list` 查看当前的集群名称列表。

例如，集群原拓扑结构如下所示：

主机 IP	服务
10.0.1.3	TiDB + TiFlash
10.0.1.4	TiDB + PD
10.0.1.5	TiKV + Monitor
10.0.1.1	TiKV
10.0.1.2	TiKV

6.2.1.1 扩容 TiDB/PD/TiKV 节点

如果要添加一个 TiDB 节点，IP 地址为 10.0.1.5，可以按照如下步骤进行操作。

> **注意：**
>
> 添加 PD 节点和添加 TiDB 节点的步骤类似。添加 TiKV 节点前，建议预先根据集群的负载情况调整 PD 调度参数。

6.2.1.1.1 1. 编写扩容拓扑配置

> **注意：**
>
> - 默认情况下，可以不填写端口以及目录信息。但在单机多实例场景下，则需要分配不同的端口以及目录，如果有端口或目录冲突，会在部署或扩容时提醒。
> - 从 TiUP v1.0.0 开始，扩容配置会继承原集群配置的 global 部分。

在 scale-out.yaml 文件添加扩容拓扑配置：

```
vi scale-out.yaml
```

```
tidb_servers:
  - host: 10.0.1.5
    ssh_port: 22
    port: 4000
    status_port: 10080
    deploy_dir: /data/deploy/install/deploy/tidb-4000
    log_dir: /data/deploy/install/log/tidb-4000
```

TiKV 配置文件参考：

```
tikv_servers:
  - host: 10.0.1.5
    ssh_port: 22
    port: 20160
    status_port: 20180
    deploy_dir: /data/deploy/install/deploy/tikv-20160
    data_dir: /data/deploy/install/data/tikv-20160
    log_dir: /data/deploy/install/log/tikv-20160
```

PD 配置文件参考：

```
pd_servers:
```

```
  - host: 10.0.1.5
    ssh_port: 22
    name: pd-1
    client_port: 2379
    peer_port: 2380
    deploy_dir: /data/deploy/install/deploy/pd-2379
    data_dir: /data/deploy/install/data/pd-2379
    log_dir: /data/deploy/install/log/pd-2379
```

可以使用 `tiup cluster edit-config <cluster-name>` 查看当前集群的配置信息，因为其中的 `global` 和 `server_configs` 参数配置默认会被 `scale-out.yaml` 继承，因此也会在 `scale-out.yaml` 中生效。

6.2.1.1.2 2. 执行扩容命令

```
tiup cluster scale-out <cluster-name> scale-out.yaml
```

> **注意：**
>
> 此处假设当前执行命令的用户和新增的机器打通了互信，如果不满足已打通互信的条件，需要通过 `-p` 来输入新机器的密码，或通过 `-i` 指定私钥文件。

预期输出 Scaled cluster `<cluster-name>` out successfully 信息，表示扩容操作成功。

6.2.1.1.3 3. 检查集群状态

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

打开浏览器访问监控平台 http://10.0.1.5:3000，监控整个集群和新增节点的状态。

扩容后，集群拓扑结构如下所示：

主机 IP	服务
10.0.1.3	TiDB + TiFlash
10.0.1.4	TiDB + PD
10.0.1.5	TiDB + TiKV + Monitor
10.0.1.1	TiKV
10.0.1.2	TiKV

6.2.1.2 扩容 TiFlash 节点

如果要添加一个 TiFlash 节点，其 IP 地址为 `10.0.1.4`，可以按照如下步骤进行操作。

> **注意：**
>
> 在原有 TiDB 集群上新增 TiFlash 组件需要注意：
>
> 1. 首先确认当前 TiDB 的版本支持 TiFlash，否则需要先升级 TiDB 集群至 4.0 RC 以上版本。
> 2. 执行 `tiup ctl:<cluster-version> pd -u http://<pd_ip>:<pd_port> config set enable-placement-rules true` 命令，以开启 PD 的 Placement Rules 功能。或通过pd-ctl 执行对应的命令。

6.2.1.2.1 1. 添加节点信息到 scale-out.yaml 文件

编写 scale-out.yaml 文件，添加该 TiFlash 节点信息（目前只支持 ip，不支持域名）：

```
tiflash_servers:
  - host: 10.0.1.4
```

6.2.1.2.2 2. 运行扩容命令

```
tiup cluster scale-out <cluster-name> scale-out.yaml
```

> **注意：**
>
> 此处假设当前执行命令的用户和新增的机器打通了互信，如果不满足已打通互信的条件，需要通过 `-p` 来输入新机器的密码，或通过 `-i` 指定私钥文件。

6.2.1.2.3 3. 查看集群状态

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

打开浏览器访问监控平台 http://10.0.1.5:3000，监控整个集群和新增节点的状态。

扩容后，集群拓扑结构如下所示：

主机 IP	服务
10.0.1.3	TiDB + TiFlash
10.0.1.4	TiDB + PD + TiFlash
10.0.1.5	TiDB+ TiKV + Monitor
10.0.1.1	TiKV
10.0.1.2	TiKV

6.2.1.3 扩容 TiCDC 节点

如果要添加 TiCDC 节点，IP 地址为 10.0.1.3、10.0.1.4，可以按照如下步骤进行操作。

6.2.1.3.1 1. 添加节点信息到 scale-out.yaml 文件

编写 scale-out.yaml 文件：

```
cdc_servers:
  - host: 10.0.1.3
    gc-ttl: 86400
    data_dir: /data/deploy/install/data/cdc-8300
  - host: 10.0.1.4
    gc-ttl: 86400
    data_dir: /data/deploy/install/data/cdc-8300
```

6.2.1.3.2 2. 运行扩容命令

```
tiup cluster scale-out <cluster-name> scale-out.yaml
```

> 注意：
>
> 此处假设当前执行命令的用户和新增的机器打通了互信，如果不满足已打通互信的条件，需要通过 -p 来输入新机器的密码，或通过 -i 指定私钥文件。

6.2.1.3.3 3. 查看集群状态

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

打开浏览器访问监控平台 http://10.0.1.5:3000，监控整个集群和新增节点的状态。

扩容后，集群拓扑结构如下所示：

主机 IP	服务
10.0.1.3	TiDB + TiFlash + TiCDC
10.0.1.4	TiDB + PD + TiFlash + TiCDC
10.0.1.5	TiDB+ TiKV + Monitor
10.0.1.1	TiKV
10.0.1.2	TiKV

6.2.1.4 缩容 TiDB/PD/TiKV 节点

如果要移除 IP 地址为 10.0.1.5 的一个 TiKV 节点，可以按照如下步骤进行操作。

注意：

- 移除 TiDB、 PD 节点和移除 TiKV 节点的步骤类似。
- 由于 TiKV、 TiFlash 和 TiDB Binlog 组件是异步下线的，且下线过程耗时较长，所以 TiUP 对 TiKV、 TiFlash 和 TiDB Binlog 组件做了特殊处理，详情参考下线特殊处理。

注意：

TiKV 中的 PD Client 会缓存 PD 节点的列表。

- 在 v4.0.3 版本之前，TiKV 不会定期自动更新该 PD 节点列表的缓存，只有在 PD leader 发生切换或 TiKV 重启加载最新配置后才会更新。为避免 TiKV 缓存的 PD 节点列表过旧的风险，在扩缩容 PD 完成后，PD 集群中应至少包含一个扩缩容操作前就已经存在的 PD 节点成员。如果不满足该条件，你需要手动执行 PD transfer leader 操作以更新 TiKV 中的 PD 缓存列表。
- 在 v4.0.3 及以后的版本中，增加了定期自动更新 PD 节点的机制，可以降低 TiKV 缓存的 PD 节点列表过旧这一问题出现的概率。但仍应尽量避免在扩容新 PD 后直接一次性缩容所有扩容前就已经存在的 PD 节点。如果需要，请确保在下线所有之前存在的 PD 节点前将 PD 的 leader 切换至新扩容的 PD 节点。

6.2.1.4.1 1. 查看节点 ID 信息

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

```
Starting /root/.tiup/components/cluster/v0.4.6/cluster display <cluster-name>

TiDB Cluster: <cluster-name>

TiDB Version: v4.0.16

ID              Role         Host        Ports                           Status  Data Dir
    ↪           Deploy Dir

--              ----         ----        -----                           ------  --------
    ↪           ----------

10.0.1.3:8300   cdc          10.0.1.3    8300                            Up      data/cdc-8300
    ↪           deploy/cdc-8300

10.0.1.4:8300   cdc          10.0.1.4    8300                            Up      data/cdc-8300
    ↪           deploy/cdc-8300
```

```
10.0.1.4:2379   pd           10.0.1.4    2379/2380                   Healthy data/pd-2379
    ↪           deploy/pd-2379

10.0.1.1:20160  tikv         10.0.1.1    20160/20180                 Up      data/tikv-20160
    ↪           deploy/tikv-20160

10.0.1.2:20160  tikv         10.0.1.2    20160/20180                 Up      data/tikv-20160
    ↪           deploy/tikv-20160

10.0.1.5:20160  tikv         10.0.1.5    20160/20180                 Up      data/tikv-20160
    ↪           deploy/tikv-20160

10.0.1.3:4000   tidb         10.0.1.3    4000/10080                  Up      -
    ↪            deploy/tidb-4000

10.0.1.4:4000   tidb         10.0.1.4    4000/10080                  Up      -
    ↪            deploy/tidb-4000

10.0.1.5:4000   tidb         10.0.1.5    4000/10080                  Up      -
    ↪           deploy/tidb-4000

10.0.1.3:9000   tiflash      10.0.1.3    9000/8123/3930/20170/20292/8234  Up      data/tiflash
    ↪ -9000        deploy/tiflash-9000

10.0.1.4:9000   tiflash      10.0.1.4    9000/8123/3930/20170/20292/8234  Up      data/tiflash
    ↪ -9000        deploy/tiflash-9000

10.0.1.5:9090   prometheus   10.0.1.5    9090                        Up      data/prometheus
    ↪ -9090     deploy/prometheus-9090

10.0.1.5:3000   grafana      10.0.1.5    3000                        Up      -
    ↪           deploy/grafana-3000

10.0.1.5:9093   alertmanager 10.0.1.5    9093/9094                   Up      data/
    ↪ alertmanager-9093  deploy/alertmanager-9093
```

6.2.1.4.2 2. 执行缩容操作

```
tiup cluster scale-in <cluster-name> --node 10.0.1.5:20160
```

其中 --node 参数为需要下线节点的 ID。

预期输出 Scaled cluster <cluster-name> in successfully 信息，表示缩容操作成功。

6.2.1.4.3 3. 检查集群状态

下线需要一定时间，下线节点的状态变为 Tombstone 就说明下线成功。

执行如下命令检查节点是否下线成功：

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

打开浏览器访问监控平台 http://10.0.1.5:3000，监控整个集群的状态。

调整后，拓扑结构如下：

Host IP	Service
10.0.1.3	TiDB + TiFlash + TiCDC
10.0.1.4	TiDB + PD + TiFlash + TiCDC
10.0.1.5	TiDB + Monitor（TiKV 已删除）
10.0.1.1	TiKV
10.0.1.2	TiKV

6.2.1.5 缩容 TiFlash 节点

如果要缩容 IP 地址为 10.0.1.4 的一个 TiFlash 节点，可以按照如下步骤进行操作。

6.2.1.5.1 1. 根据 TiFlash 剩余节点数调整数据表的副本数

在下线节点之前，确保 TiFlash 集群剩余节点数大于等于所有数据表的最大副本数，否则需要修改相关表的 TiFlash 副本数。

1. 在 TiDB 客户端中针对所有副本数大于集群剩余 TiFlash 节点数的表执行：

```
alter table <db-name>.<table-name> set tiflash replica 0;
```

2. 等待相关表的 TiFlash 副本被删除（按照查看表同步进度一节操作，查不到相关表的同步信息时即为副本被删除）。

6.2.1.5.2 2. 执行缩容操作

接下来，请任选下列方案其一进行缩容。

方案一：通过 TiUP 缩容 TiFlash 节点

1. 通过以下命令确定需要下线的节点名称：

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

2. 执行 scale-in 命令来下线节点，假设步骤 1 中获得该节点名为 10.0.1.4:9000

```
tiup cluster scale-in <cluster-name> --node 10.0.1.4:9000
```

方案二：手动缩容 TiFlash 节点

在特殊情况下（比如需要强制下线节点），或者 TiUP 操作失败的情况下，可以使用以下方法手动下线 TiFlash 节点。

1. 使用 pd-ctl 的 store 命令在 PD 中查看该 TiFlash 节点对应的 store id。

    - 在pd-ctl (tidb-ansible 目录下的 `resources/bin` 包含对应的二进制文件) 中输入 store 命令。
    - 若使用 TiUP 部署，可以调用以下命令代替 `pd-ctl`：

    ```
    tiup ctl:<cluster-version> pd -u http://<pd_ip>:<pd_port> store
    ```

    > 注意：
    > 如果集群中有多个 PD 实例，只需在以上命令中指定一个活跃 PD 实例的 IP: 端口即可。

2. 在 pd-ctl 中下线该 TiFlash 节点。

    - 在 pd-ctl 中输入 `store delete <store_id>`，其中 `<store_id>` 为上一步查到的该 TiFlash 节点对应的 store id。
    - 若通过 TiUP 部署，可以调用以下命令代替 `pd-ctl`：

    ```
    tiup ctl:<cluster-version> pd -u http://<pd_ip>:<pd_port> store delete <store_id>
    ```

    > 注意：
    > 如果集群中有多个 PD 实例，只需在以上命令中指定一个活跃 PD 实例的 IP: 端口即可。

3. 等待该 TiFlash 节点对应的 store 消失或者 state_name 变成 Tombstone 再关闭 TiFlash 进程。

    如果等待较长时间后，该节点仍然无法正常消失或者状态变成 Tombstone，可以考虑以下命令，把节点强制踢出集群（慎用）：

    > 警告：
    >
    > - 以下命令会直接丢弃该 TiFlash 节点上的副本，有可能导致查询失败。
    > - 一旦使用后请勿再使用 PD Control 将该节点从 PD 元信息中删除，否则会导致集群 Region 信息不一致。
    > - 该命令仅可对 TiFlash 节点使用。请勿使用该命令强制踢出任何 TiKV 节点，否则会导致数据丢失。
    > - 使用该命令后，被踢出节点上的副本信息仍然会存在于其他副本的成员信息里，PD 会发起调度来修复清除被踢出节点的副本信息。因此使用命令前，需考虑该节点丢失后，是否有足够的副本数选举出 leader，否则无法执行调度进行修复清除，导致服务不可用。

    ```
    curl -X POST 'http://<pd-address>/pd/api/v1/store/<store_id>/state?state=Tombstone'
    ```

4. 手动删除 TiFlash 的数据文件，具体位置可查看在集群拓扑配置文件中 TiFlash 配置部分下的 data_dir 目录。
5. 手动更新 TiUP 的集群配置文件，在编辑模式中手动删除我们已经下线的 TiFlash 节点信息：

```
tiup cluster edit-config <cluster-name>
```

> **注意：**
>
> 如果在集群中所有的 TiFlash 节点停止运行之前，没有取消所有同步到 TiFlash 的表，则需要手动在 PD 中清除同步规则，否则无法成功完成 TiFlash 节点的下线。

手动在 PD 中清除同步规则的步骤如下：

1. 查询当前 PD 实例中所有与 TiFlash 相关的的数据同步规则。

```
curl http://<pd_ip>:<pd_port>/pd/api/v1/config/rules/group/tiflash
```

```
[
  {
    "group_id": "tiflash",
    "id": "table-45-r",
    "override": true,
    "start_key": "7480000000000000FF2D5F720000000000FA",
    "end_key": "7480000000000000FF2E00000000000000F8",
    "role": "learner",
    "count": 1,
    "label_constraints": [
      {
        "key": "engine",
        "op": "in",
        "values": [
          "tiflash"
        ]
      }
    ]
  }
]
```

2. 删除所有与 TiFlash 相关的数据同步规则。以 `id` 为 `table-45-r` 的规则为例，通过以下命令可以删除该规则。

```
curl -v -X DELETE http://<pd_ip>:<pd_port>/pd/api/v1/config/rule/tiflash/table-45-r
```

6.2.1.5.3 3. 查看集群状态

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

打开浏览器访问监控平台 http://10.0.1.5:3000，监控整个集群的状态。

调整后，拓扑结构如下：

Host IP	Service
10.0.1.3	TiDB + TiFlash + TiCDC
10.0.1.4	TiDB + PD + TiCDC （ TiFlash 已删除 ）
10.0.1.5	TiDB + Monitor
10.0.1.1	TiKV
10.0.1.2	TiKV

6.2.1.6 缩容 TiCDC 节点

如果要缩容 IP 地址为 10.0.1.4 的一个 TiCDC 节点，可以按照如下步骤进行操作。

6.2.1.6.1 1. 下线该 TiCDC 节点

```
tiup cluster scale-in <cluster-name> --node 10.0.1.4:8300
```

6.2.1.6.2 2. 查看集群状态

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

打开浏览器访问监控平台 http://10.0.1.5:3000，监控整个集群的状态。

调整后，拓扑结构如下：

Host IP	Service
10.0.1.3	TiDB + TiFlash + TiCDC
10.0.1.4	TiDB + PD + (TiCDC 已删除)
10.0.1.5	TiDB + Monitor
10.0.1.1	TiKV
10.0.1.2	TiKV

6.2.2 使用 TiDB Ansible 扩容缩容 TiDB 集群

TiDB 集群可以在不影响线上服务的情况下进行扩容和缩容。

警告：

- 对于生产环境，推荐使用 TiUP 进行集群扩缩容操作。自 v4.0 版本起，PingCAP 不再提供 TiDB Ansible 的扩缩容支持（废弃）。继续使用 TiDB Ansible 对 TiDB 集群进行扩缩容操作可能存在风险，因此不推荐该方式。
- 以下缩容示例中，被移除的节点没有混合部署其他服务；如果混合部署了其他服务，不能按如下操作。

假设拓扑结构如下所示：

Name	Host IP	Services
node1	172.16.10.1	PD1
node2	172.16.10.2	PD2
node3	172.16.10.3	PD3, Monitor
node4	172.16.10.4	TiDB1
node5	172.16.10.5	TiDB2
node6	172.16.10.6	TiKV1
node7	172.16.10.7	TiKV2
node8	172.16.10.8	TiKV3
node9	172.16.10.9	TiKV4

6.2.2.1 扩容 TiDB/TiKV 节点

例如，如果要添加两个 TiDB 节点（node101、node102），IP 地址为 172.16.10.101、172.16.10.102，可以进行如下操作：

1. 编辑 `inventory.ini` 文件和 `hosts.ini` 文件，添加节点信息。

    - 编辑 `inventory.ini`：

```
[tidb_servers]
172.16.10.4
172.16.10.5
172.16.10.101
172.16.10.102

[pd_servers]
172.16.10.1
172.16.10.2
172.16.10.3

[tikv_servers]
172.16.10.6
172.16.10.7
172.16.10.8
```

```
172.16.10.9

[monitored_servers]
172.16.10.1
172.16.10.2
172.16.10.3
172.16.10.4
172.16.10.5
172.16.10.6
172.16.10.7
172.16.10.8
172.16.10.9
172.16.10.101
172.16.10.102

[monitoring_servers]
172.16.10.3

[grafana_servers]
172.16.10.3
```

现在拓扑结构如下所示：

Name	Host IP	Services
node1	172.16.10.1	PD1
node2	172.16.10.2	PD2
node3	172.16.10.3	PD3, Monitor
node4	172.16.10.4	TiDB1
node5	172.16.10.5	TiDB2
node101	172.16.10.101	TiDB3
node102	172.16.10.102	TiDB4
node6	172.16.10.6	TiKV1
node7	172.16.10.7	TiKV2
node8	172.16.10.8	TiKV3
node9	172.16.10.9	TiKV4

- 编辑 `hosts.ini`：

```
[servers]
172.16.10.1
172.16.10.2
172.16.10.3
172.16.10.4
172.16.10.5
172.16.10.6
```

```
172.16.10.7
172.16.10.8
172.16.10.9
172.16.10.101
172.16.10.102

[all:vars]
username = tidb
ntp_server = pool.ntp.org
```

2. 初始化新增节点。

    1. 在中控机上配置部署机器 SSH 互信及 sudo 规则：

    ```
    ansible-playbook -i hosts.ini create_users.yml -l 172.16.10.101,172.16.10.102 -u root -
        ↪ k
    ```

    2. 在部署目标机器上安装 NTP 服务：

    ```
    ansible-playbook -i hosts.ini deploy_ntp.yml -u tidb -b
    ```

    3. 在部署目标机器上初始化节点：

    ```
    ansible-playbook bootstrap.yml -l 172.16.10.101,172.16.10.102
    ```

> 注意：
> 如果 `inventory.ini` 中为节点配置了别名，如 `node101 ansible_host=172.16.10.101`，执行 ansible-playbook 时 `-l` 请指定别名，以下步骤类似。例如：`ansible-playbook bootstrap.yml -l node101,node102`

3. 部署新增节点：

```
ansible-playbook deploy.yml -l 172.16.10.101,172.16.10.102
```

4. 启动新节点服务：

```
ansible-playbook start.yml -l 172.16.10.101,172.16.10.102
```

5. 更新 Prometheus 配置并重启：

```
ansible-playbook rolling_update_monitor.yml --tags=prometheus
```

6. 打开浏览器访问监控平台：`http://172.16.10.3:3000`，监控整个集群和新增节点的状态。

可使用同样的步骤添加 TiKV 节点。但如果要添加 PD 节点，则需手动更新一些配置文件。

6.2.2.2 扩容 PD 节点

例如，如果要添加一个 PD 节点（node103），IP 地址为 172.16.10.103，可以进行如下操作：

1. 编辑 inventory.ini 文件，添加节点信息置于 [pd_servers] 主机组最后一行：

```
[tidb_servers]
172.16.10.4
172.16.10.5

[pd_servers]
172.16.10.1
172.16.10.2
172.16.10.3
172.16.10.103

[tikv_servers]
172.16.10.6
172.16.10.7
172.16.10.8
172.16.10.9

[monitored_servers]
172.16.10.4
172.16.10.5
172.16.10.1
172.16.10.2
172.16.10.3
172.16.10.103
172.16.10.6
172.16.10.7
172.16.10.8
172.16.10.9

[monitoring_servers]
172.16.10.3

[grafana_servers]
172.16.10.3
```

现在拓扑结构如下所示：

Name	Host IP	Services
node1	172.16.10.1	PD1
node2	172.16.10.2	PD2
node3	172.16.10.3	PD3, Monitor

Name	Host IP	Services
node103	172.16.10.103	PD4
node4	172.16.10.4	TiDB1
node5	172.16.10.5	TiDB2
node6	172.16.10.6	TiKV1
node7	172.16.10.7	TiKV2
node8	172.16.10.8	TiKV3
node9	172.16.10.9	TiKV4

2. 初始化新增节点：

```
ansible-playbook bootstrap.yml -l 172.16.10.103
```

3. 部署新增节点：

```
ansible-playbook deploy.yml -l 172.16.10.103
```

4. 登录新增的 PD 节点，编辑启动脚本：`{deploy_dir}/scripts/run_pd.sh`

    1. 移除 `--initial-cluster="xxxx" \` 配置，注意这里不能在行开头加注释符 #。
    2. 添加 `--join="http://172.16.10.1:2379" \`，IP 地址（172.16.10.1）可以是集群内现有 PD IP 地址中的任意一个。
    3. 在新增 PD 节点中启动 PD 服务：

    ```
    {deploy_dir}/scripts/start_pd.sh
    ```

> 注意：
> 启动前，需要通过PD Control 工具确认当前 PD 节点的 health 状态均为 “true”，否则 PD 服务会启动失败，同时日志中报错 `["join meet error"] [error="etcdserver: unhealthy cluster"]`。

    4. 使用 pd-ctl 检查新节点是否添加成功：

    ```
    /home/tidb/tidb-ansible/resources/bin/pd-ctl -u "http://172.16.10.1:2379" -d member
    ```

5. 启动监控服务：

```
ansible-playbook start.yml -l 172.16.10.103
```

> 注意：
> 如果使用了别名（inventory_name），则也需要使用 `-l` 指定别名。

6. 更新集群的配置：

```
ansible-playbook deploy.yml
```

7. 重启 Prometheus，新增扩容的 PD 节点的监控：

```
ansible-playbook stop.yml --tags=prometheus
ansible-playbook start.yml --tags=prometheus
```

8. 打开浏览器访问监控平台：`http://172.16.10.3:3000`，监控整个集群和新增节点的状态。

> **注意：**
>
> TiKV 中的 PD Client 会缓存 PD 节点列表，但目前不会定期自动更新，只有在 PD leader 发生切换或 TiKV 重启加载最新配置后才会更新；为避免 TiKV 缓存的 PD 节点列表过旧的风险，在扩缩容 PD 完成后，PD 集群中至少要包含两个扩缩容操作前就已经存在的 PD 节点成员，如果不满足该条件需要手动执行 PD transfer leader 操作，更新 TiKV 中的 PD 缓存列表。

6.2.2.3 缩容 TiDB 节点

例如，如果要移除一个 TiDB 节点（node5），IP 地址为 172.16.10.5，可以进行如下操作：

1. 停止 node5 节点上的服务：

```
ansible-playbook stop.yml -l 172.16.10.5
```

2. 编辑 `inventory.ini` 文件，移除节点信息：

```
[tidb_servers]
172.16.10.4
#172.16.10.5  # 注释被移除节点

[pd_servers]
172.16.10.1
172.16.10.2
172.16.10.3

[tikv_servers]
172.16.10.6
172.16.10.7
172.16.10.8
172.16.10.9

[monitored_servers]
172.16.10.4
#172.16.10.5  # 注释被移除节点
```

```
172.16.10.1
172.16.10.2
172.16.10.3
172.16.10.6
172.16.10.7
172.16.10.8
172.16.10.9

[monitoring_servers]
172.16.10.3

[grafana_servers]
172.16.10.3
```

现在拓扑结构如下所示：

Name	Host IP	Services
node1	172.16.10.1	PD1
node2	172.16.10.2	PD2
node3	172.16.10.3	PD3, Monitor
node4	172.16.10.4	TiDB1
node5	172.16.10.5	TiDB2 已删除
node6	172.16.10.6	TiKV1
node7	172.16.10.7	TiKV2
node8	172.16.10.8	TiKV3
node9	172.16.10.9	TiKV4

3. 更新 Prometheus 配置并重启：

```
ansible-playbook rolling_update_monitor.yml --tags=prometheus
```

4. 打开浏览器访问监控平台：`http://172.16.10.3:3000`，监控整个集群的状态。

6.2.2.4 缩容 TiKV 节点

例如，如果要移除一个 TiKV 节点（node9），IP 地址为 172.16.10.9，可以进行如下操作：

1. 使用 `pd-ctl` 从集群中移除节点：

    1. 查看 node9 节点的 store id：

    ```
    /home/tidb/tidb-ansible/resources/bin/pd-ctl -u "http://172.16.10.1:2379" -d store
    ```

    2. 从集群中移除 node9，假如 store id 为 10：

    ```
    /home/tidb/tidb-ansible/resources/bin/pd-ctl -u "http://172.16.10.1:2379" -d store
        ↪ delete 10
    ```

2. 使用 Grafana 或者 pd-ctl 检查节点是否下线成功（下线需要一定时间，下线节点的状态变为 Tombstone 就说明下线成功了）：

```
/home/tidb/tidb-ansible/resources/bin/pd-ctl -u "http://172.16.10.1:2379" -d store 10
```

3. 下线成功后，停止 node9 上的服务：

```
ansible-playbook stop.yml -l 172.16.10.9
```

4. 编辑 inventory.ini 文件，移除节点信息：

```
[tidb_servers]
172.16.10.4
172.16.10.5

[pd_servers]
172.16.10.1
172.16.10.2
172.16.10.3

[tikv_servers]
172.16.10.6
172.16.10.7
172.16.10.8
#172.16.10.9  # 注释被移除节点

[monitored_servers]
172.16.10.4
172.16.10.5
172.16.10.1
172.16.10.2
172.16.10.3
172.16.10.6
172.16.10.7
172.16.10.8
#172.16.10.9  # 注释被移除节点

[monitoring_servers]
172.16.10.3

[grafana_servers]
172.16.10.3
```

现在拓扑结构如下所示：

Name	Host IP	Services
node1	172.16.10.1	PD1

Name	Host IP	Services
node2	172.16.10.2	PD2
node3	172.16.10.3	PD3, Monitor
node4	172.16.10.4	TiDB1
node5	172.16.10.5	TiDB2
node6	172.16.10.6	TiKV1
node7	172.16.10.7	TiKV2
node8	172.16.10.8	TiKV3
node9	172.16.10.9	TiKV4 已删除

5. 更新 Prometheus 配置并重启：

```
ansible-playbook rolling_update_monitor.yml --tags=prometheus
```

6. 打开浏览器访问监控平台：http://172.16.10.3:3000，监控整个集群的状态。

6.2.2.5 缩容 PD 节点

例如，如果要移除一个 PD 节点（node2），IP 地址为 172.16.10.2，可以进行如下操作：

1. 使用 pd-ctl 从集群中移除节点：

    1. 查看 node2 节点的 name：

    ```
    /home/tidb/tidb-ansible/resources/bin/pd-ctl -u "http://172.16.10.1:2379" -d member
    ```

    2. 从集群中移除 node2，假如 name 为 pd2：

    ```
    /home/tidb/tidb-ansible/resources/bin/pd-ctl -u "http://172.16.10.1:2379" -d member
        ↪ delete name pd2
    ```

2. 使用 pd-ctl 检查节点是否下线成功（PD 下线会很快，结果中没有 node2 节点信息即为下线成功）：

```
/home/tidb/tidb-ansible/resources/bin/pd-ctl -u "http://172.16.10.1:2379" -d member
```

3. 下线成功后，停止 node2 上的服务：

```
ansible-playbook stop.yml -l 172.16.10.2
```

> 注意：
> 此示例中 172.16.10.2 服务器上只有 PD 节点才可以如上操作，如果该服务器上还有其他服务（例如 TiDB），则还需要使用 -t 来指定服务（例如 -t tidb）。

4. 编辑 inventory.ini 文件，移除节点信息：

```
[tidb_servers]
172.16.10.4
172.16.10.5

[pd_servers]
172.16.10.1
#172.16.10.2  # 注释被移除节点
172.16.10.3

[tikv_servers]
172.16.10.6
172.16.10.7
172.16.10.8
172.16.10.9

[monitored_servers]
172.16.10.4
172.16.10.5
172.16.10.1
#172.16.10.2  # 注释被移除节点
172.16.10.3
172.16.10.6
172.16.10.7
172.16.10.8
172.16.10.9

[monitoring_servers]
172.16.10.3

[grafana_servers]
172.16.10.3
```

现在拓扑结构如下所示：

Name	Host IP	Services
node1	172.16.10.1	PD1
node2	172.16.10.2	PD2 已删除
node3	172.16.10.3	PD3, Monitor
node4	172.16.10.4	TiDB1
node5	172.16.10.5	TiDB2
node6	172.16.10.6	TiKV1
node7	172.16.10.7	TiKV2
node8	172.16.10.8	TiKV3
node9	172.16.10.9	TiKV4

5. 更新集群的配置：

```
ansible-playbook deploy.yml
```

6. 重启 Prometheus，移除缩容的 PD 节点的监控：

```
ansible-playbook stop.yml --tags=prometheus
ansible-playbook start.yml --tags=prometheus
```

7. 打开浏览器访问监控平台：`http://172.16.10.3:3000`，监控整个集群的状态。

> **注意：**
>
> TiKV 中的 PD Client 会缓存 PD 节点列表，但目前不会定期自动更新，只有在 PD leader 发生切换或 TiKV 重启加载最新配置后才会更新；为避免 TiKV 缓存的 PD 节点列表过旧的风险，在扩缩容 PD 完成后，PD 集群中至少要包含两个扩缩容操作前就已经存在的 PD 节点成员，如果不满足该条件需要手动执行 PD transfer leader 操作，更新 TiKV 中的 PD 缓存列表。

6.2.3 使用 TiDB Operator

6.3 备份与恢复

6.3.1 使用 BR 工具（推荐）

6.3.1.1 备份与恢复工具 BR 简介

BR 全称为 Backup & Restore，是 TiDB 分布式备份恢复的命令行工具，用于对 TiDB 集群进行数据备份和恢复。BR 只支持在 TiDB v3.1 及以上版本使用。

相比Dumpling，BR 更适合大数据量的场景。

本文介绍了 BR 的工作原理、推荐部署配置、使用限制以及几种使用方式。

6.3.1.1.1 工作原理

BR 将备份或恢复操作命令下发到各个 TiKV 节点。TiKV 收到命令后执行相应的备份或恢复操作。

在一次备份或恢复中，各个 TiKV 节点都会有一个对应的备份路径，TiKV 备份时产生的备份文件将会保存在该路径下，恢复时也会从该路径读取相应的备份文件。

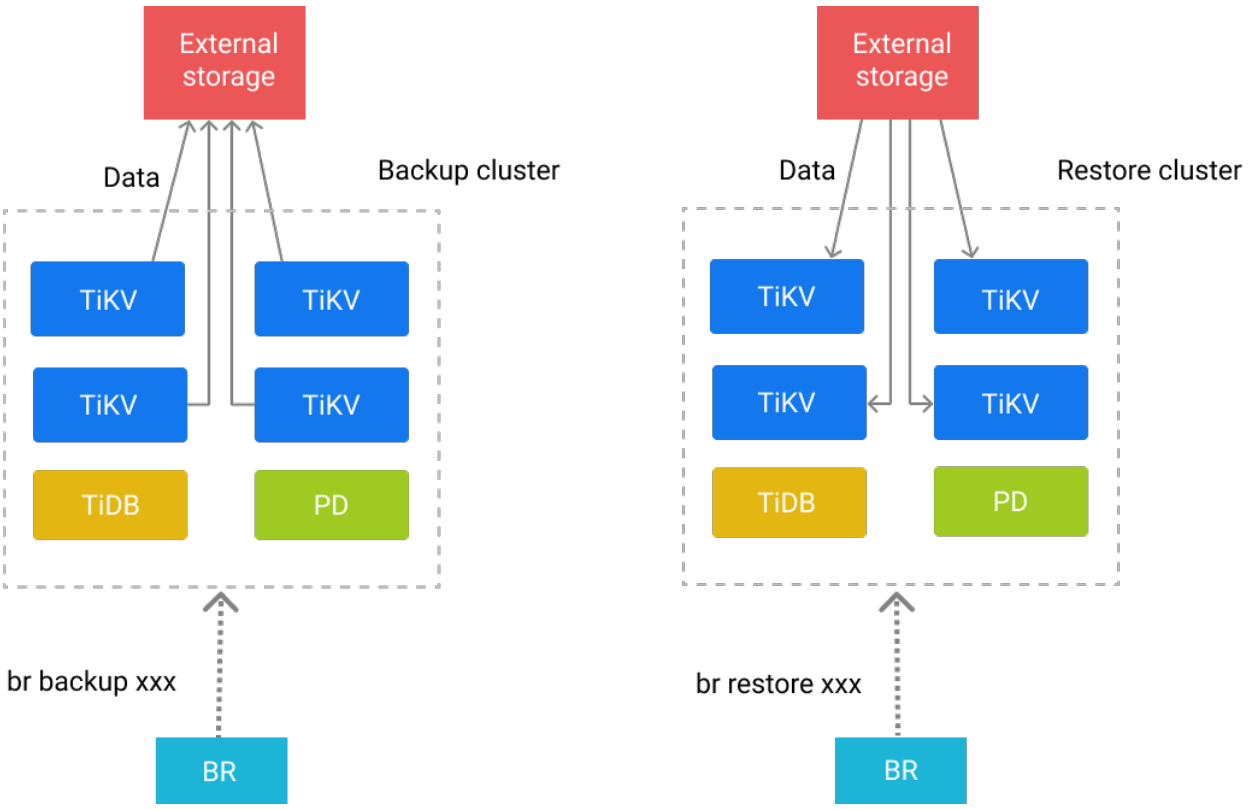


图 27: br-arch

更多信息请参阅备份恢复设计方案。

备份文件类型

备份路径下会生成以下两种类型文件：

- SST 文件：存储 TiKV 备份下来的数据信息
- `backupmeta` 文件：存储本次备份的元信息，包括备份文件数、备份文件的 Key 区间、备份文件大小和备份文件 Hash (sha256) 值
- `backup.lock` 文件：用于防止多次备份到同一目录

SST 文件命名格式

SST 文件以 `storeID_regionID_regionEpoch_keyHash_cf` 的格式命名。格式名的解释如下：

- storeID：TiKV 节点编号
- regionID：Region 编号
- regionEpoch：Region 版本号
- keyHash：Range startKey 的 Hash (sha256) 值，确保唯一性
- cf：RocksDB 的 ColumnFamily（默认为 `default` 或 `write`）

6.3.1.1.2 部署使用 BR 工具

推荐部署配置

- 推荐 BR 部署在 PD 节点上。
- 推荐使用一块高性能 SSD 网盘，挂载到 BR 节点和所有 TiKV 节点上，网盘推荐万兆网卡，否则带宽有可能成为备份恢复时的性能瓶颈。

> 注意：
>
> - 如果没有挂载网盘或者使用其他共享存储，那么 BR 备份的数据会生成在各个 TiKV 节点上。由于 BR 只备份 leader 副本，所以各个节点预留的空间需要根据 leader size 来预估。
> - 同时由于 v4.0 默认使用 leader count 进行平衡，所以会出现 leader size 差别大的问题，导致各个节点备份数据不均衡。

使用限制

下面是使用 BR 进行备份恢复的几条限制：

- BR 恢复到 TiCDC / Drainer 的上游集群时，恢复数据无法由 TiCDC / Drainer 同步到下游。
- BR 只支持在 new_collations_enabled_on_first_bootstrap 开关值相同的集群之间进行操作。这是因为 BR 仅备份 KV 数据。如果备份集群和恢复集群采用不同的排序规则，数据校验会不通过。所以恢复集群时，你需要确保 `select VARIABLE_VALUE from mysql.tidb where VARIABLE_NAME='new_collation_enabled'` 语句的开关值查询结果与备份时的查询结果相一致，才可以进行恢复。

兼容性

BR 和 TiDB 集群的兼容性问题分为以下两方面：

- BR 部分版本和 TiDB 集群的接口不兼容
- 某些功能在开启或关闭状态下，会导致 KV 格式发生变化，因此备份和恢复期间如果没有统一开启或关闭，就会带来不兼容的问题

下表整理了会导致 KV 格式发生变化的功能。

功能	相关 issue	解决方式
New collation	#352	确保恢复时集群的 new_collations_enabled_on_first_bootstrap ↪ 变量值和备份时的一致，否则会导致数据索引不一致和 checksum 通不过。
恢复集群开启 TiCDC 同步	#364	TiKV 暂不能将 BR ingest 的 SST 文件下推到 TiCDC，因此使用 BR 恢复时候需要关闭 TiCDC。

在上述功能确保备份恢复一致的前提下，BR 和 TiKV/TiDB/PD 还可能因为版本内部协议不一致/接口不一致出现不兼容的问题，因此 BR 内置了版本检查。

版本检查

BR 内置版本会在执行备份和恢复操作前，对 TiDB 集群版本和自身版本进行对比检查。如果大版本不匹配（比如 BR v4.x 和 TiDB v5.x 上），BR 会提示退出。如要跳过版本检查，可以通过设置 `--check-requirements=false` 强行跳过版本检查，但是这样可能会引入版本不兼容的问题。TiDB v4.0 用 BR 备份后，不完全支持恢复到 v5.0 以及之后版本，详细信息见 BR 版本检查（stable 版文档）。

运行 BR 的最低机型配置要求

运行 BR 的最低机型配置要求如下：

CPU	内存	硬盘类型	网络
1 核	4 GB	HDD	千兆网卡

一般场景下（备份恢复的表少于 1000 张），BR 在运行期间的 CPU 消耗不会超过 200%，内存消耗不会超过 1 GB。但在备份和恢复大量数据表时，BR 的内存消耗可能会上升到 3 GB 以上。在实际测试中，备份 24000 张表大概需要消耗 2.7 GB 内存，CPU 消耗维持在 100% 以下。

最佳实践

下面是使用 BR 进行备份恢复的几种推荐操作：

- 推荐在业务低峰时执行备份操作，这样能最大程度地减少对业务的影响。
- BR 支持在不同拓扑的集群上执行恢复，但恢复期间对在线业务影响很大，建议低峰期或者限速 (`rate-limit`) 执行恢复。
- BR 备份最好串行执行。不同备份任务并行会导致备份性能降低，同时也会影响在线业务。
- BR 恢复最好串行执行。不同恢复任务并行会导致 Region 冲突增多，恢复的性能降低。
- 推荐在 `-s` 指定的备份路径上挂载一个共享存储，例如 NFS。这样能方便收集和管理备份文件。
- 在使用共享存储时，推荐使用高吞吐的存储硬件，因为存储的吞吐会限制备份或恢复的速度。

使用方式

目前支持以下几种方式来运行 BR 工具，分别是通过 SQL 语句、命令行工具或在 Kubernetes 环境下进行备份恢复。

通过 SQL 语句

在 v4.0.2 及以上版本的 TiDB 中，支持直接通过 SQL 语句进行备份恢复，具体使用示例见：

- Backup 语法
- Restore 语法

通过命令行工具

在 v3.1 以上的 TiDB 版本中，支持通过命令行工具进行备份恢复。

首先需要下载一个 BR 工具的二进制包，详见下载链接。

通过命令行工具进行备份恢复的具体操作见使用备份与恢复工具 BR。

在 Kubernetes 环境下

目前支持使用 BR 工具备份 TiDB 集群数据到兼容 S3 的存储、 Google Cloud Storage 以及持久卷，并作恢复：

> **注意：**
>
> Amazon S3 和 Google Cloud Storage (GCS) 参数描述见外部存储文档。

- 备份 TiDB 集群数据到兼容 S3 的存储

- 恢复 S3 兼容存储上的备份数据
- 备份 TiDB 集群到 Google Cloud Storage
- 恢复 Google Cloud Storage 上的备份数据
- 备份 TiDB 集群到持久卷
- 恢复持久卷上的备份数据

6.3.1.1.3 BR 相关文档

- 使用 BR 命令行备份恢复
- BR 备份与恢复场景示例
- BR 常见问题
- 外部存储

6.3.1.2 使用 BR 命令行进行备份恢复

本文介绍如何 BR 命令行进行 TiDB 集群数据的备份和恢复。

在阅读本文前，请确保你已通读备份与恢复工具 BR 简介，尤其是使用限制和最佳实践这两节。

6.3.1.2.1 BR 命令行描述

一条 br 命令是由子命令、选项和参数组成的。子命令即不带 - 或者 -- 的字符。选项即以 - 或者 -- 开头的字符。参数即子命令或选项字符后紧跟的、并传递给命令和选项的字符。

以下是一条完整的 br 命令行：

```
br backup full --pd "${PDIP}:2379" -s "local:///tmp/backup"
```

命令行各部分的解释如下：

- backup：br 的子命令
- full：backup 的子命令
- -s 或 --storage：备份保存的路径
- "local:///tmp/backup"：-s 的参数，保存的路径为各个 TiKV 节点本地磁盘的 /tmp/backup
- --pd：PD 服务地址
- "${PDIP}:2379"：--pd 的参数

> 注意：
>
> 在使用 local storage 的时候，备份数据会分散在各个节点的本地文件系统中。
>
> 不建议在生产环境中备份到本地磁盘，因为在日后恢复的时候，必须手动聚集这些数据才能完成恢复工作（见恢复集群数据）。
>
> 聚集这些备份数据可能会造成数据冗余和运维上的麻烦，而且在不聚集这些数据便直接恢复的时候会遇到颇为迷惑的 SST file not found 报错。
>
> 建议在各个节点挂载 NFS 网盘，或者直接备份到 S3 对象存储中。

命令和子命令

BR 由多层命令组成。目前，BR 包含 backup、 restore 和 version 三个子命令:

- br backup 用于备份 TiDB 集群
- br restore 用于恢复 TiDB 集群

以上三个子命令可能还包含这些子命令：

- full：可用于备份或恢复全部数据。
- db：可用于备份或恢复集群中的指定数据库。
- table：可用于备份或恢复集群指定数据库中的单张表。

常用选项

- --pd：用于连接的选项，表示 PD 服务地址，例如 "${PDIP}:2379"。
- -h/--help：获取所有命令和子命令的使用帮助。例如 br backup --help。
- -V (或 --version)：检查 BR 版本。
- --ca：指定 PEM 格式的受信任 CA 的证书文件路径。
- --cert：指定 PEM 格式的 SSL 证书文件路径。
- --key：指定 PEM 格式的 SSL 证书密钥文件路径。
- --status-addr：BR 向 Prometheus 提供统计数据的监听地址。

6.3.1.2.2 使用 BR 命令行备份集群数据

使用 br backup 命令来备份集群数据。可选择添加 full 或 table 子命令来指定备份的范围：全部集群数据或单张表的数据。

如果 BR 的版本低于 v4.0.8，而且备份时间可能超过设定的tikv_gc_life_time（默认 10m0s，即表示 10 分钟），需要手动将该参数调大。

例如，将 tikv_gc_life_time 调整为 720h：

```
mysql -h${TiDBIP} -P4000 -u${TIDB_USER} ${password_str} -Nse \
    "update mysql.tidb set variable_value='720h' where variable_name='tikv_gc_life_time'";
```

自 v4.0.8 起 BR 已经支持自适应 GC，无需手动调整 tikv_gc_life_time。

备份全部集群数据

要备份全部集群数据，可使用 br backup full 命令。该命令的使用帮助可以通过 br backup full -h 或 br backup full --help 来获取。

用例：将所有集群数据备份到各个 TiKV 节点的 /tmp/backup 路径，同时也会将备份的元信息文件 backupmeta 写到该路径下。

> 注意：

- 经测试，在全速备份的情况下，如果备份盘和服务盘不同，在线备份会让只读线上服务的 QPS 下降 15%~25% 左右。如果希望降低影响，请参考 `--ratelimit` 进行限速。
- 假如备份盘和服务盘相同，备份将会和服务争夺 I/O 资源，这可能会让只读线上服务的 QPS 骤降一半以上。请尽量禁止将在线服务的数据备份到 TiKV 的数据盘。

```
br backup full \
    --pd "${PDIP}:2379" \
    --storage "local:///tmp/backup" \
    --ratelimit 128 \
    --log-file backupfull.log
```

以上命令中，`--ratelimit` 选项限制了每个 TiKV 执行备份任务的速度上限（单位 MiB/s）。`--log-file` 选项指定把 BR 的 log 写到 `backupfull.log` 文件中。

备份期间有进度条在终端中显示。当进度条前进到 100% 时，说明备份已完成。在完成备份后，BR 为了确保数据安全性，还会校验备份数据。进度条效果如下：

```
br backup full \
    --pd "${PDIP}:2379" \
    --storage "local:///tmp/backup" \
    --ratelimit 128 \
    --log-file backupfull.log
Full Backup <---------/.....................................................> 17.12%.
```

备份单个数据库的数据

要备份集群中指定单个数据库的数据，可使用 `br backup db` 命令。同样可通过 `br backup db -h` 或 `br backup db --help` 来获取子命令 db 的使用帮助。

用例：将数据库 `test` 备份到各个 TiKV 节点的 `/tmp/backup` 路径，同时也会将备份的元信息文件 `backupmeta` 写到该路径下。

```
br backup db \
    --pd "${PDIP}:2379" \
    --db test \
    --storage "local:///tmp/backup" \
    --ratelimit 128 \
    --log-file backuptable.log
```

db 子命令的选项为 `--db`，用来指定数据库名。其他选项的含义与备份全部集群数据相同。

备份期间有进度条在终端中显示。当进度条前进到 100% 时，说明备份已完成。在完成备份后，BR 为了确保数据安全性，还会校验备份数据。

备份单张表的数据

要备份集群中指定单张表的数据，可使用 `br backup table` 命令。同样可通过 `br backup table -h` 或 `br backup table --help` 来获取子命令 `table` 的使用帮助。

用例：将表 test.usertable 备份到各个 TiKV 节点的 /tmp/backup 路径，同时也会将备份的元信息文件 backupmeta 写到该路径下。

```
br backup table \
    --pd "${PDIP}:2379" \
    --db test \
    --table usertable \
    --storage "local:///tmp/backup" \
    --ratelimit 128 \
    --log-file backuptable.log
```

table 子命令有 --db 和 --table 两个选项，分别用来指定数据库名和表名。其他选项的含义与备份全部集群数据相同。

备份期间有进度条在终端中显示。当进度条前进到 100% 时，说明备份已完成。在完成备份后，BR 为了确保数据安全性，还会校验备份数据。

使用表库过滤功能备份多张表的数据

如果你需要以更复杂的过滤条件来备份多个表，执行 br backup full 命令，并使用 --filter 或 -f 来指定表库过滤规则。

用例：以下命令将所有 db*.tbl* 形式的表格数据备份到每个 TiKV 节点上的 /tmp/backup 路径，并将 backupmeta 文件写入该路径。

```
br backup full \
    --pd "${PDIP}:2379" \
    --filter 'db*.tbl*' \
    --storage "local:///tmp/backup" \
    --ratelimit 128 \
    --log-file backupfull.log
```

备份数据到 Amazon S3 后端存储

如果备份的存储并不是在本地，而是在 Amazon 的 S3 后端存储，那么需要在 storage 子命令中指定 S3 的存储路径，并且赋予 BR 节点和 TiKV 节点访问 Amazon S3 的权限。

这里可以参照 AWS 官方文档在指定的 Region 区域中创建一个 S3 桶 Bucket，如果有需要，还可以参照 AWS 官方文档 在 Bucket 中创建一个文件夹 Folder。

> 注意：
>
> 要完成一次备份，通常 TiKV 和 BR 需要的最小权限为 s3:ListBucket，s3:PutObject 和 s3:
> ↪ AbortMultipartUpload。

将有权限访问该 S3 后端存储的账号的 SecretKey 和 AccessKey 作为环境变量传入 BR 节点，并且通过 BR 将权限传给 TiKV 节点。

```
export AWS_ACCESS_KEY_ID=${AccessKey}
export AWS_SECRET_ACCESS_KEY=${SecretKey}
```

在进行 BR 备份时，显示指定参数 `--s3.region` 和 `--send-credentials-to-tikv`, `--s3.region` 表示 S3 存储所在的区域，`--send-credentials-to-tikv` 表示将 S3 的访问权限传递给 TiKV 节点。

```
br backup full \
    --pd "${PDIP}:2379" \
    --storage "s3://${Bucket}/${Folder}" \
    --s3.region "${region}" \
    --send-credentials-to-tikv=true \
    --ratelimit 128 \
    --log-file backuptable.log
```

增量备份

如果想要备份增量，只需要在备份的时候指定上一次的备份时间戳 `--lastbackupts` 即可。

注意增量备份有以下限制：

- 增量备份需要与前一次全量备份在不同的路径下
- GC safepoint 必须在 `lastbackupts` 之前

```
br backup full\
    --pd ${PDIP}:2379 \
    --ratelimit 128 \
    -s local:///home/tidb/backupdata/incr \
    --lastbackupts ${LAST_BACKUP_TS}
```

以上命令会备份 `(LAST_BACKUP_TS, current PD timestamp]` 之间的增量数据。

你可以使用 `validate` 指令获取上一次备份的时间戳，示例如下：

```
LAST_BACKUP_TS=`br validate decode --field="end-version" -s local:///home/tidb/backupdata | tail
    ↪ -n1`
```

示例备份的增量数据记录 `(LAST_BACKUP_TS, current PD timestamp]` 之间的数据变更，以及这段时间内的 DDL。在恢复的时候，BR 会先把所有 DDL 恢复，而后才会恢复数据。

Raw KV 备份（实验性功能）

> **警告：**
>
> Raw KV 备份功能还在实验中，没有经过完备的测试。暂时请避免在生产环境中使用该功能。

在某些使用场景下，TiKV 可能会独立于 TiDB 运行。考虑到这点，BR 也提供跳过 TiDB 层，直接备份 TiKV 中数据的功能：

```
br backup raw --pd $PD_ADDR \
    -s "local://$BACKUP_DIR" \
    --start 31 \
    --ratelimit 128 \
    --end 3130303030303030 \
    --format hex \
    --cf default
```

以上命令会备份 default CF 上 [0x31, 0x3130303030303030) 之间的所有键到 $BACKUP_DIR 去。

这里，--start 和 --end 的参数会先依照 --format 指定的方式解码，再被送到 TiKV 上去，目前支持以下解码方式：

- “raw”：不进行任何操作，将输入的字符串直接编码为二进制格式的键。
- “hex”：将输入的字符串视作十六进制数字。这是默认的编码方式。
- “escape”：对输入的字符串进行转义之后，再编码为二进制格式。

6.3.1.2.3 使用 BR 命令行恢复集群数据

使用 br restore 命令来恢复备份数据。可选择添加 full、db 或 table 子命令来指定恢复操作的范围：全部集群数据、某个数据库或某张数据表。

> 注意：
>
> 如果使用本地存储，在恢复前必须将所有备份的 SST 文件复制到各个 TiKV 节点上 --storage 指定的目录下。
>
> 即使每个 TiKV 节点最后只需要读取部分 SST 文件，这些节点也需要有所有 SST 文件的完全访问权限。原因如下：
>
> - 数据被复制到了多个 Peer 中。在读取 SST 文件时，这些文件必须要存在于所有 Peer 中。这与数据的备份不同，在备份时，只需从单个节点读取。
> - 在数据恢复的时候，每个 Peer 分布的位置是随机的，事先并不知道哪个节点将读取哪个文件。
>
> 使用共享存储可以避免这些情况。例如，在本地路径上安装 NFS，或使用 S3。利用这些网络存储，各个节点都可以自动读取每个 SST 文件，此时上述注意事项不再适用。

恢复全部备份数据

要将全部备份数据恢复到集群中来，可使用 br restore full 命令。该命令的使用帮助可以通过 br restore ↪ full -h 或 br restore full --help 来获取。

用例：将 /tmp/backup 路径中的全部备份数据恢复到集群中。

```
br restore full \
```

```
    --pd "${PDIP}:2379" \
    --storage "local:///tmp/backup" \
    --ratelimit 128 \
    --log-file restorefull.log
```

以上命令中，`--ratelimit` 选项限制了每个 TiKV 执行恢复任务的速度上限（单位 MiB/s）。`--log-file` 选项指定把 BR 的 log 写到 `restorefull.log` 文件中。

恢复期间还有进度条会在终端中显示，当进度条前进到 100% 时，说明恢复已完成。在完成恢复后，BR 为了确保数据安全性，还会校验恢复数据。进度条效果如下：

```
br restore full \
    --pd "${PDIP}:2379" \
    --storage "local:///tmp/backup" \
    --ratelimit 128 \
    --log-file restorefull.log
Full Restore <---------/.............................................> 17.12%.
```

恢复单个数据库的数据

要将备份数据中的某个数据库恢复到集群中，可以使用 `br restore db` 命令。该命令的使用帮助可以通过 `br restore db -h` 或 `br restore db --help` 来获取。

用例：将 `/tmp/backup` 路径中备份数据中的某个数据库恢复到集群中。

```
br restore db \
    --pd "${PDIP}:2379" \
    --db "test" \
    --ratelimit 128 \
    --storage "local:///tmp/backup" \
    --log-file restorefull.log
```

以上命令中 `--db` 选项指定了需要恢复的数据库名字。其余选项的含义与恢复全部备份数据相同。

> **注意：**
>
> 恢复备份数据的时候，`--db` 选项指定的数据库名必须与执行备份时候 `--db`选项指定的数据库名相同，否则无法恢复成功。由于备份数据的元文件`backupmeta` 记录了该数据库名，因此只能将数据恢复到同名的数据库。推荐做法是把备份文件恢复到另一个集群的同名数据库中。

恢复单张表的数据

要将备份数据中的某张数据表恢复到集群中，可以使用 `br restore table` 命令。该命令的使用帮助可通过 `br restore table -h` 或 `br restore table --help` 来获取。

用例：将 `/tmp/backup` 路径下的备份数据中的某个数据表恢复到集群中。

```
br restore table \
    --pd "${PDIP}:2379" \
    --db "test" \
    --table "usertable" \
    --ratelimit 128 \
    --storage "local:///tmp/backup" \
    --log-file restorefull.log
```

使用表库功能过滤恢复数据

如果你需要用复杂的过滤条件来恢复多个表，执行 `br restore full` 命令，并用 `--filter` 或 `-f` 指定使用表库过滤。

用例：以下命令将备份在 `/tmp/backup` 路径的表的子集恢复到集群中。

```
br restore full \
    --pd "${PDIP}:2379" \
    --filter 'db*.tbl*' \
    --storage "local:///tmp/backup" \
    --log-file restorefull.log
```

从 Amazon S3 后端存储恢复数据

如果需要恢复的数据并不是存储在本地，而是在 Amazon 的 S3 后端，那么需要在 `storage` 子命令中指定 S3 的存储路径，并且赋予 BR 节点和 TiKV 节点访问 Amazon S3 的权限。

> **注意：**
>
> 要完成一次恢复，通常 TiKV 和 BR 需要的最小权限为 `s3:ListBucket` 和 `s3:GetObject`。

将有权限访问该 S3 后端存储的账号的 `SecretKey` 和 `AccessKey` 作为环境变量传入 BR 节点，并且通过 BR 将权限传给 TiKV 节点。

```
export AWS_ACCESS_KEY_ID=${AccessKey}
export AWS_SECRET_ACCESS_KEY=${SecretKey}
```

在进行 BR 恢复时，显示指定参数 `--s3.region` 和 `--send-credentials-to-tikv`, `--s3.region` 表示 S3 存储所在的区域，`--send-credentials-to-tikv` 表示将 S3 的访问权限传递给 TiKV 节点。`--storage`参数中的 Bucket 和 Folder 分别代表需要恢复的数据所在的 S3 存储桶和文件夹。

```
br restore full \
    --pd "${PDIP}:2379" \
    --storage "s3://${Bucket}/${Folder}" \
    --s3.region "${region}" \
    --ratelimit 128 \
    --send-credentials-to-tikv=true \
```

```
    --log-file restorefull.log
```

以上命令中 `--table` 选项指定了需要恢复的表名。其余选项的含义与恢复单个数据库相同。

增量恢复

增量恢复的方法和使用 BR 进行全量恢复的方法并无差别。需要注意，恢复增量数据的时候，需要保证备份时指定的 `last backup ts` 之前备份的数据已经全部恢复到目标集群。

恢复创建在 `mysql` 数据库下的表（实验性功能）

BR 可以并且会默认备份 `mysql` 数据库下的表。

在执行恢复时，`mysql` 下的表默认不会被恢复。如果需要恢复 `mysql` 下的用户创建的表，可以通过table filter 来显式地包含目标表。以下示例中要恢复目标用户表为 `mysql.usertable`；该命令会在执行正常的恢复的同时恢复 `mysql.usertable`。

```
br restore full -f '*.*' -f '!mysql.*' -f 'mysql.usertable' -s $external_storage_url --ratelimit
    ↪ 128
```

在如上的命令中，`-f '*.*'` 用于覆盖掉默认的规则，`-f '!mysql.*'` 指示 BR 不要恢复 `mysql` 中的表，除非另有指定。`-f 'mysql.usertable'` 则指定需要恢复 `mysql.usertable`。具体原理请参考table filter 的文档。

如果只需要恢复 `mysql.usertable`，而无需恢复其他表，可以使用以下命令：

```
br restore full -f 'mysql.usertable' -s $external_storage_url --ratelimit 128
```

> 警告：
>
> 虽然系统表（例如 `mysql.tidb` 等）可以通过 BR 进行备份和恢复，但是部分系统表在恢复之后可能会出现非预期的状况，已知的异常如下：
>
> - 统计信息表（`mysql.stat_*`）无法被恢复。
> - 系统变量表（`mysql.tidb`，`mysql.global_variables`）无法被恢复。
> - 用户信息表（`mysql.user`，`mysql.columns_priv`，等等）无法被恢复。
> - GC 数据无法被恢复。
>
> 恢复系统表可能还存在更多兼容性问题。为了防止意外发生，请避免在生产环境中恢复系统表。

Raw KV 恢复（实验性功能）

> 警告：
>
> Raw KV 恢复功能还在实验中，没有经过完备的测试。暂时请避免在生产环境中使用该功能。

和Raw KV 备份相似地，恢复 Raw KV 的命令如下：

```
br restore raw --pd $PD_ADDR \
    -s "local://$BACKUP_DIR" \
    --start 31 \
    --end 3130303030303030 \
    --ratelimit 128 \
    --format hex \
    --cf default
```

以上命令会将范围在 [0x31, 0x3130303030303030) 的已备份键恢复到 TiKV 集群中。这里键的编码方式和备份时相同。

在线恢复（实验性功能）

> **警告：**
>
> 在线恢复功能还在实验中，没有经过完备的测试，同时还依赖 PD 的不稳定特性 Placement Rules。暂时请避免在生产环境中使用该功能。

在恢复的时候，写入过多的数据会影响在线集群的性能。为了尽量避免影响线上业务，BR 支持通过Placement rules 隔离资源。让下载、导入 SST 的工作仅仅在指定的几个节点（下称“恢复节点”）上进行，具体操作如下：

1. 配置 PD，启动 Placement rules：

   ```
   echo "config set enable-placement-rules true" | pd-ctl
   ```

2. 编辑恢复节点 TiKV 的配置文件，在 server 一项中指定：

   ```
   [server]
   labels = { exclusive = "restore" }
   ```

3. 启动恢复节点的 TiKV，使用 BR 恢复备份的文件，和非在线恢复相比，这里只需要加上 --online 标志即可：

   ```
   br restore full \
       -s "local://$BACKUP_DIR" \
       --ratelimit 128 \
       --pd $PD_ADDR \
       --online
   ```

6.3.1.3 BR 备份与恢复场景示例

BR 是一款分布式的快速备份和恢复工具。

本文展示了以下几种备份和恢复场景下的 BR 操作过程：

- 将单表数据备份到网络盘（推荐生产环境使用）
- 从网络磁盘恢复备份数据（推荐生产环境使用）
- 将单表数据备份到本地磁盘（推荐测试环境试用）
- 从本地磁盘恢复备份数据（推荐测试环境试用）

以帮助读者达到以下目标：

- 正确使用网络盘或本地盘进行备份或恢复
- 通过相关监控指标了解备份或恢复的状态
- 了解在备份或恢复时如何调优性能
- 处理备份时可能发生的异常

6.3.1.3.1 目标读者

你需要对 TiDB 和 TiKV 有一定的了解。

在阅读本文前，请确保你已通读备份与恢复工具 BR 简介，尤其是使用限制和最佳实践这两节。

6.3.1.3.2 环境准备

本节介绍 TiDB 的推荐部署方式、BR 使用示例中的集群版本、TiKV 集群硬件信息和集群配置。

你可以根据自己的硬件和配置来预估备份恢复的性能。

部署方式

推荐使用TiUP 部署 TiDB 集群，再下载TiDB Toolkit 获取 BR 工具。

集群版本

- TiDB: v4.0.2
- TiKV: v4.0.2
- PD: v4.0.2
- BR: v4.0.2

> **注意：**
>
> v4.0.2 为编写本文档时的最新版本。推荐读者使用最新版本 TiDB/TiKV/PD/BR，同时需要确保 BR 版本和 TiDB 相同。

TiKV 集群硬件信息

- 操作系统：CentOS Linux release 7.6.1810 (Core)
- CPU：16-Core Common KVM processor
- RAM：32GB
- 硬盘：500G SSD * 2

- 网卡：万兆网卡

配置

BR 可以直接将命令下发到 TiKV 集群来执行备份和恢复，不依赖 TiDB server 组件，因此无需对 TiDB server 进行配置。

- TiKV: 默认配置
- PD : 默认配置

6.3.1.3.3 使用场景

本节描述以下几种使用场景：

- 将单表数据备份到网络盘（推荐生产环境使用）
- 从网络磁盘恢复备份数据（推荐生产环境使用）
- 将单表数据备份到本地磁盘（推荐测试环境试用）
- 从本地磁盘恢复备份数据（推荐测试环境试用）

推荐使用网络盘来进行备份和恢复操作，这样可以省去收集备份数据文件的繁琐步骤。尤其在 TiKV 集群规模较大的情况下，使用网络盘可以大幅提升操作效率。

在使用 BR 进行备份或恢复操作前，需要先进行如下准备工作.

备份前的准备工作

如果你使用的是 TiDB v4.0.8 及以上版本，相应版本的 BR 工具已支持自适应 GC，会自动将 backupTS（默认是最新的 PD timestamp）注册到 PD 的 `safePoint`，保证 TiDB 的 GC Safe Point 在备份期间不会向前移动，即可避免手动设置 GC。

关于 `br backup` 命令的具体使用方法，参见使用备份与恢复工具 BR。

如果你使用的是 TiDB v4.0.7 及以下版本，则需要在 BR 备份前后，按照以下步骤手动设置 GC：

1. 运行`br backup` 命令前，查询 TiDB 集群的`tikv_gc_life_time` 配置项的值，并使用 MySQL 客户端将该项调整至合适的值，确保备份期间不会发生Garbage Collection (GC)。

```
SELECT * FROM mysql.tidb WHERE VARIABLE_NAME = 'tikv_gc_life_time';
UPDATE mysql.tidb SET VARIABLE_VALUE = '720h' WHERE VARIABLE_NAME = 'tikv_gc_life_time';
```

2. 在备份完成后，将该参数调回原来的值。

```
UPDATE mysql.tidb SET VARIABLE_VALUE = '10m' WHERE VARIABLE_NAME = 'tikv_gc_life_time';
```

恢复前的准备工作

使用 BR 进行恢复前的准备工作如下：

运行`br restore` 命令前，需要检查新集群，确保集群内没有同名的表。

将单表数据备份到网络盘（推荐生产环境使用）

使用 `br backup` 命令，将单表数据 `--db batchmark --table order_line` 备份到指定的网络盘路径 `local:///br_data` 下。

前置要求

- 备份前的准备工作。
- 配置一台高性能 SSD 硬盘主机为 NFS server 存储数据。其他所有 BR 节点、TiKV 节点和 TiFlash 节点为 NFS client，挂载相同的路径（例如 `/br_data`）到 NFS server 上以访问 NFS server。
- NFS server 和 NFS client 间的数据传输速率至少要达到备份集群的 `TiKV 实例数 * 150MB/s`。否则网络 I/O 有可能成为性能瓶颈。

> 注意：
>
> - 因为备份时候只备份单副本 (leader) 数据，所以即使集群中存在 TiFlash 副本，无需挂载 TiFlash 节点 BR 也能完成备份。
> - BR 在恢复数据时，会恢复全部副本的数据。因此在恢复时，TiFlash 节点需要有备份数据的访问权限 BR 才能完成恢复，此时也必须将 TiFlash 节点挂载到 NFS server 上。

部署拓扑

部署拓扑如下图所示：

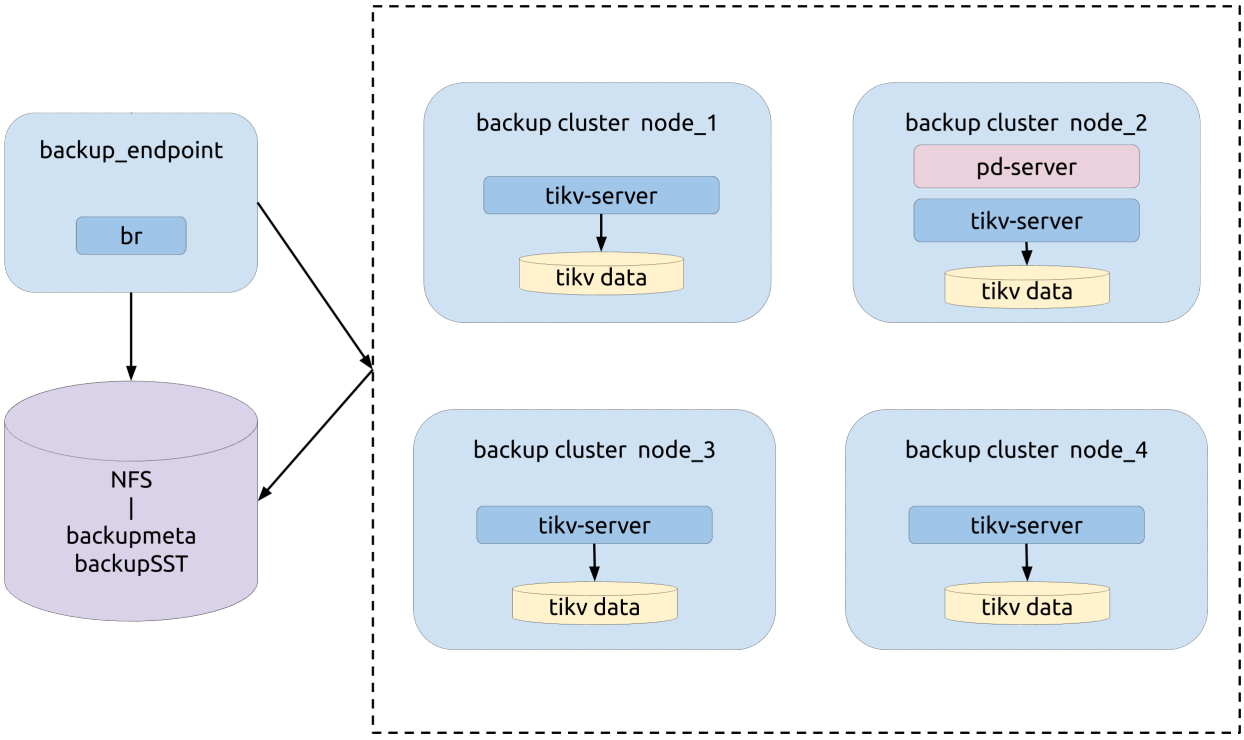


图 28: img

运行备份

备份操作前，在 TiDB 中使用 `admin checksum table order_line` 命令获得备份目标表 `--db batchmark --table order_line` 的统计信息。统计信息示例如下：

```
+-----------+------------+---------------------+-----------+--------------+
| Db_name   | Table_name | Checksum_crc64_xor  | Total_kvs | Total_bytes  |
+-----------+------------+---------------------+-----------+--------------+
| batchmark | order_line | 10912722838344822475 | 5659888624 | 370385538778 |
+-----------+------------+---------------------+-----------+--------------+
1 row in set (5 min 47.59 sec)
```

图 29: img

运行 BR 备份命令：

```
bin/br backup table \
    --db batchmark \
    --table order_line \
    -s local:///br_data \
    --pd ${PD_ADDR}:2379 \
    --log-file backup-nfs.log
```

备份过程中的运行指标

在 BR 备份过程中，需关注以下监控面版中的运行指标来了解备份的状态。

Backup CPU Utilization：参与备份的 TiKV 节点（例如 backup-worker 和 backup-endpoint）的 CPU 使用率。

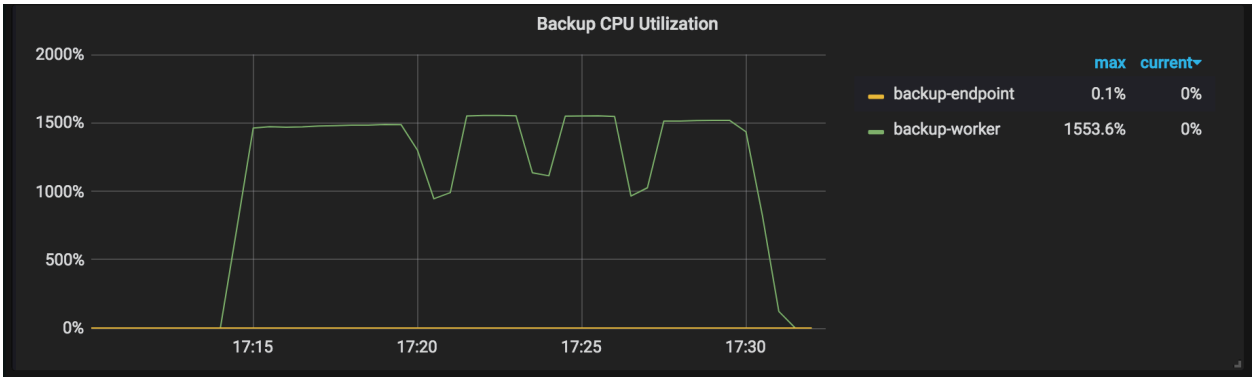


图 30: img

IO Utilization：参与备份的 TiKV 节点的 I/O 使用率。

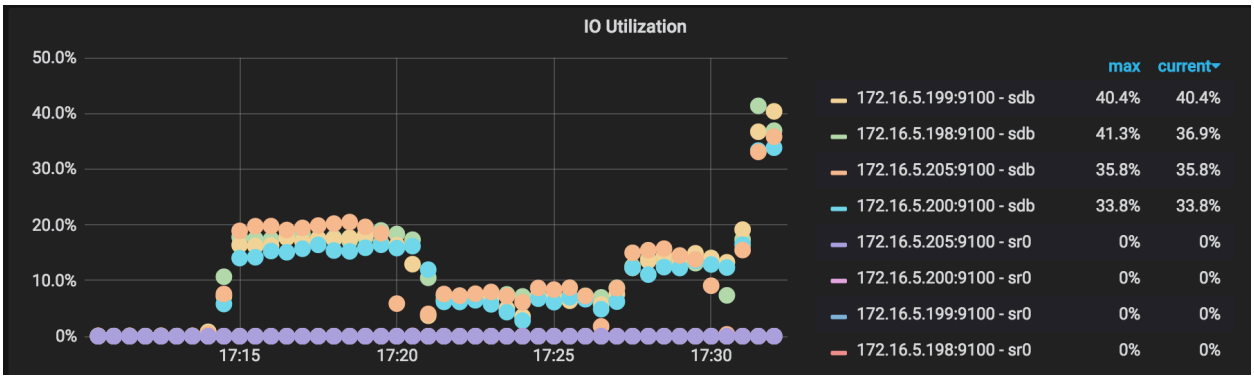


图 31: img

BackupSST Generation Throughput：参与备份的 TiKV 节点生成 backupSST 文件的吞吐。正常时单个 TiKV 节点的吞吐在 150MB/s 左右。

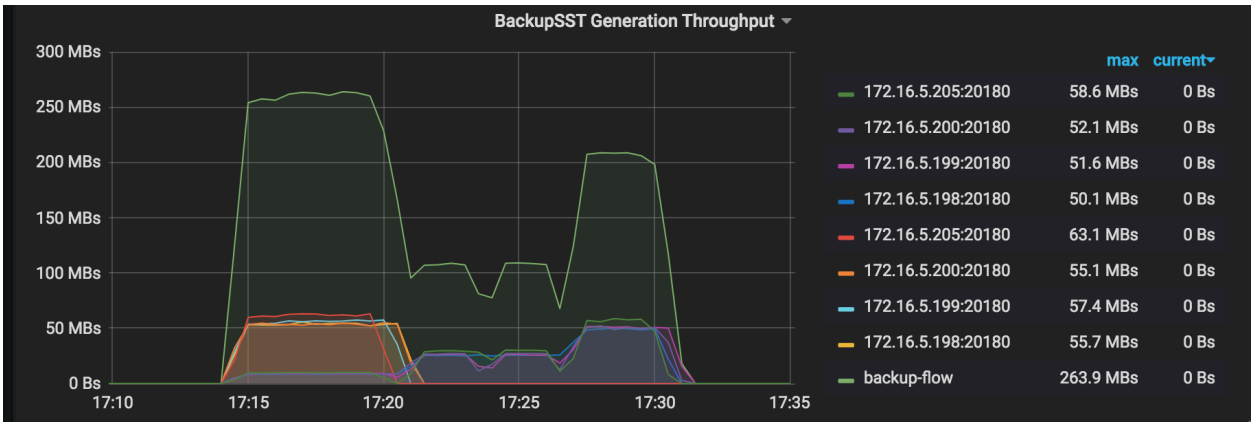


图 32: img

One Backup Range Duration：备份一个 range 的操作耗时，包括扫描耗时 (scan KV) 和保存耗时（保存为 backupSST 文件）。

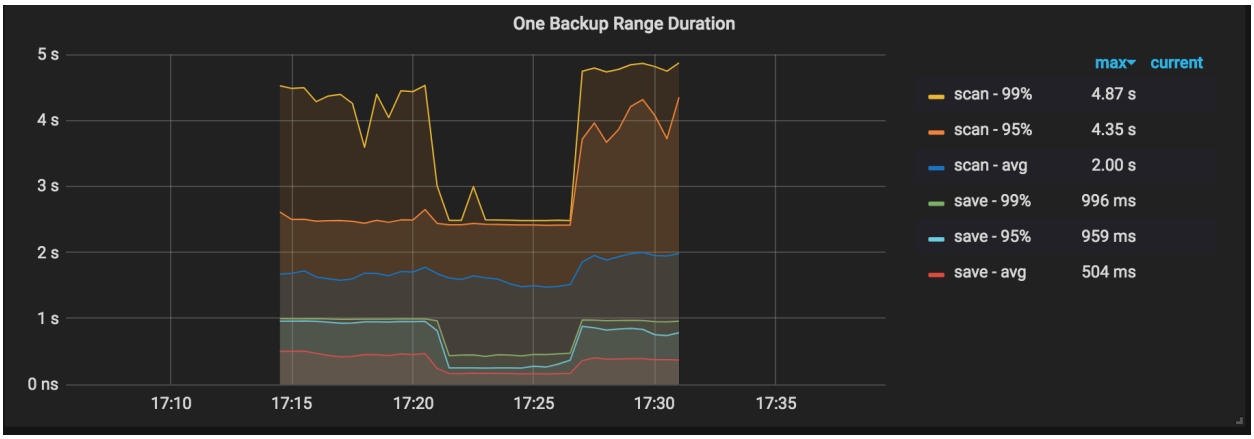


图 33: img

One Backup Subtask Duration：一次备份任务会被拆分成多个子任务。该监控项显示子任务的耗时。

注意：

- 虽然本次任务是备份单表，但因为表中有 3 个索引，所以正常会拆分成 4 个子任务。
- 下图中有 13 个点，说明有 9 次 (13-4) 重试。备份过程中可能发生 Region 调度行为，少量重试是正常的。

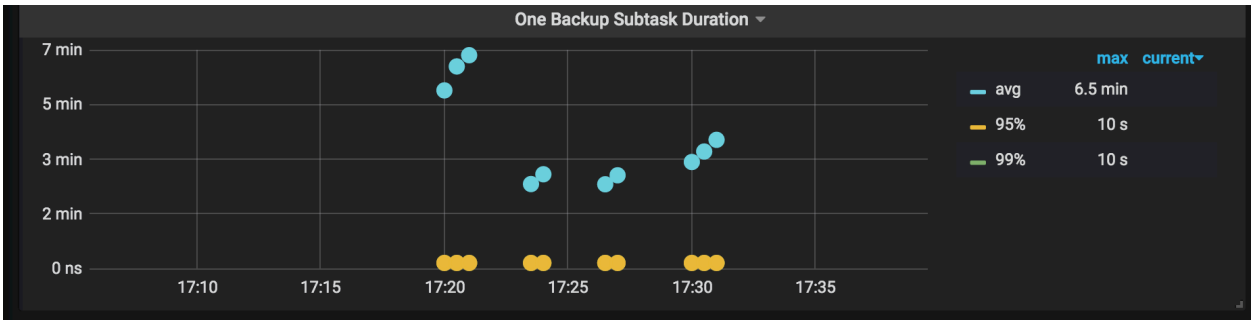


图 34: img

Backup Errors：备份过程中的错误。正常时无错误。即使出现少量错误，备份操作也有重试机制，可能会导致备份时间增加，但不会影响备份的正确性。

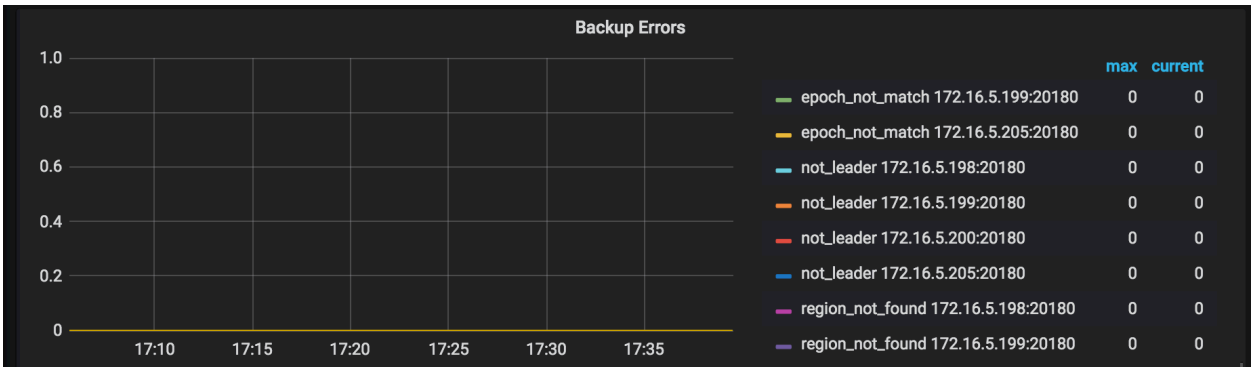


图 35: img

Checksum Request Duration：对备份集群执行 admin checksum 的耗时统计。

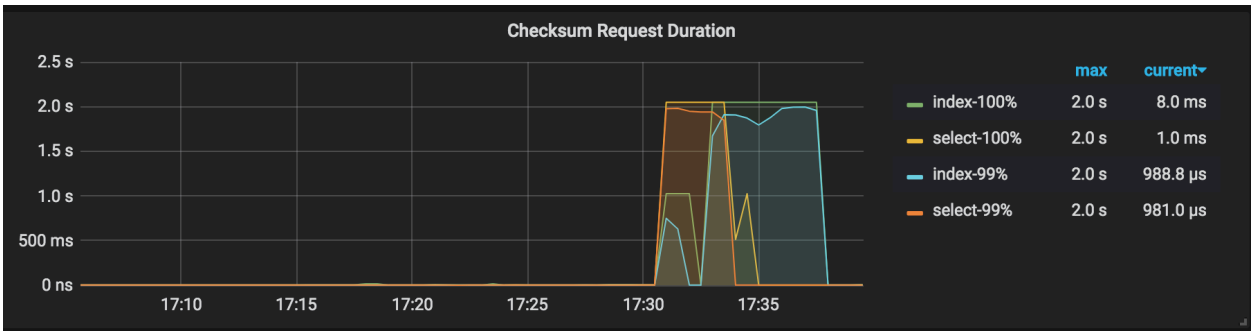


图 36: img

结果解读

BR 会在备份结束时输出备份总结到控制台。

同时使用 BR 前已设置日志的存放路径。从路径下存放的日志中可以获取此次备份的相关统计信息。在日志中搜关键字 “summary”，可以看到以下信息：

```
["Full backup Success summary:
    total backup ranges: 2,
    total success: 2,
    total failed: 0,
    total take(Full backup time): 31.802912166s,
    total take(real time): 49.799662427s,
    total size(MB): 5997.49,
    avg speed(MB/s): 188.58,
    total kv: 120000000"]
    ["backup checksum"=17.907153678s]
    ["backup fast checksum"=349.333µs]
    ["backup total regions"=43]
```

```
    [BackupTS=422618409346269185]
    [Size=826765915]
```

以上日志信息中包含以下内容：

- 备份耗时：`total take(Full backup time): 31.802912166s`
- 程序运行总耗时：`total take(real time): 49.799662427s`
- 备份数据大小：`total size(MB): 5997.49`
- 备份吞吐：`avg speed(MB/s): 188.58`
- 备份 KV 对数：`total kv: 120000000`
- 校验耗时：`["backup checksum"=17.907153678s]`
- 计算各表 checksum、KV 和 bytes 信息总和的耗时：`["backup fast checksum"=349.333µs]`
- 备份 Region 总数：`["backup total regions"=43]`
- 备份存档经压缩后在磁盘中的实际大小：`[Size=826765915]`
- 备份存档的快照时间戳：`[BackupTS=422618409346269185]`

通过以上数据可以计算得到单个 TiKV 实例的吞吐为：`avg speed(MB/s)/tikv_count` = 62.86。

性能调优

如果 TiKV 的资源使用没有出现明显的瓶颈（例如备份过程中的运行指标中的 Backup CPU Utilization 最高为 1500% 左右，IO Utilization 普遍低于 30%），可以尝试调大 `--concurrency`（默认是 4）参数以进行性能调优。该方法不适用于存在许多小表的场景。

示例如下：

```
bin/br backup table \
    --db batchmark \
    --table order_line \
    -s local:///br_data/ \
    --pd ${PD_ADDR}:2379 \
    --log-file backup-nfs.log \
    --concurrency 16
```

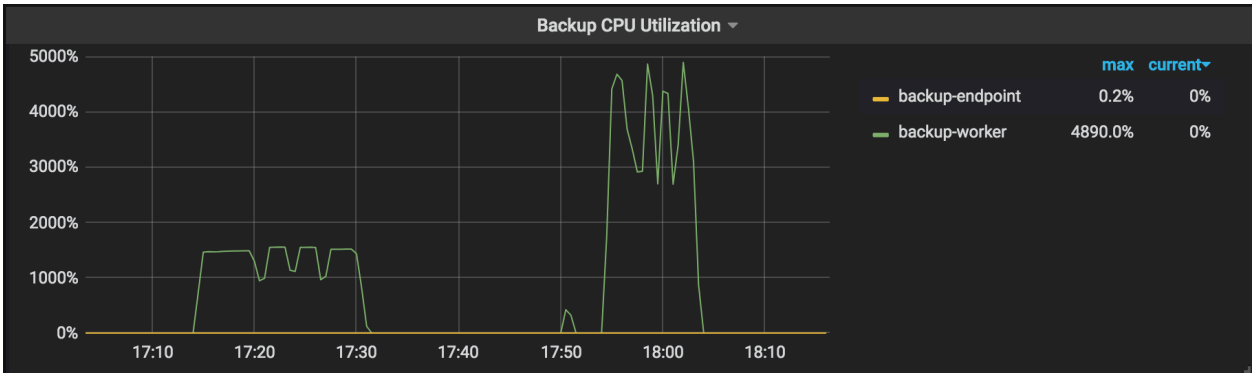


图 37: img

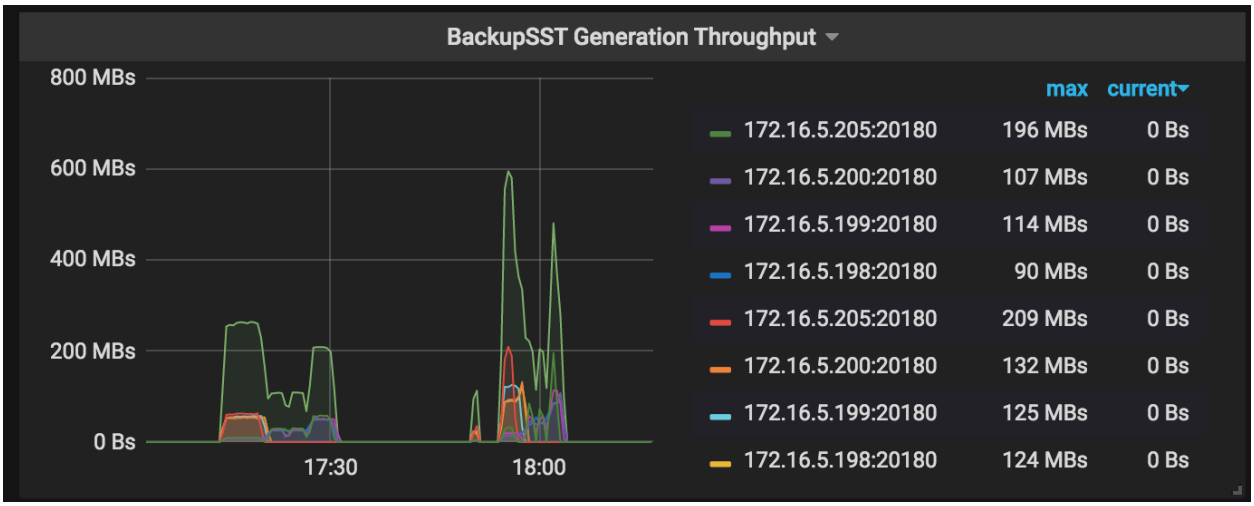


图 38: img

性能调优后的结果如下所示（保持数据大小不变）：

- 备份耗时：`total take(s)` 从 `986.43` 减少到 `535.53`
- 数据大小：`total size(MB): 353227.18`
- 备份吞吐：`avg speed(MB/s)` 从 `358.09` 提升到 `659.59`
- 单个 TiKV 实例的吞吐：`avg speed(MB/s)/tikv_count` 从 `89` 提升到 `164.89`

从网络磁盘恢复备份数据（推荐生产环境使用）

使用 `br restore` 命令，将一份完整的备份数据恢复到一个离线集群。暂不支持恢复到在线集群。

前置要求

- 恢复前的准备工作。

部署拓扑

部署拓扑如下图所示：

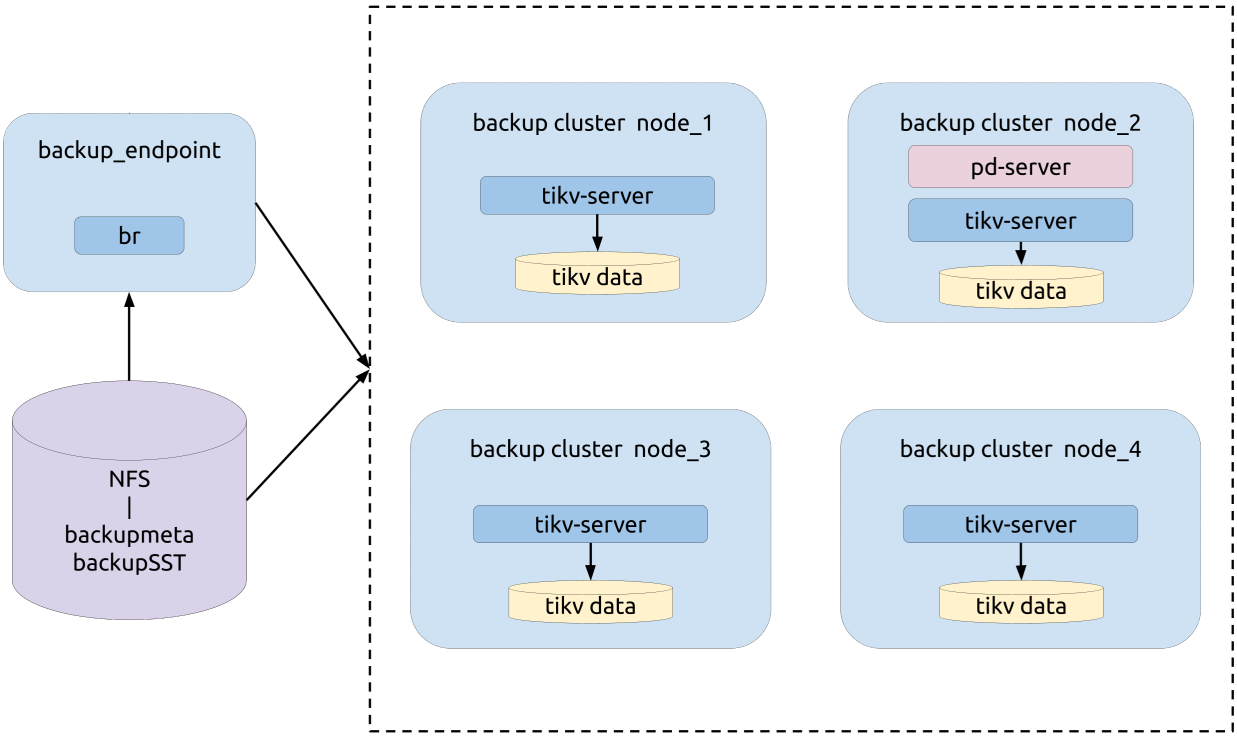


图 39: img

运行恢复

运行 `br restore` 命令：

```
bin/br restore table --db batchmark --table order_line -s local:///br_data --pd 172.16.5.198:2379
    ↪ --log-file restore-nfs.log
```

恢复过程中的运行指标

在 BR 恢复过程中，需关注以下监控面版中的运行指标来了解恢复的状态。

CPU Utilization：参与恢复的 TiKV 节点 CPU 使用率。

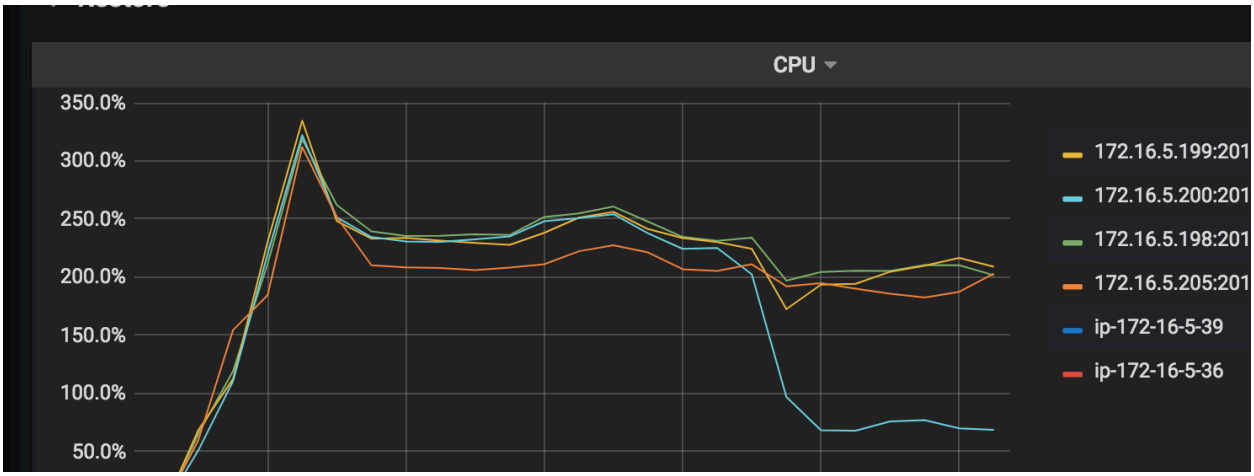


图 40: img

IO Utilization：参与恢复的 TiKV 节点的 I/O 使用率。

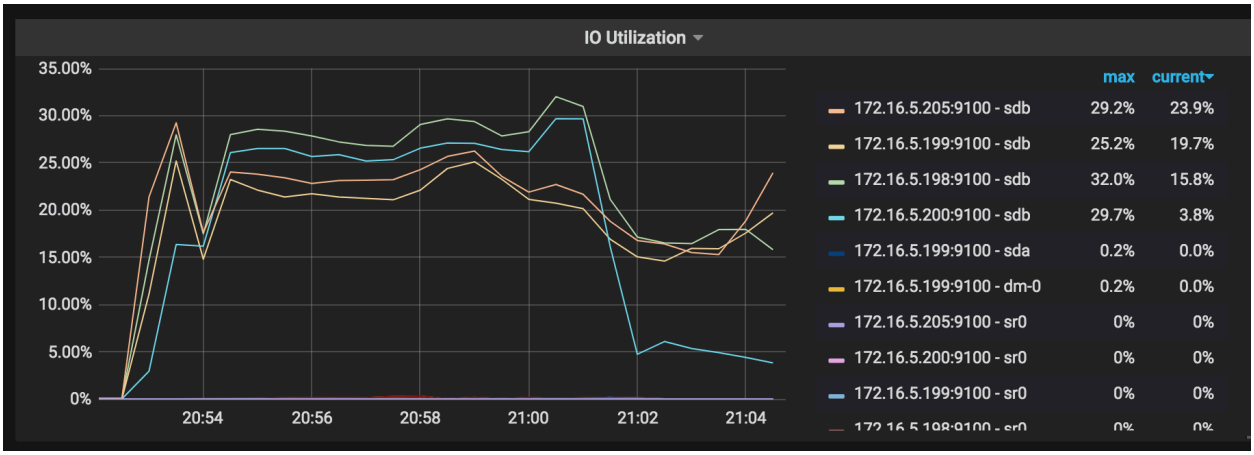


图 41: img

Region 分布：Region 分布越均匀，说明恢复资源利用越充分。

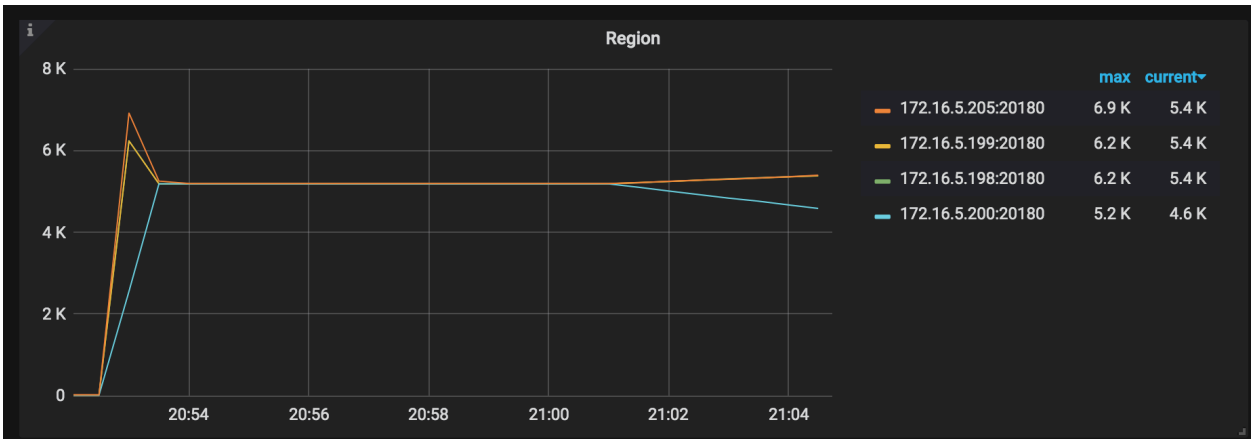


图 42: img

Process SST Duration：处理 SST 文件的延迟。恢复一张表时时，如果 `tableID` 发生了变化，需要对 `tableID` 进行 `rewrite`，否则会进行 `rename`。通常 `rewrite` 延迟要高于 `rename` 的延迟。

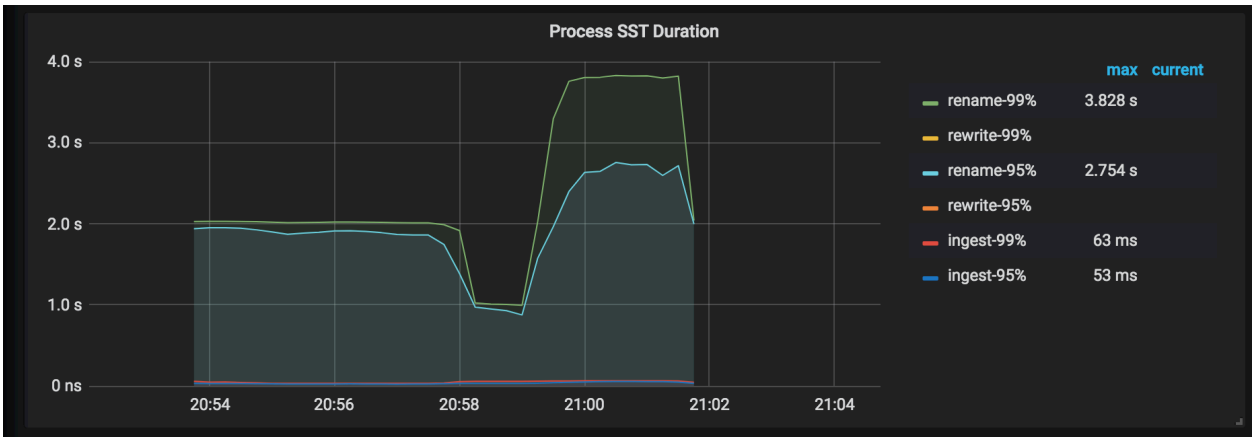


图 43: img

DownLoad SST Throughput：从 External Storage 下载 SST 文件的吞吐。

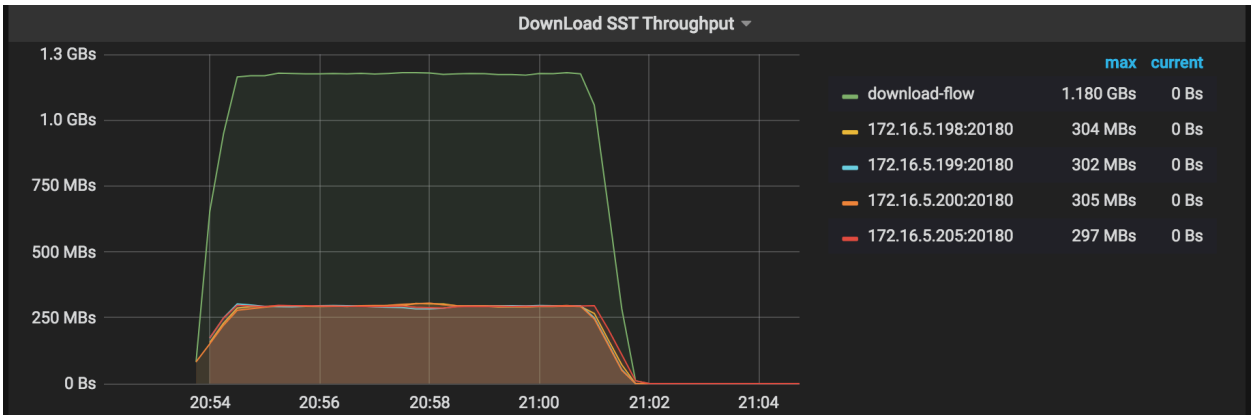


图 44: img

Restore Errors：恢复过程中的错误。

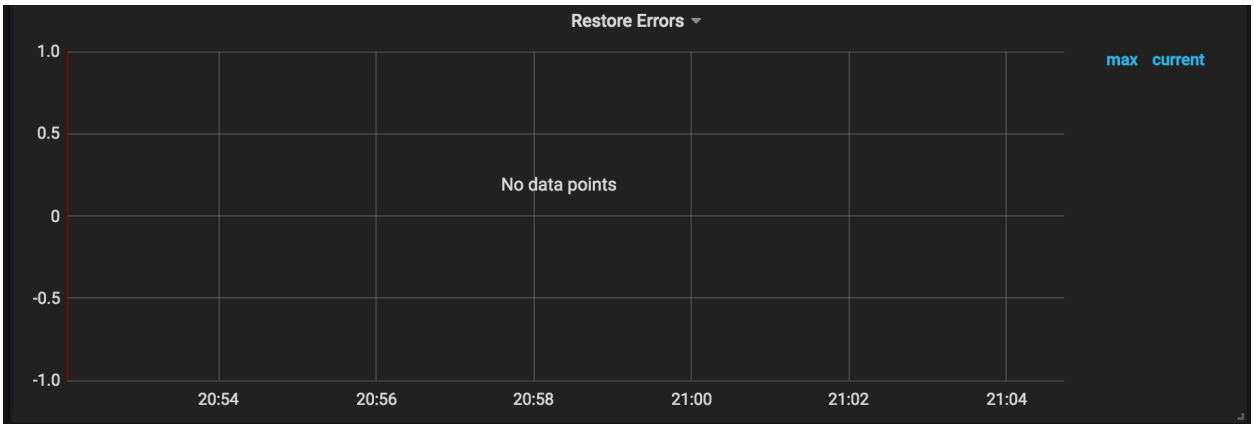


图 45: img

Checksum Request duration：对恢复集群执行 admin checksum 的耗时统计，会比备份时的 checksum 延迟高。

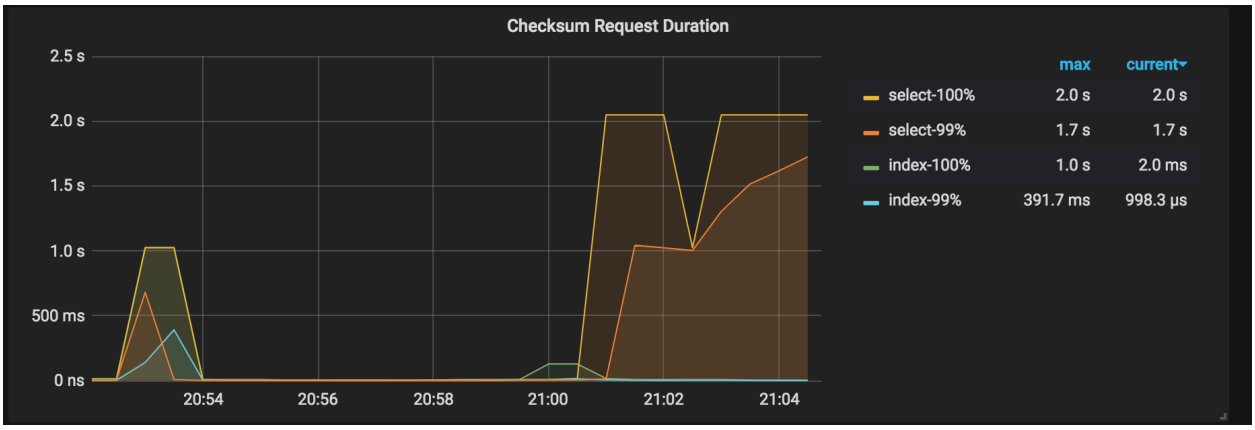


图 46: img

结果解读

使用 BR 前已设置日志的存放路径。从路径下存放的日志中可以获取此次恢复的相关统计信息。在日志中搜关键字“summary”，可以看到以下信息：

```
["Table Restore summary:
    total restore tables: 1,
    total success: 1,
    total failed: 0,
    total take(Full restore time): 17m1.001611365s,
    total take(real time): 16m1.371611365s,
    total kv: 5659888624,
    total size(MB): 353227.18,
    avg speed(MB/s): 367.42"]
    ["restore files"=9263]
    ["restore ranges"=6888]
    ["split region"=49.049182743s]
    ["restore checksum"=6m34.879439498s]
    [Size=48693068713]
```

以上日志信息中包含以下内容：

- 恢复耗时：`total take(Full restore time): 17m1.001611365s`
- 程序运行总耗时：`total take(real time): 16m1.371611365s`
- 恢复数据大小：`total size(MB): 353227.18`
- 恢复 KV 对数：`total kv: 5659888624`
- 恢复吞吐：`avg speed(MB/s): 367.42`
- Region Split 耗时：`take=49.049182743s`
- 校验耗时：`restore checksum=6m34.879439498s`
- 恢复存档在磁盘中的实际大小：`[Size=48693068713]`

根据上表数据可以计算得到：

- 单个 TiKV 吞吐：avg speed(MB/s)/tikv_count = 91.8
- 单个 TiKV 平均恢复速度：total size(MB)/(split time + restore time)/tikv_count = 87.4

性能调优

如果 TiKV 资源使用没有明显的瓶颈，可以尝试调大 --concurrency 参数（默认为 128），示例如下：

```
bin/br restore table --db batchmark --table order_line -s local:///br_data/ --pd
    ↪ 172.16.5.198:2379 --log-file restore-concurrency.log --concurrency 1024
```

性能调优后的结果如下表所示（保持数据大小不变）：

- 恢复耗时：total take(s) 从 961.37 减少到 443.49
- 恢复吞吐：avg speed(MB/s) 从 367.42 提升到 796.47
- 单个 TiKV 实例的吞吐：avg speed(MB/s)/tikv_count 从 91.8 提升到 199.1
- 单个 TiKV 实例的平均恢复速度：total size(MB)/(split time + restore time)/tikv_count 从 87.4 提升到 162.3

将单表数据备份到本地磁盘（推荐测试环境试用）

使用 br backup 命令，将单表数据 --db batchmark --table order_line 备份到指定的本地磁盘路径 local
↪ :///home/tidb/backup_local 下。

前置要求

- 备份前的准备工作。
- 各个 TiKV 节点有单独的磁盘用来存放 backupSST 数据。
- backup_endpoint 节点有单独的磁盘用来存放备份的 backupmeta 文件。
- TiKV 和 backup_endpoint 节点需要有相同的备份目录，例如 /home/tidb/backup_local。

部署拓扑

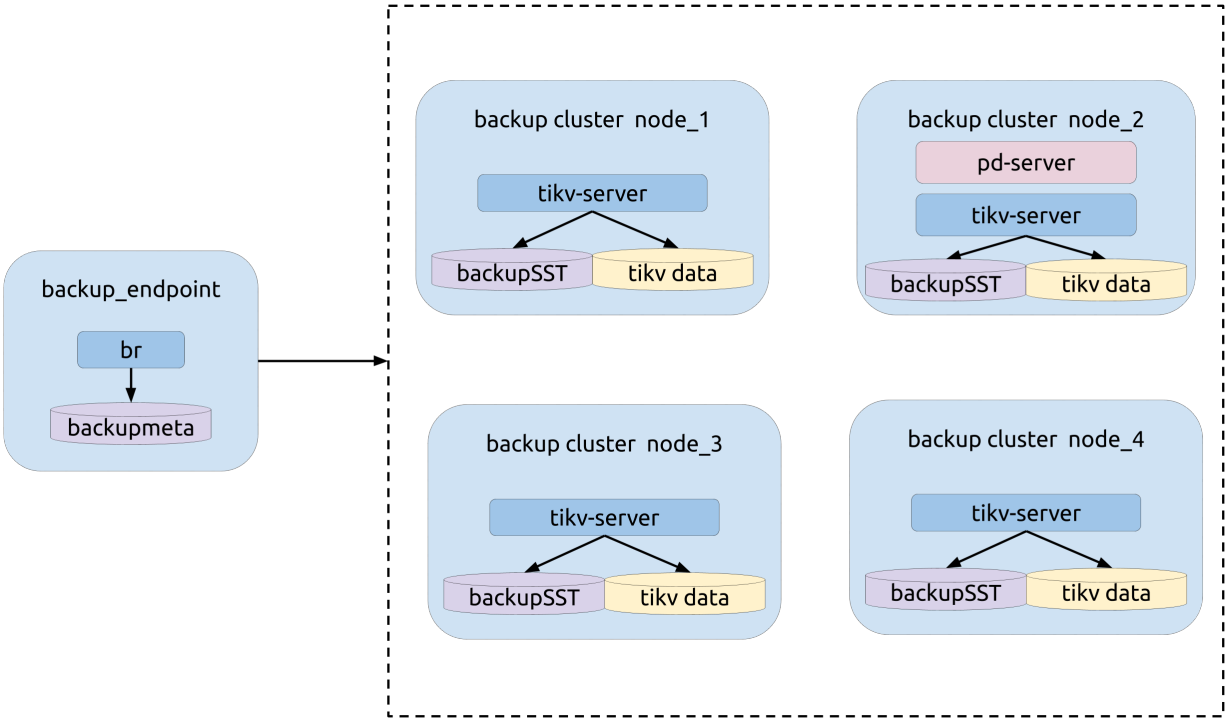


图 47: img

运行备份

备份前在 TiDB 里通过 `admin checksum table order_line` 获得备份的目标表 `--db batchmark --table order_line` 的统计信息。统计信息示例如下：

```
+-----------+------------+--------------------+-----------+-------------+
| Db_name   | Table_name | Checksum_crc64_xor | Total_kvs | Total_bytes |
+-----------+------------+--------------------+-----------+-------------+
| batchmark | order_line | 10912722838344822475 | 5659888624 | 370385538778 |
+-----------+------------+--------------------+-----------+-------------+
1 row in set (5 min 47.59 sec)
```

图 48: img

运行 `br backup` 命令：

```
bin/br backup table \
    --db batchmark \
    --table order_line \
    -s local:///home/tidb/backup_local/ \
    --pd ${PD_ADDR}:2379 \
    --log-file backup_local.log
```

运行备份时，参考备份过程中的运行指标对相关指标进行监控，以了解备份状态。

结果解读

使用 BR 前已设置日志的存放路径。从该路径下存放的日志获取此次备份的相关统计信息。在日志中搜关键字 “summary”，可以看到以下信息：

```
["Table backup summary: total backup ranges: 4, total success: 4, total failed: 0, total take(s):
    ↪  551.31, total kv: 5659888624, total size(MB): 353227.18, avg speed(MB/s): 640.71"] ["
    ↪ backup total regions"=6795] ["backup checksum"=6m33.962719217s] ["backup fast checksum
    ↪ "=22.995552ms]
```

以上日志信息中包含以下内容：

- 备份耗时：`total take(s): 551.31`
- 数据大小：`total size(MB): 353227.18`
- 备份吞吐：`avg speed(MB/s): 640.71`
- 校验耗时：`take=6m33.962719217s`

根据上表数据可以计算得到单个 TiKV 实例的吞吐：`avg speed(MB/s)/tikv_count` = 160。

从本地磁盘恢复备份数据（推荐测试环境试用）

使用 `br restore` 命令，将一份完整的备份数据恢复到一个离线集群。暂不支持恢复到在线集群。

前置要求

- 恢复前的准备工作。
- 集群中没有与备份数据相同的库表。目前 BR 不支持 table route。
- 集群中各个 TiKV 节点有单独的磁盘用来存放要恢复的 backupSST 数据。
- `restore_endpoint` 节点有单独的磁盘用来存放要恢复的 `backupmeta` 文件。
- 集群中 TiKV 和 `restore_endpoint` 节点需要有相同的备份目录，例如 `/home/tidb/backup_local/`。

如果备份数据存放在本地磁盘，那么需要执行以下的步骤：

1. 汇总所有 backupSST 文件到一个统一的目录下。
2. 将汇总后的 backupSST 文件到复制到集群的所有 TiKV 节点下。
3. 将 `backupmeta` 文件复制到到 `restore endpoint` 节点下。

部署拓扑

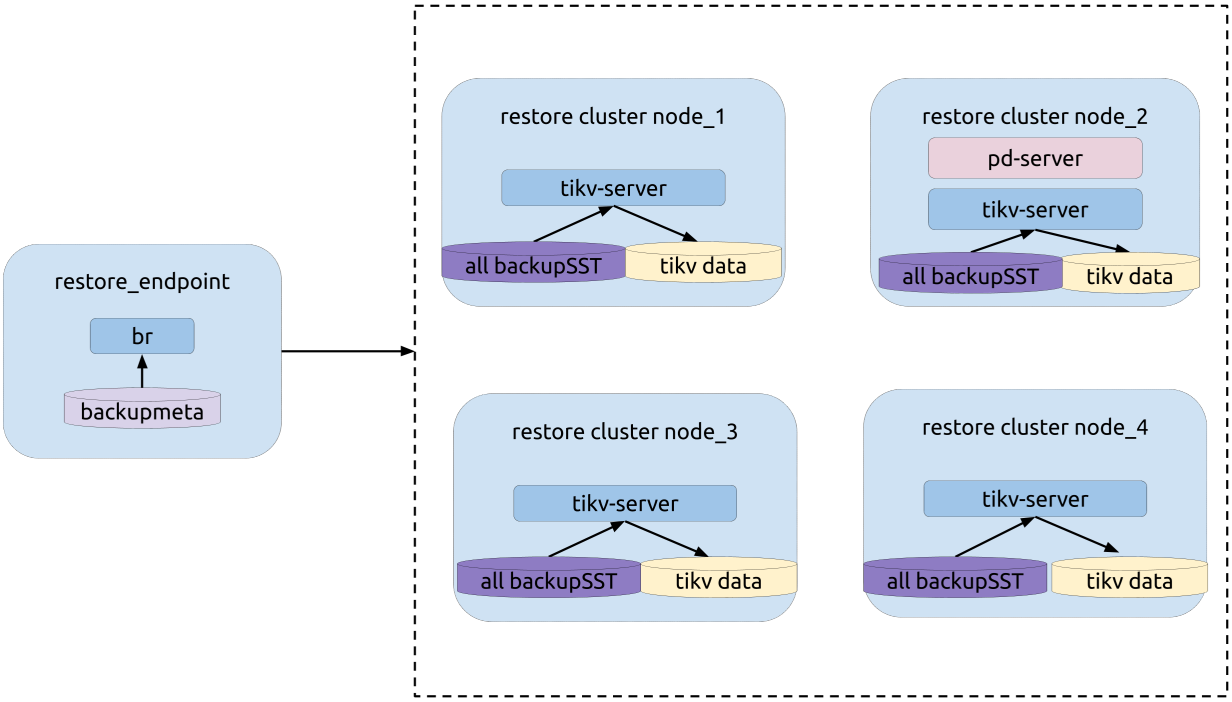


图 49: img

运行恢复

运行 `br restore` 命令：

```
bin/br restore table --db batchmark --table order_line -s local:///home/tidb/backup_local/ --pd
    ↪ 172.16.5.198:2379 --log-file restore_local.log
```

运行恢复时，参考恢复过程中的运行指标对相关指标进行监控，以了解恢复状态。

结果解读

使用 BR 前已设置日志的存放路径。从该日志中可以获取此次恢复的相关统计信息。在日志中搜关键字“summary”，可以看到以下信息：

```
["Table Restore summary: total restore tables: 1, total success: 1, total failed: 0, total take(s
    ↪ ): 908.42, total kv: 5659888624, total size(MB): 353227.18, avg speed(MB/s): 388.84"] ["
    ↪ restore files"=9263] ["restore ranges"=6888] ["split region"=58.7885518s] ["restore
    ↪ checksum"=6m19.349067937s]
```

以上日志信息中包含以下内容：

- 恢复耗时：`total take(s): 908.42`
- 数据大小：`total size(MB): 353227.18`
- 恢复吞吐：`avg speed(MB/s): 388.84`
- Region Split 耗时：`take=58.7885518s`

- 校验耗时：take=6m19.349067937s

根据上表数据可以计算得到：

- 单个 TiKV 实例的吞吐：avg speed(MB/s)/tikv_count = 97.2
- 单个 TiKV 实例的平均恢复速度：total size(MB)/(split time + restore time)/tikv_count = 92.4

6.3.1.3.4 备份过程中的异常处理

本节介绍如何处理备份过程中出现的常见错误。

备份日志中出现 key locked Error

日志中的错误消息：log - ["backup occur kv error"][error="{\"KvError\":{\"locked\":

如果在备份过程中遇到 key 被锁住，目前 BR 会尝试清锁。少量报错不会影响备份的正确性。

备份失败

日志中的错误消息：log - Error: msg:"Io(Custom { kind: AlreadyExists, error: \"[5_5359_42_123_default
↪ .sst] is already exists in /dir/backup_local/\" })"

若备份失败并出现以上错误消息，采取以下其中一种操作后再重新备份：

- 更换备份数据目录。例如将 /dir/backup-2020-01-01/ 改为 /dir/backup_local/。
- 删除所有 TiKV 和 BR 节点的备份目录。

6.3.1.4 外部存储

Backup & Restore (BR)、TiDB Lightning 和 Dumpling 皆支持在本地文件系统和 Amazon S3 上读写数据；另外 BR 亦支持 Google Cloud Storage (GCS)。通过传入不同 URL scheme 到 BR 的 --storage (-s) 参数、TiDB Lightning 的 -d 参数及 Dumpling 中的 --output (-o) 参数，可以区分不同的存储方式。

6.3.1.4.1 Scheme

TiDB 迁移工具支持以下存储服务：

服务	Scheme	示例
本地文件系统（分布在各节点上）	local	local:///path/to/dest/
Amazon S3 及其他兼容 S3 的服务	s3	s3://bucket-name/prefix/of/dest/
GCS	gcs, gs	gcs://bucket-name/prefix/of/dest/
不写入任何存储（仅作为基准测试）	noop	noop://

6.3.1.4.2 URL 参数

S3 和 GCS 等云存储有时需要额外的连接配置，你可以为这类配置指定参数。例如：

- 用 Dumpling 导出数据到 S3

```
./dumpling -u root -h 127.0.0.1 -P 3306 -B mydb -F 256MiB \
    -o 's3://my-bucket/sql-backup?region=us-west-2'
```

- 用 TiDB Lightning 从 S3 导入数据

```
./tidb-lightning --tidb-port=4000 --pd-urls=127.0.0.1:2379 --backend=local --sorted-kv-dir=/
    ↪ tmp/sorted-kvs \
    -d 's3://my-bucket/sql-backup?region=us-west-2'
```

- 用 TiDB Lightning 从 S3 导入数据（使用路径类型的请求模式）。如果你使用的是 TiDB v4.0.11 及以下版本，需要设置 `force-path-style=true` 后才能使用路径类型的请求模式。

```
./tidb-lightning --tidb-port=4000 --pd-urls=127.0.0.1:2379 --backend=local --sorted-kv-dir=/
    ↪ tmp/sorted-kvs \
    -d 's3://my-bucket/sql-backup?force-path-style=true&endpoint=http://10.154.10.132:8088'
```

- 用 BR 备份到 GCS

```
./br backup full -u 127.0.0.1:2379 \
    -s 'gcs://bucket-name/prefix'
```

S3 的 URL 参数

URL 参数	描述
access-key	访问密钥
secret-access-key	secret 访问密钥
region	Amazon S3 服务区域（默认为 us-east-1）
use-accelerate-endpoint	是否在 Amazon S3 上使用加速端点（默认为 false）
endpoint	S3 兼容服务自定义端点的 URL（例如 https://s3.example.com/）
force-path-style	使用 path-style，而不是 virtual-hosted style（默认为 false）
storage-class	上传对象的存储类别（例如 STANDARD、STANDARD_IA）
sse	用于加密上传的服务器端加密算法（可以设置为空，AES256 或 aws:kms）
sse-kms-key-id	如果 sse 设置为 aws:kms，则使用该参数指定 KMS ID
acl	上传对象的 canned ACL（例如，private、authenticated-read）

注意：

不建议在存储 URL 中直接传递访问密钥和 secret 访问密钥，因为这些密钥是明文记录的。迁移工具尝试按照以下顺序从环境中推断这些密钥：

1. $AWS_ACCESS_KEY_ID 和 $AWS_SECRET_ACCESS_KEY 环境变量。
2. $AWS_ACCESS_KEY 和 $AWS_SECRET_KEY 环境变量。

3. 工具节点上的共享凭证文件，路径由 $AWS_SHARED_CREDENTIALS_FILE 环境变量指定。
4. 工具节点上的共享凭证文件，路径为 ~/.aws/credentials。
5. 当前 Amazon EC2 容器的 IAM 角色。
6. 当前 Amazon ECS 任务的 IAM 角色。

GCS 的 URL 参数

URL 参数	描述
credentials-file	迁移工具节点上的凭证 JSON 文件的路径
storage-class	上传对象的存储类别（例如 STANDARD、COLDLINE）
predefined-acl	上传对象的预定义 ACL（例如 private、project-private）

如果没有指定 credentials-file，迁移工具尝试按照以下顺序从环境中推断出凭证：

1. 工具节点上位于 $GOOGLE_APPLICATION_CREDENTIALS 环境变量所指定路径的文件内容。
2. 工具节点上位于 ~/.config/gcloud/application_default_credentials.json 的文件内容。
3. 在 GCE 或 GAE 中运行时，从元数据服务器中获取的凭证。

6.3.1.4.3 命令行参数

除了使用 URL 参数，BR 和 Dumpling 工具亦支持从命令行指定这些配置，例如：

```
./dumpling -u root -h 127.0.0.1 -P 3306 -B mydb -F 256MiB \
    -o 's3://my-bucket/sql-backup' \
    --s3.region 'us-west-2'
```

如果同时指定了 URL 参数和命令行参数，命令行参数会覆盖 URL 参数。

S3 的命令行参数

命令行参数	描述
--s3.region	Amazon S3 服务区域（默认为 us-east-1）
--s3.endpoint	S3 兼容服务自定义端点的 URL（例如 https://s3.example.com/）
--s3.storage-class	上传对象的存储类别（例如 STANDARD、STANDARD_IA）
--s3.sse	用于加密上传的服务器端加密算法（可以设置为空，AES256 或 aws:kms）
--s3.sse-kms-key-id	如果 --s3.sse 设置为 aws:kms，则使用该参数指定 KMS ID
--s3.acl	上传对象的 canned ACL（例如，private、authenticated-read）
--s3.provider	S3 兼容服务类型（支持 aws、alibaba、ceph、netease 或 other）

GCS 的命令行参数

命令行参数	描述
--gcs.credentials-file	迁移工具节点上的凭证 JSON 文件的路径
--gcs.storage-class	上传对象的存储类别（例如 STANDARD、COLDLINE）

命令行参数	描述
`--gcs.predefined-acl`	上传对象的预定义 ACL（例如 `private`、`project-private`）

6.3.1.4.4 BR 向 TiKV 发送凭证

在默认情况下，使用 S3 和 GCS 存储时，BR 会将凭证发送到每个 TiKV 节点，以减少设置的复杂性。

但是，这个操作不适合云端环境，因为每个节点都有自己的角色和权限。在这种情况下，你需要用 `--send-credentials-to-tikv=false`（或简写为 `-c=0`）来禁止发送凭证：

```
./br backup full -c=0 -u pd-service:2379 -s 's3://bucket-name/prefix'
```

使用 SQL 进行备份恢复时，可加上 `SEND_CREDENTIALS_TO_TIKV = FALSE` 选项：

```
BACKUP DATABASE * TO 's3://bucket-name/prefix' SEND_CREDENTIALS_TO_TIKV = FALSE;
```

此参数不适用于 TiDB Lightning 和 Dumping，因为目前它们都是单机程序。

6.3.1.5 Backup & Restore 常见问题

本文列出了在使用 Backup & Restore (BR) 时，可能会遇到的问题及相应的解决方法。

如果遇到未包含在此文档且无法解决的问题，可以在 AskTUG 社区中提问。

6.3.1.5.1 恢复的时候，报错 `could not read local://...:download sst failed`，该如何处理？

在恢复的时候，每个节点都必须能够访问到所有的备份文件（SST files），默认情况下，假如使用 local storage，备份文件会分散在各个节点中，此时是无法直接恢复的，必须将每个 TiKV 节点的备份文件拷贝到其它所有 TiKV 节点才能恢复。

建议在备份的时候挂载一块 NFS 网盘作为备份盘，详见将单表数据备份到网络盘。

6.3.1.5.2 BR 备份时，对集群影响多大？

使用 `sysbench` 的 `oltp_read_only` 场景全速备份到非服务盘，对集群的影响依照表结构的不同，对集群 QPS 的影响在 15%~25% 之间。

如果需要控制备份带来的影响，可以使用 `--ratelimit` 参数限速。

6.3.1.5.3 BR 会备份系统表吗？在数据恢复的时候，这些系统表会冲突吗？

全量备份的时候会过滤掉系统库（`information_schema`，`performance_schema`，`mysql`）。参考备份原理。

因为这些系统库根本不可能存在于备份中，恢复的时候自然不可能发生冲突。

6.3.1.5.4 BR 遇到 Permission denied 或者 No such file or directory 错误，即使用 root 运行 BR 也无法解决，该如何处理？

需要确认 TiKV 是否有访问备份目录的权限。如果是备份，确认是否有写权限；如果是恢复，确认是否有读权限。

在进行备份操作时，如果使用本地磁盘或 NFS 作为存储介质，请确保执行 BR 的用户和启动 TiKV 的用户相同（如果 BR 和 TiKV 位于不同的机器，则需要用户的 UID 相同），否则很备份可能会出现该问题。

使用 root 运行 BR 仍旧有可能会因为磁盘权限而失败，因为备份文件 (SST) 的保存是由 TiKV 执行的。

> 注意：
>
> 在恢复的时候也可能遇到同样的问题。
>
> 使用 BR 进行数据的恢复时，检验读权限的时机是在第一次读取 SST 文件时，考虑到执行 DDL 的耗时，这个时刻可能会离开始运行 BR 的时间很远。这样可能会出现等了很长时间之后遇到 Permission denied 错误失败的情况。
>
> 因此，最好在恢复前提前检查权限。

6.3.1.5.5 BR 遇到错误信息 `Io(Os...)`，该如何处理？

这类问题几乎都是 TiKV 在写盘的时候遇到的系统调用错误。例如遇到 `Io(Os { code: 13, kind: PermissionDenied...})` 或者 `Io(Os { code: 2, kind: NotFound...})` 这类错误信息，首先检查备份目录的挂载方式和文件系统，试试看备份到其它文件夹或者其它硬盘。

目前已知备份到 samba 搭建的网盘时可能会遇到 `Code: 22(invalid argument)` 错误。

6.3.1.5.6 BR 遇到错误信息 `rpc error: code = Unavailable desc =...`，该如何处理？

该问题一般是因为使用 BR 恢复数据的时候，恢复集群的性能不足导致的。可以从恢复集群的监控或者 TiKV 日志来辅助确认。

要解决这类问题，可以尝试扩大集群资源，以及调小恢复时的并发度 (concurrency)，打开限速 (ratelimit) 设置。

6.3.1.5.7 使用 local storage 的时候，BR 备份的文件会存在哪里？

在使用 local storage 的时候，会在运行 BR 的节点生成 `backupmeta`，在各个 Region 的 Leader 节点生成备份文件。

6.3.1.5.8 备份数据有多大，备份会有副本吗？

备份的时候仅仅在每个 Region 的 Leader 处生成该 Region 的备份文件。因此备份的大小等于数据大小，不会有多余的副本数据。所以最终的总大小大约是 TiKV 数据总量除以副本数。

但是假如想要从本地恢复数据，因为每个 TiKV 都必须要能访问到所有备份文件，在最终恢复的时候会有等同于恢复时 TiKV 节点数量的副本。

6.3.1.5.9 BR 恢复到 TiCDC / Drainer 的上游集群时，要注意些什么？

- BR 恢复的数据无法被同步到下游，因为 BR 直接导入 SST 文件，而下游集群目前没有办法获得上游的 SST 文件。
- 在 4.0.3 版本之前，BR 恢复时产生的 DDL jobs 还可能会让 TiCDC / Drainer 执行异常的 DDL。所以，如果一定要在 TiCDC / Drainer 的上游集群执行恢复，请将 BR 恢复的所有表加入 TiCDC / Drainer 的阻止名单。

TiCDC 可以通过配置项中的 `filter.rules` 项完成，Drainer 则可以通过`syncer.ignore-table` 完成。

6.3.1.5.10 BR 会备份表的 SHARD_ROW_ID_BITS 和 PRE_SPLIT_REGIONS 信息吗？恢复出来的表会有多个 Region 吗？

会的，BR 会备份表的`SHARD_ROW_ID_BITS 和 PRE_SPLIT_REGIONS` 信息，并恢复成多个 Region。

6.3.1.5.11 使用 BR 恢复备份数据后，SQL 查询报错 `region is unavailable`

如果 BR 备份的集群有 TiFlash，恢复时会将 TiFlash 信息存进 `TableInfo`。此时如果恢复的集群没有 TiFlash，则会报该错误。计划在未来版本中修复该错误。

6.3.1.5.12 BR 是否支持就地 (in-place) 全量恢复某个历史备份？

不支持。

6.3.1.5.13 在 Kubernetes 环境中如何使用 BR 进行增量备份？

可以使用 kubectl 执行 `kubectl -n ${namespace} get bk ${name}` 以获得上次 BR 备份 `commitTs` 字段，该字段的内容可作为 `--lastbackupts` 使用。

6.3.1.5.14 BR backupTS 如何转化成 Unix 时间?

BR backupTS 默认是在备份开始前，从 PD 获取到的最新时间戳。可以使用 `pd-ctl tso timestamp` 来解析该时间戳，以获得精确值，也可以通过 `backupTS >> 18` 来快速获取估计值。

6.3.1.5.15 BR 恢复存档后是否需要对表执行 ANALYZE 以更新 TiDB 在表和索引上留下的统计信息？

BR 不会备份统计信息（v4.0.9 除外）。所以在恢复存档后需要手动执行 `ANALYZE TABLE` 或等待 TiDB 自动进行 ANALYZE。

BR v4.0.9 备份统计信息使 BR 消耗过多内存，为保证备份过程正常，从 v4.0.10 开始默认关闭备份统计信息的功能。

如果不对表执行 ANALYZE，TiDB 会因统计信息不准确而选不中最优化的执行计划。如果查询性能不是重点关注项，可以忽略 ANALYZE。

6.3.2 使用 Dumpling/TiDB Lightning 备份与恢复

> **警告：**
>
> 本文提供的备份恢复方法已不再推荐，强烈推荐使用BR 工具进行备份恢复，以获得更好的工具体验。

本文档将详细介绍如何使用 Dumpling/TiDB Lightning 对 TiDB 进行全量备份与恢复。增量备份和同步可使用TiDB Binlog。

这里假定 TiDB 服务器信息如下：

服务器名称	服务器地址	端口	用户名	密码
TiDB	127.0.0.1	4000	root	*

在这个备份恢复过程中，会用到下面的工具：

- Dumpling：从 TiDB 导出数据
- TiDB Lightning：导入数据到 TiDB

6.3.2.1 Dumpling/TiDB Lightning 全量备份恢复最佳实践

为了快速地备份恢复数据（特别是数据量巨大的库），可以参考以下建议：

- 导出来的数据文件应当尽可能的小，可以通过设置选项 `-F` 来控制导出来的文件大小。如果后续使用 TiDB Lightning 对备份文件进行恢复，建议把 `dumpling -F` 选项的值设置为 `256m`。
- 如果导出的表中有些表的行数非常多，可以通过设置选项 `-r` 来开启表内并发。

6.3.2.2 从 TiDB 备份数据

使用 `dumpling` 从 TiDB 备份数据的命令如下：

```
./bin/dumpling -h 127.0.0.1 -P 4000 -u root -t 32 -F 256m -T test.t1 -T test.t2 -o ./var/test
```

上述命令中，用 `-T test.t1 -T test.t2` 表明只导出 `test.t1`，`test.t2` 两张表。更多导出数据筛选方式可以参考筛选导出的数据。

`-t 32` 表明使用 32 个线程来导出数据。`-F 256m` 是将实际的表切分成一定大小的 chunk，这里的 chunk 大小为 256MB。

从 v4.0.0 版本开始，Dumpling 可以自动延长 GC 时间（Dumpling 需要访问 TiDB 集群的 PD 地址），而 v4.0.0 之前的版本，需要手动调整 GC 时间，否则 `dumpling` 备份时可能出现以下报错：

```
Could not read data from testSchema.testTable: GC life time is shorter than transaction duration,
    ↪  transaction starts at 2019-08-05 21:10:01.451 +0800 CST, GC safe point is 2019-08-05
    ↪ 21:14:53.801 +0800 CST
```

手动调整 GC 时间的步骤如下：

1. 执行 dumpling 命令前，查询 TiDB 集群的GC 值并在 MySQL 客户端执行下列语句将其调整为合适的值：

```
SELECT * FROM mysql.tidb WHERE VARIABLE_NAME = 'tikv_gc_life_time';
```

```
+--
    ↪ --------------------+----------------------------------------------------------------------------------
    ↪
| VARIABLE_NAME          | VARIABLE_VALUE
    ↪                                                                                                  |
+--
    ↪ --------------------+----------------------------------------------------------------------------------
    ↪
| tikv_gc_life_time      | 10m0s
    ↪
    ↪ |
+--
    ↪ --------------------+----------------------------------------------------------------------------------
    ↪
1 rows in set (0.02 sec)
```

```
UPDATE mysql.tidb SET VARIABLE_VALUE = '720h' WHERE VARIABLE_NAME = 'tikv_gc_life_time';
```

2. 执行 dumpling 命令后，将 TiDB 集群的 GC 值恢复到第 1 步中的初始值：

```
UPDATE mysql.tidb SET VARIABLE_VALUE = '10m' WHERE VARIABLE_NAME = 'tikv_gc_life_time';
```

6.3.2.3 向 TiDB 恢复数据

使用 TiDB Lightning 将之前导出的数据导入到 TiDB，完成恢复操作。具体的使用方法见TiDB Lightning 使用文档

6.4 读取历史数据

本文档介绍 TiDB 如何读取历史版本数据，包括具体的操作流程以及历史数据的保存策略。

6.4.1 功能说明

TiDB 实现了通过标准 SQL 接口读取历史数据功能，无需特殊的 client 或者 driver。当数据被更新、删除后，依然可以通过 SQL 接口将更新/删除前的数据读取出来。

另外即使在更新数据之后，表结构发生了变化，TiDB 依旧能用旧的表结构将数据读取出来。

6.4.2 操作流程

为支持读取历史版本数据，引入了一个新的 system variable: tidb_snapshot，这个变量是 Session 范围有效，可以通过标准的 Set 语句修改其值。其值为文本，能够存储 TSO 和日期时间。TSO 即是全局授时的时间戳，是从 PD 端获取的; 日期时间的格式可以为：“2016-10-08 16:45:26.999”，一般来说可以只写到秒，比如”2016-10-08 16:45:26”。当这个变量被设置时，TiDB 会用这个时间戳建立 Snapshot（没有开销，只是创建数据结构），随后所有的 Select 操作都会在这个 Snapshot 上读取数据。

> 注意：
>
> TiDB 的事务是通过 PD 进行全局授时，所以存储的数据版本也是以 PD 所授时间戳作为版本号。在生成 Snapshot 时，是以 tidb_snapshot 变量的值作为版本号，如果 TiDB Server 所在机器和 PD Server 所在机器的本地时间相差较大，需要以 PD 的时间为准。

当读取历史版本操作结束后，可以结束当前 Session 或者是通过 Set 语句将 tidb_snapshot 变量的值设为 “”，即可读取最新版本的数据。

6.4.3 历史数据保留策略

TiDB 使用 MVCC 管理版本，当更新/删除数据时，不会做真正的数据删除，只会添加一个新版本数据，所以可以保留历史数据。历史数据不会全部保留，超过一定时间的历史数据会被彻底删除，以减小空间占用以及避免历史版本过多引入的性能开销。

TiDB 使用周期性运行的 GC（Garbage Collection，垃圾回收）来进行清理，关于 GC 的详细介绍参见TiDB 垃圾回收 (GC)。

这里需要重点关注的是 `tikv_gc_life_time` 和 `tikv_gc_safe_point` 这条。`tikv_gc_life_time` 用于配置历史版本保留时间，可以手动修改；`tikv_gc_safe_point` 记录了当前的 safePoint，用户可以安全地使用大于 safePoint 的时间戳创建 snapshot 读取历史版本。safePoint 在每次 GC 开始运行时自动更新。

6.4.4 示例

1. 初始化阶段，创建一个表，并插入几行数据：

```
create table t (c int);
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
insert into t values (1), (2), (3);
```

```
Query OK, 3 rows affected (0.00 sec)
```

2. 查看表中的数据：

```
select * from t;
```

```
+------+
| c    |
+------+
|    1 |
|    2 |
|    3 |
+------+
3 rows in set (0.00 sec)
```

3. 查看当前时间：

```
select now();
```

```
+---------------------+
| now()               |
+---------------------+
| 2016-10-08 16:45:26 |
+---------------------+
1 row in set (0.00 sec)
```

4. 更新某一行数据：

```
update t set c=22 where c=2;
```

```
Query OK, 1 row affected (0.00 sec)
```

5. 确认数据已经被更新：

```
select * from t;
```

```
+------+
| c    |
+------+
|    1 |
|   22 |
|    3 |
+------+
3 rows in set (0.00 sec)
```

6. 设置一个特殊的环境变量，这个是一个 session scope 的变量，其意义为读取这个时间之前的最新的一个版本。

```
set @@tidb_snapshot="2016-10-08 16:45:26";
```

```
Query OK, 0 rows affected (0.00 sec)
```

> 注意：
>
> - 这里的时间设置的是 update 语句之前的那个时间。
> - 在 tidb_snapshot 前须使用 @@ 而非 @，因为 @@ 表示系统变量，@ 表示用户变量。

这里读取到的内容即为 update 之前的内容，也就是历史版本：

```
select * from t;
```

```
+------+
| c    |
+------+
|    1 |
|    2 |
|    3 |
+------+
3 rows in set (0.00 sec)
```

7. 清空这个变量后，即可读取最新版本数据：

```
set @@tidb_snapshot="";
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
select * from t;
```

```
+------+
| c    |
+------+
|    1 |
|   22 |
|    3 |
+------+
3 rows in set (0.00 sec)
```

> 注意：
>
> 在 tidb_snapshot 前须使用 @@ 而非 @，因为 @@ 表示系统变量，@ 表示用户变量。

6.5 时区支持

TiDB 使用的时区由 time_zone 全局变量和 session 变量决定。time_zone 的默认值是 System，System 对应的实际时区在 TiDB 集群 bootstrap 初始化时设置。具体逻辑如下：

- 优先使用 TZ 环境变量
- 如果失败，则从 /etc/localtime 的实际软链地址提取。
- 如果上面两种都失败则使用 UTC 作为系统时区。

在运行过程中可以修改全局时区：

```
SET GLOBAL time_zone = timezone;
```

TiDB 还可以通过设置 session 变量 time_zone 为每个连接维护各自的时区。默认条件下，这个值取的是全局变量 time_zone 的值。修改 session 使用的时区：

```
SET time_zone = timezone;
```

查看当前使用的时区的值：

```
SELECT @@global.time_zone, @@session.time_zone;
```

设置 time_zone 的值的格式：

- ‘SYSTEM’ 表明使用系统时间
- 相对于 UTC 时间的偏移，比如 ‘+10:00’ 或者 ‘-6:00’
- 某个时区的名字，比如 ‘Europe/Helsinki’，‘US/Eastern’ 或 ‘MET’

NOW() 和 CURTIME() 的返回值都受到时区设置的影响。

> 注意：
>
> 只有 Timestamp 数据类型的值是受时区影响的。可以理解为，Timestamp 数据类型的实际表示使用的是（字面值 + 时区信息）。其它时间和日期类型，比如 Datetime/Date/Time 是不包含时区信息的，所以也不受到时区变化的影响。

```
create table t (ts timestamp, dt datetime);
```

```
Query OK, 0 rows affected (0.02 sec)
```

```
set @@time_zone = 'UTC';
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
insert into t values ('2017-09-30 11:11:11', '2017-09-30 11:11:11');
```

```
Query OK, 1 row affected (0.00 sec)
```

```
set @@time_zone = '+8:00';
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
select * from t;
```

```
+---------------------|---------------------+
| ts                  | dt                  |
+---------------------|---------------------+
| 2017-09-30 19:11:11 | 2017-09-30 11:11:11 |
+---------------------|---------------------+
1 row in set (0.00 sec)
```

上面的例子中，无论怎么调整时区的值，Datetime 类型字段的值是不受影响的，而 Timestamp 则随着时区改变，显示的值会发生变化。其实 Timestamp 持久化到存储的值始终没有变化过，只是根据时区的不同显示值不同。

Timestamp 类型和 Datetime 等类型的值，两者相互转换的过程中，会涉及到时区。这种情况一律基于 session 的当前 `time_zone` 时区处理。

另外，用户在导数据的过程中，也要需注意主库和从库之间的时区设定是否一致。

6.6 日常巡检

TiDB 作为分布式数据库，对比单机数据库机制更加复杂，其自带的监控项也很丰富。为了更便捷地运维 TiDB，本文介绍了运维 TiDB 集群需要常关注的关键性能指标。

图 50: 实例面板

以上实例面板的各指标说明如下：

- 状态：用于查看实例状态是否正常，如果在线可忽略此项。
- 启动时间：是关键指标。如果有发现启动时间有变动，那么需要排查组件重启的原因。
- 版本、部署路径、Git 哈希值：通过查看这三个指标可以避免有部署路径和版本不一致或者错误的情况。

6.6.1.2 主机面板

图 51: 主机面板

通过主机面板可以查看 CPU、内存、磁盘使用率。当任何资源的使用率超过 80% 时，推荐扩容对应组件。

6.6.1.3 SQL 分析面板

图 52: SQL 分析面板

通过 SQL 分析面板可以分析对集群影响较大的慢 SQL，然后进行对应的 SQL 优化。

6.6.1.4 Region 信息面板

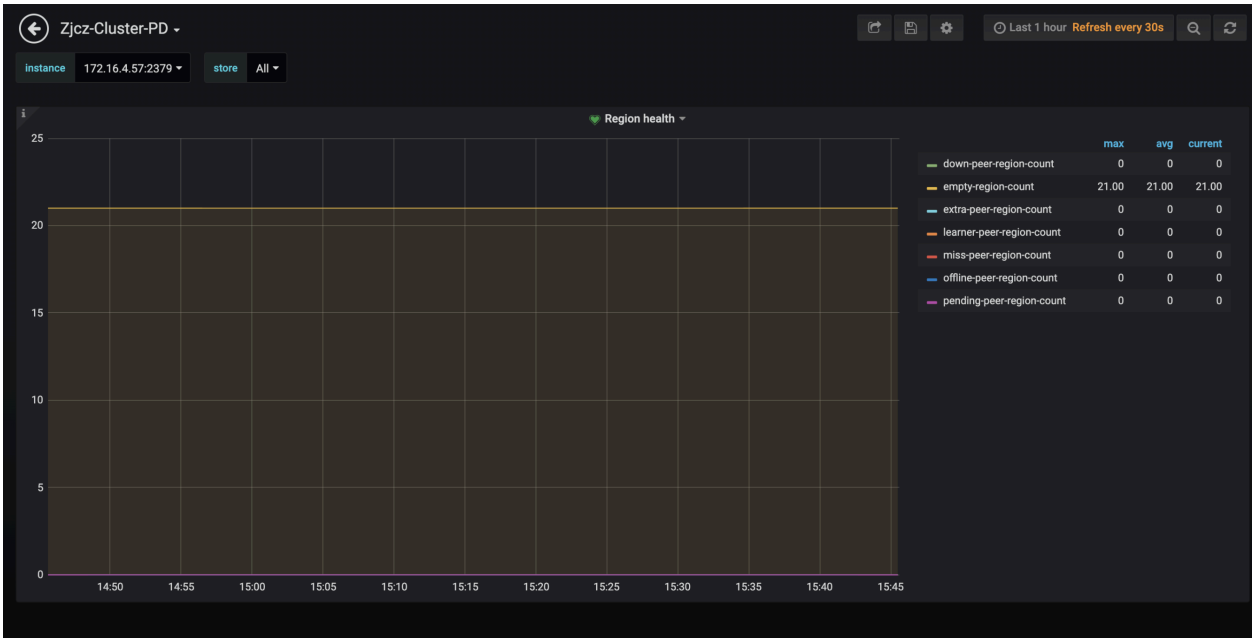


图 53: Region 信息面板

以上 Region 信息面板说明如下：

- `miss-peer-region-count`：缺副本的 Region 数量，不会一直大于 0。
- `extra-peer-region-count`：多副本的 Region 数量，调度过程中会产生。
- `empty-region-count`：空 Region 的数量，一般是 `TRUNCATE TABLE/DROP TABLE` 语句导致。如果数量较多，可以考虑开启跨表 Region merge。
- `pending-peer-region-count`：Raft log 落后的 Region 数量。由于调度产生少量的 pending peer 是正常的，但是如果持续很高，可能有问题。
- `down-peer-region-count`：Raft leader 上报有不响应 peer 的 Region 数量。
- `offline-peer-region-count`：peer 下线过程中的 Region 数量。

原则上来说，该监控面板偶尔有数据是符合预期的。但长期有数据，需要排查是否存在问题。

6.6.1.5 KV Request Duration

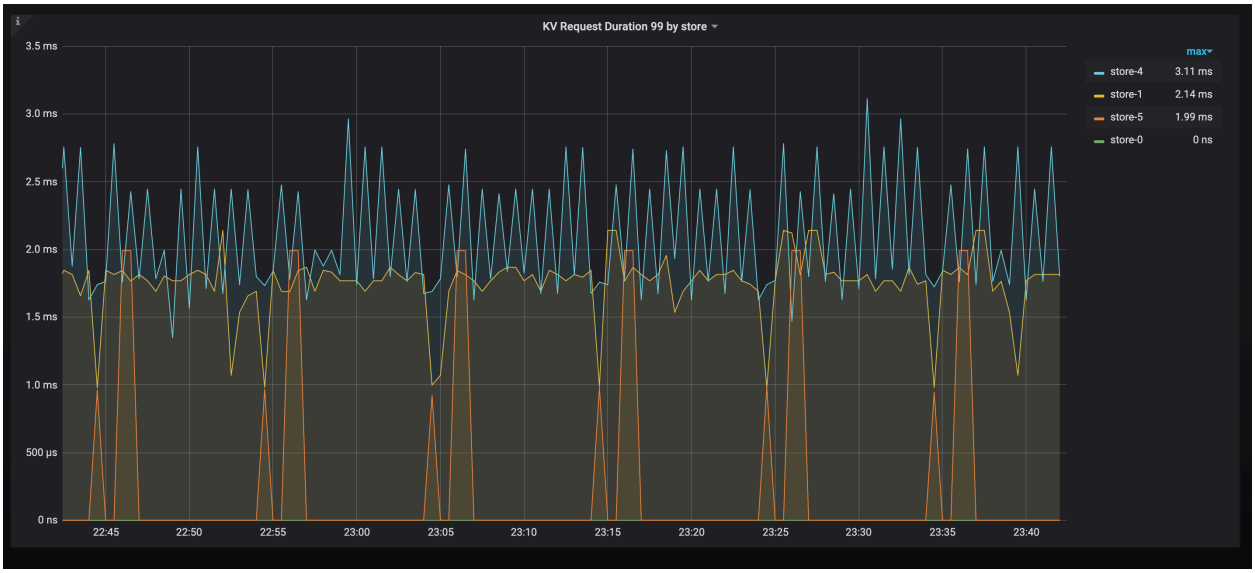


图 54: TiKV 相应时间

TiKV 当前.99（百分位）的响应时间。如果发现有明显高的节点，可以排查是否有热点，或者相关节点性能较差。

6.6.1.6 PD TSO Wait Duration

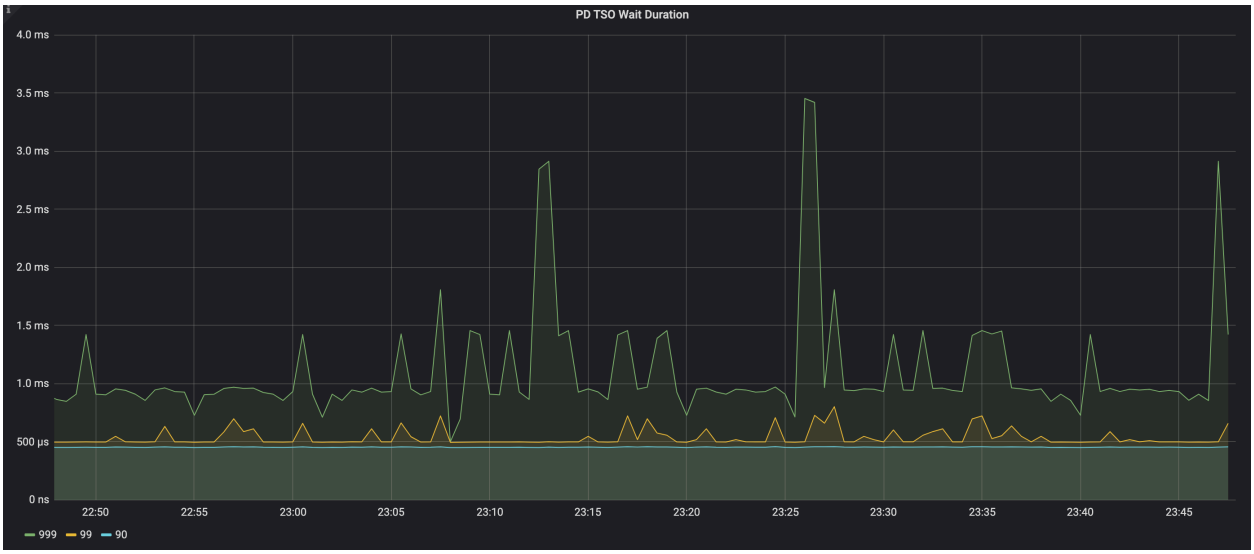


图 55: TiDB 从 PD 获取 TSO 的时间

TiDB 从 PD 获取 TSO 的时间。如果相关响应时间较高，一般常见原因如下：

- TiDB 到 PD 的网络延迟较高，可以手动 Ping 一下网络延迟。

- TiDB 压力较高，导致获取较慢。
- PD 服务器压力较高，导致获取较慢。

6.6.1.7 Overview 面板

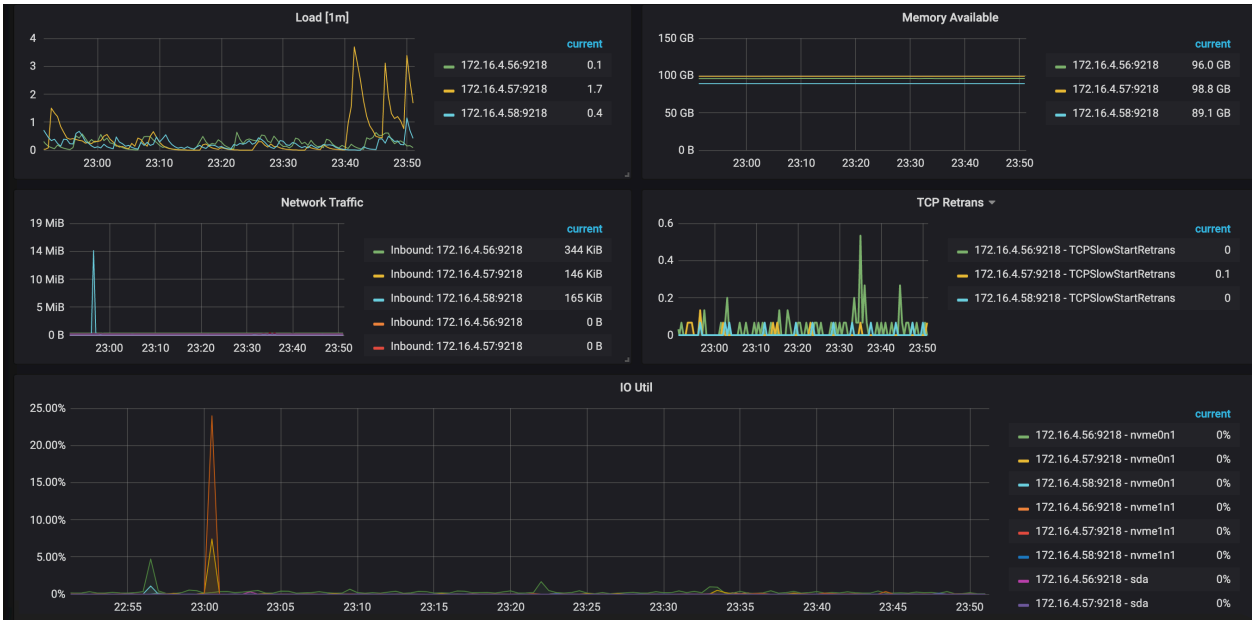


图 56: Overview 面板

以上面板展示常见的负载、内存、网络、IO 监控。发现有瓶颈时，推荐扩容或者优化集群拓扑，优化 SQL、集群参数等。

6.6.1.8 异常监控面板

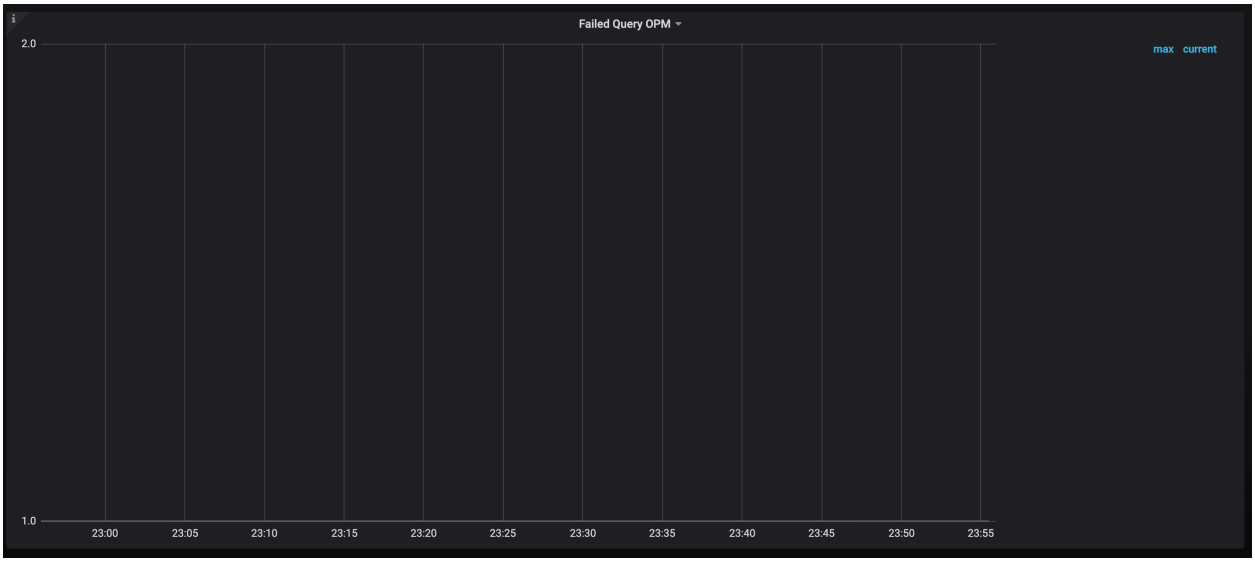


图 57: 异常监控面板

以上面板展示每个 TiDB 实例上，执行 SQL 语句发生的错误，并按照错误类型进行统计，例如语法错误、主键冲突等。

6.6.1.9 GC 状态面板

```
(root@127.0.0.1) [(none)]>select * from mysql.tidb;
+-----------------------+------------------------------------------------------------------------------------------------+-----------------------------------------------------------+
| VARIABLE_NAME         | VARIABLE_VALUE                                                                                 | COMMENT                                                   |
+-----------------------+------------------------------------------------------------------------------------------------+-----------------------------------------------------------+
| bootstrapped          | True                                                                                           | Bootstrap flag. Do not delete.                            |
| tidb_server_version   | 44                                                                                             | Bootstrap version. Do not delete.                         |
| system_tz             | Asia/Shanghai                                                                                  | TiDB Global System Timezone.                              |
| new_collation_enabled | False                                                                                          | If the new collations are enabled. Do no                  |
| tikv_gc_leader_uuid   | 5c8c8865ca80006                                                                                | Current GC worker leader UUID. (DO NOT                    |
| tikv_gc_leader_desc   | host:wangjundeMacBook-Pro.local, pid:81322, start at 2020-05-20 20:49:21.735312 +0800 CST m=+7.904021075 | Host name and pid of current GC leader.                   |
| tikv_gc_leader_lease  | 20200522-00:06:30 +0800                                                                        | Current GC worker leader lease. (DO NOT                   |
| tikv_gc_enable        | true                                                                                           | Current GC enable status                                  |
| tikv_gc_run_interval  | 10m0s                                                                                          | GC run interval, at least 10m, in Go fo                   |
| tikv_gc_life_time     | 10m0s                                                                                          | All versions within life time will not                    |
| tikv_gc_last_run_time | 20200521-23:55:30 +0800                                                                        | The time when last GC starts. (DO NOT E                   |
| tikv_gc_safe_point    | 20200521-23:45:30 +0800                                                                        | All versions after safe point can be ac                   |
| tikv_gc_auto_concurrency | true                                                                                        | Let TiDB pick the concurrency automatica                  |
| tikv_gc_mode          | distributed                                                                                    | Mode of GC, "central" or "distributed"                    |
+-----------------------+------------------------------------------------------------------------------------------------+-----------------------------------------------------------+
14 rows in set (0.01 sec)
```

图 58: GC 状态面板

以上面板展示最后 GC（垃圾清理）的时间，观察 GC 是否正常。如果 GC 发生异常，可能会造成历史数据存留过多，影响访问效率。

6.7 TiFlash 集群运维

本文介绍TiFlash 集群运维的一些常见操作，包括查看 TiFlash 版本，以及 TiFlash 重要日志及系统表。

6.7.1 查看 TiFlash 版本

查看 TiFlash 版本有以下两种方法：

- 假设 TiFlash 的二进制文件名为 `tiflash`，则可以通过 `./tiflash version` 方式获取 TiFlash 版本。

  但是由于 TiFlash 的运行依赖于动态库 `libtiflash_proxy.so`，因此需要将包含动态库 `libtiflash_proxy.so` 的目录路径添加到环境变量 `LD_LIBRARY_PATH` 后，上述命令才能正常执行。

  例如，当 `tiflash` 和 `libtiflash_proxy.so` 在同一个目录下时，切换到该目录后，可以通过如下命令查看 TiFlash 版本：

```
LD_LIBRARY_PATH=./ ./tiflash version
```

- 在 TiFlash 日志（日志路径见配置文件 tiflash.toml [logger] 部分）中查看 TiFlash 版本，例如：

```
<information>: TiFlash version: TiFlash 0.2.0 master-375035282451103999f3863c691e2fc2
```

6.7.2 TiFlash 重要日志介绍

为了更好地兼容 TiDB 的日志格式，TiFlash 在 v4.0.5 中修改了原有的日志格式，因此会有两个不同版本的日志。

如果你的 TiDB 集群版本 < 4.0.5：

日志信息	日志含义
[23] <Information> KVStore: Start to persist [region 47, applied: term 6 index 10]	在 TiFlash 中看到类似日志代表数据开始同步（该日志开头方括号内的数字代表线程号，下同）
[30] <Debug> CoprocessorHandler: grpc::Status DB::CoprocessorHandler::execute()	Handling DAG request，该日志代表 TiFlash 开始处理一个 Coprocessor 请求
[30] <Debug> CoprocessorHandler: grpc::Status DB::CoprocessorHandler::execute()	Handle DAG request done，该日志代表 TiFlash 完成 Coprocessor 请求的处理

如果你的 TiDB 集群版本 >= 4.0.5：

日志信息	日志含义
[INFO] [<unknown>] [“KVStore: Start to persist [region 47, applied: term 6 index 10]”] [thread_id=23]	在 TiFlash 中看到类似日志代表数据开始同步

日志信息	日志含义
[DEBUG] [<unknown>] [“CoprocessorHandler: grpc::Status DB::CoprocessorHandler::execute(): Handling DAG request”] [thread_id=30]	该日志代表 TiFlash 开始处理一个 Coprocessor 请求
[DEBUG] [<unknown>] [“CoprocessorHandler: grpc::Status DB::CoprocessorHandler::execute(): Handle DAG request done”] [thread_id=30]	该日志代表 TiFlash 完成 Coprocessor 请求的处理

你可以找到一个 Coprocessor 请求的开始或结束，然后通过日志前面打印的线程号找到该 Coprocessor 请求的其他相关日志。

6.7.3 TiFlash 系统表

`information_schema.tiflash_replica` 系统表的列名及含义如下：

列名	含义
TABLE_SCHEMA	数据库名
TABLE_NAME	表名
TABLE_ID	表 ID
REPLICA_COUNT	TiFlash 副本数
AVAILABLE	是否可用（0/1）
PROGRESS	同步进度 [0.0~1.0]

6.8 TiUP 常见运维操作

本文介绍了使用 TiUP 运维 TiDB 集群的常见操作，包括查看集群列表、启动集群、查看集群状态、修改配置参数、关闭集群、销毁集群等。

6.8.1 查看集群列表

TiUP cluster 组件可以用来管理多个 TiDB 集群，在每个 TiDB 集群部署完毕后，该集群会出现在 TiUP 的集群列表里，可以使用 list 命令来查看。

```
tiup cluster list
```

6.8.2 启动集群

启动集群操作会按 PD -> TiKV -> Pump -> TiDB -> TiFlash -> Drainer -> TiCDC -> Prometheus -> Grafana -> Alertmanager 的顺序启动整个 TiDB 集群所有组件：

```
tiup cluster start ${cluster-name}
```

> 注意：
>
> 你需要将 `${cluster-name}` 替换成实际的集群名字，若忘记集群名字，可通过 `tiup cluster list` 查看。

该命令支持通过 `-R` 和 `-N` 参数来只启动部分组件。

例如，下列命令只启动 PD 组件：

```
tiup cluster start ${cluster-name} -R pd
```

下列命令只启动 `1.2.3.4` 和 `1.2.3.5` 这两台机器上的 PD 组件：

```
tiup cluster start ${cluster-name} -N 1.2.3.4:2379,1.2.3.5:2379
```

> 注意：
>
> 若通过 `-R` 和 `-N` 启动指定组件，需要保证启动顺序正确（例如需要先启动 PD 才能启动 TiKV），否则可能导致启动失败。

6.8.3 查看集群状态

集群启动之后需要检查每个组件的运行状态，以确保每个组件工作正常。TiUP 提供了 display 命令，节省了登录到每台机器上去查看进程的时间。

```
tiup cluster display ${cluster-name}
```

6.8.4 修改配置参数

集群运行过程中，如果需要调整某个组件的参数，可以使用 `edit-config` 命令来编辑参数。具体的操作步骤如下：

1. 以编辑模式打开该集群的配置文件：

   ```
   tiup cluster edit-config ${cluster-name}
   ```

2. 设置参数：

   首先确定配置的生效范围，有以下两种生效范围：

   - 如果配置的生效范围为该组件全局，则配置到 server_configs。例如：

   ```
   server_configs:
     tidb:
       log.slow-threshold: 300
   ```

   - 如果配置的生效范围为某个节点，则配置到具体节点的 config 中。例如：

   ```
   tidb_servers:
   - host: 10.0.1.11
       port: 4000
       config:
           log.slow-threshold: 300
   ```

   参数的格式参考 TiUP 配置参数模版。

   配置项层次结构使用 . 表示。

   关于组件的更多配置参数说明，可参考 tidb config.toml.example、tikv config.toml.example 和 pd config.toml.example。

3. 执行 reload 命令滚动分发配置、重启相应组件：

   ```
   tiup cluster reload ${cluster-name} [-N <nodes>] [-R <roles>]
   ```

6.8.4.1 示例

如果要调整 tidb-server 中事务大小限制参数 txn-total-size-limit 为 1G，该参数位于 performance 模块下，调整后的配置如下：

```
server_configs:
  tidb:
    performance.txn-total-size-limit: 1073741824
```

然后执行 tiup cluster reload ${cluster-name} -R tidb 命令滚动重启。

6.8.5 Hotfix 版本替换

常规的升级集群请参考升级文档，但是在某些场景下（例如 Debug），可能需要用一个临时的包替换正在运行的组件，此时可以用 patch 命令：

```
tiup cluster patch --help
```

```
Replace the remote package with a specified package and restart the service

Usage:
  tiup cluster patch <cluster-name> <package-path> [flags]

Flags:
  -h, --help                    帮助信息
  -N, --node strings            指定被替换的节点
      --overwrite               在未来的 scale-out 操作中使用当前指定的临时包
  -R, --role strings            指定被替换的服务类型
      --transfer-timeout int    transfer leader 的超时时间

Global Flags:
      --native-ssh        使用系统默认的 SSH 客户端
      --wait-timeout int  等待操作超时的时间
      --ssh-timeout int   SSH 连接的超时时间
  -y, --yes               跳过所有的确认步骤
```

例如，有一个 TiDB 实例的 hotfix 包放在 `/tmp/tidb-hotfix.tar.gz` 目录下。如果此时想要替换集群上的所有 TiDB 实例，则可以执行以下命令：

```
tiup cluster patch test-cluster /tmp/tidb-hotfix.tar.gz -R tidb
```

或者只替换其中一个 TiDB 实例：

```
tiup cluster patch test-cluster /tmp/tidb-hotfix.tar.gz -N 172.16.4.5:4000
```

6.8.6 重命名集群

部署并启动集群后，可以通过 `tiup cluster rename` 命令来对集群重命名：

```
tiup cluster rename ${cluster-name} ${new-name}
```

> 注意：
>
> - 重命名集群会重启监控（Prometheus 和 Grafana）。
> - 重命名集群之后 Grafana 可能会残留一些旧集群名的面板，需要手动删除这些面板。

6.8.7 关闭集群

关闭集群操作会按 Alertmanager -> Grafana -> Prometheus -> TiCDC -> Drainer -> TiFlash -> TiDB -> Pump -> TiKV -> PD 的顺序关闭整个 TiDB 集群所有组件（同时也会关闭监控组件）：

```
tiup cluster stop ${cluster-name}
```

和 start 命令类似，stop 命令也支持通过 -R 和 -N 参数来只停止部分组件。

例如，下列命令只停止 TiDB 组件：

```
tiup cluster stop ${cluster-name} -R tidb
```

下列命令只停止 1.2.3.4 和 1.2.3.5 这两台机器上的 TiDB 组件：

```
tiup cluster stop ${cluster-name} -N 1.2.3.4:4000,1.2.3.5:4000
```

6.8.8 清除集群数据

此操作会关闭所有服务，并清空其数据目录或/和日志目录，并且无法恢复，需要谨慎操作。

清空集群所有服务的数据，但保留日志：

```
tiup cluster clean ${cluster-name} --data
```

清空集群所有服务的日志，但保留数据：

```
tiup cluster clean ${cluster-name} --log
```

清空集群所有服务的数据和日志：

```
tiup cluster clean ${cluster-name} --all
```

清空 Prometheus 以外的所有服务的日志和数据：

```
tiup cluster clean ${cluster-name} --all --ignore-role prometheus
```

清空节点 172.16.13.11:9000 以外的所有服务的日志和数据：

```
tiup cluster clean ${cluster-name} --all --ignore-node 172.16.13.11:9000
```

清空部署在 172.16.13.12 以外的所有服务的日志和数据：

```
tiup cluster clean ${cluster-name} --all --ignore-node 172.16.13.12
```

6.8.9 销毁集群

销毁集群操作会关闭服务，清空数据目录和部署目录，并且无法恢复，需要谨慎操作。

```
tiup cluster destroy ${cluster-name}
```

6.9 TiDB Ansible 常见运维操作

> **警告：**
>
> 对于生产环境，推荐使用 TiUP 进行集群运维操作。自 v4.0 版本起，PingCAP 不再提供 TiDB Ansible 的运维支持（废弃）。继续使用 TiDB Ansible 运维 TiDB 集群可能存在风险，因此不推荐该运维方式。

6.9.1 启动集群

此操作会按顺序启动整个 TiDB 集群所有组件（包括 PD、TiDB、TiKV 等组件和监控组件）。

```
ansible-playbook start.yml
```

6.9.2 关闭集群

此操作会按顺序关闭整个 TiDB 集群所有组件（包括 PD、TiDB、TiKV 等组件和监控组件）。

```
ansible-playbook stop.yml
```

6.9.3 清除集群数据

此操作会关闭 TiDB、Pump、TiKV、PD 服务，并清空 Pump、TiKV、PD 数据目录。

```
ansible-playbook unsafe_cleanup_data.yml
```

6.9.4 销毁集群

此操作会关闭集群，并清空部署目录，若部署目录为挂载点，会报错，可忽略。

```
ansible-playbook unsafe_cleanup.yml
```

6.10 在线修改集群配置

> **警告：**
>
> 该功能目前是实验性阶段，不建议在生产环境中使用。

在线配置变更主要是通过利用 SQL 对包括 TiDB、TiKV 以及 PD 在内的各组件的配置进行在线更新。用户可以通过在线配置变更对各组件进行性能调优而无需重启集群组件。但目前在线修改 TiDB 实例配置的方式和修改其他组件 (TiKV, PD) 的有所不同。

6.10.1 常用操作

6.10.1.1 查看实例配置

可以通过 SQL 语句 show config 来直接查看集群所有实例的配置信息，结果如下：

```
show config;
```

```
+--
    ↪ ----+-----------------+-------------------------------------------------------------+-----------------------
    ↪
| Type | Instance        | Name                                                        | Value
    ↪
    ↪ |
+--
    ↪ ----+-----------------+-------------------------------------------------------------+-----------------------
    ↪
| tidb | 127.0.0.1:4001  | advertise-address                                           | 127.0.0.1
    ↪
    ↪ |
| tidb | 127.0.0.1:4001  | alter-primary-key                                           | false
    ↪
    ↪ |
| tidb | 127.0.0.1:4001  | binlog.binlog-socket                                        |
    ↪
    ↪ |
| tidb | 127.0.0.1:4001  | binlog.enable                                               | false
    ↪
    ↪ |
| tidb | 127.0.0.1:4001  | binlog.ignore-error                                         | false
    ↪
    ↪ |
| tidb | 127.0.0.1:4001  | binlog.strategy                                             | range
    ↪
    ↪ |
| tidb | 127.0.0.1:4001  | binlog.write-timeout                                        | 15s
    ↪
    ↪ |
| tidb | 127.0.0.1:4001  | check-mb4-value-in-utf8                                     | true
    ↪
    ↪ |

...
```

还可以根据对应的字段进行过滤，如：

```
show config where type='tidb'
```

```
show config where instance in (...)
show config where name like '%log%'
show config where type='tikv' and name='log-level'
```

6.10.1.2 在线修改 TiKV 配置

> **注意：**
>
> 在线修改 TiKV 配置项后，同时会自动修改 TiKV 的配置文件。但还需要使用 `tiup edit-config` 命令来修改对应的配置项，否则 `upgrade` 和 `reload` 等运维操作会将在线修改配置后的结果覆盖。修改配置的操作请参考：使用 TiUP 修改配置。执行 `tiup edit-config` 后不需要执行 `tiup reload` 操作。

执行 SQL 语句 `set config`，可以结合实例地址或组件类型来修改单个实例配置或全部实例配置，如：

修改全部 TiKV 实例配置：

> **注意：**
>
> 建议使用反引号包裹变量名称。

```
set config tikv `split.qps-threshold`=1000
```

修改单个 TiKV 实例配置：

```
set config "127.0.0.1:20180" `split.qps-threshold`=1000
```

设置成功会返回 `Query OK`：

```
Query OK, 0 rows affected (0.01 sec)
```

在批量修改时如果有错误发生，会以 warning 的形式返回：

```
set config `tikv log-level`='warn';
```

```
Query OK, 0 rows affected, 1 warning (0.04 sec)
```

```
show warnings;
```

```
+--
    ↪ -------+------+-----------------------------------------------------------------------------------------
    ↪
```

```
| Level   | Code | Message
    ↪
    ↪ |
+--
    ↪ -------+------+-------------------------------------------------------------------------------------------------
    ↪
| Warning | 1105 | bad request to http://127.0.0.1:20180/config: fail to update, error: "config
    ↪ log-level can not be changed" |
+--
    ↪ -------+------+-------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

批量修改配置不保证原子性，可能出现某些实例成功，而某些失败的情况。如使用 `set tikv key=val` 命令修改整个 TiKV 集群配置时，可能有部分实例失败，请执行 `show warnings` 进行查看。

如遇到部分修改失败的情况，需要重新执行对应的修改语句，或通过修改单个实例的方式完成修改。如果因网络或者机器故障等原因无法访问到的 TiKV，需要等到恢复后再次进行修改。

针对 TiKV 可在线修改的配置项，如果成功修改后，修改的结果会被持久化到配置文件中，后续以配置文件中的配置为准。某些配置项名称可能和 TiDB 预留关键字冲突，如 `limit`、`key` 等，对于此类配置项，需要用反引号 ` 包裹起来，如 `` `raftstore.raft-log-gc-size-limit` ``。

支持的配置项列表如下：

配置项	简介
raftstore.sync-log	数据、log 落盘是否同步
raftstore.raft-entry-max-size	单个日志最大大小
raftstore.raft-log-gc-tick-interval	删除 Raft 日志的轮询任务调度间隔时间

配置项	简介
raftstore.raft-log-gc-threshold	允许残余的 Raft 日志个数，软限制
raftstore.raft-log-gc-count-limit	允许残余的 Raft 日志个数，硬限制
raftstore.raft-log-gc-size-limit	允许残余的 Raft 日志大小，硬限制
raftstore.raft-entry-cache-life-time	内存中日志 cache 允许的最长残留时间
raftstore.reject-transfer-leader-duration	控制迁移 leader 到新加节点的最小时间

配置项	简介
raftstore.region-check-tick-interval	检查 Region 是否需要分裂的时间间隔
raftstore.region-split-check-diff	允许 Region 数据超过指定大小的最大值
raftstore.region-compact-check-interval	检查是否需要人工触发 RocksDB compaction 的时间间隔
raftstore.region-compact-check-step	每轮校验人工 compaction 时，一次性检查的 Region 个数

配置项	简介
raftstore.region-compact-min-tombstones	触发 RocksDB compaction 需要的 tombstone 个数
raftstore.region-compact-tombstones-percent	触发 RocksDB compaction 需要的 tombstone 所占比例
raftstore.pd-heartbeat-tick-interval	触发 Region 对 PD 心跳的时间间隔
raftstore.pd-store-heartbeat-tick-interval	触发 store 对 PD 心跳的时间间隔

配置项	简介
raftstore.snap-mgr-gc-tick-interval	触发回收过期 snapshot 文件的时间间隔
raftstore.snap-gc-timeout	snapshot 文件的最长保存时间
raftstore.lock-cf-compact-interval	触发对 lock CF 进行 compact 检查的时间间隔
raftstore.lock-cf-compact-bytes-threshold	触发对 lock CF 进行 compact 的大小
raftstore.messages-per-tick	每轮处理的消息最大个数

配置项	简介
raftstore.max-peer-down-duration	副本允许的最长未响应时间
raftstore.max-leader-missing-duration	允许副本处于无主状态的最长时间，超过将会向 PD 校验自己是否已经被删除
raftstore.abnormal-leader-missing-duration	允许副本处于无主状态的时间，超过将视为异常，标记在 metrics 和日志中

配置项	简介
raftstore.peer-stale-state-check-interval	触发检验副本是否处于无主状态的时间间隔
raftstore.consistency-check-interval	触发一致性检查的时间间隔
raftstore.region-store-max-leader-lease	主可信任期的最长时间
raftstore.allow-remove-leader	允许删除主开关
raftstore.merge-check-tick-interval	触发 Merge 完成检查的时间间隔
raftstore.cleanup-import-sst-interval	触发检查过期 SST 文件的时间间隔

配置项	简介
raftstore.local-read-batch-size	一轮处理读请求的最大个数
raftstore.hibernate-timeout	启动后进入静默状态前需要等待的最短时间，在该时间段内不会进入静默状态（未 release）
coprocessor.split-region-on-table	开启按 table 分裂 Region 的开关
coprocessor.batch-split-limit	批量分裂 Region 的阈值

配置项	简介
coprocessor.region-max-size	Region 容量空间的最大值
coprocessor.region-split-size	分裂后新 Region 的大小
coprocessor.region-max-keys	最多允许的 key 的个数
coprocessor.region-split-keys	分裂后新 Region 的 key 的个数
pessimistic-txn.wait-for-lock-timeout	悲观事务遇到锁后的最长等待时间
pessimistic-txn.wake-up-delay-duration	悲观事务被重新唤醒的时间

配置项	简介
pessimistic-txn.pipelined	是否开启流水线式加悲观锁流程
gc.ratio-threshold	跳过 Region GC 的阈值（GC 版本个数/key 个数）
gc.batch-keys	一轮处理 key 的个数
gc.max-write-bytes-per-sec	一秒可写入 RocksDB 的最大字节数
gc.enable-compaction-filter	是否使用 compaction filter

配置项	简介
gc.compaction.filter-skip-version-check	是否跳过 compaction filter 的集群版本检查（未 release）
{db-name}.max-total-wal-size	WAL 总大小限制
{db-name}.max-background-jobs	RocksDB 后台线程个数
{db-name}.max-open-files	RocksDB 可以打开的文件总数
{db-name}.compaction-readahead-size	Compaction 时候 readahead 的大小
{db-name}.bytes-per-sync	异步同步的限速速率

配置项	简介
{db-name}.wal-bytes-per-sync	WAL 同步的限速速率
{db-name}.writable-file-max-buffer-size	WritableFileWrite 所使用的最大的 buffer 大小
{db-name}.{cf-name}.block-cache-size	block cache 大小
{db-name}.{cf-name}.write-buffer-size	memtable 大小
{db-name}.{cf-name}.max-write-buffer-number	最大 memtable 个数
{db-name}.{cf-name}.max-bytes-for-level-base	base level (L-1) 最大字节数
{db-name}.{cf-name}.target-file-size-base	base level 的目标文件大小

配置项	简介
{db-name}.{cf-name}.level0-file-num-compaction-trigger	触发 compaction 的 L0 文件最大个数
{db-name}.{cf-name}.level0-slowdown-writes-trigger	触发 write stall 的 L0 文件最大个数
{db-name}.{cf-name}.level0-stop-writes-trigger	完全阻停写入的 L0 文件最大个数
{db-name}.{cf-name}.max-compaction-bytes	一次 compaction 最大写入字节数
{db-name}.{cf-name}.max-bytes-for-level-multiplier	每一层的默认放大倍数
{db-name}.{cf-name}.disable-auto-compactions	自动 compaction 的开关

配置项	简介
{db-name}.{cf-name}.soft-pending-compaction-bytes-limit	pending compaction bytes 的软限制
{db-name}.{cf-name}.hard-pending-compaction-bytes-limit	pending compaction bytes 的硬限制
{db-name}.{cf-name}.titan.blob-run-mode	处理 blob 文件的模式
storage.block-cache.capacity	共享 block cache 的大小（自 v4.0.3 起支持）
backup.num-threads	backup 线程的数量（自 v4.0.3 起支持）

配置项	简介
split.qps-threshold	对 Region 执行 load-base-split 的阈值。如果读 QPS 连续 10 秒内均超过这个值，则进行 split
split.split-balance-score	load-base-split 的控制参数，确保 split 后左右访问尽量均匀

配置项	简介
split.split-contained-score	load-base-split 的控制参数，尽量减少 split 后跨 Region 访问
cdc.incremental-scan-concurrency	增量扫描历史数据任务的最大并发执行个数（自 v4.0.15 起支持）
cdc.incremental-scan-speed-limit	增量扫描历史数据的速度上限（自 v4.0.15 起支持）

配置项	简介
cdc.min-ts-interval	定期推进 Resolved TS 的时间间隔（自 v4.0.15 起支持）
cdc.sink-memory-quota	缓存在内存中的 TiCDC 数据变更事件占用内存的上限（自 v4.0.15 起支持）

上述前缀为 {db-name} 或 {db-name}.{cf-name} 的是 RocksDB 相关的配置项。db-name 的取值可为 rocksdb 或 raftdb。

- 当 db-name 为 rocksdb 时，cf-name 的可取值有：defaultcf、writecf、lockcf、raftcf；
- 当 db-name 为 raftdb 时，cf-name 的可取值有：defaultcf。

具体配置项的意义可参考TiKV 配置文件描述

6.10.1.3 在线修改 PD 配置

PD 暂不支持单个实例拥有独立配置。所有实例共享一份配置，可以通过下列方式修改 PD 的配置项：

```
set config pd `log.level`='info'
```

设置成功会返回 Query OK：

```
Query OK, 0 rows affected (0.01 sec)
```

针对 PD 可在线修改的配置项，成功修改后则会持久化到 etcd 中，不会对配置文件进行持久化，后续以 etcd 中的配置为准。同上，若和 TiDB 预留关键字冲突，需要用反引号 ` 包裹此类配置项，例如 `schedule.leader-schedule-limit`。

支持配置项列表如下：

配置项	简介
log.level	日志级别
cluster-version	集群的版本
schedule.max-merge-region-size	控制 Region Merge 的 size 上限（单位是 MB）
schedule.max-merge-region-keys	控制 Region Merge 的 key 数量上限
schedule.patrol-region-interval	控制 replicaChecker 检查 Region 健康状态的运行频率
schedule.split-merge-interval	控制对同一个 Region 做 split 和 merge 操作的间隔
schedule.max-snapshot-count	控制单个 store 最多同时接收或发送的 snapshot 数量
schedule.max-pending-peer-count	控制单个 store 的 pending peer 上限
schedule.max-store-down-time	PD 认为失联 store 无法恢复的时间
schedule.leader-schedule-policy	用于控制 leader 调度的策略
schedule.leader-schedule-limit	可以控制同时进行 leader 调度的任务个数
schedule.region-schedule-limit	可以控制同时进行 Region 调度的任务个数
schedule.replica-schedule-limit	可以控制同时进行 replica 调度的任务个数
schedule.merge-schedule-limit	控制同时进行的 Region Merge 调度的任务
schedule.hot-region-schedule-limit	可以控制同时进行的热点调度的任务个数
schedule.hot-region-cache-hits-threshold	用于设置 Region 被视为热点的阈值
schedule.high-space-ratio	用于设置 store 空间充裕的阈值
schedule.low-space-ratio	用于设置 store 空间不足的阈值
schedule.tolerant-size-ratio	控制 balance 缓冲区大小
schedule.enable-remove-down-replica	用于开启自动删除 DownReplica 的特性
schedule.enable-replace-offline-replica	用于开启迁移 OfflineReplica 的特性
schedule.enable-make-up-replica	用于开启补充副本的特性
schedule.enable-remove-extra-replica	用于开启删除多余副本的特性
schedule.enable-location-replacement	用于开启隔离级别检查
schedule.enable-cross-table-merge	用于开启跨表 Merge
schedule.enable-one-way-merge	用于开启单向 Merge（只允许和下一个相邻的 Region Merge）
replication.max-replicas	用于设置副本的数量
replication.location-labels	用于设置 TiKV 集群的拓扑信息
replication.enable-placement-rules	开启 Placement Rules
replication.strictly-match-label	开启 label 检查
pd-server.use-region-storage	开启独立的 Region 存储
pd-server.max-gap-reset-ts	用于设置最大的重置 timestamp 的间隔（BR）
pd-server.key-type	用于设置集群 key 的类型
pd-server.metric-storage	用于设置集群 metrics 的存储地址
pd-server.dashboard-address	用于设置 dashboard 的地址
replication-mode.replication-mode	备份的模式

具体配置项意义可参考PD 配置文件描述。

6.10.1.4 在线修改 TiDB 配置

在线修改 TiDB 配置的方式和 TiKV/PD 有所不同，用户通过系统变量来完成修改。

下面例子展示了如何通过变量 `tidb_slow_log_threshold` 在线修改配置项 `slow-threshold`。

`slow-threshold` 默认值是 300 毫秒，可以通过设置系统变量 `tidb_slow_log_threshold` 将其修改为 200 毫秒：

```
set tidb_slow_log_threshold = 200;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
select @@tidb_slow_log_threshold;
```

```
+----------------------------+
| @@tidb_slow_log_threshold |
+----------------------------+
| 200                        |
+----------------------------+
1 row in set (0.00 sec)
```

支持在线修改的配置项和相应的 TiDB 系统变量如下：

配置项	对应变量	简介
mem-quota-query	tidb_mem_quota_query	查询语句的内存使用限制
log.enable-slow-log	tidb_enable_slow_log	慢日志的开关
log.slow-threshold	tidb_slow_log_threshold	慢日志阈值
log.expensive-threshold	tidb_expensive_query_time_threshold	expensive 查询阈值

7 监控与告警

7.1 TiDB 监控框架概述

TiDB 使用开源时序数据库 Prometheus 作为监控和性能指标信息存储方案，使用 Grafana 作为可视化组件进行展示。

7.1.1 Prometheus 在 TiDB 中的应用

Prometheus 是一个拥有多维度数据模型的、灵活的查询语句的时序数据库。Prometheus 作为热门的开源项目，拥有活跃的社区及众多的成功案例。

Prometheus 提供了多个组件供用户使用。目前，TiDB 使用了以下组件：

- Prometheus Server：用于收集和存储时间序列数据。
- Client 代码库：用于定制程序中需要的 Metric。
- Alertmanager：用于实现报警机制。

其结构如下图所示：

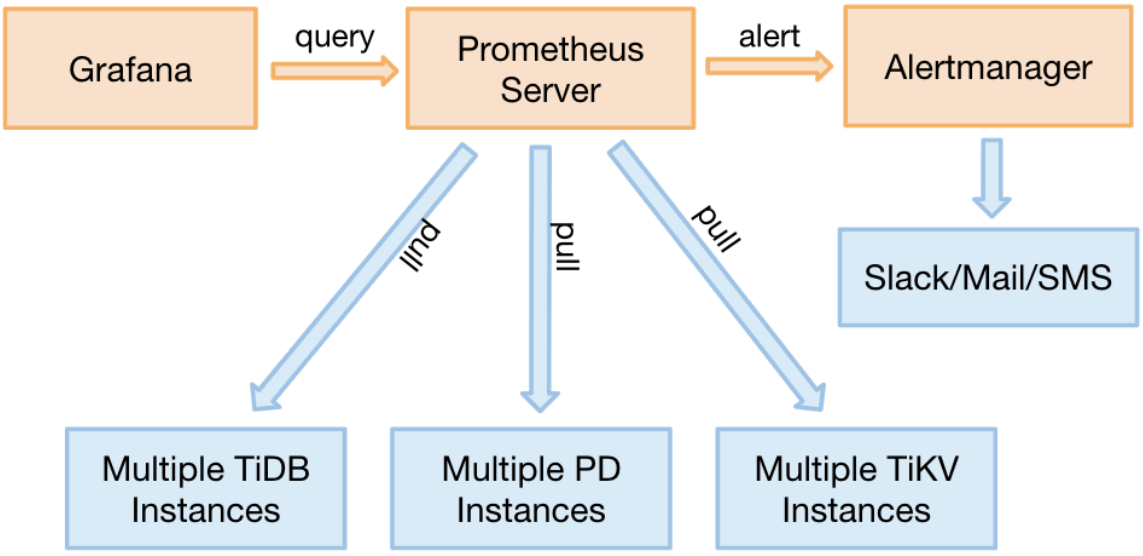


图 59: Prometheus in TiDB

7.1.2 Grafana 在 TiDB 中的应用

Grafana 是一个开源的 metric 分析及可视化系统。TiDB 使用 Grafana 来展示 TiDB 集群各组件的相关监控，监控项分组如下图所示：

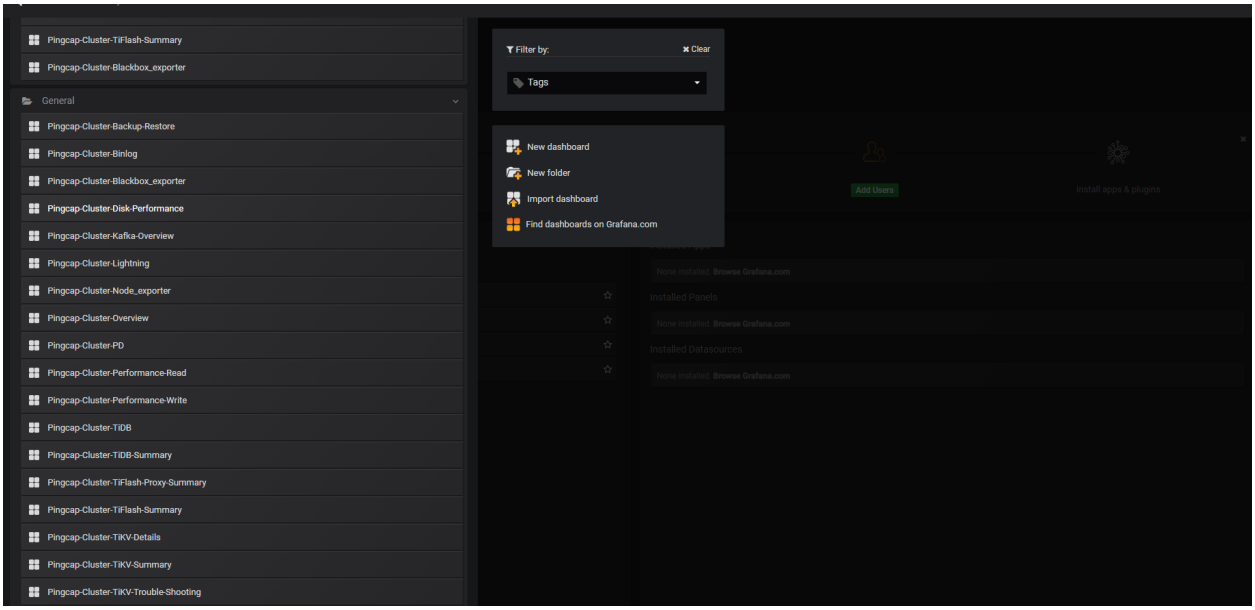


图 60: Grafana monitored_groups

- {TiDB_Cluster_name}-Backup-Restore：备份恢复相关的监控项。
- {TiDB_Cluster_name}-Binlog：TiDB Binlog 相关的监控项。
- {TiDB_Cluster_name}-Blackbox_exporter：网络探活相关监控项。
- {TiDB_Cluster_name}-Disk-Performance：磁盘性能相关监控项。
- {TiDB_Cluster_name}-Kafka-Overview：Kafka 相关监控项。
- {TiDB_Cluster_name}-Lightning：TiDB Lightning 组件相关监控项。
- {TiDB_Cluster_name}-Node_exporter：操作系统相关监控项。
- {TiDB_Cluster_name}-Overview：重要组件监控概览。
- {TiDB_Cluster_name}-PD：PD server 组件相关监控项。
- {TiDB_Cluster_name}-Performance-Read：读性能相关监控项。
- {TiDB_Cluster_name}-Performance-Write：写性能相关监控项。
- {TiDB_Cluster_name}-TiDB：TiDB server 组件详细监控项。
- {TiDB_Cluster_name}-TiDB-Summary：TiDB server 相关监控项概览。
- {TiDB_Cluster_name}-TiFlash-Proxy-Summary：数据同步到 TiFlash 的代理 server 监控项概览。
- {TiDB_Cluster_name}-TiFlash-Summary：TiFlash server 相关监控项概览。
- {TiDB_Cluster_name}-TiKV-Details：TiKV server 组件详细监控项。
- {TiDB_Cluster_name}-TiKV-Summary：TiKV server 监控项概览。
- {TiDB_Cluster_name}-TiKV-Trouble-Shooting：TiKV 错误诊断相关监控项。
- {TiDB_Cluster_name}-TiCDC：TiCDC 组件详细监控项。

每个分组包含多个监控项页签，页签中包含多个详细的监控项信息。以 Overview 监控组为例，其中包含 5 个页签，每个页签内有相应的监控指标看板，如下图所示：

图 61: Grafana Overview

7.2 TiDB 集群监控 API

TiDB 提供了以下两种接口来监控集群状态：

- 状态接口：通过 HTTP 接口对外汇报组件的信息。
- Metrics 接口：使用 Prometheus 记录组件中各种操作的详细信息，使用 Grafana 进行可视化展示。

7.2.1 使用状态接口

状态接口用于监控组件的一些基本信息，并且可以作为 keepalive 的监测接口。另外，通过 PD 的状态接口可以看到整个 TiKV 集群的详细信息。

7.2.1.1 TiDB Server

- TiDB API 地址：`http://${host}:${port}`

- 默认端口：10080

以下示例中，通过访问 http://${host}:${port}/status 获取当前 TiDB Server 的状态，并判断该 TiDB Server 是否存活。结果以 JSON 格式返回：

```
curl http://127.0.0.1:10080/status
```

```
{
    connections: 0,  # 当前 TiDB Server 上的客户端连接数
    version: "5.7.25-TiDB-v4.0.0-rc-141-g7267747ae",  # TiDB 版本号
    git_hash: "7267747ae0ec624dffc3fdedb00f1ed36e10284b"  # TiDB 当前代码的 Git Hash
}
```

7.2.1.2 PD Server

- PD API 地址：http://${host}:${port}/pd/api/v1/${api_name}
- 默认端口：2379
- 各类 api_name 详细信息：参见 PD API Doc

通过该接口可以获取当前所有 TiKV 节点的状态以及负载均衡信息。下面以一个单节点的 TiKV 集群为例，说明用户需要了解的信息：

```
curl http://127.0.0.1:2379/pd/api/v1/stores
```

```
{
  "count": 1,  # TiKV 节点数量
  "stores": [  # TiKV 节点的列表
    # 集群中单个 TiKV 节点的信息
    {
      "store": {
        "id": 1,
        "address": "127.0.0.1:20160",
        "version": "4.0.0-rc.2",
        "status_address": "172.16.5.90:20382",
        "git_hash": "2fdb2804bf8ffaab4b18c4996970e19906296497",
        "start_timestamp": 1590029618,
        "deploy_path": "/data2/tidb_test/v4.0.rc.2/tikv-20372/bin",
        "last_heartbeat": 1590030038949235439,
        "state_name": "Up"
      },
      "status": {
        "capacity": "3.581TiB",  # 存储总容量
        "available": "3.552TiB",  # 存储剩余容量
        "used_size": "31.77MiB",
        "leader_count": 174,
```

```
        "leader_weight": 1,
        "leader_score": 174,
        "leader_size": 174,
        "region_count": 531,
        "region_weight": 1,
        "region_score": 531,
        "region_size": 531,
        "start_ts": "2020-05-21T10:53:38+08:00",
        "last_heartbeat_ts": "2020-05-21T11:00:38.949235439+08:00",
        "uptime": "7m0.949235439s"
      }
    }
  ]
```

7.2.2 使用 metrics 接口

Metrics 接口用于监控整个集群的状态和性能。

- 如果使用 TiDB Ansible 部署 TiDB 集群，监控系统（Prometheus 和 Grafana）会同时部署。
- 如果使用其他方式部署 TiDB 集群，在使用 metrics 接口前，需先部署 Prometheus 和 Grafana。

成功部署 Prometheus 和 Grafana 之后，配置 Grafana。

7.3 TiDB 集群监控部署

本文档适用于希望手动部署 TiDB 监控报警系统的用户。TiUP 部署方式，会同时自动部署监控报警系统，无需手动部署。

7.3.1 部署 Prometheus 和 Grafana

假设 TiDB 的拓扑结构如下：

节点	主机 IP	服务
Node1	192.168.199.113	PD1, TiDB, node_export, Prometheus, Grafana
Node2	192.168.199.114	PD2, node_export
Node3	192.168.199.115	PD3, node_export
Node4	192.168.199.116	TiKV1, node_export
Node5	192.168.199.117	TiKV2, node_export
Node6	192.168.199.118	TiKV3, node_export

7.3.1.1 第 1 步：下载二进制包

下载二进制包：

```
wget https://download.pingcap.org/prometheus-2.8.1.linux-amd64.tar.gz
wget https://download.pingcap.org/node_exporter-0.17.0.linux-amd64.tar.gz
wget https://download.pingcap.org/grafana-6.1.6.linux-amd64.tar.gz
```

解压二进制包：

```
tar -xzf prometheus-2.8.1.linux-amd64.tar.gz
tar -xzf node_exporter-0.17.0.linux-amd64.tar.gz
tar -xzf grafana-6.1.6.linux-amd64.tar.gz
```

7.3.1.2 第 2 步：在 Node1，Node2，Node3，Node4 上启动 node_exporter

```
cd node_exporter-0.17.0.linux-amd64
```

启动 node_exporter 服务：

```
./node_exporter --web.listen-address=":9100" \
    --log.level="info" &
```

7.3.1.3 第 3 步：在 Node1 上启动 Prometheus

编辑 Prometheus 的配置文件：

```
cd prometheus-2.8.1.linux-amd64 &&
vi prometheus.yml
```

```
...

global:
  scrape_interval:     15s
  evaluation_interval: 15s
  # scrape_timeout 设置为全局默认值 (10s)
  external_labels:
    cluster: 'test-cluster'
    monitor: "prometheus"

scrape_configs:
  - job_name: 'overwritten-nodes'
    honor_labels: true  # 不要覆盖 job 和实例的 label
    static_configs:
    - targets:
      - '192.168.199.113:9100'
      - '192.168.199.114:9100'
      - '192.168.199.115:9100'
      - '192.168.199.116:9100'
```

```
      - '192.168.199.117:9100'
      - '192.168.199.118:9100'

  - job_name: 'tidb'
    honor_labels: true  # 不要覆盖 job 和实例的 label
    static_configs:
    - targets:
      - '192.168.199.113:10080'

  - job_name: 'pd'
    honor_labels: true  # 不要覆盖 job 和实例的 label
    static_configs:
    - targets:
      - '192.168.199.113:2379'
      - '192.168.199.114:2379'
      - '192.168.199.115:2379'

  - job_name: 'tikv'
    honor_labels: true  # 不要覆盖 job 和实例的 label
    static_configs:
    - targets:
      - '192.168.199.116:20180'
      - '192.168.199.117:20180'
      - '192.168.199.118:20180'

...
```

启动 Prometheus 服务：

```
./prometheus \
    --config.file="./prometheus.yml" \
    --web.listen-address=":9090" \
    --web.external-url="http://192.168.199.113:9090/" \
    --web.enable-admin-api \
    --log.level="info" \
    --storage.tsdb.path="./data.metrics" \
    --storage.tsdb.retention="15d" &
```

7.3.1.4 第 4 步：在 Node1 上启动 Grafana

编辑 Grafana 的配置文件：

```
cd grafana-6.1.6 &&
vi conf/grafana.ini
```

```
...

[paths]
data = ./data
logs = ./data/log
plugins = ./data/plugins
[server]
http_port = 3000
domain = 192.168.199.113
[database]
[session]
[analytics]
check_for_updates = true
[security]
admin_user = admin
admin_password = admin
[snapshots]
[users]
[auth.anonymous]
[auth.basic]
[auth.ldap]
[smtp]
[emails]
[log]
mode = file
[log.console]
[log.file]
level = info
format = text
[log.syslog]
[event_publisher]
[dashboards.json]
enabled = false
path = ./data/dashboards
[metrics]
[grafana_net]
url = https://grafana.net

...
```

启动 Grafana 服务：

```
./bin/grafana-server \
    --config="./conf/grafana.ini" &
```

7.3.2 配置 Grafana

本小节介绍如何配置 Grafana。

7.3.2.1 第 1 步：添加 Prometheus 数据源

1. 登录 Grafana 界面。

    - 默认地址：`http://localhost:3000`
    - 默认账户：admin
    - 默认密码：admin

> **注意：**
>
> Change Password 步骤可以选择 Skip。

2. 点击 Grafana 侧边栏菜单 Configuration 中的 Data Source。
3. 点击 Add data source。
4. 指定数据源的相关信息：

    - 在 Name 处，为数据源指定一个名称。
    - 在 Type 处，选择 Prometheus。
    - 在 URL 处，指定 Prometheus 的 IP 地址。
    - 根据需求指定其它字段。

5. 点击 Add 保存新的数据源。

7.3.2.2 第 2 步：导入 Grafana 面板

执行以下步骤，为 PD Server、 TiKV Server 和 TiDB Server 分别导入 Grafana 面板：

1. 点击侧边栏的 Grafana 图标。
2. 在侧边栏菜单中，依次点击 Dashboards > Import 打开 Import Dashboard 窗口。
3. 点击 Upload .json File 上传对应的 JSON 文件（下载 TiDB Grafana 配置文件。

> **注意：**
>
> TiKV、 PD 和 TiDB 面板对应的 JSON 文件分别为 `tikv_summary.json`，`tikv_details.json`，`tikv_trouble_shooting.json`，`pd.json`，`tidb.json`，`tidb_summary.json`。

4. 点击 Load。
5. 选择一个 Prometheus 数据源。
6. 点击 Import，Prometheus 面板即导入成功。

7.3.3 查看组件 metrics

在顶部菜单中，点击 New dashboard，选择要查看的面板。

图 62: view dashboard

可查看以下集群组件信息：

- TiDB Server:
  - query 处理时间，可以看到延迟和吞吐
  - ddl 过程监控
  - TiKV client 相关的监控
  - PD client 相关的监控
- PD Server:
  - 命令执行的总次数
  - 某个命令执行失败的总次数
  - 某个命令执行成功的耗时统计
  - 某个命令执行失败的耗时统计
  - 某个命令执行完成并返回结果的耗时统计
- TiKV Server:
  - GC 监控
  - 执行 KV 命令的总次数
  - Scheduler 执行命令的耗时统计
  - Raft propose 命令的总次数
  - Raft 执行命令的耗时统计
  - Raft 执行命令失败的总次数
  - Raft 处理 ready 状态的总次数

7.4 将 Grafana 监控数据导出成快照

在故障诊断中，监控数据十分重要。当你请求远程协助时，技术支持人员有时需要查看 Grafana Dashboard 以确认问题所在。MetricsTool 用于将 Grafana Dashboard 的快照导出为本地文件，并将快照可视化。因此，你可以在不泄露 Grafana 服务器上其他敏感信息的前提下，将监控数据以快照形式分享给外部人员，同时也方便外部人员准确识读数据图表。

7.4.1 使用方法

可以通过访问 https://metricstool.pingcap.com/ 来使用 MetricsTool。它主要提供以下三种功能：

- 导出快照：提供一段在浏览器开发者工具上运行的用户脚本。你可以使用这个脚本在任意 Grafana v6.x.x 服务器上下载当前 Dashboard 中所有可见面板的快照。

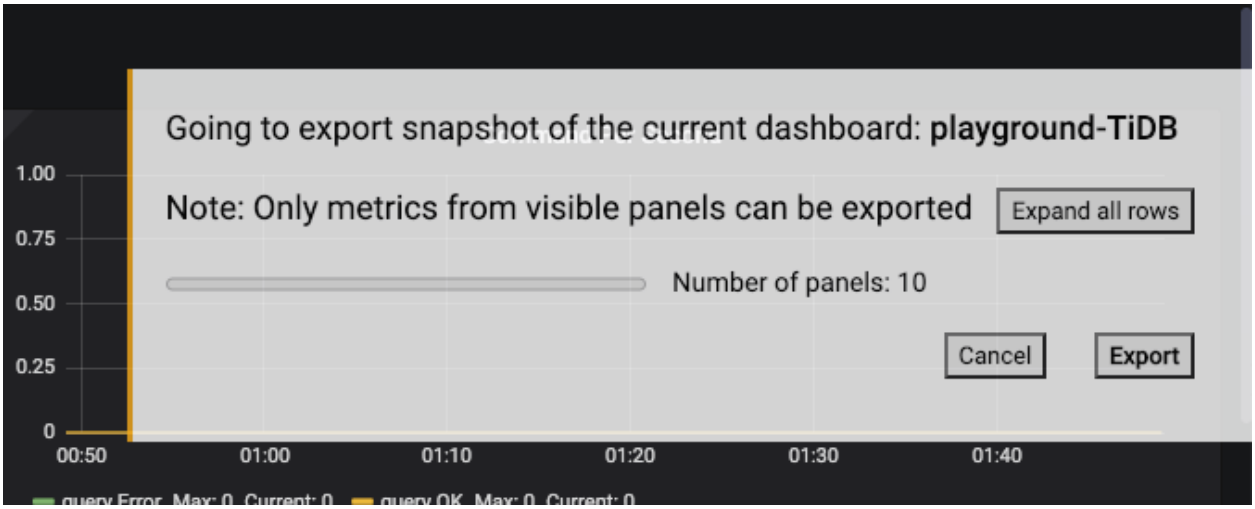


图 63: 运行用户脚本后的 MetricsTool Exporter 截图

- 快照可视化：通过网页端可视化工具 Visualizer 将快照导出文件可视化。快照经过可视化后，操作起来与实际的 Grafana Dashboard 无异。

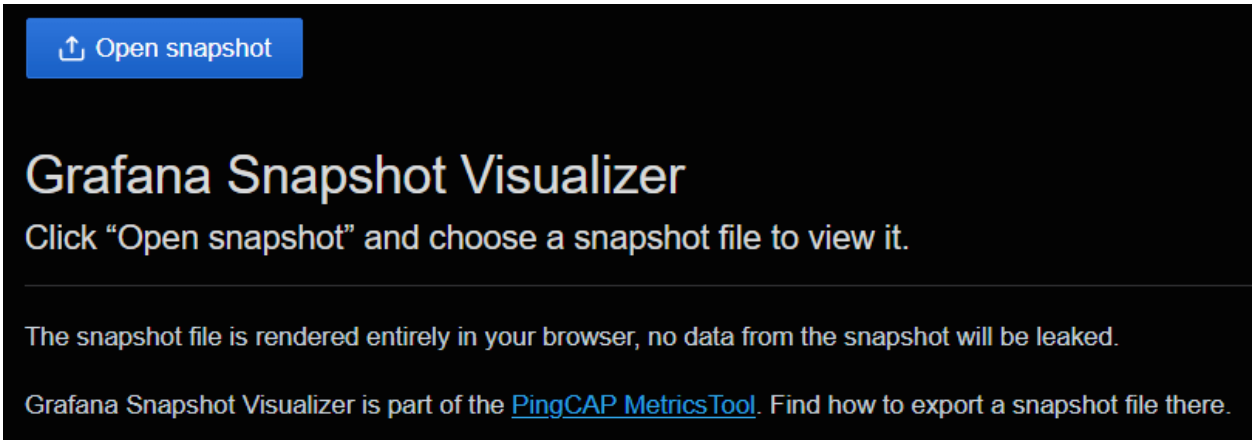


图 64: MetricsTool Visualizer 截图

- 导入快照：介绍如何将导出的快照重新导入到已有的 Grafana 实例中。

7.4.2 FAQs

7.4.2.1 与直接截图及导出 PDF 相比，MetricTool 有什么优势？

MetricsTool 导出的快照文件包含快照生成时的监控指标实际数值。你可以通过 Visualizer 与渲染的图表进行交互，比如切换序列、选择一个较小的时间范围以及检查特定时间点的监控数据值等，就像在操作一个实际的 Grafana Dashboard 一样，因此它比 PDF 文件和截图更强大。

7.4.2.2 快照文件里都包含什么？

快照文件包含所选时间范围内所有图表和面板数据的值，但不保存数据源的原始监控指标，所以无法在 Visualizer 中编辑查询表达式。

7.4.2.3 Visualizer 会将上传的快照文件保存到 PingCAP 的服务器上吗？

不会。快照文件解析全部在浏览器中完成，Visualizer 不会将任何信息发送给 PingCAP。你可以放心地使用 Visualizer 查看带有敏感信息的快照文件，不用担心信息会泄露给第三方。

7.4.2.4 MetricsTool 可以导出除 Grafana 外其他监控工具的数据吗？

不能。目前该工具仅支持在 Grafana v6.x.x 上使用。

7.4.2.5 可以在所有监控指标数据都加载完毕前就运行脚本吗？

可以。虽然脚本会弹出提示，让你等所有监控数据加载完毕后再运行，但可以手动跳过等待并导出快照，以免有些监控数据加载的时间过长。

7.4.2.6 快照文件可视化后，可以通过网页链接分享吗？

不能。但你可以分享快照文件，并说明如何使用 Visualizer 查看。如果确实需要通过网页链接分享，可以尝试使用 Grafana 内置的 `snapshot.raintank.io` 服务。但在这样做之前，要确保不会泄漏隐私信息。

7.5 TiDB 集群报警规则

本文介绍了 TiDB 集群中各组件的报警规则，包括 TiDB、TiKV、PD、TiDB Binlog、Node_exporter 和 Blackbox_exporter 的各报警项的规则描述及处理方法。

按照严重程度由高到低，报警项可分为紧急级别 > 严重级别 > 警告级别三类。该分级适用于以下各组件的报警项。

严重程度	说明
紧急级别	最高严重程度，服务不可用，通常由于服务停止或节点故障导致，此时需要马上进行人工干预
严重级别	服务可用性下降，需要用户密切关注异常指标
警告级别	对某一问题或错误的提醒

7.5.1 TiDB 报警规则

本节介绍了 TiDB 组件的报警项。

7.5.1.1 紧急级别报警项

7.5.1.1.1 TiDB_schema_error

- 报警规则：

  ```
  increase(tidb_session_schema_lease_error_total{type="outdated"}[15m])> 0
  ```

- 规则描述：

  TiDB 在一个 Lease 时间内没有重载到最新的 Schema 信息。如果 TiDB 无法继续对外提供服务，则报警。

- 处理方法：

  该问题通常由于 TiKV Region 不可用或超时导致，需要看 TiKV 的监控指标定位问题。

7.5.1.1.2 TiDB_tikvclient_region_err_total

- 报警规则：

  ```
  increase(tidb_tikvclient_region_err_total[10m])> 6000
  ```

- 规则描述：

  TiDB 访问 TiKV 时发生了 Region 错误。如果在 10 分钟之内该错误多于 6000 次，则报警。

- 处理方法：

  查看 TiKV 的监控状态。

7.5.1.1.3 TiDB_domain_load_schema_total

- 报警规则：

  ```
  increase(tidb_domain_load_schema_total{type="failed"}[10m])> 10
  ```

- 规则描述：

  TiDB 重载最新的 Schema 信息失败的总次数。如果在 10 分钟之内重载失败次数超过 10 次，则报警。

- 处理方法：

  参考TiDB_schema_error 的处理方法。

7.5.1.1.4 TiDB_monitor_keep_alive

- 报警规则：

  ```
  increase(tidb_monitor_keep_alive_total[10m])< 100
  ```

- 规则描述：

  表示 TiDB 的进程是否仍然存在。如果在 10 分钟之内 `tidb_monitor_keep_alive_total` 增加次数少于 100，则 TiDB 的进程可能已经退出，此时会报警。

- 处理方法：

  - 检查 TiDB 进程是否 OOM。
  - 检查机器是否发生了重启。

7.5.1.2 严重级别报警项

7.5.1.2.1 TiDB_server_panic_total

- 报警规则：

  ```
  increase(tidb_server_panic_total[10m])> 0
  ```

- 规则描述：

  发生崩溃的 TiDB 线程的数量。当出现崩溃的时候会报警。该线程通常会被恢复，否则 TiDB 会频繁重启。

- 处理方法：

  收集 panic 日志，定位原因。

7.5.1.3 警告级别报警项

7.5.1.3.1 TiDB_memory_abnormal

- 报警规则：

```
go_memstats_heap_inuse_bytes{job="tidb"} > 1e+10
```

- 规则描述：

  对 TiDB 内存使用量的监控。如果内存使用大于 10 G，则报警。

- 处理方法：

  通过 HTTP API 来排查 goroutine 泄露的问题。

7.5.1.3.2 TiDB_query_duration

- 报警规则：

```
histogram_quantile(0.99, sum(rate(tidb_server_handle_query_duration_seconds_bucket[1m]))BY (
↪ le, instance))> 1
```

- 规则描述：

  TiDB 处理请求的延时。如果.99 的延迟大于 1 秒，则报警。

- 处理方法：

  查看 TiDB 的日志，搜索 SLOW_QUERY 和 TIME_COP_PROCESS 关键字，查找慢 SQL。

7.5.1.3.3 TiDB_server_event_error

- 报警规则：

```
increase(tidb_server_event_total{type=~"server_start|server_hang"}[15m])> 0
```

- 规则描述：

  TiDB 服务中发生的事件数量。当出现以下事件的时候会报警：

  1. start：TiDB 服务启动。
  2. hang：当发生了 Critical 级别的事件时（目前只有 Binlog 写不进去一种情况），TiDB 进入 hang 模式，并等待人工 Kill。

- 处理方法：

  - 重启 TiDB 以恢复服务。
  - 检查 TiDB Binlog 服务是否正常。

7.5.1.3.4 TiDB_tikvclient_backoff_seconds_count

- 报警规则：

  ```
  increase(tidb_tikvclient_backoff_seconds_count[10m])> 10
  ```

- 规则描述：

  TiDB 访问 TiKV 发生错误时发起重试的次数。如果在 10 分钟之内重试次数多于 10 次，则报警。

- 处理方法：

  查看 TiKV 的监控状态。

7.5.1.3.5 TiDB_monitor_time_jump_back_error

- 报警规则：

  ```
  increase(tidb_monitor_time_jump_back_total[10m])> 0
  ```

- 规则描述：

  如果 TiDB 所在机器的时间发生了回退，则报警。

- 处理方法：

  排查 NTP 配置。

7.5.1.3.6 TiDB_ddl_waiting_jobs

- 报警规则：

  ```
  sum(tidb_ddl_waiting_jobs)> 5
  ```

- 规则描述：

  如果 TiDB 中等待执行的 DDL 任务的数量大于 5，则报警。

- 处理方法：

  通过 `admin show ddl` 语句检查是否有耗时的 add index 操作正在执行。

7.5.2 PD 报警规则

本节介绍了 PD 组件的报警项。

7.5.2.1 紧急级别报警项

7.5.2.1.1 PD_cluster_down_store_nums

- 报警规则：

```
(sum(pd_cluster_status{type="store_down_count"})by (instance)> 0)and (sum(etcd_server_is_leader
↪ )by (instance)> 0)
```

- 规则描述：

PD 长时间（默认配置是 30 分钟）没有收到 TiKV/TiFlash 心跳。

- 处理方法：

  - 检查 TiKV/TiFlash 进程是否正常、网络是否隔离以及负载是否过高，并尽可能地恢复服务。
  - 如果确定 TiKV/TiFlash 无法恢复，可做下线处理。

7.5.2.2 严重级别报警项

7.5.2.2.1 PD_etcd_write_disk_latency

- 报警规则：

```
histogram_quantile(0.99, sum(rate(etcd_disk_wal_fsync_duration_seconds_bucket[1m]))by (
↪ instance, job, le))> 1
```

- 规则描述：

fsync 操作延迟大于 1s，代表 etcd 写盘慢，这很容易引起 PD leader 超时或者 TSO 无法及时存盘等问题，从而导致整个集群停止服务。

- 处理方法：

  - 排查写入慢的原因。可能是由于其他服务导致系统负载过高。可以检查 PD 本身是否占用了大量 CPU 或 IO 资源。
  - 可尝试重启 PD 或手动 transfer leader 至其他的 PD 来恢复服务。
  - 如果由于环境原因无法恢复，可将有问题的 PD 下线替换。

7.5.2.2.2 PD_miss_peer_region_count

- 报警规则：

```
(sum(pd_regions_status{type="miss_peer_region_count"})by (instance)> 100)and (sum(etcd_server_is_leader
↪ )by (instance)> 0)
```

- 规则描述：

Region 的副本数小于 `max-replicas` 配置的值。这通常是由于 TiKV 宕机等问题导致一段时间内一些 Region 缺副本。

- 处理方法：

  - 查看是否有 TiKV 宕机或在做下线操作，尝试定位问题产生的原因。
  - 观察 region health 面板，查看 `miss_peer_region_count` 是否在不断减少。

7.5.2.3 警告级别报警项

7.5.2.3.1 PD_cluster_lost_connect_store_nums

- 报警规则：

```
(sum(pd_cluster_status{type="store_disconnected_count"})by (instance)> 0)and (sum(etcd_server_is_leader
↪ )by (instance)> 0)
```

- 规则描述：

  PD 在 20 秒之内未收到 TiKV/TiFlash 上报心跳。正常情况下是每 10 秒收到 1 次心跳。

- 处理方法：

  - 排查是否在重启 TiKV/TiFlash。
  - 检查 TiKV/TiFlash 进程是否正常、网络是否隔离以及负载是否过高，并尽可能地恢复服务。
  - 如果确定 TiKV/TiFlash 无法恢复，可做下线处理。
  - 如果确定 TiKV/TiFlash 可以恢复，但在短时间内还无法恢复，可以考虑延长 `max-down-time` 配置，防止超时后 TiKV/TiFlash 被判定为无法恢复并开始搬移数据。

7.5.2.3.2 PD_cluster_low_space

- 报警规则：

```
(sum(pd_cluster_status{type="store_low_space_count"})by (instance)> 0)and (sum(etcd_server_is_leader
↪ )by (instance)> 0)
```

- 规则描述：

  表示 TiKV/TiFlash 节点空间不足。

- 处理方法：

  - 检查集群中的空间是否普遍不足。如果是，则需要扩容。
  - 检查 Region balance 调度是否有问题。如果有问题，会导致数据分布不均衡。
  - 检查是否有文件占用了大量磁盘空间，比如日志、快照、core dump 等文件。
  - 降低该节点的 Region weight 来减少数据量。
  - 无法释放空间时，可以考虑主动下线该节点，防止由于磁盘空间不足而宕机。

7.5.2.3.3 PD_etcd_network_peer_latency

- 报警规则：

```
histogram_quantile(0.99, sum(rate(etcd_network_peer_round_trip_time_seconds_bucket[1m]))by (
↪ To, instance, job, le))> 1
```

- 规则描述：

  PD 节点之间网络延迟高，严重情况下会导致 leader 超时和 TSO 存盘超时，从而影响集群服务。

- 处理方法：

  - 检查网络状况和系统负载情况。
  - 如果由于环境原因无法恢复，可将有问题的 PD 下线替换。

7.5.2.3.4 PD_tidb_handle_requests_duration

- 报警规则：

```
histogram_quantile(0.99, sum(rate(pd_client_request_handle_requests_duration_seconds_bucket{
↪ type="tso"}[1m]))by (instance, job, le))> 0.1
```

- 规则描述：

  PD 处理 TSO 请求耗时过长，一般是由于负载过高。

- 处理方法：

  - 检查服务器负载状况。
  - 使用 pprof 抓取 PD 的 CPU profile 进行分析。
  - 手动切换 PD leader。
  - 如果是环境问题，则将有问题的 PD 下线替换。

7.5.2.3.5 PD_down_peer_region_nums

- 报警规则：

```
(sum(pd_regions_status{type="down-peer-region-count"})by (instance)> 0)and (sum(etcd_server_is_leader
↪ )by (instance)> 0)
```

- 规则描述：

  Raft leader 上报有不响应 peer 的 Region 数量。

- 处理方法：

  - 检查是否有 TiKV 宕机，或刚发生重启，或者繁忙。
  - 观察 region health 面板，检查 down_peer_region_count 是否在不断减少。
  - 检查是否有 TiKV 之间网络不通。

7.5.2.3.6 PD_pending_peer_region_count

- 报警规则：

```
(sum(pd_regions_status{type="pending-peer-region-count"})by (instance)> 100)and (sum(etcd_server_is_leader
↪ )by (instance)> 0)
```

- 规则描述：

  Raft log 落后的 Region 过多。由于调度产生少量的 pending peer 是正常的，但是如果持续很高，就可能有问题。

- 处理方法：

  - 观察 region health 面板，检查 pending_peer_region_count 是否在不断减少。
  - 检查 TiKV 之间的网络状况，特别是带宽是否足够。

7.5.2.3.7 PD_leader_change

- 报警规则：

```
count(changes(pd_tso_events{type="save"}[10m])> 0)>= 2
```

- 规则描述：

  近期发生了 PD leader 切换。

- 处理方法：

  - 排除人为因素，比如重启 PD、手动 transfer leader 或调整 leader 优先级等。
  - 检查网络状况和系统负载情况。
  - 如果由于环境原因无法恢复，可将有问题的 PD 下线替换。

7.5.2.3.8 TiKV_space_used_more_than_80%

- 报警规则：

```
sum(pd_cluster_status{type="storage_size"})/ sum(pd_cluster_status{type="storage_capacity"})
↪ * 100 > 80
```

- 规则描述：

  集群空间占用超过 80%。

- 处理方法：

  - 确认是否需要扩容。
  - 排查是否有文件占用了大量磁盘空间，比如日志、快照或 core dump 等文件。

7.5.2.3.9 PD_system_time_slow

- 报警规则：

```
changes(pd_tso_events{type="system_time_slow"}[10m])>= 1
```

- 规则描述：

  系统时间可能发生回退。

- 处理方法：

  检查系统时间设置是否正确。

7.5.2.3.10 PD_no_store_for_making_replica

- 报警规则：

```
increase(pd_checker_event_count{type="replica_checker", name="no_target_store"}[1m])> 0
```

- 规则描述：

  没有合适的 store 用来补副本。

- 处理方法：
  - 检查 store 是否空间不足。
  - 根据 label 配置（如果有这个配置的话）来检查是否有可以补副本的 store。

7.5.3 TiKV 报警规则

本节介绍了 TiKV 组件的报警项。

7.5.3.1 紧急级别报警项

7.5.3.1.1 TiKV_memory_used_too_fast

- 报警规则：

```
process_resident_memory_bytes{job=~"tikv",instance=~".*"} - (process_resident_memory_bytes{
↪ job=~"tikv",instance=~".*"} offset 5m)> 5*1024*1024*1024
```

- 规则描述：

  目前没有和内存相关的 TiKV 的监控，你可以通过 Node_exporter 监控集群内机器的内存使用情况。如上规则表示，如果在 5 分钟之内内存使用超过 5GB（TiKV 内存占用的太快），则报警。

- 处理方法：

  调整 `rockdb.defaultcf` 和 `rocksdb.writecf` 的 `block-cache-size` 的大小。

7.5.3.1.2 TiKV_GC_can_not_work

- 报警规则：

```
sum(increase(tikv_gcworker_gc_tasks_vec{task="gc"}[1d]))< 1
```

> 注意：
> 由于 3.0 中引入了分布式 GC 且 GC 不会在 TiDB 执行，因此 `tidb_tikvclient_gc_action_result` 指标虽然在 3.* 以上版本中存在，但是不会有值。

- 规则描述：

  在 24 小时内一个 TiKV 实例上没有成功执行 GC，说明 GC 不能正常工作了。短期内 GC 不运行不会造成太大的影响，但如果 GC 一直不运行，版本会越来越多，从而导致查询变慢。

- 处理方法：

  1. 执行 `select VARIABLE_VALUE from mysql.tidb where VARIABLE_NAME="tikv_gc_leader_desc"` 来找到 gc leader 对应的 `tidb-server`；
  2. 查看该 `tidb-server` 的日志，grep gc_worker tidb.log；
  3. 如果发现这段时间一直在 resolve locks（最后一条日志是 `start resolve locks`）或者 delete ranges（最后一条日志是 `start delete {number} ranges`），说明 GC 进程是正常的。否则需要报备开发人员 support@pingcap.com 进行处理。

7.5.3.2 严重级别报警项

7.5.3.2.1 TiKV_server_report_failure_msg_total

- 报警规则：

```
sum(rate(tikv_server_report_failure_msg_total{type="unreachable"}[10m]))BY (store_id)> 10
```

- 规则描述：

  表明无法连接远端的 TiKV。

- 处理方法：

  1. 检查网络是否通畅。
  2. 检查远端 TiKV 是否挂掉。
  3. 如果远端 TiKV 没有挂掉，检查压力是否太大，参考TiKV_channel_full_total 处理方法。

7.5.3.2.2 TiKV_channel_full_total

- 报警规则：

```
sum(rate(tikv_channel_full_total[10m]))BY (type, instance)> 0
```

- 规则描述：

  该错误通常是因为 Raftstore 线程卡死，TiKV 的压力已经非常大了。

- 处理方法：

  1. 观察 Raft Propose 监控，看这个报警的 TiKV 节点是否明显有比其他 TiKV 高很多。如果是，表明这个 TiKV 上有热点，需要检查热点调度是否能正常工作。
  2. 观察 Raft IO 监控，看延迟是否升高。如果延迟很高，表明磁盘可能有瓶颈。一个能缓解但不怎么安全的办法是将 `sync-log` 改成 `false`。
  3. 观察 Raft Process 监控，看 tick duration 是否很高。如果是，需要在 `[raftstore]` 配置下加上 `raft-base-tick-interval = “2s”`。

7.5.3.2.3 TiKV_write_stall

- 报警规则：

```
delta(tikv_engine_write_stall[10m])> 0
```

- 规则描述：

  RocksDB 写入压力太大，出现了 stall。

- 处理方法：

  1. 观察磁盘监控，排除磁盘问题。
  2. 看 TiKV 是否有写入热点。
  3. 在 `[rocksdb]` 和 `[raftdb]` 配置下调大 `max-sub-compactions` 的值。

7.5.3.2.4 TiKV_raft_log_lag

- 报警规则：

```
histogram_quantile(0.99, sum(rate(tikv_raftstore_log_lag_bucket[1m]))by (le, instance))>
↪ 5000
```

- 规则描述：

  这个值偏大，表明 Follower 已经远远落后于 Leader，Raft 没法正常同步了。可能的原因是 Follower 所在的 TiKV 卡住或者挂掉了。

7.5.3.2.5 TiKV_async_request_snapshot_duration_seconds

- 报警规则：

```
histogram_quantile(0.99, sum(rate(tikv_storage_engine_async_request_duration_seconds_bucket{
↪ type="snapshot"}[1m]))by (le, instance, type))> 1
```

- 规则描述：

  这个值偏大，表明 Raftstore 负载压力很大，可能已经卡住。

- 处理方法：

  参考TiKV_channel_full_total 的处理方法。

7.5.3.2.6 TiKV_async_request_write_duration_seconds

- 报警规则：

```
histogram_quantile(0.99, sum(rate(tikv_storage_engine_async_request_duration_seconds_bucket{
↪ type="write"}[1m]))by (le, instance, type))> 1
```

- 规则描述：

  这个值偏大，表明 Raft write 耗时很长。

- 处理方法：

  1. 检查 Raftstore 上的压力，参考TiKV_channel_full_total 的处理方法。
  2. 检查 apply worker 线程的压力。

7.5.3.2.7 TiKV_coprocessor_request_wait_seconds

- 报警规则：

```
histogram_quantile(0.9999, sum(rate(tikv_coprocessor_request_wait_seconds_bucket[1m]))by (le
↪ , instance, req))> 10
```

- 规则描述：

  这个值偏大，表明 Coprocessor worker 压力很大。可能有比较慢的任务卡住了 Coprocessor 线程。

- 处理方法：

    1. 从 TiDB 日志中查看慢查询日志，看查询是否用到了索引或全表扫，或者看是否需要做 analyze。
    2. 排查是否有热点。
    3. 查看 Coprocessor 监控，看 `coporcessor table/index scan` 里 `total` 和 `process` 是否匹配。如果相差太大，表明做了太多的无效查询。看是否有 `over seek bound`，如果有，表明版本太多，GC 工作不及时，需要增大并行 GC 的线程数。

7.5.3.2.8 TiKV_raftstore_thread_cpu_seconds_total

- 报警规则：

```
sum(rate(tikv_thread_cpu_seconds_total{name=~"raftstore_.*"}[1m]))by (instance, name)> 1.6
```

- 规则描述：

    Raftstore 线程压力太大。

- 处理方法：

    参考TiKV_channel_full_total 的处理方法。

7.5.3.2.9 TiKV_raft_append_log_duration_secs

- 报警规则：

```
histogram_quantile(0.99, sum(rate(tikv_raftstore_append_log_duration_seconds_bucket[1m]))by
↪ (le, instance))> 1
```

- 规则描述：

    表示 append Raft log 的耗时，如果高，通常是因为 IO 太忙了。

7.5.3.2.10 TiKV_raft_apply_log_duration_secs

- 报警规则：

```
histogram_quantile(0.99, sum(rate(tikv_raftstore_apply_log_duration_seconds_bucket[1m]))by (
↪ le, instance))> 1
```

- 规则描述：

    表示 apply Raft log 耗时，如果高，通常是因为 IO 太忙了。

7.5.3.2.11 TiKV_scheduler_latch_wait_duration_seconds

- 报警规则：

```
histogram_quantile(0.99, sum(rate(tikv_scheduler_latch_wait_duration_seconds_bucket[1m]))by
↪ (le, instance, type))> 1
```

- 规则描述：

  Scheduler 中写操作获取内存锁时的等待时间。如果这个值高，表明写操作冲突较多，也可能是某些引起冲突的操作耗时较长，阻塞了其它等待相同锁的操作。

- 处理方法：

  1. 查看 Scheduler-All 监控中的 scheduler command duration，看哪一个命令耗时最大。
  2. 查看 Scheduler-All 监控中的 scheduler scan details，看 `total` 和 `process` 是否匹配。如果相差太大，表明有很多无效的扫描，另外观察是否有 `over seek bound`，如果太多，表明 GC 不及时。
  3. 查看 Storage 监控中的 storage async snapshot/write duration，看是否 Raft 操作不及时。

7.5.3.2.12 `TiKV_thread_apply_worker_cpu_seconds`

- 报警规则：

```
sum(rate(tikv_thread_cpu_seconds_total{name="apply_worker"}[1m]))by (instance)> 1.8
```

- 规则描述：

  Apply Raft log 线程压力太大，通常是因为写入太猛了。

7.5.3.2.13 `TiDB_tikvclient_gc_action_fail`（基本不发生，只在特殊配置下才会发生）

- 报警规则：

```
sum(increase(tidb_tikvclient_gc_action_result{type="fail"}[1m]))> 10
```

> 注意：
> 由于 3.0 中引入了分布式 GC 且 GC 不会在 TiDB 执行，因此 `tidb_tikvclient_gc_action_result` 指标虽然在 3.* 以上版本中存在，但是不会有值。

- 规则描述：

  GC 失败的 Region 较多。

- 处理方法：

  1. 一般是因为并行 GC 开的太高了，可以适当降低 GC 并行度。你需要先确认 GC 失败是由于服务器繁忙导致的。
  2. 通过执行 `update set VARIABLE_VALUE="{number}" where VARIABLE_NAME="tikv_gc_concurrency"` 适当降低并行度。

7.5.3.3 警告级别报警项

7.5.3.3.1 TiKV_leader_drops

- 报警规则：

```
delta(tikv_pd_heartbeat_tick_total{type="leader"}[30s])< -10
```

- 规则描述：

  该问题通常是因为 Raftstore 线程卡住了。

- 处理方法：

  1. 参考TiKV_channel_full_total 的处理方法。
  2. 如果 TiKV 压力很小，考虑 PD 的调度是否太频繁。可以查看 PD 页面的 Operator Create 面板，排查 PD 产生调度的类型和数量。

7.5.3.3.2 TiKV_raft_process_ready_duration_secs

- 报警规则：

```
histogram_quantile(0.999, sum(rate(tikv_raftstore_raft_process_duration_secs_bucket{type='
↪ ready'}[1m]))by (le, instance, type))> 2
```

- 规则描述：

  表示处理 Raft ready 的耗时。这个值大，通常是因为 append log 任务卡住了。

7.5.3.3.3 TiKV_raft_process_tick_duration_secs

- 报警规则：

```
histogram_quantile(0.999, sum(rate(tikv_raftstore_raft_process_duration_secs_bucket{type='
↪ tick' }[1m]))by (le, instance, type))> 2
```

- 规则描述：

  表示处理 Raft tick 的耗时，这个值大，通常是因为 Region 太多导致的。

- 处理方法：

  1. 考虑使用更高等级的日志，比如 warn 或者 error。
  2. 在 [raftstore] 配置下添加 raft-base-tick-interval = “2s”。

7.5.3.3.4 TiKV_scheduler_context_total

- 报警规则：

```
abs(delta( tikv_scheduler_context_total[5m]))> 1000
```

- 规则描述：

  Scheduler 正在执行的写命令数量。这个值高，表示任务完成得不及时。

- 处理方法：

  参考TiKV_scheduler_latch_wait_duration_seconds 的处理方法。

7.5.3.3.5 TiKV_scheduler_command_duration_seconds

- 报警规则：

```
histogram_quantile(0.99, sum(rate(tikv_scheduler_command_duration_seconds_bucket[1m]))by (le
↪ , instance, type)/ 1000)> 1
```

- 规则描述：

  表明 Scheduler 执行命令的耗时。

- 处理方法：

  参考TiKV_scheduler_latch_wait_duration_seconds 的处理方法。

7.5.3.3.6 TiKV_coprocessor_outdated_request_wait_seconds

- 报警规则：

```
delta(tikv_coprocessor_outdated_request_wait_seconds_count[10m])> 0
```

- 规则描述：

  Coprocessor 已经过期的请求等待的时间。这个值高，表示 Coprocessor 压力已经非常大了。

- 处理方法：

  参考TiKV_coprocessor_request_wait_seconds 的处理方法。

7.5.3.3.7 TiKV_coprocessor_request_error

- 报警规则：

```
increase(tikv_coprocessor_request_error{reason=!"meet_lock"}[10m])> 100
```

- 规则描述：

  Coprocessor 的请求错误。

- 处理方法：

  Coprocessor 错误的主要原因分为 “lock”、“outdated” 和 “full” 等。“outdated” 表示请求超时，很可能是由于排队时间过久，或者单个请求的耗时比较长。“full” 表示 Coprocessor 的请求队列已经满了，可能是正在执行的请求比较耗时，导致新来的请求都在排队。耗时比较长的查询需要查看下对应的执行计划是否正确。

7.5.3.3.8 TiKV_coprocessor_request_lock_error

- 报警规则：

```
increase(tikv_coprocessor_request_error{reason="meet_lock"}[10m])> 10000
```

- 规则描述：

  Coprocessor 请求锁的错误。

- 处理方法：

  Coprocessor 错误的主要原因分为“lock”、“outdated”、“full”等。“lock”表示读到的数据正在写入，需要等待一会再读（TiDB 内部会自动重试）。少量这种错误不用关注，如果有大量这种错误，需要查看写入和查询是否有冲突。

7.5.3.3.9 TiKV_coprocessor_pending_request

- 报警规则：

  ```
  delta(tikv_coprocessor_pending_request[10m])> 5000
  ```

- 规则描述：

  Coprocessor 排队的请求。

- 处理方法：

  参考TiKV_coprocessor_request_wait_seconds 的处理方法。

7.5.3.3.10 TiKV_batch_request_snapshot_nums

- 报警规则：

  ```
  sum(rate(tikv_thread_cpu_seconds_total{name=~"cop_.*"}[1m]))by (instance)/ (count(tikv_thread_cpu_seconds_t
  ↪ {name=~"cop_.*"})* 0.9)/ count(count(tikv_thread_cpu_seconds_total)by (instance))> 0
  ```

- 规则描述：

  某个 TiKV 的 Coprocessor CPU 使用率超过了 90%。

7.5.3.3.11 TiKV_pending_task

- 报警规则：

  ```
  sum(tikv_worker_pending_task_total)BY (instance,name)> 1000
  ```

- 规则描述：

  TiKV 等待的任务数量。

- 处理方法：

  查看是哪一类任务的值偏高，通常 Coprocessor、apply worker 这类任务都可以在其他指标里找到解决办法。

7.5.3.3.12 TiKV_low_space

- 报警规则：

  ```
  sum(tikv_store_size_bytes{type="available"})by (instance)/ sum(tikv_store_size_bytes{type="
  ↪ capacity"})by (instance)< 0.2
  ```

- 规则描述：

  TiKV 数据量超过节点配置容量或物理磁盘容量的 80%。

- 处理方法：

  确认节点空间均衡情况，做好扩容计划。

7.5.3.3.13 TiKV_approximate_region_size

- 报警规则：

```
histogram_quantile(0.99, sum(rate(tikv_raftstore_region_size_bucket[1m]))by (le))> 1073741824
```

- 规则描述：

  TiKV split checker 扫描到的最大的 Region approximate size 在 1 分钟内持续大于 1 GB。

- 处理方法：

  Region 分裂的速度不及写入的速度。为缓解这种情况，建议更新到支持 batch-split 的版本 (>= 2.1.0-rc1)。如暂时无法更新，可以使用 `pd-ctl operator add split-region <region_id> --policy=approximate` 手动分裂 Region。

7.5.4 TiDB Binlog 报警规则

关于 TiDB Binlog 报警规则的详细描述，参见TiDB Binlog 集群监控报警文档。

7.5.5 Node_exporter 主机报警规则

本节介绍了 Node_exporter 主机的报警项。

7.5.5.1 紧急级别报警项

7.5.5.1.1 NODE_disk_used_more_than_80%

- 报警规则：

```
node_filesystem_avail_bytes{fstype=~"(ext.|xfs)", mountpoint!~"/boot"} / node_filesystem_size_bytes
{fstype=~"(ext.|xfs)", mountpoint!~"/boot"} * 100 <= 20
```

- 规则描述：

  机器磁盘空间使用率超过 80%。

- 处理方法：

  登录机器，执行 `df -h` 命令，查看磁盘空间使用率，做好扩容计划。

7.5.5.1.2 NODE_disk_inode_more_than_80%

- 报警规则：

```
node_filesystem_files_free{fstype=~"(ext.|xfs)"} / node_filesystem_files{fstype=~"(ext.|xfs)
"} * 100 < 20
```

- 规则描述：

  机器磁盘挂载目录文件系统 inode 使用率超过 80%。

- 处理方法：

  登录机器，执行 `df -i` 命令，查看磁盘挂载目录文件系统 inode 使用率，做好扩容计划。

7.5.5.1.3 NODE_disk_readonly

- 报警规则：

```
node_filesystem_readonly{fstype=~"(ext.|xfs)"} == 1
```

- 规则描述：

  磁盘挂载目录文件系统只读，无法写入数据，一般是因为磁盘故障或文件系统损坏。

- 处理方法：

  - 登录机器创建文件测试是否正常。
  - 检查该服务器硬盘指示灯是否正常，如异常，需更换磁盘并修复该机器文件系统。

7.5.5.2 严重级别报警项

7.5.5.2.1 NODE_memory_used_more_than_80%

- 报警规则：

```
(((node_memory_MemTotal-node_memory_MemFree-node_memory_Cached)/(node_memory_MemTotal)*100))
↪ >= 80
```

- 规则描述：

  机器内存使用率超过 80%。

- 处理方法：

  - 在 Grafana Node Exporter 页面查看该主机的 Memory 面板，检查 Used 是否过高，Available 内存是否过低。
  - 登录机器，执行 `free -m` 命令查看内存使用情况，执行 top 看是否有异常进程的内存使用率过高。

7.5.5.3 警告级别报警项

7.5.5.3.1 NODE_node_overload

- 报警规则：

```
(node_load5 / count without (cpu, mode)(node_cpu{mode="system"}))> 1
```

- 规则描述：

  机器 CPU 负载较高。

- 处理方法：

  - 在 Grafana Node exporter 页面上查看该主机的 CPU Usage 及 Load Average，检查是否过高。
  - 登录机器，执行 top 查看 load average 及 CPU 使用率，看是否是异常进程的 CPU 使用率过高。

7.5.5.3.2 NODE_cpu_used_more_than_80%

- 报警规则：

  ```
  avg(irate(node_cpu_seconds_total{mode="idle"}[5m]))by(instance)* 100 <= 20
  ```

- 规则描述：

  机器 CPU 使用率超过 80%。

- 处理方法：

  - 在 Grafana Node exporter 页面上查看该主机的 CPU Usage 及 Load Average，检查是否过高。
  - 登录机器，执行 top 查看 load average 及 CPU 使用率，看是否是异常进程的 CPU 使用率过高。

7.5.5.3.3 NODE_tcp_estab_num_more_than_50000

- 报警规则：

  ```
  node_netstat_Tcp_CurrEstab > 50000
  ```

- 规则描述：

  机器 `establish` 状态的 TCP 链接超过 50,000。

- 处理方法：

  登录机器执行 `ss -s` 可查看当前系统 `estab` 状态的 TCP 链接数，执行 `netstat` 查看是否有异常链接。

7.5.5.3.4 NODE_disk_read_latency_more_than_32ms

- 报警规则：

  ```
  ((rate(node_disk_read_time_seconds_total{device=~".+"}[5m])/ rate(node_disk_reads_completed_total
  ↪ {device=~".+"}[5m]))or (irate(node_disk_read_time_seconds_total{device=~".+"}[5m])/ irate
  ↪ (node_disk_reads_completed_total{device=~".+"}[5m])))* 1000 > 32
  ```

- 规则描述：

  磁盘读延迟超过 32 毫秒。

- 处理方法：

  - 查看 Grafana Disk Performance Dashboard 观察磁盘使用情况。
  - 查看 Disk Latency 面板观察磁盘的读延迟。
  - 查看 Disk IO Utilization 面板观察 IO 使用率。

7.5.5.3.5 NODE_disk_write_latency_more_than_16ms

- 报警规则：

```
((rate(node_disk_write_time_seconds_total{device=~".+"}[5m])/ rate(node_disk_writes_completed_total
↪ {device=~".+"}[5m]))or (irate(node_disk_write_time_seconds_total{device=~".+"}[5m])/
↪ irate(node_disk_writes_completed_total{device=~".+"}[5m])))> 16
```

- 规则描述：

  机器磁盘写延迟超过 16 毫秒。

- 处理方法：

  - 查看 Grafana Disk Performance Dashboard 观察磁盘使用情况。
  - 查看 Disk Latency 面板可查看磁盘的写延迟。
  - 查看 Disk IO Utilization 面板可查看 IO 使用率。

7.5.6 Blackbox_exporter TCP、ICMP 和 HTTP 报警规则

本节介绍了 Blackbox_exporter TCP、ICMP 和 HTTP 的报警项。

7.5.6.1 紧急级别报警项

7.5.6.1.1 TiDB_server_is_down

- 报警规则：

```
probe_success{group="tidb"} == 0
```

- 规则描述：

  TiDB 服务端口探测失败。

- 处理方法：

  - 检查 TiDB 服务所在机器是否宕机。
  - 检查 TiDB 进程是否存在。
  - 检查监控机与 TiDB 服务所在机器之间网络是否正常。

7.5.6.1.2 Pump_server_is_down

- 报警规则：

```
probe_success{group="pump"} == 0
```

- 规则描述：

  Pump 服务端口探测失败。

- 处理方法：

  - 检查 Pump 服务所在机器是否宕机。
  - 检查 Pump 进程是否存在。
  - 检查监控机与 Pump 服务所在机器之间网络是否正常。

7.5.6.1.3 Drainer_server_is_down

- 报警规则：

  ```
  probe_success{group="drainer"} == 0
  ```

- 规则描述：

  Drainer 服务端口探测失败。

- 处理方法：

  - 检查 Drainer 服务所在机器是否宕机。
  - 检查 Drainer 进程是否存在。
  - 检查监控机与 Drainer 服务所在机器之间网络是否正常。

7.5.6.1.4 TiKV_server_is_down

- 报警规则：

  ```
  probe_success{group="tikv"} == 0
  ```

- 规则描述：

  TiKV 服务端口探测失败。

- 处理方法：

  - 检查 TiKV 服务所在机器是否宕机。
  - 检查 TiKV 进程是否存在。
  - 检查监控机与 TiKV 服务所在机器之间网络是否正常。

7.5.6.1.5 PD_server_is_down

- 报警规则：

  ```
  probe_success{group="pd"} == 0
  ```

- 规则描述：

  PD 服务端口探测失败。

- 处理方法：

  - 检查 PD 服务所在机器是否宕机。
  - 检查 PD 进程是否存在。
  - 检查监控机与 PD 服务所在机器之间网络是否正常。

7.5.6.1.6 Node_exporter_server_is_down

- 报警规则：

  ```
  probe_success{group="node_exporter"} == 0
  ```

- 规则描述：

  Node_exporter 服务端口探测失败。

- 处理方法：

  - 检查 Node_exporter 服务所在机器是否宕机。
  - 检查 Node_exporter 进程是否存在。
  - 检查监控机与 Node_exporter 服务所在机器之间网络是否正常。

7.5.6.1.7 Blackbox_exporter_server_is_down

- 报警规则：

  ```
  probe_success{group="blackbox_exporter"} == 0
  ```

- 规则描述：

  Blackbox_exporter 服务端口探测失败。

- 处理方法：

  - 检查 Blackbox_exporter 服务所在机器是否宕机。
  - 检查 Blackbox_exporter 进程是否存在。
  - 检查监控机与 Blackbox_exporter 服务所在机器之间网络是否正常。

7.5.6.1.8 Grafana_server_is_down

- 报警规则：

  ```
  probe_success{group="grafana"} == 0
  ```

- 规则描述：

  Grafana 服务端口探测失败。

- 处理方法：

  - 检查 Grafana 服务所在机器是否宕机。
  - 检查 Grafana 进程是否存在。
  - 检查监控机与 Grafana 服务所在机器之间网络是否正常。

7.5.6.1.9 Pushgateway_server_is_down

- 报警规则：

  ```
  probe_success{group="pushgateway"} == 0
  ```

- 规则描述：

  Pushgateway 服务端口探测失败。

- 处理方法：

  - 检查 Pushgateway 服务所在机器是否宕机。
  - 检查 Pushgateway 进程是否存在。
  - 检查监控机与 Pushgateway 服务所在机器之间网络是否正常。

7.5.6.1.10 Kafka_exporter_is_down

- 报警规则：

  ```
  probe_success{group="kafka_exporter"} == 0
  ```

- 规则描述：

  Kafka_exporter 服务端口探测失败。

- 处理方法：

  - 检查 Kafka_exporter 服务所在机器是否宕机。
  - 检查 Kafka_exporter 进程是否存在。
  - 检查监控机与 Kafka_exporter 服务所在机器之间网络是否正常。

7.5.6.1.11 Pushgateway_metrics_interface

- 报警规则：

  ```
  probe_success{job="blackbox_exporter_http"} == 0
  ```

- 规则描述：

  Pushgateway 服务 http 接口探测失败。

- 处理方法：

  - 检查 Pushgateway 服务所在机器是否宕机。
  - 检查 Pushgateway 进程是否存在。
  - 检查监控机与 Pushgateway 服务所在机器之间网络是否正常。

7.5.6.2 警告级别报警项

7.5.6.2.1 BLACKER_ping_latency_more_than_1s

- 报警规则：

  ```
  max_over_time(probe_duration_seconds{job=~"blackbox_exporter.*_icmp"}[1m])> 1
  ```

- 规则描述：

  Ping 延迟超过 1 秒。

- 处理方法：

  - 在 Grafana Blackbox Exporter dashboard 上检查两个节点间的 ping 延迟是否太高。
  - 在 Grafana Blackbox Exporter dashboard 的 tcp 面板上检查是否有丢包。

7.6 TiFlash 报警规则

本文介绍了 TiFlash 集群的报警规则。

7.6.1 TiFlash_schema_error

- 报警规则：

  ```
  increase(tiflash_schema_apply_count{type="failed"}[15m])> 0
  ```

- 规则描述：

  出现 schema apply 错误时报警。

- 处理方法：

  可能是逻辑问题，联系 TiFlash 开发人员。

7.6.2 TiFlash_schema_apply_duration

- 报警规则：

  ```
  histogram_quantile(0.99, sum(rate(tiflash_schema_apply_duration_seconds_bucket[1m]))BY (le,
  ↪ instance))> 20
  ```

- 规则描述：

  apply 时间超过 20 秒的概率超过 99% 时报警。

- 处理方法：

  可能是 TiFlash TMT 引擎内部问题，联系 TiFlash 开发人员。

7.6.3 TiFlash_raft_read_index_duration

- 报警规则：

```
histogram_quantile(0.99, sum(rate(tiflash_raft_read_index_duration_seconds_bucket[1m]))BY (
↪ le, instance))> 3
```

- 规则描述：

read index 时间超过 3 秒的概率超过 99% 时报警。

> 注意：
>
> read index 请求是发送给 TiKV leader 的 kvproto 请求，TiKV region 的重试，或 Store 的繁忙/网络问题都可能导致 read index 请求时间过长。

- 处理方法：

可能 TiKV 集群分裂/迁移频繁，导致频繁重试，可以查看 TiKV 集群状态确认。

7.6.4 TiFlash_raft_wait_index_duration

- 报警规则：

```
histogram_quantile(0.99, sum(rate(tiflash_raft_wait_index_duration_seconds_bucket[1m]))BY (
↪ le, instance))> 2
```

- 规则描述：

TiFlash 等待 Region Raft Index 的时间超过 2 秒的概率超过 99% 时报警。

- 处理方法：

可能 TiKV 和 Proxy 的通信出现问题，联系 TiFlash 开发人员确认。

8 故障诊断

8.1 慢查询日志

TiDB 会将执行时间超过slow-threshold（默认值为 300 毫秒）的语句输出到slow-query-file（默认值：“tidb-slow.log”）日志文件中，用于帮助用户定位慢查询语句，分析和解决 SQL 执行的性能问题。

TiDB 默认启用慢查询日志，可以修改配置enable-slow-log 来启用或禁用它。

8.1.1 日志示例

```
## Time: 2019-08-14T09:26:59.487776265+08:00
## Txn_start_ts: 410450924122144769
## User@Host: root[root] @ localhost [127.0.0.1]
## Conn_ID: 3086
```

```
## Exec_retry_time: 5.1 Exec_retry_count: 3
## Query_time: 1.527627037
## Parse_time: 0.000054933
## Compile_time: 0.000129729
## Rewrite_time: 0.000000003 Preproc_subqueries: 2 Preproc_subqueries_time: 0.000000002
## Process_time: 0.07 Request_count: 1 Total_keys: 131073 Process_keys: 131072 Prewrite_time:
    ↪ 0.335415029 Commit_time: 0.032175429 Get_commit_ts_time: 0.000177098
    ↪ Local_latch_wait_time: 0.106869448 Write_keys: 131072 Write_size: 3538944 Prewrite_region
    ↪ : 1
## DB: test
## Is_internal: false
## Digest: 50a2e32d2abbd6c1764b1b7f2058d428ef2712b029282b776beb9506a365c0f1
## Stats: t:pseudo
## Num_cop_tasks: 1
## Cop_proc_avg: 0.07 Cop_proc_p90: 0.07 Cop_proc_max: 0.07 Cop_proc_addr: 172.16.5.87:20171
## Cop_wait_avg: 0 Cop_wait_p90: 0 Cop_wait_max: 0 Cop_wait_addr: 172.16.5.87:20171
## Cop_backoff_regionMiss_total_times: 200 Cop_backoff_regionMiss_total_time: 0.2
    ↪ Cop_backoff_regionMiss_max_time: 0.2 Cop_backoff_regionMiss_max_addr: 127.0.0.1
    ↪ Cop_backoff_regionMiss_avg_time: 0.2 Cop_backoff_regionMiss_p90_time: 0.2
## Cop_backoff_rpcPD_total_times: 200 Cop_backoff_rpcPD_total_time: 0.2
    ↪ Cop_backoff_rpcPD_max_time: 0.2 Cop_backoff_rpcPD_max_addr: 127.0.0.1
    ↪ Cop_backoff_rpcPD_avg_time: 0.2 Cop_backoff_rpcPD_p90_time: 0.2
## Cop_backoff_rpcTiKV_total_times: 200 Cop_backoff_rpcTiKV_total_time: 0.2
    ↪ Cop_backoff_rpcTiKV_max_time: 0.2 Cop_backoff_rpcTiKV_max_addr: 127.0.0.1
    ↪ Cop_backoff_rpcTiKV_avg_time: 0.2 Cop_backoff_rpcTiKV_p90_time: 0.2
## Mem_max: 525211
## Disk_max: 65536
## Prepared: false
## Plan_from_cache: false
## Succ: true
## Plan: tidb_decode_plan('
    ↪ ZJAwCTMyXzcJMAkyMAlkYXRhOlRhYmxlU2Nhbl82CjEJMTBfNgkxAR0AdAEY1Dp0LCByYW5nZTpbLWluZiwraW5mXSwga2VlcCBvcmRlc
    ↪ ==')
use test;
insert into t select * from t;
```

8.1.2 字段含义说明

> **注意：**
>
> 慢查询日志中所有时间相关字段的单位都是“秒”

Slow Query 基础信息：

- Time：表示日志打印时间。
- Query_time：表示执行这个语句花费的时间。
- Parse_time：表示这个语句在语法解析阶段花费的时间。
- Compile_time：表示这个语句在查询优化阶段花费的时间。
- Query：表示 SQL 语句。慢日志里面不会打印 Query，但映射到内存表后，对应的字段叫 Query。
- Digest：表示 SQL 语句的指纹。
- Txn_start_ts：表示事务的开始时间戳，也是事务的唯一 ID，可以用这个值在 TiDB 日志中查找事务相关的其他日志。
- Is_internal：表示是否为 TiDB 内部的 SQL 语句。true 表示 TiDB 系统内部执行的 SQL 语句，false 表示用户执行的 SQL 语句。
- Index_ids：表示语句涉及到的索引的 ID。
- Succ：表示语句是否执行成功。
- Backoff_time：表示语句遇到需要重试的错误时在重试前等待的时间，常见的需要重试的错误有以下几种：遇到了 lock、Region 分裂、tikv server is busy。
- Plan：表示语句的执行计划，用 select tidb_decode_plan('xxx...') SQL 语句可以解析出具体的执行计划。
- Prepared：表示这个语句是否是 Prepare 或 Execute 的请求。
- Plan_from_cache：表示这个语句是否命中了执行计划缓存。
- Rewrite_time：表示这个语句在查询改写阶段花费的时间。
- Preproc_subqueries：表示这个语句中被提前执行的子查询个数，如 where id in (select if from t) 这个子查询就可能被提前执行。
- Preproc_subqueries_time：表示这个语句中被提前执行的子查询耗时。
- Exec_retry_count：表示这个语句执行的重试次数。一般出现在悲观事务中，上锁失败时重试执行该语句。
- Exec_retry_time：表示这个语句的重试执行时间。例如某个查询一共执行了三次（前两次失败），则 Exec_retry_time 表示前两次的执行时间之和，Query_time 减去 Exec_retry_time 则为最后一次执行时间。

和事务执行相关的字段：

- Prewrite_time：表示事务两阶段提交中第一阶段（prewrite 阶段）的耗时。
- Commit_time：表示事务两阶段提交中第二阶段（commit 阶段）的耗时。
- Get_commit_ts_time：表示事务两阶段提交中第二阶段（commit 阶段）获取 commit 时间戳的耗时。
- Local_latch_wait_time：表示事务两阶段提交中第二阶段（commit 阶段）发起前在 TiDB 侧等锁的耗时。
- Write_keys：表示该事务向 TiKV 的 Write CF 写入 Key 的数量。
- Write_size：表示事务提交时写 key 或 value 的总大小。
- Prewrite_region：表示事务两阶段提交中第一阶段（prewrite 阶段）涉及的 TiKV Region 数量。每个 Region 会触发一次远程过程调用。

和内存使用相关的字段：

- Mem_max：表示执行期间 TiDB 使用的最大内存空间，单位为 byte。

和硬盘使用相关的字段：

- Disk_max: 表示执行期间 TiDB 使用的最大硬盘空间，单位为 byte。

和 SQL 执行的用户相关的字段：

- User：表示执行语句的用户名。
- Conn_ID：表示用户的链接 ID，可以用类似 con:3 的关键字在 TiDB 日志中查找该链接相关的其他日志。
- DB：表示执行语句时使用的 database。

和 TiKV Coprocessor Task 相关的字段：

- Request_count：表示这个语句发送的 Coprocessor 请求的数量。
- Total_keys：表示 Coprocessor 扫过的 key 的数量。
- Process_time：执行 SQL 在 TiKV 的处理时间之和，因为数据会并行的发到 TiKV 执行，这个值可能会超过 Query_time。
- Wait_time：表示这个语句在 TiKV 的等待时间之和，因为 TiKV 的 Coprocessor 线程数是有限的，当所有的 Coprocessor 线程都在工作的时候，请求会排队；当队列中有某些请求耗时很长的时候，后面的请求的等待时间都会增加。
- Process_keys：表示 Coprocessor 处理的 key 的数量。相比 total_keys，processed_keys 不包含 MVCC 的旧版本。如果 processed_keys 和 total_keys 相差很大，说明旧版本比较多。
- Cop_proc_avg：cop-task 的平均执行时间。
- Cop_proc_p90：cop-task 的 P90 分位执行时间。
- Cop_proc_max：cop-task 的最大执行时间。
- Cop_proc_addr：执行时间最长的 cop-task 所在地址。
- Cop_wait_avg：cop-task 的平均等待时间。
- Cop_wait_p90：cop-task 的 P90 分位等待时间。
- Cop_wait_max：cop-task 的最大等待时间。
- Cop_wait_addr：等待时间最长的 cop-task 所在地址。
- Cop_backoff_{backoff-type}_total_times：因某种错误造成的 backoff 总次数。
- Cop_backoff_{backoff-type}_total_time：因某种错误造成的 backoff 总时间。
- Cop_backoff_{backoff-type}_max_time：因某种错误造成的最大 backoff 时间。
- Cop_backoff_{backoff-type}_max_addr：因某种错误造成的最大 backoff 时间的 cop-task 地址。
- Cop_backoff_{backoff-type}_avg_time：因某种错误造成的平均 backoff 时间。
- Cop_backoff_{backoff-type}_p90_time：因某种错误造成的 P90 分位 backoff 时间。

8.1.3 相关系统变量

- tidb_slow_log_threshold：设置慢日志的阈值，执行时间超过阈值的 SQL 语句将被记录到慢日志中。默认值是 300 ms。
- tidb_query_log_max_len：设置慢日志记录 SQL 语句的最大长度。默认值是 4096 byte。
- tidb_redact_log：设置慢日志记录 SQL 时是否将用户数据脱敏用 ? 代替。默认值是 0 ，即关闭该功能。
- tidb_enable_collect_execution_info：设置是否记录执行计划中各个算子的物理执行信息，默认值是 1。该功能对性能的影响约为 3%。开启该项后查看 Plan 的示例如下：

```
> select tidb_decode_plan('jAOIMAk1XzE3CTAJMQlmdW5jczpjb3VudChDb2x1bW4jNyktPkMJC/
    ↪ BMNQkxCXRpbWU6MTAuOTMxNTA1bXMsIGxvb3BzOjIJMzcyIEJ5dGVzCU4vQQoxCTMyXzE4CTAJMQlpbmRleDpTdHJlYW1BZ2dfOQkxCXQ
    ↪ ');
```

```
+--
    ↪ ----------------------------------------------------------------------------------------------------
    ↪
| tidb_decode_plan('jAOIMAk1XzE3CTAJMQlmdW5jczpjb3VudChDb2x1bW4jNyktPkMJC/
    ↪ BMNQkxCXRpbWU6MTAuOTMxNTA1bXMsIGxvb3BzOjIJMzcyIEJ5dGVzCU4vQQoxCTMyXzE4CTAJMQlpbmRleDpTdHJlYW1BZ2dfOQkxCXQ
    ↪  |
+----------------------------------------------------------------------------------------------------
    ↪
|     id                    task     estRows               operator info
    ↪                                                    actRows     execution info
    ↪                                                                memory       disk
    ↪                                |
|     StreamAgg_17          root     1                     funcs:count(Column#7)->Column#5
    ↪                                  1          time:10.931505ms, loops:2
    ↪                                                     372 Bytes    N/A
    ↪                                |
|     └─IndexReader_18      root     1                     index:StreamAgg_9
    ↪                                                  1          time:10.927685ms, loops:2, rpc
    ↪ num: 1, rpc time:10.884355ms, proc keys:25007    206 Bytes    N/A
    ↪                                |
|       └─StreamAgg_9       cop      1                     funcs:count(1)->Column#7
    ↪                                          1          time:11ms, loops:25
    ↪                                                     N/A          N/A
    ↪                                |
|         └─IndexScan_16    cop      31281.857819905217    table:t, index:idx(a), range:[-inf
    ↪ ,50000), keep order:false    25007      time:11ms, loops:25
    ↪                                                     N/A          N/A
    ↪                                |
+----------------------------------------------------------------------------------------------------
    ↪
```

在性能测试中可以关闭自动收集算子的执行信息：

```
set @@tidb_enable_collect_execution_info=0;
```

Plan 字段显示的格式和EXPLAIN 或者EXPLAIN ANALYZE 大致一致。可以查看EXPLAIN 或者EXPLAIN ANALYZE 文档了解更多关于执行计划的信息。

更多详细信息，可以参见TiDB 专用系统变量和语法。

8.1.4 慢日志内存映射表

用户可通过查询 INFORMATION_SCHEMA.SLOW_QUERY 表来查询慢查询日志中的内容，表中列名和慢日志中字段名一一对应，表结构可查看SLOW_QUERY 表 中的介绍。

> **注意：**
>
> 每次查询 SLOW_QUERY 表时，TiDB 都会去读取和解析一次当前的慢查询日志。

TiDB 4.0 中，SLOW_QUERY 已经支持查询任意时间段的慢日志，即支持查询已经被 rotate 的慢日志文件的数据。用户查询时只需要指定 TIME 时间范围即可定位需要解析的慢日志文件。如果查询不指定时间范围，则仍然只解析当前的慢日志文件，示例如下：

不指定时间范围时，只会解析当前 TiDB 正在写入的慢日志文件的慢查询数据：

```
select count(*),
       min(time),
       max(time)
from slow_query;
```

```
+----------+----------------------------+----------------------------+
| count(*) | min(time)                  | max(time)                  |
+----------+----------------------------+----------------------------+
| 122492   | 2020-03-11 23:35:20.908574 | 2020-03-25 19:16:38.229035 |
+----------+----------------------------+----------------------------+
```

指定查询 2020-03-10 00:00:00 到 2020-03-11 00:00:00 时间范围后，会定位指定时间范围内的慢日志文件后解析慢查询数据：

```
select count(*),
       min(time),
       max(time)
from slow_query
where time > '2020-03-10 00:00:00'
  and time < '2020-03-11 00:00:00';
```

```
+----------+----------------------------+----------------------------+
| count(*) | min(time)                  | max(time)                  |
+----------+----------------------------+----------------------------+
| 2618049  | 2020-03-10 00:00:00.427138 | 2020-03-10 23:00:22.716728 |
+----------+----------------------------+----------------------------+
```

> **注意：**
>
> 如果指定时间范围内的慢日志文件被删除，或者并没有慢查询，则查询结果会返回空。

TiDB 4.0 中新增了CLUSTER_SLOW_QUERY 系统表，用来查询所有 TiDB 节点的慢查询信息，表结构在 SLOW_QUERY 的基础上多增加了 INSTANCE 列，表示该行慢查询信息来自的 TiDB 节点地址。使用方式和SLOW_QUERY 系统表一样。

关于查询 CLUSTER_SLOW_QUERY 表，TiDB 会把相关的计算和判断下推到其他节点执行，而不是把其他节点的慢查询数据都取回来在一台 TiDB 上执行。

8.1.5 查询 SLOW_QUERY / CLUSTER_SLOW_QUERY 示例

8.1.5.1 搜索 Top N 的慢查询

查询 Top 2 的用户慢查询。is_internal=false 表示排除 TiDB 内部的慢查询，只看用户的慢查询：

```
select query_time, query
from information_schema.slow_query
where is_internal = false  -- 排除 TiDB 内部的慢查询 SQL
order by query_time desc
limit 2;
```

输出样例：

```
+--------------+-----------------------------------------------------------------+
| query_time   | query                                                           |
+--------------+-----------------------------------------------------------------+
| 12.77583857  | select * from t_slim, t_wide where t_slim.c0=t_wide.c0;         |
|  0.734982725 | select t0.c0, t1.c1 from t_slim t0, t_wide t1 where t0.c0=t1.c0; |
+--------------+-----------------------------------------------------------------+
```

8.1.5.2 搜索某个用户的 Top N 慢查询

下面例子中搜索 test 用户执行的慢查询 SQL，且按执行消耗时间逆序排序显式前 2 条：

```
select query_time, query, user
from information_schema.slow_query
where is_internal = false  -- 排除 TiDB 内部的慢查询 SQL
  and user = "test"        -- 查找的用户名
order by query_time desc
limit 2;
```

输出样例：

```
+-------------+-----------------------------------------------------------------+----------------+
    ↪
| Query_time  | query                                                           | user
    ↪           |
+-------------+-----------------------------------------------------------------+----------------+
    ↪
```

```
| 0.676408014 | select t0.c0, t1.c1 from t_slim t0, t_wide t1 where t0.c0=t1.c1; | test
    ↪           |
+-------------+------------------------------------------------------------------+----------------+
    ↪
```

8.1.5.3 根据 SQL 指纹搜索同类慢查询

在得到 Top N 的慢查询 SQL 后，可通过 SQL 指纹继续搜索同类慢查询 SQL。

先获取 Top N 的慢查询和对应的 SQL 指纹：

```
select query_time, query, digest
from information_schema.slow_query
where is_internal = false
order by query_time desc
limit 1;
```

输出样例：

```
+-------------+-----------------------------+-------------------------------------------------------------------+
    ↪
| query_time  | query                       | digest
    ↪                                                                 |
+-------------+-----------------------------+-------------------------------------------------------------------+
    ↪
| 0.302558006 | select * from t1 where a=1; | 4751
    ↪ cb6008fda383e22dacb601fde85425dc8f8cf669338d55d944bafb46a6fa |
+-------------+-----------------------------+-------------------------------------------------------------------+
    ↪
```

再根据 SQL 指纹搜索同类慢查询：

```
select query, query_time
from information_schema.slow_query
where digest = "4751cb6008fda383e22dacb601fde85425dc8f8cf669338d55d944bafb46a6fa";
```

输出样例：

```
+-----------------------------+-------------+
| query                       | query_time  |
+-----------------------------+-------------+
| select * from t1 where a=1; | 0.302558006 |
| select * from t1 where a=2; | 0.401313532 |
+-----------------------------+-------------+
```

8.1.5.4 搜索统计信息为 pseudo 的慢查询 SQL 语句

```
select query, query_time, stats
from information_schema.slow_query
where is_internal = false
  and stats like '%pseudo%';
```

输出样例：

```
+----------------------------+-------------+---------------------------------+
| query                      | query_time  | stats                           |
+----------------------------+-------------+---------------------------------+
| select * from t1 where a=1; | 0.302558006 | t1:pseudo                       |
| select * from t1 where a=2; | 0.401313532 | t1:pseudo                       |
| select * from t1 where a>2; | 0.602011247 | t1:pseudo                       |
| select * from t1 where a>3; | 0.50077719  | t1:pseudo                       |
| select * from t1 join t2;   | 0.931260518 | t1:407872303825682445,t2:pseudo |
+----------------------------+-------------+---------------------------------+
```

8.1.5.5 查询执行计划发生变化的慢查询

由于统计信息过时，或者统计信息因为误差无法精确反映数据的真实分布情况时，可能导致同类型 SQL 的执行计划发生改变导致执行变慢，可以用以下 SQL 查询哪些 SQL 具有不同的执行计划：

```
select count(distinct plan_digest) as count,
       digest,
       min(query)
from cluster_slow_query
group by digest
having count > 1
limit 3\G
```

输出样例：

```
***************************[ 1. row ]***************************
count      | 2
digest     | 17b4518fde82e32021877878bec2bb309619d384fca944106fcaf9c93b536e94
min(query) | SELECT DISTINCT c FROM sbtest25 WHERE id BETWEEN ? AND ? ORDER BY c [arguments:
    ↪ (291638, 291737)];
***************************[ 2. row ]***************************
count      | 2
digest     | 9337865f3e2ee71c1c2e740e773b6dd85f23ad00f8fa1f11a795e62e15fc9b23
min(query) | SELECT DISTINCT c FROM sbtest22 WHERE id BETWEEN ? AND ? ORDER BY c [arguments:
    ↪ (215420, 215519)];
***************************[ 3. row ]***************************
count      | 2
digest     | db705c89ca2dfc1d39d10e0f30f285cbbadec7e24da4f15af461b148d8ffb020
```

```
min(query) | SELECT DISTINCT c FROM sbtest11 WHERE id BETWEEN ? AND ? ORDER BY c [arguments:
    ↪ (303359, 303458)];
```

然后可以用查询结果中的 SQL 指纹进一步查询不同的 plan

```
select min(plan),
       plan_digest
from cluster_slow_query
where digest='17b4518fde82e32021877878bec2bb309619d384fca944106fcaf9c93b536e94'
group by plan_digest\G
```

输出样例：

```
*************************** 1. row ***************************
  min(plan):    Sort_6                  root    100.00131380758702      sbtest.sbtest25.c:asc
        └─HashAgg_10            root    100.00131380758702      group by:sbtest.sbtest25.c,
            ↪ funcs:firstrow(sbtest.sbtest25.c)->sbtest.sbtest25.c
          └─TableReader_15      root    100.00131380758702      data:TableRangeScan_14
            └─TableScan_14      cop     100.00131380758702      table:sbtest25, range
                ↪ :[502791,502890], keep order:false
plan_digest: 6afbbd21f60ca6c6fdf3d3cd94f7c7a49dd93c00fcf8774646da492e50e204ee
*************************** 2. row ***************************
  min(plan):    Sort_6                  root    1                       sbtest.sbtest25.c:asc
        └─HashAgg_12            root    1                       group by:sbtest.sbtest25.c,
            ↪ funcs:firstrow(sbtest.sbtest25.c)->sbtest.sbtest25.c
          └─TableReader_13      root    1                       data:HashAgg_8
            └─HashAgg_8         cop     1                       group by:sbtest.sbtest25.c,
              └─TableScan_11    cop     1.2440069558121831      table:sbtest25, range
                  ↪ :[472745,472844], keep order:false
```

8.1.5.6 查询集群各个 TIDB 节点的慢查询数量

```
select instance, count(*) from information_schema.cluster_slow_query where time >= "2020-03-06
    ↪ 00:00:00" and time < now() group by instance;
```

输出样例：

```
+---------------+----------+
| instance      | count(*) |
+---------------+----------+
| 0.0.0.0:10081 | 124      |
| 0.0.0.0:10080 | 119771   |
+---------------+----------+
```

8.1.5.7 查询仅出现在异常时间段的慢日志

假如发现 2020-03-10 13:24:00 ~ 2020-03-10 13:27:00 的 QPS 降低或者延迟上升等问题，可能是由于突然出现大查询导致的，可以用下面 SQL 查询仅出现在异常时间段的慢日志，其中 2020-03-10 13:20:00 ~ 2020-03-10 13:23:00 为正常时间段。

```
SELECT * FROM
    (SELECT /*+ AGG_TO_COP(), HASH_AGG() */ count(*),
         min(time),
         sum(query_time) AS sum_query_time,
         sum(Process_time) AS sum_process_time,
         sum(Wait_time) AS sum_wait_time,
         sum(Commit_time),
         sum(Request_count),
         sum(process_keys),
         sum(Write_keys),
         max(Cop_proc_max),
         min(query),min(prev_stmt),
         digest
    FROM information_schema.CLUSTER_SLOW_QUERY
    WHERE time >= '2020-03-10 13:24:00'
            AND time < '2020-03-10 13:27:00'
            AND Is_internal = false
    GROUP BY  digest) AS t1
WHERE t1.digest NOT IN
    (SELECT /*+ AGG_TO_COP(), HASH_AGG() */ digest
    FROM information_schema.CLUSTER_SLOW_QUERY
    WHERE time >= '2020-03-10 13:20:00'
            AND time < '2020-03-10 13:23:00'
    GROUP BY  digest)
ORDER BY  t1.sum_query_time DESC limit 10\G
```

输出样例：

```
***************************[ 1. row ]***************************
count(*)           | 200
min(time)          | 2020-03-10 13:24:27.216186
sum_query_time     | 50.114126194
sum_process_time   | 268.351
sum_wait_time      | 8.476
sum(Commit_time)   | 1.044304306
sum(Request_count) | 6077
sum(process_keys)  | 202871950
sum(Write_keys)    | 319500
max(Cop_proc_max)  | 0.263
min(query)         | delete from test.tcs2 limit 5000;
min(prev_stmt)     |
```

```
digest              | 24bd6d8a9b238086c9b8c3d240ad4ef32f79ce94cf5a468c0b8fe1eb5f8d03df
```

8.1.6 解析其他的 TiDB 慢日志文件

TiDB 通过 session 变量 `tidb_slow_query_file` 控制查询 `INFORMATION_SCHEMA.SLOW_QUERY` 时要读取和解析的文件，可通过修改改 session 变量的值来查询其他慢查询日志文件的内容：

```
set tidb_slow_query_file = "/path-to-log/tidb-slow.log"
```

8.1.7 用 pt-query-digest 工具分析 TiDB 慢日志

可以用 `pt-query-digest` 工具分析 TiDB 慢日志。

> **注意：**
>
> 建议使用 pt-query-digest 3.0.13 及以上版本。

示例如下：

```
pt-query-digest --report tidb-slow.log
```

输出样例：

```
## 320ms user time, 20ms system time, 27.00M rss, 221.32M vsz
## Current date: Mon Mar 18 13:18:51 2019
## Hostname: localhost.localdomain
## Files: tidb-slow.log
## Overall: 1.02k total, 21 unique, 0 QPS, 0x concurrency _________________
## Time range: 2019-03-18-12:22:16 to 2019-03-18-13:08:52
## Attribute          total     min     max     avg     95%  stddev  median
## ============     ======= ======= ======= ======= ======= ======= =======
## Exec time           218s    10ms     13s   213ms    30ms      1s    19ms
## Query size       175.37k       9   2.01k  175.89  158.58  122.36  158.58
## Commit time         46ms     2ms     7ms     3ms     7ms     1ms     3ms
## Conn ID               71       1      16    8.88   15.25    4.06    9.83
## Process keys     581.87k       2 103.15k  596.43  400.73   3.91k  400.73
## Process time         31s     1ms     10s    32ms    19ms   334ms    16ms
## Request coun       1.97k       1      10    2.02    1.96    0.33    1.96
## Total keys       636.43k       2 103.16k  652.35  793.42   3.97k  400.73
## Txn start ts     374.38E       0  16.00E 375.48P   1.25P  89.05T   1.25P
## Wait time          943ms     1ms    19ms     1ms     2ms     1ms   972us
.
.
.
```

8.1.7.1 定位问题语句的方法

并不是所有 SLOW_QUERY 的语句都是有问题的。会造成集群整体压力增大的，是那些 process_time 很大的语句。wait_time 很大，但 process_time 很小的语句通常不是问题语句，是因为被问题语句阻塞，在执行队列等待造成的响应时间过长。

8.1.8 admin show slow 命令

除了获取 TiDB 日志，还有一种定位慢查询的方式是通过 admin show slow SQL 命令：

```
admin show slow recent N;
```

```
admin show slow top [internal | all] N;
```

recent N 会显示最近的 N 条慢查询记录，例如：

```
admin show slow recent 10;
```

top N 则显示最近一段时间（大约几天）内，最慢的查询记录。如果指定 internal 选项，则返回查询系统内部 SQL 的慢查询记录；如果指定 all 选项，返回系统内部和用户 SQL 汇总以后的慢查询记录；默认只返回用户 SQL 中的慢查询记录。

```
admin show slow top 3;
admin show slow top internal 3;
admin show slow top all 5;
```

由于内存限制，保留的慢查询记录的条数是有限的。当命令查询的 N 大于记录条数时，返回的结果记录条数会小于 N。

输出内容详细说明，如下：

列名	描述
start	SQL 语句执行开始时间
duration	SQL 语句执行持续时间
details	执行语句的详细信息
succ	SQL 语句执行是否成功，1: 成功，0: 失败
conn_id	session 连接 ID
transcation_ts	事务提交的 commit ts
user	执行该语句的用户名
db	执行该 SQL 涉及到 database
table_ids	执行该 SQL 涉及到表的 ID
index_ids	执行该 SQL 涉及到索引 ID
internal	表示为 TiDB 内部的 SQL 语句
digest	表示 SQL 语句的指纹
sql	执行的 SQL 语句

8.2 分析慢查询

处理慢查询分为两步：

1. 从大量查询中定位出哪一类查询比较慢
2. 分析这类慢查询的原因

第一步可以通过慢日志、statement-summary 方便地定位，推荐直接使用TiDB Dashboard，它整合了这两个功能，且能方便直观地在浏览器中展示出来。本文聚焦第二步。

首先将慢查询归因成两大类：

- 优化器问题：如选错索引，选错 Join 类型或顺序。
- 系统性问题：将非优化器问题都归结于此类。如：某个 TiKV 实例忙导致处理请求慢，Region 信息过期导致查询变慢。

实际中，优化器问题可能造成系统性问题。例如对于某类查询，优化器应使用索引，但却使用了全表扫。这可能导致这类 SQL 消耗大量资源，造成某些 KV 实例 CPU 飚高等问题。表现上看就是一个系统性问题，但本质是优化器问题。

分析优化器问题需要有判断执行计划是否合理的能力，而系统性问题的定位相对简单，因此面对慢查询推荐的分析过程如下：

1. 定位查询瓶颈：即查询过程中耗时多的部分
2. 分析系统性问题：根据瓶颈点，结合当时的监控/日志等信息，分析可能的原因
3. 分析优化器问题：分析是否有更好的执行计划

接下来会分别介绍上面几点。

8.2.1 定位查询瓶颈

定位查询瓶颈需要对查询过程有一个大致理解，TiDB 处理查询过程的关键阶段都在 performance-map 图中了。

查询的耗时信息可以从下面几种方式获得：

- 慢日志（推荐直接在TiDB Dashboard 中查看）
- `explain analyze` 语句

他们的侧重点不同：

- 慢日志记录了 SQL 从解析到返回，几乎所有阶段的耗时，较为全面（在 TiDB Dashboard 中可以直观地查询和分析慢日志）；
- `explain analyze` 可以拿到 SQL 实际执行中每个执行算子的耗时，对执行耗时有更细分的统计；

总的来说，利用慢日志和 `explain analyze` 可以比较准确地定位查询的瓶颈点，帮助你判断这条 SQL 慢在哪个模块 (TiDB/TiKV)，慢在哪个阶段，下面会有一些例子。

另外在 4.0.3 之后，慢日志中的 `Plan` 字段也会包含 SQL 的执行信息，也就是 `explain analyze` 的结果，这样一来 SQL 的所有耗时信息都可以在慢日志中找到。

8.2.2 分析系统性问题

对于系统性问题，我们根据执行阶段，分成三个大类：

1. TiKV 处理慢：如 coprocessor 处理数据慢
2. TiDB 执行慢：主要指执行阶段，如某个 Join 算子处理数据慢
3. 其他关键阶段慢：如取时间戳慢

拿到一个慢查询，我们应该先根据已有信息判断大致是哪个大类，再具体分析。

8.2.2.1 TiKV 处理慢

如果是 TiKV 处理慢，可以很明显的通过 `explain analyze` 中看出来。例如下面这个例子，可以看到 `StreamAgg_8` 和 `TableFullScan_15` 这两个 `tikv-task`（在 `task` 列可以看出这两个任务类型是 `cop[tikv]`）花费了 `170ms`，而 TiDB 部分的算子耗时，减去这 `170ms` 后，耗时占比非常小，说明瓶颈在 TiKV。

```
+--
    ↪ -------------------------+---------+---------+-----------+---------------+--------------------------------
    ↪
| id                          | estRows | actRows | task      | access object | execution info
    ↪                                                                          | operator info
    ↪                    | memory    | disk |
+--
    ↪ -------------------------+---------+---------+-----------+---------------+--------------------------------
    ↪
| StreamAgg_16                | 1.00    | 1       | root      |               | time:170.08572ms,
    ↪ loops:2                                                                | funcs:count(Column#5)->
    ↪ Column#3 | 372 Bytes | N/A  |
| └─TableReader_17            | 1.00    | 1       | root      |               | time:170.080369ms
    ↪ , loops:2, rpc num: 1, rpc time:17.023347ms, proc keys:28672 | data:StreamAgg_8
    ↪                 | 202 Bytes | N/A  |
|   └─StreamAgg_8             | 1.00    | 1       | cop[tikv] |               | time:170ms, loops
    ↪ :29                                                                    | funcs:count(1)->Column#5
    ↪        | N/A       | N/A  |
|     └─TableFullScan_15      | 7.00    | 28672   | cop[tikv] | table:t       | time:170ms, loops
    ↪ :29                                                                    | keep order:false, stats:
    ↪ pseudo  | N/A       | N/A  |
+--
    ↪ -------------------------+---------+---------+-----------+---------------+--------------------------------
    ↪
```

另外在慢日志中，`Cop_process` 和 `Cop_wait` 字段也可以帮助判断，如下面这个例子，查询整个耗时是 180.85ms 左右，而最大的那个 `coptask` 就消耗了 `171ms`，可以说明对这个查询而言，瓶颈在 TiKV 侧。

慢日志中的各个字段的说明可以参考慢查询日志中的字段含义说明

```
## Query_time: 0.18085
...
## Num_cop_tasks: 1
## Cop_process: Avg_time: 170ms P90_time: 170ms Max_time: 170ms Max_addr: 10.6.131.78
## Cop_wait: Avg_time: 1ms P90_time: 1ms Max_time: 1ms Max_Addr: 10.6.131.78
```

根据上述方式判断是 TiKV 慢后，可以依次排查 TiKV 慢的原因。

8.2.2.1.1 TiKV 实例忙

一条 SQL 可能会去从多个 TiKV 上拿数据，如果某个 TiKV 响应慢，可能拖慢整个 SQL 的处理速度。

慢日志中的 Cop_wait 可以帮忙判断这个问题：

```
## Cop_wait: Avg_time: 1ms P90_time: 2ms Max_time: 110ms Max_Addr: 10.6.131.78
```

如上图，发给 10.6.131.78 的一个 cop-task 等待了 110ms 才被执行，可以判断是当时该实例忙，此时可以打开当时的 CPU 监控辅助判断。

8.2.2.1.2 过期 key 多

如果 TiKV 上过期的数据比较多，在扫描的时候则需要处理这些不必要的数据，影响处理速度。

这可以通过 Total_keys 和 Processed_keys 判断，如果两者相差较大，则说明旧版本的 key 太多：

```
...
## Total_keys: 2215187529 Processed_keys: 1108056368
...
```

8.2.2.2 其他关键阶段慢

8.2.2.2.1 取 TS 慢

可以对比慢日志中的 Wait_TS 和 Query_time ，因为 TS 有预取操作，通常来说 Wait_TS 应该很低。

```
## Query_time: 0.0300000
...
## Wait_TS: 0.02500000
```

8.2.2.2.2 Region 信息过期

TiDB 侧 Region 信息可能过期，此时 TiKV 可能返回 regionMiss 的错误，然后 TiDB 会从 PD 去重新获取 Region 信息，这些信息会被反应在 Cop_backoff 信息内，失败的次数和总耗时都会被记录下来。

```
## Cop_backoff_regionMiss_total_times: 200 Cop_backoff_regionMiss_total_time: 0.2
    ↪ Cop_backoff_regionMiss_max_time: 0.2 Cop_backoff_regionMiss_max_addr: 127.0.0.1
    ↪ Cop_backoff_regionMiss_avg_time: 0.2 Cop_backoff_regionMiss_p90_time: 0.2
## Cop_backoff_rpcPD_total_times: 200 Cop_backoff_rpcPD_total_time: 0.2
    ↪ Cop_backoff_rpcPD_max_time: 0.2 Cop_backoff_rpcPD_max_addr: 127.0.0.1
    ↪ Cop_backoff_rpcPD_avg_time: 0.2 Cop_backoff_rpcPD_p90_time: 0.2
```

8.2.2.2.3 子查询被提前执行

对于带有非关联子查询的语句，子查询部分可能被提前执行，如：`select * from t1 where a = (select max(a)from t2)`，`select max(a)from t2` 部分可能在优化阶段被提前执行。这种查询用 `explain analyze` 看不到对应的耗时，如下：

```
mysql> explain analyze select count(*) from t where a=(select max(t1.a) from t t1, t t2 where t1.
    ↪ a=t2.a);
+--
    ↪ ---------------------------+----------+---------+-----------+---------------+-------------------------+
    ↪
| id                           | estRows  | actRows | task      | access object | execution info
    ↪           | operator info                     | memory    | disk |
+--
    ↪ ---------------------------+----------+---------+-----------+---------------+-------------------------+
    ↪
| StreamAgg_59                 | 1.00     | 1       | root      |               | time:4.69267ms,
    ↪  loops:2  | funcs:count(Column#10)->Column#8 | 372 Bytes | N/A  |
| └─TableReader_60             | 1.00     | 1       | root      |               | time:4.690428
    ↪ ms, loops:2 | data:StreamAgg_48                | 141 Bytes | N/A  |
|   └─StreamAgg_48             | 1.00     |         | cop[tikv] |               | time:0ns,
    ↪ loops:0        | funcs:count(1)->Column#10        | N/A       | N/A  |
|     └─Selection_58           | 16384.00 |         | cop[tikv] |               | time:0ns,
    ↪ loops:0        | eq(test.t.a, 1)                  | N/A       | N/A  |
|       └─TableFullScan_57     | 16384.00 | -1      | cop[tikv] | table:t       | time:0s, loops
    ↪ :0         | keep order:false                 | N/A       | N/A  |
+--
    ↪ ---------------------------+----------+---------+-----------+---------------+-------------------------+
    ↪
5 rows in set (7.77 sec)
```

不过可以从慢日志中排查这种情况：

```
## Query_time: 7.770634843
...
## Rewrite_time: 7.765673663 Preproc_subqueries: 1 Preproc_subqueries_time: 7.765231874
```

可以看到有 1 个子查询被提前执行，花费了 7.76s。

8.2.2.3 TiDB 执行慢

这里我们假设 TiDB 的执行计划正确（不正确的情况在分析优化器问题这一节中说明），但是执行上很慢；

解决这类问题主要靠调整参数或利用 hint，并结合 `explain analyze` 对 SQL 进行调整。

8.2.2.3.1 并发太低

如果发现瓶颈在有并发的算子上，可以通过调整并发度来尝试提速，如下面的执行计划中：

```
mysql> explain analyze select sum(t1.a) from t t1, t t2 where t1.a=t2.a;
+--
    ↪ -------------------------------+-------------+----------+----------+--------------+----------------
    ↪
| id                              | estRows      | actRows   | task      | access object |
    ↪ execution info                                                                    |
    ↪ operator info                                  | memory           | disk    |
+--
    ↪ -------------------------------+-------------+----------+----------+--------------+----------------
    ↪
| HashAgg_11                      | 1.00         | 1         | root      |               | time
    ↪ :9.666832189s, loops:2, PartialConcurrency:4, FinalConcurrency:4                | funcs:
    ↪ sum(Column#6)->Column#5                  | 322.125 KB       | N/A     |
| └─Projection_24                 | 268435456.00 | 268435456 | root      |               | time
    ↪ :9.098644711s, loops:262145, Concurrency:4                                      | cast(
    ↪ test.t.a, decimal(65,0) BINARY)->Column#6 | 199 KB           | N/A     |
|   └─HashJoin_14                 | 268435456.00 | 268435456 | root      |               | time
    ↪ :6.616773501s, loops:262145, Concurrency:5, probe collision:0, build:881.404µs  | inner
    ↪ join, equal:[eq(test.t.a, test.t.a)]     | 131.75 KB        | 0 Bytes |
|     ├─TableReader_21(Build)     | 16384.00     | 16384     | root      |               | time
    ↪ :6.553717ms, loops:17                                                           | data:
    ↪ Selection_20                             | 33.6318359375 KB | N/A     |
|     │ └─Selection_20            | 16384.00     |           | cop[tikv] |               | time
    ↪ :0ns, loops:0                                                                   | not(
    ↪ isnull(test.t.a))                         | N/A              | N/A     |
|     │   └─TableFullScan_19      | 16384.00     | -1        | cop[tikv] | table:t2      | time
    ↪ :0s, loops:0                                                                    | keep
    ↪ order:false                              | N/A              | N/A     |
|     └─TableReader_18(Probe)     | 16384.00     | 16384     | root      |               | time
    ↪ :6.880923ms, loops:17                                                           | data:
    ↪ Selection_17                             | 33.6318359375 KB | N/A     |
|       └─Selection_17            | 16384.00     |           | cop[tikv] |               | time
    ↪ :0ns, loops:0                                                                   | not(
    ↪ isnull(test.t.a))                         | N/A              | N/A     |
|         └─TableFullScan_16      | 16384.00     | -1        | cop[tikv] | table:t1      | time
    ↪ :0s, loops:0                                                                    | keep
    ↪ order:false                              | N/A              | N/A     |
```

```
+--
    ↪ ------------------------------+--------------+-----------+-----------+---------------+----------------
    ↪
9 rows in set (9.67 sec)
```

发现耗时主要在 HashJoin_14 和 Projection_24，可以酌情通过 SQL 变量来提高他们的并发度进行提速。

system-variables 中有所有的系统变量，如想提高 HashJoin_14 的并发度，则可以修改变量 tidb_hash_join_concurrency
↪ 。

8.2.2.3.2 产生了落盘

执行慢的另一个原因是执行过程中，因为到达内存限制，产生了落盘，这点在执行计划和慢日志中都能看到：

```
+--
    ↪ ----------------------+-----------+---------+-----------+---------------+--------------------------------+
    ↪
| id                      | estRows   | actRows | task      | access object | execution info
    ↪               | operator info        | memory                | disk           |
+--
    ↪ ----------------------+-----------+---------+-----------+---------------+--------------------------------+
    ↪
| Sort_4                  | 462144.00 | 462144  | root      |               | time:2.02848898s,
    ↪ loops:453  | test.t.a             | 149.68795776367188 MB | 219.3203125 MB |
| └─TableReader_8         | 462144.00 | 462144  | root      |               | time:616.211272ms,
    ↪  loops:453 | data:TableFullScan_7 | 197.49601364135742 MB | N/A            |
|   └─TableFullScan_7     | 462144.00 | -1      | cop[tikv] | table:t       | time:0s, loops:0
    ↪              | keep order:false     | N/A                   | N/A            |
+--
    ↪ ----------------------+-----------+---------+-----------+---------------+--------------------------------+
    ↪
```

```
...
## Disk_max: 229974016
...
```

8.2.2.3.3 做了笛卡尔积的 Join

做笛卡尔积的 Join 会产生 左边孩子行数 * 右边孩子行数 这么多数据，效率较低，应该尽量避免；

目前对于产生笛卡尔积的 Join 会在执行计划中显示的标明 CARTESIAN，如下：

```
mysql> explain select * from t t1, t t2 where t1.a>t2.a;
+--
    ↪ ---------------------------+-------------+-----------+---------------+--------------------------------
    ↪
```

```
| id                           | estRows     | task      | access object | operator info
    ↪                                        |
+--
    ↪ -----------------------------+-------------+-----------+---------------+-------------------------------
    ↪
| HashJoin_8                   | 99800100.00 | root      |               | CARTESIAN inner join,
    ↪ other cond:gt(test.t.a, test.t.a) |
| ├─TableReader_15(Build)      | 9990.00     | root      |               | data:Selection_14
    ↪                                       |
| │ └─Selection_14             | 9990.00     | cop[tikv] |               | not(isnull(test.t.a))
    ↪                                     |
| │   └─TableFullScan_13       | 10000.00    | cop[tikv] | table:t2      | keep order:false,
    ↪ stats:pseudo                        |
| └─TableReader_12(Probe)      | 9990.00     | root      |               | data:Selection_11
    ↪                                       |
|   └─Selection_11             | 9990.00     | cop[tikv] |               | not(isnull(test.t.a))
    ↪                                     |
|     └─TableFullScan_10       | 10000.00    | cop[tikv] | table:t1      | keep order:false,
    ↪ stats:pseudo                        |
+--
    ↪ -----------------------------+-------------+-----------+---------------+-------------------------------
    ↪
```

8.2.3 分析优化器问题

分析优化问题需要有判断执行计划是否合理的能力，这需要对优化过程和各算子有一定了解。

下面是一组例子，假设表结构为 `create table t (id int, a int, b int, c int, primary key(id), key(a) ↪ , key(b, c))`：

1. `select * from t`: 没有过滤条件，会扫全表，所以会用 `TableFullScan` 算子读取数据；
2. `select a from t where a=2`：有过滤条件且只读索引列，所以会用 `IndexReader` 算子读取数据；
3. `select * from t where a=2`：在 `a` 有过滤条件，但索引 `a` 不能完全覆盖需要读取的内容，因此会采用 `IndexLookup`；
4. `select b from t where c=3`：多列索引没有前缀条件就用不上，所以会用 `IndexFullScan`；
5. …

上面举例了数据读入相关的算子，在理解 TiDB 执行计划中描述了更多算子的情况。

另外阅读SQL 性能调优整个小节能增加你对 TiDB 优化器的了解，帮助判断执行计划是否合理。

由于大多数优化器问题在SQL 性能调优已经有解释，这里就直接列举出来跳转过去：

1. 索引选择错误
2. Join 顺序错误
3. 表达式未下推

8.3 SQL 诊断

> **警告：**
>
> 该功能目前为实验特性，不建议在生产环境中使用。

SQL 诊断功能是在 TiDB 4.0 版本中引入的特性，用于提升 TiDB 问题定位的效率。TiDB 4.0 版本以前，用户需要使用不同的工具以异构的方式获取不同信息。新的 SQL 诊断系统对这些离散的信息进行了整体设计，它整合系统各个维度的信息，通过系统表的方式向上层提供一致的接口，提供监控汇总与自动诊断，方便用户查询集群信息。

SQL 诊断共分三大块：

- 集群信息表：TiDB 4.0 诊断系统添加了集群信息表，为原先离散的各实例信息提供了统一的获取方式。它将整个集群的集群拓扑、硬件信息、软件信息、内核参数、监控、系统信息、慢查询、语句、日志完全整合在表中，让用户能够统一使用 SQL 进行查询。
- 集群监控表：TiDB 4.0 诊断系统添加了集群监控系统表，所有表都在 `metrics_schema` 中，可以通过 SQL 语句来查询监控信息。比起原先的可视化监控，SQL 查询监控允许用户对整个集群的所有监控进行关联查询，并对比不同时间段的结果，迅速找出性能瓶颈。由于 TiDB 集群的监控指标数量较大，SQL 诊断还提供了监控汇总表，让用户能够更便捷地从众多监控中找出异常的监控项。
- 自动诊断：尽管用户可以手动执行 SQL 来查询集群信息表、集群监控表与汇总表来定位问题，但自动诊断可以快速对常见异常进行定位。SQL 诊断基于已有的集群信息表和监控表，提供了与之相关的诊断结果表与诊断汇总表来执行自动诊断。

8.3.1 集群信息表

集群信息表将一个集群中的所有实例的信息都汇聚在一起，让用户仅通过一条 SQL 就能查询整个集群相关信息。集群信息表列表如下：

- 集群拓扑表`information_schema.cluster_info` 用于获取集群当前的拓扑信息，以及各个实例的版本、版本对应的 Git Hash、各实例的启动时间、各实例的运行时间。
- 集群配置表`information_schema.cluster_config` 用于获取集群当前所有实例的配置。对于 TiDB 4.0 之前的版本，用户必须逐个访问各个实例的 HTTP API 才能获取这些配置信息。
- 集群硬件表`information_schema.cluster_hardware` 用于快速查询集群硬件信息。
- 集群负载表`information_schema.cluster_load` 用于查询集群不同实例以及不同硬件类型的负载信息。
- 内核参数表`information_schema.cluster_systeminfo` 用于查询集群不同实例的内核配置信息。目前支持查询 sysctl 的信息。
- 集群日志表`information_schema.cluster_log` 用于集群日志查询，通过将查询条件下推到各个实例，降低日志查询对集群的影响，性能影响小于等 grep 命令。

TiDB 4.0 之前的系统表，只能查看当前实例信息，TiDB 4.0 实现了对应的集群表，可以在单个 TiDB 实例上拥有整个集群的全局视图。这些表目前都位于 `information_schema` 中，查询方式与其他 `information_schema` 系统表一致。

8.3.2 集群监控表

为了能够动态地观察并对比不同时间段的集群情况，TiDB 4.0 诊断系统添加了集群监控系统表。所有监控表都在 metrics_schema 中，可以通过 SQL 的方式查询监控信息。SQL 查询监控允许用户对整个集群的所有监控进行关联查询，并对比不同时间段的结果，迅速找出性能瓶颈。

- information_schema.metrics_tables：由于目前添加的系统表数量较多，因此用户可以通过该表查询这些监控表的相关元信息。

由于 TiDB 集群的监控指标数量较大，因此 TiDB 4.0 提供以下监控汇总表：

- 监控汇总表information_schema.metrics_summary 用于汇总所有监控数据，以提升用户排查各监控指标的效率。
- 监控汇总并按 label 聚合表information_schema.metrics_summary_by_label 同样用于汇总所有监控数据，但该表会对各项监控的不同的 label 进行聚合统计。

8.3.3 自动诊断

以上集群信息表和集群监控表均需要用户手动执行 SQL 语句来排查集群问题。TiDB 4.0 中添加了自动诊断功能，根据已有的基础信息表，提供诊断相关的系统表，使诊断自动执行。自动诊断相关的系统表如下：

- 诊断结果表information_schema.inspection_result 用于展示对系统的诊断结果。诊断是惰性触发，使用 select * from inspection_result 会触发所有诊断规则对系统进行诊断，并在结果中展示系统中的故障或风险。
- 诊断汇总表information_schema.inspection_summary 用于对特定链路或模块的监控进行汇总，用户可以根据整个模块或链路的上下文来排查定位问题。

8.4 定位消耗系统资源多的查询

TiDB 会将执行时间超过tidb_expensive_query_time_threshold 限制（默认值为 60s），或使用内存超过mem-quota-query 限制（默认值为 1 GB）的语句输出到tidb-server 日志文件（默认文件为 “tidb.log”）中，用于在语句执行结束前定位消耗系统资源多的查询语句（以下简称为 expensive query），帮助用户分析和解决语句执行的性能问题。

注意，expensive query 日志和慢查询日志的区别是，慢查询日志是在语句执行完后才打印，expensive query 日志可以将正在执行的语句的相关信息打印出来。当一条语句在执行过程中达到资源使用阈值时（执行时间/使用内存量），TiDB 会即时将这条语句的相关信息写入日志。

8.4.1 Expensive query 日志示例

```
[2020/02/05 15:32:25.096 +08:00] [WARN] [expensivequery.go:167] [expensive_query] [cost_time
    ↪ =60.008338935s] [wait_time=0s] [request_count=1] [total_keys=70] [process_keys=65] [
    ↪ num_cop_tasks=1] [process_avg_time=0s] [process_p90_time=0s] [process_max_time=0s] [
    ↪ process_max_addr=10.0.1.9:20160] [wait_avg_time=0.002s] [wait_p90_time=0.002s] [
    ↪ wait_max_time=0.002s] [wait_max_addr=10.0.1.9:20160] [stats=t:pseudo] [conn_id=60026] [
    ↪ user=root] [database=test] [table_ids="[122]"] [txn_start_ts=414420273735139329] [mem_max
    ↪ ="1035 Bytes (1.0107421875 KB)"] [sql="insert into t select sleep(1) from t"]
```

8.4.2 字段含义说明

基本字段：

- cost_time：日志打印时语句已经花费的执行时间。
- stats：语句涉及到的表或索引使用的统计信息版本。值为 pesudo 时表示无可用统计信息，需要对表或索引进行 analyze。
- table_ids：语句涉及到的表的 ID。
- txn_start_ts：事务的开始时间戳，也是事务的唯一 ID，可以用这个值在 TiDB 日志中查找事务相关的其他日志。
- sql：SQL 语句。

和内存使用相关的字段：

- mem_max：日志打印时语句已经使用的内存空间。该项使用两种单位标识内存使用量，分别为 Bytes 以及易于阅读的自适应单位（比如 MB、GB 等）。

和 SQL 执行的用户相关的字段：

- user：执行语句的用户名。
- conn_id：用户的连接 ID，可以用类似 con:60026 的关键字在 TiDB 日志中查找该连接相关的其他日志。
- database：执行语句时使用的 database。

和 TiKV Coprocessor Task 相关的字段：

- wait_time：该语句在 TiKV 的等待时间之和，因为 TiKV 的 Coprocessor 线程数是有限的，当所有的 Coprocessor 线程都在工作的时候，请求会排队；当队列中有某些请求耗时很长的时候，后面的请求的等待时间都会增加。
- request_count：该语句发送的 Coprocessor 请求的数量。
- total_keys：Coprocessor 扫过的 key 的数量。
- processed_keys：Coprocessor 处理的 key 的数量。与 total_keys 相比，processed_keys 不包含 MVCC 的旧版本。如果 processed_keys 和 total_keys 相差很大，说明旧版本比较多。
- num_cop_tasks：该语句发送的 Coprocessor 请求的数量。
- process_avg_time：Coprocessor 执行 task 的平均执行时间。
- process_p90_time：Coprocessor 执行 task 的 P90 分位执行时间。
- process_max_time：Coprocessor 执行 task 的最长执行时间。
- process_max_addr：task 执行时间最长的 Coprocessor 所在地址。
- wait_avg_time：Coprocessor 上 task 的等待时间。
- wait_p90_time：Coprocessor 上 task 的 P90 分位等待时间。
- wait_max_time：Coprocessor 上 task 的最长等待时间。
- wait_max_addr：task 等待时间最长的 Coprocessor 所在地址。

8.5 Statement Summary Tables

针对 SQL 性能相关的问题，MySQL 在 performance_schema 提供了 statement summary tables，用来监控和统计 SQL。例如其中的一张表 events_statements_summary_by_digest，提供了丰富的字段，包括延迟、执行次数、扫描行数、全表扫描次数等，有助于用户定位 SQL 问题。

为此，从 4.0.0-rc.1 版本开始，TiDB 在 information_schema（而不是 performance_schema）中提供与 events_statements_summary_by_digest 功能相似的系统表：

- statements_summary
- statements_summary_history
- cluster_statements_summary
- cluster_statements_summary_history

本文将详细介绍这些表，以及如何利用它们来排查 SQL 性能问题。

8.5.1 statements_summary

statements_summary 是 information_schema 里的一张系统表，它把 SQL 按 SQL digest 和 plan digest 分组，统计每一组的 SQL 信息。

此处的 SQL digest 与 slow log 里的 SQL digest 一样，是把 SQL 规一化后算出的唯一标识符。SQL 的规一化会忽略常量、空白符、大小写的差别。即语法一致的 SQL 语句，其 digest 也相同。

例如：

```
SELECT * FROM employee WHERE id IN (1, 2, 3) AND salary BETWEEN 1000 AND 2000;
select * from EMPLOYEE where ID in (4, 5) and SALARY between 3000 and 4000;
```

归一化后都是：

```
select * from employee where id in (...) and salary between ? and ?;
```

此处的 plan digest 是把执行计划规一化后算出的唯一标识符。执行计划的规一化会忽略常量的差别。由于相同的 SQL 可能产生不同的执行计划，所以可能分到多个组，同一个组内的执行计划是相同的。

statements_summary 用于保存 SQL 监控指标聚合后的结果。一般来说，每一项监控指标都包含平均值和最大值。例如执行延时对应 AVG_LATENCY 和 MAX_LATENCY 两个字段，分别是平均延时和最大延时。

为了监控指标的即时性，statements_summary 里的数据定期被清空，只展现最近一段时间内的聚合结果。清空周期由系统变量 tidb_stmt_summary_refresh_interval 设置。如果刚好在清空之后进行查询，显示的数据可能很少。

以下为查询 statements_summary 的部分结果：

```
  SUMMARY_BEGIN_TIME: 2020-01-02 11:00:00
    SUMMARY_END_TIME: 2020-01-02 11:30:00
           STMT_TYPE: Select
         SCHEMA_NAME: test
              DIGEST: 0611cc2fe792f8c146cc97d39b31d9562014cf15f8d41f23a4938ca341f54182
```

```
         DIGEST_TEXT: select * from employee where id = ?
         TABLE_NAMES: test.employee
         INDEX_NAMES: NULL
         SAMPLE_USER: root
          EXEC_COUNT: 3
         SUM_LATENCY: 1035161
         MAX_LATENCY: 399594
         MIN_LATENCY: 301353
         AVG_LATENCY: 345053
   AVG_PARSE_LATENCY: 57000
   MAX_PARSE_LATENCY: 57000
 AVG_COMPILE_LATENCY: 175458
 MAX_COMPILE_LATENCY: 175458
 ...........
             AVG_MEM: 103
             MAX_MEM: 103
             AVG_DISK: 65535
             MAX_DISK: 65535
   AVG_AFFECTED_ROWS: 0
          FIRST_SEEN: 2020-01-02 11:12:54
           LAST_SEEN: 2020-01-02 11:25:24
   QUERY_SAMPLE_TEXT: select * from employee where id=3100
    PREV_SAMPLE_TEXT:
         PLAN_DIGEST: f415b8d52640b535b9b12a9c148a8630d2c6d59e419aad29397842e32e8e5de3
                PLAN:  Point_Get_1     root    1       table:employee, handle:3100
```

> **注意：**
>
> 在 TiDB 中，statement summary tables 中字段的时间单位是纳秒 (ns)，而 MySQL 中的时间单位是皮秒 (ps)。

8.5.2 statements_summary_history

statements_summary_history 的表结构与 statements_summary 完全相同，用于保存历史时间段的数据。通过历史数据，可以排查过去出现的异常，也可以对比不同时间的监控指标。

字段 SUMMARY_BEGIN_TIME 和 SUMMARY_END_TIME 代表历史时间段的开始时间和结束时间。

8.5.3 cluster_statements_summary 和 cluster_statements_summary_history

statements_summary 和 statements_summary_history 仅显示单台 TiDB server 的 statement summary 数据。要查询整个集群的数据，需要查询 cluster_statements_summary 和 cluster_statements_summary_history。

cluster_statements_summary 显示各台 TiDB server 的 statements_summary 数据，cluster_statements_summary_history 显示各台 TiDB server 的 statements_summary_history 数据。这两张表用字段 INSTANCE 表示 TiDB server 的地址，其他字段与 statements_summary 相同。

8.5.4 参数配置

以下系统变量用于控制 statement summary：

- tidb_enable_stmt_summary：是否打开 statement summary 功能。1 代表打开，0 代表关闭，默认打开。statement summary 关闭后，系统表里的数据会被清空，下次打开后重新统计。经测试，打开后对性能几乎没有影响。
- tidb_stmt_summary_refresh_interval：statements_summary 的清空周期，单位是秒 (s)，默认值是 1800。
- tidb_stmt_summary_history_size：statements_summary_history 保存每种 SQL 的历史的数量，默认值是 24。
- tidb_stmt_summary_max_stmt_count：statement summary tables 保存的 SQL 种类数量。v4.0.14 前，默认 200 条。自 v4.0.14，默认 3000 条。当 SQL 种类超过该值时，会移除最近没有使用的 SQL。
- tidb_stmt_summary_max_sql_length：字段 DIGEST_TEXT 和 QUERY_SAMPLE_TEXT 的最大显示长度，默认值是 4096。
- tidb_stmt_summary_internal_query：是否统计 TiDB 的内部 SQL。1 代表统计，0 代表不统计，默认不统计。

> 注意：
>
> 当一种 SQL 因为达到 tidb_stmt_summary_max_stmt_count 限制要被移除时，TiDB 会移除该 SQL 语句种类在所有时间段的数据。因此，即使一个时间段内的 SQL 种类数量没有达到上限，显示的 SQL 语句数量也会比实际的少。如遇到该情况，建议调大 tidb_stmt_summary_max_stmt_count 的值。

statement summary 配置示例如下：

```
set global tidb_enable_stmt_summary = true;
set global tidb_stmt_summary_refresh_interval = 1800;
set global tidb_stmt_summary_history_size = 24;
```

以上配置生效后，statements_summary 每 30 分钟清空一次。因为 24 * 30 分钟 = 12 小时，所以 statements_summary_history 保存最近 12 小时的历史数据。

以上几个系统变量都有 global 和 session 两种作用域，它们的生效方式与其他系统变量不一样：

- 设置 global 变量后整个集群立即生效
- 设置 session 变量后当前 TiDB server 立即生效，这对于调试单个 TiDB server 比较有用
- 优先读 session 变量，没有设置过 session 变量才会读 global 变量
- 把 session 变量设为空字符串，将会重新读 global 变量

> **注意：**
>
> tidb_stmt_summary_history_size、tidb_stmt_summary_max_stmt_count、tidb_stmt_summary_max_sql_length 这些配置都影响内存占用，建议根据实际情况调整，不宜设置得过大。

8.5.5 目前的限制

Statement summary tables 现在还存在以下限制：

- TiDB server 重启后以上 4 张表的 statement summary 会全部丢失。因为 statement summary tables 全部都是内存表，不会持久化数据，所以一旦 server 被重启，statement summary 随之丢失。

8.5.6 排查示例

下面用两个示例问题演示如何利用 statement summary 来排查。

8.5.6.1 SQL 延迟比较大，是不是服务端的问题？

例如客户端显示 employee 表的点查比较慢，那么可以按 SQL 文本来模糊查询：

```
SELECT avg_latency, exec_count, query_sample_text
    FROM information_schema.statements_summary
    WHERE digest_text LIKE 'select * from employee%';
```

结果如下，avg_latency 是 1 ms 和 0.3 ms，在正常范围，所以可以判定不是服务端的问题，继而排查客户端或网络问题。

```
+-------------+------------+----------------------------------------+
| avg_latency | exec_count | query_sample_text                      |
+-------------+------------+----------------------------------------+
|     1042040 |          2 | select * from employee where name='eric' |
|      345053 |          3 | select * from employee where id=3100     |
+-------------+------------+----------------------------------------+
2 rows in set (0.00 sec)
```

8.5.6.2 哪类 SQL 的总耗时最高？

假如上午 10:00 到 10:30 的 QPS 明显下降，可以从历史表中找出当时耗时最高的三类 SQL：

```
SELECT sum_latency, avg_latency, exec_count, query_sample_text
    FROM information_schema.statements_summary_history
    WHERE summary_begin_time='2020-01-02 10:00:00'
    ORDER BY sum_latency DESC LIMIT 3;
```

结果显示以下三类 SQL 的总延迟最高，所以这些 SQL 需要重点优化。

```
+-------------+-------------+-----------+-----------------------------------------------------------------------------
    ↪
| sum_latency | avg_latency | exec_count | query_sample_text
    ↪                                                          |
+-------------+-------------+-----------+-----------------------------------------------------------------------------
    ↪
|     7855660 |     1122237 |          7 | select avg(salary) from employee where company_id=2013
    ↪                 |
|     7241960 |     1448392 |          5 | select * from employee join company on employee.
    ↪ company_id=company.id |
|     2084081 |     1042040 |          2 | select * from employee where name='eric'
    ↪                               |
+-------------+-------------+-----------+-----------------------------------------------------------------------------
    ↪
3 rows in set (0.00 sec)
```

8.5.7 表的字段介绍

下面介绍 `statements_summary` 表中各个字段的含义。

SQL 的基础信息：

- STMT_TYPE：SQL 语句的类型
- SCHEMA_NAME：执行这类 SQL 的当前 schema
- DIGEST：这类 SQL 的 digest
- DIGEST_TEXT：规一化后的 SQL
- QUERY_SAMPLE_TEXT：这类 SQL 的原 SQL 语句，多条语句只取其中一条
- TABLE_NAMES：SQL 中涉及的所有表，多张表用，分隔
- INDEX_NAMES：SQL 中使用的索引名，多个索引用，分隔
- SAMPLE_USER：执行这类 SQL 的用户名，多个用户名只取其中一个
- PLAN_DIGEST：执行计划的 digest
- PLAN：原执行计划，多条语句只取其中一条的执行计划
- PLAN_CACHE_HITS：这类 SQL 语句命中 plan cache 的总次数
- PLAN_IN_CACHE：这类 SQL 语句的上次执行是否命中了 plan cache

执行时间相关的信息：

- SUMMARY_BEGIN_TIME：当前统计的时间段的开始时间
- SUMMARY_END_TIME：当前统计的时间段的结束时间
- FIRST_SEEN：这类 SQL 的首次出现时间
- LAST_SEEN：这类 SQL 的最后一次出现时间

在 TiDB server 上的执行数据：

- EXEC_COUNT：这类 SQL 的总执行次数
- SUM_ERRORS：执行过程中遇到的 error 的总数
- SUM_WARNINGS：执行过程中遇到的 warning 的总数
- SUM_LATENCY：这类 SQL 的总延时
- MAX_LATENCY：这类 SQL 的最大延时
- MIN_LATENCY：这类 SQL 的最小延时
- AVG_LATENCY：这类 SQL 的平均延时
- AVG_PARSE_LATENCY：解析器的平均延时
- MAX_PARSE_LATENCY：解析器的最大延时
- AVG_COMPILE_LATENCY：优化器的平均延时
- MAX_COMPILE_LATENCY：优化器的最大延时
- AVG_MEM：使用的平均内存，单位 byte
- MAX_MEM：使用的最大内存，单位 byte
- AVG_DISK：使用的平均硬盘空间，单位 byte
- MAX_DISK：使用的最大硬盘空间，单位 byte

和 TiKV Coprocessor Task 相关的字段：

- SUM_COP_TASK_NUM：发送 Coprocessor 请求的总数
- MAX_COP_PROCESS_TIME：cop-task 的最大处理时间
- MAX_COP_PROCESS_ADDRESS：执行时间最长的 cop-task 所在地址
- MAX_COP_WAIT_TIME：cop-task 的最大等待时间
- MAX_COP_WAIT_ADDRESS：等待时间最长的 cop-task 所在地址
- AVG_PROCESS_TIME：SQL 在 TiKV 的平均处理时间
- MAX_PROCESS_TIME：SQL 在 TiKV 的最大处理时间
- AVG_WAIT_TIME：SQL 在 TiKV 的平均等待时间
- MAX_WAIT_TIME：SQL 在 TiKV 的最大等待时间
- AVG_BACKOFF_TIME：SQL 遇到需要重试的错误时在重试前的平均等待时间
- MAX_BACKOFF_TIME：SQL 遇到需要重试的错误时在重试前的最大等待时间
- AVG_TOTAL_KEYS：Coprocessor 扫过的 key 的平均数量
- MAX_TOTAL_KEYS：Coprocessor 扫过的 key 的最大数量
- AVG_PROCESSED_KEYS：Coprocessor 处理的 key 的平均数量。相比 avg_total_keys，avg_processed_keys 不包含 MVCC 的旧版本。如果 avg_total_keys 和 avg_processed_keys 相差很大，说明旧版本比较多
- MAX_PROCESSED_KEYS：Coprocessor 处理的 key 的最大数量

和事务相关的字段：

- AVG_PREWRITE_TIME：prewrite 阶段消耗的平均时间
- MAX_PREWRITE_TIME prewrite 阶段消耗的最大时间
- AVG_COMMIT_TIME：commit 阶段消耗的平均时间
- MAX_COMMIT_TIME：commit 阶段消耗的最大时间
- AVG_GET_COMMIT_TS_TIME：获取 commit_ts 的平均时间
- MAX_GET_COMMIT_TS_TIME：获取 commit_ts 的最大时间
- AVG_COMMIT_BACKOFF_TIME：commit 时遇到需要重试的错误时在重试前的平均等待时间
- MAX_COMMIT_BACKOFF_TIME：commit 时遇到需要重试的错误时在重试前的最大等待时间

- AVG_RESOLVE_LOCK_TIME：解决事务的锁冲突的平均时间
- MAX_RESOLVE_LOCK_TIME：解决事务的锁冲突的最大时间
- AVG_LOCAL_LATCH_WAIT_TIME：本地事务等待的平均时间
- MAX_LOCAL_LATCH_WAIT_TIME：本地事务等待的最大时间
- AVG_WRITE_KEYS：写入 key 的平均数量
- MAX_WRITE_KEYS：写入 key 的最大数量
- AVG_WRITE_SIZE：写入的平均数据量，单位 byte
- MAX_WRITE_SIZE：写入的最大数据量，单位 byte
- AVG_PREWRITE_REGIONS：prewrite 涉及的平均 Region 数量
- MAX_PREWRITE_REGIONS：prewrite 涉及的最大 Region 数量
- AVG_TXN_RETRY：事务平均重试次数
- MAX_TXN_RETRY：事务最大重试次数
- SUM_BACKOFF_TIMES：这类 SQL 遇到需要重试的错误后的总重试次数
- BACKOFF_TYPES：遇到需要重试的错误时的所有错误类型及每种类型重试的次数，格式为 类型:次数。如有多种错误则用 , 分隔，例如 txnLock:2,pdRPC:1
- AVG_AFFECTED_ROWS：平均影响行数
- PREV_SAMPLE_TEXT：当 SQL 是 COMMIT 时，该字段为 COMMIT 的前一条语句；否则该字段为空字符串。当 SQL 是 COMMIT 时，按 digest 和 prev_sample_text 一起分组，即不同 prev_sample_text 的 COMMIT 也会分到不同的行

8.6 TiDB 集群故障诊断

当试用 TiDB 遇到问题时，请先参考本篇文档。如果问题未解决，请按文档要求收集必要的信息通过 Github 提供给 TiDB 开发者。

8.6.1 如何给 TiDB 开发者报告错误

当使用 TiDB 遇到问题并且通过后面所列信息无法解决时，请收集以下信息并创建新 Issue:

- 具体的出错信息以及正在执行的操作
- 当前所有组件的状态
- 出问题组件 log 中的 error/fatal/panic 信息
- 机器配置以及部署拓扑
- dmesg 中 TiDB 组件相关的问题

8.6.2 数据库连接不上

首先请确认集群的各项服务是否已经启动，包括 tidb-server、pd-server、tikv-server。请用 ps 命令查看所有进程是否在。如果某个组件的进程已经不在了，请参考对应的章节排查错误。

如果所有的进程都在，请查看 tidb-server 的日志，看是否有报错？常见的错误包括：

- InformationSchema is out of date

  无法连接 tikv-server，请检查 pd-server 以及 tikv-server 的状态和日志。

- panic

  程序有错误，请将具体的 panic log 提供给 TiDB 开发者。

  如果是清空数据并重新部署服务，请确认以下信息：

- pd-server、tikv-server 数据都已清空

  tikv-server 存储具体的数据，pd-server 存储 tikv-server 中数据的元信息。如果只清空 pd-server 或只清空 tikv-server 的数据，会导致两边数据不匹配。

- 清空 pd-server 和 tikv-server 的数据并重启后，也需要重启 tidb-server

  集群 ID 是由 pd-server 在集群初始化时随机分配，所以重新部署集群后，集群 ID 会发生变化。tidb-server 业务需要重启以获取新的集群 ID。

8.6.3 tidb-server 启动报错

tidb-server 无法启动的常见情况包括：

- 启动参数错误

  请参考TiDB 命令行参数。

- 端口被占用：`lsof -i:port`

  请确保 tidb-server 启动所需要的端口未被占用。

- 无法连接 pd-server

  首先检查 pd-server 的进程状态和日志，确保 pd-server 成功启动，对应端口已打开：`lsof -i:port`。

  若 pd-server 正常，则需要检查 tidb-server 机器和 pd-server 对应端口之间的连通性，确保网段连通且对应服务端口已添加到防火墙白名单中，可通过 nc 或 curl 工具检查。

  例如，假设 tidb 服务位于 `192.168.1.100`，无法连接的 pd 位于 `192.168.1.101`，且 2379 为其 client port，则可以在 tidb 机器上执行 `nc -v -z 192.168.1.101 2379`，测试是否可以访问端口。或使用 `curl -v 192.168.1.101:2379/pd/api/v1/leader` 直接检查 pd 是否正常服务。

8.6.4 tikv-server 启动报错

- 启动参数错误

  请参考TiKV 启动参数文档。

- 端口被占用：`lsof -i:port`

  请确保 tikv-server 启动所需要的端口未被占用：`lsof -i:port`。

- 无法连接 pd-server

  首先检查 pd-server 的进程状态和日志。确保 pd-server 成功启动，对应端口已打开：`lsof -i:port`。

  若 pd-server 正常，则需要检查 tikv-server 机器和 pd-server 对应端口之间的连通性，确保网段连通且对应服务端口已添加到防火墙白名单中，可通过 nc 或 curl 工具检查。具体命令参考上一节。

- 文件被占用

  不要在一个数据库文件目录上打开两个 tikv。

8.6.5 pd-server 启动报错

- 启动参数错误

  请参考PD 命令行参数文档。

- 端口被占用：`lsof -i:port`

  请确保 pd-server 启动所需要的端口未被占用：`lsof -i:port`。

8.6.6 TiDB/TiKV/PD 进程异常退出

- 进程是否是启动在前台

  当前终端退出给其所有子进程发送 HUP 信号，从而导致进程退出。

- 是否是在命令行用过 nohup+& 方式直接运行

  这样依然可能导致进程因终端连接突然中断，作为终端 SHELL 的子进程被杀掉。

  推荐将启动命令写在脚本中，通过脚本运行（相当于二次 fork 启动）。

8.6.7 TiKV 进程异常重启

- 检查 dmesg 或者 syslog 里面是否有 OOM 信息

  如果有 OOM 信息并且杀掉的进程为 TiKV，请减少 TiKV 的 RocksDB 的各个 CF 的 `block-cache-size` 值。

- 检查 TiKV 日志是否有 panic 的 log

  提交 Issue 并附上 panic 的 log。

8.6.8 TiDB panic

请提供 panic 的 log。

8.6.9 连接被拒绝

- 请确保操作系统的网络参数正确，包括但不限于
  - 连接字符串中的端口和 tidb-server 启动的端口需要一致
  - 请保证防火墙的配置正确

8.6.10 Too many open files

在启动进程之前，请确保 `ulimit -n` 的结果足够大，推荐设为 unlimited 或者是大于 1000000。

8.6.11 数据库访问超时，系统负载高

首先检查SLOW-QUERY 日志，判断是否是因为某条 SQL 语句导致。如果未能解决，请提供如下信息：

- 部署的拓扑结构
  - tidb-server/pd-server/tikv-server 部署了几个实例
  - 这些实例在机器上是如何分布的
- 机器的硬件配置
  - CPU 核数
  - 内存大小
  - 硬盘类型（SSD 还是机械硬盘）
  - 是实体机还是虚拟机
- 机器上除了 TiDB 集群之外是否还有其他服务
- pd-server 和 tikv-server 是否分开部署
- 目前正在进行什么操作
- 用 `top -H` 命令查看当前占用 CPU 的线程名
- 最近一段时间的网络/IO 监控数据是否有异常

8.7 TiDB 集群问题导图

本篇文档总结了使用 TiDB 及其组件时的常见错误。遇到相关错误时，可以通过本文档的问题导图来排查错误原因并进行处理。

8.7.1 1. 服务不可用

8.7.1.1 1.1 客户端报 `Region is Unavailable` 错误

- 1.1.1 `Region is Unavailable` 一般是由于 Region 在一段时间不可用（可能会遇到 `TiKV server is busy`；或者发送给 TiKV 的请求由于 `not leader` 或者 `epoch not match` 等原因被打回；又或者请求 TiKV 超时等），TiDB 内部会进行 `backoff` 重试。`backoff` 的时间超过一定阈值（默认 20s）后就会报错给客户端。如果 `backoff` 在阈值内，客户端对该错误无感知。
- 1.1.2 多台 TiKV 同时内存不足 (OOM)，导致 Region 在一定时期内没有 Leader，见案例 case-991。
- 1.1.3 TiKV 报 `TiKV server is busy` 错误，超过 `backoff` 时间，参考4.3 客户端报 `server is busy` 错误。`TiKV server is busy` 属于内部流控机制，后续可能不计入 `backoff` 时间。
- 1.1.4 多台 TiKV 启动不了，导致 Region 没有 Leader。单台物理主机部署多个 TiKV 实例，一个物理机挂掉，由于 label 配置错误导致 Region 没有 Leader，见案例 case-228。
- 1.1.5 follower apply 落后，成为 Leader 之后把收到的请求以 `epoch not match` 理由打回，见案例 case-958（TiKV 内部需要优化该机制）。

8.7.1.2 1.2 PD 异常导致服务不可用

查看本文档5. PD 问题。

8.7.2 2. 延迟明显升高

8.7.2.1 2.1 延迟短暂升高

- 2.1.1 TiDB 执行计划不对导致延迟升高，请参考3.3 执行计划不对。
- 2.1.2 PD 出现选举问题或者 OOM 问题，请参考5.2 PD 选举问题 和5.3 PD OOM 问题。
- 2.1.3 某些 TiKV 大量掉 Leader，请参考4.4 某些 TiKV 大量掉 Leader。

8.7.2.2 2.2 Latency 持续升高

- 2.2.1 TiKV 单线程瓶颈
  - 单个 TiKV Region 过多，导致单个 gRPC 线程成为瓶颈（查看监控：Grafana -> TiKV-details -> Thread CPU/gRPC CPU Per Thread），v3.x 以上版本可以开启 Hibernate Region 特性解决该问题，见案例 case-612。
  - v3.0 之前版本 Raftstore 单线程或者 apply 单线程到达瓶颈（查看监控：Grafana -> TiKV-details -> Thread CPU/raft store CPU 和 Async apply CPU 超过 80%）。可以选择扩容 TiKV（v2.x 版本）实例，或者升级到多线程模型的 v3.x 版本。
- 2.2.2 CPU load 升高。
- 2.2.3 TiKV 写入慢，请参考4.5 TiKV 写入慢。
- 2.2.4 TiDB 执行计划不对，请参考3.3 执行计划不对。

8.7.3 3. TiDB 问题

8.7.3.1 3.1 DDL

- 3.1.1 修改 `decimal` 字段长度时报错 `"ERROR 1105 (HY000): unsupported modify decimal column precision"`。TiDB 暂时不支持修改 `decimal` 字段长度。
- 3.1.2 TiDB DDL job 卡住不动/执行很慢（通过 `admin show ddl jobs` 可以查看 DDL 进度）：
  - 原因 1：与外部组件（PD/TiKV）的网络问题。
  - 原因 2：早期版本（v3.0.8 之前）TiDB 内部自身负载很重（高并发下可能产生了很多协程）。
  - 原因 3：早期版本（v2.1.15 & v3.0.0-rc1 之前）PD 实例删除 TiDB key 无效的问题，会导致每次 DDL 变更都需要等 2 个 lease（很慢）。
  - 其他未知原因，请上报 bug。
  - 解决方法：原因 1 需要检查与外部组件的网络问题；原因 2 和 3 已经修复，需要升级到高版本；其他原因，可选择以下兜底方案进行 DDL owner 迁移。
  - DDL owner 迁移方案：
    * 如果与该 TiDB 集群可以网络互通，执行重新进行 owner 选举命令：`curl -X POST http://{TiDBIP}:10080/ddl/owner/resign`
    * 如果与该 TiDB 集群不可以网络互通，需旁路下线，通过 `tidb-ctl` 工具，从 PD 集群的 etcd 中直接删除 DDL owner，之后也会重新选举：`tidb-ctl etcd delowner [LeaseID] [flags] + ownerKey`

- 3.1.3 TiDB 日志中报 `information schema is changed` 的错误：
  - 原因 1：正在执行的 DML 所涉及的表和正在执行 DDL 的表相同，可以通过命令 `admin show ddl job` 查看正在运行的 DDL 操作。
  - 原因 2：当前执行的 DML 时间太久，且这段时间内执行了很多 DDL（新版本 `lock table` 也会导致 schema 版本变化），导致中间 `schema version` 变更超过 1024 个版本数。
  - 原因 3：当前执行 DML 请求的 TiDB 实例长时间不能加载到新的 `schema information`（与 PD 或者 TiKV 网络问题等都会导致此问题），而这段时间内执行了很多 DDL 语句（也包括 `lock table` 语句），导致中间 `schema version` 变更超过 1024 个版本数。
  - 解决方法：前 2 种原因都不会导致业务问题，相应的 DML 会在失败后重试；第 3 种原因需要检查 TiDB 实例和 PD 及 TiKV 的网络情况。
  - 背景知识：`schema version` 的增长数量与每个 DDL 变更操作的 `schema state` 个数一致，例如 `create table` 操作会有 1 个版本变更，`add column` 操作会有 4 个版本变更（详情可以参考 online schema change），所以太多的 column 变更操作会导致 `schema version` 增长得很快。
- 3.1.4 TiDB 日志中报 `information schema is out of date` 的错误：
  - 原因 1：执行 DML 的 TiDB 被 `graceful kill` 后准备退出，且此 DML 对应的事务执行时间超过一个 DDL lease，在事务提交的时候会报此错。
  - 原因 2：TiDB 在执行 DML 时，有一段时间连不上 PD 和 TiKV，导致以下问题：
    * TiDB 在超过一个 DDL Lease（默认 45s）的时间内没有加载到新的 schema；或者
    * TiDB 断开与 PD 之间带 `keep alive` 设置的连接。
  - 原因 3：TiKV 压力大或网络超时，通过监控 Grafana -> TiDB 和 TiKV 节点的负载情况来确认是否是该原因。
  - 解决方法：第 1 种原因，在 TiDB 启动时手动重试该 DML 即可；第 2 种原因，需要检查 TiDB 实例和 PD 及 TiKV 的网络波动情况；第 3 种原因，需要检查 TiKV 为什么繁忙，参考4. TiKV 问题。

8.7.3.2 3.2 OOM 问题

- 3.2.1 现象
  - 客户端：客户端收到 TiDB server 报错 `ERROR 2013 (HY000): Lost connection to MySQL server during query`
  - 日志：
    * `dmesg -T | grep tidb-server` 结果中有事故发生附近时间点的 OOM-killer 的日志。
    * tidb.log 中可以 `grep` 到事故发生后附近时间的 `"Welcome to TiDB"` 的日志（即 TiDB server 发生重启）。
    * tidb_stderr.log 中能 `grep` 到 `fatal error: "runtime: out of memory"` 或 `"cannot allocate memory"`。
    * v2.1.8 及其之前的版本，tidb_stderr.log 中能 `grep` 到 `fatal error: stack overflow`。
  - 监控：TiDB server 实例所在机器可用内存迅速回升
- 3.2.2 定位造成 OOM 的 SQL（目前所有版本都无法完成精准定位，需要在发现 SQL 后再做进一步分析，确认 OOM 是否的确由该 SQL 造成）：

- > = v3.0.0 的版本, 可以在 tidb.log 中 grep "expensive_query"，该 log 会记录运行超时、或使用内存超过阈值的 SQL。
- < v3.0.0 的版本, 通过 grep "memory exceeds quota" 定位运行时内存超限的 SQL。

> **注意：**
>
> 单条 SQL 内存阈值的默认值为 1GB，可通过 tidb_mem_quota_query 系统变量进行设置，作用域为 SESSION，单位为 Byte。也可以通过配置项热加载的方式，对配置文件中的 mem-quota-query 项进行修改，单位为 Byte。

- 3.2.3 缓解 OOM 问题
  - 通过开启 SWAP 的方式，可以缓解由于大查询使用内存过多而造成的 OOM 问题。但该方法由于存在 I/O 开销，会在内存空间不足时对大查询性能造成一定影响。性能回退程度受剩余内存量、读写盘速度影响。
- 3.2.4 OOM 常见原因
  - SQL 中包含 join，通过 explain 查看发现该 join 选用 HashJoin 算法且 inner 端的表很大。
  - 单条 UPDATE/DELETE 涉及的查询数据量太大，见案例 case-882。
  - SQL 中包含 Union 连接的多条子查询，见案例 case-1828。

8.7.3.3 3.3 执行计划不对

- 3.3.1 现象
  - SQL 相比于之前的执行时间有较大程度变慢，执行计划突然发生改变。如果慢日志中输出了执行计划，可以直接对比执行计划。
  - SQL 执行时间相比于其他数据库（例如 MySQL）有较大差距。可以对比其他数据库执行计划，例如 Join Order 是否不同。
  - 慢日志中 SQL 执行时间 Scan Keys 数目较大。
- 3.3.2 排查执行计划问题
  - explain analyze {SQL}。在执行时间可以接受的情况下，对比 explain analyze 结果中 count 和 execution info 中 rows 的数目差距。如果在 TableScan/IndexScan 行上发现比较大的差距，很大可能是统计信息出问题；如果在其他行上发现较大差距，则也有可能是非统计信息问题。
  - select count(*)。在执行计划中包含 join 等情况下，explain analyze 可能耗时过长；此时可以通过对 TableScan/IndexScan 上的条件进行 select count(*)，并对比 explain 结果中的 row count 信息，确定是不是统计信息的问题。
- 3.3.3 缓解问题
  - v3.0 及以上版本可以使用 SQL Bind 功能固定执行计划。

- 更新统计信息。在大致确定问题是由统计信息导致的情况下，先dump 统计信息保留现场。如果是由于统计信息过期导致，例如 show stats_meta 中 modify count/row count 大于某个值（例如 0.3）或者表中存在时间列的索引情况下，可以先尝试 analyze table 恢复；如果配置了 auto analyze，可以查看系统变量 tidb_auto_analyze_ratio 是否过大（例如大于 0.3），以及当前时间是否在 tidb_auto_analyze_start_time 和 tidb_auto_analyze_end_time 范围内。
- 其他情况，请上报 bug。

8.7.3.4 3.4 SQL 执行报错

- 3.4.1 客户端报 ERROR 1265(01000)Data Truncated 错误。原因是 TiDB 内在计算 Decimal 类型处理精度的时候，和 MySQL 不兼容。该错误已于 v3.0.10 中修复 (#14438)，具体原因如下：

  在 MySQL 内，如果两个大精度 Decimal 做除法运算，超出最大小数精度时 (30)，会只保留 30 位且不报错；TiDB 在计算结果上，也是这样实现的，但是在内部表示 Decimal 的结构体内，有一个表示小数精度的字段，还是保留的真实精度；

  比如 (0.1^30)/ 10，TiDB 和 MySQL 的结果都为 0，是正确的，因为精度最多 30；但是 TiDB 内表示精度的那个字段，还是 31；

  多次 Decimal 除法计算后，虽然结果正确，但是这个精度可能越来越大，最终超过 TiDB 内的另一个阈值 72，此时就会报 Data Truncated 的错误；Decimal 的乘法计算就不会有这个问题，因为绕过越界，会直接把精度设置为最大精度限制。

  解决方法：可以通过手动加 Cast(xx as decimal(a, b)) 来绕过这个问题，a 和 b 就是目标的精度。

8.7.4 4. TiKV 问题

8.7.4.1 4.1 TiKV panic 启动不了

- 4.1.1 sync-log = false,机器断电之后出现 unexpected raft log index: last_index X < applied_index ↪ Y 错误。符合预期，需通过 tikv-ctl 工具恢复 Region。
- 4.1.2 虚拟机部署 TiKV，kill 虚拟机或物理机断电，出现 entries[X, Y] is unavailable from storage 错误。符合预期，虚拟机的 fsync 不可靠，需通过 tikv-ctl 工具恢复 Region。
- 4.1.3 其他原因（非预期，需报 bug）。

8.7.4.2 4.2 TiKV OOM

- 4.2.1 block-cache 配置太大导致 OOM：
  - 在监控 Grafana -> TiKV-details 选中对应的 instance 后，查看 RocksDB 的 block cache size 监控来确认是否是该问题。
  - 同时，请检查 [storage.block-cache] capacity = # "1GB" 参数是否设置合理，默认情况下 TiKV 的 block-cache 设置为机器总内存的 45%；在 container 部署时，需要显式指定该参数，因为 TiKV 获取的是物理机的内存，可能会超出 container 的内存限制。
- 4.2.2 Coprocessor 收到大量大查询，返回的数据量太大，gRPC 的发送速度跟不上 Coprocessor 往外输出数据的速度，导致 OOM：

  - 可以通过检查监控：Grafana -> TiKV-details -> coprocessor overview 的 `response size` 是否超过 `network outbound` 流量来确认是否属于这种情况。

- 4.2.3 其他部分占用太多内存（非预期，需报 bug）。

8.7.4.3 4.3 客户端报 `server is busy` 错误

通过查看监控：Grafana -> TiKV -> errors 确认具体 busy 原因。`server is busy` 是 TiKV 自身的流控机制，TiKV 通过这种方式告知 `tidb/ti-client` 当前 TiKV 的压力过大，稍后再尝试。

- 4.3.1 TiKV RocksDB 出现 `write stall`。一个 TiKV 包含两个 RocksDB 实例，一个用于存储 Raft 日志，位于 `data/raft`。另一个用于存储真正的数据，位于 `data/db`。通过 `grep "Stalling" RocksDB` 日志查看 stall 的具体原因，RocksDB 日志是 LOG 开头的文件，LOG 为当前日志。

  - `level0 sst` 太多导致 stall，可以添加参数 `[rocksdb] max-sub-compactions = 2`（或者 3），加快 level0 sst 往下 compact 的速度。该参数的意思是将从 level0 到 level1 的 compaction 任务最多切成 `max-sub-compactions` 个子任务交给多线程并发执行，见案例 case-815。
  - `pending compaction bytes` 太多导致 stall，磁盘 I/O 能力在业务高峰跟不上写入，可以通过调大对应 Column Family (CF) 的 `soft-pending-compaction-bytes-limit` 和 `hard-pending-compaction-bytes-limit` 参数来缓解：
    * 如果 `pending compaction bytes` 达到该阈值，RocksDB 会放慢写入速度。默认值 64GB，`[rocksdb.defaultcf] soft-pending-compaction-bytes-limit = "128GB"`。
    * 如果 `pending compaction bytes` 达到该阈值，RocksDB 会 stop 写入，通常不太可能触发该情况，因为在达到 `soft-pending-compaction-bytes-limit` 的阈值之后会放慢写入速度。默认值 256GB，`hard-pending-compaction-bytes-limit = "512GB"`。
    * 如果磁盘 IO 能力持续跟不上写入，建议扩容。如果磁盘的吞吐达到了上限（例如 SATA SSD 的吞吐相对 NVME SSD 会低很多）导致 write stall，但是 CPU 资源又比较充足，可以尝试采用压缩率更高的压缩算法来缓解磁盘的压力，用 CPU 资源换磁盘资源。
    * 比如 default cf compaction 压力比较大，调整参数 `[rocksdb.defaultcf] compression-per-level = ["no", "no", "lz4", "lz4", "lz4", "zstd", "zstd"]` 改成 `compression-per-level = ["no", "no", "zstd", "zstd", "zstd", "zstd", "zstd"]`。
  - memtable 太多导致 stall。该问题一般发生在瞬间写入量比较大，并且 memtable flush 到磁盘的速度比较慢的情况下。如果磁盘写入速度不能改善，并且只有业务峰值会出现这种情况，可以通过调大对应 CF 的 `max-write-buffer-number` 来缓解：
    * 例如 `[rocksdb.defaultcf] max-write-buffer-number = 8`（默认值 5），同时请求注意在高峰期可能会占用更多的内存，因为可能存在于内存中的 memtable 会更多。

- 4.3.2 `scheduler too busy`

  - 写入冲突严重，`latch wait duration` 比较高，查看监控：Grafana -> TiKV-details -> scheduler prewrite 或者 scheduler commit 的 `latch wait duration`。scheduler 写入任务堆积，导致超过了 `[storage] scheduler-pending-write-threshold = "100MB"` 设置的阈值。可通过查看 MVCC_CONFLICT_COUNTER 对应的 metric 来确认是否属于该情况。
  - 写入慢导致写入堆积，该 TiKV 正在写入的数据超过了 `[storage] scheduler-pending-write-threshold = "100MB"` 设置的阈值。请参考4.5 TiKV 写入慢。

- 4.3.3 `raftstore is busy`，主要是消息的处理速度没有跟上接收消息的速度。短时间的 `channel full` 不会影响服务，长时间持续出现该错误可能会导致 Leader 切换走。
  - `append log` 遇到了 stall，参考4.3.1 客户端报 `server is busy` 错误。
  - `append log duration` 比较高，导致处理消息不及时，可以参考4.5 TiKV 写入慢 分析为什么 `append log duration` 比较高。
  - 瞬间收到大量消息（查看 TiKV Raft messages 面板），Raftstore 没处理过来，通常情况下短时间的 `channel full` 不会影响服务。
- 4.3.4 TiKV Coprocessor 排队，任务堆积超过了 `Coprocessor 线程数 * readpool.coprocessor.max-tasks-per-worker-[normal|low|high]`。大量大查询导致 Coprocessor 出现了堆积情况，需要确认是否由于执行计划变化而导致了大量扫表操作，请参考3.3 执行计划不对。

8.7.4.4 4.4 某些 TiKV 大量掉 Leader

- 4.4.1 TiKV 重启，导致重新选举。
  - TiKV `panic` 之后又被 systemd 重新拉起正常运行，可以通过查看 TiKV 的日志来确认是否有 `panic`，这种情况属于非预期，需报 bug。
  - 被第三者 `stop/kill`，被 systemd 重新拉起。查看 `dmesg` 和 `TiKV log` 确认原因。
  - TiKV 发生 OOM 导致重启了，参考4.2 TiKV OOM 问题。
  - 动态调整 THP 导致 hung 住，见案例 case-500。
- 4.4.2 查看监控：Grafana -> TiKV-details -> errors 面板 `server is busy`，看到 TiKV RocksDB 出现 write stall 导致发生重新选举，请参考4.3.1。
- 4.4.3 网络隔离导致重新选举。

8.7.4.5 4.5 TiKV 写入慢

- 4.5.1 通过查看 TiKV gRPC 的 `prewrite/commit/raw-put`(仅限 raw kv 集群) duration 确认确实是 TiKV 写入慢了。通常情况下可以按照 performance-map 来定位到底哪个阶段慢了，下面列出几种常见的情况。
- 4.5.2 scheduler CPU 繁忙（仅限 transaction kv）。prewrite/commit 的 `scheduler command duration` 比 `scheduler latch wait duration + storage async write duration` 更长，并且 scheduler worker CPU 比较高，例如超过 `scheduler-worker-pool-size * 100%` 的 80%，并且或者整个机器的 CPU 资源比较紧张。如果写入量很大，确认下是否 `[storage] scheduler-worker-pool-size` 配置得太小。其他情况，需报 bug。
- 4.5.3 Append log 慢。TiKV Grafana 的 Raft IO/`append log duration` 比较高，通常情况下是由于写盘慢了，可以检查 RocksDB - Raft 的 `WAL Sync Duration max` 值来确认，否则可能需要报 bug。
- 4.5.4 Raftstore 线程繁忙。TiKV Grafana 的 Raft Propose/`propose wait duration` 明显高于 `append log duration`。请查看以下情况：
  - `[raftstore] store-pool-size` 配置是否过小（该值建议在 [1,5] 之间，不建议太大）。
  - 机器的 CPU 是不是不够。
- 4.5.5 apply 慢了。TiKV Grafana 的 Raft IO/`apply log duration` 比较高，通常会伴随着 Raft Propose/`apply wait duration` 比较高。可能是以下原因引起的：

- `[raftstore] apply-pool-size` 配置过小（建议在 [1, 5] 之间，不建议太大），Thread CPU/apply cpu 比较高；
- 机器的 CPU 资源不够了；
- Region 写入热点问题，单个 apply 线程 CPU 使用率比较高（通过修改 Grafana 表达式，加上 by (instance, name) 来看各个线程的 CPU 使用情况），暂时对于单个 Region 的热点写入没有很好的方式，最近在优化该场景；
- 写 RocksDB 比较慢，RocksDB kv/`max write duration` 比较高（单个 Raft log 可能包含很多个 kv，写 RocksDB 的时候会把 128 个 kv 放在一个 write batch 写入到 RocksDB，所以一次 apply log 可能涉及到多次 RocksDB 的 write）；
- 其他情况，需报 bug。

- 4.5.6 Raft commit log 慢了。
  - TiKV Grafana 的 Raft IO/`commit log duration` 比较高（4.x 版本的 Grafana 才有该 metric）。每个 Region 对应一个独立的 Raft group，Raft 本身是有流控机制的，类似 TCP 的滑动窗口机制，通过参数 `[raftstore] raft-max-inflight-msgs = 256` 来控制滑动窗口的大小，如果有热点写入并且 `commit log duration` 比较高可以适度调大该参数，比如 1024。
- 4.5.7 其他情况，请参考 Performance Map 上的写入路径来分析。

8.7.5 5. PD 问题

8.7.5.1 5.1 PD 调度问题

- 5.1.1 merge 问题：
  - 跨表空 Region 无法 merge，需要修改 TiKV 的 `[coprocessor] split-region-on-table = false` 参数来解决，4.x 版本该参数默认为 false，见案例 case-896。
  - Region merge 慢，可检查监控 Grafana -> PD -> operator 面板是否有 merge 的 operator 产生，可以适当调大 `merge-schedule-limit` 参数来加速 merge。
- 5.1.2 补副本/上下线问题：
  - TiKV 磁盘使用 80% 容量，PD 不会进行补副本操作，miss peer 数量上升，需要扩容 TiKV，见案例 case-801。
  - 下线 TiKV，有 Region 长时间迁移不走。v3.0.4 版本已经修复改问题，见 #5526 和案例 case-870。
- 5.1.3 Balance 问题：
  - Leader/Region count 分布不均，见案例 case-394, case-759。主要原因是 balance 是依赖 Region/leader 的 size 去调度的，所以可能会造成 count 数量的不均衡，v4.0 新增了一个参数 `[leader-schedule-policy]`，可以调整 Leader 的调度策略，根据 “count” 或者是 “size” 进行调度。

8.7.5.2 5.2 PD 选举问题

- 5.2.1 PD 发生 Leader 切换：

- 磁盘问题，PD 所在的节点 I/O 被打满，排查是否有其他 I/O 高的组件与 PD 混部以及盘的健康情况，可通过监控 Grafana -> disk performance -> latency 和 load 等指标进行验证，必要时可以使用 fio 工具对盘进行检测，见案例 case-292。
- 网络问题，PD 日志中有 `lost the TCP streaming connection`，排查 PD 之间网络是否有问题，可通过监控 Grafana -> PD -> etcd 的 `round trip` 来验证，见案例 case-177。
- 系统 load 高，日志中能看到 `server is likely overloaded`，见案例 case-214。

- 5.2.2 PD 选不出 Leader 或者选举慢：
  - 选不出 Leader，PD 日志中有 `lease is not expired`，见 #10355。v3.0.x 版本和 v2.1.19 版本已修复该问题，见案例 case-875。
  - 选举慢，Region 加载时间长。从 PD 日志中 `grep "regions cost"`（例如日志中可能是 “load 460927 regions cost 11.77099s”），如果出现秒级，则说明较慢，v3.0 版本可开启 Region storage（设置 `use-region-storage` 为 `true`），该特性能极大缩短加载 Region 的时间，见案例 case-429。

- 5.2.3 TiDB 执行 SQL 时报 PD timeout：
  - PD 没 Leader 或者有切换，参考5.2.1 PD 选举问题 和5.2.2 PD 选举问题。
  - 网络问题，排查网络相关情况。通过监控 Grafana -> blackbox_exporter -> ping latency 确定 TiDB 到 PD Leader 的网络是否正常。
  - PD panic，需报 bug。
  - PD OOM，参考5.3 PD OOM 问题。
  - 其他原因，通过 `curl http://127.0.0.1:2379/debug/pprof/goroutine?debug=2` 抓 goroutine，报 bug。

- 5.2.4 其他问题
  - PD 报 FATAL 错误，日志中有 `range failed to find revision pair`，v3.0.8 已经修复该问题，见 #2040。详情参考案例 case-947。
  - 其他原因，需报 bug。

8.7.5.3 5.3 PD OOM

- 5.3.1 使用 `/api/v1/regions` 接口时 Region 数量过多，可能会导致 PD OOM，在 v3.0.8 版本中修复，见 #1986。
- 5.3.2 滚动升级的时候 PD OOM，gRPC 消息大小没限制，监控可看到 TCP InSegs 较大，在 v3.0.6 版本中修复，见 #1952。

8.7.5.4 5.4 Grafana 显示问题

- 5.4.1 监控 Grafana -> PD -> cluster -> role 显示 follower，Grafana 表达式问题，在 v3.0.8 版本修复，见 #1065。详情请参考案例 case-1022。

8.7.6 6. 生态 Tools 问题

8.7.6.1 6.1 TiDB Binlog 问题

- 6.1.1 TiDB Binlog 是将 TiDB 的修改同步给下游 TiDB 或者 MySQL 的工具，见 TiDB Binlog on GitHub。
- 6.1.2 Pump/Drainer Status 中 Update Time 正常更新，日志中也没有异常，但下游没有数据写入。
  - TiDB 配置中没有开启 binlog，需要修改 TiDB 配置 `[binlog]`。
- 6.1.3 Drainer 中的 sarama 报 EOF 错误。
  - Drainer 使用的 Kafka 客户端版本和 Kafka 版本不匹配，需要修改配置 `[syncer.to] kafka-version` 来解决。
- 6.1.4 Drainer 写 Kafka 失败然后 panic，Kafka 报 `Message was too large` 错误。
  - binlog 数据太大，造成写 Kafka 的单条消息太大，需要修改 Kafka 的下列配置来解决：

    ```
    message.max.bytes=1073741824
    replica.fetch.max.bytes=1073741824
    fetch.message.max.bytes=1073741824
    ```

    见案例 case-789。
- 6.1.5 上下游数据不一致
  - 部分 TiDB 节点没有开启 binlog。v3.0.6 及之后的版本可以通过访问 http://127.0.0.1:10080/info/all 接口可以检查所有节点的 binlog 状态。之前的版本可以通过查看配置文件来确认是否开启了 binlog。
  - 部分 TiDB 节点进入 ignore binlog 状态。v3.0.6 及之后的版本可以通过访问 http://127.0.0.1:10080/info/all 接口可以检查所有节点的 binlog 状态。之前的版本需要看 TiDB 的日志中是否有 ignore binlog 关键字来确认是该问题。
  - 上下游 Timestamp 列的值不一致：
    * 时区问题，需要确保 Drainer 和上下游数据库时区一致，Drainer 通过 `/etc/localtime` 获取时区，不支持 TZ 环境变量，见案例 case-826。
    * TiDB 中 Timestamp 的默认值为 `null`，MySQL 5.7 中 Timestamp 默认值为当前时间（MySQL 8 无此问题），因此当上游写入 `null` 的 Timestamp 且下游是 MySQL 5.7 时，Timestamp 列数据不一致。在开启 binlog 前，在上游执行 `set @@global.explicit_defaults_for_timestamp=on;` 可解决此问题。
  - 其他情况需报 bug。
- 6.1.6 同步慢
  - 下游是 TiDB/MySQL，上游频繁进行 DDL 操作，见案例 case-1023。
  - 下游是 TiDB/MySQL，需要同步的表中存在没有主键且没有唯一索引的表，这种情况会导致 binlog 性能下降，建议加主键或唯一索引。
  - 下游输出到文件，检查目标磁盘/网络盘是否慢。
  - 其他情况需报 bug。

- 6.1.7 Pump 无法写 binlog，报 `no space left on device` 错误。
  - 本地磁盘空间不足，Pump 无法正常写 binlog 数据。需要清理磁盘空间，然后重启 Pump。
- 6.1.8 Pump 启动时报错 `fail to notify all living drainer`。
  - Pump 启动时需要通知所有 Online 状态的 Drainer，如果通知失败则会打印该错误日志。
  - 可以使用 binlogctl 工具查看所有 Drainer 的状态是否有异常，保证 Online 状态的 Drainer 都在正常工作。如果某个 Drainer 的状态和实际运行情况不一致，则使用 binlogctl 修改状态，然后再重启 Pump。见案例fail-to-notify-all-living-drainer。
- 6.1.9 Drainer 报错 `gen update sqls failed: table xxx: row data is corruption []`。
  - 触发条件：上游做 Drop Column DDL 的时候同时在做这张表的 DML。已经在 v3.0.6 修复，见 case-820。
- 6.1.10 Drainer 同步卡住，进程活跃但 checkpoint 不更新。
  - 已知 bug 在 v3.0.4 修复，见案例 case-741。
- 6.1.11 任何组件 `panic`。
  - 需报 bug。

8.7.6.2 6.2 DM 问题

- 6.2.1 TiDB Data Migration (DM) 是能将 MySQL/MariaDB 的数据迁移到 TiDB 的迁移工具，详情见 DM on GitHub。
- 6.2.2 执行 `query-status` 或查看日志时出现 `Access denied for user 'root'@'172.31.43.27' (using password: YES)`。
  - 在所有 DM 配置文件中，数据库相关的密码都必须使用经 dmctl 加密后的密文（若数据库密码为空，则无需加密）。在 v1.0.6 及以后的版本可使用明文密码。
  - 在 DM 运行过程中，上下游数据库的用户必须具备相应的读写权限。在启动同步任务过程中，DM 会自动进行相应权限的检查。
  - 同一套 DM 集群，混合部署不同版本的 DM-worker/DM-master/dmctl，见案例 AskTUG-1049。
- 6.2.3 DM 同步任务中断并包含 `driver: bad connection` 错误。
  - 发生 `driver: bad connection` 错误时，通常表示 DM 到下游 TiDB 的数据库连接出现了异常（如网络故障、TiDB 重启等）且当前请求的数据暂时未能发送到 TiDB。
    * 1.0.0 GA 之前的版本，DM 发生该类型错误时，需要先使用 `stop-task` 命令停止任务后再使用 `start-task` 命令重启任务。
    * 1.0.0 GA 版本，增加对此类错误的自动重试机制，见 #265。
- 6.2.4 同步任务中断并包含 `invalid connection` 错误。
  - 发生 `invalid connection` 错误时，通常表示 DM 到下游 TiDB 的数据库连接出现了异常（如网络故障、TiDB 重启、TiKV busy 等）且当前请求已有部分数据发送到了 TiDB。由于 DM 中存在同步任务并发向下游复制数据的特性，因此在任务中断时可能同时包含多个错误（可通过 `query-status` 或 `query-error` 查询当前错误）：

    * 如果错误中仅包含 `invalid connection` 类型的错误，且当前处于增量复制阶段，则 DM 会自动进行重试。
    * 如果 DM 由于版本问题（v1.0.0-rc.1 后引入自动重试）等未自动进行重试或自动重试未能成功，则可尝试先使用 `stop-task` 停止任务，然后再使用 `start-task` 重启任务。

- 6.2.5 Relay 处理单元报错 `event from * in * diff from passed-in event *` 或同步任务中断并包含 `get binlog error ERROR 1236 (HY000)`、`binlog checksum mismatch, data may be corrupted` 等 binlog 获取或解析失败错误。

    - 在 DM 进行 relay log 拉取与增量同步过程中，如果遇到了上游超过 4 GB 的 binlog 文件，就可能出现这两个错误。原因是 DM 在写 relay log 时需要依据 binlog position 及文件大小对 event 进行验证，且需要保存同步的 binlog position 信息作为 checkpoint。但是 MySQL binlog position 官方定义使用 uint32 存储，所以超过 4 GB 部分的 binlog position 的 offset 值会溢出，进而出现上面的错误。
        * 对于 relay 处理单元，可通过官网步骤进行手动处理。
        * 对于 binlog replication 处理单元，可通过官网步骤进行手动处理。

- 6.2.6 DM 同步中断，日志报错 `ERROR 1236 (HY000)The slave is connecting using CHANGE MASTER TO MASTER_AUTO_POSITION = 1, but the master has purged binary logs containing GTIDs that the slave requires.`。

    - 检查 master 的 binlog 是否被 purge。
    - 检查 relay.meta 中记录的位点信息。
        * relay.meta 中记录空的 GTID 信息，DM-worker 进程在退出时、以及定时 (30s) 会把内存中的 GTID 信息保存到 relay.meta 中，在没有获取到上游 GTID 信息的情况下，把空的 GTID 信息保存到了 relay.meta 中。见案例 case-772。
        * relay.meta 中记录的 binlog event 不完整触发 recover 流程后记录错误的 GTID 信息，该问题可能会在 1.0.2 之前的版本遇到，已在 1.0.2 版本修复。

- 6.2.7 DM 同步报错 `Error 1366: incorrect utf8 value eda0bdedb29d(\ufffd\ufffd\ufffd\ufffd\ufffd\ufffd)`。

    - 该值 MySQL 8.0 和 TiDB 都不能写入成功，但是 MySQL 5.7 可以写入成功。可以开启 TiDB 动态参数 `tidb_skip_utf8_check` 参数，跳过数据格式检查。

8.7.6.3 6.3 TiDB Lightning 问题

- 6.3.1 TiDB Lightning 是快速的全量数据导入工具，见 TiDB Lightning on GitHub。

- 6.3.2 导入速度太慢。

    - `region-concurrency` 设定太高，线程间争用资源反而减低了效率。排查方法如下：
        * 从日志的开头搜寻 `region-concurrency` 能知道 TiDB Lightning 读到的参数是多少；
        * 如果 TiDB Lightning 与其他服务（如 TiKV Importer）共用一台服务器，必需手动将 `region-concurrency` 设为该服务器 CPU 数量的 75%；
        * 如果 CPU 设有限额（例如从 Kubernetes 指定的上限），TiDB Lightning 可能无法自动判断出来，此时亦需要手动调整 `region-concurrency`。

- 表结构太复杂。每条索引都会额外增加 KV 对，如果有 N 条索引，实际导入的大小就差不多是 Mydumper 文件的 N+1 倍。如果索引不太重要，可以考虑先从 schema 去掉，待导入完成后再使用 `CREATE INDEX` 加回去。
- TiDB Lightning 版本太旧。尝试使用最新的版本，可能会有改善。

- 6.3.3 `checksum failed: checksum mismatched remote vs local`

  - 原因 1：这张表可能本身已有数据，影响最终结果。
  - 原因 2：如果目标数据库的校验和全是 0，表示没有发生任何导入，有可能是集群太忙无法接收任何数据。
  - 原因 3：如果数据源是由机器生成而不是从 Mydumper 备份的，需确保数据符合表的限制，例如：
    * 自增 (AUTO_INCREMENT) 的列需要为正数，不能为 0。
    * 单一键和主键 (UNIQUE and PRIMARY KEYs) 不能有重复的值。
  - 解决办法：参考官网步骤处理。

- 6.3.4 `Checkpoint for … has invalid status:(`错误码`)`

  - 原因：断点续传已启用。TiDB Lightning 或 TiKV Importer 之前发生了异常退出。为了防止数据意外损坏，TiDB Lightning 在错误解决以前不会启动。错误码是小于 25 的整数，可能的取值是 0、3、6、9、12、14、15、17、18、20、21。整数越大，表示异常退出所发生的步骤在导入流程中越晚。
  - 解决办法：参考官网步骤处理。

- 6.3.5 `ResourceTemporarilyUnavailable("Too many open engines …: 8")`

  - 原因：并行打开的引擎文件 (engine files) 超出 tikv-importer 里的限制。这可能由配置错误引起。即使配置没问题，如果 tidb-lightning 曾经异常退出，也有可能令引擎文件残留在打开的状态，占据可用的数量。
  - 解决办法：参考官网步骤处理。

- 6.3.6 `cannot guess encoding for input file, please convert to UTF-8 manually`

  - 原因：TiDB Lightning 只支持 UTF-8 和 GB-18030 编码的表架构。此错误代表数据源不是这里任一个编码。也有可能是文件中混合了不同的编码，例如在不同的环境运行过 `ALTER TABLE`，使表架构同时出现 UTF-8 和 GB-18030 的字符。
  - 解决办法：参考官网步骤处理。

- 6.3.7 `[sql2kv] sql encode error = [types:1292]invalid time format: '{1970 1 1 0 45 0 0}'`

  - 原因：一个 timestamp 类型的时间戳记录了不存在的时间值。时间值不存在是由于夏令时切换或超出支持的范围 (1970 年 1 月 1 日至 2038 年 1 月 19 日)。
  - 解决办法：参考官网步骤处理。

8.7.7 7. 常见日志分析

8.7.7.1 7.1 TiDB

- 7.1.1 `GC life time is shorter than transaction duration`。事务执行时间太长，超过了 GC lifetime（默认 10min），可以通过修改 `mysql.tidb` 表来调整 `life time`，通常情况下不建议修改，会导致大量老版本堆积起来（如果有大量 `update` 和 `delete` 语句）。
- 7.1.2 `txn takes too much time`。事务太长时间（超过 590s）没有提交，准备提交的时候报该错误。可以通过调大 `[tikv-client] max-txn-time-use = 590` 参数，以及调大 `GC life time` 来绕过该问题（如果确实有这个需求）。通常情况下，建议看看业务是否真的需要执行这么长时间的事务。
- 7.1.3 coprocessor.go 报 `request outdated`。发往 TiKV 的 Coprocessor 请求在 TiKV 端排队时间超过了 60s，直接返回该错误。需要排查 TiKV Coprocessor 为什么排队这么严重。
- 7.1.4 region_cache.go 大量报 `switch region peer to next due to send request fail` 且 error 信息是 `context deadline exceeded`。请求 TiKV 超时触发 region cache 切换请求到其他节点，可以对日志中的 addr 字段继续 `grep "<addr> cancelled"`，根据 grep 结果：
  - `send request is cancelled`。请求发送阶段超时，可以排查监控 Grafana -> TiDB -> Batch Client/Pending Request Count by TiKV 是否大于 128，确定是否因发送远超 KV 处理能力导致发送堆积。如果 Pending Request 不多，需要排查日志确认是否因为对应 KV 有运维变更，导致短暂报出；否则非预期，需报 bug。
  - `wait response is cancelled`。请求发送到 TiKV 后超时未收到 TiKV 响应。需要排查对应地址 TiKV 的响应时间和对应 Region 在当时的 PD 和 KV 日志，确定为什么 KV 未及时响应。
- 7.1.5 distsql.go 报 `inconsistent index`。数据索引疑似发生不一致，首先对报错的信息中 index 所在表执行 `admin check table <TableName>` 命令，如果检查失败，则先通过命令关闭 GC，然后报 bug。

```
begin;
update mysql.tidb set variable_value='72h' where variable_name='tikv_gc_life_time';
commit;
```

8.7.7.2 7.2 TiKV

- 7.2.1 `key is locked`。读写冲突，读请求碰到还未提交的数据，需要等待其提交之后才能读。少量这个错误对业务无影响，大量出现这个错误说明业务读写冲突比较严重。
- 7.2.2 `write conflict`。乐观事务中的写写冲突，同时多个事务对相同的 key 进行修改，只有一个事务会成功，其他事务会自动重取 timestamp 然后进行重试，不影响业务。如果业务冲突很严重可能会导致重试多次之后事务失败，这种情况下建议使用悲观锁。
- 7.2.3 TxnLockNotFound。事务提交太慢，过了 TTL（小事务默认 3s）时间之后被其他事务回滚了，该事务会自动重试，通常情况下对业务无感知。
- 7.2.4 `PessimisticLockNotFound`。类似 TxnLockNotFound，悲观事务提交太慢被其他事务回滚了。
- 7.2.5 `stale_epoch`。请求的 epoch 太旧了，TiDB 会更新路由之后再重新发送请求，业务无感知。epoch 在 Region 发生 split/merge 以及迁移副本的时候会变化。

- 7.2.6 `peer is not leader`。请求发到了非 Leader 的副本上，TiDB 会根据该错误更新本地路由（如果错误 response 里携带了最新 Leader 是哪个副本这一信息），并且重新发送请求到最新 Leader，一般情况下业务无感知。在 v3.0 后 TiDB 在原 Leader 请求失败时会尝试其他 peer，也会导致 TiKV 频繁出现 `not leader` 日志，可以通过查看 TiDB 对应 Region 的 `switch region peer to next due to send request fail` 日志，排查发送失败根本原因，参考7.1.4 TiDB。另外也可能是由于其他原因导致一些 Region 一直没有 Leader，请参考4.4 某些 TiKV 大量掉 Leader。

8.8 TiDB 热点问题处理

本文适用于 TiDB 4.0 版本，介绍如何定位和解决读写热点问题。

TiDB 作为分布式数据库，内建负载均衡机制，尽可能将业务负载均匀地分布到不同计算或存储节点上，更好地利用上整体系统资源。然而，机制不是万能的，在一些场景下仍会有部分业务负载不能被很好地分散，影响性能，形成单点的过高负载，也称为热点。

TiDB 提供了完整的方案用于排查、解决或规避这类热点。通过均衡负载热点，可以提升整体性能，包括提高 QPS 和降低延迟等。

8.8.1 常见热点场景

8.8.1.1 TiDB 编码规则回顾

TiDB 对每个表分配一个 TableID，每一个索引都会分配一个 IndexID，每一行分配一个 RowID（默认情况下，如果表使用整数型的 Primary Key，那么会用 Primary Key 的值当做 RowID）。其中 TableID 在整个集群内唯一，IndexID/RowID 在表内唯一，这些 ID 都是 int64 类型。

每行数据按照如下规则进行编码成 Key-Value pair：

```
Key: tablePrefix{tableID}_recordPrefixSep{rowID}
Value: [col1, col2, col3, col4]
```

其中 Key 的 `tablePrefix` 和 `recordPrefixSep` 都是特定的字符串常量，用于在 KV 空间内区分其他数据。

对于 Index 数据，会按照如下规则编码成 Key-Value pair：

```
Key: tablePrefix{tableID}_indexPrefixSep{indexID}_indexedColumnsValue
Value: rowID
```

Index 数据还需要考虑 Unique Index 和非 Unique Index 两种情况，对于 Unique Index，可以按照上述编码规则。但是对于非 Unique Index，通过这种编码并不能构造出唯一的 Key，因为同一个 Index 的 `tablePrefix{tableID}` ↪ `_indexPrefixSep{indexID}` 都一样，可能有多行数据的 `ColumnsValue` 是一样的，所以对于非 Unique Index 的编码做了一点调整：

```
Key: tablePrefix{tableID}_indexPrefixSep{indexID}_indexedColumnsValue_rowID
Value: null
```

8.8.1.2 表热点

从 TiDB 编码规则可知，同一个表的数据会在以表 ID 开头为前缀的一个 range 中，数据的顺序按照 RowID 的值顺序排列。在表 insert 的过程中如果 RowID 的值是递增的，则插入的行只能在末端追加。当 Region 达到一定的大小之后会进行分裂，分裂之后还是只能在 range 范围的末端追加，永远只能在一个 Region 上进行 insert 操作，形成热点。

常见的 increment 类型自增主键就是顺序递增的，默认情况下，在主键为整数型时，会用主键值当做 RowID，此时 RowID 为顺序递增，在大量 insert 时形成表的写入热点。

同时，TiDB 中 RowID 默认也按照自增的方式顺序递增，主键不为整数类型时，同样会遇到写入热点的问题。

8.8.1.3 索引热点

索引热点与表热点类似，常见的热点场景出现在时间顺序单调递增的字段，或者插入大量重复值的场景。

8.8.2 确定存在热点问题

性能问题不一定是热点造成的，也可能存在多个因素共同影响，在排查前需要先确认是否与热点相关。

- 判断写热点依据：打开监控面板 TiKV-Trouble-Shooting 中 Hot Write 面板（如下图所示），观察 Raftstore CPU 监控是否存在个别 TiKV 节点的指标明显高于其他节点的现象。
- 判断读热点依据：打开监控面板 TIKV-Details 中 Thread_CPU，查看 coprocessor cpu 有没有明显的某个 TiKV 特别高。

8.8.3 使用 TiDB Dashboard 定位热点表

TiDB Dashboard 中的流量可视化功能可帮助用户缩小热点排查范围到表级别。以下是流量可视化功能展示的一个热力图样例，该图横坐标是时间，纵坐标是各个表和索引，颜色越亮代表其流量越大。可在工具栏中切换显示读或写流量。

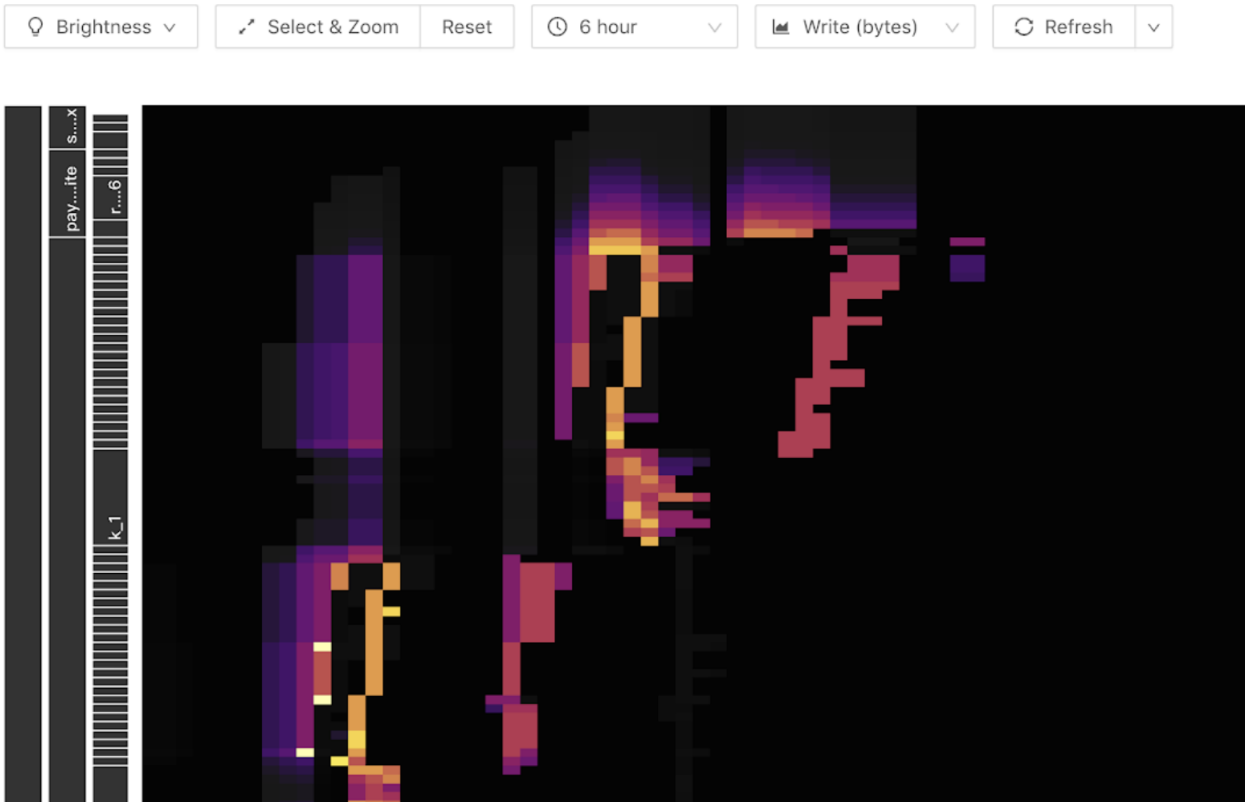


图 65: Dashboard 示例 1

当图中写入流量图出现以下明亮斜线（斜向上或斜向下）时，由于写入只出现在末端，随着表 Region 数量变多，呈现出阶梯状。此时说明该表构成了写入热点：

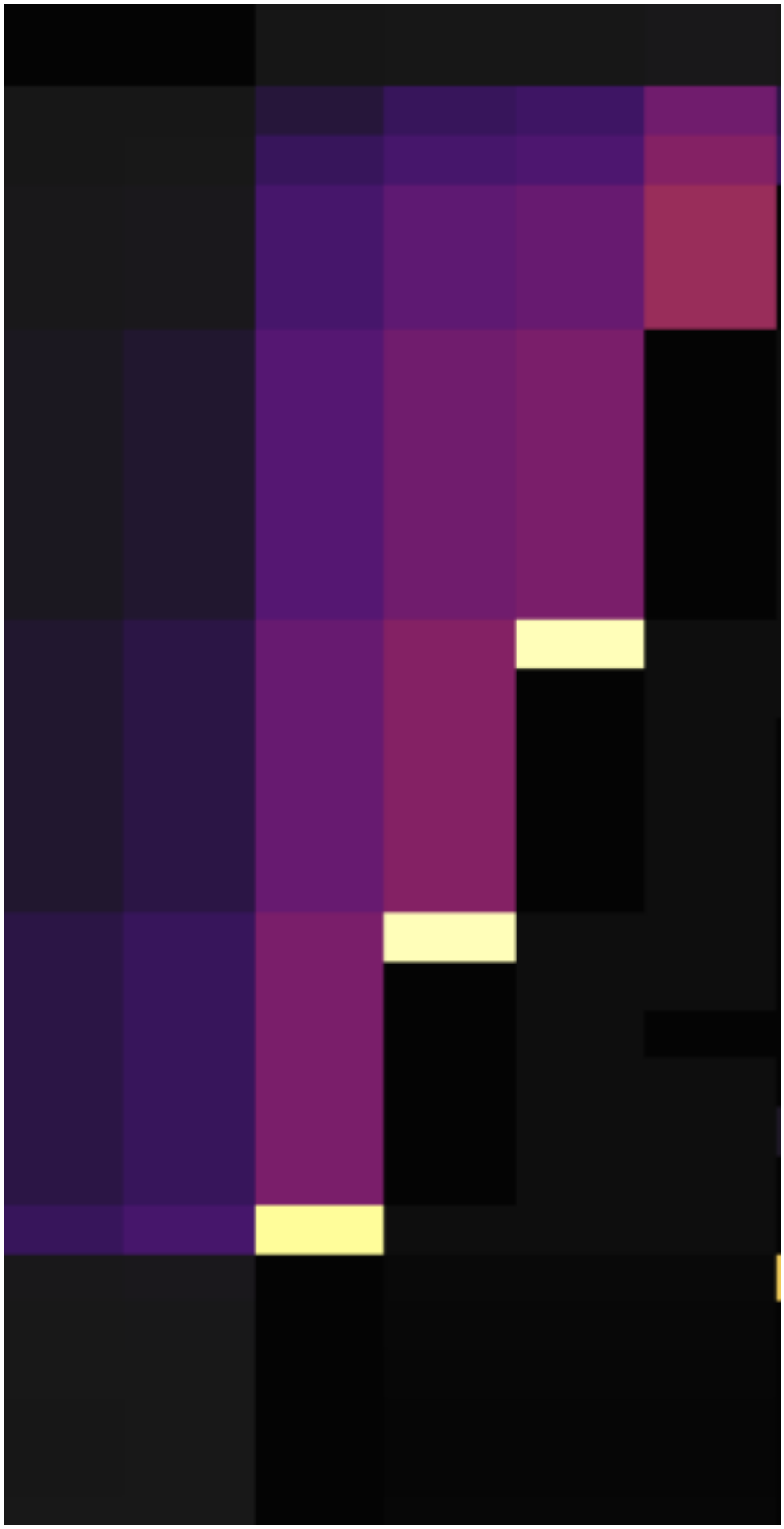


图 66: Dashboard 示例 2

对于读热点，在热力图中一般表现为一条明亮的横线，通常是有大量访问的小表，如下图所示：

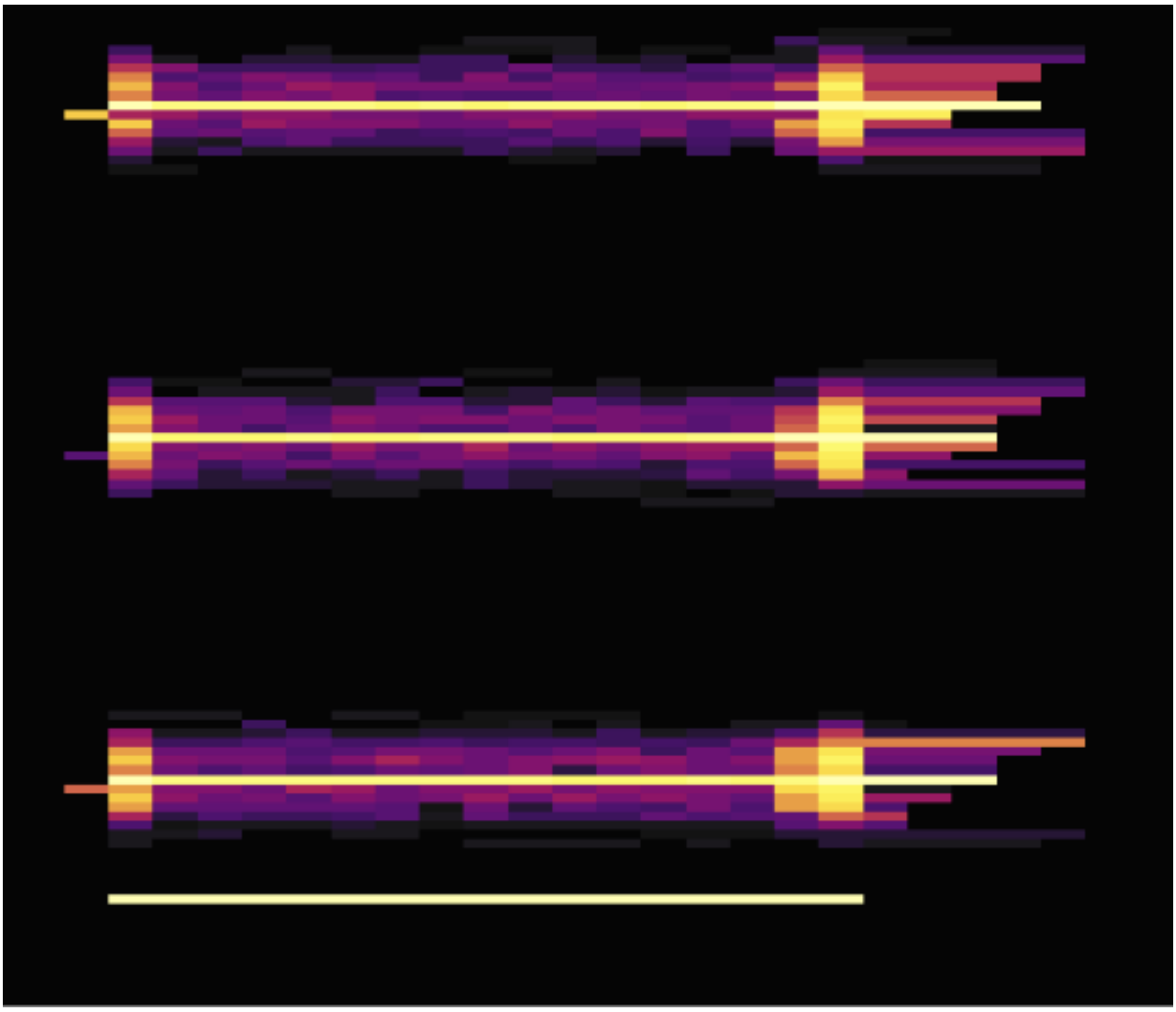


图 67: Dashboard 示例 3

将鼠标移到亮色块上，即可看到是什么表或索引具有大流量，如下图所示：

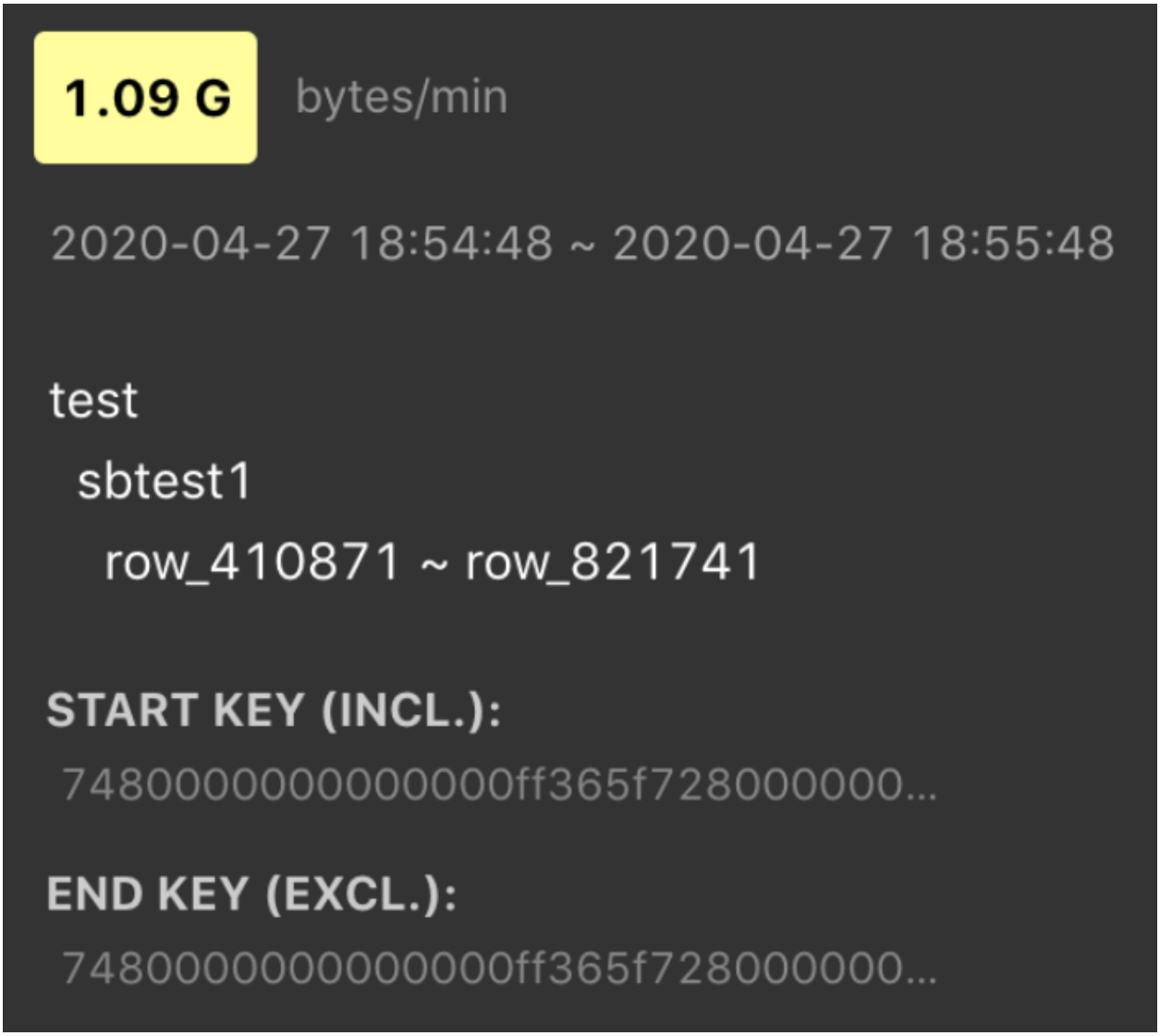


图 68: Dashboard 示例 4

> 4.0 之前版本热点定位可以参考此文档

8.8.4 使用 SHARD_ROW_ID_BITS 处理热点表

对于主键非整数或没有主键的表或者是联合主键，TiDB 会使用一个隐式的自增 RowID，大量 INSERT 时会把数据集中写入单个 Region，造成写入热点。

通过设置 SHARD_ROW_ID_BITS，可以把 RowID 打散写入多个不同的 Region，缓解写入热点问题。但是设置的过大会造成 RPC 请求数放大，增加 CPU 和网络开销。

```
SHARD_ROW_ID_BITS = 4 表示 16 个分片\
SHARD_ROW_ID_BITS = 6 表示 64 个分片\
SHARD_ROW_ID_BITS = 0 表示默认值 1 个分片
```

语句示例：

```
CREATE TABLE: CREATE TABLE t (c int) SHARD_ROW_ID_BITS = 4;
ALTER TABLE: ALTER TABLE t SHARD_ROW_ID_BITS = 4;
```

SHARD_ROW_ID_BITS 的值可以动态修改，每次修改之后，只对新写入的数据生效。

TiDB alter-primary-key 参数设置为 false 时，会使用表的整数型主键作为 RowID，因为 SHARD_ROW_ID_BITS 会改变 RowID 生成规则，所以此时无法使用 SHARD_ROW_ID_BITS 选项。在 alter-primary-key 参数设置为 true 时，TiDB 在建表时不再使用整数型主键作为 RowID，此时带有整数型主键的表也可以使用 SHARD_ROW_ID_BITS 特性。

以下是两张无主键情况下使用 SHARD_ROW_ID_BITS 打散热点后的流量图，第一张展示了打散前的情况，第二张展示了打散后的情况。

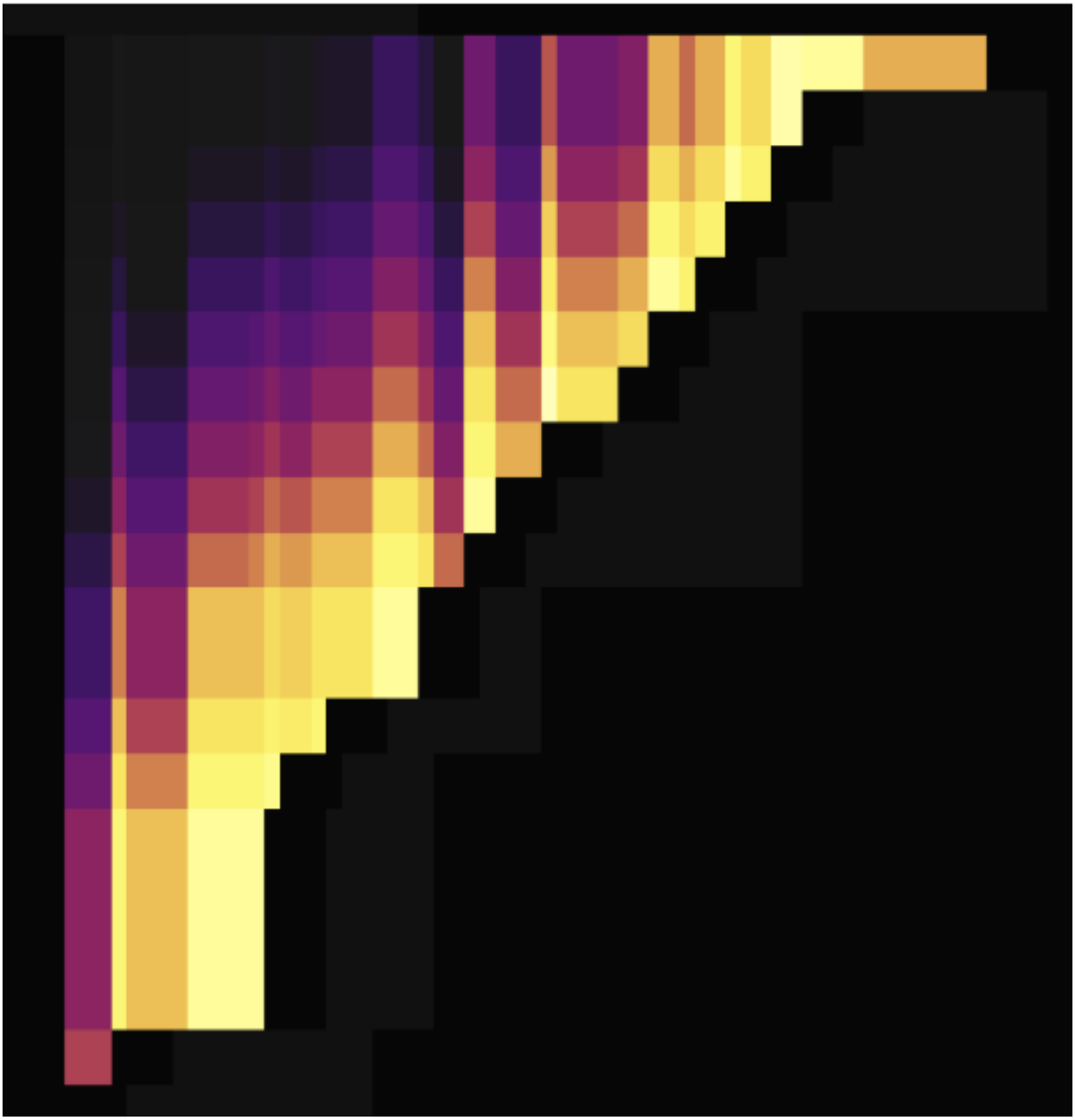


图 69: Dashboard 示例 5

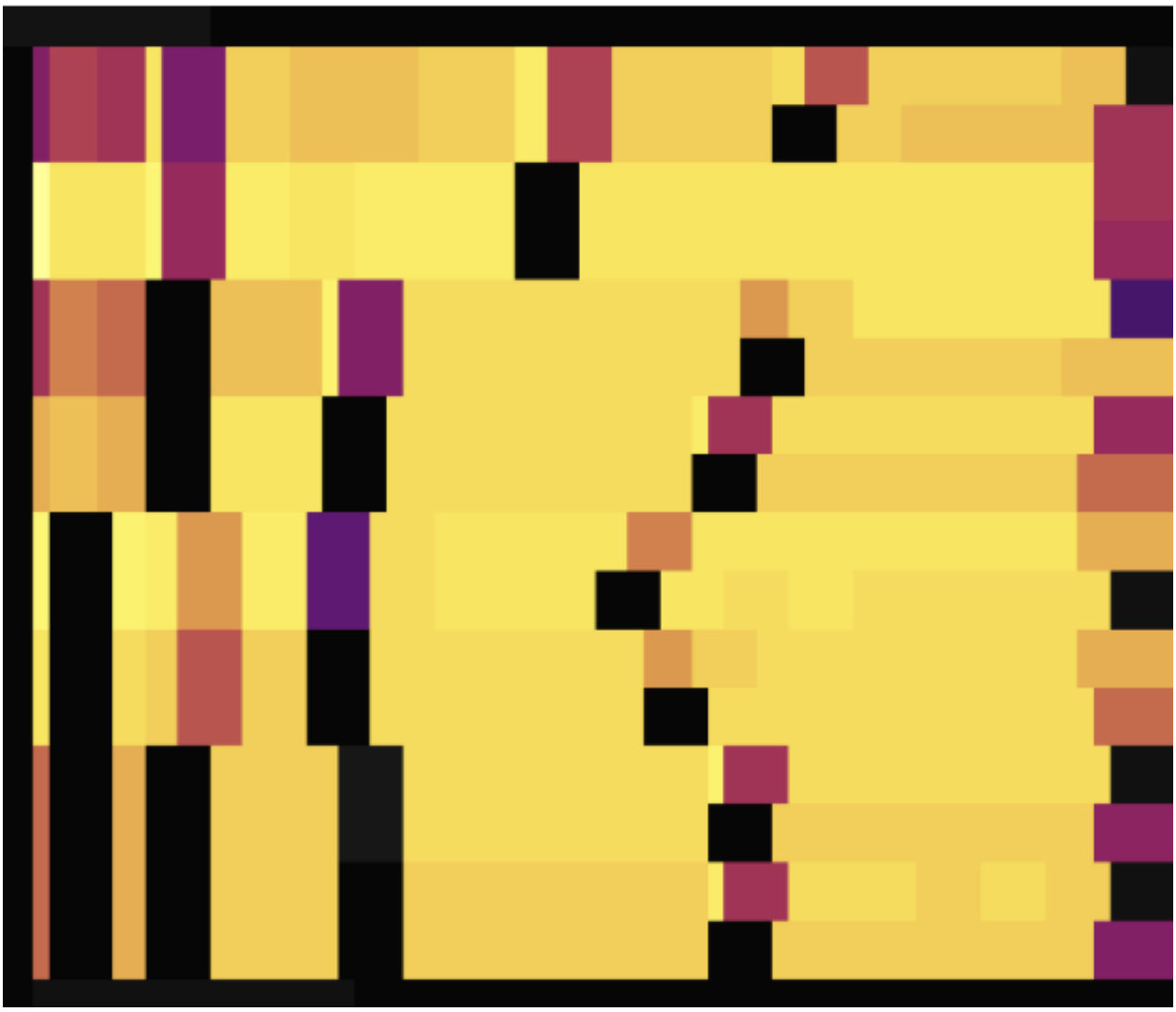


图 70: Dashboard 示例 6

从流量图可见，设置 SHARD_ROW_ID_BITS 后，流量热点由之前的只在一个 Region 上变得很分散。

8.8.5 使用 AUTO_RANDOM 处理自增主键热点表

使用 AUTO_RANDOM 处理自增主键热点表，适用于代替自增主键，解决自增主键带来的写入热点。

使用该功能后，将由 TiDB 生成随机分布且空间耗尽前不重复的主键，达到离散写入、打散写入热点的目的。

注意 TiDB 生成的主键不再是自增的主键，可使用 LAST_INSERT_ID() 获取上次分配的主键值。

将建表语句中的 AUTO_INCREMENT 改为 AUTO_RANDOM 即可使用该功能，适用于主键只需要保证唯一，不包含业务意义的场景。示例如下：

```
CREATE TABLE t (a BIGINT PRIMARY KEY AUTO_RANDOM, b varchar(255));
INSERT INTO t (b) VALUES ("foo");
```

```
SELECT * FROM t;
```

```
+------------+---+
| a          | b |
+------------+---+
| 1073741825 | b |
+------------+---+
```

```
SELECT LAST_INSERT_ID();
```

```
+------------------+
| LAST_INSERT_ID() |
+------------------+
| 1073741825       |
+------------------+
```

以下是将 AUTO_INCREMENT 表改为 AUTO_RANDOM 打散热点后的流量图，第一张是 AUTO_INCREMENT，第二张是 AUTO_RANDOM。

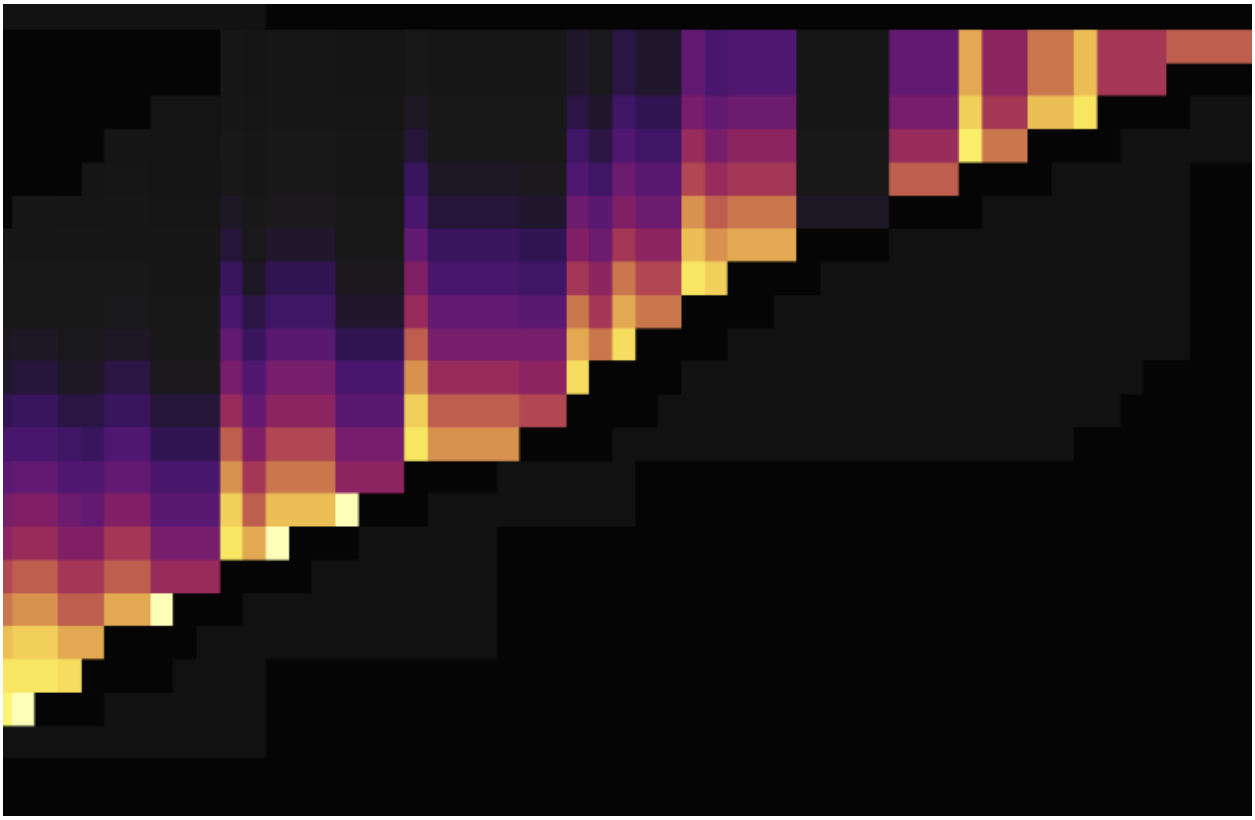


图 71: Dashboard 示例 7

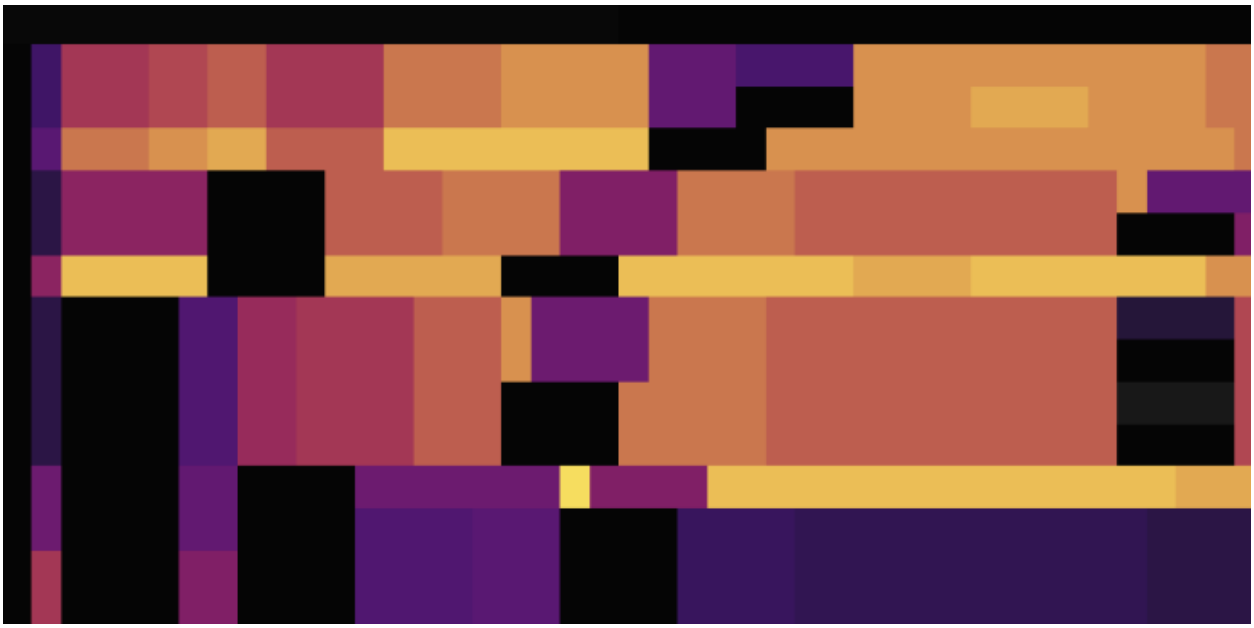


图 72: Dashboard 示例 8

由流量图可见，使用 AUTO_RANDOM 代替 AUTO_INCREMENT 能很好地打散热点。

更详细的说明参见AUTO_RANDOM 文档。

8.8.6 小表热点的优化

TiDB 从 4.0 起引入了 Coprocessor Cache 功能，支持下推计算结果缓存。开启该功能后，将在 TiDB 实例侧缓存下推给 TiKV 计算的结果，对于小表读热点能起到比较好的效果。

更详细的说明参见下推计算结果缓存文档。

其他相关资料：

- TiDB 高并发写入场景最佳实践
- Split Region 使用文档

8.9 读写延迟增加

本文档介绍读写延迟增加、抖动时的排查思路，可能的原因和解决方法。

8.9.1 常见原因

8.9.1.1 TiDB 执行计划不对导致延迟增高

查询语句的执行计划不稳定，偶尔执行计划选择错误的索引，导致查询延迟增加。

现象：

- 如果慢日志中输出了执行计划，可以直接查看执行计划。用 `select tidb_decode_plan('xxx...')` 语句可以解析出具体的执行计划。
- 监控中的 key 扫描异常升高；慢日志中 SQL 执行时间 `Scan Keys` 数目较大。
- SQL 执行时间相比于其他数据库（例如 MySQL）有较大差距。可以对比其他数据库执行计划，例如 `Join Order` 是否不同。

可能的原因：

- 统计信息不准确

解决方案：

- 更新统计信息
  - 手动 `analyze table`，配合 crontab 定期 `analyze`，维持统计信息准确度。
  - 自动 `auto analyze`，调低 `analyze ratio` 阈值，提高收集频次，并设置运行时间窗口。示例如下：
    * `set global tidb_auto_analyze_ratio=0.2;`
    * `set global tidb_auto_analyze_start_time='00:00 +0800';`
    * `set global tidb_auto_analyze_end_time='06:00 +0800';`
- 绑定执行计划
  - 修改业务 SQL，使用 `use index` 固定使用列上的索引。
  - 3.0 版本下，业务可以不用修改 SQL，使用 `create global binding` 创建 `force index` 的绑定 SQL。
  - 4.0 版本支持 SQL Plan Management，可以避免因执行计划不稳定导致的性能下降。

8.9.1.2 PD 异常

现象：

监控中 PD TSO 的 wait duration 异常升高。wait duration 代表从开始等待 PD 返回，到等待结束的时间。

可能的原因：

- 磁盘问题。PD 所在的节点 I/O 被占满，排查是否有其他 I/O 高的组件与 PD 混合部署以及磁盘的健康情况，可通过监控 Grafana -> disk performance -> latency 和 load 等指标进行验证，必要时可以使用 fio 工具对盘进行检测，见案例 case-292。
- PD 之间的网络问题。PD 日志中有 `"lost the TCP streaming connection"`，排查 PD 之间网络是否有问题，可通过监控 Grafana -> PD -> etcd 的 round trip 来验证，见案例 case-177。
- 系统负载高，日志中能看到 `"server is likely overloaded"`，见案例 case-214。
- 选举不出 leader。PD 日志中有 `"lease is not expired"`，见 issues https://github.com/etcd-io/etcd/issues/10355。v3.0.x 版本和 v2.1.19 版本已解决该问题，见案例 case-875。
- 选举慢。Region 加载时间长，从 PD 日志中 `grep "regions cost"`（例如日志中可能是 `load 460927 regions cost 11.77099s`），如果出现秒级，则说明较慢，v3.0 版本可开启 Region Storage（设置 `use-region-storage` 为 `true`），该特性能极大缩短加载 Region 的时间，见案例 case-429。

- TiDB 与 PD 之间的网络问题，应排查网络相关情况。通过监控 Grafana -> blackbox_exporter -> ping latency 确定 TiDB 到 PD leader 的网络是否正常。
- PD 报 FATAL 错误，日志中有 `"range failed to find revision pair"`。v3.0.8 中已经解决问题，见 PR https://github.com/pingcap/pd/pull/2040。详情请参考案例 case-947。
- 使用 `/api/v1/regions` 接口时 Region 数量过多可能会导致 PD OOM。已于 v3.0.8 版本修复，见 https://github.com/pingcap/pd/pull/1986。
- 滚动升级的时候 PD OOM，gRPC 消息大小没限制，监控可看到 TCP InSegs 较大，已于 v3.0.6 版本修复，见 https://github.com/pingcap/pd/pull/1952。
- PD panic。请提交 bug。
- 其他原因，通过 `curl http://127.0.0.1:2379/debug/pprof/goroutine?debug=2` 抓取 goroutine，并提交 bug。

8.9.1.3 TiKV 异常

现象：

监控中 KV Cmd Duration 异常升高。KV Cmd Duration 是 TiDB 发送请求给 TiKV 到收到回复的延迟。

可能的原因：

- 查看 gRPC duration。gRPC duration 是请求在 TiKV 端的总耗时。通过对比 TiKV 的 gRPC duration 以及 TiDB 中的 KV duration 可以发现潜在的网络问题。比如 gRPC duration 很短但是 TiDB 的 KV duration 显示很长，说明 TiDB 和 TiKV 之间网络延迟可能很高，或者 TiDB 和 TiKV 之间的网卡带宽被占满。
- TiKV 重启了导致重新选举
  - TiKV panic 之后又被 `systemd` 重新拉起正常运行，可以通过查看 TiKV 的日志来确认是否有 `panic`，这种情况属于非预期，需要报 bug。
  - 被第三者 `stop/kill`，被 `systemd` 重新拉起。查看 `dmesg` 和 `TiKV log` 确认原因。
  - TiKV 发生 OOM 导致重启了。
  - 动态调整 THP 导致 hung 住，见案例 case-500。
- 查看监控：Grafana -> TiKV-details -> errors 面板 `server is busy` 看到 TiKV RocksDB 出现 write stall 导致发生重新选举。
- TiKV 发生网络隔离导致重新选举。
- `block-cache` 配置太大导致 OOM，在监控 Grafana -> TiKV-details 选中对应的 instance 之后查看 RocksDB 的 `block cache size` 监控来确认是否是该问题。同时请检查 `[storage.block-cache] capacity = # "1GB"` 参数是否设置合理，默认情况下 TiKV 的 `block-cache` 设置为机器总内存的 45%。在容器化部署时需要显式指定该参数，因为 TiKV 获取的是物理机的内存，可能会超出单个 container 的内存限制。
- Coprocessor 收到大量大查询，返回的数据量太大，gRPC 发送速度跟不上 Coprocessor 向客户端输出数据的速度导致 OOM。可以通过检查监控：Grafana -> TiKV-details -> coprocessor overview 的 `response size` 是否超过 `network outbound` 流量来确认是否属于这种情况。

8.9.1.4 TiKV 单线程瓶颈

TiKV 中存在一些单线程线程，可能会成为瓶颈。

- 单个 TiKV Region 过多导致单个 gRPC 线程成为瓶颈（查看 Grafana -> TiKV-details -> Thread CPU/gRPC CPU Per Thread 监控），v3.x 以上版本可以开启 Hibernate Region 特性来解决，见案例 case-612。
- v3.0 之前版本 Raftstore 单线程或者 Apply 单线程到达瓶颈（Grafana -> TiKV-details -> Thread CPU/raft store CPU 和 Async apply CPU 超过 80%），可以选择扩容 TiKV（v2.x 版本）实例或者升级到多线程模型的 v3.x 版本。

8.9.1.5 CPU Load 升高

现象：

CPU 资源使用到达瓶颈

可能的原因：

- 热点问题。* 整体负载高，排查 TiDB 的 slow query 和 expensive query。对运行的 query 进行优化，如果缺索引就加索引，如果可以批量执行就批量执行。另一个方案是对集群进行扩容。

8.9.2 其它原因

8.9.2.1 集群维护

通常大多数的线上集群有 3 或 5 个 PD 节点，如果维护的主机上有 PD 组件，需要具体考虑节点是 leader 还是 follower，关闭 follower 对集群运行没有任何影响，关闭 leader 需要先切换，并在切换时有 3 妙左右的性能抖动。

8.9.2.2 少数派副本离线

TiDB 集群默认配置为 3 副本，每一个 Region 都会在集群中保存 3 份，它们之间通过 Raft 协议来选举 Leader 并同步数据。Raft 协议可以保证在数量小于副本数（注意：不是节点数）一半的节点挂掉或者隔离的情况下，仍然能够提供服务，并且不丢失任何数据。对于 3 副本集群，挂掉一个节点可能会导致性能抖动，可用性和正确性理论上不会受影响。

8.9.2.3 新增索引

由于创建索引在扫表回填索引的时候会消耗大量资源，甚至与一些频繁更新的字段会发生冲突导致正常业务受到影响。大表创建索引的过程往往会持续很长时间，所以要尽可能地平衡执行时间和集群性能之间的关系，比如选择非高频更新时间段。

参数调整：

目前主要使用 `tidb_ddl_reorg_worker_cnt` 和 `tidb_ddl_reorg_batch_size` 这两个参数来动态调整索引创建速度，通常来说它们的值越小对系统影响越小，但是执行时间越长。

一般情况下，先将值保持为默认的 4 和 256，观察集群资源使用情况和响应速度，再逐渐调大 `tidb_ddl_reorg_worker_cnt` 参数来增加并发，观察监控如果系统没有发生明显的抖动，再逐渐调大 `tidb_ddl_reorg_batch_size` 参数，但如果索引涉及的列更新很频繁的话就会造成大量冲突造成失败重试。

另外还可以通过调整参数 `tidb_ddl_reorg_priority` 为 `PRIORITY_HIGH` 来让创建索引的任务保持高优先级来提升速度，但在通用 OLTP 系统上，一般建议保持默认。

8.9.2.4 GC 压力大

TiDB 的事务的实现采用了 MVCC（多版本并发控制）机制，当新写入的数据覆盖旧的数据时，旧的数据不会被替换掉，而是与新写入的数据同时保留，并以时间戳来区分版本。GC 的任务便是清理不再需要的旧数据。

- Resolve Locks 阶段在 TiKV 一侧会产生大量的 `scan_lock` 请求，可以在 gRPC 相关的 metrics 中观察到。`scan_lock` 请求会对全部的 Region 调用。
- Delete Ranges 阶段会往 TiKV 发送少量的 `unsafe_destroy_range` 请求，也可能没有。可以在 gRPC 相关的 metrics 中和 GC 分类下的 GC tasks 中观察到。
- Do GC 阶段，默认每台 TiKV 会自动扫描本机上的 leader Region 并对每一个 leader 进行 GC，这一活动可以在 GC 分类下的 GC tasks 中观察到。

8.10 乐观事务模型下写写冲突问题排查

本文介绍 TiDB 中乐观锁下写写冲突出现的原因以及解决方案。

在 v3.0.8 版本之前，TiDB 默认采用乐观事务模型，在事务执行过程中并不会做冲突检测，而是在事务最终 COMMIT 提交时触发两阶段提交，并检测是否存在写写冲突。当出现写写冲突，并且开启了事务重试机制，则 TiDB 会在限定次数内进行重试，最终重试成功或者达到重试次数上限后，会给客户端返回结果。因此，如果 TiDB 集群中存在大量的写写冲突情况，容易导致集群的 Duration 比较高。

8.10.1 出现写写冲突的原因

TiDB 中使用 Percolator 事务模型来实现 TiDB 中的事务。Percolator 总体上就是一个二阶段提交的实现。具体的二阶段提交过程可参考乐观事务文档。

当客户端发起 `COMMIT` 请求的时候，TiDB 开始两阶段提交：

1. TiDB 从所有要写入的 Key 中选择一个作为当前事务的 Primary Key
2. TiDB 向所有的本次提交涉及到的 TiKV 发起 prewrite 请求，TiKV 判断是否所有 Key 都可以 prewrite 成功
3. TiDB 收到所有 Key 都 prewrite 成功的消息
4. TiDB 向 PD 请求 commit_ts
5. TiDB 向 Primary Key 发起第二阶段提交。Primary Key 所在的 TiKV 收到 commit 操作后，检查数据合法性，清理 prewrite 阶段留下的锁
6. TiDB 收到两阶段提交成功的信息

写写冲突发生在 prewrite 阶段，当发现有其他的事务在写当前 Key (data.commit_ts > txn.start_ts)，则会发生写写冲突。

TiDB 会根据 `tidb_disable_txn_auto_retry` 和 `tidb_retry_limit` 参数设置的情况决定是否进行重试，如果设置了不重试，或者重试次数达到上限后还是没有 prewrite 成功，则向 TiDB 返回 `Write Conflict` 错误。

8.10.2 如何判断当前集群存在写写冲突

可以通过 Grafana 监控查看集群写写冲突的情况：

- 通过 TiDB 监控面板中 KV Errors 监控栏中 KV Backoff OPS 监控指标项，查看 TiKV 中返回错误信息的数量

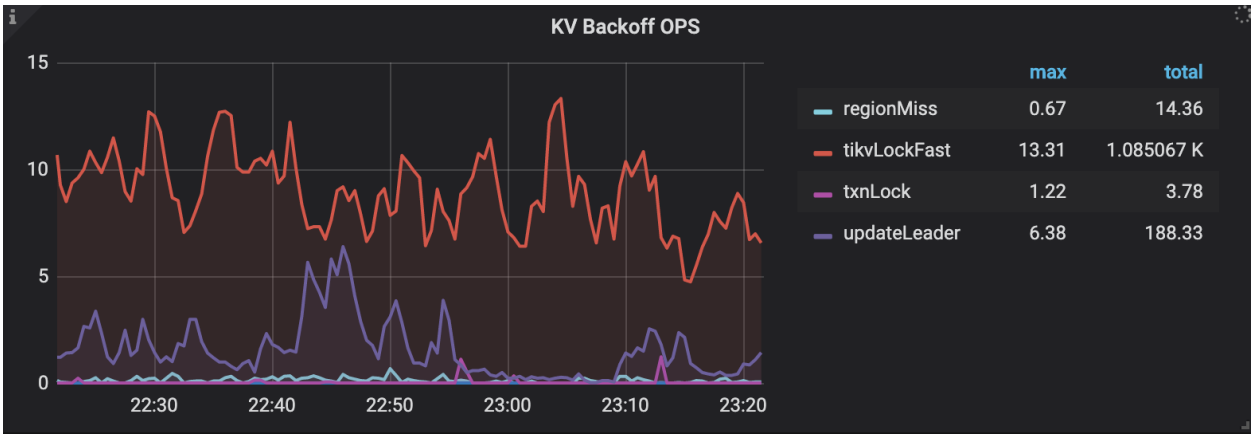


图 73: kv-backoff-ops

txnlock 表示集群中存在写写冲突，txnLockFast 表示集群中存在读写冲突。

- 通过 TiDB 监控面板中 KV Errors 监控栏中 Lock Resolve OPS 监控指标项，查看事务冲突相关的数量

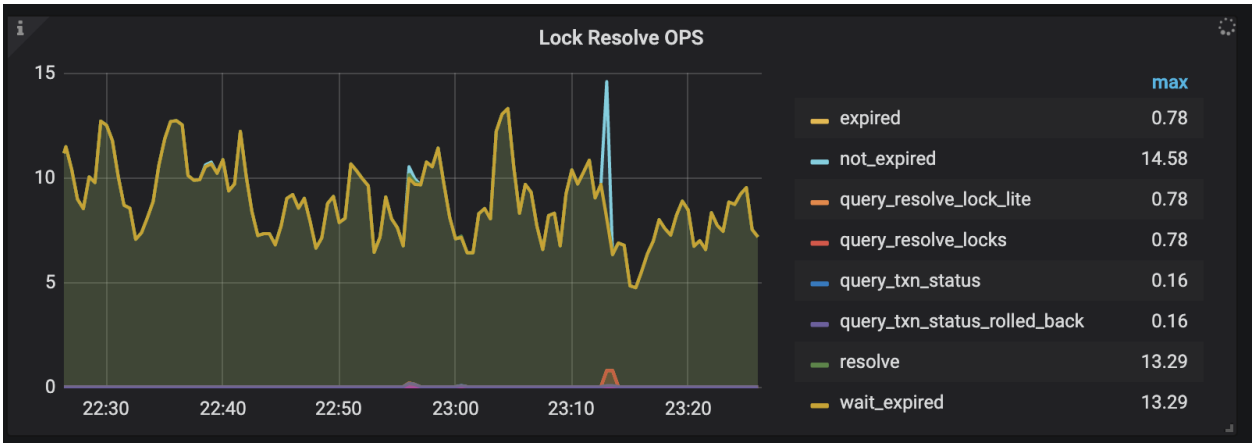


图 74: lock-resolve-ops

expired、 not_expired、 wait_expired 表示对应的 lock 状态

- 通过 TiDB 监控面板中 KV Errors 监控栏中 KV Retry Duration 监控指标项，查看 KV 重试请求的时间

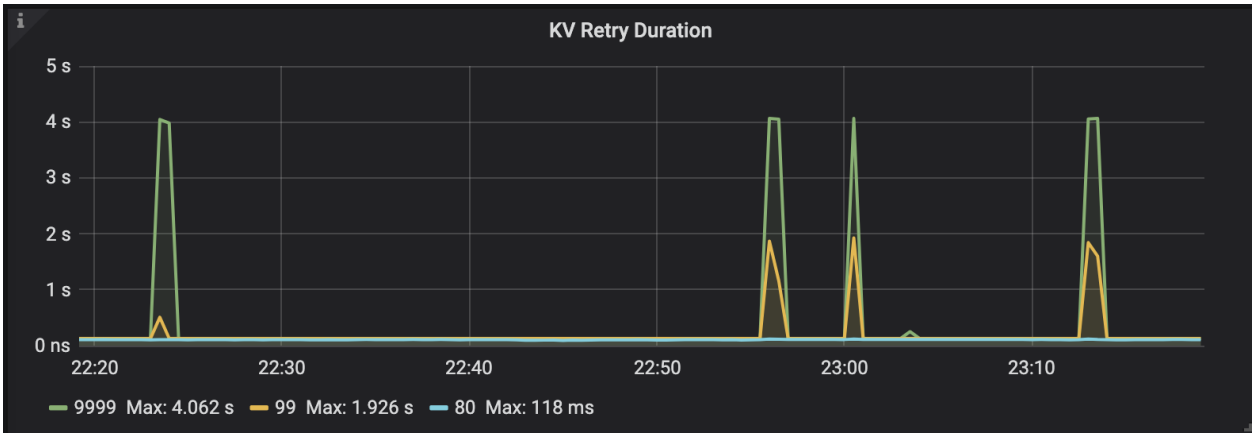


图 75: kv-retry-duration

也可以通过 TiDB 日志查看是否有 `[kv:9007]Write conflict` 关键字，如果搜索到对应关键字，则可以表明集群中存在写写冲突。

8.10.3 如何解决写写冲突问题

如果通过以上方式判断出集群中存在大量的写写冲突，建议找到冲突的数据，以及写写冲突的原因，看是否能从应用程序修改逻辑，加上重试的逻辑。当出现写写冲突的时候，可以在 TiDB 日志中看到类似的日志：

```
[2020/05/12 15:17:01.568 +08:00] [WARN] [session.go:446] ["commit failed"] [conn=3] ["finished
    ↪ txn"="Txn{state=invalid}"] [error="[kv:9007]Write conflict, txnStartTS
    ↪ =416617006551793665, conflictStartTS=416617018650001409, conflictCommitTS
    ↪ =416617023093080065, key={tableID=47, indexID=1, indexValues={string, }} primary={tableID
    ↪ =47, indexID=1, indexValues={string, }} [try again later]"]
```

关于日志的解释如下：

- `[kv:9007]Write conflict`：表示出现了写写冲突
- `txnStartTS=416617006551793665`：表示当前事务的 start_ts 时间戳，可以通过 pd-ctl 工具将时间戳转换为具体时间
- `conflictStartTS=416617018650001409`：表示冲突事务的 start_ts 时间戳，可以通过 pd-ctl 工具将时间戳转换为具体时间
- `conflictCommitTS=416617023093080065`：表示冲突事务的 commit_ts 时间戳，可以通过 pd-ctl 工具将时间戳转换为具体时间
- `key={tableID=47, indexID=1, indexValues={string, }}`：表示当前事务中冲突的数据，tableID 表示发生冲突的表的 ID，indexID 表示是索引数据发生了冲突。如果是数据发生了冲突，会打印 handle=x 表示对应哪行数据发生了冲突，indexValues 表示发生冲突的索引数据
- `primary={tableID=47, indexID=1, indexValues={string, }}`：表示当前事务中的 Primary Key 信息

通过 pd-ctl 将时间戳转换为可读时间：

```
./pd-ctl -u https://127.0.0.1:2379 tso {TIMESTAMP}
```

通过 tableID 查找具体的表名：

```
curl http://{TiDBIP}:10080/db-table/{tableID}
```

通过 indexID 查找具体的索引名：

```
SELECT * FROM INFORMATION_SCHEMA.TIDB_INDEXES WHERE TABLE_SCHEMA='{table_name}' AND TABLE_NAME='{
    ↪ table_name}' AND INDEX_ID={indexID};
```

另外在 v3.0.8 及之后版本默认使用悲观事务模式，从而避免在事务提交的时候因为冲突而导致失败，无需修改应用程序。悲观事务模式下会在每个 DML 语句执行的时候，加上悲观锁，用于防止其他事务修改相同 Key，从而保证在最后提交的 prewrite 阶段不会出现写写冲突的情况。

8.11 TiDB 磁盘 I/O 过高的处理办法

本文主要介绍如何定位和处理 TiDB 存储 I/O 过高的问题。

8.11.1 确认当前 I/O 指标

当出现系统响应变慢的时候，如果已经排查了 CPU 的瓶颈、数据事务冲突的瓶颈后，就需要从 I/O 来入手来辅助判断目前的系统瓶颈点。

8.11.1.1 从监控定位 I/O 问题

最快速的定位手段是从监控来查看整体的 I/O 情况，可以从集群部署工具 (TiDB-Ansible, TiUP) 默认会部署的监控组件 Grafana 来查看对应的 I/O 监控，跟 I/O 相关的 Dashboard 有 `Overview`，`Node_exporter`，`Disk-Performance`。

8.11.1.1.1 第一类面板

在 `Overview` > `System Info` > `IO Util` 中，可以看到集群中每个机器的 I/O 情况，该指标和 Linux iostat 监控中的 util 类似，百分比越高代表磁盘 I/O 占用越高：

- 如果监控中只有一台机器的 I/O 高，那么可以辅助判断当前有读写热点。
- 如果监控中大部分机器的 I/O 都高，那么集群现在有高 I/O 负载存在。

如果发现某台机器的 I/O 比较高，可以从监控 `Disk-Performance Dashboard` 进一步观察 I/O 的使用情况，结合 `Disk Latency`，`Disk Load` 等 metric 判断是否存在异常，必要时可以使用 fio 工具对磁盘进行检测。

8.11.1.1.2 第二类面板

TiDB 集群主要的持久化组件是 TiKV 集群，一个 TiKV 包含两个 RocksDB 实例：一个用于存储 Raft 日志，位于 data/raft，一个用于存储真正的数据，位于 data/db。

在 `TiKV-Details` > `Raft IO` 中，可以看到这两个实例磁盘写入的相关 metric：

- `Append log duration`：该监控表明了存储 Raft 日志的 RocksDB 写入的响应时间，.99 响应应该在 50ms 以内。

- `Apply log duration`：该监控表明了存储真正数据的 RocksDB 写入的响应时间，.99 响应应该在 100ms 以内。

这两个监控还有 `.. per server` 的监控面板来提供辅助查看热点写入的情况。

8.11.1.1.3 第三类面板

在 `TiKV-Details` > `Storage` 中，有关于 storage 相关情况的监控：

- `Storage command total`：收到的不同命令的个数。
- `Storage async write duration`：包括了磁盘 sync duration 等监控项，可能和 Raft IO 有关。如遇到异常情况，需要通过 log 来检查相关组件的工作状态是否正常。

8.11.1.1.4 其他面板

此外，可能还需要一些其它内容来辅助确认瓶颈是否为 I/O，并可以尝试调整一些参数。通过查看 TiKV gRPC 的 prewrite/commit/raw-put（仅限 raw kv 集群）duration，确认确实是 TiKV 写入慢了。常见的几种情况如下：

- append log 慢。TiKV Grafana 的 Raft I/O 和 append log duration 比较高，通常情况下是由于写盘慢了，可以检查 RocksDB - raft 的 WAL Sync Duration max 值来确认，否则可能需要报 bug。
- raftstore 线程繁忙。TiKV grafana 的 Raft Propose/propose wait duration 明显高于 append log duration。请查看以下两点：
  - `[raftstore]` 的 `store-pool-size` 配置是否过小（该值建议在 [1,5] 之间，不建议太大）。
  - 机器的 CPU 是不是不够了。
- apply log 慢。TiKV Grafana 的 Raft I/O 和 apply log duration 比较高，通常会伴随着 Raft Propose/apply wait duration 比较高。可能的情况如下：
  - `[raftstore]` 的 `apply-pool-size` 配置过小（建议在 [1, 5] 之间，不建议太大），Thread CPU/apply cpu 比较高；
  - 机器的 CPU 资源不够了。
  - Region 写入热点问题，单个 apply 线程 CPU 使用率比较高（通过修改 Grafana 表达式，加上 by (instance, name) 来看各个线程的 cpu 使用情况），暂时对于单个 Region 的热点写入没有很好的方式，最近在优化该场景。
  - 写 RocksDB 比较慢，RocksDB kv/max write duration 比较高（单个 Raft log 可能包含很多个 kv，写 RocksDB 的时候会把 128 个 kv 放在一个 write batch 写入到 RocksDB，所以一次 apply log 可能涉及到多次 RocksDB 的 write）。
  - 其他情况，需要报 bug。
- raft commit log 慢。TiKV Grafana 的 Raft I/O 和 commit log duration 比较高（4.x 版本的 Grafana 才有该 metric）。每个 Region 对应一个独立的 Raft group，Raft 本身是有流控机制的，类似 TCP 的滑动窗口机制，通过参数raftstore raft-max-inflight-msgs = 256 来控制滑动窗口的大小，如果有热点写入并且 commit log duration 比较高可以适度调大改参数，比如 1024。

8.11.1.2 从 log 定位 I/O 问题

- 如果客户端报 `server is busy` 错误，特别是 `raftstore is busy` 的错误信息，会和 I/O 有相关性。

  可以通过查看监控：grafana -> TiKV -> errors 监控确认具体 busy 原因。其中，`server is busy` 是 TiKV 自身的流控机制，TiKV 通过这种方式告知 `tidb/ti-client` 当前 TiKV 的压力过大，过一会儿再尝试。

- TiKV RocksDB 日志出现 `write stall`。

  可能是 `level0 sst` 太多导致 stall。可以添加参数 `[rocksdb] max-sub-compactI/Ons = 2`（或者 `3`）加快 level0 sst 往下 compact 的速度，该参数的意思是将从 level0 到 level1 的 compaction 任务最多切成 `max-sub-compactions` 个子任务交给多线程并发执行。

  如果磁盘 I/O 能力持续跟不上写入，建议扩容。如果磁盘的吞吐达到了上限（例如 SATA SSD 的吞吐相对 NVME SSD 会低很多）导致 write stall，但是 CPU 资源又比较充足，可以尝试采用压缩率更高的压缩算法来缓解磁盘的压力，用 CPU 资源换磁盘资源。

  比如 `default cf compaction` 压力比较大时，可以调整参数 `[rocksdb.defaultcf] compression-per-level = ["no", "no", "lz4", "lz4", "lz4", "zstd", "zstd"]` 改成 `compression-per-level = ["no", "no", "zstd", "zstd", "zstd", "zstd", "zstd"]`。

8.11.1.3 从告警发现 I/O 问题

集群部署工具 (TiDB-Ansible, TiUP) 默认部署的告警组件，官方已经预置了相关的告警项目和阈值，I/O 相关项包括：

- TiKV_write_stall
- TiKV_raft_log_lag
- TiKV_async_request_snapshot_duration_seconds
- TiKV_async_request_write_duration_seconds
- TiKV_raft_append_log_duration_secs
- TiKV_raft_apply_log_duration_secs

8.11.2 I/O 问题处理方案

1. 当确认为热点 I/O 问题的时候，需要参考TiDB 热点问题处理来消除相关的热点 I/O 情况。
2. 当确认整体 I/O 已经到达瓶颈的时候，且从业务侧能够判断 I/O 的能力会持续的跟不上，那么就可以利用分布式数据库的 scale 的能力，采用扩容 TiKV 节点数量的方案来获取更大的整体 I/O 吞吐量。
3. 调整上述说明中的一些参数，使用计算/内存资源来换取磁盘的存储资源。

8.12 TiDB 锁冲突问题处理

TiDB 支持完整的分布式事务，自 v3.0 版本起，提供乐观事务与悲观事务两种事务模型。本文介绍在使用乐观事务或者悲观事务的过程中常见的锁冲突问题以及解决思路。

8.12.1 乐观锁

TiDB 中事务使用两阶段提交，分为 Prewrite 和 Commit 两个阶段，示意图如下。

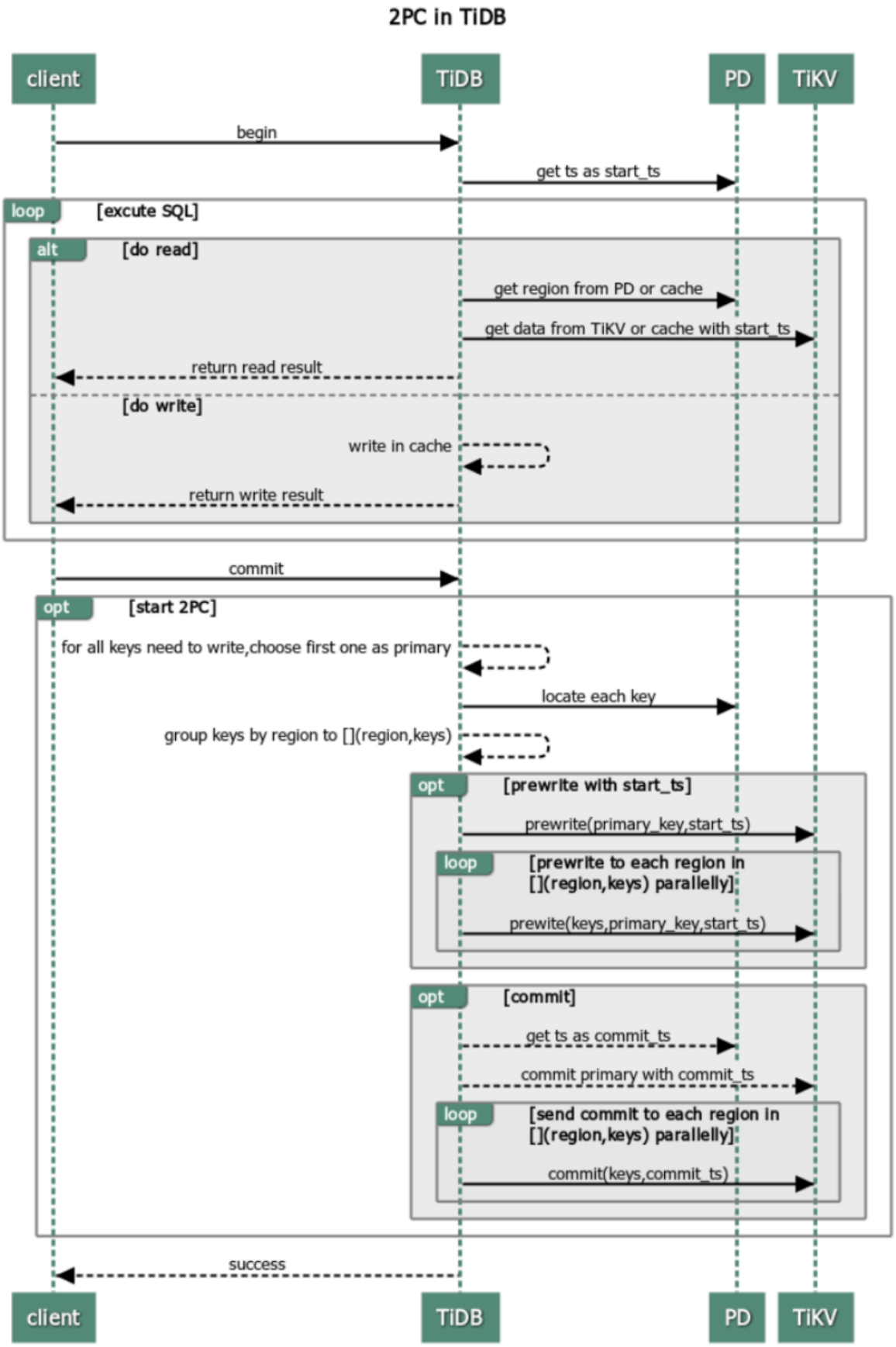


图 76: TiDB 中乐观事务的两阶段提交

相关细节本节不再赘述，详情可阅读 Percolator 和 TiDB 事务算法。

8.12.1.1 Prewrite 阶段

在两阶段提交的 Prewrite 阶段，TiDB 会对目标 key 分别上 primary lock 和 secondary lock。在冲突严重的场景中，会出现写冲突 (write conflict)、keyislocked 等报错。具体而言，这个阶段可能会遇到的锁相关的报错信息如下。

8.12.1.1.1 读写冲突

在 TiDB 中，读取数据时，会获取一个包含当前物理时间且全局唯一递增的时间戳作为当前事务的 start_ts。事务在读取时，需要读到目标 key 的 commit_ts 小于这个事务的 start_ts 的最新的数据版本。当读取时发现目标 key 上存在 lock 时，因为无法知道上锁的那个事务是在 Commit 阶段还是 Prewrite 阶段，所以就会出现读写冲突的情况，如下图：

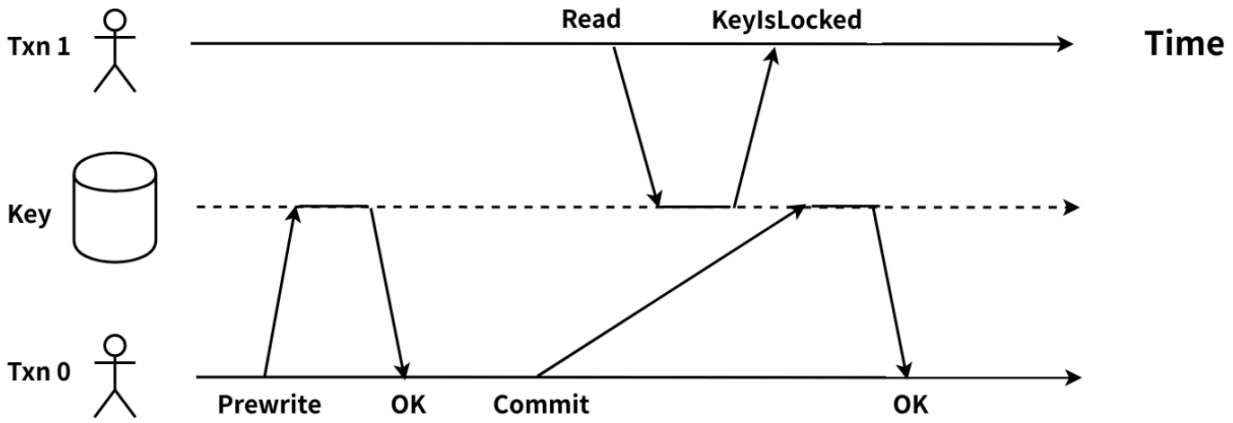


图 77: 读写冲突

分析：

Txn0 完成了 Prewrite，在 Commit 的过程中 Txn1 对该 key 发起了读请求，Txn1 需要读取 start_ts > commit_ts 最近的 key 的版本。此时，Txn1 的 `start_ts` > Txn0 的 lock_ts，需要读取的 key 上的锁信息仍未清理，故无法判断 Txn0 是否提交成功，因此 Txn1 与 Txn0 出现读写冲突。

你可以通过如下两种途径来检测当前环境中是否存在读写冲突：

1. TiDB 监控及日志

    - 通过 TiDB Grafana 监控分析：

      观察 KV Errors 下 Lock Resolve OPS 面板中的 not_expired/resolve 监控项以及 KV Backoff OPS 面板中的 txnLockFast 监控项，如果有较为明显的上升趋势，那么可能是当前的环境中出现了大量的读写冲突。其中，not_expired 是指对应的锁还没有超时，resolve 是指尝试清锁的操作，txnLockFast 代表出现了读写冲突。

- 通过 TiDB 日志分析：

  在 TiDB 的日志中可以看到下列信息：

```
[INFO] [coprocessor.go:743] ["[TIME_COP_PROCESS] resp_time:406.038899ms txnStartTS
    ↪ :416643508703592451 region_id:8297 store_addr:10.8.1.208:20160 backoff_ms:255
    ↪ backoff_types:[txnLockFast,txnLockFast] kv_process_ms:333 scan_total_write:0
    ↪ scan_processed_write:0 scan_total_data:0 scan_processed_data:0 scan_total_lock:0
    ↪  scan_processed_lock:0"]
```

  - txnStartTS：发起读请求的事务的 start_ts，如上面示例中的 416643508703592451
  - backoff_types：读写发生了冲突，并且读请求进行了 backoff 重试，重试的类型为 txnLockFast
  - backoff_ms：读请求 backoff 重试的耗时，单位为 ms ，如上面示例中的 255
  - region_id：读请求访问的目标 region 的 id

2. 通过 TiKV 日志分析：

   在 TiKV 的日志可以看到下列信息：

```
[ERROR] [endpoint.rs:454] [error-response] [err=""locked primary_lock:7480000000000004
    ↪ D35F698000000000000001038000000000004C788E0380000000004C0748 lock_version:
    ↪ 411402933858205712 key: 7480000000000004D35F7280000000004C0748 lock_ttl: 3008
    ↪ txn_size: 1""]
```

这段报错信息表示出现了读写冲突，当读数据时发现 key 有锁阻碍读，这个锁包括未提交的乐观锁和未提交的 prewrite 后的悲观锁。

- primary_lock：锁对应事务的 primary lock。
- lock_version：锁对应事务的 start_ts。
- key：表示被锁的 key。
- lock_ttl: 锁的 TTL。
- txn_size：锁所在事务在其 Region 的 key 数量，指导清锁方式。

处理建议：

- 在遇到读写冲突时会有 backoff 自动重试机制，如上述示例中 Txn1 会进行 backoff 重试，单次初始 100 ms，单次最大 3000 ms，总共最大 20000 ms
- 可以使用 TiDB Control 的子命令decoder 来查看指定 key 对应的行的 table id 以及 rowid：

```
./tidb-ctl decoder -f table_row -k "t\x00\x00\x00\x00\x00\x00\x00\x1c_r\x00\x00\x00\x00\x00\
    ↪ x00\x00\xfa"

table_id: -9223372036854775780
row_id: -9223372036854775558
```

8.12.1.1.2 KeyIsLocked 错误

事务在 Prewrite 阶段的第一步就会检查是否有写写冲突，第二步会检查目标 key 是否已经被另一个事务上锁。当检测到该 key 被 lock 后，会在 TiKV 端报出 KeyIsLocked。目前该报错信息没有打印到 TiDB 以及 TiKV 的日志中。与读写冲突一样，在出现 KeyIsLocked 时，后台会自动进行 backoff 重试。

你可以通过 TiDB Grafana 监控检测 KeyIsLocked 错误：

观察 KV Errors 下 Lock Resolve OPS 面板中的 resolve 监控项以及 KV Backoff OPS 面板中的 txnLock 监控项，会有比较明显的上升趋势，其中 resolve 是指尝试清锁的操作，txnLock 代表出现了写冲突。

处理建议：

- 监控中出现少量 txnLock，无需过多关注。后台会自动进行 backoff 重试，单次初始 200 ms，单次最大 3000 ms。
- 如果出现大量的 txnLock，需要从业务的角度评估下冲突的原因。
- 使用悲观锁模式。

8.12.1.2 Commit 阶段

当 Prewrite 全部完成时，客户端便会取得 commit_ts，然后继续两阶段提交的第二阶段。这里需要注意的是，由于 primary key 是否提交成功标志着整个事务是否提交成功，因而客户端需要在单独 commit primary key 之后再继续 commit 其余的 key。

8.12.1.2.1 锁被清除 (LockNotFound) 错误

TxnLockNotFound 错误是由于事务提交的慢了，超过了 TTL 的时间。当要提交时，发现被其他事务给 Rollback 掉了。在开启 TiDB 自动重试事务的情况下，会自动在后台进行事务重试（注意显示和隐式事务的差别）。

你可以通过如下两种途径来查看 LockNotFound 报错信息：

1. 查看 TiDB 日志

   如果出现了 TxnLockNotFound 的报错，会在 TiDB 的日志中看到下面的信息：

```
[WARN] [session.go:446] ["commit failed"] [conn=149370] ["finished txn"="Txn{state=invalid
    ↪ }"] [error="[kv:6]Error: KV error safe to retry tikv restarts txn: Txn(Mvcc(
    ↪ TxnLockNotFound{ start_ts: 412720515987275779, commit_ts: 412720519984971777, key:
    ↪ [116, 128, 0, 0, 0, 0, 1, 111, 16, 95, 114, 128, 0, 0, 0, 0, 0, 2] })) [try again
    ↪  later]"]
```

   - start_ts：出现 TxnLockNotFound 报错的事务的 start_ts，如上例中的 412720515987275779
   - commit_ts：出现 TxnLockNotFound 报错的事务的 commit_ts，如上例中的 412720519984971777

2. 查看 TiKV 日志

   如果出现了 TxnLockNotFound 的报错，在 TiKV 的日志中同样可以看到相应的报错信息：

```
Error: KV error safe to retry restarts txn: Txn(Mvcc(TxnLockNotFound)) [ERROR [Kv.rs:708] ["
    ↪ KvService::batch_raft send response fail"] [err=RemoteStoped]
```

处理建议：

- 通过检查 start_ts 和 commit_ts 之间的提交间隔，可以确认是否超过了默认的 TTL 的时间。

  查看提交间隔：

```
./pd-ctl tso [start_ts]
./pd-ctl tso [commit_ts]
```

- 建议检查下是否是因为写入性能的缓慢导致事务提交的效率差，进而出现了锁被清除的情况。
- 在关闭 TiDB 事务重试的情况下，需要在应用端捕获异常，并进行重试。

8.12.2 悲观锁

在 v3.0.8 之前，TiDB 默认使用的乐观事务模式会导致事务提交时因为冲突而失败。为了保证事务的成功率，需要修改应用程序，加上重试的逻辑。悲观事务模式可以避免这个问题，应用程序无需添加重试逻辑，就可以正常执行。

TiDB 悲观锁复用了乐观锁的两阶段提交逻辑，重点在 DML 执行时做了改造。

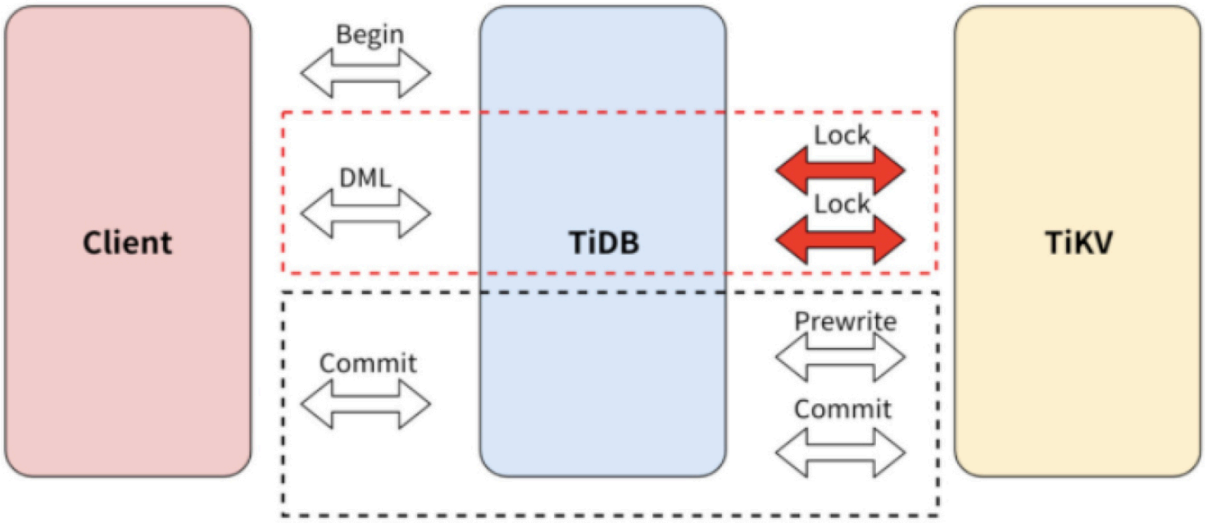


图 78: TiDB 悲观事务的提交逻辑

在两阶段提交之前增加了 Acquire Pessimistic Lock 阶段，简要步骤如下。

1. （同乐观锁）TiDB 收到来自客户端的 begin 请求，获取当前版本号作为本事务的 StartTS。
2. TiDB 收到来自客户端的更新数据的请求：TiDB 向 TiKV 发起加悲观锁请求，该锁持久化到 TiKV。
3. （同乐观锁）客户端发起 commit，TiDB 开始执行与乐观锁一样的两阶段提交。

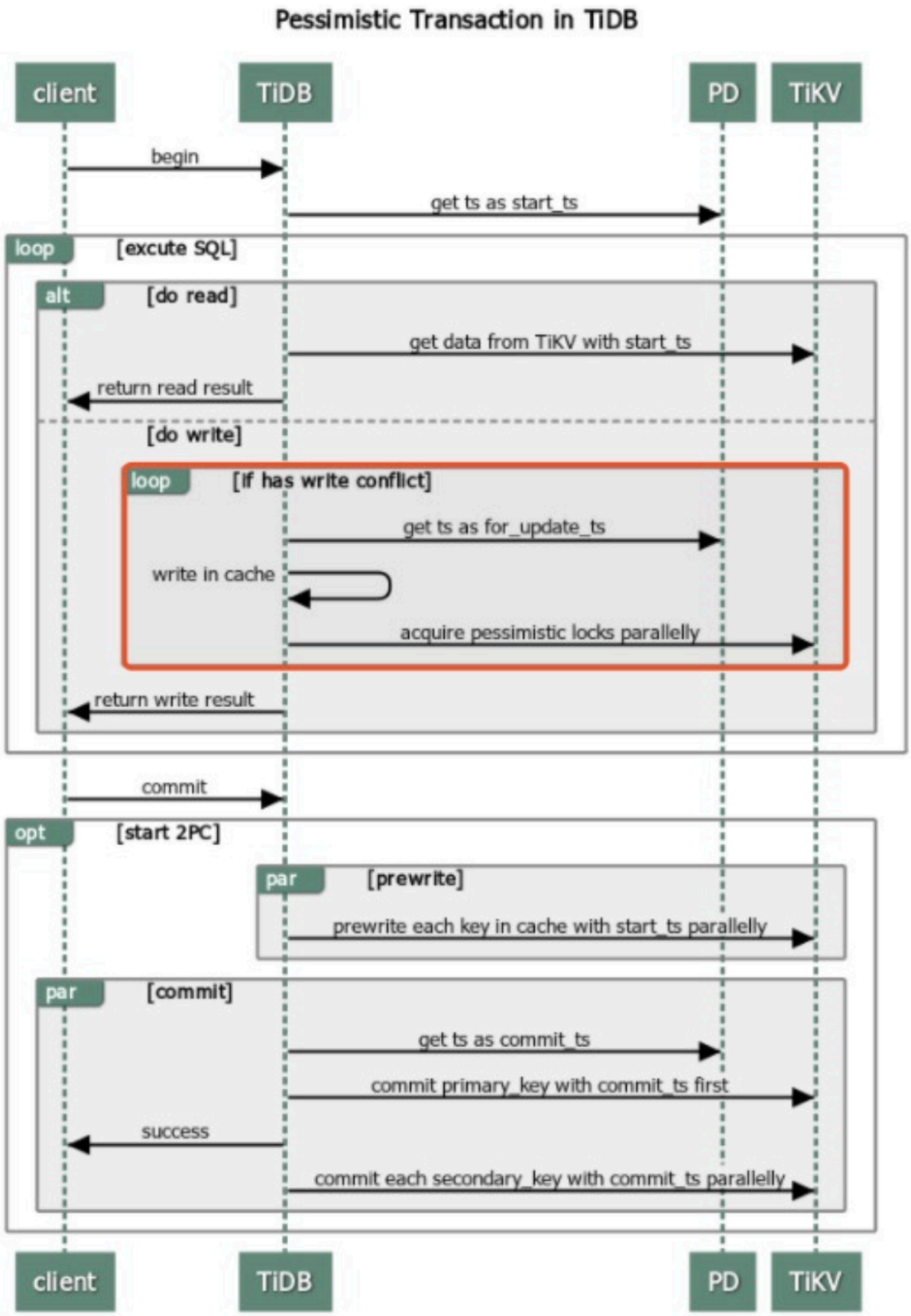


图 79: TiDB 中的悲观事务

相关细节本节不再赘述，详情可阅读 TiDB 悲观锁实现原理。

8.12.2.1 Prewrite 阶段

在悲观锁模式下，在事务的提交阶段沿用的仍然是乐观锁模式，所以在 Prewrite 阶段乐观锁遇到的锁相关的一些报错，在悲观锁模式同样会遇到。

8.12.2.1.1 读写冲突

报错信息以及处理建议同乐观锁模式。

8.12.2.2 Commit 阶段

在乐观模型下，会出现 TxnLockNotFound 错误，而在悲观锁模型下，不会出现这个问题。同样的，悲观锁也有一个 TTL 的时间。txn heartbeat 会自动的更新事务的 TTL，以确保第二个事务不会将第一个事务的锁清掉。

8.12.2.3 其他锁相关错误

8.12.2.3.1 pessimistic lock retry limit reached

在冲突非常严重的场景下，或者当发生 write conflict 时，乐观事务会直接终止，而悲观事务会尝试用最新数据重试该语句直到没有 write conflict。因为 TiDB 的加锁操作是一个写入操作，且操作过程是先读后写，需要 2 次 RPC。如果在这中间发生了 write conflict，那么会重试。每次重试都会打印日志，不用特别关注。重试次数由pessimistic-txn.max-retry-count 定义。

可通过查看 TiDB 日志查看报错信息：

悲观事务模式下，如果发生 write conflict，并且重试的次数达到了上限，那么在 TiDB 的日志中会出现含有下述关键字的报错信息。如下：

```
err="pessimistic lock retry limit reached"
```

处理建议：

- 如果上述报错出现的比较频繁，建议从业务的角度进行调整。

8.12.2.3.2 Lock wait timeout exceeded

在悲观锁模式下，事务之间出现会等锁的情况。等锁的超时时间由 TiDB 的innodb_lock_wait_timeout 参数来定义，这个是 SQL 语句层面的最大允许等锁时间，即一个 SQL 语句期望加锁，但锁一直获取不到，超过这个时间，TiDB 不会再尝试加锁，会向客户端返回相应的报错信息。

可通过查看 TiDB 日志查看报错信息：

当出现等锁超时的情况时，会向客户端返回下述报错信息：

```
ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction
```

处理建议：

- 如果出现的次数非常频繁，建议从业务逻辑的角度来进行调整。

8.12.2.3.3 TTL manager has timed out

除了有不能超出 GC 时间的限制外，悲观锁的 TTL 有上限，默认为 10 分钟，所以执行时间超过 10 分钟的悲观事务有可能提交失败。这个超时时间由 TiDB 参数 performance.max-txn-ttl 指定。

可通过查看 TiDB 日志查看报错信息：

当悲观锁的事务执行时间超过 TTL 时，会出现下述报错：

```
TTL manager has timed out, pessimistic locks may expire, please commit or rollback this
    ↪ transaction
```

处理建议：

- 当遇到该报错时，建议确认下业务逻辑是否可以进行优化，如将大事务拆分为小事务。在未使用大事务的前提下，大事务可能会触发 TiDB 的事务限制。
- 可适当调整相关参数，使其符合事务要求。

8.12.2.3.4 Deadlock found when trying to get lock

死锁是指两个或两个以上的事务在执行过程中，由于竞争资源而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去，将永远在互相等待。此时，需要终止其中一个事务使其能够继续推进下去。

TiDB 在使用悲观锁的情况下，多个事务之间出现了死锁，必定有一个事务 abort 来解开死锁。在客户端层面行为和 MySQL 一致，在客户端返回表示死锁的 Error 1213。如下：

```
[err="[executor:1213]Deadlock found when trying to get lock; try restarting transaction"]
```

处理建议：

- 如果出现非常频繁，需要调整业务代码来降低死锁发生概率。

8.13 TiFlash 常见问题

本文介绍了一些 TiFlash 常见问题、原因及解决办法。

8.13.1 TiFlash 未能正常启动

该问题可能由多个因素构成，可以通过以下步骤依次排查：

1. 检查系统环境是否是 CentOS8。

   CentOS8 中缺少 `libnsl.so` 系统库，可以通过手动安装的方式解决：

   ```
   dnf install libnsl
   ```

2. 检查系统的 `ulimit` 参数设置。

   ```
   ulimit -n 1000000
   ```

3. 使用 PD Control 工具检查在该节点（相同 IP 和 Port）是否有之前未成功下线的 TiFlash 实例，并将它们强制下线。（下线步骤参考手动缩容 TiFlash 节点）

如果遇到上述方法无法解决的问题，可以打包 TiFlash 的 log 文件夹，并在 AskTUG 社区中提问。

8.13.2 TiFlash 副本始终处于不可用状态

该问题一般由于配置错误或者环境问题导致 TiFlash 处于异常状态，可以先通过以下步骤定位问题组件：

1. 使用 pd-ctl 检查 PD 的Placement Rules 功能是否开启：

```
echo 'config show replication' | /path/to/pd-ctl -u http://<pd-ip>:<pd-port>
```

预期结果为 `"enable-placement-rules": "true"`（已开启）。如未开启，具体开启方法参考开启 Placement Rules 特性。

2. 通过 TiFlash-Summary 监控面板下的 UpTime 检查操作系统中 TiFlash 进程是否正常。
3. 通过 pd-ctl 查看 TiFlash proxy 状态是否正常：

```
echo "store" | /path/to/pd-ctl -u http://<pd-ip>:<pd-port>
```

store.labels 中含有 `{"key": "engine", "value": "tiflash"}` 信息的为 TiFlash proxy。

4. 查看 pd buddy 是否正常打印日志（日志路径的对应配置项 [flash.flash_cluster] log 设置的值，默认为 TiFlash 配置文件配置的 tmp 目录下）。
5. 检查配置的副本数是否小于等于集群 TiKV 节点数。若配置的副本数超过 TiKV 节点数，则 PD 不会向 TiFlash 同步数据；

```
echo 'config placement-rules show' | /path/to/pd-ctl -u http://<pd-ip>:<pd-port>
```

再确认 “default: count” 参数值。

> **注意：**
>
> 开启 Placement Rules 后，原先的 `max-replicas` 及 `location-labels` 配置项将不再生效。如果需要调整副本策略，应当使用 Placement Rules 相关接口。

6. 检查 TiFlash 节点对应 store 所在机器剩余的磁盘空间是否充足。默认情况下当磁盘剩余空间小于该 store 的 capacity 的 20%（通过 low-space-ratio 参数控制）时，PD 不会向 TiFlash 调度数据。

8.13.3 TiFlash 查询时间不稳定，同时错误日志中打印出大量的 Lock Exception

该问题是由于集群中存在大量写入，导致 TiFlash 查询时遇到锁并发生查询重试。

可以在 TiDB 中将查询时间戳设置为 1 秒前（例如：假设当前时间为 ‘2020-04-08 20:15:01’，可以在执行 query 前执行 `set @@tidb_snapshot='2020-04-08 20:15:00';`），来减小 TiFlash 查询碰到锁的可能性，从而减轻查询时间不稳定的程度。

8.13.4 部分查询返回 Region Unavailable 的错误

如果在 TiFlash 上的负载压力过大，会导致 TiFlash 数据同步落后，部分查询可能会返回 `Region Unavailable` 的错误。

在这种情况下，可以通过增加 TiFlash 节点数分担负载压力。

8.13.5 数据文件损坏

可依照如下步骤进行处理：

1. 参照下线 TiFlash 节点一节下线对应的 TiFlash 节点。
2. 清除该 TiFlash 节点的相关数据。
3. 重新在集群中部署 TiFlash 节点。

9 性能调优

9.1 系统调优

9.1.1 操作系统性能参数调优

本文档仅用于描述如何优化 CentOS 7 的各个子系统。

> 注意：
>
> - CentOS 7 操作系统的默认配置适用于中等负载下运行的大多数服务。调整特定子系统的性能可能会对其他子系统产生负面影响。因此在调整系统之前，请备份所有用户数据和配置信息；
> - 请在测试环境下对所有修改做好充分测试后，再应用到生产环境中。

9.1.1.1 性能分析工具

系统调优需要根据系统性能分析的结果做指导，因此本文先列出常用的性能分析方法。

9.1.1.1.1 60 秒分析法

此分析法由《性能之巅》的作者 Brendan Gregg 及其所在的 Netflix 性能工程团队公布。所用到的工具均可从发行版的官方源获取，通过分析以下清单中的输出，可定位大部分常见的性能问题。

- uptime
- dmesg | tail
- vmstat 1

- mpstat -P ALL 1
- pidstat 1
- iostat -xz 1
- free -m
- sar -n DEV 1
- sar -n TCP,ETCP 1
- top

具体用法可查询相应 man 手册。

9.1.1.1.2 perf

Perf 是 Linux 内核提供的一个重要的性能分析工具，它涵盖硬件级别（CPU/PMU 和性能监视单元）功能和软件功能（软件计数器和跟踪点）。详细用法请参考 perf Examples。

9.1.1.1.3 BCC/bpftrace

CentOS 从 7.6 版本起，内核已实现对 bpf 的支持，因此可根据上述清单的结果，选取适当的工具进行深入分析。相比 perf/ftrace，bpf 提供了可编程能力和更小的性能开销。相比 kprobe，bpf 提供了更高的安全性，更适合在生产环境上使用。关于 BCC 工具集的使用请参考 BPF Compiler Collection (BCC)。

9.1.1.2 性能调优

性能调优将根据内核子系统进行分类描述。

9.1.1.2.1 处理器 ——动态节能技术

cpufreq 是一个动态调整 CPU 频率的模块，可支持五种模式。为保证服务性能应选用 performance 模式，将 CPU 频率固定工作在其支持的最高运行频率上，不进行动态调节，操作命令为 `cpupower frequency-set --governor performance`。

9.1.1.2.2 处理器 ——中断亲和性

- 自动平衡：可通过 `irqbalance` 服务实现。
- 手动平衡：
  - 确定需要平衡中断的设备，从 CentOS 7.5 开始，系统会自动为某些设备及其驱动程序配置最佳的中断关联性。不能再手动配置其亲和性。目前已知的有使用 `be2iscsi` 驱动的设备，以及 NVMe 设置；
  - 对于其他设备，可查询其芯片手册，是否支持分发中断，若不支持，则该设备的所有中断会路由到同一个 CPU 上，无法对其进行修改。若支持，则计算 `smp_affinity` 掩码并设置对应的配置文件，具体请参考内核文档。

9.1.1.2.3 NUMA 绑核

为尽可能的避免跨 NUMA 访问内存，可以通过设置线程的 CPU 亲和性来实现 NUMA 绑核。对于普通程序，可使用 `numactl` 命令来绑定，具体用法请查询 man 手册。对于网卡中断，请参考下文网络章节。

9.1.1.2.4 内存 ——透明大页

对于数据库应用，不推荐使用 THP，因为数据库往往具有稀疏而不是连续的内存访问模式，且当高阶内存碎片化比较严重时，分配 THP 页面会出现较大的延迟。若开启针对 THP 的直接内存规整功能，也会出现系统 CPU 使用率激增的现象，因此建议关闭 THP。

```
echo never > /sys/kernel/mm/transparent_hugepage/enabled
echo never > /sys/kernel/mm/transparent_hugepage/defrag
```

9.1.1.2.5 内存 ——虚拟内存参数

- `dirty_ratio` 百分比值。当脏的 page cache 总量达到系统内存总量的这一百分比后，系统将开始使用 pdflush 操作将脏的 page cache 写入磁盘。默认值为 20%，通常不需调整。对于高性能 SSD，比如 NVMe 设备来说，降低其值有利于提高内存回收时的效率。
- `dirty_background_ratio` 百分比值。当脏的 page cache 总量达到系统内存总量的这一百分比后，系统开始在后台将脏的 page cache 写入磁盘。默认值为 10%，通常不需调整。对于高性能 SSD，比如 NVMe 设备来说，设置较低的值有利于提高内存回收时的效率。

9.1.1.2.6 存储及文件系统

内核 I/O 栈链路较长，包含了文件系统层、块设备层和驱动层。

I/O 调度器

I/O 调度程序确定 I/O 操作何时在存储设备上运行以及持续多长时间。也称为 I/O 升降机。对于 SSD 设备，宜设置为 noop。

```
echo noop > /sys/block/${SSD_DEV_NAME}/queue/scheduler
```

格式化参数 ——块大小

块是文件系统的工作单元。块大小决定了单个块中可以存储多少数据，因此决定了一次写入或读取的最小数据量。

默认块大小适用于大多数使用情况。但是，如果块大小（或多个块的大小）与通常一次读取或写入的数据量相同或稍大，则文件系统将性能更好，数据存储效率更高。小文件仍将使用整个块。文件可以分布在多个块中，但这会增加运行时开销。

使用 mkfs 命令格式化设备时，将块大小指定为文件系统选项的一部分。指定块大小的参数随文件系统的不同而不同。有关详细信息，请查询对应文件系统的 `mkfs` 手册页，比如 `man mkfs.ext4`。

挂载参数

`noatime` 读取文件时，将禁用对元数据的更新。它还启用了 nodiratime 行为，该行为会在读取目录时禁用对元数据的更新。

9.1.1.2.7 网络

网络子系统由具有敏感连接的许多不同部分组成。因此，CentOS 7 网络子系统旨在为大多数工作负载提供最佳性能，并自动优化其性能。因此，通常无需手动调整网络性能。

网络问题通常由硬件或相关设施出现问题导致的，因此在调优协议栈前，请先排除硬件问题。

尽管网络堆栈在很大程度上是自我优化的。但是在网络数据包处理过程中，以下方面可能会成为瓶颈并降低性能：

- 网卡硬件缓存：正确观察硬件层面的丢包方法是使用 `ethtool -S ${NIC_DEV_NAME}` 命令观察 drops 字段。当出现丢包现象时，主要考虑是硬/软中断的处理速度跟不上网卡接收速度。若接收缓存小于最大限制时，也可尝试增加 RX 缓存来防止丢包。查询命令为：`ethtool -g ${NIC_DEV_NAME}`，修改命令为 `ethtool -G ${NIC_DEV_NAME}`。
- 硬中断：若网卡支持 Receive-Side Scaling（RSS 也称为多网卡接收）功能，则观察 `/proc/interrputs` 网卡中断，如果出现了中断不均衡的情况，请参考处理器调优章节。若不支持 RSS 或 RSS 数量远小于物理 CPU 核数，则可配置 Receive Packet Steering（RPS，可以看作 RSS 的软件实现），及 RPS 的扩展 Receive Flow Steering (RFS)。具体设置请参考内核文档。
- 软中断：观察 `/proc/net/softnet\_stat` 监控。如果除第三列的其他列的数值在增长，则应适度调大 `net.core.netdev\_budget` 或 `net.core.dev\_weight` 值，使 softirq 可以获得更多的 CPU 时间。除此之外，也需要检查 CPU 使用情况，确定哪些任务在频繁占用 CPU，能否优化。
- 应用的套接字接收队列：监控 `ss -nmp` 的 `Recv-q` 列，若队列已满，则应考虑增大应用程序套接字的缓存大小或使用自动调整缓存的方式。除此之外，也要考虑能否优化应用层的架构，降低读取套接字的间隔。
- 以太网流控：若网卡和交换机支持流控功能，可通过使能此功能，给内核一些时间来处理网卡队列中的数据，来规避网卡缓存溢出的问题。对于网卡测，可通过 `ethtool -a ${NIC_DEV_NAME}` 命令检查是否支持/使能，并通过 `ethtool -A ${NIC_DEV_NAME}` 命令开启。对于交换机，请查询其手册。
- 中断合并：过于频繁的硬件中断会降低系统性能，而过晚的硬件中断会导致丢包。对于较新的网卡支持中断合并功能，并允许驱动自动调节硬件中断数。可通过 `ethtool -c ${NIC_DEV_NAME}` 命令检查，`ethtool -C ${NIC_DEV_NAME}` 命令开启。自适应模式使网卡可以自动调节中断合并。在自适应模式下，驱动程序将检查流量模式和内核接收模式，并实时评估合并设置，以防止数据包丢失。不同品牌的网卡具有不同的功能和默认配置，具体请参考网卡手册。
- 适配器队列：在协议栈处理之前，内核利用此队列缓存网卡接收的数据，每个 CPU 都有各自的 backlog 队列。此队列可缓存的最大 packets 数量为 `netdev\_max\_backlog`。观察 `/proc/net/softnet\_stat` 第二列，当某行的第二列持续增加，则意味着 CPU [行 -1] 队列已满，数据包被丢失，可通过持续加倍 `net.core.netdev\_max\_backlog` 值来解决。
- 发送队列：发送队列长度值确定在发送之前可以排队的数据包数量。默认值是 1000，对于 10 Gbps 足够。但若从 `ip -s link` 的输出中观察到 `TX errors` 值时，可尝试加倍该数据包数量：`ip link set dev ${NIC_DEV_NAME} txqueuelen 2000`。
- 驱动：网卡驱动通常也会提供调优参数，请查询设备硬件手册及其驱动文档。

9.2 软件调优

9.2.1 配置

9.2.1.1 TiDB 内存控制文档

目前 TiDB 已经能够做到追踪单条 SQL 查询过程中的内存使用情况，当内存使用超过一定阈值后也能采取一些操作来预防 OOM 或者排查 OOM 原因。在 TiDB 的配置文件中，我们可以使用如下配置来控制内存使用超阈值时 TiDB 的行为：

```
#### Valid options: ["log", "cancel"]
oom-action = "log"
```

- 如果上面的配置项使用的是 “log”，那么当一条 SQL 的内存使用超过一定阈值（由 session 变量 tidb_mem_quota_query 来控制）后，TiDB 会在 log 文件中打印一条 LOG，然后这条 SQL 继续执行，之后如果发生了 OOM 可以在 LOG 中找到对应的 SQL。
- 如果上面的配置项使用的是 “cancel”，那么当一条 SQL 的内存使用超过一定阈值后，TiDB 会立即中断这条 SQL 的执行并给客户端返回一个 error，error 信息中会详细写明这条 SQL 执行过程中各个占用内存比较多的物理执行算子的内存使用情况。

9.2.1.1.1 如何配置一条 SQL 执行过程中的内存使用阈值

可以在配置文件中设置每个 Query 默认的 Memory Quota，例如将其设置为 32GB：

```
mem-quota-query = 34359738368
```

此外还可通过如下几个 session 变量来控制一条 Query 中的内存使用，大多数用户只需要设置 tidb_mem_quota_query 即可，其他变量是高级配置，大多数用户不需要关心：

变量名	作用	单位	默认值
tidb_mem_quota_query	配置整条 SQL 的内存使用阈值	Byte	1 << 30 (1 GB)
tidb_mem_quota_hashjoin	配置 Hash Join 的内存使用阈值	Byte	32 << 30
tidb_mem_quota_mergejoin	配置 Merge Join 的内存使用阈值	Byte	32 << 30
tidb_mem_quota_sort	配置 Sort 的内存使用阈值	Byte	32 << 30
tidb_mem_quota_topn	配置 TopN 的内存使用阈值	Byte	32 << 30
tidb_mem_quota_indexlookupreader	配置 Index Lookup Reader 的内存使用阈值	Byte	32 << 30
tidb_mem_quota_indexlookupjoin	配置 Index Lookup Join 的内存使用阈值	Byte	32 << 30
tidb_mem_quota_nestedloopapply	配置 Nested Loop Apply 的内存使用阈值	Byte	32 << 30

几个使用例子：

配置整条 SQL 的内存使用阈值为 8GB：

```
set @@tidb_mem_quota_query = 8 << 30;
```

配置整条 SQL 的内存使用阈值为 8MB：

```
set @@tidb_mem_quota_query = 8 << 20;
```

配置整条 SQL 的内存使用阈值为 8KB：

```
set @@tidb_mem_quota_query = 8 << 10;
```

9.2.1.1.2 如何配置 tidb-server 实例使用内存的阈值

可以在配置文件中设置 tidb-server 实例的内存使用阈值。相关配置项为`server-memory-quota`。

例如，配置 tidb-server 实例的内存使用总量，将其设置成为 32 GB：

```
[performance]
server-memory-quota = 34359738368
```

在该配置下，当 tidb-server 实例内存使用到达 32 GB 时，正在执行的 SQL 语句会被随机强制终止，直至 tidb-server 实例内存使用下降到 32 GB 以下。被强制终止的 SQL 操作会向客户端返回 `Out Of Global Memory Limit!` 错误信息。

> 警告：
>
> - `server-memory-quota` 目前为实验性特性，不建议在生产环境中使用。
> - `server-memory-quota` 默认值为 0，表示无内存限制。

9.2.1.1.3 tidb-server 内存占用过高时的报警

默认配置下，tidb-server 实例会在机器内存使用达到总内存量的 80% 时打印报警日志，并记录相关状态文件。该内存使用率可以通过配置项`memory-usage-alarm-ratio` 进行设置。具体报警规则请参考该配置项的说明部分。

注意，当触发一次报警后，只有在内存使用率连续低于阈值超过 10 秒并再次达到阈值时，才会再次触发报警。此外，为避免报警时产生的状态文件积累过多，目前只会保留最近 5 次报警时所生成的状态文件。

下例通过构造一个占用大量内存的 SQL 语句触发报警，对该报警功能进行演示：

1. 配置报警比例为 `0.8`：

   ```
   mem-quota-query = 34359738368  // 将单条 SQL 内存限制调高，以便于构造占用内存较大的 SQL
   [performance]
   memory-usage-alarm-ratio = 0.8
   ```

2. 创建单表 `CREATE TABLE t(a int);` 并插入 1000 行数据。

3. 执行 `select * from t t1 join t t1 join t t3 order by t1.a`。该 SQL 语句会输出 1000000000 条记录，占用巨大的内存，进而触发报警。

4. 检查 `tidb.log` 文件，其中会记录系统总内存、系统当前内存使用量、tidb-server 实例的内存使用量以及状态文件所在目录。

   ```
   [2020/11/30 15:25:17.252 +08:00] [WARN] [memory_usage_alarm.go:141] ["tidb-server has the
       ↪ risk of OOM. Running SQLs and heap profile will be recorded in record path"] ["is
       ↪ server-memory-quota set"=false] ["system memory total"=33682427904] ["system memory
       ↪ usage"=27142864896] ["tidb-server memory usage"=22417922896] [memory-usage-alarm-
       ↪ ratio=0.8] ["record path"="/tmp/1000_tidb/MC4wLjAuMDo0MDAwLzAuMC4wLjA6MTAwODA=/tmp-
       ↪ storage/record"]
   ```

以上 Log 字段的含义如下：

- `is server-memory-quota set`：表示配置项`server-memory-quota` 是否被设置
- `system memory total`：表示当前系统的总内存
- `system memory usage`：表示当前系统的内存使用量
- `tidb-server memory usage`：表示 tidb-server 实例的内存使用量
- `memory-usage-alarm-ratio`：表示配置项`memory-usage-alarm-ratio` 的值
- `record path`：表示状态文件存放的目录

5. 通过访问状态文件所在目录（该示例中的目录为 `/tmp/1000_tidb/MC4wLjAuMDo0MDAwLzAuMC4wLjA6MTAwODA=/tmp-storage/record`），可以得到一组文件，其中包括 goroutinue、heap、running_sql 3 个文件，文件以记录状态文件的时间为后缀。这 3 个文件分别用来记录报警时的 goroutine 栈信息，堆内存使用状态，及正在运行的 SQL 信息。其中 `running_sql` 文件内的日志格式请参考`expensive-queries`。

9.2.1.2 TiKV 线程池性能调优

本文主要介绍 TiKV 线程池性能调优的主要手段，以及 TiKV 内部线程池的主要用途。

9.2.1.2.1 线程池介绍

在 TiKV 4.0 中，线程池主要由 gRPC、Scheduler、UnifyReadPool、Raftstore、Apply、RocksDB 以及其它一些占用 CPU 不多的定时任务与检测组件组成，这里主要介绍几个占用 CPU 比较多且会对用户读写请求的性能产生影响的线程池。

- gRPC 线程池：负责处理所有网络请求，它会把不同任务类型的请求转发给不同的线程池。
- Scheduler 线程池：负责检测写事务冲突，把事务的两阶段提交、悲观锁上锁、事务回滚等请求转化为 key-value 对数组，然后交给 Raftstore 线程进行 Raft 日志复制。
- Raftstore 线程池：负责处理所有的 Raft 消息以及添加新日志的提议 (Propose)、将日志写入到磁盘，当日志在多数副本中达成一致后，它就会把该日志发送给 Apply 线程。
- Apply 线程池：当收到从 Raftstore 线程池发来的已提交日志后，负责将其解析为 key-value 请求，然后写入 RocksDB 并且调用回调函数通知 gRPC 线程池中的写请求完成，返回结果给客户端。
- RocksDB 线程池：RocksDB 进行 Compact 和 Flush 任务的线程池，关于 RocksDB 的架构与 Compact 操作请参考 RocksDB: A Persistent Key-Value Store for Flash and RAM Storage。
- UnifyReadPool 线程池：TiKV 4.0 推出的新特性，由之前的 Coprocessor 线程池与 Storage Read Pool 合并而来，所有的读取请求包括 kv get、kv batch get、raw kv get、coprocessor 等都会在这个线程池中执行。

9.2.1.2.2 TiKV 的只读请求

TiKV 的读取请求分为两类：

- 一类是指定查询某一行或者某几行的简单查询，这类查询会运行在 Storage Read Pool 中。
- 另一类是复杂的聚合计算、范围查询，这类请求会运行在 Coprocessor Read Pool 中。

从 4.0 版本开始，支持两类读取请求使用同一个线程池，以减少线程数量，降低用户使用成本，默认不开启（默认点查询和 Coprocessor 请求使用不同的线程池）。用户可以通过将 `readpool.storage.use-unified-pool` 设置为 true 来打开统一线程池。

9.2.1.2.3 TiKV 线程池调优

- gRPC 线程池的大小默认配置（`server.grpc-concurrency`）是 4。由于 gRPC 线程池几乎不会有多少计算开销，它主要负责网络 IO、反序列化请求，因此该配置通常不需要调整。
  - 如果部署的机器 CPU 核数特别少（小于等于 8），可以考虑将该配置（`server.grpc-concurrency`）设置为 2。
  - 如果机器配置很高，并且 TiKV 承担了非常大量的读写请求，观察到 Grafana 上的监控 Thread CPU 的 gRPC poll CPU 的数值超过了 server.grpc-concurrency 大小的 80%，那么可以考虑适当调大 `server.grpc-concurrency` 以控制该线程池使用率在 80% 以下（即 Grafana 上的指标低于 `80% * server.grpc-concurrency` 的值）。
- Scheduler 线程池的大小配置 (`storage.scheduler-worker-pool-size`) 在 TiKV 检测到机器 CPU 核数大于等于 16 时默认为 8，小于 16 时默认为 4。它主要用于将复杂的事务请求转化为简单的 key-value 读写。但是 scheduler 线程池本身不进行任何写操作。
  - 如果检测到有事务冲突，那么它会提前返回冲突结果给客户端。
  - 如果未检测到事务冲突，那么它会把需要写入的 key-value 合并成一条 Raft 日志交给 Raftstore 线程进行 Raft 日志复制。

  通常来说为了避免过多的线程切换，最好确保 scheduler 线程池的利用率保持在 50% ~ 75% 之间。（如果线程池大小为 8 的话，那么 Grafana 上的 TiKV-Details.Thread CPU.scheduler worker CPU 应当在 400% ~ 600% 之间较为合理）
- Raftstore 线程池是 TiKV 最为复杂的一个线程池，默认大小（`raftstore.store-pool-size`）为 2，所有的写请求都会先在 Raftstore 线程 fsync 的方式写入 RocksDB（除非手动将 `raftstore.sync-log` 设置为 false；而 `raftstore.sync-log` 设置为 false，可以提升一部分写性能，但也会增加在机器故障时数据丢失的风险）。

  由于存在 I/O，Raftstore 线程理论上不可能达到 100% 的 CPU。为了尽可能地减少写磁盘次数，将多个写请求攒在一起写入 RocksDB，最好控制其整体 CPU 使用在 60% 以下（按照线程数默认值 2，则 Grafana 监控上的 TiKV-Details.Thread CPU.Raft store CPU 上的数值控制在 120% 以内较为合理）。不要为了提升写性能盲目增大 Raftstore 线程池大小，这样可能会适得其反，增加了磁盘负担让性能变差。
- UnifyReadPool 负责处理所有的读取请求。默认配置（`readpool.unified.max-thread-count`）大小为机器 CPU 数的 80%（如机器为 16 核，则默认线程池大小为 12）。

  通常建议根据业务负载特性调整其 CPU 使用率在线程池大小的 60% ~ 90% 之间 (如果用户 Grafana 上 TiKV-Details.Thread CPU.Unified read pool CPU 的峰值不超过 800%，那么建议用户将 `readpool.unified.max-thread-count` 设置为 10，过多的线程数会造成更频繁的线程切换，并且抢占其他线程池的资源)。
- RocksDB 线程池是 RocksDB 进行 Compact 和 Flush 任务的线程池，通常不需要配置。
  - 如果机器 CPU 核数较少，可将 `rocksdb.max-background-jobs` 与 `raftdb.max-background-jobs` 同时设置为 4。
  - 如果遇到了 Write Stall，可查看 Grafana 监控上 RocksDB-kv 中的 Write Stall Reason 有哪些指标不为 0。
    * 如果是由 pending compaction bytes 相关原因引起的，可将 `rocksdb.max-sub-compactions` 设置为 2 或者 3（该配置表示单次 compaction job 允许使用的子线程数量，TiKV 4.0 版本默认值为 3，3.0 版本默认值为 1）。

* 如果原因是 memtable count 相关，建议调大所有列的 max-write-buffer-number（默认为 5）。
* 如果原因是 level0 file limit 相关，建议调大如下参数为 64 或者更高：

```
rocksdb.defaultcf.level0-slowdown-writes-trigger
rocksdb.writecf.level0-slowdown-writes-trigger
rocksdb.lockcf.level0-slowdown-writes-trigger
rocksdb.defaultcf.level0-stop-writes-trigger
rocksdb.writecf.level0-stop-writes-trigger
rocksdb.lockcf.level0-stop-writes-trigger
```

9.2.1.3 TiKV 内存参数性能调优

本文档用于描述如何根据机器配置情况来调整 TiKV 的参数，使 TiKV 的性能达到最优。

TiKV 最底层使用的是 RocksDB 做为持久化存储，所以 TiKV 的很多性能相关的参数都是与 RocksDB 相关的。TiKV 使用了两个 RocksDB 实例，默认 RocksDB 实例存储 KV 数据，Raft RocksDB 实例（简称 RaftDB）存储 Raft 数据。

TiKV 使用了 RocksDB 的 Column Families (CF) 特性。

- 默认 RocksDB 实例将 KV 数据存储在内部的 default、write 和 lock 3 个 CF 内。
  - default CF 存储的是真正的数据，与其对应的参数位于 [rocksdb.defaultcf] 项中；
  - write CF 存储的是数据的版本信息 (MVCC) 以及索引相关的数据，相关的参数位于 [rocksdb.writecf] 项中；
  - lock CF 存储的是锁信息，系统使用默认参数。
- Raft RocksDB 实例存储 Raft log。
  - default CF 主要存储的是 Raft log，与其对应的参数位于 [raftdb.defaultcf] 项中。

所有的 CF 默认共同使用一个 block cache 实例。通过在 [storage.block-cache] 下设置 capacity 参数，你可以配置该 block cache 的大小。block cache 越大，能够缓存的热点数据越多，读取数据越容易，同时占用的系统内存也越多。如果要为每个 CF 使用单独的 block cache 实例，需要在 [storage.block-cache] 下设置 shared=false，并为每个 CF 配置单独的 block cache 大小。例如，可以在 [rocksdb.writecf] 下设置 block-cache-size 参数来配置 write CF 的大小。

> **注意：**
>
> 在 TiKV 3.0 之前的版本中，不支持使用 shared block cache，需要为每个 CF 单独配置 block cache。

每个 CF 有各自的 write-buffer，大小通过 write-buffer-size 控制。

9.2.1.3.1 参数说明

```
#### 日志级别，可选值为：trace，debug，warn，error，info，off
log-level = "info"

[server]
#### 监听地址
#### addr = "127.0.0.1:20160"

#### gRPC 线程池大小
#### grpc-concurrency = 4
#### TiKV 每个实例之间的 gRPC 连接数
#### grpc-raft-conn-num = 10

#### TiDB 过来的大部分读请求都会发送到 TiKV 的 Coprocessor 进行处理，该参数用于设置
#### coprocessor 线程的个数，如果业务是读请求比较多，增加 coprocessor 的线程数，但应比系统的
#### CPU 核数小。例如：TiKV 所在的机器有 32 core，在重读的场景下甚至可以将该参数设置为 30。在没有
#### 设置该参数的情况下，TiKV 会自动将该值设置为 CPU 总核数乘以 0.8。
#### end-point-concurrency = 8

#### 可以给 TiKV 实例打标签，用于副本的调度
#### labels = {zone = "cn-east-1", host = "118", disk = "ssd"}

[storage]
#### 数据目录
#### data-dir = "/tmp/tikv/store"

#### 通常情况下使用默认值就可以了。在导数据的情况下建议将该参数设置为 1024000。
#### scheduler-concurrency = 102400
#### 该参数控制写入线程的个数，当写入操作比较频繁的时候，需要把该参数调大。使用 top -H -p tikv-
    ↪ pid
#### 发现名称为 sched-worker-pool 的线程都特别忙，这个时候就需要将 scheduler-worker-pool-size
#### 参数调大，增加写线程的个数。
#### scheduler-worker-pool-size = 4

[storage.block-cache]
##### 是否为 RocksDB 的所有 CF 都创建一个 `shared block cache`。
##### ## RocksDB 使用 block cache 来缓存未压缩的数据块。较大的 block cache 可以加快读取速度。
##### 推荐开启 `shared block cache` 参数。这样只需要设置全部缓存大小，使配置过程更加方便。
##### 在大多数情况下，可以通过 LRU 算法在各 CF 间自动平衡缓存用量。
##### ## `storage.block-cache` 会话中的其余配置仅在开启 `shared block cache` 时起作用。
#### shared = true
##### `shared block cache` 的大小。正常情况下应设置为系统全部内存的 30%-50%。
##### 如果未设置该参数，则由以下字段或其默认值的总和决定。
##### ##   * rocksdb.defaultcf.block-cache-size 或系统全部内存的 25%
#####   * rocksdb.writecf.block-cache-size 或系统全部内存的 15%
```

```
#####   * rocksdb.lockcf.block-cache-size 或系统全部内存的 2%
#####   * raftdb.defaultcf.block-cache-size 或系统全部内存的 2%
##### ## 要在单个物理机上部署多个 TiKV 节点，需要显式配置该参数。
##### 否则，TiKV 中可能会出现 OOM 错误。
#### capacity = "1GB"

[pd]
#### pd 的地址
#### endpoints = ["127.0.0.1:2379","127.0.0.2:2379","127.0.0.3:2379"]

[metric]
#### 将 metrics 推送给 Prometheus pushgateway 的时间间隔
interval = "15s"
#### Prometheus pushgateway 的地址
address = ""
job = "tikv"

[raftstore]
#### 默认为 true，表示强制将数据刷到磁盘上。如果是非金融安全级别的业务场景，建议设置成 false，
#### 以便获得更高的性能。
sync-log = true

#### Raft RocksDB 目录。默认值是 [storage.data-dir] 的 raft 子目录。
#### 如果机器上有多块磁盘，可以将 Raft RocksDB 的数据放在不同的盘上，提高 TiKV 的性能。
#### raftdb-path = "/tmp/tikv/store/raft"

region-max-size = "384MB"
#### Region 分裂阈值
region-split-size = "256MB"
#### 当 Region 写入的数据量超过该阈值的时候，TiKV 会检查该 Region 是否需要分裂。为了减少检查过程
#### 中扫描数据的成本，数据过程中可以将该值设置为32MB，正常运行状态下使用默认值即可。
region-split-check-diff = "32MB"

[rocksdb]
#### RocksDB 进行后台任务的最大线程数，后台任务包括 compaction 和 flush。具体 RocksDB
    ↪ 为什么需要进行 compaction，
#### 请参考 RocksDB 的相关资料。在写流量比较大的时候（例如导数据），建议开启更多的线程，
#### 但应小于 CPU 的核数。例如在导数据的时候，32 核 CPU 的机器，可以设置成 28。
#### max-background-jobs = 8

#### RocksDB 能够打开的最大文件句柄数。
#### max-open-files = 40960

#### RocksDB MANIFEST 文件的大小限制.□# 更详细的信息请参考：https://github.com/facebook/rocksdb/
    ↪ wiki/MANIFEST
```

```
max-manifest-file-size = "20MB"

#### RocksDB write-ahead logs 目录。如果机器上有两块盘，可以将 RocksDB 的数据和 WAL 日志放在
#### 不同的盘上，提高 TiKV 的性能。
#### wal-dir = "/tmp/tikv/store"

#### 下面两个参数用于怎样处理 RocksDB 归档 WAL。
#### 更多详细信息请参考：https://github.com/facebook/rocksdb/wiki/How-to-persist-in-memory-
    ↪ RocksDB-database%3F
#### wal-ttl-seconds = 0
#### wal-size-limit = 0

#### RocksDB WAL 日志的最大总大小，通常情况下使用默认值就可以了。
#### max-total-wal-size = "4GB"

#### 可以通过该参数打开或者关闭 RocksDB 的统计信息。
#### enable-statistics = true

#### 开启 RocksDB compaction 过程中的预读功能，如果使用的是机械磁盘，建议该值至少为2MB。
#### compaction-readahead-size = "2MB"

[rocksdb.defaultcf]
#### 数据块大小。RocksDB 是按照 block 为单元对数据进行压缩的，同时 block 也是缓存在 block-cache
#### 中的最小单元（类似其他数据库的 page 概念）。
block-size = "64KB"

#### RocksDB 每一层数据的压缩方式，可选的值为：no,snappy,zlib,bzip2,lz4,lz4hc,zstd。
#### no:no:lz4:lz4:lz4:zstd:zstd 表示 level0 和 level1 不压缩，level2 到 level4 采用 lz4 压缩算法
    ↪ ，
#### level5 和 level6 采用 zstd 压缩算法,。
#### no 表示没有压缩，lz4 是速度和压缩比较为中庸的压缩算法，zlib 的压缩比很高，对存储空间比较友
#### 好，但是压缩速度比较慢，压缩的时候需要占用较多的 CPU 资源。不同的机器需要根据 CPU 以及 I/O
    ↪ 资
#### 源情况来配置怎样的压缩方式。例如：如果采用的压缩方式为"no:no:lz4:lz4:lz4:zstd:zstd"，在大量
#### 写入数据的情况下（导数据），发现系统的 I/O 压力很大（使用 iostat 发现 %util 持续 100% 或者使
#### 用 top 命令发现 iowait 特别多），而 CPU 的资源还比较充裕，这个时候可以考虑将 level0 和
#### level1 开启压缩，用 CPU 资源换取 I/O 资源。如果采用的压缩方式
#### 为"no:no:lz4:lz4:lz4:zstd:zstd"，在大量写入数据的情况下，发现系统的 I/O 压力不大，但是 CPU
#### 资源已经吃光了，top -H 发现有大量的 bg 开头的线程（RocksDB 的 compaction 线程）在运行，这
#### 个时候可以考虑用 I/O 资源换取 CPU 资源，将压缩方式改成"no:no:no:lz4:lz4:zstd:zstd"。总之，目
#### 的是为了最大限度地利用系统的现有资源，使 TiKV 的性能在现有的资源情况下充分发挥。
compression-per-level = ["no", "no", "lz4", "lz4", "lz4", "zstd", "zstd"]

#### RocksDB memtable 的大小。
write-buffer-size = "128MB"
```

```
#### 最多允许几个 memtable 存在。写入到 RocksDB 的数据首先会记录到 WAL 日志里面，然后会插入到
#### memtable 里面，当 memtable 的大小到达了 write-buffer-size 限定的大小的时候，当前的
#### memtable 会变成只读的，然后生成一个新的 memtable 接收新的写入。只读的 memtable 会被
#### RocksDB 的 flush 线程（max-background-flushes 参数能够控制 flush 线程的最大个数）
#### flush 到磁盘，成为 level0 的一个 sst 文件。当 flush 线程忙不过来，导致等待 flush 到磁盘的
#### memtable 的数量到达 max-write-buffer-number 限定的个数的时候，RocksDB 会将新的写入
#### stall 住，stall 是 RocksDB 的一种流控机制。在导数据的时候可以将 max-write-buffer-number
#### 的值设置的更大一点，例如 10。
max-write-buffer-number = 5

#### 当 level0 的 sst 文件个数到达 level0-slowdown-writes-trigger 指定的限度的时候，
#### RocksDB 会尝试减慢写入的速度。因为 level0 的 sst 太多会导致 RocksDB 的读放大上升。
#### level0-slowdown-writes-trigger 和 level0-stop-writes-trigger 是 RocksDB 进行流控的
#### 另一个表现。当 level0 的 sst 的文件个数到达 4（默认值），level0 的 sst 文件会和 level1 中
#### 有 overlap 的 sst 文件进行 compaction，缓解读放大的问题。
level0-slowdown-writes-trigger = 20

#### 当 level0 的 sst 文件个数到达 level0-stop-writes-trigger 指定的限度的时候，RocksDB 会
#### stall 住新的写入。
level0-stop-writes-trigger = 36

#### 当 level1 的数据量大小达到 max-bytes-for-level-base 限定的值的时候，会触发 level1 的
#### sst 和 level2 种有 overlap 的 sst 进行 compaction。
#### 黄金定律：max-bytes-for-level-base 的设置的第一参考原则就是保证和 level0 的数据量大致相
#### 等，这样能够减少不必要的 compaction。例如压缩方式为"no:no:lz4:lz4:lz4:lz4:lz4"，那么
#### max-bytes-for-level-base 的值应该是 write-buffer-size 的大小乘以 4，因为 level0 和
#### level1 都没有压缩，而且 level0 触发 compaction 的条件是 sst 的个数到达 4（默认值）。在
#### level0 和 level1 都采取了压缩的情况下，就需要分析下 RocksDB 的日志，看一个 memtable 的压
#### 缩成一个 sst 文件的大小大概是多少，例如 32MB，那么 max-bytes-for-level-base 的建议值就应
#### 该是 32MB * 4 = 128MB。
max-bytes-for-level-base = "512MB"

#### sst 文件的大小。level0 的 sst 文件的大小受 write-buffer-size 和 level0 采用的压缩算法的
#### 影响，target-file-size-base 参数用于控制 level1-level6 单个 sst 文件的大小。
target-file-size-base = "32MB"

[rocksdb.writecf]
#### 保持和 rocksdb.defaultcf.compression-per-level 一致。
compression-per-level = ["no", "no", "lz4", "lz4", "lz4", "zstd", "zstd"]

#### 保持和 rocksdb.defaultcf.write-buffer-size 一致。
write-buffer-size = "128MB"
max-write-buffer-number = 5
min-write-buffer-number-to-merge = 1
```

```
#### 保持和 rocksdb.defaultcf.max-bytes-for-level-base 一致。
max-bytes-for-level-base = "512MB"
target-file-size-base = "32MB"

[raftdb]
#### RaftDB 能够打开的最大文件句柄数。
#### max-open-files = 40960

#### 可以通过该参数打开或者关闭 RaftDB 的统计信息。
#### enable-statistics = true

#### 开启 RaftDB compaction 过程中的预读功能，如果使用的是机械磁盘，建议该值至少为2MB。
#### compaction-readahead-size = "2MB"

[raftdb.defaultcf]
#### 保持和 rocksdb.defaultcf.compression-per-level 一致。
compression-per-level = ["no", "no", "lz4", "lz4", "lz4", "zstd", "zstd"]

#### 保持和 rocksdb.defaultcf.write-buffer-size 一致。
write-buffer-size = "128MB"
max-write-buffer-number = 5
min-write-buffer-number-to-merge = 1

#### 保持和 rocksdb.defaultcf.max-bytes-for-level-base 一致。
max-bytes-for-level-base = "512MB"
target-file-size-base = "32MB"
```

9.2.1.3.2 TiKV 内存使用情况

除了以上列出的 block-cache 以及 write-buffer 会占用系统内存外：

1. 需预留一些内存作为系统的 page cache
2. TiKV 在处理大的查询的时候（例如 select * from ...）会读取数据然后在内存中生成对应的数据结构返回给 TiDB，这个过程中 TiKV 会占用一部分内存

9.2.1.3.3 TiKV 机器配置推荐

1. 生产环境中，不建议将 TiKV 部署在 CPU 核数小于 8 或内存低于 32GB 的机器上
2. 如果对写入吞吐要求比较高，建议使用吞吐能力比较好的磁盘
3. 如果对读写的延迟要求非常高，建议使用 IOPS 比较高的 SSD 盘

9.2.1.4 Follower Read

当系统中存在读取热点 Region 导致 leader 资源紧张成为整个系统读取瓶颈时，启用 Follower Read 功能可明显降低 leader 的负担，并且通过在多个 follower 之间均衡负载，显著地提升整体系统的吞吐能力。本文主要介绍 Follower Read 的使用方法与实现机制。

9.2.1.4.1 概述

Follower Read 功能是指在强一致性读的前提下使用 Region 的 follower 副本来承载数据读取的任务，从而提升 TiDB 集群的吞吐能力并降低 leader 负载。Follower Read 包含一系列将 TiKV 读取负载从 Region 的 leader 副本上 offload 到 follower 副本的负载均衡机制。TiKV 的 Follower Read 可以保证数据读取的一致性，可以为用户提供强一致的数据读取能力。

> **注意：**
>
> 为了获得强一致读取的能力，在当前的实现中，follower 节点需要向 leader 节点询问当前的执行进度（即 ReadIndex），这会产生一次额外的网络请求开销，因此目前 Follower Read 的主要优势是处理隔离集群的读写请求以及提升整体读取吞吐。

9.2.1.4.2 使用方式

要开启 TiDB 的 Follower Read 功能，将 SESSION 变量 `tidb_replica_read` 的值设置为 `follower` 或 `leader-and-follower` 即可：

```
set @@tidb_replica_read = '<目标值>';
```

作用域：SESSION

默认值：leader

该变量用于设置当前会话期待的数据读取方式。

- 当设定为默认值 `leader` 或者空字符串时，TiDB 会维持原有行为方式，将所有的读取操作都发送给 leader 副本处理。
- 当设置为 `follower` 时，TiDB 会选择 Region 的 follower 副本完成所有的数据读取操作。
- 当 `tidb_replica_read` 的值设为 `leader-and-follower` 时，TiDB 可以选择任意副本来执行读取操作。
- 当设置为 `leader-and-follower` 时，读请求会在 leader 和 follower 之间负载均衡。

9.2.1.4.3 实现机制

在 Follower Read 功能出现之前，TiDB 采用 strong leader 策略将所有的读写操作全部提交到 Region 的 leader 节点上完成。虽然 TiKV 能够很均匀地将 Region 分散到多个物理节点上，但是对于每一个 Region 来说，只有 leader 副本能够对外提供服务，另外的 follower 除了时刻同步数据准备着 failover 时投票切换成为 leader 外，没有办法对 TiDB 的请求提供任何帮助。

为了允许在 TiKV 的 follower 节点进行数据读取，同时又不破坏线性一致性和 Snapshot Isolation 的事务隔离，Region 的 follower 节点需要使用 Raft ReadIndex 协议确保当前读请求可以读到当前 leader 上已经 commit 的最新数据。在 TiDB 层面，Follower Read 只需根据负载均衡策略将某个 Region 的读取请求发送到 follower 节点。

Follower 强一致读

TiKV follower 节点处理读取请求时，首先使用 Raft ReadIndex 协议与 Region 当前的 leader 进行一次交互，来获取当前 Raft group 最新的 commit index。本地 apply 到所获取的 leader 最新 commit index 后，便可以开始正常的读取请求处理流程。

Follower 副本选择策略

由于 TiKV 的 Follower Read 不会破坏 TiDB 的 Snapshot Isolation 事务隔离级别，因此 TiDB 选择 follower 的策略可以采用 round robin 的方式。目前，对于 Coprocessor 请求，Follower Read 负载均衡策略粒度是连接级别的，对于一个 TiDB 的客户端连接在某个具体的 Region 上会固定使用同一个 follower，只有在选中的 follower 发生故障或者因调度策略发生调整的情况下才会进行切换。而对于非 Coprocessor 请求（点查等），Follower Read 负载均衡策略粒度是事务级别的，对于一个 TiDB 的事务在某个具体的 Region 上会固定使用同一个 follower，同样在 follower 发生故障或者因调度策略发生调整的情况下才会进行切换。

9.2.1.5 TiFlash 性能调优

本文介绍了使 TiFlash 性能达到最优的几种方式，包括规划机器资源、TiDB 参数调优、配置 TiKV Region 大小等。

9.2.1.5.1 资源规划

对于希望节省机器资源，并且完全没有隔离要求的场景，可以使用 TiKV 和 TiFlash 联合部署。建议为 TiKV 与 TiFlash 分别留够资源，并且不要共享磁盘。

9.2.1.5.2 TiDB 相关参数调优

1. 对于 OLAP/TiFlash 专属的 TiDB 节点，建议调大读取并发数`tidb_distsql_scan_concurrency` 到 80：

```
set @@tidb_distsql_scan_concurrency = 80;
```

2. 开启 Super batch 功能：

   `tidb_allow_batch_cop` 变量用来设置从 TiFlash 读取时，是否把 Region 的请求进行合并。当查询中涉及的 Region 数量比较大，可以尝试设置该变量为 1（对带 `aggregation` 下推到 TiFlash Coprocessor 的请求生效），或设置该变量为 2（对全部下推到 TiFlash Coprocessor 请求生效）。

```
set @@tidb_allow_batch_cop = 1;
```

3. 尝试开启聚合推过 `Join` / `Union` 等 TiDB 算子的优化：

   `tidb_opt_agg_push_down` 变量用来设置优化器是否执行聚合函数下推到 Join 之前的优化操作。当查询中聚合操作执行很慢时，可以尝试设置该变量为 1。

```
set @@tidb_opt_agg_push_down = 1;
```

4. 尝试开启 `Distinct` 推过 `Join` / `Union` 等 TiDB 算子的优化：

   `tidb_opt_distinct_agg_push_down` 变量用来设置优化器是否执行带有 `Distinct` 的聚合函数（比如 `select count(distinct a)from t`）下推到 Coprocessor 的优化操作。当查询中带有 `Distinct` 的聚合操作执行很慢时，可以尝试设置该变量为 1。

```
set @@tidb_opt_distinct_agg_push_down = 1;
```

9.2.2 下推计算结果缓存

TiDB 从 4.0 起支持下推计算结果缓存（即 Coprocessor Cache 功能）。开启该功能后，将在 TiDB 实例侧缓存下推给 TiKV 计算的结果，在部分场景下起到加速效果。

9.2.2.1 配置

Coprocessor Cache 的配置均位于 TiDB 的 `tikv-client.copr-cache` 配置项中。Coprocessor 的具体开启和配置方法，见TiDB 配置文件描述。

9.2.2.2 特性说明

- 所有 SQL 在单个 TiDB 实例上的首次执行都不会被缓存。
- 缓存仅存储在 TiDB 内存中，TiDB 重启后缓存会失效。
- 不同 TiDB 实例之间不共享缓存。
- 缓存的是下推计算结果，即使缓存命中，后续仍有 TiDB 计算。
- 缓存以 Region 为单位。对 Region 的写入会导致涉及该 Region 的缓存失效。基于此原因，该功能主要会对很少变更的数据有效果。
- 下推计算请求相同时，缓存会被命中。通常在以下场景下，下推计算的请求是相同或部分相同的：
  - SQL 语句完全一致，例如重复执行相同的 SQL 语句。
    该场景下所有下推计算的请求都是一致的，所有请求都能利用上下推计算缓存。
  - SQL 语句包含一个变化的条件，其他部分一致，变化的条件是表主键或分区主键。
    该场景下一部分下推计算的请求会与之前出现过的一致，部分请求能利用上下推计算结果缓存。
  - SQL 语句包含多个变化的条件，其他部分一致，变化的条件完全匹配一个复合索引列。
    该场景下一部分下推计算的请求会与之前出现过的一致，部分请求能利用上下推计算结果缓存。
- 该功能对用户透明，开启和关闭都不影响计算结果，仅影响 SQL 执行时间。

9.2.2.3 检验缓存效果

可以通过执行 `EXPLAIN ANALYZE` 或查看 Grafana 监控面板来检查 Coprocessor 的缓存效果。

9.2.2.3.1 使用 `EXPLAIN ANALYZE`

用户可以通过`EXPLAIN ANALYZE` 语句查看读表算子中的缓存命中率，示例如下：

```
EXPLAIN ANALYZE SELECT * FROM t USE INDEX(a);
+--
    ↪ ----------------------------+-----------+---------+-----------+-----------------------+----------------
    ↪
| id                            | estRows   | actRows | task      | access object         |
    ↪ execution info
    ↪
    ↪ | operator info                 | memory               | disk |
```

```
+--
    ↪ ----------------------------+-----------+---------+-----------+-------------------------+---------------
    ↪
| IndexLookUp_6                  | 262400.00 | 262400  | root      |                         | time
    ↪ :620.513742ms, loops:258, cop_task: {num: 4, max: 5.530817ms, min: 1.51829ms, avg:
    ↪ 2.70883ms, p95: 5.530817ms, max_proc_keys: 2480, p95_proc_keys: 2480, tot_proc: 1ms,
    ↪ tot_wait: 1ms, rpc_num: 4, rpc_time: 10.816328ms, copr_cache_hit_rate: 0.75} |
    ↪                                  | 6.685169219970703 MB  | N/A   |
| ├─IndexFullScan_4(Build)       | 262400.00 | 262400  | cop[tikv] | table:t, index:a(a, c) |
    ↪ proc max:93ms, min:1ms, p80:93ms, p95:93ms, iters:275, tasks:4
    ↪
    ↪ | keep order:false, stats:pseudo | 1.7549400329589844 MB | N/A   |
| └─TableRowIDScan_5(Probe)      | 262400.00 | 0       | cop[tikv] | table:t                 |
    ↪ time:0ns, loops:0
    ↪
    ↪ | keep order:false, stats:pseudo | N/A                   | N/A   |
+--
    ↪ ----------------------------+-----------+---------+-----------+-------------------------+---------------
    ↪
3 rows in set (0.62 sec)
```

执行结果的 execution info 列有 copr_cache_hit_ratio 信息，表示下推计算结果缓存的命中率。以上示例的 0.75 表示命中率大约是 75%。

9.2.2.3.2 查看 Grafana 监控面板

在 Grafana 监控中，tidb 命名空间下 distsql 子系统中可见 copr-cache 面板，该面板监控整个集群中下推计算结果缓存的命中次数、未命中次数和缓存丢弃次数。

9.3 SQL 性能调优

9.3.1 SQL 性能调优

SQL 是一种声明性语言。一条 SQL 语句描述的是最终结果应该如何，而非按顺序执行的步骤。TiDB 会优化 SQL 语句的执行，语义上允许以任何顺序执行查询的各部分，前提是能正确返回语句所描述的最终结果。

SQL 性能优化的过程，可以理解为 GPS 导航的过程。你提供地址后，GPS 软件利用各种统计信息（例如以前的行程、速度限制等元数据，以及实时交通信息）规划出一条最省时的路线。这与 TiDB 中的 SQL 性能优化过程相对应。

本章节包括以下文档，可帮助你更好地理解查询执行计划：

- 理解 TiDB 执行计划介绍如何使用 EXPLAIN 语句来理解 TiDB 是如何执行某个查询的。
- SQL 优化流程概览介绍 TiDB 可以使用的几种优化，以提高查询性能。
- 控制执行计划介绍如何控制执行计划的生成。TiDB 的执行计划非最优时，建议控制执行计划。

9.3.2 理解 TiDB 执行计划

9.3.2.1 EXPLAIN 概览

> **注意：**
>
> 使用 MySQL 客户端连接到 TiDB 时，为避免输出结果在终端中换行，可先执行 `pager less -S` 命令。执行命令后，新的 EXPLAIN 的输出结果不再换行，可按右箭头 → 键水平滚动阅读输出结果。

使用 EXPLAIN 可查看 TiDB 执行某条语句时选用的执行计划。也就是说，TiDB 在考虑上数百或数千种可能的执行计划后，最终认定该执行计划消耗的资源最少、执行的速度最快。

EXPLAIN 示例如下：

```
CREATE TABLE t (id INT NOT NULL PRIMARY KEY auto_increment, a INT NOT NULL, pad1 VARCHAR(255),
    ↪ INDEX(a));
INSERT INTO t VALUES (1, 1, 'aaa'),(2,2, 'bbb');
EXPLAIN SELECT * FROM t WHERE a = 1;
```

返回的结果如下：

```
Query OK, 0 rows affected (0.96 sec)

Query OK, 2 rows affected (0.02 sec)
Records: 2  Duplicates: 0  Warnings: 0

+--
    ↪ ----------------------------+---------+-----------+--------------------+-----------------------------
    ↪
| id                            | estRows | task      | access object      | operator info
    ↪                               |
+--
    ↪ ----------------------------+---------+-----------+--------------------+-----------------------------
    ↪
| IndexLookUp_10                | 10.00   | root      |                    |
    ↪                                          |
| ├─IndexRangeScan_8(Build)     | 10.00   | cop[tikv] | table:t, index:a(a) | range:[1,1], keep
    ↪ order:false, stats:pseudo |
| └─TableRowIDScan_9(Probe)     | 10.00   | cop[tikv] | table:t             | keep order:false,
    ↪ stats:pseudo              |
+--
    ↪ ----------------------------+---------+-----------+--------------------+-----------------------------
    ↪
3 rows in set (0.00 sec)
```

EXPLAIN 实际不会执行查询。EXPLAIN ANALYZE 可用于实际执行查询并显示执行计划。如果 TiDB 所选的执行计划非最优，可用 EXPLAIN 或 EXPLAIN ANALYZE 来进行诊断。有关 EXPLAIN 用法的详细内容，参阅以下文档：

- 索引查询的执行计划
- Join 查询的执行计划
- 子查询的执行计划
- 聚合查询的执行计划
- 视图查询的执行计划
- 分区查询的执行计划
- 开启 IndexMerge 查询的执行计划

9.3.2.1.1 解读 EXPLAIN 的返回结果

EXPLAIN 的返回结果包含以下字段：

- id 为算子名，或执行 SQL 语句需要执行的子任务。详见算子简介。
- estRows 为显示 TiDB 预计会处理的行数。该预估数可能基于字典信息（例如访问方法基于主键或唯一键），或基于 CMSketch 或直方图等统计信息估算而来。
- task 显示算子在执行语句时的所在位置。详见Task 简介。
- access-object 显示被访问的表、分区和索引。显示的索引为部分索引。以上示例中 TiDB 使用了 a 列的索引。尤其是在有组合索引的情况下，该字段显示的信息很有参考意义。
- operator info 显示访问表、分区和索引的其他信息。详见operator info 结果。

算子简介

算子是为返回查询结果而执行的特定步骤。真正执行扫表（读盘或者读 TiKV Block Cache）操作的算子有如下几类：

- TableFullScan：全表扫描。
- TableRangeScan：带有范围的表数据扫描。
- TableRowIDScan：根据上层传递下来的 RowID 扫描表数据。时常在索引读操作后检索符合条件的行。
- IndexFullScan：另一种“全表扫描”，扫的是索引数据，不是表数据。
- IndexRangeScan：带有范围的索引数据扫描操作。

TiDB 会汇聚 TiKV/TiFlash 上扫描的数据或者计算结果，这种“数据汇聚”算子目前有如下几类：

- TableReader：将 TiKV 上底层扫表算子 TableFullScan 或 TableRangeScan 得到的数据进行汇总。
- IndexReader：将 TiKV 上底层扫表算子 IndexFullScan 或 IndexRangeScan 得到的数据进行汇总。
- IndexLookUp：先汇总 Build 端 TiKV 扫描上来的 RowID，再去 Probe 端上根据这些 RowID 精确地读取 TiKV 上的数据。Build 端是 IndexFullScan 或 IndexRangeScan 类型的算子，Probe 端是 TableRowIDScan 类型的算子。
- IndexMerge：和 IndexLookupReader 类似，可以看做是它的扩展，可以同时读取多个索引的数据，有多个 Build 端，一个 Probe 端。执行过程也很类似，先汇总所有 Build 端 TiKV 扫描上来的 RowID，再去 Probe 端上根据这些 RowID 精确地读取 TiKV 上的数据。Build 端是 IndexFullScan 或 IndexRangeScan 类型的算子，Probe 端是 TableRowIDScan 类型的算子。

算子的执行顺序

算子的结构是树状的，但在查询执行过程中，并不严格要求子节点任务在父节点之前完成。TiDB 支持同一查询内的并行处理，即子节点“流入”父节点。父节点、子节点和同级节点可能并行执行查询的一部分。

在以上示例中，`├─IndexRangeScan_8(Build)` 算子为 `a(a)` 索引所匹配的行查找内部 RowID。`└─TableRowIDScan_9 ↪ (Probe)` 算子随后从表中检索这些行。

Build 总是先于 Probe 执行，并且 Build 总是出现在 Probe 前面。即如果一个算子有多个子节点，子节点 ID 后面有 Build 关键字的算子总是先于有 Probe 关键字的算子执行。TiDB 在展现执行计划的时候，Build 端总是第一个出现，接着才是 Probe 端。

范围查询

在 WHERE/HAVING/ON 条件中，TiDB 优化器会分析主键或索引键的查询返回。如数字、日期类型的比较符，如大于、小于、等于以及大于等于、小于等于，字符类型的 LIKE 符号等。

若要使用索引，条件必须是“Sargable”(Search ARGument ABLE) 的。例如条件 `YEAR(date_column)< 1992` 不能使用索引，但 `date_column < '1992-01-01` 就可以使用索引。

推荐使用同一类型的数据以及同一类型的字符串和排序规则进行比较，以避免引入额外的 `cast` 操作而导致不能利用索引。

可以在范围查询条件中使用 AND（求交集）和 OR（求并集）进行组合。对于多维组合索引，可以对多个列使用条件。例如对组合索引 `(a, b, c)`：

- 当 a 为等值查询时，可以继续求 b 的查询范围。
- 当 b 也为等值查询时，可以继续求 c 的查询范围。
- 反之，如果 a 为非等值查询，则只能求 a 的范围。

Task 简介

目前 TiDB 的计算任务分为两种不同的 task：cop task 和 root task。Cop task 是指使用 TiKV 中的 Coprocessor 执行的计算任务，root task 是指在 TiDB 中执行的计算任务。

SQL 优化的目标之一是将计算尽可能地下推到 TiKV 中执行。TiKV 中的 Coprocessor 能支持大部分 SQL 内建函数（包括聚合函数和标量函数）、SQL LIMIT 操作、索引扫描和表扫描。但是，所有的 Join 操作都只能作为 root task 在 TiDB 上执行。

`operator info` 结果

EXPLAIN 返回结果中 `operator info` 列可显示诸如条件下推等信息。本文以上示例中，`operator info` 结果各字段解释如下：

- `range: [1,1]` 表示查询的 WHERE 字句 `(a = 1)` 被下推到了 TiKV，对应的 task 为 `cop[tikv]`。
- `keep order:false` 表示该查询的语义不需要 TiKV 按顺序返回结果。如果查询指定了排序（例如 `SELECT ↪ * FROM t WHERE a = 1 ORDER BY id`），该字段的返回结果为 `keep order:true`。
- `stats:pseudo` 表示 estRows 显示的预估数可能不准确。TiDB 定期在后台更新统计信息。也可以通过执行 `ANALYZE TABLE t` 来手动更新统计信息。

EXPLAIN 执行后，不同算子返回不同的信息。你可以使用 Optimizer Hints 来控制优化器的行为，以此控制物理算子的选择。例如 `/*+ HASH_JOIN(t1, t2)*/` 表示优化器将使用 Hash Join 算法。详细内容见Optimizer Hints。

9.3.2.1.2 算子相关的系统变量

TiDB 在 MySQL 的基础上，定义了一些专用的系统变量和语法用来优化性能。其中一些系统变量和具体的算子相关，比如算子的并发度，算子的内存使用上限，是否允许使用分区表等。这些都可以通过系统变量进行控制，从而影响各个算子执行的效率。

如果读者想要详细了解所有的系统变量及其使用规则，可以参见系统变量和语法。

9.3.2.1.3 另请参阅

- EXPLAIN
- EXPLAIN ANALYZE
- ANALYZE TABLE
- TRACE
- TiDB in Action
- 系统变量

9.3.2.2 使用 EXPLAIN 解读执行计划

SQL 是一种声明性语言，因此用户无法根据 SQL 语句直接判断一条查询的执行是否有效率。用户首先要使用EXPLAIN 语句查看当前的执行计划。

以 bikeshare 数据库示例（英文）中的一个 SQL 语句为例，该语句统计了 2017 年 7 月 1 日的行程次数：

```
EXPLAIN SELECT count(*) FROM trips WHERE start_date BETWEEN '2017-07-01 00:00:00' AND '2017-07-01
    ↪  23:59:59';
```

```
+--
    ↪ ----------------------------+----------+-----------+---------------+--------------------------------
    ↪
| id                           | estRows  | task      | access object | operator info
    ↪
    ↪ |
+--
    ↪ ----------------------------+----------+-----------+---------------+--------------------------------
    ↪
| StreamAgg_20                 | 1.00     | root      |               | funcs:count(Column#13)->
    ↪ Column#11
    ↪                                                                                                  |
| └─TableReader_21             | 1.00     | root      |               | data:StreamAgg_9
    ↪
    ↪ |
|   └─StreamAgg_9              | 1.00     | cop[tikv] |               | funcs:count(1)->Column
    ↪ #13
    ↪
    ↪ |
```

```
|     └─Selection_19           | 250.00   | cop[tikv] |               | ge(bikeshare.trips.
    ↪ start_date, 2017-07-01 00:00:00.000000), le(bikeshare.trips.start_date, 2017-07-01
    ↪ 23:59:59.000000) |
|       └─TableFullScan_18     | 10000.00 | cop[tikv] | table:trips   | keep order:false, stats:
    ↪ pseudo
    ↪                                                                                    |
+--
    ↪ ---------------------------+----------+-----------+---------------+--------------------------------
    ↪
5 rows in set (0.00 sec)
```

以上是该查询的执行计划结果。从 `└─TableFullScan_18` 算子开始向上看，查询的执行过程如下（非最佳执行计划）：

1. Coprocessor (TiKV) 读取整张 `trips` 表的数据，作为一次 TableFullScan 操作，再将读取到的数据传递给 Selection_19 算子。Selection_19 算子仍在 TiKV 内。
2. Selection_19 算子根据谓词 `WHERE start_date BETWEEN ..` 进行数据过滤。预计大约有 250 行数据满足该过滤条件（基于统计信息以及算子的执行逻辑估算而来）。`└─TableFullScan_18` 算子显示 `stats:pseudo`，表示该表没有实际统计信息，执行 `ANALYZE TABLE trips` 收集统计信息后，预计的估算的数字会更加准确。
3. COUNT 函数随后应用于满足过滤条件的行，这一过程也是在 TiKV (`cop[tikv]`) 中的 StreamAgg_9 算子内完成的。TiKV coprocessor 能执行一些 MySQL 内置函数，COUNT 是其中之一。
4. StreamAgg_9 算子执行的结果会被传递给 TableReader_21 算子（位于 TiDB 进程中，即 `root` 任务）。执行计划中，TableReader_21 算子的 `estRows` 为 1，表示该算子将从每个访问的 TiKV Region 接收一行数据。这一请求过程的详情，可参阅`EXPLAIN ANALYZE`。
5. StreamAgg_20 算子随后对 `└─TableReader_21` 算子传来的每行数据计算 COUNT 函数的结果。StreamAgg_20 是根算子，会将结果返回给客户端。

> 注意：
>
> 要查看 TiDB 中某张表的 Region 信息，可执行`SHOW TABLE REGIONS` 语句。

9.3.2.2.1 评估当前的性能

`EXPLAIN` 语句只返回查询的执行计划，并不执行该查询。若要获取实际的执行时间，可执行该查询，或使用 `EXPLAIN ANALYZE` 语句：

```
EXPLAIN ANALYZE SELECT count(*) FROM trips WHERE start_date BETWEEN '2017-07-01 00:00:00' AND '
    ↪ 2017-07-01 23:59:59';
```

```
+--
    ↪ --------------------------+----------+----------+----------+--------------+-----------------------
    ↪
| id                          | estRows  | actRows  | task      | access object | execution info
    ↪
    ↪
    ↪ | operator info
    ↪
    ↪ | memory    | disk |
+--
    ↪ --------------------------+----------+----------+----------+--------------+-----------------------
    ↪
| StreamAgg_20                | 1.00     | 1        | root      |               | time
    ↪ :1.031417203s, loops:2
    ↪
    ↪ | funcs:count(Column#13)->Column#11
    ↪                                                                                                |
    ↪ 632 Bytes | N/A  |
| └─TableReader_21            | 1.00     | 56       | root      |               | time
    ↪ :1.031408123s, loops:2, cop_task: {num: 56, max: 782.147269ms, min: 5.759953ms, avg:
    ↪ 252.005927ms, p95: 609.294603ms, max_proc_keys: 910371, p95_proc_keys: 704775, tot_proc:
    ↪ 11.524s, tot_wait: 580ms, rpc_num: 56, rpc_time: 14.111932641s} | data:StreamAgg_9
    ↪
    ↪ | 328 Bytes | N/A  |
|   └─StreamAgg_9             | 1.00     | 56       | cop[tikv] |               | proc max:640
    ↪ ms, min:8ms, p80:276ms, p95:480ms, iters:18695, tasks:56
    ↪
    ↪ | funcs:count(1)->Column#13
    ↪
    ↪ | N/A       | N/A  |
|     └─Selection_19          | 250.00   | 11409    | cop[tikv] |               | proc max:640
    ↪ ms, min:8ms, p80:276ms, p95:476ms, iters:18695, tasks:56
    ↪
    ↪ | ge(bikeshare.trips.start_date, 2017-07-01 00:00:00.000000), le(bikeshare.trips.
    ↪ start_date, 2017-07-01 23:59:59.000000) | N/A       | N/A  |
|       └─TableFullScan_18    | 10000.00 | 19117643 | cop[tikv] | table:trips   | proc max:612
    ↪ ms, min:8ms, p80:248ms, p95:460ms, iters:18695, tasks:56
    ↪
    ↪ | keep order:false, stats:pseudo
    ↪                                                                                              |
    ↪  N/A       | N/A  |
+--
    ↪ --------------------------+----------+----------+----------+--------------+-----------------------
    ↪
```

```
5 rows in set (1.03 sec)
```

执行以上示例查询耗时 1.03 秒，说明执行性能较为理想。

以上 EXPLAIN ANALYZE 的结果中，actRows 表明一些 estRows 预估数不准确（预估返回 10000 行数据但实际返回 19117643 行）。└─TableFullScan_18 算子的 operator info 列 (stats:pseudo) 信息也表明该算子的预估数不准确。

如果先执行ANALYZE TABLE 再执行 EXPLAIN ANALYZE，预估数与实际数会更接近：

```
ANALYZE TABLE trips;
EXPLAIN ANALYZE SELECT count(*) FROM trips WHERE start_date BETWEEN '2017-07-01 00:00:00' AND '
    ↪ 2017-07-01 23:59:59';
```

```
Query OK, 0 rows affected (10.22 sec)

+--
    ↪ ---------------------------+-------------+----------+-----------+---------------+-----------------------
    ↪
| id                           | estRows     | actRows  | task      | access object | execution
    ↪ info
    ↪
    ↪ | operator info
    ↪
    ↪ | memory    | disk |
+--
    ↪ ---------------------------+-------------+----------+-----------+---------------+-----------------------
    ↪
| StreamAgg_20                 | 1.00        | 1        | root      |               | time
    ↪ :926.393612ms, loops:2
    ↪
    ↪ | funcs:count(Column#13)->Column#11
    ↪                                                                                                 |
    ↪ 632 Bytes | N/A  |
| └─TableReader_21             | 1.00        | 56       | root      |               | time
    ↪ :926.384792ms, loops:2, cop_task: {num: 56, max: 850.94424ms, min: 6.042079ms, avg:
    ↪ 234.987725ms, p95: 495.474806ms, max_proc_keys: 910371, p95_proc_keys: 704775, tot_proc:
    ↪ 10.656s, tot_wait: 904ms, rpc_num: 56, rpc_time: 13.158911952s} | data:StreamAgg_9
    ↪
    ↪ | 328 Bytes | N/A  |
|   └─StreamAgg_9              | 1.00        | 56       | cop[tikv] |               | proc max
    ↪ :592ms, min:4ms, p80:244ms, p95:480ms, iters:18695, tasks:56
    ↪
    ↪ | funcs:count(1)->Column#13
    ↪
    ↪ | N/A       | N/A  |
```

```
|     └─Selection_19           | 432.89      | 11409    | cop[tikv] |               | proc max
    ↪ :592ms, min:4ms, p80:244ms, p95:480ms, iters:18695, tasks:56
    ↪
    ↪ | ge(bikeshare.trips.start_date, 2017-07-01 00:00:00.000000), le(bikeshare.trips.
    ↪ start_date, 2017-07-01 23:59:59.000000) | N/A       | N/A  |
|       └─TableFullScan_18     | 19117643.00 | 19117643 | cop[tikv] | table:trips   | proc max
    ↪ :564ms, min:4ms, p80:228ms, p95:456ms, iters:18695, tasks:56
    ↪
    ↪ | keep order:false
    ↪
    ↪ | N/A       | N/A  |
+--
    ↪ ---------------------------+-------------+----------+-----------+---------------+-----------------------
    ↪
5 rows in set (0.93 sec)
```

执行 ANALYZE TABLE 后，可以看到 └─TableFullScan_18 算子的预估行数是准确的，└─Selection_19 算子的预估行数也更接近实际行数。以上两个示例中的执行计划（即 TiDB 执行查询所使用的一组算子）未改变，但过时的统计信息常常导致 TiDB 选择到非最优的执行计划。

除 ANALYZE TABLE 外，达到tidb_auto_analyze_ratio 阈值后，TiDB 会自动在后台重新生成统计数据。若要查看 TiDB 有多接近该阈值（即 TiDB 判断统计数据有多健康），可执行SHOW STATS_HEALTHY 语句。

```
SHOW STATS_HEALTHY;
```

```
+-----------+------------+----------------+---------+
| Db_name   | Table_name | Partition_name | Healthy |
+-----------+------------+----------------+---------+
| bikeshare | trips      |                |     100 |
+-----------+------------+----------------+---------+
1 row in set (0.00 sec)
```

9.3.2.2.2 确定优化方案

当前执行计划是有效率的：

- 大部分任务是在 TiKV 内处理的，需要通过网络传输给 TiDB 处理的仅有 56 行数据，每行都满足过滤条件，而且都很短。
- 在 TiDB (StreamAgg_20) 中和在 TiKV (└─StreamAgg_9) 中汇总行数都使用了 Stream Aggregate，该算法在内存使用方面很有效率。

当前执行计划存在的最大问题在于谓词 start_date BETWEEN '2017-07-01 00:00:00' AND '2017-07-01 ↪ 23:59:59' 并未立即生效，先是 TableFullScan 算子读取所有行数据，然后才进行过滤选择。可以在 SHOW CREATE TABLE trips 的返回结果中找出问题原因：

```
SHOW CREATE TABLE trips\G
```

```
*************************** 1. row ***************************
       Table: trips
Create Table: CREATE TABLE `trips` (
  `trip_id` bigint(20) NOT NULL AUTO_INCREMENT,
  `duration` int(11) NOT NULL,
  `start_date` datetime DEFAULT NULL,
  `end_date` datetime DEFAULT NULL,
  `start_station_number` int(11) DEFAULT NULL,
  `start_station` varchar(255) DEFAULT NULL,
  `end_station_number` int(11) DEFAULT NULL,
  `end_station` varchar(255) DEFAULT NULL,
  `bike_number` varchar(255) DEFAULT NULL,
  `member_type` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`trip_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin AUTO_INCREMENT=20477318
1 row in set (0.00 sec)
```

以上返回结果显示，start_date 列没有索引。要将该谓词下推到 index reader 算子，还需要一个索引。添加索引如下：

```
ALTER TABLE trips ADD INDEX (start_date);
```

```
Query OK, 0 rows affected (2 min 10.23 sec)
```

> **注意：**
>
> 你可通过执行ADMIN SHOW DDL JOBS 语句来查看 DDL 任务的进度。TiDB 中的默认值的设置较为保守，因此添加索引不会对生产环境下的负载造成太大影响。测试环境下，可以考虑调大tidb_ddl_reorg_batch_size 和tidb_ddl_reorg_worker_cnt 的值。在参照系统上，将批处理大小设为 10240，将 worker count 并发度设置为 32，该系统可获得 10 倍的性能提升（较之使用默认值）。

添加索引后，可以使用 EXPLAIN 重复该查询。在以下返回结果中，可见 TiDB 选择了新的执行计划，而且不再使用 TableFullScan 和 Selection 算子。

```
EXPLAIN SELECT count(*) FROM trips WHERE start_date BETWEEN '2017-07-01 00:00:00' AND '2017-07-01
    ↪  23:59:59';
```

```
+--
    ↪ -------------------------+---------+-----------+-----------------------------------------+----------
    ↪
| id                          | estRows | task      | access object                           |
    ↪  operator info                                                  |
```

```
+--
    ↪ -------------------------+---------+-----------+----------------------------------------+-----------
    ↪
| StreamAgg_17                | 1.00    | root      |                                        |
    ↪  funcs:count(Column#13)->Column#11                                |
| └─IndexReader_18            | 1.00    | root      |
    ↪ | index:StreamAgg_9                                              |
|   └─StreamAgg_9             | 1.00    | cop[tikv] |
    ↪ | funcs:count(1)->Column#13                                       |
|     └─IndexRangeScan_16     | 8471.88 | cop[tikv] | table:trips, index:start_date(start_date)
    ↪ | range:[2017-07-01 00:00:00,2017-07-01 23:59:59], keep order:false |
+--
    ↪ -------------------------+---------+-----------+----------------------------------------+-----------
    ↪
4 rows in set (0.00 sec)
```

若要比较实际的执行时间，可再次使用 EXPLAIN ANALYZE 语句：

```
EXPLAIN ANALYZE SELECT count(*) FROM trips WHERE start_date BETWEEN '2017-07-01 00:00:00' AND '
    ↪ 2017-07-01 23:59:59';
```

```
+--
    ↪ -------------------------+---------+---------+-----------+-------------------------------------+-
    ↪
| id                          | estRows | actRows | task      | access object
    ↪                           | execution info
    ↪
    ↪ | operator info                                                 | memory    | disk |
+--
    ↪ -------------------------+---------+---------+-----------+-------------------------------------+-
    ↪
| StreamAgg_17                | 1.00    | 1       | root      |
    ↪                                         | time:4.516728ms, loops:2
    ↪                                                                                 |
    ↪  funcs:count(Column#13)->Column#11                                | 372 Bytes | N/A  |
| └─IndexReader_18            | 1.00    | 1       | root      |
    ↪                                         | time:4.514278ms, loops:2, cop_task: {num: 1,
    ↪ max:4.462288ms, proc_keys: 11409, rpc_num: 1, rpc_time: 4.457148ms} | index:StreamAgg_9
    ↪                                              | 238 Bytes | N/A  |
|   └─StreamAgg_9             | 1.00    | 1       | cop[tikv] |
    ↪                                         | time:4ms, loops:12
    ↪
    ↪ | funcs:count(1)->Column#13                                       | N/A       | N/A  |
|     └─IndexRangeScan_16     | 8471.88 | 11409   | cop[tikv] | table:trips, index:start_date(
    ↪ start_date) | time:4ms, loops:12
    ↪
```

```
    ↪ | range:[2017-07-01 00:00:00,2017-07-01 23:59:59], keep order:false | N/A       | N/A  |
+--
    ↪ -------------------------+---------+---------+-----------+-----------------------------------+-
    ↪
4 rows in set (0.00 sec)
```

从以上结果可看出，查询时间已从 1.03 秒减少到 0.0 秒。

> **注意：**
>
> 以上示例另一个可用的优化方案是coprocessor cache。如果你无法添加索引，可考虑开启 coprocessor cache 功能。开启后，只要算子上次执行以来 Region 未作更改，TiKV 将从缓存中返回值。这也有助于减少 TableFullScan 和 Selection 算子的大部分运算成本。

9.3.2.3 用 EXPLAIN 查看索引查询的执行计划

SQL 查询可能会使用索引，可以通过 EXPLAIN 语句来查看索引查询的执行计划。本文提供多个示例，以帮助用户理解索引查询是如何执行的。

TiDB 支持以下使用索引的算子来提升查询速度：

- IndexLookup
- IndexReader
- Point_Get 和 Batch_Point_Get
- IndexFullScan

本文档中的示例都基于以下数据：

```
CREATE TABLE t1 (
 id INT NOT NULL PRIMARY KEY auto_increment,
 intkey INT NOT NULL,
 pad1 VARBINARY(1024),
 INDEX (intkey)
);

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1024), RANDOM_BYTES(1024) FROM dual;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1024), RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 10000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1024), RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 10000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1024), RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 10000;
```

9.3.2.3.1 IndexLookup

TiDB 从二级索引检索数据时会使用 IndexLookup 算子。例如，以下所有查询均会在 intkey 列的索引上使用 IndexLookup 算子：

```
EXPLAIN SELECT * FROM t1 WHERE intkey = 123;
EXPLAIN SELECT * FROM t1 WHERE intkey < 10;
EXPLAIN SELECT * FROM t1 WHERE intkey BETWEEN 300 AND 310;
EXPLAIN SELECT * FROM t1 WHERE intkey IN (123,29,98);
EXPLAIN SELECT * FROM t1 WHERE intkey >= 99 AND intkey <= 103;
```

```
+--
    ↪ ----------------------------+---------+-----------+-------------------------------+-------------------
    ↪
| id                            | estRows | task      | access object                 | operator
    ↪  info                       |
+--
    ↪ ----------------------------+---------+-----------+-------------------------------+-------------------
    ↪
| IndexLookUp_10                | 1.00    | root      |                               |
    ↪                                       |
| ├─IndexRangeScan_8(Build)     | 1.00    | cop[tikv] | table:t1, index:intkey(intkey) | range
    ↪ :[123,123], keep order:false |
| └─TableRowIDScan_9(Probe)     | 1.00    | cop[tikv] | table:t1                       | keep
    ↪ order:false                   |
+--
    ↪ ----------------------------+---------+-----------+-------------------------------+-------------------
    ↪
3 rows in set (0.00 sec)

+--
    ↪ ----------------------------+---------+-----------+-------------------------------+-------------------
    ↪
| id                            | estRows | task      | access object                 | operator
    ↪  info                       |
+--
    ↪ ----------------------------+---------+-----------+-------------------------------+-------------------
    ↪
| IndexLookUp_10                | 3.60    | root      |                               |
    ↪                                       |
| ├─IndexRangeScan_8(Build)     | 3.60    | cop[tikv] | table:t1, index:intkey(intkey) | range
    ↪ :[-inf,10), keep order:false |
| └─TableRowIDScan_9(Probe)     | 3.60    | cop[tikv] | table:t1                       | keep
    ↪ order:false                   |
+--
    ↪ ----------------------------+---------+-----------+-------------------------------+-------------------
```

```
    ↪
3 rows in set (0.00 sec)

+--
    ↪ ----------------------------+---------+-----------+-------------------------------+-------------------
    ↪
| id                            | estRows | task      | access object                 | operator
    ↪  info                      |
+--
    ↪ ----------------------------+---------+-----------+-------------------------------+-------------------
    ↪
| IndexLookUp_10                | 5.67    | root      |                               |
    ↪                                      |
| ├─IndexRangeScan_8(Build)     | 5.67    | cop[tikv] | table:t1, index:intkey(intkey) | range
    ↪ :[300,310], keep order:false |
| └─TableRowIDScan_9(Probe)     | 5.67    | cop[tikv] | table:t1                      | keep
    ↪ order:false                  |
+--
    ↪ ----------------------------+---------+-----------+-------------------------------+-------------------
    ↪
3 rows in set (0.00 sec)

+--
    ↪ ----------------------------+---------+-----------+-------------------------------+-------------------
    ↪
| id                            | estRows | task      | access object                 | operator
    ↪  info                                        |
+--
    ↪ ----------------------------+---------+-----------+-------------------------------+-------------------
    ↪
| IndexLookUp_10                | 4.00    | root      |                               |
    ↪                                                    |
| ├─IndexRangeScan_8(Build)     | 4.00    | cop[tikv] | table:t1, index:intkey(intkey) | range
    ↪ :[29,29], [98,98], [123,123], keep order:false |
| └─TableRowIDScan_9(Probe)     | 4.00    | cop[tikv] | table:t1                      | keep
    ↪ order:false                                  |
+--
    ↪ ----------------------------+---------+-----------+-------------------------------+-------------------
    ↪
3 rows in set (0.00 sec)

+--
    ↪ ----------------------------+---------+-----------+-------------------------------+-------------------
    ↪
| id                            | estRows | task      | access object                 | operator
```

```
    ↪  info                    |
+--
    ↪ ----------------------------+---------+-----------+-------------------------------+-------------------
    ↪
| IndexLookUp_10                | 6.00    | root      |                               |
    ↪                                    |
| ├─IndexRangeScan_8(Build)     | 6.00    | cop[tikv] | table:t1, index:intkey(intkey) | range
    ↪ :[99,103], keep order:false |
| └─TableRowIDScan_9(Probe)     | 6.00    | cop[tikv] | table:t1                      | keep
    ↪ order:false                 |
+--
    ↪ ----------------------------+---------+-----------+-------------------------------+-------------------
    ↪
3 rows in set (0.00 sec)
```

IndexLookup 算子有以下两个子节点：

- ├─IndexRangeScan_8(Build) 算子节点对 intkey 列的索引执行范围扫描，并检索内部的 RowID 值（对此表而言，即为主键）。
- └─TableRowIDScan_9(Probe) 算子节点随后从表数据中检索整行。

IndexLookup 任务分以上两步执行。如果满足条件的行较多，SQL 优化器可能会根据统计信息选择使用 TableFullScan 算子。在以下示例中，很多行都满足 intkey > 100 这一条件，因此优化器选择了 TableFullScan：

```
EXPLAIN SELECT * FROM t1 WHERE intkey > 100;
```

```
+-------------------------+---------+-----------+---------------+-------------------------+
| id                      | estRows | task      | access object | operator info           |
+-------------------------+---------+-----------+---------------+-------------------------+
| TableReader_7           | 898.50  | root      |               | data:Selection_6        |
| └─Selection_6           | 898.50  | cop[tikv] |               | gt(test.t1.intkey, 100) |
|   └─TableFullScan_5     | 1010.00 | cop[tikv] | table:t1      | keep order:false        |
+-------------------------+---------+-----------+---------------+-------------------------+
3 rows in set (0.00 sec)
```

IndexLookup 算子能在带索引的列上有效优化 LIMIT：

```
EXPLAIN SELECT * FROM t1 ORDER BY intkey DESC LIMIT 10;
```

```
+--
    ↪ ----------------------------+---------+-----------+-------------------------------+-------------------
    ↪
| id                            | estRows | task      | access object                 |
    ↪ operator info                      |
+--
    ↪ ----------------------------+---------+-----------+-------------------------------+-------------------
    ↪
```

```
| IndexLookUp_21                | 10.00   | root      |                                    | limit
    ↪ embedded(offset:0, count:10) |
| ├─Limit_20(Build)             | 10.00   | cop[tikv] |                                    | offset
    ↪ :0, count:10                |
| │ └─IndexFullScan_18          | 10.00   | cop[tikv] | table:t1, index:intkey(intkey) | keep
    ↪ order:true, desc            |
| └─TableRowIDScan_19(Probe)    | 10.00   | cop[tikv] | table:t1                           | keep
    ↪ order:false, stats:pseudo   |
+--
    ↪ ----------------------------+---------+-----------+--------------------------------+-----------------
    ↪
4 rows in set (0.00 sec)
```

以上示例中，TiDB 从 intkey 索引读取最后 20 行，然后从表数据中检索这些行的 RowID 值。

9.3.2.3.2 IndexReader

TiDB 支持覆盖索引优化 (covering index optimization)。如果 TiDB 能从索引中检索出所有行，就会跳过 IndexLookup 任务中通常所需的第二步（即从表数据中检索整行）。示例如下：

```
EXPLAIN SELECT * FROM t1 WHERE intkey = 123;
EXPLAIN SELECT id FROM t1 WHERE intkey = 123;
```

```
+--
    ↪ ----------------------------+---------+-----------+--------------------------------+-----------------
    ↪
| id                            | estRows | task      | access object                  | operator
    ↪  info                       |
+--
    ↪ ----------------------------+---------+-----------+--------------------------------+-----------------
    ↪
| IndexLookUp_10                | 1.00    | root      |                                |
    ↪                                       |
| ├─IndexRangeScan_8(Build)     | 1.00    | cop[tikv] | table:t1, index:intkey(intkey) | range
    ↪ :[123,123], keep order:false |
| └─TableRowIDScan_9(Probe)     | 1.00    | cop[tikv] | table:t1                       | keep
    ↪ order:false                 |
+--
    ↪ ----------------------------+---------+-----------+--------------------------------+-----------------
    ↪
3 rows in set (0.00 sec)

+--
    ↪ ----------------------+---------+-----------+--------------------------------+-----------------
    ↪
```

```
| id                        | estRows | task      | access object                    | operator info
    ↪                        |
+--
    ↪ -----------------------+---------+-----------+----------------------------------+-----------------------
    ↪
| Projection_4              | 1.00    | root      |                                  | test.t1.id
    ↪                          |
| └─IndexReader_6           | 1.00    | root      |                                   | index:
    ↪ IndexRangeScan_5             |
|   └─IndexRangeScan_5      | 1.00    | cop[tikv] | table:t1, index:intkey(intkey) | range
    ↪ :[123,123], keep order:false |
+--
    ↪ -----------------------+---------+-----------+--------------------------------+-----------------------
    ↪
3 rows in set (0.00 sec)
```

以上结果中，id 也是内部的 RowID 值，因此 id 也存储在 intkey 索引中。部分 └─IndexRangeScan_5 任务使用 intkey 索引后，可直接返回 RowID 值。

9.3.2.3.3 Point_Get 和 Batch_Point_Get

TiDB 直接从主键或唯一键检索数据时会使用 Point_Get 或 Batch_Point_Get 算子。这两个算子比 IndexLookup 更有效率。示例如下：

```
EXPLAIN SELECT * FROM t1 WHERE id = 1234;
EXPLAIN SELECT * FROM t1 WHERE id IN (1234,123);

ALTER TABLE t1 ADD unique_key INT;
UPDATE t1 SET unique_key = id;
ALTER TABLE t1 ADD UNIQUE KEY (unique_key);

EXPLAIN SELECT * FROM t1 WHERE unique_key = 1234;
EXPLAIN SELECT * FROM t1 WHERE unique_key IN (1234, 123);
```

```
+-------------+---------+------+---------------+---------------+
| id          | estRows | task | access object | operator info |
+-------------+---------+------+---------------+---------------+
| Point_Get_1 | 1.00    | root | table:t1      | handle:1234   |
+-------------+---------+------+---------------+---------------+
1 row in set (0.00 sec)

+--
    ↪ ----------------+---------+------+---------------+-------------------------------------------------+
    ↪
| id                | estRows | task | access object | operator info
    ↪                                    |
```

```
+--
    ↪ -----------------+---------+------+---------------+-------------------------------------------------+
    ↪
| Batch_Point_Get_1 | 2.00    | root | table:t1      | handle:[1234 123], keep order:false, desc:
    ↪ false |
+--
    ↪ -----------------+---------+------+---------------+-------------------------------------------------+
    ↪
1 row in set (0.00 sec)

Query OK, 0 rows affected (0.27 sec)

Query OK, 1010 rows affected (0.06 sec)
Rows matched: 1010  Changed: 1010  Warnings: 0

Query OK, 0 rows affected (0.37 sec)

+-------------+---------+------+-----------------------------------------+---------------+
| id          | estRows | task | access object                           | operator info |
+-------------+---------+------+-----------------------------------------+---------------+
| Point_Get_1 | 1.00    | root | table:t1, index:unique_key(unique_key) |               |
+-------------+---------+------+-----------------------------------------+---------------+
1 row in set (0.00 sec)

+--
    ↪ -----------------+---------+------+-----------------------------------------+-----------------------------
    ↪
| id                | estRows | task | access object                           | operator info
    ↪                 |
+--
    ↪ -----------------+---------+------+-----------------------------------------+-----------------------------
    ↪
| Batch_Point_Get_1 | 2.00    | root | table:t1, index:unique_key(unique_key) | keep order:false,
    ↪  desc:false |
+--
    ↪ -----------------+---------+------+-----------------------------------------+-----------------------------
    ↪
1 row in set (0.00 sec)
```

9.3.2.3.4 IndexFullScan

索引是有序的，所以优化器可以使用 `IndexFullScan` 算子来优化常见的查询，例如在索引值上使用 `MIN` 或 `Max` 函数：

```
EXPLAIN SELECT MIN(intkey) FROM t1;
```

```
EXPLAIN SELECT MAX(intkey) FROM t1;
```

```
+--
    ↪ ---------------------------+---------+-----------+--------------------------------+-----------------------
    ↪
| id                           | estRows | task      | access object                  | operator
    ↪ info                       |
+--
    ↪ ---------------------------+---------+-----------+--------------------------------+-----------------------
    ↪
| StreamAgg_12                 | 1.00    | root      |                                | funcs:min
    ↪ (test.t1.intkey)->Column#4 |
| └─Limit_16                   | 1.00    | root      |                                 | offset
    ↪ :0, count:1                    |
|   └─IndexReader_29           | 1.00    | root      |                                 | index:
    ↪ Limit_28                       |
|     └─Limit_28               | 1.00    | cop[tikv] |                                 | offset
    ↪ :0, count:1                    |
|       └─IndexFullScan_27     | 1.00    | cop[tikv] | table:t1, index:intkey(intkey) | keep
    ↪ order:true                      |
+--
    ↪ ---------------------------+---------+-----------+--------------------------------+-----------------------
    ↪
5 rows in set (0.00 sec)

+--
    ↪ ---------------------------+---------+-----------+--------------------------------+-----------------------
    ↪
| id                           | estRows | task      | access object                  | operator
    ↪ info                       |
+--
    ↪ ---------------------------+---------+-----------+--------------------------------+-----------------------
    ↪
| StreamAgg_12                 | 1.00    | root      |                                | funcs:max
    ↪ (test.t1.intkey)->Column#4 |
| └─Limit_16                   | 1.00    | root      |                                 | offset
    ↪ :0, count:1                    |
|   └─IndexReader_29           | 1.00    | root      |                                 | index:
    ↪ Limit_28                       |
|     └─Limit_28               | 1.00    | cop[tikv] |                                 | offset
    ↪ :0, count:1                    |
|       └─IndexFullScan_27     | 1.00    | cop[tikv] | table:t1, index:intkey(intkey) | keep
    ↪ order:true, desc               |
+--
    ↪ ---------------------------+---------+-----------+--------------------------------+-----------------------
```

```
    ↪
5 rows in set (0.00 sec)
```

以上语句的执行过程中，TiDB 在每一个 TiKV Region 上执行 IndexFullScan 操作。虽然算子名为 FullScan 即全扫描，TiDB 只读取第一行 (└─Limit_28)。每个 TiKV Region 返回各自的 MIN 或 MAX 值给 TiDB，TiDB 再执行流聚合运算来过滤出一行数据。即使表为空，带 MAX 或 MIN 函数的流聚合运算也能保证返回 NULL 值。

相反，在没有索引的值上执行 MIN 函数会在每一个 TiKV Region 上执行 TableFullScan 操作。该查询会要求在 TiKV 中扫描所有行，但 TopN 计算可保证每个 TiKV Region 只返回一行数据给 TiDB。尽管 TopN 能减少 TiDB 和 TiKV 之间的多余数据传输，但该查询的效率仍远不及以上示例 (MIN 能够使用索引)。

```
EXPLAIN SELECT MIN(pad1) FROM t1;
```

```
+--
    ↪ ----------------------------+---------+-----------+---------------+--------------------------------+
    ↪
| id                            | estRows | task      | access object | operator info
    ↪                      |
+--
    ↪ ----------------------------+---------+-----------+---------------+--------------------------------+
    ↪
| StreamAgg_13                  | 1.00    | root      |               | funcs:min(test.t1.pad1)
    ↪ ->Column#4 |
| └─TopN_14                     | 1.00    | root      |               | test.t1.pad1, offset:0,
    ↪  count:1   |
|   └─TableReader_23            | 1.00    | root      |               | data:TopN_22
    ↪                       |
|     └─TopN_22                 | 1.00    | cop[tikv] |               | test.t1.pad1, offset:0,
    ↪  count:1   |
|       └─Selection_21          | 1008.99 | cop[tikv] |               | not(isnull(test.t1.pad1
    ↪ ))         |
|         └─TableFullScan_20    | 1010.00 | cop[tikv] | table:t1      | keep order:false
    ↪                   |
+--
    ↪ ----------------------------+---------+-----------+---------------+--------------------------------+
    ↪
6 rows in set (0.00 sec)
```

执行以下语句时，TiDB 将使用 IndexFullScan 算子扫描索引中的每一行：

```
EXPLAIN SELECT SUM(intkey) FROM t1;
EXPLAIN SELECT AVG(intkey) FROM t1;
```

```
+--
    ↪ -------------------------+---------+-----------+-------------------------------+-----------------------
    ↪
```

```
| id                          | estRows | task      | access object                   | operator
    ↪ info                        |
+--
    ↪ -------------------------+---------+-----------+---------------------------------+----------------------
    ↪
| StreamAgg_20                | 1.00    | root      |                                 | funcs:sum(
    ↪ Column#6)->Column#4         |
| └─IndexReader_21            | 1.00    | root      |                                 | index:
    ↪ StreamAgg_8                    |
|   └─StreamAgg_8             | 1.00    | cop[tikv] |                                 | funcs:sum(
    ↪ test.t1.intkey)->Column#6 |
|     └─IndexFullScan_19      | 1010.00 | cop[tikv] | table:t1, index:intkey(intkey) | keep order
    ↪ :false                      |
+--
    ↪ -------------------------+---------+-----------+---------------------------------+----------------------
    ↪
4 rows in set (0.00 sec)

+--
    ↪ -------------------------+---------+-----------+---------------------------------+----------------------
    ↪
| id                          | estRows | task      | access object                   | operator
    ↪ info                                                          |
+--
    ↪ -------------------------+---------+-----------+---------------------------------+----------------------
    ↪
| StreamAgg_20                | 1.00    | root      |                                 | funcs:avg(
    ↪ Column#7, Column#8)->Column#4                                 |
| └─IndexReader_21            | 1.00    | root      |                                 | index:
    ↪ StreamAgg_8                                                      |
|   └─StreamAgg_8             | 1.00    | cop[tikv] |                                 | funcs:
    ↪ count(test.t1.intkey)->Column#7, funcs:sum(test.t1.intkey)->Column#8 |
|     └─IndexFullScan_19      | 1010.00 | cop[tikv] | table:t1, index:intkey(intkey) | keep order
    ↪ :false                                                        |
+--
    ↪ -------------------------+---------+-----------+---------------------------------+----------------------
    ↪
4 rows in set (0.00 sec)
```

以上示例中，IndexFullScan 比 TableFullScan 更有效率，因为 (intkey + RowID) 索引中值的长度小于整行的长度。

以下语句不支持使用 IndexFullScan 算子，因为涉及该表中的其他列：

```
EXPLAIN SELECT AVG(intkey), ANY_VALUE(pad1) FROM t1;
```

```
+--
    ↪ ---------------------------+---------+-----------+--------------+--------------------------------
    ↪
| id                           | estRows | task      | access object | operator info
    ↪
    ↪ |
+--
    ↪ ---------------------------+---------+-----------+--------------+--------------------------------
    ↪
| Projection_4                 | 1.00    | root      |               | Column#4, any_value(test.
    ↪ t1.pad1)->Column#5
    ↪                                                                                              |
| └─StreamAgg_16               | 1.00    | root      |               | funcs:avg(Column#10,
    ↪ Column#11)->Column#4, funcs:firstrow(Column#12)->test.t1.pad1
    ↪                                      |
|   └─TableReader_17           | 1.00    | root      |               | data:StreamAgg_8
    ↪
    ↪ |
|     └─StreamAgg_8            | 1.00    | cop[tikv] |               | funcs:count(test.t1.
    ↪ intkey)->Column#10, funcs:sum(test.t1.intkey)->Column#11, funcs:firstrow(test.t1.pad1)->
    ↪ Column#12 |
|       └─TableFullScan_15     | 1010.00 | cop[tikv] | table:t1      | keep order:false
    ↪
    ↪ |
+--
    ↪ ---------------------------+---------+-----------+--------------+--------------------------------
    ↪
5 rows in set (0.00 sec)
```

9.3.2.3.5 其他类型查询的执行计划

- Join 查询的执行计划
- 子查询的执行计划
- 聚合查询的执行计划
- 视图查询的执行计划
- 分区查询的执行计划
- 开启 IndexMerge 查询的执行计划

9.3.2.4 用 EXPLAIN 查看 JOIN 查询的执行计划

SQL 查询中可能会使用 JOIN 进行表连接，可以通过 EXPLAIN 语句来查看 JOIN 查询的执行计划。本文提供多个示例，以帮助用户理解表连接查询是如何执行的。

在 TiDB 中，SQL 优化器需要确定数据表的连接顺序，且要判断对于某条特定的 SQL 语句，哪一种 Join 算法最为高效。

本文档使用的示例数据如下:

```
CREATE TABLE t1 (id BIGINT NOT NULL PRIMARY KEY auto_increment, pad1 BLOB, pad2 BLOB, pad3 BLOB,
    ↪ int_col INT NOT NULL DEFAULT 0);
CREATE TABLE t2 (id BIGINT NOT NULL PRIMARY KEY auto_increment, t1_id BIGINT NOT NULL, pad1 BLOB,
    ↪  pad2 BLOB, pad3 BLOB, INDEX(t1_id));
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM
    ↪ dual;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
```

```
UPDATE t1 SET int_col = 1 WHERE pad1 = (SELECT pad1 FROM t1 ORDER BY RAND() LIMIT 1);
SELECT SLEEP(1);
ANALYZE TABLE t1, t2;
```

9.3.2.4.1 Index Join

如果预计需要连接的行数较少（一般小于 1 万行），推荐使用 Index Join 算法。这个算法与 MySQL 主要使用的 Join 算法类似。在下表的示例中，TableReader_28(Build) 算子首先读取表 t1，然后根据在 t1 中匹配到的每行数据，依次探查表 t2 中的数据：

```
EXPLAIN SELECT /*+ INL_JOIN(t1, t2) */ * FROM t1 INNER JOIN t2 ON t1.id = t2.t1_id;
```

```
+--
    ↪ ------------------------------+-----------+-----------+-----------------------------+----------------
    ↪
| id                            | estRows   | task      | access object               |
    ↪ operator info                                                                   |
+--
    ↪ ------------------------------+-----------+-----------+-----------------------------+----------------
    ↪
| IndexJoin_10                  | 180000.00 | root      |                             | inner
    ↪ join, inner:IndexLookUp_9, outer key:test.t1.id, inner key:test.t2.t1_id |
| ├─TableReader_28(Build)       | 142020.00 | root      |                             | data:
    ↪ TableFullScan_27                                                        |
| │ └─TableFullScan_27          | 142020.00 | cop[tikv] | table:t1                    | keep
    ↪ order:false                                                             |
| └─IndexLookUp_9(Probe)        | 1.27      | root      |                             |
    ↪                                                                                 |
|   ├─IndexRangeScan_7(Build)   | 1.27      | cop[tikv] | table:t2, index:t1_id(t1_id) | range
    ↪ : decided by [eq(test.t2.t1_id, test.t1.id)], keep order:false          |
|   └─TableRowIDScan_8(Probe)   | 1.27      | cop[tikv] | table:t2                    | keep
    ↪ order:false                                                             |
+--
    ↪ ------------------------------+-----------+-----------+-----------------------------+----------------
    ↪
6 rows in set (0.00 sec)
```

Index Join 算法对内存消耗较小，但如果需要执行大量探查操作，运行速度可能会慢于其他 Join 算法。以下面这条查询语句为例：

```
SELECT * FROM t1 INNER JOIN t2 ON t1.id=t2.t1_id WHERE t1.pad1 = 'value' and t2.pad1='value';
```

在 Inner Join 操作中，TiDB 会先执行 Join Reorder 算法，所以不能确定会先读取 t1 还是 t2。假设 TiDB 先读取了 t1 来构建 Build 端，那么 TiDB 会在探查 t2 前先根据谓词 t1.col = 'value' 筛选数据，但接下来每次探查 t2 时都要应用谓词 t2.col='value'。所以对于这条语句，Index Join 算法可能不如其他 Join 算法高效。

但如果 Build 端的数据量比 Probe 端小，且 Probe 端的数据已预先建立了索引，那么这种情况下 Index Join 算法效率更高。在下面这段查询语句中，因为 Index Join 比 Hash Join 效率低，所以 SQL 优化器选择了 Hash Join 算法：

```
-- 删除已有索引
ALTER TABLE t2 DROP INDEX t1_id;

EXPLAIN ANALYZE SELECT /*+ INL_JOIN(t1, t2) */  * FROM t1 INNER JOIN t2 ON t1.id = t2.t1_id WHERE
    ↪  t1.int_col = 1;
EXPLAIN ANALYZE SELECT /*+ HASH_JOIN(t1, t2) */  * FROM t1 INNER JOIN t2 ON t1.id = t2.t1_id
    ↪ WHERE t1.int_col = 1;
EXPLAIN ANALYZE SELECT * FROM t1 INNER JOIN t2 ON t1.id = t2.t1_id WHERE t1.int_col = 1;
```

```
Query OK, 0 rows affected (0.29 sec)

+--
    ↪ --------------------------+----------+---------+-----------+--------------+--------------------------
    ↪
| id                          | estRows  | actRows | task      | access object | execution info
    ↪
    ↪ | operator info                                                                 | memory
    ↪                 | disk |
+--
    ↪ --------------------------+----------+---------+-----------+--------------+--------------------------
    ↪
| IndexJoin_13                | 90000.00 | 20000   | root      |               | time:613.19955ms
    ↪ , loops:21, inner:{total:42.494047ms, concurrency:5, task:12, construct:33.149671ms,
    ↪ fetch:9.322956ms, build:8.66µs}, probe:32.435355ms
    ↪
    ↪ | inner join, inner:TableReader_9, outer key:test.t2.t1_id, inner key:test.t1.id |
    ↪ 269.63341903686523 MB | N/A  |
| ├─TableReader_19(Build)     | 90000.00 | 90000   | root      |               | time:586.613252
    ↪ ms, loops:95, cop_task: {num: 3, max: 205.893949ms, min: 185.051354ms, avg: 194.878702ms,
    ↪  p95: 205.893949ms, max_proc_keys: 31715, p95_proc_keys: 31715, tot_proc: 332ms, tot_wait
    ↪ : 4ms, rpc_num: 4, rpc_time: 584.907774ms, copr_cache_hit_ratio: 0.00}, backoff{
    ↪ regionMiss: 2ms} | data:TableFullScan_18
    ↪                                                          | 182.624906539917 MB   | N/A  |
| │ └─TableFullScan_18        | 90000.00 | 90000   | cop[tikv] | table:t2      | time:0ns, loops
    ↪ :0, tikv_task:{proc max:72ms, min:64ms, p80:72ms, p95:72ms, iters:102, tasks:3}
    ↪
    ↪ | keep order:false                                                              | N/A
    ↪                   | N/A  |
| └─TableReader_9(Probe)      | 0.00     | 5       | root      |               | time:8.432051ms
    ↪ , loops:14, cop_task: {num: 14, max: 629.805µs, min: 226.129µs, avg: 420.979µs, p95:
    ↪ 629.805µs, max_proc_keys: 4, p95_proc_keys: 4, rpc_num: 15, rpc_time: 5.953229ms,
    ↪ copr_cache_hit_ratio: 0.00}
    ↪                                                                               | data:
```

```
    ↪ Selection_8                                                    | N/A
    ↪                   | N/A  |
|   └─Selection_8            | 0.00     | 5       | cop[tikv] |               | time:0ns, loops
    ↪ :0, tikv_task:{proc max:0s, min:0s, p80:0s, p95:0s, iters:14, tasks:14}
    ↪
    ↪ | eq(test.t1.int_col, 1)                                                | N/A
    ↪                   | N/A  |
|     └─TableRangeScan_7      | 1.00     | 25      | cop[tikv] | table:t1      | time:0ns, loops
    ↪ :0, tikv_task:{proc max:0s, min:0s, p80:0s, p95:0s, iters:14, tasks:14}
    ↪
    ↪ | range: decided by [test.t2.t1_id], keep order:false                   | N/A
    ↪                   | N/A  |
+--
    ↪ --------------------------+----------+---------+-----------+--------------+-------------------------------
    ↪
6 rows in set (0.61 sec)

+--
    ↪ --------------------------+----------+---------+-----------+--------------+-------------------------------
    ↪
| id                          | estRows  | actRows | task      | access object | execution info
    ↪
    ↪ | operator info                                  | memory                | disk    |
+--
    ↪ --------------------------+----------+---------+-----------+--------------+-------------------------------
    ↪
| HashJoin_19                 | 90000.00 | 20000   | root      |               | time:406.098528
    ↪ ms, loops:22, build_hash_table:{total:148.574644ms, fetch:146.843636ms, build:1.731008ms
    ↪ }, probe:{concurrency:5, total:2.026547436s, max:406.039309ms, probe:205.337813ms, fetch
    ↪ :1.821209623s}                                          | inner join, equal:[eq(test.t1.
    ↪ id, test.t2.t1_id)] | 30.00731658935547 MB  | 0 Bytes |
| ├─TableReader_22(Build)     | 71.01    | 10000   | root      |               | time
    ↪ :147.072725ms, loops:12, cop_task: {num: 3, max: 145.847743ms, min: 50.932527ms, avg:
    ↪ 113.009029ms, p95: 145.847743ms, max_proc_keys: 31724, p95_proc_keys: 31724, tot_proc:
    ↪ 284ms, rpc_num: 3, rpc_time: 338.950488ms, copr_cache_hit_ratio: 0.00} | data:
    ↪ Selection_21                                | 29.679713249206543 MB | N/A     |
| │ └─Selection_21            | 71.01    | 10000   | cop[tikv] |               | time:0ns,
    ↪ loops:0, tikv_task:{proc max:132ms, min:48ms, p80:132ms, p95:132ms, iters:83, tasks:3}
    ↪
    ↪ | eq(test.t1.int_col, 1)                          | N/A                   | N/A     |
| │   └─TableFullScan_20      | 71010.00 | 71010   | cop[tikv] | table:t1      | time:0ns,
    ↪ loops:0, tikv_task:{proc max:128ms, min:48ms, p80:128ms, p95:128ms, iters:83, tasks:3}
    ↪
    ↪ | keep order:false                                | N/A                   | N/A     |
| └─TableReader_24(Probe)     | 90000.00 | 90000   | root      |               | time
```

```
    ↪ :365.918504ms, loops:91, cop_task: {num: 3, max: 398.62145ms, min: 338.460345ms, avg:
    ↪ 358.732721ms, p95: 398.62145ms, max_proc_keys: 31715, p95_proc_keys: 31715, tot_proc: 536
    ↪ ms, rpc_num: 3, rpc_time: 1.076128895s, copr_cache_hit_ratio: 0.00}  | data:
    ↪ TableFullScan_23                          | 182.62489891052246 MB | N/A     |
|   └─TableFullScan_23         | 90000.00 | 90000   | cop[tikv] | table:t2      | time:0ns,
    ↪ loops:0, tikv_task:{proc max:100ms, min:40ms, p80:100ms, p95:100ms, iters:102, tasks:3}
    ↪
    ↪ | keep order:false                                | N/A                   | N/A     |
+--
    ↪ ---------------------------+----------+---------+-----------+--------------+-----------------------------
    ↪
6 rows in set (0.41 sec)

+--
    ↪ ---------------------------+----------+---------+-----------+--------------+-----------------------------
    ↪
| id                          | estRows  | actRows | task      | access object | execution info
    ↪
    ↪ | operator info                                   | memory                | disk    |
+--
    ↪ ---------------------------+----------+---------+-----------+--------------+-----------------------------
    ↪
| HashJoin_20                 | 90000.00 | 20000   | root      |               | time:441.897092
    ↪ ms, loops:21, build_hash_table:{total:138.600864ms, fetch:136.353899ms, build:2.246965ms
    ↪ }, probe:{concurrency:5, total:2.207403854s, max:441.850032ms, probe:148.01937ms, fetch
    ↪ :2.059384484s}                                          | inner join, equal:[eq(test.
    ↪ t1.id, test.t2.t1_id)] | 30.00731658935547 MB  | 0 Bytes |
| ├─TableReader_25(Build)     | 71.01    | 10000   | root      |               | time
    ↪ :138.081807ms, loops:12, cop_task: {num: 3, max: 134.702901ms, min: 53.356202ms, avg:
    ↪ 93.372186ms, p95: 134.702901ms, max_proc_keys: 31724, p95_proc_keys: 31724, tot_proc: 236
    ↪ ms, rpc_num: 3, rpc_time: 280.017658ms, copr_cache_hit_ratio: 0.00}   | data:Selection_24
    ↪                                  | 29.680171966552734 MB | N/A     |
| | └─Selection_24            | 71.01    | 10000   | cop[tikv] |               | time:0ns,
    ↪ loops:0, tikv_task:{proc max:80ms, min:52ms, p80:80ms, p95:80ms, iters:83, tasks:3}
    ↪
    ↪ | eq(test.t1.int_col, 1)                          | N/A                   | N/A     |
| |   └─TableFullScan_23       | 71010.00 | 71010   | cop[tikv] | table:t1      | time:0ns,
    ↪ loops:0, tikv_task:{proc max:80ms, min:52ms, p80:80ms, p95:80ms, iters:83, tasks:3}
    ↪
    ↪ | keep order:false                                | N/A                   | N/A     |
| └─TableReader_22(Probe)     | 90000.00 | 90000   | root      |               | time
    ↪ :413.560548ms, loops:91, cop_task: {num: 3, max: 432.938474ms, min: 231.263355ms, avg:
    ↪ 365.710741ms, p95: 432.938474ms, max_proc_keys: 31715, p95_proc_keys: 31715, tot_proc:
    ↪ 488ms, rpc_num: 3, rpc_time: 1.097021983s, copr_cache_hit_ratio: 0.00} | data:
    ↪ TableFullScan_21                          | 182.62489891052246 MB | N/A     |
```

```
|   └─TableFullScan_21          | 90000.00 | 90000    | cop[tikv] | table:t2      | time:0ns,
    ↪ loops:0, tikv_task:{proc max:80ms, min:80ms, p80:80ms, p95:80ms, iters:102, tasks:3}
    ↪
    ↪ | keep order:false                                  | N/A                   | N/A      |
+--
    ↪ ---------------------------+----------+---------+-----------+--------------+-------------------------
    ↪
6 rows in set (0.44 sec)
```

在上面所示的 Index Join 操作中，`t1.int_col` 一列的索引被删除了。如果加上这个索引，操作执行速度可以从 0.61 秒 提高到 0.14 秒，如下表所示：

```
-- 重新添加索引
ALTER TABLE t2 ADD INDEX (t1_id);

EXPLAIN ANALYZE SELECT /*+ INL_JOIN(t1, t2) */  * FROM t1 INNER JOIN t2 ON t1.id = t2.t1_id WHERE
    ↪  t1.int_col = 1;
EXPLAIN ANALYZE SELECT /*+ HASH_JOIN(t1, t2) */  * FROM t1 INNER JOIN t2 ON t1.id = t2.t1_id
    ↪ WHERE t1.int_col = 1;
EXPLAIN ANALYZE SELECT * FROM t1 INNER JOIN t2 ON t1.id = t2.t1_id WHERE t1.int_col = 1;
```

```
Query OK, 0 rows affected (3.65 sec)

+--
    ↪ -------------------------------+----------+---------+-----------+---------------------------+---------
    ↪
| id                              | estRows  | actRows | task      | access object
    ↪                | execution info
    ↪
    ↪ | operator info                                                                 |
    ↪ memory               | disk |
+--
    ↪ -------------------------------+----------+---------+-----------+---------------------------+---------
    ↪
| IndexJoin_11                    | 90000.00 | 0       | root      |
    ↪                                | time:136.876686ms, loops:1, inner:{total:114.948158ms,
    ↪ concurrency:5, task:7, construct:5.329114ms, fetch:109.610054ms, build:2.38µs}, probe
    ↪ :1.699799ms
    ↪
    ↪ | inner join, inner:IndexLookUp_10, outer key:test.t1.id, inner key:test.t2.t1_id |
    ↪ 29.864535331726074 MB | N/A  |
| ├─TableReader_32(Build)         | 10000.00 | 10000   | root      |
    ↪                                | time:95.755212ms, loops:12, cop_task: {num: 3, max:
    ↪ 95.652443ms, min: 30.758712ms, avg: 57.545129ms, p95: 95.652443ms, max_proc_keys: 31724,
    ↪ p95_proc_keys: 31724, tot_proc: 124ms, rpc_num: 3, rpc_time: 172.528417ms,
    ↪ copr_cache_hit_ratio: 0.00}                     | data:Selection_31
```

```
    ↪                                                                  | 29.679298400878906 MB | N
    ↪ /A  |
| | └─Selection_31                | 10000.00 | 10000    | cop[tikv] |
    ↪                               | time:0ns, loops:0, tikv_task:{proc max:44ms, min:28ms, p80
    ↪ :44ms, p95:44ms, iters:84, tasks:3}
    ↪
    ↪ | eq(test.t1.int_col, 1)                                                                  | N/A
    ↪                     | N/A  |
| |   └─TableFullScan_30          | 71010.00 | 71010    | cop[tikv] | table:t1
    ↪                       | time:0ns, loops:0, tikv_task:{proc max:44ms, min:28ms, p80:44ms,
    ↪ p95:44ms, iters:84, tasks:3}
    ↪
    ↪ | keep order:false                                                                        | N/A
    ↪                     | N/A  |
| └─IndexLookUp_10(Probe)         | 9.00     | 0        | root      |
    ↪                               | time:103.93801ms, loops:7
    ↪
    ↪ |                                                                                        |
    ↪ 2.1787109375 KB       | N/A  |
|   ├─IndexRangeScan_8(Build)     | 9.00     | 0        | cop[tikv] | table:t2, index:t1_id(t1_id
    ↪ ) | time:0s, loops:0, cop_task: {num: 7, max: 23.969244ms, min: 12.003682ms, avg:
    ↪ 14.659066ms, p95: 23.969244ms, tot_proc: 100ms, rpc_num: 7, rpc_time: 102.435966ms,
    ↪ copr_cache_hit_ratio: 0.00}, tikv_task:{proc max:24ms, min:12ms, p80:16ms, p95:24ms,
    ↪ iters:7, tasks:7} | range: decided by [eq(test.t2.t1_id, test.t1.id)], keep order:false
    ↪             | N/A                   | N/A  |
|   └─TableRowIDScan_9(Probe)     | 9.00     | 0        | cop[tikv] | table:t2
    ↪                       | time:0ns, loops:0
    ↪
    ↪ | keep order:false                                                                        | N/A
    ↪                     | N/A  |
+--
    ↪ -------------------------------+----------+---------+-----------+-----------------------------+---------
    ↪
7 rows in set (0.14 sec)

+--
    ↪ ----------------------------+----------+---------+-----------+---------------+-------------------------
    ↪
| id                          | estRows  | actRows | task      | access object | execution info
    ↪
    ↪ | operator info                                    | memory                | disk    |
+--
    ↪ ----------------------------+----------+---------+-----------+---------------+-------------------------
    ↪
| HashJoin_31                 | 90000.00 | 0       | root      |               | time:402.263795
```

```
    ↪ ms, loops:1, build_hash_table:{total:128.467151ms, fetch:126.871282ms, build:1.595869ms},
    ↪  probe:{concurrency:5, total:2.010969815s, max:402.212295ms, probe:8.924769ms, fetch
    ↪ :2.002045046s}                                          | inner join, equal:[eq(test.
    ↪ t1.id, test.t2.t1_id)] | 29.689788818359375 MB | 0 Bytes |
| ├─TableReader_34(Build)      | 10000.00 | 10000   | root      |               | time
    ↪ :126.765972ms, loops:11, cop_task: {num: 3, max: 126.721293ms, min: 54.375481ms, avg:
    ↪ 84.518849ms, p95: 126.721293ms, max_proc_keys: 31724, p95_proc_keys: 31724, tot_proc: 208
    ↪ ms, rpc_num: 3, rpc_time: 253.478218ms, copr_cache_hit_ratio: 0.00} | data:Selection_33
    ↪                                  | 29.679292678833008 MB | N/A     |
| │ └─Selection_33             | 10000.00 | 10000   | cop[tikv] |               | time:0ns,
    ↪ loops:0, tikv_task:{proc max:72ms, min:56ms, p80:72ms, p95:72ms, iters:84, tasks:3}
    ↪
    ↪ | eq(test.t1.int_col, 1)                              | N/A                   | N/A     |
| │   └─TableFullScan_32       | 71010.00 | 71010   | cop[tikv] | table:t1      | time:0ns,
    ↪ loops:0, tikv_task:{proc max:72ms, min:56ms, p80:72ms, p95:72ms, iters:84, tasks:3}
    ↪
    ↪ | keep order:false                                    | N/A                   | N/A     |
| └─TableReader_36(Probe)      | 90000.00 | 90000   | root      |               | time
    ↪ :400.447175ms, loops:90, cop_task: {num: 3, max: 400.838264ms, min: 309.474053ms, avg:
    ↪ 341.01943ms, p95: 400.838264ms, max_proc_keys: 31719, p95_proc_keys: 31719, tot_proc: 528
    ↪ ms, rpc_num: 3, rpc_time: 1.02298055s, copr_cache_hit_ratio: 0.00} | data:
    ↪ TableFullScan_35                               | 182.62786674499512 MB | N/A     |
|   └─TableFullScan_35         | 90000.00 | 90000   | cop[tikv] | table:t2      | time:0ns,
    ↪ loops:0, tikv_task:{proc max:108ms, min:72ms, p80:108ms, p95:108ms, iters:102, tasks:3}
    ↪
    ↪ | keep order:false                                    | N/A                   | N/A     |
+--
    ↪ ----------------------------+----------+---------+-----------+---------------+----------------------------
    ↪
6 rows in set (0.40 sec)

+--
    ↪ ----------------------------+----------+---------+-----------+---------------+----------------------------
    ↪
| id                           | estRows  | actRows | task      | access object | execution info
    ↪
    ↪ | operator info                                       | memory                | disk    |
+--
    ↪ ----------------------------+----------+---------+-----------+---------------+----------------------------
    ↪
| HashJoin_32                  | 90000.00 | 0       | root      |               | time:356.282882
    ↪ ms, loops:1, build_hash_table:{total:154.187155ms, fetch:151.259305ms, build:2.92785ms},
    ↪ probe:{concurrency:5, total:1.781087041s, max:356.238312ms, probe:7.406146ms, fetch
    ↪ :1.773680895s}                                            | inner join, equal:[eq(test
    ↪ .t1.id, test.t2.t1_id)] | 29.689788818359375 MB | 0 Bytes |
```

```
| ├─TableReader_41(Build)      | 10000.00 | 10000   | root      |               | time
    ↪ :151.190175ms, loops:11, cop_task: {num: 3, max: 151.055697ms, min: 56.214348ms, avg:
    ↪ 96.70463ms, p95: 151.055697ms, max_proc_keys: 31724, p95_proc_keys: 31724, tot_proc: 240
    ↪ ms, rpc_num: 3, rpc_time: 290.019942ms, copr_cache_hit_ratio: 0.00}   | data:Selection_40
    ↪                                  | 29.679292678833008 MB | N/A     |
| | └─Selection_40             | 10000.00 | 10000   | cop[tikv] |               | time:0ns,
    ↪ loops:0, tikv_task:{proc max:80ms, min:56ms, p80:80ms, p95:80ms, iters:84, tasks:3}
    ↪
    ↪ | eq(test.t1.int_col, 1)                          | N/A                  | N/A     |
| |   └─TableFullScan_39       | 71010.00 | 71010   | cop[tikv] | table:t1      | time:0ns,
    ↪ loops:0, tikv_task:{proc max:80ms, min:56ms, p80:80ms, p95:80ms, iters:84, tasks:3}
    ↪
    ↪ | keep order:false                                | N/A                  | N/A     |
| └─TableReader_43(Probe)      | 90000.00 | 90000   | root      |               | time:354.68523
    ↪ ms, loops:90, cop_task: {num: 3, max: 354.313475ms, min: 328.460762ms, avg: 345.530558ms,
    ↪  p95: 354.313475ms, max_proc_keys: 31719, p95_proc_keys: 31719, tot_proc: 508ms, rpc_num:
    ↪  3, rpc_time: 1.036492374s, copr_cache_hit_ratio: 0.00} | data:TableFullScan_42
    ↪                              | 182.62786102294922 MB | N/A     |
|   └─TableFullScan_42         | 90000.00 | 90000   | cop[tikv] | table:t2      | time:0ns,
    ↪ loops:0, tikv_task:{proc max:84ms, min:64ms, p80:84ms, p95:84ms, iters:102, tasks:3}
    ↪
    ↪ | keep order:false                                | N/A                  | N/A     |
+--
    ↪ ---------------------------+----------+---------+-----------+---------------+-----------------------------
    ↪
6 rows in set (0.36 sec)
```

> **注意：**
>
> 在上方示例中，SQL 优化器之所以选择了性能较差的 Hash Join 算法，而不是 Index Join 算法，原因在于查询优化是一个 NP 完全问题，可能会选择不太理想的计划。如果需要频繁调用这个查询，建议通过执行计划管理的方式将 Hint 与 SQL 语句绑定，这样要比在发送给 TiDB 的 SQL 语句中插入 Hint 更容易管理。

Index Join 相关算法

如果使用 Hint `INL_JOIN` 进行 Index Join 操作，TiDB 会在连接外表之前创建一个中间结果的 Hash Table。TiDB 同样也支持使用 Hint `INL_HASH_JOIN` 在外表上建 Hash Table。而如果内表列集合与外表的相匹配，则可以应用 Hint `INL_MERGE_JOIN`。以上所述的 Index Join 相关算法都由 SQL 优化器自动选择。

配置

Index Join 算法的性能受以下系统变量影响：

- `tidb_index_join_batch_size`（默认值：`25000`）- `index lookup join` 操作的 batch 大小。

- `tidb_index_lookup_join_concurrency`（默认值：4）- 可以并发执行的 index lookup 任务数。

9.3.2.4.2 Hash Join

在 Hash Join 操作中，TiDB 首先读取 Build 端的数据并将其缓存在 Hash Table 中，然后再读取 Probe 端的数据，使用 Probe 端的数据来探查 Hash Table 以获得所需行。与 Index Join 算法相比，Hash Join 要消耗更多内存，但如果需要连接的行数很多，运行速度会比 Index Join 快。TiDB 中的 Hash Join 算子是多线程的，并且可以并发执行。

下面是一个 Hash Join 示例：

```
EXPLAIN SELECT /*+ HASH_JOIN(t1, t2) */ * FROM t1, t2 WHERE t1.id = t2.id;
```

```
+--
    ↪ --------------------------+----------+----------+---------------+--------------------------------
    ↪
| id                          | estRows    | task      | access object | operator info
    ↪                                  |
+--
    ↪ --------------------------+----------+----------+---------------+--------------------------------
    ↪
| HashJoin_27                 | 142020.00 | root      |               | inner join, equal:[eq(
    ↪ test.t1.id, test.t2.id)] |
| ├─TableReader_29(Build)     | 142020.00 | root      |               | data:TableFullScan_28
    ↪                           |
| │ └─TableFullScan_28        | 142020.00 | cop[tikv] | table:t1      | keep order:false
    ↪                                |
| └─TableReader_31(Probe)     | 180000.00 | root      |               | data:TableFullScan_30
    ↪                           |
|   └─TableFullScan_30        | 180000.00 | cop[tikv] | table:t2      | keep order:false
    ↪                                |
+--
    ↪ --------------------------+----------+----------+---------------+--------------------------------
    ↪
5 rows in set (0.00 sec)
```

TiDB 会按照以下顺序执行 HashJoin_27 算子：

1. 将 Build 端数据缓存在内存中。
2. 根据缓存数据在 Build 端构造一个 Hash Table。
3. 读取 Probe 端的数据。
4. 使用 Probe 端的数据来探查 Hash Table。
5. 将符合条件的结果返回给用户。

EXPLAIN 返回结果中的 `operator info` 一列记录了 HashJoin_27 的其他信息，包括该查询是 Inner Join 还是 Outer Join 以及 Join 的条件是什么等。在上面给出的示例中，该查询为 Inner Join，Join 条件是 `equal:[eq(test.t1.id, test.t2.id)]`，与查询语句中的 `WHERE t1.id = t2.id` 部分对应。下面例子中其他几个 Join 算子的 operator info 和此处类似。

运行数据

如果在执行操作时，内存使用超过了tidb_mem_quota_query 规定的值（默认为 1GB），且 oom-use-tmp-storage 的值为 true （默认为 true），那么 TiDB 会尝试使用临时存储，在磁盘上创建 Hash Join 的 Build 端。EXPLAIN ANALYZE 返回结果中的 execution info 一栏记录了有关内存使用情况等运行数据。下面的例子展示了 tidb_mem_quota_query 的值分别设为 1GB（默认）及 500MB 时，EXPLAIN ANALYZE 的返回结果（当内存配额设为 500MB 时，磁盘用作临时存储区）：

```
EXPLAIN ANALYZE SELECT /*+ HASH_JOIN(t1, t2) */ * FROM t1, t2 WHERE t1.id = t2.id;
SET tidb_mem_quota_query=500 * 1024 * 1024;
EXPLAIN ANALYZE SELECT /*+ HASH_JOIN(t1, t2) */ * FROM t1, t2 WHERE t1.id = t2.id;
```

```
+--
    ↪ --------------------------+-----------+---------+-----------+---------------+--------------------------------
    ↪
| id                          | estRows   | actRows | task      | access object | execution info
    ↪
    ↪ | operator info                                  | memory                | disk    |
+--
    ↪ --------------------------+-----------+---------+-----------+---------------+--------------------------------
    ↪
| HashJoin_27                 | 142020.00 | 71010   | root      |               | time:647.508572
    ↪ ms, loops:72, build_hash_table:{total:579.254415ms, fetch:566.91012ms, build:12.344295ms
    ↪ }, probe:{concurrency:5, total:3.23315006s, max:647.520113ms, probe:330.884716ms, fetch
    ↪ :2.902265344s}                                          | inner join, equal:[eq(test.
    ↪ t1.id, test.t2.id)] | 209.61642456054688 MB | 0 Bytes |
| ├─TableReader_29(Build)     | 142020.00 | 71010   | root      |               | time
    ↪ :567.088247ms, loops:72, cop_task: {num: 2, max: 569.809411ms, min: 369.67451ms, avg:
    ↪ 469.74196ms, p95: 569.809411ms, max_proc_keys: 39245, p95_proc_keys: 39245, tot_proc: 400
    ↪ ms, rpc_num: 2, rpc_time: 939.447231ms, copr_cache_hit_ratio: 0.00}   | data:
    ↪ TableFullScan_28                          | 210.2100534439087 MB  | N/A     |
| │ └─TableFullScan_28        | 142020.00 | 71010   | cop[tikv] | table:t1      | proc max:64ms,
    ↪  min:48ms, p80:64ms, p95:64ms, iters:79, tasks:2
    ↪
    ↪ | keep order:false                                | N/A                   | N/A     |
| └─TableReader_31(Probe)     | 180000.00 | 90000   | root      |               | time
    ↪ :337.233636ms, loops:91, cop_task: {num: 3, max: 569.790741ms, min: 332.758911ms, avg:
    ↪ 421.543165ms, p95: 569.790741ms, max_proc_keys: 31719, p95_proc_keys: 31719, tot_proc:
    ↪ 500ms, rpc_num: 3, rpc_time: 1.264570696s, copr_cache_hit_ratio: 0.00} | data:
    ↪ TableFullScan_30                          | 267.1126985549927 MB  | N/A     |
|   └─TableFullScan_30        | 180000.00 | 90000   | cop[tikv] | table:t2      | proc max:84ms,
    ↪  min:72ms, p80:84ms, p95:84ms, iters:102, tasks:3
    ↪
    ↪ | keep order:false                                | N/A                   | N/A     |
+--
    ↪ --------------------------+-----------+---------+-----------+---------------+--------------------------------
```

```
    ↪
5 rows in set (0.65 sec)

Query OK, 0 rows affected (0.00 sec)

+--
    ↪ -------------------------+----------+---------+----------+--------------+-------------------------
    ↪
| id                         | estRows    | actRows | task      | access object | execution info
    ↪
    ↪ | operator info                                | memory                | disk
    ↪                 |
+--
    ↪ -------------------------+----------+---------+----------+--------------+-------------------------
    ↪
| HashJoin_27                | 142020.00 | 71010   | root      |               | time:963.983353
    ↪ ms, loops:72, build_hash_table:{total:775.961447ms, fetch:503.789677ms, build:272.17177ms
    ↪ }, probe:{concurrency:5, total:4.805454793s, max:963.973133ms, probe:922.156835ms, fetch
    ↪ :3.883297958s}                                          | inner join, equal:[eq(test.t1.
    ↪ id, test.t2.id)] | 93.53974533081055 MB  | 210.7459259033203 MB |
| ├─TableReader_29(Build)    | 142020.00 | 71010   | root      |               | time
    ↪ :504.062018ms, loops:72, cop_task: {num: 2, max: 509.276857ms, min: 402.66386ms, avg:
    ↪ 455.970358ms, p95: 509.276857ms, max_proc_keys: 39245, p95_proc_keys: 39245, tot_proc:
    ↪ 384ms, rpc_num: 2, rpc_time: 911.893237ms, copr_cache_hit_ratio: 0.00}  | data:
    ↪ TableFullScan_28                        | 210.20934200286865 MB | N/A
    ↪ |
| │ └─TableFullScan_28        | 142020.00 | 71010   | cop[tikv] | table:t1      | proc max:88ms,
    ↪  min:72ms, p80:88ms, p95:88ms, iters:79, tasks:2
    ↪
    ↪ | keep order:false                              | N/A                   | N/A
    ↪                   |
| └─TableReader_31(Probe)    | 180000.00 | 90000   | root      |               | time
    ↪ :363.058382ms, loops:91, cop_task: {num: 3, max: 412.659191ms, min: 358.489688ms, avg:
    ↪ 391.463008ms, p95: 412.659191ms, max_proc_keys: 31719, p95_proc_keys: 31719, tot_proc:
    ↪ 484ms, rpc_num: 3, rpc_time: 1.174326746s, copr_cache_hit_ratio: 0.00} | data:
    ↪ TableFullScan_30                        | 267.11340618133545 MB | N/A
    ↪ |
|   └─TableFullScan_30        | 180000.00 | 90000   | cop[tikv] | table:t2      | proc max:92ms,
    ↪  min:64ms, p80:92ms, p95:92ms, iters:102, tasks:3
    ↪
    ↪ | keep order:false                              | N/A                   | N/A
    ↪                   |
+--
    ↪ -------------------------+----------+---------+----------+--------------+-------------------------
    ↪
```

```
5 rows in set (0.98 sec)
```

配置

Hash Join 算法的性能受以下系统变量影响：

- tidb_mem_quota_query（默认值：1GB）- 如果某条查询的内存消耗超出了配额，TiDB 会尝试将 Hash Join 的 Build 端移到磁盘上以节省内存。
- tidb_hash_join_concurrency（默认值：5）- 可以并发执行的 Hash Join 任务数量。

9.3.2.4.3 Merge Join

Merge Join 是一种特殊的 Join 算法。当两个关联表要 Join 的字段需要按排好的顺序读取时，适用 Merge Join 算法。由于 Build 端和 Probe 端的数据都会读取，这种算法的 Join 操作是流式的，类似“拉链式合并”的高效版。Merge Join 占用的内存要远低于 Hash Join，但 Merge Join 不能并发执行。

下面是一个使用 Merge Join 的例子：

```
EXPLAIN SELECT /*+ MERGE_JOIN(t1, t2) */ * FROM t1, t2 WHERE t1.id = t2.id;
```

```
+--
    ↪ --------------------------+-----------+-----------+---------------+--------------------------------
    ↪
| id                          | estRows   | task      | access object | operator info
    ↪                                       |
+--
    ↪ --------------------------+-----------+-----------+---------------+--------------------------------
    ↪
| MergeJoin_7                 | 142020.00 | root      |               | inner join, left key:test
    ↪ .t1.id, right key:test.t2.id |
| ├─TableReader_12(Build)     | 180000.00 | root      |               | data:TableFullScan_11
    ↪                                     |
| │ └─TableFullScan_11        | 180000.00 | cop[tikv] | table:t2      | keep order:true
    ↪                                       |
| └─TableReader_10(Probe)     | 142020.00 | root      |               | data:TableFullScan_9
    ↪                                      |
|   └─TableFullScan_9         | 142020.00 | cop[tikv] | table:t1      | keep order:true
    ↪                                       |
+--
    ↪ --------------------------+-----------+-----------+---------------+--------------------------------
    ↪
5 rows in set (0.00 sec)
```

TiDB 会按照以下顺序执行 Merge Join 算子：

1. 从 Build 端把一个 Join Group 的数据全部读取到内存中。
2. 读取 Probe 端的数据。

3. 将 Probe 端的每行数据与 Build 端的一个完整 Join Group 比较，依次查看是否匹配（除了满足等值条件以外，还有其他非等值条件，这里的“匹配”主要是指查看是否满足非等值条件）。Join Group 指的是所有 Join Key 上值相同的数据。

9.3.2.4.4 其他类型查询的执行计划

- 索引查询的执行计划
- 子查询的执行计划
- 聚合查询的执行计划
- 视图查询的执行计划
- 分区查询的执行计划
- 开启 IndexMerge 查询的执行计划

9.3.2.5 用 EXPLAIN 查看子查询的执行计划

TiDB 会执行多种子查询相关的优化，以提升子查询的执行性能。本文档介绍一些常见子查询的优化方式，以及如何解读 EXPLAIN 语句返回的执行计划信息。

本文档所使用的示例表数据如下:

```
CREATE TABLE t1 (id BIGINT NOT NULL PRIMARY KEY auto_increment, pad1 BLOB, pad2 BLOB, pad3 BLOB,
    ↪ int_col INT NOT NULL DEFAULT 0);
CREATE TABLE t2 (id BIGINT NOT NULL PRIMARY KEY auto_increment, t1_id BIGINT NOT NULL, pad1 BLOB,
    ↪  pad2 BLOB, pad3 BLOB, INDEX(t1_id));
CREATE TABLE t3 (
 id INT NOT NULL PRIMARY KEY auto_increment,
 t1_id INT NOT NULL,
 UNIQUE (t1_id)
);

INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM
    ↪ dual;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
```

```
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024), 0 FROM t1
    ↪  a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t2 SELECT NULL, a.id, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪  t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
UPDATE t1 SET int_col = 1 WHERE pad1 = (SELECT pad1 FROM t1 ORDER BY RAND() LIMIT 1);
INSERT INTO t3 SELECT NULL, id FROM t1 WHERE id < 1000;

SELECT SLEEP(1);
ANALYZE TABLE t1, t2, t3;
```

9.3.2.5.1 Inner join（无`UNIQUE`约束的子查询）

以下示例中，`IN`子查询会从表`t2`中搜索一列ID。为保证语义正确性，TiDB需要保证`t1_id`列的值具有唯一性。使用`EXPLAIN`可查看到该查询的执行计划去掉重复项并执行`Inner Join`内连接操作：

```
EXPLAIN SELECT * FROM t1 WHERE id IN (SELECT t1_id FROM t2);
```

```
+--
    ↪ ------------------------------+----------+-----------+------------------------------+--------------------
    ↪
| id                             | estRows  | task      | access object                | operator
    ↪  info                                                     |
+--
    ↪ ------------------------------+----------+-----------+------------------------------+--------------------
    ↪
```

```
| IndexMergeJoin_19               | 45.00    | root      |                                | inner
    ↪ join, inner:TableReader_14, outer key:test.t2.t1_id, inner key:test.t1.id |
| ├─HashAgg_38(Build)             | 45.00    | root      |                                | group
    ↪ by:test.t2.t1_id, funcs:firstrow(test.t2.t1_id)->test.t2.t1_id            |
| │ └─IndexReader_39              | 45.00    | root      |                                | index:
    ↪ HashAgg_31                                                                |
| │   └─HashAgg_31                | 45.00    | cop[tikv] |                                | group
    ↪ by:test.t2.t1_id,                                                         |
| │     └─IndexFullScan_37        | 90000.00 | cop[tikv] | table:t2, index:t1_id(t1_id) | keep
    ↪ order:false                                                                |
| └─TableReader_14(Probe)         | 1.00     | root      |                                | data:
    ↪ TableRangeScan_13                                                          |
|   └─TableRangeScan_13           | 1.00     | cop[tikv] | table:t1                       | range:
    ↪ decided by [test.t2.t1_id], keep order:true                               |
+--
    ↪ -----------------------------+----------+-----------+------------------------------+------------------
    ↪
7 rows in set (0.00 sec)
```

由上述查询结果可知，TiDB 首先执行 Index Join 索引连接（即 Merge Join 合并连接的变体）操作，开始读取 t2.t1_id 列的索引。先是 └─HashAgg_31 算子的部分任务在 TiKV 中对 t1_id 值进行去重，然后├─HashAgg_38(
↪ Build) 算子的部分任务在 TiDB 中对 t1_id 值再次进行去重。去重操作由聚合函数 firstrow(test.t2.t1_id) 执行，之后会将操作结果与 t1 表的主键相连接。

9.3.2.5.2 Inner join（有 UNIQUE 约束的子查询）

在上述示例中，为了确保 t1_id 值在与表 t1 连接前具有唯一性，需要执行聚合运算。在以下示例中，由于 UNIQUE 约束已能确保 t3.t1_id 列值的唯一：

```
EXPLAIN SELECT * FROM t1 WHERE id IN (SELECT t1_id FROM t3);
```

```
+--
    ↪ --------------------------+---------+-----------+----------------------------+-----------------------
    ↪
| id                          | estRows | task      | access object              | operator
    ↪ info                                                                        |
+--
    ↪ --------------------------+---------+-----------+----------------------------+-----------------------
    ↪
| IndexMergeJoin_20           | 999.00  | root      |                            | inner join,
    ↪ inner:TableReader_15, outer key:test.t3.t1_id, inner key:test.t1.id |
| ├─IndexReader_39(Build)     | 999.00  | root      |                            | index:
    ↪ IndexFullScan_38                                                            |
| │ └─IndexFullScan_38        | 999.00  | cop[tikv] | table:t3, index:t1_id(t1_id) | keep order:
    ↪ false                                                                       |
```

```
| └─TableReader_15(Probe)     | 1.00    | root      |                               | data:
    ↪ TableRangeScan_14                                                                    |
|   └─TableRangeScan_14       | 1.00    | cop[tikv] | table:t1                      | range:
    ↪ decided by [test.t3.t1_id], keep order:true                                        |
+--
    ↪ --------------------------+---------+-----------+-------------------------------+----------------------
    ↪
5 rows in set (0.00 sec)
```

从语义上看，因为约束保证了 t3.t1_id 列值的唯一性，TiDB 可以直接执行 INNER JOIN 查询。

9.3.2.5.3 Semi Join（关联查询）

在前两个示例中，通过 HashAgg 聚合操作或通过 UNIQUE 约束保证子查询数据的唯一性之后，TiDB 才能够执行 Inner Join 操作。这两种连接均使用了 Index Join（Merge Join 的变体）。

下面的例子中，TiDB 优化器则选择了一种不同的执行计划：

```
EXPLAIN SELECT * FROM t1 WHERE id IN (SELECT t1_id FROM t2 WHERE t1_id != t1.int_col);
```

```
+--
    ↪ --------------------------+----------+----------+-------------------------------+----------------------
    ↪
| id                          | estRows   | task      | access object                 | operator
    ↪ info
    ↪
    ↪ |
+--
    ↪ --------------------------+----------+----------+-------------------------------+----------------------
    ↪
| MergeJoin_9                 | 45446.40  | root      |                               | semi join,
    ↪  left key:test.t1.id, right key:test.t2.t1_id, other cond:ne(test.t2.t1_id, test.t1.
    ↪ int_col) |
| ├─IndexReader_24(Build)     | 180000.00 | root      |                               | index:
    ↪ IndexFullScan_23
    ↪                                                                                    |
| │ └─IndexFullScan_23        | 180000.00 | cop[tikv] | table:t2, index:t1_id(t1_id) | keep
    ↪ order:true
    ↪                                                                                          |
| └─TableReader_22(Probe)     | 56808.00  | root      |                               | data:
    ↪ Selection_21
    ↪                                                                                              |
|   └─Selection_21            | 56808.00  | cop[tikv] |                               | ne(test.
    ↪ t1.id, test.t1.int_col)
    ↪                                                                                  |
```

```
|     └─TableFullScan_20      | 71010.00  | cop[tikv] | table:t1                    | keep
    ↪ order:true
    ↪                                                                                 |
+--
    ↪ -------------------------+----------+----------+-----------------------------+-------------------
    ↪
6 rows in set (0.00 sec)
```

由上述查询结果可知，TiDB 执行了 Semi Join。不同于 Inner Join，Semi Join 仅允许右键 (t2.t1_id) 上的第一个值，也就是该操作将去除 Join 算子任务中的重复数据。Join 算法也包含 Merge Join，会按照排序顺序同时从左侧和右侧读取数据，这是一种高效的 Zipper Merge。

可以将原语句视为关联子查询，因为它引入了子查询外的 t1.int_col 列。然而，EXPLAIN 语句的返回结果显示的是关联子查询去关联后的执行计划。条件 t1_id != t1.int_col 会被重写为 t1.id != t1.int_col。TiDB 可以从表 t1 中读取数据并且在 └─Selection_21 中执行此操作，因此这种去关联和重写操作会极大提高执行效率。

9.3.2.5.4 Anti Semi Join （NOT IN 子查询）

在以下示例中，除非子查询中存在 t3.t1_id，否则该查询将（从语义上）返回表 t3 中的所有行：

```
EXPLAIN SELECT * FROM t3 WHERE t1_id NOT IN (SELECT id FROM t1 WHERE int_col < 100);
```

```
+--
    ↪ -------------------------+---------+----------+--------------+-----------------------------------
    ↪
| id                          | estRows | task     | access object | operator info
    ↪                                                                          |
+--
    ↪ -------------------------+---------+----------+--------------+-----------------------------------
    ↪
| IndexMergeJoin_20           | 1598.40 | root     |              | anti semi join, inner:
    ↪ TableReader_15, outer key:test.t3.t1_id, inner key:test.t1.id |
| ├─TableReader_28(Build)     | 1998.00 | root     |              | data:TableFullScan_27
    ↪                                                             |
| │ └─TableFullScan_27        | 1998.00 | cop[tikv] | table:t3     | keep order:false
    ↪                                                                  |
| └─TableReader_15(Probe)     | 1.00    | root     |              | data:Selection_14
    ↪                                                                      |
|   └─Selection_14            | 1.00    | cop[tikv] |              | lt(test.t1.int_col, 100)
    ↪                                                            |
|     └─TableRangeScan_13     | 1.00    | cop[tikv] | table:t1     | range: decided by [test.t3
    ↪ .t1_id], keep order:true                                  |
+--
    ↪ -------------------------+---------+----------+--------------+-----------------------------------
    ↪
```

```
6 rows in set (0.00 sec)
```

上述查询首先读取了表 `t3`，然后根据主键开始探测 (probe) 表 `t1`。连接类型是 anti semi join，即反半连接：之所以使用 anti，是因为上述示例有不存在匹配值（即 `NOT IN`）的情况；使用 `Semi Join` 则是因为仅需要匹配第一行后就可以停止查询。

9.3.2.5.5 其他类型查询的执行计划

- 索引查询的执行计划
- Join 查询的执行计划
- 聚合查询的执行计划
- 视图查询的执行计划
- 分区查询的执行计划
- 开启 IndexMerge 查询的执行计划

9.3.2.6 用 EXPLAIN 查看聚合查询执行计划

SQL 查询中可能会使用聚合计算，可以通过 `EXPLAIN` 语句来查看聚合查询的执行计划。本文提供多个示例，以帮助用户理解聚合查询是如何执行的。

SQL 优化器会选择以下任一算子实现数据聚合：

- Hash Aggregation
- Stream Aggregation

为了提高查询效率，数据聚合在 Coprocessor 层和 TiDB 层均会执行。现有示例如下：

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY auto_increment, pad1 BLOB, pad2 BLOB, pad3 BLOB);
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM dual;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    ↪ JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    ↪ JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    ↪ JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    ↪ JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    ↪ JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    ↪ JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    ↪ JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    ↪ JOIN t1 b JOIN t1 c LIMIT 10000;
```

```
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    ↪ JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    ↪ JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    ↪ JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    ↪ JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    ↪ JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    ↪ JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM t1 a
    ↪ JOIN t1 b JOIN t1 c LIMIT 10000;
SELECT SLEEP(1);
ANALYZE TABLE t1;
```

以上示例创建表格 `t1` 并插入数据后，再执行`SHOW TABLE REGIONS` 语句。从以下 `SHOW TABLE REGIONS` 的执行结果可知，表 `t1` 被切分为多个 Region：

```
SHOW TABLE t1 REGIONS;
```

```
+--
    ↪ ---------+--------------+--------------+-----------+-----------------+-------+------------+--------------
    ↪
| REGION_ID | START_KEY    | END_KEY      | LEADER_ID | LEADER_STORE_ID | PEERS | SCATTERING |
    ↪ WRITTEN_BYTES | READ_BYTES | APPROXIMATE_SIZE(MB) | APPROXIMATE_KEYS |
+--
    ↪ ---------+--------------+--------------+-----------+-----------------+-------+------------+--------------
    ↪
|        64 | t_64_        | t_64_r_31766 |        65 |               1 | 65    |          0 |
    ↪          1325 |  102033520 |                   98 |            52797 |
|        66 | t_64_r_31766 | t_64_r_63531 |        67 |               1 | 67    |          0 |
    ↪          1325 |   72522521 |                  104 |            78495 |
|        68 | t_64_r_63531 | t_64_r_95296 |        69 |               1 | 69    |          0 |
    ↪          1325 |          0 |                  104 |            95433 |
|         2 | t_64_r_95296 |              |         3 |               1 | 3     |          0 |
    ↪          1501 |          0 |                   81 |            63211 |
+--
    ↪ ---------+--------------+--------------+-----------+-----------------+-------+------------+--------------
    ↪
4 rows in set (0.00 sec)
```

使用 EXPLAIN 查看以下聚合语句的执行计划。可以看到 `└─StreamAgg_8` 算子先执行在 TiKV 内每个 Region 上，然后 TiKV 的每个 Region 会返回一行数据给 TiDB，TiDB 在 `StreamAgg_16` 算子上对每个 Region 返回的数据进行聚合：

```
EXPLAIN SELECT COUNT(*) FROM t1;
```

```
+--
    ↪ -------------------------+----------+----------+--------------+-------------------------------+
    ↪
| id                          | estRows   | task      | access object | operator info
    ↪                   |
+--
    ↪ -------------------------+----------+----------+--------------+-------------------------------+
    ↪
| StreamAgg_16                | 1.00      | root      |               | funcs:count(Column#7)->
    ↪ Column#5 |
| └─TableReader_17            | 1.00      | root      |               | data:StreamAgg_8
    ↪                 |
|   └─StreamAgg_8             | 1.00      | cop[tikv] |               | funcs:count(1)->Column#7
    ↪        |
|     └─TableFullScan_15      | 242020.00 | cop[tikv] | table:t1      | keep order:false
    ↪                 |
+--
    ↪ -------------------------+----------+----------+--------------+-------------------------------+
    ↪
4 rows in set (0.00 sec)
```

同样，通过执行 EXPLAIN ANALYZE 语句可知，actRows 与 SHOW TABLE REGIONS 返回结果中的 Region 数匹配，这是因为执行使用了 TableFullScan 全表扫并且没有二级索引：

```
EXPLAIN ANALYZE SELECT COUNT(*) FROM t1;
```

```
+--
    ↪ -------------------------+----------+---------+----------+--------------+-------------------------
    ↪
| id                          | estRows   | actRows | task      | access object | execution info
    ↪
    ↪ | operator info                   | memory    | disk |
+--
    ↪ -------------------------+----------+---------+----------+--------------+-------------------------
    ↪
| StreamAgg_16                | 1.00      | 1       | root      |               | time:12.609575ms
    ↪ , loops:2
    ↪
    ↪ | funcs:count(Column#7)->Column#5 | 372 Bytes | N/A  |
| └─TableReader_17            | 1.00      | 4       | root      |               | time:12.605155
    ↪ ms, loops:2, cop_task: {num: 4, max: 12.538245ms, min: 9.256838ms, avg: 10.895114ms, p95:
    ↪  12.538245ms, max_proc_keys: 31765, p95_proc_keys: 31765, tot_proc: 48ms, rpc_num: 4,
    ↪ rpc_time: 43.530707ms, copr_cache_hit_ratio: 0.00} | data:StreamAgg_8                |
```

```
    ↪ 293 Bytes | N/A  |
|   └─StreamAgg_8            | 1.00      | 4       | cop[tikv] |               | proc max:12ms,
    ↪ min:12ms, p80:12ms, p95:12ms, iters:122, tasks:4
    ↪
    ↪ | funcs:count(1)->Column#7        | N/A       | N/A  |
|     └─TableFullScan_15     | 242020.00 | 121010  | cop[tikv] | table:t1      | proc max:12ms,
    ↪ min:12ms, p80:12ms, p95:12ms, iters:122, tasks:4
    ↪
    ↪ | keep order:false                | N/A       | N/A  |
+--
    ↪ -------------------------+-----------+---------+-----------+---------------+----------------------------
    ↪
4 rows in set (0.01 sec)
```

9.3.2.6.1 Hash Aggregation

Hash Aggregation 算法在执行聚合时使用 Hash 表存储中间结果。此算法采用多线程并发优化，执行速度快，但与 Stream Aggregation 算法相比会消耗较多内存。

下面是一个使用 Hash Aggregation（即 HashAgg 算子）的例子：

```
EXPLAIN SELECT /*+ HASH_AGG() */ count(*) FROM t1;
```

```
+--
    ↪ -------------------------+-----------+-----------+---------------+--------------------------------+
    ↪
| id                         | estRows   | task      | access object | operator info
    ↪                   |
+--
    ↪ -------------------------+-----------+-----------+---------------+--------------------------------+
    ↪
| HashAgg_9                  | 1.00      | root      |               | funcs:count(Column#6)->
    ↪ Column#5 |
| └─TableReader_10           | 1.00      | root      |               | data:HashAgg_5
    ↪                   |
|   └─HashAgg_5              | 1.00      | cop[tikv] |               | funcs:count(1)->Column#6
    ↪        |
|     └─TableFullScan_8      | 242020.00 | cop[tikv] | table:t1      | keep order:false
    ↪                 |
+--
    ↪ -------------------------+-----------+-----------+---------------+--------------------------------+
    ↪
4 rows in set (0.00 sec)
```

`operator info` 列显示，用于聚合数据的 Hash 函数为 `funcs:count(1)->Column#6`。

9.3.2.6.2 Stream Aggregation

Stream Aggregation 算法通常会比 Hash Aggregation 算法占用更少的内存。但是此算法要求数据按顺序发送，以便对依次到达的值实现流式数据聚合。

下面是一个使用 Stream Aggregation 的例子：

```
CREATE TABLE t2 (id INT NOT NULL PRIMARY KEY, col1 INT NOT NULL);
INSERT INTO t2 VALUES (1, 9),(2, 3),(3,1),(4,8),(6,3);
EXPLAIN SELECT /*+ STREAM_AGG() */ col1, count(*) FROM t2 GROUP BY col1;
```

```
Query OK, 0 rows affected (0.11 sec)

Query OK, 5 rows affected (0.01 sec)
Records: 5  Duplicates: 0  Warnings: 0

+--
    ↪ ----------------------------+----------+-----------+---------------+--------------------------------
    ↪
| id                           | estRows  | task      | access object | operator info
    ↪                                                                                                  |
+--
    ↪ ----------------------------+----------+-----------+---------------+--------------------------------
    ↪
| Projection_4                 | 8000.00  | root      |               | test.t2.col1, Column#3
    ↪                                                                                                  |
| └─StreamAgg_8                | 8000.00  | root      |               | group by:test.t2.col1,
    ↪ funcs:count(1)->Column#3, funcs:firstrow(test.t2.col1)->test.t2.col1 |
|   └─Sort_13                  | 10000.00 | root      |               | test.t2.col1
    ↪                                                                                                  |
|     └─TableReader_12         | 10000.00 | root      |               | data:TableFullScan_11
    ↪                                                                                                  |
|       └─TableFullScan_11     | 10000.00 | cop[tikv] | table:t2      | keep order:false, stats:
    ↪ pseudo                                                                                           |
+--
    ↪ ----------------------------+----------+-----------+---------------+--------------------------------
    ↪
5 rows in set (0.00 sec)
```

以上示例中，可以在 col1 上添加索引来消除 └─Sort_13 算子。添加索引后，TiDB 就可以按顺序读取数据并消除 └─Sort_13 算子。

```
ALTER TABLE t2 ADD INDEX (col1);
EXPLAIN SELECT /*+ STREAM_AGG() */ col1, count(*) FROM t2 GROUP BY col1;
```

```
Query OK, 0 rows affected (0.28 sec)
```

```
+--
    ↪ ---------------------------+---------+-----------+---------------------------+-----------------------------
    ↪
| id                           | estRows | task      | access object             | operator info
    ↪                                                                                                            |
+--
    ↪ ---------------------------+---------+-----------+---------------------------+-----------------------------
    ↪
| Projection_4                 | 4.00    | root      |                           | test.t2.col1,
    ↪  Column#3                                                                                                  |
| └─StreamAgg_14               | 4.00    | root      |                           | group by:
    ↪ test.t2.col1, funcs:count(Column#4)->Column#3, funcs:firstrow(test.t2.col1)->test.t2.col1
    ↪  |
|   └─IndexReader_15           | 4.00    | root      |                           | index:
    ↪ StreamAgg_8
    ↪                                                                                                                |
|     └─StreamAgg_8            | 4.00    | cop[tikv] |                           | group by:
    ↪ test.t2.col1, funcs:count(1)->Column#4
    ↪                                                          |
|       └─IndexFullScan_13     | 5.00    | cop[tikv] | table:t2, index:col1(col1) | keep order:
    ↪ true, stats:pseudo                                                                                           |
+--
    ↪ ---------------------------+---------+-----------+---------------------------+-----------------------------
    ↪
5 rows in set (0.00 sec)
```

9.3.2.6.3 其他类型查询的执行计划

- 索引查询的执行计划
- Join 查询的执行计划
- 子查询的执行计划
- 视图查询的执行计划
- 分区查询的执行计划
- 开启 IndexMerge 查询的执行计划

9.3.2.7 用 EXPLAIN 查看带视图的 SQL 执行计划

EXPLAIN 语句返回的结果会显示视图引用的表和索引，而不是视图本身的名称。这是因为视图是一张虚拟表，本身并不存储任何数据。视图的定义会和查询语句的其余部分在 SQL 优化过程中进行合并。

参考 bikeshare 数据库示例（英文），以下两个示例查询的执行方式类似：

```
ALTER TABLE trips ADD INDEX (duration);
CREATE OR REPLACE VIEW long_trips AS SELECT * FROM trips WHERE duration > 3600;
EXPLAIN SELECT * FROM long_trips;
EXPLAIN SELECT * FROM trips WHERE duration > 3600;
```

```
Query OK, 0 rows affected (2 min 10.11 sec)

Query OK, 0 rows affected (0.13 sec)

+--
    ↪ ------------------------------+------------+-----------+-----------------------------------+---------
    ↪
| id                            | estRows    | task      | access object
    ↪                          | operator info                       |
+--
    ↪ ------------------------------+------------+-----------+-----------------------------------+---------
    ↪
| IndexLookUp_12                | 6372547.67 | root      |
    ↪                                        |                                   |
| ├─IndexRangeScan_10(Build)    | 6372547.67 | cop[tikv] | table:trips, index:duration(duration
    ↪ ) | range:(3600,+inf], keep order:false |
| └─TableRowIDScan_11(Probe)    | 6372547.67 | cop[tikv] | table:trips
    ↪                           | keep order:false                    |
+--
    ↪ ------------------------------+------------+-----------+-----------------------------------+---------
    ↪
3 rows in set (0.00 sec)

+--
    ↪ ------------------------------+-----------+-----------+-----------------------------------+----------
    ↪
| id                            | estRows   | task      | access object                       |
    ↪  operator info                       |
+--
    ↪ ------------------------------+-----------+-----------+-----------------------------------+----------
    ↪
| IndexLookUp_10                | 833219.37 | root      |                                     |
    ↪                                       |
| ├─IndexRangeScan_8(Build)     | 833219.37 | cop[tikv] | table:trips, index:duration(duration)
    ↪ | range:(3600,+inf], keep order:false |
| └─TableRowIDScan_9(Probe)     | 833219.37 | cop[tikv] | table:trips
    ↪ | keep order:false                    |
+--
    ↪ ------------------------------+-----------+-----------+-----------------------------------+----------
    ↪
3 rows in set (0.00 sec)
```

同样，该视图中的谓词被下推至基表：

```
EXPLAIN SELECT * FROM long_trips WHERE bike_number = 'W00950';
```

```
EXPLAIN SELECT * FROM trips WHERE bike_number = 'W00950';
```

```
+--
    ↪ ------------------------------+---------+-----------+-------------------------------------+-----------
    ↪
| id                            | estRows | task      | access object                       |
    ↪ operator info                                |
+--
    ↪ ------------------------------+---------+-----------+-------------------------------------+-----------
    ↪
| IndexLookUp_14                | 3.33    | root      |                                     |
    ↪                                                |
| ├─IndexRangeScan_11(Build)    | 3333.33 | cop[tikv] | table:trips, index:duration(duration) |
    ↪  range:(3600,+inf], keep order:false, stats:pseudo |
| └─Selection_13(Probe)         | 3.33    | cop[tikv] |                                     |
    ↪  eq(bikeshare.trips.bike_number, "W00950")         |
|   └─TableRowIDScan_12         | 3333.33 | cop[tikv] | table:trips                         |
    ↪  keep order:false, stats:pseudo                    |
+--
    ↪ ------------------------------+---------+-----------+-------------------------------------+-----------
    ↪
4 rows in set (0.00 sec)

+--
    ↪ ----------------------+-------------+-----------+--------------+-------------------------------
    ↪
| id                     | estRows     | task      | access object | operator info
    ↪                            |
+--
    ↪ ----------------------+-------------+-----------+--------------+-------------------------------
    ↪
| TableReader_7          | 43.00       | root      |               | data:Selection_6
    ↪                           |
| └─Selection_6          | 43.00       | cop[tikv] |               | eq(bikeshare.trips.
    ↪ bike_number, "W00950") |
|   └─TableFullScan_5    | 19117643.00 | cop[tikv] | table:trips   | keep order:false
    ↪                           |
+--
    ↪ ----------------------+-------------+-----------+--------------+-------------------------------
    ↪
3 rows in set (0.00 sec)
```

执行以上第一条语句时使用了索引，满足视图定义，接着在 TiDB 读取行时应用了 `bike_number = 'W00950'` 条件。执行以上第二条语句时，不存在满足该语句的索引，因此使用了 `TableFullScan`。

TiDB 使用的索引可以同时满足视图定义和语句本身，如以下组合索引所示：

```
ALTER TABLE trips ADD INDEX (bike_number, duration);
EXPLAIN SELECT * FROM long_trips WHERE bike_number = 'W00950';
EXPLAIN SELECT * FROM trips WHERE bike_number = 'W00950';
```

```
Query OK, 0 rows affected (2 min 31.20 sec)

+--
    ↪ ------------------------------+----------+-----------+-------------------------------------------------------
    ↪
| id                            | estRows  | task      | access object
    ↪                                        | operator info
    ↪                                        |
+--
    ↪ ------------------------------+----------+-----------+-------------------------------------------------------
    ↪
| IndexLookUp_13                | 63725.48 | root      |
    ↪                                                      |
    ↪                                                      |
| ├─IndexRangeScan_11(Build)    | 63725.48 | cop[tikv] | table:trips, index:bike_number(
    ↪ bike_number, duration) | range:("W00950" 3600,"W00950" +inf], keep order:false |
| └─TableRowIDScan_12(Probe)    | 63725.48 | cop[tikv] | table:trips
    ↪                                          | keep order:false
    ↪                                     |
+--
    ↪ ------------------------------+----------+-----------+-------------------------------------------------------
    ↪
3 rows in set (0.00 sec)

+--
    ↪ ------------------------------+----------+-----------+-------------------------------------------------------
    ↪
| id                            | estRows  | task      | access object
    ↪                                        | operator info                                 |
+--
    ↪ ------------------------------+----------+-----------+-------------------------------------------------------
    ↪
| IndexLookUp_10                | 19117.64 | root      |
    ↪                                                      |
    ↪                                                |
| ├─IndexRangeScan_8(Build)     | 19117.64 | cop[tikv] | table:trips, index:bike_number(
    ↪ bike_number, duration) | range:["W00950","W00950"], keep order:false |
| └─TableRowIDScan_9(Probe)     | 19117.64 | cop[tikv] | table:trips
    ↪                                          | keep order:false                              |
+--
```

```
    ↪ ----------------------------+----------+-----------+-------------------------------------------------------------
    ↪
3 rows in set (0.00 sec)
```

在第一条语句中，TiDB 能够使用组合索引的两个部分 (bike_number, duration)。在第二条语句，TiDB 仅使用了索引 (bike_number, duration) 的第一部分 bike_number。

9.3.2.7.1 其他类型查询的执行计划

- 索引查询的执行计划
- Join 查询的执行计划
- 子查询的执行计划
- 聚合查询的执行计划
- 分区查询的执行计划
- 开启 IndexMerge 查询的执行计划

9.3.2.8 用 EXPLAIN 查看分区查询的执行计划

使用 EXPLAIN 语句可以查看 TiDB 在执行查询时需要访问的分区。由于存在分区裁剪，所显示的分区通常只是所有分区的一个子集。本文档介绍了常见分区表的一些优化方式，以及如何解读 EXPLAIN 语句返回的执行计划信息。

本文档所使用的示例数据如下:

```
CREATE TABLE t1 (
 id BIGINT NOT NULL auto_increment,
 d date NOT NULL,
 pad1 BLOB,
 pad2 BLOB,
 pad3 BLOB,
 PRIMARY KEY (id,d)
) PARTITION BY RANGE (YEAR(d)) (
 PARTITION p2016 VALUES LESS THAN (2017),
 PARTITION p2017 VALUES LESS THAN (2018),
 PARTITION p2018 VALUES LESS THAN (2019),
 PARTITION p2019 VALUES LESS THAN (2020),
 PARTITION pmax VALUES LESS THAN MAXVALUE
);

INSERT INTO t1 (d, pad1, pad2, pad3) VALUES
 ('2016-01-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2016-06-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2016-09-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2017-01-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2017-06-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2017-09-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
```

```
 ('2018-01-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2018-06-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2018-09-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2019-01-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2019-06-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2019-09-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2020-01-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2020-06-01', RANDOM_BYTES(102), RANDOM_BYTES(1024), RANDOM_BYTES(1024)),
 ('2020-09-01', RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024));

INSERT INTO t1 SELECT NULL, a.d, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪ t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, a.d, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪ t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, a.d, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪ t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, a.d, RANDOM_BYTES(1024), RANDOM_BYTES(1024), RANDOM_BYTES(1024) FROM
    ↪ t1 a JOIN t1 b JOIN t1 c LIMIT 10000;

SELECT SLEEP(1);
ANALYZE TABLE t1;
```

以下示例解释了基于新建分区表 `t1` 的一条语句：

```
EXPLAIN SELECT COUNT(*) FROM t1 WHERE d = '2017-06-01';
```

```
+--
    ↪ ----------------------------+---------+-----------+---------------------------+--------------------------
    ↪
| id                            | estRows | task      | access object             | operator info
    ↪                              |
+--
    ↪ ----------------------------+---------+-----------+---------------------------+--------------------------
    ↪
| StreamAgg_21                  | 1.00    | root      |                           | funcs:count(
    ↪ Column#8)->Column#6             |
| └─TableReader_22              | 1.00    | root      |                           | data:
    ↪ StreamAgg_10                        |
|   └─StreamAgg_10              | 1.00    | cop[tikv] |                           | funcs:count
    ↪ (1)->Column#8                   |
|     └─Selection_20            | 8.87    | cop[tikv] |                           | eq(test.t1.d,
    ↪  2017-06-01 00:00:00.000000) |
|       └─TableFullScan_19      | 8870.00 | cop[tikv] | table:t1, partition:p2017 | keep order:
    ↪ false                           |
+--
    ↪ ----------------------------+---------+-----------+---------------------------+--------------------------
```

```
    ↪
5 rows in set (0.01 sec)
```

由上述 EXPLAIN 结果可知，从最末尾的 └─TableFullScan_19 算子开始，再返回到根部的 StreamAgg_21 算子的执行过程如下：

- TiDB 成功地识别出只需要访问一个分区（p2017），并将该信息在 access object 列中注明。
- └─TableFullScan_19 算子先对整个分区进行扫描，然后执行 └─Selection_20 算子筛选起始日期为 2017-06-01 00:00:00.000000 的行。
- 之后，└─Selection_20 算子匹配的行在 Coprocessor 中进行流式聚合，Coprocessor 本身就可以理解聚合函数 count。
- 每个 Coprocessor 请求会发送一行数据给 TiDB 的 └─TableReader_22 算子，然后将数据在 StreamAgg_21 算子下进行流式聚合，再将一行数据返回给客户端。

以下示例中，分区裁剪不会消除任何分区：

```
EXPLAIN SELECT COUNT(*) FROM t1 WHERE YEAR(d) = 2017;
```

```
+--
    ↪ ---------------------------------+----------+-----------+--------------------------+-----------------
    ↪
| id                                | estRows  | task      | access object            |
    ↪ operator info                     |
+--
    ↪ ---------------------------------+----------+-----------+--------------------------+-----------------
    ↪
| HashAgg_20                        | 1.00     | root      |                          | funcs:
    ↪ count(Column#7)->Column#6  |
| └─PartitionUnion_21               | 5.00     | root      |                          |
    ↪                                   |
|   ├─StreamAgg_36                  | 1.00     | root      |                          | funcs:
    ↪ count(Column#9)->Column#7  |
|   │ └─TableReader_37              | 1.00     | root      |                          | data:
    ↪ StreamAgg_25                |
|   │   └─StreamAgg_25              | 1.00     | cop[tikv] |                          | funcs:
    ↪ count(1)->Column#9         |
|   │     └─Selection_35            | 6000.00  | cop[tikv] |                          | eq(
    ↪ year(test.t1.d), 2017)        |
|   │       └─TableFullScan_34      | 7500.00  | cop[tikv] | table:t1, partition:p2016 | keep
    ↪ order:false                 |
|   ├─StreamAgg_55                  | 1.00     | root      |                          | funcs:
    ↪ count(Column#11)->Column#7 |
|   │ └─TableReader_56              | 1.00     | root      |                          | data:
    ↪ StreamAgg_44                |
|   │   └─StreamAgg_44              | 1.00     | cop[tikv] |                          | funcs:
    ↪ count(1)->Column#11        |
```

```
|   |   └─Selection_54             | 14192.00 | cop[tikv] |                           | eq(
    ↪ year(test.t1.d), 2017)       |
|   |     └─TableFullScan_53       | 17740.00 | cop[tikv] | table:t1, partition:p2017 | keep
    ↪ order:false                  |
|   ├─StreamAgg_74                 | 1.00     | root      |                           | funcs:
    ↪ count(Column#13)->Column#7 |
|   | └─TableReader_75             | 1.00     | root      |                           | data:
    ↪ StreamAgg_63                 |
|   |   └─StreamAgg_63             | 1.00     | cop[tikv] |                           | funcs:
    ↪ count(1)->Column#13          |
|   |     └─Selection_73           | 3977.60  | cop[tikv] |                           | eq(
    ↪ year(test.t1.d), 2017)       |
|   |       └─TableFullScan_72     | 4972.00  | cop[tikv] | table:t1, partition:p2018 | keep
    ↪ order:false                  |
|   ├─StreamAgg_93                 | 1.00     | root      |                           | funcs:
    ↪ count(Column#15)->Column#7 |
|   | └─TableReader_94             | 1.00     | root      |                           | data:
    ↪ StreamAgg_82                 |
|   |   └─StreamAgg_82             | 1.00     | cop[tikv] |                           | funcs:
    ↪ count(1)->Column#15          |
|   |     └─Selection_92           | 20361.60 | cop[tikv] |                           | eq(
    ↪ year(test.t1.d), 2017)       |
|   |       └─TableFullScan_91     | 25452.00 | cop[tikv] | table:t1, partition:p2019 | keep
    ↪ order:false                  |
|   └─StreamAgg_112                | 1.00     | root      |                           | funcs:
    ↪ count(Column#17)->Column#7 |
|     └─TableReader_113            | 1.00     | root      |                           | data:
    ↪ StreamAgg_101                |
|       └─StreamAgg_101            | 1.00     | cop[tikv] |                           | funcs:
    ↪ count(1)->Column#17          |
|         └─Selection_111          | 8892.80  | cop[tikv] |                           | eq(
    ↪ year(test.t1.d), 2017)       |
|           └─TableFullScan_110    | 11116.00 | cop[tikv] | table:t1, partition:pmax  | keep
    ↪ order:false                  |
+--
    ↪ ---------------------------------+----------+-----------+---------------------------+-----------------
    ↪
27 rows in set (0.00 sec)
```

由上述 EXPLAIN 结果可知：

- TiDB 认为需要访问所有分区 (p2016..pMax)。这是因为 TiDB 将谓词 YEAR (d) = 2017 视为 non-sargable。这个问题并非是 TiDB 特有的。
- 在扫描每个分区时，Selection 算子将筛选出年份不为 2017 的行。
- 在每个分区上会执行流式聚合，以计算匹配的行数。

- └─PartitionUnion_21 算子会合并访问每个分区后的结果。

9.3.2.8.1 其他类型查询的执行计划

- 索引查询的执行计划
- Join 查询的执行计划
- 子查询的执行计划
- 聚合查询的执行计划
- 视图查询的执行计划
- 开启 IndexMerge 查询的执行计划

9.3.2.9 用 EXPLAIN 查看开启 IndexMerge 的 SQL 执行计划

IndexMerge 是 TiDB v4.0 中引入的一种对表的新访问方式。在这种访问方式下，TiDB 优化器可以选择对一张表使用多个索引，并将每个索引的返回结果进行合并。在某些场景下，这种访问方式能够减少大量不必要的数据扫描，提升查询的执行效率。

```
EXPLAIN SELECT * from t where a = 1 or b = 1;
+--
    ↪ ----------------------+----------+-----------+---------------+---------------------------------+
    ↪
| id                      | estRows  | task      | access object | operator info
    ↪                         |
+--
    ↪ ----------------------+----------+-----------+---------------+---------------------------------+
    ↪
| TableReader_7           | 8000.00  | root      |               | data:Selection_6
    ↪                       |
| └─Selection_6           | 8000.00  | cop[tikv] |               | or(eq(test.t.a, 1), eq(test.t
    ↪ .b, 1)) |
|   └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo       |
+--
    ↪ ----------------------+----------+-----------+---------------+---------------------------------+
    ↪
set @@tidb_enable_index_merge = 1;
explain select * from t use index(idx_a, idx_b) where a > 1 or b > 1;
+--
    ↪ ----------------------------+---------+-----------+-------------------------+-----------------------
    ↪
| id                            | estRows | task      | access object           | operator info
    ↪                                 |
+--
    ↪ ----------------------------+---------+-----------+-------------------------+-----------------------
    ↪
```

```
| IndexMerge_16                 | 6666.67 | root      |                          |
    ↪                                                 |
| ├─IndexRangeScan_13(Build)    | 3333.33 | cop[tikv] | table:t, index:idx_a(a) | range:(1,+inf
    ↪ ], keep order:false, stats:pseudo |
| ├─IndexRangeScan_14(Build)    | 3333.33 | cop[tikv] | table:t, index:idx_b(b) | range:(1,+inf
    ↪ ], keep order:false, stats:pseudo |
| └─TableRowIDScan_15(Probe)    | 6666.67 | cop[tikv] | table:t                 | keep order:
    ↪ false, stats:pseudo                |
+--
    ↪ ------------------------------+---------+-----------+-------------------------+-----------------------------
    ↪
```

例如，在上述示例中，过滤条件是使用 OR 条件连接的 WHERE 子句。在启用 IndexMerge 前，每个表只能使用一个索引，不能将 a = 1 下推到索引 a，也不能将 b = 1 下推到索引 b。当 t 中存在大量数据时，全表扫描的效率会很低。针对这类场景，TiDB 引入了对表的新访问方式 IndexMerge。

在 IndexMerge 访问方式下，优化器可以选择对一张表使用多个索引，并将每个索引的返回结果进行合并，生成以上示例中后一个 IndexMerge 的执行计划。此时的 IndexMerge_16 算子有三个子节点，其中 IndexRangeScan_13 和 IndexRangeScan_14 根据范围扫描得到符合条件的所有 RowID，再由 TableRowIDScan_15 算子根据这些 RowID 精确地读取所有满足条件的数据。

其中对于 IndexRangeScan/TableRangeScan 一类按范围进行的扫表操作，EXPLAIN 表中 operator info 列相比于其他扫表操作，多了被扫描数据的范围这一信息。比如上面的例子中，IndexRangeScan_13 算子中的 range:(1,+inf] 这一信息表示该算子扫描了从 1 到正无穷这个范围的数据。

> 注意：
>
> 目前，TiDB 的 IndexMerge 特性在 TiDB 4.0.0-rc.1 版本中默认关闭。同时 4.0 版本中的 IndexMerge 目前支持的场景仅限于析取范式（or 连接的表达式），暂不支持合取范式（and 连接的表达式）。开启 IndexMerge 特性有以下方法：
>
> - 设置系统变量 tidb_enable_index_merge=1；
> - 在查询中使用 SQL 优化器 Hint USE_INDEX_MERGE。
>
> SQL Hint 的优先级高于系统变量。

9.3.2.9.1 其他类型查询的执行计划

- 索引查询的执行计划
- Join 查询的执行计划
- 子查询的执行计划
- 聚合查询的执行计划
- 视图查询的执行计划
- 分区查询的执行计划

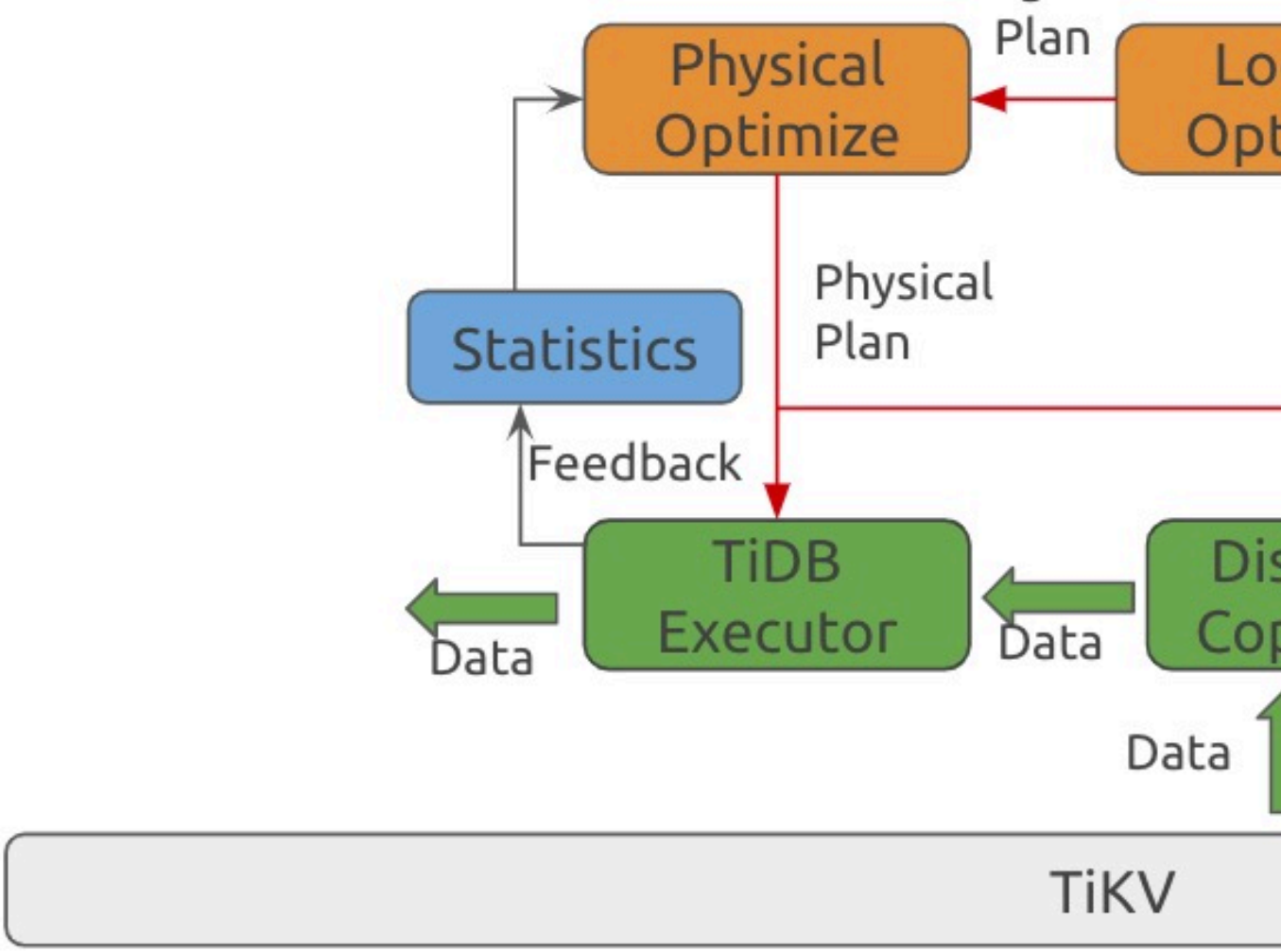


图 80: SQL Optimization

在经过了 parser 对原始查询文本的解析以及一些简单的合法性验证后，TiDB 首先会对查询做一些逻辑上的等价变化，详细的变化可以查询逻辑优化章节。

通过这些等价变化，使得这个查询在逻辑执行计划上可以变得更易于处理。在等价变化结束之后，TiDB 会得到一个与原始查询等价的查询计划结构，之后根据数据分布、以及一个算子具体的执行开销，来获得一个最终的执行计划，这部分内容可以查询物理优化章节。

同时，TiDB 在执行PREPARE 语句时，可以选择开启缓存来降低 TiDB 生成执行计划的开销，这部分内容会在执行计划缓存一节中介绍。

9.3.3.2 逻辑优化

9.3.3.2.1 逻辑优化

本章节将对一些比较关键的逻辑改写进行说明，帮助大家理解 TiDB 如何生成最终的查询计划。比如在 TiDB 输入 `select * from t where t.a in (select t1.a from t1 where t1.b=t.b)` 这个查询时，在最终的执行计划中将看不到这个 `t.a in (select t1.a from t1 where t1.b=t.b)` 这个 IN 子查询的存在，这便是因为 TiDB 对这里进行了一些改写。

本章节会介绍如下几个关键改写：

- 子查询相关的优化
- 列裁剪
- 关联子查询去关联
- Max/Min 消除
- 谓词下推
- 分区裁剪
- TopN 和 Limit 下推
- Join Reorder

9.3.3.2.2 子查询相关的优化

本文主要介绍子查询相关的优化。

通常会遇到如下情况的子查询：

- `NOT IN (SELECT ... FROM ...)`
- `NOT EXISTS (SELECT ... FROM ...)`
- `IN (SELECT ... FROM ..)`
- `EXISTS (SELECT ... FROM ...)`
- `... >/>=/</<=/=/!= (SELECT ... FROM ...)`

有时，子查询中包含了非子查询中的列，如 `select * from t where t.a in (select * from t2 where t.b =t2.b)` 中，子查询中的 `t.b` 不是子查询中的列，而是从子查询外面引入的列。这种子查询通常会被称为关联子查询，外部引入的列会被称为关联列，关联子查询相关的优化参见关联子查询去关联。本文主要关注不涉及关联列的子查询。

子查询默认会以理解 TiDB 执行计划中提到的 `semi join` 作为默认的执行方式，同时对于一些特殊的子查询，TiDB 会做一些逻辑上的替换使得查询可以获得更好的执行性能。

`... < ALL (SELECT ... FROM ...)` 或者 `... > ANY (SELECT ... FROM ...)`

对于这种情况，可以将 `ALL` 或者 `ANY` 用 `MAX` 以及 `MIN` 来代替。不过由于在表为空时，`MAX(EXPR)` 以及 `MIN(EXPR)` 的结果会为 `NULL`，其表现形式和 `EXPR` 是有 `NULL` 值的结果一样。以及外部表达式结果为 `NULL` 时也会影响表达式的最终结果，因此这里完整的改写会是如下的形式：

- `t.id < all(select s.id from s)` 会被改写为 `t.id < min(s.id)and if(sum(s.id is null)!= 0, null, true)`。
- `t.id < any (select s.id from s)` 会被改写为 `t.id < max(s.id)or if(sum(s.id is null)!= 0, null, false)`。

`... != ANY (SELECT ... FROM ...)`

对于这种情况，当子查询中不同值的个数只有一种的话，那只要和这个值对比就即可。如果子查询中不同值的个数多于 1 个，那么必然会有不相等的情况出现。因此这样的子查询可以采取如下的改写手段：

- `select * from t where t.id != any (select s.id from s)` 会被改写为 `select t.* from t, (select s.id, count(distinct s.id)as cnt_distinct from s)where (t.id != s.id or cnt_distinct > 1)`

`... = ALL (SELECT ... FROM ...)`

对于这种情况，当子查询中不同值的个数多于一种的话，那么这个表达式的结果必然为假。因此这样的子查询在 TiDB 中会改写为如下的形式：

- `select * from t where t.id = all (select s.id from s)` 会被改写为 `select t.* from t, (select s.id, count(distinct s.id)as cnt_distinct from s)where (t.id = s.id and cnt_distinct <= 1)`

`... IN (SELECT ... FROM ...)`

对于这种情况，会将其 `IN` 的子查询改写为 `SELECT ... FROM ... GROUP ...` 的形式，然后将 `IN` 改写为普通的 `JOIN` 的形式。如 `select * from t1 where t1.a in (select t2.a from t2)` 会被改写为 `select t1.* from t1, (select distinct(a)a from t2)t2 where t1.a = t2.a` 的形式。同时这里的 `DISTINCT` 可以在 `t2.a` 具有 `UNIQUE` 属性时被自动消去。

```
explain select * from t1 where t1.a in (select t2.a from t2);
```

```
+--
    ↪ ---------------------------+---------+-----------+----------------------+-----------------------------
    ↪
| id                           | estRows | task      | access object          | operator info
    ↪                                                                 |
+--
    ↪ ---------------------------+---------+-----------+----------------------+-----------------------------
    ↪
| IndexJoin_12                 | 9990.00 | root      |                        | inner join, inner
    ↪ :TableReader_11, outer key:test.t2.a, inner key:test.t1.a |
| ├─HashAgg_21(Build)          | 7992.00 | root      |                        | group by:test.t2
    ↪ .a, funcs:firstrow(test.t2.a)->test.t2.a                  |
| │ └─IndexReader_28           | 9990.00 | root      |                        | index:
    ↪ IndexFullScan_27                                                |
| │   └─IndexFullScan_27       | 9990.00 | cop[tikv] | table:t2, index:idx(a) | keep order:false
    ↪ , stats:pseudo                                            |
| └─TableReader_11(Probe)      | 1.00    | root      |                        | data:
    ↪ TableRangeScan_10                                               |
|   └─TableRangeScan_10        | 1.00    | cop[tikv] | table:t1               | range: decided
    ↪ by [test.t2.a], keep order:false, stats:pseudo             |
+--
    ↪ ---------------------------+---------+-----------+----------------------+-----------------------------
    ↪
```

这个改写会在 `IN` 子查询相对较小，而外部查询相对较大时产生更好的执行性能。因为不经过改写的情况下，我们无法使用以 `t2` 为驱动表的 `index join`。同时这里的弊端便是，当改写删成的聚合无法被自动消去且 `t2` 表比较大时，反而会影响查询的性能。目前 TiDB 中使用tidb_opt_insubq_to_join_and_agg 变量来控制这个优化的打开与否。当遇到不合适这个优化的情况可以手动关闭。

`EXISTS` 子查询以及 `... >/>=/</<=/=/!= (SELECT ... FROM ...)`

当前对于这种场景的子查询，当它不是关联子查询时，TiDB 会在优化阶段提前展开它，将其直接替换为一个结果集直接判断结果。如下图中，EXISTS 会提前在优化阶段被执行为 TRUE，从而不会在最终的执行结果中看到它。

```
create table t1(a int);
create table t2(a int);
insert into t2 values(1);
explain select * from t1 where exists (select * from t2);
```

```
+--
    ↪ ---------------------+----------+-----------+---------------+------------------------------+
    ↪
| id                    | estRows  | task      | access object | operator info
    ↪ |
+--
    ↪ ---------------------+----------+-----------+---------------+------------------------------+
    ↪
| TableReader_12        | 10000.00 | root      |               | data:TableFullScan_11
    ↪ |
| └─TableFullScan_11    | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:pseudo
    ↪  |
+--
    ↪ ---------------------+----------+-----------+---------------+------------------------------+
    ↪
```

9.3.3.2.3 列裁剪

列裁剪的基本思想在于：对于算子中实际用不上的列，优化器在优化的过程中没有必要保留它们。对这些列的删除会减少 I/O 资源占用，并为后续的优化带来便利。下面给出一个列重复的例子：

假设表 t 里面有 a b c d 四列，执行如下语句：

```
select a from t where b > 5
```

在该查询的过程中，t 表实际上只有 a, b 两列会被用到，而 c, d 的数据则显得多余。对应到该语句的查询计划，Selection 算子会用到 b 列，下面接着的 DataSource 算子会用到 a, b 两列，而剩下 c, d 两列则都可以裁剪掉，DataSource 算子在读数据时不需要将它们读进来。

出于上述考量，TiDB 会在逻辑优化阶段进行自上而下的扫描，裁剪不需要的列，减少资源浪费。该扫描过程称作“列裁剪”，对应逻辑优化规则中的 columnPruner。如果要关闭这个规则，可以在参照优化规则及表达式下推的黑名单中的关闭方法。

9.3.3.2.4 关联子查询去关联

子查询相关的优化中介绍了当没有关联列时，TiDB 是如何处理子查询的。由于为关联子查询解除关联依赖比较复杂，本文档中会介绍一些简单的场景以及这个优化规则的适用范围。

简介

以 `select * from t1 where t1.a < (select sum(t2.a)from t2 where t2.b = t1.b)` 为例，这里子查询 `t1.a < (select sum(t2.a)from t2 where t2.b = t1.b)` 中涉及了关联列上的条件 `t2.b=t1.b`，不过恰好由于这是一个等值条件，因此可以将其等价的改写为 `select t1.* from t1, (select b, sum(a)sum_a from t2 group by b)t2 where t1.b = t2.b and t1.a < t2.sum_a;`。这样，一个关联子查询就被重新改写为 JOIN 的形式。

TiDB 之所以要进行这样的改写，是因为关联子查询每次子查询执行时都是要和它的外部查询结果绑定的。在上面的例子中，如果 `t1.a` 有一千万个值，那这个子查询就要被重复执行一千万次，因为 `t2.b=t1.b` 这个条件会随着 `t1.a` 值的不同而发生变化。当通过一些手段将关联依赖解除后，这个子查询就只需要被执行一次了。

限制

这种改写的弊端在于，在关联没有被解除时，优化器是可以使用关联列上的索引的。也就是说，虽然这个子查询可能被重复执行多次，但是每次都可以使用索引过滤数据。而解除关联的变换上，通常是会导致关联列的位置发生改变而导致虽然子查询只被执行了一次，但是单次执行的时间会比没有解除关联时的单次执行时间长。

因此，在外部的值比较少的情况下，不解除关联依赖反而可能对执行性能更优帮助。这时可以通过优化规则及表达式下推的黑名单中关闭子查询去关联优化规则的方式来关闭这个优化。

样例

```
create table t1(a int, b int);
create table t2(a int, b int, index idx(b));
explain select * from t1 where t1.a < (select sum(t2.a) from t2 where t2.b = t1.b);
```

```
+--
    ↪ --------------------------------+----------+-----------+---------------+--------------------------------
    ↪
| id                                | estRows  | task      | access object | operator info
    ↪                                                                                  |
+--
    ↪ --------------------------------+----------+-----------+---------------+--------------------------------
    ↪
| HashJoin_11                       | 9990.00  | root      |               | inner join, equal:[eq
    ↪ (test.t1.b, test.t2.b)], other cond:lt(cast(test.t1.a), Column#7)  |
| ├─HashAgg_23(Build)               | 7992.00  | root      |               | group by:test.t2.b,
    ↪ funcs:sum(Column#8)->Column#7, funcs:firstrow(test.t2.b)->test.t2.b |
| │ └─TableReader_24                | 7992.00  | root      |               | data:HashAgg_16
    ↪                                                                      |
| │   └─HashAgg_16                  | 7992.00  | cop[tikv] |               | group by:test.t2.b,
    ↪ funcs:sum(test.t2.a)->Column#8                                       |
| │     └─Selection_22              | 9990.00  | cop[tikv] |               | not(isnull(test.t2.b
    ↪ ))                                                                   |
| │       └─TableFullScan_21        | 10000.00 | cop[tikv] | table:t2      | keep order:false,
    ↪ stats:pseudo                                                           |
| └─TableReader_15(Probe)           | 9990.00  | root      |               | data:Selection_14
    ↪                                                                        |
```

```
|   └─Selection_14                | 9990.00  | cop[tikv] |               | not(isnull(test.t1.b
    ↪ ))                                                                    |
|     └─TableFullScan_13          | 10000.00 | cop[tikv] | table:t1      | keep order:false,
    ↪ stats:pseudo                                                            |
+--
    ↪ -------------------------------+----------+-----------+---------------+-------------------------------
    ↪
```

上面是优化生效的情况，可以看到 HashJoin_11 是一个普通的 inner join。

接下来，关闭关联规则：

```
insert into mysql.opt_rule_blacklist values("decorrelate");
admin reload opt_rule_blacklist;
explain select * from t1 where t1.a < (select sum(t2.a) from t2 where t2.b = t1.b);
```

```
+--
    ↪ -------------------------------------+----------+-----------+-----------------------+----------------
    ↪
| id                                  | estRows  | task      | access object         |
    ↪ operator info                                                          |
+--
    ↪ -------------------------------------+----------+-----------+-----------------------+----------------
    ↪
| Projection_10                       | 10000.00 | root      |                       | test.
    ↪ t1.a, test.t1.b                                                     |
| └─Apply_12                           | 10000.00 | root      |                       |
    ↪ CARTESIAN inner join, other cond:lt(cast(test.t1.a), Column#7)              |
|   ├─TableReader_14(Build)           | 10000.00 | root      |                       | data:
    ↪ TableFullScan_13                                                    |
|   │ └─TableFullScan_13              | 10000.00 | cop[tikv] | table:t1              | keep
    ↪ order:false, stats:pseudo                                           |
|   └─MaxOneRow_15(Probe)             | 1.00     | root      |                       |
    ↪                                                                         |
|     └─HashAgg_27                    | 1.00     | root      |                       | funcs
    ↪ :sum(Column#10)->Column#7                                           |
|       └─IndexLookUp_28              | 1.00     | root      |                       |
    ↪                                                                         |
|         ├─IndexRangeScan_25(Build)  | 10.00    | cop[tikv] | table:t2, index:idx(b) | range
    ↪ : decided by [eq(test.t2.b, test.t1.b)], keep order:false, stats:pseudo |
|         └─HashAgg_17(Probe)         | 1.00     | cop[tikv] |                       | funcs
    ↪ :sum(test.t2.a)->Column#10                                          |
|           └─TableRowIDScan_26       | 10.00    | cop[tikv] | table:t2              | keep
    ↪ order:false, stats:pseudo                                           |
+--
    ↪ -------------------------------------+----------+-----------+-----------------------+----------------
    ↪
```

在执行了关闭关联规则的语句后，可以在 IndexRangeScan_25(Build) 的 operator info 中看到 range: decided by [eq(test.t2.b, test.t1.b)]。这部分信息就是关联依赖未被解除时，TiDB 使用关联条件进行索引范围查询的显示结果。

9.3.3.2.5 Max/Min 函数消除规则

在 SQL 中包含了 max/min 函数时，查询优化器会尝试使用 max/min 消除优化规则来将 max/min 聚合函数转换为 TopN 算子，从而能够有效地利用索引进行查询。

根据 select 语句中 max/min 函数的个数，这一优化规则有以下两种表现形式：

- 只有一个 max/min 函数时的优化规则
- 存在多个 max/min 函数时的优化规则

只有一个 max/min 函数时的优化规则

当一个 SQL 满足以下条件时，就会应用这个规则：

- 只有一个聚合函数，且为 max 或者 min 函数。
- 聚合函数没有相应的 group by 语句。

例如：

```
select max(a) from t
```

这时 max/min 消除优化规则会将其重写为：

```
select max(a) from (select a from t where a is not null order by a desc limit 1) t
```

这个新的 SQL 语句在 a 列存在索引（或 a 列是某个联合索引的前缀）时，能够利用索引只扫描一行数据来得到最大或者最小值，从而避免对整个表的扫描。

上述例子最终得到的执行计划如下:

```
mysql> explain select max(a) from t;
+------------------------------+---------+-----------+-------------------------+-------------------------------
    ↪
| id                           | estRows | task      | access object           | operator info
    ↪                         |
+------------------------------+---------+-----------+-------------------------+-------------------------------
    ↪
| StreamAgg_13                 | 1.00    | root      |                         | funcs:max(test.t
    ↪ .a)->Column#4       |
| └─Limit_17                   | 1.00    | root      |                         | offset:0, count
    ↪ :1                  |
|   └─IndexReader_27           | 1.00    | root      |                         | index:Limit_26
    ↪                         |
```

```
|     └─Limit_26               | 1.00    | cop[tikv] |                          | offset:0, count
    ↪ :1                    |
|       └─IndexFullScan_25     | 1.00    | cop[tikv] | table:t, index:idx_a(a) | keep order:true
    ↪ , desc, stats:pseudo |
+------------------------------+---------+-----------+-------------------------+-------------------------------
    ↪
5 rows in set (0.00 sec)
```

存在多个 max/min 函数时的优化规则

当一个 SQL 满足以下条件时，就会应用这个规则：

- 有多个聚合函数，且所有的聚合函数都是 max/min
- 聚合函数没有相应的 `group by` 语句。
- 每个 max/min 聚合函数参数中的列都有索引能够保序。

下面是一个简单的例子：

```
select max(a) - min(a) from t
```

优化规则会先检查 a 列是否存在索引能够为其保序，如果存在，这个 SQL 会先被重写为两个子查询的笛卡尔积：

```
select max_a - min_a
from
    (select max(a) as max_a from t) t1,
    (select min(a) as min_a from t) t2
```

这样，两个子句中的 max/min 函数就可以使用上述“只有一个 max/min 函数时的优化规则”分别进行优化，最终重写为：

```
select max_a - min_a
from
    (select max(a) as max_a from (select a from t where a is not null order by a desc limit 1) t)
        ↪  t1,
    (select min(a) as min_a from (select a from t where a is not null order by a asc limit 1) t)
        ↪ t2
```

同样的，如果 a 列能够使用索引保序，那这个优化只会扫描两行数据，避免了对整个表的扫描。但如果 a 列没有可以保序的索引，这个变换会使原本只需一次的全表扫描变成两次，因此这个规则就不会被应用。

最后得到的执行计划：

```
mysql> explain select max(a)-min(a) from t;
+----------------------------------+---------+-----------+-------------------------+-------------------------
    ↪
| id                               | estRows | task      | access object           | operator
    ↪ info                        |
```

```
+-----------------------------------+---------+-----------+-------------------------+-----------------------------
    ↪
| Projection_17                     | 1.00    | root      |                         | minus(
    ↪ Column#4, Column#5)->Column#6 |
| └─HashJoin_18                     | 1.00    | root      |                         | CARTESIAN
    ↪  inner join                |
|   ├─StreamAgg_45(Build)           | 1.00    | root      |                         | funcs:min
    ↪ (test.t.a)->Column#5       |
|   │ └─Limit_49                    | 1.00    | root      |                         | offset:0,
    ↪  count:1                   |
|   │   └─IndexReader_59            | 1.00    | root      |                         | index:
    ↪ Limit_58                     |
|   │     └─Limit_58                | 1.00    | cop[tikv] |                         | offset:0,
    ↪  count:1                   |
|   │       └─IndexFullScan_57      | 1.00    | cop[tikv] | table:t, index:idx_a(a) | keep
    ↪ order:true, stats:pseudo       |
|   └─StreamAgg_24(Probe)           | 1.00    | root      |                         | funcs:max
    ↪ (test.t.a)->Column#4       |
|     └─Limit_28                    | 1.00    | root      |                         | offset:0,
    ↪  count:1                   |
|       └─IndexReader_38            | 1.00    | root      |                         | index:
    ↪ Limit_37                     |
|         └─Limit_37                | 1.00    | cop[tikv] |                         | offset:0,
    ↪  count:1                   |
|           └─IndexFullScan_36      | 1.00    | cop[tikv] | table:t, index:idx_a(a) | keep
    ↪ order:true, desc, stats:pseudo |
+-----------------------------------+---------+-----------+-------------------------+-----------------------------
    ↪
12 rows in set (0.01 sec)
```

9.3.3.2.6 谓词下推

本文档介绍 TiDB 逻辑优化规则中的谓词下推规则，旨在让读者对谓词下推形成理解，并了解常见的谓词下推适用及不适用的场景。

谓词下推将查询语句中的过滤表达式计算尽可能下推到距离数据源最近的地方，以尽早完成数据的过滤，进而显著地减少数据传输或计算的开销。

示例

以下通过一些例子对谓词下推优化进行说明，其中示例 1、2、3 为谓词下推适用的案例，示例 4、5、6 为谓词下推不适用的案例。

示例 1: 谓词下推到存储层

```
create table t(id int primary key, a int);
explain select * from t where a < 1;
```

```
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
| id                    | estRows  | task      | access object | operator info
    ↪                  |
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
| TableReader_7           | 3323.33  | root      |               | data:Selection_6
    ↪               |
| └─Selection_6           | 3323.33  | cop[tikv] |               | lt(test.t.a, 1)
    ↪                |
|   └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo |
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
3 rows in set (0.00 sec)
```

在该查询中，将谓词 `a < 1` 下推到 TiKV 上对数据进行过滤，可以减少由于网络传输带来的开销。

示例 2: 谓词下推到存储层

```
create table t(id int primary key, a int not null);
explain select * from t where a < substring('123', 1, 1);
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
| id                    | estRows  | task      | access object | operator info
    ↪                  |
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
| TableReader_7           | 3323.33  | root      |               | data:Selection_6
    ↪               |
| └─Selection_6           | 3323.33  | cop[tikv] |               | lt(test.t.a, 1)
    ↪                |
|   └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo |
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
```

该查询与示例 1 中的查询生成了完成一样的执行计划，这是因为谓词 `a < substring('123', 1, 1)` 的 `substring` 的入参均为常量，因此可以提前计算，进而简化得到等价的谓词 `a < 1`。进一步的，可以将 `a < 1` 下推至 TiKV 上。

示例 3: 谓词下推到 join 下方

```
create table t(id int primary key, a int not null);
create table s(id int primary key, a int not null);
explain select * from t join s on t.a = s.a where t.a < 1;
+--
    ↪ ---------------------------+----------+-----------+---------------+--------------------------------
    ↪
| id                           | estRows  | task      | access object | operator info
    ↪                               |
+--
    ↪ ---------------------------+----------+-----------+---------------+--------------------------------
    ↪
| HashJoin_8                   | 4154.17  | root      |               | inner join, equal:[eq(
    ↪ test.t.a, test.s.a)] |
| ├─TableReader_15(Build)      | 3323.33  | root      |               | data:Selection_14
    ↪                            |
| │ └─Selection_14             | 3323.33  | cop[tikv] |               | lt(test.s.a, 1)
    ↪                              |
| │   └─TableFullScan_13       | 10000.00 | cop[tikv] | table:s       | keep order:false, stats:
    ↪ pseudo             |
| └─TableReader_12(Probe)      | 3323.33  | root      |               | data:Selection_11
    ↪                            |
|   └─Selection_11             | 3323.33  | cop[tikv] |               | lt(test.t.a, 1)
    ↪                              |
|     └─TableFullScan_10       | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo             |
+--
    ↪ ---------------------------+----------+-----------+---------------+--------------------------------
    ↪
7 rows in set (0.00 sec)
```

在该查询中，将谓词 `t.a < 1` 下推到 join 前进行过滤，可以减少 join 时的计算开销。

此外，这条 SQL 执行的是内连接，且 ON 条件是 `t.a = s.a`，可以由 `t.a < 1` 推导出谓词 `s.a < 1`，并将其下推至 join 运算前对 s 表进行过滤，可以进一步减少 join 时的计算开销。

示例 4: 存储层不支持的谓词无法下推

```
create table t(id int primary key, a int not null);
desc select * from t where substring('123', a, 1) = '1';
+--
    ↪ ----------------------+---------+-----------+---------------+--------------------------------+
    ↪
| id                      | estRows | task      | access object | operator info
    ↪                         |
+--
    ↪ ----------------------+---------+-----------+---------------+--------------------------------+
```

```
    ↪
| Selection_7             | 2.00    | root      |               | eq(substring("123", test.t.a,
    ↪ 1), "1") |
| └─TableReader_6         | 2.00    | root      |               | data:TableFullScan_5
    ↪                    |
|   └─TableFullScan_5     | 2.00    | cop[tikv] | table:t       | keep order:false, stats:pseudo
    ↪          |
+--
    ↪ ----------------------+---------+-----------+---------------+--------------------------------+
    ↪
```

在该查询中，存在谓词 `substring('123', a, 1)= '1'`。

从 explain 结果中可以看到，该谓词没有被下推到 TiKV 上进行计算，这是因为 TiKV coprocessor 中没有对 `substring` 内置函数进行支持，因此无法将其下推到 TiKV 上。

示例 5: 外连接中内表上的谓词不能下推

```
create table t(id int primary key, a int not null);
create table s(id int primary key, a int not null);
explain select * from t left join s on t.a = s.a where s.a is null;
+--
    ↪ ----------------------------+----------+-----------+---------------+--------------------------------
    ↪
| id                            | estRows  | task      | access object | operator info
    ↪                                  |
+--
    ↪ ----------------------------+----------+-----------+---------------+--------------------------------
    ↪
| Selection_7                   | 10000.00 | root      |               | isnull(test.s.a)
    ↪                                 |
| └─HashJoin_8                  | 12500.00 | root      |               | left outer join, equal
    ↪ :[eq(test.t.a, test.s.a)] |
|   ├─TableReader_13(Build)     | 10000.00 | root      |               | data:TableFullScan_12
    ↪                           |
|   │ └─TableFullScan_12        | 10000.00 | cop[tikv] | table:s       | keep order:false, stats
    ↪ :pseudo                   |
|   └─TableReader_11(Probe)     | 10000.00 | root      |               | data:TableFullScan_10
    ↪                           |
|     └─TableFullScan_10        | 10000.00 | cop[tikv] | table:t       | keep order:false, stats
    ↪ :pseudo                   |
+--
    ↪ ----------------------------+----------+-----------+---------------+--------------------------------
    ↪
6 rows in set (0.00 sec)
```

在该查询中，内表 s 上存在谓词 `s.a is null`。

从 explain 中可以看到，该谓词没有被下推到 join 前进行计算，这是因为外连接在不满足 on 条件时会对内表填充 NULL，而在该查询中 `s.a is null` 用来对 join 后的结果进行过滤，如果将其下推到 join 前在内表上进行过滤，则下推前后不等价, 因此不可进行下推。

示例 6: 谓词中包含用户变量时不能下推

```
create table t(id int primary key, a char);
set @a = 1;
explain select * from t where a < @a;
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
| id                      | estRows  | task      | access object | operator info
    ↪                   |
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
| Selection_5             | 8000.00  | root      |               | lt(test.t.a, getvar("a"))
    ↪      |
| └─TableReader_7         | 10000.00 | root      |               | data:TableFullScan_6
    ↪           |
|   └─TableFullScan_6     | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo |
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
3 rows in set (0.00 sec)
```

在该查询中，表 t 上存在谓词 `a < @a`，其中 @a 为值为 1 的用户变量。

从 explain 中可以看到，该谓词没有像示例 2 中一样，将谓词简化为 `a < 1` 并下推到 TiKV 上进行计算。这是因为，用户变量 @a 的值可能会某些场景下在查询过程中发生改变，且 TiKV 对于用户变量 @a 的值不可知，因此 TiDB 不会将 @a 替换为 1，且不会下推至 TiKV 上进行计算。

一个帮助理解的例子如下：

```
create table t(id int primary key, a int);
insert into t values(1, 1), (2,2);
set @a = 1;
select id, a, @a:=@a+1 from t where a = @a;
+----+------+----------+
| id | a    | @a:=@a+1 |
+----+------+----------+
|  1 |    1 | 2        |
|  2 |    2 | 3        |
+----+------+----------+
2 rows in set (0.00 sec)
```

可以从在该查询中看到，@a 的值会在查询过程中发生改变，因此如果将 a = @a 替换为 a = 1 并下推至 TiKV，则优化前后不等价。

9.3.3.2.7 分区裁剪

分区裁剪是只有当目标表为分区表时，才可以进行的一种优化方式。分区裁剪通过分析查询语句中的过滤条件，只选择可能满足条件的分区，不扫描匹配不上的分区，进而显著地减少计算的数据量。

例如：

```
CREATE TABLE t1 (
 id INT NOT NULL PRIMARY KEY,
 pad VARCHAR(100)
)
PARTITION BY RANGE COLUMNS(id) (
 PARTITION p0 VALUES LESS THAN (100),
 PARTITION p1 VALUES LESS THAN (200),
 PARTITION p2 VALUES LESS THAN (MAXVALUE)
);
INSERT INTO t1 VALUES (1, 'test1'),(101, 'test2'), (201, 'test3');
EXPLAIN SELECT * FROM t1 WHERE id BETWEEN 80 AND 120;
```

```
+--
    ↪ -------------------------+---------+-----------+-----------------------+-----------------------------
    ↪
| id                          | estRows | task      | access object         | operator info
    ↪                                   |
+--
    ↪ -------------------------+---------+-----------+-----------------------+-----------------------------
    ↪
| PartitionUnion_8            | 80.00   | root      |                       |
    ↪                                                 |
| ├─TableReader_10            | 40.00   | root      |                       | data:
    ↪ TableRangeScan_9                              |
| │ └─TableRangeScan_9        | 40.00   | cop[tikv] | table:t1, partition:p0 | range:[80,120],
    ↪ keep order:false, stats:pseudo |
| └─TableReader_12            | 40.00   | root      |                       | data:
    ↪ TableRangeScan_11                             |
|   └─TableRangeScan_11       | 40.00   | cop[tikv] | table:t1, partition:p1 | range:[80,120],
    ↪ keep order:false, stats:pseudo |
+--
    ↪ -------------------------+---------+-----------+-----------------------+-----------------------------
    ↪
5 rows in set (0.00 sec)
```

分区裁剪的使用场景

分区表有 Range 分区和 hash 分区两种形式，分区裁剪对两种分区表也有不同的使用场景。

分区裁剪在 Hash 分区表上的应用

Hash 分区表上可以使用分区裁剪的场景

只有等值比较的查询条件能够支持 Hash 分区表的裁剪。

```
create table t (x int) partition by hash(x) partitions 4;
explain select * from t where x = 1;
```

```
+--
    ↪ ----------------------+----------+-----------+----------------------+-------------------------------+
    ↪
| id                      | estRows  | task      | access object        | operator info
    ↪                  |
+--
    ↪ ----------------------+----------+-----------+----------------------+-------------------------------+
    ↪
| TableReader_8           | 10.00    | root      |                      | data:Selection_7
    ↪               |
| └─Selection_7           | 10.00    | cop[tikv] |                      | eq(test.t.x, 1)
    ↪                |
|   └─TableFullScan_6     | 10000.00 | cop[tikv] | table:t, partition:p1 | keep order:false,
    ↪ stats:pseudo |
+--
    ↪ ----------------------+----------+-----------+----------------------+-------------------------------+
    ↪
```

在这条 SQL 中，由条件 `x = 1` 可以知道所有结果均在一个分区上。数值 1 在经过 Hash 后，可以确定其在分区 p1 中。因此只需要扫描分区 p1 , 而无需访问一定不会出现相关结果的 p2 、p3 、p4 分区。从执行计划来看，其中只出现了一个 TableFullScan 算子，且在 access object 中指定了 p1 分区，确认 partition pruning 生效了。

Hash 分区表上不能使用分区裁剪的场景

场景一

不能确定查询结果只在一个分区上的条件：如 in, between, `>` `<` `>=` `<=` 等查询条件，不能使用分区裁剪的优化。

```
create table t (x int) partition by hash(x) partitions 4;
explain select * from t where x > 2;
```

```
+--
    ↪ ----------------------------+----------+-----------+----------------------+-------------------------------
    ↪
| id                            | estRows  | task      | access object        | operator info
    ↪                  |
+--
    ↪ ----------------------------+----------+-----------+----------------------+-------------------------------
    ↪
```

```
| Union_10                     | 13333.33 | root      |                       |
    ↪                                 |
| ├─TableReader_13             | 3333.33  | root      |                       | data:
    ↪ Selection_12               |
| │ └─Selection_12             | 3333.33  | cop[tikv] |                       | gt(test.t.x, 2)
    ↪                 |
| │   └─TableFullScan_11       | 10000.00 | cop[tikv] | table:t, partition:p0 | keep order:false
    ↪ , stats:pseudo |
| ├─TableReader_16             | 3333.33  | root      |                       | data:
    ↪ Selection_15               |
| │ └─Selection_15             | 3333.33  | cop[tikv] |                       | gt(test.t.x, 2)
    ↪                 |
| │   └─TableFullScan_14       | 10000.00 | cop[tikv] | table:t, partition:p1 | keep order:false
    ↪ , stats:pseudo |
| ├─TableReader_19             | 3333.33  | root      |                       | data:
    ↪ Selection_18               |
| │ └─Selection_18             | 3333.33  | cop[tikv] |                       | gt(test.t.x, 2)
    ↪                 |
| │   └─TableFullScan_17       | 10000.00 | cop[tikv] | table:t, partition:p2 | keep order:false
    ↪ , stats:pseudo |
| └─TableReader_22             | 3333.33  | root      |                       | data:
    ↪ Selection_21               |
|   └─Selection_21             | 3333.33  | cop[tikv] |                       | gt(test.t.x, 2)
    ↪                 |
|     └─TableFullScan_20       | 10000.00 | cop[tikv] | table:t, partition:p3 | keep order:false
    ↪ , stats:pseudo |
+--
    ↪ ----------------------------+----------+-----------+-----------------------+-------------------------
    ↪
```

在这条 SQL 中，`x > 2` 条件无法确定对应的 Hash Partition，所以不能使用分区裁剪。

场景二

由于分区裁剪的规则优化是在查询计划的生成阶段，对于执行阶段才能获取到过滤条件的场景，无法利用分区裁剪的优化。

```
create table t (x int) partition by hash(x) partitions 4;
explain select * from t2 where x = (select * from t1 where t2.x = t1.x and t2.x < 2);
```

```
+--
    ↪ ----------------------------------+----------+-----------+-----------------------+-----------------
    ↪
| id                                  | estRows  | task      | access object         | operator
    ↪  info                                |
+--
    ↪ ----------------------------------+----------+-----------+-----------------------+-----------------
```

```
    ↪
| Projection_13                         | 9990.00  | root      |                          | test.t2.
    ↪ x                                     |
| └─Apply_15                            | 9990.00  | root      |                          | inner
    ↪ join, equal:[eq(test.t2.x, test.t1.x)] |
|   ├─TableReader_18(Build)             | 9990.00  | root      |                          | data:
    ↪ Selection_17                          |
|   │ └─Selection_17                     | 9990.00  | cop[tikv] |                          | not(
    ↪ isnull(test.t2.x))                    |
|   │   └─TableFullScan_16               | 10000.00 | cop[tikv] | table:t2                 | keep
    ↪ order:false, stats:pseudo             |
|   └─Selection_19(Probe)               | 0.80     | root      |                          | not(
    ↪ isnull(test.t1.x))                    |
|     └─MaxOneRow_20                    | 1.00     | root      |                          |
    ↪                                       |
|       └─Union_21                      | 2.00     | root      |                          |
    ↪                                       |
|         ├─TableReader_24              | 2.00     | root      |                          | data:
    ↪ Selection_23                          |
|         │ └─Selection_23              | 2.00     | cop[tikv] |                          | eq(test
    ↪ .t2.x, test.t1.x), lt(test.t2.x, 2)   |
|         │   └─TableFullScan_22        | 2500.00  | cop[tikv] | table:t1, partition:p0   | keep
    ↪ order:false, stats:pseudo             |
|         └─TableReader_27              | 2.00     | root      |                          | data:
    ↪ Selection_26                          |
|           └─Selection_26              | 2.00     | cop[tikv] |                          | eq(test
    ↪ .t2.x, test.t1.x), lt(test.t2.x, 2)   |
|             └─TableFullScan_25        | 2500.00  | cop[tikv] | table:t1, partition:p1   | keep
    ↪ order:false, stats:pseudo             |
+--
    ↪ ------------------------------------+----------+-----------+--------------------------+-------------------
    ↪
```

这个查询每从 `t2` 读取一行，都会去分区表 `t1` 上进行查询，理论上这时会满足 `t1.x = val` 的过滤条件，但实际上由于分区裁剪只作用于查询计划生成阶段，而不是执行阶段，因而不会做裁剪。

分区裁剪在 Range 分区表上的应用

Range 分区表上可以使用分区裁剪的场景

场景一

等值比较的查询条件可以使用分区裁剪。

```
create table t (x int) partition by range (x) (
    partition p0 values less than (5),
    partition p1 values less than (10),
    partition p2 values less than (15)
```

```
    );
explain select * from t where x = 3;
```

```
+--
    ↪ ----------------------+----------+-----------+----------------------+------------------------------+
    ↪
| id                      | estRows  | task      | access object        | operator info
    ↪                  |
+--
    ↪ ----------------------+----------+-----------+----------------------+------------------------------+
    ↪
| TableReader_8           | 10.00    | root      |                      | data:Selection_7
    ↪               |
| └─Selection_7           | 10.00    | cop[tikv] |                      | eq(test.t.x, 3)
    ↪                |
|   └─TableFullScan_6     | 10000.00 | cop[tikv] | table:t, partition:p0 | keep order:false,
    ↪ stats:pseudo |
+--
    ↪ ----------------------+----------+-----------+----------------------+------------------------------+
    ↪
```

使用 in 条件的等值比较查询条件也可以使用分区裁剪。

```
create table t (x int) partition by range (x) (
    partition p0 values less than (5),
    partition p1 values less than (10),
    partition p2 values less than (15)
    );
explain select * from t where x in(1,13);
```

```
+--
    ↪ -------------------------+----------+-----------+----------------------+------------------------------
    ↪
| id                         | estRows  | task      | access object        | operator info
    ↪                  |
+--
    ↪ -------------------------+----------+-----------+----------------------+------------------------------
    ↪
| Union_8                    | 40.00    | root      |                      |
    ↪                             |
| ├─TableReader_11           | 20.00    | root      |                      | data:Selection_10
    ↪               |
| │ └─Selection_10           | 20.00    | cop[tikv] |                      | in(test.t.x, 1,
    ↪ 13)           |
| │   └─TableFullScan_9      | 10000.00 | cop[tikv] | table:t, partition:p0 | keep order:false,
    ↪  stats:pseudo |
```

```
| └─TableReader_14            | 20.00    | root      |                       | data:Selection_13
    ↪                |
|   └─Selection_13            | 20.00    | cop[tikv] |                       | in(test.t.x, 1,
    ↪ 13)            |
|     └─TableFullScan_12      | 10000.00 | cop[tikv] | table:t, partition:p2 | keep order:false,
    ↪  stats:pseudo |
+--
    ↪ -------------------------+----------+-----------+-----------------------+-------------------------------
    ↪
```

在这条 SQL 中，由条件 `x in(1,13)` 可以知道所有结果只会分布在几个分区上。经过分析，发现所有 `x = 1` 的记录都在分区 `p0` 上, 所有 `x = 13` 的记录都在分区 `p2` 上，因此只需要访问 `p0`、`p2` 这两个分区，

场景二

区间比较的查询条件如 `between, > < = >= <=` 可以使用分区裁剪。

```
create table t (x int) partition by range (x) (
    partition p0 values less than (5),
    partition p1 values less than (10),
    partition p2 values less than (15)
    );
explain select * from t where x between 7 and 14;
```

```
+--
    ↪ -------------------------+----------+-----------+-----------------------+-------------------------------
    ↪
| id                          | estRows  | task      | access object         | operator info
    ↪                     |
+--
    ↪ -------------------------+----------+-----------+-----------------------+-------------------------------
    ↪
| Union_8                     | 500.00   | root      |                       |
    ↪                                 |
| ├─TableReader_11             | 250.00   | root      |                       | data:Selection_10
    ↪                  |
| │ └─Selection_10             | 250.00   | cop[tikv] |                       | ge(test.t.x, 7),
    ↪ le(test.t.x, 14) |
| │   └─TableFullScan_9        | 10000.00 | cop[tikv] | table:t, partition:p1 | keep order:false,
    ↪  stats:pseudo    |
| └─TableReader_14             | 250.00   | root      |                       | data:Selection_13
    ↪                  |
|   └─Selection_13             | 250.00   | cop[tikv] |                       | ge(test.t.x, 7),
    ↪ le(test.t.x, 14) |
|     └─TableFullScan_12       | 10000.00 | cop[tikv] | table:t, partition:p2 | keep order:false,
    ↪  stats:pseudo    |
```

```
+--
    ↪ -------------------------+----------+-----------+----------------------+-------------------------------
    ↪
```

场景三

分区表达式为 fn(col) 的简单形式，查询条件是 > < = >= <= 之一，且 fn 是单调函数，可以使用分区裁剪。

关于 fn 函数，对于任意 x y，如果 x > y，则 fn(x)> fn(y)，那么这种是严格递增的单调函数。非严格递增的单调函数也可以符合分区裁剪要求，只要函数 fn 满足：对于任意 x y，如果 x > y，则 fn(x)>= fn(y)。理论上，所有满足单调条件（严格或者非严格）的函数都支持分区裁剪。目前，TiDB 支持的单调函数如下：

```
unix_timestamp
to_days
```

例如，分区表达式是 fn(col) 形式，fn 为我们支持的单调函数 to_days，就可以使用分区裁剪：

```
create table t (id datetime) partition by range (to_days(id)) (
    partition p0 values less than (to_days('2020-04-01')),
    partition p1 values less than (to_days('2020-05-01')));
explain select * from t where id > '2020-04-18';
```

```
+--
    ↪ ----------------------+----------+-----------+----------------------+-------------------------------
    ↪
| id                      | estRows  | task      | access object        | operator info
    ↪                             |
+--
    ↪ ----------------------+----------+-----------+----------------------+-------------------------------
    ↪
| TableReader_8           | 3333.33  | root      |                      | data:Selection_7
    ↪                           |
| └─Selection_7           | 3333.33  | cop[tikv] |                      | gt(test.t.id,
    ↪ 2020-04-18 00:00:00.000000) |
|   └─TableFullScan_6     | 10000.00 | cop[tikv] | table:t, partition:p1 | keep order:false,
    ↪ stats:pseudo            |
+--
    ↪ ----------------------+----------+-----------+----------------------+-------------------------------
    ↪
```

Range 分区表上不能使用分区裁剪的场景

由于分区裁剪的规则优化是在查询计划的生成阶段，对于执行阶段才能获取到过滤条件的场景，无法利用分区裁剪的优化。

```
create table t1 (x int) partition by range (x) (
    partition p0 values less than (5),
    partition p1 values less than (10));
```

```
create table t2 (x int);
explain select * from t2 where x < (select * from t1 where t2.x < t1.x and t2.x < 2);
```

```
+--
    ↪ ----------------------------------+----------+-----------+-----------------------+------------------
    ↪
| id                                  | estRows  | task      | access object         | operator
    ↪  info                                          |
+--
    ↪ ----------------------------------+----------+-----------+-----------------------+------------------
    ↪
| Projection_13                       | 9990.00  | root      |                       | test.t2.
    ↪ x                                              |
| └─Apply_15                          | 9990.00  | root      |                       |
    ↪ CARTESIAN inner join, other cond:lt(test.t2.x, test.t1.x) |
|   ├─TableReader_18(Build)           | 9990.00  | root      |                       | data:
    ↪ Selection_17                                   |
|   │ └─Selection_17                  | 9990.00  | cop[tikv] |                       | not(
    ↪ isnull(test.t2.x))                             |
|   │   └─TableFullScan_16            | 10000.00 | cop[tikv] | table:t2              | keep
    ↪ order:false, stats:pseudo                      |
|   └─Selection_19(Probe)             | 0.80     | root      |                       | not(
    ↪ isnull(test.t1.x))                             |
|     └─MaxOneRow_20                  | 1.00     | root      |                       |
    ↪                                                |
|       └─Union_21                    | 2.00     | root      |                       |
    ↪                                                |
|         ├─TableReader_24            | 2.00     | root      |                       | data:
    ↪ Selection_23                                   |
|         │ └─Selection_23            | 2.00     | cop[tikv] |                       | lt(test
    ↪ .t2.x, 2), lt(test.t2.x, test.t1.x)            |
|         │   └─TableFullScan_22      | 2.50     | cop[tikv] | table:t1, partition:p0 | keep
    ↪ order:false, stats:pseudo                      |
|         └─TableReader_27            | 2.00     | root      |                       | data:
    ↪ Selection_26                                   |
|           └─Selection_26            | 2.00     | cop[tikv] |                       | lt(test
    ↪ .t2.x, 2), lt(test.t2.x, test.t1.x)            |
|             └─TableFullScan_25      | 2.50     | cop[tikv] | table:t1, partition:p1 | keep
    ↪ order:false, stats:pseudo                      |
+--
    ↪ ----------------------------------+----------+-----------+-----------------------+------------------
    ↪
14 rows in set (0.00 sec)
```

这个查询每从 `t2` 读取一行，都会去分区表 `t1` 上进行查询，理论上这时会满足 `t1.x > val` 的过滤条件，但实

际上由于分区裁剪只作用于查询计划生成阶段，而不是执行阶段，因而不会做裁剪。

9.3.3.2.8 TopN 和 Limit 下推

SQL 中的 LIMIT 子句在 TiDB 查询计划树中对应 Limit 算子节点，ORDER BY 子句在查询计划树中对应 Sort 算子节点，此外，我们会将相邻的 Limit 和 Sort 算子组合成 TopN 算子节点，表示按某个排序规则提取记录的前 N 项。从另一方面来说，Limit 节点等价于一个排序规则为空的 TopN 节点。

和谓词下推类似，TopN（及 Limit，下同）下推将查询计划树中的 TopN 计算尽可能下推到距离数据源最近的地方，以尽早完成数据的过滤，进而显著地减少数据传输或计算的开销。

如果要关闭这个规则，可参照优化规则及表达式下推的黑名单中的关闭方法。

示例

以下通过一些例子对 TopN 下推进行说明。

示例 1：下推到存储层 Coprocessor

```
create table t(id int primary key, a int not null);
explain select * from t order by a limit 10;
```

```
+---------------------------+----------+-----------+---------------+--------------------------------+
    ↪
| id                        | estRows  | task      | access object | operator info
    ↪                   |
+---------------------------+----------+-----------+---------------+--------------------------------+
    ↪
| TopN_7                    | 10.00    | root      |               | test.t.a, offset:0, count
    ↪ :10   |
| └─TableReader_15           | 10.00    | root      |               | data:TopN_14
    ↪                    |
|   └─TopN_14                | 10.00    | cop[tikv] |               | test.t.a, offset:0, count
    ↪ :10   |
|     └─TableFullScan_13     | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo |
+---------------------------+----------+-----------+---------------+--------------------------------+
    ↪
4 rows in set (0.00 sec)
```

在该查询中，将 TopN 算子节点下推到 TiKV 上对数据进行过滤，每个 Coprocessor 只向 TiDB 传输 10 条记录。在 TiDB 将数据整合后，再进行最终的过滤。

示例 2：TopN 下推过 Join 的情况（排序规则仅依赖于外表中的列）

```
create table t(id int primary key, a int not null);
create table s(id int primary key, a int not null);
explain select * from t left join s on t.a = s.a order by t.a limit 10;
```

```
+---------------------------------+----------+-----------+--------------+--------------------------------
    ↪
| id                              | estRows  | task      | access object | operator info
    ↪                                    |
+---------------------------------+----------+-----------+--------------+--------------------------------
    ↪
| TopN_12                         | 10.00    | root      |               | test.t.a, offset:0,
    ↪ count:10                     |
| └─HashJoin_17                    | 12.50    | root      |               | left outer join,
    ↪ equal:[eq(test.t.a, test.s.a)] |
|   ├─TopN_18(Build)               | 10.00    | root      |               | test.t.a, offset:0,
    ↪ count:10                     |
|   │ └─TableReader_26             | 10.00    | root      |               | data:TopN_25
    ↪                                      |
|   │   └─TopN_25                  | 10.00    | cop[tikv] |               | test.t.a, offset:0,
    ↪ count:10                     |
|   │     └─TableFullScan_24       | 10000.00 | cop[tikv] | table:t       | keep order:false,
    ↪ stats:pseudo                   |
|   └─TableReader_30(Probe)        | 10000.00 | root      |               | data:
    ↪ TableFullScan_29                            |
|     └─TableFullScan_29           | 10000.00 | cop[tikv] | table:s       | keep order:false,
    ↪ stats:pseudo                   |
+---------------------------------+----------+-----------+--------------+--------------------------------
    ↪
8 rows in set (0.01 sec)
```

在该查询中，TopN 算子的排序规则仅依赖于外表 t 中的列，可以将 TopN 下推到 Join 之前进行一次计算，以减少 Join 时的计算开销。除此之外，TiDB 同样将 TopN 下推到了存储层中。

示例 3：TopN 不能下推过 Join 的情况

```
create table t(id int primary key, a int not null);
create table s(id int primary key, a int not null);
explain select * from t join s on t.a = s.a order by t.id limit 10;
```

```
+-------------------------------+----------+-----------+--------------+--------------------------------
    ↪
| id                            | estRows  | task      | access object | operator info
    ↪                               |
+-------------------------------+----------+-----------+--------------+--------------------------------
    ↪
| TopN_12                       | 10.00    | root      |               | test.t.id, offset:0,
    ↪ count:10              |
| └─HashJoin_16                  | 12500.00 | root      |               | inner join, equal:[eq(
    ↪ test.t.a, test.s.a)] |
```

```
|   ├─TableReader_21(Build)     | 10000.00 | root      |               | data:TableFullScan_20
    ↪                       |
|   |  └─TableFullScan_20       | 10000.00 | cop[tikv] | table:s       | keep order:false, stats
    ↪ :pseudo             |
|   └─TableReader_19(Probe)     | 10000.00 | root      |               | data:TableFullScan_18
    ↪                       |
|     └─TableFullScan_18        | 10000.00 | cop[tikv] | table:t       | keep order:false, stats
    ↪ :pseudo             |
+-------------------------------+----------+-----------+---------------+-----------------------------------
    ↪
6 rows in set (0.00 sec)
```

TopN 无法下推过 Inner Join。以上面的查询为例，如果先 Join 得到 100 条记录，再做 TopN 可以剩余 10 条记录。而如果在 TopN 之前就过滤到剩余 10 条记录，做完 Join 之后可能就剩下 5 条了，导致了结果的差异。

同理，TopN 无法下推到 Outer Join 的内表上。在 TopN 的排序规则涉及多张表上的列时，也无法下推，如 `t.a+s.a`。只有当 TopN 的排序规则仅依赖于外表上的列时，才可以下推。

示例 4：TopN 转换成 Limit 的情况

```
create table t(id int primary key, a int not null);
create table s(id int primary key, a int not null);
explain select * from t left join s on t.a = s.a order by t.id limit 10;
```

```
+---------------------------------+----------+-----------+---------------+-----------------------------------
    ↪
| id                              | estRows  | task      | access object | operator info
    ↪                                    |
+---------------------------------+----------+-----------+---------------+-----------------------------------
    ↪
| TopN_12                         | 10.00    | root      |               | test.t.id, offset:0,
    ↪ count:10                   |
| └─HashJoin_17                    | 12.50    | root      |               | left outer join,
    ↪ equal:[eq(test.t.a, test.s.a)] |
|   ├─Limit_21(Build)              | 10.00    | root      |               | offset:0, count:10
    ↪                               |
|   |  └─TableReader_31            | 10.00    | root      |               | data:Limit_30
    ↪                                  |
|   |    └─Limit_30                | 10.00    | cop[tikv] |               | offset:0, count:10
    ↪                               |
|   |      └─TableFullScan_29      | 10.00    | cop[tikv] | table:t       | keep order:true,
    ↪ stats:pseudo                    |
|   └─TableReader_35(Probe)        | 10000.00 | root      |               | data:
    ↪ TableFullScan_34                          |
|     └─TableFullScan_34           | 10000.00 | cop[tikv] | table:s       | keep order:false,
    ↪ stats:pseudo                   |
```

```
+---------------------------------+----------+----------+--------------+-------------------------------------------------
    ↪
8 rows in set (0.00 sec)
```

在上面的查询中，TopN 首先推到了外表 t 上。然后因为它要对 t.id 进行排序，而 t.id 是表 t 的主键，可以直接按顺序读出 (keep order:true)，从而省略了 TopN 中的排序，将其简化为 Limit。

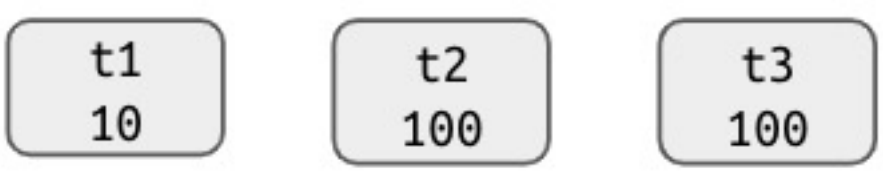
9.3.3.2.9 Join Reorder 算法简介

在实际的业务场景中，多个表的 Join 语句是很常见的，而 Join 的执行效率和各个表参与 Join 的顺序有关系。如 select * from t1, t2, t3 where t1.a=t2.a and t3.a=t2.a，这个 SQL 中可能的执行顺序有“t1 和 t2 先做 Join，然后再和 t3 做 Join”以及“t2 和 t3 先做 Join，然后再和 t1 做 Join”两种情况。根据 t1 和 t3 的数据量及数据分布，这两种执行顺序会有不同的性能表现。

因此优化器需要实现一种决定 Join 顺序的算法。目前 TiDB 中使用的算法是 Join Reorder 算法，又称贪心算法。

Join Reorder 算法实例

以三个表 t1、t2、t3 的 Join 为例。
序。



之后选定其中最小
一对。

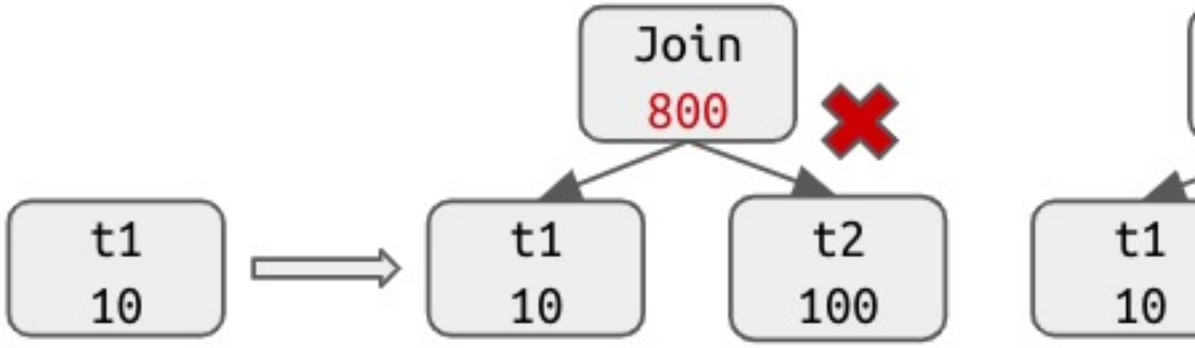


图 82: join-reorder-2

然后进入下一轮的选择，如果这时是四个表，那么就继续比较输出结果集的大小，进行选择。这里只有三个表，因此就直接得到了最终的 Join 结果。

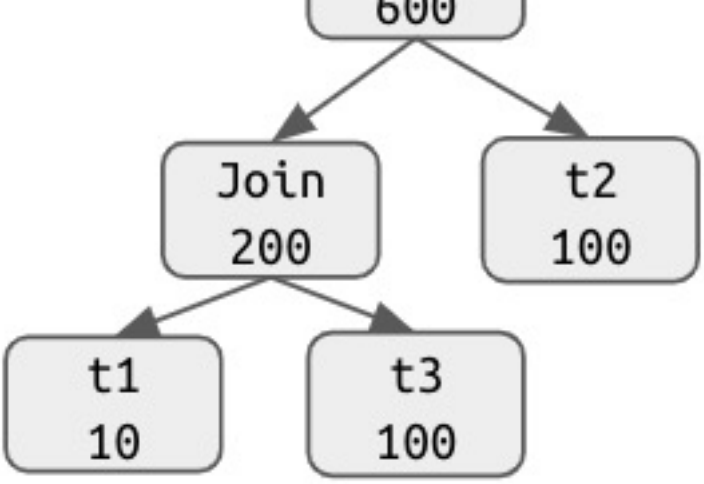


图 83: join-reorder-3

以上就是当前 TiDB 中使用的 Join reorder 算法。

Join reorder 算法限制

当前的 Join Reorder 算法存在如下限制

- 目前并不支持 Outer Join 的 Join Reorder
- 受结果集的计算算法所限并不会保证一定会选到合适的 Join order

目前 TiDB 中支持使用 STRAIGHT_JOIN 语法来强制指定一种 Join 顺序，参见语法元素说明。

9.3.3.3 物理优化

9.3.3.3.1 物理优化

物理优化是基于代价的优化，为上一阶段产生的逻辑执行计划制定物理执行计划。这一阶段中，优化器会为逻辑执行计划中的每个算子选择具体的物理实现。逻辑算子的不同物理实现有着不同的时间复杂度、资源消耗和物理属性等。在这个过程中，优化器会根据数据的统计信息来确定不同物理实现的代价，并选择整体代价最小的物理执行计划。

理解 TiDB 执行计划 文档已对每个物理算子进行了一些介绍。在本章我们会重点介绍以下方面：

- 在索引的选择中会介绍 TiDB 在一张表有多个索引时，如何选择最优的索引进行表的访问。
- 在统计信息简介中会介绍 TiDB 收集了哪些统计信息来获得表的数据分布情况
- 在错误索引的解决方案中会介绍当发现 TiDB 索引选错时，你应该使用那些手段来让它使用正确的索引
- 在Distinct 优化中会介绍在物理优化中会做的一个有关 DISTINCT 关键字的优化，在这一小节中会介绍它的优缺点以及如何使用它。

9.3.3.3.2 索引的选择

从存储层读取数据是 SQL 计算过程中最为耗时的部分之一，TiDB 目前支持从不同的存储和不同的索引中读取数据，索引选择得是否合理将很大程度上决定一个查询的运行速度。

本章节将介绍 TiDB 如何选择索引去读入数据，以及相关的一些控制索引选择的方式。

读表

在介绍索引的选择之前，首先要了解 TiDB 有哪些读表的方式，这些方式的触发条件是什么，不同方式有什么区别，各有什么优劣。

读表算子

读表算子	触发条件	适用场景	说明
PointGet/BatchPoint	读表的范围是一个或多个单点范围	任何场景	如果能被触发，通常被认为是最快的算子，因为其直接调用 kvget 的接口进行计算，不走 coprocessor
TableReader	无	任何场景	从 TiKV 端直接扫描表数据，一般被认为是效率最低的算子，除非在 _tidb_rowid ↪ 这一列上存在范围查询，或者无其他可以选择的读表算子时，才会选择这个算子

读表算子	触发条件	适用场景	说明
TableReader	表在 TiFlash 节点上存在副本	需要读取的列比较少，但是需要计算的行很多	TiFlash 是列式存储，如果需要对少量的列和大量的行进行计算，一般会选择这个算子
IndexReader	表有一个或多个索引，且计算所需的列被包含在索引里	存在较小的索引上的范围查询，或者对索引列有顺序需求的时候	当存在多个索引的时候，会根据估算代价选择合理的索引
IndexLookupReader	表有一个或多个索引，且计算所需的列不完全被包含在索引里	同 IndexReader	因为计算列不完全被包含在索引里，所以读完索引后需要回表，这里会比 IndexReader 多一些开销

注意：

TableReader 是基于 `_tidb_rowid` 的索引，TiFlash 是列存索引，所以索引的选择即是读表算子的选择。

索引的选择

TiDB 在选择索引时，会基于每个读表算子的代价估算，在此基础上提供了启发式规则“Skyline-Pruning”，以降低错误估算导致选错索引的概率。

Skyline-Pruning

Skyline-Pruning 是一个针对索引的启发式过滤规则，评判一个索引的好坏需要从以下三个维度进行衡量：

- 选择该索引读表时，是否需要回表（即该索引生成的计划是 IndexReader 还是 IndexLookupReader）。不用回表的索引在这个维度上优于需要回表的索引。
- 选择该索引是否能满足一定的顺序。因为索引的读取可以保证某些列集合的顺序，所以满足查询要求顺序的索引在这个维度上优于不满足的索引。
- 索引的列涵盖了多少访问条件。“访问条件”指的是可以转化为某列范围的 `where` 条件，如果某个索引的列集合涵盖的访问条件越多，那么它在这个维度上更优。

对于这三种维度，如果某个索引 `idx_a` 在三个维度上都不比 `idx_b` 差，且有一个维度比 `idx_b` 好，那么就会优先选择 `idx_a`。

基于代价选择

在使用 Skyline-Pruning 规则排除了不合适的索引之后，索引的选择完全基于代价估算，读表的代价估算需要考虑以下几个方面：

- 索引的每行数据在存储层的平均长度。
- 索引生成的查询范围的行数量。
- 索引的回表代价。
- 索引查询时的范围数量。

根据这些因子和代价模型，优化器会选择一个代价最低的索引进行读表。

代价选择调优的常见问题

1. 估算的行数量不准确？

   一般是统计信息过期或者准确度不够造成的，可以重新执行 `analyze table` 或者修改 `analyze table` 的参数。

2. 统计信息准确，为什么读 TiFlash 更快，而优化器选择了 TiKV？

   目前区别 TiFlash 和 TiKV 的代价模型还比较粗糙，可以调小 `tidb_opt_seek_factor` 的值，让优化器倾向于选择 TiFlash。

3. 统计信息准确，某个索引要回表，但是它比另一个不用回表的索引实际执行更快，为什么选择了不用回表的索引？

   碰到这种情况，可能是代价估算时对于回表的代价计算得过大，可以调小 `tidb_opt_network_factor`，降低回表的代价。

控制索引的选择

通过Optimizer Hints 可以实现单条查询对索引选择的控制。

- USE_INDEX/IGNORE_INDEX 可以强制优化器使用/不使用某些索引。
- READ_FROM_STORAGE 可以强制优化器对于某些表选择 TiKV/TiFlash 的存储引擎进行查询。

9.3.3.3.3 统计信息简介

在索引的选择章节中，提到了 TiDB 会使用统计信息来决定选择哪个索引。在 TiDB 中，我们维护的统计信息包括表的总行数，列的等深直方图，Count-Min Sketch，Null 值的个数，平均长度，不同值的数目等等。本文将简单介绍直方图和 Count-Min Sketch，以及详细介绍统计信息的收集和维护。

直方图简介

直方图是一种对数据分布情况进行描述的工具，它会按照数据的值大小进行分桶，并用一些简单的数据来描述每个桶，比如落在桶里的值的个数。在 TiDB 中，会对每个表具体的列构建一个等深直方图，区间查询的估算便是借助该直方图来进行。

等深直方图，就是让落入每个桶里的值数量尽量相等。举个例子，比方说对于给定的集合 {1.6, 1.9, 1.9, 2.0, 2.4, 2.6, 2.7, 2.7, 2.8, 2.9, 3.4, 3.5}，并且生成 4 个桶，那么最终的等深直方图就会如下图所示，包含四个桶 [1.6, 1.9]，[2.0, 2.6]，[2.7, 2.8]，[2.9, 3.5]，其桶深均为 3。

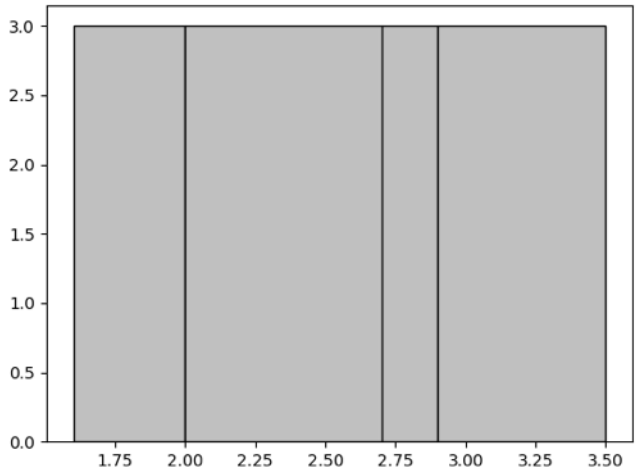


图 84: 等深直方图示例

在手动收集统计信息一节中有控制直方图桶数量上限的参数。当桶数量越多，直方图的估算精度就越高，不过也会同时增大统计信息的内存使用，可以视具体情况来做调整。

Count-Min Sketch

Count-Min Sketch 是一种哈希结构，当查询中出现诸如 `a = 1` 或者 IN 查询（如 `a in (1, 2, 3)`）这样的等值查询时，TiDB 便会使用这个数据结构来进行估算。

由于 Count-Min Sketch 是一个哈希结构，就有出现哈希碰撞的可能。当在 EXPLAIN 语句中发现等值查询的估算偏离实际值较大时，就可以认为是一个比较大的值和一个比较小的值被哈希到了一起。这时有以下两种手段来避免这个情况：

- 修改手动收集统计信息中提到的 WITH NUM TOPN 参数。TiDB 会将出现频率前 x 大的数据单独储存，之后的数据再储存到 Count-Min Sketch 中。因此可以调大这个值来避免一个比较大的值和一个比较小的值被哈希到一起。在 TiDB 中，这个参数的默认值是 20，最大可以设置为 1024。
- 修改统计信息的收集 -手动收集中提到的 WITH NUM CMSKETCH DEPTH 和 WITH NUM CMSKETCH WIDTH 两个参数，这两个参数会影响哈希的桶数和碰撞概率，可是适当调大来减少冲突概率，同时它会影响统计信息的内存使用，可以视具体情况来调整。在 TiDB 中，DEPTH 的默认值是 5，WIDTH 的默认值是 2048。

统计信息的收集

手动收集

可以通过执行 ANALYZE 语句来收集统计信息。

> 注意：
>
> 在 TiDB 中执行 ANALYZE TABLE 语句比在 MySQL 或 InnoDB 中耗时更长。InnoDB 采样的只是少量页面，但 TiDB 会完全重构一系列统计信息。适用于 MySQL 的脚本会误以为执行 ANALYZE TABLE 耗时较短。
>
> 如需更快的分析速度，可将 tidb_enable_fast_analyze 设置为 1 来打开快速分析功能。该参数的默认值为 0。
>
> 快速分析功能开启后，TiDB 会随机采样约 10000 行的数据来构建统计信息。因此在数据分布不均匀或者数据量比较少的情况下，统计信息的准确度会比较差。可能导致执行计划不优，比如选错索引。如果可以接受普通 ANALYZE 语句的执行时间，则推荐关闭快速分析功能。

全量收集

可以通过以下几种语法进行全量收集。

收集 TableNameList 中所有表的统计信息：

```
ANALYZE TABLE TableNameList [WITH NUM BUCKETS|TOPN|CMSKETCH DEPTH|CMSKETCH WIDTH|SAMPLES];
```

- WITH NUM BUCKETS 用于指定生成直方图的桶数量上限。
- WITH NUM TOPN 用于指定生成 TOPN 数目的上限。
- WITH NUM CMSKETCH DEPTH 用于指定 CM Sketch 的长。
- WITH NUM CMSKETCH WIDTH 用于指定 CM Sketch 的宽。
- WITH NUM SAMPLES 用于指定采样的数目。

收集 TableName 中所有的 IndexNameList 中的索引列的统计信息：

```
ANALYZE TABLE TableName INDEX [IndexNameList] [WITH NUM BUCKETS|TOPN|CMSKETCH DEPTH|CMSKETCH
    ↪ WIDTH|SAMPLES];
```

IndexNameList 为空时会收集所有索引列的统计信息。

收集 TableName 中所有的 PartitionNameList 中分区的统计信息：

```
ANALYZE TABLE TableName PARTITION PartitionNameList [WITH NUM BUCKETS|TOPN|CMSKETCH DEPTH|
    ↪ CMSKETCH WIDTH|SAMPLES];
```

收集 TableName 中所有的 PartitionNameList 中分区的索引列统计信息：

```
ANALYZE TABLE TableName PARTITION PartitionNameList INDEX [IndexNameList] [WITH NUM BUCKETS|TOPN|
    ↪ CMSKETCH DEPTH|CMSKETCH WIDTH|SAMPLES];
```

增量收集

对于类似时间列这样的单调不减列，在进行全量收集后，可以使用增量收集来单独分析新增的部分，以提高分析的速度。

> 注意：
>
> 1. 目前只有索引提供了增量收集的功能
> 2. 使用增量收集时，必须保证表上只有插入操作，且应用方需要保证索引列上新插入的值是单调不减的，否则会导致统计信息不准，影响 TiDB 优化器选择合适的执行计划

可以通过以下几种语法进行增量收集。

增量收集 TableName 中所有的 IndexNameList 中的索引列的统计信息：

```
ANALYZE INCREMENTAL TABLE TableName INDEX [IndexNameList] [WITH NUM BUCKETS|TOPN|CMSKETCH DEPTH|
    ↪ CMSKETCH WIDTH|SAMPLES];
```

增量收集 TableName 中所有的 PartitionNameList 中分区的索引列统计信息：

```
ANALYZE INCREMENTAL TABLE TableName PARTITION PartitionNameList INDEX [IndexNameList] [WITH NUM
    ↪ BUCKETS|TOPN|CMSKETCH DEPTH|CMSKETCH WIDTH|SAMPLES];
```

自动更新

在发生增加，删除以及修改语句时，TiDB 会自动更新表的总行数以及修改的行数。这些信息会定期持久化下来，更新的周期是 20 * stats-lease，stats-lease 的默认值是 3s，如果将其指定为 0，那么将不会自动更新。

和统计信息自动更新相关的三个系统变量如下：

系统变量名	默认值	功能
tidb_auto_analyze_ratio	0.5	自动更新阈值
tidb_auto_analyze_start_time	00:00 +0000	一天中能够进行自动更新的开始时间
tidb_auto_analyze_end_time	23:59 +0000	一天中能够进行自动更新的结束时间

当某个表 tbl 的修改行数与总行数的比值大于 tidb_auto_analyze_ratio，并且当前时间在 tidb_auto_analyze_start_time
↪ 和 tidb_auto_analyze_end_time 之间时，TiDB 会在后台执行 ANALYZE TABLE tbl 语句自动更新这个表的统

计信息。

在 v4.0.9 版本之前，执行查询语句时，TiDB 会以`feedback-probability`的概率收集反馈信息，并将其用于更新直方图和 Count-Min Sketch。从 v4.0.9 版本开始，该功能默认关闭，暂不建议开启此功能。

控制 ANALYZE 并发度

执行 ANALYZE 语句的时候，你可以通过一些参数来调整并发度，以控制对系统的影响。

tidb_build_stats_concurrency

目前 ANALYZE 执行的时候会被切分成一个个小的任务，每个任务只负责某一个列或者索引。`tidb_build_stats_concurrency` 可以控制同时执行的任务的数量，其默认值是 4。

tidb_distsql_scan_concurrency

在执行分析普通列任务的时候，`tidb_distsql_scan_concurrency` 可以用于控制一次读取的 Region 数量，其默认值是 15。

tidb_index_serial_scan_concurrency

在执行分析索引列任务的时候，`tidb_index_serial_scan_concurrency` 可以用于控制一次读取的 Region 数量，其默认值是 1。

查看 ANALYZE 状态

在执行 ANALYZE 时，可以通过 SQL 语句来查看当前 ANALYZE 的状态。

语法如下：

```
SHOW ANALYZE STATUS [ShowLikeOrWhere];
```

该语句会输出 ANALYZE 的状态，可以通过使用 ShowLikeOrWhere 来筛选需要的信息。

目前 `SHOW ANALYZE STATUS` 会输出 7 列，具体如下：

语法元素	说明
table_schema	数据库名
table_name	表名
partition_name	分区名
job_info	任务具体信息。如果分析索引则会包含索引名
row_count	已经分析的行数
start_time	任务开始执行的时间
state	任务状态，包括 pending（等待）、running（正在执行）、finished（执行成功）和 failed（执行失败）

统计信息的查看

你可以通过一些语句来查看统计信息的状态。

表的元信息

你可以通过 `SHOW STATS_META` 来查看表的总行数以及修改的行数等信息。

语法如下：

其中，`ShowLikeOrWhereOpt` 部分的语法图为：

```
SHOW STATS_META [ShowLikeOrWhere];
```

目前 SHOW STATS_META 会输出 6 列，具体如下：

语法元素	说明
db_name	数据库名
table_name	表名
partition_name	分区名
update_time	更新时间
modify_count	修改的行数
row_count	总行数

> 注意：
>
> 在 TiDB 根据 DML 语句自动更新总行数以及修改的行数时，update_time 也会被更新，因此并不能认为 update_time 是最近一次发生 Analyze 的时间。

表的健康度信息

通过 SHOW STATS_HEALTHY 可以查看表的统计信息健康度，并粗略估计表上统计信息的准确度。当 modify_count >= row_count 时，健康度为 0；当 modify_count < row_count 时，健康度为 (1 - modify_count/row_count) * 100。

SHOW STATS_HEALTHY 的语法图为：

图 85: ShowStatsHealthy

其中，ShowLikeOrWhereOpt 部分的语法图为：

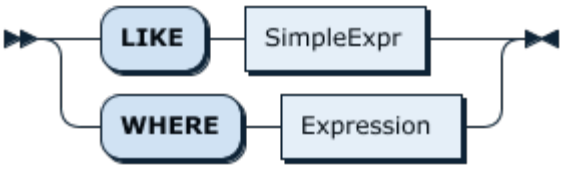


图 86: ShowLikeOrWhereOpt

目前，SHOW STATS_HEALTHY 会输出 4 列，具体如下：

语法元素	说明
db_name	数据库名

语法元素	说明
table_name	表名
partition_name	分区名
healthy	健康度

列的元信息

你可以通过 SHOW STATS_HISTOGRAMS 来查看列的不同值数量以及 NULL 数量等信息。

语法如下：

```
SHOW STATS_HISTOGRAMS [ShowLikeOrWhere];
```

该语句会输出所有列的不同值数量以及 NULL 数量等信息，你可以通过 ShowLikeOrWhere 来筛选需要的信息。

目前 SHOW STATS_HISTOGRAMS 会输出 8 列，具体如下：

语法元素	说明
db_name	数据库名
table_name	表名
partition_name	分区名
column_name	根据 is_index 来变化：is_index 为 0 时是列名，为 1 时是索引名
is_index	是否是索引列
update_time	更新时间
distinct_count	不同值数量
null_count	NULL 的数量
avg_col_size	列平均长度

直方图桶的信息

你可以通过 SHOW STATS_BUCKETS 来查看直方图每个桶的信息。

语法如下：

```
SHOW STATS_BUCKETS [ShowLikeOrWhere];
```

语法图：

SHOW STATUS_BUCKETS:

图 87: SHOW STATS_BUCKETS

该语句会输出所有桶的信息，你可以通过 ShowLikeOrWhere 来筛选需要的信息。

目前 SHOW STATS_BUCKETS 会输出 10 列，具体如下：

语法元素	说明
db_name	数据库名
table_name	表名
partition_name	分区名
column_name	根据 is_index 来变化：is_index 为 0 时是列名，为 1 时是索引名
is_index	是否是索引列
bucket_id	桶的编号
count	所有落在这个桶及之前桶中值的数量
repeats	最大值出现的次数
lower_bound	最小值
upper_bound	最大值

删除统计信息

可以通过执行 DROP STATS 语句来删除统计信息。

语法如下：

```
DROP STATS TableName;
```

该语句会删除 TableName 中所有的统计信息。

统计信息的导入导出

导出统计信息

统计信息的导出接口如下。

通过以下接口可以获取数据库 ${db_name} 中的表 ${table_name} 的 json 格式的统计信息：

```
http://${tidb-server-ip}:${tidb-server-status-port}/stats/dump/${db_name}/${table_name}
```

通过以下接口可以获取数据库 ${db_name} 中的表 ${table_name} 在指定时间上的 json 格式统计信息。指定的时间应在 GC SafePoint 之后。

```
http://${tidb-server-ip}:${tidb-server-status-port}/stats/dump/${db_name}/${table_name}/${
    ↪ yyyyMMddHHmmss}
```

通过以下接口可以获取数据库 ${db_name} 中的表 ${table_name} 在指定时间上的 json 格式统计信息。指定的时间应在 GC SafePoint 之后。

```
http://${tidb-server-ip}:${tidb-server-status-port}/stats/dump/${db_name}/${table_name}/${yyyy-MM
    ↪ -dd HH:mm:ss}
```

导入统计信息

导入的统计信息一般是通过统计信息导出接口得到的 json 文件。

语法如下：

```
LOAD STATS 'file_name';
```

`file_name` 为要导入的统计信息的文件名。

另请参阅

- DROP STATS

9.3.3.3.4 错误索引的解决方案

在观察到某个查询的执行速度达不到预期时，可能是它的索引使用有误，这时就需要通过一些手段来解决。通常可以先使用表的健康度信息来查看统计信息的健康度。根据健康度可以分为以下两种情况处理。

健康度较低

这意味着距离 TiDB 上次 ANALYZE 很久了。这时可以先使用 ANALYZE 命令对统计信息进行更新。更新之后如果索引的使用上还是错误的，可以查看下一小节。

健康度接近 100%

这时意味着刚刚结束 ANALYZE 命令或者结束后不久。这时可能和 TiDB 对行数的估算逻辑有关。

对于等值查询，错误索引可能是由Count-Min Sketch 引起的。这时可以先检查是不是这种特殊情况，然后进行对应的处理。

如果经过检查发现不是上面的可能情况，可以使用Optimizer Hints 中提到的 USE_INDEX 或者 `use index` 来强制选择索引。同时也可以使用执行计划管理中提到的方式来非侵入地更改查询的行为。

其他情况

除去上述情况外，也存在因为数据的更新导致现有所有索引都不再适合的情况。这时就需要对条件和数据分布进行分析，查看是否有新的索引可以加快查询速度，然后使用ADD INDEX 命令增加新的索引。

9.3.3.3.5 Distinct 优化

本文档介绍可用于 DISTINCT 的优化，包括简单 DISTINCT 和聚合函数 DISTINCT 的优化。

简单 DISTINCT

通常简单的 DISTINCT 会被优化成 GROUP BY 来执行。例如：

```
mysql> explain select DISTINCT a from t;
+--
    ↪ ----------------------+---------+-----------+---------------+-------------------------------------------
    ↪
| id                    | estRows | task      | access object | operator info
    ↪                                        |
+--
    ↪ ----------------------+---------+-----------+---------------+-------------------------------------------
    ↪
| HashAgg_6             | 2.40    | root      |               | group by:test.t.a, funcs:
    ↪ firstrow(test.t.a)->test.t.a |
| └─TableReader_11      | 3.00    | root      |               | data:TableFullScan_10
    ↪                                  |
```

```
|   └─TableFullScan_10     | 3.00    | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo                        |
+--
    ↪ ------------------------+---------+-----------+---------------+--------------------------------
    ↪
3 rows in set (0.00 sec)
```

聚合函数 DISTINCT

通常来说，带有 DISTINCT 的聚合函数会单线程的在 TiDB 侧执行。使用系统变量 tidb_opt_distinct_agg_push_down ↪ 或者 TiDB 的配置项 distinct-agg-push-down 控制优化器是否执行带有 DISTINCT 的聚合函数（比如 select ↪ count(distinct a)from t）下推到 Coprocessor 的优化操作。

在以下示例中，tidb_opt_distinct_agg_push_down 开启前，TiDB 需要从 TiKV 读取所有数据，并在 TiDB 侧执行 disctinct。tidb_opt_distinct_agg_push_down 开启后，distinct a 被下推到了 Coprocessor，在 HashAgg_5 里新增了一个 group by 列 test.t.a。

```
mysql> desc select count(distinct a) from test.t;
+--
    ↪ ----------------------+----------+-----------+---------------+--------------------------------
    ↪
| id                      | estRows  | task      | access object | operator info
    ↪                             |
+--
    ↪ ----------------------+----------+-----------+---------------+--------------------------------
    ↪
| StreamAgg_6             | 1.00     | root      |               | funcs:count(distinct test.t.a)
    ↪ ->Column#4 |
| └─TableReader_10        | 10000.00 | root      |               | data:TableFullScan_9
    ↪                       |
|   └─TableFullScan_9     | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo           |
+--
    ↪ ----------------------+----------+-----------+---------------+--------------------------------
    ↪
3 rows in set (0.01 sec)

mysql> set session tidb_opt_distinct_agg_push_down = 1;
Query OK, 0 rows affected (0.00 sec)

mysql> desc select count(distinct a) from test.t;
+--
    ↪ ------------------------+----------+-----------+---------------+--------------------------------
    ↪
| id                        | estRows  | task      | access object | operator info
    ↪                             |
+--
```

```
    ↪ ------------------------+----------+-----------+---------------+--------------------------------
    ↪
| HashAgg_8                 | 1.00     | root      |               | funcs:count(distinct test.t.
    ↪ a)->Column#3 |
| └─TableReader_9           | 1.00     | root      |               | data:HashAgg_5
    ↪                             |
|   └─HashAgg_5             | 1.00     | cop[tikv] |               | group by:test.t.a,
    ↪                            |
|     └─TableFullScan_7     | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo           |
+--
    ↪ ------------------------+----------+-----------+---------------+--------------------------------
    ↪
4 rows in set (0.00 sec)
```

9.3.3.4 执行计划缓存

> **警告：**
>
> 该功能目前为实验特性，不建议在生产环境中使用。

TiDB 支持对 Prepare / Execute 请求的执行计划缓存。

Prepare / Execute 请求有两种形式：

- 在 binary 通信协议下，用 COM_STMT_PREPARE 和 COM_STMT_EXECUTE 命令执行一般的参数化 SQL 查询；
- 在文本通信协议下，用 COM_QUERY 命令执行 Prepare / Execute SQL 查询；

优化器对这两种请求形式的处理是一致的：Prepare 时将参数化的 SQL 查询解析成 AST（抽象语法树），每次 Execute 时根据保存的 AST 和具体的参数值生成执行计划。

当开启执行计划缓存后，每条 Prepare 语句的第一次 Execute 会检查当前查询是否可以使用执行计划缓存，如果可以则将生成的执行计划放进一个由 LRU 链表构成的缓存中；在后续的 Execute 中，会先从缓存中获取执行计划，并检查是否可用，如果获取和检查成功则跳过生成执行计划这一步，否则重新生成执行计划并放入缓存中。

在当前版本中，当 Prepare 语句符合以下条件任何一条，查询不能使用执行计划缓存：

- 查询包含除 ? 外的变量（系统变量或用户自定义变量）；
- 查询包含子查询；
- 查询包含不能被缓存的函数表达式，如 current_user()、database()、last_insert_id() 等；
- 查询的 Order By 语句中包含 ?；
- 查询的 Group By 语句中包含 ?；
- 查询的 Limit [Offset] 语句中包含 ?；

- 查询包含的 Window 函数的 window frame 定义含有 ?；
- 查询引用了分区表；

LRU 链表是设计成 session 级别的缓存，因为 Prepare / Execute 不能跨 session 执行。LRU 链表的每个元素是一个 key-value 对，value 是执行计划，key 由如下几部分组成：

- 执行 Execute 时所在数据库的名字；
- Prepare 语句的标识符，即紧跟在 PREPARE 关键字后的名字；
- 当前的 schema 版本，每条执行成功的 DDL 语句会修改 schema 版本；
- 执行 Execute 时的 SQL Mode；
- 当前设置的时区，即系统变量 time_zone 的值；

key 中任何一项变动（如切换数据库，重命名 Prepare 语句，执行 DDL，或修改 SQL Mode / time_zone 的值），或 LRU 淘汰机制触发都会导致 Execute 时无法命中执行计划缓存。

成功从缓存中获取到执行计划后，TiDB 会先检查执行计划是否依然合法，如果当前 Execute 在显式事务里执行，并且引用的表在事务前序语句中被修改，而缓存的执行计划对该表访问不包含 UnionScan 算子，则它不能被执行。

在通过合法性检测后，会根据当前最新参数值，对执行计划的扫描范围做相应调整，再用它执行获取数据。

关于执行计划缓存和查询性能有两点值得注意：

- 考虑到不同 Execute 的参数会不同，执行计划缓存为了保证适配性会禁止一些和具体参数值密切相关的激进查询优化手段，导致对特定的一些参数值，查询计划可能不是最优。比如查询的过滤条件为 where a > ? and a < ?，第一次 Execute 时参数分别为 2 和 1，考虑到这两个参数下次执行时可能会是 1 和 2，优化器不会生成对当前参数最优的 TableDual 执行计划。
- 如果不考虑缓存失效和淘汰，一份执行计划缓存会对应各种不同的参数取值，理论上也会导致某些取值下执行计划非最优。比如查询过滤条件为 where a < ?，假如第一次执行 Execute 时用的参数值为 1，此时优化器生成最优的 IndexScan 执行计划放入缓存，在后续执行 Exeucte 时参数变为 10000，此时 TableScan 可能才是更优执行计划，但由于执行计划缓存，执行时还是会使用先前生成的 IndexScan。因此执行计划缓存更适用于查询较为简单（查询编译耗时占比较高）且执行计划较为固定的业务场景。

目前执行计划缓存功能默认关闭，可以通过打开配置文件中prepare-plan-cache 项 启用这项功能。

> 注意：
>
> 执行计划缓存功能仅针对 Prepare / Execute 请求，对普通查询无效。

在开启了执行计划缓存功能后，可以通过 session 级别的系统变量 last_plan_from_cache 查看上一条 Execute 语句是否使用了缓存的执行计划，例如：

```
MySQL [test]> create table t(a int);
Query OK, 0 rows affected (0.00 sec)
```

```
MySQL [test]> prepare stmt from 'select * from t where a = ?';
Query OK, 0 rows affected (0.00 sec)

MySQL [test]> set @a = 1;
Query OK, 0 rows affected (0.00 sec)

-- 第一次 execute 生成执行计划放入缓存
MySQL [test]> execute stmt using @a;
Empty set (0.00 sec)

MySQL [test]> select @@last_plan_from_cache;
+------------------------+
| @@last_plan_from_cache |
+------------------------+
| 0                      |
+------------------------+
1 row in set (0.00 sec)

-- 第二次 execute 命中缓存
MySQL [test]> execute stmt using @a;
Empty set (0.00 sec)

MySQL [test]> select @@last_plan_from_cache;
+------------------------+
| @@last_plan_from_cache |
+------------------------+
| 1                      |
+------------------------+
1 row in set (0.00 sec)
```

如果发现某一组 Prepare / Execute 由于执行计划缓存导致了非预期行为，可以通过 SQL Hint ignore_plan_cache() 让该组语句不使用缓存。还是用上述的 stmt 为例：

```
MySQL [test]> prepare stmt from 'select /*+ ignore_plan_cache() */ * from t where a = ?';
Query OK, 0 rows affected (0.00 sec)

MySQL [test]> set @a = 1;
Query OK, 0 rows affected (0.00 sec)

MySQL [test]> execute stmt using @a;
Empty set (0.00 sec)

MySQL [test]> select @@last_plan_from_cache;
+------------------------+
| @@last_plan_from_cache |
+------------------------+
```

```
| 0                      |
+------------------------+
1 row in set (0.00 sec)

MySQL [test]> execute stmt using @a;
Empty set (0.00 sec)

MySQL [test]> select @@last_plan_from_cache;
+------------------------+
| @@last_plan_from_cache |
+------------------------+
| 0                      |
+------------------------+
1 row in set (0.00 sec)
```

9.3.4 控制执行计划

9.3.4.1 控制执行计划

SQL 性能调优的前两个章节介绍了如何理解 TiDB 的执行计划以及 TiDB 如何生成一个执行计划。本章节将介绍当你确定了执行计划所存在的问题时，可以使用哪些手段来控制执行计划的生成。本章节主要包括以下三方面内容：

- Optimizer Hints中，我们会介绍如何使用 Hint 来指导 TiDB 生成执行计划。
- 但是使用 Hint 会侵入性地更改 SQL，在一些场景下并不能简单的插入 Hint。在执行计划管理中，我们会介绍 TiDB 如何使用另一种语法来非侵入地控制执行计划的生成，同时还会介绍后台自动对执行计划进行演进的手段，来减轻诸如版本升级等原因造成的执行计划不稳定，而导致集群性能下降的问题。
- 最后在优化规则及表达式下推的黑名单中，我们会介绍黑名单的使用。

9.3.4.2 Optimizer Hints

TiDB 支持 Optimizer Hints 语法，它基于 MySQL 5.7 中介绍的类似 comment 的语法，例如 `/*+ HINT_NAME(t1, t2)*/`。当 TiDB 优化器选择的不是最优查询计划时，建议使用 Optimizer Hints。

> **注意：**
>
> MySQL 命令行客户端在 5.7.7 版本之前默认清除了 Optimizer Hints。如果需要在这些早期版本的客户端中使用 `Hint` 语法，需要在启动客户端时加上 `--comments` 选项，例如 `mysql -h 127.0.0.1 -P 4000 -uroot --comments`。

9.3.4.2.1 语法

Optimizer Hints 不区分大小写，通过 `/*+ ... */` 注释的形式跟在 SELECT、UPDATE 或 DELETE 关键字的后面。INSERT 关键字后不支持 Optimizer Hints。

多个不同的 Hint 之间需用逗号隔开，例如：

```
SELECT /*+ USE_INDEX(t1, idx1), HASH_AGG(), HASH_JOIN(t1) */ count(*) FROM t t1, t t2 WHERE t1.a
    ↪ = t2.b;
```

可以通过`Explain` / `Explain Analyze` 语句的输出，来查看 Optimizer Hints 对查询执行计划的影响。

如果 Optimizer Hints 包含语法错误或不完整，查询语句不会报错，而是按照没有 Optimizer Hints 的情况执行。如果 Hint 不适用于当前语句，TiDB 会返回 Warning，用户可以在查询结束后通过 `Show Warnings` 命令查看具体信息。

> **注意：**
>
> 如果注释不是跟在指定的关键字后，会被当作是普通的 MySQL comment，注释不会生效，且不会上报 warning。

TiDB 目前支持的 Optimizer Hints 根据生效范围的不同可以划分为两类：第一类是在查询块范围生效的 Hint，例如`/*+ HASH_AGG()*/`；第二类是在整个查询范围生效的 Hint，例如`/*+ MEMORY_QUOTA(1024 MB)*/`。

每条语句中每一个查询和子查询都对应着一个不同的查询块，每个查询块有自己对应的名字。以下面这条语句为例：

```
SELECT * FROM (SELECT * FROM t) t1, (SELECT * FROM t) t2;
```

该查询语句有 3 个查询块，最外面一层 SELECT 所在的查询块的名字为 `sel_1`，两个 SELECT 子查询的名字依次为 `sel_2` 和 `sel_3`。其中数字序号根据 SELECT 出现的位置从左到右计数。如果分别用 DELETE 和 UPDATE 查询替代第一个 SELECT 查询，则对应的查询块名字分别为 `del_1` 和 `upd_1`。

9.3.4.2.2 查询块范围生效的 Hint

这类 Hint 可以跟在查询语句中任意 SELECT、UPDATE 或 DELETE 关键字的后面。通过在 Hint 中使用查询块名字可以控制 Hint 的生效范围，以及准确标识查询中的每一个表（有可能表的名字或者别名相同），方便明确 Hint 的参数指向。若不显式地在 Hint 中指定查询块，Hint 默认作用于当前查询块。以如下查询为例：

```
SELECT /*+ HASH_JOIN(@sel_1 t1@sel_1, t3) */ * FROM (SELECT t1.a, t1.b FROM t t1, t t2 WHERE t1.a
    ↪  = t2.a) t1, t t3 WHERE t1.b = t3.b;
```

该 Hint 在 `sel_1` 这个查询块中生效，参数分别为 `sel_1` 中的 `t1` 表（`sel_2` 中也有一个 `t1` 表）和 `t3` 表。

如上例所述，在 Hint 中使用查询块名字的方式有两种：第一种是作为 Hint 的第一个参数，与其他参数用空格隔开。除 `QB_NAME` 外，本节所列的所有 Hint 除自身明确列出的参数外都有一个隐藏的可选参数 `@QB_NAME`，通过使用这个参数可以指定该 Hint 的生效范围；第二种在 Hint 中使用查询块名字的方式是在参数中的某一个表名后面加 `@QB_NAME`，用以明确指出该参数是哪个查询块中的表。

> **注意：**
>
> Hint 声明的位置必须在指定生效的查询块之中或之前，不能是在之后的查询块中，否则无法生效。

QB_NAME

当查询语句是包含多层嵌套子查询的复杂语句时，识别某个查询块的序号和名字很可能会出错，Hint QB_NAME 可以方便我们使用查询块。QB_NAME 是 Query Block Name 的缩写，用于为某个查询块指定新的名字，同时查询块原本默认的名字依然有效。例如：

```
SELECT /*+ QB_NAME(QB1) */ * FROM (SELECT * FROM t) t1, (SELECT * FROM t) t2;
```

这条 Hint 将最外层 SELECT 查询块的命名为 QB1，此时 QB1 和默认名称 sel_1 对于这个查询块来说都是有效的。

> **注意：**
>
> 上述例子中，如果指定的 QB_NAME 为 sel_2，并且不给原本 sel_2 对应的第二个查询块指定新的 QB_NAME，则第二个查询块的默认名字 sel_2 会失效。

MERGE_JOIN(t1_name [, tl_name …])

MERGE_JOIN(t1_name [, tl_name ...]) 提示优化器对指定表使用 Sort Merge Join 算法。这个算法通常会占用更少的内存，但执行时间会更久。当数据量太大，或系统内存不足时，建议尝试使用。例如：

```
SELECT /*+ MERGE_JOIN(t1, t2) */ * FROM t1, t2 WHERE t1.id = t2.id;
```

> **注意：**
>
> MERGE_JOIN 的别名是 TIDB_SMJ，在 3.0.x 及之前版本仅支持使用该别名；之后的版本同时支持使用这两种名称，但推荐使用 MERGE_JOIN。

INL_JOIN(t1_name [, tl_name …])

INL_JOIN(t1_name [, tl_name ...]) 提示优化器对指定表使用 Index Nested Loop Join 算法。这个算法可能会在某些场景更快，消耗更少系统资源，有的场景会更慢，消耗更多系统资源。对于外表经过 WHERE 条件过滤后结果集较小（小于 1 万行）的场景，可以尝试使用。例如：

```
SELECT /*+ INL_JOIN(t1, t2) */ * FROM t1, t2 WHERE t1.id = t2.id;
```

INL_JOIN() 中的参数是建立查询计划时内表的候选表，比如 INL_JOIN(t1) 只会考虑使用 t1 作为内表构建查询计划。表如果指定了别名，就只能使用表的别名作为 INL_JOIN() 的参数；如果没有指定别名，则用表的本

名作为其参数。比如在 `SELECT /*+ INL_JOIN(t1)*/ * FROM t t1, t t2 WHERE t1.a = t2.b;` 中，`INL_JOIN()` 的参数只能使用 t 的别名 t1 或 t2，不能用 t。

> **注意：**
>
> `INL_JOIN` 的别名是 `TIDB_INLJ`，在 3.0.x 及之前版本仅支持使用该别名；之后的版本同时支持使用这两种名称，但推荐使用 `INL_JOIN`。

INL_HASH_JOIN

`INL_HASH_JOIN(t1_name [, tl_name])` 提示优化器使用 Index Nested Loop Hash Join 算法。该算法与 Index Nested Loop Join 使用条件完全一样，两者的区别是 `INL_JOIN` 会在连接的内表上建哈希表，而 `INL_HASH_JOIN` 会在连接的外表上建哈希表，后者对于内存的使用是有固定上限的，而前者使用的内存使用取决于内表匹配到的行数。

INL_MERGE_JOIN

`INL_MERGE_JOIN(t1_name [, tl_name])` 提示优化器使用 Index Nested Loop Merge Join 算法。这个 Hint 的适用场景和 `INL_JOIN` 一致，相比于 `INL_JOIN` 和 `INL_HASH_JOIN` 会更节省内存，但使用条件会更苛刻：join keys 中的内表列集合是内表使用的索引的前缀，或内表使用的索引是 join keys 中的内表列集合的前缀。

HASH_JOIN(t1_name [, tl_name …])

`HASH_JOIN(t1_name [, tl_name ...])` 提示优化器对指定表使用 Hash Join 算法。这个算法多线程并发执行，执行速度较快，但会消耗较多内存。例如：

```sql
SELECT /*+ HASH_JOIN(t1, t2) */ * FROM t1, t2 WHERE t1.id = t2.id;
```

> **注意：**
>
> `HASH_JOIN` 的别名是 `TIDB_HJ`，在 3.0.x 及之前版本仅支持使用该别名；之后的版本同时支持使用这两种名称，推荐使用 `HASH_JOIN`。

HASH_AGG()

`HASH_AGG()` 提示优化器对指定查询块中所有聚合函数使用 Hash Aggregation 算法。这个算法多线程并发执行，执行速度较快，但会消耗较多内存。例如：

```sql
SELECT /*+ HASH_AGG() */ count(*) FROM t1, t2 WHERE t1.a > 10 GROUP BY t1.id;
```

STREAM_AGG()

`STREAM_AGG()` 提示优化器对指定查询块中所有聚合函数使用 Stream Aggregation 算法。这个算法通常会占用更少的内存，但执行时间会更久。数据量太大，或系统内存不足时，建议尝试使用。例如：

```sql
SELECT /*+ STREAM_AGG() */ count(*) FROM t1, t2 WHERE t1.a > 10 GROUP BY t1.id;
```

USE_INDEX(t1_name, idx1_name [, idx2_name …])

`USE_INDEX(t1_name, idx1_name [, idx2_name ...])` 提示优化器对指定表仅使用给出的索引。

下面例子的效果等价于 `SELECT * FROM t t1 use index(idx1, idx2);`：

```
SELECT /*+ USE_INDEX(t1, idx1, idx2) */ * FROM t1;
```

> 注意：
>
> 当该 Hint 中只指定表名，不指定索引名时，表示不考虑使用任何索引，而是选择全表扫。

IGNORE_INDEX(t1_name, idx1_name [, idx2_name …])

`IGNORE_INDEX(t1_name, idx1_name [, idx2_name ...])` 提示优化器对指定表忽略给出的索引。

下面例子的效果等价于 `SELECT * FROM t t1 ignore index(idx1, idx2);`：

```
SELECT /*+ IGNORE_INDEX(t1, idx1, idx2) */ * FROM t t1;
```

AGG_TO_COP()

`AGG_TO_COP()` 提示优化器将指定查询块中的聚合函数下推到 coprocessor。如果优化器没有下推某些适合下推的聚合函数，建议尝试使用。例如：

```
SELECT /*+ AGG_TO_COP() */ sum(t1.a) FROM t t1;
```

LIMIT_TO_COP()

`LIMIT_TO_COP()` 提示优化器将指定查询块中的 `Limit` 和 `TopN` 算子下推到 coprocessor。优化器没有下推 `Limit` 或者 `TopN` 算子时建议尝试使用该提示。例如：

```
SELECT /*+ LIMIT_TO_COP() */ * FROM t WHERE a = 1 AND b > 10 ORDER BY c LIMIT 1;
```

READ_FROM_STORAGE(TIFLASH[t1_name [, tl_name …]], TIKV[t2_name [, tl_name …]])

`READ_FROM_STORAGE(TIFLASH[t1_name [, tl_name ...]], TIKV[t2_name [, tl_name ...]])` 提示优化器从指定的存储引擎来读取指定的表，目前支持的存储引擎参数有 `TIKV` 和 `TIFLASH`。如果为表指定了别名，就只能使用表的别名作为 `READ_FROM_STORAGE()` 的参数；如果没有指定别名，则用表的本名作为其参数。例如：

```
SELECT /*+ READ_FROM_STORAGE(TIFLASH[t1], TIKV[t2]) */ t1.a FROM t t1, t t2 WHERE t1.a = t2.a;
```

> 注意：
>
> 如果需要提示优化器使用的表不在同一个数据库内，需要显式指定数据库名。例如 `SELECT /*+ READ_FROM_STORAGE(TIFLASH[test1.t1,test2.t2])*/ t1.a FROM test1 ↪ .t t1, test2.t t2 WHERE t1.a = t2.a;`。

USE_INDEX_MERGE(t1_name, idx1_name [, idx2_name …])

USE_INDEX_MERGE(t1_name, idx1_name [, idx2_name ...]) 提示优化器通过 index merge 的方式来访问指定的表，其中索引列表为可选参数。若显式地指出索引列表，会尝试在索引列表中选取索引来构建 index merge。若不给出索引列表，会尝试在所有可用的索引中选取索引来构建 index merge。例如：

```
SELECT /*+ USE_INDEX_MERGE(t1, idx_a, idx_b, idx_c) */ * FROM t1 WHERE t1.a > 10 OR t1.b > 10;
```

当对同一张表有多个 USE_INDEX_MERGE Hint 时，优化器会从这些 Hint 指定的索引列表的并集中尝试选取索引。

> 注意：
>
> USE_INDEX_MERGE 的参数是索引名，而不是列名。对于主键索引，索引名为 primary。

目前该 Hint 生效的条件较为苛刻，包括：

- 如果查询有除了全表扫以外的单索引扫描方式可以选择，优化器不会选择 index merge；
- 如果查询在显式事务里，且该条查询之前的语句已经涉及写入，优化器不会选择 index merge；

9.3.4.2.3 查询范围生效的 Hint

这类 Hint 只能跟在语句中第一个 SELECT、UPDATE 或 DELETE 关键字的后面，等同于在当前这条查询运行时对指定的系统变量进行修改，其优先级高于现有系统变量的值。

> 注意：
>
> 这类 Hint 虽然也有隐藏的可选变量 @QB_NAME，但就算指定了该值，Hint 还是会在整个查询范围生效。

NO_INDEX_MERGE()

NO_INDEX_MERGE() 会关闭优化器的 index merge 功能。

下面的例子不会使用 index merge：

```
SELECT /*+ NO_INDEX_MERGE() */ * FROM t WHERE t.a > 0 or t.b > 0;
```

除了 Hint 外，系统变量 tidb_enable_index_merge 也能决定是否开启该功能。

> 注意：
>
> NO_INDEX_MERGE 优先级高于 USE_INDEX_MERGE，当这两类 Hint 同时存在时，USE_INDEX_MERGE 不会生效。

USE_TOJA(boolean_value)

参数 boolean_value 可以是 TRUE 或者 FALSE。USE_TOJA(TRUE) 会开启优化器尝试将 in (subquery) 条件转换为 join 和 aggregation 的功能。相对地，USE_TOJA(FALSE) 会关闭该功能。

下面的例子会将 in (SELECT t2.a FROM t2)subq 转换为等价的 join 和 aggregation：

```
SELECT /*+ USE_TOJA(TRUE) */ t1.a, t1.b FROM t1 WHERE t1.a in (SELECT t2.a FROM t2) subq;
```

除了 Hint 外，系统变量 tidb_opt_insubq_to_join_and_agg 也能决定是否开启该功能。

MAX_EXECUTION_TIME(N)

MAX_EXECUTION_TIME(N) 把语句的执行时间限制在 N 毫秒以内，超时后服务器会终止这条语句的执行。

下面的 Hint 设置了 1000 毫秒（即 1 秒）超时：

```
SELECT /*+ MAX_EXECUTION_TIME(1000) */ * FROM t1 inner join t2 WHERE t1.id = t2.id;
```

除了 Hint 之外，系统变量 global.max_execution_time 也能对语句执行时间进行限制。

MEMORY_QUOTA(N)

MEMORY_QUOTA(N) 用于限制语句执行时的内存使用。该 Hint 支持 MB 和 GB 两种单位。内存使用超过该限制时会根据当前设置的内存超限行为来打出一条 log 或者终止语句的执行。

下面的 Hint 设置了 1024 MB 的内存限制：

```
SELECT /*+ MEMORY_QUOTA(1024 MB) */ * FROM t;
```

除了 Hint 外，系统变量 tidb_mem_quota_query 也能限制语句执行的内存使用。

READ_CONSISTENT_REPLICA()

READ_CONSISTENT_REPLICA() 会开启从数据一致的 TiKV follower 节点读取数据的特性。

下面的例子会从 follower 节点读取数据：

```
SELECT /*+ READ_CONSISTENT_REPLICA() */ * FROM t;
```

除了 Hint 外，环境变量 tidb_replica_read 设为 'follower' 或者 'leader' 也能决定是否开启该特性。

IGNORE_PLAN_CACHE()

IGNORE_PLAN_CACHE() 提示优化器在处理当前 prepare 语句时不使用 plan cache。

该 Hint 用于在prepare-plan-cache 开启的场景下临时对某类查询禁用 plan cache。

以下示例强制该 prepare 语句不使用 plan cache：

```
prepare stmt FROM 'SELECT  /*+ IGNORE_PLAN_CACHE() */ * FROM t WHERE t.id = ?';
```

9.3.4.3 执行计划管理 (SPM)

执行计划管理，又称 SPM (SQL Plan Management)，是通过执行计划绑定，对执行计划进行人为干预的一系列功能，包括执行计划绑定、自动捕获绑定、自动演进绑定等。

9.3.4.3.1 执行计划绑定 (SQL Binding)

执行计划绑定是 SPM 的基础。在优化器 Hints 中介绍了可以通过 Hint 的方式选择指定的执行计划，但有时需要在不修改 SQL 语句的情况下干预执行计划的选择。执行计划绑定功能使得可以在不修改 SQL 语句的情况下选择指定的执行计划。

创建绑定

```
CREATE [GLOBAL | SESSION] BINDING FOR BindableStmt USING BindableStmt;
```

该语句可以在 GLOBAL 或者 SESSION 作用域内为 SQL 绑定执行计划。目前支持的可创建执行计划绑定的 SQL 类型（BindableStmt）包括：SELECT，DELETE，UPDATE 和带有 SELECT 子查询的 INSERT / REPLACE。

其中，有两类特定的语法由于语法冲突不能创建执行计划绑定，例如：

```
-- 类型一：使用 `join` 关键字但不通过 `using` 关键字指定关联列的笛卡尔积
create global binding for
    select * from t t1 join t t2
using
    select * from t t1 join t t2;

-- 类型二：包含了 `using` 关键字的 `delete` 语句
create global binding for
    delete from t1 using t1 join t2 on t1.a = t2.a
using
    delete from t1 using t1 join t2 on t1.a = t2.a;
```

可以通过等价的 SQL 改写绕过这个语法冲突的问题。例如，上述两个例子可以改写为：

```
-- 类型一的第一种改写：为 `join` 关键字添加 `using` 子句
create global binding for
    select * from t t1 join t t2 using (a)
using
    select * from t t1 join t t2 using (a);

-- 类型一的第二种改写：去掉 `join` 关键字
create global binding for
    select * from t t1, t t2
using
    select * from t t1, t t2;

-- 类型二的改写：去掉 `delete` 语句中的 `using` 关键字
create global binding for
    delete t1 from t1 join t2 on t1.a = t2.a
using
    delete t1 from t1 join t2 on t1.a = t2.a;
```

> **注意：**
>
> 在对带 SELECT 子查询的 INSERT / REPLACE 语句创建执行计划绑定时，需要将想要绑定的优化器 Hints 指定在 SELECT 子查询中，而不是 INSERT / REPLACE 关键字后，不然优化器 Hints 不会生效。

例如：

```
-- Hint 能生效的用法
create global binding for
    insert into t1 select * from t2 where a > 1 and b = 1
using
    insert into t1 select /*+ use_index(@sel_1 t2, a) */ * from t2 where a > 1 and b = 1;

-- Hint 不能生效的用法
create global binding for
    insert into t1 select * from t2 where a > 1 and b = 1
using
    insert /*+ use_index(@sel_1 t2, a) */ into t1 select * from t2 where a > 1 and b = 1;
```

如果在创建执行计划绑定时不指定作用域，隐式作用域 SESSION 会被使用。TiDB 优化器会将被绑定的 SQL 进行“标准化”处理，然后存储到系统表中。在处理 SQL 查询时，只要“标准化”后的 SQL 和系统表中某个被绑定的 SQL 语句一致，并且系统变量 tidb_use_plan_baselines 的值为 on（其默认值为 on），即可使用相应的优化器 Hint。如果存在多个可匹配的执行计划，优化器会从中选择代价最小的一个进行绑定。

标准化：把 SQL 中的常量变成变量参数，对空格和换行符等做标准化处理，并对查询引用到的表显式指定数据库。例如：

```
select * from t where a >    1
-- 标准化后：
select * from test . t where a > ?
```

值得注意的是，如果一条 SQL 语句在 GLOBAL 和 SESSION 作用域内都有与之绑定的执行计划，因为优化器在遇到 SESSION 绑定时会忽略 GLOBAL 绑定的执行计划，该语句在 SESSION 作用域内绑定的执行计划会屏蔽掉语句在 GLOBAL 作用域内绑定的执行计划。

例如：

```
-- 创建一个 global binding，指定其使用 sort merge join
create global binding for
    select * from t1, t2 where t1.id = t2.id
using
    select /*+ merge_join(t1, t2) */ * from t1, t2 where t1.id = t2.id;

-- 从该 SQL 的执行计划中可以看到其使用了 global binding 中指定的 sort merge join
explain select * from t1, t2 where t1.id = t2.id;
```

```
-- 创建另一个 session binding，指定其使用 hash join
create binding for
    select * from t1, t2 where t1.id = t2.id
using
    select /*+ hash_join(t1, t2) */ * from t1, t2 where t1.id = t2.id;

-- 从该 SQL 的执行计划中可以看到其使用了 session binding 中指定的 hash join，而不是 global
    ↪ binding 中指定的 sort merge join
explain select * from t1, t2 where t1.id = t2.id;
```

第一个 `select` 语句在执行时优化器会通过 GLOBAL 作用域内的绑定为其加上 `sm_join(t1, t2)` hint，`explain` 出的执行计划中最上层的节点为 MergeJoin。而第二个 `select` 语句在执行时优化器则会忽视 GLOBAL 作用域内的绑定而使用 SESSION 作用域内的绑定为该语句加上 `hash_join(t1, t2)` hint，`explain` 出的执行计划中最上层的节点为 HashJoin。

每个标准化的 SQL 只能同时有一个通过 `CREATE BINDING` 创建的绑定。对相同的标准化 SQL 创建多个绑定时，会保留最后一个创建的绑定，之前的所有绑定（创建的和演进出来的）都会被删除。但 session 绑定和 global 绑定仍然允许共存，不受这个逻辑影响。

另外，创建绑定时，TiDB 要求 session 处于某个数据库上下文中，也就是执行过 `use ${database}` 或者客户端连接时指定了数据库。

需要注意的是原始 SQL 和绑定 SQL 在参数化以及去掉 Hint 后文本必须相同，否则创建会失败，例如：

```
CREATE BINDING FOR SELECT * FROM t WHERE a > 1 USING SELECT * FROM t use index(idx) WHERE a > 2;
```

可以创建成功，因为原始 SQL 和绑定 SQL 在参数化以及去掉 Hint 后文本都是 `select * from test . t where a > ?`，而

```
CREATE BINDING FOR SELECT * FROM t WHERE a > 1 USING SELECT * FROM t use index(idx) WHERE b > 2;
```

则不可以创建成功，因为原始 SQL 在经过处理后是 `select * from test . t where a > ?`，而绑定 SQL 在经过处理后是 `select * from test . t where b > ?`。

> **注意：**
>
> 对于 PREPARE / EXECUTE 语句组，或者用二进制协议执行的查询，创建执行计划绑定的对象应当是查询语句本身，而不是 PREPARE / EXECUTE 语句。

删除绑定

```
DROP [GLOBAL | SESSION] BINDING FOR BindableStmt;
```

该语句可以在 GLOBAL 或者 SESSION 作用域内删除指定的执行计划绑定，在不指定作用域时默认作用域为 SESSION。

一般来说，SESSION 作用域的绑定主要用于测试或在某些特殊情况下使用。若需要集群中所有的 TiDB 进程都生效，则需要使用 GLOBAL 作用域的绑定。SESSION 作用域对 GLOBAL 作用域绑定的屏蔽效果会持续到该 SESSION 结束。

承接上面关于 SESSION 绑定屏蔽 GLOBAL 绑定的例子，继续执行：

```
-- 删除 session 中创建的 binding
drop session binding for select * from t1, t2 where t1.id = t2.id;

-- 重新查看该 SQL 的执行计划
explain select * from t1,t2 where t1.id = t2.id;
```

在这里 SESSION 作用域内被删除掉的绑定会屏蔽 GLOBAL 作用域内相应的绑定，优化器不会为 `select` 语句添加 `sm_join(t1, t2)` hint，`explain` 给出的执行计划中最上层节点并不被 hint 固定为 MergeJoin，而是由优化器经过代价估算后自主进行选择。

查看绑定

```
SHOW [GLOBAL | SESSION] BINDINGS [ShowLikeOrWhere];
```

该语句会输出 GLOBAL 或者 SESSION 作用域内的执行计划绑定，在不指定作用域时默认作用域为 SESSION。目前 `SHOW BINDINGS` 会输出 8 列，具体如下：

列名	说明
original_sql	参数化后的原始 SQL
bind_sql	带 Hint 的绑定 SQL
default_db	默认数据库名
status	状态，包括 using（正在使用）、deleted（已删除）、invalid（无效）、rejected（演进时被拒绝）和 pending verify（等待演进验证）
create_time	创建时间
update_time	更新时间
charset	字符集
collation	排序规则

列名	说明
source	创建方式，包括 manual（由 `create [global] binding` 生成）、capture（由 tidb 自动创建生成）和 evolve（由 tidb 自动演进生成）

9.3.4.3.2 自动捕获绑定 (Baseline Capturing)

通过将 `tidb_capture_plan_baselines` 的值设置为 on（其默认值为 `off`）可以打开自动捕获绑定功能。

> **注意：**
>
> 自动绑定功能依赖于Statement Summary，因此在使用自动绑定之前需打开 Statement Summary 开关。

开启自动绑定功能后，每隔 `bind-info-lease`（默认值为 `3s`）会遍历一次 Statement Summary 中的历史 SQL 语句，并为至少出现两次的 SQL 语句自动捕获绑定。绑定的执行计划为 Statement Summary 中记录执行这条语句时使用的执行计划。

对于以下几种 SQL 语句，TiDB 不会自动捕获绑定：

- EXPLAIN 和 EXPLAIN ANALYZE 语句；
- TiDB 内部执行的 SQL 语句，比如统计信息自动加载使用的 SELECT 查询；
- 存在手动创建的执行计划绑定的 SQL 语句；

对于 PREPARE / EXECUTE 语句组，或通过二进制协议执行的查询，TiDB 会为真正的查询（而不是 PREPARE / EXECUTE 语句）自动捕获绑定。

> **注意：**
>
> 由于 TiDB 存在一些内嵌 SQL 保证一些功能的正确性，所以自动捕获绑定时会默认屏蔽内嵌 SQL。

9.3.4.3.3 自动演进绑定 (Baseline Evolution)

自动演进绑定，在 TiDB 4.0.0-rc 版本引入，是执行计划管理的重要功能之一。

由于某些数据变更后，原先绑定的执行计划可能是一个不优的计划。为了解决该问题，引入自动演进绑定功能来自动优化已经绑定的执行计划。

另外自动演进绑定还可以一定程度上避免统计信息改动后，对执行计划带来的抖动。

使用方式

通过以下语句可以开启自动演进绑定功能：

```
set global tidb_evolve_plan_baselines = on;
```

tidb_evolve_plan_baselines 的默认值为 off。

> 注意：
>
> 自动演进绑定功能目前不是 GA (Generally Available) 状态，不推荐在生产环境打开该功能。

在打开自动演进功能后，如果优化器选出的最优执行计划不在之前绑定的执行计划之中，会将其记录为待验证的执行计划。每隔 bind-info-lease（默认值为 3s），会选出一个待验证的执行计划，将其和已经绑定的执行计划中代价最小的比较实际运行时间。如果待验证的运行时间更优的话（目前判断标准是运行时间小于等于已绑定执行计划运行时间的 2/3），会将其标记为可使用的绑定。以下示例描述上述过程。

假如有表 t 定义如下：

```
create table t(a int, b int, key(a), key(b));
```

在表 t 上进行如下查询：

```
select * from t where a < 100 and b < 100;
```

表上满足条件 a < 100 的行很少。但由于某些原因，优化器没能选中使用索引 a 这个最优执行计划，而是误选了速度慢的全表扫，那么用户首先可以通过如下语句创建一个绑定：

```
create global binding for select * from t where a < 100 and b < 100 using select * from t use
    ↪ index(a) where a < 100 and b < 100;
```

当以上查询语句再次执行时，优化器会在刚创建绑定的干预下选择使用索引 a，进而降低查询时间。

假如随着在表中进行插入和修改，表中满足条件 a < 100 的行变得越来越多，而满足条件 b < 100 的行变得越来越少，这时再在绑定的干预下使用索引 a 可能就不是最优了。

绑定的演进可以解决这类问题。当优化器感知到表数据变化后，会对这条查询生成使用索引 b 的执行计划。但由于绑定的存在，这个执行计划不会被采纳和执行，不过它会被存在后台的演进列表里。在演进过程中，如果它被验证为执行时间明显低于使用索引 a 的执行时间（即当前绑定的执行计划），那么索引 b 会被加入到可用的绑定列表中。在此之后，当这条查询再次被执行时，优化器首先生成使用索引 b 的执行计划，并确认它在绑定列表中，所以会采纳它并执行，进而可以在数据变化后降低这条查询的执行时间。

为了减少自动演进对集群的影响，可以通过设置 tidb_evolve_plan_task_max_time 来限制每个执行计划运行的最长时间，其默认值为 600s。实际在验证执行计划时，计划的最长运行时间还会被限制为不超过已验证执

行计划的运行时间的两倍；通过 `tidb_evolve_plan_task_start_time` 和 `tidb_evolve_plan_task_end_time` 可以限制运行演进任务的时间窗口，默认值分别为 `00:00 +0000` 和 `23:59 +0000`。

注意事项

由于自动演进绑定会自动地创建新的绑定，当查询的环境发生变动时，自动创建的绑定可能会有多种行为的选择。这里列出一些注意事项：

- 自动演进只会对存在至少一个 global 绑定的标准化 SQL 进行演进。
- 由于创建新的绑定会删除之前所有绑定（对于一条标准化 SQL），自动演进的绑定也会在手动重新创建绑定后被删除。
- 所有和计算过程相关的 hint，在演进时都会被保留。计算过程相关的 hint 有如下几种：

Hint	说明
memory_quota	查询过程最多可以使用多少内存
use_toja	优化器是否考虑把子查询转化为 join
use_cascades	是否使用 cascades 优化器
no_index_merge	优化器是否考虑将 index merge 作为一个读表选项
read_consistent_replica	是否强制读表时使用 follower read
max_execution_time	查询过程最多消耗多少时间

- `read_from_storage` 是一个非常特别的 hint，因为它指定了读表时选择从 TiKV 读还是从 TiFlash 读。由于 TiDB 提供隔离读的功能，当隔离条件变化时，这个 hint 对演进出来的执行计划影响很大，所以当最初创建的绑定中存在这个 hint，TiDB 会无视其所有演进的绑定。

9.3.4.4 优化规则与表达式下推的黑名单

本文主要介绍优化规则的黑名单与表达式下推的黑名单。

9.3.4.4.1 优化规则黑名单

优化规则黑名单是针对优化规则的调优手段之一，主要用于手动禁用一些优化规则。

重要的优化规则

优化规则	规则名称	简介
列裁剪	column_prune	对于上层算子不需要的列，不在下层算子输出该列，减少计算
子查询去关联	decorrelate	尝试对相关子查询进行改写，将其转换为普通 join 或 aggregation 计算
聚合消除	aggregation_eliminate	尝试消除执行计划中的某些不必要的聚合算子
投影消除	projection_eliminate	消除执行计划中不必要的投影算子
最大最小消除	max_min_eliminate	改写聚合中的 max/min 计算，转化为 `order by + limit 1`
谓词下推	predicate_push_down	尝试将执行计划中过滤条件下推到离数据源更近的算子上
外连接消除	outer_join_eliminate	尝试消除执行计划中不必要的 left join 或者 right join
分区裁剪	partition_processor	将分区表查询改成为用 union all，并裁剪掉不满足过滤条件的分区
聚合下推	aggregation_push_down	尝试将执行计划中的聚合算子下推到更底层的计算节点

优化规则	规则名称	简介
TopN 下推	topn_push_down	尝试将执行计划中的 TopN 算子下推到离数据源更近的算子上
Join 重排序	join_reorder	对多表 join 确定连接顺序

禁用优化规则

当某些优化规则在一些特殊查询中的优化结果不理想时，可以使用优化规则黑名单禁用一些优化规则。

使用方法

> 注意：
>
> 以下操作都需要数据库的 super privilege 权限。每个优化规则都有各自的名字，比如列裁剪的名字是“column_prune”。所有优化规则的名字都可以在重要的优化规则表格中第二列查到。

- 如果你想禁用某些规则，可以在 `mysql.opt_rule_blacklist` 表中写入规则的名字，例如：

```
INSERT INTO mysql.opt_rule_blacklist VALUES("join_reorder"), ("topn_push_down");
```

执行以下 SQL 语句可让禁用规则立即生效，包括相应 TiDB Server 的所有旧链接：

```
ADMIN reload opt_rule_blacklist;
```

> 注意：
>
> `admin reload opt_rule_blacklist` 只对执行该 SQL 语句的 TiDB server 生效。若需要集群中所有 TiDB server 生效，需要在每台 TiDB server 上执行该 SQL 语句。

- 需要解除一条规则的禁用时，需要删除表中禁用该条规则的相应数据，再执行 `admin reload`：

```
DELETE FROM mysql.opt_rule_blacklist WHERE name IN ("join_reorder", "topn_push_down");
admin reload opt_rule_blacklist;
```

9.3.4.4.2 表达式下推黑名单

表达式下推黑名单是针对表达式下推的调优手段之一，主要用于对于某些存储类型手动禁用一些表达式。

已支持下推的表达式

表达式分类	具体操作
逻辑运算	AND (&&), OR (||), NOT (!)
比较运算	<, <=, =, != (<>), >, >=, <=>, IN(), IS NULL, LIKE, IS TRUE, IS FALSE, COALESCE()
数值运算	+, -, *, /, ABS(), CEIL(), CEILING(), FLOOR()

表达式分类	具体操作
控制流运算	CASE, IF(), IFNULL()
JSON 运算	JSON_TYPE(json_val), JSON_EXTRACT(json_doc, path[, path] ⋯), JSON_UNQUOTE(json_val), JSON_OBJECT(key, val[, key, val] ⋯), JSON_ARRAY([val[, val] ⋯]), JSON_MERGE(json_doc, json_doc[, json_doc] ⋯), JSON_SET(json_doc, path, val[, path, val] ⋯), JSON_INSERT(json_doc, path, val[, path, val] ⋯), JSON_REPLACE(json_doc, path, val[, path, val] ⋯), JSON_REMOVE(json_doc, path[, path] ⋯)
日期运算	DATE_FORMAT()

禁止特定表达式下推

当函数的计算过程由于下推而出现异常时，可通过黑名单功能禁止其下推来快速恢复业务。具体而言，你可以将上述支持的函数或运算符名加入黑名单 `mysql.expr_pushdown_blacklist` 中，以禁止特定表达式下推。

`mysql.expr_pushdown_blacklist` 的 schema 如下：

```
DESC mysql.expr_pushdown_blacklist;
```

```
+------------+--------------+------+------+-------------------+-------+
| Field      | Type         | Null | Key  | Default           | Extra |
+------------+--------------+------+------+-------------------+-------+
| name       | char(100)    | NO   |      | NULL              |       |
| store_type | char(100)    | NO   |      | tikv,tiflash,tidb |       |
| reason     | varchar(200) | YES  |      | NULL              |       |
+------------+--------------+------+------+-------------------+-------+
3 rows in set (0.00 sec)
```

以上结果字段解释如下：

- `name`：禁止下推的函数名。
- `store_type`：用于指明希望禁止该函数下推到哪些组件进行计算。组件可选 `tidb`、`tikv` 和 `tiflash`。`store_type` 不区分大小写，如果需要禁止向多个存储引擎下推，各个存储之间需用逗号隔开。
  - `store_type` 为 `tidb` 时表示在读取 TiDB 内存表时，是否允许该函数在其他 TiDB Server 上执行。
  - `store_type` 为 `tikv` 时表示是否允许该函数在 TiKV Server 的 Coprocessor 模块中执行。
  - `store_type` 为 `tiflash` 时表示是否允许该函数在 TiFlash Server 的 Coprocessor 模块中执行。
- `reason`：用于记录该函数被加入黑名单的原因。

使用方法

加入黑名单

如果要将一个或多个函数或运算符加入黑名单，执行以下步骤：

1. 向 `mysql.expr_pushdown_blacklist` 插入对应的函数名或运算符名以及希望禁止下推的存储引擎集合。

2. 执行 `admin reload expr_pushdown_blacklist;`。

移出黑名单

如果要将一个或多个函数及运算符移出黑名单，执行以下步骤：

1. 从 `mysql.expr_pushdown_blacklist` 表中删除对应的函数名或运算符名。
2. 执行 `admin reload expr_pushdown_blacklist;`。

> **注意：**
>
> `admin reload expr_pushdown_blacklist` 只对执行该 SQL 语句的 TiDB server 生效。若需要集群中所有 TiDB server 生效，需要在每台 TiDB server 上执行该 SQL 语句。

表达式黑名单用法示例

以下示例首先将运算符 < 及 > 加入黑名单，然后将运算符 > 从黑名单中移出。

黑名单是否生效可以从 `explain` 结果中进行观察（参见EXPLAIN 简介）。

1. 对于以下 SQL 语句，`where` 条件中的 `a < 2` 和 `a > 2` 可以下推到 TiKV 进行计算。

```
EXPLAIN SELECT * FROM t WHERE a < 2 AND a > 2;
```

```
+--
    ↪ ----------------------+----------+-----------+--------------+--------------------------------+
    ↪
| id                    | estRows  | task      | access object | operator info
    ↪                     |
+--
    ↪ ----------------------+----------+-----------+--------------+--------------------------------+
    ↪
| TableReader_7           | 0.00     | root      |               | data:Selection_6
    ↪                   |
| └─Selection_6           | 0.00     | cop[tikv] |               | gt(ssb_1.t.a, 2), lt(
    ↪ ssb_1.t.a, 2) |
|   └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo      |
+--
    ↪ ----------------------+----------+-----------+--------------+--------------------------------+
    ↪
3 rows in set (0.00 sec)
```

2. 往`mysql.expr_pushdown_blacklist`表中插入禁用表达式,并且执行`admin reload expr_pushdown_blacklist`
   ↪ 。

```
INSERT INTO mysql.expr_pushdown_blacklist VALUES('<','tikv',''), ('>','tikv','');
```

```
Query OK, 2 rows affected (0.01 sec)
Records: 2  Duplicates: 0  Warnings: 0
```

```
ADMIN reload expr_pushdown_blacklist;
```

```
Query OK, 0 rows affected (0.00 sec)
```

3. 重新观察执行计划，发现表达式下推黑名单生效，where 条件中的 < 和 > 没有被下推到 TiKV Coprocessor 上。

```
EXPLAIN SELECT * FROM t WHERE a < 2 and a > 2;
```

```
+--
    ↪ ----------------------+----------+-----------+--------------+--------------------------------+
    ↪
| id                      | estRows  | task      | access object | operator info
    ↪                       |
+--
    ↪ ----------------------+----------+-----------+--------------+--------------------------------+
    ↪
| Selection_7             | 10000.00 | root      |               | gt(ssb_1.t.a, 2), lt(
    ↪ ssb_1.t.a, 2) |
| └─TableReader_6         | 10000.00 | root      |               | data:TableFullScan_5
    ↪                |
|   └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo      |
+--
    ↪ ----------------------+----------+-----------+--------------+--------------------------------+
    ↪
3 rows in set (0.00 sec)
```

4. 将某一表达式 (> 大于) 禁用规则从黑名单表中删除，并且执行 admin reload expr_pushdown_blacklist ↪ 。

```
DELETE FROM mysql.expr_pushdown_blacklist WHERE name = '>';
```

```
Query OK, 1 row affected (0.01 sec)
```

```
ADMIN reload expr_pushdown_blacklist;
```

```
Query OK, 0 rows affected (0.00 sec)
```

5. 重新观察执行计划，可以看到只有 > 表达式被重新下推到 TiKV Coprocessor，< 表达式仍然被禁用下推。

```
EXPLAIN SELECT * FROM t WHERE a < 2 AND a > 2;
```

```
+--
    ↪ ------------------------+----------+-----------+---------------+-------------------------------+
    ↪
| id                        | estRows  | task      | access object | operator info
    ↪                  |
+--
    ↪ ------------------------+----------+-----------+---------------+-------------------------------+
    ↪
| Selection_8               | 0.00     | root      |               | lt(ssb_1.t.a, 2)
    ↪               |
| └─TableReader_7           | 0.00     | root      |               | data:Selection_6
    ↪               |
|   └─Selection_6           | 0.00     | cop[tikv] |               | gt(ssb_1.t.a, 2)
    ↪               |
|     └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t       | keep order:false,
    ↪ stats:pseudo |
+--
    ↪ ------------------------+----------+-----------+---------------+-------------------------------+
    ↪
4 rows in set (0.00 sec)
```

10 教程

10.1 同城多数据中心部署 TiDB

作为 NewSQL 数据库，TiDB 兼顾了传统关系型数据库的优秀特性、NoSQL 数据库可扩展性以及跨数据中心场景下的高可用。本文档旨在介绍同城多数据中心部署 TiDB 方案。

10.1.1 了解 Raft 协议

Raft 是一种分布式一致性算法，在 TiDB 集群的多种组件中，PD 和 TiKV 都通过 Raft 实现了数据的容灾。Raft 的灾难恢复能力通过如下机制实现：

- Raft 成员的本质是日志复制和状态机。Raft 成员之间通过复制日志来实现数据同步；Raft 成员在不同条件下切换自己的成员状态，其目标是选出 leader 以提供对外服务。
- Raft 是一个表决系统，它遵循多数派协议，在一个 Raft Group 中，某成员获得大多数投票，它的成员状态就会转变为 leader。也就是说，当一个 Raft Group 还保有大多数节点 (majority) 时，它就能够选出 leader 以提供对外服务。

遵循 Raft 可靠性的特点，放到现实场景中：

- 想克服任意 1 台服务器 (host) 的故障，应至少提供 3 台服务器。
- 想克服任意 1 个机柜 (rack) 的故障，应至少提供 3 个机柜。
- 想克服任意 1 个数据中心（dc，又称机房）的故障，应至少提供 3 个数据中心。
- 想应对任意 1 个城市的灾难场景，应至少规划 3 个城市用于部署。

可见，原生 Raft 协议对于偶数副本的支持并不是很友好，考虑跨城网络延迟影响，或许同城三数据中心是最适合部署 Raft 的高可用及容灾方案。

10.1.2 同城三数据中心方案

同城三数据中心方案，即同城存有三个机房部署 TiDB 集群，同城三数据中心间的数据同步通过集群自身内部（Raft 协议）完成。同城三数据中心可同时对外进行读写服务，任意中心发生故障不影响数据一致性。

10.1.2.1 简易架构图

集群 TiDB、TiKV 和 PD 组件分别分布在 3 个不同的数据中心，这是最常规且高可用性最高的方案。

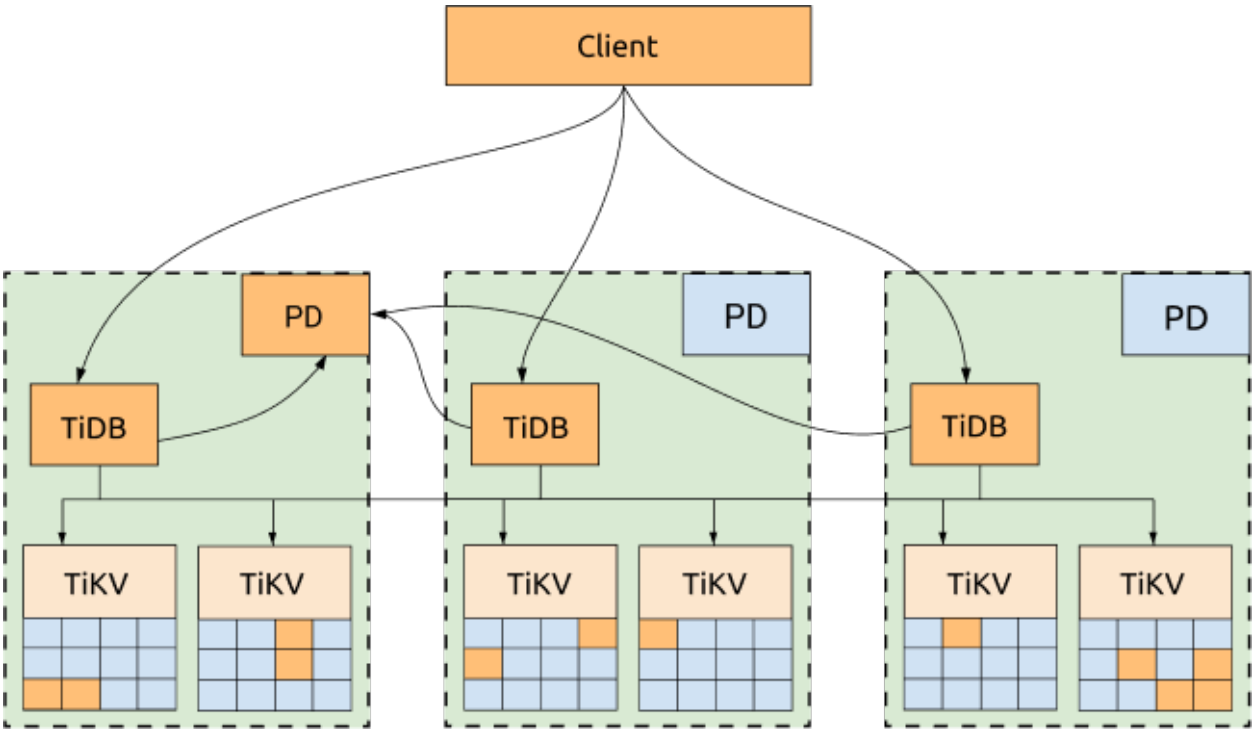


图 88: 三中心部署

优点：

- 所有数据的副本分布在三个数据中心，具备高可用和容灾能力
- 任何一个数据中心失效后，不会产生任何数据丢失 (RPO = 0)
- 任何一个数据中心失效后，其他两个数据中心会自动发起 leader election，并在合理长的时间内（通常情况 20s 以内）自动恢复服务

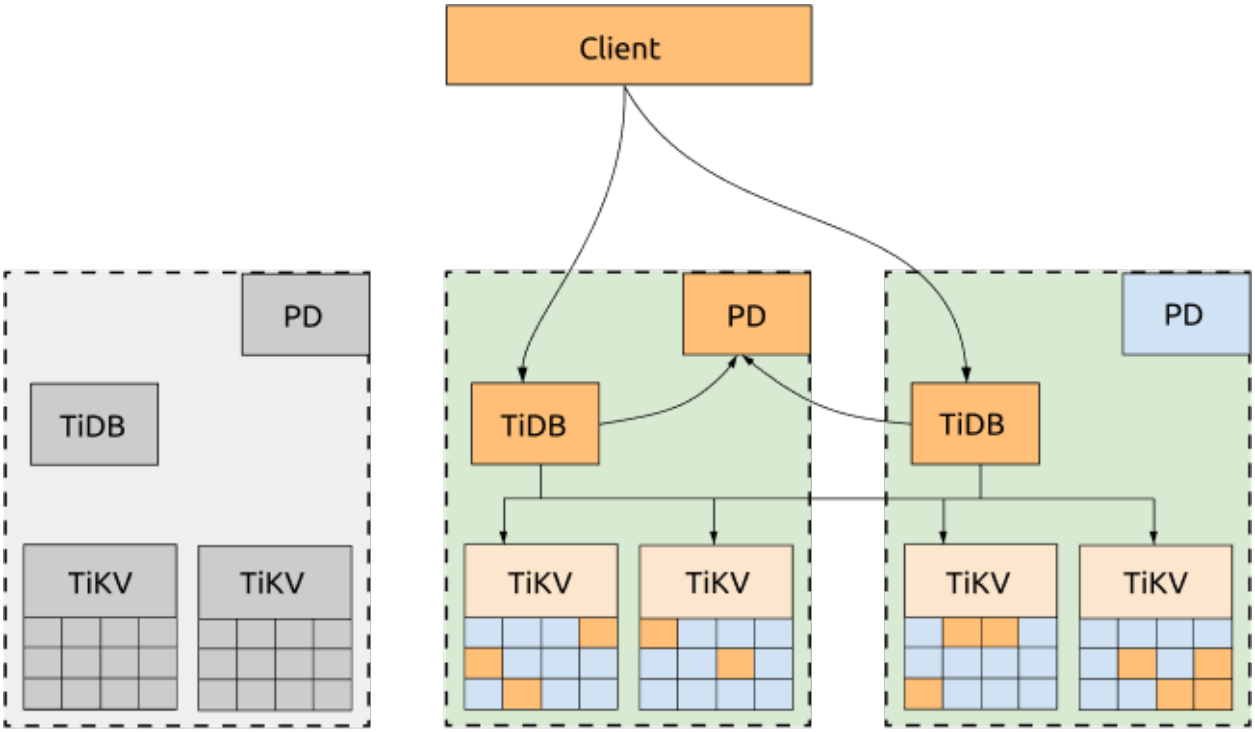


图 89: 三中心部署容灾

缺点：

性能受网络延迟影响。具体影响如下：

- 对于写入的场景，所有写入的数据需要同步复制到至少 2 个数据中心，由于 TiDB 写入过程使用两阶段提交，故写入延迟至少需要 2 倍数据中心间的延迟。
- 对于读请求来说，如果数据 leader 与发起读取的 TiDB 节点不在同一个数据中心，也会受网络延迟影响。
- TiDB 中的每个事务都需要向 PD leader 获取 TSO，当 TiDB 与 PD leader 不在同一个数据中心时，它上面运行的事务也会因此受网络延迟影响，每个有写入的事务会获取两次 TSO。

10.1.2.2 架构优化图

如果不需要每个数据中心同时对外提供服务，可以将业务流量全部派发到一个数据中心，并通过调度策略把 Region leader 和 PD leader 都迁移到同一个数据中心。这样一来，不管是从 PD 获取 TSO，还是读取 Region，都不会受数据中心间网络的影响。当该数据中心失效后，PD leader 和 Region leader 会自动在其它数据中心选出，只需要把业务流量转移至其他存活的数据中心即可。

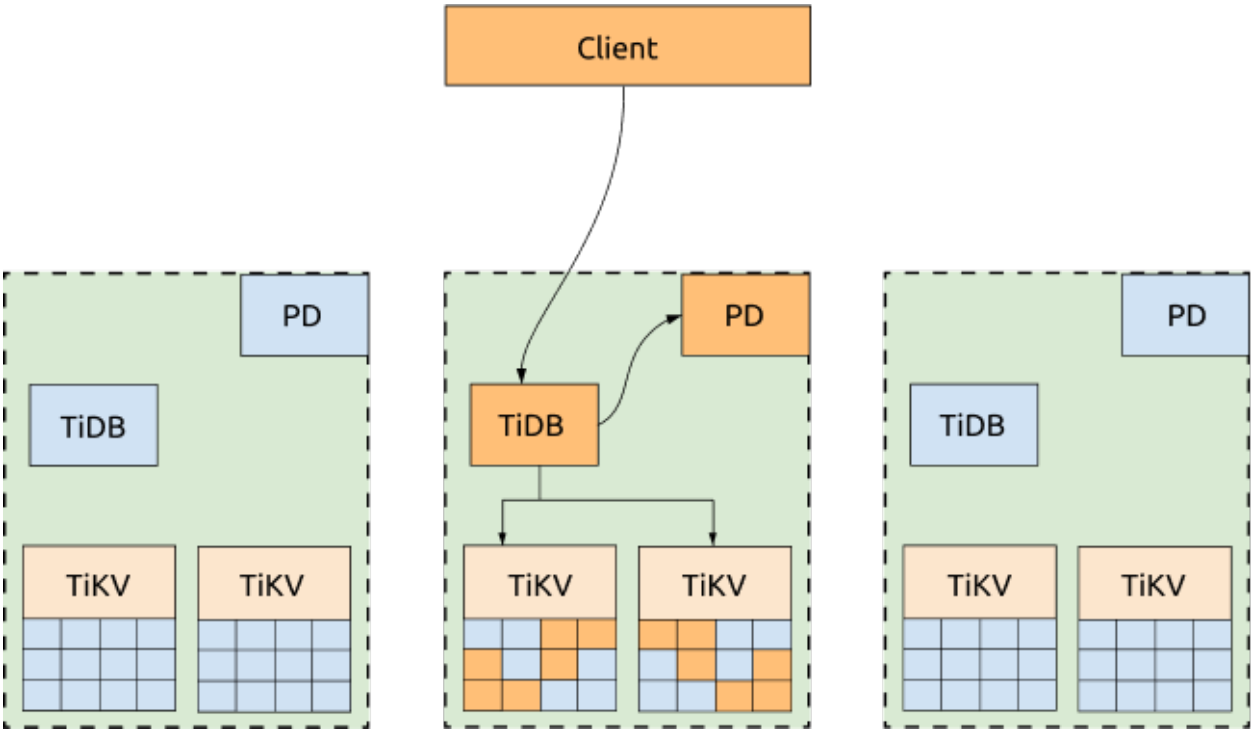


图 90: 三中心部署读性能优化

优点：

集群 TSO 获取能力以及读取性能有所提升。具体调度策略设置模板参照如下：

```
-- 其他机房统一驱逐 leader 到业务流量机房
config set label-property reject-leader LabelName labelValue

-- 迁移 PD leader 并设置优先级
member leader transfer pdName1
member leader_priority pdName1 5
member leader_priority pdName2 4
member leader_priority pdName3 3
```

缺点：

- 写入场景仍受数据中心网络延迟影响，这是因为遵循 Raft 多数派协议，所有写入的数据需要同步复制到至少 2 个数据中心
- TiDB Server 数据中心级别单点
- 业务流量纯走单数据中心，性能受限于单数据中心网络带宽压力
- TSO 获取能力以及读取性能受限于业务流量数据中心集群 PD、TiKV 组件是否正常，否则仍受跨数据中心网络交互影响

10.1.2.3 样例部署图

10.1.2.3.1 样例拓扑架构

下面假设某城存有 IDC1、IDC2、IDC3 三机房，机房 IDC 中存有两套机架，每个机架存有三台服务器，不考虑混布以及单台机器多实例部署下，同城三数据中心架构集群（3 副本）部署参考如下：

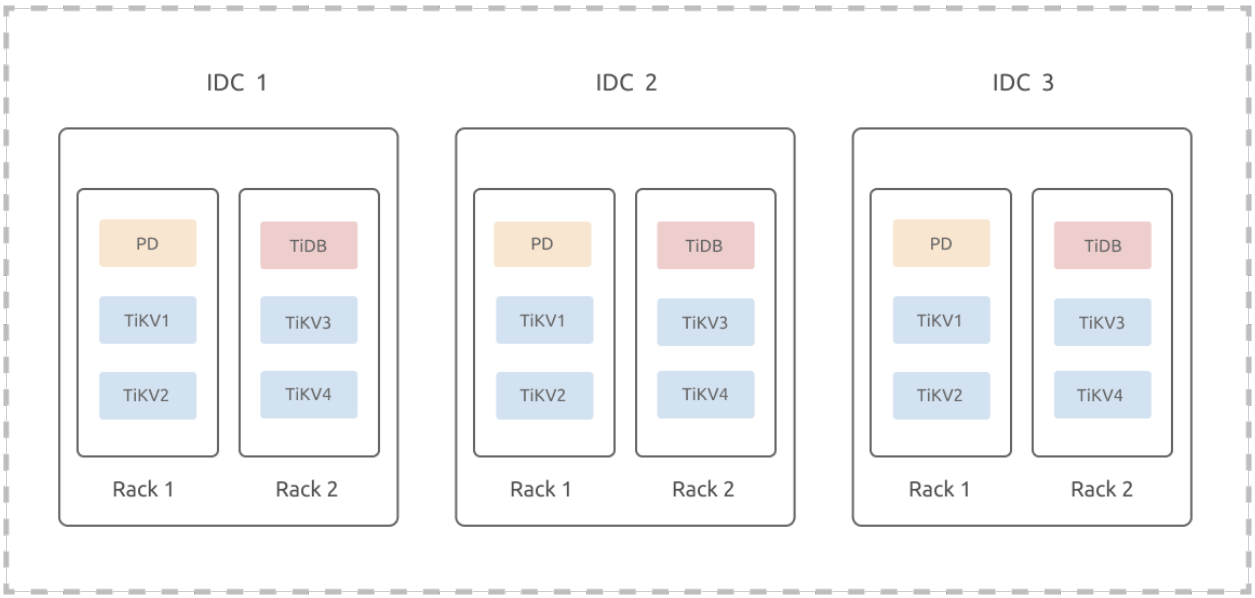


图 91: 同城三数据中心集群部署

10.1.2.3.2 TiKV Labels 简介

TiKV 是一个 Multi-Raft 系统，其数据按 Region（默认 96M）切分，每个 Region 的 3 个副本构成了一个 Raft Group。假设一个 3 副本 TiDB 集群，由于 Region 的副本数与 TiKV 实例数量无关，则一个 Region 的 3 个副本只会被调度到其中 3 个 TiKV 实例上，也就是说即使集群扩容 N 个 TiKV 实例，其本质仍是一个 3 副本集群。

由于 3 副本的 Raft Group 只能容忍 1 副本故障，当集群被扩容到 N 个 TiKV 实例时，这个集群依然只能容忍一个 TiKV 实例的故障。2 个 TiKV 实例的故障可能会导致某些 Region 丢失多个副本，整个集群的数据也不再完整，访问到这些 Region 上的数据的 SQL 请求将会失败。而 N 个 TiKV 实例中同时有两个发生故障的概率是远远高于 3 个 TiKV 中同时有两个发生故障的概率的，也就是说 Multi-Raft 系统集群扩容 TiKV 实例越多，其可用性是逐渐降低的。

正因为 Multi-Raft TiKV 系统局限性，Labels 标签应运而出，其主要用于描述 TiKV 的位置信息。Label 信息随着部署或滚动更新操作刷新到 TiKV 的启动配置文件中，启动后的 TiKV 会将自己最新的 Label 信息上报给 PD，PD 根据用户登记的 Label 名称（也就是 Label 元信息），结合 TiKV 的拓扑进行 Region 副本的最优调度，从而提高系统可用性。

10.1.2.3.3 TiKV Labels 样例规划

针对 TiKV Labels 标签，你需要根据已有的物理资源、容灾能力容忍度等方面进行设计与规划，进而提升系统的可用性和容灾能力。并根据已规划的拓扑架构，在集群初始化配置文件中进行配置（此处省略其他非重点项）：

```
server_configs:
```

```
  pd:
    replication.location-labels: ["zone","dc","rack","host"]

tikv_servers:
  - host: 10.63.10.30
    config:
      server.labels: { zone: "z1", dc: "d1", rack: "r1", host: "30" }
  - host: 10.63.10.31
    config:
      server.labels: { zone: "z1", dc: "d1", rack: "r1", host: "31" }
  - host: 10.63.10.32
    config:
      server.labels: { zone: "z1", dc: "d1", rack: "r2", host: "32" }
  - host: 10.63.10.33
    config:
      server.labels: { zone: "z1", dc: "d1", rack: "r2", host: "33" }

  - host: 10.63.10.34
    config:
      server.labels: { zone: "z2", dc: "d1", rack: "r1", host: "34" }
  - host: 10.63.10.35
    config:
      server.labels: { zone: "z2", dc: "d1", rack: "r1", host: "35" }
  - host: 10.63.10.36
    config:
      server.labels: { zone: "z2", dc: "d1", rack: "r2", host: "36" }
  - host: 10.63.10.37
    config:
      server.labels: { zone: "z2", dc: "d1", rack: "r2", host: "37" }

  - host: 10.63.10.38
    config:
      server.labels: { zone: "z3", dc: "d1", rack: "r1", host: "38" }
  - host: 10.63.10.39
    config:
      server.labels: { zone: "z3", dc: "d1", rack: "r1", host: "39" }
  - host: 10.63.10.40
    config:
      server.labels: { zone: "z3", dc: "d1", rack: "r2", host: "40" }
  - host: 10.63.10.41
    config:
      server.labels: { zone: "z3", dc: "d1", rack: "r2", host: "41" }
```

本例中，zone 表示逻辑可用区层级，用于控制副本的隔离（当前集群 3 副本）。

不直接采用 dc，rack，host 三层 Label 结构的原因是考虑到将来可能发生 dc（数据中心）的扩容，假设新扩容

的 dc 编号是 d2，d3，d4，则只需在对应可用区下扩容 dc；rack 扩容只需在对应 dc 下扩容即可。

如果直接采用 dc，rack，host 三层 Label 结构，那么扩容 dc 操作可能需重打 Label，TiKV 数据整体需要 Rebalance。

10.1.2.4 高可用和容灾分析

同城多数据中心方案提供的保障是，任意一个数据中心故障时，集群能自动恢复服务，不需要人工介入，并能保证数据一致性。注意，各种调度策略都是用于帮助性能优化的，当发生故障时，调度机制总是第一优先考虑可用性而不是性能。

10.2 两地三中心部署

本文档简要介绍两地三中心部署的架构模型及配置。

10.2.1 简介

两地三中心架构，即生产数据中心、同城灾备中心、异地灾备中心的高可用容灾方案。在这种模式下，两个城市的三个数据中心互联互通，如果一个数据中心发生故障或灾难，其他数据中心可以正常运行并对关键业务或全部业务实现接管。相比同城多中心方案，两地三中心具有跨城级高可用能力，可以应对城市级自然灾害。

TiDB 分布式数据库通过 Raft 算法原生支持两地三中心架构的建设，并保证数据库集群数据的一致性和高可用性。而且因同城数据中心网络延迟相对较小，可以把业务流量同时派发到同城两个数据中心，并通过控制 Region Leader 和 PD Leader 分布实现同城数据中心共同负载业务流量的设计。

10.2.2 架构

本文以北京和西安为例，阐述 TiDB 分布式数据库两地三中心架构的部署模型。

本例中，北京有两个机房 IDC1 和 IDC2，异地西安一个机房 IDC3。北京同城两机房之间网络延迟低于 3 ms，北京与西安之间的网络使用 ISP 专线，延迟约 20 ms。

下图为集群部署架构图，具体如下：

- 集群采用两地三中心部署方式，分别为北京 IDC1，北京 IDC2，西安 IDC3；
- 集群采用 5 副本模式，其中 IDC1 和 IDC2 分别放 2 个副本，IDC3 放 1 个副本；TiKV 按机柜打 Label，既每个机柜上有一份副本。
- 副本间通过 Raft 协议保证数据的一致性和高可用，对用户完全透明。

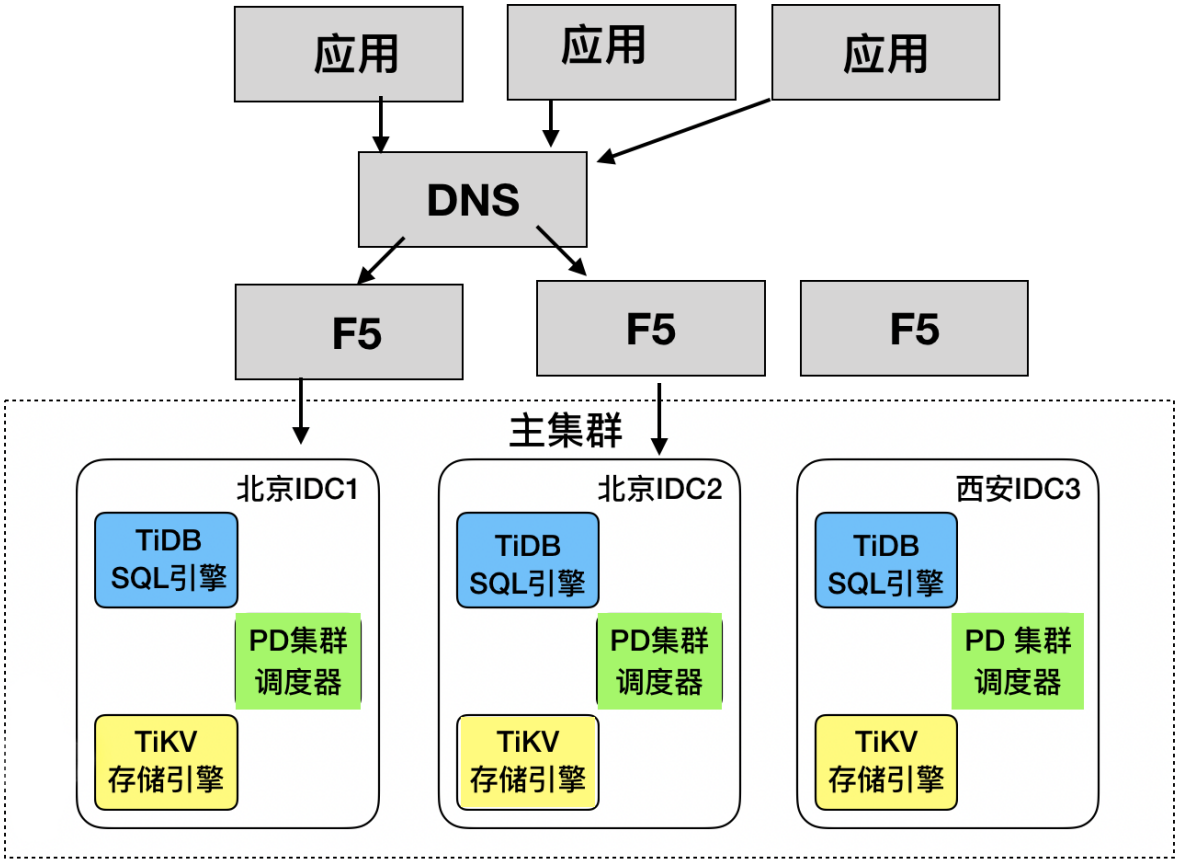


图 92: 两地三中心集群架构图

该架构具备高可用能力，同时通过 PD 调度限制了 Region Leader 尽量只出现在同城的两个数据中心，这相比于三数据中心，即 Region Leader 分布不受限制的方案有以下优缺点：

- 优点
  - Region Leader 都在同城低延迟机房，数据写入速度更优。
  - 两中心可同时对外提供服务，资源利用率更高。
  - 可保证任一数据中心失效后，服务可用并且不发生数据丢失。
- 缺点
  - 因为数据一致性是基于 Raft 算法实现，当同城两个数据中心同时失效时，因为异地灾备中心只剩下一份副本，不满足 Raft 算法大多数副本存活的要求。最终将导致集群暂时不可用，需要从一副本恢复集群，只会丢失少部分还没同步的热数据。这种情况出现的概率是比较小的。
  - 由于使用到了网络专线，导致该架构下网络设施成本较高。
  - 两地三中心需设置 5 副本，数据冗余度增加，增加空间成本。

10.2.2.1 详细示例

北京、西安两地三中心配置详解：

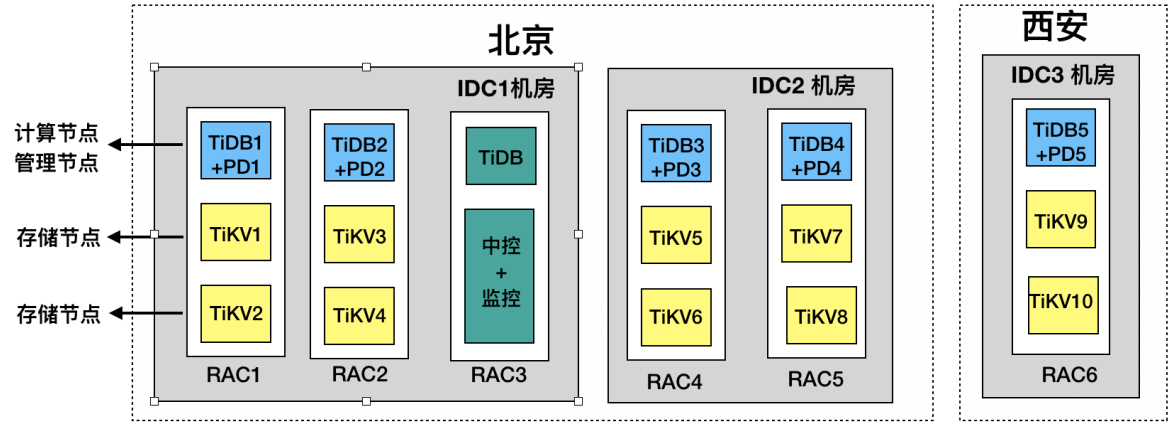


图 93: 两地三中心配置详图

- 如上图所示，北京有两个机房 IDC1 和 IDC2，机房 IDC1 中有三套机架 RAC1、 RAC2、 RAC3，机房 IDC2 有机架 RAC4、 RAC5；西安机房 IDC3 有机架 RAC6。
- 如上图中 RAC1 机架所示，TiDB、 PD 服务部署在同一台服务器上，还有两台 TiKV 服务器；每台 TiKV 服务器部署 2 个 TiKV 实例（tikv-server），RAC2、 RAC4、 RAC5、 RAC6 类似。
- 机架 RAC3 上安放 TiDB Server 及中控 + 监控服务器。部署 TiDB Server，用于日常管理维护、备份使用。中控 + 监控服务器上部署 TiDB Ansible、 Prometheus、 Grafana 以及恢复工具；
- 另可增加备份服务器，其上部署 Mydumper 及 Drainer，Drainer 以输出 file 文件的方式将 binlog 数据保存到指定位置，实现增量备份的目的。

10.2.3 配置

10.2.3.1 示例

以下为一个 `tiup topology.yaml` 文件示例：

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

server_configs:
  tikv:
    server.grpc-compression-type: gzip
  pd:
```

```
    replication.location-labels:  ["dc","rack","zone","host"]
    schedule.tolerant-size-ratio: 20.0

pd_servers:
  - host: 10.63.10.10
    name: "pd-10"
  - host: 10.63.10.11
    name: "pd-11"
  - host: 10.63.10.12
    name: "pd-12"
  - host: 10.63.10.13
    name: "pd-13"
  - host: 10.63.10.14
    name: "pd-14"

tidb_servers:
  - host: 10.63.10.10
  - host: 10.63.10.11
  - host: 10.63.10.12
  - host: 10.63.10.13
  - host: 10.63.10.14

tikv_servers:
  - host: 10.63.10.30
    config:
      server.labels: { dc: "1", zone: "1", rack: "1", host: "30" }
  - host: 10.63.10.31
    config:
      server.labels: { dc: "1", zone: "2", rack: "2", host: "31" }
  - host: 10.63.10.32
    config:
      server.labels: { dc: "2", zone: "3", rack: "3", host: "32" }
  - host: 10.63.10.33
    config:
      server.labels: { dc: "2", zone: "4", rack: "4", host: "33" }
  - host: 10.63.10.34
    config:
      server.labels: { dc: "3", zone: "5", rack: "5", host: "34" }
      raftstore.raft-min-election-timeout-ticks: 1000
      raftstore.raft-max-election-timeout-ticks: 1200

monitoring_servers:
  - host: 10.63.10.60

grafana_servers:
```

```
  - host: 10.63.10.60

alertmanager_servers:
  - host: 10.63.10.60
```

10.2.3.2 Labels 设计

在两地三中心部署方式下，对于 Labels 的设计需要充分考虑到系统的可用性和容灾能力，建议根据部署的物理结构来定义 DC、ZONE、RACK、HOST 四个等级。

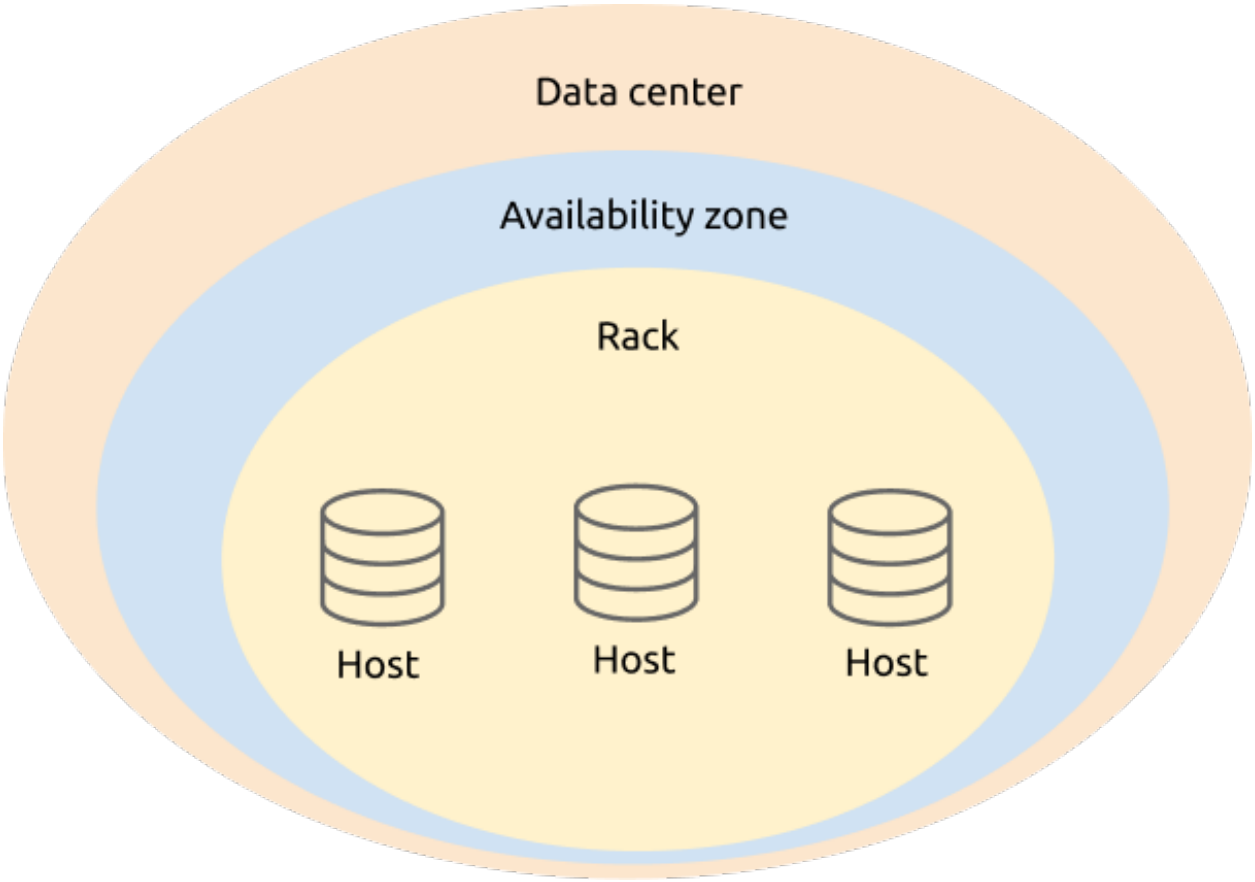


图 94: Label 逻辑定义图

PD 设置中添加 TiKV label 的等级配置。

```
server_configs:
  pd:
    replication.location-labels:  ["dc","zone","rack","host"]
```

tikv_servers 设置基于 TiKV 真实物理部署位置的 Label 信息，方便 PD 进行全局管理和调度。

```
tikv_servers:
```

```
- host: 10.63.10.30
  config:
    server.labels: { dc: "1", zone: "1", rack: "1", host: "30" }
- host: 10.63.10.31
  config:
    server.labels: { dc: "1", zone: "2", rack: "2", host: "31" }
- host: 10.63.10.32
  config:
    server.labels: { dc: "2", zone: "3", rack: "3", host: "32" }
- host: 10.63.10.33
  config:
    server.labels: { dc: "2", zone: "4", rack: "4", host: "33" }
- host: 10.63.10.34
  config:
    server.labels: { dc: "3", zone: "5", rack: "5", host: "34" }
```

10.2.3.3 参数配置优化

在两地三中心的架构部署中，从性能优化的角度，除了常规参数配置外，还需要对集群中相关组件参数进行调整。

- 启用 TiKV gRPC 消息压缩。由于涉及到集群中的数据在网络中传输，需要开启 gRPC 消息压缩，降低网络流量。

```
server.grpc-compression-type: gzip
```

- 调整 PD balance 缓冲区大小，提高 PD 容忍度，因为 PD 会根据节点情况计算出各个对象的 score 作为调度的依据，当两个 store 的 leader 或 Region 的得分差距小于指定倍数的 Region size 时，PD 会认为此时 balance 达到均衡状态。

```
schedule.tolerant-size-ratio: 20.0
```

- 异地 DC3 TiKV 节点网络优化，单独修改异地 TiKV 此参数，拉长异地副本参与选举的时间，尽量避免异地 TiKV 中的副本参与 Raft 选举。

```
raftstore.raft-min-election-timeout-ticks: 1000
raftstore.raft-max-election-timeout-ticks: 1200
```

- 调度设置。在集群启动后，通过 `tiup ctl:<cluster-version> pd` 工具进行调度策略修改。修改 TiKV Raft 副本数按照安装时规划好的副本数进行设置，在本例中为 5 副本。

```
config set max-replicas 5
```

- 禁止向异地机房调度 Raft Leader，当 Raft Leader 在异地数据中心时，会造成不必要的本地数据中心与异地数据中心间的网络消耗，同时由于网络带宽和延迟的影响，也会对 TiDB 的集群性能产生影响。需要禁用异地中心的 Raft leader 的调度。

```
config set label-property reject-leader dc 3
```

- 设置 PD 的优先级，为了避免出现异地数据中心的 PD 成为 Leader，可以将本地数据中心的 PD 优先级调高（数字越大，优先级越高），将异地的 PD 优先级调低。

```
member leader_priority PD-10 5
member leader_priority PD-11 5
member leader_priority PD-12 5
member leader_priority PD-13 5
member leader_priority PD-14 1
```

10.3 最佳实践

10.3.1 TiDB 最佳实践

本文档总结使用 TiDB 时的一些最佳实践，主要涉及 SQL 使用和 OLAP/OLTP 优化技巧，特别是一些 TiDB 专有的优化开关。

建议先阅读讲解 TiDB 原理的三篇文章 (讲存储，说计算，谈调度)，再来看这篇文章。

10.3.1.1 前言

数据库是一个通用的基础组件，在开发过程中会考虑到多种目标场景，在具体的业务场景中，需要根据业务的实际情况对数据的参数或者使用方式进行调整。

TiDB 是一个兼容 MySQL 协议和语法的分布式数据库，但是由于其内部实现，特别是支持分布式存储以及分布式事务，使得一些使用方法和 MySQL 有所区别。

10.3.1.2 基本概念

TiDB 的最佳实践与其实现原理密切相关，建议读者先了解一些基本的实现机制，包括 Raft、分布式事务、数据分片、负载均衡、SQL 到 KV 的映射方案、二级索引的实现方法、分布式执行引擎。下面会做一点简单的介绍，更详细的信息可以参考 PingCAP 公众号以及知乎专栏的一些文章。

10.3.1.2.1 Raft

Raft 是一种一致性协议，能提供强一致的数据复制保证，TiDB 最底层用 Raft 来同步数据。每次写入都要写入多数副本，才能对外返回成功，这样即使丢掉少数副本，也能保证系统中还有最新的数据。比如最大 3 副本的话，每次写入 2 副本才算成功，任何时候，只丢失一个副本的情况下，存活的两个副本中至少有一个具有最新的数据。

相比 Master-Slave 方式的同步，同样是保存三副本，Raft 的方式更为高效，写入的延迟取决于最快的两个副本，而不是最慢的那个副本。所以使用 Raft 同步的情况下，异地多活成为可能。在典型的两地三中心场景下，每次写入只需要本数据中心以及离得近的一个数据中心写入成功就能保证数据的一致性，而并不需要三个数据中心都写成功。但是这并不意味着在任何场景都能构建跨机房部署的业务，当写入量比较大时候，机房之间的带宽和延迟成为关键因素，如果写入速度超过机房之间的带宽，或者是机房之间延迟过大，整个 Raft 同步机制依然无法很好的运转。

10.3.1.2.2 分布式事务

TiDB 提供完整的分布式事务，事务模型是在 Google Percolator 的基础上做了一些优化。具体的实现可以参考《Percolator 和 TiDB 事务算法》这篇文章。本文档只讨论以下几点：

- 乐观锁

  TiDB 的乐观事务模型，只有在真正提交的时候，才会做冲突检测。如果有冲突，则需要重试。这种模型在冲突严重的场景下，会比较低效，因为重试之前的操作都是无效的，需要重复做。举一个比较极端的例子，就是把数据库当做计数器用，如果访问的并发度比较高，那么一定会有严重的冲突，导致大量的重试甚至是超时。但是如果访问冲突并不十分严重，那么乐观锁模型具备较高的效率。在冲突严重的场景下，推荐使用悲观锁，或在系统架构层面解决问题，比如将计数器放在 Redis 中。

- 悲观锁

  TiDB 的悲观事务模型，悲观事务的行为和 MySQL 基本一致，在执行阶段就会上锁，先到先得，避免冲突情况下的重试，可以保证有较多冲突的事务的成功率。悲观锁同时解决了希望通过 `select for update` 对数据提前锁定的场景。但如果业务场景本身冲突较少，乐观锁的性能会更有优势。

- 事务大小限制

  由于分布式事务要做两阶段提交，并且底层还需要做 Raft 复制，如果一个事务非常大，会使得提交过程非常慢，并且会卡住下面的 Raft 复制流程。为了避免系统出现被卡住的情况，我们对事务的大小做了限制：

  - 单个事务包含的 SQL 语句不超过 5000 条（默认）
    * 单条 KV entry 不超过 6MB（默认）
    * KV entry 的总大小不超过 10G

  在 Google 的 Cloud Spanner 上面，也有类似的限制。

10.3.1.2.3 数据分片

TiKV 自动将底层数据按照 Key 的 Range 进行分片。每个 Region 是一个 Key 的范围，从 `StartKey` 到 EndKey 的左闭右开区间。Region 中的 Key-Value 总量超过一定值，就会自动分裂。这部分用户不需要担心。

10.3.1.2.4 负载均衡

PD 会根据整个 TiKV 集群的状态，对集群的负载进行调度。调度是以 Region 为单位，以 PD 配置的策略为调度逻辑，自动完成。

10.3.1.2.5 SQL on KV

TiDB 自动将 SQL 结构映射为 KV 结构。具体的可以参考《三篇文章了解 TiDB 技术内幕 - 说计算》这篇文档。简单来说，TiDB 执行了以下操作：

- 一行数据映射为一个 KV，Key 以 `TableID` 构造前缀，以行 ID 为后缀
- 一条索引映射为一个 KV，Key 以 `TableID+IndexID` 构造前缀，以索引值构造后缀

可以看到，对于一个表中的数据或者索引，会具有相同的前缀，这样在 TiKV 的 Key 空间内，这些 Key-Value 会在相邻的位置。那么当写入量很大，并且集中在一个表上面时，就会造成写入的热点，特别是连续写入的数据中某些索引值也是连续的 (比如 update time 这种按时间递增的字段)，会在很少的几个 Region 上形成写入热点，成为整个系统的瓶颈。同样，如果所有的数据读取操作也都集中在很小的一个范围内 (比如在连续的几万或者十几万行数据上)，那么可能造成数据的访问热点。

10.3.1.2.6 二级索引

TiDB 支持完整的二级索引，并且是全局索引，很多查询可以通过索引来优化。如果利用好二级索引，对业务非常重要，很多 MySQL 上的经验在 TiDB 这里依然适用，不过 TiDB 还有一些自己的特点，需要注意，这一节主要讨论在 TiDB 上使用二级索引的一些注意事项。

- 二级索引是否越多越好

  二级索引能加速查询，但是要注意新增一个索引是有副作用的，上一节介绍了索引的存储模型，那么每增加一个索引，在插入一条数据的时候，就要新增一个 Key-Value，所以索引越多，写入越慢，并且空间占用越大。另外过多的索引也会影响优化器运行时间，并且不合适的索引会误导优化器。所以索引并不是越多越好。

- 对哪些列建索引比较合适

  上文提到，索引很重要但不是越多越好，因此需要根据具体的业务特点创建合适的索引。原则上需要对查询中需要用到的列创建索引，目的是提高性能。下面几种情况适合创建索引：

  - 区分度比较大的列，通过索引能显著地减少过滤后的行数
  - 有多个查询条件时，可以选择组合索引，注意需要把等值条件的列放在组合索引的前面

  这里举一个例子，假设常用的查询是 `select * from t where c1 = 10 and c2 = 100 and c3 > 10`，那么可以考虑建立组合索引 `Index cidx (c1, c2, c3)`，这样可以用查询条件构造出一个索引前缀进行 Scan。

- 通过索引查询和直接扫描 Table 的区别

  TiDB 实现了全局索引，所以索引和 Table 中的数据并不一定在一个数据分片上，通过索引查询的时候，需要先扫描索引，得到对应的行 ID，然后通过行 ID 去取数据，所以可能会涉及到两次网络请求，会有一定的性能开销。

  如果查询涉及到大量的行，那么扫描索引是并发进行，只要第一批结果已经返回，就可以开始去取 Table 的数据，所以这里是一个并行 + Pipeline 的模式，虽然有两次访问的开销，但是延迟并不会很大。

  以下情况不会涉及到两次访问的问题：

  - 索引中的列已经满足了查询需求。比如 Table `t` 上面的列 `c` 有索引，查询是 `select c from t where c > 10;`，这个时候，只需要访问索引，就可以拿到所需要的全部数据。这种情况称之为覆盖索引 (Covering Index)。所以如果很关注查询性能，可以将部分不需要过滤但是需要在查询结果中返回的列放入索引中，构造成组合索引，比如这个例子：`select c1, c2 from t where c1 > 10;`，要优化这个查询可以创建组合索引 `Index c12 (c1, c2)`。
  - 表的 Primary Key 是整数类型。在这种情况下，TiDB 会将 Primary Key 的值当做行 ID，所以如果查询条件是在 PK 上面，那么可以直接构造出行 ID 的范围，直接扫描 Table 数据，获取结果。

- 查询并发度

数据分散在很多 Region 上，所以 TiDB 在做查询的时候会并发进行，默认的并发度比较保守，因为过高的并发度会消耗大量的系统资源，且对于 OLTP 类型的查询，往往不会涉及到大量的数据，较低的并发度已经可以满足需求。对于 OLAP 类型的 Query，往往需要较高的并发度。所以 TiDB 支持通过 System Variable 来调整查询并发度。

- tidb_distsql_scan_concurrency

在进行扫描数据的时候的并发度，这里包括扫描 Table 以及索引数据。

- tidb_index_lookup_size

如果是需要访问索引获取行 ID 之后再访问 Table 数据，那么每次会把一批行 ID 作为一次请求去访问 Table 数据，这个参数可以设置 Batch 的大小，较大的 Batch 会使得延迟增加，较小的 Batch 可能会造成更多的查询次数。这个参数的合适大小与查询涉及的数据量有关。一般不需要调整。

- tidb_index_lookup_concurrency

如果是需要访问索引获取行 ID 之后再访问 Table 数据，每次通过行 ID 获取数据时候的并发度通过这个参数调节。

- 通过索引保证结果顺序

  索引除了可以用来过滤数据之外，还能用来对数据排序，首先按照索引的顺序获取行 ID，然后再按照行 ID 的返回顺序返回行的内容，这样可以保证返回结果按照索引列有序。前面提到了扫索引和获取 Row 之间是并行 + Pipeline 模式，如果要求按照索引的顺序返回 Row，那么这两次查询之间的并发度设置的太高并不会降低延迟，所以默认的并发度比较保守。可以通过tidb_index_serial_scan_concurrency 变量进行并发度调整。

- 逆序索引

  目前 TiDB 支持对索引进行逆序 Scan，目前速度比顺序 Scan 慢一些，通常情况下慢 20%，在数据频繁修改造成版本较多的情况下，会慢的更多。如果可能，建议避免对索引的逆序 Scan。

10.3.1.3 场景与实践

上一节我们讨论了一些 TiDB 基本的实现机制及其对使用带来的影响，本节我们从具体的使用场景出发，谈一些更为具体的操作实践。我们以从部署到支撑业务这条链路为序，进行讨论。

10.3.1.3.1 部署

在部署之前请务必阅读TiDB 部署建议以及对硬件的需求。

推荐通过TiUP 部署 TiDB 集群，这个工具可以部署、停止、销毁、升级整个集群，非常方便易用。非常不推荐手动部署，后期的维护和升级会很麻烦。

10.3.1.3.2 导入数据

为了提高导入数据期间的写入性能，可以对 TiKV 的参数进行调优，具体的文档查看TiKV 性能参数调优。

10.3.1.3.3 写入

上面提到了 TiDB 对单个事务的大小有限制，这层限制是在 KV 层面，反映在 SQL 层面的话，简单来说一行数据会映射为一个 KV entry，每多一个索引，也会增加一个 KV entry。

> 注意：：
>
> 对事务的大小限制，要考虑 TiDB 做编码以及事务额外 Key 的开销，在使用的时候，建议每个事务的行数不超过 200 行，且单行数据小于 100k，否则可能性能不佳。

建议无论是 Insert，Update 还是 Delete 语句，都通过分 Batch 或者是加 Limit 的方式限制。

在删除大量数据的时候，建议使用 `Delete * from t where xx limit 5000;` 这样的方案，通过循环来删除，用 `Affected Rows == 0` 作为循环结束条件。

如果一次删除的数据量非常大，这种循环的方式会越来越慢，因为每次删除都是从前向后遍历，前面的删除之后，短时间内会残留不少删除标记（后续会被 GC 清理掉），影响后面的 `Delete` 语句。如果有可能，建议把 `Where` 条件细化。举个例子，假设要删除 2017-05-26 当天的所有数据，那么可以这样做：

```
for i from 0 to 23:
    while affected_rows > 0:
        delete * from t where insert_time >= i:00:00 and insert_time < (i+1):00:00 limit 5000;
        affected_rows = select affected_rows()
```

上面是一段伪代码，意思就是要把大块的数据拆成小块删除，以避免删除过程中前面的 Delete 语句影响后面的 Delete 语句。

10.3.1.3.4 查询

看业务的查询需求以及具体的语句，可以参考TiDB 专用系统变量和语法这篇文档。可以通过 SET 语句控制 SQL 执行的并发度，另外通过 Hint 控制 Join 物理算子选择。

另外 MySQL 标准的索引选择 Hint 语法，也可以用，通过 `Use Index/Ignore Index hint` 控制优化器选择索引。

如果是个 OLTP 和 OLAP 混合类型的业务，可以把 TP 请求和 AP 请求发送到不同的 tidb-server 上，这样能够减小 AP 业务对于 TP 业务的影响。承载 AP 业务的 tidb-server 推荐使用高配的机器，比如 CPU 核数比较多，内存比较大。

但彻底的隔离 OLTP 和 OLAP，推荐将 OLAP 的业务跑在 TiFlash 上。TiFlash 是列存引擎，在 OLAP 的分析查询场景上，性能极具亮点，TiFlash 可以在存储层上做到物理隔离，并可做到一致性读取。

10.3.1.3.5 监控和日志

Metrics 系统是了解系统状态的最佳方法，建议所有的用户都部署监控系统。

TiDB 使用 Grafana + Prometheus 监控系统状态。如果使用 TiUP 部署集群，那么会自动部署和配置监控系统。

监控系统中的监控项很多，大部分是给 TiDB 开发者查看的内容，如果没有对源代码比较深入的了解，并没有必要了解这些监控项。我们会精简出一些和业务相关或者是系统关键组件状态相关的监控项，放在一个独立的 `overview` 面板中，供用户使用。

除了监控之外，查看日志也是了解系统状态的常用方法。TiDB 的三个组件 tidb-server/tikv-server/pd-server 都有一个 `--log-file` 的参数。如果启动的时候设置了这个参数，那么日志会保存着参数所设置的文件的位置，另外会自动的按天对 Log 文件做归档。如果没有设置 `--log-file` 参数，日志会输出在 `stderr` 中。

从 4.0 版本开始，从解决易用性的角度出发，提供了TiDB Dashboard UI 系统，通过浏览器访问 `http://PD_IP:PD_PORT/dashboard` 即可打开 TiDB Dashboard。TiDB Dashboard 可以提供集群状态、性能分析、流量可视化、SQL 诊断、日志搜索等功能。

10.3.1.3.6 文档

了解一个系统或者解决使用中的问题最好的方法是阅读文档，明白实现原理。TiDB 有大量的官方文档，希望大家在遇到问题的时候能先尝试通过文档或者搜索 Issue list 寻找解决方案。官方文档查看 docs-cn。如果希望阅读英文文档，可以查看 docs。

其中的FAQ 和故障诊断章节建议大家仔细阅读。另外 TiDB 还有一些不错的工具，也有配套的文档，具体的见各项工具的 GitHub 页面。

除了文档之外，还有很多不错的文章介绍 TiDB 的各项技术细节内幕，大家可以关注下面这些文章发布渠道：

- 公众号：微信搜索 PingCAP
- 知乎专栏：TiDB 的后花园
- 官方博客

10.3.1.4 TiDB 的最佳适用场景

简单来说，TiDB 适合具备下面这些特点的场景：

- 数据量大，单机保存不下
- 不希望做 Sharding 或者懒得做 Sharding
- 访问模式上没有明显的热点
- 需要事务、需要强一致、需要灾备
- 希望 Real-Time HTAP，减少存储链路

10.3.2 开发 Java 应用使用 TiDB 的最佳实践

本文主要介绍如何开发 Java 应用程序以更好地使用 TiDB，包括开发中的常见问题与最佳实践。

10.3.2.1 Java 应用中的数据库相关组件

通常 Java 应用中和数据库相关的常用组件有：

- 网络协议：客户端通过标准 MySQL 协议和 TiDB 进行网络交互。
- JDBC API 及实现：Java 应用通常使用 JDBC (Java Database Connectivity) 来访问数据库。JDBC 定义了访问数据库 API，而 JDBC 实现完成标准 API 到 MySQL 协议的转换，常见的 JDBC 实现是 MySQL Connector/J，此外有些用户可能使用 MariaDB Connector/J。
- 数据库连接池：为了避免每次创建连接，通常应用会选择使用数据库连接池来复用连接，JDBC DataSource 定义了连接池 API，开发者可根据实际需求选择使用某种开源连接池实现。

- 数据访问框架：应用通常选择通过数据访问框架 (MyBatis, Hibernate) 的封装来进一步简化和管理数据库访问操作。
- 业务实现：业务逻辑控制着何时发送和发送什么指令到数据库，其中有些业务会使用 Spring Transaction 切面来控制管理事务的开始和提交逻辑。

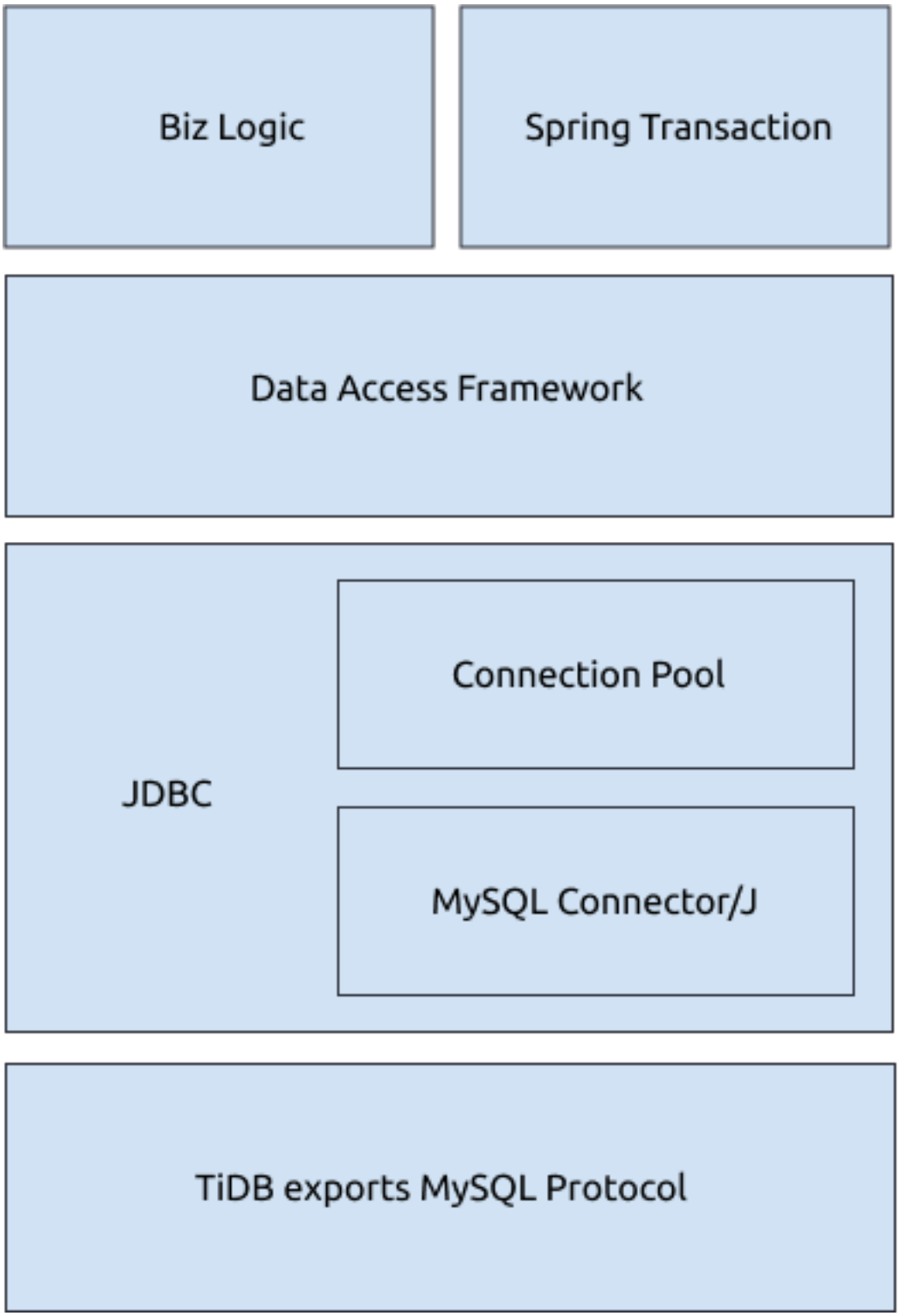


图 95: Java Component

如上图所示，应用可能使用 Spring Transaction 来管理控制事务非手工启停，通过类似 MyBatis 的数据访问框架管理生成和执行 SQL，通过连接池获取已池化的长连接，最后通过 JDBC 接口调用实现通过 MySQL 协议和 TiDB

完成交互。

接下来将分别介绍使用各个组件时可能需要关注的问题。

10.3.2.2 JDBC

Java 应用尽管可以选择在不同的框架中封装，但在最底层一般会通过调用 JDBC 来与数据库服务器进行交互。对于 JDBC，需要关注的主要有：API 的使用选择和 API Implementer 的参数配置。

10.3.2.2.1 JDBC API

对于基本的 JDBC API 使用可以参考 JDBC 官方教程，本文主要强调几个比较重要的 API 选择。

使用 Prepare API

对于 OLTP 场景，程序发送给数据库的 SQL 语句在去除参数变化后都是可穷举的某几类，因此建议使用预处理语句 (Prepared Statements) 代替普通的文本执行，并复用预处理语句来直接执行，从而避免 TiDB 重复解析和生成 SQL 执行计划的开销。

目前多数上层框架都会调用 Prepare API 进行 SQL 执行，如果直接使用 JDBC API 进行开发，注意选择使用 Prepare API。

另外需要注意 MySQL Connector/J 实现中默认只会做客户端的语句预处理，会将 ? 在客户端替换后以文本形式发送到服务端，所以除了要使用 Prepare API，还需要在 JDBC 连接参数中配置 `useServerPrepStmts = true`，才能在 TiDB 服务器端进行语句预处理（下面参数配置章节有详细介绍）。

使用 Batch 批量插入更新

对于批量插入更新，如果插入记录较多，可以选择使用 addBatch/executeBatch API。通过 addBatch 的方式将多条 SQL 的插入更新记录先缓存在客户端，然后在 executeBatch 时一起发送到数据库服务器。

> 注意：
>
> 对于 MySQL Connector/J 实现，默认 Batch 只是将多次 addBatch 的 SQL 发送时机延迟到调用 executeBatch 的时候，但实际网络发送还是会一条条的发送，通常不会降低与数据库服务器的网络交互次数。
>
> 如果希望 Batch 网络发送，需要在 JDBC 连接参数中配置 `rewriteBatchedStatements = true`（下面参数配置章节有详细介绍）。

使用 StreamingResult 流式获取执行结果

一般情况下，为提升执行效率，JDBC 会默认提前获取查询结果并将其保存在客户端内存中。但在查询返回超大结果集的场景中，客户端会希望数据库服务器减少向客户端一次返回的记录数，等客户端在有限内存处理完一部分后再去向服务器要下一批。

在 JDBC 中通常有以下两种处理方式：

- 设置 `FetchSize` 为 `Integer.MIN_VALUE` 让客户端不缓存，客户端通过 StreamingResult 的方式从网络连接上流式读取执行结果。

- 使用 Cursor Fetch，首先需设置 `FetchSize` 为正整数，且在 JDBC URL 中配置 `useCursorFetch = true`。

TiDB 中同时支持两种方式，但更推荐使用第一种将 `FetchSize` 设置为 `Integer.MIN_VALUE` 的方式，比第二种功能实现更简单且执行效率更高。

10.3.2.2.2 MySQL JDBC 参数

JDBC 实现通常通过 JDBC URL 参数的形式来提供实现相关的配置。这里以 MySQL 官方的 Connector/J 来介绍参数配置（如果使用的是 MariaDB，可以参考 MariaDB 的类似配置）。因为配置项较多，这里主要关注几个可能影响到性能的参数。

Prepare 相关参数

`useServerPrepStmts`

默认情况下，`useServerPrepStmts` 的值为 `false`，即尽管使用了 Prepare API，也只会在客户端做“prepare”。因此为了避免服务器重复解析的开销，如果同一条 SQL 语句需要多次使用 Prepare API，则建议设置该选项为 `true`。

在 TiDB 监控中可以通过 Query Summary > QPS By Instance 查看请求命令类型，如果请求中 `COM_QUERY` 被 `COM_STMT_EXECUTE` 或 `COM_STMT_PREPARE` 代替即生效。

`cachePrepStmts`

虽然 `useServerPrepStmts = true` 能让服务端执行预处理语句，但默认情况下客户端每次执行完后会 close 预处理语句，并不会复用，这样预处理的效率甚至不如文本执行。所以建议开启 `useServerPrepStmts = true` 后同时配置 `cachePrepStmts = true`，这会让客户端缓存预处理语句。

在 TiDB 监控中可以通过 Query Summary > QPS By Instance 查看请求命令类型，如果类似下图，请求中 `COM_STMT_EXECUTE` 数目远远多于 `COM_STMT_PREPARE` 即生效。

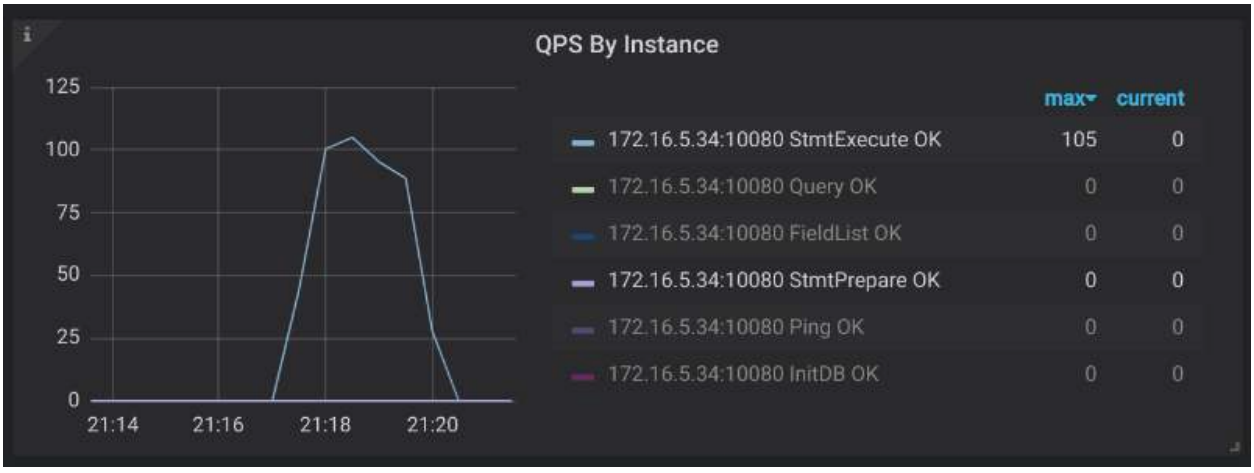


图 96: QPS By Instance

另外，通过 `useConfigs = maxPerformance` 配置会同时配置多个参数，其中也包括 `cachePrepStmts = true`。

`prepStmtCacheSqlLimit`

在配置 cachePrepStmts 后还需要注意 prepStmtCacheSqlLimit 配置（默认为 256），该配置控制客户端缓存预处理语句的最大长度，超过该长度将不会被缓存。

在一些场景 SQL 的长度可能超过该配置，导致预处理 SQL 不能复用，建议根据应用 SQL 长度情况决定是否需要调大该值。

在 TiDB 监控中通过 Query Summary > QPS by Instance 查看请求命令类型，如果已经配置了 cachePrepStmts = true，但 COM_STMT_PREPARE 还是和 COM_STMT_EXECUTE 基本相等且有 COM_STMT_CLOSE，需要检查这个配置项是否设置得太小。

prepStmtCacheSize

prepStmtCacheSize 控制缓存的预处理语句数目（默认为 25），如果应用需要预处理的 SQL 种类很多且希望复用预处理语句，可以调大该值。

和上一条类似，在监控中通过 Query Summary > QPS by Instance 查看请求中 COM_STMT_EXECUTE 数目是否远远多于 COM_STMT_PREPARE 来确认是否正常。

Batch 相关参数

在进行 batch 写入处理时推荐配置 rewriteBatchedStatements = true，在已经使用 addBatch 或 executeBatch 后默认 JDBC 还是会一条条 SQL 发送，例如：

```
pstmt = prepare(“insert into t (a) values(?)”);
pstmt.setInt(1, 10);
pstmt.addBatch();
pstmt.setInt(1, 11);
pstmt.addBatch();
pstmt.setInt(1, 12);
pstmt.executeBatch();
```

虽然使用了 batch 但发送到 TiDB 语句还是单独的多条 insert：

```
insert into t(a) values(10);
insert into t(a) values(11);
insert into t(a) values(12);
```

如果设置 rewriteBatchedStatements = true，发送到 TiDB 的 SQL 将是：

```
insert into t(a) values(10),(11),(12);
```

需要注意的是，insert 语句的改写，只能将多个 values 后的值拼接成一整条 SQL，insert 语句如果有其他差异将无法被改写。例如：

```
insert into t (a) values (10) on duplicate key update a = 10;
insert into t (a) values (11) on duplicate key update a = 11;
insert into t (a) values (12) on duplicate key update a = 12;
```

上述 insert 语句将无法被改写成一条语句。该例子中，如果将 SQL 改写成如下形式：

```
insert into t (a) values (10) on duplicate key update a = values(a);
insert into t (a) values (11) on duplicate key update a = values(a);
insert into t (a) values (12) on duplicate key update a = values(a);
```

即可满足改写条件，最终被改写成：

```
insert into t (a) values (10), (11), (12) on duplicate key update a = values(a);
```

批量更新时如果有 3 处或 3 处以上更新，则 SQL 语句会改写为 multiple-queries 的形式并发送，这样可以有效减少客户端到服务器的请求开销，但副作用是会产生较大的 SQL 语句，例如这样：

```
update t set a = 10 where id = 1; update t set a = 11 where id = 2; update t set a = 12 where id
    ↪ = 3;
```

另外，因为一个客户端 bug，批量更新时如果要配置 `rewriteBatchedStatements = true` 和 `useServerPrepStmts = true`，推荐同时配置 `allowMultiQueries = true` 参数来避免这个 bug。

执行前检查参数

通过监控可能会发现，虽然业务只向集群进行 insert 操作，却看到有很多多余的 select 语句。通常这是因为 JDBC 发送了一些查询设置类的 SQL 语句（例如 `select @@session.transaction_read_only`）。这些 SQL 对 TiDB 无用，推荐配置 `useConfigs = maxPerformance` 来避免额外开销。

`useConfigs = maxPerformance` 会包含一组配置：

```
cacheServerConfiguration = true
useLocalSessionState = true
elideSetAutoCommits = true
alwaysSendSetIsolation = false
enableQueryTimeouts = false
```

配置后查看监控，可以看到多余语句减少。

10.3.2.3 连接池

TiDB (MySQL) 连接建立是比较昂贵的操作（至少对于 OLTP），除了建立 TCP 连接外还需要进行连接鉴权操作，所以客户端通常会把 TiDB (MySQL) 连接保存到连接池中进行复用。

Java 的连接池实现很多 (HikariCP, tomcat-jdbc, durid, c3p0, dbcp)，TiDB 不会限定使用的连接池，应用可以根据业务特点自行选择连接池实现。

10.3.2.3.1 连接数配置

比较常见的是应用需要根据自身情况配置合适的连接池大小，以 HikariCP 为例：

- `maximumPoolSize`：连接池最大连接数，配置过大会导致 TiDB 消耗资源维护无用连接，配置过小则会导致应用获取连接变慢，所以需根据应用自身特点配置合适的值，可参考这篇文章。
- `minimumIdle`：连接池最小空闲连接数，主要用于在应用空闲时存留一些连接以应对突发请求，同样是需要根据业务情况进行配置。

应用在使用连接池时，需要注意连接使用完成后归还连接，推荐应用使用对应的连接池相关监控（如 `metricRegistry`），通过监控能及时定位连接池问题。

10.3.2.3.2 探活配置

连接池维护到 TiDB 的长连接，TiDB 默认不会主动关闭客户端连接（除非报错），但一般客户端到 TiDB 之间还会有 LVS 或 HAProxy 之类的网络代理，它们通常会在连接空闲一定时间后主动清理连接。除了注意代理的 idle 配置外，连接池还需要进行保活或探测连接。

如果常在 Java 应用中看到以下错误：

```
The last packet sent successfully to the server was 3600000 milliseconds ago. The driver has not
    ↪ received any packets from the server. com.mysql.jdbc.exceptions.jdbc4.
    ↪ CommunicationsException: Communications link failure
```

如果 `n milliseconds ago` 中的 n 如果是 0 或很小的值，则通常是执行的 SQL 导致 TiDB 异常退出引起的报错，推荐查看 TiDB stderr 日志；如果 n 是一个非常大的值（比如这里的 3600000），很可能是因为这个连接空闲太久然后被中间 proxy 关闭了，通常解决方式除了调大 proxy 的 idle 配置，还可以让连接池执行以下操作：

- 每次使用连接前检查连接是否可用。
- 使用单独线程定期检查连接是否可用。
- 定期发送 test query 保活连接。

不同的连接池实现可能会支持其中一种或多种方式，可以查看所使用的连接池文档来寻找对应配置。

10.3.2.4 数据访问框架

业务应用通常会使用某种数据访问框架来简化数据库的访问。

10.3.2.4.1 MyBatis

MyBatis 是目前比较流行的 Java 数据访问框架，主要用于管理 SQL 并完成结果集和 Java 对象的来回映射工作。MyBatis 和 TiDB 兼容性很好，从历史 issue 可以看出 MyBatis 很少出现问题。这里主要关注如下几个配置。

Mapper 参数

MyBatis 的 Mapper 中支持两种参数：

- `select 1 from t where id = #{param1}` 会作为预处理语句，被转换为 `select 1 from t where id = ?` 进行预处理，并使用实际参数来复用执行，通过配合前面的 Prepare 连接参数能获得最佳性能。
- `select 1 from t where id = ${param2}` 会做文本替换为 `select 1 from t where id = 1` 执行，如果这条语句被预处理为不同参数，可能会导致 TiDB 缓存大量的预处理语句，并且以这种方式执行 SQL 有注入安全风险。

动态 SQL Batch

动态 SQL - foreach

要支持将多条 insert 语句自动重写为 `insert ... values(...), (...), ...` 的形式，除了前面所说的在 JDBC 配置 `rewriteBatchedStatements = true` 外，MyBatis 还可以使用动态 SQL 来半自动生成 batch insert。比如下面的 mapper:

```
<insert id="insertTestBatch" parameterType="java.util.List" fetchSize="1">
  insert into test
   (id, v1, v2)
  values
  <foreach item="item" index="index" collection="list" separator=",">
  (
   #{item.id}, #{item.v1}, #{item.v2}
  )
  </foreach>
  on duplicate key update v2 = v1 + values(v1)
</insert>
```

会生成一个 `insert on duplicate key update` 语句，values 后面的 `(?, ?, ?)` 数目是根据传入的 list 个数决定，最终效果和使用 `rewriteBatchStatements = true` 类似，可以有效减少客户端和 TiDB 的网络交互次数，同样需要注意预处理后超过 `prepStmtCacheSqlLimit` 限制导致不缓存预处理语句的问题。

Streaming 结果

前面介绍了在 JDBC 中如何使用流式读取结果，除了 JDBC 相应的配置外，在 MyBatis 中如果希望读取超大结果集合也需要注意：

- 可以通过在 mapper 配置中对单独一条 SQL 设置 `fetchSize`（见上一段代码段），效果等同于调用 JDBC `setFetchSize`
- 可以使用带 `ResultHandler` 的查询接口来避免一次获取整个结果集
- 可以使用 `Cursor` 类来进行流式读取

对于使用 xml 配置映射，可以通过在映射 `<select>` 部分配置 `fetchSize="-2147483648"(Integer.MIN_VALUE)` 来流式读取结果。

```
<select id="getAll" resultMap="postResultMap" fetchSize="-2147483648">
  select * from post;
</select>
```

而使用代码配置映射，则可以使用 `@Options(fetchSize = Integer.MIN_VALUE)` 并返回 `Cursor` 从而让 SQL 结果能被流式读取。

```
@Select("select * from post")
@Options(fetchSize = Integer.MIN_VALUE)
Cursor<Post> queryAllPost();
```

10.3.2.4.2 ExecutorType

在 `openSession` 的时候可以选择 `ExecutorType`，MyBatis 支持三种 executor：

- Simple：每次执行都会向 JDBC 进行预处理语句的调用（如果 JDBC 配置有开启 `cachePrepStmts`，重复的预处理语句会复用）。

- Reuse：在 executor 中缓存预处理语句，这样不用 JDBC 的 cachePrepStmts 也能减少重复预处理语句的调用。
- Batch：每次更新只有在 addBatch 到 query 或 commit 时才会调用 executeBatch 执行，如果 JDBC 层开启了 rewriteBatchStatements，则会尝试改写，没有开启则会一条条发送。

通常默认值是 Simple，需要在调用 openSession 时改变 ExecutorType。如果是 Batch 执行，会遇到事务中前面的 update 或 insert 都非常快，而在读数据或 commit 事务时比较慢的情况，这实际上是正常的，在排查慢 SQL 时需要注意。

10.3.2.5 Spring Transaction

在应用代码中业务可能会通过使用 Spring Transaction 和 AOP 切面的方式来启停事务。

通过在方法定义上添加 @Transactional 注解标记方法，AOP 将会在方法前开启事务，方法返回结果前 commit 事务。如果遇到类似业务，可以通过查找代码 @Transactional 来确定事务的开启和关闭时机。需要特别注意有内嵌的情况，如果发生内嵌，Spring 会根据 Propagation 配置使用不同的行为，因为 TiDB 未支持 savepoint，所以不支持嵌套事务。

10.3.2.6 其他

10.3.2.6.1 排查工具

在 Java 应用发生问题并且不知道业务逻辑情况下，使用 JVM 强大的排查工具会比较有用。这里简单介绍几个常用工具：

jstack

jstack 对应于 Go 中的 pprof/goroutine，可以比较方便地排查进程卡死的问题。

通过执行 jstack pid，即可输出目标进程中所有线程的线程 id 和堆栈信息。输出中默认只有 Java 堆栈，如果希望同时输出 JVM 中的 C++ 堆栈，需要加 -m 选项。

通过多次 jstack 可以方便地发现卡死问题（比如：都通过 Mybatis BatchExecutor flush 调用 update）或死锁问题（比如：测试程序都在抢占应用中某把锁导致没发送 SQL）

另外，top -p $PID -H 或者 Java swiss knife 都是常用的查看线程 ID 的方法。通过 printf "%x\n" pid 把线程 ID 转换成 16 进制，然后去 jstack 输出结果中找对应线程的栈信息，可以定位”某个线程占用 CPU 比较高，不知道它在执行什么”的问题。

jmap & mat

和 Go 中的 pprof/heap 不同，jmap 会将整个进程的内存快照 dump 下来（go 是分配器的采样），然后可以通过另一个工具 mat 做分析。

通过 mat 可以看到进程中所有对象的关联信息和属性，还可以观察线程运行的状态。比如：我们可以通过 mat 找到当前应用中有多少 MySQL 连接对象，每个连接对象的地址和状态信息是什么。

需要注意 mat 默认只会处理 reachable objects，如果要排查 young gc 问题可以在 mat 配置中设置查看 unreachable objects。另外对于调查 young gc 问题（或者大量生命周期较短的对象）的内存分配，用 Java Flight Recorder 比较方便。

trace

线上应用通常无法修改代码，又希望在 Java 中做动态插桩来定位问题，推荐使用 btrace 或 arthas trace。它们可以在不重启进程的情况下动态插入 trace 代码。

火焰图

Java 应用中获取火焰图较繁琐，可参阅 Java Flame Graphs Introduction: Fire For Everyone! 来手动获取。

10.3.2.7 总结

本文从常用的和数据库交互的 Java 组件的角度，阐述了开发 Java 应用程序使用 TiDB 的常见问题与解决办法。TiDB 是高度兼容 MySQL 协议的数据库，基于 MySQL 开发的 Java 应用的最佳实践也多适用于 TiDB。

欢迎大家在 ASK TUG 踊跃发言，和我们一起分享讨论 Java 应用使用 TiDB 的实践技巧或遇到的问题。

10.3.3 HAProxy 在 TiDB 中的最佳实践

本文介绍 HAProxy 在 TiDB 中的最佳配置和使用方法。HAProxy 提供 TCP 协议下的负载均衡能力，TiDB 客户端通过连接 HAProxy 提供的浮动 IP 即可对数据进行操作，实现 TiDB Server 层的负载均衡。

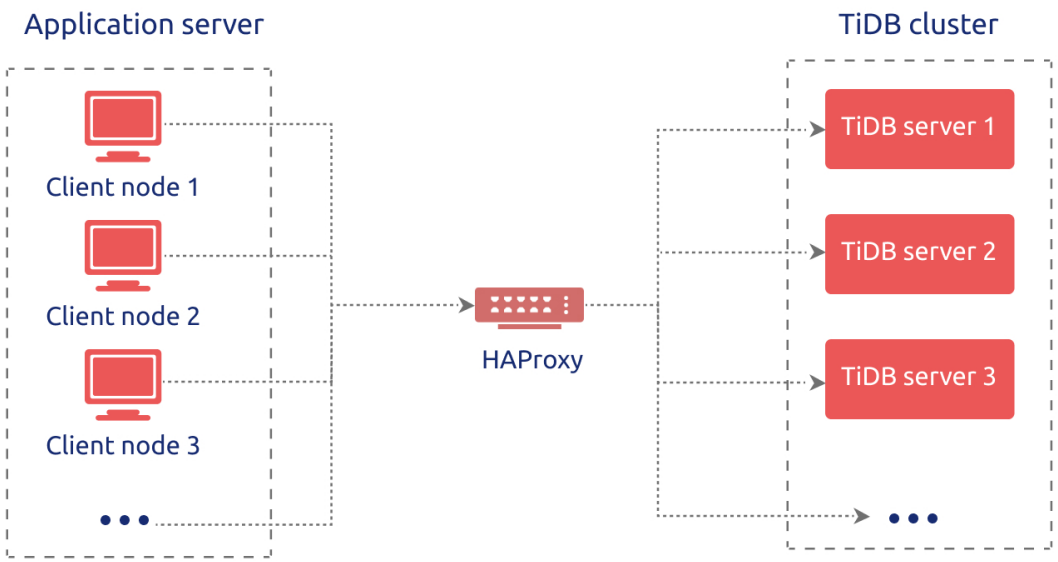


图 97: HAProxy 在 TiDB 中的最佳实践

10.3.3.1 HAProxy 简介

HAProxy 是由 C 语言编写的自由开放源码的软件，为基于 TCP 和 HTTP 协议的应用程序提供高可用性、负载均衡和代理服务。因为 HAProxy 能够快速、高效使用 CPU 和内存，所以目前使用非常广泛，许多知名网站诸如 GitHub、Bitbucket、Stack Overflow、Reddit、Tumblr、Twitter 和 Tuenti 以及亚马逊网络服务系统都在使用 HAProxy。

HAProxy 由 Linux 内核的核心贡献者 Willy Tarreau 于 2000 年编写，他现在仍然负责该项目的维护，并在开源社区免费提供版本迭代。最新的稳定版本 2.0.0 于 2019 年 8 月 16 日发布，带来更多优秀的特性。

10.3.3.2 HAProxy 部分核心功能介绍

- 高可用性：HAProxy 提供优雅关闭服务和无缝切换的高可用功能；
- 负载均衡：L4 (TCP) 和 L7 (HTTP) 两种负载均衡模式，至少 9 类均衡算法，比如 roundrobin，leastconn，random 等；
- 健康检查：对 HAProxy 配置的 HTTP 或者 TCP 模式状态进行检查；
- 会话保持：在应用程序没有提供会话保持功能的情况下，HAProxy 可以提供该项功能；
- SSL：支持 HTTPS 通信和解析；
- 监控与统计：通过 web 页面可以实时监控服务状态以及具体的流量信息。

10.3.3.3 准备环境

在部署 HAProxy 之前，需准备好以下环境。

10.3.3.3.1 硬件要求

根据官方文档，对 HAProxy 的服务器硬件配置有以下建议，也可以根据负载均衡环境进行推算，在此基础上提高服务器配置。

硬件资源	最低配置
CPU	2 核，3.5 GHz
内存	16 GB
存储容量	50 GB（SATA 盘）
网卡	万兆网卡

10.3.3.3.2 依赖软件

根据官方文档，对操作系统和依赖包有以下建议，如果通过 yum 源部署安装 HAProxy 软件，依赖包无需单独安装。

操作系统

操作系统版本	架构
Linux 2.4	x86、x86_64、Alpha、SPARC、MIPS 和 PA-RISC
Linux 2.6 或 3.x	x86、x86_64、ARM、SPARC 和 PPC64
Solaris 8 或 9	UltraSPARC II 和 UltraSPARC III
Solaris 10	Opteron 和 UltraSPARC
FreeBSD 4.10 ~ 10	x86
OpenBSD 3.1 及以上版本	i386、AMD64、macppc、Alpha 和 SPARC64
AIX 5.1 ~ 5.3	Power™

依赖包

- epel-release
- gcc

- systemd-devel

执行如下命令安装依赖包：

```
yum -y install epel-release gcc systemd-devel
```

10.3.3.4 部署 HAProxy

HAProxy 配置 Database 负载均衡场景操作简单，以下部署操作具有普遍性，不具有特殊性，建议根据实际场景，个性化配置相关的配置文件。

10.3.3.4.1 安装 HAProxy

1. 使用 yum 安装 HAProxy：

   ```
   yum -y install haproxy
   ```

2. 验证 HAProxy 安装是否成功：

   ```
   which haproxy
   ```

   ```
   /usr/sbin/haproxy
   ```

HAProxy 命令介绍

执行如下命令查看命令行参数及基本用法：

```
haproxy --help
```

参数	说明
-v	显示简略的版本信息。
-vv	显示详细的版本信息。
-d	开启 debug 模式。
-db	禁用后台模式和多进程模式。
-dM [<byte >]	执行分配内存。
-V	启动过程显示配置和轮询信息。

参数	说明
-D	开启守护进程模式。
-C <dir>	在加载配置文件之前更改目录位置至 <dir>。
-W	主从模式。
-q	静默模式，不输出信息。
-c	只检查配置文件并在尝试绑定之前退出。
-n <limit>	设置每个进程的最大总连接数为 <limit>。
-m <limit>	设置所有进程的最大可用内存为 <limit>（单位：MB）。
-N <limit>	设置单点最大连接数为 <limit>，默认为 2000。
-L <name>	将本地实例对等名称改为 <name>，默认为本地主机名。
-p <file>	将 HAProxy 所有子进程的 PID 信息写入 <file>。
-de	禁止使用 epoll(7)，epoll(7) 仅在 Linux 2.6 和某些定制的 Linux 2.4 系统上可用。
-dp	禁止使用 epoll(2)，可改用 select(2)。

参数	说明
`-dS`	禁止使用 splice(2)，splice(2) 在一些旧版 Linux 内核上不可用。
`-dR`	禁止使用 SO_REUSEPORT。
`-dr`	忽略服务器地址解析失败。
`-dV`	禁止在服务器端使用 SSL。
`-sf <` `↪ pidlist` `↪ >`	启动后，向 pidlist 中的 PID 发送 `finish` 信号，收到此信号的进程在退出之前将等待所有会话完成，即优雅停止服务。此选项必须最后指定，后跟任意数量的 PID。从技术上讲，SIGTTOU 和 SIGUSR1 都被发送。

参数	说明
-st < pidlist >	启动后，向 pidlist 中的 PID 发送 terminate 信号，收到此信号的进程将立即终止，关闭所有活动会话。此选项必须最后指定，后跟任意数量的 PID。从技术上讲，SIGTTOU 和 SIGTERM 都被发送。
-x < unix_socket >	连接指定的 socket 并从旧进程中获取所有 listening socket，然后，使用这些 socket 而不是绑定新的。
-S <bind >[,< bind_options >...]	主从模式下，创建绑定到主进程的 socket，此 socket 可访问每个子进程的 socket。

更多有关 HAProxy 命令参数的信息，可参阅 Management Guide of HAProxy 和 General Commands Manual of HAProxy。

10.3.3.4.2 配置 HAProxy

yum 安装过程中会生成配置模版，你也可以根据实际场景自定义配置如下配置项。

```
global                                  # 全局配置。
   log         127.0.0.1 local2         # 定义全局的 syslog 服务器，最多可以定义两个。
   chroot      /var/lib/haproxy         # 更改当前目录并为启动进程设置超级用户权限，
       ↪ 从而提高安全性。
   pidfile     /var/run/haproxy.pid     # 将 HAProxy 进程的 PID 写入 pidfile。
   maxconn     4000                     # 每个 HAProxy 进程所接受的最大并发连接数。
   user        haproxy                  # 同 UID 参数。
```

```
    group       haproxy                     # 同 GID 参数，建议使用专用用户组。
    nbproc      40                          # 在后台运行时创建的进程数。在启动多个进程转发请求时，
        ↪ 确保该值足够大，保证 HAProxy 不会成为瓶颈。
    daemon                                  # 让 HAProxy 以守护进程的方式工作于后台，
        ↪ 等同于命令行参数“-D”的功能。当然，也可以在命令行中用“-db”参数将其禁用。
    stats socket /var/lib/haproxy/stats     # 统计信息保存位置。

defaults                                    # 默认配置。
    log global                              # 日志继承全局配置段的设置。
    retries 2                               # 向上游服务器尝试连接的最大次数，
        ↪ 超过此值便认为后端服务器不可用。
    timeout connect  2s                     # HAProxy 与后端服务器连接超时时间。
        ↪ 如果在同一个局域网内，可设置成较短的时间。
    timeout client 30000s                   # 客户端与 HAProxy 连接后，数据传输完毕，
        ↪ 即非活动连接的超时时间。
    timeout server 30000s                   # 服务器端非活动连接的超时时间。

listen admin_stats                          # frontend 和 backend 的组合体，
    ↪ 此监控组的名称可按需进行自定义。
    bind 0.0.0.0:8080                       # 监听端口。
    mode http                               # 监控运行的模式，此处为 `http` 模式。
    option httplog                          # 开始启用记录 HTTP 请求的日志功能。
    maxconn 10                              # 最大并发连接数。
    stats refresh 30s                       # 每隔 30 秒自动刷新监控页面。
    stats uri /haproxy                      # 监控页面的 URL。
    stats realm HAProxy                     # 监控页面的提示信息。
    stats auth admin:pingcap123             # 监控页面的用户和密码，可设置多个用户名。
    stats hide-version                      # 隐藏监控页面上的 HAProxy 版本信息。
    stats  admin if TRUE                    # 手工启用或禁用后端服务器（HAProxy 1.4.9
        ↪ 及之后版本开始支持）。

listen tidb-cluster                         # 配置 database 负载均衡。
    bind 0.0.0.0:3390                       # 浮动 IP 和 监听端口。
    mode tcp                                # HAProxy 要使用第 4 层的传输层。
    balance leastconn                       # 连接数最少的服务器优先接收连接。`leastconn`
        ↪ 建议用于长会话服务，例如 LDAP、SQL、TSE 等，而不是短会话协议，如 HTTP。该算法是动态的
        ↪ ，对于启动慢的服务器，服务器权重会在运行中作调整。
    server tidb-1 10.9.18.229:4000 check inter 2000 rise 2 fall 3       # 检测 4000 端口，
        ↪ 检测频率为每 2000 毫秒一次。如果 2 次检测为成功，则认为服务器可用；如果 3 次检测为失败
        ↪ ，则认为服务器不可用。
    server tidb-2 10.9.39.208:4000 check inter 2000 rise 2 fall 3
    server tidb-3 10.9.64.166:4000 check inter 2000 rise 2 fall 3
```

10.3.3.4.3 启动 HAProxy

- 方法一：执行 haproxy，默认读取 /etc/haproxy/haproxy.cfg（推荐）。

```
haproxy -f /etc/haproxy/haproxy.cfg
```

- 方法二：使用 systemd 启动 HAProxy。

```
systemctl start haproxy.service
```

10.3.3.4.4 停止 HAProxy

- 方法一：使用 kill -9。

1. 执行如下命令：

```
ps -ef | grep haproxy
```

2. 终止 HAProxy 相关的 PID 进程：

```
kill -9 ${haproxy.pid}
```

- 方法二：使用 systemd。

```
systemctl stop haproxy.service
```

10.3.4 TiDB 高并发写入场景最佳实践

在 TiDB 的使用过程中，一个典型场景是高并发批量写入数据到 TiDB。本文阐述了该场景中的常见问题，旨在给出一个业务的最佳实践，帮助读者避免因使用 TiDB 不当而影响业务开发。

10.3.4.1 目标读者

本文假设你已对 TiDB 有一定的了解，推荐先阅读 TiDB 原理相关的三篇文章（讲存储，说计算，谈调度），以及 TiDB Best Practice。

10.3.4.2 高并发批量插入场景

高并发批量插入的场景通常出现在业务系统的批量任务中，例如清算以及结算等业务。此类场景存在以下特点：

- 数据量大
- 需要短时间内将历史数据入库
- 需要短时间内读取大量数据

这就对 TiDB 提出了以下挑战：

- 写入/读取能力是否可以线性水平扩展
- 随着数据持续大并发写入，数据库性能是否稳定不衰减

对于分布式数据库来说，除了本身的基础性能外，最重要的就是充分利用所有节点能力，避免让单个节点成为瓶颈。

10.3.4.3 TiDB 数据分布原理

如果要解决以上挑战，需要从 TiDB 数据切分以及调度的原理开始讲起。这里只作简单说明，详情可参阅谈调度。

TiDB 以 Region 为单位对数据进行切分，每个 Region 有大小限制（默认 96M）。Region 的切分方式是范围切分。每个 Region 会有多副本，每一组副本，称为一个 Raft Group。每个 Raft Group 中由 Leader 负责执行这块数据的读 & 写（TiDB 即将支持 Follower-Read）。Leader 会自动地被 PD 组件均匀调度在不同的物理节点上，用以均分读写压力。

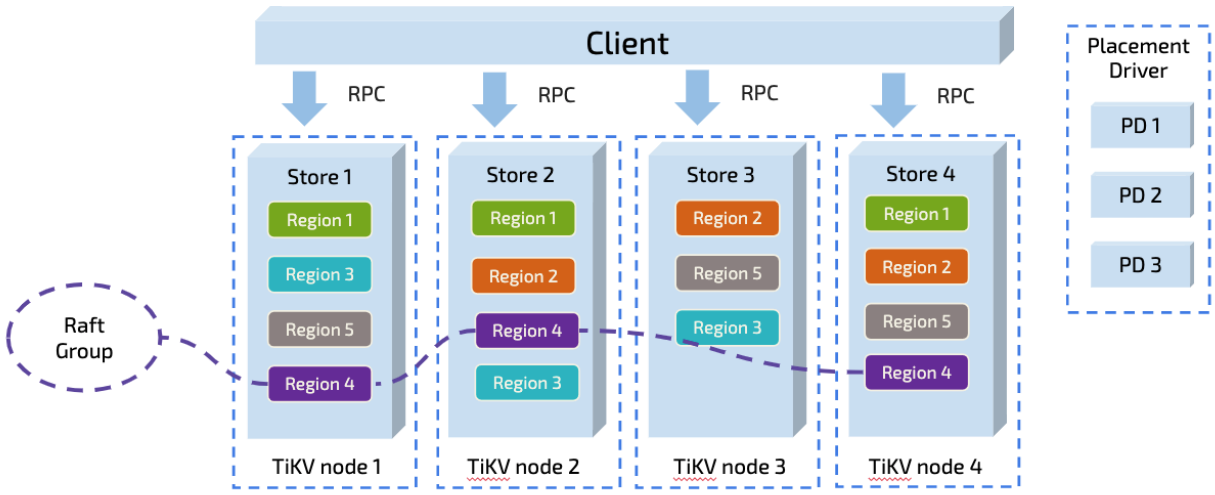


图 98: TiDB 数据概览

只要业务的写入没有 AUTO_INCREMENT 的主键，或没有单调递增的索引（即没有业务上的写入热点，更多细节可参阅 TiDB 正确使用方式），从原理上来说，TiDB 依靠这个架构可具备线性扩展的读写能力，并且可以充分利用分布式资源。从这一点看，TiDB 尤其适合高并发批量写入场景的业务。

但理论场景和实际情况往往存在不同。以下实例说明了热点是如何产生的。

10.3.4.4 热点产生的实例

以下为一张示例表：

```
CREATE TABLE IF NOT EXISTS TEST_HOTSPOT(
      id         BIGINT PRIMARY KEY,
      age        INT,
      user_name  VARCHAR(32),
      email      VARCHAR(128)
)
```

这个表的结构非常简单，除了 id 为主键以外，没有额外的二级索引。将数据写入该表的语句如下，id 通过随机数离散生成：

```
INSERT INTO TEST_HOTSPOT(id, age, user_name, email) values(%v, %v, '%v', '%v');
```

负载是短时间内密集地执行以上写入语句。

以上操作看似符合理论场景中的 TiDB 最佳实践，业务上没有热点产生。只要有足够的机器，就可以充分利用 TiDB 的分布式能力。要验证是否真的符合最佳实践，可以在实验环境中进行测试。

部署拓扑 2 个 TiDB 节点，3 个 PD 节点，6 个 TiKV 节点。请忽略 QPS，因为测试只是为了阐述原理，并非 benchmark。

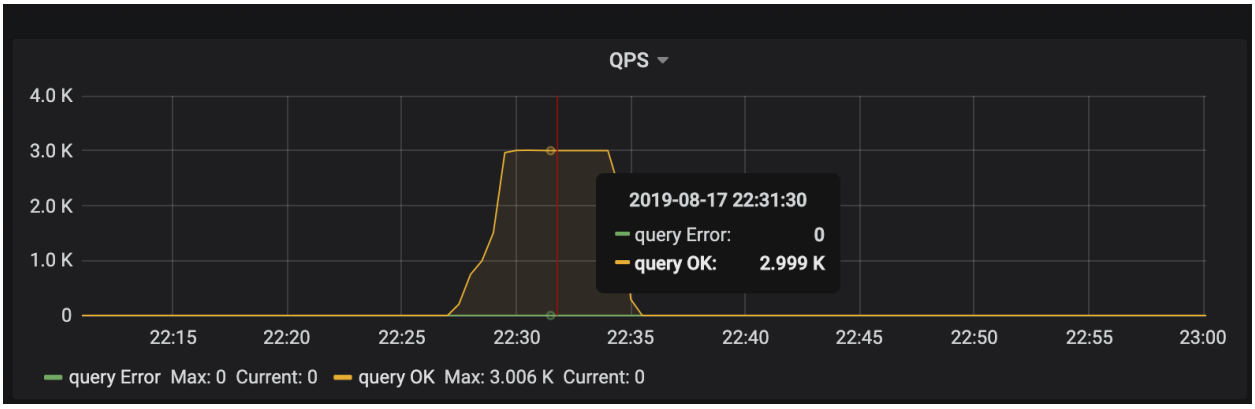


图 99: QPS1

客户端在短时间内发起了“密集”的写入，TiDB 收到的请求是 3K QPS。理论上，压力应该均摊给 6 个 TiKV 节点。但是从 TiKV 节点的 CPU 使用情况上看，存在明显的写入倾斜（tikv - 3 节点是写入热点）：

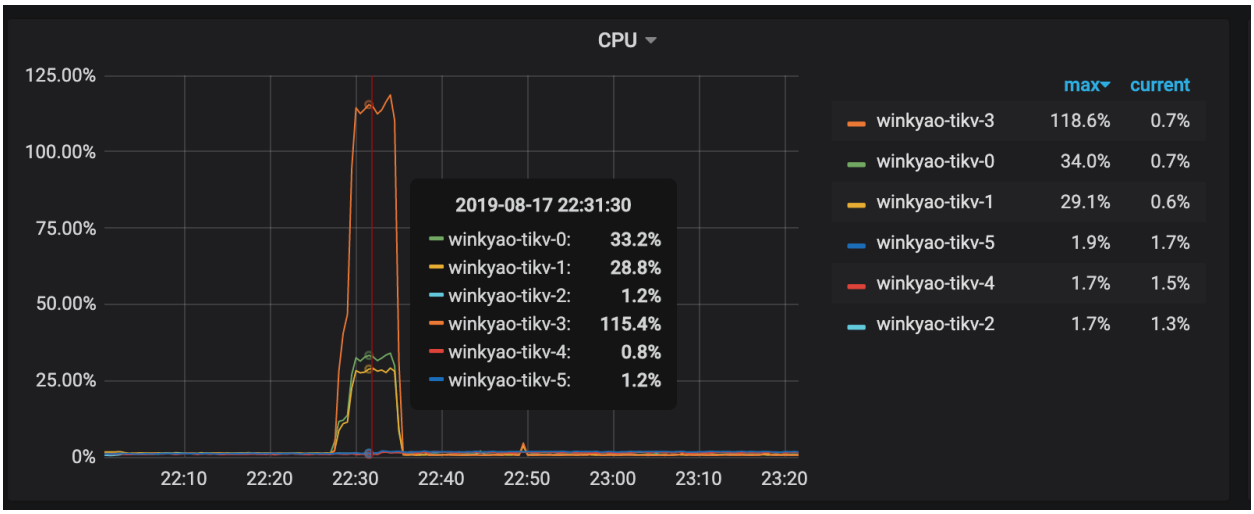


图 100: QPS2

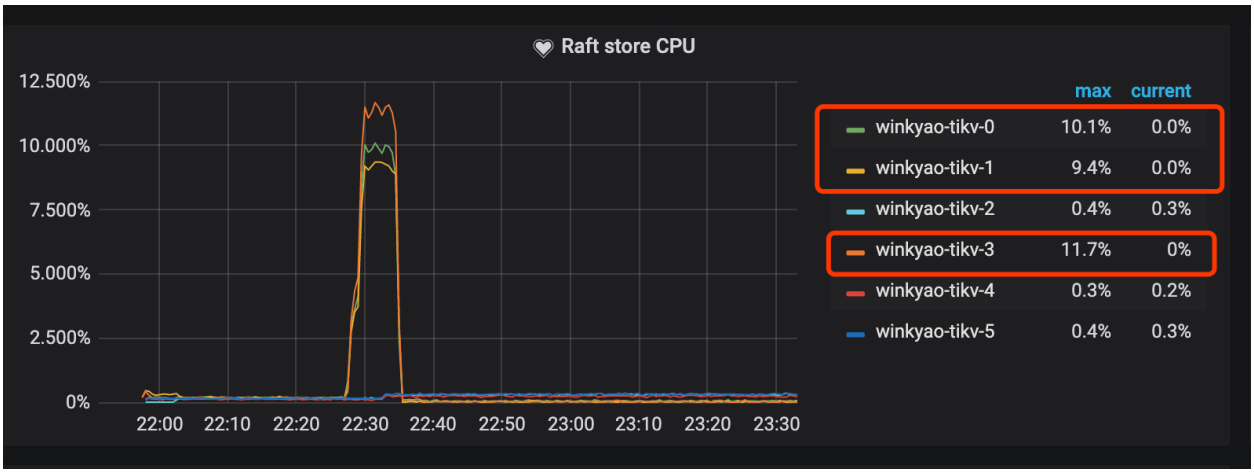


图 101: QPS3

Raft store CPU 为 `raftstore` 线程的 CPU 使用率，通常代表写入的负载。在这个场景下 tikv-3 为 Raft Leader，tikv-0 和 tikv-1 是 Raft 的 Follower，其他的 TiKV 节点的负载几乎为空。

从 PD 的监控中也可以证明热点的产生：

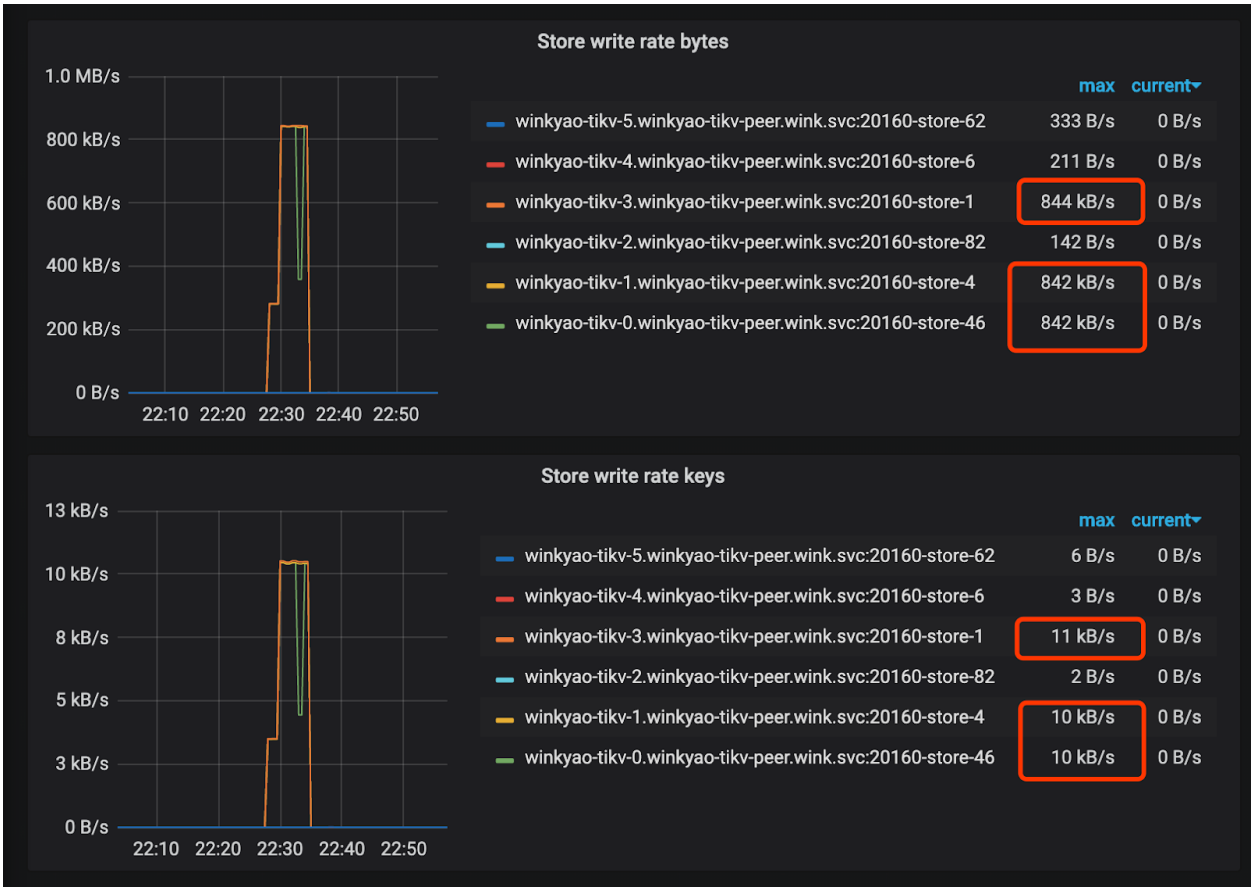


图 102: QPS4

10.3.4.5 热点问题产生的原因

以上测试并未达到理论场景中最佳实践，因为刚创建表的时候，这个表在 TiKV 中只会对应为一个 Region，范围是：

```
[CommonPrefix + TableID, CommonPrefix + TableID + 1)
```

短时间内大量数据会持续写入到同一个 Region 上。

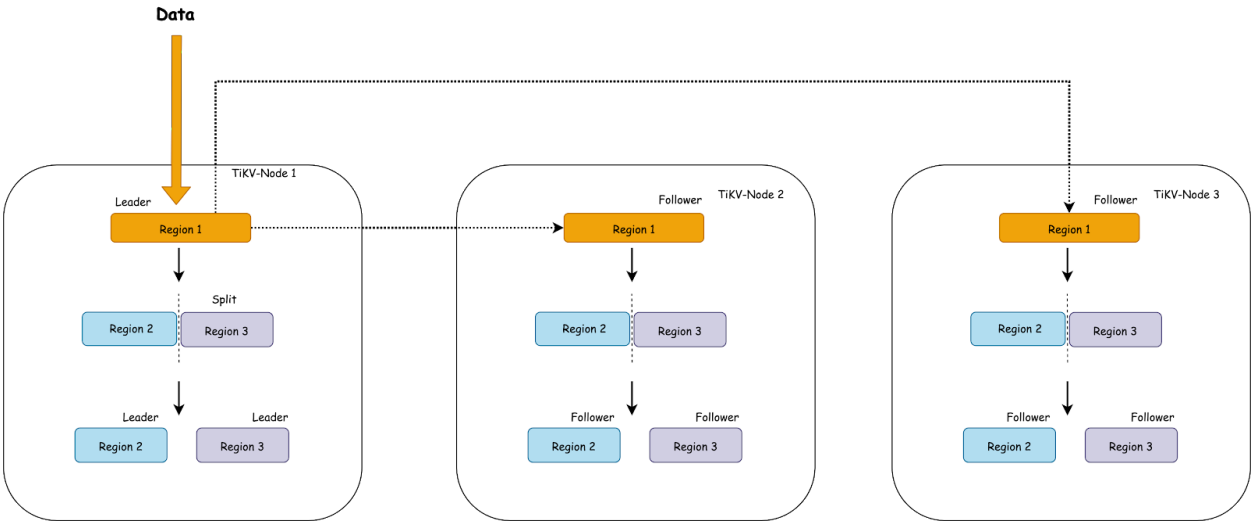


图 103: TiKV Region 分裂流程

上图简单描述了这个过程，随着数据持续写入，TiKV 会将一个 Region 切分为多个。但因为首先发起选举的是原 Leader 所在的 Store，所以新切分好的两个 Region 的 Leader 很可能还会在原 Store 上。新切分好的 Region 2，3 上，也会重复之前发生在 Region 1 上的过程。也就是压力会密集地集中在 TiKV-Node 1 上。

在持续写入的过程中，PD 发现 Node 1 中产生了热点，会将 Leader 均分到其他的 Node 上。如果 TiKV 的节点数多于副本数的话，TiKV 会尽可能将 Region 迁移到空闲的节点上。这两个操作在数据插入的过程中，也能在 PD 监控中得到印证：

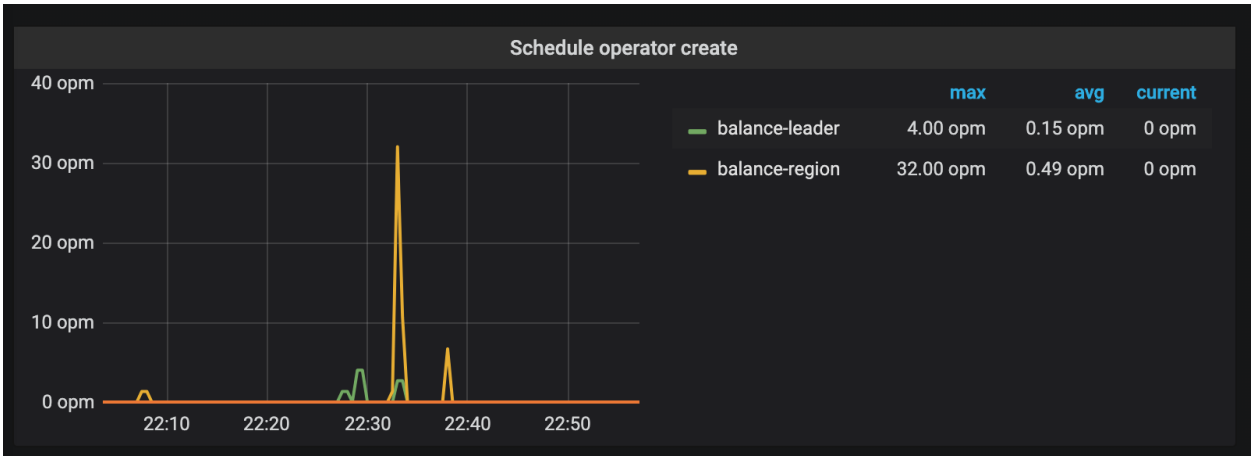


图 104: QPS5

在持续写入一段时间后，整个集群会被 PD 自动地调度成一个压力均匀的状态，到那个时候整个集群的能力才会真正被利用起来。在大多数情况下，以上热点产生的过程是没有问题的，这个阶段属于表 Region 的预热阶段。

但是对于高并发批量密集写入场景来说，应该避免这个阶段。

10.3.4.6 热点问题的规避方法

为了达到场景理论中的最佳性能，可跳过这个预热阶段，直接将 Region 切分为预期的数量，提前调度到集群的各个节点中。

TiDB 在 v3.0.x 以及 v2.1.13 后支持一个叫Split Region 的新特性。这个特性提供了新的语法：

```
SPLIT TABLE table_name [INDEX index_name] BETWEEN (lower_value) AND (upper_value) REGIONS
    ↪ region_num
```

```
SPLIT TABLE table_name [INDEX index_name] BY (value_list) [, (value_list)]
```

但是 TiDB 并不会自动提前完成这个切分操作。原因如下：

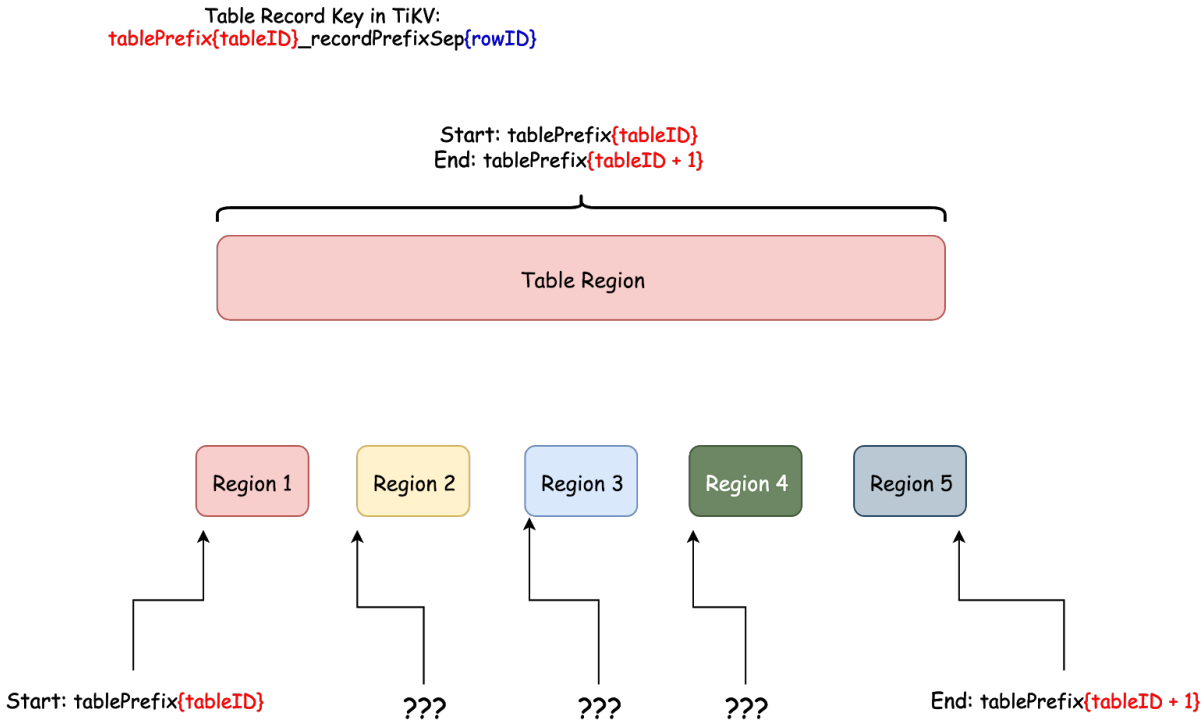


图 105: Table Region Range

从上图可知，根据行数据 key 的编码规则，行 ID (rowID) 是行数据中唯一可变的部分。在 TiDB 中，rowID 是一个 Int64 整型。但是用户不一定能将 Int64 整型范围均匀切分成需要的份数，然后均匀分布在不同的节点上，还需要结合实际情况。

如果行 ID 的写入是完全离散的，那么上述方式是可行的。如果行 ID 或者索引有固定的范围或者前缀（例如，只在 [2000w，5000w) 的范围内离散插入数据），这种写入依然在业务上不产生热点，但是如果按上面的方式进行切分，那么有可能一开始数据仍只写入到某个 Region 上。

作为一款通用数据库，TiDB 并不对数据的分布作假设，所以开始只用一个 Region 来对应一个表。等到真实数据插入进来以后，TiDB 自动根据数据的分布来作切分。这种方式是较通用的。

所以 TiDB 提供了 Split Region 语法，专门针对短时批量写入场景作优化。基于以上案例，下面尝试用 Split Region 语法提前切散 Region，再观察负载情况。

由于测试的写入数据在正数范围内完全离散，所以用以下语句，在 Int64 空间内提前将表切分为 128 个 Region：

```
SPLIT TABLE TEST_HOTSPOT BETWEEN (0) AND (9223372036854775807) REGIONS 128;
```

切分完成以后，可以通过 SHOW TABLE test_hotspot REGIONS; 语句查看打散的情况。如果 SCATTERING 列值全部为 0，代表调度成功。

也可以通过 table-regions.py 脚本，查看 Region 的分布。目前分布已经比较均匀了：

```
[root@172.16.4.4 scripts]# python table-regions.py --host 172.16.4.3 --port 31453 test
    ↪ test_hotspot
[RECORD - test.test_hotspot] - Leaders Distribution:
  total leader count: 127
  store: 1, num_leaders: 21, percentage: 16.54%
  store: 4, num_leaders: 20, percentage: 15.75%
  store: 6, num_leaders: 21, percentage: 16.54%
  store: 46, num_leaders: 21, percentage: 16.54%
  store: 82, num_leaders: 23, percentage: 18.11%
  store: 62, num_leaders: 21, percentage: 16.54%
```

再重新运行写入负载：

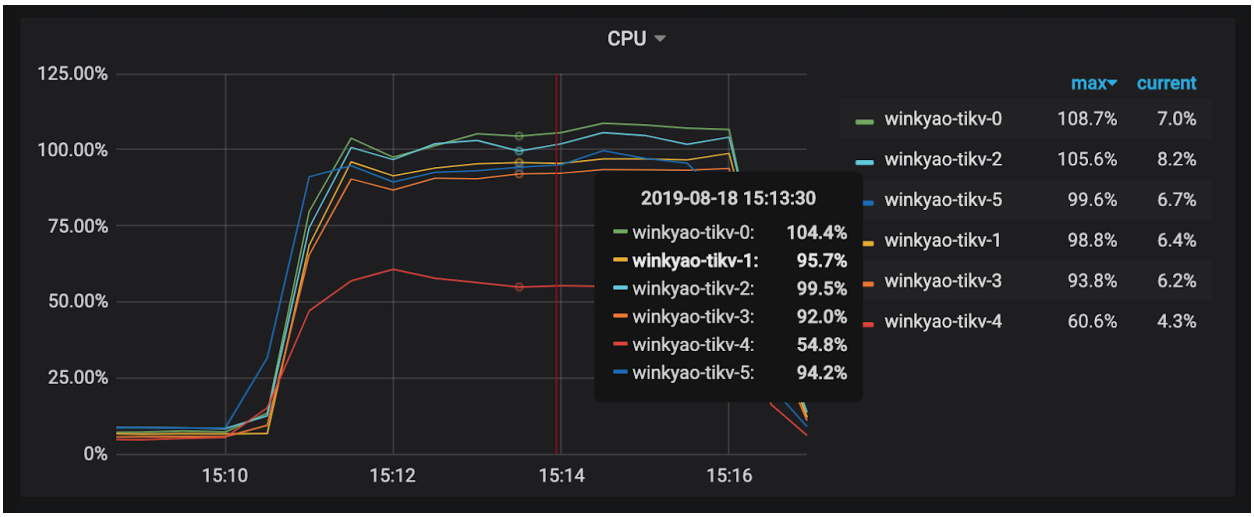


图 106: QPS6

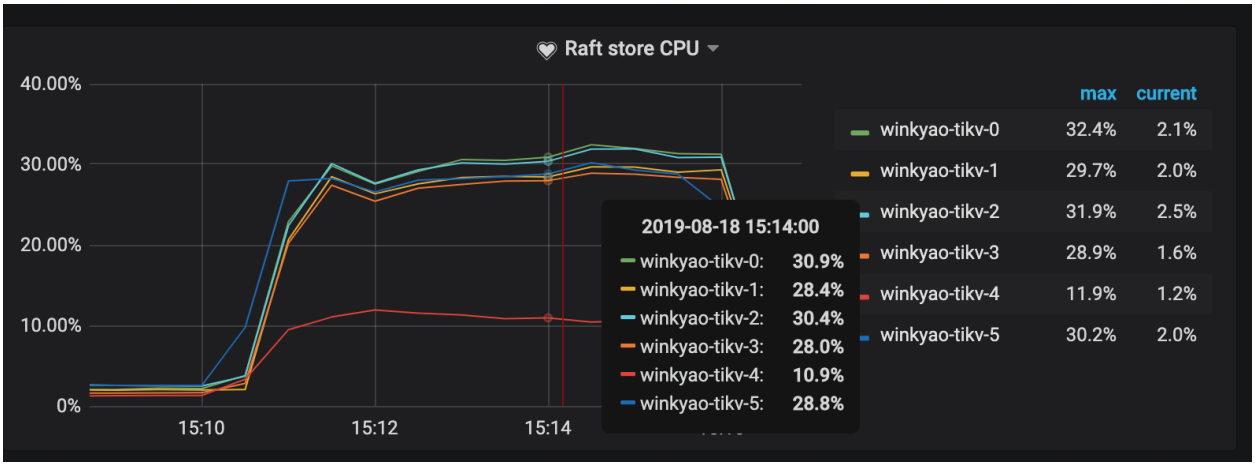


图 107: QPS7

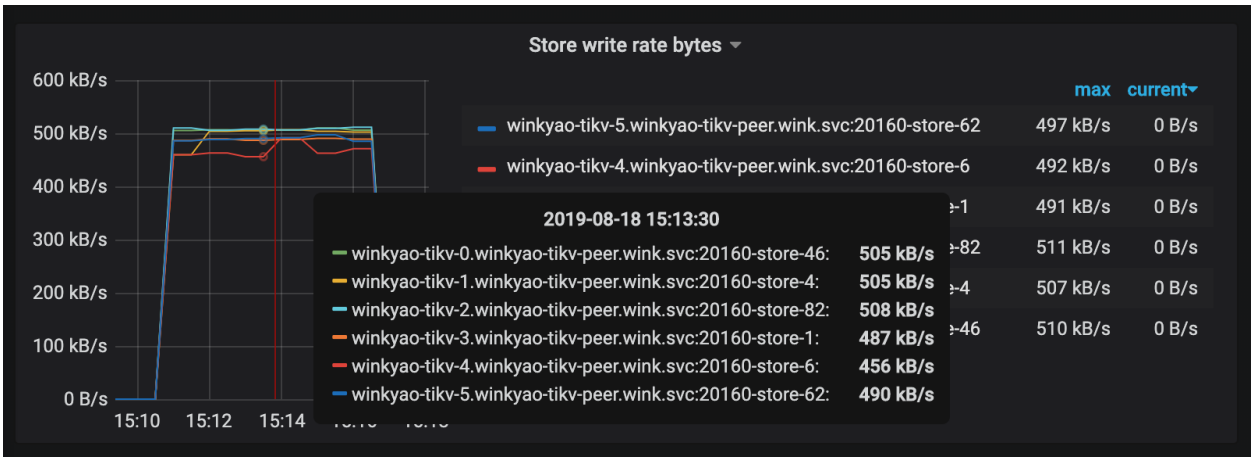


图 108: QPS8

可以看到已经消除了明显的热点问题了。

本示例仅为一个简单的表，还有索引热点的问题需要考虑。读者可参阅Split Region 文档来了解如何预先切散索引相关的 Region。

10.3.4.6.1 更复杂的热点问题

问题一：

如果表没有主键或者主键不是整数类型，而且用户也不想自己生成一个随机分布的主键 ID 的话，TiDB 内部有一个隐式的 `_tidb_rowid` 列作为行 ID。在不使用 `SHARD_ROW_ID_BITS` 的情况下，`_tidb_rowid` 列的值基本也为单调递增，此时也会有写热点存在（参阅`SHARD_ROW_ID_BITS` 的详细说明）。

要避免由 `_tidb_rowid` 带来的写入热点问题，可以在建表时，使用 `SHARD_ROW_ID_BITS` 和 `PRE_SPLIT_REGIONS` 这两个建表选项（参阅`PRE_SPLIT_REGIONS` 的详细说明）。

SHARD_ROW_ID_BITS 用于将 _tidb_rowid 列生成的行 ID 随机打散。PRE_SPLIT_REGIONS 用于在建完表后预先进行 Split region。

> **注意：**
>
> PRE_SPLIT_REGIONS 的值必须小于或等于 SHARD_ROW_ID_BITS。

以下全局变量会影响 PRE_SPLIT_REGIONS 的行为，需要特别注意：

- tidb_scatter_region：该变量用于控制建表完成后是否等待预切分和打散 Region 完成后再返回结果。如果建表后有大批量写入，需要设置该变量值为 1，表示等待所有 Region 都切分和打散完成后再返回结果给客户端。否则未打散完成就进行写入会对写入性能影响有较大的影响。

示例：

```
create table t (a int, b int) SHARD_ROW_ID_BITS = 4 PRE_SPLIT_REGIONS=3;
```

- SHARD_ROW_ID_BITS = 4 表示 tidb_rowid 的值会随机分布成 16 （16=2^4）个范围区间。
- PRE_SPLIT_REGIONS=3 表示建完表后提前切分出 8 (2^3) 个 Region。

开始写数据进表 t 后，数据会被写入提前切分好的 8 个 Region 中，这样也避免了刚开始建表完后因为只有一个 Region 而存在的写热点问题。

问题二：

如果表的主键为整数类型，并且该表使用了 AUTO_INCREMENT 来保证主键唯一性（不需要连续或递增）的表而言，由于 TiDB 直接使用主键行值作为 _tidb_rowid，此时无法使用 SHARD_ROW_ID_BITS 来打散热点。

要解决上述热点问题，可以利用 AUTO_RANDOM 列属性（参阅AUTO_RANDOM 的详细说明），将 AUTO_INCREMENT 改为 AUTO_RANDOM，插入数据时让 TiDB 自动为整型主键列分配一个值，消除行 ID 的连续性，从而达到打散热点的目的。

10.3.4.7 参数配置

TiDB 2.1 版本中在 SQL 层引入了 latch 机制，用于在写入冲突比较频繁的场景中提前发现事务冲突，减少 TiDB 和 TiKV 事务提交时写写冲突导致的重试。通常，跑批场景使用的是存量数据，所以并不存在事务的写入冲突。可以把 TiDB 的 latch 功能关闭，以减少为细小对象分配内存：

```
[txn-local-latches]
enabled = false
```

10.3.5 使用 Grafana 监控 TiDB 的最佳实践

使用 TiDB Ansible 部署 TiDB 集群时，会同时部署一套Grafana + Prometheus 的监控平台，用于收集和展示 TiDB 集群各个组件和机器的 metric 信息。本文主要介绍使用 TiDB 监控的最佳实践，旨在帮助 TiDB 用户高效利用丰富的 metric 信息来分析 TiDB 的集群状态或进行故障诊断。

10.3.5.1 监控架构

Prometheus 是一个拥有多维度数据模型和灵活查询语句的时序数据库。Grafana 是一个开源的 metric 分析及可视化系统。

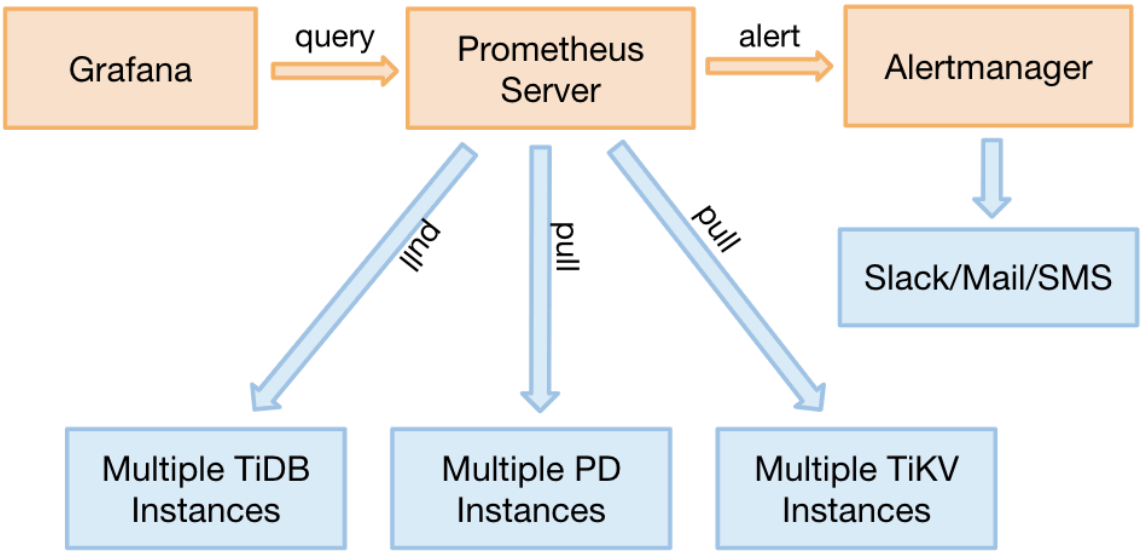


图 109: TiDB 监控整体架构

从 TiDB 2.1.3 版本开始，监控可以支持 pull，这是一个非常好的调整，它有以下几个优点：

- 如果 Prometheus 需要迁移，无需重启整个 TiDB 集群。调整前，因为组件要调整 push 的目标地址，迁移 Prometheus 需要重启整个集群。
- 支持部署 2 套独立的 Grafana + Prometheus 的监控平台（非 HA），防止监控的单点。方法是使用 TiDB Ansible 用不同的 IP 各执行一次部署命令。
- 去掉了 Pushgateway 这个单点组件。

10.3.5.2 监控数据的来源与展示

TiDB 的 3 个核心组件（TiDB server、TiKV server 和 PD server）可以通过 HTTP 接口来获取 metric 数据。这些 metric 均是从程序代码中上传的，默认端口如下：

组件	端口
TiDB server	10080
TiKV server	20181
PD server	2379

下面以 TiDB server 为例，展示如何通过 HTTP 接口查看一个语句的 QPS 数据：

```
curl http://__tidb_ip__:10080/metrics |grep tidb_executor_statement_total
```

```
### 可以看到实时 QPS 数据，并根据不同 type 对 SQL 语句进行了区分，value 是 counter 类型的累计值（
    ↪ 科学计数法）。
tidb_executor_statement_total{type="Delete"} 520197
tidb_executor_statement_total{type="Explain"} 1
tidb_executor_statement_total{type="Insert"} 7.20799402e+08
tidb_executor_statement_total{type="Select"} 2.64983586e+08
tidb_executor_statement_total{type="Set"} 2.399075e+06
tidb_executor_statement_total{type="Show"} 500531
tidb_executor_statement_total{type="Use"} 466016
```

这些数据会存储在 Prometheus 中，然后在 Grafana 上进行展示。在面板上点击鼠标右键会出现 Edit 按钮（或直接按 E 键），如下图所示：

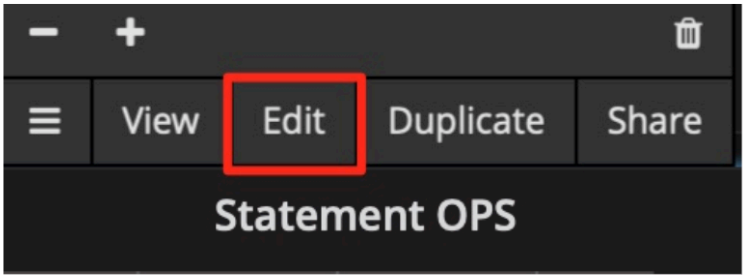


图 110: Metrics 面板的编辑入口

点击 Edit 按钮之后，在 Metrics 面板上可以看到利用该 metric 的 query 表达式。面板上一些细节的含义如下：

- `rate[1m]`：表示 1 分钟的增长速率，只能用于 counter 类型的数据。
- `sum`：表示 value 求和。
- `by type`：表示将求和后的数据按 metric 原始值中的 type 进行分组。
- `Legend format`：表示指标名称的格式。
- `Resolution`：默认打点步长是 15s，Resolution 表示是否将多个样本数据合并成一个点。

Metrics 面板中的表达式如下：

图 111: Metric 面板中的表达式

Prometheus 支持很多表达式与函数，更多表达式请参考 Prometheus 官网页面。

10.3.5.3 Grafana 使用技巧

本小节介绍高效利用 Grafana 监控分析 TiDB 指标的七个技巧。

10.3.5.3.1 技巧 1：查看所有维度并编辑表达式

在监控数据的来源与展示一节的示例中，数据是按照 type 进行分组的。如果你想知道是否还能按其它维度分组，并快速查看还有哪些维度，可采用以下技巧：在 query 的表达式上只保留指标名称，不做任何计算，Legend format 也留空。这样就能显示出原始的 metric 数据。比如，下图能看到有 3 个维度（instance、job 和 type）：

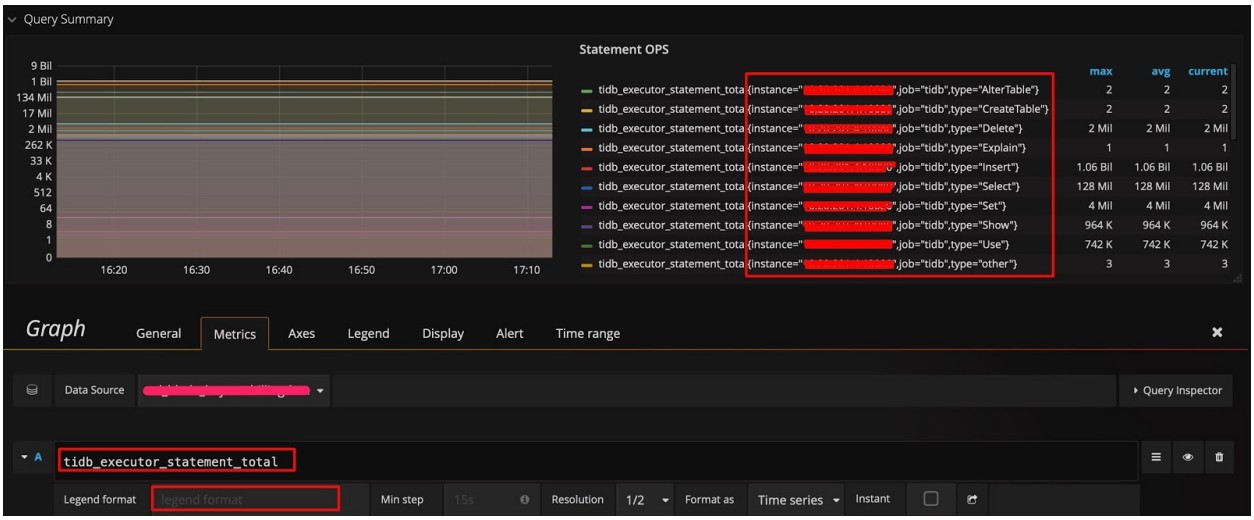


图 112: 编辑表达式并查看所有维度

然后调整表达式，在原有的 type 后面加上 instance 这个维度，在 Legend format 处增加 {{instance}}，就可以看到每个 TiDB server 上执行的不同类型 SQL 语句的 QPS 了。如下图所示：

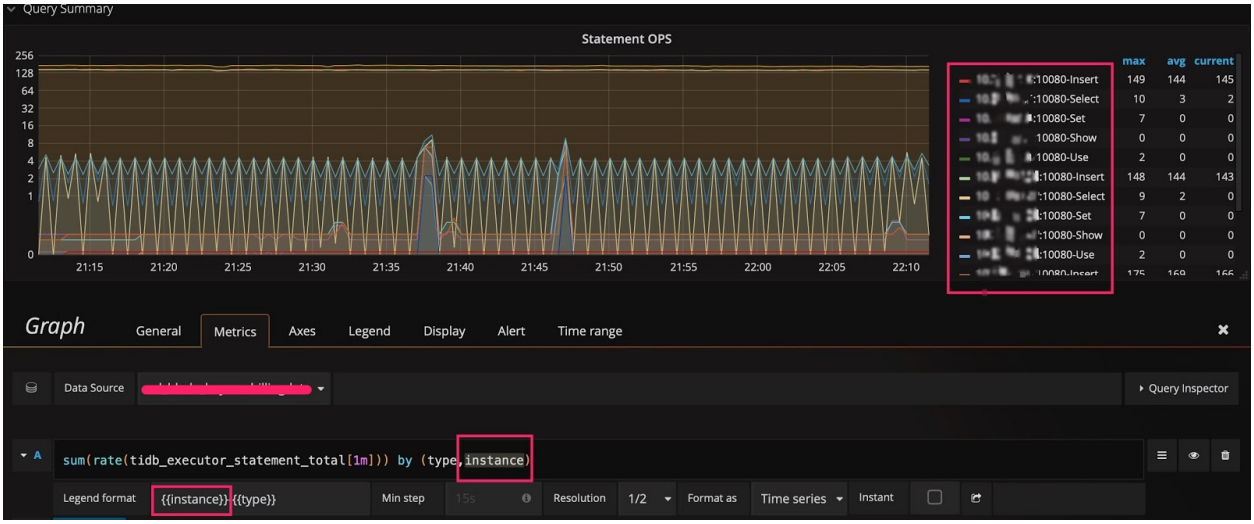


图 113: 给表达式增加一个 instance 维度

10.3.5.3.2 技巧 2：调整 Y 轴标尺的计算方式

以 Query Duration 指标为例，默认的比例尺采用 2 的对数计算，显示上会将差距缩小。为了观察到明显的变化，可以将比例尺改为线性，从下面两张图中可以看到显示上的区别，明显发现那个时刻有个 SQL 语句运行较慢。

当然也不是所有场景都适合用线性，比如观察 1 个月的性能趋势，用线性可能就会有很多噪点，不好观察。

标尺默认的比例尺为 2 的对数：

图 114: 标尺默认的比例尺为 2 的对数

将标尺的比例尺调整为线性：

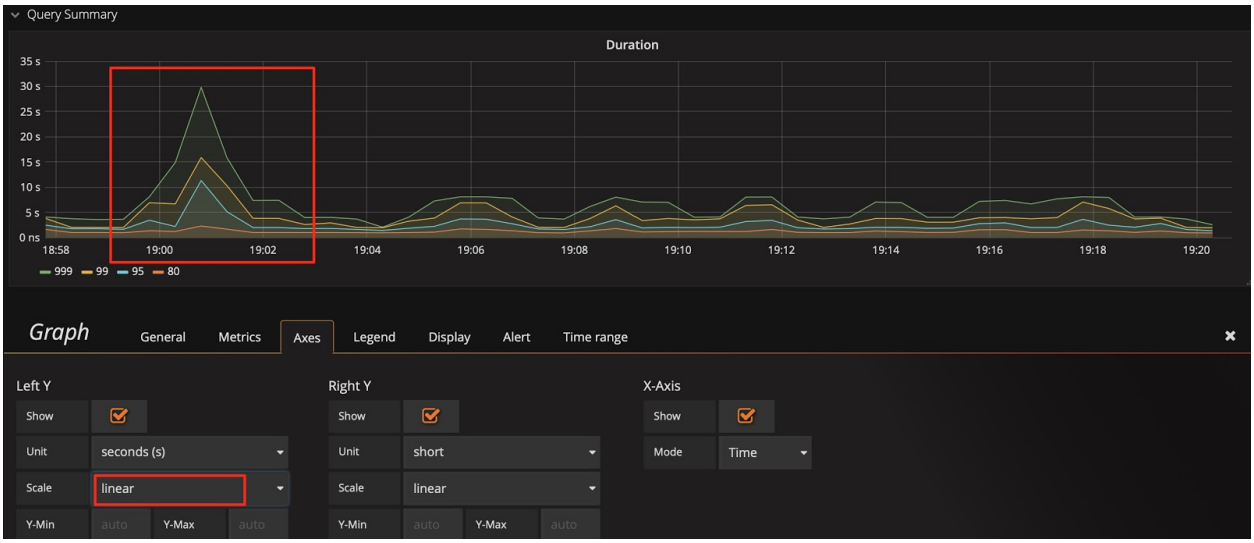


图 115: 调整标尺的比例尺为线性

> **建议：**
>
> 结合技巧 1，会发现这里还有一个 `sql_type` 的维度，可以立刻分析出是 `SELECT` 慢还是 `UPDATE` 慢；并且可以分析出是哪个 instance 上的语句慢。

10.3.5.3.3 技巧 3：调整 Y 轴基线，放大变化

有时已经用了线性比例尺，却还是看不出变化趋势。比如下图中，在扩容后想观察 `Store size` 的实时变化效果，但由于基数较大，观察不到微弱的变化。这时可以将 Y 轴最小值从 0 改为 `auto`，将上部放大。观察下面两张图的区别，可以看出数据已开始迁移了。

基线默认为 0：

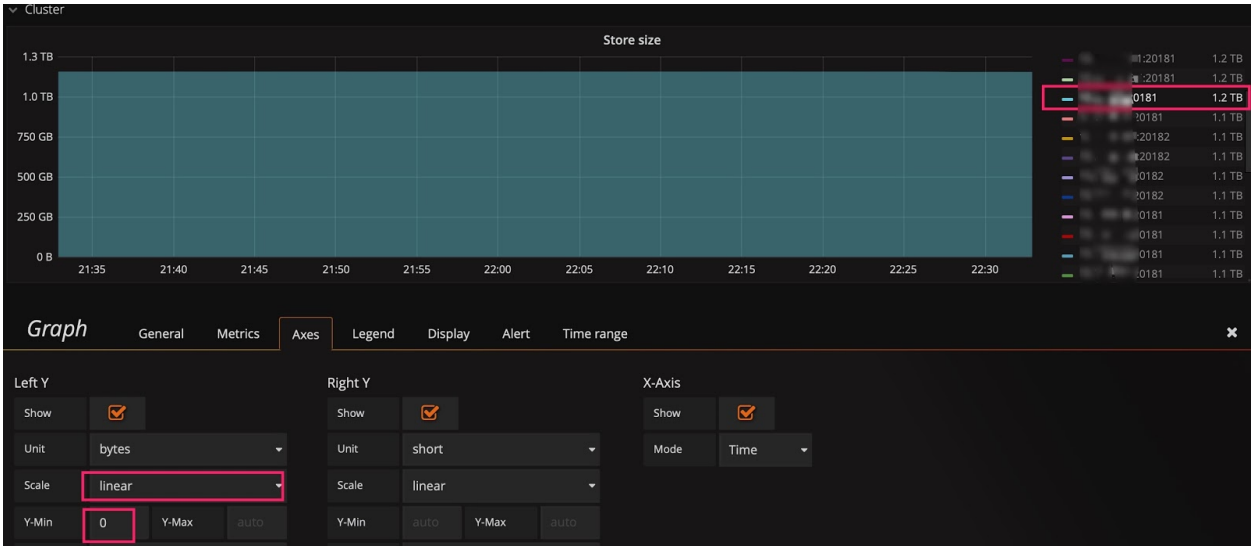


图 116: 基线默认为 0

将基线调整为 auto：

图 117: 调整基线为 auto

10.3.5.3.4 技巧 4：标尺联动

在 Settings 面板中，有一个 Graph Tooltip 设置项，默认使用 Default。

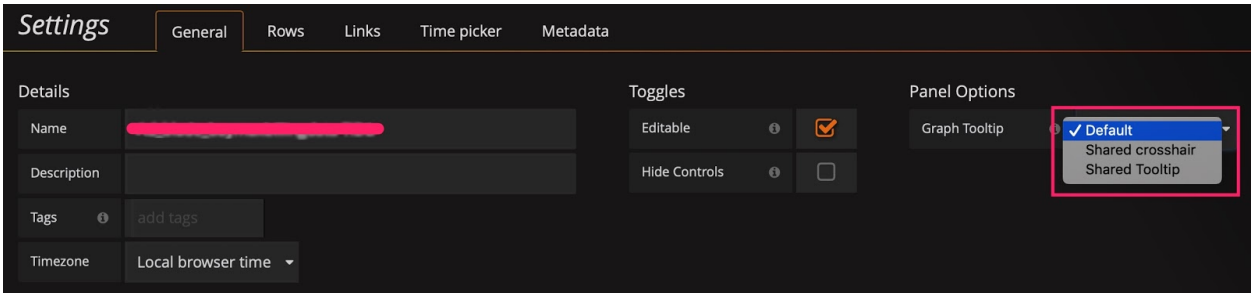


图 118: 图形展示工具

下面将图形展示工具分别调整为 Shared crosshair 和 Shared Tooltip 看看效果。可以看到标尺能联动展示了，方便排查问题时确认 2 个指标的关联性。

将图形展示工具调整为 Shared crosshair：

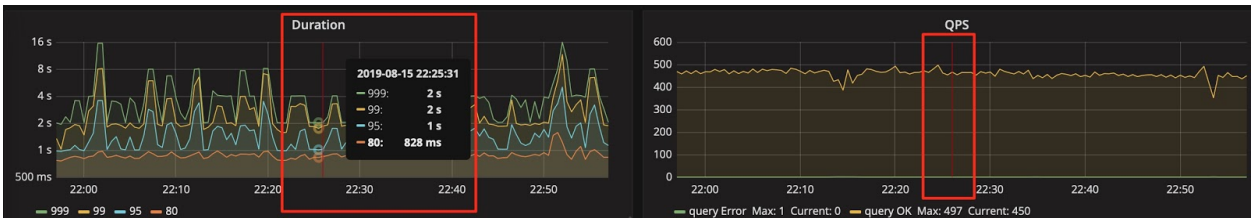


图 119: 调整图形展示工具为 Shared crosshair

将图形展示工具调整为 Shared Tooltip：

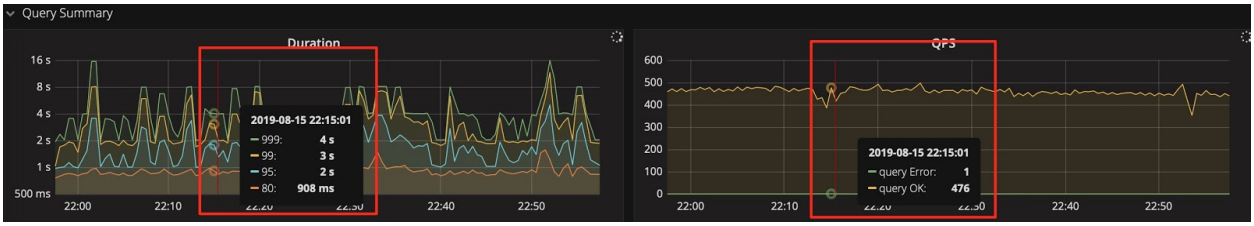


图 120: 调整图形展示工具为 Shared Tooltip

10.3.5.3.5 技巧 5：手动输入 ip：端口号查看历史信息

PD 的 dashboard 只展示当前 leader 的 metric 信息，而有时想看历史上 PD leader 当时的状况，但是 instance 下拉列表中已不存在这个成员了。此时，可以手动输入 ip:2379 来查看当时的数据。

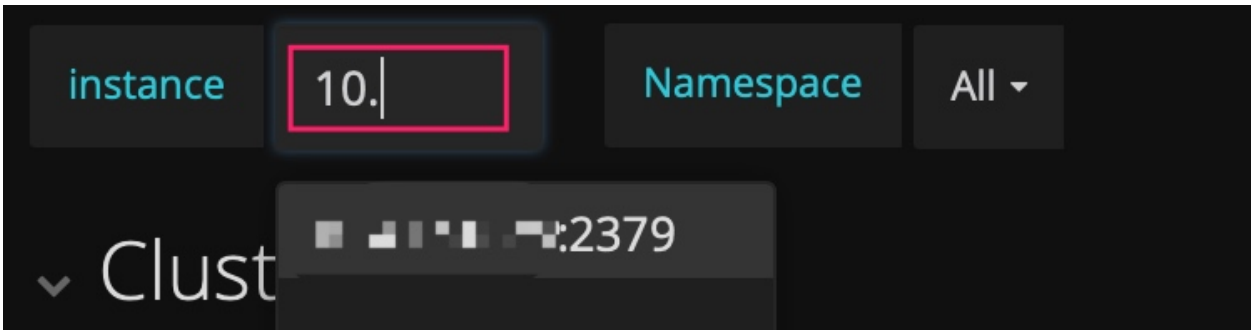


图 121: 查看历史 metric 信息

10.3.5.3.6 技巧 6：巧用 Avg 函数

通常默认图例中只有 Max 和 Current 函数。当指标波动较大时，可以增加 Avg 等其它汇总函数的图例，来看一段时间的整体趋势。

增加 Avg 等汇总函数：

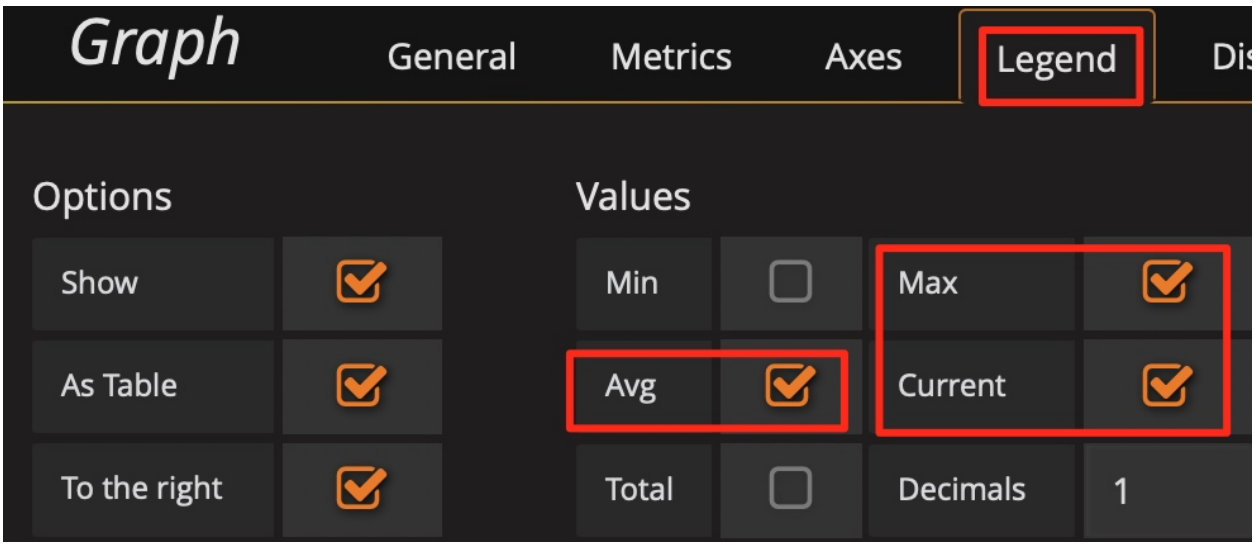


图 122: 增加 Avg 等汇总函数

然后查看整体趋势：

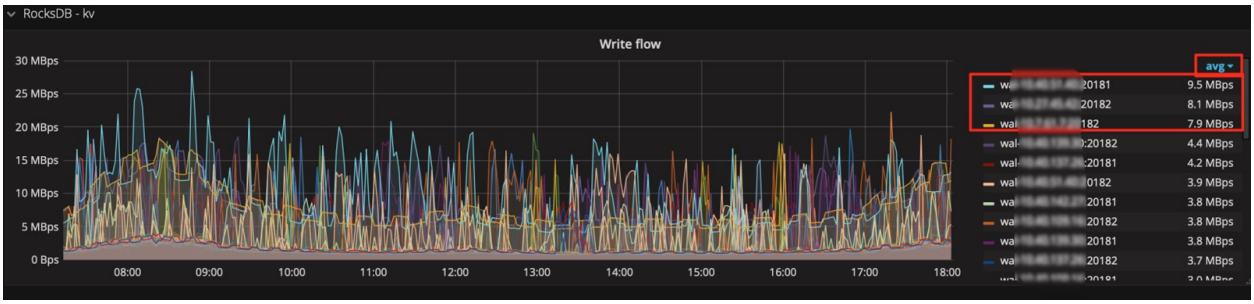


图 123: 增加 Avg 函数查看整体趋势

10.3.5.3.7 技巧 7：使用 Prometheus 的 API 接口获得表达式的结果

Grafana 通过 Prometheus 的接口获取数据，你也可以用该接口来获取数据，这个用法还可以衍生出许多功能：

- 自动获取集群规模、状态等信息。
- 对表达式稍加改动给报表提供数据，如统计每天的 QPS 总量、每天的 QPS 峰值和每天的响应时间。
- 将重要的指标进行定期健康巡检。

Prometheus 的 API 接口如下：

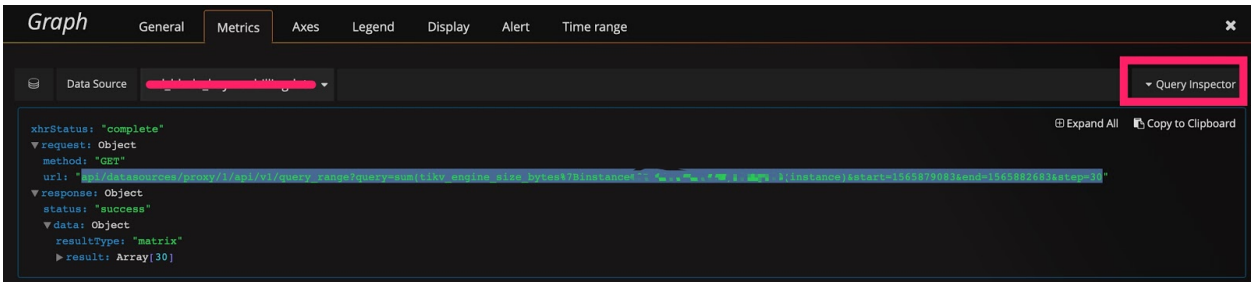


图 124: Prometheus 的 API 接口

```
curl -u user:pass 'http://__grafana_ip__:3000/api/datasources/proxy/1/api/v1/query_range?query=
    ↪ sum(tikv_engine_size_bytes%7Binstancexxxxxxxxx20181%22%7D)%20by%20(instance)&start
    ↪ =1565879269&end=1565882869&step=30' |python -m json.tool
```

```
{
    "data": {
        "result": [
            {
                "metric": {
                    "instance": "xxxxxxxxxx:20181"
                },
                "values": [
                    [
```

```
                    1565879269,
                    "1006046235280"
                ],
                [
                    1565879299,
                    "1006057877794"
                ],
                [
                    1565879329,
                    "1006021550039"
                ],
                [
                    1565879359,
                    "1006021550039"
                ],
                [
                    1565882869,
                    "1006132630123"
                ]
            ]
        }
    ],
    "resultType": "matrix"
},
"status": "success"
}
```

10.3.5.4 总结

Grafana + Prometheus 监控平台是一套非常强大的组合工具，用好这套工具可以为分析节省很多时间，提高效率，更重要的是，我们可以更容易发现问题。在运维 TiDB 集群，尤其是数据量大的情况下，这套工具能派上大用场。

10.3.6 PD 调度策略最佳实践

本文将详细介绍 PD 调度系统的原理，并通过几个典型场景的分析和处理方式，分享调度策略的最佳实践和调优方式，帮助大家在使用过程中快速定位问题。本文假定你对 TiDB，TiKV 以及 PD 已经有一定的了解，相关核心概念如下：

- Leader/Follower/Learner
- Operator
- Operator Step
- Pending/Down
- Region/Peer/Raft Group

- Region Split
- Scheduler
- Store

> 注意：
>
> 本文内容基于 TiDB 3.0 版本，更早的版本（2.x）缺少部分功能的支持，但是基本原理类似，也可以以本文作为参考。

10.3.6.1 PD 调度原理

该部分介绍调度系统涉及到的相关原理和流程。

10.3.6.1.1 调度流程

宏观上来看，调度流程大体可划分为 3 个部分：

1. 信息收集

   TiKV 节点周期性地向 PD 上报 `StoreHeartbeat` 和 `RegionHeartbeat` 两种心跳消息：

   - `StoreHeartbeat` 包含 Store 的基本信息、容量、剩余空间和读写流量等数据。
   - `RegionHeartbeat` 包含 Region 的范围、副本分布、副本状态、数据量和读写流量等数据。

   PD 梳理并转存这些信息供调度进行决策。

2. 生成调度

   不同的调度器从自身的逻辑和需求出发，考虑各种限制和约束后生成待执行的 Operator。这里所说的限制和约束包括但不限于：

   - 不往处于异常状态中（如断连、下线、繁忙、空间不足或在大量收发 snapshot 等）的 Store 添加副本
   - Balance 时不选择状态异常的 Region
   - 不尝试把 Leader 转移给 Pending Peer
   - 不尝试直接移除 Leader
   - 不破坏 Region 各种副本的物理隔离
   - 不破坏 Label property 等约束

3. 执行调度

   调度执行的步骤为：

   a. Operator 先进入一个由 `OperatorController` 管理的等待队列。

   b. `OperatorController` 会根据配置以一定的并发量从等待队列中取出 Operator 并执行。执行的过程就是依次把每个 Operator Step 下发给对应 Region 的 Leader。

   c. 标记 Operator 为 “finish” 或 “timeout” 状态，然后从执行列表中移除。

10.3.6.1.2 负载均衡

Region 负载均衡调度主要依赖 balance-leader 和 balance-region 两个调度器。二者的调度目标是将 Region 均匀地分散在集群中的所有 Store 上，但它们各有侧重：balance-leader 关注 Region 的 Leader，目的是分散处理客户端请求的压力；balance-region 关注 Region 的各个 Peer，目的是分散存储的压力，同时避免出现爆盘等状况。

balance-leader 与 balance-region 有着相似的调度流程：

1. 根据不同 Store 的对应资源量的情况分别打分。
2. 不断从得分较高的 Store 选择 Leader 或 Peer 迁移到得分较低的 Store 上。

两者的分数计算上也有一定差异：balance-leader 比较简单，使用 Store 上所有 Leader 所对应的 Region Size 加和作为得分。因为不同节点存储容量可能不一致，计算 balance-region 得分会分以下三种情况：

- 当空间富余时使用数据量计算得分（使不同节点数据量基本上均衡）
- 当空间不足时由使用剩余空间计算得分（使不同节点剩余空间基本均衡）
- 处于中间态时则同时考虑两个因素做加权和当作得分

此外，为了应对不同节点可能在性能等方面存在差异的问题，还可为 Store 设置负载均衡的权重。leader-weight 和 region-weight 分别用于控制 Leader 权重以及 Region 权重（默认值都为 “1”）。假如把某个 Store 的 leader-weight 设为 “2”，调度稳定后，则该节点的 Leader 数量约为普通节点的 2 倍；假如把某个 Store 的 region-weight 设为 “0.5”，那么调度稳定后该节点的 Region 数量约为其他节点的一半。

10.3.6.1.3 热点调度

热点调度对应的调度器是 hot-region-scheduler。在 3.0 版本中，统计热点 Region 的方式为：

1. 根据 Store 上报的信息，统计出持续一段时间读或写流量超过一定阈值的 Region。
2. 用与负载均衡类似的方式把这些 Region 分散开来。

对于写热点，热点调度会同时尝试打散热点 Region 的 Peer 和 Leader；对于读热点，由于只有 Leader 承载读压力，热点调度会尝试将热点 Region 的 Leader 打散。

10.3.6.1.4 集群拓扑感知

让 PD 感知不同节点分布的拓扑是为了通过调度使不同 Region 的各个副本尽可能分散，保证高可用和容灾。PD 会在后台不断扫描所有 Region，当发现 Region 的分布不是当前的最优化状态时，会生成调度以替换 Peer，将 Region 调整至最佳状态。

负责这个检查的组件叫 replicaChecker（跟 Scheduler 类似，但是不可关闭）。它依赖于 location-labels 配置项来进行调度。比如配置 [zone,rack,host] 定义了三层的拓扑结构：集群分为多个 zone（可用区），每个 zone 下有多个 rack（机架），每个 rack 下有多个 host（主机）。PD 在调度时首先会尝试将 Region 的 Peer 放置在不同的 zone，假如无法满足（比如配置 3 副本但总共只有 2 个 zone）则保证放置在不同的 rack；假如 rack 的数量也不足以保证隔离，那么再尝试 host 级别的隔离，以此类推。

10.3.6.1.5 缩容及故障恢复

缩容是指预备将某个 Store 下线，通过命令将该 Store 标记为 “Offline“ 状态，此时 PD 通过调度将待下线节点上的 Region 迁移至其他节点。

故障恢复是指当有 Store 发生故障且无法恢复时，有 Peer 分布在对应 Store 上的 Region 会产生缺少副本的状况，此时 PD 需要在其他节点上为这些 Region 补副本。

这两种情况的处理过程基本上是一样的。`replicaChecker` 检查到 Region 存在异常状态的 Peer 后，生成调度在健康的 Store 上创建新副本替换异常的副本。

10.3.6.1.6 Region merge

Region merge 指的是为了避免删除数据后大量小甚至空的 Region 消耗系统资源，通过调度把相邻的小 Region 合并的过程。Region merge 由 `mergeChecker` 负责，其过程与 `replicaChecker` 类似：PD 在后台遍历，发现连续的小 Region 后发起调度。

具体来说，当某个新分裂出来的 Region 存在的时间超过配置项`split-merge-interval` 的值（默认 1h）后，如果同时满足以下情况，该 Region 会触发 Region merge 调度：

- 该 Region 大小小于配置项`max-merge-region-size` 的值（默认 20MiB）
- 该 Region 中 key 的数量小于配置项`max-merge-region-keys` 的值（默认 200000）

10.3.6.2 查询调度状态

你可以通过观察 PD 相关的 Metrics 或使用 pd-ctl 工具等方式查看调度系统状态。更具体的信息可以参考PD 监控和PD Control。

10.3.6.2.1 Operator 状态

Grafana PD/Operator 页面展示了 Operator 的相关统计信息。其中比较重要的有：

- Schedule Operator Create：Operator 的创建情况
- Operator finish duration：Operator 执行耗时的情况
- Operator Step duration：不同 Operator Step 执行耗时的情况

查询 Operator 的 pd-ctl 命令有：

- `operator show`：查询当前调度生成的所有 Operator
- `operator show [admin | leader | region]`：按照类型查询 Operator

10.3.6.2.2 Balance 状态

Grafana PD/Statistics - Balance 页面展示了负载均衡的相关统计信息，其中比较重要的有：

- Store Leader/Region score：每个 Store 的得分
- Store Leader/Region count：每个 Store 的 Leader/Region 数量
- Store available：每个 Store 的剩余空间

使用 pd-ctl 的 `store` 命令可以查询 Store 的得分、数量、剩余空间和 weight 等信息。

10.3.6.2.3 热点调度状态

Grafana PD/Statistics - hotspot 页面展示了热点 Region 的相关统计信息，其中比较重要的有：

- Hot write Region’s leader/peer distribution：写热点 Region 的 Leader/Peer 分布情况
- Hot read Region’s leader distribution：读热点 Region 的 Leader 分布情况

使用 pd-ctl 同样可以查询上述信息，可以使用的命令有：

- `hot read`：查询读热点 Region 信息
- `hot write`：查询写热点 Region 信息
- `hot store`：按 Store 统计热点分布情况
- `region topread [limit]`：查询当前读流量最大的 Region
- `region topwrite [limit]`：查询当前写流量最大的 Region

10.3.6.2.4 Region 健康度

Grafana PD/Cluster/Region health 面板展示了异常 Region 的相关统计信息，包括 Pending Peer、Down Peer、Offline Peer，以及副本数过多或过少的 Region。

通过 pd-ctl 的 `region check` 命令可以查看具体异常的 Region 列表：

- `region check miss-peer`：缺副本的 Region
- `region check extra-peer`：多副本的 Region
- `region check down-peer`：有副本状态为 Down 的 Region
- `region check pending-peer`：有副本状态为 Pending 的 Region

10.3.6.3 调度策略控制

使用 pd-ctl 可以从以下三个方面来调整 PD 的调度策略。更具体的信息可以参考PD Control。

10.3.6.3.1 启停调度器

pd-ctl 支持动态创建和删除 Scheduler，你可以通过这些操作来控制 PD 的调度行为，如：

- `scheduler show`：显示当前系统中的 Scheduler
- `scheduler remove balance-leader-scheduler`：删除（停用）balance region 调度器
- `scheduler add evict-leader-scheduler 1`：添加移除 Store 1 的所有 Leader 的调度器

10.3.6.3.2 手动添加 Operator

PD 支持直接通过 pd-ctl 来创建或删除 Operator，如：

- `operator add add-peer 2 5`：在 Store 5 上为 Region 2 添加 Peer
- `operator add transfer-leader 2 5`：将 Region 2 的 Leader 迁移至 Store 5
- `operator add split-region 2`：将 Region 2 拆分为 2 个大小相当的 Region
- `operator remove 2`：取消 Region 2 当前待执行的 Operator

10.3.6.3.3 调度参数调整

使用 pd-ctl 执行 `config show` 命令可以查看所有的调度参数，执行 `config set {key} {value}` 可以调整对应参数的值。常见调整如下：

- `leader-schedule-limit`：控制 Transfer Leader 调度的并发数
- `region-schedule-limit`：控制增删 Peer 调度的并发数
- `enable-replace-offline-replica`：开启节点下线的调度
- `enable-location-replacement`：开启调整 Region 隔离级别相关的调度
- `max-snapshot-count`：每个 Store 允许的最大收发 Snapshot 的并发数

10.3.6.4 典型场景分析与处理

该部分通过几个典型场景及其应对方式说明 PD 调度策略的最佳实践。

10.3.6.4.1 Leader/Region 分布不均衡

PD 的打分机制决定了一般情况下，不同 Store 的 Leader Count 和 Region Count 不能完全说明负载均衡状态，所以需要从 TiKV 的实际负载或者存储空间占用来判断是否有负载不均衡的状况。

确认 Leader/Region 分布不均衡后，首先观察不同 Store 的打分情况。

如果不同 Store 的打分是接近的，说明 PD 认为此时已经是均衡状态了，可能的原因有：

- 存在热点导致负载不均衡。可以参考热点分布不均匀中的解决办法进行分析处理。
- 存在大量空 Region 或小 Region，因此不同 Store 的 Leader 数量差别特别大，导致 Raftstore 负担过重。此时需要开启Region Merge 并尽可能加速合并。
- 不同 Store 的软硬件环境存在差异。可以酌情调整 `leader-weight` 和 `region-weight` 来控制 Leader/Region 的分布。
- 其他不明原因。仍可以通过调整 `leader-weight` 和 `region-weight` 来控制 Leader/Region 的分布。

如果不同 Store 的分数差异较大，需要进一步检查 Operator 的相关 Metrics，特别关注 Operator 的生成和执行情况，这时大体上又分两种情况：

- 生成的调度是正常的，但是调度的速度很慢。可能的原因有：
  - 调度速度受限于 limit 配置。PD 默认配置的 limit 比较保守，在不对正常业务造成显著影响的前提下，可以酌情将 `leader-schedule-limit` 或 `region-schedule-limit` 调大一些。此外，`max-pending-peer-count` 以及 `max-snapshot-count` 限制也可以放宽。
  - 系统中同时运行有其他的调度任务产生竞争，导致 balance 速度上不去。这种情况下如果 balance 调度的优先级更高，可以先停掉其他的调度或者限制其他调度的速度。例如 Region 没均衡的情况下做下线节点操作，下线的调度与 Region Balance 会抢占 `region-schedule-limit` 配额，此时你可以调小 `replica-schedule-limit` 以限制下线调度的速度，或者设置 `enable-replace-offline-replica = false` 来暂时关闭下线流程。
  - 调度执行得太慢。可以通过 Operator step duration 进行判断。通常不涉及到收发 Snapshot 的 Step（比如 `TransferLeader`，`RemovePeer`，`PromoteLearner` 等）的完成时间应该在毫秒级，涉及到 Snapshot 的 Step（如 `AddLearner`，`AddPeer` 等）的完成时间为数十秒。如果耗时明显过高，可能是 TiKV 压力过大或者网络等方面的瓶颈导致的，需要具体情况具体分析。

- 没能生成对应的 balance 调度。可能的原因有：
  - 调度器未被启用。比如对应的 Scheduler 被删除了，或者 limit 被设置为 “0”。
  - 由于其他约束无法进行调度。比如系统中有 `evict-leader-scheduler`，此时无法把 Leader 迁移至对应的 Store。再比如设置了 Label property，也会导致部分 Store 不接受 Leader。
  - 集群拓扑的限制导致无法均衡。比如 3 副本 3 数据中心的集群，由于副本隔离的限制，每个 Region 的 3 个副本都分别分布在不同的数据中心，假如这 3 个数据中心的 Store 数不一样，最后调度就会收敛在每个数据中心均衡，但是全局不均衡的状态。

10.3.6.4.2 节点下线速度慢

这个场景需要从 Operator 相关的 Metrics 入手，分析 Operator 的生成执行情况。

如果调度在正常生成，只是速度很慢，可能的原因有：

- 调度速度受限于 limit 配置。可以适当调大 `replica-schedule-limit`，`max-pending-peer-count` 以及 `max-snapshot-count` 限制也可以放宽。
- 系统中同时运行有其他的调度任务产生竞争。处理方法参考Leader/Region 分布不均衡。
- 下线单个节点时，由于待操作的 Region 有很大一部分（3 副本配置下约 1/3）的 Leader 都集中在下线的节点上，下线速度会受限于这个单点生成 Snapshot 的速度。你可以通过手动给该节点添加一个 `evict-leader-scheduler` 调度器迁走 Leader 来加速。

如果没有对应的 Operator 调度生成，可能的原因有：

- 下线调度被关闭，或者 `replica-schedule-limit` 被设为 “0”。
- 找不到节点来转移 Region。例如相同 Label 的替代节点可用容量都不足 20%，PD 为了避免爆盘的风险会停止调度。这种情况需要添加更多节点，或者删除一些数据释放空间。

10.3.6.4.3 节点上线速度慢

目前 PD 没有对节点上线特殊处理。节点上线实际上是依靠 balance region 机制来调度的，所以参考Leader/Region 分布不均衡 中的排查步骤即可。

10.3.6.4.4 热点分布不均匀

热点调度的问题大体上可以分为以下几种情况：

- 从 PD 的 Metrics 能看出来有不少 hot Region，但是调度速度跟不上，不能及时地把热点 Region 分散开来。

解决方法：调大 `hot-region-schedule-limit` 并减少其他调度器的 limit 配额，从而加快热点调度的速度。还可调小 `hot-region-cache-hits-threshold` 使 PD 对更快响应流量的变化。

- 单一 Region 形成热点，比如大量请求频繁 scan 一个小表，这个可以从业务角度或者 Metrics 统计的热点信息看出来。由于单 Region 热点现阶段无法使用打散的手段来消除，需要确认热点 Region 后手动添加 `split-region` 调度将这样的 Region 拆开。

- 从 PD 的统计来看没有热点，但是从 TiKV 的相关 Metrics 可以看出部分节点负载明显高于其他节点，成为整个系统的瓶颈。这是因为目前 PD 统计热点 Region 的维度比较单一，仅针对流量进行分析，在某些场景下无法准确定位热点。例如部分 Region 有大量的点查请求，从流量上来看并不显著，但是过高的 QPS 导致关键模块达到瓶颈。

  解决方法：首先从业务层面确定形成热点的 table，然后添加 `scatter-range-scheduler` 调度器使这个 table 的所有 Region 均匀分布。TiDB 也在其 HTTP API 中提供了相关接口来简化这个操作，具体可以参考 TiDB HTTP API 文档。

10.3.6.4.5 Region Merge 速度慢

Region Merge 速度慢也很有可能是受到 limit 配置的限制（`merge-schedule-limit` 及 `region-schedule-limit`），或者是与其他调度器产生了竞争。具体来说，可有如下处理方式：

- 假如已经从相关 Metrics 得知系统中有大量的空 Region，这时可以通过把 `max-merge-region-size` 和 `max-merge-region-keys` 调整为较小值来加快 Merge 速度。这是因为 Merge 的过程涉及到副本迁移，所以 Merge 的 Region 越小，速度就越快。如果生成 Merge Operator 的速度很快，想进一步加快 Region Merge 过程，还可以把 `patrol-region-interval` 调整为“10ms”，这个能加快巡检 Region 的速度，但是会消耗更多的 CPU 资源。
- 创建过大量表后（包括执行 `Truncate Table` 操作）又清空了。此时如果开启了 split table 特性，这些空 Region 是无法合并的，此时需要调整以下参数关闭这个特性：
  - TiKV: 将 `split-region-on-table` 设为 `false`，该参数不支持动态修改。
  - PD:
    * `key-type` 设为 `txn` 或者 `raw`，该参数支持动态修改。
    * 或者 `key-type` 保持 `table`，同时设置 `enable-cross-table-merge`为 `true`，该参数支持动态修改。

> 注意：
> 在开启 `placement-rules`后，请合理切换 `txn`和 `raw`，避免无法正常解码 key。

- 对于 3.0.4 和 2.1.16 以前的版本，Region 中 Key 的个数（`approximate_keys`）在特定情况下（大部分发生在删表之后）统计不准确，造成 keys 的统计值很大，无法满足 `max-merge-region-keys` 的约束。你可以通过调大 `max-merge-region-keys` 来避免这个问题。

10.3.6.4.6 TiKV 节点故障处理策略

没有人工介入时，PD 处理 TiKV 节点故障的默认行为是，等待半小时之后（可通过 `max-store-down-time` 配置调整），将此节点设置为 Down 状态，并开始为涉及到的 Region 补充副本。

实践中，如果能确定这个节点的故障是不可恢复的，可以立即做下线处理，这样 PD 能尽快补齐副本，降低数据丢失的风险。与之相对，如果确定这个节点是能恢复的，但可能半小时之内来不及，则可以把 `max-store-down-time` 临时调整为比较大的值，这样能避免超时之后产生不必要的副本补充，造成资源浪费。

10.3.7 海量 Region 集群调优最佳实践

在 TiDB 的架构中，所有数据以一定 key range 被切分成若干 Region 分布在多个 TiKV 实例上。随着数据的写入，一个集群中会产生上百万个甚至千万个 Region。单个 TiKV 实例上产生过多的 Region 会给集群带来较大的负担，影响整个集群的性能表现。

本文将介绍 TiKV 核心模块 Raftstore 的工作流程，海量 Region 导致性能问题的原因，以及优化性能的方法。

10.3.7.1 Raftstore 的工作流程

一个 TiKV 实例上有多个 Region。Region 消息是通过 Raftstore 模块驱动 Raft 状态机来处理的。这些消息包括 Region 上读写请求的处理、Raft log 的持久化和复制、Raft 的心跳处理等。但是，Region 数量增多会影响整个集群的性能。为了解释这一点，需要先了解 TiKV 的核心模块 Raftstore 的工作流程。

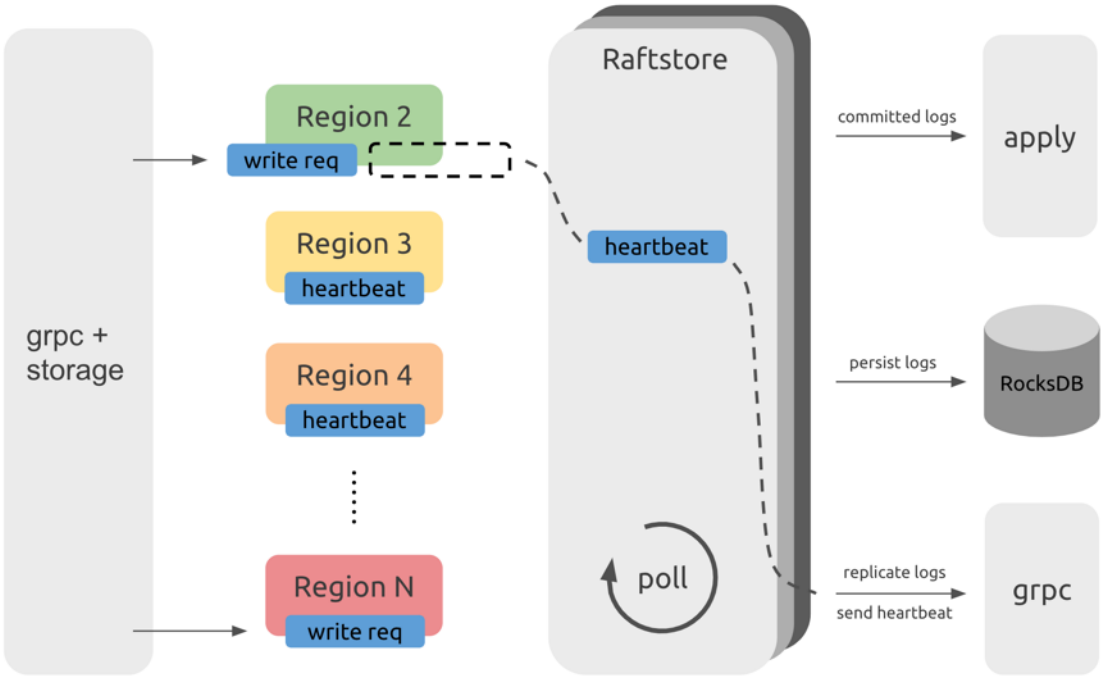


图 125: 图 1 Raftstore 处理流程示意图

> **注意：**
>
> 该图仅为示意，不代表代码层面的实际结构。

上图是 Raftstore 处理流程的示意图。如图所示，从 TiDB 发来的请求会通过 gRPC 和 storage 模块变成最终的 KV 读写消息，并被发往相应的 Region，而这些消息并不会被立即处理而是被暂存下来。Raftstore 会轮询检查每个 Region 是否有需要处理的消息。如果 Region 有需要处理的消息，那么 Raftstore 会驱动 Raft 状态机去处理这些消息，并根据这些消息所产生的状态变更去进行后续操作。例如，在有写请求时，Raft 状态机需要将日志落盘并且将日志发送给其他 Region 副本；在达到心跳间隔时，Raft 状态机需要将心跳信息发送给其他 Region 副本。

10.3.7.2 性能问题

从 Raftstore 处理流程示意图可以看出，需要依次处理各个 Region 的消息。那么在 Region 数量较多的情况下，Raftstore 需要花费一些时间去处理大量 Region 的心跳，从而带来一些延迟，导致某些读写请求得不到及时处理。如果读写压力较大，Raftstore 线程的 CPU 使用率容易达到瓶颈，导致延迟进一步增加，进而影响性能表现。

通常在有负载的情况下，如果 Raftstore 的 CPU 使用率达到了 85% 以上，即可视为达到繁忙状态且成为了瓶颈，同时 `propose wait duration` 可能会高达百毫秒级别。

> 注意：
>
> - Raftstore 的 CPU 使用率是指单线程的情况。如果是多线程 Raftstore，可等比例放大使用率。
> - 由于 Raftstore 线程中有 I/O 操作，所以 CPU 使用率不可能达到 100%。

10.3.7.2.1 性能监控

可在 Grafana 的 TiKV 面板下查看相关的监控 metrics：

- Thread-CPU 下的 `Raft store CPU`

  参考值：低于 `raftstore.store-pool-size * 85%`。

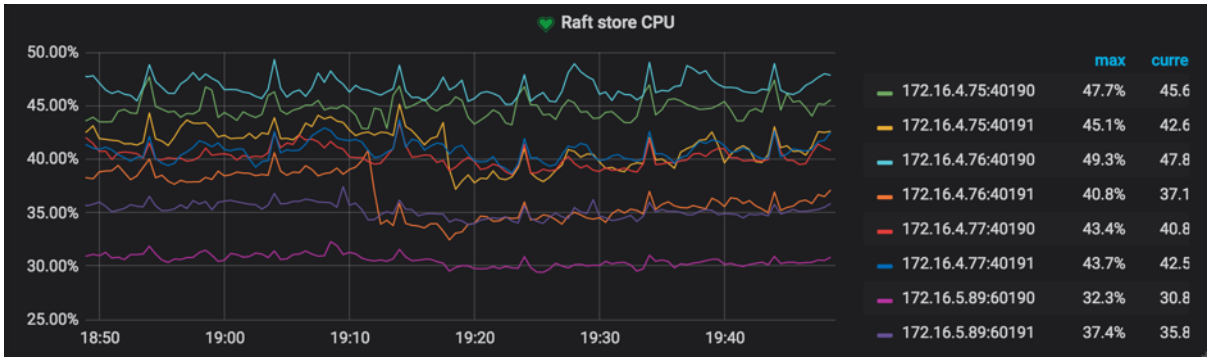


图 126: 图 2 查看 Raftstore CPU

- Raft Propose 下的 `Propose wait duration`

  `Propose wait duration` 是从发送请求给 Raftstore，到 Raftstore 真正开始处理请求之间的延迟时间。如果该延迟时间较长，说明 Raftstore 比较繁忙或者处理 append log 比较耗时导致 Raftstore 不能及时处理请求。

  参考值：低于 50-100ms。

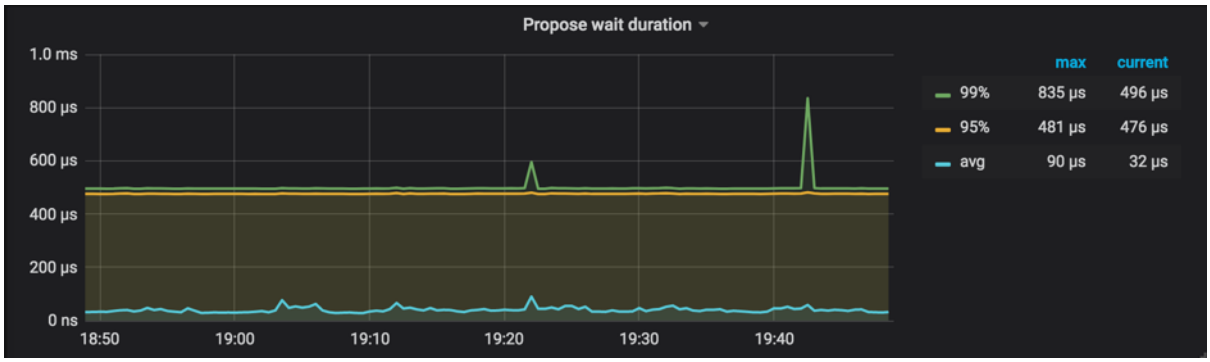


图 127: 图 3 查看 Propose wait duration

10.3.7.3 性能优化方法

找到性能问题的根源后，可从以下两个方向来解决性能问题：

- 减少单个 TiKV 实例的 Region 数
- 减少单个 Region 的消息数

10.3.7.3.1 方法一：增加 TiKV 实例

如果 I/O 资源和 CPU 资源都比较充足，可在单台机器上部署多个 TiKV 实例，以减少单个 TiKV 实例上的 Region 个数；或者增加 TiKV 集群的机器数。

10.3.7.3.2 方法二：调整 `raft-base-tick-interval`

除了减少 Region 个数外，还可以通过减少 Region 单位时间内的消息数量来减小 Raftstore 的压力。例如，在 TiKV 配置中适当调大 `raft-base-tick-interval`：

```
[raftstore]
raft-base-tick-interval = "2s"
```

`raft-base-tick-interval` 是 Raftstore 驱动每个 Region 的 Raft 状态机的时间间隔，也就是每隔该时长就需要向 Raft 状态机发送一个 tick 消息。增加该时间间隔，可以有效减少 Raftstore 的消息数量。

需要注意的是，该 tick 消息的间隔也决定了 `election timeout` 和 `heartbeat` 的间隔。示例如下：

```
raft-election-timeout = raft-base-tick-interval * raft-election-timeout-ticks
raft-heartbeat-interval = raft-base-tick-interval * raft-heartbeat-ticks
```

如果 Region Follower 在 `raft-election-timeout` 间隔内未收到来自 Leader 的心跳，就会判断 Leader 出现故障而发起新的选举。`raft-heartbeat-interval` 是 Leader 向 Follower 发送心跳的间隔，因此调大 `raft-base-tick-interval` 可以减少单位时间内 Raft 发送的网络消息，但也会让 Raft 检测到 Leader 故障的时间更长。

10.3.7.3.3 方法三：提高 Raftstore 并发数

从 v3.0 版本起，Raftstore 已经扩展为多线程，极大降低了 Raftstore 线程成为瓶颈的可能性。

TiKV 默认将 `raftstore.store-pool-size` 配置为 2。如果 Raftstore 出现瓶颈，可以根据实际情况适当调高该参数值，但不建议设置过高以免引入不必要的线程切换开销。

10.3.7.3.4 方法四：开启 Hibernate Region 功能

在实际情况中，读写请求并不会均匀分布到每个 Region 上，而是集中在少数的 Region 上。那么可以尽量减少暂时空闲的 Region 的消息数量，这也就是 Hibernate Region 的功能。无必要时可不进行 `raft-base-tick`，即不驱动空闲 Region 的 Raft 状态机，那么就不会触发这些 Region 的 Raft 产生心跳信息，极大地减小了 Raftstore 的工作负担。

Hibernate Region 在 TiKV master 分支上默认开启。可根据实际情况和需求来开启该功能。Hibernate Region 的配置说明请参考配置 Hibernate Region。

10.3.7.3.5 方法五：开启 Region Merge

> 注意：
>
> 从 TiDB v3.0 开始，Region Merge 默认开启。

开启 Region Merge 也能减少 Region 的个数。与 Region Split 相反，Region Merge 是通过调度把相邻的小 Region 合并的过程。在集群中删除数据或者执行 Drop Table/Truncate Table 语句后，可以将小 Region 甚至空 Region 进行合并以减少资源的消耗。

通过 pd-ctl 设置以下参数即可开启 Region Merge：

```
>> pd-ctl config set max-merge-region-size 20
>> pd-ctl config set max-merge-region-keys 200000
>> pd-ctl config set merge-schedule-limit 8
```

详情请参考如何配置 Region Merge（英文）和PD 配置文件描述。

同时，默认配置的 Region Merge 的参数设置较为保守，可以根据需求参考PD 调度策略最佳实践 中提供的方法加快 Region Merge 过程的速度。

10.3.7.4 其他问题和解决方案

10.3.7.4.1 切换 PD Leader 的速度慢

PD 需要将 Region Meta 信息持久化在 etcd 上，以保证切换 PD Leader 节点后 PD 能快速继续提供 Region 路由服务。随着 Region 数量的增加，etcd 出现性能问题，使得 PD 在切换 Leader 时从 etcd 获取 Region Meta 信息的速度较慢。在百万 Region 量级时，从 etcd 获取信息的时间可能需要十几秒甚至几十秒。

因此从 v3.0 版本起，PD 默认开启配置项 `use-region-storage`，将 Region Meta 信息存在本地的 LevelDB 中，并通过其他机制同步 PD 节点间的信息。

10.3.7.4.2 PD 路由信息更新不及时

在 TiKV 中，pd-worker 模块将 Region Meta 信息定期上报给 PD，在 TiKV 重启或者切换 Region Leader 时需要通过统计信息重新计算 Region 的 `approximate size/keys`。因此在 Region 数量较多的情况下，pd-worker 单线程可能成

为瓶颈，造成任务得不到及时处理而堆积起来。因此 PD 不能及时获取某些 Region Meta 信息以致路由信息更新不及时。该问题不会影响实际的读写，但可能导致 PD 调度不准确以及 TiDB 更新 Region cache 时需要多几次 round-trip。

可在 TiKV Grafana 面板中查看 Task 下的 Worker pending tasks 来确定 pd-worker 是否有任务堆积。通常来说，pending tasks 应该维持在一个比较低的值。

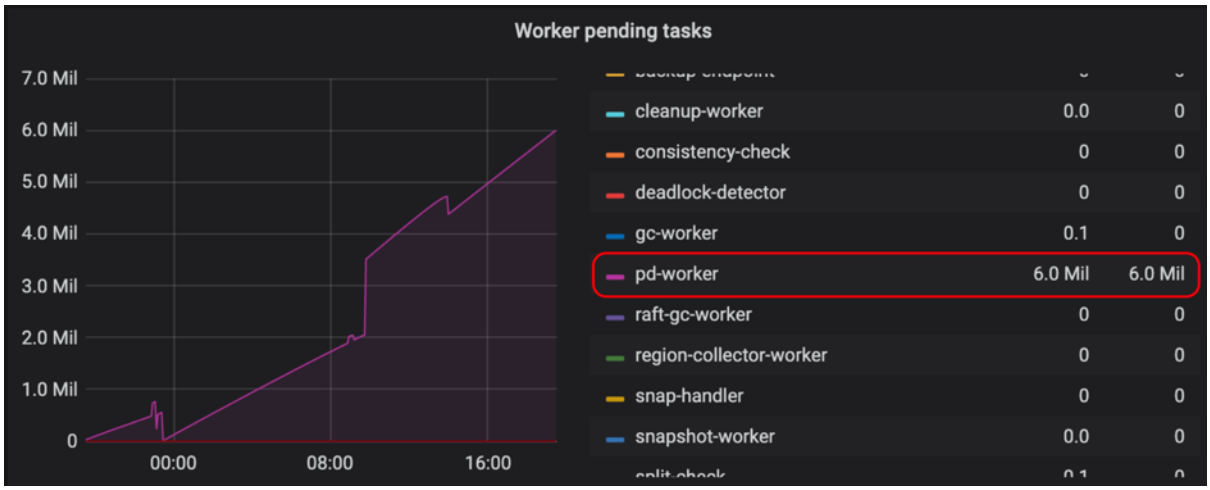


图 128: 图 4 查看 pd-worker

老版本 TiDB (< v3.0.5) pd-worker 的效率有一些缺陷，如果碰到类似问题，建议升级至最新版本。

10.3.7.4.3 Prometheus 查询 metrics 的速度慢

在大规模集群中，随着 TiKV 实例数的增加，Prometheus 查询 metrics 时的计算压力较大，导致 Grafana 查看 metrics 的速度较慢。从 v3.0 版本起设置了一些 metrics 的预计算，让这个问题有所缓解。

10.3.8 三节点混合部署的最佳实践

在对性能要求不高且需要控制成本的场景下，将 TiDB、TiKV、PD 混合部署在三台机器上是一个可行的方案。

本文以 TPC-C 作为工作负载，提供一些在三节点混合部署场景下部署和参数调整的建议。

10.3.8.1 部署环境和测试方法

三台物理机，每台机器有 16 个 CPU 核心且内存为 32 GB。每节点（台机器）上混合部署 1 TiDB （实例）+ 1 TiKV （实例）+ 1 PD （实例）。

由于 PD 和 TiKV 都会存储信息到磁盘，磁盘的写入读取延迟会直接影响到 PD 和 TiKV 的服务延迟，为了防止 PD 和 TiKV 对磁盘资源的争抢导致相互影响，推荐 PD 和 TiKV 采用不同的磁盘。

在 TiUP bench 中使用 TPC-C 5000 Warehouse 数据，每次以 `terminals` 参数为 128 的并发度测试 12 小时。主要观察集群性能的稳定性指标。

下图是默认参数配置下，12 小时内集群的 QPS 监控，可以看倒有比较明显的抖动。

图 129: QPS with default config

调整参数后，稳定性得到了改善。

图 130: QPS with modified config

10.3.8.2 参数调整

上图中出现抖动的主要原因是，默认的线程池和后台任务资源分配针对的是资源比较充足的机器。在混合部署的场景下，因为可用资源需要被多个组件共享，所以需要通过配置参数进行限制。

本次测试最终采用的配置如下：

```
tikv:
    readpool.unified.max-thread-count: 6
    server.grpc-concurrency: 2
    storage.scheduler-worker-pool-size: 2
    gc.max-write-bytes-per-sec: 300K
    rocksdb.max-background-jobs: 3
    rocksdb.max-sub-compactions: 1
    rocksdb.rate-bytes-per-sec: “200M”

  tidb:
    performance.committer-concurrency: 4
    performance.max-procs: 8
```

接下来会分别介绍这几个参数的意义和调整方法。

10.3.8.2.1 TiKV 线程池大小配置

本节的配置项会调整一些与前台业务有关的线程池的资源配比。缩减这些线程池会损失一些性能，但在混合部署场景下可用资源有限，本身也难以达到很高的性能，所以选择牺牲一小部分性能去换取整体的稳定性。

可以先以默认配置为基础运行一次实际负载的测试，观察各线程池的实际使用量，再修改这些配置项，缩减利用率不高的线程池大小。

`readpool.unified.max-thread-count`

该参数默认取值为机器线程数的 80%，因为是混合部署场景，你需要手动计算并指定该值。可以先设置成期望 TiKV 使用的 CPU 线程数的 80%。

`server.grpc-concurrency`

该参数默认为 4。因为现有部署方案的 CPU 资源有限，实际请求数也不会很多。可以把该参数值调低，后续观察监控面板保持其使用率在 80% 以下即可。

本次测试最后选择设置该参数值为 2，通过 gRPC poll CPU 面板观察，利用率正好在 80% 左右。

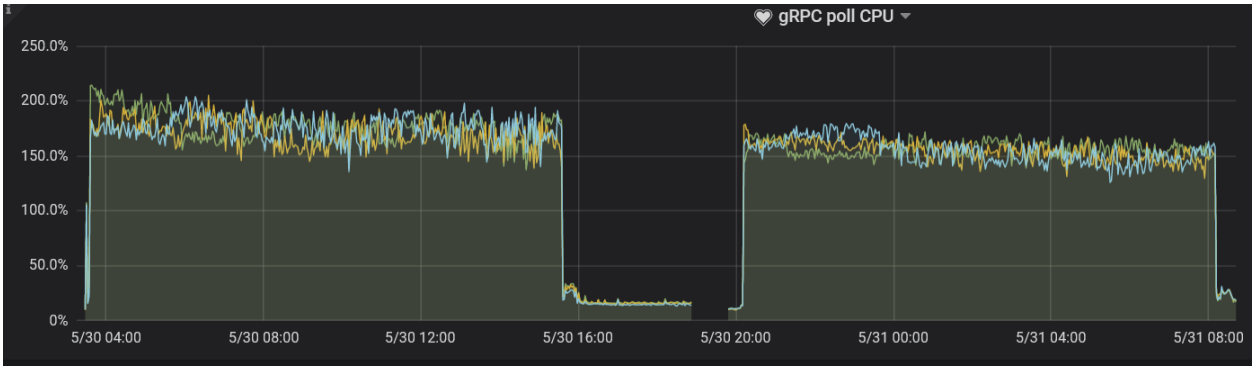


图 131: gRPC Pool CPU

`storage.scheduler-worker-pool-size`

在 TiKV 检测到机器 CPU 数大于或等于 16 时，该参数值默认为 8。CPU 数小于 16 时，该参数值默认为 4。它主要用于将复杂的事务请求转化为简单的 key-value 读写。但是 scheduler 线程池本身不进行任何写操作。

一般来说该线程池的利用率保持在 50% - 75% 之间是比较好的。和 gRPC 线程池情况类似，混合部署时该参数默认取值偏大，资源利用不充分。本次测试最后选择取值为 2，通过 Scheduler worker CPU 面板观察，利用率比较符合最佳实践。

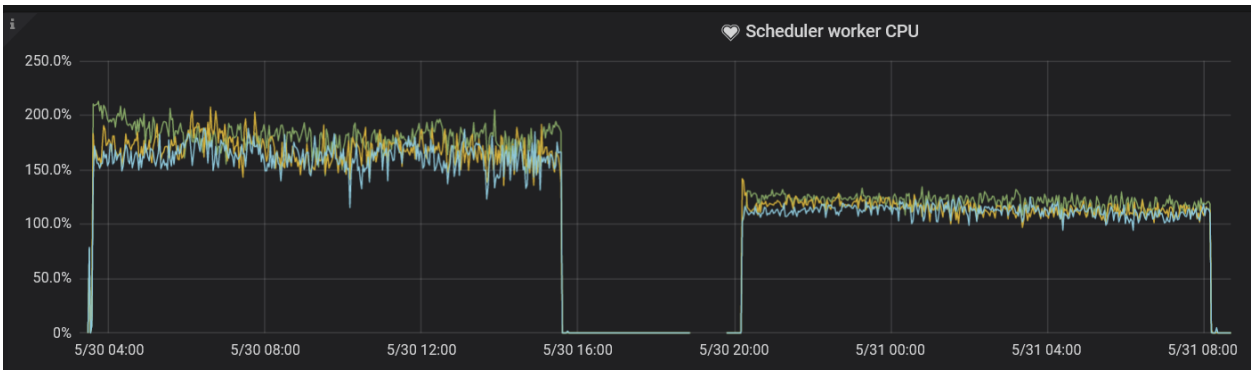


图 132: Scheduler Worker CPU

10.3.8.2.2 TiKV 后台任务资源配置

除前台任务之外，TiKV 上还会持续地有后台任务进行数据整理和过期数据的清除。默认配置为这些后台任务分配了比较多的资源以应对大流量的写入。

混合部署场景下，默认配置就不是很合适了，需要通过以下参数对这些后台任务地资源使用量进行限制。

`rocksdb.max-background-jobs` 和 `rocksdb.max-sub-compactions`

RocksDB 线程池是进行 Compact 和 Flush 任务的线程池，默认大小为 8。这明显超出了实际可以使用的资源，需要限制。`rocksdb.max-sub-compactions` 是单个 compaction 任务的子任务并发数，默认值为 3，在写入流量不大的情况下可以进行限制。

这次测试最终将 `rocksdb.max-background-jobs` 设置为 3，将 `rocksdb.max-sub-compactions` 设置为 1。在 12 小时的 TPC-C 负载下没有发生 Write Stall，根据实际负载进行这两项参数的优化时，可以逐步调低这两个配置，并通过监控观察。

- 如果遇到了 Write Stall，可以先调大 `rocksdb.max-background-jobs` 的取值。
- 如果还是存在问题，可将 `rocksdb.max-sub-compactions` 设置为 2 或者 3。

`rocksdb.rate-bytes-per-sec`

该参数用于限制后台 compaction 任务的磁盘流量。默认配置下没有进行任何限制。为了避免 compaction 挤占前台服务资源，可以根据硬盘的顺序读写速度进行调整，为正常的服务保留足够多的磁盘带宽。

类似 compaction 线程池的调整方法，调整该参数值后，也根据是否存在 Write Stall 来判断取值是否合理。

`gc.max_write_bytes_per_sec`

因为 TiDB 使用的是 MVCC 的模型，TiKV 还需要周期性地在后台清除旧版本的数据。当可用资源有限的时候，这个操作会引起周期性的性能抖动。可以用 `gc.max_write_bytes_per_sec` 来限制这一操作的资源使用。

图 133: GC Impact

除了在配置文件中设置该参数之外，还可以通过 tikv-ctl 动态调节，为调整该参数提供便利：

```
tiup ctl:<cluster-version> tikv --host=${ip:port} modify-tikv-config -n gc.
    ↪ max_write_bytes_per_sec -v ${limit}
```

> **注意：**
>
> 对于更新频繁的业务场景，限制 GC 流量可能会导致 MVCC 版本堆积，进而影响读取性能。所以这个参数的取值需要进行多次尝试，在性能抖动和性能衰退之间找一个比较平衡的取值。

10.3.8.2.3 TiDB 参数调整

一般通过系统变量调整 TiDB 执行算子的优化参数，例如 `tidb_hash_join_concurrency`、`tidb_index_lookup_join_concurrency` 等。

本测试中没有对这些参数进行调整。如果实际的业务负载测试中，出现执行算子消耗过多 CPU 资源的情况，可以针对业务场景对特定的算子资源使用量进行限制。这部分内容可以参考TiDB 系统变量文档。

`performance.max-procs`

该参数控制着整个 Go 进程能使用的 CPU 核心数量，默认情况下为当前机器或者 cgroups 的 CPU 数量。

Go 运行时会定期使用一定比例的线程进行 GC 等后台工作。在混合部署模式下，如果不对这一参数进行限制，GC 等后台操作会过多占用 CPU 资源。

10.4 Placement Rules 使用文档

> **注意：**
>
> 在配合使用 TiFlash 场景下，Placement Rules 功能进行过大量测试，可以在生产环境中使用。除配合使用 TiFlash 的场景外，单独开启 Placement Rules 没有经过大量测试，因此，不建议在生产环境单独开启该功能。

Placement Rules 是 PD 在 4.0 版本引入的试验特性，它是一套副本规则系统，用于指导 PD 针对不同类型的数据生成对应的调度。通过组合不同的调度规则，用户可以精细地控制任何一段连续数据的副本数量、存放位置、主机类型、是否参与 Raft 投票、是否可以担任 Raft leader 等属性。

10.4.1 规则系统介绍

整个规则系统的配置由多条规则即 Rule 组成。每条 Rule 可以指定不同的副本数量、 Raft 角色、放置位置等属性，以及这条规则生效的 key range。PD 在进行调度时，会先根据 Region 的 key range 在规则系统中查到该 Region 对应的规则，然后再生成对应的调度，来使得 Region 副本的分布情况符合 Rule。

多条规则的 key range 可以有重叠部分的，即一个 Region 能匹配到多条规则。这种情况下 PD 根据 Rule 的属性来决定规则是相互覆盖还是同时生效。如果有多条规则同时生效，PD 会按照规则的堆叠次序依次去生成调度进行规则匹配。

此外，为了满足不同来源的规则相互隔离的需求，还引入了分组（Group）的概念。如果某条规则不希望与系统中的其他规则相互影响（比如被覆盖），可以使用单独的分组。

Placement Rules 示意图如下所示：

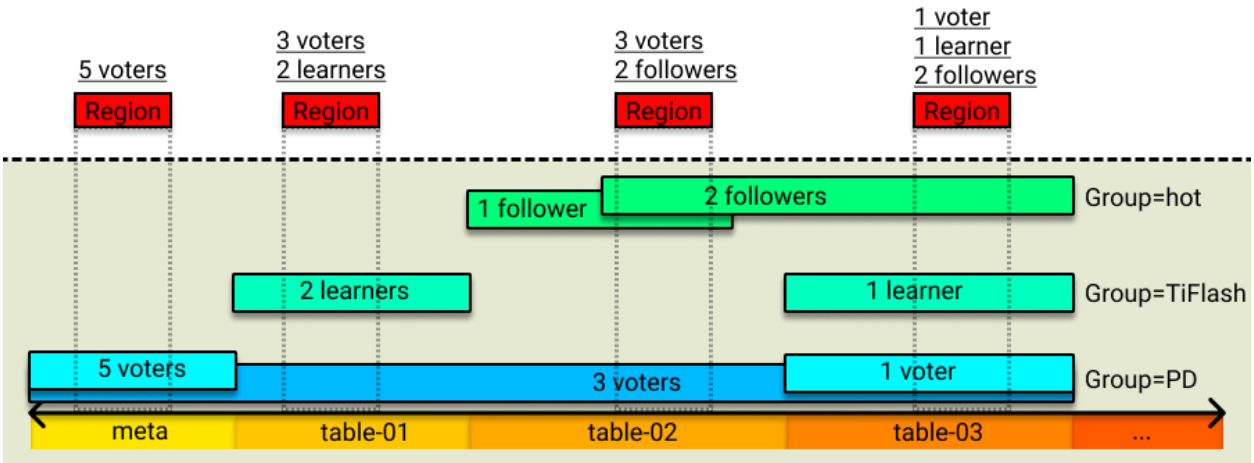


图 134: Placement rules overview

10.4.1.1 规则字段

以下是每条规则中各个字段的具体含义：

字段名	类型及约束	说明
GroupID	string	分组 ID，标识规则的来源
ID	string	分组内唯一 ID
Index	int	分组内堆叠次序
Override	true/false	是否覆盖 index 的更小 Rule（限分组内）
StartKey	string，十六进制编码	适用 Range 起始 key
EndKey	string，十六进制编码	适用 Range 终止 key
Role	string	副本角色，包括 leader/follower/learner
Count	int，正整数	副本数量
LabelConstraint	[]Constraint	用于按 label 筛选节点
LocationLabels	[]string	用于物理隔离

LabelConstraint 与 Kubernetes 中的功能类似，支持通过 in、notIn、exists 和 notExists 四种原语来筛选 label。这四种原语的意义如下：

- in：给定 key 的 label value 包含在给定列表中。

- `notIn`：给定 key 的 label value 不包含在给定列表中。
- `exists`：包含给定的 label key。
- `notExists`：不包含给定的 label key。

`LocationLabels` 的意义和作用与 PD v4.0 之前的版本相同。比如配置 `[zone,rack,host]` 定义了三层的拓扑结构：集群分为多个 zone（可用区），每个 zone 下有多个 rack（机架），每个 rack 下有多个 host（主机）。PD 在调度时首先会尝试将 Region 的 Peer 放置在不同的 zone，假如无法满足（比如配置 3 副本但总共只有 2 个 zone）则保证放置在不同的 rack；假如 rack 的数量也不足以保证隔离，那么再尝试 host 级别的隔离，以此类推。

10.4.2 配置规则操作步骤

本节的操作步骤以使用pd-ctl 工具为例，涉及到的命令也支持通过 HTTP API 进行调用。

10.4.2.1 开启 Placement Rules 特性

默认情况下，Placement Rules 特性是关闭的。要开启这个特性，可以集群初始化以前设置 PD 配置文件：

```
[replication]
enable-placement-rules = true
```

这样，PD 在初始化成功后会开启这个特性，并根据 `max-replicas` 及 `location-labels` 配置生成对应的规则：

```
{
  "group_id": "pd",
  "id": "default",
  "start_key": "",
  "end_key": "",
  "role": "voter",
  "count": 3,
  "location_labels": ["zone", "rack", "host"]
}
```

如果是已经初始化过的集群，也可以通过 pd-ctl 进行在线开启：

```
pd-ctl config placement-rules enable
```

PD 同样将根据系统的 `max-replicas` 及 `location-labels` 生成默认的规则。

> **注意：**
>
> 开启 Placement Rules 后，原先的 `max-replicas` 及 `location-labels` 配置项将不再生效。如果需要调整副本策略，应当使用 Placement Rules 相关接口。

10.4.2.2 关闭 Placement Rules 特性

使用 pd-ctl 可以关闭 Placement Rules 特性，切换为之前的调度策略。

```
pd-ctl config placement-rules disable
```

> **注意：**
>
> 关闭 Placement Rules 后，PD 将使用原先的 `max-replicas` 及 `location-labels` 配置。在 Placement Rules 开启期间对 Rule 的修改不会导致这两项配置的同步更新。此外，设置好的所有 Rule 都会保留在系统中，会在下次开启 Placement Rules 时被使用。

10.4.2.3 使用 pd-ctl 设置规则

> **注意：**
>
> 规则的变更将实时地影响 PD 调度，不恰当的规则设置可能导致副本数较少，影响系统的高可用。

pd-ctl 支持使用多种方式查看系统中的 Rule，输出是 json 格式的 Rule 或 Rule 列表：

- 查看所有规则列表

```
pd-ctl config placement-rules show
```

- 查看 PD Group 的所有规则列表

```
pd-ctl config placement-rules show --group=pd
```

- 查看对应 Group 和 ID 的某条规则

```
pd-ctl config placement-rules show --group=pd --id=default
```

- 查看 Region 所匹配的规则列表

```
pd-ctl config placement-rules show --region=2
```

  上面的例子中 2 为 Region ID。

新增和编辑规则是类似的，需要把对应的规则写进文件，然后使用 `save` 命令保存至 PD：

```
cat > rules.json <<EOF
[
    {
        "group_id": "pd",
        "id": "rule1",
        "role": "voter",
        "count": 3,
        "location_labels": ["zone", "rack", "host"]
    },
    {
        "group_id": "pd",
        "id": "rule2",
        "role": "voter",
        "count": 2,
        "location_labels": ["zone", "rack", "host"]
    }
]
EOF
pd-ctl config placement save --in=rules.json
```

以上操作会将 rule1、rule2 两条规则写入 PD，如果系统中已经存在 GroupID+ID 相同的规则，则会覆盖该规则。

如果需要删除某条规则，只需要将规则的 count 置为 0 即可，对应 GroupID+ID 相同的规则会被删除。以下命令将删除 pd/rule2 这条规则：

```
cat > rules.json <<EOF
[
    {
        "group_id": "pd",
        "id": "rule2"
    }
]
EOF
pd-ctl config placement save --in=rules.json
```

pd-ctl 还支持通过 load 命令将规则直接转存至文件以方便进行修改，只需要将查看命令的 show 改为 load：

```
pd-ctl config placement-rules load
```

以上命令将所有规则转存至 rules.json 文件。

```
pd-ctl config placement-rules load --group=pd --out=rule.txt
```

以上命令将 PD Group 的规则转存至 rule.txt 文件。

10.4.2.4 使用 tidb-ctl 查询表相关的 key range

若需要针对元数据或某个特定的表进行特殊配置，可以通过 tidb-ctl 的 keyrange 命令来查询相关的 key。注意要添加 --encode 返回 PD 中的表示形式。

```
tidb-ctl keyrange --database test --table ttt --encode
```

```
global ranges:
  meta: (6d00000000000000f8, 6e00000000000000f8)
  table: (7400000000000000f8, 7500000000000000f8)
table ttt ranges: (NOTE: key range might be changed after DDL)
  table: (7480000000000000ff2d00000000000000f8, 7480000000000000ff2e00000000000000f8)
  table indexes: (7480000000000000ff2d5f690000000000fa, 7480000000000000ff2d5f720000000000fa)
    index c2: (7480000000000000ff2d5f698000000000ff0000010000000000fa, 7480000000000000
        ↪ ff2d5f698000000000ff0000020000000000fa)
    index c3: (7480000000000000ff2d5f698000000000ff0000020000000000fa, 7480000000000000
        ↪ ff2d5f698000000000ff0000030000000000fa)
    index c4: (7480000000000000ff2d5f698000000000ff0000030000000000fa, 7480000000000000
        ↪ ff2d5f698000000000ff0000040000000000fa)
  table rows: (7480000000000000ff2d5f720000000000fa, 7480000000000000ff2e00000000000000f8)
```

> **注意：**
>
> DDL 等操作会导致 table ID 发生变化，需要同步更新对应的规则。

10.4.3 典型场景示例

本部分介绍 Placement Rules 的使用场景示例。

10.4.3.1 场景一：普通的表使用 3 副本，元数据使用 5 副本提升集群容灾能力

只需要增加一条规则，将 key range 限定在 meta 数据的范围，并把 count 值设为 5。添加规则示例如下：

```
{
  "group_id": "pd",
  "id": "meta",
  "index": 1,
  "override": true,
  "start_key": "6d00000000000000f8",
  "end_key": "6e00000000000000f8",
  "role": "voter",
  "count": 5,
  "location_labels": ["zone", "rack", "host"]
}
```

10.4.3.2 场景二：5 副本按 2-2-1 的比例放置在 3 个数据中心，且第 3 个中心不产生 Leader

创建三条规则，分别设置副本数为 2、2、1，并且在每个规则内通过 `label_constraints` 将副本限定在对应的数据中心内。另外，不需要 leader 的数据中心将 `role` 改为 `follower`。

```
[
    {
        "group_id": "pd",
        "id": "zone1",
        "start_key": "",
        "end_key": "",
        "role": "voter",
        "count": 2,
        "label_constraints": [
            {"key": "zone", "op": "in", "values": ["zone1"]}
        ],
        "location_labels": ["rack", "host"]
    },
    {
        "group_id": "pd",
        "id": "zone2",
        "start_key": "",
        "end_key": "",
        "role": "voter",
        "count": 2,
        "label_constraints": [
            {"key": "zone", "op": "in", "values": ["zone2"]}
        ],
        "location_labels": ["rack", "host"]
    },
    {
        "group_id": "pd",
        "id": "zone3",
        "start_key": "",
        "end_key": "",
        "role": "follower",
        "count": 1,
        "label_constraints": [
            {"key": "zone", "op": "in", "values": ["zone3"]}
        ],
        "location_labels": ["rack", "host"]
    }
]
```

10.4.3.3 场景三：为某张表添加 2 个 TiFlash Learner 副本

为表的 row key 单独添加一条规则，限定数量为 2，并且通过 label_constraints 保证副本产生在 engine=tiflash 的节点。注意这里使用了单独的 group_id，保证这条规则不会与系统中其他来源的规则互相覆盖或产生冲突。

```
{
  "group_id": "tiflash",
  "id": "learner-replica-table-ttt",
  "start_key": "748000000000000ff2d5f720000000000fa",
  "end_key": "748000000000000ff2e00000000000000f8",
  "role": "learner",
  "count": 2,
  "label_constraints": [
    {"key": "engine", "op": "in", "values": ["tiflash"]}
  ],
  "location_labels": ["host"]
}
```

10.4.3.4 场景四：为某张表在有高性能磁盘的北京节点添加 2 个 Follower 副本

这个例子展示了比较复杂的 label_constraints 配置，下面的例子限定了副本放置在 bj1 或 bj2 机房，且磁盘类型不能为 hdd。

```
{
  "group_id": "follower-read",
  "id": "follower-read-table-ttt",
  "start_key": "748000000000000ff2d00000000000000f8",
  "end_key": "748000000000000ff2e00000000000000f8",
  "role": "follower",
  "count": 2,
  "label_constraints": [
    {"key": "zone", "op": "in", "values": ["bj1", "bj2"]},
    {"key": "disk", "op": "notIn", "values": ["hdd"]}
  ],
  "location_labels": ["host"]
}
```

10.5 Load Base Split

Load Base Split 是 TiKV 在 4.0 版本引入的特性，旨在解决 Region 访问分布不均匀造成的热点问题，比如小表的全表扫描。

10.5.1 场景描述

在 TiDB 中，当流量集中在某些节点时很容易形成热点。PD 会尝试通过调度 Hot Region，尽可能让这些 Hot Region 均匀分布在各个节点上，以求获得更好的性能。

但是 PD 的调度的最小粒度是 Region。如果集群的热点数目少于节点数目，或者说存在某几个热点流量远高于其他 Region，对 PD 的热点调度来说，能做到的也只是让热点从一个节点转移到另一个节点，而无法让整个集群承担负载。

这种场景在读请求居多的 workload 中尤为常见。例如对小表的全表扫描和索引查找，或者是对一些字段的频繁访问。

在此之前解决此类问题的办法是手动输入命令去拆分一个或几个热点 Region，但是这样的操作存在以下两个问题：

- 均匀拆分 Region 并不一定是最好的选择，请求可能集中在某几个 Key 上，即使均匀拆分后热点可能仍然集中在其中一个 Region 上，可能需要经过多次均匀拆分才能达到目标。
- 人工介入不够及时和易用。

10.5.2 实现原理

Load Base Split 会基于统计信息自动拆分 Region。通过统计去识别出那些读流量在 10s 内持续超过阈值的 Region，并在合适的位置将这些 Region 拆分。在选择拆分的位置时，会尽可能平衡拆分后两个 Region 的访问量，并尽量避免跨 Region 的访问。

Load Base Split 后的 Region 不会被迅速 Merge。一方面，PD 的 MergeChecker 会跳过 hot Region ，另一方面 PD 也会针对心跳信息中的 QPS去进行判断，避免 Merge 两个 QPS 很高的 Region。

10.5.3 使用方法

目前的 Load Base Split 的控制参数为 `split.qps-threshold`。这个阈值的含义是：如果连续 10s 内，某个 Region 每秒的各类读请求之和超过 `split.qps-threshold`，那么就对此 Region 进行拆分。

目前默认开启 Load Base Split，但配置相对保守，默认为 3000。如果想要关闭这个功能，可以将这个阈值调到足够高即可。

目前有两种办法修改配置：

- 通过 SQL 语句修改，例如：

```
set config tikv split.qps-threshold=3000
```

- 通过 TiKV 修改，例如：

```
curl -X POST "http://ip:status_port/config" -H "accept: application/json" -d '{"split.qps-
    ↪ threshold":"3000"}'
```

同理，目前也有两种办法查看配置：

- 通过 SQL 查看，例如：

```
show config where type='tikv' and name like '%split.qps-threshold%'
```

- 通过 TiKV 查看，例如：

```
curl "http://ip:status_port/config"
```

> 注意：
>
> 从 v4.0.0-rc.2 起可以使用 SQL 语句来修改和查看配置。

10.6 Store Limit

Store Limit 是 PD 在 3.0 版本引入的特性，旨在能够更加细粒度地控制调度的速度，针对不同调度场景进行调优。

10.6.1 实现原理

PD 的调度是以 operator 为单位执行的。一个 operator 可能包含多个调度操作。示例如下：

```
"replace-down-replica {mv peer: store [2] to [3]} (kind:region,replica, region:10(4,5), createAt
    ↪ :2020-05-18 06:40:25.775636418 +0000 UTC m=+2168762.679540369, startAt:2020-05-18
    ↪ 06:40:25.775684648 +0000 UTC m=+2168762.679588599, currentStep:0, steps:[add learner peer
    ↪  20 on store 3, promote learner peer 20 on store 3 to voter, remove peer on store 2])"
```

以上示例中，`replace-down-replica` 这个 operator 具体包含以下操作：

1. 在 `store 3` 上添加一个 learner peer，ID 为 20。
2. 将 `store 3` 上 ID 为 20 的 learner peer 提升为 voter。
3. 删除 `store 2` 上的 peer。

Store Limit 是通过在内存中维护了一个 store ID 到令牌桶的映射，来实现 store 级别的限速。这里不同的操作对应不同的令牌桶，目前仅支持限制添加 learner/peer 和删除 peer 两种操作的速度，即对应于每个 store 存在两种类型的令牌桶。

每次 operator 产生后会检查所包含的操作对应的令牌桶中是否有足够的 token。如果 token 充足才会将该 operator 加入到调度的队列中，同时从令牌桶中拿走对应的 token，否则该 operator 被丢弃。令牌桶会按照固定的速率补充 token，从而实现限速的目的。

Store Limit 与 PD 其他 limit 相关的参数（如 `region-schedule-limit`，`leader-schedule-limit` 等）不同的是，Store Limit 限制的主要是 operator 的消费速度，而其他的 limit 主要是限制 operator 的产生速度。引入 Store Limit 特性之前，调度的限速主要是全局的，所以即使限制了全局的速度，但还是有可能存在调度都集中在部分 store 上面，因而影响集群的性能。而 Store Limit 通过将限速的粒度进一步细化，可以更好的控制调度的行为。

10.6.2 使用方法

Store Limit 相关的参数可以通过 `pd-ctl` 进行设置。

> **注意：**
>
> 4.0.2 版本之后（包括 4.0.2 版本）废弃了 `store-balance-rate` 参数且命令有部分变化。

10.6.2.1 查看当前 store 的 limit 设置

查看当前 store 的 limit 示例如下：

```
store limit                          // 显示所有 store 添加 learner/peer 的速度上限 （
    ↪ 如不设置具体类型，则显示的是添加 learner/peer 的速度）。
store limit region-add               // 显示所有 store 添加 learner/peer 的速度上限。
store limit region-remove            // 显示所有 store 删除 peer 的速度上限。
```

4.0.2 版本之后：

```
store limit                          // 显示所有 store 添加和删除 peer 的速度上限。
store limit add-peer                 // 显示所有 store 添加 peer 的速度上限。
store limit remove-peer              // 显示所有 store 删除 peer 的速度上限。
```

10.6.2.2 设置全部 store 的 limit

设置全部 store 的 limit 示例如下：

```
store limit all 5                    // 设置所有 store 添加 learner/peer 的速度上限为每分钟 5 个（
    ↪ 如不设置具体类型，则默认设置的是添加 learner/peer 的速度）。
store limit all 5 region-add         // 设置所有 store 添加 learner/peer 的速度上限为每分钟 5 个。
store limit all 5 region-remove      // 设置所有 store 删除 peer 的速度上限为每分钟 5 个。
```

4.0.2 版本之后：

```
store limit all 5                    // 设置所有 store 添加和删除 peer 的速度上限为每分钟 5 个。
store limit all 5 add-peer           // 设置所有 store 添加 peer 的速度上限为每分钟 5 个。
store limit all 5 remove-peer        // 设置所有 store 删除 peer 的速度上限为每分钟 5 个。
```

10.6.2.3 设置单个 store 的 limit

设置单个 store 的 limit 示例如下：

```
store limit 1 5                      // 设置 store 1 添加 learner/peer 的速度上限为每分钟 5 个（
    ↪ 如不设置具体类型，则默认设置的是添加 learner/peer 的速度）。
store limit 1 5 region-add           // 设置 store 1 添加 learner/peer 的速度上限为每分钟 5 个。
store limit 1 5 region-remove        // 设置 store 1 删除 peer 的速度上限为每分钟 5 个。
```

4.0.2 版本之后：

```
store limit 1 5                    // 设置 store 1 添加和删除 peer 的速度上限为每分钟 5 个。
store limit 1 5 add-peer           // 设置 store 1 添加 peer 的速度上限为每分钟 5 个。
store limit 1 5 remove-peer        // 设置 store 1 删除 peer 的速度上限为每分钟 5 个。
```

10.6.2.4 持久化 store limit 修改

由于 store limit 是一个内存中的映射关系，所以上述的修改在切换 leader 或者 PD 重启后会被重置。如果同时想要持久化修改，可以同时使用下面的方法进行设置：

```
config set store-balance-rate 20    // 将所有 store 添加 learner/peer 和删除 peer
    ↪ 的速度上限为每分钟 20 个
```

11 TiDB 工具

11.1 TiDB 工具功能概览

本文档从工具的功能出发，介绍部分工具的功能以及它们之间的替代关系。

11.1.1 部署运维工具

TiDB 提供了 TiUP 和 TiDB Operator 两种部署运维工具，满足你在不同系统环境下的部署运维需求。

11.1.1.1 在物理机或虚拟机上部署运维 TiDB - TiUP

TiUP 是在物理机或虚拟机上的 TiDB 包管理器，管理着 TiDB 的众多的组件，如 TiDB、PD、TiKV 等。当你想要运行 TiDB 生态中任何组件时，只需要执行一行 TiUP 命令即可。

TiUP cluster 是 TiUP 提供的使用 Golang 编写的集群管理组件，通过 TiUP cluster 组件就可以进行日常的运维工作，包括部署、启动、关闭、销毁、弹性扩缩容、升级 TiDB 集群，以及管理 TiDB 集群参数。

基本信息：

- 术语及核心概念
- 使用 TiUP 部署 TiDB 集群
- TiUP 组件管理
- 适用 TiDB 版本：v4.0 及以上

11.1.1.2 在 Kubernetes 上部署运维 TiDB - TiDB Operator

TiDB Operator 是 Kubernetes 上的 TiDB 集群自动运维系统，提供包括部署、升级、扩缩容、备份恢复、配置变更的 TiDB 全生命周期管理。借助 TiDB Operator，TiDB 可以无缝运行在公有云或自托管的 Kubernetes 集群上。

基本信息：

- TiDB Operator 架构
- 在 Kubernetes 上部署运维 TiDB 快速上手
- 适用 TiDB 版本：v2.1 及以上

11.1.2 数据管理工具

TiDB 提供了丰富的数据管理工具，例如导入导出、备份恢复、增量同步、数据迁移等。

11.1.2.1 全量导出

Dumpling 是一个用于从 MySQL/TiDB 进行全量逻辑导出的工具。

基本信息：

- Dumpling 的输入：MySQL/TiDB 集群
- Dumpling 的输出：SQL/CSV 文件
- 适用 TiDB 版本：所有版本
- Kubernetes 支持：尚未支持

> 注意：
>
> PingCAP 之前维护的 Mydumper 工具 fork 自 Mydumper project，针对 TiDB 的特性进行了优化。从 v7.5.0 开始，Mydumper 废弃，其绝大部分功能已经被Dumpling 取代，强烈建议切换到 Dumpling。

11.1.2.2 全量导入

TiDB Lightning 是一个用于将全量数据导入到 TiDB 集群的工具。

使用 TiDB Lightning 导入数据到 TiDB 时，有三种模式：

- `local` 模式：TiDB Lightning 将数据解析为有序的键值对，并直接将其导入 TiKV。这种模式一般用于导入大量的数据（TB 级别）到新集群，但在数据导入过程中集群无法提供正常的服务。
- `importer` 模式：和 `local` 模式类似，但是需要部署额外的组件 `tikv-importer` 协助完成键值对的导入。对于 4.0 以上的目标集群，请优先使用 `local` 模式进行导入。
- `tidb` 模式：以 TiDB/MySQL 作为后端，这种模式相比 `local` 和 `importer` 模式的导入速度较慢，但是可以在线导入，同时也支持将数据导入到 MySQL。

基本信息：

- TiDB Lightning 的输入：
  - Dumpling 输出文件
  - 其他格式兼容的 CSV 文件
- 适用 TiDB 版本：v2.1 及以上
- Kubernetes 支持：使用 TiDB Lightning 快速恢复 Kubernetes 上的 TiDB 集群数据

> **注意：**
>
> 原 Loader 工具已停止维护，不再推荐使用。相关场景请使用 TiDB Lightning 的 `tidb` 模式进行替代，详细信息请参考TiDB Lightning TiDB-backend 文档。

11.1.2.3 备份和恢复

BR 是一个对 TiDB 进行分布式备份和恢复的工具，可以高效地对大数据量的 TiDB 集群进行数据备份和恢复。

基本信息：

- 备份输出和恢复输入的文件类型：SST + `backupmeta` 文件
- 适用 TiDB 版本：v3.1 及 v4.0
- Kubernetes 支持：使用 BR 工具备份 TiDB 集群数据到兼容 S3 的存储, 使用 BR 工具恢复 S3 兼容存储上的备份数据

11.1.2.4 TiDB 增量日志同步

TiDB Binlog 是收集 TiDB 的增量 binlog 数据，并提供准实时同步和备份的工具。该工具可用于 TiDB 集群间的增量数据同步，如将其中一个 TiDB 集群作为另一个 TiDB 集群的从集群。

基本信息：

- TiDB Binlog 的输入：TiDB 集群
- TiDB Binlog 的输出：TiDB 集群、 MySQL、 Kafka 或者增量备份文件
- 适用 TiDB 版本：v2.1 及以上
- Kubernetes 支持：TiDB Binlog 运维文档，Kubernetes 上的 TiDB Binlog Drainer 配置

11.1.2.5 数据迁入

TiDB Data Migration (DM) 是将 MySQL/MariaDB 数据迁移到 TiDB 的工具，支持全量数据和增量数据的迁移。

基本信息：

- DM 的输入：MySQL/MariaDB
- DM 的输出：TiDB 集群
- 适用 TiDB 版本：所有版本
- Kubernetes 支持：开发中

如果数据量在 TB 级别以下，推荐直接使用 DM 迁移 MySQL/MariaDB 数据到 TiDB（迁移的过程包括全量数据的导出导入和增量数据的同步）。

如果数据量在 TB 级别，推荐的迁移步骤如下：

1. 使用Dumpling 导出 MySQL/MariaDB 全量数据。
2. 使用TiDB Lightning 将全量导出数据导入 TiDB 集群。

3. 使用 DM 迁移 MySQL/MariaDB 增量数据到 TiDB。

> 注意：
>
> - 原 Syncer 工具已停止维护，不再推荐使用，相关场景请使用 DM 的增量迁移模式进行替代。

11.2 TiDB 工具适用场景

本文档从生态工具的适用场景出发，介绍部分常见场景下的生态工具选择。

11.2.1 在物理机或虚拟机上部署运维 TiDB

当需要在物理机或虚拟机上部署运维 TiDB 时，你可以先安装TiUP，再通过 TiUP 管理 TiDB 的众多组件，如 TiDB、PD、 TiKV 等。

11.2.2 在 Kubernetes 上部署运维 TiDB

当需要在 Kubernetes 上部署运维 TiDB 时，你可以先创建 Kubernetes 集群，部署TiDB Operator，然后使用 TiDB Operator 部署运维 TiDB 集群。

11.2.3 从 CSV 导入数据到 TiDB

当需要将其他工具导出的格式兼容的 CSV files 导入到 TiDB 时，可使用TiDB Lightning。

11.2.4 从 MySQL/Aurora 导入全量数据

当需要从 MySQL/Aurora 导入全量数据时，可先使用Dumpling 将数据导出为 SQL dump files，然后再使用TiDB Lightning 将数据导入到 TiDB 集群。

11.2.5 从 MySQL/Aurora 迁移数据

当既需要从 MySQL/Aurora 导入全量数据，又需要迁移增量数据时，可使用 TiDB Data Migration (DM) 完成全量数据和增量数据的迁移。

如果全量数据量较大（TB 级别），则可先使用Dumpling 与TiDB Lightning 完成全量数据的迁移，再使用 DM 完成增量数据的迁移。

11.2.6 TiDB 集群备份与恢复

当需要对 TiDB 集群进行备份或在之后对 TiDB 集群进行恢复时，可使用BR。

另外，BR 也可以对 TiDB 的数据进行增量备份和增量恢复。

11.2.7 迁出数据到 TiDB

当需要将 TiDB 集群的数据迁出到其他 TiDB 集群时，可使用Dumpling 从 TiDB 将全量数据导出为 SQL dump files，然后再使用TiDB Lightning 将数据导入到 TiDB。

如果还需要执行增量数据的迁移，则可使用TiDB Binlog。

11.2.8 TiDB 增量数据订阅

当需要订阅 TiDB 增量数据的变更时，可使用TiDB Binlog。

11.3 TiDB 工具下载

本页面汇总了 TiDB 工具官方维护版本的下载链接。

11.3.1 TiUP

TiUP 安装过程十分简洁，无论是 Darwin 还是 Linux 操作系统，执行一行命令即可安装成功。详情请参考安装 TiUP。

11.3.2 TiDB Operator

TiDB Operator 运行在 Kubernetes 集群。在搭建好 Kubernetes 集群后，你可以选择在线或者离线部署 TiDB Operator。详情请参考在 Kubernetes 上部署 TiDB Operator。

11.3.3 TiDB Binlog

如需下载最新版本的TiDB Binlog，直接下载 TiDB 安装包即可，因为 TiDB Binlog 包含在 TiDB 安装包中。

安装包	操作系统	架构	SHA256 校验和
https ↪ ://download ↪ .pingcap ↪ .org ↪ /tidb ↪ -{version}- ↪ linux ↪ - ↪ amd64 ↪ . ↪ tar ↪ . ↪ gz (TiDB Binlog)	Linux	amd64	https ↪ ://download ↪ .pingcap ↪ .org ↪ /tidb ↪ -{version}- ↪ linux ↪ - ↪ amd64 ↪ . ↪ sha256 ↪

> **注意：**
>
> 下载链接中的 `{version}` 为 TiDB 的版本号。例如，`v4.0.16` 版本的下载链接为 `https://download.pingcap.org/tidb-v4.0.16-linux-amd64.tar.gz`。

11.3.4 TiDB Lightning

使用下表中的链接下载TiDB Lightning：

安装包	操作系统	架构	SHA256 校验和
https ↪ ://download ↪ .pingcap ↪ .org ↪ /tidb ↪ -toolkit ↪ -{version} ↪ -linux ↪ -amd64 ↪ .tar ↪ .gz	Linux	amd64	https ↪ ://download ↪ .pingcap ↪ .org ↪ /tidb ↪ -toolkit ↪ -{version} ↪ -linux ↪ -amd64 ↪ .sha256 ↪

注意：

下载链接中的 {version} 为 TiDB Lightning 的版本号。例如，v4.0.16 版本的下载链接为 https ↪ ://download.pingcap.org/tidb-toolkit-v4.0.16-linux-amd64.tar.gz。

11.3.5 备份和恢复 (BR) 工具

使用下表中的链接下载BR 工具：

安装包	操作系统	架构	SHA256 校验和
https ↪ ://download ↪ .pingcap ↪ .org ↪ /tidb ↪ -toolkit ↪ -{version} ↪ -linux ↪ -amd64 ↪ .tar ↪ .gz	Linux	amd64	https ↪ ://download ↪ .pingcap ↪ .org ↪ /tidb ↪ -toolkit ↪ -{version} ↪ -linux ↪ -amd64 ↪ .sha256 ↪

> **注意：**
>
> 下载链接中的 {version} 为 BR 的版本号。例如，v4.0.16 版本的下载链接为 https://download ↪ .pingcap.org/tidb-toolkit-v4.0.16-linux-amd64.tar.gz。

11.3.6 TiDB DM (Data Migration)

使用下表中的链接下载 DM：

安装包	操作系统	架构	SHA256 校验和
https ↪ ://download ↪ .pingcap ↪ .org ↪ /dm ↪ -{version}- ↪ linux ↪ - ↪ amd64 ↪ .tar ↪ .gz	Linux	amd64	https ↪ ://download ↪ .pingcap ↪ .org ↪ /dm ↪ -{version}- ↪ linux ↪ - ↪ amd64 ↪ .sha256 ↪

> **注意：**
>
> 下载链接中的 {version} 为 DM 的版本号。例如，v2.0.3 版本的下载链接为 https://download.pingcap.org/dm-v2.0.3-linux-amd64.tar.gz。可以通过 DM Release 查看当前已发布版本。

11.3.7 Dumpling

使用下表中的链接下载Dumpling:

安装包	操作系统	架构	SHA256 校验和
https ↪ ://download ↪ .pingcap ↪ .org ↪ /tidb ↪ -toolkit ↪ -{version}- ↪ linux ↪ -amd64 ↪ .tar ↪ .gz	Linux	amd64	https ↪ ://download ↪ .pingcap ↪ .org ↪ /tidb ↪ -toolkit ↪ -{version}- ↪ linux ↪ -amd64 ↪ .sha256 ↪

注意：

下载链接中的 `{version}` 为 Dumpling 的版本号。例如，`v4.0.16` 版本的下载链接为 `https ↪ ://download.pingcap.org/tidb-toolkit-v4.0.16-linux-amd64.tar.gz`。可以通过 Dumpling Release 查看当前已发布版本。Dumpling 已支持 arm64 linux，将下载链接中的 amd64 替换为 arm64，即表示 arm64 版 Dumpling。

11.3.8 Syncer，Loader 和 Mydumper

如需下载最新版本的Syncer，Loader 或Mydumper，直接下载 tidb-enterprise-tools 安装包即可，因为这些工具均包含在此安装包中。

安装包	操作系统	架构	SHA256 校验和
tidb-enterprise-tools-nightly-linux-amd64.tar.gz	Linux	amd64	tidb-enterprise-tools-nightly-linux-amd64.sha256

tidb-enterprise-tools 安装包包含以下工具：

- Syncer
- Loader
- Mydumper
- ddl_checker
- sync-diff-inspector

11.4 TiUP

11.4.1 TiUP 文档地图

11.4.1.1 使用文档

- TiUP 概览：对 TiUP 进行整体介绍，如何安装和基本的用法以及相关术语
- TiUP 术语：解释使用 TiUP 过程中可能用到的术语，了解 TiUP 的核心概念
- TiUP 组件管理：详细介绍 TiUP 所有命令，如何使用 TiUP 下载、更新、删除组件
- TiUP FAQ：TiUP 使用过程中的常见问题，包含 TiUP 第三方组件的 FAQ
- TiUP 故障排查：如果在使用 TiUP 过程中遇到问题，可以参考故障排查文档的解决方案
- TiUP 参考手册：TiUP 详细参考手册，包含各类命令、组件、镜像。

11.4.1.2 资源

- AskTUG TiUP 主题
- TiUP Issues：TiUP Github Issues 列表

11.4.2 TiUP 简介

在各种系统软件和应用软件的安装管理中，包管理器均有着广泛的应用，包管理工具的出现大大简化了软件的安装和升级维护工作。例如，几乎所有使用 RPM 的 Linux 都会使用 Yum 来进行包管理，而 Anaconda 则可以非常方便地管理 python 的环境和相关软件包。

在早期的 TiDB 生态中，没有专门的包管理工具，使用者只能通过相应的配置文件和文件夹命名来手动管理，如 Prometheus 等第三方监控报表工具甚至需要额外的特殊管理，这样大大提升了运维管理难度。

从 TiDB 4.0 版本开始，TiUP 作为新的工具，承担着包管理器的角色，管理着 TiDB 生态下众多的组件，如 TiDB、PD、TiKV 等。用户想要运行 TiDB 生态中任何组件时，只需要执行 TiUP 一行命令即可，相比以前，极大地降低了管理难度。

11.4.2.1 安装 TiUP

TiUP 安装过程十分简洁，无论是 Darwin 还是 Linux 操作系统，执行一行命令即可安装成功：

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

该命令将 TiUP 安装在 `$HOME/.tiup` 文件夹下，之后安装的组件以及组件运行产生的数据也会放在该文件夹下。同时，它还会自动将 `$HOME/.tiup/bin` 加入到 Shell Profile 文件的 PATH 环境变量中，这样你就可以直接使用 TiUP 了。

例如，你可以查看 TiUP 的版本：

```
tiup --version
```

> **注意：**
>
> TiUP 默认会收集使用情况信息，并将这些信息分享给 PingCAP 用于改善产品。若要了解所收集的信息详情及如何禁用该行为，请参见遥测。

11.4.2.2 TiUP 生态介绍

TiUP 的直接功能是作为 TiDB 生态中的包管理器，但这并不是它的最终使命。TiUP 的愿景是将 TiDB 生态中所有工具的使用门槛降到极致，这个仅仅靠包管理功能是做不到的，还需要引入一些额外的包来丰富这个系统，它们一起加入到 TiUP 生态中，让 TiDB 的世界变得更简单。

TiUP 系列文档的主要内容就是介绍 TiUP 及这些包的功能和使用方式。

在 TiUP 生态中，你可以通过在任何命令后加上 `--help` 的方式来获得帮助信息，比如通过以下命令获取 TiUP 本身的帮助信息:

```
tiup --help
```

```
TiUP is a command-line component management tool that can help to download and install
TiDB platform components to the local system. You can run a specific version of a component via
"tiup <component>[:version]". If no version number is specified, the latest version installed
locally will be used. If the specified component does not have any version installed locally,
the latest stable version will be downloaded from the repository.

Usage:
  tiup [flags] <command> [args...]
  tiup [flags] <component> [args...]
```

```
Available Commands:
  install     Install a specific version of a component
  list        List the available TiDB components or versions
  uninstall   Uninstall components or versions of a component
  update      Update tiup components to the latest version
  status      List the status of instantiated components
  clean       Clean the data of instantiated components
  mirror      Manage a repository mirror for TiUP components
  help        Help about any command or component

Components Manifest:
  use "tiup list" to fetch the latest components manifest

Flags:
  -B, --binary <component>[:version]   Print binary path of a specific version of a component <
      ↪ component>[:version]
                                       and the latest version installed will be selected if no
                                           ↪ version specified
      --binpath string                 Specify the binary path of component instance
  -h, --help                           help for tiup
      --skip-version-check             Skip the strict version check, by default a version must
          ↪ be a valid SemVer string
  -T, --tag string                     Specify a tag for component instance
  -v, --version                        version for tiup

Component instances with the same "tag" will share a data directory ($TIUP_HOME/data/$tag):
  $ tiup --tag mycluster playground

Examples:
  $ tiup playground                    # Quick start
  $ tiup playground nightly            # Start a playground with the latest nightly version
  $ tiup install <component>[:version] # Install a component of specific version
  $ tiup update --all                  # Update all installed components to the latest version
  $ tiup update --nightly              # Update all installed components to the nightly version
  $ tiup update --self                 # Update the "tiup" to the latest version
  $ tiup list                          # Fetch the latest supported components list
  $ tiup status                        # Display all running/terminated instances
  $ tiup clean <name>                  # Clean the data of running/terminated instance (Kill
      ↪ process if it's running)
  $ tiup clean --all                   # Clean the data of all running/terminated instances

Use "tiup [command] --help" for more information about a command.
```

输出的帮助信息较长，你可以只关注两部分：

- 可用的命令
  - install：用于安装组件
  - list：查看可用组件列表
  - uninstall：卸载组件
  - update：更新组件版本
  - status：查看组件运行记录
  - clean：清除组件运行记录
  - mirror：从官方镜像克隆一个私有镜像
  - help：输出帮助信息
- 可用的组件
  - playground：在本机启动集群
  - client：连接本机的集群
  - cluster：部署用于生产环境的集群
  - bench：对数据库进行压力测试

> 注意：
>
> - 可用的组件会持续增加，以 `tiup list` 输出结果为准。
> - 组件的可用版本列表也会持续增加，以 `tiup list <component>` 输出结果为准。

命令和组件的区别在于，命令是 TiUP 自带的，用于进行包管理的操作。而组件是 TiUP 通过包管理操作安装的独立组件包。比如执行 `tiup list` 命令，TiUP 会直接运行自己内部的代码，而执行 `tiup playground` 命令则会先检查本地有没有叫做 playground 的组件包，若没有则先从镜像上下载过来，然后运行这个组件包。

11.4.3 TiUP 术语及核心概念

本文主要说明 TiUP 的重要术语和核心概念。

11.4.3.1 TiUP 组件

TiUP 程序只包含少数几个命令，用来下载、更新、卸载组件。TiUP 通过各种组件来扩展其功能。组件是一个可以运行的程序或脚本，通过 `tiup <component>` 运行组件时，TiUP 会添加一组环境变量，并为该程序创建好对应的数据目录，然后运行该程序。

通过运行 `tiup <component>` 命令，你可以运行支持的 TiUP 组件，其中运行的逻辑为：

1. 如果用户通过 `tiup <component>[:version]` 运行指定某个组件的特定版本：
   - 组件在本地未安装任何版本，则从镜像服务器下载最新稳定版本
   - 组件在本地安装有其他版本，但是没有用户指定的版本，则从镜像服务器下载用户指定版本
   - 如果本地已经安装指定版本，则设置环境变量来运行已经安装的版本
2. 如果用户通过 `tiup <component>` 运行某个组件，且未指定任何版本：

- 组件在本地未安装任何版本，则从镜像服务器下载最新稳定版本
- 如果本地已经安装部分版本，则设置环境变量来运行已经安装的版本中的最新版本

11.4.3.2 TiUP 镜像仓库

TiUP 的所有组件都从镜像仓库 (mirrors) 下载，镜像仓库包含各个组件的 TAR 包以及对应的元信息（版本、入口启动文件、校验和）。TiUP 默认使用 PingCAP 官方的镜像仓库。用户可以通过 `TIUP_MIRRORS` 环境变量自定义镜像仓库。

镜像仓库可以是本地文件目录或在线 HTTP 服务器：

```
1. TIUP_MIRRORS=/path/to/local tiup list
2. TIUP_MIRRORS=https://private-mirrors.example.com tiup list
```

11.4.4 使用 TiUP 命令管理组件

TiUP 主要通过以下一些命令来管理组件：

- list：查询组件列表，用于了解可以安装哪些组件，以及这些组件可选哪些版本
- install：安装某个组件的特定版本
- update：升级某个组件到最新的版本
- uninstall：卸载组件
- status：查看组件运行状态
- clean：清理组件实例
- help：打印帮助信息，后面跟其他 TiUP 命令则是打印该命令的使用方法

本文介绍常用的组件管理操作及相应命令。

11.4.4.1 查询组件列表

你可以使用 `tiup list` 命令来查询组件列表。该命令用法如下：

- `tiup list`：查看当前有哪些组件可以安装
- `tiup list ${component}`：查看某个组件有哪些版本可以安装

你也可以在命令中组合使用以下参数 (flag)：

- `--installed`：查看本地已经安装了哪些组件，或者已经安装了某个组件的哪些版本
- `--all`：显式隐藏的组件
- `--verbose`：显式所有列（安装的版本、支持的平台）

示例一：查看当前已经安装的所有组件

```
tiup list --installed
```

示例二：从服务器获取 TiKV 所有可安装版本组件列表

```
tiup list tikv
```

11.4.4.2 安装组件

你可以使用 `tiup install` 命令来安装组件。该命令的用法如下：

- `tiup install <component>`：安装指定组件的最新稳定版
- `tiup install <component>:[version]`：安装指定组件的指定版本

示例一：使用 TiUP 安装最新稳定版的 TiDB

```
tiup install tidb
```

示例二：使用 TiUP 安装 nightly 版本的 TiDB

```
tiup install tidb:nightly
```

示例三：使用 TiUP 安装 v3.0.6 版本的 TiKV

```
tiup install tikv:v3.0.6
```

11.4.4.3 升级组件

在官方组件提供了新版之后，你可以使用 `tiup update` 命令来升级组件。除了以下几个参数，该命令的用法基本和 `tiup install` 相同：

- `--all`：升级所有组件
- `--nightly`：升级至 nightly 版本
- `--self`：升级 TiUP 自己至最新版本
- `--force`：强制升级至最新版本

示例一：升级所有组件至最新版本

```
tiup update --all
```

示例二：升级所有组件至 nightly 版本

```
tiup update --all --nightly
```

示例三：升级 TiUP 至最新版本

```
tiup update --self
```

11.4.4.4 运行组件

安装完成之后，你可以使用 `tiup <component>` 命令来启动相应的组件：

```
tiup [flags] <component>[:version] [args...]

Flags:
  -T, --tag string                    为组件实例指定 tag
```

该命令需要提供一个组件的名字以及可选的版本，若不提供版本，则使用该组件已安装的最新稳定版。

在组件启动之前，TiUP 会先为它创建一个目录，然后将组件放到该目录中运行。组件会将所有数据生成在该目录中，目录的名字就是该组件运行时指定的 tag 名称。如果不指定 tag，则会随机生成一个 tag 名称，并且在实例终止时自动删除工作目录。

如果想要多次启动同一个组件并复用之前的工作目录，就可以在启动时用 `--tag` 指定相同的名字。指定 tag 后，在实例终止时就不会自动删除工作目录，方便下次启动时复用。

示例一：运行 v3.0.8 版本的 TiDB

```
tiup tidb:v3.0.8
```

示例二：指定 tag 运行 TiKV

```
tiup --tag=experiment tikv
```

11.4.4.4.1 查询组件运行状态

你可以使用 `tiup status` 命令来查看组件的运行状态：

```
tiup status
```

运行该命令会得到一个实例列表，每行一个实例。列表中包含这些列：

- Name：实例的 tag 名称
- Component：实例的组件名称
- PID：实例运行的进程 ID
- Status：实例状态，`RUNNING` 表示正在运行，`TERM` 表示已经终止
- Created Time：实例的启动时间
- Directory：实例的工作目录，可以通过 `--tag` 指定
- Binary：实例的可执行程序，可以通过 `--binpath` 指定
- Args：实例的运行参数

11.4.4.4.2 清理组件实例

你可以使用 `tiup clean` 命令来清理组件实例，并删除工作目录。如果在清理之前实例还在运行，会先 kill 相关进程。该命令用法如下：

```
tiup clean [tag] [flags]
```

支持以下参数：

- `--all`：清除所有的实例信息

其中 tag 表示要清理的实例 tag，如果使用了 `--all` 则不传递 tag。

示例一：清理 tag 名称为 `experiment` 的组件实例

```
tiup clean experiment
```

示例二：清理所有组件实例

```
tiup clean --all
```

11.4.4.4.3 卸载组件

TiUP 安装的组件会占用本地磁盘空间，如果不想保留过多老版本的组件，可以先查看当前安装了哪些版本的组件，然后再卸载某个组件。

你可以使用 `tiup uninstall` 命令来卸载某个组件的所有版本或者特定版本，也支持卸载所有组件。该命令用法如下：

```
tiup uninstall [component][:version] [flags]
```

支持的参数：

- `--all`：卸载所有的组件或版本
- `--self`：卸载 TiUP 自身

component 为要卸载的组件名称，version 为要卸载的版本，这两个都可以省略，省略任何一个都需要加上 `--all` 参数：

- 若省略版本，加 `--all` 表示卸载该组件所有版本
- 若版本和组件都省略，则加 `--all` 表示卸载所有组件及其所有版本

示例一：卸载 v3.0.8 版本的 TiDB

```
tiup uninstall tidb:v3.0.8
```

示例二：卸载所有版本的 TiKV

```
tiup uninstall tikv --all
```

示例三：卸载所有已经安装的组件

```
tiup uninstall --all
```

11.4.5 TiUP FAQ

11.4.5.1 TiUP 是否可以不使用官方镜像源？

TiUP 支持通过环境变量 `TIUP_MIRRORS` 指定镜像源，镜像源的地址可以是一个本地目录或 HTTP 服务器地址。如果用户的环境不能访问网络，可以建立自己的离线镜像源使用 TiUP。

如果在使用非官方镜像之后想要切回官方镜像可以采取以下任一措施：

- 将 `TIUP_MIRRORS` 变量设置成官方镜像的地址：`https://tiup-mirrors.pingcap.com`。
- 先确保 `TIUP_MIRRORS` 变量没有设置，再使用执行 `tiup mirror set https://tiup-mirrors.pingcap.com` 命令。

11.4.5.2 如何将自己编写的组件放入 TiUP 镜像仓库？

TiUP 暂时不支持外部开发的组件，但是 TiUP Team 已经制定了 TiUP 组件开发规范，同时正在开发 tiup-publish 组件，完成 tiup-publish 组件后，开发者可以通过 `tiup publish <comp> <version>` 将自己开发的组件发布到 TiUP 的官方镜像仓库。

11.4.5.3 tiup-playground 和 tiup-cluster 有什么区别？

TiUP Playground 组件主要定位是快速上手和搭建单机的开发环境，支持 Linux/MacOS，要运行一个指定版本的 TiUP 集群更加简单。TiUP Cluster 组件主要是部署生产环境集群，通常是一个大规模的集群，还包含运维相关操作。

11.4.5.4 怎么样编写 tiup-cluster 组件的拓扑文件？

可以参考拓扑文件的样例，样例中包含了：

1. 两地三中心
2. 最小部署拓扑
3. 完整拓扑文件

可以根据自己的需求选择不同的模板，进行编辑。

11.4.5.5 同一个主机是否可以部署多个实例？

同一个主机可以使用 TiUP Cluster 部署多个实例，但是需要配置不同的端口和目录信息，否则可能导致目录以及端口冲突。

11.4.5.6 是否可以检测同一个集群内的端口和目录冲突？

同一个集群的端口和目录冲突会在部署和扩容的时候进行检测，如果有目录和端口冲突，本次部署或扩容会中断。

11.4.5.7 是否可以检测不同集群的端口和目录冲突？

如果不同集群是由同一个 TiUP 中控机部署的，会在部署和扩容时进行检测，如果属于不同的 TiUP 中控机，目前不支持检测。

11.4.5.8 集群部署期间，TiUP 收到报错 `ssh: handshake failed: read tcp 10.10.10.34:38980 -> 10.10.10.34:3600: read: connection reset by peer`

该报错可能是因为 TiUP 默认并发超过 ssh 默认最大连接数导致的，可尝试加大默认连接数，然后重启 sshd 服务解决：

```
vi /etc/ssh/sshd_config
MaxSessions 1000
MaxStartups 1000
```

11.4.6 TiUP 故障排查

本文介绍 TiUP 使用过程中一些常见的故障及排查方式，如果本文不包含你目前遇到的问题，可以通过以下方式求助：

1. Github Issues 新建一个 Issue。
2. 在 AskTUG 提交你的问题。

11.4.6.1 1. TiUP 命令故障排查

11.4.6.1.1 1.1 使用 `tiup list` 看不到最新的组件列表

TiUP 并不会每次都从镜像服务器更新最新的组件列表，可以通过 `tiup list` 来强制刷新组件列表。

11.4.6.1.2 1.2 使用 `tiup list <component>` 看不到一个组件的最新版本信息

同 1.1 一样，组件的版本信息只会在本地无缓存的情况下从镜像服务器获取，可以通过 `tiup list <component>` 刷新组件列表。

11.4.6.1.3 1.3 下载组件的过程中中断

如果下载组件的过程中网络中断，可能是由于网络不稳定导致的，可以尝试重新下载，如果多次不能成功下载，请反馈到 Github Issues，可能是由于 CDN 服务器导致的。

11.4.6.1.4 1.4 下载组件过程中出现 checksum 错误

由于 CDN 会有短暂的缓存时间，导致新的 checksum 文件和组件包不匹配，建议过 5 分钟后重试，如果依然不匹配，请反馈到 Github Issues。

11.4.6.2 2. TiUP Cluster 组件故障排查

11.4.6.2.1 2.1 部署过程中提示 `unable to authenticate, attempted methods [none publickey]`

由于部署时会向远程主机上传组件包，以及进行初始化，这个过程需要连接到远程主机，该错误是由于找不到连接到远程主机的 SSH 私钥导致的。请确认你是否通过 `tiup cluster deploy -i identity_file` 指定该私钥。

1. 如果没有指定 `-i` 参数，可能是由于 TiUP 没有自动找到私钥路径，建议通过 `-i` 显式指定私钥路径。
2. 如果指定了 `-i` 参数，可能是由于指定的私钥不能登录，可以通过手动执行 `ssh -i identity_file user@remote` 命令来验证。
3. 如果是通过密码登录远程主机，请确保指定了 `-p` 参数，同时输入了正确的登录密码。

11.4.6.2.2 2.2 使用 TiUP Cluster 升级中断

为了避免用户误用，TiUP Cluster 不支持指定部分节点升级，所以升级失败之后，需要重新进行升级操作，包括升级过程中的幂等操作。

升级操作会分为以下几步：

1. 首先备份所有节点的老版本组件
2. 分发新的组件到远程
3. 滚动重启所有组件

如果升级操作在滚动重启时中断，可以不用重复进行 `tiup cluster upgrade` 操作，而是通过 `tiup cluster restart -N <node1> -N <node2>` 来重启未完成重启的节点。如果同一组件的未重启节点数量比较多，也可以通过 `tiup cluster restart -R <component>` 来重启某一个类型的组件。

11.4.6.2.3 2.3 升级发现 node_exporter-9100.service/blackbox_exporter-9115.service 不存在

这种情况可能是由于之前的集群是由 TiDB Ansible 迁移过来的，且之前 TiDB Ansible 未部署 exporter 导致的。要解决这种情况，可以暂时通过手动从其他节点复制缺少的文件到新的节点。后续我们会在迁移过程中补全缺失的组件。

11.4.7 TiUP

TiUP 在 TiDB 生态中承担包管理器的功能，管理着 TiDB 生态下众多的组件，如 TiDB、PD、TiKV 等。

11.4.7.1 语法

```
tiup [flags] <command> [args...]        # 执行命令
### or
tiup [flags] <component> [args...]      # 运行组件
```

使用 `help` 命令可以获取特定命令的信息，每个命令的摘要都显示了其参数及其用法。必须参数显示在尖括号中，可选参数显示在方括号中。

`<command>` 代表命令名字，支持的命令列表请参考下方命令清单，`<component>` 代表组件名，支持的组件列表请参考下方组件清单。

11.4.7.2 选项

11.4.7.2.1 -B, –binary

打印指定组件的二进制文件路径：

- 执行 `tiup -B/--binary <component>` 将打印已安装的 `<component>` 组件的最新稳定版路径，若 `<component>` 组件未安装，则报错
- 执行 `tiup -B/--binary <component>:<version>` 将打印已经安装的 `<component>` 组件的 `<version>` 版本所在的路径，若该版本未安装，则报错

- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

> **注意：**
>
> 该选项只能用于 `tiup [flags] <component> [args...]` 格式的命令。

11.4.7.2.2 –binpath (string)

指定要执行的组件的路径：执行一个组件时，如果不想使用 TiUP 镜像中的二进制文件，可以使用该参数使用自定义路径的二进制文件替换之。

> **注意：**
>
> 该选项只能用于 `tiup [flags] <component> [args...]` 格式的命令。

11.4.7.2.3 –skip-version-check

> **注意：**
>
> 该选项自 `v1.3.0` 版本起已废弃。

- 跳过版本号合法性检查，默认指定的版本号只能是 Semantic Version。
- 数据类型：BOOLEAN
- 该选项默认关闭，默认值为 `false`。在命令中添加该选项，并传入 `true` 值或不传值，均可开启此功能。

11.4.7.2.4 -T, –tag (string)

对启动的组件指定一个 `tag`：有的组件在执行过程中需要使用磁盘存储，TiUP 会分配一个临时目录作为该组件本次执行的存储目录，如果希望分配固定目录，可以用 `-T/--tag` 来指定目录名字，这样多次执行使用同样的 `tag` 就能读写到同一批文件。

11.4.7.2.5 -v, –version

打印 TiUP 的版本

11.4.7.2.6 –help

打印帮助信息

11.4.7.3 命令清单

TiUP 包含众多的命令，这些命令又包含了许多子命令，具体命令及其子命令的说明请参考对应的链接：

- install：安装组件
- list：查看组件列表
- uninstall：卸载组件
- update：升级已安装的组件
- status：查看组件运行状态
- clean：清理组件数据目录
- mirror：镜像管理
- telemetry：遥测开关
- completion：TiUP 命令补全
- env：查看 TiUP 相关环境变量
- help：查看特定命令或组件的帮助文档

11.4.7.4 组件清单

- cluster：生产环境 TiDB 集群管理
- dm：生产环境 DM 集群管理

11.4.8 通过 TiUP 部署 TiDB 集群的拓扑文件配置

通过 TiUP 部署或扩容 TiDB 集群时，需要提供一份拓扑文件（示例）来描述集群拓扑。

同样，修改集群配置也是通过编辑拓扑文件来实现的，区别在于修改配置时仅允许修改部分字段。本文档介绍拓扑文件的各个区块以各区块中的各字段。

11.4.8.1 文件结构

一个通过 TiUP 部署的 TiDB 集群拓扑文件可能包含以下区块：

- global：集群全局配置，其中一些是集群的默认值，可以在实例里面单独配置
- monitored：监控服务配置，即 blackbox exporter 和 node exporter，每台机器上都会部署一个 node exporter 和一个 blackbox exporter
- server_configs：组件全局配置，可单独针对每个组件配置，若在实例中存在同名配置项，那么以实例中配置的为准
- pd_servers：PD 实例的配置，用来指定 PD 组件部署到哪些机器上
- tidb_servers：TiDB 实例的配置，用来指定 TiDB 组件部署到哪些机器上
- tikv_servers：TiKV 实例的配置，用来指定 TiKV 组件部署到哪些机器上
- tiflash_servers：TiFlash 实例的配置，用来指定 TiFlash 组件部署到哪些机器上
- pump_servers：Pump 实例的配置，用来指定 Pump 组件部署到哪些机器上
- drainer_servers：Drainer 实例的配置，用来指定 Drainer 组件部署到哪些机器上
- cdc_servers：CDC 实例的配置，用来指定 CDC 组件部署到哪些机器上
- tispark_masters：TiSpark Master 实例的配置，用来指定 TiSpark Master 组件部署到哪台机器上，仅允许部署一个 TiSpark Master 节点

- tispark_workers：TiSpark Worker 实例的配置，用来指定 TiSpark Worker 组件部署到哪些机器上
- monitoring_servers：用来指定 Prometheus 部署在哪机器上，TiUP 支持部署多台 Prometheus 实例，但真实投入使用的只有第一个
- grafana_servers：Grafana 实例的配置，用来指定 Grafana 部署在哪台机器上
- alertmanager_servers：Alertemanager 实例的配置，用来指定 Alertmanager 部署在哪些机器上

11.4.8.1.1 global

global 区块为集群的全局配置，包含以下字段：

- user：以什么用户来启动部署的集群，默认值：“tidb”，如果 <user> 字段指定的用户在目标机器上不存在，会自动尝试创建
- group：自动创建用户时指定用户所属的用户组，默认和 <user> 字段值相同，若指定的组不存在，则自动创建
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，默认值：22
- enable_tls：是否对集群启用 TLS，启用之后组件之间、客户端与组件之间都必须使用生成的 TLS 证书链接，启用后无法关闭，默认值：false
- deploy_dir：每个组件的部署目录，默认值：“deploy”。其应用规则如下：
  - 如果在实例级别配置了绝对路径的 deploy_dir，那么实际部署目录为该实例设定的 deploy_dir
  - 对于每个实例，如果用户未配置 deploy_dir，其默认值为相对路径 <component-name>-<component-port>
  - 如果 global.deploy_dir 为绝对路径，那么组件会部署到 <global.deploy_dir>/<instance.deploy_dir> 目录
  - 如果 global.deploy_dir 为相对路径，那么组件会部署到 /home/<global.user>/<global.deploy_dir>/<instance.deploy_dir> 目录
- data_dir：数据目录，默认值：“data”。其应用规则如下：
  - 如果在实例级别配置了绝对路径的 data_dir，那么实际数据目录为该实例设定的 data_dir
  - 对于每个实例，如果用户未配置 data_dir，其默认值为 <global.data_dir>
  - 如果 data_dir 为相对路径，那么组件数据将放到 <deploy_dir>/<data_dir> 中，其中 <deploy_dir> 的计算规则请参考 deploy_dir 字段的应用规则
- log_dir：数据目录，默认值：“log”。其应用规则如下：
  - 如果在实例级别配置了绝对路径的 log_dir，那么实际日志目录为该实例设定的 log_dir
  - 对于每个实例，如果用户未配置 log_dir，其默认值为 <global.log_dir>
  - 如果 log_dir 为相对路径，那么组件日志将放到 <deploy_dir>/<log_dir> 中，其中 <deploy_dir> 的计算规则请参考 deploy_dir 字段的应用规则
- os：目标机器的操作系统，该字段决定了向目标机器推送适配哪个操作系统的组件，默认值：linux
- arch：目标机器的 CPU 架构，该字段决定了向目标机器推送哪个平台的二进制包，支持 amd64 和 arm64，默认值：amd64
- resource_control：运行时资源控制，该字段下所有配置都将写入 systemd 的 service 文件中，默认无限制。支持控制的资源如下：
  - memory_limit: 限制运行时最大内存，例如“2G”表示最多使用 2GB 内存
  - cpu_quota：限制运行时最大 CPU 占用率，例如“200%”

- io_read_bandwidth_max：读磁盘 I/O 的最大带宽，例如：“/dev/disk/by-path/pci-0000:00:1f.2-scsi-0:0:0:0 100M”
- io_write_bandwidth_max：写磁盘 I/O 的最大带宽，例如：“/dev/disk/by-path/pci-0000:00:1f.2-scsi-0:0:0:0 100M”
- limit_core：控制 core dump 的大小

global 配置示例：

```
global:
  user: "tidb"
  resource_control:
    memory_limit: "2G"
```

上述配置指定使用 tidb 用户启动集群，同时限制每个组件运行时最多只能使用 2GB 内存。

11.4.8.1.2 monitored

monitored 用于配置目标机上的监控服务：node_exporter 和 blackbox_exporter。包含以下字段：

- node_exporter_port：node_exporter 的服务端口，默认值：9100
- blackbox_exporter_port：blackbox_exporter 的服务端口，默认值：9115
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- data_dir：指定数据目录，若不指定，或指定为相对目录，则按照 global 中配置的 data_dir 生成
- log_dir：指定日志目录，若不指定，或指定为相对目录，则按照 global 中配置的 log_dir 生成

monitored 配置示例：

```
monitored:
  node_exporter_port: 9100
  blackbox_exporter_port: 9115
```

上述配置指定了 node_exporter 使用 9100 端口，blackbox_exporter 使用 9115 端口。

11.4.8.1.3 server_configs

server_configs 用于配置服务，生成各组件的配置文件，类似 global 区块，该区块内的配置可以在具体的实例中被覆盖。主要包含以下字段：

- tidb：TiDB 服务的相关配置，支持的完整配置请参考TiDB 配置文件描述
- tikv：TiKV 服务的相关配置，支持的完整配置请参考TiKV 配置文件描述
- pd：PD 服务的相关配置，支持的完整配置请参考PD 配置文件描述
- tiflash：TiFlash 服务的相关配置，支持的完整配置请参考TiFlash 配置参数
- tiflash_learner：每个 TiFlash 中内置了一个特殊的 TiKV，该配置项用于配置这个特殊的 TiKV，一般不建议修改这个配置项下的内容
- pump：Pump 服务的相关配置，支持的完整配置请参考TiDB Binlog 配置说明
- drainer：Drainer 服务的相关配置，支持的完整配置请参考TiDB Binlog 配置说明

- cdc：CDC 服务的相关配置，支持的完整配置请参考TiCDC 安装部署

server_configs 配置示例：

```
server_configs:
  tidb:
    run-ddl: true
    lease: "45s"
    split-table: true
    token-limit: 1000
  tikv:
    log-level: "info"
    readpool.unified.min-thread-count: 1
```

上述配置指定了 TiDB 和 TiKV 的全局配置。

11.4.8.1.4 pd_servers

pd_servers 指定了将 PD 的服务部署到哪些机器上，同时可以指定每台机器上的服务配置。pd_servers 是一个数组，每个数组的元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- listen_host：当机器上有多个 IP 时，可以指定服务的监听 IP，默认为 0.0.0.0
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- name：指定该 PD 实例的名字，不同实例的名字必须唯一，否则无法部署
- client_port：指定 PD 的客户端连接端口，默认值：2379
- peer_port：指定 PD 之间互相通信的端口，默认值：2380
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- data_dir：指定数据目录，若不指定，或指定为相对目录，则按照 global 中配置的 data_dir 生成
- log_dir：指定日志目录，若不指定，或指定为相对目录，则按照 global 中配置的 log_dir 生成
- numa_node：为该实例分配 NUMA 策略，如果指定了该参数，需要确保目标机装了 numactl。在指定该参数的情况下会通过 numactl 分配 cpubind 和 membind 策略。该字段参数为 string 类型，字段值填 NUMA 节点的 ID，例如 “0,1”
- config：该字段配置规则和 server_configs 里的 pd 配置规则相同，若配置了该字段，会将该字段内容和 server_configs 里的 pd 内容合并（若字段重叠，以本字段内容为准），然后生成配置文件并下发到 host 指定的机器
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- resource_control：针对该服务的资源控制。如果配置了该字段，会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global 中的 resource_control

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host

- listen_host
- name
- client_port
- peer_port
- deploy_dir
- data_dir
- log_dir
- arch
- os

pd_servers 配置示例：

```
pd_servers:
  - host: 10.0.1.11
    config:
      schedule.max-merge-region-size: 20
      schedule.max-merge-region-keys: 200000
  - host: 10.0.1.12
```

上述配置指定了将 PD 部署到 10.0.1.11 和 10.0.1.12，并针对 10.0.1.11 的 PD 进行一些特殊配置。

11.4.8.1.5 tidb_servers

tidb_servers 指定了将 TiDB 服务部署到哪些机器上，同时可以指定每台机器上的服务配置。tidb_servers 是一个数组，每个数组的元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- listen_host：当机器上有多个 IP 时，可以指定服务的监听 IP，默认为 0.0.0.0
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- port：TiDB 服务的监听端口，用于提供给 MySQL 客户端连接，默认值：4000
- status_port：TiDB 状态服务的监听端口，用于外部通过 http 请求查看 TiDB 服务的状态，默认值：10080
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- log_dir：指定日志目录，若不指定，或指定为相对目录，则按照 global 中配置的 log_dir 生成
- numa_node：为该实例分配 NUMA 策略，如果指定了该参数，需要确保目标机装了 numactl，在指定该参数的情况下会通过 numactl 分配 cpubind 和 membind 策略。该字段参数为 string 类型，字段值填 NUMA 节点的 ID，例如 “0,1”
- config：该字段配置规则和 server_configs 里的 tidb 配置规则相同，若配置了该字段，会将该字段内容和 server_configs 里的 tidb 内容合并（若字段重叠，以本字段内容为准），然后生成配置文件并下发到 host 指定的机器
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- resource_control：针对该服务的资源控制，如果配置了该字段，会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global 中的 resource_control

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- listen_host
- port
- status_port
- deploy_dir
- log_dir
- arch
- os

tidb_servers 配置示例：

```
tidb_servers:
  - host: 10.0.1.14
    config:
      log.level: warn
      log.slow-query-file: tidb-slow-overwrited.log
  - host: 10.0.1.15
```

11.4.8.1.6 tikv_servers

tikv_servers 约定了将 TiKV 服务部署到哪些机器上，同时可以指定每台机器上的服务配置。tikv_servers 是一个数组，每个数组元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- listen_host：当机器上有多个 IP 时，可以指定服务的监听 IP，默认为 0.0.0.0
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- port：TiKV 服务的监听端口，默认值：20160
- status_port：TiKV 状态服务的监听端口，默认值：20180
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- data_dir：指定数据目录，若不指定，或指定为相对目录，则按照 global 中配置的 data_dir 生成
- log_dir：指定日志目录，若不指定，或指定为相对目录，则按照 global 中配置的 log_dir 生成
- numa_node：为该实例分配 NUMA 策略，如果指定了该参数，需要确保目标机装了 numactl，在指定该参数的情况下会通过 numactl 分配 cpubind 和 membind 策略。该字段参数为 string 类型，字段值填 NUMA 节点的 ID，例如 “0,1”
- config：该字段配置规则和 server_configs 里的 tikv 配置规则相同，若配置了该字段，会将该字段内容和 server_configs 里的 tikv 内容合并（若字段重叠，以本字段内容为准），然后生成配置文件并下发到 host 指定的机器
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- resource_control：针对该服务的资源控制，如果配置了该字段，会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global 中的 resource_control

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- listen_host
- port
- status_port
- deploy_dir
- data_dir
- log_dir
- arch
- os

tikv_servers 配置示例：

```
tikv_servers:
  - host: 10.0.1.14
    config:
      server.labels: { zone: "zone1", host: "host1" }
  - host: 10.0.1.15
    config:
      server.labels: { zone: "zone1", host: "host2" }
```

11.4.8.1.7 tiflash_servers

tiflash_servers 约定了将 TiFlash 服务部署到哪些机器上，同时可以指定每台机器上的服务配置。该区块是一个数组，每个数组元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- tcp_port：TiFlash TCP 服务的端口，默认 9000
- http_port：TiFlash HTTP 服务的端口，默认 8123
- flash_service_port：TiFlash 提供服务的端口，TiDB 通过该端口从 TiFlash 读数据，默认 3930
- metrics_port：TiFlash 的状态端口，用于输出 metric 数据，默认 8234
- flash_proxy_port：内置 TiKV 的端口，默认 20170
- flash_proxy_status_port：内置 TiKV 的状态端口，默认为 20292
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- data_dir：指定数据目录，若不指定，或指定为相对目录，则按照 global 中配置的 data_dir 生成，TiFlash 的数据目录支持多个，采用逗号分割
- log_dir：指定日志目录，若不指定，或指定为相对目录，则按照 global 中配置的 log_dir 生成
- tmp_path: TiFlash 临时文件的存放路径，默认使用 [path 或者 storage.latest.dir 的第一个目录] + “/tmp”
- numa_node：为该实例分配 NUMA 策略，如果指定了该参数，需要确保目标机装了 numactl，在指定该参数的情况下会通过 numactl 分配 cpubind 和 membind 策略。该字段参数为 string 类型，字段值填 NUMA 节点的 ID，例如 “0,1”

- config：该字段配置规则和 server_configs 里的 tiflash 配置规则相同，若配置了该字段，会将该字段内容和 server_configs 里的 tiflash 内容合并（若字段重叠，以本字段内容为准），然后生成配置文件并下发到 host 指定的机器
- learner_config：每个 TiFlash 中内置了一个特殊的 TiKV 模块，该配置项用于配置这个特殊的 TiKV 模块，一般不建议修改这个配置项下的内容
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- resource_control：针对该服务的资源控制，如果配置了该字段，会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global 中的 resource_control

以上字段中，data_dir 在部署完成之后只能新增目录，而下列字段部署完成之后不能再修改：

- host
- tcp_port
- http_port
- flash_service_port
- flash_proxy_port
- flash_proxy_status_port
- metrics_port
- deploy_dir
- log_dir
- tmp_path
- arch
- os

tiflash_servers 配置示例：

```
tiflash_servers:
  - host: 10.0.1.21
  - host: 10.0.1.22
```

11.4.8.1.8 pump_servers

pump_servers 约定了将 TiDB Binlog 组件的 Pump 服务部署到哪些机器上，同时可以指定每台机器上的服务配置。pump_servers 是一个数组，每个数组元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- port：Pump 服务的监听端口，默认 8250
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- data_dir：指定数据目录，若不指定，或指定为相对目录，则按照 global 中配置的 data_dir 生成
- log_dir：指定日志目录，若不指定，或指定为相对目录，则按照 global 中配置的 log_dir 生成

- numa_node：为该实例分配 NUMA 策略，如果指定了该参数，需要确保目标机装了 numactl，在指定该参数的情况下会通过 numactl 分配 cpubind 和 membind 策略。该字段参数为 string 类型，字段值填 NUMA 节点的 ID，例如 “0,1”
- config：该字段配置规则和 server_configs 里的 pump 配置规则相同，若配置了该字段，会将该字段内容和 server_configs 里的 pump 内容合并（若字段重叠，以本字段内容为准），然后生成配置文件并下发到 host 指定的机器
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- resource_control：针对该服务的资源控制，如果配置了该字段，会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global 中的 resource_control

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- port
- deploy_dir
- data_dir
- log_dir
- arch
- os

pump_servers 配置示例：

```
pump_servers:
  - host: 10.0.1.21
    config:
      gc: 7
  - host: 10.0.1.22
```

11.4.8.1.9 drainer_servers

drainer_servers 约定了将 TiDB Binlog 组件的 Drainer 服务部署到哪些机器上，同时可以指定每台机器上的服务配置，drainer_servers 是一个数组，每个数组元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- port：Drainer 服务的监听端口，默认 8249
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- data_dir：指定数据目录，若不指定，或指定为相对目录，则按照 global 中配置的 data_dir 生成
- log_dir：指定日志目录，若不指定，或指定为相对目录，则按照 global 中配置的 log_dir 生成
- commit_ts：Drainer 启动的时候会去读取 checkpoint，如果读取不到，就会使用该字段做为初次启动开始的同步时间点，该字段默认为 -1（从 PD 总获取最新时间戳作为 commit_ts）

- numa_node：为该实例分配 NUMA 策略，如果指定了该参数，需要确保目标机装了 numactl，在指定该参数的情况下会通过 numactl 分配 cpubind 和 membind 策略。该字段参数为 string 类型，字段值填 NUMA 节点的 ID，例如 “0,1”
- config：该字段配置规则和 server_configs 里的 drainer 配置规则相同，若配置了该字段，会将该字段内容和 server_configs 里的 drainer 内容合并（若字段重叠，以本字段内容为准），然后生成配置文件并下发到 host 指定的机器
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- resource_control：针对该服务的资源控制，如果配置了该字段，会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global 中的 resource_control

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- port
- deploy_dir
- data_dir
- log_dir
- commit_ts
- arch
- os

drainer_servers 配置示例：

```
drainer_servers:
  - host: 10.0.1.21
    config:
      syncer.db-type: "mysql"
      syncer.to.host: "127.0.0.1"
      syncer.to.user: "root"
      syncer.to.password: ""
      syncer.to.port: 3306
      syncer.ignore-table:
        - db-name: test
          tbl-name: log
        - db-name: test
          tbl-name: audit
```

11.4.8.1.10 cdc_servers

cdc_servers 约定了将 TiCDC 服务部署到哪些机器上，同时可以指定每台机器上的服务配置，cdc_servers 是一个数组，每个数组元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略

- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- port：TiCDC 服务的监听端口，默认 8300
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- data_dir：指定数据目录。若不指定该字段或指定为相对目录，数据目录则按照 global 中配置的 data_dir 生成。该字段自集群版本 v4.0.14 起生效。在早于该版本的集群中，该字段不生效。
- log_dir：指定日志目录，若不指定，或指定为相对目录，则按照 global 中配置的 log_dir 生成
- gc-ttl：TiCDC 在 PD 设置的服务级别 GC safepoint 的 TTL (Time To Live) 时长，单位为秒，默认值为 86400，即 24 小时
- tz：TiCDC 服务使用的时区。TiCDC 在内部转换 timestamp 等时间数据类型和向下游同步数据时使用该时区，默认为进程运行本地时区。
- numa_node：为该实例分配 NUMA 策略，如果指定了该参数，需要确保目标机装了 numactl，在指定该参数的情况下会通过 numactl 分配 cpubind 和 membind 策略。该字段参数为 string 类型，字段值填 NUMA 节点的 ID，例如 “0,1”
- config：该字段配置规则和 server_configs 里的 cdc 内容合并（若字段重叠，以本字段内容为准），然后生成配置文件并下发到 host 指定的机器
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- resource_control：针对该服务的资源控制，如果配置了该字段，会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global 中的 resource_control

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- port
- deploy_dir
- data_dir
- log_dir
- arch
- os

cdc_servers 配置示例：

```
cdc_servers:
  - host: 10.0.1.20
    gc-ttl: 86400
    data_dir: "/cdc-data"
  - host: 10.0.1.21
    gc-ttl: 86400
    data_dir: "/cdc-data"
```

11.4.8.1.11 tispark_masters

tispark_masters 约定了将 TiSpark 的 master 节点部署到哪些机器上，同时可以指定每台机器上的服务配置，tispark_masters 是一个数组，每个数组元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- listen_host：当机器上有多个 IP 时，可以指定服务的监听 IP，默认为 0.0.0.0
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- port：Spark 的监听端口，节点之前通讯用，默认 7077
- web_port：Spark 的网页端口，提供网页服务，可查看任务情况，默认 8080
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- java_home：指定要使用的 JRE 环境所在的路径。该参数对应系统环境变量 JAVA_HOME
- spark_config：用于配置 TiSpark 服务，生成配置文件并下发到 host 指定的机器
- spark_env：配置 Spark 启动时的环境变量
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- listen_host
- port
- web_port
- deploy_dir
- arch
- os

tispark_masters 配置示例：

```
tispark_masters:
  - host: 10.0.1.21
    spark_config:
      spark.driver.memory: "2g"
      spark.eventLog.enabled: "False"
      spark.tispark.grpc.framesize: 2147483647
      spark.tispark.grpc.timeout_in_sec: 100
      spark.tispark.meta.reload_period_in_sec: 60
      spark.tispark.request.command.priority: "Low"
      spark.tispark.table.scan_concurrency: 256
    spark_env:
      SPARK_EXECUTOR_CORES: 5
      SPARK_EXECUTOR_MEMORY: "10g"
      SPARK_WORKER_CORES: 5
      SPARK_WORKER_MEMORY: "10g"
  - host: 10.0.1.22
```

11.4.8.1.12 tispark_workers

tispark_workers 约定了将 TiSpark 的 worker 节点部署到哪些机器上，同时可以指定每台机器上的服务配置，tispark_workers 是一个数组，每个数组元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- listen_host：当机器上有多个 IP 时，可以指定服务的监听 IP，默认为 0.0.0.0
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- port：Spark 的监听端口，节点之前通讯用，默认 7077
- web_port：Spark 的网页端口，提供网页服务，可查看任务情况，默认 8080
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- java_home：指定要使用的 JRE 环境所在的路径。该参数对应系统环境变量 JAVA_HOME
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- listen_host
- port
- web_port
- deploy_dir
- arch
- os

tispark_workers 配置示例：

```
tispark_workers:
  - host: 10.0.1.22
  - host: 10.0.1.23
```

11.4.8.1.13 monitoring_servers

monitoring_servers 约定了将 Prometheus 服务部署到哪台机器上，同时可以指定这台机器上的服务配置，monitoring_servers 是一个数组，每个数组元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- port：指定 Prometheus 提供服务的端口，默认值：9090
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- data_dir：指定数据目录，若不指定，或指定为相对目录，则按照 global 中配置的 data_dir 生成
- log_dir：指定日志目录，若不指定，或指定为相对目录，则按照 global 中配置的 log_dir 生成
- numa_node：为该实例分配 NUMA 策略，如果指定了该参数，需要确保目标机装了 numactl，在指定该参数的情况下会通过 numactl 分配 cpubind 和 membind 策略。该字段参数为 string 类型，字段值填 NUMA 节点的 ID，例如 “0,1”
- storage_retention：Prometheus 监控数据保留时间，默认 “30d”
- rule_dir：该字段指定一个本地目录，该目录中应当含有完整的 *.rules.yml 文件，这些文件会在集群配置初始化阶段被传输到目标机器上，作为 Prometheus 的规则

- remote_config：用于支持将 Prometheus 数据写到远端，或从远端读取数据，该字段下有两个配置：
  - remote_write：参考 Prometheus <remote_write> 文档
  - remote_read：参考 Prometheus <remote_read> 文档
- external_alertmanagers：若配置了 external_alertmanagers，Prometheus 会将配置行为报警通知到集群外的 Alertmanager。该字段为一个数组，数组的元素为每个外部的 Alertmanager，由 host 和 web_port 字段构成
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- resource_control：针对该服务的资源控制，如果配置了该字段，会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global 中的 resource_control

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- port
- deploy_dir
- data_dir
- log_dir
- arch
- os

monitoring_servers 配置示例：

```
monitoring_servers:
  - host: 10.0.1.11
    rule_dir: /local/rule/dir
    remote_config:
      remote_write:
      - queue_config:
          batch_send_deadline: 5m
          capacity: 100000
          max_samples_per_send: 10000
          max_shards: 300
        url: http://127.0.0.1:8003/write
      remote_read:
      - url: http://127.0.0.1:8003/read
    external_alertmanagers:
    - host: 10.1.1.1
      web_port: 9093
    - host: 10.1.1.2
      web_port: 9094
```

11.4.8.1.14 grafana_servers

grafana_servers 约定了将 Grafana 服务部署到哪台机器上，同时可以指定这台机器上的服务配置，grafana_servers 是一个数组，每个数组元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- port：指定 Grafana 提供服务的端口，默认值：3000
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- username：Grafana 登录界面的用户名
- password：Grafana 对应的密码
- dashboard_dir：该字段指定一个本地目录，该目录中应当含有完整的 dashboard(*.json) 文件，这些文件会在集群配置初始化阶段被传输到目标机器上，作为 Grafana 的 dashboards
- resource_control：针对该服务的资源控制，如果配置了该字段，会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global 中的 resource_control

> 注意：
>
> 如果配置了 grafana_servers 的 dashboard_dir 字段，在执行 tiup cluster rename 命令进行集群重命名后，需要完成以下操作：
>
> 1. 对于本地的 dashboards 目录中的 *.json 文件，将 datasource 字段的值更新为新的集群名（这是因为 datasource 是以集群名命名的）
> 2. 执行 tiup cluster reload -R grafana 命令

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- port
- deploy_dir
- arch
- os

grafana_servers 配置示例：

```
grafana_servers:
  - host: 10.0.1.11
    dashboard_dir: /local/dashboard/dir
```

11.4.8.1.15 alertmanager_servers

alertmanager_servers 约定了将 Alertmanager 服务部署到哪些机器上，同时可以指定这台机器上的服务配置，alertmanager_servers 是一个数组，每个数组元素包含以下字段：

- host：指定部署到哪台机器，字段值填 IP 地址，不可省略
- ssh_port：指定连接目标机器进行操作的时候使用的 SSH 端口，若不指定，则使用 global 区块中的 ssh_port
- web_port：指定 Alertmanager 提供网页服务的端口，默认值：9093
- cluster_port：指定 Alertmanger 和其他 Alertmanager 通讯的端口，默认值：9094
- deploy_dir：指定部署目录，若不指定，或指定为相对目录，则按照 global 中配置的 deploy_dir 生成
- data_dir：指定数据目录，若不指定，或指定为相对目录，则按照 global 中配置的 data_dir 生成
- log_dir：指定日志目录，若不指定，或指定为相对目录，则按照 global 中配置的 log_dir 生成
- numa_node：为该实例分配 NUMA 策略，如果指定了该参数，需要确保目标机装了 numactl，在指定该参数的情况下会通过 numactl 分配 cpubind 和 membind 策略。该字段参数为 string 类型，字段值填 NUMA 节点的 ID，例如 “0,1”
- config_file：该字段指定一个本地文件，该文件会在集群配置初始化阶段被传输到目标机器上，作为 Alertmanager 的配置
- os：host 字段所指定的机器的操作系统，若不指定该字段，则默认为 global 中的 os
- arch：host 字段所指定的机器的架构，若不指定该字段，则默认为 global 中的 arch
- resource_control：针对该服务的资源控制，如果配置了该字段，会将该字段和 global 中的 resource_control 内容合并（若字段重叠，以本字段内容为准），然后生成 systemd 配置文件并下发到 host 指定机器。resource_control 的配置规则同 global 中的 resource_control

以上所有字段中，部分字段部署完成之后不能再修改。如下所示：

- host
- web_port
- cluster_port
- deploy_dir
- data_dir
- log_dir
- arch
- os

alertmanager_servers 配置示例：

```
alertmanager_servers:
  - host: 10.0.1.11
    config_file: /local/config/file
  - host: 10.0.1.12
    config_file: /local/config/file
```

11.4.9 TiUP 镜像参考指南

TiUP 镜像是 TiUP 的组件仓库，存放了一系列的组件和这些组件的元信息。镜像有两种存在形式：

- 本地磁盘上的目录：用于服务本地的 TiUP 客户端，文档中将称之为本地镜像
- 基于远程的磁盘目录启动的 HTTP 镜像：服务远程的 TiUP 客户端，文档中将称之为远程镜像

11.4.9.1 镜像的创建与更新

镜像可以通过以下两种方式创建：

- 通过命令 `tiup mirror init` 从零生成
- 通过命令 `tiup mirror clone` 从已有镜像克隆

在创建镜像之后，可以通过 `tiup mirror` 相关命令来给镜像添加组件或删除组件，无论是通过何种方式更新镜像，TiUP 都不会从镜像中删除任何文件，而是通过增加文件并分配新版本号的方式更新。

11.4.9.2 镜像结构

一个典型的镜像目录结构如下：

```
+ <mirror-dir>                                  # 镜像根目录
|-- root.json                                   # 镜像根证书
|-- {2..N}.root.json                            # 镜像根证书
|-- {1..N}.index.json                           # 组件/用户索引
|-- {1..N}.{component}.json                     # 组件元信息
|-- {component}-{version}-{os}-{arch}.tar.gz    # 组件二进制包
|-- snapshot.json                               # 镜像最新快照
|-- timestamp.json                              # 镜像最新时间戳
|--+ commits                                    # 镜像更新日志（可删除）
   |--+ commit-{ts1..tsN}
      |-- {N}.root.json
      |-- {N}.{component}.json
      |-- {N}.index.json
      |-- {component}-{version}-{os}-{arch}.tar.gz
      |-- snapshot.json
      |-- timestamp.json
|--+ keys                                       # 镜像私钥（可移动到其他位置）
   |-- {hash1..hashN}-root.json                 # 根证书私钥
   |-- {hash}-index.json                        # 索引私钥
   |-- {hash}-snapshot.json                     # 快照私钥
   |-- {hash}-timestamp.json                    # 时间戳私钥
```

注意：

- commits 目录是在更新镜像过程中产生的日志，用于回滚镜像，磁盘空间不足时可以定期删除旧的文件夹
- keys 文件夹中存放的私钥较敏感，建议单独妥善保管

11.4.9.2.1 根证书

在 TiUP 镜像中，根证书用于存放其他元信息文件的公钥，每次获取到任何元信息文件（*.json）都需要根据其文件类型（root，index，snapshot，timestamp）在当前已安装的 root.json 中找到对应的公钥，然后用公钥验证其签名是否合法。

根证书文件格式如下：

```
{
    "signatures": [                                         # 每个元信息文件有一系列的签名，
        ↪ 签名由该文件对应的几个私钥签出
        {
            "keyid": "{id-of-root-key-1}",                  # 第一个参与签名私钥的 ID，该 ID
                ↪ 由私钥对应的公钥内容哈希得到
            "sig": "{signature-by-root-key-1}"              # 该私钥对此文件 signed
                ↪ 部分签名的结果
        },
        ...
        {
            "keyid": "{id-of-root-key-N}",                  # 第 N 个参与签名私钥的 ID
            "sig": "{signature-by-root-key-N}"              # 该私钥对此文件 signed
                ↪ 部分签名的结果
        }
    ],
    "signed": {                                             # 被签名的部分
        "_type": "root",                                    # 该字段说明本文件的类型，root.
            ↪ json 的类型就是 root
        "expires": "{expiration-date-of-this-file}",        # 该文件的过期时间，
            ↪ 过期后客户端会拒绝此文件
        "roles": {                                          # root.json 中的 roles
            ↪ 用来记录对各个元文件签名的密钥
            "{role:index,root,snapshot,timestamp}": {       # 涉及的元文件类型包括 index,
                ↪ root, snapshot, timestamp
                "keys": {                                   # 只有 keys
                    ↪ 中记录的密钥签名才是合法的
                    "{id-of-the-key-1}": {                  # 用于签名 {role} 的第 1 个密钥
                        ↪ ID
                        "keytype": "rsa",                   # 密钥类型，目前固定为 rsa
                        "keyval": {                         # 密钥的 payload
                            "public": "{public-key-content}"    # 表示公钥内容
                        },
                        "scheme": "rsassa-pss-sha256"       # 目前固定为 rsassa-pss-sha256
                    },
                    "{id-of-the-key-N}": {                  # 用于签名 {role} 的第 N 个密钥
                        ↪ ID
                        "keytype": "rsa",
                        "keyval": {
```

```
                        "public": "{public-key-content}"
                    },
                    "scheme": "rsassa-pss-sha256"
                }
            },
            "threshold": {N},                                   # threshold 指示该元文件需要至少
                ↪ N 个密钥签名
            "url": "/{role}.json"                               # url 是指该文件的获取地址，对于
                ↪ index 文件，需要在前面加上版本号，即 /{N}.index.json
        }
    },
    "spec_version": "0.1.0",                                    # 本文件遵循的规范版本，
        ↪ 未来变更文件结构需要升级版本号，目前为 0.1.0
    "version": {N}                                              # 本文件的版本号，
        ↪ 每次更新文件需要创建一个新的 {N+1}.root.json，并将其 version 设置为 N + 1
}
}
```

11.4.9.2.2 索引

索引文件记录了镜像中所有的组件以及组件的所有者信息。

其格式如下：

```
{
    "signatures": [                                             # 该文件的签名
        {
            "keyid": "{id-of-index-key-1}",                     # 第一个参与签名的 key 的 ID
            "sig": "{signature-by-index-key-1}",                # 该私钥对此文件 signed
                ↪ 部分签名的结果
        },
        ...
        {
            "keyid": "{id-of-root-key-N}",                      # 第 N 个参与签名私钥的 ID
            "sig": "{signature-by-root-key-N}"                  # 该私钥对此文件 signed
                ↪ 部分签名的结果
        }
    ],
    "signed": {
        "_type": "index",                                       # 指示该文件类型
        "components": {                                         # 组件列表
            "{component1}": {                                   # 第一个组件的名称
                "hidden": {bool},                               # 是否是隐藏组件
                "owner": "{owner-id}",                          # 组件管理员 ID
                "standalone": {bool},                           # 该组件是否可独立运行
```

```
                "url": "/{component}.json",                     # 获取组件的地址，需要加上版本号
                    ↪ : /{N}.{component}.json
                "yanked": {bool}                                # 该组件是否已被标记为删除
            },
            ...
            "{componentN}": {                                   # 第 N 个组件的名称
                ...
            },
        },
        "default_components": ["{component1}.."{componentN}"], # 镜像必须包含的默认组件，
            ↪ 该字段目前固定为空（未启用）
        "expires": "{expiration-date-of-this-file}",            # 该文件的过期时间，
            ↪ 过期后客户端会拒绝此文件
        "owners": {
            "{owner1}": {                                       # 第一个属主的 ID
                "keys": {                                       # 只有 keys
                    ↪ 中记录的密钥签名才是合法的
                    "{id-of-the-key-1}": {                      # 该属主的第一个密钥
                        "keytype": "rsa",                       # 密钥类型，目前固定为 rsa
                        "keyval": {                             # 密钥的 payload
                            "public": "{public-key-content}"    # 表示公钥内容
                        },
                        "scheme": "rsassa-pss-sha256"           # 目前固定为 rsassa-pss-sha256
                    },
                    ...
                    "{id-of-the-key-N}": {                      # 该属主的第 N 个密钥
                        ...
                    }
                },
                "name": "{owner-name}",                         # 该属主的名字
                "threshod": {N}                                 #
                    ↪ 指示该属主拥有的组件必须含有至少 N 个合法签名
            },
            ...
            "{ownerN}": {                                       # 第 N 个属主的 ID
                ...
            }
        }
        "spec_version": "0.1.0",                                # 本文件遵循的规范版本，
            ↪ 未来变更文件结构需要升级版本号，目前为 0.1.0
        "version": {N}                                          # 本文件的版本号，
            ↪ 每次更新文件需要创建一个新的 {N+1}.index.json，并将其 version 设置为 N + 1
    }
}
```

11.4.9.2.3 组件

组件元信息文件记录了特定组件的平台以及版本信息。

其格式如下：

```
{
    "signatures": [                                          # 该文件的签名
        {
            "keyid": "{id-of-index-key-1}",                  # 第一个参与签名的 key 的 ID
            "sig": "{signature-by-index-key-1}",             # 该私钥对此文件 signed
                ↪ 部分签名的结果
        },
        ...
        {
            "keyid": "{id-of-root-key-N}",                   # 第 N 个参与签名私钥的 ID
            "sig": "{signature-by-root-key-N}"               # 该私钥对此文件 signed
                ↪ 部分签名的结果
        }
    ],
    "signed": {
        "_type": "component",                                # 指示该文件类型
        "description": "{description-of-the-component}",     # 该组件的描述信息
        "expires": "{expiration-date-of-this-file}",         # 该文件的过期时间，
            ↪ 过期后客户端会拒绝此文件
        "id": "{component-id}",                              # 该组件的 ID，具有全局唯一性
        "nightly": "{nightly-cursor}",                       # nightly 游标，值为最新的
            ↪ nightly 的版本号（如 v5.0.0-nightly-20201209）
        "platforms": {                                       # 该组件支持的平台（如 darwin/
            ↪ amd64，linux/arm64 等）
            "{platform-pair-1}": {
                "{version-1}": {                             # Semantic Version 版本号（如 v1
                    ↪ .0.0 等）
                    "dependencies": null,                    # 用于约定组件之间的依赖关系，
                        ↪ 该字段尚未使用，固定为 null
                    "entry": "{entry}",                      # 入口二进制文件位于 tar
                        ↪ 包的相对路径
                    "hashs": {                               # tar 包的 checksum，我们使用
                        ↪ sha256 和 sha512
                        "sha256": "{sum-of-sha256}",
                        "sha512": "{sum-of-sha512}",
                    },
                    "length": {length-of-tar},               # tar 包的长度
                    "released": "{release-time}",            # 该版本的 release 时间
                    "url": "{url-of-tar}",                   # tar 包的下载地址
                    "yanked": {bool}                         # 该版本是否已被禁用
                }
```

```
            },
            ...
            "{platform-pair-N}": {
                ...
            }
        },
        "spec_version": "0.1.0",                                    # 本文件遵循的规范版本，
            ↪ 未来变更文件结构需要升级版本号，目前为 0.1.0
        "version": {N}                                              # 本文件的版本号，
            ↪ 每次更新文件需要创建一个新的 {N+1}.{component}.json，并将其 version 设置为 N + 1
}
```

11.4.9.2.4 快照

快照文件记录了各个元文件当前的版本号。

其格式如下：

```
{
    "signatures": [                                                 # 该文件的签名
        {
            "keyid": "{id-of-index-key-1}",                         # 第一个参与签名的 key 的 ID
            "sig": "{signature-by-index-key-1}",                    # 该私钥对此文件 signed
                ↪ 部分签名的结果
        },
        ...
        {
            "keyid": "{id-of-root-key-N}",                          # 第 N 个参与签名私钥的 ID
            "sig": "{signature-by-root-key-N}"                      # 该私钥对此文件 signed
                ↪ 部分签名的结果
        }
    ],
    "signed": {
        "_type": "snapshot",                                        # 指示该文件类型
        "expires": "{expiration-date-of-this-file}",                # 该文件的过期时间，
            ↪ 过期后客户端会拒绝此文件
        "meta": {                                                   # 其他元文件的信息
            "/root.json": {
                "length": {length-of-json-file},                    # root.json 的长度
                "version": {version-of-json-file}                   # root.json 的 version
            },
            "/index.json": {
                "length": {length-of-json-file},
                "version": {version-of-json-file}
            },
            "/{component-1}.json": {
```

```
                "length": {length-of-json-file},
                "version": {version-of-json-file}
            },
            ...
            "/{component-N}.json": {
                ...
            }
        },
        "spec_version": "0.1.0",                                    # 本文件遵循的规范版本，
            ↪ 未来变更文件结构需要升级版本号，目前为 0.1.0
        "version": 0                                                # 本文件的版本号，固定为 0
    }
```

11.4.9.2.5 时间戳

时间戳文件记录了当前快照 checksum。

其文件格式如下：

```
{
    "signatures": [                                                 # 该文件的签名
        {
            "keyid": "{id-of-index-key-1}",                         # 第一个参与签名的 key 的 ID
            "sig": "{signature-by-index-key-1}",                    # 该私钥对此文件 signed
                ↪ 部分签名的结果
        },
        ...
        {
            "keyid": "{id-of-root-key-N}",                          # 第 N 个参与签名私钥的 ID
            "sig": "{signature-by-root-key-N}"                      # 该私钥对此文件 signed
                ↪ 部分签名的结果
        }
    ],
    "signed": {
        "_type": "timestamp",                                       # 指示该文件类型
        "expires": "{expiration-date-of-this-file}",                # 该文件的过期时间，
            ↪ 过期后客户端会拒绝此文件
        "meta": {                                                   # snapshot.json 的信息
            "/snapshot.json": {
                "hashes": {
                    "sha256": "{sum-of-sha256}"                     # snapshot.json 的 sha256
                },
                "length": {length-of-json-file}                     # snapshot.json 的长度
            }
        },
```

```
    "spec_version": "0.1.0",                                    # 本文件遵循的规范版本，
        ↪ 未来变更文件结构需要升级版本号，目前为 0.1.0
    "version": {N}                                              # 本文件的版本号，
        ↪ 每次更新文件需要覆盖 timestamp.json，并将其 version 设置为 N + 1
```

11.4.9.3 客户端工作流程

客户端通过以下逻辑保证从镜像下载到的文件是安全的：

- 客户端安装时随 binary 附带了一个 root.json
- 客户端运行时以已有的 root.json 为基础，做如下操作：
  1. 获取 root.json 中的 version，记为 N
  2. 向镜像请求 {N+1}.root.json，若成功，使用 root.json 中记录的公钥验证该文件是否合法
- 向镜像请求 timestamp.json，并使用 root.json 中记录的公钥验证该文件是否合法
- 检查 timestamp.json 中记录的 snapshot.json 的 checksum 和本地的 snapshot.json 的 checksum 是否吻合
  - 若不吻合，则向镜像请求最新的 snapshot.json 并使用 root.json 中记录的公钥验证该文件是否合法
- 对于 index.json 文件，从 snapshot.json 中获取其版本号 N，并请求 {N}.index.json，然后使用 root.json 中记录的公钥验证该文件是否合法
- 对于组件（如 tidb.json，tikv.json），从 snapshot.json 中获取其版本号 N，并请求 {N}.{component}.json，然后使用 index.json 中记录的公钥验证该文件是否合法
- 对于组件 tar 文件，从 {component}.json 中获取其 url 及 checksum，请求 url 得到 tar 包，并验证 checksum 是否正确

11.4.10 TiUP 组件文档

11.4.10.1 本地快速部署 TiDB 集群

TiDB 集群是由多个组件构成的分布式系统，一个典型的 TiDB 集群至少由 3 个 PD 节点、3 个 TiKV 节点和 2 个 TiDB 节点构成。对于想要快速体验 TiDB 的用户来说，手工部署这么多组件是非常耗时且麻烦的事情。本文介绍 TiUP 中的 playground 组件，以及如何通过 playground 组件快速搭建一套本地的 TiDB 测试环境。

11.4.10.1.1 playground 组件介绍

playground 组件的基本用法：

```
tiup playground ${version} [flags]
```

如果直接执行 `tiup playground` 命令，则 TiUP playground 会使用本地安装的 TiDB/TiKV/PD 组件或者安装这些组件的稳定版本，来启动一个由 1 个 TiKV、1 个 TiDB、1 个 PD 实例构成的集群。该命令实际做了以下事情：

- 因为该命令没有指定 playground 的版本，TiUP 会先查找已安装的 playground 的最新版本，假设已安装的 playground 最新版为 v0.0.6，则该命令相当于 tiup playground:v0.0.6
- 如果 playground 从未安装过任何版本的 TiDB/TiKV/PD 组件，TiUP 会先安装这些组件的最新稳定版，然后再启动运行这些组件的实例

- 因为该命令没有指定 TiDB/PD/TiKV 各组件的版本，默认情况下，它会使用各组件的最新发布版本，假设当前为 v4.0.0-rc，则该命令相当于 tiup playground:v0.0.6 v4.0.0-rc
- 因为该命令也没有指定各组件的个数，默认情况下，它会启动由 1 个 TiDB、1 个 TiKV 和 1 个 PD 实例构成的最小化集群
- 在依次启动完各个 TiDB 组件后，playground 会提醒集群启动成功，并告诉你一些有用的信息，譬如如何通过 MySQL 客户端连接集群、如何访问TiDB Dashboard 等

playground 的命令行参数说明：

```
Flags:
      --db int                    设置集群中的 TiDB 数量（默认为1）
      --db.binpath string         指定 TiDB 二进制文件的位置（开发调试用，可忽略）
      --db.config string          指定 TiDB 的配置文件（开发调试用，可忽略）
      --db.host host              指定 TiDB 的监听地址
      --drainer int               设置集群中 Drainer 数据
      --drainer.binpath string    指定 Drainer 二进制文件的位置（开发调试用，可忽略）
      --drainer.config string     指定 Drainer 的配置文件
  -h, --help                      打印帮助信息
      --host string               设置每个组件的监听地址（默认为 127.0.0.1），
          ↪ 如果要提供给别的电脑访问，可设置为 0.0.0.0
      --kv int                    设置集群中的 TiKV 数量（默认为1）
      --kv.binpath string         指定 TiKV 二进制文件的位置（开发调试用，可忽略）
      --kv.config string          指定 TiKV 的配置文件（开发调试用，可忽略）
      --monitor                   是否启动监控
      --pd int                    设置集群中的 PD 数量（默认为1）
      --pd.binpath string         指定 PD 二进制文件的位置（开发调试用，可忽略）
      --pd.config string          指定 PD 的配置文件（开发调试用，可忽略）
      --pump int                  指定集群中 Pump 的数量（非 0 的时候 TiDB 会开启 TiDB Binlog）
      --pump.binpath string       指定 Pump 二进制文件的位置（开发调试用，可忽略）
      --pump.config string        指定 Pump 的配置文件（开发调试用，可忽略）
      --tiflash int               设置集群中 TiFlash 数量（默认为0）
      --tiflash.binpath string    指定 TiFlash 的二进制文件位置（开发调试用，可忽略）
      --tiflash.config string     指定 TiFlash 的配置文件（开发调试用，可忽略）
```

11.4.10.1.2 使用示例

查看可用的 TiDB 版本

```
tiup list tidb
```

启动一个指定版本的 TiDB 集群

```
tiup playground ${version}
```

将 `${version}` 替换为所需的版本号。

启动一个每日构建版的 TiDB 集群

```
tiup playground nightly
```

nightly 就是这个集群的版本号，这里指定为每日构建版本。

覆盖 PD 的默认配置

首先，你需要复制 PD 的配置模版。假设你将复制的配置文件放置在 `~/config/pd.toml`，按需修改一些内容后，执行以下命令可以覆盖 PD 的默认配置：

```
tiup playground --pd.config ~/config/pd.toml
```

替换默认的二进制文件

默认启动 playground 时，各个组件都是使用官方镜像组件包中的二进制文件启动的，如果本地编译了一个临时的二进制文件想要放入集群中测试，可以使用 `--{comp}.binpath` 这个参数替换，例如执行以下命令替换 TiDB 的二进制文件：

```
tiup playground --db.binpath /xx/tidb-server
```

启动多个组件实例

默认情况下各启动一个 TiDB、TiKV 和 PD 实例，如果希望启动多个，可以加上如下参数：

```
tiup playground --db 3 --pd 3 --kv 3
```

11.4.10.1.3 快速连接到由 playground 启动的 TiDB 集群

TiUP 提供了 `client` 组件，用于自动寻找并连接 playground 在本地启动的 TiDB 集群，使用方式为：

```
tiup client
```

该命令会在控制台上提供当前机器上由 playground 启动的 TiDB 集群列表，选中需要连接的 TiDB 集群，点击回车后，可以打开一个自带的 MySQL 客户端以连接 TiDB。

11.4.10.1.4 查看已启动集群的信息

```
tiup playground display
```

可以看到如下信息：

```
Pid    Role     Uptime
---    ----     ------
84518  pd       35m22.929404512s
84519  tikv     35m22.927757153s
84520  pump     35m22.92618275s
86189  tidb     exited
86526  tidb     34m28.293148663s
86190  drainer  35m19.91349249s
```

11.4.10.1.5 扩容集群

扩容集群的命令行参数与启动集群的相似。以下命令可以扩容两个 TiDB：

```
tiup playground scale-out --db 2
```

11.4.10.1.6 缩容集群

可在 `tiup playground scale-in` 命令中指定 `pid`，以缩容对应的实例。可以通过 `tiup playground display` 命令查看 `pid`。

```
tiup playground scale-in --pid 86526
```

11.4.10.2 使用 TiUP 部署运维 TiDB 线上集群

本文重在介绍如何使用 TiUP 的 cluster 组件，如果需要线上部署的完整步骤，可参考使用 TiUP 部署 TiDB 集群。

与 playground 组件用于部署本地测试集群类似，cluster 组件用于快速部署生产集群。对比 playground，cluster 组件提供了更强大的生产集群管理功能，包括对集群的升级、缩容、扩容甚至操作、审计等。

cluster 组件的帮助文档如下：

```
tiup cluster
```

```
Starting component `cluster`: /home/tidb/.tiup/components/cluster/v1.8.0/cluster
Deploy a TiDB cluster for production

Usage:
  tiup cluster [flags]
  tiup [command]

Available Commands:
  deploy      部署集群
  start       启动已部署的集群
  stop        停止集群
  restart     重启集群
  scale-in    集群缩容
  scale-out   集群扩容
  clean       清理数据
  destroy     销毁集群
  upgrade     升级集群
  exec        在集群的一个或多个机器上执行命令
  display     获取集群信息
  list        获取集群列表
  audit       查看集群操作日志
  import      导入一个由 TiDB-Ansible 部署的集群
  edit-config 编辑 TiDB 集群的配置
  reload      用于必要时重载集群配置
```

```
  patch       使用临时的组件包替换集群上已部署的组件
  help        打印 Help 信息

Flags:
  -h, --help              帮助信息
      --native-ssh        使用系统默认的 SSH 客户端
      --wait-timeout int  等待操作超时的时间
      --ssh-timeout int   SSH 连接超时时间
  -y, --yes               跳过所有的确认步骤
```

11.4.10.2.1 部署集群

部署集群的命令为 `tiup cluster deploy`，一般用法为：

```
tiup cluster deploy <cluster-name> <version> <topology.yaml> [flags]
```

该命令需要提供集群的名字、集群使用的 TiDB 版本，以及一个集群的拓扑文件。

拓扑文件的编写可参考示例。以一个最简单的拓扑为例，将下列文件保存为 `/tmp/topology.yaml`：

> **注意：**
>
> TiUP Cluster 组件的部署和扩容拓扑是使用 yaml 语法编写，所以需要注意缩进。

```
---

pd_servers:
  - host: 172.16.5.134
    name: pd-134
  - host: 172.16.5.139
    name: pd-139
  - host: 172.16.5.140
    name: pd-140

tidb_servers:
  - host: 172.16.5.134
  - host: 172.16.5.139
  - host: 172.16.5.140

tikv_servers:
  - host: 172.16.5.134
  - host: 172.16.5.139
  - host: 172.16.5.140
```

```
grafana_servers:
  - host: 172.16.5.134

monitoring_servers:
  - host: 172.16.5.134
```

TiUP 默认部署在 amd64 架构上运行的 binary，若目标机器为 arm64 架构，可以在拓扑文件中进行配置：

```
global:
  arch: "arm64"           # 让所有机器默认使用 arm64 的 binary

tidb_servers:
  - host: 172.16.5.134
    arch: "amd64"         # 这台机器会使用 amd64 的 binary
  - host: 172.16.5.139
    arch: "arm64"         # 这台机器会使用 arm64 的 binary
  - host: 172.16.5.140    # 没有配置 arch 字段的机器，会使用 global 中的默认值，这个例子中是
      ↪ arm64

...
```

假如我们想要使用 TiDB 的 v4.0.16 版本，集群名字为 prod-cluster，则执行以下命令：

```
tiup cluster deploy -p prod-cluster v4.0.16 /tmp/topology.yaml
```

执行过程中会再次确认拓扑结构并提示输入目标机器上的 root 密码（-p 表示使用密码）：

```
Please confirm your topology:
TiDB Cluster: prod-cluster
TiDB Version: v4.0.16
Type        Host          Ports        Directories
----        ----          -----        -----------
pd          172.16.5.134  2379/2380    deploy/pd-2379,data/pd-2379
pd          172.16.5.139  2379/2380    deploy/pd-2379,data/pd-2379
pd          172.16.5.140  2379/2380    deploy/pd-2379,data/pd-2379
tikv        172.16.5.134  20160/20180  deploy/tikv-20160,data/tikv-20160
tikv        172.16.5.139  20160/20180  deploy/tikv-20160,data/tikv-20160
tikv        172.16.5.140  20160/20180  deploy/tikv-20160,data/tikv-20160
tidb        172.16.5.134  4000/10080   deploy/tidb-4000
tidb        172.16.5.139  4000/10080   deploy/tidb-4000
tidb        172.16.5.140  4000/10080   deploy/tidb-4000
prometheus  172.16.5.134  9090         deploy/prometheus-9090,data/prometheus-9090
grafana     172.16.5.134  3000         deploy/grafana-3000
Attention:
    1. If the topology is not what you expected, check your yaml file.
    1. Please confirm there is no port/directory conflicts in same host.
Do you want to continue? [y/N]:
```

输入密码后 tiup-cluster 便会下载需要的组件并部署到对应的机器上，当看到以下提示时说明部署成功：

```
Deployed cluster `prod-cluster` successfully
```

11.4.10.2.2 查看集群列表

集群部署成功后，可以通过 `tiup cluster list` 命令在集群列表中查看该集群：

```
tiup cluster list
```

```
Starting /root/.tiup/components/cluster/v1.8.0/cluster list
Name          User  Version    Path                                            PrivateKey
----          ----  -------    ----                                            ----------
prod-cluster  tidb  v4.0.16    /root/.tiup/storage/cluster/clusters/prod-cluster  /root/.tiup/
    ↪ storage/cluster/clusters/prod-cluster/ssh/id_rsa
```

11.4.10.2.3 启动集群

集群部署成功后，可以执行以下命令启动该集群。如果忘记了部署的集群的名字，可以使用 `tiup cluster list` 命令查看。

```
tiup cluster start prod-cluster
```

11.4.10.2.4 检查集群状态

如果想查看集群中每个组件的运行状态，逐一登录到各个机器上查看显然很低效。因此，TiUP 提供了 `tiup cluster display` 命令，用法如下：

```
tiup cluster display prod-cluster
```

```
Starting /root/.tiup/components/cluster/v1.8.0/cluster display prod-cluster
TiDB Cluster: prod-cluster
TiDB Version: v4.0.16
ID                  Role        Host          Ports        Status     Data Dir
    ↪ Deploy Dir
--                  ----        ----          -----        ------     --------
    ↪ ----------
172.16.5.134:3000   grafana     172.16.5.134  3000         Up         -
    ↪ deploy/grafana-3000
172.16.5.134:2379   pd          172.16.5.134  2379/2380    Up|L       data/pd-2379
    ↪ deploy/pd-2379
172.16.5.139:2379   pd          172.16.5.139  2379/2380    Up|UI      data/pd-2379
    ↪ deploy/pd-2379
172.16.5.140:2379   pd          172.16.5.140  2379/2380    Up         data/pd-2379
    ↪ deploy/pd-2379
```

```
172.16.5.134:9090   prometheus  172.16.5.134  9090         Up         data/prometheus-9090
    ↪ deploy/prometheus-9090
172.16.5.134:4000   tidb        172.16.5.134  4000/10080   Up         -
    ↪ deploy/tidb-4000
172.16.5.139:4000   tidb        172.16.5.139  4000/10080   Up         -
    ↪ deploy/tidb-4000
172.16.5.140:4000   tidb        172.16.5.140  4000/10080   Up         -
    ↪ deploy/tidb-4000
172.16.5.134:20160  tikv        172.16.5.134  20160/20180  Up         data/tikv-20160
    ↪ deploy/tikv-20160
172.16.5.139:20160  tikv        172.16.5.139  20160/20180  Up         data/tikv-20160
    ↪ deploy/tikv-20160
172.16.5.140:20160  tikv        172.16.5.140  20160/20180  Up         data/tikv-20160
    ↪ deploy/tikv-20160
```

Status 列用 Up 或者 Down 表示该服务是否正常。对于 PD 组件，同时可能会带有 |L 表示该 PD 是 Leader，|UI 表示该 PD 运行着TiDB Dashboard。

11.4.10.2.5 缩容节点

> **注意：**
>
> 本节只展示缩容命令的语法示例，线上扩缩容具体步骤可参考使用 TiUP 扩容缩容 TiDB 集群。

缩容即下线服务，最终会将指定的节点从集群中移除，并删除遗留的相关文件。

由于 TiKV 和 TiDB Binlog 组件的下线是异步的（需要先通过 API 执行移除操作）并且下线过程耗时较长（需要持续观察节点是否已经下线成功），所以对 TiKV 和 TiDB Binlog 组件做了特殊处理：

- 对 TiKV 及 Binlog 组件的操作
  - TiUP cluster 通过 API 将其下线后直接退出而不等待下线完成
  - 等之后再执行集群操作相关的命令时，会检查是否存在已经下线完成的 TiKV 或者 Binlog 节点。如果不存在，则继续执行指定的操作；如果存在，则执行如下操作：
    1. 停止已经下线掉的节点的服务
    2. 清理已经下线掉的节点的相关数据文件
    3. 更新集群的拓扑，移除已经下线掉的节点
- 对其他组件的操作
  - 下线 PD 组件时，会通过 API 将指定节点从集群中删除掉（这个过程很快），然后停掉指定 PD 的服务并且清除该节点的相关数据文件
  - 下线其他组件时，直接停止并且清除节点的相关数据文件

缩容命令的基本用法：

```
tiup cluster scale-in <cluster-name> -N <node-id>
```

它需要指定至少两个参数，一个是集群名字，另一个是节点 ID。节点 ID 可以参考上一节使用 `tiup cluster display` 命令获取。

比如想缩容 172.16.5.140 上的 TiKV 节点，可以执行：

```
tiup cluster scale-in prod-cluster -N 172.16.5.140:20160
```

通过 `tiup cluster display` 可以看到该 TiKV 已经被标记为 `Offline`：

```
tiup cluster display prod-cluster
```

```
Starting /root/.tiup/components/cluster/v1.8.0/cluster display prod-cluster
TiDB Cluster: prod-cluster
TiDB Version: v4.0.16
ID                  Role        Host          Ports        Status     Data Dir
    ↪ Deploy Dir
--                  ----        ----          -----        ------     --------
    ↪ ----------
172.16.5.134:3000   grafana     172.16.5.134  3000         Up         -
    ↪ deploy/grafana-3000
172.16.5.134:2379   pd          172.16.5.134  2379/2380    Up|L       data/pd-2379
    ↪ deploy/pd-2379
172.16.5.139:2379   pd          172.16.5.139  2379/2380    Up|UI      data/pd-2379
    ↪ deploy/pd-2379
172.16.5.140:2379   pd          172.16.5.140  2379/2380    Up         data/pd-2379
    ↪ deploy/pd-2379
172.16.5.134:9090   prometheus  172.16.5.134  9090         Up         data/prometheus-9090
    ↪ deploy/prometheus-9090
172.16.5.134:4000   tidb        172.16.5.134  4000/10080   Up         -
    ↪ deploy/tidb-4000
172.16.5.139:4000   tidb        172.16.5.139  4000/10080   Up         -
    ↪ deploy/tidb-4000
172.16.5.140:4000   tidb        172.16.5.140  4000/10080   Up         -
    ↪ deploy/tidb-4000
172.16.5.134:20160  tikv        172.16.5.134  20160/20180  Up         data/tikv-20160
    ↪ deploy/tikv-20160
172.16.5.139:20160  tikv        172.16.5.139  20160/20180  Up         data/tikv-20160
    ↪ deploy/tikv-20160
172.16.5.140:20160  tikv        172.16.5.140  20160/20180  Offline    data/tikv-20160
    ↪ deploy/tikv-20160
```

待 PD 将其数据调度到其他 TiKV 后，该节点会被自动删除。

11.4.10.2.6 扩容节点

> **注意：**
>
> 本节只用于展示扩容命令的语法示例，线上扩缩容可参考使用 TiUP 扩容缩容 TiDB 集群。

扩容的内部逻辑与部署类似，TiUP cluster 组件会先保证节点的 SSH 连接，在目标节点上创建必要的目录，然后执行部署并且启动服务。其中 PD 节点的扩容会通过 join 方式加入到集群中，并且会更新与 PD 有关联的服务的配置；其他服务直接启动加入到集群中。所有服务在扩容时都会做正确性验证，最终返回是否扩容成功。

例如，在集群 `tidb-test` 中扩容一个 TiKV 节点和一个 PD 节点：

1. 新建 scale.yaml 文件，添加新增的 TiKV 和 PD 节点 IP：

> **注意：**
>
> 需要新建一个拓扑文件，文件中只写入扩容节点的描述信息，不要包含已存在的节点。

```
---

pd_servers:
  - host: 172.16.5.140

tikv_servers:
  - host: 172.16.5.140
```

2. 执行扩容操作。TiUP cluster 根据 scale.yaml 文件中声明的端口、目录等信息在集群中添加相应的节点：

```
tiup cluster scale-out tidb-test scale.yaml
```

执行完成之后可以通过 `tiup cluster display tidb-test` 命令检查扩容后的集群状态。

11.4.10.2.7 滚动升级

> **注意：**
>
> 本节只用于展示命令的语法示例，线上升级请参考使用 TiUP 升级 TiDB。

滚动升级功能借助 TiDB 的分布式能力，升级过程中尽量保证对前端业务透明、无感知。升级时会先检查各个组件的配置文件是否合理，如果配置有问题，则报错退出；如果配置没有问题，则工具会逐个节点进行升级。其中对不同节点有不同的操作。

不同节点的操作

- 升级 PD 节点
  - 优先升级非 Leader 节点
  - 所有非 Leader 节点升级完成后再升级 Leader 节点
    * 工具会向 PD 发送一条命令将 Leader 迁移到升级完成的节点上
    * 当 Leader 已经切换到其他节点之后，再对旧的 Leader 节点做升级操作
  - 同时升级过程中，若发现有不健康的节点，工具会中止本次升级并退出，此时需要由人工判断、修复后再执行升级。
- 升级 TiKV 节点
  - 先在 PD 中添加一个迁移对应 TiKV 上 Region leader 的调度，通过迁移 Leader 确保升级过程中不影响前端业务
  - 等待迁移 Leader 完成之后，再对该 TiKV 节点进行升级更新
  - 等更新后的 TiKV 正常启动之后再移除迁移 Leader 的调度
- 升级其他服务
  - 正常停止服务再更新

升级操作

升级命令参数如下：

```
Usage:
  tiup cluster upgrade <cluster-name> <version> [flags]

Flags:
      --force                 在不 transfer leader 的情况下强制升级（危险操作）
  -h, --help                  帮助手册
      --transfer-timeout int  transfer leader 的超时时间

Global Flags:
      --native-ssh        使用系统默认的 SSH 客户端
      --wait-timeout int  等待操作超时的时间
      --ssh-timeout int   SSH 连接的超时时间
  -y, --yes               跳过所有的确认步骤
```

例如，把集群升级到 v4.0.16 的命令为：

```
tiup cluster upgrade tidb-test v4.0.16
```

11.4.10.2.8 更新配置

如果想要动态更新组件的配置，TiUP cluster 组件为每个集群保存了一份当前的配置，如果想要编辑这份配置，则执行 `tiup cluster edit-config <cluster-name>` 命令。例如：

```
tiup cluster edit-config prod-cluster
```

然后 TiUP cluster 组件会使用 vi 打开配置文件供编辑（如果你想要使用其他编辑器，请使用 EDITOR 环境变量自定义编辑器，例如 `export EDITOR=nano`），编辑完之后保存即可。此时的配置并没有应用到集群，如果想要让它生效，还需要执行：

```
tiup cluster reload prod-cluster
```

该操作会将配置发送到目标机器，重启集群，使配置生效。

> 注意：
>
> 对于监控组件，可以通过执行 `tiup cluster edit-config` 命令在对应实例上添加自定义配置路径来进行配置自定义，例如：

```
---

grafana_servers:
  - host: 172.16.5.134
    dashboard_dir: /path/to/local/dashboards/dir

monitoring_servers:
  - host: 172.16.5.134
    rule_dir: /path/to/local/rules/dir

alertmanager_servers:
  - host: 172.16.5.134
    config_file: /path/to/local/alertmanager.yml
```

路径内容格式如下：

- `grafana_servers` 的 `dashboard_dir` 字段指定的文件夹中应当含有完整的 `*.json` 文件。
- `monitoring_servers` 的 `rule_dir` 字段定义的文件夹中应当含有完整的 `*.rules.yml` 文件。
- `alertmanager_servers` 的 `config_file` 格式请参考 Alertmanager 配置模板。

在执行 `tiup reload` 时，TiUP 会将中控机上对应的配置上传到目标机器对应的配置目录中，上传之前会删除目标机器中已有的旧配置文件。如果想要修改某一个配置文件，请确保将所有的（包含未修改的）配置文件都放在同一个目录中。例如，要修改 Grafana 的 `tidb.json` 文件，可以先将 Grafana 的 `dashboards` 目录中所有的 `*.json` 文件拷贝到本地目录中，再修改 `tidb.json` 文件。否则最终的目标机器上将缺失其他的 JSON 文件。

> 注意：
>
> 如果配置了 `grafana_servers` 的 `dashboard_dir` 字段，在执行 `tiup cluster rename` 命令进行集群重命名后，需要完成以下操作：

1. 在本地的 dashboards 目录中，将集群名修改为新的集群名。
2. 在本地的 dashboards 目录中，将 datasource 更新为新的集群名（datasource 是以集群名命名的）。
3. 执行 tiup cluster reload -R grafana 命令。

11.4.10.2.9 更新组件

常规的升级集群可以使用 upgrade 命令，但是在某些场景下（例如 Debug），可能需要用一个临时的包替换正在运行的组件，此时可以用 patch 命令：

```
tiup cluster patch --help
```

```
Replace the remote package with a specified package and restart the service

Usage:
  tiup cluster patch <cluster-name> <package-path> [flags]

Flags:
  -h, --help                  帮助信息
  -N, --node strings          指定被替换的节点
      --overwrite             在未来的 scale-out 操作中使用当前指定的临时包
  -R, --role strings          指定被替换的服务类型
      --transfer-timeout int  transfer leader 的超时时间

Global Flags:
      --native-ssh        使用系统默认的 SSH 客户端
      --wait-timeout int  等待操作超时的时间
      --ssh-timeout int   SSH 连接的超时时间
  -y, --yes               跳过所有的确认步骤
```

例如，有一个 TiDB 的 hotfix 包放在 /tmp/tidb-hotfix.tar.gz，如果此时想要替换集群上的所有 TiDB，则可以执行：

```
tiup cluster patch test-cluster /tmp/tidb-hotfix.tar.gz -R tidb
```

或者只替换其中一个 TiDB：

```
tiup cluster patch test-cluster /tmp/tidb-hotfix.tar.gz -N 172.16.4.5:4000
```

11.4.10.2.10 导入 TiDB Ansible 集群

> **注意：**
>
> TiUP cluster 组件对 TiSpark 的支持目前为实验性特性，暂不支持导入启用了 TiSpark 组件的集群。

在 TiUP 之前，一般使用 TiDB Ansible 部署 TiDB 集群，import 命令用于将这部分集群过渡给 TiUP 接管。import 命令用法如下：

```
tiup cluster import --help
```

```
Import an exist TiDB cluster from TiDB-Ansible

Usage:
  tiup cluster import [flags]

Flags:
  -d, --dir string         TiDB-Ansible 的目录，默认为当前目录
  -h, --help               import 的帮助信息
      --inventory string   inventory 文件的名字（默认为 "inventory.ini"）
      --no-backup          不备份 Ansible 目录，用于存在多个 inventory 文件的 Ansible 目录
  -r, --rename NAME        重命名被导入的集群

Global Flags:
      --native-ssh         使用系统默认的 SSH 客户端
      --wait-timeout int   等待操作超时的时间
      --ssh-timeout int    SSH 连接的超时时间
  -y, --yes                跳过所有的确认步骤
```

例如，导入一个 TiDB Ansible 集群：

```
cd tidb-ansible
tiup cluster import
```

或者

```
tiup cluster import --dir=/path/to/tidb-ansible
```

11.4.10.2.11 查看操作日志

操作日志的查看可以借助 audit 命令，其用法如下：

```
Usage:
  tiup cluster audit [audit-id] [flags]

Flags:
  -h, --help   help for audit
```

在不使用 [audit-id] 参数时，该命令会显示执行的命令列表，如下：

```
tiup cluster audit
```

```
Starting component `cluster`: /home/tidb/.tiup/components/cluster/v1.8.0/cluster audit
ID      Time                       Command
--      ----                       -------
4BLhr0  2022-01-29T13:25:09+08:00  /home/tidb/.tiup/components/cluster/v1.8.0/cluster deploy test
    ↪  v4.0.16 /tmp/topology.yaml
4BKWjF  2022-01-28T23:36:57+08:00  /home/tidb/.tiup/components/cluster/v1.8.0/cluster deploy test
    ↪  v4.0.16 /tmp/topology.yaml
4BKVwH  2022-01-28T23:02:08+08:00  /home/tidb/.tiup/components/cluster/v1.8.0/cluster deploy test
    ↪  v4.0.16 /tmp/topology.yaml
4BKKH1  2022-01-28T16:39:04+08:00  /home/tidb/.tiup/components/cluster/v1.8.0/cluster destroy
    ↪ test
4BKKDx  2022-01-28T16:36:57+08:00  /home/tidb/.tiup/components/cluster/v1.8.0/cluster deploy test
    ↪  v4.0.16 /tmp/topology.yaml
```

第一列为 audit-id，如果想看某个命令的执行日志，则传入这个 audit-id：

```
tiup cluster audit 4BLhr0
```

11.4.10.2.12 在集群节点机器上执行命令

exec 命令可以很方便地到集群的机器上执行命令，其使用方式如下：

```
Usage:
  tiup cluster exec <cluster-name> [flags]

Flags:
      --command string   需要执行的命令（默认是 "ls"）
  -h, --help             帮助信息
  -N, --node strings     指定要执行的节点 ID （节点 id 通过 display 命令获取）
  -R, --role strings     指定要执行的 role
      --sudo             是否使用 root（默认为 false）
```

例如，如果要到所有的 TiDB 节点上执行 ls /tmp：

```
tiup cluster exec test-cluster --command='ls /tmp'
```

11.4.10.2.13 集群控制工具 (controllers)

在 TiUP 之前，我们用 tidb-ctl、tikv-ctl、pd-ctl 等工具操控集群，为了方便下载和使用，TiUP 将它们集成到了统一的组件 ctl 中：

```
Usage:
  tiup ctl:<cluster-version> {tidb/pd/tikv/binlog/etcd} [flags]
```

```
Flags:
  -h, --help   help for tiup
```

这个命令和之前的命令对应关系为：

```
tidb-ctl [args] = tiup ctl tidb [args]
pd-ctl [args] = tiup ctl pd [args]
tikv-ctl [args] = tiup ctl tikv [args]
binlogctl [args] = tiup ctl bindlog [args]
etcdctl [args] = tiup ctl etcd [args]
```

例如，以前查看 store 的命令为 `pd-ctl -u http://127.0.0.1:2379 store`，集成到 TiUP 中的命令为：

```
tiup ctl:<cluster-version> pd -u http://127.0.0.1:2379 store
```

11.4.10.2.14 部署机环境检查

使用 check 子命令可以对部署机的环境进行一系列检查，并输出检查结果。通过执行 check 子命令，可以发现常见的不合理配置或不支持情况。命令参数列表如下：

```
Usage:
  tiup cluster check <topology.yml | cluster-name> [flags]

Flags:
      --apply                  Try to fix failed checks
      --cluster                Check existing cluster, the input is a cluster name.
      --enable-cpu             Enable CPU thread count check
      --enable-disk            Enable disk IO (fio) check
      --enable-mem             Enable memory size check
  -h, --help                   help for check
  -i, --identity_file string   The path of the SSH identity file. If specified, public key
      ↪ authentication will be used.
  -p, --password               Use password of target hosts. If specified, password
      ↪ authentication will be used.
      --user string            The user name to login via SSH. The user must has root (or sudo)
          ↪ privilege.
```

默认情况下，此功能用于在部署前进行环境检查，通过指定 `--cluster` 参数切换模式，也可以用于对已部署集群的部署机进行检查，例如：

```
#### check deploy servers before deploy
tiup cluster check topology.yml --user tidb -p

#### check deploy servers of an existing cluster
tiup cluster check <cluster-name> --cluster
```

其中，CPU 线程数检查、内存大小检查和磁盘性能检查三项默认关闭，对于生产环境，建议将此三项检测开启并确保通过，以获得最佳性能。

- CPU：线程数大于等于 16 为通过检查
- 内存：物理内存总大小大于等于 32 GB 为通过检查
- 磁盘：对 data_dir 所在分区执行 fio 测试并记录结果

在运行检测时，若指定了 --apply 参数，程序将尝试对其中未通过的项目自动修复。自动修复仅限于部分可通过修改配置或系统参数调整的项目，其它未修复的项目需要根据实际情况手工处理。

环境检查不是部署集群的必需流程。对于生产环境建议在部署前执行环境检查并通过所有检测项。如果未通过全部检查项，也可能正常部署和运行集群，但可能无法获得最佳性能表现。

11.4.10.2.15 使用中控机系统自带的 SSH 客户端连接集群

在以上所有操作中，涉及到对集群机器的操作都是通过 TiUP 内置的 SSH 客户端连接集群执行命令，但是在某些场景下，需要使用系统自带的 SSH 客户端来对集群执行操作，比如：

- 使用 SSH 插件来做认证
- 使用定制的 SSH 客户端

此时可以通过命令行参数 --native-ssh 启用系统自带命令行：

- 部署集群: `tiup cluster deploy <cluster-name> <version> <topo> --native-ssh`
- 启动集群: `tiup cluster start <cluster-name> --native-ssh`
- 升级集群: `tiup cluster upgrade ... --native-ssh`

所有涉及集群操作的步骤都可以加上 --native-ssh 来使用系统自带的客户端。

也可以使用环境变量 TIUP_NATIVE_SSH 来指定是否使用本地 SSH 客户端，避免每个命令都需要添加 --native-ssh 参数：

```
export TIUP_NATIVE_SSH=true
#### 或者
export TIUP_NATIVE_SSh=1
#### 或者
export TIUP_NATIVE_SSH=enable
```

若环境变量和 --native-ssh 同时指定，则以 --native-ssh 为准。

> 注意：
>
> 在部署集群的步骤中，若需要使用密码的方式连接 (-p)，或者密钥文件设置了 passphrase，则需要保证中控机上安装了 sshpass，否则连接时会报错。

11.4.10.2.16 迁移中控机与备份

TiUP 相关的数据都存储在用户 home 目录的 `.tiup` 目录下，若要迁移中控机只需要拷贝 `.tiup` 目录到对应目标机器即可。

1. 在原机器 home 目录下执行 `tar czvf tiup.tar.gz .tiup`。
2. 把 `tiup.tar.gz` 拷贝到目标机器 home 目录。
3. 在目标机器 home 目录下执行 `tar xzvf tiup.tar.gz`。
4. 添加 `.tiup` 目录到 PATH 环境变量。

   如使用 bash 并且是 `tidb` 用户，在 `~/.bashrc` 中添加 `export PATH=/home/tidb/.tiup/bin:$PATH` 后执行 `source ~/.bashrc`，根据使用的 shell 与用户做相应调整。

> **注意：**
>
> 为了避免中控机磁盘损坏等异常情况导致 TiUP 数据丢失，建议定时备份 `.tiup` 目录。

11.4.10.3 搭建私有镜像

在构建私有云时，通常会使用隔离的网络环境，此时无法访问 TiUP 的官方镜像。因此，TiUP 提供了构建私有镜像的方案，它主要由 mirror 指令来实现，该方案也可用于离线部署。

11.4.10.3.1 mirror 指令介绍

`mirror` 指令的帮助文档如下：

```
tiup mirror --help
```

```
The 'mirror' command is used to manage a component repository for TiUP, you can use
it to create a private repository, or to add new component to an existing repository.
The repository can be used either online or offline.
It also provides some useful utilities to help managing keys, users and versions
of components or the repository itself.

Usage:
  tiup mirror <command> [flags]

Available Commands:
  init        Initialize an empty repository
  sign        Add signatures to a manifest file
  genkey      Generate a new key pair
  clone       Clone a local mirror from remote mirror and download all selected components
  publish     Publish a component
```

```
Flags:
  -h, --help          help for mirror
      --repo string   Path to the repository

Global Flags:
      --skip-version-check   Skip the strict version check, by default a version must be a valid
          ↪ SemVer string

Use "tiup mirror [command] --help" for more information about a command.
```

构建本地镜像将会使用 `tiup mirror clone` 指令，其基本用法如下：

```
tiup mirror clone <target-dir> [global-version] [flags]
```

- `target-dir`：指需要把克隆下来的数据放到哪个目录里。
- `global-version`：用于为所有组件快速设置一个共同的版本。

`tiup mirror clone` 命令提供了很多可选参数，日后可能会提供更多。但这些参数其实可以分为四类：

1. 克隆时是否使用前缀匹配方式匹配版本

   如果指定了 `--prefix` 参数，则会才用前缀匹配方式匹配克隆的版本号。例：指定 `--prefix` 时，填写版本 “v4.0.0” 将会匹配 “v4.0.0-rc.1”，“v4.0.0-rc.2”，“v4.0.0”

2. 是否全量克隆

   如果指定了 `--full` 参数，则会完整地克隆官方镜像。

   > 注意：
   > 如果既不指定 `--full` 参数，又不指定 `global-version` 或克隆的 component 版本，那么 TiUP 就只会克隆一些元信息。

3. 限定只克隆特定平台的包

   如果只想克隆某个平台的包，那么可以使用 `--os` 和 `--arch` 来限定：

   - 只想克隆 linux 平台的，则执行 `tiup mirror clone <target-dir> [global-version] --os=linux`
   - 只想克隆 amd64 架构的，则执行 `tiup mirror clone <target-dir> [global-version] --arch=amd64`
   - 只想克隆 linux/amd64 的，则执行 `tiup mirror clone <target-dir> [global-version] --os=linux --arch=amd64`

4. 限定只克隆组件的特定版本

   如果只想克隆某个组件的某一个版本而不是所有版本，则使用 `--<component>=<version>` 来限定，例如：

   - 只想克隆 TiDB 的 v4.0.0 版本，则执行 `tiup mirror clone <target-dir> --tidb v4.0.0`
   - 只想克隆 TiDB 的 v4.0.0 版本，以及 TiKV 的所有版本，则执行 `tiup mirror clone <target-dir> --tidb v4.0.0 --tikv all`
   - 克隆一个集群的所有组件的 v4.0.0 版本，则执行 `tiup mirror clone <target-dir> v4.0.0`

11.4.10.3.2 使用示例

使用 TiUP 离线安装 TiDB 集群

使用 `tiup mirror clone` 命令克隆的仓库可以在主机之间共享。可以通过 SCP、NFS 共享文件，也可以通过 HTTP 或 HTTPS 协议使用仓库。用 `tiup mirror set <location>` 命令来指定仓库的位置。

参考使用 TiUP 离线部署安装 TiUP 离线镜像，部署并启动 TiDB 集群。

11.4.10.4 使用 TiUP bench 组件压测 TiDB

在测试数据库性能时，经常需要对数据库进行压测，为了满足这一需求，TiUP 集成了 bench 组件。目前，TiUP bench 组件提供 TPC-C 和 TPC-H 两种压测的 workload，其命令参数如下：

```
tiup bench
```

```
Starting component `bench`: /home/tidb/.tiup/components/bench/v1.3.0/bench
Benchmark database with different workloads

Usage:
  tiup bench [command]

Available Commands:
  help        帮助信息
  tpcc        以 TPC-C 作为 workload 压测
  tpch        以 TPC-H 作为 workload 压测

Flags:
      --count int           总执行次数，0 表示无限次
  -D, --db string           被压测数据库名称（默认为 "test"）
  -d, --driver string       数据库驱动：mysql
      --dropdata            在 prepare 之前清除历史数据
  -h, --help                bench 命令自身的帮助信息
  -H, --host string         数据库的主机地址（默认 "127.0.0.1"）
      --ignore-error        忽略压测时数据库报出的错误
      --interval duration   两次报告输出时间的间隔（默认 10s）
      --isolation int       隔离级别 0: Default, 1: ReadUncommitted,
                            2: ReadCommitted, 3: WriteCommitted, 4: RepeatableRead,
                            5: Snapshot, 6: Serializable, 7: Linerizable
      --max-procs int       runtime.GOMAXPROCS
  -p, --password string     数据库密码
  -P, --port int            数据库端口（默认 4000）
      --pprof string        pprof 地址
      --silence             压测过程中不打印错误信息
      --summary             只打印 Summary
  -T, --threads int         压测并发线程数（默认 16）
      --time duration       总执行时长（默认 2562047h47m16.854775807s）
  -U, --user string         压测时使用的数据库用户（默认 "root"）
```

下文分别介绍如何使用 TiUP 运行 TPC-C 测试和 TPC-H 测试。

11.4.10.4.1 使用 TiUP 运行 TPC-C 测试

TiUP bench 组件支持如下运行 TPC-C 测试的命令和参数：

```
Available Commands:
  check       检查数据一致性
  cleanup     清除数据
  prepare     准备数据
  run         开始压测

Flags:
      --check-all         运行所有的一致性检测
  -h, --help              tpcc 的帮助信息
      --parts int         分区仓库 的数量(默认 1)
      --warehouses int    仓库的数量 (默认 10)
```

TPC-C 测试步骤

1. 通过 HASH 使用 4 个分区创建 4 个仓库：

   ```
   tiup bench tpcc --warehouses 4 --parts 4 prepare
   ```

2. 运行 TPC-C 测试：

   ```
   tiup bench tpcc --warehouses 4 run
   ```

3. 清理数据：

   ```
   tiup bench tpcc --warehouses 4 cleanup
   ```

4. 检查一致性：

   ```
   tiup bench tpcc --warehouses 4 check
   ```

5. 生成 CSV 文件：

   ```
   tiup bench tpcc --warehouses 4 prepare --output-dir data --output-type=csv
   ```

6. 为指定的表生成 CSV 文件：

   ```
   tiup bench tpcc --warehouses 4 prepare --output-dir data --output-type=csv --tables history,
       ↪ orders
   ```

11.4.10.4.2 使用 TiUP 运行 TPC-H 测试

TiUP bench 组件支持如下运行 TPC-H 测试的命令和参数：

```
Available Commands:
  cleanup     清除数据
  prepare     准备数据
  run         开始压测

Flags:
      --check             检查输出数据，只有 scale 因子为 1 时有效
  -h, --help              tpch 的帮助信息
      --queries string    所有的查询语句（默认 "q1,q2,q3,q4,q5,q6,q7,q8,q9,q10,q11,q12,q13,q14,q15
          ↪ ,q16,q17,q18,q19,q20,q21,q22")
      --sf int            scale 因子
```

11.4.10.4.3 TPC-H 测试步骤

1. 准备数据：

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

2. 运行 TPC-H 测试，根据是否检查结果执行相应命令：

   - 检查结果：

```
tiup bench tpch --sf=1 --check=true run
```

   - 不检查结果：

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

3. 清理数据：

```
tiup bench tpch cleanup
```

11.5 TiDB Operator

TiDB Operator 是 Kubernetes 上的 TiDB 集群自动运维系统，提供包括部署、升级、扩缩容、备份恢复、配置变更的 TiDB 全生命周期管理。借助 TiDB Operator，TiDB 可以无缝运行在公有云或自托管的 Kubernetes 集群上。

TiDB Operator 的文档目前独立于 TiDB 文档，文档名称为 TiDB in Kubernetes 用户文档。要访问 TiDB Operator 的文档，请点击以下链接：

- TiDB in Kubernetes 用户文档

11.6 Backup & Restore (BR)

11.6.1 备份与恢复工具 BR 简介

BR 全称为 Backup & Restore，是 TiDB 分布式备份恢复的命令行工具，用于对 TiDB 集群进行数据备份和恢复。BR 只支持在 TiDB v3.1 及以上版本使用。

相比Dumpling，BR 更适合大数据量的场景。

本文介绍了 BR 的工作原理、推荐部署配置、使用限制以及几种使用方式。

11.6.1.1 工作原理

BR 将备份或恢复操作命令下发到各个 TiKV 节点。TiKV 收到命令后执行相应的备份或恢复操作。

在一次备份或恢复中，各个 TiKV 节点都会有一个对应的备份路径，TiKV 备份时产生的备份文件将会保存在该路径下，恢复时也会从该路径读取相应的备份文件。

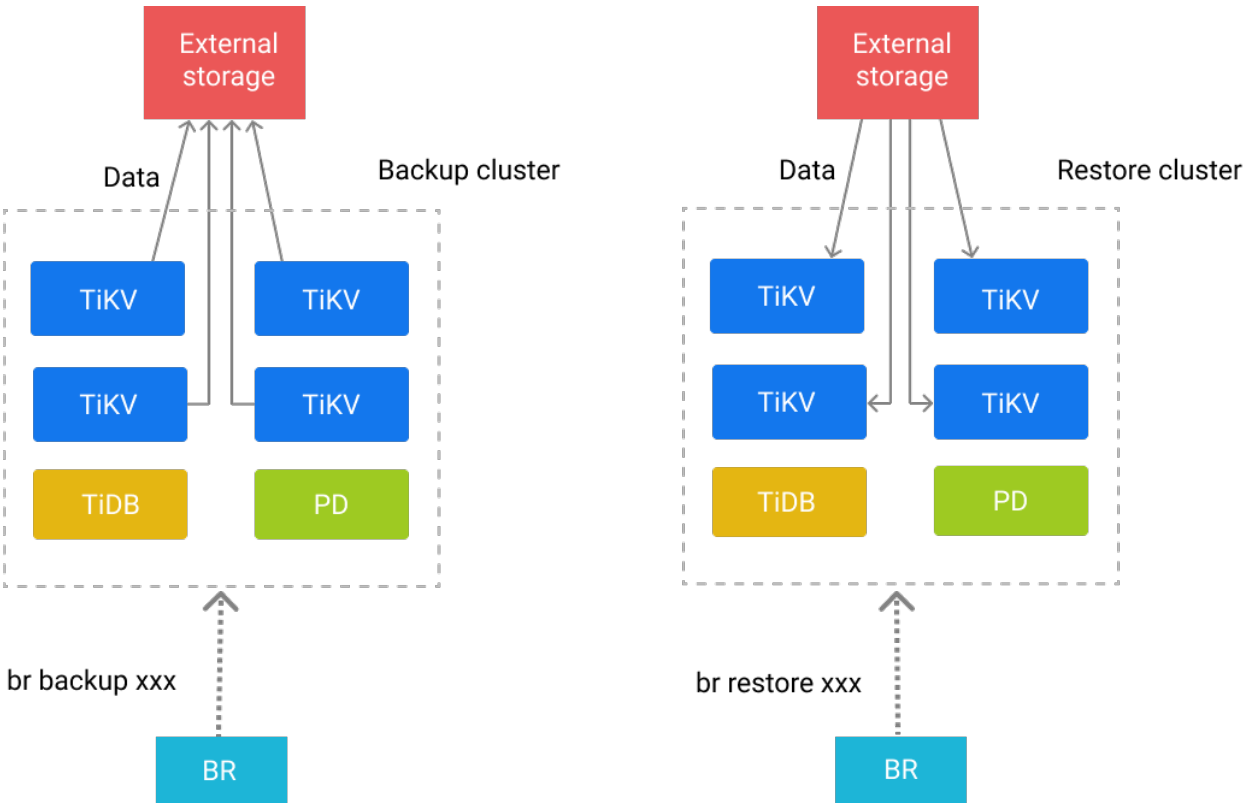


图 135: br-arch

更多信息请参阅备份恢复设计方案。

11.6.1.1.1 备份文件类型

备份路径下会生成以下两种类型文件：

- SST 文件：存储 TiKV 备份下来的数据信息
- backupmeta 文件：存储本次备份的元信息，包括备份文件数、备份文件的 Key 区间、备份文件大小和备份文件 Hash (sha256) 值
- backup.lock 文件：用于防止多次备份到同一目录

11.6.1.1.2 SST 文件命名格式

SST 文件以 storeID_regionID_regionEpoch_keyHash_cf 的格式命名。格式名的解释如下：

- storeID：TiKV 节点编号
- regionID：Region 编号
- regionEpoch：Region 版本号
- keyHash：Range startKey 的 Hash (sha256) 值，确保唯一性
- cf：RocksDB 的 ColumnFamily（默认为 default 或 write）

11.6.1.2 部署使用 BR 工具

11.6.1.2.1 推荐部署配置

- 推荐 BR 部署在 PD 节点上。
- 推荐使用一块高性能 SSD 网盘，挂载到 BR 节点和所有 TiKV 节点上，网盘推荐万兆网卡，否则带宽有可能成为备份恢复时的性能瓶颈。

> 注意：
>
> - 如果没有挂载网盘或者使用其他共享存储，那么 BR 备份的数据会生成在各个 TiKV 节点上。由于 BR 只备份 leader 副本，所以各个节点预留的空间需要根据 leader size 来预估。
> - 同时由于 v4.0 默认使用 leader count 进行平衡，所以会出现 leader size 差别大的问题，导致各个节点备份数据不均衡。

11.6.1.2.2 使用限制

下面是使用 BR 进行备份恢复的几条限制：

- BR 恢复到 TiCDC / Drainer 的上游集群时，恢复数据无法由 TiCDC / Drainer 同步到下游。
- BR 只支持在 new_collations_enabled_on_first_bootstrap 开关值相同的集群之间进行操作。这是因为 BR 仅备份 KV 数据。如果备份集群和恢复集群采用不同的排序规则，数据校验会不通过。所以恢复集群时，你需要确保 select VARIABLE_VALUE from mysql.tidb where VARIABLE_NAME='new_collation_enabled'; 语句的开关值查询结果与备份时的查询结果相一致，才可以进行恢复。

11.6.1.2.3 兼容性

BR 和 TiDB 集群的兼容性问题分为以下两方面：

- BR 部分版本和 TiDB 集群的接口不兼容
- 某些功能在开启或关闭状态下，会导致 KV 格式发生变化，因此备份和恢复期间如果没有统一开启或关闭，就会带来不兼容的问题

下表整理了会导致 KV 格式发生变化的功能。

功能	相关 issue	解决方式
New collation	#352	确保恢复时集群的 new_collations_enabled_on_first_bootstrap 变量值和备份时的一致，否则会导致数据索引不一致和 checksum 通不过。
恢复集群开启 TiCDC 同步	#364	TiKV 暂不能将 BR ingest 的 SST 文件下推到 TiCDC，因此使用 BR 恢复时候需要关闭 TiCDC。

在上述功能确保备份恢复一致的前提下，BR 和 TiKV/TiDB/PD 还可能因为版本内部协议不一致/接口不一致出现不兼容的问题，因此 BR 内置了版本检查。

版本检查

BR 内置版本会在执行备份和恢复操作前，对 TiDB 集群版本和自身版本进行对比检查。如果大版本不匹配（比如 BR v4.x 和 TiDB v5.x 上），BR 会提示退出。如要跳过版本检查，可以通过设置 `--check-requirements=false` 强行跳过版本检查，但是这样可能会引入版本不兼容的问题。TiDB v4.0 用 BR 备份后，不完全支持恢复到 v5.0 以及之后版本，详细信息见 BR 版本检查（stable 版文档）。

11.6.1.2.4 运行 BR 的最低机型配置要求

运行 BR 的最低机型配置要求如下：

CPU	内存	硬盘类型	网络
1 核	4 GB	HDD	千兆网卡

一般场景下（备份恢复的表少于 1000 张），BR 在运行期间的 CPU 消耗不会超过 200%，内存消耗不会超过 1 GB。但在备份和恢复大量数据表时，BR 的内存消耗可能会上升到 3 GB 以上。在实际测试中，备份 24000 张表大概需要消耗 2.7 GB 内存，CPU 消耗维持在 100% 以下。

11.6.1.2.5 最佳实践

下面是使用 BR 进行备份恢复的几种推荐操作：

- 推荐在业务低峰时执行备份操作，这样能最大程度地减少对业务的影响。
- BR 支持在不同拓扑的集群上执行恢复，但恢复期间对在线业务影响很大，建议低峰期或者限速 (`rate-limit`) 执行恢复。
- BR 备份最好串行执行。不同备份任务并行会导致备份性能降低，同时也会影响在线业务。
- BR 恢复最好串行执行。不同恢复任务并行会导致 Region 冲突增多，恢复的性能降低。
- 推荐在 `-s` 指定的备份路径上挂载一个共享存储，例如 NFS。这样能方便收集和管理备份文件。
- 在使用共享存储时，推荐使用高吞吐的存储硬件，因为存储的吞吐会限制备份或恢复的速度。

11.6.1.2.6 使用方式

目前支持以下几种方式来运行 BR 工具，分别是通过 SQL 语句、命令行工具或在 Kubernetes 环境下进行备份恢复。

通过 SQL 语句

在 v4.0.2 及以上版本的 TiDB 中，支持直接通过 SQL 语句进行备份恢复，具体使用示例见：

- Backup 语法
- Restore 语法

通过命令行工具

在 v3.1 以上的 TiDB 版本中，支持通过命令行工具进行备份恢复。

首先需要下载一个 BR 工具的二进制包，详见下载链接。

通过命令行工具进行备份恢复的具体操作见使用备份与恢复工具 BR。

在 Kubernetes 环境下

目前支持使用 BR 工具备份 TiDB 集群数据到兼容 S3 的存储、 Google Cloud Storage 以及持久卷，并作恢复：

> 注意：
>
> Amazon S3 和 Google Cloud Storage (GCS) 参数描述见外部存储文档。

- 备份 TiDB 集群数据到兼容 S3 的存储
- 恢复 S3 兼容存储上的备份数据
- 备份 TiDB 集群到 Google Cloud Storage
- 恢复 Google Cloud Storage 上的备份数据
- 备份 TiDB 集群到持久卷
- 恢复持久卷上的备份数据

11.6.1.3 BR 相关文档

- 使用 BR 命令行备份恢复
- BR 备份与恢复场景示例
- BR 常见问题
- 外部存储

11.6.2 使用 BR 命令行进行备份恢复

本文介绍如何 BR 命令行进行 TiDB 集群数据的备份和恢复。

在阅读本文前，请确保你已通读备份与恢复工具 BR 简介，尤其是使用限制和最佳实践这两节。

11.6.2.1 BR 命令行描述

一条 br 命令是由子命令、选项和参数组成的。子命令即不带 - 或者 -- 的字符。选项即以 - 或者 -- 开头的字符。参数即子命令或选项字符后紧跟的、并传递给命令和选项的字符。

以下是一条完整的 br 命令行：

```
br backup full --pd "${PDIP}:2379" -s "local:///tmp/backup"
```

命令行各部分的解释如下：

- `backup`：br 的子命令
- `full`：backup 的子命令
- `-s` 或 `--storage`：备份保存的路径
- `"local:///tmp/backup"`：-s 的参数，保存的路径为各个 TiKV 节点本地磁盘的 `/tmp/backup`
- `--pd`：PD 服务地址

- "${PDIP}:2379"：--pd 的参数

> 注意：
>
> 在使用 local storage 的时候，备份数据会分散在各个节点的本地文件系统中。
>
> 不建议在生产环境中备份到本地磁盘，因为在日后恢复的时候，必须手动聚集这些数据才能完成恢复工作（见恢复集群数据）。
>
> 聚集这些备份数据可能会造成数据冗余和运维上的麻烦，而且在不聚集这些数据便直接恢复的时候会遇到颇为迷惑的 SST file not found 报错。
>
> 建议在各个节点挂载 NFS 网盘，或者直接备份到 S3 对象存储中。

11.6.2.1.1 命令和子命令

BR 由多层命令组成。目前，BR 包含 backup、restore 和 version 三个子命令:

- br backup 用于备份 TiDB 集群
- br restore 用于恢复 TiDB 集群

以上三个子命令可能还包含这些子命令：

- full：可用于备份或恢复全部数据。
- db：可用于备份或恢复集群中的指定数据库。
- table：可用于备份或恢复集群指定数据库中的单张表。

11.6.2.1.2 常用选项

- --pd：用于连接的选项，表示 PD 服务地址，例如 "${PDIP}:2379"。
- -h/--help：获取所有命令和子命令的使用帮助。例如 br backup --help。
- -V (或 --version)：检查 BR 版本。
- --ca：指定 PEM 格式的受信任 CA 的证书文件路径。
- --cert：指定 PEM 格式的 SSL 证书文件路径。
- --key：指定 PEM 格式的 SSL 证书密钥文件路径。
- --status-addr：BR 向 Prometheus 提供统计数据的监听地址。

11.6.2.2 使用 BR 命令行备份集群数据

使用 br backup 命令来备份集群数据。可选择添加 full 或 table 子命令来指定备份的范围：全部集群数据或单张表的数据。

如果 BR 的版本低于 v4.0.8，而且备份时间可能超过设定的tikv_gc_life_time（默认 10m0s，即表示 10 分钟），需要手动将该参数调大。

例如，将 tikv_gc_life_time 调整为 720h：

```
mysql -h${TiDBIP} -P4000 -u${TIDB_USER} ${password_str} -Nse \
    "update mysql.tidb set variable_value='720h' where variable_name='tikv_gc_life_time'";
```

自 v4.0.8 起 BR 已经支持自适应 GC，无需手动调整 tikv_gc_life_time。

11.6.2.2.1 备份全部集群数据

要备份全部集群数据，可使用 br backup full 命令。该命令的使用帮助可以通过 br backup full -h 或 br backup full --help 来获取。

用例：将所有集群数据备份到各个 TiKV 节点的 /tmp/backup 路径，同时也会将备份的元信息文件 backupmeta 写到该路径下。

> 注意：
>
> - 经测试，在全速备份的情况下，如果备份盘和服务盘不同，在线备份会让只读线上服务的 QPS 下降 15%~25% 左右。如果希望降低影响，请参考 --ratelimit 进行限速。
> - 假如备份盘和服务盘相同，备份将会和服务争夺 I/O 资源，这可能会让只读线上服务的 QPS 骤降一半以上。请尽量禁止将在线服务的数据备份到 TiKV 的数据盘。

```
br backup full \
    --pd "${PDIP}:2379" \
    --storage "local:///tmp/backup" \
    --ratelimit 128 \
    --log-file backupfull.log
```

以上命令中，--ratelimit 选项限制了每个 TiKV 执行备份任务的速度上限（单位 MiB/s）。--log-file 选项指定把 BR 的 log 写到 backupfull.log 文件中。

备份期间有进度条在终端中显示。当进度条前进到 100% 时，说明备份已完成。在完成备份后，BR 为了确保数据安全性，还会校验备份数据。进度条效果如下：

```
br backup full \
    --pd "${PDIP}:2379" \
    --storage "local:///tmp/backup" \
    --ratelimit 128 \
    --log-file backupfull.log
Full Backup <---------/.............................................> 17.12%.
```

11.6.2.2.2 备份单个数据库的数据

要备份集群中指定单个数据库的数据，可使用 br backup db 命令。同样可通过 br backup db -h 或 br backup
↪ db --help 来获取子命令 db 的使用帮助。

用例：将数据库 test 备份到各个 TiKV 节点的 /tmp/backup 路径，同时也会将备份的元信息文件 backupmeta 写到该路径下。

```
br backup db \
    --pd "${PDIP}:2379" \
    --db test \
    --storage "local:///tmp/backup" \
    --ratelimit 128 \
    --log-file backuptable.log
```

db 子命令的选项为 --db，用来指定数据库名。其他选项的含义与备份全部集群数据相同。

备份期间有进度条在终端中显示。当进度条前进到 100% 时，说明备份已完成。在完成备份后，BR 为了确保数据安全性，还会校验备份数据。

11.6.2.2.3 备份单张表的数据

要备份集群中指定单张表的数据，可使用 br backup table 命令。同样可通过 br backup table -h 或 br backup table --help 来获取子命令 table 的使用帮助。

用例：将表 test.usertable 备份到各个 TiKV 节点的 /tmp/backup 路径，同时也会将备份的元信息文件 backupmeta 写到该路径下。

```
br backup table \
    --pd "${PDIP}:2379" \
    --db test \
    --table usertable \
    --storage "local:///tmp/backup" \
    --ratelimit 128 \
    --log-file backuptable.log
```

table 子命令有 --db 和 --table 两个选项，分别用来指定数据库名和表名。其他选项的含义与备份全部集群数据相同。

备份期间有进度条在终端中显示。当进度条前进到 100% 时，说明备份已完成。在完成备份后，BR 为了确保数据安全性，还会校验备份数据。

11.6.2.2.4 使用表库过滤功能备份多张表的数据

如果你需要以更复杂的过滤条件来备份多个表，执行 br backup full 命令，并使用 --filter 或 -f 来指定表库过滤规则。

用例：以下命令将所有 db*.tbl* 形式的表格数据备份到每个 TiKV 节点上的 /tmp/backup 路径，并将 backupmeta 文件写入该路径。

```
br backup full \
    --pd "${PDIP}:2379" \
    --filter 'db*.tbl*' \
    --storage "local:///tmp/backup" \
    --ratelimit 128 \
```

```
    --log-file backupfull.log
```

11.6.2.2.5 备份数据到 Amazon S3 后端存储

如果备份的存储并不是在本地，而是在 Amazon 的 S3 后端存储，那么需要在 storage 子命令中指定 S3 的存储路径，并且赋予 BR 节点和 TiKV 节点访问 Amazon S3 的权限。

这里可以参照 AWS 官方文档在指定的 Region 区域中创建一个 S3 桶 Bucket，如果有需要，还可以参照 AWS 官方文档 在 Bucket 中创建一个文件夹 Folder。

> **注意：**
>
> 要完成一次备份，通常 TiKV 和 BR 需要的最小权限为 s3:ListBucket，s3:PutObject 和 s3:AbortMultipartUpload。

将有权限访问该 S3 后端存储的账号的 SecretKey 和 AccessKey 作为环境变量传入 BR 节点，并且通过 BR 将权限传给 TiKV 节点。

```
export AWS_ACCESS_KEY_ID=${AccessKey}
export AWS_SECRET_ACCESS_KEY=${SecretKey}
```

在进行 BR 备份时，显示指定参数 --s3.region 和 --send-credentials-to-tikv, --s3.region 表示 S3 存储所在的区域，--send-credentials-to-tikv 表示将 S3 的访问权限传递给 TiKV 节点。

```
br backup full \
    --pd "${PDIP}:2379" \
    --storage "s3://${Bucket}/${Folder}" \
    --s3.region "${region}" \
    --send-credentials-to-tikv=true \
    --ratelimit 128 \
    --log-file backuptable.log
```

11.6.2.2.6 增量备份

如果想要备份增量，只需要在备份的时候指定上一次的备份时间戳 --lastbackupts 即可。

注意增量备份有以下限制：

- 增量备份需要与前一次全量备份在不同的路径下
- GC safepoint 必须在 lastbackupts 之前

```
br backup full\
    --pd ${PDIP}:2379 \
    --ratelimit 128 \
    -s local:///home/tidb/backupdata/incr \
    --lastbackupts ${LAST_BACKUP_TS}
```

以上命令会备份 (LAST_BACKUP_TS, current PD timestamp] 之间的增量数据。

你可以使用 validate 指令获取上一次备份的时间戳，示例如下：

```
LAST_BACKUP_TS=`br validate decode --field="end-version" -s local:///home/tidb/backupdata | tail
    ↪ -n1`
```

示例备份的增量数据记录 (LAST_BACKUP_TS, current PD timestamp] 之间的数据变更，以及这段时间内的 DDL。在恢复的时候，BR 会先把所有 DDL 恢复，而后才会恢复数据。

11.6.2.2.7 Raw KV 备份（实验性功能）

> **警告：**
>
> Raw KV 备份功能还在实验中，没有经过完备的测试。暂时请避免在生产环境中使用该功能。

在某些使用场景下，TiKV 可能会独立于 TiDB 运行。考虑到这点，BR 也提供跳过 TiDB 层，直接备份 TiKV 中数据的功能：

```
br backup raw --pd $PD_ADDR \
    -s "local://$BACKUP_DIR" \
    --start 31 \
    --ratelimit 128 \
    --end 3130303030303030 \
    --format hex \
    --cf default
```

以上命令会备份 default CF 上 [0x31, 0x3130303030303030) 之间的所有键到 $BACKUP_DIR 去。

这里，--start 和 --end 的参数会先依照 --format 指定的方式解码，再被送到 TiKV 上去，目前支持以下解码方式：

- “raw”：不进行任何操作，将输入的字符串直接编码为二进制格式的键。
- “hex”：将输入的字符串视作十六进制数字。这是默认的编码方式。
- “escape”：对输入的字符串进行转义之后，再编码为二进制格式。

11.6.2.3 使用 BR 命令行恢复集群数据

使用 br restore 命令来恢复备份数据。可选择添加 full、db 或 table 子命令来指定恢复操作的范围：全部集群数据、某个数据库或某张数据表。

> **注意：**

如果使用本地存储，在恢复前必须将所有备份的 SST 文件复制到各个 TiKV 节点上 `--storage` 指定的目录下。

即使每个 TiKV 节点最后只需要读取部分 SST 文件，这些节点也需要有所有 SST 文件的完全访问权限。原因如下：

- 数据被复制到了多个 Peer 中。在读取 SST 文件时，这些文件必须要存在于所有 Peer 中。这与数据的备份不同，在备份时，只需从单个节点读取。
- 在数据恢复的时候，每个 Peer 分布的位置是随机的，事先并不知道哪个节点将读取哪个文件。

使用共享存储可以避免这些情况。例如，在本地路径上安装 NFS，或使用 S3。利用这些网络存储，各个节点都可以自动读取每个 SST 文件，此时上述注意事项不再适用。

11.6.2.3.1 恢复全部备份数据

要将全部备份数据恢复到集群中来，可使用 `br restore full` 命令。该命令的使用帮助可以通过 `br restore full -h` 或 `br restore full --help` 来获取。

用例：将 `/tmp/backup` 路径中的全部备份数据恢复到集群中。

```
br restore full \
    --pd "${PDIP}:2379" \
    --storage "local:///tmp/backup" \
    --ratelimit 128 \
    --log-file restorefull.log
```

以上命令中，`--ratelimit` 选项限制了每个 TiKV 执行恢复任务的速度上限（单位 MiB/s）。`--log-file` 选项指定把 BR 的 log 写到 `restorefull.log` 文件中。

恢复期间还有进度条会在终端中显示，当进度条前进到 100% 时，说明恢复已完成。在完成恢复后，BR 为了确保数据安全性，还会校验恢复数据。进度条效果如下：

```
br restore full \
    --pd "${PDIP}:2379" \
    --storage "local:///tmp/backup" \
    --ratelimit 128 \
    --log-file restorefull.log
Full Restore <---------/.............................................> 17.12%.
```

11.6.2.3.2 恢复单个数据库的数据

要将备份数据中的某个数据库恢复到集群中，可以使用 `br restore db` 命令。该命令的使用帮助可以通过 `br restore db -h` 或 `br restore db --help` 来获取。

用例：将 `/tmp/backup` 路径中备份数据中的某个数据库恢复到集群中。

```
br restore db \
    --pd "${PDIP}:2379" \
    --db "test" \
    --ratelimit 128 \
    --storage "local:///tmp/backup" \
    --log-file restorefull.log
```

以上命令中 `--db` 选项指定了需要恢复的数据库名字。其余选项的含义与恢复全部备份数据相同。

> **注意：**
>
> 恢复备份数据的时候，`--db` 选项指定的数据库名必须与执行备份时候 `--db`选项指定的数据库名相同，否则无法恢复成功。由于备份数据的元文件`backupmeta` 记录了该数据库名，因此只能将数据恢复到同名的数据库。推荐做法是把备份文件恢复到另一个集群的同名数据库中。

11.6.2.3.3 恢复单张表的数据

要将备份数据中的某张数据表恢复到集群中，可以使用 `br restore table` 命令。该命令的使用帮助可通过 `br restore table -h` 或 `br restore table --help` 来获取。

用例：将 `/tmp/backup` 路径下的备份数据中的某个数据表恢复到集群中。

```
br restore table \
    --pd "${PDIP}:2379" \
    --db "test" \
    --table "usertable" \
    --ratelimit 128 \
    --storage "local:///tmp/backup" \
    --log-file restorefull.log
```

11.6.2.3.4 使用表库功能过滤恢复数据

如果你需要用复杂的过滤条件来恢复多个表，执行 `br restore full` 命令，并用 `--filter` 或 `-f` 指定使用表库过滤。

用例：以下命令将备份在 `/tmp/backup` 路径的表的子集恢复到集群中。

```
br restore full \
    --pd "${PDIP}:2379" \
    --filter 'db*.tbl*' \
    --storage "local:///tmp/backup" \
    --log-file restorefull.log
```

11.6.2.3.5 从 Amazon S3 后端存储恢复数据

如果需要恢复的数据并不是存储在本地，而是在 Amazon 的 S3 后端，那么需要在 `storage` 子命令中指定 S3 的存储路径，并且赋予 BR 节点和 TiKV 节点访问 Amazon S3 的权限。

> **注意：**
>
> 要完成一次恢复，通常 TiKV 和 BR 需要的最小权限为 `s3:ListBucket` 和 `s3:GetObject`。

将有权限访问该 S3 后端存储的账号的 SecretKey 和 AccessKey 作为环境变量传入 BR 节点，并且通过 BR 将权限传给 TiKV 节点。

```
export AWS_ACCESS_KEY_ID=${AccessKey}
export AWS_SECRET_ACCESS_KEY=${SecretKey}
```

在进行 BR 恢复时，显示指定参数 `--s3.region` 和 `--send-credentials-to-tikv`, `--s3.region` 表示 S3 存储所在的区域，`--send-credentials-to-tikv` 表示将 S3 的访问权限传递给 TiKV 节点。`--storage`参数中的 Bucket 和 Folder 分别代表需要恢复的数据所在的 S3 存储桶和文件夹。

```
br restore full \
    --pd "${PDIP}:2379" \
    --storage "s3://${Bucket}/${Folder}" \
    --s3.region "${region}" \
    --ratelimit 128 \
    --send-credentials-to-tikv=true \
    --log-file restorefull.log
```

以上命令中 `--table` 选项指定了需要恢复的表名。其余选项的含义与恢复单个数据库相同。

11.6.2.3.6 增量恢复

增量恢复的方法和使用 BR 进行全量恢复的方法并无差别。需要注意，恢复增量数据的时候，需要保证备份时指定的 `last backup ts` 之前备份的数据已经全部恢复到目标集群。

11.6.2.3.7 恢复创建在 `mysql` 数据库下的表（实验性功能）

BR 可以并且会默认备份 `mysql` 数据库下的表。

在执行恢复时，`mysql` 下的表默认不会被恢复。如果需要恢复 `mysql` 下的用户创建的表，可以通过table filter 来显式地包含目标表。以下示例中要恢复目标用户表为 `mysql.usertable`；该命令会在执行正常的恢复的同时恢复 `mysql.usertable`。

```
br restore full -f '*.*' -f '!mysql.*' -f 'mysql.usertable' -s $external_storage_url --ratelimit
    ↪ 128
```

在如上的命令中，`-f '*.*'` 用于覆盖掉默认的规则，`-f '!mysql.*'` 指示 BR 不要恢复 `mysql` 中的表，除非另有指定。`-f 'mysql.usertable'` 则指定需要恢复 `mysql.usertable`。具体原理请参考table filter 的文档。

如果只需要恢复 `mysql.usertable`，而无需恢复其他表，可以使用以下命令：

```
br restore full -f 'mysql.usertable' -s $external_storage_url --ratelimit 128
```

> 警告：
>
> 虽然系统表（例如 `mysql.tidb` 等）可以通过 BR 进行备份和恢复，但是部分系统表在恢复之后可能会出现非预期的状况，已知的异常如下：
>
> - 统计信息表（`mysql.stat_*`）无法被恢复。
> - 系统变量表（`mysql.tidb`，`mysql.global_variables`）无法被恢复。
> - 用户信息表（`mysql.user`，`mysql.columns_priv`，等等）无法被恢复。
> - GC 数据无法被恢复。
>
> 恢复系统表可能还存在更多兼容性问题。为了防止意外发生，请避免在生产环境中恢复系统表。

11.6.2.3.8 Raw KV 恢复（实验性功能）

> 警告：
>
> Raw KV 恢复功能还在实验中，没有经过完备的测试。暂时请避免在生产环境中使用该功能。

和Raw KV 备份相似地，恢复 Raw KV 的命令如下：

```
br restore raw --pd $PD_ADDR \
    -s "local://$BACKUP_DIR" \
    --start 31 \
    --end 3130303030303030 \
    --ratelimit 128 \
    --format hex \
    --cf default
```

以上命令会将范围在 `[0x31, 0x3130303030303030)` 的已备份键恢复到 TiKV 集群中。这里键的编码方式和备份时相同。

11.6.2.3.9 在线恢复（实验性功能）

> **警告：**
>
> 在线恢复功能还在实验中，没有经过完备的测试，同时还依赖 PD 的不稳定特性 Placement Rules。暂时请避免在生产环境中使用该功能。

在恢复的时候，写入过多的数据会影响在线集群的性能。为了尽量避免影响线上业务，BR 支持通过Placement rules 隔离资源。让下载、导入 SST 的工作仅仅在指定的几个节点（下称“恢复节点”）上进行，具体操作如下：

1. 配置 PD，启动 Placement rules：

```
echo "config set enable-placement-rules true" | pd-ctl
```

2. 编辑恢复节点 TiKV 的配置文件，在 server 一项中指定：

```
[server]
labels = { exclusive = "restore" }
```

3. 启动恢复节点的 TiKV，使用 BR 恢复备份的文件，和非在线恢复相比，这里只需要加上 --online 标志即可：

```
br restore full \
    -s "local://$BACKUP_DIR" \
    --ratelimit 128 \
    --pd $PD_ADDR \
    --online
```

11.6.3 BR 备份与恢复场景示例

BR 是一款分布式的快速备份和恢复工具。

本文展示了以下几种备份和恢复场景下的 BR 操作过程：

- 将单表数据备份到网络盘（推荐生产环境使用）
- 从网络磁盘恢复备份数据（推荐生产环境使用）
- 将单表数据备份到本地磁盘（推荐测试环境试用）
- 从本地磁盘恢复备份数据（推荐测试环境试用）

以帮助读者达到以下目标：

- 正确使用网络盘或本地盘进行备份或恢复
- 通过相关监控指标了解备份或恢复的状态
- 了解在备份或恢复时如何调优性能
- 处理备份时可能发生的异常

11.6.3.1 目标读者

你需要对 TiDB 和 TiKV 有一定的了解。

在阅读本文前，请确保你已通读备份与恢复工具 BR 简介，尤其是使用限制和最佳实践这两节。

11.6.3.2 环境准备

本节介绍 TiDB 的推荐部署方式、BR 使用示例中的集群版本、TiKV 集群硬件信息和集群配置。

你可以根据自己的硬件和配置来预估备份恢复的性能。

11.6.3.2.1 部署方式

推荐使用TiUP 部署 TiDB 集群，再下载TiDB Toolkit 获取 BR 工具。

11.6.3.2.2 集群版本

- TiDB: v4.0.2
- TiKV: v4.0.2
- PD: v4.0.2
- BR: v4.0.2

> **注意：**
>
> v4.0.2 为编写本文档时的最新版本。推荐读者使用最新版本 TiDB/TiKV/PD/BR，同时需要确保 BR 版本和 TiDB 相同。

11.6.3.2.3 TiKV 集群硬件信息

- 操作系统：CentOS Linux release 7.6.1810 (Core)
- CPU：16-Core Common KVM processor
- RAM：32GB
- 硬盘：500G SSD * 2
- 网卡：万兆网卡

11.6.3.2.4 配置

BR 可以直接将命令下发到 TiKV 集群来执行备份和恢复，不依赖 TiDB server 组件，因此无需对 TiDB server 进行配置。

- TiKV: 默认配置
- PD : 默认配置

11.6.3.3 使用场景

本节描述以下几种使用场景：

- 将单表数据备份到网络盘（推荐生产环境使用）
- 从网络磁盘恢复备份数据（推荐生产环境使用）
- 将单表数据备份到本地磁盘（推荐测试环境试用）
- 从本地磁盘恢复备份数据（推荐测试环境试用）

推荐使用网络盘来进行备份和恢复操作，这样可以省去收集备份数据文件的繁琐步骤。尤其在 TiKV 集群规模较大的情况下，使用网络盘可以大幅提升操作效率。

在使用 BR 进行备份或恢复操作前，需要先进行如下准备工作.

11.6.3.3.1 备份前的准备工作

如果你使用的是 TiDB v4.0.8 及以上版本，相应版本的 BR 工具已支持自适应 GC，会自动将 backupTS（默认是最新的 PD timestamp）注册到 PD 的 `safePoint`，保证 TiDB 的 GC Safe Point 在备份期间不会向前移动，即可避免手动设置 GC。

关于 `br backup` 命令的具体使用方法，参见使用备份与恢复工具 BR。

如果你使用的是 TiDB v4.0.7 及以下版本，则需要在 BR 备份前后，按照以下步骤手动设置 GC：

1. 运行`br backup` 命令前，查询 TiDB 集群的`tikv_gc_life_time` 配置项的值，并使用 MySQL 客户端将该项调整至合适的值，确保备份期间不会发生Garbage Collection (GC)。

```
SELECT * FROM mysql.tidb WHERE VARIABLE_NAME = 'tikv_gc_life_time';
UPDATE mysql.tidb SET VARIABLE_VALUE = '720h' WHERE VARIABLE_NAME = 'tikv_gc_life_time';
```

2. 在备份完成后，将该参数调回原来的值。

```
UPDATE mysql.tidb SET VARIABLE_VALUE = '10m' WHERE VARIABLE_NAME = 'tikv_gc_life_time';
```

11.6.3.3.2 恢复前的准备工作

使用 BR 进行恢复前的准备工作如下：

运行`br restore` 命令前，需要检查新集群，确保集群内没有同名的表。

11.6.3.3.3 将单表数据备份到网络盘（推荐生产环境使用）

使用 `br backup` 命令，将单表数据 `--db batchmark --table order_line` 备份到指定的网络盘路径 `local:///br_data` 下。

前置要求

- 备份前的准备工作。
- 配置一台高性能 SSD 硬盘主机为 NFS server 存储数据。其他所有 BR 节点、TiKV 节点和 TiFlash 节点为 NFS client，挂载相同的路径（例如 `/br_data`）到 NFS server 上以访问 NFS server。

- NFS server 和 NFS client 间的数据传输速率至少要达到备份集群的 TiKV 实例数 * 150MB/s。否则网络 I/O 有可能成为性能瓶颈。

> 注意：
>
> - 因为备份时候只备份单副本 (leader) 数据，所以即使集群中存在 TiFlash 副本，无需挂载 TiFlash 节点 BR 也能完成备份。
> - BR 在恢复数据时，会恢复全部副本的数据。因此在恢复时，TiFlash 节点需要有备份数据的访问权限 BR 才能完成恢复，此时也必须将 TiFlash 节点挂载到 NFS server 上。

部署拓扑

部署拓扑如下图所示：

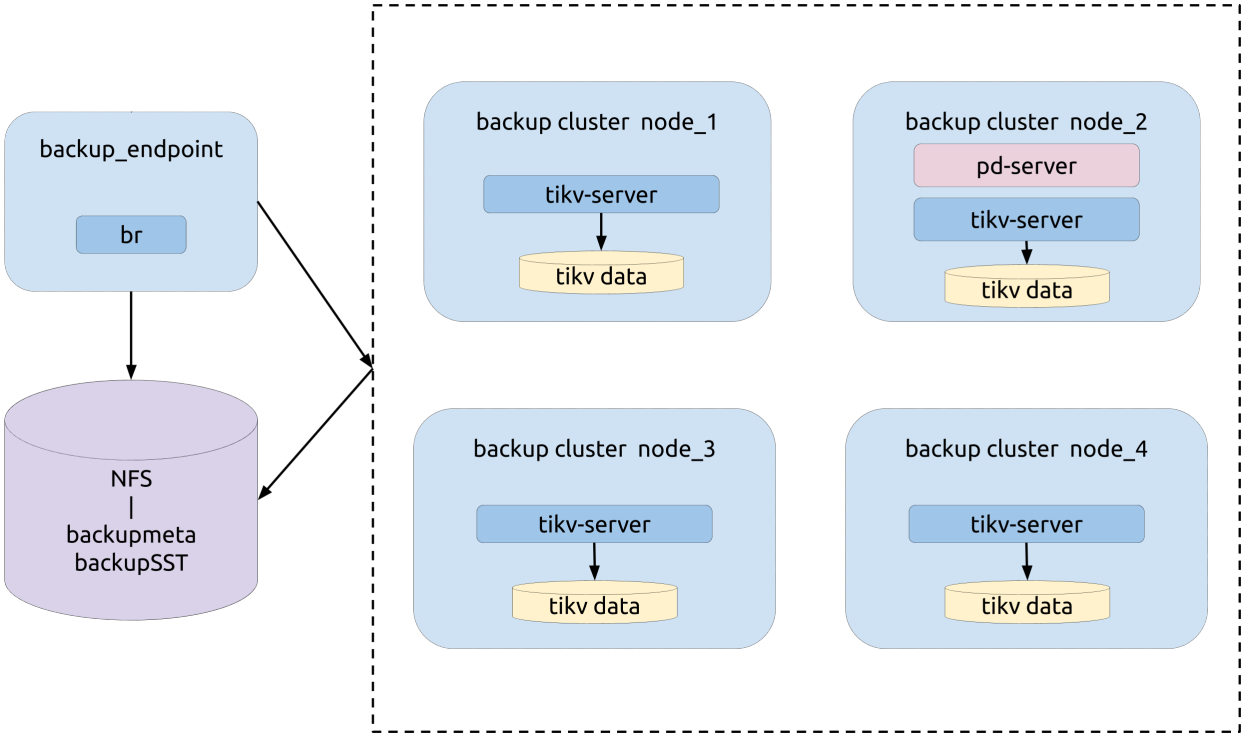


图 136: img

运行备份

备份操作前，在 TiDB 中使用 `admin checksum table order_line` 命令获得备份目标表 `--db batchmark --table order_line` 的统计信息。统计信息示例如下：

```
+----------+------------+---------------------+------------+--------------+
| Db_name  | Table_name | Checksum_crc64_xor  | Total_kvs  | Total_bytes  |
+----------+------------+---------------------+------------+--------------+
| batchmark | order_line | 10912722838344822475 | 5659888624 | 370385538778 |
+----------+------------+---------------------+------------+--------------+
1 row in set (5 min 47.59 sec)
```

图 137: img

运行 BR 备份命令：

```
bin/br backup table \
    --db batchmark \
    --table order_line \
    -s local:///br_data \
    --pd ${PD_ADDR}:2379 \
    --log-file backup-nfs.log
```

备份过程中的运行指标

在 BR 备份过程中，需关注以下监控面版中的运行指标来了解备份的状态。

Backup CPU Utilization：参与备份的 TiKV 节点（例如 backup-worker 和 backup-endpoint）的 CPU 使用率。

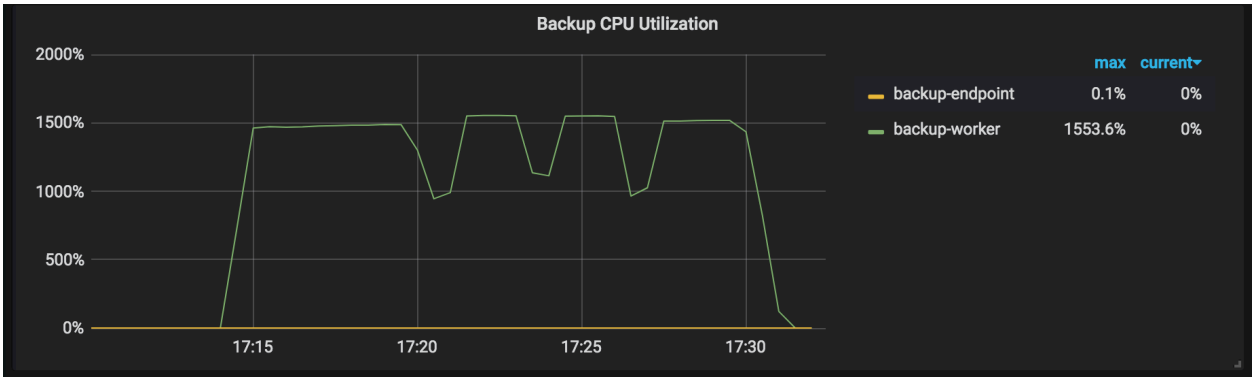


图 138: img

IO Utilization：参与备份的 TiKV 节点的 I/O 使用率。

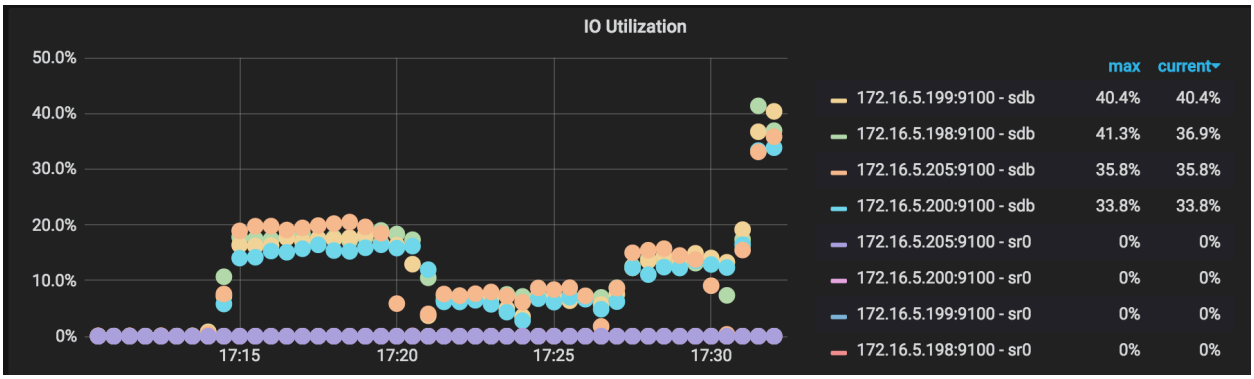


图 139: img

BackupSST Generation Throughput：参与备份的 TiKV 节点生成 backupSST 文件的吞吐。正常时单个 TiKV 节点的吞吐在 150MB/s 左右。

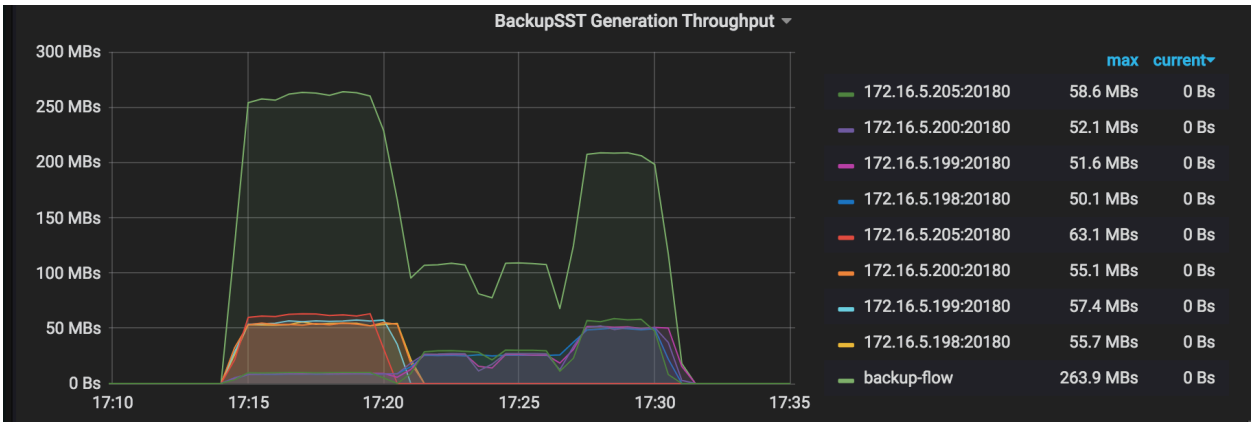


图 140: img

One Backup Range Duration：备份一个 range 的操作耗时，包括扫描耗时 (scan KV) 和保存耗时（保存为 backupSST 文件）。

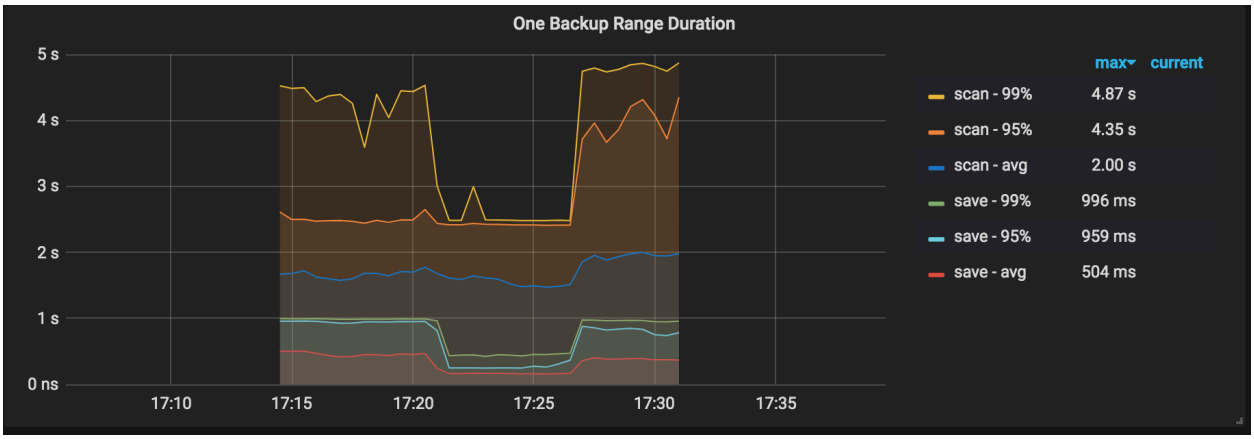


图 141: img

One Backup Subtask Duration：一次备份任务会被拆分成多个子任务。该监控项显示子任务的耗时。

注意：

- 虽然本次任务是备份单表，但因为表中有 3 个索引，所以正常会拆分成 4 个子任务。
- 下图中有 13 个点，说明有 9 次 (13-4) 重试。备份过程中可能发生 Region 调度行为，少量重试是正常的。

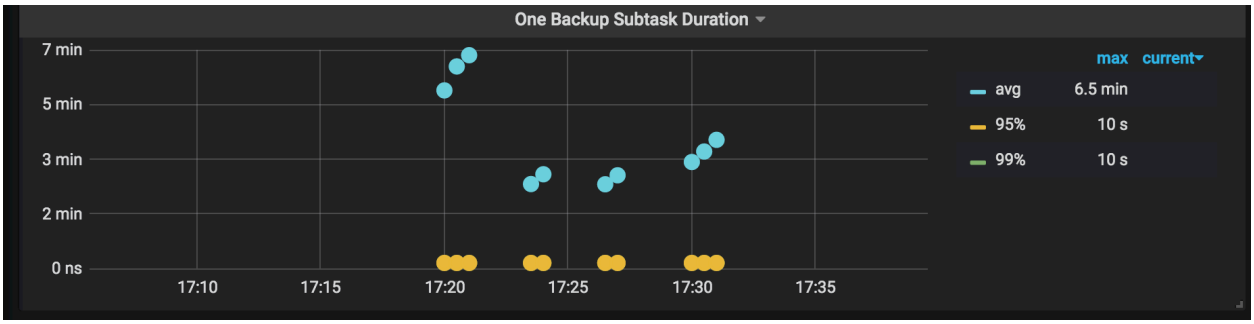


图 142: img

Backup Errors：备份过程中的错误。正常时无错误。即使出现少量错误，备份操作也有重试机制，可能会导致备份时间增加，但不会影响备份的正确性。

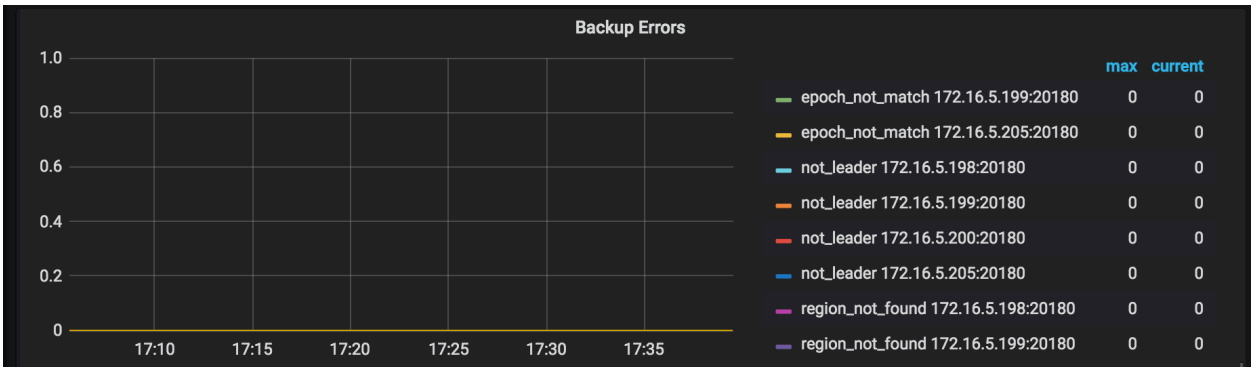


图 143: img

Checksum Request Duration：对备份集群执行 admin checksum 的耗时统计。

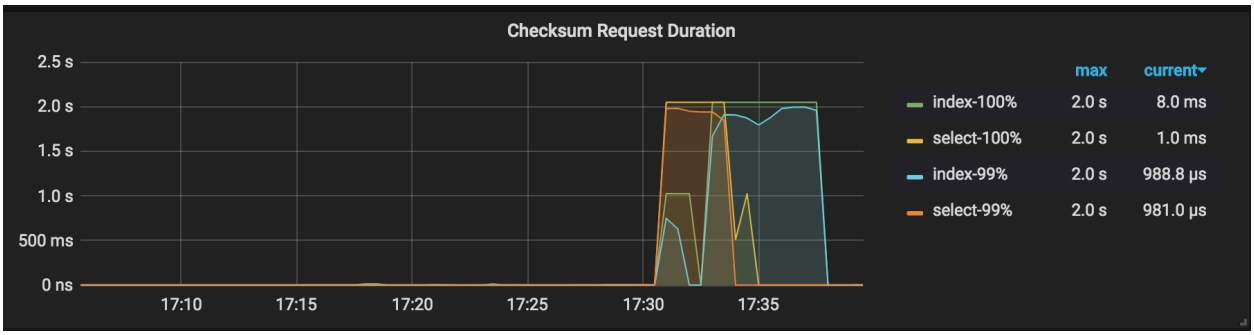


图 144: img

结果解读

BR 会在备份结束时输出备份总结到控制台。

同时使用 BR 前已设置日志的存放路径。从路径下存放的日志中可以获取此次备份的相关统计信息。在日志中搜关键字 “summary”，可以看到以下信息：

```
["Full backup Success summary:
    total backup ranges: 2,
    total success: 2,
    total failed: 0,
    total take(Full backup time): 31.802912166s,
    total take(real time): 49.799662427s,
    total size(MB): 5997.49,
    avg speed(MB/s): 188.58,
    total kv: 120000000"]
    ["backup checksum"=17.907153678s]
    ["backup fast checksum"=349.333µs]
    ["backup total regions"=43]
```

```
    [BackupTS=422618409346269185]
    [Size=826765915]
```

以上日志信息中包含以下内容：

- 备份耗时：`total take(Full backup time): 31.802912166s`
- 程序运行总耗时：`total take(real time): 49.799662427s`
- 备份数据大小：`total size(MB): 5997.49`
- 备份吞吐：`avg speed(MB/s): 188.58`
- 备份 KV 对数：`total kv: 120000000`
- 校验耗时：`["backup checksum"=17.907153678s]`
- 计算各表 checksum、KV 和 bytes 信息总和的耗时：`["backup fast checksum"=349.333µs]`
- 备份 Region 总数：`["backup total regions"=43]`
- 备份存档经压缩后在磁盘中的实际大小：`[Size=826765915]`
- 备份存档的快照时间戳：`[BackupTS=422618409346269185]`

通过以上数据可以计算得到单个 TiKV 实例的吞吐为：`avg speed(MB/s)/tikv_count` = 62.86。

性能调优

如果 TiKV 的资源使用没有出现明显的瓶颈（例如备份过程中的运行指标中的 Backup CPU Utilization 最高为 1500% 左右，IO Utilization 普遍低于 30%），可以尝试调大 `--concurrency`（默认是 4）参数以进行性能调优。该方法不适用于存在许多小表的场景。

示例如下：

```
bin/br backup table \
    --db batchmark \
    --table order_line \
    -s local:///br_data/ \
    --pd ${PD_ADDR}:2379 \
    --log-file backup-nfs.log \
    --concurrency 16
```

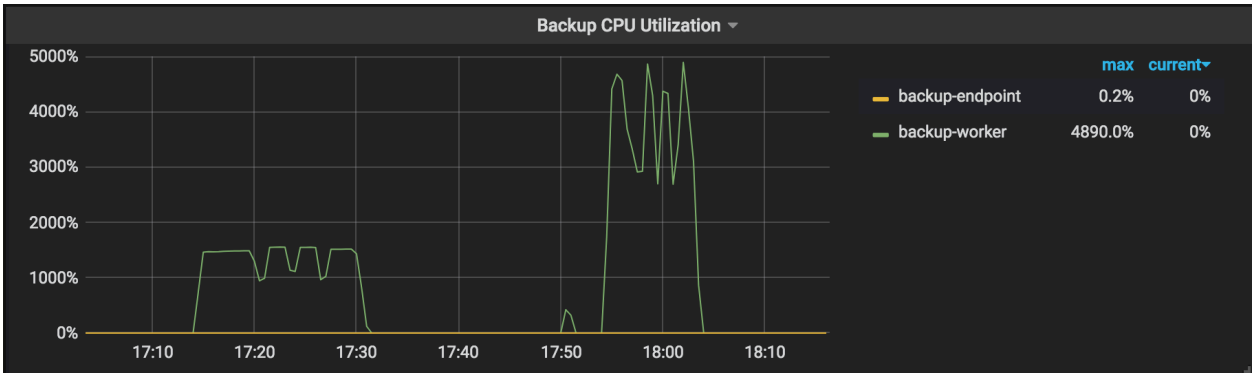


图 145: img

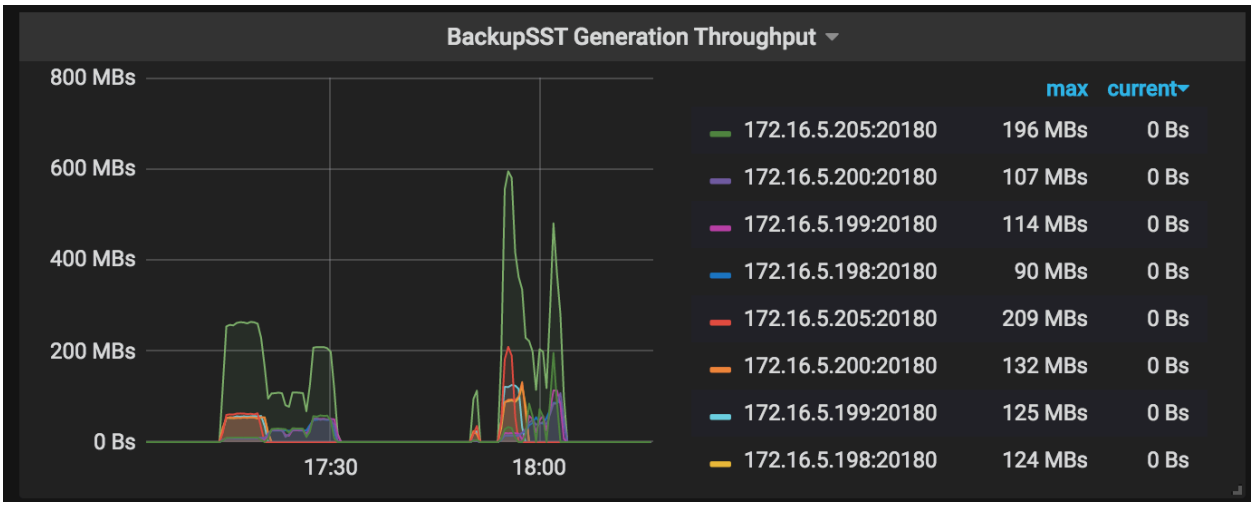


图 146: img

性能调优后的结果如下所示（保持数据大小不变）：

- 备份耗时：`total take(s)` 从 `986.43` 减少到 `535.53`
- 数据大小：`total size(MB): 353227.18`
- 备份吞吐：`avg speed(MB/s)` 从 `358.09` 提升到 `659.59`
- 单个 TiKV 实例的吞吐：`avg speed(MB/s)/tikv_count` 从 `89` 提升到 `164.89`

11.6.3.3.4 从网络磁盘恢复备份数据（推荐生产环境使用）

使用 `br restore` 命令，将一份完整的备份数据恢复到一个离线集群。暂不支持恢复到在线集群。

前置要求

- 恢复前的准备工作。

部署拓扑

部署拓扑如下图所示：

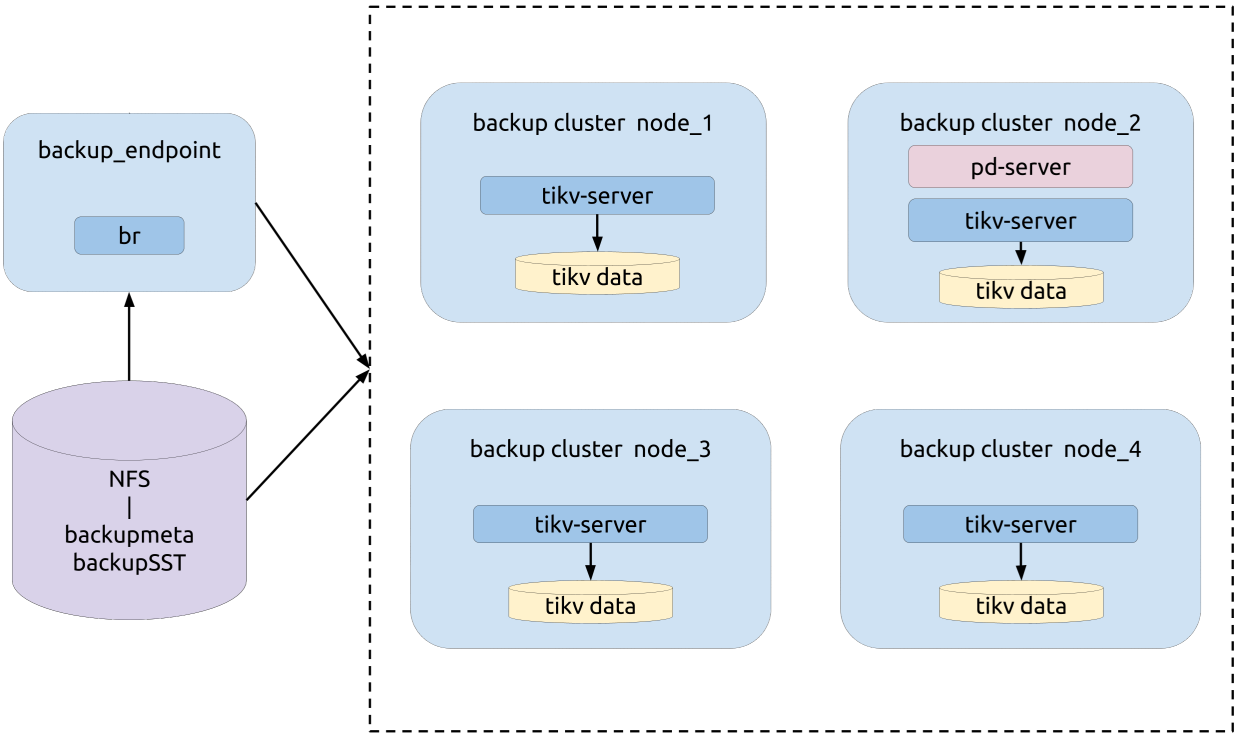


图 147: img

运行恢复

运行 `br restore` 命令：

```
bin/br restore table --db batchmark --table order_line -s local:///br_data --pd 172.16.5.198:2379
    ↪ --log-file restore-nfs.log
```

恢复过程中的运行指标

在 BR 恢复过程中，需关注以下监控面版中的运行指标来了解恢复的状态。

CPU Utilization：参与恢复的 TiKV 节点 CPU 使用率。

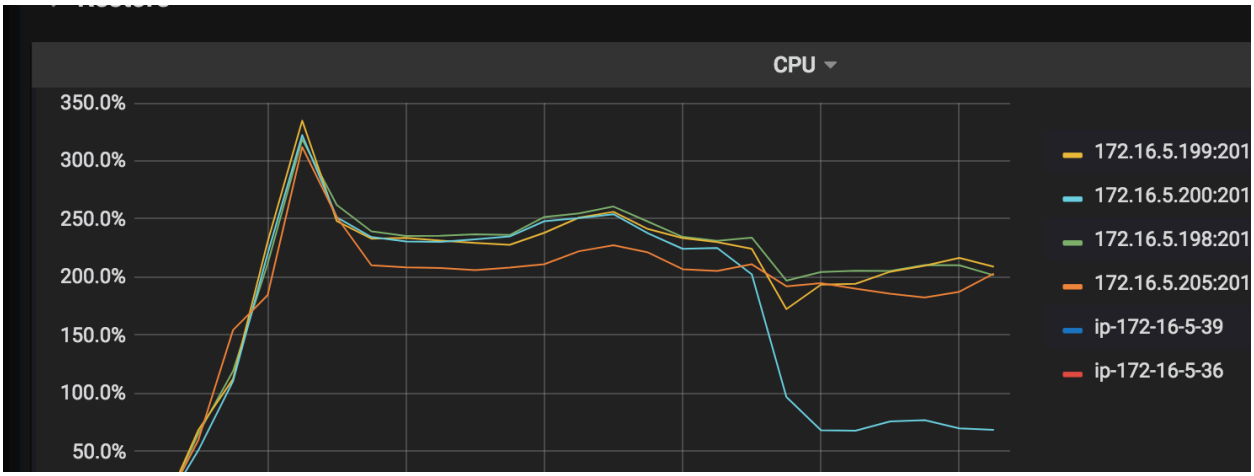


图 148: img

IO Utilization：参与恢复的 TiKV 节点的 I/O 使用率。

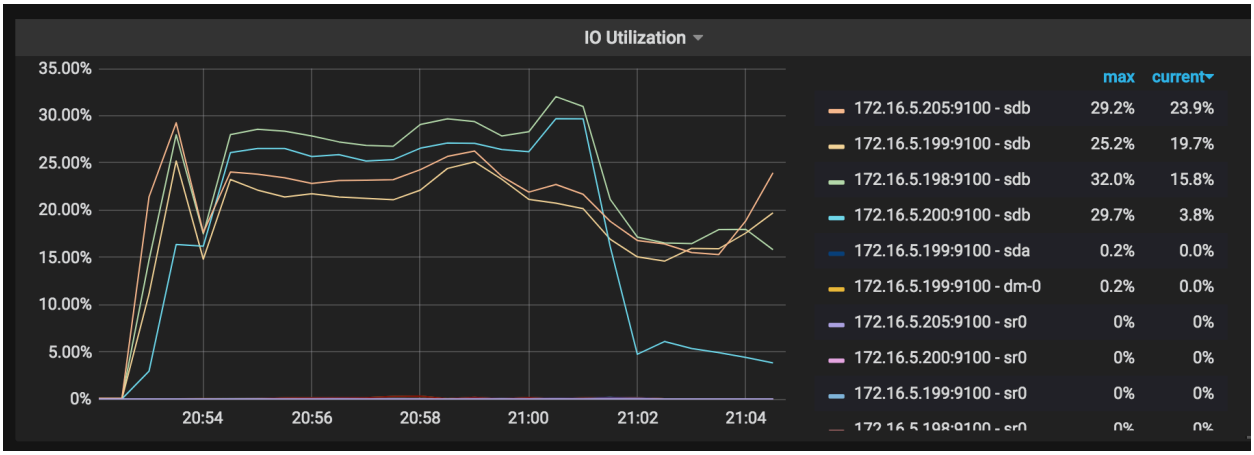


图 149: img

Region 分布：Region 分布越均匀，说明恢复资源利用越充分。

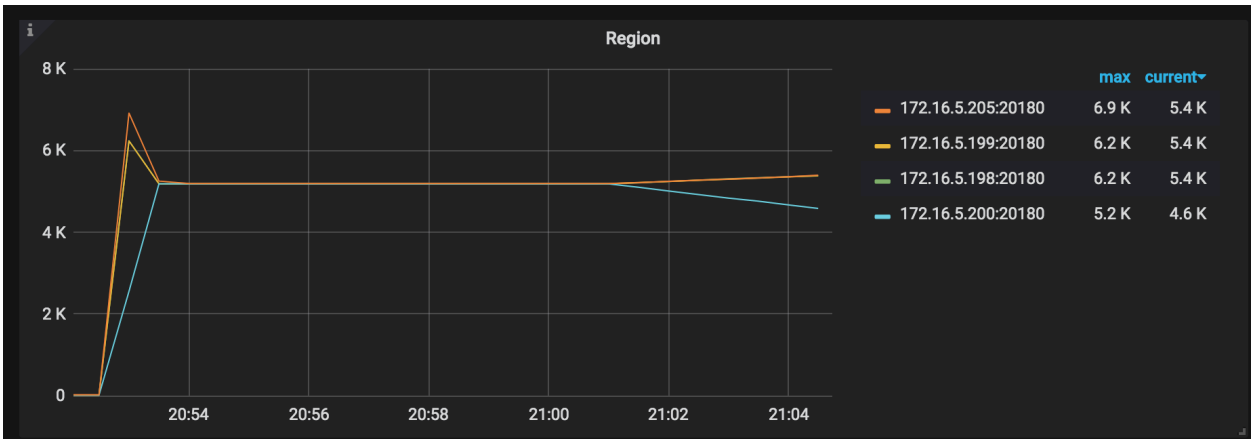


图 150: img

Process SST Duration：处理 SST 文件的延迟。恢复一张表时时，如果 `tableID` 发生了变化，需要对 `tableID` 进行 `rewrite`，否则会进行 `rename`。通常 `rewrite` 延迟要高于 `rename` 的延迟。

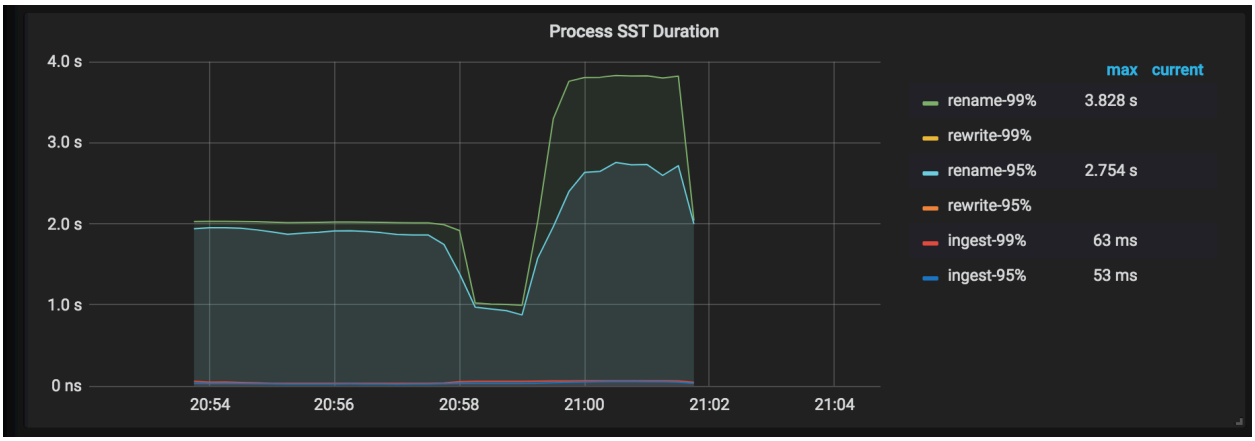


图 151: img

DownLoad SST Throughput：从 External Storage 下载 SST 文件的吞吐。

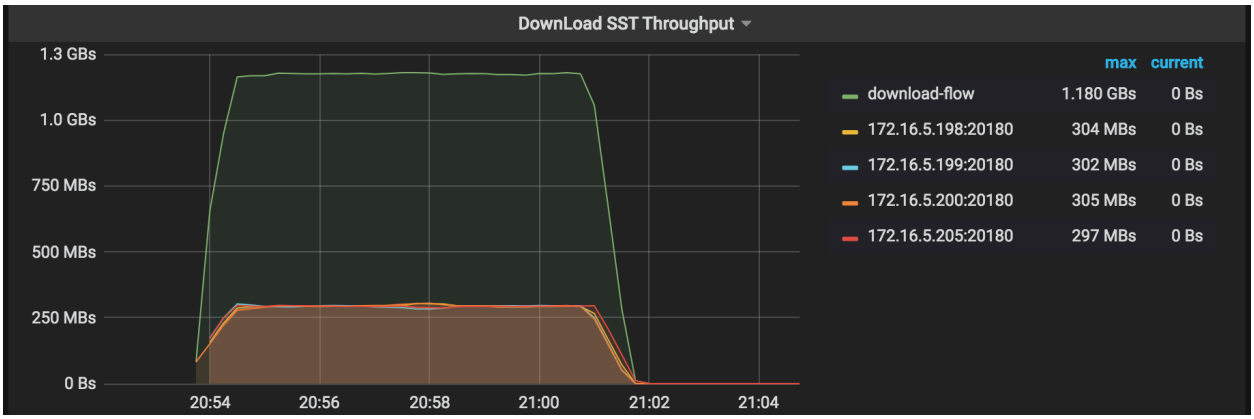


图 152: img

Restore Errors：恢复过程中的错误。

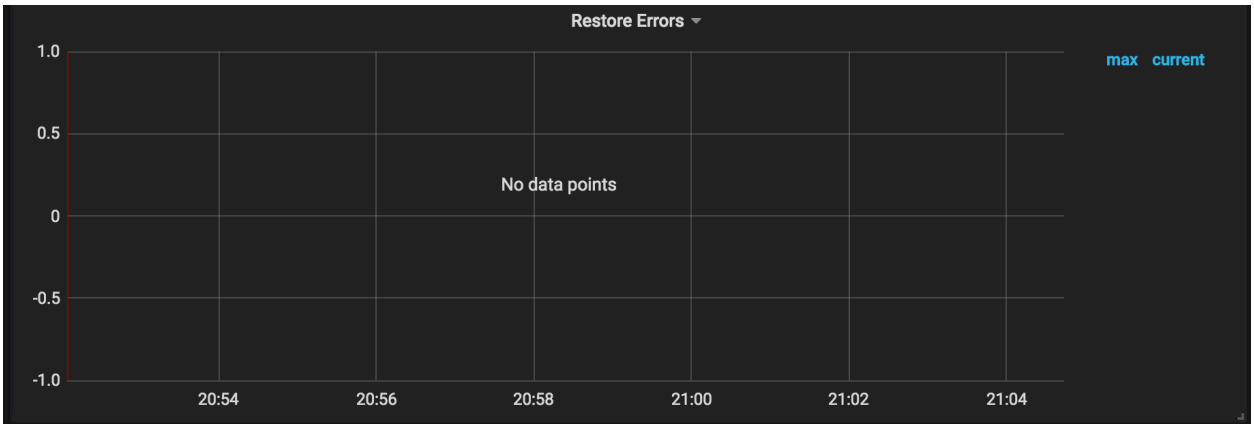


图 153: img

Checksum Request duration：对恢复集群执行 admin checksum 的耗时统计，会比备份时的 checksum 延迟高。

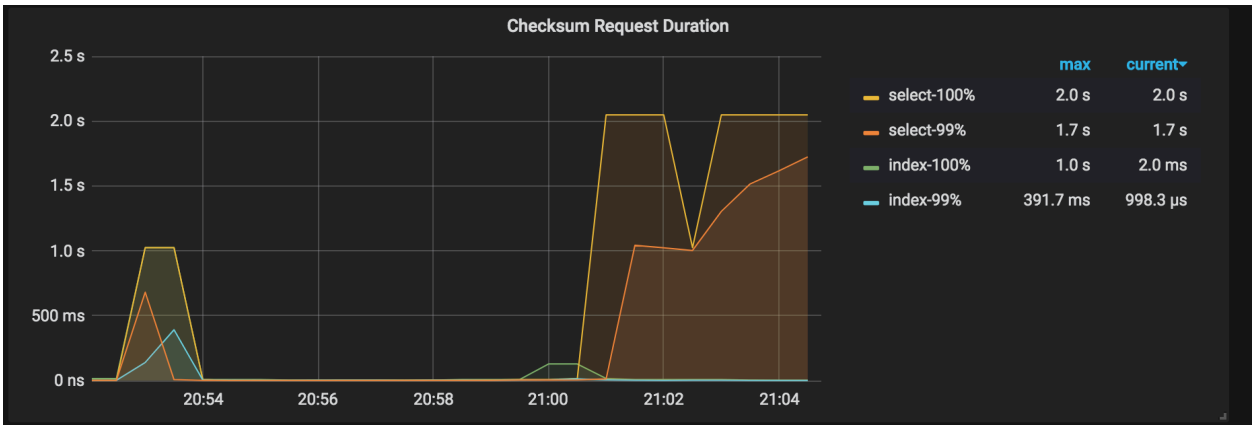


图 154: img

结果解读

使用 BR 前已设置日志的存放路径。从路径下存放的日志中可以获取此次恢复的相关统计信息。在日志中搜关键字“summary”，可以看到以下信息：

```
["Table Restore summary:
    total restore tables: 1,
    total success: 1,
    total failed: 0,
    total take(Full restore time): 17m1.001611365s,
    total take(real time): 16m1.371611365s,
    total kv: 5659888624,
    total size(MB): 353227.18,
    avg speed(MB/s): 367.42"]
    ["restore files"=9263]
    ["restore ranges"=6888]
    ["split region"=49.049182743s]
    ["restore checksum"=6m34.879439498s]
    [Size=48693068713]
```

以上日志信息中包含以下内容：

- 恢复耗时：`total take(Full restore time): 17m1.001611365s`
- 程序运行总耗时：`total take(real time): 16m1.371611365s`
- 恢复数据大小：`total size(MB): 353227.18`
- 恢复 KV 对数：`total kv: 5659888624`
- 恢复吞吐：`avg speed(MB/s): 367.42`
- Region Split 耗时：`take=49.049182743s`
- 校验耗时：`restore checksum=6m34.879439498s`
- 恢复存档在磁盘中的实际大小：`[Size=48693068713]`

根据上表数据可以计算得到：

- 单个 TiKV 吞吐：avg speed(MB/s)/tikv_count = 91.8
- 单个 TiKV 平均恢复速度：total size(MB)/(split time + restore time)/tikv_count = 87.4

性能调优

如果 TiKV 资源使用没有明显的瓶颈，可以尝试调大 --concurrency 参数（默认为 128），示例如下：

```
bin/br restore table --db batchmark --table order_line -s local:///br_data/ --pd
    ↪ 172.16.5.198:2379 --log-file restore-concurrency.log --concurrency 1024
```

性能调优后的结果如下表所示（保持数据大小不变）：

- 恢复耗时：total take(s) 从 961.37 减少到 443.49
- 恢复吞吐：avg speed(MB/s) 从 367.42 提升到 796.47
- 单个 TiKV 实例的吞吐：avg speed(MB/s)/tikv_count 从 91.8 提升到 199.1
- 单个 TiKV 实例的平均恢复速度：total size(MB)/(split time + restore time)/tikv_count 从 87.4 提升到 162.3

11.6.3.3.5 将单表数据备份到本地磁盘（推荐测试环境试用）

使用 br backup 命令，将单表数据 --db batchmark --table order_line 备份到指定的本地磁盘路径 local
↪ :///home/tidb/backup_local 下。

前置要求

- 备份前的准备工作。
- 各个 TiKV 节点有单独的磁盘用来存放 backupSST 数据。
- backup_endpoint 节点有单独的磁盘用来存放备份的 backupmeta 文件。
- TiKV 和 backup_endpoint 节点需要有相同的备份目录，例如 /home/tidb/backup_local。

部署拓扑

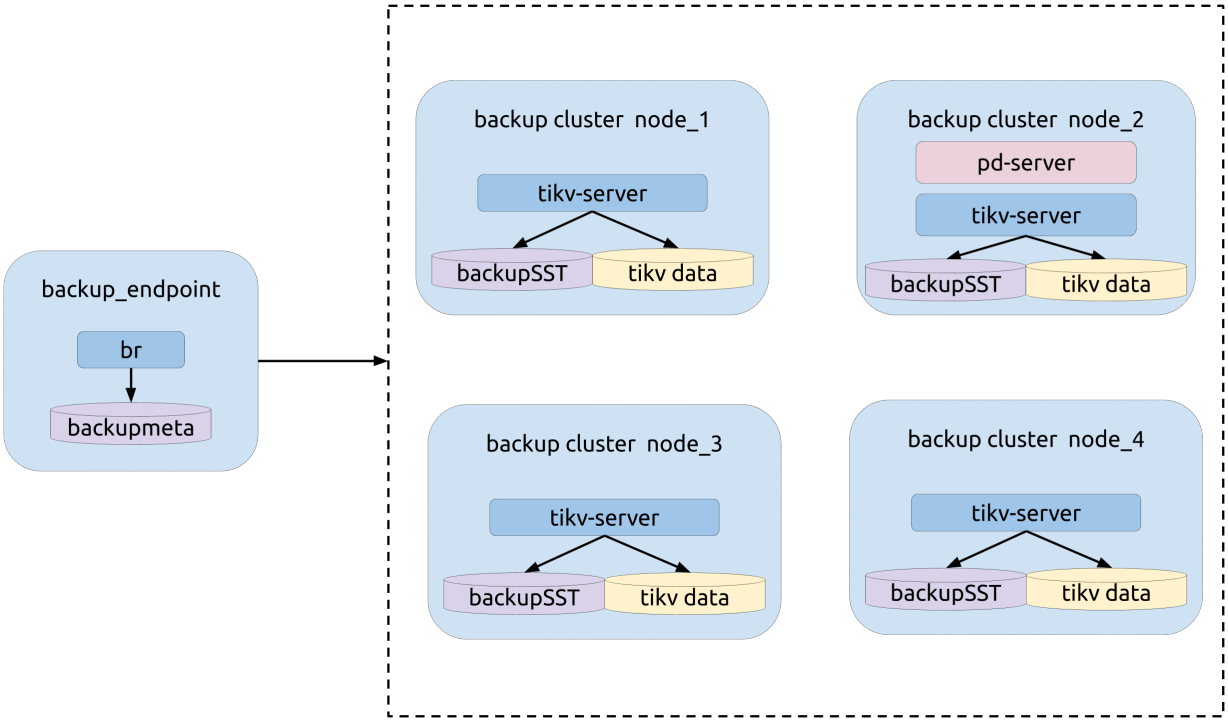


图 155: img

运行备份

备份前在 TiDB 里通过 `admin checksum table order_line` 获得备份的目标表 `--db batchmark --table order_line` 的统计信息。统计信息示例如下：

```
+-----------+------------+--------------------+-----------+-------------+
| Db_name   | Table_name | Checksum_crc64_xor | Total_kvs | Total_bytes |
+-----------+------------+--------------------+-----------+-------------+
| batchmark | order_line | 10912722838344822475 | 5659888624 | 370385538778 |
+-----------+------------+--------------------+-----------+-------------+
1 row in set (5 min 47.59 sec)
```

图 156: img

运行 `br backup` 命令：

```
bin/br backup table \
    --db batchmark \
    --table order_line \
    -s local:///home/tidb/backup_local/ \
    --pd ${PD_ADDR}:2379 \
    --log-file backup_local.log
```

运行备份时，参考备份过程中的运行指标对相关指标进行监控，以了解备份状态。

结果解读

使用 BR 前已设置日志的存放路径。从该路径下存放的日志获取此次备份的相关统计信息。在日志中搜关键字 “summary”，可以看到以下信息：

```
["Table backup summary: total backup ranges: 4, total success: 4, total failed: 0, total take(s):
    ↪  551.31, total kv: 5659888624, total size(MB): 353227.18, avg speed(MB/s): 640.71"] ["
    ↪ backup total regions"=6795] ["backup checksum"=6m33.962719217s] ["backup fast checksum
    ↪ "=22.995552ms]
```

以上日志信息中包含以下内容：

- 备份耗时：`total take(s): 551.31`
- 数据大小：`total size(MB): 353227.18`
- 备份吞吐：`avg speed(MB/s): 640.71`
- 校验耗时：`take=6m33.962719217s`

根据上表数据可以计算得到单个 TiKV 实例的吞吐：`avg speed(MB/s)/tikv_count` = 160。

11.6.3.3.6 从本地磁盘恢复备份数据（推荐测试环境试用）

使用 `br restore` 命令，将一份完整的备份数据恢复到一个离线集群。暂不支持恢复到在线集群。

前置要求

- 恢复前的准备工作。
- 集群中没有与备份数据相同的库表。目前 BR 不支持 table route。
- 集群中各个 TiKV 节点有单独的磁盘用来存放要恢复的 backupSST 数据。
- `restore_endpoint` 节点有单独的磁盘用来存放要恢复的 `backupmeta` 文件。
- 集群中 TiKV 和 `restore_endpoint` 节点需要有相同的备份目录，例如 `/home/tidb/backup_local/`。

如果备份数据存放在本地磁盘，那么需要执行以下的步骤：

1. 汇总所有 backupSST 文件到一个统一的目录下。
2. 将汇总后的 backupSST 文件到复制到集群的所有 TiKV 节点下。
3. 将 `backupmeta` 文件复制到到 `restore endpoint` 节点下。

部署拓扑

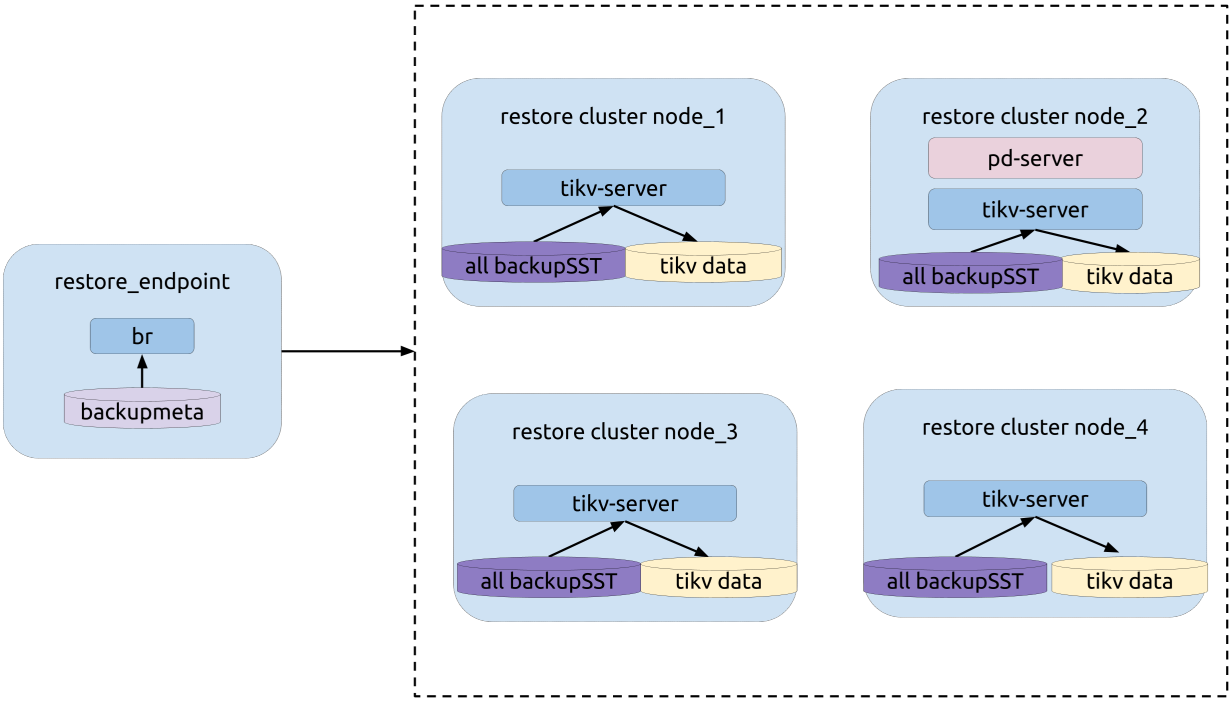


图 157: img

运行恢复

运行 `br restore` 命令：

```
bin/br restore table --db batchmark --table order_line -s local:///home/tidb/backup_local/ --pd
    ↪ 172.16.5.198:2379 --log-file restore_local.log
```

运行恢复时，参考恢复过程中的运行指标对相关指标进行监控，以了解恢复状态。

结果解读

使用 BR 前已设置日志的存放路径。从该日志中可以获取此次恢复的相关统计信息。在日志中搜关键字“summary”，可以看到以下信息：

```
["Table Restore summary: total restore tables: 1, total success: 1, total failed: 0, total take(s
    ↪ ): 908.42, total kv: 5659888624, total size(MB): 353227.18, avg speed(MB/s): 388.84"] ["
    ↪ restore files"=9263] ["restore ranges"=6888] ["split region"=58.7885518s] ["restore
    ↪ checksum"=6m19.349067937s]
```

以上日志信息中包含以下内容：

- 恢复耗时：`total take(s): 908.42`
- 数据大小：`total size(MB): 353227.18`
- 恢复吞吐：`avg speed(MB/s): 388.84`
- Region Split 耗时：`take=58.7885518s`

- 校验耗时：`take=6m19.349067937s`

根据上表数据可以计算得到：

- 单个 TiKV 实例的吞吐：`avg speed(MB/s)/tikv_count = 97.2`
- 单个 TiKV 实例的平均恢复速度：`total size(MB)/(split time + restore time)/tikv_count = 92.4`

11.6.3.4 备份过程中的异常处理

本节介绍如何处理备份过程中出现的常见错误。

11.6.3.4.1 备份日志中出现 key locked Error

日志中的错误消息：`log - ["backup occur kv error"][error="{\"KvError\":{\"locked\":`

如果在备份过程中遇到 key 被锁住，目前 BR 会尝试清锁。少量报错不会影响备份的正确性。

11.6.3.4.2 备份失败

日志中的错误消息：`log - Error: msg:"Io(Custom { kind: AlreadyExists, error: \"[5_5359_42_123_default .sst] is already exists in /dir/backup_local/\" })"`

若备份失败并出现以上错误消息，采取以下其中一种操作后再重新备份：

- 更换备份数据目录。例如将 `/dir/backup-2020-01-01/` 改为 `/dir/backup_local/`。
- 删除所有 TiKV 和 BR 节点的备份目录。

11.6.4 外部存储

Backup & Restore (BR)、TiDB Lightning 和 Dumpling 皆支持在本地文件系统和 Amazon S3 上读写数据；另外 BR 亦支持 Google Cloud Storage (GCS)。通过传入不同 URL scheme 到 BR 的 `--storage` (`-s`) 参数、TiDB Lightning 的 `-d` 参数及 Dumpling 中的 `--output` (`-o`) 参数，可以区分不同的存储方式。

11.6.4.1 Scheme

TiDB 迁移工具支持以下存储服务：

服务	Scheme	示例
本地文件系统（分布在各节点上）	local	`local:///path/to/dest/`
Amazon S3 及其他兼容 S3 的服务	s3	`s3://bucket-name/prefix/of/dest/`
GCS	gcs, gs	`gcs://bucket-name/prefix/of/dest/`
不写入任何存储（仅作为基准测试）	noop	`noop://`

11.6.4.2 URL 参数

S3 和 GCS 等云存储有时需要额外的连接配置，你可以为这类配置指定参数。例如：

- 用 Dumpling 导出数据到 S3

```
./dumpling -u root -h 127.0.0.1 -P 3306 -B mydb -F 256MiB \
    -o 's3://my-bucket/sql-backup?region=us-west-2'
```

- 用 TiDB Lightning 从 S3 导入数据

```
./tidb-lightning --tidb-port=4000 --pd-urls=127.0.0.1:2379 --backend=local --sorted-kv-dir=/
    ↪ tmp/sorted-kvs \
    -d 's3://my-bucket/sql-backup?region=us-west-2'
```

- 用 TiDB Lightning 从 S3 导入数据（使用路径类型的请求模式）。如果你使用的是 TiDB v4.0.11 及以下版本，需要设置 `force-path-style=true` 后才能使用路径类型的请求模式。

```
./tidb-lightning --tidb-port=4000 --pd-urls=127.0.0.1:2379 --backend=local --sorted-kv-dir=/
    ↪ tmp/sorted-kvs \
    -d 's3://my-bucket/sql-backup?force-path-style=true&endpoint=http://10.154.10.132:8088'
```

- 用 BR 备份到 GCS

```
./br backup full -u 127.0.0.1:2379 \
    -s 'gcs://bucket-name/prefix'
```

11.6.4.2.1 S3 的 URL 参数

URL 参数	描述
access-key	访问密钥
secret-access-key	secret 访问密钥
region	Amazon S3 服务区域（默认为 us-east-1）
use-accelerate-endpoint	是否在 Amazon S3 上使用加速端点（默认为 false）
endpoint	S3 兼容服务自定义端点的 URL（例如 https://s3.example.com/）
force-path-style	使用 path-style，而不是 virtual-hosted style（默认为 false）
storage-class	上传对象的存储类别（例如 STANDARD、STANDARD_IA）
sse	用于加密上传的服务器端加密算法（可以设置为空，AES256 或 aws:kms）
sse-kms-key-id	如果 sse 设置为 aws:kms，则使用该参数指定 KMS ID
acl	上传对象的 canned ACL（例如，private、authenticated-read）

注意：

不建议在存储 URL 中直接传递访问密钥和 secret 访问密钥，因为这些密钥是明文记录的。迁移工具尝试按照以下顺序从环境中推断这些密钥：

1. $AWS_ACCESS_KEY_ID 和 $AWS_SECRET_ACCESS_KEY 环境变量。

2. $AWS_ACCESS_KEY 和 $AWS_SECRET_KEY 环境变量。
3. 工具节点上的共享凭证文件，路径由 $AWS_SHARED_CREDENTIALS_FILE 环境变量指定。
4. 工具节点上的共享凭证文件，路径为 ~/.aws/credentials。
5. 当前 Amazon EC2 容器的 IAM 角色。
6. 当前 Amazon ECS 任务的 IAM 角色。

11.6.4.2.2 GCS 的 URL 参数

URL 参数	描述
credentials-file	迁移工具节点上的凭证 JSON 文件的路径
storage-class	上传对象的存储类别（例如 STANDARD、COLDLINE）
predefined-acl	上传对象的预定义 ACL（例如 private、project-private）

如果没有指定 credentials-file，迁移工具尝试按照以下顺序从环境中推断出凭证：

1. 工具节点上位于 $GOOGLE_APPLICATION_CREDENTIALS 环境变量所指定路径的文件内容。
2. 工具节点上位于 ~/.config/gcloud/application_default_credentials.json 的文件内容。
3. 在 GCE 或 GAE 中运行时，从元数据服务器中获取的凭证。

11.6.4.3 命令行参数

除了使用 URL 参数，BR 和 Dumpling 工具亦支持从命令行指定这些配置，例如：

```
./dumpling -u root -h 127.0.0.1 -P 3306 -B mydb -F 256MiB \
    -o 's3://my-bucket/sql-backup' \
    --s3.region 'us-west-2'
```

如果同时指定了 URL 参数和命令行参数，命令行参数会覆盖 URL 参数。

11.6.4.3.1 S3 的命令行参数

命令行参数	描述
--s3.region	Amazon S3 服务区域（默认为 us-east-1）
--s3.endpoint	S3 兼容服务自定义端点的 URL（例如 https://s3.example.com/）
--s3.storage-class	上传对象的存储类别（例如 STANDARD、STANDARD_IA）
--s3.sse	用于加密上传的服务器端加密算法（可以设置为空，AES256 或 aws:kms）
--s3.sse-kms-key-id	如果 --s3.sse 设置为 aws:kms，则使用该参数指定 KMS ID
--s3.acl	上传对象的 canned ACL（例如，private、authenticated-read）
--s3.provider	S3 兼容服务类型（支持 aws、alibaba、ceph、netease 或 other）

11.6.4.3.2 GCS 的命令行参数

命令行参数	描述
--gcs.credentials-file	迁移工具节点上的凭证 JSON 文件的路径
--gcs.storage-class	上传对象的存储类别（例如 STANDARD、COLDLINE）
--gcs.predefined-acl	上传对象的预定义 ACL（例如 private、project-private）

11.6.4.4 BR 向 TiKV 发送凭证

在默认情况下，使用 S3 和 GCS 存储时，BR 会将凭证发送到每个 TiKV 节点，以减少设置的复杂性。

但是，这个操作不适合云端环境，因为每个节点都有自己的角色和权限。在这种情况下，你需要用 --send-credentials-to-tikv=false（或简写为 -c=0）来禁止发送凭证：

```
./br backup full -c=0 -u pd-service:2379 -s 's3://bucket-name/prefix'
```

使用 SQL 进行备份恢复时，可加上 SEND_CREDENTIALS_TO_TIKV = FALSE 选项：

```
BACKUP DATABASE * TO 's3://bucket-name/prefix' SEND_CREDENTIALS_TO_TIKV = FALSE;
```

此参数不适用于 TiDB Lightning 和 Dumping，因为目前它们都是单机程序。

11.6.5 Backup & Restore 常见问题

本文列出了在使用 Backup & Restore (BR) 时，可能会遇到的问题及相应的解决方法。

如果遇到未包含在此文档且无法解决的问题，可以在 AskTUG 社区中提问。

11.6.5.1 恢复的时候，报错 could not read local://...:download sst failed，该如何处理？

在恢复的时候，每个节点都必须能够访问到所有的备份文件（SST files），默认情况下，假如使用 local storage，备份文件会分散在各个节点中，此时是无法直接恢复的，必须将每个 TiKV 节点的备份文件拷贝到其它所有 TiKV 节点才能恢复。

建议在备份的时候挂载一块 NFS 网盘作为备份盘，详见将单表数据备份到网络盘。

11.6.5.2 BR 备份时，对集群影响多大？

使用 sysbench 的 oltp_read_only 场景全速备份到非服务盘，对集群的影响依照表结构的不同，对集群 QPS 的影响在 15%~25% 之间。

如果需要控制备份带来的影响，可以使用 --ratelimit 参数限速。

11.6.5.3 BR 会备份系统表吗？在数据恢复的时候，这些系统表会冲突吗？

全量备份的时候会过滤掉系统库（information_schema，performance_schema，mysql）。参考备份原理。

因为这些系统库根本不可能存在于备份中，恢复的时候自然不可能发生冲突。

11.6.5.4 BR 遇到 Permission denied 或者 No such file or directory 错误，即使用 root 运行 BR 也无法解决，该如何处理？

需要确认 TiKV 是否有访问备份目录的权限。如果是备份，确认是否有写权限；如果是恢复，确认是否有读权限。

在进行备份操作时，如果使用本地磁盘或 NFS 作为存储介质，请确保执行 BR 的用户和启动 TiKV 的用户相同（如果 BR 和 TiKV 位于不同的机器，则需要用户的 UID 相同），否则很备份可能会出现该问题。

使用 root 运行 BR 仍旧有可能会因为磁盘权限而失败，因为备份文件 (SST) 的保存是由 TiKV 执行的。

> **注意：**
>
> 在恢复的时候也可能遇到同样的问题。
>
> 使用 BR 进行数据的恢复时，检验读权限的时机是在第一次读取 SST 文件时，考虑到执行 DDL 的耗时，这个时刻可能会离开始运行 BR 的时间很远。这样可能会出现等了很长时间之后遇到 Permission denied 错误失败的情况。
>
> 因此，最好在恢复前提前检查权限。

11.6.5.5 BR 遇到错误信息 `Io(Os...)`，该如何处理？

这类问题几乎都是 TiKV 在写盘的时候遇到的系统调用错误。例如遇到 `Io(Os { code: 13, kind: PermissionDenied...})` 或者 `Io(Os { code: 2, kind: NotFound...})` 这类错误信息，首先检查备份目录的挂载方式和文件系统，试试看备份到其它文件夹或者其它硬盘。

目前已知备份到 samba 搭建的网盘时可能会遇到 `Code: 22(invalid argument)` 错误。

11.6.5.6 BR 遇到错误信息 `rpc error: code = Unavailable desc =...`，该如何处理？

该问题一般是因为使用 BR 恢复数据的时候，恢复集群的性能不足导致的。可以从恢复集群的监控或者 TiKV 日志来辅助确认。

要解决这类问题，可以尝试扩大集群资源，以及调小恢复时的并发度 (concurrency)，打开限速 (ratelimit) 设置。

11.6.5.7 使用 local storage 的时候，BR 备份的文件会存在哪里？

在使用 local storage 的时候，会在运行 BR 的节点生成 `backupmeta`，在各个 Region 的 Leader 节点生成备份文件。

11.6.5.8 备份数据有多大，备份会有副本吗？

备份的时候仅仅在每个 Region 的 Leader 处生成该 Region 的备份文件。因此备份的大小等于数据大小，不会有多余的副本数据。所以最终的总大小大约是 TiKV 数据总量除以副本数。

但是假如想要从本地恢复数据，因为每个 TiKV 都必须要能访问到所有备份文件，在最终恢复的时候会有等同于恢复时 TiKV 节点数量的副本。

11.6.5.9 BR 恢复到 TiCDC / Drainer 的上游集群时，要注意些什么？

- BR 恢复的数据无法被同步到下游，因为 BR 直接导入 SST 文件，而下游集群目前没有办法获得上游的 SST 文件。
- 在 4.0.3 版本之前，BR 恢复时产生的 DDL jobs 还可能会让 TiCDC / Drainer 执行异常的 DDL。所以，如果一定要在 TiCDC / Drainer 的上游集群执行恢复，请将 BR 恢复的所有表加入 TiCDC / Drainer 的阻止名单。

TiCDC 可以通过配置项中的 `filter.rules` 项完成，Drainer 则可以通过`syncer.ignore-table` 完成。

11.6.5.10 BR 会备份表的 SHARD_ROW_ID_BITS 和 PRE_SPLIT_REGIONS 信息吗？恢复出来的表会有多个 Region 吗？

会的，BR 会备份表的`SHARD_ROW_ID_BITS 和 PRE_SPLIT_REGIONS` 信息，并恢复成多个 Region。

11.6.5.11 使用 BR 恢复备份数据后，SQL 查询报错 `region is unavailable`

如果 BR 备份的集群有 TiFlash，恢复时会将 TiFlash 信息存进 `TableInfo`。此时如果恢复的集群没有 TiFlash，则会报该错误。计划在未来版本中修复该错误。

11.6.5.12 BR 是否支持就地 (in-place) 全量恢复某个历史备份？

不支持。

11.6.5.13 在 Kubernetes 环境中如何使用 BR 进行增量备份？

可以使用 kubectl 执行 `kubectl -n ${namespace} get bk ${name}` 以获得上次 BR 备份 `commitTs` 字段，该字段的内容可作为 `--lastbackupts` 使用。

11.6.5.14 BR backupTS 如何转化成 Unix 时间?

BR backupTS 默认是在备份开始前，从 PD 获取到的最新时间戳。可以使用 `pd-ctl tso timestamp` 来解析该时间戳，以获得精确值，也可以通过 `backupTS >> 18` 来快速获取估计值。

11.6.5.15 BR 恢复存档后是否需要对表执行 ANALYZE 以更新 TiDB 在表和索引上留下的统计信息？

BR 不会备份统计信息（v4.0.9 除外）。所以在恢复存档后需要手动执行 `ANALYZE TABLE` 或等待 TiDB 自动进行 ANALYZE。

BR v4.0.9 备份统计信息使 BR 消耗过多内存，为保证备份过程正常，从 v4.0.10 开始默认关闭备份统计信息的功能。

如果不对表执行 ANALYZE，TiDB 会因统计信息不准确而选不中最优化的执行计划。如果查询性能不是重点关注项，可以忽略 ANALYZE。

11.7 TiDB Binlog

11.7.1 TiDB Binlog 简介

TiDB Binlog 是一个用于收集 TiDB 的 binlog，并提供准实时备份和同步功能的商业工具。

TiDB Binlog 支持以下功能场景：

- 数据同步：同步 TiDB 集群数据到其他数据库
- 实时备份和恢复：备份 TiDB 集群数据，同时可以用于 TiDB 集群故障时恢复

11.7.1.1 TiDB Binlog 整体架构

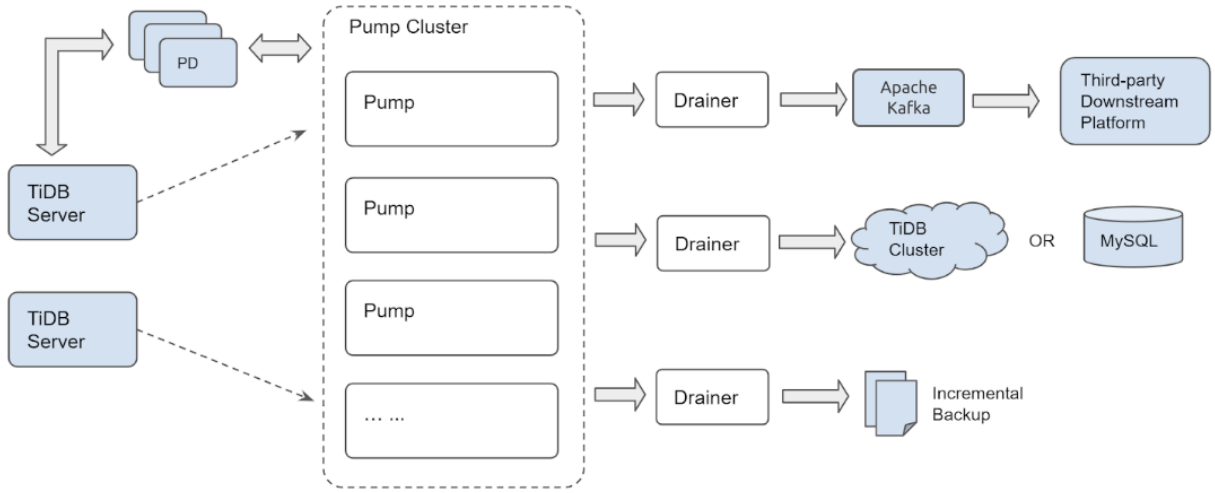


图 158: TiDB Binlog 架构

TiDB Binlog 集群主要分为 Pump 和 Drainer 两个组件，以及 binlogctl 工具：

11.7.1.1.1 Pump

Pump 用于实时记录 TiDB 产生的 Binlog，并将 Binlog 按照事务的提交时间进行排序，再提供给 Drainer 进行消费。

11.7.1.1.2 Drainer

Drainer 从各个 Pump 中收集 Binlog 进行归并，再将 Binlog 转化成 SQL 或者指定格式的数据，最终同步到下游。

11.7.1.1.3 binlogctl 工具

`binlogctl` 是一个 TiDB Binlog 配套的运维工具，具有如下功能：

- 获取 TiDB 集群当前的 TSO

- 查看 Pump/Drainer 状态
- 修改 Pump/Drainer 状态
- 暂停/下线 Pump/Drainer

11.7.1.2 主要特性

- 多个 Pump 形成一个集群，可以水平扩容。
- TiDB 通过内置的 Pump Client 将 Binlog 分发到各个 Pump。
- Pump 负责存储 Binlog，并将 Binlog 按顺序提供给 Drainer。
- Drainer 负责读取各个 Pump 的 Binlog，归并排序后发送到下游。
- Drainer 支持relay log 功能，通过 relay log 保证下游集群的一致性状态。

11.7.1.3 注意事项

- TiDB Binlog 与 TiDB v4.0.7 版本开始引入的以下特性不兼容，无法一起使用：
  - TiDB 系统变量tidb_enable_amend_pessimistic_txn：两个功能存在兼容性问题，一起使用会造成 TiDB Binlog 复制数据不一致的正确性问题。
- Drainer 支持将 Binlog 同步到 MySQL、 TiDB、 Kafka 或者本地文件。如果需要将 Binlog 同步到其他 Drainer 不支持的类型的系统中，可以设置 Drainer 将 Binlog 同步到 Kafka，然后根据 binlog consumer protocol 进行定制处理，参考Binlog Consumer Client 用户文档。
- 如果 TiDB Binlog 用于增量恢复，可以设置配置项 `db-type="file"`，Drainer 会将 binlog 转化为指定的 proto buffer 格式的数据，再写入到本地文件中。这样就可以使用Reparo 恢复增量数据。

  关于 `db-type` 的取值，应注意：
  - 如果 TiDB 版本 < 2.1.9，则 `db-type="pb"`。
  - 如果 TiDB 版本 > = 2.1.9，则 `db-type="file"` 或 `db-type="pb"`。
- 如果下游为 MySQL/TiDB，数据同步后可以使用sync-diff-inspector 进行数据校验。

11.7.2 TiDB Binlog 教程

本文档主要介绍如何使用 TiDB Binlog 将数据从 TiDB 推送到 MariaDB 实例。文中的 TiDB Binlog 集群包含 Pump 和 Drainer 的单个节点，TiDB 集群包含 TiDB、 TiKV 和 Placement Driver (PD) 各组件的单个节点。

希望上手实践 TiDB Binlog 工具的用户需要对TiDB 架构有一定的了解，最好有创建过 TiDB 集群的经验。该文档也有助于简单快速了解 TiDB Binlog 架构以及相关概念。

> **警告：**
>
> 该文档中部署 TiDB 的操作指导不适用于在生产或研发环境中部署 TiDB 的情况。

该文档假设用户使用的是现代 Linux 发行版本中的 x86-64。示例中使用的是 VMware 中运行的 CentOS 7 最小化安装。建议在一开始就进行清洁安装，以避免受现有环境中未知情况的影响。如果不想使用本地虚拟环境，也可以使用云服务启动 CentOS 7 VM。

11.7.2.1 TiDB Binlog 简介

TiDB Binlog 用于收集 TiDB 中二进制日志数据、提供实时数据备份和同步以及将 TiDB 集群的数据增量同步到下游。

TiDB Binlog 支持以下功能场景：

- 增量备份，将 TiDB 集群中的数据增量同步到另一个集群，或通过 Kafka 增量同步到选择的下游。
- 当使用 TiDB DM (Data Migration) 将数据从上游 MySQL 或者 MariaDB 迁移到 TiDB 集群时，可使用 TiDB Binlog 保持 TiDB 集群与其一个独立下游 MySQL 或 MariaDB 实例或集群同步。当 TiDB 集群上游数据迁移过程中出现问题，下游数据同步过程中可使用 TiDB Binlog 恢复数据到原先的状态。

更多信息参考TiDB Binlog Cluster 版本用户文档。

11.7.2.2 架构

TiDB Binlog 集群由 Pump 和 Drainer 两个组件组成。一个 Pump 集群中有若干个 Pump 节点。TiDB 实例连接到各个 Pump 节点并发送 binlog 数据到 Pump 节点。Pump 集群连接到 Drainer 节点，Drainer 将接收到的更新数据转换到某个特定下游（例如 Kafka、另一个 TiDB 集群或 MySQL 或 MariaDB Server）指定的正确格式。

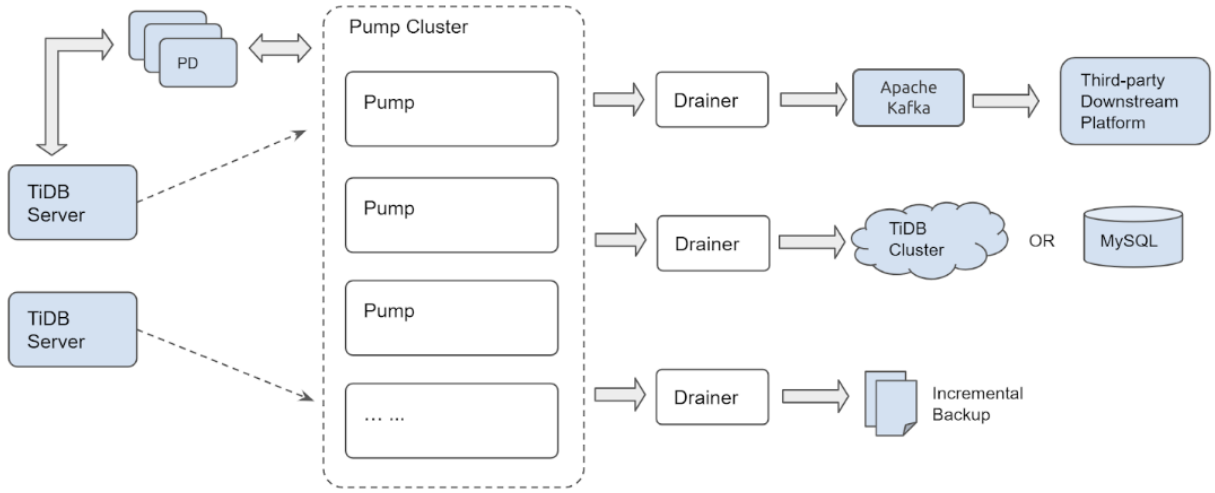


图 159: TiDB Binlog architecture

Pump 的集群架构能确保 TiDB 或 Pump 集群中有新的实例加入或退出时更新数据不会丢失。

11.7.2.3 安装

由于 RHEL/CentOS 7 的默认包装库中包括 MariaDB Server，本示例选择的是 MariaDB Server。后续除了安装服务器，也需要安装客户端。安装指令如下：

```
sudo yum install -y mariadb-server
```

```
curl -LO https://download.pingcap.org/tidb-latest-linux-amd64.tar.gz | tar xzf - &&
cd tidb-latest-linux-amd64
```

预期输出：

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  368M  100  368M    0     0  8394k      0  0:00:44  0:00:44 --:--:-- 11.1M
```

11.7.2.4 配置

通过执行以下步骤配置一个 TiDB 集群，该集群包括 pd-server、tikv-server 和 tidb-server 各组件的单个实例。

1. 填充配置文件：

```
printf > pd.toml %s\\n 'log-file="pd.log"' 'data-dir="pd.data"' &&
printf > tikv.toml %s\\n 'log-file="tikv.log"' '[storage]' 'data-dir="tikv.data"' '[pd]' '
    ↪ endpoints=["127.0.0.1:2379"]' '[rocksdb]' max-open-files=1024 '[raftdb]' max-open-
    ↪ files=1024 &&
printf > pump.toml %s\\n 'log-file="pump.log"' 'data-dir="pump.data"' 'addr="127.0.0.1:8250"
    ↪ ' 'advertise-addr="127.0.0.1:8250"' 'pd-urls="http://127.0.0.1:2379"' &&
printf > tidb.toml %s\\n 'store="tikv"' 'path="127.0.0.1:2379"' '[log.file]' 'filename="tidb
    ↪ .log"' '[binlog]' 'enable=true' &&
printf > drainer.toml %s\\n 'log-file="drainer.log"' '[syncer]' 'db-type="mysql"' '[syncer.
    ↪ to]' 'host="127.0.0.1"' 'user="root"' 'password=""' 'port=3306'
```

2. 查看配置细节：

```
for f in *.toml; do echo "$f:"; cat "$f"; echo; done
```

预期输出：

```
drainer.toml:
log-file="drainer.log"
[syncer]
db-type="mysql"
[syncer.to]
host="127.0.0.1"
user="root"
password=""
port=3306

pd.toml:
log-file="pd.log"
data-dir="pd.data"
```

```
pump.toml:
log-file="pump.log"
data-dir="pump.data"
addr="127.0.0.1:8250"
advertise-addr="127.0.0.1:8250"
pd-urls="http://127.0.0.1:2379"

tidb.toml:
store="tikv"
path="127.0.0.1:2379"
[log.file]
filename="tidb.log"
[binlog]
enable=true

tikv.toml:
log-file="tikv.log"
[storage]
data-dir="tikv.data"
[pd]
endpoints=["127.0.0.1:2379"]
[rocksdb]
max-open-files=1024
[raftdb]
max-open-files=1024
```

11.7.2.5 启动程序

现在可启动各个组件。推荐启动顺序依次为 Placement Driver (PD)、 TiKV、 Pump（TiDB 发送 binlog 日志必须连接 Pump 服务）、 TiDB。

1. 启动所有服务：

```
./bin/pd-server --config=pd.toml &>pd.out &
```

```
[1] 20935
```

```
./bin/tikv-server --config=tikv.toml &>tikv.out &
```

```
[2] 20944
```

```
./bin/pump --config=pump.toml &>pump.out &
```

```
[3] 21050
```

```
sleep 3 &&
./bin/tidb-server --config=tidb.toml &>tidb.out &
```

```
[4] 21058
```

2. 如果执行 jobs，可以看到后台正在运行的程序，列表如下：

```
jobs
```

```
[1]   Running                 ./bin/pd-server --config=pd.toml &>pd.out &
[2]   Running                 ./bin/tikv-server --config=tikv.toml &>tikv.out &
[3]-  Running                 ./bin/pump --config=pump.toml &>pump.out &
[4]+  Running                 ./bin/tidb-server --config=tidb.toml &>tidb.out &
```

如果有服务启动失败（例如出现 “Exit 1” 而不是 “Running”），尝试重启单个组件。

11.7.2.6 连接

按以上步骤操作后，TiDB 的 4 个组件开始运行。接下来可以使用以下 MariaDB 或 MySQL 命令行客户端，通过 4000 端口连接到 TiDB 服务：

```
mysql -h 127.0.0.1 -P 4000 -u root -e 'select tidb_version();'
```

预期输出：

```
*************************** 1. row ***************************
tidb_version(): Release Version: v3.0.0-beta.1-154-gd5afff70c
Git Commit Hash: d5afff70cdd825d5fab125c8e52e686cc5fb9a6e
Git Branch: master
UTC Build Time: 2019-04-24 03:10:00
GoVersion: go version go1.12 linux/amd64
Race Enabled: false
TiKV Min Version: 2.1.0-alpha.1-ff3dd160846b7d1aed9079c389fc188f7f5ea13e
Check Table Before Drop: false
```

连接后 TiDB 集群已开始运行，pump 读取集群中的 binlog 数据，并在其数据目录中将 binlog 数据存储为 relay log。下一步是启动一个可供 drainer 写入的 MariaDB Server。

1. 启动 drainer：

```
sudo systemctl start mariadb &&
./bin/drainer --config=drainer.toml &>drainer.out &
```

如果你的操作系统更易于安装 MySQL，只需保证监听 3306 端口。另外，可使用密码为空的 “root” 用户连接到 MySQL，或调整 drainer.toml 连接到 MySQL。

```
mysql -h 127.0.0.1 -P 3306 -u root
```

预期输出：

```
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 20
Server version: 5.5.60-MariaDB MariaDB Server

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MariaDB [(none)]>
```

```
show databases;
```

预期输出：

```
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| test               |
| tidb_binlog        |
+--------------------+
5 rows in set (0.01 sec)
```

如下表格是包含 checkpoint 表格的 tidb_binlog 数据库。drainer 使用 checkpoint 表格，记录 TiDB 集群中的 binlog 已经更新到了哪个位置。

```
use tidb_binlog;
```

```
Database changed
```

```
select * from checkpoint;
```

```
+--------------------+-------------------------------------------+
| clusterID          | checkPoint                                |
+--------------------+-------------------------------------------+
| 6678715361817107733 | {"commitTS":407637466476445697,"ts-map":{}} |
+--------------------+-------------------------------------------+
1 row in set (0.00 sec)
```

打开另一个连接到 TiDB 的客户端，创建一个表格并插入几行数据。建议在 GNU Screen 软件中操作，从而同时打开多个客户端。

```
mysql -h 127.0.0.1 -P 4000 --prompt='TiDB [\d]> ' -u root
```

```
create database tidbtest;
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
use tidbtest;
```

```
Database changed
```

```
create table t1 (id int unsigned not null AUTO_INCREMENT primary key);
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
insert into t1 () values (),(),(),(),();
```

```
Query OK, 5 rows affected (0.01 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
select * from t1;
```

```
+----+
| id |
+----+
|  1 |
|  2 |
|  3 |
|  4 |
|  5 |
+----+
5 rows in set (0.00 sec)
```

切换回 MariaDB 客户端可看到新的数据库、新的表格和最近插入的行数据。

```
use tidbtest;
```

```
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
```

```
show tables;
```

```
+--------------------+
| Tables_in_tidbtest |
+--------------------+
```

```
| t1                 |
+--------------------+
1 row in set (0.00 sec)
```

```
select * from t1;
```

```
+----+
| id |
+----+
|  1 |
|  2 |
|  3 |
|  4 |
|  5 |
+----+
5 rows in set (0.00 sec)
```

可看到查询 MariaDB 时插入到 TiDB 相同的行数据，表明 TiDB Binlog 安装成功。

11.7.2.7 binlogctl

加入到集群的 Pump 和 Drainer 的数据存储在 Placement Driver (PD) 中。binlogctl 可用于查询和修改状态信息。更多信息请参考binlogctl guide。

使用 `binlogctl` 查看集群中 Pump 和 Drainer 的当前状态：

```
./bin/binlogctl -cmd drainers
```

```
[2019/04/11 17:44:10.861 -04:00] [INFO] [nodes.go:47] ["query node"] [type=drainer] [node="{
    ↪ NodeID: localhost.localdomain:8249, Addr: 192.168.236.128:8249, State: online,
    ↪ MaxCommitTS: 407638907719778305, UpdateTime: 2019-04-11 17:44:10 -0400 EDT}"]
```

```
./bin/binlogctl -cmd pumps
```

```
[2019/04/11 17:44:13.904 -04:00] [INFO] [nodes.go:47] ["query node"] [type=pump] [node="{NodeID:
    ↪ localhost.localdomain:8250, Addr: 192.168.236.128:8250, State: online, MaxCommitTS:
    ↪ 407638914024079361, UpdateTime: 2019-04-11 17:44:13 -0400 EDT}"]
```

如果结束 Drainer 进程，集群会改进程设置 “已暂停（即集群等待 Drainer 重新加入）” 的状态。

```
pkill drainer &&
./bin/binlogctl -cmd drainers
```

预期输出：

```
[2019/04/11 17:44:22.640 -04:00] [INFO] [nodes.go:47] ["query node"] [type=drainer] [node="{
    ↪ NodeID: localhost.localdomain:8249, Addr: 192.168.236.128:8249, State: paused,
    ↪ MaxCommitTS: 407638915597467649, UpdateTime: 2019-04-11 17:44:18 -0400 EDT}"]
```

使用 binlogctl 的“NodeIDs”可控制单个对应节点。在该情况下，Drainer 的节点 ID 是“localhost.localdomain:8249”，Pump 的节点 ID 是“localhost.localdomain:8250”。

本文档中的 binlogctl 主要用于集群重启。如果在 TiDB 集群中终止并尝试重启所有的进程，由于 Pump 无法连接 Drainer 且认为 Drainer 依旧“在线”，Pump 会拒绝启动。这里的进程并不包括下游的 MySQL 或 MariaDB 或 Drainer。

以下有三个方案可解决上述问题：

- 使用 binlogctl 停止 Drainer，而不是结束进程：

```
./bin/binlogctl --pd-urls=http://127.0.0.1:2379 --cmd=drainers &&
./bin/binlogctl --pd-urls=http://127.0.0.1:2379 --cmd=pause-drainer --node-id=localhost.
    ↪ localdomain:8249
```

- 在启动 Pump 之前先启动 Drainer。
- 在启动 PD 之后但在启动 Drainer 和 Pump 之前，使用 binlogctl 更新已暂定 Drainer 的状态。

```
./bin/binlogctl --pd-urls=http://127.0.0.1:2379 --cmd=update-drainer --node-id=localhost.
    ↪ localdomain:8249 --state=paused
```

11.7.2.8 清理

在 shell 终端里可启动创建集群的所有进程（pd-server、tikv-server、pump、tidb-server、drainer）。可通过在 shell 终端中执行 pkill -P $$ 停止 TiDB 集群服务和 TiDB Binlog 进程。按一定的顺序停止这些进程有利于留出足够的时间彻底关闭每个组件。

```
for p in tidb-server drainer pump tikv-server pd-server; do pkill "$p"; sleep 1; done
```

预期输出：

```
[4]-  Done                    ./bin/tidb-server --config=tidb.toml &>tidb.out
[5]+  Done                    ./bin/drainer --config=drainer.toml &>drainer.out
[3]+  Done                    ./bin/pump --config=pump.toml &>pump.out
[2]+  Done                    ./bin/tikv-server --config=tikv.toml &>tikv.out
[1]+  Done                    ./bin/pd-server --config=pd.toml &>pd.out
```

如果需要所有服务退出后重启集群，可以使用一开始启动服务的命令。如以上binlogctl 部分所述，需要先启动 Drainer 再启动 Pump，最后启动 tidb-server。

```
./bin/pd-server --config=pd.toml &>pd.out &
./bin/tikv-server --config=tikv.toml &>tikv.out &
./bin/drainer --config=drainer.toml &>drainer.out &
sleep 3
./bin/pump --config=pump.toml &>pump.out &
sleep 3
./bin/tidb-server --config=tidb.toml &>tidb.out &
```

如果有组件启动失败，请尝试单独重启该组件。

11.7.2.9 总结

本文档介绍了如何通过设置 TiDB Binlog，使用单个 Pump 和 Drainer 组成的集群同步 TiDB 集群数据到下游的 MariaDB。可以发现，TiDB Binlog 是用于获取处理 TiDB 集群中更新数据的综合性平台工具。

在更稳健的开发、测试或生产部署环境中，可以使用多个 TiDB 服务以实现高可用性和扩展性。使用多个 Pump 实例可以避免 Pump 集群中的问题影响发送到 TiDB 实例的应用流量。或者可以使用增加的 Drainer 实例同步数据到不同的下游或实现数据增量备份。

11.7.3 TiDB Binlog 集群部署

11.7.3.1 服务器要求

Pump 和 Drainer 均可部署和运行在 Intel x86-64 架构的 64 位通用硬件服务器平台上。在开发、测试和生产环境下，对服务器硬件配置的要求和建议如下：

服务	部署数量	CPU	磁盘	内存
Pump	3	8 核 +	SSD, 200 GB+	16G
Drainer	1	8 核 +	SAS, 100 GB+（如果输出 binlog 为本地文件，磁盘大小视保留数据天数而定）	16G

11.7.3.2 使用 TiUP 部署 TiDB Binlog

推荐使用 TiUP 部署 TiDB Binlog，详细部署方式参考TiUP 部署 TiDB Binlog。

11.7.3.3 使用 TiDB Ansible 部署 TiDB Binlog

11.7.3.3.1 第 1 步：下载 TiDB Ansible

1. 以 TiDB 用户登录中控机并进入 `/home/tidb` 目录。以下为 TiDB Ansible 分支与 TiDB 版本的对应关系，版本选择可咨询官方 info@pingcap.com。

TiDB Ansible 分支	TiDB 版本	备注
master	master 版本	包含最新特性，每日更新。

2. 使用以下命令从 GitHub TiDB Ansible 项目上下载 TiDB Ansible 相应分支，默认的文件夹名称为 `tidb-ansible`。

    - 下载 master 版本：

```
git clone https://github.com/pingcap/tidb-ansible.git
```

11.7.3.3.2 第 2 步：部署 Pump

1. 修改 tidb-ansible/inventory.ini 文件

    1. 设置 `enable_binlog = True`，表示 TiDB 集群开启 binlog。

```
## binlog trigger
enable_binlog = True
```

2. 为 pump_servers 主机组添加部署机器 IP。

```
## Binlog Part
[pump_servers]
172.16.10.72
172.16.10.73
172.16.10.74
```

默认 Pump 保留 7 天数据，如需修改可修改 tidb-ansible/conf/pump.yml（TiDB 3.0.0~3.0.2 版本中为 tidb-ansible/conf/pump-cluster.yml）文件中 gc 变量值，并取消注释。

```
global:
  # an integer value to control the expiry date of the binlog data, which indicates for
      ↪  how long (in days) the binlog data would be stored
  # must be bigger than 0
  # gc: 7
```

请确保部署目录有足够空间存储 binlog，详见调整部署目录，也可为 Pump 设置单独的部署目录。

```
## Binlog Part
[pump_servers]
pump1 ansible_host=172.16.10.72 deploy_dir=/data1/pump
pump2 ansible_host=172.16.10.73 deploy_dir=/data2/pump
pump3 ansible_host=172.16.10.74 deploy_dir=/data3/pump
```

2. 部署并启动含 Pump 组件的 TiDB 集群

参照上文配置完 inventory.ini 文件后，从以下两种方式中选择一种进行部署。

方式一：在已有的 TiDB 集群上增加 Pump 组件，需按以下步骤逐步进行。

1. 部署 pump_servers 和 node_exporters

```
ansible-playbook deploy.yml --tags=pump -l ${pump1_ip},${pump2_ip},[${alias1_name},${
    ↪ alias2_name}]
```

> 注意：
> 以上命令中，逗号后不要加空格，否则会报错。

2. 启动 pump_servers

```
ansible-playbook start.yml --tags=pump
```

3. 更新并重启 tidb_servers

```
ansible-playbook rolling_update.yml --tags=tidb
```

4. 更新监控信息

```
ansible-playbook rolling_update_monitor.yml --tags=prometheus
```

方式二：从零开始部署含 Pump 组件的 TiDB 集群

使用 TiDB Ansible 部署 TiDB 集群，方法参考使用 TiDB Ansible 部署 TiDB 集群。

3. 查看 Pump 服务状态

使用 binlogctl 查看 Pump 服务状态，pd-urls 参数请替换为集群 PD 地址，结果 State 为 online 表示 Pump 启动成功。

```
cd /home/tidb/tidb-ansible &&
resources/bin/binlogctl -pd-urls=http://172.16.10.72:2379 -cmd pumps
```

```
INFO[0000] pump: {NodeID: ip-172-16-10-72:8250, Addr: 172.16.10.72:8250, State: online,
    ↪ MaxCommitTS: 403051525690884099, UpdateTime: 2018-12-25 14:23:37 +0800 CST}
INFO[0000] pump: {NodeID: ip-172-16-10-73:8250, Addr: 172.16.10.73:8250, State: online,
    ↪ MaxCommitTS: 403051525703991299, UpdateTime: 2018-12-25 14:23:36 +0800 CST}
INFO[0000] pump: {NodeID: ip-172-16-10-74:8250, Addr: 172.16.10.74:8250, State: online,
    ↪ MaxCommitTS: 403051525717360643, UpdateTime: 2018-12-25 14:23:35 +0800 CST}
```

11.7.3.3.3 第 3 步：部署 Drainer

1. 获取 initial_commit_ts 的值

Drainer 初次启动时需要获取 initial_commit_ts 这个时间戳信息。

- 如果从最近的时间点开始同步，initial_commit_ts 使用 -1 即可。
- 如果下游为 MySQL 或 TiDB，为了保证数据的完整性，需要进行全量数据的备份与恢复。此时 initial_commit_ts 的值必须是全量备份的时间戳。

  如果使用 mydumper 进行全量备份，可以在导出目录中找到 metadata 文件，其中的 Pos 字段值即全量备份的时间戳。metadata 文件示例如下：

```
Started dump at: 2019-12-30 13:25:41
SHOW MASTER STATUS:
        Log: tidb-binlog
        Pos: 413580274257362947
        GTID:

Finished dump at: 2019-12-30 13:25:41
```

2. 修改 `tidb-ansible/inventory.ini` 文件

为 `drainer_servers` 主机组添加部署机器 IP，initial_commit_ts 请设置为获取的 initial_commit_ts，仅用于 Drainer 第一次启动。

- 以下游为 MySQL 为例，别名为 `drainer_mysql`。

```
[drainer_servers]
drainer_mysql ansible_host=172.16.10.71 initial_commit_ts="402899541671542785"
```

- 以下游为 file 为例，别名为 drainer_file。

```
[drainer_servers]
drainer_file ansible_host=172.16.10.71 initial_commit_ts="402899541671542785"
```

3. 修改配置文件

- 以下游为 MySQL 为例

```
cd /home/tidb/tidb-ansible/conf &&
cp drainer.toml drainer_mysql_drainer.toml &&
vi drainer_mysql_drainer.toml
```

> **注意：**
> 配置文件名命名规则为 别名_drainer.toml，否则部署时无法找到自定义配置文件。

db-type 设置为 “mysql”，配置下游 MySQL 信息。

```
[syncer]
# downstream storage, equal to --dest-db-type
# Valid values are "mysql", "file", "tidb", "kafka".
db-type = "mysql"

# the downstream MySQL protocol database
[syncer.to]
host = "172.16.10.72"
user = "root"
password = "123456"
port = 3306
```

- 以下游为增量备份文件为例

```
cd /home/tidb/tidb-ansible/conf &&
cp drainer.toml drainer_file_drainer.toml &&
vi drainer_file_drainer.toml
```

db-type 设置为 “file”。

```
[syncer]
# downstream storage, equal to --dest-db-type
# Valid values are "mysql", "file", "tidb", "kafka".
db-type = "file"

# Uncomment this if you want to use "file" as "db-type".
```

```
[syncer.to]
# default data directory: "{{ deploy_dir }}/data.drainer"
dir = "data.drainer"
```

4. 部署 Drainer

```
ansible-playbook deploy_drainer.yml
```

5. 启动 Drainer

```
ansible-playbook start_drainer.yml
```

11.7.3.4 使用 Binary 部署 TiDB Binlog

11.7.3.4.1 下载官方 Binary

```
wget https://download.pingcap.org/tidb-{version}-linux-amd64.tar.gz &&
wget https://download.pingcap.org/tidb-{version}-linux-amd64.sha256
```

检查文件完整性，返回 ok 则正确：

```
sha256sum -c tidb-{version}-linux-amd64.sha256
```

对于 v2.1.0 GA 及以上版本，Pump 和 Drainer 已经包含在 TiDB 的下载包中，其他版本需要单独下载 Pump 和 Drainer:

```
wget https://download.pingcap.org/tidb-binlog-latest-linux-amd64.tar.gz &&
wget https://download.pingcap.org/tidb-binlog-latest-linux-amd64.sha256
```

检查文件完整性，返回 ok 则正确：

```
sha256sum -c tidb-binlog-latest-linux-amd64.sha256
```

11.7.3.4.2 使用样例

假设有三个 PD，一个 TiDB，另外有两台机器用于部署 Pump，一台机器用于部署 Drainer。各个节点信息如下：

```
TiDB="192.168.0.10"
PD1="192.168.0.16"
PD2="192.168.0.15"
PD3="192.168.0.14"
Pump="192.168.0.11"
Pump="192.168.0.12"
Drainer="192.168.0.13"
```

下面以此为例，说明 Pump/Drainer 的使用。

1. 使用 binary 部署 Pump

    - Pump 命令行参数说明（以在 “192.168.0.11” 上部署为例）

```
Usage of Pump:
-L string
    日志输出信息等级设置：debug，info，warn，error，fatal（默认 "info"）
-V
    打印版本信息
-addr string
    Pump 提供服务的 RPC 地址(-addr="192.168.0.11:8250")
-advertise-addr string
    Pump 对外提供服务的 RPC 地址(-advertise-addr="192.168.0.11:8250")
-config string
    配置文件路径，如果你指定了配置文件，Pump 会首先读取配置文件的配置；
    如果对应的配置在命令行参数里面也存在，Pump
        ↪ 就会使用命令行参数的配置来覆盖配置文件里的配置。
-data-dir string
    Pump 数据存储位置路径
-gc int
    Pump 只保留多少天以内的数据（默认 7）
-heartbeat-interval int
    Pump 向 PD 发送心跳间隔（单位 秒）
-log-file string
    log 文件路径
-log-rotate string
    log 文件切换频率，hour/day
-metrics-addr string
    Prometheus Pushgateway 地址，不设置则禁止上报监控信息
-metrics-interval int
    监控信息上报频率（默认 15，单位 秒）
-node-id string
    Pump 节点的唯一识别 ID，如果不指定，程序会根据主机名和监听端口自动生成
-pd-urls string
    PD 集群节点的地址 (-pd-urls="http://192.168.0.16:2379,http://192.168.0.15:2379,http
        ↪ ://192.168.0.14:2379")
-fake-binlog-interval int
    Pump 节点生成 fake binlog 的频率（默认 3，单位 秒）
```

    - Pump 配置文件（以在 “192.168.0.11” 上部署为例）

```
# Pump Configuration

# Pump 绑定的地址
addr = "192.168.0.11:8250"

# Pump 对外提供服务的地址
```

```
advertise-addr = "192.168.0.11:8250"

# Pump 只保留多少天以内的数据（默认 7）
gc = 7

# Pump 数据存储位置路径
data-dir = "data.pump"

# Pump 向 PD 发送心跳的间隔（单位 秒）
heartbeat-interval = 2

# PD 集群节点的地址（英文逗号分割，中间不加空格）
pd-urls = "http://192.168.0.16:2379,http://192.168.0.15:2379,http://192.168.0.14:2379"

# [security]
# 如无特殊安全设置需要，该部分一般都注解掉
# 包含与集群连接的受信任 SSL CA 列表的文件路径
# ssl-ca = "/path/to/ca.pem"
# 包含与集群连接的 PEM 形式的 X509 certificate 的路径
# ssl-cert = "/path/to/drainer.pem"
# 包含与集群链接的 PEM 形式的 X509 key 的路径
# ssl-key = "/path/to/drainer-key.pem"

# [storage]
# 设置为 true（默认值）来保证可靠性，确保 binlog 数据刷新到磁盘
# sync-log = true

# 当可用磁盘容量小于该设置值时，Pump 将停止写入数据
# 42 MB -> 42000000, 42 mib -> 44040192
# default: 10 gib
# stop-write-at-available-space = "10 gib"

# Pump 内嵌的 LSM DB 设置，除非对该部分很了解，否则一般注解掉
# [storage.kv]
# block-cache-capacity = 8388608
# block-restart-interval = 16
# block-size = 4096
# compaction-L0-trigger = 8
# compaction-table-size = 67108864
# compaction-total-size = 536870912
# compaction-total-size-multiplier = 8.0
# write-buffer = 67108864
# write-L0-pause-trigger = 24
# write-L0-slowdown-trigger = 17
```

- 启动示例

```
./bin/pump -config pump.toml
```

如果命令行参数与配置文件中的参数重合，则使用命令行设置的参数的值。

2. 使用 binary 部署 Drainer

- Drainer 命令行参数说明（以在 “192.168.0.13” 上部署为例）

```
Usage of Drainer
-L string
    日志输出信息等级设置：debug，info，warn，error，fatal（默认 "info"）
-V
    打印版本信息
-addr string
    Drainer 提供服务的地址(-addr="192.168.0.13:8249")
-c int
    同步下游的并发数，该值设置越高同步的吞吐性能越好（default 1）
-cache-binlog-count int
    缓存中的 binlog 数目限制（默认 8）
    如果上游的单个 binlog 较大导致 Drainer 出现 OOM 时，可尝试调小该值减少内存使用
-config string
    配置文件路径，Drainer 会首先读取配置文件的配置；
    如果对应的配置在命令行参数里面也存在，Drainer
        ↪ 就会使用命令行参数的配置来覆盖配置文件里面的配置
-data-dir string
    Drainer 数据存储位置路径（默认 "data.drainer"）
-dest-db-type string
    Drainer 下游服务类型（默认为 mysql，支持 tidb、kafka、file）
-detect-interval int
    向 PD 查询在线 Pump 的时间间隔（默认 10，单位 秒）
-disable-detect
    是否禁用冲突监测
-disable-dispatch
    是否禁用拆分单个 binlog 的 SQL 的功能，如果设置为 true，则每个 binlog
    按顺序依次还原成单个事务进行同步（下游服务类型为 MySQL，该项设置为 False）
-ignore-schemas string
    db 过滤列表（默认 "INFORMATION_SCHEMA,PERFORMANCE_SCHEMA,mysql,test"），
    不支持对 ignore schemas 的 table 进行 rename DDL 操作
-initial-commit-ts（默认为 `-1`）
    如果 Drainer 没有相关的断点信息，可以通过该项来设置相关的断点信息
    该参数值为 `-1` 时，Drainer 会自动从 PD 获取一个最新的时间戳
-log-file string
    log 文件路径
-log-rotate string
    log 文件切换频率，hour/day
```

```
-metrics-addr string
    Prometheus Pushgateway 地址，不设置则禁止上报监控信息
-metrics-interval int
    监控信息上报频率（默认 15，单位：秒）
-node-id string
    drainer 节点的唯一识别 ID，如果不指定，程序会根据主机名和监听端口自动生成
-pd-urls string
    PD 集群节点的地址 (-pd-urls="http://192.168.0.16:2379,http://192.168.0.15:2379,http
        ↪ ://192.168.0.14:2379")
-safe-mode
    是否开启安全模式使得下游 MySQL/TiDB 可被重复写入
    即将 insert 语句换为 replace 语句，将 update 语句拆分为 delete + replace 语句
-txn-batch int
    输出到下游数据库一个事务的 SQL 数量（默认 1）
```

- Drainer 配置文件（以在 “192.168.0.13” 上部署为例）

```
# Drainer Configuration.

# Drainer 提供服务的地址("192.168.0.13:8249")
addr = "192.168.0.13:8249"

# Drainer 对外提供服务的地址
advertise-addr = "192.168.0.13:8249"

# 向 PD 查询在线 Pump 的时间间隔（默认 10，单位 秒）
detect-interval = 10

# Drainer 数据存储位置路径（默认 "data.drainer"）
data-dir = "data.drainer"

# PD 集群节点的地址（英文逗号分割，中间不加空格）
pd-urls = "http://192.168.0.16:2379,http://192.168.0.15:2379,http://192.168.0.14:2379"

# log 文件路径
log-file = "drainer.log"

# Drainer 从 Pump 获取 binlog 时对数据进行压缩，值可以为 "gzip"，如果不配置则不进行压缩
# compressor = "gzip"

# [security]
# 如无特殊安全设置需要，该部分一般都注解掉
# 包含与集群连接的受信任 SSL CA 列表的文件路径
# ssl-ca = "/path/to/ca.pem"
# 包含与集群连接的 PEM 形式的 X509 certificate 的路径
# ssl-cert = "/path/to/pump.pem"
```

```
# 包含与集群链接的 PEM 形式的 X509 key 的路径
# ssl-key = "/path/to/pump-key.pem"

# Syncer Configuration
[syncer]
# 如果设置了该项，会使用该 sql-mode 解析 DDL 语句，此时如果下游是 MySQL 或 TiDB 则
# 下游的 sql-mode 也会被设置为该值
# sql-mode = "STRICT_TRANS_TABLES,NO_ENGINE_SUBSTITUTION"

# 输出到下游数据库一个事务的 SQL 语句数量（默认 20）
txn-batch = 20

# 同步下游的并发数，该值设置越高同步的吞吐性能越好（默认 16）
worker-count = 16

# 是否禁用拆分单个 binlog 的 SQL 的功能，如果设置为 true，则按照每个 binlog
# 顺序依次还原成单个事务进行同步（下游服务类型为 MySQL，该项设置为 False）
disable-dispatch = false

# safe mode 会使写下游 MySQL/TiDB 可被重复写入
# 会用 replace 替换 insert 语句，用 delete + replace 替换 update 语句
safe-mode = false

# Drainer 下游服务类型（默认为 mysql）
# 参数有效值为 "mysql", "tidb", "file", "kafka"
db-type = "mysql"

# 事务的 commit ts 若在该列表中，则该事务将被过滤，不会同步至下游
ignore-txn-commit-ts = []

# db 过滤列表（默认 "INFORMATION_SCHEMA,PERFORMANCE_SCHEMA,mysql,test"），
# 不支持对 ignore schemas 的 table 进行 rename DDL 操作
ignore-schemas = "INFORMATION_SCHEMA,PERFORMANCE_SCHEMA,mysql"

# replicate-do-db 配置的优先级高于 replicate-do-table。如果配置了相同的库名，
    ↪ 支持使用正则表达式进行配置。
# 以 '~' 开始声明使用正则表达式

# replicate-do-db = ["~^b.*","s1"]

# [syncer.relay]
# 保存 relay log 的目录，空值表示不开启。
# 只有下游是 TiDB 或 MySQL 时该配置才生效。
# log-dir = ""
# 每个文件的大小上限
```

```
# max-file-size = 10485760

# [[syncer.replicate-do-table]]
# db-name ="test"
# tbl-name = "log"

# [[syncer.replicate-do-table]]
# db-name ="test"
# tbl-name = "~^a.*"

# 忽略同步某些表
# [[syncer.ignore-table]]
# db-name = "test"
# tbl-name = "log"

# db-type 设置为 mysql 时，下游数据库服务器参数
[syncer.to]
host = "192.168.0.13"
user = "root"
# 如果你不想在配置文件中写明文密码，则可以使用 `./binlogctl -cmd encrypt -text string`
    ↪ 生成加密的密码
# 如果配置了 encrypted_password 且非空，那么配置的 password 不生效。encrypted_password
    ↪ 和 password 无法同时生效。
password = ""
encrypted_password = ""
port = 3306

[syncer.to.checkpoint]
# 当 checkpoint type 是 mysql 或 tidb 时可以开启该选项，以改变保存 checkpoint 的数据库
# schema = "tidb_binlog"
# 目前只支持 mysql 或者 tidb 类型。可以去掉注释来控制 checkpoint 保存的位置。
# db-type 默认的 checkpoint 保存方式是:
# mysql/tidb -> 对应的下游 mysql/tidb
# file/kafka -> file in `data-dir`
# type = "mysql"
# host = "127.0.0.1"
# user = "root"
# password = ""
# 使用 `./binlogctl -cmd encrypt -text string` 加密的密码
# encrypted_password 非空时 password 会被忽略
# encrypted_password = ""
# port = 3306

# db-type 设置为 file 时，存放 binlog 文件的目录
# [syncer.to]
```

```
# dir = "data.drainer"

# db-type 设置为 kafka 时，Kafka 相关配置
# [syncer.to]
# kafka-addrs 和 zookeeper-addrs 只需要一个，两者都有时程序会优先用 zookeeper 中的
    ↪ kafka 地址
# zookeeper-addrs = "127.0.0.1:2181"
# kafka-addrs = "127.0.0.1:9092"
# kafka-version = "0.8.2.0"
# kafka-max-messages = 1024

# 保存 binlog 数据的 Kafka 集群的 topic 名称，默认值为 <cluster-id>_obinlog
# 如果运行多个 Drainer 同步数据到同一个 Kafka 集群，每个 Drainer 的 topic-name
    ↪ 需要设置不同的名称
# topic-name = ""
```

- 启动示例

> **注意：**
> 如果下游为 MySQL/TiDB，为了保证数据的完整性，在 Drainer 初次启动前需要获取 `initial-commit-ts` 的值，并进行全量数据的备份与恢复。详细信息参见部署 Drainer。

初次启动时使用参数 `initial-commit-ts`，命令如下：

```
./bin/drainer -config drainer.toml -initial-commit-ts {initial-commit-ts}
```

如果命令行参数与配置文件中的参数重合，则使用命令行设置的参数的值。

注意：

- 在运行 TiDB 时，需要保证至少一个 Pump 正常运行。
- 通过给 TiDB 增加启动参数 `enable-binlog` 来开启 binlog 服务。尽量保证同一集群的所有 TiDB 都开启了 binlog 服务，否则在同步数据时可能会导致上下游数据不一致。如果要临时运行一个不开启 binlog 服务的 TiDB 实例，需要在 TiDB 的配置文件中设置 `run_ddl= false`。
- Drainer 不支持对 ignore schemas（在过滤列表中的 schemas）的 table 进行 rename DDL 操作。
- 在已有的 TiDB 集群中启动 Drainer，一般需要全量备份并且获取快照时间戳，然后导入全量备份，最后启动 Drainer 从对应的快照时间戳开始同步增量数据。
- 下游使用 MySQL 或 TiDB 时应当保证上下游数据库的 sql_mode 具有一致性，即下游数据库同步每条 SQL 语句时的 sql_mode 应当与上游数据库执行该条 SQL 语句时的 sql_mode 保持一致。可以在上下游分别执行 `select @@sql_mode;` 进行查询和比对。
- 如果存在上游 TiDB 能运行但下游 MySQL 不支持的 DDL 语句时（例如下游 MySQL 使用 InnoDB 引擎时同步语句 `CREATE TABLE t1(a INT)ROW_FORMAT=FIXED;`），Drainer 也会同步失败，此时可以在 Drainer 配置中跳过该事务，同时在下游手动执行兼容的语句，详见跳过事务。

11.7.4 TiDB Binlog 集群运维

本文首先介绍 Pump 和 Drainer 的状态及启动、退出的内部处理流程，然后说明如何通过 binlogctl 工具或者直接在 TiDB 执行 SQL 操作来管理 binlog 集群，最后的 FAQ 部分会介绍一些常见问题以及处理方法。

11.7.4.1 Pump/Drainer 的状态

Pump/Drainer 中状态的定义：

- online：正常运行中
- pausing：暂停中
- paused：已暂停
- closing：下线中
- offline：已下线

这些状态由 Pump/Drainer 服务自身进行维护，并定时将状态信息更新到 PD 中。

11.7.4.2 Pump/Drainer 的启动、退出流程

11.7.4.2.1 Pump

- 启动：Pump 启动时会通知所有 online 状态的 Drainer，如果通知成功，则 Pump 将状态设置为 online，否则 Pump 将报错，然后将状态设置为 paused 并退出进程。
- 退出：Pump 进程正常退出前要选择进入暂停或者下线状态；非正常退出（kill -9、进程 panic、宕机）都依然保持 online 状态。
  - 暂停：使用 kill（非 kill -9）、Ctrl+C 或者使用 binlogctl 的 pause-pump 命令都可以暂停 Pump。接收到暂停指令后，Pump 会变更状态为 pausing，并停止接受 binlog 的写请求，也停止向 Drainer 提供 binlog 数据。安全退出所有线程后，更新状态为 paused 然后退出进程。
  - 下线：仅在使用 binlogctl 的 offline-pump 命令的情况下才会下线 Pump。接收到下线指令后，Pump 会变更状态为 closing，并停止接受 binlog 的写请求。Pump 继续向 Drainer 提供 binlog，等待所有 binlog 数据都被 Drainer 消费后再将状态设置为 offline 并退出进程。

11.7.4.2.2 Drainer

- 启动：Drainer 启动时将状态设置为 online，并尝试从所有非 offline 状态的 Pump 获取 binlog，如果获取 binlog 失败，会不断尝试重新获取。
- 退出：Drainer 进程正常退出前要选择进入暂停或者下线状态；非正常退出（kill -9 、进程 panic、宕机）都依然保持 online 状态。
  - 暂停：使用 kill（非 kill -9）、Ctrl+C 或者使用 binlogctl 的 pause-drainer 命令都可以暂停 Drainer。接收到指令后，Drainer 会变更状态为 pausing，并停止从 Pump 获取 binlog。安全退出所有线程后，更新状态为 paused 然后退出进程。

- 下线：仅在使用 binlogctl 的 offline-drainer 命令的情况下才会下线 Drainer。接收到下线指令后，Drainer 变更状态为 closing，并停止从 Pump 获取 binlog。安全退出所有线程后，更新状态为 offline 然后退出进程。

关于 Pump/Drainer 暂停、下线、状态查询、状态修改等具体的操作方法，参考如下 binlogctl 工具的使用方法介绍。

11.7.4.3 使用 binlogctl 工具管理 Pump/Drainer

binlogctl 支持如下这些功能：

- 查看 Pump/Drainer 状态
- 暂停/下线 Pump/Drainer
- Pump/Drainer 异常状态处理

详细的介绍和使用方法请参考binlogctl 工具。

11.7.4.4 使用 TiDB SQL 管理 Pump/Drainer

要查看和管理 binlog 相关的状态，可在 TiDB 中执行相应的 SQL 语句。

- 查看 TiDB 是否开启 binlog，0 代表关闭，1 代表开启

```
show variables like "log_bin";
```

```
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| log_bin       |  0    |
+---------------+-------+
```

- 查看 Pump/Drainer 状态

```
show pump status;
```

```
+--------|----------------|--------|--------------------|---------------------|
| NodeID |     Address    | State  |   Max_Commit_Ts    |    Update_Time      |
+--------|----------------|--------|--------------------|---------------------|
| pump1  | 127.0.0.1:8250 | Online | 408553768673342237 | 2019-05-01 00:00:01 |
+--------|----------------|--------|--------------------|---------------------|
| pump2  | 127.0.0.2:8250 | Online | 408553768673342335 | 2019-05-01 00:00:02 |
+--------|----------------|--------|--------------------|---------------------|
```

```
show drainer status;
```

```
+----------|----------------|--------|--------------------|---------------------|
|  NodeID  |     Address    | State  |   Max_Commit_Ts    |    Update_Time      |
+----------|----------------|--------|--------------------|---------------------|
| drainer1 | 127.0.0.3:8249 | Online | 408553768673342532 | 2019-05-01 00:00:03 |
+----------|----------------|--------|--------------------|---------------------|
| drainer2 | 127.0.0.4:8249 | Online | 408553768673345531 | 2019-05-01 00:00:04 |
+----------|----------------|--------|--------------------|---------------------|
```

- 异常情况下修改 Pump/Drainer 状态

```
change pump to node_state ='paused' for node_id 'pump1';
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
change drainer to node_state ='paused' for node_id 'drainer1';
```

```
Query OK, 0 rows affected (0.01 sec)
```

该 SQL 的功能和 binlogctl 中的 update-pump 和 update-drainer 命令的功能一样，因此也只有在 Pump/Drainer 异常的情况下使用。

注意：

1. 查看 binlog 开启状态以及 Pump/Drainer 状态的功能在 TiDB v2.1.7 及以上版本中支持。
2. 修改 Pump/Drainer 状态的功能在 TiDB v3.0.0-rc.1 及以上版本中支持。该功能只修改 PD 中存储的 Pump/Drainer 状态，如果需要暂停/下线节点，仍然需要使用 `binlogctl`。

11.7.5 TiDB Binlog 配置说明

本文档介绍 TiDB Binlog 的各项配置说明。

11.7.5.1 Pump

本节介绍 Pump 的配置项。可以在 Pump Configuration 中查看完整的 Pump 配置文件示例。

11.7.5.1.1 addr

- HTTP API 的监听地址，格式为 `host:port`。
- 默认：`"127.0.0.1:8250"`

11.7.5.1.2 advertise-addr

- 对外可访问的 HTTP API 地址。这个地址会被注册到 PD，格式为 `host:port`。
- 默认：与 `addr` 的配置相同。

11.7.5.1.3 socket

- HTTP API 监听的 Unix socket 地址。
- 默认：`""`

11.7.5.1.4 pd-urls

- 由逗号分隔的 PD URL 列表。如果指定了多个地址，PD 客户端在连接一个地址时出错时会自动尝试连接另一个地址。
- 默认：`"http://127.0.0.1:2379"`

11.7.5.1.5 data-dir

- 本地存放 binlog 及其索引的目录。
- 默认：`"data.pump"`

11.7.5.1.6 heartbeat-interval

- 心跳间隔，即每隔指定秒数向 PD 汇报最新的状态。
- 默认：2

11.7.5.1.7 gen-binlog-interval

- 指定写入 fake binlog 的间隔秒数。
- 默认：3

11.7.5.1.8 gc

- 指定 binlog 可在本地存储的天数（整型）。超过指定天数的 binlog 会被自动删除。
- 默认：7

11.7.5.1.9 log-file

- 保存日志文件的路径。如果为空，日志不会被保存。
- 默认：`""`

11.7.5.1.10 log-level

- Log 等级。
- 默认：`"info"`

11.7.5.1.11 node-id

- Pump 节点的 ID，用于在集群中识别这个进程。
- 默认：主机名:端口号，例如 `node-1:8250`。

11.7.5.1.12 security

以下是与安全相关的配置项。

ssl-ca

- 包含可信 SSL 证书或 CA 证书列表的文件路径，例如 `/path/to/ca.pem`。
- 默认：`""`

ssl-cert

- 包含 PEM 格式编码的 X509 证书文件路径，例如 `/path/to/pump.pem`。
- 默认：`""`

ssl-key

- 包含 PEM 格式编码的 X509 Key 文件路径，例如 `/path/to/pump-key.pem`。
- 默认：`""`

11.7.5.1.13 storage

以下是与存储相关的配置项。

sync-log

- 指定是否在每次批量写入 binlog 后使用 `fsync` 以确保数据安全。
- 默认：`true`

kv_chan_cap

- 在 Pump 接收写入请求时会先将请求放入一个缓冲区，该项指定缓冲区能存放的请求数量。
- 默认：1048576（即 2 的 20 次方）

slow_write_threshold

- 写入单个 binlog 的耗时超过该项设定的秒数就被认为是慢写入，并输出一条包含 "take a long time to write binlog" 的日志。
- 默认：1

stop-write-at-available-space

- 可用存储空间低于指定值时不再接收 binlog 写入请求。可以用例如 900 MB、5 GB、12 GiB 的格式指定空间大小。如果集群中 Pump 节点多于一个，那么在某个 Pump 节点因为空间不足而拒绝写入时，TiDB 端会自动写入到其他 Pump 节点。
- 默认：10 GiB

kv

目前 Pump 的存储是基于 GoLevelDB 实现的。storage 下还有一个 kv 子分组可以用于调整 GoLevelDB 的配置，支持的配置项包括：

- block-cache-capacity
- block-restart-interval
- block-size
- compaction-L0-trigger
- compaction-table-size
- compaction-total-size
- compaction-total-size-multiplier
- write-buffer
- write-L0-pause-trigger
- write-L0-slowdown-trigger

配置具体含义可在 GoLevelDB 文档中查看。

11.7.5.2 Drainer

本节介绍 Drainer 的配置项。可以在 Drainer Configuration 中查看完整的配置文件示例。

11.7.5.2.1 addr

- HTTP API 的监听的地址，格式为 host:port。
- 默认："127.0.0.1:8249"

11.7.5.2.2 advertise-addr

- 对外可访问的 HTTP API 地址，这个地址会被注册到 PD，格式为 host:port。
- 默认：设定成与 addr 相同的配置

11.7.5.2.3 log-file

- 日志的文件保存路径。如果为空，日志不会被保存。
- 默认：""

11.7.5.2.4 log-level

- Log 等级。
- 默认："info"

11.7.5.2.5 node-id

- Drainer 节点 ID，用于在集群中识别这个进程。
- 默认：主机名:端口号，例如 node-1:8249。

11.7.5.2.6 data-dir

- 用于存放 Drainer 运行中需要保存的文件的目录。
- 默认："data.drainer"

11.7.5.2.7 detect-interval

- 每隔指定秒数从 PD 更新一次 Pumps 信息，以获取节点加入或离开等事件。
- 默认：5

11.7.5.2.8 pd-urls

- 由逗号分隔的 PD URL 列表。如果指定了多个地址，PD 客户端在连接一个地址出错时会自动尝试连接另一个地址。
- 默认："http://127.0.0.1:2379"

11.7.5.2.9 initial-commit-ts

- 指定从哪个事务提交时间点（事务的 commit ts）之后开始同步。这个配置仅适用于初次开始同步的 Drainer 节点。如果下游已经有 checkpoint 存在，则会根据 checkpoint 里记录的时间进行同步。
- commit ts（即 commit timestamp）是 TiDB 事务的提交时间点。该时间点是从 PD 获取的全局唯一递增的时间戳，作为当前事务的唯一 ID。典型的 initial-commit-ts 配置可以通过以下方式获得：
  - BR 备份的元信息（即 backupmeta）中记录的 backup TS
  - Dumpling 备份的元信息（即 metadata）中记录的 Pos
  - PD Control 中 tso 命令返回的结果
- 默认：-1。Drainer 会从 PD 得到一个最新的 timestamp 作为初始时间。即从当前的时间点开始同步。

11.7.5.2.10 synced-check-time

- 通过 HTTP API 访问 `/status` 路径可以查询 Drainer 同步的状态。`synced-check-time` 指定距离上次成功同步的时间超过多少分钟可以认为是 `synced`，即同步完成。
- 默认：5

11.7.5.2.11 compressor

- 指定 Pump 与 Drainer 间的数据传输所用的压缩算法。目前仅支持一种算法，即 `gzip`。
- 默认：""，表示不压缩。

11.7.5.2.12 security

以下是与 Drainer 安全相关的配置。

ssl-ca

- 包含可信 SSL 证书或 CA 证书列表的文件路径，例如 `/path/to/ca.pem`。
- 默认：""

ssl-cert

- 包含 PEM 格式编码的 X509 证书文件路径，例如 `/path/to/drainer.pem`。
- 默认：""

ssl-key

- 包含 PEM 格式编码的 X509 Key 文件路径，例如 `/path/to/drainer-key.pem`。
- 默认：""

11.7.5.2.13 syncer

syncer 分组包含一些与同步下游相关的配置项。

db-type

下游类型，目前支持的选项有：

- `mysql`
- `tidb`
- `kafka`
- `file`

默认：`mysql`

sql-mode

- 当下游为 mysql/tidb 类型时，可以指定 SQL mode，如果超过一个 mode，则用逗号隔开。
- 默认：""

ignore-txn-commit-ts

- 同步时，该项所指定的 commit timestamp 的 binlog 会被忽略，例如 [416815754209656834, 421349811963822081] 。
- 默认：[]

ignore-schemas

- 同步时忽略指定数据库的变更。如果超过一个需忽略的数据库，则用逗号隔开。如果一个 binlog 中的变更全部被过滤掉，则忽略整个 binlog。
- 默认："INFORMATION_SCHEMA,PERFORMANCE_SCHEMA,mysql"

ignore-table

同步时忽略指定表格的变更。在 toml 中可以用以下方式指定多个需忽略的表格：

```
[[syncer.ignore-table]]
db-name = "test"
tbl-name = "log"

[[syncer.ignore-table]]
db-name = "test"
tbl-name = "audit"
```

如果一个 binlog 中的变更全部被过滤掉，则忽略整个 binlog。

默认：[]

replicate-do-db

- 指定要同步的数据库，例如 [db1, db2]。
- 默认：[]

replicate-do-table

指定要同步的表格，示例如下：

```
[[syncer.replicate-do-table]]
db-name ="test"
tbl-name = "log"

[[syncer.replicate-do-table]]
db-name ="test"
tbl-name = "~^a.*"
```

默认：[]

txn-batch

- 当下游为 mysql/tidb 类型时，会将 DML 分批执行。这个配置可以用于设置每个事务中包含多少个 DML。
- 默认：20

worker-count

- 当下游为 mysql/tidb 类型时，会并发执行 DML，worker-count 可以指定并发数。
- 默认：16

disable-dispatch

- 关掉并发，强制将 worker-count 置为 1。
- 默认：false

safe-mode

如果打开 Safe mode，Drainer 会对同步的变更作以下修改，使其变成可重入的操作：

- Insert 变为 Replace Into
- Update 变为 Delete 和 Replace Into

默认：false

11.7.5.2.14 syncer.to

不同类型的下游配置都在 syncer.to 分组。以下按配置类型进行介绍。

mysql/tidb

用于连接下游数据库的配置项：

- host：如果没有设置，会尝试检查环境变量 MYSQL_HOST，默认值为 "localhost"。
- port：如果没有设置，会尝试检查环境变量 MYSQL_PORT，默认值为 3306。
- user：如果没有设置，会尝试检查环境变量 MYSQL_USER，默认值为 "root"。
- password：如果没有设置，会尝试检查环境变量 MYSQL_PSWD，默认值为 ""。

file

- dir：指定用于保存 binlog 的目录。如果不指定该项，则使用 data-dir。

kafka

当下游为 kafka 时，有效的配置包括：

- zookeeper-addrs
- kafka-addrs
- kafka-version
- kafka-max-messages
- topic-name

11.7.5.2.15 syncer.to.checkpoint

以下是 `syncer.to.checkpoint` 相关的配置项。

11.7.5.2.16 type

- 指定用哪种方式保存同步进度。
- 目前支持的选项：`mysql` 和 `tidb`
- 默认：与下游类型相同。例如 `file` 类型的下游进度保存在本地文件系统，`mysql` 类型的下游进度保存在下游数据库。当明确指定要使用 `mysql` 或 `tidb` 保存同步进度时，需要指定以下配置项：
  - `schema`：默认为 `"tidb_binlog"`。

> **注意：**
> 在同个 TiDB 集群中部署多个 Drainer 时，需要为每个 Drainer 节点指定不同的 checkpoint schema，否则两个实例的同步进度会互相覆盖。

  - `host`
  - `user`
  - `password`
  - `port`

11.7.5.3 TiDB Binlog 配置说明

本文档介绍 TiDB Binlog 的各项配置说明。

11.7.5.3.1 Pump

本节介绍 Pump 的配置项。可以在 Pump Configuration 中查看完整的 Pump 配置文件示例。

addr

- HTTP API 的监听地址，格式为 `host:port`。
- 默认：`"127.0.0.1:8250"`

advertise-addr

- 对外可访问的 HTTP API 地址。这个地址会被注册到 PD，格式为 `host:port`。
- 默认：与 `addr` 的配置相同。

socket

- HTTP API 监听的 Unix socket 地址。
- 默认：`""`

pd-urls

- 由逗号分隔的 PD URL 列表。如果指定了多个地址，PD 客户端在连接一个地址时出错时会自动尝试连接另一个地址。
- 默认：`"http://127.0.0.1:2379"`

data-dir

- 本地存放 binlog 及其索引的目录。
- 默认：`"data.pump"`

heartbeat-interval

- 心跳间隔，即每隔指定秒数向 PD 汇报最新的状态。
- 默认：2

gen-binlog-interval

- 指定写入 fake binlog 的间隔秒数。
- 默认：3

gc

- 指定 binlog 可在本地存储的天数（整型）。超过指定天数的 binlog 会被自动删除。
- 默认：7

log-file

- 保存日志文件的路径。如果为空，日志不会被保存。
- 默认：`""`

log-level

- Log 等级。
- 默认：`"info"`

node-id

- Pump 节点的 ID，用于在集群中识别这个进程。
- 默认：主机名:端口号，例如 node-1:8250。

security

以下是与安全相关的配置项。

ssl-ca

- 包含可信 SSL 证书或 CA 证书列表的文件路径，例如 `/path/to/ca.pem`。
- 默认：`""`

ssl-cert

- 包含 PEM 格式编码的 X509 证书文件路径，例如 `/path/to/pump.pem`。
- 默认：`""`

ssl-key

- 包含 PEM 格式编码的 X509 Key 文件路径，例如 `/path/to/pump-key.pem`。
- 默认：`""`

storage

以下是与存储相关的配置项。

sync-log

- 指定是否在每次批量写入 binlog 后使用 `fsync` 以确保数据安全。
- 默认：`true`

kv_chan_cap

- 在 Pump 接收写入请求时会先将请求放入一个缓冲区，该项指定缓冲区能存放的请求数量。
- 默认：1048576（即 2 的 20 次方）

slow_write_threshold

- 写入单个 binlog 的耗时超过该项设定的秒数就被认为是慢写入，并输出一条包含 `"take a long time to write binlog"` 的日志。
- 默认：1

stop-write-at-available-space

- 可用存储空间低于指定值时不再接收 binlog 写入请求。可以用例如 `900 MB`、`5 GB`、`12 GiB` 的格式指定空间大小。如果集群中 Pump 节点多于一个，那么在某个 Pump 节点因为空间不足而拒绝写入时，TiDB 端会自动写入到其他 Pump 节点。
- 默认：`10 GiB`

kv

目前 Pump 的存储是基于 GoLevelDB 实现的。`storage` 下还有一个 kv 子分组可以用于调整 GoLevelDB 的配置，支持的配置项包括：

- block-cache-capacity

- block-restart-interval
- block-size
- compaction-L0-trigger
- compaction-table-size
- compaction-total-size
- compaction-total-size-multiplier
- write-buffer
- write-L0-pause-trigger
- write-L0-slowdown-trigger

配置具体含义可在 GoLevelDB 文档中查看。

11.7.5.3.2 Drainer

本节介绍 Drainer 的配置项。可以在 Drainer Configuration 中查看完整的配置文件示例。

addr

- HTTP API 的监听的地址，格式为 `host:port`。
- 默认：`"127.0.0.1:8249"`

advertise-addr

- 对外可访问的 HTTP API 地址，这个地址会被注册到 PD，格式为 `host:port`。
- 默认：设定成与 `addr` 相同的配置

log-file

- 日志的文件保存路径。如果为空，日志不会被保存。
- 默认：`""`

log-level

- Log 等级。
- 默认：`"info"`

node-id

- Drainer 节点 ID，用于在集群中识别这个进程。
- 默认：主机名:端口号，例如 `node-1:8249`。

data-dir

- 用于存放 Drainer 运行中需要保存的文件的目录。
- 默认：`"data.drainer"`

detect-interval

- 每隔指定秒数从 PD 更新一次 Pumps 信息，以获取节点加入或离开等事件。
- 默认：5

pd-urls

- 由逗号分隔的 PD URL 列表。如果指定了多个地址，PD 客户端在连接一个地址出错时会自动尝试连接另一个地址。
- 默认："http://127.0.0.1:2379"

initial-commit-ts

- 指定从哪个事务提交时间点（事务的 commit ts）之后开始同步。这个配置仅适用于初次开始同步的 Drainer 节点。如果下游已经有 checkpoint 存在，则会根据 checkpoint 里记录的时间进行同步。
- commit ts（即 commit timestamp）是 TiDB 事务的提交时间点。该时间点是从 PD 获取的全局唯一递增的时间戳，作为当前事务的唯一 ID。典型的 initial-commit-ts 配置可以通过以下方式获得：
  - BR 备份的元信息（即 backupmeta）中记录的 backup TS
  - Dumpling 备份的元信息（即 metadata）中记录的 Pos
  - PD Control 中 tso 命令返回的结果
- 默认：-1。Drainer 会从 PD 得到一个最新的 timestamp 作为初始时间。即从当前的时间点开始同步。

synced-check-time

- 通过 HTTP API 访问 /status 路径可以查询 Drainer 同步的状态。synced-check-time 指定距离上次成功同步的时间超过多少分钟可以认为是 synced，即同步完成。
- 默认：5

compressor

- 指定 Pump 与 Drainer 间的数据传输所用的压缩算法。目前仅支持一种算法，即 gzip。
- 默认：""，表示不压缩。

security

以下是与 Drainer 安全相关的配置。

ssl-ca

- 包含可信 SSL 证书或 CA 证书列表的文件路径，例如 /path/to/ca.pem。
- 默认：""

ssl-cert

- 包含 PEM 格式编码的 X509 证书文件路径，例如 /path/to/drainer.pem。

- 默认：""

ssl-key

- 包含 PEM 格式编码的 X509 Key 文件路径，例如 /path/to/drainer-key.pem。
- 默认：""

syncer

syncer 分组包含一些与同步下游相关的配置项。

db-type

下游类型，目前支持的选项有：

- mysql
- tidb
- kafka
- file

默认：mysql

sql-mode

- 当下游为 mysql/tidb 类型时，可以指定 SQL mode，如果超过一个 mode，则用逗号隔开。
- 默认：""

ignore-txn-commit-ts

- 同步时，该项所指定的 commit timestamp 的 binlog 会被忽略，例如 [416815754209656834，421349811963822081] ↪ 。
- 默认：[]

ignore-schemas

- 同步时忽略指定数据库的变更。如果超过一个需忽略的数据库，则用逗号隔开。如果一个 binlog 中的变更全部被过滤掉，则忽略整个 binlog。
- 默认："INFORMATION_SCHEMA,PERFORMANCE_SCHEMA,mysql"

ignore-table

同步时忽略指定表格的变更。在 toml 中可以用以下方式指定多个需忽略的表格：

```
[[syncer.ignore-table]]
db-name = "test"
tbl-name = "log"

[[syncer.ignore-table]]
db-name = "test"
tbl-name = "audit"
```

如果一个 binlog 中的变更全部被过滤掉，则忽略整个 binlog。

默认：[]

replicate-do-db

- 指定要同步的数据库，例如 [db1, db2]。
- 默认：[]

replicate-do-table

指定要同步的表格，示例如下：

```
[[syncer.replicate-do-table]]
db-name ="test"
tbl-name = "log"

[[syncer.replicate-do-table]]
db-name ="test"
tbl-name = "~^a.*"
```

默认：[]

txn-batch

- 当下游为 mysql/tidb 类型时，会将 DML 分批执行。这个配置可以用于设置每个事务中包含多少个 DML。
- 默认：20

worker-count

- 当下游为 mysql/tidb 类型时，会并发执行 DML，worker-count 可以指定并发数。
- 默认：16

disable-dispatch

- 关掉并发，强制将 worker-count 置为 1。
- 默认：false

safe-mode

如果打开 Safe mode，Drainer 会对同步的变更作以下修改，使其变成可重入的操作：

- Insert 变为 Replace Into
- Update 变为 Delete 和 Replace Into

默认：false

syncer.to

不同类型的下游配置都在 syncer.to 分组。以下按配置类型进行介绍。

mysql/tidb

用于连接下游数据库的配置项：

- host：如果没有设置，会尝试检查环境变量 MYSQL_HOST，默认值为 "localhost"。
- port：如果没有设置，会尝试检查环境变量 MYSQL_PORT，默认值为 3306。
- user：如果没有设置，会尝试检查环境变量 MYSQL_USER，默认值为 "root"。
- password：如果没有设置，会尝试检查环境变量 MYSQL_PSWD，默认值为 ""。

file

- dir：指定用于保存 binlog 的目录。如果不指定该项，则使用 data-dir。

kafka

当下游为 kafka 时，有效的配置包括：

- zookeeper-addrs
- kafka-addrs
- kafka-version
- kafka-max-messages
- topic-name

syncer.to.checkpoint

以下是 syncer.to.checkpoint 相关的配置项。

type

- 指定用哪种方式保存同步进度。
- 目前支持的选项：mysql 和 tidb
- 默认：与下游类型相同。例如 file 类型的下游进度保存在本地文件系统，mysql 类型的下游进度保存在下游数据库。当明确指定要使用 mysql 或 tidb 保存同步进度时，需要指定以下配置项：
  - schema：默认为 "tidb_binlog"。

> 注意：
> 在同个 TiDB 集群中部署多个 Drainer 时，需要为每个 Drainer 节点指定不同的 checkpoint schema，否则两个实例的同步进度会互相覆盖。

  - host
  - user
  - password
  - port

11.7.6 TiDB Binlog 版本升级方法

如未特别指明，文中出现的 TiDB Binlog 均指最新的Cluster 版本。

本文会分 TiDB Ansible 部署和手动部署两种情况介绍 TiDB Binlog 版本升级的方法，另外有一小节介绍如何从更早的不兼容版本（Kafka/Local 版本）升级到最新版本。

11.7.6.1 TiDB Ansible 部署

本节适用于使用 TiDB Ansible Playbook 部署的情况。

11.7.6.1.1 升级 Pump

1. 将新版本的二进制文件 pump 复制到 `{{ resources_dir }}/bin` 目录中
2. 执行 `ansible-playbook rolling_update.yml --tags=pump` 命令来滚动升级 Pump

11.7.6.1.2 升级 Drainer

1. 将新版本的二进制文件 `drainer` 复制到 `{{ resources_dir }}/bin` 目录中
2. 执行 `ansible-playbook stop_drainer.yml --tags=drainer` 命令
3. 执行 `ansible-playbook start_drainer.yml --tags=drainer` 命令

11.7.6.2 手动部署

11.7.6.2.1 升级 Pump

对集群里的每个 Pump 逐一升级，确保集群中总有 Pump 可以接收 TiDB 发来的 Binlog。

1. 用新版本的 pump 替换原来的文件
2. 重启 Pump 进程

11.7.6.2.2 升级 Drainer

1. 用新版本的 `drainer` 替换原来的文件
2. 重启 Drainer 进程

11.7.6.3 从 Kafka/Local 版本升级到 Cluster 版本

新版本的 TiDB（v2.0.8-binlog、v2.1.0-rc.5 及以上版本）不兼容 Kafka 版本以及 Local 版本的 TiDB Binlog，集群升级到新版本后只能使用 Cluster 版本的 TiDB Binlog。如果在升级前已经使用了 Kafka / Local 版本的 TiDB Binlog，必须将其升级到 Cluster 版本。

TiDB Binlog 版本与 TiDB 版本的对应关系如下：

TiDB Binlog 版本	TiDB 版本	说明
Local	TiDB 1.0 及更低版本	

TiDB Binlog 版本	TiDB 版本	说明
Kafka	TiDB 1.0 ~ TiDB 2.1 RC5	TiDB 1.0 支持 local 版本和 Kafka 版本的 TiDB Binlog。
Cluster	TiDB v2.0.8-binlog，TiDB 2.1 RC5 及更高版本	TiDB v2.0.8-binlog 是一个支持 Cluster 版本 TiDB Binlog 的 2.0 特殊版本。

11.7.6.3.1 升级流程

> **注意：**
>
> 如果能接受重新导全量数据，则可以直接废弃老版本，按TiDB Binlog 集群部署中的步骤重新部署。

如果想从原来的 checkpoint 继续同步，使用以下升级流程：

1. 部署新版本 Pump。

2. 暂停 TiDB 集群业务。
3. 更新 TiDB 以及配置，写 Binlog 到新的 Pump Cluster。
4. TiDB 集群重新接入业务。
5. 确认老版本的 Drainer 已经将老版本的 Pump 的数据完全同步到下游。

   查询 Drainer 的 `status` 接口，示例命令如下：

```
curl 'http://172.16.10.49:8249/status'
```

```
{"PumpPos":{"172.16.10.49:8250":{"offset":32686}},"Synced": true ,"DepositWindow":{"Upper
    ↪ ":398907800202772481,"Lower":398907799455662081}}
```

   如果返回的 Synced 为 true，则可以认为 Binlog 数据已经全部同步到了下游。

6. 启动新版本 Drainer；
7. 下线老版本的 Pump、 Drainer 以及依赖的 Kafka 和 ZooKeeper。

11.7.7 TiDB Binlog 集群监控

使用 TiDB Ansible 成功部署 TiDB Binlog 集群后，可以进入 Grafana Web 界面（默认地址: http://grafana_ip:3000，默认账号：admin，密码：admin）查看 Pump 和 Drainer 的运行状态。

11.7.7.1 监控指标

11.7.7.1.1 Pump

metric 名称	说明
Storage Size	记录磁盘的总空间大小 (capacity)，以及可用磁盘空间大小 (available)
Metadata	记录每个 Pump 的可删除 binlog 的最大 tso (gc_tso)，以及保存的 binlog 的最大的 commit tso (max_commit_tso)。
Write Binlog QPS by Instance	每个 Pump 接收到的写 binlog 请求的 QPS

metric 名称	说明
Write Binlog Latency	记录每个 Pump 写 binlog 的延迟时间
Storage Write Binlog Size	Pump 写 binlog 数据的大小
Storage Write Binlog Latency	Pump 中的 storage 模块写 binlog 数据的延迟
Pump Storage Error By Type	Pump 遇到的 error 数量，按照 error 的类型进行统计
Query TiKV	Pump 通过 TiKV 查询事务状态的次数

11.7.7.1.2 Drainer

metric 名称	说明
Checkpoint TSO	Drainer 已经同步到下游的 binlog 的最大 TSO 对应的时间。可以通过该指标估算同步延迟时间
Pump Handle TSO	记录 Drainer 从各个 Pump 获取到的 binlog 的最大 TSO 对应的时间
Pull Binlog QPS by Pump NodeID	Drainer 从每个 Pump 获取 binlog 的 QPS
95% Binlog Reach Duration By Pump	记录 binlog 从写入 Pump 到被 Drainer 获取到这个过程的延迟时间
Error By Type	Drainer 遇到的 error 数量，按照 error 的类型进行统计
SQL Query Time	Drainer 在下游执行 SQL 的耗时
Drainer Event	各种类型 event 的数量，event 包括 ddl、insert、delete、update、flush、savepoint
Execute Time	写入 binlog 到同步下游模块所消耗的时间
95% Binlog Size	Drainer 从各个 Pump 获取到 binlog 数据的大小
DDL Job Count	Drainer 处理的 DDL 的数量
Queue Size	Drainer 内部工作队列大小

11.7.7.2 监控报警规则

本节介绍了 TiDB Binlog 组件的报警项及相应的报警规则。根据严重级别，报警项可分为三类，按照严重程度由高到低依次为：紧急级别 (Emergency)、重要级别 (Critical)、警告级别 (Warning)。

11.7.7.2.1 紧急级别报警项

紧急级别的报警通常由于服务停止或节点故障导致，此时需要马上进行人工干预操作。

`binlog_pump_storage_error_count`

- 报警规则：

  ```
  changes(binlog_pump_storage_error_count[1m])> 0
  ```

- 规则描述：

  Pump 写 binlog 到本地存储时失败。

- 处理方法：

  确认 `pump_storage_error` 监控是否存在错误，查看 Pump 日志确认原因。

11.7.7.2.2 重要级别报警项

对于重要级别的报警，需要密切关注异常指标。

`binlog_drainer_checkpoint_high_delay`

- 报警规则：

  ```
  (time()- binlog_drainer_checkpoint_tso / 1000)> 3600
  ```

- 规则描述：

  Drainer 同步落后延迟超过 1 个小时。

- 处理方法：

  - 判断从 Pump 获取数据是否太慢：

    监控 Pump handle tso 可以看每个 Pump 最近一条消息的时间，是不是有延迟特别大的 Pump，确认对应 Pump 正常运行。

  - 根据 Drainer event 和 Drainer execute latency 来判断是否下游同步太慢：

    * 如果 Drainer execute time 过大，则检查到目标库网络带宽和延迟，以及目标库状态。
    * 如果 Drainer execute time 不大，Drainer event 过小，则增加 work count 和 batch 进行重试。

  - 如果上面都不满足或者操作后没有改观，则报备开发人员 support@pingcap.com 进行处理。

11.7.7.2.3 警告级别报警项

警告级别的报警是对某一问题或错误的提醒。

`binlog_pump_write_binlog_rpc_duration_seconds_bucket`

- 报警规则：

  ```
  histogram_quantile(0.9, rate(binlog_pump_rpc_duration_seconds_bucket{method="WriteBinlog"}[5
  ↪ m]))> 1
  ```

- 规则描述：

  Pump 处理 TiDB 写 Binlog 请求耗时过大。

- 处理方法：
  - 确认磁盘性能压力，通过 node exported 查看 disk performance 监控。
  - 如果 disk latency 和 util 都很低，那么报备开发人员 support@pingcap.com 进行处理。

binlog_pump_storage_write_binlog_duration_time_bucket

- 报警规则：

  ```
  histogram_quantile(0.9, rate(binlog_pump_storage_write_binlog_duration_time_bucket{type="
  ↪ batch"}[5m]))> 1
  ```

- 规则描述：

  Pump 写本地 binlog 到本地盘的耗时。

- 处理方法：

  确认 Pump 本地盘情况，进行修复。

binlog_pump_storage_available_size_less_than_20G

- 报警规则：

  ```
  binlog_pump_storage_storage_size_bytes{type="available"} < 20 * 1024 * 1024 * 1024
  ```

- 规则描述：

  Pump 剩余可用磁盘空间不足 20 G。

- 处理方法：

  监控确认 Pump 的 gc_tso 是否正常。如果不正常，调整 Pump 的 GC 时间配置或者下线对应 Pump。

binlog_drainer_checkpoint_tso_no_change_for_1m

- 报警规则：

  ```
  changes(binlog_drainer_checkpoint_tso[1m])< 1
  ```

- 规则描述：

  Drainer 的 checkpoint 在 1 分钟内没有更新。

- 处理方法：

  确认所有非下线的 Pump 是否正常运行。

binlog_drainer_execute_duration_time_more_than_10s

- 报警规则：

  ```
  histogram_quantile(0.9, rate(binlog_drainer_execute_duration_time_bucket[1m]))> 10
  ```

- 规则描述：

  Drainer 同步到 TiDB 的事务耗时。如果这个值过大，会影响 Drainer 同步。

- 处理方法：
  - 查看 TiDB 集群的状态。
  - 查看 Drainer 日志或监控，如果是 DDL 操作导致了该问题，则忽略。

11.7.8 Reparo 使用文档

Reparo 是 TiDB Binlog 的一个配套工具，用于增量的恢复。使用 TiDB Binlog 中的 Drainer 将 binlog 按照 protobuf 格式输出到文件，通过这种方式来备份增量数据。当需要恢复增量数据时，使用 Reparo 解析文件中的 binlog，并将其应用到 TiDB / MySQL 中。

下载链接：tidb-binlog-cluster-latest-linux-amd64.tar.gz

11.7.8.1 Reparo 使用

11.7.8.1.1 命令行参数说明

```
Usage of Reparo:
-L string
    日志输出信息等级设置：debug, info, warn, error, fatal（默认值：info）。
-V 打印版本信息。
-c int
    同步下游的并发数，该值设置越高同步的吞吐性能越好（默认 16）。
-config string
    配置文件路径，如果指定了配置文件，Reparo 会首先读取配置文件的配置；
        ↪ 如果对应的配置在命令行参数里面也存在，Reparo
        ↪ 就会使用命令行参数的配置来覆盖配置文件里面的。
-data-dir string
    Drainer 输出的 protobuf 格式 binlog 文件的存储路径（默认值： data.drainer）。
-dest-type string
    下游服务类型。 取值为 print, mysql（默认值：print）。当值为 print 时，只做解析打印到标准输出
        ↪ ，不执行 SQL；如果为 mysql，则需要在配置文件内配置 host、port、user、password 等信息
        ↪ 。
-log-file string
    log 文件路径。
-log-rotate string
    log 文件切换频率，取值为 hour、day。
-start-datetime string
    用于指定开始恢复的时间点，格式为 “2006-01-02 15:04:05”。如果不设置该参数则从最早的 binlog
        ↪ 文件开始恢复。
-stop-datetime string
    用于指定结束恢复的时间点，格式同上。如果不设置该参数则恢复到最后一个 binlog 文件。
-safe-mode bool
    指定是否开启安全模式，开启后可支持反复同步。
-txn-batch int
    输出到下游数据库一个事务的 SQL 语句数量（默认 20）。
```

11.7.8.1.2 配置文件说明

```
### Drainer 输出的 protobuf 格式 binlog 文件的存储路径。
```

```
data-dir = "./data.drainer"

### 日志输出信息等级设置：debug, info, warn, error, fatal（默认值：info）。
log-level = "info"

### 使用 start-datetime 和 stop-datetime 来选择恢复指定时间范围内的 binlog，格式为 “2006-01-02
    ↪ 15:04:05”。
### start-datetime = ""
### stop-datetime = ""

### start-tso、stop-tso 分别对应 start-datetime 和 stop-datetime，也是用于恢复指定时间范围内的
    ↪ binlog，用 tso 的值来设置。如果已经设置了 start-datetime 和 stop-datetime，就不需要再设置
    ↪  start-tso 和 stop-tso。
### start-tso = 0
### stop-tso = 0

### 下游服务类型。 取值为 print, mysql（默认值：print）。当值为 print 时，只做解析打印到标准输出
    ↪ ，不执行 SQL；如果为 mysql，则需要在 [dest-db] 中配置 host、port、user、password 等信息。
dest-type = "mysql"

### 输出到下游数据库一个事务的 SQL 语句数量（默认 20）。
txn-batch = 20

### 同步下游的并发数，该值设置越高同步的吞吐性能越好（默认 16）。
worker-count = 16

### 安全模式配置。取值为 true 或 false（默认值：false）。当值为 true 时，Reparo 会将 update
    ↪ 语句拆分为 delete + replace 语句。
safe-mode = false

### replicate-do-db 和 replicate-do-table 用于指定恢复的库和表，replicate-do-db 的优先级高于
    ↪ replicate-do-table。支持使用正则表达式来配置，需要以 '~' 开始声明使用正则表达式。
### 注：replicate-do-db 和 replicate-do-table 使用方式与 Drainer 的使用方式一致。
### replicate-do-db = ["~^b.*","s1"]
### [[replicate-do-table]]
### db-name ="test"
### tbl-name = "log"
### [[replicate-do-table]]
### db-name ="test"
### tbl-name = "~^a.*"

### 如果 dest-type 设置为 mysql，需要配置 dest-db。
[dest-db]
host = "127.0.0.1"
port = 3309
```

```
user = "root"
password = ""
```

11.7.8.1.3 启动示例

```
./bin/reparo -config reparo.toml
```

> 注意：
>
> - data-dir 用于指定 Drainer 输出的 binlog 文件目录。
> - start-datatime 和 start-tso 效果一样，只是时间格式上的区别，用于指定开始恢复的时间点；如果不指定，则默认在第一个 binlog 文件开始恢复。
> - stop-datetime 和 stop-tso 效果一样，只是时间格式上的区别，用于指定结束恢复的时间点；如果不指定，则恢复到最后一个 binlog 文件的结尾。
> - dest-type 指定目标类型，取值为 `mysql`、`print`。当值为 `mysql` 时，可以恢复到 MySQL/TiDB 等使用或兼容 MySQL 协议的数据库，需要在配置下面的 [dest-db] 填写数据库信息；当取值为 `print` 的时候，只是打印 binlog 信息，通常用于 debug，以及查看 binlog 的内容，此时不需要填写 `[dest-db]`。
> - replicate-do-db 用于指定恢复的库，不指定的话，则全部都恢复。
> - replicate-do-table 用于指定要恢复的表，不指定的话，则全部都恢复。

11.7.9 binlogctl 工具

Binlog Control（以下简称 binlogctl）是 TiDB Binlog 的命令行工具，用于管理 TiDB Binlog 集群。

binlogctl 支持如下这些功能：

- 查看 Pump/Drainer 状态
- 暂停/下线 Pump/Drainer
- Pump/Drainer 异常状态处理

使用 binlogctl 的场景：

- 同步出现故障/检查运行情况，需要查看 Pump/Drainer 的状态
- 维护集群，需要暂停/下线 Pump/Drainer
- Pump/Drainer 异常退出，状态没有更新，或者状态不符合预期，对业务造成影响

11.7.9.1 binlogctl 下载

> 注意：
>
> 建议使用的 Control 工具版本与集群版本保持一致。

binlogctl 下载链接：

```
wget https://download.pingcap.org/tidb-{version}-linux-amd64.tar.gz &&
wget https://download.pingcap.org/tidb-{version}-linux-amd64.sha256
```

检查文件完整性，返回 ok 则正确：

```
sha256sum -c tidb-{version}-linux-amd64.sha256
```

检查文件完整性，返回 ok 则正确：

```
sha256sum -c tidb-enterprise-tools-latest-linux-amd64.sha256
```

11.7.9.2 binlogctl 使用说明

命令行参数：

```
Usage of binlogctl:
-V
输出 binlogctl 的版本信息
-cmd string
    命令模式，包括 "generate_meta"（已废弃）, "pumps", "drainers", "update-pump" ,"update-drainer
        ↪ ", "pause-pump", "pause-drainer", "offline-pump", "offline-drainer"
-data-dir string
    保存 Drainer 的 checkpoint 的文件的路径（默认 "binlog_position"）（已废弃）
-node-id string
    Pump/Drainer 的 ID
-pd-urls string
    PD 的地址，如果有多个，则用"," 连接（默认 "http://127.0.0.1:2379"）
-ssl-ca string
    SSL CAs 文件的路径
-ssl-cert string
        PEM 格式的 X509 认证文件的路径
-ssl-key string
        PEM 格式的 X509 key 文件的路径
-time-zone string
    如果设置时区，在 "generate_meta" 模式下会打印出获取到的 tso 对应的时间。例如 "Asia/Shanghai"
        ↪ 为 CST 时区，"Local" 为本地时区
-show-offline-nodes
    在用 `-cmd pumps` 或 `-cmd drainers` 命令时使用，这两个命令默认不显示 offline 的节点，
        ↪ 仅当明确指定 `-show-offline-nodes` 时会显示
```

命令示例：

- 查询所有的 Pump/Drainer 的状态：

  设置 cmd 为 pumps 或者 drainers 来查看所有 Pump 或者 Drainer 的状态。例如：

```
bin/binlogctl -pd-urls=http://127.0.0.1:2379 -cmd pumps
```

```
[2019/04/28 09:29:59.016 +00:00] [INFO] [nodes.go:48] ["query node"] [type=pump] [node="{
    ↪ NodeID: 1.1.1.1:8250, Addr: pump:8250, State: online, MaxCommitTS:
    ↪ 408012403141509121, UpdateTime: 2019-04-28 09:29:57 +0000 UTC}"]
```

```
bin/binlogctl -pd-urls=http://127.0.0.1:2379 -cmd drainers
```

```
[2019/04/28 09:29:59.016 +00:00] [INFO] [nodes.go:48] ["query node"] [type=drainer] [node="{
    ↪ NodeID: 1.1.1.1:8249, Addr: 1.1.1.1:8249, State: online, MaxCommitTS:
    ↪ 408012403141509121, UpdateTime: 2019-04-28 09:29:57 +0000 UTC}"]
```

- 暂停/下线 Pump/Drainer

  binlogctl 提供以下命令暂停/下线服务：

cmd	说明	示例
pause-pump	暂停 Pump	`bin/binlogctl -pd-urls=http://127.0.0.1:2379 -cmd pause-pump -node-id ip-127-0-0-1:8250`
pause-drainer	暂停 Drainer	`bin/binlogctl -pd-urls=http://127.0.0.1:2379 -cmd pause-drainer -node-id ip-127-0-0-1:8249`
offline-pump	下线 Pump	`bin/binlogctl -pd-urls=http://127.0.0.1:2379 -cmd offline-pump -node-id ip-127-0-0-1:8250`
offline-drainer	下线 Drainer	`bin/binlogctl -pd-urls=http://127.0.0.1:2379 -cmd offline-drainer -node-id ip-127-0-0-1:8249`

  binlogctl 会发送 HTTP 请求给 Pump/Drainer，Pump/Drainer 收到命令后会主动执行对应的退出流程。

- 异常情况下修改 Pump/Drainer 的状态

  在服务正常运行以及符合流程的暂停、下线过程中，Pump/Drainer 的状态都是可以正确的。但是在一些异常情况下 Pump/Drainer 无法正确维护自己的状态，可能会影响数据同步任务，在这种情况下需要使用 binlogctl 修复状态信息。

  设置 cmd 为 `update-pump` 或者 `update-drainer` 来更新 Pump 或者 Drainer 的状态。Pump 和 Drainer 的状态可以为 paused 或者 offline。例如：

```
bin/binlogctl -pd-urls=http://127.0.0.1:2379 -cmd update-pump -node-id ip-127-0-0-1:8250 -
    ↪ state paused
```

> **注意：**
>
> Pump/Drainer 在正常运行过程中会定期在 PD 中更新自己的状态，而这条命令是直接去修改 Pump/Drainer 保存在 PD 中的状态，所以在 Pump/Drainer 服务正常的情况下使用这些命令是没有意义的。仅在 Pump/Drainer 服务异常的情况下使用，具体哪些场景下使用这条命令可以参考 FAQ。

11.7.10 Binlog Consumer Client 用户文档

目前 Drainer 提供了多种输出方式，包括 MySQL、TiDB、file 等。但是用户往往有一些自定义的需求，比如输出到 Elasticsearch、Hive 等，这些需求 Drainer 现在还没有实现，因此 Drainer 增加了输出到 Kafka 的功能，将 binlog 数据解析后按一定的格式再输出到 Kafka 中，用户编写代码从 Kafka 中读出数据再进行处理。

11.7.10.1 配置 Kafka Drainer

修改 Drainer 的配置文件，设置输出为 Kafka，相关配置如下：

```
[syncer]
db-type = "kafka"

[syncer.to]
### Kafka 地址
kafka-addrs = "127.0.0.1:9092"
### Kafka 版本号
kafka-version = "0.8.2.0"
```

11.7.10.2 自定义开发

11.7.10.2.1 数据格式

首先需要了解 Drainer 写入到 Kafka 中的数据格式：

```
// Column 保存列的数据，针对数据的类型，保存在对应的变量中
message Column {
  // 数据是否为 null
  optional bool is_null = 1 [ default = false ];
  // 保存 int 类型的数据
  optional int64 int64_value = 2;
  // 保存 uint、enum, set 类型的数据
  optional uint64 uint64_value = 3;
  // 保存 float、double 类型的数据
  optional double double_value = 4;
  // 保存 bit、blob、binary、json 类型的数据
  optional bytes bytes_value = 5;
  // 保存 date、time、decimal、text、char 类型的数据
  optional string string_value = 6;
}

// ColumnInfo 保存列的信息，包括列名、类型、是否为主键
message ColumnInfo {
  optional string name = 1 [ (gogoproto.nullable) = false ];
  // MySQL 中小写的列字段类型
  // https://dev.mysql.com/doc/refman/8.0/en/data-types.html
```

```
  // numeric 类型：int bigint smallint tinyint float double decimal bit
  // string 类型：text longtext mediumtext char tinytext varchar
  // blob longblob mediumblob binary tinyblob varbinary
  // enum set
  // json 类型：json
  optional string mysql_type = 2 [ (gogoproto.nullable) = false ];
  optional bool is_primary_key = 3 [ (gogoproto.nullable) = false ];
}

// Row 保存一行的具体数据
message Row { repeated Column columns = 1; }

// MutationType 表示 DML 的类型
enum MutationType {
  Insert = 0;
  Update = 1;
  Delete = 2;
}

// Table 包含一个表的数据变更
message Table {
  optional string schema_name = 1;
  optional string table_name = 2;
  repeated ColumnInfo column_info = 3;
  repeated TableMutation mutations = 4;
}

// TableMutation 保存一行数据的变更
message TableMutation {
  required MutationType type = 1;
  // 修改后的数据
  required Row row = 2;
  // 修改前的数据，只对 Update MutationType 有效
  optional Row change_row = 3;
}

// DMLData 保存一个事务所有的 DML 造成的数据变更
message DMLData {
  // `tables` 包含事务中所有表的数据变更
  repeated Table tables = 1;
}

// DDLData 保存 DDL 的信息
message DDLData {
  // 当前使用的数据库
```

```
  optional string schema_name = 1;
  // 相关表
  optional string table_name = 2;
  // `ddl_query` 是原始的 DDL 语句 query
  optional bytes ddl_query = 3;
}

// BinlogType 为 Binlog 的类型，分为 DML 和 DDL
enum BinlogType {
  DML = 0; //  Has `dml_data`
  DDL = 1; //  Has `ddl_query`
}

// Binlog 保存一个事务所有的变更，Kafka 中保存的数据为该结构数据序列化后的结果
message Binlog {
  optional BinlogType type = 1 [ (gogoproto.nullable) = false ];
  optional int64 commit_ts = 2 [ (gogoproto.nullable) = false ];
  optional DMLData dml_data = 3;
  optional DDLData ddl_data = 4;
}
```

查看数据格式的具体定义，参见 binlog.proto。

11.7.10.2.2 Driver

TiDB-Tools 项目提供了用于读取 Kafka 中 binlog 数据的 Driver，具有如下功能：

- 读取 Kafka 的数据
- 根据 commit ts 查找 binlog 在 kafka 中的储存位置

使用该 Driver 时，用户需要配置如下信息：

- KafkaAddr：Kafka 集群的地址
- CommitTS：从哪个 commit ts 开始读取 binlog
- Offset：从 Kafka 哪个 offset 开始读取，如果设置了 CommitTS 就不用配置该参数
- ClusterID：TiDB 集群的 cluster ID
- Topic: Kafka Topic 名称，如果 Topic 名称为空，将会使用 drainer `<ClusterID>_obinlog` 中的默认名称

用户以包的形式引用 Driver 的代码即可使用，可以参考 Driver 中提供的示例代码来学习如何使用 Driver 以及 binlog 数据的解析，目前提供了两个例子：

- 使用该 Driver 将数据同步到 MySQL，该示例包含将 binlog 转化为 SQL 的具体方法
- 使用该 Driver 将数据打印出来

Driver 项目地址：TiDB Binlog Driver。

注意：

- 示例代码仅仅用于示范如何使用 Driver，如果需要用于生产环境需要优化代码。
- 目前仅提供了 golang 版本的 Driver 以及示例代码。如果需要使用其他语言，用户需要根据 binlog 的 proto 文件生成相应语言的代码文件，并自行开发程序读取 Kafka 中的 binlog 数据、解析数据、输出到下游。也欢迎用户优化 example 代码，以及提交其他语言的示例代码到 TiDB-Tools。

11.7.11 TiDB Binlog Relay Log

Drainer 同步 binlog 时会拆分上游的事务，并将拆分的事务并发同步到下游。在极端情况下，上游集群不可用并且 Drainer 异常退出后，下游集群（MySQL 或 TiDB）可能处于数据不一致的中间状态。在此场景下，Drainer 借助 relay log 可以确保将下游集群同步到一个一致的状态。

11.7.11.1 Drainer 同步时的一致性状态

下游集群达到一致的状态是指：下游集群的数据等同于上游设置了 `tidb_snapshot = ts` 的快照。

checkpoint 状态一致性是指：Drainer checkpoint 通过 `consistent` 保存了同步的一致性状态。Drainer 运行时 `consistent` 为 `false`，Drainer 正常退出后 `consistent` 更新为 `true`。

查询下游 checkpoint 表的示例如下：

```
mysql> select  * from tidb_binlog.checkpoint;
+--------------------+--------------------------------------------------------------------+
| clusterID          | checkPoint                                                         |
+--------------------+--------------------------------------------------------------------+
| 6791641053252586769 | {"consistent":false,"commitTS":414529105591271429,"ts-map":{}} |
+--------------------+--------------------------------------------------------------------+
```

11.7.11.2 工作原理

Drainer 开启 relay log 后会先将 binlog event 写到磁盘上，然后再同步给下游集群。如果上游集群不可用，Drainer 可以通过读取 relay log 把下游集群恢复到一个一致的状态。

注意：

若同时丢失 relay log 数据，该方法将不可用，不过这是概率极小的事件。此外可以使用 NFS 等网络文件系统来保证 relay log 的数据安全。

11.7.11.2.1 Drainer 从 relay log 消费 binlog 的触发场景

如果 Drainer 启动时无法连接到上游集群的 PD，并且探测到 checkpoint 的 `consistent = false`，此时会尝试读取 relay log，并将下游集群恢复到一致的状态。然后 Drainer 进程将 checkpoint 的 `consistent` 设置为 `true` 后主动退出。

11.7.11.2.2 Relay log 的清理（GC）机制

Drainer 在将数据同步到下游之前，会先将数据写入到 relay log 文件中。当一个 relay log 文件大小达到 10MB（默认）并且当前事务的 binlog 数据被写入完成后，Drainer 就会开始将数据写入到下一个 relay log 文件中。当 Drainer 将数据成功同步到下游后，就会自动清除当前正在写入的 relay log 文件以外其他已完成同步的 relay log 文件。

11.7.11.3 配置

在 Drainer 中添加以下配置来开启 relay log 功能：

```
[syncer.relay]
### 保存 relay log 的目录，空值表示不开启。
### 只有下游是 TiDB 或 MySQL 时该配置才有生效。
log-dir = "/dir/to/save/log"
### 单个 relay log 文件大小限制（单位：字节）。
### 超出该值后会将 binlog 数据写入到下一个 relay log 文件。
max-file-size = 10485760
```

11.7.12 集群间双向同步

> **警告：**
>
> 目前双向同步属于实验特性，尚未经过完备的测试，不建议在生产环境中使用该功能。

本文档介绍如何将一个 TiDB 集群的数据双向同步到另一个 TiDB 集群、双向同步的实现原理、如何开启双向同步、以及如何同步 DDL 操作。

11.7.12.1 使用场景

当用户需要在两个 TiDB 集群之间双向同步数据时，可使用 TiDB Binlog 进行操作。例如要将集群 A 的数据同步到集群 B，而且要将集群 B 的数据同步到集群 A。

> **注意：**
>
> 集群间双向同步的前提条件是，写入两个集群的数据必须保证无冲突，即在两个集群中，不会同时修改同一张表的同一主键和具有唯一索引的行。

使用场景示例图如下：

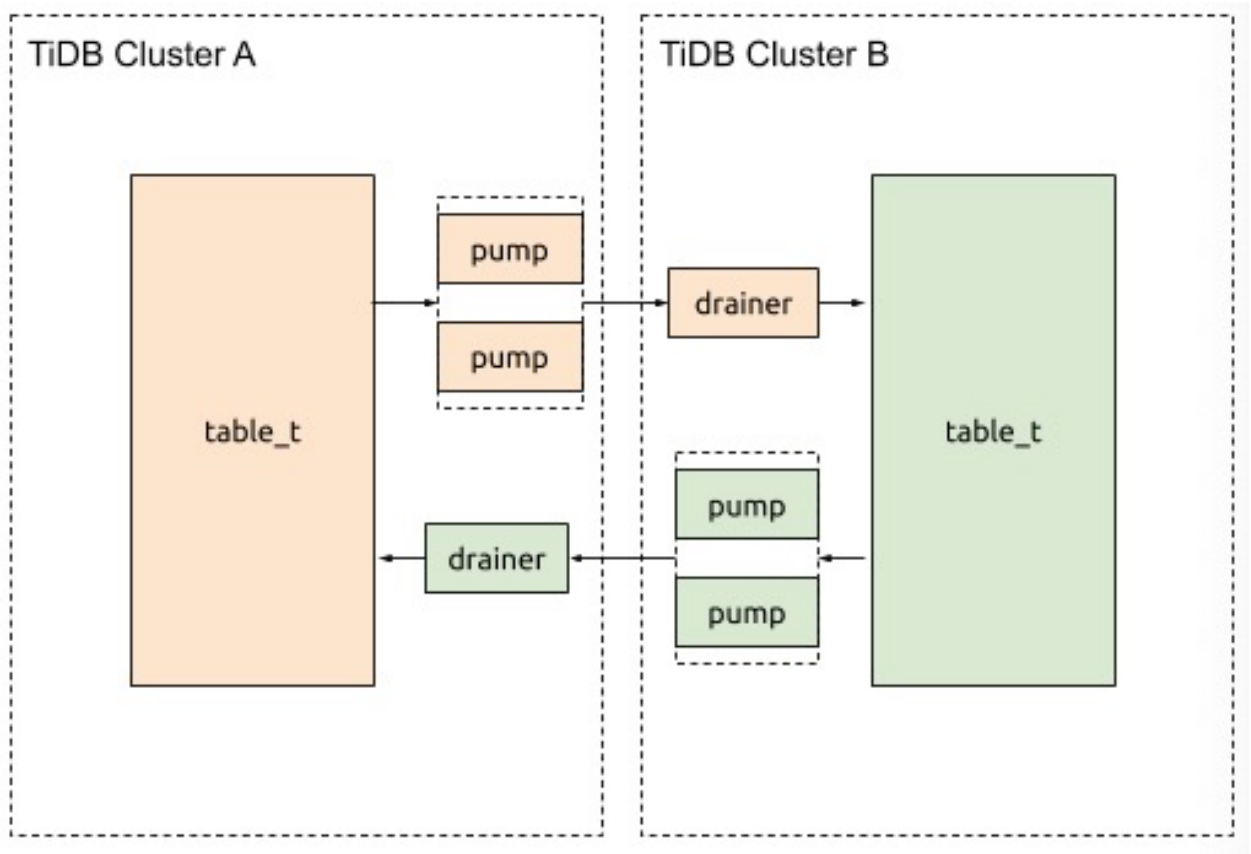


图 160: 使用场景示例图

11.7.12.2 实现原理

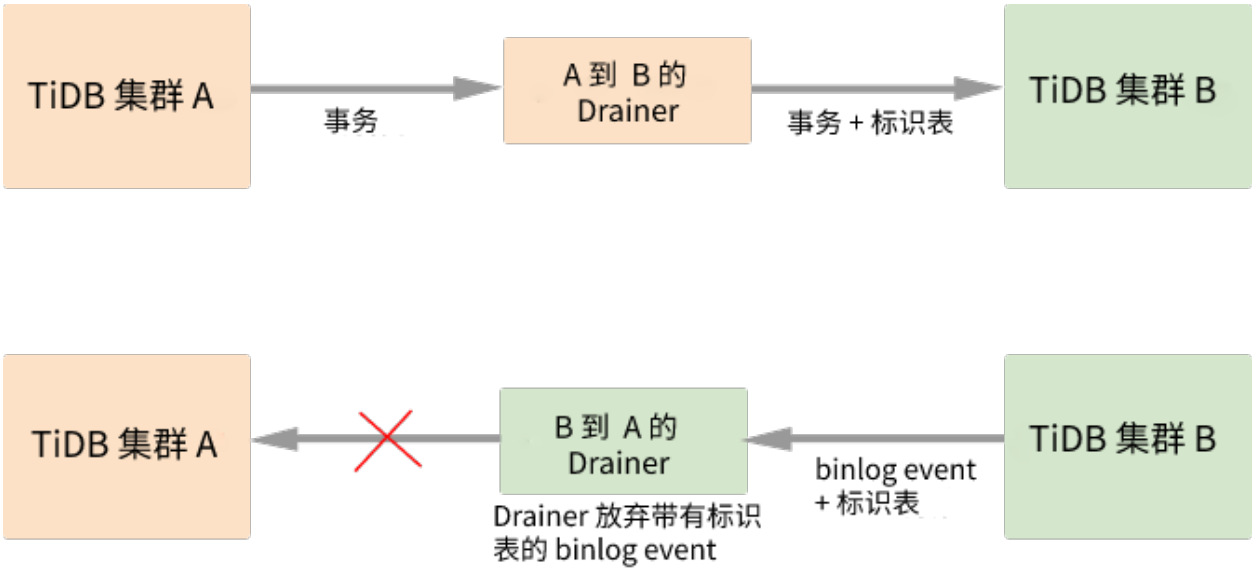


图 161: 原理示例图

在 A 和 B 两个集群间开启双向同步，则写入集群 A 的数据会同步到集群 B 中，然后这部分数据又会继续同步到集群 A，这样就会出现无限循环同步的情况。如上图所示，在同步数据的过程中 Drainer 对 binlog 加上标记，通过过滤掉有标记的 binlog 来避免循环同步。详细的实现流程如下：

1. 为两个集群分别启动 TiDB Binlog 同步程序。
2. 待同步的事务经过 A 的 Drainer 时，Drainer 为事务加入 `_drainer_repl_mark` 标识表，并在表中写入本次 DML event 更新，将事务同步至集群 B。
3. 集群 B 向集群 A 返回带有 `_drainer_repl_mark` 标识表的 binlog event。集群 B 的 Drainer 在解析该 binlog event 时发现带有 DML event 的标识表，放弃同步该 binlog event 到集群 A。

将集群 B 中的数据同步到集群 A 的流程与以上流程相同，两个集群可以互为上下游。

> 注意：
>
> - 更新 `_drainer_repl_mark` 标识表时，一定要有数据改动才会产生 binlog。
> - DDL 操作没有事务概念，因此采取单向同步的方案，见同步 DDL

Drainer 与下游的每个连接可以使用一个 ID 以避免冲突。`channel_id` 用来表示进行双向同步的一个通道。A 和 B 两个集群进行双向同步的配置应使用相同的值。

当有添加或者删除列时，要同步到下游的数据可能会出现多列或者少列的情况。Drainer 通过添加配置来允许这种情况，会忽略多了的列值或者给少了的列写入默认值。

11.7.12.3 标识表

`_drainer_repl_mark` 标识表的结构如下：

```
CREATE TABLE `_drainer_repl_mark` (
  `id` bigint(20) NOT NULL,
  `channel_id` bigint(20) NOT NULL DEFAULT '0',
  `val` bigint(20) DEFAULT '0',
  `channel_info` varchar(64) DEFAULT NULL,
  PRIMARY KEY (`id`,`channel_id`)
);
```

Drainer 使用如下 SQL 语句更新 _drainer_repl_mark 可保证数据改动，从而保证产生 binlog：

```
update drainer_repl_mark set val = val + 1 where id = ? && channel_id = ?;
```

11.7.12.4 同步 DDL

因为 Drainer 无法为 DDL 操作加入标识表，所以采用单向同步的方式来同步 DDL 操作。

比如集群 A 到集群 B 开启了 DDL 同步，则集群 B 到集群 A 会关闭 DDL 同步。即 DDL 操作全部在 A 上执行。

> 注意：
>
> DDL 操作无法在两个集群上同时执行。执行 DDL 时，若同时有 DML 操作或者 DML binlog 没同步完，会可能出现 DML 同步的上下游表结构不一致的情况。

11.7.12.5 配置并开启双向同步

若要在集群 A 和集群 B 间进行双向同步，假设统一在集群 A 上执行 DDL。在集群 A 到集群 B 的同步路径上，向 Drainer 添加以下配置：

```
[syncer]
loopback-control = true
channel-id = 1 # 互相同步的两个集群配置相同的 ID。
sync-ddl = true # 需要同步 DDL 操作

[syncer.to]
### 1 表示 SyncFullColumn，2 表示 SyncPartialColumn。
### 若设为 SyncPartialColumn，Drainer 会允许下游表结构比当前要同步的数据多或者少列
### 并且去掉 SQL mode 的 STRICT_TRANS_TABLES，来允许少列的情况，并插入零值到下游。
sync-mode = 2

### 忽略 checkpoint 表。
[[syncer.ignore-table]]
db-name = "tidb_binlog"
tbl-name = "checkpoint"
```

在集群 B 到集群 A 的同步路径上，向 Drainer 添加以下配置：

```
[syncer]
loopback-control = true
channel-id = 1 # 互相同步的两个集群配置相同的 ID。
sync-ddl = false  # 不需要同步 DDL 操作。

[syncer.to]
### 1 表示 SyncFullColumn，2 表示 SyncPartialColumn。
### 若设为 SyncPartialColumn，Drainer 会允许下游表结构比当前要同步的数据多或者少列
### 并且去掉 SQL mode 的 STRICT_TRANS_TABLES，来允许少列的情况，并插入零值到下游。
sync-mode = 2

### 忽略 checkpoint 表。
[[syncer.ignore-table]]
db-name = "tidb_binlog"
tbl-name = "checkpoint"
```

11.7.13 TiDB Binlog 术语表

本文档介绍 TiDB Binlog 相关术语。

11.7.13.1 Binlog

在 TiDB Binlog 中，Binlog 通常指 TiDB 写的二进制日志数据，也指 Drainer 写到 Kafka 或者文件的二进制日志数据，前者与后者的格式不同。此外，TiDB 的 binlog 格式与 MySQL 的 binlog 格式也不同。

11.7.13.2 Binlog event

TiDB 写的 DML Binlog 有 3 种 event，分别为：INSERT、UPDATE 和 DELETE。在 Drainer 监控面板上可以看到同步数据对应的不同 event 的个数。

11.7.13.3 Checkpoint

Checkpoint 指保存的断点信息，记录了 Drainer 同步到下游的 commit-ts。Drainer 重启时可以读取 checkpoint，并从对应的 commit-ts 开始同步数据到下游。

11.7.13.4 Safe mode

Safe mode 指增量复制过程中，当表结构中存在主键或唯一索引时，用于支持幂等导入 DML 的模式。

该模式将来自上游的 INSERT 改写为 REPLACE，将 UPDATE 改写为 DELETE 与 REPLACE，然后再向下游执行。在 Drainer 启动的前 5 分钟，会自动启动 Safe mode；另外也可以在配置文件中通过 safe-mode 参数手动开启，但该配置仅在下游是 MySQL 或 TiDB 时有效。

11.7.14 故障诊断

11.7.14.1 TiDB Binlog 故障诊断

本文总结了在 TiDB Binlog 的使用过程中遇到问题的诊断流程，并指引用户通过监控、状态、日志等信息查找相应的解决方案。

如果你在使用 TiDB Binlog 时出现了异常，请尝试以下方式排查问题：

1. 查看各个监控指标是否异常，参见TiDB Binlog 集群监控。
2. 使用binlogctl 工具查看各个 Pump、 Drainer 的状态是否有异常。
3. 查看 Pump、 Drainer 日志中是否有 ERROR、 WARN，并根据详细的日志信息初步判断问题原因。

通过以上方式定位到问题后，在FAQ 以及常见错误及修复 中查找解决方案，如果没有查找到解决方案或者提供的解决方案无法解决问题，请提交 issue 或者联系相关技术支持人员。

11.7.14.2 TiDB Binlog 常见错误修复

本文档介绍 TiDB Binlog 中常见的错误以及修复方法。

11.7.14.2.1 Drainer 同步数据到 Kafka 时报错 “kafka server: Message was too large, server rejected it to avoid allocation error”

报错原因：如果在 TiDB 中执行了大事务，则生成的 binlog 数据会比较大，可能超过了 Kafka 的消息大小限制。

解决方法：需要调整 Kafka 的配置参数，如下所示。

```
message.max.bytes=1073741824
replica.fetch.max.bytes=1073741824
fetch.message.max.bytes=1073741824
```

11.7.14.2.2 Pump 报错 “no space left on device”

报错原因：本地磁盘空间不足，Pump 无法正常写 binlog 数据。

解决方法：需要清理磁盘空间，然后重启 Pump。

11.7.14.2.3 Pump 启动时报错 “fail to notify all living drainer”

报错原因：Pump 启动时需要通知所有 Online 状态的 Drainer，如果通知失败则会打印该错误日志。

解决方法：可以使用binlogctl 工具查看所有 Drainer 的状态是否有异常，保证 Online 状态的 Drainer 都在正常工作。如果某个 Drainer 的状态和实际运行情况不一致，则使用 binlogctl 修改状态，然后再重启 Pump。

11.7.14.2.4 TiDB Binlog 同步中发现数据丢失

需要确认所有 TiDB 实例均开启了 TiDB Binlog，并且运行状态正常。如果集群版本大于 v3.0，可以使用 `curl {TiDB_IP}:{STATUS_PORT}/info/all` 命令确认所有 TiDB 实例上的 TiDB Binlog 状态。

11.7.14.2.5 当上游事务较大时，Pump 报错 `rpc error: code = ResourceExhausted desc = trying to send message larger than max (2191430008 vs. 2147483647)`

出现该错误的原因是 TiDB 发送给 Pump 的 gRPC message 超过限值。可以在启动 Pump 时通过指定 `max-message-size` 来调整 Pump 可接受 gRPC message 的最大大小。

11.7.14.2.6 Drainer 输出 file 格式的增量数据，数据有什么清理机制吗？数据会被删除吗？

- 在 v3.0.x 版本的 Drainer 中，file 格式的增量数据没有任何清理机制。
- 在 v4.0.x 版本中，有基于时间的数据清理机制，详见 Drainer 的 `retention-time` 配置项。

11.7.15 TiDB Binlog 常见问题

本文介绍 TiDB Binlog 使用过程中的常见问题及解决方案。

11.7.15.1 开启 binog 对 TiDB 的性能有何影响？

- 对于查询无影响。
- 对于有写入或更新数据的事务有一点性能影响。延迟上，在 Prewrite 阶段要并发写一条 p-binlog 成功后才可以提交事务，一般写 binlog 比 KV Prewrite 快，所以不会增加延迟。可以在 Pump 的监控面板看到写 binlog 的响应时间。

11.7.15.2 TiDB Binlog 的同步延迟一般为多少？

TiDB Binlog 的同步延迟为秒级别，在非业务高峰时延迟一般为 3 秒左右。

11.7.15.3 Drainer 同步下游 TiDB/MySQL 的帐号需要哪些权限？

Drainer 同步帐号需要有如下权限：

- Insert
- Update
- Delete
- Create
- Drop
- Alter
- Execute
- Index
- Select

11.7.15.4 Pump 磁盘快满了怎么办？

确认 GC 正常：

- 确认 pump 监控面板 gc_tso 时间是否与配置一致。

如 gc 正常以下调整可以降低单个 pump 需要的空间大小：

- 调整 pump GC 参数减少保留数据天数。
- 添加 pump 结点。

11.7.15.5 Drainer 同步中断怎么办？

使用以下 binlogctl 命令查看 Pump 状态是否正常，以及是否全部非 `offline` 状态的 Pump 都在正常运行。

```
binlogctl -cmd pumps
```

查看 Drainer 监控与日志是否有对应报错，根据具体问题进行处理。

11.7.15.6 Drainer 同步下游 TiDB/MySQL 慢怎么办？

特别关注以下监控项：

- 通过 Drainer 监控 drainer event，可以看到 Drainer 当前每秒同步 Insert/Update/Delete 事件到下游的速度。
- 通过 Drainer 监控 sql query time，可以看到 Drainer 在下游执行 SQL 的响应时间。

同步慢的可能原因与解决方案：

- 同步的数据库包含没有主键或者唯一索引的表，需要给表加上主键。
- Drainer 与下游之间延迟大，可以调大 Drainer `worker-count` 参数（跨机房同步建议将 Drainer 部署在下游）。
- 下游负载不高，可以尝试调大 Drainer `worker-count` 参数。

11.7.15.7 假如有一个 Pump crash 了会怎样？

Drainer 会因为获取不到这个 Pump 的数据没法同步数据到下游。如果这个 Pump 能恢复，Drainer 就能恢复同步。

如果 Pump 没法恢复，可采用以下方式进行处理：

1. 使用binlogctl 将该 Pump 状态修改为 `offline`（丢失这个 Pump 的数据）
2. Drainer 获取到的数据会丢失这个 Pump 上的数据，下游跟上游数据会不一致，需要重新做全量 + 增量同步。具体步骤如下：
    1. 停止当前 Drainer。
    2. 上游做全量备份。
    3. 清理掉下游数据，包括 checkpoint 表 `tidb_binlog.checkpoint`。
    4. 使用上游的全量备份恢复下游数据。
    5. 部署 Drainer，使用 `initialCommitTs=`{从全量备份获取快照的时间戳}。

11.7.15.8 什么是 checkpoint？

Checkpoint 记录了 Drainer 同步到下游的 commit-ts，Drainer 重启时可以读取 checkpoint 接着从对应 commit-ts 同步数据到下游。Drainer 日志 `["write save point"] [ts=411222863322546177]` 表示保存对应时间戳的 checkpoint。

下游类型不同，checkpoint 的保存方式也不同：

- 下游 MySQL/TiDB 保存在 `tidb_binlog.checkpoint` 表。
- 下游 kafka/file 保存在对应配置目录里的文件。

因为 kafka/file 的数据内容包含了 commit-ts，所以如果 checkpoint 丢失，可以消费下游最新的一条数据看写到下游数据的最新 commit-ts。

Drainer 启动的时候会去读取 checkpoint，如果读取不到，就会使用配置的 `initial-commit-ts` 做为初次启动开始的同步时间点。

11.7.15.9 Drainer 机器发生故障，下游数据还在，如何在新机器上重新部署 Drainer？

如果下游数据还在，只要保证能从对应 checkpoint 接着同步即可。

假如 checkpoint 还在，可采用以下方式进行处理：

1. 部署新的 Drainer 并启动即可（参考 checkpoint 介绍，Drainer 可以读取 checkpoint 接着同步）。
2. 使用binlogctl 将老的 Drainer 状态修改成 `offline`。

假如 checkpoint 不在，可以如下处理：

1. 获取之前 Drainer 的 checkpoint `commit-ts`，做为新部署 Drainer 的 `initial-commit-ts` 配置来部署新的 Drainer。
2. 使用binlogctl 将老的 Drainer 状态修改成 `offline`。

11.7.15.10 如何用全量 + binlog 备份文件来恢复一个集群？

1. 清理集群数据并使用全部备份恢复数据。
2. 使用 reparo 设置 `start-tso = {全量备份文件快照时间戳 +1}`，`end-ts = 0`（或者指定时间点），恢复到备份文件最新的数据。

11.7.15.11 主从同步开启 `ignore-error` 触发 critical error 后如何重新部署？

TiDB 配置开启 `ignore-error` 写 binlog 失败后触发 critical error 告警，后续都不会再写 binlog，所以会有 binlog 数据丢失。如果要恢复同步，需要如下处理：

1. 停止当前 Drainer。
2. 重启触发 critical error 的 `tidb-server` 实例重新开始写 binlog（触发 critical error 后不会再写 binlog 到 pump）。
3. 上游做全量备份。
4. 清理掉下游数据包括 checkpoint 表 `tidb_binlog.checkpoint`。
5. 使用上游的全量备份恢复下游。
6. 部署 Drainer，使用 `initialCommitTs= {从全量备份获取快照的时间戳}`。

11.7.15.12 同步时出现上游数据库支持但是下游数据库执行会出错的 DDL，应该怎么办？

1. 查看 drainer.log 日志，查找 `exec failed` 找到 Drainer 退出前最后一条执行失败的 DDL。
2. 将 DDL 改为下游兼容支持的版本，在下游数据库中手动执行。
3. 查看 drainer.log 日志，查找执行失败的 DDL 语句，可以查询到该 DDL 的 commit-ts。例如：

```
[2020/05/21 09:51:58.019 +08:00] [INFO] [syncer.go:398] ["add ddl item to syncer, you can
    ↪ add this commit ts to `ignore-txn-commit-ts` to skip this ddl if needed"] [sql="
    ↪ ALTER TABLE `test` ADD INDEX (`index1`)"] ["commit ts"=416815754209656834]。
```

4. 编辑 `drainer.toml` 配置文件，在 `ignore-txn-commit-ts` 项中添加该 commit-ts，重启 Drainer。

在绝大部分情况下，TiDB 和 MySQL 的语句都是兼容的。用户需要注意的是上下游的 `sql_mode` 应当保持一致。

11.7.15.13 在什么情况下暂停和下线 Pump/Drainer？

首先需要通过以下内容来了解 Pump/Drainer 的状态定义和启动、退出的流程。

暂停主要针对临时需要停止服务的场景，例如：

- 版本升级：停止进程后使用新的 binary 启动服务。
- 服务器维护：需要对服务器进行停机维护。退出进程，等维护完成后重启服务。

下线主要针对永久（或长时间）不再使用该服务的场景，例如：

- Pump 缩容：不再需要那么多 Pump 服务了，所以下线部分服务。
- 同步任务取消：不再需要将数据同步到某个下游，所以下线对应的 Drainer。
- 服务器迁移：需要将服务迁移到其他服务器。下线服务，在新的服务器上重新部署。

11.7.15.14 可以通过哪些方式暂停 Pump/Drainer？

- 直接 kill 进程。

> 注意：
> 不能使用 `kill -9` 命令，否则 Pump/Drainer 无法对信号进行处理。

- 如果 Pump/Drainer 运行在前台，则可以通过按下 Ctrl+C 来暂停。
- 使用 binlogctl 的 `pause-pump` 或 `pause-drainer` 命令。

11.7.15.15 可以使用 binlogctl 的 update-pump/update-drainer 命令来下线 Pump/Drainer 服务吗？

不可以。使用 update-pump/update-drainer 命令会直接修改 PD 中保存的状态信息，并且不会通知 Pump/Drainer 做相应的操作。使用不当时，可能会干扰数据同步，某些情况下还可能会造成数据不一致的严重后果。例如：

- 当 Pump 正常运行或者处于暂停状态时，如果使用 update-pump 将该 Pump 设置为 offline，那么 Drainer 会放弃获取处于 offline 状态的 Pump 的 binlog 数据，导致该 Pump 最新的 binlog 数据没有同步到 Drainer，造成上下游数据不一致。
- 当 Drainer 正常运行时，使用 update-drainer 命令将该 Drainer 设置为 offline。如果这时启动一个 Pump 节点，Pump 只会通知 online 状态的 Drainer，导致该 Drainer 没有及时获取到该 Pump 的 binlog 数据，造成上下游数据不一致。

11.7.15.16 可以使用 binlogctl 的 update-pump/update-drainer 命令来暂停 Pump/Drainer 服务吗？

不可以。update-pump/update-drainer 命令直接修改 PD 中保存的状态信息。执行这个命令并不会通知 Pump/Drainer 做相应的操作，而且使用不当会使数据同步中断，甚至造成数据丢失。

11.7.15.17 什么情况下使用 binlogctl 的 update-pump 命令设置 Pump 状态为 paused？

在某些异常情况下，Pump 没有正确维护自己的状态，实际上状态应该为 paused。这时可以使用 update-pump 对状态进行修正，例如：

- Pump 异常退出（可能由 panic 或者误操作执行 kill -9 命令直接 kill 掉进程导致），Pump 保存在 PD 中的状态仍然为 online。如果暂时不需要重启 Pump 恢复服务，可以使用 update-pump 把该 Pump 状态设置为 paused，避免对 TiDB 写 binlog 和 Drainer 获取 binlog 造成干扰。

11.7.15.18 什么情况下使用 binlogctl 的 update-drainer 命令设置 Drainer 状态为 paused？

在某些异常情况下，Drainer 没有正确维护自己的状态，，对数据同步造成了影响，实际上状态应该为 paused。这时可以使用 update-drainer 对状态进行修正，例如：

- Drainer 异常退出（出现 panic 直接退出进程，或者误操作执行 kill -9 命令直接 kill 掉进程），Drainer 保存在 PD 中的状态仍然为 online。当 Pump 启动时无法正常通知该 Drainer（报错 notify drainer ...），导致 Pump 无法正常运行。这个时候可以使用 update-drainer 将 Drainer 状态更新为 paused，再启动 Pump。

11.7.15.19 可以通过哪些方式下线 Pump/Drainer？

目前只可以使用 binlogctl 的 offline-pump 和 offline-drainer 命令来下线 Pump 和 Drainer。

11.7.15.20 什么情况下使用 binlogctl 的 update-pump 命令设置 Pump 状态为 offline?

> **警告：**
>
> 仅在可以容忍 binlog 数据丢失、上下游数据不一致或者确认不再需要使用该 Pump 存储的 binlog 数据的情况下，才能使用 update-pump 修改 Pump 状态为 offline。

可以使用 update-pump 修改 Pump 状态为 offline 的情况有：

- 在某些情况下，Pump 异常退出进程，且无法恢复服务，同步就会中断。如果希望恢复同步且可以容忍部分 binlog 数据丢失，可以使用 update-pump 命令将该 Pump 状态设置为 offline，则 Drainer 会放弃拉取该 Pump 的 binlog 然后继续同步数据。
- 有从历史任务遗留下来且不再使用的 Pump 且进程已经退出（例如测试使用的服务），之后不再需要使用该服务，使用 update-pump 将该 Pump 设置为 offline。

在其他情况下一定要使用 offline-pump 命令让 Pump 走正常的下线处理流程。

11.7.15.21 Pump 进程已经退出，且状态为 paused，现在不想使用这个 Pump 节点了，能否用 binlogctl 的 update-pump 命令设置节点状态为 offline？

Pump 以 paused 状态退出进程时，不保证所有 binlog 数据被下游 Drainer 消费。所以这样做会有上下游数据不一致的风险。正确的做法是重新启动 Pump，然后使用 offline-pump 下线该 Pump。

11.7.15.22 什么情况下使用 binlogctl 的 update-drainer 命令设置 Drainer 状态为 offline？

- 有从历史任务遗留下来且不再使用的 Drainer 且进程已经退出（例如测试使用的服务），之后不再需要使用该服务，使用 update-drainer 将该 Drainer 设置为 offline。

11.7.15.23 可以使用 change pump、change drainer 等 SQL 操作来暂停或者下线 Pump/Drainer 服务吗？

目前还不支持。这种 SQL 操作会直接修改 PD 中保存的状态，在功能上等同于使用 binlogctl 的 update-pump、update-drainer 命令。如果需要暂停或者下线，仍然要使用 binlogctl。

11.7.15.24 TiDB 写入 binlog 失败导致 TiDB 卡住，日志中出现 listener stopped, waiting for manual stop

在 TiDB v3.0.12 以及之前，binlog 写入失败会导致 TiDB 报 fatal error。但是 TiDB 不会自动退出只是停止服务，看起来像服务卡住。TiDB 日志中可看到 listener stopped, waiting for manual stop。

遇到该问题需要根据具体情况判断是什么原因导致 binlog 写入失败。如果是 binlog 写入下游缓慢导致的，可以考虑扩容 Pump 或增加写 binlog 的超时时间。

TiDB 在 v3.0.13 版本中已优化此逻辑，写入 binlog 失败将使事务执行失败返回报错，而不会导致 TiDB 卡住。

11.7.15.25 TiDB 向 Pump 写入了重复的 binlog？

TiDB 在写入 binlog 失败或者超时的情况下，会重试将 binlog 写入到下一个可用的 Pump 节点直到写入成功。所以如果写入到某个 Pump 节点较慢，导致 TiDB 超时（默认 15s），此时 TiDB 判定写入失败并尝试写入下一个 Pump 节点。如果超时的 Pump 节点实际也写入成功，则会出现同一条 binlog 被写入到多个 Pump 节点。Drainer 在处理 binlog 的时候，会自动去重 TSO 相同的 binlog，所以这种重复的写入对下游无感知，不会对同步逻辑产生影响。

11.7.15.26 在使用全量 + 增量方式恢复的过程中，Reparo 中断了，可以使用日志里面最后一个 TSO 恢复同步吗？

可以。Reparo 不会在启动时自动开启 safe-mode 模式，需要手动操作：

1. Reparo 中断后，记录日志中最后一个 TSO，记为 `checkpoint-tso`。
2. 修改 Reparo 配置文件，将配置项 `start-tso` 设为 `checkpoint-tso + 1`，将 `stop-tso` 设为 `checkpoint-tso + 80,000,000,000`（大概是 `checkpoint-tso` 延后 5 分钟），将 `safe-mode` 设置为 `true`。启动 Reparo，Reparo 会将数据同步到 `stop-tso` 后自动停止。
3. Reparo 自动停止后，将 `start-tso` 设置为 `checkpoint tso + 80,000,000,001`，将 `stop-tso` 设置为 `0`，将 `safe-mode` 设为 `false`。启动 Reparo 继续同步。

11.8 TiDB Lightning

11.8.1 TiDB Lightning 简介

TiDB Lightning 是一个将全量数据高速导入到 TiDB 集群的工具，可在此下载。

TiDB Lightning 有以下两个主要的使用场景：

- 迅速导入大量新数据。
- 恢复所有备份数据。

目前，TiDB Lightning 支持：

- 导入Dumpling、CSV 或Amazon Aurora Parquet 输出格式的数据源。
- 从本地盘或Amazon S3 云盘读取数据。

11.8.1.1 TiDB Lightning 整体架构

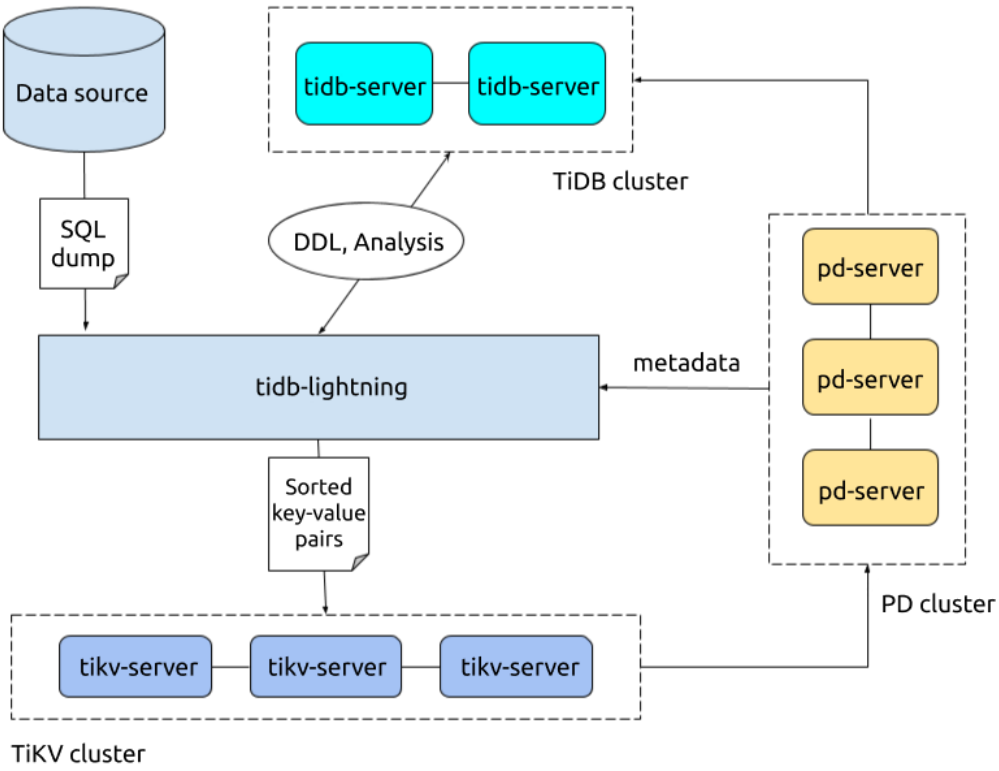


图 162: TiDB Lightning 整体架构

TiDB Lightning 整体工作原理如下：

1. 在导入数据之前，`tidb-lightning` 会自动将 TiKV 集群切换为“导入模式” (import mode)，优化写入效率并停止自动压缩。
2. `tidb-lightning` 会在目标数据库建立架构和表，并获取其元数据。
3. 每张表都会被分割为多个连续的区块，这样来自大表 (200 GB+) 的数据就可以用增量方式并行导入。
4. `tidb-lightning` 会为每一个区块准备一个“引擎文件 (engine file)”来处理键值对。`tidb-lightning` 会并发读取 SQL dump，将数据源转换成与 TiDB 相同编码的键值对，然后将这些键值对排序写入本地临时存储文件中。
5. 当一个引擎文件数据写入完毕时，`tidb-lightning` 便开始对目标 TiKV 集群数据进行分裂和调度，然后导入数据到 TiKV 集群。

   引擎文件包含两种：数据引擎与索引引擎，各自又对应两种键值对：行数据和次级索引。通常行数据在数据源里是完全有序的，而次级索引是无序的。因此，数据引擎文件在对应区块写入完成后会被立即上传，而所有的索引引擎文件只有在整张表所有区块编码完成后才会执行导入。

6. 整张表相关联的所有引擎文件完成导入后，tidb-lightning 会对比本地数据源及下游集群的校验和 (checksum)，确保导入的数据无损，然后让 TiDB 分析 (ANALYZE) 这些新增的数据，以优化日后的操作。同时，tidb-lightning 调整 AUTO_INCREMENT 值防止之后新增数据时发生冲突。

   表的自增 ID 是通过行数的上界估计值得到的，与表的数据文件总大小成正比。因此，最后的自增 ID 通常比实际行数大得多。这属于正常现象，因为在 TiDB 中自增 ID 不一定是连续分配的。

7. 在所有步骤完毕后，tidb-lightning 自动将 TiKV 切换回“普通模式” (normal mode)，此后 TiDB 集群可以正常对外提供服务。

如果需要导入的目标集群是 v3.x 或以下的版本，需要使用 Importer-backend 来完成数据的导入。在这个模式下，tidb-lightning 需要将解析的键值对通过 gRPC 发送给 tikv-importer 并由 tikv-importer 完成数据的导入；TiDB Lightning 还支持使用 TiDB-backend 作为后端导入数据。TiDB-backend 使用和 Loader 类似，tidb-lightning 将数据转换为 INSERT 语句，然后直接在目标集群上执行这些语句。详见TiDB Lightning Backends。

11.8.1.2 使用限制

TiDB Lightning 与 TiFlash 一起使用时需要注意：

- 如果集群版本小于 v4.0.6，若先对表创建 TiFlash 副本，再使用 TiDB Lightning 导入数据，会导致数据导入失败。需要在使用 TiDB Lightning 成功导入数据至表后，再对相应的表创建 TiFlash 副本。
- 如果集群版本以及 TiDB Lightning 版本均大于等于 v4.0.6，无论是否已为一张表创建 TiFlash 副本，你均可以使用 TiDB Lightning 导入数据至该表。但该场景下 TiDB Lightning 导入数据耗费的时间更长，具体取决于 TiDB Lightning 部署机器的网卡带宽、TiFlash 节点的 CPU 及磁盘负载、TiFlash 副本数等因素。

11.8.2 TiDB Lightning 教程

TiDB Lightning 是一个将全量数据高速导入到 TiDB 集群的工具，目前支持 SQL 或 CSV 输出格式的数据源。你可以在以下两种场景下使用 TiDB Lightning：

- 迅速导入大量新数据。
- 备份恢复所有数据。

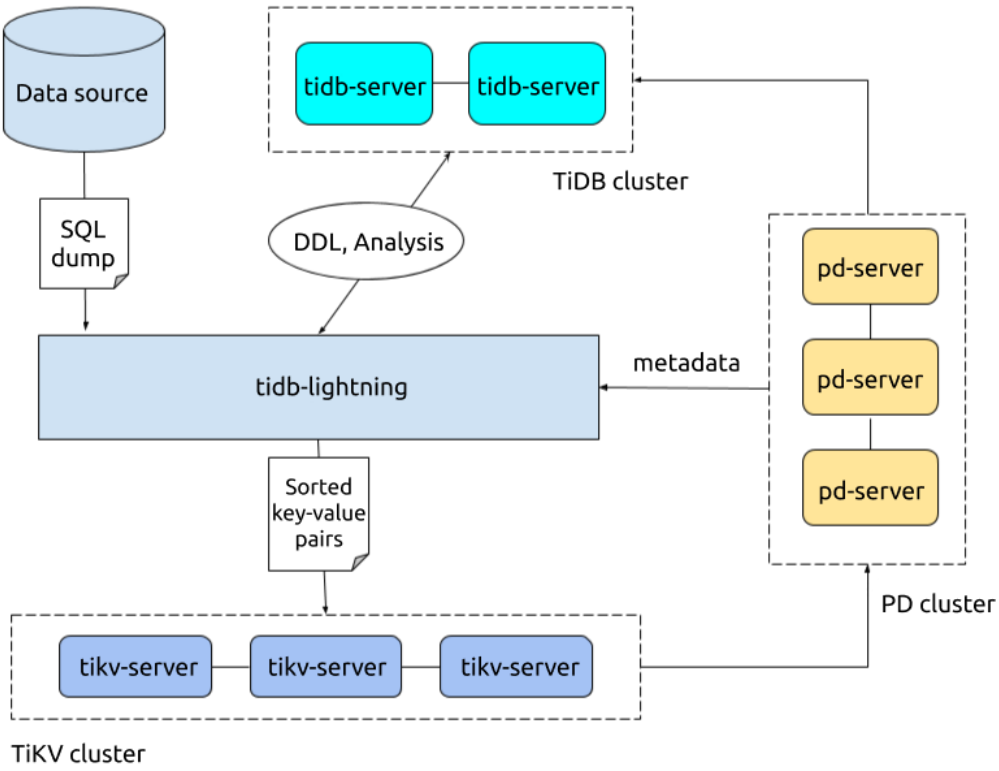


图 163: TiDB Lightning 整体架构

本教程假设使用的是若干新的、纯净版 CentOS 7 实例，你可以（使用 VMware、VirtualBox 及其他工具）在本地虚拟化或在供应商提供的平台上部署一台小型的云虚拟主机。因为 TiDB Lightning 对计算机资源消耗较高，建议分配 16 GB 以上的内存以及 32 核以上的 CPU 以获取最佳性能。

> **警告：**
>
> 本教程中的部署方法只适用于测试及功能体验，并不适用于生产或开发环境。

11.8.2.1 准备全量备份数据

我们使用dumpling 从 MySQL 导出数据，如下：

```
./bin/dumpling -h 127.0.0.1 -P 3306 -u root -t 16 -F 256MB -B test -f 'test.t[12]' -o /data/
    ↪ my_database/
```

其中：

- `-B test`：从 `test` 数据库导出。
- `-f test.t[12]`：只导出 `test.t1` 和 `test.t2` 这两个表。
- `-t 16`：使用 16 个线程导出数据。
- `-F 256MB`：将每张表切分成多个文件，每个文件大小约为 256 MB。

这样全量备份数据就导出到了 `/data/my_database` 目录中。

11.8.2.2 部署 TiDB Lightning

11.8.2.2.1 第 1 步：部署 TiDB 集群

在开始数据导入之前，需先部署一套要进行导入的 TiDB 集群（版本要求 2.0.9 以上），本教程使用 TiDB 4.0.3 版本。部署方法可参考 TiDB 部署方式。

11.8.2.2.2 第 2 步：下载 TiDB Lightning 安装包

通过以下链接获取 TiDB Lightning 安装包（选择与 TiDB 集群相同的版本）：

- v4.0.3: tidb-toolkit-v4.0.3-linux-amd64.tar.gz

11.8.2.2.3 第 3 步：启动 `tidb-lightning`

1. 将安装包里的 `bin/tidb-lightning` 及 `bin/tidb-lightning-ctl` 上传至部署 TiDB Lightning 的服务器。
2. 将数据源也上传到同样的服务器。
3. 配置 `tidb-lightning.toml`。

```
[lightning]
# 日志
level = "info"
file = "tidb-lightning.log"

[tikv-importer]
# 选择使用的 local 后端
backend = "local"
# 设置排序的键值对的临时存放地址，目标路径需要是一个空目录
sorted-kv-dir = "/mnt/ssd/sorted-kv-dir"

[mydumper]
# 源数据目录。
data-source-dir = "/data/my_datasource/"

# 配置通配符规则，默认规则会过滤 mysql、 sys、 INFORMATION_SCHEMA、 PERFORMANCE_SCHEMA、
    ↪ METRICS_SCHEMA、 INSPECTION_SCHEMA 系统数据库下的所有表
# 若不配置该项，导入系统表时会出现“找不到 schema”的异常
```

```
filter = ['*.*', '!mysql.*', '!sys.*', '!INFORMATION_SCHEMA.*', '!PERFORMANCE_SCHEMA.*', '!
    ↪ METRICS_SCHEMA.*', '!INSPECTION_SCHEMA.*']
[tidb]
# 目标集群的信息
host = "172.16.31.2"
port = 4000
user = "root"
password = "rootroot"
# 表架构信息在从 TiDB 的“状态端口”获取。
status-port = 10080
# 集群 pd 的地址
pd-addr = "172.16.31.3:2379"
```

4. 配置合适的参数运行 tidb-lightning。如果直接在命令行中用 nohup 启动程序，可能会因为 SIGHUP 信号而退出，建议把 nohup 放到脚本里面，如：

```
#!/bin/bash
nohup ./tidb-lightning -config tidb-lightning.toml > nohup.out &
```

11.8.2.2.4 第 4 步：检查数据

导入完毕后，TiDB Lightning 会自动退出。若导入成功，日志的最后一行会显示 tidb lightning exit。

如果出错，请参见TiDB Lightning 常见问题。

11.8.2.3 总结

本教程对 TiDB Lightning 进行了简单的介绍，并快速部署了一套简单的 TiDB Lightning 集群，将全量备份数据导入到 TiDB 集群中。

关于 TiDB Lightning 的详细功能和使用，参见TiDB Lightning 简介。

11.8.3 TiDB Lightning 部署与执行

本文主要介绍 TiDB Lightning 使用 Local-backend 进行数据导入的硬件需求，以及使用 TiDB Ansible 部署与手动部署 TiDB Lightning 这两种部署方式。

如果使用 Local-backend 进行数据导入，TiDB Lightning 运行后，TiDB 集群将无法正常对外提供服务。如果你不希望 TiDB 集群的对外服务受到影响，可以参考TiDB Lightning TiDB-backend 中的硬件需求与部署方式进行数据导入。

11.8.3.1 注意事项

在使用 TiDB Lightning 前，需注意以下事项：

- 若 tidb-lightning 崩溃，集群会留在“导入模式”。若忘记转回“普通模式”，集群会产生大量未压缩的文件，继而消耗 CPU 并导致延迟。此时，需要使用 tidb-lightning-ctl 手动将集群转回“普通模式”：

```
bin/tidb-lightning-ctl --switch-mode=normal
```

- TiDB Lightning 需要下游 TiDB 有如下权限：

权限	作用域
SELECT	Tables
INSERT	Tables
UPDATE	Tables
DELETE	Tables
CREATE	Databases, tables
DROP	Databases, tables
ALTER	Tables

如果配置项 `checksum = true`，则 TiDB Lightning 需要有下游 TiDB admin 用户权限。

11.8.3.2 硬件需求

`tidb-lightning`为资源密集程序，为了优化效能，建议硬件配置如下：

- 32+ 逻辑核 CPU
- 20GB+ 内存
- 足够储存整个数据源的 SSD 硬盘，读取速度越快越好
- 使用万兆网卡，带宽需要 1GB/s 以上
- 运行过程默认会占满 CPU，建议单独部署。条件不允许的情况下可以和其他组件（比如 `tikv-server`）部署在同一台机器上，然后通过配置 `region-concurrency` 限制 `tidb-lightning` 使用 CPU 资源。

> 注意：
>
> - `tidb-lightning` 是 CPU 密集型程序，如果和其它程序混合部署，需要通过 `region-concurrency` 限制 `tidb-lightning` 的 CPU 实际占用核数，否则会影响其他程序的正常运行。建议将混合部署机器上 75% 的 CPU 资源分配给 `tidb-lightning`。例如，机器为 32 核，则 `tidb-lightning` 的 `region-concurrency` 可设为 “24”。

此外，目标 TiKV 集群必须有足够空间接收新导入的数据。除了标准硬件配置以外，目标 TiKV 集群的总存储空间必须大于 数据源大小 × 副本数量 × 2。例如集群默认使用 3 副本，那么总存储空间需为数据源大小的 6 倍以上。

11.8.3.3 导出数据

使用dumpling 从 MySQL 导出数据，如下：

```
./bin/dumpling -h 127.0.0.1 -P 3306 -u root -t 16 -F 256MB -B test -f 'test.t[12]' -o /data/
    ↪ my_database/
```

其中：

- `-B test`：从 `test` 数据库导出。
- `-f test.t[12]`：只导出 `test.t1` 和 `test.t2` 这两个表。
- `-t 16`：使用 16 个线程导出数据。
- `-F 256MB`：将每张表切分成多个文件，每个文件大小约为 256 MB。

如果数据源是 CSV 文件，请参考CSV 支持获取配置信息。

11.8.3.4 部署 TiDB Lightning

本节介绍 TiDB Lightning 的两种部署方式：使用 TiDB Ansible 部署和手动部署。

11.8.3.4.1 使用 TiDB Ansible 部署 TiDB Lightning

TiDB Lightning 可随 TiDB 集群一起用TiDB Ansible 部署。

1. 编辑 `inventory.ini`，为 `tidb-lightning` 配置一个 IP。

    ```
    ...
    [lightning_server]
    192.168.20.10

    ...
    ```

2. 修改 `group_vars/*.yml` 的变量配置 `tidb-lightning`。

    - `group_vars/lightning_server.yml`

        ```
        ---
        dummy:

        # 提供监控告警的端口。需对监控服务器（monitoring_server）开放。
        tidb_lightning_pprof_port: 8289

        # 获取数据源（Dumpling SQL dump 或 CSV）的路径。
        data_source_dir: "{{ deploy_dir }}/mydumper"
        ```

3. 开始部署。

    ```
    ansible-playbook bootstrap.yml &&
    ansible-playbook deploy.yml
    ```

4. 将数据源写入 data_source_dir 指定的路径。

5. 登录 tidb-lightning 的服务器，编辑 conf/tidb-lighting.toml 如下配置项：

```
[tikv-importer]
# 选择使用 local 模式
backend = "local"
# 设置排序的键值对的临时存放地址，目标路径需要是一个空目录
"sorted-kv-dir" = "/mnt/ssd/sorted-kv-dir"

[tidb]
# pd-server 的地址，填一个即可
pd-addr = "172.16.31.4:2379"
```

6. 登录 tidb-lightning 的服务器，并执行以下命令来启动 Lightning，开始导入过程。

```
scripts/start_lightning.sh
```

11.8.3.4.2 手动部署 TiDB Lightning

第 1 步：部署 TiDB 集群

在开始数据导入之前，需先部署一套要进行导入的 TiDB 集群 (版本要求 2.0.9 以上)，建议使用最新版。部署方法可参考使用 TiUP 部署 TiDB 集群。

第 2 步：下载 TiDB Lightning 安装包

在工具下载页面下载 TiDB Lightning 安装包（需选择与 TiDB 集群相同的版本）。

第 3 步：启动 tidb-lightning

1. 从安装包上传 bin/tidb-lightning 及 bin/tidb-lightning-ctl。

2. 将数据源写入到同样的机器。

3. 配置 tidb-lightning.toml。对于没有出现在下述模版中的配置，TiDB Lightning 给出配置错误的提醒并退出。

   sorted-kv-dir 设置排序的键值对的临时存放地址，目标路径必须是一个空目录，目录空间须大于待导入数据集的大小。

```
[lightning]

# 转换数据的并发数，默认为逻辑 CPU 数量，不需要配置。
# 混合部署的情况下可以配置为逻辑 CPU 的 75% 大小。
# region-concurrency =

# 日志
level = "info"
file = "tidb-lightning.log"
```

```
[tikv-importer]
# backend 设置为 local 模式
backend = "local"
# 设置本地临时存储路径
sorted-kv-dir = "/mnt/ssd/sorted-kv-dir"

[mydumper]
# Mydumper 源数据目录。
data-source-dir = "/data/my_database"

[tidb]
# 目标集群的信息。tidb-server 的监听地址，填一个即可。
host = "172.16.31.1"
port = 4000
user = "root"
password = ""
# 表架构信息在从 TiDB 的“状态端口”获取。
status-port = 10080
# pd-server 的地址，填一个即可
pd-addr = "172.16.31.4:2379"
```

上面仅列出了 tidb-lightning 的基本配置信息。完整配置信息请参考tidb-lightning 配置说明。

4. 运行 tidb-lightning。如果直接在命令行中用 nohup 启动程序，可能会因为 SIGHUP 信号而退出，建议把 nohup 放到脚本里面，如：

```
#!/bin/bash
nohup ./tidb-lightning -config tidb-lightning.toml > nohup.out &
```

11.8.3.5 升级 TiDB Lightning

你可以通过替换二进制文件升级 TiDB Lightning，无需其他配置。重启 TiDB Lightning 的具体操作参见FAQ。

如果当前有运行的导入任务，推荐任务完成后再升级 TiDB Lightning。否则，你可能需要从头重新导入，因为无法保证断点可以跨版本工作。

11.8.4 TiDB Lightning 配置参数

你可以使用配置文件或命令行配置 TiDB Lightning。本文主要介绍 TiDB Lightning 的全局配置、任务配置和 TiKV Importer 的配置，以及如何使用命令行进行参数配置。

11.8.4.1 配置文件

TiDB Lightning 的配置文件分为“全局”和“任务”两种类别，二者在结构上兼容。只有当服务器模式开启时，全局配置和任务配置才会有区别；默认情况下，服务器模式为禁用状态，此时 TiDB Lightning 只会执行一个任务，且全局和任务配置使用同一配置文件。

11.8.4.1.1 TiDB Lightning 全局配置

```
##### tidb-lightning 全局配置

[lightning]
### 用于进度展示 web 界面、拉取 Prometheus 监控项、暴露调试数据和提交导入任务（服务器模式下）的
    ↪ HTTP 端口。设置为 0 时为禁用状态。
status-addr = ':8289'

### 服务器模式，默认值为 false，命令启动后会开始导入任务。如果改为 true，命令启动后会等待用户在
    ↪ web 界面上提交任务。
### 详情参见“TiDB Lightning web 界面”文档
server-mode = false

### 日志
level = "info"
file = "tidb-lightning.log"
max-size = 128 # MB
max-days = 28
max-backups = 14
```

11.8.4.1.2 TiDB Lightning 任务配置

```
##### tidb-lightning 任务配置

[lightning]
### 启动之前检查集群是否满足最低需求。
### check-requirements = true

### 引擎文件的最大并行数。
### 每张表被切分成一个用于存储索引的“索引引擎”和若干存储行数据的“数据引擎”。
### 这两项设置控制两种引擎文件的最大并发数。
### 这两项设置的值会影响 tikv-importer 的内存和磁盘用量。
### 两项数值之和不能超过 tikv-importer 的 max-open-engines 的设定。
index-concurrency = 2
table-concurrency = 6

### 数据的并发数。默认与逻辑 CPU 的数量相同。
### 混合部署的情况下可以将其大小配置为逻辑 CPU 数的 75%，以限制 CPU 的使用。
### region-concurrency =

### I/O 最大并发数。I/O 并发量太高时，会因硬盘内部缓存频繁被刷新
### 而增加 I/O 等待时间，导致缓存未命中和读取速度降低。
### 对于不同的存储介质，此参数可能需要调整以达到最佳效率。
io-concurrency = 5
```

```
[security]
### 指定集群中用于 TLS 连接的证书和密钥。
### CA 的公钥证书。如果留空，则禁用 TLS。
### ca-path = "/path/to/ca.pem"
### 此服务的公钥证书。
### cert-path = "/path/to/lightning.pem"
### 该服务的密钥。
### key-path = "/path/to/lightning.key"

[checkpoint]
### 是否启用断点续传。
### 导入数据时，TiDB Lightning 会记录当前表导入的进度。
### 所以即使 TiDB Lightning 或其他组件异常退出，在重启时也可以避免重复再导入已完成的数据。
enable = true
### 存储断点的数据库名称。
schema = "tidb_lightning_checkpoint"
### 存储断点的方式。
###  - file：存放在本地文件系统。
###  - mysql：存放在兼容 MySQL 的数据库服务器。
driver = "file"

### dsn 是数据源名称 (data source name)，表示断点的存放位置。
### 若 driver = "file"，则 dsn 为断点信息存放的文件路径。
#若不设置该路径，则默认存储路径为“/tmp/CHECKPOINT_SCHEMA.pb”。
### 若 driver = "mysql"，则 dsn 为“用户:密码@tcp(地址:端口)/”格式的 URL。
### 若不设置该 URL，则默认会使用 [tidb] 部分指定的 TiDB 服务器来存储断点。
### 为减少目标 TiDB 集群的压力，建议指定另一台兼容 MySQL 的数据库服务器来存储断点。
### dsn = "/tmp/tidb_lightning_checkpoint.pb"

### 所有数据导入成功后是否保留断点。设置为 false 时为删除断点。
### 保留断点有利于进行调试，但会泄漏关于数据源的元数据。
### keep-after-success = false

[tikv-importer]
### 选择后端：“importer” 或 “local” 或 “tidb”
### "local"：通过在本地排序生成 SST 文件的方式导入数据，适用于快速导入大量数据，但导入期间下游
    ↪ TiDB 无法对外提供服务，并且导入的目标表必须为空。
### "importer"：和 “local“ 原理类似，但需要额外部署 “tikv-importer“ 组件。如无特殊情况，
    ↪ 推荐使用 “local” 后端。
### "tidb"：通过执行 SQL 语句的方式导入数据，速度较慢，但导入期间下游 TiDB 可正常提供服务，
    ↪ 导入的目标表可以不为空。
### backend = "local"
### 当后端是 “importer” 时，tikv-importer 的监听地址（需改为实际地址）。
addr = "172.16.31.10:8287"
```

```
### 当后端是 “tidb” 时，插入重复数据时执行的操作。
### - replace：新数据替代已有数据
### - ignore：保留已有数据，忽略新数据
### - error：中止导入并报错
### on-duplicate = "replace"
### 当后端是 “local” 时，控制生成 SST 文件的大小，最好跟 TiKV 里面的 Region 大小保持一致，
    ↪ 默认是 96 MB。
### region-split-size = 100_663_296
### 当后端是 “local” 时，一次请求中发送的 KV 数量。
### send-kv-pairs = 32768
### 当后端是 “local” 时，本地进行 KV 排序的路径。如果磁盘性能较低（如使用机械盘），建议设置成与
    ↪ `data-source-dir` 不同的磁盘，这样可有效提升导入性能。
### sorted-kv-dir = ""
### 当后端是 “local” 时，TiKV 写入 KV 数据的并发度。当 TiDB Lightning 和 TiKV
    ↪ 直接网络传输速度超过万兆的时候，可以适当增加这个值。
### range-concurrency = 16

[mydumper]
### 设置文件读取的区块大小，确保该值比数据源的最长字符串长。
read-block-size = "64KiB" # 默认值

### （源数据文件）单个导入区块大小的最小值。
### TiDB Lightning 根据该值将一张大表分割为多个数据引擎文件。
### batch-size = 107_374_182_400 # Byte（默认为 100 GB）

### 引擎文件需按顺序导入。由于并行处理，多个数据引擎几乎在同时被导入，
### 这样形成的处理队列会造成资源浪费。因此，为了合理分配资源，TiDB Lightning
### 稍微增大了前几个区块的大小。该参数也决定了比例系数，即在完全并发下
### “导入”和“写入”过程的持续时间比。这个值可以通过计算 1 GiB 大小的
### 单张表的（导入时长/写入时长）得到。在日志文件中可以看到精确的时间。
### 如果“导入”更快，区块大小的差异就会更小；比值为 0 时则说明区块大小一致。
### 取值范围为（0 <= batch-import-ratio < 1）。
batch-import-ratio = 0.75

### 本地源数据目录或外部存储 URL
data-source-dir = "/data/my_database"
### 如果 no-schema = true，那么 TiDB Lightning 假设目标 TiDB 集群上
### 已有表结构，并且不会执行 `CREATE TABLE` 语句。
no-schema = false
### 指定包含 `CREATE TABLE` 语句的表结构文件的字符集。只支持下列选项：
###  - utf8mb4：表结构文件必须使用 UTF-8 编码，否则会报错。
###  - gb18030：表结构文件必须使用 GB-18030 编码，否则会报错。
###  - auto：自动判断文件编码是 UTF-8 还是 GB-18030，两者皆非则会报错（默认）。
###  - binary：不尝试转换编码。
### 注意：**数据** 文件始终解析为 binary 文件。
```

```
character-set = "auto"

### “严格”格式的导入数据可加快处理速度。
### strict-format = true 要求：
### 在 CSV 文件的所有记录中，每条数据记录的值不可包含字符换行符（U+000A 和 U+000D，即 \r 和 \n）
### 甚至被引号包裹的字符换行符都不可包含，即换行符只可用来分隔行。
### 导入数据源为严格格式时，TiDB Lightning 会快速定位大文件的分割位置进行并行处理。
### 但是如果输入数据为非严格格式，可能会将一条完整的数据分割成两部分，导致结果出错。
### 为保证数据安全而非追求处理速度，默认值为 false。
strict-format = false

### 如果 strict-format = true，TiDB Lightning 会将 CSV 大文件分割为多个文件块进行并行处理。max-
    ↪ region-size 是分割后每个文件块的最大大小。
### max-region-size = "256MiB" # 默认值

### 只导入与该通配符规则相匹配的表。详情见相应章节。
filter = ['*.*']

### 配置 CSV 文件的解析方式。
[mydumper.csv]
### 字段分隔符，应为单个 ASCII 字符。
separator = ','
### 引用定界符，可为单个 ASCII 字符或空字符串。
delimiter = '"'
### CSV 文件是否包含表头。
### 如果 header = true，将跳过首行。
header = true
### CSV 文件是否包含 NULL。
### 如果 not-null = true，CSV 所有列都不能解析为 NULL。
not-null = false
### 如果 not-null = false（即 CSV 可以包含 NULL），
### 为以下值的字段将会被解析为 NULL。
null = '\N'
### 是否对字段内“\“进行转义
backslash-escape = true
### 如果有行以分隔符结尾，删除尾部分隔符。
trim-last-separator = false

[tidb]
### 目标集群的信息。tidb-server 的地址，填一个即可。
host = "172.16.31.1"
port = 4000
user = "root"
password = ""
### 表结构信息从 TiDB 的“status-port”获取。
```

```
status-port = 10080
### pd-server 的地址，填一个即可。
pd-addr = "172.16.31.4:2379"
### tidb-lightning 引用了 TiDB 库，并生成产生一些日志。
### 设置 TiDB 库的日志等级。
log-level = "error"

### 设置 TiDB 会话变量，提升 Checksum 和 Analyze 的速度。
### 各参数定义可参阅”控制 Analyze 并发度“文档
build-stats-concurrency = 20
distsql-scan-concurrency = 100
index-serial-scan-concurrency = 20
checksum-table-concurrency = 16

### 解析和执行 SQL 语句的默认 SQL 模式。
sql-mode = "ONLY_FULL_GROUP_BY,NO_ENGINE_SUBSTITUTION"
### `max-allowed-packet` 设置数据库连接允许的最大数据包大小，
### 对应于系统参数中的 `max_allowed_packet`。 如果设置为 0，
### 会使用下游数据库 global 级别的 `max_allowed_packet`。
max-allowed-packet = 67_108_864

### SQL 连接是否使用 TLS。可选值为：
###  * ""            - 如果填充了 [tidb.security] 部分，则强制使用 TLS（与 "cluster" 情况相同），
    ↪ 否则与 "false" 情况相同
###  * "false"       - 禁用 TLS
###  * "cluster"     - 强制使用 TLS 并使用 [tidb.security] 部分中指定的 CA 验证服务器的证书
###  * "skip-verify" - 强制使用 TLS，但不验证服务器的证书（不安全！）
###  * "preferred"   - 与 "skip-verify" 相同，但是如果服务器不支持 TLS，则会退回到未加密的连接
### tls = ""
### 指定证书和密钥用于 TLS 连接 MySQL。
### 默认为 [security] 部分的副本。
### [tidb.security]
### CA 的公钥证书。设置为空字符串可禁用 SQL 的 TLS。
### ca-path = "/path/to/ca.pem"
### 该服务的公钥证书。默认为 `security.cert-path` 的副本
### cert-path = "/path/to/lightning.pem"
### 此服务的私钥。默认为 `security.key-path` 的副本
### key-path = "/path/to/lightning.key"

### 对于 Local Backend 和 Importer Backend 模式，数据导入完成后，TiDB Lightning 可以自动执行
    ↪ Checksum 和 Analyze 操作。
### 在生产环境中，建议总是开启 Checksum 和 Analyze。
### 执行的顺序为：Checksum -> Analyze。
### 注意：对于 TiDB Backend，无须执行这两个阶段，因此在实际运行时总是会直接跳过。
[post-restore]
```

```
### 配置是否在导入完成后对每一个表执行 `ADMIN CHECKSUM TABLE <table>` 操作来验证数据的完整性。
### 可选的配置项：
### - "required"（默认）。在导入完成后执行 CHECKSUM 检查，如果 CHECKSUM 检查失败，则会报错退出。
### - "optional"。在导入完成后执行 CHECKSUM 检查，如果报错，会输出一条 WARN 日志并忽略错误。
### - "off"。导入结束后不执行 CHECKSUM 检查。
### 默认值为 "required"。从 v4.0.8 开始，checksum 的默认值由此前的 "true" 改为 "required"。
### # 注意：
### 1. Checksum 对比失败通常表示导入异常（数据丢失或数据不一致），因此建议总是开启 Checksum。
### 2. 考虑到与旧版本的兼容性，依然可以在本配置项设置 `true` 和 `false` 两个布尔值，其效果与 `
    ↪ required` 和 `off` 相同。
checksum = "required"
### 配置是否在 CHECKSUM 结束后对所有表逐个执行 `ANALYZE TABLE <table>` 操作。
### 此配置的可选配置项与 `checksum` 相同，但默认值为 "optional"。
analyze = "optional"

### 设置周期性后台操作。
### 支持的单位：h（时）、m（分）、s（秒）。
[cron]
### TiDB Lightning 自动刷新导入模式状态的持续时间，该值应小于 TiKV 对应的设定值。
switch-mode = "5m"
### 在日志中打印导入进度的持续时间。
log-progress = "5m"
```

11.8.4.1.3 TiKV Importer 配置参数

```
### TiKV Importer 配置文件模版

### 日志文件
log-file = "tikv-importer.log"
### 日志等级：trace, debug, info, warn, error 和 off
log-level = "info"

### 状态服务器的监听地址。
### Prometheus 可以从这个地址抓取监控指标。
status-server-address = "0.0.0.0:8286"

[server]
### tikv-importer 的监听地址，tidb-lightning 需要连到这个地址进行数据写入。
addr = "0.0.0.0:8287"
### gRPC 服务器的线程池大小。
grpc-concurrency = 16

[metric]
### 当使用 Prometheus Pushgateway 时会涉及相关设置。通常可以通过 Prometheus 从
    ↪ 状态服务器地址中抓取指标。
```

```
### 给 Prometheus 客户端推送的 job 名称。
job = "tikv-importer"
### 给 Prometheus 客户端推送的间隔。
interval = "15s"
### Prometheus Pushgateway 的地址。
address = ""

[rocksdb]
### background job 的最大并发数。
max-background-jobs = 32

[rocksdb.defaultcf]
### 数据在刷新到硬盘前能存于内存的容量上限。
write-buffer-size = "1GB"
### 内存中写缓冲器的最大数量。
max-write-buffer-number = 8

### 各个压缩层级使用的算法。
### 第 0 层的算法用于压缩 KV 数据。
### 第 6 层的算法用于压缩 SST 文件。
### 第 1 至 5 层的算法目前尚未使用。
compression-per-level = ["lz4", "no", "no", "no", "no", "no", "lz4"]

[rocksdb.writecf]
### 同上
compression-per-level = ["lz4", "no", "no", "no", "no", "no", "lz4"]

[security]
### TLS 证书的路径。空字符串表示禁用安全连接。
### ca-path = ""
### cert-path = ""
### key-path = ""

[import]
### 存储引擎文件的文件夹路径
import-dir = "/mnt/ssd/data.import/"
### 处理 RPC 请求的线程数
num-threads = 16
### 导入 job 的并发数。
num-import-jobs = 24
### 预处理 Region 最长时间。
### max-prepare-duration = "5m"
### 把要导入的数据切分为这个大小的 Region。
#region-split-size = "512MB"
### 设置 stream-channel-window 的大小。
```

```
### channel 满了之后 stream 会处于阻塞状态。
### stream-channel-window = 128
### 同时打开引擎文档的最大数量。
max-open-engines = 8
### Importer 上传至 TiKV 的最大速度（字节/秒）。
### upload-speed-limit = "512MB"
### 目标存储可用空间比率（store_available_space/store_capacity）的最小值。
### 如果目标存储空间的可用比率低于该值，Importer 将会暂停上传 SST
### 来为 PD 提供足够时间进行 Regions 负载均衡。
min-available-ratio = 0.05
```

11.8.4.2 命令行参数

11.8.4.2.1 tidb-lightning

使用 tidb-lightning 可以对下列参数进行配置：

参数	描述	对应配置项
–config file	从 file 读取全局设置。如果没有指定则使用默认设置。	
-V	输出程序的版本	
-d directory	读取数据的本地目录或外部存储 URL	mydumper.data-source-dir
-L level	日志的等级：debug、info、warn、error 或 fatal (默认为 info)	lightning.log-level
-f rule	表库过滤的规则 (可多次指定)	mydumper.filter
–backend backend	选择后端的模式：importer、local 或 tidb	tikv-importer.backend
–log-file file	日志文件路径（默认是 /tmp 中的临时文件）	lightning.log-file
–status-addr ip:port	TiDB Lightning 服务器的监听地址	lightning.status-port
–importer host:port	TiKV Importer 的地址	tikv-importer.addr
–pd-urls host:port	PD endpoint 的地址	tidb.pd-addr
–tidb-host host	TiDB Server 的 host	tidb.host
–tidb-port port	TiDB Server 的端口（默认为 4000）	tidb.port
–tidb-status port	TiDB Server 的状态端口的（默认为 10080）	tidb.status-port
–tidb-user user	连接到 TiDB 的用户名	tidb.user
–tidb-password password	连接到 TiDB 的密码	tidb.password
–no-schema	忽略表结构文件，直接从 TiDB 中获取表结构信息	mydumper.no-schema
–enable-checkpoint bool	是否启用断点 (默认值为 true)	checkpoint.enable
–analyze level	导入后分析表信息，可选值为 required、optional（默认值）、off	post-restore.analyze
–checksum level	导入后比较校验和，可选值为 required（默认值）、optional、off	post-restore.checksum
–check-requirements bool	开始之前检查集群版本兼容性（默认值为 true）	lightning.check-requirements
–ca file	TLS 连接的 CA 证书路径	security.ca-path
–cert file	TLS 连接的证书路径	security.cert-path
–key file	TLS 连接的私钥路径	security.key-path
–server-mode	在服务器模式下启动 TiDB Lightning	lightning.server-mode

如果同时对命令行参数和配置文件中的对应参数进行更改，命令行参数将优先生效。例如，在 `cfg.toml` 文件中，不管对日志等级做出什么修改，运行 `./tidb-lightning -L debug --config cfg.toml` 命令总是将日志级别设置为 “debug”。

11.8.4.2.2 `tidb-lightning-ctl`

使用 `tidb-lightning-ctl` 可以对下列参数进行配置：

参数	描述
–compact	执行 full compact
–switch-mode mode	将每个 TiKV Store 切换到指定模式（normal 或 import）
–fetch-mode	打印每个 TiKV Store 的当前模式
–import-engine uuid	将 TiKV Importer 上关闭的引擎文件导入到 TiKV 集群
–cleanup-engine uuid	删除 TiKV Importer 上的引擎文件
–checkpoint-dump folder	将当前的断点以 CSV 格式存储到文件夹中
–checkpoint-error-destroy tablename	删除断点，如果报错则删除该表
–checkpoint-error-ignore tablename	忽略指定表中断点的报错
–checkpoint-remove tablename	无条件删除表的断点

tablename 必须是`db`.`tbl` 中的限定表名（包括反引号），或关键词 `all`。

此外，上表中所有 `tidb-lightning` 的参数也适用于 `tidb-lightning-ctl`。

11.8.4.2.3 `tikv-importer`

使用 `tikv-importer` 可以对下列参数进行配置：

参数	描述	对应配置项
-C, –config file	从 file 读取配置。如果没有指定，则使用默认设置	
-V, –version	输出程序的版本	
-A, –addr ip:port	TiKV Importer 服务器的监听地址	`server.addr`
–status-server ip:port	状态服务器的监听地址	`status-server-address`
–import-dir dir	引擎文件的存储目录	`import.import-dir`
–log-level level	日志的等级：trace、debug、info、warn、error 或 off	`log-level`
–log-file file	日志文件路径	`log-file`

11.8.5 主要功能

11.8.5.1 TiDB Lightning 断点续传

大量的数据导入一般耗时数小时至数天，长时间运行的进程会有一定机率发生非正常中断。如果每次重启都从头开始，就会浪费掉之前已成功导入的数据。为此，TiDB Lightning 提供了 “断点续传” 的功能，即使 `tidb-lightning` 崩溃，在重启时仍然接着之前的进度继续工作。

本文主要介绍 TiDB Lightning 断点续传的启用与配置、断点的存储，以及断点续传的控制。

11.8.5.1.1 断点续传的启用与配置

```
[checkpoint]
#### 启用断点续传。
#### 导入时，TiDB Lightning 会记录当前进度。
#### 若 TiDB Lightning 或其他组件异常退出，在重启时可以避免重复再导入已完成的数据。
enable = true

#### 存储断点的方式
####  - file：存放在本地文件系统（要求 v2.1.1 或以上）
####  - mysql：存放在兼容 MySQL 的数据库服务器
driver = "file"

#### 存储断点的架构名称（数据库名称）
#### 仅在 driver = "mysql" 时生效
#### schema = "tidb_lightning_checkpoint"

#### 断点的存放位置
#### # 若 driver = "file"，此参数为断点信息存放的文件路径。
#### 如果不设置该参数则默认为 `/tmp/CHECKPOINT_SCHEMA.pb`
#### # 若 driver = "mysql"，此参数为数据库连接参数（DSN），格式为“用户:密码@tcp(地址:端口)/”。
#### 默认会重用 [tidb] 设置目标数据库来存储断点。
#### 为避免加重目标集群的压力，建议另外使用一个兼容 MySQL 的数据库服务器。
#### dsn = "/tmp/tidb_lightning_checkpoint.pb"

#### 导入成功后是否保留断点。默认为删除。
#### 保留断点可用于调试，但有可能泄漏数据源的元数据。
#### keep-after-success = false
```

11.8.5.1.2 断点的存储

TiDB Lightning 支持两种存储方式：本地文件或 MySQL 数据库。

- 若 `driver = "file"`，断点会存放在一个本地文件，其路径由 dsn 参数指定。由于断点会频繁更新，建议将这个文件放到写入次数不受限制的盘上，例如 RAM disk。
- 若 `driver = "mysql"`，断点可以存放在任何兼容 MySQL 5.7 或以上的数据库中，包括 MariaDB 和 TiDB。在没有选择的情况下，默认会存在目标数据库里。

目标数据库在导入期间会有大量的操作，若使用目标数据库来存储断点会加重其负担，甚至有可能造成通信超时丢失数据。因此，强烈建议另外部署一台兼容 MySQL 的临时数据库服务器。此数据库也可以安装在 `tidb-lightning` 的主机上。导入完毕后可以删除。

11.8.5.1.3 断点续传的控制

若 tidb-lightning 因不可恢复的错误而退出（例如数据出错），重启时不会使用断点，而是直接报错离开。为保证已导入的数据安全，这些错误必须先解决掉才能继续。使用 tidb-lightning-ctl 工具可以标示已经恢复。

--checkpoint-error-destroy

```
tidb-lightning-ctl --checkpoint-error-destroy='`schema`.`table`'
```

该命令会让失败的表从头开始整个导入过程。选项中的架构和表名必须以反引号 (`) 包裹，而且区分大小写。

- 如果导入 `schema`.`table` 这个表曾经出错，这条命令会：
    1. 从目标数据库移除 (DROP) 这个表，清除已导入的数据。
    2. 将断点重设到 “未开始” 的状态。
- 如果 `schema`.`table` 没有出错，则无操作。

传入 “all” 会对所有表进行上述操作。这是最方便、安全但保守的断点错误解决方法：

```
tidb-lightning-ctl --checkpoint-error-destroy=all
```

--checkpoint-error-ignore

```
tidb-lightning-ctl --checkpoint-error-ignore='`schema`.`table`' &&
tidb-lightning-ctl --checkpoint-error-ignore=all
```

如果导入 `schema`.`table` 这个表曾经出错，这条命令会清除出错状态，如同没事发生过一样。传入 “all” 会对所有表进行上述操作。

> **注意：**
>
> 除非确定错误可以忽略，否则不要使用这个选项。如果错误是真实的话，可能会导致数据不完全。启用校验和 (CHECKSUM) 可以防止数据出错被忽略。

--checkpoint-remove

```
tidb-lightning-ctl --checkpoint-remove='`schema`.`table`' &&
tidb-lightning-ctl --checkpoint-remove=all
```

无论是否有出错，把表的断点清除。

--checkpoint-dump

```
tidb-lightning-ctl --checkpoint-dump=output/directory
```

将所有断点备份到传入的文件夹，主要用于技术支持。此选项仅于 driver = "mysql" 时有效。

11.8.5.2 表库过滤

TiDB 生态工具默认情况下作用于所有数据库，但实际使用中，往往只需要作用于其中的部分子集。例如，用户只想处理 `foo*` 和 `bar*` 形式的表，而无需对其他表进行操作。

从 TiDB 4.0 起，所有 TiDB 生态系统工具都使用一个通用的过滤语法来定义子集。本文档介绍如何使用表库过滤功能。

11.8.5.2.1 使用表库过滤

命令行

在命令行中使用多个 `-f` 或 `--filter` 参数，即可在 TiDB 生态工具中应用表库过滤规则。每个过滤规则均采用 `db.table` 形式，支持通配符（详情见下一节）。以下为各个工具中的使用示例：

- BR：

```
./br backup full -f 'foo*.*' -f 'bar*.*' -s 'local:///tmp/backup'
#                ^~~~~~~~~~~~~~~~~~~~~~
./br restore full -f 'foo*.*' -f 'bar*.*' -s 'local:///tmp/backup'
#                 ^~~~~~~~~~~~~~~~~~~~~~
```

- Dumpling：

```
./dumpling -f 'foo*.*' -f 'bar*.*' -P 3306 -o /tmp/data/
#          ^~~~~~~~~~~~~~~~~~~~~~
```

- TiDB Lightning：

```
./tidb-lightning -f 'foo*.*' -f 'bar*.*' -d /tmp/data/ --backend tidb
#                ^~~~~~~~~~~~~~~~~~~~~~
```

TOML 配置文件

在 TOML 文件中，表库过滤规则以字符串数组的形式指定。以下为各个工具中的使用示例：

- TiDB Lightning：

```
[mydumper]
filter = ['foo*.*', 'bar*.*']
```

- TiCDC：

```
[filter]
rules = ['foo*.*', 'bar*.*']

[[sink.dispatchers]]
matcher = ['db1.*', 'db2.*', 'db3.*']
dispatcher = 'ts'
```

11.8.5.2.2 表库过滤语法

直接使用表名

每条表库过滤规则由“库”和“表”组成，两部分之间以英文句号 (.) 分隔。只有表名与规则完全相符的表才会被接受。

```
db1.tbl1
db2.tbl2
db3.tbl3
```

表名只由有效的标识符组成，例如：

- 数字（0 到 9）
- 字母（a 到 z，A 到 Z）
- $
- _
- 非 ASCII 字符（U+0080 到 U+10FFFF）

其他 ASCII 字符均为保留字。部分标点符号有特殊含义，详情见下一节。

使用通配符

表名的两个部分均支持使用通配符（详情见 fnmatch(3)）。

- *：匹配零个或多个字符。
- ?：匹配一个字符。
- [a-z]：匹配“a”和“z”之间的一个字符。
- [!a-z]：匹配不在“a”和“z”之间的一个字符。

```
db[0-9].tbl[0-9a-f][0-9a-f]
data.*
*.backup_*
```

此处，“字符”指的是一个 Unicode 码位，例如：

- U+00E9 (é) 是 1 个字符。
- U+0065 U+0301 (é) 是 2 个字符。
- U+1F926 U+1F3FF U+200D U+2640 U+FE0F (🤦🏿‍♀️) 是 5 个字符。

使用文件导入

如需导入一个文件作为过滤规则，请在规则的开头加上一个“@”来指定文件名。库表过滤解析器将导入文件中的每一行都解析为一条额外的过滤规则。

例如，config/filter.txt 文件有以下内容：

```
employees.*
*.WorkOrder
```

以下两条表库过滤命令是等价的：

```
./dumpling -f '@config/filter.txt'
./dumpling -f 'employees.*' -f '*.WorkOrder'
```

导入的文件里不能使用过滤规则导入另一个文件。

注释与空行

导入的过滤规则文件中，每一行开头和结尾的空格都会被去除。此外，空行（空字符串）也将被忽略。

行首的 # 表示该行是注释，会被忽略。而不在行首的 # 则会被认为是语法错误。

```
#### 这是一行注释
db.table   # 这一部分不是注释，且可能引起错误
```

排除规则

在一条过滤规则的开头加上 !，则表示符合这条规则的表不会被 TiDB 生态工具处理。通过应用排除规则，库表过滤可以作为屏蔽名单来使用。

```
*.*
#^ 注意：必须先添加 *.* 规则来包括所有表
!*.Password
!employees.salaries
```

转义字符

如果需要将特殊字符转化为标识符，可以在特殊字符前加上反斜杠 \。

```
db\.with\.dots.*
```

为了简化语法并向上兼容，不支持下列字符序列：

- 在行尾去除空格后使用 \（使用 [ ] 来匹配行尾的空格）。
- 在 \ 后使用数字或字母（[0-9a-zA-Z]）。特别是类似 C 的转义序列，如 \0、\r、\n、\t 等序列，目前在表库过滤规则中无意义。

引号包裹的标识符

除了 \ 之外，还可以用 " 和 ` 来控制特殊字符。

```
"db.with.dots"."tbl\1"
`db.with.dots`.`tbl\2`
```

也可以通过输入两次引号，将引号包含在标识符内。

```
"foo""bar".`foo``bar`
#### 等价于：
foo\"bar.foo\`bar
```

用引号包裹的标识符不可以跨越多行。

用引号只包裹标识符的一部分是无效的，例如：

```
"this is "invalid*.*
```

正则表达式

如果你需要使用较复杂的过滤规则，可以将每个匹配模型写为正则表达式，以 `/` 为分隔符：

```
/^db\d{2,}$/./^tbl\d{2,}$/
```

这类正则表示使用 Go dialect。只要标识符中有一个子字符串与正则表达式匹配，则视为匹配该模型。例如，`/b/` 匹配 `db01`。

> 注意：
>
> 正则表达式中的每一个 `/` 都需要转义为 `\/`，包括在 `[...]` 里面的 `/`。不允许在 `\Q...\E` 之间放置一个未转义的 `/`。

11.8.5.2.3 使用多个过滤规则

当表的名称与过滤列表中所有规则均不匹配时，默认情况下这些表被忽略。

要建立一个屏蔽名单，必须使用显式的 `*.*` 作为第一条过滤规则，否则所有表均被排除。

```
#### 所有表均被过滤掉
./dumpling -f '!*.Password'

#### 只有 “Password” 表被过滤掉，其余表仍保留
./dumpling -f '*.*' -f '!*.Password'
```

如果一个表的名称与过滤列表中的多个规则匹配，则以最后匹配的规则为准。例如：

```
#### rule 1
employees.*
#### rule 2
!*.dep*
#### rule 3
*.departments
```

过滤结果如下：

表名	规则 1	规则 2	规则 3	结果
irrelevant.table				默认（拒绝）
employees.employees	√			规则 1（接受）
employees.dept_emp	√	√		规则 2（拒绝）

表名	规则 1	规则 2	规则 3	结果
employees.departments	√	√	√	规则 3（接受）
else.departments		√	√	规则 3（接受）

注意：

在 TiDB 生态工具的默认配置中，系统库总是被排除。系统库有以下六个：

- INFORMATION_SCHEMA
- PERFORMANCE_SCHEMA
- METRICS_SCHEMA
- INSPECTION_SCHEMA
- mysql
- sys

11.8.5.3 CSV 支持与限制

本文介绍如何使用 TiDB Lightning 从 CSV 文件迁移数据到 TiDB。关于如何从 MySQL 生成 CSV 文件，可以参考使用 Dumpling 导出到 CSV 文件。

TiDB Lightning 支持读取 CSV（逗号分隔值）的数据源，以及其他定界符格式如 TSV（制表符分隔值）。

11.8.5.3.1 文件名

包含整张表的 CSV 文件需命名为 `db_name.table_name.csv`，该文件会被解析为数据库 `db_name` 里名为 `table_name` 的表。

如果一个表分布于多个 CSV 文件，这些 CSV 文件命名需加上文件编号的后缀，如 `db_name.table_name.003.csv`。数字部分不需要连续但必须递增，并用零填充。

文件扩展名必须为 `*.csv`，即使文件的内容并非逗号分隔。

11.8.5.3.2 表结构

CSV 文件是没有表结构的。要导入 TiDB，就必须为其提供表结构。可以通过以下任一方法实现：

- 创建包含 DDL 语句 `CREATE TABLE` 的文件 `db_name.table_name-schema.sql` 以及包含 `CREATE DATABASE` DDL 语句的文件 `db_name-schema-create.sql`。
- 首先在 TiDB 中直接创建空表，然后在 `tidb-lightning.toml` 中设置 `[mydumper] no-schema = true`。

11.8.5.3.3 配置

CSV 格式可在 `tidb-lightning.toml` 文件中 `[mydumper.csv]` 下配置。大部分设置项在 MySQL `LOAD DATA` 语句中都有对应的项目。

```
[mydumper.csv]
#### 字段分隔符，必须为 ASCII 字符。
separator = ','
#### 引用定界符，可以为 ASCII 字符或空字符。
delimiter = '"'
#### CSV 文件是否包含表头。
#### 如果为 true，首行将会被跳过。
header = true
#### CSV 是否包含 NULL。
#### 如果为 true，CSV 文件的任何列都不能解析为 NULL。
not-null = false
#### 如果 `not-null` 为 false（即 CSV 可以包含 NULL），
#### 为以下值的字段将会被解析为 NULL。
null = '\N'
#### 是否解析字段内的反斜线转义符。
backslash-escape = true
#### 是否移除以分隔符结束的行。
trim-last-separator = false
```

`separator`

- 指定字段分隔符。
- 必须为单个 ASCII 字符。
- 常用值：
  - CSV 用 `','`
  - TSV 用 `"\t"`
- 对应 LOAD DATA 语句中的 `FIELDS TERMINATED BY` 项。

`delimiter`

- 指定引用定界符。
- 如果 `delimiter` 为空，所有字段都会被取消引用。
- 常用值：
  - `'"'` 使用双引号引用字段，和 RFC 4180 一致。
  - `''` 不引用
- 对应 LOAD DATA 语句中的 `FIELDS ENCLOSED BY` 项。

`header`

- 是否所有 CSV 文件都包含表头行。

- 如为 true，第一行会被用作列名。如为 false，第一行并无特殊性，按普通的数据行处理。

`not-null` 和 `null`

- `not-null` 决定是否所有字段不能为空。
- 如果 `not-null` 为 false，设定了 `null` 的字符串会被转换为 SQL NULL 而非具体数值。
- 引用不影响字段是否为空。

  例如有如下 CSV 文件：

```
A,B,C
\N,"\N",
```

  在默认设置（`not-null = false; null = '\N'`）下，列 A and B 导入 TiDB 后都将会转换为 NULL。列 C 是空字符串 `''`，但并不会解析为 NULL。

`backslash-escape`

- 是否解析字段内的反斜线转义符。
- 如果 `backslash-escape` 为 true，下列转义符会被识别并转换。

转义符	转换为
`\0`	空字符 (U+0000)
`\b`	退格 (U+0008)
`\n`	换行 (U+000A)
`\r`	回车 (U+000D)
`\t`	制表符 (U+0009)
`\Z`	Windows EOF (U+001A)

  其他情况下（如 `\"`）反斜线会被移除，仅在字段中保留其后面的字符（`"`），这种情况下，保留的字符仅作为普通字符，特殊功能（如界定符）都会失效。

- 引用不会影响反斜线转义符的解析与否。
- 对应 LOAD DATA 语句中的 `FIELDS ESCAPED BY '\'` 项。

`trim-last-separator`

- 将 `separator` 字段当作终止符，并移除尾部所有分隔符。

  例如有如下 CSV 文件：

```
A,,B,,
```

- 当 `trim-last-separator = false`，该文件会被解析为包含 5 个字段的行 `('A', '', 'B', '', '')`。

- 当 `trim-last-separator = true`，该文件会被解析为包含 3 个字段的行 `('A', '', 'B')`。

不可配置项

TiDB Lightning 并不完全支持 `LOAD DATA` 语句中的所有配置项。例如：

- 行终止符只能是 CR（`\r`），LF（`\n`）或 CRLF（`\r\n`），也就是说，无法自定义 `LINES TERMINATED BY`。
- 不可使用行前缀（`LINES STARTING BY`）。
- 不可跳过表头（`IGNORE n LINES`）。如有表头，必须是有效的列名。
- 定界符和分隔符只能为单个 ASCII 字符。

11.8.5.3.4 设置 `strict-format` 启用严格格式

导入文件的大小统一约为 256 MB 时，TiDB Lightning 可达到最佳工作状态。如果导入单个 CSV 大文件，TiDB Lightning 只能使用一个线程来处理，这会降低导入速度。

要解决此问题，可先将 CSV 文件分割为多个文件。对于通用格式的 CSV 文件，在没有读取整个文件的情况下无法快速确定行的开始和结束位置。因此，默认情况下 TiDB Lightning 不会自动分割 CSV 文件。但如果你确定待导入的 CSV 文件符合特定的限制要求，则可以启用 `strict-format` 设置。启用后，TiDB Lightning 会将单个 CSV 大文件分割为单个大小为 256 MB 的多个文件块进行并行处理。

```
[mydumper]
strict-format = true
```

严格格式的 CSV 文件中，每个字段仅占一行，即必须满足以下条件之一：

- 分隔符为空；
- 每个字段不包含 CR（`\r`）或 LF（`\n`）。

如果 CSV 文件不是严格格式但 `strict-format` 被误设为 `true`，跨多行的单个完整字段会被分割成两部分，导致解析失败，甚至不报错地导入已损坏的数据。

11.8.5.3.5 通用配置

CSV

默认设置已按照 RFC 4180 调整。

```
[mydumper.csv]
separator = ','
delimiter = '"'
header = true
not-null = false
null = '\N'
backslash-escape = true
trim-last-separator = false
```

示例内容：

```
ID,Region,Count
1,"East",32
2,"South",\N
3,"West",10
4,"North",39
```

TSV

```
[mydumper.csv]
separator = "\t"
delimiter = ''
header = true
not-null = false
null = 'NULL'
backslash-escape = false
trim-last-separator = false
```

示例内容：

```
ID    Region    Count
1     East      32
2     South     NULL
3     West      10
4     North     39
```

TPC-H DBGEN

```
[mydumper.csv]
separator = '|'
delimiter = ''
header = false
not-null = true
backslash-escape = false
trim-last-separator = true
```

示例内容：

```
1|East|32|
2|South|0|
3|West|10|
4|North|39|
```

11.8.5.4 TiDB Lightning 后端

TiDB Lightning 的后端决定 `tidb-lightning` 组件将如何把将数据导入到目标集群中。目前，TiDB Lightning 支持以下后端：

- Importer-backend（默认）
- Local-backend
- TiDB-backend

以上几种后端导入数据的区别如下：

- Importer-backend：tidb-lightning 先将 SQL 或 CSV 数据编码成键值对，由 tikv-importer 对写入的键值对进行排序，然后把这些键值对 Ingest 到 TiKV 节点中。
- Local-backend：tidb-lightning 先将数据编码成键值对并排序存储在本地临时目录，然后将这些键值对以 SST 文件的形式上传到各个 TiKV 节点，然后由 TiKV 将这些 SST 文件 Ingest 到集群中。和 Importer-backend 原理相同，不过不依赖额外的 tikv-importer 组件。
- TiDB-backend：tidb-lightning 先将数据编码成 INSERT 语句，然后直接在 TiDB 节点上运行这些 SQL 语句进行数据导入。

后端	Local-backend	Importer-backend	TiDB-backend
速度	快 (~500 GB/小时)	快 (~400 GB/小时)	慢 (~50 GB/小时)
资源使用率	高	高	低
占用网络带宽	高	中	低
导入时是否满足 ACID	否	否	是
目标表	必须为空	必须为空	可以不为空
额外组件	无	tikv-importer	无
支持 TiDB 集群版本	>= v4.0.0	全部	全部
是否影响 TiDB 对外提供服务	是	是	否

11.8.5.4.1 如何选择后端模式

- 如果导入的目标集群为 v4.0 或以上版本，请优先考虑使用 Local-backend 模式。Local-backend 部署更简单并且性能也较其他两个模式更高
- 如果目标集群为 v3.x 或以下，则建议使用 Importer-backend 模式
- 如果需要导入的集群为生产环境线上集群，或需要导入的表中已包含有数据，则最好使用 TiDB-backend 模式

11.8.5.4.2 TiDB Lightning Local-backend

自 TiDB 4.0.3 版本起，TiDB Lightning 引入了 Local-backend 特性。该特性支持导入数据到 v4.0.0 以上的 TiDB 集群。

部署和配置 TiDB Lightning

TiDB Lightning Local-backend 模式的部署方法见TiDB Lightning 部署与执行。

11.8.5.4.3 TiDB Lightning TiDB-backend

部署和配置 TiDB Lightning

使用 TiDB-backend 时，你无需部署 tikv-importer。与标准部署过程相比，部署 TiDB-backend 时有如下不同：

- 可以跳过所有涉及 tikv-importer 的步骤。
- 必须更改相应配置申明使用的是 TiDB-backend。

硬件需求

使用 TiDB-backend 时，TiDB Lightning 的速度仅受限于 TiDB 执行 SQL 语句的速度。因此，即使是低配的机器也足够发挥出最佳性能。推荐的硬件配置如下：

- 16 逻辑核 CPU
- 足够储存整个数据源的 SSD 硬盘，读取速度越快越好
- 千兆网卡

使用 TiDB Ansible 部署

1. inventory.ini 文件中，[importer_server] 部分可以留空。

```
...

[importer_server]
# keep empty

[lightning_server]
192.168.20.10

...
```

2. 忽略 group_vars/all.yml 文件中 tikv_importer_port 部分的设置，group_vars/importer_server.yml 文件也不需要修改。但是你需要在 conf/tidb-lightning.yml 文件中将 backend 设置更改为 tidb。

```
...
tikv_importer:
    backend: "tidb"   # <-- 改成 “tidb”
...
```

3. 启动、部署集群。
4. 为 TiDB Lightning 挂载数据源。
5. 启动 tidb-lightning。

手动部署

手动部署时，你无需下载和配置 tikv-importer，TiDB Lightning 可在此下载。

在运行 tidb-lightning 之前，在配置文件中加上如下几行：

```
[tikv-importer]
backend = "tidb"
```

或者在用命令行启动 `tidb-lightning` 时，传入参数 `--backend tidb`。

冲突解决

TiDB-backend 支持导入到已填充的表（非空表）。但是，新数据可能会与旧数据的唯一键冲突。你可以通过使用如下任务配置来控制遇到冲突时的默认行为：

```
[tikv-importer]
backend = "tidb"
on-duplicate = "replace" # 或者 “error”、“ignore”
```

设置	冲突时默认行为	对应 SQL 语句
replace	新数据替代旧数据	`REPLACE INTO ...`
ignore	保留旧数据，忽略新数据	`INSERT IGNORE INTO ...`
error	中止导入	`INSERT INTO ...`

从 Loader 迁移到 TiDB Lightning TiDB-backend

当需要将数据导入到 TiDB 集群时，TiDB Lightning TiDB-backend 可以完全取代Loader。下表说明了如何将Loader 的配置迁移到TiDB Lightning 配置中：

Loader

TiDB Lightning

```
#### 日志级别
log-level = "info"
#### 日志的输出目录
log-file = "loader.log"
#### Prometheus
status-addr = ":8272"
#### 线程数
pool-size = 16
```

```
[lightning]
#### 日志级别
level = "info"
#### 日志的输出目录。如果未指定该位置目录，默认为执行命令的所在目录。
file = "tidb-lightning.log"
#### Prometheus
pprof-port = 8289
#### 并发度（最好使用默认设置）
#region-concurrency = 16
```

```
#### 断点数据库名
checkpoint-schema = "tidb_loader"
```

```
[checkpoint]
#### 断点存储
enable = true
schema = "tidb_lightning_checkpoint"
#### 断点默认存储在本地的文件系统，这样更高效。但你也可以
#### 选择将断点存储在目标数据库中，设置如下：
#### driver = "mysql"
```

```
[tikv-importer]
#### 使用 TiDB-backend
backend = "tidb"
```

```
#### 数据源目录
dir = "/data/export/"
```

```
[mydumper]
#### 数据源目录
data-source-dir = "/data/export"
```

```
[db]
#### TiDB 连接参数
host = "127.0.0.1"
port = 4000
user = "root"
password = ""
#sql-mode = ""
```

```
[tidb]
#### TiDB 连接参数
host = "127.0.0.1"
port = 4000
#### 在 TiDB-backend 模式下，该参数为可选参数
#### status-port = 10080
user = "root"
password = ""
#sql-mode = ""
```

```
#### [[route-rules]]
#### Table routes
#### schema-pattern = "shard_db_*"
#### table-pattern = "shard_table_*"
#### target-schema = "shard_db"
#### target-table = "shard_table"
```

```
#### [[routes]]
#### Table routes
#### schema-pattern = "shard_db_*"
#### table-pattern = "shard_table_*"
#### target-schema = "shard_db"
#### target-table = "shard_table"
```

11.8.5.4.4 TiDB Lightning Importer-backend

部署 Importer-backend

本节介绍 TiDB Lightning 使用 Importer 模式的两种部署方式：使用 TiDB Ansible 部署和手动部署。

硬件需求

`tidb-lightning` 和 `tikv-importer` 这两个组件皆为资源密集程序，建议各自单独部署。

为了优化效能，建议硬件配置如下：

- `tidb-lightning`
  - 32+ 逻辑核 CPU
  - 足够储存整个数据源的 SSD 硬盘，读取速度越快越好
  - 使用万兆网卡，带宽需 300 MB/s 以上
  - 运行过程默认会占满 CPU 资源，因此建议将 `tidb-lightning` 部署到一台单独的机器上。条件不允许的情况下可以和其他组件（比如 `tidb-server`）部署在同一台机器上，然后通过配置 `region-concurrency` 限制 `tidb-lightning` 使用 CPU 资源。
- `tikv-importer`
  - 32+ 逻辑核 CPU
  - 40 GB+ 内存
  - 1 TB+ SSD 硬盘，IOPS 越高越好（要求 ≥8000）
    - 硬盘必须大于最大的 N 个表的大小总和，其中 N = `max(index-concurrency, table-concurrency)`。
  - 使用万兆网卡，带宽需 300 MB/s 以上
  - 运行过程中 CPU、I/O 和网络带宽资源都可能占满，建议单独部署。

如果机器充裕的话，可以部署多套 `tidb-lightning` + `tikv-importer`，然后将源数据以表为粒度进行切分，并发导入。

使用 TiDB Ansible 部署 TiDB Lightning

TiDB Lightning 可随 TiDB 集群一起用TiDB Ansible 部署。

1. 编辑 `inventory.ini`，分别配置一个 IP 来部署 `tidb-lightning` 和 `tikv-importer`。

```
...

[importer_server]
192.168.20.9

[lightning_server]
192.168.20.10

...
```

2. 修改 group_vars/*.yml 的变量配置这两个工具。

    - group_vars/all.yml

    ```
    ...
    # tikv-importer 的监听端口。需对 tidb-lightning 服务器开放。
    tikv_importer_port: 8287
    ...
    ```

    - group_vars/lightning_server.yml

    ```
    ---
    dummy:

    # 提供监控告警的端口。需对监控服务器 (monitoring_server) 开放。
    tidb_lightning_pprof_port: 8289

    # 获取数据源 ( Mydumper SQL dump 或 CSV ) 的路径。
    data_source_dir: "{{ deploy_dir }}/mydumper"
    ```

    - group_vars/importer_server.yml

    ```
    ---
    dummy:

    # 储存引擎文件的路径。需存放在空间足够大的分区。
    import_dir: "{{ deploy_dir }}/data.import"
    ```

3. 开始部署。

```
ansible-playbook bootstrap.yml &&
ansible-playbook deploy.yml
```

4. 将数据源写入 data_source_dir 指定的路径。

5. 登录 tikv-importer 的服务器，并执行以下命令来启动 Importer。

```
scripts/start_importer.sh
```

6. 登录 tidb-lightning 的服务器，并执行以下命令来启动 Lightning，开始导入过程。

```
scripts/start_lightning.sh
```

7. 完成后，在 tikv-importer 的服务器执行 scripts/stop_importer.sh 来关闭 Importer。

手动部署 TiDB Lightning

第 1 步：部署 TiDB 集群

在开始数据导入之前，需先部署一套要进行导入的 TiDB 集群 (版本要求 2.0.9 以上)，建议使用最新版。部署方法可参考使用 TiUP 部署 TiDB 集群。

第 2 步：下载 TiDB Lightning 安装包

在工具下载页面下载 TiDB Lightning 安装包（需选择与 TiDB 集群相同的版本）。

第 3 步：启动 tikv-importer

1. 从安装包上传 bin/tikv-importer。
2. 配置 tikv-importer.toml。

```
# TiKV Importer 配置文件模版

# 日志文件。
log-file = "tikv-importer.log"
# 日志等级：trace、debug、info、warn、error、off。
log-level = "info"

# 状态服务器的监听地址。
status-server-address = "0.0.0.0:8286"

[server]
# tikv-importer 监听的地址，tidb-lightning 需要连到这个地址进行数据写入。
addr = "0.0.0.0:8287"

[metric]
# 给 Prometheus 客户端的推送任务名称。
job = "tikv-importer"
# 给 Prometheus 客户端的推送间隔。
interval = "15s"
# Prometheus Pushgateway 地址。
address = ""

[import]
# 存储引擎文档 (engine file) 的文件夹路径。
```

```
import-dir = "/mnt/ssd/data.import/"
```

上面仅列出了 tikv-importer 的基本配置。完整配置请参考tikv-importer 配置说明。

3. 运行 tikv-importer。

```
nohup ./tikv-importer -C tikv-importer.toml > nohup.out &
```

第 4 步：启动 tidb-lightning

1. 从安装包上传 bin/tidb-lightning 及 bin/tidb-lightning-ctl。
2. 将数据源写入到同样的机器。
3. 配置 tidb-lightning.toml。对于没有出现在下述模版中的配置，TiDB Lightning 给出配置错误的提醒并退出。

```
[lightning]

# 转换数据的并发数，默认为逻辑 CPU 数量，不需要配置。
# 混合部署的情况下可以配置为逻辑 CPU 的 75% 大小。
# region-concurrency =

# 日志
level = "info"
file = "tidb-lightning.log"

[tikv-importer]
# tikv-importer 的监听地址，需改成 tikv-importer 服务器的实际地址。
addr = "172.16.31.10:8287"

[mydumper]
# Mydumper 源数据目录。
data-source-dir = "/data/my_database"

[tidb]
# 目标集群的信息。tidb-server 的监听地址，填一个即可。
host = "172.16.31.1"
port = 4000
user = "root"
password = ""
# 表架构信息在从 TiDB 的“状态端口”获取。
status-port = 10080
```

上面仅列出了 tidb-lightning 的基本配置信息。完整配置信息请参考tidb-lightning 配置说明。

4. 运行 tidb-lightning。如果直接在命令行中用 nohup 启动程序，可能会因为 SIGHUP 信号而退出，建议把 nohup 放到脚本里面，如：

```
#!/bin/bash
nohup ./tidb-lightning -config tidb-lightning.toml > nohup.out &
```

11.8.5.5 TiDB Lightning Web 界面

TiDB Lightning 支持在网页上查看导入进度或执行一些简单任务管理，这就是 TiDB Lightning 的服务器模式。本文将介绍服务器模式下的 Web 界面和一些常见操作。

启用服务器模式的方式有如下几种：

1. 在启动 `tidb-lightning` 时加上命令行参数 `--server-mode`。

```
./tidb-lightning --server-mode --status-addr :8289
```

2. 在配置文件中设置 `lightning.server-mode`。

```
[lightning]
server-mode = true
status-addr = ':8289'
```

TiDB Lightning 启动后，可以访问 `http://127.0.0.1:8289` 来管理程序（实际的 URL 取决于你的 `status-addr` 设置）。

服务器模式下，TiDB Lightning 不会立即开始运行，而是通过用户在 web 页面提交（多个）任务来导入数据。

11.8.5.5.1 TiDB Lightning Web 首页

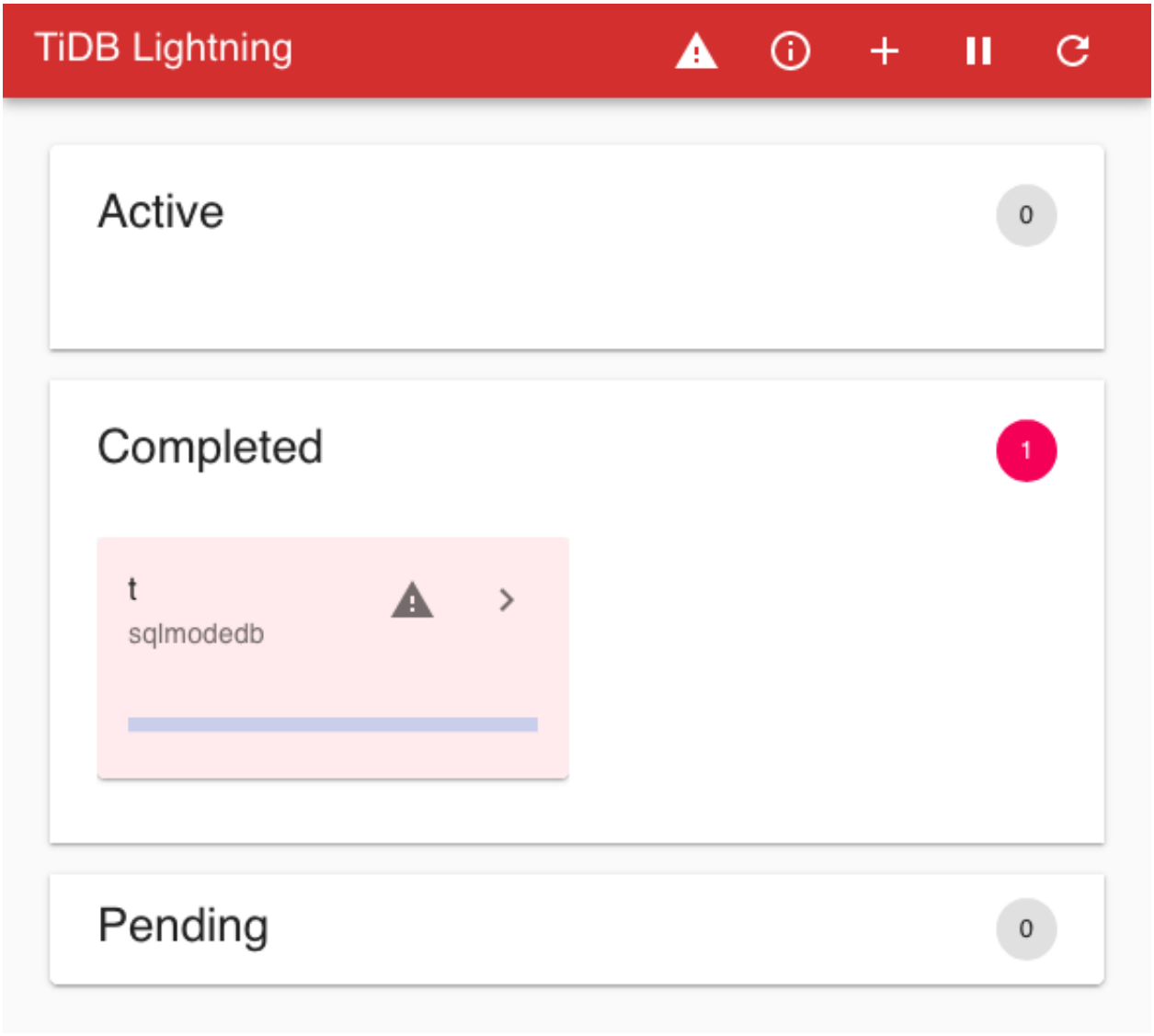


图 164: TiDB Lightning Web 首页

标题栏上图标所对应的功能，从左到右依次为：

图标	功能
“TiDB Lightning”	点击即返回首页
□	显示前一个任务的所有错误信息
ⓘ	列出当前及队列中的任务，可能会出现一个标记提示队列中任务的数量
+	提交单个任务
□/▸	暂停/继续当前操作
□	设置网页自动刷新

标题栏下方的三个面板显示了不同状态下的所有表：

- Active：当前正在导入这些表
- Completed：这些表导入成功或失败
- Pending：这些表还没有被处理

每个面板都包含用于描述表状态的卡片。

11.8.5.5.2 提交任务

点击标题栏的 + 图标提交任务。

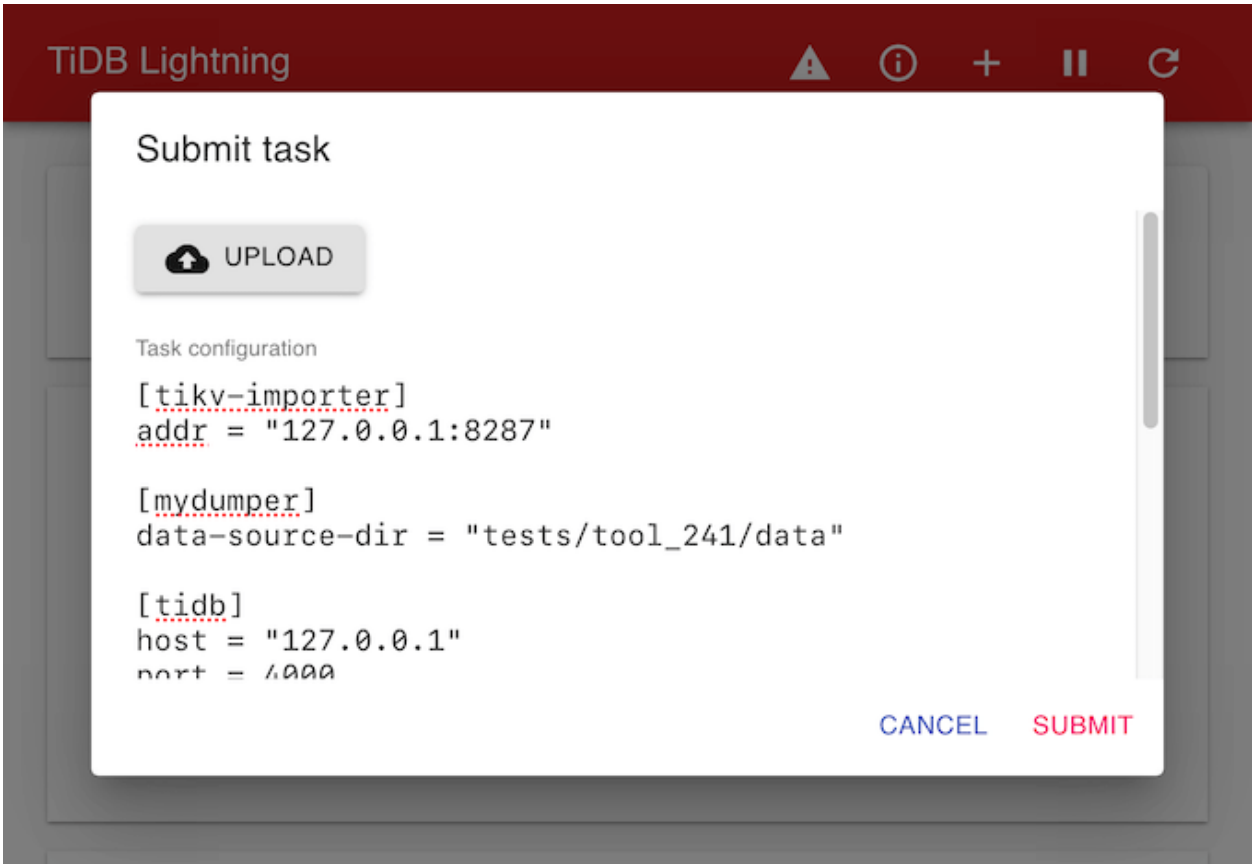


图 165: 提交任务对话框

任务 (task) 为 TOML 格式的文件，具体参考TiDB Lightning 任务配置。你也可以点击 UPLOAD 上传一个本地的 TOML 文件。

点击 SUBMIT 运行任务。如果当前有任务正在运行，新增任务会加入队列并在当前任务结束后执行。

11.8.5.5.3 查看导入进度

点击首页表格卡片上的 > 图标，查看表格导入的详细进度。

图 166: 表格导入进度

该页显示每张表的引擎文件的导入过程。

点击标题栏上的 TiDB Lightning 返回首页。

11.8.5.5.4 管理任务

单击标题栏上的 ⓘ 图标来管理当前及队列中的任务。

图 167: 任务管理页面

每个任务都是依据提交时间来标记。点击该任务将显示 JSON 格式的配置文件。

点击任务上的 ▯ 可以对该任务进行管理。你可以立即停止任务，或重新排序队列中的任务。

11.8.6 TiDB Lightning 监控告警

`tidb-lightning` 支持使用 Prometheus 采集监控指标 (metrics)。本文主要介绍 TiDB Lightning 的监控配置与监控指标。

11.8.6.1 监控配置

- 如果是使用 TiDB Ansible 部署 TiDB Lightning，只要将服务器地址加到 `inventory.ini` 文件里的 `[monitored_servers]` 部分即可。
- 如果是手动部署 TiDB Lightning，则参照以下步骤进行配置。

只要 Prometheus 能发现 `tidb-lightning` 和 `tikv-importer` 的监控地址，就能收集对应的监控指标。

监控的端口可在 `tidb-lightning.toml` 中配置：

```
[lightning]
### 用于调试和 Prometheus 监控的 HTTP 端口。输入 0 关闭。
```

```
pprof-port = 8289

...
```

监控的端口也可在 tikv-importer.toml 配置：

```
### 状态服务器的监听地址
status-server-address = '0.0.0.0:8286'
```

配置 Prometheus 后，tidb-lightning 才能发现服务器。配置方法如下，将服务器地址直接添加至 scrape_configs 部分：

```
...
scrape_configs:
  - job_name: 'lightning'
    static_configs:
      - targets: ['192.168.20.10:8289']
  - job_name: 'tikv-importer
    static_configs:
      - targets: ['192.168.20.9:8286']
```

11.8.6.2 Grafana 面板

Grafana 的可视化面板可以让你在网页上监控 Prometheus 指标。

使用 TiDB Ansible 部署 TiDB 集群时，会同时部署一套 Grafana + Prometheus 的监控系统。

如果使用其他方式部署 TiDB Lightning，需先导入面板的 JSON 文件。

11.8.6.2.1 第一行：速度面板

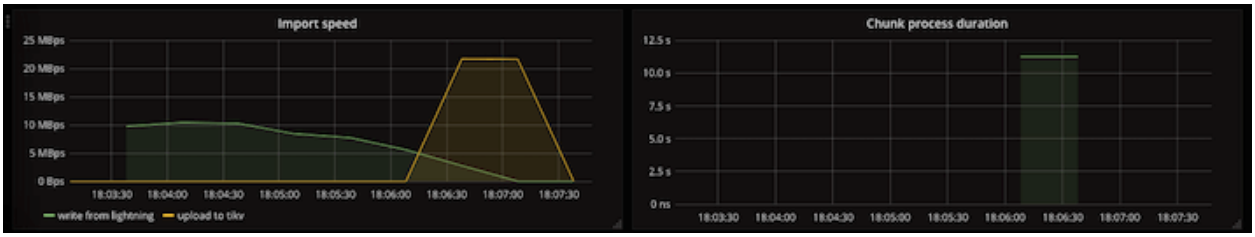


图 168: 第一行速度面板

面板名称	序列	描述
Import speed	write from lightning	从 TiDB Lightning 向 TiKV Importer 发送键值对的速度，取决于每个表的复杂性
Import speed	upload to tikv	从 TiKV Importer 上传 SST 文件到所有 TiKV 副本的总体速度
Chunk process duration		完全编码单个数据文件所需的平均时间

有时导入速度会降到 0，这是为了平衡其他部分的速度，属于正常现象。

11.8.6.2.2 第二行：进度面板

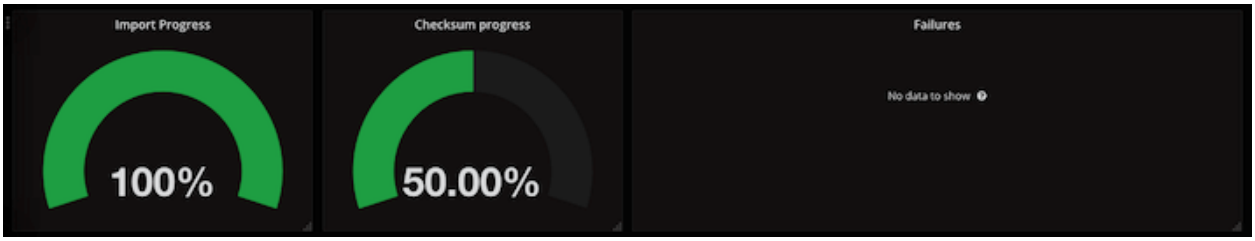


图 169: 第二行进度面板

面板名称	描述
Import progress	已编码的文件所占百分比
Checksum progress	已导入的表所占百分比
Failures	导入失败的表的数量以及故障点，通常为空

11.8.6.2.3 第三行：资源使用面板

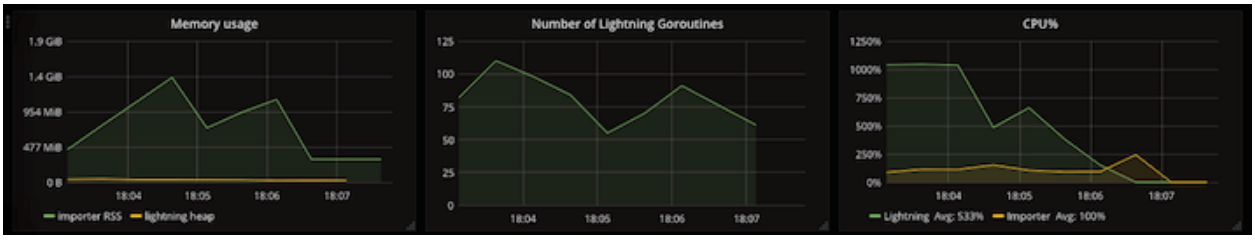


图 170: 第三行资源使用面板

面板名称	描述
Memory usage	每个服务占用的内存
Number of Lightning Goroutines	TiDB Lightning 使用的运行中的 goroutines 数量
CPU%	每个服务使用的逻辑 CPU 数量

11.8.6.2.4 第四行：配额使用面板

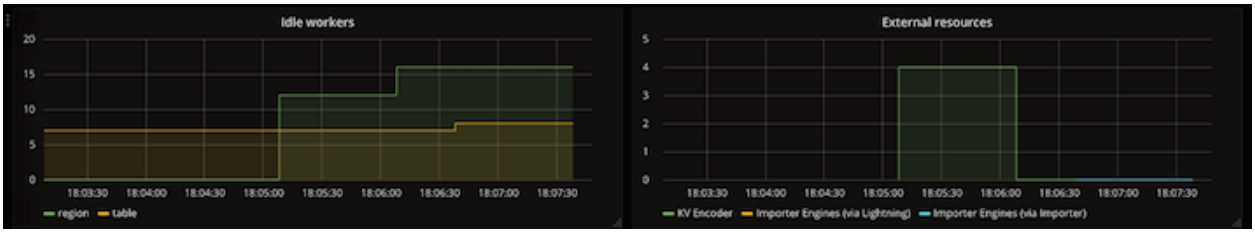


图 171: 第四行配额使用面板

面板名称	序列	描述
Idle workers	io	未使用的 `io-concurrency` 的数量，通常接近配置值（默认为 5），接近 0 时表示磁盘运行太慢
Idle workers	closed-engine	已关闭但未清理的引擎数量，通常接近 `index-concurrency` 与 `table-concurrency` 的和（默认为 8），接近 0 时表示 TiDB Lightning 比 TiKV Importer 快，导致 TiDB Lightning 延迟

面板名称	序列	描述
Idle workers	table	未使用的 `table-concurrency` 的数量，通常为 0，直到进程结束
Idle workers	index	未使用的 `index-concurrency` 的数量，通常为 0，直到进程结束
Idle workers	region	未使用的 `region-concurrency` 的数量，通常为 0，直到进程结束
External resources	KV Encoder	已激活的 KV encoder 的数量，通常与 `region-concurrency` 的数量相同，直到进程结束

面板名称	序列	描述
External resources	Importer Engines	打开的引擎文件数量，不应超过 max-open-engines 的设置

11.8.6.2.5 第五行：读取速度面板

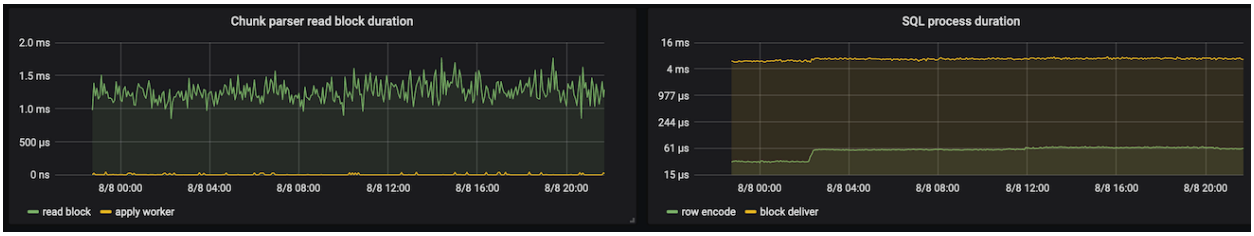


图 172: 第五行读取速度面板

面板名称	序列	描述
Chunk parser read block duration	read block	读取一个字节块来准备解析时所消耗的时间
Chunk parser read block duration	apply worker	等待 io-concurrency 空闲所消耗的时间
SQL process duration	row encode	解析和编码单行所消耗的时间
SQL process duration	block deliver	将一组键值对发送到 TiKV Importer 所消耗的时间

如果上述项的持续时间过长，则表示 TiDB Lightning 使用的磁盘运行太慢或 I/O 太忙。

11.8.6.2.6 第六行：存储空间面板

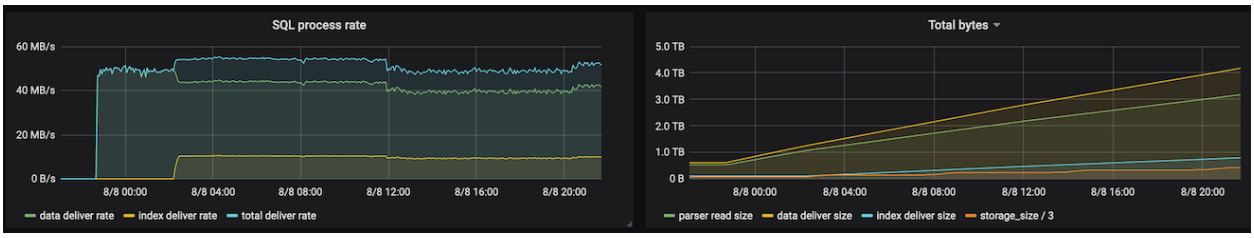


图 173: 第六行存储空间面板

面板名称	序列	描述
SQL process rate	data deliver rate	向 TiKV Importer 发送数据键值对的速度
SQL process rate	index deliver rate	向 TiKV Importer 发送索引键值对的速度
SQL process rate	total deliver rate	发送数据键值对及索引键值对的速度之和
Total bytes	parser read size	TiDB Lightning 正在读取的字节数
Total bytes	data deliver size	已发送到 TiKV Importer 的数据键值对的字节数
Total bytes	index deliver size	已发送到 TiKV Importer 的索引键值对的字节数
Total bytes	storage_size/3	TiKV 集群占用的存储空间大小的 1/3（3 为默认的副本数量）

11.8.6.2.7 第七行：导入速度面板

图 174: 第七行导入速度面板

面板名称	序列	描述
Delivery duration	Range delivery	将一个 range 的键值对上传到 TiKV 集群所消耗的时间
Delivery duration	SST delivery	将单个 SST 文件上传到 TiKV 集群所消耗的时间
SST process duration	Split SST	将键值对流切分成若干 SST 文件所消耗的时间
SST process duration	SST upload	上传单个 SST 文件所消耗的时间
SST process duration	SST ingest	ingest 单个 SST 文件所消耗的时间
SST process duration	SST size	单个 SST 文件的大小

11.8.6.3 监控指标

本节将详细描述 `tikv-importer` 和 `tidb-lightning` 的监控指标。

11.8.6.3.1 tikv-importer

tikv-importer 的监控指标皆以 tikv_import_* 为前缀。

- tikv_import_rpc_duration（直方图）

  完成一次 RPC 用时直方图。标签：

  - request：所执行 RPC 请求的类型
    - switch_mode — 将一个 TiKV 节点切换为 import/normal 模式
    - open_engine — 打开引擎文件
    - write_engine — 接收数据并写入引擎文件
    - close_engine — 关闭一个引擎文件
    - import_engine — 导入一个引擎文件到 TiKV 集群中
    - cleanup_engine — 删除一个引擎文件
    - compact_cluster — 显式压缩 TiKV 集群
    - upload — 上传一个 SST 文件
    - ingest — Ingest 一个 SST 文件
    - compact — 显式压缩一个 TiKV 节点
  - result：RPC 请求的执行结果
    - ok
    - error

- tikv_import_write_chunk_bytes（直方图）

  从 TiDB Lightning 接收的键值对区块大小（未压缩）的直方图。

- tikv_import_write_chunk_duration（直方图）

  从 tidb-lightning 接收每个键值对区块所需时间的直方图。

- tikv_import_upload_chunk_bytes（直方图）

  上传到 TiKV 的每个 SST 文件区块大小（压缩）的直方图。

- tikv_import_range_delivery_duration（直方图）

  将一个 range 的键值对发送至 dispatch-job 任务所需时间的直方图。

- tikv_import_split_sst_duration（直方图）

  将 range 从引擎文件中分离到单个 SST 文件中所需时间的直方图。

- tikv_import_sst_delivery_duration（直方图）

  将 SST 文件从 dispatch-job 任务发送到 ImportSSTJob 任务所需时间的直方图

- tikv_import_sst_recv_duration（直方图）

  ImportSSTJob 任务接收从 dispatch-job 任务发送过来的 SST 文件所需时间的直方图。

- tikv_import_sst_upload_duration（直方图）

  从 ImportSSTJob 任务上传 SST 文件到 TiKV 节点所需时间的直方图。

- `tikv_import_sst_chunk_bytes`（直方图）

  上传到 TiKV 节点的 SST 文件（压缩）大小的直方图。

- `tikv_import_sst_ingest_duration`（直方图）

  将 SST 文件传入至 TiKV 所需时间的直方图。

- `tikv_import_each_phase`（测量仪）

  表示运行阶段。值为 1 时表示在阶段内运行，值为 0 时表示在阶段内运行。标签：

  - phase：prepare / import

- `tikv_import_wait_store_available_count`（计数器）

  计算出现 TiKV 节点没有充足空间上传 SST 文件现象的次数。标签：

  - store_id：TiKV 存储 ID。

- `tikv_import_upload_chunk_duration`（直方图）

  上传到 TiKV 的每个区块所需时间的直方图。

11.8.6.3.2 tidb-lightning

`tidb-lightning` 的监控指标皆以 `lightning_*` 为前缀。

- `lightning_importer_engine`（计数器）

  计算已开启及关闭的引擎文件数量。标签：

  - type:
    * open
    * closed

- `lightning_idle_workers`（计量表盘）

  计算闲置的 worker。标签：

  - name：
    * `table` — 未使用的 `table-concurrency` 的数量，通常为 0，直到进程结束
    * `index` — 未使用的 `index-concurrency` 的数量，通常为 0，直到进程结束
    * `region` — 未使用的 `region-concurrency` 的数量，通常为 0，直到进程结束
    * `io` — 未使用的 `io-concurrency` 的数量，通常接近配置值（默认为 5），接近 0 时表示磁盘运行太慢
    * `closed-engine` — 已关闭但未清理的引擎数量，通常接近 `index-concurrency` 与 `table-concurrency` 的和（默认为 8），接近 0 时表示 TiDB Lightning 比 TiKV Importer 快，导致 TiDB Lightning 延迟

- `lightning_kv_encoder`（计数器）

  计算已开启及关闭的 KV 编码器。KV 编码器是运行于内存的 TiDB 实例，用于将 SQL 的 `INSERT` 语句转换成键值对。此度量的净值（开启减掉关闭）在正常情况下不应持续增长。标签：

  - type:
    * open
    * closed

- lightning_tables（计数器）

  计算处理过的表及其状态。标签：

  - state：表的状态，表明当前应执行的操作
    * pending — 等待处理
    * written — 所有数据已编码和传输
    * closed — 所有对应的引擎文件已关闭
    * imported — 所有引擎文件已上传到目标集群
    * altered_auto_inc — 自增 ID 已改
    * checksum — 已计算校验和
    * analyzed — 已进行统计信息分析
    * completed — 表格已完全导入并通过验证
  - result：当前操作的执行结果
    * success — 成功
    * failure — 失败（未完成）

- lightning_engines（计数器）

  计算处理后引擎文件的数量以及其状态。标签：

  - state：引擎文件的状态，表明当前应执行的操作
    * pending — 等待处理
    * written — 所有数据已编码和传输
    * closed — 引擎文件已关闭
    * imported — 当前引擎文件已上传到目标集群
    * completed — 当前引擎文件已完全导入
  - result：当前操作的执行结果
    * success — 成功
    * failure — 失败（未完成）

- lightning_chunks（计数器）

  计算处理过的 Chunks 及其状态。标签：

  - state: 单个 Chunk 的状态，表明该 Chunk 当前所处的阶段
    * estimated —（非状态）当前任务中 Chunk 的数量
    * pending — 已载入但未执行
    * running — 正在编码和发送数据
    * finished — 该 Chunk 已处理完毕
    * failed — 处理过程中发生错误

- lightning_import_seconds（直方图）

  导入每个表所需时间的直方图。

- `lightning_row_read_bytes`（直方图）

  单行 SQL 数据大小的直方图。

- `lightning_row_encode_seconds`（直方图）

  解码单行 SQL 数据到键值对所需时间的直方图。

- `lightning_row_kv_deliver_seconds`（直方图）

  发送一组与单行 SQL 数据对应的键值对所需时间的直方图。

- `lightning_block_deliver_seconds`（直方图）

  每个键值对中的区块传送到 `tikv-importer` 所需时间的直方图。

- `lightning_block_deliver_bytes`（直方图）

  发送到 Importer 的键值对中区块（未压缩）的大小的直方图。

- `lightning_chunk_parser_read_block_seconds`（直方图）

  数据文件解析每个 SQL 区块所需时间的直方图。

- `lightning_checksum_seconds`（直方图）

  计算表中 Checksum 所需时间的直方图。

- `lightning_apply_worker_seconds`（直方图）

  获取闲置 worker 等待时间的直方图 (参见 `lightning_idle_workers` 计量表盘)。标签：

  ```
  - name:
      * table
      * index
      * region
      * io
      * closed-engine
  ```

11.8.7 TiDB Lightning 常见问题

本文列出了一些使用 TiDB Lightning 时可能会遇到的问题与解决办法。

11.8.7.1 TiDB Lightning 对 TiDB/TiKV/PD 的最低版本要求是多少？

TiDB Lightning 的版本应与集群相同。如果使用 Local-backend 模式，最低版本要求为 4.0.0。如果使用 Importer-backend 或 TiDB-backend 模式最低版本要求是 2.0.9，但建议使用最新的稳定版本 3.0。

11.8.7.2 TiDB Lightning 支持导入多个库吗？

支持。

11.8.7.3 TiDB Lightning 对下游数据库的账号权限要求是怎样的？

TiDB Lightning 需要以下权限：

- SELECT
- UPDATE
- ALTER
- CREATE
- DROP

如果选择TiDB-backend 模式，或目标数据库用于存储断点，则 TiDB Lightning 额外需要以下权限：

- INSERT
- DELETE

Local-backend 和 Importer-backend 无需以上两个权限，因为数据直接被 Ingest 到 TiKV 中，所以绕过了 TiDB 的权限系统。只要 TiKV、 TiKV Importer 和 TiDB Lightning 的端口在集群之外不可访问，就可以保证安全。

如果 TiDB Lightning 配置项 `checksum = true`，则 TiDB Lightning 需要有下游 TiDB admin 用户权限。

11.8.7.4 TiDB Lightning 在导数据过程中某个表报错了，会影响其他表吗？进程会马上退出吗？

如果只是个别表报错，不会影响整体。报错的那个表会停止处理，继续处理其他的表。

11.8.7.5 如何正确重启 TiDB Lightning？

如果使用 Importer-backend，根据 `tikv-importer` 的状态，重启 TiDB Lightning 的基本顺序如下：

如果 `tikv-importer` 仍在运行：

1. 结束 `tidb-lightning` 进程。
2. 执行修改操作（如修复数据源、更改设置、更换硬件等）。
3. 如果上面的修改操作更改了任何表，你还需要清除对应的断点。
4. 重启 `tidb-lightning`。

如果 `tikv-importer` 需要重启：

1. 结束 `tidb-lightning` 进程。
2. 结束 `tikv-importer` 进程。
3. 执行修改操作（如修复数据源、更改设置、更换硬件等）。
4. 重启 `tikv-importer`。
5. 重启 `tidb-lightning` 并等待，直到程序因校验和错误（如果有的话）而失败。
    - 重启 `tikv-importer` 将清除所有仍在写入的引擎文件，但是 `tidb-lightning` 并不会感知到该操作。从 v3.0 开始，最简单的方法是让 `tidb-lightning` 继续，然后再重试。
6. 清除失败的表及断点。
7. 再次重启 `tidb-lightning`。

如果使用 Local-backend 和 TiDB-backend，操作和 Importer-backend 的 `tikv-importer` 仍在运行时相同。

11.8.7.6 如何校验导入的数据的正确性？

TiDB Lightning 默认会对导入数据计算校验和 (checksum)，如果校验和不一致就会停止导入该表。可以在日志看到相关的信息。

TiDB 也支持从 MySQL 命令行运行 `ADMIN CHECKSUM TABLE` 指令来计算校验和。

```
ADMIN CHECKSUM TABLE `schema`.`table`;
```

```
+---------+------------+--------------------+-----------+-------------+
| Db_name | Table_name | Checksum_crc64_xor | Total_kvs | Total_bytes |
+---------+------------+--------------------+-----------+-------------+
| schema  | table      | 5505282386844578743 |         3 |          96 |
+---------+------------+--------------------+-----------+-------------+
1 row in set (0.01 sec)
```

11.8.7.7 TiDB Lightning 支持哪些格式的数据源？

TiDB Lightning 只支持两种格式的数据源：

1. Dumpling 生成的 SQL dump
2. 储存在本地文件系统的CSV 文件

11.8.7.8 我已经在下游创建好库和表了，TiDB Lightning 可以忽略建库建表操作吗？

可以。在配置文档中的 `[mydumper]` 部分将 `no-schema` 设置为 `true` 即可。`no-schema=true` 会默认下游已经创建好所需的数据库和表，如果没有创建，会报错。

11.8.7.9 有些不合法的数据，能否通过关掉严格 SQL 模式 (Strict SQL Mode) 来导入？

可以。TiDB Lightning 默认的 `sql_mode` 为 `"STRICT_TRANS_TABLES,NO_ENGINE_SUBSTITUTION"`。

这个设置不允许一些非法的数值，例如 `1970-00-00` 这样的日期。可以修改配置文件 `[tidb]` 下的 `sql-mode` 值。

```
...
[tidb]
sql-mode = ""
...
```

11.8.7.10 可以启用一个 `tikv-importer`，同时有多个 `tidb-lightning` 进程导入数据吗？

只要每个 TiDB Lightning 操作的表互不相同就可以。

11.8.7.11 如何正确结束 `tikv-importer` 进程？

根据部署方式，选择相应操作结束进程

- 使用 TiDB Ansible 部署：在 Importer 的服务器上运行 `scripts/stop_importer.sh`。
- 手动部署：如果 `tikv-importer` 正在前台运行，可直接按 Ctrl+C 退出。否则，可通过 `ps aux | grep tikv-importer` 获取进程 ID，然后通过 `kill «pid»` 结束进程。

11.8.7.12 如何正确结束 tidb-lightning 进程？

根据部署方式，选择相应操作结束进程

- 使用 TiDB Ansible 部署：在 Lightning 的服务器上运行 `scripts/stop_lightning.sh`。
- 手动部署：如果 `tidb-lightning` 正在前台运行，可直接按 Ctrl+C 退出。否则，可通过 `ps aux | grep tidb-lightning` 获取进程 ID，然后通过 `kill -2 «pid»` 结束进程。

11.8.7.13 tidb-lightning 在服务器上运行，进程莫名其妙地退出了，是怎么回事呢？

这种情况可能是启动方式不正确，导致收到 SIGHUP 信号而退出。此时 `tidb-lightning.log` 通常有如下日志：

```
[2018/08/10 07:29:08.310 +08:00] [INFO] [main.go:41] ["got signal to exit"] [signal=hangup]
```

不推荐在命令行中直接使用 nohup 启动进程，推荐使用脚本启动 `tidb-lightning`。

另外，如果从 TiDB Lightning 的 log 的最后一条日志显示遇到的错误是 “Context canceled”，需要在日志中搜索第一条 “ERROR” 级别的日志。在这条日志之前，通常也会紧跟有一条 “got signal to exit”，表示 TiDB Lighting 是收到中断信号然后退出的。

11.8.7.14 为什么用过 TiDB Lightning 之后，TiDB 集群变得又慢又耗 CPU？

如果 `tidb-lightning` 异常退出，集群可能仍处于 “导入模式” (import mode)，该模式不适用于生产环境。此时可执行以下命令查看当前使用的模式：

```
tidb-lightning-ctl --fetch-mode
```

可执行以下命令强制切换回 “普通模式” (normal mode)：

```
tidb-lightning-ctl --switch-mode=normal
```

11.8.7.15 TiDB Lightning 可以使用千兆网卡吗？

使用 TiDB Lightning 建议配置万兆网卡。不推荐使用千兆网卡，尤其是在部署 `tikv-importer` 的机器上。

千兆网卡的总带宽只有 120 MB/s，而且需要与整个 TiKV 集群共享。在使用 TiDB Lightning 导入时，极易用尽所有带宽，继而因 PD 无法联络集群使集群断连。为了避免这种情况，你可以在`tikv-importer` 的配置文件中限制上传速度。

```
[import]
### Importer 上传至 TiKV 的最大速度（字节/秒）。
### 建议将该速度设为 100 MB/s 或更小。
upload-speed-limit = "100MB"
```

11.8.7.16 为什么 TiDB Lightning 需要在 TiKV 集群预留这么多空间？

当使用默认的 3 副本设置时，TiDB Lightning 需要 TiKV 集群预留数据源大小 6 倍的空间。多出来的 2 倍是算上下列没储存在数据源的因素的保守估计：

- 索引会占据额外的空间
- RocksDB 的空间放大效应

11.8.7.17 TiDB Lightning 使用过程中是否可以重启 TiKV Importer？

不能，TiKV Importer 会在内存中存储一些引擎文件，重启后，`tidb-lightning` 会因连接失败而停止。此时，你需要清除失败的断点，因为这些 TiKV Importer 特有的信息丢失了。你可以在之后重启 TiDB Lightning。

11.8.7.18 如何清除所有与 TiDB Lightning 相关的中间数据？

1. 删除断点文件。

```
tidb-lightning-ctl --config conf/tidb-lightning.toml --checkpoint-remove=all
```

   如果出于某些原因而无法运行该命令，你可以尝试手动删除 `/tmp/tidb_lightning_checkpoint.pb` 文件。

2. 如果使用 Local-backend，删除配置中 `sorted-kv-dir` 对应的目录；如果使用 Importer-backend，删除 `tikv-importer` 所在机器上的整个 `import` 文件目录。

3. 如果需要的话，删除 TiDB 集群上创建的所有表和库。

11.8.7.19 TiDB Lightning 报错 `could not find first pair, this shouldn't happen`

报错原因是遍历本地排序的文件时出现异常，可能在 TiDB Lightning 打开的文件数量超过系统的上限时发生报错。在 Linux 系统中，可以使用 `ulimit -n` 命令确认此值是否过小。建议在导入期间将此设置调整为 1000000（即 `ulimit -n 1000000`）。

11.8.7.20 TiDB Lightning 导入速度太慢

TiDB Lightning 的正常速度为每条线程每 2 分钟导入一个 256 MB 的数据文件，如果速度远慢于这个数值就是有问题。导入的速度可以检查日志提及 `restore chunk … takes` 的记录，或者观察 Grafana 的监控信息。

导入速度太慢一般有几个原因：

原因 1：`region-concurrency` 设定太高，线程间争用资源反而减低了效率。

1. 从日志的开头搜寻 `region-concurrency` 能知道 TiDB Lightning 读到的参数是多少。
2. 如果 TiDB Lightning 与其他服务（如 TiKV Importer）共用一台服务器，必需手动将 `region-concurrency` 设为该服务器 CPU 数量的 75%。
3. 如果 CPU 设有限额（例如从 Kubernetes 指定的上限），TiDB Lightning 可能无法自动判断出来，此时亦需要手动调整 `region-concurrency`。

原因 2：表结构太复杂。

每条索引都会额外增加键值对。如果有 N 条索引，实际导入的大小就差不多是 Dumpling 文件的 N+1 倍。如果索引不太重要，可以考虑先从 schema 去掉，待导入完成后再使用 `CREATE INDEX` 加回去。

原因 3: 单个文件过大。

把源数据分割为单个大小约为 256 MB 的多个文件时，TiDB Lightning 会并行处理数据，达到最佳效果。如果导入的单个文件过大，TiDB Lightning 可能无响应。

如果源数据是 CSV 格式文件，并且所有的 CSV 文件内都不存在包含字符换行符的字段 (U+000A 及 U+000D)，则可以启用 `strict-format`，TiDB Lightning 会自动分割大文件。

```
[mydumper]
strict-format = true
```

原因 4：TiDB Lightning 版本太旧。

试试最新的版本吧！可能会有改善。

11.8.7.21 `checksum failed: checksum mismatched remote vs local`

原因：本地数据源跟目标数据库某个表的校验和不一致。这通常有更深层的原因：

1. 这张表可能本身已有数据，影响最终结果。
2. 如果目标数据库的校验和全是 0，表示没有发生任何导入，有可能是集群太忙无法接收任何数据。
3. 如果数据源是由机器生成而不是从 Dumpling 备份的，需确保数据符合表的限制，例如：
    - 自增 (AUTO_INCREMENT) 的列需要为正数，不能为 0。
    - 唯一键和主键 (UNIQUE and PRIMARY KEYs) 不能有重复的值。
4. 如果 TiDB Lightning 之前失败停机过，但没有正确重启，可能会因为数据不同步而出现校验和不一致。

解决办法：

1. 使用 `tidb-lightning-ctl` 把出错的表删除，然后重启 TiDB Lightning 重新导入那些表。

    ```
    tidb-lightning-ctl --config conf/tidb-lightning.toml --checkpoint-error-destroy=all
    ```

2. 把断点存放在外部数据库（修改 `[checkpoint] dsn`），减轻目标集群压力。
3. 参考如何正确重启 TiDB Lightning中的解决办法。

11.8.7.22 `Checkpoint for … has invalid status:`（错误码）

原因：断点续传已启用。TiDB Lightning 或 TiKV Importer 之前发生了异常退出。为了防止数据意外损坏，TiDB Lightning 在错误解决以前不会启动。

错误码是小于 25 的整数，可能的取值是 0、3、6、9、12、14、15、17、18、20、21。整数越大，表示异常退出所发生的步骤在导入流程中越晚。

解决办法：

如果错误原因是非法数据源，使用 `tidb-lightning-ctl` 删除已导入数据，并重启 TiDB Lightning。

```
tidb-lightning-ctl --config conf/tidb-lightning.toml --checkpoint-error-destroy=all
```

其他解决方法请参考断点续传的控制。

11.8.7.23 ResourceTemporarilyUnavailable("Too many open engines …: …")

原因：并行打开的引擎文件 (engine files) 超出 tikv-importer 里的限制。这可能由配置错误引起。即使配置没问题，如果 tidb-lightning 曾经异常退出，也有可能令引擎文件残留在打开的状态，占据可用的数量。

解决办法：

1. 提高 tikv-importer.toml 内 max-open-engines 的值。这个设置主要由内存决定，计算公式为：

   最大内存使用量 ≈ max-open-engines × write-buffer-size × max-write-buffer-number

2. 降低 table-concurrency + index-concurrency，使之低于 max-open-engines。

3. 重启 tikv-importer 来强制移除所有引擎文件 (默认值为 ./data.import/)。这样也会丢弃导入了一半的表，所以启动 TiDB Lightning 前必须清除过期的断点记录：

   ```
   tidb-lightning-ctl --config conf/tidb-lightning.toml --checkpoint-error-destroy=all
   ```

11.8.7.24 cannot guess encoding for input file, please convert to UTF-8 manually

原因：TiDB Lightning 只支持 UTF-8 和 GB-18030 编码的表架构。此错误代表数据源不是这里任一个编码。也有可能是文件中混合了不同的编码，例如，因为在不同的环境运行过 ALTER TABLE，使表架构同时出现 UTF-8 和 GB-18030 的字符。

解决办法：

1. 编辑数据源，保存为纯 UTF-8 或 GB-18030 的文件。
2. 手动在目标数量库创建所有的表，然后设置 [mydumper] no-schema = true 跳过创建表的步骤。
3. 设置 [mydumper] character-set = "binary" 跳过这个检查。但是这样可能使数据库出现乱码。

11.8.7.25 [sql2kv] sql encode error = [types:1292]invalid time format: '{1970 1 1 …}'

原因: 一个 timestamp 类型的时间戳记录了不存在的时间值。时间值不存在是由于夏时制切换或超出支持的范围（1970 年 1 月 1 日至 2038 年 1 月 19 日）。

解决办法:

1. 确保 TiDB Lightning 与数据源时区一致。

   - 手动部署的话，通过设定 $TZ 环境变量强制时区设定。

     强制使用 Asia/Shanghai 时区：

     ```
     TZ='Asia/Shanghai' bin/tidb-lightning -config tidb-lightning.toml
     ```

2. 导出数据时，必须加上 --skip-tz-utc 选项。

3. 确保整个集群使用的是同一最新版本的 tzdata (2018i 或更高版本)。

   如果你使用的是 CentOS 机器，你可以运行 yum info tzdata 命令查看 tzdata 的版本及是否有更新。然后运行 yum upgrade tzdata 命令升级 tzdata。

11.8.7.26 [Error 8025: entry too large, the max entry size is 6291456]

原因：TiDB Lightning 生成的单行 KV 超过了 TiDB 的限制。

解决办法:

目前无法绕过 TiDB 的限制，只能忽略这张表，确保其它表顺利导入。

11.8.7.27 switch-mode 时遇到 rpc error: code = Unimplemented ...

原因：集群中有不支持 switch-mode 的节点。目前已知的组件中，4.0.0-rc.2 之前的 TiFlash 不支持 switch-mode 操作。

解决办法：

- 如果集群中有 TiFlash 节点，可以将集群更新到 4.0.0-rc.2 或更新版本。
- 如果不方便升级，可以临时禁用 TiFlash。

11.8.7.28 tidb lightning encountered error: TiDB version too old, expected '>=4.0.0', found '3.0.18'

TiDB Lightning Local-backend 只支持导入到 v4.0.0 及以上版本的 TiDB 集群。如果尝试使用 Local-backend 导入到 v2.x 或 v3.x 的集群，就会报以上错误。此时可以修改配置使用 Importer-backend 或 TiDB-backend 进行导入。

部分 nightly 版本的 TiDB 集群的版本可能类似 4.0.0-beta.2。这种版本的 TiDB Lightning 实际支持 Local-backend，如果使用 nightly 版本遇到该报错，可以通过设置配置 check-requirements = false 跳过版本检查。在设置此参数之前，请确保 TiDB Lightning 的配置支持对应的版本，否则无法保证导入成功。

11.8.7.29 restore table test.district failed: unknown columns in header [...]

出现该错误通常是因为 CSV 格式的数据文件不包含 header（第一行也是数据），因此需要在 TiDB Lightning 的配置文件中增加如下配置项：

```
[mydumper.csv]
header = false
```

11.8.8 TiDB Lightning 术语表

本术语表提供了 TiDB Lightning 相关的术语和定义，这些术语会出现在 TiDB Lightning 的日志、监控指标、配置和文档中。

11.8.8.1 A

11.8.8.1.1 Analyze

统计信息分析。指重建 TiDB 表中的统计信息，即运行ANALYZE TABLE 语句。

因为 TiDB Lightning 不通过 TiDB 导入数据，统计信息不会自动更新，所以 TiDB Lightning 在导入后显式地分析每个表。如果不需要该操作，可以将 post-restore.analyze 设置为 false。

11.8.8.1.2 AUTO_INCREMENT_ID

用于为自增列分配默认值的自增 ID 计数器。每张表都有一个相关联的 AUTO_INCREMENT_ID 计数器。在 TiDB 中，该计数器还用于分配行 ID。

因为 TiDB Lightning 不通过 TiDB 导入数据，AUTO_INCREMENT_ID 计数器不会自动更新，所以 TiDB Lightning 显式地将 AUTO_INCREMENT_ID 改为一个有效值。即使表中没有自增列，这步仍是会执行。

11.8.8.2 B

11.8.8.2.1 Backend

也称作 Back end（后端），用于接受 TiDB Lightning 解析结果。

详情参阅TiDB Lightning Backends。

11.8.8.3 C

11.8.8.3.1 Checkpoint

断点。用于保证 TiDB Lightning 在导入数据时不断地将进度保存到本地文件或远程数据库中。这样即使进程崩溃，TiDB Lightning 也能从中间状态恢复。

详情参见TiDB Lightning 断点续传。

11.8.8.3.2 Checksum

校验和。一种用于验证导入数据正确性的方法。

在 TiDB Lightning 中，表的校验和是由 3 个数字组成的集合，由该表中每个键值对的内容计算得出。这些数字分别是：

- 键值对的数量
- 所有键值对的总长度
- 每个键值对 CRC-64-ECMA 按位异或的结果

TiDB Lightning 通过比较每个表的本地校验和和远程校验和来验证导入数据的正确性。如果有任一对校验和不匹配，导入进程就会停止。如果你需要跳过校验和检查，可以将 `post-restore.checksum` 设置为 `false`。

遇到校验和不匹配的问题时，参考常见问题进行处理。

11.8.8.3.3 Chunk

一段连续的源数据，通常相当于数据源中的单个文件。

如果单个文件太大，TiDB Lightning 可以将单个文件拆分成多个文件块。

11.8.8.3.4 Compaction

压缩。指将多个小 SST 文件合并为一个大 SST 文件并清理已删除的条目。TiDB Lightning 导入数据时，TiKV 在后台会自动压缩数据。

> **注意：**
>
> 出于遗留原因，你仍然可以将 TiDB Lightning 配置为在每次导入表时进行显式压缩，但是官方不推荐采用该操作，且该操作的相关设置默认是禁用的。
>
> 技术细节参阅 RocksDB 关于压缩的说明。

11.8.8.4 D

11.8.8.4.1 Data engine

数据引擎。用于对实际的行数据进行排序的引擎。

当一个表数据很多的时候，表的数据会被放置在多个数据引擎中以改善任务流水线并节省 TiKV Importer 的空间。默认条件下，每 100 GB 的 SQL 数据会打开一个新的数据引擎（可通过 `mydumper.batch-size` 配置项进行更改）。

TiDB Lightning 同时处理多个数据引擎（可通过 `lightning.table-concurrency` 配置项进行更改）。

11.8.8.5 E

11.8.8.5.1 Engine

引擎。在 TiKV Importer 中，一个引擎就是一个用于排序键值对的 RocksDB 实例。

TiDB Lightning 通过引擎将数据传送到 TiKV Importer 中。TiDB Lightning 先打开一个引擎，向其发送未排序的键值对，然后关闭引擎。随后，引擎会对收到的键值对进行排序操作。这些关闭的引擎可以进一步上传至 TiKV store 中为Ingest 做准备。

引擎使用 TiKV Importer 的 `import-dir` 作为临时存储，有时也会被称为引擎文件 (engine files)。

另见数据引擎和索引引擎。

11.8.8.6 F

11.8.8.6.1 Filter

配置列表，用于指定需要导入或不允许导入的表。

详情见表库过滤。

11.8.8.7 I

11.8.8.7.1 Import mode

导入模式。指通过降低读取速度和减少空间使用，来优化 TiKV 写入的配置模式。

导入过程中，TiDB Lightning 自动在导入模式和普通模式中来回切换。如果 TiKV 卡在导入模式，你可以使用 `tidb-lightning-ctl` 强制切换回普通模式。

11.8.8.7.2 Index engine

索引引擎。用于对索引进行排序的引擎。

不管表中有多少索引，每张表都只对应一个索引引擎。

TiDB Lightning 可同时处理多个索引引擎（可通过 `lightning.index-concurrency` 配置项进行更改）。由于每张表正好对应一个索引引擎，`lightning.index-concurrency` 配置项也限定了可同时处理的表的最大数量。

11.8.8.7.3 Ingest

指将SST 文件的全部内容插入到 RocksDB（TiKV）store 中的操作。

与逐个插入键值对相比，Ingest 的效率非常高。因此，该操作直接决定了 TiDB Lightning 的性能。

技术细节参阅 RocksDB 关于创建、Ingest SST 文件的 wiki 页面。

11.8.8.8 K

11.8.8.8.1 KV pair

即 key-value pair（键值对）。

11.8.8.8.2 KV encoder

用于将 SQL 或 CSV 行解析为键值对的例程。多个 KV encoder 会并行运行以加快处理速度。

11.8.8.9 L

11.8.8.9.1 Local checksum

本地校验和。在将键值对发送到 TiKV Importer 前，由 TiDB Lightning 计算的表的校验和。

11.8.8.10 N

11.8.8.10.1 Normal mode

普通模式。未启用导入模式时的模式。

11.8.8.11 P

11.8.8.11.1 Post-processing

指整个数据源被解析发送到 TiKV Importer 之后的一段时间。此时 TiDB Lightning 正在等待 TiKV Importer 上传、Ingest SST 文件。

11.8.8.12 R

11.8.8.12.1 Remote checksum

远程校验和。指导入 TiDB 后所计算的表的校验和。

11.8.8.13 S

11.8.8.13.1 Scattering

指随机再分配Region 中 leader 和 peer 的操作。Scattering 确保导入的数据在 TiKV store 中均匀分布，这样可以降低 PD 调度的压力。

11.8.8.13.2 Splitting

指 TiKV Importer 在上传之前会将单个引擎文件拆分为若干小SST 文件的操作。这是因为引擎文件通常很大（约为 100 GB），在 TiKV 中不适合视为单一的Region。拆分的文件大小可通过 `import.region-split-size` 配置项更改。

11.8.8.13.3 SST file

Sorted string table file（排序字符串表文件）。SST 文件是一种在 RocksDB 中（因而也是 TiKV 中）键值对集合在本地的存储形式。

TiKV Importer 从关闭的引擎中生成 SST 文件。这些 SST 文件接着被上传、 ingest 到 TiKV store 中。

11.9 TiDB Data Migration

TiDB Data Migration (DM) 是一款便捷的数据迁移工具，支持从与 MySQL 协议兼容的数据库（MySQL、 MariaDB、Aurora MySQL）到 TiDB 的全量数据迁移和增量数据同步。使用 DM 工具有利于简化数据迁移过程，降低数据迁移运维成本。

11.9.1 版本说明

DM 工具的稳定版本包括 v5.3、v2.0、v1.0。其中，v1.0 为较旧的版本，不推荐使用。建议使用 DM 的最新稳定版本 v5.3。

DM 工具的文档目前独立于 TiDB 文档。要访问 DM 工具的文档，请点击以下链接：

- DM v5.3 文档
- DM v2.0 文档

- DM v1.0 文档

> 注意：
>
> - DM 的 GitHub 代码仓库已于 2021 年 10 月迁移至 pingcap/tiflow。如有任何关于 DM 的问题，请在 pingcap/tiflow 仓库提交，以获得后续反馈。
> - 在较早版本中（v1.0 和 v2.0），DM 采用独立于 TiDB 的版本号。从 DM v5.3 起，DM 采用与 TiDB 相同的版本号。DM v2.0 的下一个版本为 DM v5.3。DM v2.0 到 v5.3 无兼容性变更，升级过程与正常升级无差异。

11.9.2 基本功能

本节介绍 DM 工具的核心功能模块。

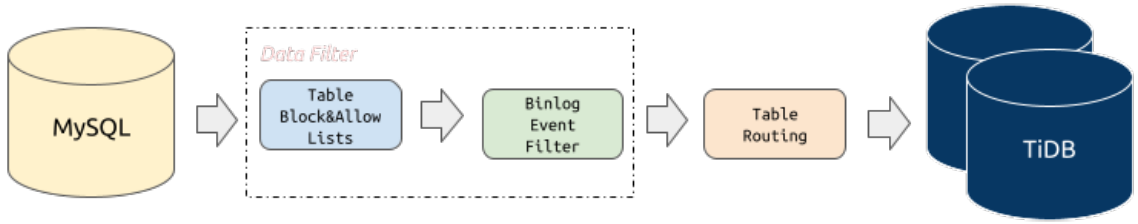


图 175: DM Core Features

11.9.2.1 Block & allow lists

Block & Allow Lists 的过滤规则类似于 MySQL `replication-rules-db/replication-rules-table`，用于过滤或指定只迁移某些数据库或某些表的所有操作。

11.9.2.2 Binlog event filter

Binlog Event Filter 用于过滤源数据库中特定表的特定类型操作，比如过滤掉表 `test.sbtest` 的 `INSERT` 操作或者过滤掉库 `test` 下所有表的 `TRUNCATE TABLE` 操作。

11.9.2.3 Table routing

Table Routing 是将源数据库的表迁移到下游指定表的路由功能，比如将源数据表 test.sbtest1 的表结构和数据迁移到 TiDB 的表 test.sbtest2。它也是分库分表合并迁移所需的一个核心功能。

11.9.3 高级功能

11.9.3.1 分库分表合并迁移

DM 支持对源数据的分库分表进行合并迁移，但有一些使用限制，详细信息请参考悲观模式分库分表合并迁移使用限制和乐观模式分库分表合并迁移使用限制。

11.9.3.2 对第三方 Online Schema Change 工具变更过程的同步优化

在 MySQL 生态中，gh-ost 与 pt-osc 等工具被广泛使用，DM 对其变更过程进行了特殊的优化，以避免对不必要的中间数据进行迁移。详细信息可参考 online-ddl。

11.9.3.3 使用 SQL 表达式过滤某些行变更

在增量同步阶段，DM 支持配置 SQL 表达式过滤掉特定的行变更，以实现对同步数据的更精细控制。详细信息可参考使用 SQL 表达式过滤某些行变更。

11.9.4 使用限制

在使用 DM 工具之前，需了解以下限制：

- 数据库版本要求
  - MySQL 版本 > 5.5
  - MariaDB 版本 >= 10.1.2

> **注意：**
> 如果上游 MySQL/MariaDB servers 间构成主从复制结构，则需要 MySQL 版本高于 5.7.1 或者 MariaDB 版本等于或高于 10.1.3。

> **警告：**
> 使用 DM 从 MySQL v8.0 迁移数据到 TiDB 目前为实验特性（从 DM v2.0 引入），不建议在生产环境下使用。

- DDL 语法兼容性限制
  - 目前，TiDB 部分兼容 MySQL 支持的 DDL 语句。因为 DM 使用 TiDB parser 来解析处理 DDL 语句，所以目前仅支持 TiDB parser 支持的 DDL 语法。详见TiDB DDL 语法支持。

- DM 遇到不兼容的 DDL 语句时会报错。要解决此报错，需要使用 dmctl 手动处理，要么跳过该 DDL 语句，要么用指定的 DDL 语句来替换它。详见如何处理不兼容的 DDL 语句。

- 分库分表数据冲突合并
  - 如果业务分库分表之间存在数据冲突，可以参考自增主键冲突处理来解决；否则不推荐使用 DM 进行迁移，如果进行迁移则有冲突的数据会相互覆盖造成数据丢失。
  - 分库分表 DDL 同步限制，参见悲观模式下分库分表合并迁移使用限制以及乐观模式下分库分表合并迁移使用限制。

- 数据源 MySQL 实例切换
  - 当 DM-worker 通过虚拟 IP（VIP）连接到 MySQL 且要切换 VIP 指向的 MySQL 实例时，DM 内部不同的 connection 可能会同时连接到切换前后不同的 MySQL 实例，造成 DM 拉取的 binlog 与从上游获取到的其他状态不一致，从而导致难以预期的异常行为甚至数据损坏。如需切换 VIP 指向的 MySQL 实例，请参考虚拟 IP 环境下的上游主从切换对 DM 手动执行变更。

11.10 TiCDC

11.10.1 TiCDC 简介

> **注意：**
>
> TiCDC 从 v4.0.6 起成为正式功能，可用于生产环境。

TiCDC 是一款通过拉取 TiKV 变更日志实现的 TiDB 增量数据同步工具，具有将数据还原到与上游任意 TSO 一致状态的能力，同时提供开放数据协议 (TiCDC Open Protocol)，支持其他系统订阅数据变更。

11.10.1.1 TiCDC 架构

TiCDC 运行时是一种无状态节点，通过 PD 内部的 etcd 实现高可用。TiCDC 集群支持创建多个同步任务，向多个不同的下游进行数据同步。

TiCDC 的系统架构如下图所示：

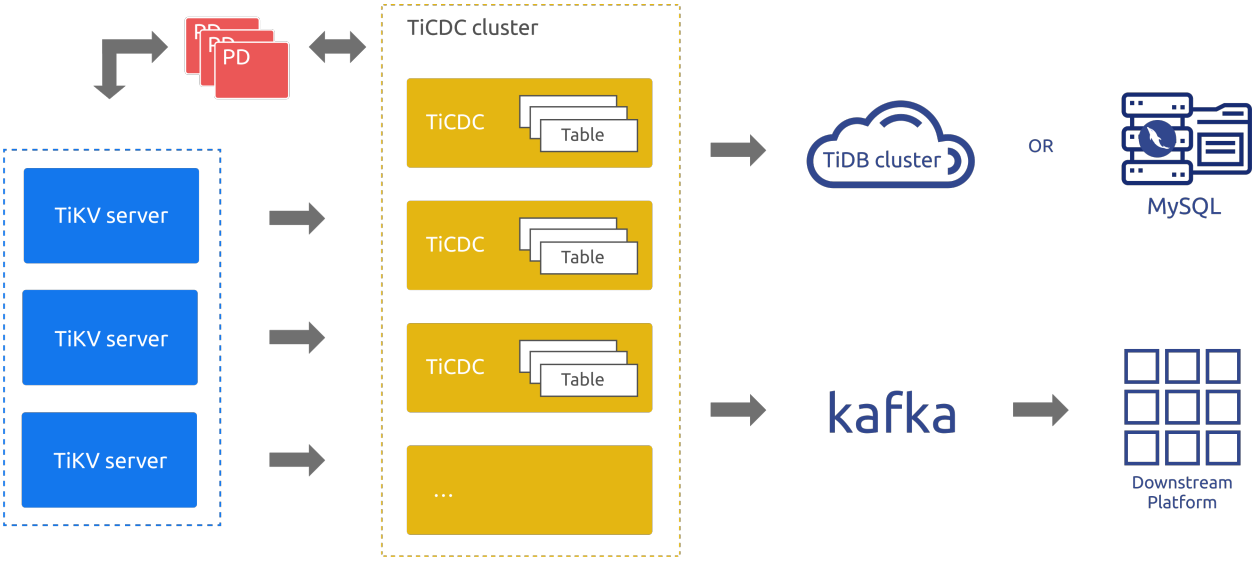


图 176: TiCDC architecture

11.10.1.1.1 系统角色

- TiKV CDC 组件：只输出 key-value (KV) change log。
  - 内部逻辑拼装 KV change log。
  - 提供输出 KV change log 的接口，发送数据包括实时 change log 和增量扫的 change log。
- `capture`：TiCDC 运行进程，多个 `capture` 组成一个 TiCDC 集群，负责 KV change log 的同步。
  - 每个 `capture` 负责拉取一部分 KV change log。
  - 对拉取的一个或多个 KV change log 进行排序。
  - 向下游还原事务或按照 TiCDC Open Protocol 进行输出。

11.10.1.2 同步功能介绍

本部分介绍 TiCDC 的同步功能。

11.10.1.2.1 sink 支持

目前 TiCDC sink 模块支持同步数据到以下下游：

- MySQL 协议兼容的数据库，提供最终一致性支持。
- 以 TiCDC Open Protocol 输出到 Kafka，可实现行级别有序、最终一致性或严格事务一致性三种一致性保证。

11.10.1.2.2 同步顺序保证和一致性保证

数据同步顺序

- TiCDC 对于所有的 DDL/DML 都能对外输出至少一次。

- TiCDC 在 TiKV/TiCDC 集群故障期间可能会重复发相同的 DDL/DML。对于重复的 DDL/DML：
  - MySQL sink 可以重复执行 DDL，对于在下游可重入的 DDL（譬如 truncate table）直接执行成功；对于在下游不可重入的 DDL（譬如 create table），执行失败，TiCDC 会忽略错误继续同步。
  - Kafka sink 会发送重复的消息，但重复消息不会破坏 Resolved Ts 的约束，用户可以在 Kafka 消费端进行过滤。

数据同步一致性

- MySQL sink
  - TiCDC 不拆分单表事务，保证单表事务的原子性。
  - TiCDC 不保证下游事务的执行顺序和上游完全一致。
  - TiCDC 以表为单位拆分跨表事务，不保证跨表事务的原子性。
  - TiCDC 保证单行的更新与上游更新顺序一致。
- Kafka sink
  - TiCDC 提供不同的数据分发策略，可以按照表、主键或 ts 等策略分发数据到不同 Kafka partition。
  - 不同分发策略下 consumer 的不同实现方式，可以实现不同级别的一致性，包括行级别有序、最终一致性或跨表事务一致性。
  - TiCDC 没有提供 Kafka 消费端实现，只提供了TiCDC 开放数据协议，用户可以依据该协议实现 Kafka 数据的消费端。

11.10.1.3 同步限制

TiCDC 只能同步至少存在一个有效索引的表，有效索引的定义如下：

- 主键 (`PRIMARY KEY`) 为有效索引。
- 同时满足下列条件的唯一索引 (`UNIQUE INDEX`) 为有效索引：
  - 索引中每一列在表结构中明确定义非空 (`NOT NULL`)。
  - 索引中不存在虚拟生成列 (`VIRTUAL GENERATED COLUMNS`)。

TiCDC 从 v4.0.8 版本开始，可通过修改任务配置来同步没有有效索引的表，但在数据一致性的保证上有所减弱。具体使用方法和注意事项参考同步没有有效索引的表。

11.10.1.3.1 暂不支持的场景

目前 TiCDC（4.0 发布版本）暂不支持的场景如下：

- 暂不支持单独使用 RawKV 的 TiKV 集群。
- 暂不支持 TiDB 4.0 中创建 SEQUENCE 的 DDL 操作 和SEQUENCE 函数。在上游 TiDB 使用 SEQUENCE 时，TiCDC 将会忽略掉上游执行的 SEQUENCE DDL 操作/函数，但是使用 SEQUENCE 函数的 DML 操作可以正确地同步。
- 暂不支持TiKV Hibernate Region。TiCDC 会使 Region 无法进入静默状态。
- 对上游存在较大事务的场景提供部分支持，详见：FAQ：TiCDC 是否支持同步大事务？有什么风险吗？。

11.10.1.4 TiCDC 安装和部署

要安装 TiCDC，可以选择随新集群一起部署，也可以对现有 TiDB 集群新增 TiCDC 组件。详请参阅TiCDC 安装部署。

11.10.1.5 TiCDC 集群管理和同步任务管理

目前支持使用 `cdc cli` 工具或 HTTP 接口来管理 TiCDC 集群状态和数据同步任务。详细操作见：

- 使用 `cdc cli` 工具来管理集群状态和数据同步
- 使用 HTTP 接口管理集群状态和数据同步

11.10.1.6 TiCDC 常见问题

在使用 TiCDC 过程中经常遇到的问题以及相对应的解决方案请参考TiCDC 常见问题。

11.10.1.7 TiCDC 开放数据协议

TiCDC Open Protocol 是一种行级别的数据变更通知协议，为监控、缓存、全文索引、分析引擎、异构数据库的主从复制等提供数据源。TiCDC 遵循 TiCDC Open Protocol，向 MQ (Message Queue) 等第三方数据媒介复制 TiDB 的数据变更。详细信息参考TiCDC 开放数据协议。

11.10.1.8 `sort-dir` 及 `data-dir` 配置项的兼容性说明

`sort-dir` 配置项用于给 TiCDC 内部的排序器指定临时文件目录，其作用在各版本有过如下兼容性更改：

版本	sort-engine 的使用	说明	使用建议
版本	sort-engine 的使用	说明	使用建议
v4.0.11 及之前的 v4.0 版本，v5.0.0-rc	作为 changefeed 配置项，给 sorter 和 Sorter 指定临时文件目录	在这些版本中，sorter 和 sorter 均不是正式功能 (GA)，不推荐在生产环境中使用。如果有多个 changefeed 被配	不推荐在生产环境中使用 Unified Sorter

版本	sort-engine 的使用	说明	使用建议
v4.0.12，v4.0.13，v5.0.0 及 v5.0.1	作为 change-feed 配置项或 cdc server 配置项	在默认情况下 change-feed 的 sort-dir 配置不会生效，而 cdc server 的 sort-dir 配置默认为 /tmp/cdc_sort。	需要通过 cdc server 命令行参数（或 TiUP）配置 sort-dir

版本	sort-engine 的使用	说明	使用建议
v4.0.14 及之后的 v4.0 版本，v5.0.3 及之后的 v5.0 版本，更新的版本	sort-dir 被弃用，建议配置 data-dir	data-dir 可以通过最新版本的 TiUP 进行配置。这些版本中 unified sorter 是默认开启的，升级时请确保 data-dir 已经被正	需要通过 cdc server 命令行参数（或 TiUP）配置 data-dir

版本	sort-engine 的使用说明	使用建议

11.10.2 TiCDC 安装部署

本文档介绍 TiCDC 集群的软硬件环境要求，以及如何安装部署 TiCDC 集群。你可以选择随新集群一起部署 TiCDC，也可以对原有 TiDB 集群新增 TiCDC 组件。通常推荐使用 TiUP 完成部署，如有特殊情况也可以用 binary 部署。

11.10.2.1 软件和硬件环境推荐配置

在生产环境中，TiCDC 的软件和硬件配置推荐如下：

Linux 操作系统平台	版本
Red Hat Enterprise Linux	7.3 及以上
CentOS	7.3 及以上

CPU	内存	硬盘类型	网络	TiCDC 集群实例数量（生产环境最低要求）
16 核 +	64 GB+	SSD	万兆网卡（2 块最佳）	2

更多信息参见TiDB 软件和硬件环境建议配置。

11.10.2.2 使用 TiUP 部署包含 TiCDC 组件的全新 TiDB 集群

在使用 TiUP 部署全新 TiDB 集群时，支持同时部署 TiCDC 组件。只需在 TiUP 启动 TiDB 集群时的配置文件中加入 TiCDC 部分即可，详细操作参考编辑初始化配置文件，具体可配置字段参考通过 TiUP 配置 cdc_servers，具体可配置字段参考通过 TiUP 配置 cdc_servers。

11.10.2.3 使用 TiUP 在原有 TiDB 集群上新增 TiCDC 组件

目前也支持在原有 TiDB 集群上使用 TiUP 新增 TiCDC 组件，操作步骤如下：

1. 首先确认当前 TiDB 的版本支持 TiCDC，否则需要先升级 TiDB 集群至 4.0.0 rc.1 或更新版本。TiCDC 在 4.0.6 版本已经 GA，建议使用 4.0.6 及以后的版本。
2. 参考扩容 TiDB/TiKV/PD/TiCDC 节点章节对 TiCDC 进行部署。

11.10.2.4 使用 binary 在原有 TiDB 集群上新增 TiCDC 组件（不推荐）

假设 PD 集群有一个可以提供服务的 PD 节点（client URL 为 10.0.10.25:2379）。若要部署三个 TiCDC 节点，可以按照以下命令启动集群。只需要指定相同的 PD 地址，新启动的节点就可以自动加入 TiCDC 集群。

```
cdc server --pd=http://10.0.10.25:2379 --log-file=ticdc_1.log --addr=0.0.0.0:8301 --advertise-
    ↪ addr=127.0.0.1:8301
cdc server --pd=http://10.0.10.25:2379 --log-file=ticdc_2.log --addr=0.0.0.0:8302 --advertise-
    ↪ addr=127.0.0.1:8302
cdc server --pd=http://10.0.10.25:2379 --log-file=ticdc_3.log --addr=0.0.0.0:8303 --advertise-
    ↪ addr=127.0.0.1:8303
```

11.10.2.5 TiCDC cdc server 命令行参数说明

对于 cdc server 命令中可用选项解释如下：

- gc-ttl：TiCDC 在 PD 设置的服务级别 GC safepoint 的 TTL (Time To Live) 时长，和 TiCDC 同步任务所能够停滞的时长。单位为秒，默认值为 86400，即 24 小时。注意：TiCDC 同步任务的停滞会影响 TiCDC GC safepoint 的推进，即会影响上游 TiDB GC 的推进，详情可以参考TiCDC GC safepoint 的完整行为 。
- pd：PD client 的 URL。
- addr：TiCDC 的监听地址，提供服务的 HTTP API 查询地址和 Prometheus 查询地址。
- advertise-addr：TiCDC 对外访问地址。
- tz：TiCDC 服务使用的时区。TiCDC 在内部转换 timestamp 等时间数据类型和向下游同步数据时使用该时区，默认为进程运行本地时区。(注意如果同时指定 tz 参数和 sink-uri 中的 time-zone 参数，TiCDC 进程内部使用 tz 指定的时区，sink 向下游执行时使用 time-zone 指定的时区)
- log-file：TiCDC 进程运行日志的地址，默认为 cdc.log。
- log-level：TiCDC 进程运行时默认的日志级别，默认为 info。
- ca：TiCDC 使用的 CA 证书文件路径，PEM 格式，可选。
- cert：TiCDC 使用的证书文件路径，PEM 格式，可选。
- key：TiCDC 使用的证书密钥文件路径，PEM 格式，可选。
- config：可选项，表示 TiCDC 使用的配置文件地址。TiCDC 从 v5.0.0 开始支持该选项，TiUP 从 v1.4.0 开始支持在部署 TiCDC 时使用该配置。
- data-dir：指定 TiCDC 需要使用磁盘储存文件时使用的目录。目前 Unified Sorter 会使用该目录储存临时文件，请确保该目录所在设备可用空间充足。对于使用 TiUP 的用户，本选项可以通过配置 cdc_servers 小节中的 data_dir 来指定或默认使用 global 中 data_dir 路径。自 TiDB v4.0.14 起有效.
- sort-dir：指定排序引擎使用的临时文件目录。自 TiDB v4.0.13 起已经无效，请不要使用。

11.10.3 TiCDC 运维操作及任务管理

本文档介绍如何通过 TiCDC 提供的命令行工具 cdc cli 和 HTTP 接口两种方式来管理 TiCDC 集群和同步任务。

11.10.3.1 使用 TiUP 升级 TiCDC

本部分介绍如何使用 TiUP 来升级 TiCDC 集群。在以下例子中，假设需要将 TiCDC 组件和整个 TiDB 集群升级到 v4.0.16。

```
tiup update --self && \
tiup update --all && \
tiup cluster upgrade <cluster-name> v4.0.16
```

11.10.3.1.1 升级的注意事项

- TiCDC v4.0.2 对 changefeed 的配置做了调整，请参阅配置文件兼容注意事项。
- 升级期间遇到的问题及其解决办法，请参阅使用 TiUP 升级 TiDB。

11.10.3.2 使用加密传输 (TLS) 功能

请参阅为 TiDB 组件间通信开启加密传输。

11.10.3.3 使用 cdc cli 工具来管理集群状态和数据同步

本部分介绍如何使用 cdc cli 工具来管理集群状态和数据同步。cdc cli 是指通过 cdc binary 执行 cli 子命令。在以下描述中，通过 cdc binary 直接执行 cli 命令，PD 的监听 IP 地址为 10.0.10.25，端口为 2379。

> **注意：**
>
> PD 监听的 IP 和端口对应为 pd-server 启动时指定的 advertise-client-urls 参数。多个 pd-server 会包含多个该参数，用户可以指定其中任意一个或多个参数。例如 --pd=http://10.0.10.25:2379 或 --pd=http://10.0.10.25:2379,http://10.0.10.26:2379,http://10.0.10.27:2379。

如果你使用的 TiCDC 是用 TiUP 部署的，需要将以下命令中的 cdc cli 替换为 tiup ctl:<cluster-version> cdc。

11.10.3.3.1 管理 TiCDC 服务进程 (capture)

- 查询 capture 列表：

```
cdc cli capture list --pd=http://10.0.10.25:2379
```

```
[
  {
    "id": "806e3a1b-0e31-477f-9dd6-f3f2c570abdd",
    "is-owner": true,
    "address": "127.0.0.1:8300"
  },
  {
    "id": "ea2a4203-56fe-43a6-b442-7b295f458ebc",
    "is-owner": false,
    "address": "127.0.0.1:8301"
  }
]
```

  - id：服务进程的 ID。
  - is-owner：表示该服务进程是否为 owner 节点。
  - address：该服务进程对外提供接口的地址。

11.10.3.3.2 管理同步任务 (changefeed)

同步任务状态流转

本功能适用于 4.0.16 及更新版本。

同步任务状态标识了同步任务的运行情况。在 TiCDC 运行过程中，同步任务可能会运行出错、手动暂停、恢复，或达到指定的 TargetTs，这些行为都可以导致同步任务状态发生变化。本节描述 TiCDC 同步任务的各状态以及状态之间的流转关系。

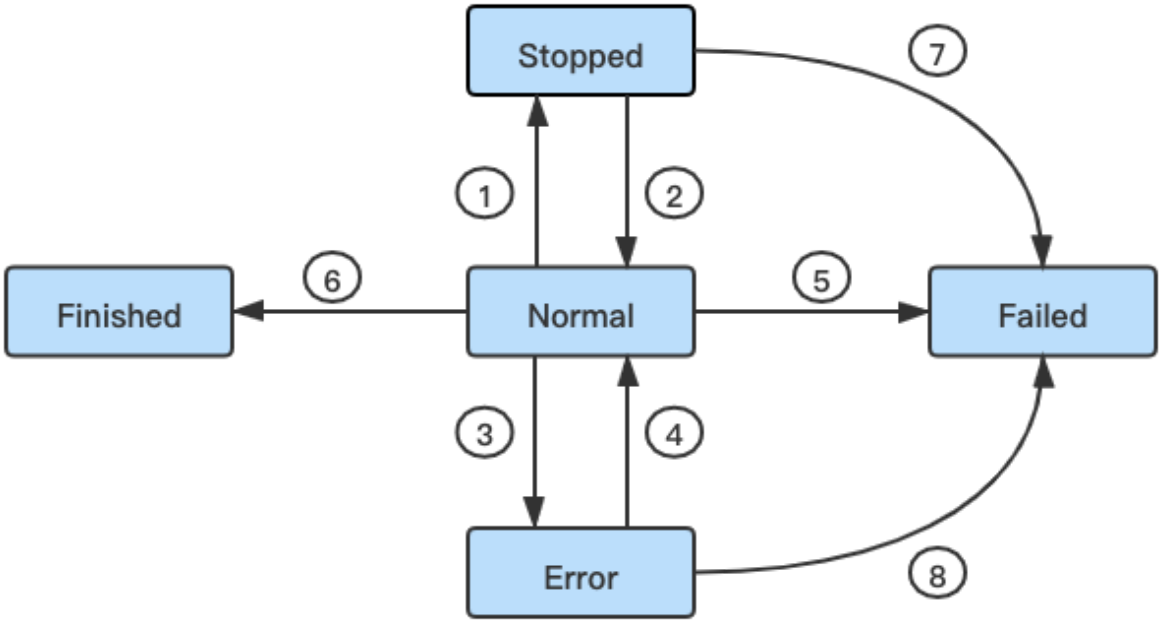


图 177: TiCDC state transfer

以上状态流转图中的状态说明如下：

- Normal：同步任务正常进行，checkpoint-ts 正常推进。
- Stopped：同步任务停止，由于用户手动暂停 (pause) changefeed。处于这个状态的 changefeed 会阻挡 GC 推进。
- Error：同步任务报错，由于某些可恢复的错误导致同步无法继续进行，处于这个状态的 changefeed 会不断尝试继续推进，直到状态转为 Normal。处于这个状态的 changefeed 会阻挡 GC 推进。
- Finished：同步任务完成，同步任务进度已经达到预设的 TargetTs。处于这个状态的 changefeed 不会阻挡 GC 推进。
- Failed：同步任务失败。由于发生了某些不可恢复的错误，导致同步无法继续进行，并且无法恢复。处于这个状态的 changefeed 不会阻挡 GC 推进。

以上状态流转图中的编号说明如下：

- ① 执行 changefeed pause 命令。

- ② 执行 changefeed resume 恢复同步任务。
- ③ changefeed 运行过程中发生可恢复的错误，自动进行恢复。
- ④ 执行 changefeed resume 恢复同步任务。
- ⑤ changefeed 运行过程中发生不可恢复的错误。
- ⑥ changefeed 已经进行到预设的 TargetTs，同步自动停止。
- ⑦ changefeed 停滞时间超过 gc-ttl 所指定的时长，不可被恢复。
- ⑧ changefeed 尝试自动恢复过程中发生不可恢复的错误。

创建同步任务

使用以下命令来创建同步任务：

```
cdc cli changefeed create --pd=http://10.0.10.25:2379 --sink-uri="mysql://root:123456@127
    ↪ .0.0.1:3306/" --changefeed-id="simple-replication-task" --sort-engine="unified"
```

```
Create changefeed successfully!
ID: simple-replication-task
Info: {"sink-uri":"mysql://root:123456@127.0.0.1:3306/","opts":{},"create-time":"2020-03-12T22
    ↪ :04:08.103600025+08:00","start-ts":415241823337054209,"target-ts":0,"admin-job-type":0,"
    ↪ sort-engine":"unified","sort-dir":".","config":{"case-sensitive":true,"filter":{"rules
    ↪ ":["*.*"],"ignore-txn-start-ts":null,"ddl-allow-list":null},"mounter":{"worker-num":16},"
    ↪ sink":{"dispatchers":null,"protocol":"default"},"cyclic-replication":{"enable":false,"
    ↪ replica-id":0,"filter-replica-ids":null,"id-buckets":0,"sync-ddl":false},"scheduler":{"
    ↪ type":"table-number","polling-time":-1}},"state":"normal","history":null,"error":null}
```

- --changefeed-id：同步任务的 ID，格式需要符合正则表达式 ^[a-zA-Z0-9]+(\-[a-zA-Z0-9]+)*$。如果不指定该 ID，TiCDC 会自动生成一个 UUID（version 4 格式）作为 ID。
- --sink-uri：同步任务下游的地址，需要按照以下格式进行配置，目前 scheme 支持 mysql/tidb/kafka ↪ /pulsar。

  ```
  [scheme]://[userinfo@][host]:[port][/path]?[query_parameters]
  ```

  URI 中包含特殊字符时，需要以 URL 编码对特殊字符进行处理。
- --start-ts：指定 changefeed 的开始 TSO。TiCDC 集群将从这个 TSO 开始拉取数据。默认为当前时间。
- --target-ts：指定 changefeed 的目标 TSO。TiCDC 集群拉取数据直到这个 TSO 停止。默认为空，即 TiCDC 不会自动停止。
- --sort-engine：指定 changefeed 使用的排序引擎。因 TiDB 和 TiKV 使用分布式架构，TiCDC 需要对数据变更记录进行排序后才能输出。该项支持 unified（默认）/memory/file：
  - unified：优先使用内存排序，内存不足时则自动使用硬盘暂存数据。该选项默认开启。
  - memory：在内存中进行排序。不建议使用，同步大量数据时易引发 OOM。
  - file：完全使用磁盘暂存数据。已经弃用，不建议在任何情况使用。
- --config：指定 changefeed 配置文件。

- `--sort-dir`: 用于指定排序器使用的临时文件目录。自 TiDB v4.0.13 起已经无效，请不要使用。

Sink URI 配置 `mysql/tidb`

配置样例如下所示：

```
--sink-uri="mysql://root:123456@127.0.0.1:3306/?worker-count=16&max-txn-row=5000"
```

URI 中可配置的的参数如下：

参数	解析
`root`	下游数据库的用户名
`123456`	下游数据库密码
`127.0.0.1`	下游数据库的 IP
`3306`	下游数据的连接端口
`worker-count`	向下游执行 SQL 的并发度（可选，默认值为 16）
`max-txn-row`	向下游执行 SQL 的 batch 大小（可选，默认值为 256）
`ssl-ca`	连接下游 MySQL 实例所需的 CA 证书文件路径（可选）
`ssl-cert`	连接下游 MySQL 实例所需的证书文件路径（可选）
`ssl-key`	连接下游 MySQL 实例所需的证书密钥文件路径（可选）
`time-zone`	连接下游 MySQL 实例时使用的时区名称，从 v4.0.8 开始生效。（可选。如果不指定该参数，使用 TiCDC 服务进程的时区；如果指定该参数但使用空值，则表示连接 MySQL 时不指定时区，使用下游默认时区）

Sink URI 配置 `kafka`

配置样例如下所示：

```
--sink-uri="kafka://127.0.0.1:9092/topic-name?kafka-version=2.4.0&partition-num=6&max-message-
    ↪ bytes=67108864&replication-factor=1"
```

URI 中可配置的的参数如下：

参数	解析
`127.0.0.1`	下游 Kafka 对外提供服务的 IP
`9092`	下游 Kafka 的连接端口
`topic-name`	变量，使用的 Kafka topic 名字
`kafka-version`	下游 Kafka 版本号（可选，默认值 `2.4.0`，目前支持的最低版本为 `0.11.0.2`，最高版本为 `2.7.0`。该值需要与下游 Kafka 的实际版本保持一致）
`kafka-client-id`	指定同步任务的 Kafka 客户端的 ID（可选，默认值为 `TiCDC_sarama_producer_`同步任务的 `ID`）
`partition-num`	下游 Kafka partition 数量（可选，不能大于实际 partition 数量，否则创建同步任务会失败。在 v4.0.16 前，该参数默认值为 4；自 v4.0.16 起，默认值为 3。）
`max-message-bytes`	每次向 Kafka broker 发送消息的最大数据量（可选。在 v4.0.16 前，该参数默认值为 `512MB`；v4.0.16 该参数默认值为 `1MB`；自 v4.0.17 起，默认值为 `10MB`。）

参数	解析
replication-factor	kafka 消息保存副本数（可选，默认值 1）
protocol	输出到 kafka 消息协议，可选值有 default、canal、avro、maxwell（默认值为 default）
auto-create-topic	当传入的 topic-name 在 Kafka 集群不存在时，TiCDC 是否要自动创建该 topic（可选，默认值 true）
max-batch-size	从 v4.0.9 引入。如果消息协议支持将多条变更记录输出到一条 kafka 消息，该参数指定一条 kafka 消息中变更记录的最多数量，目前仅对 Kafka 的 protocol 为 default 时有效（可选，默认值为 16。自 v4.0.13 起，默认值由 4096 改为 16）
ca	连接下游 Kafka 实例所需的 CA 证书文件路径（可选）
cert	连接下游 Kafka 实例所需的证书文件路径（可选）
key	连接下游 Kafka 实例所需的证书密钥文件路径（可选）

最佳实践：

- TiCDC 推荐用户自行创建 Kafka Topic，你至少需要设置该 Topic 每次向 Kafka broker 发送消息的最大数据量和下游 Kafka partition 的数量。在创建 changefeed 的时候，这两项设置分别对应 max-message-bytes 和 partition-num 参数。
- 如果你在创建 changefeed 时，使用了尚未存在的 Topic，那么 TiCDC 会尝试使用 partition-num 和 replication-factor 参数自行创建 Topic。建议明确指定这两个参数。

> 注意：
>
> 当 protocol 为 default 时，TiCDC 会尽量避免产生长度超过 max-message-bytes 的消息。但如果单条数据变更记录需要超过 max-message-bytes 个字节来表示，为了避免静默失败，TiCDC 会试图输出这条消息并在日志中输出 Warning。

TiCDC 集成 Kafka Connect (Confluent Platform)

> 警告：
>
> 当前该功能为实验特性，不建议在生产环境中使用。

配置样例如下所示：

```
--sink-uri="kafka://127.0.0.1:9092/topic-name?kafka-version=2.4.0&protocol=avro&partition-num=6&
    ↪ max-message-bytes=67108864&replication-factor=1"
--opts registry="http://127.0.0.1:8081"
```

如要使用 Confluent 提供的 data connectors 向关系型或非关系型数据库传输数据，应当选择 avro 协议，并在 opts 中提供 Confluent Schema Registry 的 URL。请注意，avro 协议和 Confluent 集成目前均为实验特性。

集成具体步骤详见TiDB 集成 Confluent Platform 快速上手指南。

Sink URI 配置 pulsar

配置样例如下所示：

```
--sink-uri="pulsar://127.0.0.1:6650/topic-name?connectionTimeout=2s"
```

URI 中可配置的的参数如下：

参数	解析
connectionTimeout	连接下游 Pulsar 的超时时间。可选参数，默认值为 30s。
operationTimeout	对下游 Pulsar 进行操作的超时时间（例如创建 topic）。可选参数，默认值为 30s。
tlsTrustCertsFilePath	连接下游 Pulsar 实例所需的 CA 证书文件路径（可选）
tlsAllowInsecureConnection	在开启 TLS 之后是否允许非加密连接（可选）
tlsValidateHostname	是否校验下游 Pulsar 证书中的 host name（可选）
maxConnectionsPerBroker	下游单个 Pulsar broker 最多允许的连接数（可选，默认值为 1）
auth.tls	使用 TLS 模式认证下游 Pulsar（可选，示例 auth=tls&auth.tlsCertFile=/path/to/cert&auth.tlsKeyFile=/path/to/key）
auth.token	使用 token 模式认证下游（可选，示例 auth=token&auth.token=secret-token 或者 auth=token&auth.file=path/to/secret-token-file）
name	TiCDC 中 Pulsar producer 名字（可选）
maxPendingMessages	Pending 消息队列的最大大小，例如，等待接收来自 Pulsar 的确认的消息（可选，默认值为 1000）
disableBatching	禁止自动批量发送消息（可选）
batchingMaxPublishDelay	设置发送消息的批处理时间（默认值为 10ms）
compressionType	设置发送消息时使用的压缩算法（可选 NONE，LZ4，ZLIB 和 ZSTD，默认值为 NONE）
hashingScheme	用于选择发送分区的哈希算法（可选 JavaStringHash 和 Murmur3，默认值为 JavaStringHash）
properties.*	在 TiCDC 中 Pulsar producer 上添加用户定义的属性（可选，示例 properties.location=Hangzhou）

更多关于 Pulsar 的参数解释，参见 pulsar-client-go ClientOptions 文档 和 pulsar-client-go ProducerOptions 文档

使用同步任务配置文件

如需设置更多同步任务的配置，比如指定同步单个数据表，请参阅同步任务配置文件描述。

使用配置文件创建同步任务的方法如下：

```
cdc cli changefeed create --pd=http://10.0.10.25:2379 --sink-uri="mysql://root:123456@127
    ↪ .0.0.1:3306/" --config changefeed.toml
```

其中 changefeed.toml 为同步任务的配置文件。

查询同步任务列表

使用以下命令来查询同步任务列表：

```
cdc cli changefeed list --pd=http://10.0.10.25:2379
```

```
[{
    "id": "simple-replication-task",
    "summary": {
      "state": "normal",
      "tso": 417886179132964865,
      "checkpoint": "2020-07-07 16:07:44.881",
      "error": null
    }
}]
```

- checkpoint 即为 TiCDC 已经将该时间点前的数据同步到了下游。
- state 为该同步任务的状态：
  - normal: 正常同步
  - stopped: 停止同步（手动暂停或出错）
  - removed: 已删除任务（只在指定 --all 选项时才会显示该状态的任务。未指定时，可通过 query 查询该状态的任务）
  - finished: 任务已经同步到指定 target-ts，处于已完成状态（只在指定 --all 选项时才会显示该状态的任务。未指定时，可通过 query 查询该状态的任务）

查询特定同步任务

使用 changefeed query 命令可以查询特定同步任务（对应某个同步任务的信息和状态），指定 --simple 或 -s 参数会简化输出，提供最基本的同步状态和 checkpoint 信息。不指定该参数会输出详细的任务配置、同步状态和同步表信息。

```
cdc cli changefeed query -s --pd=http://10.0.10.25:2379 --changefeed-id=simple-replication-task
```

```
{
 "state": "normal",
 "tso": 419035700154597378,
 "checkpoint": "2020-08-27 10:12:19.579",
 "error": null
}
```

以上命令中：

- `state` 代表当前 changefeed 的同步状态，各个状态必须和 `changefeed list` 中的状态相同。
- `tso` 代表当前 changefeed 中已经成功写入下游的最大事务 TSO。
- `checkpoint` 代表当前 changefeed 中已经成功写入下游的最大事务 TSO 对应的时间。
- `error` 记录当前 changefeed 是否有错误发生。

```
cdc cli changefeed query --pd=http://10.0.10.25:2379 --changefeed-id=simple-replication-task
```

```
{
  "info": {
    "sink-uri": "mysql://127.0.0.1:3306/?max-txn-row=20\u0026worker-number=4",
    "opts": {},
    "create-time": "2020-08-27T10:33:41.687983832+08:00",
    "start-ts": 419036036249681921,
    "target-ts": 0,
    "admin-job-type": 0,
    "sort-engine": "unified",
    "sort-dir": ".",
    "config": {
      "case-sensitive": true,
      "enable-old-value": false,
      "filter": {
        "rules": [
          "*.*"
        ],
        "ignore-txn-start-ts": null,
        "ddl-allow-list": null
      },
      "mounter": {
        "worker-num": 16
      },
      "sink": {
        "dispatchers": null,
        "protocol": "default"
      },
      "cyclic-replication": {
        "enable": false,
        "replica-id": 0,
        "filter-replica-ids": null,
        "id-buckets": 0,
        "sync-ddl": false
      },
      "scheduler": {
        "type": "table-number",
        "polling-time": -1
      }
```

```
    },
    "state": "normal",
    "history": null,
    "error": null
  },
  "status": {
    "resolved-ts": 419036036249681921,
    "checkpoint-ts": 419036036249681921,
    "admin-job-type": 0
  },
  "count": 0,
  "task-status": [
    {
      "capture-id": "97173367-75dc-490c-ae2d-4e990f90da0f",
      "status": {
        "tables": {
          "47": {
            "start-ts": 419036036249681921,
            "mark-table-id": 0
          }
        },
        "operation": null,
        "admin-job-type": 0
      }
    }
  ]
}
```

以上命令中：

- `info` 代表查询 changefeed 的同步配置。
- `status` 代表查询 changefeed 的同步状态信息。
  - `resolved-ts` 代表当前 changefeed 中已经成功从 TiKV 发送到 TiCDC 的最大事务 TS。
  - `checkpoint-ts` 代表当前 changefeed 中已经成功写入下游的最大事务 TS。
  - `admin-job-type` 代表一个 changefeed 的状态：
    * 0: 状态正常。
    * 1: 任务暂停，停止任务后所有同步 `processor` 会结束退出，同步任务的配置和同步状态都会保留，可以从 `checkpoint-ts` 恢复任务。
    * 2: 任务恢复，同步任务从 `checkpoint-ts` 继续同步。
    * 3: 任务已删除，接口请求后会结束所有同步 `processor`，并清理同步任务配置信息。同步状态保留，只提供查询，没有其他实际功能。
- `task-status` 代表查询 changefeed 所分配的各个同步子任务的状态信息。

11.10.3.3.3 停止同步任务

使用以下命令来停止同步任务：

```
cdc cli changefeed pause --pd=http://10.0.10.25:2379 --changefeed-id simple-replication-task
```

以上命令中：

- `--changefeed-id=uuid` 为需要操作的 changefeed ID。

11.10.3.3.4 恢复同步任务

使用以下命令恢复同步任务：

```
cdc cli changefeed resume --pd=http://10.0.10.25:2379 --changefeed-id simple-replication-task
```

以上命令中：

- `--changefeed-id=uuid` 为需要操作的 changefeed ID。

11.10.3.3.5 删除同步任务

使用以下命令删除同步任务：

```
cdc cli changefeed remove --pd=http://10.0.10.25:2379 --changefeed-id simple-replication-task
```

- `--changefeed-id=uuid` 为需要操作的 changefeed ID。

删除任务后会保留任务的同步状态信息 24 小时（主要用于记录同步的 checkpoint），24 小时内不能创建同名的任务。如果希望彻底删除任务信息，可以指定 `--force` 或 `-f` 参数删除，删除后 changefeed 的所有信息都会被清理，可以立即创建同名的 changefeed。

```
cdc cli changefeed remove --pd=http://10.0.10.25:2379 --changefeed-id simple-replication-task --
    ↪ force
```

11.10.3.3.6 更新同步任务配置

TiCDC 从 v4.0.4 开始支持非动态修改同步任务配置，修改 changefeed 配置需要按照 暂停任务 -> 修改配置 -> ↪ 恢复任务 的流程。

```
cdc cli changefeed pause -c test-cf --pd=http://10.0.10.25:2379
cdc cli changefeed update -c test-cf --pd=http://10.0.10.25:2379 --sink-uri="mysql
    ↪ ://127.0.0.1:3306/?max-txn-row=20&worker-number=8" --config=changefeed.toml
cdc cli changefeed resume -c test-cf --pd=http://10.0.10.25:2379
```

当前支持修改的配置包括：

- changefeed 的 `sink-uri`
- changefeed 配置文件及文件内所有配置
- changefeed 是否使用文件排序和排序目录
- changefeed 的 `target-ts`

11.10.3.3.7 管理同步子任务处理单元 (processor)

- 查询 processor 列表：

```
cdc cli processor list --pd=http://10.0.10.25:2379
```

```
[
        {
                "id": "9f84ff74-abf9-407f-a6e2-56aa35b33888",
                "capture-id": "b293999a-4168-4988-a4f4-35d9589b226b",
                "changefeed-id": "simple-replication-task"
        }
]
```

- 查询特定 processor，对应于某个节点处理的同步子任务信息和状态：

```
cdc cli processor query --pd=http://10.0.10.25:2379 --changefeed-id=simple-replication-task
    ↪ --capture-id=b293999a-4168-4988-a4f4-35d9589b226b
```

```
{
  "status": {
    "tables": {
      "56": {    # 56 表示同步表 id，对应 TiDB 中表的 tidb_table_id
        "start-ts": 417474117955485702,
        "mark-table-id": 0  # mark-table-id 是用于环形复制时标记表的 id，对应于 TiDB
            ↪ 中标记表的 tidb_table_id
      }
    },
    "operation": null,
    "admin-job-type": 0
  },
  "position": {
    "checkpoint-ts": 417474143881789441,
    "resolved-ts": 417474143881789441,
    "count": 0
  }
}
```

以上命令中：

- status.tables 中每一个作为 key 的数字代表同步表的 id，对应 TiDB 中表的 tidb_table_id。
- mark-table-id 是用于环形复制时标记表的 id，对应于 TiDB 中标记表的 tidb_table_id。
- resolved-ts 代表当前 processor 中已经排序数据的最大 TSO。
- checkpoint-ts 代表当前 processor 已经成功写入下游的事务的最大 TSO。

11.10.3.4 使用 HTTP 接口管理集群状态和数据同步

目前 HTTP 接口提供一些基础的查询和运维功能。在以下接口描述中，假设 TiCDC server 的监听 IP 地址为 `127.0.0.1`，端口为 8300（在启动 TiCDC server 时通过 `--addr=ip:port` 指定绑定的 IP 和端口）。

11.10.3.4.1 获取 TiCDC server 状态信息的接口

使用以下命令获取 CDC server 状态信息的接口：

```
curl http://127.0.0.1:8300/status
```

```
{
 "version": "0.0.1",
 "git_hash": "863f8ea889b144244ff53593a45c47ad22d37396",
 "id": "6d92386a-73fc-43f3-89de-4e337a42b766", # capture id
 "pid": 12102    # cdc server pid
}
```

11.10.3.4.2 驱逐 owner 节点

```
curl -X POST http://127.0.0.1:8300/capture/owner/resign
```

以上命令仅对 owner 节点请求有效。

```
{
 "status": true,
 "message": ""
}
```

```
curl -X POST http://127.0.0.1:8301/capture/owner/resign
```

以上命令对非 owner 节点请求返回错误。

```
election: not leader
```

11.10.3.4.3 手动调度表到其他节点

```
curl -X POST http://127.0.0.1:8300/capture/owner/move_table -d 'cf-id=cf060953-036c-4f31-899f-5
    ↪ afa0ad0c2f9&target-cp-id=6f19a6d9-0f8c-4dc9-b299-3ba7c0f216f5&table-id=49'
```

参数说明

参数名	说明
`cf-id`	进行调度的 Changefeed ID
`target-cp-id`	目标 Capture ID
`table-id`	需要调度的 Table ID

以上命令仅对 owner 节点请求有效。对非 owner 节点将会返回错误。

```
{
 "status": true,
 "message": ""
}
```

11.10.3.4.4 动态调整 TiCDC server 日志级别

```
curl -X POST -d '"debug"' http://127.0.0.1:8301/admin/log
```

POST 参数表示新的日志级别，支持 zap 提供的日志级别：“debug”、“info”、“warn”、“error”、“dpanic”、“panic”、“fatal”。该接口参数为 JSON 编码，需要注意引号的使用：`'"debug"'`。

11.10.3.5 同步任务配置文件描述

本部分详细介绍了同步任务的配置。

```
### 指定配置文件中涉及的库名、表名是否为大小写敏感
### 该配置会同时影响 filter 和 sink 相关配置，默认为 true
case-sensitive = true

### 是否输出 old value，从 v4.0.5 开始支持
enable-old-value = false

[filter]
### 忽略指定 start_ts 的事务
ignore-txn-start-ts = [1, 2]

### 过滤器规则
### 过滤规则语法：https://docs.pingcap.com/zh/tidb/stable/table-filter#表库过滤语法
rules = ['*.*', '!test.*']

[mounter]
### mounter 线程数，用于解码 TiKV 输出的数据
worker-num = 16

[sink]
### 对于 MQ 类的 Sink，可以通过 dispatchers 配置 event 分发器
### 支持 default、ts、rowid、table 四种分发器，分发规则如下：
### - default：有多个唯一索引（包括主键）时按照 table 模式分发；只有一个唯一索引（或主键）按照
    ↪ rowid 模式分发；如果开启了 old value 特性，按照 table 分发
### - ts：以行变更的 commitTs 做 Hash 计算并进行 event 分发
### - rowid：以表的主键或者唯一索引列名和列值做 Hash 计算并进行 event 分发
### - table：以表的 schema 名和 table 名做 Hash 计算并进行 event 分发
### matcher 的匹配语法和过滤器规则语法相同
```

```
dispatchers = [
    {matcher = ['test1.*', 'test2.*'], dispatcher = "ts"},
    {matcher = ['test3.*', 'test4.*'], dispatcher = "rowid"},
]
### 对于 MQ 类的 Sink，可以指定消息的协议格式
### 目前支持 default、canal、avro 和 maxwell 四种协议。default 为 TiCDC Open Protocol
protocol = "default"

[cyclic-replication]
### 是否开启环形同步
enable = false
### 当前 TiCDC 的复制 ID
replica-id = 1
### 需要过滤掉的同步 ID
filter-replica-ids = [2,3]
### 是否同步 DDL
sync-ddl = true
```

11.10.3.5.1 配置文件兼容性的注意事项

- TiCDC v4.0.0 中移除了 `ignore-txn-commit-ts`，添加了 `ignore-txn-start-ts`，使用 start_ts 过滤事务。
- TiCDC v4.0.2 中移除了 `db-dbs/db-tables/ignore-dbs/ignore-tables`，添加了 `rules`，使用新版的数据库和数据表过滤规则，详细语法参考表库过滤。

11.10.3.6 环形同步

> **警告：**
>
> 目前环形同步属于实验特性，尚未经过完备的测试，不建议在生产环境中使用该功能。

环形同步功能支持在多个独立的 TiDB 集群间同步数据。比如有三个 TiDB 集群 A、B 和 C，它们都有一个数据表 `test.user_data`，并且各自对它有数据写入。环形同步功能可以将 A、B 和 C 对 `test.user_data` 的写入同步其它集群上，使三个集群上的 `test.user_data` 达到最终一致。

11.10.3.6.1 环形同步使用示例

在三个集群 A、B 和 C 上开启环形复制，其中 A 到 B 的同步使用两个 TiCDC。A 作为三个集群的 DDL 入口。

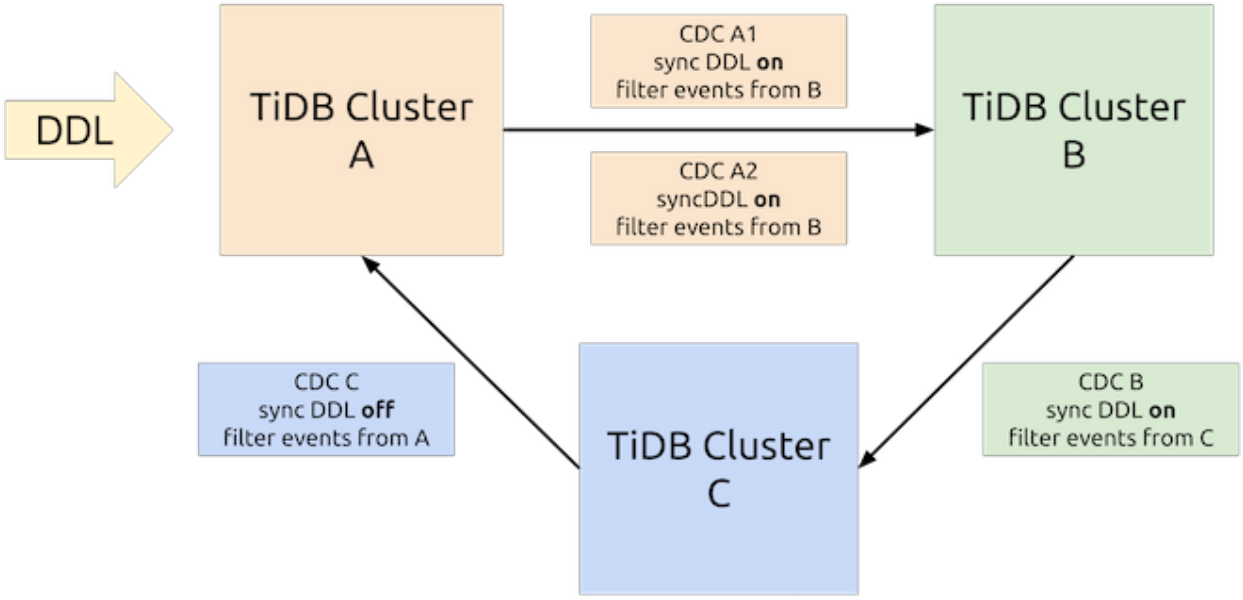


图 178: TiCDC cyclic replication

使用环形同步功能时，需要设置同步任务的创建参数：

- `--cyclic-replica-id`：用于指定为上游集群的写入指定来源 ID，需要确保每个集群 ID 的唯一性。
- `--cyclic-filter-replica-ids`：用于指定需要过滤的写入来源 ID，通常为下游集群的 ID。
- `--cyclic-sync-ddl`：用于指定是否同步 DDL 到下游。

环形同步任务创建步骤如下：

1. 在 TiDB 集群 A，B 和 C 上启动 TiCDC 组件。

```
# 在 TiDB 集群 A 上启动 TiCDC 组件。
cdc server \
    --pd="http://${PD_A_HOST}:${PD_A_PORT}" \
    --log-file=ticdc_1.log \
    --addr=0.0.0.0:8301 \
    --advertise-addr=127.0.0.1:8301

# 在 TiDB 集群 B 上启动 TiCDC 组件。
cdc server \
    --pd="http://${PD_B_HOST}:${PD_B_PORT}" \
    --log-file=ticdc_2.log \
    --addr=0.0.0.0:8301 \
    --advertise-addr=127.0.0.1:8301

# 在 TiDB 集群 C 上启动 TiCDC 组件。
cdc server \
```

```
    --pd="http://${PD_C_HOST}:${PD_C_PORT}" \
    --log-file=ticdc_3.log \
    --addr=0.0.0.0:8301 \
    --advertise-addr=127.0.0.1:8301
```

2. 在 TiDB 集群 A，B 和 C 上创建环形同步需要使用的标记数据表 (mark table)。

```
# 在 TiDB 集群 A 上创建标记数据表。
cdc cli changefeed cyclic create-marktables \
    --cyclic-upstream-dsn="root@tcp(${TIDB_A_HOST}:${TIDB_A_PORT})/" \
    --pd="http://${PD_A_HOST}:${PD_A_PORT}"

# 在 TiDB 集群 B 上创建标记数据表。
cdc cli changefeed cyclic create-marktables \
    --cyclic-upstream-dsn="root@tcp(${TIDB_B_HOST}:${TIDB_B_PORT})/" \
    --pd="http://${PD_B_HOST}:${PD_B_PORT}"

# 在 TiDB 集群 C 上创建标记数据表。
cdc cli changefeed cyclic create-marktables \
    --cyclic-upstream-dsn="root@tcp(${TIDB_C_HOST}:${TIDB_C_PORT})/" \
    --pd="http://${PD_C_HOST}:${PD_C_PORT}"
```

3. 在 TiDB 集群 A，B 和 C 上创建环形同步任务。

```
# 在 TiDB 集群 A 上创建环形同步任务。
cdc cli changefeed create \
    --sink-uri="mysql://root@${TiDB_B_HOST}/" \
    --pd="http://${PD_A_HOST}:${PD_A_PORT}" \
    --cyclic-replica-id 1 \
    --cyclic-filter-replica-ids 2 \
    --cyclic-sync-ddl true

# 在 TiDB 集群 B 上创建环形同步任务。
cdc cli changefeed create \
    --sink-uri="mysql://root@${TiDB_C_HOST}/" \
    --pd="http://${PD_B_HOST}:${PD_B_PORT}" \
    --cyclic-replica-id 2 \
    --cyclic-filter-replica-ids 3 \
    --cyclic-sync-ddl true

# 在 TiDB 集群 C 上创建环形同步任务。
cdc cli changefeed create \
    --sink-uri="mysql://root@${TiDB_A_HOST}/" \
    --pd="http://${PD_C_HOST}:${PD_C_PORT}" \
    --cyclic-replica-id 3 \
    --cyclic-filter-replica-ids 1 \
```

```
    --cyclic-sync-ddl false
```

11.10.3.6.2 环形同步使用说明

- 在创建环形同步任务前，必须使用 `cdc cli changefeed cyclic create-marktables` 创建环形同步功能使用到的标记表。
- 开启环形同步的数据表名字需要符合正则表达式 `^[a-zA-Z0-9_]+$`。
- 在创建环形同步任务前，开启环形复制的数据表必须已创建完毕。
- 开启环形复制后，不能创建一个会被环形同步任务同步的表。
- 在多集群同时写入时，为了避免业务出错，请避免执行 DDL 语句，比如 `ADD COLUMN/DROP COLUMN` 等。
- 如果想在线执行 DDL 语句，需要确保满足以下条件：
  - 业务兼容 DDL 语句执行前后的表结构。
  - 多个集群的 TiCDC 组件构成一个单向 DDL 同步链，不能成环。例如以上在 TiDB 集群 A，B 和 C 上创建环形同步任务的示例中，只有 C 集群的 TiCDC 组件关闭了 `sync-ddl`。
  - DDL 语句必须在单向 DDL 同步链的开始集群上执行，例如示例中的 A 集群。

11.10.3.7 输出行变更的历史值从 v4.0.5 版本开始引入

> **警告：**
>
> 目前输出行变更历史值属于实验特性，尚未经过完备的测试，不建议在生产环境中使用该功能。

在默认配置下同步任务输出的 TiCDC Open Protocol 行变更数据只包含变更后的值，不包含变更前行的值，因此该输出数据不支持 TiDB 4.0 新的 Collation 框架，也不满足 TiCDC Open Protocol 的消费端使用行变更历史值的需求。

从 v4.0.5 开始，TiCDC 支持输出行变更数据的历史值。若要开启该特性，需要在 changefeed 的配置文件的根级别指定以下配置：

```
enable-old-value = true
```

开启该特性后，TiCDC Open Protocol 的输出格式参考TiCDC 开放数据协议 - Row Changed Event，使用 MySQL sink 时也会自动支持的 TiDB 4.0 新 Collation 特性。

11.10.3.8 同步没有有效索引的表

从 v4.0.8 开始，TiCDC 支持通过修改任务配置来同步没有有效索引的表。若要开启该特性，需要在 changefeed 配置文件的根级别进行如下指定：

```
enable-old-value = true
force-replicate = true
```

警告：

对于没有有效索引的表，INSERT 和 REPLACE 等操作不具备可重入性，因此会有数据冗余的风险。TiCDC 在同步过程中只保证数据至少分发一次，因此开启该特性同步没有有效索引的表，一定会导致数据冗余出现。如果不能接受数据冗余，建议增加有效索引，譬如增加具有 AUTO RANDOM 属性的主键列。

11.10.3.9 Unified Sorter 功能

Unified Sorter 是 TiCDC 中的排序引擎功能，自 v4.0.9 引入，用于缓解以下场景造成的内存溢出问题：

- 如果 TiCDC 数据订阅任务的暂停中断时间长，其间积累了大量的增量更新数据需要同步。
- 从较早的时间点启动数据订阅任务，业务写入量大，积累了大量的更新数据需要同步。

对 v4.0.13 版本之后的 `cdc cli` 创建的 changefeed，默认开启 Unified Sorter。对 v4.0.13 版本前已经存在的 changefeed，则使用之前的配置。

要确定一个 changefeed 上是否开启了 Unified Sorter 功能，可执行以下示例命令查看（假设 PD 实例的 IP 地址为 `http://10.0.10.25:2379`）：

```
cdc cli --pd="http://10.0.10.25:2379" changefeed query --changefeed-id=simple-replication-task |
    ↪ grep 'sort-engine'
```

以上命令的返回结果中，如果 `sort-engine` 的值为 “unified”，则说明 Unified Sorter 已在该 changefeed 上开启。

注意：

- 如果服务器使用机械硬盘或其他有延迟或吞吐有瓶颈的存储设备，请谨慎开启 Unified Sorter。
- Unified Sorter 默认使用 `data_dir` 储存临时文件。建议保证硬盘的空闲容量大于等于 500 GiB。对于生产环境，建议保证每个节点上的磁盘可用空间大于（业务允许的最大 `checkpoint-ts` 延迟 * 业务高峰上游写入流量。此外，如果在 changefeed 创建后预期需要同步大量历史数据，请确保每个节点的空闲容量大于等于要追赶的同步数据。
- Unified Sorter 默认开启，如果您的服务器不符合以上条件，并希望关闭 Unified Sorter，请手动将 changefeed 的 `sort-engine` 设为 `memory`。
- 如需在已使用 `memory` 排序的 changefeed 上开启 Unified Sorter，参见同步任务中断，尝试再次启动后 TiCDC 发生 OOM，如何处理回答中提供的方法。

11.10.4 TiCDC 常见问题和故障处理

本文档总结了在使用 TiCDC 过程中经常遇到的问题，给出合适的运维方法。本文档还总结了常见的运行故障，并给出相对应的解决方案。

> **注意：**
>
> 本文档 `cdc cli` 命令中指定 PD 地址为 `--pd=http://10.0.10.25:2379`，用户在使用时需根据实际地址进行替换。

11.10.4.1 TiCDC 创建任务时如何选择 start-ts？

首先需要理解同步任务的 `start-ts` 对应于上游 TiDB 集群的一个 TSO，同步任务会从这个 TSO 开始请求数据。所以同步任务的 `start-ts` 需要满足以下两个条件：

- `start-ts` 的值需要大于 TiDB 集群当前的 `tikv_gc_safe_point`，否则创建任务时会报错。
- 启动任务时，需要保证下游已经具有 `start-ts` 之前的所有数据。对于同步到消息队列等场景，如果不需要保证上下游数据的一致，可根据业务场景放宽此要求。

如果不指定 `start-ts` 或者指定 `start-ts=0`，在启动任务的时候会去 PD 获取一个当前 TSO，并从该 TSO 开始同步。

11.10.4.2 为什么 TiCDC 创建任务时提示部分表不能同步？

在使用 `cdc cli changefeed create` 创建同步任务时会检查上游表是否符合同步限制。如果存在表不满足同步限制，会提示 `some tables are not eligible to replicate` 并列出这些不满足的表。用户选择 `Y` 或 `y` 则会继续创建同步任务，并且同步过程中自动忽略这些表的所有更新。用户选择其他输入，则不会创建同步任务。

11.10.4.3 如何查看 TiCDC 同步任务的状态？

可以使用 `cdc cli` 查询同步任务的状态。例如：

```
cdc cli changefeed list --pd=http://10.0.10.25:2379
```

上述命令输出如下：

```
[{
    "id": "4e24dde6-53c1-40b6-badf-63620e4940dc",
    "summary": {
      "state": "normal",
      "tso": 417886179132964865,
      "checkpoint": "2020-07-07 16:07:44.881",
      "error": null
    }
}]
```

- `checkpoint`：即为 TiCDC 已经将该时间点前的数据同步到了下游。
- `state` 为该同步任务的状态：

- `normal`：正常同步。
- `stopped`：停止同步（手动暂停或出错）。
- `removed`：已删除任务。

> 注意：
>
> 该功能在 TiCDC 4.0.3 版本引入。

11.10.4.4 TiCDC 同步任务出现中断

11.10.4.4.1 如何判断 TiCDC 同步任务出现中断？

- 通过 Grafana 检查同步任务的 `changefeed checkpoint` 监控项。注意选择正确的 `changefeed id`。如果该值不发生变化或者查看 `checkpoint lag` 是否不断增大，可能同步任务出现中断。
- 通过 Grafana 检查 `exit error count` 监控项，该监控项大于 0 代表同步任务出现错误。
- 通过 `cdc cli changefeed list` 和 `cdc cli changefeed query` 命令查看同步任务的状态信息。任务状态为 `stopped` 代表同步中断，`error` 项会包含具体的错误信息。任务出错后可以在 TiCDC server 日志中搜索 `error on running processor` 查看错误堆栈，帮助进一步排查问题。
- 部分极端异常情况下 TiCDC 出现服务重启，可以在 TiCDC server 日志中搜索 FATAL 级别的日志排查问题。

11.10.4.4.2 如何查看 TiCDC 同步任务是否被人为终止？

可以使用 `cdc cli` 查询同步任务是否被人为终止。例如：

```
cdc cli changefeed query --pd=http://10.0.10.25:2379 --changefeed-id 28c43ffc-2316-4f4f-a70b-
    ↪ d1a7c59ba79f
```

上述命令的输出中 `admin-job-type` 标志这个同步的任务的状态：

- 0: 任务进行中，没有被人为停止。
- 1: 任务暂停，停止任务后所有同步 `processor` 会结束退出，同步任务的配置和同步状态都会保留，可以从 `checkpoint-ts` 恢复任务。
- 2: 任务恢复，同步任务从 `checkpoint-ts` 继续同步。
- 3: 任务已删除，接口请求后会结束所有同步 `processor`，并清理同步任务配置信息。同步状态保留，只提供查询，没有其他实际功能。

11.10.4.4.3 如何处理 TiCDC 同步任务的中断？

目前已知可能发生的同步中断包括以下场景：

- 下游持续异常，TiCDC 多次重试后仍然失败。
  - 该场景下 TiCDC 会保存任务信息，由于 TiCDC 已经在 PD 中设置的 service GC safepoint，在 `gc-ttl` 的有效期内，同步任务 checkpoint 之后的数据不会被 TiKV GC 清理掉。

  - 处理方法：用户可以在下游恢复正常后，通过 HTTP 接口恢复同步任务。

- 因下游存在不兼容的 SQL 语句，导致同步不能继续。
  - 该场景下 TiCDC 会保存任务信息，由于 TiCDC 已经在 PD 中设置的 service GC safepoint，在 `gc-ttl` 的有效期内，同步任务 checkpoint 之后的数据不会被 TiKV GC 清理掉。
  - 处理方法：
    1. 用户需先通过 `cdc cli changefeed query` 查询同步任务状态信息，记录 `checkpoint-ts` 值。
    2. 使用新的任务配置文件，增加`ignore-txn-start-ts` 参数跳过指定 `start-ts` 对应的事务。
    3. 通过 HTTP API 停止旧的同步任务，使用 `cdc cli changefeed create` ，指定新的任务配置文件，指定 `start-ts` 为刚才记录的 `checkpoint-ts`，启动新的同步任务恢复同步。

- 在 v4.0.13 及之前的版本中 TiCDC 同步分区表存在问题，导致同步停止。
  - 该场景下 TiCDC 会保存任务信息，由于 TiCDC 已经在 PD 中设置的 service GC safepoint，在 `gc-ttl` 的有效期内，同步任务 checkpoint 之后的数据不会被 TiKV GC 清理掉。
  - 处理方法：
    1. 通过 `cdc cli changefeed pause -c <changefeed-id>` 暂停同步。
    2. 等待约一分钟后，通过 `cdc cli changefeed resume -c <changefeed-id>` 恢复同步。

11.10.4.4.4 同步任务中断，尝试再次启动后 TiCDC 发生 OOM，应该如何处理？

升级 TiDB 集群和 TiCDC 集群到最新版本。该 OOM 问题在 v4.0.14 及之后的 v4.0 版本，v5.0.2 及之后的 v5.0 版本，更新的版本上已得到缓解。

在这些版本上，可以开启 Unified Sorter 排序功能，该功能会在系统内存不足时使用磁盘进行排序。启用的方式是创建同步任务时在 `cdc cli` 内传入 `--sort-engine=unified`，使用示例如下：

```
cdc cli changefeed update -c <changefeed-id> --sort-engine="unified" --pd=http://10.0.10.25:2379
```

如果无法升级到上述版本，需要在之前的版本上开启 Unified Sorter，可以在创建同步任务时在 `cdc cli` 内传入 `--sort-engine=unified` 和 `--sort-dir=/path/to/sort_dir`，使用示例如下：

```
cdc cli changefeed update -c <changefeed-id> --sort-engine="unified" --sort-dir="/data/cdc/sort"
    ↪ --pd=http://10.0.10.25:2379
```

> 注意：
>
> - TiCDC 从 4.0.9 版本起支持 Unified Sorter 排序引擎。
> - TiCDC（4.0 发布版本）还不支持动态修改排序引擎。在修改排序引擎设置前，请务必确保 changefeed 已经停止 (stopped)。
> - `sort-dir` 在不同版本之间有不同的行为，请参考sort-dir 及 data-dir 配置项的兼容性说明，谨慎配置。
> - 目前 Unified Sorter 排序引擎为实验特性，在数据表较多 (>= 100) 时可能出现性能问题，影响同步速度，故不建议在生产环境中使用。开启 Unified Sorter 前请保证各 TiCDC 节点机器上有足够硬盘空间。如果积攒的数据总量有可能超过 1 TB，则不建议使用 TiCDC 进行同步。

11.10.4.5 TiCDC 的 gc-ttl 是什么？

从 TiDB v4.0.0-rc.1 版本起，PD 支持外部服务设置服务级别 GC safepoint。任何一个服务可以注册更新自己服务的 GC safepoint。PD 会保证任何晚于该 GC safepoint 的 KV 数据不会在 TiKV 中被 GC 清理掉。

在 TiCDC 中启用了这一功能，用来保证 TiCDC 在不可用、或同步任务中断情况下，可以在 TiKV 内保留 TiCDC 需要消费的数据不被 GC 清理掉。

启动 TiCDC server 时可以通过 `gc-ttl` 指定 GC safepoint 的 TTL，默认值为 24 小时。在 TiCDC 中这个值有如下两重含义：

- 当 TiCDC 服务全部停止后，由 TiCDC 在 PD 所设置的 GC safepoint 保存的最长时间。
- TiCDC 中某个同步任务中断或者被手动停止时所能停滞的最长时间，若同步任务停滞时间超过 `gc-ttl` 所设置的值，那么该同步任务就会进入 `failed` 状态，无法被恢复，并且不会继续影响 GC safepoint 的推进。

以上第二种行为是在 TiCDC v4.0.13 版本及之后版本中新增的。目的是为了防止 TiCDC 中某个同步任务停滞时间过长，导致上游 TiKV 集群的 GC safepoint 长时间不推进，保留的旧数据版本过多，进而影响上游集群性能。

> 注意在某些应用场景中，比如使用 Dumpling/BR 全量同步后使用 TiCDC 接增量同步时，默认的 `gc-ttl` 为 24 小时可能无法满足需求。此时应该根据实际情况，在启动 TiCDC server 时指定 `gc-ttl` 的值。

11.10.4.6 TiCDC GC safepoint 的完整行为是什么

TiCDC 服务启动后，如果有任务开始同步，TiCDC owner 会根据所有同步任务最小的 checkpoint-ts 更新到 PD service GC safepoint，service GC safepoint 可以保证该时间点及之后的数据不被 GC 清理掉。如果 TiCDC 中某个同步任务中断、或者被用户主动停止，则该任务的 checkpoint-ts 不会再改变，PD 对应的 service GC safepoint 最终会停滞在该任务的 checkpoint-ts 处不再更新。

如果该同步任务停滞的时间超过了 `gc-ttl` 指定的时长，那么该同步任务就会进入 `failed` 状态，并且无法被恢复，PD 对应的 service GC safepoint 就会继续推进。

TiCDC 为 service GC safepoint 设置的存活有效期为 24 小时，即 TiCDC 服务中断 24 小时内恢复能保证数据不因 GC 而丢失。

11.10.4.7 如何处理 TiCDC 创建同步任务或同步到 MySQL 时遇到 `Error 1298: Unknown or incorrect time zone: 'UTC'` 错误？

这是因为下游 MySQL 没有加载时区，可以通过 mysql_tzinfo_to_sql 命令加载时区，加载后就可以正常创建任务或同步任务。

```
mysql_tzinfo_to_sql /usr/share/zoneinfo | mysql -u root mysql -p
```

显示类似于下面的输出则表示导入已经成功：

```
Enter password:
Warning: Unable to load '/usr/share/zoneinfo/iso3166.tab' as time zone. Skipping it.
Warning: Unable to load '/usr/share/zoneinfo/leap-seconds.list' as time zone. Skipping it.
Warning: Unable to load '/usr/share/zoneinfo/zone.tab' as time zone. Skipping it.
Warning: Unable to load '/usr/share/zoneinfo/zone1970.tab' as time zone. Skipping it.
```

如果下游是特殊的 MySQL 环境（某种公有云 RDS 或某些 MySQL 衍生版本等），使用上述方式导入时区失败，就需要通过 sink-uri 中的 `time-zone` 参数指定下游的 MySQL 时区。可以首先在 MySQL 中查询其使用的时区：

```
show variables like '%time_zone%';
```

```
+------------------+--------+
| Variable_name    | Value  |
+------------------+--------+
| system_time_zone | CST    |
| time_zone        | SYSTEM |
+------------------+--------+
```

然后在创建同步任务和创建 TiCDC 服务时使用该时区：

```
cdc cli changefeed create --sink-uri="mysql://root@127.0.0.1:3306/?time-zone=CST" --pd=http
    ↪ ://10.0.10.25:2379
```

> 注意：
>
> CST 可能是以下四个不同时区的缩写：
>
> - 美国中部时间：Central Standard Time (USA) UT-6:00
> - 澳大利亚中部时间：Central Standard Time (Australia) UT+9:30
> - 中国标准时间：China Standard Time UT+8:00
> - 古巴标准时间：Cuba Standard Time UT-4:00
>
> 在中国，CST 通常表示中国标准时间，使用时请注意甄别。

11.10.4.8 如何理解 TiCDC 时区和上下游数据库系统时区之间的关系？

	上游时区	TiCDC 时区	下游时区
配置方式	见时区支持	启动 ticdc server 时的 -- ↪ tz ↪ 参数	sink-uri 中的 time ↪ -zone ↪ 参数

	上游时区	TiCDC 时区	下游时区
说明	上游 TiDB 的时区，影响 timestamp 类型的 DML 操作和与 timestamp 类型列相关的 DDL 操作。	TiCDC 会将假设上游 TiDB 的时区和 TiCDC 时区配置相同，对 timestamp 类型的列进行相关处理。	下游 MySQL 将按照下游的时区设置对 DML 和 DDL 操作中包含的 timestamp 进行处理。

注意：

请谨慎设置 TiCDC server 的时区，因为该时区会用于时间类型的转换。上游时区、TiCDC 时区和下游时区应该保持一致。TiCDC server 时区使用的优先级如下：

- 最优先使用 `--tz` 传入的时区。

- 没有 `--tz` 参数，会尝试读取 TZ 环境变量设置的时区。
- 如果还没有 TZ 环境变量，会从 TiCDC server 运行机器的默认时区。

11.10.4.9 创建同步任务时，如果不指定 `--config` 配置文件，TiCDC 的默认的行为是什么？

在使用 `cdc cli changefeed create` 命令时如果不指定 `--config` 参数，TiCDC 会按照以下默认行为创建同步任务：

- 同步所有的非系统表
- 不开启 old value 功能
- 不同步不包含有效索引的表

11.10.4.10 如何处理升级 TiCDC 后配置文件不兼容的问题？

请参阅配置文件兼容注意事项。

11.10.4.11 TiCDC 是否支持输出 Canal 格式的变更数据？

支持。要开启 Canal 格式输出，只需在 `--sink-uri` 中指定 protocol 为 `canal` 即可，例如：

```
cdc cli changefeed create --pd=http://10.0.10.25:2379 --sink-uri="kafka://127.0.0.1:9092/cdc-test
    ↪ ?kafka-version=2.4.0&protocol=canal" --config changefeed.toml
```

注意：

- 该功能在 TiCDC 4.0.2 版本引入。
- 目前 TiCDC 仅支持将 Canal 格式的变更数据输出到 Kafka。

更多信息请参考创建同步任务。

11.10.4.12 为什么 TiCDC 到 Kafka 的同步任务延时越来越大？

- 请参考如何查看 TiCDC 同步任务的状态？ 检查下同步任务的状态是否正常。
- 请适当调整 Kafka 的以下参数：
  - `message.max.bytes`，将 Kafka 的 `server.properties` 中该参数调大到 1073741824 (1 GB)。
  - `replica.fetch.max.bytes`，将 Kafka 的 `server.properties` 中该参数调大到 1073741824 (1 GB)。
  - `fetch.message.max.bytes`，适当调大 `consumer.properties` 中该参数，确保大于 `message.max.bytes`。

11.10.4.13 TiCDC 把数据同步到 Kafka 时，是把一个事务内的所有变更都写到一个消息中吗？如果不是，是根据什么划分的？

不是，根据配置的分发策略不同，有不同的划分方式，包括 `default`、`row id`、`table`、`ts`。更多请参考同步任务配置文件描述。

11.10.4.14 TiCDC 把数据同步到 Kafka 时，能在 TiDB 中控制单条消息大小的上限吗？

- 从 v4.0.9 起，可以通过 `max-message-bytes` 控制每次向 Kafka broker 发送消息的最大数据量（可选，默认值 10MB）；通过 `max-batch-size` 参数指定每条 kafka 消息中变更记录的最大数量，目前仅对 Kafka 的 `protocol` 为 `default` 时有效（可选，默认值 16）。
- v4.0.9 以前，TiCDC 向 Kafka 发送的消息批量的大小最大为 512MB。

11.10.4.15 TiCDC 把数据同步到 Kafka 时，一条消息中会不会包含多种数据变更？

会，一条消息中可能出现多个 `update` 或 `delete`，`update` 和 `delete` 也有可能同时存在。

11.10.4.16 TiCDC 把数据同步到 Kafka 时，如何查看 TiCDC Open protocol 输出变更数据中的时间戳、表名和库名？

这些信息包含在 Kafka 消息的 Key 中，比如：

```
{
    "ts":<TS>,
    "scm":<Schema Name>,
    "tbl":<Table Name>,
    "t":1
}
```

更多信息请参考Open protocol Event 格式定义

11.10.4.17 TiCDC 把数据同步到 Kafka 时，如何确定一条消息中包含的数据变更发生在哪个时间点？

把 Kafka 消息的 Key 中的 `ts` 右移 18 位即得 unix timestamp。

11.10.4.18 TiCDC Open protocol 如何标示 null 值？

Open protocol 的输出中 type = 6 即为 null，比如：

类型	Code	输出示例	说明
Null	6	`{"t":6,"v":null}`	

更多信息请参考Open protocol Event 格式定义。

11.10.4.19 TiCDC 启动任务的 start-ts 时间戳与当前时间差距较大，任务执行过程中同步中断，出现错误 [CDC:ErrBufferReachLimit]

自 v4.0.9 起可以尝试开启 unified sorter 特性进行同步；或者使用 BR 工具进行一次增量备份和恢复，然后从新的时间点开启 TiCDC 同步任务。TiCDC 将会在后续版本中对该问题进行优化。

11.10.4.20 如何区分 TiCDC Open Protocol 中的 Row Changed Event 是 INSERT 事件还是 UPDATE 事件？

如果没有开启 Old Value 功能，用户无法区分 TiCDC Open Protocol 中的 Row Changed Event 是 INSERT 事件还是 UPDATE 事件。如果开启了 Old Value 功能，则可以通过事件中的字段判断事件类型：

- 如果同时存在 "p" 和 "u" 字段为 UPDATE 事件
- 如果只存在 "u" 字段则为 INSERT 事件
- 如果只存在 "d" 字段则为 DELETE 事件

更多信息请参考Open protocol Row Changed Event 格式定义。

11.10.4.21 TiCDC 占用多少 PD 的存储空间

TiCDC 使用 PD 内部的 etcd 来存储元数据并定期更新。因为 etcd 的多版本并发控制 (MVCC) 以及 PD 默认的 compaction 间隔是 1 小时，TiCDC 占用的 PD 存储空间与 1 小时内元数据的版本数量成正比。在 v4.0.5、v4.0.6、v4.0.7 三个版本中 TiCDC 存在元数据写入频繁的问题，如果 1 小时内有 1000 张表创建或调度，就会用尽 etcd 的存储空间，出现 `etcdserver: mvcc: database space exceeded` 错误。出现这种错误后需要清理 etcd 存储空间，参考 etcd maintaince space-quota。推荐使用这三个版本的用户升级到 v4.0.9 及以后版本。

11.10.4.22 TiCDC 支持同步大事务吗？有什么风险吗？

TiCDC 对大事务（大小超过 5 GB）提供部分支持，根据场景不同可能存在以下风险：

- 当 TiCDC 内部处理能力不足时，可能出现同步任务报错 `ErrBufferReachLimit`。
- 当 TiCDC 内部处理能力不足或 TiCDC 下游吞吐能力不足时，可能出现内存溢出 (OOM)。

当遇到上述错误时，建议将包含大事务部分的增量数据通过 BR 进行增量恢复，具体操作如下：

1. 记录因为大事务而终止的 changefeed 的 `checkpoint-ts`，将这个 TSO 作为 BR 增量备份的 `--lastbackupts`，并执行增量备份。
2. 增量备份结束后，可以在 BR 日志输出中找到类似 `["Full backup Failed summary : total backup ranges: 0, total success: 0, total failed: 0"] [BackupTS=421758868510212097]` 的日志，记录其中的 BackupTS。
3. 执行增量恢复。
4. 建立一个新的 changefeed，从 BackupTS 开始同步任务。
5. 删除旧的 changefeed。

11.10.4.23 当 changefeed 的下游为类 MySQL 数据库时，TiCDC 执行了一个耗时较长的 DDL 语句，阻塞了所有其他 changefeed，应该怎样处理？

1. 首先暂停执行耗时较长的 DDL 的 changefeed。此时可以观察到，这个 changefeed 暂停后，其他的 changefeed 不再阻塞了。
2. 在 TiCDC log 中搜寻 `apply job` 字段，确认耗时较长的 DDL 的 `start-ts`。如果 TiCDC 版本小于或等于 v4.0.13，日志中没有打印该 `start-ts`，可以查询 TiDB DDL history，找到该条 DDL 语句的 `binlog.TableInfo.update_timestamp` 字段，该字段即为所需的 `start-ts`。
3. 手动在下游执行该 DDL 语句，执行完毕后进行下面的操作。
4. 修改 changefeed 配置，将上述 `start-ts` 添加到 `ignore-txn-start-ts` 配置项中。
5. 恢复被暂停的 changefeed。

11.10.4.24 TiCDC 集群升级到 v4.0.8 之后，changefeed 报错 `[CDC:ErrKafkaInvalidConfig]Canal requires old value to be enabled`

自 v4.0.8 起，如果 changefeed 使用 `canal` 或者 `maxwell` 协议输出，TiCDC 会自动开启 Old Value 功能。但是，如果 TiCDC 是从较旧版本升级到 v4.0.8 或以上版本的，在 changefeed 使用 `canal` 或 `maxwell` 协议的同时 TiCDC 的 Old Value 功能会被禁用。此时，会出现该报错。可以按照以下步骤解决该报错：

1. 将 changefeed 配置文件中 `enable-old-value` 的值设为 `true`。
2. 使用 `cdc cli changefeed pause` 暂停同步任务。

```
cdc cli changefeed pause -c test-cf --pd=http://10.0.10.25:2379
```

3. 使用 `cdc cli changefeed update` 更新原有 changefeed 的配置。

```
cdc cli changefeed update -c test-cf --pd=http://10.0.10.25:2379 --sink-uri="mysql
    ↪ ://127.0.0.1:3306/?max-txn-row=20&worker-number=8" --config=changefeed.toml
```

4. 使用 `cdc cli changefeed resume` 恢复同步任务。

```
cdc cli changefeed resume -c test-cf --pd=http://10.0.10.25:2379
```

11.10.4.25 使用 TiCDC 创建 changefeed 时报错 `[tikv:9006]GC life time is shorter than transaction duration, transaction starts at xx, GC safe point is yy`

解决方案：需要执行 `pd-ctl service-gc-safepoint --pd <pd-addrs>` 命令查询当前的 GC safepoint 与 service GC safepoint。如果 GC safepoint 小于 TiCDC changefeed 同步任务的开始时间戳 `start-ts`，则用户可以直接在 `cdc cli create changefeed` 命令后加上 `--disable-gc-check` 参数创建 changefeed。

如果 `pd-ctl service-gc-safepoint --pd <pd-addrs>` 的结果中没有 `gc_worker service_id`：

- 如果 PD 的版本 <= v4.0.8，详见 PD issue #3128
- 如果 PD 是由 v4.0.8 或更低版本滚动升级到新版，详见 PD issue #3366
- 对于其他情况，请将上述命令执行结果反馈到 AskTUG 论坛。

11.10.4.26 使用 TiCDC 创建同步任务时将 enable-old-value 设置为 true 后，上游的 INSERT/UPDATE 语句经 TiCDC 同步到下游后变为 REPLACE INTO

TiCDC 创建 changefeed 时会默认指定 safe-mode 为 true，从而为上游的 INSERT/UPDATE 语句生成 REPLACE INTO 的执行语句。

目前用户暂时无法修改 safe-mode 设置，因此该问题暂无解决办法。

11.10.4.27 使用 TiCDC 同步消息到 Kafka 时 Kafka 报错 Message was too large

v4.0.8 或更低版本的 TiCDC，仅在 Sink URI 中为 Kafka 配置 max-message-bytes 参数不能有效控制输出到 Kafka 的消息大小，需要在 Kafka server 配置中加入如下配置以增加 Kafka 接收消息的字节数限制。

```
### broker 能接收消息的最大字节数
message.max.bytes=2147483648
### broker 可复制的消息的最大字节数
replica.fetch.max.bytes=2147483648
### 消费者端的可读取的最大消息字节数
fetch.message.max.bytes=2147483648
```

11.10.4.28 TiCDC 同步时，在下游执行 DDL 语句失败会有什么表现，如何恢复？

从 v4.0.11 开始，如果某条 DDL 语句执行失败，同步任务 (changefeed) 会自动停止，checkpoint-ts 断点时间戳为该条出错 DDL 语句的结束时间戳 (finish-ts) 减去一。如果希望让 TiCDC 在下游重试执行这条 DDL 语句，可以使用 cdc cli changefeed resume 恢复同步任务。例如：

```
cdc cli changefeed resume -c test-cf --pd=http://10.0.10.25:2379
```

如果希望跳过这条出错的 DDL 语句，可以将 changefeed 的 start-ts 设为报错时的 checkpoint-ts 加上一，然后通过 cdc cli changefeed resume 恢复同步任务。假设报错时的 checkpoint-ts 为 415241823337054209，可以进行如下操作来跳过该 DDL 语句：

```
cdc cli changefeed update -c test-cf --pd=http://10.0.10.25:2379 --start-ts 415241823337054210

cdc cli changefeed resume -c test-cf --pd=http://10.0.10.25:2379
```

> 注意：
>
> 以上步骤仅适用于 TiCDC v4.0.11 及以上版本。在其它版本中（v4.0.11 以下），DDL 执行失败后 changefeed 的 checkpoint-ts 为该 DDL 语句的 finish-ts。使用 cdc cli changefeed resume 恢复同步任务后不会重试该 DDL 语句，而是直接跳过执行该 DDL 语句。

11.10.4.29 同步 DDL 到下游 MySQL 5.7 时间类型字段默认值不一致

比如上游 TiDB 的建表语句为 `create table test (id int primary key, ts timestamp)`，TiCDC 同步该语句到下游 MySQL 5.7，MySQL 使用默认配置，同步得到的表结构如下所示，timestamp 字段默认值会变成 CURRENT_TIMESTAMP：

```
mysql root@127.0.0.1:test> show create table test;
+-------+-----------------------------------------------------------------------------------------+
| Table | Create Table                                                                            |
+-------+-----------------------------------------------------------------------------------------+
| test  | CREATE TABLE `test` (                                                                   |
|       |   `id` int(11) NOT NULL,                                                                |
|       |   `ts` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,        |
|       |   PRIMARY KEY (`id`)                                                                    |
|       | ) ENGINE=InnoDB DEFAULT CHARSET=latin1                                                  |
+-------+-----------------------------------------------------------------------------------------+
1 row in set
```

产生表结构不一致的原因是 `explicit_defaults_for_timestamp` 的默认值在 TiDB 和 MySQL 5.7 不同。从 TiCDC v4.0.13 版本开始，同步到 MySQL 会自动设置 session 变量 `explicit_defaults_for_timestamp = ON`，保证同步时间类型时上下游行为一致。对于 v4.0.13 以前的版本，同步时间类型时需要注意 `explicit_defaults_for_timestamp` 默认值不同带来的兼容性问题。

11.10.5 TiCDC 重要监控指标详解

使用 TiUP 部署 TiDB 集群时，一键部署的监控系统面板包含 TiCDC 面板。本文档对 TiCDC 监控面板上的各项指标进行详细说明。在日常运维中，运维人员可通过观察 TiCDC 面板上的指标了解 TiCDC 当前的状态。

本文档的对指标的介绍基于以下同步任务，即使用默认配置同步数据到 MySQL。

```
cdc cli changefeed create --pd=http://10.0.10.25:2379 --sink-uri="mysql://root:123456@127
    ↪ .0.0.1:3306/" --changefeed-id="simple-replication-task"
```

下图显示了 TiCDC Dashboard 各监控面板：

图 179: TiCDC Dashboard - Overview

各监控面板说明如下：

- Server：TiDB 集群中 TiKV 节点和 TiCDC 节点的概要信息
- Changefeed：TiCDC 同步任务的详细信息
- Events：TiCDC 内部数据流转的详细信息
- TiKV：TiKV 中和 TiCDC 相关的详细信息

11.10.5.1 Server 面板

Server 面板示例如下：

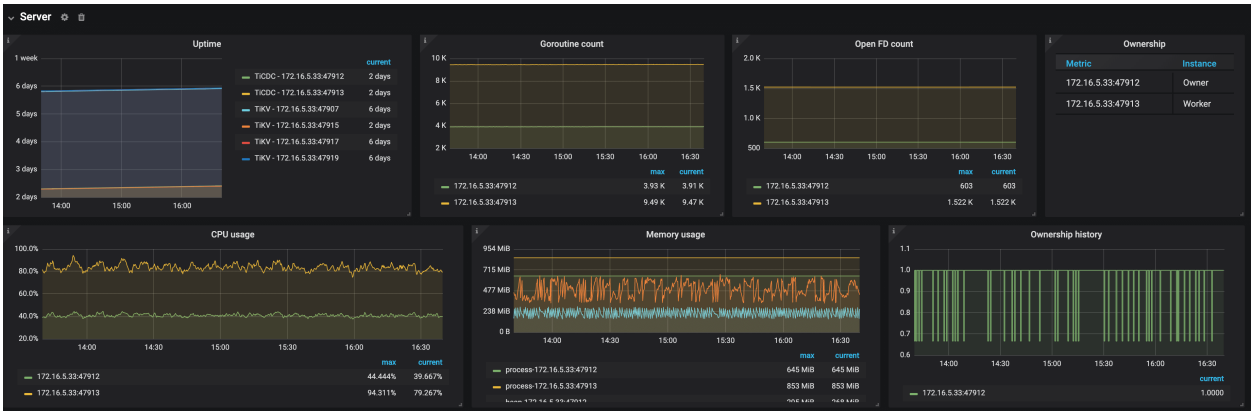


图 180: TiCDC Dashboard - Server metrics

Server 面板的各指标说明如下：

- Uptime：TiKV 节点和 TiCDC 节点已经运行的时间
- Goroutine count：TiCDC 节点 Goroutine 的个数
- Open FD count：TiCDC 节点打开的文件句柄个数
- Ownership：TiCDC 集群中节点的当前状态
- Ownership history：TiCDC 集群中 Owner 节点的历史记录
- CPU usage：TiCDC 节点使用的 CPU
- Memory usage：TiCDC 节点使用的内存

11.10.5.2 Changefeed 面板

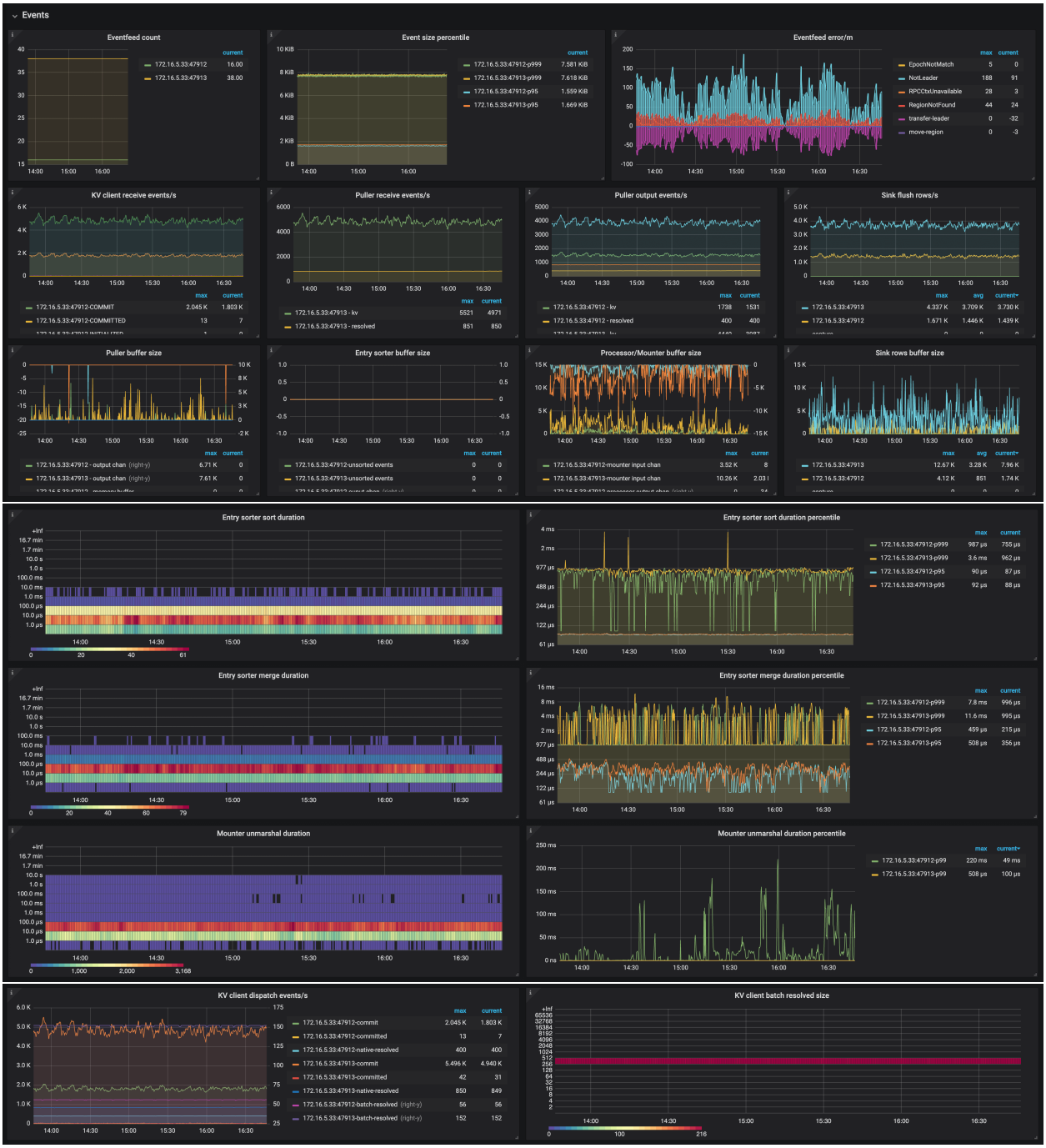
Changefeed 面板示例如下：

Changefeed 面板的各指标说明如下：

- Changefeed table count：一个同步任务中分配到各个 TiCDC 节点同步的数据表个数
- Processor resolved ts：TiCDC 节点内部状态中已同步的时间点
- Table resolved ts：同步任务中各数据表的同步进度
- Changefeed checkpoint：同步任务同步到下游的进度，正常情况下绿柱应和黄线相接
- PD etcd requests/s：TiCDC 节点每秒向 PD 读写数据的次数
- Exit error count：每分钟内导致同步中断的错误发生次数
- Changefeed checkpoint lag：同步任务上下游数据的进度差（以时间计算）
- Changefeed resolved ts lag：TiCDC 节点内部同步状态与上游的进度差（以时间计算）
- Flush sink duration：TiCDC 异步刷写数据入下游的耗时直方图
- Flush sink duration percentile：每秒钟中 95%、99% 和 99.9% 的情况下，TiCDC 异步刷写数据入下游所花费的时间
- Sink write duration：TiCDC 将一个事务的更改写到下游的耗时直方图
- Sink write duration percentile：每秒钟中 95%、99% 和 99.9% 的情况下，TiCDC 将一个事务的更改写到下游所花费的时间
- MySQL sink conflict detect duration：MySQL 写入冲突检测耗时直方图
- MySQL sink conflict detect duration percentile：每秒钟中 95%、99% 和 99.9% 的情况下，MySQL 写入冲突检测耗时
- MySQL sink worker load：TiCDC 节点中写 MySQL 线程的负载情况

11.10.5.3 Events 面板

Events 面板示例如下：



Events 面板的各指标说明如下：

- Eventfeed count：TiCDC 节点中 Eventfeed RPC 的个数
- Event size percentile：每秒钟中 95% 和 99.9% 的情况下，TiCDC 收到的来自 TiKV 的数据变更消息大小
- Eventfeed error/m：TiCDC 节点中每分钟 Eventfeed RPC 遇到的错误个数
- KV client receive events/s：TiCDC 节点中 KV client 模块每秒收到来自 TiKV 的数据变更个数
- Puller receive events/s：TiCDC 节点中 Puller 模块每秒收到来自 KV client 模块的数据变更个数
- Puller output events/s：TiCDC 节点中 Puller 模块每秒输出到 Sorter 模块的数据变更个数
- Sink flush rows/s：TiCDC 节点每秒写到下游的数据变更的个数

- Puller buffer size：TiCDC 节点中缓存在 Puller 模块中的数据变更个数
- Entry sorter buffer size：TiCDC 节点中缓存在 Sorter 模块中的数据变更个数
- Processor/Mounter buffer size：TiCDC 节点中缓存在 Processor 模块和 Mounter 模块中的数据变更个数
- Sink row buffer size：TiCDC 节点中缓存在 Sink 模块中的数据变更个数
- Entry sorter sort duration：TiCDC 节点对数据变更进行排序的耗时直方图
- Entry sorter sort duration percentile：每秒钟中 95%，99% 和 99.9% 的情况下，TiCDC 排序数据变更所花费的时间
- Entry sorter merge duration：TiCDC 节点合并排序后的数据变更的耗时直方图
- Entry sorter merge duration percentile：每秒钟中 95%，99% 和 99.9% 的情况下，TiCDC 合并排序后的数据变更所花费的时间
- Mounter unmarshal duration：TiCDC 节点解码数据变更的耗时直方图
- Mounter unmarshal duration percentile：每秒钟中 95%，99% 和 99.9% 的情况下，TiCDC 解码数据变更所花费的时间
- KV client dispatch events/s：TiCDC 节点内部 KV client 模块每秒分发数据变更的个数
- KV client batch resolved size：TiKV 批量发给 TiCDC 的 resolved ts 消息的大小

11.10.5.4 TiKV 面板

TiKV 面板示例如下：

TiKV 面板的各指标说明如下：

- CDC endpoint CPU：TiKV 节点上 CDC endpoint 线程使用的 CPU
- CDC worker CPU：TiKV 节点上 CDC worker 线程使用的 CPU
- Min resolved ts：TiKV 节点上最小的 resolved ts
- Min resolved region：TiKV 节点上最小的 resolved ts 的 Region ID
- Resolved ts lag duration percentile：TiKV 节点上最小的 resolved ts 与当前时间的差距
- Initial scan duration：TiKV 节点与 TiCDC 建立链接时增量扫的耗时直方图
- Initial scan duration percentile：每秒钟中 95%、99% 和 99.9% 的情况下，TiKV 节点增量扫的耗时
- Memory without block cache：TiKV 节点在减去 RocksDB block cache 后使用的内存
- CDC pending bytes in memory：TiKV 节点中 CDC 模块使用的内存
- Captured region count：TiKV 节点上捕获数据变更的 Region 个数

11.10.6 TiCDC Open Protocol

11.10.6.1 概述

TiCDC Open Protocol 是一种行级别的数据变更通知协议，为监控、缓存、全文索引、分析引擎、异构数据库的主从复制等提供数据源。TiCDC 遵循 TiCDC Open Protocol，向 MQ(Message Queue) 等第三方数据媒介复制 TiDB 的数据变更。

TiCDC Open Protocol 以 Event 为基本单位向下游复制数据变更事件，Event 分为三类：

- Row Changed Event：代表一行的数据变化，在行发生变更时该 Event 被发出，包含变更后该行的相关信息。
- DDL Event：代表 DDL 变更，在上游成功执行 DDL 后发出，DDL Event 会广播到每一个 MQ Partition 中。
- Resolved Event：代表一个特殊的时间点，表示在这个时间点前的收到的 Event 是完整的。

11.10.6.2 协议约束

- 在绝大多数情况下，一个版本的 Row Changed Event 只会发出一次，但是特殊情况（节点故障、网络分区等）下，同一版本的 Row Changed Event 可能会多次发送。
- 同一张表中的每一个版本第一次发出的 Row Changed Event 在 Event 流中一定是按 TS (timestamp) 顺序递增的。
- Resolved Event 会被周期性的广播到各个 MQ Partition，Resolved Event 意味着任何 TS 小于 Resolved Event TS 的 Event 已经发送给下游。
- DDL Event 将被广播到各个 MQ Partition。
- 一行数据的多个 Row Changed Event 一定会被发送到同一个 MQ Partition 中。

11.10.6.3 Message 格式定义

一个 Message 中包含一个或多个 Event，按照以下格式排列：

Key:

Offset(Byte)	0~7	8~15	16~(15+ 长度 1)	…	…
参数	协议版本号	长度 1	Event Key1	长度 N	Event KeyN

Value:

Offset(Byte)	0~7	8~(7+ 长度 1)	…	…
参数	长度 1	Event Value1	长度 N	Event ValueN

- 长度N代表第 N 个 Key/Value 的长度
- 长度及协议版本号均为大端序 int64 类型
- 当前协议版本号为 1

11.10.6.4 Event 格式定义

本部分介绍 Row Changed Event、 DDL Event 和 Resolved Event 的格式定义。

11.10.6.4.1 Row Changed Event

- Key:

```
{
    "ts":<TS>,
    "scm":<Schema Name>,
    "tbl":<Table Name>,
    "t":1
}
```

参数	类型	说明
TS	Number	造成 Row 变更的事务的 TS
Schema Name	String	Row 所在的 Schema 的名字
Table Name	String	Row 所在的 Table 的名字

- Value:

  `Insert` 事件，输出新增的行数据。

```
{
    "u":{
        <Column Name>:{
            "t":<Column Type>,
            "h":<Where Handle>,
            "f":<Flag>,
            "v":<Column Value>
        },
        <Column Name>:{
            "t":<Column Type>,
            "h":<Where Handle>,
            "f":<Flag>,
```

```
            "v":<Column Value>
        }
    }
}
```

Update 事件，输出新增的行数据（“u”）以及修改前的行数据（“p”），仅当 Old Value 特性开启时，才会输出修改前的行数据。

```
{
    "u":{
        <Column Name>:{
            "t":<Column Type>,
            "h":<Where Handle>,
            "f":<Flag>,
            "v":<Column Value>
        },
        <Column Name>:{
            "t":<Column Type>,
            "h":<Where Handle>,
            "f":<Flag>,
            "v":<Column Value>
        }
    },
    "p":{
        <Column Name>:{
            "t":<Column Type>,
            "h":<Where Handle>,
            "f":<Flag>,
            "v":<Column Value>
        },
        <Column Name>:{
            "t":<Column Type>,
            "h":<Where Handle>,
            "f":<Flag>,
            "v":<Column Value>
        }
    }
}
```

Delete 事件，输出被删除的行数据。当 Old Value 特性开启时，Delete 事件中包含被删除的行数据中的所有列；当 Old Value 特性关闭时，Delete 事件中仅包含HandleKey 列。

```
{
    "d":{
        <Column Name>:{
            "t":<Column Type>,
            "h":<Where Handle>,
```

```
            "f":<Flag>,
            "v":<Column Value>
        },
        <Column Name>:{
            "t":<Column Type>,
            "h":<Where Handle>,
            "f":<Flag>,
            "v":<Column Value>
        }
    }
}
```

参数	类型	说明
Column Name	String	列名
Column Type	Number	列类型，详见：Column 的类型码
Where Handle	Bool	表示该列是否可以作为 Where 筛选条件，当该列在表内具有唯一性时，Where Handle 为 true。
Flag	Number	列标志位，详见：列标志位
Column Value	Any	列值

11.10.6.4.2 DDL Event

- Key:

```
{
    "ts":<TS>,
    "scm":<Schema Name>,
    "tbl":<Table Name>,
    "t":2
}
```

参数	类型	说明
TS	Number	进行 DDL 变更的事务的 TS
Schema Name	String	DDL 变更的 Schema 的名字，可能为空字符串
Table Name	String	DDL 变更的 Table 的名字，可能为空字符串

- Value:

```
{
    "q":<DDL Query>,
    "t":<DDL Type>
}
```

参数	类型	说明
DDL Query	String	DDL Query SQL
DDL Type	String	DDL 类型，详见：DDL 的类型码

11.10.6.4.3 Resolved Event

- Key:

```
{
    "ts":<TS>,
    "t":3
}
```

参数	类型	说明
TS	Number	Resolved TS，任意小于该 TS 的 Event 已经发送完毕

- Value: None

11.10.6.5 Event 流的输出示例

本部分展示并描述 Event 流的输出日志。

假设在上游执行以下 SQL 语句，MQ Partition 数量为 2：

```
CREATE TABLE test.t1(id int primary key, val varchar(16));
```

如以下执行日志中的 Log 1、Log 3 所示，DDL Event 将被广播到所有 MQ Partition，Resolved Event 会被周期性地广播到各个 MQ Partition：

```
1. [partition=0] [key="{\"ts\":415508856908021766,\"scm\":\"test\",\"tbl\":\"t1\",\"t\":2}"] [
    ↪ value="{\"q\":\"CREATE TABLE test.t1(id int primary key, val varchar(16))\",\"t\":3}"]
2. [partition=0] [key="{\"ts\":415508856908021766,\"t\":3}"] [value=]
3. [partition=1] [key="{\"ts\":415508856908021766,\"scm\":\"test\",\"tbl\":\"t1\",\"t\":2}"] [
    ↪ value="{\"q\":\"CREATE TABLE test.t1(id int primary key, val varchar(16))\",\"t\":3}"]
4. [partition=1] [key="{\"ts\":415508856908021766,\"t\":3}"] [value=]
```

在上游执行以下 SQL 语句：

```
BEGIN;
INSERT INTO test.t1(id, val) VALUES (1, 'aa');
INSERT INTO test.t1(id, val) VALUES (2, 'aa');
UPDATE test.t1 SET val = 'bb' WHERE id = 2;
INSERT INTO test.t1(id, val) VALUES (3, 'cc');
COMMIT;
```

- 如以下执行日志中的 Log 5 和 Log 6 所示，同一张表内的 Row Changed Event 可能会根据主键被分派到不同的 Partition，但同一行的变更一定会分派到同一个 Partition，方便下游并发处理。
- 如 Log 6 所示，在一个事务内对同一行进行多次修改，只会发出一个 Row Changed Event。
- Log 8 是 Log 7 的重复 Event。 Row Changed Event 可能重复，但每个版本的 Event 第一次发出的次序一定是有序的。

```
5. [partition=0] [key="{\"ts\":415508878783938562,\"scm\":\"test\",\"tbl\":\"t1\",\"t\":1}"] [
    ↪ value="{\"u\":{\"id\":{\"t\":3,\"h\":true,\"v\":1},\"val\":{\"t\":15,\"v\":\"YWE=\"}}}"]
6. [partition=1] [key="{\"ts\":415508878783938562,\"scm\":\"test\",\"tbl\":\"t1\",\"t\":1}"] [
    ↪ value="{\"u\":{\"id\":{\"t\":3,\"h\":true,\"v\":2},\"val\":{\"t\":15,\"v\":\"YmI=\"}}}"]
7. [partition=0] [key="{\"ts\":415508878783938562,\"scm\":\"test\",\"tbl\":\"t1\",\"t\":1}"] [
    ↪ value="{\"u\":{\"id\":{\"t\":3,\"h\":true,\"v\":3},\"val\":{\"t\":15,\"v\":\"Y2M=\"}}}"]
8. [partition=0] [key="{\"ts\":415508878783938562,\"scm\":\"test\",\"tbl\":\"t1\",\"t\":1}"] [
    ↪ value="{\"u\":{\"id\":{\"t\":3,\"h\":true,\"v\":3},\"val\":{\"t\":15,\"v\":\"Y2M=\"}}}"]
```

在上游执行以下 SQL 语句：

```
BEGIN;
DELETE FROM test.t1 WHERE id = 1;
UPDATE test.t1 SET val = 'dd' WHERE id = 3;
UPDATE test.t1 SET id = 4, val = 'ee' WHERE id = 2;
COMMIT;
```

- Log 9 是 Delete 类型的 Row Changed Event，这种类型的 Event 只包含主键列或唯一索引列。
- Log 13 和 Log 14 是 Resolved Event。Resolved Event 意味着在这个 Partition 中，任意小于 Resolved TS 的 Event（包括 Row Changed Event 和 DDL Event）已经发送完毕。

```
9. [partition=0] [key="{\"ts\":415508881418485761,\"scm\":\"test\",\"tbl\":\"t1\",\"t\":1}"] [
    ↪ value="{\"d\":{\"id\":{\"t\":3,\"h\":true,\"v\":1}}}"]
10. [partition=1] [key="{\"ts\":415508881418485761,\"scm\":\"test\",\"tbl\":\"t1\",\"t\":1}"] [
    ↪ value="{\"d\":{\"id\":{\"t\":3,\"h\":true,\"v\":2}}}"]
11. [partition=0] [key="{\"ts\":415508881418485761,\"scm\":\"test\",\"tbl\":\"t1\",\"t\":1}"] [
    ↪ value="{\"u\":{\"id\":{\"t\":3,\"h\":true,\"v\":3},\"val\":{\"t\":15,\"v\":\"ZGQ=\"}}}"]
12. [partition=0] [key="{\"ts\":415508881418485761,\"scm\":\"test\",\"tbl\":\"t1\",\"t\":1}"] [
    ↪ value="{\"u\":{\"id\":{\"t\":3,\"h\":true,\"v\":4},\"val\":{\"t\":15,\"v\":\"ZWU=\"}}}"]
13. [partition=0] [key="{\"ts\":415508881038376963,\"t\":3}"] [value=]
14. [partition=1] [key="{\"ts\":415508881038376963,\"t\":3}"] [value=]
```

11.10.6.6 消费端协议解析

目前 TiCDC 没有提供 Open Protocol 协议解析的标准实现，但是提供了 Golang 版本和 Java 版本的解析 demo。用户可以参考本文档提供的数据格式和以下 demo 实现消费端协议解析。

- Golang 例子
- Java 例子

11.10.6.7 Column 的类型码

Column 的类型码用于标识 Row Changed Event 中列的数据类型。

类型	Code	输出示例	说明
TINYINT/BOOL	1	{“t”:1,“v”:1}	
SMALLINT	2	{“t”:2,“v”:1}	
INT	3	{“t”:3,“v”:123}	
FLOAT	4	{“t”:4,“v”:153.123}	
DOUBLE	5	{“t”:5,“v”:153.123}	
NULL	6	{“t”:6,“v”:null}	
TIMESTAMP	7	{“t”:7,“v”:“1973-12-30 15:30:00”}	
BIGINT	8	{“t”:8,“v”:123}	
MEDIUMINT	9	{“t”:9,“v”:123}	
DATE	10/14	{“t”:10,“v”:“2000-01-01”}	
TIME	11	{“t”:11,“v”:“23:59:59”}	
DATETIME	12	{“t”:12,“v”:“2015-12-20 23:58:58”}	
YEAR	13	{“t”:13,“v”:1970}	

类型	Code	输出示例	说明
VARCHAR/VARBINARY	15/253	{ “t” :15, “v” : “测试” } / { “t” :15, “v” : “\\x89PNG\\r\\n\\x1a\\n” }	编码为 UTF-8；当上游类型为 VARBINARY 时，将对不可见的字符转义
BIT	16	{ “t” :16, “v” :81}	
JSON	245	{ “t” :245, “v” : “{\” key1\ “: \” value1\ “}” }	
DECIMAL	246	{ “t” :246, “v” : “129012.1230000” }	
ENUM	247	{ “t” :247, “v” :1}	
SET	248	{ “t” :248, “v” :3}	
TINYTEXT/TINYBLOB	249	{ “t” :249, “v” : “5rWL6K+VdGV4dA==” }	编码为 Base64
MEDIUMTEXT/MEDIUMBLOB	250	{ “t” :250, “v” : “5rWL6K+VdGV4dA==” }	编码为 Base64

类型	Code	输出示例	说明
LONGTEXT/LONGBLOB	251	{ “t” :251, “v”:”value” }	编码为 Base64
TEXT/BLOB	252	{ “t” :252, “v”:”value” }	编码为 Base64
CHAR/BINARY	254	{ “t” :254, “v”:”测试” } / { “t” :254, “v”:”\\x89PNG\\r\\n\\x1a\\n” }	编码为 UTF-8；当上游类型为 BINARY 时，将对不可见的字符转义
GEOMETRY	255		尚不支持

11.10.6.8 DDL 的类型码

DDL 的类型码用于标识 DDL Event 中的 DDL 语句的类型。

类型	Code
Create Schema	1
Drop Schema	2
Create Table	3
Drop Table	4
Add Column	5
Drop Column	6
Add Index	7
Drop Index	8
Add Foreign Key	9
Drop Foreign Key	10
Truncate Table	11
Modify Column	12
Rebase Auto ID	13
Rename Table	14
Set Default Value	15
Shard RowID	16
Modify Table Comment	17
Rename Index	18
Add Table Partition	19
Drop Table Partition	20
Create View	21
Modify Table Charset And Collate	22
Truncate Table Partition	23
Drop View	24
Recover Table	25
Modify Schema Charset And Collate	26
Lock Table	27
Unlock Table	28
Repair Table	29
Set TiFlash Replica	30
Update TiFlash Replica Status	31
Add Primary Key	32
Drop Primary Key	33
Create Sequence	34
Alter Sequence	35
Drop Sequence	36

11.10.6.9 列标志位

列标志位以 Bit flags 形式标记列的相关属性。

位移	值	名称	说明
1	0x01	BinaryFlag	该列是否为二进制编码列
2	0x02	HandleKeyFlag	该列是否为 Handle 列
3	0x04	GeneratedColumnFlag	该列是否为生成列
4	0x08	PrimaryKeyFlag	该列是否为主键列
5	0x10	UniqueKeyFlag	该列是否为唯一索引列
6	0x20	MultipleKeyFlag	该列是否为组合索引列
7	0x40	NullableFlag	该列是否为可空列
8	0x80	UnsignedFlag	该列是否为无符号列

示例：

若某列 Flag 值为 85，则代表这一列为可空列、唯一索引列、生成列、二进制编码列。

```
85 == 0b_101_0101
   == NullableFlag | UniqueKeyFlag | GeneratedColumnFlag | BinaryFlag
```

若某列 Flag 值为 46，则代表这一列为组合索引列、主键列、生成列、Handle 列。

```
46 == 0b_010_1110
   == MultipleKeyFlag | PrimaryKeyFlag | GeneratedColumnFlag | HandleKeyFlag
```

> 注意：
>
> - BinaryFlag 仅在列为 BLOB/TEXT（包括 TINYBLOB/TINYTEXT、BINARY/CHAR 等）类型时才有意义。当上游列为 BLOB 类型时，BinaryFlag 置 1；当上游列为 TEXT 类型时，BinaryFlag 置 0。
> - 若要同步上游的一张表，TiCDC 会选择一个有效索引作为 Handle Index。Handle Index 包含的列的 HandleKeyFlag 置 1。

11.10.7 TiDB 集成 Confluent Platform 快速上手指南

本文档介绍如何使用TiCDC 将 TiDB 集成到 Confluent Platform。

> 警告：
>
> 当前该功能为实验特性，请勿在生产环境中使用。

Confluent Platform 是一个以 Apache Kafka 为核心的流数据处理平台，可以借助官方或第三方的 sink connector 将数据源连接到关系型或非关系型数据库。

你可以使用 TiCDC 组件和 Avro 协议来集成 TiDB 和 Confluent Platform。TiCDC 能将数据更改以 Confluent Platform 能识别的格式流式传输到 Kafka。下文详细介绍了集成的操作步骤。

11.10.7.1 环境准备

> **注意：**
>
> 本教程使用 JDBC sink connector 将 TiDB 的数据同步到下游的关系型数据库中。为了简化操作，此处以 SQLite 为例。

- 确保 Zookeeper、Kafka 和 Schema Registry 已正确安装。推荐参照 Confluent Platform 快速入门指南部署本地测试环境。
- 通过以下命令确保 JDBC sink connector 已安装。返回结果中预期包含 `jdbc-sink`。

```
confluent local services connect connector list
```

11.10.7.2 集成步骤

1. 将下方的配置样例保存为 `jdbc-sink-connector.json` 文件：

```
{
  "name": "jdbc-sink-connector",
  "config": {
    "connector.class": "io.confluent.connect.jdbc.JdbcSinkConnector",
    "tasks.max": "1",
    "topics": "testdb_test",
    "connection.url": "jdbc:sqlite:/tmp/test.db",
    "connection.ds.pool.size": 5,
    "table.name.format": "test",
    "auto.create": true,
    "auto.evolve": true
  }
}
```

2. 运行下方命令新建一个 JDBC sink connector 实例（假设 Kafka 监听的 IP 地址与端口是 `127.0.0.1:8083`）：

```
curl -X POST -H "Content-Type: application/json" -d jdbc-sink-connector.json http
    ↪ ://127.0.0.1:8083/connectors
```

3. 通过以下任一方式部署 TiCDC。如果已经部署了 TiCDC，可以跳过这一步。

    - 使用 TiUP 部署包含 TiCDC 组件的全新 TiDB 集群
    - 使用 TiUP 在原有 TiDB 集群上新增 TiCDC 组件
    - 使用 binary 在原有 TiDB 集群上新增 TiCDC 组件（不推荐）

    在继续接下来的操作之前，请先确保 TiDB 和 TiCDC 集群处于健康状态。

4. 运行下面的 cdc cli 命令，新建一个同步任务 changefeed：

```
./cdc cli changefeed create --pd="http://127.0.0.1:2379" --sink-uri="kafka://127.0.0.1:9092/
    ↪ testdb_test?protocol=avro" --opts "registry=http://127.0.0.1:8081"
```

> 注意：
> 请确保 PD、 Kafka 和 Schema Registry 在各自的默认端口上运行。

11.10.7.3 数据同步测试

TiDB 与 Confluent Platform 成功集成后，你可以按照以下步骤来测试数据同步功能。

1. 在 TiDB 集群中新建 testdb 数据库：

```
CREATE DATABASE IF NOT EXISTS testdb;
```

在 testdb 数据库中创建 test 数据表：

```
USE testdb;
CREATE TABLE test (
    id INT PRIMARY KEY,
    v TEXT
);
```

> 注意：
> 如果需要修改数据库名或数据表名，要相应地修改 jdbc-sink-connector.json 文件中 topics 的参数值。

2. 向 test 数据表中插入数据：

```
INSERT INTO test (id, v) values (1, 'a');
INSERT INTO test (id, v) values (2, 'b');
INSERT INTO test (id, v) values (3, 'c');
INSERT INTO test (id, v) values (4, 'd');
```

3. 等待数据被同步到下游数据库中，并检查下游的数据：

```
sqlite3 test.db
sqlite> SELECT * from test;
```

11.10.8 TiCDC 术语表

本术语表提供 TiCDC 相关的术语和定义，这些术语会出现在 TiCDC 的日志、监控指标、配置和文档中。

11.10.8.1 B

11.10.8.1.1 变更数据

从上游 TiDB 集群写入 TiCDC 的数据，包括 DML 操作引发的数据变更和 DDL 操作引发的表结构变更。

11.10.8.2 C

11.10.8.2.1 Capture

单个 TiCDC 实例。多个 Capture 组成一个 TiCDC 集群，Capture 上运行集群中的同步任务。

11.10.8.2.2 Changefeed

TiCDC 中的单个同步任务。同步任务将一个 TiDB 集群中数张表的变更数据输出到一个指定的下游中。

11.10.8.3 O

11.10.8.3.1 Owner

一个特殊角色的Capture，负责管理 TiCDC 集群和调度 TiCDC 集群中的同步任务。该角色由 Capture 选举产生，在任意时刻最多只存在一个。

11.10.8.4 P

11.10.8.4.1 Processor

TiCDC 同步任务会在 TiCDC 实例上分配数据表，Processor 指这些数据表的同步处理单元。处理任务包括变更数据的拉取、排序、还原和分发。

11.11 使用 Dumpling 导出数据

使用数据导出工具 Dumpling，你可以把存储在 TiDB 或 MySQL 中的数据导出为 SQL 或 CSV 格式，用于逻辑全量备份。Dumpling 也支持将数据导出到 Amazon S3 中。

下图展示了使用 Dumpling 导出数据的场景。

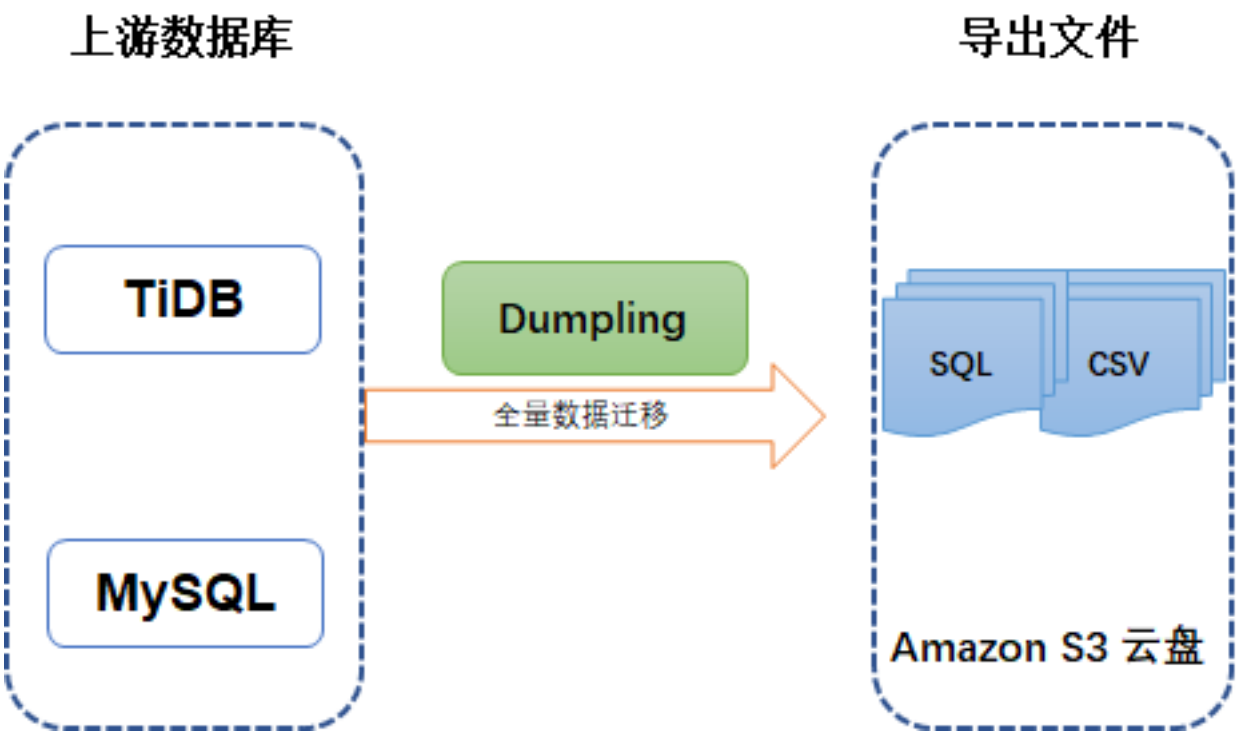


图 181: Dumpling 导出数据

要快速了解 Dumpling 的基本功能，建议先观看下面的培训视频（时长 28 分钟）。注意本视频只作为功能介绍、学习参考，具体操作步骤和最新功能，请以文档内容为准。

你可以通过下列任意方式获取 Dumpling：

- TiUP 执行 `tiup install dumpling` 命令。获取后，使用 `tiup dumpling ...` 命令运行 Dumpling。
- 下载包含 Dumpling 的tidb-toolkit 安装包。

更多详情，可以使用 –help 选项查看，或参考Dumpling 主要选项表。

使用 Dumpling 时，需要在已经启动的集群上执行导出命令。

TiDB 还提供了其他工具，你可以根据需要选择使用：

- 如果需要直接备份 SST 文件（键值对），或者对延迟不敏感的增量备份，请使用备份工具BR。
- 如果需要实时的增量备份，请使用TiCDC。
- 所有的导出数据都可以用TiDB Lightning 导回到 TiDB。

> 注意：
>
> PingCAP 之前维护的 Mydumper 工具 fork 自 mydumper project，针对 TiDB 的特性进行了优化。关于 Mydumper 的更多信息，请参考 v4.0 版 Mydumper 使用文档。从 v7.5.0 开始，Mydumper 废弃，其绝大部分功能已经被Dumpling 取代，强烈建议切换到 Dumpling。

相比 Mydumper，Dumpling 做了如下改进：

- 支持导出多种数据形式，包括 SQL/CSV。
- 支持全新的 table-filter，筛选数据更加方便。
- 支持导出到 Amazon S3 云盘。
- 针对 TiDB 进行了更多优化：
  - 支持配置 TiDB 单条 SQL 内存限制。
  - 针对 TiDB v4.0.0 及更新版本支持自动调整 TiDB GC 时间。
  - 使用 TiDB 的隐藏列 `_tidb_rowid` 优化了单表内数据的并发导出性能。
  - 对于 TiDB 可以设置tidb_snapshot 的值指定备份数据的时间点，从而保证备份的一致性，而不是通过 `FLUSH TABLES WITH READ LOCK` 来保证备份一致性。

11.11.1 从 TiDB/MySQL 导出数据

11.11.1.1 需要的权限

- SELECT
- RELOAD
- LOCK TABLES
- REPLICATION CLIENT

11.11.1.2 导出为 SQL 文件

本文假设在 `127.0.0.1:4000` 有一个 TiDB 实例，并且这个 TiDB 实例中有无密码的 root 用户。

Dumpling 默认导出数据格式为 SQL 文件。也可以通过设置 `--filetype sql` 导出数据到 SQL 文件：

```
dumpling \
  -u root \
  -P 4000 \
  -h 127.0.0.1 \
  --filetype sql \
  -t 8 \
  -o /tmp/test \
  -r 200000 \
  -F 256MiB
```

以上命令中：

- `-h`、`-P`、`-u` 分别代表地址、端口、用户。如果需要密码验证，可以使用 `-p $YOUR_SECRET_PASSWORD` 将密码传给 Dumpling。
- `-o` 用于选择存储导出文件的目录，支持本地文件路径或外部存储 URL 格式。
- `-t` 用于指定导出的线程数。增加线程数会增加 Dumpling 并发度提高导出速度，但也会加大数据库内存消耗，因此不宜设置过大。
- `-r` 用于指定单个文件的最大行数，指定该参数后 Dumpling 会开启表内并发加速导出，同时减少内存使用。

- `-F` 选项用于指定单个文件的最大大小，单位为 MiB，可接受类似 5GiB 或 8KB 的输入。如果你想使用 TiDB Lightning 将该文件加载到 TiDB 实例中，建议将 `-F` 选项的值保持在 256 MiB 或以下。

> **注意：**
>
> 如果导出的单表大小超过 10 GB，强烈建议使用 `-r` 和 `-F` 参数。

11.11.1.3 导出为 CSV 文件

假如导出数据的格式是 CSV（使用 `--filetype csv` 即可导出 CSV 文件），还可以使用 `--sql <SQL>` 导出指定 SQL 选择出来的记录，例如，导出 `test.sbtest1` 中所有 `id < 100` 的记录：

```
./dumpling \
  -u root \
  -P 4000 \
  -h 127.0.0.1 \
  -o /tmp/test \
  --filetype csv \
  --sql 'select * from `test`.`sbtest1` where id < 100'
```

> **注意：**
>
> - `--sql` 选项仅仅可用于导出 CSV 的场景。
> - 该命令将在要导出的所有表上执行 `select * from <table-name> where id < 100` 语句。如果部分表没有指定的字段，那么导出会失败。
> - Dumpling 导出不区分字符串与关键字。如果导入的数据是 Boolean 类型的 `true` 和 `false`，导出时会被转换为 1 和 0。

11.11.1.4 输出文件格式

- `metadata`：此文件包含导出的起始时间，以及 master binary log 的位置。

```
cat metadata
```

```
Started dump at: 2020-11-10 10:40:19
SHOW MASTER STATUS:
        Log: tidb-binlog
        Pos: 420747102018863124

Finished dump at: 2020-11-10 10:40:20
```

- `{schema}-schema-create.sql`：创建 schema 的 SQL 文件。

```
cat test-schema-create.sql
```

```
CREATE DATABASE `test` /*!40100 DEFAULT CHARACTER SET utf8mb4 */;
```

- `{schema}.{table}-schema.sql`：创建 table 的 SQL 文件

```
cat test.t1-schema.sql
```

```
CREATE TABLE `t1` (
  `id` int(11) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin;
```

- `{schema}.{table}.{0001}.{sql|csv}`：数据源文件

```
cat test.t1.0.sql
```

```
/*!40101 SET NAMES binary*/;
INSERT INTO `t1` VALUES
(1);
```

- `*-schema-view.sql`、`*-schema-trigger.sql`、`*-schema-post.sql`：其他导出文件

11.11.1.5 导出到 Amazon S3 云盘

Dumpling 在 v4.0.8 及更新版本支持导出到 Amazon S3 云盘。如果需要将数据备份到 Amazon S3 后端存储，那么需要在 `-o` 参数中指定 Amazon S3 的存储路径。

可以参照 AWS 官方文档 - 如何创建 S3 存储桶在指定的 `Region` 区域中创建一个 S3 桶 `Bucket`。如有需要，还可以参照 AWS 官方文档 - 创建文件夹在 Bucket 中创建一个文件夹 `Folder`。

将有权限访问该 Amazon S3 后端存储的账号的 `SecretKey` 和 `AccessKey` 作为环境变量传入 Dumpling 节点。

```
export AWS_ACCESS_KEY_ID=${AccessKey}
export AWS_SECRET_ACCESS_KEY=${SecretKey}
```

Dumpling 同时还支持从 `~/.aws/credentials` 读取凭证文件。更多 Dumpling 存储配置可以参考外部存储。

在进行 Dumpling 备份时，显式指定参数 `--s3.region`，即表示 Amazon S3 存储所在的区域，例如 `ap-northeast-1`。

```
./dumpling \
  -u root \
  -P 4000 \
  -h 127.0.0.1 \
  -r 200000 \
  -o "s3://${Bucket}/${Folder}" \
  --s3.region "${region}"
```

11.11.1.6 筛选导出的数据

11.11.1.6.1 使用 --where 选项筛选数据

默认情况下，Dumpling 会导出排除系统数据库（包括 mysql、sys、INFORMATION_SCHEMA、PERFORMANCE_SCHEMA、METRICS_SCHEMA 和 INSPECTION_SCHEMA）外所有其他数据库。你可以使用 --where <SQL where expression> 来指定要导出的记录。

```
./dumpling \
  -u root \
  -P 4000 \
  -h 127.0.0.1 \
  -o /tmp/test \
  --where "id < 100"
```

上述命令将会导出各个表的 id < 100 的数据。注意 --where 参数无法与 --sql 一起使用。

11.11.1.6.2 使用 --filter 选项筛选数据

Dumpling 可以通过 --filter 指定 table-filter 来筛选特定的库表。table-filter 的语法与 .gitignore 相似，详细语法参考表库过滤。

```
./dumpling \
  -u root \
  -P 4000 \
  -h 127.0.0.1 \
  -o /tmp/test \
  -r 200000 \
  --filter "employees.*" \
  --filter "*.WorkOrder"
```

上述命令将会导出 employees 数据库的所有表，以及所有数据库中的 WorkOrder 表。

11.11.1.6.3 使用 -B 或 -T 选项筛选数据

Dumpling 也可以通过 -B 或 -T 选项导出特定的数据库/数据表。

> 注意：
>
> - --filter 选项与 -T 选项不可同时使用。
> - -T 选项只能接受完整的 库名.表名 形式，不支持只指定表名。例：Dumpling 无法识别 -T WorkOrder。

例如通过指定：

- `-B employees` 导出 `employees` 数据库
- `-T employees.WorkOrder` 导出 `employees.WorkOrder` 数据表

11.11.1.7 通过并发提高 Dumpling 的导出效率

默认情况下，导出的文件会存储到 `./export-<current local time>` 目录下。常用选项如下：

- `-t` 用于指定导出的线程数。增加线程数会增加 Dumpling 并发度提高导出速度，但也会加大数据库内存消耗，因此不宜设置过大。
- `-r` 选项用于指定单个文件的最大记录数，或者说，数据库中的行数。开启后 Dumpling 会开启表内并发，提高导出大表的速度。当上游为 TiDB 且版本为 v3.0 或更新版本时，该参数大于 0 表示使用 TiDB region 信息划分表内并发，具体取值将不再生效。
- `--compress gzip` 选项可以用于压缩导出的数据。压缩可以显著降低导出数据的大小，同时如果存储的写入 I/O 带宽不足，可以使用该选项来加速导出。但该选项也有副作用，由于该选项会对每个文件单独压缩，因此会增加 CPU 消耗。

利用以上选项可以提高 Dumpling 的导出速度。

11.11.1.8 调整 Dumpling 的数据一致性选项

> **注意：**
>
> 数据一致性选项的默认值为 `auto`。在大多数场景下，你不需要调整该选项。

Dumpling 通过 `--consistency <consistency level>` 标志控制导出数据“一致性保证”的方式。对于 TiDB 来说，默认情况下，会通过获取某个时间戳的快照来保证一致性（即 `--consistency snapshot`）。在使用 snapshot 来保证一致性的时候，可以使用 `--snapshot` 选项指定要备份的时间戳。还可以使用以下的一致性级别：

- `flush`：使用 `FLUSH TABLES WITH READ LOCK` 短暂地中断备份库的 DML 和 DDL 操作、保证备份连接的全局一致性和记录 POS 信息。所有的备份连接启动事务后释放该锁。推荐在业务低峰或者 MySQL 备份库上进行全量备份。
- `snapshot`：获取指定时间戳的一致性快照并导出。
- `lock`：为待导出的所有表上读锁。
- `none`：不做任何一致性保证。
- `auto`：对 MySQL 使用 `flush`，对 TiDB 使用 `snapshot`。

操作完成之后，你可以在 `/tmp/test` 查看导出的文件：

```
$ ls -lh /tmp/test | awk '{print $5 "\t" $9}'

140B  metadata
66B   test-schema-create.sql
300B  test.sbtest1-schema.sql
```

```
190K  test.sbtest1.0.sql
300B  test.sbtest2-schema.sql
190K  test.sbtest2.0.sql
300B  test.sbtest3-schema.sql
190K  test.sbtest3.0.sql
```

11.11.1.9 导出 TiDB 的历史数据快照

Dumping 可以通过 `--snapshot` 指定导出某个tidb_snapshot 的数据。

`--snapshot` 选项可设为 TSO（`SHOW MASTER STATUS` 输出的 `Position` 字段）或有效的 `datetime` 时间（`YYYY-MM-DD hh:mm:ss` 形式），例如：

```
./dumpling --snapshot 417773951312461825
./dumpling --snapshot "2020-07-02 17:12:45"
```

即可导出 TSO 为 417773951312461825 或 2020-07-02 17:12:45 时的 TiDB 历史数据快照。

11.11.1.10 控制导出 TiDB 大表时的内存使用

Dumping 导出 TiDB 较大单表时，可能会因为导出数据过大导致 TiDB 内存溢出 (OOM)，从而使连接中断导出失败。可以通过以下参数减少 TiDB 的内存使用。

- 设置 `-r` 参数，可以划分导出数据区块减少 TiDB 扫描数据的内存开销，同时也可开启表内并发提高导出效率。
- 调小 `--tidb-mem-quota-query` 参数到 8589934592 (8GB) 或更小。可控制 TiDB 单条查询语句的内存使用。
- 调整 `--params "tidb_distsql_scan_concurrency=5"` 参数，即设置导出时的 session 变量tidb_distsql_scan_concurrency 从而减少 TiDB scan 操作的并发度。

11.11.1.11 导出大规模数据时的 TiDB GC 设置

如果导出的 TiDB 版本为 v4.0.0 或更新版本，并且 Dumping 可以访问 TiDB 集群的 PD 地址，Dumping 会自动配置延长 GC 时间且不会对原集群造成影响。

其他情况下，假如导出的数据量非常大，可以提前调长 GC 时间，以避免因为导出过程中发生 GC 导致导出失败：

```
update mysql.tidb set VARIABLE_VALUE = '720h' where VARIABLE_NAME = 'tikv_gc_life_time';
```

操作结束之后，再恢复 GC 时间为默认值 `10m`：

```
update mysql.tidb set VARIABLE_VALUE = '10m' where VARIABLE_NAME = 'tikv_gc_life_time';
```

11.11.2 Dumpling 主要选项表

主要选项	用途	默认值
-V 或 –version	输出 Dumpling 版本并直接退出	
-B 或 –database	导出指定数据库	
-T 或 –tables-list	导出指定数据表	
-f 或 –filter	导出能匹配模式的表，语法可参考table-filter	[*.*,!/^(mysql|sys|INFORMATION_SCHEMA|PERFORMANCE_SCHEMA|METRICS_SCHEMA|INSPECTION_SCHEMA)$/.*] (导出除系统库外的所有库表)

主要选项	用途	默认值
–case-sensitive	table-filter 是否大小写敏感	false，大小写不敏感
-h 或 –host	连接的数据库主机的地址	“127.0.0.1”
-t 或 –threads	备份并发线程数	4
-r 或 –rows	将 table 划分成 row 行数据，一般针对大表操作并发生成多个文件。	
-L 或 –logfile	日志输出地址，为空时会输出到控制台	“”

主要选项	用途	默认值
–loglevel	日志级别 {debug,info,warn,error,dpanic,panic,fatal}	“info”
–logfmt	日志输出格式 {text,json}	“text”
-d 或 –no-data	不导出数据，适用于只导出 schema 场景	
–no-header	导出 csv 格式的 table 数据，不生成 header	
-W 或 –no-views	不导出 view	true
-m 或 –no-schemas	不导出 schema，只导出数据	
-s 或 –statement-size	控制 INSERT SQL 语句的大小，单位 bytes	

主要选项	用途	默认值
-F 或 –filesize	将 table 数据划分出来的文件大小，需指明单位 (如 128B, 64 ↪ KiB ↪ , 32 ↪ MiB ↪ , 1.5 ↪ GiB ↪)	
–filetype	导出文件类型 (csv/sql)	“sql”
-o 或 –output	导出本地文件路径或外部存储 URL	“./export-${time}”

主要选项	用途	默认值
-S 或 –sql	根据指定的 sql 导出数据，该选项不支持并发导出	

主要选项	用途	默认值
–consistency	flush: dump 前用 FTWRL snap-shot: 通过 TSO 来指定 dump 某个快照时间点的 TiDB 数据 lock: 对需要 dump 的所有表执行 lock ↪ ↪ tables ↪ ↪ read ↪ 命令 none: 不加锁 dump，无法保证一致性 auto: 对 MySQL 使用 –consistency flush；对 TiDB	“auto”

主要选项	用途	默认值
–snapshot	snapshot tso，只在 consistency=snapshot 下生效	
–where	对备份的数据表通过 where 条件指定范围	
-p 或 –password	连接的数据库主机的密码	
-P 或 –port	连接的数据库主机的端口	4000
-u 或 –user	连接的数据库主机的用户名	“root”
–dump-empty-database	导出空数据库的建库语句	true

主要选项	用途	默认值
–ca	用于 TLS 连接的 certifi-cate au-thor-ity 文件的地址	
–cert	用于 TLS 连接的 client certifi-cate 文件的地址	
–key	用于 TLS 连接的 client pri-vate key 文件的地址	
–csv-delimiter	csv 文件中字符类型变量的定界符	’ “’

主要选项	用途	默认值
–csv-separator	csv 文件中各值的分隔符	‘,’
–csv-null-value	csv 文件空值的表示	“\N”
–escape-backslash	使用反斜杠 (\) 来转义导出文件中的特殊字符	true

主要选项	用途	默认值
–output-filename-template	以 golang tem-plate 格式表示的数据文件名格式支持 {{. ↪ DB ↪ }} ↪ 、 {{. ↪ Table ↪ }} ↪ 、 {{. ↪ Index ↪ }} ↪ 三个参数分别表示数据文件的库名、表名、分块 ID	‘{{.DB}}.{{.Table}}.{{.Index}}’

主要选项	用途	默认值
–status-addr	Dumpling 的服务地址，包含了 Prometheus 拉取 metrics 信息及 pprof 调试的地址	“:8281”

主要选项	用途	默认值
–tidb-mem-quota-query	单条 dumpling 命令导出 SQL 语句的内存限制，单位为 byte。对于 v4.0.10 或以上版本，若不设置该参数，默认使用 TiDB 中的 mem-quota-query 配置项值作为内存限制值。对于 v4.0.10 以下版本，该参数值默认为 32 GB	34359738368

主要选项	用途	默认值
–params	为需导出的数据库连接指定 session 变量，可接受的格式: “character_set_client=latin1,character_set_connection=latin1”	

11.12 sync-diff-inspector

11.12.1 sync-diff-inspector 用户文档

sync-diff-inspector 是一个用于校验 MySQL / TiDB 中两份数据是否一致的工具。该工具提供了修复数据的功能 (适用于修复少量不一致的数据)。

主要功能：

- 对比表结构和数据
- 如果数据不一致，则生成用于修复数据的 SQL 语句
- 支持不同库名或表名的数据校验
- 支持分库分表场景下的数据校验
- 支持TiDB 主从集群的数据校验

GitHub 地址：sync-diff-inspector

下载地址：tidb-community-toolkit-v4.0.16-linux-amd64

11.12.1.1 sync-diff-inspector 的使用

11.12.1.1.1 使用限制

- 对于 MySQL 和 TiDB 之间的数据同步不支持在线校验，需要保证上下游校验的表中没有数据写入，或者保证某个范围内的数据不再变更，通过配置 range 来校验这个范围内的数据。

- 不支持 JSON、 BIT、 BINARY、 BLOB 等类型的数据，在校验时需要设置 `ignore-columns` 忽略检查这些类型的数据。
- FLOAT、 DOUBLE 等浮点数类型在 TiDB 和 MySQL 中的实现方式不同，在计算 checksum 时可能存在差异，如果发现因为这些类型的数据导致的数据校验不一致，需要设置 `ignore-columns` 忽略这些列的检查。
- 支持对不包含主键或者唯一索引的表进行校验，但是如果数据不一致，生成的用于修复的 SQL 可能无法正确修复数据。

11.12.1.1.2 数据库权限

sync-diff-inspector 需要获取表结构信息、查询数据、建 checkpoint 库保存断点信息，需要的数据库权限如下：

- 上游数据库
  - SELECT（查数据进行对比）
  - SHOW_DATABASES (查看库名)
  - RELOAD (查看表结构)
- 下游数据库
  - SELECT （查数据进行对比）
  - CREATE （创建 checkpoint 库和表）
  - DELETE （删除 checkpoint 表中的信息）
  - INSERT （写入 checkpoint 表）
  - UPDATE（修改 checkpoint 表）
  - SHOW_DATABASES (查看库名)
  - RELOAD (查看表结构)

11.12.1.1.3 配置文件说明

sync-diff-inspector 的配置总共分为三个部分：

- Global config: 通用配置，包括日志级别、划分 chunk 的大小、校验的线程数量等。
- Tables config: 配置校验哪些表，如果有的表在上下游有一定的映射关系或者有一些特殊要求，则需要对指定的表进行配置。
- Databases config: 配置上下游数据库实例。

下面是一个完整配置文件的说明：

```
### Diff Configuration.

########################## Global config #########################

### 日志级别，可以设置为 info、debug
log-level = "info"
```

```
### sync-diff-inspector 根据主键 / 唯一键 / 索引将数据划分为多个 chunk，
### 对每一个 chunk 的数据进行对比。使用 chunk-size 设置 chunk 的大小
chunk-size = 1000

### 检查数据的线程数量
check-thread-count = 4

### 抽样检查的比例，如果设置为 100 则检查全部数据
sample-percent = 100

### 通过计算 chunk 的 checksum 来对比数据，如果不开启则逐行对比数据
use-checksum = true

### 如果设置为 true 则只会通过计算 checksum 来校验数据，如果上下游的 checksum
    ↪ 不一致也不会查出数据再进行校验
only-use-checksum = false

### 是否使用上次校验的 checkpoint，如果开启，则只校验上次未校验以及校验失败的 chunk
use-checkpoint = true

### 不对比数据
ignore-data-check = false

### 不对比表结构
ignore-struct-check = false

### 保存用于修复数据的 sql 的文件名称
fix-sql-file = "fix.sql"

########################## Tables config ##########################

### 如果需要对比大量的不同库名或者表名的表的数据，或者用于校验上游多个分表与下游总表的数据，
    ↪ 可以通过 table-rule 来设置映射关系
### 可以只配置 schema 或者 table 的映射关系，也可以都配置
#[[table-rules]]
    # schema-pattern 和 table-pattern 支持通配符 *?
    #schema-pattern = "test_*"
    #table-pattern = "t_*"
    #target-schema = "test"
    #target-table = "t"

### 配置需要对比的*目标数据库*中的表
[[check-tables]]
    # 目标库中数据库的名称
```

```
    schema = "test"

    # 需要检查的表
    tables = ["test1", "test2", "test3"]

    # 支持使用正则表达式配置检查的表，需要以“~”开始，
    # 下面的配置会检查所有表名以“test”为前缀的表
    # tables = ["~^test.*"]
    # 下面的配置会检查配置库中所有的表
    # tables = ["~^"]

### 对部分表进行特殊的配置，配置的表必须包含在 check-tables 中
[[table-config]]
    # 目标库中数据库的名称
    schema = "test"

    # 表名
    table = "test3"

    # 指定用于划分 chunk 的列，如果不配置该项，sync-diff-inspector 会选取一个合适的列（
        ↪ 主键 / 唯一键 / 索引）
    index-fields = "id"

    # 指定检查的数据的范围，需要符合 sql 中 where 条件的语法
    range = "age > 10 AND age < 20"

    # 如果是对比多个分表与总表的数据，则设置为 true
    is-sharding = false

    # 在某些情况下字符类型的数据的排序会不一致，通过指定 collation 来保证排序的一致，
    # 需要与数据库中 charset 的设置相对应
    # collation = "latin1_bin"

    # 忽略某些列的检查，例如 sync-diff-inspector 目前还不支持的一些类型（json，bit，blob 等），
    # 或者是浮点类型数据在 TiDB 和 MySQL 中的表现可能存在差异，可以使用 ignore-columns
        ↪ 忽略检查这些列
    # ignore-columns = ["name"]

### 下面是一个对比不同库名和表名的两个表的配置示例
[[table-config]]
    # 目标库名
    schema = "test"

    # 目标表名
    table = "test2"
```

```
    # 非分库分表场景，设置为 false
    is-sharding = false

    # 源数据的配置
    [[table-config.source-tables]]
        # 源库的实例 id
        instance-id = "source-1"
        # 源数据库的名称
        schema = "test"
        # 源表的名称
        table  = "test1"

########################### Databases config ##########################

### 源数据库实例的配置
[[source-db]]
    host = "127.0.0.1"
    port = 3306
    user = "root"
    password = "123456"
    # 源数据库实例的 id，唯一标识一个数据库实例
    instance-id = "source-1"
    # 使用 TiDB 的 snapshot 功能，如果开启的话会使用历史数据进行对比
    # snapshot = "2016-10-08 16:45:26"
    # 设置数据库的 sql-mode，用于解析表结构
    # sql-mode = ""

### 目标数据库实例的配置
[target-db]
    host = "127.0.0.1"
    port = 4000
    user = "root"
    password = "123456"
    # 使用 TiDB 的 snapshot 功能，如果开启的话会使用历史数据进行对比
    # snapshot = "2016-10-08 16:45:26"
    # 设置数据库的 sql-mode，用于解析表结构
    # sql-mode = ""
```

11.12.1.1.4 运行 sync-diff-inspector

执行如下命令：

```
./bin/sync_diff_inspector --config=./config.toml
```

该命令最终会在日志中输出一个检查报告，说明每个表的检查情况。如果数据存在不一致的情况，sync-diff-inspector 会生成 SQL 修复不一致的数据，并将这些 SQL 语句保存到 `fix.sql` 文件中。

日志

sync-diff-inspector 会在运行时定期（间隔 10s）输出校验进度到日志中，格式如下：

```
[2020/11/12 17:47:00.170 +08:00] [INFO] [checkpoint.go:276] ["summary info"] [instance_id=target]
    ↪  [schema=test] [table=test_table] ["chunk num"=1000] ["success num"=80] ["failed num"=1]
    ↪ ["ignore num"=0]
```

- chunk num：总共需要校验的 chunk 数量。
- success num：已经校验数据一致的 chunk 数量。
- failed num：校验失败的 chunk 数量。校验时遇到错误和数据不一致两种情况都属于校验失败。
- ignore num：被忽略校验的 chunk 数量。当配置项 `sample-percent` 的值小于 100 时，sync-diff-inspector 会采用抽样的方式校验数据，这样就会有部分 chunk 被忽略校验。

校验结果

当校验结束时，sync-diff-inspector 会输出一份校验报告。

- 数据校验一致的日志示例如下：

```
[2020/11/12 17:47:00.174 +08:00] [INFO] [report.go:80] ["check result summary"] ["check
    ↪ passed num"=1] ["check failed num"=0]
[2020/11/12 17:47:00.174 +08:00] [INFO] [report.go:87] ["table check result"] [schema=test]
    ↪ [table=test_table] ["struct equal"=true] ["data equal"=true]
[2020/11/12 17:47:00.174 +08:00] [INFO] [main.go:75] ["check data finished"] [cost
    ↪ =353.462744ms]
[2020/11/12 17:47:00.174 +08:00] [INFO] [main.go:69] ["check pass!!!"]
```

- 数据校验不一致或者遇到错误时的日志示例如下：

```
[2020/11/12 18:16:17.068 +08:00] [INFO] [checkpoint.go:276] ["summary info"] [instance_id=
    ↪ target] [schema=test] [table=test1] ["chunk num"=1] ["success num"=0] ["failed num
    ↪ "=1] ["ignore num"=0]
[2020/11/12 18:16:17.071 +08:00] [INFO] [report.go:80] ["check result summary"] ["check
    ↪ passed num"=0] ["check failed num"=1]
[2020/11/12 18:16:17.071 +08:00] [INFO] [report.go:87] ["table check result"] [schema=test]
    ↪ [table=test_table] ["struct equal"=true] ["data equal"=false]
[2020/11/12 18:16:17.071 +08:00] [INFO] [main.go:75] ["check data finished"] [cost
    ↪ =319.849706ms]
[2020/11/12 18:16:17.071 +08:00] [WARN] [main.go:66] ["check failed!!!"]
```

校验通过和未通过的表的个数打印在 `check result summary` 中。所有表的校验结果的打印在 `table check ↪ result` 中。

11.12.1.1.5 注意事项

- sync-diff-inspector 在校验数据时会消耗一定的服务器资源，需要避免在业务高峰期间校验。
- TiDB 使用的 collation 为 `utf8_bin`。如果对 MySQL 和 TiDB 的数据进行对比，需要注意 MySQL 中表的 collation 设置。如果表的主键 / 唯一键为 varchar 类型，且 MySQL 中 collation 设置与 TiDB 不同，可能会因为排序问题导致最终校验结果不正确，需要在 sync-diff-inspector 的配置文件中增加 collation 设置。
- sync-diff-inspector 会优先使用 TiDB 的统计信息来划分 chunk，需要尽量保证统计信息精确，可以在业务空闲期手动执行 `analyze table {table_name}`。
- table-rule 的规则需要特殊注意，例如设置了 `schema-pattern="test1"`，`target-schema="test2"`，会对比 source 中的 `test1` 库和 target 中的 `test2` 库；如果 source 中有 `test2` 库，该库也会和 target 中的 `test2` 库进行对比。
- 生成的 `fix.sql` 仅作为修复数据的参考，需要确认后再执行这些 SQL 修复数据。

11.12.2 不同库名或表名的数据校验

用户在使用 DM 等同步工具时，可以设置 `route-rules` 将数据同步到下游指定表中。sync-diff-inspector 提供了校验不同库名、表名的表的功能。

下面是一个简单的例子：

```
########################### Tables config ##########################

### 配置需要对比的*目标数据库*中的表
[[check-tables]]
    # 目标库中数据库的名称
    schema = "test_2"

    # 需要检查的表
    tables = ["t_2"]

### 下面是一个对比不同库名和表名的两个表的配置示例
[[table-config]]
    # 目标库名
    schema = "test_2"

    # 目标表名
    table = "t_2"

    # 源数据的配置
    [[table-config.source-tables]]
        # 源库的实例 id
        instance-id = "source-1"
        # 源数据库的名称
        schema = "test_1"
        # 源表的名称
        table  = "t_1"
```

使用该配置会对下游的 test_2.t_2 与实例 source-1 中的 test_1.t_1 进行校验。

如果需要校验大量的不同库名或者表名的表，可以通过 table-rule 设置映射关系来简化配置。可以只配置 schema 或者 table 的映射关系，也可以都配置。例如上游库 test_1 中的所有表都同步到了下游的 test_2 库中，可以使用如下配置进行校验：

```
########################### Tables config ##########################

### 配置需要对比的*目标数据库*中的表
[[check-tables]]
    # 目标库中数据库的名称
    schema = "test_2"

    # 检查所有表
    tables = ["~^"]

[[table-rules]]
    # schema-pattern 和 table-pattern 支持通配符 *?
    schema-pattern = "test_1"
    #table-pattern = ""
    target-schema = "test_2"
    #target-table = ""
```

11.12.3 分库分表场景下的数据校验

sync-diff-inspector 支持对分库分表场景进行数据校验。例如有多个 MySQL 实例，使用同步工具 DM 同步到一个 TiDB 中，用户可以使用 sync-diff-inspector 对上下游数据进行校验。

11.12.3.1 使用 table-config 进行配置

使用 table-config 对 table-0 进行特殊配置，设置 is-sharding=true，并且在 table-config.source-tables 中配置上游表信息。这种配置方式需要对所有分表进行设置，适合上游分表数量较少，且分表的命名规则没有规律的场景。场景如图所示：

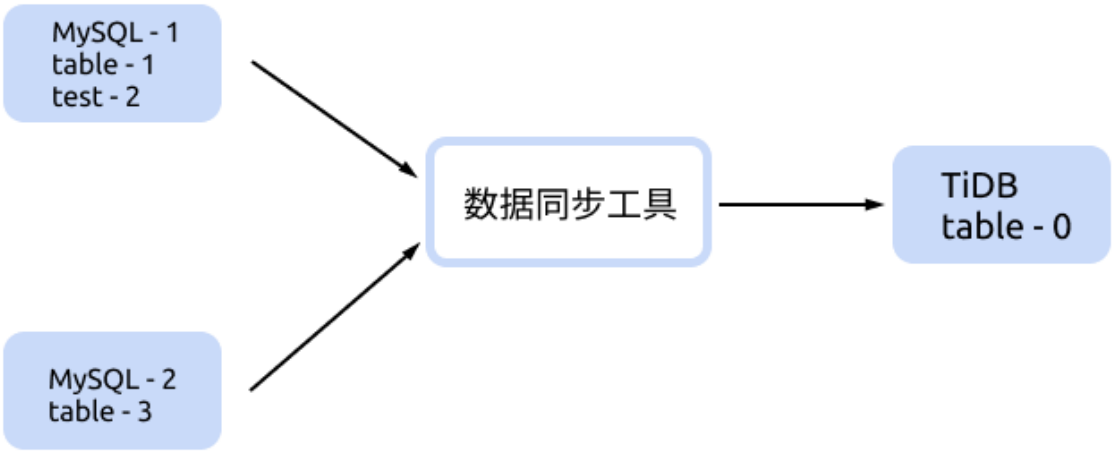


图 182: shard-table-sync-1

sync-diff-inspector 完整的示例配置如下：

```
### Diff Configuration.

########################### Global config ###########################

### 日志级别，可以设置为 info、debug
log-level = "info"

### sync-diff-inspector 根据主键 / 唯一键 / 索引将数据划分为多个 chunk，
### 对每一个 chunk 的数据进行对比。使用 chunk-size 设置 chunk 的大小
chunk-size = 1000

### 检查数据的线程数量
check-thread-count = 4

### 抽样检查的比例，如果设置为 100 则检查全部数据
sample-percent = 100

### 通过计算 chunk 的 checksum 来对比数据，如果不开启则逐行对比数据
use-checksum = true

### 如果设置为 true 则只会通过计算 checksum 来校验数据，如果上下游的 checksum
    ↪ 不一致也不会查出数据再进行校验
only-use-checksum = false

### 是否使用上次校验的 checkpoint，如果开启，则只校验上次未校验以及校验失败的 chunk
```

```
use-checkpoint = true

### 不对比数据
ignore-data-check = false

### 不对比表结构
ignore-struct-check = false

### 保存用于修复数据的 sql 的文件名称
fix-sql-file = "fix.sql"

########################## Tables config ##########################

### 配置需要对比的目标数据库中的表
[[check-tables]]
    # 库的名称
    schema = "test"

    # 需要检查的表的名称
    tables = ["table-0"]

### 配置该表对应的分表的相关配置
[[table-config]]
    # 目标库的名称
    schema = "test"

    # 目标库中表的名称
    table = "table-0"

    # 为分库分表场景下数据的对比，设置为 true
    is-sharding = true

    # 源数据表的配置
    [[table-config.source-tables]]
    # 源数据库实例的 id
    instance-id = "MySQL-1"
    schema = "test"
    table  = "table-1"

    [[table-config.source-tables]]
    # 源数据库实例的 id
    instance-id = "MySQL-1"
    schema = "test"
    table  = "test-2"
```

```
    [[table-config.source-tables]]
    # 源数据库实例的 id
    instance-id = "MySQL-2"
    schema = "test"
    table  = "table-3"

########################### Databases config ##########################

### 源数据库实例的配置
[[source-db]]
    host = "127.0.0.1"
    port = 3306
    user = "root"
    password = "123456"
    instance-id = "MySQL-1"

### 源数据库实例的配置
[[source-db]]
    host = "127.0.0.2"
    port = 3306
    user = "root"
    password = "123456"
    instance-id = "MySQL-2"

### 目标数据库实例的配置
[target-db]
    host = "127.0.0.3"
    port = 4000
    user = "root"
    password = "123456"
    instance-id = "target-1"
```

11.12.3.2 使用 table-rules 进行配置

当上游分表较多，且所有分表的命名都符合一定的规则时，则可以使用 `table-rules` 进行配置。场景如图所示：

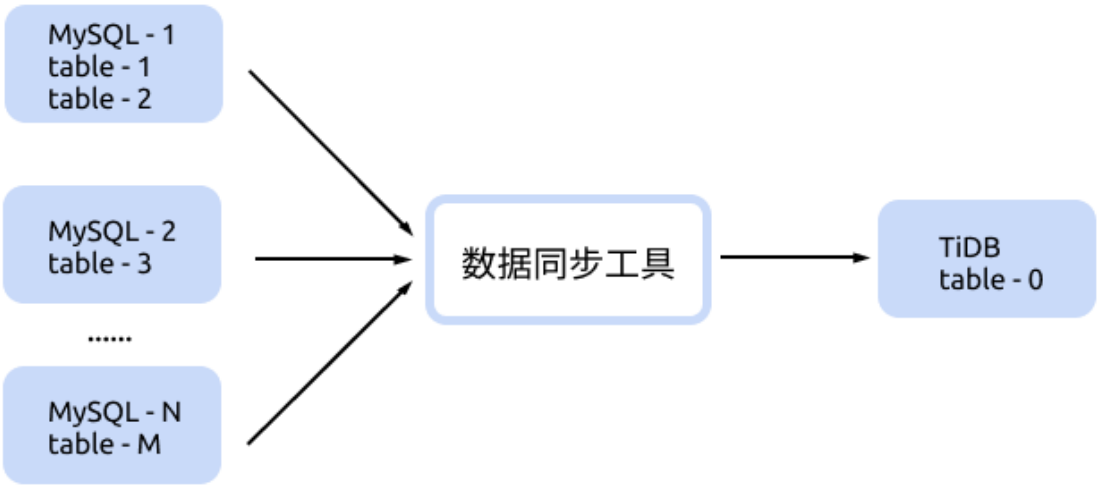


图 183: shard-table-sync-2

sync-diff-inspector 完整的示例配置如下：

```
### Diff Configuration.

########################### Global config ###########################

### 日志级别，可以设置为 info、debug
log-level = "info"

### sync-diff-inspector 根据主键 / 唯一键 / 索引将数据划分为多个 chunk，
### 对每一个 chunk 的数据进行对比。使用 chunk-size 设置 chunk 的大小
chunk-size = 1000

### 检查数据的线程数量
check-thread-count = 4

### 抽样检查的比例，如果设置为 100 则检查全部数据
sample-percent = 100

### 通过计算 chunk 的 checksum 来对比数据，如果不开启则逐行对比数据
use-checksum = true

### 如果设置为 true 则只会通过计算 checksum 来校验数据，如果上下游的 checksum
    ↪ 不一致也不会查出数据再进行校验
only-use-checksum = false

### 是否使用上次校验的 checkpoint，如果开启，则只校验上次未校验以及校验失败的 chunk
```

```
use-checkpoint = true

### 不对比数据
ignore-data-check = false

### 不对比表结构
ignore-struct-check = false

### 保存用于修复数据的 sql 的文件名称
fix-sql-file = "fix.sql"

########################### Tables config ########################

### 配置需要对比的目标数据库中的表
[[check-tables]]
    # 库的名称
    schema = "test"

    # 需要检查的表的名称
    tables = ["table-0"]

### 通过 table-rule 来设置上游分表与下游总表的映射关系。可以只配置 schema 或者 table 的映射关系，
    ↪ 也可以都配置
[[table-rules]]
    # schema-pattern 和 table-pattern 支持通配符 *?
    # 在 source-db 中配置的上游数据库中所有满足 schema-pattern 和 table-pattern 规则的表都为
        ↪ target-schema.target-table 的分表
    schema-pattern = "test"
    table-pattern = "table-*"
    target-schema = "test"
    target-table = "table-0"

########################### Databases config ########################

### 源数据库实例的配置
[[source-db]]
    host = "127.0.0.1"
    port = 3306
    user = "root"
    password = "123456"
    instance-id = "MySQL-1"

### 源数据库实例的配置
[[source-db]]
    host = "127.0.0.2"
```

```
    port = 3306
    user = "root"
    password = "123456"
    instance-id = "MySQL-2"

### 目标数据库实例的配置
[target-db]
    host = "127.0.0.3"
    port = 4000
    user = "root"
    password = "123456"
    instance-id = "target-1"
```

11.12.4 TiDB 主从集群的数据校验

用户可以使用 TiDB Binlog 搭建 TiDB 的主从集群，Drainer 在把数据同步到 TiDB 时，保存 checkpoint 的同时也会将上下游的 TSO 对应关系保存为 `ts-map`。在 sync-diff-inspector 中配置 `snapshot` 即可对 TiDB 主从集群的数据进行校验。

11.12.4.1 获取 ts-map

在下游 TiDB 中执行以下 SQL 语句：

```
select * from tidb_binlog.checkpoint;
```

```
+--------------------+-----------------------------------------------------------------------------------------------
    ↪
| clusterID          | checkPoint
    ↪
    ↪ |
+--------------------+-----------------------------------------------------------------------------------------------
    ↪
| 6711243465327639221 | {"commitTS":409622383615541249,"ts-map":{"primary-ts
    ↪ ":409621863377928194,"secondary-ts":409621863377928345}} |
+--------------------+-----------------------------------------------------------------------------------------------
    ↪
```

从结果中可以获取 ts-map 信息。

11.12.4.2 配置 snapshot

使用上一步骤获取的 ts-map 信息来配置上下游数据库的 snapshot 信息。其中的 `Databases config` 部分示例配置如下：

```
########################### Databases config ########################
```

```
### 源数据库实例的配置
[[source-db]]
    host = "127.0.0.1"
    port = 4000
    user = "root"
    password = "123456"
    # 源数据库实例的 id，唯一标识一个数据库实例
    instance-id = "source-1"
    # 使用 TiDB 的 snapshot 功能，对应 ts-map 中的 master-ts
    snapshot = "409621863377928194"

### 目标数据库实例的配置
[target-db]
    host = "127.0.0.1"
    port = 4001
    user = "root"
    password = "123456"
    # 使用 TiDB 的 snapshot 功能，对应 ts-map 中的 slave-ts
    snapshot = "409621863377928345"
```

11.12.4.3 注意事项

- Drainer 的 db-type 需要设置为 tidb，这样才会在 checkpoint 中保存 ts-map。
- 需要调整 TiKV 的 GC 时间，保证在校验时 snapshot 对应的历史数据不会被执行 GC。建议调整为 1 个小时，在校验后再还原 GC 设置。

11.13 Loader 使用文档

> **警告：**
>
> Loader 目前已经不再维护，其功能已经完全被TiDB Lightning 的 TiDB backend 功能取代，强烈建议切换到 TiDB Lightning。

11.13.1 Loader 简介

Loader 是由 PingCAP 开发的数据导入工具，用于向 TiDB 中导入数据。

Loader 包含在 tidb-enterprise-tools 安装包中，可在此下载。

11.13.2 为什么我们要做这个工具

当数据量比较大的时候，如果用 mysqldump 这样的工具迁移数据会比较慢。我们尝试了 Mydumper/myloader 套件，能够多线程导出和导入数据。在使用过程中，Mydumper 问题不大，但是 myloader 由于缺乏出错重试、断

点续传这样的功能，使用起来很不方便。所以我们开发了 loader，能够读取 Mydumper 的输出数据文件，通过 MySQL protocol 向 TiDB/MySQL 中导入数据。

11.13.3 Loader 有哪些优点

- 多线程导入
- 支持表级别的并发导入，分散写入热点
- 支持对单个大表并发导入，分散写入热点
- 支持 Mydumper 数据格式
- 出错重试
- 断点续导
- 通过 system variable 优化 TiDB 导入数据速度

11.13.4 使用方法

11.13.4.1 注意事项

请勿使用 loader 导入 MySQL 实例中 `mysql` 系统数据库到下游 TiDB。

如果 Mydumper 使用 -m 参数，会导出不带表结构的数据，这时 loader 无法导入数据。

如果使用默认的 `checkpoint-schema` 参数，在导完一个 database 数据库后，请 `drop database tidb_loader` 后再开始导入下一个 database。

推荐数据库开始导入的时候，明确指定 `checkpoint-schema = "tidb_loader"` 参数。

11.13.4.2 参数说明

```
-L string
    log 级别设置，可以设置为 debug, info, warn, error, fatal (默认为 "info")
-P int
    TiDB/MySQL 的端口 (默认为 4000)
-V
    打印 loader 版本
-c string
    指定配置文件启动 loader
-checkpoint-schema string
    checkpoint 数据库名，loader 在运行过程中会不断的更新这个数据库，在中断并恢复后，
        ↪ 会通过这个库获取上次运行的进度 (默认为 "tidb_loader")
-d string
     需要导入的数据存放路径 (default "./")
-h string
     TiDB 服务 host IP  (default "127.0.0.1")
-p string
    TiDB 账户密码
-status-addr string
    Prometheus 可以通过该地址拉取 Loader metrics，也是 Loader 的 pprof 调试地址 (默认为
        ↪ ":8272")。
```

```
  -t int
      线程数（默认为 16）. 每个线程同一时刻只能操作一个数据文件。
  -u string
      TiDB 的用户名（默认为 "root"）
```

11.13.4.3 配置文件

除了使用命令行参数外，还可以使用配置文件来配置，配置文件的格式如下：

```
## 日志输出等级；可以设置为 debug, info, warn, error, fatal（默认为 "info"）
log-level = "info"

## 指定 loader 日志目录
log-file = "loader.log"

## 需要导入的数据存放路径（default "./"）
dir = "./"

### Prometheus 可以通过该地址拉取 Loader metrics，也是 Loader 的 pprof 调试地址（默认为 ":8272"）
    ↪ 。
status-addr = ":8272"

## checkpoint 数据库名，loader 在运行过程中会不断的更新这个数据库，在中断并恢复后，
## 会通过这个库获取上次运行的进度（默认为 "tidb_loader"）
checkpoint-schema = "tidb_loader"

## 线程数（默认为 16）. 每个线程同一时刻只能操作一个数据文件。
pool-size = 16

## 目标数据库信息
[db]
host = "127.0.0.1"
user = "root"
password = ""
port = 4000

## 导入数据时数据库连接所使用的 session 级别的 `sql_mode`。如果 `sql-mode` 的值没有提供或者设置为
    ↪  "@DownstreamDefault"，会使用下游 global 级别的 `sql_mode`。
## sql-mode = ""
## `max-allowed-packet` 设置数据库连接允许的最大数据包大小，对应于系统参数中的 `
    ↪ max_allowed_packet`。 如果设置为 0，会使用下游数据库 global 级别的 `max_allowed_packet`。
max-allowed-packet = 67108864

## sharding 同步规则，采用 wildcharacter
## 1\. 星号字符（*）可以匹配零个或者多个字符，
```

```
##    例子，doc* 匹配 doc 和 document，但是和 dodo 不匹配；
##    星号只能放在 pattern 结尾，并且一个 pattern 中只能有一个
## 2\. 问号字符（?）匹配任一一个字符

## [[route-rules]]
## pattern-schema = "shard_db_*"
## pattern-table = "shard_table_*"
## target-schema = "shard_db"
## target-table = "shard_table"
```

11.13.4.4 使用示例

通过命令行参数：

```
./bin/loader -d ./test -h 127.0.0.1 -u root -P 4000
```

或者使用配置文件“config.toml”：

```
./bin/loader -c=config.toml
```

11.13.5 FAQ

11.13.5.1 合库合表场景案例说明

根据配置文件的 route-rules 可以支持将分库分表的数据导入到同一个库同一个表中，但是在开始前需要检查分库分表规则：

- 是否可以利用 route-rules 的语义规则表示
- 分表中是否包含唯一递增主键，或者合并后是否包含数据上有冲突的唯一索引或者主键

Loader 需要配置文件中开启 route-rules 参数以提供合库合表功能

- 如果使用该功能，必须填写 `pattern-schema` 与 `target-schema`
- 如果 `pattern-table` 与 `target-table` 为空，将不进行表名称合并或转换

```
[[route-rules]]
pattern-schema = "example_db"
pattern-table = "table_*"
target-schema = "example_db"
target-table = "table"
```

11.13.5.2 全量导入过程中遇到报错`packet for query is too large. Try adjusting the 'max_allowed_packet' variable`

11.13.5.2.1 原因

- MySQL client 和 MySQL/TiDB Server 都有 `max_allowed_packet` 配额的限制，如果在使用过程中违反其中任何一个 `max_allowed_packet` 配额，客户端程序就会收到对应的报错。目前最新版本的 Loader 和 TiDB Server 的默认 `max_allowed_packet` 配额都为 64M。
  - 请使用最新版本，或者最新稳定版本的工具。下载页面。
- Loader 的全量数据导入处理模块不支持对 dump sqls 文件进行切分，原因是 Mydumper 采用了最简单的编码实现，正如 Mydumper 代码注释 `/* Poor man's data dump code */` 所言。如果在 Loader 实现文件切分，那么需要在 `TiDB parser` 基础上实现一个完备的解析器才能正确的处理数据切分，但是随之会带来以下的问题：
  - 工作量大
  - 复杂度高，不容易保证正确性
  - 性能的极大降低

11.13.5.2.2 解决方案

- 依据上面的原因，在代码层面不能简单的解决这个困扰，我们推荐的方式是：利用 Mydumper 提供的控制 `Insert Statement` 大小的功能 `-s, --statement-size: Attempted size of INSERT statement in bytes, default 1000000`。

  依据默认的 `--statement-size` 设置，Mydumper 默认生成的 `Insert Statement` 大小会尽量接近在 `1M` 左右，使用默认值就可以确保绝大部分情况不会出现该问题。

  有时候在 dump 过程中会出现下面的 `WARN` log，但是这个报错不影响 dump 的过程，只是表达了 dump 的表可能是宽表。

```
Row bigger than statement_size for xxx
```

- 如果宽表的单行超过了 `64M`，那么需要修改以下两个配置，并且使之生效。
  - 在 TiDB Server 执行 `set @@global.max_allowed_packet=134217728（134217728 = 128M）`
  - 根据实际情况为 Loader 的配置文件中的 db 配置增加 `max-allowed-packet=128M`，然后重启进程或者任务

11.14 Mydumper 使用文档

11.14.1 Mydumper 简介

> **警告：**
>
> PingCAP 之前维护的 Mydumper 工具 fork 自 mydumper project，针对 TiDB 的特性进行了优化。从 v7.5.0 开始，Mydumper 废弃，其绝大部分功能已经被Dumpling 取代，强烈建议切换到 Dumpling。

11.14.1.1 相比于普通的 Mydumper，此工具有哪些改进之处？

- 对于 TiDB 可以设置tidb_snapshot 的值指定备份数据的时间点，从而保证备份的一致性，而不是通过 `FLUSH TABLES WITH READ LOCK` 来保证备份一致性。
- 使用 TiDB 的隐藏列 `_tidb_rowid` 优化了单表内数据的并发导出性能。

11.14.2 基本用法

11.14.2.1 新添参数

- `-z` 或 `--tidb-snapshot`：设置 `tidb_snapshot` 用于备份。默认值为当前 TSO（`SHOW MASTER STATUS` 输出的 `Position` 字段）。此参数可设为 TSO 或有效的 `datetime` 时间，例如：`-z "2016-10-08 16:45:26"`。

11.14.2.2 需要的权限

- SELECT
- RELOAD
- LOCK TABLES
- REPLICATION CLIENT

11.14.2.3 使用举例

执行如下命令从 TiDB 备份数据，需要根据实际情况添加命令行参数：

```
./bin/mydumper -h 127.0.0.1 -u root -P 4000
```

11.14.3 表内并发 Dump

11.14.3.1 原理

Mydumper 首先计算 `min(_tidb_rowid)` 和 `max(_tidb_rowid)`，然后按照 `-r` 设定的值对表划分 chunks，将 chunks 分配到不同线程并发导出。

11.14.3.2 并发 Dump 相关参数

- `-t` 或 `--threads`：并发线程数，默认值为 4。
- `-r` 或 `--rows`：每个 chunks 包含的最大行数。设置该值后，Mydumper 将会忽略 `--chunk-filesize` 值。

11.14.3.3 示例

以下是一条完整的 Mydumper 命令：

```
./bin/mydumper -h 127.0.0.1 -u root -P 4000 -r 10000 -t 4
```

11.14.3.4 支持 _tidb_rowid 索引的 TiDB 版本

由于表内并发使用 TiDB 的隐藏列 _tidb_rowid，数据库需要支持 _tidb_rowid 索引才能发挥并发导出的优势。

以下 TiDB 版本支持 _tidb_rowid 索引：

- v2.1.3 及以上
- v3.0 及以上，包括 v3.1 及未来版本

11.14.3.5 性能评估

在 Dump 操作前需要进行性能评估。由于并发 Scan 操作对 TiDB、TiKV 集群都会产生一定压力，所以需要评估与测试 Dump 操作对数据库集群和业务的影响。

11.14.4 FAQ

11.14.4.1 使用的 --tidb-snapshot 导出时报错，怎么处理？

需要执行命令时增加一个参数 –skip-tz-utc，如果不设置的话 Mydumper 会预先设置 UTC 时区，然后在设置 tidb-snapshot 的时候会做时区转化，就会有问题

11.14.4.2 如何判断使用的 Mydumper 是否为 PingCAP 优化的版本？

执行如下命令：

```
./bin/mydumper -V
```

如果输出结果中包含 githash（如下列示例输出中的 d3e6fec8b069daee772d0dbaa47579f67a5947e7），则使用的 Mydumper 为 PingCAP 优化的版本。

```
mydumper 0.9.5 (d3e6fec8b069daee772d0dbaa47579f67a5947e7), built against MySQL 5.7.24
```

11.14.4.3 使用 Loader 恢复 Mydumper 备份出来的数据时报错 “invalid mydumper files for there are no -schema-create.sql files found”，应该如何解决？

检查使用 Mydumper 备份数据时是否使用了 -T 或者 --tables-list 配置，如果使用了这些配置，Mydumper 就不会生成包含建库 SQL 的文件。

解决方法：在 Mydumper 备份数据目录下创建文件 {schema-name}-schema-create.sql，在文件中写入 “CREATE DATABASE {schema-name}”，再运行 Loader 即可。

11.14.4.4 为什么使用 Mydumper 导出来的 TIMESTAMP 类型的数据和数据库中的数据不一致？

检查一下运行 Mydumper 的服务器的时区与数据库的时区是否一致，Mydumper 会根据运行所在服务器的时区对 TIMESTAMP 类型的数据进行转化，可以给 Mydumper 加上 --skip-tz-utc 参数禁止这种转化。

11.14.4.5 如何配置 Mydumper 的参数 -F，--chunk-filesize？

Mydumper 在备份时会根据这个参数的值把每个表的数据划分成多个 chunk，每个 chunk 保存到一个文件中，大小约为 chunk-filesize。根据这个参数把数据切分到多个文件中，这样就可以利用 Loader/TiDB Lightning 的并行处理逻辑提高导入速度。如果后续使用 Loader 对备份文件进行恢复，建议把该参数的值设置为 64（单位 MB）；如果使用 TiDB Lightning 恢复，则建议设置为 256（单位 MB）。

11.14.4.6 如何配置 Mydumper 的参数 -s --statement-size？

Mydumper 使用该参数控制 Insert Statement 的大小，默认值为 10000000（约 1 MB）。使用该参数来尽量避免在恢复数据时报以下错误：

```
packet for query is too large. Try adjusting the 'max_allowed_packet' variable
```

默认值在绝大部分情况下都可以满足需求，但是如果表为宽表，单行数据的大小可能超过 statement-size 的限制，Mydumper 会报如下的 Warning：

```
Row bigger than statement_size for xxx
```

此时恢复数据时仍然会报 packet for query is too large 的错误日志，这个时候需要修改以下两个配置（以设置为 128M 为例）：

- 在 TiDB Server 执行 set @@global.max_allowed_packet=134217728（134217728 = 128M）。
- 根据实际情况为 Loader 的配置文件或者 DM task 配置文件中的 db 配置增加类似 max-allowed-packet=128M 的语句，然后重启进程或者任务。

11.14.4.7 如何设置 Mydumper 的参数 -l，--long-query-guard？

把该参数设置为预估备份需要消耗的时间，如果 Mydumper 运行时间超过该参数的值，就会报错退出。推荐初次备份设置为 7200（单位：秒），之后可根据具体备份时间进行调整。

11.14.4.8 如何设置 Mydumper 的参数 --tidb-force-priority？

仅当备份 TiDB 的数据时才可以设置该参数，值可以为 LOW_PRIORITY，DELAYED 或者 HIGH_PRIORITY。如果不希望数据备份对线上业务造成影响，推荐将该参数设置为 LOW_PRIORITY；如果备份的优先级更高，则可以设置为 HIGH_PRIORITY。

11.14.4.9 Mydumper 备份 TiDB 数据报错“GC life time is shorter than transaction duration”应该怎么解决？

Mydumper 备份 TiDB 数据时为了保证数据的一致性使用了 TiDB 的 snapshot 特性，如果备份过程中 snapshot 对应的历史数据被 TiDB GC 处理了，则会报该错误。解决步骤如下：

1. 在备份前，使用 MySQL 客户端查询 TiDB 集群的 tikv_gc_life_time 的值，并将其调整为一个合适的值：

```
SELECT * FROM mysql.tidb WHERE VARIABLE_NAME = 'tikv_gc_life_time';
```

```
+----------------------+-------------------------------------------------------------------------------------------
    ↪
| VARIABLE_NAME         | VARIABLE_VALUE
    ↪                                                                                                  |
+----------------------+-------------------------------------------------------------------------------------------
    ↪
| tikv_gc_life_time     | 10m0s
    ↪
    ↪ |
+----------------------+-------------------------------------------------------------------------------------------
    ↪
1 rows in set (0.02 sec)
```

```
update mysql.tidb set VARIABLE_VALUE = '720h' where VARIABLE_NAME = 'tikv_gc_life_time';
```

2. 备份完成后，将 `tikv_gc_life_time` 调整为原来的值：

```
update mysql.tidb set VARIABLE_VALUE = '10m0s' where VARIABLE_NAME = 'tikv_gc_life_time';
```

11.14.4.10 Mydumper 的参数 `--tidb-rowid` 是否需要配置？

如果设置该参数为 true，则导出的数据中会包含 TiDB 的隐藏列的数据。将数据恢复到 TiDB 的时候使用隐藏列会有数据不一致的风险，目前不推荐使用该参数。

11.14.4.11 Mydumper 报错 “Segmentation fault” 怎么解决？

该 bug 已修复。如果仍然报错，可尝试升级到最新版本。

11.14.4.12 Mydumper 报错 “Error dumping table ({schema}.{table}) data: line …… (total length …)” 怎么解决？

Mydumper 解析 SQL 时报错，可尝试使用最新版本。如果仍然报错，可以提 issue 到 mydumper/issues。

11.14.4.13 Mydumper 报错 “Failed to set tidb_snapshot: parsing time ” 20190901-10:15:00 +0800” as ” 20190901-10:15:00 +0700 MST”: cannot parse ” ” as ” MST” ” 如何解决？

检查 TiDB 的版本是否低于 v2.1.11。如果是的话，需要升级 TiDB 到 v2.1.11 或以上版本。

11.14.4.14 未来是否计划让 PingCAP 对 Mydumper 的改动合并到上游？

是的，PingCAP 团队计划将对 Mydumper 的改动合并到上游。参见 PR #155。

11.15 Syncer 使用文档

> **警告：**
>
> Syncer 目前已经不再维护，其功能已经完全被 TiDB Data Migration取代，强烈建议切换到 TiDB DM。

11.15.1 Syncer 简介

Syncer 是一个数据导入工具，能方便地将 MySQL 的数据增量导入到 TiDB。

Syncer 包含在 tidb-enterprise-tools 安装包中，可在此下载。

11.15.2 Syncer 架构

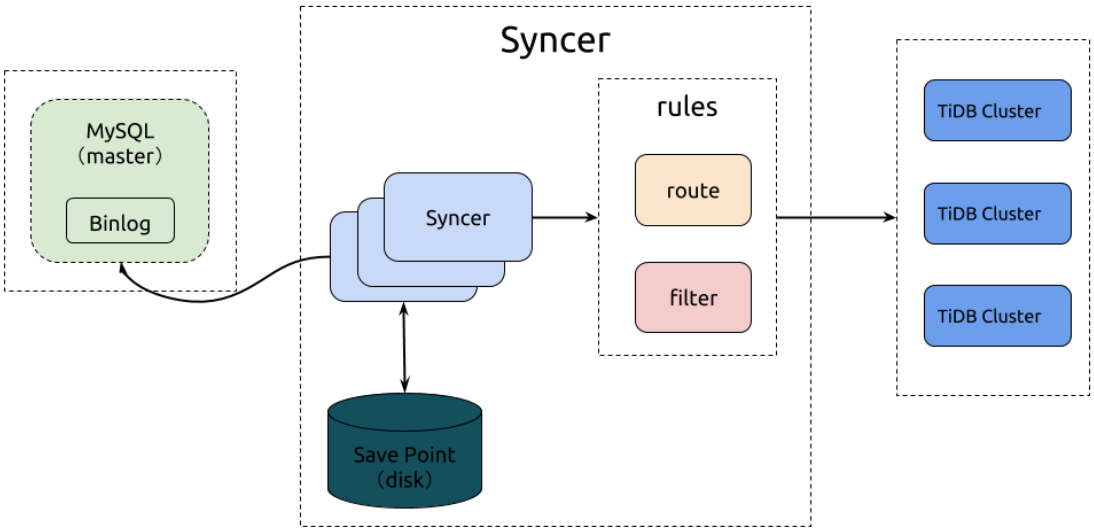


图 184: Syncer 架构

11.15.3 Syncer 部署位置

Syncer 可以部署在任一台可以连通对应的 MySQL 和 TiDB 集群的机器上，推荐部署在 TiDB 集群。

11.15.4 Syncer 增量导入数据示例

使用前请详细阅读Syncer 同步前预检查。

11.15.4.1 设置同步开始的 position

设置 Syncer 的 meta 文件, 这里假设 meta 文件是 `syncer.meta`:

```
cat syncer.meta
```

```
binlog-name = "mysql-bin.000003"
binlog-pos = 930143241
binlog-gtid = "2bfabd22-fff7-11e6-97f7-f02fa73bcb01:1-23,61ccbb5d-c82d-11e6-ac2e-487b6bd31bf7
    ↪ :1-4"
```

> **注意：**
>
> - `syncer.meta` 只需要第一次使用的时候配置，后续 Syncer 同步新的 binlog 之后会自动将其更新到最新的 position。
> - 如果使用 binlog position 同步则只需要配置 `binlog-name` 和 `binlog-pos`；如果使用 `binlog-gtid` 同步则需要设置 `binlog-gtid`，且启动 Syncer 时带有 `--enable-gtid`。

11.15.4.2 启动 Syncer

Syncer 的命令行参数说明：

```
Usage of Syncer:
  -L string
      日志等级: debug, info, warn, error, fatal (默认为 "info")
  -V
      输出 Syncer 版本；默认 false
  -auto-fix-gtid
      当 mysql master/slave 切换时，自动修复 gtid 信息；默认 false
  -b int
      batch 事务大小 (默认 100)
  -c int
      Syncer 处理 batch 线程数 (默认 16)
  -config string
      指定相应配置文件启动 Sycner 服务；如 `--config config.toml`
  -enable-ansi-quotes
      使用 ANSI_QUOTES sql_mode 来解析 SQL 语句
  -enable-gtid
      使用 gtid 模式启动 Syncer；默认 false，开启前需要上游 MySQL 开启 GTID 功能
  -flavor string
      上游数据库实例类型，目前支持 "mysql" 和 "mariadb"
  -log-file string
      指定日志文件目录；如 `--log-file ./syncer.log`
  -log-rotate string
```

```
    指定日志切割周期，hour/day（默认 "day"）
  -max-retry int
    SQL 请求由于网络异常等原因出错时的最大重试次数（默认值为 100）
  -meta string
    指定 Syncer 上游 meta 信息文件 （默认与配置文件相同目录下 "syncer.meta"）
  -persistent-dir string
    指定同步过程中历史 schema 结构的保存文件地址，如果设置为空，则不保存历史 schema 结构；
        ↪ 如果不为空，则根据 binlog 里面包含的数据的 column 长度选择 schema 来还原 DML 语句
  -safe-mode
    指定是否开启 safe mode，让 Syncer 在任何情况下可重入
  -server-id int
    指定 MySQL slave sever-id（默认 101）
  -status-addr string
    指定 Syncer metric 信息；如 `--status-addr 127:0.0.1:10088`
  -timezone string
    目标数据库使用的时区，请使用 IANA 时区标识符，如 `Asia/Shanghai`
```

Syncer 的配置文件 config.toml：

```
log-level = "info"
log-file = "syncer.log"
log-rotate = "day"

server-id = 101

### meta 文件地址
meta = "./syncer.meta"
worker-count = 16
batch = 100
flavor = "mysql"

### Prometheus 可以通过该地址拉取 Syncer metrics，也是 Syncer 的 pprof 调试地址
status-addr = ":8271"

### 如果设置为 true，Syncer 遇到 DDL 语句时就会停止退出
stop-on-ddl = false

### SQL 请求由于网络异常等原因出错时的最大重试次数
max-retry = 100

### 指定目标数据库使用的时区，binlog 中所有 timestamp 字段会按照该时区进行转换，默认使用 Syncer
    ↪ 本地时区
## timezone = "Asia/Shanghai"

### 跳过 DDL 语句，格式为 **前缀完全匹配**，如：`DROP TABLE ABC` 至少需要填入 `DROP TABLE`
## skip-ddls = ["ALTER USER", "CREATE USER"]
```

```
### 在使用 route-rules 功能后，
### replicate-do-db & replicate-ignore-db 匹配合表之后 (target-schema & target-table) 数值
### 优先级关系: replicate-do-db --> replicate-do-table --> replicate-ignore-db --> replicate-
    ↪ ignore-table
### 指定要同步数据库名；支持正则匹配，表达式语句必须以 `~` 开始
#replicate-do-db = ["~^b.*","s1"]

### 指定 **忽略** 同步数据库；支持正则匹配，表达式语句必须以 `~` 开始
#replicate-ignore-db = ["~^b.*","s1"]

## skip-dmls 支持跳过 DML binlog events，type 字段的值可为：'insert'，'update' 和 'delete'
## 跳过 foo.bar 表的所有 delete 语句
## [[skip-dmls]]
## db-name = "foo"
## tbl-name = "bar"
## type = "delete"
## # 跳过所有表的 delete 语句
## [[skip-dmls]]
## type = "delete"
## # 跳过 foo.* 表的 delete 语句
## [[skip-dmls]]
## db-name = "foo"
## type = "delete"

### 指定要同步的 db.table 表
### db-name 与 tbl-name 不支持 `db-name ="dbname，dbname2"` 格式
#[[replicate-do-table]]
#db-name ="dbname"
#tbl-name = "table-name"

#[[replicate-do-table]]
#db-name ="dbname1"
#tbl-name = "table-name1"

### 指定要同步的 db.table 表；支持正则匹配，表达式语句必须以 `~` 开始
#[[replicate-do-table]]
#db-name ="test"
#tbl-name = "~^a.*"

### 指定 **忽略** 同步数据库
### db-name & tbl-name 不支持 `db-name ="dbname，dbname2"` 语句格式
#[[replicate-ignore-table]]
#db-name = "your_db"
#tbl-name = "your_table"
```

```
### 指定要 **忽略** 同步数据库名；支持正则匹配，表达式语句必须以 `~` 开始
#[[replicate-ignore-table]]
#db-name ="test"
#tbl-name = "~^a.*"

## sharding 同步规则，采用 wildcharacter
## 1. 星号字符 (*) 可以匹配零个或者多个字符,
##    例子, doc* 匹配 doc 和 document, 但是和 dodo 不匹配;
##    星号只能放在 pattern 结尾，并且一个 pattern 中只能有一个
## 2. 问号字符 (?) 匹配任一一个字符

#[[route-rules]]
#pattern-schema = "route_*"
#pattern-table = "abc_*"
#target-schema = "route"
#target-table = "abc"

#[[route-rules]]
#pattern-schema = "route_*"
#pattern-table = "xyz_*"
#target-schema = "route"
#target-table = "xyz"

[from]
host = "127.0.0.1"
user = "root"
password = ""
port = 3306

[to]
host = "127.0.0.1"
user = "root"
password = ""
port = 4000
```

启动 Syncer：

```
./bin/syncer -config config.toml
```

```
2016/10/27 15:22:01 binlogsyncer.go:226: [info] begin to sync binlog from position (mysql-bin
    ↪ .000003, 1280)
2016/10/27 15:22:01 binlogsyncer.go:130: [info] register slave for master server 127.0.0.1:3306
2016/10/27 15:22:01 binlogsyncer.go:552: [info] rotate to (mysql-bin.000003, 1280)
2016/10/27 15:22:01 syncer.go:549: [info] rotate binlog to (mysql-bin.000003, 1280)
```

11.15.4.3 在 MySQL 中插入新的数据

```
INSERT INTO t1 VALUES (4, 4), (5, 5);
```

登录到 TiDB 查看：

```
mysql -h127.0.0.1 -P4000 -uroot -p
```

```
select * from t1;
```

```
+----+------+
| id | age  |
+----+------+
|  1 |    1 |
|  2 |    2 |
|  3 |    3 |
|  4 |    4 |
|  5 |    5 |
+----+------+
```

Syncer 每隔 30s 会输出当前的同步统计，如下所示：

```
2017/06/08 01:18:51 syncer.go:934: [info] [syncer]total events = 15, total tps = 130, recent tps
    ↪ = 4,
master-binlog = (ON.000001, 11992), master-binlog-gtid=53ea0ed1-9bf8-11e6-8bea-64006a897c73:1-74,
syncer-binlog = (ON.000001, 2504), syncer-binlog-gtid = 53ea0ed1-9bf8-11e6-8bea-64006a897c73:1-17
2017/06/08 01:19:21 syncer.go:934: [info] [syncer]total events = 15, total tps = 191, recent tps
    ↪ = 2,
master-binlog = (ON.000001, 11992), master-binlog-gtid=53ea0ed1-9bf8-11e6-8bea-64006a897c73:1-74,
syncer-binlog = (ON.000001, 2504), syncer-binlog-gtid = 53ea0ed1-9bf8-11e6-8bea-64006a897c73:1-35
```

由上述示例可见，使用 Syncer 可以自动将 MySQL 的更新同步到 TiDB。

11.15.5 Syncer 配置说明

11.15.5.1 指定数据库同步

本部分将通过实际案例描述 Syncer 同步数据库参数的优先级关系。

- 如果使用 route-rules 规则，参考Sharding 同步支持
- 优先级：replicate-do-db –> replicate-do-table –> replicate-ignore-db –> replicate-ignore-table

```
## 指定同步 ops 数据库
## 指定同步以 ti 开头的数据库
replicate-do-db = ["ops","~^ti.*"]

## china 数据库下有 guangzhou / shanghai / beijing 等多张表，只同步 shanghai 与 beijing 表。
```

```
## 指定同步 china 数据库下 shanghai 表
[[replicate-do-table]]
db-name ="china"
tbl-name = "shanghai"

## 指定同步 china 数据库下 beijing 表
[[replicate-do-table]]
db-name ="china"
tbl-name = "beijing"

## ops 数据库下有 ops_user / ops_admin / weekly 等数据表，只需要同步 ops_user 表。
## 因 replicate-do-db 优先级比 replicate-do-table 高，所以此处设置只同步 ops_user 表无效，
    ↪ 实际工作会同步 ops 整个数据库
[[replicate-do-table]]
db-name ="ops"
tbl-name = "ops_user"

## history 数据下有 2017_01 2017_02 ... 2017_12 / 2016_01  2016_02 ... 2016_12 等多张表，
    ↪ 只需要同步 2017 年的数据表
[[replicate-do-table]]
db-name ="history"
tbl-name = "~^2017_.*"

## 忽略同步 ops 与 fault 数据库
## 忽略同步以 www 开头的数据库
### 因 replicate-do-db 优先级比 replicate-ignore-db 高，所以此处忽略同步 ops 不生效。
replicate-ignore-db = ["ops","fault","~^www"]

## fault 数据库下有 faults / user_feedback / ticket 等数据表
## 忽略同步 user_feedback 数据表
## 因 replicate-ignore-db 优先级比 replicate-ignore-table 高，所以此处设置只忽略同步
    ↪ user_feedback 表无效，实际工作会忽略同步 fault 整个数据库
[[replicate-ignore-table]]
db-name = "fault"
tbl-name = "user_feedback"

## order 数据下有 2017_01 2017_02 ... 2017_12 / 2016_01  2016_02 ... 2016_12 等多张表，忽略 2016
    ↪ 年的数据表
[[replicate-ignore-table]]
db-name ="order"
tbl-name = "~^2016_.*"
```

11.15.5.2 Sharding 同步支持

根据配置文件的 route-rules，支持将分库分表的数据导入到同一个库同一个表中，但是在开始前需要检查分

库分表规则，如下：

- 是否可以利用 route-rules 的语义规则表示
- 分表中是否包含唯一递增主键，或者合并后是否包含数据上有冲突的唯一索引或者主键

暂时对 DDL 支持不完善。

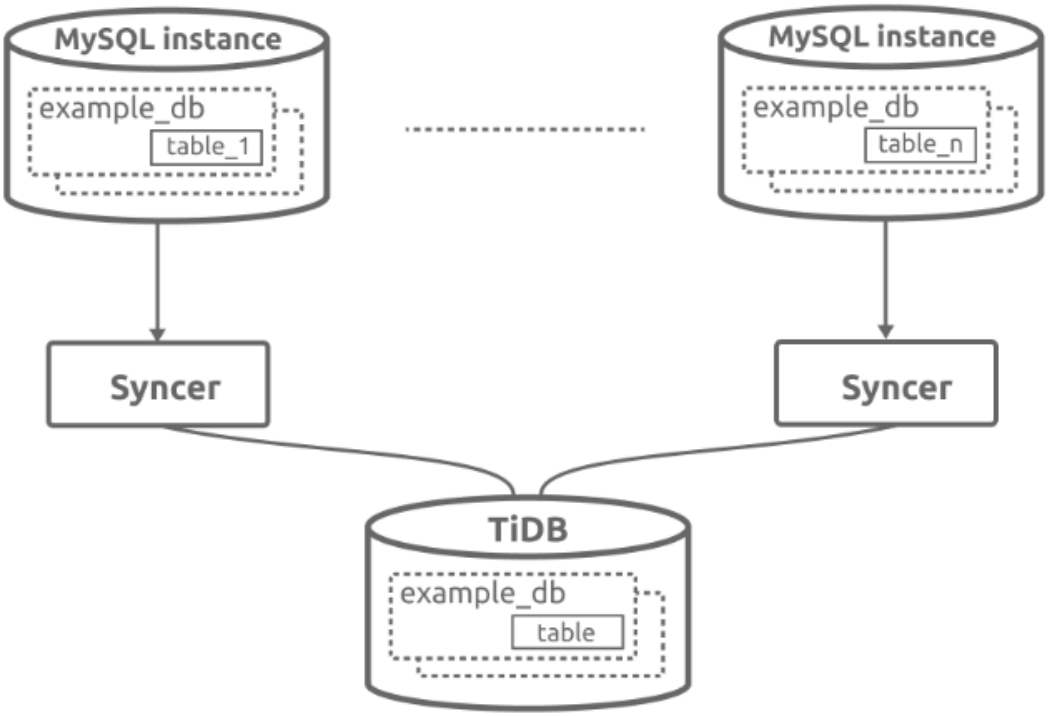


图 185: sharding

11.15.5.2.1 分库分表同步示例

1. 只需在所有 MySQL 实例下面，启动 Syncer, 并设置 route-rules。
2. `replicate-do-db` & `replicate-ignore-db` 与 route-rules 同时使用场景下，`replicate-do-db` & `replicate-ignore-db` 需要指定 route-rules 中 `target-schema` & `target-table` 的内容。

```
## 场景如下:
## 数据库A 下有 order_2016 / history_2016 等多个数据库
## 数据库B 下有 order_2017 / history_2017 等多个数据库
## 指定同步数据库A  order_2016 数据库，数据表如下 2016_01 2016_02 ... 2016_12
## 指定同步数据库B  order_2017 数据库，数据表如下 2017_01 2017_02 ... 2017_12
```

```
## 表内使用 order_id 作为主键，数据之间主键不冲突
## 忽略同步 history_2016 与 history_2017 数据库
## 目标库需要为 order ，目标数据表为 order_2017 / order_2016

## Syncer 获取到上游数据后，发现 route-rules 规则启用，先做合库合表操作，再进行 do-db & do-table
    ↪ 判定
### 此处需要设置 target-schema & target-table 判定需要同步的数据库
[[replicate-do-table]]
db-name ="order"
tbl-name = "order_2016"

[[replicate-do-table]]
db-name ="order"
tbl-name = "order_2017"

[[route-rules]]
pattern-schema = "order_2016"
pattern-table = "2016_??"
target-schema = "order"
target-table = "order_2016"

[[route-rules]]
pattern-schema = "order_2017"
pattern-table = "2017_??"
target-schema = "order"
target-table = "order_2017"
```

11.15.5.3 Syncer 同步前检查

1. 检查数据库版本。

   使用 `select @@version;` 命令检查数据库版本。目前，Syncer 只支持以下版本：

   - 5.5 < MySQL 版本 < 8.0
   - MariaDB 版本 >= 10.1.2（更早版本的 binlog 部分字段类型格式与 MySQL 不一致）

   > **注意：**
   >
   > 如果上游 MySQL/MariaDB server 间构成主从复制结构，则
   >
   > - 5.7.1 < MySQL 版本 < 8.0
   > - MariaDB 版本 >= 10.1.3

2. 检查源库 `server-id`。

   可通过以下命令查看 `server-id`：

```
show global variables like 'server_id';
```

```
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| server_id     | 1     |
+---------------+-------+
1 row in set (0.01 sec)
```

- 结果为空或者为 0，Syncer 无法同步数据。
- Syncer `server-id` 与 MySQL `server-id` 不能相同，且必须在 MySQL cluster 中唯一。

3. 检查 Binlog 相关参数。

    1. 检查 MySQL 是否开启了 binlog。

       使用如下命令确认是否开启了 binlog：

       ```
       show global variables like 'log_bin';
       ```

       ```
       +--------------------+---------+
       | Variable_name      | Value   |
       +--------------------+---------+
       | log_bin            | ON      |
       +--------------------+---------+
       1 row in set (0.00 sec)
       ```

       如果结果是 `log_bin = OFF`，则需要开启 binlog，开启方式请参考官方文档。

    2. binlog 格式必须为 `ROW`，且参数 `binlog_row_image` 必须设置为 `FULL`，可使用如下命令查看参数设置：

       ```
       select variable_name, variable_value from information_schema.global_variables where
           ↪ variable_name in ('binlog_format','binlog_row_image');
       ```

       ```
       +------------------+----------------+
       | variable_name    | variable_value |
       +------------------+----------------+
       | BINLOG_FORMAT    | ROW            |
       | BINLOG_ROW_IMAGE | FULL           |
       +------------------+----------------+
       2 rows in set (0.001 sec)
       ```

       - 如果以上设置出现错误，则需要修改磁盘上的配置文件，然后重启 MySQL。
       - 将配置的更改持久化存储在磁盘上很重要，这样在 MySQL 重启之后才能显示相应更改。
       - 由于现有的连接会保留全局变量原先的值，所以不可以使用 `SET` 语句动态修改这些设置。

4. 检查用户权限。

1. 全量导出的 Mydumper 需要的用户权限。

    - Mydumper 导出数据至少拥有以下权限：`select`, `reload`。
    - Mydumper 操作对象为 RDS 时，可以添加 `--no-locks` 参数，避免申请 `reload` 权限。

2. 增量同步 Syncer 需要的上游 MySQL/MariaDB 用户权限。

    需要上游 MySQL 同步账号至少赋予以下权限：

    ```
    select, replication slave, replication client
    ```

3. 下游 TiDB 需要的权限

权限	作用域
SELECT	Tables
INSERT	Tables
UPDATE	Tables
DELETE	Tables
CREATE	Databases,tables
DROP	Databases, tables
ALTER	Tables
INDEX	Tables

    为所同步的数据库或者表，执行下面的 GRANT 语句：

    ```
    GRANT SELECT,INSERT,UPDATE,DELETE,CREATE,DROP,ALTER,INDEX  ON db.table TO 'your_user'@'
        ↪ your_wildcard_of_host';
    ```

5. 检查 SQL mode。

    必须确认上下游的 SQL mode 一致；如果不一致，则会出现数据同步的错误。

    ```
    show variables like '%sql_mode%';
    ```

    ```
    +---------------+--------------------------------------------------------------------------------------------+
        ↪
    | Variable_name | Value
        ↪                                                                                                    |
    +---------------+--------------------------------------------------------------------------------------------+
        ↪
    | sql_mode      | ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_AUTO_CREATE_USER,
        ↪ NO_ENGINE_SUBSTITUTION |
    +---------------+--------------------------------------------------------------------------------------------+
        ↪
    1 row in set (0.01 sec)
    ```

6. 检查字符集。

    TiDB 和 MySQL 的字符集的兼容性不同，详见TiDB 支持的字符集。

7. 检查同步的表是否都有主键或者唯一索引。

   如果表没有主键或唯一索引，就没法实现幂等，并且此时在下游更新每条数据时都是扫全表，可能导致同步速度变慢，所以建议同步的表都加上主键。

11.15.6 监控方案

Syncer 使用开源时序数据库 Prometheus 作为监控和性能指标信息存储方案，使用 Grafana 作为可视化组件进行展示，配合 AlertManager 来实现报警。其方案如下图所示：

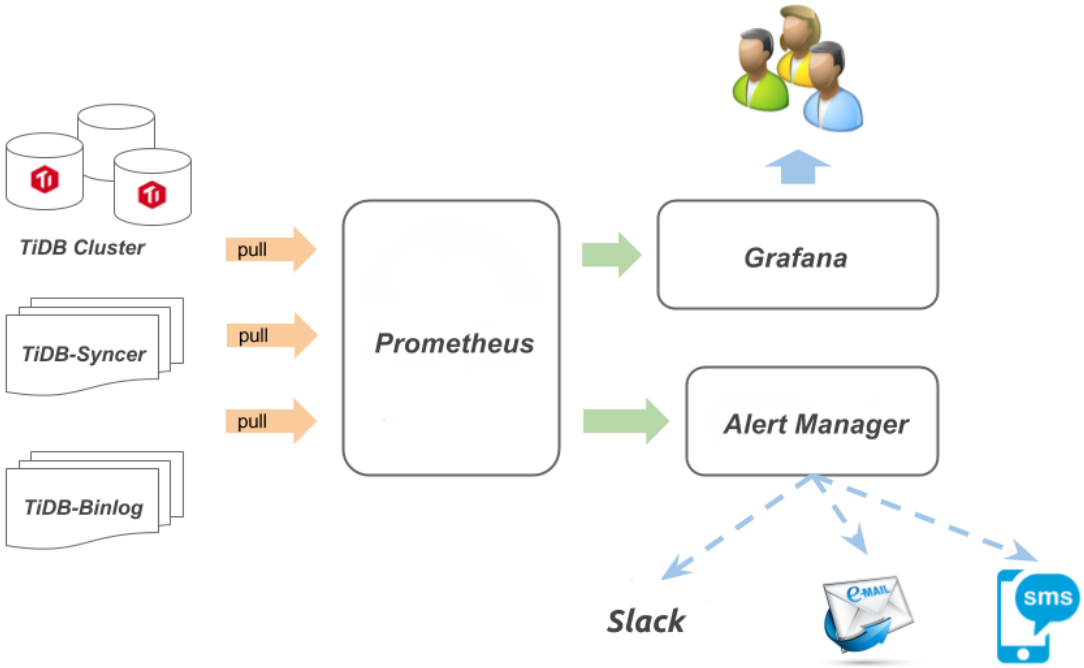


图 186: monitor_scheme

11.15.6.1 配置 Syncer 监控与告警

Syncer 对外提供 metric 接口，需要 Prometheus 主动获取数据。配置 Syncer 监控与告警的操作步骤如下：

1. 在 Prometheus 中添加 Syncer job 信息，将以下内容刷新到 Prometheus 配置文件，重启 Prometheus 后生效。

```
  - job_name: 'syncer_ops' // 任务名字，区分数据上报
    static_configs:
```

```
      - targets: ['10.1.1.4:10086'] // Syncer 监听地址与端口，通知 Prometheus 获取 Syncer
          ↪ 的监控数据。
```

2. 配置 Prometheus 告警，将以下内容刷新到 `alert.rule` 配置文件，重启 Prometheus 后生效。

```
# syncer
ALERT syncer_status
  IF  syncer_binlog_file{node='master'} - ON(instance, job) syncer_binlog_file{node='syncer
      ↪ '} > 1
  FOR 1m
  LABELS {channels="alerts", env="test-cluster"}
  ANNOTATIONS {
  summary = "syncer status error",
  description="alert: syncer_binlog_file{node='master'} - ON(instance, job)
      ↪ syncer_binlog_file{node='syncer'} > 1 instance: {{     $labels.instance }} values:
      ↪  {{ $value }}",
  }
```

11.15.6.1.1 Grafana 配置

- 进入 Grafana Web 界面（默认地址: `http://localhost:3000`，默认账号: admin 密码: admin）
- 导入 dashboard 配置文件

  点击 Grafana Logo -> 点击 Dashboards -> 点击 Import -> 选择需要的 Dashboard 配置文件上传 -> 选择对应的 data source

11.15.6.2 Grafana Syncer metrics 说明

11.15.6.2.1 title: binlog events

- metrics: `rate(syncer_binlog_event_count[1m])`
- info: 统计 Syncer 每秒已经收到的 binlog 个数

11.15.6.2.2 title: binlog event transform

- metrics: `histogram_quantile(0.8, sum(rate(syncer_binlog_event_bucket[1m]))by (le))`
- info: Syncer 把 binlog 转换为 SQL 语句的耗时

11.15.6.2.3 title: transaction latency

- metrics: `histogram_quantile(0.95, sum(rate(syncer_txn_cost_in_second_bucket[1m]))by (le))`
- info: Syncer 在下游 TiDB 执行 transaction 的耗时

11.15.6.2.4 title: transaction tps

- metrics: `rate(syncer_txn_cost_in_second_count[1m])`
- info: Syncer 在下游 TiDB 每秒执行的 transaction 个数

11.15.6.2.5 title: binlog file gap

- metrics: `syncer_binlog_file{node="master"} - ON(instance, job)syncer_binlog_file{node="syncer"}`
- info: Syncer 已经同步到的 binlog position 的文件编号距离上游 MySQL 当前 binlog position 的文件编号的值；注意 MySQL 当前 binlog position 是定期查询，在一些情况下该 metrics 会出现负数的情况

11.15.6.2.6 title: binlog skipped events

- metrics: `rate(syncer_binlog_skipped_events_total[1m])`
- info: Syncer 跳过的 binlog 的个数，你可以在配置文件中配置 `skip-ddls` 和 `skip-dmls` 来跳过指定的 binlog

11.15.6.2.7 title: position of binlog position

- metrics: `syncer_binlog_pos{node="syncer"}` and `syncer_binlog_pos{node="master"}`
- info: 需配合 `file number of binlog position` 一起看。`syncer_binlog_pos{node="master"}` 表示上游 MySQL 当前 binlog position 的 position 值，`syncer_binlog_pos{node="syncer"}` 表示上游 Syncer 已经同步到的 binlog position 的 position 值

11.15.6.2.8 title: file number of binlog position

- metrics: `syncer_binlog_file{node="syncer"}` and `syncer_binlog_file{node="master"}`
- info: 需要配置 `position of binlog position` 一起看。`syncer_binlog_file{node="master"}` 表示上游 MySQL 当前 binlog position 的文件编号，`syncer_binlog_file{node="syncer"}` 表示上游 Syncer 已经同步到的 binlog 位置的文件编号

11.15.6.2.9 title: execution jobs

- metrics: `sum(rate(syncer_add_jobs_total[1m]))by (queueNo)`
- info: Syncer 把 binlog 转换成 SQL 语句后，将 SQL 语句以 jobs 的方式加到执行队列中，这个 metrics 表示已经加入执行队列的 jobs 总数

11.15.6.2.10 title: pending jobs

- metrics: `sum(rate(syncer_add_jobs_total[1m])- rate(syncer_finished_jobs_total[1m]))by (queueNo)`
- info: 已经加入执行队列但是还没有执行的 jobs 数量

11.16 TiSpark

11.16.1 TiSpark 快速上手

为了让大家快速体验TiSpark，通过 TiDB Ansible 安装的 TiDB 集群中默认已集成 Spark、TiSpark jar 包及 TiSpark sample data。

11.16.1.1 部署信息

- Spark 默认部署在 TiDB 实例部署目录下 spark 目录中
- TiSpark jar 包默认部署在 Spark 部署目录 jars 文件夹下：`spark/jars/tispark-${name_with_version}.jar`
- TiSpark 示例数据和导入脚本可点击 TiSpark 示例数据下载。

```
tispark-sample-data/
```

11.16.1.2 环境准备

11.16.1.2.1 在 TiDB 实例上安装 JDK

在 Oracle JDK 官方下载页面 下载 JDK 1.8 当前最新版，本示例中下载的版本为 `jdk-8u141-linux-x64.tar.gz`。

解压并根据您的 JDK 部署目录设置环境变量，编辑 `~/.bashrc` 文件，比如：

```
export JAVA_HOME=/home/pingcap/jdk1.8.0_144 &&
export PATH=$JAVA_HOME/bin:$PATH
```

验证 JDK 有效性：

```
java -version
```

```
java version "1.8.0_144"
Java(TM) SE Runtime Environment (build 1.8.0_144-b01)
Java HotSpot(TM) 64-Bit Server VM (build 25.144-b01, mixed mode)
```

11.16.1.2.2 导入样例数据

假设 TiDB 集群已启动，其中一台 TiDB 实例服务 IP 为 192.168.0.2，端口为 4000，用户名为 root, 密码为空。

```
wget http://download.pingcap.org/tispark-sample-data.tar.gz && \
tar -zxvf tispark-sample-data.tar.gz && \
cd tispark-sample-data
```

修改 `sample_data.sh` 中 TiDB 登录信息，比如：

```
mysql --local-infile=1 -h 192.168.0.2 -P 4000 -u root < dss.ddl
```

执行脚本

```
./sample_data.sh
```

> **注意：**
>
> 执行脚本的机器上需要安装 MySQL client，CentOS 用户可通过 `yum -y install mysql`来安装。

登录 TiDB 并验证数据包含 TPCH_001 库及以下表：

```
mysql -uroot -P4000 -h192.168.0.2
```

```
show databases;
```

```
+--------------------+
| Database           |
+--------------------+
| INFORMATION_SCHEMA |
| PERFORMANCE_SCHEMA |
| TPCH_001           |
| mysql              |
| test               |
+--------------------+
5 rows in set (0.00 sec)
```

```
use TPCH_001;
```

```
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
```

```
show tables;
```

```
+--------------------+
| Tables_in_TPCH_001 |
+--------------------+
| CUSTOMER           |
| LINEITEM           |
| NATION             |
| ORDERS             |
| PART               |
| PARTSUPP           |
| REGION             |
```

```
| SUPPLIER           |
+--------------------+
8 rows in set (0.00 sec)
```

11.16.1.3 使用范例

进入 spark 部署目录启动 spark-shell:

```
cd spark &&
bin/spark-shell
```

然后像使用原生 Spark 一样查询 TiDB 表：

```
scala> spark.sql("select count(*) from lineitem").show
```

结果为

```
+--------+
|count(1)|
+--------+
|   60175|
+--------+
```

下面执行另一个复杂一点的 Spark SQL:

```
scala> spark.sql(
      """select
        |   l_returnflag,
        |   l_linestatus,
        |   sum(l_quantity) as sum_qty,
        |   sum(l_extendedprice) as sum_base_price,
        |   sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
        |   sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
        |   avg(l_quantity) as avg_qty,
        |   avg(l_extendedprice) as avg_price,
        |   avg(l_discount) as avg_disc,
        |   count(*) as count_order
        |from
        |   lineitem
        |where
        |   l_shipdate <= date '1998-12-01' - interval '90' day
        |group by
        |   l_returnflag,
        |   l_linestatus
        |order by
        |   l_returnflag,
        |   l_linestatus
```

```
      """.stripMargin).show
```

结果为：

```
+------------+------------+---------+--------------+--------------+
|l_returnflag|l_linestatus|  sum_qty|sum_base_price|sum_disc_price|
+------------+------------+---------+--------------+--------------+
|           A|           F|380456.00|  532348211.65|505822441.4861|
|           N|           F|  8971.00|   12384801.37| 11798257.2080|
|           N|           O|742802.00| 1041502841.45|989737518.6346|
|           R|           F|381449.00|  534594445.35|507996454.4067|
+------------+------------+---------+--------------+--------------+
(续)
-----------------+---------+------------+--------+-----------+
       sum_charge|  avg_qty|   avg_price|avg_disc|count_order|
-----------------+---------+------------+--------+-----------+
 526165934.000839|25.575155|35785.709307|0.050081|      14876|
  12282485.056933|25.778736|35588.509684|0.047759|        348|
1029418531.523350|25.454988|35691.129209|0.049931|      29181|
 528524219.358903|25.597168|35874.006533|0.049828|      14902|
-----------------+---------+------------+--------+-----------+
```

更多样例请参考 pingcap/tispark-test

11.16.2 TiSpark 用户指南

TiSpark 是 PingCAP 为解决用户复杂 OLAP 需求而推出的产品。它借助 Spark 平台，同时融合 TiKV 分布式集群的优势，和 TiDB 一起为用户一站式解决 HTAP (Hybrid Transactional/Analytical Processing) 的需求。TiSpark 依赖于 TiKV 集群和 Placement Driver (PD)，也需要你搭建一个 Spark 集群。

本文简单介绍如何部署和使用 TiSpark。本文假设你对 Spark 有基本认知。你可以参阅 Apache Spark 官网 了解 Spark 的相关信息。

11.16.2.1 概述

TiSpark 是将 Spark SQL 直接运行在分布式存储引擎 TiKV 上的 OLAP 解决方案。其架构图如下：

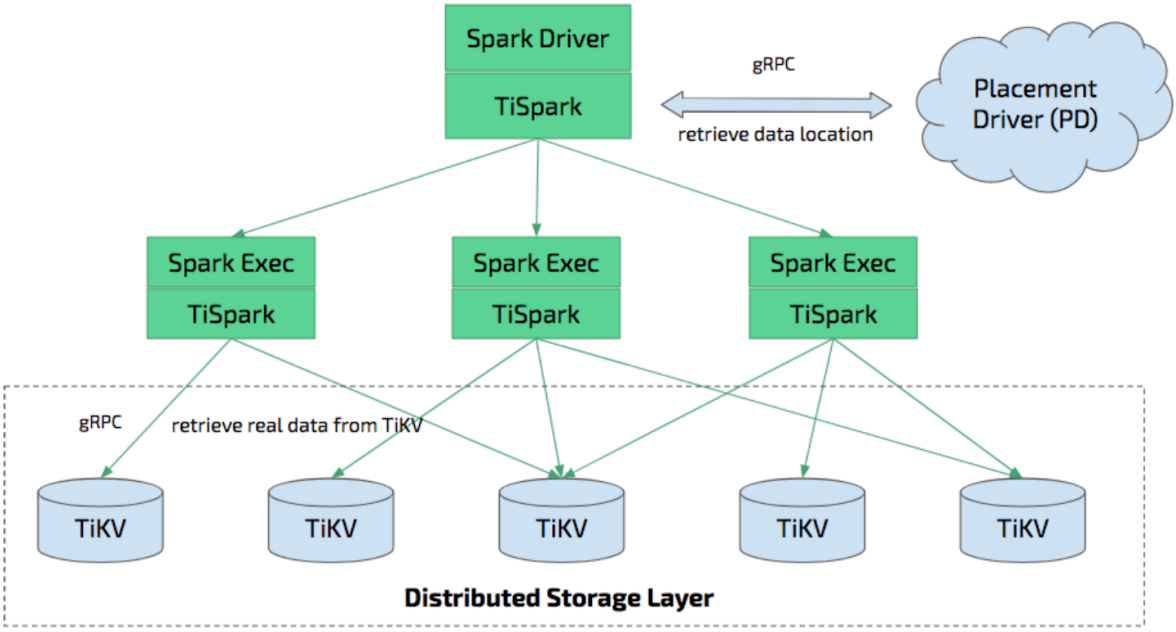


图 187: TiSpark Architecture

- TiSpark 深度整合了 Spark Catalyst 引擎, 可以对计算提供精确的控制，使 Spark 能够高效的读取 TiKV 中的数据，提供索引支持以实现高速的点查。
- 通过多种计算下推减少 Spark SQL 需要处理的数据大小，以加速查询；利用 TiDB 的内建的统计信息选择更优的查询计划。
- 从数据集群的角度看，TiSpark + TiDB 可以让用户无需进行脆弱和难以维护的 ETL，直接在同一个平台进行事务和分析两种工作，简化了系统架构和运维。
- 用户借助 TiSpark 项目可以在 TiDB 上使用 Spark 生态圈提供的多种工具进行数据处理。例如，使用 TiSpark 进行数据分析和 ETL；使用 TiKV 作为机器学习的数据源；借助调度系统产生定时报表等等。
- 除此之外，TiSpark 还提供了分布式写入 TiKV 的功能。相比使用 Spark 结合 JDBC 的方式写入 TiDB，分布式写入 TiKV 可以实现事务（要么全部数据写入成功，要么全部都写入失败），并且写入速度会更快。

11.16.2.2 环境准备

现有 TiSpark 2.x 版本支持 Spark 2.3.x 和 Spark 2.4.x。如果你希望使用 Spark 2.1.x 版本，需使用 TiSpark 1.x。

TiSpark 需要 JDK 1.8+ 以及 Scala 2.11（Spark2.0+ 默认 Scala 版本）。

TiSpark 可以在 YARN，Mesos，Standalone 等任意 Spark 模式下运行。

11.16.2.3 推荐配置

本部分描述了 TiKV 与 TiSpark 集群分开部署、Spark 与 TiSpark 集群独立部署，以及 TiSpark 与 TiKV 集群混合部署的建议配置。

11.16.2.3.1 TiKV 与 TiSpark 集群分开部署的配置

对于 TiKV 与 TiSpark 分开部署的场景，可以参考如下建议配置：

- 硬件配置建议

  普通场景可以参考TiDB 和 TiKV 硬件配置建议，但是如果是偏重分析的场景，可以将 TiKV 节点增加到至少 64G 内存。

11.16.2.3.2 Spark 与 TiSpark 集群独立部署的配置

关于 Spark 的详细硬件推荐配置请参考官网，如下是 TiSpark 所需环境的简单描述：

Spark 推荐 32G 内存以上的配额。请在配置中预留 25% 的内存给操作系统。

Spark 推荐每台计算节点配备 CPU 累计 8 到 16 核以上。你可以初始设定分配所有 CPU 核给 Spark。

Spark 的具体配置方式也请参考官方说明。以下为根据 `spark-env.sh` 配置的范例：

```
SPARK_EXECUTOR_CORES: 5
SPARK_EXECUTOR_MEMORY: 10g
SPARK_WORKER_CORES: 5
SPARK_WORKER_MEMORY: 10g
```

在 `spark-defaults.conf` 中，增加如下配置：

```
spark.tispark.pd.addresses $your_pd_servers
spark.sql.extensions org.apache.spark.sql.TiExtensions
```

在 CDH spark 版本中添加如下配置：

```
spark.tispark.pd.addresses=$your_pd_servers
spark.sql.extensions=org.apache.spark.sql.TiExtensions
```

`your_pd_servers` 是用逗号分隔的 PD 地址，每个地址使用 地址:端口 的格式。

例如你有一组 PD 在`10.16.20.1,10.16.20.2,10.16.20.3`,那么 PD 配置格式是`10.16.20.1:2379,10.16.20.2:2379,10.16.20.3:23`
↪ 。

11.16.2.3.3 TiSpark 与 TiKV 集群混合部署的配置

对于 TiKV 与 TiSpark 混合部署的场景，需在原有 TiKV 预留资源之外累加 Spark 所需部分，并分配 25% 的内存作为系统本身占用。

11.16.2.4 部署 TiSpark

TiSpark 的 jar 包可以在 TiSpark Releases 页面下载对应版本的 jar 包并拷贝到合适的目录。

11.16.2.4.1 已有 Spark 集群的部署方式

如果在已有 Spark 集群上运行 TiSpark，无需重启集群。可以使用 Spark 的 `--jars` 参数将 TiSpark 作为依赖引入：

```
spark-shell --jars $TISPARK_FOLDER/tispark-${name_with_version}.jar
```

11.16.2.4.2 没有 Spark 集群的部署方式

如果没有使用中的 Spark 集群，推荐使用 Saprk Standalone 方式部署。这里简单介绍下 Standalone 部署方式。如果遇到问题，可以去官网寻求帮助；也欢迎在我们的 GitHub 上提 issue。

如果是使用 TiDB Ansible 部署的 TiDB 集群，也可以通过 TiDB Ansible 来部署 Spark Standalone 集群，TiSpark 也会同时部署。

下载安装包并安装

你可以在 Download Apache Spark™ 页面下载 Apache Spark。

对于 Standalone 模式且无需 Hadoop 支持，则选择 Spark 2.3.x 或者 Spark 2.4.x 且带有 Hadoop 依赖的 Pre-build with Apache Hadoop 2.x 任意版本。如有需要配合使用的 Hadoop 集群，则选择对应的 Hadoop 版本号。你也可以选择从源代码自行构建以配合官方 Hadoop 2.x 之前的版本。

如果你已经有了 Spark 二进制文件，并且当前 PATH 为 SPARKPATH，需将 TiSpark jar 包拷贝到 `${SPARKPATH}/jars` 目录下。

启动 Master

在选中的 Spark Master 节点执行如下命令：

```
cd $SPARKPATH
```

```
./sbin/start-master.sh
```

在这步完成以后，屏幕上会打印出一个 log 文件。检查 log 文件确认 Spark-Master 是否启动成功。你可以打开 http://spark-master-hostname:8080 查看集群信息（如果你没有改动 Spark-Master 默认 Port Numebr）。在启动 Spark-Worker 的时候，也可以通过这个面板来确认 Worker 是否已经加入集群。

启动 Worker

类似地，可以用如下命令启动 Spark-Worker 节点：

```
./sbin/start-slave.sh spark://spark-master-hostname:7077
```

命令返回以后，即可通过刚才的面板查看这个 Worker 是否已经正确地加入了 Spark 集群。在所有 Worker 节点重复刚才的命令。确认所有的 Worker 都可以正确连接 Master，这样你就拥有了一个 Standalone 模式的 Spark 集群。

Spark SQL shell 和 JDBC 服务器

当前版本的 TiSpark 可以直接使用 `spark-sql` 和 Spark 的 ThriftServer JDBC 服务器。

11.16.2.5 一个使用范例

假设你已经按照上述步骤成功启动了 TiSpark 集群，下面简单介绍如何使用 Spark SQL 来做 OLAP 分析。这里我们用名为 tpch 数据库中的 lineitem 表作为范例。

假设你的 PD 节点位于 192.168.1.100，端口为 2379，在 `$SPARK_HOME/conf/spark-defaults.conf` 加入：

```
spark.tispark.pd.addresses 192.168.1.100:2379
```

```
spark.sql.extensions org.apache.spark.sql.TiExtensions
```

然后在 Spark-Shell 里像原生 Spark 一样输入下面的命令：

```
spark.sql("use tpch")
```

```
spark.sql("select count(*) from lineitem").show
```

结果为：

```
+-------------+
| Count (1)   |
+-------------+
| 600000000   |
+-------------+
```

Spark SQL 交互 Shell 和原生 Spark 一致：

```
spark-sql> use tpch;
```

```
Time taken: 0.015 seconds
```

```
spark-sql> select count(*) from lineitem;
```

```
2000
Time taken: 0.673 seconds, Fetched 1 row(s)
```

SQuirreLSQL 和 hive-beeline 可以使用 JDBC 连接 Thrift 服务器。例如，使用 beeline 连接：

```
./beeline
```

```
Beeline version 1.2.2 by Apache Hive
```

```
beeline> !connect jdbc:hive2://localhost:10000
```

```
1: jdbc:hive2://localhost:10000> use testdb;
```

```
+---------+--+
| Result  |
+---------+--+
+---------+--+
No rows selected (0.013 seconds)
```

```
select count(*) from account;
```

```
+-----------+--+
| count(1)  |
+-----------+--+
| 1000000   |
+-----------+--+
1 row selected (1.97 seconds)
```

11.16.2.6 和 Hive 一起使用 TiSpark

TiSpark 可以和 Hive 混合使用。在启动 Spark 之前，需要添加 HADOOP_CONF_DIR 环境变量指向 Hadoop 配置目录并且将 `hive-site.xml` 拷贝到 `$SPARK_HOME/conf` 目录下。

```
val tisparkDF = spark.sql("select * from tispark_table").toDF
tisparkDF.write.saveAsTable("hive_table") // save table to hive
spark.sql("select * from hive_table a, tispark_table b where a.col1 = b.col1").show // join table
    ↪  across Hive and Tispark
```

11.16.2.7 通过 TiSpark 将 DataFrame 批量写入 TiDB

TiSpark 从 v2.3 版本开始原生支持将 DataFrame 批量写入 TiDB 集群，该写入模式通过 TiKV 的两阶段提交协议实现。

TiSpark 批量写入相比 Spark + JDBC 写入，有以下特点：

比较的方面	TiSpark 批量写入	Spark + JDBC 写入
原子性	DataFrame 的数据要么全部写入成功，要么全部写入失败	如果在写入过程中 spark 任务失败退出，会出现部分数据写入成功的情况
隔离性	写入过程中其他事务对正在写入的数据不可见	写入过程中其他事务能看到部分写入成功的数据
错误恢复	失败后只需要重新运行 Spark 程序	需要业务来实现幂等，例如失败后需要先清理部分写入成功的数据，再重新运行 Spark 程序，并且需要设置 `spark.task.maxFailures=1`，防止 task 内重试导致数据重复
速度	直接写入 TiKV，速度更快	通过 TiDB 再写入 TiKV，对速度会有影响

以下通过 scala API 演示如何使用 TiSpark 批量写入：

```
// select data to write
val df = spark.sql("select * from tpch.ORDERS")

// write data to tidb
df.write.
  format("tidb").
  option("tidb.addr", "127.0.0.1").
  option("tidb.port", "4000").
  option("tidb.user", "root").
  option("tidb.password", "").
  option("database", "tpch").
  option("table", "target_orders").
  mode("append").
  save()
```

如果写入的数据量比较大，且写入时间超过 10 分钟，则需要保证 GC 时间大于写入时间。

```
update mysql.tidb set VARIABLE_VALUE="6h" where VARIABLE_NAME="tikv_gc_life_time";
```

详细使用手册请参考该文档。

11.16.2.8 通过 JDBC 将 DataFrame 写入 TiDB

除了使用 TiSpark 将 DataFrame 批量写入 TiDB 集群以外，也可以使用 Spark 原生的 JDBC 支持进行写入：

```
import org.apache.spark.sql.execution.datasources.jdbc.JDBCOptions

val customer = spark.sql("select * from customer limit 100000")
// you might repartition source to make it balance across nodes
// and increase concurrency
val df = customer.repartition(32)
df.write
.mode(saveMode = "append")
.format("jdbc")
.option("driver", "com.mysql.jdbc.Driver")
 // replace host and port as your and be sure to use rewrite batch
.option("url", "jdbc:mysql://127.0.0.1:4000/test?rewriteBatchedStatements=true")
.option("useSSL", "false")
// As tested, 150 is good practice
.option(JDBCOptions.JDBC_BATCH_INSERT_SIZE, 150)
.option("dbtable", s"cust_test_select") // database name and table name here
.option("isolationLevel", "NONE") // recommended to set isolationLevel to NONE if you have a
    ↪ large DF to load.
.option("user", "root") // TiDB user here
.save()
```

推荐将 `isolationLevel` 设置为 `NONE`，否则单一大事务有可能造成 TiDB 服务器内存溢出。

> **注意：**
>
> TiSpark 使用 JDBC 时默认 `isolationLevel` 为 `READ_UNCOMMITTED`，会造成事务隔离级别不支持的错误。推荐将 `isolationLevel` 设置为 `NONE`。

11.16.2.9 统计信息

TiSpark 可以使用 TiDB 的统计信息：

1. 选择代价最低的索引或扫表访问
2. 估算数据大小以决定是否进行广播优化

如果希望使用统计信息支持，需要确保所涉及的表已经被分析。请阅读这份文档了解如何进行表分析。

从 TiSpark 2.0 开始，统计信息将会默认被读取。

11.16.2.10 TiSpark FAQ

- Q. 是独立部署还是和现有 Spark / Hadoop 集群共用资源？

  A. 可以利用现有 Spark 集群无需单独部署，但是如果现有集群繁忙，TiSpark 将无法达到理想速度。

- Q. 是否可以和 TiKV 混合部署？

  A. 如果 TiDB 以及 TiKV 负载较高且运行关键的线上任务，请考虑单独部署 TiSpark；并且考虑使用不同的网卡保证 OLTP 的网络资源不被侵占而影响线上业务。如果线上业务要求不高或者机器负载不大，可以考虑与 TiKV 混合部署。

- Q. Spark 执行中报 warning：WARN ObjectStore:568 - Failed to get database

  A. Warning 忽略即可，原因是 Spark 找不到对应的 hive 库，因为这个库是在 TIKV 中，而不是在 hive 中。可以考虑调整 log4j 日志，将该参数添加到 spark 下 conf 里 log4j 文件 (如果后缀是 template 那先 mv 成后缀 properties)。

- Q. Spark 执行中报 java.sql.BatchUpdateException: Data Truncated

  A. 写入的数据长度超过了数据库定义的数据类型的长度，可以确认 target table 的字段长度，进行调整。

- Q. TiSpark 任务是否默认读取 Hive 的元数据？

  A. TiSpark 通过读取 hive-site 里的 meta 来搜寻 hive 的库。如果搜寻不到，就通过读取 tidb meta 搜寻 tidb 库。如果不需要该行为，可不在 hive site 中配置 hive 的 meta。

- Q. TiSpark 执行 Spark 任务时报：Error：java.io.InvalidClassException: com.pingcap.tikv.region.TiRegion; local class incompatible: stream classdesc serialVersionUID ···

  A. 该报错日志中显示 serialVersionUID 冲突，说明存在不同版本的 class 和 TiRegion。因为 TiRegion 是 TiSpark 独有的，所以可能存在多个版本的 TiSpark 包。要解决该报错，请确保集群中各节点的 TiSpark 依赖包版本一致。

12 参考指南

12.1 架构

12.1.1 TiDB 整体架构

与传统的单机数据库相比，TiDB 具有以下优势：

- 纯分布式架构，拥有良好的扩展性，支持弹性的扩缩容
- 支持 SQL，对外暴露 MySQL 的网络协议，并兼容大多数 MySQL 的语法，在大多数场景下可以直接替换 MySQL
- 默认支持高可用，在少数副本失效的情况下，数据库本身能够自动进行数据修复和故障转移，对业务透明
- 支持 ACID 事务，对于一些有强一致需求的场景友好，例如：银行转账
- 具有丰富的工具链生态，覆盖数据迁移、同步、备份等多种场景

在内核设计上，TiDB 分布式数据库将整体架构拆分成了多个模块，各模块之间互相通信，组成完整的 TiDB 系统。对应的架构图如下：

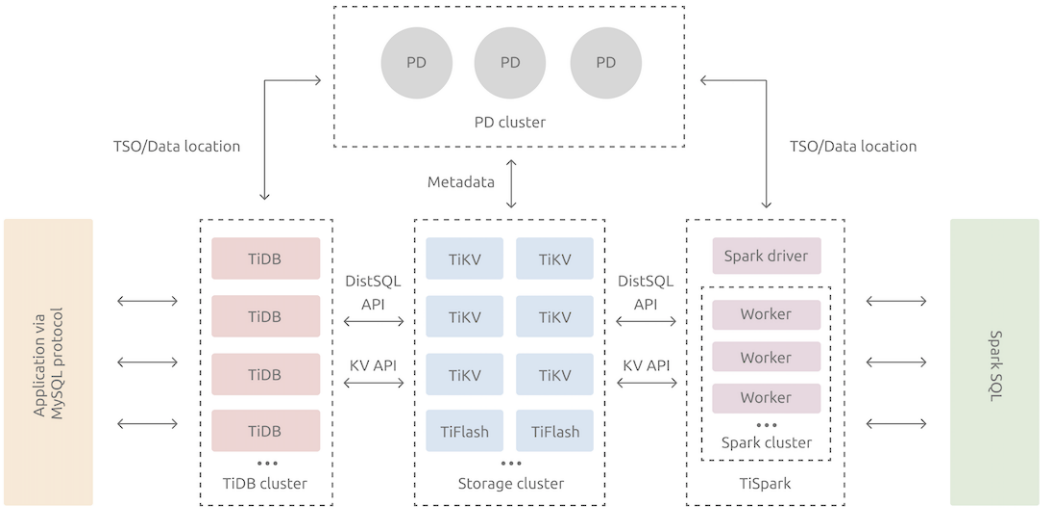


图 188: architecture

- TiDB Server：SQL 层，对外暴露 MySQL 协议的连接 endpoint，负责接受客户端的连接，执行 SQL 解析和优化，最终生成分布式执行计划。TiDB 层本身是无状态的，实践中可以启动多个 TiDB 实例，通过负载均衡组件（如 LVS、HAProxy 或 F5）对外提供统一的接入地址，客户端的连接可以均匀地分摊在多个 TiDB 实例上以达到负载均衡的效果。TiDB Server 本身并不存储数据，只是解析 SQL，将实际的数据读取请求转发给底层的存储节点 TiKV（或 TiFlash）。

- PD (Placement Driver) Server：整个 TiDB 集群的元信息管理模块，负责存储每个 TiKV 节点实时的数据分布情况和集群的整体拓扑结构，提供 TiDB Dashboard 管控界面，并为分布式事务分配事务 ID。PD 不仅存储元信息，同时还会根据 TiKV 节点实时上报的数据分布状态，下发数据调度命令给具体的 TiKV 节点，可以说是整个集群的“大脑”。此外，PD 本身也是由至少 3 个节点构成，拥有高可用的能力。建议部署奇数个 PD 节点。
- 存储节点
  - TiKV Server：负责存储数据，从外部看 TiKV 是一个分布式的提供事务的 Key-Value 存储引擎。存储数据的基本单位是 Region，每个 Region 负责存储一个 Key Range（从 StartKey 到 EndKey 的左闭右开区间）的数据，每个 TiKV 节点会负责多个 Region。TiKV 的 API 在 KV 键值对层面提供对分布式事务的原生支持，默认提供了 SI (Snapshot Isolation) 的隔离级别，这也是 TiDB 在 SQL 层面支持分布式事务的核心。TiDB 的 SQL 层做完 SQL 解析后，会将 SQL 的执行计划转换为对 TiKV API 的实际调用。所以，数据都存储在 TiKV 中。另外，TiKV 中的数据都会自动维护多副本（默认为三副本），天然支持高可用和自动故障转移。
  - TiFlash：TiFlash 是一类特殊的存储节点。和普通 TiKV 节点不一样的是，在 TiFlash 内部，数据是以列式的形式进行存储，主要的功能是为分析型的场景加速。

12.1.2 TiDB 数据库的存储

本文主要介绍 TiKV 的一些设计思想和关键概念。

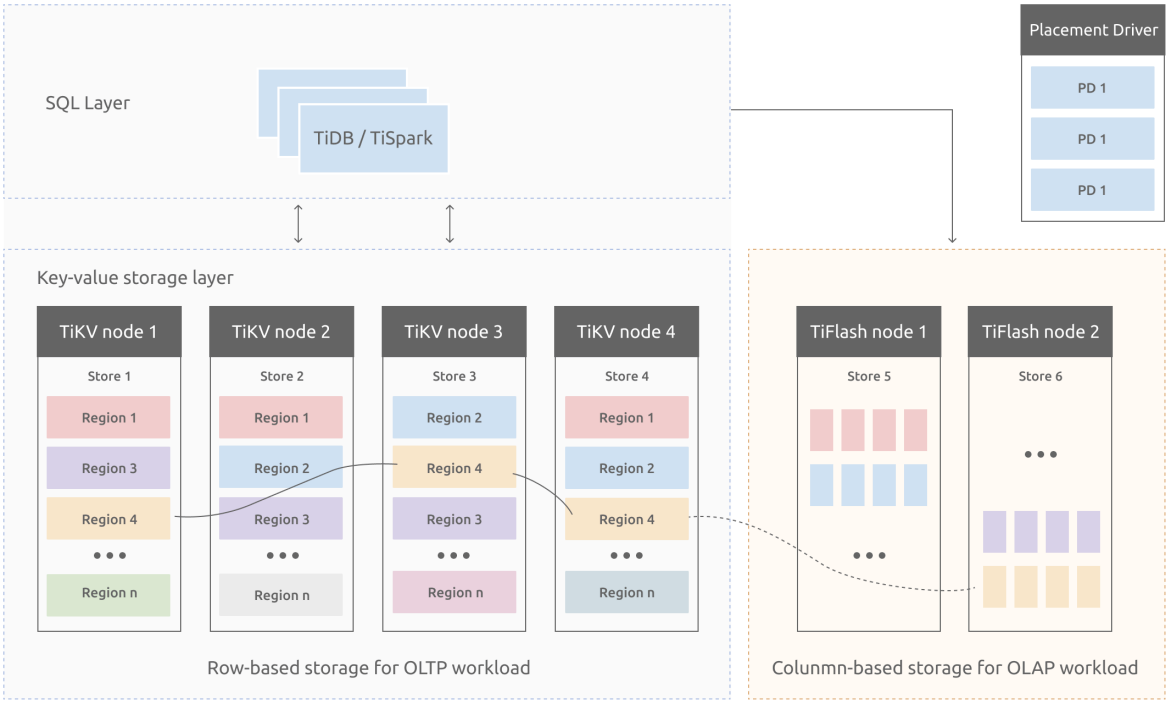


图 189: storage-architecture

12.1.2.1 Key-Value Pairs（键值对）

作为保存数据的系统，首先要决定的是数据的存储模型，也就是数据以什么样的形式保存下来。TiKV 的选择是 Key-Value 模型，并且提供有序遍历方法。

TiKV 数据存储的两个关键点：

1. 这是一个巨大的 Map（可以类比一下 C++ 的 std::map），也就是存储的是 Key-Value Pairs（键值对）
2. 这个 Map 中的 Key-Value pair 按照 Key 的二进制顺序有序，也就是可以 Seek 到某一个 Key 的位置，然后不断地调用 Next 方法以递增的顺序获取比这个 Key 大的 Key-Value。

注意，本文所说的 TiKV 的 KV 存储模型和 SQL 中的 Table 无关。本文不讨论 SQL 中的任何概念，专注于讨论如何实现 TiKV 这样一个高性能、高可靠性、分布式的 Key-Value 存储。

12.1.2.2 本地存储 (RocksDB)

任何持久化的存储引擎，数据终归要保存在磁盘上，TiKV 也不例外。但是 TiKV 没有选择直接向磁盘上写数据，而是把数据保存在 RocksDB 中，具体的数据落地由 RocksDB 负责。这个选择的原因是开发一个单机存储引擎工作量很大，特别是要做一个高性能的单机引擎，需要做各种细致的优化，而 RocksDB 是由 Facebook 开源的一个非常优秀的单机 KV 存储引擎，可以满足 TiKV 对单机引擎的各种要求。这里可以简单的认为 RocksDB 是一个单机的持久化 Key-Value Map。

12.1.2.3 Raft 协议

接下来 TiKV 的实现面临一件更难的事情：如何保证单机失效的情况下，数据不丢失，不出错？

简单来说，需要想办法把数据复制到多台机器上，这样一台机器无法服务了，其他的机器上的副本还能提供服务；复杂来说，还需要这个数据复制方案是可靠和高效的，并且能处理副本失效的情况。TiKV 选择了 Raft 算法。Raft 是一个一致性协议，本文只会对 Raft 做一个简要的介绍，细节问题可以参考它的论文。Raft 提供几个重要的功能：

- Leader（主副本）选举
- 成员变更（如添加副本、删除副本、转移 Leader 等操作）
- 日志复制

TiKV 利用 Raft 来做数据复制，每个数据变更都会落地为一条 Raft 日志，通过 Raft 的日志复制功能，将数据安全可靠地同步到复制组的每一个节点中。不过在实际写入中，根据 Raft 的协议，只需要同步复制到多数节点，即可安全地认为数据写入成功。

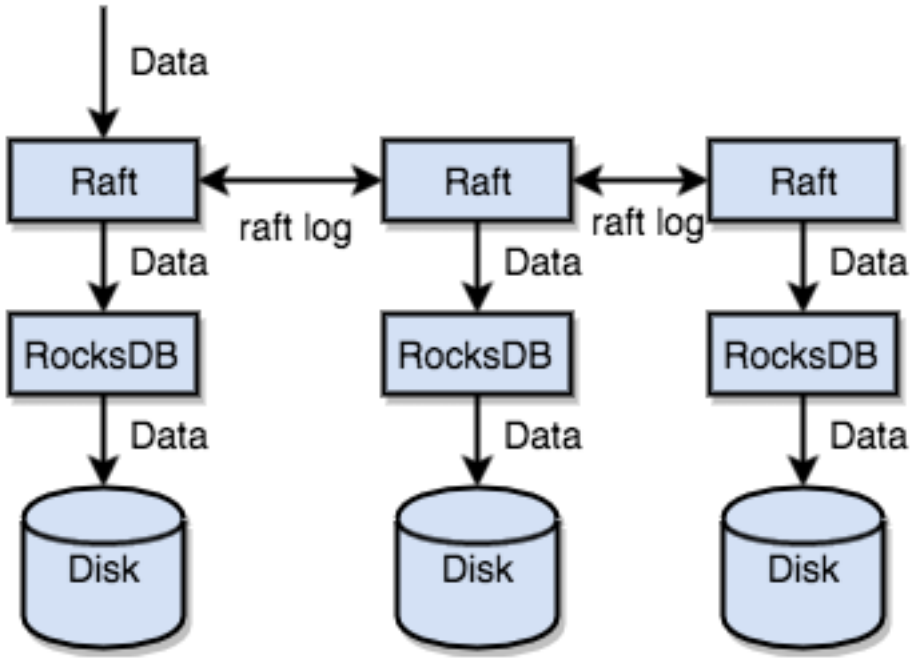


图 190: Raft in TiDB

总结一下，通过单机的 RocksDB，TiKV 可以将数据快速地存储在磁盘上；通过 Raft，将数据复制到多台机器上，以防单机失效。数据的写入是通过 Raft 这一层的接口写入，而不是直接写 RocksDB。通过实现 Raft，TiKV 变成了一个分布式的 Key-Value 存储，少数几台机器宕机也能通过原生的 Raft 协议自动把副本补全，可以做到对业务无感知。

12.1.2.4 Region

首先，为了便于理解，在此节，假设所有的数据都只有一个副本。前面提到，TiKV 可以看做是一个巨大的有序的 KV Map，那么为了实现存储的水平扩展，数据将被分散在多台机器上。对于一个 KV 系统，将数据分散在多台机器上有两种比较典型的方案：

- Hash：按照 Key 做 Hash，根据 Hash 值选择对应的存储节点。
- Range：按照 Key 分 Range，某一段连续的 Key 都保存在一个存储节点上。

TiKV 选择了第二种方式，将整个 Key-Value 空间分成很多段，每一段是一系列连续的 Key，将每一段叫做一个 Region，并且会尽量保持每个 Region 中保存的数据不超过一定的大小，目前在 TiKV 中默认是 96MB。每一个 Region 都可以用 [StartKey，EndKey) 这样一个左闭右开区间来描述。

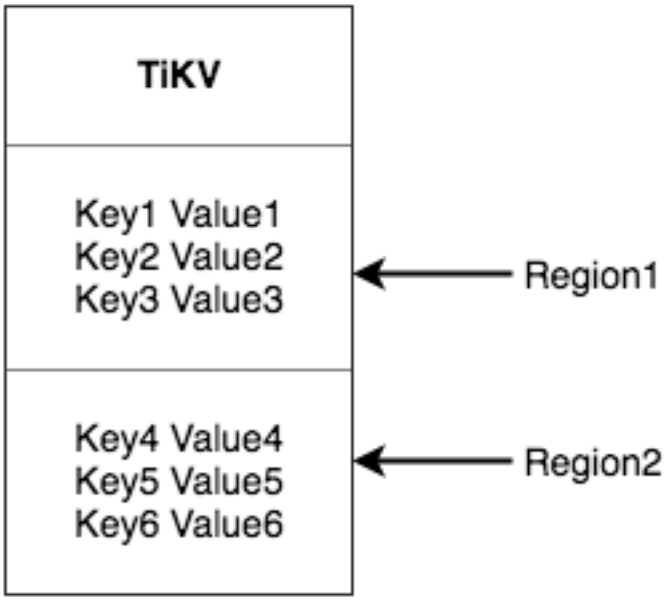


图 191: Region in TiDB

注意，这里的 Region 还是和 SQL 中的表没什么关系。这里的讨论依然不涉及 SQL，只和 KV 有关。

将数据划分成 Region 后，TiKV 将会做两件重要的事情：

- 以 Region 为单位，将数据分散在集群中所有的节点上，并且尽量保证每个节点上服务的 Region 数量差不多。
- 以 Region 为单位做 Raft 的复制和成员管理。

这两点非常重要：

- 先看第一点，数据按照 Key 切分成很多 Region，每个 Region 的数据只会保存在一个节点上面（暂不考虑多副本）。TiDB 系统会有一个组件（PD）来负责将 Region 尽可能均匀的散布在集群中所有的节点上，这样一方面实现了存储容量的水平扩展（增加新的节点后，会自动将其他节点上的 Region 调度过来），另一方面也实现了负载均衡（不会出现某个节点有很多数据，其他节点上没什么数据的情况）。同时为了保证上层客户端能够访问所需要的数据，系统中也会有一个组件（PD）记录 Region 在节点上面的分布情况，也就是通过任意一个 Key 就能查询到这个 Key 在哪个 Region 中，以及这个 Region 目前在哪个节点上（即 Key 的位置路由信息）。至于负责这两项重要工作的组件（PD），会在后续介绍。
- 对于第二点，TiKV 是以 Region 为单位做数据的复制，也就是一个 Region 的数据会保存多个副本，TiKV 将每一个副本叫做一个 Replica。Replica 之间是通过 Raft 来保持数据的一致，一个 Region 的多个 Replica 会保存在不同的节点上，构成一个 Raft Group。其中一个 Replica 会作为这个 Group 的 Leader，其他的 Replica 作为 Follower。默认情况下，所有的读和写都是通过 Leader 进行，读操作在 Leader 上即可完成，而写操作再由 Leader 复制给 Follower。

大家理解了 Region 之后，应该可以理解下面这张图：

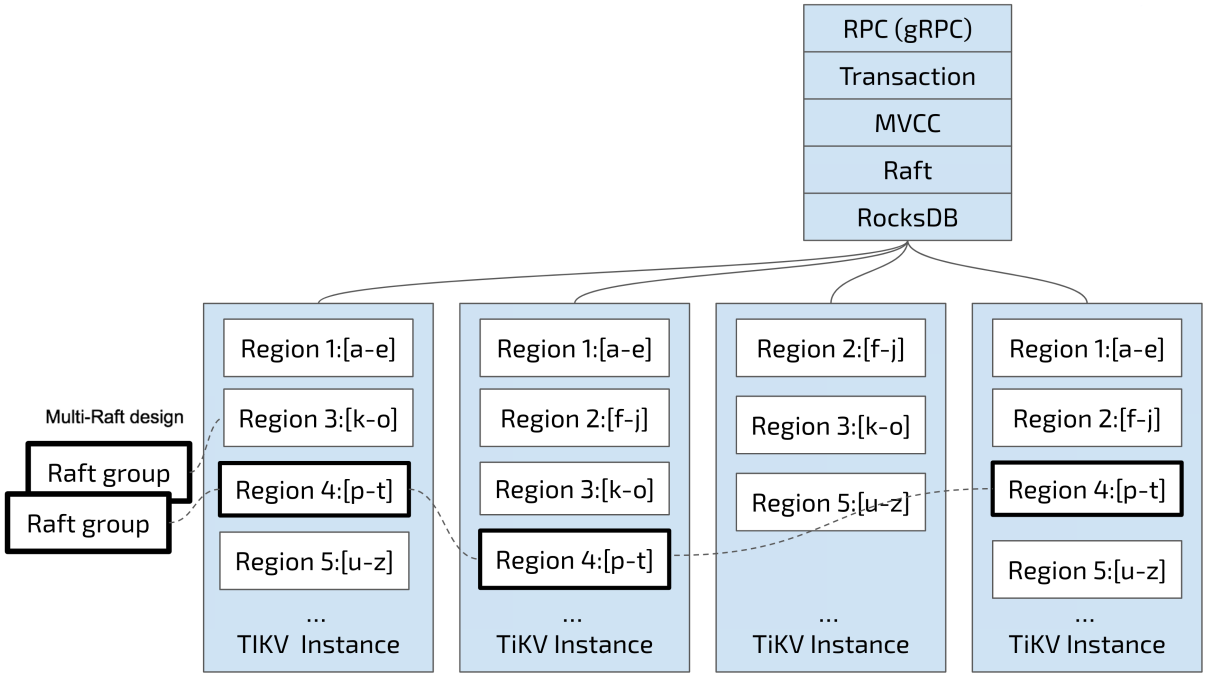


图 192: TiDB Storage

以 Region 为单位做数据的分散和复制，TiKV 就成为了一个分布式的具备一定容灾能力的 KeyValue 系统，不用再担心数据存不下，或者是磁盘故障丢失数据的问题。

12.1.2.5 MVCC

很多数据库都会实现多版本并发控制 (MVCC)，TiKV 也不例外。设想这样的场景：两个客户端同时去修改一个 Key 的 Value，如果没有数据的多版本控制，就需要对数据上锁，在分布式场景下，可能会带来性能以及死锁问题。TiKV 的 MVCC 实现是通过在 Key 后面添加版本号来实现，简单来说，没有 MVCC 之前，可以把 TiKV 看做这样的：

```
Key1 -> Value
Key2 -> Value
……
KeyN -> Value
```

有了 MVCC 之后，TiKV 的 Key 排列是这样的：

```
Key1_Version3 -> Value
Key1_Version2 -> Value
Key1_Version1 -> Value
……
Key2_Version4 -> Value
Key2_Version3 -> Value
Key2_Version2 -> Value
```

```
Key2_Version1 -> Value
......
KeyN_Version2 -> Value
KeyN_Version1 -> Value
......
```

注意，对于同一个 Key 的多个版本，版本号较大的会被放在前面，版本号小的会被放在后面（见Key-Value 一节, Key 是有序的排列），这样当用户通过一个 Key + Version 来获取 Value 的时候，可以通过 Key 和 Version 构造出 MVCC 的 Key，也就是 Key_Version。然后可以直接通过 RocksDB 的 SeekPrefix(Key_Version) API，定位到第一个大于等于这个 Key_Version 的位置。

12.1.2.6 分布式 ACID 事务

TiKV 的事务采用的是 Google 在 BigTable 中使用的事务模型：Percolator，TiKV 根据这篇论文实现，并做了大量的优化。详细介绍参见事务概览。

12.1.3 TiDB 数据库的计算

TiDB 在 TiKV 提供的分布式存储能力基础上，构建了兼具优异的交易处理能力与良好的数据分析能力的计算引擎。本文首先从数据映射算法入手介绍 TiDB 如何将库表中的数据映射到 TiKV 中的 (Key, Value) 键值对，然后描述 TiDB 元信息管理方式，最后介绍 TiDB SQL 层的主要架构。

对于计算层依赖的存储方案，本文只介绍基于 TiKV 的行存储结构。针对分析型业务的特点，TiDB 推出了作为 TiKV 扩展的列存储方案TiFlash。

12.1.3.1 表数据与 Key-Value 的映射关系

本小节介绍 TiDB 中数据到 (Key, Value) 键值对的映射方案。这里的数据主要包括以下两个方面：

- 表中每一行的数据，以下简称表数据
- 表中所有索引的数据，以下简称索引数据

12.1.3.1.1 表数据与 Key-Value 的映射关系

在关系型数据库中，一个表可能有很多列。要将一行中各列数据映射成一个 (Key, Value) 键值对，需要考虑如何构造 Key。首先，OLTP 场景下有大量针对单行或者多行的增、删、改、查等操作，要求数据库具备快速读取一行数据的能力。因此，对应的 Key 最好有一个唯一 ID（显示或隐式的 ID），以方便快速定位。其次，很多 OLAP 型查询需要进行全表扫描。如果能够将一个表中所有行的 Key 编码到一个区间内，就可以通过范围查询高效完成全表扫描的任务。

基于上述考虑，TiDB 中的表数据与 Key-Value 的映射关系作了如下设计：

- 为了保证同一个表的数据放在一起，方便查找，TiDB 会为每个表分配一个表 ID，用 TableID 表示。表 ID 是一个整数，在整个集群内唯一。
- TiDB 会为表中每行数据分配一个行 ID，用 RowID 表示。行 ID 也是一个整数，在表内唯一。对于行 ID，TiDB 做了一个小优化，如果某个表有整数型的主键，TiDB 会使用主键的值当做这一行数据的行 ID。

每行数据按照如下规则编码成 (Key, Value) 键值对：

```
Key:   tablePrefix{TableID}_recordPrefixSep{RowID}
Value: [col1, col2, col3, col4]
```

其中 tablePrefix 和 recordPrefixSep 都是特定的字符串常量，用于在 Key 空间内区分其他数据。其具体值在后面的小结中给出。

12.1.3.1.2 索引数据和 Key-Value 的映射关系

TiDB 同时支持主键和二级索引（包括唯一索引和非唯一索引）。与表数据映射方案类似，TiDB 为表中每个索引分配了一个索引 ID，用 IndexID 表示。

对于主键和唯一索引，需要根据键值快速定位到对应的 RowID，因此，按照如下规则编码成 (Key, Value) 键值对：

```
Key:   tablePrefix{tableID}_indexPrefixSep{indexID}_indexedColumnsValue
Value: RowID
```

对于不需要满足唯一性约束的普通二级索引，一个键值可能对应多行，需要根据键值范围查询对应的 RowID。因此，按照如下规则编码成 (Key, Value) 键值对：

```
Key:   tablePrefix{TableID}_indexPrefixSep{IndexID}_indexedColumnsValue_{RowID}
Value: null
```

12.1.3.1.3 映射关系小结

上述所有编码规则中的 tablePrefix、recordPrefixSep 和 indexPrefixSep 都是字符串常量，用于在 Key 空间内区分其他数据，定义如下：

```
tablePrefix     = []byte{'t'}
recordPrefixSep = []byte{'r'}
indexPrefixSep  = []byte{'i'}
```

另外请注意，上述方案中，无论是表数据还是索引数据的 Key 编码方案，一个表内所有的行都有相同的 Key 前缀，一个索引的所有数据也都有相同的前缀。这样具有相同的前缀的数据，在 TiKV 的 Key 空间内，是排列在一起的。因此只要小心地设计后缀部分的编码方案，保证编码前和编码后的比较关系不变，就可以将表数据或者索引数据有序地保存在 TiKV 中。采用这种编码后，一个表的所有行数据会按照 RowID 顺序地排列在 TiKV 的 Key 空间中，某一个索引的数据也会按照索引数据的具体的值（编码方案中的 indexedColumnsValue）顺序地排列在 Key 空间内。

12.1.3.1.4 Key-Value 映射关系示例

最后通过一个简单的例子，来理解 TiDB 的 Key-Value 映射关系。假设 TiDB 中有如下这个表：

```
CREATE TABLE User (
    ID int,
    Name varchar(20),
    Role varchar(20),
```

```
    Age int,
    PRIMARY KEY (ID),
    KEY idxAge (Age)
);
```

假设该表中有 3 行数据：

```
1, "TiDB", "SQL Layer", 10
2, "TiKV", "KV Engine", 20
3, "PD", "Manager", 30
```

首先每行数据都会映射为一个 (Key, Value) 键值对，同时该表有一个 int 类型的主键，所以 RowID 的值即为该主键的值。假设该表的 TableID 为 10，则其存储在 TiKV 上的表数据为：

```
t10_r1 --> ["TiDB", "SQL Layer", 10]
t10_r2 --> ["TiKV", "KV Engine", 20]
t10_r3 --> ["PD", "Manager", 30]
```

除了主键外，该表还有一个非唯一的普通二级索引 idxAge，假设这个索引的 IndexID 为 1，则其存储在 TiKV 上的索引数据为：

```
t10_i1_10_1 --> null
t10_i1_20_2 --> null
t10_i1_30_3 --> null
```

以上例子展示了 TiDB 中关系模型到 Key-Value 模型的映射规则，以及选择该方案背后的考量。

12.1.3.2 元信息管理

TiDB 中每个 Database 和 Table 都有元信息，也就是其定义以及各项属性。这些信息也需要持久化，TiDB 将这些信息也存储在了 TiKV 中。

每个 Database/Table 都被分配了一个唯一的 ID，这个 ID 作为唯一标识，并且在编码为 Key-Value 时，这个 ID 都会编码到 Key 中，再加上 m_ 前缀。这样可以构造出一个 Key，Value 中存储的是序列化后的元信息。

除此之外，TiDB 还用一个专门的 (Key, Value) 键值对存储当前所有表结构信息的最新版本号。这个键值对是全局的，每次 DDL 操作的状态改变时其版本号都会加 1。目前，TiDB 把这个键值对持久化存储在 PD Server 中，其 Key 是 “/tidb/ddl/global_schema_version”，Value 是类型为 int64 的版本号值。TiDB 采用 Online Schema 变更算法，有一个后台线程在不断地检查 PD Server 中存储的表结构信息的版本号是否发生变化，并且保证在一定时间内一定能够获取版本的变化。

12.1.3.3 SQL 层简介

TiDB 的 SQL 层，即 TiDB Server，负责将 SQL 翻译成 Key-Value 操作，将其转发给共用的分布式 Key-Value 存储层 TiKV，然后组装 TiKV 返回的结果，最终将查询结果返回给客户端。

这一层的节点都是无状态的，节点本身并不存储数据，节点之间完全对等。

12.1.3.3.1 SQL 运算

最简单的方案就是通过上一节所述的表数据与 Key-Value 的映射关系方案，将 SQL 查询映射为对 KV 的查询，再通过 KV 接口获取对应的数据，最后执行各种计算。

比如 `select count(*)from user where name = "TiDB"` 这样一个 SQL 语句，它需要读取表中所有的数据，然后检查 `name` 字段是否是 `TiDB`，如果是的话，则返回这一行。具体流程如下：

1. 构造出 Key Range：一个表中所有的 RowID 都在 `[0, MaxInt64)` 这个范围内，使用 0 和 MaxInt64 根据行数据的 Key 编码规则，就能构造出一个 `[StartKey, EndKey)`的左闭右开区间。
2. 扫描 Key Range：根据上面构造出的 Key Range，读取 TiKV 中的数据。
3. 过滤数据：对于读到的每一行数据，计算 `name = "TiDB"` 这个表达式，如果为真，则向上返回这一行，否则丢弃这一行数据。
4. 计算 `Count(*)`：对符合要求的每一行，累计到 `Count(*)` 的结果上面。

整个流程示意图如下：

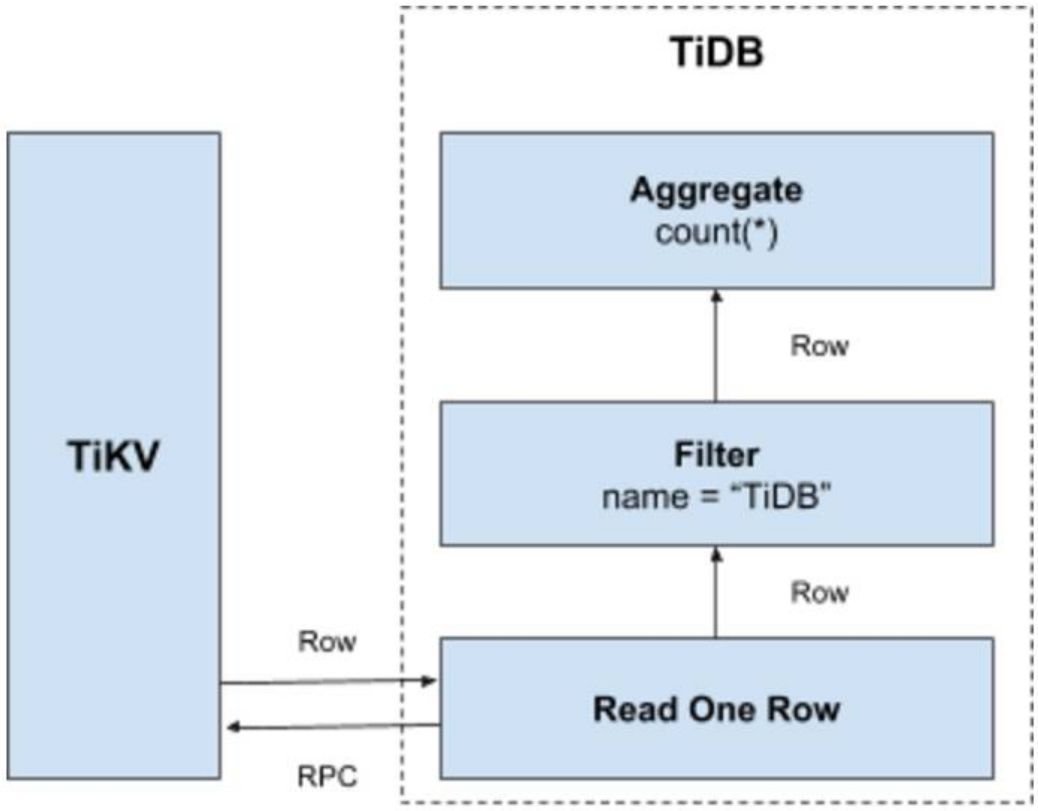


图 193: naive sql flow

这个方案是直观且可行的，但是在分布式数据库的场景下有一些显而易见的问题：

- 在扫描数据的时候，每一行都要通过 KV 操作从 TiKV 中读取出来，至少有一次 RPC 开销，如果需要扫描的数据很多，那么这个开销会非常大。
- 并不是所有的行都满足过滤条件 `name = "TiDB"`，如果不满足条件，其实可以不读取出来。
- 此查询只要求返回符合要求行的数量，不要求返回这些行的值。

12.1.3.3.2 分布式 SQL 运算

为了解决上述问题，计算应该需要尽量靠近存储节点，以避免大量的 RPC 调用。首先，SQL 中的谓词条件 `name = "TiDB"` 应被下推到存储节点进行计算，这样只需要返回有效的行，避免无意义的网络传输。然后，聚合函数 `Count(*)` 也可以被下推到存储节点，进行预聚合，每个节点只需要返回一个 `Count(*)` 的结果即可，再由 SQL 层将各个节点返回的 `Count(*)` 的结果累加求和。

以下是数据逐层返回的示意图：

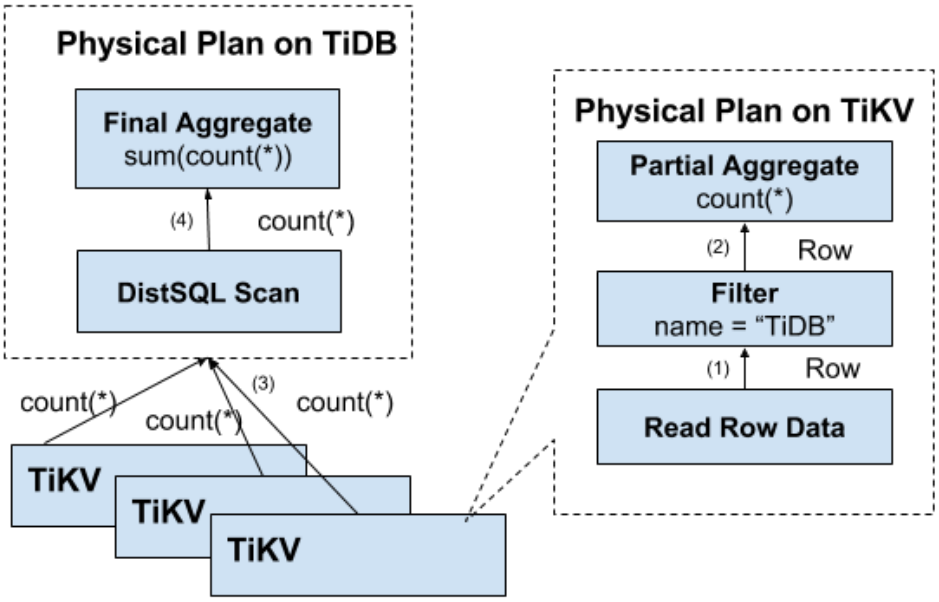


图 194: dist sql flow

12.1.3.3.3 SQL 层架构

通过上面的例子，希望大家对 SQL 语句的处理有一个基本的了解。实际上 TiDB 的 SQL 层要复杂得多，模块以及层次非常多，下图列出了重要的模块以及调用关系：

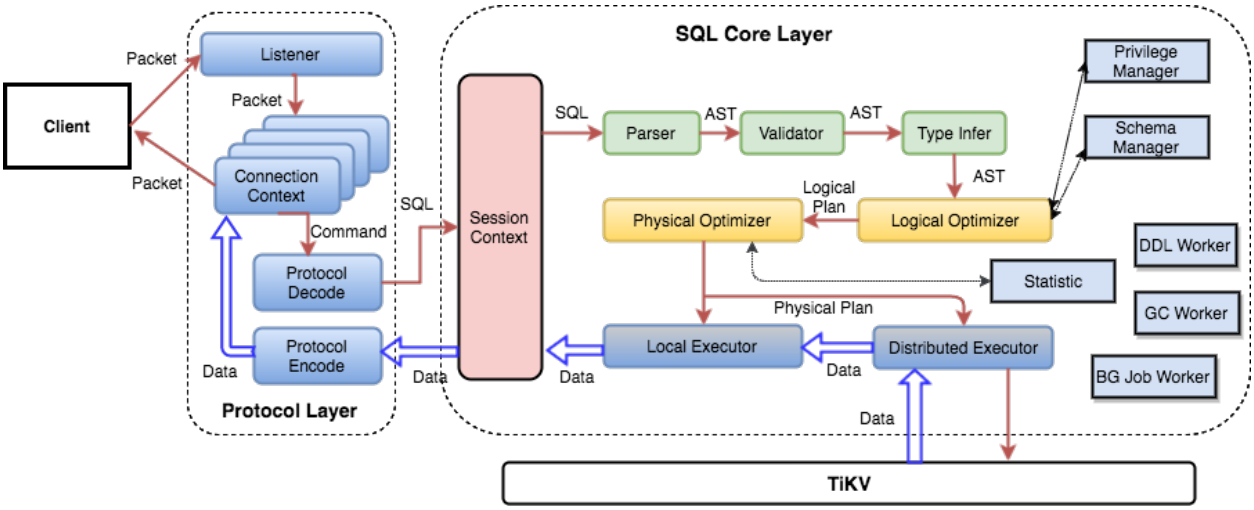


图 195: tidb sql layer

用户的 SQL 请求会直接或者通过 Load Balancer 发送到 TiDB Server，TiDB Server 会解析 MySQL Protocol Packet，获取请求内容，对 SQL 进行语法解析和语义分析，制定和优化查询计划，执行查询计划并获取和处理数据。数据全部存储在 TiKV 集群中，所以在这个过程中 TiDB Server 需要和 TiKV 交互，获取数据。最后 TiDB Server 需要将查询结果返回给用户。

12.1.4 TiDB 数据库的调度

PD (Placement Driver) 是 TiDB 集群的管理模块，同时也负责集群数据的实时调度。本文档介绍一下 PD 的设计思想和关键概念。

12.1.4.1 场景描述

TiKV 集群是 TiDB 数据库的分布式 KV 存储引擎，数据以 Region 为单位进行复制和管理，每个 Region 会有多个副本 (Replica)，这些副本会分布在不同的 TiKV 节点上，其中 Leader 负责读/写，Follower 负责同步 Leader 发来的 Raft log。

需要考虑以下场景：

- 为了提高集群的空间利用率，需要根据 Region 的空间占用对副本进行合理的分布。
- 集群进行跨机房部署的时候，要保证一个机房掉线，不会丢失 Raft Group 的多个副本。
- 添加一个节点进入 TiKV 集群之后，需要合理地将集群中其他节点上的数据搬到新增节点。
- 当一个节点掉线时，需要考虑快速稳定地进行容灾。
  - 从节点的恢复时间来看
    - 如果节点只是短暂掉线（重启服务），是否需要进行调度。
    - 如果节点是长时间掉线（磁盘故障，数据全部丢失），如何进行调度。
  - 假设集群需要每个 Raft Group 有 N 个副本，从单个 Raft Group 的副本个数来看
    - 副本数量不够（例如节点掉线，失去副本），需要选择适当的机器的进行补充。
    - 副本数量过多（例如掉线的节点又恢复正常，自动加入集群），需要合理的删除多余的副本。

- 读/写通过 Leader 进行，Leader 的分布只集中在少量几个节点会对集群造成影响。
- 并不是所有的 Region 都被频繁的访问，可能访问热点只在少数几个 Region，需要通过调度进行负载均衡。
- 集群在做负载均衡的时候，往往需要搬迁数据，这种数据的迁移可能会占用大量的网络带宽、磁盘 IO 以及 CPU，进而影响在线服务。

以上问题和场景如果多个同时出现，就不太容易解决，因为需要考虑全局信息。同时整个系统也是在动态变化的，因此需要一个中心节点，来对系统的整体状况进行把控和调整，所以有了 PD 这个模块。

12.1.4.2 调度的需求

对以上的问题和场景进行分类和整理，可归为以下两类：

第一类：作为一个分布式高可用存储系统，必须满足的需求，包括几种：

- 副本数量不能多也不能少
- 副本需要根据拓扑结构分布在不同属性的机器上
- 节点宕机或异常能够自动合理快速地进行容灾

第二类：作为一个良好的分布式系统，需要考虑的地方包括：

- 维持整个集群的 Leader 分布均匀
- 维持每个节点的储存容量均匀
- 维持访问热点分布均匀
- 控制负载均衡的速度，避免影响在线服务
- 管理节点状态，包括手动上线/下线节点

满足第一类需求后，整个系统将具备强大的容灾功能。满足第二类需求后，可以使得系统整体的资源利用率更高且合理，具备良好的扩展性。

为了满足这些需求，首先需要收集足够的信息，比如每个节点的状态、每个 Raft Group 的信息、业务访问操作的统计等；其次需要设置一些策略，PD 根据这些信息以及调度的策略，制定出尽量满足前面所述需求的调度计划；最后需要一些基本的操作，来完成调度计划。

12.1.4.3 调度的基本操作

调度的基本操作指的是为了满足调度的策略。上述调度需求可整理为以下三个操作：

- 增加一个副本
- 删除一个副本
- 将 Leader 角色在一个 Raft Group 的不同副本之间 transfer（迁移）。

刚好 Raft 协议通过 `AddReplica`、`RemoveReplica`、`TransferLeader` 这三个命令，可以支撑上述三种基本操作。

12.1.4.4 信息收集

调度依赖于整个集群信息的收集，简单来说，调度需要知道每个 TiKV 节点的状态以及每个 Region 的状态。TiKV 集群会向 PD 汇报两类消息，TiKV 节点信息和 Region 信息：

每个 TiKV 节点会定期向 PD 汇报节点的状态信息

TiKV 节点（Store）与 PD 之间存在心跳包，一方面 PD 通过心跳包检测每个 Store 是否存活，以及是否有新加入的 Store；另一方面，心跳包中也会携带这个 Store 的状态信息，主要包括：

- 总磁盘容量
- 可用磁盘容量
- 承载的 Region 数量
- 数据写入/读取速度
- 发送/接受的 Snapshot 数量（副本之间可能会通过 Snapshot 同步数据）
- 是否过载
- labels 标签信息（标签是具备层级关系的一系列 Tag，能够感知拓扑信息）

通过使用 `pd-ctl` 可以查看到 TiKV Store 的状态信息。TiKV Store 的状态具体分为 Up，Disconnect，Offline，Down，Tombstone。各状态的关系如下：

- Up：表示当前的 TiKV Store 处于提供服务的状态。
- Disconnect：当 PD 和 TiKV Store 的心跳信息丢失超过 20 秒后，该 Store 的状态会变为 Disconnect 状态，当时间超过 `max-store-down-time` 指定的时间后，该 Store 会变为 Down 状态。
- Down：表示该 TiKV Store 与集群失去连接的时间已经超过了 `max-store-down-time` 指定的时间，默认 30 分钟。超过该时间后，对应的 Store 会变为 Down，并且开始在存活的 Store 上补足各个 Region 的副本。
- Offline：当对某个 TiKV Store 通过 PD Control 进行手动下线操作，该 Store 会变为 Offline 状态。该状态只是 Store 下线的中间状态，处于该状态的 Store 会将其上的所有 Region 搬离至其它满足搬迁条件的 Up 状态 Store。当该 Store 的 `leader_count` 和 `region_count` (在 PD Control 中获取) 均显示为 0 后，该 Store 会由 Offline 状态变为 Tombstone 状态。在 Offline 状态下，禁止关闭该 Store 服务以及其所在的物理服务器。下线过程中，如果集群里不存在满足搬迁条件的其它目标 Store（例如没有足够的 Store 能够继续满足集群的副本数量要求），该 Store 将一直处于 Offline 状态。
- Tombstone：表示该 TiKV Store 已处于完全下线状态，可以使用 `remove-tombstone` 接口安全地清理该状态的 TiKV。

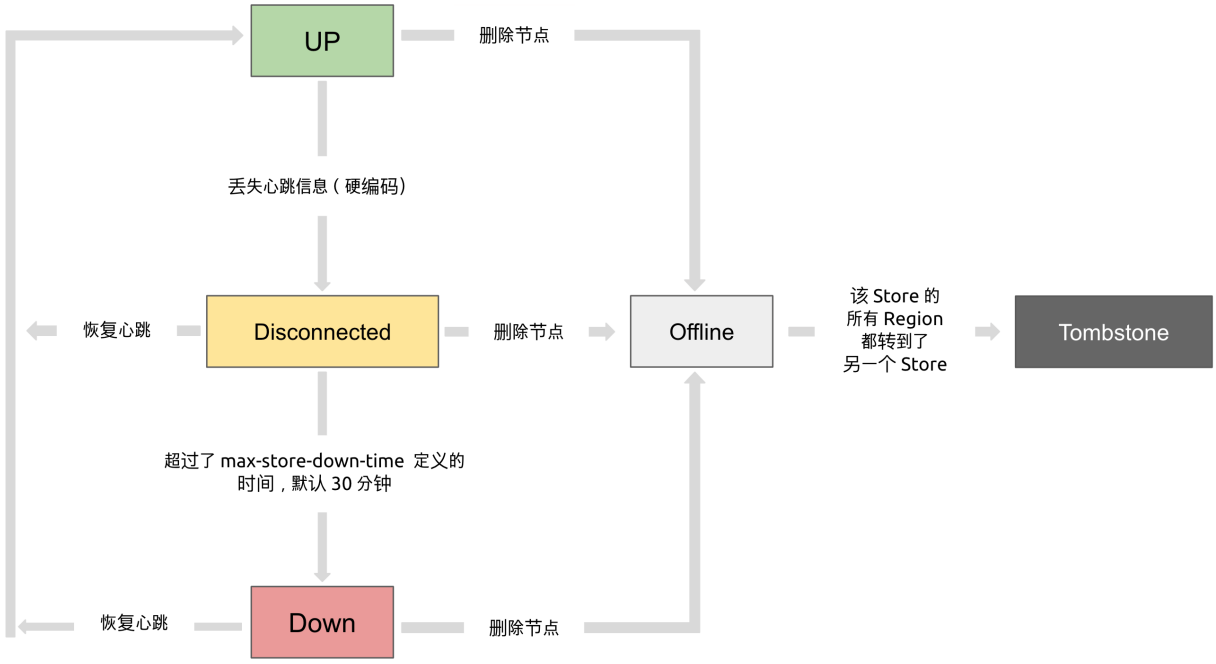


图 196: TiKV store status relationship

每个 Raft Group 的 Leader 会定期向 PD 汇报 Region 的状态信息

每个 Raft Group 的 Leader 和 PD 之间存在心跳包，用于汇报这个Region 的状态，主要包括下面几点信息：

- Leader 的位置
- Followers 的位置
- 掉线副本的个数
- 数据写入/读取的速度

PD 不断的通过这两类心跳消息收集整个集群的信息，再以这些信息作为决策的依据。

除此之外，PD 还可以通过扩展的接口接受额外的信息，用来做更准确的决策。比如当某个 Store 的心跳包中断的时候，PD 并不能判断这个节点是临时失效还是永久失效，只能经过一段时间的等待（默认是 30 分钟），如果一直没有心跳包，就认为该 Store 已经下线，再决定需要将这个 Store 上面的 Region 都调度走。

但是有的时候，是运维人员主动将某台机器下线，这个时候，可以通过 PD 的管理接口通知 PD 该 Store 不可用，PD 就可以马上判断需要将这个 Store 上面的 Region 都调度走。

12.1.4.5 调度的策略

PD 收集了这些信息后，还需要一些策略来制定具体的调度计划。

一个 Region 的副本数量正确

当 PD 通过某个 Region Leader 的心跳包发现这个 Region 的副本数量不满足要求时，需要通过 Add/Remove Replica 操作调整副本数量。出现这种情况的可能原因是：

- 某个节点掉线，上面的数据全部丢失，导致一些 Region 的副本数量不足
- 某个掉线节点又恢复服务，自动接入集群，这样之前已经补足了副本的 Region 的副本数量过多，需要删除某个副本
- 管理员调整副本策略，修改了 max-replicas 的配置

一个 Raft Group 中的多个副本不在同一个位置

注意这里用的是『同一个位置』而不是『同一个节点』。在一般情况下，PD 只会保证多个副本不落在一个节点上，以避免单个节点失效导致多个副本丢失。在实际部署中，还可能出现下面这些需求：

- 多个节点部署在同一台物理机器上
- TiKV 节点分布在多个机架上，希望单个机架掉电时，也能保证系统可用性
- TiKV 节点分布在多个 IDC 中，希望单个机房掉电时，也能保证系统可用性

这些需求本质上都是某一个节点具备共同的位置属性，构成一个最小的『容错单元』，希望这个单元内部不会存在一个 Region 的多个副本。这个时候，可以给节点配置 labels 并且通过在 PD 上配置 location-labels 来指名哪些 label 是位置标识，需要在副本分配的时候尽量保证一个 Region 的多个副本不会分布在具有相同的位置标识的节点上。

副本在 Store 之间的分布均匀分配

由于每个 Region 的副本中存储的数据容量上限是固定的，通过维持每个节点上面副本数量的均衡，使得各节点间承载的数据更均衡。

Leader 数量在 Store 之间均匀分配

Raft 协议要求读取和写入都通过 Leader 进行，所以计算的负载主要在 Leader 上面，PD 会尽可能将 Leader 在节点间分散开。

访问热点数量在 Store 之间均匀分配

每个 Store 以及 Region Leader 在上报信息时携带了当前访问负载的信息，比如 Key 的读取/写入速度。PD 会检测出访问热点，且将其在节点之间分散开。

各个 Store 的存储空间占用大致相等

每个 Store 启动的时候都会指定一个 `Capacity` 参数，表明这个 Store 的存储空间上限，PD 在做调度的时候，会考虑节点的存储空间剩余量。

控制调度速度，避免影响在线服务

调度操作需要耗费 CPU、内存、磁盘 IO 以及网络带宽，需要避免对线上服务造成太大影响。PD 会对当前正在进行的操作数量进行控制，默认的速度控制是比较保守的，如果希望加快调度（比如停服务升级或者增加新节点，希望尽快调度），那么可以通过调节 PD 参数动态加快调度速度。

12.1.4.6 调度的实现

本节介绍调度的实现

PD 不断地通过 Store 或者 Leader 的心跳包收集整个集群信息，并且根据这些信息以及调度策略生成调度操作序列。每次收到 Region Leader 发来的心跳包时，PD 都会检查这个 Region 是否有待进行的操作，然后通过心跳包的回复消息，将需要进行的操作返回给 Region Leader，并在后面的心跳包中监测执行结果。

注意这里的操作只是给 Region Leader 的建议，并不保证一定能得到执行，具体是否会执行以及什么时候执行，由 Region Leader 根据当前自身状态来定。

12.2 存储引擎 TiKV

12.2.1 TiKV 简介

TiKV 是一个分布式事务型的键值数据库，提供了满足 ACID 约束的分布式事务接口，并且通过 Raft 协议 保证了多副本数据一致性以及高可用。TiKV 作为 TiDB 的存储层，为用户写入 TiDB 的数据提供了持久化以及读写服务，同时还存储了 TiDB 的统计信息数据。

12.2.1.1 整体架构

与传统的整节点备份方式不同，TiKV 参考 Spanner 设计了 multi-raft-group 的副本机制。将数据按照 key 的范围划分成大致相等的切片（下文统称为 Region），每一个切片会有多个副本（通常是 3 个），其中一个副本是 Leader，提供读写服务。TiKV 通过 PD 对这些 Region 以及副本进行调度，以保证数据和读写负载都均匀地分散在各个 TiKV 上，这样的设计保证了整个集群资源的充分利用并且可以随着机器数量的增加水平扩展。

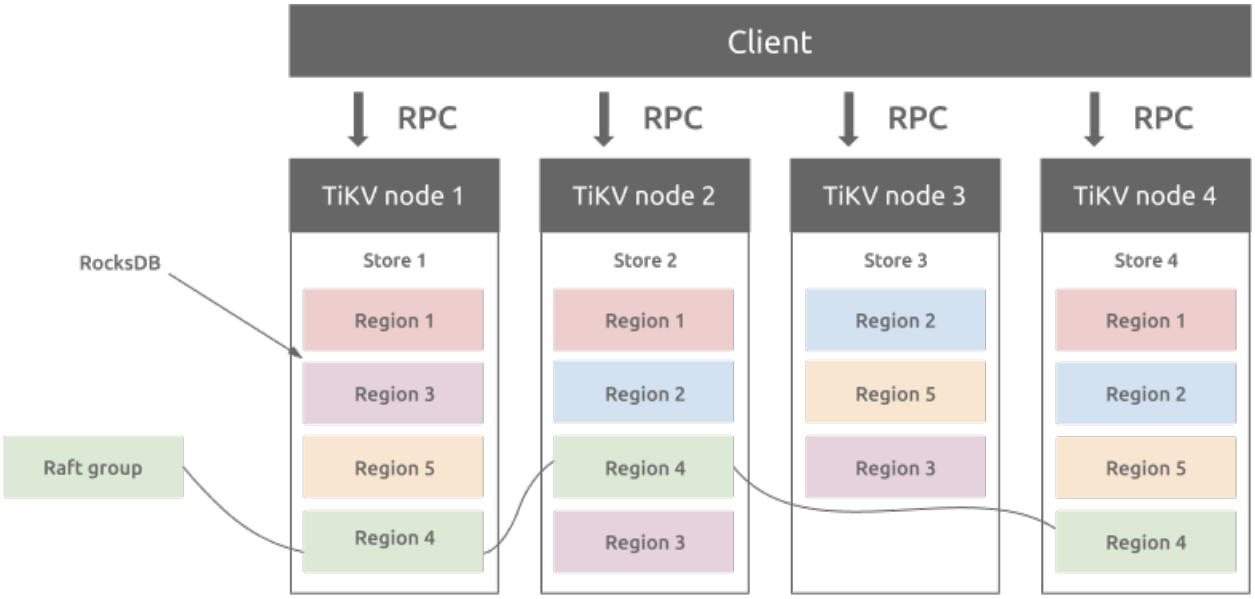


图 197: TiKV 架构

12.2.1.1.1 Region 与 RocksDB

虽然 TiKV 将数据按照范围切割成了多个 Region，但是同一个节点的所有 Region 数据仍然是不加区分地存储于同一个 RocksDB 实例上，而用于 Raft 协议复制所需要的日志则存储于另一个 RocksDB 实例。这样设计的原因是因为随机 I/O 的性能远低于顺序 I/O，所以 TiKV 使用同一个 RocksDB 实例来存储这些数据，以便不同 Region 的写入可以合并在一次 I/O 中。

12.2.1.1.2 Region 与 Raft 协议

Region 与副本之间通过 Raft 协议来维持数据一致性，任何写请求都只能在 Leader 上写入，并且需要写入多数副本后（默认配置为 3 副本，即所有请求必须至少写入两个副本成功）才会返回客户端写入成功。

当某个 Region 的大小超过一定限制（默认是 144MB）后，TiKV 会将它分裂为两个或者更多个 Region，以保证各个 Region 的大小是大致接近的，这样更有利于 PD 进行调度决策。同样，当某个 Region 因为大量的删除请求导致 Region 的大小变得更小时，TiKV 会将比较小的两个相邻 Region 合并为一个。

当 PD 需要把某个 Region 的一个副本从一个 TiKV 节点调度到另一个上面时，PD 会先为这个 Raft Group 在目标节点上增加一个 Learner 副本（虽然会复制 Leader 的数据，但是不会计入写请求的多数副本中）。当这个 Learner 副本的进度大致追上 Leader 副本时，Leader 会将它变更为 Follower，之后再移除操作节点的 Follower 副本，这样就完成了 Region 副本的一次调度。

Leader 副本的调度原理也类似，不过需要在目标节点的 Learner 副本变为 Follower 副本后，再执行一次 Leader Transfer，让该 Follower 主动发起一次选举成为新 Leader，之后新 Leader 负责删除旧 Leader 这个副本。

12.2.1.2 分布式事务

TiKV 支持分布式事务，用户（或者 TiDB）可以一次性写入多个 key-value 而不必关心这些 key-value 是否处于同一个数据切片 (Region) 上，TiKV 通过两阶段提交保证了这些读写请求的 ACID 约束，详见TiDB 乐观事务模型。

12.2.1.3 计算加速

TiKV 通过协处理器 (Coprocessor) 可以为 TiDB 分担一部分计算：TiDB 会将可以由存储层分担的计算下推。能否下推取决于 TiKV 是否可以支持相关下推。计算单元仍然是以 Region 为单位，即 TiKV 的一个 Coprocessor 计算请求中不会计算超过一个 Region 的数据。

12.2.2 RocksDB 简介

RocksDB 是由 Facebook 基于 LevelDB 开发的一款提供键值存储与读写功能的 LSM-tree 架构引擎。用户写入的键值对会先写入磁盘上的 WAL (Write Ahead Log)，然后再写入内存中的跳表（SkipList，这部分结构又被称作 MemTable）。LSM-tree 引擎由于将用户的随机修改（插入）转化为了对 WAL 文件的顺序写，因此具有比 B 树类存储引擎更高的写吞吐。

内存中的数据达到一定阈值后，会刷到磁盘上生成 SST 文件 (Sorted String Table)，SST 又分为多层（默认至多 6 层），每一层的数据达到一定阈值后会挑选一部分 SST 合并到下一层，每一层的数据是上一层的 10 倍（因此 90% 的数据存储在最后一层）。

RocksDB 允许用户创建多个 ColumnFamily，这些 ColumnFamily 各自拥有独立的内存跳表以及 SST 文件，但是共享同一个 WAL 文件，这样的好处是可以根据应用特点为不同的 ColumnFamily 选择不同的配置，但是又没有增加对 WAL 的写次数。

12.2.2.1 TiKV 架构

TiKV 的系统架构如下图所示：

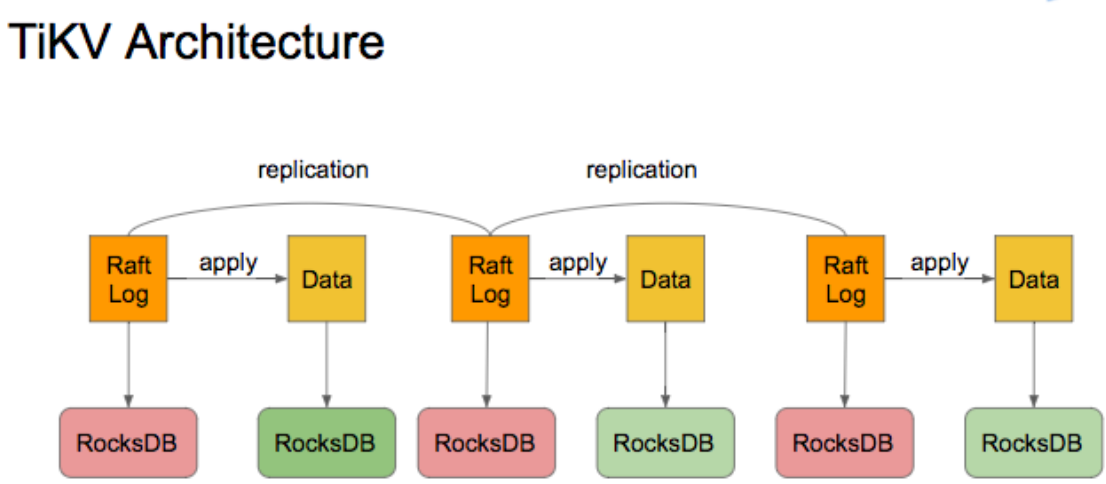


图 198: TiKV RocksDB

RocksDB 作为 TiKV 的核心存储引擎，用于存储 Raft 日志以及用户数据。每个 TiKV 实例中有两个 RocksDB 实例，一个用于存储 Raft 日志（通常被称为 raftdb），另一个用于存储用户数据以及 MVCC 信息（通常被称为 kvdb）。kvdb 中有四个 ColumnFamily：raft、lock、default 和 write：

- raft 列：用于存储各个 Region 的元信息。仅占极少量空间，用户可以不必关注。
- lock 列：用于存储悲观事务的悲观锁以及分布式事务的一阶段 Prewrite 锁。当用户的事务提交之后，lock cf 中对应的数据会很快删除掉，因此大部分情况下 lock cf 中的数据也很少（少于 1GB）。如果 lock cf 中的数据大量增加，说明有大量事务等待提交，系统出现了 bug 或者故障。
- write 列：用于存储用户真实的写入数据以及 MVCC 信息（该数据所属事务的开始时间以及提交时间）。当用户写入了一行数据时，如果该行数据长度小于 255 字节，那么会被存储 write 列中，否则的话该行数据会被存入到 default 列中。由于 TiDB 的非 unique 索引存储的 value 为空，unique 索引存储的 value 为主键索引，因此二级索引只会占用 writecf 的空间。
- default 列：用于存储超过 255 字节长度的数据。

12.2.2.2 RocksDB 的内存占用

为了提高读取性能以及减少对磁盘的读取，RocksDB 将存储在磁盘上的文件都按照一定大小切分成 block（默认是 64KB），读取 block 时先去内存中的 BlockCache 中查看该块数据是否存在，存在的话则可以直接从内存中读取而不必访问磁盘。

BlockCache 按照 LRU 算法淘汰低频访问的数据，TiKV 默认将系统总内存大小的 45% 用于 BlockCache，用户也可以自行修改 `storage.block-cache.capacity` 配置设置为合适的值，但是不建议超过系统总内存的 60%。

写入 RocksDB 中的数据会写入 MemTable，当一个 MemTable 的大小超过 128MB 时，会切换到一个新的 MemTable 来提供写入。TiKV 中一共有 2 个 RocksDB 实例，合计 4 个 ColumnFamily，每个 ColumnFamily 的单个 MemTable 大小限制是 128MB，最多允许 5 个 MemTable 存在，否则会阻塞前台写入，因此这部分占用的内存最多为 4 x 5 x 128MB = 2.5GB。这部分占用内存较少，不建议用户自行更改。

12.2.2.3 RocksDB 的空间占用

- 多版本：RocksDB 作为一个 LSM-tree 结构的键值存储引擎，MemTable 中的数据会首先被刷到 L0。L0 层的 SST 之间的范围可能存在重叠（因为文件顺序是按照生成的顺序排列），因此同一个 key 在 L0 中可能存在多个版本。当文件从 L0 合并到 L1 的时候，会按照一定大小（默认是 8MB）切割为多个文件，同一层的文件的范围互不重叠，所以 L1 及其以后的层每一层的 key 都只有一个版本。
- 空间放大：RocksDB 的每一层文件总大小都是上一层的 x 倍，在 TiKV 中这个配置默认是 10，因此 90% 的数据存储在最后一层，这也意味着 RocksDB 的空间放大不超过 1.11 (L0 层的数据较少，可以忽略不计)
- TiKV 的空间放大：TiKV 在 RocksDB 之上还有一层自己的 MVCC，当用户写入一个 key 的时候，实际上写入到 RocksDB 的是 key + commit_ts，也就是说，用户的更新和删除都是会写入新的 key 到 RocksDB。TiKV 每隔一段时间会删除旧版本的数据（通过 RocksDB 的 Delete 接口），因此可以认为用户存储在 TiKV 上的数据的实际空间放大为，1.11 加最近 10 分钟内写入的数据（假设 TiKV 回收旧版本数据足够及时）。详情见《TiDB in Action》。

12.2.2.4 RocksDB 后台线程与 Compact

RocksDB 中，将内存中的 MemTable 转化为磁盘上的 SST 文件，以及合并各个层级的 SST 文件等操作都是在后台线程池中执行的。后台线程池的默认大小是 8，当机器 CPU 数量小于等于 8 时，则后台线程池默认大小为 CPU 数量减一。通常来说，用户不需要更改这个配置。如果用户在一个机器上部署了多个 TiKV 实例，或者机器的读负载比较高而写负载比较低，那么可以适当调低 `rocksdb/max-background-jobs` 至 3 或者 4。

12.2.2.5 WriteStall

RocksDB 的 L0 与其他层不同，L0 的各个 SST 是按照生成顺序排列，各个 SST 之间的 key 范围存在重叠，因此查询的时候必须依次查询 L0 中的每一个 SST。为了不影响查询性能，当 L0 中的文件数量过多时，会触发 WriteStall 阻塞写入。

如果用户遇到了写延迟突然大幅度上涨，可以先查看 Grafana RocksDB KV 面板 WriteStall Reason 指标，如果是 L0 文件数量过多引起的 WriteStall，可以调整下面几个配置到 64，详细见《TiDB in Action》。

```
rocksdb.defaultcf.level0-slowdown-writes-trigger
rocksdb.writecf.level0-slowdown-writes-trigger
rocksdb.lockcf.level0-slowdown-writes-trigger
rocksdb.defaultcf.level0-stop-writes-trigger
rocksdb.writecf.level0-stop-writes-trigger
rocksdb.lockcf.level0-stop-writes-trigger
```

12.2.3 Titan 介绍

Titan 是基于 RocksDB 的高性能单机 key-value 存储引擎插件。

当 value 较大的时候，Titan 在写、更新和点读等场景下性能都优于 RocksDB。但与此同时，Titan 会占用更多硬盘空间和部分舍弃范围查询。随着 SSD 价格的降低，Titan 的优势会更加突出，让用户更容易做出选择。

12.2.3.1 核心特性

- 支持将 value 从 LSM-tree 中分离出来单独存储，以降低写放大。
- 已有 RocksDB 实例可以平滑地升级到 Titan，这意味着升级过程不需要人工干预，并且不会影响线上服务。
- 100% 兼容目前 TiKV 所使用的所有 RocksDB 的特性。

12.2.3.2 适用场景

Titan 适合在以下场景中使用：

- 前台写入量较大，RocksDB 大量触发 compaction 消耗大量 I/O 带宽或者 CPU 资源，造成 TiKV 前台读写性能较差。
- 前台写入量较大，由于 I/O 带宽瓶颈或 CPU 瓶颈的限制，RocksDB compaction 进度落后较多频繁造成 write stall。
- 前台写入量较大，RocksDB 大量触发 compaction 造成 I/O 写入量较大，影响 SSD 盘的寿命。

开启 Titan 需要考虑以下前提条件：

- Value 较大。即 value 平均大小比较大，或者数据中大 value 的数据总大小占比比较大。目前 Titan 默认 1KB 以上大小的 value 是大 value，根据实际情况 512B 以上大小的 value 也可以看作是大 value。注：由于 TiKV Raft 层的限制，写入 TiKV 的 value 大小还是无法超过 8MB 的限制，可通过`raft-entry-max-size`配置项调整该限制。
- 没有范围查询或者对范围查询性能不敏感。Titan 存储数据的顺序性较差，所以相比 RocksDB 范围查询的性能较差，尤其是大范围查询。在测试中 Titan 范围查询性能相比 RocksDB 下降 40% 到数倍不等。
- 磁盘剩余空间足够。Titan 降低写放大是通过牺牲空间放大达到的。另外由于 Titan 逐个压缩 value，压缩率比 RocksDB（逐个压缩 block）要差。这两个因素一起造成 Titan 占用磁盘空间比 RocksDB 要多，这是正常现象。根据实际情况和不同的配置，Titan 磁盘空间占用可能会比 RocksDB 多一倍。

性能提升请参考 Titan 的设计与实现。

12.2.3.3 架构与实现

Titan 的基本架构如下图所示：

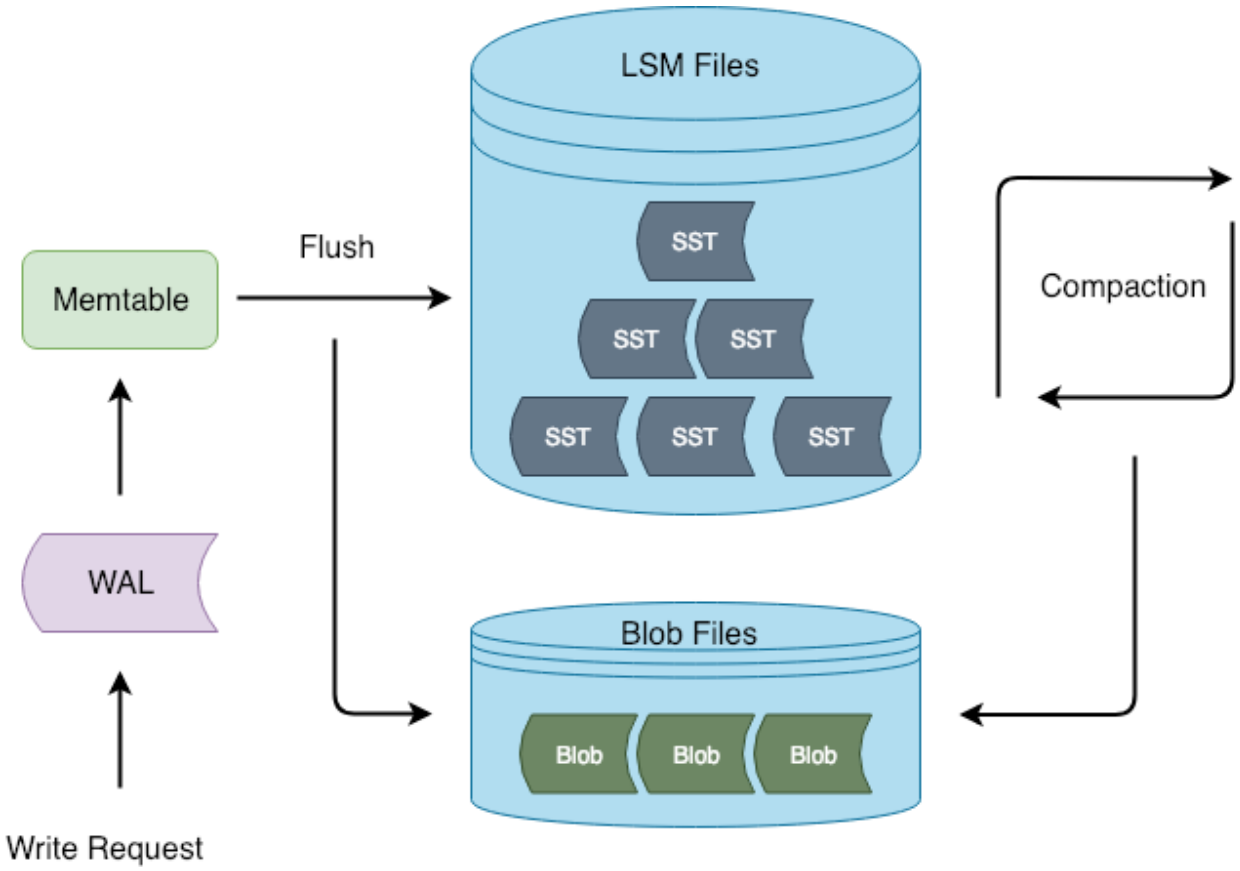


图 199: Architecture

Titan 在 Flush 和 Compaction 的时候将 value 分离出 LSM-tree，这样写入流程可以和 RocksDB 保持一致，减少对 RocksDB 的侵入性改动。

12.2.3.3.1 BlobFile

BlobFile 是用来存放从 LSM-tree 中分离出来的 value 的文件，其格式如下图所示：

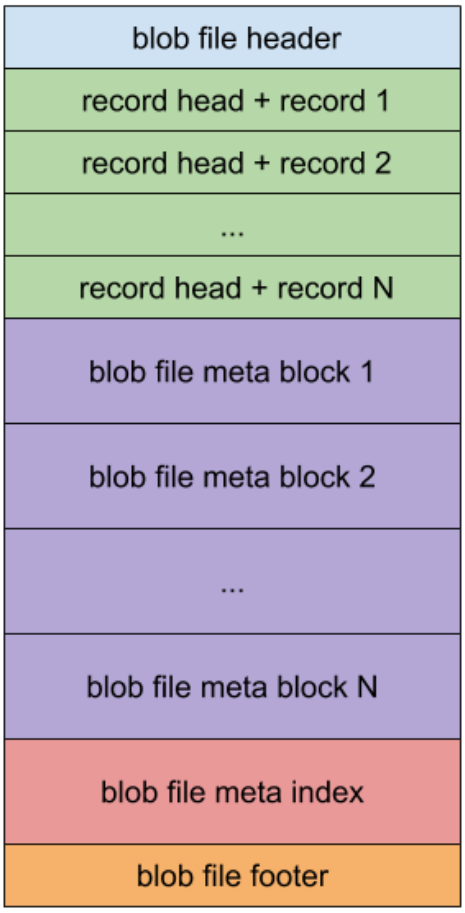


图 200: BlobFile

BlobFile 由 blob record 、meta block、meta index block 和 footer 组成。其中每个 blob record 用于存放一个 key-value 对；meta block 支持可扩展性，可以用来存放和 BlobFile 相关的一些属性；meta index block 用于检索 meta block。

BlobFile 的实现上有几点值得关注的地方：

- BlobFile 中的 key-value 是有序存放的，目的是在实现 iterator 的时候可以通过 prefetch 的方式提高顺序读取的性能。
- 每个 blob record 都保留了 value 对应的 user key 的拷贝，这样做的目的是在进行 GC 的时候，可以通过查询 user key 是否更新来确定对应 value 是否已经过期，但同时也带来了一定的写放大。
- BlobFile 支持 blob record 粒度的 compression，并且支持多种 compression algorithm，包括 Snappy、 LZ4 和 Zstd 等，目前 Titan 默认使用的 compression algorithm 是 LZ4。

12.2.3.3.2 TitanTableBuilder

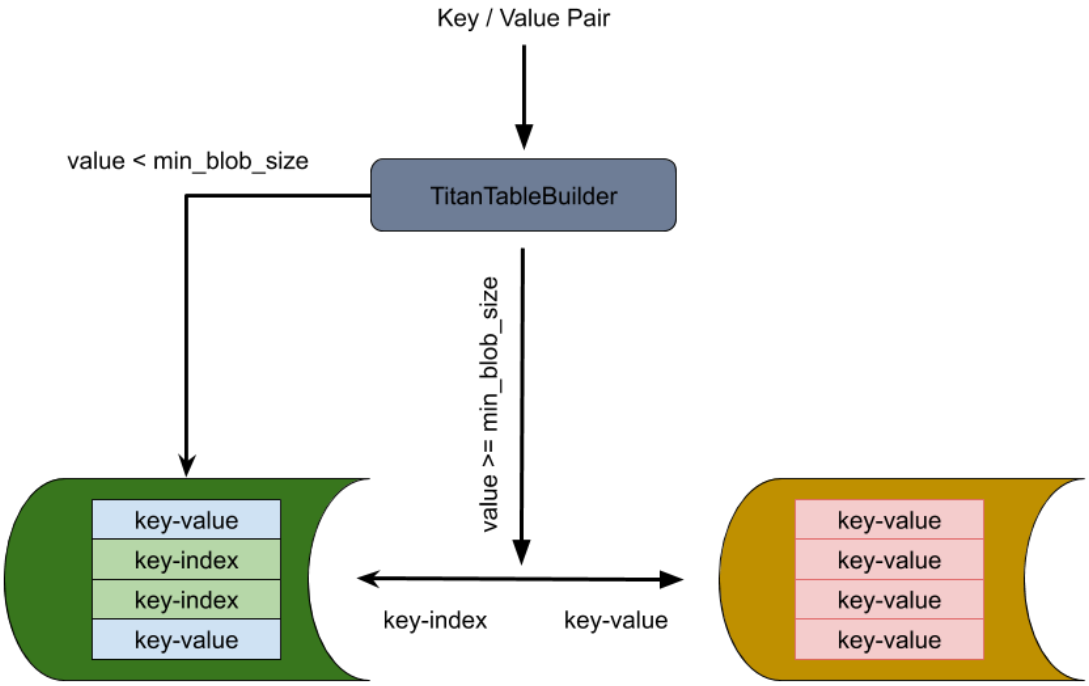


图 201: TitanTableBuilder

TitanTableBuilder 是实现分离 key-value 的关键，它通过判断 value size 的大小来决定是否将 value 分离到 BlobFile 中去。如果 value size 大于等于 `min_blob_size` 则将 value 分离到 BlobFile，并生成 index 写入 SST；如果 value size 小于 `min_blob_size` 则将 value 直接写入 SST。

该流程还支持将 Titan 降级回 RocksDB。在 RocksDB 做 compaction 的时候，将分离出来的 value 重新写回新生成的 SST 文件中。

12.2.3.4 Garbage Collection

Garbage Collection (GC) 的目的是回收空间。由于在 LSM-tree compaction 进行回收 key 时，储存在 blob 文件中的 value 并不会一同被删除，因此需要 GC 定期来将已经作废的 value 删除掉。在 Titan 中有两种 GC 方式可供选择：

- 定期整合重写 Blob 文件将作废的 value 剔除（传统 GC）
- 在 LSM-tree compaction 的时候同时进行 blob 文件的重写（Level-Merge）

12.2.3.4.1 传统 GC

Titan 使用 RocksDB 的 TablePropertiesCollector 和 EventListener 来收集 GC 所需的统计信息。

TablePropertiesCollector

RocksDB 允许使用自定义的 TablePropertiesCollector 来搜集 SST 上的 properties 并写入到对应文件中去。Titan 通过一个自定义的 TablePropertiesCollector —— BlobFileSizeCollector 来搜集每个 SST 中有多少数据是存放在哪些 BlobFile 上的，将它收集到的 properties 命名为 BlobFileSizeProperties，它的工作流程和数据格式如下图所示：

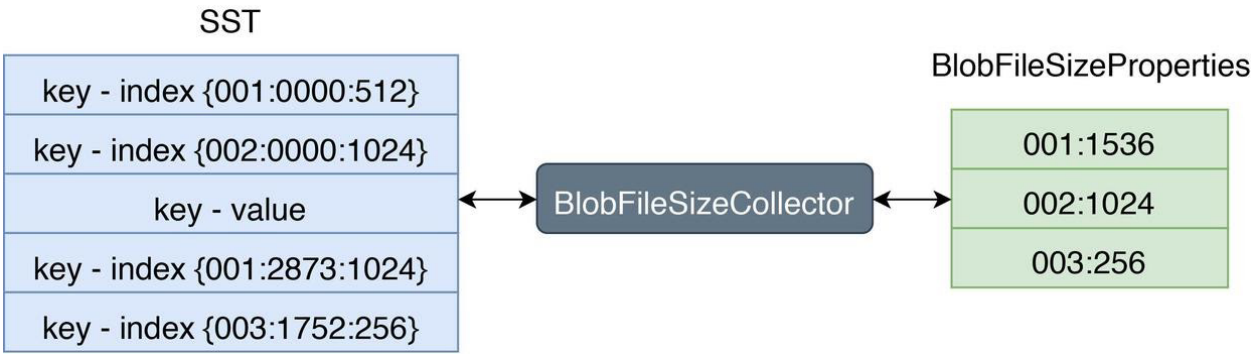


图 202: BlobFileSizeProperties

左边 SST 中 Index 的格式为：第一列代表 BlobFile 的文件 ID，第二列代表 blob record 在 BlobFile 中的 offset，第三列代表 blob record 的 size。右边 BlobFileSizeProperties 中的每一行代表一个 BlobFile 以及 SST 中有多少数据保存在这个 BlobFile 中，第一列代表 BlobFile 的文件 ID，第二列代表数据大小。

EventListener

RocksDB 是通过 Compaction 来丢弃旧版本数据以回收空间的，因此每次 Compaction 完成后 Titan 中的某些 BlobFile 中便可能有部分或全部数据过期。因此便可以通过监听 Compaction 事件来触发 GC，搜集比对 Compaction 中输入输出 SST 的 BlobFileSizeProperties 来决定挑选哪些 BlobFile 进行 GC。其流程大概如下图所示：

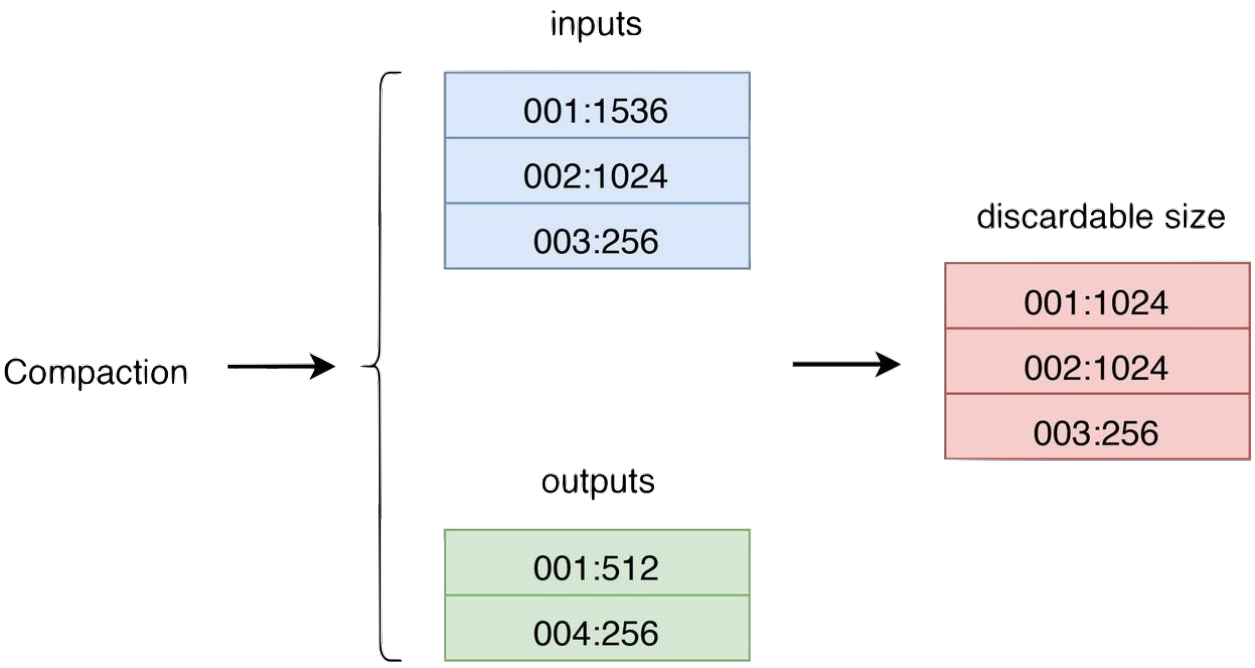


图 203: EventListener

inputs 代表参与 Compaction 的所有 SST 的 BlobFileSizeProperties，outputs 代表 Compaction 生成的所有 SST 的 BlobFileSizeProperties，discardable size 是通过计算 inputs 和 outputs 得出的每个 BlobFile 被丢弃的数据大小，第一列代表 BlobFile 的文件 ID，第二列代表被丢弃的数据大小。

Titan 会为每个有效的 BlobFile 在内存中维护一个 discardable size 变量，每次 Compaction 结束之后都对相应的 BlobFile 的 discardable size 变量进行累加。注意，在每次重启后会扫描一遍所有的 SST 的 BlobFileSizeProperties 重新构建每个有效 BlobFile 的 discardable size 变量。每次 GC 开始时就可以通过挑选 discardable size 最大的几个 BlobFile 来作为候选的文件。为了减小写放大，我们可以容忍一定的空间放大，所以 Titan 只有在 BlobFile 可丢弃的数据达到一定比例之后才会对其进行 GC。

GC 的方式就是对于这些选中的 BlobFile 文件，依次通过查询其中每个 value 相应的 key 的 blob index 是否存在或者更新来确定该 value 是否作废，最终将未作废的 value 归并排序生成新的 BlobFile，并将这些 value 更新后的 blob index 通过 WriteCallback 或者 MergeOperator 的方式写回到 SST 中。在完成 GC 后，这些原来的 BlobFile 文件并不会立即被删除，Titan 会在写回 blob index 后记录 RocksDB 最新的 sequence number，等到最旧 snapshot 的 sequence 超过这个记录的 sequence number 时 BlobFile 才能被删除。这个是因为在写回 blob index 后，还是可能通过之前的 snapshot 访问到老的 blob index，因此需要确保没有 snapshot 会访问到这个老的 blob index 后才能安全删除相应 BlobFile。

12.2.3.4.2 Level Merge

Level Merge 是 Titan 新加入的一种策略，它的核心思想是 LSM-tree 在进行 Compaction 的同时，对 SST 文件对应的 BlobFile 进行归并重写产生新的 BlobFile。其流程大概如下图所示：

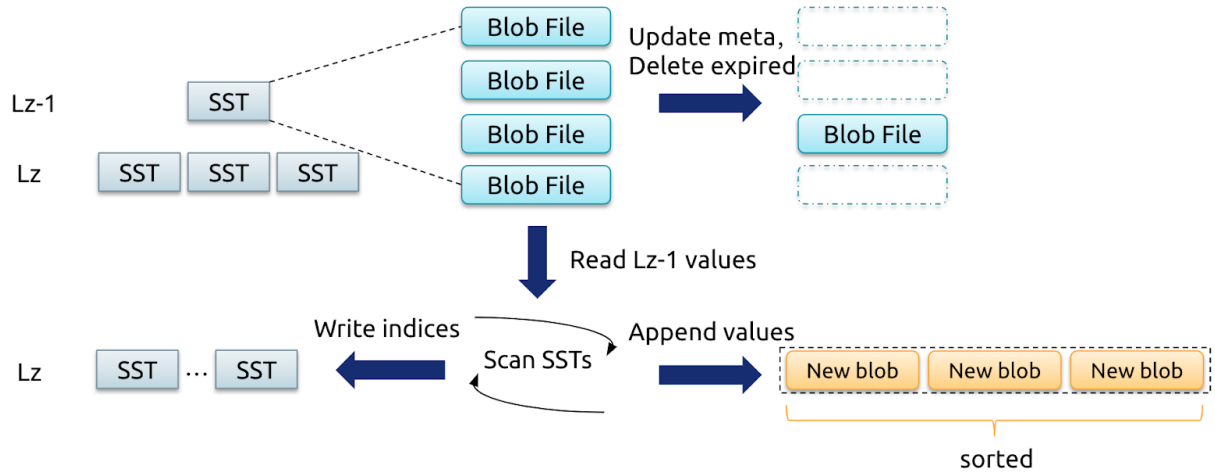


图 204: LevelMerge

Level z-1 和 Level z 的 SST 进行 Compaction 时会对 KV 对有序读写一遍，这时就可以对这些 SST 中所涉及的 BlobFile 的 value 有序写到新的 BlobFile 中，并在生成新的 SST 时将 key 的 blob index 进行更新。对于 Compaction 中被删除的 key，相应的 value 也不会写到新的 BlobFile 中，相当于完成了 GC。

相比于传统 GC，Level Merge 这种方式在 LSM-tree 进行 Compaction 的同时就完成了 Blob GC，不再需要查询 LSM-tree 的 blob index 情况和写回新 blob index 到 LSM-tree 中，减小了 GC 对前台操作影响。同时通过不断的重写 BlobFile，减小了 BlobFile 之间的相互重叠，提高系统整体有序性，也就是提高了 Scan 性能。当然将 BlobFile 以类似 tiering compaction 的方式分层会带来写放大，考虑到 LSM-tree 中 99% 的数据都落在最后两层，因此 Titan 仅对 LSM-tree 中 Compaction 到最后两层数据对应的 BlobFile 进行 Level Merge。

Range Merge

Range Merge 是基于 Level Merge 的一个优化。考虑如下两种情况，会导致最底层的有序性越来越差：

- 开启 `level_compaction_dynamic_level_bytes`，此时 LSM-tree 各层动态增长，随数据量增大最后一层的 sorted run 会越来越多。
- 某个 range 被频繁 Compaction 导致该 range 的 sorted runs 较多。

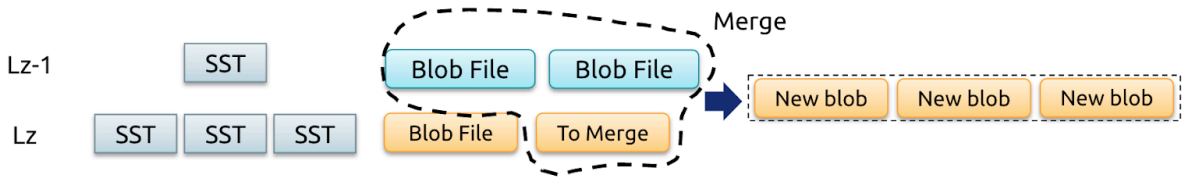


图 205: RangeMerge

因此需要通过 Range Merge 操作维持 sorted run 在一定水平，即在 OnCompactionComplete 时统计该 range 的 sorted run 数量，若数量过多则将涉及的 BlobFile 标记为 ToMerge，在下一次的 Compaction 中进行重写。

12.2.4 Titan 配置

本文档介绍如何通过Titan 配置项来开启、关闭 Titan、相关参数以及 Level Merge 功能。

12.2.4.1 开启 Titan

Titan 对 RocksDB 兼容，也就是说，使用 RocksDB 存储引擎的现有 TiKV 实例可以直接开启 Titan。

- 方法一：如果使用 TiUP 部署的集群，开启的方法是执行 `tiup cluster edit-config ${cluster-name}` 命令，再编辑 TiKV 的配置文件。编辑 TiKV 配置文件示例如下：

```
  tikv:
    rocksdb.titan.enabled: true
```

  重新加载配置，同时也会在线滚动重启 TiKV：

```
tiup cluster reload ${cluster-name} -R tikv
```

  具体命令，可参考通过 TiUP 修改配置参数。

- 方法二：直接编辑 TiKV 配置文件开启 Titan（生产环境不推荐）。

```
[rocksdb.titan]
enabled = true
```

开启 Titan 以后，原有的数据并不会马上移入 Titan 引擎，而是随着前台写入和 RocksDB compaction 的进行，逐步进行 key-value 分离并写入 Titan。可以通过观察 TiKV Details - Titan kv - blob file size 监控面版确认数据保存在 Titan 中部分的大小。

如果需要加速数据移入 Titan，可以通过 tikv-ctl 执行一次全量 compaction，具体参考手动 compact。

> **警告：**
>
> 在不开启 Titan 功能的情况下，RocksDB 无法读取已经迁移到 Titan 的数据。如果在打开过 Titan 的 TiKV 实例上错误地关闭了 Titan（误设置 `rocksdb.titan.enabled = false`），启动 TiKV 会失败，TiKV log 中出现 `You have disabled titan when its data directory is not empty` 错误。如需要关闭 Titan，参考关闭 Titan 一节。

12.2.4.2 相关参数介绍

使用 TiUP 调整参数，请参考修改配置参数。

- Titan GC 线程数。

  当从 TiKV Details - Thread CPU - RocksDB CPU 监控中观察到 Titan GC 线程长期处于满负荷状态时，应该考虑增加 Titan GC 线程池大小。

```
[rocksdb.titan]
max-background-gc = 1
```

- value 的大小阈值。

  当写入的 value 小于这个值时，value 会保存在 RocksDB 中，反之则保存在 Titan 的 blob file 中。根据 value 大小的分布，增大这个值可以使更多 value 保存在 RocksDB，读取这些小 value 的性能会稍好一些；减少这个值可以使更多 value 保存在 Titan 中，进一步减少 RocksDB compaction。

```
[rocksdb.defaultcf.titan]
min-blob-size = "1KB"
```

- Titan 中 value 所使用的压缩算法。Titan 中压缩是以 value 为单元的。

```
[rocksdb.defaultcf.titan]
blob-file-compression = "lz4"
```

- Titan 中 value 的缓存大小。

  更大的缓存能提高 Titan 读性能，但过大的缓存会造成 OOM。建议在数据库稳定运行后，根据监控把 RocksDB block cache (`storage.block-cache.capacity`) 设置为 store size 减去 blob file size 的大小，`blob-cache-size` 设置为 内存大小 * 50% 再减去 block cache 的大小。这是为了保证 block cache 足够缓存整个 RocksDB 的前提下，blob cache 尽量大。

```
[rocksdb.defaultcf.titan]
blob-cache-size = 0
```

- 当一个 blob file 中无用数据（相应的 key 已经被更新或删除）比例超过以下阈值时，将会触发 Titan GC。

```
discardable-ratio = 0.5
```

将此文件有用的数据重写到另一个文件。这个值可以估算 Titan 的写放大和空间放大的上界（假设关闭压缩）。公式是：

写放大上界 = 1 / discardable_ratio

空间放大上界 = 1 / (1 - discardable_ratio)

可以看到，减少这个阈值可以减少空间放大，但是会造成 Titan 更频繁 GC；增加这个值可以减少 Titan GC，减少相应的 I/O 带宽和 CPU 消耗，但是会增加磁盘空间占用。

- 以下选项限制 RocksDB compaction 的 I/O 速率，以达到在流量高峰时，限制 RocksDB compaction 减少其 I/O 带宽和 CPU 消耗对前台读写性能的影响。

  当开启 Titan 时，该选项限制 RocksDB compaction 和 Titan GC 的 I/O 速率总和。当发现在流量高峰时 RocksDB compaction 和 Titan GC 的 I/O 和/或 CPU 消耗过大，可以根据磁盘 I/O 带宽和实际写入流量适当配置这个选项。

```
[rocksdb]
rate-bytes-per-sec = 0
```

12.2.4.3 关闭 Titan（实验功能）

通过设置 `rocksdb.defaultcf.titan.blob-run-mode` 参数可以关闭 Titan。`blob-run-mode` 可以设置为以下几个值之一：

- 当设置为 `normal` 时，Titan 处于正常读写的状态。
- 当设置为 `read-only` 时，新写入的 value 不论大小均会写入 RocksDB。
- 当设置为 `fallback` 时，新写入的 value 不论大小均会写入 RocksDB，并且当 RocksDB 进行 compaction 时，会自动把所碰到的存储在 Titan blob file 中的 value 移回 RocksDB。

当需要关闭 Titan 时，可以设置 `blob-run-mode = "fallback"`，并通过 tikv-ctl 执行全量 compaction。此后通过监控确认 blob file size 降到 0 以后，可以更改 `rocksdb.titan.enabled = false` 并重启 TiKV。

> **警告：**
>
> 关闭 Titan 是实验性功能，非必要不建议使用。

12.2.4.4 Level Merge（实验功能）

TiKV 4.0 中 Titan 提供新的算法提升范围查询性能并降低 Titan GC 对前台写入性能的影响。这个新的算法称为Level Merge。Level Merge 可以通过以下选项开启：

```
[rocksdb.defaultcf.titan]
level-merge = true
```

开启 Level Merge 的好处如下：

- 大幅提升 Titan 的范围查询性能。
- 减少了 Titan GC 对前台写入性能的影响，提升写入性能。
- 减少 Titan 空间放大，减少磁盘空间占用（默认配置下的比较）。

相应地，Level Merge 的写放大会比 Titan 稍高，但依然低于原生的 RocksDB。

12.3 存储引擎 TiFlash

12.3.1 TiFlash 简介

TiFlash 是 TiDB HTAP 形态的关键组件，它是 TiKV 的列存扩展，在提供了良好的隔离性的同时，也兼顾了强一致性。列存副本通过 Raft Learner 协议异步复制，但是在读取的时候通过 Raft 校对索引配合 MVCC 的方式获得 Snapshot Isolation 的一致性隔离级别。这个架构很好地解决了 HTAP 场景的隔离性以及列存同步的问题。

12.3.1.1 整体架构

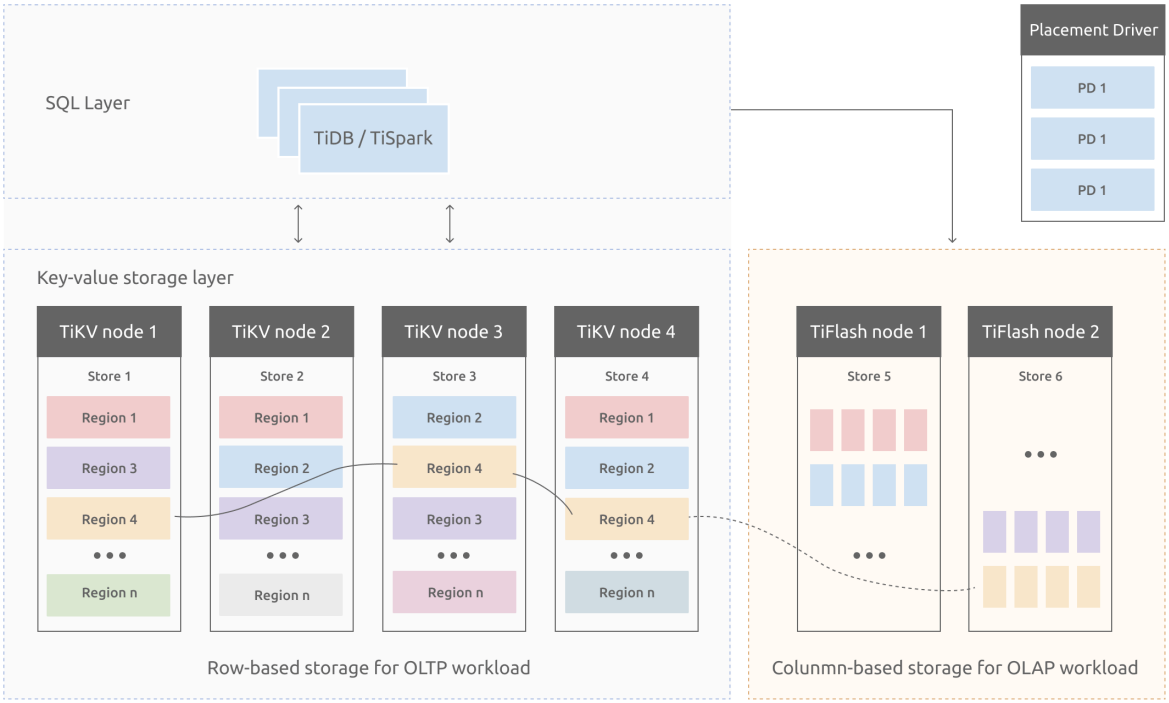


图 206: TiFlash 架构

上图为 TiDB HTAP 形态架构，其中包含 TiFlash 节点。

TiFlash 提供列式存储，且拥有借助 ClickHouse 高效实现的协处理器层。除此以外，它与 TiKV 非常类似，依赖同样的 Multi-Raft 体系，以 Region 为单位进行数据复制和分散（详情见《说存储》一文）。

TiFlash 以低消耗不阻塞 TiKV 写入的方式，实时复制 TiKV 集群中的数据，并同时提供与 TiKV 一样的一致性读取，且可以保证读取到最新的数据。TiFlash 中的 Region 副本与 TiKV 中完全对应，且会跟随 TiKV 中的 Leader 副本同时进行分裂与合并。

TiFlash 可以兼容 TiDB 与 TiSpark，用户可以选择使用不同的计算引擎。

TiFlash 推荐使用和 TiKV 不同的节点以做到 Workload 隔离，但在无业务隔离的前提下，也可以选择与 TiKV 同节点部署。

TiFlash 暂时无法直接接受数据写入，任何数据必须先写入 TiKV 再同步到 TiFlash。TiFlash 以 learner 角色接入 TiDB 集群，TiFlash 支持表粒度的数据同步，部署后默认情况下不会同步任何数据，需要按照按表构建 TiFlash 副本一节完成指定表的数据同步。

TiFlash 主要包含三个组件，除了主要的存储引擎组件，另外包含 tiflash proxy 和 pd buddy 组件，其中 tiflash proxy 主要用于处理 Multi-Raft 协议通信的相关工作，pd buddy 负责与 PD 协同工作，将 TiKV 数据按表同步到 TiFlash。

对于按表构建 TiFlash 副本的流程，TiDB 接收到相应的 DDL 命令后 pd buddy 组件会通过 TiDB 的 status 端口获取到需要同步的数据表信息，然后会将需要同步的数据信息发送到 PD，PD 根据该信息进行相关的数据调度。

12.3.1.2 核心特性

TiFlash 主要有异步复制、一致性、智能选择、计算加速等几个核心特性。

12.3.1.2.1 异步复制

TiFlash 中的副本以特殊角色 (Raft Learner) 进行异步的数据复制。这表示当 TiFlash 节点宕机或者网络高延迟等状况发生时，TiKV 的业务仍然能确保正常进行。

这套复制机制也继承了 TiKV 体系的自动负载均衡和高可用：并不用依赖附加的复制管道，而是直接以多对多方式接收 TiKV 的数据传输；且只要 TiKV 中数据不丢失，就可以随时恢复 TiFlash 的副本。

12.3.1.2.2 一致性

TiFlash 提供与 TiKV 一样的快照隔离支持，且保证读取数据最新（确保之前写入的数据能被读取）。这个一致性是通过对数据进行复制进度校验做到的。

每次收到读取请求，TiFlash 中的 Region 副本会向 Leader 副本发起进度校对（一个非常轻的 RPC 请求），只有当进度确保至少所包含读取请求时间戳所覆盖的数据之后才响应读取。

12.3.1.2.3 智能选择

TiDB 可以自动选择使用 TiFlash 列存或者 TiKV 行存，甚至在同一查询内混合使用提供最佳查询速度。这个选择机制与 TiDB 选取不同索引提供查询类似：根据统计信息判断读取代价并作出合理选择。

12.3.1.2.4 计算加速

TiFlash 对 TiDB 的计算加速分为两部分：列存本身的读取效率提升以及为 TiDB 分担计算。其中分担计算的原理和 TiKV 的协处理器一致：TiDB 会将可以由存储层分担的计算下推。能否下推取决于 TiFlash 是否可以支持相关下推。具体介绍请参阅“TiFlash 支持的计算下推”一节。

12.3.1.3 另请参阅

- 全新部署一个包含 TiFlash 节点的集群，请参考使用 TiUP 部署 TiDB 集群
- 已有集群新增一个 TiFlash 节点，请参考扩容 TiFlash 节点
- 部署完成后，如何使用 TiFlash 节点，请参考使用 TiFlash
- TiFlash 常用运维操作，请参考TiFlash 运维
- TiFlash 性能调优，请参考TiFlash 性能调优
- TiFlash 配置参数介绍，请参考TiFlash 参数
- TiFlash 监控说明，请参考TiFlash 监控
- TiFlash 报警规则，请参考TiFlash 报警
- TiFlash 常见问题处理，请参考TiFlash 常见问题

12.3.2 使用 TiFlash

TiFlash 部署完成后并不会自动同步数据，而需要手动指定需要同步的表。

用户可以使用 TiDB 或者 TiSpark 读取 TiFlash，TiDB 适合用于中等规模的 OLAP 计算，而 TiSpark 适合大规模的 OLAP 计算，用户可以根据自己的场景和使用习惯自行选择。具体参见：

- 使用 TiDB 读取 TiFlash
- 使用 TiSpark 读取 TiFlash

12.3.2.1 按表构建 TiFlash 副本

TiFlash 接入 TiKV 集群后，默认不会开始同步数据。可通过 MySQL 客户端向 TiDB 发送 DDL 命令来为特定的表建立 TiFlash 副本：

```
ALTER TABLE table_name SET TIFLASH REPLICA count
```

该命令的参数说明如下：

- count 表示副本数，0 表示删除。

对于相同表的多次 DDL 命令，仅保证最后一次能生效。例如下面给出的操作 tpch50 表的两条 DDL 命令中，只有第二条删除副本的命令能生效：

为表建立 2 个副本：

```
ALTER TABLE `tpch50`.`lineitem` SET TIFLASH REPLICA 2
```

删除副本：

```
ALTER TABLE `tpch50`.`lineitem` SET TIFLASH REPLICA 0
```

注意事项：

- 假设有一张表 t 已经通过上述的 DDL 语句同步到 TiFlash，则通过以下语句创建的表也会自动同步到 TiFlash：

```
CREATE TABLE table_name like t
```

- 如果集群版本 < v4.0.6，若先对表创建 TiFlash 副本，再使用 TiDB Lightning 导入数据，会导致数据导入失败。需要在使用 TiDB Lightning 成功导入数据至表后，再对相应的表创建 TiFlash 副本。
- 如果集群版本以及 TiDB Lightning 版本均 >= v4.0.6，无论一个表是否已经创建 TiFlash 副本，你均可以使用 TiDB Lightning 导入数据至该表。但注意此情况会导致 TiDB Lightning 导入数据耗费的时间延长，具体取决于 TiDB Lightning 部署机器的网卡带宽、TiFlash 节点的 CPU 及磁盘负载、TiFlash 副本数等因素。
- 不推荐同步 1000 张以上的表，这会降低 PD 的调度性能。这个限制将在后续版本去除。

12.3.2.1.1 查看表同步进度

可通过如下 SQL 语句查看特定表（通过 WHERE 语句指定，去掉 WHERE 语句则查看所有表）的 TiFlash 副本的状态：

```
SELECT * FROM information_schema.tiflash_replica WHERE TABLE_SCHEMA = '<db_name>' and TABLE_NAME
    ↪ = '<table_name>'
```

查询结果中：

- AVAILABLE 字段表示该表的 TiFlash 副本是否可用。1 代表可用，0 代表不可用。副本状态为可用之后就不再改变，如果通过 DDL 命令修改副本数则会重新计算同步进度。
- PROGRESS 字段代表同步进度，在 0.0~1.0 之间，1 代表至少 1 个副本已经完成同步。

12.3.2.2 使用 TiDB 读取 TiFlash

TiDB 提供三种读取 TiFlash 副本的方式。如果添加了 TiFlash 副本，而没有做任何 engine 的配置，则默认使用 CBO 方式。

12.3.2.2.1 智能选择

对于创建了 TiFlash 副本的表，TiDB 优化器会自动根据代价估算选择是否使用 TiFlash 副本。具体有没有选择 TiFlash 副本，可以通过 desc 或 explain analyze 语句查看，例如：

```
desc select count(*) from test.t;
```

```
+--------------------------+---------+--------------+---------------+--------------------------------+
    ↪
| id                       | estRows | task         | access object | operator info
    ↪                  |
+--------------------------+---------+--------------+---------------+--------------------------------+
    ↪
| StreamAgg_9              | 1.00    | root         |               | funcs:count(1)->Column#4
    ↪       |
| └─TableReader_17         | 1.00    | root         |               | data:TableFullScan_16
    ↪          |
```

```
|   └─TableFullScan_16     | 1.00    | cop[tiflash] | table:t       | keep order:false, stats:
    ↪ pseudo |
+--------------------------+---------+--------------+---------------+--------------------------------+
    ↪
3 rows in set (0.00 sec)
```

```
explain analyze select count(*) from test.t;
```

```
+--------------------------+---------+---------+--------------+---------------+--------------------------------
    ↪
| id                       | estRows | actRows | task         | access object | execution info
    ↪                                                        | operator info                  |
    ↪ memory    | disk |
+--------------------------+---------+---------+--------------+---------------+--------------------------------
    ↪
| StreamAgg_9              | 1.00    | 1       | root         |               | time:83.8372ms,
    ↪ loops:2                                                | funcs:count(1)->Column#4       |
    ↪ 372 Bytes | N/A  |
| └─TableReader_17         | 1.00    | 1       | root         |               | time:83.7776ms,
    ↪ loops:2, rpc num: 1, rpc time:83.5701ms, proc keys:0 | data:TableFullScan_16          |
    ↪ 152 Bytes | N/A  |
|   └─TableFullScan_16     | 1.00    | 1       | cop[tiflash] | table:t       | time:43ms, loops
    ↪ :1                                                     | keep order:false, stats:pseudo | N
    ↪ /A       | N/A  |
+--------------------------+---------+---------+--------------+---------------+--------------------------------
    ↪
```

`cop[tiflash]` 表示该任务会发送至 TiFlash 进行处理。如果没有选择 TiFlash 副本，可尝试通过 `analyze table` 语句更新统计信息后，再查看 `explain analyze` 结果。

需要注意的是，如果表仅有单个 TiFlash 副本且相关节点无法服务，智能选择模式下的查询会不断重试，需要指定 Engine 或者手工 Hint 来读取 TiKV 副本。

12.3.2.2.2 Engine 隔离

Engine 隔离是通过配置变量来指定所有的查询均使用指定 engine 的副本，可选 engine 为 “tikv”、“tidb” 和 “tiflash”（其中 “tidb” 表示 TiDB 内部的内存表区，主要用于存储一些 TiDB 系统表，用户不能主动使用），分别有 2 个配置级别：

1. TiDB 实例级别，即 INSTANCE 级别。在 TiDB 的配置文件添加如下配置项：

   ```
   [isolation-read]
   engines = ["tikv", "tidb", "tiflash"]
   ```

   实例级别的默认配置为 `["tikv", "tidb", "tiflash"]`。

2. 会话级别，即 SESSION 级别。设置语句：

```
set @@session.tidb_isolation_read_engines = "逗号分隔的 engine list";
```

或者

```
set SESSION tidb_isolation_read_engines = "逗号分隔的 engine list";
```

会话级别的默认配置继承自 TiDB 实例级别的配置。

最终的 engine 配置为会话级别配置，即会话级别配置会覆盖实例级别配置。比如实例级别配置了 “tikv”，而会话级别配置了 “tiflash”，则会读取 TiFlash 副本。当 engine 配置为 “tikv, tiflash”，即可以同时读取 TiKV 和 TiFlash 副本，优化器会自动选择。

> 注意：
>
> 由于TiDB Dashboard 等组件需要读取一些存储于 TiDB 内存表区的系统表，因此建议实例级别 engine 配置中始终加入 “tidb” engine。

如果查询中的表没有对应 engine 的副本，比如配置了 engine 为 “tiflash” 而该表没有 TiFlash 副本，则查询会报该表不存在该 engine 副本的错。

12.3.2.2.3 手工 Hint

手工 Hint 可以在满足 engine 隔离的前提下，强制 TiDB 对于某张或某几张表使用指定的副本，使用方法为：

```
select /*+ read_from_storage(tiflash[table_name]) */ ... from table_name;
```

如果在查询语句中对表设置了别名，在 Hint 语句中必须使用别名才能使 Hint 生效。比如：

```
select /*+ read_from_storage(tiflash[alias_a,alias_b]) */ ... from table_name_1 as alias_a,
    ↪ table_name_2 as alias_b where alias_a.column_1 = alias_b.column_2;
```

其中 `tiflash[]` 是提示优化器读取 TiFlash 副本，亦可以根据需要使用 `tikv[]` 来提示优化器读取 TiKV 副本。更多关于该 Hint 语句的语法可以参考READ_FROM_STORAGE。

如果 Hint 指定的表在指定的引擎上不存在副本，则 Hint 会被忽略，并产生 warning。另外 Hint 必须在满足 engine 隔离的前提下才会生效，如果 Hint 中指定的引擎不在 engine 隔离列表中，Hint 同样会被忽略，并产生 warning。

> 注意：
>
> MySQL 命令行客户端在 5.7.7 版本之前默认清除了 Optimizer Hints。如果需要在这些早期版本的客户端中使用 `Hint` 语法，需要在启动客户端时加上 `--comments` 选项，例如 `mysql -h 127.0.0.1 -P 4000 -uroot --comments`。

12.3.2.2.4 三种方式之间关系的总结

上述三种读取 TiFlash 副本的方式中，Engine 隔离规定了总的可使用副本 engine 的范围，手工 Hint 可以在该范围内进一步实现语句级别及表级别的细粒度的 engine 指定，最终由 CBO 在指定的 engine 范围内根据代价估算最终选取某个 engine 上的副本。

> **注意：**
>
> TiDB 4.0.3 版本之前，在非只读 SQL 语句中（比如 `INSERT INTO ... SELECT`、`SELECT ... FOR UPDATE`、`UPDATE ...`、`DELETE ...`）读取 TiFlash，行为是未定义。TiDB 4.0.3 以及后续的版本，TiDB 内部会对非只读 SQL 语句忽略 TiFlash 副本以保证数据写入、更新、删除的正确性。对应的，如果使用了智能选择的方式，TiDB 会自动选择非 TiFlash 副本；如果使用了Engine 隔离的方式指定仅读取 TiFlash 副本，则查询会报错；而如果使用了手工 Hint 的方式，则 Hint 会被忽略。

12.3.2.3 使用 TiSpark 读取 TiFlash

TiSpark 目前提供类似 TiDB 中 engine 隔离的方式读取 TiFlash，方式是通过配置参数 `spark.tispark.isolation_read_engines`。参数值默认为 `tikv,tiflash`，表示根据 CBO 自动选择从 TiFlash 或从 TiKV 读取数据。如果将该参数值设置成 `tiflash`，表示强制从 TiFlash 读取数据。

> **注意：**
>
> 设为 `tiflash` 时，所有查询的表都会只读取 TiFlash 副本，设为 `tikv` 则只读取 TiKV 副本。设为 `tiflash` 时，要求查询所用到的表都必须已创建了 TiFlash 副本，对于未创建 TiFlash 副本的表的查询会报错。

可以使用以下任意一种方式进行设置：

1. 在 `spark-defaults.conf` 文件中添加：

```
spark.tispark.isolation_read_engines tiflash
```

2. 在启动 Spark shell 或 Thrift server 时，启动命令中添加 `--conf spark.tispark.isolation_read_engines=tiflash`
3. Spark shell 中实时设置：`spark.conf.set("spark.tispark.isolation_read_engines", "tiflash")`
4. Thrift server 通过 beeline 连接后实时设置：`set spark.tispark.isolation_read_engines=tiflash`

12.3.2.4 TiFlash 支持的计算下推

> **注意：**
>
> TiDB 4.0.2 版本之前，TiFlash 不支持 TiDB 新排序规则框架，所以在 TiDB 开启新框架下的排序规则支持后不支持任何表达式的下推，TiDB 4.0.2 以及后续的版本取消了这个限制。

TiFlash 支持谓词、聚合下推计算以及表连接，下推的计算可以帮助 TiDB 进行分布式加速。暂不支持的计算类型是 `Full Outer Join` 和 `DISTINCT COUNT`，会在后续版本逐步优化。

目前下推连接 (Join) 的功能需要通过以下会话变量开启（暂不支持 `Full Outer Join`）：

```
set @@session.tidb_opt_broadcast_join=1
```

目前 TiFlash 支持了有限的常用表达式下推，支持下推的表达式包括：

```
+, -, /, *, >=, <=, =, !=, <, >, ifnull, isnull, bitor, in, bitand, or, and, like, not, case when
    ↪ , month, substr, timestampdiff, date_format, from_unixtime, json_length, if, bitneg,
    ↪ bitxor, round without fraction, cast(int as decimal), min, max, sum, count, avg,
    ↪ approx_count_distinct
```

目前 TiFlash 不支持下推的情况包括：

- 所有包含 Time 类型的表达式均不能下推
- 在聚合函数或者 WHERE 条件中包含了不在上述列表中的表达式，聚合或者相关的谓词过滤均不能下推

如查询遇到不支持的下推计算，则需要依赖 TiDB 完成剩余计算，可能会很大程度影响 TiFlash 加速效果。对于暂不支持的表达式，将会在后续陆续加入支持，也可以联系官方沟通。

12.4 系统变量

TiDB 系统变量的行为与 MySQL 相似但有一些不同，变量的作用范围可以是全局范围有效 (Global Scope)、实例级别有效 (Instance Scope) 或会话级别有效 (Session Scope)，或组合了上述多个范围。其中：

- 对 GLOBAL 作用域变量的更改，设置后只对新 TiDB 连接会话生效，当前活动连接会话不受影响。更改会被持久化，重启后仍然生效。
- 对 INSTANCE 作用域变量的更改，设置后会立即对当前 TiDB 实例所有活动连接会话或新连接会话生效，其他 TiDB 实例不生效。更改不会被持久化，重启 TiDB 后会失效。

使用SET 语句可以设置变量的作用范围为全局级别、实例级别或会话级别。

```
## 以下两个语句等价地改变一个 Session 变量
SET tidb_distsql_scan_concurrency = 10;
SET SESSION tidb_distsql_scan_concurrency = 10;

## 以下两个语句等价地改变一个 Global 变量
```

```
SET @@global.tidb_distsql_scan_concurrency = 10;
SET  GLOBAL tidb_distsql_scan_concurrency = 10;
```

注意：

- 在 TiDB 中，GLOBAL 变量的设置即使重启后也仍然有效。每隔 2 秒，其他 TiDB server 会获取到对变量设置的更改。详情见 TiDB #14531。
- 此外，由于应用和连接器通常需要读 MySQL 变量，为了兼容这一需求，在 TiDB 中，部分 MySQL 5.7 的变量既可读取也可设置。例如，尽管 JDBC 连接器不依赖于查询缓存 (query cache) 的行为，但仍然可以读取和设置查询缓存。

12.4.1 变量参考

12.4.1.1 autocommit

- 作用域：SESSION | GLOBAL
- 默认值：ON
- 用于设置在非显式事务时是否自动提交事务。更多信息，请参见事务概述。

12.4.1.2 allow_auto_random_explicit_insert 从 v4.0.3 版本开始引入

- 作用域：SESSION（v4.0.5 开始为 SESSION | GLOBAL）
- 默认值：0
- 是否允许在 INSERT 语句中显式指定含有 AUTO_RANDOM 属性的列的值，1 为允许，0 为不允许。

12.4.1.3 datadir

- 作用域：NONE
- 默认值：/tmp/tidb
- 这个变量表示数据存储的位置，位置可以是本地路径。如果数据存储在 TiKV 上，则可以是指向 PD 服务器的路径。
- 如果变量值的格式为 ip_address:port，表示 TiDB 在启动时连接到的 PD 服务器。

12.4.1.4 ddl_slow_threshold

- 作用域：INSTANCE
- 默认值：300
- 耗时超过该阈值的 DDL 操作会被输出到日志，单位为毫秒。

12.4.1.5 foreign_key_checks

- 作用域：NONE
- 默认值：OFF
- 为保持兼容，TiDB 对外键检查返回 OFF。

12.4.1.6 hostname

- 作用域：NONE
- 默认值：(系统主机名)
- 这个变量一个只读变量，表示 TiDB server 的主机名。

12.4.1.7 innodb_lock_wait_timeout

- 作用域：SESSION | GLOBAL
- 默认值：50
- 悲观事务语句等锁时间，单位为秒。

12.4.1.8 last_plan_from_cache 从 v4.0 版本开始引入

- 作用域：SESSION
- 默认值：0
- 这个变量用来显示上一个 `execute` 语句所使用的执行计划是不是直接从 plan cache 中取出来的。

12.4.1.9 last_plan_from_binding 从 v4.0 版本开始引入

作用域：SESSION

默认值：0

这个变量用来显示上一条执行的语句所使用的执行计划是不是来自 binding 的执行计划。

12.4.1.10 license

- 作用域：NONE
- 默认值：Apache License 2.0
- 这个变量表示 TiDB 服务器的安装许可证。

12.4.1.11 max_execution_time

- 作用域：SESSION | GLOBAL
- 默认值：0
- 语句最长执行时间，单位为毫秒。默认值 (0) 表示无限制。

> **注意：**
>
> max_execution_time 目前对所有类型的语句生效，并非只对 SELECT 语句生效，与 MySQL 不同（只对SELECT 语句生效）。实际精度在 100ms 级别，而非更准确的毫秒级别。

12.4.2 interactive_timeout

- 作用域：SESSION | GLOBAL
- 默认值：28800
- 该变量表示交互式用户会话的空闲超时，单位为秒。交互式用户会话是指使用 CLIENT_INTERACTIVE 选项调用 mysql_real_connect() API 建立的会话（例如：MySQL shell 客户端）。该变量与 MySQL 完全兼容。

12.4.3 sql_mode

- 作用域：SESSION | GLOBAL
- 默认值：ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO ↪ ,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION
- 这个变量控制许多 MySQL 兼容行为。详情见SQL 模式。

12.4.3.1 sql_select_limit 从 v4.0.2 版本开始引入

- 作用域：SESSION | GLOBAL
- 默认值：2^64 - 1 (18446744073709551615)
- SELECT 语句返回的最大行数。

12.4.3.2 tidb_allow_batch_cop 从 v4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 默认值：0
- 这个变量用于控制 TiDB 向 TiFlash 发送 coprocessor 请求的方式，有以下几种取值：
  - 0：从不批量发送请求
  - 1：aggregation 和 join 的请求会进行批量发送
  - 2：所有的 cop 请求都会批量发送

12.4.3.3 tidb_allow_remove_auto_inc 从 v2.1.18 和 v3.0.4 版本开始引入

- 作用域：SESSION
- 默认值：0
- 这个变量用来控制是否允许通过 ALTER TABLE MODIFY 或 ALTER TABLE CHANGE 来移除某个列的 AUTO_INCREMENT 属性。默认 (0) 为不允许。

12.4.3.4 tidb_auto_analyze_end_time

- 作用域：GLOBAL
- 默认值：23:59 +0000
- 这个变量用来设置一天中允许自动 ANALYZE 更新统计信息的结束时间。例如，只允许在凌晨 1:00 至 3:00 之间自动更新统计信息，可以设置如下：

```
- tidb_auto_analyze_start_time='01:00 +0000'
- tidb_auto_analyze_end_time='03:00 +0000'
```

12.4.3.5 tidb_auto_analyze_ratio

- 作用域：GLOBAL
- 默认值：0.5
- 这个变量用来设置 TiDB 在后台自动执行ANALYZE TABLE 更新统计信息的阈值。0.5 指的是当表中超过 50% 的行被修改时，触发自动 ANALYZE 更新。可以指定 tidb_auto_analyze_start_time 和 tidb_auto_analyze_end_time 来限制自动 ANALYZE 的时间

> 注意：
>
> 只有在 TiDB 的启动配置文件中开启了 run-auto-analyze 选项，该 TiDB 才会触发 auto_analyze。

12.4.3.6 tidb_auto_analyze_start_time

- 作用域：GLOBAL
- 默认值：00:00 +0000
- 这个变量用来设置一天中允许自动 ANALYZE 更新统计信息的开始时间。例如，只允许在凌晨 1:00 至 3:00 之间自动更新统计信息，可以设置如下：

```
- tidb_auto_analyze_start_time='01:00 +0000'
- tidb_auto_analyze_end_time='03:00 +0000'
```

12.4.3.7 tidb_backoff_lock_fast

- 作用域：SESSION | GLOBAL
- 默认值：100
- 这个变量用来设置读请求遇到锁的 backoff 时间。

12.4.3.8 tidb_backoff_weight

- 作用域：SESSION | GLOBAL
- 默认值：2
- 这个变量用来给 TiDB 的 backoff 最大时间增加权重，即内部遇到网络或其他组件（TiKV、PD）故障时，发送重试请求的最大重试时间。可以通过这个变量来调整最大重试时间，最小值为 1。

  例如，TiDB 向 PD 取 TSO 的基础超时时间是 15 秒，当 tidb_backoff_weight = 2 时，取 TSO 的最大超时时间为：基础时间 * 2 等于 30 秒。

  在网络环境较差的情况下，适当增大该变量值可以有效缓解因为超时而向应用端报错的情况；而如果应用端希望更快地接到报错信息，则应该尽量减小该变量的值。

12.4.3.9 tidb_capture_plan_baselines 从 v4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 默认值：off
- 这个变量用于控制是否开启自动捕获绑定功能。该功能依赖 Statement Summary，因此在使用自动绑定之前需打开 Statement Summary 开关。
- 开启该功能后会定期遍历一次 Statement Summary 中的历史 SQL 语句，并为至少出现两次的 SQL 语句自动创建绑定。

12.4.3.10 tidb_check_mb4_value_in_utf8

- 作用域：INSTANCE
- 默认值：1
- 这个变量用来设置是否开启对字符集为 UTF8 类型的数据做合法性检查，默认值 1 表示开启检查。这个默认行为和 MySQL 是兼容的。
- 如果是旧版本升级时，可能需要关闭该选项，否则由于旧版本（v2.1.1 及之前）没有对数据做合法性检查，所以旧版本写入非法字符串是可以写入成功的，但是新版本加入合法性检查后会报写入失败。具体可以参考升级后常见问题。

12.4.3.11 tidb_config

- 作用域：SESSION
- 默认值：“”
- 这个变量是一个只读变量，用来获取当前 TiDB Server 的配置信息。

12.4.3.12 tidb_constraint_check_in_place

- 作用域：SESSION | GLOBAL
- 默认值：0

- 该变量仅适用于乐观事务模型。当这个变量设置为 0 时，唯一索引的重复值检查会被推迟到事务提交时才进行。这有助于提高性能，但对于某些应用，可能导致非预期的行为。详情见约束。

  - 乐观事务模型下将 tidb_constraint_check_in_place 设置为 0：

```
create table t (i int key);
insert into t values (1);
begin optimistic;
insert into t values (1);
```

```
Query OK, 1 row affected
```

```
tidb> commit; -- 事务提交时才检查
```

```
ERROR 1062 : Duplicate entry '1' for key 'PRIMARY'
```

  - 乐观事务模型下将 tidb_constraint_check_in_place 设置为 1：

```
set @@tidb_constraint_check_in_place=1;
begin optimistic;
insert into t values (1);
```

```
ERROR 1062 : Duplicate entry '1' for key 'PRIMARY'
```

悲观事务模型中，始终默认执行约束检查。

12.4.3.13 tidb_current_ts

- 作用域：SESSION
- 默认值：0
- 这个变量是一个只读变量，用来获取当前事务的时间戳。

12.4.3.14 tidb_ddl_error_count_limit

- 作用域：GLOBAL
- 默认值：512
- 这个变量用来控制 DDL 操作失败重试的次数。失败重试次数超过该参数的值后，会取消出错的 DDL 操作。

12.4.4 tidb_ddl_reorg_batch_size

- 作用域：GLOBAL
- 默认值：256

- 这个变量用来设置 DDL 操作 `re-organize` 阶段的 batch size。比如 `ADD INDEX` 操作，需要回填索引数据，通过并发 `tidb_ddl_reorg_worker_cnt` 个 worker 一起回填数据，每个 worker 以 batch 为单位进行回填。
  - 如果 `ADD INDEX` 操作时有较多 `UPDATE` 操作或者 `REPLACE` 等更新操作，batch size 越大，事务冲突的概率也会越大，此时建议调小 batch size 的值，最小值是 32。
  - 在没有事务冲突的情况下，batch size 可设为较大值，最大值是 10240，这样回填数据的速度更快，但是 TiKV 的写入压力也会变大。

12.4.4.1 tidb_ddl_reorg_priority

- 作用域：SESSION
- 默认值：PRIORITY_LOW
- 这个变量用来设置 `ADD INDEX` 操作 `re-organize` 阶段的执行优先级，可设置为 `PRIORITY_LOW/PRIORITY_NORMAL/PRIORITY_HIGH`。

12.4.4.2 tidb_ddl_reorg_worker_cnt

- 作用域：GLOBAL
- 默认值：4
- 这个变量用来设置 DDL 操作 `re-organize` 阶段的并发度。

12.4.4.3 tidb_disable_txn_auto_retry

- 作用域：SESSION | GLOBAL
- 默认值：on
- 这个变量用来设置是否禁用显式的乐观事务自动重试，设置为 ON 时，不会自动重试，如果遇到事务冲突需要在应用层重试。

  如果将该变量的值设为 `off`，TiDB 将会自动重试事务，这样在事务提交时遇到的错误更少。需要注意的是，这样可能会导致数据更新丢失。

  这个变量不会影响自动提交的隐式事务和 TiDB 内部执行的事务，它们依旧会根据 `tidb_retry_limit` 的值来决定最大重试次数。

  关于是否需要禁用自动重试，请参考重试的局限性。

  该变量只适用于乐观事务，不适用于悲观事务。悲观事务的重试次数由`max_retry_count` 控制。

12.4.4.4 tidb_enable_amend_pessimistic_txn 从 v4.0.7 版本开始引入

- 作用域：SESSION | GLOBAL
- 默认值：0
- 这个变量用于控制是否开启 `AMEND TRANSACTION` 特性。在悲观事务模式下开启该特性后，如果该事务相关的表存在并发 DDL 操作和 SCHEMA VERSION 变更，TiDB 会尝试对该事务进行 amend 操作，修正该事务的提交内容，使其和最新的有效 SCHEMA VERSION 保持一致，从而成功提交该事务而不返回 `Information schema is changed` 报错。该特性对以下并发 DDL 变更生效：

- ADD COLUMN 或 DROP COLUMN 类型的 DDL 操作。
- MODIFY COLUMN 或 CHANGE COLUMN 类型的 DDL 操作，且只对增大字段长度的操作生效。
- ADD INDEX 或 DROP INDEX 类型的 DDL 操作，且操作的索引列须在事务开启之前创建。

> **注意：**
>
> 目前该特性可能造成事务语义的变化，且与 TiDB Binlog 存在部分不兼容的场景，可以参考事务语义行为区别和与 TiDB Binlog 兼容问题汇总了解更多关于该特性的使用注意事项。

12.4.4.5 tidb_enable_cascades_planner

> **警告：**
>
> 目前 cascades planner 为实验特性，不建议在生产环境中使用。

- 作用域：SESSION | GLOBAL
- 默认值：0
- 这个变量用于控制是否开启 cascades planner。

12.4.4.6 tidb_enable_chunk_rpc 从 v4.0 版本开始引入

- 作用域：SESSION
- 默认值：1
- 这个变量用来设置是否启用 Coprocessor 的 Chunk 数据编码格式。

12.4.4.7 tidb_enable_fast_analyze

> **警告：**
>
> 目前快速分析功能为实验特性，不建议在生产环境中使用。

- 作用域：SESSION | GLOBAL
- 默认值：0
- 这个变量用来控制是否启用统计信息快速分析功能。默认值 0 表示不开启。
- 快速分析功能开启后，TiDB 会随机采样约 10000 行的数据来构建统计信息。因此在数据分布不均匀或者数据量比较少的情况下，统计信息的准确度会比较低。这可能导致执行计划不优，比如选错索引。如果可以接受普通 ANALYZE 语句的执行时间，则推荐关闭快速分析功能。

12.4.4.8 tidb_enable_index_merge 从 v4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 默认值：0
- 这个变量用于控制是否开启 index merge 功能。

12.4.4.9 tidb_enable_noop_functions 从 v4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 默认值：0
- 这个变量用于控制是否开启 get_lock 和 release_lock 这两个没有实现的函数。需要注意的是，当前版本的 TiDB 这两个函数永远返回 1。

12.4.4.10 tidb_enable_slow_log

- 作用域：INSTANCE
- 默认值：1
- 这个变量用于控制是否开启 slow log 功能，默认开启。

12.4.4.11 tidb_enable_stmt_summary 从 v3.0.4 版本开始引入

- 作用域：SESSION | GLOBAL
- 默认值：1（受配置文件影响，这里给出的是默认配置文件取值）
- 这个变量用来控制是否开启 statement summary 功能。如果开启，SQL 的耗时等执行信息将被记录到系统表 information_schema.STATEMENTS_SUMMARY 中，用于定位和排查 SQL 性能问题。

12.4.4.12 tidb_enable_table_partition

- 作用域：SESSION | GLOBAL
- 默认值：“on”
- 这个变量用来设置是否开启 TABLE PARTITION 特性。目前变量支持以下三种值：
  - 默认值 on 表示开启 TiDB 当前已实现了的分区表类型，目前 range partition、 hash partition 以及 range column 单列的场景会生效。
  - auto 目前作用和 on 一样。
  - off 表示关闭 TABLE PARTITION 特性，此时语法还是保持兼容，只是创建的表并不是真正的分区表，而是普通的表。
- 注意，目前 TiDB 只支持 range partition 和 hash partition。

12.4.4.13 tidb_enable_telemetry 从 v4.0.2 版本开始引入

- 作用域：GLOBAL
- 默认值：1
- 这个变量用于动态地控制 TiDB 遥测功能是否开启。设置为 0 可以关闭 TiDB 遥测功能。当所有 TiDB 实例都设置enable-telemetry 为 false 时将忽略该系统变量并总是关闭 TiDB 遥测功能。参阅遥测了解该功能详情。

12.4.4.14 tidb_enable_vectorized_expression 从 v4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 默认值：1
- 这个变量用于控制是否开启向量化执行。

12.4.4.15 tidb_enable_window_function

- 作用域：SESSION | GLOBAL
- 默认值：1
- 这个变量用来控制是否开启窗口函数的支持。默认值 1 代表开启窗口函数的功能。
- 由于窗口函数会使用一些保留关键字，可能导致原先可以正常执行的 SQL 语句在升级 TiDB 后无法被解析语法，此时可以将 tidb_enable_window_function 设置为 0。

12.4.4.16 tidb_evolve_plan_baselines 从 v4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 默认值：off
- 这个变量用于控制是否启用自动演进绑定功能。该功能的详细介绍和使用方法可以参考自动演进绑定。
- 为了减少自动演进对集群的影响，可以进行以下配置：
  - 设置 tidb_evolve_plan_task_max_time，限制每个执行计划运行的最长时间，其默认值为 600s；
  - 设置tidb_evolve_plan_task_start_time 和 tidb_evolve_plan_task_end_time，限制运行演进任务的时间窗口，默认值分别为 00:00 +0000 和 23:59 +0000。

12.4.4.17 tidb_evolve_plan_task_end_time 从 v4.0 版本开始引入

- 作用域：GLOBAL
- 默认值：23:59 +0000
- 这个变量用来设置一天中允许自动演进的结束时间。

12.4.4.18 tidb_evolve_plan_task_max_time 从 v4.0 版本开始引入

- 作用域：GLOBAL
- 默认值：600
- 该变量用于限制自动演进功能中，每个执行计划运行的最长时间，单位为秒。

12.4.4.19 tidb_evolve_plan_task_start_time 从 v4.0 版本开始引入

- 作用域：GLOBAL
- 默认值：00:00 +0000
- 这个变量用来设置一天中允许自动演进的开始时间。

12.4.4.20 tidb_expensive_query_time_threshold

- 作用域：INSTANCE
- 默认值：60
- 这个变量用来控制打印 expensive query 日志的阈值时间，单位是秒，默认值是 60 秒。expensive query 日志和慢日志的差别是，慢日志是在语句执行完后才打印，expensive query 日志可以把正在执行中的语句且执行时间超过阈值的语句及其相关信息打印出来。

12.4.4.21 tidb_force_priority

- 作用域：INSTANCE
- 默认值：NO_PRIORITY
- 这个变量用于改变 TiDB server 上执行的语句的默认优先级。例如，你可以通过设置该变量来确保正在执行 OLAP 查询的用户优先级低于正在执行 OLTP 查询的用户。
- 可设置为 NO_PRIORITY、 LOW_PRIORITY、 DELAYED 或 HIGH_PRIORITY。

12.4.4.22 tidb_general_log

- 作用域：INSTANCE
- 默认值：0
- 这个变量用来设置是否在日志里记录所有的 SQL 语句。该功能默认关闭。如果系统运维人员在定位问题过程中需要追踪所有 SQL 记录，可考虑开启该功能。
- 通过查询 "GENERAL_LOG" 字符串可以定位到该功能在日志中的所有记录。日志会记录以下内容：
  - conn：当前会话对应的 ID
  - user：当前会话用户
  - schemaVersion：当前 schema 版本
  - txnStartTS：当前事务的开始时间戳
  - forUpdateTS：事务模型为悲观事务时，SQL 语句的当前时间戳。悲观事务内发生写冲突时，会重试当前执行语句，该时间戳会被更新。重试次数由max-retry-count 配置。事务模型为乐观事务时，该条目与 txnStartTS 等价。
  - isReadConsistency：当前事务隔离级别是否是读已提交 (RC)
  - current_db：当前数据库名
  - txn_mode：事务模型。可选值：OPTIMISTIC（乐观事务模型），或 PESSIMISTIC（悲观事务模型）
  - sql：当前查询对应的 SQL 语句

12.4.4.23 tidb_build_stats_concurrency

- 作用域：SESSION
- 默认值：4
- 这个变量用来设置 ANALYZE 语句执行时并发度。
- 当这个变量被设置得更大时，会对其它的查询语句执行性能产生一定影响。

12.4.4.24 tidb_checksum_table_concurrency

- 作用域：SESSION
- 默认值：4
- 这个变量用来设置 ADMIN CHECKSUM TABLE 语句执行时扫描索引的并发度。当这个变量被设置得更大时，会对其它的查询语句执行性能产生一定影响。

12.4.4.25 tidb_distsql_scan_concurrency

- 作用域：SESSION | GLOBAL
- 默认值：15
- 这个变量用来设置 scan 操作的并发度。
- AP 类应用适合较大的值，TP 类应用适合较小的值。对于 AP 类应用，最大值建议不要超过所有 TiKV 节点的 CPU 核数。
- 若表的分区较多可以适当调小该参数，避免 TiKV 内存溢出 (OOM)。

12.4.4.26 tidb_hash_join_concurrency

- 作用域：SESSION | GLOBAL
- 默认值：5
- 这个变量用来设置 hash join 算法的并发度。

12.4.4.27 tidb_hashagg_final_concurrency

- 作用域：SESSION | GLOBAL
- 默认值：4
- 这个变量用来设置并行 hash aggregation 算法 final 阶段的执行并发度。对于聚合函数参数不为 distinct 的情况，HashAgg 分为 partial 和 final 阶段分别并行执行。

12.4.4.28 tidb_hashagg_partial_concurrency

- 作用域：SESSION | GLOBAL
- 默认值：4
- 这个变量用来设置并行 hash aggregation 算法 partial 阶段的执行并发度。对于聚合函数参数不为 distinct 的情况，HashAgg 分为 partial 和 final 阶段分别并行执行。

12.4.4.29 tidb_index_join_batch_size

- 作用域：SESSION | GLOBAL
- 默认值：25000
- 这个变量用来设置 index lookup join 操作的 batch 大小，AP 类应用适合较大的值，TP 类应用适合较小的值。

12.4.4.30 tidb_index_lookup_concurrency

- 作用域：SESSION | GLOBAL
- 默认值：4
- 这个变量用来设置 index lookup 操作的并发度，AP 类应用适合较大的值，TP 类应用适合较小的值。

12.4.4.31 tidb_index_lookup_join_concurrency

- 作用域：SESSION | GLOBAL
- 默认值：4
- 这个变量用来设置 index lookup join 算法的并发度。

12.4.4.32 tidb_index_lookup_size

- 作用域：SESSION | GLOBAL
- 默认值：20000
- 这个变量用来设置 index lookup 操作的 batch 大小，AP 类应用适合较大的值，TP 类应用适合较小的值。

12.4.4.33 tidb_index_serial_scan_concurrency

- 作用域：SESSION | GLOBAL
- 默认值：1
- 这个变量用来设置顺序 scan 操作的并发度，AP 类应用适合较大的值，TP 类应用适合较小的值。

12.4.4.34 tidb_projection_concurrency

- 作用域：SESSION | GLOBAL
- 默认值：4
- 这个变量用来设置 Projection 算子的并发度。

12.4.4.35 tidb_window_concurrency 从 v4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 默认值：4
- 这个变量用于设置 window 算子的并发度。

12.4.4.36 tidb_union_concurrency

- 作用域：SESSION | GLOBAL
- 默认值：4
- 这个变量用于设置 union 算子的并发度。

12.4.4.37 tidb_init_chunk_size

- 作用域：SESSION | GLOBAL
- 默认值：32
- 这个变量用来设置执行过程中初始 chunk 的行数。默认值是 32，可设置的范围是 1 ~ 32。

12.4.4.38 tidb_isolation_read_engines 从 v4.0 版本开始引入

- 作用域：SESSION
- 默认值：tikv, tiflash, tidb
- 这个变量用于设置 TiDB 在读取数据时可以使用的存储引擎列表。

12.4.4.39 tidb_low_resolution_tso

- 作用域：SESSION
- 默认值：0
- 这个变量用来设置是否启用低精度 tso 特性，开启该功能之后新事务会使用一个每 2s 更新的 ts 来读取数据。
- 主要场景是在可以容忍读到旧数据的情况下，降低小的只读事务获取 tso 的开销。

12.4.4.40 tidb_max_chunk_size

- 作用域：SESSION | GLOBAL
- 默认值：1024
- 最小值：32
- 这个变量用来设置执行过程中一个 chunk 最大的行数，设置过大可能引起缓存局部性的问题。

12.4.4.41 tidb_max_delta_schema_count

- 作用域：GLOBAL
- 默认值：1024
- 这个变量用来设置缓存 schema 版本信息（对应版本修改的相关 table IDs）的个数限制，可设置的范围 100 - 16384。此变量在 2.1.18 及之后版本支持。

12.4.4.42 tidb_mem_quota_query

- 作用域：SESSION
- 默认值：1 GB
- 这个变量用来设置一条查询语句的内存使用阈值。
- 如果一条查询语句执行过程中使用的内存空间超过该阈值，会触发 TiDB 启动配置文件中 OOMAction 项所指定的行为。该变量的初始值由配置项mem-quota-query 配置。

12.4.4.43 tidb_memory_usage_alarm_ratio

- 作用域：INSTANCE
- 默认值：0.8
- TiDB 内存使用占总内存的比例超过一定阈值时会报警。该功能的详细介绍和使用方法可以参考memory-usage-alarm-ratio。
- 该变量的初始值可通过memory-usage-alarm-ratio 进行配置。

12.4.4.44 tidb_metric_query_range_duration 从 v4.0 版本开始引入

- 作用域：SESSION
- 默认值：60
- 这个变量设置了查询 METRIC_SCHEMA 时生成的 Prometheus 语句的 range duration，单位为秒。

12.4.4.45 tidb_metric_query_step 从 v4.0 版本开始引入

- 作用域：SESSION
- 默认值：60
- 这个变量设置了查询 METRIC_SCHEMA 时生成的 Prometheus 语句的 step，单位为秒。

12.4.4.46 tidb_multi_statement_mode 从 v4.0.11 版本开始引入

- 作用域：SESSION | GLOBAL
- 默认值：从 v4.0.14 起，默认值为 OFF。对于 v4.0.14 前的 v4.0 版本，默认值为 WARN。
- 可选值：OFF，ON 和 WARN
- 该变量用于控制是否在同一个 COM_QUERY 调用中执行多个查询。
- 为了减少 SQL 注入攻击的影响，TiDB 目前默认不允许在同一 COM_QUERY 调用中执行多个查询。该变量可用作早期 TiDB 版本的升级路径选项。该变量值与是否允许多语句行为的对照表如下：

客户端设置	tidb_multi_statement_mode 值	是否允许多语句
Multiple Statements = ON	OFF	允许
Multiple Statements = ON	ON	允许
Multiple Statements = ON	WARN	允许
Multiple Statements = OFF	OFF	不允许
Multiple Statements = OFF	ON	允许
Multiple Statements = OFF	WARN	允许 + 警告提示

注意：

只有默认值 OFF 才是安全的。如果用户业务是专为早期 TiDB 版本而设计的，那么需要将该变量值设为 ON。如果用户业务需要多语句支持，建议用户使用客户端提供的设置，不要使用 tidb_multi_statement_mode 变量进行设置。

- go-sql-driver (multiStatements)
- Connector/J (allowMultiQueries)
- PHP mysqli (mysqli_multi_query)

12.4.4.47 tidb_opt_agg_push_down

- 作用域：SESSION
- 默认值：0
- 这个变量用来设置优化器是否执行聚合函数下推到 Join，Projection 和 UnionAll 之前的优化操作。当查询中聚合操作执行很慢时，可以尝试设置该变量为 1。

12.4.4.48 tidb_opt_correlation_exp_factor

- 作用域：SESSION | GLOBAL
- 默认值：1
- 当交叉估算方法不可用时，会采用启发式估算方法。这个变量用来控制启发式方法的行为。当值为 0 时不用启发式估算方法，大于 0 时，该变量值越大，启发式估算方法越倾向 index scan，越小越倾向 table scan。

12.4.4.49 tidb_opt_correlation_threshold

- 作用域：SESSION | GLOBAL
- 默认值：0.9
- 这个变量用来设置优化器启用交叉估算 row count 方法的阈值。如果列和 handle 列之间的顺序相关性超过这个阈值，就会启用交叉估算方法。
- 交叉估算方法可以简单理解为，利用这个列的直方图来估算 handle 列需要扫的行数。

12.4.4.50 tidb_opt_distinct_agg_push_down

- 作用域：SESSION
- 默认值：0

- 这个变量用来设置优化器是否执行带有 Distinct 的聚合函数（比如 select count(distinct a)from t）下推到 Coprocessor 的优化操作。当查询中带有 Distinct 的聚合操作执行很慢时，可以尝试设置该变量为 1。

在以下示例中，tidb_opt_distinct_agg_push_down 开启前，TiDB 需要从 TiKV 读取所有数据，并在 TiDB 侧执行 disctinct。tidb_opt_distinct_agg_push_down 开启后，distinct a 被下推到了 Coprocessor，在 HashAgg_5 里新增里一个 group by 列 test.t.a。

```
mysql> desc select count(distinct a) from test.t;
+--
    ↪ ----------------------+----------+-----------+---------------+--------------------------------
    ↪
| id                      | estRows  | task      | access object | operator info
    ↪                           |
+--
    ↪ ----------------------+----------+-----------+---------------+--------------------------------
    ↪
| StreamAgg_6             | 1.00     | root      |               | funcs:count(distinct test.t.a)
    ↪ ->Column#4 |
| └─TableReader_10        | 10000.00 | root      |               | data:TableFullScan_9
    ↪                      |
|   └─TableFullScan_9     | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo           |
+--
    ↪ ----------------------+----------+-----------+---------------+--------------------------------
    ↪
3 rows in set (0.01 sec)

mysql> set session tidb_opt_distinct_agg_push_down = 1;
Query OK, 0 rows affected (0.00 sec)

mysql> desc select count(distinct a) from test.t;
+--
    ↪ ------------------------+----------+-----------+---------------+--------------------------------
    ↪
| id                        | estRows  | task      | access object | operator info
    ↪                           |
+--
    ↪ ------------------------+----------+-----------+---------------+--------------------------------
    ↪
| HashAgg_8                 | 1.00     | root      |               | funcs:count(distinct test.t.
    ↪ a)->Column#3 |
| └─TableReader_9           | 1.00     | root      |               | data:HashAgg_5
    ↪                           |
|   └─HashAgg_5             | 1.00     | cop[tikv] |               | group by:test.t.a,
    ↪                         |
```

```
|     └─TableFullScan_7     | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo           |
+--
    ↪ ------------------------+----------+-----------+---------------+-------------------------------------------------
    ↪
4 rows in set (0.00 sec)
```

12.4.4.51 tidb_opt_insubq_to_join_and_agg

- 作用域：SESSION | GLOBAL
- 默认值：1
- 这个变量用来设置是否开启优化规则：将子查询转成 join 和 aggregation。

  例如，打开这个优化规则后，会将下面子查询做如下变化：

```
select * from t where t.a in (select aa from t1);
```

  将子查询转成如下 join：

```
select t.* from t, (select aa from t1 group by aa) tmp_t where t.a = tmp_t.aa;
```

  如果 t1 在列 aa 上有 unique 且 not null 的限制，可以直接改写为如下，不需要添加 aggregation。

```
select t.* from t, t1 where t.a=t1.aa;
```

12.4.4.52 tidb_opt_write_row_id

- 作用域：SESSION
- 默认值：0
- 这个变量用来设置是否允许 INSERT、REPLACE 和 UPDATE 操作 _tidb_rowid 列，默认是不允许操作。该选项仅用于 TiDB 工具导数据时使用。

12.4.4.53 tidb_query_log_max_len

- 作用域：INSTANCE
- 默认值：4096 (bytes)
- 最长的 SQL 输出长度。当语句的长度大于 query-log-max-len，将会被截断输出。

示例：

```
set tidb_query_log_max_len = 20;
```

12.4.4.54 tidb_pprof_sql_cpu 从 v4.0 版本开始引入

- 作用域：INSTANCE
- 默认值：0
- 这个变量用来控制是否在 profile 输出中标记出对应的 SQL 语句，用于定位和排查性能问题。

12.4.4.55 tidb_record_plan_in_slow_log

- 作用域：INSTANCE
- 默认值：1
- 这个变量用于控制是否在 slow log 里包含慢查询的执行计划。

12.4.4.56 tidb_replica_read 从 v4.0 版本开始引入

- 作用域：SESSION
- 默认值：leader
- 这个变量用于控制 TiDB 读取数据的位置，有以下三个选择：
  - leader：只从 leader 节点读取
  - follower：只从 follower 节点读取
  - leader-and-follower：从 leader 或 follower 节点读取

更多细节，见Follower Read。

12.4.4.57 tidb_retry_limit

- 作用域：SESSION | GLOBAL
- 默认值：10
- 这个变量用来设置乐观事务的最大重试次数。一个事务执行中遇到可重试的错误（例如事务冲突、事务提交过慢或表结构变更）时，会根据该变量的设置进行重试。注意当 `tidb_retry_limit = 0` 时，也会禁用自动重试。该变量仅适用于乐观事务，不适用于悲观事务。

12.4.4.58 tidb_row_format_version

- 作用域：GLOBAL
- 默认值：2
- 控制新保存数据的表数据格式版本。TiDB v4.0 中默认使用版本号为 2 的新表数据格式保存新数据。
- 但如果从 4.0.0 之前的版本升级到 4.0.0，不会改变表数据格式版本，TiDB 会继续使用版本为 1 的旧格式写入表中，即只有新创建的集群才会默认使用新表数据格式。
- 需要注意的是修改该变量不会对已保存的老数据产生影响，只会对修改变量后的新写入数据使用对应版本格式保存。

12.4.4.59 tidb_scatter_region

- 作用域：GLOBAL
- 默认值：OFF
- TiDB 默认会在建表时为新表分裂 Region。开启该变量后，会在建表语句执行时，同步打散刚分裂出的 Region。适用于批量建表后紧接着批量写入数据，能让刚分裂出的 Region 先在 TiKV 分散而不用等待 PD 进行调度。为了保证后续批量写入数据的稳定性，建表语句会等待打散 Region 完成后再返回建表成功，建表语句执行时间会是该变量关闭时的数倍。
- 如果建表时设置了 SHARD_ROW_ID_BITS 和 PRE_SPLIT_REGIONS，建表成功后会均匀切分出指定数量的 Region。
- 默认值：0
- TiDB 默认会在建表时为新表分裂 Region。开启该变量后，会在建表语句执行时，同步打散刚分裂出的 Region。适用于批量建表后紧接着批量写入数据，能让刚分裂出的 Region 先在 TiKV 分散而不用等待 PD 进行调度。为了保证后续批量写入数据的稳定性，建表语句会等待打散 Region 完成后再返回建表成功，建表语句执行时间会是关闭该变量的数倍。

12.4.4.60 tidb_skip_isolation_level_check

- 作用域：SESSION | GLOBAL
- 默认值：0
- 开启这个开关之后，如果对 tx_isolation 赋值一个 TiDB 不支持的隔离级别，不会报错，有助于兼容其他设置了（但不依赖于）不同隔离级别的应用。

```
tidb> set tx_isolation='serializable';
ERROR 8048 (HY000): The isolation level 'serializable' is not supported. Set
    ↪ tidb_skip_isolation_level_check=1 to skip this error
tidb> set tidb_skip_isolation_level_check=1;
Query OK, 0 rows affected (0.00 sec)

tidb> set tx_isolation='serializable';
Query OK, 0 rows affected, 1 warning (0.00 sec)
```

12.4.4.61 tidb_skip_utf8_check

- 作用域：SESSION | GLOBAL
- 默认值：0
- 这个变量用来设置是否跳过 UTF-8 字符的验证。
- 验证 UTF-8 字符需要消耗一定的性能，当可以确认输入的字符串为有效的 UTF-8 字符时，可以将其设置为 1。

12.4.4.62 tidb_slow_log_threshold

- 作用域：INSTANCE
- 默认值：300
- 输出慢日志的耗时阈值。当查询大于这个值，就会当做是一个慢查询，输出到慢查询日志。默认为 300 ms。

示例：

```
set tidb_slow_log_threshold = 200;
```

12.4.4.63 tidb_enable_collect_execution_info

- 作用域：INSTANCE
- 默认值：1
- 这个变量用于控制是否同时将各个执行算子的执行信息记录入 slow query log 中。

12.4.4.64 tidb_redact_log

- 作用域：SESSION | GLOBAL
- 默认值：0
- 这个变量用于控制在记录 TiDB 日志和慢日志时，是否将 SQL 中的用户信息遮蔽。
- 将该变量设置为 1 即开启后，假设执行的 SQL 为 `insert into t values (1,2)`，在日志中记录的 SQL 会是 `insert into t values (?,?)`，即用户输入的信息被遮蔽。

12.4.4.65 tidb_slow_query_file

- 作用域：SESSION
- 默认值：“”
- 查询 INFORMATION_SCHEMA.SLOW_QUERY 只会解析配置文件中 `slow-query-file` 设置的慢日志文件名，默认是“tidb-slow.log”。但如果想要解析其他的日志文件，可以通过设置 session 变量 `tidb_slow_query_file` 为具体的文件路径，然后查询 INFORMATION_SCHEMA.SLOW_QUERY 就会按照设置的路径去解析慢日志文件。更多详情可以参考SLOW_QUERY 文档。

12.4.4.66 tidb_snapshot

- 作用域：SESSION
- 默认值：“”
- 这个变量用来设置当前会话期待读取的历史数据所处时刻。比如当设置为 `"2017-11-11 20:20:20"` 时或者一个 TSO 数字“400036290571534337”，当前会话将能读取到该时刻的数据。

12.4.4.67 tidb_stmt_summary_history_size 从 v4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 默认值：24（受配置文件影响，这里给出的是默认配置文件取值）
- 这个变量设置了statement summary tables 的历史记录容量。

12.4.4.68 tidb_stmt_summary_internal_query 从 v4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 默认值：0（受配置文件影响，这里给出的是默认配置文件取值）
- 这个变量用来控制是否在statement summary tables 中包含 TiDB 内部 SQL 的信息。

12.4.4.69 tidb_stmt_summary_max_sql_length 从 v4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 默认值：4096（受配置文件影响，这里给出的是默认配置文件取值）
- 这个变量控制statement summary tables 显示的 SQL 字符串长度。

12.4.4.70 tidb_stmt_summary_max_stmt_count 从 v4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 默认值：v4.0.14 前为 200。自 v4.0.14 起为 3000（受配置文件影响，这里给出的是默认配置文件取值）
- 这个变量设置了statement summary tables 在内存中保存的语句的最大数量。

12.4.4.71 tidb_stmt_summary_refresh_interval 从 v4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 默认值：1800（受配置文件影响，这里给出的是默认配置文件取值）
- 这个变量设置了statement summary tables 的刷新时间，单位为秒。

12.4.4.72 tidb_store_limit 从 v3.0.4 和 v4.0 版本开始引入

- 作用域：INSTANCE | GLOBAL
- 默认值：0
- 这个变量用于限制 TiDB 同时向 TiKV 发送的请求的最大数量，0 表示没有限制。

12.4.4.73 tidb_txn_mode

- 作用域：SESSION | GLOBAL
- 默认值：“pessimistic”
- 这个变量用于设置事务模式。TiDB v3.0 支持了悲观事务，自 v3.0.8 开始，默认使用悲观事务模式。
- 但如果从 3.0.7 及之前的版本升级到 >= 3.0.8 的版本，不会改变默认事务模型，即只有新创建的集群才会默认使用悲观事务模型。
- 将该变量设置为“optimistic”或“”时，将会使用乐观事务模式。

12.4.4.74 tidb_use_plan_baselines 从 v4.0 版本开始引入

- 作用域：SESSION | GLOBAL
- 默认值：on
- 这个变量用于控制是否开启执行计划绑定功能，默认打开，可通过赋值 off 来关闭。关于执行计划绑定功能的使用可以参考执行计划绑定文档。

12.4.4.75 tidb_wait_split_region_finish

- 作用域：SESSION
- 默认值：1
- 由于打散 region 的时间可能比较长，主要由 PD 调度以及 TiKV 的负载情况所决定。这个变量用来设置在执行 SPLIT REGION 语句时，是否同步等待所有 region 都打散完成后再返回结果给客户端。默认 1 代表等待打散完成后再返回结果。0 代表不等待 Region 打散完成就返回。
- 需要注意的是，在 region 打散期间，对正在打散 region 上的写入和读取的性能会有一定影响，对于批量写入，导数据等场景，还是建议等待 region 打散完成后再开始导数据。

12.4.4.76 tidb_wait_split_region_timeout

- 作用域：SESSION
- 默认值：300
- 这个变量用来设置 SPLIT REGION 语句的执行超时时间，单位是秒，默认值是 300 秒，如果超时还未完成，就返回一个超时错误。

12.4.4.77 time_zone

- 作用域：SESSION | GLOBAL
- 默认值：SYSTEM
- 数据库所使用的时区。这个变量值可以写成时区偏移的形式，如 ‘-8:00’，也可以写成一个命名时区，如 ‘America/Los_Angeles’。

12.4.4.78 transaction_isolation

- 作用域：SESSION | GLOBAL
- 默认值：REPEATABLE-READ
- 这个变量用于设置事务隔离级别。TiDB 为了兼容 MySQL，支持可重复读 (REPEATABLE-READ)，但实际的隔离级别是快照隔离。详情见事务隔离级别。

12.4.4.79 tx_isolation

这个变量是 transaction_isolation 的别名。

12.4.4.80 version

- 作用域：NONE
- 默认值：5.7.25-TiDB-(tidb version)
- 这个变量的值是 MySQL 的版本和 TiDB 的版本，例如 ‘5.7.25-TiDB-v4.0.0-beta.2-716-g25e003253’。

12.4.4.81 version_comment

- 作用域：NONE
- 默认值：(string)
- 这个变量的值是 TiDB 版本号的其他信息，例如 ‘TiDB Server (Apache License 2.0) Community Edition, MySQL 5.7 compatible’。

12.4.4.82 wait_timeout

- 作用域：SESSION | GLOBAL
- 默认值：0
- 这个变量表示用户会话的空闲超时，单位为秒。0 代表没有时间限制。

12.4.4.83 warning_count

- 作用域: SESSION
- 默认值：0
- 这个只读变量表示之前执行语句中出现的警告数。

12.4.4.84 windowing_use_high_precision

- 作用域：SESSION | GLOBAL
- 默认值：ON
- 这个变量用于控制计算窗口函数时是否采用高精度模式。

12.4.4.85 tidb_enable_rate_limit_action

> 注意：
>
> 该变量默认开启，但可能导致内存不受tidb_mem_quota_query 控制，从而加剧 OOM 风险，因此建议将该变量值调整为 OFF。

- 作用域：SESSION | GLOBAL
- 默认值：ON
- 这个变量控制是否为读数据的算子开启动态内存控制功能。读数据的算子默认启用tidb_disql_scan_concurrency 所允许的最大线程数来读取数据。当单条 SQL 语句的内存使用每超过tidb_mem_quota_query 一次，读数据的算子会停止一个线程。
- 当读数据的算子只剩 1 个线程且当单条 SQL 语句的内存使用继续超过tidb_mem_quota_query 时，该 SQL 语句会触发其它的内存控制行为。

12.5 配置文件参数

12.5.1 TiDB 配置文件描述

TiDB 配置文件比命令行参数支持更多的选项。你可以在 config/config.toml.example 找到默认值的配置文件，重命名为 config.toml 即可。本文档只介绍未包含在命令行参数中的参数。

12.5.1.0.1 split-table

- 为每个 table 建立单独的 Region。
- 默认值：true
- 如果需要创建大量的表，我们建议把这个参数设置为 false。

12.5.1.0.2 token-limit

- 可以同时执行请求的 session 个数
- 默认值：1000

12.5.1.0.3 mem-quota-query

- 单条 SQL 语句可以占用的最大内存阈值，单位为字节。
- 默认值：1073741824
- 注意：当集群从 v2.0.x 或 v3.0.x 版本直接升级至 v4.0.9 及以上版本时，该配置默认值为 34359738368。
- 超过该值的请求会被 oom-action 定义的行为所处理。
- 该值作为系统变量tidb_mem_quota_query 的初始值。

12.5.1.0.4 oom-use-tmp-storage

- 设置是否在单条 SQL 语句的内存使用超出 mem-quota-query 限制时为某些算子启用临时磁盘。
- 默认值：true

12.5.1.0.5 tmp-storage-path

- 单条 SQL 语句的内存使用超出 mem-quota-query 限制时，某些算子的临时磁盘存储位置。
- 默认值：<操作系统临时文件夹>/<操作系统用户ID>_tidb/MC4wLjAuMDo0MDAwLzAuMC4wLjA6MTAwODA=/tmp-storage。其中 MC4wLjAuMDo0MDAwLzAuMC4wLjA6MTAwODA= 是对 <host>:<port>/<statusHost>:<statusPort> 进行 Base64 编码的输出结果。
- 此配置仅在 oom-use-tmp-storage 为 true 时有效。

12.5.1.0.6 tmp-storage-quota

- tmp-storage-path 存储使用的限额，单位为字节。
- 当单条 SQL 语句使用临时磁盘，导致 TiDB server 的总体临时磁盘总量超过 tmp-storage-quota 时，当前 SQL 操作会被取消，并返回 Out Of Global Storage Quota! 错误。
- 当 tmp-storage-quota 小于 0 时则没有上述检查与限制。
- 默认值: -1
- 当 tmp-storage-path 的剩余可用容量低于 tmp-storage-quota 所定义的值时，TiDB server 启动时将会报出错误并退出。

12.5.1.0.7 oom-action

> **警告：**
>
> 目前 oom-action 为实验功能，会对写入过程中的内存进行统计。如果用户希望根据该特性取消写入操作，不建议在生产环境中将参数值配置为 cancel。

- 当 TiDB 中单条 SQL 的内存使用超出 mem-quota-query 限制且不能再利用临时磁盘时的行为。
- 默认值：“log”
- 目前合法的选项为 [“log”，“cancel”]。设置为 “log” 时，仅输出日志。设置为 “cancel” 时，取消执行该 SQL 操作，并输出日志。

12.5.1.0.8 lower-case-table-names

- 这个选项可以设置 TiDB 的系统变量 lower-case-table-names 的值。
- 默认值：2
- 具体可以查看 MySQL 关于这个变量的描述

> **注意：**
>
> 目前 TiDB 只支持将该选项的值设为 2，即按照大小写来保存表名，按照小写来比较（不区分大小写）。

12.5.1.0.9 lease

- DDL 租约超时时间。
- 默认值：45s
- 单位：秒

12.5.1.0.10 compatible-kill-query

- 设置 KILL 语句的兼容性。
- 默认值：false
- TiDB 中 KILL xxx 的行为和 MySQL 中的行为不相同。为杀死一条查询，在 TiDB 里需要加上 TIDB 关键词，即 KILL TIDB xxx。但如果把 compatible-kill-query 设置为 true，则不需要加上 TIDB 关键词。
- 这种区别很重要，因为当用户按下 Ctrl+C 时，MySQL 命令行客户端的默认行为是：创建与后台的新连接，并在该新连接中执行 KILL 语句。如果负载均衡器或代理已将该新连接发送到与原始会话不同的 TiDB 服务器实例，则该错误会话可能被终止，从而导致使用 TiDB 集群的业务中断。只有当您确定在 KILL 语句中引用的连接正好位于 KILL 语句发送到的服务器上时，才可以启用 compatible-kill-query。

12.5.1.0.11 check-mb4-value-in-utf8

- 开启检查 utf8mb4 字符的开关，如果开启此功能，字符集是 utf8，且在 utf8 插入 mb4 字符，系统将会报错。
- 默认值：true

12.5.1.0.12 treat-old-version-utf8-as-utf8mb4

- 将旧表中的 utf8 字符集当成 utf8mb4 的开关。
- 默认值：true

12.5.1.0.13 alter-primary-key

- 用于控制添加或者删除主键功能。
- 默认值：false
- 默认情况下，不支持增删主键。将此变量被设置为 true 后，支持增删主键功能。不过对在此开关开启前已经存在的表，且主键是整型类型时，即使之后开启此开关也不支持对此列表删除主键。

12.5.1.0.14 server-version

- 用来修改 TiDB 在以下情况下返回的版本号:
  - 当使用内置函数 VERSION() 时。
  - 当与客户端初始连接，TiDB 返回带有服务端版本号的初始握手包时。具体可以查看 MySQL 初始握手包的描述。
- 默认值：“”
- 默认情况下，TiDB 版本号格式为：5.7.${mysql_latest_minor_version}-TiDB-${tidb_version}。

12.5.1.0.15 repair-mode

- 用于开启非可信修复模式，启动该模式后，可以过滤 repair-table-list 名单中坏表的加载。
- 默认值：false
- 默认情况下，不支持修复语法，默认启动时会加载所有表信息。

12.5.1.0.16 repair-table-list

- 配合 repair-mode 为 true 时使用，用于列出实例中需要修复的坏表的名单，该名单的写法为 [“db.table1”,“db.table2”…]。
- 默认值：[]
- 默认情况下，该 list 名单为空，表示没有所需修复的坏表信息。

12.5.1.0.17 new_collations_enabled_on_first_bootstrap

- 用于开启新的 collation 支持
- 默认值：false
- 注意：该配置项只有在初次初始化集群时生效，初始化集群后，无法通过更改该配置项打开或关闭新的 collation 框架；4.0 版本之前的 TiDB 集群升级到 4.0 时，由于集群已经初始化过，该参数无论如何配置，都作为 false 处理。

12.5.1.0.18 max-server-connections

- TiDB 中同时允许的最大客户端连接数，用于资源控制。
- 默认值：0
- 默认情况下，TiDB 不限制客户端连接数。当本配置项的值大于 0 且客户端连接数到达此值时，TiDB 服务端将会拒绝新的客户端连接。

12.5.1.0.19 max-index-length

- 用于设置新建索引的长度限制。
- 默认值：3072
- 单位：byte。

- 目前的合法值范围 [3072，3072*4]。MySQL 和 TiDB v3.0.11 之前版本（不包含 v3.0.11）没有此配置项，不过都对新建索引的长度做了限制。MySQL 对此的长度限制为 3072，TiDB 在 v3.0.7 以及之前版本该值为 3072*4，在 v3.0.7 之后版本（包含 v3.0.8、v3.0.9 和 v3.0.10）的该值为 3072。为了与 MySQL 和 TiDB 之前版本的兼容，添加了此配置项。

12.5.1.0.20 `enable-telemetry` 从 v4.0.2 版本开始引入

- 是否开启 TiDB 遥测功能。
- 默认值：true
- 如果所有 TiDB 实例上该选项都设置为 `false`，那么将完全禁用 TiDB 遥测功能，且忽略`tidb_enable_telemetry` ↪ 系统变量。参阅遥测了解该功能详情。

12.5.1.0.21 `enable-table-lock` 从 v4.0.0 版本开始引入

> **警告：**
>
> 表级锁 (Table Lock) 为实验特性，不建议在生产环境中使用。

- 控制是否开启表级锁特性。
- 默认值：false
- 表级锁用于协调多个 session 之间对同一张表的并发访问。目前已支持的锁种类包括 `READ`、`WRITE` 和 `WRITE LOCAL`。当该配置项为 `false` 时，执行 `LOCK TABLE` 和 `UNLOCK TABLE` 语句不会生效，并且会报“LOCK/UNLOCK TABLES is not supported”的警告。

12.5.1.1 log

日志相关的配置项。

12.5.1.1.1 `level`

- 指定日志的输出级别, 可选项为 [debug, info, warn, error, fatal]
- 默认值：“info”

12.5.1.1.2 `format`

- 指定日志输出的格式，可选项为 [json, text, console]。
- 默认值：“text”

12.5.1.1.3 enable-timestamp

- 是否在日志中输出时间戳。
- 默认值：true
- 如果设置为 false，那么日志里面将不会输出时间戳。

> **注意：**
>
> 考虑后向兼容性，原来的配置项 `disable-timestamp` 仍然有效，但如果和 `enable-timestamp` 配置的值在语义上冲突（例如在配置中把 `enable-timestamp` 和 `disable-timestamp` 同时设置为 `true`），则 TiDB 会忽略 `disable-timestamp` 的值。在未来的版本中，`disable-timestamp` 配置项将被彻底移除，请废弃 `disable-timestamp` 的用法，使用语义上更易于理解的 `enable-timestamp`。

12.5.1.1.4 enable-slow-log

- 是否开启慢查询日志
- 默认值：true
- 可以设置成 `true` 或 `false` 来启用或或禁用慢查询日志。

12.5.1.1.5 slow-query-file

- 慢查询日志的文件名。
- 默认值：“tidb-slow.log”，注：由于 TiDB V2.1.8 更新了慢日志格式，所以将慢日志单独输出到了慢日志文件。V2.1.8 之前的版本，该变量的默认值是 “”。
- 设置后，慢查询日志会单独输出到该文件。

12.5.1.1.6 slow-threshold

- 输出慢日志的耗时阈值。
- 默认值：300ms
- 当查询大于这个值，就会当做是一个慢查询，输出到慢查询日志。

12.5.1.1.7 record-plan-in-slow-log

- 在慢日志中记录执行计划
- 默认值：1
- 0 表示关闭，1 表示开启，默认开启，该值作为系统变量`tidb_record_plan_in_slow_log` 的初始值。

12.5.1.1.8 expensive-threshold

- 输出 `expensive` 操作的行数阈值。
- 默认值：10000
- 当查询的行数（包括中间结果，基于统计信息）大于这个值，我们就会当成是一个 `expensive` 的操作，输出一个前缀带有 `[EXPENSIVE_QUERY]` 的日志。

12.5.1.1.9 query-log-max-len

- 最长的 SQL 输出长度。
- 默认值：4096
- 当语句的长度大于 query-log-max-len，将会被截断输出。

12.5.1.2 log.file

日志文件相关的配置项。

filename

- 一般日志文件名字。
- 默认值：“”
- 如果设置，会输出一般日志到这个文件。

max-size

- 日志文件的大小限制。
- 默认值：300
- 单位：MB
- 最大设置上限为 4096。

max-days

- 日志最大保留的天数。
- 默认值：0
- 默认不清理；如果设置了参数值，在 max-days 之后 TiDB 会清理过期的日志文件。

max-backups

- 保留的日志的最大数量。
- 默认值：0
- 默认全部保存；如果设置为 7，会最多保留 7 个老的日志文件。

12.5.1.3 security

安全相关配置。

12.5.1.3.1 ssl-ca

- PEM 格式的受信任 CA 的证书文件路径。
- 默认值：“”
- 当同时设置了该选项和 --ssl-cert、--ssl-key 选项时，TiDB 将在客户端出示证书的情况下根据该选项指定的受信任的 CA 列表验证客户端证书。若验证失败，则连接会被终止。
- 即使设置了该选项，若客户端没有出示证书，则安全连接仍然继续，不会进行客户端证书验证。

12.5.1.3.2 ssl-cert

- PEM 格式的 SSL 证书文件路径。
- 默认值：“”
- 当同时设置了该选项和 `--ssl-key` 选项时，TiDB 将接受（但不强制）客户端使用 TLS 安全地连接到 TiDB。
- 若指定的证书或私钥无效，则 TiDB 会照常启动，但无法接受安全连接。

12.5.1.3.3 ssl-key

- PEM 格式的 SSL 证书密钥文件路径，即 `--ssl-cert` 所指定的证书的私钥。
- 默认值：“”
- 目前 TiDB 不支持加载由密码保护的私钥。

12.5.1.3.4 cluster-ssl-ca

- CA 根证书，用于用 tls 连接 TiKV/PD
- 默认值：“”

12.5.1.3.5 cluster-ssl-cert

- ssl 证书文件路径，用于用 tls 连接 TiKV/PD
- 默认值：“”

12.5.1.3.6 cluster-ssl-key

- ssl 私钥文件路径，用于用 tls 连接 TiKV/PD
- 默认值：“”

12.5.1.4 performance

性能相关配置。

12.5.1.4.1 max-procs

- TiDB 的 CPU 使用数量。
- 默认值：0
- 默认值为 0 表示使用机器上所有的 CPU；如果设置成 n，那么 TiDB 会使用 n 个 CPU 数量。

12.5.1.4.2 max-memory

- Prepare cache LRU 使用的最大内存限制。当 Prepare cache LRU 的内存使用超过 `performance.max-memory * (1 - prepared-plan-cache.memory-guard-ratio)` 时，会剔除 LRU 中的元素。
- 默认值：0
- 这个配置只有在 `prepared-plan-cache.enabled` 为 `true` 的情况才会生效。当 LRU 的 size 大于 `prepared-plan-cache.capacity` 时，也会剔除 LRU 中的元素。

12.5.1.4.3 server-memory-quota 从 v4.0.9 版本开始引入

> **警告：**
>
> server-memory-quota 目前为实验性特性，不建议在生产环境中使用。

- tidb-server 实例内存的使用限制，单位为字节。
- 默认值：0
- 默认值为 0 表示无内存限制。

12.5.1.4.4 memory-usage-alarm-ratio 从 v4.0.9 版本开始引入

- tidb-server 实例内存使用占总内存的比例超过一定阈值时会报警。该配置项的有效范围为 0 到 1。如果配置该选项为 0 或 1，则表示关闭内存阈值报警功能。
- 默认值：0.8
- 当内存阈值报警功能开启时，如果配置项server-memory-quota 未设置，则内存报警阈值为 memory-usage-alarm-ratio * 系统内存大小；如果 server-memory-quota 被设置且大于 0，则内存报警阈值为 memory-usage-alarm-ratio * server-memory-quota。
- 当 TiDB 检测到 tidb-server 的内存使用超过了阈值，则会认为存在内存溢出的风险，会将当前正在执行的所有 SQL 语句中内存使用最高的 10 条语句和运行时间最长的 10 条语句以及 heap profile 记录到目录tmp-storage-path/record 中，并输出一条包含关键字 tidb-server has the risk of OOM 的日志。
- 该值作为系统变量tidb_memory_usage_alarm_ratio 的初始值。

12.5.1.4.5 txn-entry-size-limit 从 v4.0.10 版本开始引入

- TiDB 单行数据的大小限制
- 默认值：6291456 (Byte)
- 事务中单个 key-value 记录的大小限制。若超出该限制，TiDB 将会返回 entry too large 错误。该配置项的最大值不超过 125829120（表示 120MB）。
- 注意，TiKV 有类似的限制。若单个写入请求的数据量大小超出raft-entry-max-size，默认为 8MB，TiKV 会拒绝处理该请求。当表的一行记录较大时，需要同时修改这两个配置。

12.5.1.4.6 txn-total-size-limit

- TiDB 单个事务大小限制
- 默认值：104857600 (Byte)
- 单个事务中，所有 key-value 记录的总大小不能超过该限制。该配置项的最大值不超过 10737418240（表示 10GB）。注意，如果使用了以 Kafka 为下游消费者的 binlog，如：arbiter 集群，该配置项的值不能超过 1073741824（表示 1GB），因为这是 Kafka 的处理单条消息的最大限制，超过该限制 Kafka 将会报错。

12.5.1.4.7 max-txn-ttl

- 单个事务持锁的最长时间，超过该时间，该事务的锁可能会被其他事务清除，导致该事务无法成功提交。
- 默认值：600000
- 单位：毫秒
- 超过此时间的事务只能执行提交或者回滚，提交不一定能够成功。

12.5.1.4.8 committer-concurrency

- 在单个事务的提交阶段，用于执行提交操作相关请求的 goroutine 数量
- 默认值：16
- 若提交的事务过大，事务提交时的流控队列等待耗时可能会过长，可以通过调大该配置项来加速提交。

12.5.1.4.9 stmt-count-limit

- TiDB 单个事务允许的最大语句条数限制。
- 默认值：5000
- 在一个事务中，超过 stmt-count-limit 条语句后还没有 rollback 或者 commit，TiDB 将会返回 statement count 5001 exceeds the transaction limitation, autocommit = false 错误。该限制只在可重试的乐观事务中生效，如果使用悲观事务或者关闭了事务重试，事务中的语句数将不受此限制。

12.5.1.4.10 tcp-keep-alive

- TiDB 在 TCP 层开启 keepalive。
- 默认值：true

12.5.1.4.11 cross-join

- 默认值：true
- 默认可以执行在做 join 时两边表没有任何条件（where 字段）的语句；如果设置为 false，则有这样的 join 语句出现时，server 会拒绝执行

12.5.1.4.12 stats-lease

- TiDB 重载统计信息，更新表行数，检查是否需要自动 analyze，利用 feedback 更新统计信息以及加载列的统计信息的时间间隔。
- 默认值：3s
  - 每隔 stats-lease 时间，TiDB 会检查统计信息是否有更新，如果有会将其更新到内存中
  - 每隔 20 * stats-lease 时间，TiDB 会将 DML 产生的总行数以及修改的行数变化更新到系统表中
  - 每隔 stats-lease 时间，TiDB 会检查是否有表或者索引需要自动 analyze
  - 每隔 stats-lease 时间，TiDB 会检查是否有列的统计信息需要被加载到内存中
  - 每隔 200 * stats-lease 时间，TiDB 会将内存中缓存的 feedback 写入系统表中

  - 每隔 5 * stats-lease 时间，TiDB 会读取系统表中的 feedback，更新内存中缓存的统计信息
- 当 stats-lease 为 0s 时，TiDB 会以 3s 的时间间隔周期性的读取系统表中的统计信息并更新内存中缓存的统计信息。但不会自动修改统计信息相关系统表，具体来说，TiDB 不再自动修改这些表：
  - mysql.stats_meta：TiDB 不再自动记录事务中对某张表的修改行数，也不会更新到这个系统表中
  - mysql.stats_histograms/mysql.stats_buckets 和 mysql.stats_top_n：TiDB 不再自动 analyze 和主动更新统计信息
  - mysql.stats_feedback：TiDB 不再根据被查询的数据反馈的部分统计信息更新表和索引的统计信息

12.5.1.4.13 run-auto-analyze

- TiDB 是否做自动的 Analyze。
- 默认值：true

12.5.1.4.14 feedback-probability

- TiDB 对查询收集统计信息反馈的概率。
- 默认值：0
- 此功能默认关闭，暂不建议开启。如果开启此功能，对于每一个查询，TiDB 会以 feedback-probability 的概率收集查询的反馈，用于更新统计信息。

12.5.1.4.15 query-feedback-limit

- 在内存中缓存的最大 Query Feedback 数量，超过这个数量的 Feedback 会被丢弃。
- 默认值：512

12.5.1.4.16 pseudo-estimate-ratio

- 修改过的行数/表的总行数的比值，超过该值时系统会认为统计信息已经过期，会采用 pseudo 的统计信息。
- 默认值：0.8
- 最小值为 0；最大值为 1。

12.5.1.4.17 force-priority

- 把所有的语句优先级设置为 force-priority 的值。
- 默认值：NO_PRIORITY
- 可选值：NO_PRIORITY, LOW_PRIORITY, HIGH_PRIORITY, DELAYED。

12.5.1.4.18 distinct-agg-push-down

- 设置优化器是否执行将带有 Distinct 的聚合函数（比如 select count(distinct a)from t）下推到 Coprocessor 的优化操作。
- 默认值：false
- 该变量作为系统变量tidb_opt_distinct_agg_push_down 的初始值。

12.5.1.5 prepared-plan-cache

prepare 语句的 Plan cache 设置。

> **警告：**
> 当前该功能为实验特性，不建议在生产环境中使用。

12.5.1.5.1 enabled

- 开启 prepare 语句的 plan cache。
- 默认值：false

12.5.1.5.2 capacity

- 缓存语句的数量。
- 默认值：100
- 类型为 uint，小于 0 的值会被转化为大整数。

12.5.1.5.3 memory-guard-ratio

- 用于防止超过 performance.max-memory, 超过 max-memory * (1 - prepared-plan-cache.memory-guard-ratio) 会剔除 LRU 中的元素。
- 默认值：0.1
- 最小值为 0；最大值为 1。

12.5.1.6 tikv-client

12.5.1.6.1 grpc-connection-count

- 跟每个 TiKV 之间建立的最大连接数。
- 默认值：4

12.5.1.6.2 grpc-keepalive-time

- TiDB 与 TiKV 节点之间 rpc 连接 keepalive 时间间隔，如果超过该值没有网络包，grpc client 会 ping 一下 TiKV 查看是否存活。
- 默认值：10
- 单位：秒

12.5.1.6.3 grpc-keepalive-timeout

- TiDB 与 TiKV 节点 rpc keepalive 检查的超时时间
- 默认值：3
- 单位：秒

12.5.1.6.4 commit-timeout

- 执行事务提交时，最大的超时时间。
- 默认值：41s
- 这个值必须是大于两倍 Raft 选举的超时时间。

12.5.1.6.5 max-batch-size

- 批量发送 rpc 封包的最大数量，如果不为 0，将使用 BatchCommands api 发送请求到 TiKV，可以在并发度高的情况降低 rpc 的延迟，推荐不修改该值。
- 默认值：128

12.5.1.6.6 max-batch-wait-time

- 等待 max-batch-wait-time 纳秒批量将此期间的数据包封装成一个大包发送给 TiKV 节点，仅在 tikv-client.max-batch-size 值大于 0 时有效，不推荐修改该值。
- 默认值：0
- 单位：纳秒

12.5.1.6.7 batch-wait-size

- 批量向 TiKV 发送的封包最大数量，不推荐修改该值。
- 默认值：8
- 若此值为 0 表示关闭此功能。

12.5.1.6.8 overload-threshold

- TiKV 的负载阈值，如果超过此阈值，会收集更多的 batch 封包，来减轻 TiKV 的压力。仅在 tikv-client.max-batch-size 值大于 0 时有效，不推荐修改该值。
- 默认值：200

12.5.1.7 tikv-client.copr-cache 从 v4.0.0 版本开始引入

本部分介绍 Coprocessor Cache 相关的配置项。

12.5.1.7.1 enable

- 是否开启下推计算结果缓存。
- 默认值：false（即不开启）

12.5.1.7.2 capacity-mb

- 缓存的总数据量大小。当缓存空间满时，旧缓存条目将被逐出。
- 默认值：1000.0
- 单位：MB
- 类型：Float

12.5.1.7.3 admission-max-result-mb

- 指定能被缓存的最大单个下推计算结果集。若单个下推计算在 Coprocessor 上返回的结果集小于该参数指定的大小，则结果集会被缓存。调大该值可以缓存更多种类下推请求，但也将导致缓存空间更容易被占满。注意，每个下推计算结果集大小一般都会小于 Region 大小，因此将该值设置得远超过 Region 大小没有意义。
- 默认值：10.0
- 单位：MB
- 类型：Float

12.5.1.7.4 admission-min-process-ms

- 指定能被缓存的单个下推计算结果集的最短计算时间。若单个下推计算在 Coprocessor 上的计算时间小于该参数指定的时间，则结果集不会被缓存。处理得很快的请求没有必要进行缓存，仅对处理时间很长的请求进行缓存，减少缓存被逐出的概率，这是本配置参数的意义。
- 默认值：5
- 单位：ms

12.5.1.7.5 admission-max-ranges 从 v4.0.8 版本开始引入

- 指定能被缓存的单个下推计算结果集的最大范围数量。如果下推计算存在的范围数量超过该配置项指定的数量，则结果集不会被缓存。一般认为当范围数量过多时，解析范围是计算的主要开销，这样 Coprocessor Cache 带来的额外计算开销会较大。
- 默认值：500
- 类型：uint

12.5.1.8 binlog

TiDB Binlog 相关配置。

12.5.1.8.1 enable

- binlog 开关。
- 默认值：false

12.5.1.8.2 write-timeout

- 写 binlog 的超时时间，推荐不修改该值。
- 默认值：15s
- 单位：秒

12.5.1.8.3 ignore-error

- 忽略写 binlog 发生的错误时处理开关，推荐不修改该值。
- 默认值：false
- 如果设置为 `true`，发生错误时，TiDB 会停止写入 binlog，并且在监控项 `tidb_server_critical_error_total` 上计数加 1；如果设置为 `false`，写入 binlog 失败，会停止整个 TiDB 的服务。

12.5.1.8.4 binlog-socket

- binlog 输出网络地址。
- 默认值：“”

12.5.1.8.5 strategy

- binlog 输出时选择 pump 的策略，仅支持 hash，range 方法。
- 默认值：“range”

12.5.1.9 status

TiDB 服务状态相关配置。

12.5.1.9.1 report-status

- 开启 HTTP API 服务的开关。
- 默认值：true

12.5.1.9.2 record-db-qps

- 输出与 database 相关的 QPS metrics 到 Prometheus 的开关。
- 默认值：false

12.5.1.10 stmt-summary 从 v3.0.4 版本开始引入

系统表statement summary tables 的相关配置。

12.5.1.10.1 max-stmt-count

- 系统表statement summary tables 中保存的 SQL 种类的最大数量。
- 默认值：v4.0.14 前为 200，自 v4.0.14 起为 3000

12.5.1.10.2 max-sql-length

- 系统表statement summary tables 中 DIGEST_TEXT 和 QUERY_SAMPLE_TEXT 列的最大显示长度。
- 默认值：4096

12.5.1.11 pessimistic-txn

12.5.1.11.1 enable

- 开启悲观事务支持，悲观事务使用方法请参考TiDB 悲观事务模式。
- 默认值：true

12.5.1.11.2 max-retry-count

- 悲观事务中单个语句最大重试次数，重试次数超过该限制，语句执行将会报错。
- 默认值：256

12.5.1.12 experimental

experimental 部分为 TiDB 实验功能相关的配置。该部分从 v3.1.0 开始引入。

12.5.1.12.1 allow-expression-index 从 v4.0.0 版本开始引入

- 用于控制是否能创建表达式索引。
- 默认值：false

12.5.2 TiKV 配置文件描述

TiKV 配置文件比命令行参数支持更多的选项。你可以在 etc/config-template.toml 找到默认值的配置文件，重命名为 config.toml 即可。

本文档只阐述未包含在命令行参数中的参数，命令行参数参见TiKV 配置参数。

12.5.2.1 全局配置

12.5.2.1.1 log-level

- 日志等级。
- 可选值：“trace”，“debug”，“info”，“warning”，“error”，“critical”
- 默认值：“info”

12.5.2.1.2 log-file

- 日志文件。如果未设置该项，日志会默认输出到 “stderr”。
- 默认值：“”

12.5.2.1.3 log-format

- 日志的格式。
- 可选值：“json”，“text”
- 默认值：“text”

12.5.2.1.4 log-rotation-timespan

- 轮换日志的时间跨度。当超过该时间跨度，日志文件会被轮换，即在当前日志文件的文件名后附加一个时间戳，并创建一个新文件。
- 默认值：“24h”

12.5.2.1.5 log-rotation-size

- 触发日志轮换的文件大小。一旦日志文件大小超过指定的阈值，日志文件将被轮换，将旧文件被置于新文件中，新文件名即旧文件名加上时间戳后缀。
- 默认值：“300MB”

12.5.2.1.6 slow-log-file

- 存储慢日志的文件。
- 如果未设置本项但设置了 `log-file`，慢日志将输出至 `log-file` 指定的日志文件中。如果本项和 `log-file` 均未设置，所有日志默认输出到 “stderr”。
- 默认值：“”

12.5.2.1.7 slow-log-threshold

- 输出慢日志的阈值。处理时间超过该阈值后会输出慢日志。
- 默认值：“1s”

12.5.2.2 server

服务器相关的配置项。

12.5.2.2.1 status-thread-pool-size

- HTTP API 服务的工作线程数量。
- 默认值：1
- 最小值：1

12.5.2.2.2 grpc-compression-type

- gRPC 消息的压缩算法，取值：none，deflate，gzip。
- 默认值：none

12.5.2.2.3 grpc-concurrency

- gRPC 工作线程的数量。
- 默认值：4
- 最小值：1

12.5.2.2.4 grpc-concurrent-stream

- 一个 gRPC 链接中最多允许的并发请求数量。
- 默认值：1024
- 最小值：1

12.5.2.2.5 grpc-memory-pool-quota

- gRPC 可使用的内存大小限制。
- 默认值: 无限制
- 建议仅在出现内存不足 (OOM) 的情况下限制内存使用。需要注意，限制内存使用可能会导致卡顿。

12.5.2.2.6 grpc-raft-conn-num

- tikv 节点之间用于 raft 通讯的链接最大数量。
- 默认值：1
- 最小值：1

12.5.2.2.7 grpc-stream-initial-window-size

- gRPC stream 的 window 大小。
- 默认值：2MB
- 单位：KB|MB|GB
- 最小值：1KB

12.5.2.2.8 grpc-keepalive-time

- gRPC 发送 keep alive ping 消息的间隔时长。
- 默认值：10s
- 最小值：1s

12.5.2.2.9 grpc-keepalive-timeout

- 关闭 gRPC 链接的超时时长。
- 默认值：3s
- 最小值：1s

12.5.2.2.10 concurrent-send-snap-limit

- 同时发送 snapshot 的最大个数。
- 默认值：32
- 最小值：1

12.5.2.2.11 concurrent-recv-snap-limit

- 同时接受 snapshot 的最大个数。
- 默认值：32
- 最小值：1

12.5.2.2.12 end-point-recursion-limit

- endpoint 下推查询请求解码消息时，最多允许的递归层数。
- 默认值：1000
- 最小值：1

12.5.2.2.13 end-point-request-max-handle-duration

- endpoint 下推查询请求处理任务最长允许的时长。
- 默认值：60s
- 最小值：1s

12.5.2.2.14 snap-max-write-bytes-per-sec

- 处理 snapshot 时最大允许使用的磁盘带宽。
- 默认值：100MB
- 单位：KB|MB|GB
- 最小值：1KB

12.5.2.2.15 end-point-slow-log-threshold

- endpoint 下推查询请求输出慢日志的阈值，处理时间超过阈值后会输出慢日志。
- 默认值：1s
- 最小值：0

12.5.2.3 readpool.unified

> **警告：**
>
> 该功能目前为实验特性，不建议在生产环境中使用。

统一处理读请求的线程池相关的配置项。该线程池自 4.0 版本起取代原有的 storage 和 coprocessor 线程池。

12.5.2.3.1 `min-thread-count`

- 统一处理读请求的线程池最少的线程数量。
- 默认值：1

12.5.2.3.2 `max-thread-count`

- 统一处理读请求的线程池最多的线程数量。
- 默认值：CPU * 0.8，但最少为 4

12.5.2.3.3 `stack-size`

- 统一处理读请求的线程池中线程的栈大小。
- 默认值：10MB
- 单位：KB|MB|GB
- 最小值：2MB

12.5.2.3.4 `max-tasks-per-worker`

- 统一处理读请求的线程池中单个线程允许积压的最大任务数量，超出后会返回 Server Is Busy。
- 默认值：2000
- 最小值：2

12.5.2.4 readpool.storage

存储线程池相关的配置项。

12.5.2.4.1 `use-unified-pool`

- 是否使用统一的读取线程池（在`readpool.unified` 中配置）处理存储请求。该选项值为 false 时，使用单独的存储线程池。通过本节 (`readpool.storage`) 中的其余配置项配置单独的线程池。
- 默认值：false

12.5.2.4.2 high-concurrency

- 处理高优先级读请求的线程池线程数量。
- 当 8 ≤ cpu num ≤ 16 时，默认值为 cpu_num * 0.5；当 cpu num 大于 8 时，默认值为 4；当 cpu num 大于 16 时，默认值为 8。
- 最小值：1

12.5.2.4.3 normal-concurrency

- 处理普通优先级读请求的线程池线程数量。
- 当 8 ≤ cpu num ≤ 16 时，默认值为 cpu_num * 0.5；当 cpu num 大于 8 时，默认值为 4；当 cpu num 大于 16 时，默认值为 8。
- 最小值：1

12.5.2.4.4 low-concurrency

- 处理低优先级读请求的线程池线程数量。
- 当 8 ≤ cpu num ≤ 16 时，默认值为 cpu_num * 0.5；当 cpu num 大于 8 时，默认值为 4；当 cpu num 大于 16 时，默认值为 8。
- 最小值：1

12.5.2.4.5 max-tasks-per-worker-high

- 高优先级线程池中单个线程允许积压的最大任务数量，超出后会返回 Server Is Busy。
- 默认值：2000
- 最小值：2

12.5.2.4.6 max-tasks-per-worker-normal

- 普通优先级线程池中单个线程允许积压的最大任务数量，超出后会返回 Server Is Busy。
- 默认值：2000
- 最小值：2

12.5.2.4.7 max-tasks-per-worker-low

- 低优先级线程池中单个线程允许积压的最大任务数量，超出后会返回 Server Is Busy。
- 默认值：2000
- 最小值：2

12.5.2.4.8 stack-size

- Storage 读线程池中线程的栈大小。
- 默认值：10MB
- 单位：KB|MB|GB
- 最小值：2MB

12.5.2.5 readpool.coprocessor

协处理器线程池相关的配置项。

12.5.2.5.1 use-unified-pool

- 是否使用统一的读取线程池（在`readpool.unified` 中配置）处理协处理器请求。该选项值为 false 时，使用单独的协处理器线程池。通过本节 (`readpool.coprocessor`) 中的其余配置项配置单独的线程池。
- 默认值：如果本节 (`readpool.coprocessor`) 中没有其他配置，默认为 true。否则，为了升级兼容性，默认为 false，请根据需要更改`readpool.unified` 中的配置后再启用该选项。

12.5.2.5.2 high-concurrency

- 处理高优先级 Coprocessor 请求（如点查）的线程池线程数量。
- 默认值：CPU * 0.8
- 最小值：1

12.5.2.5.3 normal-concurrency

- 处理普通优先级 Coprocessor 请求的线程池线程数量。
- 默认值：CPU * 0.8
- 最小值：1

12.5.2.5.4 low-concurrency

- 处理低优先级 Coprocessor 请求（如扫表）的线程池线程数量。
- 默认值：CPU * 0.8
- 最小值：1

12.5.2.5.5 max-tasks-per-worker-high

- 高优先级线程池中单个线程允许积压的任务数量，超出后会返回 Server Is Busy。
- 默认值：2000
- 最小值：2

12.5.2.5.6 max-tasks-per-worker-normal

- 普通优先级线程池中单个线程允许积压的任务数量，超出后会返回 Server Is Busy。
- 默认值：2000
- 最小值：2

12.5.2.5.7 max-tasks-per-worker-low

- 低优先级线程池中单个线程允许积压的任务数量，超出后会返回 Server Is Busy。
- 默认值：2000
- 最小值：2

12.5.2.5.8 stack-size

Coprocessor 线程池中线程的栈大小，默认值：10，单位：KiB|MiB|GiB。

- 默认值：10MB
- 单位：KB|MB|GB
- 最小值：2MB

12.5.2.6 storage

存储相关的配置项。

12.5.2.6.1 scheduler-concurrency

- scheduler 内置一个内存锁机制，防止同时对一个 key 进行操作。每个 key hash 到不同的槽。
- 默认值：524288
- 最小值：1

12.5.2.6.2 scheduler-worker-pool-size

- scheduler 线程个数，主要负责写入之前的事务一致性检查工作。如果 CPU 核心数量大于等于 16，默认为 8；否则默认为 4。
- 默认值：4
- 最小值：1

12.5.2.6.3 scheduler-pending-write-threshold

- 写入数据队列的最大值，超过该值之后对于新的写入 TiKV 会返回 Server Is Busy 错误。
- 默认值：100MB
- 单位: MB|GB

12.5.2.6.4 reserve-space

- TiKV 启动时预占额外空间的临时文件大小。临时文件名为 space_placeholder_file，位于 storage.data-dir 目录下。TiKV 磁盘空间耗尽无法正常启动需要紧急干预时，可以删除该文件，并且将 reserve-space 设置为 0MB。
- 默认值：2GB
- 单位: MB|GB

12.5.2.7 storage.block-cache

RocksDB 多个 CF 之间共享 block cache 的配置选项。当开启时，为每个 CF 单独配置的 block cache 将无效。

12.5.2.7.1 `shared`

- 是否开启共享 block cache。
- 默认值：true

12.5.2.7.2 `capacity`

- 共享 block cache 的大小。
- 默认值：系统总内存大小的 45%
- 单位：KB|MB|GB

12.5.2.8 raftstore

raftstore 相关的配置项。

12.5.2.8.1 `sync-log`

- 数据、log 落盘是否 sync。
- 默认值：true

> **警告：**
>
> 将该值设置为 false 可能会导致数据丢失。因此强烈建议不要修改此配置。

12.5.2.8.2 `prevote`

- 开启 Prevote 的开关，开启有助于减少隔离恢复后对系统造成的抖动。
- 默认值：true

12.5.2.8.3 `raftdb-path`

- raft 库的路径，默认存储在 storage.data-dir/raft 下。
- 默认值：“”

12.5.2.8.4 `raft-base-tick-interval`

- 状态机 tick 一次的间隔时间。
- 默认值：1s
- 最小值：大于 0

12.5.2.8.5 raft-heartbeat-ticks

- 发送心跳时经过的 tick 个数，即每隔 raft-base-tick-interval * raft-heartbeat-ticks 时间发送一次心跳。
- 默认值：2
- 最小值：大于 0

12.5.2.8.6 raft-election-timeout-ticks

- 发起选举时经过的 tick 个数，即如果处于无主状态，大约经过 raft-base-tick-interval * raft-election-timeout-ticks 时间以后发起选举。
- 默认值：10
- 最小值：raft-heartbeat-ticks

12.5.2.8.7 raft-min-election-timeout-ticks

- 发起选举时至少经过的 tick 个数，如果为 0，则表示使用 raft-election-timeout-ticks，不能比 raft-election-timeout-ticks 小。
- 默认值：0
- 最小值：0

12.5.2.8.8 raft-max-election-timeout-ticks

- 发起选举时最多经过的 tick 个数，如果为 0，则表示使用 raft-election-timeout-ticks * 2。
- 默认值：0
- 最小值：0

12.5.2.8.9 raft-max-size-per-msg

- 产生的单个消息包的大小限制，软限制。
- 默认值：1MB
- 最小值：0
- 单位：MB

12.5.2.8.10 raft-max-inflight-msgs

- 待确认日志个数的数量，如果超过这个数量将会减缓发送日志的个数。
- 默认值：256
- 最小值：大于 0

12.5.2.8.11 raft-entry-max-size

- 单个日志最大大小，硬限制。
- 默认值：8MB
- 最小值：0
- 单位：MB|GB

12.5.2.8.12 raft-log-gc-tick-interval

- 删除 raft 日志的轮询任务调度间隔时间，0 表示不启用。
- 默认值：10s
- 最小值：0

12.5.2.8.13 raft-log-gc-threshold

- 允许残余的 raft 日志个数，这是一个软限制。
- 默认值：50
- 最小值：1

12.5.2.8.14 raft-log-gc-count-limit

- 允许残余的 raft 日志个数，这是一个硬限制。默认值为按照每个日志 1MB 而计算出来的 3/4 region 大小所能容纳的日志个数。
- 最小值：0

12.5.2.8.15 raft-log-gc-size-limit

- 允许残余的 raft 日志大小，这是一个硬限制，默认为 region 大小的 3/4。
- 最小值：大于 0

12.5.2.8.16 raft-entry-cache-life-time

- 内存中日志 cache 允许的最长残留时间。
- 默认值：30s
- 最小值：0

12.5.2.8.17 raft-reject-transfer-leader-duration

- 新节点保护时间，控制迁移 leader 到新加节点的最小时间，设置过小容易导致迁移 leader 失败。
- 默认值：3s
- 最小值：0

12.5.2.8.18 hibernate-regions (实验特性)

- 打开或关闭静默 Region。打开后，如果 Region 长时间处于非活跃状态，即被自动设置为静默状态。静默状态的 Region 可以降低 Leader 和 Follower 之间心跳信息的系统开销。可以通过 peer-stale-state-check-interval 调整 Leader 和 Follower 之间的心跳间隔。
- 默认值：false

12.5.2.8.19 split-region-check-tick-interval

- 检查 region 是否需要分裂的时间间隔，0 表示不启用。
- 默认值：10s
- 最小值：0

12.5.2.8.20 region-split-check-diff

- 允许 region 数据超过指定大小的最大值，默认为 region 大小的 1/16。
- 最小值：0

12.5.2.8.21 region-compact-check-interval

- 检查是否需要人工触发 rocksdb compaction 的时间间隔，0 表示不启用。
- 默认值：5m
- 最小值：0

12.5.2.8.22 region-compact-check-step

- 每轮校验人工 compaction 时，一次性检查的 region 个数。
- 默认值：100
- 最小值：0

12.5.2.8.23 region-compact-min-tombstones

- 触发 rocksdb compaction 需要的 tombstone 个数。
- 默认值：10000
- 最小值：0

12.5.2.8.24 region-compact-tombstones-percent

- 触发 rocksdb compaction 需要的 tombstone 所占比例。
- 默认值：30
- 最小值：1
- 最大值：100

12.5.2.8.25 pd-heartbeat-tick-interval

- 触发 region 对 PD 心跳的时间间隔，0 表示不启用。
- 默认值：1m
- 最小值：0

12.5.2.8.26 pd-store-heartbeat-tick-interval

- 触发 store 对 PD 心跳的时间间隔，0 表示不启用。
- 默认值：10s
- 最小值：0

12.5.2.8.27 snap-mgr-gc-tick-interval

- 触发回收过期 snapshot 文件的时间间隔，0 表示不启用。
- 默认值：1m
- 最小值：0

12.5.2.8.28 snap-gc-timeout

- snapshot 文件的最长保存时间。
- 默认值：4h
- 最小值：0

12.5.2.8.29 lock-cf-compact-interval

- 触发对 lock CF compact 检查的时间间隔。
- 默认值：10m
- 最小值：0

12.5.2.8.30 lock-cf-compact-bytes-threshold

- 触发对 lock CF 进行 compact 的大小。
- 默认值：256MB
- 最小值：0
- 单位：MB

12.5.2.8.31 notify-capacity

- region 消息队列的最长长度。
- 默认值：40960
- 最小值：0

12.5.2.8.32 messages-per-tick

- 每轮处理的消息最大个数。
- 默认值：4096
- 最小值：0

12.5.2.8.33 max-peer-down-duration

- 副本允许的最长未响应时间，超过将被标记为 down，后续 PD 会尝试将其删掉。
- 默认值：5m
- 最小值：0

12.5.2.8.34 max-leader-missing-duration

- 允许副本处于无主状态的最长时间，超过将会向 PD 校验自己是否已经被删除。
- 默认值：2h
- 最小值：> abnormal-leader-missing-duration

12.5.2.8.35 abnormal-leader-missing-duration

- 允许副本处于无主状态的时间，超过将视为异常，标记在 metrics 和日志中。
- 默认值：10m
- 最小值：> peer-stale-state-check-interval

12.5.2.8.36 peer-stale-state-check-interval

- 触发检验副本是否处于无主状态的时间间隔。
- 默认值：5m
- 最小值：> 2 * election-timeout

12.5.2.8.37 leader-transfer-max-log-lag

- 尝试转移领导权时被转移者允许的最大日志缺失个数。
- 默认值：128
- 最小值：10

12.5.2.8.38 snap-apply-batch-size

- 当导入 snapshot 文件需要写数据时，内存写缓存的大小
- 默认值：10MB
- 最小值：0
- 单位：MB

12.5.2.8.39 consistency-check-interval

- 触发一致性检查的时间间隔, 0 表示不启用。
- 默认值：0s
- 最小值：0

12.5.2.8.40 raft-store-max-leader-lease

- region 主可信任期的最长时间。
- 默认值：9s
- 最小值：0

12.5.2.8.41 right-derive-when-split

- 为 true 时，以最大分裂 key 为起点的 region 复用原 region 的 key；否则以原 region 起点 key 作为起点的 region 复用原 region 的 key。
- 默认值：true

12.5.2.8.42 allow-remove-leader

- 允许删除主开关。
- 默认值：false

12.5.2.8.43 merge-max-log-gap

- 进行 merge 时，允许的最大日志缺失个数。
- 默认值：10
- 最小值：> raft-log-gc-count-limit

12.5.2.8.44 merge-check-tick-interval

- 触发 merge 完成检查的时间间隔。
- 默认值：10s
- 最小值：大于 0

12.5.2.8.45 use-delete-range

- 开启 rocksdb delete_range 接口删除数据的开关。
- 默认值：false

12.5.2.8.46 cleanup-import-sst-interval

- 触发检查过期 SST 文件的时间间隔，0 表示不启用。
- 默认值：10m
- 最小值：0

12.5.2.8.47 local-read-batch-size

- 一轮处理读请求的最大个数。
- 默认值：1024
- 最小值：大于 0

12.5.2.8.48 apply-max-batch-size

- 一轮处理数据落盘的最大请求个数。
- 默认值：1024
- 最小值：大于 0

12.5.2.8.49 apply-pool-size

- 处理数据落盘的线程池线程数。
- 默认值：2
- 最小值：大于 0

12.5.2.8.50 store-max-batch-size

- 一轮处理的最大请求个数。
- 默认值：1024
- 最小值：大于 0

12.5.2.8.51 store-pool-size

- 处理 raft 的线程池线程数。
- 默认值：2
- 最小值：大于 0

12.5.2.8.52 future-poll-size

- 驱动 future 的线程池线程数。
- 默认值：1
- 最小值：大于 0

12.5.2.9 coprocessor

coprocessor 相关的配置项。

12.5.2.9.1 split-region-on-table

- 开启按 table 分裂 Region 的开关，建议仅在 TiDB 模式下使用。
- 默认值：false

12.5.2.9.2 batch-split-limit

- 批量分裂 Region 的阈值，调大该值可加速分裂 Region。
- 默认值：10
- 最小值：1

12.5.2.9.3 region-max-size

- Region 容量空间最大值，超过时系统分裂成多个 Region。
- 默认值：144MB
- 单位：KB|MB|GB

12.5.2.9.4 region-split-size

- 分裂后新 Region 的大小，此值属于估算值。
- 默认值：96MB
- 单位：KB|MB|GB

12.5.2.9.5 region-max-keys

- Region 最多允许的 key 的个数，超过时系统分裂成多个 Region。
- 默认值：1440000

12.5.2.9.6 region-split-keys

- 分裂后新 Region 的 key 的个数，此值属于估算值。
- 默认值：960000

12.5.2.10 rocksdb

rocksdb 相关的配置项。

12.5.2.10.1 max-background-jobs

- RocksDB 后台线程个数。
- 默认值：8
- 最小值：2

12.5.2.10.2 max-background-flushes

- RocksDB 用于刷写 memtable 的最大后台线程数。
- 默认值：2
- 最小值：1

12.5.2.10.3 max-sub-compactions

- RocksDB 进行 subcompaction 的并发个数。
- 默认值：3
- 最小值：1

12.5.2.10.4 max-open-files

- RocksDB 可以打开的文件总数。
- 默认值：40960
- 最小值：-1

12.5.2.10.5 max-manifest-file-size

- RocksDB Manifest 文件最大大小。
- 默认值：128MB
- 最小值：0
- 单位：B|KB|MB|GB

12.5.2.10.6 create-if-missing

- 自动创建 DB 开关。
- 默认值：true

12.5.2.10.7 wal-recovery-mode

- WAL 恢复模式，取值：0（TolerateCorruptedTailRecords），1（AbsoluteConsistency），2（PointInTimeRecovery），3（SkipAnyCorruptedRecords）。
- 默认值：2
- 最小值：0
- 最大值：3

12.5.2.10.8 wal-dir

- WAL 存储目录，默认：“tmp/tikv/store”。
- 默认值：/tmp/tikv/store

12.5.2.10.9 wal-ttl-seconds

- 归档 WAL 生存周期，超过该值时，系统会删除相关 WAL。
- 默认值：0
- 最小值：0
- 单位：秒

12.5.2.10.10 wal-size-limit

- 归档 WAL 大小限制，超过该值时，系统会删除相关 WAL。
- 默认值：0
- 最小值：0
- 单位：B|KB|MB|GB

12.5.2.10.11 enable-statistics

- 开启 RocksDB 的统计信息。
- 默认值：true

12.5.2.10.12 stats-dump-period

- 将统计信息输出到日志中的间隔时间。
- 默认值：10m

12.5.2.10.13 compaction-readahead-size

- 异步 Sync 限速速率。
- 默认值：0
- 最小值：0
- 单位：B|KB|MB|GB

12.5.2.10.14 writable-file-max-buffer-size

- WritableFileWrite 所使用的最大的 buffer 大小。
- 默认值：1MB
- 最小值：0
- 单位：B|KB|MB|GB

12.5.2.10.15 use-direct-io-for-flush-and-compaction

- flush 或者 compaction 开启 DirectIO 的开关。
- 默认值：false

12.5.2.10.16 rate-bytes-per-sec

- Rate Limiter 限制速率。
- 默认值：0
- 最小值：0
- 单位：Bytes

12.5.2.10.17 rate-limiter-mode

- Rate LImiter 模式，取值：1（ReadOnly），2（WriteOnly），3（AllIo）。
- 默认值：2
- 最小值：1
- 最大值：3

12.5.2.10.18 auto-tuned

- 开启自动优化 Rate LImiter 的配置的开关。
- 默认值：false

12.5.2.10.19 enable-pipelined-write

- 开启 Pipelined Write 的开关。
- 默认值：true

12.5.2.10.20 bytes-per-sync

- 异步 Sync 限速速率。
- 默认值：1MB
- 最小值：0
- 单位：B|KB|MB|GB

12.5.2.10.21 wal-bytes-per-sync

- WAL Sync 限速速率，默认：512KB。
- 默认值：512KB
- 最小值：0
- 单位：B|KB|MB|GB

12.5.2.10.22 info-log-max-size

- Info 日志的最大大小。
- 默认值：1GB
- 最小值：0
- 单位：B|KB|MB|GB

12.5.2.10.23 info-log-roll-time

- 日志截断间隔时间，如果为 0s 则不截断。
- 默认值：0s

12.5.2.10.24 info-log-keep-log-file-num

- 保留日志文件最大个数。
- 默认值：10
- 最小值：0

12.5.2.10.25 info-log-dir

- 日志存储目录。
- 默认值：“”

12.5.2.11 rocksdb.titan

Titan 相关的配置项。

12.5.2.11.1 enabled

- 开启 Titan 开关。
- 默认值：false

12.5.2.11.2 dirname

- Titan Blob 文件存储目录。
- 默认值：titandb

12.5.2.11.3 disable-gc

- 关闭 Titan 对 Blob 文件的 GC 的开关。
- 默认值：false

12.5.2.11.4 max-background-gc

- Titan 后台 GC 的线程个数。
- 默认值：4
- 最小值：1

12.5.2.12 rocksdb.defaultcf | rocksdb.writecf | rocksdb.lockcf

rocksdb defaultcf、 rocksdb writecf 和 rocksdb lockcf 相关的配置项。

12.5.2.12.1 block-size

- 一个 RocksDB block 的默认大小。
- defaultcf 默认值：64KB
- writecf 默认值：64KB
- lockcf 默认值：16KB
- 最小值：1KB
- 单位：KB|MB|GB

12.5.2.12.2 block-cache-size

- 一个 RocksDB block 的默认缓存大小。
- defaultcf 默认值：机器总内存 * 25%
- writecf 默认值：机器总内存 * 15%
- lockcf 默认值：机器总内存 * 2%
- 最小值：0
- 单位：KB|MB|GB

12.5.2.12.3 disable-block-cache

- 开启 block cache 开关。
- 默认值：false

12.5.2.12.4 cache-index-and-filter-blocks

- 开启缓存 index 和 filter 的开关。
- 默认值：true

12.5.2.12.5 pin-l0-filter-and-index-blocks

- 是否 pin 住 L0 的 index 和 filter。
- 默认值：true

12.5.2.12.6 use-bloom-filter

- 开启 bloom filter 的开关。
- 默认值：true

12.5.2.12.7 optimize-filters-for-hits

- 开启优化 filter 的命中率的开关。
- defaultcf 默认值：true
- writecf 默认值：false
- lockcf 默认值：false

12.5.2.12.8 whole_key_filtering

- 开启将整个 key 放到 bloom filter 中的开关。
- defaultcf 默认值：true
- writecf 默认值：false
- lockcf 默认值：false

12.5.2.12.9 bloom-filter-bits-per-key

bloom filter 为每个 key 预留的长度。

- 默认值：10
- 单位：字节

12.5.2.12.10 block-based-bloom-filter

- 开启每个 block 建立 bloom filter 的开关。
- 默认值：false

12.5.2.12.11 read-amp-bytes-per-bit

- 开启读放大统计的开关，0：不开启，> 0 开启。
- 默认值：0
- 最小值：0

12.5.2.12.12 compression-per-level

- 每一层默认压缩算法，默认：前两层为 No，后面 5 层为 lz4。
- 默认值：[“no”，“no”，“lz4”，“lz4”，“lz4”，“zstd”，“zstd”]

12.5.2.12.13 write-buffer-size

- memtable 大小。
- defaultcf 默认值："128MB"
- writecf 默认值："128MB"
- lockcf 默认值："32MB"
- 最小值：0
- 单位：KB|MB|GB

12.5.2.12.14 max-write-buffer-number

- 最大 memtable 个数。
- 默认值：5
- 最小值：0

12.5.2.12.15 min-write-buffer-number-to-merge

- 触发 flush 的最小 memtable 个数。
- 默认值：1
- 最小值：0

12.5.2.12.16 max-bytes-for-level-base

- base level (L1) 最大字节数，一般设置为 memtable 大小 4 倍。
- defaultcf 默认值："512MB"
- writecf 默认值："512MB"
- lockcf 默认值："128MB"
- 最小值：0
- 单位：KB|MB|GB

12.5.2.12.17 target-file-size-base

- base level 的目标文件大小。
- 默认值：8MB
- 最小值：0
- 单位：KB|MB|GB

12.5.2.12.18 level0-file-num-compaction-trigger

- 触发 compaction 的 L0 文件最大个数。
- defaultcf 默认值：4
- writecf 默认值：4
- lockcf 默认值：1
- 最小值：0

12.5.2.12.19 level0-slowdown-writes-trigger

- 触发 write stall 的 L0 文件最大个数。
- 默认值：20
- 最小值：0

12.5.2.12.20 level0-stop-writes-trigger

- 完全阻停写入的 L0 文件最大个数。
- 默认值：36
- 最小值：0

12.5.2.12.21 max-compaction-bytes

- 一次 compaction 最大写入字节数，默认 2GB。
- 默认值：2GB
- 最小值：0
- 单位：KB|MB|GB

12.5.2.12.22 compaction-pri

- Compaction 优先类型
- 可选择值：0 (ByCompensatedSize)，1 (OldestLargestSeqFirst)，2 (OldestSmallestSeqFirst)，3 (MinOverlappingRatio)。
- defaultcf 默认值：3
- writecf 默认值：3
- lockcf 默认值：1

12.5.2.12.23 dynamic-level-bytes

- 开启 dynamic level bytes 优化的开关。
- 默认值：true

12.5.2.12.24 num-levels

- RocksDB 文件最大层数。
- 默认值：7

12.5.2.12.25 max-bytes-for-level-multiplier

- 每一层的默认放大倍数。
- 默认值：10

12.5.2.12.26 compaction-style

- Compaction 方法。
- 可选值（仅可输入数字）：0 (Level)，1 (Universal)，2 (Fifo)
- 默认值：0

12.5.2.12.27 disable-auto-compactions

- 是否关闭自动 compaction
- 默认值：false

12.5.2.12.28 soft-pending-compaction-bytes-limit

- pending compaction bytes 的软限制。
- 默认值：64GB
- 单位：KB|MB|GB

12.5.2.12.29 hard-pending-compaction-bytes-limit

- pending compaction bytes 的硬限制。
- 默认值：256GB
- 单位：KB|MB|GB

12.5.2.13 rocksdb.defaultcf.titan

rocksdb defaultcf titan 相关的配置项。

12.5.2.13.1 min-blob-size

- 最小存储在 Blob 文件中 value 大小，低于该值的 value 还是存在 LSM-Tree 中。
- 默认值：1KB
- 最小值：0
- 单位：KB|MB|GB

12.5.2.13.2 blob-file-compression

- Blob 文件所使用的压缩算法，可选值：no、 snappy、 zlib、 bz2、 lz4、 lz4hc、 zstd。
- 默认值：lz4

12.5.2.13.3 blob-cache-size

- Blob 文件的 cache 大小，默认：0GB。
- 默认值：0GB
- 最小值：0
- 单位：KB|MB|GB

12.5.2.13.4 min-gc-batch-size

- 做一次 GC 所要求的最低 Blob 文件大小总和。
- 默认值：16MB
- 最小值：0
- 单位：KB|MB|GB

12.5.2.13.5 max-gc-batch-size

- 做一次 GC 所要求的最高 Blob 文件大小总和。
- 默认值：64MB
- 最小值：0
- 单位：KB|MB|GB

12.5.2.13.6 discardable-ratio

- Blob 文件 GC 的触发比例，如果某 Blob 文件中的失效 value 的比例高于该值才可能被 GC 选中。
- 默认值：0.5
- 最小值：0
- 最大值：1

12.5.2.13.7 sample-ratio

- 进行 GC 时，对 Blob 文件进行采样时读取数据占整个文件的比例。
- 默认值：0.1
- 最小值：0
- 最大值：1

12.5.2.13.8 merge-small-file-threshold

- Blob 文件的大小小于该值时，无视 discardable-ratio 仍可能被 GC 选中。
- 默认值：8MB
- 最小值：0
- 单位：KB|MB|GB

12.5.2.13.9 blob-run-mode

- Titan 的运行模式选择，可选值：
  - “normal”：value size 超过 min-blob-size 的数据会写入到 blob 文件。
  - “read_only”：不再写入新数据到 blob，原有 blob 内的数据仍然可以读取。
  - “fallback”：将 blob 内的数据写回 LSM。
- 默认值：“normal”

12.5.2.13.10 level-merge

- 是否通过开启 level-merge 来提升读性能，副作用是写放大会比不开启更大。
- 默认值：false

12.5.2.13.11 gc-merge-rewrite

- 是否开启使用 merge operator 来进行 Titan GC 写回操作，减少 Titan GC 对于前台写入的影响。
- 默认值：false

12.5.2.14 raftdb

raftdb 相关配置项。

12.5.2.14.1 max-background-jobs

- RocksDB 后台线程个数。
- 默认值：4
- 最小值：2

12.5.2.14.2 max-sub-compactions

- RocksDB 进行 subcompaction 的并发数。
- 默认值：2
- 最小值：1

12.5.2.14.3 wal-dir

- WAL 存储目录。
- 默认值：/tmp/tikv/store

12.5.2.15 security

安全相关配置项。

12.5.2.15.1 ca-path

- CA 文件路径
- 默认值：“”

12.5.2.15.2 cert-path

- 包含 X509 证书的 PEM 文件路径
- 默认值：“”

12.5.2.15.3 key-path

- 包含 X509 key 的 PEM 文件路径
- 默认值：“”

12.5.2.15.4 redact-info-log 从 v4.0.8 版本开始引入

- 若开启该选项，日志中的用户数据会以 ? 代替。
- 默认值：false

12.5.2.16 security.encryption

静态加密 (TDE) 有关的配置项。

12.5.2.16.1 data-encryption-method

- 数据文件的加密方法。
- 可选值："plaintext"，"aes128-ctr"，"aes192-ctr"，"aes256-ctr"
- 选择 "plaintext" 以外的值则表示启用加密功能。此时必须指定主密钥。
- 默认值："plaintext"

12.5.2.16.2 data-key-rotation-period

- 指定 TiKV 轮换数据密钥的频率。
- 默认值：7d

12.5.2.16.3 enable-file-dictionary-log

- 启用优化，以减少 TiKV 管理加密元数据时的 I/O 操作和互斥锁竞争。
- 此配置参数默认启用，为避免可能出现的兼容性问题，请参考静态加密 - TiKV 版本间兼容性。
- 默认值：true

12.5.2.16.4 master-key

- 指定启用加密时的主密钥。若要了解如何配置主密钥，可以参考静态加密 - 配置加密。

12.5.2.16.5 previous-master-key

- 指定轮换新主密钥时的旧主密钥。旧主密钥的配置格式与主密钥相同。若要了解如何配置主密钥，可以参考静态加密 - 配置加密。

12.5.2.17 import

用于 TiDB Lightning 导入及 BR 恢复相关的配置项。

12.5.2.17.1 num-threads

- 处理 RPC 请求线程数。
- 默认值：8
- 最小值：1

12.5.2.17.2 num-import-jobs

- 并发导入工作任务数。
- 默认值：8
- 最小值：1

12.5.2.18 backup

用于 BR 备份相关的配置项。

12.5.2.18.1 num-threads

- 处理备份的工作线程数。
- 默认值：CPU * 0.75，但最大为 32
- 最小值：1

12.5.2.19 cdc 从 v4.0.5 版本开始引入

用于 TiCDC 捕捉变更数据相关的配置项。

12.5.2.19.1 min-ts-interval

- 定期推进 Resolved TS 的时间间隔。
- 默认值：1s

12.5.2.19.2 sink-memory-quota

- 缓存在内存中的 TiCDC 数据变更事件占用内存的上限。
- 默认值：512MB

12.5.2.19.3 incremental-scan-threads

- 增量扫描历史数据任务的线程个数。
- 默认值：4，即 4 个线程

12.5.2.19.4 incremental-scan-concurrency

- 增量扫描历史数据任务的最大并发执行个数。
- 默认值：6，即最多并发执行 6 个任务
- 注意：incremental-scan-concurrency 需要大于等于 incremental-scan-threads，否则 TiKV 启动会报错。

12.5.2.20 pessimistic-txn

12.5.2.20.1 enabled

- 开启悲观事务支持，悲观事务使用方法请参考TiDB 悲观事务模式。
- 默认值：true

12.5.2.20.2 wait-for-lock-timeout

- 悲观事务在 TiKV 中等待其他事务释放锁的最长时间。若超时则会返回错误给 TiDB 并由 TiDB 重试加锁，语句最长等锁时间由 innodb_lock_wait_timeout 控制。
- 默认值：1s
- 最小值：1ms

12.5.2.20.3 wake-up-delay-duration

- 悲观事务释放锁时，只会唤醒等锁事务中 start_ts 最小的事务，其他事务将会延迟 wake-up-delay-duration 之后被唤醒。
- 默认值：20ms

12.5.2.20.4 pipelined

- 开启流水线式加悲观锁流程。开启该功能后，TiKV 在检测数据满足加锁要求后，立刻通知 TiDB 执行后面的请求，并异步写入悲观锁，从而降低大部分延迟，显著提升悲观事务的性能。但有较低概率出现悲观锁异步写入失败的情况，可能会导致悲观事务提交失败。
- 默认值：false

12.5.3 TiFlash 配置参数

本文介绍了与部署使用 TiFlash 相关的配置参数。

12.5.3.1 PD 调度参数

可通过pd-ctl 调整参数。如果你使用 TiUP 部署，可以用 tiup ctl:<cluster-version> pd 代替 pd-ctl -u <pd_ip:pd_port> 命令。

- replica-schedule-limit：用来控制 replica 相关 operator 的产生速度（涉及到下线、补副本的操作都与该参数有关）

> 注意：
> 不要超过 region-schedule-limit，否则会影响正常 TiKV 之间的 Region 调度。

- store-balance-rate：用于限制每个 TiKV store 或 TiFlash store 的 Region 调度速度。注意这个参数只对新加入集群的 store 有效，如果想立刻生效请用下面的方式。

> **注意：**
>
> 4.0.2 版本之后（包括 4.0.2 版本）废弃了 `store-balance-rate` 参数且 `store limit` 命令有部分变化。该命令变化的细节请参考store-limit 文档。

- 使用 `pd-ctl -u <pd_ip:pd_port> store limit <store_id> <value>` 命令单独设置某个 store 的 Region 调度速度。（`store_id` 可通过 `pd-ctl -u <pd_ip:pd_port> store` 命令获得）如果没有单独设置，则继承 `store-balance-rate` 的设置。你也可以使用 `pd-ctl -u <pd_ip:pd_port> store limit` 命令查看当前设置值。

12.5.3.2 TiFlash 配置参数

12.5.3.2.1 配置文件 tiflash.toml

```
#### TiFlash TCP/HTTP 等辅助服务的监听 host。建议配置成 0.0.0.0，即监听本机所有 IP 地址。
listen_host = "0.0.0.0"
#### TiFlash TCP 服务的端口
tcp_port = 9000
#### TiFlash HTTP 服务的端口
http_port = 8123
#### 数据块元信息的内存 cache 大小限制，通常不需要修改
mark_cache_size = 5368709120
#### 数据块 min-max 索引的内存 cache 大小限制，通常不需要修改
minmax_index_cache_size = 5368709120

#### TiFlash 数据的存储路径。如果有多个目录，以英文逗号分隔。
#### 从 v4.0.9 版本开始，不推荐使用 path 及 path_realtime_mode 参数。推荐使用 [storage]
    ↪ 下的配置项代替，这样在多盘部署的场景下能更好地利用节点性能。
### path = "/tidb-data/tiflash-9000"
#### 或
### path = "/ssd0/tidb-data/tiflash,/ssd1/tidb-data/tiflash,/ssd2/tidb-data/tiflash"
#### 默认为 false。如果设为 true，且 path 配置了多个目录，表示在第一个目录存放最新数据，
    ↪ 在其他目录存放较旧的数据。
### path_realtime_mode = false

#### TiFlash 临时文件的存放路径。通常使用 [`path` 或者 `storage.latest.dir` 的第一个目录] + "/tmp
    ↪ "
tmp_path = "/tidb-data/tiflash-9000/tmp"

#### 存储路径相关配置，从 v4.0.9 开始生效
[storage]
    [storage.main]
    ## 用于存储主要的数据，该目录列表中的数据占总数据的 90% 以上。
    dir = [ "/tidb-data/tiflash-9000" ]
```

```
    ## 或
    # dir = [ "/ssd0/tidb-data/tiflash", "/ssd1/tidb-data/tiflash" ]

    ## storage.main.dir 存储目录列表中每个目录的最大可用容量。
    ## * 在未定义配置项，或者列表中全填 0 时，会使用目录所在的硬盘容量
    ## * 以 byte 为单位。目前不支持如 "10GB" 的设置
    ## * capacity 列表的长度应当与 dir 列表长度保持一致
    ## 例如:
    # capacity = [ 10737418240, 10737418240 ]

    [storage.latest]
    ## 用于存储最新的数据，大约占总数据量的 10% 以内，需要较高的 IOPS。
    ## 默认情况该项可留空。在未配置或者为空列表的情况下，会使用 storage.main.dir 的值。
    # dir = [ ]
    ## storage.latest.dir 存储目录列表中，每个目录的最大可用容量。
    # capacity = [ 10737418240, 10737418240 ]

[flash]
    tidb_status_addr = tidb status 端口地址 # 多个地址以逗号分割
    service_addr =  TiFlash raft 服务 和 coprocessor 服务监听地址

### 多个 TiFlash 节点会选一个 master 来负责往 PD 增删 placement rule，通过 flash.flash_cluster
    ↪ 中的参数控制。
[flash.flash_cluster]
    refresh_interval = master 定时刷新有效期
    update_rule_interval = master 定时向 tidb 获取 TiFlash 副本状态并与 pd 交互
    master_ttl = master 选出后的有效期
    cluster_manager_path = pd buddy 所在目录的绝对路径
    log = pd buddy log 路径

[flash.proxy]
    addr = proxy 监听地址
    advertise-addr = 外部访问 addr 的地址，不填则默认是 addr
    data-dir = proxy 数据存储路径
    config = proxy 配置文件路径
    log-file = proxy log 路径
    log-level = proxy log 级别，默认为 "info"
    status-addr = 拉取 proxy metrics | status 信息的监听地址
    advertise-status-addr = 外部访问 status-addr 的地址，不填则默认是 status-addr

[logger]
    level = log 级别（支持 trace、debug、information、warning、error）
    log = TiFlash log 路径
    errorlog = TiFlash error log 路径
    size = 单个日志文件的大小
```

```
    count = 最多保留日志文件个数

[raft]
    pd_addr = pd 服务地址 # 多个地址以逗号隔开

[status]
    metrics_port = Prometheus 拉取 metrics 信息的端口

[profiles]

[profiles.default]
    ## 存储引擎的 segment 分裂是否使用逻辑分裂。使用逻辑分裂可以减小写放大，提高写入速度，
        ↪ 但是会造成一定程度的硬盘空间回收不及时。默认为 true
    dt_enable_logical_split = true
    ## 单次 coprocessor 查询过程中，对中间数据的内存限制，单位为 byte，默认为 0，表示不限制
    max_memory_usage = 0
    ## 所有查询过程中，对中间数据的内存限制，单位为 byte，默认为 0，表示不限制
    max_memory_usage_for_all_queries = 0
    ## 从 v4.0.11 引入，表示 TiFlash Coprocessor 最多同时执行的 cop 请求数量。
        ↪ 如果请求数量超过了该配置指定的值，多出的请求会排队等待。如果设为 0 或不设置，
        ↪ 则使用默认值，即物理核数的两倍。
    cop_pool_size = 0
    ## 从 v4.0.11 引入，表示 TiFlash Coprocessor 最多同时执行的 batch 请求数量。
        ↪ 如果请求数量超过了该配置指定的值，多出的请求会排队等待。如果设为 0 或不设置，
        ↪ 则使用默认值，即物理核数的两倍。
    batch_cop_pool_size = 0

#### 安全相关配置，从 v4.0.5 开始生效
[security]
    ## 包含可信 SSL CA 列表的文件路径。如果你设置了该值，`cert_path` 和 `key_path`
        ↪ 中的路径也需要填写
    # ca_path = "/path/to/ca.pem"
    ## 包含 PEM 格式的 X509 certificate 文件路径
    # cert_path = "/path/to/tiflash-server.pem"
    ## 包含 PEM 格式的 X509 key 文件路径
    # key_path = "/path/to/tiflash-server-key.pem"

    ## 从 v4.0.10 引入。若开启该选项，日志中的用户数据会以 `?` 代替显示
    ## 注意，tiflash-learner 对应的安全配置选项为 `security.redact-info-log`，需要在 tiflash-
        ↪ learner.toml 中另外开启
    # redact_info_log = false
```

12.5.3.2.2 配置文件 tiflash-learner.toml

```
[server]
```

```
    engine-addr = 外部访问 TiFlash coprocessor 服务的地址
[raftstore]
    ## 控制处理 snapshot 的线程数，默认为 2。设为 0 则关闭多线程优化
    snap-handle-pool-size = 2
    ## 控制 raft store 持久化 WAL 的最小间隔。通过适当增大延迟以减少 IOPS 占用，默认为 4ms，设为
        ↪ 0ms 则关闭该优化。
    store-batch-retry-recv-timeout = "4ms"
```

除以上几项外，其余功能参数和 TiKV 的配置相同。需要注意的是：`tiflash.toml [flash.proxy]` 中的配置项会覆盖 `tiflash-learner.toml` 中的重合参数；key 为 `engine` 的 `label` 是保留项，不可手动配置。

12.5.3.2.3 多盘部署

TiFlash 支持单节点多盘部署。如果你的部署节点上有多块硬盘，可以通过以下的方式配置参数，提高节点的硬盘 I/O 利用率。TiUP 中参数配置格式参照详细 TiFlash 配置模版。

TiDB 集群版本低于 v4.0.9

TiDB v4.0.9 之前的版本中，TiFlash 只支持将存储引擎中的主要数据分布在多盘上。通过 `path`（TiUP 中为 `data_dir`）和 `path_realtime_mode` 这两个参数配置多盘部署。

多个数据存储目录在 `path` 中以英文逗号分隔，比如 `/nvme_ssd_a/data/tiflash,/sata_ssd_b/data/tiflash,/sata_ssd_c/data/tiflash`。如果你的节点上有多块硬盘，推荐把性能最好的硬盘目录放在最前面，以更好地利用节点性能。

如果节点上有多块相同规格的硬盘，可以把 `path_realtime_mode` 参数留空（或者把该值明确地设为 `false`）。这表示数据会在所有的存储目录之间进行均衡。但由于最新的数据仍然只会被写入到第一个目录，因此该目录所在的硬盘会较其他硬盘繁忙。

如果节点上有多块规格不一致的硬盘，推荐把 `path_relatime_mode` 参数设置为 `true`，并且把性能最好的硬盘目录放在 `path` 参数内的最前面。这表示第一个目录只会存放最新数据，较旧的数据会在其他目录之间进行均衡。注意此情况下，第一个目录规划的容量大小需要占总容量的约 10%。

TiDB 集群版本为 v4.0.9 及以上

TiDB v4.0.9 及之后的版本中，TiFlash 支持将存储引擎的主要数据和新数据都分布在多盘上。多盘部署时，推荐使用 `[storage]` 中的参数，以更好地利用节点的 I/O 性能。但 TiFlash 仍然支持TiDB 集群版本低于 v4.0.9 中的参数。

如果节点上有多块相同规格的硬盘，推荐把硬盘目录填到 `storage.main.dir` 列表中，`storage.latest.dir` 列表留空。TiFlash 会在所有存储目录之间分摊 I/O 压力以及进行数据均衡。

如果节点上有多块规格不同的硬盘，推荐把 I/O 性能较好的硬盘目录配置在 `storage.latest.dir` 中，把 I/O 性能较一般的硬盘目录配置在 `storage.main.dir` 中。例如节点上有一块 NVMe-SSD 硬盘加上两块 SATA-SSD 硬盘，你可以把 `storage.latest.dir` 设为 `["/nvme_ssd_a/data/tiflash"]` 以及把 `storage.main.dir` 设为 `["/sata_ssd_b/data/tiflash", "/sata_ssd_c/data/tiflash"]`。TiFlash 会根据两个目录列表分别进行 I/O 压力分摊及数据均衡。需要注意此情况下，`storage.latest.dir` 中规划的容量大小需要占总规划容量的约 10%。

> **警告：**

- `[storage]` 参数从 TiUP v1.2.5 版本开始支持。如果你的 TiDB 版本为 v4.0.9 及以上，请确保你的 TiUP 版本不低于 v1.2.5，否则 `[storage]` 中定义的数据目录不会被 TiUP 纳入管理。
- 在 TiFlash 节点改为使用 `[storage]` 配置后，如果将集群版本降级到低于 v4.0.9，可能导致 TiFlash 部分数据丢失。

12.5.4 PD 配置文件描述

PD 配置文件比命令行参数支持更多的选项。你可以在 conf/config.toml 找到默认的配置文件。

本文档只阐述未包含在命令行参数中的参数，命令行参数参见PD 配置参数。

12.5.4.0.1 `lease`

- PD Leader Key 租约超时时间，超时系统重新选举 Leader。
- 默认：3
- 单位：秒

12.5.4.0.2 `tso-save-interval`

- TSO 分配的时间窗口, 实时持久存储。
- 默认：3s

12.5.4.0.3 `initial-cluster-state`

- 集群初始状态
- 默认：new

12.5.4.0.4 `enable-prevote`

- 开启 raft prevote 的开关。
- 默认：true

12.5.4.0.5 `quota-backend-bytes`

- 元信息数据库存储空间的大小，默认 8GiB。
- 默认：8589934592

12.5.4.0.6 `auto-compaction-mod`

- 元信息数据库自动压缩的模式，可选项为 periodic（按周期），revision（按版本数）。
- 默认：periodic

12.5.4.0.7 auto-compaction-retention

- compaction-mode 为 periodic 时为元信息数据库自动压缩的间隔时间；compaction-mode 设置为 revision 时为自动压缩的版本数。
- 默认：1h

12.5.4.0.8 force-new-cluster

- 强制让该 PD 以一个新集群启动，且修改 raft 成员数为 1。
- 默认：false

12.5.4.0.9 tick-interval

- etcd raft 的 tick 周期。
- 默认：100ms

12.5.4.0.10 election-interval

- etcd leader 选举的超时时间。
- 默认：3s

12.5.4.0.11 use-region-storage

- 开启独立的 region 存储。
- 默认：false

12.5.4.1 security

安全相关配置项。

12.5.4.1.1 cacert-path

- CA 文件路径
- 默认：“”

12.5.4.1.2 cert-path

- 包含 X509 证书的 PEM 文件路径
- 默认：“”

12.5.4.1.3 key-path

- 包含 X509 key 的 PEM 文件路径
- 默认：“”

12.5.4.1.4 `redact-info-log` 从 v4.0.10 版本开始引入

- 控制 PD 日志脱敏的开关
- 该配置项值设为 true 时将对 PD 日志脱敏，遮蔽日志中的用户信息。
- 默认值：false

12.5.4.2 log

日志相关的配置项。

12.5.4.2.1 `level`

- 日志等级，可指定为“DEBUG”，“INFO”，“WARNING”，“ERROR”，“CRITICAL”。
- 默认值：“INFO”

12.5.4.2.2 `format`

- 日志格式，可指定为“text”，“json”，“console”。
- 默认值：“text”

12.5.4.2.3 `disable-timestamp`

- 是否禁用日志中自动生成的时间戳。
- 默认：false

12.5.4.3 log.file

日志文件相关的配置项。

12.5.4.3.1 `max-size`

- 单个日志文件最大大小，超过该值系统自动切分成多个文件。
- 默认：300
- 单位：MiB
- 最小值为 1

12.5.4.3.2 `max-days`

- 日志保留的最长天数。
- 默认：0

12.5.4.3.3 `max-backups`

- 日志文件保留的最大个数。
- 默认：0

12.5.4.4 metric

监控相关的配置项。

12.5.4.4.1 `interval`

- 向 Prometheus 推送监控指标数据的间隔时间。
- 默认: 15s

12.5.4.5 schedule

调度相关的配置项。

12.5.4.5.1 `max-merge-region-size`

- 控制 Region Merge 的 size 上限，当 Region Size 大于指定值时 PD 不会将其与相邻的 Region 合并。
- 默认: 20

12.5.4.5.2 `max-merge-region-keys`

- 控制 Region Merge 的 key 上限，当 Region key 大于指定值时 PD 不会将其与相邻的 Region 合并。
- 默认: 200000

12.5.4.5.3 `patrol-region-interval`

- 控制 replicaChecker 检查 Region 健康状态的运行频率，越短则运行越快，通常状况不需要调整
- 默认: 100ms

12.5.4.5.4 `split-merge-interval`

- 控制对同一个 Region 做 split 和 merge 操作的间隔，即对于新 split 的 Region 一段时间内不会被 merge。
- 默认: 1h

12.5.4.5.5 `max-snapshot-count`

- 控制单个 store 最多同时接收或发送的 snapshot 数量，调度受制于这个配置来防止抢占正常业务的资源。
- 默认: 3

12.5.4.5.6 `max-pending-peer-count`

- 控制单个 store 的 pending peer 上限，调度受制于这个配置来防止在部分节点产生大量日志落后的 Region。
- 默认： 16

12.5.4.5.7 max-store-down-time

- PD 认为失联 store 无法恢复的时间，当超过指定的时间没有收到 store 的心跳后，PD 会在其他节点补充副本。
- 默认：30m

12.5.4.5.8 leader-schedule-limit

- 同时进行 leader 调度的任务个数。
- 默认：4

12.5.4.5.9 region-schedule-limit

- 同时进行 Region 调度的任务个数
- 默认：2048

12.5.4.5.10 hot-region-schedule-limit

- 控制同时进行的 hot Region 任务。该配置项独立于 Region 调度。
- 默认值：4

12.5.4.5.11 hot-region-cache-hits-threshold

- 设置识别热点 Region 所需的分钟数。只有当 Region 处于热点状态持续时间超过此分钟数时，PD 才会参与热点调度。
- 默认值：3

12.5.4.5.12 replica-schedule-limit

- 同时进行 replica 调度的任务个数。
- 默认：64

12.5.4.5.13 merge-schedule-limit

- 同时进行的 Region Merge 调度的任务，设置为 0 则关闭 Region Merge。
- 默认：8

12.5.4.5.14 high-space-ratio

- 设置 store 空间充裕的阈值。当节点的空间占用比例小于该阈值时，PD 调度时会忽略节点的剩余空间，主要根据实际数据量进行均衡。此配置仅在 `region-score-formula-version = v1` 时生效。
- 默认：0.7
- 最小值：大于 0
- 最大值：小于 1

12.5.4.5.15 low-space-ratio

- 设置 store 空间不足的阈值。当某个节点的空间占用比例超过该阈值时，PD 会尽可能避免往该节点迁移数据，同时主要根据节点剩余空间大小进行调度，避免对应节点的磁盘空间被耗尽。
- 默认：0.8
- 最小值：大于 0
- 最大值：小于 1

12.5.4.5.16 tolerant-size-ratio

- 控制 balance 缓冲区大小。
- 默认：0 (为 0 为自动调整缓冲区大小)
- 最小值：0

12.5.4.5.17 disable-remove-down-replica

- 关闭自动删除 DownReplica 的特性的开关，当设置为 true 时，PD 不会自动清理宕机状态的副本。
- 默认：false

12.5.4.5.18 disable-replace-offline-replica

- 关闭迁移 OfflineReplica 的特性的开关，当设置为 true 时，PD 不会迁移下线状态的副本。
- 默认：false

12.5.4.5.19 disable-make-up-replica

- 关闭补充副本的特性的开关，当设置为 true 时，PD 不会为副本数不足的 Region 补充副本。
- 默认：false

12.5.4.5.20 disable-remove-extra-replica

- 关闭删除多余副本的特性开关，当设置为 true 时，PD 不会为副本数过多的 Region 删除多余副本。
- 默认：false

12.5.4.5.21 disable-location-replacement

- 关闭隔离级别检查的开关，当设置为 true 时，PD 不会通过调度来提升 Region 副本的隔离级别。
- 默认：false

12.5.4.5.22 store-balance-rate

- 控制 TiKV 每分钟最多允许做 add peer 相关操作的次数。
- 默认：15

12.5.4.6 replication

副本相关的配置项。

12.5.4.6.1 max-replicas

- 所有副本数量，即 leader 与 follower 数量之和。默认为 3，即 1 个 leader 和 2 个 follower。
- 默认：3

12.5.4.6.2 location-labels

- TiKV 集群的拓扑信息。
- 默认：[]
- 配置集群拓扑

12.5.4.6.3 strictly-match-label

- 打开强制 TiKV Label 和 PD 的 location-labels 是否匹配的检查
- 默认：false

12.5.4.6.4 enable-placement-rules

- 打开 placement-rules
- 默认：false
- 参考Placement Rules 使用文档
- 4.0 实验性特性

12.5.4.7 label-property

标签相关的配置项。

12.5.4.7.1 key

- 拒绝 leader 的 store 带有的 label key。
- 默认：“”

12.5.4.7.2 value

- 拒绝 leader 的 store 带有的 label value。
- 默认：“”

12.5.4.8 dashboard

PD 中内置的TiDB Dashboard 相关配置项。

12.5.4.8.1 tidb-cacert-path

- CA 根证书文件路径。可配置该路径来使用 TLS 连接 TiDB 的 SQL 服务。
- 默认值：“”

12.5.4.8.2 tidb-cert-path

- SSL 证书文件路径。可配置该路径来使用 TLS 连接 TiDB 的 SQL 服务。
- 默认值：“”

12.5.4.8.3 tidb-key-path

- SSL 私钥文件路径。可配置该路径来使用 TLS 连接 TiDB 的 SQL 服务。
- 默认值：“”

12.5.4.8.4 public-path-prefix

- 通过反向代理访问 TiDB Dashboard 时，配置反向代理提供服务的路径前缀。
- 默认：“/dashboard”
- 若不通过反向代理访问 TiDB Dashboard，请勿配置该项，否则可能导致 TiDB Dashboard 无法正常访问。关于该配置的详细使用场景，参见通过反向代理使用 TiDB Dashboard。

12.5.4.8.5 enable-telemetry

- 是否启用 TiDB Dashboard 遥测功能。
- 默认：true
- 参阅遥测了解该功能详情。

12.6 CLI

12.6.1 TiKV Control 使用说明

TiKV Control（以下简称 tikv-ctl）是 TiKV 的命令行工具，用于管理 TiKV 集群。它的安装目录如下：

- 如果是使用 TiDB Ansible 部署的集群，在 `ansible` 目录下的 `resources/bin` 子目录下。
- 如果是使用 TiUP 部署的集群，在 `~/.tiup/components/ctl/{VERSION}/` 目录下。

12.6.1.1 通过 TiUP 使用 TiKV Control

> **注意：**
>
> 建议使用的 Control 工具版本与集群版本保持一致。

tikv-ctl 也集成在了 tiup 命令中。执行以下命令，即可调用 tikv-ctl 工具：

```
tiup ctl:<cluster-version> tikv
```

```
Starting component `ctl`: /home/tidb/.tiup/components/ctl/v4.0.8/ctl tikv
TiKV Control (tikv-ctl)
Release Version:   4.0.8
Edition:           Community
Git Commit Hash:   83091173e960e5a0f5f417e921a0801d2f6635ae
Git Commit Branch: heads/refs/tags/v4.0.8
UTC Build Time:    2020-10-30 08:40:33
Rust Version:      rustc 1.42.0-nightly (0de96d37f 2019-12-19)
Enable Features:   jemalloc mem-profiling portable sse protobuf-codec
Profile:           dist_release

A tool for interacting with TiKV deployments.

USAGE:
    TiKV Control (tikv-ctl) [FLAGS] [OPTIONS] [SUBCOMMAND]

FLAGS:
    -h, --help                    Prints help information
        --skip-paranoid-checks    Skip paranoid checks when open rocksdb
    -V, --version                 Prints version information

OPTIONS:
        --ca-path <ca_path>              Set the CA certificate path
        --cert-path <cert_path>          Set the certificate path
        --config <config>                Set the config for rocksdb
        --db <db>                        Set the rocksdb path
        --decode <decode>                Decode a key in escaped format
        --encode <encode>                Encode a key in escaped format
        --to-hex <escaped-to-hex>        Convert an escaped key to hex key
        --to-escaped <hex-to-escaped>    Convert a hex key to escaped key
        --host <host>                    Set the remote host
        --key-path <key_path>            Set the private key path
        --pd <pd>                        Set the address of pd
        --raftdb <raftdb>                Set the raft rocksdb path

SUBCOMMANDS:
    bad-regions           Get all regions with corrupt raft
    cluster               Print the cluster id
    compact               Compact a column family in a specified range
    compact-cluster       Compact the whole cluster in a specified range in one or more column
        ↪ families
    consistency-check     Force a consistency-check for a specified region
```

```
    decrypt-file          Decrypt an encrypted file
    diff                  Calculate difference of region keys from different dbs
    dump-snap-meta        Dump snapshot meta file
    encryption-meta       Dump encryption metadata
    fail                  Inject failures to TiKV and recovery
    help                  Prints this message or the help of the given subcommand(s)
    metrics               Print the metrics
    modify-tikv-config    Modify tikv config, eg. tikv-ctl --host ip:port modify-tikv-config -n
                          rocksdb.defaultcf.disable-auto-compactions -v true
    mvcc                  Print the mvcc value
    print                 Print the raw value
    raft                  Print a raft log entry
    raw-scan              Print all raw keys in the range
    recover-mvcc          Recover mvcc data on one node by deleting corrupted keys
    recreate-region       Recreate a region with given metadata, but alloc new id for it
    region-properties     Show region properties
    scan                  Print the range db range
    size                  Print region size
    split-region          Split the region
    store                 Print the store id
    tombstone             Set some regions on the node to tombstone by manual
    unsafe-recover        Unsafely recover the cluster when the majority replicas are failed
```

你可以在 `tiup ctl:<cluster-version> tikv` 后面再接上相应的参数与子命令。

12.6.1.2 通用参数

tikv-ctl 提供以下两种运行模式：

- 远程模式。通过 `--host` 选项接受 TiKV 的服务地址作为参数。在此模式下，如果 TiKV 启用了 SSL，则 tikv-ctl 也需要指定相关的证书文件，例如：

```
tikv-ctl --ca-path ca.pem --cert-path client.pem --key-path client-key.pem --host
    ↪ 127.0.0.1:20160 <subcommands>
```

  某些情况下，tikv-ctl 与 PD 进行通信，而不与 TiKV 通信。此时你需要使用 `--pd` 选项而非 `--host` 选项，例如：

```
tikv-ctl --pd 127.0.0.1:2379 compact-cluster
```

```
store:"127.0.0.1:20160" compact db:KV cf:default range:([], []) success!
```

- 本地模式。通过 `--db` 选项来指定本地 TiKV 数据的目录路径。在此模式下，需要停止正在运行的 TiKV 实例。

以下如无特殊说明，所有命令都同时支持这两种模式。

除此之外，tikv-ctl 还有两个简单的命令 `--to-hex` 和 `--to-escaped`，用于对 key 的形式作简单的变换。一般使用 escaped 形式，示例如下：

```
tikv-ctl --to-escaped 0xaaff
```

```
\252\377
```

```
tikv-ctl --to-hex "\252\377"
```

```
AAFF
```

> 注意：
>
> 在命令行上指定 escaped 形式的 key 时，需要用双引号引起来，否则 bash 会将反斜杠吃掉，导致结果错误。

12.6.1.3 各项子命令及部分参数、选项

下面逐一对 tikv-ctl 支持的子命令进行举例说明。有的子命令支持很多可选参数，要查看全部细节，可运行 `tikv-ctl --help <subcommand>`。

12.6.1.3.1 查看 Raft 状态机的信息

`raft` 子命令可以查看 Raft 状态机在某一时刻的状态。状态信息包括 RegionLocalState、 RaftLocalState 和 RegionApplyState 三个结构体，及某一条 log 对应的 Entries。

你可以使用 `region` 和 `log` 两个子命令分别查询以上信息。两条子命令都同时支持远程模式和本地模式。其用法及输出内容如下所示：

```
tikv-ctl --host 127.0.0.1:20160 raft region -r 2
```

```
region id: 2
region state key: \001\003\000\000\000\000\000\000\000\002\001
region state: Some(region {id: 2 region_epoch {conf_ver: 3 version: 1} peers {id: 3 store_id: 1}
    ↪ peers {id: 5 store_id: 4} peers {id: 7 store_id: 6}})
raft state key: \001\002\000\000\000\000\000\000\000\002\002
raft state: Some(hard_state {term: 307 vote: 5 commit: 314617} last_index: 314617)
apply state key: \001\002\000\000\000\000\000\000\000\002\003
apply state: Some(applied_index: 314617 truncated_state {index: 313474 term: 151})
```

12.6.1.3.2 查看 Region 的大小

使用 `size` 命令可以查看 Region 的大小：

```
tikv-ctl --db /path/to/tikv/db size -r 2
```

```
region id: 2
cf default region size: 799.703 MB
cf write region size: 41.250 MB
cf lock region size: 27616
```

12.6.1.3.3 扫描查看给定范围的 MVCC

scan 命令的 --from 和 --to 参数接受两个 escaped 形式的 raw key，并用 --show-cf 参数指定只需要查看哪些列族。

```
tikv-ctl --db /path/to/tikv/db scan --from 'zm' --limit 2 --show-cf lock,default,write
```

```
key: zmBootstr\377a\377pKey\000\000\377\000\000\373\000\000\000\000\000\377\000\000s
    ↪ \000\000\000\000\000\372
        write cf value: start_ts: 399650102814441473 commit_ts: 399650102814441475 short_value:
            ↪ "20"
key: zmDB:29\000\000\377\000\374\000\000\000\000\000\000\377\000H\000\000\000\000\000\000\371
        write cf value: start_ts: 399650105239273474 commit_ts: 399650105239273475 short_value:
            ↪ "\000\000\000\000\000\000\000\002"
        write cf value: start_ts: 399650105199951882 commit_ts: 399650105213059076 short_value:
            ↪ "\000\000\000\000\000\000\000\001"
```

12.6.1.3.4 查看给定 key 的 MVCC

与 scan 命令类似，mvcc 命令可以查看给定 key 的 MVCC：

```
tikv-ctl --db /path/to/tikv/db mvcc -k "zmDB
    ↪ :29\000\000\377\000\374\000\000\000\000\000\000\377\000H\000\000\000\000\000\000\371" --
    ↪ show-cf=lock,write,default
```

```
key: zmDB:29\000\000\377\000\374\000\000\000\000\000\000\377\000H\000\000\000\000\000\000\371
        write cf value: start_ts: 399650105239273474 commit_ts: 399650105239273475 short_value:
            ↪ "\000\000\000\000\000\000\000\002"
        write cf value: start_ts: 399650105199951882 commit_ts: 399650105213059076 short_value:
            ↪ "\000\000\000\000\000\000\000\001"
```

> 注意：
>
> 该命令中，key 同样需要是 escaped 形式的 raw key。

12.6.1.3.5 打印某个 key 的值

打印某个 key 的值需要用到 `print` 命令。示例从略。

12.6.1.3.6 打印 Region 的 properties 信息

为了记录 Region 的状态信息，TiKV 将一些数据写入 Region 的 SST 文件中。你可以用子命令 `region-properties` 运行 tikv-ctl 来查看这些 properties 信息。例如：

```
tikv-ctl --host localhost:20160 region-properties -r 2
```

```
num_files: 0
num_entries: 0
num_deletes: 0
mvcc.min_ts: 18446744073709551615
mvcc.max_ts: 0
mvcc.num_rows: 0
mvcc.num_puts: 0
mvcc.num_versions: 0
mvcc.max_row_versions: 0
middle_key_by_approximate_size:
```

这些 properties 信息可以用于检查某个 Region 是否健康或者修复不健康的 Region。例如，使用 `middle_key_approximate_size` 可以手动分裂 Region。

12.6.1.3.7 手动 compact 单个 TiKV 的数据

`compact` 命令可以对单个 TiKV 进行手动 compact。如果指定 `--from` 和 `--to` 选项，那么它们的参数也是 escaped raw key 形式的。

- `--host` 参数可以指定要 compact 的 TiKV。
- `-d` 参数可以指定要 compact 的 RocksDB，有 `kv` 和 `raft` 参数值可以选。
- `--threads` 参数可以指定 compact 的并发数，默认值是 8。一般来说，并发数越大，compact 的速度越快，但是也会对服务造成影响，所以需要根据情况选择合适的并发数。
- `--bottommost` 参数可以指定 compact 是否包括最下层的文件。可选值为 `default`、`skip` 和 `force`，默认为 `default`。
  - `default` 表示只有开启了 Compaction Filter 时 compact 才会包括最下层文件。
  - `skip` 表示 compact 不包括最下层文件。
  - `force` 表示 compact 总是包括最下层文件。

```
tikv-ctl --host 127.0.0.1:20160 compact -d kv
```

```
success!
```

12.6.1.3.8 手动 compact 整个 TiKV 集群的数据

`compact-cluster` 命令可以对整个 TiKV 集群进行手动 compact。该命令参数的含义和使用与 `compact` 命令一样。

12.6.1.3.9 设置一个 Region 副本为 tombstone 状态

tombstone 命令常用于没有开启 sync-log，因为机器掉电导致 Raft 状态机丢失部分写入的情况。它可以在一个 TiKV 实例上将一些 Region 的副本设置为 Tombstone 状态，从而在重启时跳过这些 Region，避免因为这些 Region 的副本的 Raft 状态机损坏而无法启动服务。这些 Region 应该在其他 TiKV 上有足够多的健康的副本以便能够继续通过 Raft 机制进行读写。

一般情况下，可以先在 PD 上将 Region 的副本通过 `remove-peer` 命令删除掉：

```
pd-ctl>> operator add remove-peer <region_id> <store_id>
```

然后再用 tikv-ctl 在那个 TiKV 实例上将 Region 的副本标记为 tombstone 以便跳过启动时对他的健康检查：

```
tikv-ctl --db /path/to/tikv/db tombstone -p 127.0.0.1:2379 -r <region_id>
```

```
success!
```

但是有些情况下，当不能方便地从 PD 上移除这个副本时，可以指定 tikv-ctl 的 `--force` 选项来强制设置它为 tombstone：

```
tikv-ctl --db /path/to/tikv/db tombstone -p 127.0.0.1:2379 -r <region_id>,<region_id> --force
```

```
success!
```

> 注意：
>
> - 该命令只支持本地模式
> - -p 选项的参数指定 PD 的 endpoints，无需 `http` 前缀。指定 PD 的 endpoints 是为了询问 PD 是否可以安全切换至 Tombstone 状态。

12.6.1.3.10 向 TiKV 发出 consistency-check 请求

`consistency-check` 命令用于在某个 Region 对应的 Raft 副本之间进行一致性检查。如果检查失败，TiKV 自身会 panic。如果 `--host` 指定的 TiKV 不是这个 Region 的 Leader，则会报告错误。

```
tikv-ctl --host 127.0.0.1:20160 consistency-check -r 2
```

```
success!
```

```
tikv-ctl --host 127.0.0.1:20161 consistency-check -r 2
```

```
DebugClient::check_region_consistency: RpcFailure(RpcStatus { status: Unknown, details: Some("
    ↪ StringError(\"Leader is on store 1\")") })
```

注意：

- 该命令只支持远程模式。
- 即使该命令返回了成功信息，也需要检查是否有 TiKV panic 了。因为该命令只是向 Leader 请求进行一致性检查，但整个检查流程是否成功并不能在客户端知道。

12.6.1.3.11 Dump snapshot 元文件

这条子命令可以用于解析指定路径下的 Snapshot 元文件并打印结果。

12.6.1.3.12 打印 Raft 状态机出错的 Region

前面 `tombstone` 命令可以将 Raft 状态机出错的 Region 设置为 Tombstone 状态，避免 TiKV 启动时对它们进行检查。在运行 `tombstone` 命令之前，可使用 `bad-regions` 命令找到出错的 Region，以便将多个工具组合起来进行自动化的处理。

```
tikv-ctl --db /path/to/tikv/db bad-regions
```

```
all regions are healthy
```

命令执行成功后会打印以上信息，否则会打印出有错误的 Region 列表。目前可以检出的错误包括 `last index`、`commit index` 和 `apply index` 之间的不匹配，以及 Raft log 的丢失。其他一些情况，比如 Snapshot 文件损坏等仍然需要后续的支持。

12.6.1.3.13 查看 Region 属性

本地查看部署在 `/path/to/tikv` 的 TiKV 上面 Region 2 的 properties 信息：

```
tikv-ctl --db /path/to/tikv/data/db region-properties -r 2
```

在线查看运行在 `127.0.0.1:20160` 的 TiKV 上面 Region 2 的 properties 信息：

```
tikv-ctl --host 127.0.0.1:20160 region-properties -r 2
```

12.6.1.3.14 动态修改 TiKV 的配置

使用 `modify-tikv-config` 命令可以动态修改配置参数。目前可动态修改的 TiKV 配置与具体的修改行为与 SQL 动态修改配置功能相同，可参考在线修改 TiKV 配置。

- `-n` 用于指定完整的配置名。支持动态修改的配置名可以参考在线修改 TiKV 配置中支持的配置项列表。
- `-v` 用于指定配置值。

设置 `shared block cache` 的大小：

```
tikv-ctl --host ip:port modify-tikv-config -n storage.block-cache.capacity -v 10GB
```

```
success
```

当禁用 shared block cache 时，为 write CF 设置 block cache size：

```
tikv-ctl --host ip:port modify-tikv-config -n rocksdb.writecf.block-cache-size -v 256MB
```

```
success
```

```
tikv-ctl --host ip:port modify-tikv-config -n raftdb.defaultcf.disable-auto-compactions -v true
```

```
success
```

```
tikv-ctl --host ip:port modify-tikv-config -n raftstore.sync-log -v false
```

```
success
```

12.6.1.3.15 强制 Region 从多副本失败状态恢复服务（慎用）

unsafe-recover remove-fail-stores 命令可以将故障机器从指定 Region 的 peer 列表中移除。运行命令之前，需要目标 TiKV 先停掉服务以便释放文件锁。

-s 选项接受多个以逗号分隔的 store_id，并使用 -r 参数来指定包含的 Region。如果要对某一个 store 上的全部 Region 都执行这个操作，可简单指定 --all-regions。

> 警告：
>
> - 此功能使用不当可能导致集群难以恢复，存在风险。请悉知潜在的风险，尽量避免在生产环境中使用。
> - 如果使用 --all-regions，必须在剩余所有连入集群的 store 上执行此命令。需要保证这些健康的 store 都停掉服务后再进行恢复，否则期间 Region 副本之间的 peer 列表不一致会导致执行 split-region 或者 remove-peer 时报错进而引起其他元数据的不一致，最终引发 Region 不可用。
> - 一旦执行了 remove-fail-stores，不可再重新启动被移除的节点并将其加入集群，否则会导致元数据的不一致，最终引发 Region 不可用。

```
tikv-ctl --db /path/to/tikv/db unsafe-recover remove-fail-stores -s 3 -r 1001,1002
```

```
success!
```

```
tikv-ctl --db /path/to/tikv/db unsafe-recover remove-fail-stores -s 4,5 --all-regions
```

之后启动 TiKV，这些 Region 便可以使用剩下的健康副本继续提供服务了。此命令常用于多个 TiKV store 损坏或被删除的情况。

> 注意：
>
> - 一般来说，您需要为指定 Region 的 peers 所在的每个 store 运行此命令。
> - 该命令只支持本地模式。在运行成功后，会打印 `success!`。

12.6.1.3.16 恢复损坏的 MVCC 数据

`recover-mvcc` 命令用于 MVCC 数据损坏导致 TiKV 无法正常运行的情况。为了从不同种类的不一致情况中恢复，该命令会交叉检查 3 个 CF（“default”，“write”，“lock”）。

- `-r` 选项可以通过 `region_id` 指定包含的 Region。
- `-p` 选项可以指定 PD 的 endpoints。

```
tikv-ctl --db /path/to/tikv/db recover-mvcc -r 1001,1002 -p 127.0.0.1:2379
```

```
success!
```

> 注意：
>
> - 该命令只支持本地模式。在运行成功后，会打印 `success!`。
> - `-p` 选项指定 PD 的 endpoint，不使用 `http` 前缀，用于查询指定的 `region_id` 是否有效。
> - 对于指定 Region 的 peers 所在的每个 store，均须执行该命令。

12.6.1.3.17 Ldb 命令

`ldb` 命令行工具提供多种数据访问以及数据库管理命令。下方列出了一些示例用法。详细信息请在运行 `tikv-ctl ldb` 命令时查看帮助消息或查阅 RocksDB 文档。

数据访问序列的示例如下：

用 HEX 格式 dump 现有 RocksDB 数据：

```
tikv-ctl ldb --hex --db=/tmp/db dump
```

Dump 现有 RocksDB 的声明：

```
tikv-ctl ldb --hex manifest_dump --path=/tmp/db/MANIFEST-000001
```

您可以通过 `--column_family=<string>` 指定查询的目标列族。

通过 `--try_load_options` 命令加载数据库选项文件以打开数据库。在数据库运行时，建议您保持该命令为开启的状态。如果您使用默认配置打开数据库，LSM-tree 存储组织可能会出现混乱，且无法自动恢复。

12.6.1.3.18 打印加密元数据

`encryption-meta` 命令用于打印加密元数据。该子命令可以打印两种加密元数据：数据文件的加密信息，以及所有的数据加密密钥。

使用 `encryption-meta dump-file` 子命令打印数据文件的加密信息。你需要创建一个 TiKV 配置文件用以指定 TiKV 的数据目录：

```
### conf.toml
[storage]
data-dir = "/path/to/tikv/data"
```

`--path` 选项可以指定数据文件的绝对或者相对路径。如果指定的文件是明文存储的，本命令有可能没有输出。如果不指定 `--path` 选项，本命令打印所有数据文件的加密信息。

```
$ tikv-ctl --config=./conf.toml encryption-meta dump-file --path=/path/to/tikv/data/db/CURRENT
/path/to/tikv/data/db/CURRENT: key_id: 9291156302549018620 iv: E3C2FDBF63FC03BFC28F265D7E78283F
    ↪ method: Aes128Ctr
```

使用 `encryption-meta dump-key` 打印数据加密密钥。使用本命令的时候，除了在 TiKV 配置文件中指定 TiKV 的数据目录以外，还需要指定当前的主加密密钥。请参阅静态加密文档关于配置 TiKV 主加密密钥的说明。使用本命令时 `security.encryption.previous-master-key` 配置项不生效，即使配置文件中使用了该配置，本命令也不会触发更换主加密密钥。

```
### conf.toml
[storage]
data-dir = "/path/to/tikv/data"

[security.encryption.master-key]
type = "kms"
key-id = "0987dcba-09fe-87dc-65ba-ab0987654321"
region = "us-west-2"
```

注意如果使用了 AWS KMS 作为主加密密钥，使用本命令时 `tikv-ctl` 需要该 KMS 密钥的访问权限。KMS 访问权限可以通过环境变量、AWS 默认配置文件或 IAM 的方式传递给 `tikv-ctl`。详情请参阅相关 AWS 文档。

`--ids` 选项可以指定以逗号分隔的数据加密密钥 id 列表。如果不指定 `--ids` 选项，本命令打印所有的数据加密密钥，以及最新的数据加密密钥的 id。

本命令会输出一个警告，提示本命令会泄漏敏感数据。根据提示输入 “I consent” 即可。

```
$ ./tikv-ctl --config=./conf.toml encryption-meta dump-key
This action will expose encryption key(s) as plaintext. Do not output the result in file on disk.
Type "I consent" to continue, anything else to exit: I consent
current key id: 9291156302549018620
9291156302549018620: key: 8B6B6B8F83D36BE2467ED55D72AE808B method: Aes128Ctr creation_time:
    ↪ 1592938357
```

```
$ ./tikv-ctl --config=./conf.toml encryption-meta dump-key --ids=9291156302549018620
This action will expose encryption key(s) as plaintext. Do not output the result in file on disk.
```

```
Type "I consent" to continue, anything else to exit: I consent
9291156302549018620: key: 8B6B6B8F83D36BE2467ED55D72AE808B method: Aes128Ctr creation_time:
    ↪ 1592938357
```

> **注意：**
>
> 本命令会以明文方式打印数据加密密钥。在生产环境中，请勿将本命令的输出重定向到磁盘文件中。即使使用以后删除该文件也不能保证文件内容从磁盘中干净清除。

12.6.2 PD Control 使用说明

PD Control 是 PD 的命令行工具，用于获取集群状态信息和调整集群。

12.6.2.1 安装方式

> **注意：**
>
> 建议使用的 Control 工具版本与集群版本保持一致。

12.6.2.1.1 使用 TiUP

可直接通过 `tiup ctl:<cluster-version> pd -u http://<pd_ip>:<pd_port> [-i]` 使用。

12.6.2.1.2 下载安装包

如需下载最新版本的 `pd-ctl`，直接下载 TiDB 安装包即可，因为 `pd-ctl` 包含在 TiDB 安装包中。

安装包	操作系统	架构	SHA256 校验和
https ↪ ://download ↪ .pingcap ↪ .org ↪ /tidb ↪ -{version}- ↪ linux ↪ -amd64 ↪ .tar ↪ .gz	Linux	amd64	https ↪ ://download ↪ .pingcap ↪ .org ↪ /tidb ↪ -{version}- ↪ linux ↪ -amd64 ↪ .sha256 ↪
(pd-ctl)			

> **注意：**
>
> 下载链接中的 `{version}` 为 TiDB 的版本号。例如 `v4.0.0-rc.2` 版本的下载链接为 `https://download.pingcap.org/tidb-v4.0.0-rc.2-linux-amd64.tar.gz`。也可以使用 `latest` 替代 `{version}` 来下载最新的未发布版本。

12.6.2.1.3 源码编译

1. Go Version 1.13 以上
2. 在 PD 项目根目录使用 `make` 或者 `make pd-ctl` 命令进行编译，生成 bin/pd-ctl

12.6.2.2 简单例子

单命令模式：

```
./pd-ctl store -u http://127.0.0.1:2379
```

交互模式：

```
./pd-ctl -i -u http://127.0.0.1:2379
```

使用环境变量：

```
export PD_ADDR=http://127.0.0.1:2379 &&
./pd-ctl
```

使用 TLS 加密：

```
./pd-ctl -u https://127.0.0.1:2379 --cacert="path/to/ca" --cert="path/to/cert" --key="path/to/key
    ↪ "
```

12.6.2.3 命令行参数 (flags)

12.6.2.3.1 --cacert

- 指定 PEM 格式的受信任 CA 证书的文件路径
- 默认值： “”

12.6.2.3.2 --cert

- 指定 PEM 格式的 SSL 证书的文件路径
- 默认值： “”

12.6.2.3.3 --detach / -d

- 使用单命令行模式 (不进入 readline)
- 默认值: true

12.6.2.3.4 --help / -h

- 输出帮助信息
- 默认值： false

12.6.2.3.5 --interact / -i

- 使用交互模式 (进入 readline)
- 默认值： false

12.6.2.3.6 --key

- 指定 PEM 格式的 SSL 证书密钥文件路径，即 --cert 所指定的证书的私钥
- 默认值: “”

12.6.2.3.7 --pd / -u

- 指定 PD 的地址
- 默认地址：http://127.0.0.1:2379
- 环境变量：PD_ADDR

12.6.2.3.8 --version / -V

- 打印版本信息并退出
- 默认值: false

12.6.2.4 命令 (command)

12.6.2.4.1 cluster

用于显示集群基本信息。

示例：

```
>> cluster
```

```
{
  "id": 6493707687106161130,
  "max_peer_count": 3
}
```

12.6.2.4.2 config [show | set <option> <value> | placement-rules]

用于显示或调整配置信息。示例如下。

显示 scheduling 的相关 config 信息：

```
>> config show
```

```
{
  "replication": {
    "enable-placement-rules": "false",
    "location-labels": "",
    "max-replicas": 3,
    "strictly-match-label": "false"
  },
  "schedule": {
    "enable-cross-table-merge": "false",
    "enable-debug-metrics": "false",
    "enable-location-replacement": "true",
    "enable-make-up-replica": "true",
```

```
    "enable-one-way-merge": "false",
    "enable-remove-down-replica": "true",
    "enable-remove-extra-replica": "true",
    "enable-replace-offline-replica": "true",
    "high-space-ratio": 0.7,
    "hot-region-cache-hits-threshold": 3,
    "hot-region-schedule-limit": 4,
    "leader-schedule-limit": 4,
    "leader-schedule-policy": "count",
    "low-space-ratio": 0.8,
    "max-merge-region-keys": 200000,
    "max-merge-region-size": 20,
    "max-pending-peer-count": 16,
    "max-snapshot-count": 3,
    "max-store-down-time": "30m0s",
    "merge-schedule-limit": 8,
    "patrol-region-interval": "100ms",
    "region-schedule-limit": 2048,
    "replica-schedule-limit": 64,
    "scheduler-max-waiting-operator": 5,
    "split-merge-interval": "1h0m0s",
    "tolerant-size-ratio": 0
  }
}
```

显示所有的 config 信息：

```
>> config show all
```

显示 replication 的相关 config 信息：

```
>> config show replication
```

```
{
  "max-replicas": 3,
  "location-labels": "",
  "strictly-match-label": "false",
  "enable-placement-rules": "false"
}
```

显示目前集群版本，是目前集群 TiKV 节点的最低版本，并不对应 binary 的版本：

```
>> config show cluster-version
```

```
"4.0.0"
```

- max-snapshot-count 控制单个 store 最多同时接收或发送的 snapshot 数量，调度受制于这个配置来防止抢占正常业务的资源。当需要加快补副本或 balance 速度时可以调大这个值。

  设置最大 snapshot 为 16：

```
>> config set max-snapshot-count 16
```

- max-pending-peer-count 控制单个 store 的 pending peer 上限，调度受制于这个配置来防止在部分节点产生大量日志落后的 Region。需要加快补副本或 balance 速度可以适当调大这个值，设置为 0 则表示不限制。

  设置最大 pending peer 数量为 64：

```
>> config set max-pending-peer-count 64
```

- max-merge-region-size 控制 Region Merge 的 size 上限（单位是 M）。当 Region Size 大于指定值时 PD 不会将其与相邻的 Region 合并。设置为 0 表示不开启 Region Merge 功能。

  设置 Region Merge 的 size 上限为 16 M：

```
>> config set max-merge-region-size 16
```

- max-merge-region-keys 控制 Region Merge 的 keyCount 上限。当 Region KeyCount 大于指定值时 PD 不会将其与相邻的 Region 合并。

  设置 Region Merge 的 keyCount 上限为 50000：

```
>> config set max-merge-region-keys 50000
```

- split-merge-interval 控制对同一个 Region 做 split 和 merge 操作的间隔，即对于新 split 的 Region 一段时间内不会被 merge。

  设置 split 和 merge 的间隔为 1 天：

```
>> config set split-merge-interval 24h
```

- enable-one-way-merge 用于控制是否只允许和相邻的后一个 Region 进行合并。当设置为 false 时，PD 允许与相邻的前后 Region 进行合并。

  设置只允许和相邻的后一个 Region 合并：

```
>> config set enable-one-way-merge true
```

- enable-cross-table-merge 用于开启跨表 Region 的合并。当设置为 false 时，PD 不会合并不同表的 Region。该选项只在键类型为“table”时生效。

  设置允许跨表合并：

```
>> config set enable-cross-table-merge true
```

- key-type 用于指定集群的键编码类型。支持的类型有 ["table", "raw", "txn"]，默认值为“table”。

  - 如果集群中不存在 TiDB 实例，key-type 的值为“raw”或“txn”。此时，无论 enable-cross-table-merge 设置为何，PD 均可以跨表合并 Region。

  - 如果集群中存在 TiDB 实例，key-type 的值应当为 “table”。此时，enable-cross-table-merge 的设置决定了 PD 是否能跨表合并 Region。如果 key-type 的值为 “raw”，placement rules 不生效。

  启用跨表合并：

```
>> config set key-type raw
```

- patrol-region-interval 控制 replicaChecker 检查 Region 健康状态的运行频率，越短则运行越快，通常状况不需要调整。

  设置 replicaChecker 的运行频率为 50 毫秒：

```
>> config set patrol-region-interval 50ms
```

- max-store-down-time 为 PD 认为失联 store 无法恢复的时间，当超过指定的时间没有收到 store 的心跳后，PD 会在其他节点补充副本。

  设置 store 心跳丢失 30 分钟开始补副本：

```
>> config set max-store-down-time 30m
```

- 通过调整 leader-schedule-limit 可以控制同时进行 leader 调度的任务个数。这个值主要影响 leader balance 的速度，值越大调度得越快，设置为 0 则关闭调度。Leader 调度的开销较小，需要的时候可以适当调大。

  最多同时进行 4 个 leader 调度：

```
>> config set leader-schedule-limit 4
```

- 通过调整 region-schedule-limit 可以控制同时进行 Region 调度的任务个数。这个值可以避免创建过多的 Region balance operator。默认值为 2048，对所有大小的集群都足够。设置为 0 则关闭调度。Region 调度的速度通常受到 store-limit 的限制，但除非你熟悉该设置，否则不推荐自定义该参数。

  最多同时进行 2 个 Region 调度：

```
>> config set region-schedule-limit 2
```

- 通过调整 replica-schedule-limit 可以控制同时进行 replica 调度的任务个数。这个值主要影响节点挂掉或者下线的时候进行调度的速度，值越大调度得越快，设置为 0 则关闭调度。Replica 调度的开销较大，所以这个值不宜调得太大。

  最多同时进行 4 个 replica 调度：

```
>> config set replica-schedule-limit 4
```

- merge-schedule-limit 控制同时进行的 Region Merge 调度的任务，设置为 0 则关闭 Region Merge。Merge 调度的开销较大，所以这个值不宜调得过大。

  最多同时进行 16 个 merge 调度：

```
>> config set merge-schedule-limit 16
```

- `hot-region-schedule-limit` 控制同时进行的 Hot Region 调度的任务，设置为 0 则关闭调度。这个值不宜调得过大，否则可能对系统性能造成影响。

  最多同时进行 4 个 Hot Region 调度：

```
>> config set hot-region-schedule-limit 4
```

- `hot-region-cache-hits-threshold` 用于设置识别热点 Region 所需的分钟数，只有 Region 处于热点状态持续时间超过该分钟数后，才能参与热点调度。

- `tolerant-size-ratio` 控制 balance 缓冲区大小。当两个 store 的 leader 或 Region 的得分差距小于指定倍数的 Region size 时，PD 会认为此时 balance 达到均衡状态。

  设置缓冲区为约 20 倍平均 RegionSize：

```
>> config set tolerant-size-ratio 20
```

- `low-space-ratio` 用于设置 store 空间不足的阈值。当节点的空间占用比例超过指定值时，PD 会尽可能避免往对应节点迁移数据，同时主要针对剩余空间大小进行调度，避免对应节点磁盘空间被耗尽。

  设置空间不足阈值为 0.9：

```
config set low-space-ratio 0.9
```

- `high-space-ratio` 用于设置 store 空间充裕的阈值。当节点的空间占用比例小于指定值时，PD 调度时会忽略剩余空间这个指标，主要针对实际数据量进行均衡。

  设置空间充裕阈值为 0.5：

```
config set high-space-ratio 0.5
```

- `cluster-version` 集群的版本，用于控制某些 Feature 是否开启，处理兼容性问题。通常是集群正常运行的所有 TiKV 节点中的最低版本，需要回滚到更低的版本时才进行手动设置。

  设置 cluster version 为 1.0.8：

```
config set cluster-version 1.0.8
```

- `leader-schedule-policy` 用于选择 Leader 的调度策略，可以选择按照 `size` 或者 `count` 来进行调度。

- `scheduler-max-waiting-operator` 用于控制每个调度器同时存在的 operator 的个数。

- `enable-remove-down-replica` 用于开启自动删除 DownReplica 的特性。当设置为 false 时，PD 不会自动清理宕机状态的副本。

- `enable-replace-offline-replica` 用于开启迁移 OfflineReplica 的特性。当设置为 false 时，PD 不会迁移下线状态的副本。

- `enable-make-up-replica` 用于开启补充副本的特性。当设置为 false 时，PD 不会为副本数不足的 Region 补充副本。

- `enable-remove-extra-replica` 用于开启删除多余副本的特性。当设置为 false 时，PD 不会为副本数过多的 Region 删除多余副本。

- enable-location-replacement 用于开启隔离级别检查。当设置为 false 时，PD 不会通过调度来提升 Region 副本的隔离级别。
- enable-debug-metrics 用于开启 debug 的 metrics。当设置为 true 时，PD 会开启一些 metrics，比如 balance-tolerant-size 等。
- enable-placement-rules 用于开启 placement rules。
- store-limit-mode 用于控制 store 限速机制的模式。主要有两种模式：auto 和 manual。auto 模式下会根据 load 自动进行平衡调整（实验性功能）。

12.6.2.4.3 config placement-rules [disable | enable | load | save | show]

关于 config placement-rules 的具体用法，参考Placement Rules 使用文档。

12.6.2.4.4 health

用于显示集群健康信息。示例如下。

显示健康信息：

```
>> health
```

```
[
  {
    "name": "pd",
    "member_id": 13195394291058371180,
    "client_urls": [
      "http://127.0.0.1:2379"
      ......
    ],
    "health": true
  }
  ......
]
```

12.6.2.4.5 hot [read | write | store]

用于显示集群热点信息。示例如下。

显示读热点信息：

```
>> hot read
```

显示写热点信息：

```
>> hot write
```

显示所有 store 的读写信息：

```
>> hot store
```

12.6.2.4.6 label [store <name> <value>]

用于显示集群标签信息。示例如下。

显示所有 label：

```
>> label
```

显示所有包含 label 为 “zone”：“cn” 的 store：

```
>> label store zone cn
```

12.6.2.4.7 member [delete | leader_priority | leader [show | resign | transfer <member_name>]]

用于显示 PD 成员信息，删除指定成员，设置成员的 leader 优先级。示例如下。

显示所有成员的信息：

```
>> member
```

```
{
  "header": {......},
  "members": [......],
  "leader": {......},
  "etcd_leader": {......},
}
```

下线 “pd2”：

```
>> member delete name pd2
```

```
Success!
```

使用 id 下线节点：

```
>> member delete id 1319539429105371180
```

```
Success!
```

显示 leader 的信息：

```
>> member leader show
```

```
{
  "name": "pd",
  "member_id": 13155432540099656863,
  "peer_urls": [......],
  "client_urls": [......]
}
```

将 leader 从当前成员移走：

```
>> member leader resign
```

```
......
```

将 leader 迁移至指定成员：

```
>> member leader transfer pd3
```

```
......
```

12.6.2.4.8 operator [check | show | add | remove]

用于显示和控制调度操作。

示例：

```
>> operator show                                    // 显示所有的 operators
>> operator show admin                              // 显示所有的 admin operators
>> operator show leader                             // 显示所有的 leader operators
>> operator show region                             // 显示所有的 Region operators
>> operator add add-peer 1 2                        // 在 store 2 上新增 Region 1 的一个副本
>> operator add add-learner 1 2                     // 在 store 2 上新增 Region 1 的一个
    ↪ learner 副本
>> operator add remove-peer 1 2                     // 移除 store 2 上的 Region 1 的一个副本
>> operator add transfer-leader 1 2                 // 把 Region 1 的 leader 调度到 store 2
>> operator add transfer-region 1 2 3 4             // 把 Region 1 调度到 store 2,3,4
>> operator add transfer-peer 1 2 3                 // 把 Region 1 在 store 2 上的副本调度到
    ↪ store 3
>> operator add merge-region 1 2                    // 将 Region 1 与 Region 2 合并
>> operator add split-region 1 --policy=approximate // 将 Region 1 对半拆分成两个 Region，
    ↪ 基于粗略估计值
>> operator add split-region 1 --policy=scan        // 将 Region 1 对半拆分成两个 Region，
    ↪ 基于精确扫描值
>> operator remove 1                                // 把 Region 1 的调度操作删掉
>> operator check 1                                 // 查看 Region 1 相关 operator 的状态
```

其中，Region 的分裂都是尽可能地从靠近中间的位置开始。对这个位置的选择支持两种策略，即 scan 和 approximate。它们之间的区别是，前者通过扫描这个 Region 的方式来确定中间的 key，而后者是通过查看 SST 文件中记录的统计信息，来得到近似的位置。一般来说，前者更加精确，而后者消耗更少的 I/O，可以更快地完成。

12.6.2.4.9 ping

用于显示ping PD 所需要花费的时间

示例：

```
>> ping
```

```
time: 43.12698ms
```

12.6.2.4.10 region <region_id> [--jq="<query string>"]

用于显示 Region 信息。使用 jq 格式化输出请参考jq-格式化 -json-输出示例。示例如下。

显示所有 Region 信息：

```
>> region
```

```
{
  "count": 1,
  "regions": [......]
}
```

显示 Region id 为 2 的信息：

```
>> region 2
```

```
{
  "id": 2,
  "start_key": "7480000000000000FF1D00000000000000F8",
  "end_key": "7480000000000000FF1F00000000000000F8",
  "epoch": {
    "conf_ver": 1,
    "version": 15
  },
  "peers": [
    {
      "id": 40,
      "store_id": 3
    }
  ],
  "leader": {
    "id": 40,
    "store_id": 3
  },
  "written_bytes": 0,
  "read_bytes": 0,
  "written_keys": 0,
  "read_keys": 0,
  "approximate_size": 1,
  "approximate_keys": 0
}
```

12.6.2.4.11 region key [--format=raw|encode|hex] <key>

用于查询某个 key 位于哪一个 Region 上，支持 raw、encoding 和 hex 格式。使用 encoding 格式时，key 需要使用单引号。

Hex 格式（默认）示例：

```
>> region key 7480000000000000FF1300000000000000F8
{
  "region": {
    "id": 2,
    ......
  }
}
```

Raw 格式示例：

```
>> region key --format=raw abc
```

```
{
  "region": {
    "id": 2,
    ......
  }
}
```

Encoding 格式示例：

```
>> region key --format=encode 't\200\000\000\000\000\000\000\377\035_r
    ↪ \200\000\000\000\000\377\017U\320\000\000\000\000\000\000\372'
```

```
{
  "region": {
    "id": 2,
    ......
  }
}
```

12.6.2.4.12 region scan

用于获取所有 Region。

示例：

```
>> region scan
```

```
{
  "count": 20,
```

```
  "regions": [......],
}
```

12.6.2.4.13 region sibling <region_id>

用于查询某个 Region 相邻的 Region。

示例：

```
>> region sibling 2
```

```
{
  "count": 2,
  "regions": [......],
}
```

12.6.2.4.14 region startkey [--format=raw|encode|hex] <key> <limit>

用于查询从某个 key 开始的所有 Region。

示例：

```
>> region startkey --format=raw abc
```

```
{
  "count": 16,
  "regions": [......],
}
```

12.6.2.4.15 region store <store_id>

用于查询某个 store 上面所有的 Region。

示例：

```
>> region store 2
```

```
{
  "count": 10,
  "regions": [......],
}
```

12.6.2.4.16 region topread [limit]

用于查询读流量最大的 Region。 limit 的默认值是 16。

示例：

```
>> region topread
```

```
{
  "count": 16,
  "regions": [......],
}
```

12.6.2.4.17 region topwrite [limit]

用于查询写流量最大的 Region。 limit 的默认值是 16。

示例：

```
>> region topwrite
```

```
{
  "count": 16,
  "regions": [......],
}
```

12.6.2.4.18 region topconfver [limit]

用于查询 conf version 最大的 Region。 limit 的默认值是 16。

示例：

```
>> region topconfver
```

```
{
  "count": 16,
  "regions": [......],
}
```

12.6.2.4.19 region topversion [limit]

用于查询 version 最大的 Region。 limit 的默认值是 16。

示例：

```
>> region topversion
```

```
{
  "count": 16,
  "regions": [......],
}
```

12.6.2.4.20 region topsize [limit]

用于查询 approximate size 最大的 Region。 limit 的默认值是 16。

示例：

```
>> region topsize
```

```
{
  "count": 16,
  "regions": [......],
}
```

12.6.2.4.21 region check [miss-peer | extra-peer | down-peer | pending-peer | offline-peer | empty-region | hist-size | hist-keys]

用于查询处于异常状态的 Region，各类型的意义如下

- miss-peer：缺副本的 Region
- extra-peer：多副本的 Region
- down-peer：有副本状态为 Down 的 Region
- pending-peer：有副本状态为 Pending 的 Region

示例：

```
>> region check miss-peer
```

```
{
  "count": 2,
  "regions": [......],
}
```

12.6.2.4.22 scheduler [show | add | remove | pause | resume | config]

用于显示和控制调度策略。

示例：

```
>> scheduler show                                  // 显示所有的 schedulers
>> scheduler add grant-leader-scheduler 1          // 把 store 1 上的所有 Region 的 leader 调度到
    ↪ store 1
>> scheduler add evict-leader-scheduler 1          // 把 store 1 上的所有 Region 的 leader 从
    ↪ store 1 调度出去
>> scheduler config evict-leader-scheduler         // v4.0.0 起，展示该调度器具体在哪些 store 上
>> scheduler add shuffle-leader-scheduler          // 随机交换不同 store 上的 leader
>> scheduler add shuffle-region-scheduler          // 随机调度不同 store 上的 Region
>> scheduler remove grant-leader-scheduler-1       // 把对应的调度器删掉，`-1` 对应 store ID
>> scheduler pause balance-region-scheduler 10     // 暂停运行 balance-region 调度器 10 秒
>> scheduler pause all 10                          // 暂停运行所有的调度器 10 秒
>> scheduler resume balance-region-scheduler       // 继续运行 balance-region 调度器
>> scheduler resume all                            // 继续运行所有的调度器
>> scheduler config balance-hot-region-scheduler   // 显示 balance-hot-region 调度器的配置
```

12.6.2.4.23 scheduler config balance-hot-region-scheduler

用于查看和控制 balance-hot-region-scheduler 策略。

示例：

```
>> scheduler config balance-hot-region-scheduler  // 显示 balance-hot-region 调度器的所有配置
{
  "min-hot-byte-rate": 100,
  "min-hot-key-rate": 10,
  "max-zombie-rounds": 3,
  "max-peer-number": 1000,
  "byte-rate-rank-step-ratio": 0.05,
  "key-rate-rank-step-ratio": 0.05,
  "count-rank-step-ratio": 0.01,
  "great-dec-ratio": 0.95,
  "minor-dec-ratio": 0.99,
  "src-tolerance-ratio": 1.02,
  "dst-tolerance-ratio": 1.02
}
```

- min-hot-byte-rate 指计数的最小字节，通常为 100。

```
>> scheduler config balance-hot-region-scheduler set min-hot-byte-rate 100
```

- min-hot-key-rate 指计数的最小 key，通常为 10。

```
>> scheduler config balance-hot-region-scheduler set min-hot-key-rate 10
```

- `max-zombie-rounds` 指一个 operator 可被纳入 pending influence 所允许的最大心跳次数。如果将它设置为更大的值，更多的 operator 可能会被纳入 pending influence。通常用户不需要修改这个值。pending influence 指的是在调度中产生的、但仍生效的影响。

```
>> scheduler config balance-hot-region-scheduler set max-zombie-rounds 3
```

- `max-peer-number` 指最多要被解决的 peer 数量。这个配置可避免调度器处理速度过慢。

```
>> scheduler config balance-hot-region-scheduler set max-peer-number 1000
```

- `byte-rate-rank-step-ratio`、`key-rate-rank-step-ratio` 和 `count-rank-step-ratio` 分别控制 byte、key、count 的 step ranks。rank step ratio 决定了计算 rank 时的 step 值。`great-dec-ratio` 和 `minor-dec-ratio` 控制 dec 的 rank。通常用户不需要修改这些配置项。

```
>> scheduler config balance-hot-region-scheduler set byte-rate-rank-step-ratio 0.05
```

- `src-tolerance-ratio` 和 `dst-tolerance-ratio` 是期望调度器的配置项。`tolerance-ratio` 的值越小，调度就越容易。当出现冗余调度时，你可以适当调大这个值。

```
>> scheduler config balance-hot-region-scheduler set src-tolerance-ratio 1.05
```

12.6.2.4.24 `store [delete | label | weight | remove-tombstone | limit ] <store_id> [--jq="<query string>"]`

用于显示 store 信息或者删除指定 store。使用 jq 格式化输出请参考jq-格式化 -json-输出示例。示例如下。

显示所有 store 信息：

```
>> store
```

```
{
  "count": 3,
  "stores": [...]
}
```

获取 store id 为 1 的 store：

```
>> store 1
```

```
......
```

下线 store id 为 1 的 store：

```
>> store delete 1
```

```
......
```

设置 store id 为 1 的 store 的键为 “zone” 的 label 的值为 “cn”：

```
>> store label 1 zone cn
```

设置 store id 为 1 的 store 的 leader weight 为 5，Region weight 为 10：

```
>> store weight 1 5 10
```

```
>> store remove-tombstone                // 删除所有 tombstone 状态的 store
```

store limit 命令的的使用方法可以参考Store Limit。

> 注意：
>
> 使用 pd-ctl 可以查看 TiKV 节点的状态信息，即 Up，Disconnect，Offline，Down，或 Tombstone。如需查看各个状态之间的关系，请参考TiKV Store 状态之间的关系。

12.6.2.4.25 log [fatal | error | warn | info | debug]

用于设置 PD leader 的日志级别。

```
>> log warn
```

12.6.2.4.26 tso

用于解析 TSO 到物理时间和逻辑时间。示例如下。

解析 TSO：

```
>> tso 395181938313123110
```

```
system:  2017-10-09 05:50:59 +0800 CST
logic:  120102
```

12.6.2.5 jq 格式化 json 输出示例

12.6.2.5.1 简化 store 的输出

```
>> store --jq=".stores[].store | { id, address, state_name}"
```

```
{"id":1,"address":"127.0.0.1:20161","state_name":"Up"}
{"id":30,"address":"127.0.0.1:20162","state_name":"Up"}
...
```

12.6.2.5.2 查询节点剩余空间

```
>> store --jq=".stores[] | {id: .store.id, available: .status.available}"
```

```
{"id":1,"available":"10 GiB"}
{"id":30,"available":"10 GiB"}
...
```

12.6.2.5.3 查询状态不为 Up 的所有节点

```
>> store --jq='.stores[].store | select(.state_name!="Up") | { id, address, state_name}'
```

```
{"id":1,"address":"127.0.0.1:20161""state_name":"Offline"}
{"id":5,"address":"127.0.0.1:20162""state_name":"Offline"}
...
```

12.6.2.5.4 查询所有的 TiFlash 节点

```
>> store --jq='.stores[].store | select(.labels | length>0 and contains([{"key":"engine","value":
    ↪ "tiflash"}])) | { id, address, state_name}'
```

```
{"id":1,"address":"127.0.0.1:20161""state_name":"Up"}
{"id":5,"address":"127.0.0.1:20162""state_name":"Up"}
...
```

12.6.2.5.5 查询 Region 副本的分布情况

```
>> region --jq=".regions[] | {id: .id, peer_stores: [.peers[].store_id]}"
```

```
{"id":2,"peer_stores":[1,30,31]}
{"id":4,"peer_stores":[1,31,34]}
...
```

12.6.2.5.6 根据副本数过滤 Region

例如副本数不为 3 的所有 Region：

```
>> region --jq=".regions[] | {id: .id, peer_stores: [.peers[].store_id] | select(length != 3)}"
```

```
{"id":12,"peer_stores":[30,32]}
{"id":2,"peer_stores":[1,30,31,32]}
```

12.6.2.5.7 根据副本 store ID 过滤 Region

例如在 store30 上有副本的所有 Region：

```
>> region --jq=".regions[] | {id: .id, peer_stores: [.peers[].store_id] | select(any(.==30))}"
```

```
{"id":6,"peer_stores":[1,30,31]}
{"id":22,"peer_stores":[1,30,32]}
...
```

还可以像这样找出在 store30 或 store31 上有副本的所有 Region：

```
>> region --jq=".regions[] | {id: .id, peer_stores: [.peers[].store_id] | select(any(.==(30,31)))
    ↪ }"
```

```
{"id":16,"peer_stores":[1,30,34]}
{"id":28,"peer_stores":[1,30,32]}
{"id":12,"peer_stores":[30,32]}
...
```

12.6.2.5.8 恢复数据时寻找相关 Region

例如当 [store1, store30, store31] 宕机时不可用时，我们可以通过查找所有 Down 副本数量大于正常副本数量的所有 Region：

```
>> region --jq=".regions[] | {id: .id, peer_stores: [.peers[].store_id] | select(length as $total
    ↪  | map(if .==(1,30,31) then . else empty end) | length>=$total-length) }"
```

```
{"id":2,"peer_stores":[1,30,31,32]}
{"id":12,"peer_stores":[30,32]}
{"id":14,"peer_stores":[1,30,32]}
...
```

或者在 [store1, store30, store31] 无法启动时，找出 store1 上可以安全手动移除数据的 Region。我们可以这样过滤出所有在 store1 上有副本并且没有其他 DownPeer 的 Region：

```
>> region --jq=".regions[] | {id: .id, peer_stores: [.peers[].store_id] | select(length>1 and any
    ↪ (.==1) and all(.!=(30,31)))}"
```

```
{"id":24,"peer_stores":[1,32,33]}
```

在 [store30, store31] 宕机时，找出能安全地通过创建 `remove-peer` Operator 进行处理的所有 Region，即有且仅有一个 DownPeer 的 Region：

```
>> region --jq=".regions[] | {id: .id, remove_peer: [.peers[].store_id] | select(length>1) | map(
    ↪ if .==(30,31) then . else empty end) | select(length==1)}"
```

```
{"id":12,"remove_peer":[30]}
{"id":4,"remove_peer":[31]}
{"id":22,"remove_peer":[30]}
...
```

12.6.3 TiDB Control 使用说明

TiDB Control 是 TiDB 的命令行工具，用于获取 TiDB 状态信息，多用于调试。本文介绍了 TiDB Control 的主要功能和各个功能的使用方法。

12.6.3.1 获取 TiDB Control

本节提供了两种方式获取 TiDB Control 工具。

> **注意：**
>
> 建议使用的 Control 工具版本与集群版本保持一致。

12.6.3.1.1 通过 TiUP 安装

在安装 TiUP 之后，可以使用 `tiup ctl:<cluster-version> tidb` 命令来获取 TiDB Control 的二进制程序以及运行 TiDB Control。

12.6.3.1.2 从源代码编译安装

编译环境要求： Go Version 1.13 以上

编译步骤： 在 TiDB Control 项目根目录，使用 `make` 命令进行编译，生成 tidb-ctl。

编译文档： 帮助文档在 doc 文件夹下，如丢失或需要更新，可通过 `make doc` 命令生成帮助文档。

12.6.3.2 使用介绍

`tidb-ctl` 的使用由命令（包括子命令）、选项和参数组成。命令即不带 - 或者 -- 的字符，选项即带有 - 或者 -- 的字符，参数即命令或选项字符后紧跟的传递给命令和选项的字符。

如： `tidb-ctl schema in mysql -n db`

- schema: 命令
- in: schema 的子命令
- mysql: in 的参数
- -n: 选项
- db: -n 的参数

目前，TiDB Control 包含以下子命令。

- `tidb-ctl base64decode` 用于 BASE64 解码
- `tidb-ctl decoder` 用于 KEY 解码
- `tidb-ctl etcd` 用于操作 etcd
- `tidb-ctl log` 用于格式化日志文件，将单行的堆栈信息展开
- `tidb-ctl mvcc` 用于获取 MVCC 信息
- `tidb-ctl region` 用于获取 Region 信息
- `tidb-ctl schema` 用于获取 Schema 信息
- `tidb-ctl table` 用于获取 Table 信息

12.6.3.2.1 获取帮助

`tidb-ctl -h/--help` 用于获取帮助信息。tidb-ctl 由多层命令组成，tidb-ctl 及其所有子命令都可以通过 `-h/--help` 来获取使用帮助。

以获取 Schema 信息为例：

通过 `tidb-ctl schema -h` 可以获取这个子命令的使用帮助。schema 有两个子命令 ——in 和 tid。in 用来通过数据库名获取数据库中所有表的表结构，tid 用来通过全数据库唯一的 table_id 获取表的表结构。

12.6.3.2.2 全局参数

`tidb-ctl` 有 4 个与连接相关的全局参数，分别为：

- `--host` TiDB 服务地址
- `--port` TiDB status 端口
- `--pdhost` PD 服务地址
- `--pdport` PD 服务端口
- `--ca` 连接使用的 TLS CA 文件路径
- `--ssl-key` 连接使用的 TLS 密钥文件路径
- `--ssl-cert` 连接使用的 TLS 证书文件路径

其中 `--pdhost` 和 `--pdport` 主要是用于 `etcd` 子命令，例如：`tidb-ctl etcd ddlinfo`。如不添加地址和端口将使用默认值，TiDB/PD 服务默认的地址是 127.0.0.1 (服务地址只能使用 IP 地址)，TiDB 服务端口默认的端口是 10080，PD 服务端口默认的端口是 2379 连接选项是全局选项，适用于以下所有命令。

12.6.3.2.3 schema 命令

in 子命令

in 子命令用来通过数据库名获取数据库中所有表的表结构。

`tidb-ctl schema in {数据库名}`

如：`tidb-ctl schema in mysql` 将得到以下结果：

```
[
    {
        "id": 13,
        "name": {
            "O": "columns_priv",
            "L": "columns_priv"
        },
              ...
        "update_timestamp": 399494726837600268,
        "ShardRowIDBits": 0,
        "Partition": null
    }
]
```

结果将以 json 形式展示，内容较长，这里做了截断。

如希望指定表名，可以使用 `tidb-ctl schema in {数据库名} -n {表名}` 进行过滤。

如：`tidb-ctl schema in mysql -n db` 将得到 mysql 库中 db 表的表结构，结果如下：

```
{
    "id": 9,
    "name": {
        "O": "db",
        "L": "db"
    },
    ...
    "Partition": null
}
```

这里同样做了截断。

如使用的 TiDB 地址不为默认地址和端口，可以使用命令行参数 `--host`, `--port` 选项，如：`tidb-ctl --host 172.16.55.88 --port 8898 schema in mysql -n db`。

tid 子命令

tid 子命令用来通过表的 id 获取数据库中表的表结构。

通过使用 in 子命令查询到数据库中表的 id，之后可以通过 tid 子命令查看表的详细信息。

例如，查询到 `mysql.stat_meta` 表的 id 是 21，可以通过 `tidb-ctl schema tid -i 21` 查看表的详细信息。

```
{
 "id": 21,
 "name": {
  "O": "stats_meta",
  "L": "stats_meta"
 },
 "charset": "utf8mb4",
```

```
 "collate": "utf8mb4_bin",
  ...
}
```

同 in 子命令一样，如果使用的 TiDB 地址不是默认的地址和端口，需要通过 --host 和 --port 参数指定 TiDB 的地址和 status 端口。

12.6.3.2.4 base64decode 命令

base64decode 用来解码 base64 数据。

基本用法：

```
tidb-ctl base64decode [base64_data]
tidb-ctl base64decode [db_name.table_name] [base64_data]
tidb-ctl base64decode [table_id] [base64_data]
```

1. 准备环境，执行以下 SQL

```
use test;
create table t (a int, b varchar(20),c datetime default current_timestamp , d timestamp
    ↪ default current_timestamp, unique index(a));
insert into t (a,b,c) values(1,"哈哈 hello",NULL);
alter table t add column e varchar(20);
```

2. 用 HTTP API 接口获取 MVCC 数据

```
curl "http://$IP:10080/mvcc/index/test/t/a/1?a=1"
```

```
{
 "info": {
  "writes": [
   {
    "start_ts": 407306449994645510,
    "commit_ts": 407306449994645513,
    "short_value": "AAAAAAAAAAE="    # unique index a 存的值是对应行的 handle id.
   }
  ]
 }
}%
```

```
curl "http://$IP:10080/mvcc/key/test/t/1"
```

```
{
 "info": {
  "writes": [
```

```
   {
    "start_ts": 407306588892692486,
    "commit_ts": 407306588892692489,
    "short_value": "CAIIAggEAhjlk4jlk4ggaGVsbG8IBgAICAmAgIDwjYuuORk="  # handle id 为 1
        ↪ 的行数据。
   }
  ]
 }
}%
```

3. 用 base64decode 解码 handle id (uint64).

```
tidb-ctl base64decode AAAAAAAAAAE=
```

```
hex: 0000000000000001
uint64: 1
```

4. 用 base64decode 解码行数据。

```
./tidb-ctl base64decode test.t CAIIAggEAhjlk4jlk4ggaGVsbG8IBgAICAmAgIDwjYuuORk=
```

```
a:      1
b:      哈哈 hello
c is NULL
d:      2019-03-28 05:35:30
e not found in data
```

如果 test.t 的 table id 是 60，你也可以使用下列命令获得同样结果：

```
./tidb-ctl base64decode 60 CAIIAggEAhjlk4jlk4ggaGVsbG8IBgAICAmAgIDwjYuuORk=
```

```
a:      1
b:      哈哈 hello
c is NULL
d:      2019-03-28 05:35:30
e not found in data
```

12.6.3.2.5 decoder 命令

- 以下示例解码 row key，index key 类似。

```
./tidb-ctl decoder "t\x00\x00\x00\x00\x00\x00\x00\x1c_r\x00\x00\x00\x00\x00\x00\x00\xfa"
```

```
format: table_row
table_id: -9223372036854775780
row_id: -9223372036854775558
```

- 以下示例解码 value

```
./tidb-ctl decoder AhZoZWxsbyB3b3JsZAiAEA==
```

```
format: index_value
index_value[0]: {type: bytes, value: hello world}
index_value[1]: {type: bigint, value: 1024}
```

12.6.3.2.6 etcd 命令

- `tidb-ctl etcd ddlinfo` 获取 DDL 信息。
- `tidb-ctl etcd putkey KEY VALUE` 添加 KEY VALUE 到 etcd (所有的 KEY 会添加到 `/tidb/ddl/all_schema_versions/` 之下)。

```
tidb-ctl etcd putkey "foo" "bar"
```

实际是添加 KEY 为 `/tidb/ddl/all_schema_versions/foo`，VALUE 为 bar 的键值对到 etcd 中。

- `tidb-ctl etcd delkey` 删除 etcd 中的 KEY,只有前缀以 `/tidb/ddl/fg/owner/` 和 `/tidb/ddl/all_schema_versions/` 开头才允许被删除。

```
tidb-ctl etcd delkey "/tidb/ddl/fg/owner/foo" &&
tidb-ctl etcd delkey "/tidb/ddl/all_schema_versions/bar"
```

12.6.3.2.7 log 命令

TiDB 错误日志的堆栈信息是一行的格式，可以使用 `tidb-ctl log` 将堆栈信息格式化成多行形式。

12.6.3.2.8 keyrange 命令

keyrange 子命令用于查询全局或表相关的关键 key range 信息，以十六进制形式输出。

- 使用 `tidb-ctl keyrange` 命令查看全局的关键 key range。

```
tidb-ctl keyrange
```

```
global ranges:
  meta: (6d, 6e)
  table: (74, 75)
```

- 添加 `--encode` 选项可以显示 encode 过的 key (与 TiKV 及 PD 中的格式相同)。

```
tidb-ctl keyrange --encode
```

```
global ranges:
  meta: (6d00000000000000f8, 6e00000000000000f8)
  table: (7400000000000000f8, 7500000000000000f8)
```

- 使用 `tidb-ctl keyrange --database={db} --table={tbl}` 命令查看全局和表相关的关键 key range。

```
tidb-ctl keyrange --database test --table ttt
```

```
global ranges:
  meta: (6d, 6e)
  table: (74, 75)
table ttt ranges: (NOTE: key range might be changed after DDL)
  table: (74800000000000002f, 748000000000000030)
  table indexes: (748000000000000002f5f69, 748000000000000002f5f72)
    index c2: (748000000000000002f5f698000000000000001, 74800000000000002
        ↪ f5f698000000000000002)
    index c3: (748000000000000002f5f698000000000000002, 74800000000000002
        ↪ f5f698000000000000003)
    index c4: (748000000000000002f5f698000000000000003, 74800000000000002
        ↪ f5f698000000000000004)
  table rows: (748000000000000002f5f72, 748000000000000030)
```

12.6.4 PD Recover 使用文档

PD Recover 是对 PD 进行灾难性恢复的工具，用于恢复无法正常启动或服务的 PD 集群。

12.6.4.1 安装 PD Recover

要使用 PD Recover，你可以从源代码编译，也可以直接下载 TiDB 安装包。

12.6.4.1.1 从源代码编译

- Go：PD Recover 使用了 Go 模块，请安装 Go v1.16 或更新版本。
- 在 PD 根目录下，运行 `make pd-recover` 命令来编译源代码并生成 `bin/pd-recover`。

> **注意：**
>
> 一般来说，用户不需要编译源代码，因为发布的二进制文件或 Docker 中已包含 PD Recover 工具。开发者可以参考以上步骤来编译源代码。

12.6.4.1.2 下载 TiDB 安装包

PD Recover 包含在 TiDB 安装包中。直接下载 TiDB 安装包即可获取最新版本的 PD Recover。

安装包	操作系统	架构	SHA256 校验和
https ↪ ://download ↪ .pingcap ↪ .org ↪ /tidb ↪ -{version}- ↪ linux ↪ - ↪ amd64 ↪ .tar ↪ .gz (pd-recover)	Linux	amd64	https ↪ ://download ↪ .pingcap ↪ .org ↪ /tidb ↪ -{version}- ↪ linux ↪ - ↪ amd64 ↪ .sha256 ↪

注意：

{version} 是 TiDB 的版本号。例如，v4.0.16 的安装包下载链接为 https://download.pingcap.org/tidb-v4.0.16-linux-amd64.tar.gz。你也可以用 latest 替换 {version}，下载最新的未发布版本。

12.6.4.2 快速开始

12.6.4.2.1 获取 Cluster ID

一般在 PD、 TiKV 或 TiDB 的日志中都可以获取 Cluster ID。你可以直接在服务器上查看日志以获取 Cluster ID。

从 PD 日志获取 Cluster ID（推荐）

使用以下命令，从 PD 日志中获取 Cluster ID：

```
cat {{/path/to}}/pd.log | grep "init cluster id"
```

```
[2019/10/14 10:35:38.880 +00:00] [INFO] [server.go:212] ["init cluster id"] [cluster-id
    ↪ =6747551640615446306]
...
```

或者也可以从 TiDB 或 TiKV 的日志中获取。

从 TiDB 日志获取 Cluster ID

使用以下命令，从 TiDB 日志中获取 Cluster ID：

```
cat {{/path/to}}/tidb.log | grep "init cluster id"
```

```
2019/10/14 19:23:04.688 client.go:161: [info] [pd] init cluster id 6747551640615446306
...
```

从 TiKV 日志获取 Cluster ID

使用以下命令，从 TiKV 日志中获取 Cluster ID：

```
cat {{/path/to}}/tikv.log | grep "connect to PD cluster"
```

```
[2019/10/14 07:06:35.278 +00:00] [INFO] [tikv-server.rs:464] ["connect to PD cluster
    ↪ 6747551640615446306"]
...
```

12.6.4.2.2 获取已分配 ID

在指定已分配 ID 时，需指定一个比当前最大的已分配 ID 更大的值。可以从监控中获取已分配 ID，也可以直接在服务器上查看日志。

从监控中获取已分配 ID（推荐）

要从监控中获取已分配的 ID，需要确保你所查看的监控指标是上一任 PD Leader 的指标。可从 PD Dashboard 中 Current ID allocation 面板获取最大的已分配 ID。

从 PD 日志获取已分配 ID

要从 PD 日志中获取分配的 ID，需要确保你所查看的日志是上一任 PD Leader 的日志。运行以下命令获取最大的已分配 ID：

```
cat {{/path/to}}/pd*.log | grep "idAllocator allocates a new id" |  awk -F'=' '{print $2}' | awk
    ↪ -F']' '{print $1}' | sort -r -n | head -n 1
```

```
4000
...
```

你也可以在所有 PD server 中运行上述命令，找到最大的值。

12.6.4.2.3 部署一套新的 PD 集群

部署新的 PD 集群之前，需要停止当前的 PD 集群，然后删除旧的数据目录（用 `--data-dir` 指定）。

12.6.4.2.4 使用 pd-recover

```
./pd-recover -endpoints http://10.0.1.13:2379 -cluster-id 6747551640615446306 -alloc-id 10000
```

12.6.4.2.5 重启整个集群

当出现 `recovery is successful` 的提示信息时，重启整个集群。

12.6.4.3 常见问题

12.6.4.3.1 获取 Cluster ID 时发现有多个 Cluster ID

新建 PD 集群时，会生成新的 Cluster ID。可以通过日志判断旧集群的 Cluster ID。

12.6.4.3.2 执行 pd-recover 时返回错误 `dial tcp 10.0.1.13:2379: connect: connection refused`

执行 pd-recover 时需要 PD 提供服务，请先部署并启动 PD 集群。

12.7 命令行参数

12.7.1 TiDB 配置参数

在启动 TiDB 时，你可以使用命令行参数或环境变量来配置 TiDB。本文将详细介绍 TiDB 的命令行启动参数。TiDB 的默认端口为 4000（客户端请求）与 10080（状态报告）。

12.7.1.1 `--advertise-address`

- 登录 TiDB 的 IP 地址
- 默认：“”
- 必须确保用户和集群中的其他机器都能够访问到该 IP 地址

12.7.1.2 `--config`

- 配置文件
- 默认：“”
- 如果你指定了配置文件，TiDB 会首先读取配置文件的配置。如果对应的配置在命令行参数里面也存在，TiDB 就会使用命令行参数的配置来覆盖配置文件中的配置。详细的配置项请参阅TiDB 配置文件描述。

12.7.1.3 `--config-check`

- 检查配置文件的有效性并退出
- 默认：false

12.7.1.4 --config-strict

- 增强配置文件的有效性
- 默认：false

12.7.1.5 --cors

- 用于设置 TiDB HTTP 状态服务的 Access-Control-Allow-Origin
- 默认：“”

12.7.1.6 --host

- TiDB 服务监听的 host
- 默认：“0.0.0.0”
- 0.0.0.0 默认会监听所有的网卡地址。如果有多块网卡，可以指定对外提供服务的网卡，如 192.168.100.113

12.7.1.7 --enable-binlog

- 是否产生 TiDB Binlog
- 默认：false

12.7.1.8 -L

- Log 级别
- 默认：“info”
- 可选项为：debug、 info、 warn、 error、 fatal

12.7.1.9 --lease

- Schema lease 的持续时间。除非你知道更改该值带来的后果，否则你的更改操作是危险的。
- 默认：45s

12.7.1.10 --log-file

- Log 文件
- 默认：“”
- 如果未设置该参数，log 会默认输出到“stderr”；如果设置了该参数，log 会输出到对应的文件中。每天凌晨，log 会自动轮转使用一个新的文件，并且将以前的文件改名备份

12.7.1.11 --log-slow-query

- 慢查询日志文件路径
- 默认：“”
- 如果未设置该参数，log 会默认输出到 --log-file 指定的文件中

12.7.1.12 --metrics-addr

- Prometheus Pushgateway 地址
- 默认：“”
- 如果该参数为空，TiDB 不会将统计信息推送给 Pushgateway。参数格式示例：`--metrics-addr =192.168.100.115:9091`

12.7.1.13 --metrics-interval

- 推送统计信息到 Prometheus Pushgateway 的时间间隔
- 默认：15s
- 设置为 0 表示不推送统计信息给 Pushgateway。示例：`--metrics-interval=2` 指每两秒推送到 Pushgateway

12.7.1.14 -P

- TiDB 服务监听端口
- 默认：“4000”
- TiDB 服务会使用该端口接受 MySQL 客户端发来的请求

12.7.1.15 --path

- 对于本地存储引擎“mocktikv”来说，path 指定的是实际的数据存放路径
- 当 `--store = tikv` 时，必须指定 path；当 `--store = mocktikv` 时，如果不指定 path，会使用默认值。
- 对于“TiKV”存储引擎来说，path 指定的是实际的 PD 地址。假如在 192.168.100.113:2379、192.168.100.114:2379 和 192.168.100.115:2379 上面部署了 PD，那么 path 为“192.168.100.113:2379, 192.168.100.114:2379, 192.168.100.115:2379”
- 默认：“/tmp/tidb”
- 可以通过 `tidb-server --store=mocktikv --path=""` 来启动一个纯内存引擎的 TiDB

12.7.1.16 --proxy-protocol-networks

- 允许使用 PROXY 协议连接 TiDB 的代理服务器地址列表。
- 默认：“”
- 通常情况下，通过反向代理使用 TiDB 时，TiDB 会将反向代理服务器的 IP 地址视为客户端 IP 地址。对于支持 PROXY 协议的反向代理（如 HAProxy），开启 PROXY 协议后能让反向代理透传客户端真实的 IP 地址给 TiDB。
- 配置该参数后，TiDB 将允许配置的源 IP 地址使用 PROXY 协议连接到 TiDB，且拒绝这些源 IP 地址使用非 PROXY 协议连接。若该参数为空，则任何源 IP 地址都不能使用 PROXY 协议连接到 TiDB。地址可以使用 IP 地址格式 (192.168.1.50) 或者 CIDR 格式 (192.168.1.0/24)，并可用 , 分隔多个地址，或用 * 代表所有 IP 地址。

> **警告：**
>
> 需谨慎使用 * 符号，因为它可能引入安全风险，允许来自任何 IP 的客户端自行汇报其 IP 地址。另外，它可能会导致部分直接连接 TiDB 的内部组件无法使用，例如 TiDB Dashboard。

12.7.1.17 --proxy-protocol-header-timeout

- PROXY Protocol 请求头读取超时时间
- 默认：5
- 单位：秒

> **注意：**
>
> 不要将该参数配置为 0。除非特殊情况，一般使用默认值即可。

12.7.1.18 --report-status

- 用于打开或者关闭服务状态监听端口
- 默认：true
- 将参数值设置为 `true` 表明开启状态监听端口；设置为 `false` 表明关闭状态监听端口

12.7.1.19 --run-ddl

- tidb-server 是否运行 DDL 语句，集群内至少需要有一台 tidb-server 设置该参数
- 默认：true
- 值可以为 `true` 或者 `false`。设置为 `true` 表明自身会运行 DDL；设置为 `false` 表明自身不会运行 DDL

12.7.1.20 --socket string

- TiDB 服务使用 unix socket file 方式接受外部连接
- 默认：“”
- 例如可以使用 “/tmp/tidb.sock” 来打开 unix socket file

12.7.1.21 --status

- TiDB 服务状态监听端口
- 默认：“10080”
- 该端口用于展示 TiDB 内部数据，包括 prometheus 统计和 pprof
- Prometheus 统计可以通过 `http://host:status_port/metrics` 访问
- pprof 数据可以通过 `http://host:status_port/debug/pprof` 访问

12.7.1.22 --status-host

- TiDB 服务状态监听 host
- 默认：“0.0.0.0”

12.7.1.23 --store

- 用来指定 TiDB 底层使用的存储引擎
- 默认：“mocktikv”
- 可以选择“mocktikv”（本地存储引擎）或者“tikv”（分布式存储引擎）

12.7.1.24 --token-limit

- TiDB 中同时允许运行的 Session 数量，用于流量控制
- 默认：1000
- 如果当前运行的连接多于该 token-limit，那么请求会阻塞，等待已经完成的操作释放 Token

12.7.1.25 -V

- 输出 TiDB 的版本
- 默认：“”

12.7.1.26 --plugin-dir

- plugin 存放目录
- 默认：“/data/deploy/plugin”

12.7.1.27 --plugin-load

- 需要加载的 plugin 名称，多个 plugin 以“,”逗号分隔
- 默认：“”

12.7.1.28 --affinity-cpus

- 设置 TiDB server CPU 亲和性，以“,”逗号分隔，例如“1,2,3”
- 默认：“”

12.7.1.29 --repair-mode

- 是否开启修复模式，仅用于数据修复场景
- 默认：false

12.7.1.30 --repair-list

- 修复模式下需要修复的表名
- 默认：“”

12.7.1.31 --require-secure-transport

- 是否要求客户端使用安全传输模式
- 默认：false

12.7.2 TiKV 配置参数

TiKV 的命令行参数支持一些可读性好的单位转换。

- 文件大小（以 bytes 为单位）：KB, MB, GB, TB, PB（也可以全小写）
- 时间（以毫秒为单位）：ms, s, m, h

12.7.2.1 -A, --addr

- TiKV 监听地址
- 默认：“127.0.0.1:20160”
- 如果部署一个集群，--addr 必须指定当前主机的 IP 地址，例如“192.168.100.113:20160”，如果是运行在 docker 则需要指定为“0.0.0.0:20160”

12.7.2.2 --advertise-addr

- TiKV 对外访问地址。
- 默认：${addr}
- 在某些情况下，比如 Docker 或者 NAT 网络环境，客户端并不能通过 --addr 的地址来访问到 TiKV。这时候，你可以设置 --advertise-addr 来让客户端访问 TiKV。
- 例如，docker 内部 IP 地址为 172.17.0.1，而宿主机的 IP 地址为 192.168.100.113 并且设置了端口映射 -p 20160:20160，那么可以设置为 --advertise-addr=“192.168.100.113:20160”，客户端可以通过 192.168.100.113:20160 来找到这个服务

12.7.2.3 --status-addr

- TiKV 服务状态监听端口
- 默认：“20180”
- Prometheus 统计可以通过 http://host:status_port/metrics 访问
- Profile 数据可以通过 http://host:status_port/debug/pprof/profile 访问

12.7.2.4 --advertise-status-addr

- TiKV 对外访问服务状态地址
- 默认：使用 --status-addr
- 在某些情况下，例如 docker 或者 NAT 网络环境，客户端并不能通过 --status-addr 的地址来访问到 TiKV。此时，你可以设置 --advertise-status-addr 来让客户端访问 TiKV。
- 例如，Docker 内部 IP 地址为 172.17.0.1，而宿主机的 IP 地址为 192.168.100.113 并且设置了端口映射 -p 20180:20180，那么可以设置 --advertise-status-addr="192.168.100.113:20180"，客户端可以通过 192.168.100.113:20180 来找到这个服务。

12.7.2.5 -C, --config

- 配置文件
- 默认：“”
- 如果你指定了配置文件，TiKV 会首先读取配置文件的配置。然后如果对应的配置在命令行参数里面也存在，TiKV 就会使用命令行参数的配置来覆盖配置文件里面的

12.7.2.6 --capacity

- TiKV 存储数据的容量
- 默认：0（无限）
- PD 需要使用这个值来对整个集群做 balance 操作。（提示：你可以使用 10GB 来替代 10737418240，从而简化参数的传递）

12.7.2.7 --data-dir

- TiKV 数据存储路径
- 默认：“/tmp/tikv/store”

12.7.2.8 -L

- Log 级别
- 默认：“info”
- 我们能选择 trace, debug, info, warn, error, 或者 off

12.7.2.9 --log-file

- Log 文件
- 默认：“”
- 如果没设置这个参数，log 会默认输出到“stderr”，如果设置了，log 就会输出到对应的文件里面，在每天凌晨，log 会自动轮转使用一个新的文件，并且将以前的文件改名备份

12.7.2.10 --pd

- PD 地址列表。
- 默认：“”
- TiKV 必须使用这个值连接 PD，才能正常工作。使用逗号来分隔多个 PD 地址，例如：192.168.100.113:2379, 192.168.100.114:2379, 192.168.100.115:2379

12.7.3 TiFlash 命令行参数

本文介绍了 TiFlash 的命令行启动参数。

12.7.3.1 server --config-file

- 指定 TiFlash 的配置文件路径
- 默认：“”
- 必须指定配置文件，详细的配置项请参阅TiFlash 配置参数

12.7.4 PD 配置参数

PD 可以通过命令行参数或环境变量配置。

12.7.4.1 --advertise-client-urls

- 用于外部访问 PD 的 URL 列表。
- 默认：${client-urls}
- 在某些情况下，例如 Docker 或者 NAT 网络环境，客户端并不能通过 PD 自己监听的 client URLs 来访问到 PD，这时候，你就可以设置 advertise URLs 来让客户端访问。
- 例如，Docker 内部 IP 地址为 172.17.0.1，而宿主机的 IP 地址为 192.168.100.113 并且设置了端口映射 -p 2379:2379，那么可以设置为 --advertise-client-urls="http://192.168.100.113:2379"，客户端可以通过 http://192.168.100.113:2379 来找到这个服务。

12.7.4.2 --advertise-peer-urls

- 用于其他 PD 节点访问某个 PD 节点的 URL 列表。
- 默认：${peer-urls}
- 在某些情况下，例如 Docker 或者 NAT 网络环境，其他节点并不能通过 PD 自己监听的 peer URLs 来访问到 PD，这时候，你就可以设置 advertise URLs 来让其他节点访问。
- 例如，Docker 内部 IP 地址为 172.17.0.1，而宿主机的 IP 地址为 192.168.100.113 并且设置了端口映射 -p 2380:2380，那么可以设置为 advertise-peer-urls="http://192.168.100.113:2380"，其他 PD 节点可以通过 http://192.168.100.113:2380 来找到这个服务。

12.7.4.3 --client-urls

- PD 监听的客户端 URL 列表。
- 默认：http://127.0.0.1:2379
- 如果部署一个集群，--client-urls 必须指定当前主机的 IP 地址，例如 http://192.168.100.113:2379"，如果是运行在 Docker 则需要指定为 http://0.0.0.0:2379。

12.7.4.4 --peer-urls

- PD 节点监听其他 PD 节点的 URL 列表。
- 默认：http://127.0.0.1:2380
- 如果部署一个集群，--peer-urls 必须指定当前主机的 IP 地址，例如 http://192.168.100.113:2380，如果是运行在 Docker 则需要指定为 http://0.0.0.0:2380。

12.7.4.5 --config

- 配置文件。
- 默认：“”
- 如果你指定了配置文件，PD 会首先读取配置文件的配置。然后如果对应的配置在命令行参数里面也存在，PD 就会使用命令行参数的配置来覆盖配置文件里面的。

12.7.4.6 --data-dir

- PD 存储数据路径。
- 默认：default.${name}

12.7.4.7 --initial-cluster

- 初始化 PD 集群配置。
- 默认："{name}=http://{advertise-peer-url}"
- 例如，如果 name 是“pd”, 并且 advertise-peer-urls 是 http://192.168.100.113:2380，那么 initial-cluster 就是 pd=http://192.168.100.113:2380。
- 如果你需要启动三台 PD，那么 initial-cluster 可能就是 pd1=http://192.168.100.113:2380, pd2=http://192.168.100.114:2380, pd3=192.168.100.115:2380。

12.7.4.8 --join

- 动态加入 PD 集群。
- 默认：“”
- 如果你想动态将一台 PD 加入集群，你可以使用 --join="${advertise-client-urls}"，advertise-client-url 是当前集群里面任意 PD 的 advertise-client-url，你也可以使用多个 PD 的，需要用逗号分隔。

12.7.4.9 -L

- Log 级别。
- 默认：“info”
- 我们能选择 debug, info, warn, error 或者 fatal。

12.7.4.10 --log-file

- Log 文件。
- 默认：“”
- 如果没设置这个参数，log 会默认输出到“stderr”，如果设置了，log 就会输出到对应的文件里面，在每天凌晨，log 会自动轮转使用一个新的文件，并且将以前的文件改名备份。

12.7.4.11 --log-rotate

- 是否开启日志切割。
- 默认：true
- 当值为 true 时，按照 PD 配置文件中 [log.file] 信息执行。

12.7.4.12 --name

- 当前 PD 的名字。
- 默认：“pd”
- 如果你需要启动多个 PD，一定要给 PD 使用不同的名字

12.7.4.13 --cacert

- CA 文件路径，用于开启 TLS。
- 默认：“”

12.7.4.14 --cert

- 包含 X509 证书的 PEM 文件路径，用户开启 TLS。
- 默认：“”

12.7.4.15 --key

- 包含 X509 key 的 PEM 文件路径，用于开启 TLS。
- 默认：“”

12.7.4.16 --metrics-addr

- 指定 Prometheus Pushgateway 的地址。
- 默认：“”
- 如果留空，则不开启 Prometheus Push。

12.7.4.17 --force-new-cluster

- 强制使用当前节点创建新的集群。
- 默认：false
- 仅用于在 PD 丢失多数副本的情况下恢复服务，可能会产生部分数据丢失。

12.7.4.18 -V, --version

- 输出版本信息并退出。

12.8 监控指标

12.8.1 Overview 面板重要监控指标详解

使用 TiDB Ansible 或 TiUP 部署 TiDB 集群时，一键部署监控系统 (Prometheus & Grafana)，监控架构参见TiDB 监控框架概述。

目前 Grafana Dashboard 整体分为 PD、 TiDB、 TiKV、 Node_exporter、 Overview 等。

对于日常运维，我们单独挑选出重要的 Metrics 放在 Overview 页面，方便日常运维人员观察集群组件 (PD, TiDB, TiKV) 使用状态以及集群使用状态。

以下为 Overview Dashboard 监控说明：

12.8.1.1 Services Port Status

- Services Up：各服务在线节点数量

12.8.1.2 PD

- PD role：当前 PD 的角色
- Storage capacity：TiDB 集群总可用数据库空间大小
- Current storage size：TiDB 集群目前已用数据库空间大小，TiKV 多副本的空间占用也会包含在内
- Normal stores：处于正常状态的节点数目
- Abnormal stores：处于异常状态的节点数目，正常情况应当为 0
- Number of Regions：当前集群的 Region 总量，请注意 Region 数量与副本数无关
- 99% completed_cmds_duration_seconds：单位时间内，99% 的 pd-server 请求执行时间小于监控曲线的值，一般 <= 5ms
- Handle_requests_duration_seconds：PD 发送请求的网络耗时
- Region health：每个 Region 的状态，通常情况下，pending 的 peer 应该少于 100，miss 的 peer 不能一直大于 0
- Hot write Region’s leader distribution：每个 TiKV 实例上是写入热点的 leader 的数量
- Hot read Region’s leader distribution：每个 TiKV 实例上是读取热点的 leader 的数量
- Region heartbeat report：TiKV 向 PD 发送的心跳个数
- 99% Region heartbeat latency：99% 的情况下，心跳的延迟

12.8.1.3 TiDB

- Statement OPS：不同类型 SQL 语句每秒执行的数量。按 SELECT、 INSERT、 UPDATE 等来统计
- Duration：执行的时间
  - 客户端网络请求发送到 TiDB，到 TiDB 执行结束后返回给客户端的时间。一般情况下，客户端请求都是以 SQL 语句的形式发送，但也可以包含 COM_PING、COM_SLEEP、COM_STMT_FETCH、COM_SEND_LONG_DATA 之类的命令执行的时间
  - 由于 TiDB 支持 Multi-Query，因此，可以接受客户端一次性发送的多条 SQL 语句，如：`select 1; select 1; select 1;`。此时，统计的执行时间是所有 SQL 执行完之后的总时间
- CPS By Instance：每个 TiDB 实例上的命令统计。按照命令和执行结果成功或失败来统计

- Failed Query OPM：每个 TiDB 实例上，每秒钟执行 SQL 语句发生错误按照错误类型的统计（例如语法错误、主键冲突等）。包含了错误所属的模块和错误码
- Connection count：每个 TiDB 的连接数
- Memory Usage：每个 TiDB 实例的内存使用统计，分为进程占用内存和 Golang 在堆上申请的内存
- Transaction OPS：每秒事务执行数量统计
- Transaction Duration：事务执行的时间
- KV Cmd OPS：KV 命令执行数量统计
- KV Cmd Duration 99：KV 命令执行的时间
- PD TSO OPS：TiDB 每秒从 PD 获取 TSO 的数量
- PD TSO Wait Duration：TiDB 等待从 PD 获取 TS 的时间
- TiClient Region Error OPS：TiKV 返回 Region 相关错误信息的数量
- Lock Resolve OPS：TiDB 清理锁操作的数量。当 TiDB 的读写请求遇到锁时，会尝试进行锁清理
- Load Schema Duration：TiDB 从 TiKV 获取 Schema 的时间
- KV Backoff OPS：TiKV 返回错误信息的数量

12.8.1.4 TiKV

- leader：各个 TiKV 节点上 Leader 的数量分布
- region：各个 TiKV 节点上 Region 的数量分布
- CPU：各个 TiKV 节点的 CPU 使用率
- Memory：各个 TiKV 节点的内存使用量
- store size：每个 TiKV 实例的使用的存储空间的大小
- cf size：每个列族的大小
- channel full：每个 TiKV 实例上 channel full 错误的数量，正常情况下应当为 0
- server report failures：每个 TiKV 实例上报错的消息个数，正常情况下应当为 0
- scheduler pending commands：每个 TiKV 实例上 pending 命令的个数
- coprocessor executor count：TiKV 每秒收到的 coprocessor 操作数量，按照 coprocessor 类型统计
- coprocessor request duration：处理 coprocessor 读请求所花费的时间
- raft store CPU：raftstore 线程的 CPU 使用率，线程数量默认为 2 (通过 `raftstore.store-pool-size` 配置)。如果单个线程使用率超过 80%，说明使用率很高
- Coprocessor CPU：coprocessor 线程的 CPU 使用率

12.8.1.5 System Info

- Vcores：CPU 核心数量
- Memory：内存总大小
- CPU Usage：CPU 使用率，最大为 100%
- Load [1m]：1 分钟的负载情况
- Memory Available：剩余内存大小
- Network Traffic：网卡流量统计
- TCP Retrans：TCP 重传数量统计
- IO Util：磁盘使用率，最高为 100%，一般到 80% - 90% 就需要考虑加节点

12.8.1.6 图例

图 207: overview

12.8.2 TiDB 重要监控指标详解

使用 TiDB Ansible 或 TiUP 部署 TiDB 集群时，一键部署监控系统 (Prometheus & Grafana)，监控架构参见TiDB 监控框架概述。

目前 Grafana Dashboard 整体分为 PD、TiDB、TiKV、Node_exporter、Overview 等。TiDB 分为 TiDB 和 TiDB Summary 面板，两个面板的区别如下：

- TiDB 面板：提供尽可能全面的信息，供排查集群异常。
- TiDB Summary 面板：将 TiDB 面板中用户最为关心的部分抽取出来，并做了些许修改。主要用于提供数据库日常运行中用户关心的数据，如 QPS、TPS、响应延迟等，以便作为外部展示、汇报用的监控信息。

以下为 TiDB Dashboard 部分监控说明：

12.8.2.1 说明

- Query Summary

  - Duration：执行时间
    * 客户端网络请求发送到 TiDB，到 TiDB 执行结束后返回给客户端的时间。一般情况下，客户端请求都是以 SQL 语句的形式发送，但也可以包含 COM_PING、COM_SLEEP、COM_STMT_FETCH、COM_SEND_LONG_DATA 之类的命令执行时间
    * 由于 TiDB 支持 Multi-Query，因此，可以接受客户端一次性发送多条 SQL 语句，如 `select 1; select 1; select 1;`。此时，统计的执行时间是所有 SQL 语句执行完之后的总时间
  - Command Per Second：TiDB 每秒处理的命令数。按照执行结果成功或失败来统计
  - QPS：所有 TiDB 实例上的每秒执行的 SQL 语句数量。按 SELECT、INSERT、UPDATE 类型进行了区分
  - CPS By Instance：每个 TiDB 实例上的命令统计。按照命令和执行结果成功或失败来统计
  - Failed Query OPM：每个 TiDB 实例上，对每秒钟执行 SQL 语句发生的错误按照错误类型的统计（例如语法错误、主键冲突等）。包含了错误所属的模块和错误码
  - Slow query：慢查询处理时间统计（整个慢查询耗时、Coprocessor 耗时、Coprocessor 调度等待时间），慢查询分为 internal 和 general SQL 语句
  - Connection Idle Duration：空闲连接的持续时间
  - 999/99/95/80 Duration：不同类型的 SQL 语句执行耗时统计（不同百分位）

- Query Detail
  - Duration 80/95/99/999 By Instance：每个 TiDB 实例执行 SQL 语句的耗时统计（不同百分位）。
  - Failed Query OPM Detail：每个 TiDB 实例执行 SQL 语句发生的错误按照错误类型统计（例如语法错误、主键冲突等）。
  - Internal SQL OPS：整个 TiDB 集群内部 SQL 语句执行的 QPS。内部 SQL 语句是 TiDB 内部自动执行的 SQL 语句，一般由用户 SQL 语句来触发或者内部定时任务触发。

- Server
  - Uptime：每个 TiDB 实例的运行时间
  - Memory Usage：每个 TiDB 实例的内存使用统计，分为进程占用内存和 Golang 在堆上申请的内存
  - CPU Usage：每个 TiDB 实例的 CPU 使用统计
  - Connection Count：每个 TiDB 的连接数
  - Open FD Count：每个 TiDB 实例的打开的文件描述符统计
  - Disconnection Count：每个 TiDB 实例断开连接的数量
  - Event OPM：每个 TiDB 实例关键事件统计，例如 start，close，graceful-shutdown，kill，hang 等
  - Goroutine Count：每个 TiDB 实例的 Goroutine 数量
  - Prepare Statement Count：每个 TiDB 实例现存的 Prepare 语句数以及总数
  - Keep Alive OPM：每个 TiDB 实例每分钟刷新监控的次数，通常不需要关注
  - Panic And Critical Error：TiDB 中出现的 Panic、Critical Error 数量
  - Time Jump Back OPS：每个 TiDB 实例上每秒操作系统时间回跳的次数
  - Get Token Duration：每个连接获取 Token 的耗时

  - Skip Binlog Count：TiDB 写入 Binlog 失败的数量
  - Client Data Traffic：TiDB 和客户端的数据流量统计
  - Handshake Error OPS：每个 TiDB 实例每秒握手错误的次数

- Transaction
  - Transaction OPS：每秒事务的执行数量
  - Duration：事务执行耗时
  - Transaction Statement Num：事务中的 SQL 语句数量

  - Transaction Retry Num：事务重试次数
  - Session Retry Error OPS：每秒事务重试时遇到的错误数量，分为重试失败和超过最大重试次数两种类型
  - Commit Token Wait Duration：事务提交时的流控队列等待耗时。当出现较长等待时，代表提交事务过大，正在限流。如果系统还有资源可以使用，可以通过增大 TiDB 配置文件中 `committer-concurrency` 值来加速提交
  - KV Transaction OPS：每个 TiDB 内部每秒执行的事务数量
    * 一个用户的事务，在 TiDB 内部可能会触发多次事务执行，其中包含，内部元数据的读取，用户事务原子性地多次重试执行等
    * TiDB 内部的定时任务也会通过事务来操作数据库，这部分也包含在这个面板里
  - KV Transaction Duration：每个 TiDB 内部执行事务的耗时
  - Transaction Regions Num：事务操作的 Region 数量

  - Transaction Write KV Num Rate and Sum：事务写入 KV 的速率和总计
  - Transaction Write KV Num：事务操作的 KV 数量
  - Statement Lock Keys：单个语句的加锁个数
  - Send HeartBeat Duration：事务发送心跳的时间间隔
  - Transaction Write Size Bytes Rate and sum：事务写入字节数的速率和总计
  - Transaction Write Size Bytes：事务写入的数据大小
  - Acquire Pessimistic Locks Duration：加锁所消耗的时间
  - TTL Lifetime Reach Counter：事务的 TTL 达到了上限的数量。TTL 上限默认值 10 分钟，它的含义是从悲观事务第一次加锁，或者乐观事务的第一个 prewrite 开始，超过了 10 分钟。可以通过修改 TiDB 配置文件中 `max-txn-ttl` 来改变 TTL 寿命上限
  - Load Safepoint OPS：加载 Safepoint 的次数统计。Safepoint 作用是在事务读数据时，保证不读到 Safepoint 之前的数据，保证数据安全。因为，Safepoint 之前的数据有可能被 GC 清理掉

  - Pessimistic Statement Retry OPS：悲观语句重试次数统计。当语句尝试加锁时，可能遇到写入冲突，此时，语句会重新获取新的 snapshot 并再次加锁
  - Async Commit Transaction Counter：启用 Async commit 机制的事务数量，分为成功、失败两种

- Executor

  - Parse Duration：SQL 语句解析耗时统计
  - Compile Duration：将解析后的 SQL AST 编译成执行计划耗时统计
  - Execution Duration：执行 SQL 语句执行计划耗时统计
  - Expensive Executor OPS：每秒消耗系统资源比较多的算子统计，包括 Merge Join，Hash Join，Index Look Up Join，Hash Agg，Stream Agg，Sort，TopN 等
  - Queries Using Plan Cache OPS：每秒使用 Plan Cache 的查询数量统计

- Distsql

  - Distsql Duration：Distsql 处理的时长
  - Distsql QPS：Distsql 的数量统计
  - Distsql Partial QPS：每秒 Partial Results 的数量
  - Scan Keys Num：每个 Query 扫描的 Key 的数量
  - Scan Keys Partial Num：每一个 Partial Result 扫描的 Key 的数量
  - Partial Num：每个 SQL 语句 Partial Results 的数量

- KV Errors
  - KV Backoff Duration：KV 每个请求重试的总时间。TiDB 向 TiKV 发送请求时可能遇到错误，TiDB 对每个向 TiKV 的请求都有重试机制，这里记录的是一个请求重试的总时间
  - TiClient Region Error OPS：TiKV 返回 Region 相关错误信息的数量
  - KV Backoff OPS：TiKV 返回错误信息的数量
  - Lock Resolve OPS：TiDB 清理锁操作的数量。当 TiDB 的读写请求遇到锁时，会尝试进行锁清理
  - Other Errors OPS：其他类型的错误数量，包括清锁和更新 SafePoint
- KV Request
  - KV Request OPS：KV Request 执行次数，根据 TiKV 显示
  - KV Request Duration 99 by store：KV Request 执行时间，根据 TiKV 显示
  - KV Request Duration 99 by type：KV Request 执行时间，根据类型显示
- PD Client
  - PD Client CMD OPS：PD Client 每秒执行命令数量统计
  - PD Client CMD Duration：PD Client 执行命令耗时
  - PD Client CMD Fail OPS：PD Client 每秒执行命令失败统计
  - PD TSO OPS：TiDB 每秒从 PD 获取 TSO 的数量
  - PD TSO Wait Duration：TiDB 等待从 PD 返回 TSO 的时间
  - PD TSO RPC Duration：TiDB 从向 PD 发送获取 TSO 的请求到接收到 TSO 花费的时间
  - Start TSO Wait Duration：TiDB 从向 PD 发送获取 start tso 请求开始到开始等待 tso 返回的时间
- Schema Load
  - Load Schema Duration：TiDB 从 TiKV 获取 Schema 的时间
  - Load Schema OPS：TiDB 从 TiKV 每秒获取 Schema 的数量统计
  - Schema Lease Error OPM：Schema Lease 出错，包括 change 和 outdate 两种，change 代表 schema 发生了变化，outdate 代表无法更新 schema，属于较严重错误，出现 outdate 错误时会报警
  - Load Privilege OPS：TiDB 从 TiKV 每秒获取权限信息的数量统计
- DDL
  - DDL Duration 95：DDL 语句处理时间的 95% 分位
  - Batch Add Index Duration 100：创建索引时每个 Batch 所花费的最大时间统计
  - DDL Waiting Jobs Count：等待的 DDL 任务数量
  - DDL META OPM：DDL 每分钟获取 META 的次数
  - DDL Worker Duration 99：每个 DDL worker 执行时间 99% 分位
  - Deploy Syncer Duration：Schema Version Syncer 初始化，重启，清空等操作耗时
  - Owner Handle Syncer Duration：DDL Owner 在执行更新，获取以及检查 Schema Version 的耗时
  - Update Self Version Duration：Schema Version Syncer 更新版本信息耗时
  - DDL OPM：DDL 语句的每秒执行次数
  - DDL add index progress in percentage：添加索引的进度展示
- Statistics
  - Auto Analyze Duration 95：自动 ANALYZE 耗时统计
  - Auto Analyze QPS：自动 ANALYZE 数量统计
  - Stats Inaccuracy Rate：统计信息不准确度统计
  - Pseudo Estimation OPS：使用假的统计信息优化 SQL 的数量统计

  - Dump Feedback OPS：存储统计信息 Feedback 的数量统计
  - Store Query Feedback QPS：存储合并查询的 Feedback 信息的每秒操作数量，该操作在 TiDB 内存中进行
  - Significant Feedback：重要的 Feedback 更新统计信息的数量统计
  - Update Stats OPS：利用 Feedback 更新统计信息的数量统计
  - Fast Analyze Status 100：快速收集统计信息的状态统计

- Owner
  - New ETCD Session Duration 95：创建一个新的 etcd 会话花费的时间。TiDB 通过 etcd client 连接 PD 中的 etcd 保存/读取部分元数据信息。这里记录了创建会话花费的时间
  - Owner Watcher OPS：DDL owner watch PD 的 etcd 的元数据的 goroutine 的每秒操作次数

- Meta
  - AutoID QPS：AutoID 相关操作的数量统计，包括全局 ID 分配、单个 Table AutoID 分配、单个 Table AutoID Rebase 三种操作
  - AutoID Duration：AutoID 相关操作的耗时
  - Region Cache Error OPS：TiDB 缓存的 region 信息每秒遇到的错误次数
  - Meta Operations Duration 99：元数据操作延迟

- GC
  - Worker Action OPM：GC 相关操作的数量，包括 run_job，resolve_lock，delete_range 等操作
  - Duration 99：GC 相关操作的耗时统计
  - Config：GC 的数据保存时长（life time）和 GC 运行间隔（run interval）配置
  - GC Failure OPM：GC 相关操作失败数量统计
  - Delete Range Failure OPM：Delete range 失败的次数
  - Too Many Locks Error OPM：GC 清锁过多错误的数量
  - Action Result OPM：GC 相关操作结果数量
  - Delete Range Task Status：Delete range 的任务状态，包含完成和失败状态
  - Push Task Duration 95：将 GC 子任务推送给 GC worker 的耗时

- Batch Client
  - Pending Request Count by TiKV：等待处理的 Batch 消息数量
  - Wait Duration 95：等待处理的 Batch 消息延迟
  - Batch Client Unavailable Duration 95：Batch 客户端不可用的时间
  - No Available Connection Counter：Batch 客户端找不到可用链接的次数

12.8.3 PD 重要监控指标详解

使用 TiDB Ansible 或 TiUP 部署 TiDB 集群时，一键部署监控系统 (Prometheus & Grafana)，监控架构参见TiDB 监控框架概述。

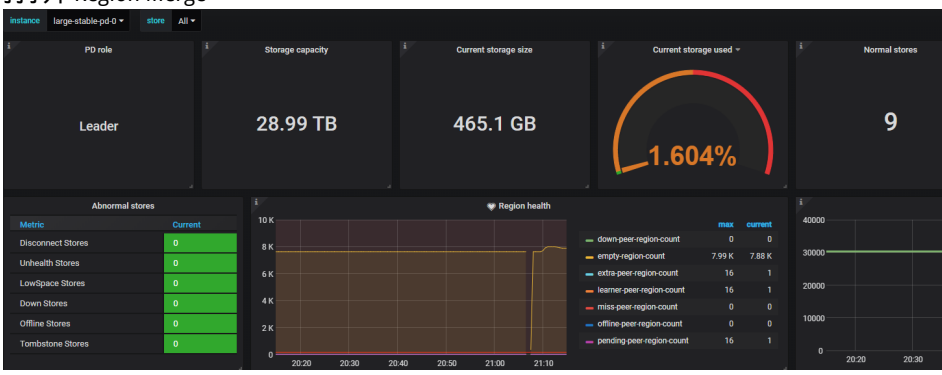
目前 Grafana Dashboard 整体分为 PD、 TiDB、 TiKV、 Node_exporter、 Overview 等。

对于日常运维，我们通过观察 PD 面板上的 Metrics，可以了解 PD 当前的状态。

以下为 PD Dashboard 监控说明：

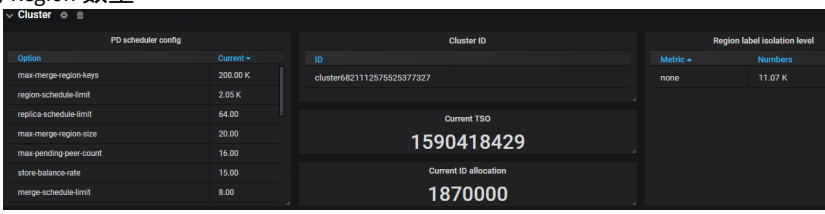
- PD role：当前 PD 的角色

- Storage capacity：TiDB 集群总可用数据库空间大小
- Current storage size：TiDB 集群目前已用数据库空间大小
- Current storage usage：TiDB 集群存储空间的使用率
- Normal stores：处于正常状态的节点数目
- Number of Regions：当前集群的 Region 总量
- Abnormal stores：处于异常状态的节点数目，正常情况应当为 0
- Region health：集群所有 Region 的状态。通常情况下，pending 或 down 的 peer 应该少于 100，miss 的 peer 不能一直大于 0，empty Region 过多需及时打开 Region Merge
- Current peer count：当前集群 peer 的总量



12.8.3.1 Cluster

- PD scheduler config：PD 调度配置列表
- Cluster ID：集群的 cluster id，唯一标识
- Current TSO：当前分配 TSO 的物理时间戳部分
- Current ID allocation：当前可分配 ID 的最大值
- Region label isolation level：不同 label 所在的 level 的 Region 数量
- Label distribution：集群中 TiKV 节点的 label 分布情况



12.8.3.2 Operator

- Schedule operator create：新创建的不同 operator 的数量，单位 opm 代表一分钟内创建的个数
- Schedule operator check：已检查的 operator 的次数，主要检查是否当前步骤已经执行完成，如果是，则执行下一个步骤
- Schedule operator finish：已完成调度的 operator 的数量
- Schedule operator timeout：已超时的 operator 的数量
- Schedule operator replaced or canceled：已取消或者被替换的 operator 的数量
- Schedule operators count by state：不同状态的 operator 的数量
- Operator finish duration：已完成的 operator 所花费的最长时间
- Operator step duration：已完成的 operator 的步骤所花费的最长时间

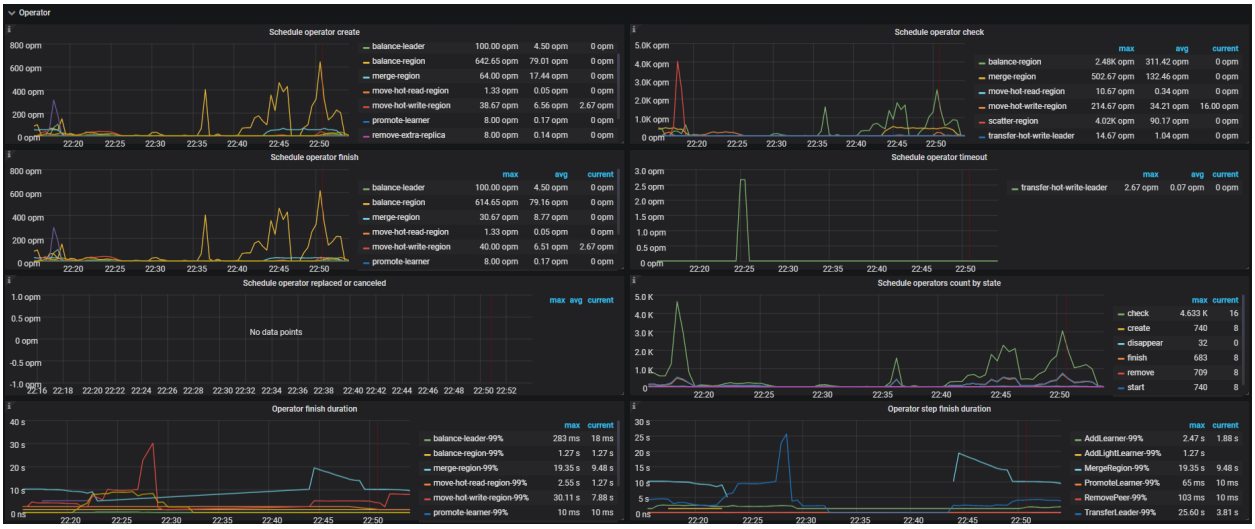


图 208: PD Dashboard - Operator metrics

12.8.3.3 Statistics - Balance

- Store capacity：每个 TiKV 实例的总的空间大小
- Store available：每个 TiKV 实例的可用空间大小
- Store used：每个 TiKV 实例的已使用空间大小
- Size amplification：每个 TiKV 实例的空间放大比率
- Size available ratio：每个 TiKV 实例的可用空间比率
- Store leader score：每个 TiKV 实例的 leader 分数
- Store Region score：每个 TiKV 实例的 Region 分数
- Store leader size：每个 TiKV 实例上所有 leader 的大小
- Store Region size：每个 TiKV 实例上所有 Region 的大小
- Store leader count：每个 TiKV 实例上所有 leader 的数量
- Store Region count：每个 TiKV 实例上所有 Region 的数量

图 209: PD Dashboard - Balance metrics

12.8.3.4 Statistics - hot write

- Hot Region’s leader distribution：每个 TiKV 实例上成为写入热点的 leader 的数量
- Total written bytes on hot leader Regions：每个 TiKV 实例上所有成为写入热点的 leader 的总的写入流量大小
- Hot write Region’s peer distribution：每个 TiKV 实例上成为写入热点的 peer 的数量
- Total written bytes on hot peer Regions：每个 TiKV 实例上所有成为写入热点的 peer 的写入流量大小
- Store Write rate bytes：每个 TiKV 实例总的写入的流量
- Store Write rate keys：每个 TiKV 实例总的写入 keys
- Hot cache write entry number：每个 TiKV 实例进入热点统计模块的 peer 的数量
- Selector events：热点调度中选择器的事件发生次数
- Direction of hotspot move leader：热点调度中 leader 的调度方向，正数代表调入，负数代表调出
- Direction of hotspot move peer：热点调度中 peer 的调度方向，正数代表调入，负数代表调出

图 210: PD Dashboard - Hot write metrics

12.8.3.5 Statistics - hot read

- Hot Region’s leader distribution：每个 TiKV 实例上成为读取热点的 leader 的数量
- Total read bytes on hot leader Regions：每个 TiKV 实例上所有成为读取热点的 leader 的总的读取流量大小
- Store read rate bytes：每个 TiKV 实例总的读取的流量
- Store read rate keys：每个 TiKV 实例总的读取 keys
- Hot cache read entry number：每个 TiKV 实例进入热点统计模块的 peer 的数量

图 211: PD Dashboard - Hot read metrics

12.8.3.6 Scheduler

- Scheduler is running：所有正在运行的 scheduler
- Balance leader movement：leader 移动的详细情况
- Balance Region movement：Region 移动的详细情况
- Balance leader event：balance leader 的事件数量
- Balance Region event：balance Region 的事件数量
- Balance leader scheduler：balance-leader scheduler 的状态
- Balance Region scheduler：balance-region scheduler 的状态
- Replica checker：replica checker 的状态
- Rule checker：rule checker 的状态
- Region merge checker：merge checker 的状态
- Filter target：尝试选择 Store 作为调度 taget 时没有通过 Filter 的计数
- Filter source：尝试选择 Store 作为调度 source 时没有通过 Filter 的计数
- Balance Direction：Store 被选作调度 target 或 source 的次数
- Store Limit：Store 的调度限流状态

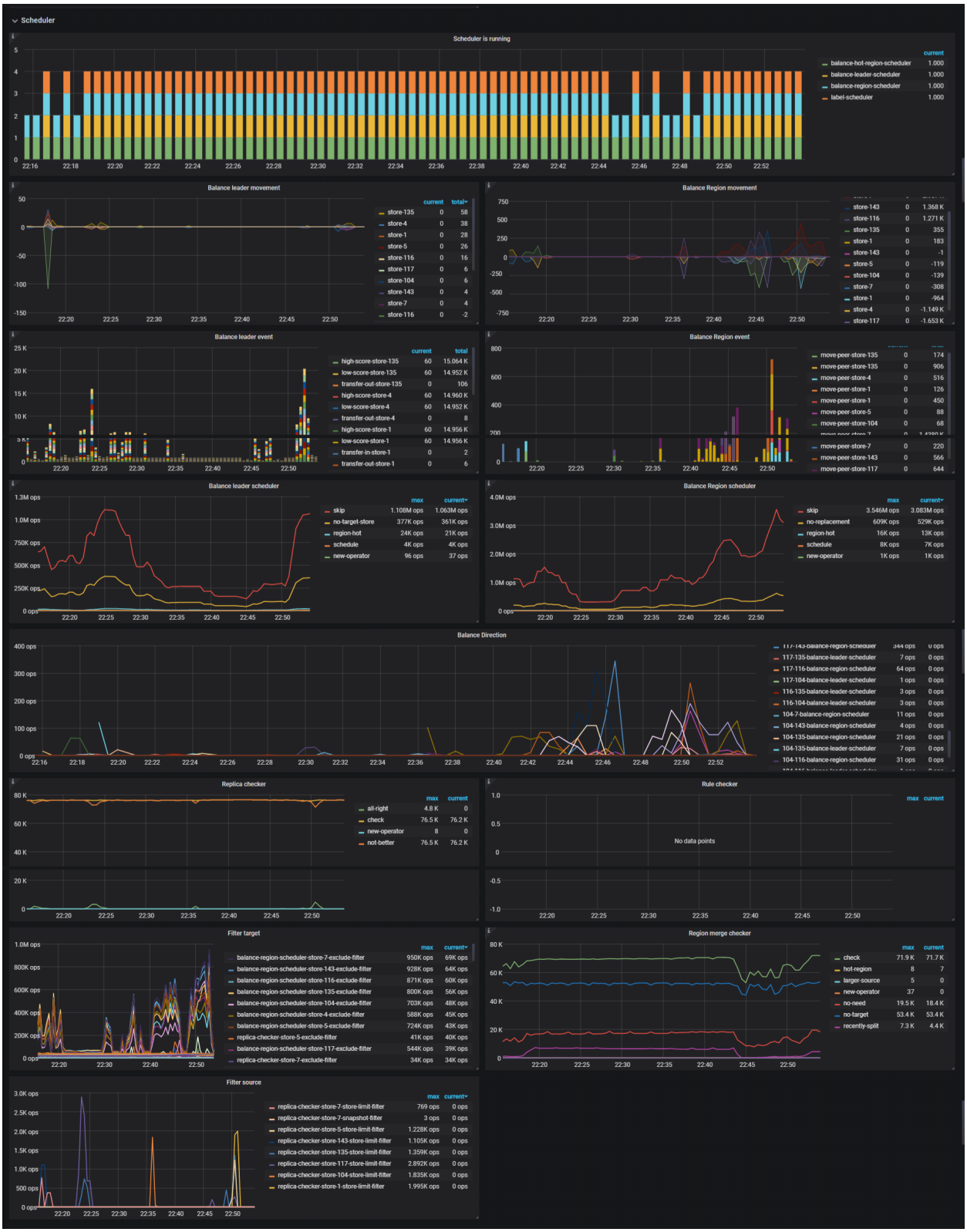


图 212: PD Dashboard - Scheduler metrics

12.8.3.7 gRPC

- Completed commands rate：gRPC 命令的完成速率
- 99% Completed commands duration：99% 命令的最长消耗时间

图 213: PD Dashboard - gRPC metrics

12.8.3.8 etcd

- Handle transactions count：etcd 的事务个数
- 99% Handle transactions duration：99% 的情况下，处理 etcd 事务所需花费的时间
- 99% WAL fsync duration：99% 的情况下，持久化 WAL 所需花费的时间，这个值通常应该小于 1s
- 99% Peer round trip time seconds：99% 的情况下，etcd 的网络延时，这个值通常应该小于 1s
- etcd disk WAL fsync rate：etcd 持久化 WAL 的速率
- Raft term：当前 Raft 的 term
- Raft committed index：最后一次 commit 的 Raft index
- Raft applied index：最后一次 apply 的 Raft index

图 214: PD Dashboard - etcd metrics

12.8.3.9 TiDB

- PD Server TSO handle time and Client recv time：从 PD 开始处理 TSO 请求到 client 端接收到 TSO 的总耗时
- Handle requests count：TiDB 的请求数量
- Handle requests duration：每个请求所花费的时间，99% 的情况下，应该小于 100ms

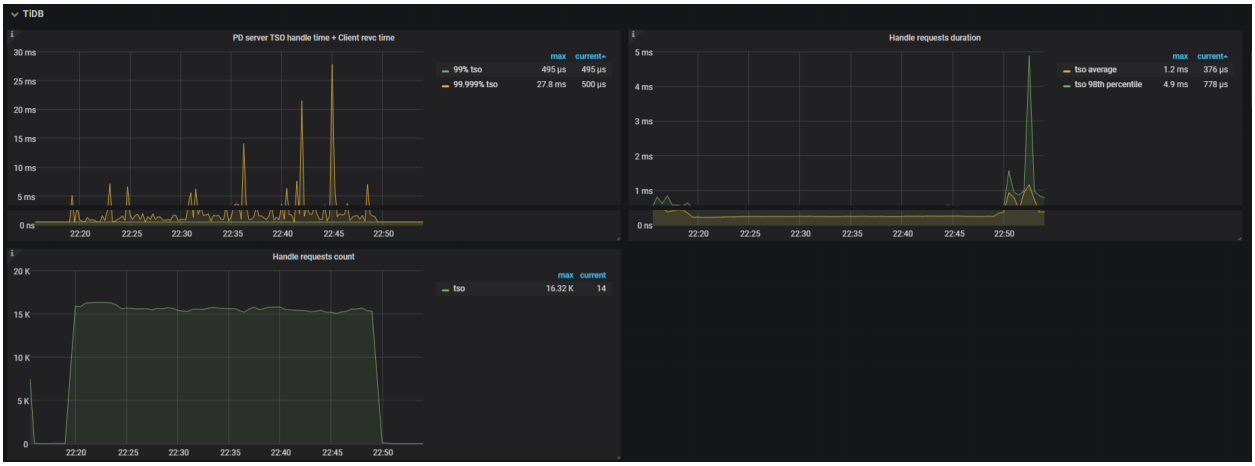


图 215: PD Dashboard - TiDB metrics

12.8.3.10 Heartbeat

- Heartbeat region event QPS：心跳处理 region 的 QPS，包括更新缓存和持久化
- Region heartbeat report：TiKV 向 PD 发送的心跳个数
- Region heartbeat report error：TiKV 向 PD 发送的异常的心跳个数
- Region heartbeat report active：TiKV 向 PD 发送的正常的心跳个数
- Region schedule push：PD 向 TiKV 发送的调度命令的个数
- 99% Region heartbeat latency：99% 的情况下，心跳的延迟

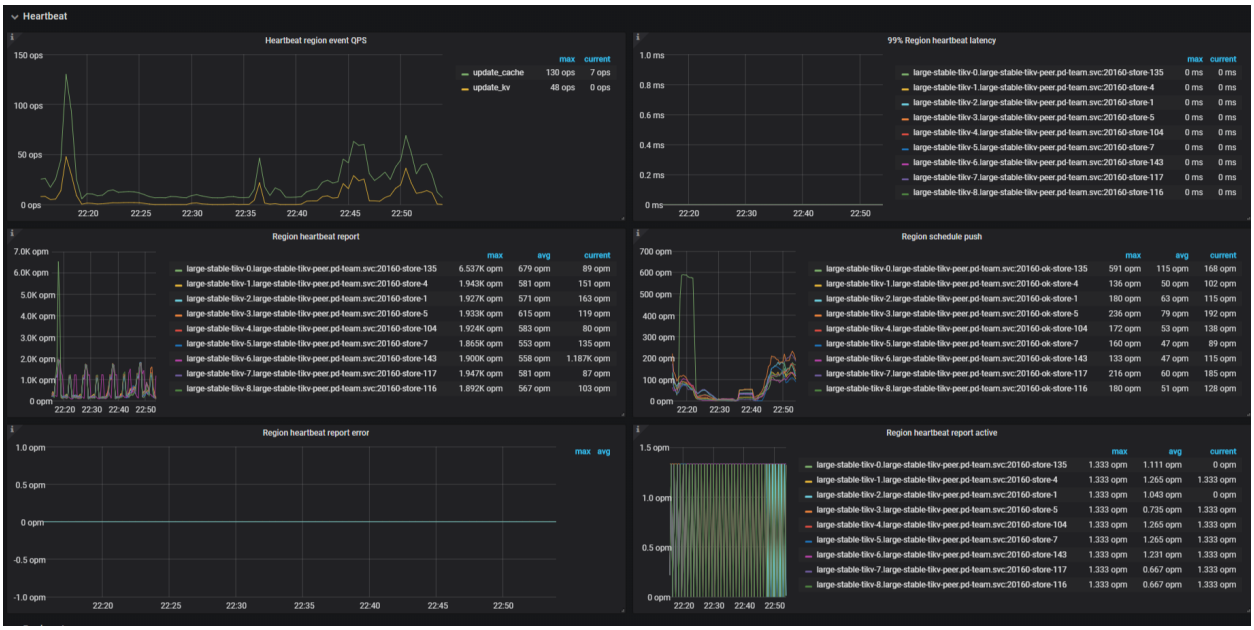


图 216: PD Dashboard - Heartbeat metrics

12.8.3.11 Region storage

- Syncer Index：Leader 记录 Region 变更历史的最大 index
- history last index：Follower 成功同步的 Region 变更历史的 index

图 217: PD Dashboard - Region storage

12.8.4 TiKV 监控指标详解

使用 TiDB Ansible 或 TiUP 部署 TiDB 集群时，一键部署监控系统 (Prometheus & Grafana)，监控架构参见TiDB 监控框架概述。

目前 Grafana Dashboard 整体分为 PD、TiDB、TiKV、Node_exporter、Overview 等。

对于日常运维，我们通过观察 TiKV-Details 面板上的指标，可以了解 TiKV 当前的状态。根据 性能地图 可以检查集群的状态是否符合预期。

以下为 TiKV-Details 默认的监控信息：

12.8.4.1 Cluster

- Store size：每个 TiKV 实例的使用的存储空间的大小
- Available size：每个 TiKV 实例的可用的存储空间的大小
- Capacity size：每个 TiKV 实例的存储容量的大小
- CPU：每个 TiKV 实例 CPU 的使用率
- Memory：每个 TiKV 实例内存的使用情况
- IO utilization：每个 TiKV 实例 IO 的使用率
- MBps：每个 TiKV 实例写入和读取的数据量大小
- QPS：每个 TiKV 实例上各种命令的 QPS
- Errps：每个 TiKV 实例上 gRPC 消息失败的速率
- leader：每个 TiKV 实例 leader 的个数
- Region：每个 TiKV 实例 Region 的个数
- Uptime：自上次重启以来 TiKV 正常运行的时间

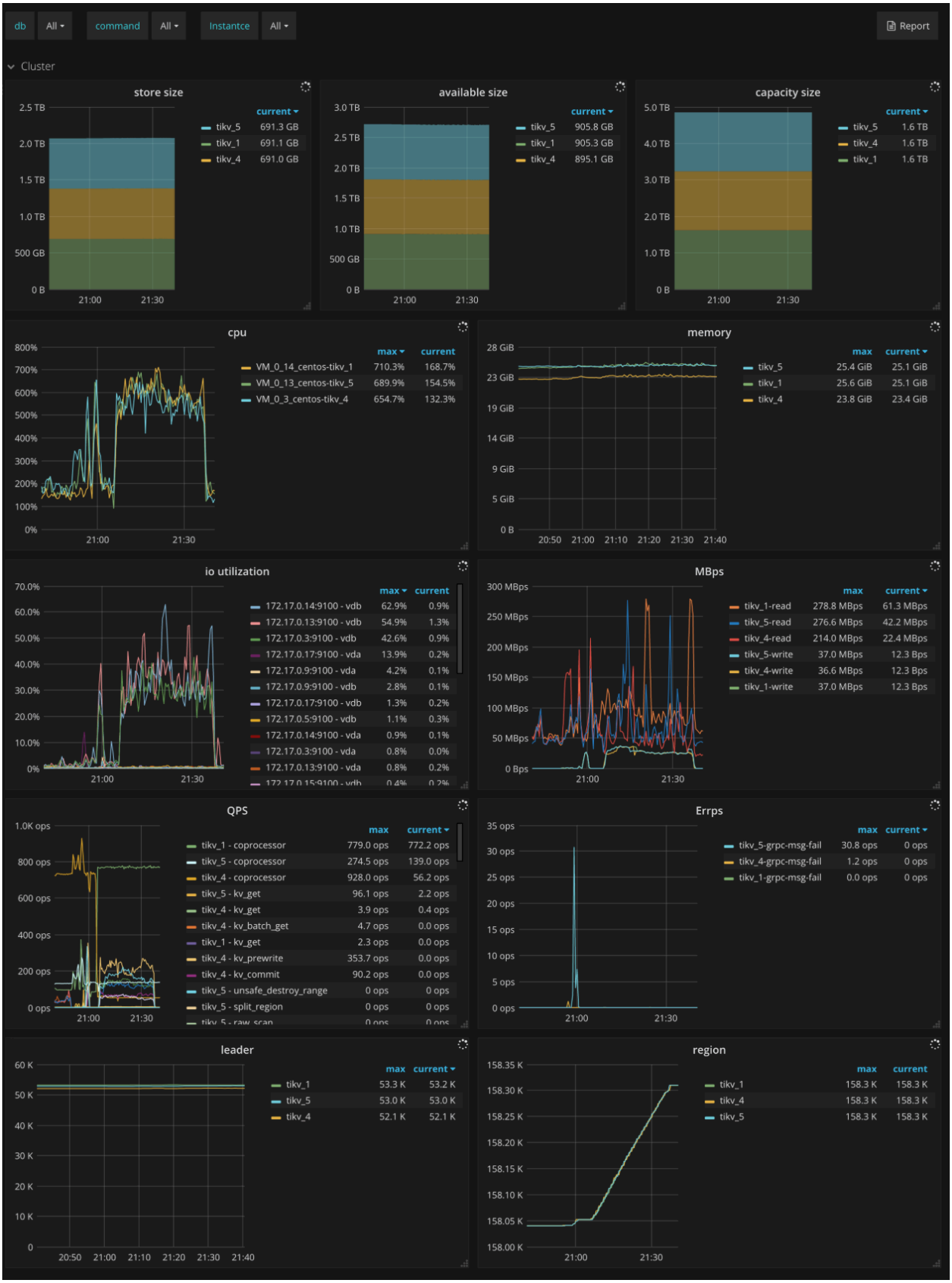


图 218: TiKV Dashboard - Cluster metrics

12.8.4.2 Errors

- Critical error：严重错误的数量
- Server is busy：各种会导致 TiKV 实例暂时不可用的事件个数，如 write stall，channel full 等，正常情况下应当为 0
- Server report failures：server 报错的消息个数，正常情况下应当为 0
- Raftstore error：每个 TiKV 实例上 raftstore 发生错误的个数
- Scheduler error：每个 TiKV 实例上 scheduler 发生错误的个数
- Coprocessor error：每个 TiKV 实例上 coprocessor 发生错误的个数
- gRPC message error：每个 TiKV 实例上 gRPC 消息发生错误的个数
- Leader drop：每个 TiKV 实例上 drop leader 的个数
- Leader missing：每个 TiKV 实例上 missing leader 的个数

图 219: TiKV Dashboard - Errors metrics

12.8.4.3 Server

- CF size：每个列族的大小
- Store size：每个 TiKV 实例的使用的存储空间的大小
- Channel full：每个 TiKV 实例上 channel full 错误的数量，正常情况下应当为 0
- Active written leaders：各个 TiKV 实例中正在被写入的 Leader 的数量
- Approximate Region size：每个 Region 近似的大小
- Approximate Region size Histogram：每个 Region 近似大小的直方图
- Region average written keys：每个 TiKV 实例上所有 Region 的平均 key 写入个数
- Region average written bytes：每个 TiKV 实例上所有 Region 的平均写入大小

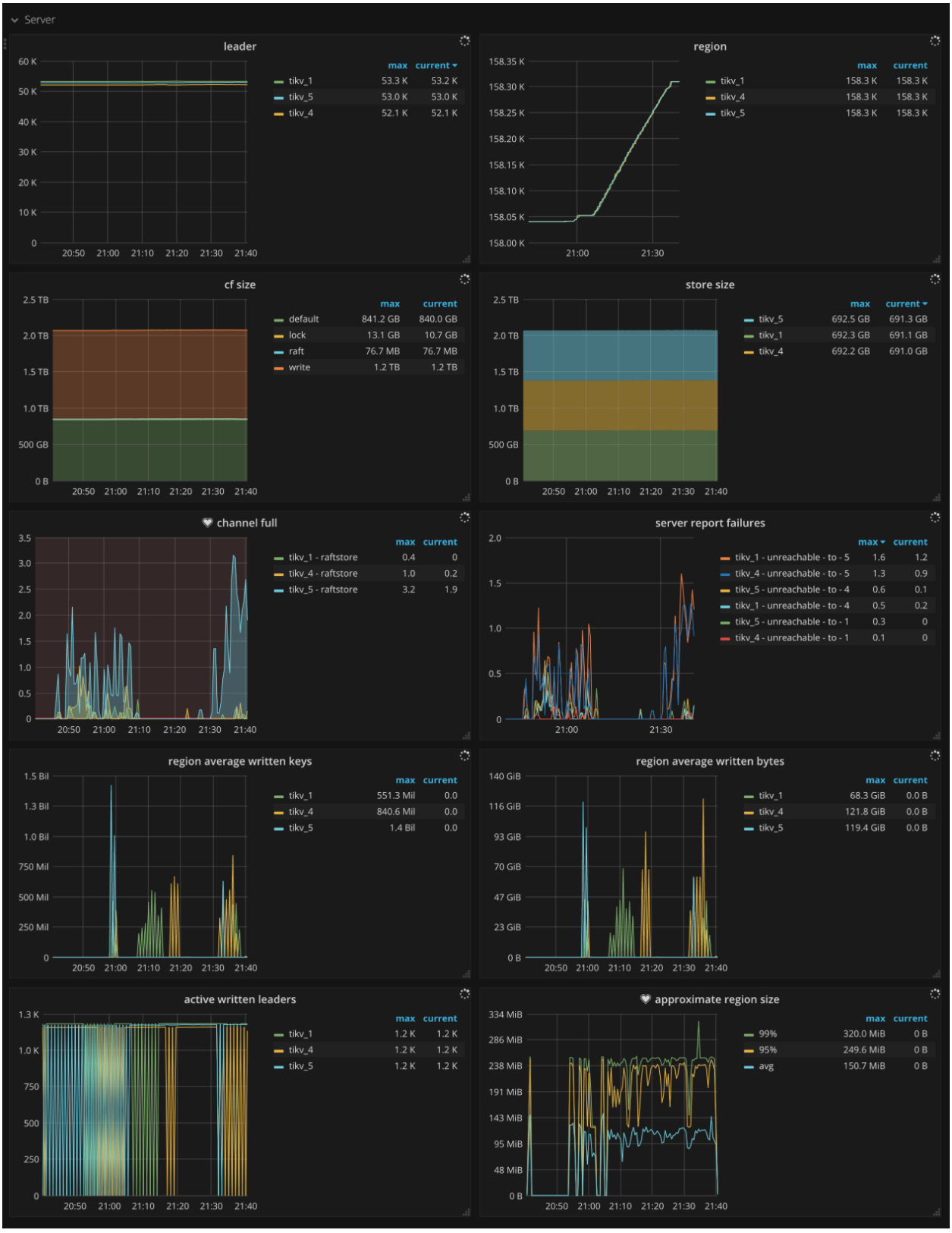


图 220: TiKV Dashboard - Server metrics

12.8.4.4 gRPC

- gRPC message count：每种 gRPC 请求的速度
- gRPC message failed：失败的 gRPC 请求的速度
- 99% gRPC message duration：99% gRPC 请求的执行时间小于该值
- Average gRPC message duration：gRPC 请求平均的执行时间
- gRPC batch size：TiDB 与 TiKV 之间 grpc 请求的 batch 大小
- raft message batch size：TiKV 与 TiKV 之间 raft 消息的 batch 大小

12.8.4.5 Thread CPU

- Raft store CPU：raftstore 线程的 CPU 使用率，通常应低于 80% * `raftstore.store-pool-size`
- Async apply CPU：async apply 线程的 CPU 使用率，通常应低于 90% * `raftstore.apply-pool-size`
- Scheduler worker CPU：scheduler worker 线程的 CPU 使用率，通常应低于 90% * `storage.scheduler-worker-pool-size`
- gRPC poll CPU：gRPC 线程的 CPU 使用率，通常应低于 80% * `server.grpc-concurrency`
- Unified read pool CPU：unified read pool 线程的 CPU 使用率
- Storage ReadPool CPU：storage read pool 线程的 CPU 使用率
- Coprocessor CPU：coprocessor 线程的 CPU 使用率
- RocksDB CPU：RocksDB 线程的 CPU 使用率
- Split check CPU：split check 线程的 CPU 使用率
- GC worker CPU：GC worker 线程的 CPU 使用率
- Snapshot worker CPU：snapshot worker 线程的 CPU 使用率

12.8.4.6 PD

- PD requests：TiKV 发送给 PD 的请求速度
- PD request duration (average)：TiKV 发送给 PD 的请求处理的平均时间
- PD heartbeats：发送给 PD 的心跳的速度
- PD validate peers：TiKV 发送给 PD 用于验证 TiKV 的 peer 有效的消息的速度

12.8.4.7 Raft IO

- Apply log duration：Raft apply 日志所花费的时间
- Apply log duration per server：每个 TiKV 实例上 Raft apply 日志所花费的时间
- Append log duration：Raft append 日志所花费的时间
- Append log duration per server：每个 TiKV 实例上 Raft append 日志所花费的时间
- Commit log duration：Raft commit 日志所花费的时间
- Commit log duration per server：每个 TiKV 实例上 Raft commit 日志所花费的时间

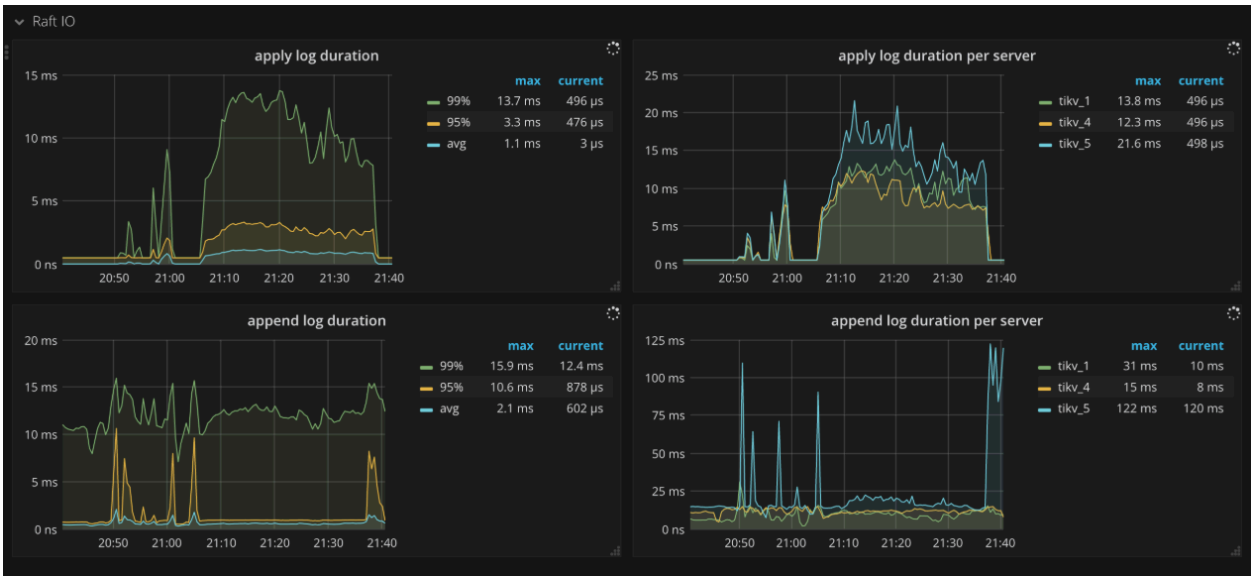


图 221: TiKV Dashboard - Raft IO metrics

12.8.4.8 Raft process

- Ready handled：Raft 中不同 ready 类型的 ops
- 0.99 Duration of Raft store events：99% 的 raftstore 事件所花费的时间
- Process ready duration：处理 ready 所花费的时间
- Process ready duration per server：每个 TiKV 实例处理 ready 所花费的时间，99.99% 的情况下，应该小于 2s

图 222: TiKV Dashboard - Raft process metrics

12.8.4.9 Raft message

- Sent messages per server：每个 TiKV 实例发送 Raft 消息的 ops
- Flush messages per server：每个 TiKV 实例中 raft client 往外 flush Raft 消息的 ops
- Receive messages per server：每个 TiKV 实例接受 Raft 消息的 ops
- Messages：发送不同类型的 Raft 消息的 ops
- Vote：Raft 投票消息发送的 ops
- Raft dropped messages：每秒钟丢弃不同类型的 Raft 消息的个数

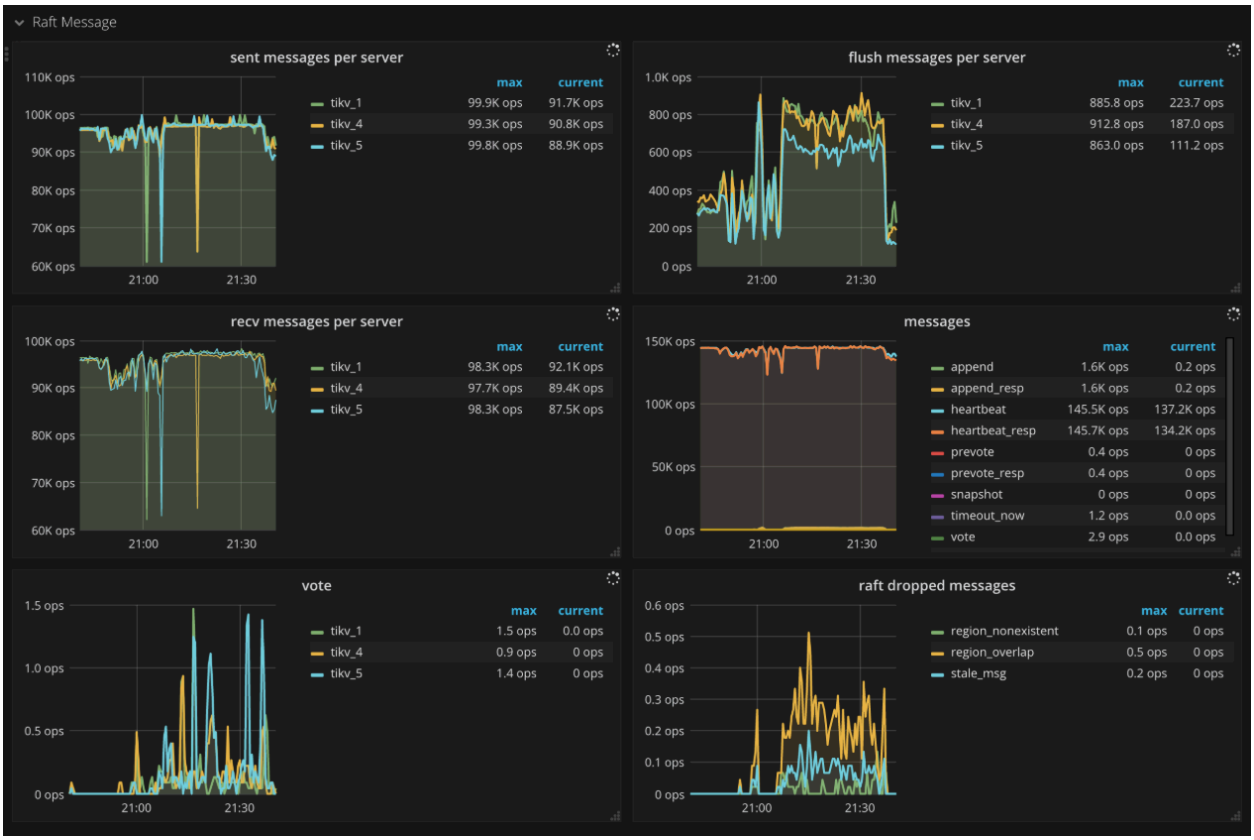


图 223: TiKV Dashboard - Raft message metrics

12.8.4.10 Raft propose

- Raft apply proposals per ready：在一个 batch 内，apply proposal 时每个 ready 中包含 proposal 的个数的直方图
- Raft read/write proposals：不同类型的 proposal 的 ops
- Raft read proposals per server：每个 TiKV 实例发起读 proposal 的 ops
- Raft write proposals per server：每个 TiKV 实例发起写 proposal 的 ops
- Propose wait duration：proposal 的等待时间的直方图
- Propose wait duration per server：每个 TiKV 实例上每个 proposal 的等待时间的直方图
- Apply wait duration：apply 的等待时间的直方图
- Apply wait duration per server：每个 TiKV 实例上每个 apply 的等待时间的直方图
- Raft log speed：peer propose 日志的平均速度

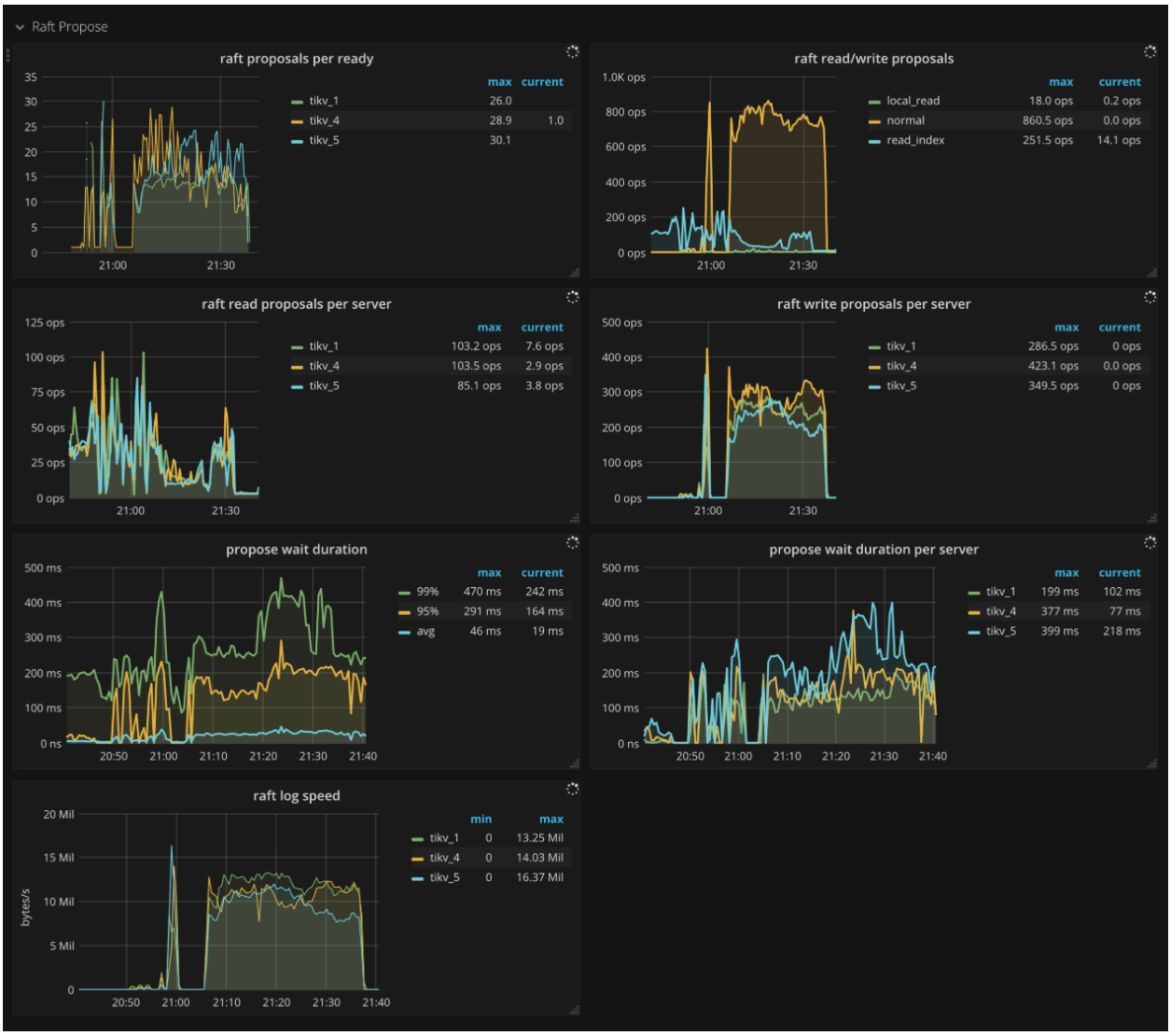


图 224: TiKV Dashboard - Raft propose metrics

12.8.4.11 Raft admin

- Admin proposals：admin proposal 的 ops
- Admin apply：apply 命令的 ops
- Check split：split check 命令的 ops
- 99.99% Check split duration：99.99% 的情况下，split check 所需花费的时间

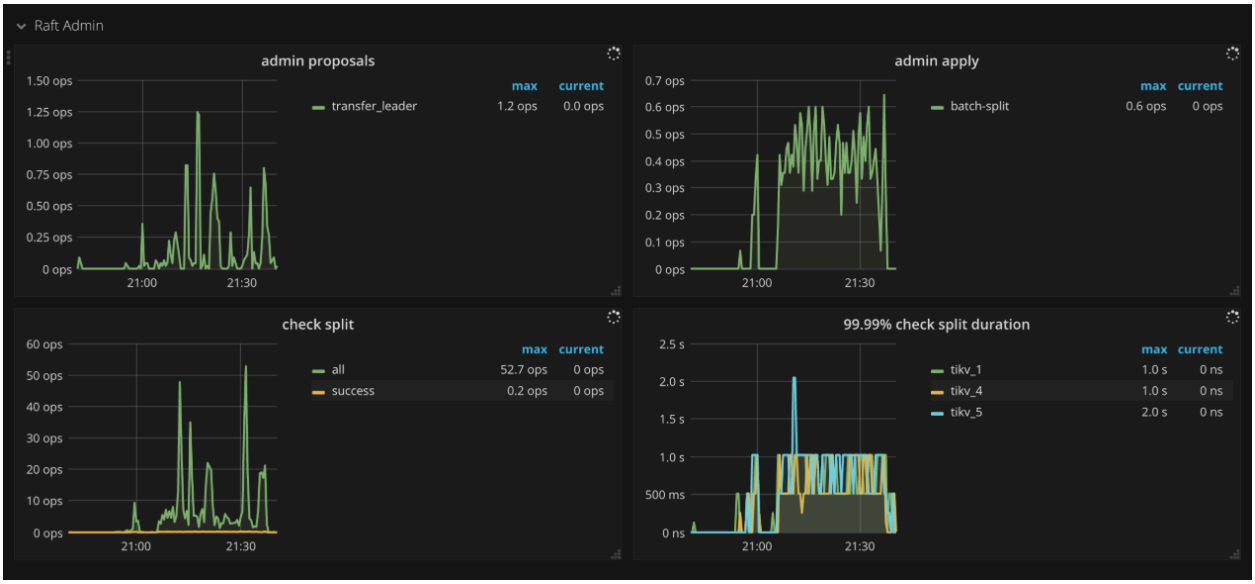


图 225: TiKV Dashboard - Raft admin metrics

12.8.4.12 Local reader

- Local reader requests：所有请求的总数以及 local read 线程拒绝的请求数量

图 226: TiKV Dashboard - Local reader metrics

12.8.4.13 Unified Read Pool

- Time used by level：在 unified read pool 中每个级别使用的时间，级别 0 指小查询

- Level 0 chance：在 unified read pool 中调度的 level 0 任务的比例
- Running tasks：在 unified read pool 中并发运行的任务数量

12.8.4.14 Storage

- Storage command total：收到不同命令的 ops
- Storage async request error：异步请求出错的 ops
- Storage async snapshot duration：异步处理 snapshot 所花费的时间，99% 的情况下，应该小于 1s
- Storage async write duration：异步写所花费的时间，99% 的情况下，应该小于 1s

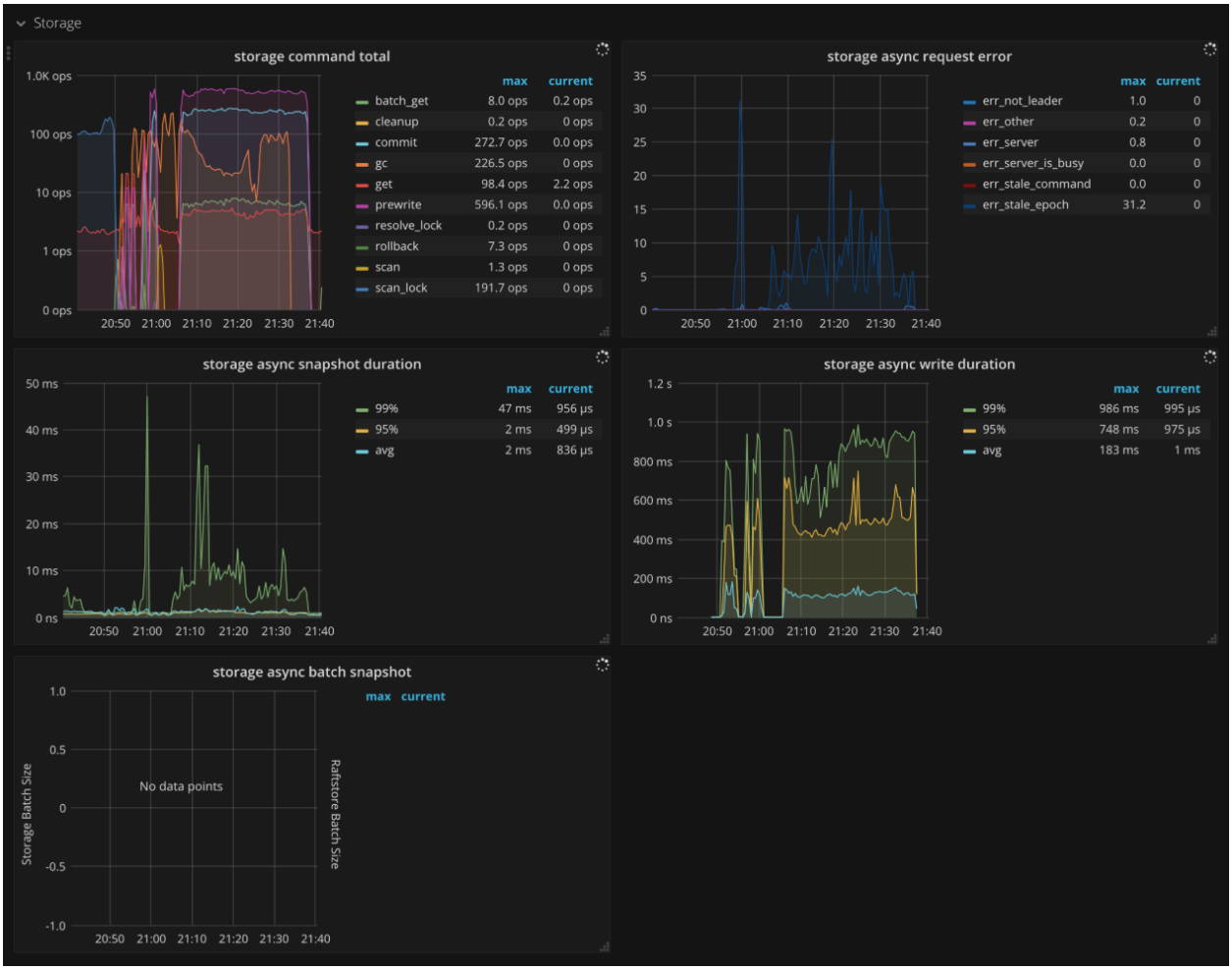


图 227: TiKV Dashboard - Storage metrics

12.8.4.15 Scheduler

- Scheduler stage total：每种命令不同阶段的 ops，正常情况下，不会在短时间内出现大量的错误
- Scheduler writing bytes：每个 TiKV 实例正在处理的命令的写入字节数量
- Scheduler priority commands：不同优先级命令的 ops

- Scheduler pending commands：每个 TiKV 实例上 pending 命令的 ops

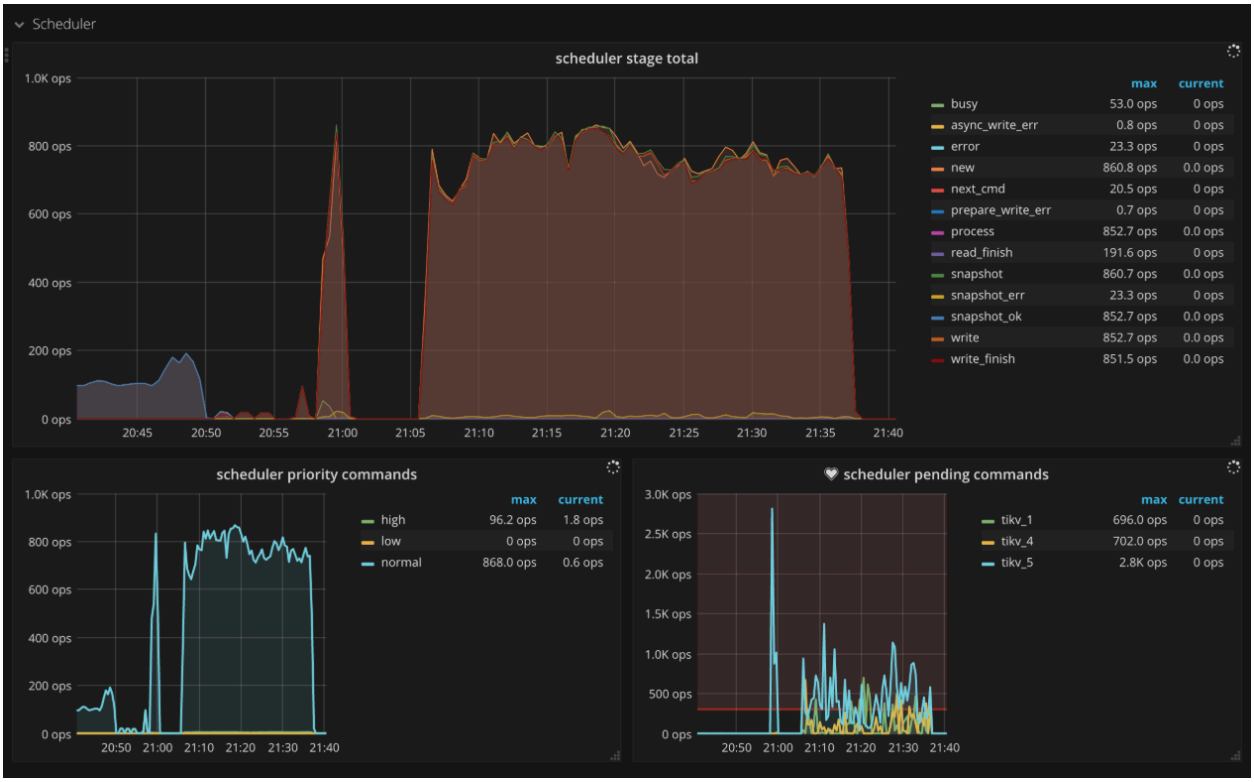


图 228: TiKV Dashboard - Scheduler metrics

12.8.4.16 Scheduler - commit

- Scheduler stage total：commit 中每个命令所处不同阶段的 ops，正常情况下，不会在短时间内出现大量的错误
- Scheduler command duration：执行 commit 命令所需花费的时间，正常情况下，应该小于 1s
- Scheduler latch wait duration：由于 latch wait 造成的时间开销，正常情况下，应该小于 1s
- Scheduler keys read：commit 命令读取 key 的个数
- Scheduler keys written：commit 命令写入 key 的个数
- Scheduler scan details：执行 commit 命令时，扫描每个 CF 中 key 的详细情况
- Scheduler scan details [lock]：执行 commit 命令时，扫描每个 lock CF 中 key 的详细情况
- Scheduler scan details [write]：执行 commit 命令时，扫描每个 write CF 中 key 的详细情况
- Scheduler scan details [default]：执行 commit 命令时，扫描每个 default CF 中 key 的详细情况

图 229: TiKV Dashboard - Scheduler commit metrics

12.8.4.17 Scheduler - pessimistic_rollback

- Scheduler stage total：pessimistic_rollback 中每个命令所处不同阶段的 ops，正常情况下，不会在短时间内出现大量的错误
- Scheduler command duration：执行 pessimistic_rollback 命令所需花费的时间，正常情况下，应该小于 1s
- Scheduler latch wait duration：由于 latch wait 造成的时间开销，正常情况下，应该小于 1s
- Scheduler keys read：pessimistic_rollback 命令读取 key 的个数
- Scheduler keys written：pessimistic_rollback 命令写入 key 的个数

- Scheduler scan details：执行 pessimistic_rollback 命令时，扫描每个 CF 中 key 的详细情况
- Scheduler scan details [lock]：执行 pessimistic_rollback 命令时，扫描每个 lock CF 中 key 的详细情况
- Scheduler scan details [write]：执行 pessimistic_rollback 命令时，扫描每个 write CF 中 key 的详细情况
- Scheduler scan details [default]：执行 pessimistic_rollback 命令时，扫描每个 default CF 中 key 的详细情况

12.8.4.18 Scheduler - prewrite

- Scheduler stage total：prewrite 中每个命令所处不同阶段的 ops，正常情况下，不会在短时间内出现大量的错误
- Scheduler command duration：执行 prewrite 命令所需花费的时间，正常情况下，应该小于 1s
- Scheduler latch wait duration：由于 latch wait 造成的时间开销，正常情况下，应该小于 1s
- Scheduler keys read：prewrite 命令读取 key 的个数
- Scheduler keys written：prewrite 命令写入 key 的个数
- Scheduler scan details：执行 prewrite 命令时，扫描每个 CF 中 key 的详细情况
- Scheduler scan details [lock]：执行 prewrite 命令时，扫描每个 lock CF 中 key 的详细情况
- Scheduler scan details [write]：执行 prewrite 命令时，扫描每个 write CF 中 key 的详细情况
- Scheduler scan details [default]：执行 prewrite 命令时，扫描每个 default CF 中 key 的详细情况

12.8.4.19 Scheduler - rollback

- Scheduler stage total：rollback 中每个命令所处不同阶段的 ops，正常情况下，不会在短时间内出现大量的错误
- Scheduler command duration：执行 rollback 命令所需花费的时间，正常情况下，应该小于 1s
- Scheduler latch wait duration：由于 latch wait 造成的时间开销，正常情况下，应该小于 1s
- Scheduler keys read：rollback 命令读取 key 的个数
- Scheduler keys written：rollback 命令写入 key 的个数
- Scheduler scan details：执行 rollback 命令时，扫描每个 CF 中 key 的详细情况
- Scheduler scan details [lock]：执行 rollback 命令时，扫描每个 lock CF 中 key 的详细情况
- Scheduler scan details [write]：执行 rollback 命令时，扫描每个 write CF 中 key 的详细情况
- Scheduler scan details [default]：执行 rollback 命令时，扫描每个 default CF 中 key 的详细情况

12.8.4.20 GC

- MVCC versions：每个 key 的版本个数
- MVCC delete versions：GC 删除掉的每个 key 的版本个数
- GC tasks：由 gc_worker 处理的 GC 任务的个数
- GC tasks Duration：执行 GC 任务时所花费的时间
- GC keys (write CF)：在 GC 过程中，write CF 中受影响的 key 的个数
- TiDB GC worker actions：TiDB GC worker 的不同 action 的个数
- TiDB GC seconds：TiDB 执行 GC 花费的时间
- GC speed：GC 每秒删除的 key 的数量
- TiKV AutoGC Working：Auto GC 管理器的工作状态
- ResolveLocks Progress：GC 第一阶段（ResolveLocks）的进度
- TiKV Auto GC Progress：GC 第二阶段的进度

- TiKV Auto GC SafePoint：TiKV GC 的 safe point 的数值，safe point 为当前 GC 的时间戳
- GC lifetime：TiDB 设置的 GC lifetime
- GC interval：TiDB 设置的 GC 间隔

12.8.4.21 Snapshot

- Rate snapshot message：发送 Raft snapshot 消息的速率
- 99% Handle snapshot duration：99% 的情况下，处理 snapshot 所需花费的时间
- Snapshot state count：不同状态的 snapshot 的个数
- 99.99% Snapshot size：99.99% 的 snapshot 的大小
- 99.99% Snapshot KV count：99.99% 的 snapshot 包含的 key 的个数

12.8.4.22 Task

- Worker handled tasks：worker 每秒钟处理的任务的数量
- Worker pending tasks：当前 worker 中，每秒钟 pending 和 running 的任务的数量，正常情况下，应该小于 1000
- FuturePool handled tasks：future pool 每秒钟处理的任务的数量
- FuturePool pending tasks：当前 future pool 中，每秒钟 pending 和 running 的任务的数量

12.8.4.23 Coprocessor Overview

- Request duration：从收到 coprocessor 请求到处理结束所消耗的总时间
- Total Requests：每种类型的总请求的 ops
- Handle duration：每分钟实际处理 coprocessor 请求所消耗的时间的直方图
- Total Request Errors：Coprocessor 每秒请求错误的数量，正常情况下，短时间内不应该有大量的错误
- Total KV Cursor Operations：各种类型的 KV cursor 操作的总数量的 ops，例如 select、index、analyze_table、analyze_index、checksum_table、checksum_index 等
- KV Cursor Operations：每秒各种类型的 KV cursor 操作的数量，以直方图形式显示
- Total RocksDB Perf Statistics：RocksDB 性能统计数据
- Total Response Size：coprocessor 回应的数据大小

12.8.4.24 Coprocessor Detail

- Handle duration：每秒钟实际处理 coprocessor 请求所消耗的时间的直方图
- 95% Handle duration by store：每秒钟中 95% 的情况下，每个 TiKV 实例处理 coprocessor 请求所花费的时间
- Wait duration：coprocessor 每秒钟内请求的等待时间，99.99% 的情况下，应该小于 10s
- 95% Wait duration by store：每秒钟 95% 的情况下，每个 TiKV 实例上 coprocessor 请求的等待时间
- Total DAG Requests：DAG 请求的总数量的 ops
- Total DAG Executors：DAG executor 的总数量的 ops
- Total Ops Details (Table Scan)：coprocessor 中请求为 select 的 scan 过程中每秒钟各种事件发生的次数
- Total Ops Details (Index Scan)：coprocessor 中请求为 index 的 scan 过程中每秒钟各种事件发生的次数
- Total Ops Details by CF (Table Scan)：coprocessor 中对于每个 CF 请求为 select 的 scan 过程中每秒钟各种事件发生的次数
- Total Ops Details by CF (Index Scan)：coprocessor 中对于每个 CF 请求为 index 的 scan 过程中每秒钟各种事件发生的次数

12.8.4.25 Threads

- Threads state：TiKV 线程的状态
- Threads IO：TiKV 各个线程的 I/O 流量
- Thread Voluntary Context Switches：TiKV 线程自主切换的次数
- Thread Nonvoluntary Context Switches：TiKV 线程被动切换的次数

12.8.4.26 RocksDB - kv/raft

- Get operations：get 操作的 ops
- Get duration：get 操作的耗时
- Seek operations：seek 操作的 ops
- Seek duration：seek 操作的耗时
- Write operations：write 操作的 ops
- Write duration：write 操作的耗时
- WAL sync operations：sync WAL 操作的 ops
- Write WAL duration：write 操作中写 WAL 的耗时
- WAL sync duration：sync WAL 操作的耗时
- Compaction operations：compaction 和 flush 操作的 ops
- Compaction duration：compaction 和 flush 操作的耗时
- SST read duration：读取 SST 所需的时间
- Write stall duration：由于 write stall 造成的时间开销，正常情况下应为 0
- Memtable size：每个 CF 的 memtable 的大小
- Memtable hit：memtable 的命中率
- Block cache size：block cache 的大小。如果将 `shared block cache` 禁用，即为每个 CF 的 block cache 的大小
- Block cache hit：block cache 的命中率
- Block cache flow：不同 block cache 操作的流量
- Block cache operations 不同 block cache 操作的个数
- Keys flow：不同操作造成的 key 的流量
- Total keys：每个 CF 中 key 的个数
- Read flow：不同读操作的流量
- Bytes / Read：每次读的大小
- Write flow：不同写操作的流量
- Bytes / Write：每次写的大小
- Compaction flow：compaction 相关的流量
- Compaction pending bytes：等待 compaction 的大小
- Read amplification：每个 TiKV 实例的读放大
- Compression ratio：每一层的压缩比
- Number of snapshots：每个 TiKV 的 snapshot 的数量
- Oldest snapshots duration：最旧的 snapshot 保留的时间
- Number files at each level：每一层的文件个数
- Ingest SST duration seconds：ingest SST 所花费的时间
- Stall conditions changed of each CF：每个 CF stall 的原因

12.8.4.27 Titan - All

- Blob file count：Titan blob 文件的数量
- Blob file size：Titan blob 文件总大小
- Live blob size：有效 blob record 的总大小
- Blob cache hit：Titan 的 blob cache 命中率
- Iter touched blob file count：单个 Iterator 所涉及到 blob 文件的数量
- Blob file discardable ratio distribution：blob 文件的失效 blob record 比例的分布情况
- Blob key size：Titan 中 blob key 的大小
- Blob value size：Titan 中 blob value 的大小
- Blob get operations：blob 的 get 操作的数量
- Blob get duration：blob 的 get 操作的耗时
- Blob iter operations：blob 的 iter 操作的耗时
- Blob seek duration：blob 的 seek 操作的耗时
- Blob next duration：blob 的 next 操作的耗时
- Blob prev duration：blob 的 prev 操作的耗时
- Blob keys flow：Titan blob 读写的 key 数量
- Blob bytes flow：Titan blob 读写的 bytes 数量
- Blob file read duration：blob 文件的读取耗时
- Blob file write duration：blob 文件的写入耗时
- Blob file sync operations：blob 文件 sync 次数
- Blob file sync duration：blob 文件 sync 耗时
- Blob GC action：Titan GC 细分动作的次数
- Blob GC duration：Titan GC 的耗时
- Blob GC keys flow：Titan GC 读写的 key 数量
- Blob GC bytes flow：Titan GC 读写的 bytes 数量
- Blob GC input file size：Titan GC 输入文件的大小
- Blob GC output file size：Titan GC 输出文件的大小
- Blob GC file count：Titan GC 涉及的 blob 文件数量

12.8.4.28 Lock manager

- Thread CPU：lock manager 的线程 CPU 使用率
- Handled tasks：lock manager 处理的任务数量
- Waiter lifetime duration：事务等待锁释放的时间
- Wait table：wait table 的状态信息，包括锁的数量和等锁事务的数量
- Deadlock detect duration：处理死锁检测请求的耗时
- Detect error：死锁检测遇到的错误数量，包含死锁的数量
- Deadlock detector leader：死锁检测器 leader 所在节点的信息

12.8.4.29 Memory

- Allocator Stats：内存分配器的统计信息

12.8.4.30 Backup

- Backup CPU：backup 的线程 CPU 使用率
- Range Size：backup range 的大小直方图
- Backup Duration：backup 的耗时
- Backup Flow：backup 总的字节大小
- Disk Throughput：实例磁盘的吞吐量
- Backup Range Duration：backup range 的耗时
- Backup Errors：backup 中发生的错误数量

12.8.4.31 Encryption

- Encryption data keys：正在使用的加密 data key 的总数量
- Encrypted files：被加密的文件数量
- Encryption initialized：显示加密是否被启用，1 代表已经启用
- Encryption meta files size：加密相关的元数据文件的大小
- Encrypt/decrypt data nanos：每次加密/解密数据的耗时的直方图
- Read/write encryption meta duration：每秒钟读写加密文件所耗费的时间

12.8.4.32 面板常见参数的解释

12.8.4.32.1 gRPC 消息类型

1. 使用事务型接口的命令：

    - kv_get：事务型的 get 命令，获取指定 ts 能读到的最新版本数据
    - kv_scan：扫描连续的一段数据
    - kv_prewrite：2PC 的第一阶段，预写入事务要提交的数据
    - kv_pessimistic_lock：对 key 加悲观锁，防止其他事务修改
    - kv_pessimistic_rollback：删除 key 上的悲观锁
    - kv_txn_heart_beat：更新悲观事务或大事务的 `lock_ttl` 以防止其被回滚
    - kv_check_txn_status：检查事务的状态
    - kv_commit：2PC 的第二阶段，提交 prewrite 阶段写入的数据
    - kv_cleanup：回滚一个事务（此命令将会在 4.0 中废除）
    - kv_batch_get：与 `kv_get` 类似，一次性获取批量 key 的 value
    - kv_batch_rollback：批量回滚多个预写的事务
    - kv_scan_lock：扫描所有版本号在 `max_version` 之前的锁，用于清理过期的事务
    - kv_resolve_lock：根据事务状态，提交或回滚事务的锁
    - kv_gc：触发垃圾回收
    - kv_delete_range：从 TiKV 中删除连续的一段数据

2. 非事务型的裸命令：

    - raw_get：获取 key 所对应的 value
    - raw_batch_get：获取一批 key 所对应的 value
    - raw_scan：扫描一段连续的数据
    - raw_batch_scan：扫描多段连续的数据

- raw_put：写入一个 key/value 对
- raw_batch_put：直接写入一批 key/value 对
- raw_delete：删除一个 key/value 对
- raw_batch_delete：删除一批 key/value 对
- raw_delete_range：删除连续的一段区间

12.8.5 TiFlash 集群监控

使用 TiDB Ansible 或 TiUP 部署 TiDB 集群时，一键部署监控系统 (Prometheus & Grafana)，监控架构参见TiDB 监控框架概述。

目前 Grafana Dashboard 整体分为 PD、 TiDB、 TiKV、 Node_exporter、 Overview 等。

TiFlash 面板一共包括 TiFlash-Summary、 TiFlash-Proxy-Summary、 TiFlash-Proxy-Details。通过面板上的指标，可以了解 TiFlash 当前的状态。其中 TiFlash-Proxy-Summary、 TiFlash-Proxy-Details 主要为 TiFlash 的 Raft 层信息，其监控指标信息可参考TiKV 监控指标详解。

> 注意：
>
> 低版本的 TiFlash 监控信息较不完善，如有需要推荐使用 v4.0.5 或更高版本的 TiDB 集群。

以下为 TiFlash-Summary 默认的监控信息：

12.8.5.1 Server

- Store size：每个 TiFlash 实例的使用的存储空间的大小。
- Available size：每个 TiFlash 实例的可用的存储空间的大小。
- Capacity size：每个 TiFlash 实例的存储容量的大小。
- Uptime：自上次重启以来 TiFlash 正常运行的时间。
- Memory：每个 TiFlash 实例内存的使用情况。
- CPU Usage：每个 TiFlash 实例 CPU 的使用率。
- FSync OPS：每个 TiFlash 实例每秒进行 fsync 操作的次数。
- File Open OPS：每个 TiFlash 实例每秒进行 open 操作的次数。
- Opened File Count：当前每个 TiFlash 实例打开的文件句柄数。

> 注意：
>
> Store size、FSync OPS、File Open OPS、Opened File Count 目前仅包含了 TiFlash 存储层的统计指标，未包括 TiFlash-Proxy 内的信息。

12.8.5.2 Coprocessor

- Request QPS：所有 TiFlash 实例收到的 coprocessor 请求数量。其中 batch 是 batch 请求数量，batch_cop 是 batch 请求中的 coprocessor 请求数量，cop 是直接通过 coprocessor 接口发送的 coprocessor 请求数量，cop_dag 是所有 coprocessor 请求中 dag 请求数量，super_batch 是开启 super batch 特性的请求数量。
- Executor QPS：所有 TiFlash 实例收到的请求中，每种 dag 算子的数量，其中 table_scan 是扫表算子，selection 是过滤算子，aggregation 是聚合算子，top_n 是 TopN 算子，limit 是 limit 算子。
- Request Duration：所有 TiFlash 实例处理 coprocessor request 总时间，总时间为接收到该 coprocessor 请求至请求应答完毕的时间。
- Error QPS：所有 TiFlash 实例处理 coprocessor 请求的错误数量。其中 meet_lock 为读取的数据有锁，region_not_found 为 Region 不存在，epoch_not_match 为读取的 Region epoch 与本地不一致，kv_client_error 为与 TiKV 通信产生的错误，internal_error 为 TiFlash 内部系统错误，other 为其他错误。
- Request Handle Duration：所有 TiFlash 实例处理 coprocessor 请求处理时间，处理时间为该 coprocessor 请求开始执行到执行结束的时间。
- Response Bytes/Seconds：所有 TiFlash 实例应答总字节数。
- Cop task memory usage：所有 TiFlash 实例处理 coprocessor 请求占用的总内存。
- Handling Request Number：所有 TiFlash 实例正在处理的 coprocessor 请求数量之和。请求的分类与 Request QPS 中的相同.

12.8.5.3 DDL

- Schema Version：每个 TiFlash 实例目前缓存的 schema 版本。
- Schema Apply OPM：所有 TiFlash 实例每分钟 apply 同步 TiDB schema diff 的次数。diff apply 是正常的单次 apply 过程，如果 diff apply 失败，则 failed apply +1，并回退到 full apply，拉取最新的 schema 信息以更新 TiFlash 的 schema 版本。
- Schema Internal DDL OPM：所有 TiFlash 实例每分钟执行的内部 DDL 次数。
- Schema Apply Duration：所有 TiFlash 实例 apply schema 消耗的时间。

12.8.5.4 Storage

- Write Command OPS：所有 TiFlash 实例存储层每秒收到的写请求数量。
- Write Amplification：每个 TiFlash 实例写放大倍数（实际磁盘写入量/逻辑数据写入量）。total 为自此次启动以来的写放大倍数，5min 为最近 5 分钟内的写放大倍数。
- Read Tasks OPS：每个 TiFlash 实例每秒存储层内部读取任务的数量。
- Rough Set Filter Rate：每个 TiFlash 实例最近 1 分钟内读取的 packet 数被存储层粗糙索引过滤的比例。
- Internal Tasks OPS：所有 TiFlash 实例每秒进行内部数据整理任务的次数。
- Internal Tasks Duration：所有 TiFlash 实例进行内部数据整理任务消耗的时间。
- Page GC Tasks OPM：所有 TiFlash 实例每分钟进行 Delta 部分数据整理任务的次数。
- Page GC Tasks Duration：所有 TiFlash 实例进行 Delta 部分数据整理任务消耗的时间分布。
- Disk Write OPS：所有 TiFlash 实例每秒进行磁盘写入的次数。
- Disk Read OPS：所有 TiFlash 实例每秒进行磁盘读取的次数。
- Write flow：所有 TiFlash 实例磁盘写操作的流量。
- Read flow：所有 TiFlash 实例磁盘读操作的流量。

> **注意：**
>
> 目前这部分监控指标仅包含了 TiFlash 存储层的统计指标，未包括 TiFlash-Proxy 内的信息。

12.8.5.5 Raft

- Read Index OPS：每个 TiFlash 实例每秒触发 read_index 请求的次数，等于请求触发的 Region 总数。
- Read Index Duration：所有 TiFlash 实例在进行 read_index 消耗的时间，主要消耗在于和 Region leader 的交互和重试时间。
- Wait Index Duration：所有 TiFlash 实例在进行 wait_index 消耗的时间，即拿到 read_index 请求后，等待本地的 Region index >= read_index 所花费的时间。

12.9 安全加固

12.9.1 为 TiDB 客户端服务端间通信开启加密传输

TiDB 服务端与客户端之间默认采用非加密连接，因而具备监视信道流量能力的第三方可以知悉 TiDB 服务端与客户端之间发送和接受的数据，包括但不限于查询语句内容、查询结果等。若信道是不可信的，例如客户端是通过公网连接到 TiDB 服务端的，则非加密连接容易造成信息泄露，建议使用加密连接确保安全性。

TiDB 服务端支持启用基于 TLS（传输层安全）协议的加密连接，协议与 MySQL 加密连接一致，现有 MySQL 客户端如 MySQL 运维工具和 MySQL 驱动等能直接支持。TLS 的前身是 SSL，因而 TLS 有时也被称为 SSL，但由于 SSL 协议有已知安全漏洞，TiDB 实际上并未支持。TiDB 支持的 TLS/SSL 协议版本为 TLS 1.0、TLS 1.1、TLS 1.2、TLS 1.3。

使用加密连接后，连接将具有以下安全性质：

- 保密性：流量明文无法被窃听；
- 完整性：流量明文无法被篡改；
- 身份验证（可选）：客户端和服务端能验证双方身份，避免中间人攻击。

TiDB 的加密连接支持默认是关闭的，必须在 TiDB 服务端通过配置开启加密连接的支持后，才能在客户端中使用加密连接，要使用加密连接必须同时满足以下两个条件：

1. TiDB 服务端配置开启加密连接的支持
2. 客户端指定使用加密连接

另外，与 MySQL 一致，TiDB 加密连接是以单个连接为单位的，默认情况下是可选的。因而对于开启了加密连接支持的 TiDB 服务端，客户端既可以选择通过加密连接安全地连接到该 TiDB 服务端，也可以选择使用普通的非加密连接。如需强制要求客户端使用加密连接可以通过以下两种方式进行配置:

- 通过在启动参数中配置 `--require-secure-transport` 要求所有用户必须使用加密连接来连接到 TiDB。
- 通过在创建用户 (`create user`)，赋予权限 (`grant`) 或修改已有用户 (`alter user`) 时指定 `require ssl` 要求指定用户必须使用加密连接来连接 TiDB。以创建用户为例：

```
create user 'u1'@'%'  require ssl;
```

> 注意：
>
> 如果登录用户已配置使用TiDB 证书鉴权功能校验用户证书，也会隐式要求对应用户必须使用加密连接连接 TiDB。

12.9.1.1 配置 TiDB 启用加密连接支持

在启动 TiDB 时，至少需要在配置文件中同时指定 ssl-cert 和 ssl-key 参数，才能使 TiDB 服务端接受加密连接。还可以指定 ssl-ca 参数进行客户端身份验证（请参见配置启用身份验证章节）。

- ssl-cert：指定 SSL 证书文件路径
- ssl-key：指定证书文件对应的私钥
- ssl-ca：可选，指定受信任的 CA 证书文件路径

参数指定的文件都为 PEM 格式。另外目前 TiDB 尚不支持加载有密码保护的私钥，因此必须提供一个没有密码的私钥文件。若提供的证书或私钥无效，则 TiDB 服务端将照常启动，但并不支持客户端加密连接到 TiDB 服务端。

上述证书及密钥可以使用 OpenSSL 签发和生成，也可以使用 MySQL 自带的工具 mysql_ssl_rsa_setup 快捷生成：

```
mysql_ssl_rsa_setup --datadir=./certs
```

以上命令将在 certs 目录下生成以下文件：

```
certs
├── ca-key.pem
├── ca.pem
├── client-cert.pem
├── client-key.pem
├── private_key.pem
├── public_key.pem
├── server-cert.pem
└── server-key.pem
```

对应的 TiDB 配置文件参数为：

```
[security]
ssl-cert = "certs/server-cert.pem"
ssl-key = "certs/server-key.pem"
```

若证书参数无误，则 TiDB 在启动时将会输出 secure connection is enabled，否则 TiDB 会输出 secure connection is NOT ENABLED。

12.9.1.2 配置 MySQL 客户端使用加密连接

MySQL 5.7 及以上版本自带的客户端默认尝试使用安全连接，若服务端不支持安全连接则自动退回到使用非安全连接；MySQL 5.7 以下版本自带的客户端默认采用非安全连接。

可以通过命令行参数修改客户端的连接行为，包括：

- 强制使用安全连接，若服务端不支持安全连接则连接失败 (`--ssl-mode=REQUIRED`)
- 尝试使用安全连接，若服务端不支持安全连接则退回使用不安全连接
- 使用不安全连接 (`--ssl-mode=DISABLED`)

详细信息请参阅 MySQL 文档中关于客户端配置安全连接的部分。

12.9.1.3 配置启用身份验证

若在 TiDB 服务端或 MySQL 客户端中未指定 `ssl-ca` 参数，则默认不会进行客户端或服务端身份验证，无法抵御中间人攻击，例如客户端可能会“安全地”连接到了一个伪装的服务端。可以在服务端和客户端中配置 `ssl-ca` 参数进行身份验证。一般情况下只需验证服务端身份，但也可以验证客户端身份进一步增强安全性。

- 若要使 MySQL 客户端验证 TiDB 服务端身份，TiDB 服务端需至少配置 `ssl-cert` 和 `ssl-key` 参数，客户端需至少指定 `--ssl-ca` 参数，且 `--ssl-mode` 至少为 `VERIFY_CA`。必须确保 TiDB 服务端配置的证书（`ssl-cert`）是由客户端 `--ssl-ca` 参数所指定的 CA 所签发的，否则身份验证失败。
- 若要使 TiDB 服务端验证 MySQL 客户端身份，TiDB 服务端需配置 `ssl-cert`、`ssl-key`、`ssl-ca` 参数，客户端需至少指定 `--ssl-cert`、`--ssl-key` 参数。必须确保服务端配置的证书和客户端配置的证书都是由服务端配置指定的 `ssl-ca` 签发的。
- 若要进行双向身份验证，请同时满足上述要求。

默认情况，服务端对客户端的身份验证是可选的。若客户端在 TLS 握手时未出示自己的身份证书，也能正常建立 TLS 连接。但也可以通过在创建用户 (`create user`)，赋予权限 (`grant`) 或修改已有用户 (`alter user`) 时指定 `require x509` 要求客户端需进行身份验证，以创建用户为例：

```
create user 'u1'@'%'  require x509;
```

> 注意：
>
> 如果登录用户已配置使用TiDB 证书鉴权功能校验用户证书，也会隐式要求对应用户需进行身份验证。

12.9.1.4 检查当前连接是否是加密连接

可以通过 `SHOW STATUS LIKE "%Ssl%";` 了解当前连接的详细情况，包括是否使用了安全连接、安全连接采用的加密协议、TLS 版本号等。

以下是一个安全连接中执行该语句的结果。由于客户端支持的 TLS 版本号和加密协议会有所不同，执行结果相应地也会有所变化。

```
SHOW STATUS LIKE "%Ssl%";
```

```
......
| Ssl_verify_mode | 5                           |
| Ssl_version     | TLSv1.2                     |
| Ssl_cipher      | ECDHE-RSA-AES128-GCM-SHA256  |
......
```

除此以外，对于 MySQL 自带客户端，还可以使用 STATUS 或 \s 语句查看连接情况：

```
\s
```

```
...
SSL: Cipher in use is ECDHE-RSA-AES128-GCM-SHA256
...
```

12.9.1.5 支持的 TLS 版本及密钥交换协议和加密算法

TiDB 支持的 TLS 版本及密钥交换协议和加密算法由 Golang 官方库决定。

12.9.1.5.1 支持的 TLS 版本

- TLS 1.0
- TLS 1.1
- TLS 1.2
- TLS 1.3

12.9.1.5.2 支持的密钥交换协议及加密算法

- TLS_RSA_WITH_AES_128_CBC_SHA
- TLS_RSA_WITH_AES_256_CBC_SHA
- TLS_RSA_WITH_AES_128_CBC_SHA256
- TLS_RSA_WITH_AES_128_GCM_SHA256
- TLS_RSA_WITH_AES_256_GCM_SHA384
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
- TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
- TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256
- TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256
- TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384

- TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384
- TLS_AES_128_GCM_SHA256
- TLS_AES_256_GCM_SHA384
- TLS_CHACHA20_POLY1305_SHA256

12.9.1.6 重加载证书、密钥和 CA

在需要替换证书、密钥或 CA 时，可以在完成对应文件替换后，对运行中的 TiDB 实例执行`ALTER INSTANCE ↪ RELOAD TLS` 语句从原配置的证书 (`ssl-cert`)、密钥 (`ssl-key`) 和 CA (`ssl-ca`) 的路径重新加证书、密钥和 CA，而无需重启 TiDB 实例。

新加载的证书密钥和 CA 将在语句执行成功后对新建立的连接生效，不会影响语句执行前已建立的连接。

12.9.1.7 另请参阅

- 为 TiDB 组件间通信开启加密传输

12.9.2 为 TiDB 组件间通信开启加密传输

本部分介绍如何为 TiDB 集群内各部组件间开启加密传输，一旦开启以下组件间均将使用加密传输：

- TiDB 与 TiKV、 PD
- TiKV 与 PD
- TiDB Control 与 TiDB，TiKV Control 与 TiKV，PD Control 与 PD
- TiKV、 PD、 TiDB 各自集群内内部通讯

目前暂不支持只开启其中部分组件的加密传输。

12.9.2.1 配置开启加密传输

1. 准备证书。

   推荐为 TiDB、 TiKV、 PD 分别准备一个 Server 证书，并保证可以相互验证，而它们的 Control 工具则可选择共用 Client 证书。

   有多种工具可以生成自签名证书，如 `openssl`，`easy-rsa`，`cfssl`。

   这里提供一个使用 `openssl` 生成证书的示例：生成自签名证书。

2. 配置证书。

   - TiDB

     在 `config` 文件或命令行参数中设置：

```
[security]
# Path of file that contains list of trusted SSL CAs for connection with cluster
    ↪ components.
cluster-ssl-ca = "/path/to/ca.pem"
```

```
# Path of file that contains X509 certificate in PEM format for connection with cluster
    ↪  components.
cluster-ssl-cert = "/path/to/tidb-server.pem"
# Path of file that contains X509 key in PEM format for connection with cluster
    ↪ components.
cluster-ssl-key = "/path/to/tidb-server-key.pem"
```

- TiKV

  在 config 文件或命令行参数中设置，并设置相应的 URL 为 https：

```
[security]
# set the path for certificates. Empty string means disabling secure connectoins.
ca-path = "/path/to/ca.pem"
cert-path = "/path/to/tikv-server.pem"
key-path = "/path/to/tikv-server-key.pem"
```

- PD

  在 config 文件或命令行参数中设置，并设置相应的 URL 为 https：

```
[security]
# Path of file that contains list of trusted SSL CAs. if set, following four settings
    ↪ shouldn't be empty
cacert-path = "/path/to/ca.pem"
# Path of file that contains X509 certificate in PEM format.
cert-path = "/path/to/pd-server.pem"
# Path of file that contains X509 key in PEM format.
key-path = "/path/to/pd-server-key.pem"
```

- TiFlash（从 v4.0.5 版本开始引入）

  在 tiflash.toml 文件中设置，将 http_port 项改为 https_port:

```
[security]
# Path of file that contains list of trusted SSL CAs. if set, following four settings
    ↪ shouldn't be empty
ca_path = "/path/to/ca.pem"
# Path of file that contains X509 certificate in PEM format.
cert_path = "/path/to/tiflash-server.pem"
# Path of file that contains X509 key in PEM format.
key_path = "/path/to/tiflash-server-key.pem"
```

  在 tiflash-learner.toml 文件中设置，

```
[security]
# Sets the path for certificates. The empty string means that secure connections are
    ↪ disabled.
ca-path = "/path/to/ca.pem"
cert-path = "/path/to/tiflash-server.pem"
key-path = "/path/to/tiflash-server-key.pem"
```

- TiCDC

  在启动命令行中设置，并设置相应的 URL 为 `https`：

```
cdc server --pd=https://127.0.0.1:2379 --log-file=ticdc.log --addr=0.0.0.0:8301 --
    ↪ advertise-addr=127.0.0.1:8301 --ca=/path/to/ca.pem --cert=/path/to/ticdc-cert.
    ↪ pem --key=/path/to/ticdc-key.pem
```

此时 TiDB 集群各个组件间已开启加密传输。

> 注意：
> 若 TiDB 集群各个组件间开启加密传输后，在使用 tidb-ctl、 tikv-ctl 或 pd-ctl 工具连接集群时，需要指定 client 证书，示例：

```
./tidb-ctl -u https://127.0.0.1:10080 --ca /path/to/ca.pem --ssl-cert /path/to/client.pem --
    ↪ ssl-key /path/to/client-key.pem
```

```
./pd-ctl -u https://127.0.0.1:2379 --cacert /path/to/ca.pem --cert /path/to/client.pem --key
    ↪  /path/to/client-key.pem
```

```
./tikv-ctl --host="127.0.0.1:20160" --ca-path="/path/to/ca.pem" --cert-path="/path/to/client
    ↪ .pem" --key-path="/path/to/clinet-key.pem"
```

12.9.2.2 认证组件调用者身份

通常被调用者除了校验调用者提供的密钥、证书和 CA 有效性外，还需要校验调用方身份以防止拥有有效证书的非法访问者进行访问（例如：TiKV 只能被 TiDB 访问，需阻止拥有合法证书但非 TiDB 的其他访问者访问 TiKV）。

如希望进行组件调用者身份认证，需要在生证书时通过 Common Name 标识证书使用者身份，并在被调用者配置检查证书 Common Name 列表来检查调用者身份。

- TiDB

  在 `config` 文件或命令行参数中设置：

```
[security]
cluster-verify-cn = [
  "TiDB-Server",
  "TiKV-Control",
]
```

- TiKV

  在 `config` 文件或命令行参数中设置：

```
[security]
cert-allowed-cn = [
    "TiDB-Server", "PD-Server", "TiKV-Control", "RawKvClient1",
]
```

- PD

  在 config 文件或命令行参数中设置：

```
[security]
cert-allowed-cn = ["TiKV-Server", "TiDB-Server", "PD-Control"]
```

- TiCDC

  在启动命令行中设置：

```
cdc server --pd=https://127.0.0.1:2379 --log-file=ticdc.log --addr=0.0.0.0:8301 --advertise-
    ↪ addr=127.0.0.1:8301 --ca=/path/to/ca.pem --cert=/path/to/ticdc-cert.pem --key=/path/
    ↪ to/ticdc-key.pem --cert-allowed-cn="client1,client2"
```

- TiFlash（从 v4.0.5 版本开始引入）

  在 tiflash.toml 文件中设置：

```
[security]
cert_allowed_cn = ["TiKV-Server", "TiDB-Server"]
```

  在 tiflash-learner.toml 文件中设置：

```
[security]
cert-allowed-cn = ["PD-Server", "TiKV-Server", "TiFlash-Server"]
```

12.9.2.3 证书重加载

TiDB、PD 和 TiKV 和各种 Client 都会在每次新建相互通讯的连接时重新读取当前的证书和密钥文件内容，实现证书和密钥的重加载。目前暂不支持 CA 的重加载。

12.9.2.4 另请参阅

- 为 TiDB 客户端服务端间通信开启加密传输

12.9.3 生成自签名证书

本文档提供使用 openssl 生成自签名证书的一个示例，用户也可以根据自己的需求生成符合要求的证书和密钥。

假设实例集群拓扑如下：

Name	Host IP	Services
node1	172.16.10.11	PD1, TiDB1
node2	172.16.10.12	PD2
node3	172.16.10.13	PD3
node4	172.16.10.14	TiKV1
node5	172.16.10.15	TiKV2
node6	172.16.10.16	TiKV3

12.9.3.1 安装 OpenSSL

对于 Debian 或 Ubuntu 操作系统：

```
apt install openssl
```

对于 RedHat 或 CentOS 操作系统：

```
yum install openssl
```

也可以参考 OpenSSL 官方的下载文档 进行安装。

12.9.3.2 生成 CA 证书

CA 的作用是签发证书。实际情况中，请联系你的管理员签发证书或者使用信任的 CA 机构。CA 会管理多个证书对，这里只需生成原始的一对证书，步骤如下：

1. 生成 root 密钥：

```
openssl genrsa -out root.key 4096
```

2. 生成 root 证书：

```
openssl req -new -x509 -days 1000 -key root.key -out root.crt
```

3. 验证 root 证书：

```
openssl x509 -text -in root.crt -noout
```

12.9.3.3 签发各个组件的证书

12.9.3.3.1 集群中可能使用到的证书

- tidb certificate 由 TiDB 使用，为其他组件和客户端验证 TiDB 身份。
- tikv certificate 由 TiKV 使用，为其他组件和客户端验证 TiKV 身份。
- pd certificate 由 PD 使用，为其他组件和客户端验证 PD 身份。
- client certificate 用于 PD、 TiKV、 TiDB 验证客户端。例如 `pd-ctl`，`tikv-ctl` 等。

12.9.3.3.2 给 TiKV 实例签发证书

给 TiKV 实例签发证书的步骤如下：

1. 生成该证书对应的私钥：

```
openssl genrsa -out tikv.key 2048
```

2. 拷贝一份 OpenSSL 的配置模板文件。

   模板文件可能存在多个位置，请以实际位置为准：

```
cp /usr/lib/ssl/openssl.cnf .
```

   如果不知道实际位置，请在根目录下查找：

```
find / -name openssl.cnf
```

3. 编辑 `openssl.cnf`，在 `[ req ]` 字段下加入 `req_extensions = v3_req`，然后在 `[ v3_req ]` 字段下加入 `subjectAltName = @alt_names`。最后新建一个字段，并编辑 SAN 的信息：

```
[ alt_names ]
IP.1 = 127.0.0.1
IP.2 = 172.16.10.14
IP.3 = 172.16.10.15
IP.4 = 172.16.10.16
```

4. 保存 `openssl.cnf` 文件后，生成证书请求文件（在这一步也可以为该证书指定 Common Name，其作用是让服务端验证接入的客户端的身份，各个组件默认不会开启验证，需要在配置文件中启用该功能才生效）：

```
openssl req -new -key tikv.key -out tikv.csr -config openssl.cnf
```

5. 签发生成证书：

```
openssl x509 -req -days 365 -CA root.crt -CAkey root.key -CAcreateserial -in tikv.csr -out
    ↪ tikv.crt -extensions v3_req -extfile openssl.cnf
```

6. 验证证书携带 SAN 字段信息（可选）：

```
openssl x509 -text -in tikv.crt -noout
```

7. 确认在当前目录下得到如下文件：

```
root.crt
tikv.crt
tikv.key
```

为其它 TiDB 组件签发证书的过程类似，此文档不再赘述。

12.9.3.3.3 为客户端签发证书

为客户端签发证书的步骤如下。

1. 生成该证书对应的私钥：

```
openssl genrsa -out client.key 2048
```

2. 生成证书请求文件（在这一步也可以为该证书指定 Common Name，其作用是让服务端验证接入的客户端的身份，默认不会开启对各个组件的验证，需要在配置文件中启用该功能才生效）

```
openssl req -new -key client.key -out client.csr
```

3. 签发生成证书：

```
openssl x509 -req -days 365 -CA root.crt -CAkey root.key -CAcreateserial -in client.csr -out
    ↪ client.crt
```

12.9.4 静态加密从 v4.0.0 版本开始引入

静态加密 (encryption at rest) 即在存储数据时进行数据加密。对于数据库，静态加密功能也叫透明数据加密 (TDE)，区别于传输数据加密 (TLS) 或使用数据加密（很少使用）。SSD 驱动器、文件系统、云供应商等都可进行静态加密。但区别于这些加密方式，若 TiKV 在存储数据前就进行数据加密，攻击者则必须通过数据库的身份验证才能访问数据。例如，即使攻击者获得物理机的访问权限时，也无法通过复制磁盘上的文件来访问数据。

TiKV 从 v4.0.0 起支持静态加密，即在 CTR 模式下使用 AES 对数据文件进行透明加密。要启用静态加密，用户须提供一个加密密钥，即主密钥。可以通过 AWS Key Management Service（推荐），即 KMS，提供主密钥 (master key)，也可以指定将密钥以明文形式存储在文件中。TiKV 自动轮换 (rotate) 用于加密实际数据文件的密钥，主密钥则可以由用户手动轮换。请注意，静态加密仅加密静态数据（即磁盘上的数据），而不加密网络传输中的数据。建议 TLS 与静态加密一起使用。

同样，从 v4.0.0 起，Backup & Restore (BR) 支持对备份到 S3 的数据进行 S3 服务端加密 (SSE)。BR S3 服务端加密也支持使用用户自行创建的 AWS KMS 密钥进行加密。

12.9.4.1 注意事项及使用限制

当前版本 (v4.0.0) 的 TiKV 加密功能还有待改善，需要在启用加密前注意以下事项：

- TiDB 集群部署后，大部分用户数据都存储在 TiKV 节点上。这部分数据可使用加密功能进行加密。但有少量元数据存储在 PD 节点上（例如用作 TiKV Region 边界的二级索引键）。截至 v4.0.0，PD 尚未支持静态加密功能。建议使用存储级加密（例如文件系统加密）来保护存储在 PD 中的敏感数据。
- TiFlash 从 v4.0.5 开始支持静态加密功能，详情参阅TiFlash 静态加密。TiKV 与 v4.0.5 之前版本的 TiFlash 一起部署时，存储在 TiFlash 中的数据不会被加密。
- TiKV 当前不从核心转储 (core dumps) 中排除加密密钥和用户数据。建议在使用静态加密时禁用 TiKV 进程的核心转储。
- TiKV 使用文件的绝对路径跟踪已加密的数据文件。一旦 TiKV 节点开启了加密功能，用户就不应更改数据文件的路径配置，例如 `storage.data-dir`、`raftstore.raftdb-path`、`rocksdb.wal-dir` 和 `raftdb.wal-dir`。
- TiKV 信息日志包含用于调试的用户数据。信息日志不会被加密。

12.9.4.2 功能概述

TiKV 当前支持的加密算法包括 AES128-CTR、AES192-CTR 和 AES256-CTR。TiKV 使用信封加密 (envelop encryption)，所以启用加密后，TiKV 使用以下两种类型的密钥：

- 主密钥 (master key)：主密钥由用户提供，用于加密 TiKV 生成的数据密钥。用户在 TiKV 外部进行主密钥的管理。
- 数据密钥 (data key)：数据密钥由 TiKV 生成，是实际用于加密的密钥。TiKV 会自动轮换数据密钥。

多个 TiKV 实例可共用一个主密钥。在生产环境中，推荐通过 AWS KMS 提供主密钥。首先通过 AWS KMS 创建用户主密钥 (CMK)，然后在配置文件中将 CMK 密钥的 ID 提供给 TiKV。TiKV 进程在运行时可以通过 IAM 角色访问 KMS CMK。如果 TiKV 无法访问 KMS CMK，TiKV 就无法启动或重新启动。详情参阅 AWS 文档中的 KMS and IAM。

用户也可以通过文件形式提供主密钥。该文件须包含一个用十六进制字符串编码的 256 位（32 字节）密钥，并以换行符结尾（即 `\n`），且不包含其他任何内容。将密钥存储在磁盘上会泄漏密钥，因此密钥文件仅适合存储在 RAM 内存的 `tempfs` 中。

数据密钥由 TiKV 生成并传递给底层存储引擎（即 RocksDB）。RocksDB 写入的所有文件，包括 SST 文件，WAL 文件和 MANIFEST 文件，均由当前数据密钥加密。TiKV 使用的其他临时文件（可能包括用户数据）也由相同的数据密钥加密。默认情况下，TiKV 每周自动轮换数据密钥，但是该时间段是可配置的。密钥轮换时，TiKV 不会重写全部现有文件来替换密钥，但如果集群的写入量恒定，则 RocksDB compaction 会将使用最新的数据密钥对数据重新加密。TiKV 跟踪密钥和加密方法，并使用密钥信息对读取的内容进行解密。

无论用户配置了哪种数据加密方法，数据密钥都使用 AES256-GCM 算法进行加密，以方便对主密钥进行验证。所以当使用文件而不是 KMS 方式指定主密钥时，主密钥必须为 256 位（32 字节）。

12.9.4.3 配置加密

要启用加密，你可以在 TiKV 的配置文件中添加加密部分：

```
[security.encryption]
data-encryption-method = aes128-ctr
data-key-rotation-period = 7d
```

`data-encryption-method` 的可选值为 `"aes128-ctr"`、`"aes192-ctr"`、`"aes256-ctr"` 和 `"plaintext"`。默认值为 `"plaintext"`，即默认不开启加密功能。`data-key-rotation-period` 指定 TiKV 轮换密钥的频率。可以为新 TiKV 群集或现有 TiKV 群集开启加密，但只有启用后写入的数据才保证被加密。要禁用加密，请在配置文件中删除 `data-encryption-method`，或将该参数值为 `"plaintext"`，然后重启 TiKV。

如果启用了加密（即 `data-encryption-method` 的值不是 `"plaintext"`），则必须指定主密钥。要使用 AWS KMS 方式指定为主密钥，请在 `[security.encryption]` 部分之后添加 `[security.encryption.master-key]` 部分：

```
[security.encryption.master-key]
type = "kms"
key-id = "0987dcba-09fe-87dc-65ba-ab0987654321"
region = "us-west-2"
endpoint = "https://kms.us-west-2.amazonaws.com"
```

`key-id` 指定 KMS CMK 的密钥 ID。`region` 为 KMS CMK 的 AWS 区域名。除非你使用非 AWS 提供的 AWS KMS 兼容服务，`endpoint` 通常无需指定。

若要使用文件方式指定主密钥，主密钥配置应如下所示：

```
[security.encryption.master-key]
type = "file"
path = "/path/to/key/file"
```

以上示例中，path 为密钥文件的路径。该文件须包含一个 256 位（32 字节）的十六进制字符串，并以换行符结尾（即 \n），且不包含其他任何内容。密钥文件示例如下：

```
3b5896b5be691006e0f71c3040a29495ddcad20b14aff61806940ebd780d3c62
```

12.9.4.4 轮换主密钥

要轮换主密钥，你必须在配置中同时指定新的主密钥和旧的主密钥，然后重启 TiKV。用 security.encryption.master-key 指定新的主密钥，用 security.encryption.previous-master-key 指定旧的主密钥。security.encryption.previous-master-key 配置的格式与 encryption.master-key 相同。重启时，TiKV 必须能同时访问旧主密钥和新主密钥，但一旦 TiKV 启动并运行，就只需访问新密钥。此后，可以将 encryption.previous-master-key 项保留在配置文件中。即使重启时，TiKV 也只会在无法使用新的主密钥解密现有数据时尝试使用旧密钥。

TiKV 当前不支持在线轮换主密钥，因此你需要重启 TiKV 进行主密钥轮换。建议对运行中的、提供在线查询的 TiKV 集群进行滚动重启。

轮换 KMS CMK 的配置示例如下：

```
[security.encryption.master-key]
type = "kms"
key-id = "50a0c603-1c6f-11e6-bb9e-3fadde80ce75"
region = "us-west-2"

[security.encryption.previous-master-key]
type = "kms"
key-id = "0987dcba-09fe-87dc-65ba-ab0987654321"
region = "us-west-2"
```

12.9.4.5 监控和调试

要监控静态加密（如果 TiKV 中部署了 Grafana 组件），可以查看 TiKV-Details -> Encryption 面板中的监控项：

- Encryption initialized：如果在 TiKV 启动期间初始化了加密，则为 1，否则为 0。进行主密钥轮换时可通过该监控项确认主密钥轮换是否已完成。
- Encryption data keys：现有数据密钥的数量。每次轮换数据密钥后，该数字都会增加 1。通过此监控指标可以监测数据密钥是否按预期轮换。
- Encrypted files：当前的加密数据文件数量。为先前未加密的群集启用加密时，将此数量与数据目录中的当前数据文件进行比较，可通过此监控指标估计已经被加密的数据量。
- Encryption meta file size：加密元数据文件的大小。
- Read/Write encryption meta duration：对用于加密的元数据进行操作带来的额外开销。

在调试方面，可使用 tikv-ctl 命令查看加密元数据（例如使用的加密方法和数据密钥列表）。该操作可能会暴露密钥，因此不推荐在生产环境中使用。详情参阅TiKV Control。

12.9.4.5.1 TiKV 版本间兼容性

为了减少 TiKV 管理加密元数据造成的 I/O 操作和互斥锁竞争引发的开销，TiKV v4.0.9 对此进行了优化。此优化可以通过 TiKV 配置文件中的 security.encryption.enable-file-dictionary-log 参数启用或关闭。此配置参数仅在 TiKV v4.0.9 或更高版本中生效。

该配置项默认开启，此时 TiKV v4.0.8 或更早期的版本无法识别加密元数据的数据格式。例如，假设你正在使用 TiKV v4.0.9 或更高版本，开启了静态加密和默认开启了 enable-file-dictionary-log 配置。如果将集群降级到 TiKV v4.0.8 或更早版本，TiKV 将无法启动，并且信息日志中会有类似如下的报错：

```
[2020/12/07 07:26:31.106 +08:00] [ERROR] [mod.rs:110] ["encryption: failed to load file
    ↪ dictionary."]
[2020/12/07 07:26:33.598 +08:00] [FATAL] [lib.rs:483] ["called `Result::unwrap()` on an `Err`
    ↪ value: Other(\"[components/encryption/src/encrypted_file/header.rs:18]: unknown version
    ↪ 2\")"]
```

为了避免上面所示错误，你可以首先将 security.encryption.enable-file-dictionary-log 设置为 false，然后启动 TiKV v4.0.9 或更高版本。TiKV 成功启动后，加密元数据的数据格式将降级为 TiKV 早期版本可以识别的格式。此时，你可再将 TiKV 集群降级到较早的版本。

12.9.4.6 BR S3 服务端加密

使用 BR 备份数据到 S3 时，若要启用 S3 服务端加密，需要传递 --s3.sse 参数并将参数值设置为 aws:kms。S3 将使用自己的 KMS 密钥进行加密。示例如下：

```
./br backup full --pd <pd-address> --storage "s3://<bucket>/<prefix>" --s3.sse aws:kms
```

若要使用用户创建和拥有的自定义 AWS KMS CMK，需另外传递 --s3.sse-kms-key-id 参数。此时，BR 进程和集群中的所有 TiKV 节点都需访问该 KMS CMK（例如，通过 AWS IAM），并且该 KMS CMK 必须与存储备份的 S3 bucket 位于同一 AWS 区域。建议通过 AWS IAM 向 BR 进程和 TiKV 节点授予对 KMS CMK 的访问权限。参见 AWS 文档中的 IAM。示例如下：

```
./br backup full --pd <pd-address> --storage "s3://<bucket>/<prefix>" --s3.region <region> --s3.
    ↪ sse aws:kms --s3.sse-kms-key-id 0987dcba-09fe-87dc-65ba-ab0987654321
```

恢复备份时，不需要也不可指定 --s3.sse 和 --s3.sse-kms-key-id 参数。S3 将自动相应进行解密。用于恢复备份数据的 BR 进程和集群中的 TiKV 节点也需要访问 KMS CMK，否则恢复将失败。示例如下：

```
./br restore full --pd <pd-address> --storage "s3://<bucket>/<prefix> --s3.region <region>"
```

12.9.4.7 TiFlash 静态加密从 v4.0.5 版本开始引入

TiFlash 从 v4.0.5 起支持静态加密。数据密钥由 TiFlash 生成。TiFlash（包括 TiFlash Proxy）写入的所有文件，包括数据文件、Schema 文件、临时文件等，均由当前数据密钥加密。TiFlash 支持的加密算法、加密配置方法（配置项在 tiflash-learner.toml 中）和监控项含义等均与 TiKV 一致。

如果 TiFlash 中部署了 Grafana 组件，可以查看 TiFlash-Proxy-Details -> Encryption。

12.9.5 日志脱敏

TiDB 在提供详细的日志信息时，可能会把数据库敏感的数据（例如用户数据）打印出来，造成数据安全方面的风险。因此 TiDB、TiKV 等组件各提供了一个配置项开关，开关打开后，会隐藏日志中包含的用户数据值。

12.9.5.1 TiDB 组件日志脱敏

TiDB 侧的日志脱敏需要将`global.tidb_redact_log` 的值设为 1。该变量值默认为 0，即关闭脱敏。

可以通过 set 语法，设置全局系统变量 tidb_redact_log，示例如下：

```
set @@global.tidb_redact_log=1;
```

设置后，所有新 session 产生的日志都会脱敏：

```
create table t (a int, unique key (a));
Query OK, 0 rows affected (0.00 sec)

insert into t values (1),(1);
ERROR 1062 (23000): Duplicate entry '1' for key 'a'
```

打印出的错误日志如下：

```
[2020/10/20 11:45:49.539 +08:00] [INFO] [conn.go:800] ["command dispatched failed"] [conn=5] [
    ↪ connInfo="id:5, addr:127.0.0.1:57222 status:10, collation:utf8_general_ci,  user:root"] [
    ↪ command=Query] [status="inTxn:0, autocommit:1"] [sql="insert into t values ( ? ) , ( ? )
    ↪ "] [txn_mode=OPTIMISTIC] [err="[kv:1062]Duplicate entry '?' for key 'a'"]
```

从以上报错日志可以看到，开启 tidb_redact_log 后，报错信息里的敏感内容被隐藏掉了（目前是用问号替代）。日志里中也会把敏感信息隐藏掉，以此规避数据安全风险。

12.9.5.2 TiKV 组件日志脱敏

TiKV 侧的日志脱敏需要将`security.redact-info-log` 的值设为 true。该配置项值默认为 false，即关闭脱敏。

12.9.5.3 PD 组件日志脱敏

PD 侧的日志脱敏需要将`security.redact-info-log` 的值设为 true。该配置项值默认为 false，即关闭脱敏。

12.9.5.4 TiFlash 组件日志脱敏

TiFlash 侧的日志脱敏需要将 tiflash-server 中`security.redact_info_log` 配置项的值以及 tiflash-learner 中`security ↪ .redact-info-log` 配置项的值均设为 true。两配置项默认值均为 false，即关闭脱敏。

12.10 权限

12.10.1 与 MySQL 安全特性差异

除以下功能外，TiDB 支持与 MySQL 5.7 类似的安全特性。

- 仅支持 `mysql_native_password` 密码验证或证书验证登录方案。
- 不支持外部身份验证方式（如 LDAP）。
- 不支持列级别权限设置。
- 不支持密码过期，最后一次密码变更记录以及密码生存期。#9709
- 不支持权限属性 `max_questions`，`max_updated`，`max_connections` 以及 `max_user_connections`。
- 不支持密码验证。#9741

12.10.2 权限管理

TiDB 的权限管理系统按照 MySQL 的权限管理进行实现，TiDB 支持大部分的 MySQL 的语法和权限类型。

本文档主要介绍 TiDB 权限相关操作、各项操作需要的权限以及权限系统的实现。

12.10.2.1 权限相关操作

12.10.2.1.1 授予权限

授予 xxx 用户对数据库 `test` 的读权限：

```
GRANT SELECT ON test.* TO 'xxx'@'%';
```

为 xxx 用户授予所有数据库，全部权限：

```
GRANT ALL PRIVILEGES ON *.* TO 'xxx'@'%';
```

GRANT 为一个不存在的用户授予权限时，默认并不会自动创建用户。该行为受 SQL Mode 中的 NO_AUTO_CREATE_USER 控制。如果从 SQL Mode 中去掉 NO_AUTO_CREATE_USER，当 GRANT 的目标用户不存在时，TiDB 会自动创建用户。

```
select @@sql_mode;
```

```
| @@sql_mode
    ↪
    ↪ |
+-------------------------------------------------------------------------------------------------------------------------
    ↪
| ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION_BY_ZERO,
    ↪ NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION |
+-------------------------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

```
set @@sql_mode='ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,
    ↪ ERROR_FOR_DIVISION_BY_ZERO,NO_ENGINE_SUBSTITUTION';
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
SELECT * FROM mysql.user WHERE user='xxxx';
```

```
Empty set (0.00 sec)
```

```
GRANT ALL PRIVILEGES ON test.* TO 'xxxx'@'%' IDENTIFIED BY 'yyyyy';
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
SELECT user,host FROM mysql.user WHERE user='xxxx';
```

```
+------|------+
| user | host |
+------|------+
| xxxx | %    |
+------|------+
1 row in set (0.00 sec)
```

上述示例中，`xxxx@%` 即自动添加的用户。

`GRANT` 还可以模糊匹配地授予用户数据库的权限：

```
GRANT ALL PRIVILEGES ON `te%`.* TO genius;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
SELECT user,host,db FROM mysql.db WHERE user='genius';
```

```
+--------|------|-----+
| user   | host | db  |
+--------|------|-----+
| genius | %    | te% |
+--------|------|-----+
1 row in set (0.00 sec)
```

这个例子中通过 `%` 模糊匹配，所有 `te` 开头的数据库，都被授予了权限。

12.10.2.1.2 收回权限

REVOKE 语句与 GRANT 对应：

```
REVOKE ALL PRIVILEGES ON `test`.* FROM 'genius'@'localhost';
```

> 注意：
>
> REVOKE 收回权限时只做精确匹配，若找不到记录则报错。而 GRANT 授予权限时可以使用模糊匹配。

```
REVOKE ALL PRIVILEGES ON `te%`.* FROM 'genius'@'%';
```

```
ERROR 1141 (42000): There is no such grant defined for user 'genius' on host '%'
```

关于模糊匹配和转义，字符串和 identifier：

```
GRANT ALL PRIVILEGES ON `te\%`.* TO 'genius'@'localhost';
```

```
Query OK, 0 rows affected (0.00 sec)
```

上述例子是精确匹配名为 te% 的数据库，注意使用 \ 转义字符。

以单引号包含的部分，是一个字符串。以反引号包含的部分，是一个 identifier。注意下面的区别：

```
GRANT ALL PRIVILEGES ON 'test'.* TO 'genius'@'localhost';
```

```
ERROR 1064 (42000): You have an error in your SQL syntax; check the
manual that corresponds to your MySQL server version for the right
syntax to use near ''test'.* to 'genius'@'localhost'' at line 1
```

```
GRANT ALL PRIVILEGES ON `test`.* TO 'genius'@'localhost';
```

```
Query OK, 0 rows affected (0.00 sec)
```

如果想将一些特殊的关键字做为表名，可以用反引号包含起来。比如：

```
CREATE TABLE `select` (id int);
```

```
Query OK, 0 rows affected (0.27 sec)
```

12.10.2.1.3 查看为用户分配的权限

SHOW GRANTS 语句可以查看为用户分配了哪些权限。例如：

查看当前用户的权限：

```
SHOW GRANTS;
```

```
+-------------------------------------------------------------+
| Grants for User                                             |
+-------------------------------------------------------------+
| GRANT ALL PRIVILEGES ON *.* TO 'root'@'%' WITH GRANT OPTION |
+-------------------------------------------------------------+
```

或者：

```
SHOW GRANTS FOR CURRENT_USER();
```

查看某个特定用户的权限：

```
SHOW GRANTS FOR 'user'@'host';
```

例如，创建一个用户 rw_user@192.168.% 并为其授予 test.write_table 表的写权限，和全局读权限。

```
CREATE USER `rw_user`@`192.168.%`;
GRANT SELECT ON *.* TO `rw_user`@`192.168.%`;
GRANT INSERT, UPDATE ON `test`.`write_table` TO `rw_user`@`192.168.%`;
```

查看用户 rw_user@192.168.% 的权限。

```
SHOW GRANTS FOR `rw_user`@`192.168.%`;
```

```
+---------------------------------------------------------------------+
| Grants for rw_user@192.168.%                                        |
+---------------------------------------------------------------------+
| GRANT Select ON *.* TO 'rw_user'@'192.168.%'                        |
| GRANT Insert,Update ON test.write_table TO 'rw_user'@'192.168.%' |
+---------------------------------------------------------------------+
```

12.10.2.2 TiDB 各操作需要的权限

TiDB 用户目前拥有的权限可以在 INFORMATION_SCHEMA.USER_PRIVILEGES 表中查找到。

权限类型	权限变量名	权限简述
ALL	AllPriv	所有权限
Drop	DropPriv	删除 schema/table
Index	IndexPriv	创建/删除 index
Alter	AlterPriv	执行 ALTER 语句

权限类型	权限变量名	权限简述
Super	SuperPriv	所有权限
Create	CreatePriv	创建 schema/table
Select	SelectPriv	读取表内容
Insert	InsertPriv	插入数据到表
Update	UpdatePriv	更新表中数据
Delete	DeletePriv	删除表中数据
Reload	ReloadPriv	执行 FLUSH 语句
Config	ConfigPriv	动态加载配置
Trigger	TriggerPriv	尚未使用
Process	ProcessPriv	显示正在运行的任务
Execute	ExecutePriv	执行 execute 语句
Drop Role	DropRolePriv	执行 drop role
Show View	ShowViewPriv	执行 show create view
References	ReferencesPriv	尚未使用
Create View	CreateViewPriv	创建视图
Create User	CreateUserPriv	创建用户
Create Role	CreateRolePriv	执行 create role
Show Databases	ShowDBPriv	显示 database 内的表情况

12.10.2.2.1 ALTER

- 对于所有的 ALTER 语句，均需要用户对所操作的表拥有 ALTER 权限。
- 除 ALTER...DROP 和 ALTER...RENAME TO 外，均需要对所操作表拥有 INSERT 和 CREATE 权限。
- 对于 ALTER...DROP 语句，需要对表拥有 DROP 权限。
- 对于 ALTER...RENAME TO 语句，需要对重命名前的表拥有 DROP 权限，对重命名后的表拥有 CREATE 和 INSERT 权限。

> **注意：**
>
> 根据 MySQL 5.7 文档中的说明，对表进行 ALTER 操作需要 INSERT 和 CREATE 权限，但在 MySQL 5.7.25 版本实际情况中，该操作仅需要 ALTER 权限。目前，TiDB 中的 ALTER 权限与 MySQL 实际行为保持一致。

12.10.2.2.2 CREATE DATABASE

需要拥有全局 CREATE 权限。

12.10.2.2.3 CREATE INDEX

需要对所操作的表拥有 INDEX 权限。

12.10.2.2.4 CREATE TABLE

需要对要创建的表所在的数据库拥有 CREATE 权限；若使用 CREATE TABLE...LIKE... 需要对相关的表拥有 SELECT 权限。

12.10.2.2.5 CREATE VIEW

需要拥有 CREATE VIEW 权限。

> 注意：
> 如果当前登录用户与创建视图的用户不同，除需要 CREATE VIEW 权限外，还需要 SUPER 权限。

12.10.2.2.6 DROP DATABASE

需要对数据库拥有 DROP 权限。

12.10.2.2.7 DROP INDEX

需要对所操作的表拥有 INDEX 权限。

12.10.2.2.8 DROP TABLES

需要对所操作的表拥有 DROP 权限。

12.10.2.2.9 LOAD DATA

LOAD DATA 需要对所操作的表拥有 INSERT 权限。

12.10.2.2.10 TRUNCATE TABLE

需要对所操作的表拥有 DROP 权限。

12.10.2.2.11 RENAME TABLE

需要对重命名前的表拥有 ALTER 和 DROP 权限，对重命名后的表拥有 CREATE 和 INSERT 权限。

12.10.2.2.12 ANALYZE TABLE

需要对所操作的表拥有 INSERT 和 SELECT 权限。

12.10.2.2.13 SHOW

SHOW CREATE TABLE 需要任意一种权限。

SHOW CREATE VIEW 需要 SHOW VIEW 权限。

SHOW GRANTS 需要拥有对 mysql 数据库的 SELECT 权限。如果是使用 SHOW GRANTS 查看当前用户权限，则不需要任何权限。

12.10.2.2.14 CREATE ROLE/USER

CREATE ROLE 需要 CREATE ROLE 权限。

CREATE USER 需要 CREATE USER 权限

12.10.2.2.15 DROP ROLE/USER

DROP ROLE 需要 DROP ROLE 权限。

DROP USER 需要 CREATE USER 权限

12.10.2.2.16 ALTER USER

ALTER USER 需要 CREATE USER 权限。

12.10.2.2.17 GRANT

GRANT 需要 GRANT 权限并且拥有 GRANT 所赋予的权限。

如果在 GRANTS 语句中创建用户，需要有 CREATE USER 权限。

GRANT ROLE 操作需要拥有 SUPER 权限。

12.10.2.2.18 REVOKE

REVOKE 需要 GRANT 权限并且拥有 REVOKE 所指定要撤销的权限。

REVOKE ROLE 操作需要拥有 SUPER 权限。

12.10.2.2.19 SET GLOBAL

使用 SET GLOBAL 设置全局变量需要拥有 SUPER 权限。

12.10.2.2.20 ADMIN

需要拥有 SUPER 权限。

12.10.2.2.21 SET DEFAULT ROLE

需要拥有 SUPER 权限。

12.10.2.2.22 KILL

使用 KILL 终止其他用户的会话需要拥有 SUPER 权限。

12.10.2.3 权限系统的实现

12.10.2.3.1 授权表

以下几张系统表是非常特殊的表，权限相关的数据全部存储在这几张表内。

- mysql.user：用户账户，全局权限
- mysql.db：数据库级别的权限
- mysql.tables_priv：表级别的权限
- mysql.columns_priv：列级别的权限，当前暂不支持

这几张表包含了数据的生效范围和权限信息。例如，mysql.user 表的部分数据：

```
SELECT User,Host,Select_priv,Insert_priv FROM mysql.user LIMIT 1;
```

```
+------|------|-------------|-------------+
| User | Host | Select_priv | Insert_priv |
+------|------|-------------|-------------+
| root | %    | Y           | Y           |
+------|------|-------------|-------------+
1 row in set (0.00 sec)
```

这条记录中，Host 和 User 决定了 root 用户从任意主机 (%) 发送过来的连接请求可以被接受，而 Select_priv 和 Insert_priv 表示用户拥有全局的 Select 和 Insert 权限。mysql.user 这张表里面的生效范围是全局的。

mysql.db 表里面包含的 Host 和 User 决定了用户可以访问哪些数据库，权限列的生效范围是数据库。

理论上，所有权限管理相关的操作，都可以通过直接对授权表的 CRUD 操作完成。

实现层面其实也只是包装了一层语法糖。例如删除用户会执行：

```
DELETE FROM mysql.user WHERE user='test';
```

但是，不推荐手动修改授权表，建议使用 DROP USER 语句：

```
DROP USER 'test';
```

12.10.2.3.2 连接验证

当客户端发送连接请求时，TiDB 服务器会对登录操作进行验证。验证过程先检查 mysql.user 表，当某条记录的 User 和 Host 和连接请求匹配上了，再去验证 Password。用户身份基于两部分信息，发起连接的客户端的 Host，以及用户名 User。如果 User 不为空，则用户名必须精确匹配。

User+Host 可能会匹配 user 表里面多行，为了处理这种情况，user 表的行是排序过的，客户端连接时会依次去匹配，并使用首次匹配到的那一行做权限验证。排序是按 Host 在前，User 在后。

12.10.2.3.3 请求验证

连接成功之后，请求验证会检测执行操作是否拥有足够的权限。

对于数据库相关请求 (INSERT，UPDATE)，先检查 mysql.user 表里面的用户全局权限，如果权限够，则直接可以访问。如果全局权限不足，则再检查 mysql.db 表。

user 表的权限是全局的，并且不管默认数据库是哪一个。比如 user 里面有 DELETE 权限，任何一行，任何的表，任何的数据库。

db表里面，User 为空是匹配匿名用户，User 里面不能有通配符。Host 和 Db 列里面可以有 % 和 _，可以模式匹配。

user 和 db 读到内存也是排序的。

tables_priv 和 columns_priv 中使用 % 是类似的，但是在Db, Table_name, Column_name 这些列不能包含 %。加载进来时排序也是类似的。

12.10.2.3.4 生效时机

TiDB 启动时，将一些权限检查的表加载到内存，之后使用缓存的数据来验证权限。系统会周期性的将授权表从数据库同步到缓存，生效则是由同步的周期决定，目前这个值设定的是 5 分钟。

修改了授权表，如果需要立即生效，可以手动调用：

```
FLUSH PRIVILEGES;
```

12.10.3 TiDB 用户账户管理

本文档主要介绍如何管理 TiDB 用户账户。

12.10.3.1 用户名和密码

TiDB 将用户账户存储在 mysql.user 系统表里面。每个账户由用户名和 host 作为标识。每个账户可以设置一个密码。

通过 MySQL 客户端连接到 TiDB 服务器，通过指定的账户和密码登录：

```
mysql --port 4000 --user xxx --password
```

使用缩写的命令行参数则是：

```
mysql -P 4000 -u xxx -p
```

12.10.3.2 添加用户

添加用户有两种方式：

- 通过标准的用户管理的 SQL 语句创建用户以及授予权限，比如 CREATE USER 和 GRANT。
- 直接通过 INSERT、UPDATE 和 DELETE 操作授权表。

推荐使用第一种方式。第二种方式修改容易导致一些不完整的修改，因此不推荐。还有另一种可选方式是使用第三方工具的图形化界面工具。

```
CREATE USER [IF NOT EXISTS] user [IDENTIFIED BY 'auth_string'];
```

设置登录密码后，auth_string 会被 TiDB 经过加密存储在 mysql.user 表中。

```
CREATE USER 'test'@'127.0.0.1' IDENTIFIED BY 'xxx';
```

TiDB 的用户账户名由一个用户名和一个主机名组成。账户名的语法为 'user_name'@'host_name'。

- user_name 大小写敏感。
- host_name 可以是一个主机名或 IP 地址。主机名或 IP 地址中允许使用通配符 % 和 _。例如，名为 '%' 的主机名可以匹配所有主机，'192.168.1.%' 可以匹配子网中的所有主机。

host 支持模糊匹配，比如：

```
CREATE USER 'test'@'192.168.10.%';
```

允许 test 用户从 192.168.10 子网的任何一个主机登录。

如果没有指定 host，则默认是所有 IP 均可登录。如果没有指定密码，默认为空：

```
CREATE USER 'test';
```

等价于：

```
CREATE USER 'test'@'%' IDENTIFIED BY '';
```

为一个不存在的用户授权时，是否会自动创建用户的行为受 sql_mode 影响。如果 sql_mode 中包含 NO_AUTO_CREATE_USER，则 GRANT 不会自动创建用户并报错。

假设 sql_mode 不包含 NO_AUTO_CREATE_USER，下面的例子用 CREATE USER 和 GRANT 语句创建了四个账户：

```
CREATE USER 'finley'@'localhost' IDENTIFIED BY 'some_pass';
```

```
GRANT ALL PRIVILEGES ON *.* TO 'finley'@'localhost' WITH GRANT OPTION;
```

```
CREATE USER 'finley'@'%' IDENTIFIED BY 'some_pass';
```

```
GRANT ALL PRIVILEGES ON *.* TO 'finley'@'%' WITH GRANT OPTION;
```

```
CREATE USER 'admin'@'localhost' IDENTIFIED BY 'admin_pass';
```

```
GRANT RELOAD,PROCESS ON *.* TO 'admin'@'localhost';
```

```
CREATE USER 'dummy'@'localhost';
```

使用 SHOW GRANTS 可以看到为一个用户授予的权限：

```
SHOW GRANTS FOR 'admin'@'localhost';
```

```
+-----------------------------------------------------+
| Grants for admin@localhost                          |
+-----------------------------------------------------+
| GRANT RELOAD, PROCESS ON *.* TO 'admin'@'localhost' |
+-----------------------------------------------------+
```

12.10.3.3 删除用户

使用 DROP USER 语句可以删除用户，例如：

```
DROP USER 'test'@'localhost';
```

这个操作会清除用户在 mysql.user 表里面的记录项，并且清除在授权表里面的相关记录。

12.10.3.4 保留用户账户

TiDB 在数据库初始化时会生成一个 'root'@'%' 的默认账户。

12.10.3.5 设置资源限制

暂不支持。

12.10.3.6 设置密码

TiDB 将密码存在 mysql.user 系统数据库里面。只有拥有 CREATE USER 权限，或者拥有 mysql 数据库权限（INSERT 权限用于创建，UPDATE 权限用于更新）的用户才能够设置或修改密码。

- 在 CREATE USER 创建用户时通过 IDENTIFIED BY 指定密码：

  ```
  CREATE USER 'test'@'localhost' IDENTIFIED BY 'mypass';
  ```

- 为一个已存在的账户修改密码，可以通过 SET PASSWORD FOR 或者 ALTER USER 语句完成：

  ```
  SET PASSWORD FOR 'root'@'%' = 'xxx';
  ```

  或者：

  ```
  ALTER USER 'test'@'localhost' IDENTIFIED BY 'mypass';
  ```

12.10.3.7 忘记 root 密码

1. 修改配置文件，在 security 部分添加 skip-grant-table：

   ```
   [security]
   skip-grant-table = true
   ```

2. 使用修改之后的配置启动 TiDB，然后使用 root 登录后修改密码：

   ```
   mysql -h 127.0.0.1 -P 4000 -u root
   ```

设置 skip-grant-table 之后，启动 TiDB 进程会增加操作系统用户检查，只有操作系统的 root 用户才能启动 TiDB 进程。

12.10.3.8 FLUSH PRIVILEGES

用户以及权限相关的信息都存储在 TiKV 服务器中，TiDB 在进程内部会缓存这些信息。一般通过 CREATE USER，GRANT 等语句来修改相关信息时，可在整个集群迅速生效。如果遇到网络或者其它因素影响，由于 TiDB 会周期性地更新缓存信息，正常情况下，最多 15 分钟左右生效。

如果授权表已被直接修改，则不会通知 TiDB 节点更新缓存，运行如下命令可使改动立即生效：

```
FLUSH PRIVILEGES;
```

详情参见权限管理。

12.10.4 基于角色的访问控制

TiDB 的基于角色的访问控制 (RBAC) 系统的实现类似于 MySQL 8.0 的 RBAC 系统。TiDB 兼容大部分 MySQL RBAC 系统的语法。

本文档主要介绍 TiDB 基于角色的访问控制相关操作及实现。

12.10.4.1 角色访问控制相关操作

角色是一系列权限的集合。用户可以创建角色、删除角色、将权限赋予角色；也可以将角色授予给其他用户，被授予的用户在启用角色后，可以得到角色所包含的权限。

12.10.4.1.1 创建角色

创建角色 app_developer，app_read 和 app_write：

```
CREATE ROLE 'app_developer', 'app_read', 'app_write';
```

角色名的格式和规范可以参考TiDB 用户账户管理。

角色会被保存在 mysql.user 表中，角色名称的主机名部分（如果省略）默认为 '%'。如果表中有同名角色或用户，角色会创建失败并报错。创建角色的用户需要拥有 CREATE ROLE 或 CREATE USER 权限。

12.10.4.1.2 授予角色权限

为角色授予权限和为用户授予权限操作相同，可参考TiDB 权限管理。

为 app_read 角色授予数据库 app_db 的读权限：

```
GRANT SELECT ON app_db.* TO 'app_read'@'%';
```

为 app_write 角色授予数据库 app_db 的写权限：

```
GRANT INSERT, UPDATE, DELETE ON app_db.* TO 'app_write'@'%';
```

为 app_developer 角色授予 app_db 数据库的全部权限：

```
GRANT ALL ON app_db.* TO 'app_developer';
```

12.10.4.1.3 将角色授予给用户

假设有一个用户拥有开发者角色，可以对 app_db 的所有操作权限；另外有两个用户拥有 app_db 的只读权限；还有一个用户拥有 app_db 的读写权限。

首先用 CREATE USER 来创建用户。

```
CREATE USER 'dev1'@'localhost' IDENTIFIED BY 'dev1pass';
CREATE USER 'read_user1'@'localhost' IDENTIFIED BY 'read_user1pass';
CREATE USER 'read_user2'@'localhost' IDENTIFIED BY 'read_user2pass';
CREATE USER 'rw_user1'@'localhost' IDENTIFIED BY 'rw_user1pass';
```

然后使用 GRANT 授予用户对应的角色。

```
GRANT 'app_developer' TO 'dev1'@'localhost';
GRANT 'app_read' TO 'read_user1'@'localhost', 'read_user2'@'localhost';
GRANT 'app_read', 'app_write' TO 'rw_user1'@'localhost';
```

用户执行将角色授予给其他用户或者收回角色的命令，需要用户拥有 SUPER 权限。将角色授予给用户时并不会启用该角色，启用角色需要额外的操作。

以下操作可能会形成一个“关系环”：

```
CREATE USER 'u1', 'u2';
CREATE ROLE 'r1', 'r2';

GRANT 'u1' TO 'u1';
GRANT 'r1' TO 'r1';

GRANT 'r2' TO 'u2';
GRANT 'u2' TO 'r2';
```

TiDB 允许这种多层授权关系存在，可以使用多层授权关系实现权限继承。

12.10.4.1.4 查看角色拥有的权限

可以通过 SHOW GRANTS 语句查看用户被授予了哪些角色。当用户查看其他用户权限相关信息时，需要对 mysql 数据库拥有 SELECT 权限。

```
SHOW GRANTS FOR 'dev1'@'localhost';
```

```
+-------------------------------------------------+
| Grants for dev1@localhost                       |
+-------------------------------------------------+
| GRANT USAGE ON *.* TO `dev1`@`localhost`        |
| GRANT `app_developer`@`%` TO `dev1`@`localhost` |
+-------------------------------------------------+
```

可以通过使用 SHOW GRANTS 的 USING 选项来查看角色对应的权限。

```
SHOW GRANTS FOR 'dev1'@'localhost' USING 'app_developer';
```

```
+-----------------------------------------------------------+
| Grants for dev1@localhost                                 |
+-----------------------------------------------------------+
| GRANT USAGE ON *.* TO `dev1`@`localhost`                  |
| GRANT ALL PRIVILEGES ON `app_db`.* TO `dev1`@`localhost` |
| GRANT `app_developer`@`%` TO `dev1`@`localhost`           |
+-----------------------------------------------------------+
```

```
SHOW GRANTS FOR 'rw_user1'@'localhost' USING 'app_read', 'app_write';
```

```
+-------------------------------------------------------------------------------+
| Grants for rw_user1@localhost                                                 |
+-------------------------------------------------------------------------------+
| GRANT USAGE ON *.* TO `rw_user1`@`localhost`                                  |
| GRANT SELECT, INSERT, UPDATE, DELETE ON `app_db`.* TO `rw_user1`@`localhost` |
| GRANT `app_read`@`%`,`app_write`@`%` TO `rw_user1`@`localhost`                |
+-------------------------------------------------------------------------------+
```

```
SHOW GRANTS FOR 'read_user1'@'localhost' USING 'app_read';
```

```
+-----------------------------------------------------------+
| Grants for read_user1@localhost                           |
+-----------------------------------------------------------+
| GRANT USAGE ON *.* TO `read_user1`@`localhost`            |
| GRANT SELECT ON `app_db`.* TO `read_user1`@`localhost` |
| GRANT `app_read`@`%` TO `read_user1`@`localhost`          |
+-----------------------------------------------------------+
```

可以使用 SHOW GRANTS 或 SHOW GRANTS FOR CURRENT_USER() 查看当前用户的权限。这两个语句有细微的差异，SHOW GRANTS 会显示当前用户的启用角色的权限，而 SHOW GRANTS FOR CURRENT_USER() 则不会显示启用角色的权限。

12.10.4.1.5 设置默认启用角色

角色在授予给用户之后，并不会生效；只有在用户启用了某些角色之后，才可以使用角色拥有的权限。

可以对用户设置默认启用的角色；用户在登录时，默认启用的角色会被自动启用。

```
SET DEFAULT ROLE
    {NONE | ALL | role [, role ] ...}
    TO user [, user ]
```

比如将 app_read 和 app_wirte 设置为 rw_user1@localhost 的默认启用角色：

```
SET DEFAULT ROLE app_read, app_write TO 'rw_user1'@'localhost';
```

将 dev1@localhost 的所有角色，设为其默认启用角色：

```
SET DEFAULT ROLE ALL TO 'dev1'@'localhost';
```

关闭 dev1@localhost 的所有默认启用角色：

```
SET DEFAULT ROLE NONE TO 'dev1'@'localhost';
```

需要注意的是，设置为默认启用角色的角色必须已经授予给那个用户。

12.10.4.1.6 在当前 session 启用角色

除了使用 SET DEFAULT ROLE 启用角色外，TiDB 还提供让用户在当前 session 启用某些角色的功能。

```
SET ROLE {
    DEFAULT
  | NONE
  | ALL
  | ALL EXCEPT role [, role ] ...
  | role [, role ] ...
}
```

例如，登录 rw_user1 后，为当前用户启用角色 app_read 和 app_write ，仅在当前 session 有效：

```
SET ROLE 'app_read', 'app_write';
```

启用当前用户的默认角色：

```
SET ROLE DEFAULT
```

启用授予给当前用户的所有角色：

```
SET ROLE ALL
```

不启用任何角色：

```
SET ROLE NONE
```

启用除 app_read 外的角色：

```
SET ROLE ALL EXCEPT 'app_read'
```

> 注意：
>
> 使用 SET ROLE 启用的角色只有在当前 session 才会有效。

12.10.4.1.7 查看当前启用角色

当前用户可以通过 CURRENT_ROLE() 函数查看当前用户启用了哪些角色。

例如，先对 rw_user1'@'localhost 设置默认角色：

```
SET DEFAULT ROLE ALL TO 'rw_user1'@'localhost';
```

用 rw_user1@localhost 登录后：

```
SELECT CURRENT_ROLE();
```

```
+--------------------------------+
| CURRENT_ROLE()                 |
+--------------------------------+
| `app_read`@`%`,`app_write`@`%` |
+--------------------------------+
```

```
SET ROLE 'app_read'; SELECT CURRENT_ROLE();
```

```
+----------------+
| CURRENT_ROLE() |
+----------------+
| `app_read`@`%` |
+----------------+
```

12.10.4.1.8 收回角色

解除角色 app_read 与用户 read_user1@localhost、 read_user2@localhost 的授权关系。

```
REVOKE 'app_read' FROM 'read_user1'@'localhost', 'read_user2'@'localhost';
```

解除角色 app_read、 app_write 与用户 rw_user1@localhost 的授权关系。

```
REVOKE 'app_read', 'app_write' FROM 'rw_user1'@'localhost';
```

解除角色授权具有原子性，如果在撤销授权操作中失败会回滚。

12.10.4.1.9 收回权限

REVOKE 语句与 GRANT 对应，可以使用 REVOKE 来撤销 app_write 的权限。

```
REVOKE INSERT, UPDATE, DELETE ON app_db.* FROM 'app_write';
```

具体可参考TiDB 权限管理。

12.10.4.1.10 删除角色

删除角色 app_read 和 app_write：

```
DROP ROLE 'app_read', 'app_write';
```

这个操作会清除角色在 mysql.user 表里面的记录项，并且清除在授权表里面的相关记录，解除和其相关的授权关系。执行删除角色的用户需要拥有 DROP ROLE 或 DROP USER 权限。

12.10.4.1.11 授权表

在原有的四张系统权限表的基础上，角色访问控制引入了两张新的系统表：

- mysql.role_edges：记录角色与用户的授权关系
- mysql.default_roles：记录每个用户默认启用的角色

以下是 mysql.role_edges 所包含的数据。

```
select * from mysql.role_edges;
```

```
+-----------+-----------+---------+---------+-------------------+
| FROM_HOST | FROM_USER | TO_HOST | TO_USER | WITH_ADMIN_OPTION |
+-----------+-----------+---------+---------+-------------------+
| %         | r_1       | %       | u_1     | N                 |
+-----------+-----------+---------+---------+-------------------+
1 row in set (0.00 sec)
```

其中 FROM_HOST 和 FROM_USER 分别表示角色的主机名和用户名，TO_HOST 和 TO_USER 分别表示被授予角色的用户的主机名和用户名。

mysql.default_roles 中包含了每个用户默认启用了哪些角色。

```
select * from mysql.default_roles;
```

```
+------+------+-------------------+-------------------+
| HOST | USER | DEFAULT_ROLE_HOST | DEFAULT_ROLE_USER |
+------+------+-------------------+-------------------+
| %    | u_1  | %                 | r_1               |
| %    | u_1  | %                 | r_2               |
+------+------+-------------------+-------------------+
2 rows in set (0.00 sec)
```

HOST 和 USER 分别表示用户的主机名和用户名，DEFAULT_ROLE_HOST 和 DEFAULT_ROLE_USER 分别表示默认启用的角色的主机名和用户名。

12.10.4.1.12 其他

由于基于角色的访问控制模块和用户管理以及权限管理结合十分紧密，因此需要参考一些操作的细节：

- TiDB 权限管理
- TiDB 用户账户管理

12.10.5 TiDB 证书鉴权使用指南

TiDB 支持基于证书鉴权的登录方式。采用这种方式，TiDB 对不同用户签发证书，使用加密连接来传输数据，并在用户登录时验证证书。相比 MySQL 用户常用的用户名密码验证方式，与 MySQL 相兼容的证书鉴权方式更安全，因此越来越多的用户使用证书鉴权来代替用户名密码验证。

在 TiDB 上使用证书鉴权的登录方法，可能需要进行以下操作：

- 创建安全密钥和证书
- 配置 TiDB 和客户端使用的证书
- 配置登录时需要校验的用户证书信息
- 更新和替换证书

本文介绍了如何进行证书鉴权的上述几个操作。

12.10.5.1 创建安全密钥和证书

目前推荐使用 OpenSSL 来生成密钥和证书，生成证书的过程和为 TiDB 客户端服务端间通信开启加密传输过程类似，下面更多演示如何在证书中配置更多需校验的属性字段。

12.10.5.1.1 生成 CA 密钥和证书

1. 执行以下命令生成 CA 密钥：

```
sudo openssl genrsa 2048 > ca-key.pem
```

命令执行后输出以下结果：

```
Generating RSA private key, 2048 bit long modulus (2 primes)
.....................+++++
.............................................+++++
e is 65537 (0x010001)
```

2. 执行以下命令生成该密钥对应的证书：

```
sudo openssl req -new -x509 -nodes -days 365000 -key ca-key.pem -out ca-cert.pem
```

3. 输入证书细节信息，示例如下：

```
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:California
Locality Name (eg, city) []:San Francisco
Organization Name (eg, company) [Internet Widgits Pty Ltd]:PingCAP Inc.
Organizational Unit Name (eg, section) []:TiDB
Common Name (e.g. server FQDN or YOUR name) []:TiDB admin
Email Address []:s@pingcap.com
```

> **注意：**
>
> 以上信息中，：后的文字为用户输入的信息。

12.10.5.1.2 生成服务端密钥和证书

1. 执行以下命令生成服务端的密钥：

```
sudo openssl req -newkey rsa:2048 -days 365000 -nodes -keyout server-key.pem -out server-req
    ↪ .pem
```

2. 输入证书细节信息，示例如下：

```
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:California
Locality Name (eg, city) []:San Francisco
Organization Name (eg, company) [Internet Widgits Pty Ltd]:PingCAP Inc.
Organizational Unit Name (eg, section) []:TiKV
Common Name (e.g. server FQDN or YOUR name) []:TiKV Test Server
Email Address []:k@pingcap.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

3. 执行以下命令生成服务端的 RSA 密钥：

```
sudo openssl rsa -in server-key.pem -out server-key.pem
```

输出结果如下：

```
writing RSA key
```

4. 使用 CA 证书签名来生成服务端的证书：

```
sudo openssl x509 -req -in server-req.pem -days 365000 -CA ca-cert.pem -CAkey ca-key.pem -
    ↪ set_serial 01 -out server-cert.pem
```

输出结果示例如下：

```
Signature ok
subject=C = US, ST = California, L = San Francisco, O = PingCAP Inc., OU = TiKV, CN = TiKV
    ↪ Test Server, emailAddress = k@pingcap.com
Getting CA Private Key
```

> **注意：**
>
> 以上结果中，用户登录时 TiDB 将强制检查 `subject` 部分的信息是否一致。

12.10.5.1.3 生成客户端密钥和证书

生成服务端密钥和证书后，需要生成客户端使用的密钥和证书。通常需要为不同的用户生成不同的密钥和证书。

1. 执行以下命令生成客户端的密钥：

```
sudo openssl req -newkey rsa:2048 -days 365000 -nodes -keyout client-key.pem -out client-req
    ↪ .pem
```

2. 输入证书细节信息，示例如下：

```
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:California
Locality Name (eg, city) []:San Francisco
Organization Name (eg, company) [Internet Widgits Pty Ltd]:PingCAP Inc.
Organizational Unit Name (eg, section) []:TiDB
Common Name (e.g. server FQDN or YOUR name) []:tpch-user1
Email Address []:zz@pingcap.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

3. 执行以下命令生成客户端 RSA 证书：

```
sudo openssl rsa -in client-key.pem -out client-key.pem
```

以上命令的输出结果如下：

```
writing RSA key
```

4. 执行以下命令，使用 CA 证书签名来生成客户端证书：

```
sudo openssl x509 -req -in client-req.pem -days 365000 -CA ca-cert.pem -CAkey ca-key.pem -
    ↪ set_serial 01 -out client-cert.pem
```

输出结果示例如下：

```
Signature ok
subject=C = US, ST = California, L = San Francisco, O = PingCAP Inc., OU = TiDB, CN = tpch-
    ↪ user1, emailAddress = zz@pingcap.com
Getting CA Private Key
```

> **注意：**
>
> 以上结果中，`subject` 部分后的信息会被用来在 `require` 中配置和要求验证。

12.10.5.1.4 验证证书

执行以下命令验证证书：

```
openssl verify -CAfile ca-cert.pem server-cert.pem client-cert.pem
```

如果验证通过，会显示以下信息：

```
server-cert.pem: OK
client-cert.pem: OK
```

12.10.5.2 配置 TiDB 和客户端使用证书

在生成证书后，需要在 TiDB 中配置服务端所使用的证书，同时让客户端程序使用客户端证书。

12.10.5.2.1 配置 TiDB 服务端

修改 TiDB 配置文件中的 `[security]` 段。这一步指定 CA 证书、服务端密钥和服务端证书存放的路径。可将 `path/to/server-cert.pem`、`path/to/server-key.pem` 和 `path/to/ca-cert.pem` 替换成实际的路径。

```
[security]
ssl-cert ="path/to/server-cert.pem"
ssl-key ="path/to/server-key.pem"
ssl-ca="path/to/ca-cert.pem"
```

启动 TiDB 日志。如果日志中有以下内容，即代表配置生效：

```
[INFO] [server.go:264] ["secure connection is enabled"] ["client verification enabled"=true]
```

12.10.5.2.2 配置客户端程序

配置客户端程序，让客户端使用客户端密钥和证书来登录 TiDB。

以 MySQL 客户端为例，可以通过指定 `ssl-cert`、`ssl-key`、`ssl-ca` 来使用新的 CA 证书、客户端密钥和证书：

```
mysql -utest -h0.0.0.0 -P4000 --ssl-cert /path/to/client-cert.new.pem --ssl-key /path/to/client-
    ↪ key.new.pem --ssl-ca /path/to/ca-cert.pem
```

> **注意：**
>
> `/path/to/client-cert.new.pem`、`/path/to/client-key.new.pem` 和 `/path/to/ca-cert.pem` 是 CA 证书、客户端密钥和客户端存放的路径。可将以上命令中的这些部分替换为实际的路径。

12.10.5.3 配置登录时需要校验的用户证书信息

使用客户端连接 TiDB 进行授权配置。先获取需要验证的用户证书信息，再对这些信息进行配置。

12.10.5.3.1 获取用户证书信息

用户证书信息可由 `require subject`、`require issuer`、`require san` 和 `require cipher` 来指定，用于检查 X.509 certificate attributes。

- `require subject`：指定用户在连接时需要提供客户端证书的 `subject` 内容。指定该选项后，不需要再配置 `require ssl` 或 `x509`。配置内容对应生成客户端密钥和证书中的录入信息。

  可以执行以下命令来获取该项的信息：

```
openssl x509 -noout -subject -in client-cert.pem | sed 's/.\{8\}//'  | sed 's/, /\//g' | sed
    ↪  's/ = /=/g' | sed 's/^/\//'
```

- `require issuer`：指定签发用户证书的 CA 证书的 `subject` 内容。配置内容对应生成 CA 密钥和证书中的录入信息。

  可以执行以下命令来获取该项的信息：

```
openssl x509 -noout -subject -in ca-cert.pem | sed 's/.\{8\}//'  | sed 's/, /\//g' | sed 's/
    ↪  = /=/g' | sed 's/^/\//'
```

- `require san`：指定签发用户证书的 CA 证书的 `Subject Alternative Name` 内容。配置内容对应生成客户端证书使用的openssl.cnf 配置文件的 `alt_names` 信息。

  - 可以执行以下命令来获取已生成证书中的 `require san` 项的信息：

```
openssl x509 -noout -extensions subjectAltName -in client.crt
```

  - `require san` 目前支持以下 `Subject Alternative Name` 检查项：

    * URI
    * IP
    * DNS

  - 多个检查项可通过逗号连接后进行配置。例如，对用户 u1 进行以下配置：

```
create user 'u1'@'%' require san 'DNS:d1,URI:spiffe://example.org/myservice1,URI:spiffe
    ↪ ://example.org/myservice2'
```

    以上配置只允许用户 u1 使用 URI 项为 `spiffe://example.org/myservice1` 或 `spiffe://example.org
    ↪ /myservice2`、DNS 项为 d1 的证书登录 TiDB。

- `require cipher`：配置该项检查客户端支持的 `cipher method`。可以使用以下语句来查看支持的列表：

```
SHOW SESSION STATUS LIKE 'Ssl_cipher_list';
```

12.10.5.3.2 配置用户证书信息

获取用户证书信息（`require subject`, `require issuer`、`require san` 和 `require cipher`）后，可在创建用户、赋予权限或更改用户时配置用户证书信息。将以下命令中的 `<replaceable>` 替换为对应的信息。可以选择配置其中一项或多项，使用空格或 `and` 分隔。

- 可以在创建用户 (`create user`) 时配置登录时需要校验的证书信息：

```
create user 'u1'@'%'  require issuer '<replaceable>' subject '<replaceable>' san '<
    ↪ replaceable>' cipher '<replaceable>';
```

- 可以在赋予权限 (`grant`) 时配置登录时需要校验的证书信息：

```
grant all on *.* to 'u1'@'%' require issuer '<replaceable>' subject '<replaceable>' san '<
    ↪ replaceable>' cipher '<replaceable>';
```

- 还可以在修改已有用户 (`alter user`) 时配置登录时需要校验的证书信息：

```
alter user 'u1'@'%' require issuer '<replaceable>' subject '<replaceable>' san '<replaceable
    ↪ >' cipher '<replaceable>';
```

配置完成后，用户在登录时 TiDB 会验证以下内容：

- 使用 SSL 登录，且证书为服务器配置的 CA 证书所签发
- 证书的 `Issuer` 信息和权限配置里的信息相匹配
- 证书的 `Subject` 信息和权限配置里的信息相匹配
- 证书的 `Subject Alternative Name` 信息和权限配置里的信息相匹配

全部验证通过后用户才能登录，否则会报 `ERROR 1045 (28000): Access denied` 错误。登录后，可以通过以下命令来查看当前链接是否使用证书登录、TLS 版本和 Cipher 算法。

连接 MySQL 客户端并执行：

```
\s
```

返回结果如下：

```
--------------
mysql  Ver 15.1 Distrib 10.4.10-MariaDB, for Linux (x86_64) using readline 5.1

Connection id:       1
Current database:    test
Current user:        root@127.0.0.1
SSL:                 Cipher in use is TLS_AES_256_GCM_SHA384
```

然后执行：

```
show variables like '%ssl%';
```

返回结果如下：

```
+---------------+---------------------------------+
| Variable_name | Value                           |
+---------------+---------------------------------+
| ssl_cert      | /path/to/server-cert.pem        |
| ssl_ca        | /path/to/ca-cert.pem            |
| have_ssl      | YES                             |
| have_openssl  | YES                             |
| ssl_key       | /path/to/server-key.pem         |
+---------------+---------------------------------+
6 rows in set (0.067 sec)
```

12.10.5.4 更新和替换证书

证书和密钥通常会周期性更新。下文介绍更新密钥和证书的流程。

CA 证书是客户端和服务端相互校验的依据，所以如果需要替换 CA 证书，则需要生成一个组合证书来在替换期间同时支持客户端和服务器上新旧证书的验证，并优先替换客户端和服务端的 CA 证书，再替换客户端和服务端的密钥和证书。

12.10.5.4.1 更新 CA 密钥和证书

1. 以替换 CA 密钥为例（假设 `ca-key.pem` 被盗），将旧的 CA 密钥和证书进行备份：

    ```
    mv ca-key.pem ca-key.old.pem && \
    mv ca-cert.pem ca-cert.old.pem
    ```

2. 生成新的 CA 密钥：

    ```
    sudo openssl genrsa 2048 > ca-key.pem
    ```

3. 用新的密钥生成新的 CA 证书：

    ```
    sudo openssl req -new -x509 -nodes -days 365000 -key ca-key.pem -out ca-cert.new.pem
    ```

> **注意：**
>
> 生成新的 CA 证书是为了替换密钥和证书，保证在线用户不受影响。所以以上命令中填写的附加信息必须与已配置的 `require issuer` 信息一致。

4. 生成组合 CA 证书：

    ```
    cat ca-cert.new.pem ca-cert.old.pem > ca-cert.pem
    ```

之后使用新生成的组合 CA 证书并重启 TiDB Server，此时服务端可以同时接受和使用新旧 CA 证书。

之后先将所有客户端用的 CA 证书也替换为新生成的组合 CA 证书，使客户端能同时和使用新旧 CA 证书。

12.10.5.4.2 更新客户端密钥和证书

> **注意：**
>
> 需要将集群中所有服务端和客户端使用的 CA 证书都替换为新生成的组合 CA 证书后才能开始进行以下步骤。

1. 生成新的客户端 RSA 密钥：

```
sudo openssl req -newkey rsa:2048 -days 365000 -nodes -keyout client-key.new.pem -out client
    ↪ -req.new.pem && \
sudo openssl rsa -in client-key.new.pem -out client-key.new.pem
```

> **注意：**
>
> 以上命令是为了替换密钥和证书，保证在线用户不受影响，所以以上命令中填写的附加信息必须与已配置的 `require subject` 信息一致。

2. 使用新的组合 CA 证书和新 CA 密钥生成新客户端证书：

```
sudo openssl x509 -req -in client-req.new.pem -days 365000 -CA ca-cert.pem -CAkey ca-key.pem
    ↪  -set_serial 01 -out client-cert.new.pem
```

3. 让客户端使用新的客户端密钥和证书来连接 TiDB （以 MySQL 客户端为例）：

```
mysql -utest -h0.0.0.0 -P4000 --ssl-cert /path/to/client-cert.new.pem --ssl-key /path/to/
    ↪ client-key.new.pem --ssl-ca /path/to/ca-cert.pem
```

> **注意：**
>
> `/path/to/client-cert.new.pem`、`/path/to/client-key.new.pem` 和 `/path/to/ca-cert.pem` 是 CA 证书、客户端密钥和客户端存放的路径。可将以上命令中的这些部分替换为实际的路径。

12.10.5.4.3 更新服务端密钥和证书

1. 生成新的服务端 RSA 密钥：

```
sudo openssl req -newkey rsa:2048 -days 365000 -nodes -keyout server-key.new.pem -out server
    ↪ -req.new.pem && \
sudo openssl rsa -in server-key.new.pem -out server-key.new.pem
```

2. 使用新的组合 CA 证书和新 CA 密钥生成新服务端证书：

```
sudo openssl x509 -req -in server-req.new.pem -days 365000 -CA ca-cert.pem -CAkey ca-key.pem
    ↪  -set_serial 01 -out server-cert.new.pem
```

3. 配置 TiDB 使用上面新生成的服务端密钥和证书并重启。参见配置 TiDB 服务端。

12.11 SQL

12.11.1 SQL 语言结构和语法

12.11.1.1 属性

12.11.1.1.1 AUTO_INCREMENT

本文介绍列属性 AUTO_INCREMENT 的基本概念、实现原理、自增相关的特性，以及使用限制。

基本概念

AUTO_INCREMENT 是用于自动填充缺省列值的列属性。当 INSERT 语句没有指定 AUTO_INCREMENT 列的具体值时，系统会自动地为该列分配一个值。

出于性能原因，自增编号是系统批量分配给每台 TiDB 服务器的值（默认 3 万个值），因此自增编号能保证唯一性，但分配给 INSERT 语句的值仅在单台 TiDB 服务器上具有单调性。

```
CREATE TABLE t(id int PRIMARY KEY AUTO_INCREMENT, c int);
```

```
INSERT INTO t(c) VALUES (1);
INSERT INTO t(c) VALUES (2);
INSERT INTO t(c) VALUES (3), (4), (5);
```

```
SELECT * FROM t;
+----+---+
| id | c |
+----+---+
| 1  | 1 |
| 2  | 2 |
| 3  | 3 |
| 4  | 4 |
| 5  | 5 |
+----+---+
5 rows in set (0.01 sec)
```

此外，AUTO_INCREMENT 还支持显式指定列值的插入语句，此时 TiDB 会保存显式指定的值：

```
INSERT INTO t(id, c) VALUES (6, 6);
```

```
SELECT * FROM t;
+----+---+
| id | c |
+----+---+
| 1  | 1 |
| 2  | 2 |
| 3  | 3 |
| 4  | 4 |
| 5  | 5 |
| 6  | 6 |
+----+---+
6 rows in set (0.01 sec)
```

以上用法和 MySQL 的 AUTO_INCREMENT 用法一致。但在隐式分配的具体值方面，TiDB 和 MySQL 之间具有较为显著的差异。

实现原理

TiDB 实现 AUTO_INCREMENT 隐式分配的原理是，对于每一个自增列，都使用一个全局可见的键值对用于记录当前已分配的最大 ID。由于分布式环境下的节点通信存在一定开销，为了避免写请求放大的问题，每个 TiDB 节点在分配 ID 时，都申请一段 ID 作为缓存，用完之后再去取下一段，而不是每次分配都向存储节点申请。例如，对于以下新建的表：

```
CREATE TABLE t(id int UNIQUE KEY AUTO_INCREMENT, c int);
```

假设集群中有两个 TiDB 实例 A 和 B，如果向 A 和 B 分别对 t 执行一条插入语句：

```
INSERT INTO t (c) VALUES (1)
```

实例 A 可能会缓存 [1,30000] 的自增 ID，而实例 B 则可能缓存 [30001,60000] 的自增 ID。各自实例缓存的 ID 将随着执行将来的插入语句被作为缺省值，顺序地填充到 AUTO_INCREMENT 列中。

基本特性

唯一性保证

> 警告：
>
> 在集群中有多个 TiDB 实例时，如果表结构中有自增 ID，建议不要混用显式插入和隐式分配 (即自增列的缺省值和自定义值)，否则可能会破坏隐式分配值的唯一性。

例如在上述示例中，依次执行如下操作：

1. 客户端向实例 B 插入一条将 id 设置为 2 的语句 INSERT INTO t VALUES (2, 1)，并执行成功。
2. 客户端向实例 A 发送 INSERT 语句 INSERT INTO t (c)(1)，这条语句中没有指定 id 的值，所以会由 A 分配。当前 A 缓存了 [1, 30000] 这段 ID，可能会分配 2 为自增 ID 的值，并把本地计数器加 1。而此时数据库中已经存在 id 为 2 的数据，最终返回 Duplicated Error 错误。

单调性保证

TiDB 保证 AUTO_INCREMENT 自增值在单台服务器上单调递增。以下示例在一台服务器上生成连续的 AUTO_INCREMENT 自增值 1-3：

```
CREATE TABLE t (a int PRIMARY KEY AUTO_INCREMENT, b timestamp NOT NULL DEFAULT NOW());
INSERT INTO t (a) VALUES (NULL), (NULL), (NULL);
SELECT * FROM t;
```

```
Query OK, 0 rows affected (0.11 sec)
Query OK, 3 rows affected (0.02 sec)
Records: 3  Duplicates: 0  Warnings: 0
+---+---------------------+
| a | b                   |
+---+---------------------+
| 1 | 2020-09-09 20:38:22 |
| 2 | 2020-09-09 20:38:22 |
| 3 | 2020-09-09 20:38:22 |
+---+---------------------+
3 rows in set (0.00 sec)
```

如果在另一台服务器上执行插入操作，那么 AUTO_INCREMENT 值的顺序可能会剧烈跳跃，这是由于每台服务器都有各自缓存的 AUTO_INCREMENT 自增值。

```
INSERT INTO t (a) VALUES (NULL);
SELECT * FROM t;
```

```
Query OK, 1 row affected (0.03 sec)

+---------+---------------------+
| a       | b                   |
+---------+---------------------+
|       1 | 2020-09-09 20:38:22 |
|       2 | 2020-09-09 20:38:22 |
|       3 | 2020-09-09 20:38:22 |
| 2000001 | 2020-09-09 20:43:43 |
+---------+---------------------+
4 rows in set (0.00 sec)
```

以下示例在最先的一台服务器上执行一个插入 INSERT 操作，生成 AUTO_INCREMENT 值 4。因为这台服务器上仍有剩余的 AUTO_INCREMENT 缓存值可用于分配。在该示例中，值的顺序不具有全局单调性：

```
INSERT INTO t (a) VALUES (NULL);
Query OK, 1 row affected (0.01 sec)

SELECT * FROM t ORDER BY b;
+---------+---------------------+
```

```
| a       | b                   |
+---------+---------------------+
|       1 | 2020-09-09 20:38:22 |
|       2 | 2020-09-09 20:38:22 |
|       3 | 2020-09-09 20:38:22 |
| 2000001 | 2020-09-09 20:43:43 |
|       4 | 2020-09-09 20:44:43 |
+---------+---------------------+
5 rows in set (0.00 sec)
```

AUTO_INCREMENT 缓存不会持久化，重启会导致缓存值失效。以下示例中，最先的一台服务器重启后，向该服务器执行一条插入操作：

```
INSERT INTO t (a) VALUES (NULL);
Query OK, 1 row affected (0.01 sec)

SELECT * FROM t ORDER BY b;
+---------+---------------------+
| a       | b                   |
+---------+---------------------+
|       1 | 2020-09-09 20:38:22 |
|       2 | 2020-09-09 20:38:22 |
|       3 | 2020-09-09 20:38:22 |
| 2000001 | 2020-09-09 20:43:43 |
|       4 | 2020-09-09 20:44:43 |
| 2030001 | 2020-09-09 20:54:11 |
+---------+---------------------+
6 rows in set (0.00 sec)
```

TiDB 服务器频繁重启可能导致 AUTO_INCREMENT 缓存值被快速消耗。在以上示例中，最先的一台服务器本来有可用的缓存值 [5-3000]。但重启后，这些值便丢失了，无法进行重新分配。

用户不应指望 AUTO_INCREMENT 值保持连续。在以下示例中，一台 TiDB 服务器的缓存值为 [2000001-2030000]。当手动插入值 2029998 时，TiDB 取用了一个新缓存区间的值：

```
INSERT INTO t (a) VALUES (2029998);
Query OK, 1 row affected (0.01 sec)

INSERT INTO t (a) VALUES (NULL);
Query OK, 1 row affected (0.01 sec)

INSERT INTO t (a) VALUES (NULL);
Query OK, 1 row affected (0.00 sec)

INSERT INTO t (a) VALUES (NULL);
Query OK, 1 row affected (0.02 sec)
```

```
INSERT INTO t (a) VALUES (NULL);
Query OK, 1 row affected (0.01 sec)

SELECT * FROM t ORDER BY b;
+---------+---------------------+
| a       | b                   |
+---------+---------------------+
|       1 | 2020-09-09 20:38:22 |
|       2 | 2020-09-09 20:38:22 |
|       3 | 2020-09-09 20:38:22 |
| 2000001 | 2020-09-09 20:43:43 |
|       4 | 2020-09-09 20:44:43 |
| 2030001 | 2020-09-09 20:54:11 |
| 2029998 | 2020-09-09 21:08:11 |
| 2029999 | 2020-09-09 21:08:11 |
| 2030000 | 2020-09-09 21:08:11 |
| 2060001 | 2020-09-09 21:08:11 |
| 2060002 | 2020-09-09 21:08:11 |
+---------+---------------------+
11 rows in set (0.00 sec)
```

以上示例插入 2030000 后，下一个值为 2060001，即顺序出现跳跃。这是因为另一台 TiDB 服务器获取了中间缓存区间 [2030001-2060000]。当部署有多台 TiDB 服务器时，AUTO_INCREMENT 值的顺序会出现跳跃，因为对缓存值的请求是交叉出现的。

缓存大小控制

TiDB 自增 ID 的缓存大小在早期版本中是对用户透明的。从 v3.1.2、v3.0.14 和 v4.0.rc-2 版本开始，TiDB 引入了 AUTO_ID_CACHE 表选项来允许用户自主设置自增 ID 分配缓存的大小。例如：

```
CREATE TABLE t(a int AUTO_INCREMENT key) AUTO_ID_CACHE 100;
Query OK, 0 rows affected (0.02 sec)

INSERT INTO t VALUES();
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

SELECT * FROM t;
+---+
| a |
+---+
| 1 |
+---+
1 row in set (0.01 sec)
```

此时如果将该列的自增缓存无效化，重新进行隐式分配：

```
DELETE FROM t;
Query OK, 1 row affected (0.01 sec)

RENAME TABLE t to t1;
Query OK, 0 rows affected (0.01 sec)

INSERT INTO t1 VALUES()
Query OK, 1 row affected (0.00 sec)

SELECT * FROM t;
+-----+
| a   |
+-----+
| 101 |
+-----+
1 row in set (0.00 sec)
```

可以看到再一次分配的值为 101，说明该表的自增 ID 分配缓存的大小为 100。

此外如果在批量插入的 INSERT 语句中所需连续 ID 长度超过 AUTO_ID_CACHE 的长度时，TiDB 会适当调大缓存以便能够保证该语句的正常插入。

自增步长和偏移量设置

从 v3.0.9 和 v4.0.rc-1 开始，和 MySQL 的行为类似，自增列隐式分配的值遵循 session 变量 @@auto_increment_increment 和 @@auto_increment_offset 的控制，其中自增列隐式分配的值 (ID) 将满足式子 (ID - auto_increment_offset)% auto_increment_increment == 0。

使用限制

目前在 TiDB 中使用 AUTO_INCREMENT 有以下限制：

- 必须定义在主键或者唯一索引的列上。
- 只能定义在类型为整数、 FLOAT 或 DOUBLE 的列上。
- 不支持与列的默认值 DEFAULT 同时指定在同一列上。
- 不支持使用 ALTER TABLE 来添加 AUTO_INCREMENT 属性。
- 支持使用 ALTER TABLE 来移除 AUTO_INCREMENT 属性。但从 TiDB 2.1.18 和 3.0.4 版本开始，TiDB 通过 session 变量 @@tidb_allow_remove_auto_inc 控制是否允许通过 ALTER TABLE MODIFY 或 ALTER TABLE CHANGE 来移除列的 AUTO_INCREMENT 属性，默认是不允许移除。

12.11.1.1.2 AUTO_RANDOM 从 v3.1.0 版本开始引入

> 注意：
>
> AUTO_RANDOM 属性已于 v4.0.3 版本成为正式功能。

使用场景

AUTO_RANDOM 用于解决大批量写数据入 TiDB 时因含有整型自增主键列的表而产生的热点问题。详情参阅TiDB 高并发写入场景最佳实践。

以下面语句建立的表为例：

```
CREATE TABLE t (a bigint PRIMARY KEY AUTO_INCREMENT, b varchar(255))
```

在以上语句所建的表上执行大量未指定主键值的 INSERT 语句，示例如下：

```
INSERT INTO t(b) VALUES ('a'), ('b'), ('c')
```

如以上语句，由于未指定主键列的值（a 列），TiDB 会使用连续自增的行值作为行 ID，可能导致单个 TiKV 节点上产生写入热点，进而影响对外提供服务的性能。要避免这种写入热点，可以在执行建表语句时为 a 列指定 AUTO_RANDOM 属性而不是 AUTO_INCREMENT 属性。示例如下：

```
CREATE TABLE t (a bigint PRIMARY KEY AUTO_RANDOM, b varchar(255))
```

或者

```
CREATE TABLE t (a bigint AUTO_RANDOM, b varchar(255), PRIMARY KEY (a))
```

此时再执行形如 INSERT INTO t(b)values... 的 INSERT 语句。

- 隐式分配：如果该 INSERT 语句没有指定整型主键列（a 列）的值，或者指定为 NULL，TiDB 会为该列自动分配值。该值不保证自增，不保证连续，只保证唯一，避免了连续的行 ID 带来的热点问题。
- 显式插入：如果该 INSERT 语句显式指定了整型主键列的值，和 AUTO_INCREMENT 属性类似，TiDB 会保存该值。注意，如果未在系统变量 @@sql_mode 中设置 NO_AUTO_VALUE_ON_ZERO，即使显式指定整型主键列的值为 0，TiDB 也会为该列自动分配值。

> 注意：
>
> 从 v4.0.3 开始，要使用显式插入的功能，需要将系统变量 @@allow_auto_random_explicit_insert 的值设置为 1（默认值为 0）。默认不支持显式插入的具体原因请参考使用限制一节。

自动分配值的计算方式如下：

该行值在二进制形式下，除去符号位的最高五位（称为 shard bits）由当前事务的开始时间决定，剩下的位数按照自增的顺序分配。

若要使用一个不同的 shard bits 的数量，可以在 AUTO_RANDOM 后面加一对括号，并在括号中指定想要的 shard bits 数量。示例如下：

```
CREATE TABLE t (a bigint PRIMARY KEY AUTO_RANDOM(3), b varchar(255))
```

以上建表语句中，shard bits 的数量为 3。shard bits 的数量的取值范围是 [1, field_max_bits)，其中 field_max_bits 为整型主键列类型占用的位长度。

创建完表后，使用 SHOW WARNINGS 可以查看当前表可支持的最大隐式分配的次数：

```
SHOW WARNINGS
```

```
+-------+------+-----------------------------------------------------------+
| Level | Code | Message                                                   |
+-------+------+-----------------------------------------------------------+
| Note  | 1105 | Available implicit allocation times: 1152921504606846976 |
+-------+------+-----------------------------------------------------------+
```

> 注意：
>
> 为保证可隐式分配的次数最大，从 v4.0.3 开始，AUTO_RANDOM 列类型只能为 BIGINT。

另外，要查看某张含有 AUTO_RANDOM 属性的表的 shard bits 数量，可以在系统表 information_schema.tables 中 TIDB_ROW_ID_SHARDING_INFO 一列看到模式为 PK_AUTO_RANDOM_BITS=x 的值，其中 x 为 shard bits 的数量。

AUTO_RANDOM 列隐式分配的值会影响 last_insert_id()。可以使用 SELECT last_insert_id() 获取上一次 TiDB 隐式分配的 ID，例如：

```
INSERT INTO t (b) VALUES ("b")
SELECT * FROM t;
SELECT last_insert_id()
```

可能得到的结果如下：

```
+------------+---+
| a          | b |
+------------+---+
| 1073741825 | b |
+------------+---+

+------------------+
| last_insert_id() |
+------------------+
| 1073741825       |
+------------------+
```

兼容性

TiDB 支持解析版本注释语法。示例如下：

```
CREATE TABLE t (a bigint PRIMARY KEY /*T![auto_rand] auto_random */)
```

```
CREATE TABLE t (a bigint PRIMARY KEY AUTO_RANDOM)
```

以上两个语句含义相同。

在 SHOW CREATE TABLE 的结果中，AUTO_RANDOM 属性会被注释掉。注释会附带一个特性标识符，例如 /*T![auto_rand] auto_random */。其中 auto_rand 表示 AUTO_RANDOM 的特性标识符，只有实现了该标识符对应特性的 TiDB 版本才能够正常解析 SQL 语句片段。

该功能支持向前兼容，即降级兼容。没有实现对应特性的 TiDB 版本则会忽略表（带有上述注释）的 AUTO_RANDOM 属性，因此能够使用含有该属性的表。

使用限制

目前在 TiDB 中使用 AUTO_RANDOM 有以下限制：

- 该属性必须指定在整数类型的主键列上，否则会报错。此外，当配置项 alter-primary-key 的值为 true 时，即使是整型主键列，也不支持使用 AUTO_RANDOM。
- 不支持使用 ALTER TABLE 来修改 AUTO_RANDOM 属性，包括添加或移除该属性。
- 不支持修改含有 AUTO_RANDOM 属性的主键列的列类型。
- 不支持与 AUTO_INCREMENT 同时指定在同一列上。
- 不支持与列的默认值 DEFAULT 同时指定在同一列上。
- 插入数据时，不建议自行显式指定含有 AUTO_RANDOM 列的值。不恰当地显式赋值，可能会导致该表提前耗尽用于自动分配的数值。

12.11.1.1.3 SHARD_ROW_ID_BITS

本文介绍表属性 SHARD_ROW_ID_BITS，它用来设置隐式 _tidb_rowid 分片数量的 bit 位数。

基本概念

对于非整数主键或没有主键的表，TiDB 会使用一个隐式的自增 rowid。大量执行 INSERT 插入语句时会把数据集中写入单个 Region，造成写入热点。

通过设置 SHARD_ROW_ID_BITS，可以把 rowid 打散写入多个不同的 Region，缓解写入热点问题。但是设置的过大会造成 RPC 请求数放大，增加 CPU 和网络开销。

- SHARD_ROW_ID_BITS = 4 表示 16 个分片
- SHARD_ROW_ID_BITS = 6 表示 64 个分片
- SHARD_ROW_ID_BITS = 0 表示默认值 1 个分片

语句示例

- CREATE TABLE：CREATE TABLE t (c int)SHARD_ROW_ID_BITS = 4;
- ALTER TABLE：ALTER TABLE t SHARD_ROW_ID_BITS = 4;

12.11.1.2 字面值

TiDB 字面值包括字符字面值、数值字面值、时间日期字面值、十六进制、二进制字面值和 NULL 字面值。以下分别对这些字面值进行一一介绍。

12.11.1.2.1 String Literals

String Literals 是一个 bytes 或者 characters 的序列，两端被单引号 ' 或者双引号 " 包围，例如：

```
'example string'
"example string"
```

如果字符串是连续的，会被合并为一个独立的 string。以下表示是一样的：

```
'a string'
'a' ' ' 'string'
"a" ' ' "string"
```

如果开启了 ANSI_QUOTES SQL MODE，那么只有单引号内的会被认为是 String Literals，对于双引号内的字符串，会被认为是一个 identifier。

字符串分为以下两种：

- 二进制字符串 (binary string)：由字节序列构成，它的 charset 和 collation 都是 `binary`，在互相比较时利用字节作为单位。
- 非二进制字符串：由字符序列构成，有除 `binary` 以外的多种 charset 和 collation，在相互比较时用字符（一个字符可能包含多个字节，取决于 charset 的选择）作为单位。

一个 String Literal 可以拥有一个可选的 `character set introducer` 和 `COLLATE clause`，可以用来指定特定的 charset 和 collation。

```
[_charset_name]'string' [COLLATE collation_name]
```

例如：

```
SELECT _latin1'string';
SELECT _binary'string';
SELECT _utf8'string' COLLATE utf8_bin;
```

你可以使用 `N'literal'` 或者 `n'literal'` 来创建使用 national character set 的字符串，下列语句是一样的：

```
SELECT N'some text';
SELECT n'some text';
SELECT _utf8'some text';
```

要在字符串中表示某些特殊字符，可以利用转义字符进行转义：

转义字符	含义
\0	ASCII NUL (X’00’) 字符
\’	单引号
\“	双引号
\b	退格符号
\n	换行符

转义字符	含义
\r	回车符
\t	tab 符（制表符）
\z	ASCII 26 (Ctrl + Z)
\\	反斜杠 \
\%	%
_	_

如果要在 ' 包围的字符串中表示 "，或者在 " 包围的字符串中表示 '，可以不使用转义字符。

更多细节见 MySQL 官方文档。

12.11.1.2.2 Numeric Literals

数值字面值包括 integer 跟 Decimal 类型跟浮点数字面值。

integer 可以包括 . 作为小数点分隔，数字前可以有 - 或者 + 来表示正数或者负数。

精确数值字面值可以表示为如下格式：`1, .2, 3.4, -5, -6.78, +9.10.`

科学记数法也是被允许的，表示为如下格式：`1.2E3, 1.2E-3, -1.2E3, -1.2E-3`。

更多细节见 MySQL 官方文档。

12.11.1.2.3 Date and Time Literals

Date 跟 Time 字面值有几种格式，例如用字符串表示，或者直接用数字表示。在 TiDB 里面，当 TiDB 期望一个 Date 的时候，它会把 `'2017-08-24'`，`'20170824'`，20170824 当做是 Date。

TiDB 的 Date 值有以下几种格式：

- `'YYYY-MM-DD'` 或者 `'YY-MM-DD'`，这里的 - 分隔符并不是严格的，可以是任意的标点符号。比如 `'2017-08-24'`，`'2017&08&24'`，`'2012@12^31'` 都是一样的。唯一需要特别对待的是 '.' 号，它被当做是小数点，用于分隔整数和小数部分。

  Date 和 Time 部分可以被 'T' 分隔，它的作用跟空格符是一样的，例如 `2017-8-24 10:42:00` 跟 `2017-8-24 T10:42:00` 是一样的。

- `'YYYYMMDDHHMMSS'` 或者 `'YYMMDDHHMMSS'`，例如 `'20170824104520'` 和 `'170824104520'` 被当做是 `'2017-08-24 10:45:20'`，但是如果你提供了一个超过范围的值，例如`'170824304520'`，那这就不是一个有效的 Date 字面值。需要注意 `YYYYMMDD HHMMSS`，`YYYYMMDD HH:MM:DD`，`YYYY-MM-DD HHMMSS` 等不正确的格式会插入失败。

- YYYYMMDDHHMMSS 或者 YYMMDDHHMMSS，注意这里没有单引号或者双引号，是一个数字。例如 20170824104520 表示为 `'2017-08-24 10:45:20'`。

DATETIME 或者 TIMESTAMP 值可以接一个小数部分，用来表示微秒（精度最多到小数点后 6 位），用小数点 . 分隔。

如果 Date 的 year 部分只有两个数字，这是有歧义的（推荐使用四个数字的格式），TiDB 会尝试用以下的规则来解释：

- year 值如果在 70-99 范围，那么被转换成 1970-1999。
- year 值如果在 00-69 范围，那么被转换成 2000-2069。

对于小于 10 的 month 或者 day 值，'2017-8-4' 跟 '2017-08-04' 是一样的。对于 Time 也是一样，比如 '2017-08-24 1:2:3' 跟 '2017-08-24 01:02:03'是一样的。

在需要 Date 或者 Time 的语境下, 对于数值，TiDB 会根据数值的长度来选定指定的格式：

- 6 个数字，会被解释为 YYMMDD。
- 12 个数字，会被解释为 YYMMDDHHMMSS。
- 8 个数字，会解释为 YYYYMMDD。
- 14 个数字，会被解释为 YYYYMMDDHHMMSS。

对于 Time 类型，TiDB 用以下格式来表示：

- 'D HH:MM:SS'，或者 'HH:MM:SS'，'HH:MM'，'D HH:MM'，'D HH'，'SS'，这里的 D 表示 days，合法的范围是 0-34。
- 数值 HHMMSS，例如 231010 被解释为'23:10:10'。
- 数值 SS，MMSS，HHMMSS 都是可以被当做 Time。

Time 类型的小数点也是 .，精度最多小数点后 6 位。

更多细节见 MySQL 官方文档。

12.11.1.2.4 Boolean Literals

常量 TRUE 和 FALSE 等于 1 和 0，它是大小写不敏感的。

```
SELECT TRUE, true, tRuE, FALSE, FaLsE, false;
```

```
+------+------+------+-------+-------+-------+
| TRUE | true | tRuE | FALSE | FaLsE | false |
+------+------+------+-------+-------+-------+
|    1 |    1 |    1 |     0 |     0 |     0 |
+------+------+------+-------+-------+-------+
1 row in set (0.00 sec)
```

12.11.1.2.5 Hexadecimal Literals

十六进制字面值是有 X 和 0x 前缀的字符串，后接表示十六进制的数字。注意 0x 是大小写敏感的，不能表示为 0X。

例:

```
X'ac12'
X'12AC'
x'ac12'
```

```
x'12AC'
0xac12
0x12AC
```

以下是不合法的十六进制字面值：

```
X'1z' (z 不是合法的十六进制值)
0X12AC (0X 必须用小写的 0x)
```

对于使用 `X'val'` 格式的十六进制字面值，`val` 必须包含偶数个字符，如果 `val` 的长度是奇数（比如 `X'A'`、`X'11A'`），可以在前面补一个 `0` 来避免语法错误。

```
select X'aff';
```

```
ERROR 1105 (HY000): line 0 column 13 near ""hex literal: invalid hexadecimal format, must even
    ↪ numbers, but 3 (total length 13)
```

```
select X'0aff';
```

```
+---------+
| X'0aff' |
+---------+
| 0x0aff  |
+---------+
1 row in set (0.00 sec)
```

默认情况，十六进制字面值是一个二进制字符串。

如果需要将一个字符串或者数字转换为十六进制字面值，可以使用内建函数 `HEX()`：

```
SELECT HEX('TiDB');
```

```
+-------------+
| HEX('TiDB') |
+-------------+
| 54694442    |
+-------------+
1 row in set (0.01 sec)
```

```
SELECT X'54694442';
```

```
+-------------+
| X'54694442' |
+-------------+
| TiDB        |
+-------------+
1 row in set (0.00 sec)
```

12.11.1.2.6 Bit-Value Literals

位值字面值用 b 或者 0b 做前缀，后接以 0 和 1 组成的二进制数字。其中 0b 是区分大小写的，0B 则会报错。

合法的 Bit-value：

- b’01’
- B’01’
- 0b01

非法的 Bit-value：

- b’2’ (2 不是二进制数值, 必须为 0 或 1)
- 0B01 (0B 必须是小写 0b)

默认情况，位值字面值是一个二进制字符串。

Bit-value 是作为二进制返回的，所以输出到 MySQL Client 可能会无法显示，如果要转换为可打印的字符，可以使用内建函数 BIN() 或者 HEX()：

```
CREATE TABLE t (b BIT(8));
INSERT INTO t SET b = b'00010011';
INSERT INTO t SET b = b'1110';
INSERT INTO t SET b = b'100101';
```

```
SELECT b+0, BIN(b), HEX(b) FROM t;
```

```
+------+--------+--------+
| b+0  | BIN(b) | HEX(b) |
+------+--------+--------+
|   19 | 10011  | 13     |
|   14 | 1110   | E      |
|   37 | 100101 | 25     |
+------+--------+--------+
3 rows in set (0.00 sec)
```

12.11.1.2.7 NULL Values

NULL 代表数据为空，它是大小写不敏感的，与 \N（大小写敏感）同义。

> 注意：
>
> NULL 跟 0 并不一样，跟空字符串 '' 也不一样。

12.11.1.3 Schema 对象名

本文介绍 TiDB SQL 语句中的模式对象名。

模式对象名用于命名 TiDB 中所有的模式对象，包括 database、table、index、column、alias 等等。在 SQL 语句中，可以通过标识符 (identifier) 来引用这些对象。

标识符可以被反引号包裹，即 `SELECT * FROM t` 也可以写成 `SELECT * FROM `t``。但如果标识符中存在至少一个特殊符号，或者它是一个保留关键字，那就必须使用反引号包裹来引用它所代表的模式对象。

```
SELECT * FROM `table` WHERE `table`.id = 20;
```

如果 SQL MODE 中设置了 `ANSI_QUOTES`，那么 TiDB 会将被双引号 " 包裹的字符串识别为 identifier。

```
MySQL [test]> CREATE TABLE "test" (a varchar(10));
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to
    ↪ your TiDB version for the right syntax to use line 1 column 19 near ""test" (a varchar
    ↪ (10))"

MySQL [test]> SET SESSION sql_mode='ANSI_QUOTES';
Query OK, 0 rows affected (0.000 sec)

MySQL [test]> CREATE TABLE "test" (a varchar(10));
Query OK, 0 rows affected (0.012 sec)
```

如果要在被引用的标识符中使用反引号这个字符，则需要重复两次反引号，例如创建一个表 a ‘b：

```
CREATE TABLE `a``b` (a int);
```

在 select 语句中，alias 部分可以用标识符或者字符串：

```
SELECT 1 AS `identifier`, 2 AS 'string';
```

```
+------------+--------+
| identifier | string |
+------------+--------+
|          1 |      2 |
+------------+--------+
1 row in set (0.00 sec)
```

更多细节，请参考 MySQL 文档。

12.11.1.3.1 Identifier Qualifiers

Object Names (对象名字) 有时可以被限定或者省略。例如在创建表的时候可以省略数据库限定名：

```
CREATE TABLE t (i int);
```

如果之前没有使用 USE 或者连接参数来设定数据库，会报 `ERROR 1046 (3D000): No database selected` 错误。此时可以指定数据库限定名：

```
CREATE TABLE test.t (i int);
```

. 的左右两端可以出现空格，table_name.col_name 等于 table_name . col_name。

如果要引用这个模式对象，那么请使用：

```
`table_name`.`col_name`
```

而不是：

```
`table_name.col_name`
```

更多细节，请参考 MySQL 文档。

12.11.1.4 关键字

本文介绍 TiDB 的关键字，对保留字和非保留字作出区分，并汇总所有的关键字以供查询使用。

关键字是 SQL 语句中具有特殊含义的单词，例如 SELECT，UPDATE，DELETE 等等。它们之中有的能够直接作为标识符，被称为非保留关键字（简称非保留字），但有需要经过特殊处理才能作为标识符的字，被称为保留关键字（简称保留字）。

对于保留字，必须使用反引号包裹，才能作为标识符被使用。例如：

```
CREATE TABLE select (a INT);
```

```
ERROR 1105 (HY000): line 0 column 19 near " (a INT)" (total length 27)
```

```
CREATE TABLE `select` (a INT);
```

```
Query OK, 0 rows affected (0.09 sec)
```

而非保留字则不需要反引号也能直接作为标识符。例如 BEGIN 和 END 是非保留字，以下语句能够正常执行：

```
CREATE TABLE `select` (BEGIN int, END int);
```

```
Query OK, 0 rows affected (0.09 sec)
```

有一种特殊情况，如果使用了限定符 .，那么也不需要用反引号：

```
CREATE TABLE test.select (BEGIN int, END int);
```

```
Query OK, 0 rows affected (0.08 sec)
```

下表列出了 TiDB 中所有的关键字。其中保留字用 (R) 来标识。窗口函数的保留字用 (R-Window) 来标识：

A

- ACCOUNT

- ACTION
- ADD (R)
- ADMIN (R)
- ADVISE
- AFTER
- AGAINST
- AGO
- ALGORITHM
- ALL (R)
- ALTER (R)
- ALWAYS
- ANALYZE (R)
- AND (R)
- ANY
- AS (R)
- ASC (R)
- ASCII
- AUTO_ID_CACHE
- AUTO_INCREMENT
- AUTO_RANDOM
- AUTO_RANDOM_BASE
- AVG
- AVG_ROW_LENGTH

B

- BACKEND
- BACKUP
- BACKUPS
- BEGIN
- BETWEEN (R)
- BIGINT (R)
- BINARY (R)
- BINDING
- BINDINGS
- BINLOG
- BIT
- BLOB (R)
- BLOCK
- BOOL
- BOOLEAN
- BOTH (R)
- BTREE
- BUCKETS (R)
- BUILTINS (R)

- BY (R)
- BYTE

C

- CACHE
- CANCEL (R)
- CAPTURE
- CASCADE (R)
- CASCADED
- CASE (R)
- CHAIN
- CHANGE (R)
- CHAR (R)
- CHARACTER (R)
- CHARSET
- CHECK (R)
- CHECKPOINT
- CHECKSUM
- CIPHER
- CLEANUP
- CLIENT
- CMSKETCH (R)
- COALESCE
- COLLATE (R)
- COLLATION
- COLUMN (R)
- COLUMNS
- COLUMN_FORMAT
- COMMENT
- COMMIT
- COMMITTED
- COMPACT
- COMPRESSED
- COMPRESSION
- CONCURRENCY
- CONFIG
- CONNECTION
- CONSISTENT
- CONSTRAINT (R)
- CONTEXT
- CONVERT (R)
- CPU
- CREATE (R)
- CROSS (R)

- CSV_BACKSLASH_ESCAPE
- CSV_DELIMITER
- CSV_HEADER
- CSV_NOT_NULL
- CSV_NULL
- CSV_SEPARATOR
- CSV_TRIM_LAST_SEPARATORS
- CUME_DIST (R-Window)
- CURRENT
- CURRENT_DATE (R)
- CURRENT_ROLE (R)
- CURRENT_TIME (R)
- CURRENT_TIMESTAMP (R)
- CURRENT_USER (R)
- CYCLE

D

- DATA
- DATABASE (R)
- DATABASES (R)
- DATE
- DATETIME
- DAY
- DAY_HOUR (R)
- DAY_MICROSECOND (R)
- DAY_MINUTE (R)
- DAY_SECOND (R)
- DDL (R)
- DEALLOCATE
- DECIMAL (R)
- DEFAULT (R)
- DEFINER
- DELAYED (R)
- DELAY_KEY_WRITE
- DELETE (R)
- DENSE_RANK (R-Window)
- DEPTH (R)
- DESC (R)
- DESCRIBE (R)
- DIRECTORY
- DISABLE
- DISCARD
- DISK
- DISTINCT (R)

- DISTINCTROW (R)
- DIV (R)
- DO
- DOUBLE (R)
- DRAINER (R)
- DROP (R)
- DUAL (R)
- DUPLICATE
- DYNAMIC

E

- ELSE (R)
- ENABLE
- ENCLOSED (R)
- ENCRYPTION
- END
- ENFORCED
- ENGINE
- ENGINES
- ENUM
- ERROR
- ERRORS
- ESCAPE
- ESCAPED (R)
- EVENT
- EVENTS
- EVOLVE
- EXCEPT (R)
- EXCHANGE
- EXCLUSIVE
- EXECUTE
- EXISTS (R)
- EXPANSION
- EXPIRE
- EXPLAIN (R)
- EXTENDED

F

- FALSE (R)
- FAULTS
- FIELDS
- FILE
- FIRST

- FIRST_VALUE (R-Window)
- FIXED
- FLOAT (R)
- FLUSH
- FOLLOWING
- FOR (R)
- FORCE (R)
- FOREIGN (R)
- FORMAT
- FROM (R)
- FULL
- FULLTEXT (R)
- FUNCTION

G

- GENERAL
- GENERATED (R)
- GLOBAL
- GRANT (R)
- GRANTS
- GROUP (R)
- GROUPS (R-Window)

H

- HASH
- HAVING (R)
- HIGH_PRIORITY (R)
- HISTORY
- HOSTS
- HOUR
- HOUR_MICROSECOND (R)
- HOUR_MINUTE (R)
- HOUR_SECOND (R)

I

- IDENTIFIED
- IF (R)
- IGNORE (R)
- IMPORT
- IMPORTS
- IN (R)
- INCREMENT

- INCREMENTAL
- INDEX (R)
- INDEXES
- INFILE (R)
- INNER (R)
- INSERT (R)
- INSERT_METHOD
- INSTANCE
- INT (R)
- INT1 (R)
- INT2 (R)
- INT3 (R)
- INT4 (R)
- INT8 (R)
- INTEGER (R)
- INTERVAL (R)
- INTO (R)
- INVISIBLE
- INVOKER
- IO
- IPC
- IS (R)
- ISOLATION
- ISSUER

J

- JOB (R)
- JOBS (R)
- JOIN (R)
- JSON

K

- KEY (R)
- KEYS (R)
- KEY_BLOCK_SIZE
- KILL (R)

L

- LABELS
- LAG (R-Window)
- LANGUAGE
- LAST

- LASTVAL
- LAST_BACKUP
- LAST_VALUE (R-Window)
- LEAD (R-Window)
- LEADING (R)
- LEFT (R)
- LESS
- LEVEL
- LIKE (R)
- LIMIT (R)
- LINEAR (R)
- LINES (R)
- LIST
- LOAD (R)
- LOCAL
- LOCALTIME (R)
- LOCALTIMESTAMP (R)
- LOCATION
- LOCK (R)
- LOGS
- LONG (R)
- LONGBLOB (R)
- LONGTEXT (R)
- LOW_PRIORITY (R)

M

- MASTER
- MATCH (R)
- MAXVALUE (R)
- MAX_CONNECTIONS_PER_HOUR
- MAX_IDXNUM
- MAX_MINUTES
- MAX_QUERIES_PER_HOUR
- MAX_ROWS
- MAX_UPDATES_PER_HOUR
- MAX_USER_CONNECTIONS
- MB
- MEDIUMBLOB (R)
- MEDIUMINT (R)
- MEDIUMTEXT (R)
- MEMORY
- MERGE
- MICROSECOND
- MINUTE

- MINUTE_MICROSECOND (R)
- MINUTE_SECOND (R)
- MINVALUE
- MIN_ROWS
- MOD (R)
- MODE
- MODIFY
- MONTH

N

- NAMES
- NATIONAL
- NATURAL (R)
- NCHAR
- NEVER
- NEXT
- NEXTVAL
- NO
- NOCACHE
- NOCYCLE
- NODEGROUP
- NODE_ID (R)
- NODE_STATE (R)
- NOMAXVALUE
- NOMINVALUE
- NONE
- NOT (R)
- NOWAIT
- NO_WRITE_TO_BINLOG (R)
- NTH_VALUE (R-Window)
- NTILE (R-Window)
- NULL (R)
- NULLS
- NUMERIC (R)
- NVARCHAR

O

- OFFSET
- ON (R)
- ONLINE
- ONLY
- ON_DUPLICATE
- OPEN

- OPTIMISTIC (R)
- OPTIMIZE (R)
- OPTION (R)
- OPTIONALLY (R)
- OR (R)
- ORDER (R)
- OUTER (R)
- OUTFILE (R)
- OVER (R-Window)

P

- PACK_KEYS
- PAGE
- PARSER
- PARTIAL
- PARTITION (R)
- PARTITIONING
- PARTITIONS
- PASSWORD
- PERCENT_RANK (R-Window)
- PER_DB
- PER_TABLE
- PESSIMISTIC (R)
- PLUGINS
- PRECEDING
- PRECISION (R)
- PREPARE
- PRE_SPLIT_REGIONS
- PRIMARY (R)
- PRIVILEGES
- PROCEDURE (R)
- PROCESS
- PROCESSLIST
- PROFILE
- PROFILES
- PUMP (R)

Q

- QUARTER
- QUERIES
- QUERY
- QUICK

R

- RANGE (R)
- RANK (R-Window)
- RATE_LIMIT
- READ (R)
- REAL (R)
- REBUILD
- RECOVER
- REDUNDANT
- REFERENCES (R)
- REGEXP (R)
- REGION (R)
- REGIONS (R)
- RELEASE (R)
- RELOAD
- REMOVE
- RENAME (R)
- REORGANIZE
- REPAIR
- REPEAT (R)
- REPEATABLE
- REPLACE (R)
- REPLICA
- REPLICATION
- REQUIRE (R)
- RESPECT
- RESTORE
- RESTORES
- RESTRICT (R)
- REVERSE
- REVOKE (R)
- RIGHT (R)
- RLIKE (R)
- ROLE
- ROLLBACK
- ROUTINE
- ROW (R)
- ROWS (R-Window)
- ROW_COUNT
- ROW_FORMAT
- ROW_NUMBER (R-Window)
- RTREE

S

- SAMPLES (R)

- SECOND
- SECONDARY_ENGINE
- SECONDARY_LOAD
- SECONDARY_UNLOAD
- SECOND_MICROSECOND (R)
- SECURITY
- SELECT (R)
- SEND_CREDENTIALS_TO_TIKV
- SEPARATOR
- SEQUENCE
- SERIAL
- SERIALIZABLE
- SESSION
- SET (R)
- SETVAL
- SHARD_ROW_ID_BITS
- SHARE
- SHARED
- SHOW (R)
- SHUTDOWN
- SIGNED
- SIMPLE
- SKIP_SCHEMA_FILES
- SLAVE
- SLOW
- SMALLINT (R)
- SNAPSHOT
- SOME
- SOURCE
- SPATIAL (R)
- SPLIT (R)
- SQL (R)
- SQL_BIG_RESULT (R)
- SQL_BUFFER_RESULT
- SQL_CACHE
- SQL_CALC_FOUND_ROWS (R)
- SQL_NO_CACHE
- SQL_SMALL_RESULT (R)
- SQL_TSI_DAY
- SQL_TSI_HOUR
- SQL_TSI_MINUTE
- SQL_TSI_MONTH
- SQL_TSI_QUARTER
- SQL_TSI_SECOND
- SQL_TSI_WEEK

- SQL_TSI_YEAR
- SSL (R)
- START
- STARTING (R)
- STATS (R)
- STATS_AUTO_RECALC
- STATS_BUCKETS (R)
- STATS_HEALTHY (R)
- STATS_HISTOGRAMS (R)
- STATS_META (R)
- STATS_PERSISTENT
- STATS_SAMPLE_PAGES
- STATUS
- STORAGE
- STORED (R)
- STRAIGHT_JOIN (R)
- STRICT_FORMAT
- SUBJECT
- SUBPARTITION
- SUBPARTITIONS
- SUPER
- SWAPS
- SWITCHES
- SYSTEM_TIME

T

- TABLE (R)
- TABLES
- TABLESPACE
- TABLE_CHECKSUM
- TEMPORARY
- TEMPTABLE
- TERMINATED (R)
- TEXT
- THAN
- THEN (R)
- TIDB (R)
- TIFLASH (R)
- TIKV_IMPORTER
- TIME
- TIMESTAMP
- TINYBLOB (R)
- TINYINT (R)
- TINYTEXT (R)

- TO (R)
- TOPN (R)
- TRACE
- TRADITIONAL
- TRAILING (R)
- TRANSACTION
- TRIGGER (R)
- TRIGGERS
- TRUE (R)
- TRUNCATE
- TYPE

U

- UNBOUNDED
- UNCOMMITTED
- UNDEFINED
- UNICODE
- UNION (R)
- UNIQUE (R)
- UNKNOWN
- UNLOCK (R)
- UNSIGNED (R)
- UPDATE (R)
- USAGE (R)
- USE (R)
- USER
- USING (R)
- UTC_DATE (R)
- UTC_TIME (R)
- UTC_TIMESTAMP (R)

V

- VALIDATION
- VALUE
- VALUES (R)
- VARBINARY (R)
- VARCHAR (R)
- VARCHARACTER (R)
- VARIABLES
- VARYING (R)
- VIEW
- VIRTUAL (R)
- VISIBLE

W

- WARNINGS
- WEEK
- WEIGHT_STRING
- WHEN (R)
- WHERE (R)
- WIDTH (R)
- WINDOW (R-Window)
- WITH (R)
- WITHOUT
- WRITE (R)

X

- X509
- XOR (R)

Y

- YEAR
- YEAR_MONTH (R)

Z

- ZEROFILL (R)

12.11.1.5 用户自定义变量

> **警告：**
>
> 当前该功能为实验特性，不建议在生产环境中使用。

本文介绍 TiDB 的用户自定义变量的概念，以及设置和读取用户自定义变量的方法。

用户自定义变量格式为 `@var_name`。组成 `var_name` 的字符可以是任何能够组成标识符 (identifier) 的字符，包括数字 `0-9`、字母 `a-zA-Z`、下划线 `_`、美元符号 `$` 以及 UTF-8 字符。此外，还包括英文句号 `.`。用户自定义变量是大小写不敏感的。

用户自定义变量跟 session 绑定，当前设置的用户变量只在当前连接中可见，其他客户端连接无法查看。

12.11.1.5.1 设置用户自定义变量

用 SET 语句可以设置用户自定义变量，语法为 SET @var_name = expr [, @var_name = expr] ...;。例如：

```
SET @favorite_db = 'TiDB';
```

```
SET @a = 'a', @b = 'b', @c = 'c';
```

其中赋值符号还可以使用 :=。例如：

```
SET @favorite_db := 'TiDB';
```

赋值符号右边的内容可以是任意合法的表达式。例如：

```
SET @c = @a + @b;
```

```
SET @c = b'1000001' + b'1000001';
```

12.11.1.5.2 读取用户自定义变量

要读取一个用户自定义变量，可以使用 SELECT 语句查询：

```
SELECT @a1, @a2, @a3
```

```
+------+------+------+
| @a1  | @a2  | @a3  |
+------+------+------+
|    1 |    2 |    4 |
+------+------+------+
```

还可以在 SELECT 语句中赋值：

```
SELECT @a1, @a2, @a3, @a4 := @a1+@a2+@a3;
```

```
+------+------+------+--------------------+
| @a1  | @a2  | @a3  | @a4 := @a1+@a2+@a3 |
+------+------+------+--------------------+
|    1 |    2 |    4 |                  7 |
+------+------+------+--------------------+
```

其中变量 @a4 在被修改或关闭连接之前，值始终为 7。

如果设置用户变量时用了十六进制字面量或者二进制字面量，TiDB 会把它当成二进制字符串。如果要将其设置成数字，那么可以手动加上 CAST 转换，或者在表达式中使用数字的运算符：

```
SET @v1 = b'1000001';
SET @v2 = b'1000001'+0;
SET @v3 = CAST(b'1000001' AS UNSIGNED);
```

```
SELECT @v1, @v2, @v3;
```

```
+------+------+------+
| @v1  | @v2  | @v3  |
+------+------+------+
| A    | 65   | 65   |
+------+------+------+
```

如果获取一个没有设置过的变量，会返回一个 NULL：

```
SELECT @not_exist;
```

```
+------------+
| @not_exist |
+------------+
| NULL       |
+------------+
```

除了 SELECT 读取用户自定义变量以外，常见的用法还有 PREPARE 语句，例如：

```
SET @s = 'SELECT SQRT(POW(?,2) + POW(?,2)) AS hypotenuse';
PREPARE stmt FROM @s;
SET @a = 6;
SET @b = 8;
EXECUTE stmt USING @a, @b;
```

```
+------------+
| hypotenuse |
+------------+
|         10 |
+------------+
```

用户自定义变量的内容不会在 SQL 语句中被当成标识符，例如：

```
SELECT * FROM t;
```

```
+---+
| a |
+---+
| 1 |
+---+
```

```
SET @col = "`a`";
SELECT @col FROM t;
```

```
+------+
| @col |
+------+
| `a`  |
+------+
```

更多细节，请参考 MySQL 文档。

12.11.1.6 表达式语法 (Expression Syntax)

表达式是一个或多个值、操作符或函数的组合。在 TiDB 中，表达式主要使用在 SELECT 语句的各个子句中，包括 Group by 子句、Where 子句、Having 子句、Join 条件以及窗口函数等。此外，部分 DDL 语句也会使用到表达式，例如建表时默认值的设置、生成列的设置，分区规则等。

表达式包含几种类型：

- 标识符，可参考模式对象名。
- 谓词、数值、字符串、日期表达式等，这些类型的字面值也是表达式。
- 函数调用，窗口函数等。可参考函数和操作符概述 和窗口函数。
- 其他，包括 paramMarker（即 ? ）、系统变量和用户变量、CASE 表达式等。

以下规则是表达式的语法，该语法基于 TiDB parser 的 parser.y 文件中所定义的规则。此外，下列语法图的可导航版本请参考 TiDB SQL 语法图。

```
Expression ::=
    ( singleAtIdentifier assignmentEq | 'NOT' | Expression ( logOr | 'XOR' | logAnd ) )
        ↪ Expression
|   'MATCH' '(' ColumnNameList ')' 'AGAINST' '(' BitExpr FulltextSearchModifierOpt ')'
|   PredicateExpr ( IsOrNotOp 'NULL' | CompareOp ( ( singleAtIdentifier assignmentEq )?
    ↪ PredicateExpr | AnyOrAll SubSelect ) )* ( IsOrNotOp ( trueKwd | falseKwd | 'UNKNOWN' ) )?

PredicateExpr ::=
    BitExpr ( BetweenOrNotOp BitExpr 'AND' BitExpr )* ( InOrNotOp ( '(' ExpressionList ')' |
        ↪ SubSelect ) | LikeOrNotOp SimpleExpr LikeEscapeOpt | RegexpOrNotOp SimpleExpr )?

BitExpr ::=
    BitExpr ( ( '|' | '&' | '<<' | '>>' | '*' | '/' | '%' | 'DIV' | 'MOD' | '^' ) BitExpr | ( '+'
        ↪  | '-' ) ( BitExpr | "INTERVAL" Expression TimeUnit ) )
|   SimpleExpr

SimpleExpr ::=
    SimpleIdent ( ( '->' | '->>' ) stringLit )?
|   FunctionCallKeyword
|   FunctionCallNonKeyword
|   FunctionCallGeneric
|   SimpleExpr ( 'COLLATE' CollationName | pipes SimpleExpr )
```

```
|   WindowFuncCall
|   Literal
|   paramMarker
|   Variable
|   SumExpr
|   ( '!' | '~' | '-' | '+' | 'NOT' | 'BINARY' ) SimpleExpr
|   'EXISTS'? SubSelect
|   ( ( '(' ( ExpressionList ',' )? | 'ROW' '(' ExpressionList ',' ) Expression | builtinCast '('
    ↪  Expression 'AS' CastType | ( 'DEFAULT' | 'VALUES' ) '(' SimpleIdent | 'CONVERT' '('
    ↪ Expression ( ',' CastType | 'USING' CharsetName ) ) ')'
|   'CASE' ExpressionOpt WhenClause+ ElseOpt 'END'
```

12.11.1.7 注释语法

本文档介绍 TiDB 支持的注释语法。

TiDB 支持三种注释风格：

- 用 # 注释一行：

```
SELECT 1+1;     # 注释文字
```

```
+------+
| 1+1  |
+------+
|    2 |
+------+
1 row in set (0.00 sec)
```

- 用 -- 注释一行：

```
SELECT 1+1;     -- 注释文字
```

```
+------+
| 1+1  |
+------+
|    2 |
+------+
1 row in set (0.00 sec)
```

用 -- 注释时，必须要在其之后留出至少一个空格，否则注释不生效：

```
SELECT 1+1--1;
```

```
+--------+
| 1+1--1 |
```

```
+--------+
|      3 |
+--------+
1 row in set (0.01 sec)
```

- 用 `/* */` 注释一块，可以注释多行：

```
SELECT 1 /* 这是行内注释文字 */ + 1;
```

```
+--------+
| 1  + 1 |
+--------+
|      2 |
+--------+
1 row in set (0.01 sec)
```

```
SELECT 1+
/*
/*> 这是一条
/*> 多行注释
/*> */
    1;
```

```
+--------------------+
| 1+

        1 |
+--------------------+
|                  2 |
+--------------------+
1 row in set (0.001 sec)
```

12.11.1.7.1 MySQL 兼容的注释语法

TiDB 也跟 MySQL 保持一致，支持一种 C 风格注释的变体：

```
/*! Specific code */
```

或者

```
/*!50110 Specific code */
```

和 MySQL 一样，TiDB 会执行注释中的语句。

例如：`SELECT /*! STRAIGHT_JOIN */ col1 FROM table1,table2 WHERE ...`

在 TiDB 中，这种写法等价于 `SELECT STRAIGHT_JOIN col1 FROM table1,table2 WHERE ...`

如果注释中指定了 Server 版本号，例如 `/*!50110 KEY_BLOCK_SIZE=1024 */`，在 MySQL 中表示只有 MySQL 的版本大于等于 5.1.10 才会处理这个 comment 中的内容。但是在 TiDB 中，这个 MySQL 版本号不会起作用，所有的 comment 都被会处理。

12.11.1.7.2 TiDB 可执行的注释语法

TiDB 也有独立的注释语法，称为 TiDB 可执行注释语法。主要分为两种：

- `/*T! Specific code */`：该语法只能被 TiDB 解析执行，而在其他数据库中会被忽略。
- `/*T![feature_id] Specific code */`：该语法用于保证 TiDB 不同版本之间的兼容性。只有在当前版本中实现了 `feature_id` 对应的功能特性的 TiDB，才会试图解析该注释里的 SQL 片段。例如 v3.1.1 中引入了 AUTO_RANDOM 特性，该版本能够将 `/*T![auto_rand] auto_random */` 解析为 `auto_random`；而 v3.0.0 中没有实现 AUTO_RANDOM 特性，则上述 SQL 语句片段会被忽略。注意前几个字符 `/*T![` 中，各字符之间没有任何空格。

12.11.1.7.3 优化器注释语法

还有一种注释会被当做是优化器 Hint 特殊对待：

```
SELECT /*+ hint */ FROM ...;
```

TiDB 支持的相关优化器 hint 详见Optimizer Hints。

> **注意：**
>
> 在 MySQL 客户端中，TiDB 可执行注释语法会被默认当成注释被清除掉。在 MySQL 客户端 5.7.7 之前的版本中，Hint 也会被默认当成注释被清除掉。推荐在启动客户端时加上 `--comments` 选项，例如 `mysql -h 127.0.0.1 -P 4000 -uroot --comments`。

更多细节，请参考 MySQL 文档。

12.11.2 SQL 语句

12.11.2.1 ADD COLUMN

`ALTER TABLE.. ADD COLUMN` 语句用于在已有表中添加列。在 TiDB 中，`ADD COLUMN` 为在线操作，不会阻塞表中的数据读写。

12.11.2.1.1 语法图

```
AlterTableStmt ::=
    'ALTER' IgnoreOptional 'TABLE' TableName ( AlterTableSpecListOpt AlterTablePartitionOpt | '
        ↪ ANALYZE' 'PARTITION' PartitionNameList ( 'INDEX' IndexNameList )?
        ↪ AnalyzeOptionListOpt )
```

```
AlterTableSpec ::=
    TableOptionList
|   'SET' 'TIFLASH' 'REPLICA' LengthNum LocationLabelList
|   'CONVERT' 'TO' CharsetKw ( CharsetName | 'DEFAULT' ) OptCollate
|   'ADD' ( ColumnKeywordOpt IfNotExists ( ColumnDef ColumnPosition | '(' TableElementList ')' )
    ↪ | Constraint | 'PARTITION' IfNotExists NoWriteToBinLogAliasOpt (
    ↪ PartitionDefinitionListOpt | 'PARTITIONS' NUM ) )
|   ( ( 'CHECK' | 'TRUNCATE' ) 'PARTITION' | ( 'OPTIMIZE' | 'REPAIR' | 'REBUILD' ) 'PARTITION'
    ↪ NoWriteToBinLogAliasOpt ) AllOrPartitionNameList
|   'COALESCE' 'PARTITION' NoWriteToBinLogAliasOpt NUM
|   'DROP' ( ColumnKeywordOpt IfExists ColumnName RestrictOrCascadeOpt | 'PRIMARY' 'KEY' | '
    ↪ PARTITION' IfExists PartitionNameList | ( KeyOrIndex IfExists | 'CHECK' ) Identifier | '
    ↪ FOREIGN' 'KEY' IfExists Symbol )
|   'EXCHANGE' 'PARTITION' Identifier 'WITH' 'TABLE' TableName WithValidationOpt
|   ( 'IMPORT' | 'DISCARD' ) ( 'PARTITION' AllOrPartitionNameList )? 'TABLESPACE'
|   'REORGANIZE' 'PARTITION' NoWriteToBinLogAliasOpt ReorganizePartitionRuleOpt
|   'ORDER' 'BY' AlterOrderItem ( ',' AlterOrderItem )*
|   ( 'DISABLE' | 'ENABLE' ) 'KEYS'
|   ( 'MODIFY' ColumnKeywordOpt IfExists | 'CHANGE' ColumnKeywordOpt IfExists ColumnName )
    ↪ ColumnDef ColumnPosition
|   'ALTER' ( ColumnKeywordOpt ColumnName ( 'SET' 'DEFAULT' ( SignedLiteral | '(' Expression ')'
    ↪ ) | 'DROP' 'DEFAULT' ) | 'CHECK' Identifier EnforcedOrNot | 'INDEX' Identifier
    ↪ IndexInvisible )
|   'RENAME' ( ( 'COLUMN' | KeyOrIndex ) Identifier 'TO' Identifier | ( 'TO' | '='? | 'AS' )
    ↪ TableName )
|   LockClause
|   AlgorithmClause
|   'FORCE'
|   ( 'WITH' | 'WITHOUT' ) 'VALIDATION'
|   'SECONDARY_LOAD'
|   'SECONDARY_UNLOAD'

ColumnDef ::=
    ColumnName ( Type | 'SERIAL' ) ColumnOptionListOpt

ColumnPosition ::=
    ( 'FIRST' | 'AFTER' ColumnName )?
```

12.11.2.1.2 示例

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT);
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
INSERT INTO t1 VALUES (NULL);
```

```
Query OK, 1 row affected (0.02 sec)
```

```
SELECT * FROM t1;
```

```
+----+
| id |
+----+
|  1 |
+----+
1 row in set (0.00 sec)
```

```
ALTER TABLE t1 ADD COLUMN c1 INT NOT NULL;
```

```
Query OK, 0 rows affected (0.28 sec)
```

```
SELECT * FROM t1;
```

```
+----+----+
| id | c1 |
+----+----+
|  1 |  0 |
+----+----+
1 row in set (0.00 sec)
```

```
ALTER TABLE t1 ADD c2 INT NOT NULL AFTER c1;
```

```
Query OK, 0 rows affected (0.28 sec)
```

```
SELECT * FROM t1;
```

```
+----+----+----+
| id | c1 | c2 |
+----+----+----+
|  1 |  0 |  0 |
+----+----+----+
1 row in set (0.00 sec)
```

12.11.2.1.3 MySQL 兼容性

- 不支持在一条语句中同时添加多列。
- 不支持将新添加的列设为 PRIMARY KEY。
- 不支持将新添加的列设为 AUTO_INCREMENT。
- 对添加生成列有局限性，具体可参考：生成列局限性。

12.11.2.1.4 另请参阅

- ADD INDEX
- CREATE TABLE

12.11.2.2 ADD INDEX

`ALTER TABLE.. ADD INDEX` 语句用于在已有表中添加一个索引。在 TiDB 中，`ADD INDEX` 为在线操作，不会阻塞表中的数据读写。

12.11.2.2.1 语法图

```
AlterTableStmt ::=
    'ALTER' IgnoreOptional 'TABLE' TableName ( AlterTableSpecListOpt AlterTablePartitionOpt | '
        ↪ ANALYZE' 'PARTITION' PartitionNameList ( 'INDEX' IndexNameList )?
        ↪ AnalyzeOptionListOpt )

AlterTableSpec ::=
    TableOptionList
|   'SET' 'TIFLASH' 'REPLICA' LengthNum LocationLabelList
|   'CONVERT' 'TO' CharsetKw ( CharsetName | 'DEFAULT' ) OptCollate
|   'ADD' ( ColumnKeywordOpt IfNotExists ( ColumnDef ColumnPosition | '(' TableElementList ')' )
    ↪ | Constraint | 'PARTITION' IfNotExists NoWriteToBinLogAliasOpt (
    ↪ PartitionDefinitionListOpt | 'PARTITIONS' NUM ) )
|   ( ( 'CHECK' | 'TRUNCATE' ) 'PARTITION' | ( 'OPTIMIZE' | 'REPAIR' | 'REBUILD' ) 'PARTITION'
    ↪ NoWriteToBinLogAliasOpt ) AllOrPartitionNameList
|   'COALESCE' 'PARTITION' NoWriteToBinLogAliasOpt NUM
|   'DROP' ( ColumnKeywordOpt IfExists ColumnName RestrictOrCascadeOpt | 'PRIMARY' 'KEY' | '
    ↪ PARTITION' IfExists PartitionNameList | ( KeyOrIndex IfExists | 'CHECK' ) Identifier | '
    ↪ FOREIGN' 'KEY' IfExists Symbol )
|   'EXCHANGE' 'PARTITION' Identifier 'WITH' 'TABLE' TableName WithValidationOpt
|   ( 'IMPORT' | 'DISCARD' ) ( 'PARTITION' AllOrPartitionNameList )? 'TABLESPACE'
|   'REORGANIZE' 'PARTITION' NoWriteToBinLogAliasOpt ReorganizePartitionRuleOpt
|   'ORDER' 'BY' AlterOrderItem ( ',' AlterOrderItem )*
|   ( 'DISABLE' | 'ENABLE' ) 'KEYS'
|   ( 'MODIFY' ColumnKeywordOpt IfExists | 'CHANGE' ColumnKeywordOpt IfExists ColumnName )
    ↪ ColumnDef ColumnPosition
|   'ALTER' ( ColumnKeywordOpt ColumnName ( 'SET' 'DEFAULT' ( SignedLiteral | '(' Expression ')'
    ↪ ) | 'DROP' 'DEFAULT' ) | 'CHECK' Identifier EnforcedOrNot | 'INDEX' Identifier
    ↪ IndexInvisible )
|   'RENAME' ( ( 'COLUMN' | KeyOrIndex ) Identifier 'TO' Identifier | ( 'TO' | '='? | 'AS' )
    ↪ TableName )
|   LockClause
|   AlgorithmClause
|   'FORCE'
|   ( 'WITH' | 'WITHOUT' ) 'VALIDATION'
```

```
|   'SECONDARY_LOAD'
|   'SECONDARY_UNLOAD'

Constraint ::=
    ConstraintKeywordOpt ConstraintElem

ConstraintKeywordOpt ::=
    ( 'CONSTRAINT' Symbol? )?

ConstraintElem ::=
    ( ( 'PRIMARY' 'KEY' | KeyOrIndex IfNotExists | 'UNIQUE' KeyOrIndexOpt ) IndexNameAndTypeOpt |
        ↪  'FULLTEXT' KeyOrIndexOpt IndexName ) '(' IndexPartSpecificationList ')'
        ↪ IndexOptionList
|   'FOREIGN' 'KEY' IfNotExists IndexName '(' IndexPartSpecificationList ')' ReferDef
|   'CHECK' '(' Expression ')' EnforcedOrNotOpt

IndexNameAndTypeOpt ::=
    IndexName ( 'USING' IndexTypeName )?
|   Identifier 'TYPE' IndexTypeName

IndexPartSpecificationList ::=
    IndexPartSpecification ( ',' IndexPartSpecification )*

IndexPartSpecification ::=
    ( ColumnName OptFieldLen | '(' Expression ')' ) Order

IndexOptionList ::=
    IndexOption*

IndexOption ::=
    'KEY_BLOCK_SIZE' '='? LengthNum
|   IndexType
|   'WITH' 'PARSER' Identifier
|   'COMMENT' stringLit
|   IndexInvisible

KeyOrIndex ::=
    'KEY'
|   'INDEX'

IndexKeyTypeOpt ::=
    ( 'UNIQUE' | 'SPATIAL' | 'FULLTEXT' )?

IndexInvisible ::=
    'VISIBLE'
```

```
|   'INVISIBLE'

IndexTypeName ::=
    'BTREE'
|   'HASH'
|   'RTREE'
```

12.11.2.2.2 示例

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, c1 INT NOT NULL);
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
INSERT INTO t1 (c1) VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 5 rows affected (0.03 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
EXPLAIN SELECT * FROM t1 WHERE c1 = 3;
```

```
+-------------------------+----------+-----------+---------------+--------------------------------+
    ↪
| id                      | estRows  | task      | access object | operator info
    ↪                  |
+-------------------------+----------+-----------+---------------+--------------------------------+
    ↪
| TableReader_7           | 10.00    | root      |               | data:Selection_6
    ↪               |
| └─Selection_6           | 10.00    | cop[tikv] |               | eq(test.t1.c1, 3)
    ↪              |
|   └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t1      | keep order:false, stats:
    ↪ pseudo |
+-------------------------+----------+-----------+---------------+--------------------------------+
    ↪
3 rows in set (0.00 sec)
```

```
ALTER TABLE t1 ADD INDEX (c1);
```

```
Query OK, 0 rows affected (0.30 sec)
```

```
EXPLAIN SELECT * FROM t1 WHERE c1 = 3;
```

```
+-----------------------+---------+-----------+-----------------------+-----------------------------------
    ↪
| id                    | estRows | task      | access object         | operator info
    ↪                               |
+-----------------------+---------+-----------+-----------------------+-----------------------------------
    ↪
| IndexReader_6         | 0.01    | root      |                       | index:IndexRangeScan_5
    ↪                       |
| └─IndexRangeScan_5    | 0.01    | cop[tikv] | table:t1, index:c1(c1) | range:[3,3], keep
    ↪ order:false, stats:pseudo |
+-----------------------+---------+-----------+-----------------------+-----------------------------------
    ↪
2 rows in set (0.00 sec)
```

12.11.2.2.3 MySQL 兼容性

- 不支持 FULLTEXT，HASH 和 SPATIAL 索引。
- 不支持 VISIBLE/INVISIBLE 索引（目前只有 master 分支上真正支持此功能）。
- 不支持降序索引（类似于 MySQL 5.7）。
- 目前尚不支持在一条中同时添加多个索引。
- 默认无法向表中添加 PRIMARY KEY，在开启 alter-primary-key 配置项后可支持此功能，详情可参考：alter-primary-key。

12.11.2.2.4 另请参阅

- 索引的选择
- 错误索引的解决方案
- CREATE INDEX
- DROP INDEX
- RENAME INDEX
- ADD COLUMN
- CREATE TABLE
- EXPLAIN

12.11.2.3 ADMIN

ADMIN 语句是 TiDB 扩展语法，用于查看 TiDB 自身的状态，并对 TiDB 中的表数据进行校验。

12.11.2.3.1 ADMIN 与 DDL 相关的扩展语句

语句	功能描述
ADMIN CANCEL DDL JOBS	取消当前正在运行的 DDL 作业

语句	功能描述
ADMIN CHECKSUM TABLE	计算表中所有行和索引的 CRC64 校验和
[ADMIN CHECK [TABLE|INDEX]](#admin-check-[table	index])
[ADMIN SHOW DDL [JOBS|QUERIES]](#admin-show-ddl-[jobs	queries])

12.11.2.3.2 admin reload 语句

```
ADMIN RELOAD expr_pushdown_blacklist;
```

以上语句用于重新加载表达式下推的黑名单。

```
ADMIN RELOAD opt_rule_blacklist;
```

以上语句用于重新加载逻辑优化规则的黑名单。

12.11.2.3.3 admin plugin 语句

```
ADMIN PLUGINS ENABLE plugin_name [, plugin_name] ...;
```

以上语句用于启用 plugin_name 插件。

```
ADMIN PLUGINS DISABLE plugin_name [, plugin_name] ...;
```

以上语句用于禁用 plugin_name 插件。

12.11.2.3.4 admin ... bindings 语句

```
ADMIN FLUSH bindings;
```

以上语句用于持久化 SQL Plan 绑定的信息。

```
ADMIN CAPTURE bindings;
```

以上语句可以将出现超过一次的 selectexecution-plan 语句生成 SQL Plan 的绑定。

```
ADMIN EVOLVE bindings;
```

开启自动绑定功能后，每隔 bind-info-lease（默认值为 3s）触发一次 SQL Plan 绑定信息的演进。以上语句用于主动触发此演进，SQL Plan 绑定详情可参考：执行计划管理。

```
ADMIN RELOAD bindings;
```

以上语句用于重新加载 SQL Plan 绑定的信息。

12.11.2.3.5 admin repair table 语句

```
ADMIN REPAIR TABLE tbl_name CREATE TABLE STATEMENT;
```

ADMIN REPAIR TABLE tbl_name CREATE TABLE STATEMENT 用于在极端情况下，对存储层中的表的元信息进行非可信的覆盖。“非可信”是指需要人为保证原表的元信息可以完全由 CREATE TABLE STATEMENT 提供。该语句需要打开配置文件项中的repair-mode 开关，并且需要确保所修复的表名在repair-table-list 名单中。

12.11.2.3.6 admin show slow 语句

```
ADMIN SHOW SLOW RECENT N;
```

```
ADMIN SHOW SLOW TOP [INTERNAL | ALL] N;
```

这两种语句的具体操作详情可参考：admin show slow 语句。

12.11.2.3.7 语句概览

```
AdminStmt ::=
    'ADMIN' ( 'SHOW' ( 'DDL' ( 'JOBS' Int64Num? WhereClauseOptional | 'JOB' 'QUERIES' NumList )?
        ↪ | TableName 'NEXT_ROW_ID' | 'SLOW' AdminShowSlow ) | 'CHECK' ( 'TABLE' TableNameList
        ↪ | 'INDEX' TableName Identifier ( HandleRange ( ',' HandleRange )* )? ) | 'RECOVER' '
        ↪ INDEX' TableName Identifier | 'CLEANUP' ( 'INDEX' TableName Identifier | 'TABLE' '
        ↪ LOCK' TableNameList ) | 'CHECKSUM' 'TABLE' TableNameList | 'CANCEL' 'DDL' 'JOBS'
        ↪ NumList | 'RELOAD' ( 'EXPR_PUSHDOWN_BLACKLIST' | 'OPT_RULE_BLACKLIST' | 'BINDINGS' )
        ↪ | 'PLUGINS' ( 'ENABLE' | 'DISABLE' ) PluginNameList | 'REPAIR' 'TABLE' TableName
        ↪ CreateTableStmt | ( 'FLUSH' | 'CAPTURE' | 'EVOLVE' ) 'BINDINGS' )
```

12.11.2.3.8 使用示例

执行以下命令，可查看正在执行的 DDL 任务中最近 10 条已经完成的 DDL 任务。未指定 NUM 时，默认只显示最近 10 条已经执行完的 DDL 任务。

```
admin show ddl jobs;
```

```
+--------+---------+------------+---------------+----------------------+----------+----------+-----------+------
    ↪
| JOB_ID | DB_NAME | TABLE_NAME | JOB_TYPE      | SCHEMA_STATE         | SCHEMA_ID | TABLE_ID |
    ↪ ROW_COUNT | START_TIME                        | END_TIME                           | STATE
    ↪           |
+--------+---------+------------+---------------+----------------------+----------+----------+-----------+------
    ↪
| 45     | test    | t1         | add index     | write reorganization | 32        | 37       | 0
    ↪           | 2019-01-10 12:38:36.501 +0800 CST |                                    |
    ↪ running       |
```

```
| 44     | test    | t1          | add index      | none                 | 32        | 37       | 0
    ↪           | 2019-01-10 12:36:55.18 +0800 CST  | 2019-01-10 12:36:55.852 +0800 CST |
    ↪ rollback done |
| 43     | test    | t1          | add index      | public               | 32        | 37       | 6
    ↪           | 2019-01-10 12:35:13.66 +0800 CST  | 2019-01-10 12:35:14.925 +0800 CST | synced
    ↪          |
| 42     | test    | t1          | drop index     | none                 | 32        | 37       | 0
    ↪           | 2019-01-10 12:34:35.204 +0800 CST | 2019-01-10 12:34:36.958 +0800 CST | synced
    ↪          |
| 41     | test    | t1          | add index      | public               | 32        | 37       | 0
    ↪           | 2019-01-10 12:33:22.62 +0800 CST  | 2019-01-10 12:33:24.625 +0800 CST | synced
    ↪          |
| 40     | test    | t1          | drop column    | none                 | 32        | 37       | 0
    ↪           | 2019-01-10 12:33:08.212 +0800 CST | 2019-01-10 12:33:09.78 +0800 CST  | synced
    ↪          |
| 39     | test    | t1          | add column     | public               | 32        | 37       | 0
    ↪           | 2019-01-10 12:32:55.42 +0800 CST  | 2019-01-10 12:32:56.24 +0800 CST  | synced
    ↪          |
| 38     | test    | t1          | create table   | public               | 32        | 37       | 0
    ↪           | 2019-01-10 12:32:41.956 +0800 CST | 2019-01-10 12:32:43.956 +0800 CST | synced
    ↪          |
| 36     | test    |             | drop table     | none                 | 32        | 34       | 0
    ↪           | 2019-01-10 11:29:59.982 +0800 CST | 2019-01-10 11:30:00.45 +0800  CST | synced
    ↪          |
| 35     | test    |             | create table   | public               | 32        | 34       | 0
    ↪           | 2019-01-10 11:29:40.741 +0800 CST | 2019-01-10 11:29:41.682 +0800 CST | synced
    ↪          |
| 33     | test    |             | create schema  | public               | 32        | 0        | 0
    ↪           | 2019-01-10 11:29:22.813 +0800 CST | 2019-01-10 11:29:23.954 +0800 CST | synced
    ↪          |
+--------+---------+------------+--------------------+----------------+-----------+----------+-----------+------
    ↪
```

执行以下命令，可查看正在执行的 DDL 任务中最近 5 条已经执行完的 DDL 任务：

```
admin show ddl jobs 5;
```

```
+--------+---------+------------+---------------+----------------------+-----------+----------+-----------+------
    ↪
| JOB_ID | DB_NAME | TABLE_NAME | JOB_TYPE      | SCHEMA_STATE         | SCHEMA_ID | TABLE_ID |
    ↪ ROW_COUNT | START_TIME                        | END_TIME                           | STATE
    ↪           |
+--------+---------+------------+---------------+----------------------+-----------+----------+-----------+------
    ↪
| 45     | test    | t1          | add index     | write reorganization | 32        | 37       | 0
    ↪           | 2019-01-10 12:38:36.501 +0800 CST |                                    |
```

```
    ↪ running       |
| 44     | test    | t1         | add index     | none                 | 32        | 37       | 0
    ↪          | 2019-01-10 12:36:55.18 +0800 CST  | 2019-01-10 12:36:55.852 +0800 CST |
    ↪ rollback done |
| 43     | test    | t1         | add index     | public               | 32        | 37       | 6
    ↪          | 2019-01-10 12:35:13.66 +0800 CST  | 2019-01-10 12:35:14.925 +0800 CST | synced
    ↪         |
| 42     | test    | t1         | drop index    | none                 | 32        | 37       | 0
    ↪          | 2019-01-10 12:34:35.204 +0800 CST | 2019-01-10 12:34:36.958 +0800 CST | synced
    ↪         |
| 41     | test    | t1         | add index     | public               | 32        | 37       | 0
    ↪          | 2019-01-10 12:33:22.62 +0800 CST  | 2019-01-10 12:33:24.625 +0800 CST | synced
    ↪         |
| 40     | test    | t1         | drop column   | none                 | 32        | 37       | 0
    ↪          | 2019-01-10 12:33:08.212 +0800 CST | 2019-01-10 12:33:09.78 +0800 CST  | synced
    ↪         |
+--------+---------+------------+---------------+----------------------+-----------+----------+-----------+------
    ↪
```

执行以下命令，可查看 test 数据库中未执行完成的 DDL 任务，包括正在执行中以及最近 5 条已经执行完但是执行失败的 DDL 任务。

```
admin show ddl jobs 5 where state!='synced' and db_name='test';
```

```
+--------+---------+------------+---------------+----------------------+-----------+----------+-----------+------
    ↪
| JOB_ID | DB_NAME | TABLE_NAME | JOB_TYPE      | SCHEMA_STATE         | SCHEMA_ID | TABLE_ID |
    ↪ ROW_COUNT | START_TIME                        | END_TIME                          | STATE
    ↪          |
+--------+---------+------------+---------------+----------------------+-----------+----------+-----------+------
    ↪
| 45     | test    | t1         | add index     | write reorganization | 32        | 37       | 0
    ↪          | 2019-01-10 12:38:36.501 +0800 CST |                                   |
    ↪ running       |
| 44     | test    | t1         | add index     | none                 | 32        | 37       | 0
    ↪          | 2019-01-10 12:36:55.18 +0800 CST  | 2019-01-10 12:36:55.852 +0800 CST |
    ↪ rollback done |
+--------+---------+------------+---------------+----------------------+-----------+----------+-----------+------
    ↪
```

- JOB_ID：每个 DDL 操作对应一个 DDL 作业，JOB_ID 全局唯一。
- DB_NAME：执行 DDL 操作的数据库的名称。
- TABLE_NAME：执行 DDL 操作的表的名称。
- JOB_TYPE：DDL 操作的类型。

- SCHEMA_STATE：schema 的当前状态。如果 JOB_TYPE 是 add index，则为 index 的状态；如果是 add column，则为 column 的状态，如果是 create table，则为 table 的状态。常见的状态有以下几种：
  - none：表示不存在。一般 drop 操作或者 create 操作失败回滚后，会变为 none 状态。
  - delete only、write only、delete reorganization、write reorganization：这四种状态是中间状态。由于中间状态转换很快，一般操作中看不到这几种状态，只有执行 add index 操作时能看到处于 write reorganization 状态，表示正在添加索引数据。
  - public：表示存在且可用。一般 create table 和 add index/column 等操作完成后，会变为 public 状态，表示新建的 table/column/index 可以正常读写了。
- SCHEMA_ID：执行 DDL 操作的数据库的 ID。
- TABLE_ID：执行 DDL 操作的表的 ID。
- ROW_COUNT：执行 add index 操作时，当前已经添加完成的数据行数。
- START_TIME：DDL 操作的开始时间。
- END_TIME：DDL 操作的结束时间。
- STATE：DDL 操作的状态。常见的状态有以下几种：
  - none：表示该操作任务已经进入 DDL 作业队列中，但尚未执行，因为还在排队等待前面的 DDL 作业完成。另一种原因可能是执行 drop 操作后，会变为 none 状态，但是很快会更新为 synced 状态，表示所有 TiDB 实例都已经同步到该状态。
  - running：表示该操作正在执行。
  - synced：表示该操作已经执行成功，且所有 TiDB 实例都已经同步该状态。
  - rollback done：表示该操作执行失败，回滚完成。
  - rollingback：表示该操作执行失败，正在回滚。
  - cancelling：表示正在取消该操作。这个状态只有在用 ADMIN CANCEL DDL JOBS 命令取消 DDL 作业时才会出现。

12.11.2.3.9 MySQL 兼容性

ADMIN 语句是 TiDB 对于 MySQL 语法的扩展。

12.11.2.4 ADMIN CANCEL DDL

ADMIN CANCEL DDL 语句用于取消当前正在运行的 DDL 作业。可以通过 ADMIN SHOW DDL JOBS 语句获取 DDL 作业的 job_id。

12.11.2.4.1 语法图

```
AdminStmt ::=
    'ADMIN' ( 'SHOW' ( 'DDL' ( 'JOBS' Int64Num? WhereClauseOptional | 'JOB' 'QUERIES' NumList )?
        ↪ | TableName 'NEXT_ROW_ID' | 'SLOW' AdminShowSlow ) | 'CHECK' ( 'TABLE' TableNameList
        ↪ | 'INDEX' TableName Identifier ( HandleRange ( ',' HandleRange )* )? ) | 'RECOVER' '
        ↪ INDEX' TableName Identifier | 'CLEANUP' ( 'INDEX' TableName Identifier | 'TABLE' '
        ↪ LOCK' TableNameList ) | 'CHECKSUM' 'TABLE' TableNameList | 'CANCEL' 'DDL' 'JOBS'
        ↪ NumList | 'RELOAD' ( 'EXPR_PUSHDOWN_BLACKLIST' | 'OPT_RULE_BLACKLIST' | 'BINDINGS' )
        ↪ | 'PLUGINS' ( 'ENABLE' | 'DISABLE' ) PluginNameList | 'REPAIR' 'TABLE' TableName
        ↪ CreateTableStmt | ( 'FLUSH' | 'CAPTURE' | 'EVOLVE' ) 'BINDINGS' )
```

```
NumList ::=
    Int64Num ( ',' Int64Num )*
```

12.11.2.4.2 示例

可以通过 ADMIN CANCEL DDL JOBS 语句取消当前正在运行的 DDL 作业，并返回对应作业是否取消成功：

```
ADMIN CANCEL DDL JOBS job_id [, job_id] ...;
```

如果取消失败，会显示失败的具体原因。

> 注意：
>
> - 只有该操作可以取消 DDL 作业，其他所有的操作和环境变更（例如机器重启、集群重启）都不会取消 DDL 作业。
> - 该操作可以同时取消多个 DDL 作业，可以通过 ADMIN SHOW DDL JOBS 语句来获取 DDL 作业的 job_id。
> - 如果希望取消的作业已经执行完毕，取消操作将失败。

12.11.2.4.3 MySQL 兼容性

ADMIN CANCEL DDL 语句是 TiDB 对 MySQL 语法的扩展。

12.11.2.4.4 另请参阅

- ADMIN SHOW DDL [JOBS|QUERIES]

12.11.2.5 ADMIN CHECKSUM TABLE

ADMIN CHECKSUM TABLE 语句用于计算表中所有行和索引的 CRC64 校验和。在 TiDB Lightning 等程序中，可通过此语句来确保导入操作成功。

12.11.2.5.1 语法图

```
AdminStmt ::=
    'ADMIN' ( 'SHOW' ( 'DDL' ( 'JOBS' Int64Num? WhereClauseOptional | 'JOB' 'QUERIES' NumList )?
        ↪ | TableName 'NEXT_ROW_ID' | 'SLOW' AdminShowSlow ) | 'CHECK' ( 'TABLE' TableNameList
        ↪ | 'INDEX' TableName Identifier ( HandleRange ( ',' HandleRange )* )? ) | 'RECOVER' '
        ↪ INDEX' TableName Identifier | 'CLEANUP' ( 'INDEX' TableName Identifier | 'TABLE' '
        ↪ LOCK' TableNameList ) | 'CHECKSUM' 'TABLE' TableNameList | 'CANCEL' 'DDL' 'JOBS'
        ↪ NumList | 'RELOAD' ( 'EXPR_PUSHDOWN_BLACKLIST' | 'OPT_RULE_BLACKLIST' | 'BINDINGS' )
        ↪ | 'PLUGINS' ( 'ENABLE' | 'DISABLE' ) PluginNameList | 'REPAIR' 'TABLE' TableName
        ↪ CreateTableStmt | ( 'FLUSH' | 'CAPTURE' | 'EVOLVE' ) 'BINDINGS' )
```

```
TableNameList ::=
    TableName ( ',' TableName )*
```

12.11.2.5.2 示例

计算表 t1 的校验和：

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY auto_increment);
INSERT INTO t1 VALUES (1),(2),(3);
ADMIN CHECKSUM TABLE t1;
```

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY auto_increment);
Query OK, 0 rows affected (0.11 sec)

INSERT INTO t1 VALUES (1),(2),(3);
Query OK, 3 rows affected (0.02 sec)
Records: 3  Duplicates: 0  Warnings: 0

ADMIN CHECKSUM TABLE t1;
+---------+------------+----------------------+-----------+-------------+
| Db_name | Table_name | Checksum_crc64_xor   | Total_kvs | Total_bytes |
+---------+------------+----------------------+-----------+-------------+
| test    | t1         | 10909174369497628533 |         3 |          75 |
+---------+------------+----------------------+-----------+-------------+
1 row in set (0.00 sec)
```

12.11.2.5.3 MySQL 兼容性

ADMIN CHECKSUM TABLE 语句是 TiDB 对 MySQL 语法的扩展。

12.11.2.6 ADMIN CHECK [TABLE|INDEX]

ADMIN CHECK [TABLE|INDEX] 语句用于校验表中数据和对应索引的一致性。

12.11.2.6.1 语法图

```
AdminStmt ::=
    'ADMIN' ( 'SHOW' ( 'DDL' ( 'JOBS' Int64Num? WhereClauseOptional | 'JOB' 'QUERIES' NumList )?
        ↪ | TableName 'NEXT_ROW_ID' | 'SLOW' AdminShowSlow ) | 'CHECK' ( 'TABLE' TableNameList
        ↪ | 'INDEX' TableName Identifier ( HandleRange ( ',' HandleRange )* )? ) | 'RECOVER' '
        ↪ INDEX' TableName Identifier | 'CLEANUP' ( 'INDEX' TableName Identifier | 'TABLE' '
        ↪ LOCK' TableNameList ) | 'CHECKSUM' 'TABLE' TableNameList | 'CANCEL' 'DDL' 'JOBS'
        ↪ NumList | 'RELOAD' ( 'EXPR_PUSHDOWN_BLACKLIST' | 'OPT_RULE_BLACKLIST' | 'BINDINGS' )
        ↪ | 'PLUGINS' ( 'ENABLE' | 'DISABLE' ) PluginNameList | 'REPAIR' 'TABLE' TableName
        ↪ CreateTableStmt | ( 'FLUSH' | 'CAPTURE' | 'EVOLVE' ) 'BINDINGS' )
```

```
TableNameList ::=
    TableName ( ',' TableName )*
```

12.11.2.6.2 示例

可以通过 ADMIN CHECK TABLE 语句校验 tbl_name 表中所有数据和对应索引的一致性：

```
ADMIN CHECK TABLE tbl_name [, tbl_name] ...;
```

若通过一致性校验，则返回空的查询结果；否则返回数据不一致的错误信息。

```
ADMIN CHECK INDEX tbl_name idx_name;
```

以上语句用于对 tbl_name 表中 idx_name 索引对应列数据和索引数据进行一致性校验。若通过校验，则返回空的查询结果；否则返回数据不一致的错误信息。

```
ADMIN CHECK INDEX tbl_name idx_name (lower_val, upper_val) [, (lower_val, upper_val)] ...;
```

以上语句用于对 tbl_name 表中 idx_name 索引对应列数据和索引数据进行一致性校验，并且指定了需要检查的数据范围。若通过校验，则返回空的查询结果；否则返回数据不一致的错误信息。

12.11.2.6.3 MySQL 兼容性

ADMIN CHECK [TABLE|INDEX] 语句是 TiDB 对 MySQL 语法的扩展。

12.11.2.6.4 另请参阅

- ADMIN REPAIR

12.11.2.7 ADMIN SHOW DDL [JOBS|QUERIES]

ADMIN SHOW DDL [JOBS|QUERIES] 语句显示了正在运行和最近完成的 DDL 作业的信息。

12.11.2.7.1 语法图

```
AdminStmt ::=
    'ADMIN' ( 'SHOW' ( 'DDL' ( 'JOBS' Int64Num? WhereClauseOptional | 'JOB' 'QUERIES' NumList )?
        ↪ | TableName 'NEXT_ROW_ID' | 'SLOW' AdminShowSlow ) | 'CHECK' ( 'TABLE' TableNameList
        ↪ | 'INDEX' TableName Identifier ( HandleRange ( ',' HandleRange )* )? ) | 'RECOVER' '
        ↪ INDEX' TableName Identifier | 'CLEANUP' ( 'INDEX' TableName Identifier | 'TABLE' '
        ↪ LOCK' TableNameList ) | 'CHECKSUM' 'TABLE' TableNameList | 'CANCEL' 'DDL' 'JOBS'
        ↪ NumList | 'RELOAD' ( 'EXPR_PUSHDOWN_BLACKLIST' | 'OPT_RULE_BLACKLIST' | 'BINDINGS' )
        ↪ | 'PLUGINS' ( 'ENABLE' | 'DISABLE' ) PluginNameList | 'REPAIR' 'TABLE' TableName
        ↪ CreateTableStmt | ( 'FLUSH' | 'CAPTURE' | 'EVOLVE' ) 'BINDINGS' )
```

```
NumList ::=
    Int64Num ( ',' Int64Num )*

WhereClauseOptional ::=
    WhereClause?
```

12.11.2.7.2 示例

ADMIN SHOW DDL

可以通过 ADMIN SHOW DDL 语句查看当前正在运行的 DDL 作业：

```
ADMIN SHOW DDL;
```

```
ADMIN SHOW DDL;
+--
    ↪ ----------+--------------------------------------+--------------+--------------+-----------------------
    ↪
| SCHEMA_VER | OWNER_ID                             | OWNER_ADDRESS | RUNNING_JOBS | SELF_ID
    ↪                                | QUERY |
+--
    ↪ ----------+--------------------------------------+--------------+--------------+-----------------------
    ↪
|         26 | 2d1982af-fa63-43ad-a3d5-73710683cc63 | 0.0.0.0:4000  |              | 2d1982af-
    ↪ fa63-43ad-a3d5-73710683cc63 |       |
+--
    ↪ ----------+--------------------------------------+--------------+--------------+-----------------------
    ↪
1 row in set (0.00 sec)
```

ADMIN SHOW DDL JOBS

ADMIN SHOW DDL JOBS 语句用于查看当前 DDL 作业队列中的所有结果（包括正在运行以及等待运行的任务）以及已执行完成的 DDL 作业队列中的最近十条结果。

```
ADMIN SHOW DDL JOBS;
```

```
ADMIN SHOW DDL JOBS;
+--
    ↪ ------+---------+-------------------+--------------+---------------------+-----------+----------+------
    ↪
| JOB_ID | DB_NAME | TABLE_NAME          | JOB_TYPE     | SCHEMA_STATE        | SCHEMA_ID |
    ↪ TABLE_ID | ROW_COUNT | START_TIME          | END_TIME            | STATE    |
+--
    ↪ ------+---------+-------------------+--------------+---------------------+-----------+----------+------
    ↪
```

```
|     59 | test    | t1                 | add index     | write reorganization |         1 |
    ↪       55 |     88576 | 2020-08-17 07:51:58 | NULL                | running |
|     60 | test    | t2                 | add index     | none                 |         1 |
    ↪       57 |         0 | 2020-08-17 07:51:59 | NULL                | none    |
|     58 | test    | t2                 | create table | public               |         1 |
    ↪       57 |         0 | 2020-08-17 07:41:28 | 2020-08-17 07:41:28 | synced  |
|     56 | test    | t1                 | create table | public               |         1 |
    ↪       55 |         0 | 2020-08-17 07:41:02 | 2020-08-17 07:41:02 | synced  |
|     54 | test    | t1                 | drop table    | none                 |         1 |
    ↪       50 |         0 | 2020-08-17 07:41:02 | 2020-08-17 07:41:02 | synced  |
|     53 | test    | t1                 | drop index    | none                 |         1 |
    ↪       50 |         0 | 2020-08-17 07:35:44 | 2020-08-17 07:35:44 | synced  |
|     52 | test    | t1                 | add index     | public               |         1 |
    ↪       50 |    451010 | 2020-08-17 07:34:43 | 2020-08-17 07:35:16 | synced  |
|     51 | test    | t1                 | create table | public               |         1 |
    ↪       50 |         0 | 2020-08-17 07:34:02 | 2020-08-17 07:34:02 | synced  |
|     49 | test    | t1                 | drop table    | none                 |         1 |
    ↪       47 |         0 | 2020-08-17 07:34:02 | 2020-08-17 07:34:02 | synced  |
|     48 | test    | t1                 | create table | public               |         1 |
    ↪       47 |         0 | 2020-08-17 07:33:37 | 2020-08-17 07:33:37 | synced  |
|     46 | mysql   | stats_extended     | create table | public               |         3 |
    ↪       45 |         0 | 2020-08-17 06:42:38 | 2020-08-17 06:42:38 | synced  |
|     44 | mysql   | opt_rule_blacklist | create table | public               |         3 |
    ↪       43 |         0 | 2020-08-17 06:42:38 | 2020-08-17 06:42:38 | synced  |
+--
    ↪ ------+---------+--------------------+--------------+----------------------+-----------+----------+------
    ↪
12 rows in set (0.00 sec)
```

由上述 ADMIN 查询结果可知：

- job_id 为 59 的 DDL 作业当前正在进行中（STATE 列显示为 running）。SCHEMA_STATE 列显示了表当前处于 write reorganization 状态，一旦任务完成，将更改为 public，以便用户会话可以公开观察到状态变更。end_time 列显示为 NULL，表明当前作业的完成时间未知。
- job_id 为 60 的 JOB_TYPE 显示为 add index，表明正在排队等待 job_id 为 59 的作业完成。当作业 59 完成时，作业 60 的 STATE 将更改为 running。
- 对于破坏性的更改（例如删除索引或删除表），当作业完成时，SCHEMA_STATE 将变为 none。对于附加更改，SCHEMA_STATE 将变为 public。

若要限制表中显示的行数，可以指定 NUM 和 WHERE 条件：

```
ADMIN SHOW DDL JOBS [NUM] [WHERE where_condition];
```

- NUM：用于查看已经执行完成的 DDL 作业队列中最近 NUM 条结果；未指定时，默认值为 10。

- WHERE：WHERE 子句，用于添加过滤条件。

ADMIN SHOW DDL JOB QUERIES

ADMIN SHOW DDL JOB QUERIES 语句用于查看 job_id 对应的 DDL 任务的原始 SQL 语句：

```
ADMIN SHOW DDL JOBS;
ADMIN SHOW DDL JOB QUERIES 51;
```

```
ADMIN SHOW DDL JOB QUERIES 51;
+--------------------------------------------------------------+
| QUERY                                                        |
+--------------------------------------------------------------+
| CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY auto_increment) |
+--------------------------------------------------------------+
1 row in set (0.02 sec)
```

只能在 DDL 历史作业队列中最近十条结果中搜索与 job_id 对应的正在运行中的 DDL 作业。

12.11.2.7.3 MySQL 兼容性

ADMIN SHOW DDL [JOBS|QUERIES] 语句是 TiDB 对 MySQL 语法的扩展。

12.11.2.7.4 另请参阅

- ADMIN CANCEL DDL

12.11.2.8 ALTER DATABASE

ALTER DATABASE 用于修改指定或当前数据库的默认字符集和排序规则。ALTER SCHEMA 跟 ALTER DATABASE 操作效果一样。

12.11.2.8.1 语法图

```
AlterDatabaseStmt ::=
    'ALTER' 'DATABASE' DBName? DatabaseOptionList

DatabaseOption ::=
    DefaultKwdOpt ( CharsetKw '='? CharsetName | 'COLLATE' '='? CollationName | 'ENCRYPTION' '='?
        ↪ EncryptionOpt )
```

12.11.2.8.2 示例

修改数据库 test 的字符集为 utf8mb4：

```
ALTER DATABASE test DEFAULT CHARACTER SET = utf8mb4;
```

```
Query OK, 0 rows affected (0.00 sec)
```

目前 TiDB 只支持部分的字符集和排序规则，详情参阅字符集支持。

12.11.2.8.3 MySQL 兼容性

ALTER DATABASE 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.8.4 另请参阅

- CREATE DATABASE
- SHOW DATABASES

12.11.2.9 ALTER INSTANCE

ALTER INSTANCE 语句用于对单个 TiDB 实例进行变更操作。目前 TiDB 仅支持 RELOAD TLS 子句。

12.11.2.9.1 RELOAD TLS

ALTER INSTANCE RELOAD TLS 语句用于从原配置的证书 (ssl-cert)、密钥 (ssl-key) 和 CA (ssl-ca) 的路径重新加证书、密钥和 CA。

新加载的证书密钥和 CA 将在语句执行成功后对新建立的连接生效，不会影响语句执行前已建立的连接。

在重加载遇到错误时默认会报错返回且继续使用变更前的密钥和证书，但在添加可选的 NO ROLLBACK ON ERROR 后遇到错误将不报错并以关闭 TLS 安全连接功能的方式处理后续请求。

12.11.2.9.2 语法图

```
AlterInstanceStmt ::=
    'ALTER' 'INSTANCE' InstanceOption

InstanceOption ::=
    'RELOAD' 'TLS' ('NO' 'ROLLBACK' 'ON' 'ERROR')?
```

12.11.2.9.3 示例

```
ALTER INSTANCE RELOAD TLS;
```

12.11.2.9.4 MySQL 兼容性

仅支持从原配置路径重加载，不支持动态修改加载路径，也不支持动态启用启动 TiDB 时未开启的 TLS 加密连接功能。

12.11.2.9.5 另请参阅

为 TiDB 客户端服务端间通信开启加密传输

12.11.2.10 ALTER TABLE

ALTER TABLE 语句用于对已有表进行修改，以符合新表结构。ALTER TABLE 语句可用于：

- ADD，DROP，或RENAME 索引
- ADD，DROP，MODIFY 或CHANGE 列

12.11.2.10.1 语法图

```
AlterTableStmt ::=
    'ALTER' IgnoreOptional 'TABLE' TableName ( AlterTableSpecListOpt AlterTablePartitionOpt | '
        ↪ ANALYZE' 'PARTITION' PartitionNameList ( 'INDEX' IndexNameList )?
        ↪ AnalyzeOptionListOpt )

TableName ::=
    Identifier ('.' Identifier)?

AlterTableSpec ::=
    TableOptionList
|   'SET' 'TIFLASH' 'REPLICA' LengthNum LocationLabelList
|   'CONVERT' 'TO' CharsetKw ( CharsetName | 'DEFAULT' ) OptCollate
|   'ADD' ( ColumnKeywordOpt IfNotExists ( ColumnDef ColumnPosition | '(' TableElementList ')' )
    ↪ | Constraint | 'PARTITION' IfNotExists NoWriteToBinLogAliasOpt (
    ↪ PartitionDefinitionListOpt | 'PARTITIONS' NUM ) )
|   ( ( 'CHECK' | 'TRUNCATE' ) 'PARTITION' | ( 'OPTIMIZE' | 'REPAIR' | 'REBUILD' ) 'PARTITION'
    ↪ NoWriteToBinLogAliasOpt ) AllOrPartitionNameList
|   'COALESCE' 'PARTITION' NoWriteToBinLogAliasOpt NUM
|   'DROP' ( ColumnKeywordOpt IfExists ColumnName RestrictOrCascadeOpt | 'PRIMARY' 'KEY' |  '
    ↪ PARTITION' IfExists PartitionNameList | ( KeyOrIndex IfExists | 'CHECK' ) Identifier | '
    ↪ FOREIGN' 'KEY' IfExists Symbol )
|   'EXCHANGE' 'PARTITION' Identifier 'WITH' 'TABLE' TableName WithValidationOpt
|   ( 'IMPORT' | 'DISCARD' ) ( 'PARTITION' AllOrPartitionNameList )? 'TABLESPACE'
|   'REORGANIZE' 'PARTITION' NoWriteToBinLogAliasOpt ReorganizePartitionRuleOpt
|   'ORDER' 'BY' AlterOrderItem ( ',' AlterOrderItem )*
|   ( 'DISABLE' | 'ENABLE' ) 'KEYS'
|   ( 'MODIFY' ColumnKeywordOpt IfExists | 'CHANGE' ColumnKeywordOpt IfExists ColumnName )
    ↪ ColumnDef ColumnPosition
|   'ALTER' ( ColumnKeywordOpt ColumnName ( 'SET' 'DEFAULT' ( SignedLiteral | '(' Expression ')'
    ↪ ) | 'DROP' 'DEFAULT' ) | 'CHECK' Identifier EnforcedOrNot | 'INDEX' Identifier
    ↪ IndexInvisible )
|   'RENAME' ( ( 'COLUMN' | KeyOrIndex ) Identifier 'TO' Identifier | ( 'TO' | '='? | 'AS' )
    ↪ TableName )
```

```
|   LockClause
|   AlgorithmClause
|   'FORCE'
|   ( 'WITH' | 'WITHOUT' ) 'VALIDATION'
|   'SECONDARY_LOAD'
|   'SECONDARY_UNLOAD'
```

12.11.2.10.2 示例

创建一张表，并插入初始数据：

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, c1 INT NOT NULL);
INSERT INTO t1 (c1) VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 0 rows affected (0.11 sec)
Query OK, 5 rows affected (0.03 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

执行以下查询需要扫描全表，因为 c1 列未被索引：

```
EXPLAIN SELECT * FROM t1 WHERE c1 = 3;
```

```
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
| id                    | estRows  | task      | access object | operator info
    ↪                  |
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
| TableReader_7           | 10.00    | root      |               | data:Selection_6
    ↪               |
| └─Selection_6           | 10.00    | cop[tikv] |               | eq(test.t1.c1, 3)
    ↪              |
|   └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t1      | keep order:false, stats:
    ↪ pseudo |
+--
    ↪ ----------------------+----------+-----------+---------------+------------------------------+
    ↪
3 rows in set (0.00 sec)
```

你可以使用ALTER TABLE .. ADD INDEX 语句在 t1 表上添加索引。添加后，EXPLAIN 的分析结果显示 SELECT * ↪ FROM t1 WHERE c1 = 3; 查询已使用效率更高的索引范围扫描：

```
ALTER TABLE t1 ADD INDEX (c1);
EXPLAIN SELECT * FROM t1 WHERE c1 = 3;
```

```
Query OK, 0 rows affected (0.30 sec)
+--
    ↪ ---------------------+---------+-----------+-----------------------+-------------------------------
    ↪
| id                     | estRows | task      | access object         | operator info
    ↪                                |
+--
    ↪ ---------------------+---------+-----------+-----------------------+-------------------------------
    ↪
| IndexReader_6          | 10.00   | root      |                       | index:IndexRangeScan_5
    ↪                       |
| └─IndexRangeScan_5     | 10.00   | cop[tikv] | table:t1, index:c1(c1) | range:[3,3], keep
    ↪ order:false, stats:pseudo |
+--
    ↪ ---------------------+---------+-----------+-----------------------+-------------------------------
    ↪
2 rows in set (0.00 sec)
```

TiDB 允许用户为 DDL 操作指定使用某一种 ALTER 算法。这仅为一种指定，并不改变实际的用于更改表的算法。如果你只想在群集的高峰时段允许即时 DDL 更改，则 ALTER 算法会很有用。示例如下：

```
ALTER TABLE t1 DROP INDEX c1, ALGORITHM=INSTANT;
```

```
Query OK, 0 rows affected (0.24 sec)
```

如果某一 DDL 操作要求使用 INPLACE 算法，而用户指定 ALGORITHM=INSTANT，会导致报错：

```
ALTER TABLE t1 ADD INDEX (c1), ALGORITHM=INSTANT;
```

```
ERROR 1846 (0A000): ALGORITHM=INSTANT is not supported. Reason: Cannot alter table by INSTANT.
    ↪ Try ALGORITHM=INPLACE.
```

但如果为 INPLACE 操作指定 ALGORITHM=COPY，会产生警告而非错误，这是因为 TiDB 将该指定解读为该算法或更好的算法。由于 TiDB 使用的算法可能不同于 MySQL，所以这一行为可用于 MySQL 兼容性。

```
ALTER TABLE t1 ADD INDEX (c1), ALGORITHM=COPY;
SHOW WARNINGS;
```

```
Query OK, 0 rows affected, 1 warning (0.25 sec)
+--
    ↪ -----+------+-----------------------------------------------------------------------------------------
    ↪
| Level | Code | Message
    ↪                                                                                                  |
+--
    ↪ -----+------+-----------------------------------------------------------------------------------------
    ↪
```

```
| Error | 1846 | ALGORITHM=COPY is not supported. Reason: Cannot alter table by COPY. Try
    ↪ ALGORITHM=INPLACE. |
+--
    ↪ -----+------+----------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

12.11.2.10.3 MySQL 兼容性

TiDB 中的 ALTER TABLE 语法主要存在以下限制：

- 单条 ALTER TABLE 语句不能完成多项操作。
- 当前不支持有损更改，例如从 BIGINT 类型更改为 INT 类型。
- 不支持空间数据类型。

其它限制可参考：TiDB 中 DDL 语句与 MySQL 的兼容性情况。

12.11.2.10.4 另请参阅

- 与 MySQL 兼容性对比
- ADD COLUMN
- DROP COLUMN
- ADD INDEX
- DROP INDEX
- RENAME INDEX
- CREATE TABLE
- DROP TABLE
- SHOW CREATE TABLE

12.11.2.11 ALTER USER

ALTER USER 语句用于更改 TiDB 权限系统内的已有用户。和 MySQL 一样，在 TiDB 权限系统中，用户是用户名和用户名所连接主机的组合。因此，可创建一个用户 'newuser2'@'192.168.1.1'，使其只能通过 IP 地址 192.168.1.1 进行连接。相同的用户名从不同主机登录时可能会拥有不同的权限。

12.11.2.11.1 语法图

```
AlterUserStmt ::=
    'ALTER' 'USER' IfExists (UserSpecList RequireClauseOpt ConnectionOptions
        ↪ PasswordOrLockOptions | 'USER' '(' ')' 'IDENTIFIED' 'BY' AuthString)

UserSpecList ::=
    UserSpec ( ',' UserSpec )*

UserSpec ::=
```

```
    Username AuthOption

Username ::=
    StringName ('@' StringName | singleAtIdentifier)? | 'CURRENT_USER' OptionalBraces

AuthOption ::=
    ( 'IDENTIFIED' ( 'BY' ( AuthString | 'PASSWORD' HashString ) | 'WITH' StringName ( 'BY'
        ↪ AuthString | 'AS' HashString )? ) )?
```

12.11.2.11.2 示例

```
CREATE USER 'newuser' IDENTIFIED BY 'newuserpassword';
```

```
Query OK, 1 row affected (0.01 sec)
```

```
SHOW CREATE USER 'newuser';
```

```
+-----------------------------------------------------------------------------------------------------------------
    ↪
| CREATE USER for newuser@%
    ↪
    ↪ |
+-----------------------------------------------------------------------------------------------------------------
    ↪
| CREATE USER 'newuser'@'%' IDENTIFIED WITH 'mysql_native_password' AS '*5806
    ↪ E04BBEE79E1899964C6A04D68BCA69B1A879' REQUIRE NONE PASSWORD EXPIRE DEFAULT ACCOUNT UNLOCK
    ↪  |
+-----------------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

```
ALTER USER 'newuser' IDENTIFIED BY 'newnewpassword';
```

```
Query OK, 0 rows affected (0.02 sec)
```

```
SHOW CREATE USER 'newuser';
```

```
+-----------------------------------------------------------------------------------------------------------------
    ↪
| CREATE USER for newuser@%
    ↪
    ↪ |
+-----------------------------------------------------------------------------------------------------------------
    ↪
```

```
| CREATE USER 'newuser'@'%' IDENTIFIED WITH 'mysql_native_password' AS '*
    ↪ FB8A1EA1353E8775CA836233E367FBDFCB37BE73' REQUIRE NONE PASSWORD EXPIRE DEFAULT ACCOUNT
    ↪ UNLOCK |
+-----------------------------------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

12.11.2.11.3 MySQL 兼容性

- 在 MySQL 中，ALTER 语句用于更改属性，例如使密码失效。但 TiDB 尚不支持此功能。

12.11.2.11.4 另请参阅

- Security Compatibility with MySQL
- CREATE USER
- DROP USER
- SHOW CREATE USER

12.11.2.12 ANALYZE

ANALYZE 语句用于更新 TiDB 在表和索引上留下的统计信息。执行大批量更新或导入记录后，或查询执行计划不是最佳时，建议运行 ANALYZE。

当 TiDB 逐渐发现这些统计数据与预估不一致时，也会自动更新其统计数据。

目前 TiDB 收集统计信息分为全量收集和增量收集两种方式，分别通过 ANALYZE TABLE 和 ANALYZE INCREMENTAL
↪ TABLE 语句来实现。关于这两种语句的详细使用方式，可参考统计信息简介。

12.11.2.12.1 语法图

```
AnalyzeTableStmt ::=
    'ANALYZE' ( 'TABLE' ( TableNameList | TableName ( 'INDEX' IndexNameList | 'PARTITION'
        ↪ PartitionNameList ( 'INDEX' IndexNameList )? ) ) | 'INCREMENTAL' 'TABLE' TableName (
        ↪ 'PARTITION' PartitionNameList )? 'INDEX' IndexNameList ) AnalyzeOptionListOpt

TableNameList ::=
    TableName (',' TableName)*

TableName ::=
    Identifier ( '.' Identifier )?
```

12.11.2.12.2 示例

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, c1 INT NOT NULL);
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
INSERT INTO t1 (c1) VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 5 rows affected (0.03 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
ALTER TABLE t1 ADD INDEX (c1);
```

```
Query OK, 0 rows affected (0.30 sec)
```

```
EXPLAIN SELECT * FROM t1 WHERE c1 = 3;
```

```
+-----------------------+---------+-----------+-----------------------+-----------------------------------------
    ↪
| id                    | estRows | task      | access object         | operator info
    ↪                               |
+-----------------------+---------+-----------+-----------------------+-----------------------------------------
    ↪
| IndexReader_6         | 10.00   | root      |                       | index:IndexRangeScan_5
    ↪                       |
| └─IndexRangeScan_5    | 10.00   | cop[tikv] | table:t1, index:c1(c1) | range:[3,3], keep
    ↪ order:false, stats:pseudo |
+-----------------------+---------+-----------+-----------------------+-----------------------------------------
    ↪
2 rows in set (0.00 sec)
```

```
analyze table t1;
```

```
Query OK, 0 rows affected (0.13 sec)
```

```
EXPLAIN SELECT * FROM t1 WHERE c1 = 3;
```

```
+-----------------------+---------+-----------+-----------------------+-----------------------------+
    ↪
| id                    | estRows | task      | access object         | operator info
    ↪                |
+-----------------------+---------+-----------+-----------------------+-----------------------------+
    ↪
| IndexReader_6         | 1.00    | root      |                       | index:IndexRangeScan_5
    ↪        |
```

```
| └─IndexRangeScan_5     | 1.00    | cop[tikv] | table:t1, index:c1(c1) | range:[3,3], keep
    ↪ order:false |
+------------------------+---------+-----------+------------------------+-----------------------------+
    ↪
2 rows in set (0.00 sec)
```

12.11.2.12.3 MySQL 兼容性

- ANALYZE TABLE 在语法上与 MySQL 类似。但 ANALYZE TABLE 在 TiDB 上的执行时间可能长得多，因为它的内部运行方式不同。
- 在 MySQL 上不支持 ANALYZE INCREMENTAL TABLE 语句，它的使用可参考增量收集文档。

TiDB 与 MySQL 在以下方面存在区别：所收集的统计信息，以及查询执行过程中统计信息是如何被使用的。虽然 TiDB 中的 ANALYZE 语句在语法上与 MySQL 类似，但存在以下差异：

- 执行 ANALYZE TABLE 时，TiDB 可能不包含最近提交的更改。若对行进行了批量更改，在执行 ANALYZE TABLE 之前，你可能需要先执行 `sleep(1)`，这样统计信息更新才能反映这些更改。参见 #16570。
- ANALYZE TABLE 在 TiDB 中的执行时间比在 MySQL 中的执行时间要长得多。但你可以通过执行 SET GLOBAL ↪ tidb_enable_fast_analyze=1 来启用快速分析，这样能部分抵消这种执行上的性能差异。快速分析利用了采样，会导致统计信息的准确性降低。因此快速分析仍是一项实验特性。

12.11.2.12.4 另请参阅

- EXPLAIN
- EXPLAIN ANALYZE

12.11.2.13 BACKUP

> **警告：**
>
> BACKUP 语句目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现 bug，请在 GitHub 上提 issue 反馈。

BACKUP 语句用于对 TiDB 集群执行分布式备份操作。

BACKUP 语句使用的引擎与BR 相同，但备份过程是由 TiDB 本身驱动，而非单独的 BR 工具。BR 工具的优势和警告也适用于 BACKUP 语句。

执行 BACKUP 需要 SUPER 权限。此外，执行备份的 TiDB 节点和集群中的所有 TiKV 节点都必须有对目标存储的读或写权限。

BACKUP 语句开始执行后将会被阻塞，直到整个备份任务完成、失败或取消。因此，执行 BACKUP 时需要准备一个持久的连接。如需取消任务，可执行KILL TIDB QUERY 语句。

一次只能执行一个 BACKUP 和RESTORE 任务。如果 TiDB server 上已经在执行一个 BACKUP 或 RESTORE 语句，新的 BACKUP 将等待前面所有的任务完成后再执行。

12.11.2.13.1 语法图

```
BackupStmt ::=
    "BACKUP" BRIETables "TO" stringLit BackupOption*

BRIETables ::=
    "DATABASE" ( '*' | DBName (',' DBName)* )
|   "TABLE" TableNameList

BackupOption ::=
    "RATE_LIMIT" '='? LengthNum "MB" '/' "SECOND"
|   "CONCURRENCY" '='? LengthNum
|   "CHECKSUM" '='? Boolean
|   "SEND_CREDENTIALS_TO_TIKV" '='? Boolean
|   "LAST_BACKUP" '='? BackupTSO
|   "SNAPSHOT" '='? ( BackupTSO | LengthNum TimestampUnit "AGO" )

Boolean ::=
    NUM | "TRUE" | "FALSE"

BackupTSO ::=
    LengthNum | stringLit
```

12.11.2.13.2 示例

备份数据库

```
BACKUP DATABASE `test` TO 'local:///mnt/backup/2020/04/';
```

```
+--
    ↪ ---------------------------+-----------+-----------------+---------------------+---------------------+
    ↪
| Destination                  | Size      | BackupTS        | Queue Time          | Execution
    ↪ Time      |
+--
    ↪ ---------------------------+-----------+-----------------+---------------------+---------------------+
    ↪
| local:///mnt/backup/2020/04/ | 248665063 | 416099531454472 | 2020-04-12 23:09:48 | 2020-04-12
    ↪ 23:09:48 |
+--
    ↪ ---------------------------+-----------+-----------------+---------------------+---------------------+
    ↪
1 row in set (58.453 sec)
```

上述示例中，test 数据库被备份到本地，数据以 SST 文件的形式存储在分布于所有 TiDB 和 TiKV 节点的 /mnt/backup/2020/04/ 目录中。

输出结果的第一行描述如下：

列名	描述
Destination	目标存储的 URL
Size	备份文件的总大小，单位为字节
BackupTS	创建备份时的快照 TSO（用于增量备份）
Queue Time	BACKUP 任务开始排队的时间戳（当前时区）
Execution Time	BACKUP 任务开始执行的时间戳（当前时区）

备份表

```
BACKUP TABLE `test`.`sbtest01` TO 'local:///mnt/backup/sbtest01/';
```

```
BACKUP TABLE sbtest02, sbtest03, sbtest04 TO 'local:///mnt/backup/sbtest/';
```

备份集群

```
BACKUP DATABASE * TO 'local:///mnt/backup/full/';
```

注意，备份中不包含系统表 (mysql.*、INFORMATION_SCHEMA.*、PERFORMANCE_SCHEMA.* 等)。

外部存储

BR 支持备份数据到 Amazon S3 或 Google Cloud Storage (GCS)：

```
BACKUP DATABASE `test` TO 's3://example-bucket-2020/backup-05/?region=us-west-2&access-key={
    ↪ YOUR_ACCESS_KEY}&secret-access-key={YOUR_SECRET_KEY}';
```

有关详细的 URL 语法，见外部存储。

当运行在云环境中时，不能分发凭证，可设置 SEND_CREDENTIALS_TO_TIKV 选项为 FALSE：

```
BACKUP DATABASE `test` TO 's3://example-bucket-2020/backup-05/?region=us-west-2'
    SEND_CREDENTIALS_TO_TIKV = FALSE;
```

性能调优

如果你需要减少网络带宽占用，可以通过 RATE_LIMIT 来限制每个 TiKV 节点的平均上传速度。

默认情况下，每个 TiKV 节点上运行 4 个备份线程。可以通过 CONCURRENCY 选项来调整这个值。

在备份完成之前，BACKUP 将对集群上的数据进行校验，以验证数据的正确性。如果你确信无需进行校验，可以通过 CHECKSUM 选项禁用这一步骤。

```
BACKUP DATABASE `test` TO 's3://example-bucket-2020/backup-06/'
    RATE_LIMIT = 120 MB/SECOND
    CONCURRENCY = 8
    CHECKSUM = FALSE;
```

快照

可以指定一个时间戳、TSO 或相对时间，来备份历史数据。

```
-- 相对时间
BACKUP DATABASE `test` TO 'local:///mnt/backup/hist01'
    SNAPSHOT = 36 HOUR AGO;
-- 时间戳（当前时区）
BACKUP DATABASE `test` TO 'local:///mnt/backup/hist02'
    SNAPSHOT = '2020-04-01 12:00:00';
-- TSO
BACKUP DATABASE `test` TO 'local:///mnt/backup/hist03'
    SNAPSHOT = 415685305958400;
```

对于相对时间，支持以下时间单位：

- MICROSECOND（微秒）
- SECOND（秒）
- MINUTE（分钟）
- HOUR（小时）
- DAY（天）
- WEEK（周）

注意，相对时间的单位遵循 SQL 标准，永远使用单数。

增量备份

提供 LAST_BACKUP 选项，只备份从上一次备份到当前快照之间的增量数据。

```
-- 时间戳（当前时区）
BACKUP DATABASE `test` TO 'local:///mnt/backup/hist02'
    LAST_BACKUP = '2020-04-01 12:00:00';

-- TSO
BACKUP DATABASE `test` TO 'local:///mnt/backup/hist03'
    LAST_BACKUP = 415685305958400;
```

12.11.2.13.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

12.11.2.13.4 另请参阅

- RESTORE
- SHOW BACKUPS

12.11.2.14 BEGIN

BEGIN 语句用于在 TiDB 内启动一个新事务，类似于 START TRANSACTION 和 SET autocommit=0 语句。

在没有 BEGIN 语句的情况下，每个语句默认在各自的事务中自动提交，从而确保 MySQL 兼容性。

12.11.2.14.1 语法图

```
BeginTransactionStmt ::=
    'BEGIN' ( 'PESSIMISTIC' | 'OPTIMISTIC' )?
|   'START' 'TRANSACTION' ( 'READ' ( 'WRITE' | 'ONLY' ( 'WITH' 'TIMESTAMP' 'BOUND' TimestampBound
    ↪  )? ) | 'WITH' 'CONSISTENT' 'SNAPSHOT' )?
```

12.11.2.14.2 示例

```
CREATE TABLE t1 (a int NOT NULL PRIMARY KEY);
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
BEGIN;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
INSERT INTO t1 VALUES (1);
```

```
Query OK, 1 row affected (0.00 sec)
```

```
COMMIT;
```

```
Query OK, 0 rows affected (0.01 sec)
```

12.11.2.14.3 MySQL 兼容性

TiDB 支持 `BEGIN PESSIMISTIC` 或 `BEGIN OPTIMISTIC` 的语法扩展，用户可以为某一个事务覆盖默认的事务模型。

12.11.2.14.4 另请参阅

- COMMIT
- ROLLBACK
- START TRANSACTION
- TiDB 乐观事务模型
- TiDB 悲观事务模型

12.11.2.15 CHANGE COLUMN

`ALTER TABLE.. CHANGE COLUMN` 语句用于在已有表上更改列，包括对列进行重命名，和将数据改为兼容类型。

12.11.2.15.1 语法图

```
AlterTableStmt ::=
    'ALTER' IgnoreOptional 'TABLE' TableName ( AlterTableSpecListOpt AlterTablePartitionOpt | '
        ↪ ANALYZE' 'PARTITION' PartitionNameList ( 'INDEX' IndexNameList )?
        ↪ AnalyzeOptionListOpt )

AlterTableSpec ::=
    TableOptionList
|   'SET' 'TIFLASH' 'REPLICA' LengthNum LocationLabelList
|   'CONVERT' 'TO' CharsetKw ( CharsetName | 'DEFAULT' ) OptCollate
|   'ADD' ( ColumnKeywordOpt IfNotExists ( ColumnDef ColumnPosition | '(' TableElementList ')' )
    ↪ | Constraint | 'PARTITION' IfNotExists NoWriteToBinLogAliasOpt (
    ↪ PartitionDefinitionListOpt | 'PARTITIONS' NUM ) )
|   ( ( 'CHECK' | 'TRUNCATE' ) 'PARTITION' | ( 'OPTIMIZE' | 'REPAIR' | 'REBUILD' ) 'PARTITION'
    ↪ NoWriteToBinLogAliasOpt ) AllOrPartitionNameList
|   'COALESCE' 'PARTITION' NoWriteToBinLogAliasOpt NUM
|   'DROP' ( ColumnKeywordOpt IfExists ColumnName RestrictOrCascadeOpt | 'PRIMARY' 'KEY' | '
    ↪ PARTITION' IfExists PartitionNameList | ( KeyOrIndex IfExists | 'CHECK' ) Identifier | '
    ↪ FOREIGN' 'KEY' IfExists Symbol )
|   'EXCHANGE' 'PARTITION' Identifier 'WITH' 'TABLE' TableName WithValidationOpt
|   ( 'IMPORT' | 'DISCARD' ) ( 'PARTITION' AllOrPartitionNameList )? 'TABLESPACE'
|   'REORGANIZE' 'PARTITION' NoWriteToBinLogAliasOpt ReorganizePartitionRuleOpt
|   'ORDER' 'BY' AlterOrderItem ( ',' AlterOrderItem )*
|   ( 'DISABLE' | 'ENABLE' ) 'KEYS'
|   ( 'MODIFY' ColumnKeywordOpt IfExists | 'CHANGE' ColumnKeywordOpt IfExists ColumnName )
    ↪ ColumnDef ColumnPosition
|   'ALTER' ( ColumnKeywordOpt ColumnName ( 'SET' 'DEFAULT' ( SignedLiteral | '(' Expression ')'
    ↪ ) | 'DROP' 'DEFAULT' ) | 'CHECK' Identifier EnforcedOrNot | 'INDEX' Identifier
    ↪ IndexInvisible )
|   'RENAME' ( ( 'COLUMN' | KeyOrIndex ) Identifier 'TO' Identifier | ( 'TO' | '='? | 'AS' )
    ↪ TableName )
|   LockClause
|   AlgorithmClause
|   'FORCE'
|   ( 'WITH' | 'WITHOUT' ) 'VALIDATION'
|   'SECONDARY_LOAD'
|   'SECONDARY_UNLOAD'

ColumnName ::=
    Identifier ( '.' Identifier ( '.' Identifier )? )?

ColumnDef ::=
    ColumnName ( Type | 'SERIAL' ) ColumnOptionListOpt

ColumnPosition ::=
```

```
    ( 'FIRST' | 'AFTER' ColumnName )?
```

12.11.2.15.2 示例

```
CREATE TABLE t1 (id int not null primary key AUTO_INCREMENT, col1 INT);
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
INSERT INTO t1 (col1) VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 5 rows affected (0.02 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
ALTER TABLE t1 CHANGE col1 col2 INT;
```

```
Query OK, 0 rows affected (0.09 sec)
```

```
ALTER TABLE t1 CHANGE col2 col3 BIGINT, ALGORITHM=INSTANT;
```

```
Query OK, 0 rows affected (0.08 sec)
```

```
ALTER TABLE t1 CHANGE col3 col3 INT;
```

```
ERROR 1105 (HY000): unsupported modify column length 11 is less than origin 20
```

```
ALTER TABLE t1 CHANGE col3 col3 BLOB;
```

```
ERROR 1105 (HY000): unsupported modify column type 252 not match origin 8
```

```
ALTER TABLE t1 CHANGE col3 col4 BIGINT, CHANGE id id2 INT NOT NULL;
```

```
ERROR 1105 (HY000): can't run multi schema change
```

12.11.2.15.3 MySQL 兼容性

- 不支持在单个 `ALTER TABLE` 语句中进行多个更改。
- 不支持有损变更，比如从 `BIGINT` 变为 `INTEGER`，或者从 `VARCHAR(255)` 变为 `VARCHAR(10)`。
- 不支持修改 `DECIMAL` 类型的精度。
- 不支持更改 `UNSIGNED` 属性。

12.11.2.15.4 另请参阅

- CREATE TABLE
- SHOW CREATE TABLE
- ADD COLUMN
- DROP COLUMN
- MODIFY COLUMN

12.11.2.16 CHANGE DRAINER

CHANGE DRAINER 语句用于修改集群中 Drainer 的状态信息。

> **注意：**
>
> Drainer 在正常运行时会自动上报状态到 PD，仅在 Drainer 处于异常情况导致实际状态与 PD 中保存的状态信息不一致时，使用该语句修改 PD 中存储的 Drainer 状态信息。

12.11.2.16.1 示例

```
SHOW DRAINER STATUS;
```

```
+----------|----------------|--------|--------------------|---------------------|
|  NodeID  |     Address    | State  |   Max_Commit_Ts    |    Update_Time      |
+----------|----------------|--------|--------------------|---------------------|
| drainer1 | 127.0.0.3:8249 | Online | 408553768673342532 | 2019-04-30 00:00:03 |
+----------|----------------|--------|--------------------|---------------------|
| drainer2 | 127.0.0.4:8249 | Online | 408553768673345531 | 2019-05-01 00:00:04 |
+----------|----------------|--------|--------------------|---------------------|
2 rows in set (0.00 sec)
```

可以看出 drainer1 已经超过一天没有更新状态，该 Drainer 处于异常状态，但是 State 仍然为 Online，使用 CHANGE DRAINER 将该 Drainer 状态修改为 paused：

```
CHANGE DRAINER TO NODE_STATE ='paused' FOR NODE_ID 'drainer1';
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
SHOW DRAINER STATUS;
```

```
+----------|----------------|--------|--------------------|---------------------|
|  NodeID  |     Address    | State  |   Max_Commit_Ts    |    Update_Time      |
+----------|----------------|--------|--------------------|---------------------|
| drainer1 | 127.0.0.3:8249 | Paused | 408553768673342532 | 2019-04-30 00:00:03 |
```

```
+----------|----------------|--------|--------------------|---------------------|
| drainer2 | 127.0.0.4:8249 | Online | 408553768673345531 | 2019-05-01 00:00:04 |
+----------|----------------|--------|--------------------|---------------------|
2 rows in set (0.00 sec)
```

12.11.2.16.2 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

12.11.2.16.3 另请参阅

- SHOW PUMP STATUS
- SHOW DRAINER STATUS
- CHANGE PUMP STATUS

12.11.2.17 CHANGE PUMP

CHANGE PUMP 语句用于修改集群中 Pump 的状态信息。

> **注意：**
>
> Pump 在正常运行时会自动上报状态到 PD，仅在 Pump 处于异常情况导致实际状态与 PD 中保存的状态信息不一致时，使用该语句修改 PD 中存储的 Pump 状态信息。

12.11.2.17.1 示例

```
SHOW PUMP STATUS;
```

```
+--------|----------------|--------|--------------------|---------------------|
| NodeID |     Address    | State  |   Max_Commit_Ts    |    Update_Time      |
+--------|----------------|--------|--------------------|---------------------|
| pump1  | 127.0.0.1:8250 | Online | 408553768673342237 | 2019-04-30 00:00:01 |
+--------|----------------|--------|--------------------|---------------------|
| pump2  | 127.0.0.2:8250 | Online | 408553768673342335 | 2019-05-01 00:00:02 |
+--------|----------------|--------|--------------------|---------------------|
2 rows in set (0.00 sec)
```

可以看出 pump1 已经超过一天没有更新状态，该 Pump 处于异常状态，但是 State 仍然为 Online，使用 CHANGE PUMP 将该 Pump 状态修改为 paused：

```
CHANGE PUMP TO NODE_STATE ='paused' FOR NODE_ID 'pump1';
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
SHOW PUMP STATUS;
```

```
+--------|----------------|--------|--------------------|---------------------|
| NodeID |     Address    | State  |   Max_Commit_Ts    |    Update_Time      |
+--------|----------------|--------|--------------------|---------------------|
| pump1  | 127.0.0.1:8250 | Paused | 408553768673342237 | 2019-04-30 00:00:01 |
+--------|----------------|--------|--------------------|---------------------|
| pump2  | 127.0.0.2:8250 | Online | 408553768673342335 | 2019-05-01 00:00:02 |
+--------|----------------|--------|--------------------|---------------------|
2 rows in set (0.00 sec)
```

12.11.2.17.2 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

12.11.2.17.3 另请参阅

- SHOW PUMP STATUS
- SHOW DRAINER STATUS
- CHANGE DRAINER STATUS

12.11.2.18 COMMIT

COMMIT 语句用于在 TiDB 服务器内部提交事务。

在不使用 BEGIN 或 START TRANSACTION 语句的情况下，TiDB 中每一个查询语句本身也会默认作为事务处理，自动提交，确保了与 MySQL 的兼容。

12.11.2.18.1 语法图

```
CommitStmt ::=
    'COMMIT' CompletionTypeWithinTransaction?

CompletionTypeWithinTransaction ::=
    'AND' ( 'CHAIN' ( 'NO' 'RELEASE' )? | 'NO' 'CHAIN' ( 'NO'? 'RELEASE' )? )
|   'NO'? 'RELEASE'
```

12.11.2.18.2 示例

```
CREATE TABLE t1 (a int NOT NULL PRIMARY KEY);
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
START TRANSACTION;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
INSERT INTO t1 VALUES (1);
```

```
Query OK, 1 row affected (0.00 sec)
```

```
COMMIT;
```

```
Query OK, 0 rows affected (0.01 sec)
```

12.11.2.18.3 MySQL 兼容性

- TiDB 3.0.8 及更新版本默认使用悲观事务模型。在乐观事务模型下，需要考虑到修改的行已被另一个事务修改，导致 COMMIT 语句可能执行失败的情况。
- 启用乐观事务模型后，UNIQUE 和 PRIMARY KEY 约束检查将延迟直至语句提交。当 COMMIT 语句失败时，这可能导致其他问题。可通过设置 tidb_constraint_check_in_place=TRUE 来改变该行为。
- TiDB 解析但忽略 ROLLBACK AND [NO] RELEASE 语法。在 MySQL 中，使用该语法可在提交事务后立即断开客户端会话。在 TiDB 中，建议使用客户端程序的 mysql_close() 来实现该功能。
- TiDB 解析但忽略 ROLLBACK AND [NO] CHAIN 语法。在 MySQL 中，使用该语法可在提交当前事务时立即以相同的隔离级别开启新事务。在 TiDB 中，推荐直接开启新事务。

12.11.2.18.4 另请参阅

- START TRANSACTION
- ROLLBACK
- BEGIN
- 事务的惰性检查

12.11.2.19 CREATE [GLOBAL|SESSION] BINDING

CREATE [GLOBAL|SESSION] BINDING 语句用于在 TiDB 中创建新的执行计划绑定。绑定可用于将优化器 Hint 插入语句中，而无需更改底层查询。

BINDING 语句可以在 GLOBAL 或者 SESSION 作用域内创建执行计划绑定。在不指定作用域时，默认的作用域为 SESSION。

被绑定的 SQL 语句会被参数化后存储到系统表中。在处理 SQL 查询时，只要参数化后的 SQL 语句和系统表中某个被绑定的 SQL 语句一致，并且系统变量 tidb_use_plan_baselines 的值为 ON（其默认值为 ON），即可使用相应的优化器 Hint。如果存在多个可匹配的执行计划，优化器会从中选择代价最小的一个进行绑定。

12.11.2.19.1 语法图

```
CreateBindingStmt ::=
    'CREATE' GlobalScope 'BINDING' 'FOR' BindableStmt 'USING' BindableStmt

GlobalScope ::=
    ( 'GLOBAL' | 'SESSION' )?

BindableStmt ::=
    ( SelectStmt | UpdateStmt | InsertIntoStmt | ReplaceIntoStmt | DeleteStmt )
```

12.11.2.19.2 示例

```
CREATE TABLE t1 (
    ->  id INT NOT NULL PRIMARY KEY auto_increment,
    ->  b INT NOT NULL,
    ->  pad VARBINARY(255),
    ->  INDEX(b)
    -> );
Query OK, 0 rows affected (0.07 sec)

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM dual;
Query OK, 1 row affected (0.01 sec)
Records: 1  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 8 rows affected (0.00 sec)
Records: 8  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 1000 rows affected (0.04 sec)
Records: 1000  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 100000 rows affected (1.74 sec)
```

```
Records: 100000  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 100000 rows affected (2.15 sec)
Records: 100000  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 100000 rows affected (2.64 sec)
Records: 100000  Duplicates: 0  Warnings: 0

SELECT SLEEP(1);
+----------+
| SLEEP(1) |
+----------+
|        0 |
+----------+
1 row in set (1.00 sec)

ANALYZE TABLE t1;
Query OK, 0 rows affected (1.33 sec)

EXPLAIN ANALYZE SELECT * FROM t1 WHERE b = 123;
+--
    ↪ ----------------------------+---------+---------+-----------+----------------------+-----------------
    ↪
| id                          | estRows | actRows | task      | access object        |
    ↪ execution info                                                          | operator info
    ↪                       | memory         | disk |
+--
    ↪ ----------------------------+---------+---------+-----------+----------------------+-----------------
    ↪
| IndexLookUp_10              | 583.00  | 297     | root      |                      | time
    ↪ :10.545072ms, loops:2, rpc num: 1, rpc time:398.359µs, proc keys:297  |
    ↪                                     | 109.1484375 KB | N/A  |
| ├─IndexRangeScan_8(Build)     | 583.00  | 297     | cop[tikv] | table:t1, index:b(b) | time:0s
    ↪ , loops:4                                                               | range:[123,123],
    ↪ keep order:false | N/A            | N/A  |
| └─TableRowIDScan_9(Probe)     | 583.00  | 297     | cop[tikv] | table:t1             | time:12
    ↪ ms, loops:4                                                             | keep order:false
    ↪                   | N/A            | N/A  |
+--
    ↪ ----------------------------+---------+---------+-----------+----------------------+-----------------
    ↪
```

```
3 rows in set (0.02 sec)

CREATE SESSION BINDING FOR
    ->  SELECT * FROM t1 WHERE b = 123
    -> USING
    ->  SELECT * FROM t1 IGNORE INDEX (b) WHERE b = 123;
Query OK, 0 rows affected (0.00 sec)

EXPLAIN ANALYZE  SELECT * FROM t1 WHERE b = 123;
+--
    ↪ ----------------------+----------+---------+----------+--------------+--------------------------------
    ↪
| id                    | estRows   | actRows | task      | access object | execution info
    ↪                                                                    | operator info      |
    ↪ memory        | disk |
+--
    ↪ ----------------------+----------+---------+----------+--------------+--------------------------------
    ↪
| TableReader_7           | 583.00    | 297     | root      |               | time:222.32506ms,
    ↪ loops:2, rpc num: 1, rpc time:222.078952ms, proc keys:301010 | data:Selection_6   |
    ↪ 88.6640625 KB | N/A  |
| └─Selection_6           | 583.00    | 297     | cop[tikv] |               | time:224ms, loops
    ↪ :298                                                          | eq(test.t1.b, 123) | N/A
    ↪           | N/A  |
|   └─TableFullScan_5     | 301010.00 | 301010  | cop[tikv] | table:t1      | time:220ms, loops
    ↪ :298                                                          | keep order:false   | N/A
    ↪           | N/A  |
+--
    ↪ ----------------------+----------+---------+----------+--------------+--------------------------------
    ↪
3 rows in set (0.22 sec)

SHOW SESSION BINDINGS\G
*************************** 1. row ***************************
Original_sql: select * from t1 where b = ?
    Bind_sql: SELECT * FROM t1 IGNORE INDEX (b) WHERE b = 123
  Default_db: test
      Status: using
 Create_time: 2020-05-22 14:38:03.456
 Update_time: 2020-05-22 14:38:03.456
     Charset: utf8mb4
   Collation: utf8mb4_0900_ai_ci
1 row in set (0.00 sec)

DROP SESSION BINDING FOR SELECT * FROM t1 WHERE b = 123;
```

```
Query OK, 0 rows affected (0.00 sec)

EXPLAIN ANALYZE  SELECT * FROM t1 WHERE b = 123;
+--
    ↪ ----------------------------+---------+---------+-----------+---------------------+-----------------
    ↪
| id                            | estRows | actRows | task      | access object       |
    ↪ execution info                                                                  | operator info
    ↪                       | memory         | disk |
+--
    ↪ ----------------------------+---------+---------+-----------+---------------------+-----------------
    ↪
| IndexLookUp_10                | 583.00  | 297     | root      |                     | time
    ↪ :5.31206ms, loops:2, rpc num: 1, rpc time:665.927µs, proc keys:297  |
    ↪                                     | 109.1484375 KB | N/A  |
| ├─IndexRangeScan_8(Build)     | 583.00  | 297     | cop[tikv] | table:t1, index:b(b) | time:0s
    ↪ , loops:4                                                             | range:[123,123], keep
    ↪ order:false | N/A             | N/A  |
| └─TableRowIDScan_9(Probe)     | 583.00  | 297     | cop[tikv] | table:t1             | time:0s
    ↪ , loops:4                                                             | keep order:false
    ↪                     | N/A             | N/A  |
+--
    ↪ ----------------------------+---------+---------+-----------+---------------------+-----------------
    ↪
3 rows in set (0.01 sec)
```

12.11.2.19.3 MySQL 兼容性

`CREATE [GLOBAL|SESSION] BINDING` 语句是 TiDB 对 MySQL 语法的扩展。

12.11.2.19.4 另请参阅

- DROP [GLOBAL|SESSION] BINDING
- SHOW [GLOBAL|SESSION] BINDINGS
- ANALYZE
- Optimizer Hints
- 执行计划管理 (SPM)

12.11.2.20 CREATE DATABASE

`CREATE DATABASE` 语句用于在 TiDB 上创建新数据库。按照 SQL 标准，“数据库”一词在 MySQL 术语中最接近“schema”。

12.11.2.20.1 语法图

```
CreateDatabaseStmt ::=
    'CREATE' 'DATABASE' IfNotExists DBName DatabaseOptionListOpt

IfNotExists ::=
    ( 'IF' 'NOT' 'EXISTS' )?

DBName ::=
    Identifier

DatabaseOptionListOpt ::=
    DatabaseOptionList?
```

12.11.2.20.2 语法说明

`CREATE DATABASE` 用于创建数据库，并可以指定数据库的默认属性（如数据库默认字符集、排序规则）。`CREATE SCHEMA` 跟 `CREATE DATABASE` 操作效果一样。

```
CREATE {DATABASE | SCHEMA} [IF NOT EXISTS] db_name
    [create_specification] ...

create_specification:
    [DEFAULT] CHARACTER SET [=] charset_name
  | [DEFAULT] COLLATE [=] collation_name
```

当创建已存在的数据库且不指定使用 `IF NOT EXISTS` 时会报错。

`create_specification` 选项用于指定数据库具体的 `CHARACTER SET` 和 `COLLATE`。目前 TiDB 只支持部分的字符集和排序规则，请参照字符集支持。

12.11.2.20.3 示例

```
CREATE DATABASE mynewdatabase;
```

```
Query OK, 0 rows affected (0.09 sec)
```

```
USE mynewdatabase;
```

```
Database changed
```

```
CREATE TABLE t1 (a int);
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
SHOW TABLES;
```

```
+-------------------------+
| Tables_in_mynewdatabase |
+-------------------------+
| t1                      |
+-------------------------+
1 row in set (0.00 sec)
```

12.11.2.20.4 MySQL 兼容性

`CREATE DATABASE` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.20.5 另请参阅

- USE
- ALTER DATABASE
- DROP DATABASE
- SHOW DATABASES

12.11.2.21 CREATE INDEX

`CREATE INDEX` 语句用于在已有表中添加新索引，功能等同于 `ALTER TABLE .. ADD INDEX`。包含该语句提供了 MySQL 兼容性。

12.11.2.21.1 语法图

```
CreateIndexStmt ::=
    'CREATE' IndexKeyTypeOpt 'INDEX' IfNotExists Identifier IndexTypeOpt 'ON' TableName '('
        ↪ IndexPartSpecificationList ')' IndexOptionList IndexLockAndAlgorithmOpt

IndexKeyTypeOpt ::=
    ( 'UNIQUE' | 'SPATIAL' | 'FULLTEXT' )?

IfNotExists ::=
    ( 'IF' 'NOT' 'EXISTS' )?

IndexTypeOpt ::=
    IndexType?

IndexPartSpecificationList ::=
    IndexPartSpecification ( ',' IndexPartSpecification )*

IndexOptionList ::=
    IndexOption*
```

```
IndexLockAndAlgorithmOpt ::=
    ( LockClause AlgorithmClause? | AlgorithmClause LockClause? )?

IndexType ::=
    ( 'USING' | 'TYPE' ) IndexTypeName

IndexPartSpecification ::=
    ( ColumnName OptFieldLen | '(' Expression ')' ) Order

IndexOption ::=
    'KEY_BLOCK_SIZE' '='? LengthNum
|   IndexType
|   'WITH' 'PARSER' Identifier
|   'COMMENT' stringLit
|   IndexInvisible

IndexTypeName ::=
    'BTREE'
|   'HASH'
|   'RTREE'

ColumnName ::=
    Identifier ( '.' Identifier ( '.' Identifier )? )?

OptFieldLen ::=
    FieldLen?

IndexNameList ::=
    ( Identifier | 'PRIMARY' )? ( ',' ( Identifier | 'PRIMARY' ) )*

KeyOrIndex ::=
    'Key' | 'Index'
```

12.11.2.21.2 示例

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, c1 INT NOT NULL);
```

```
Query OK, 0 rows affected (0.10 sec)
```

```
INSERT INTO t1 (c1) VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 5 rows affected (0.02 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
EXPLAIN SELECT * FROM t1 WHERE c1 = 3;
```

```
+-------------------------+----------+-----------+--------------+--------------------------------+
    ↪
| id                      | estRows  | task      | access object | operator info
    ↪                   |
+-------------------------+----------+-----------+--------------+--------------------------------+
    ↪
| TableReader_7           | 10.00    | root      |               | data:Selection_6
    ↪               |
| └─Selection_6           | 10.00    | cop[tikv] |               | eq(test.t1.c1, 3)
    ↪              |
|   └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t1      | keep order:false, stats:
    ↪ pseudo |
+-------------------------+----------+-----------+--------------+--------------------------------+
    ↪
3 rows in set (0.00 sec)
```

```
CREATE INDEX c1 ON t1 (c1);
```

```
Query OK, 0 rows affected (0.30 sec)
```

```
EXPLAIN SELECT * FROM t1 WHERE c1 = 3;
```

```
+------------------------+---------+-----------+-----------------------+-----------------------------------
    ↪
| id                     | estRows | task      | access object         | operator info
    ↪                               |
+------------------------+---------+-----------+-----------------------+-----------------------------------
    ↪
| IndexReader_6          | 10.00   | root      |                       | index:IndexRangeScan_5
    ↪                      |
| └─IndexRangeScan_5     | 10.00   | cop[tikv] | table:t1, index:c1(c1) | range:[3,3], keep
    ↪ order:false, stats:pseudo |
+------------------------+---------+-----------+-----------------------+-----------------------------------
    ↪
2 rows in set (0.00 sec)
```

```
ALTER TABLE t1 DROP INDEX c1;
```

```
Query OK, 0 rows affected (0.30 sec)
```

```
CREATE UNIQUE INDEX c1 ON t1 (c1);
```

```
Query OK, 0 rows affected (0.31 sec)
```

12.11.2.21.3 表达式索引

> **注意：**
>
> 该功能目前为实验特性，不建议在生产环境中使用。

如果需要使用这一特性，在TiDB 配置文件中进行以下设置：

```
allow-expression-index = true
```

TiDB 不仅能将索引建立在表中的一个或多个列上，还可以将索引建立在一个表达式上。当查询涉及表达式时，表达式索引能够加速这些查询。

考虑以下查询：

```
SELECT * FROM t WHERE lower(name) = "pingcap";
```

如果建立了如下的表达式索引，就可以使用索引加速以上查询：

```
CREATE INDEX idx ON t ((lower(name)));
```

维护表达式索引的代价比一般的索引更高，因为在插入或者更新每一行时都需要计算出表达式的值。因为表达式的值已经存储在索引中，所以当优化器选择表达式索引时，表达式的值就不需要再计算。因此，当查询速度比插入速度和更新速度更重要时，可以考虑建立表达式索引。

表达式索引的语法和限制与 MySQL 相同，是通过将索引建立在隐藏的虚拟生成列 (generated virtual column) 上来实现的。因此所支持的表达式继承了虚拟生成列的所有限制。目前，建立了索引的表达式只有在 `FIELD` 子句、`WHERE` 子句和 `ORDER BY` 子句中时，优化器才能使用表达式索引。后续将支持 `GROUP BY` 子句。

12.11.2.21.4 相关系统变量

和 `CREATE INDEX` 语句相关的系统变量有 `tidb_ddl_reorg_worker_cnt` 、`tidb_ddl_reorg_batch_size` 和 `tidb_ddl_reorg_priority`，具体可以参考系统变量。

12.11.2.21.5 MySQL 兼容性

- 不支持 `FULLTEXT`，`HASH` 和 `SPATIAL` 索引。
- 不支持降序索引（类似于 MySQL 5.7）。
- 默认无法向表中添加 `PRIMARY KEY`，在开启 `alter-primary-key` 配置项后可支持此功能，详情参考：alter-primary-key。

12.11.2.21.6 另请参阅

- 索引的选择
- 错误索引的解决方案

- ADD INDEX
- DROP INDEX
- RENAME INDEX
- ADD COLUMN
- CREATE TABLE
- EXPLAIN

12.11.2.22 CREATE ROLE

`CREATE ROLE` 语句是基于角色的访问控制 (RBAC) 操作的一部分，用于创建新角色并将新角色分配给用户。

12.11.2.22.1 语法图

```
CreateRoleStmt ::=
    'CREATE' 'ROLE' IfNotExists RoleSpec (',' RoleSpec)*

IfNotExists ::=
    ('IF' 'NOT' 'EXISTS')?

RoleSpec ::=
    Rolename
```

12.11.2.22.2 示例

创建新角色 `analyticsteam` 和新用户 `jennifer`：

```
$ mysql -uroot

CREATE ROLE analyticsteam;
Query OK, 0 rows affected (0.02 sec)

GRANT SELECT ON test.* TO analyticsteam;
Query OK, 0 rows affected (0.02 sec)

CREATE USER jennifer;
Query OK, 0 rows affected (0.01 sec)

GRANT analyticsteam TO jennifer;
Query OK, 0 rows affected (0.01 sec)
```

需要注意的是，默认情况下，用户 `jennifer` 需要执行 `SET ROLE analyticsteam` 语句才能使用与角色相关联的权限：

```
$ mysql -ujennifer

SHOW GRANTS;
```

```
+-------------------------------------------+
| Grants for User                           |
+-------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'      |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+-------------------------------------------+
2 rows in set (0.00 sec)

SHOW TABLES in test;
ERROR 1044 (42000): Access denied for user 'jennifer'@'%' to database 'test'
SET ROLE analyticsteam;
Query OK, 0 rows affected (0.00 sec)

SHOW GRANTS;
+-------------------------------------------+
| Grants for User                           |
+-------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'      |
| GRANT Select ON test.* TO 'jennifer'@'%'  |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+-------------------------------------------+
3 rows in set (0.00 sec)

SHOW TABLES IN test;
+----------------+
| Tables_in_test |
+----------------+
| t1             |
+----------------+
1 row in set (0.00 sec)
```

执行 `SET DEFAULT ROLE` 语句将用户 `jennifer` 与某一角色相关联，这样该用户无需执行 `SET ROLE` 语句就能拥有与角色相关联的权限。

```
$ mysql -uroot

SET DEFAULT ROLE analyticsteam TO jennifer;
Query OK, 0 rows affected (0.02 sec)
```

```
$ mysql -ujennifer

SHOW GRANTS;
+-------------------------------------------+
| Grants for User                           |
+-------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'      |
```

```
| GRANT Select ON test.* TO 'jennifer'@'%'     |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+----------------------------------------------+
3 rows in set (0.00 sec)

SHOW TABLES IN test;
+----------------+
| Tables_in_test |
+----------------+
| t1             |
+----------------+
1 row in set (0.00 sec)
```

12.11.2.22.3 MySQL 兼容性

CREATE ROLE 语句与 MySQL 8.0 的“角色”功能完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.22.4 另请参阅

- DROP ROLE
- GRANT <role>
- REVOKE <role>
- SET ROLE
- SET DEFAULT ROLE
- 基于角色的访问控制

12.11.2.23 CREATE SEQUENCE

CREATE SEQUENCE 语句用于在 TiDB 中创建序列对象。序列是一种与表、视图对象平级的数据库对象，用于生成自定义的序列化 ID。

12.11.2.23.1 语法图

```
CreateSequenceStmt ::=
    'CREATE' 'SEQUENCE' IfNotExists TableName CreateSequenceOptionListOpt
        ↪ CreateTableOptionListOpt

IfNotExists ::=
    ('IF' 'NOT' 'EXISTS')?

TableName ::=
    Identifier ('.' Identifier)?

CreateSequenceOptionListOpt ::=
    SequenceOption*
```

```
SequenceOptionList ::=
    SequenceOption

SequenceOption ::=
    ( 'INCREMENT' ( '='? | 'BY' ) | 'START' ( '='? | 'WITH' ) | ( 'MINVALUE' | 'MAXVALUE' | '
        ↪ CACHE' ) '='? ) SignedNum
|   'NOMINVALUE'
|   'NO' ( 'MINVALUE' | 'MAXVALUE' | 'CACHE' | 'CYCLE' )
|   'NOMAXVALUE'
|   'NOCACHE'
|   'CYCLE'
|   'NOCYCLE'
```

12.11.2.23.2 语法说明

```
CREATE [TEMPORARY] SEQUENCE [IF NOT EXISTS] sequence_name
    [ INCREMENT [ BY | = ] increment ]
    [ MINVALUE [=] minvalue | NO MINVALUE | NOMINVALUE ]
    [ MAXVALUE [=] maxvalue | NO MAXVALUE | NOMAXVALUE ]
    [ START [ WITH | = ] start ]
    [ CACHE [=] cache | NOCACHE | NO CACHE]
    [ CYCLE | NOCYCLE | NO CYCLE]
    [table_options]
```

12.11.2.23.3 参数说明

参数	默认值	描述
TEMPORARY ↪	false ↪	TiDB 暂时不支持 TEMPORARY ↪ 选项，仅在语法上做兼容。

参数	默认值	描述
INCREMENT ↪	1	指定序列的步长。其正负值可以控制序列的增长方向。
MINVALUE ↪	1 或 -9223372036854775807 ↪	指定序列的最小值。当 INCREMENT ↪ > 0 时，默认值为 1；当 INCREMENT ↪ < 0 时，默认值为 -9223372036854775807 ↪ 。

参数	默认值	描述
MAXVALUE	9223372036854775806 或 -1	指定序列的最大值。当 INCREMENT > 0 时，默认值为 9223372036854775806；当 INCREMENT < 0 时，默认值为 -1。
START	MINVALUE 或 MAXVALUE	指定序列的初始值。当 INCREMENT > 0 时，默认值为 MINVALUE；当 INCREMENT < 0 时，默认值为 MAXVALUE。

参数	默认值	描述
CACHE ↪	1000	指定每个 TiDB 本地缓存序列的大小。

参数	默认值	描述
CYCLE ↪	NO ↪ CYCLE ↪	指定序列用完之后是否要循环使用。在 CYCLE ↪ 的情况下，当 INCREMENT ↪ > 0 时，序列用完后的后续起始值为 MINVALUE ↪ ；当 INCREMENT ↪ < 0 时，序列用完后的后续起始值为 MAXVALUE ↪ 。

12.11.2.23.4 SEQUENCE 函数

主要通过表达式函数来操纵序列的使用。

- NEXTVAL 或 NEXT VALUE FOR

本质上都是 nextval() 函数，获取序列对象的下一个有效值，其参数为序列的 identifier。

- LASTVAL

  lastval() 函数，用于获取本会话上一个使用过的值。如果没有值，则为 NULL，其参数为序列的 identifier。

- SETVAL

  setval() 函数，用于设置序列的增长。其第一参数为序列的 identifier，第二个参数为 num。

> **注意：**
>
> 在 TiDB 序列的实现中，SETVAL 函数并不能改变序列增长的初始步调或循环步调。在 SETVAL 之后只会返回符合步调规律的下一个有效的序列值。

12.11.2.23.5 示例

- 创建一个默认参数的序列对象。

```
CREATE SEQUENCE seq;
```

```
Query OK, 0 rows affected (0.06 sec)
```

- 使用 nextval() 函数获取序列对象的下一个值。

```
SELECT nextval(seq);
```

```
+--------------+
| nextval(seq) |
+--------------+
|            1 |
+--------------+
1 row in set (0.02 sec)
```

- 使用 lastval() 函数获取本会话上一次调用序列对象所产生的值。

```
SELECT lastval(seq);
```

```
+--------------+
| lastval(seq) |
+--------------+
|            1 |
+--------------+
1 row in set (0.02 sec)
```

- 使用 `setval()` 函数设置序列对象当前值的位置。

```
SELECT setval(seq, 10);
```

```
+-----------------+
| setval(seq, 10) |
+-----------------+
|              10 |
+-----------------+
1 row in set (0.01 sec)
```

- 也可使用 `next value for` 语法获取序列的下一个值。

```
SELECT next value for seq;
```

```
+--------------------+
| next value for seq |
+--------------------+
|                 11 |
+--------------------+
1 row in set (0.00 sec)
```

- 创建一个默认自定义参数的序列对象。

```
CREATE SEQUENCE seq2 start 3 increment 2 minvalue 1 maxvalue 10 cache 3;
```

```
Query OK, 0 rows affected (0.01 sec)
```

- 当本会话还未使用过序列对象时，`lastval()` 函数返回 NULL 值。

```
SELECT lastval(seq2);
```

```
+---------------+
| lastval(seq2) |
+---------------+
|          NULL |
+---------------+
1 row in set (0.01 sec)
```

- 序列对象 `nextval()` 的第一个有效值为 `start` 值。

```
SELECT nextval(seq2);
```

```
+---------------+
| nextval(seq2) |
+---------------+
|             3 |
```

```
+---------------+
1 row in set (0.00 sec)
```

- 使用 `setval()` 虽然可以改变序列对象当前值的位置，但是无法改变下一个值的等差规律。

```
SELECT setval(seq2, 6);
```

```
+-----------------+
| setval(seq2, 6) |
+-----------------+
|               6 |
+-----------------+
1 row in set (0.00 sec)
```

- 使用 `nextval()` 下一个值获取时，会遵循序列定义的等差规律。

```
SELECT next value for seq2;
```

```
+---------------------+
| next value for seq2 |
+---------------------+
|                   7 |
+---------------------+
1 row in set (0.00 sec)
```

- 可以将序列的下一个值作为列的默认值来使用。

```
CREATE TABLE t(a int default next value for seq2);
```

```
Query OK, 0 rows affected (0.02 sec)
```

- 下列示例中，因为没有指定值，会直接获取 `seq2` 的默认值来使用。

```
INSERT into t values();
```

```
Query OK, 1 row affected (0.00 sec)
```

```
SELECT * from t;
```

```
+------+
| a    |
+------+
|    9 |
+------+
1 row in set (0.00 sec)
```

- 下列示例中，因为没有指定值，会直接获取 seq2 的默认值来使用。由于 seq2 的下一个值超过了上述示例 (CREATE SEQUENCE seq2 start 3 increment 2 minvalue 1 maxvalue 10 cache 3;) 的定义范围，所以会显示报错。

```
INSERT into t values();
```

```
ERROR 4135 (HY000): Sequence 'test.seq2' has run out
```

12.11.2.23.6 MySQL 兼容性

该语句是 TiDB 的扩展，序列的实现借鉴自 MariaDB。

除了 SETVAL 函数外，其他函数的 “步调 (progressions)” 与 MariaDB 一致。这里的步调是指，序列中的数在定义之后会产生一定的等差关系。SETVAL 虽然可以将序列的当前值进行移动设置，但是后续出现的值仍会遵循原有的等差关系。

示例如下：

```
1, 3, 5, ...            // 序列遵循起始为 1、步长为 2 的等差关系。
select setval(seq, 6)   // 设置序列的当前值为 6。
7, 9, 11, ...           // 后续产生值仍会遵循这个等差关系。
```

在 CYCLE 模式下，序列的起始值第一轮为 start，后续轮次将会是 MinValue (increment > 0) 或 MaxValue (increment < 0)。

12.11.2.23.7 另请参阅

- DROP SEQUENCE
- SHOW CREATE SEQUENCE

12.11.2.24 CREATE TABLE LIKE

CREATE TABLE LIKE 语句用于复制已有表的定义，但不复制任何数据。

12.11.2.24.1 语法图

```
CreateTableLikeStmt ::=
    'CREATE' OptTemporary 'TABLE' IfNotExists TableName LikeTableWithOrWithoutParen

LikeTableWithOrWithoutParen ::=
    'LIKE' TableName
|   '(' 'LIKE' TableName ')'
```

12.11.2.24.2 示例

```
CREATE TABLE t1 (a INT NOT NULL);
```

```
Query OK, 0 rows affected (0.13 sec)
```

```
INSERT INTO t1 VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 5 rows affected (0.02 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
SELECT * FROM t1;
```

```
+---+
| a |
+---+
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
+---+
5 rows in set (0.00 sec)
```

```
CREATE TABLE t2 LIKE t1;
```

```
Query OK, 0 rows affected (0.10 sec)
```

```
SELECT * FROM t2;
```

```
Empty set (0.00 sec)
```

12.11.2.24.3 Region 的预切分

如果被复制的表定义了 PRE_SPLIT_REGIONS 属性，则通过 CREATE TABLE LIKE 语句复制的表，会继承该属性并在建表时预切分 Region。关于 PRE_SPLIT_REGIONS 属性的说明，参见CREATE TABLE 语句。

12.11.2.24.4 MySQL 兼容性

CREATE TABLE LIKE 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.24.5 另请参阅

- CREATE TABLE
- SHOW CREATE TABLE

12.11.2.25 CREATE TABLE

`CREATE TABLE` 语句用于在当前所选数据库中创建新表，与 MySQL 中 `CREATE TABLE` 语句的行为类似。另可参阅单独的 `CREATE TABLE LIKE` 文档。

12.11.2.25.1 语法图

```
CreateTableStmt ::=
    'CREATE' OptTemporary 'TABLE' IfNotExists TableName ( TableElementListOpt
        ↪ CreateTableOptionListOpt PartitionOpt DuplicateOpt AsOpt CreateTableSelectOpt |
        ↪ LikeTableWithOrWithoutParen )

IfNotExists ::=
    ('IF' 'NOT' 'EXISTS')?

TableName ::=
    Identifier ('.' Identifier)?

TableElementListOpt ::=
    ( '(' TableElementList ')' )?

TableElementList ::=
    TableElement ( ',' TableElement )*

TableElement ::=
    ColumnDef
|   Constraint

ColumnDef ::=
    ColumnName ( Type | 'SERIAL' ) ColumnOptionListOpt

ColumnOptionListOpt ::=
    ColumnOption*

ColumnOptionList ::=
    ColumnOption*

ColumnOption ::=
    'NOT'? 'NULL'
|   'AUTO_INCREMENT'
|   PrimaryOpt 'KEY'
|   'UNIQUE' 'KEY'?
|   'DEFAULT' DefaultValueExpr
|   'SERIAL' 'DEFAULT' 'VALUE'
|   'ON' 'UPDATE' NowSymOptionFraction
|   'COMMENT' stringLit
```

```
|   ConstraintKeywordOpt 'CHECK' '(' Expression ')' EnforcedOrNotOrNotNullOpt
|   GeneratedAlways 'AS' '(' Expression ')' VirtualOrStored
|   ReferDef
|   'COLLATE' CollationName
|   'COLUMN_FORMAT' ColumnFormat
|   'STORAGE' StorageMedia
|   'AUTO_RANDOM' OptFieldLen

CreateTableOptionListOpt ::=
    TableOptionList?

PartitionOpt ::=
    ( 'PARTITION' 'BY' PartitionMethod PartitionNumOpt SubPartitionOpt PartitionDefinitionListOpt
        ↪  )?

DuplicateOpt ::=
    ( 'IGNORE' | 'REPLACE' )?

TableOptionList ::=
    TableOption ( ','? TableOption )*

TableOption ::=
    PartDefOption
|   DefaultKwdOpt ( CharsetKw EqOpt CharsetName | 'COLLATE' EqOpt CollationName )
|   ( 'AUTO_INCREMENT' | 'AUTO_ID_CACHE' | 'AUTO_RANDOM_BASE' | 'AVG_ROW_LENGTH' | 'CHECKSUM' | '
    ↪ TABLE_CHECKSUM' | 'KEY_BLOCK_SIZE' | 'DELAY_KEY_WRITE' | 'SHARD_ROW_ID_BITS' | '
    ↪ PRE_SPLIT_REGIONS' ) EqOpt LengthNum
|   ( 'CONNECTION' | 'PASSWORD' | 'COMPRESSION' ) EqOpt stringLit
|   RowFormat
|   ( 'STATS_PERSISTENT' | 'PACK_KEYS' ) EqOpt StatsPersistentVal
|   ( 'STATS_AUTO_RECALC' | 'STATS_SAMPLE_PAGES' ) EqOpt ( LengthNum | 'DEFAULT' )
|   'STORAGE' ( 'MEMORY' | 'DISK' )
|   'SECONDARY_ENGINE' EqOpt ( 'NULL' | StringName )
|   'UNION' EqOpt '(' TableNameListOpt ')'
|   'ENCRYPTION' EqOpt EncryptionOpt
```

TiDB 支持以下 table_option。TiDB 会解析并忽略其他 table_option 参数，例如 AVG_ROW_LENGTH、CHECKSUM、COMPRESSION、CONNECTION、DELAY_KEY_WRITE、ENGINE、KEY_BLOCK_SIZE、MAX_ROWS、MIN_ROWS、ROW_FORMAT 和 STATS_PERSISTENT。

参数	含义	举例
AUTO_INCREMENT	自增字段初始值	AUTO_INCREMENT = 5
SHARD_ROW_ID_BITS	用来设置隐式 _tidb_rowid 的分片数量的 bit 位数	SHARD_ROW_ID_BITS = 4

参数	含义	举例
PRE_SPLIT_REGIONS	用来在建表时预先均匀切分 2^(PRE_SPLIT_REGIONS) 个 Region	PRE_SPLIT_REGIONS = 4
AUTO_ID_CACHE	用来指定 Auto ID 在 TiDB 实例中 Cache 的大小，默认情况下 TiDB 会根据 Auto ID 分配速度自动调整	AUTO_ID_CACHE = 200
AUTO_RANDOM_BASE	用来指定 AutoRandom 自增部分的初始值，该参数可以被认为属于内部接口的一部分，对于用户而言请忽略	AUTO_RANDOM_BASE = 0
CHARACTER SET	指定该表所使用的字符集	CHARACTER SET = ‘utf8mb4’
COLLATE	指定该表所使用的字符集排序规则	COLLATE = ‘utf8mb4_bin’
COMMENT	注释信息	COMMENT = ‘comment info’

注意：

在 TiDB 配置文件中，split-table 默认开启。当该配置项开启时，建表操作会为每个表建立单独的 Region，详情参见TiDB 配置文件描述。

12.11.2.25.2 示例

创建一张简单表并插入一行数据：

```
CREATE TABLE t1 (a int);
DESC t1;
SHOW CREATE TABLE t1\G
INSERT INTO t1 (a) VALUES (1);
SELECT * FROM t1;
```

```
mysql> drop table if exists t1;
Query OK, 0 rows affected (0.23 sec)

mysql> CREATE TABLE t1 (a int);
Query OK, 0 rows affected (0.09 sec)

mysql> DESC t1;
+-------+---------+------+------+---------+-------+
| Field | Type    | Null | Key  | Default | Extra |
+-------+---------+------+------+---------+-------+
| a     | int(11) | YES  |      | NULL    |       |
+-------+---------+------+------+---------+-------+
1 row in set (0.00 sec)

mysql> SHOW CREATE TABLE t1\G
```

```
*************************** 1. row ***************************
       Table: t1
Create Table: CREATE TABLE `t1` (
  `a` int(11) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin
1 row in set (0.00 sec)

mysql> INSERT INTO t1 (a) VALUES (1);
Query OK, 1 row affected (0.03 sec)

mysql> SELECT * FROM t1;
+------+
| a    |
+------+
|    1 |
+------+
1 row in set (0.00 sec)
```

删除一张表。如果该表不存在，就建一张表：

```
DROP TABLE IF EXISTS t1;
CREATE TABLE IF NOT EXISTS t1 (
 id BIGINT NOT NULL PRIMARY KEY auto_increment,
 b VARCHAR(200) NOT NULL
);
DESC t1;
```

```
mysql> DROP TABLE IF EXISTS t1;
Query OK, 0 rows affected (0.22 sec)
mysql> CREATE TABLE IF NOT EXISTS t1 (
    ->  id BIGINT NOT NULL PRIMARY KEY auto_increment,
    ->  b VARCHAR(200) NOT NULL
    -> );
Query OK, 0 rows affected (0.08 sec)
mysql> DESC t1;
+-------+--------------+------+------+---------+----------------+
| Field | Type         | Null | Key  | Default | Extra          |
+-------+--------------+------+------+---------+----------------+
| id    | bigint(20)   | NO   | PRI  | NULL    | auto_increment |
| b     | varchar(200) | NO   |      | NULL    |                |
+-------+--------------+------+------+---------+----------------+
2 rows in set (0.00 sec)
```

12.11.2.25.3 MySQL 兼容性

- TiDB 不支持临时表，对于 CREATE TEMPORARY TABLE 语法，会忽略 TEMPORARY 关键字。
- 支持除空间类型以外的所有数据类型。
- 不支持 FULLTEXT，HASH 和 SPATIAL 索引。
- 为了与 MySQL 兼容，index_col_name 属性支持 length 选项，最大长度默认限制为 3072 字节。此长度限制可以通过配置项 max-index-length 更改，具体请参阅TiDB 配置文件描述。
- 为了与 MySQL 兼容，TiDB 会解析但忽略 index_col_name 属性的 [ASC | DESC] 索引排序选项。
- COMMENT 属性最多支持 1024 个字符，不支持 WITH PARSER 选项。
- TiDB 在单个表中最多支持 512 列。InnoDB 中相应的数量限制为 1017，MySQL 中的硬限制为 4096。详情参阅TiDB 使用限制。
- 当前仅支持 Range、 Hash 和 Range Columns（单列）类型的分区表，详情参阅分区表。
- TiDB 会解析并忽略 CHECK 约束，与 MySQL 5.7 相兼容。详情参阅CHECK 约束。
- TiDB 会解析并存储外键约束，但不会在 DML 语句中强制对外键进行约束检查。详情外键约束。

12.11.2.25.4 另请参阅

- 数据类型
- DROP TABLE
- CREATE TABLE LIKE
- SHOW CREATE TABLE

12.11.2.26 CREATE USER

CREATE USER 语句用于创建带有指定密码的新用户。和 MySQL 一样，在 TiDB 权限系统中，用户是用户名和用户名所连接主机的组合。因此，可创建一个用户 'newuser2'@'192.168.1.1'，使其只能通过 IP 地址 192.168.1.1 进行连接。相同的用户名从不同主机登录时可能会拥有不同的权限。

12.11.2.26.1 语法图

```
CreateUserStmt ::=
    'CREATE' 'USER' IfNotExists UserSpecList RequireClauseOpt ConnectionOptions
        ↪ PasswordOrLockOptions

IfNotExists ::=
    ('IF' 'NOT' 'EXISTS')?

UserSpecList ::=
    UserSpec ( ',' UserSpec )*

UserSpec ::=
    Username AuthOption

AuthOption ::=
    ( 'IDENTIFIED' ( 'BY' ( AuthString | 'PASSWORD' HashString ) | 'WITH' StringName ( 'BY'
        ↪ AuthString | 'AS' HashString )? ) )?
```

```
StringName ::=
    stringLit
|   Identifier
```

12.11.2.26.2 示例

创建一个密码为 newuserpassword 的用户。

```
CREATE USER 'newuser' IDENTIFIED BY 'newuserpassword';
```

```
Query OK, 1 row affected (0.04 sec)
```

创建一个只能在 192.168.1.1 登录的用户，密码为 newuserpassword。

```
CREATE USER 'newuser2'@'192.168.1.1' IDENTIFIED BY 'newuserpassword';
```

```
Query OK, 1 row affected (0.02 sec)
```

创建一个要求在登录时使用 TLS 连接的用户。

```
CREATE USER 'newuser3'@'%' REQUIRE SSL IDENTIFIED BY 'newuserpassword';
```

```
Query OK, 1 row affected (0.02 sec)
```

创建一个要求在登录时提供指定客户端证书的用户。

```
CREATE USER 'newuser4'@'%' REQUIRE ISSUER '/C=US/ST=California/L=San Francisco/O=PingCAP'
    ↪ IDENTIFIED BY 'newuserpassword';
```

```
Query OK, 1 row affected (0.02 sec)
```

12.11.2.26.3 MySQL 兼容性

- TiDB 不支持 WITH MAX_QUERIES_PER_HOUR、WITH MAX_UPDATES_PER_HOUR、WITH MAX_USER_CONNECTIONS 等 CREATE 选项。
- TiDB 不支持 DEFAULT ROLE 选项。
- TiDB 不支持 PASSWORD EXPIRE、PASSWORD HISTORY 等有关密码限制的 CREATE 选项。
- TiDB 不支持 ACCOUNT LOCK 和 ACCOUNT UNLOCK 选项。
- 对于 TiDB 尚不支持的 CREATE 选项。这些选项可被解析，但会被忽略。

12.11.2.26.4 另请参阅

- Security Compatibility with MySQL
- DROP USER
- SHOW CREATE USER
- ALTER USER
- Privilege Management

12.11.2.27 CREATE VIEW

使用 CREATE VIEW 语句将 SELECT 语句保存为类似于表的可查询对象。TiDB 中的视图是非物化的，这意味着在查询视图时，TiDB 将在内部重写查询，以将视图定义与 SQL 查询结合起来。

12.11.2.27.1 语法图

```
CreateViewStmt ::=
    'CREATE' OrReplace ViewAlgorithm ViewDefiner ViewSQLSecurity 'VIEW' ViewName ViewFieldList '
        ↪ AS' CreateViewSelectOpt ViewCheckOption

OrReplace ::=
    ( 'OR' 'REPLACE' )?

ViewAlgorithm ::=
    ( 'ALGORITHM' '=' ( 'UNDEFINED' | 'MERGE' | 'TEMPTABLE' ) )?

ViewDefiner ::=
    ( 'DEFINER' '=' Username )?

ViewSQLSecurity ::=
    ( 'SQL' 'SECURITY' ( 'DEFINER' | 'INVOKER' ) )?

ViewName ::= TableName

ViewFieldList ::=
    ( '(' Identifier ( ',' Identifier )* ')' )?

ViewCheckOption ::=
    ( 'WITH' ( 'CASCADED' | 'LOCAL' ) 'CHECK' 'OPTION' )?
```

12.11.2.27.2 示例

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, c1 INT NOT NULL);
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
INSERT INTO t1 (c1) VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 5 rows affected (0.03 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
CREATE VIEW v1 AS SELECT * FROM t1 WHERE c1 > 2;
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
SELECT * FROM t1;
```

```
+----+----+
| id | c1 |
+----+----+
|  1 |  1 |
|  2 |  2 |
|  3 |  3 |
|  4 |  4 |
|  5 |  5 |
+----+----+
5 rows in set (0.00 sec)
```

```
SELECT * FROM v1;
```

```
+----+----+
| id | c1 |
+----+----+
|  3 |  3 |
|  4 |  4 |
|  5 |  5 |
+----+----+
3 rows in set (0.00 sec)
```

```
INSERT INTO t1 (c1) VALUES (6);
```

```
Query OK, 1 row affected (0.01 sec)
```

```
SELECT * FROM v1;
```

```
+----+----+
| id | c1 |
+----+----+
|  3 |  3 |
|  4 |  4 |
|  5 |  5 |
|  6 |  6 |
+----+----+
4 rows in set (0.00 sec)
```

```
INSERT INTO v1 (c1) VALUES (7);
```

```
ERROR 1105 (HY000): insert into view v1 is not supported now.
```

12.11.2.27.3 MySQL 兼容性

- 目前 TiDB 中的任何视图都不可被插入，也不可被更新（即不支持 INSERT VIEW，也不支持 UPDATE VIEW）。WITH CHECK OPTION 只做了语法兼容但不生效。
- 目前 TiDB 中的视图不支持 ALTER VIEW，但可以使用 CREATE OR REPLACE 替代。
- 目前 ALGORITHM 字段在 TiDB 中只做了语法兼容但不生效，TiDB 目前只支持 MERGE 算法。

12.11.2.27.4 另请参阅

- DROP VIEW
- CREATE TABLE
- SHOW CREATE TABLE
- DROP TABLE

12.11.2.28 DEALLOCATE

DEALLOCATE 语句用于为服务器端预处理语句提供 SQL 接口。

12.11.2.28.1 语法图

```
DeallocateStmt ::=
    DeallocateSym 'PREPARE' Identifier

DeallocateSym ::=
    'DEALLOCATE'
|   'DROP'

Identifier ::=
    identifier
|   UnReservedKeyword
|   NotKeywordToken
|   TiDBKeyword
```

12.11.2.28.2 示例

```
PREPARE mystmt FROM 'SELECT ? as num FROM DUAL';
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
SET @number = 5;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
EXECUTE mystmt USING @number;
```

```
+------+
| num  |
+------+
| 5    |
+------+
1 row in set (0.00 sec)
```

```
DEALLOCATE PREPARE mystmt;
```

```
Query OK, 0 rows affected (0.00 sec)
```

12.11.2.28.3 MySQL 兼容性

DEALLOCATE 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.28.4 另请参阅

- PREPARE
- EXECUTE

12.11.2.29 DELETE

DELETE 语句用于从指定的表中删除行。

12.11.2.29.1 语法图

```
DeleteFromStmt ::=
    'DELETE' TableOptimizerHints PriorityOpt QuickOptional IgnoreOptional ( 'FROM' ( TableName
        ↪ TableAsNameOpt IndexHintListOpt WhereClauseOptional OrderByOptional LimitClause |
        ↪ TableAliasRefList 'USING' TableRefs WhereClauseOptional ) | TableAliasRefList 'FROM'
        ↪ TableRefs WhereClauseOptional )
```

12.11.2.29.2 示例

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, c1 INT NOT NULL);
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
INSERT INTO t1 (c1) VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 5 rows affected (0.03 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
SELECT * FROM t1;
```

```
+----+----+
| id | c1 |
+----+----+
|  1 |  1 |
|  2 |  2 |
|  3 |  3 |
|  4 |  4 |
|  5 |  5 |
+----+----+
5 rows in set (0.00 sec)
```

```
DELETE FROM t1 WHERE id = 4;
```

```
Query OK, 1 row affected (0.02 sec)
```

```
SELECT * FROM t1;
```

```
+----+----+
| id | c1 |
+----+----+
|  1 |  1 |
|  2 |  2 |
|  3 |  3 |
|  5 |  5 |
+----+----+
4 rows in set (0.00 sec)
```

12.11.2.29.3 MySQL 兼容性

DELETE 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.29.4 另请参阅

- INSERT
- SELECT
- UPDATE
- REPLACE

12.11.2.30 DESC

DESC 语句是EXPLAIN 的别名。包含该语句提供了 MySQL 兼容性。

12.11.2.31 DESCRIBE

DESCRIBE 语句为EXPLAIN 的别名。包含该语句提供了 MySQL 兼容性。

12.11.2.32 DO

DO 语句用于执行表达式，但不返回任何结果。大部分情况下，DO 相当于不返回结果的 SELECT expr, ...,。

> **注意：**
>
> DO 只能执行表达式，所以不是所有能够用 SELECT 的地方都能用 DO 替换。例如 DO id FROM t1 就是不是合法的 SQL 语句，因为它引用了一张表。

DO 在 MySQL 中的一个主要应用场景是存储过程或者触发器。因为 TiDB 当前不支持存储过程和触发器，所以 DO 的实际使用场景较少。

12.11.2.32.1 语法图

```
DoStmt   ::= 'DO' ExpressionList

ExpressionList ::=
    Expression ( ',' Expression )*

Expression ::=
    ( singleAtIdentifier assignmentEq | 'NOT' | Expression ( logOr | 'XOR' | logAnd ) )
        ↪ Expression
|   'MATCH' '(' ColumnNameList ')' 'AGAINST' '(' BitExpr FulltextSearchModifierOpt ')'
|   PredicateExpr ( IsOrNotOp 'NULL' | CompareOp ( ( singleAtIdentifier assignmentEq )?
    ↪ PredicateExpr | AnyOrAll SubSelect ) )* ( IsOrNotOp ( trueKwd | falseKwd | 'UNKNOWN' ) )?
```

12.11.2.32.2 示例

这条 SELECT 语句会暂停执行，但同时也会返回一个结果集。

```
SELECT SLEEP(5);
```

```
+----------+
| SLEEP(5) |
+----------+
|        0 |
+----------+
1 row in set (5.00 sec)
```

如果使用 DO 的话，语句同样会暂停，但不会返回结果集。

```
DO SLEEP(5);
```

```
Query OK, 0 rows affected (5.00 sec)
```

```
DO SLEEP(1), SLEEP(1.5);
```

```
Query OK, 0 rows affected (2.50 sec)
```

12.11.2.32.3 MySQL 兼容性

DO 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.32.4 另请参阅

- SELECT

12.11.2.33 DROP [GLOBAL|SESSION] BINDING

DROP BINDING 语句用于删除指定的 SQL 绑定。绑定可用于将优化器 Hint 插入语句中，而无需更改底层查询。

BINDING 语句可以在 GLOBAL 或者 SESSION 作用域内删除执行计划绑定。在不指定作用域时，默认的作用域为 SESSION。

12.11.2.33.1 语法图

```
DropBindingStmt ::=
    'DROP' GlobalScope 'BINDING' 'FOR' BindableStmt ( 'USING' BindableStmt )?

GlobalScope ::=
    ( 'GLOBAL' | 'SESSION' )?

BindableStmt ::=
    ( SelectStmt | UpdateStmt | InsertIntoStmt | ReplaceIntoStmt | DeleteStmt )
```

12.11.2.33.2 示例

```
CREATE TABLE t1 (
    ->  id INT NOT NULL PRIMARY KEY auto_increment,
    ->  b INT NOT NULL,
    ->  pad VARBINARY(255),
    ->  INDEX(b)
    -> );
Query OK, 0 rows affected (0.07 sec)
```

```
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM dual;
Query OK, 1 row affected (0.01 sec)
Records: 1  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 8 rows affected (0.00 sec)
Records: 8  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 1000 rows affected (0.04 sec)
Records: 1000  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 100000 rows affected (1.74 sec)
Records: 100000  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 100000 rows affected (2.15 sec)
Records: 100000  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 100000 rows affected (2.64 sec)
Records: 100000  Duplicates: 0  Warnings: 0

SELECT SLEEP(1);
+----------+
| SLEEP(1) |
+----------+
|        0 |
+----------+
1 row in set (1.00 sec)

ANALYZE TABLE t1;
Query OK, 0 rows affected (1.33 sec)
```

```
EXPLAIN ANALYZE SELECT * FROM t1 WHERE b = 123;
+--
    ↪ ----------------------------+---------+---------+----------+---------------------+-----------------
    ↪
| id                            | estRows | actRows | task      | access object       |
    ↪ execution info                                                                  | operator info
    ↪                       | memory         | disk |
+--
    ↪ ----------------------------+---------+---------+----------+---------------------+-----------------
    ↪
| IndexLookUp_10                | 583.00  | 297     | root      |                     | time
    ↪ :10.545072ms, loops:2, rpc num: 1, rpc time:398.359µs, proc keys:297  |
    ↪                                     | 109.1484375 KB | N/A  |
| ├─IndexRangeScan_8(Build)     | 583.00  | 297     | cop[tikv] | table:t1, index:b(b) | time:0s
    ↪ , loops:4                                                              | range:[123,123],
    ↪ keep order:false | N/A            | N/A  |
| └─TableRowIDScan_9(Probe)     | 583.00  | 297     | cop[tikv] | table:t1             | time:12
    ↪ ms, loops:4                                                            | keep order:false
    ↪                   | N/A            | N/A  |
+--
    ↪ ----------------------------+---------+---------+----------+---------------------+-----------------
    ↪
3 rows in set (0.02 sec)

CREATE SESSION BINDING FOR
    ->  SELECT * FROM t1 WHERE b = 123
    -> USING
    ->  SELECT * FROM t1 IGNORE INDEX (b) WHERE b = 123;
Query OK, 0 rows affected (0.00 sec)

EXPLAIN ANALYZE  SELECT * FROM t1 WHERE b = 123;
+--
    ↪ ----------------------+----------+---------+----------+--------------+--------------------------------
    ↪
| id                      | estRows   | actRows | task      | access object | execution info
    ↪                                                                  | operator info      |
    ↪ memory        | disk |
+--
    ↪ ----------------------+----------+---------+----------+--------------+--------------------------------
    ↪
| TableReader_7           | 583.00    | 297     | root      |               | time:222.32506ms,
    ↪ loops:2, rpc num: 1, rpc time:222.078952ms, proc keys:301010 | data:Selection_6   |
    ↪ 88.6640625 KB | N/A  |
| └─Selection_6           | 583.00    | 297     | cop[tikv] |               | time:224ms, loops
    ↪ :298                                                          | eq(test.t1.b, 123) | N/A
```

```
    ↪           | N/A  |
|   └─TableFullScan_5     | 301010.00 | 301010  | cop[tikv] | table:t1      | time:220ms, loops
    ↪ :298                                                          | keep order:false    | N/A
    ↪           | N/A  |
+--
    ↪ ----------------------+----------+---------+----------+---------------+-------------------------------
    ↪
3 rows in set (0.22 sec)

SHOW SESSION BINDINGS\G
*************************** 1. row ***************************
Original_sql: select * from t1 where b = ?
    Bind_sql: SELECT * FROM t1 IGNORE INDEX (b) WHERE b = 123
  Default_db: test
      Status: using
 Create_time: 2020-05-22 14:38:03.456
 Update_time: 2020-05-22 14:38:03.456
     Charset: utf8mb4
   Collation: utf8mb4_0900_ai_ci
1 row in set (0.00 sec)

DROP SESSION BINDING FOR SELECT * FROM t1 WHERE b = 123;
Query OK, 0 rows affected (0.00 sec)

EXPLAIN ANALYZE  SELECT * FROM t1 WHERE b = 123;
+--
    ↪ ----------------------------+---------+---------+----------+---------------------+-----------------
    ↪
| id                          | estRows | actRows | task      | access object       |
    ↪ execution info                                                                | operator info
    ↪                     | memory         | disk |
+--
    ↪ ----------------------------+---------+---------+----------+---------------------+-----------------
    ↪
| IndexLookUp_10                | 583.00  | 297     | root      |                     | time
    ↪ :5.31206ms, loops:2, rpc num: 1, rpc time:665.927µs, proc keys:297  |
    ↪                                     | 109.1484375 KB | N/A  |
| ├─IndexRangeScan_8(Build)     | 583.00  | 297     | cop[tikv] | table:t1, index:b(b) | time:0s
    ↪ , loops:4                                                             | range:[123,123], keep
    ↪ order:false | N/A            | N/A  |
| └─TableRowIDScan_9(Probe)     | 583.00  | 297     | cop[tikv] | table:t1             | time:0s
    ↪ , loops:4                                                             | keep order:false
    ↪                   | N/A            | N/A  |
+--
    ↪ ----------------------------+---------+---------+----------+---------------------+-----------------
```

```
    ↪
3 rows in set (0.01 sec)

SHOW SESSION BINDINGS\G
Empty set (0.00 sec)
```

12.11.2.33.3 MySQL 兼容性

`DROP [GLOBAL|SESSION] BINDING` 语句是 TiDB 对 MySQL 语法的扩展。

12.11.2.33.4 另请参阅

- CREATE [GLOBAL|SESSION] BINDING
- SHOW [GLOBAL|SESSION] BINDINGS
- ANALYZE
- Optimizer Hints
- 执行计划管理 (SPM)

12.11.2.34 DROP COLUMN

`DROP COLUMN` 语句用于从指定的表中删除列。在 TiDB 中，`COLUMN` 为在线操作，不会阻塞表中的数据读写。

12.11.2.34.1 语法图

```
AlterTableStmt ::=
    'ALTER' IgnoreOptional 'TABLE' TableName ( AlterTableSpecListOpt AlterTablePartitionOpt | '
        ↪ ANALYZE' 'PARTITION' PartitionNameList ( 'INDEX' IndexNameList )?
        ↪ AnalyzeOptionListOpt )

AlterTableSpec ::=
    TableOptionList
|   'SET' 'TIFLASH' 'REPLICA' LengthNum LocationLabelList
|   'CONVERT' 'TO' CharsetKw ( CharsetName | 'DEFAULT' ) OptCollate
|   'ADD' ( ColumnKeywordOpt IfNotExists ( ColumnDef ColumnPosition | '(' TableElementList ')' )
    ↪ | Constraint | 'PARTITION' IfNotExists NoWriteToBinLogAliasOpt (
    ↪ PartitionDefinitionListOpt | 'PARTITIONS' NUM ) )
|   ( ( 'CHECK' | 'TRUNCATE' ) 'PARTITION' | ( 'OPTIMIZE' | 'REPAIR' | 'REBUILD' ) 'PARTITION'
    ↪ NoWriteToBinLogAliasOpt ) AllOrPartitionNameList
|   'COALESCE' 'PARTITION' NoWriteToBinLogAliasOpt NUM
|   'DROP' ( ColumnKeywordOpt IfExists ColumnName RestrictOrCascadeOpt | 'PRIMARY' 'KEY' |  '
    ↪ PARTITION' IfExists PartitionNameList | ( KeyOrIndex IfExists | 'CHECK' ) Identifier | '
    ↪ FOREIGN' 'KEY' IfExists Symbol )
|   'EXCHANGE' 'PARTITION' Identifier 'WITH' 'TABLE' TableName WithValidationOpt
|   ( 'IMPORT' | 'DISCARD' ) ( 'PARTITION' AllOrPartitionNameList )? 'TABLESPACE'
|   'REORGANIZE' 'PARTITION' NoWriteToBinLogAliasOpt ReorganizePartitionRuleOpt
```

```
|   'ORDER' 'BY' AlterOrderItem ( ',' AlterOrderItem )*
|   ( 'DISABLE' | 'ENABLE' ) 'KEYS'
|   ( 'MODIFY' ColumnKeywordOpt IfExists | 'CHANGE' ColumnKeywordOpt IfExists ColumnName )
    ↪ ColumnDef ColumnPosition
|   'ALTER' ( ColumnKeywordOpt ColumnName ( 'SET' 'DEFAULT' ( SignedLiteral | '(' Expression ')'
    ↪ ) | 'DROP' 'DEFAULT' ) | 'CHECK' Identifier EnforcedOrNot | 'INDEX' Identifier
    ↪ IndexInvisible )
|   'RENAME' ( ( 'COLUMN' | KeyOrIndex ) Identifier 'TO' Identifier | ( 'TO' | '='? | 'AS' )
    ↪ TableName )
|   LockClause
|   AlgorithmClause
|   'FORCE'
|   ( 'WITH' | 'WITHOUT' ) 'VALIDATION'
|   'SECONDARY_LOAD'
|   'SECONDARY_UNLOAD'

ColumnName ::=
    Identifier ( '.' Identifier ( '.' Identifier )? )?
```

12.11.2.34.2 示例

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, col1 INT NOT NULL, col2 INT NOT NULL
    ↪ );
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
INSERT INTO t1 (col1,col2) VALUES (1,1),(2,2),(3,3),(4,4),(5,5);
```

```
Query OK, 5 rows affected (0.02 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
SELECT * FROM t1;
```

```
+----+------+------+
| id | col1 | col2 |
+----+------+------+
|  1 |    1 |    1 |
|  2 |    2 |    2 |
|  3 |    3 |    3 |
|  4 |    4 |    4 |
|  5 |    5 |    5 |
+----+------+------+
5 rows in set (0.01 sec)
```

```
ALTER TABLE t1 DROP COLUMN col1, DROP COLUMN col2;
```

```
ERROR 1105 (HY000): can't run multi schema change
```

```
SELECT * FROM t1;
```

```
+----+------+------+
| id | col1 | col2 |
+----+------+------+
|  1 |    1 |    1 |
|  2 |    2 |    2 |
|  3 |    3 |    3 |
|  4 |    4 |    4 |
|  5 |    5 |    5 |
+----+------+------+
5 rows in set (0.00 sec)
```

```
ALTER TABLE t1 DROP COLUMN col1;
```

```
Query OK, 0 rows affected (0.27 sec)
```

```
SELECT * FROM t1;
```

```
+----+------+
| id | col2 |
+----+------+
|  1 |    1 |
|  2 |    2 |
|  3 |    3 |
|  4 |    4 |
|  5 |    5 |
+----+------+
5 rows in set (0.00 sec)
```

12.11.2.34.3 MySQL 兼容性

- 目前不支持在一条语句中同时删除多个列。
- 目前不支持删除主键列或索引列。

12.11.2.34.4 另请参阅

- ADD COLUMN
- SHOW CREATE TABLE
- CREATE TABLE

12.11.2.35 DROP DATABASE

DROP DATABASE 语句用于永久删除指定的数据库 schema，以及删除所有在 schema 中创建的表和视图。与被删数据库相关联的用户权限不受影响。

12.11.2.35.1 语法图

```
DropDatabaseStmt ::=
    'DROP' 'DATABASE' IfExists DBName

IfExists ::= ( 'IF' 'EXISTS' )?
```

12.11.2.35.2 示例

```
SHOW DATABASES;
```

```
+--------------------+
| Database           |
+--------------------+
| INFORMATION_SCHEMA |
| PERFORMANCE_SCHEMA |
| mysql              |
| test               |
+--------------------+
4 rows in set (0.00 sec)
```

```
DROP DATABASE test;
```

```
Query OK, 0 rows affected (0.25 sec)
```

```
SHOW DATABASES;
```

```
+--------------------+
| Database           |
+--------------------+
| INFORMATION_SCHEMA |
| PERFORMANCE_SCHEMA |
| mysql              |
+--------------------+
3 rows in set (0.00 sec)
```

12.11.2.35.3 MySQL 兼容性

DROP DATABASE 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.35.4 另请参阅

- CREATE DATABASE
- ALTER DATABASE

12.11.2.36 DROP INDEX

DROP INDEX 语句用于从指定的表中删除索引，并在 TiKV 中将空间标记为释放。

12.11.2.36.1 语法图

```
AlterTableDropIndexStmt ::=
    'ALTER' IgnoreOptional 'TABLE' AlterTableDropIndexSpec

IgnoreOptional ::=
    'IGNORE'?

TableName ::=
    Identifier ('.' Identifier)?

AlterTableDropIndexSpec ::=
    'DROP' ( KeyOrIndex | 'FOREIGN' 'KEY' ) IfExists Identifier

KeyOrIndex ::=
    'KEY'
|   'INDEX'

IfExists ::= ( 'IF' 'EXISTS' )?
```

12.11.2.36.2 示例

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, c1 INT NOT NULL);
```

```
Query OK, 0 rows affected (0.10 sec)
```

```
INSERT INTO t1 (c1) VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 5 rows affected (0.02 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
EXPLAIN SELECT * FROM t1 WHERE c1 = 3;
```

```
+-------------------------+----------+-----------+--------------+--------------------------------+
    ↪
| id                      | estRows  | task      | access object | operator info
    ↪                  |
+-------------------------+----------+-----------+--------------+--------------------------------+
    ↪
| TableReader_7           | 10.00    | root      |               | data:Selection_6
    ↪               |
| └─Selection_6           | 10.00    | cop[tikv] |               | eq(test.t1.c1, 3)
    ↪              |
|   └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t1      | keep order:false, stats:
    ↪ pseudo |
+-------------------------+----------+-----------+--------------+--------------------------------+
    ↪
3 rows in set (0.00 sec)
```

```
CREATE INDEX c1 ON t1 (c1);
```

```
Query OK, 0 rows affected (0.30 sec)
```

```
EXPLAIN SELECT * FROM t1 WHERE c1 = 3;
```

```
+------------------------+---------+-----------+-----------------------+-----------------------------------
    ↪
| id                     | estRows | task      | access object         | operator info
    ↪                              |
+------------------------+---------+-----------+-----------------------+-----------------------------------
    ↪
| IndexReader_6          | 0.01    | root      |                       | index:IndexRangeScan_5
    ↪                       |
| └─IndexRangeScan_5     | 0.01    | cop[tikv] | table:t1, index:c1(c1) | range:[3,3], keep
    ↪ order:false, stats:pseudo |
+------------------------+---------+-----------+-----------------------+-----------------------------------
    ↪
2 rows in set (0.00 sec)
```

```
ALTER TABLE t1 DROP INDEX c1;
```

```
Query OK, 0 rows affected (0.30 sec)
```

12.11.2.36.3 MySQL 兼容性

- 默认不支持删除 `PRIMARY KEY`，在开启 `alter-primary-key` 配置项后可支持此功能，详情参考：alter-primary-key。

12.11.2.36.4 另请参阅

- SHOW INDEX
- CREATE INDEX
- ADD INDEX
- RENAME INDEX

12.11.2.37 DROP ROLE

使用 `DROP ROLE` 语句可删除已用 `CREATE ROLE` 语句创建的角色。

12.11.2.37.1 语法图

```
DropRoleStmt ::=
    'DROP' 'ROLE' ( 'IF' 'EXISTS' )? RolenameList

RolenameList ::=
    Rolename ( ',' Rolename )*
```

12.11.2.37.2 示例

创建新角色 `analyticsteam` 和新用户 `jennifer`：

```
$ mysql -uroot

CREATE ROLE analyticsteam;
Query OK, 0 rows affected (0.02 sec)

GRANT SELECT ON test.* TO analyticsteam;
Query OK, 0 rows affected (0.02 sec)

CREATE USER jennifer;
Query OK, 0 rows affected (0.01 sec)

GRANT analyticsteam TO jennifer;
Query OK, 0 rows affected (0.01 sec)
```

需要注意的是，默认情况下，用户 `jennifer` 需要执行 `SET ROLE analyticsteam` 语句才能使用与角色相关联的权限：

```
$ mysql -ujennifer

SHOW GRANTS;
+--------------------------------------------+
| Grants for User                            |
+--------------------------------------------+
```

```
| GRANT USAGE ON *.* TO 'jennifer'@'%'         |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+----------------------------------------------+
2 rows in set (0.00 sec)

SHOW TABLES in test;
ERROR 1044 (42000): Access denied for user 'jennifer'@'%' to database 'test'
SET ROLE analyticsteam;
Query OK, 0 rows affected (0.00 sec)

SHOW GRANTS;
+----------------------------------------------+
| Grants for User                              |
+----------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'         |
| GRANT Select ON test.* TO 'jennifer'@'%'     |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+----------------------------------------------+
3 rows in set (0.00 sec)

SHOW TABLES IN test;
+----------------+
| Tables_in_test |
+----------------+
| t1             |
+----------------+
1 row in set (0.00 sec)
```

执行 `SET DEFAULT ROLE` 语句将用户 `jennifer` 与某一角色相关联，这样该用户无需执行 `SET ROLE` 语句就能拥有与角色相关联的权限。

```
$ mysql -uroot

SET DEFAULT ROLE analyticsteam TO jennifer;
Query OK, 0 rows affected (0.02 sec)
```

```
$ mysql -ujennifer

SHOW GRANTS;
+----------------------------------------------+
| Grants for User                              |
+----------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'         |
| GRANT Select ON test.* TO 'jennifer'@'%'     |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+----------------------------------------------+
```

```
3 rows in set (0.00 sec)

SHOW TABLES IN test;
+----------------+
| Tables_in_test |
+----------------+
| t1             |
+----------------+
1 row in set (0.00 sec)
```

删除角色 analyticsteam：

```
$ mysql -uroot

DROP ROLE analyticsteam;
Query OK, 0 rows affected (0.02 sec)
```

Jennifer 不再具有与 analyticsteam 关联的默认角色，或不能再将 analyticsteam 设为启用角色：

```
$ mysql -ujennifer

SHOW GRANTS;
+--------------------------------------+
| Grants for User                      |
+--------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%' |
+--------------------------------------+
1 row in set (0.00 sec)

SET ROLE analyticsteam;
ERROR 3530 (HY000): `analyticsteam`@`%` is is not granted to jennifer@%
```

12.11.2.37.3 MySQL 兼容性

DROP ROLE 语句与 MySQL 8.0 的角色功能完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.37.4 另请参阅

- CREATE ROLE
- GRANT <role>
- REVOKE <role>
- SET ROLE
- SET DEFAULT ROLE
- 基于角色的访问控制

12.11.2.38 DROP SEQUENCE

`DROP SEQUENCE` 语句用于删除序列对象。

12.11.2.38.1 语法图

```
DropSequenceStmt ::=
    'DROP' 'SEQUENCE' IfExists TableNameList

IfExists ::= ( 'IF' 'EXISTS' )?

TableNameList ::=
    TableName ( ',' TableName )*

TableName ::=
    Identifier ('.' Identifier)?
```

12.11.2.38.2 示例

```
DROP SEQUENCE seq;
```

```
Query OK, 0 rows affected (0.10 sec)
```

```
DROP SEQUENCE seq, seq2;
```

```
Query OK, 0 rows affected (0.03 sec)
```

12.11.2.38.3 MySQL 兼容性

该语句是 TiDB 的扩展，序列的实现借鉴自 MariaDB。

12.11.2.38.4 另请参阅

- CREATE SEQUENCE
- SHOW CREATE SEQUENCE

12.11.2.39 DROP STATS

`DROP STATS` 语句用于从当前所选定的数据库中删除选定表的统计信息。

12.11.2.39.1 语法图

```
DropStatsStmt ::=
    'DROP' 'STATS' TableName

TableName ::=
    Identifier ('.' Identifier)?
```

12.11.2.39.2 示例

```
CREATE TABLE t(a INT);
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
SHOW STATS_META WHERE db_name='test' and table_name='t';
```

```
+---------+------------+----------------+---------------------+--------------+-----------+
| Db_name | Table_name | Partition_name | Update_time         | Modify_count | Row_count |
+---------+------------+----------------+---------------------+--------------+-----------+
| test    | t          |                | 2020-05-25 20:34:33 |            0 |         0 |
+---------+------------+----------------+---------------------+--------------+-----------+
1 row in set (0.00 sec)
```

```
DROP STATS t;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
SHOW STATS_META WHERE db_name='test' and table_name='t';
```

```
Empty set (0.00 sec)
```

12.11.2.39.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

12.11.2.39.4 另请参阅

- 统计信息简介

12.11.2.40 DROP TABLE

`DROP TABLE` 语句用于从当前所选的数据库中删除表。如果表不存在则会报错，除非使用 `IF EXISTS` 修饰符。

12.11.2.40.1 语法图

```
DropTableStmt ::=
    'DROP' OptTemporary TableOrTables IfExists TableNameList RestrictOrCascadeOpt

TableOrTables ::=
    'TABLE'
|   'TABLES'

TableNameList ::=
    TableName ( ',' TableName )*
```

12.11.2.40.2 示例

```
CREATE TABLE t1 (a INT);
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
DROP TABLE t1;
```

```
Query OK, 0 rows affected (0.22 sec)
```

```
DROP TABLE table_not_exists;
```

```
ERROR 1051 (42S02): Unknown table 'test.table_not_exists'
```

```
DROP TABLE IF EXISTS table_not_exists;
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
CREATE VIEW v1 AS SELECT 1;
```

```
Query OK, 0 rows affected (0.10 sec)
```

```
DROP TABLE v1;
```

```
Query OK, 0 rows affected (0.23 sec)
```

12.11.2.40.3 MySQL 兼容性

- 在尝试删除不存在的表时，使用 `IF EXISTS` 删除表不会返回警告。Issue #7867
- 目前 `RESTRICT` 和 `CASCADE` 仅在语法上支持。

12.11.2.40.4 另请参阅

- CREATE TABLE
- SHOW CREATE TABLE
- SHOW TABLES

12.11.2.41 DROP USER

DROP USER 语句用于从 TiDB 系统数据库中删除用户。如果用户不存在，使用关键词 IF EXISTS 可避免出现警告。执行 DROP USER 语句需要拥有 CREATE USER 权限。

12.11.2.41.1 语法图

```
DropUserStmt ::=
    'DROP' 'USER' ( 'IF' 'EXISTS' )? UsernameList

Username ::=
    StringName ('@' StringName | singleAtIdentifier)? | 'CURRENT_USER' OptionalBraces
```

12.11.2.41.2 示例

```
DROP USER 'idontexist';
```

```
ERROR 1396 (HY000): Operation DROP USER failed for idontexist@%
```

```
DROP USER IF EXISTS 'idontexist';
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
CREATE USER 'newuser' IDENTIFIED BY 'mypassword';
```

```
Query OK, 1 row affected (0.02 sec)
```

```
GRANT ALL ON test.* TO 'newuser';
```

```
Query OK, 0 rows affected (0.03 sec)
```

```
SHOW GRANTS FOR 'newuser';
```

```
+-------------------------------------------------+
| Grants for newuser@%                            |
+-------------------------------------------------+
| GRANT USAGE ON *.* TO 'newuser'@'%'             |
| GRANT ALL PRIVILEGES ON test.* TO 'newuser'@'%' |
+-------------------------------------------------+
2 rows in set (0.00 sec)
```

```
REVOKE ALL ON test.* FROM 'newuser';
```

```
Query OK, 0 rows affected (0.03 sec)
```

```
SHOW GRANTS FOR 'newuser';
```

```
+------------------------------------+
| Grants for newuser@%               |
+------------------------------------+
| GRANT USAGE ON *.* TO 'newuser'@'%' |
+------------------------------------+
1 row in set (0.00 sec)
```

```
DROP USER 'newuser';
```

```
Query OK, 0 rows affected (0.14 sec)
```

```
SHOW GRANTS FOR 'newuser';
```

```
ERROR 1141 (42000): There is no such grant defined for user 'newuser' on host '%'
```

12.11.2.41.3 MySQL 兼容性

- 在 TiDB 中删除不存在的用户时，使用 `IF EXISTS` 可避免出现警告。Issue #10196。

12.11.2.41.4 另请参阅

- CREATE USER
- ALTER USER
- SHOW CREATE USER
- Privilege Management

12.11.2.42 DROP VIEW

`DROP VIEW` 语句用于从当前所选定的数据库中删除视图对象。视图所引用的基表不受影响。

12.11.2.42.1 语法图

```
DropViewStmt ::=
    'DROP' 'VIEW' ( 'IF' 'EXISTS' )? TableNameList RestrictOrCascadeOpt

TableNameList ::=
    TableName ( ',' TableName )*
```

```
TableName ::=
    Identifier ('.' Identifier)?
```

12.11.2.42.2 示例

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, c1 INT NOT NULL);
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
INSERT INTO t1 (c1) VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 5 rows affected (0.03 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
CREATE VIEW v1 AS SELECT * FROM t1 WHERE c1 > 2;
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
SELECT * FROM t1;
```

```
+----+----+
| id | c1 |
+----+----+
|  1 |  1 |
|  2 |  2 |
|  3 |  3 |
|  4 |  4 |
|  5 |  5 |
+----+----+
5 rows in set (0.00 sec)
```

```
SELECT * FROM v1;
```

```
+----+----+
| id | c1 |
+----+----+
|  3 |  3 |
|  4 |  4 |
|  5 |  5 |
+----+----+
3 rows in set (0.00 sec)
```

```
DROP VIEW v1;
```

```
Query OK, 0 rows affected (0.23 sec)
```

```
SELECT * FROM t1;
```

```
+----+----+
| id | c1 |
+----+----+
|  1 |  1 |
|  2 |  2 |
|  3 |  3 |
|  4 |  4 |
|  5 |  5 |
+----+----+
5 rows in set (0.00 sec)
```

12.11.2.42.3 MySQL 兼容性

DROP VIEW 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.42.4 See also

- CREATE VIEW
- DROP TABLE

12.11.2.43 EXECUTE

EXECUTE 语句为服务器端预处理语句提供 SQL 接口。

12.11.2.43.1 语法图

```
ExecuteStmt ::=
    'EXECUTE' Identifier ( 'USING' UserVariable ( ',' UserVariable )* )?
```

12.11.2.43.2 示例

```
PREPARE mystmt FROM 'SELECT ? as num FROM DUAL';
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
SET @number = 5;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
EXECUTE mystmt USING @number;
```

```
+------+
| num  |
+------+
| 5    |
+------+
1 row in set (0.00 sec)
```

```
DEALLOCATE PREPARE mystmt;
```

```
Query OK, 0 rows affected (0.00 sec)
```

12.11.2.43.3 MySQL 兼容性

EXECUTE 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.43.4 另请参阅

- PREPARE
- DEALLOCATE

12.11.2.44 EXPLAIN ANALYZE

EXPLAIN ANALYZE 语句的工作方式类似于 EXPLAIN，主要区别在于前者实际上会执行语句。这样可以将查询计划中的估计值与执行时所遇到的实际值进行比较。如果估计值与实际值显著不同，那么应考虑在受影响的表上运行 ANALYZE TABLE。

> **注意：**
>
> 在使用 EXPLAIN ANALYZE 执行 DML 语句时，数据的修改操作会被正常执行。但目前 DML 语句还无法展示执行计划。

12.11.2.44.1 语法图

```
ExplainSym ::=
    'EXPLAIN'
|   'DESCRIBE'
|    'DESC'

ExplainStmt ::=
```

```
    ExplainSym ( TableName ColumnName? | 'ANALYZE'? ExplainableStmt | 'FOR' 'CONNECTION' NUM | '
        ↪ FORMAT' '=' ( stringLit | ExplainFormatType ) ( 'FOR' 'CONNECTION' NUM |
        ↪ ExplainableStmt ) )

ExplainableStmt ::=
    SelectStmt
|   DeleteFromStmt
|   UpdateStmt
|   InsertIntoStmt
|   ReplaceIntoStmt
|   UnionStmt
```

12.11.2.44.2 EXPLAIN ANALYZE 输出格式

和 EXPLAIN 不同，EXPLAIN ANALYZE 会执行对应的 SQL 语句，记录其运行时信息，和执行计划一并返回出来。因此，可以将 EXPLAIN ANALYZE 视为 EXPLAIN 语句的扩展。EXPLAIN ANALYZE 语句的返回结果相比 EXPLAIN，增加了 actRows，execution info，memory，disk 这几列信息：

属性名	含义
actRows	算子实际输出的数据条数。
execution info	算子的实际执行信息。time 表示从进入算子到离开算子的全部 wall time，包括所有子算子操作的全部执行时间。如果该算子被父算子多次调用 (loops)，这个时间就是累积的时间。loops 是当前算子被父算子调用的次数。
memory	算子占用内存空间的大小。
disk	算子占用磁盘空间的大小。

12.11.2.44.3 示例

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, c1 INT NOT NULL);
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
INSERT INTO t1 (c1) VALUES (1), (2), (3);
```

```
Query OK, 3 rows affected (0.02 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
EXPLAIN ANALYZE SELECT * FROM t1 WHERE id = 1;
```

```
+-------------+---------+---------+------+---------------+-----------------------------------------------------------
    ↪
```

```
| id          | estRows | actRows | task | access object | execution info
    ↪                                                  | operator info | memory | disk |
+-------------+---------+---------+------+---------------+-----------------------------------------------------------
    ↪
| Point_Get_1 | 1.00    | 1       | root | table:t1      | time:757.205µs, loops:2, Get:{num_rpc
    ↪ :1, total_time:697.051µs} | handle:1      | N/A    | N/A  |
+-------------+---------+---------+------+---------------+-----------------------------------------------------------
    ↪
1 row in set (0.01 sec)
```

```
EXPLAIN ANALYZE SELECT * FROM t1;
```

```
+------------------+---------+---------+-----------+--------------+--------------------------------------------------
    ↪
| id               | estRows | actRows | task      | access object | execution info
    ↪
    ↪ | operator info                  | memory    | disk |
+------------------+---------+---------+-----------+--------------+--------------------------------------------------
    ↪
| TableReader_5    | 13.00   | 13      | root      |               | time:923.459µs, loops:2,
    ↪ cop_task: {num: 4, max: 839.788µs, min: 779.374µs, avg: 810.926µs, p95: 839.788µs,
    ↪ max_proc_keys: 12, p95_proc_keys: 12, rpc_num: 4, rpc_time: 3.116964ms,
    ↪ copr_cache_hit_ratio: 0.00} | data:TableFullScan_4           | 632 Bytes | N/A  |
| └─TableFullScan_4 | 13.00   | 13      | cop[tikv] | table:t1      | proc max:0s, min:0s, p80:0
    ↪ s, p95:0s, iters:4, tasks:4
    ↪
    ↪ | keep order:false, stats:pseudo | N/A       | N/A  |
+------------------+---------+---------+-----------+--------------+--------------------------------------------------
    ↪
2 rows in set (0.00 sec)
```

12.11.2.44.4 算子执行信息介绍

`execution info` 信息除了基本的 `time` 和 `loop` 信息外，还包含算子特有的执行信息，主要包含了该算子发送 RPC 请求的耗时信息以及其他步骤的耗时。

Point_Get

`Point_Get` 算子可能包含以下执行信息：

- `Get:{num_rpc:1, total_time:697.051µs}`：向 TiKV 发送 `Get` 类型的 RPC 请求的数量（`num_rpc`）和所有 RPC 请求的总耗时（`total_time`）。
- `ResolveLock:{num_rpc:1, total_time:12.117495ms}`：读数据遇到锁后，进行 resolve lock 的时间。一般在读写冲突的场景下会出现。
- `regionMiss_backoff:{num:11, total_time:2010 ms},tikvRPC_backoff:{num:11, total_time:10691 ms }`：RPC 请求失败后，会在等待 backoff 的时间后重试，包括了 backoff 的类型（如 regionMiss，tikvRPC），backoff 等待的总时间 (total_time) 和 backoff 的总次数 (num)。

Batch_Point_Get

Batch_Point_get 算子的执行信息和 Point_Get 算子类似，不过 Batch_Point_Get 一般向 TiKV 发送 BatchGet 类型的 RPC 请求来读取数据。

BatchGet:{num_rpc:2, total_time:83.13µs}：向 TiKV 发送 BatchGet 类型的 RPC 请求的数量（num_rpc）和所有 RPC 请求的总耗时（total_time）。

TableReader

TableReader 算子可能包含以下执行信息：

```
cop_task: {num: 6, max: 1.07587ms, min: 844.312µs, avg: 919.601µs, p95: 1.07587ms, max_proc_keys:
    ↪  16, p95_proc_keys: 16, tot_proc: 1ms, tot_wait: 1ms, rpc_num: 6, rpc_time: 5.313996ms,
    ↪ copr_cache_hit_ratio: 0.00}
```

- cop_task：包含 cop task 的相关信息，如：
  - num：cop task 的数量
  - max,min,avg,p95：所有 cop task 中执行时间的最大值，最小值，平均值和 P95 值。
  - max_proc_keys, p95_proc_keys：所有 cop task 中 tikv 扫描 kv 数据的最大值，P95 值，如果 max 和 p95 的值差距很大，说明数据分布不太均匀。
  - rpc_num, rpc_time：向 TiKV 发送 Cop 类型的 RPC 请求总数量和总时间。
  - copr_cache_hit_ratio：cop task 请求的 Coprocessor Cache 缓存命中率。Coprocessor Cache 配置。
- backoff：包含不同类型的 backoff 以及等待总耗时。

Insert

Insert 算子可能包含以下执行信息：

```
prepare:109.616µs, check_insert:{total_time:1.431678ms, mem_insert_time:667.878µs, prefetch:763.8
    ↪ µs, rpc:{BatchGet:{num_rpc:1, total_time:699.166µs},Get:{num_rpc:1, total_time:378.276µs
    ↪ }}}
```

- prepare：准备写入前的耗时，包括表达式，默认值相关的计算等。
- check_insert：这个信息一般出现在 insert ignore 和 insert on duplicate 语句中，包含冲突检查和写入 TiDB 事务缓存的耗时。注意，这个耗时不包含事务提交的耗时。具体包含以下信息：
  - total_time：check_insert 步骤的总耗时。
  - mem_insert_time：将数据写入 TiDB 事务缓存的耗时。
  - prefetch：从 TiKV 中获取需要检查冲突的数据的耗时，该步骤主要是向 TiKV 发送 BatchGet 类型的 RPC 请求的获取数据。
  - rpc：向 TiKV 发送 RPC 请求的总耗时，一般包含 BatchGet 和 Get 两种类型的 RPC 耗时，其中：
    * BatchGet 请求是 prefetch 步骤发送的 RPC 请求。
    * Get 请求是 insert on duplicate 语句在执行 duplicate update 时发送的 RPC 请求。
- backoff：包含不同类型的 backoff 以及等待总耗时。

IndexJoin

IndexJoin 算子有 1 个 outer worker 和 N 个 inner worker 并行执行，其 join 结果的顺序和 outer table 的顺序一致，具体执行流程如下：

1. Outer worker 读取 N 行 outer table 的数据，然后包装成一个 task 发送给 result channel 和 inner worker channel。
2. Inner worker 从 inner worker channel 里面接收 task，然后根据 task 生成需要读取 inner table 的 key ranges 范围，然后读取相应范围的 inner table 行数据，并生成一个 inner table row 的 hash table。
3. IndexJoin 的主线程从 result channel 中接收 task，然后等待 inner worker 执行完这个 task。
4. IndexJoin 的主线程用 outer table rows 和 inner table rows 的 hash table 做 join 。

IndexJoin 算子包含以下执行信息：

```
inner:{total:4.297515932s, concurrency:5, task:17, construct:97.96291ms, fetch:4.164310088s,
    ↪ build:35.219574ms}, probe:53.574945ms
```

- inner：inner worker 的执行信息，具体如下：
  - total：inner worker 的总耗时。
  - concurrency：inner worker 的数量。
  - task：inner worker 处理 task 的总数量。
  - construct：inner worker 读取 task 对应的 inner table rows 之前的准备时间。
  - fetch：inner worker 读取 inner table rows 的总耗时。
  - build: inner worker 构造 inner table rows 对应的 hash table 的总耗时。
- probe：IndexJoin 主线程用 outer table rows 和 inner table rows 的 hash table 做 join 的总耗时。

IndexHashJoin

IndexHashJoin 算子和 IndexJoin 算子执行流程类似，也有 1 个 outer worker 和 N 个 inner worker 并行执行，但是其 join 结果的顺序是不和 outer table 一致。具体执行流程如下：

1. Outer worker 读取 N 行 out table 的数据，然后包装成一个 task 发送给 inner worker channel。
2. Inner worker 从 inner worker channel 里面接收 task，然后做以下三件事情，其中步骤 a 和 b 是并行执行。
   a. 用 outer table rows 生成一个 hash table。
   b. 根据 task 生成 key 的范围后，读取 inner table 相应范围的行数据。
   c. 用 inner table rows 和 outer table rows 的 hash table 做 join，然后把 join 结果发送给 result channel。
3. IndexHashJoin 的主线程从 result channel 中接收 join 结果。

IndexHashJoin 算子包含以下执行信息：

```
inner:{total:4.429220003s, concurrency:5, task:17, construct:96.207725ms, fetch:4.239324006s,
    ↪ build:24.567801ms, join:93.607362ms}
```

- inner：inner worker 的执行信息，具体如下：
  - total：inner worker 的总耗时。

  - concurrency：inner worker 的数量。
  - task：inner worker 处理 task 的总数量。
  - construct：inner worker 读取 task 对应的 inner table rows 之前的准备时间。
  - fetch：inner worker 读取 inner table rows 的总耗时。
  - build: inner worker 构造 outer table rows 对应的 hash table 的总耗时。
  - join: inner worker 用 inner table rows 和 outer table rows 的 hash table 做 join 的总耗时。

HashJoin

HashJoin 算子有一个 inner worker，一个 outer worker 和 N 个 join worker，其具体执行逻辑如下：

1. inner worker 读取 inner table rows 并构造 hash table。
2. outer worker 读取 outer table rows, 然后包装成 task 发送给 join worker。
3. 等待第 1 步的 hash table 构造完成。
4. join worker 用 task 里面的 outer table rows 和 hash table 做 join，然后把 join 结果发送给 result channel。
5. HashJoin 的主线程从 result channel 中接收 join 结果。

HashJoin 算子包含以下执行信息：

```
build_hash_table:{total:146.071334ms, fetch:110.338509ms, build:35.732825ms}, probe:{concurrency
    ↪ :5, total:857.162518ms, max:171.48271ms, probe:125.341665ms, fetch:731.820853ms}
```

- build_hash_table: 读取 inner table 的数据并构造 hash table 的执行信息：
  - total：总耗时。
  - fetch：读取 inner table 数据的总耗时。
  - build：构造 hash table 的总耗时。
- probe: join worker 的执行信息：
  - concurrency：join worker 的数量。
  - total：所有 join worker 执行的总耗时。
  - max：单个 join worker 执行的最大耗时。
  - probe: 用 outer table rows 和 hash table 做 join 的总耗时。
  - fetch：join worker 等待读取 outer table rows 数据的总耗时。

lock_keys 执行信息

在悲观事务中执行 DML 语句时，算子的执行信息还可能包含 lock_keys 的执行信息，示例如下：

```
lock_keys: {time:94.096168ms, region:6, keys:8, lock_rpc:274.503214ms, rpc_count:6}
```

- time：执行 lock_keys 操作的总耗时。
- region：执行 lock_keys 操作涉及的 Region 数量。
- keys：需要 Lock 的 Key 的数量。
- lock_rpc：向 TiKV 发送 Lock 类型的 RPC 总耗时。因为可以并行发送多个 RPC 请求，所以总 RPC 耗时可能比 lock_keys 操作总耗时大。
- rpc_count：向 TiKV 发送 Lock 类型的 RPC 总数量。

commit_txn 执行信息

在 autocommit=1 的事务中执行写入类型的 DML 语句时，算子的执行信息还会包括事务提交的耗时信息，示例如下：

```
commit_txn: {prewrite:48.564544ms, wait_prewrite_binlog:47.821579, get_commit_ts:4.277455ms,
    ↪ commit:50.431774ms, region_num:7, write_keys:16, write_byte:536}
```

- prewrite：事务 2PC 提交阶段中 prewrite 阶段的耗时。
- wait_prewrite_binlog:：等待写 prewrite Binlog 的耗时。
- get_commit_ts：获取事务提交时间戳的耗时。
- commit：事务 2PC 提交阶段中，commit 阶段的耗时。
- write_keys：事务中写入 key 的数量。
- write_byte：事务中写入 key-value 的总字节数量，单位是 byte。

12.11.2.44.5 MySQL 兼容性

EXPLAIN ANALYZE 是 MySQL 8.0 的功能，但该语句在 MySQL 中的输出格式和可能的执行计划都与 TiDB 有较大差异。

12.11.2.44.6 另请参阅

- Understanding the Query Execution Plan
- EXPLAIN
- ANALYZE TABLE
- TRACE

12.11.2.45 EXPLAIN

EXPLAIN 语句仅用于显示查询的执行计划，而不执行查询。EXPLAIN ANALYZE 可执行查询，补充 EXPLAIN 语句。如果 EXPLAIN 的输出与预期结果不匹配，可考虑在查询的每个表上执行 ANALYZE TABLE。

语句 DESC 和 DESCRIBE 是 EXPLAIN 的别名。EXPLAIN <tableName> 的替代用法记录在SHOW [FULL] COLUMNS FROM 下。

TiDB 支持 EXPLAIN [options] FOR CONNECTION connection_id，但与 MySQL 的 EXPLAIN FOR 有一些区别，请参见EXPLAIN FOR CONNECTION。

12.11.2.45.1 语法图

```
ExplainSym ::=
    'EXPLAIN'
|   'DESCRIBE'
|   'DESC'

ExplainStmt ::=
```

```
    ExplainSym ( TableName ColumnName? | 'ANALYZE'? ExplainableStmt | 'FOR' 'CONNECTION' NUM | '
        ↪ FORMAT' '=' ( stringLit | ExplainFormatType ) ( 'FOR' 'CONNECTION' NUM |
        ↪ ExplainableStmt ) )

ExplainableStmt ::=
    SelectStmt
|   DeleteFromStmt
|   UpdateStmt
|   InsertIntoStmt
|   ReplaceIntoStmt
|   UnionStmt
```

12.11.2.45.2 EXPLAIN 输出格式

> **注意：**
>
> 使用 MySQL 客户端连接到 TiDB 时，为避免输出结果在终端中换行，可先执行 `pager less -S` 命令。执行命令后，新的 EXPLAIN 的输出结果不再换行，可按右箭头 → 键水平滚动阅读输出结果。

目前 TiDB 的 EXPLAIN 会输出 5 列，分别是：`id`，`estRows`，`task`，`access object`，`operator info`。执行计划中每个算子都由这 5 列属性来描述，EXPLAIN 结果中每一行描述一个算子。每个属性的具体含义如下：

属性名	含义
id	算子的 ID，是算子在整个执行计划中唯一的标识。在 TiDB 2.1 中，ID 会格式化地显示算子的树状结构。数据从孩子结点流向父亲结点，每个算子的父亲结点有且仅有一个。
estRows	算子预计将会输出的数据条数，基于统计信息以及算子的执行逻辑估算而来。在 4.0 之前叫 count。
task	算子属于的 task 种类。目前的执行计划分成为两种 task，一种叫 root task，在 tidb-server 上执行，一种叫 cop task，在 TiKV 或者 TiFlash 上并行执行。当前的执行计划在 task 级别的拓扑关系是一个 root task 后面可以跟许多 cop task，root task 使用 cop task 的输出结果作为输入。cop task 中执行的也即是 TiDB 下推到 TiKV 或者 TiFlash 上的任务，每个 cop task 分散在 TiKV 或者 TiFlash 集群中，由多个进程共同执行。
access object	算子所访问的数据项信息。包括表 `table`，表分区 `partition` 以及使用的索引 `index`（如果有）。只有直接访问数据的算子才拥有这些信息。
operator info	算子的其它信息。各个算子的 operator info 各有不同，可参考下面的示例解读。

12.11.2.45.3 示例

```
EXPLAIN SELECT 1;
```

```
+-------------------+---------+------+---------------+---------------+
| id                | estRows | task | access object | operator info |
+-------------------+---------+------+---------------+---------------+
| Projection_3      | 1.00    | root |               | 1->Column#1   |
| └─TableDual_4     | 1.00    | root |               | rows:1        |
+-------------------+---------+------+---------------+---------------+
2 rows in set (0.00 sec)
```

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, c1 INT NOT NULL);
```

```
Query OK, 0 rows affected (0.10 sec)
```

```
INSERT INTO t1 (c1) VALUES (1), (2), (3);
```

```
Query OK, 3 rows affected (0.02 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
EXPLAIN SELECT * FROM t1 WHERE id = 1;
```

```
+-------------+---------+------+---------------+---------------+
| id          | estRows | task | access object | operator info |
+-------------+---------+------+---------------+---------------+
| Point_Get_1 | 1.00    | root | table:t1      | handle:1      |
+-------------+---------+------+---------------+---------------+
1 row in set (0.00 sec)
```

```
DESC SELECT * FROM t1 WHERE id = 1;
```

```
+-------------+---------+------+---------------+---------------+
| id          | estRows | task | access object | operator info |
+-------------+---------+------+---------------+---------------+
| Point_Get_1 | 1.00    | root | table:t1      | handle:1      |
+-------------+---------+------+---------------+---------------+
1 row in set (0.00 sec)
```

```
DESCRIBE SELECT * FROM t1 WHERE id = 1;
```

```
+-------------+---------+------+---------------+---------------+
| id          | estRows | task | access object | operator info |
+-------------+---------+------+---------------+---------------+
| Point_Get_1 | 1.00    | root | table:t1      | handle:1      |
+-------------+---------+------+---------------+---------------+
1 row in set (0.00 sec)
```

```
EXPLAIN INSERT INTO t1 (c1) VALUES (4);
```

```
+----------+---------+------+---------------+---------------+
| id       | estRows | task | access object | operator info |
+----------+---------+------+---------------+---------------+
| Insert_1 | N/A     | root |               | N/A           |
+----------+---------+------+---------------+---------------+
1 row in set (0.00 sec)
```

```
EXPLAIN UPDATE t1 SET c1=5 WHERE c1=3;
```

```
+--------------------------+---------+-----------+--------------+--------------------------------+
    ↪
| id                       | estRows | task      | access object | operator info
    ↪                  |
+--------------------------+---------+-----------+--------------+--------------------------------+
    ↪
| Update_4                 | N/A     | root      |               | N/A
    ↪                            |
| └─TableReader_8           | 0.00    | root      |               | data:Selection_7
    ↪               |
|   └─Selection_7           | 0.00    | cop[tikv] |               | eq(test.t1.c1, 3)
    ↪              |
|     └─TableFullScan_6     | 3.00    | cop[tikv] | table:t1      | keep order:false, stats:
    ↪ pseudo |
+--------------------------+---------+-----------+--------------+--------------------------------+
    ↪
4 rows in set (0.00 sec)
```

```
EXPLAIN DELETE FROM t1 WHERE c1=3;
```

```
+--------------------------+---------+-----------+--------------+--------------------------------+
    ↪
| id                       | estRows | task      | access object | operator info
    ↪                  |
+--------------------------+---------+-----------+--------------+--------------------------------+
    ↪
| Delete_4                 | N/A     | root      |               | N/A
    ↪                            |
| └─TableReader_8           | 0.00    | root      |               | data:Selection_7
    ↪               |
|   └─Selection_7           | 0.00    | cop[tikv] |               | eq(test.t1.c1, 3)
    ↪              |
|     └─TableFullScan_6     | 3.00    | cop[tikv] | table:t1      | keep order:false, stats:
    ↪ pseudo |
```

```
+---------------------------+---------+-----------+--------------+--------------------------------+
    ↪
4 rows in set (0.01 sec)
```

如果未指定 FORMAT，或未指定 FORMAT ="row"，那么 EXPLAIN 语句将以表格格式输出结果。更多信息，可参阅 Understand the Query Execution Plan。

除 MySQL 标准结果格式外，TiDB 还支持 DotGraph。需按照下列所示指定 FORMAT ="dot"：

```
create table t(a bigint, b bigint);
desc format = "dot" select A.a, B.b from t A join t B on A.a > B.b where A.a < 10;
```

```
+-----------------------------------------------------------------------------------------------------------------------------------------------------------------
    ↪
| dot contents
    ↪
    ↪ |
+-----------------------------------------------------------------------------------------------------------------------------------------------------------------
    ↪
|
digraph Projection_8 {
subgraph cluster8{
node [style=filled, color=lightgrey]
color=black
label = "root"
"Projection_8" -> "HashJoin_9"
"HashJoin_9" -> "TableReader_13"
"HashJoin_9" -> "Selection_14"
"Selection_14" -> "TableReader_17"
}
subgraph cluster12{
node [style=filled, color=lightgrey]
color=black
label = "cop"
"Selection_12" -> "TableFullScan_11"
}
subgraph cluster16{
node [style=filled, color=lightgrey]
color=black
label = "cop"
"Selection_16" -> "TableFullScan_15"
}
"TableReader_13" -> "Selection_12"
"TableReader_17" -> "Selection_16"
}
 |
```

```
+------------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

如果你的计算机上安装了 dot 程序（在 graphviz 包中），可使用以下方法生成 PNG 文件：

```
dot xx.dot -T png -O
```

The xx.dot is the result returned by the above statement.

如果你的计算机上未安装 dot 程序，可将结果复制到 本网站 以获取树形图：

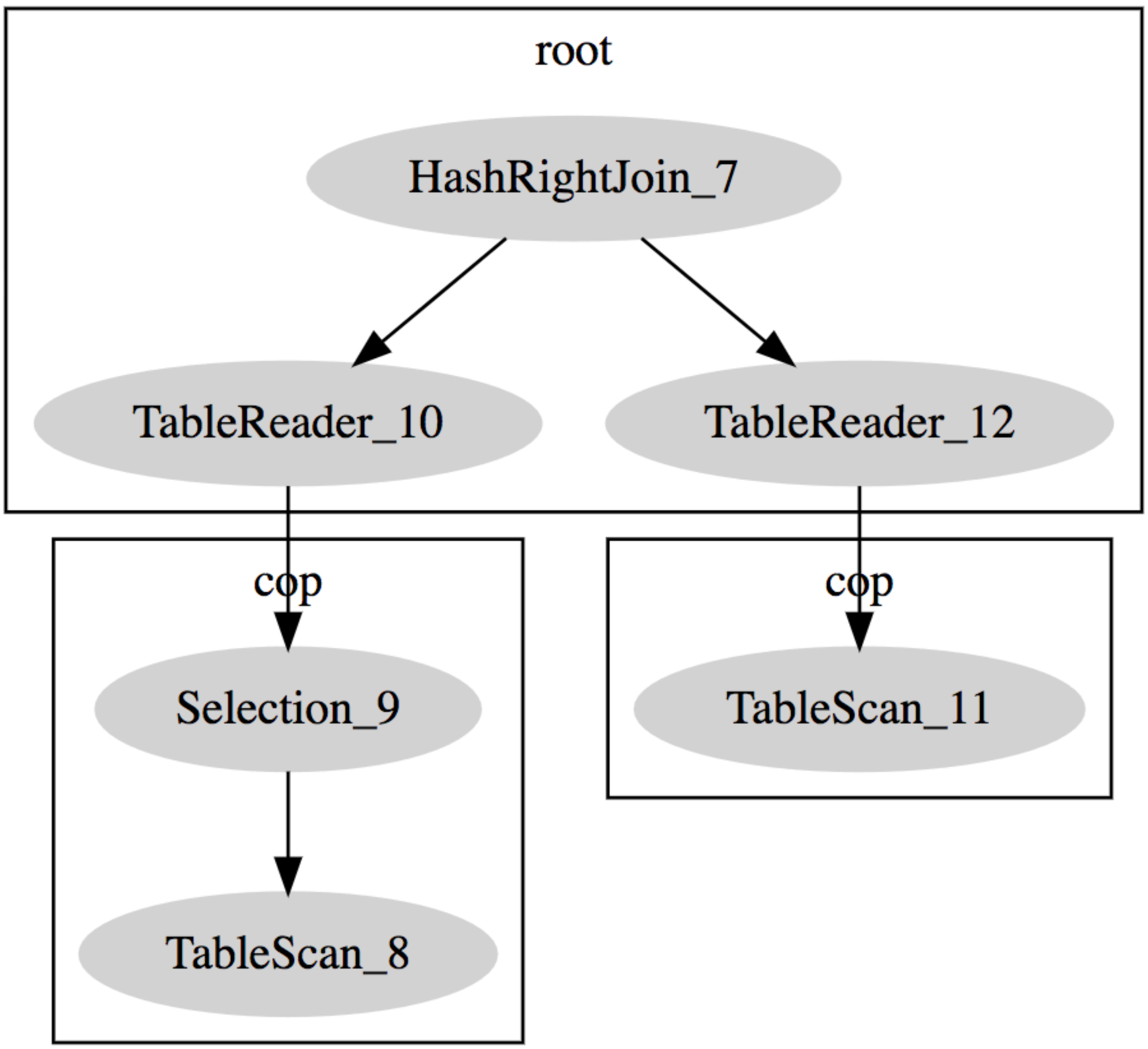


图 230: Explain Dot

12.11.2.45.4 MySQL 兼容性

- EXPLAIN 的格式和 TiDB 中潜在的执行计划都与 MySQL 有很大不同。
- TiDB 不像 MySQL 那样支持 EXPLAIN FORMAT = JSON。
- TiDB 目前不支持插入语句的 EXPLAIN。

12.11.2.45.5 EXPLAIN FOR CONNECTION

EXPLAIN FOR CONNECTION 用于获得一个连接中当前正在执行 SQL 的执行计划或者是最后执行 SQL 的执行计划，其输出格式与 EXPLAIN 完全一致。但 TiDB 中的实现与 MySQL 不同，除了输出格式之外，还有以下区别：

- 如果连接处于睡眠状态，MySQL 返回为空，而 TiDB 返回的是最后执行的查询计划。
- 如果获取当前会话连接的执行计划，MySQL 会报错，而 TiDB 会正常返回。
- MySQL 的文档中指出，MySQL 要求登录用户与被查询的连接相同，或者拥有 PROCESS 权限，而 TiDB 则要求登录用户与被查询的连接相同，或者拥有 SUPER 权限。

12.11.2.45.6 另请参阅

- 理解 TiDB 执行计划
- EXPLAIN ANALYZE
- ANALYZE TABLE
- TRACE

12.11.2.46 FLASHBACK TABLE

在 TiDB 4.0 中，引入了 FLASHBACK TABLE 语法，其功能是在 Garbage Collection (GC) life time 时间内，可以用 FLASHBACK TABLE 语句来恢复被 DROP 或 TRUNCATE 删除的表以及数据。

查询集群的 tikv_gc_safe_point 和 tikv_gc_life_time。只要被 DROP 或 TRUNCATE 删除的表是在 tikv_gc_safe_point 时间之后，都能用 FLASHBACK TABLE 语法来恢复。

```
sql  select * from mysql.tidb where variable_name in ('tikv_gc_safe_point','tikv_gc_life_time');
```

12.11.2.46.1 语法

```
FLASHBACK TABLE table_name [TO other_table_name]
```

语法图

```
FlashbackTableStmt ::=
    'FLASHBACK' 'TABLE' TableName FlashbackToNewName

TableName ::=
    Identifier ( '.' Identifier )?

FlashbackToNewName ::=
    ( 'TO' Identifier )?
```

12.11.2.46.2 注意事项

如果删除了一张表并过了 GC lifetime，就不能再用 FLASHBACK TABLE 语句来恢复被删除的数据了，否则会返回错误，错误类似于 `Can't find dropped/truncated table 't' in GC safe point 2020-03-16 16:34:52 +0800 CST`。

在开启 TiDB Binlog 时使用 FLASHBACK TABLE 需要注意以下情况：

- 下游从集群也支持 FLASHBACK TABLE
- 从集群的 GC lifetime 一定要长于主集群的 GC lifetime。上下游同步存在的延迟可能也会造成下游恢复数据失败。

如果 Binlog 同步出错，则需要在 Binlog 过滤掉该表，同时手动全量重新导入该表的数据。

12.11.2.46.3 示例

- 恢复被 DROP 删除的表数据：

```
DROP TABLE t;
```

```
FLASHBACK TABLE t;
```

- 恢复被 TRUNCATE 的表数据，由于被 TRUNCATE 的表还存在，所以需要重命名被恢复的表，否则会报错表 t 已存在。

```
TRUNCATE TABLE t;
```

```
FLASHBACK TABLE t TO t1;
```

12.11.2.46.4 工作原理

TiDB 在删除表时，实际上只删除了表的元信息，并将需要删除的表数据（行数据和索引数据）写一条数据到 `mysql.gc_delete_range` 表。TiDB 后台的 GC Worker 会定期从 `mysql.gc_delete_range` 表中取出超过 GC lifetime 相关范围的 key 进行删除。

所以，FLASHBACK TABLE 只需要在 GC Worker 还没删除表数据前，恢复表的元信息并删除 `mysql.gc_delete_range` 表中相应的行记录即可。恢复表的元信息可以用 TiDB 的快照读实现。具体的快照读内容可以参考读取历史数据文档。下面是 FLASHBACK TABLE t TO t1 的工作流程：

1. 从 DDL History job 中查找 `drop table` 或者 `truncate table` 类型的操作，且操作的表名是 t 的第一个 DDL job，若没找到，则返回错误。
2. 检查 DDL job 的开始时间，是否在 `tikv_gc_safe_point` 之前。如果是`tikv_gc_safe_point` 之前，说明被 DROP 或 TRUNCATE 删除的表已经被 GC 清理掉，返回错误。
3. 用 DDL job 的开始时间作为 snapshot 读取历史数据，读取表的元信息。
4. 删除 `mysql.gc_delete_range` 中和表 t 相关的 GC 任务。
5. 将表的元信息中的 name 修改成 `t1`，并用该元信息新建一个表。注意：这里只是修改了表名，但是 table ID 不变，依旧是之前被删除的表 t 的 table ID。

可以发现，从表 t 被删除，到表 t 被 FLASHBACK 恢复到 t1，一直都是对表的元信息进行操作，而表的用户数据一直未被修改过。被恢复的表 t1 和之前被删除的表 t 的 table ID 相同，所以表 t1 才能读取表t 的用户数据。

> **注意：**
>
> 不能用 FLASHBACK 多次恢复同一个被删除的表，因为 FLASHBACK 所恢复表的 table ID 还是被删除表的 table ID，而 TiDB 要求所有还存在的表 table ID 必须全局唯一。

FLASHBACK TABLE 是通过快照读获取表的元信息后，再走一次类似于 CREATE TABLE 的建表流程，所以 FLASHBACK TABLE 实际上也是一种 DDL 操作。

12.11.2.46.5 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

12.11.2.47 FLUSH PRIVILEGES

FLUSH PRIVILEGES 语句可触发 TiDB 从权限表中重新加载权限的内存副本。在对如 mysql.user 一类的表进行手动编辑后，应当执行 FLUSH PRIVILEGES。使用如 GRANT 或 REVOKE 一类的权限语句后，不需要执行 FLUSH PRIVILEGES 语句。执行 FLUSH PRIVILEGES 语句的用户需要拥有 RELOAD 权限。

12.11.2.47.1 语法图

```
FlushStmt ::=
    'FLUSH' NoWriteToBinLogAliasOpt FlushOption

NoWriteToBinLogAliasOpt ::=
    ( 'NO_WRITE_TO_BINLOG' | 'LOCAL' )?

FlushOption ::=
    'PRIVILEGES'
|   'STATUS'
|    'TIDB' 'PLUGINS' PluginNameList
|    'HOSTS'
|   LogTypeOpt 'LOGS'
|   TableOrTables TableNameListOpt WithReadLockOpt
```

12.11.2.47.2 示例

```
FLUSH PRIVILEGES;
```

```
Query OK, 0 rows affected (0.01 sec)
```

12.11.2.47.3 MySQL 兼容性

`FLUSH PRIVILEGES` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.47.4 另请参阅

- GRANT <privileges>
- REVOKE <privileges>
- Privilege Management

12.11.2.48 FLUSH STATUS

`FLUSH STATUS` 语句用于提供 MySQL 兼容性，但在 TiDB 上并无作用。因为 TiDB 使用 Prometheus 和 Grafana 而非 `SHOW STATUS` 来进行集中度量收集。

12.11.2.48.1 语法图

```
FlushStmt ::=
    'FLUSH' NoWriteToBinLogAliasOpt FlushOption

NoWriteToBinLogAliasOpt ::=
    ( 'NO_WRITE_TO_BINLOG' | 'LOCAL' )?

FlushOption ::=
    'PRIVILEGES'
|   'STATUS'
|    'TIDB' 'PLUGINS' PluginNameList
|    'HOSTS'
|   LogTypeOpt 'LOGS'
|   TableOrTables TableNameListOpt WithReadLockOpt
```

12.11.2.48.2 示例

```
show status;
```

```
+--------------------+--------------------------------------+
| Variable_name      | Value                                |
+--------------------+--------------------------------------+
| Ssl_cipher_list    |                                      |
| server_id          | 93e2e07d-6bb4-4a1b-90b7-e035fae154fe |
| ddl_schema_version | 141                                  |
| Ssl_verify_mode    | 0                                    |
| Ssl_version        |                                      |
| Ssl_cipher         |                                      |
+--------------------+--------------------------------------+
6 rows in set (0.01 sec)
```

```
show global status;
```

```
+--------------------+--------------------------------------+
| Variable_name      | Value                                |
+--------------------+--------------------------------------+
| Ssl_cipher         |                                      |
| Ssl_cipher_list    |                                      |
| Ssl_verify_mode    | 0                                    |
| Ssl_version        |                                      |
| server_id          | 93e2e07d-6bb4-4a1b-90b7-e035fae154fe |
| ddl_schema_version | 141                                  |
+--------------------+--------------------------------------+
6 rows in set (0.00 sec)
```

```
flush status;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
show status;
```

```
+--------------------+--------------------------------------+
| Variable_name      | Value                                |
+--------------------+--------------------------------------+
| Ssl_cipher         |                                      |
| Ssl_cipher_list    |                                      |
| Ssl_verify_mode    | 0                                    |
| Ssl_version        |                                      |
| ddl_schema_version | 141                                  |
| server_id          | 93e2e07d-6bb4-4a1b-90b7-e035fae154fe |
+--------------------+--------------------------------------+
6 rows in set (0.00 sec)
```

12.11.2.48.3 MySQL 兼容性

- FLUSH STATUS 语句仅用于提供 MySQL 兼容性。

12.11.2.48.4 另请参阅

- SHOW [GLOBAL | SESSION] STATUS

12.11.2.49 FLUSH TABLES

FLUSH TABLES 语句用于提供 MySQL 兼容性，但在 TiDB 中并无有效用途。

12.11.2.49.1 语法图

```
FlushStmt ::=
    'FLUSH' NoWriteToBinLogAliasOpt FlushOption

NoWriteToBinLogAliasOpt ::=
    ( 'NO_WRITE_TO_BINLOG' | 'LOCAL' )?

FlushOption ::=
    'PRIVILEGES'
|   'STATUS'
|    'TIDB' 'PLUGINS' PluginNameList
|    'HOSTS'
|    LogTypeOpt 'LOGS'
|    TableOrTables TableNameListOpt WithReadLockOpt

LogTypeOpt ::=
    ( 'BINARY' | 'ENGINE' | 'ERROR' | 'GENERAL' | 'SLOW' )?

TableOrTables ::=
    'TABLE'
|   'TABLES'

TableNameListOpt ::=
    TableNameList?

WithReadLockOpt ::=
    ( 'WITH' 'READ' 'LOCK' )?
```

12.11.2.49.2 示例

```
FLUSH TABLES;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
FLUSH TABLES WITH READ LOCK;
```

```
ERROR 1105 (HY000): FLUSH TABLES WITH READ LOCK is not supported.  Please use @@tidb_snapshot
```

12.11.2.49.3 MySQL 兼容性

- TiDB 没有 MySQL 中的表缓存这一概念。所以，`FLUSH TABLES` 因 MySQL 兼容性会在 TiDB 中解析出但会被忽略掉。
- 因为 TiDB 目前不支持锁表，所以`FLUSH TABLES WITH READ LOCK` 语句会产生错误。建议使用Historical reads来实现锁表。

12.11.2.49.4 另请参阅

- Read historical data

12.11.2.50 GRANT <privileges>

GRANT <privileges> 语句用于为 TiDB 中已存在的用户分配权限。TiDB 中的权限系统同 MySQL 一样，都基于数据库/表模式来分配凭据。执行 GRANT <privileges> 语句需要拥有分配的权限，并且拥有 GRANT OPTION 权限。

12.11.2.50.1 语法图

```
GrantStmt ::=
    'GRANT' PrivElemList 'ON' ObjectType PrivLevel 'TO' UserSpecList RequireClauseOpt
        ↪ WithGrantOptionOpt

PrivElemList ::=
    PrivElem ( ',' PrivElem )*

PrivElem ::=
    PrivType ( '(' ColumnNameList ')' )?

PrivType ::=
    'ALL' 'PRIVILEGES'?
|    'ALTER' 'ROUTINE'?
|   'CREATE' ( 'USER' | 'TEMPORARY' 'TABLES' | 'VIEW' | 'ROLE' | 'ROUTINE' )?
|   'TRIGGER'
|   'DELETE'
|   'DROP' 'ROLE'?
|    'PROCESS'
|   'EXECUTE'
|   'INDEX'
|   'INSERT'
|   'SELECT'
|   'SUPER'
|   'SHOW' ( 'DATABASES' | 'VIEW' )
|   'UPDATE'
|    'GRANT' 'OPTION'
|    'REFERENCES'
|    'REPLICATION' ( 'SLAVE' | 'CLIENT' )
|    'USAGE'
|   'RELOAD'
|   'FILE'
|   'CONFIG'
|   'LOCK' 'TABLES'
|   'EVENT'
|   'SHUTDOWN'
```

```
ObjectType ::=
    'TABLE'?

PrivLevel ::=
    '*' ( '.' '*' )?
|    Identifier ( '.' ( '*' | Identifier ) )?

UserSpecList ::=
    UserSpec ( ',' UserSpec )*
```

12.11.2.50.2 示例

```
CREATE USER 'newuser' IDENTIFIED BY 'mypassword';
```

```
Query OK, 1 row affected (0.02 sec)
```

```
GRANT ALL ON test.* TO 'newuser';
```

```
Query OK, 0 rows affected (0.03 sec)
```

```
SHOW GRANTS FOR 'newuser';
```

```
+-------------------------------------------------+
| Grants for newuser@%                            |
+-------------------------------------------------+
| GRANT USAGE ON *.* TO 'newuser'@'%'             |
| GRANT ALL PRIVILEGES ON test.* TO 'newuser'@'%' |
+-------------------------------------------------+
2 rows in set (0.00 sec)
```

12.11.2.50.3 MySQL 兼容性

- 与 MySQL 类似，USAGE 权限表示登录 TiDB 服务器的能力。
- 目前不支持列级权限。
- 与 MySQL 类似，不存在 NO_AUTO_CREATE_USER sql 模式时，GRANT 语句将在用户不存在时自动创建一个空密码的新用户。删除此 sql-mode（默认情况下已启用）会带来安全风险。

12.11.2.50.4 另请参阅

- GRANT <role>
- REVOKE <privileges>
- SHOW GRANTS
- 权限管理

12.11.2.51 GRANT <role>

GRANT <role> 语句用于将之前创建的角色授予给现有用户。用户可以通过 SET ROLE <rolename> 语句拥有角色权限，或者通过 SET ROLE ALL 语句拥有被授予的所有角色。

12.11.2.51.1 语法图

```
GrantRoleStmt ::=
    'GRANT' RolenameList 'TO' UsernameList

RolenameList ::=
    Rolename ( ',' Rolename )*

UsernameList ::=
    Username ( ',' Username )*
```

12.11.2.51.2 示例

创建新角色 analyticsteam 和新用户 jennifer：

```
$ mysql -uroot

CREATE ROLE analyticsteam;
Query OK, 0 rows affected (0.02 sec)
GRANT SELECT ON test.* TO analyticsteam;
Query OK, 0 rows affected (0.02 sec)
CREATE USER jennifer;
Query OK, 0 rows affected (0.01 sec)
GRANT analyticsteam TO jennifer;
Query OK, 0 rows affected (0.01 sec)
```

需要注意的是，默认情况下，用户 jennifer 需要执行 SET ROLE analyticsteam 语句才能使用与角色相关联的权限：

```
$ mysql -ujennifer

SHOW GRANTS;
+-------------------------------------------+
| Grants for User                           |
+-------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'      |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+-------------------------------------------+
2 rows in set (0.00 sec)

SHOW TABLES in test;
ERROR 1044 (42000): Access denied for user 'jennifer'@'%' to database 'test'
```

```
SET ROLE analyticsteam;
Query OK, 0 rows affected (0.00 sec)

SHOW GRANTS;
+-------------------------------------------+
| Grants for User                           |
+-------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'      |
| GRANT Select ON test.* TO 'jennifer'@'%'  |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+-------------------------------------------+
3 rows in set (0.00 sec)

SHOW TABLES IN test;
+----------------+
| Tables_in_test |
+----------------+
| t1             |
+----------------+
1 row in set (0.00 sec)
```

执行 `SET DEFAULT ROLE` 语句将用户 `jennifer` 与某一角色相关联，这样该用户无需执行 `SET ROLE` 语句就能拥有与角色相关联的权限。

```
$ mysql -uroot

SET DEFAULT ROLE analyticsteam TO jennifer;
Query OK, 0 rows affected (0.02 sec)
```

```
$ mysql -ujennifer

SHOW GRANTS;
+-------------------------------------------+
| Grants for User                           |
+-------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'      |
| GRANT Select ON test.* TO 'jennifer'@'%'  |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+-------------------------------------------+
3 rows in set (0.00 sec)

SHOW TABLES IN test;
+----------------+
| Tables_in_test |
+----------------+
| t1             |
```

```
+----------------+
1 row in set (0.00 sec)
```

12.11.2.51.3 MySQL 兼容性

GRANT <role> 语句与 MySQL 8.0 的角色功能完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.51.4 另请参阅

- GRANT <privileges>
- CREATE ROLE
- DROP ROLE
- REVOKE <role>
- SET ROLE
- SET DEFAULT ROLE
- 基于角色的访问控制

12.11.2.52 INSERT

使用 INSERT 语句在表中插入新行。

12.11.2.52.1 语法图

```
InsertIntoStmt ::=
    'INSERT' TableOptimizerHints PriorityOpt IgnoreOptional IntoOpt TableName
        ↪ PartitionNameListOpt InsertValues OnDuplicateKeyUpdate

TableOptimizerHints ::=
    hintComment?

PriorityOpt ::=
    ( 'LOW_PRIORITY' | 'HIGH_PRIORITY' | 'DELAYED' )?

IgnoreOptional ::=
    'IGNORE'?

IntoOpt  ::= 'INTO'?

TableName ::=
    Identifier ( '.' Identifier )?

PartitionNameListOpt ::=
    ( 'PARTITION' '(' Identifier ( ',' Identifier )* ')' )?

InsertValues ::=
```

```
    '(' ( ColumnNameListOpt ')' ( ValueSym ValuesList | SelectStmt | '(' SelectStmt ')' |
        ↪ UnionStmt ) | SelectStmt ')' )
|   ValueSym ValuesList
|   SelectStmt
|   UnionStmt
|   'SET' ColumnSetValue? ( ',' ColumnSetValue )*

OnDuplicateKeyUpdate ::=
    ( 'ON' 'DUPLICATE' 'KEY' 'UPDATE' AssignmentList )?
```

12.11.2.52.2 示例

```
CREATE TABLE t1 (a INT);
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
CREATE TABLE t2 LIKE t1;
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
INSERT INTO t1 VALUES (1);
```

```
Query OK, 1 row affected (0.02 sec)
```

```
INSERT INTO t1 (a) VALUES (1);
```

```
Query OK, 1 row affected (0.01 sec)
```

```
INSERT INTO t2 SELECT * FROM t1;
```

```
Query OK, 2 rows affected (0.01 sec)
Records: 2  Duplicates: 0  Warnings: 0
```

```
SELECT * FROM t1;
```

```
+------+
| a    |
+------+
|    1 |
|    1 |
+------+
2 rows in set (0.00 sec)
```

```
SELECT * FROM t2;
```

```
+------+
| a    |
+------+
|    1 |
|    1 |
+------+
2 rows in set (0.00 sec)
```

```
INSERT INTO t2 VALUES (2),(3),(4);
```

```
Query OK, 3 rows affected (0.02 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
SELECT * FROM t2;
```

```
+------+
| a    |
+------+
|    1 |
|    1 |
|    2 |
|    3 |
|    4 |
+------+
5 rows in set (0.00 sec)
```

12.11.2.52.3 MySQL 兼容性

INSERT 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.52.4 另请参阅

- DELETE
- SELECT
- UPDATE
- REPLACE

12.11.2.53 KILL TIDB

KILL TIDB 语句用于终止 TiDB 中的连接。

12.11.2.53.1 语法图

```
KillStmt ::= KillOrKillTiDB ( 'CONNECTION' | 'QUERY' )? NUM

KillOrKillTiDB ::= 'KILL' 'TIDB'?
```

12.11.2.53.2 示例

```
SHOW PROCESSLIST;
```

```
+------+------+-----------+------+---------+------+-------+------------------+
| Id   | User | Host      | db   | Command | Time | State | Info             |
+------+------+-----------+------+---------+------+-------+------------------+
|    1 | root | 127.0.0.1 | test | Query   |    0 | 2     | SHOW PROCESSLIST |
|    2 | root | 127.0.0.1 |      | Sleep   |    4 | 2     |                  |
+------+------+-----------+------+---------+------+-------+------------------+
2 rows in set (0.00 sec)
```

```
KILL TIDB 2;
```

```
Query OK, 0 rows affected (0.00 sec)
```

12.11.2.53.3 MySQL 兼容性

- 按照设计，`KILL TIDB` 语句默认与 MySQL 不兼容。负载均衡器后面通常放有多个 TiDB 服务器，这种默认不兼容有助于防止在错误的 TiDB 服务器上终止连接。
- `KILL TIDB` 语句是 TiDB 的扩展语法。如果正尝试终止的会话位于同一个 TiDB 服务器上，可在配置文件里设置`compatible-kill-query = true`。

12.11.2.53.4 另请参阅

- SHOW [FULL] PROCESSLIST

12.11.2.54 LOAD DATA

`LOAD DATA` 语句用于将数据批量加载到 TiDB 表中。

12.11.2.54.1 语法图

```
LoadDataStmt ::=
    'LOAD' 'DATA' LocalOpt 'INFILE' stringLit DuplicateOpt 'INTO' 'TABLE' TableName CharsetOpt
        ↪ Fields Lines IgnoreLines ColumnNameOrUserVarListOptWithBrackets LoadDataSetSpecOpt
```

12.11.2.54.2 参数说明

用户可以使用 LocalOpt 参数来指定导入的数据文件位于客户端或者服务端。目前 TiDB 只支持从客户端进行数据导入，因此在导入数据时 LocalOpt 应设置成 Local。

用户可以使用 Fields 和 Lines 参数来指定如何处理数据格式，使用 FIELDS TERMINATED BY 来指定每个数据的分隔符号，使用 FIELDS ENCLOSED BY 来指定消除数据的包围符号。如果用户希望以某个字符为结尾切分每行数据，可以使用 LINES TERMINATED BY 来指定行的终止符。

例如对于以下格式的数据：

```
"bob","20","street 1"\r\n
"alice","33","street 1"\r\n
```

如果想分别提取 bob、20、street 1，可以指定数据的分隔符号为 ','，数据的包围符号为 '\"'。可以写成：

```
FIELDS TERMINATED BY ',' ENCLOSED BY '\"' LINES TERMINATED BY '\r\n'
```

如果不指定处理数据的参数，将会按以下参数处理

```
FIELDS TERMINATED BY '\t' ENCLOSED BY ''
LINES TERMINATED BY '\n'
```

用户可以通过 IGNORE number LINES 参数来忽略文件开始的 number 行，例如可以使用 IGNORE 1 LINES 来忽略文件的首行。

另外，TiDB 目前对参数 DuplicateOpt、CharsetOpt、LoadDataSetSpecOpt 仅支持语法解析。

12.11.2.54.3 示例

```
CREATE TABLE trips (
    trip_id bigint NOT NULL PRIMARY KEY AUTO_INCREMENT,
    duration integer not null,
    start_date datetime,
    end_date datetime,
    start_station_number integer,
    start_station varchar(255),
    end_station_number integer,
    end_station varchar(255),
    bike_number varchar(255),
    member_type varchar(255)
    );
```

```
Query OK, 0 rows affected (0.14 sec)
```

通过 LOAD DATA 导入数据，指定数据的分隔符为逗号，忽略包围数据的引号，并且忽略文件的第一行数据。

如果此时遇到 ERROR 1148 (42000): the used command is not allowed with this TiDB version 报错信息。可以参考以下文档解决：

ERROR 1148 (42000): the used command is not allowed with this TiDB version 问题的处理方法

```
LOAD DATA LOCAL INFILE '/mnt/evo970/data-sets/bikeshare-data/2017Q4-capitalbikeshare-tripdata.csv
    ↪ ' INTO TABLE trips FIELDS TERMINATED BY ',' ENCLOSED BY '\"' LINES TERMINATED BY '\r\n'
    ↪ IGNORE 1 LINES (duration, start_date, end_date, start_station_number, start_station,
    ↪ end_station_number, end_station, bike_number, member_type);
```

```
Query OK, 815264 rows affected (39.63 sec)
Records: 815264  Deleted: 0  Skipped: 0  Warnings: 0
```

LOAD DATA 也支持使用十六进制 ASCII 字符表达式或二进制 ASCII 字符表达式作为 FIELDS ENCLOSED BY 和 FIELDS TERMINATED BY 的参数。示例如下：

```
LOAD DATA LOCAL INFILE '/mnt/evo970/data-sets/bikeshare-data/2017Q4-capitalbikeshare-tripdata.csv
    ↪ ' INTO TABLE trips FIELDS TERMINATED BY x'2c' ENCLOSED BY b'100010' LINES TERMINATED BY '
    ↪ \r\n' IGNORE 1 LINES (duration, start_date, end_date, start_station_number, start_station
    ↪ , end_station_number, end_station, bike_number, member_type);
```

以上示例中 x'2c' 是字符 , 的十六进制表示，b'100010' 是字符 " 的二进制表示。

12.11.2.54.4 MySQL 兼容性

- 默认情况下，TiDB 每 20,000 行会进行一次提交。这类似于 MySQL NDB Cluster，但并非 InnoDB 存储引擎的默认配置。

> 注意：
>
> - 这种拆分事务提交的方式是以打破事务的原子性和隔离性为代价的，使用该特性时，使用者需要保证没有其他对正在处理的表的任何操作，并且在出现报错时，需要及时人工介入，检查数据的一致性和完整性。因此，不建议对读写频繁的表使用 LOAD DATA 语句。
> - 无论以多少行为一个事务提交，LOAD DATA 都不会被显式事务中的ROLLBACK 语句回滚。
> - LOAD DATA 语句始终以乐观事务模式执行，不受 TiDB 事务模式设置的影响。

12.11.2.54.5 另请参阅

- INSERT
- 乐观事务模型
- 悲观事务模式

12.11.2.55 LOAD STATS

LOAD STATS 语句用于将统计信息加载到 TiDB 中。

12.11.2.55.1 语法图

```
LoadStatsStmt ::=
    'LOAD' 'STATS' stringLit
```

12.11.2.55.2 参数说明

用户直接指定统计信息文件路径，统计信息文件可通过访问 API `http://${tidb-server-ip}:${tidb-server-status-port}/stats/dump/${db_name}/${table_name}` 进行下载。

路径可以是相对路径，也可以是绝对路径，如果是相对路径，会从启动 `tidb-server` 的路径为起点寻找对应文件。

下面是一个绝对路径的例子：

```
LOAD STATS '/tmp/stats.json';
```

```
Query OK, 0 rows affected (0.00 sec)
```

12.11.2.55.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

12.11.2.55.4 另请参阅

- 统计信息

12.11.2.56 MODIFY COLUMN

`ALTER TABLE .. MODIFY COLUMN` 语句用于修改已有表上的列，包括列的数据类型和属性。若要同时重命名，可改用`CHANGE COLUMN` 语句。

12.11.2.56.1 语法图

```
AlterTableStmt ::=
    'ALTER' IgnoreOptional 'TABLE' TableName ( AlterTableSpecListOpt AlterTablePartitionOpt | '
        ↪ ANALYZE' 'PARTITION' PartitionNameList ( 'INDEX' IndexNameList )?
        ↪ AnalyzeOptionListOpt )

AlterTableSpec ::=
    TableOptionList
|   'SET' 'TIFLASH' 'REPLICA' LengthNum LocationLabelList
|   'CONVERT' 'TO' CharsetKw ( CharsetName | 'DEFAULT' ) OptCollate
|   'ADD' ( ColumnKeywordOpt IfNotExists ( ColumnDef ColumnPosition | '(' TableElementList ')' )
    ↪ | Constraint | 'PARTITION' IfNotExists NoWriteToBinLogAliasOpt (
    ↪ PartitionDefinitionListOpt | 'PARTITIONS' NUM ) )
```

```
|   ( ( 'CHECK' | 'TRUNCATE' ) 'PARTITION' | ( 'OPTIMIZE' | 'REPAIR' | 'REBUILD' ) 'PARTITION'
    ↪ NoWriteToBinLogAliasOpt ) AllOrPartitionNameList
|   'COALESCE' 'PARTITION' NoWriteToBinLogAliasOpt NUM
|   'DROP' ( ColumnKeywordOpt IfExists ColumnName RestrictOrCascadeOpt | 'PRIMARY' 'KEY' | '
    ↪ PARTITION' IfExists PartitionNameList | ( KeyOrIndex IfExists | 'CHECK' ) Identifier | '
    ↪ FOREIGN' 'KEY' IfExists Symbol )
|   'EXCHANGE' 'PARTITION' Identifier 'WITH' 'TABLE' TableName WithValidationOpt
|   ( 'IMPORT' | 'DISCARD' ) ( 'PARTITION' AllOrPartitionNameList )? 'TABLESPACE'
|   'REORGANIZE' 'PARTITION' NoWriteToBinLogAliasOpt ReorganizePartitionRuleOpt
|   'ORDER' 'BY' AlterOrderItem ( ',' AlterOrderItem )*
|   ( 'DISABLE' | 'ENABLE' ) 'KEYS'
|   ( 'MODIFY' ColumnKeywordOpt IfExists | 'CHANGE' ColumnKeywordOpt IfExists ColumnName )
    ↪ ColumnDef ColumnPosition
|   'ALTER' ( ColumnKeywordOpt ColumnName ( 'SET' 'DEFAULT' ( SignedLiteral | '(' Expression ')'
    ↪ ) | 'DROP' 'DEFAULT' ) | 'CHECK' Identifier EnforcedOrNot | 'INDEX' Identifier
    ↪ IndexInvisible )
|   'RENAME' ( ( 'COLUMN' | KeyOrIndex ) Identifier 'TO' Identifier | ( 'TO' | '='? | 'AS' )
    ↪ TableName )
|   LockClause
|   AlgorithmClause
|   'FORCE'
|   ( 'WITH' | 'WITHOUT' ) 'VALIDATION'
|   'SECONDARY_LOAD'
|   'SECONDARY_UNLOAD'

ColumnKeywordOpt ::= 'COLUMN'?

ColumnDef ::=
    ColumnName ( Type | 'SERIAL' ) ColumnOptionListOpt

ColumnPosition ::=
    ( 'FIRST' | 'AFTER' ColumnName )?
```

12.11.2.56.2 示例

```
CREATE TABLE t1 (id int not null primary key AUTO_INCREMENT, col1 INT);
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
INSERT INTO t1 (col1) VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 5 rows affected (0.02 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
ALTER TABLE t1 MODIFY col1 BIGINT;
```

```
Query OK, 0 rows affected (0.09 sec)
```

```
SHOW CREATE TABLE t1\G;
```

```
*************************** 1. row ***************************
       Table: t1
Create Table: CREATE TABLE `t1` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `col1` bigint(20) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin AUTO_INCREMENT=30001
1 row in set (0.00 sec)
```

12.11.2.56.3 MySQL 兼容性

- 不支持在单个 ALTER TABLE 语句修改多个列，例如：

```
ALTER TABLE t1 MODIFY col1 BIGINT, MODIFY id BIGINT NOT NULL;
ERROR 1105 (HY000): Unsupported multi schema change
```

- 不支持有损变更，以及部分数据类型的更改（包括整数改为字符串或 BLOB 格式等）。例如：

```
CREATE TABLE t1 (col1 BIGINT);
ALTER TABLE t1 MODIFY col1 INT;
ERROR 8200 (HY000): Unsupported modify column length 11 is less than origin 20
```

- 不支持修改 decimal 类型的精度。例如：

```
CREATE TABLE t (a DECIMAL(5, 3));
ALTER TABLE t MODIFY COLUMN a DECIMAL(6, 3);
ERROR 8200 (HY000): Unsupported modify column: can't change decimal column precision
```

12.11.2.56.4 另请参阅

- CREATE TABLE
- SHOW CREATE TABLE
- ADD COLUMN
- DROP COLUMN
- CHANGE COLUMN

12.11.2.57 PREPARE

PREPARE 语句为服务器端预处理语句提供 SQL 接口。

12.11.2.57.1 语法图

```
PreparedStmt ::=
    'PREPARE' Identifier 'FROM' PrepareSQL

PrepareSQL ::=
    stringLit
|   UserVariable
```

12.11.2.57.2 示例

```
PREPARE mystmt FROM 'SELECT ? as num FROM DUAL';
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
SET @number = 5;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
EXECUTE mystmt USING @number;
```

```
+------+
| num  |
+------+
| 5    |
+------+
1 row in set (0.00 sec)
```

```
DEALLOCATE PREPARE mystmt;
```

```
Query OK, 0 rows affected (0.00 sec)
```

12.11.2.57.3 MySQL 兼容性

PREPARE 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.57.4 另请参阅

- EXECUTE
- DEALLOCATE

12.11.2.58 RECOVER TABLE

RECOVER TABLE 的功能是恢复被删除的表及其数据。在 DROP TABLE 后，在 GC life time 时间内，可以用 RECOVER TABLE 语句恢复被删除的表以及其数据。

12.11.2.58.1 语法

```
RECOVER TABLE table_name
```

```
RECOVER TABLE BY JOB ddl_job_id
```

语法图

```
RecoverTableStmt ::=
    'RECOVER' 'TABLE' ( 'BY' 'JOB' Int64Num | TableName Int64Num? )

TableName ::=
    Identifier ( '.' Identifier )?

Int64Num ::= NUM

NUM ::= intLit
```

12.11.2.58.2 注意事项

如果删除表后并过了 GC lifetime，就不能再用 RECOVER TABLE 来恢复被删除的表了，执行 RECOVER TABLE 语句会返回类似错误：`snapshot is older than GC safe point 2019-07-10 13:45:57 +0800 CST`。

对于 3.0.0 及之后的 TiDB 版本，不推荐在使用 TiDB Binlog 的情况下使用 RECOVER TABLE 功能。

TiDB Binlog 在 3.0.1 支持 RECOVER TABLE 后，可在下面的情况下使用 RECOVER TABLE：

- 3.0.1+ 版本的 TiDB Binlog
- 主从集群都使用 TiDB 3.0
- 从集群 GC lifetime 一定要长于主集群（不过由于上下游同步的延迟，可能也会造成下游 recover 失败）

TiDB Binlog 同步错误处理

当使用 TiDB Binlog 同步工具时，上游 TiDB 使用 RECOVER TABLE 后，TiDB Binlog 可能会因为下面几个原因造成同步中断：

- 下游数据库不支持 RECOVER TABLE 语句。类似错误：`check the manual that corresponds to your MySQL server version for the right syntax to use near 'RECOVER TABLE table_name'`。
- 上下游数据库的 GC lifetime 不一样。类似错误：`snapshot is older than GC safe point 2019-07-10 13:45:57 +0800 CST`。
- 上下游数据库的同步延迟。类似错误：`snapshot is older than GC safe point 2019-07-10 13:45:57 +0800 CST`。

只能通过重新全量导入被删除的表来恢复 TiDB Binlog 的数据同步。

12.11.2.58.3 示例

- 根据表名恢复被删除的表。

```
DROP TABLE t;
```

```
RECOVER TABLE t;
```

根据表名恢复被删除的表需满足以下条件：

  - 最近 DDL JOB 历史中找到的第一个 DROP TABLE 操作，且
  - DROP TABLE 所删除的表的名称与 RECOVER TABLE 语句指定表名相同

- 根据删除表时的 DDL JOB ID 恢复被删除的表。

  如果第一次删除表 t 后，又新建了一个表 t，然后又把新建的表 t 删除了，此时如果想恢复最开始删除的表 t, 就需要用到指定 DDL JOB ID 的语法了。

```
DROP TABLE t;
```

```
ADMIN SHOW DDL JOBS 1;
```

上面这个语句用来查找删除表 t 时的 DDL JOB ID，这里是 53：

```
+--------+---------+------------+------------+--------------+-----------+----------+-----------+-----------
    ↪
| JOB_ID | DB_NAME | TABLE_NAME | JOB_TYPE   | SCHEMA_STATE | SCHEMA_ID | TABLE_ID |
    ↪ ROW_COUNT | START_TIME                         | STATE  |
+--------+---------+------------+------------+--------------+-----------+----------+-----------+-----------
    ↪
| 53     | test    |            | drop table | none         | 1         | 41       | 0
    ↪         | 2019-07-10 13:23:18.277 +0800 CST | synced |
+--------+---------+------------+------------+--------------+-----------+----------+-----------+-----------
    ↪
```

```
RECOVER TABLE BY JOB 53;
```

根据删除表时的 DDL JOB ID 恢复被删除的表，会直接用 DDL JOB ID 找到被删除表进行恢复。如果指定的 DDL JOB ID 的 DDL JOB 不是 DROP TABLE 类型，会报错。

12.11.2.58.4 原理

TiDB 在删除表时，实际上只删除了表的元信息，并将需要删除的表数据（行数据和索引数据）写一条数据到 mysql.gc_delete_range 表。TiDB 后台的 GC Worker 会定期从 mysql.gc_delete_range 表中取出超过 GC lifetime 相关范围的 key 进行删除。

所以，RECOVER TABLE 只需要在 GC Worker 还没删除表数据前，恢复表的元信息并删除 mysql.gc_delete_range 表中相应的行记录就可以了。恢复表的元信息可以用 TiDB 的快照读实现。具体的快照读内容可以参考读取历史数据文档。

TiDB 中表的恢复是通过快照读获取表的元信息后，再走一次类似于 CREATE TABLE 的建表流程，所以 RECOVER
↪ TABLE 实际上也是一种 DDL。

12.11.2.58.5 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

12.11.2.59 RENAME INDEX

`ALTER TABLE .. RENAME INDEX` 语句用于对已有索引进行重命名。这在 TiDB 中是即时操作的，仅需更改元数据。

12.11.2.59.1 语法图

```
AlterTableStmt ::=
    'ALTER' IgnoreOptional 'TABLE' TableName ( AlterTableSpecListOpt AlterTablePartitionOpt | '
        ↪ ANALYZE' 'PARTITION' PartitionNameList ( 'INDEX' IndexNameList )?
        ↪ AnalyzeOptionListOpt )

KeyOrIndex ::=
    'KEY'
|   'INDEX'
```

12.11.2.59.2 示例

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, c1 INT NOT NULL, INDEX col1 (c1));
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
SHOW CREATE TABLE t1;
```

```
*************************** 1. row ***************************
       Table: t1
Create Table: CREATE TABLE `t1` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `c1` int(11) NOT NULL,
  PRIMARY KEY (`id`),
  KEY `col1` (`c1`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin
1 row in set (0.00 sec)
```

```
ALTER TABLE t1 RENAME INDEX col1 TO c1;
```

```
Query OK, 0 rows affected (0.09 sec)
```

```
SHOW CREATE TABLE t1;
```

```
*************************** 1. row ***************************
       Table: t1
Create Table: CREATE TABLE `t1` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `c1` int(11) NOT NULL,
  PRIMARY KEY (`id`),
  KEY `c1` (`c1`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin
1 row in set (0.00 sec)
```

12.11.2.59.3 MySQL 兼容性

`RENAME INDEX` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.59.4 另请参阅

- SHOW CREATE TABLE
- CREATE INDEX
- DROP INDEX
- SHOW INDEX

12.11.2.60 RENAME TABLE

`RENAME TABLE` 语句用于对已有表进行重命名。

12.11.2.60.1 语法图

```
RenameTableStmt ::=
    'RENAME' 'TABLE' TableToTable ( ',' TableToTable )*

TableToTable ::=
    TableName 'TO' TableName
```

12.11.2.60.2 示例

```
CREATE TABLE t1 (a int);
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
SHOW TABLES;
```

```
+----------------+
| Tables_in_test |
+----------------+
| t1             |
+----------------+
1 row in set (0.00 sec)
```

```
RENAME TABLE t1 TO t2;
```

```
Query OK, 0 rows affected (0.08 sec)
```

```
SHOW TABLES;
```

```
+----------------+
| Tables_in_test |
+----------------+
| t2             |
+----------------+
1 row in set (0.00 sec)
```

12.11.2.60.3 MySQL 兼容性

RENAME TABLE 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.60.4 另请参阅

- CREATE TABLE
- SHOW TABLES
- ALTER TABLE

12.11.2.61 REPLACE

从语义上看，REPLACE 语句是 DELETE 语句和 INSERT 语句的结合，可用于简化应用程序代码。

12.11.2.61.1 语法图

```
ReplaceIntoStmt ::=
    'REPLACE' PriorityOpt IntoOpt TableName PartitionNameListOpt InsertValues

PriorityOpt ::=
    ( 'LOW_PRIORITY' | 'HIGH_PRIORITY' | 'DELAYED' )?

IntoOpt ::= 'INTO'?
```

```
TableName ::=
    Identifier ( '.' Identifier )?

PartitionNameListOpt ::=
    ( 'PARTITION' '(' Identifier ( ',' Identifier )* ')' )?

InsertValues ::=
    '(' ( ColumnNameListOpt ')' ( ValueSym ValuesList | SelectStmt | '(' SelectStmt ')' |
        ↪ UnionStmt ) | SelectStmt ')' )
|   ValueSym ValuesList
|   SelectStmt
|   UnionStmt
|   'SET' ColumnSetValue? ( ',' ColumnSetValue )*
```

12.11.2.61.2 示例

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, c1 INT NOT NULL);
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
INSERT INTO t1 (c1) VALUES (1), (2), (3);
```

```
Query OK, 3 rows affected (0.02 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
SELECT * FROM t1;
```

```
+----+----+
| id | c1 |
+----+----+
|  1 |  1 |
|  2 |  2 |
|  3 |  3 |
+----+----+
3 rows in set (0.00 sec)
```

```
REPLACE INTO t1 (id, c1) VALUES(3, 99);
```

```
Query OK, 2 rows affected (0.01 sec)
```

```
SELECT * FROM t1;
```

```
+----+----+
| id | c1 |
+----+----+
|  1 |  1 |
|  2 |  2 |
|  3 | 99 |
+----+----+
3 rows in set (0.00 sec)
```

12.11.2.61.3 MySQL 兼容性

REPLACE 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.61.4 另请参阅

- DELETE
- INSERT
- SELECT
- UPDATE

12.11.2.62 RESTORE

> **警告：**
>
> RESTORE 语句目前为实验特性，不建议在生产环境中使用。该功能可能会在未事先通知的情况下发生变化或删除。如果发现 bug，请在 GitHub 上提 issue 反馈。

RESTORE 语句用于执行分布式恢复，把BACKUP 语句生成的备份文件恢复到 TiDB 集群中。

RESTORE 语句使用的引擎与BR 相同，但恢复过程是由 TiDB 本身驱动，而非单独的 BR 工具。BR 工具的优势和警告也适用于 RESTORE 语句。需要注意的是，RESTORE 语句目前不遵循 ACID 原则。

执行 RESTORE 语句前，确保集群已满足以下要求：

- 集群处于“下线”状态，当前的 TiDB 会话是唯一在访问待恢复表的活跃 SQL 连接。
- 执行全量恢复时，确保即将恢复的表不存在于集群中，因为现有的数据可能被覆盖，从而导致数据与索引不一致。
- 执行增量恢复时，表的状态应该与创建备份时 LAST_BACKUP 时间戳的状态完全一致。

执行 RESTORE 需要 SUPER 权限。此外，执行恢复操作的 TiDB 节点和集群中的所有 TiKV 节点都必须有对目标存储的读权限。

RESTORE 语句开始执行后将会被阻塞，直到整个恢复任务完成、失败或取消。因此，执行 RESTORE 时需要准备一个持久的连接。如需取消任务，可执行KILL TIDB QUERY 语句。

一次只能执行一个 BACKUP 和 RESTORE 任务。如果 TiDB server 上已经在执行一个 BACKUP 或 RESTORE 语句，新的 RESTORE 将等待前面所有的任务完成后再执行。

RESTORE 只能在 “tikv” 存储引擎上使用，如果使用 “mocktikv” 存储引擎，RESTORE 操作会失败。

12.11.2.62.1 语法图

```
RestoreStmt ::=
    "RESTORE" BRIETables "FROM" stringLit RestoreOption*

BRIETables ::=
    "DATABASE" ( '*' | DBName (',' DBName)* )
|   "TABLE" TableNameList

RestoreOption ::=
    "RATE_LIMIT" '='? LengthNum "MB" '/' "SECOND"
|   "CONCURRENCY" '='? LengthNum
|   "CHECKSUM" '='? Boolean
|   "SEND_CREDENTIALS_TO_TIKV" '='? Boolean

Boolean ::=
    NUM | "TRUE" | "FALSE"
```

12.11.2.62.2 示例

从备份文件恢复

```
RESTORE DATABASE * FROM 'local:///mnt/backup/2020/04/';
```

```
+--
    ↪ ---------------------------+-----------+----------+---------------------+---------------------+
    ↪
| Destination                  | Size      | BackupTS | Queue Time          | Execution Time
    ↪      |
+--
    ↪ ---------------------------+-----------+----------+---------------------+---------------------+
    ↪
| local:///mnt/backup/2020/04/ | 248665063 | 0        | 2020-04-21 17:16:55 | 2020-04-21 17:16:55
    ↪  |
+--
    ↪ ---------------------------+-----------+----------+---------------------+---------------------+
    ↪
1 row in set (28.961 sec)
```

上述示例中，所有数据均从本地的备份文件中恢复到集群中。RESTORE 从 SST 文件里读取数据，SST 文件存储在所有 TiDB 和 TiKV 节点的 /mnt/backup/2020/04/ 目录下。

输出结果的第一行描述如下：

列名	描述
Destination	读取的目标存储 URL
Size	备份文件的总大小，单位为字节
BackupTS	不适用
Queue Time	RESTORE 任务开始排队的时间戳（当前时区）
Execution Time	RESTORE 任务开始执行的时间戳（当前时区）

部分恢复

你可以指定恢复部分数据库或部分表数据。如果备份文件中缺失了某些数据库或表，缺失的部分将被忽略。此时，RESTORE 语句不进行任何操作即完成执行。

```
RESTORE DATABASE `test` FROM 'local:///mnt/backup/2020/04/';
```

```
RESTORE TABLE `test`.`sbtest01`, `test`.`sbtest02` FROM 'local:///mnt/backup/2020/04/';
```

外部存储

BR 支持从 Amazon S3 或 Google Cloud Storage (GCS) 恢复数据：

```
RESTORE DATABASE * FROM 's3://example-bucket-2020/backup-05/?region=us-west-2';
```

有关详细的 URL 语法，见外部存储。

当运行在云环境中时，不能分发凭证，可设置 SEND_CREDENTIALS_TO_TIKV 选项为 FALSE：

```
RESTORE DATABASE * FROM 's3://example-bucket-2020/backup-05/?region=us-west-2'
    SEND_CREDENTIALS_TO_TIKV = FALSE;
```

性能调优

如果你需要减少网络带宽占用，可以通过 RATE_LIMIT 来限制每个 TiKV 节点的平均下载速度。

默认情况下，每个 TiKV 节点上运行 128 个恢复线程。可以通过 CONCURRENCY 选项来调整这个值。

在恢复完成之前，RESTORE 将对备份文件中的数据进行校验，以验证数据的正确性。如果你确信无需进行校验，可以通过 CHECKSUM 选项禁用这一步骤。

```
RESTORE DATABASE * FROM 's3://example-bucket-2020/backup-06/'
    RATE_LIMIT = 120 MB/SECOND
    CONCURRENCY = 64
    CHECKSUM = FALSE;
```

增量恢复

增量恢复没有特殊的语法。TiDB 将识别备份文件属于全量备份或增量备份，然后执行对应的恢复操作，用户只需按照正确顺序进行增量恢复。

假设按照如下方式创建一个备份任务：

```
BACKUP DATABASE `test` TO 's3://example-bucket/full-backup'  SNAPSHOT = 413612900352000;
BACKUP DATABASE `test` TO 's3://example-bucket/inc-backup-1' SNAPSHOT = 414971854848000
    ↪ LAST_BACKUP = 413612900352000;
BACKUP DATABASE `test` TO 's3://example-bucket/inc-backup-2' SNAPSHOT = 416353458585600
    ↪ LAST_BACKUP = 414971854848000;
```

在恢复备份时，需要采取同样的顺序：

```
RESTORE DATABASE * FROM 's3://example-bucket/full-backup';
RESTORE DATABASE * FROM 's3://example-bucket/inc-backup-1';
RESTORE DATABASE * FROM 's3://example-bucket/inc-backup-2';
```

12.11.2.62.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

12.11.2.62.4 另请参阅

- BACKUP
- SHOW RESTORES

12.11.2.63 REVOKE <privileges>

`REVOKE <privileges>` 语句用于删除已有用户的权限。执行 `REVOKE <privileges>` 语句需要拥有分配的权限，并且拥有 `GRANT OPTION` 权限。

12.11.2.63.1 语法图

```
GrantStmt ::=
    'GRANT' PrivElemList 'ON' ObjectType PrivLevel 'TO' UserSpecList RequireClauseOpt
        ↪ WithGrantOptionOpt

PrivElemList ::=
    PrivElem ( ',' PrivElem )*

PrivElem ::=
    PrivType ( '(' ColumnNameList ')' )?

PrivType ::=
    'ALL' 'PRIVILEGES'?
|   'ALTER' 'ROUTINE'?
```

```
|   'CREATE' ( 'USER' | 'TEMPORARY' 'TABLES' | 'VIEW' | 'ROLE' | 'ROUTINE' )?
|    'TRIGGER'
|   'DELETE'
|    'DROP' 'ROLE'?
|    'PROCESS'
|    'EXECUTE'
|   'INDEX'
|   'INSERT'
|   'SELECT'
|   'SUPER'
|    'SHOW' ( 'DATABASES' | 'VIEW' )
|   'UPDATE'
|   'GRANT' 'OPTION'
|   'REFERENCES'
|   'REPLICATION' ( 'SLAVE' | 'CLIENT' )
|   'USAGE'
|    'RELOAD'
|   'FILE'
|   'CONFIG'
|   'LOCK' 'TABLES'
|    'EVENT'
|   'SHUTDOWN'

ObjectType ::=
    'TABLE'?

PrivLevel ::=
    '*' ( '.' '*' )?
|    Identifier ( '.' ( '*' | Identifier ) )?

UserSpecList ::=
    UserSpec ( ',' UserSpec )*
```

12.11.2.63.2 示例

```
CREATE USER 'newuser' IDENTIFIED BY 'mypassword';
```

```
Query OK, 1 row affected (0.02 sec)
```

```
GRANT ALL ON test.* TO 'newuser';
```

```
Query OK, 0 rows affected (0.03 sec)
```

```
SHOW GRANTS FOR 'newuser';
```

```
+-----------------------------------------------+
| Grants for newuser@%                          |
+-----------------------------------------------+
| GRANT USAGE ON *.* TO 'newuser'@'%'           |
| GRANT ALL PRIVILEGES ON test.* TO 'newuser'@'%' |
+-----------------------------------------------+
2 rows in set (0.00 sec)
```

```
REVOKE ALL ON test.* FROM 'newuser';
```

```
Query OK, 0 rows affected (0.03 sec)
```

```
SHOW GRANTS FOR 'newuser';
```

```
+--------------------------------------+
| Grants for newuser@%                 |
+--------------------------------------+
| GRANT USAGE ON *.* TO 'newuser'@'%' |
+--------------------------------------+
1 row in set (0.00 sec)
```

```
DROP USER 'newuser';
```

```
Query OK, 0 rows affected (0.14 sec)
```

```
SHOW GRANTS FOR 'newuser';
```

```
ERROR 1141 (42000): There is no such grant defined for user 'newuser' on host '%'
```

12.11.2.63.3 MySQL 兼容性

`REVOKE <privileges>` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.63.4 另请参阅

- GRANT <privileges>
- SHOW GRANTS
- Privilege Management

12.11.2.64 REVOKE <role>

`REVOKE <role>` 语句用于从指定的用户（或用户列表）中收回之前授予的角色。

12.11.2.64.1 语法图

```
RevokeRoleStmt ::=
    'REVOKE' RolenameList 'FROM' UsernameList

RolenameList ::=
    Rolename ( ',' Rolename )*

UsernameList ::=
    Username ( ',' Username )*
```

12.11.2.64.2 示例

创建新角色 `analyticsteam` 和新用户 `jennifer`：

```
$ mysql -uroot

CREATE ROLE analyticsteam;
Query OK, 0 rows affected (0.02 sec)

GRANT SELECT ON test.* TO analyticsteam;
Query OK, 0 rows affected (0.02 sec)

CREATE USER jennifer;
Query OK, 0 rows affected (0.01 sec)

GRANT analyticsteam TO jennifer;
Query OK, 0 rows affected (0.01 sec)
```

需要注意的是，默认情况下，用户 `jennifer` 需要执行 `SET ROLE analyticsteam` 语句才能使用与角色相关联的权限：

```
$ mysql -ujennifer

SHOW GRANTS;
+-----------------------------------------------+
| Grants for User                               |
+-----------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'          |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+-----------------------------------------------+
2 rows in set (0.00 sec)

SHOW TABLES in test;
ERROR 1044 (42000): Access denied for user 'jennifer'@'%' to database 'test'
SET ROLE analyticsteam;
```

```
Query OK, 0 rows affected (0.00 sec)

SHOW GRANTS;
+-------------------------------------------+
| Grants for User                           |
+-------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'      |
| GRANT Select ON test.* TO 'jennifer'@'%'  |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+-------------------------------------------+
3 rows in set (0.00 sec)

SHOW TABLES IN test;
+----------------+
| Tables_in_test |
+----------------+
| t1             |
+----------------+
1 row in set (0.00 sec)
```

执行 `SET DEFAULT ROLE` 语句将用户 `jennifer` 与某一角色相关联，这样该用户无需执行 `SET ROLE` 语句就能拥有与角色相关联的权限。

```
$ mysql -uroot

SET DEFAULT ROLE analyticsteam TO jennifer;
Query OK, 0 rows affected (0.02 sec)
```

```
$ mysql -ujennifer

SHOW GRANTS;
+-------------------------------------------+
| Grants for User                           |
+-------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'      |
| GRANT Select ON test.* TO 'jennifer'@'%'  |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+-------------------------------------------+
3 rows in set (0.00 sec)

SHOW TABLES IN test;
+----------------+
| Tables_in_test |
+----------------+
| t1             |
+----------------+
```

```
1 row in set (0.00 sec)
```

收回角色 analyticsteam：

```
$ mysql -uroot

REVOKE analyticsteam FROM jennifer;
Query OK, 0 rows affected (0.01 sec)
```

```
$ mysql -ujennifer

SHOW GRANTS;
+--------------------------------------+
| Grants for User                      |
+--------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%' |
+--------------------------------------+
1 row in set (0.00 sec)
```

12.11.2.64.3 MySQL 兼容性

REVOKE <role> 语句与 MySQL 8.0 的角色功能完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.64.4 另请参阅

- CREATE ROLE
- DROP ROLE
- GRANT <role>
- SET ROLE
- SET DEFAULT ROLE
- 基于角色的访问控制

12.11.2.65 ROLLBACK

ROLLBACK 语句用于还原 TiDB 内当前事务中的所有更改，作用与 COMMIT 语句相反。

12.11.2.65.1 语法图

```
RollbackStmt ::=
    'ROLLBACK' CompletionTypeWithinTransaction?

CompletionTypeWithinTransaction ::=
    'AND' ( 'CHAIN' ( 'NO' 'RELEASE' )? | 'NO' 'CHAIN' ( 'NO'? 'RELEASE' )? )
|   'NO'? 'RELEASE'
```

12.11.2.65.2 示例

```
CREATE TABLE t1 (a INT NOT NULL PRIMARY KEY);
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
BEGIN;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
INSERT INTO t1 VALUES (1);
```

```
Query OK, 1 row affected (0.00 sec)
```

```
ROLLBACK;
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
SELECT * FROM t1;
```

```
Empty set (0.01 sec)
```

12.11.2.65.3 MySQL 兼容性

- TiDB 不支持 savepoint 或 `ROLLBACK TO SAVEPOINT` 语法。
- TiDB 解析但忽略 `ROLLBACK AND [NO] RELEASE` 语法。在 MySQL 中，使用该语法可在回滚事务后立即断开客户端会话。在 TiDB 中，建议使用客户端程序的 `mysql_close()` 来实现该功能。
- TiDB 解析但忽略 `ROLLBACK AND [NO] CHAIN` 语法。在 MySQL 中，使用该语法可在回滚当前事务时立即以相同的隔离级别开启新事务。在 TiDB 中，推荐直接开启新事务。

12.11.2.65.4 另请参阅

- COMMIT
- BEGIN
- START TRANSACTION

12.11.2.66 SELECT

SELECT 语句用于从 TiDB 读取数据。

12.11.2.66.1 语法图

SelectStmt:

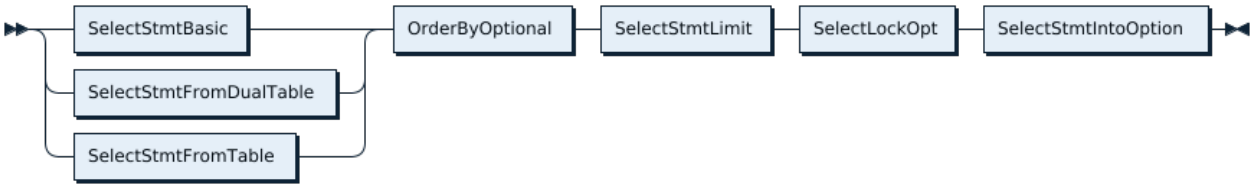


图 231: SelectStmt

FromDual:

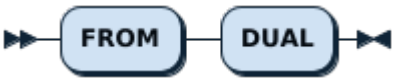


图 232: FromDual

WhereClauseOptional:

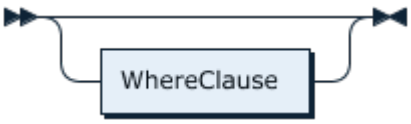


图 233: WhereClauseOptional

SelectStmtOpts:

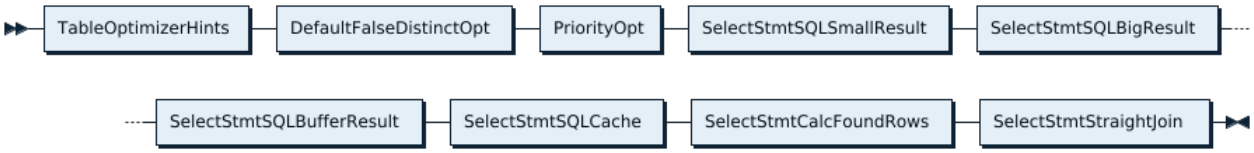


图 234: SelectStmtOpts

SelectStmtFieldList:

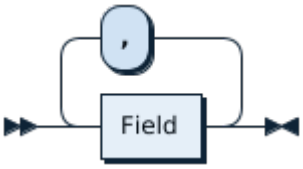


图 235: SelectStmtFieldList

TableRefsClause:

图 236: TableRefsClause

WhereClauseOptional:

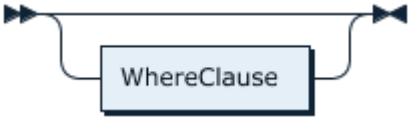


图 237: WhereClauseOptional

SelectStmtGroup:

图 238: SelectStmtGroup

HavingClause:

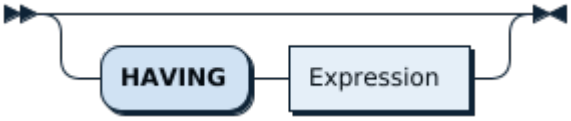


图 239: HavingClause

OrderByOptional:

图 240: OrderByOptional

SelectStmtLimit:

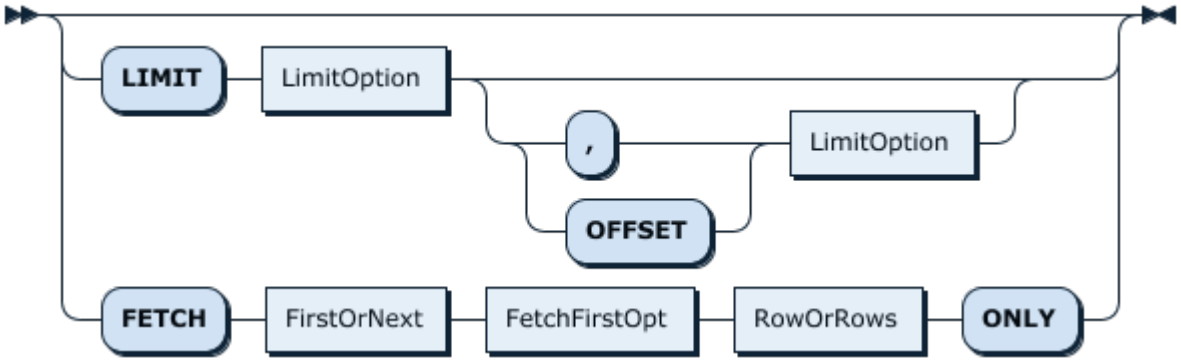


图 241: SelectStmtLimit

FirstOrNext:

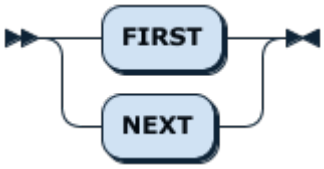


图 242: FirstOrNext

FetchFirstOpt:

图 243: FetchFirstOpt

RowOrRows:

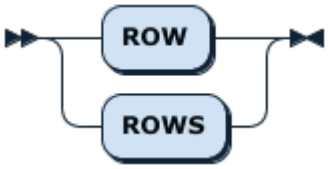


图 244: RowOrRows

SelectLockOpt:

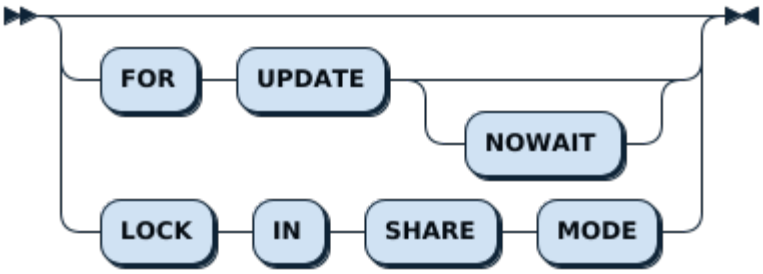


图 245: SelectLockOpt

WindowClauseOptional

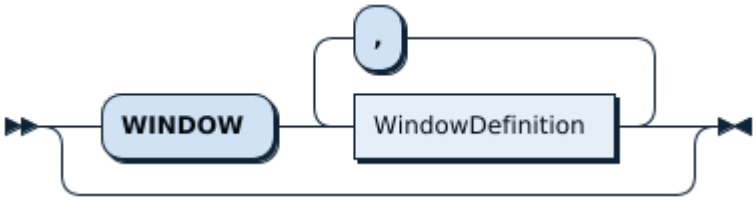


图 246: WindowClauseOptional

12.11.2.66.2 语法元素说明

语法元素	说明
TableOptimizerHints	用于控制优化器行为的 Hint，具体可参见Optimizer Hints
ALL、DISTINCT、DISTINCTROW	查询结果集中可能会包含重复值。指定 DISTINCT/DISTINCTROW 则在查询结果中过滤掉重复的行；指定 ALL 则列出所有的行。默认为 ALL。
HIGH_PRIORITY	该语句为高优先级语句，TiDB 在执行阶段会优先处理这条语句
SQL_CALC_FOUND_ROWS	TiDB 出于兼容性解析这个语法，但是不做任何处理
SQL_CACHE、 SQL_NO_CACHE	是否把请求结果缓存到 TiKV (RocksDB) 的 BlockCache 中。对于一次性的大数据量的查询，比如 count(*) 查询，为了避免冲掉 BlockCache 中用户的热点数据，建议填上 SQL_NO_CACHE
STRAIGHT_JOIN	STRAIGHT_JOIN 会强制优化器按照 FROM 子句中所使用的表的顺序做联合查询。当优化器选择的 Join 顺序并不优秀时，你可以使用这个语法来加速查询的执行
select_expr	投影操作列表，一般包括列名、表达式，或者是用 ‘*’ 表示全部列
FROM table_references	表示数据来源，数据来源可以是一个表 (select * from t;) 或者是多个表 (select * from t1 join t2;) 或者是 0 个表 (select 1+1 from dual;, 等价于 select 1+1;)
WHERE where_condition	Where 子句用于设置过滤条件，查询结果中只会包含满足条件的数据
GROUP BY	GroupBy 子句用于对查询结果集进行分组
HAVING where_condition	Having 子句与 Where 子句作用类似，Having 子句可以让过滤 GroupBy 后的各种数据，Where 子句用于在聚合前过滤记录。

语法元素	说明
ORDER BY	OrderBy 子句用于指定结果排序顺序，可以按照列、表达式或者是 select_expr 列表中某个位置的字段进行排序。
LIMIT	Limit 子句用于限制结果条数。Limit 接受一个或两个数字参数，如果只有一个参数，那么表示返回数据的最大行数；如果是两个参数，那么第一个参数表示返回数据的第一行的偏移量（第一行数据的偏移量是 0），第二个参数指定返回数据的最大条目数。另支持 FETCH FIRST/NEXT n ROW/ROWS ONLY 语法，与 LIMIT n 效果相同，其中 n 可省略，省略时与 LIMIT 1 效果相同。
Window window_definition	窗口函数的相关语法，用来进行一些分析型计算的操作，详情可见窗口函数
FOR UPDATE	对查询结果集所有行上锁（对于在查询条件内，但是不在结果集的行，将不会加锁，如事务启动后由其他事务写入的行），以监测其他事务对这些的并发修改。使用乐观事务模型时，语句执行期间不会检测锁，因此，不会像 PostgreSQL 之类的数据库一样，在当前事务结束前阻止其他事务执行 UPDATE、DELETE 和 SELECT FOR UPDATE。在事务的提交阶段 SELECT FOR UPDATE 读到的行，也会进行两阶段提交，因此，它们也可以参与事务冲突检测。如发生写入冲突，那么包含 SELECT FOR UPDATE 语句的事务会提交失败。如果没有冲突，事务将成功提交，当提交结束时，这些被加锁的行，会产生一个新版本，可以让其他尚未提交的事务，在将来提交时发现写入冲突。若使用悲观事务，则行为与其他数据库基本相同，不一致之处参考和 MySQL InnoDB 的差异。
LOCK IN SHARE MODE	TiDB 出于兼容性解析这个语法，但是不做任何处理

12.11.2.66.3 示例

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, c1 INT NOT NULL);
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
INSERT INTO t1 (c1) VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 5 rows affected (0.03 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
SELECT * FROM t1;
```

```
+----+----+
| id | c1 |
+----+----+
|  1 |  1 |
|  2 |  2 |
|  3 |  3 |
```

```
|  4 |  4 |
|  5 |  5 |
+----+----+
5 rows in set (0.00 sec)
```

12.11.2.66.4 MySQL 兼容性

- 不支持 `SELECT ... INTO @variable` 语法。
- 不支持 `SELECT ... GROUP BY ... WITH ROLLUP` 语法。
- 不支持 MySQL 5.7 中支持的 `SELECT .. GROUP BY expr` 语法，而是匹配 MySQL 8.0 的行为，不按照默认的顺序进行排序。

12.11.2.66.5 另请参阅

- INSERT
- DELETE
- UPDATE
- REPLACE

12.11.2.67 SET DEFAULT ROLE

`SET DEFAULT ROLE` 语句默认设置将特定角色应用于用户。因此，用户不必执行 `SET ROLE <rolename>` 或 `SET ROLE ALL` 语句，也可以自动具有与角色相关联的权限。

12.11.2.67.1 语法图

SetDefaultRoleStmt:

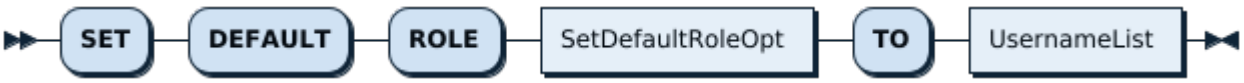


图 247: SetDefaultRoleStmt

SetDefaultRoleOpt:

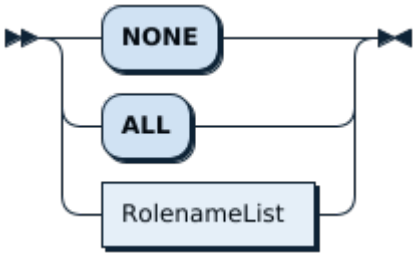


图 248: SetDefaultRoleOpt

RolenameList:

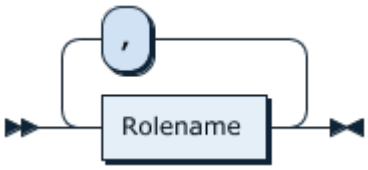


图 249: RolenameList

UsernameList:

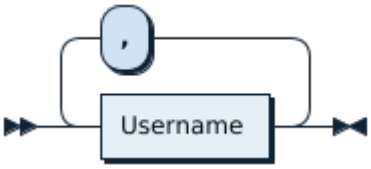


图 250: UsernameList

12.11.2.67.2 示例

创建新角色 `analyticsteam` 和新用户 `jennifer`：

```
$ mysql -uroot

CREATE ROLE analyticsteam;
Query OK, 0 rows affected (0.02 sec)
GRANT SELECT ON test.* TO analyticsteam;
Query OK, 0 rows affected (0.02 sec)
CREATE USER jennifer;
Query OK, 0 rows affected (0.01 sec)
GRANT analyticsteam TO jennifer;
Query OK, 0 rows affected (0.01 sec)
```

需要注意的是，默认情况下，用户 `jennifer` 需要执行 `SET ROLE analyticsteam` 语句才能使用与角色相关联的权限：

```
$ mysql -ujennifer

SHOW GRANTS;
+-------------------------------------------+
| Grants for User                           |
+-------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'      |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+-------------------------------------------+
2 rows in set (0.00 sec)
```

```
SHOW TABLES in test;
ERROR 1044 (42000): Access denied for user 'jennifer'@'%' to database 'test'
SET ROLE analyticsteam;
Query OK, 0 rows affected (0.00 sec)

SHOW GRANTS;
+-------------------------------------------+
| Grants for User                           |
+-------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'      |
| GRANT Select ON test.* TO 'jennifer'@'%'  |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+-------------------------------------------+
3 rows in set (0.00 sec)

SHOW TABLES IN test;
+----------------+
| Tables_in_test |
+----------------+
| t1             |
+----------------+
1 row in set (0.00 sec)
```

执行 `SET DEFAULT ROLE` 语句将用户 `jennifer` 与某一角色相关联，这样该用户无需执行 `SET ROLE` 语句就能拥有与角色相关联的权限。

```
$ mysql -uroot

SET DEFAULT ROLE analyticsteam TO jennifer;
Query OK, 0 rows affected (0.02 sec)
```

```
$ mysql -ujennifer

SHOW GRANTS;
+-------------------------------------------+
| Grants for User                           |
+-------------------------------------------+
| GRANT USAGE ON *.* TO 'jennifer'@'%'      |
| GRANT Select ON test.* TO 'jennifer'@'%'  |
| GRANT 'analyticsteam'@'%' TO 'jennifer'@'%' |
+-------------------------------------------+
3 rows in set (0.00 sec)

SHOW TABLES IN test;
+----------------+
| Tables_in_test |
```

```
+----------------+
| t1             |
+----------------+
1 row in set (0.00 sec)
```

SET DEFAULT ROLE 语句不会自动将相关角色授予 (GRANT) 用户。若尝试为 jennifer 尚未被授予的角色执行 SET DEFAULT ROLE 语句会导致以下错误：

```
SET DEFAULT ROLE analyticsteam TO jennifer;
ERROR 3530 (HY000): `analyticsteam`@`%` is is not granted to jennifer@%
```

12.11.2.67.3 MySQL 兼容性

SET DEFAULT ROLE 语句与 MySQL 8.0 的角色功能完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.67.4 另请参阅

- CREATE ROLE
- DROP ROLE
- GRANT <role>
- REVOKE <role>
- SET ROLE
- 基于角色的访问控制

12.11.2.68 SET [NAMES|CHARACTER SET]

SET NAMES，SET CHARACTER SET 和 SET CHARSET 语句用于修改当前连接的变量 character_set_client，character_set_results 和 character_set_connection。

12.11.2.68.1 语法图

SetNamesStmt:

图 251: SetNamesStmt

VariableAssignmentList:

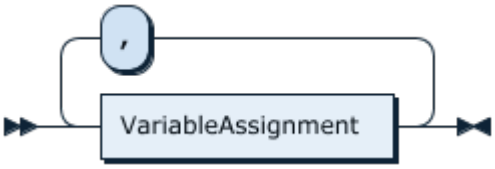


图 252: VariableAssignmentList

VariableAssignment:

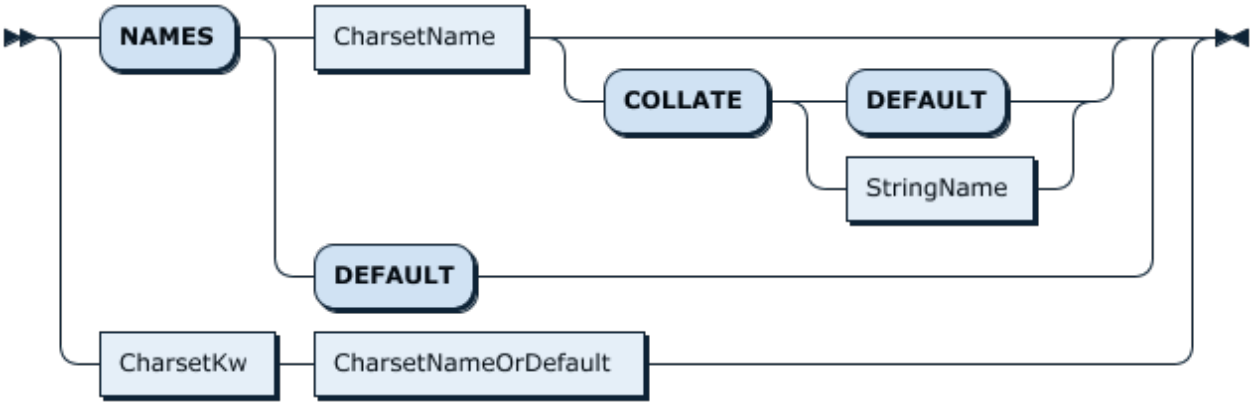


图 253: VariableAssignment

CharsetName:

图 254: CharsetName

StringName:

图 255: StringName

CharsetKw:

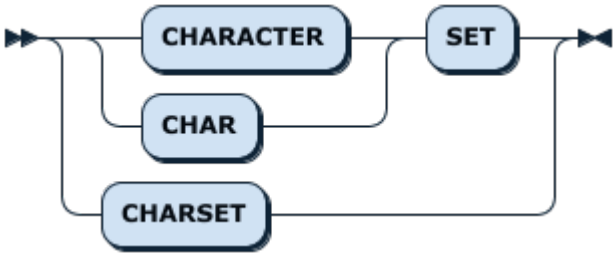


图 256: CharsetKw

CharsetNameOrDefault:

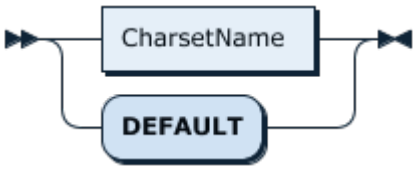


图 257: CharsetNameOrDefault

12.11.2.68.2 示例

```
SHOW VARIABLES LIKE 'character_set%';
```

```
+--------------------------+----------------------------------------------------------+
| Variable_name            | Value                                                    |
+--------------------------+----------------------------------------------------------+
| character_sets_dir       | /usr/local/mysql-5.6.25-osx10.8-x86_64/share/charsets/ |
| character_set_connection | utf8mb4                                                  |
| character_set_system     | utf8                                                     |
| character_set_results    | utf8mb4                                                  |
| character_set_client     | utf8mb4                                                  |
| character_set_database   | utf8mb4                                                  |
| character_set_filesystem | binary                                                   |
| character_set_server     | utf8mb4                                                  |
+--------------------------+----------------------------------------------------------+
8 rows in set (0.01 sec)
```

```
SET NAMES utf8;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
SHOW VARIABLES LIKE 'character_set%';
```

```
+--------------------------+----------------------------------------------------------+
| Variable_name            | Value                                                    |
+--------------------------+----------------------------------------------------------+
| character_sets_dir       | /usr/local/mysql-5.6.25-osx10.8-x86_64/share/charsets/ |
| character_set_connection | utf8                                                     |
| character_set_system     | utf8                                                     |
| character_set_results    | utf8                                                     |
| character_set_client     | utf8                                                     |
| character_set_server     | utf8mb4                                                  |
| character_set_database   | utf8mb4                                                  |
| character_set_filesystem | binary                                                   |
+--------------------------+----------------------------------------------------------+
8 rows in set (0.00 sec)
```

```
SET CHARACTER SET utf8mb4;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
SHOW VARIABLES LIKE 'character_set%';
```

```
+--------------------------+------------------------------------------------------------+
| Variable_name            | Value                                                      |
+--------------------------+------------------------------------------------------------+
| character_set_connection | utf8mb4                                                    |
| character_set_system     | utf8                                                       |
| character_set_results    | utf8mb4                                                    |
| character_set_client     | utf8mb4                                                    |
| character_sets_dir       | /usr/local/mysql-5.6.25-osx10.8-x86_64/share/charsets/ |
| character_set_database   | utf8mb4                                                    |
| character_set_filesystem | binary                                                     |
| character_set_server     | utf8mb4                                                    |
+--------------------------+------------------------------------------------------------+
8 rows in set (0.00 sec)
```

12.11.2.68.3 MySQL 兼容性

`SET [NAMES|CHARACTER SET]` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.68.4 另请参阅

- SHOW [GLOBAL|SESSION] VARIABLES
- SET <variable>
- Character Set Support

12.11.2.69 SET PASSWORD

`SET PASSWORD` 语句用于更改 TiDB 系统数据库中的用户密码。

12.11.2.69.1 语法图

SetStmt:

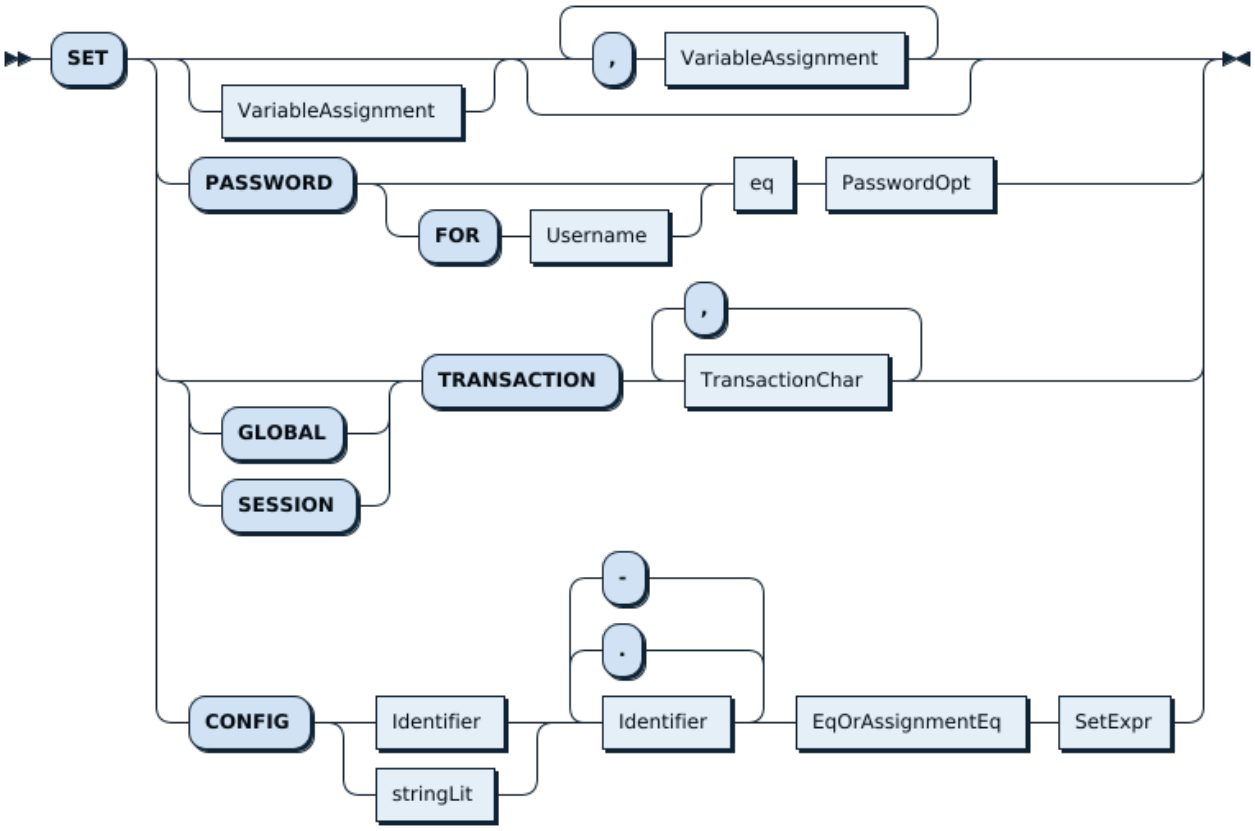


图 258: SetStmt

12.11.2.69.2 示例

```
SET PASSWORD='test';
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
CREATE USER 'newuser' IDENTIFIED BY 'test';
```

```
Query OK, 1 row affected (0.00 sec)
```

```
SHOW CREATE USER 'newuser';
```

```
+-----------------------------------------------------------------------------------------------------------------
    ↪
| CREATE USER for newuser@%
    ↪
    ↪ |
+-----------------------------------------------------------------------------------------------------------------
    ↪
```

```
| CREATE USER 'newuser'@'%' IDENTIFIED WITH 'mysql_native_password' AS '*94
    ↪ BDCEBE19083CE2A1F959FD02F964C7AF4CFC29' REQUIRE NONE PASSWORD EXPIRE DEFAULT ACCOUNT
    ↪ UNLOCK |
+-----------------------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

```
SET PASSWORD FOR 'newuser' = 'test';
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
SHOW CREATE USER 'newuser';
```

```
+-----------------------------------------------------------------------------------------------------------------------
    ↪
| CREATE USER for newuser@%
    ↪
    ↪ |
+-----------------------------------------------------------------------------------------------------------------------
    ↪
| CREATE USER 'newuser'@'%' IDENTIFIED WITH 'mysql_native_password' AS '*94
    ↪ BDCEBE19083CE2A1F959FD02F964C7AF4CFC29' REQUIRE NONE PASSWORD EXPIRE DEFAULT ACCOUNT
    ↪ UNLOCK |
+-----------------------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

```
SET PASSWORD FOR 'newuser' = PASSWORD('test');
```

上述语法是早期 MySQL 版本的过时语法。

```
Query OK, 0 rows affected (0.00 sec)
```

```
SHOW CREATE USER 'newuser';
```

```
+-----------------------------------------------------------------------------------------------------------------------
    ↪
| CREATE USER for newuser@%
    ↪
    ↪ |
+-----------------------------------------------------------------------------------------------------------------------
    ↪
| CREATE USER 'newuser'@'%' IDENTIFIED WITH 'mysql_native_password' AS '*94
    ↪ BDCEBE19083CE2A1F959FD02F964C7AF4CFC29' REQUIRE NONE PASSWORD EXPIRE DEFAULT ACCOUNT
    ↪ UNLOCK |
```

```
+-------------------------------------------------------------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

12.11.2.69.3 MySQL 兼容性

`SET PASSWORD` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.69.4 另请参阅

- CREATE USER
- Privilege Management

12.11.2.70 SET ROLE

`SET ROLE` 用于在当前用户会话中启用角色。使用 `SET ROLE` 启用角色后，用户可以使用这些角色的权限。

12.11.2.70.1 语法图

SetRoleStmt:

图 259: SetRoleStmt

SetRoleOpt:

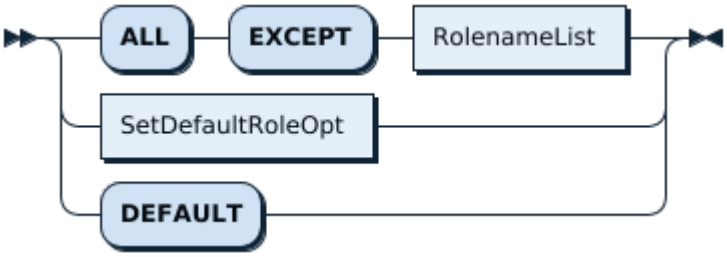


图 260: SetRoleOpt

SetDefaultRoleOpt:

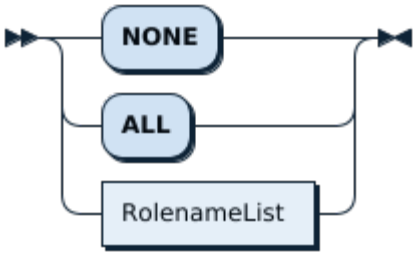


图 261: SetDefaultRoleOpt

12.11.2.70.2 示例

创建一个用户 `'u1'@'%'`, 创建三个角色 `'r1'@'%'`, `'r2'@'%'`, `'r3'@'%'` 并将这些角色授予给 `'u1'@'%'`。将 `'u1'@'%'` 的默认启用角色设置为 `'r1'@'%'`。

```
CREATE USER 'u1'@'%';
CREATE ROLE 'r1', 'r2', 'r3';
GRANT 'r1', 'r2', 'r3' TO 'u1'@'%';
SET DEFAULT ROLE 'r1' TO 'u1'@'%';
```

使用 `'u1'@'%'` 登录，执行 `SET ROLE` 将启用角色设置为 `ALL`。

```
SET ROLE ALL;
SELECT CURRENT_ROLE();
```

```
+----------------------------+
| CURRENT_ROLE()             |
+----------------------------+
| `r1`@`%`,`r2`@`%`,`r3`@`%` |
+----------------------------+
1 row in set (0.000 sec)
```

执行 `SET ROLE` 将启用角色设置为 `'r2'` 和 `'r3'`。

```
SET ROLE 'r2', 'r3';
SELECT CURRENT_ROLE();
```

```
+-------------------+
| CURRENT_ROLE()    |
+-------------------+
| `r2`@`%`,`r3`@`%` |
+-------------------+
1 row in set (0.000 sec)
```

执行 `SET ROLE` 将启用角色设置为 `DEFAULT`。

```
SET ROLE DEFAULT;
SELECT CURRENT_ROLE();
```

```
+----------------+
| CURRENT_ROLE() |
+----------------+
| `r1`@`%`       |
+----------------+
1 row in set (0.000 sec)
```

执行 SET ROLE 将启用角色设置为 NONE。

```
SET ROLE NONE;
SELECT CURRENT_ROLE();
```

```
+----------------+
| CURRENT_ROLE() |
+----------------+
|                |
+----------------+
1 row in set (0.000 sec)
```

12.11.2.70.3 MySQL 兼容性

SET ROLE 语句与 MySQL 8.0 的角色功能完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.70.4 另请参阅

- CREATE ROLE
- DROP ROLE
- GRANT <role>
- REVOKE <role>
- SET DEFAULT ROLE
- 基于角色的访问控制

12.11.2.71 SET TRANSACTION

SET TRANSACTION 语句用于在 GLOBAL 或 SESSION 的基础上更改当前的隔离级别，是 SET transaction_isolation ↪ ='new-value' 的替代语句，提供 MySQL 和 SQL 标准的兼容性。

12.11.2.71.1 语法图

SetStmt:

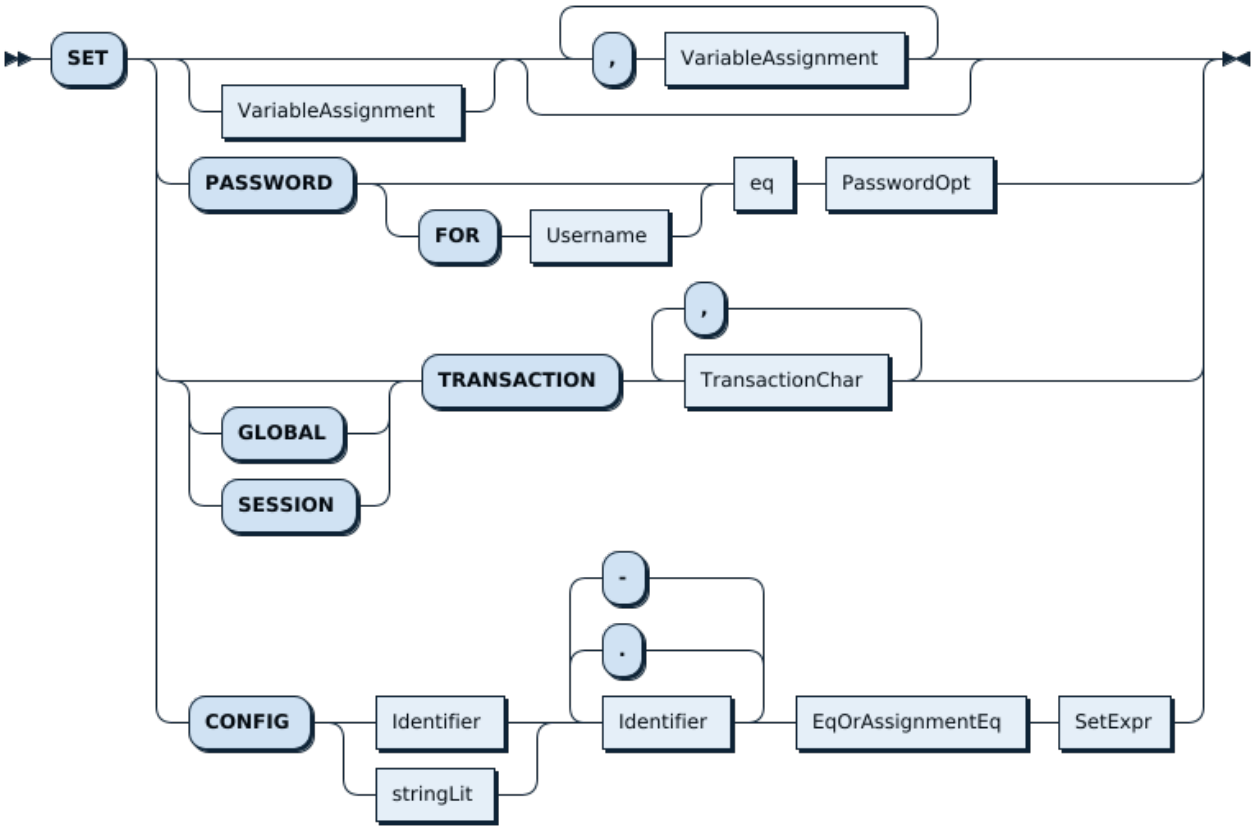


图 262: SetStmt

TransactionChar:

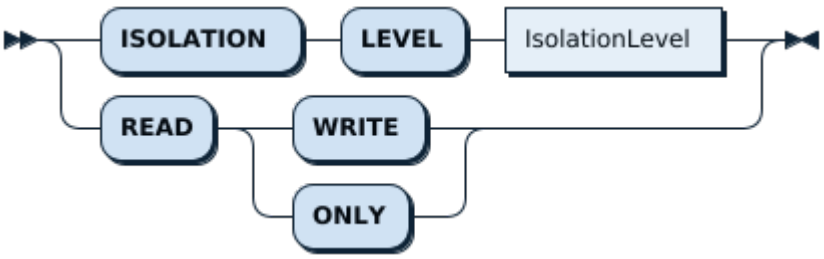


图 263: TransactionChar

IsolationLevel:

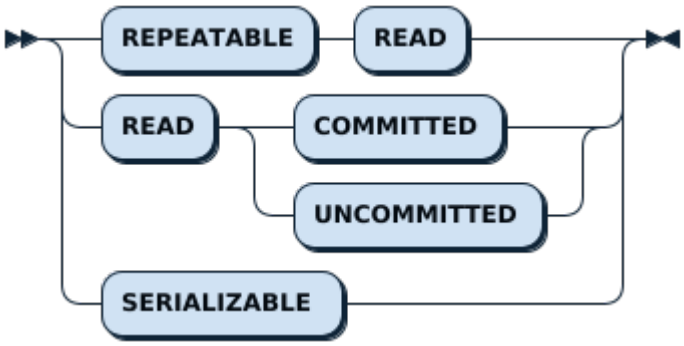


图 264: IsolationLevel

12.11.2.71.2 示例

```
SHOW SESSION VARIABLES LIKE 'transaction_isolation';
```

```
+-----------------------+-----------------+
| Variable_name         | Value           |
+-----------------------+-----------------+
| transaction_isolation | REPEATABLE-READ |
+-----------------------+-----------------+
1 row in set (0.00 sec)
```

```
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
SHOW SESSION VARIABLES LIKE 'transaction_isolation';
```

```
+-----------------------+----------------+
| Variable_name         | Value          |
+-----------------------+----------------+
| transaction_isolation | READ-COMMITTED |
+-----------------------+----------------+
1 row in set (0.01 sec)
```

```
SET SESSION transaction_isolation = 'REPEATABLE-READ';
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
SHOW SESSION VARIABLES LIKE 'transaction_isolation';
```

```
+-----------------------+-----------------+
| Variable_name         | Value           |
+-----------------------+-----------------+
```

```
| transaction_isolation | REPEATABLE-READ |
+-----------------------+-----------------+
1 row in set (0.00 sec)
```

12.11.2.71.3 MySQL 兼容性

- TiDB 支持仅在语法中将事务设置为只读的功能。
- 不支持隔离级别 READ-UNCOMMITTED 和 SERIALIZABLE。
- 通过快照隔离 (Snapshot Isolation) 技术，实现乐观事务的 REPEATABLE-READ 隔离级别，和 MySQL 兼容。
- 在悲观事务中，TiDB 支持与 MySQL 兼容的 REPEATABLE-READ 和 READ-COMMITTED 两种隔离级别。具体描述详见Isolation Levels。

12.11.2.71.4 另请参阅

- SET [GLOBAL|SESSION] <variable>
- Isolation Levels

12.11.2.72 SET [GLOBAL|SESSION] <variable>

SET [GLOBAL|SESSION] 语句用于在 SESSION 或 GLOBAL 的范围内，对某个 TiDB 的内置变量进行更改。

> **注意：**
>
> 与 MySQL 类似，对 GLOBAL 变量的更改不适用于已有连接或本地连接，只有新会话才会反映值的变化。

12.11.2.72.1 语法图

SetStmt:

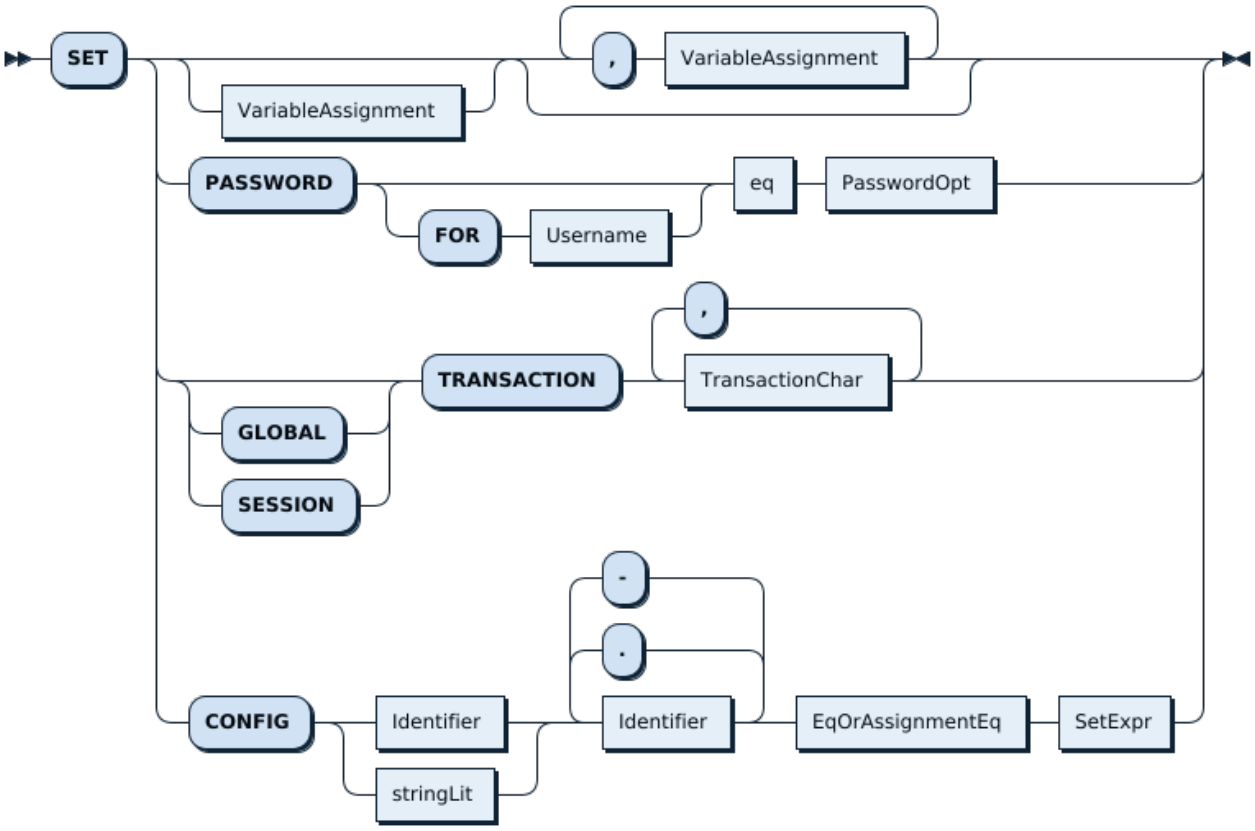


图 265: SetStmt

VariableAssignment:

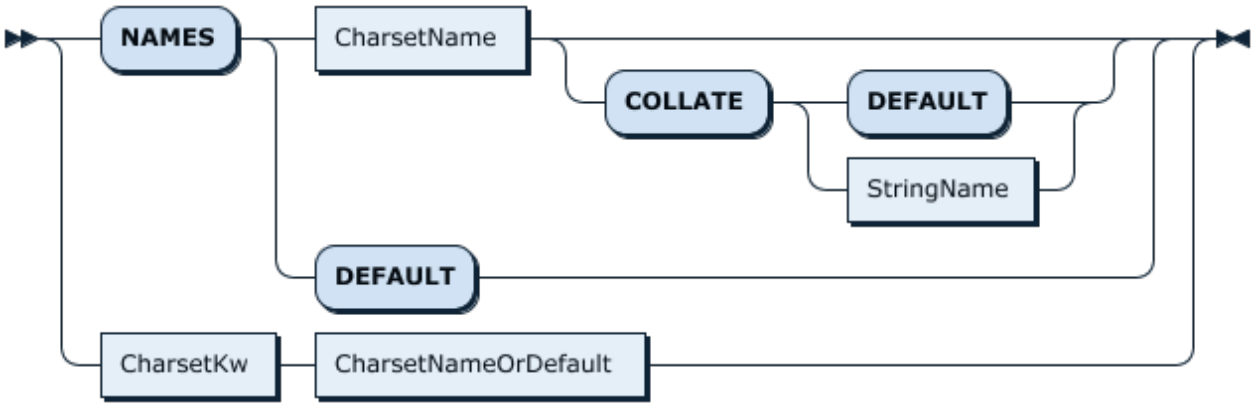


图 266: VariableAssignment

12.11.2.72.2 示例

获取 `sql_mode` 的值：

```
SHOW GLOBAL VARIABLES LIKE 'sql_mode';
```

```
+---------------+-----------------------------------------------------------------------------------------------------------
    ↪
| Variable_name | Value
    ↪
    ↪ |
+---------------+-----------------------------------------------------------------------------------------------------------
    ↪
| sql_mode      | ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,
    ↪ ERROR_FOR_DIVISION_BY_ZERO,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION |
+---------------+-----------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

```
SHOW SESSION VARIABLES LIKE 'sql_mode';
```

```
+---------------+-----------------------------------------------------------------------------------------------------------
    ↪
| Variable_name | Value
    ↪
    ↪ |
+---------------+-----------------------------------------------------------------------------------------------------------
    ↪
| sql_mode      | ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,
    ↪ ERROR_FOR_DIVISION_BY_ZERO,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION |
+---------------+-----------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

更新全局的 `sql_mode`：

```
SET GLOBAL sql_mode = 'STRICT_TRANS_TABLES,NO_AUTO_CREATE_USER';
```

```
Query OK, 0 rows affected (0.03 sec)
```

检查更新之后的 `sql_mode` 的取值，可以看到 `SESSION` 级别的值没有更新：

```
SHOW GLOBAL VARIABLES LIKE 'sql_mode';
```

```
+---------------+-----------------------------------------+
| Variable_name | Value                                   |
+---------------+-----------------------------------------+
| sql_mode      | STRICT_TRANS_TABLES,NO_AUTO_CREATE_USER |
+---------------+-----------------------------------------+
1 row in set (0.00 sec)
```

```
SHOW SESSION VARIABLES LIKE 'sql_mode';
```

```
+---------------+-------------------------------------------------------------------------------------------------
    ↪
| Variable_name | Value
    ↪
    ↪ |
+---------------+-------------------------------------------------------------------------------------------------
    ↪
| sql_mode      | ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,
    ↪ ERROR_FOR_DIVISION_BY_ZERO,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION |
+---------------+-------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

SET SESSION 则可以立即生效：

```
SET SESSION sql_mode = 'STRICT_TRANS_TABLES,NO_AUTO_CREATE_USER';
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
SHOW SESSION VARIABLES LIKE 'sql_mode';
```

```
+---------------+----------------------------------------+
| Variable_name | Value                                  |
+---------------+----------------------------------------+
| sql_mode      | STRICT_TRANS_TABLES,NO_AUTO_CREATE_USER |
+---------------+----------------------------------------+
1 row in set (0.00 sec)
```

12.11.2.72.3 MySQL 兼容性

使用 SET [GLOBAL|SESSION] <variable> 更改系统变量上，TiDB 与 MySQL 存在以下差异

- 与 MySQL 不同，TiDB 中使用 SET GLOBAL 所作的修改会应用于集群中的全部 TiDB 实例。而在 MySQL 中，修改不会应用于副本。
- TiDB 中的若干变量可读又可设置，这是与 MySQL 相兼容的要求，因为应用程序和连接器常读取 MySQL 变量。例如：JDBC 连接器同时读取和设置缓存查询的参数，尽管并不依赖这一行为。
- 即使在 TiDB 服务器重启后，SET GLOBAL 的更改也仍然有效。这样，TiDB 中的 SET GLOBAL 更类似于 MySQL 8.0 及更高版本中的 SET PERSIST。

12.11.2.72.4 另请参阅

- SHOW [GLOBAL|SESSION] VARIABLES

12.11.2.73 SHOW [BACKUPS|RESTORES]

SHOW [BACKUPS|RESTORES] 语句会列出所有队列中或正在执行的BACKUP 和RESTORE 任务。

查询 BACKUP 任务时，使用 SHOW BACKUPS 语句。查询 RESTORE 任务时，使用 SHOW RESTORES 语句。执行两个语句均需要 SUPER 权限。

12.11.2.73.1 语法图

```
ShowBRIEStmt ::=
    "SHOW" ("BACKUPS" | "RESTORES") ShowLikeOrWhere?

ShowLikeOrWhere ::=
    "LIKE" SimpleExpr
|   "WHERE" Expression
```

12.11.2.73.2 示例

在一个连接中，执行以下命令备份数据库：

```
BACKUP DATABASE `test` TO 's3://example-bucket/backup-01/?region=us-west-1';
```

在备份完成之前，在新的连接中执行 SHOW BACKUPS：

```
SHOW BACKUPS;
```

```
+--
    ↪ ------------------------------+---------+----------+--------------------+--------------------+---------
    ↪
| Destination                    | State    | Progress | Queue_Time          | Execution_Time
    ↪       | Finish_Time | Connection |
+--
    ↪ ------------------------------+---------+----------+--------------------+--------------------+---------
    ↪
| s3://example-bucket/backup-01/ | Backup   | 98.38    | 2020-04-12 23:09:03 | 2020-04-12 23:09:25
    ↪  |         NULL |           4 |
+--
    ↪ ------------------------------+---------+----------+--------------------+--------------------+---------
    ↪
1 row in set (0.00 sec)
```

输出结果的第一行描述如下：

列名	描述
Destination	目标存储的 URL（为避免泄露密钥，所有参数均不显示）
State	任务状态
Progress	当前状态的进度（百分比）

列名	描述
Queue Time	任务开始排队的时间
Execution Time	任务开始执行的时间；对于队列中任务，该值为 0000-00-00 00:00:00
Finish_Time	(暂不适用)
Connection	运行任务的连接 ID

连接 ID 可用于在KILL TIDB QUERY 语句中取消备份/恢复任务：

```
KILL TIDB QUERY 4;
```

```
Query OK, 0 rows affected (0.00 sec)
```

过滤

在 LIKE 子句中使用通配符，可以按目标存储 URL 筛选任务：

```
SHOW BACKUPS LIKE 's3://%';
```

使用 WHERE 子句，可以按列筛选任务：

```
SHOW BACKUPS WHERE `Progress` < 25.0;
```

12.11.2.73.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

12.11.2.73.4 另请参阅

- BACKUP
- RESTORE

12.11.2.74 SHOW ANALYZE STATUS

SHOW ANALYZE STATUS 语句提供 TiDB 正在执行的统计信息收集任务以及有限条历史任务记录。

12.11.2.74.1 语法图

ShowStmt:

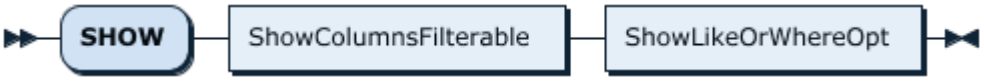


图 267: ShowStmt

ShowTargetFilterable:

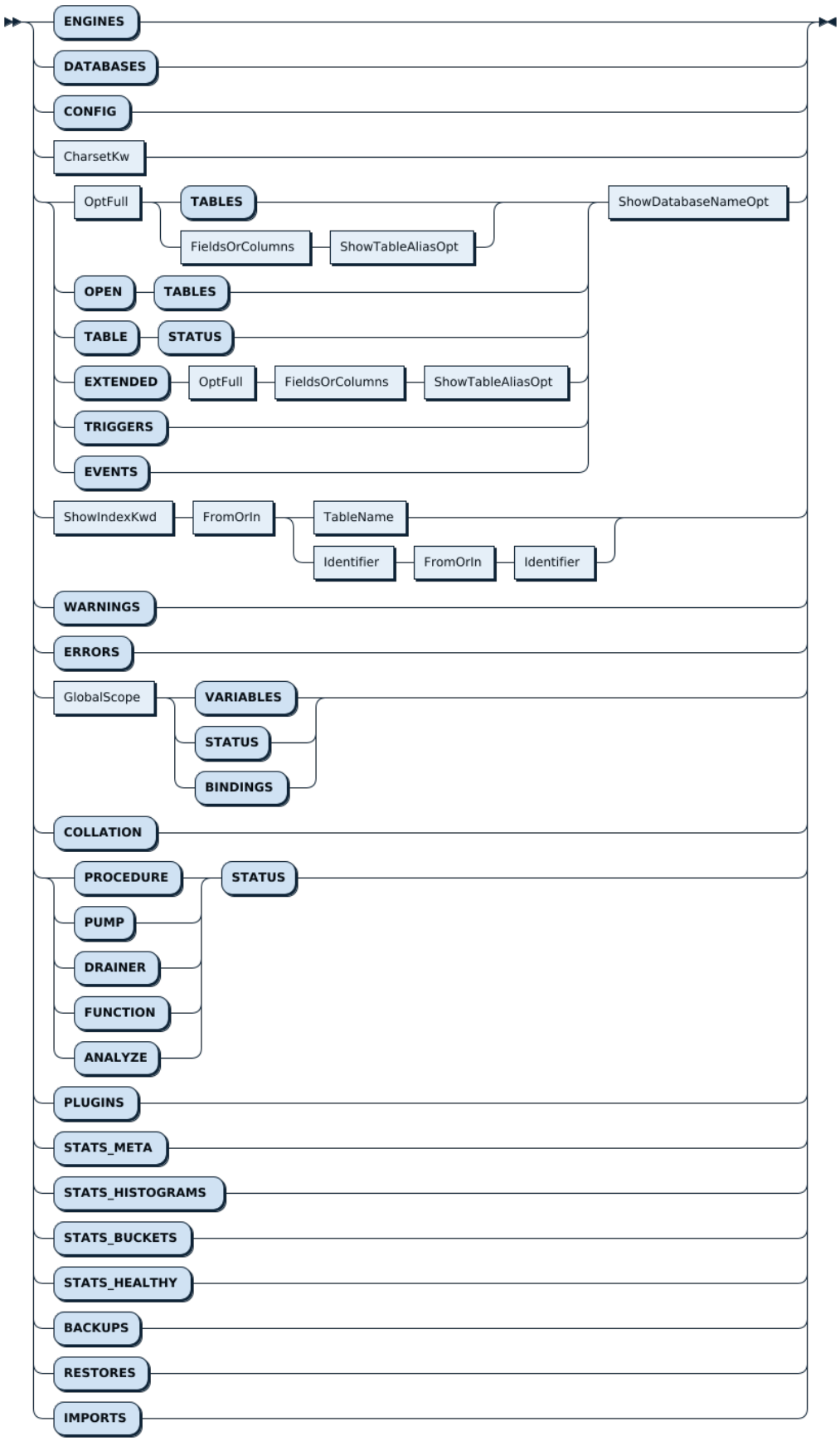


图 268: ShowTargetFilterable

12.11.2.74.2 示例

```
create table t(x int, index idx(x)) partition by hash(x) partition 4;
analyze table t;
show analyze status;
```

```
+--
    ↪ ------------+------------+---------------+------------------+---------------+---------------------+---
    ↪
| Table_schema | Table_name | Partition_name | Job_info          | Processed_rows | Start_time
    ↪          | State    |
+--
    ↪ ------------+------------+---------------+------------------+---------------+---------------------+---
    ↪
| test         | t          | p1             | analyze columns   |              0 | 2020-05-25
    ↪ 17:23:55 | finished |
| test         | t          | p0             | analyze columns   |              0 | 2020-05-25
    ↪ 17:23:55 | finished |
| test         | t          | p0             | analyze index idx |              0 | 2020-05-25
    ↪ 17:23:55 | finished |
| test         | t          | p1             | analyze index idx |              0 | 2020-05-25
    ↪ 17:23:55 | finished |
| test         | t          | p2             | analyze index idx |              0 | 2020-05-25
    ↪ 17:23:55 | finished |
| test         | t          | p3             | analyze index idx |              0 | 2020-05-25
    ↪ 17:23:55 | finished |
| test         | t          | p3             | analyze columns   |              0 | 2020-05-25
    ↪ 17:23:55 | finished |
| test         | t          | p2             | analyze columns   |              0 | 2020-05-25
    ↪ 17:23:55 | finished |
+--
    ↪ ------------+------------+---------------+------------------+---------------+---------------------+---
    ↪
8 rows in set (0.00 sec)
```

12.11.2.74.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

12.11.2.74.4 另请参阅

- ANALYZE_STATUS 表

12.11.2.75 SHOW [GLOBAL|SESSION] BINDINGS

`SHOW BINDINGS` 语句用于显示创建过的 `SQL` 绑定的相关信息。`BINDING` 语句可以在 `GLOBAL` 或者 `SESSION` 作用域内显示执行计划绑定。在不指定作用域时，默认的作用域为 `SESSION`。

12.11.2.75.1 语法图

ShowStmt:

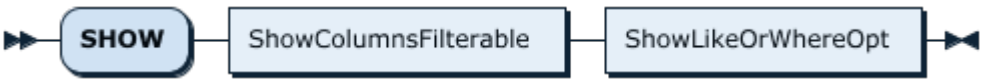


图 269: ShowStmt

ShowTargetFilterable:

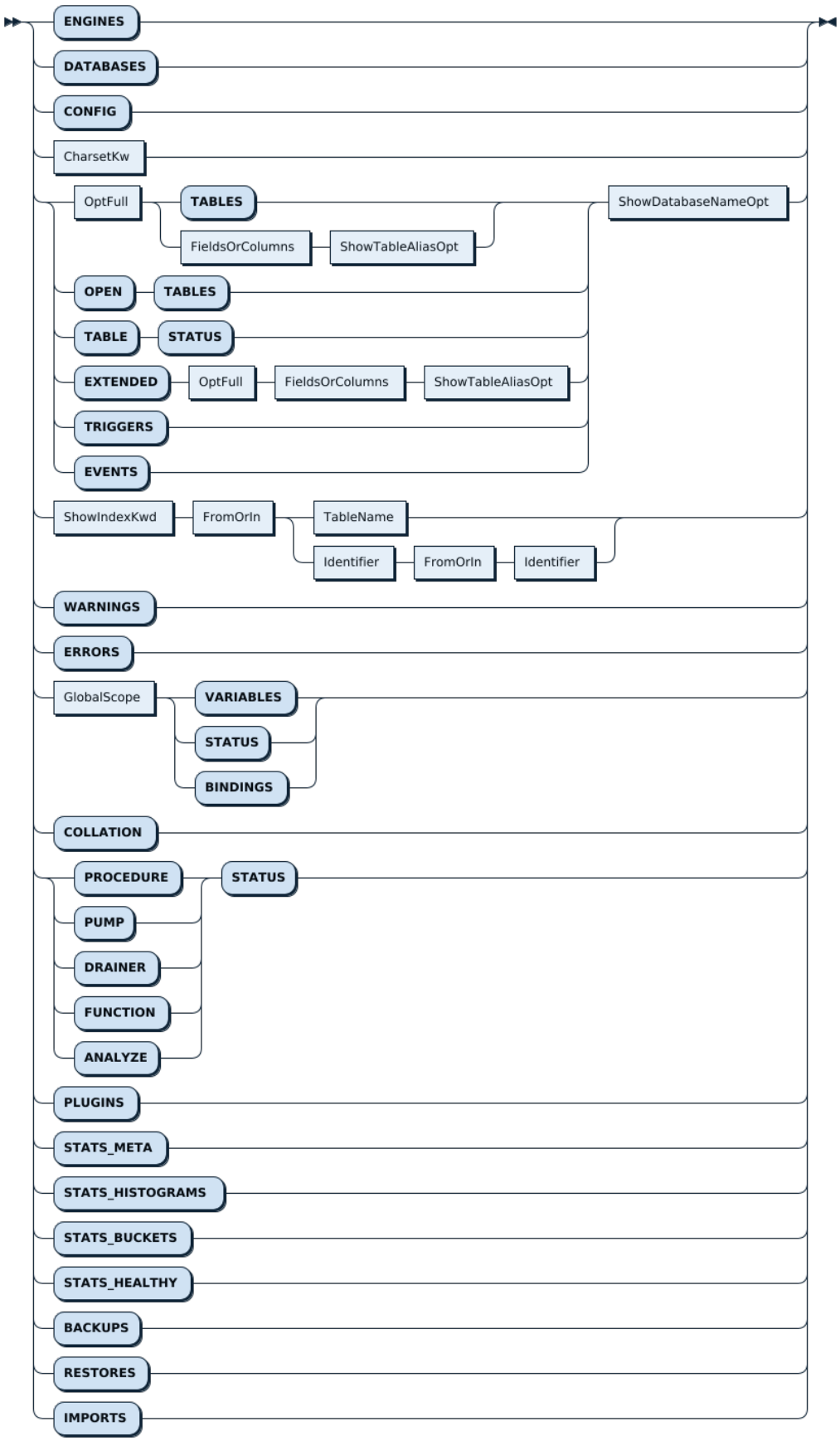


图 270: ShowTargetFilterable

GlobalScope:

图 271: GlobalScope

ShowLikeOrWhereOpt

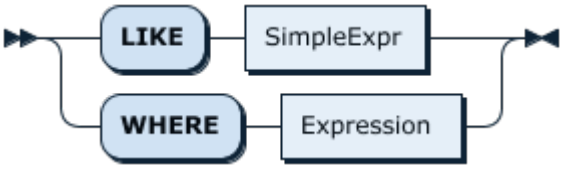


图 272: ShowLikeOrWhereOpt

12.11.2.75.2 语法说明

```
SHOW [GLOBAL | SESSION] BINDINGS [ShowLikeOrWhereOpt];
```

该语句会输出 GLOBAL 或者 SESSION 作用域内的执行计划绑定，在不指定作用域时默认作用域为 SESSION。目前 SHOW BINDINGS 会输出 8 列，具体如下：

列名	说明
original_sql	参数化后的原始 SQL
bind_sql	带 Hint 的绑定 SQL
default_db	默认数据库名
status	状态，包括 using（正在使用）、deleted（已删除）、invalid（无效）、rejected（演进时被拒绝）和 pending verify（等待演进验证）
create_time	创建时间
update_time	更新时间
charset	字符集
collation	排序规则

列名	说明
source	创建方式，包括 manual（由 create [global] binding 生成）、capture（由 tidb 自动创建生成）和 evolve（由 tidb 自动演进生成）

12.11.2.75.3 示例

```
CREATE TABLE t1 (
    ->  id INT NOT NULL PRIMARY KEY auto_increment,
    ->  b INT NOT NULL,
    ->  pad VARBINARY(255),
    ->  INDEX(b)
    -> );
Query OK, 0 rows affected (0.07 sec)

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM dual;
Query OK, 1 row affected (0.01 sec)
Records: 1  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 1 row affected (0.00 sec)
Records: 1  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 8 rows affected (0.00 sec)
Records: 8  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 1000 rows affected (0.04 sec)
Records: 1000  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 100000 rows affected (1.74 sec)
Records: 100000  Duplicates: 0  Warnings: 0
```

```
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 100000 rows affected (2.15 sec)
Records: 100000  Duplicates: 0  Warnings: 0

INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
Query OK, 100000 rows affected (2.64 sec)
Records: 100000  Duplicates: 0  Warnings: 0

SELECT SLEEP(1);
+----------+
| SLEEP(1) |
+----------+
|        0 |
+----------+
1 row in set (1.00 sec)

ANALYZE TABLE t1;
Query OK, 0 rows affected (1.33 sec)

EXPLAIN ANALYZE SELECT * FROM t1 WHERE b = 123;
+--
    ↪ ----------------------------+---------+---------+-----------+---------------------+-----------------
    ↪
| id                          | estRows | actRows | task      | access object       |
    ↪ execution info                                                                | operator info
    ↪                       | memory         | disk |
+--
    ↪ ----------------------------+---------+---------+-----------+---------------------+-----------------
    ↪
| IndexLookUp_10                | 583.00  | 297     | root      |                     | time
    ↪ :10.545072ms, loops:2, rpc num: 1, rpc time:398.359µs, proc keys:297  |
    ↪                                     | 109.1484375 KB | N/A  |
| ├─IndexRangeScan_8(Build)     | 583.00  | 297     | cop[tikv] | table:t1, index:b(b) | time:0s
    ↪ , loops:4                                                             | range:[123,123],
    ↪ keep order:false | N/A            | N/A  |
| └─TableRowIDScan_9(Probe)     | 583.00  | 297     | cop[tikv] | table:t1             | time:12
    ↪ ms, loops:4                                                           | keep order:false
    ↪                   | N/A            | N/A  |
+--
    ↪ ----------------------------+---------+---------+-----------+---------------------+-----------------
    ↪
3 rows in set (0.02 sec)
```

```
CREATE SESSION BINDING FOR
    ->  SELECT * FROM t1 WHERE b = 123
    -> USING
    ->  SELECT * FROM t1 IGNORE INDEX (b) WHERE b = 123;
Query OK, 0 rows affected (0.00 sec)

EXPLAIN ANALYZE  SELECT * FROM t1 WHERE b = 123;
+--
    ↪ ----------------------+----------+---------+----------+---------------+-------------------------------
    ↪
| id                    | estRows   | actRows | task      | access object | execution info
    ↪                                                                  | operator info      |
    ↪ memory        | disk |
+--
    ↪ ----------------------+----------+---------+----------+---------------+-------------------------------
    ↪
| TableReader_7           | 583.00    | 297     | root      |               | time:222.32506ms,
    ↪ loops:2, rpc num: 1, rpc time:222.078952ms, proc keys:301010 | data:Selection_6   |
    ↪ 88.6640625 KB | N/A  |
| └─Selection_6           | 583.00    | 297     | cop[tikv] |               | time:224ms, loops
    ↪ :298                                                          | eq(test.t1.b, 123) | N/A
    ↪           | N/A  |
|   └─TableFullScan_5     | 301010.00 | 301010  | cop[tikv] | table:t1      | time:220ms, loops
    ↪ :298                                                          | keep order:false   | N/A
    ↪           | N/A  |
+--
    ↪ ----------------------+----------+---------+----------+---------------+-------------------------------
    ↪
3 rows in set (0.22 sec)

SHOW SESSION BINDINGS\G
*************************** 1. row ***************************
Original_sql: select * from t1 where b = ?
    Bind_sql: SELECT * FROM t1 IGNORE INDEX (b) WHERE b = 123
  Default_db: test
      Status: using
 Create_time: 2020-05-22 14:38:03.456
 Update_time: 2020-05-22 14:38:03.456
     Charset: utf8mb4
   Collation: utf8mb4_0900_ai_ci
1 row in set (0.00 sec)
```

12.11.2.75.4 MySQL 兼容性

`SHOW [GLOBAL|SESSION] BINDINGS` 语句是 TiDB 对 MySQL 语法的扩展。

12.11.2.75.5 另请参阅

- CREATE [GLOBAL|SESSION] BINDING
- DROP [GLOBAL|SESSION] BINDING
- ANALYZE
- Optimizer Hints
- 执行计划管理 (SPM)

12.11.2.76 SHOW BUILTINS

`SHOW BUILTINS` 语句用于列出 TiDB 中所有的内置函数。

12.11.2.76.1 语法图

ShowBuiltinsStmt:

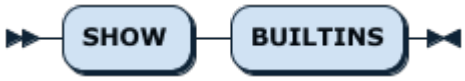


图 273: ShowBuiltinsStmt

12.11.2.76.2 示例

```
SHOW BUILTINS;
```

```
+----------------------------+
| Supported_builtin_functions |
+----------------------------+
| abs                        |
| acos                       |
| adddate                    |
| addtime                    |
| aes_decrypt                |
| aes_encrypt                |
| and                        |
| any_value                  |
| ascii                      |
| asin                       |
| atan                       |
| atan2                      |
| benchmark                  |
| bin                        |
| bit_count                  |
| bit_length                 |
| bitand                     |
| bitneg                     |
```

```
| bitor                     |
| bitxor                    |
| case                      |
| ceil                      |
| ceiling                   |
| char_func                 |
| char_length               |
| character_length          |
| charset                   |
| coalesce                  |
| coercibility              |
| collation                 |
| compress                  |
| concat                    |
| concat_ws                 |
| connection_id             |
| conv                      |
| convert                   |
| convert_tz                |
| cos                       |
| cot                       |
| crc32                     |
| curdate                   |
| current_date              |
| current_role              |
| current_time              |
| current_timestamp         |
| current_user              |
| curtime                   |
| database                  |
| date                      |
| date_add                  |
| date_format               |
| date_sub                  |
| datediff                  |
| day                       |
| dayname                   |
| dayofmonth                |
| dayofweek                 |
| dayofyear                 |
| decode                    |
| default_func              |
| degrees                   |
| des_decrypt               |
| des_encrypt               |
```

```
| div                        |
| elt                        |
| encode                     |
| encrypt                    |
| eq                         |
| exp                        |
| export_set                 |
| extract                    |
| field                      |
| find_in_set                |
| floor                      |
| format                     |
| format_bytes               |
| format_nano_time           |
| found_rows                 |
| from_base64                |
| from_days                  |
| from_unixtime              |
| ge                         |
| get_format                 |
| get_lock                   |
| getparam                   |
| getvar                     |
| greatest                   |
| gt                         |
| hex                        |
| hour                       |
| if                         |
| ifnull                     |
| in                         |
| inet6_aton                 |
| inet6_ntoa                 |
| inet_aton                  |
| inet_ntoa                  |
| insert_func                |
| instr                      |
| intdiv                     |
| interval                   |
| is_free_lock               |
| is_ipv4                    |
| is_ipv4_compat             |
| is_ipv4_mapped             |
| is_ipv6                    |
| is_used_lock               |
| isfalse                    |
```

```
| isnull                    |
| istrue                    |
| json_array                |
| json_array_append         |
| json_array_insert         |
| json_contains             |
| json_contains_path        |
| json_depth                |
| json_extract              |
| json_insert               |
| json_keys                 |
| json_length               |
| json_merge                |
| json_merge_patch          |
| json_merge_preserve       |
| json_object               |
| json_pretty               |
| json_quote                |
| json_remove               |
| json_replace              |
| json_search               |
| json_set                  |
| json_storage_size         |
| json_type                 |
| json_unquote              |
| json_valid                |
| last_day                  |
| last_insert_id            |
| lastval                   |
| lcase                     |
| le                        |
| least                     |
| left                      |
| leftshift                 |
| length                    |
| like                      |
| ln                        |
| load_file                 |
| localtime                 |
| localtimestamp            |
| locate                    |
| log                       |
| log10                     |
| log2                      |
| lower                     |
```

```
| lpad                        |
| lt                          |
| ltrim                       |
| make_set                    |
| makedate                    |
| maketime                    |
| master_pos_wait             |
| md5                         |
| microsecond                 |
| mid                         |
| minus                       |
| minute                      |
| mod                         |
| month                       |
| monthname                   |
| mul                         |
| name_const                  |
| ne                          |
| nextval                     |
| not                         |
| now                         |
| nulleq                      |
| oct                         |
| octet_length                |
| old_password                |
| or                          |
| ord                         |
| password_func               |
| period_add                  |
| period_diff                 |
| pi                          |
| plus                        |
| position                    |
| pow                         |
| power                       |
| quarter                     |
| quote                       |
| radians                     |
| rand                        |
| random_bytes                |
| regexp                      |
| release_all_locks           |
| release_lock                |
| repeat                      |
| replace                     |
```

```
| reverse                    |
| right                      |
| rightshift                 |
| round                      |
| row_count                  |
| rpad                       |
| rtrim                      |
| schema                     |
| sec_to_time                |
| second                     |
| session_user               |
| setval                     |
| setvar                     |
| sha                        |
| sha1                       |
| sha2                       |
| sign                       |
| sin                        |
| sleep                      |
| space                      |
| sqrt                       |
| str_to_date                |
| strcmp                     |
| subdate                    |
| substr                     |
| substring                  |
| substring_index            |
| subtime                    |
| sysdate                    |
| system_user                |
| tan                        |
| tidb_decode_key            |
| tidb_decode_plan           |
| tidb_is_ddl_owner          |
| tidb_parse_tso             |
| tidb_version               |
| time                       |
| time_format                |
| time_to_sec                |
| timediff                   |
| timestamp                  |
| timestampadd               |
| timestampdiff              |
| to_base64                  |
| to_days                    |
```

```
| to_seconds                    |
| trim                          |
| truncate                      |
| ucase                         |
| unaryminus                    |
| uncompress                    |
| uncompressed_length           |
| unhex                         |
| unix_timestamp                |
| upper                         |
| user                          |
| utc_date                      |
| utc_time                      |
| utc_timestamp                 |
| uuid                          |
| uuid_short                    |
| validate_password_strength    |
| version                       |
| week                          |
| weekday                       |
| weekofyear                    |
| weight_string                 |
| xor                           |
| year                          |
| yearweek                      |
+-------------------------------+
268 rows in set (0.00 sec)
```

12.11.2.76.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

12.11.2.77 SHOW CHARACTER SET

`SHOW CHARACTER SET` 语句提供 TiDB 中可用字符集的静态列表。此列表不反映当前连接或用户的任何属性。

12.11.2.77.1 语法图

ShowCharsetStmt:

图 274: ShowCharsetStmt

CharsetKw:

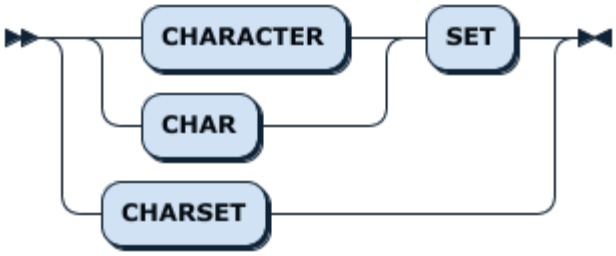


图 275: CharsetKw

12.11.2.77.2 示例

```
SHOW CHARACTER SET;
```

```
+---------+---------------+-------------------+--------+
| Charset | Description   | Default collation | Maxlen |
+---------+---------------+-------------------+--------+
| utf8    | UTF-8 Unicode | utf8_bin          |      3 |
| utf8mb4 | UTF-8 Unicode | utf8mb4_bin       |      4 |
| ascii   | US ASCII      | ascii_bin         |      1 |
| latin1  | Latin1        | latin1_bin        |      1 |
| binary  | binary        | binary            |      1 |
+---------+---------------+-------------------+--------+
5 rows in set (0.00 sec)
```

12.11.2.77.3 MySQL 兼容性

SHOW CHARACTER SET 语句功能与 MySQL 完全兼容。注意，TiDB 中字符集的默认排序规则与 MySQL 有所不同，具体可以参考与 MySQL 兼容性对比。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.77.4 另请参阅

- SHOW COLLATION
- 字符集和排序规则

12.11.2.78 SHOW COLLATION

SHOW COLLATION 语句用于提供一个静态的排序规则列表，确保与 MySQL 客户端库的兼容性。

> **注意：**
>
> SHOW COLLATION 所展示的排序规则列表与 TiDB 集群是否开启新排序规则框架有关，详情请见TiDB 字符集和排序规则。

12.11.2.78.1 语法图

ShowCollationStmt:

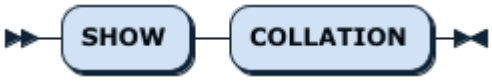


图 276: ShowCollationStmt

12.11.2.78.2 示例

若未开启新排序规则框架，仅展示二进制排序规则：

```
SHOW COLLATION;
```

```
+-------------+---------+------+---------+----------+---------+
| Collation   | Charset | Id   | Default | Compiled | Sortlen |
+-------------+---------+------+---------+----------+---------+
| utf8mb4_bin | utf8mb4 |   46 | Yes     | Yes      |       1 |
| latin1_bin  | latin1  |   47 | Yes     | Yes      |       1 |
| binary      | binary  |   63 | Yes     | Yes      |       1 |
| ascii_bin   | ascii   |   65 | Yes     | Yes      |       1 |
| utf8_bin    | utf8    |   83 | Yes     | Yes      |       1 |
+-------------+---------+------+---------+----------+---------+
5 rows in set (0.02 sec)
```

若开启了新排序规则框架，则在二进制排序规则之外，额外支持 `utf8_general_ci` 和 `utf8mb4_general_ci` 两种大小写和口音不敏感的排序规则：

```
SHOW COLLATION;
```

```
+--------------------+---------+------+---------+----------+---------+
| Collation          | Charset | Id   | Default | Compiled | Sortlen |
+--------------------+---------+------+---------+----------+---------+
| ascii_bin          | ascii   |   65 | Yes     | Yes      |       1 |
| binary             | binary  |   63 | Yes     | Yes      |       1 |
| latin1_bin         | latin1  |   47 | Yes     | Yes      |       1 |
| utf8_bin           | utf8    |   83 | Yes     | Yes      |       1 |
| utf8_general_ci    | utf8    |   33 |         | Yes      |       1 |
| utf8mb4_bin        | utf8mb4 |   46 | Yes     | Yes      |       1 |
| utf8mb4_general_ci | utf8mb4 |   45 |         | Yes      |       1 |
+--------------------+---------+------+---------+----------+---------+
7 rows in set (0.00 sec)
```

12.11.2.78.3 MySQL 兼容性

`SHOW COLLATION` 语句功能与 MySQL 完全兼容。注意，TiDB 中字符集的默认排序规则与 MySQL 有所不同，具体可参考与 MySQL 兼容性对比。如发现任何其他兼容性差异，请在 GitHub 上提交 issue。

12.11.2.78.4 另请参阅

- SHOW CHARACTER SET
- 字符集和排序规则

12.11.2.79 SHOW [FULL] COLUMNS FROM

`SHOW [FULL] COLUMNS FROM <table_name>` 语句用于以表格格式描述表或视图中的列。可选关键字 `FULL` 用于显示当前用户对该列的权限，以及表定义中的 `comment`。

`SHOW [FULL] FIELDS FROM <table_name>`、`DESC <table_name>`、`DESCRIBE <table_name>` 和 `EXPLAIN <table_name>` 语句都是 `SHOW [FULL] COLUMNS FROM` 的别名。

> **注意：**
>
> `DESC TABLE <table_name>`、`DESCRIBE TABLE <table_name>` 和 `EXPLAIN TABLE <table_name>` 与上面的语句并不等价，它们是`DESC SELECT * FROM <table_name>` 的别名。

12.11.2.79.1 语法图

ShowStmt:

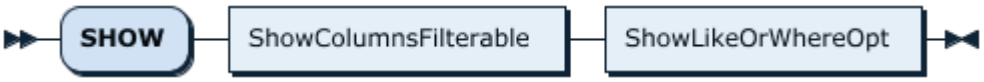


图 277: ShowStmt

ShowColumnsFilterable:

图 278: ShowColumnsFilterable

OptFull:

图 279: OptFull

FieldsOrColumns:

图 280: FieldsOrColumns

ShowTableAliasOpt:

图 281: ShowTableAliasOpt

FromOrIn:

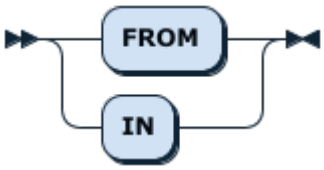


图 282: FromOrIn

TableName:

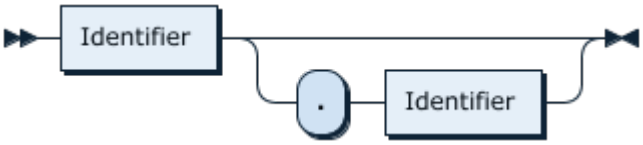


图 283: TableName

ShowDatabaseNameOpt:

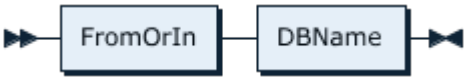


图 284: ShowDatabaseNameOpt

DBName:

图 285: DBName

ShowLikeOrWhereOpt:

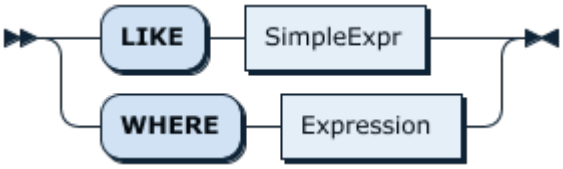


图 286: ShowLikeOrWhereOpt

12.11.2.79.2 示例

```
create view v1 as select 1;
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
show columns from v1;
```

```
+-------+-----------+------+------+---------+-------+
| Field | Type      | Null | Key  | Default | Extra |
+-------+-----------+------+------+---------+-------+
| 1     | bigint(1) | YES  |      | NULL    |       |
+-------+-----------+------+------+---------+-------+
1 row in set (0.00 sec)
```

```
desc v1;
```

```
+-------+-----------+------+------+---------+-------+
| Field | Type      | Null | Key  | Default | Extra |
+-------+-----------+------+------+---------+-------+
| 1     | bigint(1) | YES  |      | NULL    |       |
+-------+-----------+------+------+---------+-------+
1 row in set (0.00 sec)
```

```
describe v1;
```

```
+-------+-----------+------+------+---------+-------+
| Field | Type      | Null | Key  | Default | Extra |
+-------+-----------+------+------+---------+-------+
| 1     | bigint(1) | YES  |      | NULL    |       |
+-------+-----------+------+------+---------+-------+
1 row in set (0.00 sec)
```

```
explain v1;
```

```
+-------+-----------+------+------+---------+-------+
| Field | Type      | Null | Key  | Default | Extra |
+-------+-----------+------+------+---------+-------+
| 1     | bigint(1) | YES  |      | NULL    |       |
+-------+-----------+------+------+---------+-------+
1 row in set (0.00 sec)
```

```
show fields from v1;
```

```
+-------+-----------+------+------+---------+-------+
| Field | Type      | Null | Key  | Default | Extra |
+-------+-----------+------+------+---------+-------+
| 1     | bigint(1) | YES  |      | NULL    |       |
+-------+-----------+------+------+---------+-------+
1 row in set (0.00 sec)
```

```
show full columns from v1;
```

```
+-------+-----------+-----------+------+------+---------+-------+---------------------------------+---------+
    ↪
| Field | Type      | Collation | Null | Key  | Default | Extra | Privileges
    ↪                       | Comment |
+-------+-----------+-----------+------+------+---------+-------+---------------------------------+---------+
    ↪
| 1     | bigint(1) | NULL      | YES  |      | NULL    |       | select,insert,update,references
    ↪  |         |
+-------+-----------+-----------+------+------+---------+-------+---------------------------------+---------+
    ↪
1 row in set (0.00 sec)
```

```
show full columns from mysql.user;
```

```
+-----------------------+---------------+-------------+------+------+---------+-------+-----------------------------
    ↪
| Field                 | Type          | Collation   | Null | Key  | Default | Extra |
    ↪ Privileges                      | Comment |
+-----------------------+---------------+-------------+------+------+---------+-------+-----------------------------
    ↪
| Host                  | char(64)      | utf8mb4_bin | NO   | PRI  | NULL    |       | select,
    ↪ insert,update,references |         |
| User                  | char(32)      | utf8mb4_bin | NO   | PRI  | NULL    |       | select,
    ↪ insert,update,references |         |
| authentication_string | text          | utf8mb4_bin | YES  |      | NULL    |       | select,
    ↪ insert,update,references |         |
```

```
| Select_priv           | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Insert_priv           | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Update_priv           | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Delete_priv           | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Create_priv           | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Drop_priv             | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Process_priv          | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Grant_priv            | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| References_priv       | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Alter_priv            | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Show_db_priv          | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Super_priv            | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Create_tmp_table_priv | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Lock_tables_priv      | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Execute_priv          | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Create_view_priv      | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Show_view_priv        | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Create_routine_priv   | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Alter_routine_priv    | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Index_priv            | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Create_user_priv      | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Event_priv            | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
```

```
| Trigger_priv          | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Create_role_priv      | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Drop_role_priv        | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Account_locked        | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Shutdown_priv         | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Reload_priv           | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| FILE_priv             | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
| Config_priv           | enum('N','Y') | utf8mb4_bin | NO   |      | N       |       | select,
    ↪ insert,update,references |         |
+-----------------------+---------------+-------------+------+------+---------+-------+-----------------------------
    ↪
33 rows in set (0.01 sec)
```

12.11.2.79.3 MySQL 兼容性

`SHOW [FULL] COLUMNS FROM` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.79.4 另请参阅

- SHOW CREATE TABLE

12.11.2.80 SHOW CONFIG

`SHOW CONFIG` 语句用于展示 TiDB 各个组件当前正在应用的配置，请注意，配置与系统变量作用于不同维度，请不要混淆，如果希望获取系统变量信息，请使用SHOW VARIABLES 语法。

12.11.2.80.1 语法图

ShowStmt:

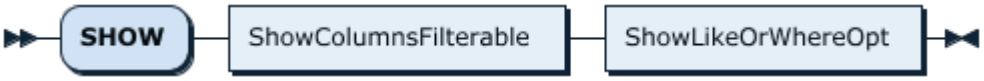


图 287: ShowStmt

ShowTargetFilterable:

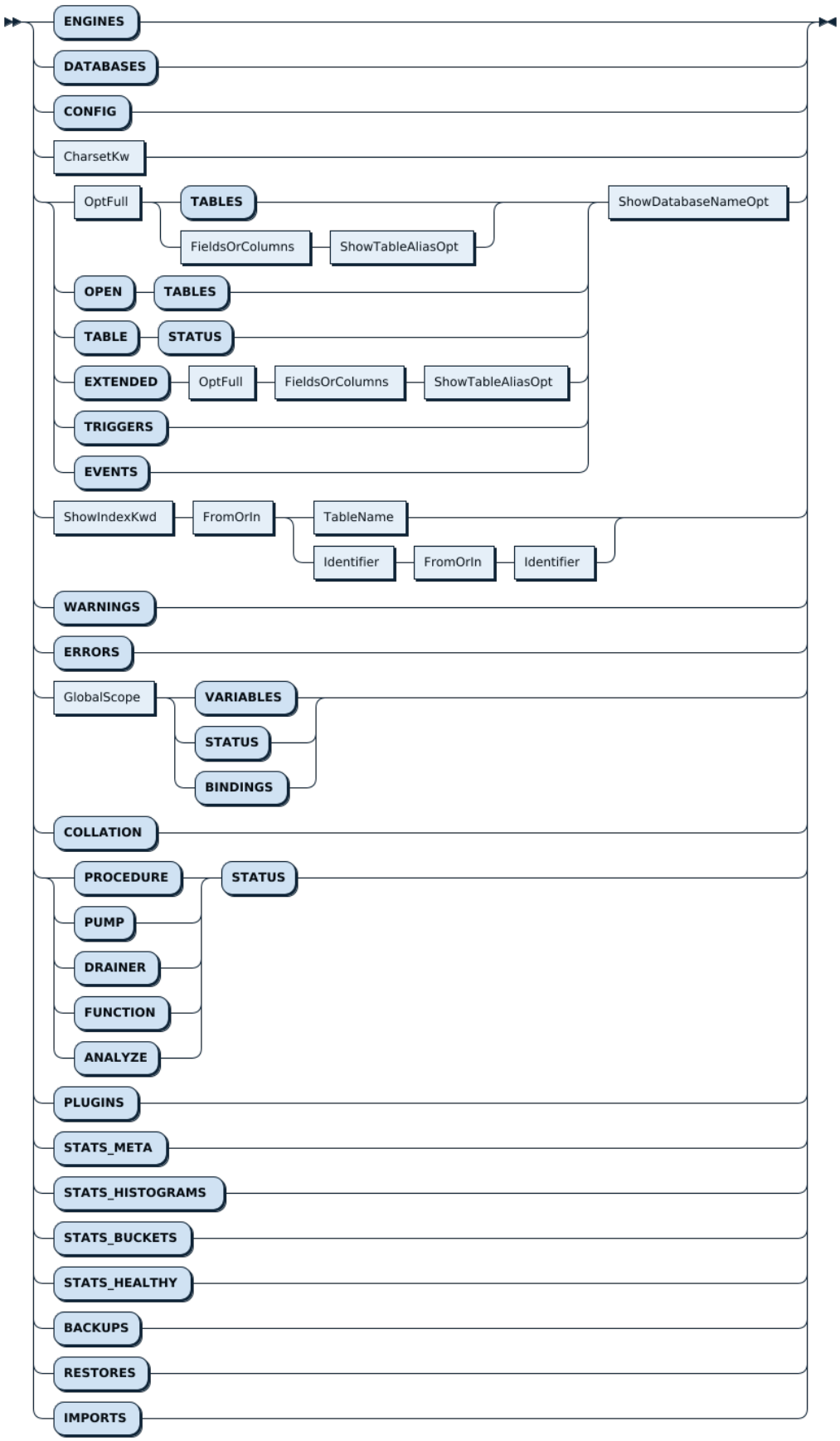


图 288: ShowTargetFilterable

12.11.2.80.2 示例

显示所有配置：

```
SHOW CONFIG;
```

```
+------+----------------+-------------------------------------------------+-------------------------------------
    ↪
| Type | Instance       | Name                                            | Value
    ↪                                                                  |
+------+----------------+-------------------------------------------------+-------------------------------------
    ↪
| tidb | 127.0.0.1:4000 | advertise-address                               | 127.0.0.1
    ↪                                                                |
| tidb | 127.0.0.1:4000 | alter-primary-key                               | false
    ↪                                                                |
| tidb | 127.0.0.1:4000 | binlog.binlog-socket                            |
    ↪                                                                  |
| tidb | 127.0.0.1:4000 | binlog.enable                                   | false
    ↪                                                                |
...
120 rows in set (0.01 sec)
```

显示 type 是 tidb 的配置：

```
SHOW CONFIG WHERE type = 'tidb' AND name = 'advertise-address';
```

```
+------+----------------+------------------+-----------+
| Type | Instance       | Name             | Value     |
+------+----------------+------------------+-----------+
| tidb | 127.0.0.1:4000 | advertise-address | 127.0.0.1 |
+------+----------------+------------------+-----------+
1 row in set (0.05 sec)
```

也可以用 LIKE 子句来显示 type 是 tidb 的配置：

```
SHOW CONFIG LIKE 'tidb';
```

```
+------+----------------+-------------------------------------------------+-------------------------------------
    ↪
| Type | Instance       | Name                                            | Value
    ↪                                                                  |
+------+----------------+-------------------------------------------------+-------------------------------------
    ↪
| tidb | 127.0.0.1:4000 | advertise-address                               | 127.0.0.1
    ↪                                                                |
| tidb | 127.0.0.1:4000 | alter-primary-key                               | false
    ↪                                                                |
```

```
| tidb | 127.0.0.1:4000 | binlog.binlog-socket                                    |
    ↪                                                                  |
| tidb | 127.0.0.1:4000 | binlog.enable                                           | false
    ↪                                                                |
...
40 rows in set (0.01 sec)
```

12.11.2.80.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

12.11.2.80.4 另请参阅

- SHOW VARIABLES

12.11.2.81 SHOW CREATE SEQUENCE

`SHOW CREATE SEQUENCE` 语句用于查看一个序列的详细信息，类似于 `SHOW CREATE TABLE` 语句。

12.11.2.81.1 语法图

ShowCreateSequenceStmt:

图 289: ShowCreateSequenceStmt

TableName:

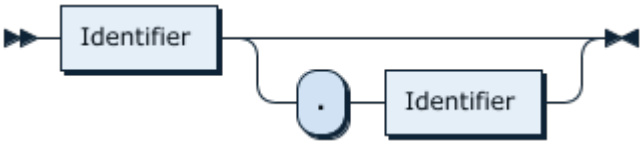


图 290: TableName

12.11.2.81.2 示例

```
CREATE SEQUENCE seq;
```

```
Query OK, 0 rows affected (0.03 sec)
```

```
SHOW CREATE SEQUENCE seq;
```

```
+-------+-------------------------------------------------------------------------------------------------------------------------------
    ↪
| Table | Create Table
    ↪
    ↪ |
+-------+-------------------------------------------------------------------------------------------------------------------------------
    ↪
| seq   | CREATE SEQUENCE `seq` start with 1 minvalue 1 maxvalue 9223372036854775806 increment by
    ↪  1 cache 1000 nocycle ENGINE=InnoDB |
+-------+-------------------------------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

12.11.2.81.3 MySQL 兼容性

该语句是 TiDB 的扩展，序列的实现借鉴自 MariaDB。

12.11.2.81.4 另请参阅

- CREATE SEQUENCE
- DROP SEQUENCE

12.11.2.82 SHOW CREATE TABLE

`SHOW CREATE TABLE` 语句用于显示用 SQL 重新创建已有表的确切语句。

12.11.2.82.1 语法图

ShowCreateTableStmt:

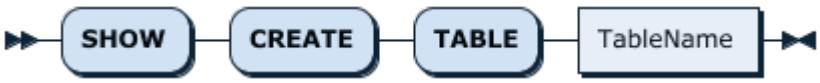


图 291: ShowCreateTableStmt

TableName:

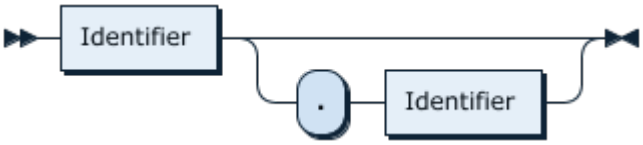


图 292: TableName

12.11.2.82.2 示例

```
CREATE TABLE t1 (a INT);
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
SHOW CREATE TABLE t1;
```

```
+-------+-----------------------------------------------------------------------------------------------------------------
    ↪
| Table | Create Table
    ↪
    ↪ |
+-------+-----------------------------------------------------------------------------------------------------------------
    ↪
| t1    | CREATE TABLE `t1` (
  `a` int(11) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin |
+-------+-----------------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

12.11.2.82.3 MySQL 兼容性

`SHOW CREATE TABLE` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.82.4 另请参阅

- CREATE TABLE
- DROP TABLE
- SHOW TABLES
- SHOW COLUMNS FROM

12.11.2.83 SHOW CREATE USER

`SHOW CREATE USER` 语句用于显示如何使用 `CREATE USER` 语法来重新创建用户。

12.11.2.83.1 语法图

ShowCreateUserStmt:

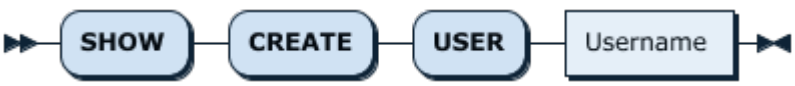


图 293: ShowCreateUserStmt

Username:

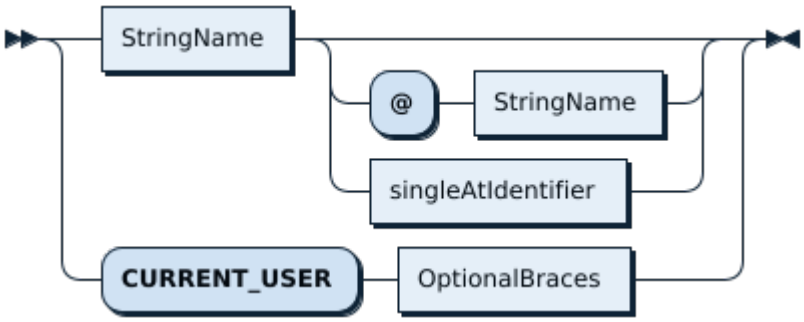


图 294: Username

12.11.2.83.2 示例

```
SHOW CREATE USER 'root';
```

```
+-------------------------------------------------------------------------------------------------------------------------------
    ↪
| CREATE USER for root@%
    ↪
    ↪ |
+-------------------------------------------------------------------------------------------------------------------------------
    ↪
| CREATE USER 'root'@'%' IDENTIFIED WITH 'mysql_native_password' AS '' REQUIRE NONE PASSWORD
    ↪ EXPIRE DEFAULT ACCOUNT UNLOCK |
+-------------------------------------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)

mysql> SHOW GRANTS FOR 'root';
+----------------------------------------+
| Grants for root@%                      |
+----------------------------------------+
| GRANT ALL PRIVILEGES ON *.* TO 'root'@'%' |
+----------------------------------------+
1 row in set (0.00 sec)
```

12.11.2.83.3 MySQL 兼容性

- SHOW CREATE USER 的输出结果旨在匹配 MySQL，但 TiDB 尚不支持若干 CREATE 选项。尚未支持的选项在语句执行过程中会被解析但会被跳过执行。详情可参阅 [security compatibility]。

12.11.2.83.4 另请参阅

- CREATE USER
- SHOW GRANTS
- DROP USER

12.11.2.84 SHOW DATABASES

`SHOW DATABASES` 语句用于显示当前用户有权访问的数据库列表。当前用户无权访问的数据库将从列表中隐藏。`information_schema` 数据库始终出现在列表的最前面。

`SHOW SCHEMAS` 是 `SHOW DATABASES` 语句的别名。

12.11.2.84.1 语法图

ShowDatabasesStmt:

图 295: ShowDatabasesStmt

ShowLikeOrWhereOpt:

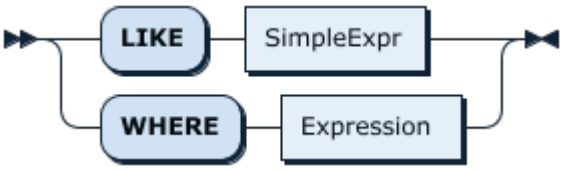


图 296: ShowLikeOrWhereOpt

12.11.2.84.2 示例

```
SHOW DATABASES;
```

```
+--------------------+
| Database           |
+--------------------+
| INFORMATION_SCHEMA |
| PERFORMANCE_SCHEMA |
| mysql              |
| test               |
+--------------------+
4 rows in set (0.00 sec)
```

```
CREATE DATABASE mynewdb;
```

```
Query OK, 0 rows affected (0.10 sec)
```

```
SHOW DATABASES;
```

```
+--------------------+
| Database           |
+--------------------+
| INFORMATION_SCHEMA |
| PERFORMANCE_SCHEMA |
| mynewdb            |
| mysql              |
| test               |
+--------------------+
5 rows in set (0.00 sec)
```

12.11.2.84.3 MySQL 兼容性

`SHOW DATABASES` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.84.4 另请参阅

- SHOW SCHEMAS
- DROP DATABASE
- CREATE DATABASE

12.11.2.85 SHOW DRAINER STATUS

`SHOW DRAINER STATUS` 语句用于显示集群中所有 Drainer 的状态信息。

12.11.2.85.1 示例

```
SHOW DRAINER STATUS;
```

```
+----------|----------------|--------|--------------------|---------------------|
|  NodeID  |     Address    | State  |   Max_Commit_Ts    |    Update_Time      |
+----------|----------------|--------|--------------------|---------------------|
| drainer1 | 127.0.0.3:8249 | Online | 408553768673342532 | 2019-05-01 00:00:03 |
+----------|----------------|--------|--------------------|---------------------|
| drainer2 | 127.0.0.4:8249 | Online | 408553768673345531 | 2019-05-01 00:00:04 |
+----------|----------------|--------|--------------------|---------------------|
2 rows in set (0.00 sec)
```

12.11.2.85.2 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

12.11.2.85.3 另请参阅

- SHOW PUMP STATUS
- CHANGE PUMP STATUS
- CHANGE DRAINER STATUS

12.11.2.86 SHOW ENGINES

`SHOW ENGINES` 语句用于列出所有支持的存储引擎，该语法仅提供 MySQL 兼容性。

12.11.2.86.1 语法图

ShowEnginesStmt:

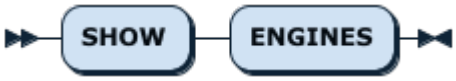


图 297: ShowEnginesStmt

```
SHOW ENGINES;
```

12.11.2.86.2 示例

```
SHOW ENGINES;
```

```
+--------+---------+----------------------------------------------------------------+--------------+------+----------
    ↪
| Engine | Support | Comment                                                        | Transactions |
    ↪ XA   | Savepoints |
+--------+---------+----------------------------------------------------------------+--------------+------+----------
    ↪
| InnoDB | DEFAULT | Supports transactions, row-level locking, and foreign keys | YES          |
    ↪ YES  | YES        |
+--------+---------+----------------------------------------------------------------+--------------+------+----------
    ↪
1 row in set (0.00 sec)
```

12.11.2.86.3 MySQL 兼容性

- `SHOW ENGINES` 语句始终只返回 InnoDB 作为其支持的引擎。但 TiDB 内部通常使用 TiKV 作为存储引擎。

12.11.2.87 SHOW ERRORS

`SHOW ERRORS` 语句用于显示已执行语句中的错误。一旦先前的语句成功执行，就会清除错误缓冲区，这时 `SHOW ERRORS` 会返回一个空集。

当前的 `sql_mode` 很大程度决定了哪些语句会产生错误与警告。

12.11.2.87.1 语法图

ShowErrorsStmt:

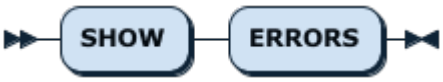


图 298: ShowErrorsStmt

12.11.2.87.2 示例

```
select invalid;
```

```
ERROR 1054 (42S22): Unknown column 'invalid' in 'field list'
```

```
create invalid;
```

```
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to
    ↪ your TiDB version for the right syntax to use line 1 column 14 near "invalid"
```

```
SHOW ERRORS;
```

```
+-------+------+-----------------------------------------------------------------------------------------------
    ↪
| Level | Code | Message
    ↪
    ↪ |
+-------+------+-----------------------------------------------------------------------------------------------
    ↪
| Error | 1054 | Unknown column 'invalid' in 'field list'
    ↪
    ↪ |
| Error | 1064 | You have an error in your SQL syntax; check the manual that corresponds to your
    ↪ TiDB version for the right syntax to use line 1 column 14 near "invalid"  |
+-------+------+-----------------------------------------------------------------------------------------------
    ↪
2 rows in set (0.00 sec)
```

```
CREATE invalid2;
```

```
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to
    ↪ your TiDB version for the right syntax to use line 1 column 15 near "invalid2"
```

```
SELECT 1;
```

```
+------+
| 1    |
+------+
|    1 |
+------+
1 row in set (0.00 sec)
```

```
SHOW ERRORS;
```

```
Empty set (0.00 sec)
```

12.11.2.87.3 MySQL 兼容性

`SHOW ERRORS` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.87.4 另请参阅

- SHOW WARNINGS

12.11.2.88 SHOW [FULL] FIELDS FROM

`SHOW [FULL] FIELDS FROM` 是`SHOW [FULL] COLUMNS FROM` 的别名。包含该语句提供了 MySQL 兼容性。

12.11.2.89 SHOW GRANTS

`SHOW GRANTS` 语句用于显示与用户关联的权限列表。与在 MySQL 中一样，`USAGE` 权限表示登录 TiDB 的能力。

12.11.2.89.1 语法图

ShowGrantsStmt:

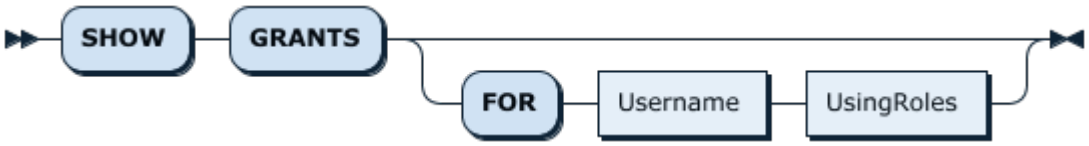


图 299: ShowGrantsStmt

Username:

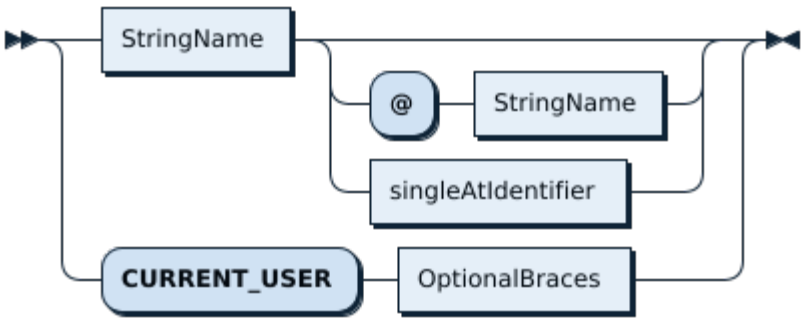


图 300: Username

UsingRoles:

图 301: UsingRoles

RolenameList:

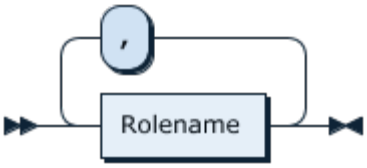


图 302: RolenameList

Rolename:

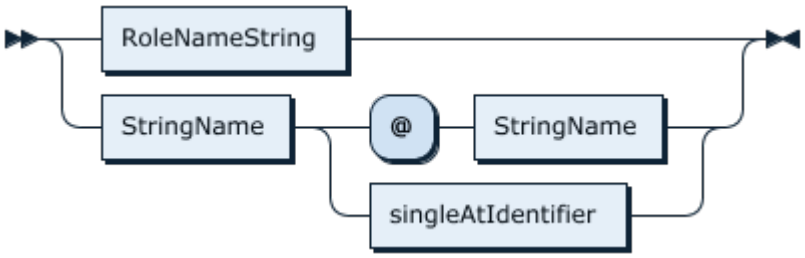


图 303: Rolename

12.11.2.89.2 示例

```
SHOW GRANTS;
```

```
+----------------------------------------+
| Grants for User                        |
```

```
+-----------------------------------------+
| GRANT ALL PRIVILEGES ON *.* TO 'root'@'%' |
+-----------------------------------------+
1 row in set (0.00 sec)
```

```
SHOW GRANTS FOR 'u1';
```

```
ERROR 1141 (42000): There is no such grant defined for user 'u1' on host '%'
```

```
CREATE USER u1;
```

```
Query OK, 1 row affected (0.04 sec)
```

```
GRANT SELECT ON test.* TO u1;
```

```
Query OK, 0 rows affected (0.04 sec)
```

```
SHOW GRANTS FOR u1;
```

```
+-----------------------------------+
| Grants for u1@%                   |
+-----------------------------------+
| GRANT USAGE ON *.* TO 'u1'@'%'    |
| GRANT Select ON test.* TO 'u1'@'%' |
+-----------------------------------+
2 rows in set (0.00 sec)
```

12.11.2.89.3 MySQL 兼容性

`SHOW GRANTS` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.89.4 另请参阅

- SHOW CREATE USER
- GRANT

12.11.2.90 SHOW INDEX [FROM|IN]

`SHOW INDEX [FROM|IN]` 语句是`SHOW INDEXES [FROM|IN]` 的别名。包含该语句提供了 MySQL 兼容性。

12.11.2.91 SHOW INDEXES [FROM|IN]

`SHOW INDEXES [FROM|IN]` 语句用于列出指定表上的索引。`SHOW INDEX [FROM | IN]` 和 `SHOW KEYS [FROM | IN]` 是该语句的别名。包含该语句提供了 MySQL 兼容性。

12.11.2.91.1 语法图

ShowIndexStmt:

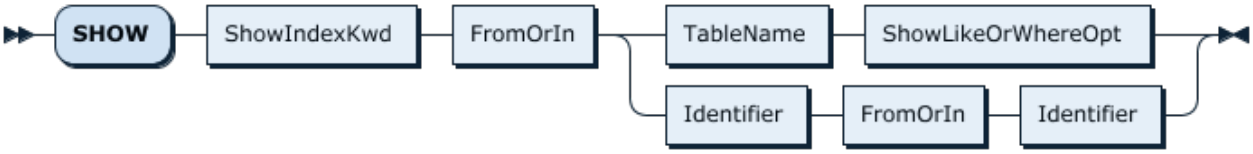


图 304: ShowIndexStmt

ShowIndexKwd:

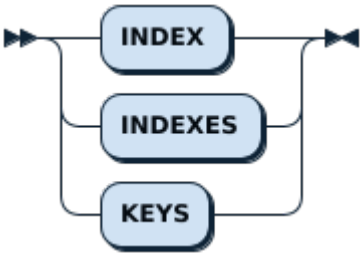


图 305: ShowIndexKwd

FromOrIn:

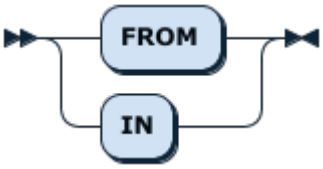


图 306: FromOrIn

TableName:

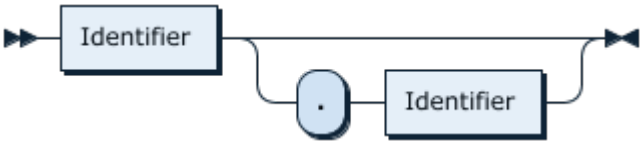


图 307: TableName

ShowLikeOrWhereOpt:

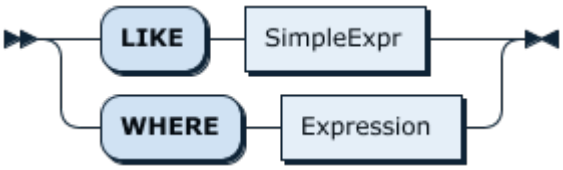


图 308: ShowLikeOrWhereOpt

12.11.2.91.2 示例

```
CREATE TABLE t1 (id int not null primary key AUTO_INCREMENT, col1 INT, INDEX(col1));
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
SHOW INDEXES FROM t1;
```

```
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+-----
    ↪
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part
    ↪  | Packed | Null | Index_type | Comment | Index_comment |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+-----
    ↪
| t1    |          0 | PRIMARY  |            1 | id          | A         |           0 |     NULL
    ↪  | NULL   |      | BTREE      |         |               |
| t1    |          1 | col1     |            1 | col1        | A         |           0 |     NULL
    ↪  | NULL   | YES  | BTREE      |         |               |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+-----
    ↪
2 rows in set (0.00 sec)
```

```
SHOW INDEX FROM t1;
```

```
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+-----
    ↪
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part
    ↪  | Packed | Null | Index_type | Comment | Index_comment |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+-----
    ↪
| t1    |          0 | PRIMARY  |            1 | id          | A         |           0 |     NULL
    ↪  | NULL   |      | BTREE      |         |               |
| t1    |          1 | col1     |            1 | col1        | A         |           0 |     NULL
    ↪  | NULL   | YES  | BTREE      |         |               |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+-----
    ↪
2 rows in set (0.00 sec)
```

```
SHOW KEYS FROM t1;
```

```
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+-----
    ↪
| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part
    ↪  | Packed | Null | Index_type | Comment | Index_comment |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+-----
    ↪
| t1    |          0 | PRIMARY  |            1 | id          | A         |           0 |     NULL
    ↪  | NULL   |      | BTREE      |         |               |
| t1    |          1 | col1     |            1 | col1        | A         |           0 |     NULL
    ↪  | NULL   | YES  | BTREE      |         |               |
+-------+------------+----------+--------------+-------------+-----------+-------------+----------+--------+-----
    ↪
2 rows in set (0.00 sec)
```

12.11.2.91.3 MySQL 兼容性

`SHOW INDEXES [FROM|IN]` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.91.4 另请参阅

- SHOW CREATE TABLE
- DROP INDEX
- CREATE INDEX

12.11.2.92 SHOW KEYS [FROM|IN]

`SHOW KEYS [FROM|IN]` 语句是`SHOW INDEXES [FROM|IN]` 语句的别名。包含该语句提供了 MySQL 兼容性。

12.11.2.93 SHOW MASTER STATUS

`SHOW MASTER STATUS` 语句用于显示集群当前最新的 TSO 信息。

12.11.2.93.1 示例

```
SHOW MASTER STATUS;
```

```
+-------------+--------------------+--------------+------------------+-------------------+
| File        | Position           | Binlog_Do_DB | Binlog_Ignore_DB | Executed_Gtid_Set |
+-------------+--------------------+--------------+------------------+-------------------+
| tidb-binlog | 416916363252072450 |              |                  |                   |
+-------------+--------------------+--------------+------------------+-------------------+
1 row in set (0.00 sec)
```

12.11.2.93.2 MySQL 兼容性

`SHOW MASTER STATUS` 语句与 MySQL 兼容，但是执行结果有差异，在 MySQL 中执行结果为 binlog 的位置信息，而在 TiDB 中为最新的 TSO 信息。

12.11.2.93.3 另请参阅

- SHOW PUMP STATUS
- SHOW DRAINER STATUS
- CHANGE PUMP STATUS
- CHANGE DRAINER STATUS

12.11.2.94 SHOW PLUGINS

`SHOW PLUGINS` 用于查看 TiDB 安装的插件，各个插件运行的状态以及插件版本信息。

12.11.2.94.1 语法图

ShowStmt:

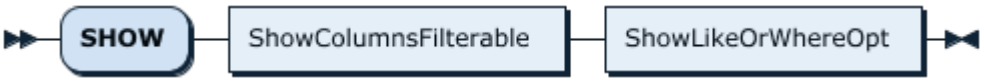


图 309: ShowStmt

ShowTargetFilterable:

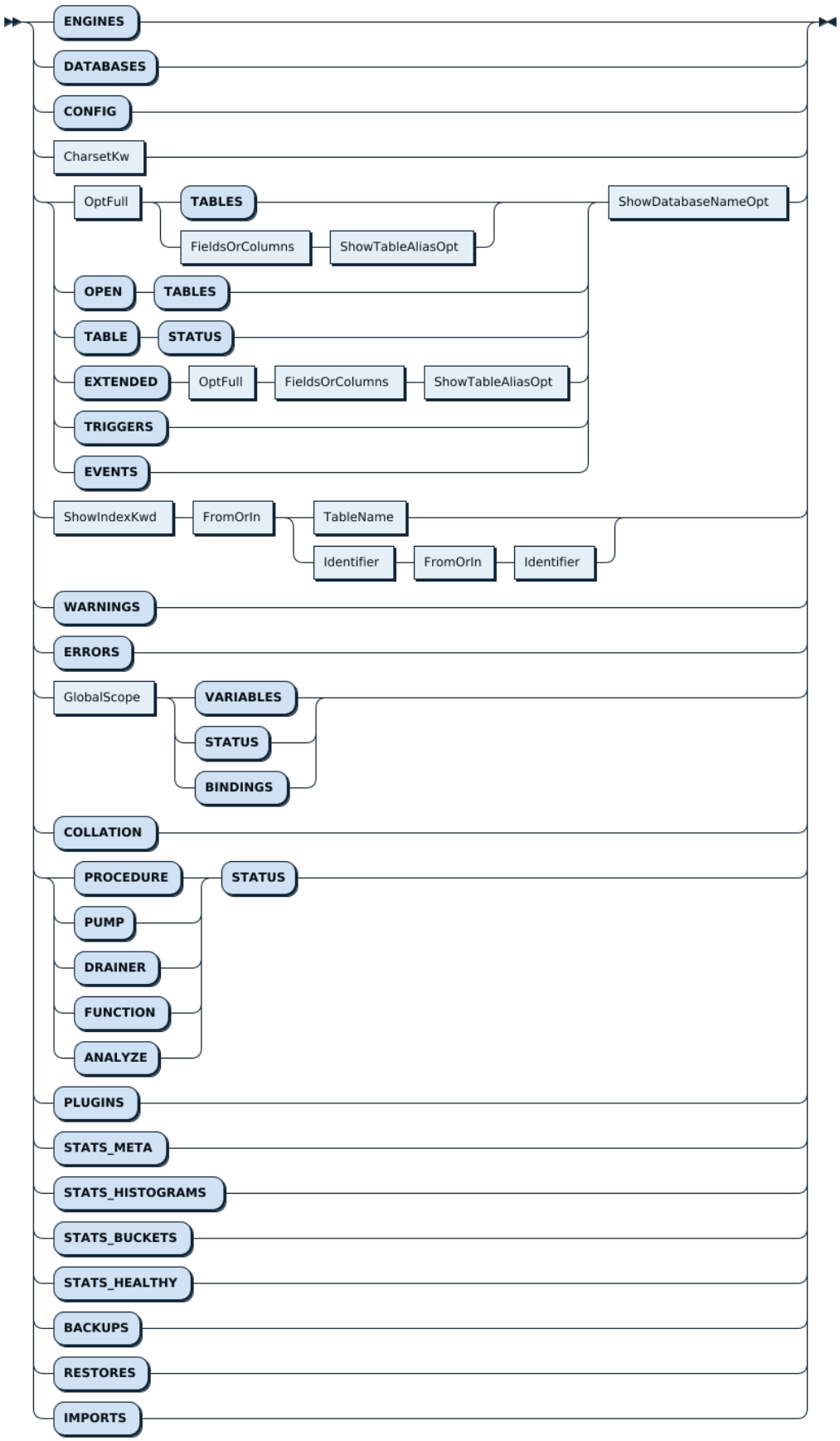


图 310: ShowTargetFilterable

12.11.2.94.2 示例

```
SHOW PLUGINS;
```

```
+-------+--------------+-------+----------------------------+---------+---------+
| Name  | Status       | Type  | Library                    | License | Version |
+-------+--------------+-------+----------------------------+---------+---------+
| audit | Ready-enable | Audit | /tmp/tidb/plugin/audit-1.so |         | 1       |
+-------+--------------+-------+----------------------------+---------+---------+
1 row in set (0.000 sec)
```

SHOW PLUGINS 也支持模糊匹配：

```
SHOW PLUGINS LIKE 'a%';
```

```
+-------+--------------+-------+----------------------------+---------+---------+
| Name  | Status       | Type  | Library                    | License | Version |
+-------+--------------+-------+----------------------------+---------+---------+
| audit | Ready-enable | Audit | /tmp/tidb/plugin/audit-1.so |         | 1       |
+-------+--------------+-------+----------------------------+---------+---------+
1 row in set (0.000 sec)
```

12.11.2.94.3 MySQL 兼容性

SHOW PLUGINS 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.95 SHOW PRIVILEGES

SHOW PRIVILEGES 语句用于显示 TiDB 中可分配权限的列表。此列表为静态列表，不反映当前用户的权限。

12.11.2.95.1 语法图

ShowStmt:

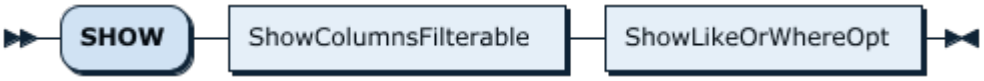


图 311: ShowStmt

12.11.2.95.2 示例

```
show privileges;
```

```
+-------------------------+---------------------------------------+-------------------------------------------------------------
    ↪
| Privilege               | Context                               | Comment
    ↪                                                 |
```

```
+-------------------------+---------------------------------------+--------------------------------
    ↪
| Alter                   | Tables                                | To alter the table
    ↪                                    |
| Alter                   | Tables                                | To alter the table
    ↪                                    |
| Alter routine           | Functions,Procedures                  | To alter or drop stored
    ↪ functions/procedures          |
| Create                  | Databases,Tables,Indexes              | To create new databases and
    ↪ tables                      |
| Create routine          | Databases                             | To use CREATE FUNCTION/
    ↪ PROCEDURE                       |
| Create temporary tables | Databases                             | To use CREATE TEMPORARY TABLE
    ↪                            |
| Create view             | Tables                                | To create new views
    ↪                                   |
| Create user             | Server Admin                          | To create new users
    ↪                                   |
| Delete                  | Tables                                | To delete existing rows
    ↪                                |
| Drop                    | Databases,Tables                      | To drop databases, tables,
    ↪ and views                   |
| Event                   | Server Admin                          | To create, alter, drop and
    ↪ execute events              |
| Execute                 | Functions,Procedures                  | To execute stored routines
    ↪                             |
| File                    | File access on server                 | To read and write files on
    ↪ the server                  |
| Grant option            | Databases,Tables,Functions,Procedures | To give to other users those
    ↪ privileges you possess   |
| Index                   | Tables                                | To create or drop indexes
    ↪                              |
| Insert                  | Tables                                | To insert data into tables
    ↪                             |
| Lock tables             | Databases                             | To use LOCK TABLES (together
    ↪ with SELECT privilege)   |
| Process                 | Server Admin                          | To view the plain text of
    ↪ currently executing queries |
| Proxy                   | Server Admin                          | To make proxy user possible
    ↪                            |
| References              | Databases,Tables                      | To have references on tables
    ↪                           |
| Reload                  | Server Admin                          | To reload or refresh tables,
    ↪ logs and privileges      |
```

```
| Replication client      | Server Admin                          | To ask where the slave or
    ↪ master servers are          |
| Replication slave       | Server Admin                          | To read binary log events
    ↪ from the master             |
| Select                  | Tables                                | To retrieve rows from table
    ↪                           |
| Show databases          | Server Admin                          | To see all databases with
    ↪ SHOW DATABASES              |
| Show view               | Tables                                | To see views with SHOW CREATE
    ↪  VIEW                     |
| Shutdown                | Server Admin                          | To shut down the server
    ↪                               |
| Super                   | Server Admin                          | To use KILL thread, SET
    ↪ GLOBAL, CHANGE MASTER, etc.   |
| Trigger                 | Tables                                | To use triggers
    ↪                                       |
| Create tablespace       | Server Admin                          | To create/alter/drop
    ↪ tablespaces                       |
| Update                  | Tables                                | To update existing rows
    ↪                               |
| Usage                   | Server Admin                          | No privileges - allow connect
    ↪  only                     |
+-------------------------+---------------------------------------+-------------------------------------------
    ↪
32 rows in set (0.00 sec)
```

12.11.2.95.3 MySQL 兼容性

`SHOW PRIVILEGES` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.95.4 另请参阅

- SHOW GRANTS
- GRAMMAR_ERROR

12.11.2.96 SHOW [FULL] PROCESSLIST

`SHOW [FULL] PROCESSLIST` 语句列出连接到相同 TiDB 服务器的当前会话。`Info` 列包含查询文本，除非指定可选关键字 `FULL`，否则文本会被截断。

12.11.2.96.1 语法图

ShowProcesslistStmt:

图 312: ShowProcesslistStmt

OptFull:

图 313: OptFull

12.11.2.96.2 示例

```
SHOW PROCESSLIST;
```

```
+------+------+-----------+------+---------+------+-------+------------------+
| Id   | User | Host      | db   | Command | Time | State | Info             |
+------+------+-----------+------+---------+------+-------+------------------+
|    1 | root | 127.0.0.1 | test | Query   |    0 | 2     | SHOW PROCESSLIST |
|    2 | root | 127.0.0.1 |      | Sleep   |    4 | 2     |                  |
+------+------+-----------+------+---------+------+-------+------------------+
2 rows in set (0.00 sec)
```

12.11.2.96.3 MySQL 兼容性

- TiDB 中的 State 列是非描述性的。在 TiDB 中，将状态表示为单个值更复杂，因为查询是并行执行的，而且每个 Go 线程在任一时刻都有不同的状态。

12.11.2.96.4 另请参阅

- KILL [TIDB]

12.11.2.97 SHOW PROFILES

SHOW PROFILES 语句目前只会返回空结果。

12.11.2.97.1 语法图

ShowStmt:

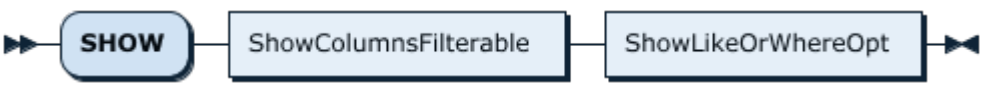


图 314: ShowStmt

12.11.2.97.2 示例

```
SHOW PROFILES
```

```
Empty set (0.00 sec)
```

12.11.2.97.3 MySQL 兼容性

该语句仅与 MySQL 兼容，无其他作用。执行 SHOW PROFILES 始终返回空结果。

12.11.2.98 SHOW PUMP STATUS

SHOW PUMP STATUS 语句用于显示集群中所有 Pump 的状态信息。

12.11.2.98.1 示例

```
SHOW PUMP STATUS;
```

```
+--------|----------------|--------|--------------------|---------------------|
| NodeID |     Address    | State  |   Max_Commit_Ts    |    Update_Time      |
+--------|----------------|--------|--------------------|---------------------|
| pump1  | 127.0.0.1:8250 | Online | 408553768673342237 | 2019-05-01 00:00:01 |
+--------|----------------|--------|--------------------|---------------------|
| pump2  | 127.0.0.2:8250 | Online | 408553768673342335 | 2019-05-01 00:00:02 |
+--------|----------------|--------|--------------------|---------------------|
2 rows in set (0.00 sec)
```

12.11.2.98.2 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

12.11.2.98.3 另请参阅

- SHOW DRAINER STATUS
- CHANGE PUMP STATUS
- CHANGE DRAINER STATUS

12.11.2.99 SHOW SCHEMAS

SHOW SCHEMAS 语句是SHOW DATABASES 的别名。包含该语句提供了 MySQL 兼容性。

12.11.2.100 SHOW STATS_HEALTHY

SHOW STATS_HEALTHY 语句可以预估统计信息的准确度，也就是健康度。健康度低的表可能会生成次优查询执行计划。

可以通过执行 ANALYZE 表命令来改善表的健康度。当表的健康度下降到低于tidb_auto_analyze_ratio 时，则会自动执行 ANALYZE 命令。

12.11.2.100.1 语法图

ShowStmt

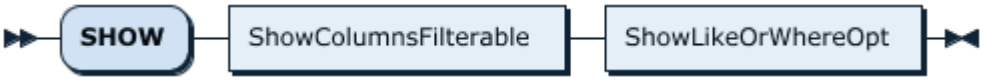


图 315: ShowStmt

ShowTargetFilterable

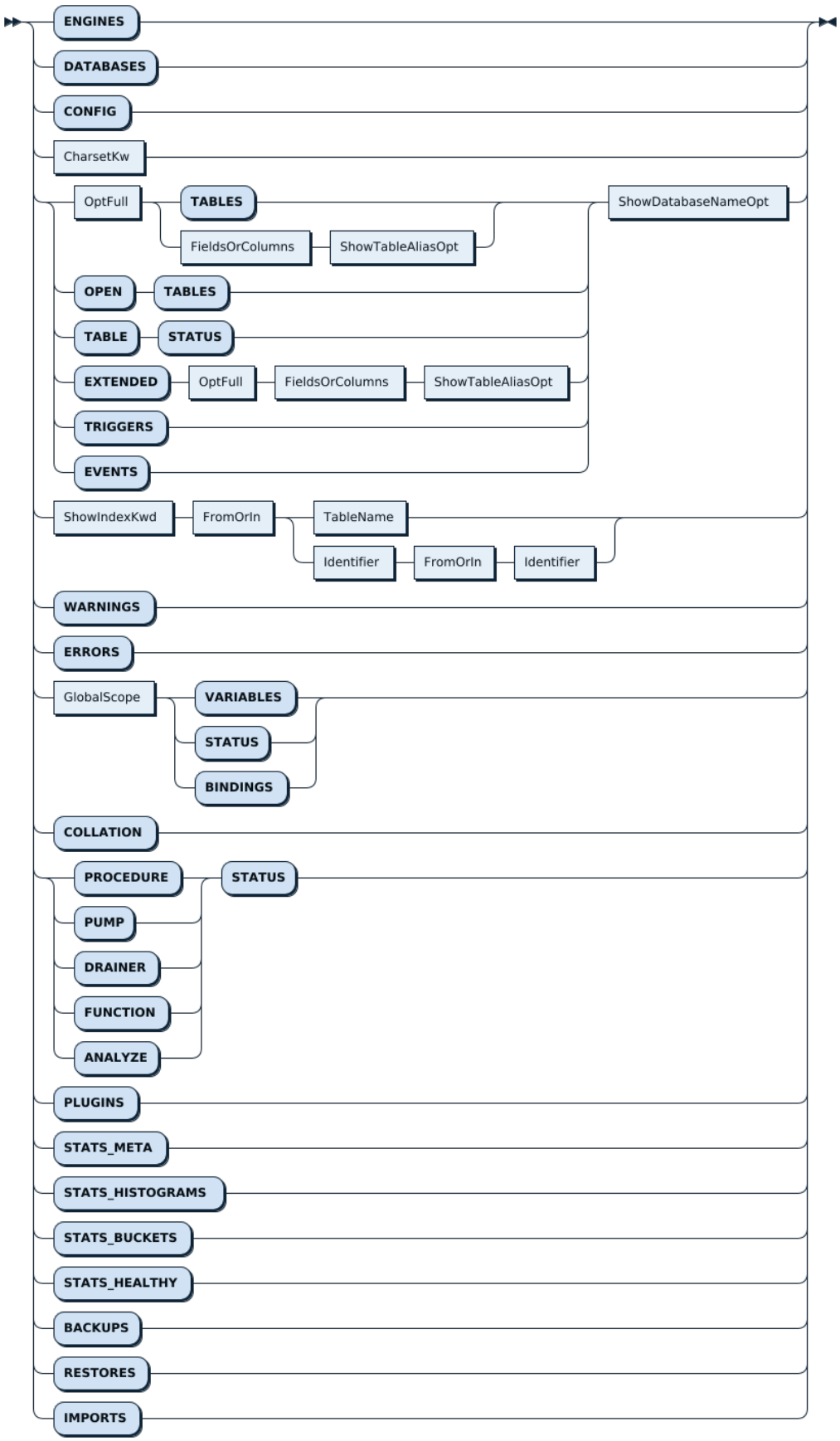


图 316: ShowTargetFilterable

ShowLikeOrWhereOpt

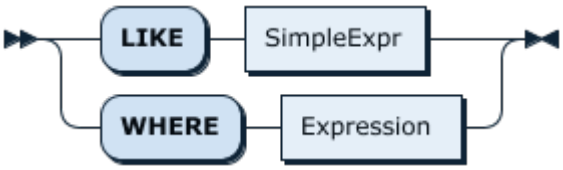


图 317: ShowLikeOrWhereOpt

12.11.2.100.2 示例

加载示例数据并运行 ANALYZE 命令：

```
CREATE TABLE t1 (
 id INT NOT NULL PRIMARY KEY auto_increment,
 b INT NOT NULL,
 pad VARBINARY(255),
 INDEX(b)
);
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM dual;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(255) FROM t1 a JOIN t1 b JOIN t1 c
    ↪ LIMIT 100000;
SELECT SLEEP(1);
ANALYZE TABLE t1;
SHOW STATS_HEALTHY; # should be 100% healthy
```

```
SHOW STATS_HEALTHY;
```

```
+---------+------------+----------------+---------+
| Db_name | Table_name | Partition_name | Healthy |
+---------+------------+----------------+---------+
| test    | t1         |                |     100 |
+---------+------------+----------------+---------+
1 row in set (0.00 sec)
```

执行批量更新来删除大约 30% 的记录，然后检查统计信息的健康度：

```
DELETE FROM t1 WHERE id BETWEEN 101010 AND 201010; # delete about 30% of records
SHOW STATS_HEALTHY;
```

```
SHOW STATS_HEALTHY;
+---------+------------+----------------+---------+
| Db_name | Table_name | Partition_name | Healthy |
+---------+------------+----------------+---------+
| test    | t1         |                |      50 |
+---------+------------+----------------+---------+
1 row in set (0.00 sec)
```

12.11.2.100.3 MySQL 兼容性

`SHOW STATS_HEALTHY` 语句是 TiDB 对 MySQL 语法的扩展。

12.11.2.100.4 另请参阅

- ANALYZE
- 统计信息简介

12.11.2.101 SHOW STATS_HISTOGRAMS

你可以使用 `SHOW STATS_HISTOGRAMS` 语句查看统计信息中直方图的相关信息。

12.11.2.101.1 语法图

ShowStmt

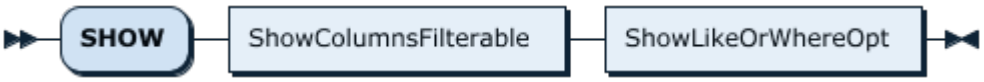


图 318: ShowStmt

ShowTargetFiltertable

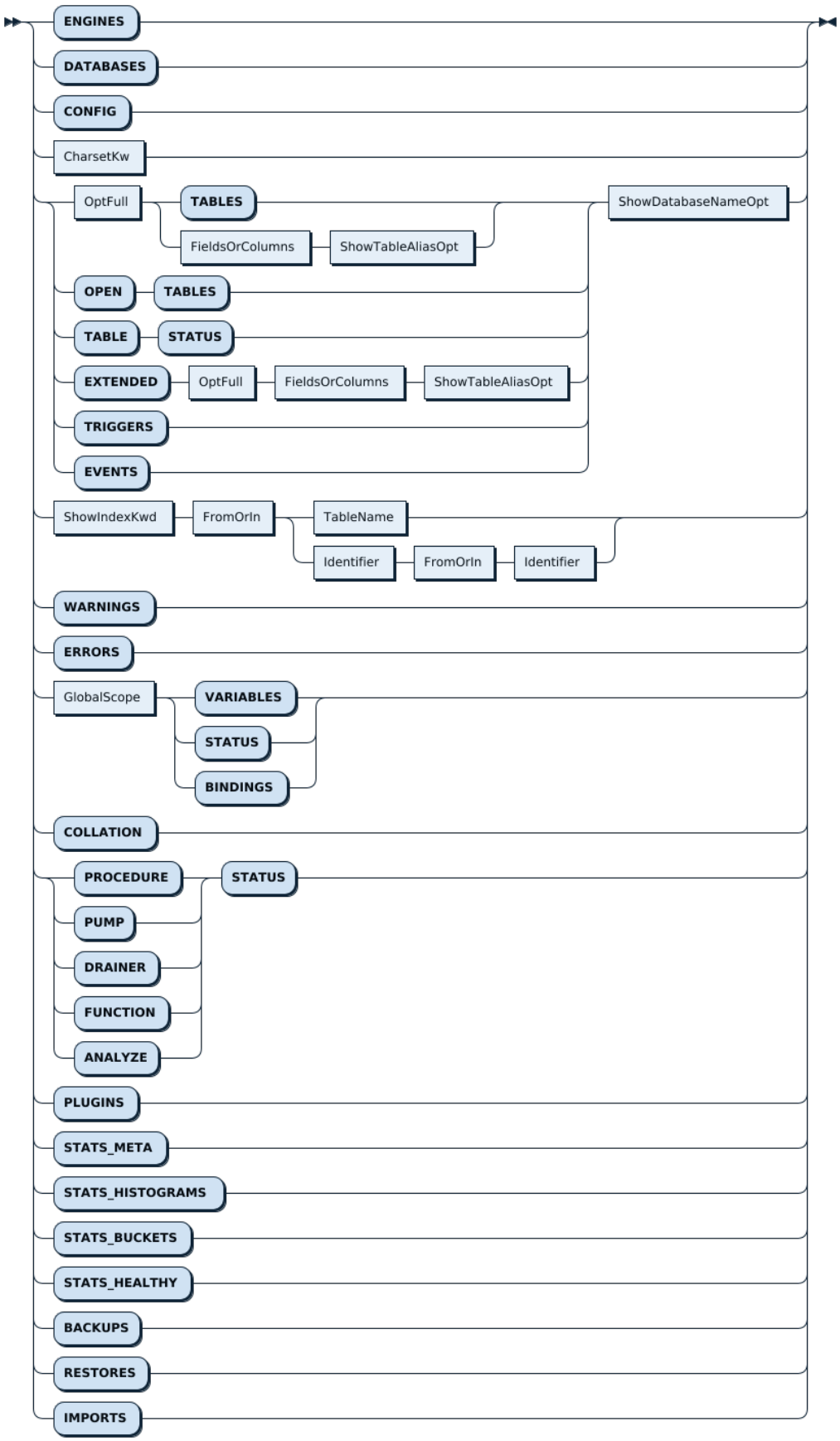


图 319: ShowTargetFilterable

ShowLikeOrWhereOpt

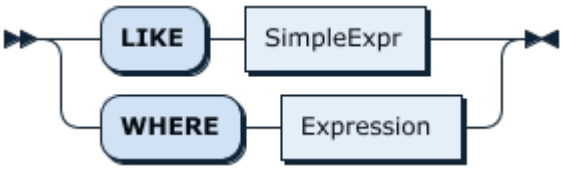


图 320: ShowLikeOrWhereOpt

12.11.2.101.2 示例

```
show stats_histograms;
```

```
+--
    ↪ -------+------------+----------------+-------------+----------+--------------------+-----------------+---
    ↪
| Db_name | Table_name | Partition_name | Column_name | Is_index | Update_time         |
    ↪ Distinct_count | Null_count | Avg_col_size | Correlation |
+--
    ↪ -------+------------+----------------+-------------+----------+--------------------+-----------------+---
    ↪
| test    | t          |                | a           |        0 | 2020-05-25 19:20:00 |
    ↪              7 |          0 |              1 |           1 |
| test    | t2         |                | a           |        0 | 2020-05-25 19:20:01 |
    ↪              6 |          0 |              8 |           0 |
| test    | t2         |                | b           |        0 | 2020-05-25 19:20:01 |
    ↪              6 |          0 |           1.67 |           1 |
+--
    ↪ -------+------------+----------------+-------------+----------+--------------------+-----------------+---
    ↪
3 rows in set (0.00 sec)
```

```
show stats_histograms where table_name = 't2';
```

```
+--
    ↪ -------+------------+----------------+-------------+----------+--------------------+-----------------+---
    ↪
| Db_name | Table_name | Partition_name | Column_name | Is_index | Update_time         |
    ↪ Distinct_count | Null_count | Avg_col_size | Correlation |
+--
    ↪ -------+------------+----------------+-------------+----------+--------------------+-----------------+---
    ↪
| test    | t2         |                | b           |        0 | 2020-05-25 19:20:01 |
    ↪              6 |          0 |           1.67 |           1 |
| test    | t2         |                | a           |        0 | 2020-05-25 19:20:01 |
    ↪              6 |          0 |              8 |           0 |
```

```
+--
    ↪ -------+------------+----------------+-------------+----------+---------------------+-----------+---
    ↪
2 rows in set (0.00 sec)
```

12.11.2.101.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

12.11.2.101.4 另请参阅

- ANALYZE
- 统计信息介绍

12.11.2.102 SHOW STATS_META

你可以通过 `SHOW STATS_META` 来查看表的总行数以及修改的行数等信息，可以通过 ShowLikeOrWhere 来筛选需要的信息。

目前 `SHOW STATS_META` 会输出 6 列，具体如下：

语法元素	说明
db_name	数据库名
table_name	表名
partition_name	分区名
update_time	更新时间
modify_count	修改的行数
row_count	总行数

> **注意：**
>
> 在 TiDB 根据 DML 语句自动更新总行数以及修改的行数时，`update_time` 也会被更新，因此并不能认为 `update_time` 是最近一次发生 Analyze 的时间。

12.11.2.102.1 语法图

ShowStmt

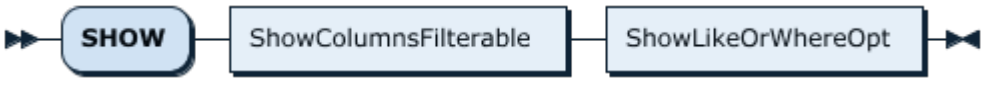


图 321: ShowStmt

ShowTargetFiltertable

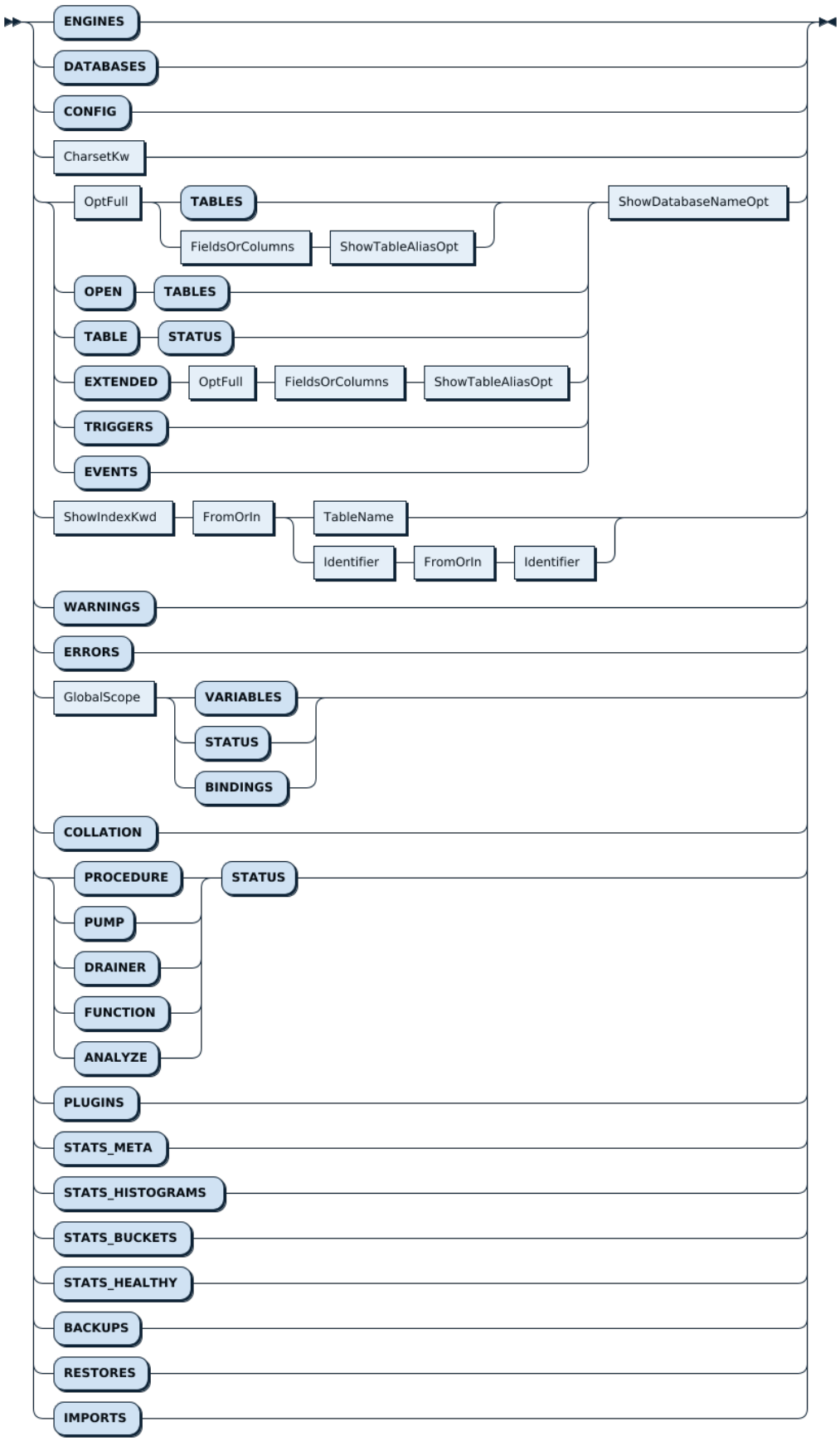


图 322: ShowTargetFilterable

ShowLikeOrWhereOpt

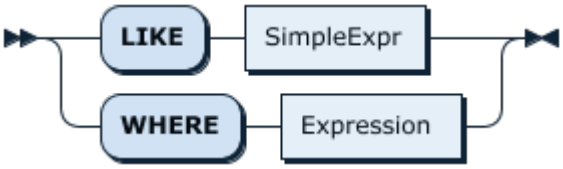


图 323: ShowLikeOrWhereOpt

12.11.2.102.2 示例

```
show stats_meta;
```

```
+---------+------------+----------------+---------------------+--------------+-----------+
| Db_name | Table_name | Partition_name | Update_time         | Modify_count | Row_count |
+---------+------------+----------------+---------------------+--------------+-----------+
| test    | t0         |                | 2020-05-15 16:58:00 |            0 |         0 |
| test    | t1         |                | 2020-05-15 16:58:04 |            0 |         0 |
| test    | t2         |                | 2020-05-15 16:58:11 |            0 |         0 |
| test    | s          |                | 2020-05-22 19:46:43 |            0 |         0 |
| test    | t          |                | 2020-05-25 12:04:21 |            0 |         0 |
+---------+------------+----------------+---------------------+--------------+-----------+
5 rows in set (0.00 sec)
```

```
show stats_meta where table_name = 't2';
```

```
+---------+------------+----------------+---------------------+--------------+-----------+
| Db_name | Table_name | Partition_name | Update_time         | Modify_count | Row_count |
+---------+------------+----------------+---------------------+--------------+-----------+
| test    | t2         |                | 2020-05-15 16:58:11 |            0 |         0 |
+---------+------------+----------------+---------------------+--------------+-----------+
1 row in set (0.00 sec)
```

12.11.2.102.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

12.11.2.102.4 另请参阅

- ANALYZE
- 统计信息介绍

12.11.2.103 SHOW [GLOBAL|SESSION] STATUS

`SHOW [GLOBAL|SESSION] STATUS` 语句用于提供 MySQL 兼容性，对 TiDB 没有作用。因为 TiDB 使用 Prometheus 和 Grafana 而非 `SHOW STATUS` 来进行集中度量收集。

12.11.2.103.1 语法图

ShowStmt:

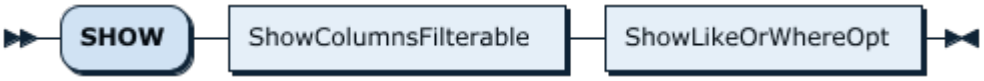


图 324: ShowStmt

ShowTargetFilterable:

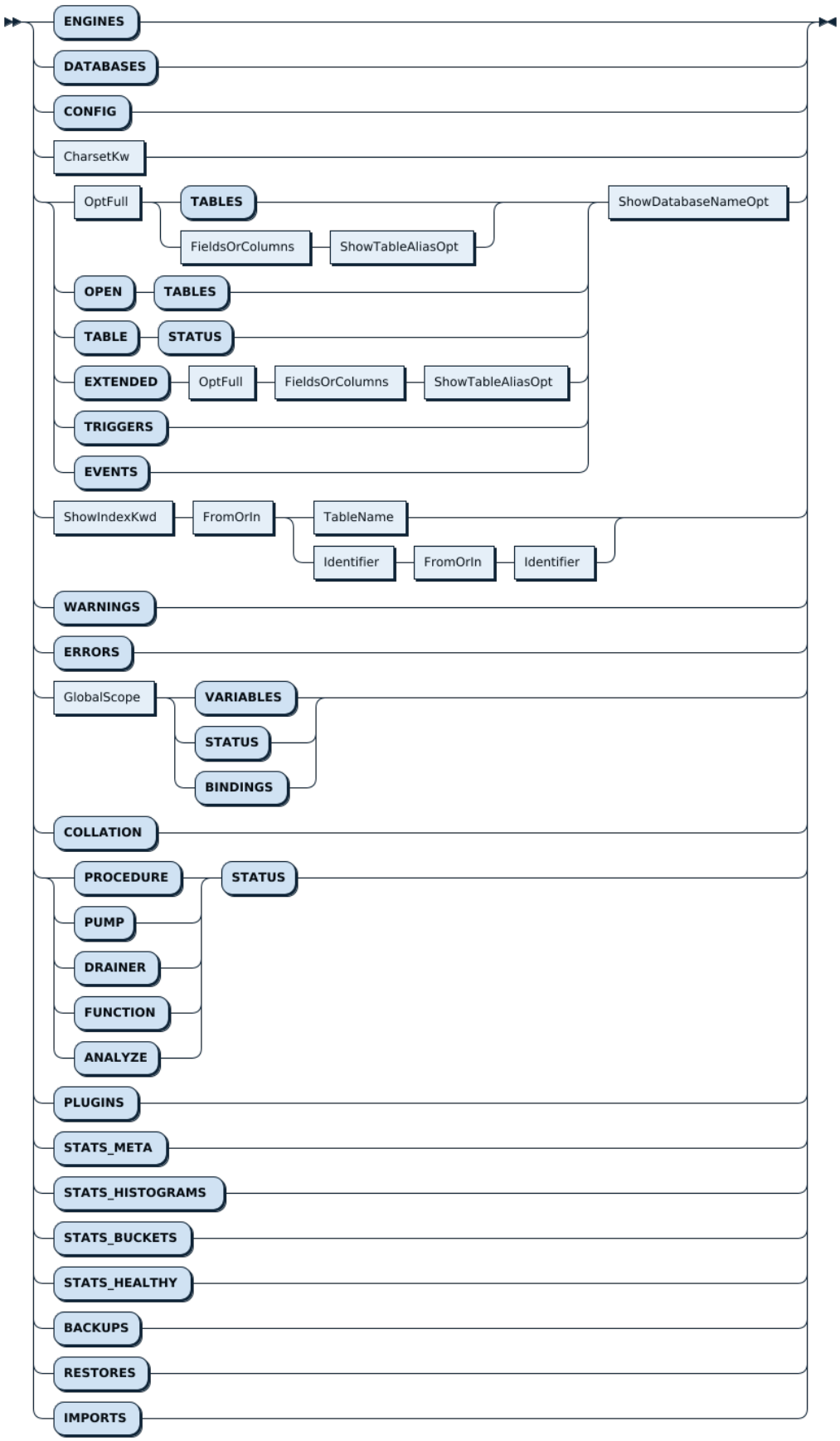


图 325: ShowTargetFilterable

GlobalScope:

图 326: GlobalScope

12.11.2.103.2 示例

```
show status;
```

```
+--------------------+--------------------------------------+
| Variable_name      | Value                                |
+--------------------+--------------------------------------+
| Ssl_cipher_list    |                                      |
| server_id          | 93e2e07d-6bb4-4a1b-90b7-e035fae154fe |
| ddl_schema_version | 141                                  |
| Ssl_verify_mode    | 0                                    |
| Ssl_version        |                                      |
| Ssl_cipher         |                                      |
+--------------------+--------------------------------------+
6 rows in set (0.01 sec)
```

```
show global status;
```

```
+--------------------+--------------------------------------+
| Variable_name      | Value                                |
+--------------------+--------------------------------------+
| Ssl_cipher         |                                      |
| Ssl_cipher_list    |                                      |
| Ssl_verify_mode    | 0                                    |
| Ssl_version        |                                      |
| server_id          | 93e2e07d-6bb4-4a1b-90b7-e035fae154fe |
| ddl_schema_version | 141                                  |
+--------------------+--------------------------------------+
6 rows in set (0.00 sec)
```

12.11.2.103.3 MySQL 兼容性

`SHOW [GLOBAL|SESSION] STATUS` 语句仅用于提供 MySQL 兼容性。

12.11.2.103.4 另请参阅

- FLUSH STATUS

12.11.2.104 SHOW TABLE NEXT_ROW_ID

SHOW TABLE NEXT_ROW_ID 语句用于显示用表中某些特殊列的详情，主要包含以下几种类型：

- TiDB 创建的 AUTO_INCREMENT 类型列，即 _tidb_rowid 列
- 用户创建的 AUTO_INCREMENT 类型列
- 用户创建的AUTO_RANDOM 类型列
- 用户创建的SEQUENCE 对象信息

12.11.2.104.1 语法图

ShowTableNextRowIDStmt:

图 327: ShowTableNextRowIDStmt

TableName:

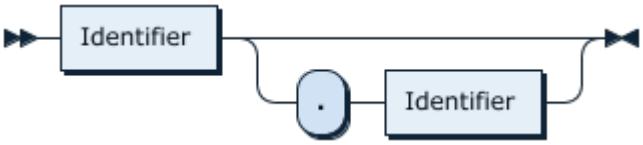


图 328: TableName

12.11.2.104.2 示例

对于新建的表，由于没有任何的 Row ID 分配，NEXT_GLOBAL_ROW_ID 值为 1

```
create table t(a int);
Query OK, 0 rows affected (0.06 sec)
```

```
show table t next_row_id;
+---------+------------+-------------+--------------------+
| DB_NAME | TABLE_NAME | COLUMN_NAME | NEXT_GLOBAL_ROW_ID |
+---------+------------+-------------+--------------------+
| test    | t          | _tidb_rowid |                  1 |
+---------+------------+-------------+--------------------+
1 row in set (0.00 sec)
```

表中写入了数据，负责写入的 TiDB Server 一次性向存储层请求了 30000 个 ID 缓存起来，NEXT_GLOBAL_ROW_ID 值为 30001

```
insert into t values (), (), ();
Query OK, 3 rows affected (0.02 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
show table t next_row_id;
+---------+------------+-------------+--------------------+
| DB_NAME | TABLE_NAME | COLUMN_NAME | NEXT_GLOBAL_ROW_ID |
+---------+------------+-------------+--------------------+
| test    | t          | _tidb_rowid |              30001 |
+---------+------------+-------------+--------------------+
1 row in set (0.00 sec)
```

12.11.2.104.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

12.11.2.104.4 另请参阅

- CREATE TABLE
- AUTO_RANDOM
- CREATE_SEQUENCE

12.11.2.105 SHOW TABLE REGIONS

SHOW TABLE REGIONS 语句用于显示 TiDB 中某个表的 Region 信息。

12.11.2.105.1 语法

```
SHOW TABLE [table_name] REGIONS [WhereClauseOptional];

SHOW TABLE [table_name] INDEX [index_name] REGIONS [WhereClauseOptional];
```

语法图

ShowTableRegionStmt:

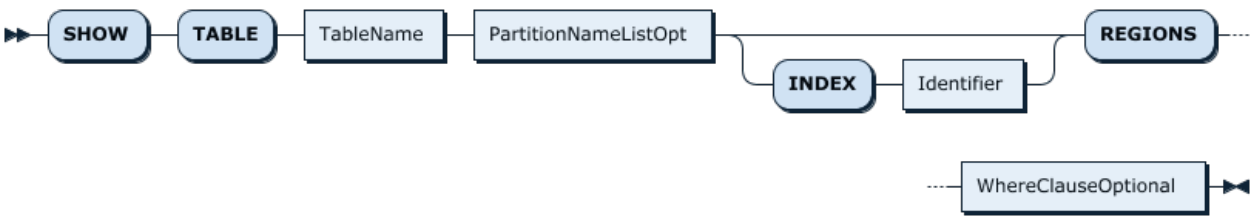


图 329: ShowTableRegionStmt

TableName:

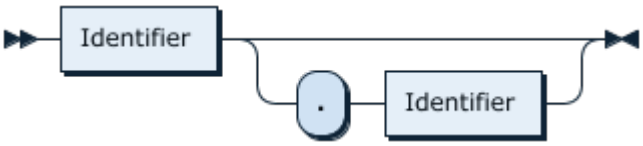


图 330: TableName

PartitionNameListOpt:

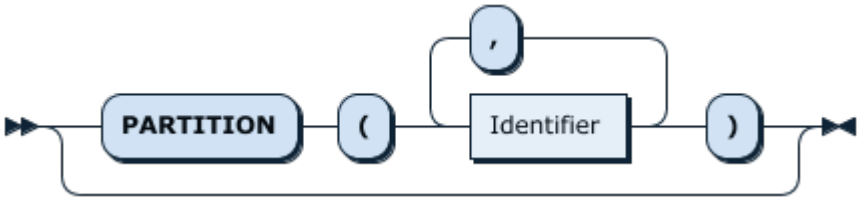


图 331: PartitionNameListOpt

WhereClauseOptional:

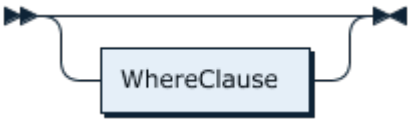


图 332: WhereClauseOptional

WhereClause:

图 333: WhereClause

`SHOW TABLE REGIONS` 会返回如下列：

- `REGION_ID`：Region 的 ID。
- `START_KEY`：Region 的 Start key。
- `END_KEY`：Region 的 End key。
- `LEADER_ID`：Region 的 Leader ID。
- `LEADER_STORE_ID`：Region leader 所在的 store (TiKV) ID。
- `PEERS`：Region 所有副本的 ID。
- `SCATTERING`：Region 是否正在调度中。1 表示正在调度。
- `WRITTEN_BYTES`：估算的 Region 在 1 个心跳周期内的写入数据量大小，单位是 byte。
- `READ_BYTES`：估算的 Region 在 1 个心跳周期内的读数据量大小，单位是 byte。
- `APPROXIMATE_SIZE(MB)`：估算的 Region 的数据量大小，单位是 MB。

- `APPROXIMATE_KEYS`：估算的 Region 内 Key 的个数。

> **注意：**
>
> `WRITTEN_BYTES`，`READ_BYTES`，`APPROXIMATE_SIZE(MB)`，`APPROXIMATE_KEYS` 的值是 PD 根据 Region 的心跳汇报信息统计，估算出来的数据，所以不是精确的数据。

12.11.2.105.2 示例

创建一个示例表，并在若干 Region 中填充足够的数据量：

```sql
CREATE TABLE t1 (
 id INT NOT NULL PRIMARY KEY auto_increment,
 b INT NOT NULL,
 pad1 VARBINARY(1024),
 pad2 VARBINARY(1024),
 pad3 VARBINARY(1024)
);
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(1024), RANDOM_BYTES(1024),
    ↪ RANDOM_BYTES(1024) FROM dual;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(1024), RANDOM_BYTES(1024),
    ↪ RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(1024), RANDOM_BYTES(1024),
    ↪ RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(1024), RANDOM_BYTES(1024),
    ↪ RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(1024), RANDOM_BYTES(1024),
    ↪ RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(1024), RANDOM_BYTES(1024),
    ↪ RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(1024), RANDOM_BYTES(1024),
    ↪ RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(1024), RANDOM_BYTES(1024),
    ↪ RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(1024), RANDOM_BYTES(1024),
    ↪ RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(1024), RANDOM_BYTES(1024),
    ↪ RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(1024), RANDOM_BYTES(1024),
    ↪ RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
INSERT INTO t1 SELECT NULL, FLOOR(RAND()*1000), RANDOM_BYTES(1024), RANDOM_BYTES(1024),
    ↪ RANDOM_BYTES(1024) FROM t1 a JOIN t1 b JOIN t1 c LIMIT 10000;
SELECT SLEEP(5);
SHOW TABLE t1 REGIONS;
```

结果显示示例表被切分成多个 Regions。REGION_ID、START_KEY 和 END_KEY 可能不完全匹配：

```
SHOW TABLE t1 REGIONS;
+--
    ↪ ---------+--------------+--------------+-----------+-----------------+-------+------------+--------------
    ↪
| REGION_ID | START_KEY     | END_KEY       | LEADER_ID | LEADER_STORE_ID | PEERS | SCATTERING |
    ↪ WRITTEN_BYTES | READ_BYTES | APPROXIMATE_SIZE(MB) | APPROXIMATE_KEYS |
+--
    ↪ ---------+--------------+--------------+-----------+-----------------+-------+------------+--------------
    ↪
|        94 | t_75_         | t_75_r_31717 |        95 |               1 | 95    |          0 |
    ↪             0 |           0 |                  112 |           207465 |
|        96 | t_75_r_31717 | t_75_r_63434 |        97 |               1 | 97    |          0 |
    ↪             0 |           0 |                   97 |                0 |
|         2 | t_75_r_63434 |              |         3 |               1 | 3     |          0 |
    ↪      269323514 |   66346110 |                  245 |           162020 |
+--
    ↪ ---------+--------------+--------------+-----------+-----------------+-------+------------+--------------
    ↪
3 rows in set (0.00 sec)
```

解释：

上面 START_KEY 列的值 t_75_r_31717 和 END_KEY 列的值 t_75_r_63434 表示主键在 31717 和 63434 之间的数据存储在该 Region 中。t_75_ 是前缀，表示这是表格 (t) 的 Region，75 是表格的内部 ID。若 START_KEY 或 END_KEY 的一对键值为空，分别表示负无穷大或正无穷大。

TiDB 会根据需要自动重新平衡 Regions。建议使用 SPLIT TABLE REGION 语句手动进行平衡：

```
SPLIT TABLE t1 BETWEEN (31717) AND (63434) REGIONS 2;
```

```
+--------------------+----------------------+
| TOTAL_SPLIT_REGION | SCATTER_FINISH_RATIO |
+--------------------+----------------------+
|                  1 |                    1 |
+--------------------+----------------------+
1 row in set (42.34 sec)
SHOW TABLE t1 REGIONS;
+-----------+--------------+--------------+-----------+-----------------+-------+------------+---------------+---
    ↪
| REGION_ID | START_KEY     | END_KEY       | LEADER_ID | LEADER_STORE_ID | PEERS | SCATTERING |
    ↪ WRITTEN_BYTES | READ_BYTES | APPROXIMATE_SIZE(MB) | APPROXIMATE_KEYS |
+-----------+--------------+--------------+-----------+-----------------+-------+------------+---------------+---
    ↪
|        94 | t_75_         | t_75_r_31717 |        95 |               1 | 95    |          0 |
    ↪             0 |           0 |                  112 |           207465 |
```

```
|        98 | t_75_r_31717 | t_75_r_47575 |        99 |                1 | 99    |          0 |
    ↪          1325 |           0 |                    53 |            12052 |
|        96 | t_75_r_47575 | t_75_r_63434 |        97 |                1 | 97    |          0 |
    ↪          1526 |           0 |                    48 |                0 |
|         2 | t_75_r_63434 |              |         3 |                1 | 3     |          0 |
    ↪             0 |    55752049 |                    60 |                0 |
+-----------+--------------+--------------+-----------+------------------+-------+------------+---------------+---
    ↪
4 rows in set (0.00 sec)
```

上面的输出结果显示 Region 96 被切分，并创建一个新的 Region 98。切分操作不会影响表中的其他 Region。输出结果同样证实：

- TOTAL_SPLIT_REGION 表示新切的 Region 数量。以上示例新切了 1 个 Region。
- SCATTER_FINISH_RATIO 表示新切的 Region 的打散成功率，1.0 表示都已经打散了。

更详细的示例如下：

```
show table t regions;
```

```
+-----------+--------------+--------------+-----------+-----------------+---------------+------------+-----------
    ↪
| REGION_ID | START_KEY    | END_KEY      | LEADER_ID | LEADER_STORE_ID | PEERS         |
    ↪ SCATTERING | WRITTEN_BYTES | READ_BYTES | APPROXIMATE_SIZE(MB) | APPROXIMATE_KEYS |
+-----------+--------------+--------------+-----------+-----------------+---------------+------------+-----------
    ↪
| 102       | t_43_r       | t_43_r_20000 | 118       | 7               | 105, 118, 119 | 0
    ↪          | 0             | 0          | 1                    | 0                |
| 106       | t_43_r_20000 | t_43_r_40000 | 120       | 7               | 107, 108, 120 | 0
    ↪          | 23            | 0          | 1                    | 0                |
| 110       | t_43_r_40000 | t_43_r_60000 | 112       | 9               | 112, 113, 121 | 0
    ↪          | 0             | 0          | 1                    | 0                |
| 114       | t_43_r_60000 | t_43_r_80000 | 122       | 7               | 115, 122, 123 | 0
    ↪          | 35            | 0          | 1                    | 0                |
| 3         | t_43_r_80000 |              | 93        | 8               | 5, 73, 93     | 0
    ↪          | 0             | 0          | 1                    | 0                |
| 98        | t_43_        | t_43_r       | 99        | 1               | 99, 100, 101  | 0
    ↪          | 0             | 0          | 1                    | 0                |
+-----------+--------------+--------------+-----------+-----------------+---------------+------------+-----------
    ↪
6 rows in set
```

解释：

- Region 102 的 START_KEY 和 END_KEY 中，t_43 是表数据前缀和 table ID，_r 是表 t record 数据的前缀，索引数据的前缀是 _i，所以 Region 102 的 START_KEY 和 END_KEY 表示用来存储 [-inf, 20000) 之前的 record 数据。其他 Region (106, 110, 114, 3) 的存储范围依次类推。

- Region 98 用来存储索引数据存储。表 t 索引数据的起始 key 是 t_43_i，处于 Region 98 的存储范围内。

查看表 t 在 store 1 上的 region，用 where 条件过滤。

```
show table t regions where leader_store_id =1;
```

```
+-----------+-----------+---------+-----------+----------------+-------------+------------+---------------+----
    ↪
| REGION_ID | START_KEY | END_KEY | LEADER_ID | LEADER_STORE_ID | PEERS        | SCATTERING |
    ↪ WRITTEN_BYTES | READ_BYTES | APPROXIMATE_SIZE(MB) | APPROXIMATE_KEYS |
+-----------+-----------+---------+-----------+----------------+-------------+------------+---------------+----
    ↪
| 98        | t_43_     | t_43_r  | 99        | 1              | 99, 100, 101 | 0          | 0
    ↪             | 0          | 1                    | 0               |
+-----------+-----------+---------+-----------+----------------+-------------+------------+---------------+----
    ↪
```

用 SPLIT TABLE REGION 语法切分索引数据的 Region，下面语法把表 t 的索引 name 数据在 [a,z] 范围内切分成 2 个 Region。

```
split table t index name between ("a") and ("z") regions 2;
```

```
+--------------------+----------------------+
| TOTAL_SPLIT_REGION | SCATTER_FINISH_RATIO |
+--------------------+----------------------+
| 2                  | 1.0                  |
+--------------------+----------------------+
1 row in set
```

现在表 t 一共有 7 个 Region，其中 5 个 Region (102, 106, 110, 114, 3) 用来存表 t 的 record 数据，另外 2 个 Region (135, 98) 用来存 name 索引的数据。

```
show table t regions;
```

```
+-----------+----------------------------+----------------------------+-----------+----------------+----------
    ↪
| REGION_ID | START_KEY                  | END_KEY                    | LEADER_ID |
    ↪ LEADER_STORE_ID | PEERS         | SCATTERING | WRITTEN_BYTES | READ_BYTES |
    ↪ APPROXIMATE_SIZE(MB) | APPROXIMATE_KEYS |
+-----------+----------------------------+----------------------------+-----------+----------------+----------
    ↪
| 102       | t_43_r                     | t_43_r_20000               | 118       | 7
    ↪               | 105, 118, 119 | 0          | 0             | 0          | 1
    ↪                   | 0               |
| 106       | t_43_r_20000               | t_43_r_40000               | 120       | 7
    ↪               | 108, 120, 126 | 0          | 0             | 0          | 1
    ↪                   | 0               |
```

```
| 110       | t_43_r_40000                | t_43_r_60000                | 112       | 9
    ↪                | 112, 113, 121 | 0          | 0             | 0          | 1
    ↪                    | 0               |
| 114       | t_43_r_60000                | t_43_r_80000                | 122       | 7
    ↪                | 115, 122, 123 | 0          | 35            | 0          | 1
    ↪                    | 0               |
| 3         | t_43_r_80000                |                             | 93        | 8
    ↪                | 73, 93, 128   | 0          | 0             | 0          | 1
    ↪                    | 0               |
| 135       | t_43_i_1_                   | t_43_i_1_016d80000000000000 | 139       | 2
    ↪                | 138, 139, 140 | 0          | 35            | 0          | 1
    ↪                    | 0               |
| 98        | t_43_i_1_016d80000000000000 | t_43_r                      | 99        | 1
    ↪                | 99, 100, 101  | 0          | 0             | 0          | 1
    ↪                    | 0               |
+-----------+-----------------------------+-----------------------------+-----------+-----------------+----------
    ↪
7 rows in set
```

12.11.2.105.3 MySQL 兼容性

SHOW TABLE REGIONS 语句是 TiDB 对 MySQL 语法的扩展。

12.11.2.105.4 另请参阅

- SPLIT REGION
- CREATE TABLE

12.11.2.106 SHOW TABLE STATUS

SHOW TABLE STATUS 语句用于显示 TiDB 中表的各种统计信息。如果显示统计信息过期，建议运行ANALYZE TABLE。

12.11.2.106.1 语法图

ShowTableStatusStmt:

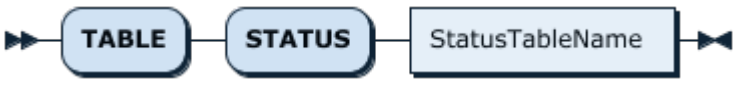


图 334: ShowTableStatusStmt

FromOrIn:

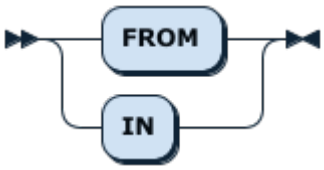


图 335: FromOrIn

StatusTableName:

图 336: StatusTableName

12.11.2.106.2 示例

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, c1 INT NOT NULL);
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
INSERT INTO t1 (c1) VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 5 rows affected (0.02 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
SHOW TABLE STATUS LIKE 't1';
```

```
*************************** 1. row ***************************
           Name: t1
         Engine: InnoDB
        Version: 10
     Row_format: Compact
           Rows: 0
 Avg_row_length: 0
    Data_length: 0
Max_data_length: 0
   Index_length: 0
      Data_free: 0
 Auto_increment: 30001
    Create_time: 2019-04-19 08:32:06
    Update_time: NULL
     Check_time: NULL
      Collation: utf8mb4_bin
       Checksum:
 Create_options:
```

```
        Comment:
1 row in set (0.00 sec)
```

```
analyze table t1;
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
SHOW TABLE STATUS LIKE 't1';
```

```
*************************** 1. row ***************************
           Name: t1
         Engine: InnoDB
        Version: 10
     Row_format: Compact
           Rows: 5
 Avg_row_length: 16
    Data_length: 80
Max_data_length: 0
   Index_length: 0
      Data_free: 0
 Auto_increment: 30001
    Create_time: 2019-04-19 08:32:06
    Update_time: NULL
     Check_time: NULL
      Collation: utf8mb4_bin
       Checksum:
 Create_options:
        Comment:
1 row in set (0.00 sec)
```

12.11.2.106.3 MySQL 兼容性

`SHOW TABLE STATUS` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.106.4 另请参阅

- SHOW TABLES
- CREATE TABLE
- DROP TABLE
- SHOW CREATE TABLE

12.11.2.107 SHOW [FULL] TABLES

`SHOW [FULL] TABLES` 语句用于显示当前所选数据库中表和视图的列表。可选关键字 `FULL` 说明表的类型是 `BASE TABLE` 还是 `VIEW`。

若要在不同的数据库中显示表，可使用 `SHOW TABLES IN DatabaseName` 语句。

12.11.2.107.1 语法图

ShowTablesStmt:

图 337: ShowTablesStmt

OptFull:

图 338: OptFull

ShowDatabaseNameOpt:

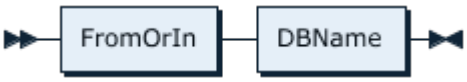


图 339: ShowDatabaseNameOpt

ShowLikeOrWhereOpt:

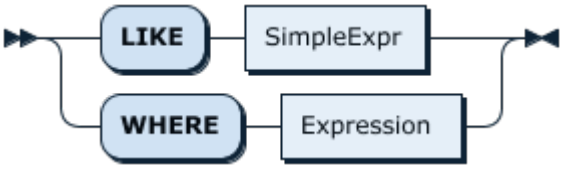


图 340: ShowLikeOrWhereOpt

12.11.2.107.2 示例

```
mysql> CREATE TABLE t1 (a int);
Query OK, 0 rows affected (0.12 sec)

mysql> CREATE VIEW v1 AS SELECT 1;
Query OK, 0 rows affected (0.10 sec)

mysql> SHOW TABLES;
+----------------+
| Tables_in_test |
```

```
+----------------+
| t1             |
| v1             |
+----------------+
2 rows in set (0.00 sec)

mysql> SHOW FULL TABLES;
+----------------+------------+
| Tables_in_test | Table_type |
+----------------+------------+
| t1             | BASE TABLE |
| v1             | VIEW       |
+----------------+------------+
2 rows in set (0.00 sec)

mysql> SHOW TABLES IN mysql;
+-------------------------+
| Tables_in_mysql         |
+-------------------------+
| GLOBAL_VARIABLES        |
| bind_info               |
| columns_priv            |
| db                      |
| default_roles           |
| expr_pushdown_blacklist |
| gc_delete_range         |
| gc_delete_range_done    |
| global_priv             |
| help_topic              |
| opt_rule_blacklist      |
| role_edges              |
| stats_buckets           |
| stats_feedback          |
| stats_histograms        |
| stats_meta              |
| stats_top_n             |
| tables_priv             |
| tidb                    |
| user                    |
+-------------------------+
20 rows in set (0.00 sec)
```

12.11.2.107.3 MySQL 兼容性

`SHOW [FULL] TABLES` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.107.4 另请参阅

- CREATE TABLE
- DROP TABLE
- SHOW CREATE TABLE

12.11.2.108 SHOW [GLOBAL|SESSION] VARIABLES

`SHOW [GLOBAL|SESSION] VARIABLES` 语句用于显示 `GLOBAL` 或 `SESSION` 范围的变量列表。如果未指定范围，则应用默认范围 `SESSION`。

12.11.2.108.1 语法图

ShowStmt:

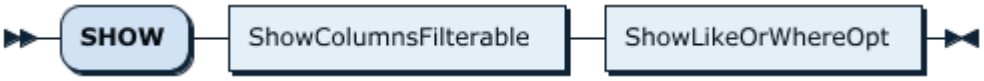


图 341: ShowStmt

ShowTargetFilterable:

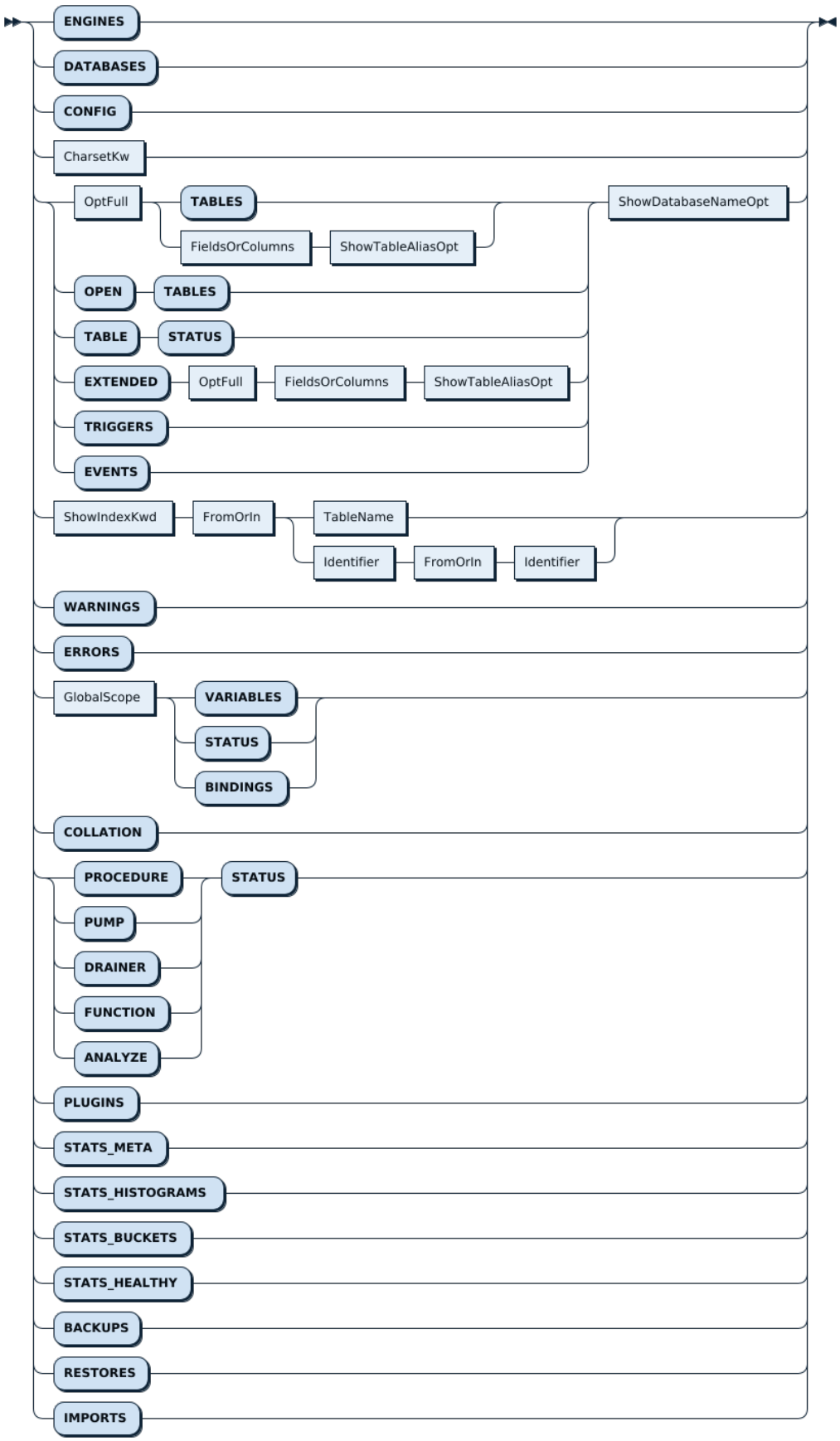


图 342: ShowTargetFilterable

GlobalScope:

图 343: GlobalScope

12.11.2.108.2 示例

查看 TiDB 定义的专用系统变量，关于这些变量的含义参见系统变量和语法。

```
SHOW GLOBAL VARIABLES LIKE 'tidb%';
```

```
+-----------------------------------+---------------------+
| Variable_name                     | Value               |
+-----------------------------------+---------------------+
| tidb_allow_batch_cop              | 0                   |
| tidb_allow_remove_auto_inc        | 0                   |
| tidb_auto_analyze_end_time        | 23:59 +0000         |
| tidb_auto_analyze_ratio           | 0.5                 |
| tidb_auto_analyze_start_time      | 00:00 +0000         |
| tidb_backoff_lock_fast            | 100                 |
| tidb_backoff_weight               | 2                   |
| tidb_batch_commit                 | 0                   |
| tidb_batch_delete                 | 0                   |
| tidb_batch_insert                 | 0                   |
| tidb_build_stats_concurrency      | 4                   |
| tidb_capture_plan_baselines       | off                 |
| tidb_check_mb4_value_in_utf8      | 1                   |
| tidb_checksum_table_concurrency   | 4                   |
| tidb_config                       |                     |
| tidb_constraint_check_in_place    | 0                   |
| tidb_current_ts                   | 0                   |
| tidb_ddl_error_count_limit        | 512                 |
| tidb_ddl_reorg_batch_size         | 256                 |
| tidb_ddl_reorg_priority           | PRIORITY_LOW        |
| tidb_ddl_reorg_worker_cnt         | 4                   |
| tidb_disable_txn_auto_retry       | 1                   |
| tidb_distsql_scan_concurrency     | 15                  |
| tidb_dml_batch_size               | 20000               |
| tidb_enable_cascades_planner      | 0                   |
| tidb_enable_chunk_rpc             | 1                   |
| tidb_enable_collect_execution_info| 1                   |
```

```
| tidb_enable_fast_analyze            | 0                   |
| tidb_enable_index_merge             | 0                   |
| tidb_enable_noop_functions          | 0                   |
| tidb_enable_radix_join              | 0                   |
| tidb_enable_slow_log                | 1                   |
| tidb_enable_stmt_summary            | 1                   |
| tidb_enable_table_partition          | on                  |
| tidb_enable_vectorized_expression   | 1                   |
| tidb_enable_window_function         | 1                   |
| tidb_evolve_plan_baselines          | off                 |
| tidb_evolve_plan_task_end_time      | 23:59 +0000         |
| tidb_evolve_plan_task_max_time      | 600                 |
| tidb_evolve_plan_task_start_time    | 00:00 +0000         |
| tidb_expensive_query_time_threshold | 60                  |
| tidb_force_priority                 | NO_PRIORITY         |
| tidb_general_log                    | 0                   |
| tidb_hash_join_concurrency          | 5                   |
| tidb_hashagg_final_concurrency      | 4                   |
| tidb_hashagg_partial_concurrency    | 4                   |
| tidb_index_join_batch_size          | 25000               |
| tidb_index_lookup_concurrency       | 4                   |
| tidb_index_lookup_join_concurrency  | 4                   |
| tidb_index_lookup_size              | 20000               |
| tidb_index_serial_scan_concurrency  | 1                   |
| tidb_init_chunk_size                | 32                  |
| tidb_isolation_read_engines         | tikv, tiflash, tidb |
| tidb_low_resolution_tso             | 0                   |
| tidb_max_chunk_size                 | 1024                |
| tidb_max_delta_schema_count         | 1024                |
| tidb_mem_quota_hashjoin             | 34359738368         |
| tidb_mem_quota_indexlookupjoin      | 34359738368         |
| tidb_mem_quota_indexlookupreader    | 34359738368         |
| tidb_mem_quota_mergejoin            | 34359738368         |
| tidb_mem_quota_nestedloopapply      | 34359738368         |
| tidb_mem_quota_query                | 1073741824          |
| tidb_mem_quota_sort                 | 34359738368         |
| tidb_mem_quota_topn                 | 34359738368         |
| tidb_metric_query_range_duration    | 60                  |
| tidb_metric_query_step              | 60                  |
| tidb_opt_agg_push_down              | 0                   |
| tidb_opt_concurrency_factor         | 3                   |
| tidb_opt_copcpu_factor              | 3                   |
| tidb_opt_correlation_exp_factor     | 1                   |
| tidb_opt_correlation_threshold      | 0.9                 |
| tidb_opt_cpu_factor                 | 3                   |
```

```
| tidb_opt_desc_factor                | 3                   |
| tidb_opt_disk_factor                | 1.5                 |
| tidb_opt_distinct_agg_push_down     | 0                   |
| tidb_opt_insubq_to_join_and_agg     | 1                   |
| tidb_opt_join_reorder_threshold     | 0                   |
| tidb_opt_memory_factor              | 0.001               |
| tidb_opt_network_factor             | 1                   |
| tidb_opt_scan_factor                | 1.5                 |
| tidb_opt_seek_factor                | 20                  |
| tidb_opt_write_row_id               | 0                   |
| tidb_optimizer_selectivity_level    | 0                   |
| tidb_pprof_sql_cpu                  | 0                   |
| tidb_projection_concurrency         | 4                   |
| tidb_query_log_max_len              | 4096                |
| tidb_record_plan_in_slow_log        | 1                   |
| tidb_replica_read                   | leader              |
| tidb_retry_limit                    | 10                  |
| tidb_row_format_version             | 2                   |
| tidb_scatter_region                 | 0                   |
| tidb_skip_isolation_level_check     | 0                   |
| tidb_skip_utf8_check                | 0                   |
| tidb_slow_log_threshold             | 300                 |
| tidb_slow_query_file                | tidb-slow.log       |
| tidb_snapshot                       |                     |
| tidb_stmt_summary_history_size      | 24                  |
| tidb_stmt_summary_internal_query    | 0                   |
| tidb_stmt_summary_max_sql_length    | 4096                |
| tidb_stmt_summary_max_stmt_count    | 3000                |
| tidb_stmt_summary_refresh_interval  | 1800                |
| tidb_store_limit                    | 0                   |
| tidb_txn_mode                       |                     |
| tidb_use_plan_baselines             | on                  |
| tidb_wait_split_region_finish       | 1                   |
| tidb_wait_split_region_timeout      | 300                 |
| tidb_window_concurrency             | 4                   |
+-------------------------------------+---------------------+
108 rows in set (0.01 sec)
```

```
SHOW GLOBAL VARIABLES LIKE 'time_zone%';
```

```
+---------------+--------+
| Variable_name | Value  |
+---------------+--------+
| time_zone     | SYSTEM |
+---------------+--------+
```

```
1 row in set (0.00 sec)
```

12.11.2.108.3 MySQL 兼容性

`SHOW [GLOBAL|SESSION] VARIABLES` 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.108.4 另请参阅

- `SET [GLOBAL|SESSION]`

12.11.2.109 SHOW WARNINGS

`SHOW WARNINGS` 语句用于显示当前客户端连接中已执行语句的报错列表。与在 MySQL 中一样，`sql_mode` 极大地影响哪些语句会导致错误与警告。

12.11.2.109.1 语法图

ShowWarningsStmt:

图 344: ShowWarningsStmt

12.11.2.109.2 示例

```
CREATE TABLE t1 (a INT UNSIGNED);
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
INSERT INTO t1 VALUES (0);
```

```
Query OK, 1 row affected (0.02 sec)
```

```
SELECT 1/a FROM t1;
```

```
+------+
| 1/a  |
+------+
| NULL |
+------+
1 row in set, 1 warning (0.00 sec)
```

```
SHOW WARNINGS;
```

```
+---------+------+---------------+
| Level   | Code | Message       |
+---------+------+---------------+
| Warning | 1365 | Division by 0 |
+---------+------+---------------+
1 row in set (0.00 sec)
```

```
INSERT INTO t1 VALUES (-1);
```

```
ERROR 1264 (22003): Out of range value for column 'a' at row 1
```

```
SELECT * FROM t1;
```

```
+------+
| a    |
+------+
|    0 |
+------+
1 row in set (0.00 sec)
```

```
SET sql_mode='';
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
INSERT INTO t1 VALUES (-1);
```

```
Query OK, 1 row affected, 1 warning (0.01 sec)
```

```
SHOW WARNINGS;
```

```
+---------+------+---------------------------+
| Level   | Code | Message                   |
+---------+------+---------------------------+
| Warning | 1690 | constant -1 overflows int |
+---------+------+---------------------------+
1 row in set (0.00 sec)
```

```
SELECT * FROM t1;
```

```
+------+
| a    |
+------+
|    0 |
```

```
|    0 |
+------+
2 rows in set (0.00 sec)
```

12.11.2.109.3 MySQL 兼容性

SHOW WARNINGS 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.109.4 另请参阅

- SHOW ERRORS

12.11.2.110 SHUTDOWN

SHUTDOWN 语句用于在 TiDB 中执行停机操作。执行 SHUTDOWN 语句需要用户拥有 SHUTDOWN privilege。

12.11.2.110.1 语法图

Statement:

图 345: Statement

12.11.2.110.2 示例

```
SHUTDOWN;
```

```
Query OK, 0 rows affected (0.00 sec)
```

12.11.2.110.3 MySQL 兼容性

> **注意：**
>
> 由于 TiDB 是分布式数据库，因此 TiDB 中的停机操作停止的是客户端连接的 TiDB 实例，而不是整个 TiDB 集群。

SHUTDOWN 语句与 MySQL 不完全兼容。如发现任何其他兼容性差异，请在 GitHub 上提交 issue。

12.11.2.111 Split Region 使用文档

在 TiDB 中新建一个表后，默认会单独切分出 1 个 Region 来存储这个表的数据，这个默认行为由配置文件中的 `split-table` 控制。当这个 Region 中的数据超过默认 Region 大小限制后，这个 Region 会开始分裂成 2 个 Region。

上述情况中，如果在新建的表上发生大批量写入，则会造成热点，因为开始只有一个 Region，所有的写请求都发生在该 Region 所在的那台 TiKV 上。

为解决上述场景中的热点问题，TiDB 引入了预切分 Region 的功能，即可以根据指定的参数，预先为某个表切分出多个 Region，并打散到各个 TiKV 上去。

12.11.2.111.1 语法图

SplitRegionStmt:

图 346: SplitRegionStmt

SplitSyntaxOption:

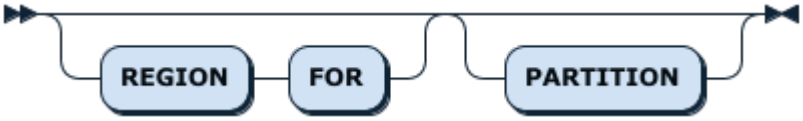


图 347: SplitSyntaxOption

TableName:

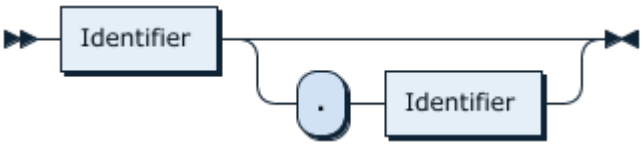


图 348: TableName

PartitionNameListOpt:

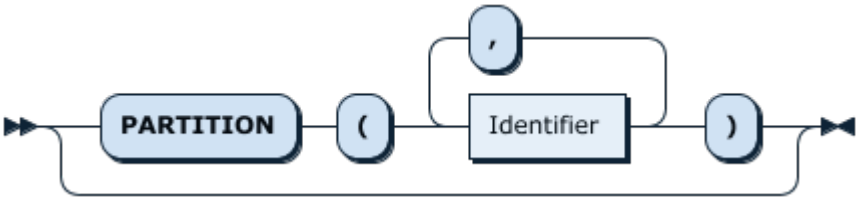


图 349: PartitionNameListOpt

SplitOption:

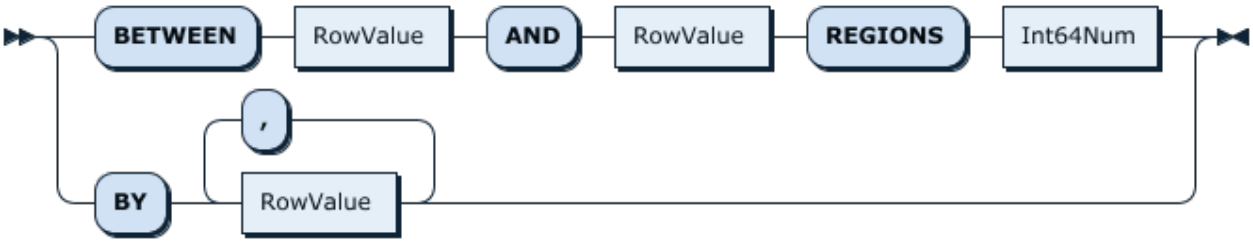


图 350: SplitOption

RowValue:

图 351: RowValue

Int64Num:

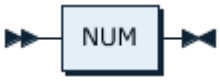


图 352: Int64Num

12.11.2.111.2 Split Region 的使用

Split Region 有 2 种不同的语法，具体如下：

```
SPLIT TABLE table_name [INDEX index_name] BETWEEN (lower_value) AND (upper_value) REGIONS
    ↪ region_num
```

`BETWEEN lower_value AND upper_value REGIONS region_num` 语法是通过指定上、下边界和 Region 数量，然后在上、下边界之间均匀切分出 `region_num` 个 Region。

```
SPLIT TABLE table_name [INDEX index_name] BY (value_list) [, (value_list)] ...
```

`BY value_list…` 语法将手动指定一系列的点，然后根据这些指定的点切分 Region，适用于数据不均匀分布的场景。

`SPLIT` 语句的返回结果示例如下：

```
+--------------------+----------------------+
| TOTAL_SPLIT_REGION | SCATTER_FINISH_RATIO |
+--------------------+----------------------+
| 4                  | 1.0                  |
+--------------------+----------------------+
```

- TOTAL_SPLIT_REGION：表示新增预切分的 Region 数量。
- SCATTER_FINISH_RATIO：表示新增预切分 Region 中，打散完成的比率。如 1.0 表示全部完成。0.5表示只有一半的 Region 已经打散完成，剩下的还在打散过程中。

> 注意：
>
> 以下会话变量会影响 SPLIT 语句的行为，需要特别注意：
>
> - tidb_wait_split_region_finish：打散 Region 的时间可能较长，由 PD 调度以及 TiKV 的负载情况所决定。这个变量用来设置在执行 SPLIT REGION 语句时，是否同步等待所有 Region 都打散完成后再返回结果给客户端。默认 1 代表等待打散完成后再返回结果。0 代表不等待 Region 打散完成就返回结果。
> - tidb_wait_split_region_timeout：这个变量用来设置 SPLIT REGION 语句的执行超时时间，单位是秒，默认值是 300 秒，如果超时还未完成 Split 操作，就返回一个超时错误。

Split Table Region

表中行数据的 key 由 table_id 和 row_id 编码组成，格式如下：

```
t[table_id]_r[row_id]
```

例如，当 table_id 是 22，row_id 是 11 时：

```
t22_r11
```

同一表中行数据的 table_id 是一样的，但 row_id 肯定不一样，所以可以根据 row_id 来切分 Region。

均匀切分

由于 row_id 是整数，所以根据指定的 lower_value、upper_value 以及 region_num，可以推算出需要切分的 key。TiDB 先计算 step（step = (upper_value - lower_value)/num），然后在 lower_value 和 upper_value 之间每隔 step 区间切一次，最终切出 num 个 Region。

例如，对于表 t，如果想要从 minInt64~maxInt64 之间均匀切割出 16 个 Region，可以用以下语句：

```
SPLIT TABLE t BETWEEN (-9223372036854775808) AND (9223372036854775807) REGIONS 16;
```

该语句会把表 t 从 minInt64 到 maxInt64 之间均匀切割出 16 个 Region。如果已知主键的范围没有这么大，比如只会在 0~1000000000 之间，那可以用 0 和 1000000000 分别代替上面的 minInt64 和 maxInt64 来切分 Region。

```
SPLIT TABLE t BETWEEN (0) AND (1000000000) REGIONS 16;
```

不均匀切分

如果已知数据不是均匀分布的，比如想要 -inf ~ 10000 切一个 Region，10000 ~ 90000 切一个 Region，90000 ~ +inf 切一个 Region，可以通过手动指定点来切分 Region，示例如下：

```
SPLIT TABLE t BY (10000), (90000);
```

Split Index Region

表中索引数据的 key 由 table_id、index_id 以及索引列的值编码组成，格式如下：

```
t[table_id]_i[index_id][index_value]
```

例如，当 table_id 是 22，index_id 是 5，index_value 是 abc 时：

```
t22_i5abc
```

同一表中同一索引数据的 table_id 和 index_id 是一样的，所以要根据 index_value 切分索引 Region。

均匀切分

索引均匀切分与行数据均匀切分的原理一样，只是计算 step 的值较为复杂，因为 index_value 可能不是整数。

upper 和 lower 的值会先编码成 byte 数组，去掉 lower 和 upper byte 数组的最长公共前缀后，从 lower 和 upper 各取前 8 字节转成 uint64，再计算 step = (upper - lower)/num。计算出 step 后再将 step 编码成 byte 数组，添加到之前 upper和 lower的最长公共前缀后面组成一个 key 后去做切分。示例如下：

如果索引 idx 的列也是整数类型，可以用如下 SQL 语句切分索引数据：

```
SPLIT TABLE t INDEX idx BETWEEN (-9223372036854775808) AND (9223372036854775807) REGIONS 16;
```

该语句会把表 t 中 idx 索引数据 Region 从 minInt64 到 maxInt64 之间均匀切割出 16 个 Region。

如果索引 idx1 的列是 varchar 类型，希望根据前缀字母来切分索引数据：

```
SPLIT TABLE t INDEX idx1 BETWEEN ("a") AND ("z") REGIONS 25;
```

该语句会把表 t 中 idx1 索引数据的 Region 从 a~z 切成 25 个 Region，region1 的范围是 [minIndexValue, b)，region2 的范围是 [b, c)，……，region25 的范围是 [y, maxIndexValue)。对于 idx1 索引以 a 为前缀的数据都会写到 region1，以 b 为前缀的索引数据都会写到 region2，以此类推。

上面的切分方法，以 y 和 z 前缀的索引数据都会写到 region 25, 因为 z 并不是一个上界，真正的上界是 z 在 ASCII 码中的下一位 {，所以更准确的切分方法如下：

```
SPLIT TABLE t INDEX idx1 BETWEEN ("a") AND ("{") REGIONS 26;
```

该语句会把表 t 中 idx1 索引数据的 Region 从 a~{ 切成 26 个 Region，region1 的范围是 [minIndexValue, b)，region2 的范围是 [b, c)，……，region25 的范围是 [y,z)，region26 的范围是 [z, maxIndexValue)。

如果索引 idx2 的列是 timestamp/datetime 等时间类型，希望根据时间区间，按年为间隔切分索引数据，示例如下：

```
SPLIT TABLE t INDEX idx2 BETWEEN ("2010-01-01 00:00:00") AND ("2020-01-01 00:00:00") REGIONS 10;
```

该语句会把表 t 中 idx2 的索引数据 Region 从 2010-01-01 00:00:00 到 2020-01-01 00:00:00 切成 10 个 Region。region1 的范围是从 [minIndexValue, 2011-01-01 00:00:00)，region2 的范围是 [2011-01-01 00:00:00, 2012-01-01 00:00:00)……

如果希望按照天为间隔切分索引，示例如下：

```
SPLIT TABLE t INDEX idx2 BETWEEN ("2020-06-01 00:00:00") AND ("2020-07-01 00:00:00") REGIONS 30;
```

该语句会将表 t 中 idx2 索引位于 2020 年 6 月份的数据按天为间隔切分成 30 个 Region。

其他索引列类型的切分方法也是类似的。

对于联合索引的数据 Region 切分，唯一不同的是可以指定多个 column 的值。

比如索引 idx3 (a, b) 包含 2 列，a 是 timestamp，b 是 int。如果只想根据 a 列做时间范围的切分，可以用切分单列时间索引的 SQL 语句来切分，lower_value 和 upper_velue 中不指定 b 列的值即可。

```
SPLIT TABLE t INDEX idx3 BETWEEN ("2010-01-01 00:00:00") AND ("2020-01-01 00:00:00") REGIONS 10;
```

如果想在时间相同的情况下，根据 b 列再做一次切分，在切分时指定 b 列的值即可。

```
SPLIT TABLE t INDEX idx3 BETWEEN ("2010-01-01 00:00:00", "a") AND ("2010-01-01 00:00:00", "z")
    ↪ REGIONS 10;
```

该语句在 a 列时间前缀相同的情况下，根据 b 列的值从 a~z 切了 10 个 Region。如果指定的 a 列的值不相同，那么可能不会用到 b 列的值。

不均匀切分

索引数据也可以根据用户指定的索引值来做切分。

假如有 idx4 (a,b)，其中 a 列是 varchar 类型，b 列是 timestamp 类型。

```
SPLIT TABLE t1 INDEX idx4 BY ("a", "2000-01-01 00:00:01"), ("b", "2019-04-17 14:26:19"), ("c", ""
    ↪ );
```

该语句指定了 3 个值，会切分出 4 个 Region，每个 Region 的范围如下。

```
region1  [ minIndexValue               , ("a", "2000-01-01 00:00:01"))
region2  [("a", "2000-01-01 00:00:01") , ("b", "2019-04-17 14:26:19"))
region3  [("b", "2019-04-17 14:26:19") , ("c", "")                    )
region4  [("c", "")                    , maxIndexValue                )
```

Split 分区表的 Region

预切分分区表的 Region 在使用上和普通表一样，差别是会为每一个 partition 都做相同的切分。

- 均匀切分的语法如下：

```
SPLIT [PARTITION] TABLE t [PARTITION] [(partition_name_list...)] [INDEX index_name] BETWEEN
    ↪ (lower_value) AND (upper_value) REGIONS region_num
```

- 不均匀切分的语法如下：

```
SPLIT [PARTITION] TABLE table_name [PARTITION (partition_name_list...)] [INDEX index_name]
    ↪ BY (value_list) [, (value_list)] ...
```

示例

1. 首先创建一个分区表。

```
create table t (a int,b int,index idx(a)) partition by hash(a) partitions 2;
```

此时建完表后会为每个 partition 都单独 split 一个 Region，用 SHOW TABLE REGIONS 语法查看该表的 Region 如下：

```
show table t regions;
```

```
+--
    ↪ ---------+-----------+---------+-----------+----------------+-----------------+------------+------
    ↪
| REGION_ID | START_KEY | END_KEY | LEADER_ID | LEADER_STORE_ID | PEERS            |
    ↪ SCATTERING | WRITTEN_BYTES | READ_BYTES | APPROXIMATE_SIZE(MB) | APPROXIMATE_KEYS |
+--
    ↪ ---------+-----------+---------+-----------+----------------+-----------------+------------+------
    ↪
| 1978      | t_1400_   | t_1401_ | 1979      | 4              | 1979, 1980, 1981 | 0
    ↪          | 0             | 0          | 1                   | 0                |
| 6         | t_1401_   |         | 17        | 4              | 17, 18, 21       | 0
    ↪          | 223           | 0          | 1                   | 0                |
+--
    ↪ ---------+-----------+---------+-----------+----------------+-----------------+------------+------
    ↪
```

2. 用 SPLIT 语法为每个 partition 切分 Region，示例如下，在 [0,10000] 范围内切分成 4 个 Region。

```
split partition table t between (0) and (10000) regions 4;
```

3. 用 SHOW TABLE REGIONS 语法查看该表的 Region。如下会发现该表现在一共有 10 个 Region，每个 partition 分别有 5 个 Region，其中 4 个 Region 是表的行数据，1 个 Region 是表的索引数据。

```
show table t regions;
```

```
+--
    ↪ ---------+--------------+--------------+-----------+----------------+-----------------+---------
    ↪
| REGION_ID | START_KEY     | END_KEY       | LEADER_ID | LEADER_STORE_ID | PEERS
    ↪            | SCATTERING | WRITTEN_BYTES | READ_BYTES | APPROXIMATE_SIZE(MB) |
    ↪ APPROXIMATE_KEYS |
+--
    ↪ ---------+--------------+--------------+-----------+----------------+-----------------+---------
    ↪
| 1998      | t_1400_r      | t_1400_r_2500 | 2001      | 5               | 2000, 2001, 2015
    ↪  | 0          | 132           | 0          | 1                   | 0
    ↪                |
| 2006      | t_1400_r_2500 | t_1400_r_5000 | 2016      | 1               | 2007, 2016, 2017
    ↪  | 0          | 35            | 0          | 1                   | 0
    ↪                |
```

```
| 2010      | t_1400_r_5000 | t_1400_r_7500 | 2012      | 2                 | 2011, 2012, 2013
    ↪  | 0           | 35            | 0           | 1                     | 0
    ↪                |
| 1978      | t_1400_r_7500 | t_1401_       | 1979      | 4                 | 1979, 1980, 1981
    ↪  | 0           | 621           | 0           | 1                     | 0
    ↪                |
| 1982      | t_1400_       | t_1400_r      | 2014      | 3                 | 1983, 1984, 2014
    ↪  | 0           | 35            | 0           | 1                     | 0
    ↪                |
| 1990      | t_1401_r      | t_1401_r_2500 | 1992      | 2                 | 1991, 1992, 2020
    ↪  | 0           | 120           | 0           | 1                     | 0
    ↪                |
| 1994      | t_1401_r_2500 | t_1401_r_5000 | 1997      | 5                 | 1996, 1997, 2021
    ↪  | 0           | 129           | 0           | 1                     | 0
    ↪                |
| 2002      | t_1401_r_5000 | t_1401_r_7500 | 2003      | 4                 | 2003, 2023, 2022
    ↪  | 0           | 141           | 0           | 1                     | 0
    ↪                |
| 6         | t_1401_r_7500 |               | 17        | 4                 | 17, 18, 21
    ↪       | 0           | 601           | 0           | 1                       | 0
    ↪                |
| 1986      | t_1401_       | t_1401_r      | 1989      | 5                 | 1989, 2018, 2019
    ↪  | 0           | 123           | 0           | 1                     | 0
    ↪                |
+--
    ↪ ---------+---------------+---------------+-----------+-------------------+-----------------+---------
    ↪
```

4. 也可以给每个分区的索引切分 Region，如将索引 idx 的 [1000,10000] 范围切分成 2 个 Region：

```
split partition table t index idx between (1000) and (10000) regions 2;
```

Split 单个分区的 Region

可以单独指定要切分的 partition，示例如下：

1. 首先创建一个分区表：

```
create table t ( a int, b int, index idx(b)) partition by range( a ) (
    partition p1 values less than (10000),
    partition p2 values less than (20000),
    partition p3 values less than (MAXVALUE) );
```

2. 为 p1 分区的 [0,10000] 预切分 2 个 Region：

```
split partition table t partition (p1) between (0) and (10000) regions 2;
```

3. 为 p2 分区的 [10000,20000] 预切分 2 个 Region：

```
split partition table t partition (p2) between (10000) and (20000) regions 2;
```

4. 用 SHOW TABLE REGIONS 语法查看该表的 Region 如下：

```
show table t regions;
```

```
+--
    ↪ ---------+----------------+----------------+-----------+-----------------+------------------+-------
    ↪
| REGION_ID | START_KEY      | END_KEY        | LEADER_ID | LEADER_STORE_ID | PEERS
    ↪            | SCATTERING | WRITTEN_BYTES | READ_BYTES | APPROXIMATE_SIZE(MB) |
    ↪ APPROXIMATE_KEYS |
+--
    ↪ ---------+----------------+----------------+-----------+-----------------+------------------+-------
    ↪
| 2040      | t_1406_        | t_1406_r_5000  | 2045      | 3               | 2043, 2045,
    ↪ 2044 | 0           | 0              | 0           | 1                 | 0
    ↪                |
| 2032      | t_1406_r_5000  | t_1407_        | 2033      | 4               | 2033, 2034,
    ↪ 2035 | 0           | 0              | 0           | 1                 | 0
    ↪                |
| 2046      | t_1407_        | t_1407_r_15000 | 2048      | 2               | 2047, 2048,
    ↪ 2050 | 0           | 35             | 0           | 1                 | 0
    ↪                |
| 2036      | t_1407_r_15000 | t_1408_        | 2037      | 4               | 2037, 2038,
    ↪ 2039 | 0           | 0              | 0           | 1                 | 0
    ↪                |
| 6         | t_1408_        |                | 17        | 4               | 17, 18, 21
    ↪       | 0           | 214            | 0           | 1                 | 0
    ↪                |
+--
    ↪ ---------+----------------+----------------+-----------+-----------------+------------------+-------
    ↪
```

5. 为 p1 和 p2 分区的索引 idx 的 [0,20000] 范围预切分 2 个 Region:

```
split partition table t partition (p1,p2) index idx between (0) and (20000) regions 2;
```

12.11.2.111.3 pre_split_regions

使用带有 SHARD_ROW_ID_BITS 的表时，如果希望建表时就均匀切分 Region，可以考虑配合 PRE_SPLIT_REGIONS 一起使用，用来在建表成功后就开始预均匀切分 2^(PRE_SPLIT_REGIONS) 个 Region。

> **注意：**
>
> PRE_SPLIT_REGIONS 必须小于等于 SHARD_ROW_ID_BITS。

以下全局变量会影响 PRE_SPLIT_REGIONS 的行为，需要特别注意：

- tidb_scatter_region：该变量用于控制建表完成后是否等待预切分和打散 Region 完成后再返回结果。如果建表后有大批量写入，需要设置该变量值为 1，表示等待所有 Region 都切分和打散完成后再返回结果给客户端。否则未打散完成就进行写入会对写入性能影响有较大的影响。

示例

```
create table t (a int, b int,index idx1(a)) shard_row_id_bits = 4 pre_split_regions=2;
```

该语句在建表后，会对这个表 t 预切分出 4 + 1 个 Region。4 (2^2) 个 Region 是用来存 table 的行数据的，1 个 Region 是用来存 idx1 索引的数据。

4 个 table Region 的范围区间如下：

```
region1:    [ -inf       ,  1<<61 )
region2:    [ 1<<61      ,  2<<61 )
region3:    [ 2<<61      ,  3<<61 )
region4:    [ 3<<61      ,  +inf  )
```

12.11.2.111.4 注意事项

Split Region 语句切分的 Region 会受到 PD 中Region merge 调度的控制，需要动态修改 Region merge 相关的配置项，避免新切分的 Region 不久后又被 PD 重新合并的情况。

12.11.2.111.5 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

12.11.2.111.6 另请参阅

- SHOW TABLE REGIONS
- Session 变量：tidb_scatter_region，tidb_wait_split_region_finish 和tidb_wait_split_region_timeout
  ↪ .

12.11.2.112 START TRANSACTION

START TRANSACTION 语句用于在 TiDB 内部启动新事务。它类似于语句 BEGIN。

在没有 START TRANSACTION 语句的情况下，每个语句都会在各自的事务中自动提交，这样可确保 MySQL 兼容性。

12.11.2.112.1 语法图

BeginTransactionStmt:

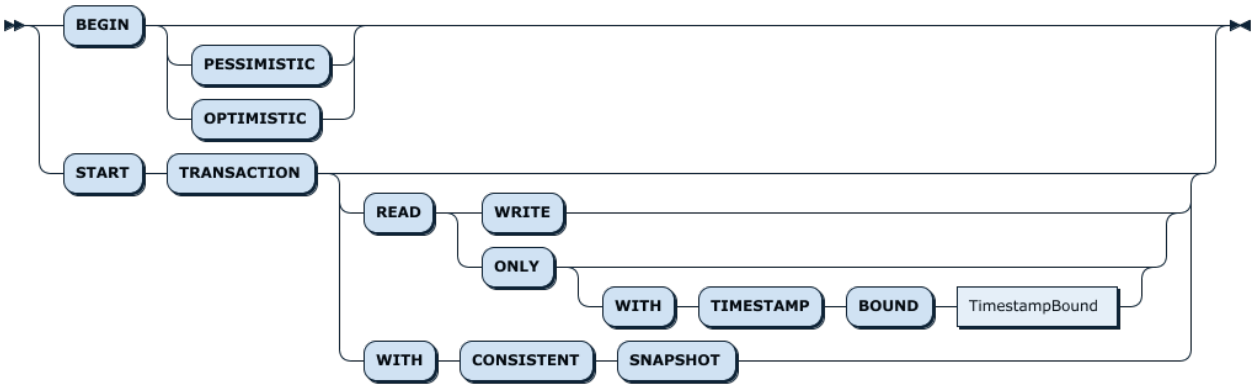


图 353: BeginTransactionStmt

12.11.2.112.2 示例

```
CREATE TABLE t1 (a int NOT NULL PRIMARY KEY);
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
START TRANSACTION;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
INSERT INTO t1 VALUES (1);
```

```
Query OK, 1 row affected (0.00 sec)
```

```
COMMIT;
```

```
Query OK, 0 rows affected (0.01 sec)
```

12.11.2.112.3 MySQL 兼容性

- 执行 START TRANSACTION 在 TiDB 中开启事务并立即生成快照。而在 MySQL 中，执行 START TRANSACTION 会开启事务但不会立即生成快照。TiDB 中的 START TRANSACTION 等同于 MySQL 中的 START TRANSACTION WITH CONSISTENT SNAPSHOT。
- 为与 MySQL 兼容，TiDB 会解析 START TRANSACTION READ ONLY 语句，但解析后 TiDB 仍允许写入操作。

12.11.2.112.4 另请参阅

- COMMIT
- ROLLBACK
- BEGIN

12.11.2.113 TRACE

TRACE 语句用于提供查询执行的详细信息，可通过 TiDB 服务器状态端口所公开的图形界面进行查看。

12.11.2.113.1 语法图

TraceStmt:

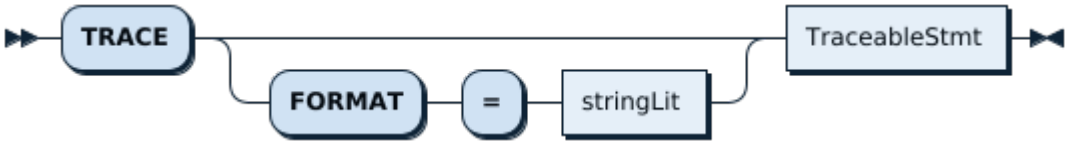


图 354: TraceStmt

TraceableStmt:

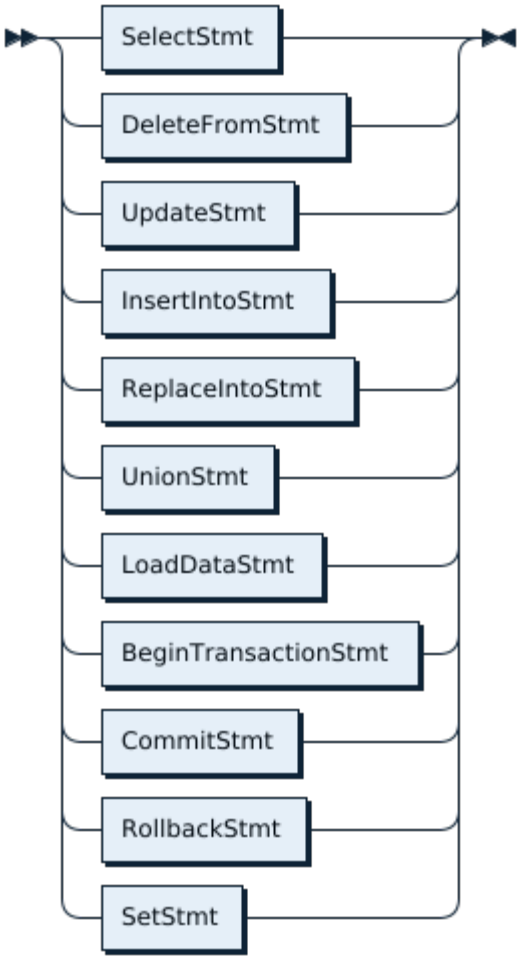


图 355: TraceableStmt

12.11.2.113.2 示例

```
trace format='row' select * from mysql.user;
```

```
+-----------------------------------------+-----------------+------------+
| operation                               | startTS         | duration   |
+-----------------------------------------+-----------------+------------+
| trace                                   | 17:03:31.938237 | 886.086µs  |
|   ├─session.Execute                      | 17:03:31.938247 | 507.812µs  |
|   │ ├─session.ParseSQL                   | 17:03:31.938254 | 22.504µs   |
|   │ ├─executor.Compile                   | 17:03:31.938321 | 278.931µs  |
|   │ │ └─session.getTxnFuture             | 17:03:31.938337 | 1.515µs    |
|   │ └─session.runStmt                    | 17:03:31.938613 | 109.578µs  |
|   │   ├─TableReaderExecutor.Open         | 17:03:31.938645 | 50.657µs   |
|   │   │ └─distsql.Select                 | 17:03:31.938666 | 21.066µs   |
|   │   │   └─RPCClient.SendRequest        | 17:03:31.938799 | 158.411µs  |
|   │   └─session.CommitTxn                | 17:03:31.938705 | 12.06µs    |
|   │     └─session.doCommitWitRetry       | 17:03:31.938709 | 2.437µs    |
```

```
|   ├─*executor.TableReaderExecutor.Next     | 17:03:31.938781 | 224.327µs  |
|   └─*executor.TableReaderExecutor.Next     | 17:03:31.939019 | 6.266µs    |
+-------------------------------------------+-----------------+------------+
13 rows in set (0.00 sec)
```

```
trace format='json' select * from mysql.user;
```

可将 JSON 格式的跟踪文件粘贴到跟踪查看器中。查看器可通过 TiDB 状态端口访问：

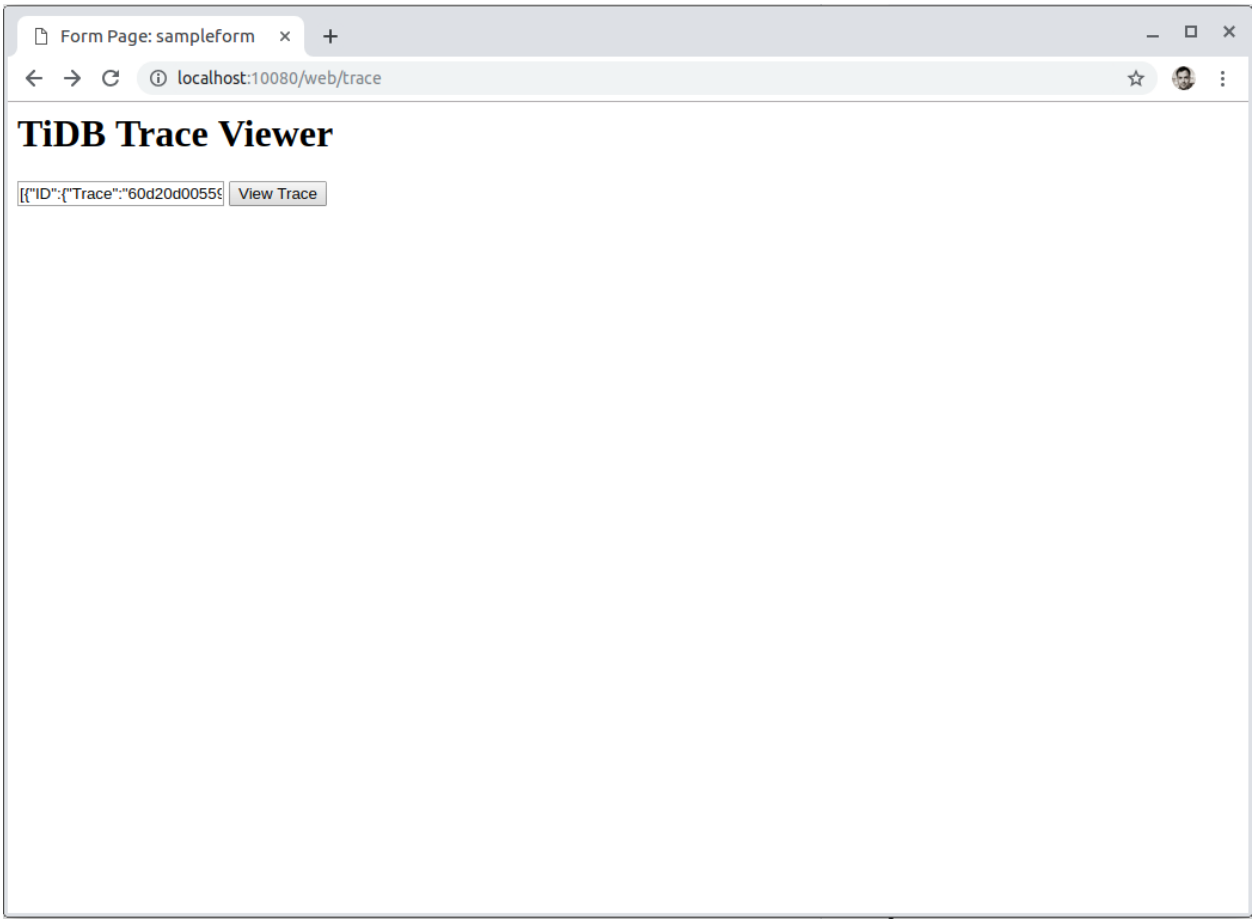


图 356: TiDB Trace Viewer-1

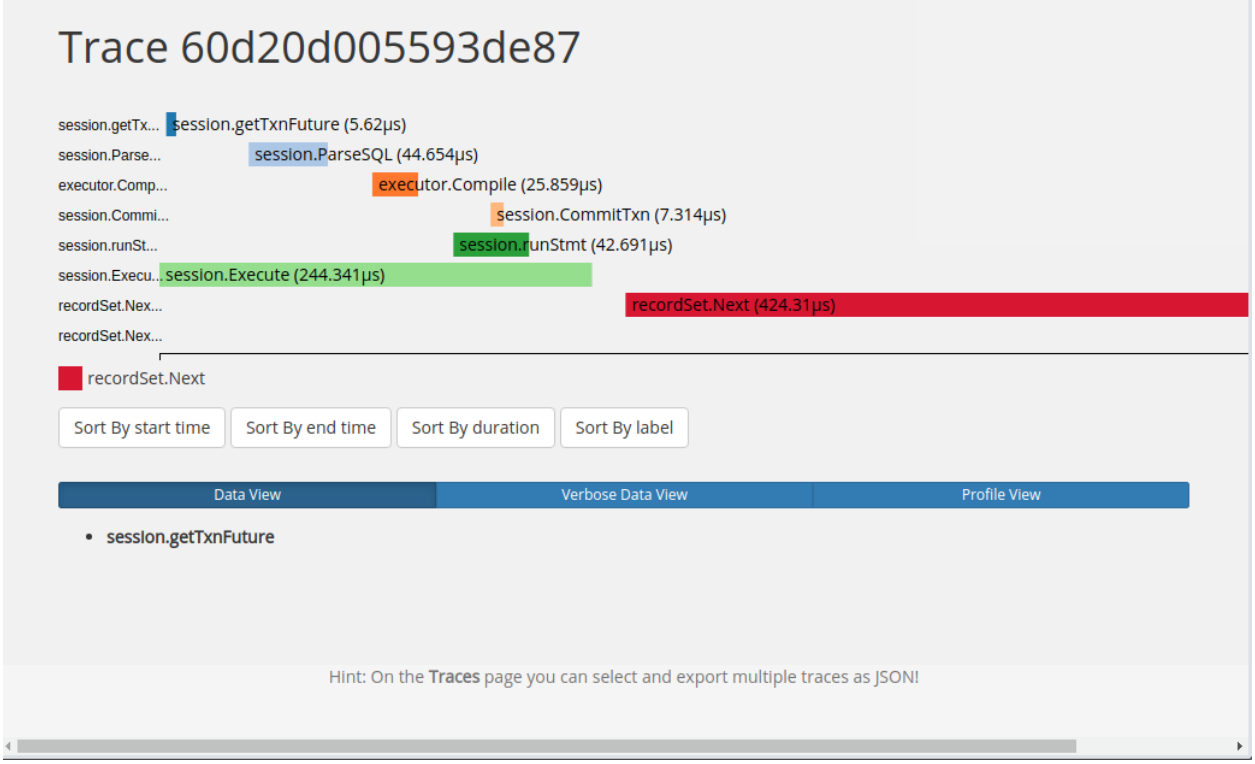


图 357: TiDB Trace Viewer-2

12.11.2.113.3 MySQL 兼容性

TRACE 语句是 TiDB 对 MySQL 语法的扩展。

12.11.2.113.4 另请参阅

- EXPLAIN ANALYZE

12.11.2.114 TRUNCATE

TRUNCATE 语句以非事务方式从表中删除所有数据。可认为 TRUNCATE 语句同 DROP TABLE + CREATE TABLE 组合在语义上相同，定义与 DROP TABLE 语句相同。

TRUNCATE TABLE tableName 和 TRUNCATE tableName 均为有效语法。

12.11.2.114.1 语法图

TruncateTableStmt:

图 358: TruncateTableStmt

OptTable:

图 359: OptTable

TableName:

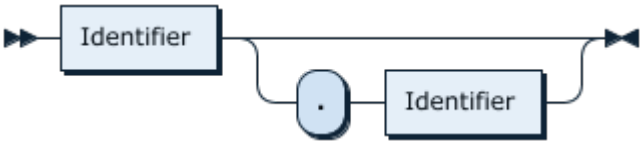


图 360: TableName

12.11.2.114.2 示例

```
CREATE TABLE t1 (a INT NOT NULL PRIMARY KEY);
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
INSERT INTO t1 VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 5 rows affected (0.01 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
SELECT * FROM t1;
```

```
+---+
| a |
+---+
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
+---+
5 rows in set (0.00 sec)
```

```
TRUNCATE t1;
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
SELECT * FROM t1;
```

```
Empty set (0.00 sec)
```

```
INSERT INTO t1 VALUES (1),(2),(3),(4),(5);
```

```
Query OK, 5 rows affected (0.01 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
TRUNCATE TABLE t1;
```

```
Query OK, 0 rows affected (0.11 sec)
```

12.11.2.114.3 MySQL 兼容性

TRUNCATE 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.114.4 另请参阅

- DROP TABLE
- DELETE
- CREATE TABLE
- SHOW CREATE TABLE

12.11.2.115 UPDATE

UPDATE 语句用于修改指定表中的数据。

12.11.2.115.1 语法图

UpdateStmt:

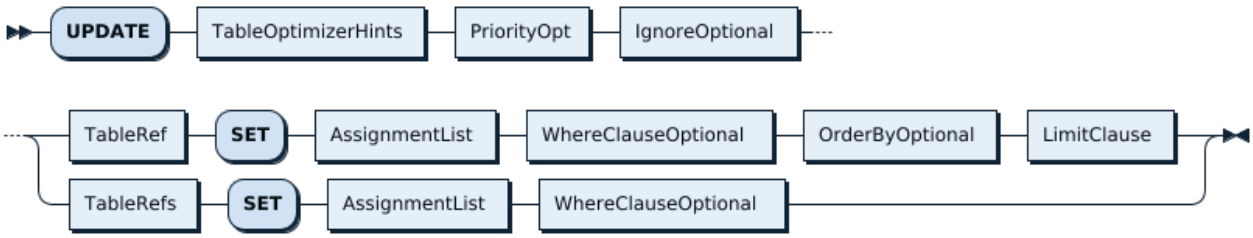


图 361: UpdateStmt

PriorityOpt:

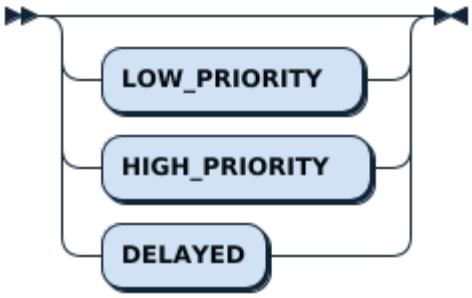


图 362: PriorityOpt

TableRef:

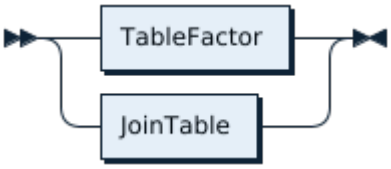


图 363: TableRef

TableRefs:

图 364: TableRefs

AssignmentList:

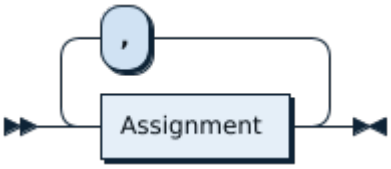


图 365: AssignmentList

WhereClauseOptional:

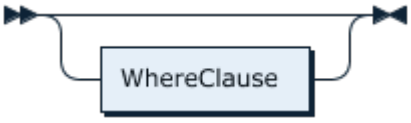


图 366: WhereClauseOptional

12.11.2.115.2 示例

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, c1 INT NOT NULL);
```

```
Query OK, 0 rows affected (0.11 sec)
```

```
INSERT INTO t1 (c1) VALUES (1), (2), (3);
```

```
Query OK, 3 rows affected (0.02 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
SELECT * FROM t1;
```

```
+----+----+
| id | c1 |
+----+----+
|  1 |  1 |
|  2 |  2 |
|  3 |  3 |
+----+----+
3 rows in set (0.00 sec)
```

```
UPDATE t1 SET c1=5 WHERE c1=3;
```

```
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

```
SELECT * FROM t1;
```

```
+----+----+
| id | c1 |
+----+----+
|  1 |  1 |
|  2 |  2 |
|  3 |  5 |
+----+----+
3 rows in set (0.00 sec)
```

12.11.2.115.3 MySQL 兼容性

UPDATE 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.115.4 另请参阅

- INSERT
- SELECT
- DELETE
- REPLACE

12.11.2.116 USE

USE 语句可为用户会话选择当前数据库。

12.11.2.116.1 语法图

UseStmt:

图 367: UseStmt

DBName:

图 368: DBName

12.11.2.116.2 示例

```
USE mysql;
```

```
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
```

```
SHOW TABLES;
```

```
+-------------------------+
| Tables_in_mysql         |
+-------------------------+
| GLOBAL_VARIABLES        |
| bind_info               |
| columns_priv            |
| db                      |
| default_roles           |
```

```
| expr_pushdown_blacklist |
| gc_delete_range         |
| gc_delete_range_done    |
| global_priv             |
| help_topic              |
| opt_rule_blacklist      |
| role_edges              |
| stats_buckets           |
| stats_feedback          |
| stats_histograms        |
| stats_meta              |
| stats_top_n             |
| tables_priv             |
| tidb                    |
| user                    |
+-------------------------+
20 rows in set (0.01 sec)
```

```
CREATE DATABASE newtest;
```

```
Query OK, 0 rows affected (0.10 sec)
```

```
USE newtest;
```

```
Database changed
```

```
SHOW TABLES;
```

```
Empty set (0.00 sec)
```

```
CREATE TABLE t1 (a int);
```

```
Query OK, 0 rows affected (0.10 sec)
```

```
SHOW TABLES;
```

```
+-------------------+
| Tables_in_newtest |
+-------------------+
| t1                |
+-------------------+
1 row in set (0.00 sec)
```

12.11.2.116.3 MySQL 兼容性

USE 语句与 MySQL 完全兼容。如发现任何兼容性差异，请在 GitHub 上提交 issue。

12.11.2.116.4 另请参阅

- CREATE DATABASE
- SHOW TABLES

12.11.3 数据类型

12.11.3.1 数据类型概述

TiDB 支持除空间类型 (SPATIAL) 之外的所有 MySQL 数据类型，包括数值型类型、字符串类型、时间和日期类型、JSON 类型。

数据类型定义一般为 `T(M[, D])`，其中：

- `T` 表示具体的类型。
- `M` 在整数类型中表示最大显示长度；在浮点数或者定点数中表示精度；在字符类型中表示最大长度。`M` 的最大值取决于具体的类型。
- `D` 表示浮点数、定点数的小数位长度。
- `fsp` 在时间和日期类型里的 `TIME`、`DATETIME` 以及 `TIMESTAMP` 中表示秒的精度，其取值范围是 `0` 到 `6`。值为 `0` 表示没有小数部分。如果省略，则默认精度为 `0`。

12.11.3.2 数据类型的默认值

在一个数据类型描述中的 `DEFAULT value` 段描述了一个列的默认值。这个默认值必须是常量，不可以是一个函数或者是表达式。但是对于时间类型，可以例外的使用 `NOW`、`CURRENT_TIMESTAMP`、`LOCALTIME`、`LOCALTIMESTAMP` 等函数作为 `DATETIME` 或者 `TIMESTAMP` 的默认值。

`BLOB`、`TEXT` 以及 `JSON` 不可以设置默认值。

如果一个列的定义中没有 `DEFAULT` 的设置。TiDB 按照如下的规则决定：

- 如果该类型可以使用 `NULL` 作为值，那么这个列会在定义时添加隐式的默认值设置 `DEFAULT NULL`。
- 如果该类型无法使用 `NULL` 作为值，那么这个列在定义时不会添加隐式的默认值设置。

对于一个设置了 `NOT NULL` 但是没有显式设置 `DEFAULT` 的列，当 `INSERT`、`REPLACE` 没有涉及到该列的值时，TiDB 根据当时的 `SQL_MODE` 进行不同的行为：

- 如果此时是 `strict sql mode`，在事务中的语句会导致事务失败并回滚，非事务中的语句会直接报错。
- 如果此时不是 `strict sql mode`，TiDB 会为这列赋值为列数据类型的隐式默认值。

此时隐式默认值的设置按照如下规则：

- 对于数值类型，它们的默认值是 `0`。当有 `AUTO_INCREMENT` 参数时，默认值会按照增量情况赋予正确的值。
- 对于除了时间戳外的日期时间类型，默认值会是该类型的“零值”。时间戳类型的默认值会是当前的时间。
- 对于除枚举以外的字符串类型，默认值会是空字符串。对于枚举类型，默认值是枚举中的第一个值。

12.11.3.3 数值类型

TiDB 支持 MySQL 所有的数值类型，按照精度可以分为：

- 整数类型（精确值）
- 浮点类型（近似值）
- 定点类型（精确值）

12.11.3.3.1 整数类型

TiDB 支持 MySQL 所有的整数类型，包括 INTEGER/INT、 TINYINT、 SMALLINT、 MEDIUMINT 以及 BIGINT，完整信息参考这篇文档。

字段说明：

语法元素	说明
M	类型显示宽度，可选
UNSIGNED	无符号数，如果不加这个标识，则为有符号数
ZEROFILL	补零标识，如果有这个标识，TiDB 会自动给类型增加 UNSIGNED 标识，但是没有做补零的操作

类型定义

BIT 类型

比特值类型。M 表示比特位的长度，取值范围从 1 到 64，其默认值是 1。

```
BIT[(M)]
```

BOOLEAN 类型

布尔类型，别名为 BOOL，和 TINYINT(1) 等价。零值被认为是 False，非零值认为是 True。在 TiDB 内部，True 存储为 1，False 存储为 0。

```
BOOLEAN
```

TINYINT 类型

TINYINT 类型。有符号数的范围是 [-128, 127]。无符号数的范围是 [0, 255]。

```
TINYINT[(M)] [UNSIGNED] [ZEROFILL]
```

SMALLINT 类型

SMALLINT 类型。有符号数的范围是 [-32768, 32767]。无符号数的范围是 [0, 65535]。

```
SMALLINT[(M)] [UNSIGNED] [ZEROFILL]
```

MEDIUMINT 类型

MEDIUMINT 类型。有符号数的范围是 [-8388608, 8388607]。无符号数的范围是 [0, 16777215]。

```
MEDIUMINT[(M)] [UNSIGNED] [ZEROFILL]
```

INTEGER 类型

INTEGER 类型，别名 INT。有符号数的范围是 [-2147483648，2147483647]。无符号数的范围是 [0，4294967295]。

```
INT[(M)] [UNSIGNED] [ZEROFILL]
```

或者：

```
INTEGER[(M)] [UNSIGNED] [ZEROFILL]
```

BIGINT 类型

BIGINT 类型。有符号数的范围是 [-9223372036854775808，9223372036854775807]。无符号数的范围是 [0，
↪ 18446744073709551615]。

```
BIGINT[(M)] [UNSIGNED] [ZEROFILL]
```

存储空间以及取值范围

每种类型对存储空间的需求以及最大/最小值如下表所示：

类型	存储空间	最小值 (有符号/无符号)	最大值 (有符号/无符号)
TINYINT	1	-128 / 0	127 / 255
SMALLINT	2	-32768 / 0	32767 / 65535
MEDIUMINT	3	-8388608 / 0	8388607 / 16777215
INT	4	-2147483648 / 0	2147483647 / 4294967295
BIGINT	8	-9223372036854775808 / 0	9223372036854775807 / 18446744073709551615

12.11.3.3.2 浮点类型

TiDB 支持 MySQL 所有的浮点类型，包括 FLOAT、DOUBLE，完整信息参考这篇文档。

字段说明：

语法元素	说明
M	小数总位数
D	小数点后位数
UNSIGNED	无符号数，如果不加这个标识，则为有符号数
ZEROFILL	补零标识，如果有这个标识，TiDB 会自动给类型增加 UNSIGNED 标识

类型定义

FLOAT 类型

单精度浮点数。允许的值范围为 -2^128 ~ +2^128，也即 -3.402823466E+38 到 -1.175494351E-38、0 和 1.175494351E-38 到 3.402823466E+38。这些是基于 IEEE 标准的理论限制。实际的范围根据硬件或操作系统的不同可能稍微小些。

FLOAT(p) 类型中，p 表示精度（以位数表示）。只使用该值来确定是否结果列的数据类型为 FLOAT 或 DOUBLE。如果 p 为从 0 到 24，数据类型变为没有 M 或 D 值的 FLOAT。如果 p 为从 25 到 53，数据类型变为没有 M 或 D 值的 DOUBLE。结果列范围与本节前面描述的单精度 FLOAT 或双精度 DOUBLE 数据类型相同。

```
FLOAT[(M,D)] [UNSIGNED] [ZEROFILL]
FLOAT(p) [UNSIGNED] [ZEROFILL]
```

> 注意：
>
> - 与在 MySQL 中一样，FLOAT 数据类型用于存储近似值。对于类似货币的精确值，建议使用 DECIMAL 类型。
> - 在 TiDB 中，FLOAT 数据类型默认的精度是 8 位，这与 MySQL 不同。在 MySQL 中，FLOAT 数据类型默认的精度是 6 位。例如，同时往 MySQL 和 TiDB 中类型为 FLOAT 的列中插入数据 123456789 和 1.23456789。在 MySQL 中查询对应的值，将会得到结果 123457000 和 1.23457。而在 TiDB 中查询对应的值，将会得到 123456790 和 1.2345679。

DOUBLE 类型

双精度浮点数，别名为 DOUBLE PRECISION。允许的值范围为：-2^1024 ~ +2^1024，也即是 -1.7976931348623157E+308 到 -2.2250738585072014E-308、0 和 2.2250738585072014E-308 到 1.7976931348623157E+308。这些是基于 IEEE 标准的理论限制。实际的范围根据硬件或操作系统的不同可能稍微小些。

```
DOUBLE[(M,D)] [UNSIGNED] [ZEROFILL]
DOUBLE PRECISION [(M,D)] [UNSIGNED] [ZEROFILL], REAL[(M,D)] [UNSIGNED] [ZEROFILL]
```

> 警告：
>
> 与在 MySQL 中一样，DOUBLE 数据类型存储近似值。对于货币之类的精确值，建议使用 DECIMAL 类型。

存储空间

每种类型对存储空间的需求如下表所示：

类型	存储空间
FLOAT	4
FLOAT(p)	如果 0 <= p <= 24 为 4 个字节, 如果 25 <= p <= 53 为 8 个字节
DOUBLE	8

12.11.3.3.3 定点类型

TiDB 支持 MySQL 所有的定点类型，包括 DECIMAL、NUMERIC，完整信息参考这篇文档。

字段说明：

语法元素	说明
M	小数总位数
D	小数点后位数
UNSIGNED	无符号数，如果不加这个标识，则为有符号数
ZEROFILL	补零标识，如果有这个标识，TiDB 会自动给类型增加 UNSIGNED 标识

类型定义

DECIMAL 类型

定点数，别名为 NUMERIC。M 是小数位数（精度）的总数，D 是小数点（标度）后面的位数。小数点和 -（负数）符号不包括在 M 中。如果 D 是 0，则值没有小数点或分数部分。如果 D 被省略，默认是 0。如果 M 被省略，默认是 10。

```
DECIMAL[(M[,D])] [UNSIGNED] [ZEROFILL]
NUMERIC[(M[,D])] [UNSIGNED] [ZEROFILL]
```

12.11.3.4 日期和时间类型

TiDB 支持 MySQL 所有的日期和时间类型，包括DATE、TIME、DATETIME、TIMESTAMP 以及YEAR。完整信息可以参考 MySQL 中的时间和日期类型。

每种类型都有有效值范围，值为 0 表示无效值。此外，TIMESTAMP 和 DATETIME 类型能自动生成新的时间值。

关于日期和时间值类型，需要注意：

- 日期部分必须是“年 -月 -日”的格式（例如 1998-09-04），而不是“月 -日 -年”或“日 -月 -年”的格式。
- 如果日期的年份部分是 2 位数，TiDB 会根据年份为两位数的具体规则进行转换。
- 如果格式必须是数值类型，TiDB 会自动将日期或时间值转换为数值类型。例如：

```
SELECT NOW(), NOW()+0, NOW(3)+0;
```

```
+---------------------+----------------+--------------------+
| NOW()               | NOW()+0        | NOW(3)+0           |
+---------------------+----------------+--------------------+
| 2012-08-15 09:28:00 | 20120815092800 | 20120815092800.889 |
+---------------------+----------------+--------------------+
```

- TiDB 可以自动将无效值转换同一类型的零值。是否进行转换取决于 SQL 模式的设置。比如：

```
show create table t1;
```

```
+--
    ↪ -----+-------------------------------------------------------------------------------------------------
    ↪
```

```
| Table | Create Table
    ↪
    ↪ |
+--
    ↪ -----+----------------------------------------------------------------------------------------------------
    ↪
| t1    | CREATE TABLE `t1` (
  `a` time DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin |
+--
    ↪ -----+----------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

```
select @@sql_mode;
```

```
+--
    ↪ ----------------------------------------------------------------------------------------------------
    ↪
| @@sql_mode
    ↪
    ↪ |
+--
    ↪ ----------------------------------------------------------------------------------------------------
    ↪
| ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,
    ↪ ERROR_FOR_DIVISION_BY_ZERO,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION |
+--
    ↪ ----------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

```
insert into t1 values (`2090-11-32:22:33:44`);
```

```
ERROR 1292 (22007): Truncated incorrect time value: `2090-11-32:22:33:44`
```

```
set @@sql_mode=``;
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
insert into t1 values (`2090-11-32:22:33:44`);
```

```
Query OK, 1 row affected, 1 warning (0.01 sec)
```

```
select * from t1;
```

```
+----------+
| a        |
+----------+
| 00:00:00 |
+----------+
1 row in set (0.01 sec)
```

- SQL 模式的不同设置，会改变 TiDB 对格式的要求。
- 如果 SQL 模式的 NO_ZERO_DATE 被禁用，TiDB 允许 DATE 和 DATETIME 列中的月份或日期为零。例如，2009-00-00 或 2009-01-00。如果使用函数计算这种日期类型，例如使用 DATE_SUB() 或 DATE_ADD() 函数，计算结果可能不正确。
- 默认情况下，TiDB 启用 NO_ZERO_DATE SQL 模式。该模式可以避免存储像 0000-00-00 这样的零值。

不同类型的零值如下表所示：

数据类型	零值
DATE	0000-00-00
TIME	00:00:00
DATETIME	0000-00-00 00:00:00
TIMESTAMP	0000-00-00 00:00:00
YEAR	0000

如果 SQL 模式允许使用无效的 DATE、DATETIME、TIMESTAMP 值，无效值会自动转换为相应的零值（0000-00-00 或 0000-00-00 00:00:00）。

12.11.3.4.1 类型定义

DATE 类型

DATE 类型只包含日期部分，不包含时间部分。DATE 类型的格式为 YYYY-MM-DD，支持的范围是 1000-01-01 到 9999-12-31。

```
DATE
```

TIME 类型

TIME 类型的格式为 HH:MM:SS[.fraction]，支持的范围是 -838:59:59.000000 到 838:59:59.000000。TIME 不仅可用于指示一天内的时间，还可用于指两个事件之间的时间间隔。fsp 参数表示秒精度，取值范围为：0 ~ 6，默认值为 0。

```
TIME[(fsp)]
```

> **注意：**
>
> 注意 TIME 的缩写形式。例如，11:12 表示 11:12:00 而不是 00:11:12。但是，1112 表示 00:11:12。这些差异取决于：字符的存在与否。

DATETIME 类型

DATETIME 类型是日期和时间的组合，格式为 YYYY-MM-DD HH:MM:SS[.fraction]。支持的范围是 1000-01-01 00:00:00.000000 到 9999-12-31 23:59:59.999999。fsp 参数表示秒精度，取值范围为 0~6，默认值为 0。TiDB 支持字符串或数字转换为 DATETIME 类型。

```
DATETIME[(fsp)]
```

TIMESTAMP 类型

TIMESTAMP 类型是日期和时间的组合，支持的范围是 UTC 时间从 1970-01-01 00:00:01.000000 到 2038-01-19 03:14:07.999999。fsp 参数表示秒精度，取值范围为 0~6，默认值为 0。在 TIMESTAMP 中，不允许零出现在月份部分或日期部分，唯一的例外是零值本身 0000-00-00 00:00:00。

```
TIMESTAMP[(fsp)]
```

时区处理

当存储 TIMESTAMP 时，TiDB 会将当前时区的 TIMESTAMP 值转换为 UTC 时区。当读取 TIMESTAMP 时，TiDB 将存储的 TIMESTAMP 值从 UTC 时区转换为当前时区（注意：DATETIME 不会这样处理）。每次连接的默认时区是服务器的本地时区，可以通过环境变量 time_zone 进行修改。

> **警告：**
>
> 和 MySQL 一样，TIMESTAMP 数据类型受 2038 年问题的影响。如果存储的值大于 2038，建议使用 DATETIME 类型。

YEAR 类型

YEAR 类型的格式为 YYYY，支持的值范围是 1901 到 2155，也支持零值 0000。

```
YEAR[(4)]
```

YEAR 类型遵循以下格式规则：

- 如果是四位数的数值，支持的范围是 1901 至 2155。
- 如果是四位数的字符串，支持的范围是 '1901' 到 '2155'。
- 如果是 1~99 之间的一位数或两位数的数字，1~69 换算成 2001_{2069}，$_{70}99$ 换算成 1970~1999。
- 支持 '0' 到 '99' 之间的一位数或两位数字符串的范围
- 将数值 0 转换为 0000，将字符串 '0' 或 '00' 转换为 '2000'。

无效的 YEAR 值会自动转换为 0000（如果用户没有使用 NO_ZERO_DATE SQL 模式）。

12.11.3.4.2 自动初始化和更新 TIMESTAMP 或 DATETIME

带有 TIMESTAMP 或 DATETIME 数据类型的列可以自动初始化为或更新为当前时间。

对于表中任何带有 TIMESTAMP 或 DATETIME 数据类型的列，你可以设置默认值，或自动更新为当前时间戳。

在定义列的时候，TIMESTAMP 和 DATETIME 可以通过 DEFAULT CURRENT_TIMESTAMP 和 ON UPDATE CURRENT_TIMESTAMP 来设置。DEFAULT 也可以设置为一个特定的值，例如 DEFAULT 0 或 DEFAULT '2000-01-01 00:00:00'。

```
CREATE TABLE t1 (
    ts TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    dt DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);
```

除非指定 DATETIME 的值为 NOT NULL，否则默认 DATETIME 的值为 NULL。指定 DATETIME 的值为 NOT NULL 时，如果没有设置默认值，则默认值为 0。

```
CREATE TABLE t1 (
    dt1 DATETIME ON UPDATE CURRENT_TIMESTAMP,          -- default NULL
    dt2 DATETIME NOT NULL ON UPDATE CURRENT_TIMESTAMP -- default 0
);
```

12.11.3.4.3 时间值的小数部分

DATETIME 和 TIMESTAMP 值最多可以有 6 位小数，精确到毫秒。如果包含小数部分，值的格式为 YYYY-MM-DD HH:MM:SS[.fraction]，小数部分的范围为 000000 到999999。必须使用小数点分隔小数部分与其他部分。

- 使用 type_name(fsp) 可以定义精确到小数的列，其中 type_name 可以是TIME、DATETIME 或 TIMESTAMP。例如：

  ```
  CREATE TABLE t1 (t TIME(3), dt DATETIME(6));
  ```

  fsp 范围是 0 到 6。

  0 表示没有小数部分。如果省略了 fsp ，默认为 0。

- 当插入包含小数部分的 TIME、DATETIME 或 TIMESTAMP 时，如果小数部分的位数过少或过多，可能需要进行四舍五入。例如：

  ```
  CREATE TABLE fractest( c1 TIME(2), c2 DATETIME(2), c3 TIMESTAMP(2) );
  ```

  ```
  Query OK, 0 rows affected (0.33 sec)
  ```

  ```
  INSERT INTO fractest VALUES
       > ('17:51:04.777', '2014-09-08 17:51:04.777',   '2014-09-08 17:51:04.777');
  ```

  ```
  Query OK, 1 row affected (0.03 sec)
  ```

```
SELECT * FROM fractest;
```

```
+-------------|-------------------------|-------------------------+
| c1          | c2                      | c3                      |
+-------------|-------------------------|-------------------------+
| 17:51:04.78 | 2014-09-08 17:51:04.78 | 2014-09-08 17:51:04.78 |
+-------------|-------------------------|-------------------------+
1 row in set (0.00 sec)
```

12.11.3.4.4 日期和时间类型的转换

在日期和时间类型之间进行转换时，有些转换可能会导致信息丢失。例如，DATE、DATETIME 和 TIMESTAMP 都有各自的有效值范围。TIMESTAMP 不能早于 UTC 时间的 1970 年，也不能晚于 UTC 时间的 2038-01-19 03:14:07。根据这个规则，1968-01-01 对于 DATE 或 DATETIME 是有效的，但当 1968-01-01 转换为 TIMESTAMP 时，就会变成 0。

DATE 的转换：

- 当 DATE 转换为 DATETIME 或 TIMESTAMP 时，会添加时间部分 00:00:00，因为 DATE 不包含任何时间信息。
- 当 DATE 转换为 TIME 时，结果是 00:00:00。

DATETIME 或 TIMESTAMP 的转换：

- 当 DATETIME 或 TIMESTAMP 转换为 DATE 时，时间和小数部分将被舍弃。例如，1999-12-31 23:59:59.499 被转换为 1999-12-31。
- 当 DATETIME 或 TIMESTAMP 转换为 TIME 时，日期部分被舍弃，因为 TIME 不包含任何日期信息。

如果要将 TIME 转换为其他时间和日期格式，日期部分会自动指定为 CURRENT_DATE()。最终的转换结果是由 TIME 和 CURRENT_DATE() 组成的日期。也就是说，如果 TIME 的值超出了 00:00:00 到 23:59:59 的范围，那么转换后的日期部分并不表示当前的日期。

当 TIME 转换为 DATE 时，转换过程类似，时间部分被舍弃。

使用 CAST() 函数可以显式地将值转换为 DATE 类型。例如：

```
date_col = CAST(datetime_col AS DATE)
```

将 TIME 和 DATETIME 转换为数字格式。例如：

```
SELECT CURTIME(), CURTIME()+0, CURTIME(3)+0;
```

```
+-----------|-------------|--------------+
| CURTIME() | CURTIME()+0 | CURTIME(3)+0 |
+-----------|-------------|--------------+
| 09:28:00  |       92800 |    92800.887 |
+-----------|-------------|--------------+
```

```
SELECT NOW(), NOW()+0, NOW(3)+0;
```

```
+---------------------|----------------|--------------------+
| NOW()               | NOW()+0        | NOW(3)+0           |
+---------------------|----------------|--------------------+
| 2012-08-15 09:28:00 | 20120815092800 | 20120815092800.889 |
+---------------------|----------------|--------------------+
```

12.11.3.4.5 年份为两位数

如果日期中包含年份为两位数，这个年份是有歧义的，并不显式地表示实际年份。

对于 DATETIME、DATE 和 TIMESTAMP 类型，TiDB 使用如下规则来消除歧义。

- 将 01 至 69 之间的值转换为 2001 至 2069 之间的值。
- 将 70 至 99 之间的值转化为 1970 至 1999 之间的值。

上述规则也适用于 YEAR 类型，但有一个例外。将数字 00 插入到 YEAR(4) 中时，结果是 0000 而不是 2000。

如果想让结果是 2000，需要指定值为 2000。

对于 MIN() 和 MAX() 等函数，年份为两位数时可能会得到错误的计算结果。建议年份为四位数时使用这类函数。

12.11.3.5 字符串类型

TiDB 支持 MySQL 所有的字符串类型，包括 CHAR、VARCHAR、BINARY、VARBINARY、BLOB、TEXT、ENUM 以及 SET，完整信息参考这篇文档。

12.11.3.5.1 类型定义

CHAR 类型

定长字符串。CHAR 列的长度固定为创建表时声明的长度。当保存 CHAR 值时，不足固定长度的字符串在后面填充空格，以达到指定的长度。M 表示列长度（字符的个数，不是字节的个数）。长度可以为从 0 到 255 的任何值。

```
[NATIONAL] CHAR[(M)] [CHARACTER SET charset_name] [COLLATE collation_name]
```

VARCHAR 类型

变长字符串。M 表示最大列长度（字符的最大个数）。VARCHAR 的空间占用大小不得超过 65535 字节。在选择 VARCHAR 长度时，应当根据最长的行的大小和使用的字符集确定。

对于不同的字符集，单个字符所占用的空间可能有所不同。以下表格是各个字符集下单个字符占用的字节数，以及 VARCHAR 列长度的取值范围：

字符集	单个字符字节数	VARCHAR 最大列长度的取值范围
ascii	1	(0, 65535]
latin1	1	(0, 65535]
binary	1	(0, 65535]
utf8	3	(0, 21845]
utf8mb4	4	(0, 16383]

```
[NATIONAL] VARCHAR(M) [CHARACTER SET charset_name] [COLLATE collation_name]
```

TEXT 类型

文本串。M 表示最大列长度（字符的最大个数），范围是 0 到 65535。在选择 TEXT 长度时，应当根据最长的行的大小和使用的字符集确定。

```
TEXT[(M)] [CHARACTER SET charset_name] [COLLATE collation_name]
```

TINYTEXT 类型

类似于TEXT，区别在于最大列长度为 255。

```
TINYTEXT [CHARACTER SET charset_name] [COLLATE collation_name]
```

MEDIUMTEXT 类型

类似于TEXT，区别在于最大列长度为 16,777,215。

```
MEDIUMTEXT [CHARACTER SET charset_name] [COLLATE collation_name]
```

LONGTEXT 类型

类似于TEXT，区别在于最大列长度为 4,294,967,295。但由于TiDB 单行的限制，TiDB 中单行存储最大不超过 6 MB。

```
LONGTEXT [CHARACTER SET charset_name] [COLLATE collation_name]
```

BINARY 类型

类似于CHAR，区别在于 BINARY 存储的是二进制字符串。

```
BINARY(M)
```

VARBINARY 类型

类似于VARCHAR，区别在于 VARBINARY 存储的是二进制字符串。

```
VARBINARY(M)
```

BLOB 类型

二进制大文件。M 表示最大列长度，单位是字节，范围是 0 到 65535。

```
BLOB[(M)]
```

TINYBLOB 类型

类似于BLOB，区别在于最大列长度为 255。

```
TINYBLOB
```

MEDIUMBLOB 类型

类似于BLOB，区别在于最大列长度为 16777215。

```
MEDIUMBLOB
```

LONGBLOB 类型

类似于BLOB，区别在于最大列长度为 4,294,967,295。但由于TiDB 单行的限制，TiDB 中单行存储最大不超过 6 MB。

```
LONGBLOB
```

ENUM 类型

枚举类型是一个字符串，它只能有一个值的字符串对象。其值必须是从一个固定集合中选取，这个固定集合在创建表的时候定义，语法是：

```
ENUM('value1','value2',...) [CHARACTER SET charset_name] [COLLATE collation_name]
```

例如：

```
ENUM('apple', 'orange', 'pear')
```

枚举类型的值在 TiDB 内部使用数值来存储，每个值会按照定义的顺序转换为一个数字，比如上面的例子中，每个字符串值都会映射为一个数字：

值	数字
NULL	NULL
”	0
‘apple’	1
‘orange’	2
‘pear’	3

更多信息参考 MySQL 枚举文档。

SET 类型

集合类型是一个包含零个或多个值的字符串，其中每个值必须是从一个固定集合中选取，这个固定集合在创建表的时候定义，语法是：

```
SET('value1','value2',...) [CHARACTER SET charset_name] [COLLATE collation_name]
```

例如：

```
SET('1', '2') NOT NULL
```

上面的例子中，这列的有效值可以是：

```
''
'1'
'2'
'1,2'
```

集合类型的值在 TiDB 内部会转换为一个 Int64 数值，每个元素是否存在用一个二进制位的 0/1 值来表示，比如这个例子 SET('a','b','c','d')，每一个元素都被映射为一个数字，且每个数字的二进制表示只会有一位是 1：

成员	十进制表示	二进制表示
‘a’	1	0001
‘b’	2	0010
‘c’	4	0100
‘d’	8	1000

这样对于值为 ('a', 'c') 的元素，其二进制表示即为 0101。

更多信息参考 MySQL 集合文档。

12.11.3.6 JSON 类型

> **警告：**
>
> 当前该功能为实验特性，不建议在生产环境中使用。

JSON 类型可以存储 JSON 这种半结构化的数据，相比于直接将 JSON 存储为字符串，它的好处在于：

1. 使用 Binary 格式进行序列化，对 JSON 的内部字段的查询、解析加快；
2. 多了 JSON 合法性验证的步骤，只有合法的 JSON 文档才可以放入这个字段中；

JSON 字段本身上，并不能创建索引。相反，可以对 JSON 文档中的某个子字段创建索引。例如：

```
CREATE TABLE city (
    id INT PRIMARY KEY,
    detail JSON,
    population INT AS (JSON_EXTRACT(detail, '$.population')),
    index index_name (population)
);
INSERT INTO city (id,detail) VALUES (1, '{"name": "Beijing", "population": 100}');
SELECT id FROM city WHERE population >= 100;
```

有关 JSON 的更多信息，可以参考JSON 函数和生成列。

12.11.4 函数与操作符

12.11.4.1 函数和操作符概述

TiDB 中函数和操作符使用方法与 MySQL 基本一致，详情参见: Functions and Operators。

在 SQL 语句中，表达式可用于诸如 SELECT 语句的 ORDER BY 或 HAVING 子句，SELECT/DELETE/UPDATE 语句的 WHERE 子句，或 SET 语句之类的地方。

可使用字面值，列名，NULL，内置函数，操作符等来书写表达式。其中有些表达式下推到 TiKV 上执行，详见下推到 TiKV 的表达式列表。

12.11.4.2 表达式求值的类型转换

TiDB 中表达式求值的类型转换与 MySQL 基本一致，详情参见 MySQL 表达式求值的类型转换。

12.11.4.3 操作符

操作符名	功能描述
AND, &&	逻辑与
=	赋值 (可用于 SET 语句中, 或用于 UPDATE 语句的 SET 中)
:=	赋值
BETWEEN ... AND ...	判断值满足范围
BINARY	将一个字符串转换为一个二进制字符串
&	位与
~	位非
|	位或
^	按位异或
CASE	case 操作符
DIV	整数除
/	除法
=	相等比较
<=>	空值安全型相等比较
>	大于
>=	大于或等于
IS	判断一个值是否等于一个布尔值
IS NOT	判断一个值是否不等于一个布尔值
IS NOT NULL	非空判断
IS NULL	空值判断
<<	左移
<	小于
<=	小于或等于
LIKE	简单模式匹配
-	减
lstinlineMOD!	求余
NOT, !	取反
NOT BETWEEN ... AND ...	判断值是否不在范围内

操作符名	功能描述
!= , <>	不等于
NOT LIKE	不符合简单模式匹配
NOT REGEXP	不符合正则表达式模式匹配
|| , OR	逻辑或
+	加
REGEXP	使用正则表达式进行模式匹配
>>	右移
RLIKE	REGEXP 同义词
*	乘
-	取反符号
XOR	逻辑亦或

12.11.4.3.1 操作符优先级

操作符优先级显示在以下列表中，从最高优先级到最低优先级。同一行显示的操作符具有相同的优先级。

```
INTERVAL
BINARY
!
- (unary minus), ~ (unary bit inversion)
^
*, /, DIV, %, MOD
-, +
<<, >>
&
|
= (comparison), <=>, >=, >, <=, <, <>, !=, IS, LIKE, REGEXP, IN
BETWEEN, CASE, WHEN, THEN, ELSE
NOT
AND, &&
XOR
OR, ||
= (assignment), :=
```

详情参见 这里.

12.11.4.3.2 比较方法和操作符

操作符名	功能描述
BETWEEN ... AND ...	判断值是否在范围内
COALESCE()	返回第一个非空值
=	相等比较
<=>	空值安全型相等比较

操作符名	功能描述
>	大于
>=	大于或等于
GREATEST()	返回最大值
IN()	判断值是否在一个值的集合内
INTERVAL()	返回一个小于第一个参数的参数的下标
IS	判断是否等于一个布尔值
IS NOT	判断是否不等于一个布尔值
IS NOT NULL	非空判断
IS NULL	空值判断
ISNULL()	判断参数是否为空
LEAST()	返回最小值
<	小于
<=	小于或等于
LIKE	简单模式匹配
NOT BETWEEN ... AND ...	判断值是否不在范围内
!=, <>	不等于
NOT IN()	判断值是否不在一个值的集合内
NOT LIKE	不满足简单模式匹配
STRCMP()	比较两个字符串

详情参见 这里.

12.11.4.3.3 逻辑操作符

操作符名	功能描述
AND, &&	逻辑与
NOT, !	逻辑非
||, OR	逻辑或
XOR	逻辑异或

详情参见 这里.

12.11.4.3.4 赋值操作符

操作符名	功能描述
=	赋值 (可用于 SET 语句中, 或用于 UPDATE 语句的 SET 中)
:=	赋值

详情参见 这里.

12.11.4.4 控制流程函数

TiDB 支持使用 MySQL 5.7 中提供的所有控制流程函数。

函数名	功能描述
CASE	Case 操作符
IF()	构建 if/else
IFNULL()	构建 Null if/else
NULLIF()	如果 expr1 = expr2，返回 NULL

12.11.4.5 字符串函数

TiDB 支持使用 MySQL 5.7 中提供的大部分字符串函数。

12.11.4.5.1 支持的函数

函数名	功能描述
ASCII()	返回最左字符的数值
BIN()	返回一个数的二进制值的字符串表示
BIT_LENGTH()	返回字符串的位长度
CHAR()	返回由整数的代码值所给出的字符组成的字符串
CHAR_LENGTH()	返回字符串的字符长度
CHARACTER_LENGTH()	与 CHAR_LENGTH() 功能相同
CONCAT()	返回连接的字符串
CONCAT_WS()	返回由分隔符连接的字符串
ELT()	返回指定位置的字符串
EXPORT_SET()	返回一个字符串，其中值位中设置的每个位，可以得到一个 on 字符串，而每个未设置的位，可以得到
FIELD()	返回参数在后续参数中出现的第一个位置
FIND_IN_SET()	返回第一个参数在第二个参数中出现的位置
FORMAT()	返回指定小数位数格式的数字
FROM_BASE64()	解码 base-64 表示的字符串，并返回结果
HEX()	返回一个十进制数或字符串值的 16 进制表示
INSERT()	在指定位置插入一个子字符串，最多不超过指定字符数
INSTR()	返回第一次出现的子字符串的索引
LCASE()	与 LOWER() 功能相同
LEFT()	返回最左侧指定长度的字符
LENGTH()	返回字符串长度，单位为字节
LIKE	进行简单模式匹配
LOCATE()	返回第一次出现的子字符串的位置
LOWER()	返回全小写的参数
LPAD()	返回字符串参数，左侧添加指定字符串
LTRIM()	去掉前缀空格
MAKE_SET()	返回一组用逗号分隔的字符串，这些字符串的位数与给定的 bits 参数对应
MID()	返回一个以指定位置开始的子字符串
NOT LIKE	否定简单模式匹配

函数名	功能描述
NOT REGEXP	REGEXP 的否定形式
OCT()	返回一个数值的八进制表示，形式为字符串
OCTET_LENGTH()	与 LENGTH() 功能相同
ORD()	返回该参数最左侧字符的字符编码
POSITION()	与 LOCATE() 功能相同
QUOTE()	使参数逃逸，为了在 SQL 语句中使用
REGEXP	使用正则表达式匹配模式
REPEAT()	以指定次数重复一个字符串
REPLACE()	替换所有出现的指定字符串
REVERSE()	反转字符串里的所有字符
RIGHT()	返回指定数量的最右侧的字符
RLIKE	与 REGEXP 功能相同
RPAD()	以指定次数添加字符串
RTRIM()	去掉后缀空格
SPACE()	返回指定数量的空格，形式为字符串
STRCMP()	比较两个字符串
SUBSTR()	返回指定的子字符串
SUBSTRING()	返回指定的子字符串
SUBSTRING_INDEX()	从一个字符串中返回指定出现次数的定界符之前的子字符串
TO_BASE64()	返回转化为 base-64 表示的字符串参数
TRIM()	去掉前缀和后缀空格
UCASE()	与 UPPER() 功能相同
UNHEX()	返回一个数的十六进制表示，形式为字符串
UPPER()	参数转换为大写形式

12.11.4.5.2 不支持的函数

- LOAD_FILE()
- MATCH
- SOUNDEX()
- SOUNDS LIKE
- WEIGHT_STRING()

12.11.4.6 数值函数与操作符

TiDB 支持使用 MySQL 5.7 中提供的所有数值函数与操作符。

12.11.4.6.1 算术操作符

操作符名	功能描述
+	加号
-	减号

操作符名	功能描述
*	乘号
/	除号
DIV	整数除法
lstinlineMOD!	模运算，取余
-	更改参数符号

12.11.4.6.2 数学函数

函数名	功能描述
POW()	返回参数的指定乘方的结果值
POWER()	返回参数的指定乘方的结果值
EXP()	返回 e（自然对数的底）的指定乘方后的值
SQRT()	返回非负数的二次方根
LN()	返回参数的自然对数
LOG()	返回第一个参数的自然对数
LOG2()	返回参数以 2 为底的对数
LOG10()	返回参数以 10 为底的对数
PI()	返回 pi 的值
TAN()	返回参数的正切值
COT()	返回参数的余切值
SIN()	返回参数的正弦值
COS()	返回参数的余弦值
ATAN()	返回参数的反正切值
ATAN2(), ATAN()	返回两个参数的反正切值
ASIN()	返回参数的反正弦值
ACOS()	返回参数的反余弦值
RADIANS()	返回由度转化为弧度的参数
DEGREES()	返回由弧度转化为度的参数
MOD()	返回余数
ABS()	返回参数的绝对值
CEIL()	返回不小于参数的最小整数值
CEILING()	返回不小于参数的最小整数值
FLOOR()	返回不大于参数的最大整数值
ROUND()	返回参数最近似的整数或指定小数位数的数值
RAND()	返回一个随机浮点值
SIGN()	返回参数的符号
CONV()	不同数基间转换数字，返回数字的字符串表示
TRUNCATE()	返回被舍位至指定小数位数的数字
CRC32()	计算循环冗余码校验值并返回一个 32 位无符号值

12.11.4.7 日期和时间函数

TiDB 支持使用 MySQL 5.7 中提供的所有日期和时间函数。

12.11.4.7.1 日期时间函数表

函数名	功能描述
ADDDATE()	将时间间隔添加到日期上
ADDTIME()	时间数值相加
CONVERT_TZ()	转换时区
CURDATE()	返回当前日期
CURRENT_DATE(), CURRENT_DATE	与 CURDATE() 同义
CURRENT_TIME(), CURRENT_TIME	与 CURTIME() 同义
CURRENT_TIMESTAMP(), CURRENT_TIMESTAMP	与 NOW() 同义
CURTIME()	返回当前时间
DATE()	从日期或日期/时间表达式中提取日期部分
DATE_ADD()	将时间间隔添加到日期上
DATE_FORMAT()	返回满足指定格式的日期/时间
DATE_SUB()	从日期减去指定的时间间隔
DATEDIFF()	返回两个日期间隔的天数
DAY()	与 DAYOFMONTH() 同义
DAYNAME()	返回星期名称
DAYOFMONTH()	返回参数对应的天数部分 (1-31)
DAYOFWEEK()	返回参数对应的星期下标
DAYOFYEAR()	返回参数代表一年的哪一天 (1-366)
EXTRACT()	提取日期/时间中的单独部分
FROM_DAYS()	将天数转化为日期
FROM_UNIXTIME()	将 Unix 时间戳格式化为日期
GET_FORMAT()	返回满足日期格式的字符串
HOUR()	提取日期/时间表达式中的小时部分
LAST_DAY	返回参数中月份的最后一天
LOCALTIME(), LOCALTIME	与 NOW() 同义
LOCALTIMESTAMP, LOCALTIMESTAMP()	与 NOW() 同义
MAKEDATE()	根据给定的年份和一年中的天数生成一个日期
MAKETIME()	根据给定的时、分、秒生成一个时间
MICROSECOND()	返回参数的微秒部分
MINUTE()	返回参数的分钟部分
MONTH()	返回参数的月份部分
MONTHNAME()	返回参数的月份名称
NOW()	返回当前日期和时间
PERIOD_ADD()	在年-月表达式上添加一段时间 (数个月)
PERIOD_DIFF()	返回间隔的月数
QUARTER()	返回参数对应的季度 (1-4)
SEC_TO_TIME()	将秒数转化为 ‘HH:MM:SS’ 的格式
SECOND()	返回秒数 (0-59)
STR_TO_DATE()	将字符串转化为日期
SUBDATE()	当传入三个参数时作为 DATE_SUB() 的同义

函数名	功能描述
SUBTIME()	从一个时间中减去一段时间
SYSDATE()	返回该方法执行时的时间
TIME()	返回参数的时间表达式部分
TIME_FORMAT()	格式化时间
TIME_TO_SEC()	返回参数对应的秒数
TIMEDIFF()	返回时间间隔
TIMESTAMP()	传入一个参数时候, 该方法返回日期或日期/时间表达式, 传入两个参数时候, 返回参
TIMESTAMPADD()	在日期/时间表达式上增加一段时间间隔
TIMESTAMPDIFF()	从日期/时间表达式中减去一段时间间隔
TO_DAYS()	将参数转化对应的天数 (从第 0 年开始)
TO_SECONDS()	将日期或日期/时间参数转化为秒数 (从第 0 年开始)
UNIX_TIMESTAMP()	返回一个 Unix 时间戳
UTC_DATE()	返回当前的 UTC 日期
UTC_TIME()	返回当前的 UTC 时间
UTC_TIMESTAMP()	返回当前的 UTC 日期和时间
WEEK()	返回参数所在的一年中的星期数
WEEKDAY()	返回星期下标
WEEKOFYEAR()	返回参数在日历中对应的一年中的星期数
YEAR()	返回参数对应的年数
YEARWEEK()	返回年数和星期数

12.11.4.8 位函数和操作符

TiDB 支持使用 MySQL 5.7 中提供的所有位函数和操作符。

位函数和操作符表

函数和操作符名	功能描述
BIT_COUNT()	返回参数二进制表示中为 1 的个数
&	位与
~	按位取反
|	位或
[^](https://dev.mysql.com/doc/refman/5.7/en/bit-functions.html#operator_bitwise-xor)	位异或
<<	左移
>>	右移

12.11.4.9 Cast 函数和操作符

Cast 函数和操作符用于将某种数据类型的值转换为另一种数据类型。TiDB 支持使用 MySQL 5.7 中提供的所有 Cast 函数和操作符。

12.11.4.9.1 Cast 函数和操作符表

函数和操作符名	功能描述
BINARY	将一个字符串转换成一个二进制字符串
CAST()	将一个值转换成一个确定类型
CONVERT()	将一个值转换成一个确定类型

12.11.4.10 加密和压缩函数

TiDB 支持使用 MySQL 5.7 中提供的大部分加密和压缩函数。

12.11.4.10.1 支持的函数

函数名	功能描述
MD5()	计算字符串的 MD5 校验和
PASSWORD()	计算并返回密码字符串
RANDOM_BYTES()	返回随机字节向量
SHA1(), SHA()	计算 SHA-1 160 位校验和
SHA2()	计算 SHA-2 校验和
AES_DECRYPT()	使用 AES 解密
AES_ENCRYPT()	使用 AES 加密
COMPRESS()	返回经过压缩的二进制字符串
UNCOMPRESS()	解压缩字符串
UNCOMPRESSED_LENG ↪ ()	返回字符串解压后的长度

12.11.4.10.2 不支持的函数

- DES_DECRYPT()、DES_ENCRYPT()、OLD_PASSWORD() 和 ENCRYPT()：这些函数在 MySQL 5.7 中被废弃，并且已在 MySQL 8.0 中移除。
- VALIDATE_PASSWORD_STRENGTH() 函数。
- 只在 MySQL 企业版中支持的函数。见 Issue #2632。

12.11.4.11 信息函数

TiDB 支持使用 MySQL 5.7 中提供的大部分信息函数。

12.11.4.11.1 支持的函数

函数名	功能描述
BENCHMARK()	循环执行一个表达式
CONNECTION_ID()	返回当前连接的连接 ID (线程 ID)
CURRENT_USER(), CURRENT_USER	返回当前用户的用户名和主机名
DATABASE()	返回默认 (当前) 的数据库名
FOUND_ROWS()	该函数返回对于一个包含 LIMIT 的 SELECT 查询语句，在不包含 LIMIT 的情况下回返回的记录数
LAST_INSERT_ID()	返回最后一条 INSERT 语句中自增列的值
ROW_COUNT()	影响的行数
SCHEMA()	与 DATABASE() 同义
SESSION_USER()	与 USER() 同义
SYSTEM_USER()	与 USER() 同义
USER()	返回客户端提供的用户名和主机名
VERSION()	返回当前 MySQL 服务器的版本信息
TIDB_VERSION()	返回当前 TiDB 服务器的版本信息

12.11.4.11.2 不支持的函数

- CHARSET()
- COERCIBILITY()
- COLLATION()

12.11.4.12 JSON 函数

> **警告：**
>
> 当前该功能为实验特性，不建议在生产环境中使用。

TiDB 支持 MySQL 5.7 GA 版本发布的大多数 JSON 函数。MySQL 5.7 发布后，又增加了更多 JSON 函数，TiDB 并未支持所有这些函数 (参见未支持的函数)。

12.11.4.12.1 创建 JSON 值的函数

函数	功能描述
JSON_ARRAY([val[, val] ···])	根据一系列元素创建一个 JSON 文档
JSON_OBJECT(key, val[, key, val] ···)	根据一系列 K/V 对创建一个 JSON 文档
JSON_QUOTE(string)	返回一个字符串，该字符串为带引号的 JSON 值

12.11.4.12.2 搜索 JSON 值的函数

函数	功能描述
JSON_CONTAINS(target, candidate[, path])	通过返回 1 或 0 来表示目标 JSON 文档中是否包含给定的 candidate JSON 文档
JSON_CONTAINS_PATH(json_doc, one_or_all, path[, path] ⋯)	通过返回 0 或 1 来表示一个 JSON 文档在给定路径是否包含数据
JSON_EXTRACT(json_doc, path[, path] ⋯)	从 JSON 文档中解出某一路径对应的子文档
->	返回执行路径后面的 JSON 列的值；JSON_EXTRACT(doc, path_literal) 的别名
->>	返回执行路径后面的 JSON 列的值和转义后的结果；JSON_UNQUOTE(JSON_EXTRACT(doc, path_literal)) 的别名
JSON_KEYS(json_doc[, path])	返回从 JSON 对象的顶级值作为 JSON array 的键，如果给定了路径参数，则从选定路径中获取顶级键
JSON_SEARCH(json_doc, one_or_all, search_str[, escape_char[, path] ⋯])	返回指定字符在 JSON 文档中的路径

12.11.4.12.3 修改 JSON 值的函数

函数	功能描述
JSON_APPEND(json_doc, path, value)	JSON_ARRAY_APPEND 的别名
JSON_ARRAY_APPEND(json_doc, path, value)	将值追加到指定路径的 JSON 数组的末尾
JSON_ARRAY_INSERT(json_doc, path, val[, path, val] ⋯)	将数组插入 JSON 文档，并返回修改后的文档
JSON_INSERT(json_doc, path, val[, path, val] ⋯)	在 JSON 文档中在某一路径下插入子文档
JSON_MERGE(json_doc, json_doc[, json_doc] ⋯)	已废弃的 JSON_MERGE_PRESERVE 别名
JSON_MERGE_PRESERVE(json_doc, json_doc[, json_doc] ⋯)	将两个或多个 JSON 文档合并成一个文档，并返回合并结果
JSON_REMOVE(json_doc, path[, path] ⋯)	移除 JSON 文档中某一路径下的子文档

函数	功能描述
JSON_REPLACE(json_doc, path, val[, path, val] ⋯)	替换 JSON 文档中的某一路径下的子文档
JSON_SET(json_doc, path, val[, path, val] ⋯)	在 JSON 文档中为某一路径设置子文档
JSON_UNQUOTE(json_val)	去掉 JSON 值外面的引号，返回结果为字符串
JSON_ARRAY_APPEND(json_doc, path, val[, path, val] ⋯)	将值添加到 JSON 文档指定数组的末尾，并返回添加结果
JSON_ARRAY_INSERT(json_doc, path, val[, path, val] ⋯)	将值插入到 JSON 文档的指定位置，并返回插入结果

12.11.4.12.4 返回 JSON 值属性的函数

函数	功能描述
JSON_DEPTH(json_doc)	返回 JSON 文档的最大深度
JSON_LENGTH(json_doc[, path])	返回 JSON 文档的长度；如果路径参数已定，则返回该路径下值的长度
JSON_TYPE(json_val)	检查某 JSON 文档内部内容的类型
JSON_VALID(json_doc)	检查 JSON 文档内容是否有效；用于将列转换为 JSON 类型之前对该列进行检查

12.11.4.12.5 效用函数

函数	功能描述
[JSON_STORAGE_SIZE(json_doc)][json_storage_size]	返回存储 JSON 值所需的大致字节大小，由于不考虑 TiKV 压缩的字节大小，因此函数的输出与 MySQL 不严格兼容

12.11.4.12.6 聚合函数

函数	功能描述
[JSON_OBJECTAGG(key, value)][json_objectagg]	提供给定键的值的聚合

12.11.4.12.7 未支持的函数

TiDB 暂未支持以下 JSON 函数。相关进展参见 TiDB #7546:

- JSON_MERGE_PATCH
- JSON_PRETTY
- JSON_ARRAYAGG

12.11.4.12.8 另请参阅

- JSON Function Reference
- JSON Data Type

12.11.4.13 GROUP BY 聚合函数

本文将详细介绍 TiDB 支持的聚合函数。

12.11.4.13.1 TiDB 支持的聚合函数

TiDB 支持的 MySQL GROUP BY 聚合函数如下所示：

函数名	功能描述
COUNT()	返回检索到的行的数目
COUNT(DISTINCT)	返回不同值的数目
SUM()	返回和
AVG()	返回平均值
MAX()	返回最大值
MIN()	返回最小值
GROUP_CONCAT()	返回连接的字符串
VARIANCE(), VAR_POP()	返回总体标准方差
STD(), STDDEV(), STDDEV_POP	返回总体标准差
VAR_SAMP()	返回采样方差
STDDEV_SAMP()	返回采样标准方差
JSON_OBJECTAGG(key, value)	将结果集返回为单个含 (key, value) 键值对的 JSON object

注意：

- 除非另有说明，否则聚合函数默认忽略 NULL 值。
- 如果在不包含 GROUP BY 子句的语句中使用聚合函数，则相当于对所有行进行分组。

另外，TiDB 还支持以下聚合函数：

- APPROX_PERCENTILE(expr, constant_integer_expr)

  该函数用于计算 expr 值的百分位数。参数 constant_integer_expr 是一个取值为区间 [1,100] 内整数的常量表达式，表示百分数。一个百分位数 Pk（k为百分数）表示数据集中至少有 k% 的数据小于等于 Pk。

  该函数中，表达式的返回结果必须为数值类型或日期与时间类型。函数不支持计算其他类型的返回结果，并直接返回 NULL。

  该函数自 v4.0.8 版本起支持使用。

  以下是一个计算第 50 百分位数的例子：

```
drop table if exists t;
create table t(a int);
insert into t values(1), (2), (3);
```

```
select approx_percentile(a, 50) from t;
```

```
+--------------------------+
| approx_percentile(a, 50) |
+--------------------------+
|                        2 |
+--------------------------+
1 row in set (0.00 sec)
```

12.11.4.13.2 GROUP BY 修饰符

TiDB 目前不支持 GROUP BY 修饰符，例如 WITH ROLLUP，将来会提供支持。详情参阅 #4250。

12.11.4.13.3 对 SQL 模式的支持

TiDB 支持 SQL 模式 ONLY_FULL_GROUP_BY，当启用该模式时，TiDB 拒绝不明确的非聚合列的查询。例如，以下查询在启用 ONLY_FULL_GROUP_BY 时是不合规的，因为 SELECT 列表中的非聚合列 “b” 在 GROUP BY 语句中不显示：

```
drop table if exists t;
create table t(a bigint, b bigint, c bigint);
insert into t values(1, 2, 3), (2, 2, 3), (3, 2, 3);
```

```
select a, b, sum(c) from t group by a;
```

```
+------+------+--------+
| a    | b    | sum(c) |
+------+------+--------+
|    1 |    2 |      3 |
|    2 |    2 |      3 |
|    3 |    2 |      3 |
+------+------+--------+
3 rows in set (0.01 sec)
```

```
set sql_mode = 'ONLY_FULL_GROUP_BY';
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
select a, b, sum(c) from t group by a;
```

```
ERROR 1055 (42000): Expression #2 of SELECT list is not in GROUP BY clause and contains
    ↪ nonaggregated column 'b' which is not functionally dependent on columns in GROUP BY
    ↪ clause; this is incompatible with sql_mode=only_full_group_by
```

目前，TiDB 默认开启 SQL 模式ONLY_FULL_GROUP_BY。

与 MySQL 的区别

TiDB 目前实现的 ONLY_FULL_GROUP_BY 没有 MySQL 5.7 严格。例如，假设我们执行以下查询，希望结果按 “c” 排序：

```
drop table if exists t;
create table t(a bigint, b bigint, c bigint);
insert into t values(1, 2, 1), (1, 2, 2), (1, 3, 1), (1, 3, 2);
select distinct a, b from t order by c;
```

要对结果进行排序，必须先清除重复。但选择保留哪一行会影响 c 的保留值，也会影响排序，并使其具有任意性。

在 MySQL 中，ORDER BY 表达式需至少满足以下条件之一，否则 DISTINCT 和 ORDER BY 查询将因不合规而被拒绝：

- 表达式等同于 SELECT 列表中的一个。
- 表达式引用并属于查询选择表的所有列都是 SELECT 列表的元素。

但是在 TiDB 中，上述查询是合规的，详情参阅 #4254。

TiDB 中另一个标准 SQL 的扩展允许 HAVING 子句中的引用使用 SELECT 列表中的别名表达式。例如：以下查询返回在 orders 中只出现一次的 name 值：

```
select name, count(name) from orders
group by name
having count(name) = 1;
```

这个 TiDB 扩展允许在聚合列的 HAVING 子句中使用别名：

```
select name, count(name) as c from orders
group by name
having c = 1;
```

标准 SQL 只支持 GROUP BY 子句中的列表达式，以下语句不合规，因为 FLOOR(value/100) 是一个非列表达式：

```
select id, floor(value/100)
from tbl_name
group by id, floor(value/100);
```

TiDB 对标准 SQL 的扩展支持 GROUP BY 子句中非列表达式，认为上述语句合规。

标准 SQL 也不支持 GROUP BY 子句中使用别名。TiDB 对标准 SQL 的扩展支持使用别名，查询的另一种写法如下：

```
select id, floor(value/100) as val
from tbl_name
group by id, val;
```

12.11.4.13.4 TiDB 不支持的聚合函数

TiDB 目前不支持的聚合函数如下所示，相关进展参阅 TiDB #7623。

- JSON_ARRAYAGG

12.11.4.14 窗口函数

TiDB 中窗口函数的使用方法与 MySQL 8.0 基本一致，详情可参见 MySQL 窗口函数。由于窗口函数会使用一些保留关键字，可能导致原先可以正常执行的 SQL 语句在升级 TiDB 后无法被解析语法，此时可以将 tidb_enable_window_function 设置为 0，该参数的默认值为 1。

TiDB 支持的窗口函数如下所示：

函数名	功能描述
CUME_DIST()	返回一组值中的累积分布
DENSE_RANK()	返回分区中当前行的排名，并且排名是连续的
FIRST_VALUE()	当前窗口中第一行的表达式值
LAG()	分区中当前行前面第 N 行的表达式值
LAST_VALUE()	当前窗口中最后一行的表达式值
LEAD()	分区中当前行后面第 N 行的表达式值
NTH_VALUE()	当前窗口中第 N 行的表达式值
NTILE()	将分区划分为 N 桶，为分区中的每一行分配桶号
PERCENT_RANK()	返回分区中小于当前行的百分比
RANK()	返回分区中当前行的排名，排名可能不连续
ROW_NUMBER()	返回分区中当前行的编号

12.11.4.15 其他函数

TiDB 支持使用 MySQL 5.7 中提供的大部分其他函数。

12.11.4.15.1 支持的函数

函数名	功能描述
ANY_VALUE()	在 ONLY_FULL_GROUP_BY 模式下，防止带有 GROUP BY 的语句报错
DEFAULT()	返回表的某一列的默认值
INET_ATON()	将 IP 地址转换为数值
INET_NTOA()	将数值转换为 IP 地址
INET6_ATON()	将 IPv6 地址转换为数值
INET6_NTOA()	将数值转换为 IPv6 地址
IS_IPV4()	判断参数是否为 IPv4 地址
IS_IPV4_COMPAT()	判断参数是否为兼容 IPv4 的地址
IS_IPV4_MAPPED()	判断参数是否为 IPv4 映射的地址
IS_IPV6()	判断参数是否为 IPv6 地址
NAME_CONST()	可以用于重命名列名
SLEEP()	让语句暂停执行几秒时间
UUID()	返回一个通用唯一识别码 (UUID)
VALUES()	定义 INSERT 语句使用的值

12.11.4.15.2 不支持的函数

函数名	功能描述
GET_LOCK()	获取命名锁，详见 TiDB #10929
RELEASE_LOCK()	释放命名锁，详见 TiDB #10929
UUID_SHORT()	基于特定假设提供唯一的 UUID，目前这些假设在 TiDB 中不存在，详见 TiDB #4620
MASTER_WAIT_POS()	与 MySQL 同步相关

12.11.4.16 精度数学

TiDB 中精度数学计算与 MySQL 中基本一致, 详情请参见: Precision Math.

- 数值类型
- DECIMAL 数据类型的特性

12.11.4.16.1 数值类型

精确数值运算的范围包括精确值数据类型 (整型和 DECIMAL 类型), 以及精确值数字字面量. 近似值数据类型和近似值数字字面量被作为浮点数来处理.

精确值数字字面量包含整数部分或小数部分, 或二者都包含. 精确值数字字面量可以包含符号位. 例如: 1, .2, 3.4, -5, -6.78, +9.10.

近似值数字字面量以一个包含尾数和指数的科学计数法表示 (基数为 10). 其中尾数和指数可以分别或同时带有符号位. 例如: 1.2E3, 1.2E-3, -1.2E3, -1.2E-3.

两个看起来相似的数字可能会被以不同的方式进行处理. 例如, 2.34 是精确值 (定点数), 而 2.3E0 是近似值 (浮点数).

DECIMAL 数据类型是定点数类型, 其运算是精确计算. FLOAT 和 DOUBLE 数据类型是浮点类型, 其运算是近似计算.

12.11.4.16.2 DECIMAL 数据类型的特性

本节讨论 DECIMAL 数据类型的特性, 主要涉及以下几点:

1. 最大位数
2. 存储格式
3. 存储要求

DECIMAL 列的声明语法为 DECIMAL(M, D). 其中参数值意义及其范围如下:

- M 表示最大的数字位数 (精度). 1<= M <= 65.
- D 表示小数点右边数字的位数 (标度). 1 <= D <= 30 且不大于 M.

M 的最大值 65 表示 DECIMAL 值的计算精确到 65 位数字. 该精度同样适用于其精确值字面量.

DECIMAL 列的值采用二进制进行存储, 其将每 9 位十进制数字包装成 4 个字节. 其中整数和小数部分分别确定所需的存储空间. 如果数字位数为 9 的倍数, 则每 9 位十进制数字各采用 4 个字节进行存储, 对于剩余不足 9 位的数字, 所需的存储空间如下表所示.

剩余数字位数	存储所需字节数
0	0
1–2	1
3–4	2
5–6	3
7–9	4

例如, 定义类型为 DECIMAL(18, 9) 的列, 其小数点两侧均各包含 9 位十进制数字, 因此, 分别需要 4 个字节的存储空间. 定义类型为 DECIMAL(20, 6) 的列, 其小数部分包含 6 位十进制数字, 整数部分包含 14 位十进制数字. 整数部分中 9 位数字需要 4 个字节进行存储, 其余 5 位数字需要 3 个字节进行存储. 小数部分 6 位数字需要 3 个字节进行存储.

DECIMAL 列不存储前导的字符 + 或字符 - 或数字 0. 如果将 +0003.1 插入到 DECIMAL(5, 1) 列中, 则将其存储为 3.1. 对于负数, 不存储字符 - 的字面值.

DECIMAL 列不允许插入大于列定义的隐含范围的值. 例如, DECIMAL(3, 0) 列范围为 -999 到 999. DECIMAL(M, D) 列小数点左边部分最多支持 M-D 位数字.

有关 DECIMAL 值的内部格式完整说明, 请参阅 TiDB 源码文件 types/mydecimal.go.

12.11.4.16.3 表达式计算

在涉及精度数学计算的表达式中，TiDB 会尽可能不做任何修改的使用每个输入的数值。比如：在计算比较函数时，参与运算的数字将不做任何改变。在严格 SQL 模式下，向一个数据列插入一个值时，如果该值处于这一列的值域范围内，这个值将直接不做任何修改的直接插入进去，提取这个值的时候，取得的值和插入的值将会是同一个值。当处于非严格 SQL 模式时，TiDB 会允许数据插入过程中发生的数据截断。

处理数值类型表达式取决于这个表达式参数的具体值：

- 当表达式参数中包含近似值时，这个表达式的结果也是近似值，TiDB 会使用浮点数对应的计算逻辑返回一个浮点数的结果
- 当表达式参数中不包含任何近似值时（也就是说表达式的参数全部是精确值），如果某个精确值包含小数部分，TiDB 会对这个表达式使用 DECIMAL 对应的计算逻辑，返回一个 DECIMAL 的结果，精确到 65 位数字
- 其他情况下，表达式只会包含整数参数，这个表达式的结果也是精确的，TiDB 会使用整数对应的计算逻辑返回一个整数结果，精度和 BIGINT 保持一致（64 位）

如果数值类型表达式中包含字符串参数，这些字符串参数将被转换成双精度浮点数，这个表达式的计算结果将是个近似值。

向一个数值类型列插入数据的具体行为会受到 SQL 模式的影响。接下来的讨论将围绕严格模式以及 ERROR_FOR_DIVISION_BY_ZERO 模式展开，如果要打开所有的限制，可以简单的使用 TRADITIONAL 模式，这个模式将同时使用严格模式以及 ERROR_FOR_DIVISION_BY_ZERO 模式：

```
SET sql_mode = 'TRADITIONAL';
```

向一个具有精确值类型（DECIMAL 或者整数类型）的列插入数据时，如果插入的数据位于该列的值域范围内将使用该数据的精确值。如果该数据的小数部分太长，将会发生数值修约，这时会有 warning 产生，具体内容可以看“数值修约”。

如果该数据整数部分太长：

- 如果没有开启严格模式，这个值会被截断并产生一个 warning
- 如果开启了严格模式，将会产生一个数据溢出的 error

如果向一个数值类型列插入字符串，如果该字符串中包含非数值部分，TiDB 将这样做类型转换：

- 在严格模式下，没有以数字开头的字符串（即使是一个空字符串）不能被被用作数字值并会返回一个 error 或者是 warning；
- 以数字开头的字符串可以被转换，不过末尾的非数字部分会被截断。如果被截断的部分包含的不全是空格，在严格模式下这回产生一个 error 或者 warning

默认情况下，如果计算的过程中发生了除数是 0 的现象将会得到一个 NULL 结果，并且不会有 warning 产生。通过设置适当的 SQL 模式，除以 0 的操作可以被限制：当设置 ERROR_FOR_DIVISION_BY_ZERO SQL 模式时，TiDB 的行为是：

- 如果设置了严格 SQL 模式，INSERT 和 UPDATE 的过程中如果发生了除以 0 的操作，正在进行的 INSERT 或者 UPDATE 操作会被禁止，并且会返回一个 error
- 如果没有设置严格 SQL 模式，除以 0 的操作仅会返回一个 warning

假设我们有如下的 SQL 语句：

```
INSERT INTO t SET i = 1/0;
```

不同的 SQL 模式将会导致不同的结果如下：

sql_mode 的值	结果
''	没有 warning，没有 error，i 被设为 NULL
strict	没有 warning，没有 error，i 被设为 NULL
ERROR_FOR_DIVISION_BY_ZERO	有 warning，没有 error，i 被设为 NULL
strict, ERROR_FOR_DIVISION_BY_ZERO	有 error，插入失败

12.11.4.16.4 数值修约

round() 函数的结果取决于他的参数是否是精确值：

- 如果参数是精确值，round() 函数将使用四舍五入的规则
- 如果参数是一个近似值，round() 表达式的结果可能和 MySQL 不太一样

```
SELECT ROUND(2.5), ROUND(25E-1);
```

```
+------------+--------------+
| ROUND(2.5) | ROUND(25E-1) |
+------------+--------------+
|          3 |            3 |
+------------+--------------+
1 row in set (0.00 sec)
```

向一个 DECIMAL 或者整数类型列插入数据时，round 的规则将采用 round half away from zero 的方式：

```
CREATE TABLE t (d DECIMAL(10,0));
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
INSERT INTO t VALUES(2.5),(2.5E0);
```

```
Query OK, 2 rows affected, 2 warnings (0.00 sec)
```

```
SELECT d FROM t;
```

```
+------+
| d    |
+------+
|    3 |
|    3 |
+------+
2 rows in set (0.00 sec)
```

12.11.4.17 下推到 TiKV 的表达式列表

当 TiDB 从 TiKV 中读取数据的时候，TiDB 会尽量下推一些表达式运算到 TiKV 中，从而减少数据传输量以及 TiDB 单一节点的计算压力。本文将介绍 TiDB 已支持下推的表达式，以及如何禁止下推特定表达式。

12.11.4.17.1 已支持下推的表达式列表

表达式分类	具体操作
逻辑运算	AND (&&), OR (||), NOT (!)
比较运算	<, <=, =, != (<>), >, >=, <=>, IN(), IS NULL, LIKE, IS TRUE, IS FALSE, COALESCE()
数值运算	+, -, *, /, ABS(), CEIL(), CEILING(), FLOOR()
控制流运算	CASE, IF(), IFNULL()
JSON 运算	JSON_TYPE(json_val), JSON_EXTRACT(json_doc, path[, path] ⋯), JSON_OBJECT(key, val[, key, val] ⋯), JSON_ARRAY([val[, val] ⋯]), JSON_MERGE(json_doc, json_doc[, json_doc] ⋯), JSON_SET(json_doc, path, val[, path, val] ⋯), JSON_INSERT(json_doc, path, val[, path, val] ⋯), JSON_REPLACE(json_doc, path, val[, path, val] ⋯), JSON_REMOVE(json_doc, path[, path] ⋯)
日期运算	DATE_FORMAT()

12.11.4.17.2 禁止特定表达式下推

当函数的计算过程由于下推而出现异常时，可通过黑名单功能禁止其下推来快速恢复业务。具体而言，你可以将上述支持的函数或运算符名加入黑名单 `mysql.expr_pushdown_blacklist` 中，以禁止特定表达式下推。

`mysql.expr_pushdown_blacklist` 的 schema 如下：

```
tidb> desc mysql.expr_pushdown_blacklist;
+------------+--------------+------+------+-------------------+-------+
| Field      | Type         | Null | Key  | Default           | Extra |
+------------+--------------+------+------+-------------------+-------+
| name       | char(100)    | NO   |      | NULL              |       |
| store_type | char(100)    | NO   |      | tikv,tiflash,tidb |       |
| reason     | varchar(200) | YES  |      | NULL              |       |
+------------+--------------+------+------+-------------------+-------+
3 rows in set (0.00 sec)
```

以上结果字段解释如下：

- `name`：禁止下推的函数名。
- `store_type`：用于指明希望禁止该函数下推到哪些组件进行计算。组件可选 `tidb`、`tikv` 和 `tiflash`。`store_type` 不区分大小写，如果需要禁止向多个存储引擎下推，各个存储之间需用逗号隔开。
  - `store_type` 为 `tidb` 时表示在读取 TiDB 内存表时，是否允许该函数在其他 TiDB Server 上执行。

  - store_type 为 tikv 时表示是否允许该函数在 TiKV Server 的 Coprocessor 模块中执行。
  - store_type 为 tiflash 时表示是否允许该函数在 TiFlash Server 的 Coprocessor 模块中执行。
- reason：用于记录该函数被加入黑名单的原因。

> 注意：：
>
> tidb 是一种特殊的 store_type，其含义是 TiDB 内存表，比如：PERFORMANCE_SCHEMA.events_statements_summary_by_digest，属于系统表的一种，非特殊情况不用考虑这种存储引擎。

加入黑名单

执行以下步骤，可将一个或多个函数或运算符加入黑名单：

1. 向 mysql.expr_pushdown_blacklist 插入对应的函数名或运算符名以及希望禁止下推的存储类型集合。
2. 执行 admin reload expr_pushdown_blacklist;。

移出黑名单

执行以下步骤，可将一个或多个函数及运算符移出黑名单：

1. 从 mysql.expr_pushdown_blacklist 表中删除对应的函数名或运算符名。
2. 执行 admin reload expr_pushdown_blacklist;。

黑名单用法示例

以下示例首先将运算符 < 及 > 加入黑名单，然后将运算符 > 从黑名单中移出。

黑名单是否生效可以从 explain 结果中进行观察（参见如何理解 explain 结果）。

```
tidb> create table t(a int);
Query OK, 0 rows affected (0.06 sec)

tidb> explain select * from t where a < 2 and a > 2;
+--
    ↪ ----------------------+----------+-----------+---------------+--------------------------------+
    ↪
| id                    | estRows  | task      | access object | operator info
    ↪                      |
+--
    ↪ ----------------------+----------+-----------+---------------+--------------------------------+
    ↪
| TableReader_7           | 0.00     | root      |               | data:Selection_6
    ↪                    |
| └─Selection_6           | 0.00     | cop[tikv] |               | gt(ssb_1.t.a, 2), lt(ssb_1.t.
    ↪ a, 2) |
```

```
|   └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo     |
+--
    ↪ ----------------------+----------+-----------+---------------+--------------------------------+
    ↪
3 rows in set (0.00 sec)

tidb> insert into mysql.expr_pushdown_blacklist values('<', 'tikv',''), ('>','tikv','');
Query OK, 2 rows affected (0.01 sec)
Records: 2  Duplicates: 0  Warnings: 0

tidb> admin reload expr_pushdown_blacklist;
Query OK, 0 rows affected (0.00 sec)

tidb> explain select * from t where a < 2 and a > 2;
+--
    ↪ ----------------------+----------+-----------+---------------+--------------------------------+
    ↪
| id                      | estRows  | task      | access object | operator info
    ↪                       |
+--
    ↪ ----------------------+----------+-----------+---------------+--------------------------------+
    ↪
| Selection_7             | 10000.00 | root      |               | gt(ssb_1.t.a, 2), lt(ssb_1.t.a
    ↪ , 2) |
| └─TableReader_6         | 10000.00 | root      |               | data:TableFullScan_5
    ↪                |
|   └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo     |
+--
    ↪ ----------------------+----------+-----------+---------------+--------------------------------+
    ↪
3 rows in set (0.00 sec)

tidb> delete from mysql.expr_pushdown_blacklist where name = '>';
Query OK, 1 row affected (0.01 sec)

tidb> admin reload expr_pushdown_blacklist;
Query OK, 0 rows affected (0.00 sec)

tidb> explain select * from t where a < 2 and a > 2;
+--
    ↪ ------------------------+----------+-----------+---------------+--------------------------------+
    ↪
| id                        | estRows  | task      | access object | operator info
```

```
    ↪                   |
+--
    ↪ ------------------------+---------+----------+--------------+-------------------------------+
    ↪
| Selection_8               | 0.00     | root      |              | lt(ssb_1.t.a, 2)
    ↪               |
| └─TableReader_7           | 0.00     | root      |              | data:Selection_6
    ↪               |
|   └─Selection_6           | 0.00     | cop[tikv] |              | gt(ssb_1.t.a, 2)
    ↪               |
|     └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t      | keep order:false, stats:
    ↪ pseudo |
+--
    ↪ ------------------------+---------+----------+--------------+-------------------------------+
    ↪
4 rows in set (0.00 sec)
```

注意：

- `admin reload expr_pushdown_blacklist` 只对执行该 SQL 语句的 TiDB server 生效。若需要集群中所有 TiDB server 生效，需要在每台 TiDB server 上执行该 SQL 语句。
- 表达式黑名单功能在 v3.0.0 及以上版本中支持。
- 在 v3.0.3 及以下版本中，不支持将某些运算符的原始名称文本（如“>”、“+”和“is null”）加入黑名单中，部分运算符在黑名单中需使用别名。已支持下推的表达式中，别名与原始名不同的运算符见下表（区分大小写）。

运算符原始名称	运算符别名
<	lt
>	gt
<=	le
>=	ge
=	eq
!=	ne
<>	ne
<=>	nulleq
|	bitor
&&	bitand
||	or
!	not
in	in
+	plus

运算符原始名称	运算符别名
-	minus
*	mul
/	div
DIV	intdiv
IS NULL	isnull
IS TRUE	istrue
IS FALSE	isfalse

12.11.5 约束

TiDB 支持的约束与 MySQL 的基本相同。

12.11.5.1 非空约束

TiDB 支持的非空约束规则与 MySQL 支持的一致。例如：

```
CREATE TABLE users (
 id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
 age INT NOT NULL,
 last_login TIMESTAMP
);
```

```
INSERT INTO users (id,age,last_login) VALUES (NULL,123,NOW());
```

```
Query OK, 1 row affected (0.02 sec)
```

```
INSERT INTO users (id,age,last_login) VALUES (NULL,NULL,NOW());
```

```
ERROR 1048 (23000): Column 'age' cannot be null
```

```
INSERT INTO users (id,age,last_login) VALUES (NULL,123,NULL);
```

```
Query OK, 1 row affected (0.03 sec)
```

- 第一条 INSERT 语句成功，因为对于定义为 AUTO_INCREMENT 的列，允许 NULL 作为其特殊值。TiDB 将为其分配下一个自动值。
- 第二条 INSERT 语句失败，因为 age 列被定义为 NOT NULL。
- 第三条 INSERT 语句成功，因为 last_login 列没有被明确地指定为 NOT NULL。默认允许 NULL 值。

12.11.5.2 CHECK 约束

TiDB 会解析并忽略 CHECK 约束。该行为与 MySQL 5.7 的相兼容。

示例如下：

```
DROP TABLE IF EXISTS users;
CREATE TABLE users (
 id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
 username VARCHAR(60) NOT NULL,
 UNIQUE KEY (username),
 CONSTRAINT min_username_length CHECK (CHARACTER_LENGTH(username) >=4)
);
INSERT INTO users (username) VALUES ('a');
SELECT * FROM users;
```

12.11.5.3 唯一约束

在 TiDB 的乐观事务中，默认会对唯一约束进行惰性检查。通过在事务提交时再进行批量检查，TiDB 能够减少网络开销、提升性能。例如：

```
DROP TABLE IF EXISTS users;
CREATE TABLE users (
 id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
 username VARCHAR(60) NOT NULL,
 UNIQUE KEY (username)
);
INSERT INTO users (username) VALUES ('dave'), ('sarah'), ('bill');
```

默认的悲观事务模式下：

```
BEGIN;
INSERT INTO users (username) VALUES ('jane'), ('chris'), ('bill');
```

```
ERROR 1062 (23000): Duplicate entry 'bill' for key 'username'
```

乐观事务模式下且 tidb_constraint_check_in_place=0：

```
BEGIN OPTIMISTIC;
INSERT INTO users (username) VALUES ('jane'), ('chris'), ('bill');
```

```
Query OK, 0 rows affected (0.00 sec)
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
INSERT INTO users (username) VALUES ('steve'),('elizabeth');
```

```
Query OK, 2 rows affected (0.00 sec)
Records: 2  Duplicates: 0  Warnings: 0
```

```
COMMIT;
```

```
ERROR 1062 (23000): Duplicate entry 'bill' for key 'username'
```

在乐观事务的示例中，唯一约束的检查推迟到事务提交时才进行。由于 `bill` 值已经存在，这一行为导致了重复键错误。

你可通过设置 `tidb_constraint_check_in_place` 为 `1` 停用此行为（该变量设置对悲观事务无效，悲观事务始终在语句执行时检查约束）。当 `tidb_constraint_check_in_place` 设置为 `1` 时，则会在执行语句时就对唯一约束进行检查。例如：

```
DROP TABLE IF EXISTS users;
CREATE TABLE users (
 id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
 username VARCHAR(60) NOT NULL,
 UNIQUE KEY (username)
);
INSERT INTO users (username) VALUES ('dave'), ('sarah'), ('bill');
```

```
SET tidb_constraint_check_in_place = 1;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
BEGIN OPTIMISTIC;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
INSERT INTO users (username) VALUES ('jane'), ('chris'), ('bill');
```

```
ERROR 1062 (23000): Duplicate entry 'bill' for key 'username'
..
```

第一条 `INSERT` 语句导致了重复键错误。这会造成额外的网络通信开销，并可能降低插入操作的吞吐量。

12.11.5.4 主键约束

与 MySQL 行为一样，主键约束包含了唯一约束，即创建了主键约束相当于拥有了唯一约束。此外，TiDB 其他的主键约束规则也与 MySQL 相似。例如：

```
CREATE TABLE t1 (a INT NOT NULL PRIMARY KEY);
```

```
Query OK, 0 rows affected (0.12 sec)
```

```
CREATE TABLE t2 (a INT NULL PRIMARY KEY);
```

```
ERROR 1171 (42000): All parts of a PRIMARY KEY must be NOT NULL; if you need NULL in a key, use
    ↪ UNIQUE instead
```

```
CREATE TABLE t3 (a INT NOT NULL PRIMARY KEY, b INT NOT NULL PRIMARY KEY);
```

```
ERROR 1068 (42000): Multiple primary key defined
```

```
CREATE TABLE t4 (a INT NOT NULL, b INT NOT NULL, PRIMARY KEY (a,b));
```

```
Query OK, 0 rows affected (0.10 sec)
```

分析：

- 表 `t2` 创建失败，因为定义为主键的列 `a` 不能允许 `NULL` 值。
- 表 `t3` 创建失败，因为一张表只能有一个主键。
- 表 `t4` 创建成功，因为虽然只能有一个主键，但 TiDB 支持定义一个多列组合作为复合主键。

除上述规则外，默认情况下，TiDB 还有一个额外限制，即一旦一张表创建成功，其主键就不能再改变。如果需要添加/删除主键，需要在 TiDB 配置文件中将 `alter-primary-key` 设置为 `true`，并重启 TiDB 实例使之生效。

当开启添加/删除主键功能以后，TiDB 允许对表添加/删除主键。但需要注意的是，对于在未开启该功能时创建的整数类型的主键的表，即使开启添加/删除主键功能，也不能删除其主键约束。

12.11.5.5 外键约束

> **注意：**
>
> TiDB 仅部分支持外键约束功能。

TiDB 支持创建外键约束。例如：

```
CREATE TABLE users (
 id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
 doc JSON
);
CREATE TABLE orders (
 id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
 user_id INT NOT NULL,
 doc JSON,
 FOREIGN KEY fk_user_id (user_id) REFERENCES users(id)
);
```

```
SELECT table_name, column_name, constraint_name, referenced_table_name, referenced_column_name
FROM information_schema.key_column_usage WHERE table_name IN ('users', 'orders');
```

```
+------------+-------------+-----------------+-----------------------+------------------------+
| table_name | column_name | constraint_name | referenced_table_name | referenced_column_name |
+------------+-------------+-----------------+-----------------------+------------------------+
| users      | id          | PRIMARY         | NULL                  | NULL                   |
| orders     | id          | PRIMARY         | NULL                  | NULL                   |
| orders     | user_id     | fk_user_id      | users                 | id                     |
+------------+-------------+-----------------+-----------------------+------------------------+
3 rows in set (0.00 sec)
```

TiDB 也支持使用 ALTER TABLE 命令来删除外键 (DROP FOREIGN KEY) 和添加外键 (ADD FOREIGN KEY)：

```
ALTER TABLE orders DROP FOREIGN KEY fk_user_id;
ALTER TABLE orders ADD FOREIGN KEY fk_user_id (user_id) REFERENCES users(id);
```

12.11.5.5.1 注意

- TiDB 支持外键是为了在将其他数据库迁移到 TiDB 时，不会因为此语法报错。但是，TiDB 不会在 DML 语句中对外键进行约束检查。例如，即使 users 表中不存在 id=123 的记录，下列事务也能提交成功：

  ```
  START TRANSACTION;
  INSERT INTO orders (user_id, doc) VALUES (123, NULL);
  COMMIT;
  ```

12.11.6 生成列

> **警告：**
>
> 当前该功能为实验特性，不建议在生产环境中使用。

本文介绍生成列的概念以及用法。

12.11.6.1 生成列的基本概念

与一般的列不同，生成列的值由列定义中表达式计算得到。对生成列进行插入或更新操作时，并不能对之赋值，只能使用 DEFAULT。

生成列包括存储生成列和虚拟生成列。存储生成列会将计算得到的值存储起来，在读取时不需要重新计算。虚拟生成列不会存储其值，在读取时会重新计算。存储生成列和虚拟生成列相比，前者在读取时性能更好，但是要占用更多的磁盘空间。

无论是存储生成列还是虚拟列，都可以在其上面建立索引。

12.11.6.2 生成列的应用

生成列的主要的作用之一：从 JSON 数据类型中解出数据，并为该数据建立索引。

MySQL 5.7 及 TiDB 都不能直接为 JSON 类型的列添加索引，即不支持在如下表结构中的 `address_info` 上建立索引：

```
CREATE TABLE person (
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    address_info JSON,
    KEY (address_info)
);
```

如果要为 JSON 列某个字段添加索引，可以抽取该字段为生成列。

以 `city` 这一 `address_info` 中的字段为例，可以为其建立一个虚拟生成列并添加索引：

```
CREATE TABLE person (
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    address_info JSON,
    city VARCHAR(64) AS (JSON_UNQUOTE(JSON_EXTRACT(address_info, '$.city'))),
    KEY (city)
);
```

该表中，`city` 列是一个虚拟生成列。并且在该列上建立了索引。以下语句能够利用索引加速语句的执行速度：

```
SELECT name, id FROM person WHERE city = 'Beijing';
```

```
EXPLAIN SELECT name, id FROM person WHERE city = 'Beijing';
```

```
+---------------------------------+---------+-----------+---------------------------------+-----------------------
    ↪
| id                              | estRows | task      | access object                   |
    ↪ operator info                                           |
+---------------------------------+---------+-----------+---------------------------------+-----------------------
    ↪
| Projection_4                    | 10.00   | root      |                                 | test.
    ↪ person.name, test.person.id                             |
| └─IndexLookUp_10                | 10.00   | root      |                                 |
    ↪                                                         |
|   ├─IndexRangeScan_8(Build)     | 10.00   | cop[tikv] | table:person, index:city(city) | range
    ↪ :["Beijing","Beijing"], keep order:false, stats:pseudo |
|   └─TableRowIDScan_9(Probe)     | 10.00   | cop[tikv] | table:person                    | keep
    ↪ order:false, stats:pseudo                               |
+---------------------------------+---------+-----------+---------------------------------+-----------------------
    ↪
```

从执行计划中，可以看出使用了 city 这个索引来读取满足 city = 'Beijing' 这个条件的行的 HANDLE，再用这个 HANDLE 来读取该行的数据。

如果 $.city 路径中无数据，则 JSON_EXTRACT 返回 NULL。如果想增加约束，city 列必须是 NOT NULL，则可按照以下方式定义虚拟生成列：

```
CREATE TABLE person (
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    address_info JSON,
    city VARCHAR(64) AS (JSON_UNQUOTE(JSON_EXTRACT(address_info, '$.city'))) NOT NULL,
    KEY (city)
);
```

12.11.6.3 生成列在 INSERT 和 UPDATE 语句中的行为

INSERT 和 UPDATE 语句都会检查生成列计算得到的值是否满足生成列的定义。未通过有效性检测的行会返回错误：

```
INSERT INTO person (name, address_info) VALUES ('Morgan', JSON_OBJECT('Country', 'Canada'));
```

```
ERROR 1048 (23000): Column 'city' cannot be null
```

12.11.6.4 索引生成列替换

当查询中出现的某个表达式与一个含索引的生成列同等时，TiDB 会将这个表达式替换为对应的生成列，这样就可以在生成查询计划时考虑使用这个索引。

例如，下面的例子为 a+1 这个表达式创建生成列并添加索引，从而加速了查询。

```
create table t(a int);
desc select a+1 from t where a+1=3;
+--
    ↪ ------------------------+----------+-----------+--------------+------------------------------+
    ↪
| id                        | estRows  | task      | access object | operator info
    ↪                  |
+--
    ↪ ------------------------+----------+-----------+--------------+------------------------------+
    ↪
| Projection_4              | 8000.00  | root      |               | plus(test.t.a, 1)->Column#3
    ↪    |
| └─TableReader_7           | 8000.00  | root      |               | data:Selection_6
    ↪               |
|   └─Selection_6           | 8000.00  | cop[tikv] |               | eq(plus(test.t.a, 1), 3)
    ↪       |
```

```
|     └─TableFullScan_5     | 10000.00 | cop[tikv] | table:t       | keep order:false, stats:
    ↪ pseudo |
+--
    ↪ ------------------------+----------+-----------+---------------+-----------------------------+
    ↪
4 rows in set (0.00 sec)

alter table t add column b bigint as (a+1) virtual;
alter table t add index idx_b(b);
desc select a+1 from t where a+1=3;
+--
    ↪ ----------------------+---------+-----------+------------------------+-------------------------------
    ↪
| id                    | estRows | task      | access object          | operator info
    ↪                               |
+--
    ↪ ----------------------+---------+-----------+------------------------+-------------------------------
    ↪
| IndexReader_6           | 10.00   | root      |                        | index:IndexRangeScan_5
    ↪                         |
| └─IndexRangeScan_5      | 10.00   | cop[tikv] | table:t, index:idx_b(b) | range:[3,3], keep
    ↪ order:false, stats:pseudo |
+--
    ↪ ----------------------+---------+-----------+------------------------+-------------------------------
    ↪
2 rows in set (0.01 sec)
```

> 注意：
>
> 只有当待替换的表达式类型和生成列类型严格相等时，才会进行转换。
>
> 上例中，a 的类型是 int，而 a+1 的列类型是 bigint，如果将生成列的类型定为 int，就不会发生替换。
>
> 关于类型转换规则，可以参见表达式求值的类型转换。

12.11.6.5 生成列的局限性

目前生成列有以下局限性：

- 不能通过 ALTER TABLE 增加存储生成列；
- 不能通过 ALTER TABLE 将存储生成列转换为普通列，也不能将普通列转换成存储生成列；
- 不能通过 ALTER TABLE 修改存储生成列的生成列表达式；
- 并未支持所有的JSON 函数；

- 目前仅当生成列是虚拟生成列时索引生成列替换规则有效，暂不支持将表达式替换为存储生成列，但仍然可以通过直接使用该生成列本身来使用索引。

12.11.7 SQL 模式

TiDB 服务器采用不同 SQL 模式来操作，且不同客户端可以应用不同模式。SQL 模式定义 TiDB 支持哪些 SQL 语法及执行哪种数据验证检查.

TiDB 启动之后采用 `SET [ SESSION | GLOBAL ] sql_mode='modes'`设置 SQL 模式。设置 GLOBAL 级别的 SQL 模式时用户需要有 SUPER 权限，并且只会影响到从设置 SQL 模式开始后续新建立的连接（注：老连接不受影响)。SESSION 级别的 SQL 模式的变化只会影响当前的客户端。

Modes 是用逗号（‘,’）间隔开的一系列不同的模式。使用 `SELECT @@sql_mode` 语句查询当前 SQL 模式，SQL 模式默认值：`ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,`
↪ `ERROR_FOR_DIVISION_BY_ZERO,NO_AUTO_CREATE_USER,NO_ENGINE_SUBSTITUTION`。

12.11.7.1 重要的 sql_mode 值

- ANSI: 符合标准 SQL，对数据进行校验，如果不符合定义类型或长度，对数据类型调整或截断保存，且返回 warning 警告。
- STRICT_TRANS_TABLES: 严格模式，对数据进行严格校验，当数据出现错误时，插入到表中，并且返回错误。
- TRADITIONAL: 采用此模式使 TiDB 的行为像“传统” SQL 数据库系统，当在列中插入不正确的值时“给出错误而不是警告”，一旦发现错误立即放弃 INSERT/UPDATE。

12.11.7.2 SQL mode 列表，如下

名称	含义
PIPES_AS_CONCAT	将“||”视为字符串连接操作符（+）(同 CONCAT())，而不视为 OR（支持)

名称	含义
ANSI_QUOTES	将"视为识别符，如果启用ANSI_QUOTES，只单引号内的会被认为是String Liter-als，双引号被解释为识别符，因此不能用双引号来引用字符串（支持）

名称	含义
IGNORE_SPACE	若开启该模式，系统忽略空格。例如：“user”和“user “是相同的（支持）

名称	含义
ONLY_FULL_GROUP_BY	如果 GROUP BY 出现的列并没有在 SELECT，HAVING，ORDER BY 中出现，此 SQL 不合法，因为不在 GROUP BY 中的列被查询展示出来不符合正常现象 (支持)

名称	含义
NO_UNSIGNED_SUBTRACTION	在减运算中，如果某个操作数没有符号，不要将结果标记为 UNSIGNED（支持）
NO_DIR_IN_CREATE	创建表时，忽视所有 INDEX DIRECTORY 和 DATA DIRECTORY 指令，该选项仅对从复制服务器有用（仅语法支持）

名称	含义
NO_KEY_OPTIONS	使用 SHOW CREATE TABLE 时不会输出 MySQL 特有的语法部分，如 ENGINE，使用 mysqldump 跨 DB 种类迁移的时需要考虑此选项（仅语法支持）

名称	含义
NO_FIELD_OPTIONS	使用 SHOW CREATE TABLE 时不会输出 MySQL 特有的语法部分，如 ENGINE，使用 mysqldump 跨 DB 种类迁移的时需要考虑此选项（仅语法支持）

名称	含义
NO_TABLE_OPTIONS	使用 SHOW CRE-ATE TABLE 时不会输出 MySQL 特有的语法部分，如 EN-GINE，使用 mysql-dump 跨 DB 种类迁移的时需要考虑此选项（仅语法支持）

名称	含义
NO_AUTO_VALUE_ON_ZERO	若启用该模式，在 AUTO_INCREMENT 列的处理传入的值是 0 或者具体数值时系统直接将该值写入此列，传入 NULL 时系统自动生成下一个序列号（支持）
NO_BACKSLASH_ESCAPES	若启用该模式，\ 反斜杠符号仅代表它自己（支持）

名称	含义
STRICT_TRANS_TABLES	对于事务存储引擎启用严格模式，insert 非法值之后，回滚整条语句（支持）
STRICT_ALL_TABLES	对于事务型表，写入非法值之后，回滚整个事务语句（支持）

名称	含义
NO_ZERO_DATE	在严格模式，不接受月或日部分为 0 的日期。如果使用 IGNORE 选项，我们为类似的日期插入’0000-00-00’。在非严格模式，可以接受该日期，但会生成警告（支持）

名称	含义
NO_ZERO_DATE	在严格模式，不要将 '0000-00-00' 做为合法日期。你仍然可以用 IGNORE 选项插入零日期。在非严格模式，可以接受该日期，但会生成警告（支持）

名称	含义
ALLOW_INVALID_DATES	不检查全部日期的合法性，仅检查月份值在 1 到 12 及日期值在 1 到 31 之间，仅适用于 DATE 和 DATA-TIME 列，TIMES-TAMP 列需要全部检查其合法性（支持）

名称	含义
ERROR_FOR_DIVISION_BY_ZERO	若启用该模式，在 INSERT 或 UPDATE 过程中，被除数为 0 值时，系统产生错误若未启用该模式，被除数为 0 值时，系统产生警告，并用 NULL 代替（支持）

名称	含义
NO_AUTO_CREATE_USER	防止 GRANT 自动创建新用户，但指定密码除外（支持）

名称	含义
HIGH_NOT_PRECEDENCE	NOT 操作符的优先级是表达式。例如：NOT a BETWEEN b AND c 被解释为 NOT (a BETWEEN b AND c)。在部份旧版本 MySQL 中，表达式被解释为 (NOT a) BETWEEN b AND c (支持)

名称	含义
NO_ENGINE_SUBSTITUTION	如果需要的存储引擎被禁用或未编译，可以防止自动替换存储引擎（仅语法支持）
PAD_CHAR_TO_FULL_LENGTH	若启用该模式，系统对于 CHAR 类型不会截断尾部空格（仅语法支持。该模式在 MySQL 8.0 中已废弃。）

名称	含义
REAL_AS_FLOAT	将 REAL 视为 FLOAT 的同义词，而不是 DOUBLE 的同义词（支持）
POSTGRESQL	等同于 PIPES_AS_CONCAT、ANSI_QUOTES、IGNORE_SPACE、NO_KEY_OPTIONS、NO_TABLE_OPTIONS、NO_FIELD_OPTIONS（仅语法支持）
MSSQL	等同于 PIPES_AS_CONCAT、ANSI_QUOTES、IGNORE_SPACE、NO_KEY_OPTIONS、NO_TABLE_OPTIONS、NO_FIELD_OPTIONS（仅语法支持）

名称	含义
DB2	等同于 PIPES_AS_CONCAT、ANSI_QUOTES、IGNORE_SPACE、NO_KEY_OPTIONS、NO_TABLE_OPTIONS、NO_FIELD_OPTIONS（仅语法支持）
MAXDB	等同于 PIPES_AS_CONCAT、ANSI_QUOTES、IGNORE_SPACE、NO_KEY_OPTIONS、NO_TABLE_OPTIONS、NO_FIELD_OPTIONS、NO_AUTO_CREATE_USER（支持）
MySQL32	等同于 NO_FIELD_OPTIONS、HIGH_NOT_PRECEDENCE（仅语法支持）
MYSQL40	等同于 NO_FIELD_OPTIONS、HIGH_NOT_PRECEDENCE（仅语法支持）

名称	含义
ANSI	等同于 REAL_AS_FLOAT、PIPES_AS_CONCAT、ANSI_QUOTES、IGNORE_SPACE (仅语法支持)
TRADITIONAL	等同于 STRICT_TRANS_TABLES、STRICT_ALL_TABLES、NO_ZERO_IN_DATE、NO_ZERO_DATE、ERROR_FOR_DIVISION_BY_ZERO、NO_AUTO_CREATE_USER (仅语法支持)
ORACLE	等同于 PIPES_AS_CONCAT、ANSI_QUOTES、IGNORE_SPACE、NO_KEY_OPTIONS、NO_TABLE_OPTIONS、NO_FIELD_OPTIONS、NO_AUTO_CREATE_USER (仅语法支持)

12.11.8 事务

12.11.8.1 TiDB 事务概览

TiDB 支持分布式事务，提供乐观事务与悲观事务两种事务模型。TiDB 3.0.8 及以后版本，TiDB 默认采用悲观事

务模型。

本文主要介绍涉及事务的常用语句、显式/隐式事务、事务的隔离级别和惰性检查，以及事务大小的限制。

常用的变量包括autocommit、 tidb_disable_txn_auto_retry、 tidb_retry_limit 以及tidb_txn_mode。

> 注意：
>
> 变量tidb_disable_txn_auto_retry 和tidb_retry_limit 仅适用于乐观事务，不适用于悲观事务。

12.11.8.1.1 常用事务语句

开启事务

要显式地开启一个新事务，既可以使用BEGIN 语句，也可以使用START TRANSACTION 语句，两者效果相同。

语法：

```
BEGIN;
```

```
START TRANSACTION;
```

```
START TRANSACTION WITH CONSISTENT SNAPSHOT;
```

如果执行以上语句时，当前 Session 正处于一个事务的中间过程，那么系统会先自动提交当前事务，再开启一个新的事务。

> 注意：
>
> 与 MySQL 不同的是，TiDB 在执行完上述语句后即会获取当前数据库快照，而 MySQL 的 BEGIN 和 START TRANSACTION 是在开启事务后的第一个从 InnoDB 读数据的 SELECT 语句（非 SELECT FOR UPDATE）后获取快照，START TRANSACTION WITH CONSISTENT SNAPSHOT 是语句执行时获取快照。因此，TiDB 中的 BEGIN、START TRANSACTION 和 START TRANSACTION WITH CONSISTENT SNAPSHOT 都等效为 MySQL 中的 START TRANSACTION WITH CONSISTENT SNAPSHOT。

提交事务

COMMIT 语句用于提交 TiDB 在当前事务中进行的所有修改。

语法：

```
COMMIT;
```

> 建议：
>
> 启用乐观事务前，请确保应用程序可正确处理 COMMIT 语句可能返回的错误。如果不确定应用程序将会如何处理，建议改为使用悲观事务。

回滚事务

ROLLBACK 语句用于回滚并撤销当前事务的所有修改。

语法：

```
ROLLBACK;
```

如果客户端连接中止或关闭，也会自动回滚该事务。

12.11.8.1.2 自动提交

为满足 MySQL 兼容性的要求，在默认情况下，TiDB 将在执行语句后立即进行 autocommit（自动提交）。

举例：

```
mysql> CREATE TABLE t1 (
    ->  id INT NOT NULL PRIMARY KEY auto_increment,
    ->  pad1 VARCHAR(100)
    -> );
Query OK, 0 rows affected (0.09 sec)

mysql> SELECT @@autocommit;
+--------------+
| @@autocommit |
+--------------+
| 1            |
+--------------+
1 row in set (0.00 sec)

mysql> INSERT INTO t1 VALUES (1, 'test');
Query OK, 1 row affected (0.02 sec)

mysql> ROLLBACK;
Query OK, 0 rows affected (0.01 sec)

mysql> SELECT * FROM t1;
+----+------+
| id | pad1 |
+----+------+
|  1 | test |
```

```
+----+------+
1 row in set (0.00 sec)
```

以上示例中，ROLLBACK 语句没产生任何效果。由于 INSERT 语句是在自动提交的情况下执行的，等同于以下单语句事务：

```
START TRANSACTION;
INSERT INTO t1 VALUES (1, 'test');
COMMIT;
```

如果已显式地启动事务，则不适用自动提交。以下示例，ROLLBACK 语句成功撤回了 INSERT 语句：

```
mysql> CREATE TABLE t2 (
    ->  id INT NOT NULL PRIMARY KEY auto_increment,
    ->  pad1 VARCHAR(100)
    -> );
Query OK, 0 rows affected (0.10 sec)

mysql> SELECT @@autocommit;
+--------------+
| @@autocommit |
+--------------+
| 1            |
+--------------+
1 row in set (0.00 sec)

mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO t2 VALUES (1, 'test');
Query OK, 1 row affected (0.02 sec)

mysql> ROLLBACK;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM t2;
Empty set (0.00 sec)
```

autocommit 是一个系统变量，可以基于 Session 或 Global 进行修改。

举例：

```
SET autocommit = 0;
```

```
SET GLOBAL autocommit = 0;
```

12.11.8.1.3 显式事务和隐式事务

> **注意：**
>
> 有些语句是隐式提交的。例如，执行 `[BEGIN|START TRANCATION]` 语句时，TiDB 会隐式提交上一个事务，并开启一个新的事务以满足 MySQL 兼容性的需求。详情参见 implicit commit。

TiDB 可以显式地使用事务（通过 `[BEGIN|START TRANSACTION]/COMMIT` 语句定义事务的开始和结束）或者隐式地使用事务（`SET autocommit = 1`）。

在自动提交状态下，使用 `[BEGIN|START TRANSACTION]` 语句会显式地开启一个事务，同时也会禁用自动提交，使隐式事务变成显式事务。直到执行 `COMMIT` 或 `ROLLBACK` 语句时才会恢复到此前默认的自动提交状态。

对于 DDL 语句，会自动提交并且不能回滚。如果运行 DDL 的时候，正在一个事务的中间过程中，会先自动提交当前事务，再执行 DDL。

12.11.8.1.4 惰性检查

执行 DML 语句时，乐观事务默认不会检查主键约束或唯一约束，而是在 `COMMIT` 事务时进行这些检查。

举例：

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY);
INSERT INTO t1 VALUES (1);
BEGIN OPTIMISTIC;
INSERT INTO t1 VALUES (1); -- MySQL 返回错误；TiDB 返回成功。
INSERT INTO t1 VALUES (2);
COMMIT; -- MySQL 提交成功；TiDB 返回错误，事务回滚。
SELECT * FROM t1; -- MySQL 返回 1 2；TiDB 返回 1。
```

```
mysql> CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY);
Query OK, 0 rows affected (0.10 sec)

mysql> INSERT INTO t1 VALUES (1);
Query OK, 1 row affected (0.02 sec)

mysql> BEGIN OPTIMISTIC;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO t1 VALUES (1); -- MySQL 返回错误；TiDB 返回成功。
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO t1 VALUES (2);
Query OK, 1 row affected (0.00 sec)

mysql> COMMIT; -- MySQL 提交成功；TiDB 返回错误，事务回滚。
```

```
ERROR 1062 (23000): Duplicate entry '1' for key 'PRIMARY'
mysql> SELECT * FROM t1; -- MySQL 返回 1 2; TiDB 返回 1。
+----+
| id |
+----+
|  1 |
+----+
1 row in set (0.01 sec)
```

惰性检查优化通过批处理约束检查并减少网络通信来提升性能。可以通过设置tidb_constraint_check_in_place ↪ = TRUE 禁用该行为。

> 注意：
>
> - 本优化仅适用于乐观事务。
> - 本优化仅对普通的 INSERT 语句生效，对 INSERT IGNORE 和 INSERT ON DUPLICATE KEY ↪ UPDATE 不会生效。

12.11.8.1.5 语句回滚

TiDB 支持语句执行失败后的原子性回滚。如果语句报错，则所做的修改将不会生效。该事务将保持打开状态，并且在发出 COMMIT 或 ROLLBACK 语句之前可以进行其他修改。

```
CREATE TABLE test (id INT NOT NULL PRIMARY KEY);
BEGIN;
INSERT INTO test VALUES (1);
INSERT INTO tset VALUES (2);  -- tset 拼写错误，使该语句执行出错。
INSERT INTO test VALUES (1),(2);  -- 违反 PRIMARY KEY 约束，语句不生效。
INSERT INTO test VALUES (3);
COMMIT;
SELECT * FROM test;
```

```
mysql> CREATE TABLE test (id INT NOT NULL PRIMARY KEY);
Query OK, 0 rows affected (0.09 sec)

mysql> BEGIN;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO test VALUES (1);
Query OK, 1 row affected (0.02 sec)

mysql> INSERT INTO tset VALUES (2);  -- tset 拼写错误，使该语句执行出错。
ERROR 1146 (42S02): Table 'test.tset' doesn't exist
```

```
mysql> INSERT INTO test VALUES (1),(2);  -- 违反 PRIMARY KEY 约束，语句不生效。
ERROR 1062 (23000): Duplicate entry '1' for key 'PRIMARY'
mysql> INSERT INTO test VALUES (3);
Query OK, 1 row affected (0.00 sec)

mysql> COMMIT;
Query OK, 0 rows affected (0.01 sec)

mysql> SELECT * FROM test;
+----+
| id |
+----+
|  1 |
|  3 |
+----+
2 rows in set (0.00 sec)
```

以上例子中，INSERT 语句执行失败之后，事务保持打开状态。最后的 INSERT 语句执行成功，并且提交了修改。

12.11.8.1.6 事务限制

由于底层存储引擎的限制，TiDB 要求单行不超过 6 MB。可以将一行的所有列根据类型转换为字节数并加和来估算单行大小。

TiDB 同时支持乐观事务与悲观事务，其中乐观事务是悲观事务的基础。由于乐观事务是先将修改缓存在私有内存中，因此，TiDB 对于单个事务的容量做了限制。

TiDB 中，单个事务的总大小默认不超过 100 MB，这个默认值可以通过配置文件中的配置项 `txn-total-size-limit` 进行修改，最大支持 10 GB。实际的单个事务大小限制还取决于服务器剩余可用内存的大小，执行事务时 TiDB 进程的内存消耗大约是事务大小的 6 倍以上。

在 4.0 以前的版本，TiDB 限制了单个事务的键值对的总数量不超过 30 万条，从 4.0 版本起 TiDB 取消了这项限制。

> **注意：**
>
> 通常，用户会开启 TiDB Binlog 将数据向下游进行同步。某些场景下，用户会使用消息中间件来消费同步到下游的 binlog，例如 Kafka。
>
> 以 Kafka 为例，Kafka 的单条消息处理能力的上限是 1 GB。因此，当把 `txn-total-size-limit` 设置为 1 GB 以上时，可能出现事务在 TiDB 中执行成功，但下游 Kafka 报错的情况。为避免这种情况出现，请用户根据最终消费者的限制来决定 `txn-total-size-limit` 的实际大小。例如：下游使用了 Kafka，则 `txn-total-size-limit` 不应超过 1 GB。

12.11.8.2 TiDB 事务隔离级别

事务隔离级别是数据库事务处理的基础，ACID 中的 “I”，即 Isolation，指的就是事务的隔离性。

SQL-92 标准定义了 4 种隔离级别：读未提交 (READ UNCOMMITTED)、读已提交 (READ COMMITTED)、可重复读 (REPEATABLE READ)、串行化 (SERIALIZABLE)。详见下表：

Isolation Level	Dirty Write	Dirty Read	Fuzzy Read	Phantom
READ UNCOMMITTED	Not Possible	Possible	Possible	Possible
READ COMMITTED	Not Possible	Not possible	Possible	Possible
REPEATABLE READ	Not Possible	Not possible	Not possible	Possible
SERIALIZABLE	Not Possible	Not possible	Not possible	Not possible

TiDB 实现了快照隔离 (Snapshot Isolation, SI) 级别的一致性。为与 MySQL 保持一致，又称其为 “可重复读”。该隔离级别不同于ANSI 可重复读隔离级别和MySQL 可重复读隔离级别。

> 注意：
>
> 在 TiDB v3.0 中，事务的自动重试功能默认为禁用状态。不建议开启自动重试功能，因为可能导致事务隔离级别遭到破坏。更多关于事务自动重试的文档说明，请参考事务重试。
>
> 从 TiDB v3.0.8 版本开始，新创建的 TiDB 集群会默认使用悲观事务模式，悲观事务中的当前读（for update 读）为不可重复读，关于悲观事务使用注意事项，请参考悲观事务模式

12.11.8.2.1 可重复读隔离级别 (Repeatable Read)

当事务隔离级别为可重复读时，只能读到该事务启动时已经提交的其他事务修改的数据，未提交的数据或在事务启动后其他事务提交的数据是不可见的。对于本事务而言，事务语句可以看到之前的语句做出的修改。

对于运行于不同节点的事务而言，不同事务启动和提交的顺序取决于从 PD 获取时间戳的顺序。

处于可重复读隔离级别的事务不能并发的更新同一行，当事务提交时发现该行在该事务启动后，已经被另一个已提交的事务更新过，那么该事务会回滚。示例如下：

```
create table t1(id int);
insert into t1 values(0);

start transaction;              |               start transaction;
select * from t1;               |               select * from t1;
update t1 set id=id+1;          |               update t1 set id=id+1; -- 如果使用悲观事务，
    ↪ 则后一个执行的 update 语句会等锁，直到持有锁的事务提交或者回滚释放行锁。
commit;                         |
                                |               commit; -- 事务提交失败，回滚。如果使用悲观事务，
                                    ↪ 可以提交成功。
```

与 ANSI 可重复读隔离级别的区别

尽管名称是可重复读隔离级别，但是 TiDB 中可重复读隔离级别和 ANSI 可重复隔离级别是不同的。按照 A Critique of ANSI SQL Isolation Levels 论文中的标准，TiDB 实现的是论文中的快照隔离级别。该隔离级别不会出现狭

义上的幻读 (A3)，但不会阻止广义上的幻读 (P3)，同时，SI 还会出现写偏斜，而 ANSI 可重复读隔离级别不会出现写偏斜，会出现幻读。

与 MySQL 可重复读隔离级别的区别

MySQL 可重复读隔离级别在更新时并不检验当前版本是否可见，也就是说，即使该行在事务启动后被更新过，同样可以继续更新。这种情况在 TiDB 会导致事务回滚，导致事务最终失败，而 MySQL 是可以更新成功的。MySQL 的可重复读隔离级别并非 Snapshot 隔离级别，MySQL 可重复读隔离级别的一致性要弱于 Snapshot 隔离级别，也弱于 TiDB 的可重复读隔离级别。

12.11.8.2.2 读已提交隔离级别 (Read Committed)

从 TiDB v4.0.0-beta 版本开始，TiDB 支持使用 Read Committed 隔离级别。由于历史原因，当前主流数据库的 Read Committed 隔离级别本质上都是 Oracle 定义的一致性读隔离级别。TiDB 为了适应这一历史原因，悲观事务中的 Read Committed 隔离级别的实质行为也是一致性读。

> 注意：
>
> Read Committed 隔离级别仅在悲观事务模式下生效。在乐观事务模式下设置事务隔离级别为 Read Committed 将不会生效，事务将仍旧使用可重复读隔离级别。

与 MySQL Read Committed 隔离级别的区别

MySQL 的 Read Committed 隔离级别大部分符合一致性读特性，但其中存在某些特例，如半一致性读 (semi-consistent read)，TiDB 没有兼容这个特殊行为。

12.11.8.2.3 更多阅读

- TiDB 的乐观事务模型
- TiDB 新特性漫谈 -悲观事务
- TiDB 新特性 -白话悲观锁
- TiKV 的 MVCC (Multi-Version Concurrency Control) 机制

12.11.8.3 TiDB 乐观事务模型

乐观事务模型下，将修改冲突视为事务提交的一部分。因此并发事务不常修改同一行时，可以跳过获取行锁的过程进而提升性能。但是并发事务频繁修改同一行（冲突）时，乐观事务的性能可能低于悲观事务。

启用乐观事务前，请确保应用程序可正确处理 `COMMIT` 语句可能返回的错误。如果不确定应用程序将会如何处理，建议改为使用悲观事务。

> 注意：
>
> 自 v3.0.8 开始，TiDB 集群默认使用悲观事务模型。但如果从 3.0.7 及之前版本创建的集群升级到 3.0.8 及之后的版本，不会改变默认事务模型，即只有新创建的集群才会默认使用悲观事务模型。

12.11.8.3.1 乐观事务原理

为支持分布式事务，TiDB 中乐观事务使用两阶段提交协议，流程如下：

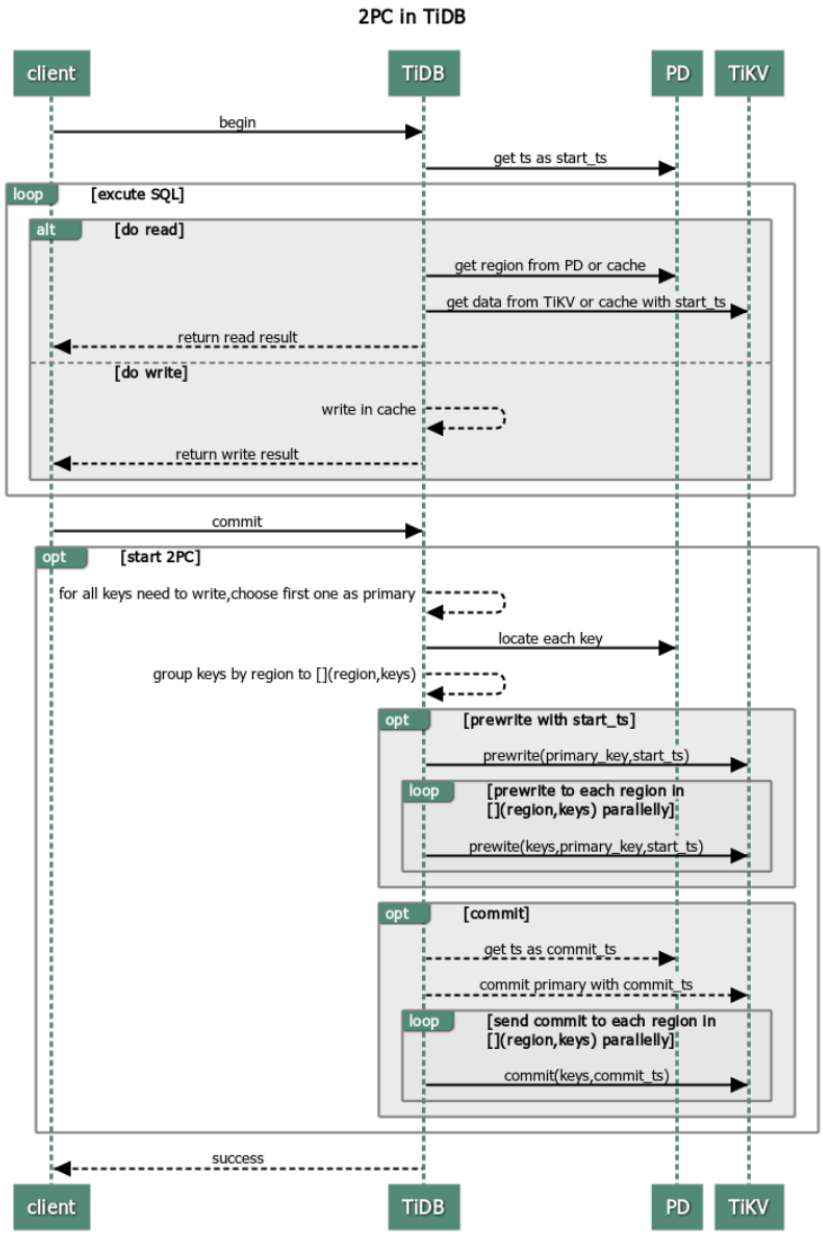


图 369: TiDB 中的两阶段提交

1. 客户端开始一个事务。

   TiDB 从 PD 获取一个全局唯一递增的时间戳作为当前事务的唯一事务 ID，这里称为该事务的 `start_ts`。TiDB 实现了多版本并发控制（MVCC），因此 `start_ts` 同时也作为该事务获取的数据库快照版本。该事务只能读到此 `start_ts` 版本可以读到的数据。

2. 客户端发起读请求。

1. TiDB 从 PD 获取数据路由信息，即数据具体存在哪个 TiKV 节点上。
2. TiDB 从 TiKV 获取 `start_ts` 版本下对应的数据。

3. 客户端发起写请求。

   TiDB 校验写入数据是否符合约束（如数据类型是否正确、是否符合非空约束等）。校验通过的数据将存放在 TiDB 中该事务的私有内存里。

4. 客户端发起 commit。

5. TiDB 开始两阶段提交，在保证事务原子性的前提下，进行数据持久化。

   1. TiDB 从当前要写入的数据中选择一个 Key 作为当前事务的 Primary Key。
   2. TiDB 从 PD 获取所有数据的写入路由信息，并将所有的 Key 按照所有的路由进行分类。
   3. TiDB 并发地向所有涉及的 TiKV 发起 prewrite 请求。TiKV 收到 prewrite 数据后，检查数据版本信息是否存在冲突或已过期。符合条件的数据会被加锁。
   4. TiDB 收到所有 prewrite 响应且所有 prewrite 都成功。
   5. TiDB 向 PD 获取第二个全局唯一递增版本号，定义为本次事务的 `commit_ts`。
   6. TiDB 向 Primary Key 所在 TiKV 发起第二阶段提交。TiKV 收到 commit 操作后，检查数据合法性，清理 prewrite 阶段留下的锁。
   7. TiDB 收到两阶段提交成功的信息。

6. TiDB 向客户端返回事务提交成功的信息。

7. TiDB 异步清理本次事务遗留的锁信息。

12.11.8.3.2 优缺点分析

通过分析 TiDB 中事务的处理流程，可以发现 TiDB 事务有如下优点：

- 实现原理简单，易于理解。
- 基于单实例事务实现了跨节点事务。
- 锁管理实现了去中心化。

但 TiDB 事务也存在以下缺点：

- 两阶段提交使网络交互增多。
- 需要一个中心化的分配时间戳服务。
- 事务数据量过大时易导致内存暴涨。

12.11.8.3.3 事务的重试

使用乐观事务模型时，在高冲突率的场景中，事务容易发生写写冲突而导致提交失败。MySQL 使用悲观事务模型，在执行 SQL 语句的过程中进行加锁并且其 Repeatable Read 隔离级别允许出现不可重复读，所以提交时一般不会出现异常。为了降低应用改造难度，TiDB 提供了数据库内部自动重试机制。

重试机制

当事务提交时，如果发现写写冲突，TiDB 内部重新执行包含写操作的 SQL 语句。你可以通过设置 `tidb_disable_txn_auto_retry = OFF` 开启自动重试，并通过 `tidb_retry_limit` 设置重试次数：

```
#### 设置是否禁用自动重试，默认为 “on”，即不重试。
tidb_disable_txn_auto_retry = OFF
#### 控制重试次数，默认为 “10”。只有自动重试启用时该参数才会生效。
#### 当 “tidb_retry_limit= 0” 时，也会禁用自动重试。
tidb_retry_limit = 10
```

你也可以修改当前 Session 或 Global 的值：

- Session 级别设置：

```
SET tidb_disable_txn_auto_retry = OFF;
```

```
SET tidb_retry_limit = 10;
```

- Global 级别设置：

```
SET GLOBAL tidb_disable_txn_auto_retry = OFF;
```

```
SET GLOBAL tidb_retry_limit = 10;
```

> 注意：
>
> tidb_retry_limit 变量决定了事务重试的最大次数。当它被设置为 0 时，所有事务都不会自动重试，包括自动提交的单语句隐式事务。这是彻底禁用 TiDB 中自动重试机制的方法。禁用自动重试后，所有冲突的事务都会以最快的方式上报失败信息（包含 try again later）给应用层。

重试的局限性

TiDB 默认不进行事务重试，因为重试事务可能会导致更新丢失，从而破坏可重复读的隔离级别。

事务重试的局限性与其原理有关。事务重试可概括为以下三个步骤：

1. 重新获取 start_ts。
2. 重新执行包含写操作的 SQL 语句。
3. 再次进行两阶段提交。

第二步中，重试时仅重新执行包含写操作的 SQL 语句，并不涉及读操作的 SQL 语句。但是当前事务中读到数据的时间与事务真正开始的时间发生了变化，写入的版本变成了重试时获取的 start_ts 而非事务一开始时获取的 start_ts。因此，当事务中存在依赖查询结果来更新的语句时，重试将无法保证事务原本可重复读的隔离级别，最终可能导致结果与预期出现不一致。

如果业务可以容忍事务重试导致的异常，或并不关注事务是否以可重复读的隔离级别来执行，则可以开启自动重试。

12.11.8.3.4 冲突检测

作为一个分布式系统，TiDB 在内存中的冲突检测是在 TiKV 中进行，主要发生在 prewrite 阶段。因为 TiDB 集群是一个分布式系统，TiDB 实例本身无状态，实例之间无法感知到彼此的存在，也就无法确认自己的写入与别的 TiDB 实例是否存在冲突，所以会在 TiKV 这一层检测具体的数据是否有冲突。

具体配置如下：

```
#### scheduler 内置一个内存锁机制，防止同时对一个 Key 进行操作。
#### 每个 Key hash 到不同的 slot。（默认为 2048000）
scheduler-concurrency = 2048000
```

此外，TiKV 支持监控等待 latch 的时间：

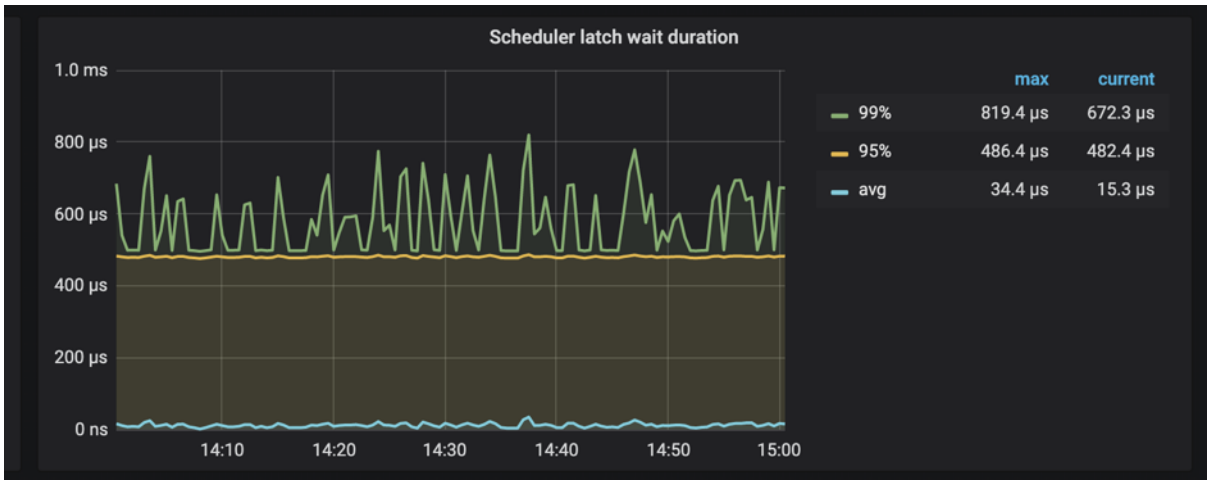


图 370: Scheduler latch wait duration

当 `Scheduler latch wait duration` 的值特别高时，说明大量时间消耗在等待锁的请求上。如果不存在底层写入慢的问题，基本上可以判断该段时间内冲突比较多。

12.11.8.3.5 更多阅读

- Percolator 和 TiDB 事务算法

12.11.8.4 TiDB 悲观事务模型

为了使 TiDB 的使用方式更加贴近传统数据库，降低用户迁移的成本，TiDB 自 v3.0 版本开始在乐观事务模型的基础上支持了悲观事务模型。本文将介绍 TiDB 悲观事务的相关特性。

> 注意：
>
> 自 v3.0.8 开始，新创建的 TiDB 集群默认使用悲观事务模型。但如果从 v3.0.7 版本及之前创建的集群升级到 >= v3.0.8 的版本，则不会改变默认的事务模型，即只有新创建的集群才会默认使用悲观事务模型。

12.11.8.4.1 事务模式的修改方法

你可以使用tidb_txn_mode 系统变量设置事务模式。执行以下命令，即可使整个集群中所有新创建 session 执行的所有显示事务（即非 autocommit 的事务）进入悲观事务模式：

```
SET GLOBAL tidb_txn_mode = 'pessimistic';
```

除此之外，还可以执行以下 SQL 语句显式地开启悲观事务：

```
BEGIN PESSIMISTIC;
```

```
BEGIN /*T! PESSIMISTIC */;
```

BEGIN PESSIMISTIC; 和 BEGIN OPTIMISTIC; 等语句的优先级高于 tidb_txn_mode 系统变量。使用这两个语句开启的事务，会忽略系统变量，从而支持悲观、乐观事务混合使用。

12.11.8.4.2 悲观事务模式的行为

悲观事务的行为和 MySQL 基本一致（不一致之处详见和 MySQL InnoDB 的差异）：

- UPDATE、DELETE 或 INSERT 语句都会读取已提交的最新数据来执行，并对所修改的行加悲观锁。
- SELECT FOR UPDATE 语句会对已提交的最新的数据而非所修改的行加上悲观锁。
- 悲观锁会在事务提交或回滚时释放。其他尝试修改这一行的写事务会被阻塞，等待悲观锁的释放。其他尝试读取这一行的事务不会被阻塞，因为 TiDB 采用多版本并发控制机制 (MVCC)。
- 如果多个事务尝试获取各自的锁，会出现死锁，并被检测器自动检测到。其中一个事务会被随机终止掉并返回兼容 MySQL 的错误码 1213。
- 通过 innodb_lock_wait_timeout 变量，设置事务等锁的超时时间（默认值为 50，单位为秒）。等锁超时后返回兼容 MySQL 的错误码 1205。如果多个事务同时等待同一个锁释放，会大致按照事务 start ts 顺序获取锁。
- 乐观事务和悲观事务可以共存，事务可以任意指定使用乐观模式或悲观模式来执行。
- 支持 FOR UPDATE NOWAIT 语法，遇到锁时不会阻塞等锁，而是返回兼容 MySQL 的错误码 3572。
- 如果 Point Get 和 Batch Point Get 算子没有读到数据，依然会对给定的主键或者唯一键加锁，阻塞其他事务对相同主键唯一键加锁或者进行写入操作。

12.11.8.4.3 和 MySQL InnoDB 的差异

1. 有些 WHERE 子句中使用了 range，TiDB 在执行这类 DML 语句和 SELECT FOR UPDATE 语句时，不会阻塞 range 内并发的 DML 语句的执行。

   举例：

```
CREATE TABLE t1 (
 id INT NOT NULL PRIMARY KEY,
 pad1 VARCHAR(100)
);
INSERT INTO t1 (id) VALUES (1),(5),(10);
```

```
BEGIN /*T! PESSIMISTIC */;
SELECT * FROM t1 WHERE id BETWEEN 1 AND 10 FOR UPDATE;
```

```
BEGIN /*T! PESSIMISTIC */;
INSERT INTO t1 (id) VALUES (6); -- 仅 MySQL 中出现阻塞。
UPDATE t1 SET pad1='new value' WHERE id = 5; -- MySQL 和 TiDB 处于等待阻塞状态。
```

产生这一行为是因为 TiDB 当前不支持 gap locking（间隙锁）。

2. TiDB 不支持 `SELECT LOCK IN SHARE MODE`。

   使用这个语句执行的时候，效果和没有加锁是一样的，不会阻塞其他事务的读写。

3. DDL 可能会导致悲观事务提交失败。

   MySQL 在执行 DDL 语句时，会被正在执行的事务阻塞住，而在 TiDB 中 DDL 操作会成功，造成悲观事务提交失败：`ERROR 1105 (HY000): Information schema is changed. [try again later]`。TiDB 事务执行过程中并发执行 `TRUNCATE TABLE` 语句，可能会导致事务报错 `table doesn't exist`。

4. `START TRANSACTION WITH CONSISTENT SNAPSHOT` 之后，MySQL 仍然可以读取到之后在其他事务创建的表，而 TiDB 不能。

5. autocommit 事务优先采用乐观事务提交。

   使用悲观事务模型时，autocommit 事务首先尝试使用开销更小的乐观事务模式提交。如果发生了写冲突，重试时才会使用悲观事务提交。所以 `tidb_retry_limit = 0` 时，autocommit 事务遇到写冲突仍会报 `Write Conflict` 错误。

   自动提交的 `SELECT FOR UPDATE` 语句不会等锁。

6. 对语句中 `EMBEDDED SELECT` 读到的相关数据不会加锁。

7. 垃圾回收 (GC) 不会影响到正在执行的事务，但悲观事务的执行时间仍有上限，默认为 10 分钟，可通过 TiDB 配置文件 `[performance]` 类别下的 `max-txn-ttl` 修改。

12.11.8.4.4 隔离级别

TiDB 在悲观事务模式下支持了 2 种隔离级别：

1. 默认使用与 MySQL 行为相同的可重复读隔离级别 (Repeatable Read)。

> 注意：
>
> 在这种隔离级别下，DML 操作会基于已提交的最新数据来执行，行为与 MySQL 相同，但与 TiDB 乐观事务不同，请参考与 MySQL 可重复读隔离级别的区别。

2. 使用SET TRANSACTION 语句可将隔离级别设置为读已提交隔离级别 (Read Committed)。

12.11.8.4.5 Pipelined 加锁流程

加悲观锁需要向 TiKV 写入数据，要经过 Raft 提交并 apply 后才能返回，相比于乐观事务，不可避免的会增加部分延迟。为了降低加锁的开销，TiKV 实现了 pipelined 加锁流程：当数据满足加锁要求时，TiKV 立刻通知 TiDB 执行后面的请求，并异步写入悲观锁，从而降低大部分延迟，显著提升悲观事务的性能。但当 TiKV 出现网络隔离或者节点宕机时，悲观锁异步写入有可能失败，从而产生以下影响：

- 无法阻塞修改相同数据的其他事务。如果业务逻辑依赖加锁或等锁机制，业务逻辑的正确性将受到影响。
- 有较低概率导致事务提交失败，但不会影响事务正确性。

如果业务逻辑依赖加锁或等锁机制，或者即使在集群异常情况下也要尽可能保证事务提交的成功率，应关闭 pipelined 加锁功能。

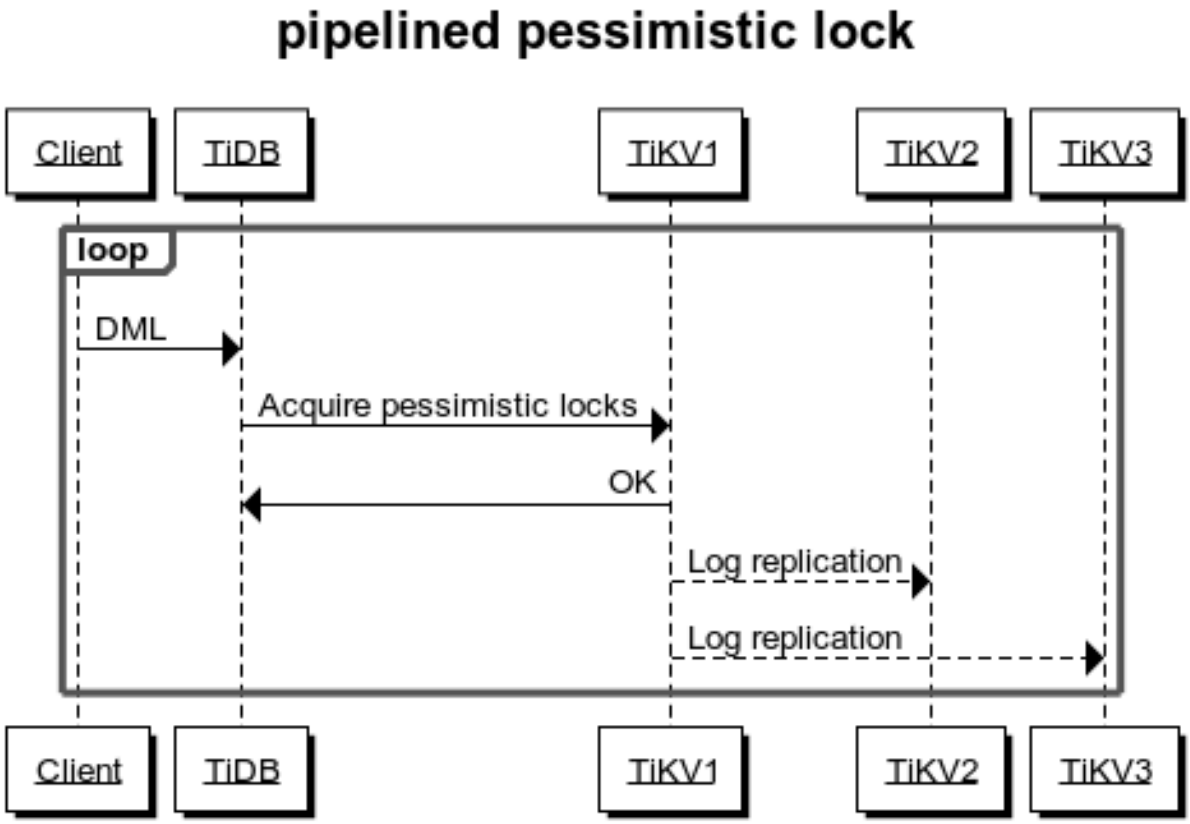


图 371: Pipelined pessimistic lock

该功能默认关闭，可修改 TiKV 配置启用：

```
[pessimistic-txn]
pipelined = true
```

若集群是 v4.0.9 及以上版本，也可通过在线修改 TiKV 配置功能动态开启该功能：

```
set config tikv pessimistic-txn.pipelined='true';
```

12.11.8.4.6 常见问题

1. TiDB 日志出现 `pessimistic write conflict, retry statement`。

   当发生 write conflict 时，乐观事务会直接终止，而悲观事务会尝试用最新数据重试该语句直到没有 write conflict，每次重试都会打印该 log，不用特别关注。

2. 执行 DML 时报错 `pessimistic lock retry limit reached`。

   悲观事务每个语句有重试次数限制，当因 write conflict 重试超过该限制时会报该错误，默认为 256 次，可通过 TiDB 配置文件 `[pessimistic-txn]` 类别下的 `max-retry-limit` 修改。

3. 悲观事务执行时间限制。

   在 v4.0 中，GC 已不会影响到正在运行的事务，但悲观事务的执行时间仍有上限，默认为 10 分钟，可通过 TiDB 配置文件 `[performance]` 类别下的 `max-txn-ttl` 修改。

12.11.9 垃圾回收 (GC)

12.11.9.1 GC 机制简介

TiDB 的事务的实现采用了 MVCC（多版本并发控制）机制，当新写入的数据覆盖旧的数据时，旧的数据不会被替换掉，而是与新写入的数据同时保留，并以时间戳来区分版本。GC 的任务便是清理不再需要的旧数据。

12.11.9.1.1 整体流程

一个 TiDB 集群中会有一个 TiDB 实例被选举为 GC leader，GC 的运行由 GC leader 来控制。

GC 会被定期触发。每次 GC 时，首先，TiDB 会计算一个称为 safe point 的时间戳，接下来 TiDB 会在保证 safe point 之后的快照全部拥有正确数据的前提下，删除更早的过期数据。每一轮 GC 分为以下三个步骤：

1. Resolve Locks。该阶段会对所有 Region 扫描 safe point 之前的锁，并清理这些锁。
2. Delete Ranges。该阶段快速地删除由于 `DROP TABLE/DROP INDEX` 等操作产生的整区间的废弃数据。
3. Do GC。该阶段每个 TiKV 节点将会各自扫描该节点上的数据，并对每一个 key 删除其不再需要的旧版本。

默认配置下，GC 每 10 分钟触发一次，每次 GC 会保留最近 10 分钟内的数据（即默认 GC life time 为 10 分钟，safe point 的计算方式为当前时间减去 GC life time）。如果一轮 GC 运行时间太久，那么在一轮 GC 完成之前，即使到了下一次触发 GC 的时间也不会开始下一轮 GC。另外，为了使持续时间较长的事务能在超过 GC life time 之后仍然可以正常运行，safe point 不会超过正在执行中的事务的开始时间 (start_ts)。

12.11.9.1.2 实现细节

Resolve Locks（清理锁）

TiDB 的事务是基于 Google Percolator 模型实现的，事务的提交是一个两阶段提交的过程。第一阶段完成时，所有涉及的 key 都会上锁，其中一个锁会被选为 Primary，其余的锁 (Secondary) 则会存储一个指向 Primary 的指针；第二阶段会将 Primary 锁所在的 key 加上一个 Write 记录，并去除锁。这里的 Write 记录就是历史上对该 key 进行写入或删除，或者该 key 上发生事务回滚的记录。Primary 锁被替换为何种 Write 记录标志着该事务提交成功与否。接下来，所有 Secondary 锁也会被依次替换。如果因为某些原因（如发生故障等），这些 Secondary 锁没有完成替换、残留了下来，那么也可以根据锁中的信息取找到 Primary，并根据 Primary 是否提交来判断整个事务是否提交。但是，如果 Primary 的信息在 GC 中被删除了，而该事务又存在未成功提交的 Secondary 锁，那么就永远无法得知该锁是否可以提交。这样，数据的正确性就无法保证。

Resolve Locks 这一步的任务即对 safe point 之前的锁进行清理。即如果一个锁对应的 Primary 已经提交，那么该锁也应该被提交；反之，则应该回滚。而如果 Primary 仍然是上锁的状态（没有提交也没有回滚），则应当将该事务视为超时失败而回滚。

Resolve Locks 的执行方式是由 GC leader 对所有的 Region 发送请求扫描过期的锁，并对扫到的锁查询 Primary 的状态，再发送请求对其进行提交或回滚。这个过程默认会并行地执行，并发数量默认与 TiKV 节点个数相同。

Delete Ranges（删除区间）

在执行 `DROP TABLE/INDEX` 等操作时，会有大量连续的数据被删除。如果对每个 key 都进行删除操作、再对每个 key 进行 GC 的话，那么执行效率和空间回收速度都可能非常的低下。事实上，这种时候 TiDB 并不会对每个 key 进行删除操作，而是将这些待删除的区间及删除操作的时间戳记录下来。Delete Ranges 会将这些时间戳在 safe point 之前的区间进行快速的物理删除。

Do GC（进行 GC 清理）

这一步即删除所有 key 的过期版本。为了保证 safe point 之后的任何时间戳都具有一致的快照，这一步删除 safe point 之前提交的数据，但是会对每个 key 保留 safe point 前的最后一次写入（除非最后一次写入是删除）。

在进行这一步时，TiDB 只需要将 safe point 发送给 PD，即可结束整轮 GC。TiKV 会自行检测到 safe point 发生了更新，会对当前节点上所有作为 Region leader 进行 GC。与此同时，GC leader 可以继续触发下一轮 GC。

> **注意：**
>
> TiDB v2.1 以及更早的版本中，Do GC 这一步是通过由 TiDB 对每个 Region 发送请求的方式实现的。在 v3.0 及更新的版本中，通过修改配置可以继续使用旧的 GC 方式，详情请参考GC 配置。

12.11.9.2 GC 配置

TiDB 的 GC 相关的配置存储于 `mysql.tidb` 系统表中，可以通过 SQL 语句对这些参数进行查询和更改：

```
select VARIABLE_NAME, VARIABLE_VALUE from mysql.tidb where VARIABLE_NAME like "tikv_gc%";
```

```
+--------------------------+-------------------------------------------------------------------------------------------------
    ↪
| VARIABLE_NAME            | VARIABLE_VALUE
    ↪                                                                                                |
```

```
+-------------------------+-------------------------------------------------------------------------
    ↪
| tikv_gc_leader_uuid       | 5afd54a0ea40005
    ↪                                                                                                       |
| tikv_gc_leader_desc       | host:tidb-cluster-tidb-0, pid:215, start at 2019-07-15
    ↪ 11:09:14.029668932 +0000 UTC m=+0.463731223 |
| tikv_gc_leader_lease      | 20190715-12:12:14 +0000
    ↪                                                                                               |
| tikv_gc_enable            | true
    ↪
    ↪ |
| tikv_gc_run_interval      | 10m0s
    ↪
    ↪ |
| tikv_gc_life_time         | 10m0s
    ↪
    ↪ |
| tikv_gc_last_run_time     | 20190715-12:09:14 +0000
    ↪                                                                                               |
| tikv_gc_safe_point        | 20190715-11:59:14 +0000
    ↪                                                                                               |
| tikv_gc_auto_concurrency | true
    ↪
    ↪ |
| tikv_gc_mode              | distributed
    ↪                                                                                                           |
+-------------------------+-------------------------------------------------------------------------
    ↪
13 rows in set (0.00 sec)
```

例如，如果需要将 GC 调整为保留最近一天以内的数据，只需执行下列语句即可：

```sql
update mysql.tidb set VARIABLE_VALUE="24h" where VARIABLE_NAME="tikv_gc_life_time";
```

注意：

mysql.tidb 系统表中除了下文列出的 GC 的配置以外，还包含一些 TiDB 用于储存部分集群状态（包括 GC 状态）的记录。请勿手动更改这些记录。其中，与 GC 有关的记录如下：

- tikv_gc_leader_uuid，tikv_gc_leader_desc 和 tikv_gc_leader_lease 用于记录 GC leader 的状态
- tikv_gc_last_run_time：最近一次 GC 运行的时间（每轮 GC 开始时更新）
- tikv_gc_safe_point：当前的 safe point（每轮 GC 开始时更新）

12.11.9.2.1 tikv_gc_enable

- 控制是否启用 GC。
- 默认值：true

12.11.9.2.2 tikv_gc_run_interval

- 指定 GC 运行时间间隔。Duration 类型，使用 Go 的 Duration 字符串格式，如 "1h30m"，"15m" 等。
- 默认值："10m0s"

12.11.9.2.3 tikv_gc_life_time

- 每次 GC 时，保留数据的时限。Duration 类型。每次 GC 时将以当前时间减去该配置的值作为 safe point。
- 默认值："10m0s"

> 注意：
>
> - 在数据更新频繁的场景下，如果将 tikv_gc_life_time 设置得比较大（如数天甚至数月），可能会有一些潜在的问题，如：
>   - 磁盘空间占用较多。
>   - 大量的历史版本会在一定程度上影响性能，尤其是范围查询（如 select count(*) from t）。
> - 如果存在运行时间很长、超过了 tikv_gc_life_time 的事务，那么在 GC 时，会保留自该事务的开始时间 (start_ts) 以来的数据，以允许该事务继续运行。例如，如果 tikv_gc_life_time 配置为 10 分钟，而某次 GC 时，集群中正在运行的事务中开始时间最早的一个事务已经运行了 15 分钟，那么本次 GC 便会保留最近 15 分钟的数据。

12.11.9.2.4 tikv_gc_mode

指定 GC 模式。可选值如下：

- "distributed"（默认）：分布式 GC 模式。在此模式下，Do GC 阶段由 TiDB 上的 GC leader 向 PD 发送 safe point，每个 TiKV 节点各自获取该 safe point 并对所有当前节点上作为 leader 的 Region 进行 GC。此模式于 TiDB 3.0 引入。
- "central"：集中 GC 模式。在此模式下，Do GC 阶段由 GC leader 向所有的 Region 发送 GC 请求。TiDB 2.1 及更早版本采用此 GC 模式。

12.11.9.2.5 tikv_gc_auto_concurrency

控制是否由 TiDB 自动决定 GC concurrency，即同时进行 GC 的线程数。

当 tikv_gc_mode 设为 "distributed"，GC concurrency 将应用于Resolve Locks 阶段。当tikv_gc_mode 设为 "central" 时，GC concurrency 将应用于 Resolve Locks 以及Do GC 两个阶段。

- `true`（默认）：自动以 TiKV 节点的个数作为 GC concurrency
- `false`：使用tikv_gc_concurrency 的值作为 GC 并发数

12.11.9.2.6 tikv_gc_concurrency

- 手动设置 GC concurrency。要使用该参数，必须将tikv_gc_auto_concurrency 设为 `false`。
- 默认值：2

12.11.9.2.7 tikv_gc_scan_lock_mode

> **警告：**
>
> Green GC 目前为实验特性，不建议在生产环境中使用。

设定 GC 的 Resolve Locks 阶段中，扫描锁的方式，即是否开启 Green GC（实验性特性）。Resolve Locks 阶段需要扫描整个集群的锁。在不开启 Green GC 的情况下，TiDB 会以 Region 为单位进行扫描。Green GC 提供了“物理扫描”的功能，即每台 TiKV 节点分别绕过 Raft 层直接扫描数据。该功能可以有效缓解Hibernate Region 功能开启时，GC 唤醒全部 Region 的现象，并一定程度上提升 Resolve Locks 阶段的执行速度。

- `"legacy"`（默认）：使用旧的扫描方式，即关闭 Green GC。
- `"physical"`：使用物理扫描的方式，即开启 Green GC。

> **注意：**
>
> 该项配置是隐藏配置。首次开启需要执行：

```
insert into mysql.tidb values ('tikv_gc_scan_lock_mode', 'legacy', '');
```

12.11.9.2.8 关于 GC 流程的说明

从 TiDB 3.0 版本起，由于对分布式 GC 模式和并行 Resolve Locks 的支持，部分配置选项的作用发生了变化。可根据下表理解不同版本中这些配置的区别：

版本/配置	Resolve Locks	Do GC
2.x	串行	并行
3.0 `tikv_gc_mode = centered`	并行	并行
`tikv_gc_auto_concurrency = false`		

版本/配置	Resolve Locks	Do GC
3.0 `tikv_gc_mode = centered` `tikv_gc_auto_concurrency = true`	自动并行	自动并行
3.0 `tikv_gc_mode = distributed` `tikv_gc_auto_concurrency = false`	并行	分布式
3.0 `tikv_gc_mode = distributed` `tikv_gc_auto_concurrency = true`（默认配置）	自动并行	分布式

表格内容说明：

- 串行：由 TiDB 逐个向 Region 发送请求。
- 并行：使用 `tikv_gc_concurrency` 选项所指定的线程数，并行地向每个 Region 发送请求。
- 自动并行：使用 TiKV 节点的个数作为线程数，并行地向每个 Region 发送请求。
- 分布式：无需 TiDB 通过对 TiKV 发送请求的方式来驱动，而是每台 TiKV 自行工作。

另外，如果 Green GC（实验特性）开启（即`tikv_gc_scan_lock_mode` 配置项设为 `"physical"`），Resolve Lock 的执行将不受上述并行配置的影响。

12.11.9.2.9 流控

TiDB 在 3.0.6 版本开始支持 GC 流控，可通过配置 `gc.max-write-bytes-per-sec` 限制 GC worker 每秒数据写入量，降低对正常请求的影响，0 为关闭该功能。该配置可通过 tikv-ctl 动态修改：

```
tikv-ctl --host=ip:port modify-tikv-config -n gc.max-write-bytes-per-sec -v 10MB
```

12.11.10 视图

TiDB 支持视图，视图是一张虚拟表，该虚拟表的结构由创建视图时的 SELECT 语句定义。使用视图一方面可以对用户只暴露安全的字段及数据，进而保证底层表的敏感字段及数据的安全。另一方面，将频繁出现的复杂查询定义为视图，可以使复杂查询更加简单便捷。

12.11.10.1 查询视图

查询一个视图和查询一张普通表类似。但是 TiDB 在真正执行查询视图时，会将视图展开成创建视图时定义的 SELECT 语句，进而执行展开后的查询语句。

12.11.10.2 查看视图的相关信息

通过以下方式，可以查看 view 相关的信息。

12.11.10.2.1 使用 SHOW CREATE TABLE view_name 或 SHOW CREATE VIEW view_name 语句

示例：

```
show create view v;
```

使用该语句可以查看 view 对应的创建语句，及创建 view 时对应的 character_set_client 及 collation_connection 系统变量值。

```
+--
    ↪ ----+-------------------------------------------------------------------------------------------------
    ↪
| View | Create View
    ↪
    ↪ | character_set_client | collation_connection |
+--
    ↪ ----+-------------------------------------------------------------------------------------------------
    ↪
| v    | CREATE ALGORITHM=UNDEFINED DEFINER=`root`@`127.0.0.1` SQL SECURITY DEFINER VIEW `v` (`a
    ↪ `) AS SELECT `s`.`a` FROM `test`.`t` LEFT JOIN `test`.`s` ON `t`.`a`=`s`.`a` | utf8
    ↪                  | utf8_general_ci      |
+--
    ↪ ----+-------------------------------------------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

12.11.10.2.2 查询 INFORMATION_SCHEMA.VIEWS 表

示例:

```
select * from information_schema.views;
```

通过查询该表可以查看 view 的相关元信息，如 TABLE_CATALOG、TABLE_SCHEMA、TABLE_NAME、VIEW_DEFINITION、CHECK_OPTION、IS_UPDATABLE、DEFINER、SECURITY_TYPE、CHARACTER_SET_CLIENT、COLLATION_CONNECTION 等。

```
+--
    ↪ -------------+--------------+------------+-----------------------------------------------------------
    ↪
| TABLE_CATALOG | TABLE_SCHEMA | TABLE_NAME | VIEW_DEFINITION
    ↪                                                          | CHECK_OPTION | IS_UPDATABLE |
    ↪ DEFINER        | SECURITY_TYPE | CHARACTER_SET_CLIENT | COLLATION_CONNECTION |
+--
    ↪ -------------+--------------+------------+-----------------------------------------------------------
    ↪
```

```
| def           | test         | v           | SELECT `s`.`a` FROM `test`.`t` LEFT JOIN `test`.`s`
    ↪  ON `t`.`a`=`s`.`a` | CASCADED     | NO           | root@127.0.0.1 | DEFINER       | utf8
    ↪                   | utf8_general_ci      |
+--
    ↪ -------------+--------------+------------+-----------------------------------------------------------------
    ↪
1 row in set (0.00 sec)
```

12.11.10.2.3 查询 HTTP API

示例:

```
curl http://127.0.0.1:10080/schema/test/v
```

通过访问 `http://{TiDBIP}:10080/schema/{db}/{view}` 可以得到对应 view 的所有元信息。

```
{
 "id": 122,
 "name": {
  "O": "v",
  "L": "v"
 },
 "charset": "utf8",
 "collate": "utf8_general_ci",
 "cols": [
  {
   "id": 1,
   "name": {
    "O": "a",
    "L": "a"
   },
   "offset": 0,
   "origin_default": null,
   "default": null,
   "default_bit": null,
   "default_is_expr": false,
   "generated_expr_string": "",
   "generated_stored": false,
   "dependences": null,
   "type": {
    "Tp": 0,
    "Flag": 0,
    "Flen": 0,
    "Decimal": 0,
    "Charset": "",
    "Collate": "",
```

```
    "Elems": null
   },
   "state": 5,
   "comment": "",
   "hidden": false,
   "version": 0
  }
 ],
 "index_info": null,
 "fk_info": null,
 "state": 5,
 "pk_is_handle": false,
 "is_common_handle": false,
 "comment": "",
 "auto_inc_id": 0,
 "auto_id_cache": 0,
 "auto_rand_id": 0,
 "max_col_id": 1,
 "max_idx_id": 0,
 "update_timestamp": 416801600091455490,
 "ShardRowIDBits": 0,
 "max_shard_row_id_bits": 0,
 "auto_random_bits": 0,
 "pre_split_regions": 0,
 "partition": null,
 "compression": "",
 "view": {
  "view_algorithm": 0,
  "view_definer": {
   "Username": "root",
   "Hostname": "127.0.0.1",
   "CurrentUser": false,
   "AuthUsername": "root",
   "AuthHostname": "%"
  },
  "view_security": 0,
  "view_select": "SELECT `s`.`a` FROM `test`.`t` LEFT JOIN `test`.`s` ON `t`.`a`=`s`.`a`",
  "view_checkoption": 1,
  "view_cols": null
 },
 "sequence": null,
 "Lock": null,
 "version": 3,
 "tiflash_replica": null
}
```

12.11.10.3 示例

以下例子将创建一个视图，并在该视图上进行查询，最后删除该视图。

```
create table t(a int, b int);
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
insert into t values(1, 1),(2,2),(3,3);
```

```
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

```
create table s(a int);
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
insert into s values(2),(3);
```

```
Query OK, 2 rows affected (0.01 sec)
Records: 2  Duplicates: 0  Warnings: 0
```

```
create view v as select s.a from t left join s on t.a = s.a;
```

```
Query OK, 0 rows affected (0.01 sec)
```

```
select * from v;
```

```
+------+
| a    |
+------+
| NULL |
|    2 |
|    3 |
+------+
3 rows in set (0.00 sec)
```

```
drop view v;
```

```
Query OK, 0 rows affected (0.02 sec)
```

12.11.10.4 局限性

目前 TiDB 中的视图有以下局限性：

- 不支持物化视图。
- TiDB 中视图为只读视图，不支持对视图进行 UPDATE、INSERT、DELETE、TRUNCATE 等写入操作。
- 对已创建的视图仅支持 DROP 的 DDL 操作，即 DROP [VIEW | TABLE]。

12.11.10.5 扩展阅读

- 创建视图
- 删除视图

12.11.11 分区表

本文介绍 TiDB 的分区表。

12.11.11.1 分区类型

本节介绍在 TiDB 中的分区类型。当前支持的类型包括 Range 分区和 Hash 分区。Range 分区可以用于解决业务中大量删除带来的性能问题，支持快速删除分区。Hash 分区则可以用于大量写入场景下的数据打散。

12.11.11.1.1 Range 分区

一个表按 Range 分区是指，对于表的每个分区中包含的所有行，按分区表达式计算的值都落在给定的范围内。Range 必须是连续的，并且不能有重叠，通过使用 `VALUES LESS THAN` 进行定义。

下列场景中，假设你要创建一个人事记录的表：

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

你可以根据需求按各种方式进行 Range 分区。其中一种方式是按 `store_id` 列进行分区：

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

PARTITION BY RANGE (store_id) (
    PARTITION p0 VALUES LESS THAN (6),
    PARTITION p1 VALUES LESS THAN (11),
    PARTITION p2 VALUES LESS THAN (16),
```

```
    PARTITION p3 VALUES LESS THAN (21)
);
```

在这个分区模式中，所有 store_id 为 1 到 5 的员工，都存储在分区 p0 里面，store_id 为 6 到 10 的员工则存储在分区 p1 里面。Range 分区要求，分区的定义必须是有序的，按从小到大递增。

新插入一行数据 (72, 'Tom', 'John', '2015-06-25', NULL, NULL, 15) 将会落到分区 p2 里面。但如果你插入一条 store_id 大于 20 的记录，则会报错，因为 TiDB 无法知晓应该将它插入到哪个分区。这种情况下，可以在建表时使用最大值：

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

PARTITION BY RANGE (store_id) (
    PARTITION p0 VALUES LESS THAN (6),
    PARTITION p1 VALUES LESS THAN (11),
    PARTITION p2 VALUES LESS THAN (16),
    PARTITION p3 VALUES LESS THAN MAXVALUE
);
```

MAXVALUE 表示一个比所有整数都大的整数。现在，所有 store_id 列大于等于 16 的记录都会存储在 p3 分区中。

你也可以按员工的职位编号进行分区，也就是使用 job_code 列的值进行分区。假设两位数字编号是用于普通员工，三位数字编号是用于办公室以及客户支持，四位数字编号是管理层职位，那么你可以这样建表：

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

PARTITION BY RANGE (job_code) (
    PARTITION p0 VALUES LESS THAN (100),
    PARTITION p1 VALUES LESS THAN (1000),
    PARTITION p2 VALUES LESS THAN (10000)
);
```

在这个例子中，所有普通员工存储在 p0 分区，办公室以及支持人员在 p1 分区，管理者在 p2 分区。

除了可以按 store_id 切分，你还可以按日期切分。例如，假设按员工离职的年份进行分区：

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

PARTITION BY RANGE ( YEAR(separated) ) (
    PARTITION p0 VALUES LESS THAN (1991),
    PARTITION p1 VALUES LESS THAN (1996),
    PARTITION p2 VALUES LESS THAN (2001),
    PARTITION p3 VALUES LESS THAN MAXVALUE
);
```

在 Range 分区中，可以基于 timestamp 列的值分区，并使用 unix_timestamp() 函数，例如：

```
CREATE TABLE quarterly_report_status (
    report_id INT NOT NULL,
    report_status VARCHAR(20) NOT NULL,
    report_updated TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
)

PARTITION BY RANGE ( UNIX_TIMESTAMP(report_updated) ) (
    PARTITION p0 VALUES LESS THAN ( UNIX_TIMESTAMP('2008-01-01 00:00:00') ),
    PARTITION p1 VALUES LESS THAN ( UNIX_TIMESTAMP('2008-04-01 00:00:00') ),
    PARTITION p2 VALUES LESS THAN ( UNIX_TIMESTAMP('2008-07-01 00:00:00') ),
    PARTITION p3 VALUES LESS THAN ( UNIX_TIMESTAMP('2008-10-01 00:00:00') ),
    PARTITION p4 VALUES LESS THAN ( UNIX_TIMESTAMP('2009-01-01 00:00:00') ),
    PARTITION p5 VALUES LESS THAN ( UNIX_TIMESTAMP('2009-04-01 00:00:00') ),
    PARTITION p6 VALUES LESS THAN ( UNIX_TIMESTAMP('2009-07-01 00:00:00') ),
    PARTITION p7 VALUES LESS THAN ( UNIX_TIMESTAMP('2009-10-01 00:00:00') ),
    PARTITION p8 VALUES LESS THAN ( UNIX_TIMESTAMP('2010-01-01 00:00:00') ),
    PARTITION p9 VALUES LESS THAN (MAXVALUE)
);
```

对于 timestamp 列，使用其它的分区表达式是不允许的。

Range 分区在下列条件之一或者多个都满足时，尤其有效：

- 删除旧数据。如果你使用之前的 employees 表的例子，你可以简单使用 ALTER TABLE employees DROP PARTITION p0; 删除所有在 1991 年以前停止继续在这家公司工作的员工记录。这会比使用 DELETE

↪ FROM employees WHERE YEAR(separated)<= 1990; 执行快得多。

- 使用包含时间或者日期的列，或者是其它按序生成的数据。
- 频繁查询分区使用的列。例如执行这样的查询 EXPLAIN SELECT COUNT(*)FROM employees WHERE ↪ separated BETWEEN '2000-01-01' AND '2000-12-31' GROUP BY store_id; 时，TiDB 可以迅速确定，只需要扫描 p2 分区的数据，因为其它的分区不满足 where 条件。

12.11.11.1.2 Hash 分区

Hash 分区主要用于保证数据均匀地分散到一定数量的分区里面。在 Range 分区中你必须为每个分区指定值的范围；在 Hash 分区中，你只需要指定分区的数量。

使用 Hash 分区时，需要在 CREATE TABLE 后面添加 PARTITION BY HASH (expr)，其中 expr 是一个返回整数的表达式。当这一列的类型是整数类型时，它可以是一个列名。此外，你很可能还需要加上 PARTITIONS num，其中 num 是一个正整数，表示将表划分多少分区。

下面的语句将创建一个 Hash 分区表，按 store_id 分成 4 个分区：

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

PARTITION BY HASH(store_id)
PARTITIONS 4;
```

如果不指定 PARTITIONS num，默认的分区数量为 1。

你也可以使用一个返回整数的 SQL 表达式。例如，你可以按入职年份分区：

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

PARTITION BY HASH( YEAR(hired) )
PARTITIONS 4;
```

最高效的 Hash 函数是作用在单列上，并且函数的单调性是跟列的值是一样递增或者递减的。

例如，date_col 是类型为 DATE 的列，表达式 TO_DAYS(date_col) 的值是直接随 date_col 的值变化的。YEAR(↪ date_col) 跟 TO_DAYS(date_col) 就不太一样，因为不是每次 date_col 变化时 YEAR(date_col) 都会得到不同的值。

作为对比，假设我们有一个类型是 INT 的 int_col 的列。考虑一下表达式 POW(5-int_col,3)+ 6，这并不是一个比较好的 Hash 函数，因为随着 int_col 的值的变化，表达式的结果不会成比例地变化。改变 int_col 的值会使表达式的结果的值变化巨大。例如，int_col 从 5 变到 6 表达式的结果变化是 -1，但是从 6 变到 7 的时候表达式的值的变化是 -7。

总而言之，表达式越接近 y = cx 的形式，它越是适合作为 Hash 函数。因为表达式越是非线性的，在各个分区上面的数据的分布越是倾向于不均匀。

理论上，Hash 分区也是可以做分区裁剪的。而实际上对于多列的情况，实现很难并且计算很耗时。因此，不推荐 Hash 分区在表达式中涉及多列。

使用 PARTITIION BY HASH 的时候，TiDB 通过表达式的结果做“取余”运算，决定数据落在哪个分区。换句话说，如果分区表达式是 expr，分区数是 num，则由 MOD(expr, num) 决定存储的分区。假设 t1 定义如下：

```
CREATE TABLE t1 (col1 INT, col2 CHAR(5), col3 DATE)
    PARTITION BY HASH( YEAR(col3) )
    PARTITIONS 4;
```

向 t1 插入一行数据，其中 col3 列的值是 ‘2005-09-15’，这条数据会被插入到分区 1 中：

```
MOD(YEAR('2005-09-01'),4)
=  MOD(2005,4)
=  1
```

12.11.11.1.3 分区对 NULL 值的处理

TiDB 允许计算结果为 NULL 的分区表达式。注意，NULL 不是一个整数类型，NULL 小于所有的整数类型值，正如 ORDER BY 的规则一样。

Range 分区对 NULL 的处理

如果插入一行到 Range 分区表，它的分区列的计算结果是 NULL，那么这一行会被插入到最小的那个分区。

```
CREATE TABLE t1 (
    c1 INT,
    c2 VARCHAR(20)
)

PARTITION BY RANGE(c1) (
    PARTITION p0 VALUES LESS THAN (0),
    PARTITION p1 VALUES LESS THAN (10),
    PARTITION p2 VALUES LESS THAN MAXVALUE
);
```

```
Query OK, 0 rows affected (0.09 sec)
```

```
select * from t1 partition(p0);
```

```
+------|--------+
| c1   | c2     |
+------|--------+
| NULL | mothra |
+------|--------+
1 row in set (0.00 sec)
```

```
select * from t1 partition(p1);
```

```
Empty set (0.00 sec)
```

```
select * from t1 partition(p2);
```

```
Empty set (0.00 sec)
```

删除 p0 后验证：

```
alter table t1 drop partition p0;
```

```
Query OK, 0 rows affected (0.08 sec)
```

```
select * from t1;
```

```
Empty set (0.00 sec)
```

Hash 分区对 NULL 的处理

在 Hash 分区中 NULL 值的处理有所不同，如果分区表达式的计算结果为 NULL，它会被当作 0 值处理。

```
CREATE TABLE th (
    c1 INT,
    c2 VARCHAR(20)
)

PARTITION BY HASH(c1)
PARTITIONS 2;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
INSERT INTO th VALUES (NULL, 'mothra'), (0, 'gigan');
```

```
Query OK, 2 rows affected (0.04 sec)
```

```
select * from th partition (p0);
```

```
+------|--------+
| c1   | c2     |
+------|--------+
| NULL | mothra |
|    0 | gigan  |
+------|--------+
2 rows in set (0.00 sec)
```

```
select * from th partition (p1);
```

```
Empty set (0.00 sec)
```

可以看到，插入的记录 `(NULL, 'mothra')` 跟 `(0, 'gigan')` 落在了同一个分区。

> **注意：**
>
> 这里 Hash 分区对 NULL 的处理跟 MySQL 的文档描述一致，但是跟 MySQL 的实际行为并不一致。也就是说，MySQL 的文档跟它的实现并不一致。
>
> TiDB 的最终行为以本文档描述为准。

12.11.11.2 分区管理

通过 `ALTER TABLE` 语句可以执行一些添加、删除、合并、切分、重定义分区的操作。

12.11.11.2.1 Range 分区管理

创建分区表：

```
CREATE TABLE members (
    id INT,
    fname VARCHAR(25),
    lname VARCHAR(25),
    dob DATE
)

PARTITION BY RANGE( YEAR(dob) ) (
    PARTITION p0 VALUES LESS THAN (1980),
    PARTITION p1 VALUES LESS THAN (1990),
    PARTITION p2 VALUES LESS THAN (2000)
);
```

删除分区：

```
ALTER TABLE members DROP PARTITION p2;
```

```
Query OK, 0 rows affected (0.03 sec)
```

清空分区：

```
ALTER TABLE members TRUNCATE PARTITION p1;
```

```
Query OK, 0 rows affected (0.03 sec)
```

> 注意：
>
> ALTER TABLE ... REORGANIZE PARTITION 在 TiDB 中暂不支持。

添加分区：

```
ALTER TABLE members ADD PARTITION (PARTITION p3 VALUES LESS THAN (2010));
```

Range 分区中，ADD PARTITION 只能在分区列表的最后面添加，如果是添加到已存在的分区范围则会报错：

```
ALTER TABLE members
    ADD PARTITION (
    PARTITION n VALUES LESS THAN (1970));
```

```
ERROR 1463 (HY000): VALUES LESS THAN value must be strictly »
   increasing for each partition
```

12.11.11.2.2 Hash 分区管理

跟 Range 分区不同，Hash 分区不能够 DROP PARTITION。

目前 TiDB 的实现暂时不支持 ALTER TABLE ... COALESCE PARTITION。对于暂不支持的分区管理语句，TiDB 会返回错误。

```
alter table members optimize partition p0;
```

```
ERROR 8200 (HY000): Unsupported optimize partition
```

12.11.11.3 分区裁剪

有一个优化叫做“分区裁剪”，它基于一个非常简单的概念：不需要扫描那些匹配不上的分区。

假设创建一个分区表 t1：

```
CREATE TABLE t1 (
    fname VARCHAR(50) NOT NULL,
    lname VARCHAR(50) NOT NULL,
    region_code TINYINT UNSIGNED NOT NULL,
    dob DATE NOT NULL
)

PARTITION BY RANGE( region_code ) (
    PARTITION p0 VALUES LESS THAN (64),
    PARTITION p1 VALUES LESS THAN (128),
    PARTITION p2 VALUES LESS THAN (192),
    PARTITION p3 VALUES LESS THAN MAXVALUE
);
```

如果你想获得这个 select 语句的结果：

```
SELECT fname, lname, region_code, dob
    FROM t1
    WHERE region_code > 125 AND region_code < 130;
```

很显然，结果必然是在分区 p1 或者 p2 里面，也就是说，我们只需要在 p1 和 p2 里面去搜索匹配的行。去掉不必要的分区就是所谓的裁剪。优化器如果能裁剪掉一部分的分区，则执行会快于处理整个不做分区的表的相同查询。

优化器可以通过 where 条件裁剪的两个场景：

- partition_column = constant
- partition_column IN (constant1, constant2, ⋯, constantN)

12.11.11.3.1 分区裁剪生效的场景

1. 分区裁剪需要使用分区表上面的查询条件，所以根据优化器的优化规则，如果查询条件不能下推到分区表，则相应的查询语句无法执行分区裁剪。

   例如：

   ```
   create table t1 (x int) partition by range (x) (
       partition p0 values less than (5),
       partition p1 values less than (10));
   create table t2 (x int);
   ```

   ```
   explain select * from t1 left join t2 on t1.x = t2.x where t2.x > 5;
   ```

在这个查询中，外连接可以简化成内连接，然后由 `t1.x = t2.x` 和 `t2.x > 5` 可以推出条件 `t1.x > 5`，于是可以分区裁剪并且只使用 `p1` 分区。

```
explain select * from t1 left join t2 on t1.x = t2.x and t2.x > 5;
```

这个查询中的 `t2.x > 5` 条件不能下推到 `t1` 分区表上面，因此 `t1` 无法分区裁剪。

2. 由于分区裁剪的规则优化是在查询计划的生成阶段，对于执行阶段才能获取到过滤条件的场景，无法利用分区裁剪的优化。

   例如：

   ```
   create table t1 (x int) partition by range (x) (
       partition p0 values less than (5),
       partition p1 values less than (10));
   ```

   ```
   explain select * from t2 where x < (select * from t1 where t2.x < t1.x and t2.x < 2);
   ```

   这个查询每从 `t2` 读取一行，都会去分区表 `t1` 上进行查询，理论上这时会满足 `t1.x > val` 的过滤条件，但实际上由于分区裁剪只作用于查询计划生成阶段，而不是执行阶段，因而不会做裁剪。

3. 由于当前实现中的一处限制，对于查询条件无法下推到 TiKV 的表达式，不支持分区裁剪。

   对于一个函数表达式 `fn(col)`，如果 TiKV 支持这个函数 `fn`，则在查询优化做谓词下推的时候，`fn(col)` 会被推到叶子节点（也就是分区），因而能够执行分区裁剪。

   如果 TiKV 不支持 `fn`，则优化阶段不会把 `fn(col)` 推到叶子节点，而是在叶子上面连接一个 Selection 节点，分区裁剪的实现没有处理这种父节点的 Selection 中的条件，因此对不能下推到 TiKV 的表达式不支持分区裁剪。

4. 对于 Hash 分区类型，只有等值比较的查询条件能够支持分区裁剪。

5. 对于 Range 分区类型，分区表达式必须是 `col` 或者 `fn(col)` 的简单形式，查询条件是 `>`、`<`、`=`、`>=`、`<=` 时才能支持分区裁剪。如果分区表达式是 `fn(col)` 形式，还要求 `fn` 必须是单调函数，才有可能分区裁剪。

   这里单调函数是指某个函数 `fn` 满足条件：对于任意 `x y`，如果 `x > y`，则 `fn(x)> fn(y)`。

   这种是严格递增的单调函数，非严格递增的单调函数也可以符合分区裁剪要求，只要函数 `fn` 满足：对于任意 `x y`，如果 `x > y`，则 `fn(x)>= fn(y)`。

   理论上所有满足单调条件（严格或者非严格）的函数都是可以支持分区裁剪。实际上，目前 TiDB 已经支持的单调函数只有：

   ```
   unix_timestamp
   to_days
   ```

   例如，分区表达式是简单列的情况：

   ```
   create table t (id int) partition by range (id) (
       partition p0 values less than (5),
       partition p1 values less than (10));
   select * from t where t > 6;
   ```

   分区表达式是 `fn(col)` 的形式，`fn` 是我们支持的单调函数 `to_days`：

```
create table t (dt datetime) partition by range (to_days(id)) (
    partition p0 values less than (to_days('2020-04-01')),
    partition p1 values less than (to_days('2020-05-01')));
select * from t where t > '2020-04-18';
```

有一处例外是 `floor(unix_timestamp(ts))` 作为分区表达式，TiDB 针对这个场景做了特殊处理，可以支持分区裁剪。

```
create table t (ts timestamp(3) not null default current_timestamp(3))
partition by range (floor(unix_timestamp(ts))) (
    partition p0 values less than (unix_timestamp('2020-04-01 00:00:00')),
    partition p1 values less than (unix_timestamp('2020-05-01 00:00:00')));
select * from t where t > '2020-04-18 02:00:42.123';
```

12.11.11.4 分区选择

SELECT 语句中支持分区选择。实现通过使用一个 PARTITION 选项实现。

```
CREATE TABLE employees  (
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    fname VARCHAR(25) NOT NULL,
    lname VARCHAR(25) NOT NULL,
    store_id INT NOT NULL,
    department_id INT NOT NULL
)

PARTITION BY RANGE(id)  (
    PARTITION p0 VALUES LESS THAN (5),
    PARTITION p1 VALUES LESS THAN (10),
    PARTITION p2 VALUES LESS THAN (15),
    PARTITION p3 VALUES LESS THAN MAXVALUE
);

INSERT INTO employees VALUES
    ('', 'Bob', 'Taylor', 3, 2), ('', 'Frank', 'Williams', 1, 2),
    ('', 'Ellen', 'Johnson', 3, 4), ('', 'Jim', 'Smith', 2, 4),
    ('', 'Mary', 'Jones', 1, 1), ('', 'Linda', 'Black', 2, 3),
    ('', 'Ed', 'Jones', 2, 1), ('', 'June', 'Wilson', 3, 1),
    ('', 'Andy', 'Smith', 1, 3), ('', 'Lou', 'Waters', 2, 4),
    ('', 'Jill', 'Stone', 1, 4), ('', 'Roger', 'White', 3, 2),
    ('', 'Howard', 'Andrews', 1, 2), ('', 'Fred', 'Goldberg', 3, 3),
    ('', 'Barbara', 'Brown', 2, 3), ('', 'Alice', 'Rogers', 2, 2),
    ('', 'Mark', 'Morgan', 3, 3), ('', 'Karen', 'Cole', 3, 2);
```

你可以查看存储在分区 p1 中的行：

```
SELECT * FROM employees PARTITION (p1);
```

```
+----|-------|--------|----------|---------------+
| id | fname | lname  | store_id | department_id |
+----|-------|--------|----------|---------------+
|  5 | Mary  | Jones  |        1 |             1 |
|  6 | Linda | Black  |        2 |             3 |
|  7 | Ed    | Jones  |        2 |             1 |
|  8 | June  | Wilson |        3 |             1 |
|  9 | Andy  | Smith  |        1 |             3 |
+----|-------|--------|----------|---------------+
5 rows in set (0.00 sec)
```

如果希望获得多个分区中的行，可以提供分区名的列表，用逗号隔开。例如，`SELECT * FROM employees PARTITION (p1, p2)` 返回分区 p1 和 p2 的所有行。

使用分区选择时，仍然可以使用 where 条件，以及 ORDER BY 和 LIMIT 等选项。使用 HAVING 和 GROUP BY 等聚合选项也是支持的。

```
SELECT * FROM employees PARTITION (p0, p2)
    WHERE lname LIKE 'S%';
```

```
+----|-------|-------|----------|---------------+
| id | fname | lname | store_id | department_id |
+----|-------|-------|----------|---------------+
|  4 | Jim   | Smith |        2 |             4 |
| 11 | Jill  | Stone |        1 |             4 |
+----|-------|-------|----------|---------------+
2 rows in set (0.00 sec)
```

```
SELECT id, CONCAT(fname, ' ', lname) AS name
    FROM employees PARTITION (p0) ORDER BY lname;
```

```
+----|----------------+
| id | name           |
+----|----------------+
|  3 | Ellen Johnson  |
|  4 | Jim Smith      |
|  1 | Bob Taylor     |
|  2 | Frank Williams |
+----|----------------+
4 rows in set (0.06 sec)
```

```
SELECT store_id, COUNT(department_id) AS c
    FROM employees PARTITION (p1,p2,p3)
    GROUP BY store_id HAVING c > 4;
```

```
+---|----------+
| c | store_id |
+---|----------+
| 5 |        2 |
| 5 |        3 |
+---|----------+
2 rows in set (0.00 sec)
```

分支选择支持所有类型的分区表，无论是 Range 分区或是 Hash 分区等。对于 Hash 分区，如果没有指定分区名，会自动使用 p0、p1、p2、……、或 pN-1 作为分区名。

在 INSERT ... SELECT 的 SELECT 中也是可以使用分区选择的。

12.11.11.5 分区的约束和限制

本节介绍当前 TiDB 分区表的一些约束和限制。

12.11.11.5.1 分区键，主键和唯一键

本节讨论分区键，主键和唯一键之间的关系。一句话总结它们之间的关系要满足的规则：分区表的每个唯一键，必须包含分区表达式中用到的所有列。

> every unique key on the table must use every column in the table's partitioning expression.

这里所指的唯一也包含了主键，因为根据主键的定义，主键必须是唯一的。例如，下面这些建表语句就是无效的：

```
CREATE TABLE t1 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    UNIQUE KEY (col1, col2)
)

PARTITION BY HASH(col3)
PARTITIONS 4;

CREATE TABLE t2 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
```

```
    UNIQUE KEY (col1),
    UNIQUE KEY (col3)
)

PARTITION BY HASH(col1 + col3)
PARTITIONS 4;
```

它们都是有唯一键但没有包含所有分区键的。

下面是一些合法的语句的例子：

```
CREATE TABLE t1 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    UNIQUE KEY (col1, col2, col3)
)

PARTITION BY HASH(col3)
PARTITIONS 4;

CREATE TABLE t2 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    UNIQUE KEY (col1, col3)
)

PARTITION BY HASH(col1 + col3)
PARTITIONS 4;
```

下例中会产生一个报错：

```
CREATE TABLE t3 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    UNIQUE KEY (col1, col2),
    UNIQUE KEY (col3)
)

PARTITION BY HASH(col1 + col3)
    PARTITIONS 4;
```

```
ERROR 1491 (HY000): A PRIMARY KEY must include all columns in the table's partitioning function
```

原因是 `col1` 和 `col3` 出现在分区键中，但是几个唯一键定义并没有完全包含它们，做如下修改后语句即为合法:

```
CREATE TABLE t3 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    UNIQUE KEY (col1, col2, col3),
    UNIQUE KEY (col1, col3)
)

PARTITION BY HASH(col1 + col3)
    PARTITIONS 4;
```

下面这个表就没法做分区了，因为无论如何都不可能找到满足条件的分区键：

```
CREATE TABLE t4 (
    col1 INT NOT NULL,
    col2 INT NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    UNIQUE KEY (col1, col3),
    UNIQUE KEY (col2, col4)
);
```

根据定义，主键也是唯一键，下面两个建表语句是无效的：

```
CREATE TABLE t5 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    PRIMARY KEY(col1, col2)
)

PARTITION BY HASH(col3)
PARTITIONS 4;

CREATE TABLE t6 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
```

```
    PRIMARY KEY(col1, col3),
    UNIQUE KEY(col2)
)

PARTITION BY HASH( YEAR(col2) )
PARTITIONS 4;
```

以上两个例子中，主键都没有包含分区表达式中的全部的列，在主键中补充缺失列后语句即为合法:

```
CREATE TABLE t5 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    PRIMARY KEY(col1, col2, col3)
)

PARTITION BY HASH(col3)
PARTITIONS 4;

CREATE TABLE t6 (
    col1 INT NOT NULL,
    col2 DATE NOT NULL,
    col3 INT NOT NULL,
    col4 INT NOT NULL,
    PRIMARY KEY(col1, col2, col3),
    UNIQUE KEY(col2)
)

PARTITION BY HASH( YEAR(col2) )
PARTITIONS 4;
```

如果既没有主键，也没有唯一键，则不存在这个限制。

DDL 变更时，添加唯一索引也需要考虑到这个限制。比如创建了这样一个表：

```
CREATE TABLE t_no_pk (c1 INT, c2 INT)
    PARTITION BY RANGE(c1) (
        PARTITION p0 VALUES LESS THAN (10),
        PARTITION p1 VALUES LESS THAN (20),
        PARTITION p2 VALUES LESS THAN (30),
        PARTITION p3 VALUES LESS THAN (40)
    );
```

```
Query OK, 0 rows affected (0.12 sec)
```

通过 ALTER TABLE 添加非唯一索引是可以的。但是添加唯一索引时，唯一索引里面必须包含 c1 列。

使用分区表时，前缀索引是不能指定为唯一属性的：

```
CREATE TABLE t (a varchar(20), b blob,
    UNIQUE INDEX (a(5)))
    PARTITION by range columns (a) (
    PARTITION p0 values less than ('aaaaa'),
    PARTITION p1 values less than ('bbbbb'),
    PARTITION p2 values less than ('ccccc'));
```

```
ERROR 1503 (HY000): A UNIQUE INDEX must include all columns in the table's partitioning function
```

12.11.11.5.2 关于函数的分区限制

只有以下函数可以用于分区表达式：

```
ABS()
CEILING()
DATEDIFF()
DAY()
DAYOFMONTH()
DAYOFWEEK()
DAYOFYEAR()
EXTRACT() (see EXTRACT() function with WEEK specifier)
FLOOR()
HOUR()
MICROSECOND()
MINUTE()
MOD()
MONTH()
QUARTER()
SECOND()
TIME_TO_SEC()
TO_DAYS()
TO_SECONDS()
UNIX_TIMESTAMP() (with TIMESTAMP columns)
WEEKDAY()
YEAR()
YEARWEEK()
```

12.11.11.5.3 兼容性

目前 TiDB 里面只实现了 Range 分区和 Hash 分区，其它的 MySQL 分区类型比如 List 分区和 Key 分区尚不支持。

对于 Range Columns 类型的分区表，目前只支持单列的场景。

分区管理方面，只要底层实现可能会涉及数据挪动的操作，目前都暂不支持。包括且不限于：调整 Hash 分区表的分区数量，修改 Range 分区表的范围，合并分区，交换分区等。

对于暂不支持的分区类型，在 TiDB 中建表时会忽略分区信息，以普通表的形式创建，并且会报 Warning。

Load Data 暂时不支持分区选择。

```
create table t (id int, val int) partition by hash(id) partitions 4;
```

普通的 Load Data 操作在 TiDB 中是支持的，如下：

```
load local data infile "xxx" into t ...
```

但 Load Data 不支持分区选择操作：

```
load local data infile "xxx" into t partition (p1)...
```

对于分区表，`select * from t` 的返回结果是分区之间无序的。这跟 MySQL 不同，MySQL 的返回结果是分区之间有序，分区内部无序。

```
create table t (id int, val int) partition by range (id) (
    partition p0 values less than (3),
    partition p1 values less than (7),
    partition p2 values less than (11));
```

```
Query OK, 0 rows affected (0.10 sec)
```

```
insert into t values (1, 2), (3, 4),(5, 6),(7,8),(9,10);
```

```
Query OK, 5 rows affected (0.01 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

TiDB 每次返回结果会不同，例如：

```
select * from t;
```

```
+------|------+
| id   | val  |
+------|------+
|    7 |    8 |
|    9 |   10 |
|    1 |    2 |
|    3 |    4 |
|    5 |    6 |
+------|------+
5 rows in set (0.00 sec)
```

MySQL 的返回结果：

```
select * from t;
```

```
+------|------+
| id   | val  |
+------|------+
|    1 |    2 |
|    3 |    4 |
|    5 |    6 |
|    7 |    8 |
|    9 |   10 |
+------|------+
5 rows in set (0.00 sec)
```

环境变量 `tidb_enable_table_partition` 可以控制是否启用分区表功能。如果该变量设置为 `off`，则建表时会忽略分区信息，以普通表的方式建表。

该变量仅作用于建表，已经建表之后再修改该变量无效。详见系统变量和语法。

12.11.12 字符集和排序规则

本文介绍了 TiDB 中支持的字符集和排序规则。

12.11.12.1 字符集和排序规则的概念

字符集 (character set) 是符号与编码的集合。TiDB 中的默认字符集是 utf8mb4，与 MySQL 8.0 及更高版本中的默认字符集匹配。

排序规则 (collation) 是在字符集中比较字符以及字符排序顺序的规则。例如，在二进制排序规则中，比较 A 和 a 的结果是不一样的：

```
SET NAMES utf8mb4 COLLATE utf8mb4_bin;
SELECT 'A' = 'a';
SET NAMES utf8mb4 COLLATE utf8mb4_general_ci;
SELECT 'A' = 'a';
```

```
mysql> SELECT 'A' = 'a';
+-----------+
| 'A' = 'a' |
+-----------+
|         0 |
+-----------+
1 row in set (0.00 sec)

mysql> SET NAMES utf8mb4 COLLATE utf8mb4_general_ci;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT 'A' = 'a';
+-----------+
```

```
| 'A' = 'a' |
+-----------+
|         1 |
+-----------+
1 row in set (0.00 sec)
```

TiDB 默认使用二进制排序规则。这一点与 MySQL 不同，MySQL 默认使用不区分大小写的排序规则。

12.11.12.2 支持的字符集和排序规则

目前 TiDB 支持以下字符集：

```
SHOW CHARACTER SET;
```

```
+---------|---------------|--------------------|--------+
| Charset | Description   | Default collation | Maxlen |
+---------|---------------|--------------------|--------+
| utf8    | UTF-8 Unicode | utf8_bin          |      3 |
| utf8mb4 | UTF-8 Unicode | utf8mb4_bin       |      4 |
| ascii   | US ASCII      | ascii_bin         |      1 |
| latin1  | Latin1        | latin1_bin        |      1 |
| binary  | binary        | binary            |      1 |
+---------|---------------|--------------------|--------+
5 rows in set (0.00 sec)
```

TiDB 支持以下排序规则：

```
mysql> show collation;
+-------------+---------+------+---------+----------+---------+
| Collation   | Charset | Id   | Default | Compiled | Sortlen |
+-------------+---------+------+---------+----------+---------+
| utf8mb4_bin | utf8mb4 |   46 | Yes     | Yes      |       1 |
| latin1_bin  | latin1  |   47 | Yes     | Yes      |       1 |
| binary      | binary  |   63 | Yes     | Yes      |       1 |
| ascii_bin   | ascii   |   65 | Yes     | Yes      |       1 |
| utf8_bin    | utf8    |   83 | Yes     | Yes      |       1 |
+-------------+---------+------+---------+----------+---------+
5 rows in set (0.01 sec)
```

> **警告：**
>
> TiDB 会错误地将 `latin1` 视为 `utf8` 的子集。当用户存储不同于 `latin1` 和 `utf8` 编码的字符时，可能会导致意外情况出现。因此强烈建议使用 `utf8mb4` 字符集。详情参阅 TiDB #18955。

> **注意：**
>
> TiDB 中的默认排序规则（后缀为 _bin 的二进制排序规则）与 MySQL 中的默认排序规则不同，后者通常是一般排序规则，后缀为 _general_ci。当用户指定了显式字符集，但依赖于待选的隐式默认排序规则时，这个差异可能导致兼容性问题。

利用以下的语句可以查看字符集对应的排序规则（以下是新的排序规则框架）下的结果：

```
SHOW COLLATION WHERE Charset = 'utf8mb4';
```

```
+--------------------+---------+------+---------+----------+---------+
| Collation          | Charset | Id   | Default | Compiled | Sortlen |
+--------------------+---------+------+---------+----------+---------+
| utf8mb4_bin        | utf8mb4 |   46 | Yes     | Yes      |       1 |
| utf8mb4_general_ci | utf8mb4 |   45 |         | Yes      |       1 |
+--------------------+---------+------+---------+----------+---------+
2 rows in set (0.00 sec)
```

12.11.12.3 TiDB 中的 utf8 和 utf8mb4

MySQL 限制字符集 utf8 为最多 3 个字节。这足以存储在基本多语言平面 (BMP) 中的字符，但不足以存储表情符号等字符。因此，建议改用字符集utf8mb4。

默认情况下，TiDB 同样限制字符集 utf8 为最多 3 个字节，以确保 TiDB 中创建的数据可以在 MySQL 中顺利恢复。你可以禁用此功能，方法是在 TiDB 配置文件中将 check-mb4-value-in-utf8 的值更改为 FALSE。

以下示例演示了在表中插入 4 字节的表情符号字符时的默认行为。utf8 字符集下 INSERT 语句不能执行，utf8mb4 字符集下可以执行 INSERT 语句：

```
mysql> CREATE TABLE utf8_test (
    ->  c char(1) NOT NULL
    -> ) CHARACTER SET utf8;
Query OK, 0 rows affected (0.09 sec)

mysql> CREATE TABLE utf8m4_test (
    ->  c char(1) NOT NULL
    -> ) CHARACTER SET utf8mb4;
Query OK, 0 rows affected (0.09 sec)

mysql> INSERT INTO utf8_test VALUES ('');
ERROR 1366 (HY000): incorrect utf8 value f09f9889( ) for column c
mysql> INSERT INTO utf8m4_test VALUES ('');
Query OK, 1 row affected (0.02 sec)

mysql> SELECT char_length(c), length(c), c FROM utf8_test;
```

```
Empty set (0.01 sec)

mysql> SELECT char_length(c), length(c), c FROM utf8m4_test;
+----------------+-----------+------+
| char_length(c) | length(c) | c    |
+----------------+-----------+------+
|              1 |         4 | 𐀀    |
+----------------+-----------+------+
1 row in set (0.00 sec)
```

12.11.12.4 不同范围的字符集和排序规则

字符集和排序规则可以在设置在不同的层次。

12.11.12.4.1 数据库的字符集和排序规则

每个数据库都有相应的字符集和排序规则。数据库的字符集和排序规则可以通过以下语句来设置：

```
CREATE DATABASE db_name
    [[DEFAULT] CHARACTER SET charset_name]
    [[DEFAULT] COLLATE collation_name]

ALTER DATABASE db_name
    [[DEFAULT] CHARACTER SET charset_name]
    [[DEFAULT] COLLATE collation_name]
```

在这里 DATABASE 可以跟 SCHEMA 互换使用。

不同的数据库之间可以使用不一样的字符集和排序规则。

通过系统变量 character_set_database 和 collation_database 可以查看到当前数据库的字符集以及排序规则：

```
CREATE SCHEMA test1 CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci;
```

```
Query OK, 0 rows affected (0.09 sec)
```

```
USE test1;
```

```
Database changed
```

```
SELECT @@character_set_database, @@collation_database;
```

```
+--------------------------|----------------------+
| @@character_set_database | @@collation_database |
+--------------------------|----------------------+
| utf8mb4                  | utf8mb4_general_ci   |
```

```
+--------------------------|----------------------+
1 row in set (0.00 sec)
```

```
CREATE SCHEMA test2 CHARACTER SET latin1 COLLATE latin1_bin;
```

```
Query OK, 0 rows affected (0.09 sec)
```

```
USE test2;
```

```
Database changed
```

```
SELECT @@character_set_database, @@collation_database;
```

```
+--------------------------|----------------------+
| @@character_set_database | @@collation_database |
+--------------------------|----------------------+
| latin1                   | latin1_bin           |
+--------------------------|----------------------+
1 row in set (0.00 sec)
```

在 INFORMATION_SCHEMA 中也可以查看到这两个值：

```
SELECT DEFAULT_CHARACTER_SET_NAME, DEFAULT_COLLATION_NAME
FROM INFORMATION_SCHEMA.SCHEMATA WHERE SCHEMA_NAME = 'db_name';
```

12.11.12.4.2 表的字符集和排序规则

表的字符集和排序规则可以通过以下语句来设置：

```
CREATE TABLE tbl_name (column_list)
    [[DEFAULT] CHARACTER SET charset_name]
    [COLLATE collation_name]]

ALTER TABLE tbl_name
    [[DEFAULT] CHARACTER SET charset_name]
    [COLLATE collation_name]
```

例如：

```
CREATE TABLE t1(a int) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci;
```

```
Query OK, 0 rows affected (0.08 sec)
```

如果表的字符集和排序规则没有设置，那么数据库的字符集和排序规则就作为其默认值。

12.11.12.4.3 列的字符集和排序规则

列的字符集和排序规则的语法如下：

```
col_name {CHAR | VARCHAR | TEXT} (col_length)
    [CHARACTER SET charset_name]
    [COLLATE collation_name]

col_name {ENUM | SET} (val_list)
    [CHARACTER SET charset_name]
    [COLLATE collation_name]
```

如果列的字符集和排序规则没有设置，那么表的字符集和排序规则就作为其默认值。

12.11.12.4.4 字符串的字符集和排序规则

每一个字符串都对应一个字符集和一个排序规则，在使用字符串时指此选项可选，如下：

```
[_charset_name]'string' [COLLATE collation_name]
```

示例如下：

```
SELECT 'string';
SELECT _utf8mb4'string';
SELECT _utf8mb4'string' COLLATE utf8mb4_general_ci;
```

规则如下：

- 规则 1：如果指定了 CHARACTER SET charset_name 和 COLLATE collation_name，则直接使用 charset_name 字符集和 collation_name 排序规则。
- 规则 2：如果指定了 CHARACTER SET charset_name 且未指定 COLLATE collation_name，则使用 charset_name 字符集和 charset_name 对应的默认排序规则。
- 规则 3：如果 CHARACTER SET charset_name 和 COLLATE collation_name 都未指定，则使用 character_set_connection 和 collation_connection 系统变量给出的字符集和排序规则。

12.11.12.4.5 客户端连接的字符集和排序规则

- 服务器的字符集和排序规则可以通过系统变量 character_set_server 和 collation_server 获取。
- 数据库的字符集和排序规则可以通过环境变量 character_set_database 和 collation_database 获取。

对于每一个客户端的连接，也有相应的变量表示字符集和排序规则：character_set_connection 和 collation_connection。

character_set_client 代表客户端的字符集。在返回结果前，服务端会把结果根据 character_set_results 转换成对应的字符集，包括结果的元信息等。

可以用以下的语句来影响这些跟客户端相关的字符集变量：

- SET NAMES 'charset_name' [COLLATE 'collation_name']

  SET NAMES 用来设定客户端会在之后的请求中使用的字符集。SET NAMES utf8mb4 表示客户端会在接下来的请求中，都使用 utf8mb4 字符集。服务端也会在之后返回结果的时候使用 utf8mb4 字符集。SET NAMES 'charset_name' 语句其实等于下面语句的组合：

```
SET character_set_client = charset_name;
SET character_set_results = charset_name;
SET character_set_connection = charset_name;
```

  COLLATE 是可选的，如果没有提供，将会用 charset_name 对应的默认排序规则设置 collation_connection ↪ 。

- SET CHARACTER SET 'charset_name'

  跟 SET NAMES 类似，等价于下面语句的组合：

```
SET character_set_client = charset_name;
SET character_set_results = charset_name;
SET charset_connection = @@charset_database;
SET collation_connection = @@collation_database;
```

12.11.12.5 服务器、数据库、表、列、字符串的字符集和排序规则的优先级

优先级从高到低排列顺序为：

字符串 > 列 > 表 > 数据库 > 服务器

12.11.12.6 字符集和排序规则的通用选择规则

- 规则 1：如果指定了 CHARACTER SET charset_name 和 COLLATE collation_name，则直接使用 charset_name 字符集和 collation_name 排序规则。
- 规则 2：如果指定了 CHARACTER SET charset_name 且未指定 COLLATE collation_name，则使用 charset_name 字符集和 charset_name 对应的默认排序规则。
- 规则 3：如果 CHARACTER SET charset_name 和 COLLATE collation_name 都未指定，则使用更高优先级的字符集和排序规则。

12.11.12.7 字符合法性检查

当指定的字符集为 utf8 或 utf8mb4 时，TiDB 仅支持合法的 utf8 字符。对于不合法的字符，会报错：incorrect ↪ utf8 value。该字符合法性检查与 MySQL 8.0 兼容，与 MySQL 5.7 及以下版本不兼容。

如果不希望报错，可以通过 set @@tidb_skip_utf8_check=1; 跳过字符检查。

12.11.12.8 排序规则支持

排序规则的语法支持和语义支持受到配置项new_collations_enabled_on_first_bootstrap 的影响。这里语法支持和语义支持有所区别。语法支持是指 TiDB 能够解析和设置排序规则；而语义支持是指 TiDB 能够在比较字符串时正确地使用排序规则。

在 4.0 版本之前，TiDB 只提供了旧的排序规则框架，能够在语法上支持的绝大部分 MySQL 排序规则，但语义上所有的排序规则都当成二进制排序规则。

4.0 版本中，TiDB 增加了新的排序规则框架用于在语义上支持不同的排序规则，保证字符串比较时严格遵循对应的排序规则，详情请见下文。

12.11.12.8.1 旧框架下的排序规则支持

在 4.0 版本之前，TiDB 中可以指定大部分 MySQL 中的排序规则，并把这些排序规则按照默认排序规则处理，即以编码字节序为字符定序。和 MySQL 不同的是，TiDB 在比较字符前按照排序规则的 PADDING 属性将字符末尾的空格删除，因此会造成以下的行为区别：

```
CREATE TABLE t(a varchar(20) charset utf8mb4 collate utf8mb4_general_ci PRIMARY KEY);
Query OK, 0 rows affected
INSERT INTO t VALUES ('A');
Query OK, 1 row affected
INSERT INTO t VALUES ('a');
Query OK, 1 row affected # TiDB 会执行成功，而在 MySQL 中，则由于 utf8mb4_general_ci 大小写不敏感
    ↪ ，报错 Duplicate entry 'a'。
INSERT INTO t VALUES ('a ');
Query OK, 1 row affected # TiDB 会执行成功，而在 MySQL 中，则由于补齐空格比较，报错 Duplicate
    ↪ entry 'a '。
```

12.11.12.8.2 新框架下的排序规则支持

TiDB 4.0 新增了完整的排序规则支持框架，从语义上支持了排序规则，并新增了配置开关new_collations_enabled_on_first_boot ↪ ，在集群初次初始化时决定是否启用新排序规则框架。在该配置开关打开之后初始化集群，可以通过 mysql.tidb 表中的 new_collation_enabled 变量确认是否启用新排序规则框架：

```
SELECT VARIABLE_VALUE FROM mysql.tidb WHERE VARIABLE_NAME='new_collation_enabled';
```

```
+----------------+
| VARIABLE_VALUE |
+----------------+
| True           |
+----------------+
1 row in set (0.00 sec)
```

在新的排序规则框架下，TiDB 能够支持 utf8_general_ci 和 utf8mb4_general_ci 这两种排序规则，与 MySQL 兼容。

使用 utf8_general_ci 或者 utf8mb4_general_ci 时，字符串之间的比较是大小写不敏感 (case-insensitive) 和口音不敏感 (accent-insensitive) 的。同时，TiDB 还修正了排序规则的 PADDING 行为：

```
CREATE TABLE t(a varchar(20) charset utf8mb4 collate utf8mb4_general_ci PRIMARY KEY);
Query OK, 0 rows affected (0.00 sec)
INSERT INTO t VALUES ('A');
Query OK, 1 row affected (0.00 sec)
```

```
INSERT INTO t VALUES ('a');
ERROR 1062 (23000): Duplicate entry 'a' for key 'PRIMARY' # TiDB 兼容了 MySQL 的 case insensitive
    ↪  collation。
INSERT INTO t VALUES ('a ');
ERROR 1062 (23000): Duplicate entry 'a ' for key 'PRIMARY' # TiDB 修正了 `PADDING` 行为，与 MySQL
    ↪  兼容。
```

> **注意：**
>
> TiDB 中 padding 的实现方式与 MySQL 的不同。在 MySQL 中，padding 是通过补齐空格实现的。而在 TiDB 中 padding 是通过裁剪掉末尾的空格来实现的。两种做法在绝大多数情况下是一致的，唯一的例外是字符串尾部包含小于空格 (0x20) 的字符时，例如 `'a' < 'a\t'` 在 TiDB 中的结果为 1，而在 MySQL 中，其等价于 `'a ' < 'a\t'`，结果为 0。

12.11.12.9 表达式中排序规则的 Coercibility 值

如果一个表达式涉及多个不同排序规则的子表达式时，需要对计算时用的排序规则进行推断，规则如下：

- 显式 COLLATE 子句的 coercibility 值为 0。
- 如果两个字符串的排序规则不兼容，这两个字符串 concat 结果的 coercibility 值为 1。目前所实现的排序规则都是互相兼容的。
- 列或者 CAST()、CONVERT() 和 BINARY() 的排序规则的 coercibility 值为 2。
- 系统常量（USER() 或者 VERSION() 返回的字符串）的 coercibility 值为 3。
- 常量的 coercibility 值为 4。
- 数字或者中间变量的 coercibility 值为 5。
- NULL 或者由 NULL 派生出的表达式的 coercibility 值为 6。

在推断排序规则时，TiDB 优先使用 coercibility 值较低的表达式的排序规则。如果 coercibility 值相同，则按以下优先级确定排序规则：

binary > utf8mb4_bin > utf8mb4_general_ci > utf8_bin > utf8_general_ci > latin1_bin > ascii_bin

如果两个子表达式的排序规则不相同，而且表达式的 coercibility 值都为 0 时，TiDB 无法推断排序规则并报错。

12.11.12.10 COLLATE 子句

TiDB 支持使用 COLLATE 子句来指定一个表达式的排序规则，该表达式的 coercibility 值为 0，具有最高的优先级。示例如下：

```
SELECT 'a' = _utf8mb4 'A' collate utf8mb4_general_ci;
```

```
+-----------------------------------------------+
| 'a' = _utf8mb4 'A' collate utf8mb4_general_ci |
+-----------------------------------------------+
```

```
|                                                1 |
+--------------------------------------------------+
1 row in set (0.00 sec)
```

更多细节，参考 Connection Character Sets and Collations。

12.11.13 系统表

12.11.13.1 mysql Schema

mysql 库里存储的是 TiDB 系统表。该设计类似于 MySQL 中的 mysql 库，其中 mysql.user 之类的表可以直接编辑。该库还包含许多 MySQL 的扩展表。

12.11.13.1.1 权限系统表

这些系统表里面包含了用户账户以及相应的授权信息：

- user 用户账户，全局权限，以及其它一些非权限的列
- db 数据库级别的权限
- tables_priv 表级的权限
- columns_priv 列级的权限

12.11.13.1.2 服务端帮助信息系统表

- help_topic 目前为空

12.11.13.1.3 统计信息相关系统表

- stats_buckets 统计信息的桶
- stats_histograms 统计信息的直方图
- stats_meta 表的元信息，比如总行数和修改数

12.11.13.1.4 GC Worker 相关系统表

- gc_delete_range

12.11.13.1.5 其它系统表

- GLOBAL_VARIABLES 全局系统变量表
- tidb 用于 TiDB 在 bootstrap 的时候记录相关版本信息

12.11.13.2 INFORMATION_SCHEMA

12.11.13.2.1 Information Schema

Information Schema 提供了一种查看系统元数据的 ANSI 标准方法。除了包含与 MySQL 兼容的表外，TiDB 还提供了许多自定义的 INFORMATION_SCHEMA 表。

许多 INFORMATION_SCHEMA 表都有相应的 SHOW 命令。查询 INFORMATION_SCHEMA 的好处是可以在表之间进行 join 操作。

与 MySQL 兼容的表

表名	描述
CHARACTER_SETS	提供 TiDB 支持的字符集列表。
COLLATIONS	提供 TiDB 支持的排序规则列表。
COLLATION_CHARACTER_SET_APPLICABILITY	说明哪些排序规则适用于哪些字符集。
COLUMNS	提供所有表中列的列表。
COLUMN_PRIVILEGES	TiDB 未实现，返回零行。
COLUMN_STATISTICS	TiDB 未实现，返回零行。
ENGINES	提供支持的存储引擎列表。
EVENTS	TiDB 未实现，返回零行。
FILES	TiDB 未实现，返回零行。
GLOBAL_STATUS	TiDB 未实现，返回零行。
GLOBAL_VARIABLES	TiDB 未实现，返回零行。
KEY_COLUMN_USAGE	描述列的键约束，例如主键约束。
OPTIMIZER_TRACE	TiDB 未实现，返回零行。
PARAMETERS	TiDB 未实现，返回零行。
PARTITIONS	提供表分区的列表。
PLUGINS	TiDB 未实现，返回零行。
PROCESSLIST	提供与 SHOW PROCESSLIST 命令类似的信息。
PROFILING	TiDB 未实现，返回零行。
REFERENTIAL_CONSTRAINTS	TiDB 未实现，返回零行。
ROUTINES	TiDB 未实现，返回零行。
SCHEMATA	提供与 SHOW DATABASES 命令类似的信息。
SCHEMA_PRIVILEGES	TiDB 未实现，返回零行。
SESSION_STATUS	TiDB 未实现，返回零行。
SESSION_VARIABLES	提供与 SHOW SESSION VARIABLES 命令类似的功能。
STATISTICS	提供有关表索引的信息。
TABLES	提供当前用户可见的表的列表。类似于 SHOW TABLES。
TABLESPACES	TiDB 未实现，返回零行。
TABLE_CONSTRAINTS	提供有关主键、唯一索引和外键的信息。
TABLE_PRIVILEGES	TiDB 未实现，返回零行。
TRIGGERS	TiDB 未实现，返回零行。
USER_PRIVILEGES	汇总与当前用户相关的权限。
VIEWS	提供当前用户可见的视图列表。类似于 SHOW FULL TABLES WHERE table_type = '

TiDB 中的扩展表

表名	描述
ANALYZE_STATUS	提供有关收集统计信息的任务的信息。
CLUSTER_CONFIG	提供有关整个 TiDB 集群的配置设置的详细信息。
CLUSTER_HARDWARE	提供在每个 TiDB 组件上发现的底层物理硬件的详细信息。
CLUSTER_INFO	提供当前集群拓扑的详细信息。
CLUSTER_LOAD	提供集群中 TiDB 服务器的当前负载信息。
CLUSTER_LOG	提供整个 TiDB 集群的日志。
CLUSTER_PROCESSLIST	提供 PROCESSLIST 表的集群级别的视图。
CLUSTER_SLOW_QUERY	提供 SLOW_QUERY 表的集群级别的视图。
CLUSTER_STATEMENTS_SUMMARY	提供 STATEMENTS_SUMMARY ↪ 表的集群级别的视图。
CLUSTER_STATEMENTS_SUMMARY_HISTORY	提供 CLUSTER_STATEMENTS_SUMMARY_HIS ↪ 表的集群级别的视图。
CLUSTER_SYSTEMINFO	提供集群中服务器的内核参数配置的详细信息。

表名	描述
DDL_JOBS	提供与 ADMIN SHOW DDL JOBS 类似的输出。
INSPECTION_RESULT	触发内部诊断检查。
INSPECTION_RULES	进行的内部诊断检查的列表。
INSPECTION_SUMMARY	重要监视指标的摘要报告。
METRICS_SUMMARY	从 Prometheus 获取的指标的摘要。
METRICS_SUMMARY_BY_LABEL	参见 METRICS_SUMMARY 表。
METRICS_TABLES	为 METRICS_SCHEMA 中的表提供 PromQL 定义。
SEQUENCES	描述了基于 MariaDB 实现的 TiDB 序列。
SLOW_QUERY	提供当前 TiDB 服务器上慢查询的信息。
STATEMENTS_SUMMARY	类似于 MySQL 中的 performance_schema 语句摘要。
STATEMENTS_SUMMARY_HISTORY	类似于 MySQL 中的 performance_schema 语句摘要历史。
TABLE_STORAGE_STATS	提供存储的表的大小的详细信息。

表名	描述
TIDB_HOT_REGIONS	提供有关哪些 Region 访问次数最多的统计信息。
TIDB_INDEXES	提供有关 TiDB 表的索引信息。
TIDB_SERVERS_INFO	提供 TiDB 服务器的列表
TIFLASH_REPLICA	提供有关 TiFlash 副本的详细信息。
TIKV_REGION_PEERS	提供 Region 存储位置的详细信息。
TIKV_REGION_STATUS	提供 Region 的统计信息。
TIKV_STORE_STATUS	提供 TiKV 服务器的基本信息。

12.11.13.2.2 ANALYZE_STATUS

ANALYZE_STATUS 表提供正在执行的收集统计信息的任务以及有限条历史任务记录。

```
USE information_schema;
DESC analyze_status;
```

```
+----------------+---------------------+------+------+---------+-------+
| Field          | Type                | Null | Key  | Default | Extra |
+----------------+---------------------+------+------+---------+-------+
| TABLE_SCHEMA   | varchar(64)         | YES  |      | NULL    |       |
| TABLE_NAME     | varchar(64)         | YES  |      | NULL    |       |
| PARTITION_NAME | varchar(64)         | YES  |      | NULL    |       |
| JOB_INFO       | varchar(64)         | YES  |      | NULL    |       |
| PROCESSED_ROWS | bigint(20) unsigned | YES  |      | NULL    |       |
| START_TIME     | datetime            | YES  |      | NULL    |       |
| STATE          | varchar(64)         | YES  |      | NULL    |       |
+----------------+---------------------+------+------+---------+-------+
7 rows in set (0.00 sec)
```

```
SELECT * FROM `ANALYZE_STATUS`;
```

```
+--
    ↪ ------------+------------+----------------+------------------+---------------+---------------------+---
    ↪
| TABLE_SCHEMA | TABLE_NAME | PARTITION_NAME | JOB_INFO          | PROCESSED_ROWS | START_TIME
    ↪           | STATE    |
+--
    ↪ ------------+------------+----------------+------------------+---------------+---------------------+---
    ↪
| test         | t          |                | analyze index idx | 2              | 2019-06-21
    ↪ 19:51:14 | finished |
| test         | t          |                | analyze columns   | 2              | 2019-06-21
    ↪ 19:51:14 | finished |
| test         | t1         | p0             | analyze columns   | 0              | 2019-06-21
    ↪ 19:51:15 | finished |
| test         | t1         | p3             | analyze columns   | 0              | 2019-06-21
    ↪ 19:51:15 | finished |
| test         | t1         | p1             | analyze columns   | 0              | 2019-06-21
    ↪ 19:51:15 | finished |
| test         | t1         | p2             | analyze columns   | 1              | 2019-06-21
    ↪ 19:51:15 | finished |
+--
    ↪ ------------+------------+----------------+------------------+---------------+---------------------+---
    ↪
6 rows in set
```

ANALYZE_STATUS 表中列的含义如下：

- TABLE_SCHEMA：表所属的数据库的名称。
- TABLE_NAME：表的名称。
- PARTITION_NAME：分区表的名称。
- JOB_INFO：ANALYZE 任务的信息。
- PROCESSED_ROWS：已经处理的行数。
- START_TIME：ANALYZE 任务的开始时间。
- STATE：ANALYZE 任务的执行状态。其值可以是 pending、 running、 finished 或者 failed。

12.11.13.2.3 CHARACTER_SETS

CHARACTER_SETS 表提供字符集相关的信息。TiDB 目前仅支持部分字符集。

```
USE information_schema;
DESC character_sets;
```

```
+----------------------+-------------+------+------+---------+-------+
| Field                | Type        | Null | Key  | Default | Extra |
+----------------------+-------------+------+------+---------+-------+
```

```
| CHARACTER_SET_NAME   | varchar(32) | YES  |      | NULL    |       |
| DEFAULT_COLLATE_NAME | varchar(32) | YES  |      | NULL    |       |
| DESCRIPTION          | varchar(60) | YES  |      | NULL    |       |
| MAXLEN               | bigint(3)   | YES  |      | NULL    |       |
+----------------------+-------------+------+------+---------+-------+
4 rows in set (0.00 sec)
```

```
SELECT * FROM `character_sets`;
```

```
+--------------------+----------------------+---------------+--------+
| CHARACTER_SET_NAME | DEFAULT_COLLATE_NAME | DESCRIPTION   | MAXLEN |
+--------------------+----------------------+---------------+--------+
| utf8               | utf8_bin             | UTF-8 Unicode |      3 |
| utf8mb4            | utf8mb4_bin          | UTF-8 Unicode |      4 |
| ascii              | ascii_bin            | US ASCII      |      1 |
| latin1             | latin1_bin           | Latin1        |      1 |
| binary             | binary               | binary        |      1 |
+--------------------+----------------------+---------------+--------+
5 rows in set (0.00 sec)
```

CHARACTER_SETS 表中列的含义如下：

- CHARACTER_SET_NAME：字符集名称
- DEFAULT_COLLATE_NAME：字符集的默认排序规则名称
- DESCRIPTION：字符集的描述信息
- MAXLEN：该字符集存储一个字符所需要的最大字节数

12.11.13.2.4 CLUSTER_CONFIG

CLUSTER_CONFIG 表用于获取集群当前所有组件实例的配置。在 TiDB 早期的版本，用户需要逐个访问各个实例的 HTTP API 才能收集到所有组件配置。TiDB v4.0 后，该表的引入提高了易用性。

```
USE information_schema;
DESC cluster_config;
```

```
+----------+--------------+------+------+---------+-------+
| Field    | Type         | Null | Key  | Default | Extra |
+----------+--------------+------+------+---------+-------+
| TYPE     | varchar(64)  | YES  |      | NULL    |       |
| INSTANCE | varchar(64)  | YES  |      | NULL    |       |
| KEY      | varchar(256) | YES  |      | NULL    |       |
| VALUE    | varchar(128) | YES  |      | NULL    |       |
+----------+--------------+------+------+---------+-------+
```

字段解释：

- TYPE：节点的类型，可取值为 tidb，pd 和 tikv。
- INSTANCE：节点的服务地址。
- KEY：配置项名。
- VALUE：配置项值。

以下示例查询 TiKV 实例的 coprocessor 相关配置：

```
SELECT * FROM cluster_config WHERE type='tikv' AND `key` LIKE 'coprocessor%';
```

```
+------+-----------------+-----------------------------------+---------+
| TYPE | INSTANCE        | KEY                               | VALUE   |
+------+-----------------+-----------------------------------+---------+
| tikv | 127.0.0.1:20165 | coprocessor.batch-split-limit     | 10      |
| tikv | 127.0.0.1:20165 | coprocessor.region-max-keys       | 1440000 |
| tikv | 127.0.0.1:20165 | coprocessor.region-max-size       | 144MiB  |
| tikv | 127.0.0.1:20165 | coprocessor.region-split-keys     | 960000  |
| tikv | 127.0.0.1:20165 | coprocessor.region-split-size     | 96MiB   |
| tikv | 127.0.0.1:20165 | coprocessor.split-region-on-table | false   |
+------+-----------------+-----------------------------------+---------+
6 rows in set (0.00 sec)
```

12.11.13.2.5 CLUSTER_HARDWARE

集群硬件表 CLUSTER_HARDWARE 提供了集群各实例所在服务器的硬件信息。

```
USE information_schema;
DESC cluster_hardware;
```

```
+-------------+--------------+------+------+---------+-------+
| Field       | Type         | Null | Key  | Default | Extra |
+-------------+--------------+------+------+---------+-------+
| TYPE        | varchar(64)  | YES  |      | NULL    |       |
| INSTANCE    | varchar(64)  | YES  |      | NULL    |       |
| DEVICE_TYPE | varchar(64)  | YES  |      | NULL    |       |
| DEVICE_NAME | varchar(64)  | YES  |      | NULL    |       |
| NAME        | varchar(256) | YES  |      | NULL    |       |
| VALUE       | varchar(128) | YES  |      | NULL    |       |
+-------------+--------------+------+------+---------+-------+
6 rows in set (0.00 sec)
```

字段解释：

- TYPE：对应集群信息表information_schema.cluster_info 中的 TYPE 字段，可取值为 tidb，pd 和 tikv。
- INSTANCE：对应于集群信息表information_schema.cluster_info 中的 INSTANCE 字段。
- DEVICE_TYPE：硬件类型。目前可以查询的硬件类型有 cpu、memory、disk 和 net。
- DEVICE_NAME：硬件名。对于不同的 DEVICE_TYPE，DEVICE_NAME 的取值不同。

  - cpu：硬件名为 cpu。
  - memory：硬件名为 memory。
  - disk：磁盘名。
  - net：网卡名。
- NAME：硬件不同的信息名，比如 cpu 有 cpu-logical-cores 和 cpu-physical-cores 两个信息名，表示逻辑核心数量和物理核心数量。
- VALUE：对应硬件信息的值。例如磁盘容量和 CPU 核数。

查询集群 CPU 信息的示例如下：

```
SELECT * FROM cluster_hardware WHERE device_type='cpu' AND device_name='cpu' AND name LIKE '%
    ↪ cores';
```

```
+------+-----------------+-------------+-------------+--------------------+-------+
| TYPE | INSTANCE        | DEVICE_TYPE | DEVICE_NAME | NAME               | VALUE |
+------+-----------------+-------------+-------------+--------------------+-------+
| tidb | 0.0.0.0:4000    | cpu         | cpu         | cpu-logical-cores  | 16    |
| tidb | 0.0.0.0:4000    | cpu         | cpu         | cpu-physical-cores | 8     |
| pd   | 127.0.0.1:2379  | cpu         | cpu         | cpu-logical-cores  | 16    |
| pd   | 127.0.0.1:2379  | cpu         | cpu         | cpu-physical-cores | 8     |
| tikv | 127.0.0.1:20165 | cpu         | cpu         | cpu-logical-cores  | 16    |
| tikv | 127.0.0.1:20165 | cpu         | cpu         | cpu-physical-cores | 8     |
+------+-----------------+-------------+-------------+--------------------+-------+
6 rows in set (0.03 sec)
```

12.11.13.2.6 CLUSTER_INFO

集群拓扑表 CLUSTER_INFO 提供集群当前的拓扑信息，以及各个节点的版本信息、版本对应的 Git Hash、各节点的启动时间、各实例的运行时间。

```
USE information_schema;
desc cluster_info;
```

```
+----------------+-------------+------+------+---------+-------+
| Field          | Type        | Null | Key  | Default | Extra |
+----------------+-------------+------+------+---------+-------+
| TYPE           | varchar(64) | YES  |      | NULL    |       |
| INSTANCE       | varchar(64) | YES  |      | NULL    |       |
| STATUS_ADDRESS | varchar(64) | YES  |      | NULL    |       |
| VERSION        | varchar(64) | YES  |      | NULL    |       |
| GIT_HASH       | varchar(64) | YES  |      | NULL    |       |
| START_TIME     | varchar(32) | YES  |      | NULL    |       |
| UPTIME         | varchar(32) | YES  |      | NULL    |       |
+----------------+-------------+------+------+---------+-------+
7 rows in set (0.00 sec)
```

字段解释：

- TYPE：节点类型，目前节点的可取值为 tidb，pd 和 tikv。
- INSTANCE：实例地址，为 IP:PORT 格式的字符串。
- STATUS_ADDRESS：HTTP API 的服务地址。部分 tikv-ctl、pd-ctl 或 tidb-ctl 命令会使用到 HTTP API 和该地址。用户也可以通过该地址获取一些额外的集群信息，详情可参考 HTTP API 文档。
- VERSION：对应节点的语义版本号。TiDB 版本为了兼容 MySQL 的版本号，以 ${mysql-version}-${tidb-version} 的格式展示版本号。
- GIT_HASH：编译节点版本时的 Git Commit Hash，用于识别两个节点是否是绝对一致的版本。
- START_TIME：对应节点的启动时间。
- UPTIME：对应节点已经运行的时间。

```
SELECT * FROM cluster_info;
```

```
+--
    ↪ ----+-----------------+-----------------+-------------+----------------------------------------+------
    ↪
| TYPE | INSTANCE        | STATUS_ADDRESS   | VERSION      | GIT_HASH
    ↪                                  | START_TIME                | UPTIME              |
+--
    ↪ ----+-----------------+-----------------+-------------+----------------------------------------+------
    ↪
| tidb | 0.0.0.0:4000    | 0.0.0.0:10080    | 4.0.0-beta.2 | 0
    ↪ df3b74f55f8f8fbde39bbd5d471783f49dc10f7 | 2020-07-05T09:25:53-06:00 | 26h39m4.352862693s
    ↪  |
| pd   | 127.0.0.1:2379  | 127.0.0.1:2379   | 4.1.0-alpha  | 1
    ↪ ad59bcbf36d87082c79a1fffa3b0895234ac862 | 2020-07-05T09:25:47-06:00 | 26h39m10.352868103s
    ↪  |
| tikv | 127.0.0.1:20165 | 127.0.0.1:20180 | 4.1.0-alpha  |
    ↪ b45e052df8fb5d66aa8b3a77b5c992ddbfbb79df | 2020-07-05T09:25:50-06:00 | 26h39m7.352869963s
    ↪   |
+--
    ↪ ----+-----------------+-----------------+-------------+----------------------------------------+------
    ↪
3 rows in set (0.00 sec)
```

12.11.13.2.7 CLUSTER_LOAD

集群负载表 CLUSTER_LOAD 提供集群各个实例所在服务器的的当前负载信息。

```
USE information_schema;
DESC cluster_load;
```

```
+-------------+--------------+------+------+---------+-------+
| Field       | Type         | Null | Key  | Default | Extra |
```

```
+-------------+-------------+------+------+---------+-------+
| TYPE        | varchar(64)  | YES  |      | NULL    |       |
| INSTANCE    | varchar(64)  | YES  |      | NULL    |       |
| DEVICE_TYPE | varchar(64)  | YES  |      | NULL    |       |
| DEVICE_NAME | varchar(64)  | YES  |      | NULL    |       |
| NAME        | varchar(256) | YES  |      | NULL    |       |
| VALUE       | varchar(128) | YES  |      | NULL    |       |
+-------------+-------------+------+------+---------+-------+
6 rows in set (0.00 sec)
```

字段解释：

- TYPE：对应于节点信息表information_schema.cluster_info 中的 TYPE 字段，可取值为 tidb，pd 和 tikv。
- INSTANCE：对应于节点信息表information_schema.cluster_info 中的 INSTANCE 字段。
- DEVICE_TYPE：硬件类型，目前可以查询的硬件类型有 cpu、memory、disk 和 net。
- DEVICE_NAME：硬件名。对于不同的 DEVICE_TYPE，DEVICE_NAME 取值不同。
  - cpu：硬件名为 cpu。
  - disk：磁盘名。
  - net：网卡名。
  - memory：硬件名为 memory。
- NAME：不同的负载类型。例如 cpu 有 load1/load5/load15 三个负载类型，分别表示 cpu 在 1min/5min/15min 内的平均负载。
- VALUE：硬件负载的值，例如 cpu 在 1min/5min/15min 内的平均负载。

查询集群当前的 CPU 负载信息示例如下：

```
SELECT * FROM cluster_load WHERE device_type='cpu' AND device_name='cpu';
```

```
+------+-----------------+-------------+-------------+--------+-------+
| TYPE | INSTANCE        | DEVICE_TYPE | DEVICE_NAME | NAME   | VALUE |
+------+-----------------+-------------+-------------+--------+-------+
| tidb | 0.0.0.0:4000    | cpu         | cpu         | load1  | 0.13  |
| tidb | 0.0.0.0:4000    | cpu         | cpu         | load5  | 0.25  |
| tidb | 0.0.0.0:4000    | cpu         | cpu         | load15 | 0.31  |
| pd   | 127.0.0.1:2379  | cpu         | cpu         | load1  | 0.13  |
| pd   | 127.0.0.1:2379  | cpu         | cpu         | load5  | 0.25  |
| pd   | 127.0.0.1:2379  | cpu         | cpu         | load15 | 0.31  |
| tikv | 127.0.0.1:20165 | cpu         | cpu         | load1  | 0.13  |
| tikv | 127.0.0.1:20165 | cpu         | cpu         | load5  | 0.25  |
| tikv | 127.0.0.1:20165 | cpu         | cpu         | load15 | 0.31  |
+------+-----------------+-------------+-------------+--------+-------+
9 rows in set (1.50 sec)
```

12.11.13.2.8 CLUSTER_LOG

集群日志表 CLUSTER_LOG 表用于查询集群当前所有 TiDB/PD/TiKV 节点日志。它通过将查询条件下推到各个节点，降低了日志查询对集群的影响。该表的查询性能优于 grep 命令。

TiDB 4.0 版本之前，要获取集群的日志，用户需要逐个登录各个节点汇总日志。TiDB 4.0 的集群日志表提供了一个全局且时间有序的日志搜索结果，为跟踪全链路事件提供了便利的手段。例如按照某一个 region id 搜索日志，可以查询该 Region 生命周期内的所有日志；类似地，通过慢日志的 txn id 搜索全链路日志，可以查询该事务在各个节点扫描的 key 数量以及流量等信息。

```
USE information_schema;
DESC cluster_log;
```

```
+----------+------------------+------+------+---------+-------+
| Field    | Type             | Null | Key  | Default | Extra |
+----------+------------------+------+------+---------+-------+
| TIME     | varchar(32)      | YES  |      | NULL    |       |
| TYPE     | varchar(64)      | YES  |      | NULL    |       |
| INSTANCE | varchar(64)      | YES  |      | NULL    |       |
| LEVEL    | varchar(8)       | YES  |      | NULL    |       |
| MESSAGE  | var_string(1024) | YES  |      | NULL    |       |
+----------+------------------+------+------+---------+-------+
5 rows in set (0.00 sec)
```

字段解释：

- TIME：日志打印时间。
- TYPE：节点的类型，可取值为 tidb，pd 和 tikv。
- INSTANCE：节点的服务地址。
- LEVEL：日志级别。
- MESSAGE：日志内容。

> 注意：
>
> - 日志表的所有字段都会下推到对应节点执行，所以为了降低使用集群日志表的开销，必须指定搜索关键字以及时间范围，然后尽可能地指定更多的条件。例如 `select * from cluster_log where message like '%ddl%' and time > '2020-05-18 20:40:00' and time<'2020-05-18 21:40:00' and type='tidb'`。
> - message 字段支持 like 和 regexp 正则表达式，对应的 pattern 会编译为 regexp。同时指定多个 message 条件，相当于 grep 命令的 pipeline 形式，例如：`select * from cluster_log where message like 'coprocessor%' and message regexp '.*slow.*' and time > '2020-05-18 20:40:00' and time<'2020-05-18 21:40:00'` 相当于在集群所有节点执行 `grep 'coprocessor' xxx.log | grep -E '.*slow.*'`。

查询某个 DDL 的执行过程示例如下：

```
SELECT time,instance,left(message,150) FROM cluster_log WHERE message LIKE '%ddl%job%ID.80%' AND
    ↪ type='tidb' AND time > '2020-05-18 20:40:00' AND time < '2020-05-18 21:40:00'
```

```
+--
    ↪ ----------------------+---------------+-----------------------------------------------------------------
    ↪
| time                     | instance       | left(message,150)
    ↪
    ↪ |
+--
    ↪ ----------------------+---------------+-----------------------------------------------------------------
    ↪
| 2020/05/18 21:37:54.784 | 127.0.0.1:4002 | [ddl_worker.go:261] ["[ddl] add DDL jobs"] ["batch
    ↪ count"=1] [jobs="ID:80, Type:create table, State:none, SchemaState:none, SchemaID:1,
    ↪ TableID:79, Ro |
| 2020/05/18 21:37:54.784 | 127.0.0.1:4002 | [ddl.go:477] ["[ddl] start DDL job"] [job="ID:80,
    ↪ Type:create table, State:none, SchemaState:none, SchemaID:1, TableID:79, RowCount:0,
    ↪ ArgLen:1, start |
| 2020/05/18 21:37:55.327 | 127.0.0.1:4000 | [ddl_worker.go:568] ["[ddl] run DDL job"] [worker="
    ↪ worker 1, tp general"] [job="ID:80, Type:create table, State:none, SchemaState:none,
    ↪ SchemaID:1, Ta |
| 2020/05/18 21:37:55.381 | 127.0.0.1:4000 | [ddl_worker.go:763] ["[ddl] wait latest schema
    ↪ version changed"] [worker="worker 1, tp general"] [ver=70] ["take time"=50.809848ms] [job
    ↪ ="ID:80, Type: |
| 2020/05/18 21:37:55.382 | 127.0.0.1:4000 | [ddl_worker.go:359] ["[ddl] finish DDL job"] [worker
    ↪ ="worker 1, tp general"] [job="ID:80, Type:create table, State:synced, SchemaState:public
    ↪ , SchemaI |
| 2020/05/18 21:37:55.786 | 127.0.0.1:4002 | [ddl.go:509] ["[ddl] DDL job is finished"] [jobID
    ↪ =80]
    ↪
    ↪ |
+-------------------------+---------------+-----------------------------------------------------------------
    ↪
```

上面查询结果记录了一个 DDL 执行的过程：

- 用户将 DDL JOB ID 为 80 的请求发给 127.0.0.1:4002 TiDB 节点。
- 127.0.0.1:4000 TiDB 节点处理这个 DDL 请求，说明此时 127.0.0.1:4000 节点是 DDL owner。
- DDL JOB ID 为 80 的请求处理完成。

12.11.13.2.9 CLUSTER_SYSTEMINFO

内核参数表 CLUSTER_SYSTEMINFO 用于查询集群所有实例所在服务器的内核配置信息。目前支持查询 sysctl 的信息。

```
USE information_schema;
DESC cluster_systeminfo;
```

```
+-------------+--------------+------+------+---------+-------+
| Field       | Type         | Null | Key  | Default | Extra |
+-------------+--------------+------+------+---------+-------+
| TYPE        | varchar(64)  | YES  |      | NULL    |       |
| INSTANCE    | varchar(64)  | YES  |      | NULL    |       |
| SYSTEM_TYPE | varchar(64)  | YES  |      | NULL    |       |
| SYSTEM_NAME | varchar(64)  | YES  |      | NULL    |       |
| NAME        | varchar(256) | YES  |      | NULL    |       |
| VALUE       | varchar(128) | YES  |      | NULL    |       |
+-------------+--------------+------+------+---------+-------+
6 rows in set (0.00 sec)
```

字段解释：

- TYPE：对应于节点信息表information_schema.cluster_info 中的 TYPE 字段，可取值为 tidb，pd 和 tikv。
- INSTANCE：对应于节点信息表information_schema.cluster_info 中的 INSTANCE 字段。
- SYSTEM_TYPE：系统类型，目前可以查询的系统类型有 system。
- SYSTEM_NAME：目前可以查询的 SYSTEM_NAME 为 sysctl。
- NAME：sysctl 对应的配置名。
- VALUE：sysctl 对应配置项的值。

查询集群所有服务器的内核版本示例如下：

```
SELECT * FROM cluster_systeminfo WHERE name LIKE '%kernel.osrelease%'
```

```
+--
    ↪ ----+------------------+-------------+-------------+-----------------+---------------------------+
    ↪
| TYPE | INSTANCE          | SYSTEM_TYPE | SYSTEM_NAME | NAME            | VALUE
    ↪                       |
+--
    ↪ ----+------------------+-------------+-------------+-----------------+---------------------------+
    ↪
| tidb | 172.16.5.40:4008  | system      | sysctl      | kernel.osrelease | 3.10.0-862.14.4.el7.
    ↪ x86_64 |
| pd   | 172.16.5.40:20379 | system      | sysctl      | kernel.osrelease | 3.10.0-862.14.4.el7.
    ↪ x86_64 |
| tikv | 172.16.5.40:21150 | system      | sysctl      | kernel.osrelease | 3.10.0-862.14.4.el7.
    ↪ x86_64 |
+--
    ↪ ----+------------------+-------------+-------------+-----------------+---------------------------+
    ↪
```

12.11.13.2.10 COLLATIONS

COLLATIONS 表提供了 CHARACTER_SETS 表中字符集对应的排序规则列表。目前 TiDB 包含该表仅为兼容 MySQL。

```
USE information_schema;
DESC collations;
```

```
+--------------------+-------------+------+------+---------+-------+
| Field              | Type        | Null | Key  | Default | Extra |
+--------------------+-------------+------+------+---------+-------+
| COLLATION_NAME     | varchar(32) | YES  |      | NULL    |       |
| CHARACTER_SET_NAME | varchar(32) | YES  |      | NULL    |       |
| ID                 | bigint(11)  | YES  |      | NULL    |       |
| IS_DEFAULT         | varchar(3)  | YES  |      | NULL    |       |
| IS_COMPILED        | varchar(3)  | YES  |      | NULL    |       |
| SORTLEN            | bigint(3)   | YES  |      | NULL    |       |
+--------------------+-------------+------+------+---------+-------+
6 rows in set (0.00 sec)
```

```
SELECT * FROM collations WHERE character_set_name='utf8mb4';
```

```
+----------------+--------------------+------+------------+-------------+---------+
| COLLATION_NAME | CHARACTER_SET_NAME | ID   | IS_DEFAULT | IS_COMPILED | SORTLEN |
+----------------+--------------------+------+------------+-------------+---------+
| utf8mb4_bin    | utf8mb4            |   46 | Yes        | Yes         |       1 |
+----------------+--------------------+------+------------+-------------+---------+
1 row in set (0.00 sec)
```

COLLATION 表中列的含义如下：

- COLLATION_NAME：排序规则名称
- CHARACTER_SET_NAME：排序规则所属的字符集名称
- ID：排序规则的 ID
- IS_DEFAULT：该排序规则是否是所属字符集的默认排序规则
- IS_COMPILED：字符集是否编译到服务器中
- SORTLEN：排序规则在对字符进行排序时，所分配内存的最小长度

12.11.13.2.11 COLLATION_CHARACTER_SET_APPLICABILITY

COLLATION_CHARACTER_SET_APPLICABILITY 表将排序规则映射至适用的字符集名称。和 COLLATIONS 表一样，包含此表只是为了兼容 MySQL。

```
USE information_schema;
DESC collation_character_set_applicability;
```

```
+--------------------+-------------+------+------+---------+-------+
| Field              | Type        | Null | Key  | Default | Extra |
+--------------------+-------------+------+------+---------+-------+
| COLLATION_NAME     | varchar(32) | NO   |      | NULL    |       |
| CHARACTER_SET_NAME | varchar(32) | NO   |      | NULL    |       |
+--------------------+-------------+------+------+---------+-------+
2 rows in set (0.00 sec)
```

```
SELECT * FROM collation_character_set_applicability WHERE character_set_name='utf8mb4';
```

```
+----------------+--------------------+
| COLLATION_NAME | CHARACTER_SET_NAME |
+----------------+--------------------+
| utf8mb4_bin    | utf8mb4            |
+----------------+--------------------+
1 row in set (0.00 sec)
```

COLLATION_CHARACTER_SET_APPLICABILITY 表中列的含义如下：

- COLLATION_NAME：排序规则名称
- CHARACTER_SET_NAME：排序规则所属的字符集名称

12.11.13.2.12 COLUMNS

COLUMNS 表提供了表的所有列的信息。

```
USE information_schema;
DESC columns;
```

```
+-------------------------+---------------+------+------+---------+-------+
| Field                   | Type          | Null | Key  | Default | Extra |
+-------------------------+---------------+------+------+---------+-------+
| TABLE_CATALOG           | varchar(512)  | YES  |      | NULL    |       |
| TABLE_SCHEMA            | varchar(64)   | YES  |      | NULL    |       |
| TABLE_NAME              | varchar(64)   | YES  |      | NULL    |       |
| COLUMN_NAME             | varchar(64)   | YES  |      | NULL    |       |
| ORDINAL_POSITION        | bigint(64)    | YES  |      | NULL    |       |
| COLUMN_DEFAULT          | text          | YES  |      | NULL    |       |
| IS_NULLABLE             | varchar(3)    | YES  |      | NULL    |       |
| DATA_TYPE               | varchar(64)   | YES  |      | NULL    |       |
| CHARACTER_MAXIMUM_LENGTH | bigint(21)   | YES  |      | NULL    |       |
| CHARACTER_OCTET_LENGTH  | bigint(21)    | YES  |      | NULL    |       |
| NUMERIC_PRECISION       | bigint(21)    | YES  |      | NULL    |       |
| NUMERIC_SCALE           | bigint(21)    | YES  |      | NULL    |       |
| DATETIME_PRECISION      | bigint(21)    | YES  |      | NULL    |       |
```

```
| CHARACTER_SET_NAME       | varchar(32)   | YES  |     | NULL    |       |
| COLLATION_NAME           | varchar(32)   | YES  |     | NULL    |       |
| COLUMN_TYPE              | text          | YES  |     | NULL    |       |
| COLUMN_KEY               | varchar(3)    | YES  |     | NULL    |       |
| EXTRA                    | varchar(30)   | YES  |     | NULL    |       |
| PRIVILEGES               | varchar(80)   | YES  |     | NULL    |       |
| COLUMN_COMMENT           | varchar(1024) | YES  |     | NULL    |       |
| GENERATION_EXPRESSION    | text          | NO   |     | NULL    |       |
+--------------------------+---------------+------+-----+---------+-------+
21 rows in set (0.00 sec)
```

```
CREATE TABLE test.t1 (a int);
SELECT * FROM columns WHERE table_schema='test' AND TABLE_NAME='t1'\G
```

```
*************************** 1. row ***************************
           TABLE_CATALOG: def
            TABLE_SCHEMA: test
              TABLE_NAME: t1
             COLUMN_NAME: a
        ORDINAL_POSITION: 1
          COLUMN_DEFAULT: NULL
             IS_NULLABLE: YES
               DATA_TYPE: int
CHARACTER_MAXIMUM_LENGTH: NULL
  CHARACTER_OCTET_LENGTH: NULL
       NUMERIC_PRECISION: 11
           NUMERIC_SCALE: 0
      DATETIME_PRECISION: NULL
      CHARACTER_SET_NAME: NULL
          COLLATION_NAME: NULL
             COLUMN_TYPE: int(11)
              COLUMN_KEY:
                   EXTRA:
              PRIVILEGES: select,insert,update,references
          COLUMN_COMMENT:
   GENERATION_EXPRESSION:
1 row in set (0.02 sec)
```

COLUMNS 表中列的含义如下：

- TABLE_CATALOG：包含列的表所属的目录的名称。该值始终为 def。
- TABLE_SCHEMA：包含列的表所属的数据库的名称。
- TABLE_NAME：包含列的表的名称。
- COLUMN_NAME：列的名称。
- ORDINAL_POSITION：表中列的位置。

- COLUMN_DEFAULT：列的默认值。如果列的显式默认值为 NULL，或者列定义中不包含 default 子句，则此值为 NULL。
- IS_NULLABLE：列的可空性。如果列中可以存储空值，则该值为 YES，否则为 NO。
- DATA_TYPE：列的数据类型。
- CHARACTER_MAXIMUM_LENGTH：对于字符串列，以字符为单位的最大长度。
- CHARACTER_OCTET_LENGTH：对于字符串列，以字节为单位的最大长度。
- NUMERIC_PRECISION：对于数字列，为数字精度。
- NUMERIC_SCALE：对于数字列，为数字刻度。
- DATETIME_PRECISION：对于时间列，小数秒精度。
- CHARACTER_SET_NAME：对于字符串列，字符集名称。
- COLLATION_NAME：对于字符串列，排序规则名称。
- COLUMN_TYPE：列类型。
- COLUMN_KEY：该列是否被索引。具体显示如下：
  - 如果此值为空，则该列要么未被索引，要么被索引且是多列非唯一索引中的第二列。
  - 如果此值是 PRI，则该列是主键，或者是多列主键中的一列。
  - 如果此值是 UNI，则该列是唯一索引的第一列。
  - 如果此值是 MUL，则该列是非唯一索引的第一列，在该列中允许给定值的多次出现。
- EXTRA：关于给定列的任何附加信息。
- PRIVILEGES：当前用户对该列拥有的权限。目前在 TiDB 中，此值为定值，一直为 select,insert,update ↪ ,references。
- COLUMN_COMMENT：列定义中包含的注释。
- GENERATION_EXPRESSION：对于生成的列，显示用于计算列值的表达式。对于未生成的列为空。

对应的 SHOW 语句如下：

```
SHOW COLUMNS FROM t1 FROM test;
```

```
+-------+---------+------+------+---------+-------+
| Field | Type    | Null | Key  | Default | Extra |
+-------+---------+------+------+---------+-------+
| a     | int(11) | YES  |      | NULL    |       |
+-------+---------+------+------+---------+-------+
1 row in set (0.00 sec)
```

12.11.13.2.13 DDL_JOBS

DDL_JOBS 表为 ADMIN SHOW DDL JOBS 命令提供了一个 INFORMATION_SCHEMA 接口。它提供了 TiDB 集群中 DDL 操作的当前状态和简短历史记录。

```
USE information_schema;
DESC ddl_jobs;
```

```
+--------------+-------------+------+------+---------+-------+
| Field        | Type        | Null | Key  | Default | Extra |
+--------------+-------------+------+------+---------+-------+
```

```
| JOB_ID       | bigint(21)  | YES  |      | NULL    |       |
| DB_NAME      | varchar(64) | YES  |      | NULL    |       |
| TABLE_NAME   | varchar(64) | YES  |      | NULL    |       |
| JOB_TYPE     | varchar(64) | YES  |      | NULL    |       |
| SCHEMA_STATE | varchar(64) | YES  |      | NULL    |       |
| SCHEMA_ID    | bigint(21)  | YES  |      | NULL    |       |
| TABLE_ID     | bigint(21)  | YES  |      | NULL    |       |
| ROW_COUNT    | bigint(21)  | YES  |      | NULL    |       |
| START_TIME   | datetime    | YES  |      | NULL    |       |
| END_TIME     | datetime    | YES  |      | NULL    |       |
| STATE        | varchar(64) | YES  |      | NULL    |       |
| QUERY        | varchar(64) | YES  |      | NULL    |       |
+--------------+-------------+------+------+---------+-------+
12 rows in set (0.00 sec)
```

```
SELECT * FROM ddl_jobs LIMIT 3\G
```

```
*************************** 1. row ***************************
      JOB_ID: 44
     DB_NAME: mysql
  TABLE_NAME: opt_rule_blacklist
    JOB_TYPE: create table
SCHEMA_STATE: public
   SCHEMA_ID: 3
    TABLE_ID: 43
   ROW_COUNT: 0
  START_TIME: 2020-07-06 15:24:27
    END_TIME: 2020-07-06 15:24:27
       STATE: synced
       QUERY: CREATE TABLE IF NOT EXISTS mysql.opt_rule_blacklist (
        name char(100) NOT NULL
    );
*************************** 2. row ***************************
      JOB_ID: 42
     DB_NAME: mysql
  TABLE_NAME: expr_pushdown_blacklist
    JOB_TYPE: create table
SCHEMA_STATE: public
   SCHEMA_ID: 3
    TABLE_ID: 41
   ROW_COUNT: 0
  START_TIME: 2020-07-06 15:24:27
    END_TIME: 2020-07-06 15:24:27
       STATE: synced
       QUERY: CREATE TABLE IF NOT EXISTS mysql.expr_pushdown_blacklist (
```

```
        name char(100) NOT NULL,
        store_type char(100) NOT NULL DEFAULT 'tikv,tiflash,tidb',
        reason varchar(200)
    );
*************************** 3. row ***************************
      JOB_ID: 40
     DB_NAME: mysql
  TABLE_NAME: stats_top_n
    JOB_TYPE: create table
SCHEMA_STATE: public
   SCHEMA_ID: 3
    TABLE_ID: 39
   ROW_COUNT: 0
  START_TIME: 2020-07-06 15:24:26
    END_TIME: 2020-07-06 15:24:27
       STATE: synced
       QUERY: CREATE TABLE if not exists mysql.stats_top_n (
        table_id bigint(64) NOT NULL,
        is_index tinyint(2) NOT NULL,
        hist_id bigint(64) NOT NULL,
        value longblob,
        count bigint(64) UNSIGNED NOT NULL,
        index tbl(table_id, is_index, hist_id)
    );
3 rows in set (0.01 sec)
```

12.11.13.2.14 ENGINES

ENGINES 表提供了关于存储引擎的信息。从和 MySQL 兼容性上考虑，TiDB 会一直将 InnoDB 描述为唯一支持的引擎。此外，ENGINES 表中其它列值也都是定值。

```
USE information_schema;
DESC engines;
```

```
+--------------+-------------+------+------+---------+-------+
| Field        | Type        | Null | Key  | Default | Extra |
+--------------+-------------+------+------+---------+-------+
| ENGINE       | varchar(64) | YES  |      | NULL    |       |
| SUPPORT      | varchar(8)  | YES  |      | NULL    |       |
| COMMENT      | varchar(80) | YES  |      | NULL    |       |
| TRANSACTIONS | varchar(3)  | YES  |      | NULL    |       |
| XA           | varchar(3)  | YES  |      | NULL    |       |
| SAVEPOINTS   | varchar(3)  | YES  |      | NULL    |       |
+--------------+-------------+------+------+---------+-------+
6 rows in set (0.00 sec)
```

```
SELECT * FROM engines;
```

```
+--
    ↪ ------+---------+----------------------------------------------------------------+--------------+------+-----
    ↪
| ENGINE | SUPPORT | COMMENT                                                        | TRANSACTIONS |
    ↪ XA   | SAVEPOINTS |
+--
    ↪ ------+---------+----------------------------------------------------------------+--------------+------+-----
    ↪
| InnoDB | DEFAULT | Supports transactions, row-level locking, and foreign keys | YES          |
    ↪ YES  | YES        |
+--
    ↪ ------+---------+----------------------------------------------------------------+--------------+------+-----
    ↪
1 row in set (0.01 sec)
```

ENGINES 表中列的含义如下：

- ENGINE：存储引擎的名称。
- SUPPORT：服务器对存储引擎的支持级别，在 TiDB 中此值一直是 DEFAULT。
- COMMENT：存储引擎的简要描述。
- TRANSACTIONS：存储引擎是否支持事务。
- XA：存储引擎是否支持 XA 事务。
- SAVEPOINTS：存储引擎是否支持 savepoints。

12.11.13.2.15 INSPECTION_RESULT

TiDB 内置了一些诊断规则，用于检测系统中的故障以及隐患。

该诊断功能可以帮助用户快速发现问题，减少用户的重复性手动工作。可使用 `select * from information_schema.inspection_result` 语句来触发内部诊断。

诊断结果表 `information_schema.inspection_result` 的表结构如下：

```
USE information_schema;
DESC inspection_result;
```

```
+----------------+-------------+------+------+---------+-------+
| Field          | Type        | Null | Key  | Default | Extra |
+----------------+-------------+------+------+---------+-------+
| RULE           | varchar(64) | YES  |      | NULL    |       |
| ITEM           | varchar(64) | YES  |      | NULL    |       |
| TYPE           | varchar(64) | YES  |      | NULL    |       |
| INSTANCE       | varchar(64) | YES  |      | NULL    |       |
| STATUS_ADDRESS | varchar(64) | YES  |      | NULL    |       |
```

```
| VALUE          | varchar(64)  | YES  |      | NULL    |       |
| REFERENCE      | varchar(64)  | YES  |      | NULL    |       |
| SEVERITY       | varchar(64)  | YES  |      | NULL    |       |
| DETAILS        | varchar(256) | YES  |      | NULL    |       |
+----------------+--------------+------+------+---------+-------+
9 rows in set (0.00 sec)
```

字段解释：

- RULE：诊断规则名称，目前实现了以下规则：
  - config：配置一致性以及合理性检测。如果同一个配置在不同实例不一致，会生成 warning 诊断结果。
  - version：版本一致性检测。如果同一类型的实例版本不同，会生成 critical 诊断结果。
  - node-load：服务器负载检测。如果当前系统负载太高，会生成对应的 warning 诊断结果。
  - critical-error：系统各个模块定义了严重的错误，如果某一个严重错误在对应时间段内超过阈值，会生成 warning 诊断结果。
  - threshold-check：诊断系统会对一些关键指标进行阈值判断，如果超过阈值会生成对应的诊断信息。
- ITEM：每一个规则会对不同的项进行诊断，该字段表示对应规则下面的具体诊断项。
- TYPE：诊断的实例类型，可取值为 tidb，pd 和 tikv。
- INSTANCE：诊断的具体实例地址。
- STATUS_ADDRESS：实例的 HTTP API 服务地址。
- VALUE：针对这个诊断项得到的值。
- REFERENCE：针对这个诊断项的参考值（阈值）。如果 VALUE 超过阈值，就会产生对应的诊断信息。
- SEVERITY：严重程度，取值为 warning 或 critical。
- DETAILS：诊断的详细信息，可能包含进一步调查的 SQL 或文档链接。

诊断示例

对当前时间的集群进行诊断。

```
SELECT * FROM information_schema.inspection_result\G
```

```
***************************[ 1. row ]***************************
RULE       | config
ITEM       | log.slow-threshold
TYPE       | tidb
INSTANCE   | 172.16.5.40:4000
VALUE      | 0
REFERENCE  | not 0
SEVERITY   | warning
DETAILS    | slow-threshold = 0 will record every query to slow log, it may affect performance
***************************[ 2. row ]***************************
RULE       | version
ITEM       | git_hash
```

```
TYPE       | tidb
INSTANCE   |
VALUE      | inconsistent
REFERENCE  | consistent
SEVERITY   | critical
DETAILS    | the cluster has 2 different tidb version, execute the sql to see more detail: select
    ↪ * from information_schema.cluster_info where type='tidb'
***************************[ 3. row ]***************************
RULE       | threshold-check
ITEM       | storage-write-duration
TYPE       | tikv
INSTANCE   | 172.16.5.40:23151
VALUE      | 130.417
REFERENCE  | < 0.100
SEVERITY   | warning
DETAILS    | max duration of 172.16.5.40:23151 tikv storage-write-duration was too slow
***************************[ 4. row ]***************************
RULE       | threshold-check
ITEM       | rocksdb-write-duration
TYPE       | tikv
INSTANCE   | 172.16.5.40:20151
VALUE      | 108.105
REFERENCE  | < 0.100
SEVERITY   | warning
DETAILS    | max duration of 172.16.5.40:20151 tikv rocksdb-write-duration was too slow
```

上述诊断结果发现了以下几个问题：

- 第一行表示 TiDB 的 `log.slow-threshold` 配置值为 0，可能会影响性能。
- 第二行表示集群中有 2 个不同的 TiDB 版本
- 第三、四行表示 TiKV 的写入延迟太大，期望时间是不超过 0.1s, 但实际值远超预期。

诊断集群在时间段“2020-03-26 00:03:00”，“2020-03-26 00:08:00”的问题。指定时间范围需要使用 `/*+ time_range ()*/` 的 SQL Hint，参考下面的查询示例：

```
select /*+ time_range("2020-03-26 00:03:00", "2020-03-26 00:08:00") */ * from information_schema.
    ↪ inspection_result\G
```

```
***************************[ 1. row ]***************************
RULE       | critical-error
ITEM       | server-down
TYPE       | tidb
INSTANCE   | 172.16.5.40:4009
VALUE      |
REFERENCE  |
SEVERITY   | critical
```

```
DETAILS   | tidb 172.16.5.40:4009 restarted at time '2020/03/26 00:05:45.670'
***************************[ 2. row ]***************************
RULE      | threshold-check
ITEM      | get-token-duration
TYPE      | tidb
INSTANCE  | 172.16.5.40:10089
VALUE     | 0.234
REFERENCE | < 0.001
SEVERITY  | warning
DETAILS   | max duration of 172.16.5.40:10089 tidb get-token-duration is too slow
```

上面的诊断结果发现了以下问题：

- 第一行表示 172.16.5.40:4009 TiDB 实例在 2020/03/26 00:05:45.670 发生了重启。
- 第二行表示 172.16.5.40:10089 TiDB 实例的最大的 get-token-duration 时间为 0.234s, 期望时间是小于 0.001s。

也可以指定条件，比如只查询 critical 严重级别的诊断结果：

```
select * from information_schema.inspection_result where severity='critical';
```

只查询 critical-error 规则的诊断结果:

```
select * from information_schema.inspection_result where rule='critical-error';
```

诊断规则介绍

诊断模块内部包含一系列的规则，这些规则会通过查询已有的监控表和集群信息表，对结果和阈值进行对比。如果结果超过阈值将生成 warning 或 critical 的结果，并在 details 列中提供相应信息。

可以通过查询 inspection_rules 系统表查询已有的诊断规则:

```
select * from information_schema.inspection_rules where type='inspection';
```

```
+-----------------+------------+---------+
| NAME            | TYPE       | COMMENT |
+-----------------+------------+---------+
| config          | inspection |         |
| version         | inspection |         |
| node-load       | inspection |         |
| critical-error  | inspection |         |
| threshold-check | inspection |         |
+-----------------+------------+---------+
```

config 诊断规则

config 诊断规则通过查询 CLUSTER_CONFIG 系统表，执行以下两个诊断规则：

- 检测相同组件的配置值是否一致，并非所有配置项都会有一致性检查，下面是一致性检查的白名单：

```
// TiDB 配置一致性检查白名单
port
status.status-port
host
path
advertise-address
status.status-port
log.file.filename
log.slow-query-file
tmp-storage-path

// PD 配置一致性检查白名单
advertise-client-urls
advertise-peer-urls
client-urls
data-dir
log-file
log.file.filename
metric.job
name
peer-urls

// TiKV 配置一致性检查白名单
server.addr
server.advertise-addr
server.status-addr
log-file
raftstore.raftdb-path
storage.data-dir
storage.block-cache.capacity
```

- 检测以下配置项的值是否符合预期。

组件	配置项	预期值
TiDB	log.slow-threshold	大于 0
TiKV	raftstore.sync-log	true

version 诊断规则

version 诊断规则通过查询 CLUSTER_INFO 系统表，检测相同组件的版本 hash 是否一致。示例如下：

```
SELECT * FROM information_schema.inspection_result WHERE rule='version'\G
```

```
***************************[ 1. row ]***************************
RULE      | version
```

```
ITEM      | git_hash
TYPE      | tidb
INSTANCE  |
VALUE     | inconsistent
REFERENCE | consistent
SEVERITY  | critical
DETAILS   | the cluster has 2 different tidb versions, execute the sql to see more detail: SELECT
    ↪  * FROM information_schema.cluster_info WHERE type='tidb'
```

`critical-error` 诊断规则

`critical-error` 诊断规则执行以下两个诊断规则：

- 通过查询metrics schema 数据库中相关的监控系统表，检测集群是否有出现以下比较严重的错误：

组件	错误名字	相关监控表	错误说明
TiDB	panic-count	tidb_panic_total_count	TiDB 出现 panic 错误
TiDB	binlog-error	tidb_binlog_error_total_count	TiDB 写 binlog 时出现的错误
TiKV	critical-error	tikv_critical_error_total_coun	TiKV 的 critical error
TiKV	scheduler-is-busy	tikv_scheduler_is_busy_total_count	TiKV 的 scheduler 太忙，会导致 TiKV 临时不可用
TiKV	coprocessor-is-busy	tikv_coprocessor_is_busy_total_count	TiKV 的 coprocessor 太忙

组件	错误名字	相关监控表	错误说明
TiKV	channel-is-full	tikv_channel_full_total_count	TiKV 出现 channel full 的错误
TiKV	tikv_engine_write_stall	tikv_engine_write_stall	TiKV 出现写入 stall 的错误

- 通过查询 `metrics_schema.up` 监控表和 `CLUSTER_LOG` 系统表，检查是否有组件发生重启。

`threshold-check` 诊断规则

`threshold-check` 诊断规则通过查询metrics schema 数据库中相关的监控系统表，检测集群中以下指标是否超出阈值：

组件	监控指标	相关监控表	预期值	说明
TiDB	tso-duration	pd_tso_wait_duration	小于 50 ms	获取事务 TSO 时间戳的等待耗时
TiDB	get-token-duration	tidb_get_token_duration	小于 1 ms	查询获取 token 的耗时，相关的 TiDB 配置参数是 token-limit

组件	监控指标	相关监控表	预期值	说明
TiDB	load-schema-duration	tidb_load_schema_duration	小于 1 s	TiDB 更新表元信息的耗时
TiKV	scheduler-cmd-duration	tikv_scheduler_command_duration	小于 0.1 s	TiKV 执行 KV cmd 请求的耗时
TiKV	handle-snapshot-duration	tikv_handle_snapshot_duration	小于 30 s	TiKV 处理 snap-shot 的耗时
TiKV	storage-write-duration	tikv_storage_async_request_duration	小于 0.1 s	TiKV 写入的延迟
TiKV	storage-snapshot-duration	tikv_storage_async_request_duration	小于 50 ms	TiKV 获取 snap-shot 的耗时
TiKV	rocksdb-write-duration	tikv_engine_write_duration	小于 100 ms	TiKV RocksDB 的写入延迟
TiKV	rocksdb-get-duration	tikv_engine_max_get_duration	小于 50 ms	TiKV RocksDB 的读取延迟
TiKV	rocksdb-seek-duration	tikv_engine_max_seek_duration	小于 50 ms	TiKV RocksDB 执行 seek 的延迟

组件	监控指标	相关监控表	预期值	说明
TiKV	scheduler-pending-cmd-coun	tikv_scheduler_pending_commands	小于 1000	TiKV 中被阻塞的命令数量
TiKV	index-block-cache-hit	tikv_block_index_cache_hit	大于 0.95	TiKV 中 index block 缓存的命中率
TiKV	filter-block-cache-hit	tikv_block_filter_cache_hit	大于 0.95	TiKV 中 filter block 缓存的命中率
TiKV	data-block-cache-hit	tikv_block_data_cache_hit	大于 0.80	TiKV 中 data block 缓存的命中率
TiKV	leader-score-balance	pd_scheduler_store_status	小于 0.05	检测各个 TiKV 实例的 leader score 是否均衡，期望实例间的差异小于 5%

组件	监控指标	相关监控表	预期值	说明
TiKV	region-score-balance	pd_scheduler_store_status	小于 0.05	检测各个 TiKV 实例的 Region score 是否均衡，期望实例间的差异小于 5%
TiKV	store-available-balance	pd_scheduler_store_status	小于 0.2	检测各个 TiKV 实例的存储可用空间大小是否均衡，期望实例间的差异小于 20%

组件	监控指标	相关监控表	预期值	说明
TiKV	region-count	pd_scheduler_store_status	小于 20000	检测各个 TiKV 实例的 Region 数量，期望单个实例的 Region 数量小于 20000
PD	region-health	pd_region_health	小于 100	检测集群中处于调度中间状态的 Region 数量，期望总数小于 100

另外还会检测 TiKV 实例的以下 thread cpu usage 是否过高:

- scheduler-worker-cpu
- coprocessor-normal-cpu
- coprocessor-high-cpu
- coprocessor-low-cpu
- grpc-cpu
- raftstore-cpu
- apply-cpu
- storage-readpool-normal-cpu
- storage-readpool-high-cpu
- storage-readpool-low-cpu
- split-check-cpu

TiDB 内置的诊断规则还在不断的完善改进中，如果你也想到了一些诊断规则，非常欢迎在 tidb repository 下提 PR 或 Issue。

12.11.13.2.16 INSPECTION_RULES

INSPECTION_RULES 表提供在检查结果中运行哪些诊断测试的信息，示例用法参见inspection-result 表。

```
USE information_schema;
DESC inspection_rules;
```

```
+---------+--------------+------+------+---------+-------+
| Field   | Type         | Null | Key  | Default | Extra |
+---------+--------------+------+------+---------+-------+
| NAME    | varchar(64)  | YES  |      | NULL    |       |
| TYPE    | varchar(64)  | YES  |      | NULL    |       |
| COMMENT | varchar(256) | YES  |      | NULL    |       |
+---------+--------------+------+------+---------+-------+
3 rows in set (0.00 sec)
```

```
SELECT * FROM inspection_rules;
```

```
+-----------------+------------+---------+
| NAME            | TYPE       | COMMENT |
+-----------------+------------+---------+
| config          | inspection |         |
| version         | inspection |         |
| node-load       | inspection |         |
| critical-error  | inspection |         |
| threshold-check | inspection |         |
| ddl             | summary    |         |
| gc              | summary    |         |
| pd              | summary    |         |
| query-summary   | summary    |         |
| raftstore       | summary    |         |
| read-link       | summary    |         |
| rocksdb         | summary    |         |
| stats           | summary    |         |
| wait-events     | summary    |         |
| write-link      | summary    |         |
+-----------------+------------+---------+
15 rows in set (0.00 sec)
```

12.11.13.2.17 INSPECTION_SUMMARY

在部分场景下，用户只需要关注特定链路或模块的监控汇总。例如当前 Coprocessor 配置的线程池为 8，如果 Coprocessor 的 CPU 使用率达到了 750%，就可以确定存在风险，或者可能提前成为瓶颈。但是部分监控会因为

用户的 workload 不同而差异较大，所以难以定义确定的阈值。排查这部分场景的问题也非常重要，所以 TiDB 提供了 inspection_summary 来进行链路汇总。

诊断汇总表 information_schema.inspection_summary 的表结构如下：

```
USE information_schema;
DESC inspection_summary;
```

```
+--------------+--------------+------+------+---------+-------+
| Field        | Type         | Null | Key  | Default | Extra |
+--------------+--------------+------+------+---------+-------+
| RULE         | varchar(64)  | YES  |      | NULL    |       |
| INSTANCE     | varchar(64)  | YES  |      | NULL    |       |
| METRICS_NAME | varchar(64)  | YES  |      | NULL    |       |
| LABEL        | varchar(64)  | YES  |      | NULL    |       |
| QUANTILE     | double       | YES  |      | NULL    |       |
| AVG_VALUE    | double(22,6) | YES  |      | NULL    |       |
| MIN_VALUE    | double(22,6) | YES  |      | NULL    |       |
| MAX_VALUE    | double(22,6) | YES  |      | NULL    |       |
| COMMENT      | varchar(256) | YES  |      | NULL    |       |
+--------------+--------------+------+------+---------+-------+
9 rows in set (0.00 sec)
```

字段解释：

- RULE：汇总规则。由于规则在持续添加，最新的规则列表可以通过 select * from inspection_rules where type='summary' 查询。
- INSTANCE：监控的具体实例。
- METRICS_NAME：监控表的名字。
- QUANTILE：对于包含 QUANTILE 的监控表有效，可以通过谓词下推指定多个百分位，例如 select * from inspection_summary where rule='ddl' and quantile in (0.80, 0.90, 0.99, 0.999) 来汇总 DDL 相关监控，查询百分位为 80/90/99/999 的结果。AVG_VALUE、MIN_VALUE、MAX_VALUE 分别表示聚合的平均值、最小值、最大值。
- COMMENT：对应监控的解释。

> 注意：
>
> 由于汇总所有结果有一定开销，建议在 SQL 的谓词中显示指定的 rule 以减小开销。例如 select * from inspection_summary where rule in ('read-link', 'ddl') 会汇总读链路和 DDL 相关的监控。

使用示例:

诊断结果表和诊断监控汇总表都可以通过 hint 的方式指定诊断的时间范围，例如 select /*+ time_range ('2020-03-07 12:00:00','2020-03-07 13:00:00')*/* from inspection_summary 是对 2020-03-07 12:00:00 -

2020-03-07 13:00:00 时间段的监控汇总。和监控汇总表一样，inspection_summary 系统表也可以通过对比两个不同时间段的数据，快速发现差异较大的监控项。

以下为一个例子，对比以下两个时间段，读系统链路的监控项:

- (2020-01-16 16:00:54.933, 2020-01-16 16:10:54.933)
- (2020-01-16 16:10:54.933, 2020-01-16 16:20:54.933)

```
SELECT
  t1.avg_value / t2.avg_value AS ratio,
  t1.*,
  t2.*
FROM
  (
    SELECT
      /*+ time_range("2020-01-16 16:00:54.933", "2020-01-16 16:10:54.933")*/ *
    FROM information_schema.inspection_summary WHERE rule='read-link'
  ) t1
  JOIN
  (
    SELECT
      /*+ time_range("2020-01-16 16:10:54.933", "2020-01-16 16:20:54.933")*/ *
    FROM information_schema.inspection_summary WHERE rule='read-link'
  ) t2
  ON t1.metrics_name = t2.metrics_name
  and t1.instance = t2.instance
  and t1.label = t2.label
ORDER BY
  ratio DESC;
```

12.11.13.2.18 KEY_COLUMN_USAGE

KEY_COLUMN_USAGE 表描述了列的键约束，比如主键约束。

```
USE information_schema;
DESC key_column_usage;
```

```
+------------------------------+--------------+------+------+---------+-------+
| Field                        | Type         | Null | Key  | Default | Extra |
+------------------------------+--------------+------+------+---------+-------+
| CONSTRAINT_CATALOG           | varchar(512) | NO   |      | NULL    |       |
| CONSTRAINT_SCHEMA            | varchar(64)  | NO   |      | NULL    |       |
| CONSTRAINT_NAME              | varchar(64)  | NO   |      | NULL    |       |
| TABLE_CATALOG                | varchar(512) | NO   |      | NULL    |       |
| TABLE_SCHEMA                 | varchar(64)  | NO   |      | NULL    |       |
| TABLE_NAME                   | varchar(64)  | NO   |      | NULL    |       |
```

```
| COLUMN_NAME                    | varchar(64)  | NO   |      | NULL    |       |
| ORDINAL_POSITION               | bigint(10)   | NO   |      | NULL    |       |
| POSITION_IN_UNIQUE_CONSTRAINT | bigint(10)   | YES  |      | NULL    |       |
| REFERENCED_TABLE_SCHEMA        | varchar(64)  | YES  |      | NULL    |       |
| REFERENCED_TABLE_NAME          | varchar(64)  | YES  |      | NULL    |       |
| REFERENCED_COLUMN_NAME         | varchar(64)  | YES  |      | NULL    |       |
+--------------------------------+--------------+------+------+---------+-------+
12 rows in set (0.00 sec)
```

```
SELECT * FROM key_column_usage WHERE table_schema='mysql' and table_name='user';
```

```
*************************** 1. row ***************************
           CONSTRAINT_CATALOG: def
            CONSTRAINT_SCHEMA: mysql
              CONSTRAINT_NAME: PRIMARY
                TABLE_CATALOG: def
                 TABLE_SCHEMA: mysql
                   TABLE_NAME: user
                  COLUMN_NAME: Host
             ORDINAL_POSITION: 1
POSITION_IN_UNIQUE_CONSTRAINT: NULL
      REFERENCED_TABLE_SCHEMA: NULL
        REFERENCED_TABLE_NAME: NULL
       REFERENCED_COLUMN_NAME: NULL
*************************** 2. row ***************************
           CONSTRAINT_CATALOG: def
            CONSTRAINT_SCHEMA: mysql
              CONSTRAINT_NAME: PRIMARY
                TABLE_CATALOG: def
                 TABLE_SCHEMA: mysql
                   TABLE_NAME: user
                  COLUMN_NAME: User
             ORDINAL_POSITION: 2
POSITION_IN_UNIQUE_CONSTRAINT: NULL
      REFERENCED_TABLE_SCHEMA: NULL
        REFERENCED_TABLE_NAME: NULL
       REFERENCED_COLUMN_NAME: NULL
2 rows in set (0.00 sec)
```

KEY_COLUMN_USAGE 表中列的含义如下：

- CONSTRAINT_CATALOG：约束所属的目录的名称。该值始终为 def。
- CONSTRAINT_SCHEMA：约束所属的数据库的名称。
- CONSTRAINT_NAME：约束名称。
- TABLE_CATALOG：表所属目录的名称。该值始终为 def。

- TABLE_SCHEMA：表所属的架构数据库的名称。
- TABLE_NAME：具有约束的表的名称。
- COLUMN_NAME：具有约束的列的名称。
- ORDINAL_POSITION：列在约束中的位置，而不是列在表中的位置。列位置从 1 开始编号。
- POSITION_IN_UNIQUE_CONSTRAINT：唯一约束和主键约束为空。对于外键约束，此列是被引用的表的键的序号位置。
- REFERENCED_TABLE_SCHEMA：约束引用的数据库的名称。目前在 TiDB 中，除了外键约束，其它约束此列的值都为 nil。
- REFERENCED_TABLE_NAME：约束引用的表的名称。目前在 TiDB 中，除了外键约束，其它约束此列的值都为 nil。
- REFERENCED_COLUMN_NAME：约束引用的列的名称。目前在 TiDB 中，除了外键约束，其它约束此列的值都为 nil。

12.11.13.2.19 METRICS_SUMMARY

由于 TiDB 集群的监控指标数量较多，为了方便用户从众多监控中找出异常的监控项，TiDB 4.0 提供了以下监控汇总表：

- information_schema.metrics_summary
- information_schema.metrics_summary_by_label

这两张表用于汇总所有监控数据，用户排查各个监控指标会更有效率。其中 information_schema.metrics_summary_by_label 会对不同的 label 进行区分统计。

```
USE information_schema;
DESC metrics_summary;
```

```
+--------------+--------------+------+------+---------+-------+
| Field        | Type         | Null | Key  | Default | Extra |
+--------------+--------------+------+------+---------+-------+
| METRICS_NAME | varchar(64)  | YES  |      | NULL    |       |
| QUANTILE     | double       | YES  |      | NULL    |       |
| SUM_VALUE    | double(22,6) | YES  |      | NULL    |       |
| AVG_VALUE    | double(22,6) | YES  |      | NULL    |       |
| MIN_VALUE    | double(22,6) | YES  |      | NULL    |       |
| MAX_VALUE    | double(22,6) | YES  |      | NULL    |       |
| COMMENT      | varchar(256) | YES  |      | NULL    |       |
+--------------+--------------+------+------+---------+-------+
7 rows in set (0.00 sec)
```

字段解释：

- METRICS_NAME：监控表名。
- QUANTILE：百分位。可以通过 SQL 语句指定 QUANTILE，例如：
  - select * from metrics_summary where quantile=0.99 指定查看百分位为 0.99 的数据。

  - `select * from metrics_summary where quantile in (0.80, 0.90, 0.99, 0.999)` 同时查看百分位为 0.80, 0.90, 0.99, 0.999 的数据。

- `SUM_VALUE`、`AVG_VALUE`、`MIN_VALUE`、`MAX_VALUE` 分别表示总和、平均值、最小值、最大值。
- `COMMENT`：对应监控的解释。

具体查询示例：

查询 `'2020-03-08 13:23:00'`, `'2020-03-08 13:33:00'` 时间范围内 TiDB 集群中平均耗时最高的三组监控项。可直接查询 `information_schema.metrics_summary` 表，并通过 `/*+ time_range()*/` 这个 hint 来指定时间范围，构造的 SQL 语句如下：

```
SELECT /*+ time_range('2020-03-08 13:23:00','2020-03-08 13:33:00') */ *
FROM information_schema.metrics_summary
WHERE metrics_name LIKE 'tidb%duration'
 AND avg_value > 0
 AND quantile = 0.99
ORDER BY avg_value DESC
LIMIT 3\G
```

```
***************************[ 1. row ]***************************
METRICS_NAME | tidb_get_token_duration
QUANTILE     | 0.99
SUM_VALUE    | 8.972509
AVG_VALUE    | 0.996945
MIN_VALUE    | 0.996515
MAX_VALUE    | 0.997458
COMMENT      |  The quantile of Duration (us) for getting token, it should be small until
    ↪ concurrency limit is reached(second)
***************************[ 2. row ]***************************
METRICS_NAME | tidb_query_duration
QUANTILE     | 0.99
SUM_VALUE    | 0.269079
AVG_VALUE    | 0.007272
MIN_VALUE    | 0.000667
MAX_VALUE    | 0.01554
COMMENT      | The quantile of TiDB query durations(second)
***************************[ 3. row ]***************************
METRICS_NAME | tidb_kv_request_duration
QUANTILE     | 0.99
SUM_VALUE    | 0.170232
AVG_VALUE    | 0.004601
MIN_VALUE    | 0.000975
MAX_VALUE    | 0.013
COMMENT      | The quantile of kv requests durations by store
```

类似的，查询 `metrics_summary_by_label` 监控汇总表示例如下：

```
SELECT /*+ time_range('2020-03-08 13:23:00','2020-03-08 13:33:00') */ *
FROM information_schema.metrics_summary_by_label
WHERE metrics_name LIKE 'tidb%duration'
 AND avg_value > 0
 AND quantile = 0.99
ORDER BY avg_value DESC
LIMIT 10\G
```

```
***************************[ 1. row ]***************************
INSTANCE     | 172.16.5.40:10089
METRICS_NAME | tidb_get_token_duration
LABEL        |
QUANTILE     | 0.99
SUM_VALUE    | 8.972509
AVG_VALUE    | 0.996945
MIN_VALUE    | 0.996515
MAX_VALUE    | 0.997458
COMMENT      |  The quantile of Duration (us) for getting token, it should be small until
    ↪ concurrency limit is reached(second)
***************************[ 2. row ]***************************
INSTANCE     | 172.16.5.40:10089
METRICS_NAME | tidb_query_duration
LABEL        | Select
QUANTILE     | 0.99
SUM_VALUE    | 0.072083
AVG_VALUE    | 0.008009
MIN_VALUE    | 0.007905
MAX_VALUE    | 0.008241
COMMENT      | The quantile of TiDB query durations(second)
***************************[ 3. row ]***************************
INSTANCE     | 172.16.5.40:10089
METRICS_NAME | tidb_query_duration
LABEL        | Rollback
QUANTILE     | 0.99
SUM_VALUE    | 0.072083
AVG_VALUE    | 0.008009
MIN_VALUE    | 0.007905
MAX_VALUE    | 0.008241
COMMENT      | The quantile of TiDB query durations(second)
```

前文提到 `metrics_summary_by_label` 表结构相对于 `metrics_summary` 多了一列 `LABEL`。以上面查询结果的第 2、3 行分别表示 `tidb_query_duration` 的 `Select` 和 `Rollback` 类型的语句平均耗时非常高。

除以上示例之外，监控汇总表可以通过对比两个时间段的全链路监控，迅速找出监控数据中变化最大的模块，快速定位瓶颈。以下示例对比两个时间段的所有监控（其中 t1 为 baseline），并按照差别最大的监控排序：

- 时间段 t1：("2020-03-03 17:08:00", "2020-03-03 17:11:00")
- 时间段 t2：("2020-03-03 17:18:00", "2020-03-03 17:21:00")

对两个时间段的监控按照 METRICS_NAME 进行 join，并按照差异值大小排序。其中 TIME_RANGE 是用于指定查询时间的 hint。

```
SELECT GREATEST(t1.avg_value,t2.avg_value)/LEAST(t1.avg_value,
         t2.avg_value) AS ratio,
         t1.metrics_name,
         t1.avg_value as t1_avg_value,
         t2.avg_value as t2_avg_value,
         t2.comment
FROM
    (SELECT /*+ time_range("2020-03-03 17:08:00", "2020-03-03 17:11:00")*/ *
    FROM information_schema.metrics_summary ) t1
JOIN
    (SELECT /*+ time_range("2020-03-03 17:18:00", "2020-03-03 17:21:00")*/ *
    FROM information_schema.metrics_summary ) t2
    ON t1.metrics_name = t2.metrics_name
ORDER BY ratio DESC LIMIT 10;
```

```
+--
    ↪ --------------+----------------------------------------+----------------+-----------------+-----------
    ↪
| ratio          | metrics_name                             | t1_avg_value   | t2_avg_value     |
    ↪  comment
    ↪                                                                                          |
+--
    ↪ --------------+----------------------------------------+----------------+-----------------+-----------
    ↪
| 5865.59537065  | tidb_slow_query_cop_process_total_time   |       0.016333 |        95.804724 |
    ↪  The total time of TiDB slow query statistics with slow query total cop process time(
    ↪ second) |
| 3648.74109023  | tidb_distsql_partial_scan_key_total_num  |   10865.666667 |  39646004.4394   |
    ↪  The total num of distsql partial scan key numbers
    ↪                                          |
|  267.002351165 | tidb_slow_query_cop_wait_total_time      |       0.003333 |         0.890008 |
    ↪  The total time of TiDB slow query statistics with slow query total cop wait time(second)
    ↪     |
|  192.43267836  | tikv_cop_total_response_total_size       | 2515333.66667  | 484032394.445    |
    ↪
    ↪
    ↪ |
|  192.43267836  | tikv_cop_total_response_size_per_seconds |   41922.227778 |   8067206.57408  |
    ↪
    ↪
```

```
    ↪ |
|  152.780296296 | tidb_distsql_scan_key_total_num          |    5304.333333 |    810397.618317 |
    ↪  The total num of distsql scan numbers
    ↪                                                      |
|  126.042290167 | tidb_distsql_execution_total_time        |       0.421622 |        53.142143 |
    ↪  The total time of distsql execution(second)
    ↪                                                    |
|  105.164020657 | tikv_cop_scan_details                    |     134.450733 |     14139.379665 |
    ↪
    ↪
    ↪ |
|  105.164020657 | tikv_cop_scan_details_total              |    8067.043981 |    848362.77991  |
    ↪
    ↪
    ↪ |
|  101.635495394 | tikv_cop_scan_keys_num                   |    1070.875    |    108838.91113  |
    ↪
    ↪
    ↪ |
+--
    ↪ --------------+------------------------------------------+----------------+-----------------+-----------
    ↪
```

上面查询结果表示：

- t2 时间段内的 `tidb_slow_query_cop_process_total_time`（TiDB 慢查询中的 `cop process` 耗时）比 t1 时间段高了 5865 倍。
- t2 时间段内的 `tidb_distsql_partial_scan_key_total_num`（TiDB 的 `distsql` 请求扫描 key 的数量）比 t1 时间段高了 3648 倍。
- t2 时间段内的 `tidb_slow_query_cop_wait_total_time`（TiDB 慢查询中的 cop 请求排队等待的耗时）比 t1 时间段高了 267 倍。
- t2 时间段内的 `tikv_cop_total_response_size`（TiKV 的 cop 请求结果的大小）比 t1 时间段高了 192 倍。
- t2 时间段内的 `tikv_cop_scan_details`（TiKV 的 cop 请求的 scan）比 t1 时间段高了 105 倍。

综上，可以马上知道 t2 时间段的 cop 请求要比 t2 时间段高很多，导致 TiKV 的 Coprocessor 过载，出现了 `cop task` 等待，可以猜测可能是 t2 时间段出现了一些大查询，或者是查询较多的负载。

实际上，在 t1 ~ t2 整个时间段内都在跑 `go-ycsb` 的压测，然后在 t2 时间段跑了 20 个 `tpch` 的查询，所以是因为 `tpch` 大查询导致了出现很多的 cop 请求。

12.11.13.2.20 METRICS_TABLES

METRICS_TABLES 表为metrics_schema 数据库中的每个视图提供 PromQL（Prometheus 查询语言）定义。

```
USE information_schema;
DESC metrics_tables;
```

```
+------------+--------------+------+------+---------+-------+
| Field      | Type         | Null | Key  | Default | Extra |
+------------+--------------+------+------+---------+-------+
| TABLE_NAME | varchar(64)  | YES  |      | NULL    |       |
| PROMQL     | varchar(64)  | YES  |      | NULL    |       |
| LABELS     | varchar(64)  | YES  |      | NULL    |       |
| QUANTILE   | double       | YES  |      | NULL    |       |
| COMMENT    | varchar(256) | YES  |      | NULL    |       |
+------------+--------------+------+------+---------+-------+
```

表 metrics_tables 的字段解释：

- TABLE_NAME：对应于 metrics_schema 中的表名。
- PROMQL：监控表的主要原理是将 SQL 映射成 PromQL，并将 Prometheus 结果转换成 SQL 查询结果。这个字段是 PromQL 的表达式模板，查询监控表数据时使用查询条件改写模板中的变量，生成最终的查询表达式。
- LABELS：监控定义的 label，每一个 label 对应监控表中的一列。SQL 中如果包含对应列的过滤，对应的 PromQL 也会改变。
- QUANTILE：百分位。对于直方图类型的监控数据，指定一个默认百分位。如果值为 0，表示该监控表对应的监控不是直方图。
- COMMENT：对这个监控表的注释。

```
SELECT * FROM metrics_tables LIMIT 5\G
```

```
*************************** 1. row ***************************
TABLE_NAME: abnormal_stores
    PROMQL: sum(pd_cluster_status{ type=~"store_disconnected_count|store_unhealth_count|
        ↪ store_low_space_count|store_down_count|store_offline_count|store_tombstone_count"})
    LABELS: instance,type
  QUANTILE: 0
   COMMENT:
*************************** 2. row ***************************
TABLE_NAME: etcd_disk_wal_fsync_rate
    PROMQL: delta(etcd_disk_wal_fsync_duration_seconds_count{$LABEL_CONDITIONS}[$RANGE_DURATION])
    LABELS: instance
  QUANTILE: 0
   COMMENT: The rate of writing WAL into the persistent storage
*************************** 3. row ***************************
TABLE_NAME: etcd_wal_fsync_duration
    PROMQL: histogram_quantile($QUANTILE, sum(rate(etcd_disk_wal_fsync_duration_seconds_bucket{
        ↪ $LABEL_CONDITIONS}[$RANGE_DURATION])) by (le,instance))
    LABELS: instance
  QUANTILE: 0.99
   COMMENT: The quantile time consumed of writing WAL into the persistent storage
```

```
*************************** 4. row ***************************
TABLE_NAME: etcd_wal_fsync_total_count
    PROMQL: sum(increase(etcd_disk_wal_fsync_duration_seconds_count{$LABEL_CONDITIONS}[
        ↪ $RANGE_DURATION])) by (instance)
    LABELS: instance
  QUANTILE: 0
   COMMENT: The total count of writing WAL into the persistent storage
*************************** 5. row ***************************
TABLE_NAME: etcd_wal_fsync_total_time
    PROMQL: sum(increase(etcd_disk_wal_fsync_duration_seconds_sum{$LABEL_CONDITIONS}[
        ↪ $RANGE_DURATION])) by (instance)
    LABELS: instance
  QUANTILE: 0
   COMMENT: The total time of writing WAL into the persistent storage
5 rows in set (0.00 sec)
```

12.11.13.2.21 PARTITIONS

PARTITIONS 表提供有关分区表的信息。

```
USE information_schema;
DESC partitions;
```

```
+-------------------------------+--------------+------+------+---------+-------+
| Field                         | Type         | Null | Key  | Default | Extra |
+-------------------------------+--------------+------+------+---------+-------+
| TABLE_CATALOG                 | varchar(512) | YES  |      | NULL    |       |
| TABLE_SCHEMA                  | varchar(64)  | YES  |      | NULL    |       |
| TABLE_NAME                    | varchar(64)  | YES  |      | NULL    |       |
| PARTITION_NAME                | varchar(64)  | YES  |      | NULL    |       |
| SUBPARTITION_NAME             | varchar(64)  | YES  |      | NULL    |       |
| PARTITION_ORDINAL_POSITION    | bigint(21)   | YES  |      | NULL    |       |
| SUBPARTITION_ORDINAL_POSITION | bigint(21)   | YES  |      | NULL    |       |
| PARTITION_METHOD              | varchar(18)  | YES  |      | NULL    |       |
| SUBPARTITION_METHOD           | varchar(12)  | YES  |      | NULL    |       |
| PARTITION_EXPRESSION          | longblob     | YES  |      | NULL    |       |
| SUBPARTITION_EXPRESSION       | longblob     | YES  |      | NULL    |       |
| PARTITION_DESCRIPTION         | longblob     | YES  |      | NULL    |       |
| TABLE_ROWS                    | bigint(21)   | YES  |      | NULL    |       |
| AVG_ROW_LENGTH                | bigint(21)   | YES  |      | NULL    |       |
| DATA_LENGTH                   | bigint(21)   | YES  |      | NULL    |       |
| MAX_DATA_LENGTH               | bigint(21)   | YES  |      | NULL    |       |
| INDEX_LENGTH                  | bigint(21)   | YES  |      | NULL    |       |
| DATA_FREE                     | bigint(21)   | YES  |      | NULL    |       |
| CREATE_TIME                   | datetime     | YES  |      | NULL    |       |
```

```
| UPDATE_TIME                   | datetime     | YES  |     | NULL    |       |
| CHECK_TIME                    | datetime     | YES  |     | NULL    |       |
| CHECKSUM                      | bigint(21)   | YES  |     | NULL    |       |
| PARTITION_COMMENT             | varchar(80)  | YES  |     | NULL    |       |
| NODEGROUP                     | varchar(12)  | YES  |     | NULL    |       |
| TABLESPACE_NAME               | varchar(64)  | YES  |     | NULL    |       |
+-------------------------------+--------------+------+-----+---------+-------+
25 rows in set (0.00 sec)
```

```
CREATE TABLE test.t1 (id INT NOT NULL PRIMARY KEY) PARTITION BY HASH (id) PARTITIONS 2;
SELECT * FROM partitions WHERE table_schema='test' AND table_name='t1'\G
```

```
*************************** 1. row ***************************
  TABLE_CATALOG: def
   TABLE_SCHEMA: test
     TABLE_NAME: t1
  PARTITION_NAME: p0
SUBPARTITION_NAME: NULL
PARTITION_ORDINAL_POSITION: 1
SUBPARTITION_ORDINAL_POSITION: NULL
  PARTITION_METHOD: HASH
SUBPARTITION_METHOD: NULL
 PARTITION_EXPRESSION: `id`
SUBPARTITION_EXPRESSION: NULL
 PARTITION_DESCRIPTION:
   TABLE_ROWS: 0
AVG_ROW_LENGTH: 0
   DATA_LENGTH: 0
MAX_DATA_LENGTH: 0
   INDEX_LENGTH: 0
      DATA_FREE: 0
    CREATE_TIME: 2020-07-06 16:35:28
    UPDATE_TIME: NULL
     CHECK_TIME: NULL
      CHECKSUM: NULL
PARTITION_COMMENT:
      NODEGROUP: NULL
 TABLESPACE_NAME: NULL
*************************** 2. row ***************************
  TABLE_CATALOG: def
   TABLE_SCHEMA: test
     TABLE_NAME: t1
  PARTITION_NAME: p1
SUBPARTITION_NAME: NULL
PARTITION_ORDINAL_POSITION: 2
```

```
SUBPARTITION_ORDINAL_POSITION: NULL
              PARTITION_METHOD: HASH
           SUBPARTITION_METHOD: NULL
          PARTITION_EXPRESSION: `id`
       SUBPARTITION_EXPRESSION: NULL
         PARTITION_DESCRIPTION:
                    TABLE_ROWS: 0
                AVG_ROW_LENGTH: 0
                   DATA_LENGTH: 0
               MAX_DATA_LENGTH: 0
                  INDEX_LENGTH: 0
                     DATA_FREE: 0
                   CREATE_TIME: 2020-07-06 16:35:28
                   UPDATE_TIME: NULL
                    CHECK_TIME: NULL
                      CHECKSUM: NULL
             PARTITION_COMMENT:
                     NODEGROUP: NULL
              TABLESPACE_NAME: NULL
2 rows in set (0.00 sec)
```

12.11.13.2.22 PROCESSLIST

PROCESSLIST 和 SHOW PROCESSLIST 的功能一样，都是查看当前正在处理的请求。

PROCESSLIST 表比 SHOW PROCESSLIST 的结果多出下面几列：

- MEM 列：MEM 显示正在处理的请求已使用的内存，单位是 byte。
- TxnStart列：显示事务的开始时间

```
USE information_schema;
DESC processlist;
```

```
+----------+---------------------+------+------+---------+-------+
| Field    | Type                | Null | Key  | Default | Extra |
+----------+---------------------+------+------+---------+-------+
| ID       | bigint(21) unsigned | NO   |      | 0       |       |
| USER     | varchar(16)         | NO   |      |         |       |
| HOST     | varchar(64)         | NO   |      |         |       |
| DB       | varchar(64)         | YES  |      | NULL    |       |
| COMMAND  | varchar(16)         | NO   |      |         |       |
| TIME     | int(7)              | NO   |      | 0       |       |
| STATE    | varchar(7)          | YES  |      | NULL    |       |
| INFO     | binary(512)         | YES  |      | NULL    |       |
| MEM      | bigint(21) unsigned | YES  |      | NULL    |       |
```

```
| TxnStart | varchar(64)         | NO   |      |         |       |
+----------+---------------------+------+------+---------+-------+
10 rows in set (0.00 sec)
```

```
SELECT * FROM processlist\G
```

```
*************************** 1. row ***************************
      ID: 16
    USER: root
    HOST: 127.0.0.1
      DB: information_schema
 COMMAND: Query
    TIME: 0
   STATE: autocommit
    INFO: SELECT * FROM processlist
     MEM: 0
TxnStart:
1 row in set (0.00 sec)
```

PROCESSLIST 表各列的含义如下：

- ID：客户连接 ID。
- USER：执行当前 PROCESS 的用户名。
- HOST：客户连接的地址。
- DB：当前连接的默认数据库名。
- COMMAND：当前 PROCESS 执行的命令类型。
- TIME：当前 PROCESS 的已经执行的时间，单位是秒。
- STATE：当前连接的状态。
- INFO：正在处理的请求语句。
- MEM：正在处理的请求已使用的内存，单位是 byte。
- TxnStart列：显示事务的开始时间

CLUSTER_PROCESSLIST

CLUSTER_PROCESSLIST 是 PROCESSLIST 对应的集群系统表，用于查询集群中所有 TiDB 节点的 PROCESSLIST 信息。CLUSTER_PROCESSLIST 表结构上比 PROCESSLIST 多一列 INSTANCE，表示该行数据来自的 TiDB 节点地址。

```
SELECT * FROM cluster_processlist;
```

```
+--
    ↪ ---------------+-----+------+----------+------+---------+------+------------+-----------------------------
    ↪
| INSTANCE         | ID  | USER | HOST     | DB   | COMMAND | TIME | STATE      | INFO
    ↪                                                  | MEM | TxnStart
    ↪                                |
```

```
+--
    ↪ ---------------+-----+------+----------+------+---------+------+------------+-----------------------------
    ↪
| 10.0.1.22:10080 | 150 | u1   | 10.0.1.1 | test | Query   | 0    | autocommit | select count(*)
    ↪ from usertable                       | 372 | 05-28 03:54:21.230(416976223923077223) |
| 10.0.1.22:10080 | 138 | root | 10.0.1.1 | test | Query   | 0    | autocommit | SELECT * FROM
    ↪ information_schema.cluster_processlist | 0   | 05-28 03:54:21.230(416976223923077220) |
| 10.0.1.22:10080 | 151 | u1   | 10.0.1.1 | test | Query   | 0    | autocommit | select count(*)
    ↪ from usertable                       | 372 | 05-28 03:54:21.230(416976223923077224) |
| 10.0.1.21:10080 | 15  | u2   | 10.0.1.1 | test | Query   | 0    | autocommit | select max(
    ↪ field0) from usertable                    | 496 | 05-28 03:54:21.230(416976223923077222)
    ↪ |
| 10.0.1.21:10080 | 14  | u2   | 10.0.1.1 | test | Query   | 0    | autocommit | select max(
    ↪ field0) from usertable                    | 496 | 05-28 03:54:21.230(416976223923077225)
    ↪ |
+--
    ↪ ---------------+-----+------+----------+------+---------+------+------------+-----------------------------
    ↪
```

12.11.13.2.23 SCHEMATA

SCHEMATA 表提供了关于数据库的信息。表中的数据与 SHOW DATABASES 语句的执行结果等价。

```
USE information_schema;
desc SCHEMATA;
```

```
+----------------------------+--------------+------+------+---------+-------+
| Field                      | Type         | Null | Key  | Default | Extra |
+----------------------------+--------------+------+------+---------+-------+
| CATALOG_NAME               | varchar(512) | YES  |      | NULL    |       |
| SCHEMA_NAME                | varchar(64)  | YES  |      | NULL    |       |
| DEFAULT_CHARACTER_SET_NAME | varchar(64)  | YES  |      | NULL    |       |
| DEFAULT_COLLATION_NAME     | varchar(32)  | YES  |      | NULL    |       |
| SQL_PATH                   | varchar(512) | YES  |      | NULL    |       |
+----------------------------+--------------+------+------+---------+-------+
5 rows in set (0.00 sec)
```

```
SELECT * FROM SCHEMATA;
```

```
+--
    ↪ ------------+--------------------+----------------------------+------------------------+----------+
    ↪
| CATALOG_NAME | SCHEMA_NAME        | DEFAULT_CHARACTER_SET_NAME | DEFAULT_COLLATION_NAME |
    ↪ SQL_PATH |
```

```
+--
    ↪ ------------+--------------------+----------------------------+------------------------+----------+
    ↪
| def           | INFORMATION_SCHEMA | utf8mb4                    | utf8mb4_bin            | NULL
    ↪     |
| def           | METRICS_SCHEMA     | utf8mb4                    | utf8mb4_bin            | NULL
    ↪     |
| def           | mysql              | utf8mb4                    | utf8mb4_bin            | NULL
    ↪     |
| def           | PERFORMANCE_SCHEMA | utf8mb4                    | utf8mb4_bin            | NULL
    ↪     |
| def           | test               | utf8mb4                    | utf8mb4_bin            | NULL
    ↪     |
+--
    ↪ ------------+--------------------+----------------------------+------------------------+----------+
    ↪
5 rows in set (0.00 sec)
```

SCHEMATA 表各列字段含义如下：

- CATALOG_NAME：数据库归属的目录名，该列值永远为 def。
- SCHEMA_NAME：数据库的名字。
- DEFAULT_CHARACTER_SET_NAME：数据库的默认字符集。
- DEFAULT_COLLATION_NAME：数据库的默认 collation。
- SQL_PATH：该项值永远为 NULL。

12.11.13.2.24 SEQUENCES

SEQUENCES 表提供了有关序列的信息。TiDB 中序列的功能是参照 MariaDB 中的类似功能来实现的。

```
USE information_schema;
DESC sequences;
```

```
+-----------------+--------------+------+------+---------+-------+
| Field           | Type         | Null | Key  | Default | Extra |
+-----------------+--------------+------+------+---------+-------+
| TABLE_CATALOG   | varchar(512) | NO   |      | NULL    |       |
| SEQUENCE_SCHEMA | varchar(64)  | NO   |      | NULL    |       |
| SEQUENCE_NAME   | varchar(64)  | NO   |      | NULL    |       |
| CACHE           | tinyint(4)   | NO   |      | NULL    |       |
| CACHE_VALUE     | bigint(21)   | YES  |      | NULL    |       |
| CYCLE           | tinyint(4)   | NO   |      | NULL    |       |
| INCREMENT       | bigint(21)   | NO   |      | NULL    |       |
| MAX_VALUE       | bigint(21)   | YES  |      | NULL    |       |
| MIN_VALUE       | bigint(21)   | YES  |      | NULL    |       |
| START           | bigint(21)   | YES  |      | NULL    |       |
```

```
| COMMENT         | varchar(64)  | YES  |      | NULL    |       |
+-----------------+--------------+------+------+---------+-------+
11 rows in set (0.00 sec)
```

```
CREATE SEQUENCE test.seq;
SELECT nextval(test.seq);
SELECT * FROM sequences\G
```

```
+-------------------+
| nextval(test.seq) |
+-------------------+
|                 1 |
+-------------------+
1 row in set (0.01 sec)
*************************** 1. row ***************************
  TABLE_CATALOG: def
SEQUENCE_SCHEMA: test
  SEQUENCE_NAME: seq
          CACHE: 1
    CACHE_VALUE: 1000
          CYCLE: 0
      INCREMENT: 1
      MAX_VALUE: 9223372036854775806
      MIN_VALUE: 1
          START: 1
        COMMENT:
1 row in set (0.00 sec)
```

12.11.13.2.25 SESSION_VARIABLES

SESSION_VARIABLES 表提供了关于 session 变量的信息。表中的数据跟 SHOW SESSION VARIABLES 语句执行结果类似。

```
USE information_schema;
DESC session_variables;
```

```
+----------------+---------------+------+------+---------+-------+
| Field          | Type          | Null | Key  | Default | Extra |
+----------------+---------------+------+------+---------+-------+
| VARIABLE_NAME  | varchar(64)   | YES  |      | NULL    |       |
| VARIABLE_VALUE | varchar(1024) | YES  |      | NULL    |       |
+----------------+---------------+------+------+---------+-------+
2 rows in set (0.00 sec)
```

```
SELECT * FROM session_variables ORDER BY variable_name LIMIT 10;
```

```
+----------------------------------+------------------+
| VARIABLE_NAME                    | VARIABLE_VALUE   |
+----------------------------------+------------------+
| allow_auto_random_explicit_insert | off              |
| auto_increment_increment         | 1                |
| auto_increment_offset            | 1                |
| autocommit                       | 1                |
| automatic_sp_privileges          | 1                |
| avoid_temporal_upgrade           | 0                |
| back_log                         | 80               |
| basedir                          | /usr/local/mysql |
| big_tables                       | 0                |
| bind_address                     | *                |
+----------------------------------+------------------+
10 rows in set (0.00 sec)
```

SESSION_VARIABLES 表各列字段含义如下：

- VARIABLE_NAME：数据库中 session 级变量的名称。
- VARIABLE_VALUE：数据库中对应该 session 变量名的具体值。

12.11.13.2.26 SLOW_QUERY

SLOW_QUERY 表中提供了当前节点的慢查询相关的信息，其内容通过解析当前节点的 TiDB 慢查询日志而来，列名和慢日志中的字段名是一一对应。关于如何使用该表调查和改善慢查询，请参考慢查询日志文档。

```
USE information_schema;
DESC slow_query;
```

```
+---------------------------+---------------------+------+------+---------+-------+
| Field                     | Type                | Null | Key  | Default | Extra |
+---------------------------+---------------------+------+------+---------+-------+
| Time                      | timestamp(6)        | YES  |      | NULL    |       |
| Txn_start_ts              | bigint(20) unsigned | YES  |      | NULL    |       |
| User                      | varchar(64)         | YES  |      | NULL    |       |
| Host                      | varchar(64)         | YES  |      | NULL    |       |
| Conn_ID                   | bigint(20) unsigned | YES  |      | NULL    |       |
| Query_time                | double              | YES  |      | NULL    |       |
| Parse_time                | double              | YES  |      | NULL    |       |
| Compile_time              | double              | YES  |      | NULL    |       |
| Rewrite_time              | double              | YES  |      | NULL    |       |
| Preproc_subqueries        | bigint(20) unsigned | YES  |      | NULL    |       |
| Preproc_subqueries_time   | double              | YES  |      | NULL    |       |
| Optimize_time             | double              | YES  |      | NULL    |       |
| Wait_TS                   | double              | YES  |      | NULL    |       |
```

```
| Prewrite_time             | double              | YES  |      | NULL    |       |
| Wait_prewrite_binlog_time | double              | YES  |      | NULL    |       |
| Commit_time               | double              | YES  |      | NULL    |       |
| Get_commit_ts_time        | double              | YES  |      | NULL    |       |
| Commit_backoff_time       | double              | YES  |      | NULL    |       |
| Backoff_types             | varchar(64)         | YES  |      | NULL    |       |
| Resolve_lock_time         | double              | YES  |      | NULL    |       |
| Local_latch_wait_time     | double              | YES  |      | NULL    |       |
| Write_keys                | bigint(22)          | YES  |      | NULL    |       |
| Write_size                | bigint(22)          | YES  |      | NULL    |       |
| Prewrite_region           | bigint(22)          | YES  |      | NULL    |       |
| Txn_retry                 | bigint(22)          | YES  |      | NULL    |       |
| Cop_time                  | double              | YES  |      | NULL    |       |
| Process_time              | double              | YES  |      | NULL    |       |
| Wait_time                 | double              | YES  |      | NULL    |       |
| Backoff_time              | double              | YES  |      | NULL    |       |
| LockKeys_time             | double              | YES  |      | NULL    |       |
| Request_count             | bigint(20) unsigned | YES  |      | NULL    |       |
| Total_keys                | bigint(20) unsigned | YES  |      | NULL    |       |
| Process_keys              | bigint(20) unsigned | YES  |      | NULL    |       |
| DB                        | varchar(64)         | YES  |      | NULL    |       |
| Index_names               | varchar(100)        | YES  |      | NULL    |       |
| Is_internal               | tinyint(1)          | YES  |      | NULL    |       |
| Digest                    | varchar(64)         | YES  |      | NULL    |       |
| Stats                     | varchar(512)        | YES  |      | NULL    |       |
| Cop_proc_avg              | double              | YES  |      | NULL    |       |
| Cop_proc_p90              | double              | YES  |      | NULL    |       |
| Cop_proc_max              | double              | YES  |      | NULL    |       |
| Cop_proc_addr             | varchar(64)         | YES  |      | NULL    |       |
| Cop_wait_avg              | double              | YES  |      | NULL    |       |
| Cop_wait_p90              | double              | YES  |      | NULL    |       |
| Cop_wait_max              | double              | YES  |      | NULL    |       |
| Cop_wait_addr             | varchar(64)         | YES  |      | NULL    |       |
| Mem_max                   | bigint(20)          | YES  |      | NULL    |       |
| Disk_max                  | bigint(20)          | YES  |      | NULL    |       |
| Succ                      | tinyint(1)          | YES  |      | NULL    |       |
| Plan_from_cache           | tinyint(1)          | YES  |      | NULL    |       |
| Plan                      | longblob            | YES  |      | NULL    |       |
| Plan_digest               | varchar(128)        | YES  |      | NULL    |       |
| Prev_stmt                 | longblob            | YES  |      | NULL    |       |
| Query                     | longblob            | YES  |      | NULL    |       |
+---------------------------+---------------------+------+------+---------+-------+
54 rows in set (0.00 sec)
```

CLUSTER_SLOW_QUERY table

CLUSTER_SLOW_QUERY 表中提供了集群所有节点的慢查询相关的信息，其内容通过解析 TiDB 慢查询日志而来，该表使用上和 SLOW_QUERY 表一样。CLUSTER_SLOW_QUERY 表结构上比 SLOW_QUERY 多一列 INSTANCE，表示该行慢查询信息来自的 TiDB 节点地址。关于如何使用该表调查和改善慢查询，请参考慢查询日志文档。

```
desc cluster_slow_query;
```

```
+---------------------------+---------------------+------+------+---------+-------+
| Field                     | Type                | Null | Key  | Default | Extra |
+---------------------------+---------------------+------+------+---------+-------+
| INSTANCE                  | varchar(64)         | YES  |      | NULL    |       |
| Time                      | timestamp(6)        | YES  |      | NULL    |       |
| Txn_start_ts              | bigint(20) unsigned | YES  |      | NULL    |       |
| User                      | varchar(64)         | YES  |      | NULL    |       |
| Host                      | varchar(64)         | YES  |      | NULL    |       |
| Conn_ID                   | bigint(20) unsigned | YES  |      | NULL    |       |
| Query_time                | double              | YES  |      | NULL    |       |
| Parse_time                | double              | YES  |      | NULL    |       |
| Compile_time              | double              | YES  |      | NULL    |       |
| Rewrite_time              | double              | YES  |      | NULL    |       |
| Preproc_subqueries        | bigint(20) unsigned | YES  |      | NULL    |       |
| Preproc_subqueries_time   | double              | YES  |      | NULL    |       |
| Optimize_time             | double              | YES  |      | NULL    |       |
| Wait_TS                   | double              | YES  |      | NULL    |       |
| Prewrite_time             | double              | YES  |      | NULL    |       |
| Wait_prewrite_binlog_time | double              | YES  |      | NULL    |       |
| Commit_time               | double              | YES  |      | NULL    |       |
| Get_commit_ts_time        | double              | YES  |      | NULL    |       |
| Commit_backoff_time       | double              | YES  |      | NULL    |       |
| Backoff_types             | varchar(64)         | YES  |      | NULL    |       |
| Resolve_lock_time         | double              | YES  |      | NULL    |       |
| Local_latch_wait_time     | double              | YES  |      | NULL    |       |
| Write_keys                | bigint(22)          | YES  |      | NULL    |       |
| Write_size                | bigint(22)          | YES  |      | NULL    |       |
| Prewrite_region           | bigint(22)          | YES  |      | NULL    |       |
| Txn_retry                 | bigint(22)          | YES  |      | NULL    |       |
| Cop_time                  | double              | YES  |      | NULL    |       |
| Process_time              | double              | YES  |      | NULL    |       |
| Wait_time                 | double              | YES  |      | NULL    |       |
| Backoff_time              | double              | YES  |      | NULL    |       |
| LockKeys_time             | double              | YES  |      | NULL    |       |
| Request_count             | bigint(20) unsigned | YES  |      | NULL    |       |
| Total_keys                | bigint(20) unsigned | YES  |      | NULL    |       |
| Process_keys              | bigint(20) unsigned | YES  |      | NULL    |       |
| DB                        | varchar(64)         | YES  |      | NULL    |       |
| Index_names               | varchar(100)        | YES  |      | NULL    |       |
| Is_internal               | tinyint(1)          | YES  |      | NULL    |       |
```

```
| Digest                    | varchar(64)         | YES  |      | NULL    |       |
| Stats                     | varchar(512)        | YES  |      | NULL    |       |
| Cop_proc_avg              | double              | YES  |      | NULL    |       |
| Cop_proc_p90              | double              | YES  |      | NULL    |       |
| Cop_proc_max              | double              | YES  |      | NULL    |       |
| Cop_proc_addr             | varchar(64)         | YES  |      | NULL    |       |
| Cop_wait_avg              | double              | YES  |      | NULL    |       |
| Cop_wait_p90              | double              | YES  |      | NULL    |       |
| Cop_wait_max              | double              | YES  |      | NULL    |       |
| Cop_wait_addr             | varchar(64)         | YES  |      | NULL    |       |
| Mem_max                   | bigint(20)          | YES  |      | NULL    |       |
| Disk_max                  | bigint(20)          | YES  |      | NULL    |       |
| Succ                      | tinyint(1)          | YES  |      | NULL    |       |
| Plan_from_cache           | tinyint(1)          | YES  |      | NULL    |       |
| Plan                      | longblob            | YES  |      | NULL    |       |
| Plan_digest               | varchar(128)        | YES  |      | NULL    |       |
| Prev_stmt                 | longblob            | YES  |      | NULL    |       |
| Query                     | longblob            | YES  |      | NULL    |       |
+---------------------------+---------------------+------+------+---------+-------+
55 rows in set (0.00 sec)
```

查询集群系统表时，TiDB 也会将相关计算下推给其他节点执行，而不是把所有节点的数据都取回来，可以查看执行计划，如下：

```
desc SELECT count(*) FROM cluster_slow_query WHERE user = 'u1';
```

```
+--
    ↪ ------------------------+----------+-----------+--------------------------+-------------------------------
    ↪
| id                        | estRows  | task      | access object            | operator info
    ↪                                      |
+--
    ↪ ------------------------+----------+-----------+--------------------------+-------------------------------
    ↪
| StreamAgg_20              | 1.00     | root      |                          | funcs:count(Column
    ↪ #53)->Column#51                      |
| └─TableReader_21          | 1.00     | root      |                           | data:StreamAgg_9
    ↪                                        |
|   └─StreamAgg_9           | 1.00     | cop[tidb] |                           | funcs:count(1)->
    ↪ Column#53                              |
|     └─Selection_19        | 10.00    | cop[tidb] |                           | eq(
    ↪ information_schema.cluster_slow_query.user, "u1") |
|       └─TableFullScan_18 | 10000.00 | cop[tidb] | table:CLUSTER_SLOW_QUERY | keep order:false,
    ↪  stats:pseudo                          |
+--
    ↪ ------------------------+----------+-----------+--------------------------+-------------------------------
```

```
    ↪
```

上面执行计划表示，会将 user = u1 条件下推给其他的 (cop) TiDB 节点执行，也会把聚合算子（即图中的 StreamAgg 算子）下推。

目前由于没有对系统表收集统计信息，所以有时会导致某些聚合算子不能下推，导致执行较慢，用户可以通过手动指定聚合下推的 SQL HINT 来将聚合算子下推，示例如下：

```
SELECT /*+ AGG_TO_COP() */ count(*) FROM cluster_slow_query GROUP BY user;
```

12.11.13.2.27 STATISTICS

STATISTICS 表提供了关于表索引的信息。

```
USE information_schema;
DESC statistics;
```

```
+---------------+---------------+------+------+---------+-------+
| Field         | Type          | Null | Key  | Default | Extra |
+---------------+---------------+------+------+---------+-------+
| TABLE_CATALOG | varchar(512)  | YES  |      | NULL    |       |
| TABLE_SCHEMA  | varchar(64)   | YES  |      | NULL    |       |
| TABLE_NAME    | varchar(64)   | YES  |      | NULL    |       |
| NON_UNIQUE    | varchar(1)    | YES  |      | NULL    |       |
| INDEX_SCHEMA  | varchar(64)   | YES  |      | NULL    |       |
| INDEX_NAME    | varchar(64)   | YES  |      | NULL    |       |
| SEQ_IN_INDEX  | bigint(2)     | YES  |      | NULL    |       |
| COLUMN_NAME   | varchar(21)   | YES  |      | NULL    |       |
| COLLATION     | varchar(1)    | YES  |      | NULL    |       |
| CARDINALITY   | bigint(21)    | YES  |      | NULL    |       |
| SUB_PART      | bigint(3)     | YES  |      | NULL    |       |
| PACKED        | varchar(10)   | YES  |      | NULL    |       |
| NULLABLE      | varchar(3)    | YES  |      | NULL    |       |
| INDEX_TYPE    | varchar(16)   | YES  |      | NULL    |       |
| COMMENT       | varchar(16)   | YES  |      | NULL    |       |
| INDEX_COMMENT | varchar(1024) | YES  |      | NULL    |       |
| IS_VISIBLE    | varchar(3)    | YES  |      | NULL    |       |
| Expression    | varchar(64)   | YES  |      | NULL    |       |
+---------------+---------------+------+------+---------+-------+
18 rows in set (0.00 sec)
```

STATISTICS 表中列的含义如下：

- TABLE_CATALOG：包含索引的表所属的目录的名称。这个值总是 def。
- TABLE_SCHEMA：包含索引的表所属的数据库的名称。
- TABLE_NAME：包含索引的表的名称。

- NON_UNIQUE：如果索引不能包含重复项，则为 0；如果可以，则为 1。
- INDEX_SCHEMA：索引所属的数据库的名称。
- INDEX_NAME：索引的名称。如果索引是主键，那么名称总是 PRIMARY。
- SEQ_IN_INDEX：索引中的列序号，从 1 开始。
- COLUMN_NAME：列名。请参见表达式列的说明。
- COLLATION：列在索引中的排序方式。取值可以是 A（升序）、D（降序）或 NULL（未排序）。
- CARDINALITY：索引中唯一值的数量的估计。要更新这个数字，执行 ANALYZE TABLE。
- SUB_PART：索引的前缀。如果只对列的部分前缀进行索引，则为索引字符的数量；如果对整个列进行索引，则为 NULL。
- PACKED：TiDB 未使用该字段。这个值总是 NULL。
- NULLABLE：如果列可能包含 NULL 值，则值为 YES；如果不包含，则值为 ''。
- INDEX_TYPE：索引的类型。
- COMMENT：其他与索引有关的信息。
- INDEX_COMMENT：在创建索引时为索引提供的带有注释属性的任何注释。
- IS_VISIBLE：优化器能否使用该索引。
- Expression 对于非表达式部分的索引键，这个值为 NULL；对于表达式部分的索引键，这个值为表达式本身。可参考表达式索引

下列语句是等价的：

```
SELECT * FROM INFORMATION_SCHEMA.STATISTICS
  WHERE table_name = 'tbl_name'
  AND table_schema = 'db_name'

SHOW INDEX
  FROM tbl_name
  FROM db_name
```

12.11.13.2.28 TABLES

TABLES 表提供了数据库里关于表的信息。

```
USE information_schema;
DESC tables;
```

```
+---------------------------+---------------+------+------+----------+-------+
| Field                     | Type          | Null | Key  | Default  | Extra |
+---------------------------+---------------+------+------+----------+-------+
| TABLE_CATALOG             | varchar(512)  | YES  |      | NULL     |       |
| TABLE_SCHEMA              | varchar(64)   | YES  |      | NULL     |       |
| TABLE_NAME                | varchar(64)   | YES  |      | NULL     |       |
| TABLE_TYPE                | varchar(64)   | YES  |      | NULL     |       |
| ENGINE                    | varchar(64)   | YES  |      | NULL     |       |
| VERSION                   | bigint(21)    | YES  |      | NULL     |       |
| ROW_FORMAT                | varchar(10)   | YES  |      | NULL     |       |
| TABLE_ROWS                | bigint(21)    | YES  |      | NULL     |       |
```

```
| AVG_ROW_LENGTH            | bigint(21)    | YES  |      | NULL      |       |
| DATA_LENGTH               | bigint(21)    | YES  |      | NULL      |       |
| MAX_DATA_LENGTH           | bigint(21)    | YES  |      | NULL      |       |
| INDEX_LENGTH              | bigint(21)    | YES  |      | NULL      |       |
| DATA_FREE                 | bigint(21)    | YES  |      | NULL      |       |
| AUTO_INCREMENT            | bigint(21)    | YES  |      | NULL      |       |
| CREATE_TIME               | datetime      | YES  |      | NULL      |       |
| UPDATE_TIME               | datetime      | YES  |      | NULL      |       |
| CHECK_TIME                | datetime      | YES  |      | NULL      |       |
| TABLE_COLLATION           | varchar(32)   | NO   |      | utf8_bin  |       |
| CHECKSUM                  | bigint(21)    | YES  |      | NULL      |       |
| CREATE_OPTIONS            | varchar(255)  | YES  |      | NULL      |       |
| TABLE_COMMENT             | varchar(2048) | YES  |      | NULL      |       |
| TIDB_TABLE_ID             | bigint(21)    | YES  |      | NULL      |       |
| TIDB_ROW_ID_SHARDING_INFO | varchar(255)  | YES  |      | NULL      |       |
+---------------------------+---------------+------+------+-----------+-------+
23 rows in set (0.00 sec)
```

```
SELECT * FROM tables WHERE table_schema='mysql' AND table_name='user'\G
```

```
*************************** 1. row ***************************
            TABLE_CATALOG: def
             TABLE_SCHEMA: mysql
               TABLE_NAME: user
               TABLE_TYPE: BASE TABLE
                   ENGINE: InnoDB
                  VERSION: 10
               ROW_FORMAT: Compact
               TABLE_ROWS: 0
           AVG_ROW_LENGTH: 0
              DATA_LENGTH: 0
          MAX_DATA_LENGTH: 0
             INDEX_LENGTH: 0
                DATA_FREE: 0
           AUTO_INCREMENT: NULL
              CREATE_TIME: 2020-07-05 09:25:51
              UPDATE_TIME: NULL
               CHECK_TIME: NULL
          TABLE_COLLATION: utf8mb4_bin
                 CHECKSUM: NULL
           CREATE_OPTIONS:
            TABLE_COMMENT:
            TIDB_TABLE_ID: 5
TIDB_ROW_ID_SHARDING_INFO: NULL
1 row in set (0.00 sec)
```

下列语句是等价的：

```
SELECT table_name FROM INFORMATION_SCHEMA.TABLES
  WHERE table_schema = 'db_name'
  [AND table_name LIKE 'wild']

SHOW TABLES
  FROM db_name
  [LIKE 'wild']
```

TABLES 表各列字段含义如下：

- TABLE_CATALOG：表所属的目录的名称。该值始终为 def。
- TABLE_SCHEMA：表所属数据库的名称。
- TABLE_NAME：表的名称。
- TABLE_TYPE：表的类型。
- ENGINE：存储引擎类型。该值暂为 ‘InnoDB’。
- VERSION：版本，默认值为 10。
- ROW_FORMAT：行格式。该值暂为 ‘Compact’。
- TABLE_ROWS：统计信息中该表所存的行数。
- AVG_ROW_LENGTH：该表中所存数据的平均行长度。平均行长度 = DATA_LENGTH / 统计信息中的行数。
- DATA_LENGTH：数据长度。数据长度 = 统计信息中的行数 × 元组各列存储长度和，这里尚未考虑 TiKV 的副本数。
- MAX_DATA_LENGTH：最大数据长度。该值暂为 0，表示没有最大数据长度的限制。
- INDEX_LENGTH：索引长度。索引长度 = 统计信息中的行数 × 索引元组各列长度和，这里尚未考虑 TiKV 的副本数。
- DATA_FREE：空间碎片。该值暂为 0。
- AUTO_INCREMENT：该表中自增主键自动增量的当前值。
- CREATE_TIME：该表的创建时间。
- UPDATE_TIME：该表的更新时间。
- CHECK_TIME：该表的检查时间。
- TABLE_COLLATION：该表的字符校验编码集。
- CHECKSUM：校验和。
- CREATE_OPTIONS：创建选项。
- TABLE_COMMENT：表的注释、备注。

表中的信息大部分定义自 MySQL，只有两列是 TiDB 新增的：

- TIDB_TABLE_ID：标识表的内部 ID，该 ID 在一个 TiDB 集群内部唯一。
- TIDB_ROW_ID_SHARDING_INFO：标识表的 Sharding 类型，可能的值为：
  - "NOT_SHARDED"：表未被 Shard。
  - "NOT_SHARDED(PK_IS_HANDLE)"：一个定义了整型主键的表未被 Shard。
  - "PK_AUTO_RANDOM_BITS={bit_number}"：一个定义了整型主键的表由于定义了 AUTO_RANDOM 而被 Shard。
  - "SHARD_BITS={bit_number}"：表使用 SHARD_ROW_ID_BITS={bit_number} 进行了 Shard。
  - NULL：表属于系统表或 View，无法被 Shard。

12.11.13.2.29 TABLE_CONSTRAINTS

TABLE_CONSTRAINTS 表记录了表的约束信息。

```
USE information_schema;
DESC table_constraints;
```

```
+--------------------+--------------+------+------+---------+-------+
| Field              | Type         | Null | Key  | Default | Extra |
+--------------------+--------------+------+------+---------+-------+
| CONSTRAINT_CATALOG | varchar(512) | YES  |      | NULL    |       |
| CONSTRAINT_SCHEMA  | varchar(64)  | YES  |      | NULL    |       |
| CONSTRAINT_NAME    | varchar(64)  | YES  |      | NULL    |       |
| TABLE_SCHEMA       | varchar(64)  | YES  |      | NULL    |       |
| TABLE_NAME         | varchar(64)  | YES  |      | NULL    |       |
| CONSTRAINT_TYPE    | varchar(64)  | YES  |      | NULL    |       |
+--------------------+--------------+------+------+---------+-------+
6 rows in set (0.00 sec)
```

```
SELECT * FROM table_constraints WHERE constraint_type='UNIQUE';
```

```
+--
    ↪ ------------------+--------------------+-------------------------+--------------------+-----------------
    ↪
| CONSTRAINT_CATALOG | CONSTRAINT_SCHEMA  | CONSTRAINT_NAME         | TABLE_SCHEMA       |
    ↪ TABLE_NAME                           | CONSTRAINT_TYPE |
+--
    ↪ ------------------+--------------------+-------------------------+--------------------+-----------------
    ↪
| def                | mysql              | name                    | mysql              |
    ↪ help_topic                           | UNIQUE          |
| def                | mysql              | tbl                     | mysql              |
    ↪ stats_meta                           | UNIQUE          |
| def                | mysql              | tbl                     | mysql              |
    ↪ stats_histograms                     | UNIQUE          |
| def                | mysql              | tbl                     | mysql              |
    ↪ stats_buckets                        | UNIQUE          |
| def                | mysql              | delete_range_index      | mysql              |
    ↪ gc_delete_range                      | UNIQUE          |
| def                | mysql              | delete_range_done_index | mysql              |
    ↪ gc_delete_range_done                 | UNIQUE          |
| def                | PERFORMANCE_SCHEMA | SCHEMA_NAME             | PERFORMANCE_SCHEMA |
    ↪ events_statements_summary_by_digest | UNIQUE          |
+--
    ↪ ------------------+--------------------+-------------------------+--------------------+-----------------
    ↪
```

```
7 rows in set (0.01 sec)
```

TABLE_CONSTRAINTS 表中列的含义如下：

- CONSTRAINT_CATALOG：约束所属的目录的名称。这个值总是 def。
- CONSTRAINT_SCHEMA：约束所属的数据库的名称。
- CONSTRAINT_NAME：约束的名称。
- TABLE_NAME：表的名称。
- CONSTRAINT_TYPE：约束的类型。取值可以是 UNIQUE、PRIMARY KEY 或者 FOREIGN KEY。UNIQUE 和 PRIMARY KEY 信息与 SHOW INDEX 语句的执行结果类似。

12.11.13.2.30 TABLE_STORAGE_STATS

TABLE_STORAGE_STATS 表提供有关由存储引擎 (TiKV) 存储的表大小的信息。

```
USE information_schema;
DESC table_storage_stats;
```

```
+--------------------+-------------+------+------+---------+-------+
| Field              | Type        | Null | Key  | Default | Extra |
+--------------------+-------------+------+------+---------+-------+
| TABLE_SCHEMA       | varchar(64) | YES  |      | NULL    |       |
| TABLE_NAME         | varchar(64) | YES  |      | NULL    |       |
| TABLE_ID           | bigint(21)  | YES  |      | NULL    |       |
| PEER_COUNT         | bigint(21)  | YES  |      | NULL    |       |
| REGION_COUNT       | bigint(21)  | YES  |      | NULL    |       |
| EMPTY_REGION_COUNT | bigint(21)  | YES  |      | NULL    |       |
| TABLE_SIZE         | bigint(64)  | YES  |      | NULL    |       |
| TABLE_KEYS         | bigint(64)  | YES  |      | NULL    |       |
+--------------------+-------------+------+------+---------+-------+
8 rows in set (0.00 sec)
```

```
CREATE TABLE test.t1 (id INT);
INSERT INTO test.t1 VALUES (1);
SELECT * FROM table_storage_stats WHERE table_schema = 'test' AND table_name = 't1'\G
```

```
*************************** 1. row ***************************
      TABLE_SCHEMA: test
        TABLE_NAME: t1
          TABLE_ID: 56
        PEER_COUNT: 1
      REGION_COUNT: 1
EMPTY_REGION_COUNT: 1
        TABLE_SIZE: 1
        TABLE_KEYS: 0
1 row in set (0.00 sec)
```

12.11.13.2.31 TIDB_HOT_REGIONS

TIDB_HOT_REGIONS 表提供了关于热点 Region 的相关信息。

```
USE information_schema;
DESC tidb_hot_regions;
```

```
+----------------+-------------+------+------+---------+-------+
| Field          | Type        | Null | Key  | Default | Extra |
+----------------+-------------+------+------+---------+-------+
| TABLE_ID       | bigint(21)  | YES  |      | NULL    |       |
| INDEX_ID       | bigint(21)  | YES  |      | NULL    |       |
| DB_NAME        | varchar(64) | YES  |      | NULL    |       |
| TABLE_NAME     | varchar(64) | YES  |      | NULL    |       |
| INDEX_NAME     | varchar(64) | YES  |      | NULL    |       |
| REGION_ID      | bigint(21)  | YES  |      | NULL    |       |
| TYPE           | varchar(64) | YES  |      | NULL    |       |
| MAX_HOT_DEGREE | bigint(21)  | YES  |      | NULL    |       |
| REGION_COUNT   | bigint(21)  | YES  |      | NULL    |       |
| FLOW_BYTES     | bigint(21)  | YES  |      | NULL    |       |
+----------------+-------------+------+------+---------+-------+
10 rows in set (0.00 sec)
```

TIDB_HOT_REGIONS 表各列字段含义如下：

- TABLE_ID：热点 Region 所在表的 ID。
- INDEX_ID：热点 Region 所在索引的 ID。
- DB_NAME：热点 Region 所在数据库对象的数据库名。
- TABLE_NAME：热点 Region 所在表的名称。
- INDEX_NAME：热点 Region 所在索引的名称。
- REGION_ID：热点 Region 的 ID。
- TYPE：热点 Region 的类型。
- MAX_HOT_DEGREE：该 Region 的最大热度。
- REGION_COUNT：所在实例的 Region 数量。
- FLOW_BYTES：该 Region 内读写的字节数量。

12.11.13.2.32 TIDB_INDEXES

TIDB_INDEXES 记录了所有表中的 INDEX 信息。

```
USE information_schema;
DESC tidb_indexes;
```

```
+---------------+---------------+------+------+---------+-------+
| Field         | Type          | Null | Key  | Default | Extra |
+---------------+---------------+------+------+---------+-------+
| TABLE_SCHEMA  | varchar(64)   | YES  |      | NULL    |       |
```

```
| TABLE_NAME    | varchar(64)   | YES  |     | NULL    |       |
| NON_UNIQUE    | bigint(21)    | YES  |     | NULL    |       |
| KEY_NAME      | varchar(64)   | YES  |     | NULL    |       |
| SEQ_IN_INDEX  | bigint(21)    | YES  |     | NULL    |       |
| COLUMN_NAME   | varchar(64)   | YES  |     | NULL    |       |
| SUB_PART      | bigint(21)    | YES  |     | NULL    |       |
| INDEX_COMMENT | varchar(2048) | YES  |     | NULL    |       |
| Expression    | varchar(64)   | YES  |     | NULL    |       |
| INDEX_ID      | bigint(21)    | YES  |     | NULL    |       |
+---------------+---------------+------+-----+---------+-------+
10 rows in set (0.00 sec)
```

INDEX_ID 是 TiDB 为每个索引分配的唯一 ID。它可以与从另一个表或 API 获得的 INDEX_ID 一起执行 join 操作。

例如，你可以在SLOW_QUERY 表 中获取某些慢查询所涉及的 TABLE_ID 和 INDEX_ID，然后使用以下 SQL 语句获取特定索引信息：

```
SELECT
 tidb_indexes.*
FROM
 tidb_indexes,
 tables
WHERE
  tidb_indexes.table_schema = tables.table_schema
 AND tidb_indexes.table_name = tidb_indexes.table_name
 AND tables.tidb_table_id = ?
 AND index_id = ?
```

TIDB_INDEXES 表中列的含义如下：

- TABLE_SCHEMA：索引所在表的所属数据库的名称。
- TABLE_NAME：索引所在表的名称。
- NON_UNIQUE：如果索引是唯一的，则为 0，否则为 1。
- KEY_NAME：索引的名称。如果索引是主键，则名称为 PRIMARY。
- SEQ_IN_INDEX：索引中列的顺序编号，从 1 开始。
- COLUMN_NAME：索引所在的列名。
- SUB_PART：索引前缀长度。如果列是部分被索引，则该值为被索引的字符数量，否则为 NULL。
- INDEX_COMMENT：创建索引时以 COMMENT 标注的注释。
- INDEX_ID：索引的 ID。

12.11.13.2.33 TIDB_SERVERS_INFO

TIDB_SERVERS_INFO 表提供了 TiDB 集群中 TiDB 服务器的信息（即 tidb-server 进程）。

```
USE information_schema;
DESC tidb_servers_info;
```

```
+---------------+-------------+------+------+---------+-------+
| Field         | Type        | Null | Key  | Default | Extra |
+---------------+-------------+------+------+---------+-------+
| DDL_ID        | varchar(64) | YES  |      | NULL    |       |
| IP            | varchar(64) | YES  |      | NULL    |       |
| PORT          | bigint(21)  | YES  |      | NULL    |       |
| STATUS_PORT   | bigint(21)  | YES  |      | NULL    |       |
| LEASE         | varchar(64) | YES  |      | NULL    |       |
| VERSION       | varchar(64) | YES  |      | NULL    |       |
| GIT_HASH      | varchar(64) | YES  |      | NULL    |       |
| BINLOG_STATUS | varchar(64) | YES  |      | NULL    |       |
+---------------+-------------+------+------+---------+-------+
8 rows in set (0.00 sec)
```

```
SELECT * FROM tidb_servers_info\G
```

```
*************************** 1. row ***************************
       DDL_ID: 771c169d-0a3a-48ea-a93c-a4d6751d3674
           IP: 0.0.0.0
         PORT: 4000
  STATUS_PORT: 10080
        LEASE: 45s
      VERSION: 5.7.25-TiDB-v4.0.0-beta.2-720-g0df3b74f5
     GIT_HASH: 0df3b74f55f8f8fbde39bbd5d471783f49dc10f7
BINLOG_STATUS: Off
1 row in set (0.00 sec)
```

12.11.13.2.34 TIFLASH_REPLICA

TIFLASH_REPLICA 表提供了有关可用的 TiFlash 副本的信息。

```
USE information_schema;
DESC tiflash_replica;
```

```
+-----------------+-------------+------+------+---------+-------+
| Field           | Type        | Null | Key  | Default | Extra |
+-----------------+-------------+------+------+---------+-------+
| TABLE_SCHEMA    | varchar(64) | YES  |      | NULL    |       |
| TABLE_NAME      | varchar(64) | YES  |      | NULL    |       |
| TABLE_ID        | bigint(21)  | YES  |      | NULL    |       |
| REPLICA_COUNT   | bigint(64)  | YES  |      | NULL    |       |
| LOCATION_LABELS | varchar(64) | YES  |      | NULL    |       |
| AVAILABLE       | tinyint(1)  | YES  |      | NULL    |       |
| PROGRESS        | double      | YES  |      | NULL    |       |
```

```
+-----------------+-------------+------+------+---------+-------+
7 rows in set (0.01 sec)
```

12.11.13.2.35 TIKV_REGION_PEERS

TIKV_REGION_PEERS 表提供了 TiKV 中单个 Region 节点的详细信息，比如它是一个 learner 还是一个 leader。

```
USE information_schema;
DESC tikv_region_peers;
```

```
+--------------+-------------+------+------+---------+-------+
| Field        | Type        | Null | Key  | Default | Extra |
+--------------+-------------+------+------+---------+-------+
| REGION_ID    | bigint(21)  | YES  |      | NULL    |       |
| PEER_ID      | bigint(21)  | YES  |      | NULL    |       |
| STORE_ID     | bigint(21)  | YES  |      | NULL    |       |
| IS_LEARNER   | tinyint(1)  | NO   |      | 0       |       |
| IS_LEADER    | tinyint(1)  | NO   |      | 0       |       |
| STATUS       | varchar(10) | YES  |      | 0       |       |
| DOWN_SECONDS | bigint(21)  | YES  |      | 0       |       |
+--------------+-------------+------+------+---------+-------+
7 rows in set (0.01 sec)
```

例如，使用以下 SQL 语句，你可以查询 WRITTEN_BYTES 最大的前 3 个 Region 所在的 TiKV 地址：

```
SELECT
 address,
 tikv.address,
 region.region_id
FROM
 tikv_store_status tikv,
 tikv_region_peers peer,
 (SELECT * FROM tikv_region_status ORDER BY written_bytes DESC LIMIT 3) region
WHERE
  region.region_id = peer.region_id
 AND peer.is_leader = 1
 AND peer.store_id = tikv.store_id;
```

TIKV_REGION_PEERS 表各列含义如下：

- REGION_ID：REGION 的 ID。
- PEER_ID：REGION 中对应的副本 PEER 的 ID。
- STORE_ID：REGION 所在 TiKV Store 的 ID。
- IS_LEARNER：PEER 是否是 LEARNER。
- IS_LEADER：PEER 是否是 LEADER。
- STATUS：PEER 的状态，一共有 3 种状态：

  - PENDING：暂时不可用状态。
  - DOWN：下线转态，该 PEER 不再提供服务。
  - NORMAL: 正常状态。
- DOWN_SECONDS：处于下线状态的时间，单位是秒。

12.11.13.2.36 TIKV_REGION_STATUS

TIKV_REGION_STATUS 表通过 PD 的 API 展示 TiKV Region 的基本信息，比如 Region ID、开始和结束键值以及读写流量。

```
USE information_schema;
DESC tikv_region_status;
```

```
+--------------------------+-------------+------+------+---------+-------+
| Field                    | Type        | Null | Key  | Default | Extra |
+--------------------------+-------------+------+------+---------+-------+
| REGION_ID                | bigint(21)  | YES  |      | NULL    |       |
| START_KEY                | text        | YES  |      | NULL    |       |
| END_KEY                  | text        | YES  |      | NULL    |       |
| TABLE_ID                 | bigint(21)  | YES  |      | NULL    |       |
| DB_NAME                  | varchar(64) | YES  |      | NULL    |       |
| TABLE_NAME               | varchar(64) | YES  |      | NULL    |       |
| IS_INDEX                 | tinyint(1)  | NO   |      | 0       |       |
| INDEX_ID                 | bigint(21)  | YES  |      | NULL    |       |
| INDEX_NAME               | varchar(64) | YES  |      | NULL    |       |
| EPOCH_CONF_VER           | bigint(21)  | YES  |      | NULL    |       |
| EPOCH_VERSION            | bigint(21)  | YES  |      | NULL    |       |
| WRITTEN_BYTES            | bigint(21)  | YES  |      | NULL    |       |
| READ_BYTES               | bigint(21)  | YES  |      | NULL    |       |
| APPROXIMATE_SIZE         | bigint(21)  | YES  |      | NULL    |       |
| APPROXIMATE_KEYS         | bigint(21)  | YES  |      | NULL    |       |
| REPLICATIONSTATUS_STATE  | varchar(64) | YES  |      | NULL    |       |
| REPLICATIONSTATUS_STATEID | bigint(21)  | YES  |      | NULL    |       |
+--------------------------+-------------+------+------+---------+-------+
17 rows in set (0.00 sec)
```

TIKV_REGION_STATUS 表中列的含义如下：

- REGION_ID：Region 的 ID。
- START_KEY：Region 的起始 key 的值。
- END_KEY：Region 的末尾 key 的值。
- TABLE_ID：Region 所属的表的 ID。
- DB_NAME：TABLE_ID 所属的数据库的名称。
- TABLE_NAME：Region 所属的表的名称。
- IS_INDEX：Region 数据是否是索引，0 代表不是索引，1 代表是索引。如果当前 Region 同时包含表数据和索引数据，会有多行记录，IS_INDEX 分别是 0 和 1。

- INDEX_ID：Region 所属的索引的 ID。如果 IS_INDEX 为 0，这一列的值就为 NULL。
- INDEX_NAME：Region 所属的索引的名称。如果 IS_INDEX 为 0，这一列的值就为 NULL。
- EPOCH_CONF_VER：Region 的配置的版本号，在增加或减少 peer 时版本号会递增。
- EPOCH_VERSION：Region 的当前版本号，在分裂或合并时版本号会递增。
- WRITTEN_BYTES：已经往 Region 写入的数据量 (bytes)。
- READ_BYTES：已经从 Region 读取的数据量 (bytes)。
- APPROXIMATE_SIZE：Region 的近似数据量 (MB)。
- APPROXIMATE_KEYS：Region 中 key 的近似数量。
- REPLICATIONSTATUS_STATE:Region 当前的同步状态,可能为 UNKNOWN / SIMPLE_MAJORITY / INTEGRITY_OVER_LABEL 其中一种状态。
- REPLICATIONSTATUS_STATEID：REPLICATIONSTATUS_STATE 对应的标识符。

通过在 EPOCH_CONF_VER、WRITTEN_BYTES 和 READ_BYTES 列上执行 ORDER BY X LIMIT Y 操作，你可以在 pd-ctl 中实现 top confver、top read 和 top write 操作。

你可以使用以下 SQL 语句查询写入数据最多的前 3 个 Region：

```
SELECT * FROM tikv_region_status ORDER BY written_bytes DESC LIMIT 3;
```

12.11.13.2.37 TIKV_STORE_STATUS

TIKV_STORE_STATUS 表通过 PD 的 API 显示了 TiKV 节点的一些基本信息，例如在集群中分配的 ID，地址和端口，状态，容量以及当前节点的 Region leader 的数量。

```
USE information_schema;
DESC tikv_store_status;
```

```
+------------------+-------------+------+------+---------+-------+
| Field            | Type        | Null | Key  | Default | Extra |
+------------------+-------------+------+------+---------+-------+
| STORE_ID         | bigint(21)  | YES  |      | NULL    |       |
| ADDRESS          | varchar(64) | YES  |      | NULL    |       |
| STORE_STATE      | bigint(21)  | YES  |      | NULL    |       |
| STORE_STATE_NAME | varchar(64) | YES  |      | NULL    |       |
| LABEL            | json        | YES  |      | NULL    |       |
| VERSION          | varchar(64) | YES  |      | NULL    |       |
| CAPACITY         | varchar(64) | YES  |      | NULL    |       |
| AVAILABLE        | varchar(64) | YES  |      | NULL    |       |
| LEADER_COUNT     | bigint(21)  | YES  |      | NULL    |       |
| LEADER_WEIGHT    | double      | YES  |      | NULL    |       |
| LEADER_SCORE     | double      | YES  |      | NULL    |       |
| LEADER_SIZE      | bigint(21)  | YES  |      | NULL    |       |
| REGION_COUNT     | bigint(21)  | YES  |      | NULL    |       |
| REGION_WEIGHT    | double      | YES  |      | NULL    |       |
| REGION_SCORE     | double      | YES  |      | NULL    |       |
| REGION_SIZE      | bigint(21)  | YES  |      | NULL    |       |
```

```
| START_TS           | datetime    | YES  |     | NULL    |       |
| LAST_HEARTBEAT_TS | datetime    | YES  |     | NULL    |       |
| UPTIME             | varchar(64) | YES  |     | NULL    |       |
+-------------------+-------------+------+-----+---------+-------+
19 rows in set (0.00 sec)
```

TIKV_STORE_STATUS 表中列的含义如下：

- STORE_ID：Store 的 ID。
- ADDRESS：Store 的地址。
- STORE_STATE：Store 状态的标识符，与 STORE_STATE_NAME 相对应。
- STORE_STATE_NAME：Store 状态的名字，为 Up / Offline / Tombstone 中的一种。
- LABEL：给 Store 设置的标签。
- VERSION：Store 的版本号。
- CAPACITY：Store 的存储容量。
- AVAILABLE：Store 的剩余存储空间。
- LEADER_COUNT：Store 上的 leader 的数量。
- LEADER_WEIGHT：Store 的 leader 权重。
- LEADER_SCORE：Store 的 leader 评分。
- LEADER_SIZE：Store 上的所有 leader 的近似总数据量 (MB)。
- REGION_COUNT：Store 上的 Region 总数。
- REGION_WEIGHT：Store 的 Region 权重。
- REGION_SCORE：Store 的 Region 评分。
- REGION_SIZE：Store 上的所有 Region 的近似总数据量 (MB)。
- START_TS：Store 启动时的时间戳。
- LAST_HEARTBEAT_TS：Store 上次发出心跳的时间戳。
- UPTIME：Store 启动以来的总时间。

12.11.13.2.38 USER_PRIVILEGES

USER_PRIVILEGES 表提供了关于全局权限的信息。该表的数据根据 mysql.user 系统表生成。

```
USE information_schema;
DESC user_privileges;
```

```
+----------------+--------------+------+------+---------+-------+
| Field          | Type         | Null | Key  | Default | Extra |
+----------------+--------------+------+------+---------+-------+
| GRANTEE        | varchar(81)  | YES  |      | NULL    |       |
| TABLE_CATALOG  | varchar(512) | YES  |      | NULL    |       |
| PRIVILEGE_TYPE | varchar(64)  | YES  |      | NULL    |       |
| IS_GRANTABLE   | varchar(3)   | YES  |      | NULL    |       |
+----------------+--------------+------+------+---------+-------+
4 rows in set (0.00 sec)
```

```
SELECT * FROM user_privileges;
```

```
+------------+---------------+-------------------------+--------------+
| GRANTEE    | TABLE_CATALOG | PRIVILEGE_TYPE          | IS_GRANTABLE |
+------------+---------------+-------------------------+--------------+
| 'root'@'%' | def           | Select                  | YES          |
| 'root'@'%' | def           | Insert                  | YES          |
| 'root'@'%' | def           | Update                  | YES          |
| 'root'@'%' | def           | Delete                  | YES          |
| 'root'@'%' | def           | Create                  | YES          |
| 'root'@'%' | def           | Drop                    | YES          |
| 'root'@'%' | def           | Process                 | YES          |
| 'root'@'%' | def           | References              | YES          |
| 'root'@'%' | def           | Alter                   | YES          |
| 'root'@'%' | def           | Show Databases          | YES          |
| 'root'@'%' | def           | Super                   | YES          |
| 'root'@'%' | def           | Execute                 | YES          |
| 'root'@'%' | def           | Index                   | YES          |
| 'root'@'%' | def           | Create User             | YES          |
| 'root'@'%' | def           | Trigger                 | YES          |
| 'root'@'%' | def           | Create View             | YES          |
| 'root'@'%' | def           | Show View               | YES          |
| 'root'@'%' | def           | Create Role             | YES          |
| 'root'@'%' | def           | Drop Role               | YES          |
| 'root'@'%' | def           | CREATE TEMPORARY TABLES | YES          |
| 'root'@'%' | def           | LOCK TABLES             | YES          |
| 'root'@'%' | def           | CREATE ROUTINE          | YES          |
| 'root'@'%' | def           | ALTER ROUTINE           | YES          |
| 'root'@'%' | def           | EVENT                   | YES          |
| 'root'@'%' | def           | SHUTDOWN                | YES          |
| 'root'@'%' | def           | RELOAD                  | YES          |
| 'root'@'%' | def           | FILE                    | YES          |
| 'root'@'%' | def           | CONFIG                  | YES          |
+------------+---------------+-------------------------+--------------+
28 rows in set (0.00 sec)
```

USER_PRIVILEGES 表中列的含义如下：

- GRANTEE：被授权的用户名称，格式为 'user_name'@'host_name'。
- TABLE_CATALOG：表所属的目录的名称。该值始终为 def。
- PRIVILEGE_TYPE：被授权的权限类型，每行只列一个权限。
- IS_GRANTABLE：如果用户有 GRANT OPTION 的权限，则为 YES，否则为 NO。

12.11.13.2.39 VIEWS

VIEWS 表提供了关于 SQL 视图的信息。

```
USE information_schema;
DESC views;
```

```
+----------------------+--------------+------+------+---------+-------+
| Field                | Type         | Null | Key  | Default | Extra |
+----------------------+--------------+------+------+---------+-------+
| TABLE_CATALOG        | varchar(512) | NO   |      | NULL    |       |
| TABLE_SCHEMA         | varchar(64)  | NO   |      | NULL    |       |
| TABLE_NAME           | varchar(64)  | NO   |      | NULL    |       |
| VIEW_DEFINITION      | longblob     | NO   |      | NULL    |       |
| CHECK_OPTION         | varchar(8)   | NO   |      | NULL    |       |
| IS_UPDATABLE         | varchar(3)   | NO   |      | NULL    |       |
| DEFINER              | varchar(77)  | NO   |      | NULL    |       |
| SECURITY_TYPE        | varchar(7)   | NO   |      | NULL    |       |
| CHARACTER_SET_CLIENT | varchar(32)  | NO   |      | NULL    |       |
| COLLATION_CONNECTION | varchar(32)  | NO   |      | NULL    |       |
+----------------------+--------------+------+------+---------+-------+
10 rows in set (0.00 sec)
```

```
CREATE VIEW test.v1 AS SELECT 1;
SELECT * FROM views\G
```

```
*************************** 1. row ***************************
       TABLE_CATALOG: def
        TABLE_SCHEMA: test
          TABLE_NAME: v1
     VIEW_DEFINITION: SELECT 1
        CHECK_OPTION: CASCADED
        IS_UPDATABLE: NO
             DEFINER: root@127.0.0.1
       SECURITY_TYPE: DEFINER
CHARACTER_SET_CLIENT: utf8mb4
COLLATION_CONNECTION: utf8mb4_0900_ai_ci
1 row in set (0.00 sec)
```

VIEWS 表中列的含义如下：

- TABLE_CATALOG：视图所属的目录的名称。该值始终为 def。
- TABLE_SCHEMA：视图所属的数据库的名称。
- TABLE_NAME：视图名称。
- VIEW_DEFINITION：视图的定义，由创建视图时 SELECT 部分的语句组成。
- CHECK_OPTION：CHECK_OPTION 的值。取值为 NONE、CASCADE 或 LOCAL。
- IS_UPDATABLE：UPDATE/INSERT/DELETE 是否对该视图可用。在 TiDB，始终为 NO。
- DEFINER：视图的创建者用户名称，格式为 'user_name'@'host_name'。

- SECURITY_TYPE：SQL SECURITY 的值，取值为 DEFINER 或 INVOKER。
- CHARACTER_SET_CLIENT：在视图创建时 session 变量 character_set_client 的值。
- COLLATION_CONNECTION：在视图创建时 session 变量 collation_connection 的值。

12.11.13.3 Metrics Schema

METRICS_SCHEMA 是基于 Prometheus 中 TiDB 监控指标的一组视图。每个表的 PromQL（Prometheus 查询语言）的源均可在INFORMATION_SCHEMA.METRICS_TABLES 表中找到。

```
use metrics_schema;
SELECT * FROM uptime;
SELECT * FROM information_schema.metrics_tables WHERE table_name='uptime'\G
```

```
+----------------------------+-----------------+------------+--------------------+
| time                       | instance        | job        | value              |
+----------------------------+-----------------+------------+--------------------+
| 2020-07-06 15:26:26.203000 | 127.0.0.1:10080 | tidb       | 123.60300016403198 |
| 2020-07-06 15:27:26.203000 | 127.0.0.1:10080 | tidb       | 183.60300016403198 |
| 2020-07-06 15:26:26.203000 | 127.0.0.1:20180 | tikv       | 123.60300016403198 |
| 2020-07-06 15:27:26.203000 | 127.0.0.1:20180 | tikv       | 183.60300016403198 |
| 2020-07-06 15:26:26.203000 | 127.0.0.1:2379  | pd         | 123.60300016403198 |
| 2020-07-06 15:27:26.203000 | 127.0.0.1:2379  | pd         | 183.60300016403198 |
| 2020-07-06 15:26:26.203000 | 127.0.0.1:9090  | prometheus | 123.72300004959106 |
| 2020-07-06 15:27:26.203000 | 127.0.0.1:9090  | prometheus | 183.72300004959106 |
+----------------------------+-----------------+------------+--------------------+
8 rows in set (0.00 sec)
*************************** 1. row ***************************
TABLE_NAME: uptime
    PROMQL: (time() - process_start_time_seconds{$LABEL_CONDITIONS})
    LABELS: instance,job
  QUANTILE: 0
   COMMENT: TiDB uptime since last restart(second)
1 row in set (0.00 sec)
```

```
show tables;
```

```
+-------------------------------------------------+
| Tables_in_metrics_schema                        |
+-------------------------------------------------+
| abnormal_stores                                 |
| etcd_disk_wal_fsync_rate                        |
| etcd_wal_fsync_duration                         |
| etcd_wal_fsync_total_count                      |
| etcd_wal_fsync_total_time                       |
| go_gc_count                                     |
| go_gc_cpu_usage                                 |
```

```
| go_gc_duration                                     |
| go_heap_mem_usage                                  |
| go_threads                                         |
| goroutines_count                                   |
| node_cpu_usage                                     |
| node_disk_available_size                           |
| node_disk_io_util                                  |
| node_disk_iops                                     |
| node_disk_read_latency                             |
| node_disk_size                                     |
..
| tikv_storage_async_request_total_time              |
| tikv_storage_async_requests                        |
| tikv_storage_async_requests_total_count            |
| tikv_storage_command_ops                           |
| tikv_store_size                                    |
| tikv_thread_cpu                                    |
| tikv_thread_nonvoluntary_context_switches          |
| tikv_thread_voluntary_context_switches             |
| tikv_threads_io                                    |
| tikv_threads_state                                 |
| tikv_total_keys                                    |
| tikv_wal_sync_duration                             |
| tikv_wal_sync_max_duration                         |
| tikv_worker_handled_tasks                          |
| tikv_worker_handled_tasks_total_num                |
| tikv_worker_pending_tasks                          |
| tikv_worker_pending_tasks_total_num                |
| tikv_write_stall_avg_duration                      |
| tikv_write_stall_max_duration                      |
| tikv_write_stall_reason                            |
| up                                                 |
| uptime                                             |
+----------------------------------------------------+
626 rows in set (0.00 sec)
```

METRICS_SCHEMA 是监控相关的 summary 表的数据源，例如metrics_summary、metrics_summary_by_label和inspection_summary。

12.11.13.3.1 更多例子

下面以 metrics_schema 中的 tidb_query_duration 监控表为例，介绍监控表相关的使用和原理，其他的监控表原理均类似。

查询 information_schema.metrics_tables 中关于 tidb_query_duration 表相关的信息如下：

```
select * from information_schema.metrics_tables where table_name='tidb_query_duration';
```

```
+--
    ↪ -------------------+-----------------------------------------------------------------------------------
    ↪
| TABLE_NAME           | PROMQL
    ↪
    ↪ | LABELS             | QUANTILE | COMMENT                                                |
+--
    ↪ -------------------+-----------------------------------------------------------------------------------
    ↪
| tidb_query_duration | histogram_quantile($QUANTILE, sum(rate(
    ↪ tidb_server_handle_query_duration_seconds_bucket{$LABEL_CONDITIONS}[$RANGE_DURATION])) by
    ↪  (le,sql_type,instance)) | instance,sql_type | 0.9      | The quantile of TiDB query
    ↪ durations(second) |
+--
    ↪ -------------------+-----------------------------------------------------------------------------------
    ↪
```

- TABLE_NAME：对应于 metrics_schema 中的表名，这里表名是 tidb_query_duration。
- PROMQL：因为监控表的原理是将 SQL 映射成 PromQL 后向 Prometheus 请求数据，并将 Prometheus 返回的结果转换成 SQL 查询结果。该字段是 PromQL 的表达式模板，查询监控表数据时使用查询条件改写模板中的变量，生成最终的查询表达式。
- LABELS：监控项定义的 label，tidb_query_duration 有两个 label，分别是 instance 和 sql_type。
- QUANTILE：百分位。直方图类型的监控数据会指定一个默认百分位。如果值为 0，表示该监控表对应的监控不是直方图。tidb_query_duration 默认查询 0.9 ，也就是 P90 的监控值。
- COMMENT：对这个监控表的解释。可以看出 tidb_query_duration 表是用来查询 TiDB query 执行的百分位时间，如 P999/P99/P90 的查询耗时，单位是秒。

再来看 tidb_query_duration 的表结构：

```
show create table metrics_schema.tidb_query_duration;
```

```
+--
    ↪ -------------------+-----------------------------------------------------------------------------------
    ↪
| Table                | Create Table
    ↪
    ↪ |
+--
    ↪ -------------------+-----------------------------------------------------------------------------------
    ↪
| tidb_query_duration | CREATE TABLE `tidb_query_duration` (
    ↪                                                                                                  |
```

```
|                     |   `time` datetime unsigned DEFAULT CURRENT_TIMESTAMP,
    ↪                                                                    |
|                     |   `instance` varchar(512) DEFAULT NULL,
    ↪                                                                      |
|                     |   `sql_type` varchar(512) DEFAULT NULL,
    ↪                                                                      |
|                     |   `quantile` double unsigned DEFAULT '0.9',
    ↪                                                                  |
|                     |   `value` double unsigned DEFAULT NULL
    ↪                                                                       |
|                     | ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin COMMENT='The
    ↪ quantile of TiDB query durations(second)' |
+--
    ↪ -------------------+-----------------------------------------------------------------------------------
    ↪
```

- `time`：监控项的时间。
- `instance` 和 `sql_type`：是 `tidb_query_duration` 这个监控项的 label。`instance` 表示监控的地址，`sql_type` 表示执行 SQL 的类似。
- `quantile`，百分位，直方图类型的监控都会有该列，表示查询的百分位时间，如 `quantile=0.9` 就是查询 P90 的时间。
- `value`：监控项的值。

下面是查询时间 [2020-03-25 23:40:00, 2020-03-25 23:42:00] 范围内的 P99 的 TiDB Query 耗时：

```
select * from metrics_schema.tidb_query_duration where value is not null and time>='2020-03-25
    ↪ 23:40:00' and time <= '2020-03-25 23:42:00' and quantile=0.99;
```

```
+---------------------+-------------------+----------+----------+----------------+
| time                | instance          | sql_type | quantile | value          |
+---------------------+-------------------+----------+----------+----------------+
| 2020-03-25 23:40:00 | 172.16.5.40:10089 | Insert   | 0.99     | 0.509929485256 |
| 2020-03-25 23:41:00 | 172.16.5.40:10089 | Insert   | 0.99     | 0.494690793986 |
| 2020-03-25 23:42:00 | 172.16.5.40:10089 | Insert   | 0.99     | 0.493460506934 |
| 2020-03-25 23:40:00 | 172.16.5.40:10089 | Select   | 0.99     | 0.152058493415 |
| 2020-03-25 23:41:00 | 172.16.5.40:10089 | Select   | 0.99     | 0.152193879678 |
| 2020-03-25 23:42:00 | 172.16.5.40:10089 | Select   | 0.99     | 0.140498483232 |
| 2020-03-25 23:40:00 | 172.16.5.40:10089 | internal | 0.99     | 0.47104        |
| 2020-03-25 23:41:00 | 172.16.5.40:10089 | internal | 0.99     | 0.11776        |
| 2020-03-25 23:42:00 | 172.16.5.40:10089 | internal | 0.99     | 0.11776        |
+---------------------+-------------------+----------+----------+----------------+
```

以上查询结果的第一行意思是，在 2020-03-25 23:40:00 时，在 TiDB 实例 172.16.5.40:10089 上，Insert 类型的语句的 P99 执行时间是 0.509929485256 秒。其他各行的含义类似，`sql_type` 列的其他值含义如下：

- Select：表示执行的 select 类型的语句。
- internal：表示 TiDB 的内部 SQL 语句，一般是统计信息更新，获取全局变量相关的内部语句。

进一步再查看上面语句的执行计划如下：

```
desc select * from metrics_schema.tidb_query_duration where value is not null and time>='
    ↪ 2020-03-25 23:40:00' and time <= '2020-03-25 23:42:00' and quantile=0.99;
```

```
+--
    ↪ ----------------+----------+------+--------------------------+-------------------------------------------
    ↪
| id                | estRows  | task | access object            | operator info
    ↪
    ↪ |
+--
    ↪ ----------------+----------+------+--------------------------+-------------------------------------------
    ↪
| Selection_5       | 8000.00  | root |                          | not(isnull(Column#5))
    ↪
    ↪ |
| └─MemTableScan_6 | 10000.00 | root | table:tidb_query_duration | PromQL:histogram_quantile
    ↪ (0.99, sum(rate(tidb_server_handle_query_duration_seconds_bucket{}[60s])) by (le,sql_type
    ↪ ,instance)), start_time:2020-03-25 23:40:00, end_time:2020-03-25 23:42:00, step:1m0s |
+--
    ↪ ----------------+----------+------+--------------------------+-------------------------------------------
    ↪
```

可以发现执行计划中有一个 PromQL, 以及查询监控的 start_time 和 end_time，还有 step 值，在实际执行时，TiDB 会调用 Prometheus 的 query_range HTTP API 接口来查询监控数据。

从以上结果可知，在 [2020-03-25 23:40:00, 2020-03-25 23:42:00] 时间范围内，每个 label 只有三个时间的值，间隔时间是 1 分钟，即执行计划中的 step 值。该间隔时间由以下两个 session 变量决定：

- tidb_metric_query_step：查询的分辨率步长。从 Prometheus 的 query_range 接口查询数据时需要指定 start_time，end_time 和 step，其中 step 会使用该变量的值。
- tidb_metric_query_range_duration：查询监控时，会将 PROMQL 中的 $RANGE_DURATION 替换成该变量的值，默认值是 60 秒。

如果想要查看不同时间粒度的监控项的值，用户可以修改上面两个 session 变量后查询监控表，示例如下：

首先修改两个 session 变量的值，将时间粒度设置为 30 秒。

> **注意：**
>
> Prometheus 支持查询的最小粒度为 30 秒。

```
set @@tidb_metric_query_step=30;
set @@tidb_metric_query_range_duration=30;
```

再查询 tidb_query_duration 监控如下，可以发现在三分钟时间范围内，每个 label 有六个时间的值，每个值时间间隔是 30 秒。

```
select * from metrics_schema.tidb_query_duration where value is not null and time>='2020-03-25
    ↪ 23:40:00' and time <= '2020-03-25 23:42:00' and quantile=0.99;
```

```
+---------------------+-------------------+----------+----------+-----------------+
| time                | instance          | sql_type | quantile | value           |
+---------------------+-------------------+----------+----------+-----------------+
| 2020-03-25 23:40:00 | 172.16.5.40:10089 | Insert   | 0.99     | 0.483285651924  |
| 2020-03-25 23:40:30 | 172.16.5.40:10089 | Insert   | 0.99     | 0.484151462113  |
| 2020-03-25 23:41:00 | 172.16.5.40:10089 | Insert   | 0.99     | 0.504576        |
| 2020-03-25 23:41:30 | 172.16.5.40:10089 | Insert   | 0.99     | 0.493577384561  |
| 2020-03-25 23:42:00 | 172.16.5.40:10089 | Insert   | 0.99     | 0.49482474311   |
| 2020-03-25 23:40:00 | 172.16.5.40:10089 | Select   | 0.99     | 0.189253402185  |
| 2020-03-25 23:40:30 | 172.16.5.40:10089 | Select   | 0.99     | 0.184224951851  |
| 2020-03-25 23:41:00 | 172.16.5.40:10089 | Select   | 0.99     | 0.151673410553  |
| 2020-03-25 23:41:30 | 172.16.5.40:10089 | Select   | 0.99     | 0.127953838989  |
| 2020-03-25 23:42:00 | 172.16.5.40:10089 | Select   | 0.99     | 0.127455434547  |
| 2020-03-25 23:40:00 | 172.16.5.40:10089 | internal | 0.99     | 0.0624          |
| 2020-03-25 23:40:30 | 172.16.5.40:10089 | internal | 0.99     | 0.12416         |
| 2020-03-25 23:41:00 | 172.16.5.40:10089 | internal | 0.99     | 0.0304          |
| 2020-03-25 23:41:30 | 172.16.5.40:10089 | internal | 0.99     | 0.06272         |
| 2020-03-25 23:42:00 | 172.16.5.40:10089 | internal | 0.99     | 0.0629333333333 |
+---------------------+-------------------+----------+----------+-----------------+
```

最后查看执行计划，也会发现执行计划中的 PromQL 以及 step 的值都已经变成了 30 秒。

```
desc select * from metrics_schema.tidb_query_duration where value is not null and time>='
    ↪ 2020-03-25 23:40:00' and time <= '2020-03-25 23:42:00' and quantile=0.99;
```

```
+--
    ↪ ----------------+----------+------+--------------------------+-----------------------------------------------
    ↪
| id                | estRows  | task | access object            | operator info
    ↪
    ↪ |
+--
    ↪ ----------------+----------+------+--------------------------+-----------------------------------------------
    ↪
| Selection_5       | 8000.00  | root |                          | not(isnull(Column#5))
    ↪
    ↪ |
```

```
| └─MemTableScan_6 | 10000.00 | root | table:tidb_query_duration | PromQL:histogram_quantile
    ↪ (0.99, sum(rate(tidb_server_handle_query_duration_seconds_bucket{}[30s])) by (le,sql_type
    ↪ ,instance)), start_time:2020-03-25 23:40:00, end_time:2020-03-25 23:42:00, step:30s |
+--
    ↪ ----------------+----------+------+--------------------------+-------------------------------------------------
    ↪
```

12.12 UI

12.12.1 TiDB Dashboard

12.12.1.1 TiDB Dashboard 介绍

TiDB Dashboard 是 TiDB 自 4.0 版本起提供的图形化界面，可用于监控及诊断 TiDB 集群。TiDB Dashboard 内置于 TiDB 的 PD 组件中，无需独立部署。

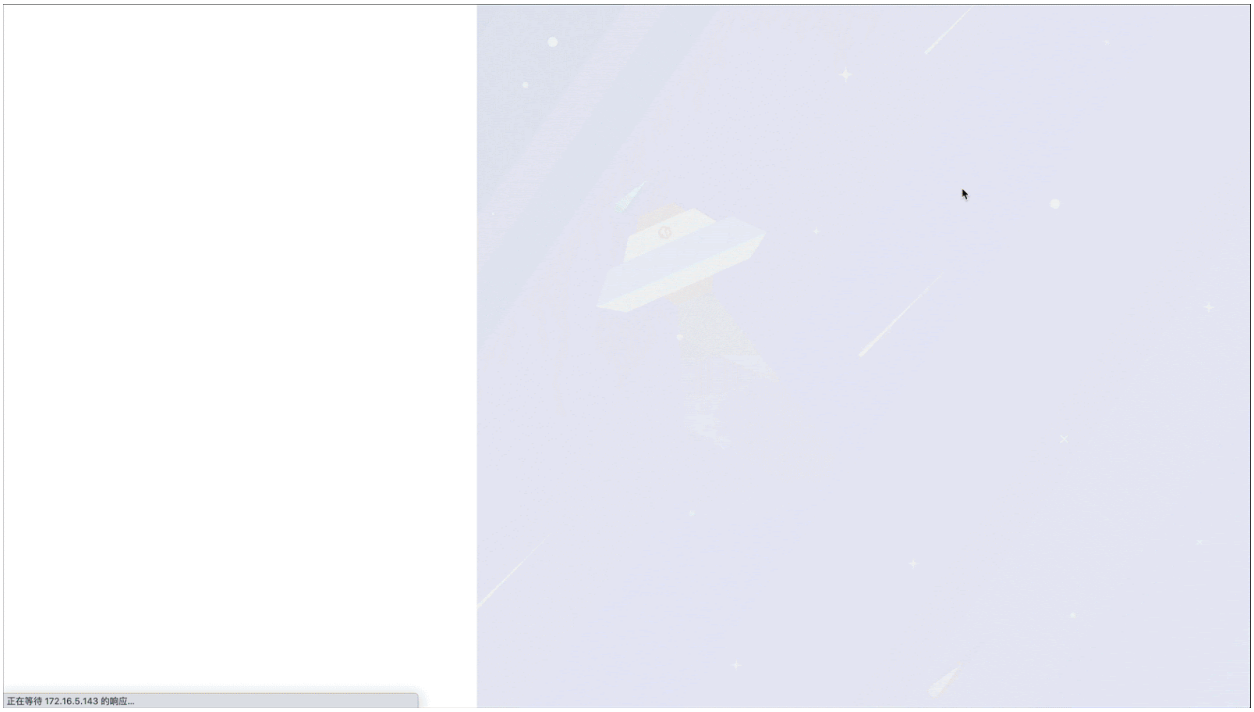


图 372: 界面

TiDB Dashboard 在 GitHub 上开源。

以下列出了 TiDB Dashboard 的主要功能，可分别点击小节内的链接进一步了解详情。

12.12.1.1.1 了解集群整体运行概况

查看集群整体 QPS 数值、执行耗时、消耗资源最多的几类 SQL 语句等概况信息。

参阅概况页面了解详情。

12.12.1.1.2 查看组件及主机运行状态

查看整个集群中 TiDB、TiKV、PD、TiFlash 组件的运行状态及其所在主机的运行状态。

参阅集群信息页面了解详情。

12.12.1.1.3 分析集群读写流量分布及趋势变化

通过热力图形式可视化地展示整个集群中读写流量随时间的变化情况，及时发现业务模式的变化，或定位性能不均衡的热点所在。

参阅流量可视化页面了解详情。

12.12.1.1.4 列出所有 SQL 查询的耗时等执行信息

列出所有 SQL 语句在集群上执行情况，了解各个阶段的执行时间、总运行次数等信息，帮助用户分析和定位集群中最消耗资源的查询，优化整体性能。

参阅SQL 语句分析页面了解详情。

12.12.1.1.5 详细了解耗时较长的 SQL 语句的执行信息

列出所有耗时较长的 SQL 语句文本及其执行信息，帮助用户定位 SQL 语句性能缓慢或发生性能抖动的原因。

参阅慢查询页面了解详情。

12.12.1.1.6 诊断常见集群问题并生成报告

自动判断集群中是否存在一些常见的风险（如配置不一致）或问题，生成报告并给出操作建议，或对比集群在不同时间段的各个指标状态，供用户分析可能存在问题的方向。

参阅集群诊断页面了解详情。

12.12.1.1.7 查询所有组件日志

按关键字、时间范围等条件快速搜索集群中所有运行实例的日志，并可打包下载到本地。

参阅日志搜索页面了解详情。

12.12.1.1.8 收集分析各个组件的性能数据

高级调试功能：无需第三方工具，在线地对各个组件进行性能分析，剖析组件实例在分析时间段内执行的各种内部操作及比例。

参阅实例性能分析页面了解详情。

> 注意：
>
> TiDB Dashboard 默认会收集使用情况信息，并将这些信息分享给 PingCAP 用于改善产品。若要了解所收集的信息详情及如何禁用该行为，请参见遥测。

12.12.1.2 运维

12.12.1.2.1 部署 TiDB Dashboard

TiDB Dashboard 界面内置于 TiDB 4.0 或更高版本的 PD 组件中，无需额外部署。只需部署标准 TiDB 集群，TiDB Dashboard 就会原生集成。

请参阅下列文档了解如何部署标准 TiDB 集群：

- 快速试用 TiDB 集群
- 生产环境部署
- Kubernetes 环境部署

> **注意：**
>
> TiDB Dashboard 目前不能在低于 4.0 版本的集群中部署或使用。

多 PD 实例部署

当集群中部署了多个 PD 实例时，其中仅有一个 PD 实例会固定地提供 TiDB Dashboard 服务。

各个 PD 首次运行时会自动协商出其中某一个实例提供 TiDB Dashboard 服务。协商完毕后，无论重启或扩容，都会固定在这个实例上运行 TiDB Dashboard 服务，除非该实例被手动缩容。其他 PD 实例不会运行 TiDB Dashboard 服务。这个协商过程无需用户介入，会自动完成。

当用户访问不提供 TiDB Dashboard 服务的 PD 实例时，浏览器将会收到重定向指令，自动引导用户重新访问提供了 TiDB Dashboard 服务的 PD 实例，从而能正常使用。流程如下图所示。

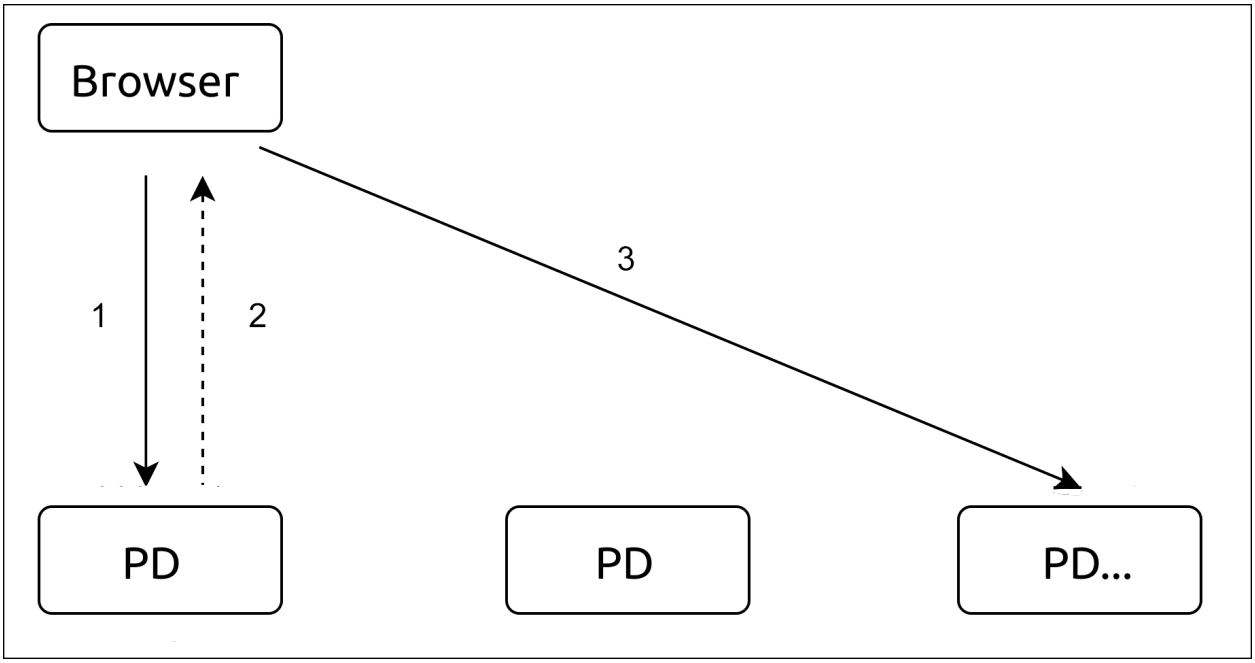


图 373: 流程示意

> **注意：**
>
> 提供 TiDB Dashboard 服务的 PD 实例不一定与 PD leader 一致。

查询实际运行 TiDB Dashboard 服务的 PD 实例

使用 TiUP 部署时，对于已启动的集群，可通过 `tiup cluster display` 命令查看哪个 PD 节点提供了 TiDB Dashboard 服务（将 CLUSTER_NAME 替换为集群名称）：

```
tiup cluster display CLUSTER_NAME --dashboard
```

输出样例如下：

```
http://192.168.0.123:2379/dashboard/
```

> **注意：**
>
> 该功能在 TiUP Cluster v1.0.3 或更高版本部署工具中提供。
>
> 升级 TiUP Cluster 步骤
>
> ```
> tiup update --self
> tiup update cluster --force
> ```

切换其他 PD 实例提供 TiDB Dashboard 服务

使用 TiUP 部署时，对于已启动的集群，可使用 `tiup ctl:<cluster-version> pd` 命令切换其他 PD 实例运行 TiDB Dashboard，或在禁用 TiDB Dashboard 的情况下重新指定一个 PD 实例运行 TiDB Dashboard：

```
tiup ctl:<cluster-version> pd -u http://127.0.0.1:2379 config set dashboard-address http
    ↪ ://9.9.9.9:2379
```

其中：

- 将 `127.0.0.1:2379` 替换为任意 PD 实例的 IP 和端口
- 将 `9.9.9.9:2379` 替换为想运行 TiDB Dashboard 服务的新 PD 实例的 IP 和端口

修改完毕后，可使用 `tiup cluster display` 命令确认修改是否生效（将 CLUSTER_NAME 替换为集群名称）：

```
tiup cluster display CLUSTER_NAME --dashboard
```

> 警告：
>
> 切换 TiDB Dashboard 将会丢失之前 TiDB Dashboard 实例所存储的本地数据，包括流量可视化历史、历史搜索记录等。

禁用 TiDB Dashboard

使用 TiUP 部署时，对于已启动的集群，可使用 `tiup ctl:<cluster-version> pd` 命令在所有 PD 实例上禁用 TiDB Dashboard（将 `127.0.0.1:2379` 替换为任意 PD 实例的 IP 和端口）：

```
tiup ctl:<cluster-version> pd -u http://127.0.0.1:2379 config set dashboard-address none
```

禁用 TiDB Dashboard 后，查询哪个 PD 实例提供 TiDB Dashboard 服务将会失败：

```
Error: TiDB Dashboard is disabled
```

浏览器访问任意 PD 实例的 TiDB Dashboard 地址也将提示失败：

```
Dashboard is not started.
```

重新启用 TiDB Dashboard

使用 TiUP 部署时，对于已启动的集群，可使用 `tiup ctl:<cluster-version> pd` 命令，要求 PD 重新协商出某一个实例运行 TiDB Dashboard（将 `127.0.0.1:2379` 替换为任意 PD 实例的 IP 和端口）：

```
tiup ctl:<cluster-version> pd -u http://127.0.0.1:2379 config set dashboard-address auto
```

修改完毕后，使用 `tiup cluster display` 命令查看 PD 自动协商出的 TiDB Dashboard 实例地址（将 CLUSTER_NAME 替换为集群名称）：

```
tiup cluster display CLUSTER_NAME --dashboard
```

还可以通过手动指定哪个 PD 实例运行 TiDB Dashboard 服务的方式重新启用 TiDB Dashboard，具体操作参见上文切换其他 PD 实例提供 TiDB Dashboard 服务。

> **警告：**
>
> 若新启用的 TiDB Dashboard 实例与禁用前的实例不一致，将会丢失之前 TiDB Dashboard 实例所存储的本地数据，包括流量可视化历史、历史搜索记录等。

下一步

- 参阅访问 TiDB Dashboard 章节了解如何访问及登录集群上的 TiDB Dashboard 界面。
- 参阅提高 TiDB Dashboard 安全性 章节了解如何增强 TiDB Dashboard 的安全性，如配置防火墙等。

12.12.1.2.2 通过反向代理使用 TiDB Dashboard

你可以使用反向代理将 TiDB Dashboard 服务安全从内部网络提供给外部网络。

操作步骤

第 1 步：获取实际 TiDB Dashboard 地址

当集群中部署有多个 PD 实例时，其中仅有一个 PD 实例会真正运行 TiDB Dashboard，因此需要确保反向代理的上游 (Upstream) 指向了正确的地址。关于该机制的详情，可参阅TiDB Dashboard 多 PD 实例部署 章节。

使用 TiUP 部署工具时，操作命令如下（将 CLUSTER_NAME 替换为集群名称）：

```
tiup cluster display CLUSTER_NAME --dashboard
```

输出即为实际 TiDB Dashboard 地址。样例如下：

```
http://192.168.0.123:2379/dashboard/
```

> **注意：**
>
> 该功能在 TiUP Cluster v1.0.3 或更高版本部署工具中提供。
>
> 升级 TiUP Cluster 步骤
>
> ```
> tiup update --self
> tiup update cluster --force
> ```

第 2 步：配置反向代理

使用 HAProxy 反向代理

HAProxy 作为反向代理时，方法如下：

1. 以在 8033 端口反向代理 TiDB Dashboard 为例，在 HAProxy 配置文件中，新增如下配置：

“‘haproxy frontend tidb_dashboard_front bind *:8033 use_backend tidb_dashboard_back if { path /dashboard } or { path_beg /dashboard/ }

backend tidb_dashboard_back mode http server tidb_dashboard 192.168.0.123:2379 “‘

其中 192.168.0.123:2379 需替换为第 1 步：获取实际 TiDB Dashboard 地址中取得的 TiDB Dashboard 实际地址中的 IP 及端口部分。

> 警告：
>
> 请务必保留 use_backend 指令中的 if 部分，确保只有该路径下的服务会被反向代理，否则将引入安全风险。参见提高 TiDB Dashboard 安全性。

2. 重启 HAProxy，以使配置生效。
3. 测试反向代理是否生效：访问 HAProxy 所在机器的 8033 端口下 /dashboard/ 地址，如 http://example.com:8033/dashboard/ ，即可访问 TiDB Dashboard。

使用 NGINX 反向代理

NGINX 作为反向代理时，方法如下：

1. 以在 8033 端口反向代理 TiDB Dashboard 为例，在 NGINX 配置文件中，新增如下配置：

```
nginx  server {  listen 8033;  location /dashboard/ { proxy_pass http://192.168.0.123:2379/dashboard
↪ /; } }
```

其中 http://192.168.0.123:2379/dashboard/ 需替换为第 1 步：获取实际 TiDB Dashboard 地址中取得的 TiDB Dashboard 实际地址。

> 警告：
>
> 请务必保留 proxy_pass 指令中的 /dashboard/ 路径，确保只有该路径下的服务会被反向代理，否则将引入安全风险。参见提高 TiDB Dashboard 安全性。

2. 重新载入 NGINX 以使配置生效：

```
shell  sudo nginx -s reload
```

3. 测试反向代理是否生效：访问 NGINX 所在机器的 8033 端口下 /dashboard/ 地址，如 http://example.com
↪ :8033/dashboard/ ，即可访问 TiDB Dashboard。

自定义路径前缀

TiDB Dashboard 默认在 /dashboard/ 路径下提供服务，即使是反向代理也是如此，例如 http://example.com:8033/dashboard/。若要配置反向代理以非默认的路径提供 TiDB Dashboard 服务，例如 http://example.com:8033/foo/ 或 http://example.com:8033/，可参考以下步骤。

第 1 步：修改 PD 配置指定 TiDB Dashboard 服务路径前缀

修改 PD 配置中 [dashboard] 类别的 public-path-prefix 配置项，可指定服务路径前缀。该配置修改后需要重启 PD 实例生效。

以 TiUP 部署且希望运行在 http://example.com:8033/foo/ 为例，可指定以下配置：

```
server_configs:
  pd:
    dashboard.public-path-prefix: /foo
```

使用 TiUP 部署全新集群时修改配置

若要全新部署集群，可在 TiUP 拓扑文件 topology.yaml 中加入上述配置项后进行部署，具体步骤参阅TiUP 部署文档。

使用 TiUP 修改已部署集群的配置

1. 以编辑模式打开该集群的配置文件（将 CLUSTER_NAME 替换为集群名称）

```
shell  tiup cluster edit-config CLUSTER_NAME
```

2. 在 server_configs 的 pd 配置下修改或新增配置项，若没有 server_configs 请在最顶层新增：

```
yaml  server_configs: pd:     dashboard.public-path-prefix: /foo
```

修改完成后的配置文件类似于：

```
yaml  server_configs: pd:     dashboard.public-path-prefix: /foo global: user: tidb ...
```

或

```
yaml  monitored:  ...  server_configs: tidb: ...  tikv: ...  pd:     dashboard.public-path-
↪ prefix: /foo ...
```

3. 滚动重启所有 PD 实例生效配置（将 CLUSTER_NAME 替换为集群名称）

```
shell  tiup cluster reload CLUSTER_NAME -R pd
```

详情请参阅TiUP 常见运维操作 - 修改配置参数。

若希望运行在根路径（如 http://example.com:8033/）下，相应的配置为：

```
server_configs:
  pd:
    dashboard.public-path-prefix: /
```

> **警告：**
>
> 修改自定义路径前缀生效后，直接访问将不能正常使用 TiDB Dashboard，您只能通过和路径前缀匹配的反向代理访问。

第 2 步：修改反向代理配置

使用 HAProxy 反向代理

以 `http://example.com:8033/foo/` 为例，HAProxy 配置如下：

```
frontend tidb_dashboard_front
  bind *:8033
  use_backend tidb_dashboard_back if { path /foo } or { path_beg /foo/ }

backend tidb_dashboard_back
  mode http
  http-request set-path %[path,regsub(^/foo/?,/dashboard/)]
  server tidb_dashboard 192.168.0.123:2379
```

其中 `192.168.0.123:2379` 需替换为第 1 步：获取实际 TiDB Dashboard 地址中取得的 TiDB Dashboard 实际地址中的 IP 及端口部分。

> **警告：**
>
> 请务必保留 `use_backend` 指令中的 `if` 部分，确保只有该路径下的服务会被反向代理，否则将引入安全风险。参见提高 TiDB Dashboard 安全性。

若希望运行在根路径（如 `http://example.com:8033/`），HAProxy 配置如下：

```
frontend tidb_dashboard_front
  bind *:8033
  use_backend tidb_dashboard_back

backend tidb_dashboard_back
  mode http
  http-request set-path /dashboard%[path]
  server tidb_dashboard 192.168.0.123:2379
```

修改配置并重启 HAProxy 后即可生效。

使用 NGINX 反向代理

以 `http://example.com:8033/foo/` 为例，相应的 NGINX 配置为：

```
server {
  listen 8033;
  location /foo/ {
    proxy_pass http://192.168.0.123:2379/dashboard/;
  }
}
```

其中 `http://192.168.0.123:2379/dashboard/` 需替换为第 1 步：获取实际 TiDB Dashboard 地址中取得的 TiDB Dashboard 实际地址。

> **警告：**
>
> 请务必保留 `proxy_pass` 指令中的 `/dashboard/` 路径，确保只有该路径下的服务会被反向代理，否则将引入安全风险。参见提高 TiDB Dashboard 安全性。

若希望运行在根路径（如 `http://example.com:8033/`），NGINX 配置为：

```
server {
  listen 8033;
  location / {
    proxy_pass http://192.168.0.123:2379/dashboard/;
  }
}
```

修改配置并重启 NGINX 后即可生效：

```
sudo nginx -s reload
```

下一步

参阅提高 TiDB Dashboard 安全性文档了解如何增强 TiDB Dashboard 的安全性，如配置防火墙等。

12.12.1.2.3 提高 TiDB Dashboard 安全性

尽管访问 TiDB Dashboard 需要登录，但它被设计为默认供可信的用户实体访问。当你希望将 TiDB Dashboard 提供给外部网络用户或不可信用户访问时，需要注意采取以下措施以避免安全漏洞。

为 TiDB `root` 用户设置强密码

TiDB Dashboard 的账号体系与 TiDB SQL 用户一致。默认部署情况下，TiDB 的 `root` 用户没有密码，因而访问 TiDB Dashboard 也不需要密码验证。这将会给恶意访问者极大的集群权限，包括执行特权 SQL 等。

建议的措施：

- 为 TiDB `root` 用户设置一个强密码。请参见TiDB 用户账户管理 了解详情。

使用防火墙阻止不可信访问

TiDB Dashboard 通过 PD Client 端口提供服务，默认为 http://IP:2379/dashboard/。尽管 TiDB Dashboard 需要验证身份，但 PD Client 端口上承载的其他 PD 内部特权接口不需要验证身份，且能进行特权操作，例如 http://IP:2379/pd/api/v1/members。因此，将 PD Client 端口直接暴露给外部网络具有极大的风险。

建议的措施：

1. 使用防火墙禁止组件外部网络或不可信网络访问任何 PD 组件的 Client 端口。

注意，TiDB、TiKV 等组件需要通过 PD Client 端口与 PD 组件进行通信，因此请勿对组件内部网络阻止访问，这将导致集群不可用。

2. 参见通过反向代理使用 TiDB Dashboard 了解如何配置反向代理将 TiDB Dashboard 服务在另一个端口上安全地提供给外部网络。

如何在多 PD 实例部署时开放 TiDB Dashboard 端口访问

> 警告：
>
> 本章节描述了一个不安全的访问方案，仅供测试环境使用。在生产环境中，不要使用本方案。请参见本文的其余章节。

在测试环境中，您可能需要配置防火墙开放 TiDB Dashboard 端口供外部访问。

当部署了多个 PD 实例时，其中仅有一个 PD 实例会真正运行 TiDB Dashboard，访问其他 PD 实例时会发生浏览器重定向，因此需要确保防火墙配置了正确的 IP 地址。关于该机制的详情，可参阅TiDB Dashboard 多 PD 实例部署 章节。

使用 TiUP 部署工具时，可使用以下命令查看实际运行 TiDB Dashboard 的 PD 实例地址（将 CLUSTER_NAME 替换为集群名称）：

```
tiup cluster display CLUSTER_NAME --dashboard
```

输出即为实际 TiDB Dashboard 地址。

> 注意：
>
> 该功能在 TiUP Cluster v1.0.3 或更高版本部署工具中提供。
>
> 升级 TiUP Cluster 步骤
>
> ```
> tiup update --self
> tiup update cluster --force
> ```

以下是一个样例输出：

```
http://192.168.0.123:2379/dashboard/
```

在这个样例中，需要为防火墙配置开放 IP 192.168.0.123 的 2379 端口入站访问，并通过 http://192.168.0.123:2379/dashboard/ 访问 TiDB Dashboard。

反向代理仅代理 TiDB Dashboard

如前文所述，PD Client 端口下提供的服务不仅有 TiDB Dashboard（位于 http://IP:2379/dashboard/），还有其他 PD 内部特权接口（如 http://IP:2379/pd/api/v1/members）。因此，使用反向代理将 TiDB Dashboard 提供给外部网络时，应当确保仅提供 /dashboard 前缀下的服务，而非该端口下所有服务，避免外部网络能通过反向代理访问到 PD 内部特权接口。

建议的措施：

- 参见通过反向代理使用 TiDB Dashboard 了解安全且推荐的反向代理配置。

为反向代理开启 TLS

为了进一步加强传输层安全性，可以为反向代理开启 TLS，甚至可以引入 mTLS 实现访问用户的证书验证。

请参阅 NGINX 文档或 HAProxy 文档了解如何为它们开启 TLS。

其他建议的安全措施

- 为 TiDB 组件间通信开启加密传输
- 为 TiDB 客户端服务端间通信开启加密传输

12.12.1.3 访问 TiDB Dashboard

通过浏览器访问 http://127.0.0.1:2379/dashboard/（将 127.0.0.1:2379 替换为实际 PD 实例的地址和端口）即可打开 TiDB Dashboard。

12.12.1.3.1 多 PD 实例访问

当集群中部署有多个 PD 实例、且您可以直接访问到每个 PD 实例地址和端口时，可以简单地将 http://127.0.0.1:2379/dashboard/ 地址中的 127.0.0.1:2379 替换为集群中任意一个 PD 实例的地址和端口进行访问。

> 注意：
>
> 当处于防火墙或反向代理等环境下、无法直接访问每个 PD 实例时，可能会无法访问 TiDB Dashboard。这通常是防火墙或反向代理没有正确配置导致的。可参阅通过反向代理使用 TiDB Dashboard 或提高 TiDB Dashboard 安全性章节了解如何在多 PD 实例情况下正确配置防火墙或反向代理规则。

12.12.1.3.2 浏览器兼容性

TiDB Dashboard 可在常见的、更新及时的桌面浏览器中使用，具体版本号为：

- Chrome >= 77
- Firefox >= 68
- Edge >= 17

> **注意：**
>
> 若使用旧版本浏览器或其他浏览器访问 TiDB Dashboard，部分界面可能不能正常工作。

12.12.1.3.3 登录

首次访问 TiDB Dashboard 将会显示用户登录界面，如下图所示，可使用 TiDB root 账号登录。

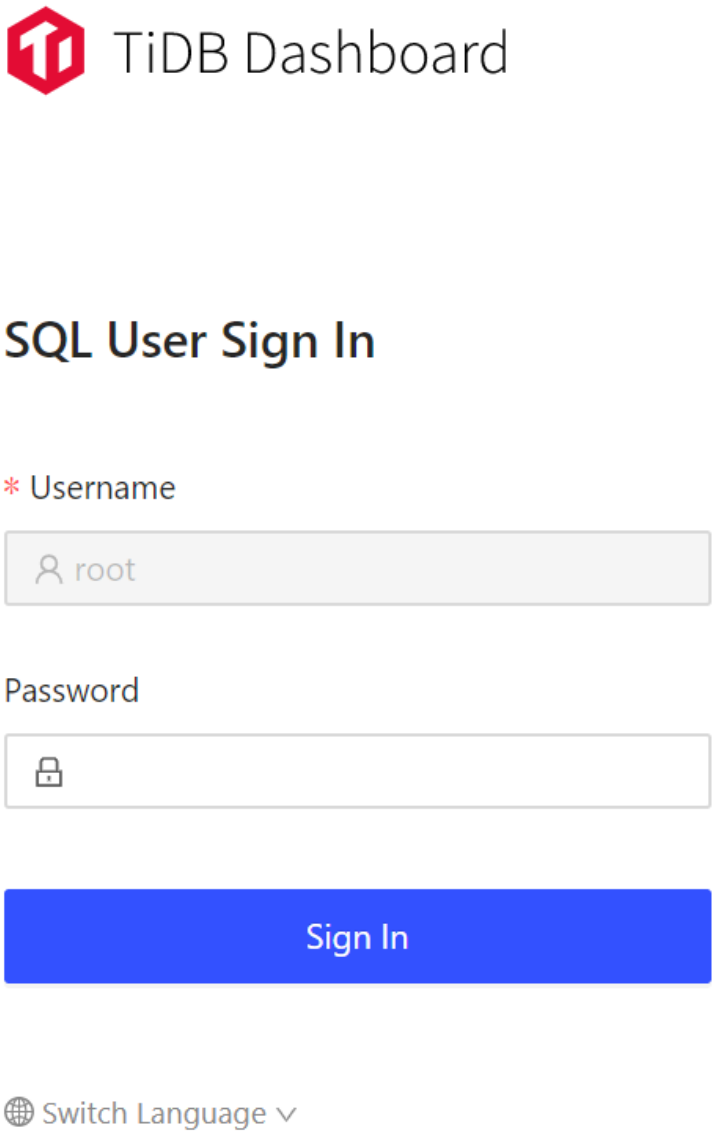


图 374: 登录界面

如果存在以下情况，则可能会登录失败：

- TiDB root 用户不存在
- PD 未启动或无法访问
- TiDB 未启动或无法访问
- root 密码错误

登录后，24 小时内将保持自动登录状态。参见登出章节了解如何登出用户。

12.12.1.3.4 切换语言

TiDB Dashboard 目前支持以下语言：

- 简体中文
- 英文

在登录界面中，可点击 Switch Language 下拉框切换界面显示语言：

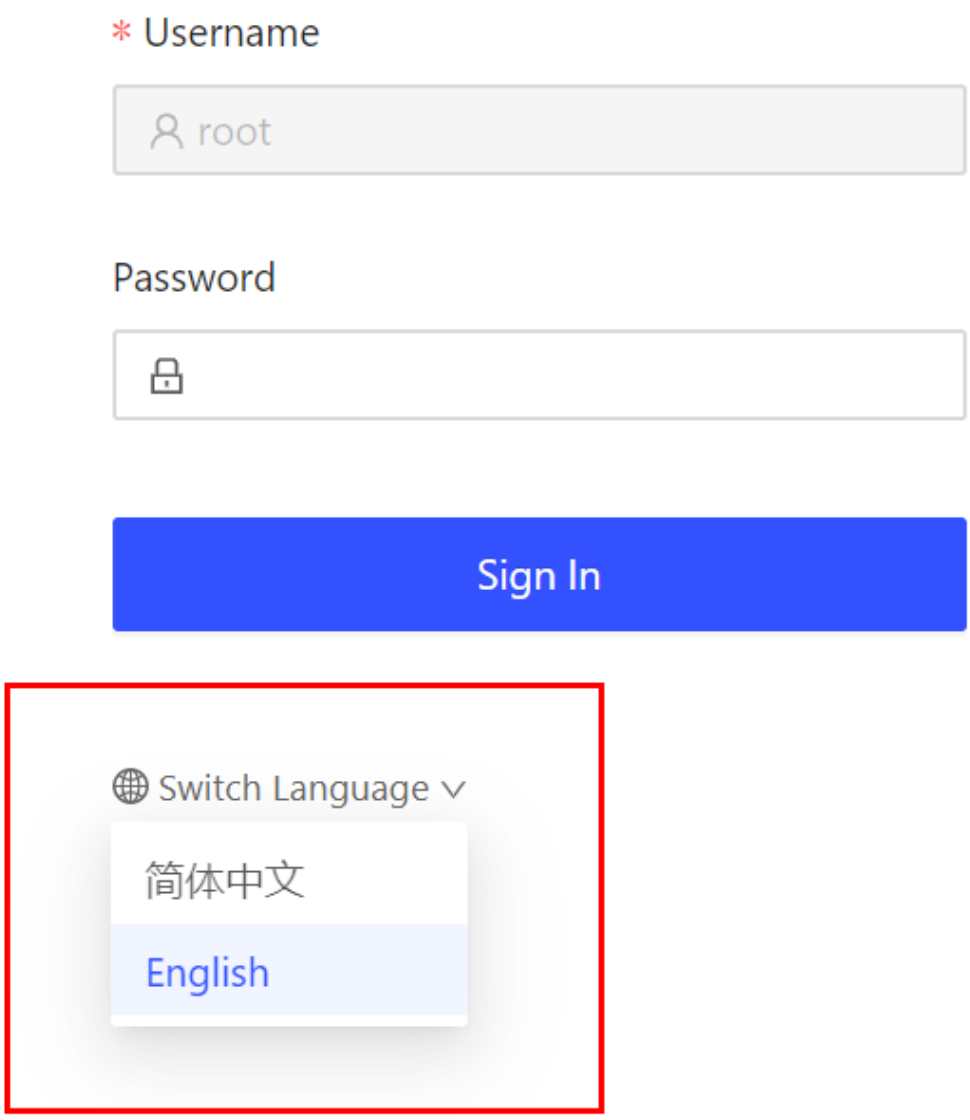


图 375: 切换语言

12.12.1.3.5 登出

登录后，在左侧导航处点击登录用户名，可切换到用户页面。在用户页面点击 登出（Logout）按钮即可登出当前用户。登出后，需重新输入用户名密码。

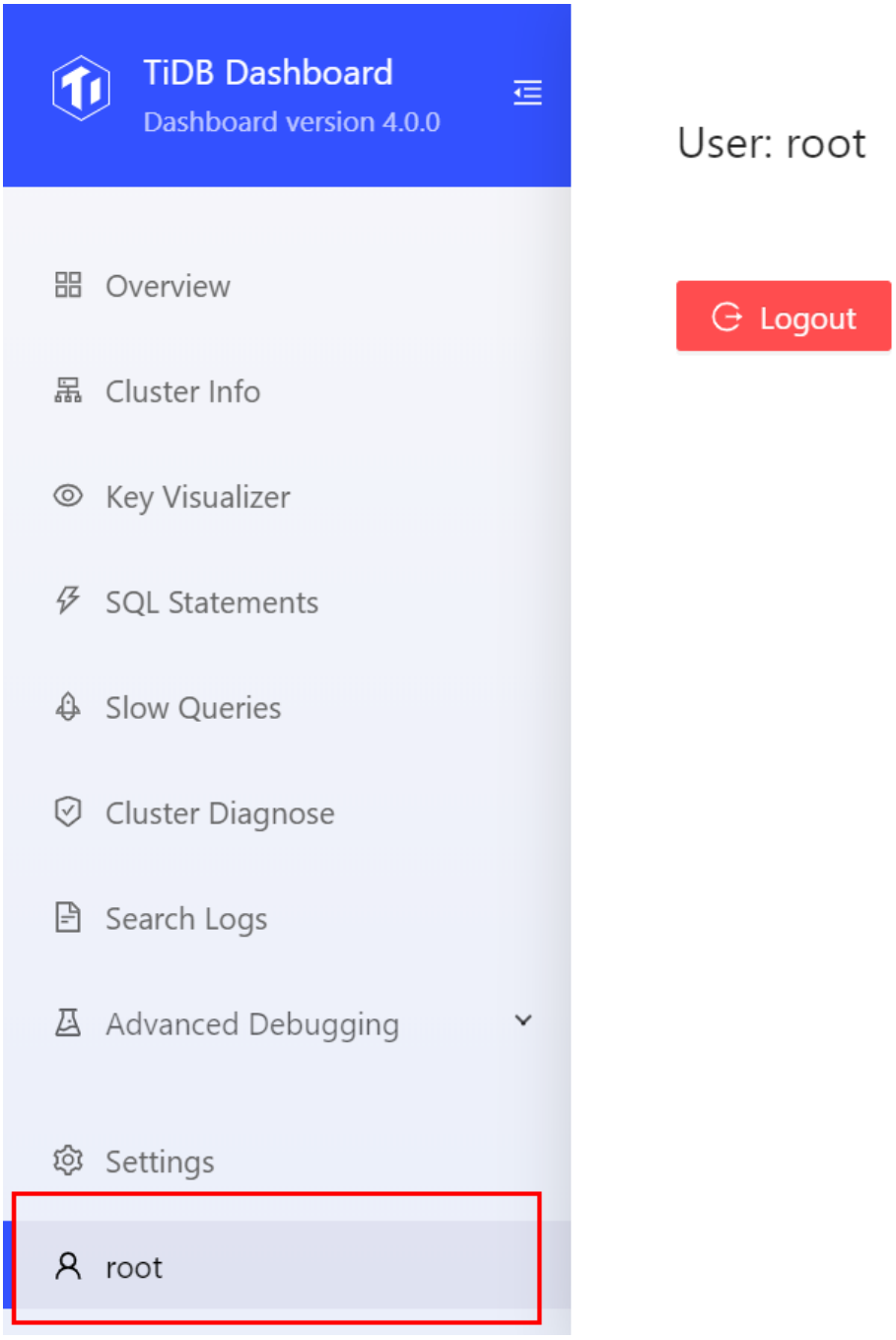


图 376: 登出

12.12.1.4 TiDB Dashboard 概况页面

该页面显示了整个集群的概况，包含以下信息：

- 整个集群的 QPS
- 整个集群的查询延迟

- 最近一段时间内累计耗时最多的若干 SQL 语句
- 最近一段时间内运行时间超过一定阈值的慢查询
- 各个实例的节点数和状态
- 监控及告警信息

12.12.1.4.1 访问

登录 TiDB Dashboard 后默认进入该页面，也可以左侧导航条点击概况（Overview）进入：

图 377: 访问

12.12.1.4.2 QPS

该区域显示最近一小时整个集群的每秒成功和失败查询数量：

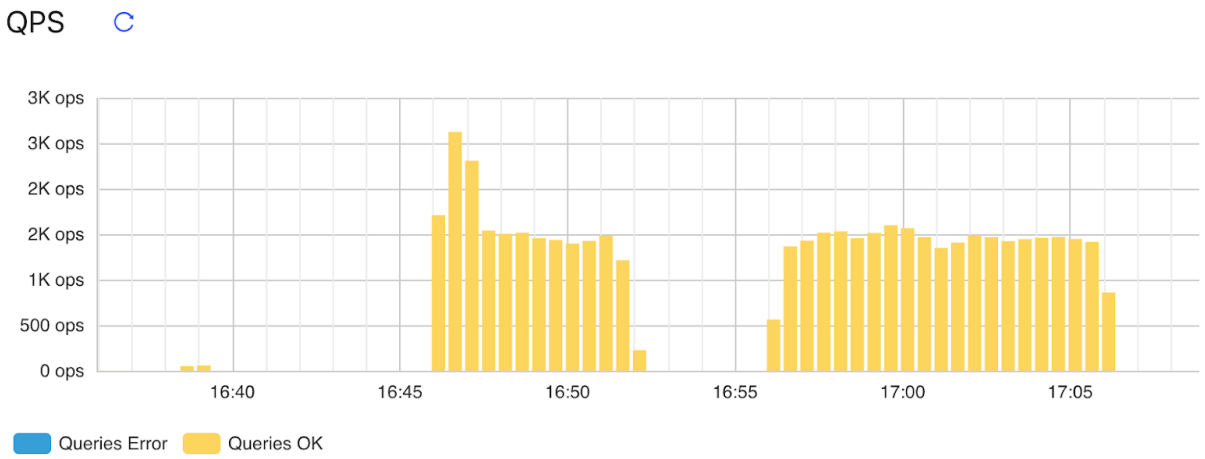


图 378: 界面

> **注意：**
>
> 该功能仅在部署了 Prometheus 监控组件的集群上可用，未部署监控组件的情况下会显示为失败。

12.12.1.4.3 延迟

该区域显示最近一小时整个集群中 99.9%、99% 和 90% 查询的延迟：

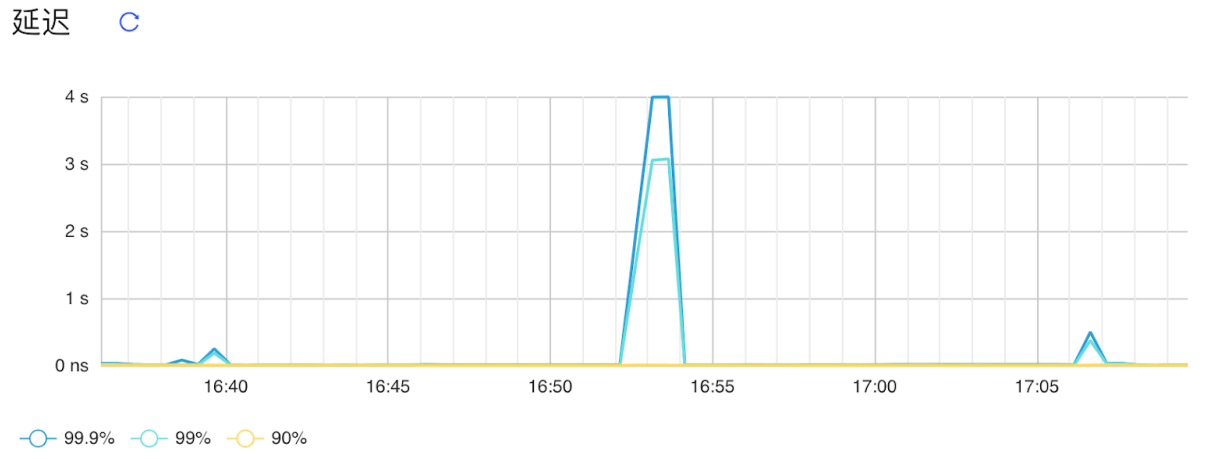


图 379: 界面

> **注意：**
>
> 该功能仅在部署了 Prometheus 监控组件的集群上可用，未部署监控组件的情况下会显示为失败。

12.12.1.4.4 Top SQL 语句

该区域显示最近一段时间内整个群集中累计耗时最长的 10 类 SQL 语句。查询参数不一样但结构一样的 SQL 会归为同一类 SQL 语句，在同一行中显示：

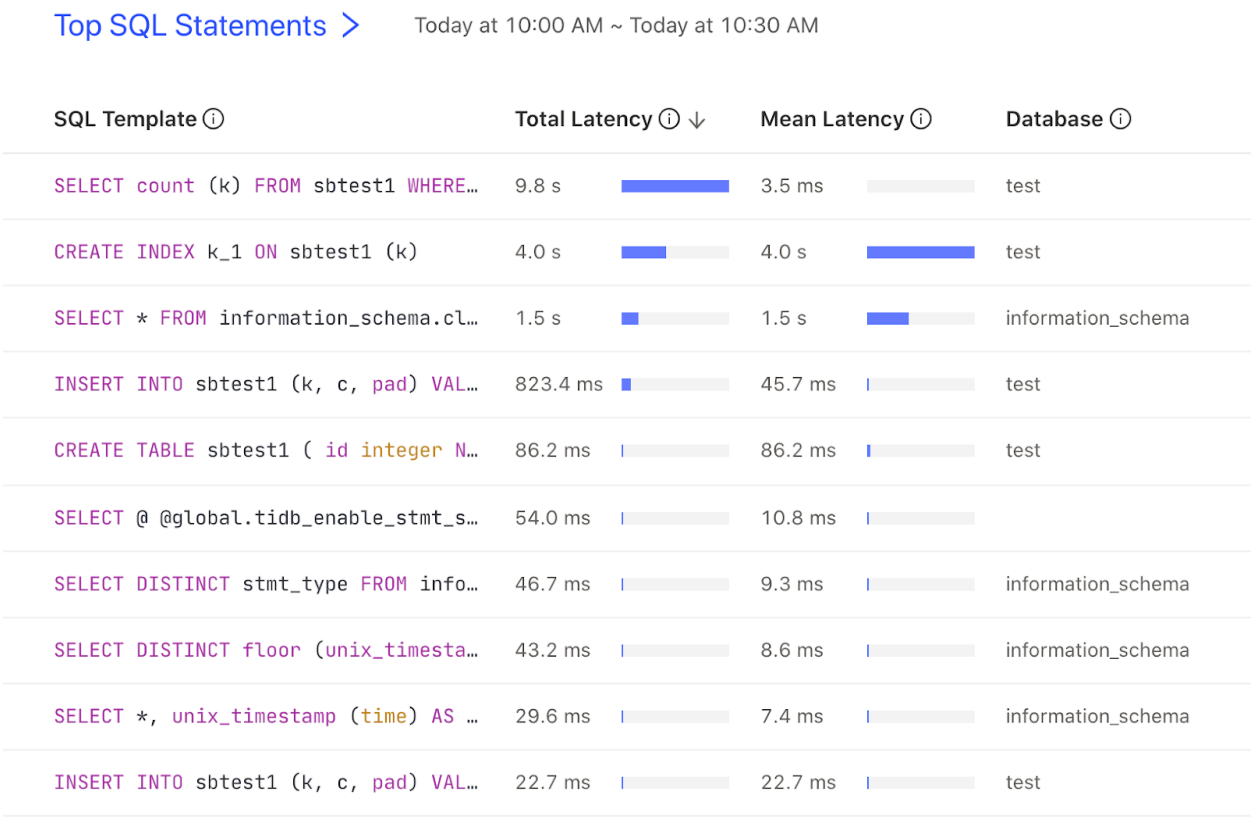


图 380: 界面

该区域显示的内容与SQL 语句分析页面一致，可点击 Top SQL 语句 (Top SQL Statements) 标题查看完整列表。关于该表格中各列详情，见SQL 语句分析页面。

> **注意：**
>
> 该功能仅在开启了 SQL 语句分析功能的集群上可用。

12.12.1.4.5 最近的慢查询

该区域默认显示最近 30 分钟内整个集群中最新的 10 条慢查询：

图 381: 界面

默认情况下运行时间超过 300ms 的 SQL 查询即会被计为慢查询并显示在该表格中。可通过调整tidb_slow_log_threshold 变量或 TiDB slow-threshold 参数调整阈值。

该区域显示的内容与慢查询页面一致，可点击最近的慢查询 (Recent Slow Queries) 标题查看完整列表。关于该表格中各列详情，见慢查询页面。

> 注意：
>
> 该功能仅在配置开启了慢查询日志的集群中可用，使用 TiUP 或 Ansible 部署的集群默认开启慢查询日志。

12.12.1.4.6 实例

该区域汇总显示了整个集群中 TiDB、 TiKV、 PD、 TiFlash 的总实例数量及异常实例数量：

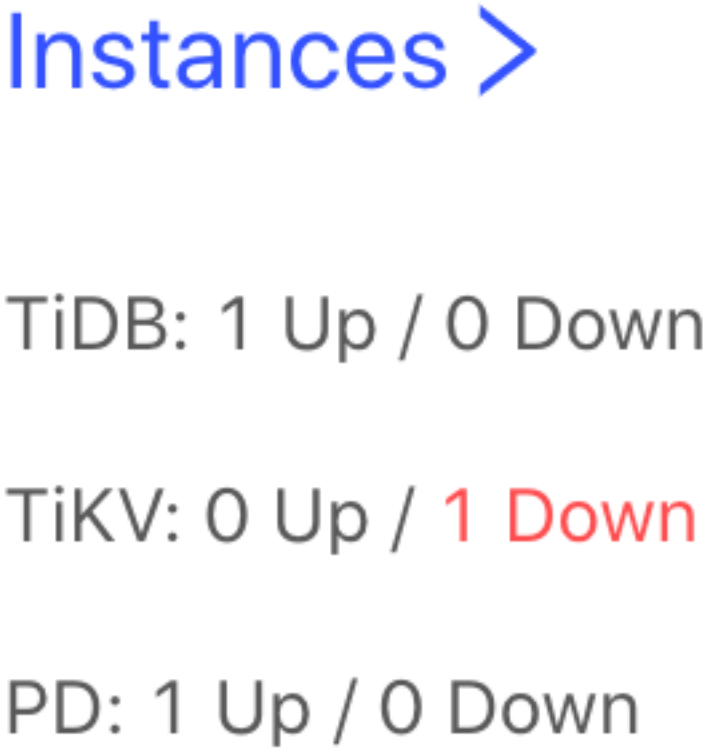


图 382: 界面

状态描述如下：

- Up：实例运行正常（含下线中的存储实例）。
- Down：实例运行异常，例如网络无法连接、进程已崩溃等。

点击实例标题可进入集群信息页面查看各个实例的详细运行状态。

12.12.1.4.7 监控和告警

该区域提供了便捷的链接方便用户查看详细监控或告警：

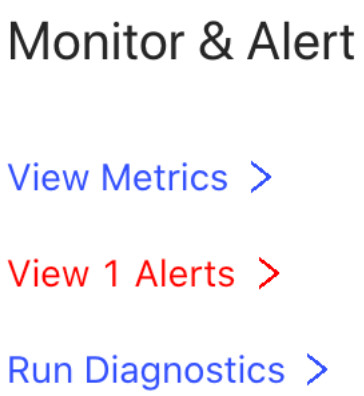


图 383: 界面

- 查看监控链接：点击后跳转至 Grafana 页面，可查看集群详细监控信息。关于 Grafana 监控面板中各个详细监控指标的解释，参见监控指标文档。
- 查看告警链接：点击后跳转至 AlertManager 页面，可查看集群详细告警信息。当集群中已有告警时，告警数量将会直接显示在链接文本上。
- 运行诊断链接：点击后跳转至集群诊断页面，参见集群诊断页面了解详情。

> 注意：
>
> 查看监控链接仅在集群中部署了 Grafana 节点时可用，查看告警链接仅在集群中部署了 AlertManager 节点时可用。

12.12.1.5 TiDB Dashboard 集群信息页面

该页面上允许用户查看整个集群中 TiDB、 TiKV、 PD、 TiFlash 组件的运行状态及其所在主机的运行状态。

12.12.1.5.1 访问

可以通过以下两种方法访问集群信息页面：

- 登录后，左侧导航条点击集群信息：

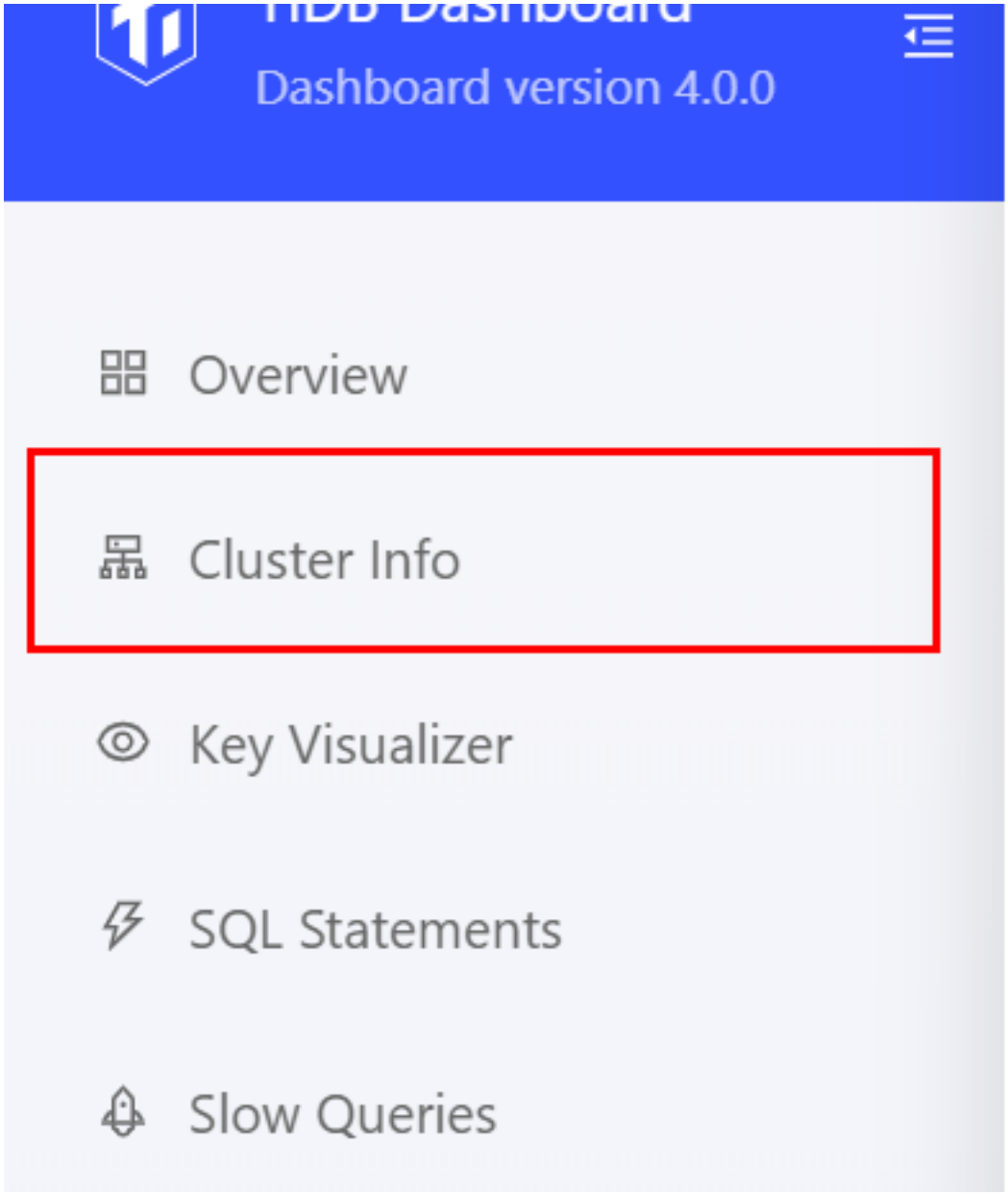


图 384: 访问

- 在浏览器中访问 http://127.0.0.1:2379/dashboard/#/cluster_info/instance（将 127.0.0.1:2379 替换为实际 PD 实例地址和端口）。

12.12.1.5.2 实例列表

点击实例可查看实例列表：

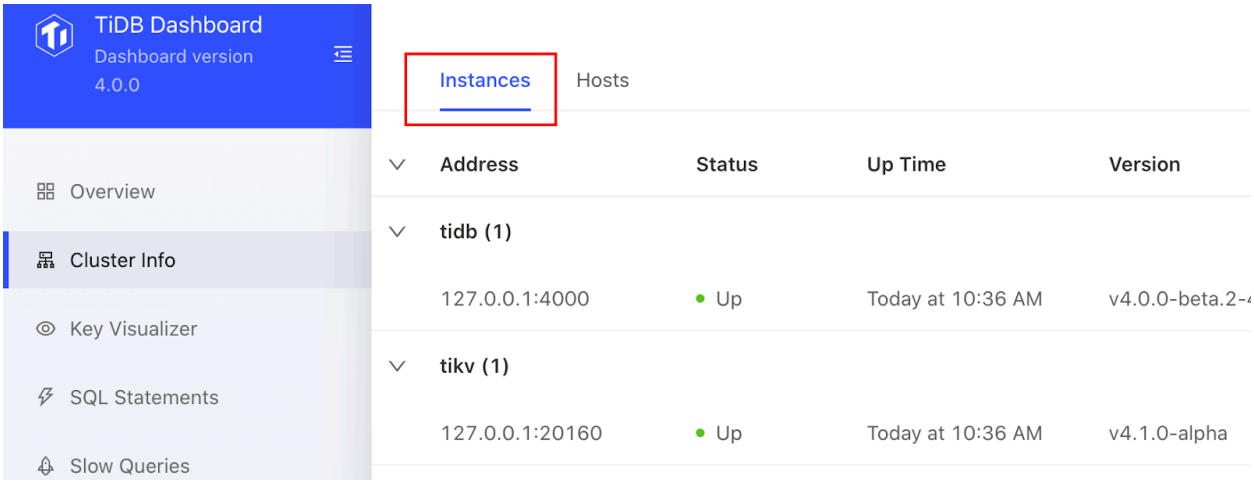


图 385: 实例

实例列表列出了该集群中 TiDB、 TiKV、 PD 和 TiFlash 组件所有实例的概况信息。

表格包含以下列：

- 地址：实例地址
- 状态：实例的运行状态
- 启动时间：实例的启动时间
- 版本：实例版本号
- 部署路径：实例二进制文件所在目录路径
- Git 哈希值：实例二进制对应的 Git 哈希值

实例的运行状态有：

- 在线 (Up)：实例正常运行。
- 离线 (Down) 或无法访问 (Unreachable)：实例未启动或对应主机存在网络问题。
- 已缩容下线 (Tombstone)：实例上的数据已被完整迁出并缩容完毕。仅 TiKV 或 TiFlash 实例存在该状态。
- 下线中 (Offline)：实例上的数据正在被迁出并缩容。仅 TiKV 或 TiFlash 实例存在该状态。
- 未知 (Unknown)：未知的实例运行状态。

> 注意：
>
> 表格中部分列仅在实例处于在线状态时能显示。

12.12.1.5.3 主机列表

点击主机可查看主机列表：

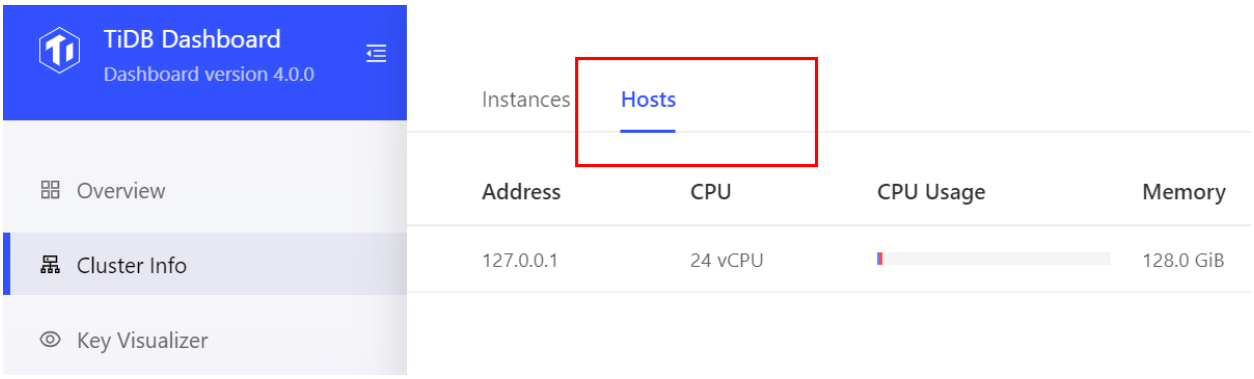


图 386: 主机

主机列表列出了该集群中 TiDB、 TiKV、 PD 和 TiFlash 组件所有实例对应主机的运行情况。

表格包含以下列：

- 地址：主机 IP 地址
- CPU：主机 CPU 逻辑核心数
- CPU 使用率：主机当前 1 秒的用户态和内核态 CPU 使用率
- 物理内存：主机总计的物理内存大小
- 内存使用率：主机当前内存使用率
- 部署磁盘：主机上运行实例所在磁盘的文件系统和磁盘挂载路径
- 磁盘使用率：主机上运行实例所在磁盘的空间使用率

> 注意：
>
> 主机列表信息由各个实例进程给出，因此当主机上所有实例都处于离线状态时，该主机信息将无法显示。

12.12.1.6 TiDB Dashboard 流量可视化页面

流量可视化页面 (Key Visualizer) 可用于分析 TiDB 集群的使用模式和排查流量热点。该页面可视化地呈现了 TiDB 集群一段时间的流量情况。

12.12.1.6.1 访问页面

可以通过以下两种方法访问 Key Visualizer 流量可视化页面：

- 登录 TiDB Dashboard 后，点击左侧导航条的 Key Visualizer（流量可视化）：

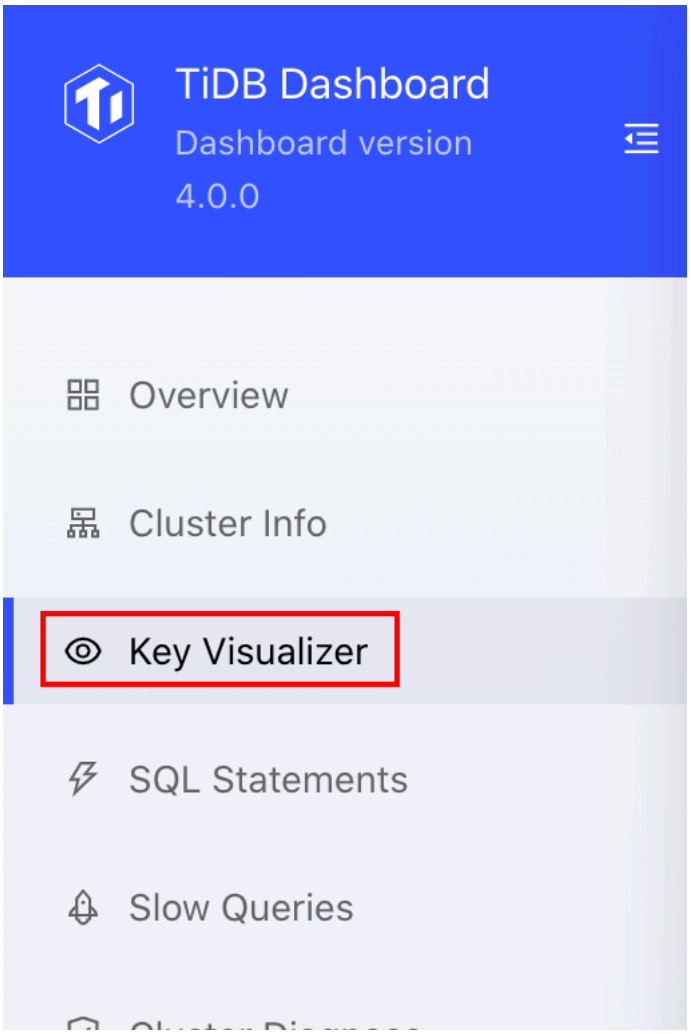


图 387: 访问

- 在浏览器中访问 `http://127.0.0.1:2379/dashboard/#/keyviz`（将 `127.0.0.1:2379` 替换为实际 PD 实例地址和端口）。

12.12.1.6.2 界面示例

流量可视化页面示例如下：

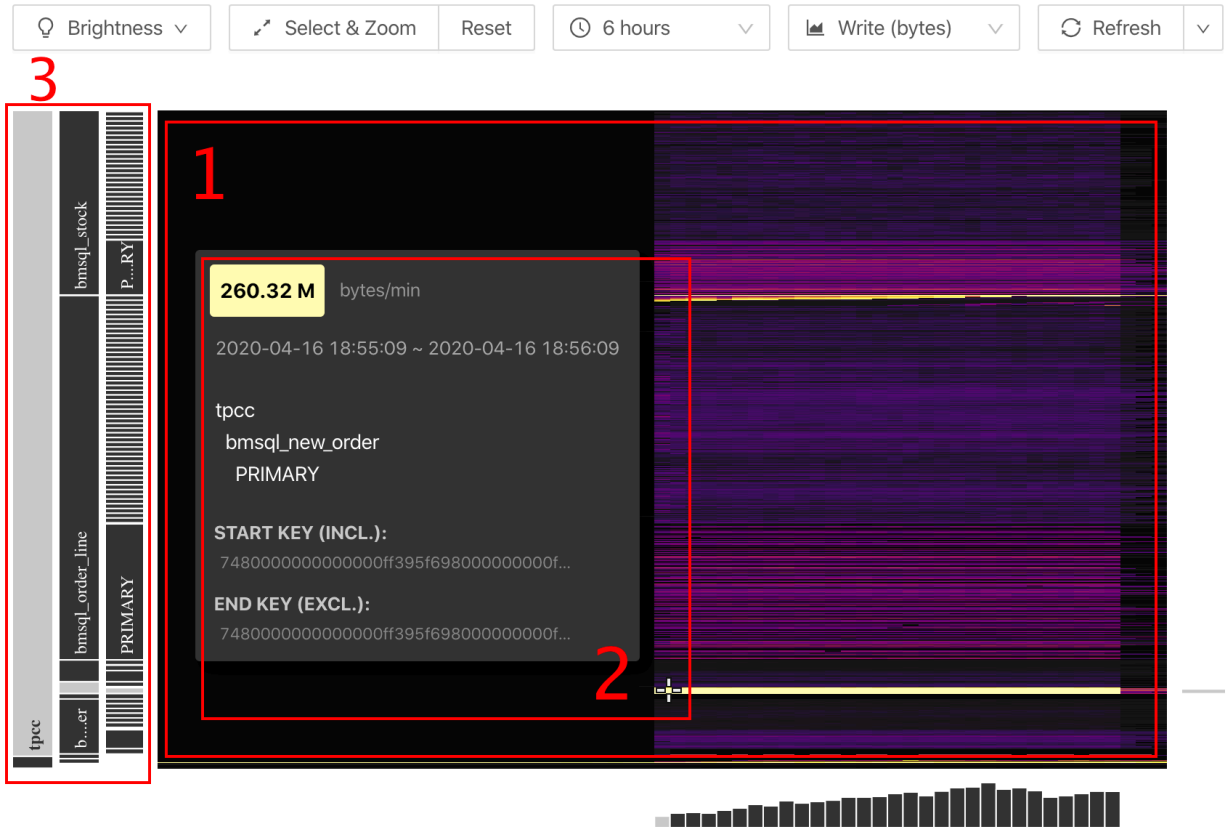


图 388: Key Visualizer 示例图

从以上流量可视化界面可以观察到以下信息：

- 一个大型热力图，显示整体访问流量随时间的变化情况。
- 热力图某个坐标的详细信息。
- 左侧为表、索引等标识信息。

12.12.1.6.3 基本概念

本节介绍流量可视化涉及的一些基本概念。

Region

在 TiDB 集群中，数据以分布式的方式存储在所有的 TiKV 实例中。TiKV 在逻辑上是一个巨大且有序的 KV Map。整个 Key-Value 空间分成很多 Region，每一个 Region 是一系列连续的 Key。

> 注意：
>
> 关于 Region 的详细介绍，请参考三篇文章了解 TiDB 技术内幕 - 说存储

热点

在使用 TiDB 的过程中，热点是一个典型的现象，它表现为大量的流量都在读写一小块数据。由于连续的数据往往由同一个 TiKV 实例处理，因此热点对应的 TiKV 实例的性能就成为了整个业务的性能瓶颈。常见的热点场景有使用自增主键连续写入相邻数据导致的写入表数据热点、时间索引下写入相邻时间数据导致的写入表索引热点等。

> 注意：
>
> 热点问题详情请参阅 TiDB 热点问题详解。

热力图

热力图是流量可视化页面的核心，它显示了一个指标随时间的变化。热力图的横轴 X 是时间，纵轴 Y 则是按 Key 排序的连续 Region，横跨 TiDB 集群上所有数据库和数据表。颜色越暗 (cold) 表示该区域的 Region 在这个时间段上读写流量较低，颜色越亮 (hot) 表示读写流量越高，即越热。

Region 压缩

一个 TiDB 集群中，Region 的数量可能多达数十万。在屏幕上是难以显示这么多 Region 信息的。因此，在一张热力图中，Region 会被压缩到约 1500 个连续范围，每个范围称为 Bucket。在一个热力图上，热的实例更需要关注，因此 Region 压缩总是倾向于将流量较小的大量 Region 压缩为一个 Bucket，而尽量让高流量的 Region 独立成 Bucket。

12.12.1.6.4 使用介绍

本节介绍如何使用流量可视化页面。

设置

首次使用流量可视化页面需要先通过设置页面手动开启此功能。参考页面指引，点击 Open Settings（打开设置）即可打开设置页面：

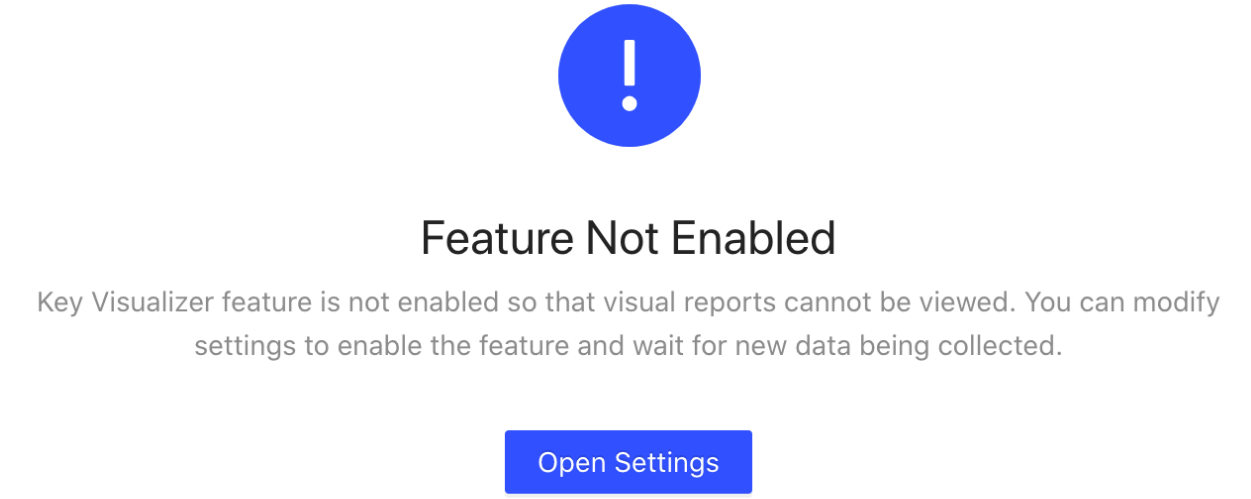


图 389: 功能未开启

在功能已经开启时，可通过右上角的设置图标打开设置页面：

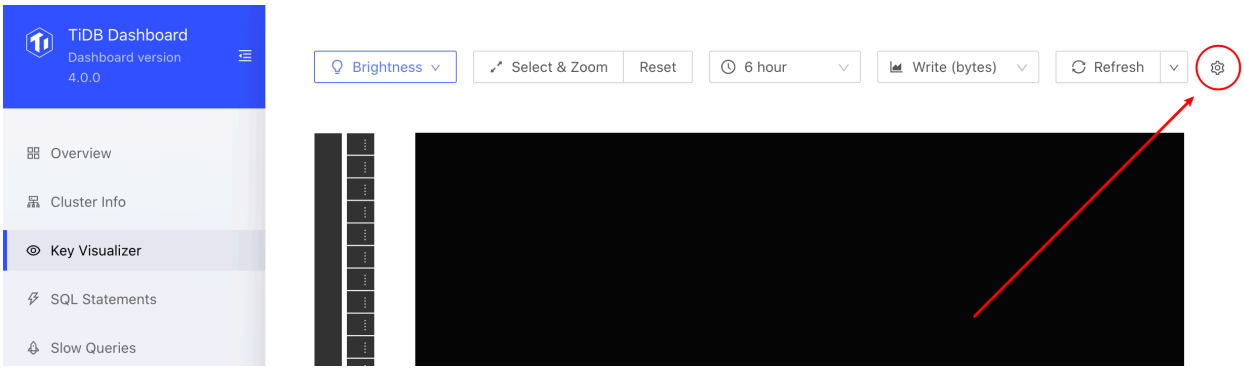


图 390: 设置按钮

设置页面如下图所示：

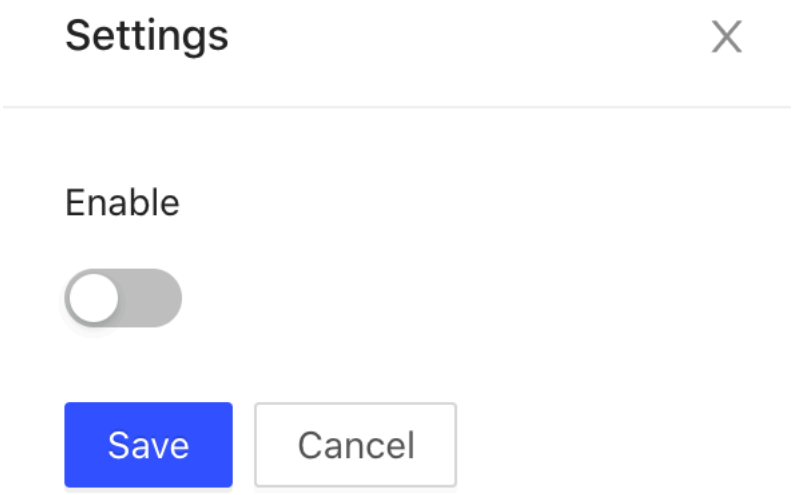


图 391: 设置页面

通过开关设定好是否开启收集，并点击 Save（保存）后生效。开启后，在界面上观察到工具栏已经可以使用：

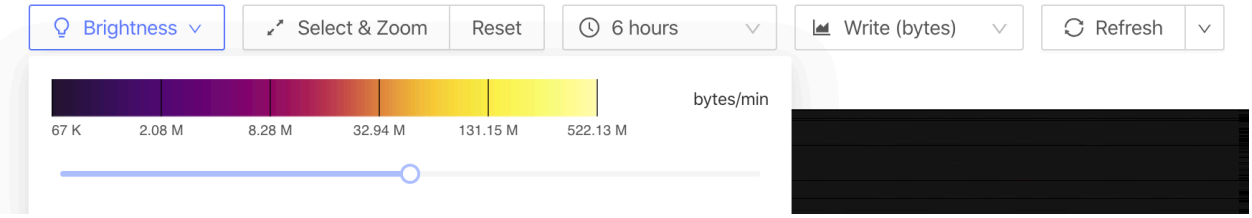


图 392: 工具栏

功能开启后，后台会持续收集数据，稍等一段时间即可看到热力图。

观察某一段时间或者 Region 范围

打开流量可视化页面时，默认会显示最近六小时整个数据库内的热力图。其中，越靠近右侧（当前时间）时，每列 Bucket 对应的时间间隔会越小。如果你想观察某个特定时间段或者特定的 Region 范围，则可以通过放大来获得更多细节。具体操作描述如下：

- 在热力图中向上或向下滚动。
- 点击 Select & Zoom（框选）按钮，然后点击并拖动以选择要放大的区域。

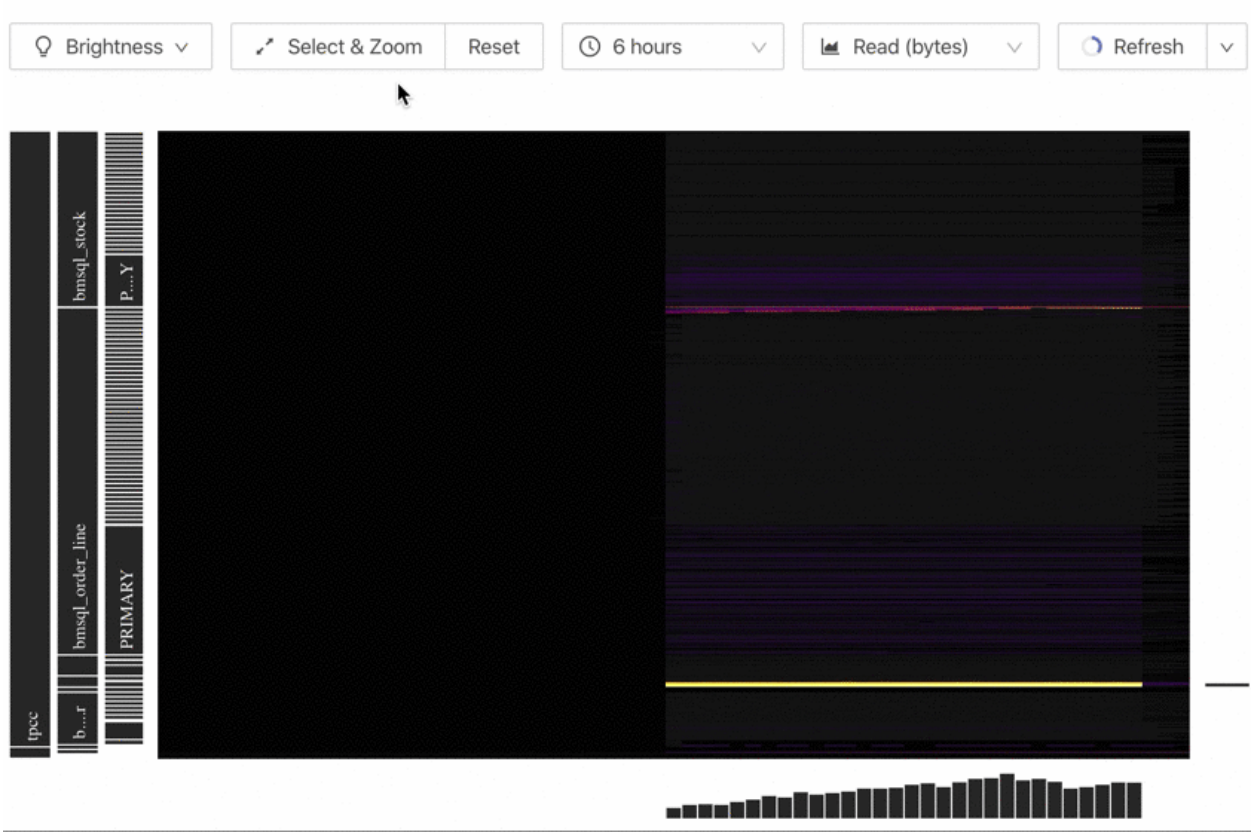


图 393: 框选

- 点击 Reset（重置）按钮，将 Region 范围重置为整个数据库。
- 点击「时间选择框」(界面的 6 hours 处)，重新选择观察时间段。

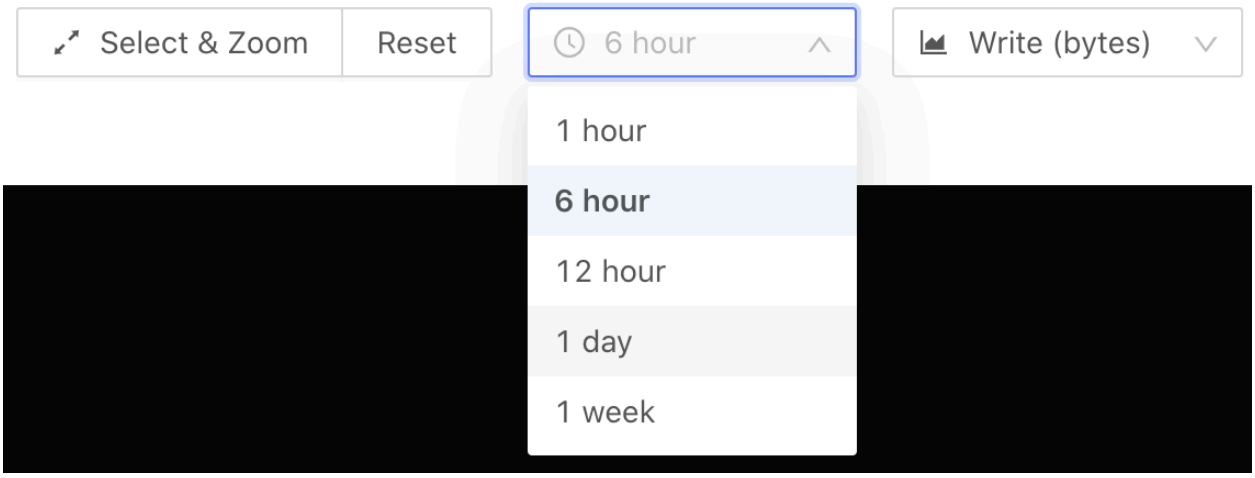


图 394: 时间选择

> 注意：
>
> 进行后三种操作，将引起热力图的重新绘制。你可能观察到热力图与放大前有较大差异。这是一个正常的现象，可能是由于在进行局部观察时，Region 压缩的粒度发生了变化，或者是局部范围内，“热”的基准发生了改变。

调整亮度

热力图使用颜色的明暗来表达一个 Bucket 的流量高低，颜色越暗 (cold) 表示该区域的 Region 在这个时间段上读写流量较低，颜色越亮 (hot) 表示读写流量越高，即越热。如果热力图中的颜色太亮或太暗，则可能很难观察到细节。此时，可以点击 Brightness（调整亮度）按钮，然后通过滑块来调节页面的亮度。

> 注意：
>
> 在显示一个区域内的热力图时，会根据区域内的流量情况来界定冷热。当整个区域流量较为平均时，即使整体流量在数值上很低，你依然有可能观察到较大的亮色区域。请注意一定要结合数值一起分析。

选择指标

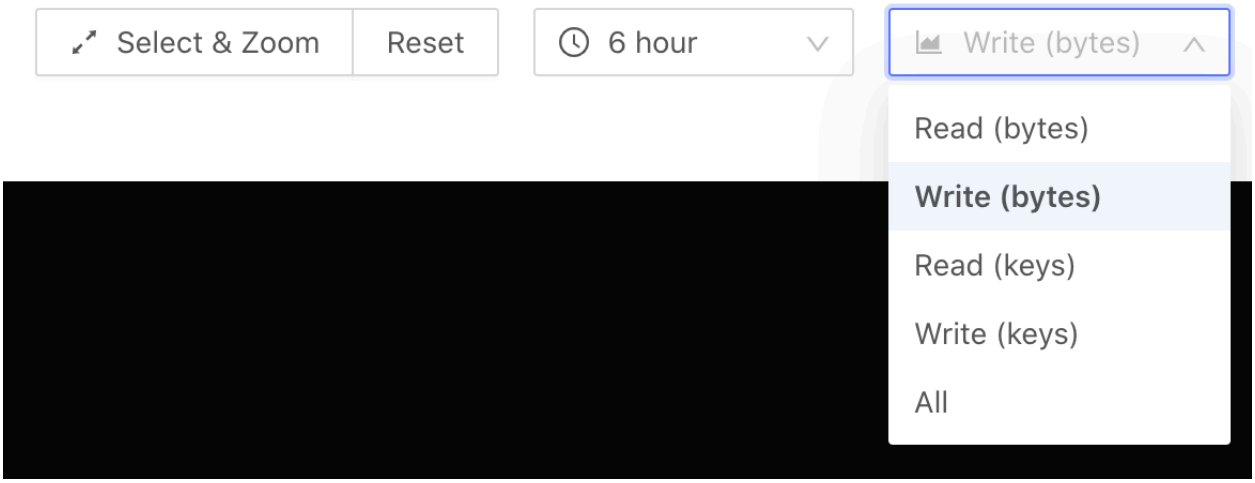


图 395: 指标选择

你可以通过「指标选择框」(以上界面中 Write (bytes) 处) 来查看你关心的指标：

- `Read (bytes)`：读取字节量
- `Write (bytes)`：写入字节量
- `Read (keys)`：读取次数

- `Write (keys)`：写入次数
- `All`：所有（读写流量的总和）

刷新与自动刷新

可以通过点击 Refresh（刷新）按钮来重新获得基于当前时间的热力图。当需要实时观察数据库的流量分布情况时，可以点击按钮右侧的向下箭头，选择一个固定的时间间隔让热力图按此间隔自动刷新。

> 注意：
>
> 如果进行了时间范围或者 Region 范围的调整，自动刷新会被关闭。

查看详情

可以将鼠标悬停在你所关注的 Bucket 上，来查看这个区域的详细信息：

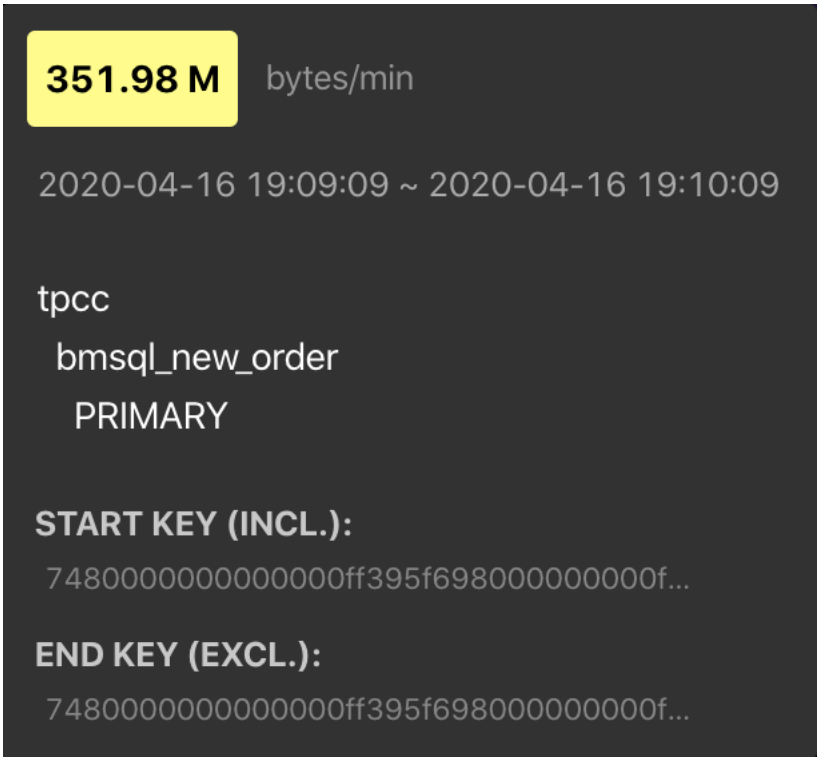


图 396: Bucket 详细信息

如果需要复制某个信息，可以进行点击 Bucket。此时相关详细信息的页面会被暂时钉住。点击你关注的信息，即可将其复制到剪切板：

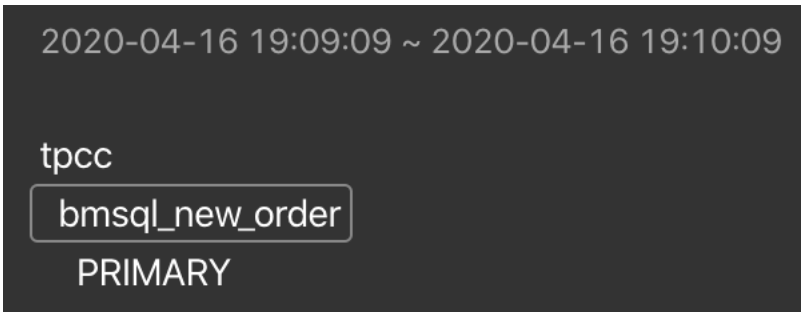


图 397: 复制 Bucket 详细信息

12.12.1.6.5 常见热力图解读

本章节选取了 Key Visualizer 中常见的四种热力图进行解读。

均衡：期望结果

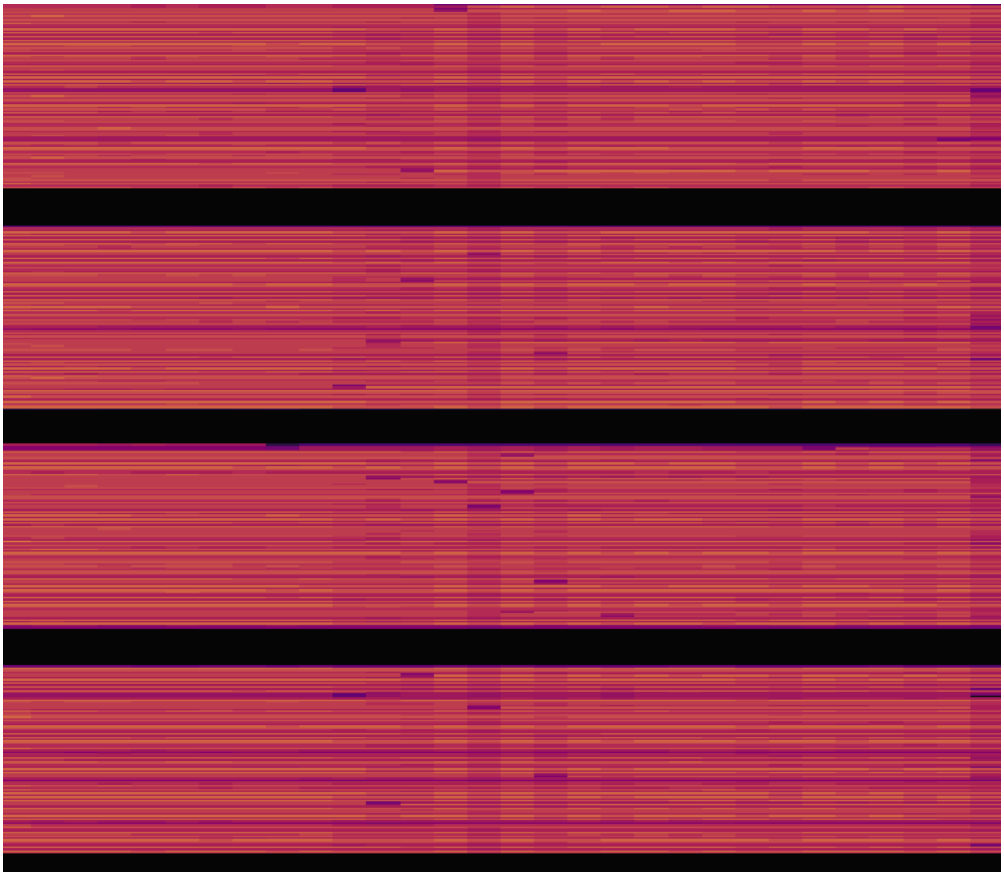


图 398: 均衡结果图

如上图所示，热力图颜色均匀或者深色和亮色混合良好，说明读取或写入在时间和 Region 空间范围上都分布得比较均衡，访问压力均匀地分摊在所有的机器上。这种负载是最适合分布式数据库的。

X 轴明暗交替：需要关注高峰期的资源情况

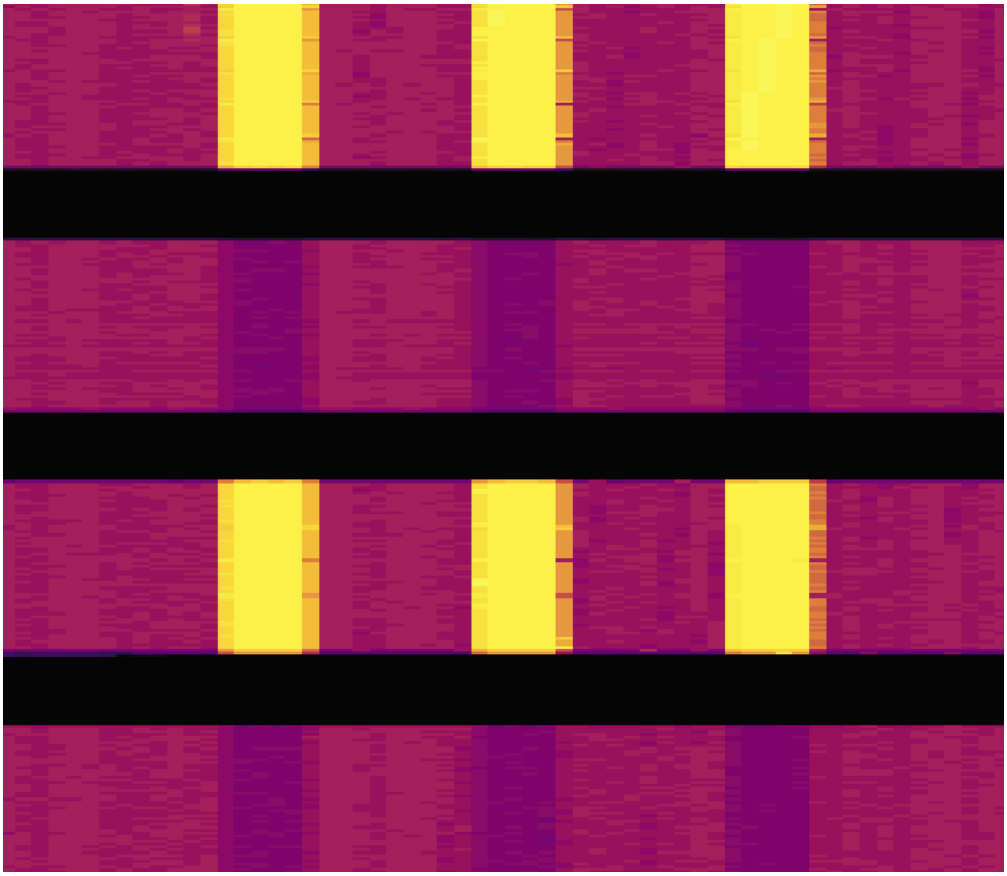


图 399: X 轴明暗交替

如上图所示，热力图在 X 轴（时间）上表现出明暗交替，但 Y 轴 (Region) 则比较均匀，说明读取或写入负载具有周期性的变化。这种情况可能出现在周期性的定时任务场景，如大数据平台每天定时从 TiDB 中抽取数据。一般来说可以关注一下使用高峰时期资源是否充裕。

Y 轴明暗交替：需要关注产生的热点聚集程度

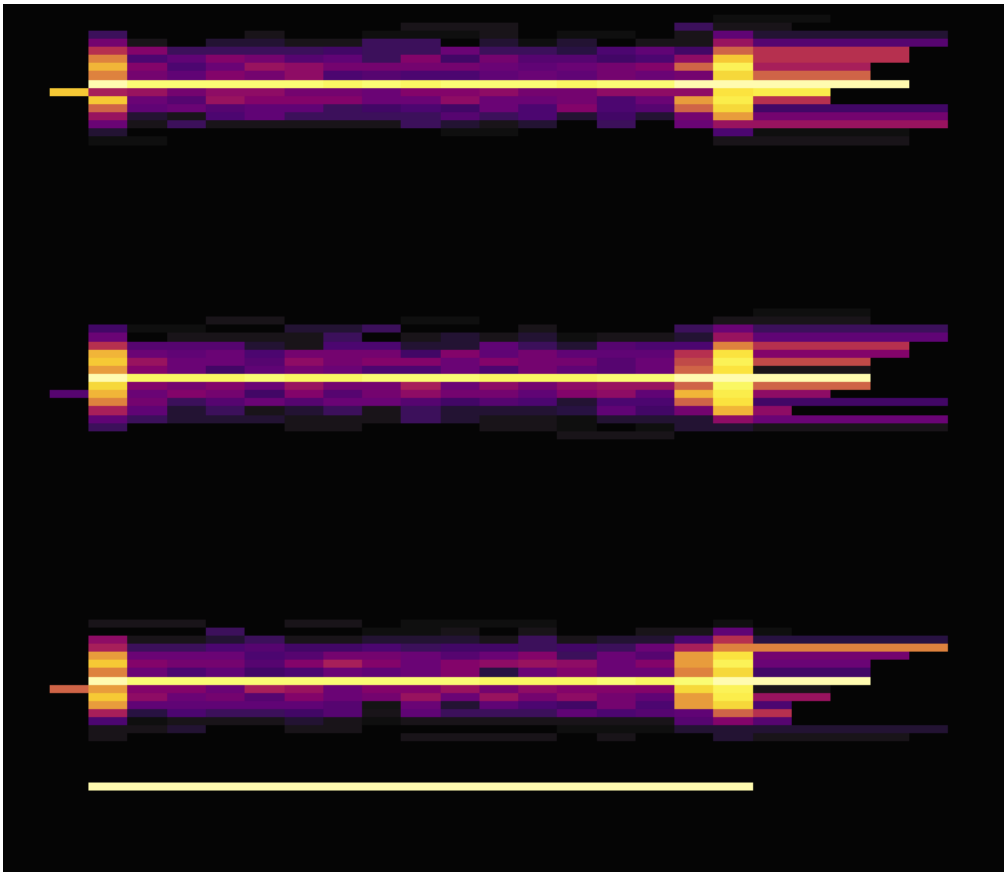


图 400: Y 轴明暗交替

如上图所示，热力图包含几个明亮的条纹，从 Y 轴来看条纹周围都是暗的，这表明明亮条纹区域的 Region 有很高的读写流量，可以从业务角度观察一下是否符合预期。例如，所有业务都关联用户表的情况下，用户表的整体流量就会很高，那么在热力图中表现为亮色区域就非常合理。

另外，明亮区域的高度（Y 轴方向的粗细）非常关键。由于 TiKV 自身拥有以 Region 为单位的热点平衡机制，因此涉及热点的 Region 越多其实越能有利于在所有 TiKV 实例上均衡流量。明亮条纹越粗、数量越多则意味着热点越分散、更多的 TiKV 能得到利用；明亮条纹越细、数量越少意味着热点越集中、热点 TiKV 越显著、越需要人工介入并关注。

明亮斜线：需要关注业务模式

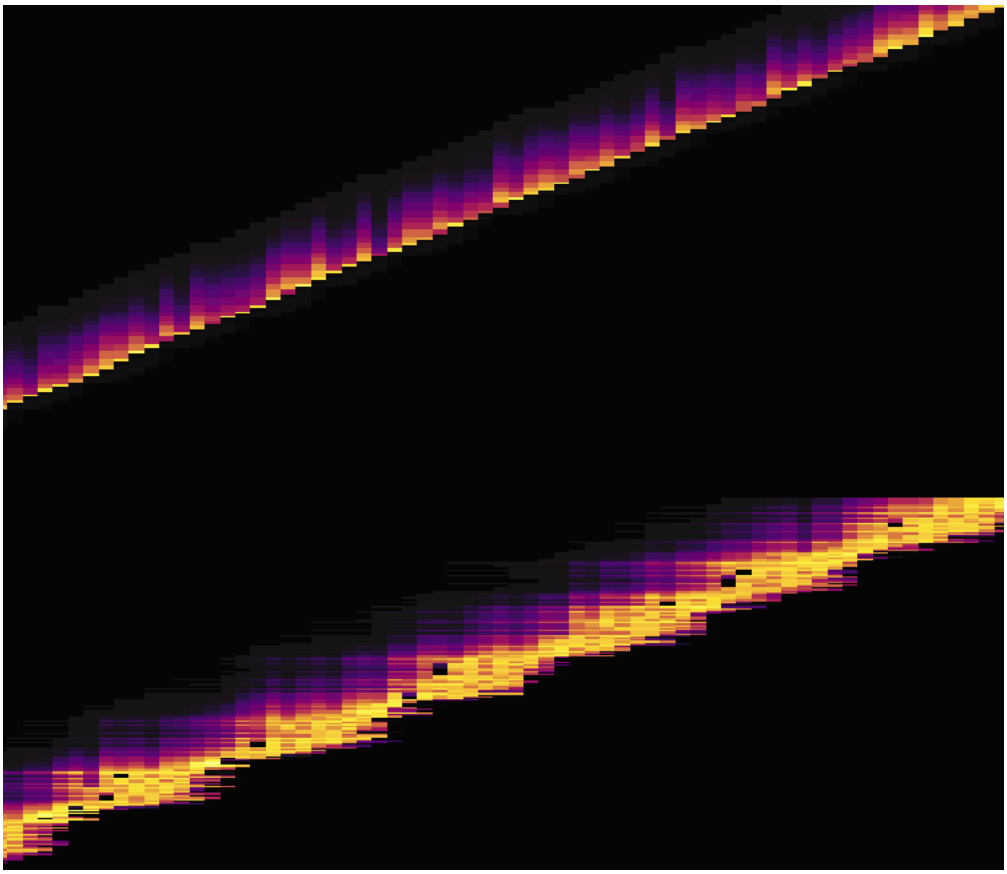


图 401: 明亮斜线

如上图所示，热力图显示了明亮的斜线，表明读写的 Region 是连续的。这种场景常常出现在带索引的数据导入或者扫描阶段。例如，向自增 ID 的表进行连续写入等等。图中明亮部分对应的 Region 是读写流量的热点，往往会成为整个集群的性能问题所在。这种时候，可能需要业务重新调整主键，尽可能打散以将压力分散在多个 Region 上，或者选择将业务任务安排在低峰期。

> 注意：
>
> 这里只是列出了几种常见的热力图模式。流量可视化页面中实际展示的是整个集群上所有数据库、数据表的热力图，因此非常有可能在不同的区域观察到不同的热力图模式，也可能观察到多种热力图模式的混合结果。使用的时候应当视实际情况灵活判断。

12.12.1.6.6 解决热点问题

TiDB 内置了不少帮助缓解常见热点问题的功能，深入了解请参考TiDB 高并发写入场景最佳实践。

12.12.1.7 TiDB Dashboard 监控关系图

TiDB Dashboard 监控关系图是 TiDB v4.0.7 起提供的新功能，可以将集群中各个内部流程的耗时监控数据绘制为关系图，帮助用户快速了解集群中各个环节的耗时及关系。

12.12.1.7.1 访问关系图

登录 TiDB Dashboard 后点击左侧导航的集群诊断可以进入此功能页面：

图 402: 生成监控关系图首页

设置区间起始时间和区间长度参数后，点击生成监控关系图按钮后，会进入监控关系图页面。

12.12.1.7.2 关系图解读

下面是一份监控耗时关系图示例，描述的是某个 TiDB 集群在 2020-07-29 16:36:00 开始往后 5 分钟内，TiDB 集群中各个监控的总耗时比例，以及各项监控之间的关系。

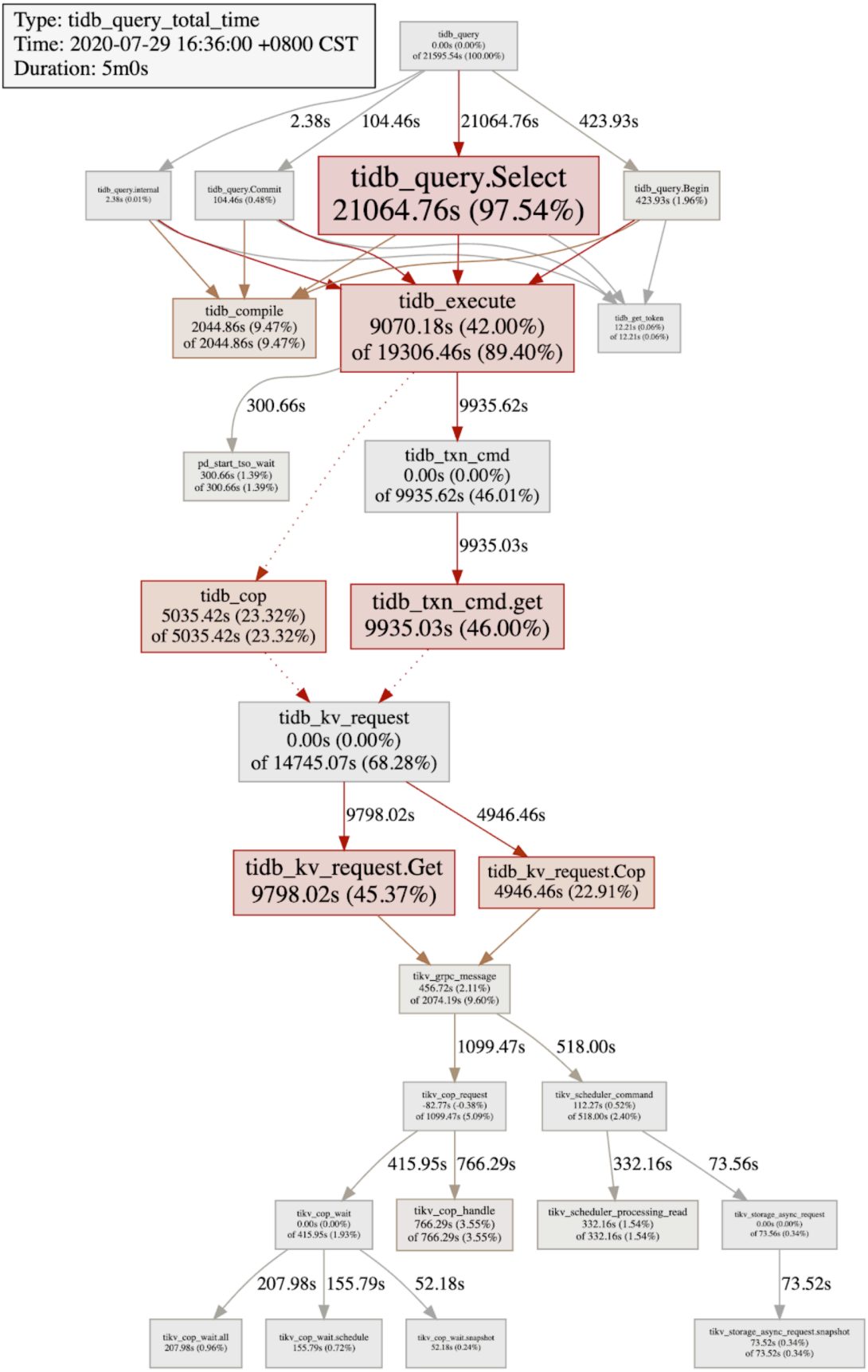


图 403: 监控关系图示例

例如以下 tidb_execute 节点监控图表示：tidb_execute 监控项的总耗时为 19306.46 秒，占总查询耗时的 89.4%，其中 tidb_execute 节点自身的耗时是 9070.18 秒，占总查询耗时的 42%。将鼠标悬停在该方框上，可以看到监控项的注释说明，总耗时、平均耗时、平均 P99 耗时等详细信息。

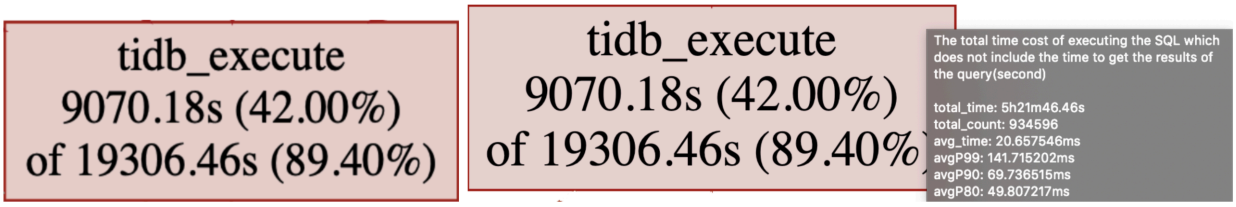


图 404: 监控关系图 tidb_execute 节点示例

节点的含义

每个方框节点代表一个监控项，包含了以下信息：

- 监控项的名称
- 监控项的总耗时
- 监控项总耗时和查询总耗时的比例

节点监控的总耗时 = 节点自身的耗时 + 子节点的耗时，所以某些节点监控图会显示节点自身的耗时和总耗时的比例。例如 tidb_execute 监控：

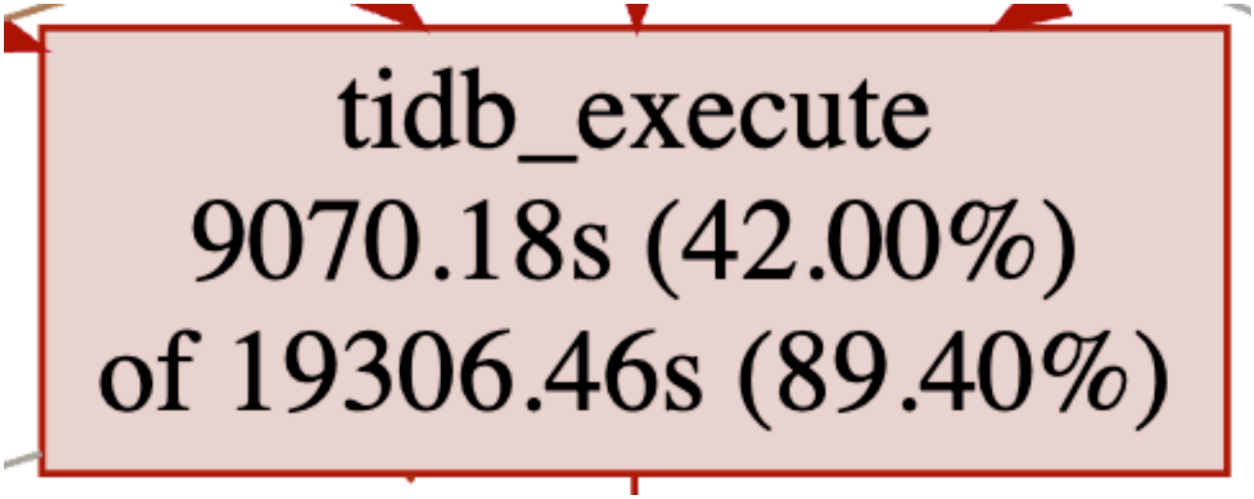


图 405: 监控关系图 tidb_execute 节点示例

- tidb_execute 是监控项的名字。该监控是指一条 SQL 请求在 TiDB 执行引擎中的执行耗时。
- 19306.46s 表示 tidb_execute 监控项消耗的总时间为 19306.46 秒。89.40% 表示 19306.46 秒占所有 SQL 查询总耗时（包括用户 SQL 和 TiDB 内部的 SQL）的比例为 89.40%。查询总耗时是 tidb_query 监控项的总耗时。

- `9070.18s` 表示 `tidb_execute` 节点自身总执行耗时是 9070.18 秒，其余部分是被其子节点消耗的时间。`42.00%` 表示 9070.18 秒占所有查询总耗时的比例为 42.00%。

将鼠标悬停在该节点后，会显示监控项的更多详细信息：

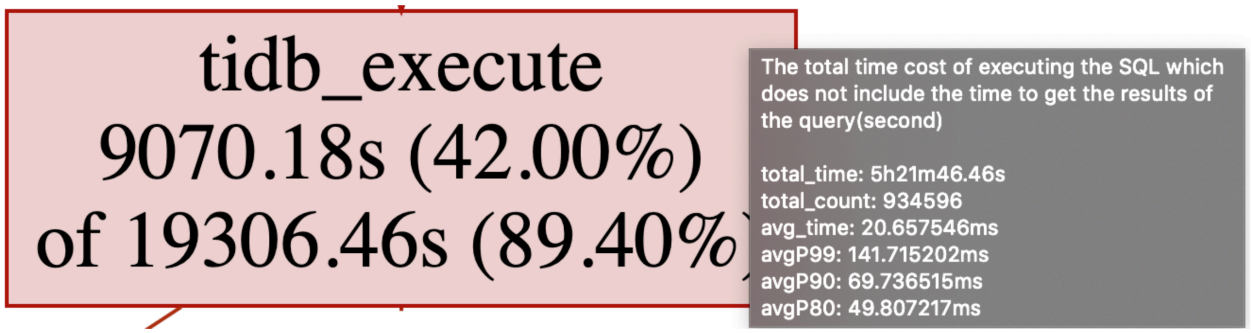


图 406: 监控关系图 tidb_execute 节点注释

上图信息为该项监控的注释说明，包括总耗时、总次数、平均耗时和平均 P99、P90、P80 耗时。

监控项之间的父子关系

下面以 `tidb_execute` 监控为例介绍该监控项相关的子节点：

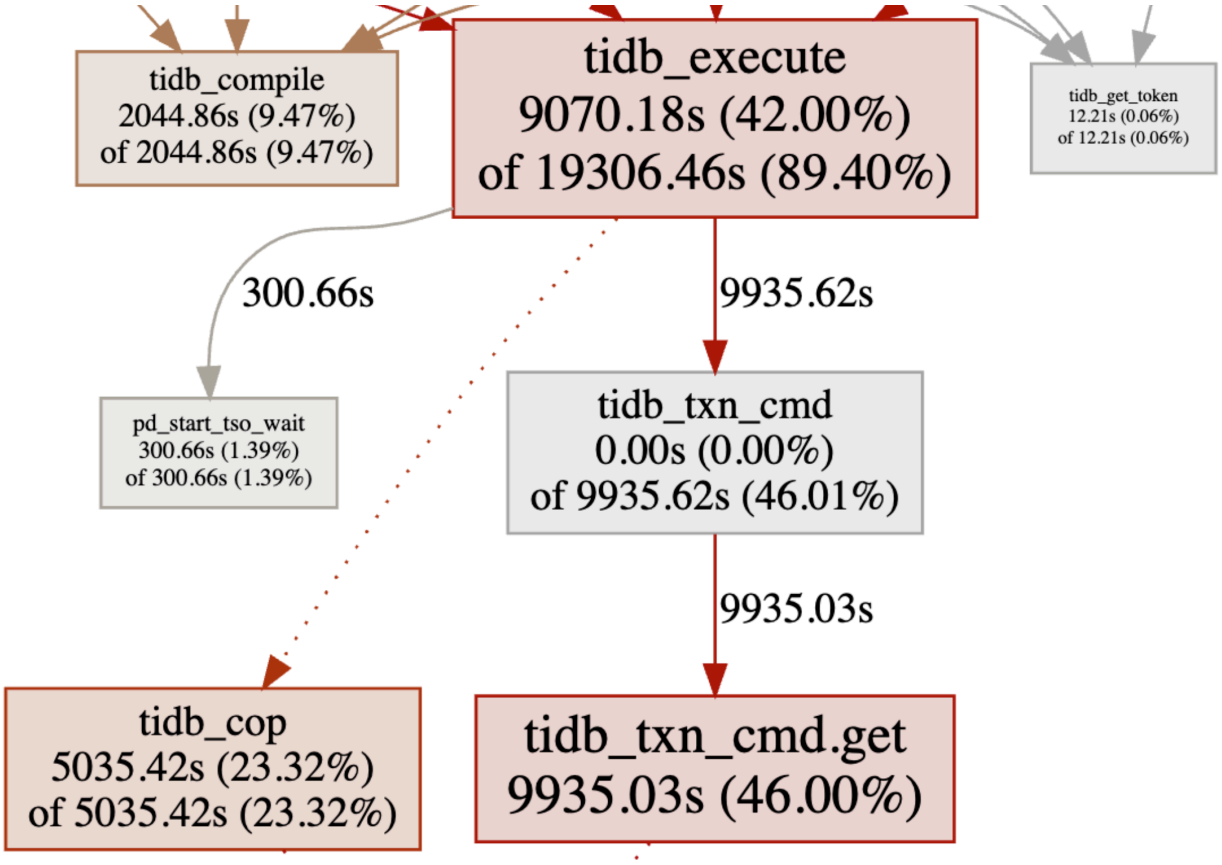


图 407: 监控关系图 tidb_execute 节点注释

可以看到，tidb_execute 包含两个子节点，分别是：

- pd_start_tso_wait：等待事务 start_tso 的耗时，其总耗时是 300.66 秒。
- tidb_txn_cmd：TiDB 执行事务相关命令的耗时，其总耗时是 9935.62 秒。

另外，tidb_execute 还有一条虚线箭头指向 tidb_cop 监控，这里虚线箭头的含义如下：

tidb_execute 包含 tidb_cop 监控的耗时，但是 cop 请求有可能并发执行。例如对两个表的进行 join 查询的 execute 耗时为 60 秒，其中 join 的两个表会并行地执行 cop 扫表请求。假如 cop 请求执行时间分别为 40 秒和 30 秒，那 cop 请求的总耗时是 70 秒，但是 execute 执行耗时只有 60 秒。所以如果父节点的耗时不完全包含子节点的耗时，就用虚线箭头来指向子节点。

> 注意：
>
> 当节点有虚线箭头指向的子节点时，该节点的本身的耗时是不准确的。例如 tidb_execute 监控中，tidb_execute 节点本身的耗时为 9070.18 = 19306.46 - 300.66 - 9935.62。这里 tidb_cop 节点的耗时并不会计入子节点耗时的计算，但实际上，tidb_execute 监控本身的耗时 9070.18 秒中包含了 tidb_cop 一部分监控节点的耗时，但无法确认具体包含了多少耗时。

tidb_kv_request 及其父节点

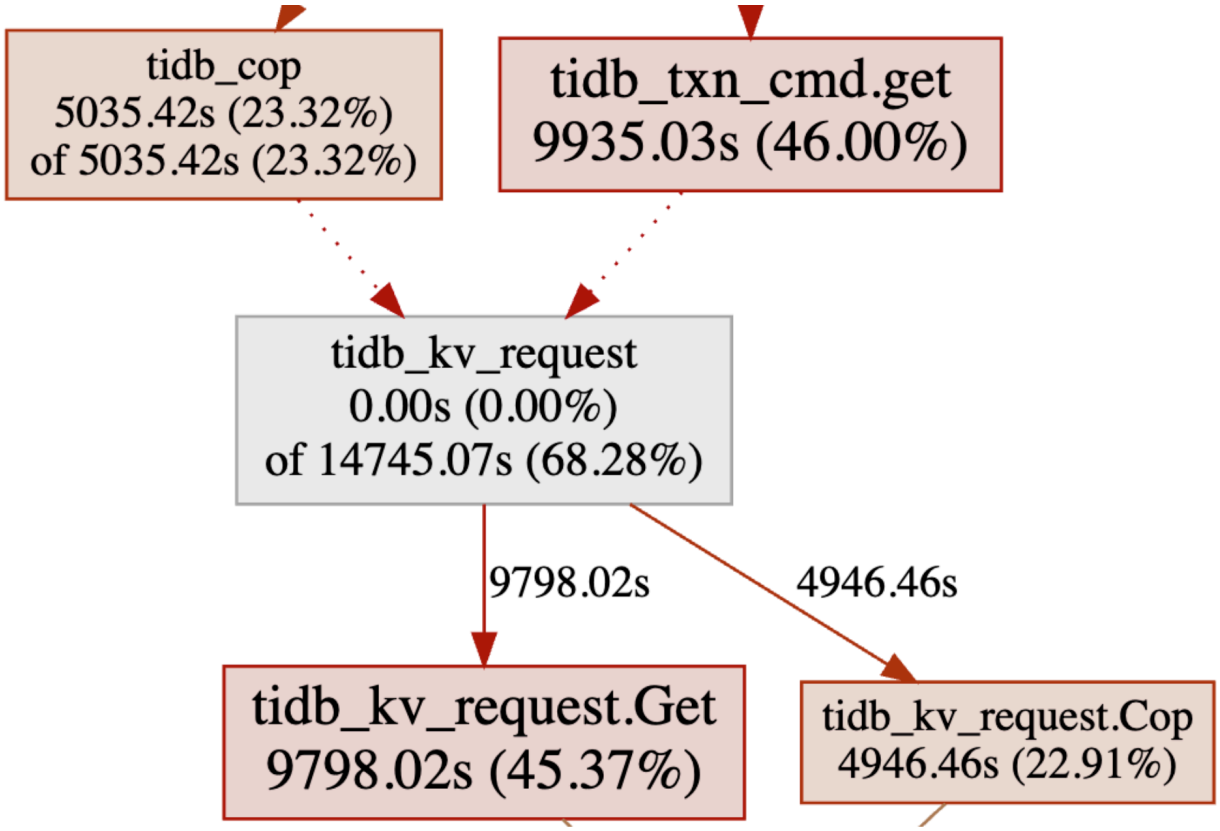


图 408: 监控关系图虚线节点关系

tidb_kv_request 的父节点 tidb_cop 和 tidb_txn_cmd.get 都用虚拟箭头指向 tidb_kv_request，这里表示：

- tidb_cop 的耗时包含部分 tidb_kv_request 的耗时
- tidb_txn_cmd.get 的耗时也包含部分的 tidb_kv_request 的耗时。

但是 tidb_cop 具体有多少耗时是 tidb_kv_request 消耗的，无法进行确认。

- tidb_kv_request.Get：TiDB 发送 Get 类型的 kv 请求的耗时。
- tidb_kv_request.Cop：TiDB 发送 Cop 类型的 kv 请求的耗时。

tidb_kv_request 与 tidb_kv_request.Get 和 tidb_kv_request.Cop 并不是父节点包含子节点的关系，而是组成关系。子节点的名称前缀是父节点的名称加上 .xxx，即为父节点的子类。这里可以理解为，TiDB 发送 kv 请求的总耗时为 14745.07 秒，其中 Get 和 Cop 类型的 kv 请求的总耗时分别为 9798.02 秒和 4946.46 秒。

12.12.1.8 SQL 语句分析

12.12.1.8.1 TiDB Dashboard SQL 语句分析执行详情页面

该页面可以查看所有 SQL 语句在集群上执行情况，常用于分析总耗时或单次耗时执行耗时较长的 SQL 语句。

在该页面中，结构一致的 SQL 查询（即使查询参数不一致）都会被归为同一个 SQL 语句，例如 `SELECT * FROM employee WHERE id IN (1, 2, 3)` 和 `select * from EMPLOYEE where ID in (4, 5)` 都属于同一 SQL 语句 `select * from employee where id in (...)`。

访问列表页面

可以通过以下两种方法访问 SQL 语句分析页面：

- 登录后，左侧导航条点击 SQL 语句分析（SQL Statements）：

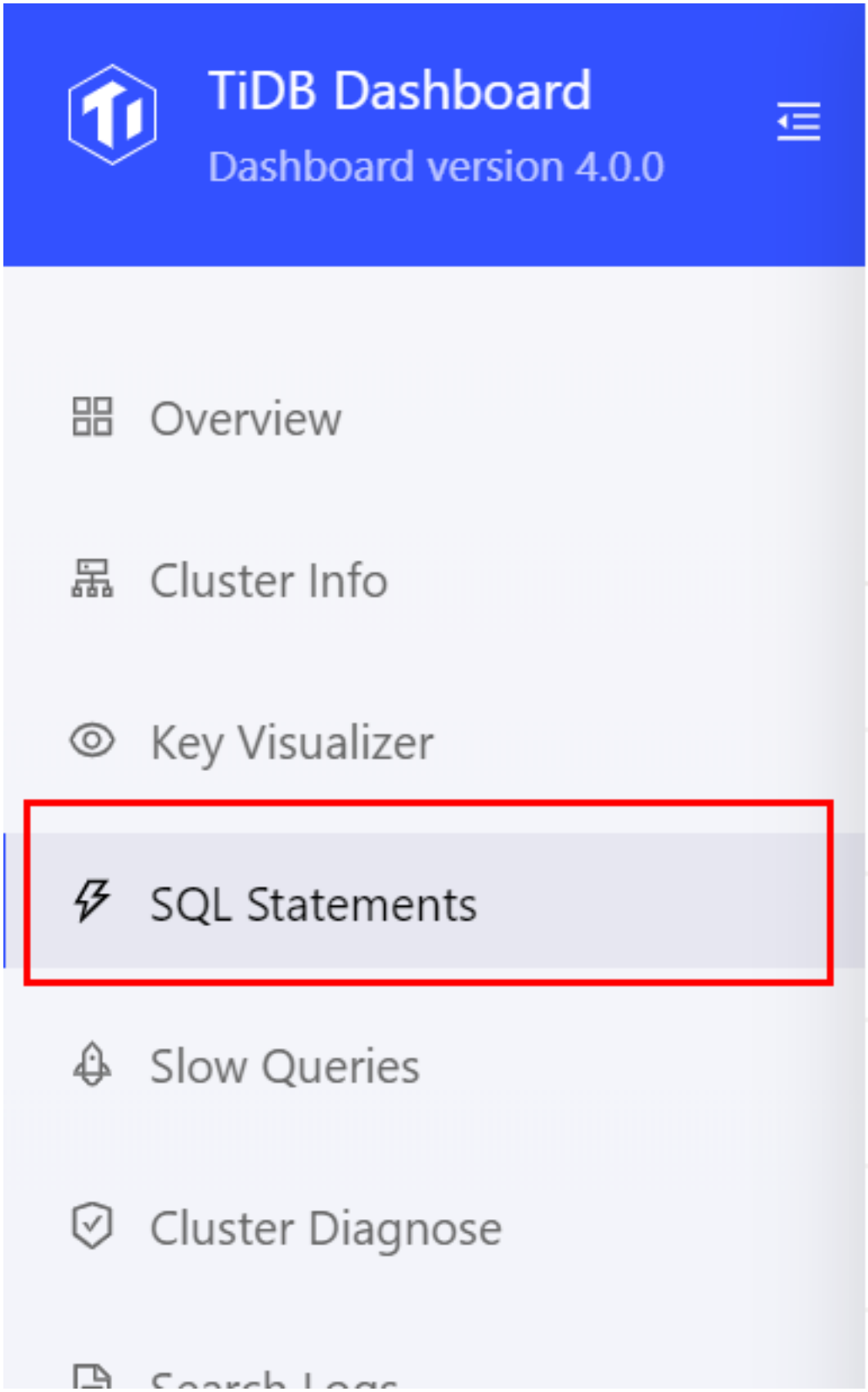

TiDB Dashboard
Dashboard version 4.0.0
Overview
Cluster Info
Key Visualizer
SQL Statements
Slow Queries
Cluster Diagnose

图 409: 访问

- 在浏览器中访问 http://127.0.0.1:2379/dashboard/#/statement（将 127.0.0.1:2379 替换为实际 PD 实例地址和端口）。

SQL 语句分析页面所展示的所有数据都来自于 TiDB Statement 系统表，参见Statement Summary Tables 文档了解该系统表的详细情况。

修改列表过滤条件

页面顶部可修改显示的时间范围、按执行 SQL 所在数据库进行过滤，或按 SQL 类型进行过滤，如下所示。默认显示最近一个收集周期（默认最近 30 分钟）内的所有 SQL 语句执行情况。

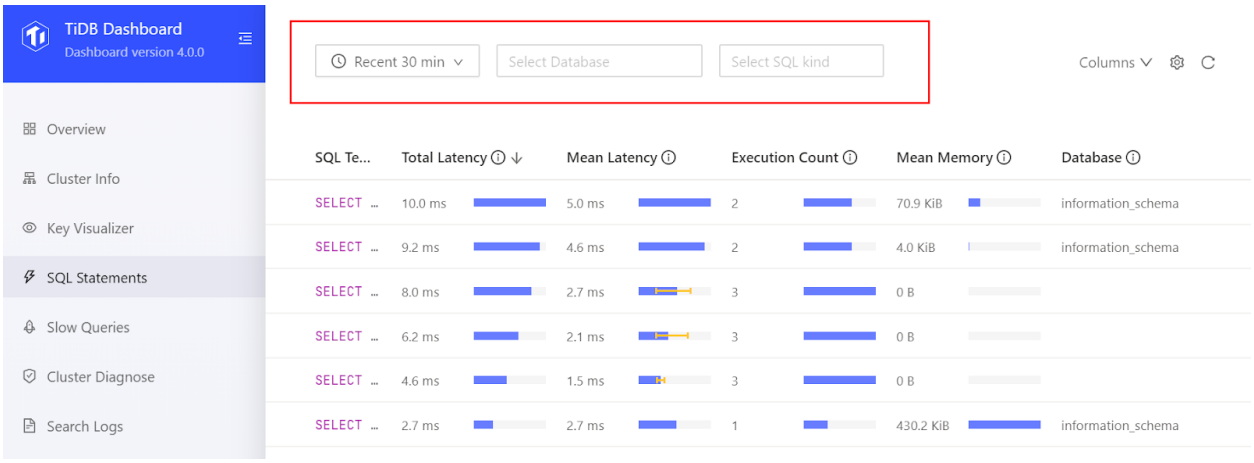


图 410: 修改过滤条件

显示其他字段

页面顶部选择列（Columns）选项中可选择显示更多列，可将鼠标移动到列名右侧的 (i) 图标处查看列的说明：

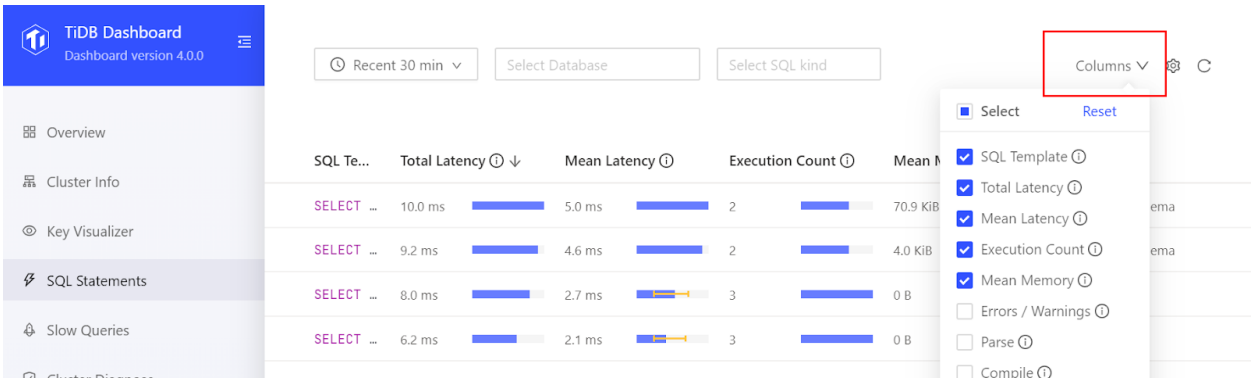


图 411: 选择列

修改列表排序依据

列表默认以累计耗时（Total Latency）从高到低进行排序，点击不同的列标题可以修改排序依据或切换排序顺序：

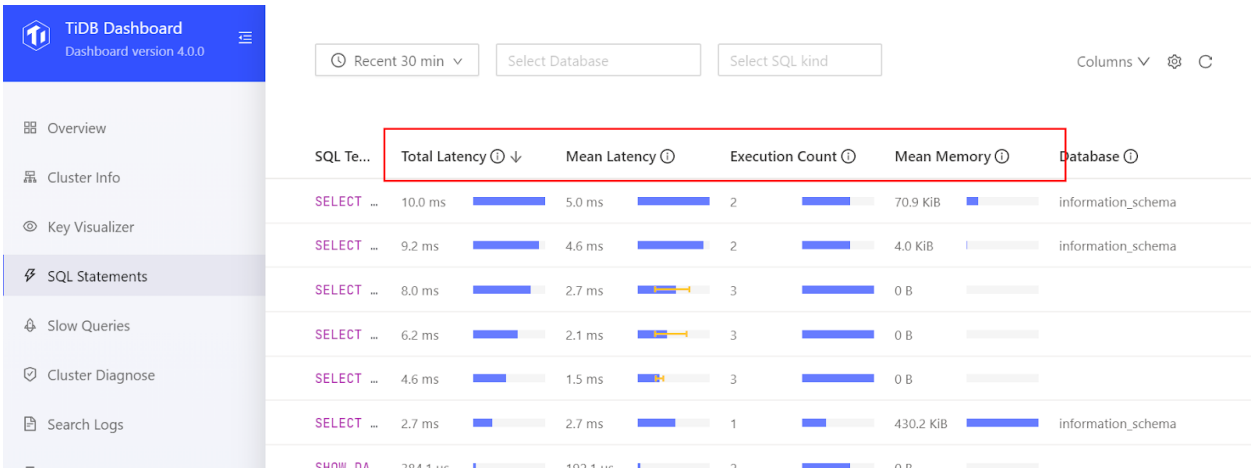


图 412: 修改列排序

修改数据收集设置

在列表页面，点击顶部右侧的设置（Settings）按钮，即可对 SQL 语句分析功能进行设置：

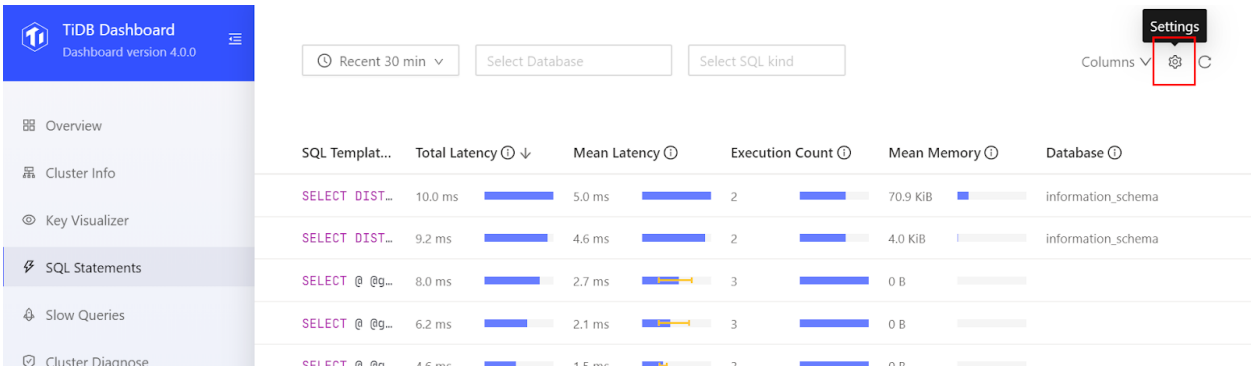


图 413: 设置入口

点击后设置界面如下图所示：

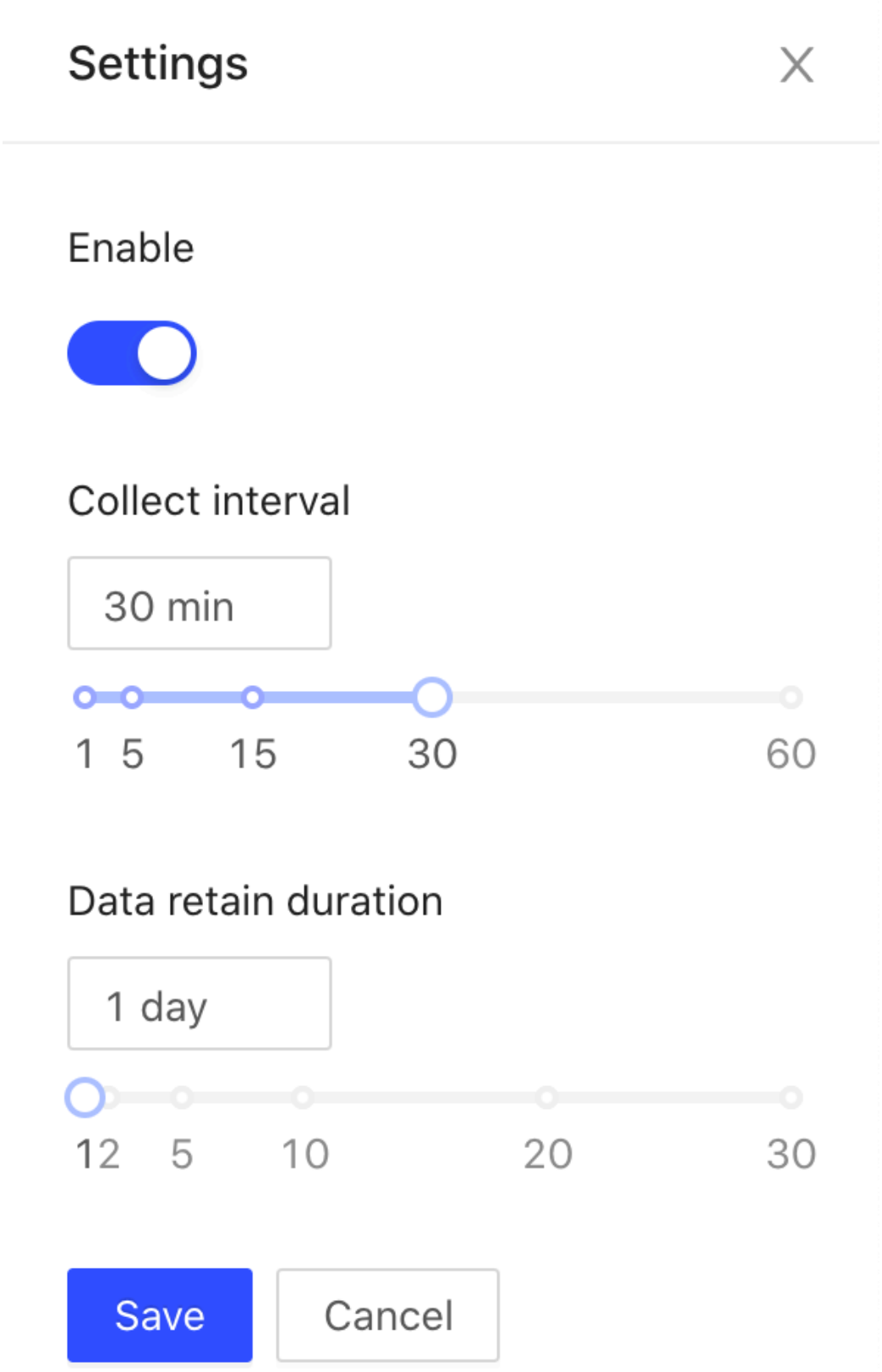

Settings
Enable
Collect interval
30 min
1 5 15 30 60
Data retain duration
1 day
12 5 10 20 30
Save
Cancel

图 414: 设置

在设置中可以选择关闭或开启 SQL 语句分析功能。在开启 SQL 语句分析功能时可以修改以下选项：

- 数据收集周期：默认 30 分钟，每次进行 SQL 语句分析的时间长度。SQL 语句分析功能每次对一段时间范围内的所有 SQL 语句进行汇总统计，如果这个时间范围过长，则统计的粒度粗，不利用定位问题；如果太短，则统计的粒度细，方便定位问题，但会导致在相同的数据保留时间内产生更多的记录，产生更多的内存占用。因此需要根据实际情况调整，在需要定位问题时适当地将值调低。
- 数据保留时间：默认 1 天，统计信息保留的时间，超过这个时间的数据会被从系统表中删除。

参见Statement Summary Tables 参数设置 文档了解详细情况。

> 注意：
>
> 由于 Statement 系统表只存放在内存中，关闭此功能后，系统表中的数据会将清空。
>
> 数据收集周期和保留时间的值会影响内存占用，因此建议根据实际情况调整，保留时间不宜设置过大。

下一步

阅读查看执行详情章节了解如何进一步查看 SQL 语句的详细执行情况。

12.12.1.8.2 TiDB Dashboard SQL 语句分析执行详情页面

在列表中点击任意一行可以进入该 SQL 语句的详情页查看更详细的信息，这此信息包括三大部分：

- SQL 语句概况：包括 SQL 模板，SQL 模板 ID，当前查看的时间范围，执行计划个数以及执行所在的数据库（下图区域 1）
- 执行计划列表：如果该 SQL 语句有多个执行计划，则显示该列表，可以选择不同的执行计划，在列表下方会显示选中的执行计划详情；如果只有一个执行计划，则该列表不显示（下图区域 2）
- 执行计划详情：显示选中的执行计划的详细信息，具体见下一小节（下图区域 3）

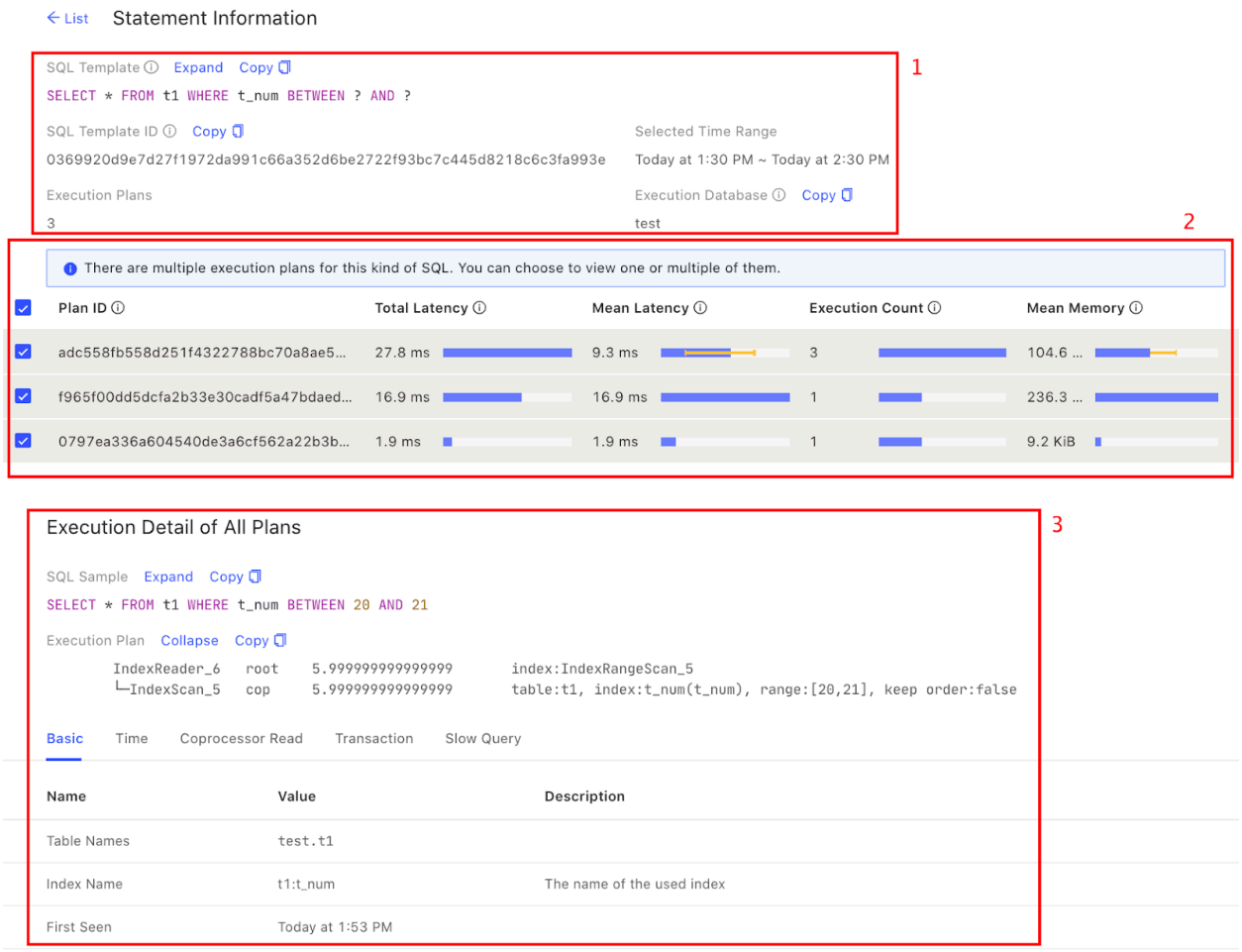


图 415: 详情

执行计划详情

执行计划详情包括以下内容：

- SQL 样本：该计划对应的实际执行的某一条 SQL 语句文本。时间范围内任何出现过的 SQL 都可能作为 SQL 样本。
- 执行计划：执行计划的完整内容，参阅理解 TiDB 执行计划文档了解如何解读执行计划。如果选择了多个执行计划，则显示的是其中任意一个。
- 其他关于该 SQL 的基本信息、执行时间、Coprocessor 读取、事务、慢查询等信息，可点击相应标签页标题切换。

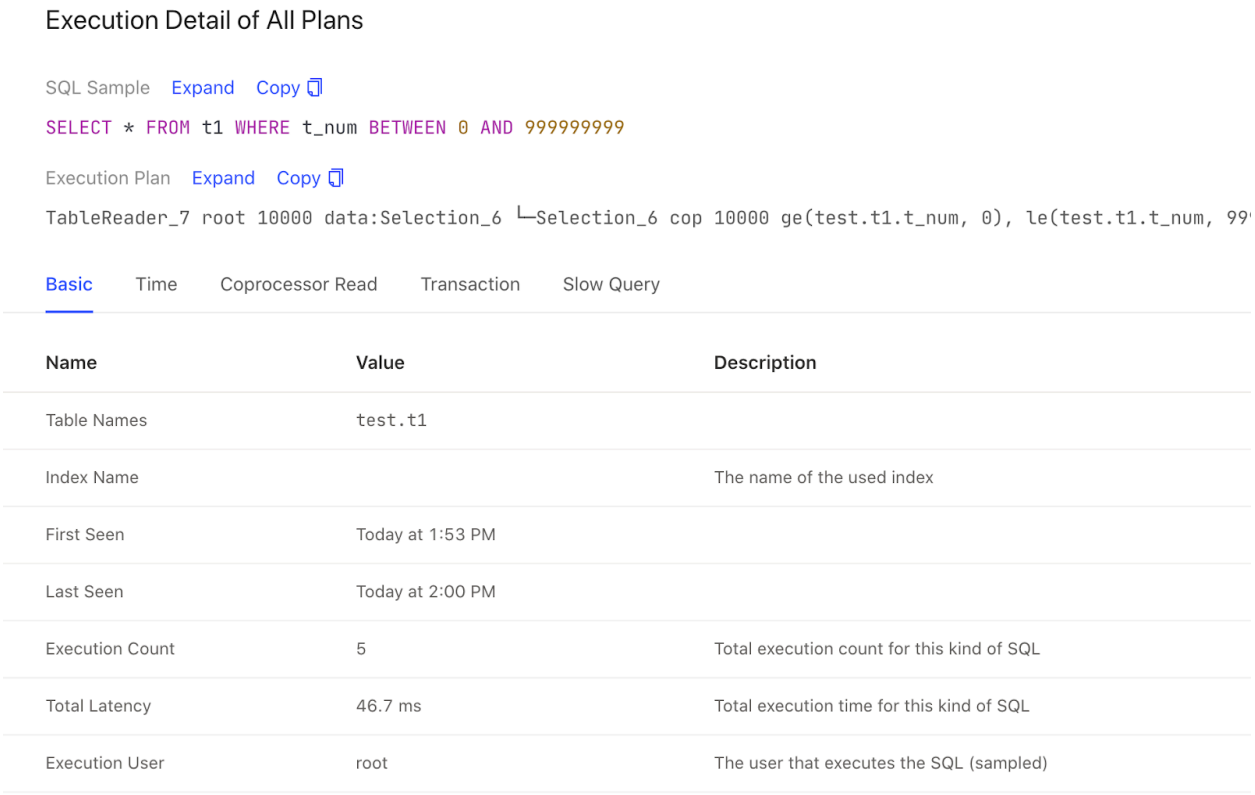


图 416: 执行计划详情

基本信息

包含关于表名、索引名、执行次数、累计耗时等信息。描述（Description）列对各个字段进行了具体描述。

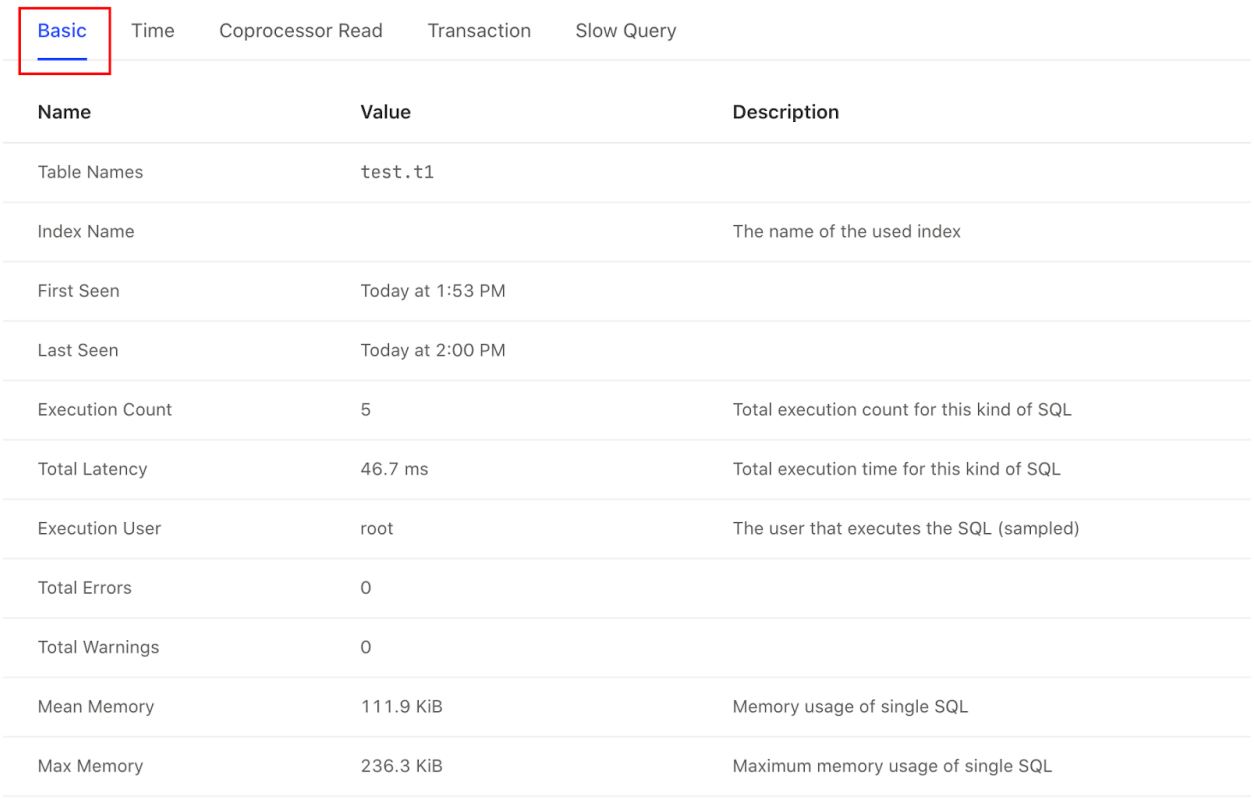


图 417: 基本信息

执行时间

显示执行计划执行的各阶段所耗费时间。

> **注意：**
>
> 由于单个 SQL 语句内部可能有并行执行的操作，因此各阶段累加时间可能超出该 SQL 语句的实际执行时间。

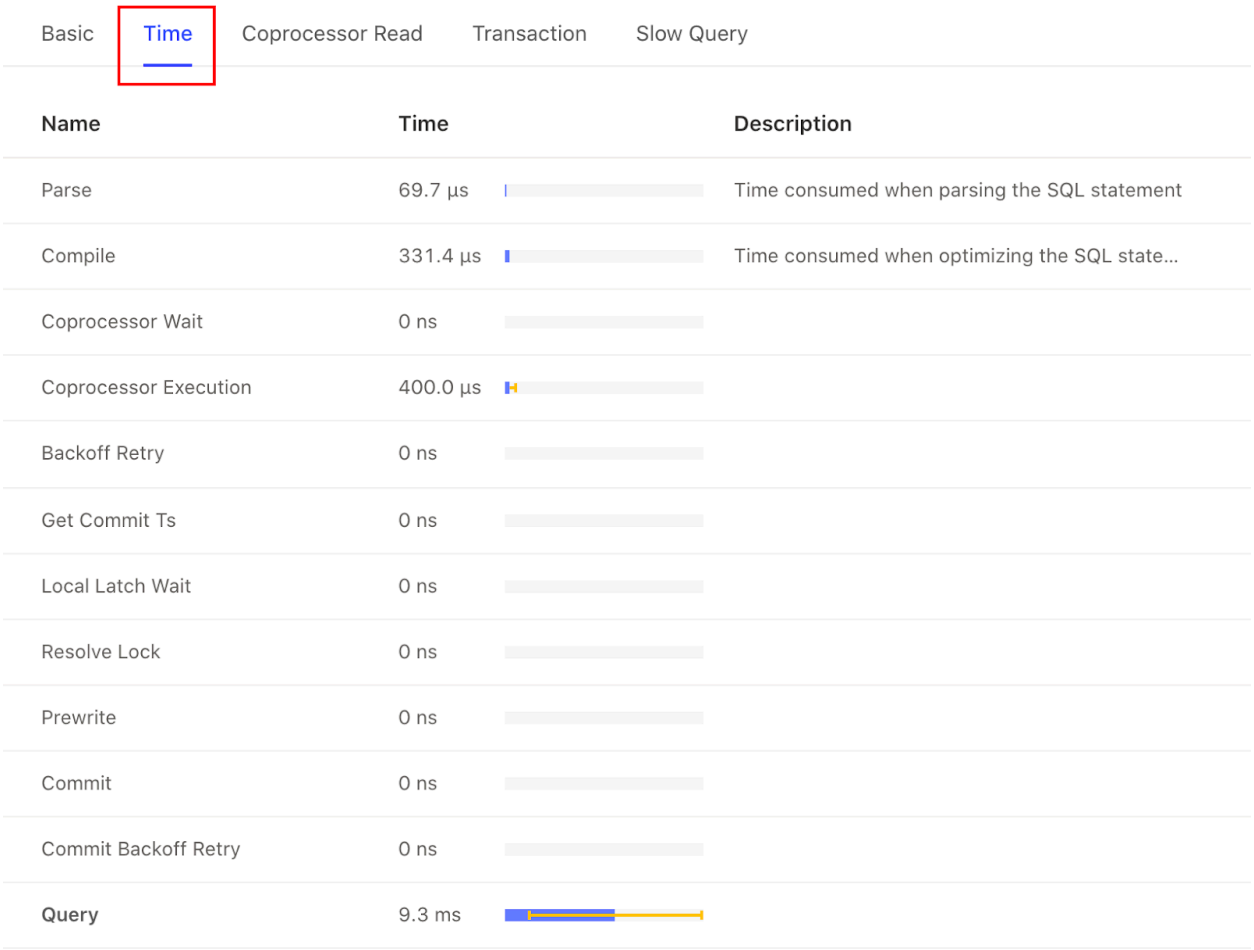


图 418: 执行时间

Coprocessor 读取

显示 Coprocessor 读取的相关信息。

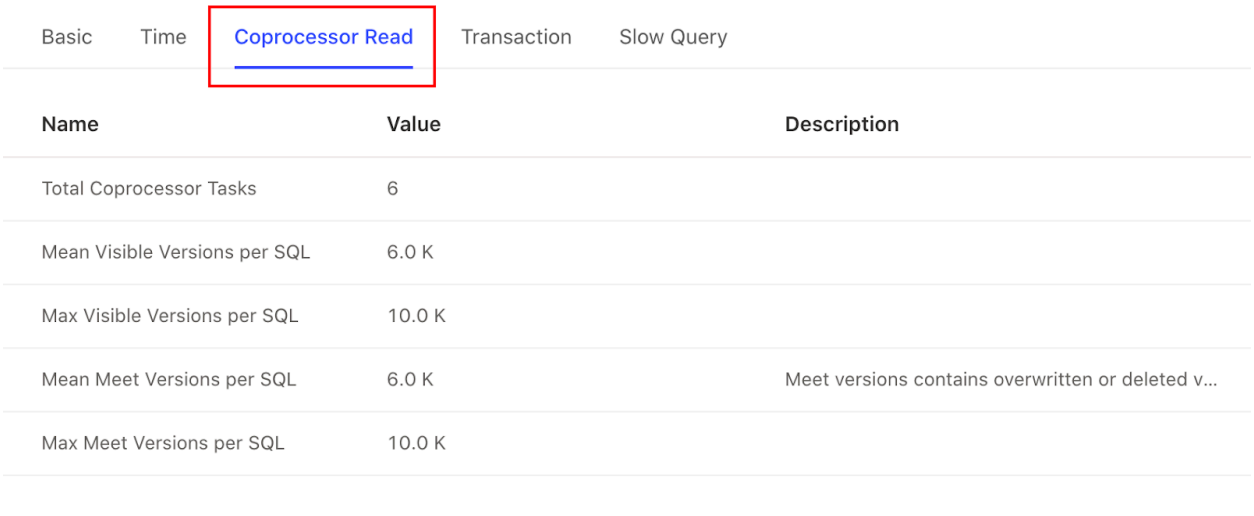


图 419: Coprocessor 读取

事务

显示执行计划与事务相关的信息，比如平均写入 key 个数，最大写入 key 个数等。

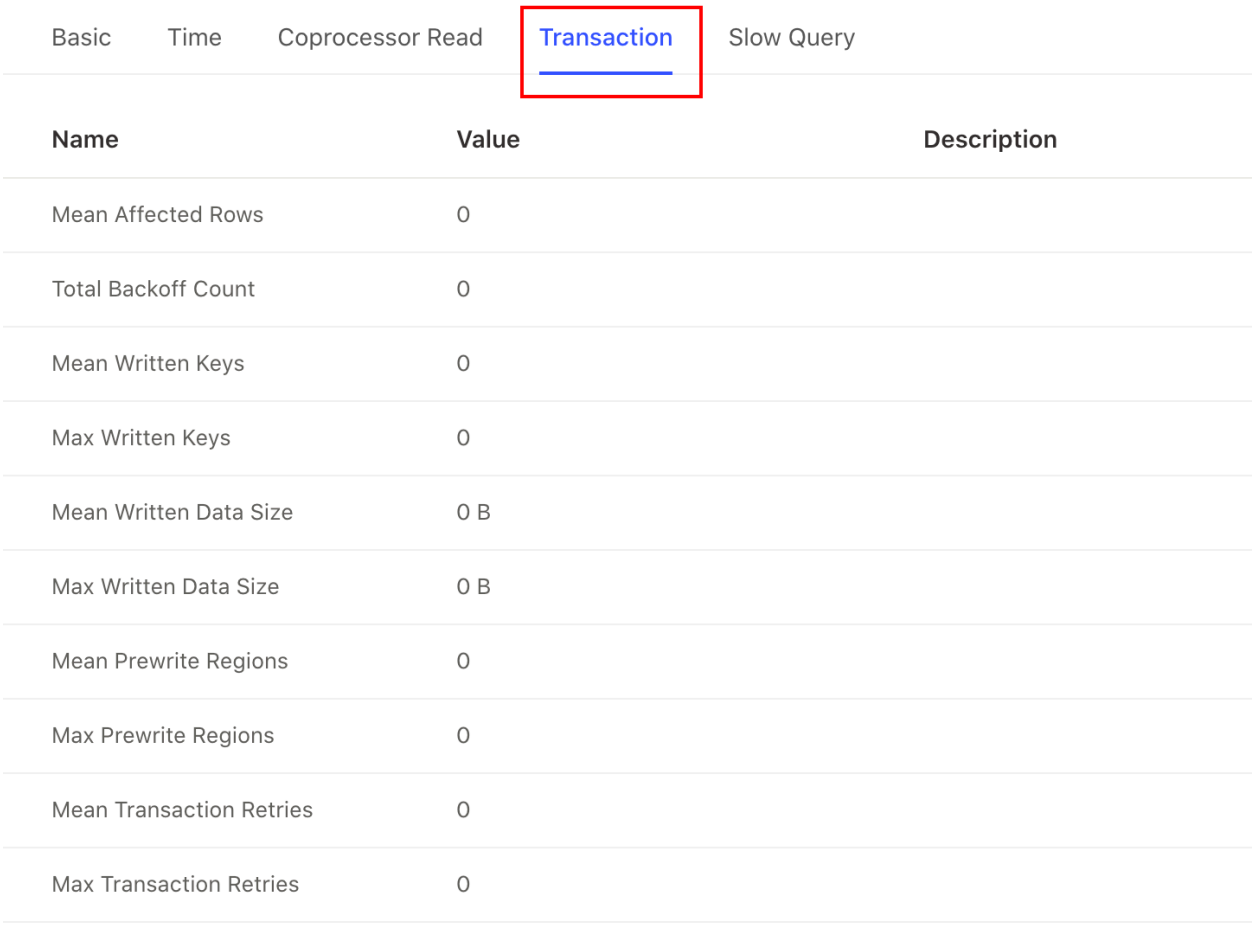


图 420: 事务

慢查询

如果该执行计划执行过慢，则在慢查询标签页下可以看到其关联的慢查询记录。

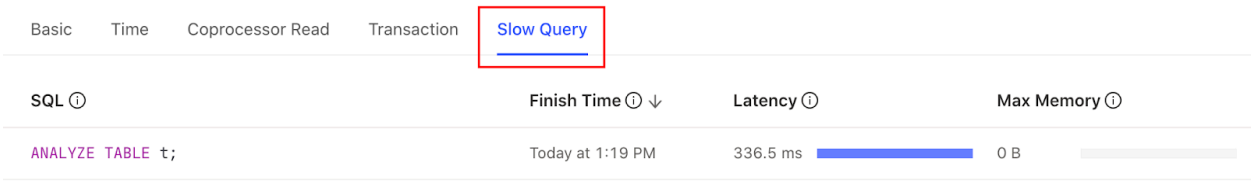


图 421: 慢查询

该区域显示的内容结构与慢查询页面一致，详见慢查询页面。

12.12.1.9 TiDB Dashboard 慢查询页面

该页面上能检索和查看集群中所有慢查询。

默认情况下，执行时间超过 300ms 的 SQL 查询就会被视为慢查询，被记录到慢查询日志中，并可通过本功能对记录到的慢查询进行查询。可调整 tidb_slow_log_threshold SESSION 变量或 TiDB slow-threshold 参数调整慢查询阈值。

> **注意：**
>
> 若关闭了慢查询日志，则本功能不可用。慢查询日志默认开启，可通过修改 TiDB 配置 enable-slow-log 开启或禁用。

12.12.1.9.1 访问列表页面

可以通过以下两种方法访问慢查询页面：

- 登录后，左侧导航条点击慢查询 (Slow Queries)：

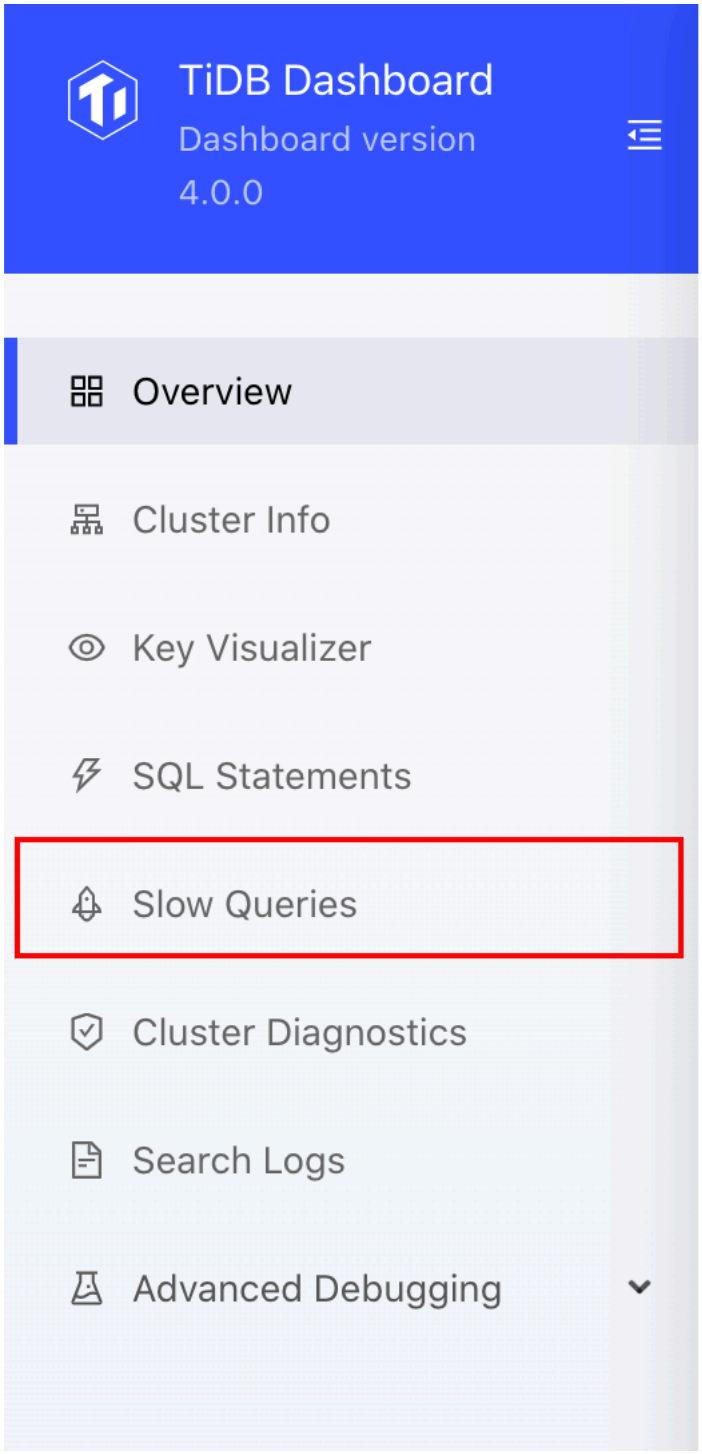


图 422: access 访问页面

- 在浏览器中访问 http://127.0.0.1:2379/dashboard/#/slow_query（将 127.0.0.1:2379 替换为任意实际 PD 地址和端口）。

慢查询页面所展示的所有数据都来自于 TiDB 慢查询系统表及慢查询日志，参见慢查询日志文档了解详细情况。

修改列表过滤条件

可按时间范围、慢查询语句关联的数据库、SQL 关键字、SQL 类型、显示的慢查询语句数量等条件过滤，筛选慢查询句。如下所示，默认显示 30 分钟内最近 100 条慢查询。

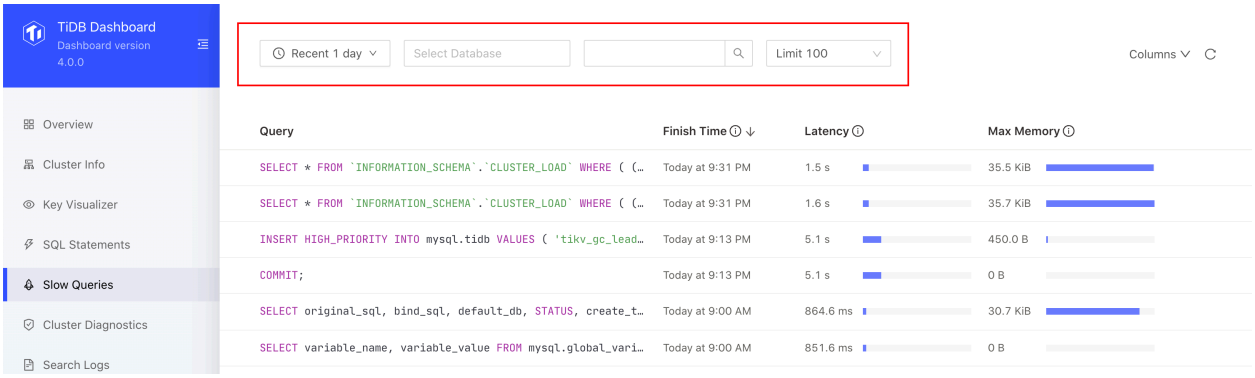


图 423: 修改列表过滤条件

显示更多列信息

页面顶部选择列 (Columns) 选项中可选择显示更多列，可将鼠标移动到列名右侧的 (i) 图标处查看列的说明：

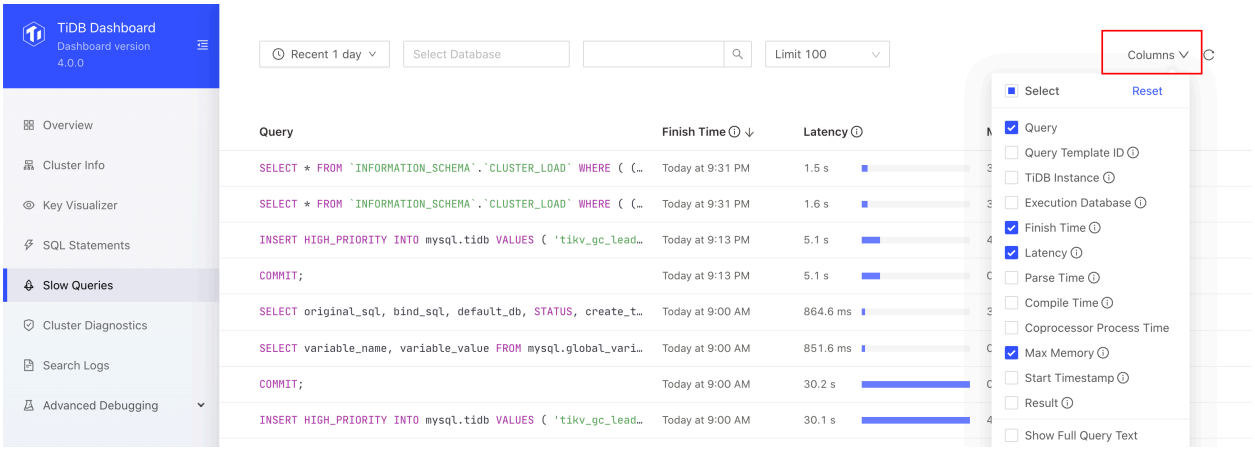


图 424: 显示更多列信息

修改列表排序依据

列表默认以结束运行时间 (Finish Time) 逆序排序，点击不同的列标题可以修改排序依据或切换排序顺序：

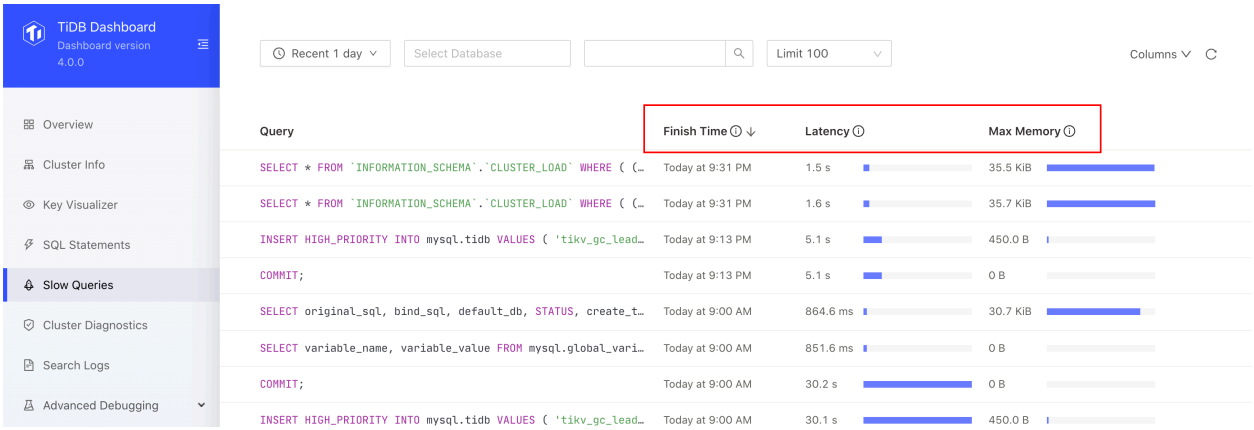


图 425: 修改列表排序依据

12.12.1.9.2 查看执行详情

在列表中点击任意一行可以显示该慢查询的详细执行信息，包含：

- SQL：慢查询 SQL 文本（下图中区域 1）
- 执行计划：慢查询的执行计划，参阅理解 TiDB 执行计划文档了解如何解读执行计划（下图中区域 2）
- 其他分类好的 SQL 执行信息（下图中区域 3）

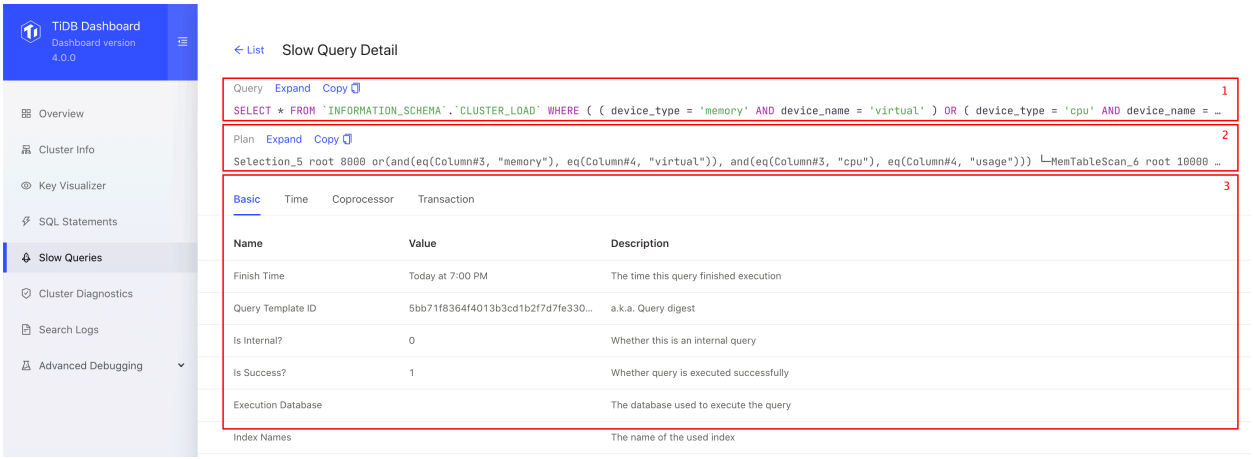


图 426: 查看执行详情

点击展开 (Expand) 链接可以展开相应项的完整内容，点击复制 (Copy) 链接可以复制完整内容到剪贴板。

点击标签页标题可切换显示不同分类的 SQL 执行信息：

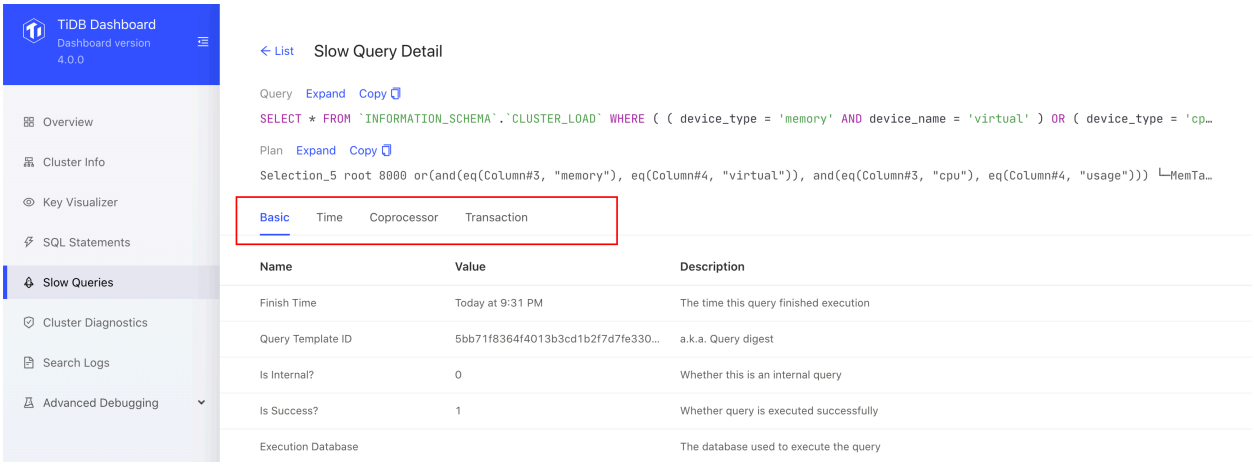


图 427: 显示不同分类执行信息

12.12.1.10 集群诊断页面

12.12.1.10.1 TiDB Dashboard 集群诊断页面

> **警告：**
>
> 该功能目前为实验特性，不建议在生产环境中使用。

集群诊断是在指定的时间范围内，对集群可能存在的问题进行诊断，并将诊断结果和一些集群相关的负载监控信息汇总成一个诊断报告。诊断报告是网页形式，通过浏览器保存后可离线浏览和传阅。

> **注意：**
>
> 集群诊断功能依赖于集群中部署有 Prometheus 监控组件，参见TiUP 或TiDB Ansible 部署文档了解如何部署监控组件。若集群中没有部署监控组件，生成的诊断报告中将提示生成失败。

访问

可以通过以下两种方法访问集群诊断页面：

- 登录后，左侧导航条点击集群诊断（Cluster Diagnose）：

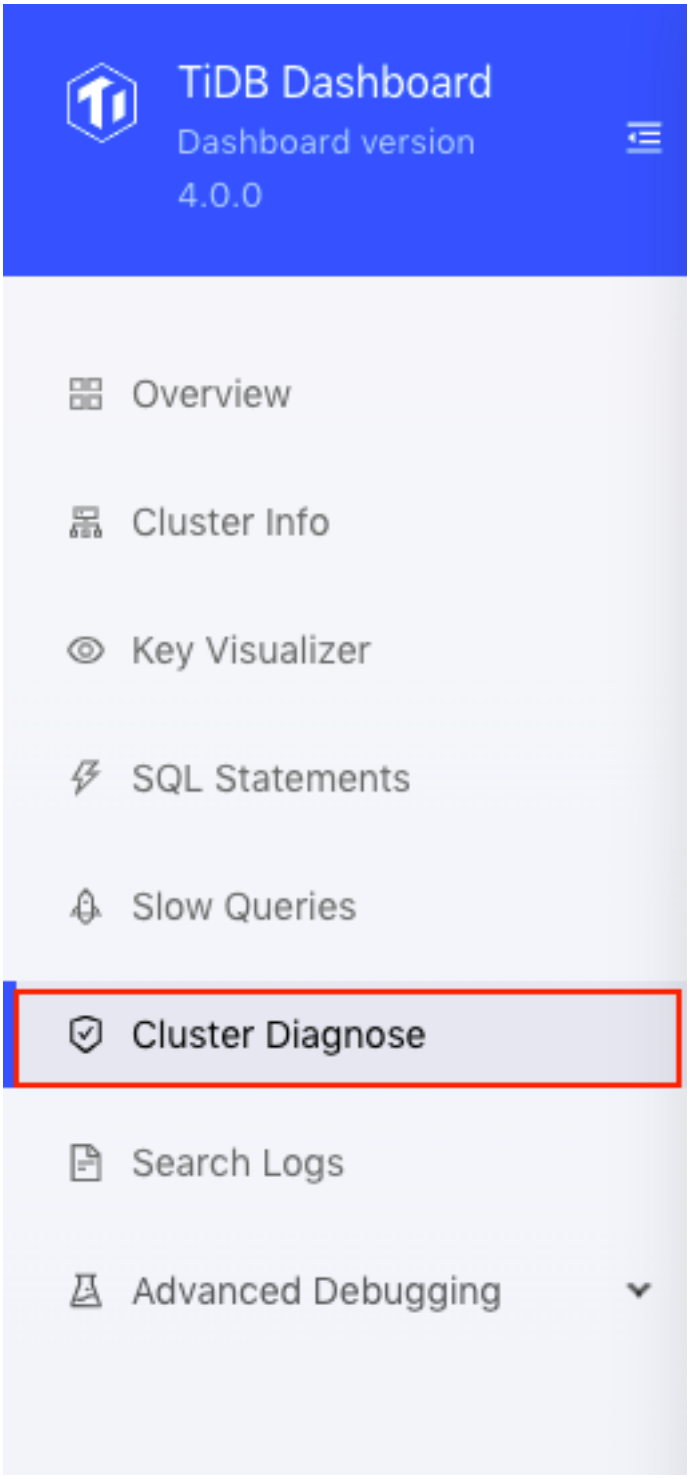


图 428: 访问

- 在浏览器中访问 http://127.0.0.1:2379/dashboard/#/diagnose（将 127.0.0.1:2379 替换为任意实际 PD 地址和端口）。

生成诊断报告

如果想对一个时间范围内的集群进行诊断，查看集群的负载等情况，可以使用以下步骤来生成一段时间范围的诊断报告：

1. 设置区间的开始时间，例如 2020-05-21 14:40:00。
2. 设置区间长度。例如 10 min 。
3. 点击开始。

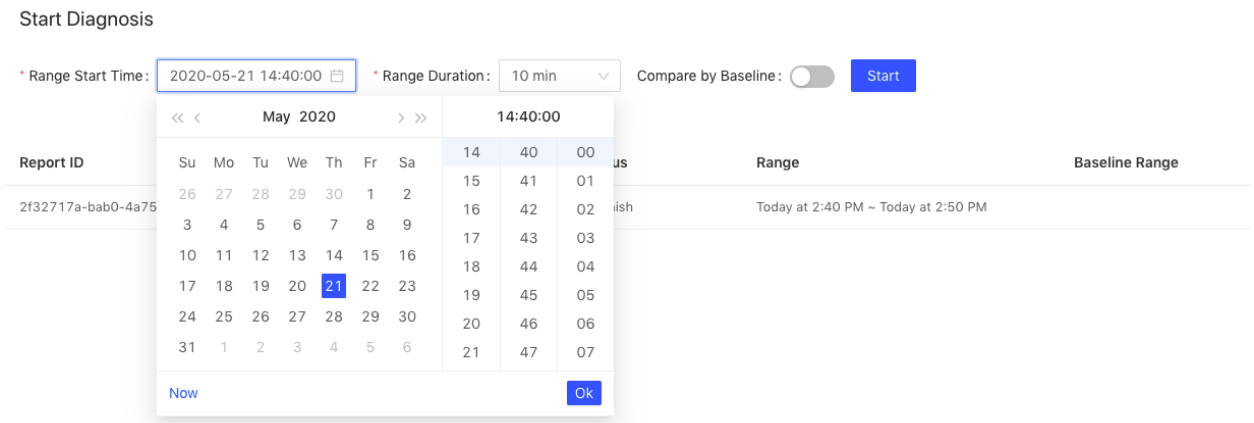


图 429: 生成单个时间段的诊断报告

> **注意：**
>
> 建议生成报告的时间范围在 1 min ~ 60 min 内，目前不建议生成超过 1 小时范围的报告。

以上操作会生成 2020-05-21 14:40:00 至 2020-05-21 14:50:00 时间范围的诊断报告。点击开始 (start) 后，会看到以下界面，生成进度 (progress) 是生成报告的进度条，生成报告完成后，点击查看报告 (View Full Report) 即可。

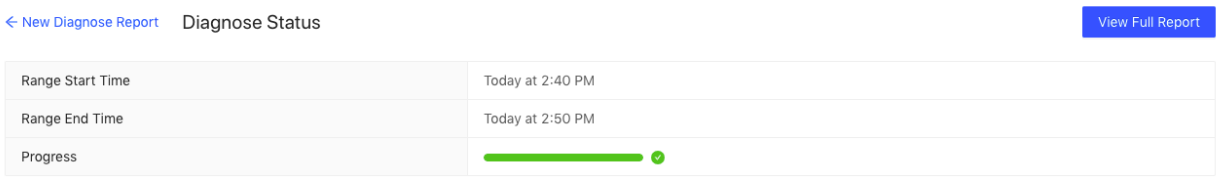


图 430: 生成报告的进度

生成对比诊断报告

如果系统在某个时间点发生异常，如 QPS 抖动或者延迟变高，可以生成一份异常时间范围和正常时间范围的对比报告，例如：

- 系统异常时间段：2020-05-21 14:40:00 ~ 2020-05-21 14:45:00，系统异常时间。
- 系统正常时间段：2020-05-21 14:30:00 ~ 2020-05-21 14:35:00，系统正常时间。

生成以上两个时间范围的对比报告的步骤如下：

1. 设置区间的开始时间，即异常时间段的开始时间，如 2020-05-21 14:40:00。
2. 设置区间长度。一般只系统异常的持续时间，例如 5 min。
3. 开启与基线区间对比开关。
4. 设置基线开始时间，即想要对比的系统正常时段的开始时间，如 2020-05-21 14:30:00。
5. 点击开始。

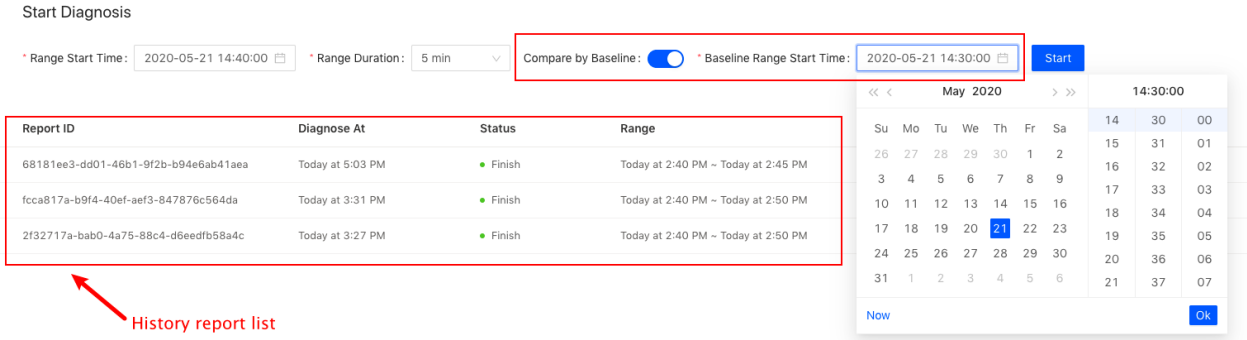


图 431: 生成对比报告

然后同样等报告生成完成后点击查看报告 (View Full Report) 即可。

另外，已生成的诊断报告会显式在诊断报告主页的列表里面，可以点击查看之前生成的报告，不用重复生成。

12.12.1.10.2 TiDB Dashboard 诊断报告

本文档主要介绍诊断报告的内容以及查看技巧，访问集群诊断和生成报告请参考诊断报告访问文档。

查看报告

诊断报告由以下几部分组成：

- 基本信息：包括生成报告的时间范围，集群的硬件信息，集群的拓扑版本信息。
- 诊断信息：显示自动诊断的结果。
- 负载信息：包括服务器，TIDB/PD/TiKV 相关的 CPU、内存等负载信息。
- 概览信息：包括 TiDB/PD/TiKV 的各个模块的耗时信息和错误信息。
- TiDB/PD/TiKV 监控信息：包括各个组件的监控信息。
- 配置信息：包括各个组件的配置信息。

报告中报表示例如下：

Total Time Consume

The table contain the event time consume in TiDB/TiKV/PD. METRIC_NAME is the event name; LABEL is the event label, such as instance, event type ...; TIME_RATIO is the TOTAL_TIME of this event devide by the TOTAL_TIME of upper event which TIME_RATIO is 1; TOTAL_TIME is the total time cost of this event; TOTAL_COUNT is the total count of this event; P999 is the max time of 0.999 quantile; P99 is the max time of 0.99 quantile; P90 is the max time of 0.90 quantile; P80 is the max time of 0.80 quantile;

METRIC_NAME	LABEL	TIME_RATIO	TOTAL_TIME	TOTAL_COUNT	P999	P99	P90	P80
tidb_query expand		1	23223.86	459625	63.54	45.55	0.1	0.06
tidb_get_token fold		0.000001	0.03	458888	0.00004	0.000001	0.000001	0.000001
|-- tidb_get_token	10.0.1.13:10080	0.000001	0.03	458888	0.00004	0.000001	0.000001	0.000001

图 432: 示例报表

上图中，最上面蓝框内的 Total Time Consume 是报表名。下方红框内的内容是对该报表意义的解释，以及报表中各个字段的含义。

报表中小按钮图标的解释如下：

- i 图标：鼠标移动到 i 图标处会显示该行的说明注释。
- expand：点击 expand 会看到这项监控更加详细的信息。如是上图中 `tidb_get_token` 的详细信息包括各个 TiDB 实例的延迟监控信息。
- fold：和 expand 相反，用于把监控的详细信息折叠起来。

所有监控基本上和 TiDB Grafna 监控面板上的监控内容相对应，发现某个模块异常后，可以在 TiDB Grafna 监控面板上查看更多详细的监控信息。

另外，报表中统计的 `TOTAL_TIME` 和 `TOTAL_COUNT` 由于是从 Prometheus 读取的监控数据，其统计会有一些计算上的精度误差。

以下介绍诊断报告的各部分内容。

基本信息

Report Time Range

Report Time Range 表显示生成报告的时间范围，包括开始时间和结束时间。

Report Time Range

START_TIME	END_TIME
2020-05-21 06:40:00	2020-05-21 06:50:00

图 433: report time range 报表

Cluster Hardware

Cluster Hardware 表显示集群中各服务器的硬件信息，包括 CPU、Memory、磁盘等信息。

Cluster Hardware

HOST	INSTANCE	CPU_CORES	MEMORY (GB)	DISK (GB)	UPTIME (DAY)
10.0.1.14	tiflash*1	20/40	125.64	nvme01: 498.638	2.9132028513567314
10.0.1.11	pd*1	20/40	125.64	sda1: 965.834	2.921851028772416
10.0.1.12	tikv*1	20/40	125.64	nvme0n1: 122.78 sda1: 965.834	2.9197623066052247
10.0.1.13	tidb*1	20/40	125.64	sda1: 498.638	2.917983893669314

图 434: Cluster Hardware 报表

上表中各个字段含义如下：

- HOST：服务器的 IP 地址。
- INSTANCE：该服务器部署的实例数量，如 pd * 1 代表这台服务器部署了 1 个 PD 实例。如 tidb * 2 pd * 1 表示这台服务器部署了 2 个 TiDB 实例和 1 个 PD 实例。
- CPU_CORES：表示服务器 CPU 的核心数，物理核心/逻辑核心。
- MEMORY：表示服务器的内存大小，单位是 GB。
- DISK：表示服务器磁盘大小，单位是 GB。
- UPTIME：服务器的启动时间，单位是 DAY。

Cluster Info

Cluster Info 为集群拓扑信息。表中信息来自 TiDB 的information_schema.cluster_info 系统表。

Cluster Info

TYPE	INSTANCE	STATUS_ADDRESS	VERSION	GIT_HASH	START_TIME	UPTIME
pd	10.0.1.11:2379	10.0.1.11:2379	4.1.0-alpha	a80b99ef0d70807d106bdc1b7c6e97a118679c7e	2020-05-18T13:47:01Z	65h40m54.520385474s
tidb	10.0.1.13:4000	10.0.1.13:10080	4.0.0-beta.2	ac30f5322e253125002663ad7c789807acfc9305	2020-05-18T13:47:10Z	65h40m45.520383242s
tiflash	10.0.1.14:3930	10.0.1.14:20292	v4.1.0-alpha-37-gacf3f673d	acf3f673dfbc3e6ffff0dda575760670993fefa6	2020-05-18T13:47:18Z	65h40m37.520390633s
tikv	10.0.1.12:20160	10.0.1.12:20180	4.1.0-alpha	6b09cadbf55311ac07fc51e1b43e3b57b54e71f2	2020-05-18T13:47:04Z	65h40m51.520389948s

图 435: Cluster Info 报表

上表中各个字段含义如下：

- TYPE：节点类型。
- INSTANCE：实例地址，为 IP:PORT 格式的字符串。
- STATUS_ADDRESS：HTTP API 的服务地址。
- VERSION：对应节点的语义版本号。
- GIT_HASH：编译节点版本时的 Git Commit Hash，用于识别两个节点是否是绝对一致的版本。
- START_TIME：对应节点的启动时间。
- UPTIME：对应节点已经运行的时间。

诊断信息

TiDB 内置自动诊断的结果，具体各字段含义以及介绍可以参考information_schema.inspection_result 系统表的内容。

负载信息

Node Load Info

Node Load Info 表显示服务器节点的负载信息，包括时间范围内，服务器以下指标的平均值 (AVG)、最大值 (MAX)、最小值 (MIN)：

- CPU 使用率，最大值是 100%
- 内存使用率
- 磁盘 I/O 使用率
- 磁盘写延迟
- 磁盘读延迟
- 磁盘每秒的读取字节数
- 磁盘每秒的写入字节数
- 节点网络每分钟收到的字节数
- 节点网络每分钟发送的字节数
- 节点正在使用的 TCP 连接数
- 节点所有的 TCP 连接数

Node Load Info

METRIC_NAME	instance	AVG	MAX	MIN
node_cpu_usage expand		40.42%	99.6%	6.98%
node_mem_usage expand		66.9%	96.28%	33.65%
node_disk_io_utilization expand		16%	97%	0%
Disk write latency expand		4000 us	50 ms	0 us
Disk read latency expand		2000 us	6000 us	0 us
tikv_disk_read_bytes expand		605.980 KB	15.842 MB	0.000 KB
tikv_disk_write_bytes expand		1.129 MB	12.235 MB	0.000 KB
node_network_in_traffic expand		225.706 KB	1.427 MB	0.000 KB
node_network_out_traffic expand		165.807 KB	1.659 MB	0.000 KB
node_tcp_in_use expand		19	47	5
node_tcp_connections expand		30	61	6

图 436: Node Load Info 报表

Instance CPU Usage

Instance CPU Usage 表显示各个 TiDB/PD/TiKV 进程的 CPU 使用率的平均值 (AVG)，最大值 (MAX)，最小值 (MIN)，这里进程 CPU 使用率最大值是 100% * CPU 逻辑核心数。

INSTANCE	JOB	AVG	MAX	MIN
172.16.5.40:10089	tidb	1891%	1906%	1878%
172.16.5.40:22151	tikv	517%	571%	459%
172.16.5.40:20151	tikv	498%	562%	444%
172.16.5.40:23151	tikv	352%	399%	311%
172.16.5.40:24799	pd	39%	45%	36%

图 437: Instance CPU Usage 报表

Instance Memory Usage

Instance Memory Usage 表显示各个 TiDB/PD/TiKV 进程占用内存字节数的平均值 (AVG)，最大值 (MAX)，最小值 (MIN)。

INSTANCE	JOB	AVG	MAX	MIN
172.16.5.40:20151	tikv	9.562 GB	9.605 GB	9.523 GB
172.16.5.40:23151	tikv	9.528 GB	9.574 GB	9.496 GB
172.16.5.40:22151	tikv	9.433 GB	9.498 GB	9.345 GB
172.16.5.40:10089	tidb	904.500 MB	911.660 MB	894.336 MB
172.16.5.40:24799	pd	61.089 MB	61.270 MB	60.980 MB

图 438: Instance Memory Usage 报表

TiKV Thread CPU Usage

TiKV Thread CPU Usage 表显示 TiKV 内部各个模块线程的 CPU 使用率的平均值 (AVG)、最大值 (MAX)、和最小值 (MIN)。这里进程 CPU 使用率最大值为 100% * 对应配置的线程数量。

TiKV Thread CPU Usage

METRIC_NAME	INSTANCE	AVG	MAX	MIN	CONFIG_KEY	CURRENT_CONFIG_VALUE
grpc expand		14.83%	16%	12.96%		
schedule worker fold		8.76%	9.47%	7.56%		
|-- sched_worker	10.0.1.12:20180	8.76%	9.47%	7.56%	storage.scheduler-worker-pool-size	4
storage read pool expand		3.24%	3.64%	2.67%		
storage read pool normal expand		3.22%	3.64%	2.67%		
Async apply fold		3.14%	3.4%	2.71%		
|-- Async apply	10.0.1.12:20180	3.14%	3.4%	2.71%	raftstore.apply-pool-size	2
raftstore expand		2.7%	2.93%	2.38%		
rocksdb expand		0.54%	2.13%	0%		
unified read pool expand		0.45%	4.24%	0%		
split_check expand		0.26%	0.84%	0%		
gc expand		0.04%	0.4%	0%		
storage read pool high expand		0.02%	0.04%	0%		
snapshot expand		0.004%	0.02%	0%		
cop normal						
cop high						
cop low						
storage read pool low expand		0%	0%	0%		
cop						

图 439: TiKV Thread CPU Usage 报表

上表中的字段解释如下：

- CONFIG_KEY：表示对应模块的相关线程数配置。
- CURRENT_CONFIG_VALUE：表示配置在生成报表时刻的当前值。

> **注意：**
>
> `CURRENT_CONFIG_VALUE` 是生成报告时的值，并不是报告时间范围内的值。目前不能获取历史时间某些配置的值。

TiDB/PD Goroutines Count

TiDB/PD Goroutines Count 表显示 TiDB/PD goroutines 数量的平均值 (AVG)，最大值 (MAX)，和最小值 (MIN)。如果 goroutines 数量超过 2000，说明该进程并发太高，会对整体请求的延迟有影响。

INSTANCE	JOB	AVG	MAX	MIN
172.16.5.40:10089	tidb	1434.33	1473	1394
172.16.5.40:24799	pd	157	157	157

图 440: TiDB/PD goroutines count 报表

概览信息

Time Consumed by Each Component

Time Consumed by Each Component 显示包括集群中 TiDB、PD、TiKV 各个模块的监控耗时以及各项耗时的占比。默认时间单位是秒。用户可以用该表快速定位哪些模块的耗时较多。

METRIC_NAME	LABEL	TIME_RATIO	TOTAL_TIME	TOTAL_COUNT	P999	P99	P90	P80
tidb_query expand		1	23223.86	459625	63.54	45.55	0.1	0.06
tidb_get_token expand		0.000001	0.03	458888	0.00004	0.000001	0.000001	0.000001
tidb_parse expand		0.00001	0.29	4603	0.0006	0.0004	0.0002	0.0001
tidb_compile expand		0.0008	17.82	463488	0.001	0.0009	0.0003	0.0002
tidb_execute expand		1	23237.39	463491	90.81	0.35	0.1	0.08
tidb_distsql_execution expand		0.0002	4.53	1064	0.13	0.1	0.02	0.01
tidb_cop expand		0.0005	12.39	1075	2.03	1.84	0.02	0.01
Transaction expand		1	23158.73	460023	8.16	7.91	5.38	0.06
tidb_transaction_local_latch_wait		0	0	0				
tidb_kv_backoff expand		0.00003	0.58	291	0.002	0.002	0.002	0.002
KV request expand		1.65	38325.58	2300949	2.03	1.84	0.04	0.03
Slow query expand		0.01	323.65	280	65.5	65.21	62.26	58.98
tidb_slow_query_cop_process		0	0	280	0.001	0.001	0.0009	0.0008
tidb_slow_query_cop_wait		0	0	280	0.001	0.001	0.0009	0.0008
DDL job		0	0	0				
tidb_ddl_worker		0	0	0				
tidb_ddl_update_self_version		0	0	0				
tidb_owner_handle_syncer		0	0	0				
Batch add index								
tidb_ddl_deploy_syncer		0	0	0				
Schema load expand		0.000008	0.19	27	0.06	0.06	0.06	0.05
TiDB meta operation expand		0.0001	3.21	1200	0.13	0.1	0.01	0.005
TiDB auto id request		0	0	0				
TiDB auto analyze expand		0.003	78.35	1	81.88	81.51	77.82	73.73
TiDB GC expand		0.000000001	0.00002	3	1	0.99	0.9	0.8
tidb_gc_push_task		0	0	0				
tidb_batch_client_unavailable		0	0	0				
tidb_batch_client_wait		0	0	0				
pd_tso_rpc expand		0.005	123.45	108540	0.02	0.008	0.003	0.002
pd_tso_wait expand		0.07	1662.29	920545	0.03	0.01	0.004	0.003
PD client cmd expand		0.15	3370.87	2761712	0.03	0.01	0.004	0.003
pd_client_request_rpc expand		0.005	123.45	108540	0.02	0.008	0.003	0.002
pd_grpc_completed_commands expand		0.00005	1.24	5255	0.01	0.01	0.006	0.005
pd_operator_finish								
pd_operator_step_finish								
Etcd transactions expand		0.000008	0.17	207	0.008	0.008	0.006	0.002
pd_region_heartbeat expand		0.00006	1.33	195	0	0	0	0
etcd_wal_fsync expand		0.00001	0.33	288	0.02	0.01	0.006	0.004
pd_peer_round_trip								

图 441: Total Time Consume 报表

上表各列的字段含义如下：

- `METRIC_NAME`：监控项的名称。
- `Label`：监控的 label 信息，点击 expand 后可以查看该项监控更加详细的各项 label 的监控信息。
- `TIME_RATIO`：该项为 `TIME_RATIO` 为 1 的监控行总时间与监控消耗的总时间的比例。如 `kv_request` 的总耗时占 `tidb_query` 总耗时的 `1.65 = 38325.58/23223.86`。因为 KV 请求会并行执行，所以所有 KV 请求的总时间有可能超过总查询 (`tidb_query`) 的执行时间。
- `TOTAL_TIME`：该项监控的总耗时。

- TOTAL_COUNT：该项监控执行的总次数。
- P999：该项监控的 P999 最大时间。
- P99：该项监控的 P99 最大时间。
- P90：该项监控的 P90 最大时间。
- P80：该项监控的 P80 最大时间。

以上监控中相关模块的耗时关系如下所示：

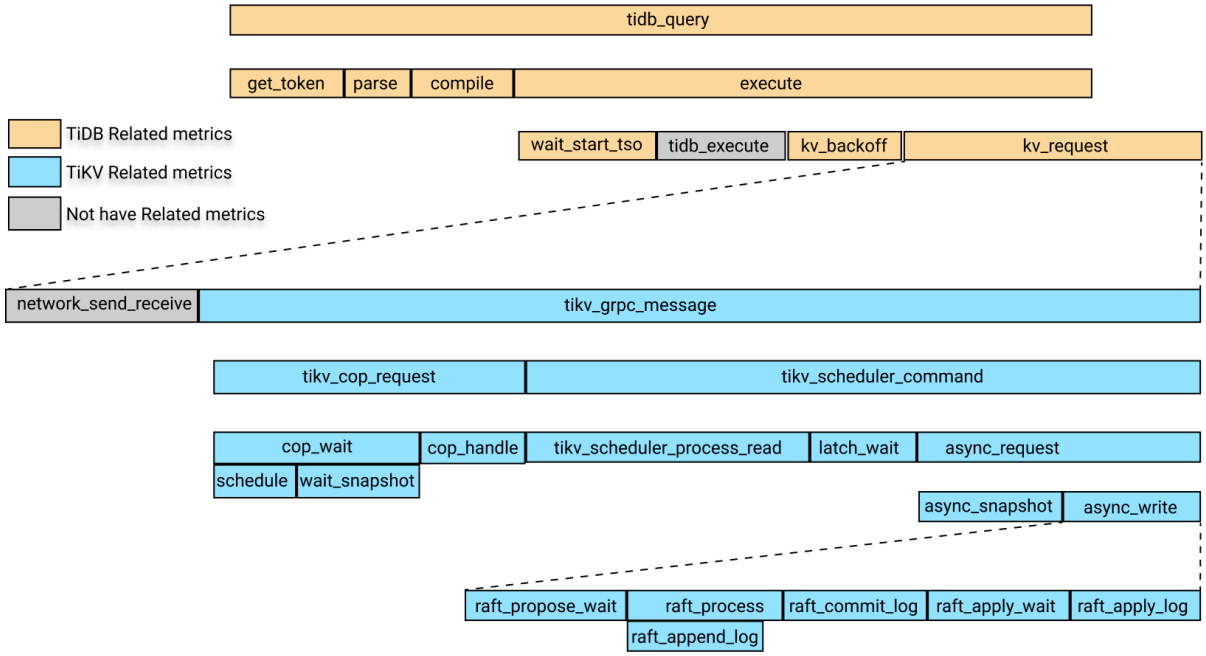


图 442: 各个模块耗时关系图

上图中，黄色部分是 TiDB 相关的监控，蓝色部分是 TiKV 相关的监控，灰色部分暂时没有具体对应的监控项。

上图中，tidb_query 的耗时包括以下部分的耗时：

- get_token
- parse
- compile
- execute

其中 execute 的耗时包括以下部分：

- wait_start_tso
- TiDB 层的执行时间，目前暂无监控
- KV 请求的时间
- tidb_kv_backoff 的时间，这是 KV 请求失败后进行 backoff 的时间

其中，KV 请求时间包含以下部分：

- 请求的网络发送以及接收耗时，目前该项暂无监控，可以大致用 KV 请求时间减去 `tikv_grpc_message` 的时间
- `tikv_grpc_message` 的耗时

其中，`tikv_grpc_message` 耗时包含以下部分：

- Coprocessor request 耗时，指用于处理 COP 类型的请求，该耗时包括以下部分：
  - `tikv_cop_wait`，请求排队等待的耗时
  - `Coprocessor handling request`，处理 COP 请求的耗时
- `tikv_scheduler_command` 耗时，该耗时包含以下部分：
  - `tikv_scheduler_processing_read`，处理读请求的耗时
  - `tikv_storage_async_request` 中获取 snapshot 的耗时（snapshot 是该项监控的 label）
  - 处理写请求的耗时，该耗时包括以下部分：
    * `tikv_scheduler_latch_wait`，等待 latch 的耗时
    * `tikv_storage_async_request` 中 write 的耗时（write 是该监控的 label）

其中，`tikv_storage_async_request` 中的 write 耗时是指 raft kv 写入的耗时，包括以下部分：

- `tikv_raft_propose_wait`
- `tikv_raft_process`，该耗时主要时间包括：
  - `tikv_raft_append_log`
  - `tikv_raft_commit_log`
  - `tikv_raft_apply_wait`
  - `tikv_raft_apply_log`

用户可以根据上述耗时之间的关系，利用 `TOTAL_TIME` 以及 P999，P99 的时间大致定位哪些模块耗时比较长，然后再看相关的监控。

> 注意：
>
> 由于 Raft KV 可能会将多个请求作为一个 batch 来写入，所以 `TOTAL_TIME` 不适用于来衡量 Raft KV 的写入相关监控项的耗时，这些监控项具体是 `tikv_raft_process`、`tikv_raft_append_log`、`tikv_raft_commit_log`、`tikv_raft_apply_wait`、`tikv_raft_apply_log`。此时用 P999 和 P99 的时间来对比各个模块的耗时更加合理。
>
> 原因是，假如有 10 个 async write 请求，Raft KV 内部将 10 个请求打包成一个 batch 执行，执行时间为 1 秒，所以每个请求的执行时间为 1 秒，10 个请求的总时间是 10 秒。但是 Raft KV 处理的总时间是 1 秒。如果用 `TOTAL_TIME` 来衡量，用户可能不明白剩余的 9 秒耗时在哪些模块下。这里从总请求数 (`TOTAL_COUNT`) 也能看出 Raft KV 的监控和其他监控的差异。

Errors Occurred in Each Component

Errors Occurred in Each Component 表显示包括 TiDB 和 TiKV 出现错误的总数。例如写 binlog 失败、`tikv server is busy`、`TiKV channel full`、`tikv write stall` 等错误，具体各项错误含义可以看行注释。

METRIC_NAME	LABEL	TOTAL_COUNT
tidb_binlog_error_total_count		0
tidb_handshake_error_total_count		0
tidb_transaction_retry_error_total_count		
tidb_kv_region_error_total_count expand		903
tidb_schema_lease_error_total_count		0
tikv_grpc_error_total_count		0
tikv_critical_error_total_count		
tikv_scheduler_is_busy_total_count		
tikv_channel_full_total_count		
tikv_coprocessor_request_error_total_count expand		1
tikv_engine_write_stall		0
tikv_server_report_failures_total_count expand		1396
tikv_storage_async_request_error expand		1176
tikv_lock_manager_detect_error_total_count		
tikv_backup_errors_total_count		
node_network_in_errors_total_count		0
node_network_out_errors_total_count		0

图 443: Errors Occurred in Each Component 报表

TiDB/PD/TiKV 的具体监控信息

这部分包括了 TiDB/PD/TiKV 更多的具体的监控信息。

TiDB 相关监控信息

Time Consumed by TiDB Component

Time Consumed by TiDB Component 表显示 TiDB 的各项监控耗时以及各项耗时的占比。和Time Consumed by Each Component 表类似，但是这个表的 label 信息会更丰富，细节更多。

TiDB Server Connections

TiDB Server Connections 表显示 TiDB 各个实例的客户端连接数。

TiDB Transaction

TiDB Transaction 表显示 TiDB 事务相关的监控。

METRIC_NAME	LABEL	TOTAL_VALUE	TOTAL_COUNT	P999	P99	P90	P80
tidb_transaction_retry_num		0	0	0	0	0	0
tidb_transaction_statement_num expand		6854588	6852371	4	4	3	2
tidb_txn_region_num expand		1070770	706887	3	2	2	2
tidb_txn_kv_write_num expand		513386	181312	234	2	2	2
tidb_txn_kv_write_size expand		266.772 MB	181296	116.913 KB	1.996 KB	1.905 KB	1.805 KB
tidb_load_safepoint_total_num expand		16					
tidb_lock_resolver_total_num expand		420514					

图 444: Transaction 报表

- TOTAL_VALUE：该项监控在报告时间段内所有值的和（SUM）。
- TOTAL_COUNT：该项监控出现的总次数。
- P999: 该项监控的 P999 最大值。
- P99: 该项监控的 P99 最大值。
- P90: 该项监控的 P90 最大值。
- P80: 该项监控的 P80 最大值。

示例：

上表中，在报告时间范围的 tidb_txn_kv_write_size 表示一共约有 181296 次事务的 KV 写入，总 kv 写入大小是 266.772 MB，其中单次事务的 KV 写入的 P999、P99、P90、P80 的最大值分别为 116.913 KB、1.996 KB、1.905 KB、1.805 KB。

DDL Owner

MIN_TIME	DDL OWNER
2020-05-21 14:40:00	10.0.1.13:10080

图 445: TiDB DDL Owner 报表

上表表示从 2020-05-21 14:40:00 开始，集群的 DDL owner 在 10.0.1.13:10080 节点。如果 owner 发生变更，上表会有多行数据，其中 MinTime 列表示已知对应 Owner 的最小时间。

> 注意：
>
> 如果 owner 信息为空，不代表这个时间段内一定没有 owner，因为这里是依靠 ddl_worker 的监控信息来判断 DDL owner 的。也可能是这个时间段内 ddl_worker 没有做任何 DDL job 导致 owner 信息为空。

TiDB 中其他部分监控表如下：

- Statistics Info：TiDB 统计信息的相关监控。
- Top 10 Slow Query：报表时间范围内 Top 10 的慢查询信息。
- Top 10 Slow Query Group By Digest：报表时间范围内 Top 10 的慢查询信息，并按照 SQL 指纹聚合。
- Slow Query With Diff Plan：报表时间范围内执行计划发生变更的 SQL 语句。

PD 相关监控信息

PD 模块相关监控的报表如下：

- Time Consumed by PD Component：PD 中相关模块的耗时监控
- Scheduled Leader/Region：报表时间范围内集群发生的 `balance-region` 和 `balance leader` 监控，比如从 `tikv_note_1` 上调度走了多少个 leader，调度进了多少个 leader。
- Cluster Status：集群的状态信息，包括总 TiKV 数量、总集群存储容量、Region 数量、离线 TiKV 的数量等信息。
- Store Status：记录各个 TiKV 节点的状态信息，包括 Region score、 leader score、 Region/leader 的数量。
- etcd Status：PD 内部的 etcd 相关信息。

TiKV 相关监控信息

TIKV 模块的相关监控报表如下：

- Time Consumed by TiKV Component：TiKV 中相关模块的耗时监控。
- Time Consumed by RocksDB：TiKV 中 RocksDB 的耗时监控。
- TiKV Error：TiKV 中各个模块相关的 error 信息。
- TiKV Engine Size：TiKV 中各个节点 column family 的存储数据大小。
- Coprocessor Info：TiKV 中 Coprocessor 模块相关的监控。
- Raft Info：TiKV 中 Raft 模块的相关监控信息。
- Snapshot Info：TiKV 中 snapshot 相关监控信息。
- GC Info：TiKV 中 GC 相关的监控信息。
- Cache Hit：TiKV 中 Rocksdb 的各个缓存的命中率监控信息。

配置信息

配置信息中，部分模块的配置信息可以显示报告时间范围内的配置值，有部分配置则因为无法获取到历史的配置值，所以是生成报告时刻的当前配置值。

在报告时间范围内，以下表包括部分配置的在报告时间范围的开始时间的值：

- Scheduler Initial Config：PD 调度相关配置在报告开始时间的初始值。
- TiDB GC Initial Config：TiDB GC 相关配置在报告开始时间的初始值。
- TiKV RocksDB Initial Config：TiKV RocksDB 相关配置在报告开始时间的初始值。
- TiKV RaftStore Initial Config：TiKV RaftStore 相关配置在报告开始时间的初始值。

在报表时间范围内，如若有些配置被修改过，以下表包括部分配置被修改的记录：

- Scheduler Config Change History
- TiDB GC Config Change History

- TiKV RocksDB Config Change History
- TiKV RaftStore Config Change History

示例：

APPROXIMATE_CHANGE_TIME	CONFIG_ITEM	VALUE
2020-05-22T20:00:00+08:00	leader-schedule-limit	4
2020-05-22T20:07:00+08:00	leader-schedule-limit	8

图 446: Scheduler Config Change History 报表

上面报表显示，`leader-schedule-limit` 配置参数在报告时间范围内有被修改过：

- `2020-05-22T20:00:00+08:00`，即报告的开始时间 `leader-schedule-limit` 的配置值为 `4`，这里并不是指该配置被修改了，只是说明在报告时间范围的开始时间其配置值是 4。
- `2020-05-22T20:07:00+08:00`，`leader-schedule-limit` 的配置值为 `8`，说明在 `2020-05-22T20:07:00+08:00` 左右，该配置的值被修改了。

下面的报表是生成报告时，TiDB、PD、TiKV 的在生成报告时刻的当前配置：

- TiDB’s Current Config
- PD’s Current Config
- TiKV’s Current Config

对比报告

生成两个时间段的对比报告，其内容和单个时间段的报告是一样的，只是加入了对比列显示两个时间段的差别。下面主要介绍对比报告中的一些特有表以及如何查看对比报表。

首先在基本信息中的 Compare Report Time Range 报表会显示出对比的两个时间段：

Compare Report Time Range

T1.START_TIME	T1.END_TIME	T2.START_TIME	T2.END_TIME
2020-05-21 06:30:00	2020-05-21 06:35:00	2020-05-21 06:40:00	2020-05-21 06:45:00

图 447: Compare Report Time Range 报表

其中 `t1` 是正常时间段，或者叫参考时间段，`t2` 是异常时间段。

下面是一些慢查询相关的报表：

- Slow Queries In Time Range t2：仅出现在 `t2` 时间段但没有出现在 `t1` 时间段的慢查询。
- Top 10 slow query in time range t1：`t1` 时间段的 Top10 慢查询。
- Top 10 slow query in time range t2：`t2` 时间段的 Top10 慢查询。

DIFF_RATIO 介绍

本部分以 Instance CPU Usage 为例介绍 DIFF_RATIO。

Instance CPU Usage

INSTANCE	JOB	t1.AVG	t1.MAX	t1.MIN	t2.AVG	t2.MAX	t2.MIN	AVG_DIFF_RATIO	MAX_DIFF_RATIO	MIN_DIFF_RATIO
172.16.5.40:10089	tidb	410%	410%	410%	1240%	1240%	1240%	2.02	2.02	2.02
172.16.5.40:20151	tikv	221%	221%	221%	617%	617%	617%	1.79	1.79	1.79
172.16.5.40:20379	pd	82%	82%	82%	78%	78%	78%	-0.05	-0.05	-0.05

图 448: Compare Instance CPU Usage 报表

- t1.AVG、t1.MAX、t1.Min 分别是 t1 时间段内 CPU 使用率的平均值、最大值、最小值。
- t2.AVG、t2.MAX、t2.Min 分别是 t2 时间段内 CPU 使用率的平均值、最大值、最小值。
- AVG_DIFF_RATIO 表示 t1 和 t2 时间段平均值的 DIFF_RATIO。
- MAX_DIFF_RATIO 表示 t1 和 t2 时间段最大值的 DIFF_RATIO。
- MIN_DIFF_RATIO 表示 t1 和 t2 时间段最小值的 DIFF_RATIO。

DIFF_RATIO：表示两个时间段的差异大小，有以下几个取值方式：

- 如果该监控仅在 t2 时间内才有值，t1 时间段没有，则 DIFF_RATIO 取值为 1。
- 如果监控项仅在 t1 时间内才有值，t1 时间段没有，则 DIFF_RATIO 取值为 -1。
- 如果 t2 时间段的值比 t1 时间段的值大，则 DIFF_RATIO = (t2.value / t1.value)- 1。
- 如果 t2 时间段的值比 t1 时间段的值小，则 DIFF_RATIO = 1 - (t1.value / t2.value)。

例如上表中，tidb 节点的平均 CPU 使用率在 t2 时间段比 t1 时间段高 2.02 倍，2.02 = 1240/410 - 1。

Maximum Different Item 报表介绍

Maximum Different Item 的报表是对比两个时间段的监控项后，按照监控项的差异大小排序，通过这个表可以很快发现两个时间段哪些监控的差异最大。示例如下：

Maximum Different Item

The maximum different metrics between two time ranges.

TABLE	METRIC_NAME	LABEL	MAX_DIFF	t1.VALUE	t2.VALUE	VALUE_TYPE
TiKV, coprocessor_info Expand	TiKV Coprocessor scan	172.16.5.40:20151,select,get,lock	28916.85	61.33	1773532	TOTAL_VALUE
TiDB, statistics_info	store_query_feedback_total_count	172.16.5.40:10089,ok	7219.00		7219	TOTAL_COUNT
overview, total_time_consume	tidb_parse	general	1201.00		1201	TOTAL_COUNT
TiDB, tidb_time_consume	tidb_slow_query_cop_wait	172.16.5.40:10089	800.00		800	TOTAL_COUNT
overview, total_time_consume	tidb_slow_query_cop_process	172.16.5.40:10089	800.00		800	TOTAL_COUNT
overview, total_time_consume	Slow query	172.16.5.40:10089	799.00	1	800	TOTAL_COUNT
TiKV, coprocessor_info	TiKV Coprocessor response	172.16.5.40:20151	534.43	15.822 MB	8.273 GB	TOTAL_VALUE
overview, total_time_consume	tidb_distsql_execution	dag	470.06	0.68	320.32	TOTAL_TIME
load, tikv_thread_cpu_usage	unified read pool	172.16.5.40:20151	412.29	0.07%	28.93%	MIN
TiKV, coprocessor_info Expand	TiKV Coprocessor scan keys	172.16.5.40:20151,select	198.39	1067055.38	212755381.33	TOTAL_VALUE
overview, total_time_consume Expand	tidb_ddl_worker	drop view	139.00		139	TOTAL_COUNT
overview, total_error Expand	tikv_storage_async_request_error	snapshot	89.00		89	TOTAL_COUNT
overview, total_time_consume Expand	Coprocessor handling request	index	78.99	0.67	53.59	TOTAL_TIME
overview, total_time_consume	tidb_cop	172.16.5.40:10089	75.11	5.93	451.31	TOTAL_TIME
overview, total_time_consume	KV request	Cop	75.10	5.93	451.26	TOTAL_TIME
overview, total_time_consume Expand	Coprocessor request	select	73.12	4.73	350.6	TOTAL_TIME
PD, pd_time_consume Expand	pd_region_heartbeat	172.16.5.40:20150,1	-73.00	73		TOTAL_COUNT
overview, total_time_consume Expand	tikv_grpc_messge	coprocessor	71.75	5.56	404.49	TOTAL_TIME
TiDB, tidb_time_consume	tidb_ddl_update_self_version	172.16.5.40:10089,ok	71.00		71	TOTAL_COUNT
overview, total_time_consume Expand	TiDB meta operation	update_ddl_job	71.00		71	TOTAL_COUNT
overview, total_time_consume Expand	tidb_owner_handle_syncer	update_global_version	71.00		71	TOTAL_COUNT
TiDB, tidb_time_consume Expand	tidb_query	172.16.5.40:10089,internal	-62.84	15.96	0.25	P999
TiKV, scheduler_info Expand	Scheduler stage	172.16.5.40:20151,scan_lock,snapshot_ok	55.45		55.45	TOTAL_VALUE
overview, total_time_consume Expand	tikv_grpc_messge	kv_scan_lock	55.00		55	TOTAL_COUNT
TiKV, tikv_time_consume Expand	tikv_scheduler_command	172.16.5.40:20151,scan_lock	55.00		55	TOTAL_COUNT
overview, total_time_consume Expand	PD client cmd	get_region	44.00		44	TOTAL_COUNT
overview, total_time_consume Expand	tikv_cop_wait	select	39.40	171	6908	TOTAL_COUNT
TiDB, tidb_time_consume	Schema load	172.16.5.40:10089	37.50	0.02	0.77	TOTAL_TIME
overview, total_time_consume	Schema load	172.16.5.40:10089	30.25	0.008	0.25	P999

图 449: Maximum Different Item 报表

- `Table`：表示这个监控项来自于对比报告中报表，如 `TiKV, coprocessor_info` 表示是 TiKV 组件下的 `coprocessor_info` 报表。
- `METRIC_NAME`：监控项名，点击 expand 可以查看该监控的不同 label 的差异对比。
- `LABEL`：监控项对应的 label。比如 `TiKV Coprocessor scan` 监控项有两个 label，分别是 instance、req、tag、sql_type，分别表示为 TiKV 地址、请求类型、操作类型和操作的 column family。
- `MAX_DIFF`：差异大小，取值为 `t1.VALUE` 和 `t2.VALUE` 的 `DIFF_RATIO` 计算。

可以从上表中发现，`t2` 时间段比 `t1` 时间段多出了大量的 Coprocessor 请求，TiDB 的解析 SQL (parse) 时间也多了很多。

12.12.1.10.3 使用 TiDB Dashboard 诊断报告定位问题

本文介绍如何使用 TiDB Dashboard 诊断报告定位问题。

对比诊断功能示例

对比报告中对比诊断的功能，通过对比两个时间段的监控项差异来尝试帮助 DBA 定位问题。先看以下示例。

大查询/写入导致 QPS 抖动或延迟上升诊断

示例 1

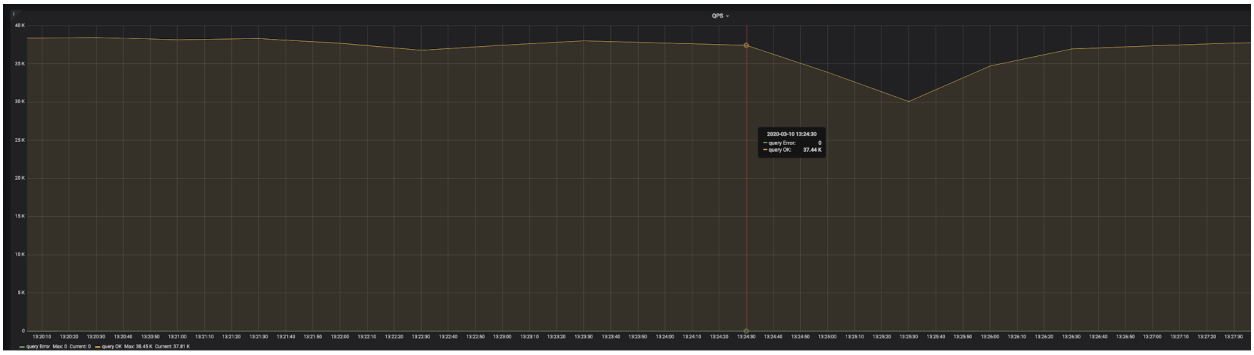


图 450: QPS 图

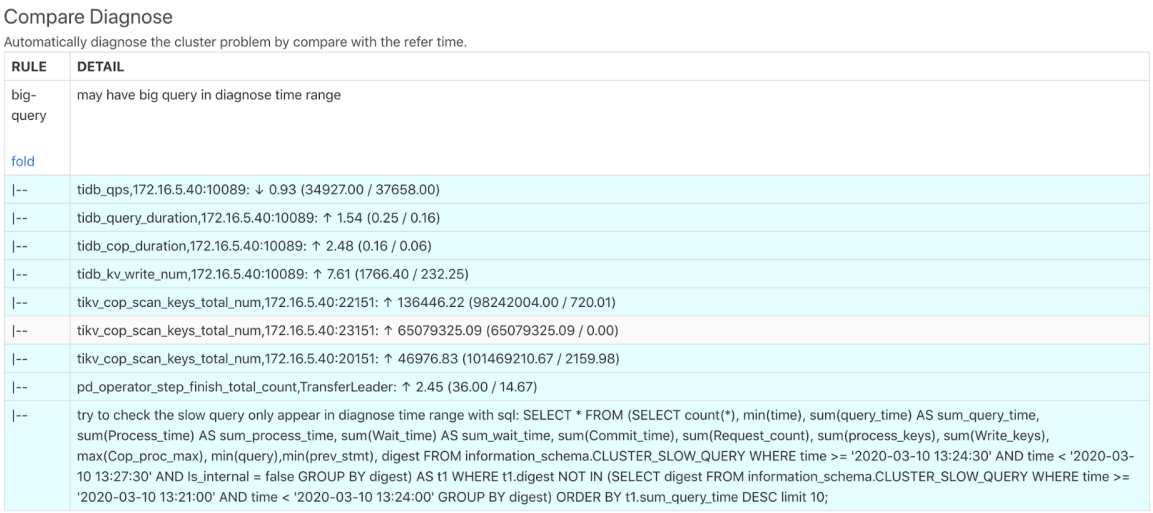
上图是 go-ycsb 压测的监控。可以发现，在 2020-03-10 13:24:30 时，QPS 突然开始下降，3 分钟后，QPS 开始恢复正常。

生成以下两个时间范围的对比报告：

- t1: 2020-03-10 13:21:00 - 2020-03-10 13:23:00，正常时间段，又叫参考时间段。
- t2: 2020-03-10 13:24:30 - 2020-03-10 13:27:30，QPS 开始下降的异常时间段。

这里两个时间间隔都是 3 分钟，因为抖动的影响范围为 3 分钟。因为诊断时会用一些监控的平均值做对比，所有间隔时间太长会导致平均值差异不明显，无法准确定位问题。

生成报告后查看 Compare Diagnose 报表内容如下：



Compare Diagnose

Automatically diagnose the cluster problem by compare with the refer time.

RULE	DETAIL
big-query fold	may have big query in diagnose time range
|--	tidb_qps,172.16.5.40:10089: ↓ 0.93 (34927.00 / 37658.00)
|--	tidb_query_duration,172.16.5.40:10089: ↑ 1.54 (0.25 / 0.16)
|--	tidb_cop_duration,172.16.5.40:10089: ↑ 2.48 (0.16 / 0.06)
|--	tidb_kv_write_num,172.16.5.40:10089: ↑ 7.61 (1766.40 / 232.25)
|--	tikv_cop_scan_keys_total_num,172.16.5.40:22151: ↑ 136446.22 (98242004.00 / 720.01)
|--	tikv_cop_scan_keys_total_num,172.16.5.40:23151: ↑ 65079325.09 (65079325.09 / 0.00)
|--	tikv_cop_scan_keys_total_num,172.16.5.40:20151: ↑ 46976.83 (101469210.67 / 2159.98)
|--	pd_operator_step_finish_total_count,TransferLeader: ↑ 2.45 (36.00 / 14.67)
|--	try to check the slow query only appear in diagnose time range with sql: SELECT * FROM (SELECT count(*), min(time), sum(query_time) AS sum_query_time, sum(Process_time) AS sum_process_time, sum(Wait_time) AS sum_wait_time, sum(Commit_time), sum(Request_count), sum(process_keys), sum(Write_keys), max(Cop_proc_max), min(query),min(prev_stmt), digest FROM information_schema.CLUSTER_SLOW_QUERY WHERE time >= '2020-03-10 13:24:30' AND time < '2020-03-10 13:27:30' AND Is_internal = false GROUP BY digest) AS t1 WHERE t1.digest NOT IN (SELECT digest FROM information_schema.CLUSTER_SLOW_QUERY WHERE time >= '2020-03-10 13:21:00' AND time < '2020-03-10 13:24:00' GROUP BY digest) ORDER BY t1.sum_query_time DESC limit 10;

图 451: 对比诊断结果

上面诊断结果显示，在诊断的时间内可能有大查询，下面的每一行的含义是：

- `tidb_qps`：QPS 下降 0.93 倍。
- `tidb_query_duration`：P999 的查询延迟上升 1.54 倍。
- `tidb_cop_duration`：P999 的 COP 请求的处理延迟上升 2.48 倍。
- `tidb_kv_write_num`：P999 的 tidb 的事务写入 kv 数量上升 7.61 倍。
- `tikv_cop_scan_keys_total_nun`：TiKV 的 coprocessor 扫描 key/value 的数量分别在 3 台 TiKV 上有很大的提升。
- `pd_operator_step_finish_total_count` 中，transfer leader 的数量上升 2.45 倍，说明异常时间段的调度比正常时间段要高。
- 提示可能有慢查询，并提示用 SQL 查询 TiDB 的慢日志。在 TiDB 中执行结果如下：

```
SELECT * FROM (SELECT count(*), min(time), sum(query_time) AS sum_query_time, sum(Process_time)
    ↪ AS sum_process_time, sum(Wait_time) AS sum_wait_time, sum(Commit_time), sum(Request_count
    ↪ ), sum(process_keys), sum(Write_keys), max(Cop_proc_max), min(query),min(prev_stmt),
    ↪ digest FROM information_schema.CLUSTER_SLOW_QUERY WHERE time >= '2020-03-10 13:24:30' AND
    ↪  time < '2020-03-10 13:27:30' AND Is_internal = false GROUP BY digest) AS t1 WHERE t1.
    ↪ digest NOT IN (SELECT digest FROM information_schema.CLUSTER_SLOW_QUERY WHERE time >= '
    ↪ 2020-03-10 13:21:00' AND time < '2020-03-10 13:24:00' GROUP BY digest) ORDER BY t1.
    ↪ sum_query_time DESC limit 10\G
***************************[ 1. row ]***************************
count(*)            | 196
min(time)           | 2020-03-10 13:24:30.204326
sum_query_time      | 46.878509117
sum_process_time    | 265.924
sum_wait_time       | 8.308
sum(Commit_time)    | 0.926820886
sum(Request_count) | 6035
```

```
sum(process_keys)   | 201453000
sum(Write_keys)     | 274500
max(Cop_proc_max)   | 0.263
min(query)          | delete from test.tcs2 limit 5000;
min(prev_stmt)      |
digest              | 24bd6d8a9b238086c9b8c3d240ad4ef32f79ce94cf5a468c0b8fe1eb5f8d03df
```

可以发现，从 13:24:30 开始有一个批量删除的大写入，一共执行了 196 次，每次删除 5000 行数据，总共耗时 46.8 秒。

示例 2

如果大查询一直没执行完，就不会记录慢日志，但仍可以进行诊断，示例如下：

图 452: QPS 图

上图中，也是在跑 go-ycsb 的压测。可以发现，在 2020-03-08 01:46:30 时，QPS 突然开始下降，并且一直没有恢复。

生成以下两个时间范围的对比报告：

- t1: 2020-03-08 01:36:00 - 2020-03-08 01:41:00，正常时间段，又叫参考时间段。
- t2: 2020-03-08 01:46:30 - 2020-03-08 01:51:30，QPS 开始下降的异常时间段。

生成报告后看 Compare Diagnose 报表的内容如下：

Compare Diagnose

Automatically diagnose the cluster problem by compare with the refer time.

RULE	DETAIL
big-query fold	may have big query in diagnose time range
|--	tidb_qps,172.16.5.40:10089: ↓ 0.85 (31881.00 / 37674.00)
|--	tidb_query_duration,172.16.5.40:10089: ↑ 1.35 (0.06 / 0.04)
|--	tidb_cop_duration,172.16.5.40:10089: ↑ 3.78 (0.08 / 0.02)
|--	tidb_kv_write_num,172.16.5.40:10089: ↑ 511.74 (511.74 / 0.00)
|--	tikv_cop_scan_keys_total_num,172.16.5.40:20151: ↑ 1277.21 (6270285.33 / 4909.36)
|--	try to check the slow query only appear in diagnose time range with sql: SELECT * FROM information_schema.cluster_log WHERE type='tidb' AND time >= '2020-03-08 01:46:30' AND time < '2020-03-08 01:51:30' AND level = 'warn' AND message LIKE '%expensive_query%'

图 453: 对比诊断结果

上面诊断结果的最后一行显示可能有慢查询，并提示用 SQL 查询 TiDB 日志中的 expensive query。在 TiDB 中执行结果如下：

```
> SELECT * FROM information_schema.cluster_log WHERE type='tidb' AND time >= '2020-03-08 01:46:30
    ↪ ' AND time < '2020-03-08 01:51:30' AND level = 'warn' AND message LIKE '%expensive_query%
    ↪ '\G
TIME     | 2020/03/08 01:47:35.846
TYPE     | tidb
INSTANCE | 172.16.5.40:4009
LEVEL    | WARN
MESSAGE  | [expensivequery.go:167] [expensive_query] [cost_time=60.085949605s] [process_time=2.52
    ↪ s] [wait_time=2.52s] [request_count=9] [total_keys=996009] [process_keys=996000] [
    ↪ num_cop_tasks=9] [process_avg_time=0.28s] [process_p90_time=0.344s] [process_max_time
    ↪ =0.344s] [process_max_addr=172.16.5.40:20150] [wait_avg_time=0.000777777s] [wait_p90_time
    ↪ =0.003s] [wait_max_time=0.003s] [wait_max_addr=172.16.5.40:20150] [stats=t_wide:pseudo] [
    ↪ conn_id=19717] [user=root] [database=test] [table_ids="[80,80]"] [txn_start_ts
    ↪ =415132076148785201] [mem_max="23583169 Bytes (22.490662574768066 MB)"] [sql="select
    ↪ count(*) from t_wide as t1 join t_wide as t2 where t1.c0>t2.c1 and t1.c2>0"]
```

以上查询结果显示，在 172.16.5.40:4009 这台 TiDB 上，2020/03/08 01:47:35.846 有一个已经执行了 60s 但还没有执行完的 expensive_query。

用对比报告定位问题

诊断有可能是误诊，使用对比报告或许可以帮助 DBA 更快速的定位问题。参考以下示例。

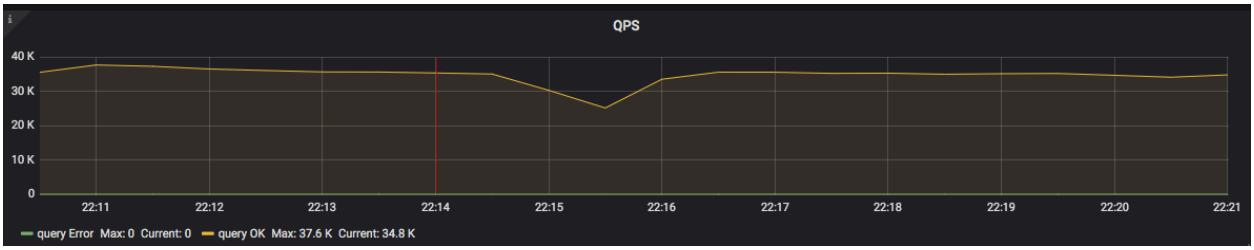


图 454: QPS 图

上图中，也是在跑 go-ycsb 的压测，可以发现，在 2020-05-22 22:14:00 时，QPS 突然开始下降，大概在持续 3 分钟后恢复。

生成以下 2 个时间范围的对比报告：

- t1: 2020-05-22 22:11:00 - 2020-05-22 22:14:00，正常时间段。
- t2: 2020-05-22 22:14:00 - 2020-05-22 22:17:00，QPS 开始下降的异常时间段。

生成对比报告后，查看 Max diff item 报表，该报表对比两个时间段的监控项后，按照监控项的差异大小排序，这个表的结果如下：

Maximum Different Item

The maximum different metrics between two time ranges.

TABLE	METRIC_NAME	LABEL	MAX_DIFF	t1.VALUE	t2.VALUE	VALUE_TYPE
TiDB, transaction	Transaction KV write count	172.16.5.40:10089	-21616.00	43234	2	P999
TiKV, coprocessor_info Expand	TiKV Coprocessor scan	172.16.5.40:20151,select,get,lock	10499.24	64	672015.48	TOTAL_VALUE
TiKV, coprocessor_info Expand	TiKV Coprocessor scan keys	172.16.5.40:20151,select	4264.45	15431.85	65823838.67	TOTAL_VALUE
TiDB, transaction	Transaction KV write size	172.16.5.40:10089	-2702.69	5.278 MB	1.999 KB	P999
overview, total_time_consume	Coprocessor handling request	select	2132.50	0.06	128.01	TOTAL_TIME
TiKV, coprocessor_info	TiKV Coprocessor response	172.16.5.40:20151	1535.00	1.792 MB	2.688 GB	TOTAL_VALUE
TiDB, statistics_info	store_query_feedback_total_count	172.16.5.40:10089,ok	1447.00		1447	TOTAL_COUNT
TiKV, scheduler_info Expand	tikv_scheduler_keys_written	172.16.5.40:20151,commit	-860.00	861	1	P99
TiKV, tikv_time_consume	Coprocessor request	172.16.5.40:20151,select	853.87	0.15	128.23	TOTAL_TIME
load, tikv_thread_cpu_usage	unified read pool	172.16.5.40:20151	667.27	0.11%	73.51%	AVG
TiKV, tikv_time_consume	Coprocessor handling request	172.16.5.40:20151,index	444.00	0.002	0.89	P999
TiKV, tikv_time_consume Expand	tikv_grpc_messge	172.16.5.40:20151,coprocessor	419.97	0.33	138.92	TOTAL_TIME
load, node_load_info	tikv_disk_read_bytes	172.16.5.40:19110,sda	-273.41	0.267 KB	0.000 KB	MAX
overview, total_time_consume Expand	KV request	Cop	244.29	0.66	161.89	TOTAL_TIME

图 455: 对比结果

从上面结果可以看出 t2 时间段新增了很多 Coprocessor 请求，可以猜测可能是 t2 时间段出现了一些大查询，或者是查询较多的负载。

实际上，在 t1 - t2 整个时间段内都在进行 go-ycsb 的压测，然后在 t2 时间段跑了 20 个 tpch 的查询，所以是因为 tpch 大查询导致了出现很多的 cop 请求。

这种大查询执行时间超过慢日志的阈值后也会记录在慢日志里面，可以继续查看 Slow Queries In Time ↪ Range t2 报表查看是否有一些慢查询。一些在 t1 时间段存在的查询，可能在 t2 时间段内就变成了慢查询，是因为 t2 时间段的其他负载影响导致该查询的执行变慢。

12.12.1.11 TiDB Dashboard 日志搜索页面

该页面上允许用户在集群中搜索所有节点上的日志，在页面上预览搜索结果和下载日志。

12.12.1.11.1 访问

登录 Dashboard 后点击左侧导航的日志搜索可以进入此功能页面：

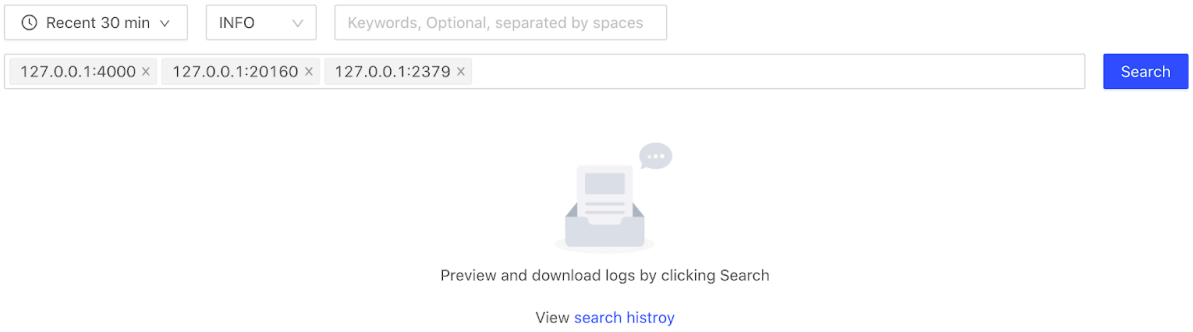


图 456: 日志搜索首页

该页面提供 4 个搜索参数，包括：

- 时间范围：限定搜索时间范围内的日志，默认值为最近 30 分钟
- 日志等级：限定最低日志等级，搜索该日志等级以上的所有日志，默认值为 INFO 等级
- 关键词：任何合法的任何字符串，可选，关键词之间以空格分割，支持正则表达式（大小写不敏感）
- 组件：选择要搜索的集群组件，多选，非空，默认选中所有组件

点击搜索按钮后，会进入搜索结果的详情页。

12.12.1.11.2 搜索结果详情

搜索结果详情页面如下图所示：

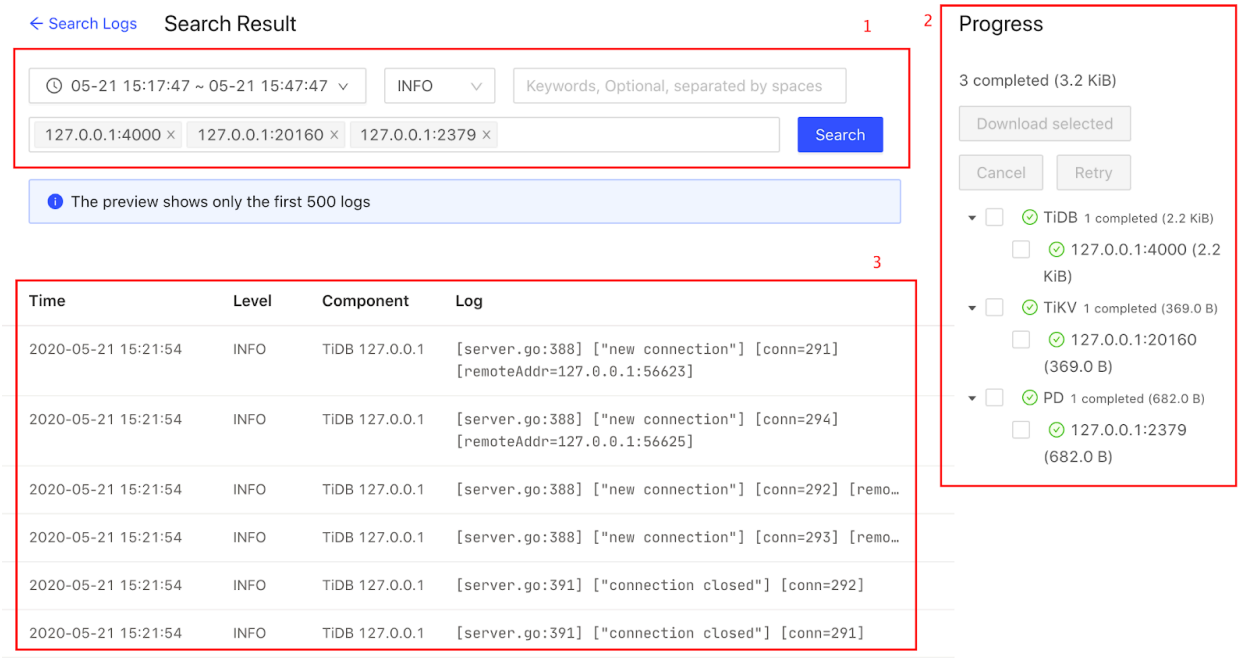


图 457: 搜索结果

整个页面分为三个部分：

- 搜索参数选项：与搜索页面的参数选项相同，可以在表单中重新选择参数并开始一次新的搜索（对应上图中框选的区域 1）
- 搜索进度：在页面右侧，展示当前搜索的进度，包括每个节点日志搜索状态和统计等（对应上图中框选的区域 2）
- 搜索结果（对应上图中框选的区域 3）：
  - 时间：日志产生的时间，时区与前端用户所处时区相同
  - 日志等级：日志等级
  - 组件类型：显示组件名及其地址
  - 日志：每条日志记录的正文部分，不包含日志的时间和日志等级。过长的日志会自动截断，鼠标单击该行可以展开查看完整内容，完整日志最长显示 512 个字符

> 注意：
>
> 在该页面上最多只显示 500 条搜索结果，完整的搜索结果可以通过下载得到。

搜索进度

在搜索进度区域中，对一个节点的搜索称为一次搜索任务，对搜索任务的状态分为：

- 运行中：开始搜索后，所有任务会进入运行中状态

- 成功：任务完成后自动转到成功，此时日志已缓存在 Dashboard 后端所在的本地磁盘中，可以提供给前端下载
- 失败：用户主动取消，或者某种原因报错退出的任务进入失败状态。任务失败时会自动清理本地临时文件

搜索进度区域包含三个控制按钮：

- 下载选中日志：下载被勾选组件的日志（只有已完成的才能被勾选），返回 tar 文件，解压得到一个或多个 zip 文件（每个组件对应一个 zip 文件），解压 zip 得到 log 文本文件
- 取消：取消所有正在运行的任务，只能在有运行中的任务时才能点击
- 重试：重试所有失败的任务，只有在有失败任务且无正在运行的任务时才能点击

12.12.1.11.3 搜索历史列表

在日志搜索首页点击查看搜索历史链接，进入搜索历史列表页面：

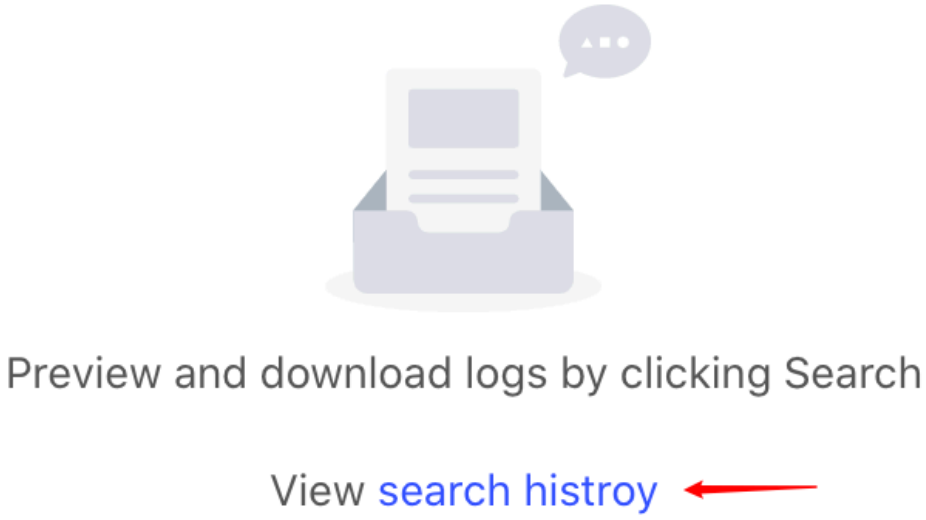


图 458: 搜索历史入口

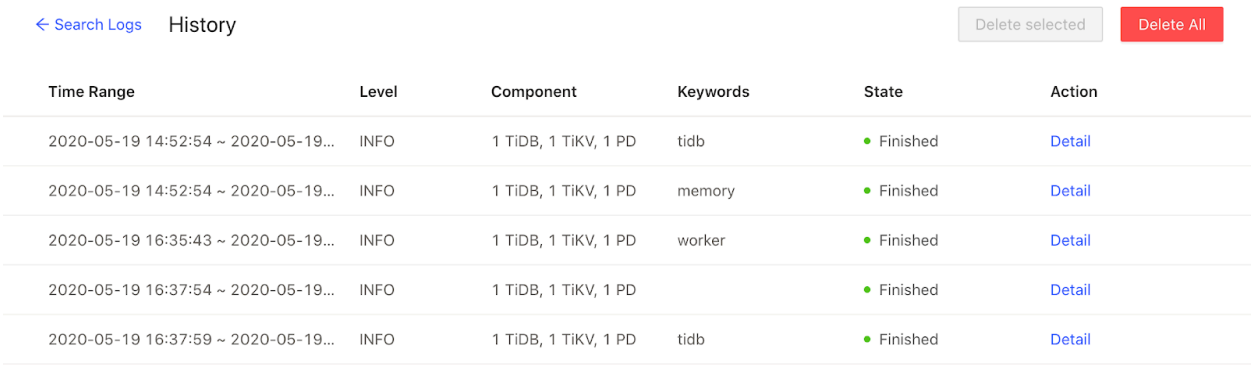


图 459: 搜索历史列表

该列表显示每次搜索的日志的时间范围、日志等级、组件、关键字以及搜索状态等信息。点击操作列的查看详情链接将跳转到此次搜索的结果详情页面。

可以对不再需要的搜索历史执行删除操作，点击右上角的删除全部任务，或者先选中需要删除的某些行后再点击删除选中的任务进行删除：

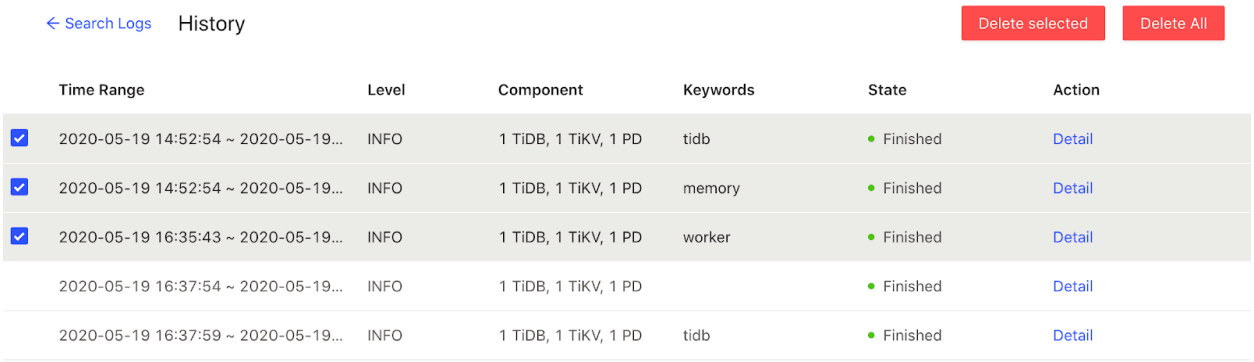


图 460: 删除搜索历史

12.12.1.12 TiDB Dashboard 实例性能分析页面

该功能允许用户对各个 TiDB、TiKV、PD 实例在不重启的情况下进行内部性能数据收集。收集到的性能数据可显示为火焰图或有向无环图，直观展现实例在性能收集的时间段内执行的各种内部操作及比例，快速了解该实例 CPU 资源消耗的主要内容。

12.12.1.12.1 访问

可以通过以下两种方法访问实例性能分析页面：

- 登录后，左侧导航条点击 高级调试 → 实例性能分析：

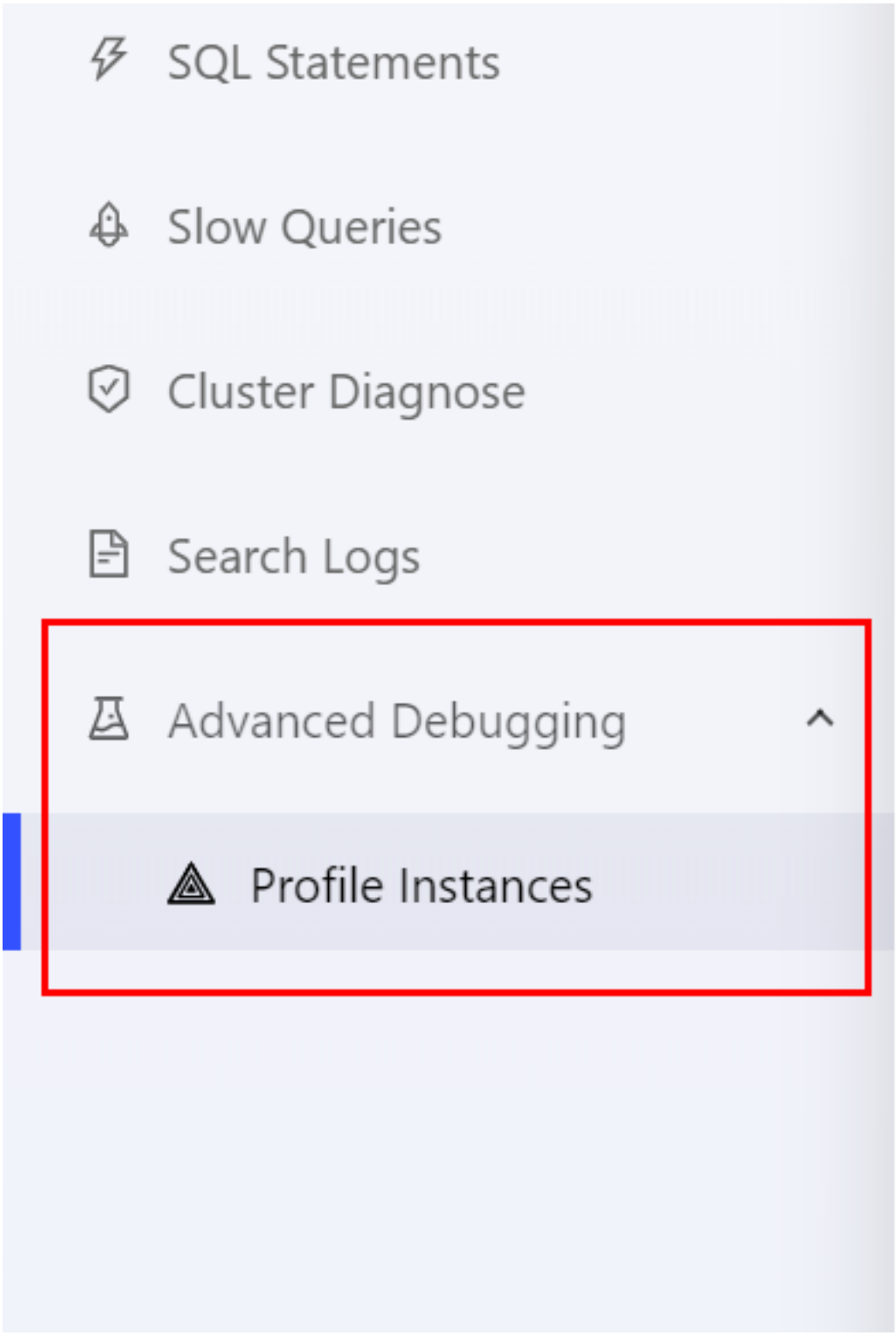


图 461: 访问

- 在浏览器中访问 http://127.0.0.1:2379/dashboard/#/instance_profiling (将 127.0.0.1:2379 替换为实际 PD 实例地址和端口)。

12.12.1.12.2 开始性能分析

在实例性能分析页面，选择至少一个想要分析的目标实例，点击开始分析（Start Profiling）按钮即可开始性能分析：

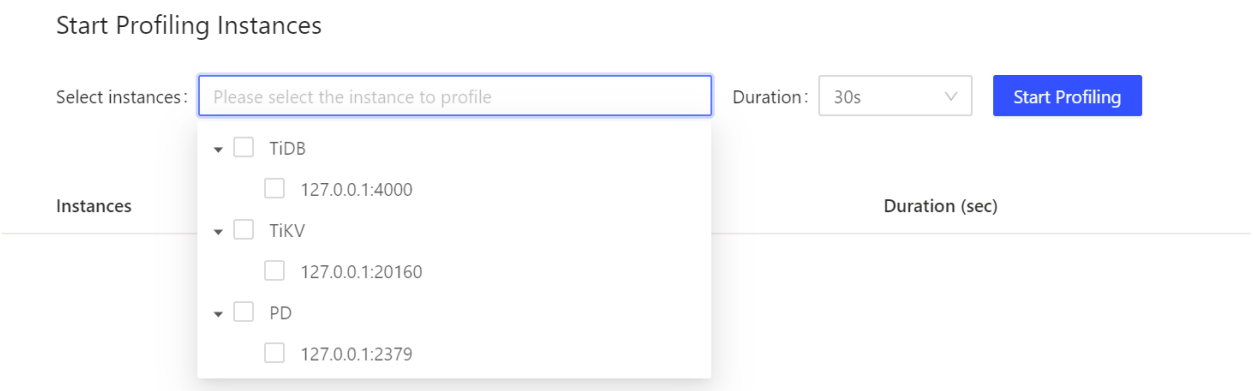


图 462: 界面

在开始性能分析前可以修改性能分析时长。性能分析所需时间取决于分析时长，默认为 30 秒。30 秒的性能分析大约需要 30 秒完成。

12.12.1.12.3 查看性能分析状态

开始性能分析后，可以看到实时性能分析状态和进度：

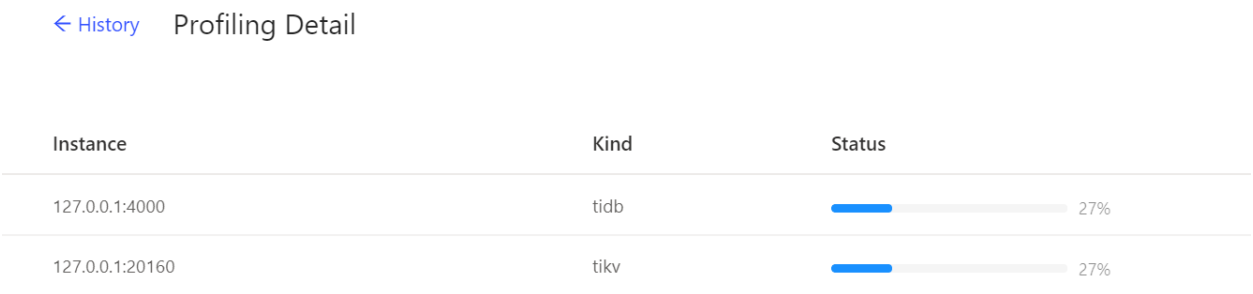


图 463: 界面

性能分析会在后台运行，刷新或退出当前页面不会终止已经运行的性能分析任务。

12.12.1.12.4 下载性能分析结果

所有实例的性能分析都完成后，可点击右上角下载按钮（Download Profiling Result）打包下载所有性能分析结果：

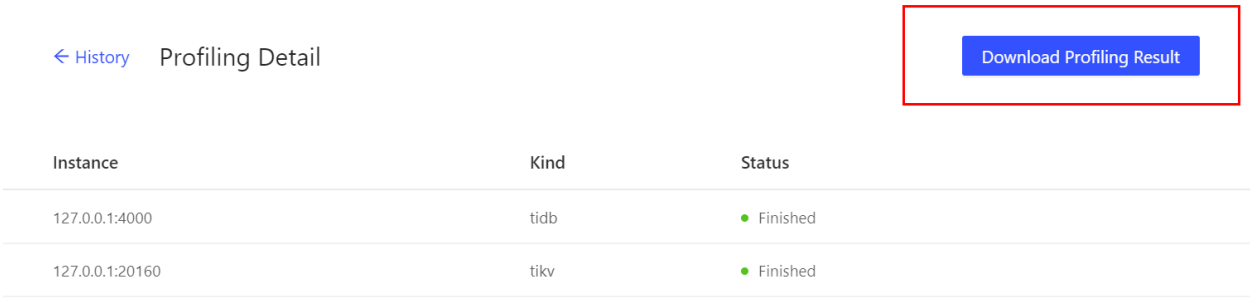


图 464: 界面

也可以点击列表中的单个实例，直接查看其性能分析结果：

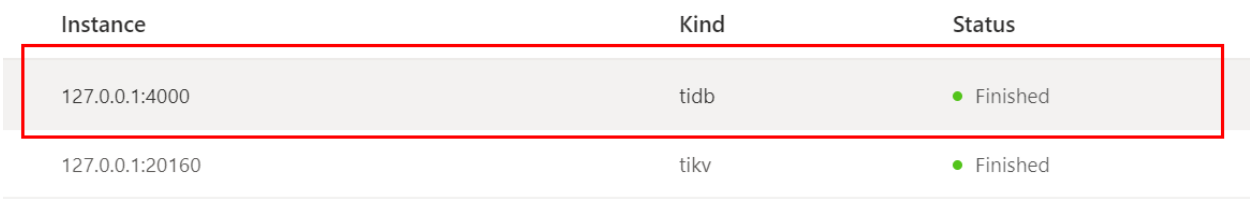


图 465: 界面

12.12.1.12.5 查看性能分析历史

在性能分析页面下方，列出了性能分析历史。点击任意一行，即可查看其状态详情：

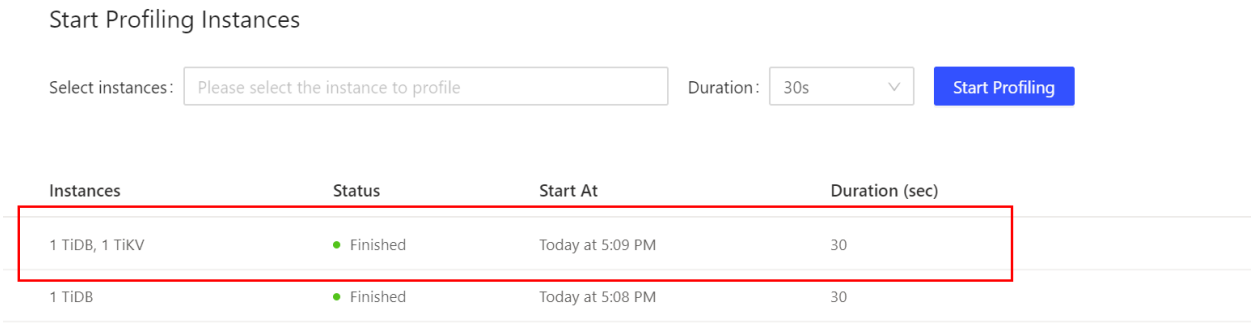


图 466: 界面

关于状态详情页的操作，参见查看性能分析状态章节。

12.12.1.13 会话管理与配置

12.12.1.13.1 分享 TiDB Dashboard 会话

> **注意：**
>
> 该功能仅在 v4.0.6 或更高版本的集群中可用。

你可以将当前的 TiDB Dashboard 会话安全地分享给其他用户访问，这样其他用户无需要知道登录账号密码即可访问 TiDB Dashboard 并进行操作。

分享者操作步骤

1. 登录 TiDB Dashboard。
2. 点击边栏左下角的用户名访问配置界面。
3. 点击分享当前会话 (Share Current Session)。

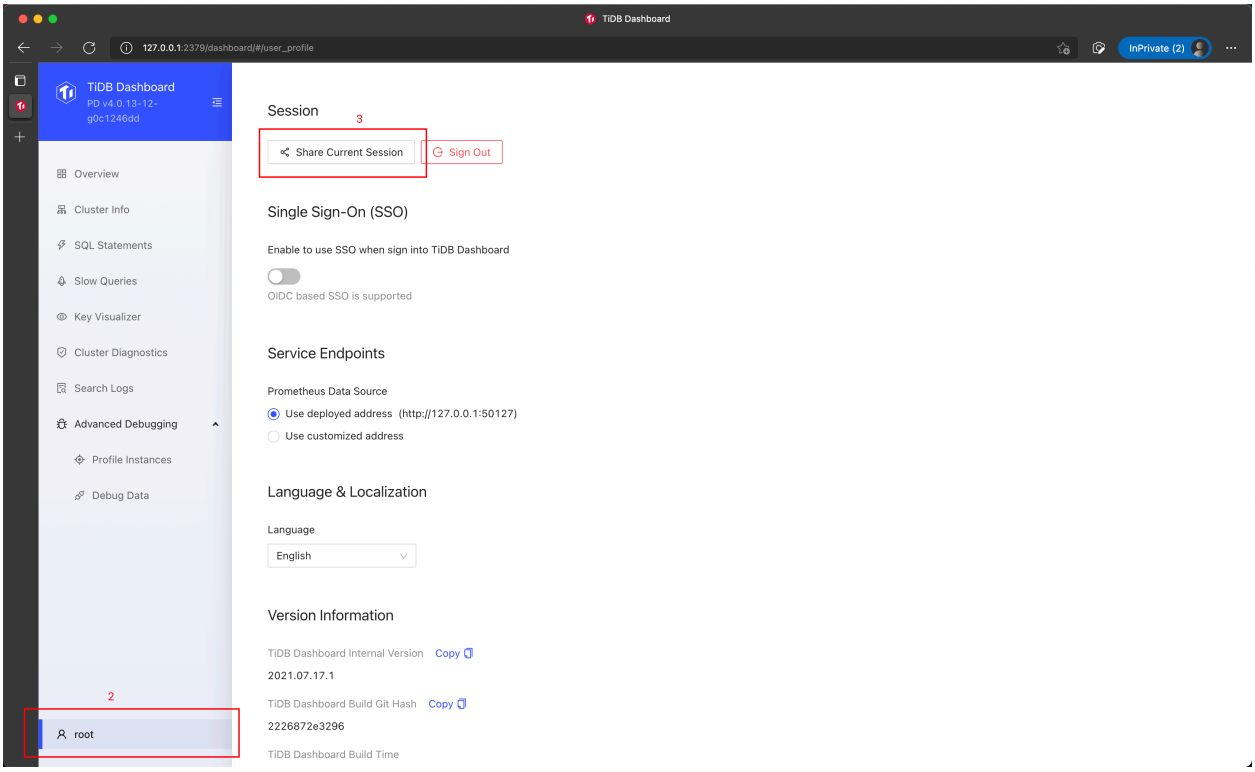


图 467: 操作示例

> **注意：**
>
> 出于安全考虑，已分享的会话中不能使用分享功能将该会话再次分享给其他人。

4. 在弹出的对话框中，对分享进行细节配置：

   - 有效时间：分享的会话在多少时间内有效。登出当前会话不影响已分享会话的有效时间。
   - 以只读权限分享：分享的会话为只读，例如不允许进行配置修改等操作。该功能仅在 v4.0.14 或更高版本中可用。

5. 点击生成授权码 (Generate Authorization Code)。

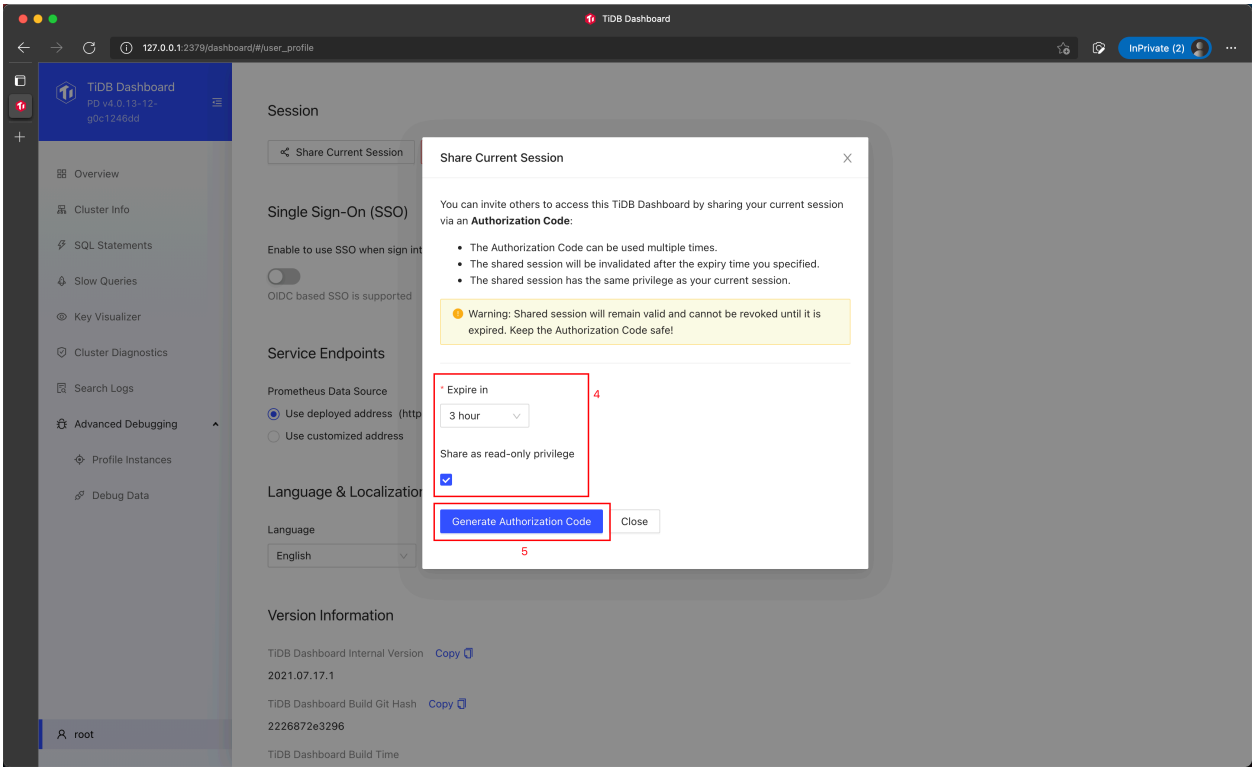


图 468: 操作示例

6. 将生成出来的授权码提供给要分享的用户。

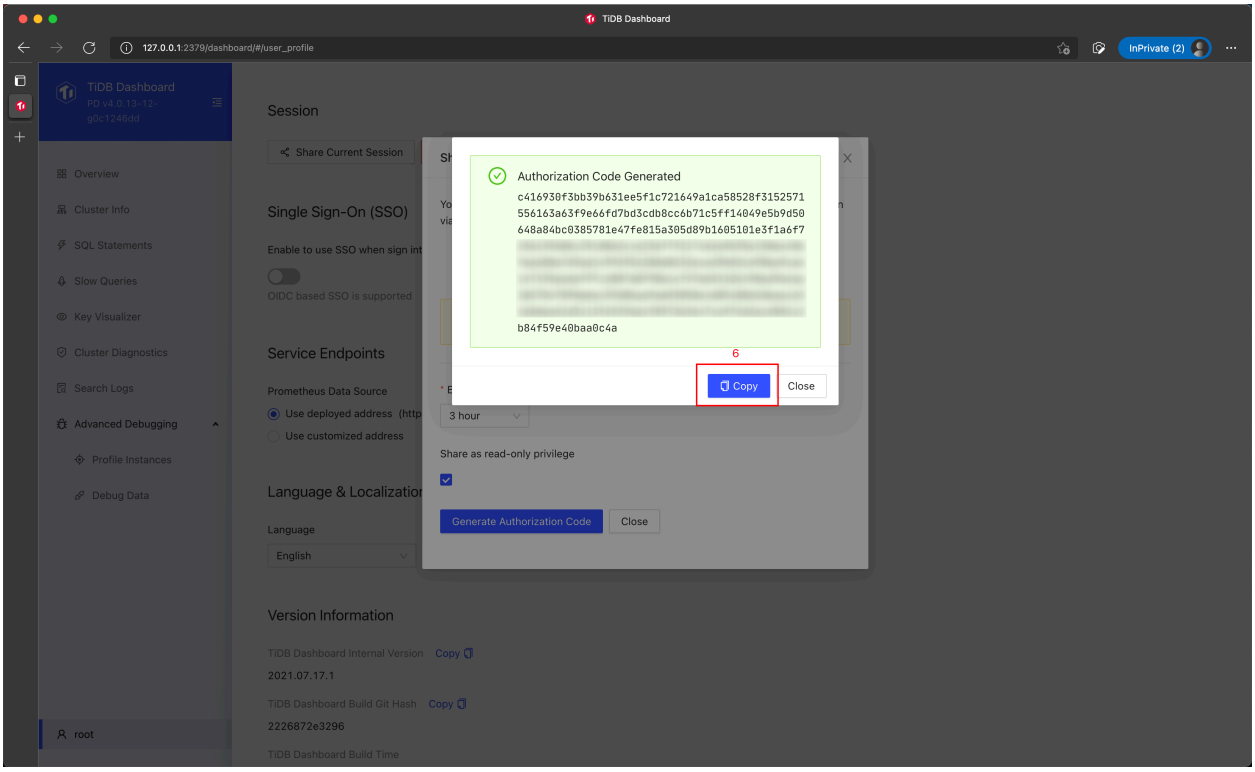


图 469: 操作示例

> **警告：**
>
> 请妥善地保管授权码。不要将授权码分发给不受信任的用户，否则他们也将具备访问和操作 TiDB Dashboard 的能力。

受邀请者操作步骤

1. 在 TiDB Dashboard 登录界面上，点击使用其他登录方式 (Use Alternative Authentication)。

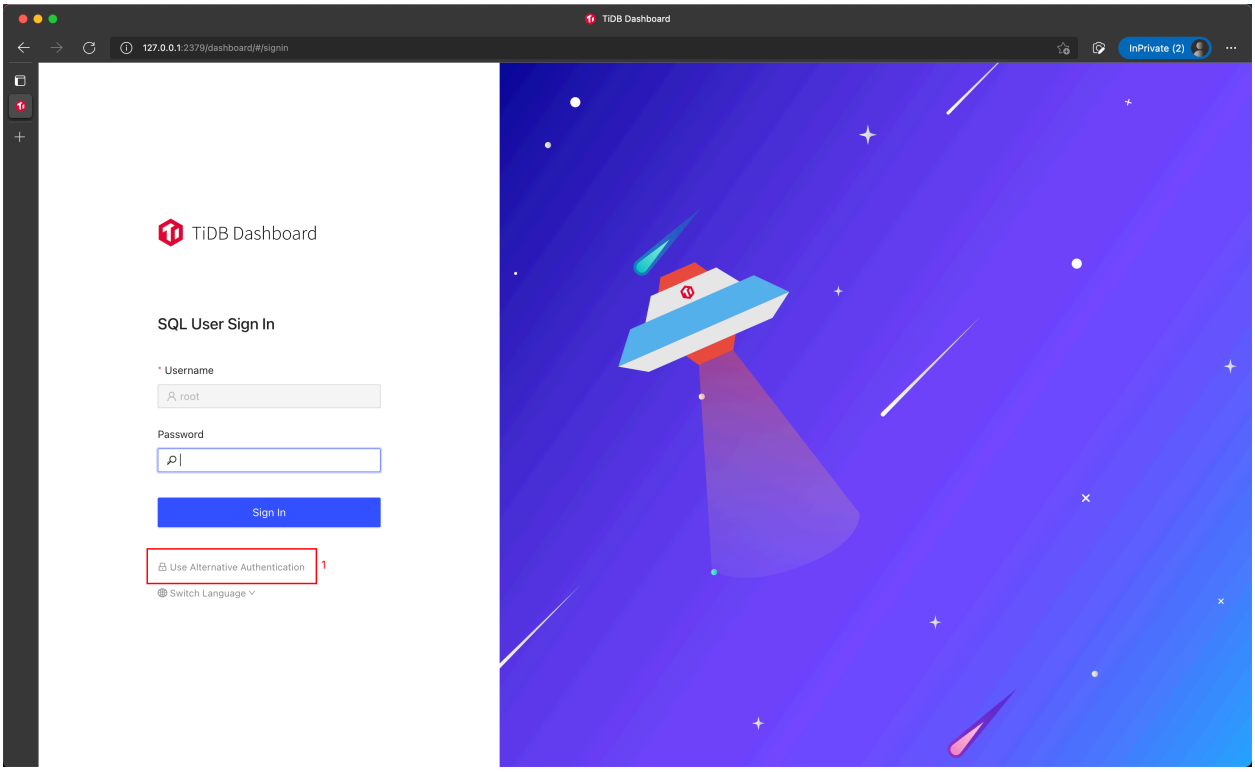


图 470: 操作示例

2. 选择使用授权码 (Authorization Code) 登录。

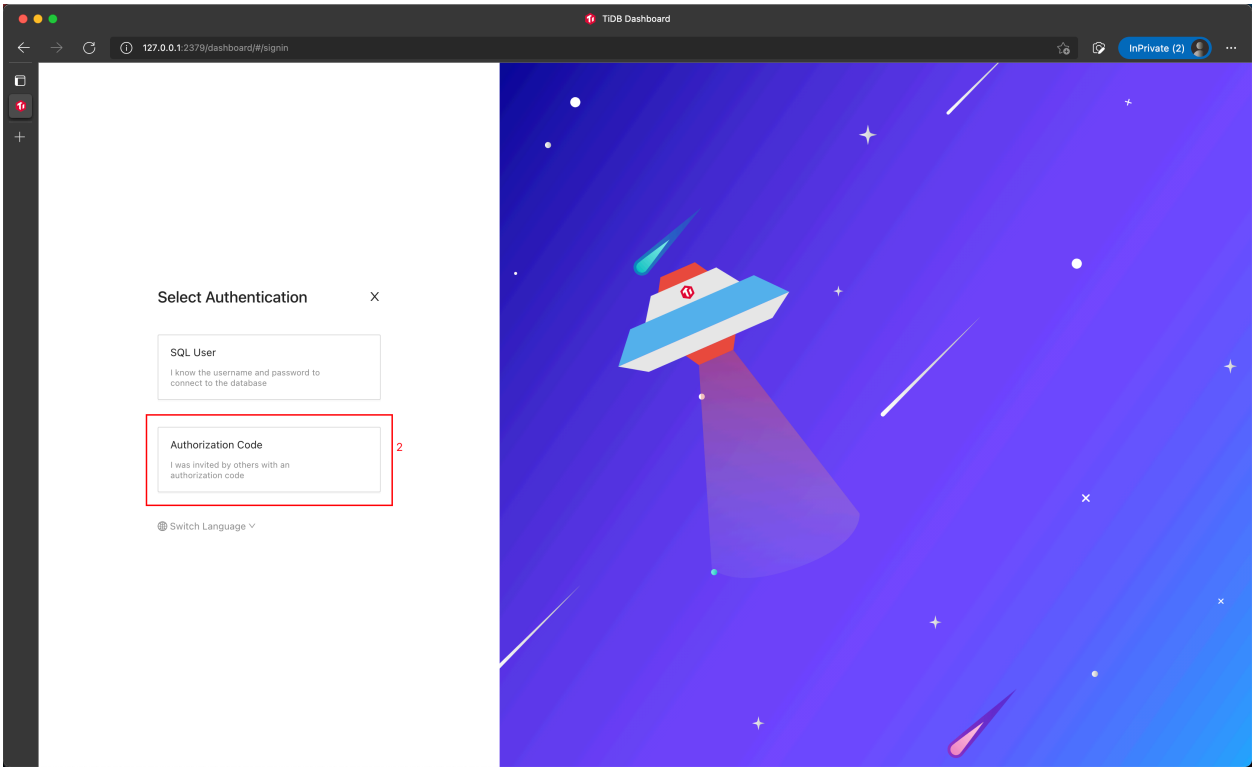


图 471: 操作示例

3. 输入从分享者取得的授权码。
4. 点击登录 (Sign In)。

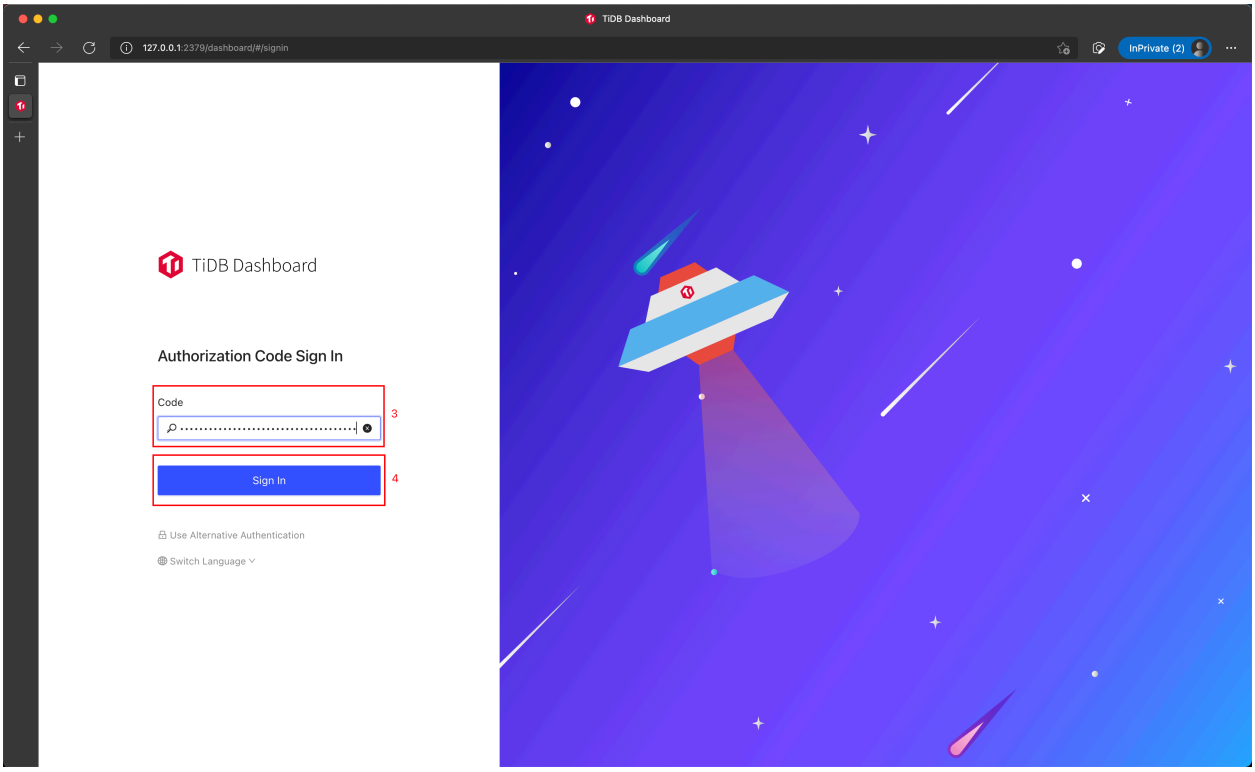


图 472: 操作示例

12.12.1.13.2 配置 TiDB Dashboard 使用 SSO 登录

TiDB Dashboard 支持基于 OIDC 协议的单点登录 (Single Sign-On)。配置 TiDB Dashboard 启用 SSO 登录后，你可以通过配置的 SSO 服务进行登录鉴权，无需输入 SQL 用户名和密码即可登录到 TiDB Dashboard。

配置 OIDC SSO

> **注意：**
>
> 该功能仅在 v4.0.14 或更高版本的集群中可用。

启用 SSO

1. 登录 TiDB Dashboard。
2. 点击边栏左下角的用户名访问配置界面。
3. 在单点登录 (Single Sign-On) 区域下，开启允许使用 SSO 登录到 TiDB Dashboard (Enable to use SSO when sign into TiDB Dashboard)。
4. 在表单中填写 OIDC Client ID 和 OIDC Discovery URL 字段。

   一般可以从 SSO 服务的提供商处获取到这两个字段信息：

- OIDC Client ID 有时也被称为 OIDC Token Issuer
- OIDC Discovery URL 有时也被称为 OIDC Token Audience。

5. 将 SQL 登录密码录入到 TiDB Dashboard 中，以便在 SSO 鉴权通过后完成登录。点击授权登录为该用户 (Authorize Impersonation) 录入密码。

   这是因为 TiDB Dashboard SSO 的原理是在 SSO 成功鉴权后，采用 TiDB Dashboard 内加密存储的 SQL 登录密码进行替代登录。

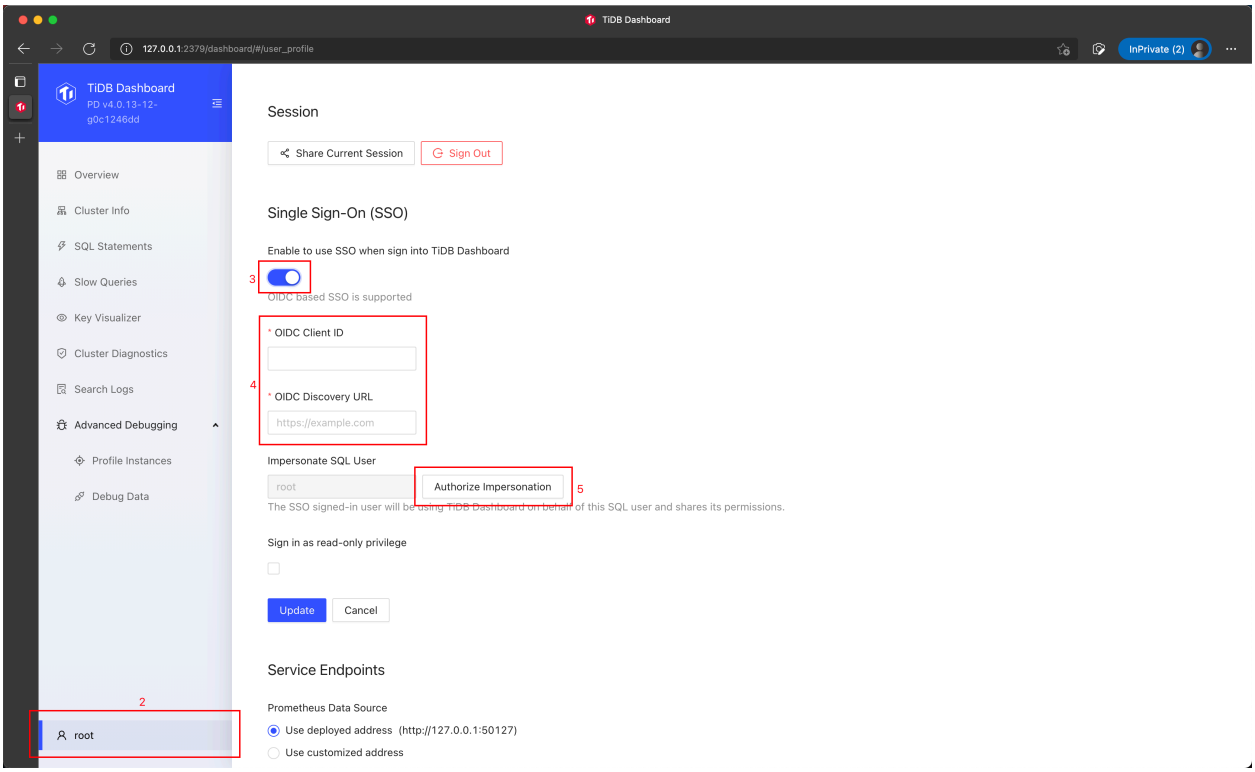


图 473: 操作示例

> **注意：**
>
> 你录入的密码将被加密存储。若 SQL 用户密码后续发生了变更，将导致 SSO 登录失败。这时可以重新录入密码使 SSO 登录恢复正常。

6. 在对话框中填写完密码后，点击授权并保存 (Authorize and Save)。

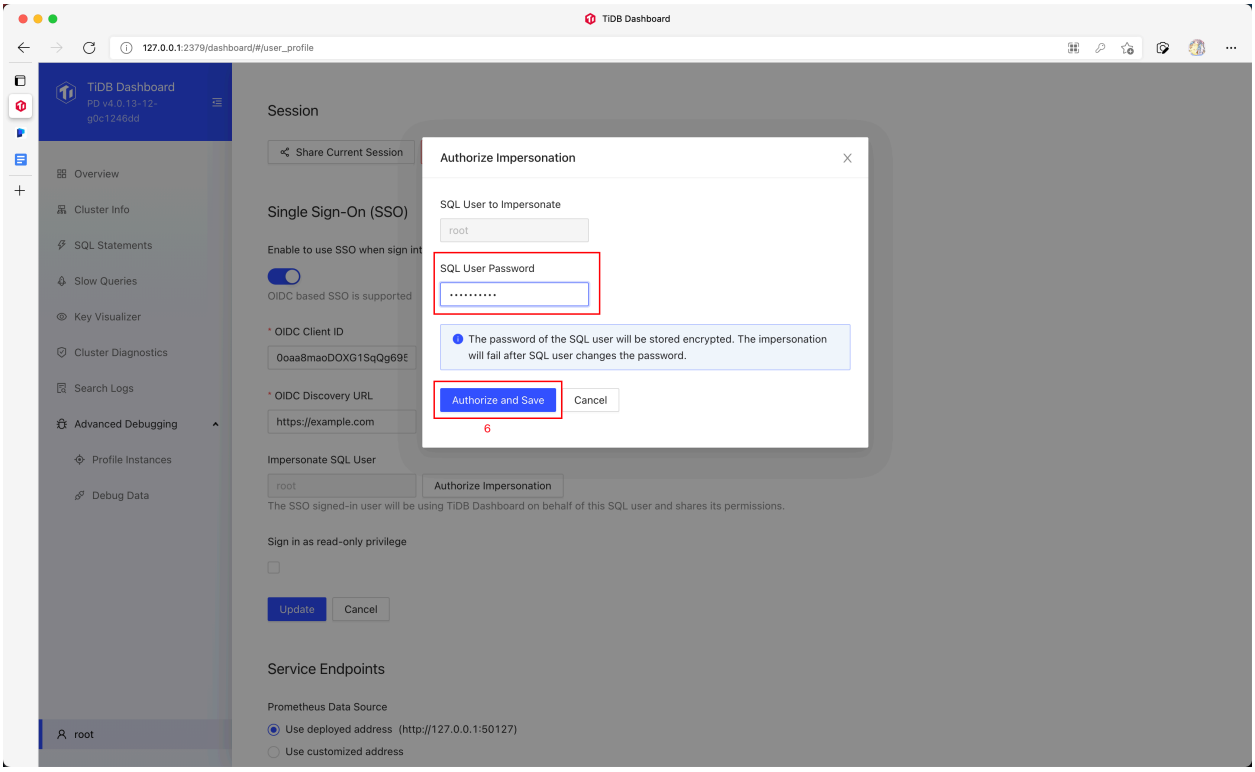


图 474: 操作示例

7. 点击更新 (Update) 保存配置。

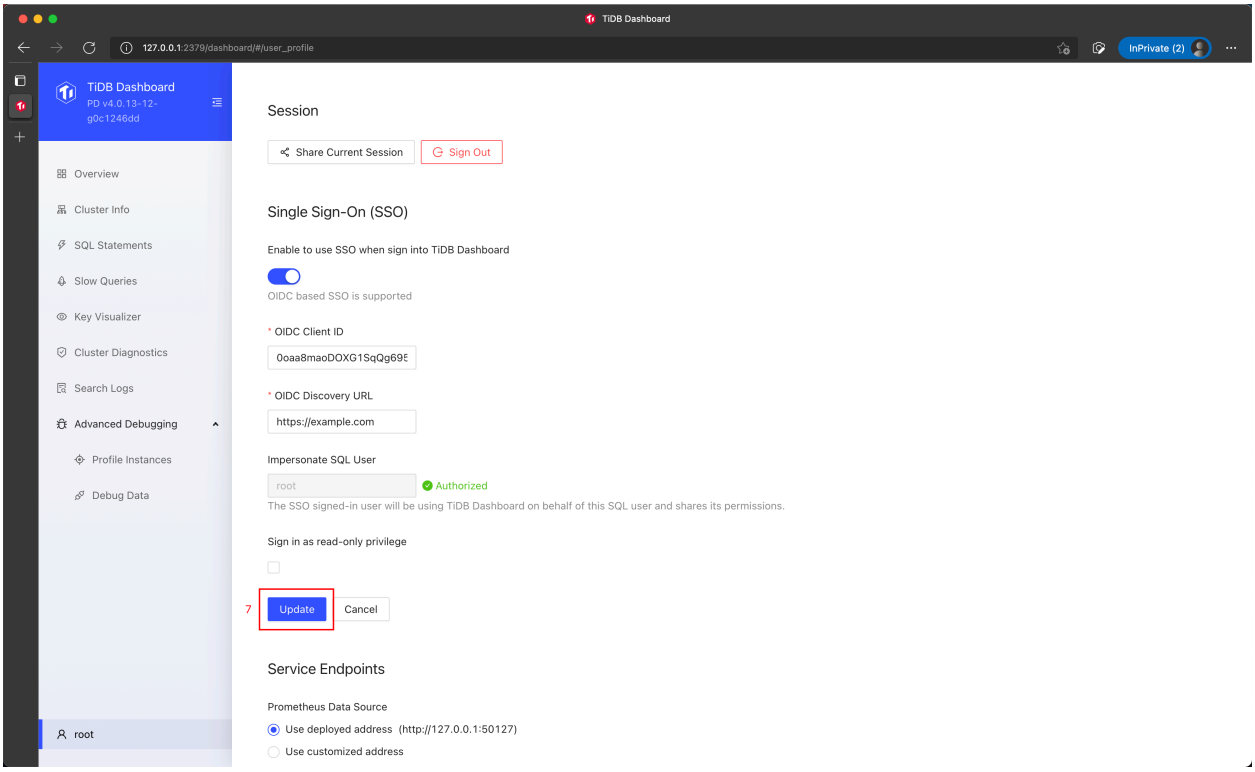


图 475: 操作示例

至此 TiDB Dashboard 中已经成功开启了 SSO 登录。

> **注意：**
>
> 出于安全原因，部分 SSO 服务还需要你进一步在 SSO 服务中配置受信任的登录和登出跳转地址，请参见 SSO 服务的具体帮助完成配置。

禁用 SSO

你可以随时禁用 SSO。禁用后，之前已录入并存储在本地的替代登录 SQL 密码将被彻底清除。禁用步骤如下：

1. 登录 TiDB Dashboard。
2. 点击边栏左下角用户名访问配置界面。
3. 在单点登录 (Single Sign-On) 区域下，关闭允许使用 SSO 登录到 TiDB Dashboard (Enable to use SSO when sign into TiDB Dashboard)。
4. 点击更新 (Update) 保存配置。

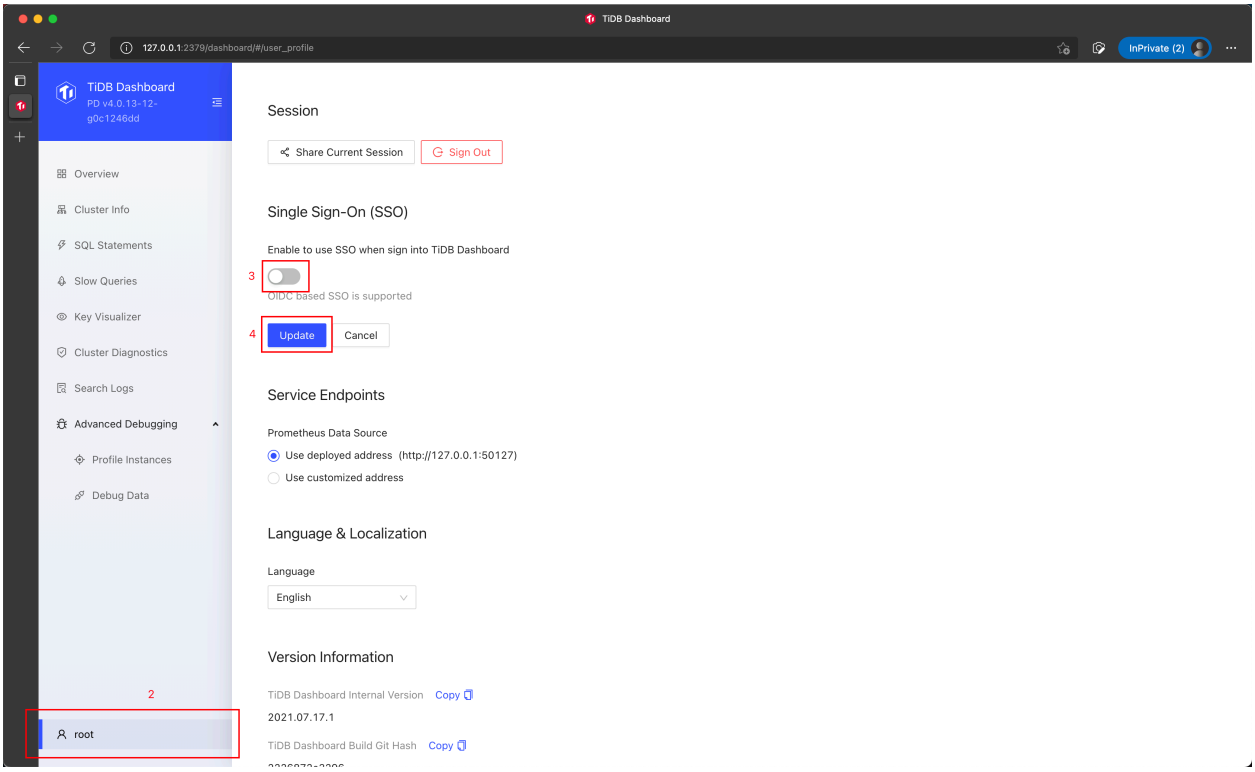


图 476: 操作示例

密码发生变更后重新录入密码

若替代登录的 SQL 用户密码发生了变更，则 SSO 登录将会失败。此时，你可以将新的登录密码录入到 TiDB Dashboard 中以恢复正常 SSO 登录功能，步骤如下：

1. 登录 TiDB Dashboard。
2. 点击边栏左下角用户名访问配置界面。
3. 在单点登录 (Single Sign-On) 区域下，点击授权登录为该用户 (Authorize Impersonation) 来录入新的密码。

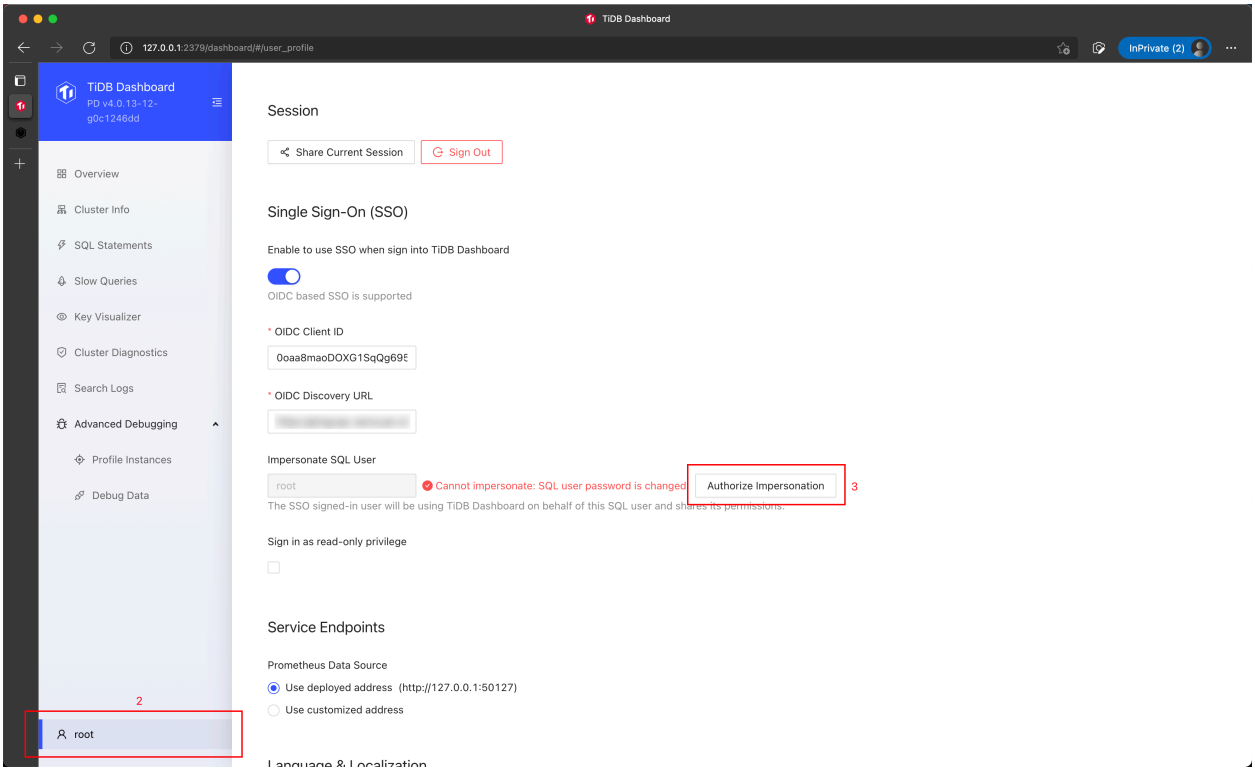


图 477: 操作示例

4. 在对话框中填写完毕密码后，点击授权并保存 (Authorize and Save)。

使用 SSO 登录

若 TiDB Dashboard 已经完成了 SSO 的配置，你可使用以下步骤完成登录：

1. 在 TiDB Dashboard 登录界面上，点击使用公司账号 SSO 登录 (Sign in via Company Account)。

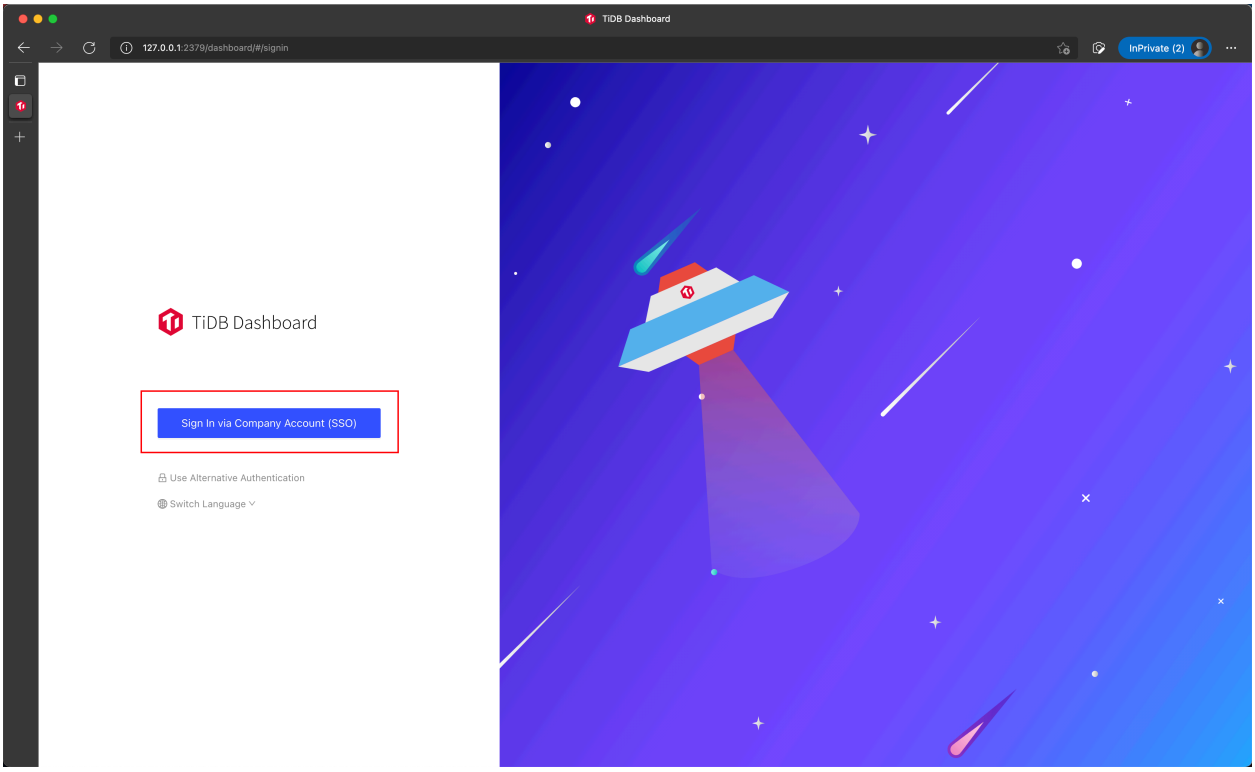


图 478: 操作示例

2. 在配置 SSO 的系统中进行登录。
3. 你将被重定向回 TiDB Dashboard 完成登录。

示例一：使用 Okta 进行 TiDB Dashboard SSO 登录认证

Okta 是一个提供 OIDC SSO 的身份认证服务。以下步骤展示了如何配置 Okta 及 TiDB Dashboard，使得 TiDB Dashboard 可以通过 Okta 进行 SSO 登录。

步骤一：配置 Okta

首先需要在 Okta 中创建一个用于集成 SSO 的 Application Integration。

1. 访问 Okta 管理后台。
2. 点击左侧边栏的 Applications > Applications。
3. 点击 Create App Integration。

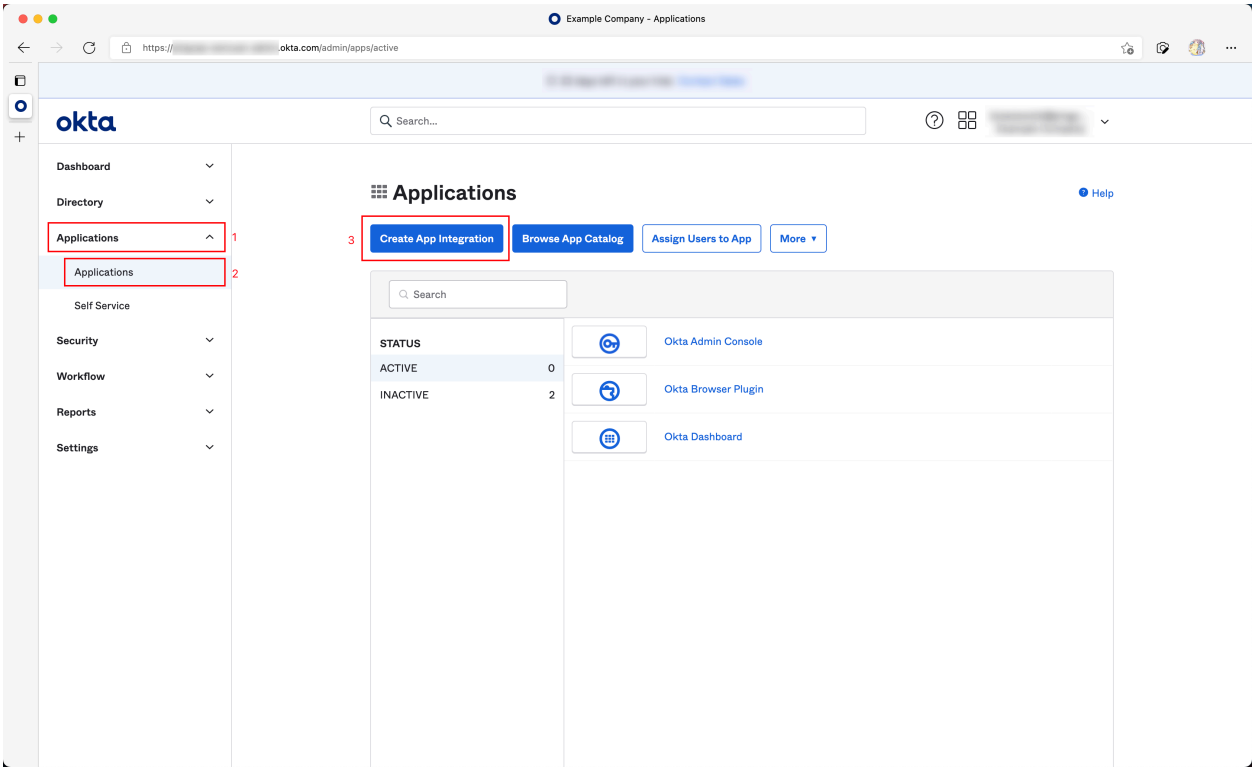


图 479: 操作示例

4. 在弹出的对话框中，Sign-in method 字段选择 OIDC - OpenID Connect。
5. Application Type 字段选择 Single-Page Application。
6. 对话框中点击 Next 按钮。

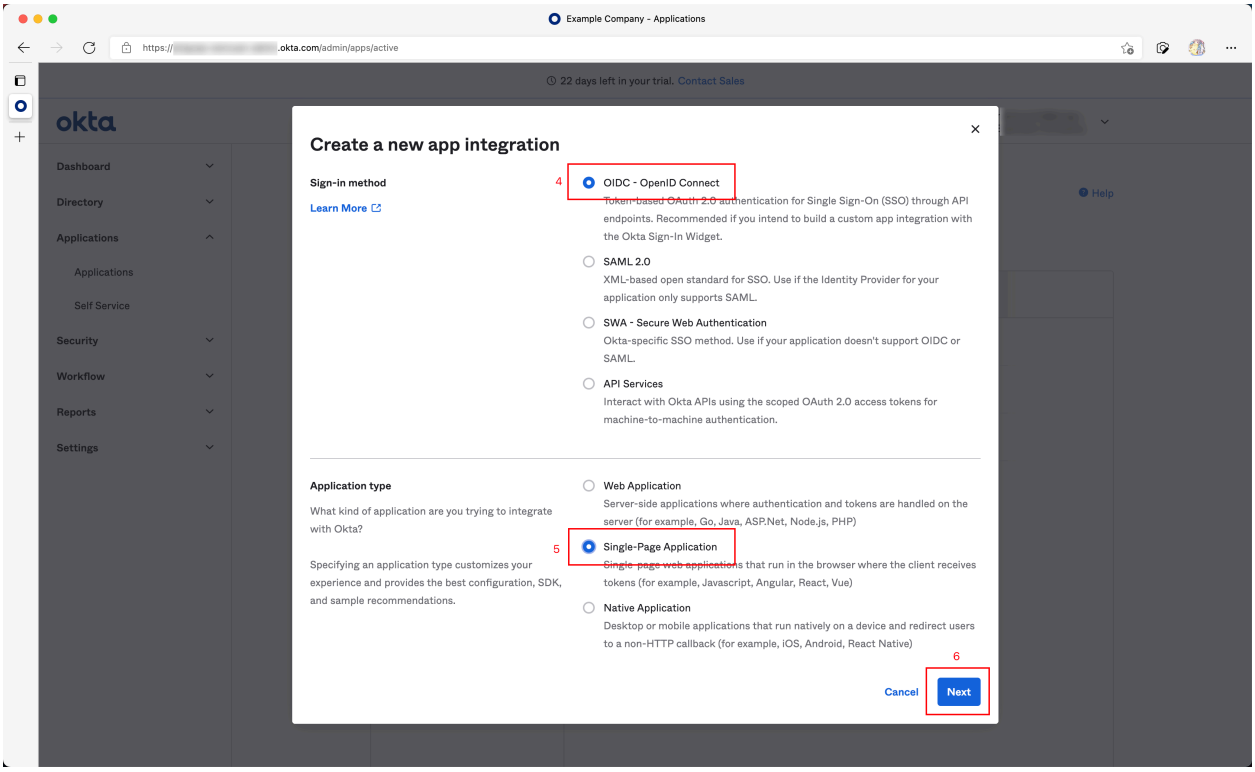


图 480: 操作示例

7. Sign-in redirect URIs 字段填写如下内容：

```
http://DASHBOARD_IP:PORT/dashboard/?sso_callback=1
```

以上内容中，将 DASHBOARD_IP:PORT 替换为你在浏览器中实际访问 TiDB Dashboard 所使用的域名（或 IP）及端口。

8. Sign-out redirect URIs 字段填写如下内容：

```
http://DASHBOARD_IP:PORT/dashboard/
```

类似地，将 DASHBOARD_IP:PORT 替换为实际的域名（或 IP）及端口。

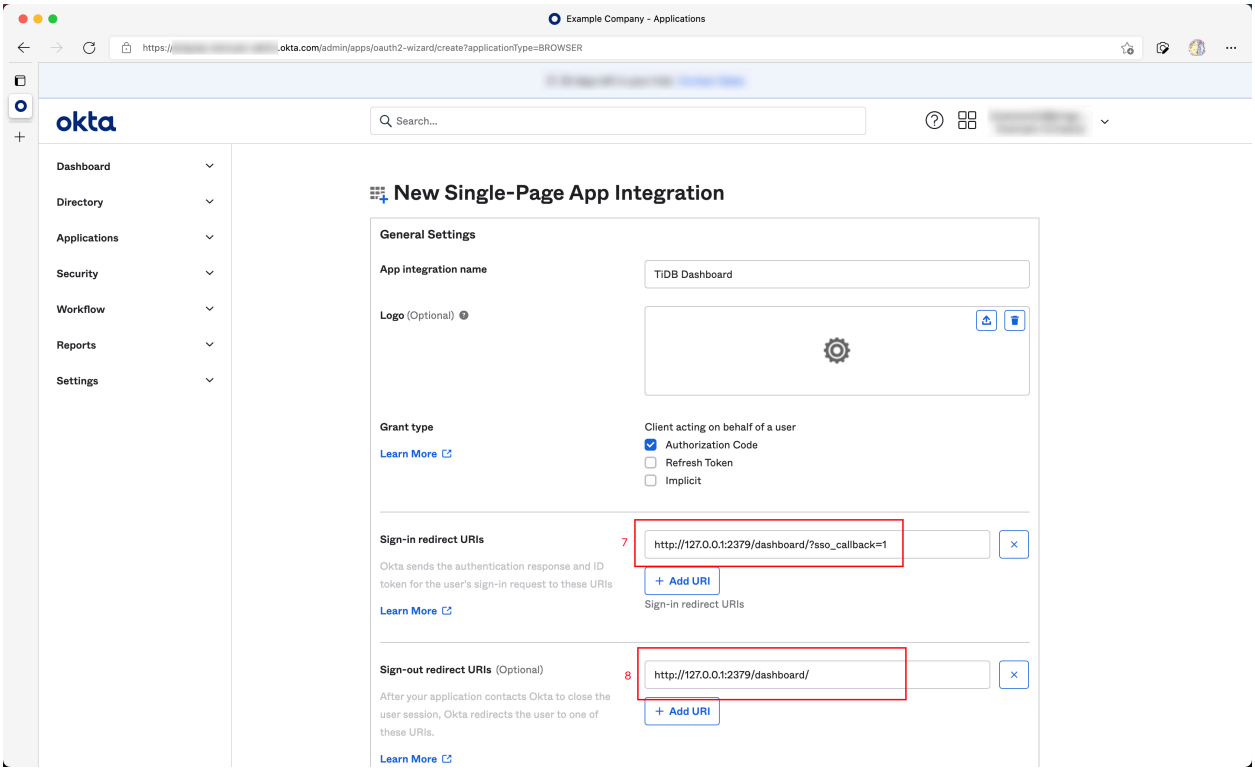


图 481: 操作示例

9. 在 Assignments 中按你的实际需求配置组织中哪些用户可以通过这个 SSO 登录 TiDB Dashboard，然后点击 Save 保存配置。

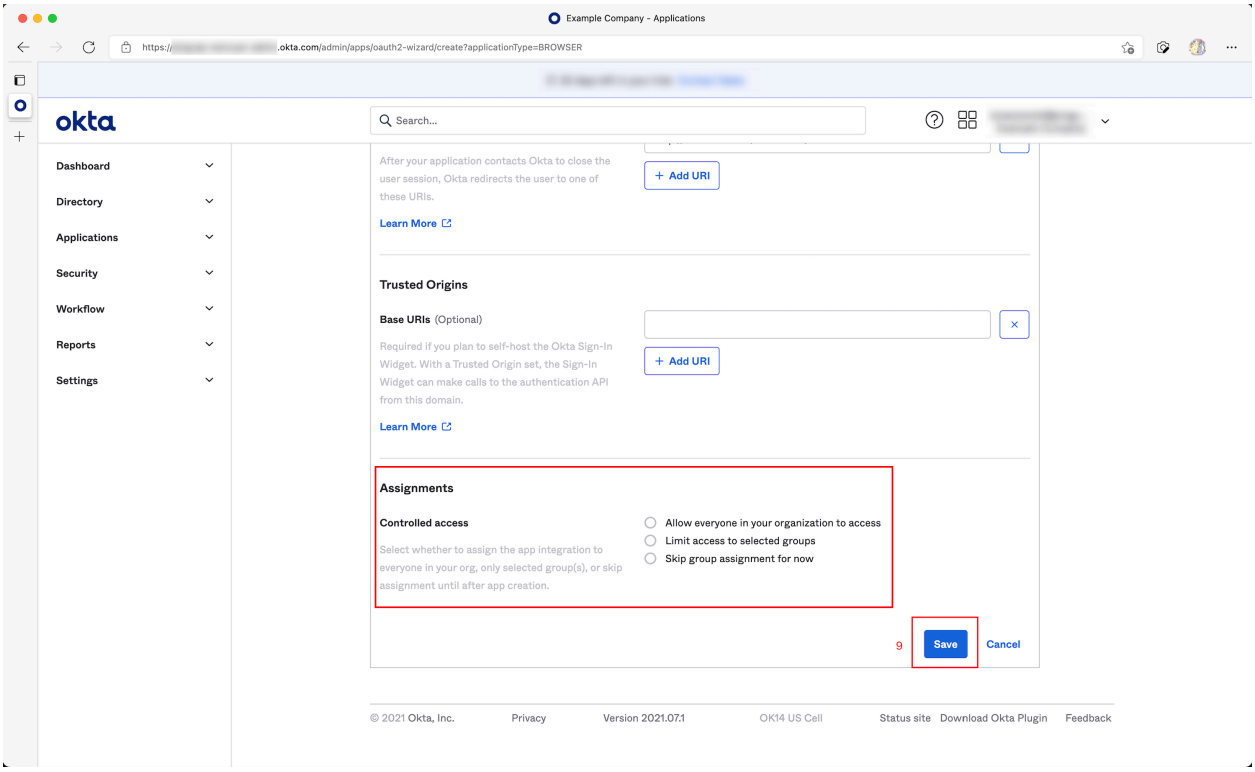


图 482: 操作示例

步骤二：获取 TiDB Dashboard 所需的配置参数并填入 TiDB Dashboard

1. 在 Okta 创建的 App Integration 中，点击 Sign On。

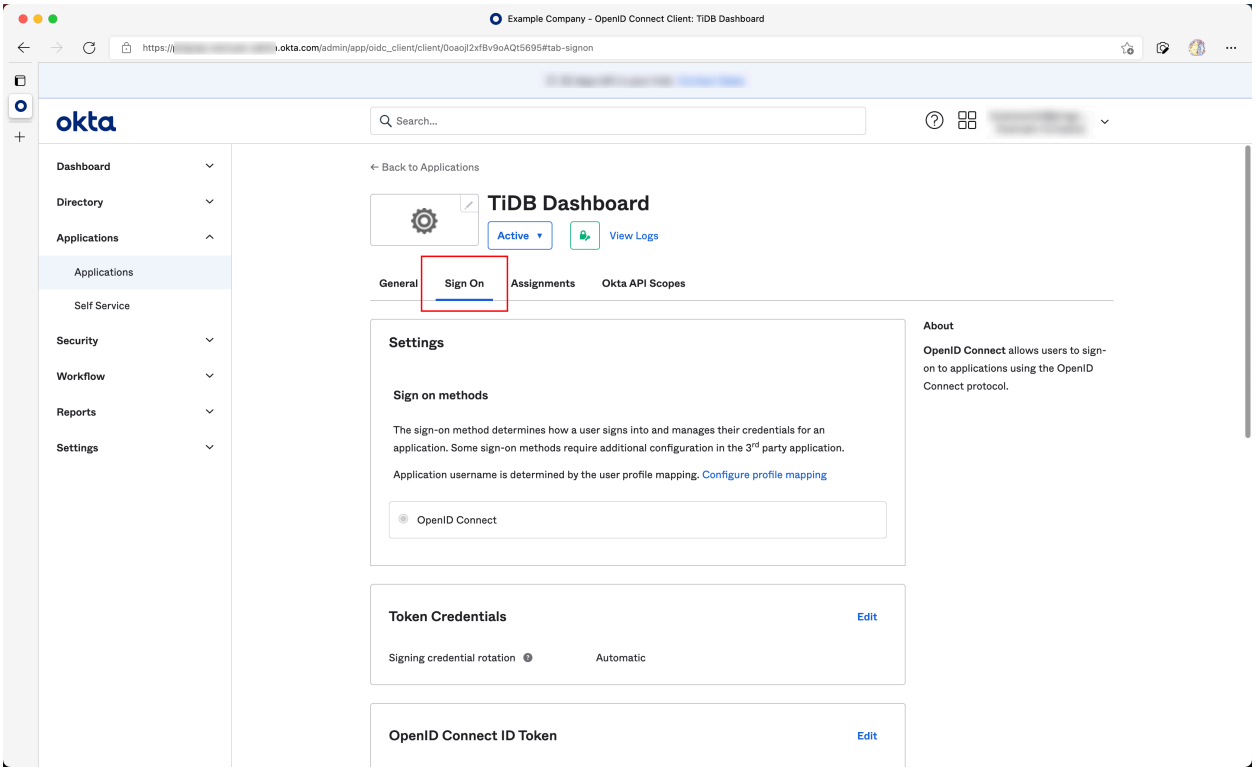


图 483: 操作示例 1

2. OpenID Connect ID Token 区域中有 Issuer 和 Audience 字段，复制这两个字段的值。

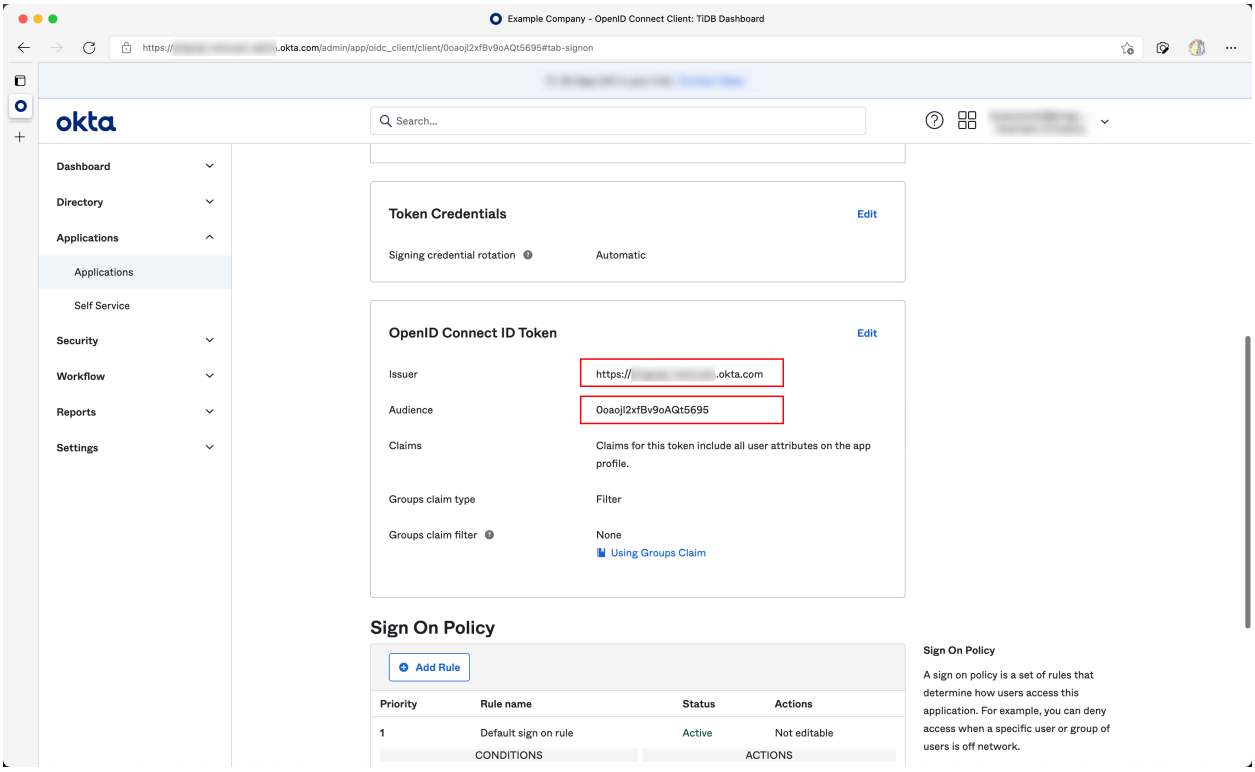


图 484: 操作示例 2

3. 打开 TiDB Dashboard 配置界面，将上一步获取到的 Issuer 填入 OIDC Client ID，将 Audience 填入 OIDC Discovery URL 后，完成授权并保存配置。示例如下：

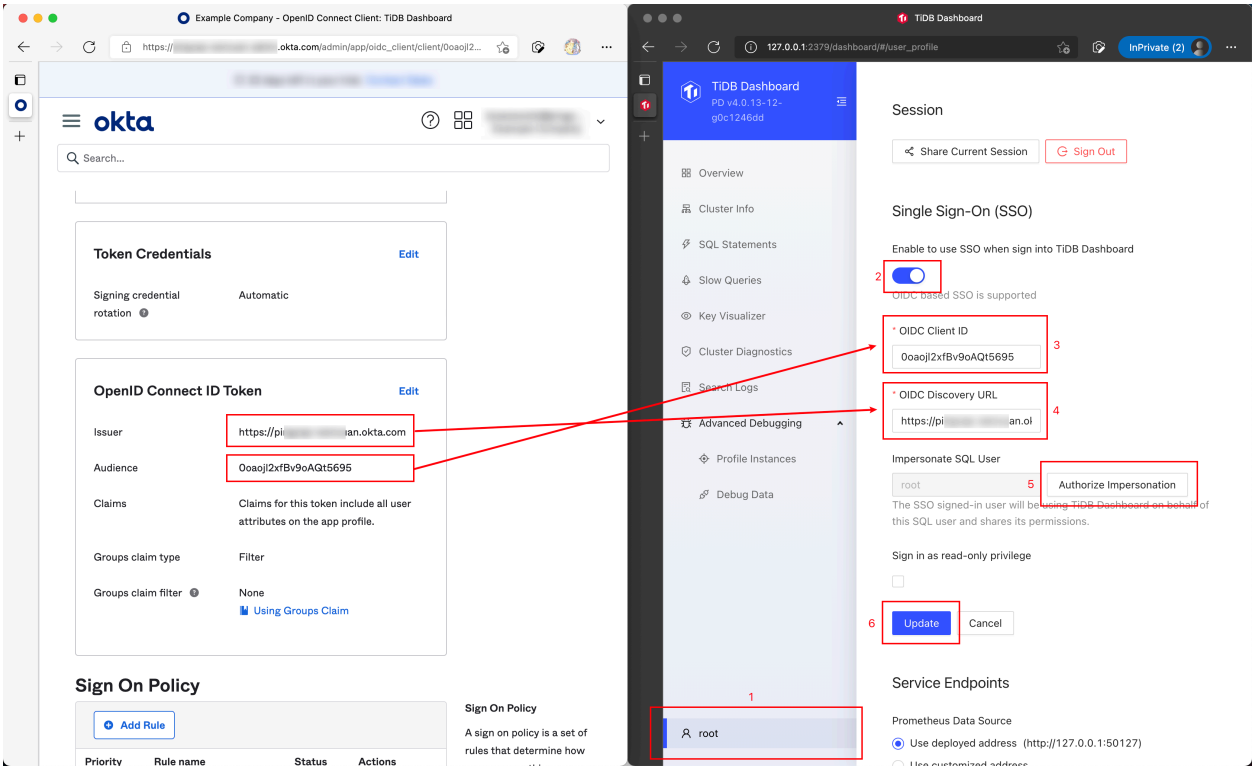


图 485: 操作示例 3

至此，TiDB Dashboard 已被配置为使用 Okta 进行 SSO 登录。

示例二：使用 Auth0 进行 TiDB Dashboard SSO 登录认证

和 Okta 类似，Auth0 也可以提供 OIDC SSO 的身份认证服务。

步骤一：配置 Auth0

1. 访问 Auth0 的管理后台。
2. 点击左侧边栏的 Applications > Applications。
3. 点击 Create Application，在弹出窗口中输入 Name，例如 “TiDB Dashboard”。在 Choose an application type 下选择 “Single Page Web Application”。

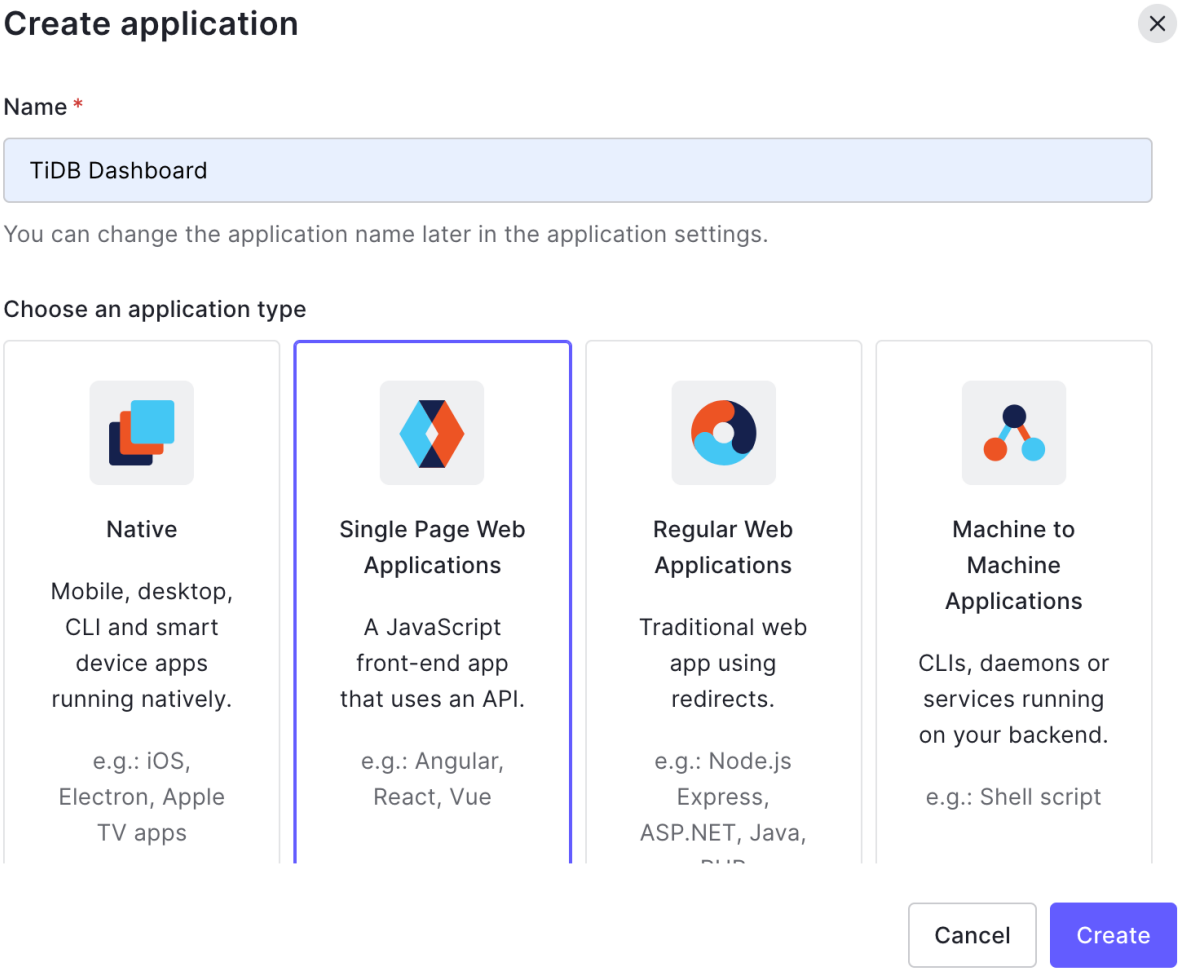


图 486: Create Application

4. 点击 Settings 栏。

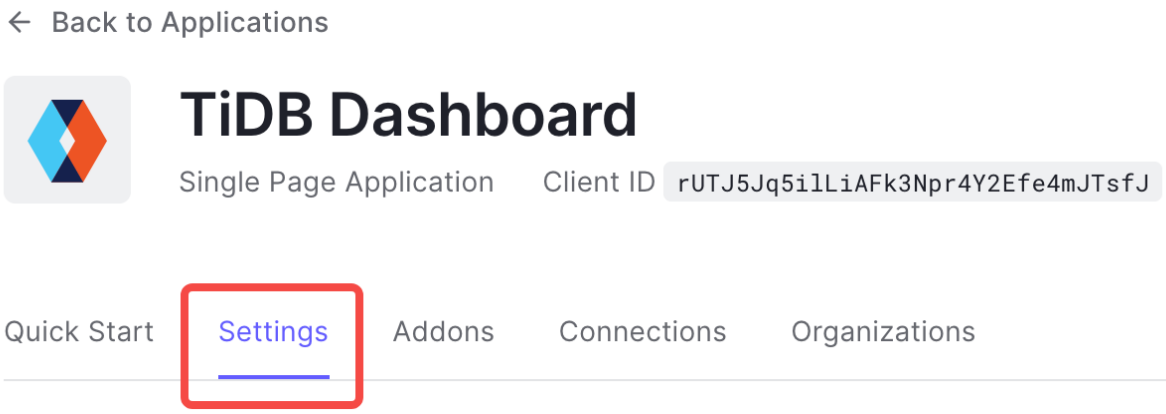


图 487: Settings

5. 在 Allowed Callback URLs 字段中填写如下内容：

```
http://DASHBOARD_IP:PORT/dashboard/?sso_callback=1
```

在以上内容中，将 DASHBOARD_IP:PORT 替换为你在浏览器中实际访问 TiDB Dashboard 所使用的域名（或 IP）及端口。

6. 在 Allowed Logout URLs 字段中填写如下内容：

```
http://DASHBOARD_IP:PORT/dashboard/
```

类似地，将 DASHBOARD_IP:PORT 替换为实际的域名（或 IP）及端口。

Allowed Callback URLs

http://localhost:3001/dashboard/?sso_callback=1,
http://127.0.0.1:2379/dashboard/?sso_callback

After the user authenticates we will only call back to any of these URLs. You can specify multiple valid URLs by comma-separating them (typically to handle different environments like QA or testing). Make sure to specify the protocol (`https://`) otherwise the callback may fail in some cases. With the exception of custom URI schemes for native clients, all callbacks should use protocol `https://`. You can use Organization URL parameters in these URLs.

Allowed Logout URLs

http://localhost:3001/dashboard/,
http://127.0.0.1:2379/dashboard/

A set of URLs that are valid to redirect to after logout from Auth0. After a user logs out from Auth0 you can redirect them with the `returnTo` query parameter. The URL that you use in `returnTo` must be listed here. You can specify multiple valid URLs by comma-separating them. You can use the star symbol as a wildcard for subdomains (`*.google.com`). Query strings and hash information are not taken into account when validating these URLs. Read more about this at https://auth0.com/docs/login/logout

图 488: Settings

7. 其它设置保持默认，点击 Save Changes 保存。

步骤二：获取 TiDB Dashboard 所需的配置参数并填入 TiDB Dashboard

1. 将 Auth0 Settings 栏 Basic Information 项的 Client ID 字段的值填入 TiDB Dashboard 的 OIDC Client ID，将 Domain 字段的值，加上 `https://` 前缀和 `/` 后缀后填入 OIDC Discovery URL 中，例如 `https://example.us.auth0.com/`。完成授权并保存配置即可。

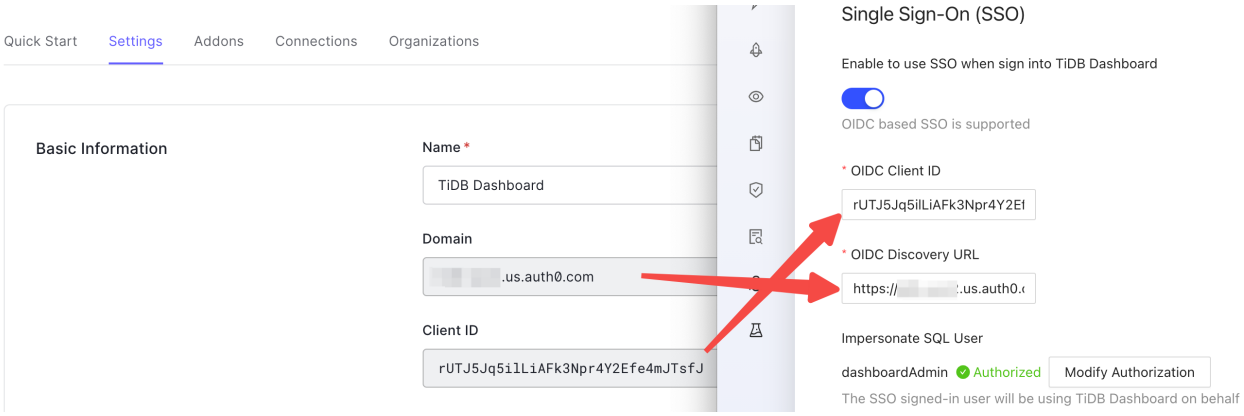


图 489: Settings

至此，TiDB Dashboard 已被配置为使用 Auth0 进行 SSO 登录。

12.12.1.14 TiDB Dashboard 常见问题

本文汇总了使用 TiDB Dashboard 过程中的常见问题与解决办法。

12.12.1.14.1 访问

已配置防火墙或反向代理，但访问后被跳转到一个内部地址无法访问 TiDB Dashboard

集群部署有多个 PD 实例的情况下，只有其中某一个 PD 实例会真正运行 TiDB Dashbaord 服务，访问其他 PD 实例时会发生浏览器端重定向。若防火墙或反向代理没有为此进行正确配置，就可能出现访问后被重定向到一个被防火墙或反向代理保护的内部地址的情况。

- 参阅TiDB Dashboard 多 PD 实例部署章节了解多 PD 实例下 TiDB Dashboard 的工作原理。
- 参阅通过反向代理使用 TiDB Dashboard 章节了解如何正确配置反向代理。
- 参阅提高 TiDB Dashboard 安全性章节了解如何正确配置防火墙。

双网卡部署时无法通过另一个网卡访问 TiDB Dashboard

PD 中的 TiDB Dashboard 出于安全考虑仅监听部署时所指定的 IP 地址（即只监听在一个网卡上），而非 `0.0.0.0`，因此当主机上安装了多个网卡时，通过另一个网卡将无法访问。

当你使用 `tiup cluster` 或 `tiup playground` 命令部署时，目前尚没有方法改变该行为。推荐使用反向代理将 TiDB Dashboard 安全地暴露给另一个网卡，具体参见通过反向代理使用 TiDB Dashboard 章节。

12.12.1.14.2 界面功能

概况页面中 QPS 及 Latency 显示 `prometheus_not_found` 错误

QPS 及 Latency 监控依赖于集群中已正常部署 Prometheus 监控实例，没有部署的情况下就会显示为错误。向集群中新部署 Prometheus 实例即可解决该问题。

若已经部署 Prometheus 监控实例但仍然显示为错误，可能的原因是您使用的部署工具（TiUP、TiDB Operator 或 TiDB Ansible）版本比较旧，没有自动汇报监控地址，导致 TiDB Dashboard 无法感知并查询监控数据。可以升级到最新的部署工具并重试。

以下给出 TiUP 部署工具的操作方法，对于其他部署工具，请参阅工具对应文档。

1. 升级 TiUP、TiUP Cluster：

```
bash  tiup update --self tiup update cluster --force
```

2. 升级后，部署包含监控节点的新集群时，应当能正常显示监控。
3. 升级后，对于现有集群，可通过再次启动集群的方法汇报监控地址（将 CLUSTER_NAME 替换为实际集群名称）：

```
bash  tiup cluster start CLUSTER_NAME
```

即使集群已经启动，请仍然执行该命令。该命令不会影响集群上正常的业务，但会刷新并上报监控地址，从而能让监控在 TiDB Dashbaord 中正常显示。

概况页面中 Top SQL 语句、最近慢查询显示 `invalid connection` 错误

可能的原因是你开启了 TiDB 的 `prepared-plan-cache` 功能。作为实验性功能，`prepared-plan-cache` 在某些版本的 TiDB 中存在一些缺陷，开启后可能会导致 TiDB Dashboard（及其他应用）出现该问题。请依据文档关闭 `prepared-plan-cache` 功能。

慢查询页面显示 unknown `field` 错误

集群升级后，如果慢查询页面出现 unknown `field` 错误，是由于升级后新版本 TiDB Dashboard 字段与浏览器缓存内的用户偏好设置的字段不兼容导致的。该问题已修复。如果你的集群版本低于 v4.0.14，可以根据以下步骤进行处理：

1. 打开 TiDB Dashboard 页面。
2. 打开浏览器的开发者工具。各浏览器的打开方式不同。

    - Firefox：菜单 □ Web 开发者 □ 切换工具箱（译者注：此处修改为最新的 Firefox Quantum），或者工具栏 □ 切换工具箱。
    - Chrome：菜单 □ 更多工具 □ 开发者工具。
    - Safari：Develop □ Show Web Inspector。如果你看不到 Develop 菜单，点开 Preferences □ Advanced，然后点击 Show Develop menu in menu bar 复选框。

    以 Chrome 为例：

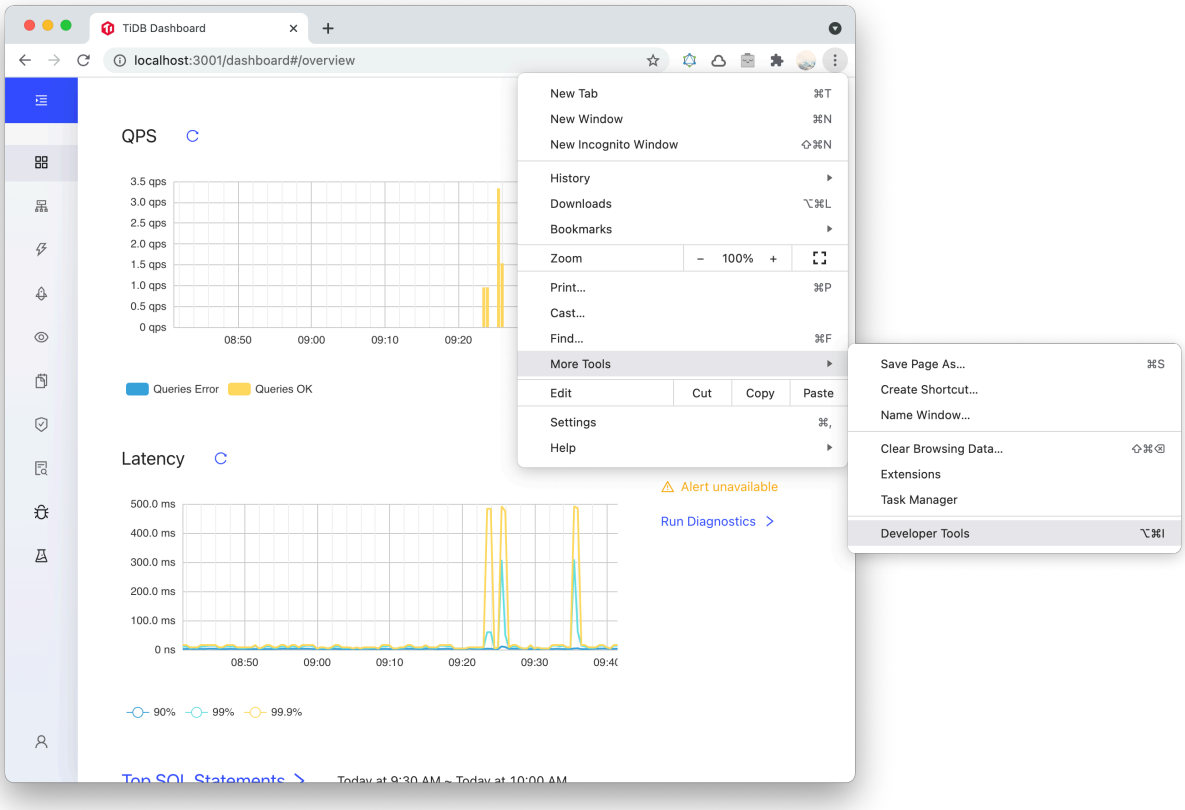


图 490: 打开开发者工具

3. 选中 Application 面板，展开 Local Storage 菜单并选中 TiDB Dashboard 页面的域名，点击 Clear All。

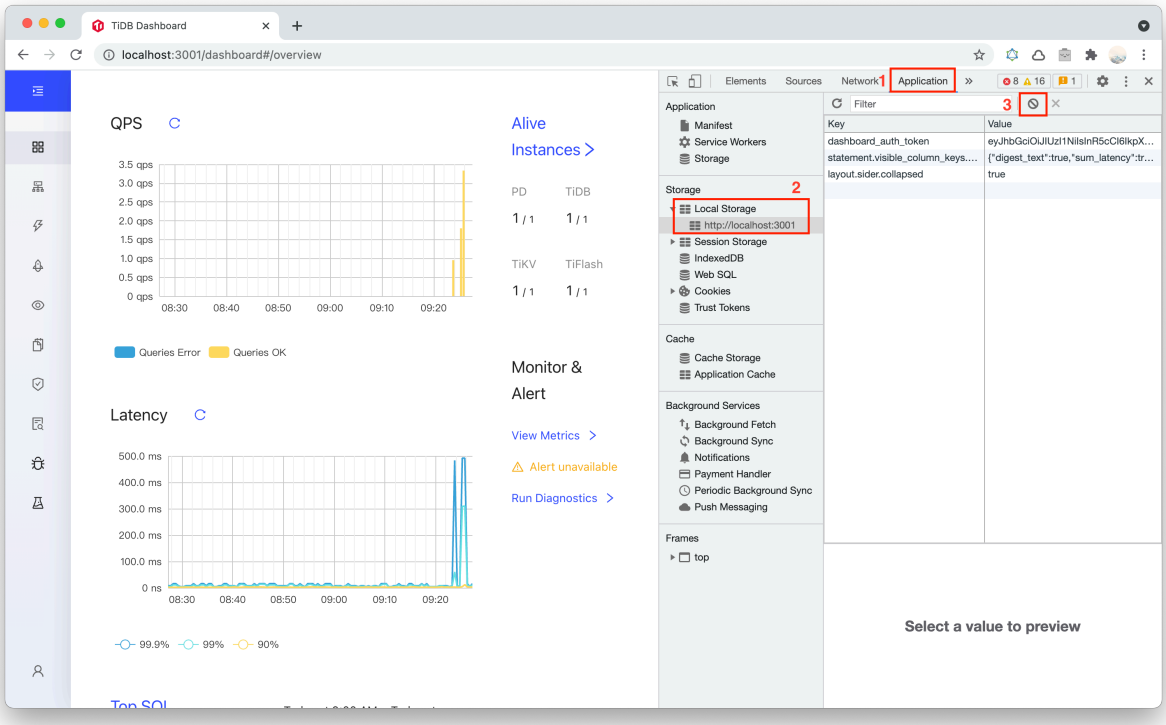


图 491: 清理 Local Storage

12.13 遥测

TiDB、TiUP 及 TiDB Dashboard 默认会收集使用情况信息，并将这些信息分享给 PingCAP 用于改善产品，例如，通过这些使用信息，PingCAP 可以了解常见的 TiDB 集群操作，从而确定新功能优先级。

12.13.1 哪些使用情况信息会被收集？

以下章节具体描述了各个组件收集并分享的使用情况信息。若收集的使用情况信息有变化，将在版本更新说明中告知。

> **注意：**
>
> 在任何情况下，集群中用户存储的数据都不会被收集。另请参阅 PingCAP 隐私声明。

12.13.1.1 TiDB

当 TiDB 遥测功能开启时，TiDB 集群将会以 24 小时为周期收集使用情况信息并分享给 PingCAP，包括（但不限于）：

- 随机生成的遥测标示符
- 集群的部署情况，包括各个组件所在的硬件信息（CPU、内存、磁盘）、组件版本号、操作系统版本号等

可以通过执行以下 SQL 语句查看 TiDB 收集的使用情况信息内容：

```
ADMIN SHOW TELEMETRY;
```

12.13.1.2 TiDB Dashboard

当 TiDB Dashboard 遥测功能开启时，用户在 TiDB Dashboard 网页界面上进行操作时会将使用情况信息分享给 PingCAP，包括（但不限于）：

- 随机生成的遥测标示符
- 界面访问情况，如访问的 TiDB Dashboard 功能页面名称
- 用户浏览器及操作系统信息，如浏览器名称和版本号、操作系统名称、屏幕分辨率等

可以使用 Chrome 开发者工具的网络功能或 Firefox 开发者工具的网络监视器功能查看 TiDB Dashboard 发送的使用情况信息内容。

12.13.1.3 TiUP

当 TiUP 遥测功能开启时，执行 TiUP 命令时会将使用情况信息分享给 PingCAP，包括（但不限于）：

- 随机生成的遥测标示符
- TiUP 命令的执行情况，如命令执行是否成功、命令执行耗时等
- 使用 TiUP 进行部署的情况，如部署的目标机器硬件信息、组件版本号、修改过的部署配置名称等

使用 TiUP 时，可通过设置 `TIUP_CLUSTER_DEBUG=enable` 环境变量输出执行命令时收集的使用情况信息，例如：

```
TIUP_CLUSTER_DEBUG=enable tiup cluster list
```

12.13.2 禁用遥测功能

12.13.2.1 部署 TiDB 时禁用 TiDB 遥测

部署 TiDB 集群时，可以为每个 TiDB 集群设置`enable-telemetry = false` 以禁用 TiDB 遥测功能。也可以在已部署的 TiDB 集群上修改该配置项，但需要重启集群后才能生效。

以下是在各个部署工具中修改遥测配置的具体步骤。

通过二进制手工部署

创建配置文件 `tidb_config.toml` 包含如下内容：

```
enable-telemetry = false
```

启动 TiDB 时指定命令行参数 `--config=tidb_config.toml` 使得该配置生效。

详情参见TiDB 配置参数、TiDB 配置文件描述。

通过 TiUP Playground 试用

创建配置文件 `tidb_config.toml` 包含如下内容：

```
enable-telemetry = false
```

启动 TiUP Playground 时，指定命令行参数 `--db.config tidb_config.toml` 使得该配置生效，如：

```
tiup playground --db.config tidb_config.toml
```

详情参见TiUP - 本地快速部署 TiDB 集群。

通过 TiUP Cluster 部署

修改部署拓扑文件 `topology.yaml`，新增（或在现有项中添加）以下内容：

```
server_configs:
  tidb:
    enable-telemetry: false
```

通过 TiDB Ansible 部署

找到部署配置文件 `tidb-ansible/conf/tidb.yml` 中以下内容：

```
## enable-telemetry: true
```

将其修改为：

```
enable-telemetry: false
```

详情参见使用 TiDB Ansible 部署。

通过 TiDB Operator 在 Kubernetes 上部署

在 `tidb-cluster.yaml` 中或者 TidbCluster Custom Resource 中配置 `spec.tidb.config.enable-telemetry: false`。

详情参见在标准 Kubernetes 上部署 TiDB 集群。

> 注意：
>
> 该配置需使用 TiDB Operator v1.1.3 或更高版本才能生效。

12.13.2.2 动态禁用 TiDB 遥测

对于已部署的 TiDB 集群，还可以修改系统全局变量`tidb_enable_telemetry` 动态禁用 TiDB 遥测功能：

```
SET GLOBAL tidb_enable_telemetry = 0;
```

配置文件的禁用优先级高于全局变量。若通过配置文件禁用了遥测功能，则全局变量的配置将不起作用，遥测功能总是处于关闭状态。

12.13.2.3 禁用 TiDB Dashboard 遥测

可以修改 PD 配置中`dashboard.enable-telemetry = false` 禁用 TiDB Dashboard 遥测功能。对于已启动的集群，该配置需要重启后才能生效。

以下列出在各个部署工具中修改遥测配置的具体步骤。

通过二进制手工部署

创建配置文件 `pd_config.toml` 包含如下内容：

```
[dashboard]
enable-telemetry = false
```

启动 PD 时指定命令行参数 `--config=pd_config.toml` 使得该配置生效。

详情参见PD 配置参数、PD 配置文件描述。

通过 TiUP Playground 试用

创建配置文件 `pd_config.toml` 包含如下内容：

```
[dashboard]
enable-telemetry = false
```

启动 TiUP Playground 时，指定命令行参数 `--pd.config pd_config.toml` 使得该配置生效，如：

```
tiup playground --pd.config pd_config.toml
```

详情参见TiUP - 本地快速部署 TiDB 集群。

通过 TiUP Cluster 部署

修改部署拓扑文件 `topology.yaml`，新增（或在现有项中添加）以下内容：

```
server_configs:
  pd:
    dashboard.enable-telemetry: false
```

通过 TiDB Ansible 部署

找到部署配置文件 `tidb-ansible/conf/pd.yml` 中以下内容：

```
dashboard:
  ...
  # enable-telemetry: true
```

将其修改为：

```
dashboard:
  ...
  enable-telemetry: false
```

详情参见使用 TiDB Ansible 部署。

通过 TiDB Operator 在 Kubernetes 上部署

在 `tidb-cluster.yaml` 中或者 TidbCluster Custom Resource 中配置 `spec.pd.config.dashboard.enable-telemetry: false`。

详情参见在标准 Kubernetes 上部署 TiDB 集群。

> **注意：**
>
> 该配置需使用 TiDB Operator v1.1.3 或更高版本才能生效。

12.13.2.4 禁用 TiUP 遥测

可通过执行以下命令禁用 TiUP 遥测功能：

```
tiup telemetry disable
```

12.13.3 查看遥测启用状态

对于 TiDB 遥测，可通过执行以下 SQL 语句查看遥测状态：

```
ADMIN SHOW TELEMETRY;
```

若 DATA_PREVIEW 列为空，说明遥测没有开启，否则说明遥测已开启。还可以从 LAST_STATUS 列了解上次分享使用情况信息的时间、是否成功等。

对于 TiUP 遥测，可通过执行以下命令查看遥测状态：

```
tiup telemetry status
```

12.13.4 使用情况信息合规性

为了满足不同国家或地区对于此类信息的合规性要求，使用情况信息会按照不同的操作者 IP 地址发送到位于不同国家的服务器，具体如下：

- 若为中国大陆 IP 地址，使用情况信息将会发送并存储于中国大陆境内的公有云服务器。
- 若为中国大陆以外 IP 地址，使用情况信息将会发送并存储于美国的公有云服务器。

可参阅 PingCAP 隐私声明了解详情。

12.14 错误码与故障诊断

本篇文档描述在使用 TiDB 过程中会遇到的问题以及解决方法。

12.14.1 错误码

TiDB 兼容 MySQL 的错误码，在大多数情况下，返回和 MySQL 一样的错误码。关于 MySQL 的错误码列表，详见 [MySQL 5.7 Error Message Reference。另外还有一些 TiDB 特有的错误码：

> 注意：
>
> 有一部分错误码属于内部错误，正常情况下 TiDB 会自行处理不会直接返回给用户，故没有在此列出。
>
> 如果您遇到了这里没有列出的错误码，请向 PingCAP 工程师或通过官方论坛寻求帮助。

- Error Number: 8001

  请求使用的内存超过 TiDB 内存使用的阈值限制。出现这种错误，可以通过调整 `mem-quota-query` 来增大单个 SQL 使用的内存上限。

- Error Number: 8002

  带有 `SELECT FOR UPDATE` 语句的事务，在遇到写入冲突时，为保证一致性无法进行重试，事务将进行回滚并返回该错误。出现这种错误，应用程序可以安全地重新执行整个事务。

- Error Number: 8003

  `ADMIN CHECK TABLE` 命令在遇到行数据跟索引不一致的时候返回该错误，在检查表中数据是否有损坏时常出现。出现该错误时，请向 PingCAP 工程师或通过官方论坛寻求帮助。

- Error Number: 8004

  单个事务过大，原因及解决方法请参考这里

- Error Number: 8005

  事务在 TiDB 中遇到了写入冲突，原因及解决方法请参考这里

- Error Number: 8018

  当执行重新载入插件时，如果之前插件没有载入过，则会出现该错误。出现该错误，进行插件首次载入即可。

- Error Number: 8019

  重新载入的插件版本与之前的插件版本不一致，无法重新载入插件并报告该错误。可重新载入插件，确保插件的版本与之前载入的插件版本一致。

- Error Number: 8020

  当表被加锁时，如果对该表执行写入操作，将出现该错误。请将表解锁后，再进行尝试写入。

- Error Number: 8021

  当向 TiKV 读取的 key 不存在时将出现该错误，该错误用于内部使用，对外表现为读到的结果为空。

- Error Number: 8022

  事务提交失败，已经回滚，应用程序可以安全的重新执行整个事务。

- Error Number: 8023

  在事务内，写入事务缓存时，设置了空值，将返回该错误。这是一个内部使用的错误，将由内部进行处理，不会返回给应用程序。

- Error Number: 8024

  非法的事务。当事务执行时，发现没有获取事务的 ID (Start Timestamp)，代表正在执行的事务是一个非法的事务，将返回该错误。通常情况下不会出现该问题，当发生时，请向 PingCAP 工程师或通过官方论坛寻求帮助。

- Error Number: 8025

  写入的单条键值对过大。TiDB 默认支持最大 6MB 的单个键值对，超过该限制可适当调整`txn-entry-size-limit`配置项以放宽限制。

- Error Number: 8026

  使用了没有实现的接口函数。该错误仅用于数据库内部，应用程序不会收到这个错误。

- Error Number: 8027

  表结构版本过期。TiDB 采用在线变更表结构的方法。当 TiDB server 表结构版本落后于整个系统的时，执行 SQL 将遇到该错误。遇到该错误，请检查该 TiDB server 与 PD leader 之间的网络。

- Error Number: 8028

  TiDB 没有表锁（在 MySQL 中称为元数据锁，在其他数据库中可能称为意向锁）。当事务执行时，TiDB 表结构发生了变化是无法被事务感知到的。因此，TiDB 在事务提交时，会对事务涉及表的结构进行检查。如果事务执行中表结构发生了变化，则事务将提交失败，并返回该错误。遇到该错误，应用程序可以安全地重新执行整个事务。

- Error Number: 8029

  当数据库内部进行数值转换发生错误时，将会出现该错误，该错误仅在内部使用，对外部应用将转换为具体类型的错误。

- Error Number: 8030

  将值转变为带符号正整数时发生了越界，将结果显示为负数。多在告警信息里出现。

- Error Number: 8031

  将负数转变为无符号数时，将负数转变为了正数。多在告警信息里出现。

- Error Number: 8032

  使用了非法的 year 格式。year 只允许 1 位、2 位和 4 位数。

- Error Number: 8033

  使用了非法的 year 值。year 的合法范围是 (1901, 2155)。

- Error Number: 8037

  week 函数中使用了非法的 mode 格式。mode 必须是一位数字，范围 [0, 7]。

- Error Number: 8038

  字段无法获取到默认值。一般作为内部错误使用，转换成其他具体错误类型后，返回给应用程序。

- Error Number: 8040

  尝试进行不支持的操作，比如在 View 和 Sequence 上进行 lock table。

- Error Number: 8047

  设置了不支持的系统变量值，通常在用户设置了数据库不支持的变量值后的告警信息里出现。

- Error Number: 8048

  设置了不支持的隔离级别，如果是使用第三方工具或框架等无法修改代码进行适配的情况，可以考虑通过 `tidb_skip_isolation_level_check` 来绕过这一检查。

```
set @@tidb_skip_isolation_level_check = 1;
```

- Error Number: 8050

  设置了不支持的权限类型，遇到该错误请参考TiDB 权限说明进行调整。

- Error Number: 8051

  TiDB 在解析客户端发送的 Exec 参数列表时遇到了未知的数据类型。如果遇到这个错误，请检查客户端是否正常，如果客户端正常请向 PingCAP 工程师或通过官方论坛寻求帮助。

- Error Number: 8052

  来自客户端的数据包的序列号错误。如果遇到这个错误，请检查客户端是否正常，如果客户端正常请向 PingCAP 工程师或通过官方论坛寻求帮助。

- Error Number: 8055

  当前快照过旧，数据可能已经被 GC。可以调大 `tikv_gc_life_time` 的值来避免该问题。新版本的 TiDB 会自动为长时间运行的事务保留数据，一般不会遇到该错误。有关 GC 的介绍和配置可以参考GC 机制简介和GC 配置文档。

```
update mysql.tidb set VARIABLE_VALUE="24h" where VARIABLE_NAME="tikv_gc_life_time";
```

- Error Number: 8059

  自动随机量可用次数用尽无法进行分配。当前没有恢复这类错误的方法。建议在使用 auto random 功能时使用 bigint 以获取最大的可分配次数，并尽量避免手动给 auto random 列赋值。相关的介绍和使用建议可以参考auto random 功能文档。

- Error Number: 8060

  非法的自增列偏移量。请检查 `auto_increment_increment` 和 `auto_increment_offset` 的取值是否符合要求。

- Error Number: 8061

  不支持的 SQL Hint。请参考Optimizer Hints 检查和修正 SQL Hint。

- Error Number: 8062

  SQL Hint 中使用了非法的 token，与 Hint 的保留字冲突。请参考Optimizer Hints 检查和修正 SQL Hint。

- Error Number: 8063

  SQL Hint 中限制内存使用量超过系统设置的上限，设置被忽略。请参考Optimizer Hints 检查和修正 SQL Hint。

- Error Number: 8064

  解析 SQL Hint 失败。请参考Optimizer Hints 检查和修正 SQL Hint。

- Error Number: 8065

  SQL Hint 中使用了非法的整数。请参考Optimizer Hints 检查和修正 SQL Hint。

- Error Number: 8066

  JSON_OBJECTAGG 函数的第二个参数是非法参数。

- Error Number: 8101

  插件 ID 格式错误，正确的格式是 `[name]-[version]` 并且 name 和 version 中不能带有 '-'。

- Error Number: 8102

  无法读取插件定义信息。请检查插件相关的配置。

- Error Number: 8103

  插件名称错误，请检查插件的配置。

- Error Number: 8104

  插件版本不匹配，请检查插件的配置。

- Error Number: 8105

  插件被重复载入。

- Error Number: 8106

  插件定义的系统变量名称没有以插件名作为开头，请联系插件的开发者进行修复。

- Error Number: 8107

  载入的插件未指定版本或指定的版本过低，请检查插件的配置。

- Error Number: 8108

  不支持的执行计划类型。该错误为内部处理的错误，如果遇到该报错请向 PingCAP 工程师或通过官方论坛寻求帮助。

- Error Number: 8109

  analyze 索引时找不到指定的索引。

- Error Number: 8110

  不能进行笛卡尔积运算，需要将配置文件里的 `cross-join` 设置为 `true`。

- Error Number: 8111

  execute 语句执行时找不到对应的 prepare 语句。

- Error Number: 8112

  execute 语句的参数个数与 prepare 语句不符合。

- Error Number: 8113

  execute 语句涉及的表结构在 prepare 语句执行后发生了变化。

- Error Number: 8115

  不支持 prepare 多行语句。

- Error Number: 8116

  不支持 prepare DDL 语句。

- Error Number: 8120

  获取不到事务的 start tso，请检查 PD Server 状态/监控/日志以及 TiDB Server 与 PD Server 之间的网络。

- Error Number: 8121

  权限检查失败，请检查数据库的权限配置。

- Error Number: 8122

  指定了通配符，但是找不到对应的表名。

- Error Number: 8123

  带聚合函数的 SQL 中返回非聚合的列，违反了 `only_full_group_by` 模式。请修改 SQL 或者考虑关闭 `only_full_group_by` 模式。

- Error Number: 8129

  TiDB 尚不支持键长度 >= 65536 的 JSON 对象。

- Error Number: 8200

  尚不支持的 DDL 语法。请参考与 MySQL DDL 的兼容性。

- Error Number: 8214

  DDL 操作被 admin cancel 操作终止。

- Error Number: 8215

  Admin Repair 表失败，如果遇到该报错请向 PingCAP 工程师或通过官方论坛寻求帮助。

- Error Number: 8216

  自动随机列使用的方法不正确，请参考auto random 功能文档进行修改。

- Error Number: 8223

  检测出数据与索引不一致的错误，如果遇到该报错请向 PingCAP 工程师或通过官方论坛寻求帮助。

- Error Number: 8224

  找不到 DDL job，请检查 restore 操作指定的 job id 是否存在。

- Error Number: 8225

  DDL 已经完成，无法被取消。

- Error Number: 8226

  DDL 几乎要完成了，无法被取消。

- Error Number: 8227

  创建 Sequence 时使用了不支持的选项，支持的选项的列表可以参考Sequence 使用文档。

- Error Number: 8228

  在 Sequence 上使用 `setval` 时指定了不支持的类型，该函数的示例可以在Sequence 使用文档中找到。

- Error Number: 8229

  事务超过存活时间，遇到该问题可以提交或者回滚当前事务，开启一个新事务。

- Error Number: 8230

  TiDB 目前不支持在新添加的列上使用 Sequence 作为默认值，如果尝试进行这类操作会返回该错误。

- Error Number: 9001

  请求 PD 超时，请检查 PD Server 状态/监控/日志以及 TiDB Server 与 PD Server 之间的网络。

- Error Number: 9002

  请求 TiKV 超时，请检查 TiKV Server 状态/监控/日志以及 TiDB Server 与 TiKV Server 之间的网络。

- Error Number: 9003

  TiKV 操作繁忙，一般出现在数据库负载比较高时，请检查 TiKV Server 状态/监控/日志。

- Error Number: 9004

  当数据库上承载的业务存在大量的事务冲突时，会遇到这种错误，请检查业务代码。

- Error Number: 9005

  某个 Raft Group 不可用，如副本数目不足，出现在 TiKV 比较繁忙或者是 TiKV 节点停机的时候，请检查 TiKV Server 状态/监控/日志。

- Error Number: 9006

  GC Life Time 间隔时间过短，长事务本应读到的数据可能被清理了，应增加 GC Life Time。

- Error Number: 9007

  事务在 TiKV 中遇到了写入冲突，原因及解决方法请参考这里。

- Error Number: 9008

  同时向 TiKV 发送的请求过多，超过了限制。请调大 `tidb_store_limit` 或将其设置为 0 来取消对请求流量的限制。

- Error Number: 9010

  TiKV 无法处理这条 raft log，请检查 TiKV Server 状态/监控/日志。

- Error Number: 9012

  请求 TiFlash 超时。请检查 TiFlash Server 状态/监控/日志以及 TiDB Server 与 TiFlash Server 之间的网络。

- Error Number: 9013

  TiFlash 操作繁忙。该错误一般出现在数据库负载比较高时。请检查 TiFlash Server 的状态/监控/日志。

12.14.2 故障诊断

参见故障诊断文档以及FAQ。

12.15 通过拓扑 label 进行副本调度

为了提升 TiDB 集群的高可用性和数据容灾能力，我们推荐让 TiKV 节点尽可能在物理层面上分散，例如让 TiKV 节点分布在不同的机架甚至不同的机房。PD 调度器根据 TiKV 的拓扑信息，会自动在后台通过调度使得 Region 的各个副本尽可能隔离，从而使得数据容灾能力最大化。

要让这个机制生效，需要在部署时进行合理配置，把集群的拓扑信息（特别是 TiKV 的位置）上报给 PD。阅读本章前，请先确保阅读TiUP 部署方案。

12.15.1 根据集群拓扑配置 labels

12.15.1.1 设置 TiKV 的 labels 配置

TiKV 支持在命令行参数或者配置文件中以键值对的形式绑定一些属性，我们把这些属性叫做标签（label）。TiKV 在启动后，会将自身的标签上报给 PD，因此我们可以使用标签来标识 TiKV 节点的地理位置。

比如集群的拓扑结构分成三层：机房（zone) -> 机架（rack）-> 主机（host），就可以使用这 3 个标签来设置 TiKV 的位置。

使用命令行参数的方式：

```
tikv-server --labels zone=<zone>,rack=<rack>,host=<host>
```

使用配置文件的方式：

```
[server]
labels = "zone=<zone>,rack=<rack>,host=<host>"
```

12.15.1.2 设置 PD 的 location-labels 配置

根据前面的描述，标签可以是用来描述 TiKV 属性的任意键值对，但 PD 无从得知哪些标签是用来标识地理位置的，而且也无从得知这些标签的层次关系。因此，PD 也需要一些配置来使得 PD 理解 TiKV 节点拓扑。

PD 上的配置叫做 location-labels，在集群初始化之前，可以通过 PD 的配置文件进行配置。

```
[replication]
location-labels = ["zone", "rack", "host"]
```

如果需要在 PD 集群初始化完成后进行配置，则需要使用 pd-ctl 工具进行在线更改：

```
pd-ctl config set location-labels zone,rack,host
```

其中，location-labels 配置是一个字符串数组，每一项与 TiKV 的 labels 的 key 是对应的，且其中每个 key 的顺序代表了不同标签的层次关系。

> 注意：
>
> 必须同时配置 PD 的 location-labels 和 TiKV 的 labels 参数，否则 PD 不会根据拓扑结构进行调度。

12.15.1.3 使用 TiUP 进行配置（推荐）

如果使用 TiUP 部署集群，可以在初始化配置文件中统一进行 location 相关配置。TiUP 会负责在部署时生成对应的 TiKV 和 PD 配置文件。

下面的例子定义了 zone/host 两层拓扑结构。集群的 TiKV 分布在三个 zone，每个 zone 内有两台主机，其中 z1 每台主机部署两个 TiKV 实例，z2 和 z3 每台主机部署 1 个实例。以下例子中 tikv-n 代表第 n 个 TiKV 节点的 IP 地址。

```
server_configs:
  pd:
    replication.location-labels: ["zone", "host"]

tikv_servers:
## z1
  - host: tikv-1
    config:
      server.labels:
        zone: z1
        host: h1
   - host: tikv-2
    config:
      server.labels:
        zone: z1
        host: h1
  - host: tikv-3
    config:
      server.labels:
        zone: z1
        host: h2
  - host: tikv-4
    config:
      server.labels:
        zone: z1
        host: h2
## z2
  - host: tikv-5
    config:
      server.labels:
        zone: z2
        host: h1
   - host: tikv-6
    config:
      server.labels:
        zone: z2
        host: h2
## z3
```

```
  - host: tikv-7
    config:
      server.labels:
        zone: z3
        host: h1
  - host: tikv-8
    config:
      server.labels:
        zone: z3
        host: h2
```

详情参阅TiUP 跨数据中心部署拓扑。

使用 TiDB Ansible 进行配置

如果使用 TiDB Ansible 部署集群，可以直接在 `inventory.ini` 文件中统一进行 location 相关配置。`tidb-ansible` 会负责在部署时生成对应的 TiKV 和 PD 配置文件。

下面的例子定义了 zone/host 两层拓扑结构。集群的 TiKV 分布在三个 zone，每个 zone 内有两台主机，其中 z1 每台主机部署两个 TiKV 实例，z2 和 z3 每台主机部署 1 个实例。

```
[tikv_servers]
## z1
tikv-1 labels="zone=z1,host=h1"
tikv-2 labels="zone=z1,host=h1"
tikv-3 labels="zone=z1,host=h2"
tikv-4 labels="zone=z1,host=h2"
## z2
tikv-5 labels="zone=z2,host=h1"
tikv-6 labels="zone=z2,host=h2"
## z3
tikv-7 labels="zone=z3,host=h1"
tikv-8 labels="zone=z3,host=h2"

[pd_servers:vars]
location_labels = ["zone", "host"]
```

12.15.2 基于拓扑 label 的 PD 调度策略

PD 在副本调度时，会按照 label 层级，保证同一份数据的不同副本尽可能分散。

下面以上一节的拓扑结构为例分析。

假设集群副本数设置为 3（`max-replicas=3`），因为总共有 3 个 zone，PD 会保证每个 Region 的 3 个副本分别放置在 z1/z2/z3，这样当任何一个数据中心发生故障时，TiDB 集群依然是可用的。

假如集群副本数设置为 5（`max-replicas=5`），因为总共只有 3 个 zone，在这一层级 PD 无法保证各个副本的隔离，此时 PD 调度器会退而求其次，保证在 host 这一层的隔离。也就是说，会出现一个 Region 的多个副本分布在同一个 zone 的情况，但是不会出现多个副本分布在同一台主机。

在 5 副本配置的前提下，如果 z3 出现了整体故障或隔离，并且 z3 在一段时间后仍然不能恢复（由 max-store ↪ -down-time 控制），PD 会通过调度补齐 5 副本，此时可用的主机只有 3 个了，故而无法保证 host 级别的隔离，于是可能出现多个副本被调度到同一台主机的情况。

总的来说，PD 能够根据当前的拓扑结构使得集群容灾能力最大化，所以如果我们希望达到某个级别的容灾能力，就需要根据拓扑结构在对应级别提供多于副本数 (max-replicas) 的机器。

13 常见问题解答 (FAQ)

13.1 FAQ

13.1.1 一、TiDB 介绍、架构、原理

13.1.1.1 1.1 TiDB 介绍及整体架构

13.1.1.1.1 1.1.1 TiDB 整体架构

TiDB 简介

13.1.1.1.2 1.1.2 TiDB 是什么？

TiDB 是一个分布式 NewSQL 数据库。它支持水平弹性扩展、ACID 事务、标准 SQL、MySQL 语法和 MySQL 协议，具有数据强一致的高可用特性，是一个不仅适合 OLTP 场景还适合 OLAP 场景的混合数据库。

13.1.1.1.3 1.1.3 TiDB 是基于 MySQL 开发的吗？

不是，虽然 TiDB 支持 MySQL 语法和协议，但是 TiDB 是由 PingCAP 团队完全自主开发的产品。

13.1.1.1.4 1.1.4 TiDB、 TiKV、 Placement Driver (PD) 主要作用？

- TiDB 是 Server 计算层，主要负责 SQL 的解析、制定查询计划、生成执行器。
- TiKV 是分布式 Key-Value 存储引擎，用来存储真正的数据，简而言之，TiKV 是 TiDB 的存储引擎。
- PD 是 TiDB 集群的管理组件，负责存储 TiKV 的元数据，同时也负责分配时间戳以及对 TiKV 做负载均衡调度。

13.1.1.1.5 1.1.5 TiDB 易用性如何？

TiDB 使用起来很简单，可以将 TiDB 集群当成 MySQL 来用，你可以将 TiDB 用在任何以 MySQL 作为后台存储服务的应用中，并且基本上不需要修改应用代码，同时你可以用大部分流行的 MySQL 管理工具来管理 TiDB。

13.1.1.1.6 1.1.6 TiDB 和 MySQL 兼容性如何？

TiDB 目前还不支持触发器、存储过程、自定义函数、外键，除此之外，TiDB 支持绝大部分 MySQL 5.7 的语法。

详情参见与 MySQL 兼容性对比。

13.1.1.1.7 1.1.7 TiDB 支持分布式事务吗？

支持。无论是一个地方的几个节点，还是跨多个数据中心的多个节点，TiDB 均支持 ACID 分布式事务。

TiDB 事务模型灵感源自 Google Percolator 模型，主体是一个两阶段提交协议，并进行了一些实用的优化。该模型依赖于一个时间戳分配器，为每个事务分配单调递增的时间戳，这样就检测到事务冲突。在 TiDB 集群中，PD 承担时间戳分配器的角色。

13.1.1.1.8 1.1.8 TiDB 支持哪些编程语言？

只要支持 MySQL Client/Driver 的编程语言，都可以直接使用 TiDB。

13.1.1.1.9 1.1.9 TiDB 是否支持其他存储引擎？

是的，除了 TiKV 之外，TiDB 还支持一些流行的单机存储引擎，比如 GolevelDB、RocksDB、BoltDB 等。如果一个存储引擎是支持事务的 KV 引擎，并且能提供一个满足 TiDB 接口要求的 Client，即可接入 TiDB。

13.1.1.1.10 1.1.10 除了官方文档，有没有其他 TiDB 知识获取途径？

目前官方文档是获取 TiDB 相关知识最主要、最及时的发布途径。除此之外，我们也有一些技术沟通群，如有需求可发邮件至 info@pingcap.com 获取，以及 AskTUG 网站 与技术专家互动交流。

13.1.1.1.11 1.1.11 TiDB 用户名长度限制？

在 TiDB 中用户名最长为 32 字符。

13.1.1.1.12 1.1.12 TiDB 是否支持 XA？

虽然 TiDB 的 JDBC 驱动用的就是 MySQL JDBC（Connector / J），但是当使用 Atomikos 的时候，数据源要配置成类似这样的配置：`type="com.mysql.jdbc.jdbc2.optional.MysqlXADataSource"`。MySQL JDBC XADataSource 连接 TiDB 的模式目前是不支持的。MySQL JDBC 中配置好的 XADataSource 模式，只对 MySQL 数据库起作用（DML 去修改 redo 等）。

Atomikos 配好两个数据源后，JDBC 驱动都要设置成 XA 模式，然后 Atomikos 在操作 TM 和 RM（DB）的时候，会通过数据源的配置，发起带有 XA 指令到 JDBC 层，JDBC 层 XA 模式启用的情况下，会对 InnoDB（如果是 MySQL 的话）下发操作一连串 XA 逻辑的动作，包括 DML 去变更 redo log 等，就是两阶段递交的那些操作。TiDB 目前的引擎版本中，没有对上层应用层 JTA / XA 的支持，不解析这些 Atomikos 发过来的 XA 类型的操作。

MySQL 是单机数据库，只能通过 XA 来满足跨数据库事务，而 TiDB 本身就通过 Google 的 Percolator 事务模型支持分布式事务，性能稳定性比 XA 要高出很多，所以不会也不需要支持 XA。

13.1.1.2 1.2 TiDB 原理

13.1.1.2.1 1.2.1 存储 TiKV

1.2.1.1 TiKV 详细解读

三篇文章了解 TiDB 技术内幕 - 说存储

13.1.1.2.2 1.2.2 计算 TiDB

1.2.2.1 TiDB 详细解读

三篇文章了解 TiDB 技术内幕 - 说计算

13.1.1.2.3 1.2.3 调度 PD

1.2.3.1 PD 详细解读

三篇文章了解 TiDB 技术内幕 - 谈调度

13.1.2 二、云上部署

13.1.2.1 2.1 公有云

13.1.2.1.1 2.1.1 目前 TiDB 云上部署都支持哪些云厂商？

关于云上部署，TiDB 支持在 Google GKE、AWS EKS 和 阿里云 ACK 上部署使用。

此外，TiDB 云上部署也已在京东云、UCloud 上线，均为数据库一级入口，欢迎大家使用。

13.1.3 三、故障排除

13.1.3.1 3.1 TiDB 自定义报错汇总

13.1.3.1.1 3.1.1 ERROR 8005 (HY000) : Write Conflict, txnStartTS is stale

可以检查 `tidb_disable_txn_auto_retry` 是否为 on。如是，将其设置为 off；如已经是 off，将 `tidb_retry_limit` 调大到不再发生该错误。

13.1.3.1.2 3.1.2 ERROR 9001 (HY000) : PD Server Timeout

请求 PD 超时，请检查 PD Server 状态/监控/日志以及 TiDB Server 与 PD Server 之间的网络。

13.1.3.1.3 3.1.3 ERROR 9002 (HY000) : TiKV Server Timeout

请求 TiKV 超时，请检查 TiKV Server 状态/监控/日志以及 TiDB Server 与 TiKV Server 之间的网络。

13.1.3.1.4 3.1.4 ERROR 9003 (HY000) : TiKV Server is Busy

TiKV 操作繁忙，一般出现在数据库负载比较高时，请检查 TiKV Server 状态/监控/日志。

13.1.3.1.5 3.1.5 ERROR 9004 (HY000) : Resolve Lock Timeout

清理锁超时，当数据库上承载的业务存在大量的事务冲突时，会遇到这种错误，请检查业务代码是否有锁争用。

13.1.3.1.6 3.1.6 ERROR 9005 (HY000) : Region is unavailable

访问的 Region 不可用，某个 Raft Group 不可用，如副本数目不足，出现在 TiKV 比较繁忙或者是 TiKV 节点停机的时候，请检查 TiKV Server 状态/监控/日志。

13.1.3.1.7 3.1.7 ERROR 9006 (HY000) : GC life time is shorter than transaction duration

GC Life Time 间隔时间过短，长事务本应读到的数据可能被清理了，可使用如下命令增加 GC Life Time：

```
update mysql.tidb set variable_value='30m' where variable_name='tikv_gc_life_time';
```

其中 30m 代表仅清理 30 分钟前的数据，这可能会额外占用一定的存储空间。

13.1.3.1.8 3.1.8 ERROR 9007 (HY000) : Write Conflict

可以检查 tidb_disable_txn_auto_retry 是否为 on。如是，将其设置为 off；如已经是 off，将 tidb_retry_limit 调大到不再发生该错误。

13.1.3.1.9 3.1.9 ERROR 8130 (HY000): client has multi-statement capability disabled

从早期版本的 TiDB 升级后，可能会出现该问题。为了减少 SQL 注入攻击的影响，TiDB 目前默认不允许在同一 COM_QUERY 调用中执行多个查询。

可通过系统变量tidb_multi_statement_mode 控制是否在同一 COM_QUERY 调用中执行多个查询。

13.1.3.2 3.2 MySQL 原生报错汇总

13.1.3.2.1 3.2.1 ERROR 2013 (HY000): Lost connection to MySQL server during query 问题的排查方法？

- log 中是否有 panic
- dmesg 中是否有 oom，命令：dmesg -T | grep -i oom
- 长时间没有访问，也会收到这个报错，一般是 tcp 超时导致的，tcp 长时间不用, 会被操作系统 kill。

13.1.3.2.2 3.2.2 ERROR 1105 (HY000): other error: unknown error Wire Error(InvalidEnumValue(4004)) 是什么意思？

这类问题一般是 TiDB 和 TiKV 版本不匹配，在升级过程尽量一起升级，避免版本 mismatch。

13.1.3.2.3 3.2.3 ERROR 1148 (42000): the used command is not allowed with this TiDB version 问题的处理方法？

这个问题是因为在执行 LOAD DATA LOCAL 语句的时候，MySQL 客户端不允许执行此语句（即 local_infile 选项为 0 ）。解决方法是在启动 MySQL 客户端时，用 --local-infile=1 选项。具体启动指令类似：mysql --local
↪ -infile=1 -u root -h 127.0.0.1 -P 4000。有些 MySQL 客户端需要设置而有些不需要设置，原因是不同版本的 MySQL 客户端对 local-infile 的默认值不同。

13.1.3.2.4 3.2.4 ERROR 9001 (HY000): PD server timeout start timestamp may fall behind safe point

这个报错一般是 TiDB 访问 PD 出了问题，TiDB 后台有个 worker 会不断地从 PD 查询 safepoint，如果超过 100s 查不成功就会报这个错。一般是因为 PD 磁盘操作过忙、反应过慢，或者 TiDB 和 PD 之间的网络有问题。TiDB 常见错误码请参考错误码与故障诊断。

13.1.3.3 3.3 TiDB 日志中的报错信息

13.1.3.3.1 3.3.1 EOF

当客户端或者 proxy 断开连接时，TiDB 不会立刻察觉连接已断开，而是等到开始往连接返回数据时，才发现连接已断开，此时日志会打印 EOF 错误。

13.2 SQL 操作常见问题

本文档介绍 TiDB 中常见的 SQL 操作问题。

13.2.1 TiDB 对哪些 MySQL variables 兼容？

详细可参考系统变量。

13.2.2 省略 ORDER BY 条件时 TiDB 中返回结果的顺序与 MySQL 中的不一致

这不是 bug。返回结果的顺序视不同情况而定，不保证顺序统一。

MySQL 中，返回结果的顺序可能较为固定，因为查询是通过单线程执行的。但升级到新版本后，查询计划也可能变化。无论是否期待返回结果有序，都推荐使用 ORDER BY 条件。

ISO/IEC 9075:1992, Database Language SQL- July 30, 1992 对此有如下表述：

> If an <order by clause> is not specified, then the table specified by the <cursor specification> is T and the ordering of rows in T is implementation-dependent. (如果未指定 <order by 条件>，通过 <cursor specification> 指定的表为 T，那么 T 表中的行顺序视执行情况而定。) 以下两条查询的结果都是合法的：

```
> select * from t;
+------+------+
| a    | b    |
+------+------+
|    1 |    1 |
|    2 |    2 |
+------+------+
2 rows in set (0.00 sec)
```

```
> select * from t; -- 不确定返回结果的顺序
+------+------+
| a    | b    |
+------+------+
|    2 |    2 |
```

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

如果 ORDER BY 中使用的列不是唯一列，该语句就不确定返回结果的顺序。以下示例中，a 列有重复值，因此只有 ORDER BY a, b 能确定返回结果的顺序。

```
> select * from t order by a;
+------+------+
| a    | b    |
+------+------+
|    1 |    1 |
|    2 |    1 |
|    2 |    2 |
+------+------+
3 rows in set (0.00 sec)
```

```
> select * from t order by a; -- 能确定 a 列的顺序，不确定 b 列的顺序
+------+------+
| a    | b    |
+------+------+
|    1 |    1 |
|    2 |    2 |
|    2 |    1 |
+------+------+
3 rows in set (0.00 sec)
```

13.2.3 TiDB 是否支持 SELECT FOR UPDATE？

支持。当 TiDB 使用悲观锁（自 TiDB v3.0 起默认使用）时，TiDB 中 SELECT FOR UPDATE 的行为与 MySQL 中的基本一致。

当 TiDB 使用乐观锁时，SELECT FOR UPDATE 不会在事务启动时对数据加锁，而是在提交事务时检查冲突。如果检查出冲突，会回滚待提交的事务。

13.2.4 TiDB 的 codec 能保证 UTF8 的字符串是 memcomparable 的吗？我们的 key 需要支持 UTF8，有什么编码建议吗？

TiDB 字符集默认就是 UTF8 而且目前只支持 UTF8，字符串就是 memcomparable 格式的。

13.2.5 一个事务中的语句数量最大是多少？

一个事务中的语句数量，默认限制最大为 5000 条。

13.2.6 TiDB 中，为什么出现后插入数据的自增 ID 反而小？

TiDB 的自增 ID (AUTO_INCREMENT) 只保证自增且唯一，并不保证连续分配。TiDB 目前采用批量分配的方式，所以如果在多台 TiDB 上同时插入数据，分配的自增 ID 会不连续。当多个线程并发往不同的 TiDB-server 插入数据的时候，有可能会出现后插入的数据自增 ID 小的情况。此外，TiDB 允许给整型类型的字段指定 AUTO_INCREMENT，且一个表只允许一个属性为 AUTO_INCREMENT 的字段。详情可参考CREATE TABLE 语法。

13.2.7 如何在 TiDB 中修改 sql_mode？

TiDB 支持将sql_mode 作为系统变量修改，与 MySQL 一致。目前，TiDB 不支持在配置文件中修改 sql_mode，但使用SET GLOBAL 对系统变量的修改将应用于集群中的所有 TiDB server，并且重启后更改依然有效。

13.2.8 用 Sqoop 批量写入 TiDB 数据，虽然配置了 --batch 选项，但还是会遇到 java.sql.BatchUpdateExecption: statement count 5001 exceeds the transaction limitation 的错误，该如何解决？

- 在 Sqoop 中，--batch 是指每个批次提交 100 条 statement，但是默认每个 statement 包含 100 条 SQL 语句，所以此时 100 * 100 = 10000 条 SQL 语句，超出了 TiDB 的事务限制 5000 条，可以增加选项 -Dsqoop.export.records.per.statement=10 来解决这个问题，完整的用法如下：

```
sqoop export \
    -Dsqoop.export.records.per.statement=10 \
    --connect jdbc:mysql://mysql.example.com/sqoop \
    --username sqoop ${user} \
    --password ${passwd} \
    --table ${tab_name} \
    --export-dir ${dir} \
    --batch
```

- 也可以选择增大 TiDB 的单个事物语句数量限制，不过此操作会导致内存增加。

13.2.9 TiDB 有像 Oracle 那样的 Flashback Query 功能么，DDL 支持么？

有，也支持 DDL。详细参考TiDB 历史数据回溯。

13.2.10 TiDB 中删除数据后会立即释放空间吗？

DELETE，TRUNCATE 和 DROP 都不会立即释放空间。对于 TRUNCATE 和 DROP 操作，在达到 TiDB 的 GC (garbage collection) 时间后（默认 10 分钟），TiDB 的 GC 机制会删除数据并释放空间。对于 DELETE 操作 TiDB 的 GC 机制会删除数据，但不会释放空间，而是当后续数据写入 RocksDB 且进行 compact 时对空间重新利用。

13.2.11 TiDB 是否支持 REPLACE INTO 语法？

支持，例外是当前 LOAD DATA 不支持 REPLACE INTO 语法。

13.2.12 数据删除后查询速度为何会变慢？

大量删除数据后，会有很多无用的 key 存在，影响查询效率。可以尝试开启Region Merge 功能，具体看参考最佳实践中的删除数据部分。

13.2.13 对数据做删除操作之后，空间回收比较慢，如何处理？

可以设置并行 GC，加快对空间的回收速度。默认并发为 1，最大可调整为 TiKV 实例数量的 50%。可使用 `update mysql.tidb set VARIABLE_VALUE="3" where VARIABLE_NAME="tikv_gc_concurrency";` 命令来调整。

13.2.14 SHOW PROCESSLIST 是否显示系统进程号？

TiDB 的 SHOW PROCESSLIST 与 MySQL 的 SHOW PROCESSLIST 显示内容基本一样，不会显示系统进程号，而 ID 表示当前的 session ID。其中 TiDB 的 `show processlist` 和 MySQL 的 `show processlist` 区别如下：

- 由于 TiDB 是分布式数据库，TiDB server 实例是无状态的 SQL 解析和执行引擎（详情可参考TiDB 整体架构），用户使用 MySQL 客户端登录的是哪个 TiDB server，`show processlist` 就会显示当前连接的这个 TiDB server 中执行的 session 列表，不是整个集群中运行的全部 session 列表；而 MySQL 是单机数据库，`show processlist` 列出的是当前整个 MySQL 数据库的全部执行 SQL 列表。
- 在查询执行期间，TiDB 中的 `State` 列不会持续更新。由于 TiDB 支持并行查询，每个语句可能同时处于多个状态，因此很难显示为某一种状态。

13.2.15 在 TiDB 中如何控制或改变 SQL 提交的执行优先级？

TiDB 支持改变per-session、全局或单个语句的优先级。优先级包括：

- HIGH_PRIORITY：该语句为高优先级语句，TiDB 在执行阶段会优先处理这条语句
- LOW_PRIORITY：该语句为低优先级语句，TiDB 在执行阶段会降低这条语句的优先级

以上两种参数可以结合 TiDB 的 DML 语言进行使用，使用方法举例如下：

1. 通过在数据库中写 SQL 的方式来调整优先级：

```
select HIGH_PRIORITY | LOW_PRIORITY count(*) from table_name;
insert HIGH_PRIORITY | LOW_PRIORITY into table_name insert_values;
delete HIGH_PRIORITY | LOW_PRIORITY from table_name;
update HIGH_PRIORITY | LOW_PRIORITY table_reference set assignment_list where
    ↪ where_condition;
replace HIGH_PRIORITY | LOW_PRIORITY into table_name;
```

2. 全表扫会自动调整为低优先级，analyze 也是默认低优先级。

13.2.16 在 TiDB 中 auto analyze 的触发策略是怎样的？

触发策略：新表达到 1000 条，并且在 1 分钟内没有写入，会自动触发。

当表的（修改数/当前总行数）大于 `tidb_auto_analyze_ratio` 的时候，会自动触发 `analyze` 语句。`tidb_auto_analyze_ratio` 的默认值为 0.5，即默认开启此功能。为了保险起见，在开启此功能的时候，保证了其最小值为 0.3。但是不能大于等于 `pseudo-estimate-ratio`（默认值为 0.8），否则会有一段时间使用 pseudo 统计信息，建议设置值为 0.5。

13.2.17 可以使用 Hints 控制优化器行为吗？

在 TiDB 中，你可以用多种方法控制查询优化器的默认行为，包括使用Optimizer Hints 和SQL 执行计划管理 (SPM)。基本用法同 MySQL 中的一致，还包含若干 TiDB 特有的用法，示例如下：`select column_name from table_name use index(index_name) where where_condition;`

13.2.18 触发 Information schema is changed 错误的原因？

TiDB 在执行 SQL 语句时，会使用当时的 schema 来处理该 SQL 语句，而且 TiDB 支持在线异步变更 DDL。那么，在执行 DML 的时候可能有 DDL 语句也在执行，而你需要确保每个 SQL 语句在同一个 schema 上执行。所以当执行 DML 时，遇到正在执行中的 DDL 操作就可能会报 `Information schema is changed` 的错误。为了避免太多的 DML 语句报错，已做了一些优化。

现在会报此错的可能原因如下（只有第一个报错原因与表有关）：

- 执行的 DML 语句中涉及的表和集群中正在执行的 DDL 的表有相同的，那么这个 DML 语句就会报此错。
- 这个 DML 执行时间很久，而这段时间内执行了很多 DDL 语句，导致中间 schema 版本变更次数超过 1024（此为默认值，可以通过 `tidb_max_delta_schema_count` 变量修改）。
- 接受 DML 请求的 TiDB 长时间不能加载到 `schema information`（TiDB 与 PD 或 TiKV 之间的网络连接故障等会导致此问题），而这段时间内执行了很多 DDL 语句，导致中间 schema 版本变更次数超过 100。
- TiDB 重启后执行第一个 DDL 操作前，执行 DML 操作，并且在执行过程中遇到了第 1 个 DDL 操作（即在执行第 1 个 DDL 操作前，启动该 DML 对应的事务，且在该 DDL 变更第一个 schema 版本后，提交该 DML 对应的事务），那么这个 DML 会报此错。

> 注意：
>
> - 目前 TiDB 未缓存所有的 schema 版本信息。
> - 对于每个 DDL 操作，schema 版本变更的数量与对应 `schema state` 变更的次数一致。
> - 不同的 DDL 操作版本变更次数不一样。例如，`create table` 操作会有 1 次 schema 版本变更；`add column` 操作有 4 次 schema 版本变更。

13.2.19 触发 Information schema is out of date 错误的原因？

当执行 DML 时，TiDB 超过一个 DDL lease 时间（默认 45s）没能加载到最新的 schema 就可能会报 `Information schema is out of date` 的错误。遇到此错的可能原因如下：

- 执行此 DML 的 TiDB 被 kill 后准备退出，且此 DML 对应的事务执行时间超过一个 DDL lease，在事务提交时会报这个错误。
- TiDB 在执行此 DML 时，有一段时间内连不上 PD 或者 TiKV，导致 TiDB 超过一个 DDL lease 时间没有 load schema，或者导致 TiDB 断开与 PD 之间带 keep alive 设置的连接。

13.2.20 高并发情况下执行 DDL 时报错的原因？

高并发情况下执行 DDL（比如批量建表）时，极少部分 DDL 可能会由于并发执行时 key 冲突而执行失败。

并发执行 DDL 时，建议将 DDL 数量保持在 20 以下，否则你需要在应用端重试失败的 DDL 语句。

13.2.21 SQL 优化

13.2.21.1 TiDB 执行计划解读

详细解读理解 TiDB 执行计划。

13.2.21.2 统计信息收集

详细解读统计信息。

13.2.21.3 Count 如何加速？

Count 就是暴力扫表，提高并发度能显著的提升速度，修改并发度可以参考 `tidb_distsql_scan_concurrency` 变量，但是也要看 CPU 和 I/O 资源。TiDB 每次查询都要访问 TiKV，在数据量小的情况下，MySQL 都在内存里，TiDB 还需要进行一次网络访问。

提升建议：

- 建议提升硬件配置，可以参考部署建议。
- 提升并发度，默认是 10，可以提升到 50 试试，但是一般提升在 2-4 倍之间。
- 测试大数据量的 count。
- 调优 TiKV 配置，可以参考性能调优。
- 可以参考下推计算结果缓存。

13.2.21.4 查看当前 DDL 的进度？

通过 `admin show ddl` 查看当前 job 进度。操作如下：

```
admin show ddl;
```

```
*************************** 1. row ***************************
  SCHEMA_VER: 140
       OWNER: 1a1c4174-0fcd-4ba0-add9-12d08c4077dc
RUNNING_JOBS: ID:121, Type:add index, State:running, SchemaState:write reorganization, SchemaID
    ↪ :1, TableID:118, RowCount:77312, ArgLen:0, start time: 2018-12-05 16:26:10.652 +0800 CST,
    ↪  Err:<nil>, ErrCount:0, SnapshotVersion:404749908941733890
     SELF_ID: 1a1c4174-0fcd-4ba0-add9-12d08c4077dc
```

从上面操作结果可知，当前正在处理的是 add index 操作。且从 RUNNING_JOBS 列的 RowCount 字段可以知道当前 add index 操作已经添加了 77312 行索引。

13.2.21.5 如何查看 DDL job？

可以使用 admin show ddl 语句查看正在运行的 DDL 作业。

- admin show ddl jobs：用于查看当前 DDL 作业队列中的所有结果（包括正在运行以及等待运行的任务）以及已执行完成的 DDL 作业队列中的最近十条结果。
- admin show ddl job queries 'job_id' [, 'job_id'] ...：用于显示 job_id 对应的 DDL 任务的原始 SQL 语句。此 job_id 只搜索正在执行中的任务以及 DDL 历史作业队列中的最近十条结果。

13.2.21.6 TiDB 是否支持基于 COST 的优化（CBO），如果支持，实现到什么程度？

是的，TiDB 使用的是基于成本的优化器 (CBO)，会对代价模型、统计信息持续优化。除此之外，TiDB 还支持 hash join、 soft-merge join 等 join 算法。

13.2.21.7 如何确定某张表是否需要做 analyze ？

可以通过 show stats_healthy 来查看 Healthy 字段，一般小于等于 60 的表需要做 analyze。

13.2.21.8 SQL 的执行计划展开成了树，ID 的序号有什么规律吗？这棵树的执行顺序会是怎么样的？

ID 没什么规律，只要是唯一就行，不过生成的时候，是有一个计数器，生成一个 plan 就加一，执行的顺序和序号无关，整个执行计划是一颗树，执行时从根节点开始，不断地向上返回数据。执行计划的理解，请参考理解 TiDB 执行计划。

13.2.21.9 TiDB 执行计划中，task cop 在一个 root 下，这个是并行的么？

目前 TiDB 的计算任务隶属于两种不同的 task：cop task 和 root task。cop task 是指被下推到 KV 端分布式执行的计算任务，root task 是指在 TiDB 端单点执行的计算任务。一般来讲 root task 的输入数据是来自于 cop task 的；但是 root task 在处理数据的时候，TiKV 上的 cop task 也可以同时处理数据，等待 TiDB 的 root task 拉取，所以从这个观点上来看，他们是并行的；但是存在数据上下游关系；在执行的过程中，某些时间段其实也是并行的，第一个 cop task 在处理 [100, 200] 的数据，第二个 cop task 在处理 [1, 100] 的数据。执行计划的理解，请参考理解 TiDB 执行计划。

13.2.22 数据库优化

13.2.22.1 TiDB 参数及调整

详情参考TiDB 配置参数。

13.2.22.2 如何打散热点

TiDB 中以 Region 分片来管理数据库，通常来讲，TiDB 的热点指的是 Region 的读写访问热点。而 TiDB 中对于非整数主键或没有主键的表，可以通过设置 SHARD_ROW_ID_BITS 来适度分解 Region 分片，以达到打散 Region 热点的效果。详情可参考官网SHARD_ROW_ID_BITS 中的介绍。

13.2.22.3 TiKV 性能参数调优

详情参考TiKV 性能参数调优。

13.3 部署运维 FAQ

本文介绍 TiDB 集群运维部署的常见问题、原因及解决方法。

13.3.1 环境准备 FAQ

操作系统版本要求如下表：

Linux 操作系统平台	版本
Red Hat Enterprise Linux	7.3 及以上
CentOS	7.3 及以上
Oracle Enterprise Linux	7.3 及以上

13.3.1.1 为什么要在 CentOS 7 上部署 TiDB 集群？

TiDB 作为一款开源分布式 NewSQL 数据库，可以很好的部署和运行在 Intel 架构服务器环境及主流虚拟化环境，并支持绝大多数的主流硬件网络，作为一款高性能数据库系统，TiDB 支持主流的 Linux 操作系统环境，具体可以参考 TiDB 的官方部署要求。

其中 TiDB 在 CentOS 7.3 的环境下进行大量的测试，同时也有很多这个操作系统的部署最佳实践，因此，我们推荐客户在部署 TiDB 的时候使用 CentOS 7.3+ 以上的 Linux 操作系统。

13.3.2 硬件要求 FAQ

TiDB 支持部署和运行在 Intel x86-64 架构的 64 位通用硬件服务器平台。对于开发、测试、及生产环境的服务器硬件配置有以下要求和建议：

13.3.2.1 开发及测试环境

组件	CPU	内存	本地存储	网络	实例数量 (最低要求)
TiDB	8 核 +	16 GB+	SAS, 200 GB+	千兆网卡	1（可与 PD 同机器）
PD	8 核 +	16 GB+	SAS, 200 GB+	千兆网卡	1（可与 TiDB 同机器）
TiKV	8 核 +	32 GB+	SSD, 200 GB+	千兆网卡	3
				服务器总计	4

13.3.2.2 生产环境

组件	CPU	内存	硬盘类型	网络	实例数量 (最低要求)
TiDB	16 核 +	48 GB+	SAS	万兆网卡（2 块最佳）	2

组件	CPU	内存	硬盘类型	网络	实例数量 (最低要求)
PD	8 核 +	16 GB+	SSD	万兆网卡（2 块最佳）	3
TiKV	16 核 +	48 GB+	SSD	万兆网卡（2 块最佳）	3
监控	8 核 +	16 GB+	SAS	千兆网卡	1
				服务器总计	9

13.3.2.3 两块网卡的目的是？万兆的目的是？

作为一个分布式集群，TiDB 对时间的要求还是比较高的，尤其是 PD 需要分发唯一的时间戳，如果 PD 时间不统一，如果有 PD 切换，将会等待更长的时间。两块网卡可以做 bond，保证数据传输的稳定，万兆可以保证数据传输的速度，千兆网卡容易出现瓶颈，我们强烈建议使用万兆网卡。

13.3.2.4 SSD 不做 RAID 是否可行？

资源可接受的话，我们建议做 RAID 10，如果资源有限，也可以不做 RAID。

13.3.2.5 TiDB 集群各个组件的配置推荐？

- TiDB 需要 CPU 和内存比较好的机器，参考官网配置要求，如果后期需要开启 TiDB Binlog，根据业务量的评估和 GC 时间的要求，也需要本地磁盘大一点，不要求 SSD 磁盘；
- PD 里面存了集群元信息，会有频繁的读写请求，对磁盘 I/O 要求相对比较高，磁盘太差会影响整个集群性能，推荐 SSD 磁盘，空间不用太大。另外集群 Region 数量越多对 CPU、内存的要求越高；
- TiKV 对 CPU、内存、磁盘要求都比较高，一定要用 SSD 磁盘。

详情可参考TiDB 软硬件环境需求。

13.3.3 安装部署 FAQ

如果用于生产环境，推荐使用 TiUP 使用 TiUP 部署 TiDB 集群。

13.3.3.1 为什么修改了 TiKV/PD 的 toml 配置文件，却没有生效？

这种情况一般是因为没有使用 `--config` 参数来指定配置文件（目前只会出现在 binary 部署的场景），TiKV/PD 会按默认值来设置。如果要使用配置文件，请设置 TiKV/PD 的 `--config` 参数。对于 TiKV 组件，修改配置后重启服务即可；对于 PD 组件，只会在第一次启动时读取配置文件，之后可以使用 pd-ctl 的方式来修改配置，详情可参考PD 配置参数。

13.3.3.2 TiDB 监控框架 Prometheus + Grafana 监控机器建议单独还是多台部署？

监控机建议单独部署。建议 CPU 8 core，内存 16 GB 以上，硬盘 500 GB 以上。

13.3.3.3 有一部分监控信息显示不出来？

查看访问监控的机器时间跟集群内机器的时间差，如果比较大，更正时间后即可显示正常。

13.3.3.4 supervise/svc/svstat 服务具体起什么作用？

- supervise 守护进程
- svc 启停服务
- svstat 查看进程状态

13.3.3.5 inventory.ini 变量参数解读

变量	含义
cluster_name	集群名称，可调整
tidb_version	TiDB 版本，TiDB Ansible 各分支默认已配置
deployment_method	部署方式，默认为 binary，可选 docker
process_supervision	进程监管方式，默认为 systemd，可选 supervise

变量	含义
timezone	修改部署目标机器时区，默认为 Asia/Shanghai, 可调整，与 set_timezone 变量结合使用
set_timezone	默认为 True，即修改部署目标机器时区，关闭可修改为 False
enable_elk	目前不支持，请忽略
enable_firewall	开启防火墙，默认不开启

变量	含义
enable_ntpd	检测部署目标机器 NTP 服务，默认为 True，请勿关闭
machine_benchmark	检测部署目标机器磁盘 IOPS，默认为 True，请勿关闭
set_hostname_by_ip	根据 IP 修改部署目标机器主机名，默认为 False

变量	含义
enable_binlog	是否部署 pump 并开启 binlog，默认为 False，依赖 Kafka 集群，参见 zookeeper_addrs 变量
zookeeper_addrs	Kafka 集群的 zookeeper 地址
enable_slow_query_log	TiDB 慢查询日志记录到单独文件 ({{ deploy_dir }}/log/tidb_slow_query.log)，默认为 False，记录到 tidb 日志

变量	含义
deploy_without_tidb	KV 模式，不部署 TiDB 服务，仅部署 PD、TiKV 及监控服务，请将 inventory.ini 文件中 tidb_servers 主机组 IP 设置为空。

13.3.3.6 TiDB 离线 Ansible 部署方案（4.0 版本后不推荐使用）

首先这不是我们建议的方式，如果中控机没有外网，也可以通过离线 Ansible 部署方式，详情可参考离线 TiDB Ansible 部署方案。

13.3.3.7 Docker Compose 快速构建集群（单机部署）

使用 docker-compose 在本地一键拉起一个集群，包括集群监控，还可以根据需求自定义各个组件的软件版本和实例个数，以及自定义配置文件，这种只限于开发环境，详细可参考官方文档。

13.3.3.8 如何单独记录 TiDB 中的慢查询日志，如何定位慢查询 SQL？

1）TiDB 中，对慢查询的定义在 tidb-ansible 的 `conf/tidb.yml` 配置文件中，`slow-threshold: 300`，这个参数是配置慢查询记录阈值的，单位是 ms。

慢查询日志默认记录到 tidb.log 中，如果希望生成单独的慢查询日志文件，修改 inventory.ini 配置文件的参数 `enable_slow_query_log` 为 True。

如上配置修改之后，需要执行 `ansible-playbook rolling_update.yml --tags=tidb`，对 tidb-server 实例进行滚动升级，升级完成后，tidb-server 将在 `tidb_slow_query.log` 文件中记录慢查询日志。

2）如果出现了慢查询，可以从 Grafana 监控定位到出现慢查询的 tidb-server 以及时间点，然后在对应节点查找日志中记录的 SQL 信息。

3）除了日志，还可以通过 `admin show slow` 命令查看，详情可参考admin show slow 命令。

13.3.3.9 首次部署 TiDB 集群时，没有配置 tikv 的 Label 信息，在后续如何添加配置 Label？

TiDB 的 Label 设置是与集群的部署架构相关的，是集群部署中的重要内容，是 PD 进行全局管理和调度的依据。如果集群在初期部署过程中没有设置 Label，需要在后期对部署结构进行调整，就需要手动通过 PD 的管理工具 pd-ctl 来添加 location-labels 信息，例如：`config set location-labels "zone,rack,host"`（根据实际的 label 层级名字配置）。

pd-ctl 的使用参考PD Control 使用说明。

13.3.3.10 为什么测试磁盘的 dd 命令用 `oflag=direct` 这个选项？

Direct 模式就是把写入请求直接封装成 I/O 指令发到磁盘，这样是为了绕开文件系统的缓存，可以直接测试磁盘的真实的 I/O 读写能力。

13.3.3.11 如何用 fio 命令测试 TiKV 实例的磁盘性能？

- 随机读测试：

```
./fio -ioengine=psync -bs=32k -fdatasync=1 -thread -rw=randread -size=10G -filename=
    ↪ fio_randread_test.txt -name='fio randread test' -iodepth=4 -runtime=60 -numjobs=4 -
    ↪ group_reporting --output-format=json --output=fio_randread_result.json
```

- 顺序写和随机读混合测试：

```
./fio -ioengine=psync -bs=32k -fdatasync=1 -thread -rw=randrw -percentage_random=100,0 -size
    ↪ =10G -filename=fio_randread_write_test.txt -name='fio mixed randread and sequential
    ↪ write test' -iodepth=4 -runtime=60 -numjobs=4 -group_reporting --output-format=json
    ↪ --output=fio_randread_write_test.json
```

13.3.3.12 使用 TiDB Ansible 部署 TiDB 集群的时候，遇到 `UNREACHABLE! "msg": "Failed to connect to the host via ssh: "` 报错是什么原因？

有两种可能性：

- ssh 互信的准备工作未做好，建议严格参照我们的官方文档步骤配置互信，并使用命令 `ansible -i inventory.ini all -m shell -a 'whoami' -b` 来验证互信配置是否成功。
- 如果涉及到单服务器分配了多角色的场景，例如多组件混合部署或单台服务器部署了多个 TiKV 实例，可能是由于 ssh 复用的机制引起这个报错，可以使用 `ansible ... -f 1` 的选项来规避这个报错。

13.3.4 集群管理 FAQ

13.3.4.1 集群日常管理

13.3.4.1.1 Ansible 常见运维操作有那些？

任务	Playbook
启动集群	ansible-playbook start.yml
停止集群	ansible-playbook stop.yml
销毁集群	ansible-playbook unsafe_cleanup.yml (若部署目录为挂载点，会报错，可忽略)
清除数据 (测试用)	ansible-playbook cleanup_data.yml
滚动升级	ansible-playbook rolling_update.yml
滚动升级 TiKV	ansible-playbook rolling_update.yml –tags=tikv
滚动升级除 PD 外模块	ansible-playbook rolling_update.yml –skip-tags=pd
滚动升级监控组件	ansible-playbook rolling_update_monitor.yml

13.3.4.1.2 TiDB 如何登录？

和 MySQL 登录方式一样，可以按照下面例子进行登录。

```
mysql -h 127.0.0.1 -uroot -P4000
```

13.3.4.1.3 TiDB 如何修改数据库系统变量？

和 MySQL 一样，TiDB 也分为静态参数和固态参数，静态参数可以直接通过`set global xxx = n`的方式进行修改，不过新参数值只限于该实例生命周期有效。

13.3.4.1.4 TiDB (TiKV) 有哪些数据目录？

默认在`--data-dir` 目录下，其中包括 backup、db、raft、snap 四个目录，分别存储备份、数据、raft 数据及镜像数据。

13.3.4.1.5 TiDB 有哪些系统表？

和 MySQL 类似，TiDB 中也有系统表，用于存放数据库运行时所需信息，具体信息参考TiDB 系统数据库文档。

13.3.4.1.6 TiDB 各节点服务器下是否有日志文件，如何管理？

默认情况下各节点服务器会在日志中输出标准错误，如果启动的时候通过 `--log-file` 参数指定了日志文件，那么日志会输出到指定的文件中，并且按天做 rotation。

13.3.4.1.7 如何规范停止 TiDB？

如果是用 TiDB Ansible 部署的，可以使用 `ansible-playbook stop.yml` 命令停止 TiDB 集群。如果不是 TiDB Ansible 部署的，可以直接 kill 掉所有服务。如果使用 kill 命令，TiDB 的组件会做 graceful 的 shutdown。

13.3.4.1.8 TiDB 里面可以执行 kill 命令吗？

- 可以 kill DML 语句，首先使用 `show processlist`，找到对应 session 的 id，然后执行 `kill tidb [session id]`。
- 可以 kill DDL 语句，首先使用 `admin show ddl jobs`，查找需要 kill 的 DDL job ID，然后执行 `admin cancel ddl jobs 'job_id' [, 'job_id'] ...`。具体可以参考admin 操作。

13.3.4.1.9 TiDB 是否支持会话超时？

TiDB 暂不支持数据库层面的会话超时，目前想要实现超时，在没 LB（Load Balancing）的时候，需要应用侧记录发起的 Session 的 ID，通过应用自定义超时，超时以后需要到发起 Query 的节点上用 `kill tidb [session id]` 来杀掉 SQL。目前建议使用应用程序来实现会话超时，当达到超时时间，应用层就会抛出异常继续执行后续的程序段。

13.3.4.1.10 TiDB 生产环境的版本管理策略是怎么样的？如何尽可能避免频繁升级？

TiDB 版本目前逐步标准化，每次 Release 都包含详细的 Change log，版本功能变化详情，生产环境是否有必要升级取决于业务系统，建议升级之前详细了解前后版本的功能差异。

版本号说明参考：Release Version: `v1.0.3-1-ga80e796`

- `v1.0.3` 表示 GA 标准版
- `1` 表示该版本 commit 1 次
- `ga80e796` 代表版本的 `git-hash`

13.3.4.1.11 分不清 TiDB master 版本之间的区别，经常用错 TiDB Ansible 版本?

TiDB 目前社区非常活跃，在 1.0 GA 版本发布后，还在不断的优化和修改 BUG，因此 TiDB 的版本更新周期比较快，会不定期有新版本发布，请关注我们的新版本发布官方网站。此外 TiDB 安装推荐使用 TiUP 进行安装。此外，在 TiDB 1.0 GA 版本后，对 TiDB 的版本号进行了统一管理，TiDB 的版本可以通过以下两种方式进行查看：

- 通过 `select tidb_version()` 进行查看
- 通过执行 `tidb-server -V` 进行查看

13.3.4.1.12 有没有图形化部署 TiDB 的工具？

暂时没有。

13.3.4.1.13 TiDB 如何进行水平扩展？

当您的业务不断增长时，数据库可能会面临三方面瓶颈，第一是存储空间，第二是计算资源，第三是读写容量，这时可以对 TiDB 集群做水平扩展。

- 如果是存储资源不够，可以通过添加 TiKV Server 节点来解决，新节点启动后，PD 会自动将其他节点的部分数据迁移过去，无需人工介入。
- 如果是计算资源不够，可以查看 TiDB Server 和 TiKV Server 节点的 CPU 消耗情况，再考虑添加 TiDB Server 节点或者是 TiKV Server 节点来解决，如添加 TiDB Server 节点，将其添加到前端 Load Balancer 配置之中即可。
- 如果是容量跟不上，一般可以考虑同时增加 TiDB Server 和 TiKV Server 节点。

13.3.4.1.14 Percolator 用了分布式锁，crash 的客户端会保持锁，会造成锁没有 release？

详细可参考 Percolator 和 TiDB 事务算法。

13.3.4.1.15 TiDB 为什么选用 gRPC 而不选用 Thrift，是因为 Google 在用吗？

不只是因为 Google 在用，有一些比较好的特性我们需要，比如流控、加密还有 Streaming。

13.3.4.1.16 like(bindo.customers.name, jason%, 92) 这个 92 代表什么？

那个是转义字符，默认是 (ASCII 92)。

13.3.4.1.17 为什么 `information_schema.tables.data_length` 记录的大小和 TiKV 监控面板上的 store size 不一样？

这是因为两者计算的角度不一样。`information_schema.tables.data_length` 是通过统计信息（平均每行的大小）得到的估算值。TiKV 监控面板上的 store size 是单个 TiKV 实例的数据文件（RocksDB 的 SST 文件）的大小总和。由于多版本和 TiKV 会压缩数据，所以两者显示的大小不一样。

13.3.4.2 PD 管理

13.3.4.2.1 访问 PD 报错：TiKV cluster is not bootstrapped

PD 的大部分 API 需要在初始化 TiKV 集群以后才能使用，如果在部署新集群的时候只启动了 PD，还没有启动 TiKV，这时候访问 PD 就会报这个错误。遇到这个错误应该先把要部署的 TiKV 启动起来，TiKV 会自动完成初始化工作，然后就可以正常访问 PD。

13.3.4.2.2 PD 启动报错：etcd cluster ID mismatch

PD 启动参数中的 `--initial-cluster` 包含了某个不属于该集群的成员。遇到这个错误时请检查各个成员的所属集群，剔除错误的成员后即可正常启动。

13.3.4.2.3 PD 能容忍的时间同步误差是多少？

理论上，时间同步误差越小越好。PD 可容忍任意时长的误差，但是，时间同步误差越大意味着 PD 分配的时间戳与真实的物理时间相差越大，这个差距会影响读历史版本等功能。

13.3.4.2.4 Client 连接是如何寻找 PD 的？

Client 连接只能通过 TiDB 访问集群，TiDB 负责连接 PD 与 TiKV，PD 与 TiKV 对 Client 透明。当 TiDB 连接任意一台 PD 的时候，PD 会告知 TiDB 当前的 leader 是谁，如果此台 PD 不是 leader，TiDB 将会重新连接至 leader PD。

13.3.4.2.5 TiKV 节点 (Store) 各状态 (Up, Disconnect, Offline, Down, Tombstone) 之间的关系是什么？

使用 `pd-ctl` 可以查看 TiKV 节点的状态信息。如需查看各个状态之间的关系，请参考TiKV Store 状态之间的关系。

13.3.4.2.6 PD 参数中 leader-schedule-limit 和 region-schedule-limit 调度有什么区别？

- leader-schedule-limit 调度是用来均衡不同 TiKV 的 leader 数，影响处理查询的负载。
- region-schedule-limit 调度是均衡不同 TiKV 的副本数，影响不同节点的数据量。

13.3.4.2.7 每个 region 的 replica 数量可配置吗？调整的方法是？

可以，目前只能调整全局的 replica 数量。首次启动时 PD 会读配置文件（conf/pd.yml），使用其中的 max-replicas 配置，之后修改需要使用 pd-ctl 配置命令 `config set max-replicas $num`，配置后可通过 `config show all` 来查看已生效的配置。调整的时候，不会影响业务，会在后台添加，注意总 TiKV 实例数总是要大于等于设置的副本数，例如 3 副本需要至少 3 个 TiKV。增加副本数量之前需要预估额外的存储需求。pd-ctl 的详细用法可参考PD Control 使用说明。

13.3.4.2.8 缺少命令行集群管理工具，整个集群的健康度当前是否正常，不好确认？

可以通过 pd-ctl 等工具来判断集群大概的状态，详细的集群状态还是需要通过监控来确认。

13.3.4.2.9 集群下线节点后，怎么删除老集群节点监控信息？

下线节点一般指 TiKV 节点通过 pd-ctl 或者监控判断节点是否下线完成。节点下线完成后，手动停止下线节点上相关的服务。从 Prometheus 配置文件中删除对应节点的 node_exporter 信息。从 Ansible inventory.ini 中删除对应节点的信息。

13.3.4.2.10 使用 PD Control 连接 PD Server 时，为什么只能通过本机 IP 连接，不能通过 127.0.0.1 连接？

因为使用 TiDB Ansible 部署的集群，PD 对外服务端口不会绑定到 127.0.0.1，所以 PD Control 不会识别 127.0.0.1。

13.3.4.3 TiDB server 管理

13.3.4.3.1 TiDB 的 lease 参数应该如何设置？

启动 TiDB Server 时，需要通过命令行参数设置 lease 参数（–lease=60），其值会影响 DDL 的速度（只会影响当前执行 DDL 的 session，其他的 session 不会受影响）。在测试阶段，lease 的值可以设为 1s，加快测试进度；在生产环境下，我们推荐这个值设为分钟级（一般可以设为 60），这样可以保证 DDL 操作的安全。

13.3.4.3.2 DDL 在正常情况下的耗时是多少？

一般情况下处理一个 DDL 操作（之前没有其他 DDL 操作在处理）的耗时基本可以分如下为三种：

- add index 操作，且此操作对应表数据行数比较少，耗时约为 3s。
- add index 操作，且此操作对应表数据行数比较多，耗时具体由表中数据行数和当时 QPS 情况定（add index 操作优先级比一般 SQL 低）。
- 其他 DDL 操作耗时约为 1s。

此外，如果接收 DDL 请求的 TiDB 和 DDL owner 所处的 TiDB 是一台，那么上面列举的第一和第三种可能的耗时应该在几十到几百毫秒。

13.3.4.3.3 为什么有的时候执行 DDL 会很慢？

可能原因如下：

- 多个 DDL 语句一起执行的时候，后面的几个 DDL 语句会比较慢。原因是当前 TiDB 集群中 DDL 操作是串行执行的。
- 在正常集群启动后，第一个 DDL 操作的执行时间可能会比较久，一般在 30s 左右，这个原因是刚启动时 TiDB 在竞选处理 DDL 的 leader。
- 由于停 TiDB 时不能与 PD 正常通信（包括停电情况）或者用 `kill -9` 指令停 TiDB 导致 TiDB 没有及时从 PD 清理注册数据，那么会影响 TiDB 启动后 10min 内的 DDL 语句处理时间。这段时间内运行 DDL 语句时，每个 DDL 状态变化都需要等待 2 * lease（默认 lease = 45s）。
- 当集群中某个 TiDB 与 PD 之间发生通信问题，即 TiDB 不能从 PD 及时获取或更新版本信息，那么这时候 DDL 操作的每个状态处理需要等待 2 * lease。

13.3.4.3.4 TiDB 可以使用 S3 作为后端存储吗？

不可以，目前 TiDB 只支持分布式存储引擎和 GolevelDB/RocksDB/BoltDB 引擎。

13.3.4.3.5 Information_schema 能否支持更多真实信息？

Information_schema 库里面的表主要是为了兼容 MySQL 而存在，有些第三方软件会查询里面的信息。在目前 TiDB 的实现中，里面大部分只是一些空表。后续随着 TiDB 的升级，会提供更多的参数信息。当前 TiDB 支持的 Information_schema 请参考TiDB 系统数据库说明文档。

13.3.4.3.6 TiDB Backoff type 主要原因?

TiDB-server 与 TiKV-server 随时进行通信，在进行大量数据操作过程中，会出现 `Server is busy` 或者 `backoff.maxsleep 20000ms` 的日志提示信息，这是由于 TiKV-server 在处理过程中系统比较忙而出现的提示信息，通常这时候可以通过系统资源监控到 TiKV 主机系统资源使用率比较高的情况出现。如果这种情况出现，可以根据资源使用情况进行相应的扩容操作。

13.3.4.3.7 TiDB TiClient type 主要原因？

TiClient Region Error 该指标描述的是在 TiDB-server 作为客户端通过 KV 接口访问 TiKV-server 进行数据操作过程中，TiDB-server 操作 TiKV-server 中的 Region 数据出现的错误类型与 metric 指标，错误类型包括 not_leader、stale_epoch。出现这些错误的情况是当 TiDB-server 根据自己的缓存信息去操作 Region leader 数据的时候，Region leader 发生了迁移或者 TiKV 当前的 Region 信息与 TiDB 缓存的路由信息不一致而出现的错误提示。一般这种情况下，TiDB-server 都会自动重新从 PD 获取最新的路由数据，重做之前的操作。

13.3.4.3.8 TiDB 同时支持的最大并发连接数？

默认情况下，每个 TiDB 服务器的最大连接数没有限制。如有需要，可以在 `config.toml` 文件中设置 `max-server-connections` 来限制最大连接数。如果并发量过大导致响应时间增加，建议通过添加 TiDB 节点进行扩容。

13.3.4.3.9 如何查看某张表创建的时间？

information_schema 库中的 tables 表里的 create_time 即为表的真实创建时间。

13.3.4.3.10 TiDB 的日志中 EXPENSIVE_QUERY 是什么意思？

TiDB 在执行 SQL 时，预估出来每个 operator 处理了超过 10000 条数据就认为这条 query 是 expensive query。可以通过修改 tidb-server 配置参数来对这个门限值进行调整，调整后需要重新启动 tidb-server。

13.3.4.4 TiKV 管理

13.3.4.4.1 TiKV 集群副本建议配置数量是多少，是不是最小高可用配置（3 个）最好？

如果是测试环境 3 副本足够；在生产环境中，不可让集群副本数低于 3，需根据架构特点、业务系统及恢复能力的需求，适当增加副本数。值得注意的是，副本升高，性能会有下降，但是安全性更高。

13.3.4.4.2 TiKV 启动报错：cluster ID mismatch

TiKV 本地存储的 cluster ID 和指定的 PD 的 cluster ID 不一致。在部署新的 PD 集群的时候，PD 会随机生成一个 cluster ID，TiKV 第一次初始化的时候会从 PD 获取 cluster ID 存储在本地，下次启动的时候会检查本地的 cluster ID 与 PD 的 cluster ID 是否一致，如果不一致则会报错并退出。出现这个错误一个常见的原因是，用户原先部署了一个集群，后来把 PD 的数据删除了并且重新部署了新的 PD，但是 TiKV 还是使用旧的数据重启连到新的 PD 上，就会报这个错误。

13.3.4.4.3 TiKV 启动报错：duplicated store address

启动参数中的地址已经被其他的 TiKV 注册在 PD 集群中了。造成该错误的常见情况：TiKV `--data-dir` 指定的路径下没有数据文件夹（删除或移动后没有更新 –data-dir），用之前参数重新启动该 TiKV。请尝试用 pd-ctl 的 store delete 功能，删除之前的 store，然后重新启动 TiKV 即可。

13.3.4.4.4 TiKV master 和 slave 用的是一样的压缩算法，为什么效果不一样?

目前来看 master 有些文件的压缩率会高一些，这个取决于底层数据的分布和 RocksDB 的实现，数据大小偶尔有些波动是正常的，底层存储引擎会根据需要调整数据。

13.3.4.4.5 TiKV block cache 有哪些特性？

TiKV 使用了 RocksDB 的 Column Family (CF) 特性，KV 数据最终存储在默认 RocksDB 内部的 default、write、lock 3 个 CF 内。

- default CF 存储的是真正的数据，与其对应的参数位于 `[rocksdb.defaultcf]` 项中。
- write CF 存储的是数据的版本信息（MVCC）、索引、小表相关的数据，相关的参数位于 `[rocksdb.writecf]` 项中。
- lock CF 存储的是锁信息，系统使用默认参数。
- Raft RocksDB 实例存储 Raft log。default CF 主要存储的是 Raft log，与其对应的参数位于 `[raftdb.defaultcf]` 项中。
- 所有 CF 共享一个 Block-cache，用于缓存数据块，加速 RocksDB 的读取速度，Block-cache 的大小通过参数 `block-cache-size` 控制，`block-cache-size` 越大，能够缓存的热点数据越多，对读取操作越有利，同时占用的系统内存也会越多。
- 每个 CF 有各自的 Write-buffer，大小通过 `write-buffer-size` 控制。

13.3.4.4.6 TiKV channel full 是什么原因？

- Raftstore 线程太忙，或者因 I/O 而卡住。可以看一下 Raftstore 的 CPU 使用情况。
- TiKV 过忙（CPU、磁盘 I/O 等），请求处理不过来。

13.3.4.4.7 TiKV 频繁切换 Region leader 是什么原因？

- 网络问题导致节点间通信卡了，查看 Report failures 监控。
- 原主 Leader 的节点卡了，导致没有及时给 Follower 发送消息。
- Raftstore 线程卡了。

13.3.4.4.8 如果一个节点挂了会影响服务吗？影响会持续多久？

TiDB 使用 Raft 在多个副本之间做数据同步（默认为每个 Region 3 个副本）。当一份备份出现问题时，其他的副本能保证数据的安全。根据 Raft 协议，当某个节点挂掉导致该节点里的 Leader 失效时，在最大 2 * lease time（leasetime 是 10 秒）时间后，通过 Raft 协议会很快将一个另外一个节点里的 Follower 选为新的 Region Leader 来提供服务。

13.3.4.4.9 TiKV 分别在哪些场景下占用大量 IO、内存、CPU（超过参数配置的多倍）?

在大量写入、读取的场景中会占用大量的磁盘 IO、内存、CPU。在执行很复杂的查询，比如会产生很大中间结果集的情况下，会消耗很多的内存和 CPU 资源。

13.3.4.4.10 TiKV 是否可以使用 SAS/SATA 盘或者进行 SSD/SAS 混合部署？

不可以使用，TiDB 在进行 OLTP 场景中，数据访问和操作需要高 IO 磁盘的支持，TiDB 作为强一致的分布式数据库，存在一定的写放大，如副本复制、存储底层 Compaction，因此，TiDB 部署的最佳实践中推荐用户使用 NVMe SSD 磁盘作为数据存储磁盘。另外，TiKV 与 PD 不能混合部署。

13.3.4.4.11 数据表 Key 的 Range 范围划分是在数据接入之前就已经划分好了吗？

不是的，这个和 MySQL 分表规则不一样，需要提前设置好，TiKV 是根据 Region 的大小动态分裂的。

13.3.4.4.12 Region 是如何进行分裂的？

Region 不是前期划分好的，但确实有 Region 分裂机制。当 Region 的大小超过参数 `region-max-size` 或 `region-max-keys` 的值时，就会触发分裂，分裂后的信息会汇报给 PD。

13.3.4.4.13 TiKV 是否有类似 MySQL 的 `innodb_flush_log_trx_commit` 参数，来保证提交数据不丢失？

是的，TiKV 单机的存储引擎目前使用两个 RocksDB 实例，其中一个存储 raft-log，TiKV 有个 sync-log 参数，在 ture 的情况下，每次提交都会强制刷盘到 raft-log，如果发生 crash 后，通过 raft-log 进行 KV 数据的恢复。

13.3.4.4.14 对 WAL 存储有什么推荐的硬件配置，例如 SSD，RAID 级别，RAID 卡 cache 策略，NUMA 设置，文件系统选择，操作系统的 IO 调度策略等？

WAL 属于顺序写，目前我们并没有单独对他进行配置，建议 SSD，RAID 如果允许的话，最好是 RAID 10，RAID 卡 cache、操作系统 I/O 调度目前没有针对性的最佳实践，Linux 7 以上默认配置即可，NUMA 没有特别建议，NUMA 内存分配策略可以尝试使用 `interleave = all`，文件系统建议 ext4。

13.3.4.4.15 在最严格的 `sync-log = true` 数据可用模式下，写入性能如何？

一般来说，开启 `sync-log` 会让性能损耗 30% 左右。关闭 `sync-log` 时的性能表现，请参见 TiDB Sysbench 性能测试报告。

13.3.4.4.16 是否可以利用 TiKV 的 Raft + 多副本达到完全的数据可靠，单机存储引擎是否需要最严格模式？

通过使用 Raft 一致性算法，数据在各 TiKV 节点间复制为多副本，以确保某个节点挂掉时数据的安全性。只有当数据已写入超过 50% 的副本时，应用才返回 ACK（三副本中的两副本）。但理论上两个节点也可能同时发生故障，所以除非是对性能要求高于数据安全的场景，一般都强烈推荐开启 `sync-log`。

另外，还有一种 `sync-log` 的替代方案，即在 Raft group 中用五个副本而非三个。这将允许两个副本同时发生故障，而仍然能保证数据安全性。

对于单机存储引擎也同样推荐打开 `sync-log` 模式。否则如果节点宕机可能会丢失最后一次写入数据。

13.3.4.4.17 使用 Raft 协议，数据写入会有多次网络的 roundtrip，实际写入延迟如何？

理论上，和单机数据库相比，数据写入会多四个网络延迟。

13.3.4.4.18 有没有类似 MySQL 的 InnoDB Memcached plugin，可以直接使用 KV 接口，可以不需要独立的 Cache？

TiKV 支持单独进行接口调用，理论上也可以起个实例做为 Cache，但 TiDB 最大的价值是分布式关系型数据库，我们原则上不对 TiKV 单独进行支持。

13.3.4.4.19 Coprocessor 组件的主要作用？

- 减少 TiDB 与 TiKV 之间的数据传输。
- 计算下推，充分利用 TiKV 的分布式计算资源。

13.3.4.4.20 IO error: No space left on device While appending to file

这是磁盘空间不足导致的，需要加节点或者扩大磁盘空间。

13.3.4.4.21 为什么 TiKV 容易出现 OOM？

TiKV 的内存占用主要来自于 RocksDB 的 block-cache，默认为系统总内存的 40%。当 TiKV 容易出现 OOM 时，检查 `block-cache-size` 配置是否过高。还需要注意，当单机部署了多个 TiKV 实例时，需要显式地配置该参数，以防止多个实例占用过多系统内存导致 OOM。

13.3.4.4.22 TiDB 数据和 RawKV 数据可存储于同一个 TiKV 集群里吗？

不可以。TiDB 数据（或使用其他事务 API 生成的数据）依赖于一种特殊的键值格式，和 RawKV API 数据（或其他基于 RawKV 的服务生成的数据）并不兼容。

13.3.4.5 TiDB 测试

13.3.4.5.1 TiDB Sysbench 基准测试结果如何？

很多用户在接触 TiDB 都习惯做一个基准测试或者 TiDB 与 MySQL 的对比测试，官方也做了一个类似测试，汇总很多测试结果后，我们发现虽然测试的数据有一定的偏差，但结论或者方向基本一致，由于 TiDB 与 MySQL 由于架构上的差别非常大，很多方面是很难找到一个基准点，所以官方的建议两点：

- 大家不要用过多精力纠结这类基准测试上，应该更多关注 TiDB 的场景上的区别。
- 大家可以直接参考 TiDB Sysbench 性能测试报告。

13.3.4.5.2 TiDB 集群容量 QPS 与节点数之间关系如何，和 MySQL 对比如何？

- 在 10 节点内，TiDB 写入能力（Insert TPS）和节点数量基本成 40% 线性递增，MySQL 由于是单节点写入，所以不具备写入扩展能力。
- MySQL 读扩容可以通过添加从库进行扩展，但写流量无法扩展，只能通过分库分表，而分库分表有很多问题，具体参考方案虽好，成本先行：数据库 Sharding+Proxy 实践解析。
- TiDB 不管是读流量、还是写流量都可以通过添加节点快速方便的进行扩展。

13.3.4.5.3 我们的 DBA 测试过 MySQL 性能，单台 TiDB 的性能没有 MySQL 性能那么好？

TiDB 设计的目标就是针对 MySQL 单台容量限制而被迫做的分库分表的场景，或者需要强一致性和完整分布式事务的场景。它的优势是通过尽量下推到存储节点进行并行计算。对于小表（比如千万级以下），不适合 TiDB，因为数据量少，Region 有限，发挥不了并行的优势，最极端的就是计数器表，几行记录高频更新，这几行在 TiDB 里，会变成存储引擎上的几个 KV，然后只落在一个 Region 里，而这个 Region 只落在一个节点上。加上后台强一致性复制的开销，TiDB 引擎到 TiKV 引擎的开销，最后表现出来的就是没有单个 MySQL 好。

13.3.4.6 TiDB 备份恢复

13.3.4.6.1 TiDB 主要备份方式？

目前，数据量大时推荐使用BR 进行备份。其他场景推荐使用Dumpling 进行备份。

尽管 TiDB 也支持使用 MySQL 官方工具 `mysqldump` 进行数据备份和恢复，但其性能低于Dumpling，并且 `mysqldump` 备份和恢复大量数据的耗费更长。

13.3.5 监控 FAQ

- Prometheus 监控框架详情可见TiDB 监控框架概述。
- 监控指标解读详细参考重要监控指标详解。

13.3.5.1 目前的监控使用方式及主要监控指标，有没有更好看的监控？

TiDB 使用 Prometheus + Grafana 组成 TiDB 数据库系统的监控系统，用户在 Grafana 上通过 dashboard 可以监控到 TiDB 的各类运行指标，包括系统资源的监控指标，包括客户端连接与 SQL 运行的指标，包括内部通信和 Region 调度的指标，通过这些指标，可以让数据库管理员更好的了解到系统的运行状态，运行瓶颈等内容。在监控指标的过程中，我们按照 TiDB 不同的模块，分别列出了各个模块重要的指标项，一般用户只需要关注这些常见的指标项。具体指标请参见官方文档。

13.3.5.2 Prometheus 监控数据默认 15 天自动清除一次，可以自己设定成 2 个月或者手动删除吗？

可以的，在 Prometheus 启动的机器上，找到启动脚本，然后修改启动参数，然后重启 Prometheus 生效。

```
--storage.tsdb.retention="60d"
```

13.3.5.3 Region Health 监控项

TiDB-2.0 版本中，PD metric 监控页面中，对 Region 健康度进行了监控，其中 Region Health 监控项是对所有 Region 副本状况的一些统计。其中 miss 是缺副本，extra 是多副本。同时也增加了按 Label 统计的隔离级别，level-1 表示这些 Region 的副本在第一级 Label 下是物理隔离的，没有配置 location label 时所有 Region 都在 level-0。

13.3.5.4 Statement Count 监控项中的 selectsimplefull 是什么意思？

代表全表扫，但是可能是很小的系统表。

13.3.5.5 监控上的 QPS 和 Statement OPS 有什么区别？

QPS 会统计执行的所有 SQL 命令，包括 use database、 load data、 begin、 commit、 set、 show、 insert、 select 等。

Statement OPS 只统计 select、 update、 insert 等业务相关的，所以 Statement OPS 的统计和业务比较相符。

13.4 升级与升级后常见问题

本文介绍 TiDB 升级与升级后的常见问题与解决办法。

13.4.1 升级常见问题

本小节列出了 TiDB 升级相关的常见问题与解决办法。

13.4.1.1 滚动升级有那些影响?

滚动升级 TiDB 服务，滚动升级期间不影响业务运行。需要配置最小集群拓扑（TiDB * 2、PD * 3、TiKV * 3），如果集群环境中有 Pump 和 Drainer 服务，建议先停止 Drainer，然后滚动升级（升级 TiDB 时会升级 Pump）。

13.4.1.2 Binary 如何升级？

Binary 不是建议的安装方式，对升级支持也不友好，建议换成TiUP 部署。

13.4.2 升级后常见问题

本小节列出了一些升级后可能会遇到的问题与解决办法。

13.4.2.1 执行 DDL 操作时遇到的字符集 (charset) 问题

TiDB 在 v2.1.0 以及之前版本（包括 v2.0 所有版本）中，默认字符集是 UTF8。从 v2.1.1 开始，默认字符集变更为 UTF8MB4。如果在 v2.1.0 及之前版本中，建表时显式指定了 table 的 charset 为 UTF8，那么升级到 v2.1.1 之后，执行 DDL 操作可能会失败。

要避免该问题，需注意以下两个要点：

- 在 v2.1.3 之前，TiDB 不支持修改 column 的 charset。所以，执行 DDL 操作时，新 column 的 charset 需要和旧 column 的 charset 保持一致。
- 在 v2.1.3 之前，即使 column 的 charset 和 table 的 charset 不一样，`show create table` 也不会显示 column 的 charset，但可以通过 HTTP API 获取 table 的元信息来查看 column 的 charset，下文提供了示例。

13.4.2.1.1 `unsupported modify column charset utf8mb4 not match origin utf8`

- 升级前：v2.1.0 及之前版本

```
create table t(a varchar(10)) charset=utf8;
```

```
Query OK, 0 rows affected
Time: 0.106s
```

```
show create table t
```

```
+-------+-------------------------------------------------------+
| Table | Create Table                                          |
+-------+-------------------------------------------------------+
| t     | CREATE TABLE `t` (                                    |
|       |   `a` varchar(10) DEFAULT NULL                        |
|       | ) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_bin |
+-------+-------------------------------------------------------+
1 row in set
Time: 0.006s
```

- 升级后：v2.1.1、v2.1.2 会出现下面的问题，v2.1.3 以及之后版本不会出现下面的问题。

```
alter table t change column a a varchar(20);
```

```
ERROR 1105 (HY000): unsupported modify column charset utf8mb4 not match origin utf8
```

解决方案：显式指定 column charset，保持和原来的 charset 一致即可。

```
alter table t change column a a varchar(22) character set utf8;
```

- 根据要点 1，此处如果不指定 column 的 charset，会用默认的 UTF8MB4，所以需要指定 column charset 保持和原来一致。
- 根据要点 2，用 HTTP API 获取 table 元信息，然后根据 column 名字和 Charset 关键字搜索即可找到 column 的 charset。

```
curl "http://$IP:10080/schema/test/t" | python -m json.tool
```

这里用了 python 的格式化 json 的工具，也可以不加，此处只是为了方便注释。

```
{
    "ShardRowIDBits": 0,
    "auto_inc_id": 0,
    "charset": "utf8",   # table 的 charset
    "collate": "",
    "cols": [            # 从这里开始列举 column 的相关信息
        {
            ...
            "id": 1,
            "name": {
                "L": "a",
                "O": "a"   # column 的名字
            },
            "offset": 0,
            "origin_default": null,
            "state": 5,
            "type": {
                "Charset": "utf8",   # column a 的 charset
                "Collate": "utf8_bin",
                "Decimal": 0,
                "Elems": null,
                "Flag": 0,
                "Flen": 10,
                "Tp": 15
            }
        }
    ],
    ...
}
```

13.4.2.1.2 unsupported modify charset from utf8mb4 to utf8

- 升级前：v2.1.1，v2.1.2

```
create table t(a varchar(10)) charset=utf8;
```

```
Query OK, 0 rows affected
Time: 0.109s
```

```
show create table t;
```

```
+-------+-------------------------------------------------------+
| Table | Create Table                                          |
+-------+-------------------------------------------------------+
| t     | CREATE TABLE `t` (                                    |
|       |   `a` varchar(10) DEFAULT NULL                        |
|       | ) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_bin |
+-------+-------------------------------------------------------+
```

上面 show create table 只显示出了 table 的 charset，但其实 column 的 charset 是 UTF8MB4，这可以通过 HTTP API 获取 schema 来确认。这是一个 bug，即此处建表时 column 的 charset 应该要和 table 保持一致为 UTF8，该问题在 v2.1.3 中已经修复。

- 升级后：v2.1.3 及之后版本

```
show create table t;
```

```
+-------+------------------------------------------------------------------------+
| Table | Create Table                                                           |
+-------+------------------------------------------------------------------------+
| t     | CREATE TABLE `t` (                                                     |
|       |   `a` varchar(10) CHARSET utf8mb4 COLLATE utf8mb4_bin DEFAULT NULL |
|       | ) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_bin                  |
+-------+------------------------------------------------------------------------+
1 row in set
Time: 0.007s
```

```
alter table t change column a a varchar(20);
```

```
ERROR 1105 (HY000): unsupported modify charset from utf8mb4 to utf8
```

解决方案：

- 因为在 v2.1.3 之后，TiDB 支持修改 column 和 table 的 charset，所以这里推荐修改 table 的 charset 为 UTF8MB4。

```
alter table t convert to character set utf8mb4;
```

- 也可以像问题 1 一样指定 column 的 charset，保持和 column 原来的 charset (UTF8MB4) 一致即可。

```
alter table t change column a a varchar(20) character set utf8mb4;
```

13.4.2.1.3 ERROR 1366 (HY000): incorrect utf8 value f09f8c80( ) for column a

TiDB 在 v2.1.1 及之前版本中，如果 charset 是 UTF8，没有对 4-byte 的插入数据进行 UTF8 Unicode encoding 检查。在 v2.1.2 及之后版本中，添加了该检查。

- 升级前：v2.1.1 及之前版本

```
create table t(a varchar(100) charset utf8);
```

```
Query OK, 0 rows affected
```

```
insert t values (unhex('f09f8c80'));
```

```
Query OK, 1 row affected
```

- 升级后：v2.1.2 及之后版本

```
insert t values (unhex('f09f8c80'));
```

```
ERROR 1366 (HY000): incorrect utf8 value f09f8c80( ) for column a
```

解决方案：

- v2.1.2 版本：该版本不支持修改 column charset，所以只能跳过 UTF8 的检查。

```
set @@session.tidb_skip_utf8_check=1;
```

```
Query OK, 0 rows affected
```

```
insert t values (unhex('f09f8c80'));
```

```
Query OK, 1 row affected
```

- v2.1.3 及之后版本：建议修改 column 的 charset 为 UTF8MB4。或者也可以设置 `tidb_skip_utf8_check` 变量跳过 UTF8 的检查。如果跳过 UTF8 的检查，在需要将数据从 TiDB 同步回 MySQL 的时候，可能会失败，因为 MySQL 会执行该检查。

```
alter table t change column a a varchar(100) character set utf8mb4;
```

```
Query OK, 0 rows affected
```

```
insert t values (unhex('f09f8c80'));
```

```
Query OK, 1 row affected
```

关于 tidb_skip_utf8_check 变量，具体来说是指跳过 UTF8 和 UTF8MB4 类型对数据的合法性检查。如果跳过这个检查，在需要将数据从 TiDB 同步回 MySQL 的时候，可能会失败，因为 MySQL 执行该检查。如果只想跳过 UTF8 类型的检查，可以设置 tidb_check_mb4_value_in_utf8 变量。

tidb_check_mb4_value_in_utf8 在 v2.1.3 版本加入 config.toml 文件，可以修改配置文件里面的 check-mb4-value-in-utf8 后重启集群生效。

tidb_check_mb4_value_in_utf8 在 v2.1.5 版本开始可以用 HTTP API 来设置，也可以用 session 变量来设置。

- HTTP API（HTTP API 只在单台服务器上生效）
  * 执行下列命令启用 HTTP API：

```
curl -X POST -d "check_mb4_value_in_utf8=1" http://{TiDBIP}:10080/settings
```

  * 执行下列命令禁用 HTTP API：

```
curl -X POST -d "check_mb4_value_in_utf8=0" http://{TiDBIP}:10080/settings
```

- Session 变量
  * 执行下列命令启用 Session 变量：

```
set @@session.tidb_check_mb4_value_in_utf8 = 1;
```

  * 执行下列命令禁用 Session 变量：

```
set @@session.tidb_check_mb4_value_in_utf8 = 0;
```

- v2.1.7 及之后版本，如果对表和 column 的字符集没有严格要求为 UTF8，也不想修改客户端代码去跳过 UTF8 检查或者手动修改 column 的 charset，可以在配置文件中把 treat-old-version-utf8-as-utf8mb4 打开。该配置的作用是自动把 v2.1.7 版本之前创建的旧版本的表和 column 的 UTF8 字符集转成 UTF8MB4。这个转换是在 TiDB load schema 时在内存中将 UTF8 转成 UTF8MB4，不会对实际存储的数据做任何修改。在配置文件中关闭 treat-old-version-utf8-as-utf8mb4 并重启 TiDB 后，以前字符集为 UTF8 的表和 column 的字符集仍然还是 UTF8。

> 注意：
>
> treat-old-version-utf8-as-utf8mb4 参数默认打开，如果客户端强制需要用 UTF8 而不用 UTF8MB4，需要在配置文件中关闭。

13.5 高可用常见问题

本文档介绍高可用相关的常见问题。

13.5.1 TiDB 数据是强一致的吗？

通过使用 Raft 一致性算法，数据在各 TiKV 节点间复制为多副本，以确保某个节点挂掉时数据的安全性。

在底层，TiKV 使用复制日志 + 状态机 (State Machine) 的模型来复制数据。对于写入请求，数据被写入 Leader，然后 Leader 以日志的形式将命令复制到它的 Follower 中。当集群中的大多数节点收到此日志时，日志会被提交，状态机会相应作出变更。

13.5.2 官方有没有三中心跨机房多活部署的推荐方案？

从 TiDB 架构来讲，支持真正意义上的跨中心异地多活，从操作层面讲，依赖数据中心之间的网络延迟和稳定性，一般建议延迟在 5ms 以下，目前我们已经有相似客户方案，具体请咨询官方 info@pingcap.com。

13.6 高可靠常见问题

本文档介绍高可靠相关的常见问题。

13.6.1 我们的安全漏洞扫描工具对 MySQL version 有要求，TiDB 是否支持修改 server 版本号呢？

TiDB 在 v3.0.8 后支持修改 server 版本号，可以通过配置文件中的`server-version`配置项进行修改。在使用 TiUP 部署集群时，可以通过 `tiup cluster edit-config <cluster-name>` 修改配置文件中以下部分来设置合适的版本号：

```
server_configs:
  tidb:
    server-version: 'YOUR_VERSION_STRING'
```

并通过 `tiup cluster reload <cluster-name> -R tidb` 命令使得以上修改生效，以避免出现安全漏洞扫描不通过的问题。

13.6.2 TiDB 支持哪些认证协议，过程是怎样的？

这一层跟 MySQL 一样，走的 SASL 认证协议，用于用户登录认证，对密码的处理流程。

客户端连接 TiDB 的时候，走的是 challenge-response（挑战 -应答）的认证模式，过程如下：

1. 客户端连接服务器；
2. 服务器发送随机字符串 `challenge` 给客户端；
3. 客户端发送 `username` + `response` 给服务器；
4. 服务器验证 `response`。

13.6.3 如何修改用户名密码和权限？

TiDB 作为分布式数据库，在 TiDB 中修改用户密码建议使用 `set password for 'root'@'%' = '0101001';` 或 `alter` 方法，不推荐使用 `update mysql.user` 的方法进行，这种方法可能会造成其它节点刷新不及时的情况。修改权限也一样，都建议采用官方的标准语法。详情可参考TiDB 用户账户管理。

13.7 迁移常见问题

13.7.1 全量数据导出导入

13.7.1.1 如何将一个运行在 MySQL 上的应用迁移到 TiDB 上？

TiDB 支持绝大多数 MySQL 语法，一般不需要修改代码。

13.7.1.2 导入导出速度慢，各组件日志中出现大量重试、EOF 错误并且没有其他错误

在没有其他逻辑出错的情况下，重试、EOF 可能是由网络问题引起的，建议首先使用相关工具排查网络连通状况。以下示例使用 iperf 进行排查：

- 在出现重试、EOF 错误的服务器端节点执行以下命令：

```
iperf3 -s
```

- 在出现重试、EOF 错误的客户端节点执行以下命令：

```
iperf3 -c <server-IP>
```

下面是一个网络连接良好的客户端节点的输出：

```
$ iperf3 -c 192.168.196.58
Connecting to host 192.168.196.58, port 5201
[  5] local 192.168.196.150 port 55397 connected to 192.168.196.58 port 5201
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-1.00   sec  18.0 MBytes   150 Mbits/sec
[  5]   1.00-2.00   sec  20.8 MBytes   175 Mbits/sec
[  5]   2.00-3.00   sec  18.2 MBytes   153 Mbits/sec
[  5]   3.00-4.00   sec  22.5 MBytes   188 Mbits/sec
[  5]   4.00-5.00   sec  22.4 MBytes   188 Mbits/sec
[  5]   5.00-6.00   sec  22.8 MBytes   191 Mbits/sec
[  5]   6.00-7.00   sec  20.8 MBytes   174 Mbits/sec
[  5]   7.00-8.00   sec  20.1 MBytes   168 Mbits/sec
[  5]   8.00-9.00   sec  20.8 MBytes   175 Mbits/sec
[  5]   9.00-10.00  sec  21.8 MBytes   183 Mbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-10.00  sec   208 MBytes   175 Mbits/sec                  sender
[  5]   0.00-10.00  sec   208 MBytes   174 Mbits/sec                  receiver

iperf Done.
```

如果输出显示网络带宽较低、带宽波动大，各组件日志中就可能出现大量重试、EOF 错误。此时你需要咨询网络服务供应商以提升网络质量。

如果输出的各指标良好，请尝试更新各组件版本。如果更新后仍无法解决问题，请移步 AskTUG 论坛寻求帮助。

13.7.1.3 不小心把 MySQL 的 user 表导入到 TiDB 了，或者忘记密码，无法登录，如何处理？

重启 TiDB 服务，配置文件中增加 `-skip-grant-table=true` 参数，无密码登录集群后，可以根据情况重建用户，或者重建 mysql.user 表，具体表结构搜索官网。

13.7.1.4 在 Loader 运行的过程中，TiDB 可以对外提供服务吗？

该操作进行逻辑插入，TiDB 仍可对外提供服务，但不要执行相关 DDL 操作。

13.7.1.5 如何导出 TiDB 数据？

你可以通过以下方式导出 TiDB 数据：

- 参考 MySQL 使用 mysqldump 导出某个表的部分数据，使用 mysqldump 加 where 条件导出。
- 使用 MySQL client 将 select 的结果输出到一个文件。

13.7.1.6 如何从 DB2、 Oracle 数据库迁移到 TiDB？

DB2、 Oracle 到 TiDB 数据迁移（增量 + 全量），通常做法有：

- 使用 Oracle 官方迁移工具，如 OGG、 Gateway（透明网关）、 CDC（Change Data Capture）。
- 自研数据导出导入程序实现。
- 导出（Spool）成文本文件，然后通过 Load infile 进行导入。
- 使用第三方数据迁移工具。

目前看来 OGG 最为合适。

13.7.1.7 用 Sqoop 批量写入 TiDB 数据，虽然配置了 `--batch` 选项，但还是会遇到 `java.sql.BatchUpdateExecption:statement count 5001 exceeds the transaction limitation` 的错误，该如何解决？

- 在 Sqoop 中，`--batch` 是指每个批次提交 100 条 statement，但是默认每个 statement 包含 100 条 SQL 语句，所以此时 100 * 100 = 10000 条 SQL 语句，超出了 TiDB 的事务限制 5000 条，可以增加选项 `-Dsqoop.export.records.per.statement=10` 来解决这个问题，完整的用法如下：

```
sqoop export \
    -Dsqoop.export.records.per.statement=10 \
    --connect jdbc:mysql://mysql.example.com/sqoop \
    --username sqoop ${user} \
    --password ${passwd} \
    --table ${tab_name} \
    --export-dir ${dir} \
    --batch
```

- 也可以选择增大 tidb 的单个事物语句数量限制，不过这个会导致内存上涨。

13.7.1.8 Dumpling 导出时引发上游数据库 OOM 或报错 “磁盘空间不足”

该问题可能有如下原因：

- 数据库主键分布不均匀，例如启用了SHARD_ROW_ID_BITS
- 上游数据库为 TiDB，导出表是分区表

在上述情况下，Dumpling 划分导出子范围时，会划分出过大的子范围，从而向上游发送结果过大的查询。请联系 AskTUG 社区专家获取实验版本的 Dumpling。

13.7.1.9 TiDB 有像 Oracle 那样的 Flashback Query 功能么，DDL 支持么？

有，也支持 DDL。详细参考TiDB 历史数据回溯。

13.7.2 在线数据同步

13.7.2.1 Syncer 架构

详细参考 解析 TiDB 在线数据同步工具 Syncer。

13.7.2.1.1 Syncer 使用文档

详细参考Syncer 使用文档。

13.7.2.1.2 如何配置监控 Syncer 运行情况？

下载 Syncer Json 导入到 Grafana，修改 Prometheus 配置文件，添加以下内容：

```
- job_name: 'syncer_ops' # 任务名字
  static_configs:
  - targets: ['10.10.1.1:10096'] # Syncer 监听地址与端口，通知 prometheus 拉取 Syncer 的数据。
```

重启 Prometheus 即可。

13.7.2.1.3 有没有现成的同步方案，可以将数据同步到 Hbase、Elasticsearh 等其他存储？

没有，目前依赖程序自行实现。

13.7.2.1.4 利用 Syncer 做数据同步的时候是否支持只同步部分表？

支持，具体参考 Syncer 使用手册Syncer 使用文档

13.7.2.1.5 频繁的执行 DDL 会影响 Syncer 同步速度吗？

频繁执行 DDL 对同步速度会有影响。对于 Sycner 来说，DDL 是串行执行的，当同步遇到了 DDL，就会以串行的方式执行，所以这种场景就会导致同步速度下降。

13.7.2.1.6 使用 Syncer gtid 的方式同步时，同步过程中会不断更新 syncer.meta 文件，如果 Syncer 所在的机器坏了，导致 syncer.meta 文件所在的目录丢失，该如何处理？

当前 Syncer 版本的没有进行高可用设计，Syncer 目前的配置信息 syncer.meta 直接存储在硬盘上，其存储方式类似于其他 MySQL 生态工具，比如 Mydumper。因此，要解决这个问题当前可以有两个方法：

- 把 syncer.meta 数据放到比较安全的磁盘上，例如磁盘做好 raid1；
- 可以根据 Syncer 定期上报到 Prometheus 的监控信息来还原出历史同步的位置信息，该方法的位置信息在大量同步数据时由于延迟会可能不准确。

13.7.2.1.7 Syncer 下游 TiDB 数据和 MySQL 数据不一致，DML 会退出么？

- 上游 MySQL 中存在数据，下游 TiDB 中该数据不存在，上游 MySQL 执行 UPDATE 或 DELETE（更新/删除）该条数据的操作时，Syncer 同步过程即不会报错退出也没有该条数据。
- 下游有主键索引或是唯一索引冲突时，执行 UPDATE 会退出，执行 INSERT 不会退出。

13.7.3 业务流量迁入

13.7.3.1 如何快速迁移业务流量？

我们建议通过 Syncer 工具搭建成多源 MySQL -> TiDB 实时同步环境，读写流量可以按照需求分阶段通过修改网络配置进行流量迁移，建议 DB 上层部署一个稳定的网络 LB（HAproxy、LVS、F5、DNS 等），这样直接修改网络配置就能实现无缝流量迁移。

13.7.3.2 TiDB 总读写流量有限制吗？

TiDB 读流量可以通过增加 TiDB server 进行扩展，总读容量无限制，写流量可以通过增加 TiKV 节点进行扩容，基本上写容量也没有限制。

13.7.3.3 Transaction too large 是什么原因，怎么解决？

TiDB 限制了单条 KV entry 不超过 6MB，可以修改配置文件中的`txn-entry-size-limit` 配置项进行调整，最大可以修改到 120MB。

分布式事务要做两阶段提交，而且底层还需要做 Raft 复制。如果一个事务非常大，提交过程会非常慢，事务写冲突概率会增加，而且事务失败后回滚会导致不必要的性能开销。所以我们设置了 key-value entry 的总大小默认不超过 100MB。如果业务需要使用大事务，可以修改配置文件中的 `txn-total-size-limit` 配置项进行调整，最大可以修改到 10G。实际的大小限制还受机器的物理内存影响。

在 Google 的 Cloud Spanner 上面，也有类似的限制。

13.7.3.4 如何批量导入？

导入数据的时候，可以分批插入，每批最好不要超过 1w 行。

13.7.3.5 TiDB 中删除数据后会立即释放空间吗？

DELETE，TRUNCATE 和 DROP 都不会立即释放空间。对于 TRUNCATE 和 DROP 操作，在达到 TiDB 的 GC (garbage collection) 时间后（默认 10 分钟），TiDB 的 GC 机制会删除数据并释放空间。对于 DELETE 操作 TiDB 的 GC 机制会删除数据，但不会释放空间，而是当后续数据写入 RocksDB 且进行 compact 时对空间重新利用。

13.7.3.6 Load 数据时可以对目标表执行 DDL 操作吗？

不可以，加载数据期间不能对目标表执行任何 DDL 操作，这会导致数据加载失败。

13.7.3.7 TiDB 是否支持 replace into 语法？

支持，但是 load data 不支持 replace into 语法。

13.7.3.8 数据删除后查询速度为何会变慢？

大量删除数据后，会有很多无用的 key 存在，影响查询效率。目前正在开发 Region Merge 功能，完善之后可以解决这个问题，具体看参考最佳实践中的删除数据部分。

13.7.3.9 数据删除最高效最快的方式？

在删除大量数据的时候，建议使用 `Delete * from t where xx limit 5000`（xx 建议在满足业务过滤逻辑下，尽量加上强过滤索引列或者直接使用主键选定范围，如 `id >= 5000*n+m and id <= 5000*(n+1)+m` 这样的方案，通过循环来删除，用 `Affected Rows == 0` 作为循环结束条件，这样避免遇到事务大小的限制。如果一次删除的数据量非常大，这种循环的方式会越来越慢，因为每次删除都是从前向后遍历，前面的删除之后，短时间内会残留不少删除标记（后续会被 GC 掉），影响后面的 Delete 语句。如果有可能，建议把 Where 条件细化。可以参考官网最佳实践。

13.7.3.10 TiDB 如何提高数据加载速度？

主要有两个方面：

- 目前已开发分布式导入工具TiDB Lightning，需要注意的是数据导入过程中为了性能考虑，不会执行完整的事务流程，所以没办法保证导入过程中正在导入的数据的 ACID 约束，只能保证整个导入过程结束以后导入数据的 ACID 约束。因此适用场景主要为新数据的导入（比如新的表或者新的索引），或者是全量的备份恢复（先 Truncate 原表再导入）。
- TiDB 的数据加载与磁盘以及整体集群状态相关，加载数据时应关注该主机的磁盘利用率，TiClient Error/Backoff/Thread CPU 等相关 metric，可以分析相应瓶颈。

13.7.3.11 对数据做删除操作之后，空间回收比较慢，如何处理？

可以设置并行 GC，加快对空间的回收速度。默认并发为 1，最大可调整为 tikv 实例数量的 50%。可使用 `update mysql.tidb set VARIABLE_VALUE="3" where VARIABLE_NAME="tikv_gc_concurrency";` 命令来调整。

14 版本发布历史

14.1 TiDB 版本发布历史

TiDB 历史版本发布声明如下：

14.1.1 4.0

- 4.0.16
- 4.0.15
- 4.0.14
- 4.0.13
- 4.0.12
- 4.0.11
- 4.0.10

- 4.0.9
- 4.0.8
- 4.0.7
- 4.0.6
- 4.0.5
- 4.0.4
- 4.0.3
- 4.0.2
- 4.0.1
- 4.0 GA
- 4.0.0-rc.2
- 4.0.0-rc.1
- 4.0.0-rc
- 4.0.0-beta.2
- 4.0.0-beta.1
- 4.0.0-beta

14.1.2 3.1

- 3.1.2
- 3.1.1
- 3.1.0 GA
- 3.1.0-rc
- 3.1.0-beta.2
- 3.1.0-beta.1
- 3.1.0-beta

14.1.3 3.0

- 3.0.20
- 3.0.19
- 3.0.18
- 3.0.17
- 3.0.16
- 3.0.15
- 3.0.14
- 3.0.13
- 3.0.12
- 3.0.11
- 3.0.10
- 3.0.9
- 3.0.8
- 3.0.7
- 3.0.6
- 3.0.5

- 3.0.4
- 3.0.3
- 3.0.2
- 3.0.1
- 3.0 GA
- 3.0.0-rc.3
- 3.0.0-rc.2
- 3.0.0-rc.1
- 3.0.0-beta.1
- 3.0.0-beta

14.1.4 2.1

- 2.1.19
- 2.1.18
- 2.1.17
- 2.1.16
- 2.1.15
- 2.1.14
- 2.1.13
- 2.1.12
- 2.1.11
- 2.1.10
- 2.1.9
- 2.1.8
- 2.1.7
- 2.1.6
- 2.1.5
- 2.1.4
- 2.1.3
- 2.1.2
- 2.1.1
- 2.1 GA
- 2.1 RC5
- 2.1 RC4
- 2.1 RC3
- 2.1 RC2
- 2.1 RC1
- 2.1 Beta

14.1.5 2.0

- 2.0.11
- 2.0.10
- 2.0.9

- 2.0.8
- 2.0.7
- 2.0.6
- 2.0.5
- 2.0.4
- 2.0.3
- 2.0.2
- 2.0.1
- 2.0
- 2.0 RC5
- 2.0 RC4
- 2.0 RC3
- 2.0 RC1
- 1.1 Beta
- 1.1 Alpha

14.1.6 1.0

- 1.0
- Pre-GA
- RC4
- RC3
- RC2
- RC1

14.2 v4.0

14.2.1 TiDB 4.0.16 Release Notes

发版日期：2021 年 12 月 17 日

TiDB **版本：**4.0.16

14.2.1.1 兼容性更改

- TiKV
  - 在 v4.0.16 以前，当把一个非法的 UTF-8 字符串转换为 Real 类型时会直接报错。自 v4.0.16 起，TiDB 会依照该字符串中的合法 UTF-8 前缀进行转换 #11466
- Tools
  - TiCDC
    * 将 Kafka Sink `max-message-bytes` 的默认值改为 1 MB，防止 TiCDC 发送过大消息到 Kafka 集群 #2962
    * 将 Kafka Sink `partition-num` 的默认值改为 3，使 TiCDC 更加平均地分发消息到各个 Kafka partition #3337

14.2.1.2 提升改进

- TiDB
  - 升级 Grafana 到 v7.5.11，规避老版本的安全漏洞
- TiKV
  - 当使用 Backup & Restore 恢复数据或使用 TiDB Lightning 的 Local-backend 导入数据时，采用 zstd 算法压缩 SST 文件，从而减小磁盘使用空间 #11469
- Tools
  - Backup & Restore (BR)
    - 增强恢复的鲁棒性 #27421
  - TiCDC
    - 为 EtcdWorker 添加 tick 频率限制，防止 PD 的 etcd 写入次数过于频繁影响 PD 服务 #3112
    - 优化 TiKV 重新加载时的速率限制控制，缓解 changefeed 初始化时 gPRC 的拥堵问题 #3110

14.2.1.3 Bug 修复

- TiDB
  - 修复在统计信息模块的估算代价中，当执行 range 转 points 时由于数值溢出而导致的查询崩溃 #23625
  - 修复当 ENUM 类型作为 IF 或 CASE WHEN 等控制函数的参数时，返回结果不正确的问题 #23114
  - 修复由于 tidb_enable_vectorized_expression 设置的值不同（on 或 off）导致 GREATEST 函数返回结果不一致的问题 #29434
  - 修复 Index Join 在使用前缀索引时某些情况下崩溃的问题 #24547
  - 修复在某些情况下 Planner 可能缓存无效 join 计划的问题 #28087
  - 修复当 sql_mode 为空时，TiDB 无法插入 null 到非 null 列的问题 #11648
  - 修正 GREATEST 和 LEAST 函数的返回值类型错误 #29019
  - 修复 grant 和 revoke 操作在授予和撤销全局权限时，报 privilege check fail 错误的问题 #29675
  - 修复当 CASE WHEN 函数和 ENUM 类型一起使用时的崩溃问题 #29357
  - 修复 microsecond 函数的向量化表达式版本结果不正确的问题 #29244
  - 修复 hour 函数在向量化表达式中执行结果错误的问题 #28643
  - 修复乐观事务冲突可能导致事务相互阻塞的问题 #11148
  - 修复 auto analyze 输出的日志信息不完整的问题 #29188
  - 修复当 SQL_MODE 为 'NO_ZERO_IN_DATE' 时，使用非法的默认时间不报错的问题 #26766
  - 修复 Grafana 上 Coprocessor Cache 监控面板不显示数据的问题。现在 Grafana 会显示 hits/miss/evict 的数据 #26338
  - 修复并发 truncate 同一个分区会导致 DDL 语句执行卡住的问题 #26229
  - 修复将 Decimal 转为 String 时长度信息错误的问题 #29417
  - 修复使用 NATUAL JOIN 连接多张表时，查询结果中多出一列的问题 #29481
  - 修复 IndexScan 使用前缀索引时，TopN 被错误下推至 indexPlan 的问题 #29711
  - 修复在 DOUBLE 类型的自增列上重试事务会导致数据错误的问题 #29892

- TiKV
  - 修复在极端情况下同时进行 Region Merge、ConfChange 和 Snapshot 时，TiKV 会出现 Panic 的问题 #11475
  - 修复 Decimal 除法计算的结果为 0 时符号为负的问题 #29586
  - 修复 TiKV 监控项中实例级别 gRPC 的平均延迟时间不准确的问题 #11299
  - 修复在缺失下游数据库时出现 TiCDC Panic 的问题 #11123
  - 修复因 channel 打满而导致的 Raft 断连的问题 #11047
  - 修复由于无法在 `Max/Min` 函数中正确识别 Int64 是否为有符号整数，导致 `Max/Min` 函数的计算结果不正确的问题 #10158
  - 修复因 Congest 错误而导致的 CDC 频繁增加 scan 重试的问题 #11082
- PD
  - 修复 TiKV 节点缩容后可能导致 Panic 的问题 #4344
  - 修复因 Region syncer 卡住而导致 leader 选举慢的问题 #3936
  - 允许 Evict Leader 调度器调度拥有不健康副本的 Region #4093
- TiFlash
  - 修复 TiFlash 在部分平台上由于缺失 nsl 库而无法启动的问题
- Tools
  - TiDB Binlog
    - 修复传输事务超过 1 GB 时 Drainer 会退出的问题 #28659
  - TiCDC
    - 修复监控 checkpoint lag 出现负值的问题 #3010
    - 修复在容器环境中 OOM 的问题 #1798
    - 修复在多个 TiKV 崩溃或强制重启时可能遇到复制中断的问题 #3288
    - 修复执行 DDL 后的内存泄漏的问题 #3174
    - 修复当发生 ErrGCTTLExceeded 错误时，changefeed 不快速失败的问题 #3111
    - 修复当上游 TiDB 实例意外退出时，TiCDC 同步任务推进可能停滞的问题 #3061
    - 修复当 TiKV 向同一 Region 发送重复请求时，TiCDC 进程 Panic 的问题 #2386
    - 修复 TiCDC 产生的 Kafka 消息体积不受 `max-message-size` 约束的问题 #2962
    - 修复 tikv_cdc_min_resolved_ts_no_change_for_1m 监控在没有 changefeed 的情况下持续更新的问题 #11017
    - 修复当写入 Kafka 消息发生错误时，TiCDC 同步任务推进可能停滞的问题 #2978
    - 修复当开启 `force-replicate` 时，可能某些没有有效索引的分区表被忽略的问题 #2834
    - 修复在创建新的 changefeed 时可能发生的内存泄漏问题 #2389
    - 修复可能因为 Sink 组件提前推进 resolved ts 导致数据不一致的问题 #3503
    - 修复当扫描存量数据耗时过长时，可能由于 TiKV 进行 GC 而导致存量数据扫描失败的问题 #2470
    - 修复 changefeed 更新命令无法识别全局命令行参数的问题 #2803

14.2.2 TiDB 4.0.15 Release Notes

发版日期：2021 年 9 月 27 日

TiDB 版本：4.0.15

14.2.2.1 兼容性更改

- TiDB
  - 修复在新会话中执行 `SHOW VARIABLES` 速度较慢的问题。该修复回退了 #21045 中的部分更改，可能会引起兼容性问题。#24326
  - 以下 Bug 修复涉及执行结果变化，可能引起兼容性变化：
    * 修复了 `greatest(datetime)union null` 返回空字符串的问题 #26532
    * 修复了 `having` 可能执行错误的问题 #26496
    * 修复了当 `between` 表达式两边的 collation 不一致会导致查询结果错误的问题 #27146
    * 修复了 `extract` 函数的参数是负数时查询结果错误的问题 #27236
    * 修复了当 `group_concat` 函数包含非 `bin` 的 collation 时查询结果错误的问题 #27429
    * 修复将 `Apply` 算子转为 `Join` 时漏掉列信息的问题 #27233
    * 修复将非法字符串转为 `DATE` 类型时的非预期行为 #26762
    * 修复开启 New Collation 时多列的 `count distinct` 返回结果错误的问题 #27091

14.2.2.2 功能增强

- TiKV
  - 支持动态修改 TiCDC 配置 #10645

14.2.2.3 提升改进

- TiDB
  - 基于直方图的 row count 来触发 auto-analyze #24237
- TiKV
  - 分离处理读写的 ready 状态以减少读延迟 #10475
  - TiKV Coprocessor 慢日志只考虑处理请求所花费的时间 #10841
  - 当 slogger 线程过载且队列已满时，删除日志而不是阻塞线程 #10841
  - 减少 Resolved TS 消息的大小以节省网络带宽 #2448
- PD
  - 提升了 PD 之间同步 Region 信息的性能 #3932
- Tools
  - Backup & Restore (BR)
    * 并发执行分裂和打散 Region 的操作，提升恢复速度 #1363
    * 遇到 PD 请求错误或 TiKV I/O 超时错误时重试 BR 任务 #27787
    * 恢复大量小表时减少空 Region 的产生，避免影响恢复后的集群运行 #1374
    * 创建表的时候自动执行 `rebase auto id` 操作，省去了单独的 `rebase auto id` DDL 操作，加快恢复速度 #1424

- Dumpling
  * 获取表信息前过滤掉不需要导出的数据库，提升 SHOW TABLE STATUS 的过滤效率 #337
  * 使用 SHOW FULL TABLES 来获取需要导出的表，因为 SHOW TABLE STATUS 在某些 MySQL 版本上运行存在问题 #322
  * 支持对 MySQL 兼容的特定数据库进行备份，这些数据库不支持 START TRANSACTION ... WITH CONSISTENT SNAPSHOT 和 SHOW CREATE TABLE 语法 #309
  * 完善 Dumpling 的警告日志，避免让用户误以为导出失败 #340
- TiDB Lightning
  * 支持导入数据到带有表达式索引或带有基于虚拟生成列的索引的表中 #1404
- TiCDC
  * TiCDC 总是内部从 TiKV 拉取 old value，提升 TiCDC 易用性 #2397
  * 当某张表的 Region 从某个 TiKV 节点全部迁移走时，减少 goroutine 资源的使用 #2284
  * 在高并发下减少 workerpool 中创建的 goroutine 数量 #2211
  * 异步执行 DDL 语句，不阻塞其他 changefeed #2295
  * 为所有 KV 客户端创建全局共享的 gRPC 连接池 #2531
  * 遇到无法恢复的 DML 错误立即退出，不进行重试 #1724
  * 优化 Unified Sorter 使用内存排序时的内存管理 #2553
  * 为执行 DDL 语句新增 Prometheus 监控指标 #2595 #2669
  * 禁止使用不同的 major 和 minor 版本启动 TiCDC 节点 #2601
  * 移除 `file sorter` 文件排序器 #2325
  * 清理被删 changefeed 的监控数据和已退出处理节点的监控数据 #2156
  * 优化 Region 初始化后的清锁算法 #2188

14.2.2.4 Bug 修复

- TiDB
  - 修复构建 range 时未正确给二进制字面值设置排序规则的问题 #23672
  - 修复当查询包含 GROUP BY 和 UNION 时报错 “index out of range” 的问题 #26553
  - 修复当 TiKV 有 tombstone store 时 TiDB 发送请求失败的问题 #23676 #24648
  - 移除文档中未记录的 `/debug/sub-optimal-plan` HTTP API #27264
  - 修复 `case when` 表达式的字符集不正确的问题 #26662
- TiKV
  - 修复数据恢复期间启用 TDE 时 BR 报告文件已存在错误的问题 #1179
  - 修复损坏的快照文件可能会造成磁盘空间无法回收的问题 #10813
  - 修复 TiKV 过于频繁删除陈旧 Region 的问题 #10680
  - 修复 TiKV 频繁重新连接 PD 客户端的问题 #9690
  - 从加密文件字典中检查陈旧的文件信息 #9115
- PD
  - 修复 PD 未能及时修复 Down Peer 副本的问题 #4077
  - 修复了 PD 在扩容 TiKV 时可能会 Panic 的问题 #3868

- TiFlash
  - 修复多盘部署时数据不一致的潜在问题
  - 修复当查询过滤条件包含诸如 CONSTANT、<、<=、>、>= 或 COLUMN 时出现错误结果的问题
  - 修复写入压力大时 metrics 中 store size 不准确的问题
  - 修复 TiFlash 多盘部署时无法恢复数据的潜在问题
  - 修复 TiFlash 长时间运行后无法回收 Delta 历史数据的潜在问题
- Tools
  - Backup & Restore (BR)
    * 修复了备份和恢复中平均速度计算不准确的问题 #1405
  - TiCDC
    * 修复集成测试中遇到由于 DDL Job 重复导致的 `ErrSchemaStorageTableMiss` 错误 #2422
    * 修复遇到 `ErrGCTTLExceeded` 错误时 changefeed 无法被删除的问题 #2391
    * 修复 `capture list` 命令输出中出现已过期 capture 的问题 #2388
    * 修复 TiCDC processor 出现死锁的问题 #2017
    * 修复重新调度一张表时多个处理器将数据写入同一张表引发的数据不一致的问题 #2230
    * 修复元数据管理出现 `EtcdWorker` 快照隔离被破坏的问题 #2557
    * 修复因为 DDL sink 错误导致 changefeed 不能被停止的问题 #2552
    * 修复一个 TiCDC Open Protocol 的问题：当一个事务中没有任何数据写入时候，TiCDC 产生一个空消息 #2612
    * 修复 TiCDC 在处理无符号 `TINYINT` 类型时崩溃的问题 #2648
    * 减小 gRPC 窗口来避免 Region 数量过多时触发内存溢出 #2202
    * 修复因 TiCDC capture 过多 Regions 出现的 OOM 问题 #2673
    * 修复将 `mysql.TypeString, mysql.TypeVarString, mysql.TypeVarchar` 等类型的数据编码为 JSON 时进程崩溃的问题 #2758
    * 修复在创建新的 changefeed 时可能发生的内存泄漏问题 #2389
    * 修复同步任务从一个表结构变更的 finish TS 开始时 DDL 处理失败的问题 #2603
    * 修复 owner 在执行 DDL 语句时崩溃可能导致 DDL 任务丢失的问题 #1260
    * 修复 `SinkManager` 中对 map 的不安全并发访问 #2298

14.2.3 TiDB 4.0.14 Release Notes

发版日期：2021 年 7 月 27 日

TiDB 版本：4.0.14

14.2.3.1 兼容性更改

- TiDB
  - 在 v4.0 中将 `tidb_multi_statement_mode` 的默认值从 `WARN` 更改为 `OFF`。建议使用客户端库的多语句功能，参考`tidb_multi_statement_mode` 文档。#25749
  - 将 Grafana 从 v6.1.16 升级到 v7.5.7 以解决两个安全漏洞，参考 Grafana 博文。

- 将系统变量 `tidb_stmt_summary_max_stmt_count` 的默认值从 200 修改为 3000 #25872

- TiKV

  - 将 `merge-check-tick-interval` 配置项的默认值从 10 修改为 2 以加快 Region 合并的速度 #9676

14.2.3.2 功能增强

- TiKV

  - 添加监控项 pending 用以监控 pending PD 心跳，帮助定位 PD 线程变慢的问题 #10008
  - 支持 virtual-host 风格的地址来让 BR 兼容类 S3 储存 #10242

- TiDB Dashboard

  - 新增 OIDC SSO 支持。通过设置兼容 OIDC 标准的 SSO 服务（例如 Okta、Auth0 等），用户可以在不输入 SQL 密码的情况下登录 TiDB Dashboard #960
  - 新增 Debug API 界面用于高级调试，通过该界面可以替代命令行方式来调用 TiDB 和 PD 的内部调试性 API #927

14.2.3.3 改进提升

- TiDB

  - 对于 `point get` 或 `batch point get` 算子，在唯一索引写入过程中，将悲观锁 LOCK 记录转化为 PUT 记录 #26223
  - 支持 MySQL 的系统变量 `init_connect` 及其相关功能 #26031
  - 支持稳定结果模式，使查询结果更稳定 #26003
  - 支持将函数 `json_unquote()` 下推到 TiKV #25721
  - 使 SQL 计划管理 (SPM) 不受字符集的影响 #23295

- TiKV

  - 关闭 TiKV 时，优先关闭 status server 来确保客户端可以正确检测关闭状态 #10504
  - 响应过期副本的消息，以确保过期副本被更快清除 #10400
  - 限制 TiCDC sink 的内存消耗 #10147
  - 当 Region 太大时，使用均匀分裂来加快分裂速度 #10275

- PD

  - 减少各调度器在同时工作时产生的冲突 #3854

- TiDB Dashboard

  - 更新 TiDB Dashboard 版本至 v2021.07.17.1 #3882
  - 支持将当前会话分享为只读的会话，禁止对分享的会话进行修改操作 #960

- Tools

- Backup & Restore (BR)
  - 恢复数据时合并小文件以提升恢复速度 #655
- Dumpling
  - 上游是 TiDB v3.x 集群时，使用 `_tidb_rowid` 来切分表以减少 TiDB 的内存使用 #306
- TiCDC
  - 优化 PD 节点缺失证书时的报错信息 #2184
  - 优化 sorter I/O 报错信息 #1976
  - 在 KV client 中新增 Region 增量扫描的并发度上限，减小 TiKV 的压力 #1926
  - 新增表内存使用量的监控项 #1884
  - 新增 TiCDC 服务端配置项 `capture-session-ttl` #2169

14.2.3.4 Bug 修复

- TiDB
  - 修复当连接一个带 `WHERE` 条件的子查询（值为 `false`）时 `SELECT` 的结果与 MySQL 不兼容的问题 #24865
  - 修复当参数是 `ENUM` 或 `SET` 类型时 `ifnull` 函数计算错误的问题 #24944
  - 修复某些情况下错误的聚合函数消除 #25202
  - 修复 Merge Join 运算中当列为 `SET` 类型时可能产生错误结果的问题 #25669
  - 修复 Cartesian Join 运算返回错误结果的问题 #25591
  - 修复 `SELECT ... FOR UPDATE` 语句进行连接运算且连接使用分区表时，可能产生异常退出情况的问题 #20028
  - 修复缓存的 `prepared` 计划被错误用于 `point get` 的问题 #24741
  - 修复 `LOAD DATA` 语句可以不正常导入非 utf8 数据的问题 #25979
  - 修复通过 HTTP API 访问统计信息时，可能导致内存泄露的问题 #24650
  - 修复执行 `ALTER USER` 语句时出现的安全性问题 #25225
  - 修复系统表 `TIKV_REGION_PEERS` 不能正确处理 `DOWN` 状态的问题 #24879
  - 修复解析 `DateTime` 时不截断非法字符串的问题 #22231
  - 修复 `select into outfile` 语句在列类型是 `YEAR` 时，可能无法产生结果的问题 #22159
  - 修复 `UNION` 子查询中出现 `NULL` 时可能导致查询结果出错的问题 #26532
  - 修复某些情况下投影算子在执行时可能造成 panic 的问题 #26534
- TiKV
  - 修复特定平台上的 duration 计算可能崩溃的问题 #related-issue
  - 修复将 `DOUBLE` 类型转换为 `DOUBLE` 的错误函数 #25200
  - 修复使用 async logger 时 panic 日志可能会丢失的问题 #8998
  - 修复开启加密后再次生成同样的 snapshot 会出现 panic 的问题 #9786 #10407
  - 修复 coprocessor 中 `json_unquote()` 函数错误的参数类型 #10176
  - 修复关机期间出现的可疑警告和来自 Raftstore 的非确定性响应 #10353 #10307
  - 修复备份线程泄漏的问题 #10287
  - 修复 Region split 过慢以及进行 Region merge 时，Region split 可能会损坏 metadata 的问题 #8456 #8783
  - 修复特定情况下 Region 心跳会导致 TiKV 不进行 split 的问题 #10111

  - 修复 TiKV 和 TiDB 间 CM Sketch 格式不一致导致统计信息错误问题 #25638
  - 修复 `apply wait duration` 指标的错误统计 #9893
  - 修复使用 Titan 时 `delete_files_in_range` 以后可能会产生 “Missing Blob” 报错的问题 #10232

- PD

  - 修复调度器在执行删除操作后可能再次出现的问题 #2572
  - 修复调度器在临时配置加载完毕前启动可能导致数据争用的问题 #3771
  - 修复打散 Region 操作可能导致 PD panic 的问题 #3761
  - 修复部分 Operator 未被正确设置优先级的问题 #3703
  - 修复从不存在的 Store 上删除 `evict-leader` 调度器时可能导致 PD panic 的问题 #3660
  - 修复了当集群内 Store 非常多时，PD 切换 Leader 慢的问题 #3697

- TiDB Dashboard

  - 修复实例性能分析界面无法获取全部 TiDB 实例信息的问题 #944
  - 修复 SQL 语句分析界面不显示执行 “计划数” 的问题 #939
  - 修复在升级集群后慢查询界面可能显示 “unknown field” 错误的问题 #902

- TiFlash

  - 修复编译 DAG 请求时出现进程崩溃的潜在问题
  - 修复读负载高的情况下进程崩溃的问题
  - 修复因列存中 split 失败导致 TiFlash 不断重启的问题
  - 修复无法删除 Delta 历史数据的潜在问题
  - 修复并发复制共享 Delta 索引导致结果错误的问题
  - 修复当数据缺失时 TiFlash 无法重启的问题
  - 修复旧的 dm 文件无法被自动清理的问题
  - 修复 `SUBSTRING` 函数包含特殊参数时引起进程崩溃的潜在问题
  - 修复将 `INT` 类型转换为 `TIME` 类型时产生错误结果的问题

- Tools

  - Backup & Restore (BR)

    * 修复不能恢复 `mysql` 库内的用户表的问题 #1142

  - TiDB Lightning

    * 修复 TiDB Lightning 解析 Parquet 文件中 `DECIMAL` 类型数据失败的问题 #1276
    * 修复 TiDB Lightning 导入大文件拆分时遇到的 `EOF` 报错问题 #1133
    * 修复 TiDB Lightning 导入含 `auto_increment` 的 `DOUBLE` 或 `FLOAT` 类型列的表时生成极大 base 值的问题 #1185
    * 修复在生成超过 4 GB 的 KV 数据时可能发生的 panic 问题 #1128

  - Dumpling

    * 使用 Dumpling 导出至 S3 存储时，不再要求 `s3:ListBucket` 权限覆盖整个 Bucket，只需要覆盖导出的前缀即可 #898

  - TiCDC

    * 修复分区表新增分区后的处理 #2205

    * 修复 TiCDC 无法读取 `/proc/meminfo` 导致崩溃的问题 #2023
    * 减少 TiCDC 运行时的内存使用 #2011 #1957
    * 修复 MySQL sink 遇到错误或暂停时，MySQL 连接会泄漏的问题 #1945
    * 修复当 start TS 小于 current TS 减去 GC TTL 时无法创建 TiCDC changefeed 的问题 #1839
    * 减少 sort heap 的内存 `malloc`，以降低 CPU 开销 #1853
    * 修复调度数据表时可能发生的同步终止问题 #1827

14.2.4 TiDB 4.0.13 Release Notes

发版日期：2021 年 5 月 28 日

TiDB 版本：4.0.13

14.2.4.1 新功能

- TiDB
    - 支持将列属性 AUTO_INCREMENT 变更为 AUTO_RANDOM #24608
    - 引入 `infoschema.client_errors_summary` 表，用以追踪返回给客户端的错误 #23267

14.2.4.2 提升改进

- TiDB
    - 当内存中的统计信息缓存是最新的时，避免后台作业频繁读取 `mysql.stats_histograms` 表造成高 CPU 使用率 #24352
- TiKV
    - 提高 `store used size` 计算过程的准确性 #9904
    - 在 EpochNotMatch 消息中返回更多的 Region 以降低 Region miss 的发生 #9731
    - 加快释放长期运行集群中堆积的内存 #10035
- PD
    - 优化 TSO 处理时间的统计指标，帮助用户判断 PD 侧的 TSO 处理时间是否过长 #3524
    - 更新 Dashboard 的版本至 v2021.03.12.1 #3469
- TiFlash
    - 自动清除过期历史数据以释放磁盘空间
- Tools
    - Backup & Restore (BR)
        * 支持备份恢复系统库 `mysql` 下的用户表 #1077
        * 更新 `checkVersion` 以检查集群版本和备份数据版本 #1090
        * 容忍在备份期间集群中出现少数 TiKV 节点宕机 #1062

- TiCDC
    * 为内部处理单元增加流程控制，避免出现内存溢出问题 #1751
    * 增加 Unified Sorter 清理陈旧临时文件的功能，禁止多个 cdc 服务共享 sort-dir 目录 #1741
    * 给 Failpoint 增加 HTTP 接口调用 #1732

14.2.4.3 Bug 修复

- TiDB
    - 修复带有子查询的 UPDATE 语句更新生成列时会 panic 的问题 #24658
    - 修复使用多列索引读取数据时返回结果重复的问题 #24634
    - 修复在 DIV 表达式中使用 BIT 类型常量作为除数造成查询结果错误的问题 #24266
    - 修复 NO_ZERO_IN_DATE SQL 模式对 DDL 语句中设置的列默认值无效的问题 #24185
    - 修复 BIT 类型列与整型列进行 UNION 并集运算时，查询结果出错的问题 #24026
    - 修复 BINARY 类型与 CHAR 类型比较时，错误地生成了 TableDual 执行计划的问题 #23917
    - 修复 insert ignore on duplicate 非预期的删除表记录的问题 #23825
    - 修复 Audit 插件导致 TiDB panic 的问题 #23819
    - 修复 HashJoin 算子未正确处理排序规则的问题 #23812
    - 修复悲观事务中，batch_point_get 处理异常值出错导致连接断开的问题 #23778
    - 修复 tidb_row_format_version 配置项的值被设置为 1，且 enable_new_collation 的值被设置为 true 时，数据索引不一致的问题 #23772
    - 修复整型列与字符串类型常量比较时，查询结果出错的问题 #23705
    - 修复 approx_percent 函数中传入 BIT 类型列时出错的问题 #23702
    - 修复执行 TiFlash 批量请求时，TiDB 误报 TiKV server timeout 的问题 #23700
    - 修复 IndexJoin 在前缀列索引上计算结果出错的问题 #23691
    - 修复由于 BINARY 类型列上排序规则处理不当导致查询结果出错的问题 #23598
    - 修复当 UPDATE 语句中存在含 HAVING 子句的连接查询时，执行崩溃的问题 #23575
    - 修复比较类型表达式中使用 NULL 常量导致 TiFlash 计算结果出错的问题 #23474
    - 修复 YEAR 类型列与字符类型常量比较结果出错的问题 #23335
    - 修复 session.group_concat_max_len 被设置得过小时，group_concat 执行崩溃的问题 #23257
    - 修复 TIME 类型列上使用 BETWEEN 表达式计算结果出错的问题 #23233
    - 修复 DELETE 语句中出现的权限检查问题 #23215
    - 修复往 DECIMAL 列中插入非法字符串时不报错的问题 #23196
    - 修复往 DECIMAL 类型列插入数据时解析出错的问题 #23152
    - 修复 USE_INDEX_MERGE hint 无法生效的问题 #22924
    - 修复使用 ENUM 或 SET 类型列作为 WHERE 过滤条件时，查询结果出错的问题 #22814
    - 修复 Clustered Index 与 New Collation 同时使用时，查询结果出错的问题 #21408
    - 修复 enable_new_collation 开启时，ANALYZE 出错的问题 #21299
    - 修复视图处理默认 ROLE 时，未正确处理相关 DEFINER 的问题 #24531
    - 修复取消 DDL Job 时卡住的问题 #24445
    - 修复 concat 函数错误处理排序规则的问题 #24300
    - 修复当 SELECT 域中包含 IN 子查询且子查询外侧表含有空值元组时，查询结果出错的问题 #24022
    - 修复逆序扫表时，TiFlash 被优化器错误选用的问题 #23974
    - 修复点查的返回结果中，列名与 MySQL 不一致的问题 #23970
    - 修复在数据库名含有大写字母的库中执行 show table status 结果为空的问题 #23958

  - 修复不同时拥有 INSERT 及 DELETE 权限的用户可以执行 REPLACE 操作的问题 #23938
  - 修复由于未正确处理排序规则导致的 concat/make_set/insert 表达式计算结果出错的问题 #23878
  - 修复在含有 RANGE 分区的表上查询时，查询崩溃的问题 #23689
  - 修复如下问题：在旧版本的集群中，若 tidb_enable_table_partition 被设置为 false，含有分区的表会被当作普通表处理。此时由旧版本升级至新版本时，在该表上执行 batch point get 查询会导致连接崩溃 #23682
  - 修复配置了 TiDB 监听 TCP 连接及 UNIX 套接字时，TCP 连接中远程主机未被正确验证的问题 #23513
  - 修复由于非默认的排序规则导致查询结果出错的问题 #22923
  - 修复 Grafana 的 Coprocessor Cache 面板不显示数据的问题 #22617
  - 修复优化器访问统计信息缓存时出错的问题 #22565

- TiKV

  - 修复因磁盘写满后 file_dict 写入不完全导致 TiKV 无法重启的问题 #9963
  - 限制 TiCDC 默认的扫描速度为 128MB/s #9983
  - 减少 TiCDC 进行初次扫描的内存使用量 #10133
  - 为 TiCDC 扫描的速度添加背压 (back pressure) 功能 #10142
  - 通过避免不必要的读取来获取 TiCDC 旧值以解决潜在的 OOM 问题 #10031
  - 修复了由于读取旧值而导致的 TiCDC OOM 问题 #10197
  - 为 S3 存储添加超时机制以避免 S3 客户端没有任何响应地挂起 #10132

- TiFlash

  - 修复未向 Prometheus 报告 delta-merge-tasks 数量的问题
  - 修复 Segment Split 期间发生进程崩溃的问题
  - 修复 Grafana 中 Region write Duration 面板位置错误的问题
  - 修复了存储引擎无法删除数据的潜在问题
  - 修复 TIME 类型转换为 INT 类型时产生错误结果的问题
  - 修复 bitwise 算子和 TiDB 行为不一致的问题
  - 修复字符串转换为 INT 时产生错误结果的问题
  - 修复连续快速写入可能导致 TiFlash 内存溢出的问题
  - 修复 Table GC 时会引发空指针的问题
  - 修复向已被删除的表写数据时 TiFlash 进程崩溃的问题
  - 修复当使用 BR 恢复数据时 TiFlash 进程可能崩溃的问题
  - 修复当使用通用 CI 排序规则时字符权重错误的问题
  - 修复被逻辑删除的表可能丢失数据的问题
  - 修复比较包含空字符的字符串时产生错误结果的问题
  - 修复输入列包含空常量时逻辑函数返回错误结果的问题
  - 修复逻辑函数仅接受数字类型输入的问题
  - 修复时间戳值为 1970-01-01 且时区偏移为负时计算结果不正确的问题
  - 修复 Decimal256 的哈希值计算结果不稳定的问题

- Tools

  - TiCDC

    * 修复当 Sorter 的输入通道卡住时，流控导致的死锁问题 #1779
    * 修复 TiCDC changefeed 断点卡住导致 TiKV GC safe point 不推进的问题 #1756

    * 回滚 explicit_defaults_for_timestamp 的改动，确保不用 SUPER 权限也可以同步数据到 MySQL #1749

  - TiDB Lightning

    * 修复在 autocommit 关闭的情况下，TiDB Lightning TiDB-backend 无法导入数据的问题

14.2.5 TiDB 4.0.12 Release Notes

发版日期：2021 年 4 月 2 日

TiDB 版本：4.0.12

14.2.5.1 新功能

- TiFlash

  - 新增工具用于检测当前 tiflash replica 的状态

14.2.5.2 改进提升

- TiDB

  - 优化 EXPLAIN 语句在 batch cop 模式下的输出信息 #23164
  - 在 EXPLAIN 语句的输出中，为无法下推到存储层的表达式增加告警信息 #23020
  - 调整 DDL 包中部分 Execute/ExecRestricted 的使用为安全 API (2) #22935
  - 调整 DDL 包中部分 Execute/ExecRestricted 的使用为安全 API (1) #22929
  - 添加 optimization-time 和 wait-TS-time 到慢日志中 #22918
  - 支持从 infoschema.partitions 表中查询 partition_id #22489
  - 添加 last_plan_from_binding 以帮助用户了解 SQL 执行计划是否与 binding 的 hint 相匹配 #21430
  - 支持在没有 pre-split 选项时也能执行 TRUNCATE 表操作 #22872
  - 为 str_to_date 表达式添加三种新的格式限定符 #22812
  - 在 metrics 监控中记录 PREPARE 执行失败的问题为 Failed Query OPM #22672
  - 当设置了 tidb_snapshot 时，不对 PREPARE 语句的执行报错 #22641

- TiKV

  - 消除短时间内大量重连接的现象 #9879
  - 对多 Tombstones 场景下的写操作和 Batch Get 进行优化 #9729
  - 将 leader-transfer-max-log-lag 的默认值改为 128，以增加切换 leader 的成功率 #9605

- PD

  - 只有当 pending-peer 或 down-peer 变更时才更新 Region Cache，减少心跳更新压力 #3471
  - 防止 split-cache 中的 Region 成为合并的目标 #3459

- TiFlash

  - 优化配置文件并删除无用项

  - 减小 TiFlash 二进制文件大小
  - 使用自适应的 GC 策略以减少内存使用

- Tools

  - TiCDC

    * 若任务的暂停同步时间超过 1 天，再次启动该任务时需要二次确认 #1497
    * 为 Old Value 功能添加监控面板 #1571

  - Backup & Restore (BR)

    * 记录 HTTP_PROXY 和 HTTPS_PROXY 环境变量 #827
    * 提升多表场景下的备份性能 #745
    * 在 service safe point 检查失败时报告错误 #826
    * 在 `backupmeta` 中记录集群版本和 BR 版本 #803
    * 遇到外部存储错误时，重试备份以便提高备份成功率 #851
    * 减少 BR 备份的内存使用 #886

  - TiDB Lightning

    * 运行 TiDB Lightning 前检查 TiDB 集群版本以防止未知错误 #787
    * 在遇到 cancel 错误时及时退出 #867
    * 添加 `tikv-importer.engine-mem-cache-size` 和 `tikv-importer.local-writer-mem-cache-size` 参数以便调整内存占用和性能之间的平衡 #866
    * Local-backend 并发运行 `batch split region` 以提高导入速度 #868
    * 从 S3 存储导入数据时，TiDB Lightning 不再要求 `s3:ListBucket` 权限 #919
    * 从 checkpoint 恢复时，TiDB Lightning 会继续使用之前的导入引擎 #924

14.2.5.3 Bug 修复

- TiDB

  - 修复当变量为十六进制字面量时，`get` 表达式出错的问题 #23372
  - 修复生成 Enum 和 Set 类型的快速执行计划时使用了错误 Collation 的问题 #23292
  - 修复 `nullif` 和 `is-null` 表达式一起使用时可能出现结果错误的问题 #23279
  - 修复自动搜集统计信息在规定时间窗口外被触发的问题 #23219
  - 修复 `point-get` 计划中 CAST 函数可能忽略错误的问题 #23211
  - 修复 CurrentDB 为空时 SPM 可能不生效的问题 #23209
  - 修复 IndexMerge 执行计划中可能出现错误过滤条件的问题 #23165
  - 修复 NULL 常量的返回类型中可能出现 NotNullFlag 的问题 #23135
  - 修复 Text 类型可能遗漏处理 Collation 的问题 #23092
  - 修复 Range 分区表处理 IN 表达式可能出错的问题 #23074
  - 修复将 TiKV 标记为 Tombstone 后，在相同地址和端口启动不同 StoreID 的新 TiKV 会持续返回 `StoreNotMatch` 的问题 #23071
  - INT 类型为 NULL 且和 YEAR 进行比较时不进行类型调整 #22844
  - 修复当表含有 `auto_random` 列 load data 时失去连接的问题 #22736
  - 修复取消 DDL 操作 panic 时可能阻塞其他 DDL 操作的问题 #23297
  - 修复进行 NULL 和 YEAR 比较时可能生成错误 key range 的问题 #23104

  - 修复创建视图成功但是使用时可能失败的问题 #23083

- TiKV

  - 修复 IN 表达式没有正确处理有符号和无符号整型数的问题 #9850
  - 修复 Ingest 操作不可重入问题 #9779
  - 修复 TiKV 在处理 JSON 向字符串转换时空格缺失的问题 #9666

- PD

  - 修复在 store 缺失 label 的情况下隔离级别错误的问题 #3474

- TiFlash

  - 修复当 binary 列的默认值前后包含 0 字节时查询结果错误的问题
  - 修复当数据库名称中包含特殊字符时无法同步数据的问题
  - 修复 IN 表达式中出现 Decimal 列时查询结果错误的问题
  - 修复 Grafana 中已打开文件数指标过高的问题
  - 修复当表达式中包含 Timestamp 类型时查询结果错误的问题
  - 修复处理 FROM_UNIXTIME 表达式时可能发生的无响应的问题
  - 修复字符串转换为整数结果不正确的问题
  - 修复 like 表达式可能返回错误结果的问题

- Tools

  - TiCDC

    * 修复 resolved ts 时间乱序的问题 #1464
    * 修复由于网络不稳定而导致的表调度出错引发的数据丢失问题 #1508
    * 修复终止 processor 时不能及时释放资源的问题 #1547
    * 修复因事务计数器未正确更新而导致下游数据库链接可能泄露的问题 #1524
    * 修复因 PD 抖动时多个 owner 共存可能导致数据表丢失的问题 #1540

  - Backup & Restore (BR)

    * 修复 WalkDir 在目标为 bucket name 的时候无返回值的问题 #733
    * 修复 status 端口无 TLS 的问题 #839

  - TiDB Lightning

    * 修复 TiKV Importer 可能忽略文件已存在的错误 #848
    * 修复 TiDB Lightning 可能使用错误的时间戳而读到错误数据的问题 #850
    * 修复 TiDB Lightning 非预期退出时可能造成 checkpoint 文件损坏的问题 #889
    * 修复由于忽略 cancel 错误而可能导致的数据错误的问题 #874

14.2.6 TiDB 4.0.11 Release Notes

发版日期：2021 年 2 月 26 日

TiDB 版本：4.0.11

14.2.6.1 新功能

- TiDB
  - 支持 `uft8_unicode_ci` 和 `utf8mb4_unicode_ci` 排序规则 #22558
- TiKV
  - 支持 `utf8mb4_unicode_ci` 排序规则 #9577
  - 支持 `cast_year_as_time` 排序规则 #9299
- TiFlash
  - 增加排队处理 Coprocessor 任务的线程池以降低内存溢出几率，并增加配置项 `cop_pool_size` 和 `batch_cop_pool_size`，默认值为 物理核数 `* 2`

14.2.6.2 改进提升

- TiDB
  - 重排由 `outer join` 简化的 `inner join` 顺序 #22402
  - Grafana 面板支持多集群 #22534
  - 为多语句问题提供替代解决方案 #22468
  - 将慢查询监控区分为 `internal` 和 `general` 两类 #22405
  - 为 `utf8_unicode_ci` 和 `utf8mb4_unicode_ci` 排序规则增加接口 #22099
- TiKV
  - 为 DBaaS 添加 server 信息的监控指标 #9591
  - Grafana dashboards 支持监控多个集群 #9572
  - 汇报 RocksDB 的监控指标到 TiDB #9316
  - 为 Coprocessor 任务记录暂停时间 #9277
  - 为 Load Base Split 添加 key 数量和大小的阀值 #9354
  - 在导入数据前检查文件是否存在 #9544
  - 改进 Fast Tune 面板 #9180
- PD
  - Grafana dashboards 支持监控多个集群 #3398
- TiFlash
  - 优化 `date_format` 函数的性能
  - 优化处理 ingest SST 时的内存开销
  - 优化 Batch Coprocessor 内部的重试逻辑以降低 Region error 的出现概率
- Tools
  - TiCDC

    * 在 capture 元信息中添加版本信息和在 changefeed 元信息中创建该 changefeed 的 CLI 版本 #1342

  - TiDB Lightning
    * 并行创建数据表以提升导入速度 #502
    * 跳过分裂小 Region 以提升导入速度 #524
    * 添加导入进度条并提升恢复进度的精确度 #506

14.2.6.3 Bug 修复

- TiDB
  - 修复异常的 unicode_ci 常数传递 #22614
  - 修复可能导致排序规则和 coercibility 错误的问题 #22602
  - 修复可能导致错误排序规则结果的问题 #22599
  - 修复不同排序规则的常数替换问题 #22582
  - 修复 like 函数使用排序规则时可能返回错误结果的问题 #22531
  - 修复 least 和 greatest 函数 duration 类型推导错误问题 #22580
  - 修复 like 函数处理 _ 宽字符后加 % 出错的问题 #22575
  - 修复比较函数 least 和 greatest 类型推导错误的问题 #22562
  - 修复使用 like 函数处理 Unicode 字符串错误的问题 #22529
  - 修复点查请求无法取得 @@tidb_snapshot 变量中快照的问题 #22527
  - 修复生成多个 join 相关 hint 可能 panic 的问题 #22518
  - 修复转换字符串为 BIT 类型不准确的问题 #22420
  - 修复插入 tidb_rowid 列时出现的 index out of range 报错问题 #22359
  - 修复缓存计划被错误地使用的问题 #22353
  - 修复 WEIGHT_STRING 函数处理过长字符串出现 panic 的问题 #22332
  - 禁止参数数量不合法时使用生成列 #22174
  - 在构造执行计划前正确地设置进程执行信息 #22148
  - 修复 IndexLookUp 执行统计不准的问题 #22136
  - 容器部署时为内存使用信息增加缓存 #22116
  - 修复解码执行计划错误的问题 #22022
  - 使用错误的窗口函数说明时提供报错 #21976
  - 使用 PREPARE 语句嵌套 EXECUTE、DEALLOCATE 或 PREPARE 时报错 #21972
  - 修复使用 INSERT IGNORE 到不存在的分区时不报错的问题 #21971
  - 统一 EXPLAIN 和 slow log 中的执行计划编码 #21964
  - 修复聚合算子下 join 出现未知列的问题 #21957
  - 修复 ceiling 函数中类型推导错误的问题 #21936
  - 修复 Double 列忽略精度的问题 #21916
  - 修复关联聚合在子查询中被计算的问题 #21877
  - 当 JSON 数据长度超过 65536 时提供报错 #21870
  - 修复 dyname 函数和 MySQL 不兼容的问题 #21850
  - 修复输入数据过长时 to_base64 函数返回 NULL 的问题 #21813
  - 修复在子查询中比较多个字段失败的问题 #21808
  - 修复 JSON 中比较浮点数的问题 #21785
  - 修复 JSON 类型比较的问题 #21718

  - 修复 `cast` 函数的 coercibility 值设置错误的问题 #21714
  - 修复使用 `IF` 函数时可能出现 panic 的问题 #21711
  - 修复 JSON 搜索返回 `NULL` 和 MySQL 不兼容的问题 #21700
  - 修复 `ORDER BY` 和 `HAVING` 子句检查 `only_full_group_by` 模式的问题 #21697
  - 修复 Day/Time 单位和 MySQL 不兼容的问题 #21676
  - 修复 `LEAD` 和 `LAG` 函数默认值类型问题 #21665
  - `LOAD DATA` 时执行检测以保证只能往基础表中导入数据 #21638
  - 修复 `addtime` 和 `subtime` 函数处理非法参数的问题 #21635
  - 将近似值的舍入规则更改为“舍入到最接近的偶数” #21628
  - 修复 `WEEK()` 在被明确读取前无法识别全局变量 `default_week_format` 的问题 #21623

- TiKV

  - 修复当设置 `PROST=1` 时构建 TiKV 失败的问题 #9604
  - 修复不匹配的内存诊断信息 #9589
  - 修复在恢复 RawKV 数据时部分 key range 的 end key 的包含性问题 #9583
  - 修复当 TiCDC 增量扫数据时读取一个被回滚的事务的某个 key 的旧值时 TiKV 可能会 panic 的问题 #9569
  - 修复使用不同配置的连接拉取同一个 Region 的变更时旧值配置不匹配的问题 #9565
  - 修复 TiVK 运行在网络接口缺少 MAC 地址的设备上会崩溃的问题（自 v4.0.9 引入）#9516
  - 修复 TiKV 在备份大 Region 时会内存溢出的问题 #9448
  - 修复 `region-split-check-diff` 无法自定义配置的问题 #9530
  - 修复系统时间回退时 TiKV 会 panic 的问题 #9542

- PD

  - 修复成员健康的监控显示不正确的问题 #3368
  - 禁止有副本的不正常 tombstone store 被清除 #3352
  - 修复 store limit 无法持久化的问题 #3403
  - 调整 `scatter range schedler` 的 limit 限制 #3401

- TiFlash

  - 修复 Decimal 类型的 `min/max` 计算结果错误的问题
  - 修复读取数据时有可能导致 crash 的问题
  - 修复 DDL 操作后写入的数据可能会在 compaction 后丢失的问题
  - 修复 Coprocessor 中错误解析 Decimal 常量的问题
  - 修复 Learner Read 过程中可能导致 crash 的问题
  - 修复 TiFlash 中除以 0 或 `NULL` 的行为与 TiDB 不一致的问题

- Tools

  - TiCDC

    * 修复 TiCDC 服务在同时发生 `ErrTaskStatusNotExists` 和 `capture` 会话关闭的情况下的非预期的退出的问题 #1240
    * 修复 `changefeed` 之间不同 Old Value 设置会互相影响的问题 #1347
    * 修复 TiCDC 服务在遇见错误的 `sort-engine` 参数时卡住的问题 #1309
    * 修复在非 Owner 节点上获取 debug 信息退出的问题 #1349

    * 修复 `ticdc_processor_num_of_tables` 和 `ticdc_processor_table_resolved_ts` 两个监控指标在增删数据表时没有被正确更新的问题 #1351
    * 修复 Processor 在添加同步数据表时退出而造成的潜在的数据丢失问题 #1363
    * 修复 Owner 在数据表迁移期间造成非正常状态的 TiCDC 服务退出的问题 #1352
    * 修复 TiCDC 服务在丢失 service GC safepoint 时没有及时退出的问题 #1367
    * 修复 KV client 可能跳过创建 event feed 的问题 #1336
    * 修复同步事务到下游时事务原子性被破坏的问题 #1375
  - Backup & Restore (BR)
    * 修复恢复备份后 TiKV 可能产生大 Region 的问题 #702
    * 修复在没有 Auto ID 的数据表上恢复 Auto ID 的问题 #720
  - TiDB Lightning
    * 修复使用 TiDB-backend 时可能触发 `column count mismatch` 的问题 #535
    * 修复 TiDB-backend 在导入数据源 column 个数和数据表 column 个数不匹配时非预期退出的问题 #528
    * 修复导入期间 TiKV 可能发生的非预期退出的问题 #554

14.2.7 TiDB 4.0.10 Release Notes

发版日期：2021 年 1 月 15 日

TiDB 版本：4.0.10

14.2.7.1 新功能

- PD
  - 添加了配置项 `enable-redact-log`，可以设置将日志中的用户数据脱敏 #3266
- TiFlash
  - 添加了配置项 `security.redact_info_log`，可以设置将日志中的用户数据脱敏

14.2.7.2 改进提升

- TiDB
  - 添加 `txn-entry-size-limit` 配置项，用于限制事务中单个 key-value 记录的大小 #21843
- PD
  - 优化了 `store-state-filter` 的监控，可以显示更加具体的原因 #3100
  - 更新 `go.etcd.io/bbolt` 依赖至 v1.3.5 #3331
- Tools
  - TiCDC

    * 为 maxwell 协议默认开启 old value 特性 #1144
    * 默认启用 unified sorter 特性 #1230
  - Dumpling
    * 支持检查未定义的参数，支持输出导出的进度 #228
  - TiDB Lightning
    * 支持重试读 S3 遇到的错误 #533

14.2.7.3 Bug 修复

- TiDB
  - 修复由于并发导致的 batch client 超时问题 #22336
  - 修复由于并发地自动捕获 SQL 绑定而导致的重复绑定问题 #22295
  - 当日志级别为 'debug' 时，让 SQL 语句绑定的自动捕获正确运行 #22293
  - 当 Region 合并正在发生时，正确地释放锁 #22267
  - 对 Datetime 类型的用户变量返回正确的值 #22143
  - 修复错误使用 Index Merge 访问方式的问题 #22124
  - 修复由于执行计划缓存导致 TiFlash 报 wrong precision 错误的问题 #21960
  - 修复由于 schema 变更导致的错误结果 #21596
  - 避免在 ALTER TABLE 中不必要地更改 column flag #21474
  - 让包含子查询块别名的 optimizer hint 生效 #21380
  - 为 IndexHashJoin 和 IndexMergeJoin 生成正确的 optimizer hint #21020
- TiKV
  - 修复了 peer 和 ready 之间的错误映射 #9409
  - 修复一些日志信息在 security.redact-info-log 设置为 true 时未脱敏的问题 #9314
- PD
  - 修复 ID 分配不是单调递增的问题 #3308 #3323
  - 修复 PD client 在某些情况下可能卡住的问题 #3285
- TiFlash
  - 修复了 TiFlash 解析老版本 TiDB 表结构失败导致 TiFlash 无法启动的问题
  - 修复了在 RedHat 系统中 TiFlash 会对 cpu_time 进行错误处理导致 TiFlash 无法启动的问题
  - 修复了将配置项 path_realtime_mode 设置为 true 时 TiFlash 无法启动的问题
  - 修复了当调用有三个参数的 substr 函数时，返回结果错误的问题
  - 修复了当 TiDB 对 Enum 枚举进行无损修改时，TiFlash 无法读取修改后的值的问题
- Tools
  - TiCDC
    * 修复 maxwell 协议的问题，包括 base64 数据输出和将 TSO 转换成 unix timestamp #1173
    * 修复过期的元数据可能引发新创建的 changefeed 异常的问题 #1184

    * 修复在关闭的 notifier 上创建 receiver 的问题#1199
    * 修复在 etcd 更新缓慢时导致内存访问量增长的问题 #1227
    * 修复 `max-batch-size` 不生效的问题 #1253
    * 修复清理过期任务信息的问题 #1280
    * 修复 MySQL sink 中由于没有调用 `rollback` 而导致回收 db conn 卡住的问题 #1285
  - Dumpling
    * 修改默认设置的 `tidb_mem_quota_query` 的行为以避免 TiDB 内存溢出 #233
  - Backup & Restore (BR)
    * 修复 BR v4.0.9 无法恢复 BR v4.0.8 保存在 GCS 上的备份 #688
    * 修复在恢复 GCS 上的备份时可能发生的 panic 问题 #673
    * 默认禁用备份统计信息以避免 BR 内存溢出 #693
  - TiDB Binlog
    * 修复在启用 `AMEND TRANSACTION` 特性时，Drainer 可能会使用错误 schema 来生成 SQL 语句的问题 #1033
  - TiDB Lightning
    * 修复未正确编码 Region key 而导致分裂 Region 失败问题 #531
    * 修复可能丢失 `CREATE TABLE` 失败的错误 #530
    * 修复使用 TiDB-backend 时遇到的 `column count mismatch` 问题 #535

14.2.8 TiDB 4.0.9 Release Notes

发版日期：2020 年 12 月 21 日

TiDB 版本：4.0.9

14.2.8.1 兼容性更改

- TiDB
  - 废弃配置文件中 `enable-streaming` 配置项 #21055
- TiKV
  - 减少开启加密时的 I/O 开销和锁冲突。该修改向下不兼容。如果需要降级至 v4.0.9 以下，需要将 `security.encryption.enable-file-dictionary-log` 配置为 `false`，并在降级前重启 #9195

14.2.8.2 新功能

- TiFlash
  - 支持将存储引擎的新数据分布在多个硬盘上，分摊 I/O 压力（实验特性）
- TiDB Dashboard

- SQL 语句分析功能的列表界面支持显示所有字段并排序 #749
- 集群拓扑页面支持缩放 #772
- SQL 语句分析及慢日志页面支持显示 SQL 语句的临时存储占用大小 #777
- SQL 语句分析及慢日志列表界面支持导出列表数据 #778
- 支持配置自定义 Prometheus 地址 #808
- 新增集群实例统计摘要页面 #815
- 慢日志详情页面新增更多时间字段 #810

14.2.8.3 优化提升

- TiDB
  - 在转换等值条件为其它条件时，通过使用启发式规则，避免生成 (index) merge join 以得到更好的执行计划 #21146
  - 区分用户变量的类型 #21107
  - 在配置文件中添加了 `performance.gogc` 配置项，用于设置 GOGC #20922
  - 提升 `Timestamp` 和 `Datetime` 类型的二进制输出结果与 MySQL 的兼容性 #21135
  - 优化用户使用 `LOCK IN SHARE MODE` SQL 语句时输出的报错信息 #21005
  - 优化在可剪切的表达式进行常量折叠时输出的错误信息，避免输出不必要的警告或错误信息 #21040
  - 优化 `LOAD DATA` 语句执行 `PREPARE` 时的报错信息 #21199
  - 修改整型列的类型时，忽略掉整型字段的零值填充的属性 #20986
  - 在 `EXPLAIN ANALYZE` 结果中输出 DML 语句执行器相关运行时的信息 #21066
  - 禁止在一条语句中对主键做出多次不同的修改 #21113
  - 添加连接空闲时间的监控项 #21301
  - 新增运行 `runtime/trace` 分析工具时系统自动临时开启慢日志的功能 #20578
- TiKV
  - 添加标记以跟踪 `split` 命令的来源 #8936
  - 支持动态修改 `pessimistic-txn.pipelined` 配置项 #9100
  - 减少运行 Backup & Restore 和 TiDB Lightning 时对系统的性能影响 #9098
  - 添加 Ingesting SST 报错的监控项 #9096
  - 阻止 Leader 在任意副本需要复制日志时进入休眠状态 #9093
  - 提高 Pipelined locking 的成功率 #9086
  - 调整配置项 `apply-max-batch-size` 和 `store-max-batch-size` 的默认值为 1024 #9020
  - 添加 `max-background-flushes` 配置 #8947
  - 默认关闭 RocksDB consistency check 以提高性能 #9029
  - 将 Region 大小的查询操作移出 `pd heartbeat worker` 以减轻其压力 #9185
- PD
  - TiKV store 转变为 `Tombstone` 状态时检查 TiKV 集群的版本号，防止用户降级和升级过程中的开启不兼容特性#3213
  - 禁止低版本的 TiKV 强制从 `Tombstone` 状态转为 Up #3206
- TiDB Dashboard

  - 对于 SQL 语句文本点击“展开”后支持保持展开状态 #775
  - 默认在新窗口打开 SQL 语句分析和慢日志详情 #816
  - 改进慢日志页面部分时间字段描述 #817
  - 改进错误信息提示，显示更完整的错误内容 #794

- TiFlash

  - 降低 Replica read 时的延迟
  - 优化 TiFlash 的错误信息
  - 优化在大数据量下，对缓存数据大小的限制
  - 添加正在处理的 Coprocessor 请求数量的 metric

- Tools

  - Backup & Restore (BR)

    * BR 不再接受存在歧义的 `--checksum false`（不会正确关闭 checksum）命令行参数，正确用法为 `--checksum=false` #588
    * 支持暂时性地调整 PD 的参数，在 BR 意外退出后，PD 能自动恢复回正常参数 #596
    * 支持恢复数据表的统计信息 #622
    * 系统自动重试 `read index not ready` 和 `proposal in merging mode` 两种错误 #626

  - TiCDC

    * 添加对 TiKV 开启 Hibernate Region 的告警规则 #1120
    * 优化 schema storage 的内存使用 #1127
    * 增加 Unified Sorter 功能，可以在数据量较大的情况下提升增量扫阶段的同步速度（实验特性）#1122
    * 支持在 TiCDC Open Protocol 中配置单条 Kafka 消息的最大大小和包含的最大行变更数量（仅在 Kafka sink 生效）#1079

  - Dumpling

    * 对导出失败部分的数据进行重试 #182
    * 支持同时设置 `-F` 和 `-r` 两个参数 #177
    * 默认不导出系统表 #194
    * 在设置 `--transactional-consistency` 参数时支持重建 MySQL 链接 #199
    * 支持使用 `-c,--compress` 参数指定 Dumpling 使用的压缩算法，空字符串代表不压缩 #202

  - TiDB Lightning

    * 默认不恢复系统表 #459
    * 支持为 auto-random 的主键设置默认值 #457
    * 完善 Local 模式下分裂 Region 的精度 #422
    * 支持给 `tikv-importer.region-split-size`、`mydumper.read-block-size`、`mydumper.batch-size` 和 `mydumper.max-region-size` 设置可读的参数（比如“2.5 GiB”）#471

  - TiDB Binlog

    * 在写下游出错时给 Drainer 设置非零退出码 #1012

14.2.8.4 Bug 修复

- TiDB
  - 修复了前缀索引和 OR 条件一起使用时结果不正确的问题 #21287
  - 修复了开启自动重试后可能出现的一处 panic #21285
  - 修复了根据列类型检查分区表定义时出现的一处问题 #21273
  - 修复了分区表对于列的类型检查的一处问题。分区表达式的值的类型和分区列的类型必须一致 #21136
  - 修复了哈希分区表对于分区名唯一性检查的问题 #21257
  - 修复非 INT 类型的值，插入到哈希分区表后结果不正确的问题 #21238
  - 修复了在部分写入类的场景中，使用了 index join 会遇到非预期报错的问题 #21249
  - 修复了在 CASE WHEN 中 BigInt 无符列的值被错误地转换成有符类型的问题 #21236
  - 修复了 index hash join 和 index merge join 没有考虑 collation 的问题 #21219
  - 修复了分区表在建表和查询时，没有考虑 collation 的问题 #21181
  - 修复了慢日志记录的查询结果可能不全的问题 #21211
  - 修复了一处数据库名大小写处理不当，导致的 DELETE 未正确删除数据的问题 #21206
  - 修复了执行 DML 语句导致 schema 的内存被覆盖的问题 #21050
  - 修复了使用 join 时，无法查询到合并后的列的问题 #21021
  - 修复了一些 semi-join 的查询结果不正确的问题 #21019
  - 修复了表锁对于 UPDATE 语句不生效的问题 #21002
  - 修复创建递归的视图出现栈溢出的问题 #21001
  - 修复了 index merge join 在执行外连接的时候，结果不符合预期的问题 #20954
  - 修复了一处事务问题，该场景下应该返回结果未知，但是却返回了执行失败 #20925
  - 修复 explain for connection 无法显示最后一次执行计划的问题 #21315
  - 修复在 Read Committed 隔离级别下，Index Merge 结果不正确的问题 #21253
  - 修复了由于事务写冲突重试导致的 auto-ID 分配失败 #21079
  - 修复了 JSON 数据无法通过 load data 无法正确导入到 TiDB 的问题 #21074
  - 修复新增加 Enum 类型列的默认值问题 #20998
  - 对于日期类型的数学计算，保留原始的数据类型信息，修复 adddate 函数插入非法值的问题 #21176
  - 修复了部分场景错误地生成了 PointGet 的执行计划，导致执行结果不正确 #21244
  - 在 ADD_DATE 函数中忽略夏令时的转换，以和 MySQL 兼容 #20888
  - 修复了插入尾部带有超出 varchar 和 char 长度限制的空白字符的字符串时报错的 bug #21282
  - 修复了对比 int 和 year 类型时没有将 [1, 69] 的整数转换为 [2001, 2060] 以及没有将 [70, 99] 的整数转换为 [1970, 1999] 的兼容性 bug #21283
  - 修复了 sum() 函数计算 Double 类型字段的结果溢出导致 panic 的问题 #21272
  - 修复了 DELETE 语句未能给 unique key 加悲观锁的问题 #20705
  - 修复了快照读能够命中 lock cache，返回错误结果的问题 #21539
  - 修复了在同一个事务中读取大量数据时可能发生的内存泄漏问题 #21129
  - 修复了在子查询中省略表别名时的语法解析错误问题 #20367
  - 修复了查询中 IN 函数的参数为 time 类型时可能返回错误结果的问题 #21290
- TiKV
  - 修复当列个数大于 255 时，下推返回错误结果集的问题 #9131
  - 修复网络隔离时 Region Merge 可能会导致数据丢失的问题 #9108

  - 修复使用 `latin1` 字符集时，`ANALYZE` 语句会导致 panic 的问题 #9082
  - 修复类型转换中将数字转成时间会得到错误结果的问题 #9031
  - 修复当开启加密时无法使用 TiDB Lightning 导入数据的问题 #8995
  - 修复使用 `0.0.0.0` 时 `advertise-status-addr` 异常的问题 #9036
  - 修复当事务删除 key 时却报 key 已存在的问题 #8930
  - 修复 RocksDB cache 映射错误导致的数据错误问题 #9029
  - 修复当 Leader 切换时 Follower Read 可能返回旧数据的问题 #9240
  - 修复悲观锁下可能读到旧值的问题 #9282
  - 修复 transfer leader 后 replica read 可能会读到旧值的问题 #9240
  - 修复 TiKV 在 profiling 结束后再收到 `SIGPROF` 会 panic 的问题 #9229

- PD

  - 修复在特殊情况下 Placement Rule 指定的 leader 绑定不生效的问题 #3208
  - 修复 `trace-region-flow` 在配置更新时被置为 `false` 的问题 #3120
  - 修复特殊情况下 safepoint 有无限 TTL 的问题 #3143

- TiDB Dashboard

  - 修复部分时间显示混杂中英文的问题 #755
  - 修复部分不兼容的浏览器中没有提示不兼容的问题 #776
  - 修复部分情况下事务时间戳显示不正确的问题 #793
  - 修复部分 SQL 文本格式化后成为无效 SQL 语句的问题 #805

- TiFlash

  - 修复 `INFORMATION_SCHEMA.CLUSTER_HARDWARE` 中可能包含未被使用的硬盘信息的问题
  - 修复 Delta cache 内存占用量估算偏少的问题
  - 修复由线程统计信息引起的内存泄露问题

- Tools

  - Backup & Restore (BR)

    * 修复因 S3 secret access keys 中存在特殊字符而导致失败的问题 #617

  - TiCDC

    * 修复某些异常情况下存在多个 Owner 的问题 #1104
    * 修复在 TiKV 节点意外退出或重启恢复情况下 TiCDC 不能正常同步的问题，该 bug 在 v4.0.8 引入 #1198
    * 修复在表初始化过程中会向 etcd 中重复写入元数据的问题 #1191
    * 修复 schema storage 缓存 TiDB 表信息的过程中因更新表信息延迟或过早 GC 导致同步中断的问题 #1114
    * 修复 schema storage 在 DDL 频繁的情况下会消耗过多内存的问题 #1127
    * 修复在同步任务暂停或取消之后会产生 goroutine 泄露的问题 #1075
    * 增加 Kafka producer 最大重试时间到 600s，避免在下游 Kafka 服务或网络抖动情况下同步中断 #1118
    * 修复 Kafka 消息所包含行变更数量不能正常生效的问题 #1112

        * 修复当 TiCDC 与 PD 间网络出现抖动，并且同时操作 TiCDC changefeed 暂停和恢复，可能会出现部分表数据没有被同步的问题 #1213
        * 修复 TiCDC 与 PD 网络不稳定情况下 TiCDC 可能出现进程非预期退出的问题 #1218
        * 在 TiCDC 内部使用全局 PD client，以及修复 PD client 被错误关闭导致同步阻塞的问题 #1217
        * 修复 TiCDC owner 节点可能在 etcd watch client 里消耗过多内存的问题 #1224
    - Dumpling
        * 修复在某些情况下 MySQL 链接关闭导致 Dumpling 卡住的问题 #190
    - TiDB Lightning
        * 修复使用错误信息编码 key 的问题 #437
        * 修复 GC life time TTL 不生效的问题 #448
        * 修复手动关闭时可能出现的 panic 问题 #484

14.2.9 TiDB 4.0.8 Release Notes

发版日期：2020 年 10 月 30 日

TiDB 版本：4.0.8

14.2.9.1 新功能

- TiDB
    - 支持聚合函数 APPROX_PERCENTILE #20197
- TiFlash
    - 支持 CAST 函数下推
- Tools
    - TiCDC
        * 支持快照级别一致性复制 #932

14.2.9.2 优化提升

- TiDB
    - 在挑选索引组合计算表达式选择率的贪心算法里优先使用选择率低的索引 #20154
    - 在 Coprocessor 运行状态中记录更多的 RPC 信息 #19264
    - 优化读取慢日志的效率，以提升慢查询性能 #20556
    - 在挑选执行计划时，优化器会在 Plan Binding 阶段等待超时的执行计划以记录更多的 Debug 信息 #20530
    - 在慢查询和慢日志中增加语句的重试时间 #20495 #20494
    - 增加系统表 table_storage_stats #20431
    - 为 INSERT/UPDATE/REPLACE 语句记录 RPC 相关的运行时信息 #20430

  - 在 EXPLAIN FOR CONNECTION 语句的结果中新增算子信息 #20384
  - 在 TiDB 日志中将客户端的连接建立/断开日志级别调整为 DEBUG #20321
  - 增加 Coprocessor Cache 的监控信息 #20293
  - 在运行时信息中记录更多的悲观锁相关参数 #20199
  - 在运行时信息和 Trace 功能中增加两个新的耗时信息 #20187
  - 在慢日志中添加事务提交的运行时信息 #20185
  - 关闭 Index Merge Join #20599
  - 为临时字符串常量增加 ISO 8601 和时区支持 #20670

- TiKV

  - 添加 Fast-Tune 监控页辅助性能诊断 #8804
  - 添加 security.redact-info-log 配置，用于从日志中删除用户数据 #8746
  - 修改 error code 的 metafile 格式 #8877
  - 开启动态修改 pessimistic-txn.pipelined 配置 #8853
  - 默认开启 memory profile 功能 #8801

- PD

  - 生成 error 的 metafile #3090
  - 为 operator 日志添加更多有用信息 #3009

- TiFlash

  - 添加关于 Raft log 的监控
  - 添加关于 cop 任务内存使用的监控
  - 在存在删除数据的情况下使 min/max 索引更加准确
  - 提高小批量数据下的查询性能
  - 添加 error.toml 文件以支持标准错误码

- Tools

  - Backup & Restore (BR)

    * 通过将 split 和 ingest 流水线来加速恢复 #427
    * 支持手动恢复 PD 的调度器 #530
    * 将移除 PD 调度器接口改为暂停调度器 #551

  - TiCDC

    * 在 MySQL sink 中定期输出统计信息 #1023

  - Dumpling

    * 支持直接导出数据到 S3 存储上 #155
    * 支持导出 View 视图 #158
    * 支持导出只包含生成列的数据表 #166

  - TiDB Lightning

    * 支持多字节的 CSV delimiter 和 separator #406
    * 通过禁止一些 PD 调度器加速导入 #408
    * 在 v4.0 集群上使用 GC-TTL 接口来防止 checksum 阶段的 GC 报错 #396

14.2.9.3 Bug 修复

- TiDB
  - 修复使用分区表时，可能遇到非预期 Panic 的问题 #20565
  - 修复外连接时，若外表有过滤条件，Index Merge Join 结果有时不正确的问题 #20427
  - 修复 BIT 类型进行转换时，由于类型长度溢出而错误地返回 NULL 的问题 #20363
  - 修复 ALTER TABLE ... 语法改变 BIT 类型的默认值，可能导致默认值错误的问题 #20340
  - 修复 BIT 类型转换为 INT64 时可能导致长度溢出错误的问题 #20312
  - 修复混合类型的列在进行条件传播优化时，可能导致结果错误的问题 #20297
  - 修复 Plan Cache 在存储过期执行计划时，可能 Panic 的问题 #20246
  - 修复 FROM_UNIXTIME 和 UNION ALL 一起使用时，返回结果会被错误地截断的问题 #20240
  - 修复 Enum 类型在转换为 Float 类型时可能导致错误结果的问题 #20235
  - 修复 RegionStore 在某些条件下会 Panic 的问题 #20210
  - 修复 BatchPointGet 请求对无符号整数的最大值进行排序时，结果错误的问题 #20205
  - 修复 Enum/Set 类型在混合 Collation 的类型判定时，结果 Collation 可能与 MySQL 不兼容的问题 #20364
  - 修复将其他类型的 0 转换为 YEAR 类型时，结果与 MySQL 不兼容的问题 #20292
  - 修复 KV Duration 监控指标中包含 store0 时，上报结果不正确的问题 #20260
  - 修复写入 Float 类型数据时，由于长度溢出提示 out of range 错误后仍然被错误地写入问题 #20252
  - 修复 NOT NULL 属性生成列允许在某些情况下写入 NULL 值的问题 #20216
  - 修复 YEAR 类型数据写入超过允许范围时，错误提示不准确的问题 #20170
  - 修复某些情况下悲观事务重试时，会报错 invalid auto-id 的问题 #20134
  - 修复 ALTER TABLE 更改 Enum/Set 类型时，未进行重复性约束检查的问题 #20046
  - 修复一些算子在并发执行时，记录的 Coprocessor Task 运行时信息错误的问题 #19947
  - 修复只读系统变量无法被作为 Session 级变量显式 SELECT 的问题 #19944
  - 修复重复 ORDER BY 条件有时会导致执行计划选择不是最优的问题 #20333
  - 修复生成 Metric Profile 时，由于字体超过允许的最大值导致失败的问题 #20637
- TiKV
  - 修复加密功能中锁冲突导致 pd-worker 处理心跳慢的问题 #8869
  - 修复错误生成 memory profile 的问题 #8790
  - 修复备份时指定 GCS 储存类别 (storage class) 报错的问题 #8763
  - 修复了重启或者新 Split 的 Learner 节点找不到 Leader 的问题 #8864
- PD
  - 修复了 TiDB Dashboard 在某些场景下引起 PD panic 的错误 #3096
  - 修复了某个 PD store 下线超过十分钟后可能引起 PD panic 的错误 #3069
- TiFlash
  - 修复了日志信息中时间戳错误的问题
  - 修复了使用多盘部署时错误的容量导致创建 TiFlash 副本失败的问题
  - 修复了 TiFlash 重启后可能提示数据文件损坏的问题
  - 修复了 TiFlash 崩溃后磁盘上可能残留损坏文件的问题

  - 修复了在写流量较小情况下，由于 Raft Learner 协议中的状态不能及时更新而导致 `wait index duration` 变长，造成查询慢的问题
  - 修复了在重放过期 Raft 日志时，proxy 会向 key-value 引擎写入大量 Region state 信息的问题

- Tools

  - Backup & Restore (BR)

    * 修复 Restore 期间可能发生的 `send on closed channel` panic 问题 #559

  - TiCDC

    * 修复 owner 因更新 GC safepoint 失败而非预期退出的问题 #979
    * 修复非预期的任务信息更新 #1017
    * 修复非预期的空 Maxwell 消息 #978

  - TiDB Lightning

    * 修复列信息错误的问题 #420
    * 修复 Local 模式下获取 Region 信息出现死循环的问题 #418

14.2.10 TiDB 4.0.7 Release Notes

发版日期：2020 年 9 月 29 日

TiDB 版本：4.0.7

14.2.10.1 新功能

- PD

  - PD 客户端中添加 `GetAllMembers` 函数，用于获取 PD 成员信息 #2980

- TiDB Dashboard

  - 支持生成统计指标关系图 #760

14.2.10.2 优化提升

- TiDB

  - 为 `join` 算子添加更多执行信息 #20093
  - 为 `EXPLAIN ANALYZE` 语句添加协处理器缓存命中率信息 #19972
  - 支持将 `ROUND` 函数下推至 TiFlash #19967
  - 在 `ANALYZE` 过程中为 `CMSketch` 添加默认值 #19927
  - 重新更改某些日志错误信息以使其脱敏 #20004
  - 支持接受来自 MySQL 8.0 客户端的连接 #19959

- TiKV

  - 支持日志输出为 JSON 格式 #8382

- PD
  - operator 统计计数器增加的时间点，从创建时改为执行完毕时 #2983
  - 将 make-up-replica operator 调整为高优先级 #2977
- TiFlash
  - 完善数据读取过程中遇到 Region meta 变更的错误处理
- Tools
  - TiCDC
    * 提升开启 Old Value 后 MySQL Sink 的同步效率 #955
  - Backup & Restore (BR)
    * 增加备份时链接中断重试 #508
  - TiDB Lightning
    * 增加动态设置 log 级别的 HTTP 接口 #393

14.2.10.3 Bug 修复

- TiDB
  - 修复向量化函数 and/or/COALESCE 因为提前计算导致的问题 #20092
  - 修复不同存储类型导致相同的执行计划摘要的问题 #20076
  - 修复错误函数 != any() 的错误表现 #20062
  - 修复当慢日志文件不存在时输出慢日志报错的问题 #20051
  - 修复当上下文取消后 Region 请求不断重试的问题 #20031
  - 修复查询 cluster_slow_query 表的时间类型在 streaming 的请求下报错的问题 #19943
  - 修复 DML 语句使用 case when 函数时可能导致 schema 改变的问题 #20095
  - 修复 slow log 中 prev_stmt 的信息未脱敏的问题 #20048
  - 修复当 tidb-server 不正常退出时没有释放表锁的问题 #20020
  - 修复当插入 ENUM 和 SET 类型的字段产生不正确的错误信息的问题 #19950
  - 修复 IsTrue 函数在某些情况下的错误表现 #19903
  - 修复在 PD 扩容或缩容情况下 CLUSTER_INFO 系统表可能不正常运行的问题 #20026
  - 修复在控制表达式中某些情况下产生不必要的错误或报警信息 #19910
  - 改变更新统计信息的方式以避免造成 OOM 的情况 #20013
- TiKV
  - 修复 TLS 握手失败后会导致 Status API 不可用的问题 #8649
  - 修复一些平台上可能存在潜在未定义行为的问题 #7782
  - 修复执行 UnsafeDestroyRange 操作时生成快照可能导致 Panic 的问题 #8681
- PD
  - 修复当 balance-region 开启时，如果存在 Region 没有 Leader，可能会导致 PD panic 的问题 #2994

  - 修复 Region 合并后 Region 大小和 Region key 数量的统计偏差 #2985
  - 修复热点统计不正确的问题 #2991
  - 修复 `redirectSchedulerDelete` 中未进行 `nil` 判断的问题 #2974

- TiFlash

  - 修正 right outer join 结果错误的问题

- Tools

  - Backup & Restore (BR)

    * 修复了在恢复数据后导致 TiDB 配置变更的错误 #509

  - Dumpling

    * 修复了在某些变量为空的情况下 metadata 解析失败的问题 #150

14.2.11 TiDB 4.0.6 Release Notes

发版日期：2020 年 9 月 15 日

TiDB 版本：4.0.6

14.2.11.1 新功能

- TiFlash

  - 在 TiFlash 中支持在广播 Join 中使用外连接

- TiDB Dashboard

  - 添加 Query 编辑和执行页面 (实验性功能) #713
  - 添加 Store 地理拓扑显示页面 #719
  - 添加集群配置调整页面 (实验性功能) #733
  - 支持共享当前 session #741
  - 支持显示 SQL 语句分析中执行计划的数量 #746

- Tools

  - TiCDC（自 v4.0.6 起，TiCDC 成为正式功能，可用于生产环境）

    * 支持输出 `maxwell` 格式的数据 #869

14.2.11.2 优化提升

- TiDB

  - 使用标准错误替换 TiDB 中的错误码和错误信息 #19888
  - 提升分区表的写性能 #19649
  - 在 Cop Runtime 统计信息中记录更多的 RPC Runtime 信息 #19264

- 禁止在 metrics_schema 和 performance_schema 中创建表 #19792
- 支持调整 Union 执行算子的并发度 #19886
- 支持在广播 Join 中使用外连接 #19664
- 添加对 process list 的 digest #19829
- 对于自动提交语句的重试转换到悲观锁模式 #19796
- 在 Str_to_date 函数中支持 %r 和 %T 的数据格式 #19693
- 使 SELECT INTO OUTFILE 需要文件权限 #19577
- 支持 stddev_pop 函数 #19541
- 添加 TiDB-Runtime 面板 #19396
- 提升 ALTER TABLE ALGORITHMS 的兼容性 #19364
- 在慢日志的计划字段中加入编码好的 INSERT/DELETED/UPDATE 计划 #19269

- TiKV

  - 优化 DropTable/TruncateTable 时导致的性能下降 #8627
  - 支持生成标准错误码的 meta 文件 #8619
  - scan detail 中增加 tombstone 个数的 metrics #8618
  - 添加 rocksdb perf context 到 Grafana 默认面板 #8467

- PD

  - 升级 Dashboard 到 v2020.09.08.1 #2928
  - 添加更多关于 store 和 Region 心跳的 metrics #2891
  - 回滚空间不足的阈值策略 #2875
  - 支持标准错误码
    * #2918 #2911 #2913 #2915 #2912
    * #2907 #2906 #2903 #2806 #2900 #2902

- TiFlash

  - 在 Grafana 中添加关于数据同步 (apply Region snapshots 和 ingest SST files) 的监控面板
  - 在 Grafana 中添加关于 write stall 的监控面板
  - 添加 dt_segment_force_merge_delta_rows 及 dt_segment_force_merge_delta_deletes 用于调整阈值以避免 write stall 发生
  - 支持在 TiFlash-Proxy 中把 raftstore.snap-handle-pool-size 设为 0 以禁用多线程同步 Region snapshot，可降低同步数据时内存消耗
  - 在 https_port 及 metrics_port 上支持 CN 检查

- Tools

  - TiCDC
    * 在初始化阶段跳过 resolved lock #910
    * 减少写 PD 的频率 #937
  - Backup & Restore (BR)
    * 在 Summary 中添加真实消耗的时间 #486
  - Dumpling

    * 支持输出带有列名的 INSERT 语句 #135
    * 将 --filesize 和 --statement-size 参数与 mydumper 保持统一 #142
  - TiDB Lightning
    * Split 的 Region 大小更加精确 #369
  - TiDB Binlog
    * 支持以 go time 的格式设置 GC 时间 #996

14.2.11.3 Bug 修复

- TiDB
  - 修复了在 Metric Profile 中 tikv_cop_wait time 的一个问题 #19881
  - 修复了 SHOW GRANTS 显示错误结果的问题 #19834
  - 修复了使用 != ALL (subq) 查询结果不正确的问题 #19831
  - 修复了转换 ENUM 和 SET 类型的一个问题 #19778
  - 增加了 SHOW STATS_META、SHOW STATS_BUCKET 的一个权限检查 #19760
  - 修复了由 builtinGreatestStringSig 和 builtinLeastStringSig 引起的列长度不匹配问题 #19758
  - 如果向量化计算抛出多余的 errors 或者 warnings，回退向量化执行到标量执行 #19749
  - 修复了在相关列类型是 Bit 时 Apply 算子出现错误的问题 #19692
  - 修复了在 MySQL 8.0 客户端中查询 processlist 和 cluster_log 时出现的问题 #19690
  - 修复了相同类型的 plan 具有不同类型的 plan digest 的错误 #19684
  - 禁止从 Decimal to Int 变更列类型 #19682
  - 修复了 SELECT ... INTO OUTFILE 返回运行时错误的问题 #19672
  - 修复了 builtinRealIsFalseSig 的不正确的实现 #19670
  - 修复了分区表达式检查漏掉括号表达式的问题 #19614
  - 修复了当在 HashJoin 上具有 Apply 算子时的查询错误 #19611
  - 修复了向量化将 Real cast 成 Time 类型时的错误的结果 #19594
  - 修复了 SHOW GRANTS 可以显示不存在用户的 grants 信息的错误 #19588
  - 修复了当在 IndexLookupJoin 上具有 Apply 算子时的查询错误 #19566
  - 修复了当在分区表上将 Apply 转化成 HashJoin 时的错误结果 #19546
  - 修复了当在 Apply 的 inner 端具有 IndexLookUp 算子时的错误结果 #19508
  - 修复了使用视图时非预期的 panic #19491
  - 修复了 anti-semi-join 查询时的不正确结果 #19477
  - 修复了删除统计信息时未删除 topN 的统计信息的错误 #19465
  - 修复了因错误使用 batch point get 时产生的错误结果 #19460
  - 修复了在带有虚拟生成列的 IndexLookupJoin 上无法找到列的错误 #19439
  - 修复了在 SELECT 和 UPDATE 查询上的不同计划比较 datum 的错误 #19403
  - 修复了 TiFlash 在 Region cache 上产生的 work index 数据争用 #19362
  - 修复了 logarithm 函数不返回 warning 的错误 #19291
  - 修复了当使用 TiDB 落盘时产生的非预期错误 #19272
  - 支持在 Index Join 的 inner 端使用单个分区表 #19197
  - 修复了对 decimal 产生的错误的 hash 键值 #19188
  - 修复了当 table EndKey 和 Region EndKey 相同时 TiDB 会产生 no regions 的错误 #19895
  - 修复了 alter partition 的非预期成功 #19891

  - 修复了在下推表达式上，默认最大允许的包长的错误 #19876
  - 修复了在 ENUM/SET 列上 Max/Min 函数的错误行为 #19869
  - 修复了当部分 TiFlash 节点下线之后，`tiflash_segments` 和 `tiflash_tables` 系统表读取失败的问题 #19748
  - 修复了 `Count()` 聚集函数的错误结果 #19628
  - 修复了 TRUNCATE 操作的运行时错误 #19445
  - 修复了 `PREPARE statement FROM @Var` 语句在 Var 包含大写字符时候会失败的错误 #19378
  - 修复了在具有大写表名的表上修改 charset 会产生 panic 的错误 #19302
  - 修复了当在包含 `tikv/tiflash` 信息时，`information_schema.statements_summary` 和 `explain` 计划的不一致性 #19159
  - 修复了在测试中 `select into outfile` 出现文件不存在的错误 #19725
  - 修复了 `INFORMATION_SCHEMA.CLUSTER_HARDWARE` 不含有 raid 设备信息的问题 #19457
  - 修复一个问题，使具有 `case-when` 表达式生成列的索引添加操作在遇到 parse 错误时能够正常退出 #19395
  - 修复 DDL 长时间重试的错误 #19488
  - 修复错误，使 `alter table db.t1 add constraint fk foreign key (c2)references t2(c1)` 语句执行不需要先执行 `use db` #19471
  - 修复使日志文件中 dispatch errors 从 Error 形式转变为 Info 信息 #19454

- TiKV

  - 修复开启 collation 时对于非 index 列统计信息估算错误的问题 #8620
  - 修复当迁移 Region 时 Green GC 可能错过 lock 的问题 #8460
  - 修复 TiKV 在极端繁忙下 Raft 成员变更可能出现 panic 的问题 #8497
  - 修复 PD client 和其他线程发起 PD sync requests 可能导致死锁的问题 #8612
  - 升级 jemalloc 到 5.2.1 以解决 huge page 的内存分配问题 #8463
  - 修复 unified thread pool 可能停止工作的问题 #8427

- PD

  - 添加 `initial-cluster-token` 配置避免启动时 cluster 之间的通信 #2922
  - 修正自动模式下 store limit 的单位 #2826
  - 添加对于 scheduler 持久化时引发的错误的处理 #2818
  - 修复 scheduler 的 http 接口的返回结果可能为空的问题 #2871 #2874

- TiFlash

  - 修复在更早版本中修改主键列名后，升级到 v4.0.4/v4.0.5 时 TiFlash 启动失败的问题
  - 修复在修改列的 nullable 属性后访问数据可能抛异常的问题
  - 修复在计算表同步状态时导致的崩溃问题
  - 修复当用户进行一些不兼容的 DDL 操作后，读取 TiFlash 数据遇到异常的问题
  - 修复从 TiDB 同步到不支持的 collation 时，抛出异常的问题
  - 修复 Grafana 中 TiFlash coprocessor executor QPS 面板始终显示为 0 的问题
  - 修复 `FROM_UNIXTIME` 函数遇到 `NULL` 值时返回错误结果的问题

- Tools

  - TiCDC

        * 解决某些场景下内存泄露的问题 #942
        * 解决 Kafka sink 可能会出现的异常退出的问题 #912
        * 解决 CRTs 小于 Resolved Ts 而异常退出的问题 #927
        * 解决同步任务可能卡在 MySQL 上的问题 #936
        * 修复 TiCDC 不合理的 Resolved Ts 超时等待 #8573
    - Backup & Restore (BR)
        * 解决数据校验期间可能出现的异常退出的问题 #479
        * 解决 PD leader 切换后可能出现的异常退出的问题 #496
    - Dumpling
        * 解决 binary 类型的 NULL 值没有被正确处理的问题 #137
    - TiDB Lightning
        * 解决 write 和 ingest 失败后依旧显示成功的问题 #381
        * 解决写 checkpoint 不及时的问题 #386

14.2.12 TiDB 4.0.5 Release Notes

发版日期：2020 年 8 月 31 日

TiDB 版本：4.0.5

14.2.12.1 兼容性变化

- TiDB
    - 修改 `drop partition` 和 `truncate partition` 的参数 #18930
    - 为 `add partition` 操作添加状态检查 #18865

14.2.12.2 新功能

- TiKV
    - 为错误定义错误码 #8387
- TiFlash
    - 支持与 TiDB 统一的 log 格式
- Tools
    - TiCDC
        * 支持加密 Kafka 链接 #764
        * 支持输出 old value #708
        * 添加列的特征的标识 #796
        * 支持输出上一版本的 DDL 和表结构 #799

14.2.12.3 优化提升

- TiDB
  - 优化 Union 场景下 DecodePlan 的开销 #18941
  - 减少 GC 在遇到 Region cache miss 错误时扫描锁的次数 #18876
  - 减少统计信息 feedback 对集群性能的影响 #18772
  - 支持在 RPC 请求返回结果前取消操作 #18580
  - 支持使用 HTTP API 生成带有相关监控项名称的 profile #18531
  - 支持分区表的预打散功能 #17863
  - 在监控面板中显示每个实例的内存使用详情 #18679
  - 在 EXPLAIN 中显示 BatchPointGet 算子的详细运行信息 #18892
  - 在 EXPLAIN 中显示 PointGet 算子的详细运行信息 #18817
  - 解决 MemTracker 潜在的死锁问题 #18395
  - 提高字符串转换为整数类型和小数类型的兼容性，支持将 JSON 转换为时间日期类型 #18159
  - 支持限制 TableReader 算子内存使用 #18392
  - 在 batch cop 请求重试时避免多次 backoff #18999
  - 提升 ALTER TABLE 的兼容性 #19270
  - 单个分区支持 IndexJoin #19151
  - 支持在 log 中存在非法字符时搜索 log #18579
- PD
  - 支持打散特殊存储引擎节点（例如 TiFlash）上的 Region #2706
  - 支持通过 API 指定某范围内的 Region 优先进行调度 #2687
  - 优化 Region 打散操作，使得 Leader 分布更均匀 #2684
  - 针对 TSO 请求添加更多测试和日志 #2678
  - 避免 Region Leader 变化时可能产生的不必要的缓存更新 #2672
  - 增加选项允许 store.GetLimit 返回 tombstone 状态的 store #2743
  - 支持 PD Leader 和 Follower 之间同步 Region Leader 变更 #2795
  - 增加查询 GC safepoint 服务的命令 #2797
  - 替换 filter 中的 region.Clone 调用，优化性能 #2801
  - 增加关闭 Region 流量统计缓存更新的选项，用于提升大规模集群的性能 #2848
- TiFlash
  - 添加更多的 Grafana 监控面板，比如 CPU、I/O、RAM 使用量，以及存储引擎的各项指标
  - 通过优化 Raft logs 的处理逻辑，减少 I/O 操作
  - 加快 add partition DDL 之后 Region 的调度速度
  - 优化 DeltaTree 引擎中 delta 数据的整理，减少读写放大
  - 通过使用多线程对 Region snapshot 进行预处理，优化从 TiKV 同步 Region 副本的性能
  - 优化系统负载较低时打开文件描述符的数量，降低系统资源占用量
  - 减少 TiFlash 重启时新创建的文件数量
  - 支持数据存储的静态加密功能
  - 支持数据传输的 TLS 功能
- Tools

- TiCDC
  * 减少了获取时间戳的频率 #801
- Backup & Restore (BR)
  * 优化了日志 #428
- Dumpling
  * 减少导出 MySQL 时持锁的时间 #121
- TiDB Lightning
  * 优化了日志 #352

14.2.12.4 Bug 修复

- TiDB
  - 修复 `builtinCastRealAsDecimalSig` 函数中未正确处理 `ErrTruncate/Overflow` 错误导致报 `should ensure all columns have the same length` 错误的问题 #18967
  - 修复 `pre_split_regions` 对分区表不生效的问题 #18837
  - 修复大事务提前终止的问题 #18813
  - 修复使用 `collation` 相关函数查询结果错误的问题 #18735
  - 修复 `getAutoIncrementID()` 函数逻辑错误导致导出工具报 `table not exist` 错误的问题 #18692
  - 修复 `select a from t having t.a` 报 `unknown column error` 的问题 #18434
  - 修复 Hash 分区表的分区键为整数类型时，写入 64 位无符号类型导致溢出 panic 的问题 #18186
  - 修复 `char` 函数行为错误的问题 #18122
  - 修复 `ADMIN REPAIR TABLE` 无法解析 range 分区表表达式中整数的问题 #17988
  - 修复 `SET CHARSET` 行为不正确的问题 #17289
  - 修复由于错误的设置 collation 导致 `collation` 函数返回错误结果的问题 #17231
  - 修复 `STR_TO_DATE` 和 MySQL 行为不一致的问题 #18727
  - 修复 `cluster_info` 表中，TiDB 版本和 PD/TiKV 不一致的问题 #18413
  - 修复悲观事务未能检查出重复数据导致可以重复写入冲突数据的问题 #19004
  - 修复 `union select for update` 存在并发竞态的问题 #19006
  - 修复自查询含有 `PointGet` 算子时返回结果错误的问题 #19046
  - 修复 `IndexLookUp` 作为 `Apply` 的内连接算子时查询结果不正确的问题 #19496
  - 修复 `anti-semi-join` 查询结果不正确的问题 #19472
  - 修复 `BatchPointGet` 查询结果不正确的问题 #19456
  - 修复 `UnionScan` 作为 `Apply` 的内连接算子时查询结果不正确的问题 #19496
  - 修复使用 `EXECUTE` 语句产生大查询日志造成 panic 的问题 #17419
  - 修复 `IndexJoin` 在使用 `ENUM` 或 `SET` 类型作为连接键报错的问题 #19235
  - 修复在索引值为 `NULL` 时无法构建出查询范围的问题 #19358
  - 修复更新全局配置导致的数据竞态问题 #17964
  - 修复修改 schema 字符集导致 panic 的问题 #19286
  - 修复修改文件夹对中间结果落盘功能的影响 #18970
  - 修复 `decimal` 类型哈希值不正确的问题 #19131
  - 修复 `PointGet` 和 `BatchPointGet` 在分区表场景下报错的问题 #19141
  - 修复共同使用 `Apply` 算子和 `UnionScan` 算子时查询结果不正确的问题 #19104

  - 修复生成列索引结果不正确的问题 #17989
  - 修复并发收集统计信息 panic 的问题 #18983

- TiKV

  - 修复开启 Hibernate Region 时，某些情况下 leader 选举慢的问题 #8292
  - 修复 Region 调度产生的一个内存泄露问题 #8357
  - 增加 `hibernate-timeout` 配置避免 leader 过快变为 Hibernate 状态 #8208

- PD

  - 修复 PD leader 切换时可能导致一段时间内 TSO 不可用的问题 #2666
  - 修复开启 Placement Rule 时，某些情况下 Region 无法调度至最佳状态的问题 #2720
  - 修复开启 Placement Rules 后，`Balance Leader` 不工作的问题 #2726
  - 修复不健康的 Store 未从负载统计信息中过滤的问题 #2805

- TiFlash

  - 修复 TiFlash 从旧版本升级到新版本的过程中，由于包含特殊字符而导致进程无法启动的问题
  - 修复 TiFlash 进程在初始化过程中，一旦出现任何异常就无法退出的问题

- Tools

  - Backup & Restore (BR)

    * 修复 total KV 和 total bytes 被计算两次的问题 #472
    * 修复切换模式不及时导致恢复缓慢的问题 #473

  - Dumpling

    * 修复 FTWRL 锁没有及时释放的问题 #128

  - TiCDC

    * 解决了同步任务不能被移除的问题 #782
    * 修正了错误的删除事件 #787
    * 解决了已停止的同步任务会卡住 GC 的问题 #797
    * 解决了网络阻塞导致同步任务不能退出的问题 #825
    * 修复在某些情况下无关数据被错误地到下游的问题 #743

  - TiDB Lightning

    * 解决了 TiDB backend 遇到空 binary/hex 的时候出现语法错误的问题 #357

14.2.13 TiDB 4.0.4 Release Notes

发版日期：2020 年 7 月 31 日

TiDB 版本：4.0.4

14.2.13.1 Bug 修复

- TiDB
  - 修复查询 `information_schema.columns` 卡死的问题 #18849
  - 修复 `PointGet` 和 `BatchPointGet` 在遇到 `in(null)` 条件时出错的问题 #18848
  - 修复 `BatchPointGet` 算子结果不正确的问题 #18815
  - 修复 `HashJoin` 算子在遇到 `set`、`enum` 类型时查询结果不正确的问题 #18859

14.2.14 TiDB 4.0.3 Release Notes

发版日期：2020 年 7 月 24 日

TiDB **版本：**4.0.3

14.2.14.1 新功能

- TiDB Dashboard
  - 显示详细的 TiDB Dashboard 版本信息 #679
  - 显示不受支持的浏览器或过时的浏览器的兼容性通知 #654
  - 支持在 SQL 语句分析页面搜索 #658
- TiFlash
  - TiFlash proxy 支持文件加密功能
- Tools
  - Backup & Restore (BR)
    - 支持使用 zstd、lz4、snappy 算法压缩备份文件 #404
  - TiCDC
    - 支持 sink-uri 中配置 Kafka 客户端的 ID #706
    - 支持离线更新同步任务的配置 #699
    - 支持自定义同步任务的 ID #727
    - 支持使用 SSL 加密链接向 MySQL 输出数据 #347
    - 支持输出 Avro 格式的变更数据 #753
    - 支持向 Apache Pulsar 输出变更数据 #751
  - Dumpling
    - 支持自定义 CSV 文件的分隔符和换行符 #116
    - 支持自定义输出文件名格式 #122

14.2.14.2 改进提升

- TiDB
  - 增加全局变量 `tidb_log_desensitization` 来控制在日志中记录 SQL 时是否脱敏 #18581
  - 默认打开 `tidb_allow_batch_cop` #18552
  - 加速 `kill tidb sesesion_id` 的执行速度 #18505
  - 函数 `tidb_decode_plan` 的结果增加表头输出 #18501
  - 配置检查器可以兼容旧版本的配置文件 #18046
  - 默认打开执行信息的收集 #18518
  - 增加系统表 `tiflash_tables` 和 `tiflash_segments` #18536
  - `AUTO RANDOM` 被移出实验特性并正式 GA，有如下的改进和兼容性修改：
    * 在配置文件中，将 `experimental.allow-auto-random` 废弃，该无论该选项如何配置，都可以在列上定义 `AUTO_RANDOM` 属性 #18613 #18623
    * 为避免显式写入 `AUTO_RANDOM` 列造成非预期的 `AUTO_RANDOM_BASE` 的更新，新增 session 变量 `tidb_allow_auto_random_explicit_insert` 用于控制 `AUTO_RANDOM` 列的显式写入，该变量默认值为 `false` #18508
    * 为避免分配空间被快速消耗，`AUTO_RANDOM` 列现在仅允许在 `BIGINT` 和 `UNSIGNED BIGINT` 列上定义，并将最大的 Shard Bit 数量限制为 15 #18538
    * 当在 `BIGINT` 列上定义 `AUTO_RANDOM` 属性，并显示插入负值的整型主键时，将不会再触发 `AUTO_RANDOM_BASE` 的更新 #17987
    * 当在 `UNSIGNED BIGINT` 列上定义 `AUTO_RANDOM` 属性，分配 ID 时将利用整数的最高位以获得更大的分配空间 #18404
    * 在 `SHOW CREATE TABLE` 的结果中支持 `AUTO_RANDOM_BASE` 属性的更新 #18316
- TiKV
  - 添加了新的配置项 `backup.num-threads` 用语控制 backup 线程池的大小 #8199
  - 收取 snapshot 时不再发送 store heartbeat #8136
  - 支持动态调整 `shared block cache` 的大小 #8232
- PD
  - 支持 JSON 格式日志 #2565
- TiDB Dashboard
  - 优化 key Visualizer 中冷表的 bucket 合并 #674
  - 重命名配置项 `disable-telemetry` 以使遥测更一致 #684
  - 切换页面时显示进度条 #661
  - 保证慢日志查询和日志查询行为的一致性，即使在空格存在的情况 #682
- TiFlash
  - 将 Grafana DDL Jobs 面板中的单位修改为 `operations per minute`
  - 在 Grafana 中新增关于 TiFlash-Proxy 的详细监控指标面板
  - 降低 TiFlash Proxy 的 IOPS

- Tools
  - TiCDC
    * 将监控指标总的表 ID 替换为表名 #695
  - Backup & Restore (BR)
    * 支持输出 JSON 格式的日志 #336
    * 支持在运行 BR 期间动态开启 pprof #372
    * 加速恢复时 DDL 的执行速度 #377
  - TiDB Lightning
    * 使用一种更加简单易懂的表过滤机制替换原先的黑白名单机制 #332

14.2.14.3 Bug 修复

- TiDB
  - 当 `IndexHashJoin` 遇到执行中发生非内存相关的错误时，返回错误而不是空结果集 #18586
  - 修复 gRPC transportReader 导致的反复异常 #18562
  - 修复因为 Green GC 不会扫描已下线 store 上的锁而可能导致数据不完整的问题 #18550
  - 非只读语句不会使用 TiFlash 引擎 #18534
  - 当查询连接异常时返回真实的错误信息 #18500
  - 修复非 repair mode 的 TiDB 节点不会重新读取修复的表元信息的错误 #18323
  - 修复当锁住的 primary key 在当前事务被插入/删除时可能造成的结果不一致问题 #18291
  - 修复数据落盘为正确生效导致的内存溢出 #18288
  - 修复 `REPLACE INTO` 语句作用在包含生成列的表时会错误报错的问题 #17907
  - 当 `IndexHashJoin` 及 `IndexMergeJoin` 执行异常时抛出 `Out Of Memory Quota!` 错误 #18527
  - 修复当 `Index Join` 使用的索引包含整型主键时，特殊情况下执行结果可能出错的问题 #18565
  - 修复当开启 new collation 时，若在事务内的更新涉及了 new collation 列，并在该事务内通过唯一索引读取更新数据时，被更新的数据无法被读取到的问题 #18703
- TiKV
  - 修复 merge 期间可能读到过期数据的问题 #8113
  - 修复聚合函数 `min/max` 下推到 TiKV 时，collation 不能正确工作的问题 #8108
- PD
  - 修复如果服务器崩溃，创建 TSO 流可能会被阻塞一段时间的问题 #2648
  - 修复 `getSchedulers` 可能导致数据争用的问题 #2638
  - 修复删除 `scheduler` 时导致死锁的问题 #2637
  - 修复 `balance-leader-scheduler` 没有考虑 placement rule 的问题 #2636
  - 修复有时无法正确设置 `safepoint` 的问题，这可能会使 BR 和 dumpling 失败 #2635
  - 修复 `hot region scheduler` 中目标 store 选择错误的问题 #2627
  - 修复 PD Leader 切换时 TSO 请求可能花费太长时间的问题 #2622
  - 修复 PD Leader 切换后过期 `scheduler` 的问题 #2608
  - 修复了启用 placement rule 时，有时 Region 的副本可能无法调整到最佳位置的问题 #2605

  - 修复了存储的部署路径不会随着部署目录移动而更新的问题 #2600
  - 修复了 `store limit` 可能为零的问题 #2588

- TiDB Dashboard

  - 修复 TiDB 扩容时的 TiDB 连接错误 #689
  - 修复 TiFlash 实例未显示在日志搜索页面的问题 #680
  - 修复概况页面刷新之后 metrics 会重置的问题 #663
  - 修复某些 TLS 方案中的连接问题 #660
  - 修复在某些情况下无法正确显示语言的下拉列表 #677

- TiFlash

  - 修复更改主键列名后 TiFlash 崩溃的问题
  - 修复 Learner Read 与 Remove Region 并发时可能的死锁问题

- Tools

  - TiCDC

    * 解决了某些场景下可能发生的 OOM 问题 #704
    * 解决了某些特殊表名可能导致 SQL 语法出错的问题 #676
    * 解决了同步任务处理单元无法正常退出的问题 #693

  - Backup & Restore (BR)

    * 解决了备份汇总报告中时间为负数的问题 #405

  - Dumpling

    * 解决了 NULL 值在有 `-r` 参数时被忽略的问题 #119
    * 解决了导出数据时 flush table 没有正常工作的问题 #117

  - TiDB Lightning

    * 解决了 `--log-file` 参数不生效的问题 #345

  - TiDB Binlog

    * 修复开启 TLS 写下游时用来保存 checkpoint 的 DB 没有开启 TLS 导致 Drainer 无法启动的问题 #988

14.2.15 TiDB 4.0.2 Release Notes

发版日期：2020 年 7 月 1 日

TiDB 版本：4.0.2

14.2.15.1 兼容性

- TiDB

  - 移除慢查询日志和 statement summary 表中的敏感信息 #18130
  - 禁止在 sequence 缓存中出现负数 #18103
  - `CLUSTER_INFO` 表中不再显示 tombstone 状态的 TiKV 和 TiFlash 节点 #17953

- 诊断规则 current-load 变更为 node-check #17660

- PD
  - 持久化 store-limit 配置项，弃用 store-balance-rate 配置 #2557

14.2.15.2 新更改

- TiDB 及 TiDB Dashboard 默认收集使用情况信息，并将这些信息分享给 PingCAP 用于改善产品 #18180。若要了解所收集的信息详情及如何禁用该行为，请参见遥测文档。

14.2.15.3 新功能

- TiDB
  - 支持在 INSERT 语句中使用 MEMORY_QUOTA() hint #18101
  - 支持基于 TLS 证书 SAN 属性的登录认证 #17698
  - REGEXP() 函数支持 collation #17581
  - 支持会话和全局变量 sql_select_limit #17604
  - 支持新增分区时自动分裂 Region 的功能 #17665
  - 支持函数 IF()/BITXOR()/BITNEG()/JSON_LENGTH() 下推到 TiFlash Coprocessor 上执行 #17651 #17592
  - 支持聚合函数 APPROX_COUNT_DISTINCT()，用于快速计算 COUNT(DISTINCT) 的近似值 #18120
  - TiFlash 支持了 collation，支持相应的函数下推 #17705
  - INFORMATION_SCHEMA.INSPECTION_RESULT 表新增 STATUS_ADDRESS 列，用于展示节点的 status 地址 #17695
  - MYSQL.BIND_INFO 表新增 SOURCE 列，用于展示 binding 的创建方式 #17587
  - PERFORMANCE_SCHEMA.EVENTS_STATEMENTS_SUMMARY_BY_DIGEST 表新增 PLAN_IN_CACHE 和 PLAN_CACHE_HITS 列，用于展示 plan cache 的使用情况 #17493
  - 新增配置项 enable-collect-execution-info 和会话级变量 tidb_enable_collect_execution_info 用于控制是否记录算子的执行信息并打印到慢查询日志中 #18073 #18072
  - 新增全局变量 tidb_slow_log_masking，用于控制是否脱敏慢查询日志中的查询 #17694
  - 增加对 TiKV 配置项 storage.block-cache.capacity 的诊断规则 #17671
  - 新增 SQL 语法 BACKUP/RESTORE 来进行数据备份恢复 #15274

- TiKV
  - TiKV Control 支持 encryption-meta 命令 #8103
  - 增加 RocksDB::WriteImpl 相关的 perf context 监控 #7991

- PD
  - 对 leader 执行 remove-peer 操作时，让这个 operator 不等待超时，立刻失败 #2551
  - 对 TiFlash 节点设置更合理的 store limit 配置默认值 #2559

- TiFlash
  - Coprocessor 支持新的聚合函数 APPROX_COUNT_DISTINCT

  - 存储引擎中的粗糙索引默认开启
  - 支持运行在 ARM 架构
  - Coprocessor 支持 JSON_LENGTH 函数下推

- TiCDC

  - 支持 Capture 节点扩容时迁移部分子任务到新加节点 #665
  - Cli 中添加清理 TiCDC GC TTL 的功能 #652
  - 在 MQ sink 中支持输出 Canal 协议 #649

14.2.15.4 改进提升

- TiDB

  - 降低当集群中 CM-Sketch 占用过多内存时，Golang 内存分配带来的查询延迟 #17545
  - 缩短 TiKV 故障恢复时集群 QPS 的恢复时间 #17681
  - 为 partition 表上的查询支持聚合函数下推到 TiKV 或者 TiFlash coprocessor #17655
  - 提升索引上等值条件的行数估算准确度 #17611

- TiKV

  - 优化 PD client panic 日志信息 #8093
  - 重新加回 `process_cpu_seconds_total` 和 `process_start_time_seconds` 监控 #8029

- TiFlash

  - 提升从旧版本升级时的兼容性
  - 降低 delta index 的内存使用量
  - 使用更高效的 delta index update 算法

- Tools

  - Backup & Restore (BR)

    * 提升多表场景下的恢复数据性能 #266

14.2.15.5 Bug 修复

- TiDB

  - 修复 `tidb_isolation_read_engines` 更改后从 plan cache 中获取的执行计划不正确的问题 #17570
  - 修复某些情况下 `EXPLAIN FOR CONNECTION` 返回运行时错误的问题 #18124
  - 修复某些情况下 `last_plan_from_cache` 结果不正确的问题 #18111
  - 修复执行 plan cache 中的 `UNIX_TIMESTAMP()` 时的运行时错误 #18002 #17673
  - 修复 `HashJoin` 算子的孩子节点返回 `NULL` 类型的结果时，计算过程中的运行时错误 #17937
  - 修复当在同一个数据库中并发执行 `DROP DATABASE` 语句和相关 DDL 语句时的运行错误 #17659
  - 修复当 `COERCIBILITY()` 的输入参数是用户变量时结果不正确的问题 #17890
  - 修复 `IndexMergeJoin` 算子偶尔卡住的问题 #18091

  - 修复 IndexMergeJoin 算子触发 oom-action 后被取消执行时卡住的问题 #17654
  - 修复 Insert 和 Replace 算子的内存统计过大的问题 #18062
  - 修复在执行 DROP DATABASE 的同时对同一个数据库中的表 DROP TABLE 时，数据不再向 TiFlash 同步的问题 #17901
  - 修复 TiDB 和对象存储服务之间 BACKUP/RESTORE 失败的问题 #17844
  - 修复权限检查失败时的错误信息 #17724
  - 修复 DELETE/UPDATE 语句的 feedback 被错误收集的问题 #17843
  - 禁止更改非 AUTO_RANDOM 表的 AUTO_RANDOM_BASE 值 #17828
  - 修复通过 ALTER TABLE ... RENAME 在数据库间移动表时，AUTO_RANDOM 列分配到错误结果的问题 #18243
  - 修复系统变量 tidb_isolation_read_engines 的值中没有 tidb 时某些系统表无法访问的问题 #17719
  - 修复 JSON 中大整数和浮点数比较的精度问题 #17717
  - 修复 COUNT() 函数的返回类型中 DECIMAL 不正确的问题 #17704
  - 修复 HEX() 函数的输入类型是二进制字符串时结果不正确的问题 #17620
  - 修复查询 INFORMATION_SCHEMA.INSPECTION_SUMMARY 表没有指定过滤条件时返回结果为空的问题 #17697
  - 修复 ALTER USER 语句使用哈希后的密码更新用户信息后，密码不符合预期的问题 #17646
  - 为 ENUM 和 SET 类型支持 collation #17701
  - 修复 CREATE TABLE 时预切分 Region 的超时机制不生效的问题 #17619
  - 修复某些情况下 DDL 后台作业重试时，schema 未正确更新导致的 DDL 原子性问题 #17608
  - 修复 FIELD() 函数的参数包含 column 时结果不正确的问题 #17562
  - 修复某些情况下 max_execution_time hint 不生效的问题 #17536
  - 修复某些情况下 EXPLAIN ANALYZE 的结果中并发信息被多次打印的问题 #17350
  - 修复对 STR_TO_DATE 函数的 %h 解析和 MySQL 不兼容问题 #17498
  - 修复 tidb_replica_read 设置成 follower，并且 Region 的 leader 和 follower/learner 之间出现网络分区后，TiDB 发送的 request 一直重试的问题 #17443
  - 修复某些情况下 TiDB 一直 ping PD 的 follower 的问题 #17947
  - 修复老版本的 range partition 表无法在 4.0 集群中加载的问题 #17983
  - 修复当多个 Region 的请求同时超时时整个 SQL 语句超时的问题 #17585
  - 修复解析日期类型的分隔符时和 MySQL 不兼容的问题 #17501
  - 修复少数情况下发给 TiKV 的请求错发给 TiFlash 的问题 #18105
  - 修复当前事务中主键被插入/删除但主键的锁却被另一事务清除可能造成结果不一致的问题 #18250

- TiKV

  - 修复 status server 的内存安全问题 #8101
  - 修复 json 数字比较的精度丢失问题 #8087
  - 修改错误的慢查询日志 #8050
  - 修复 merge 可能导致 peer 无法被移除的问题 #8048
  - 修复 tikv-ctl recover-mvcc 未清除无效的悲观锁 #8047
  - 修复一些遗漏的 Titan 监控 #7997
  - 修复向 TiCDC 返回 duplicated error 的问题 #7887

- PD

  - 验证 pd-server.dashboard-address 配置项的正确性 #2517

- 修复设置 store-limit-mode 为 auto 时可能引起 PD panic 的问题 #2544
- 修复某些情况下热点不能识别的问题 #2463
- 修复某些情况下 Placement Rules 会使 store 状态变更为 tombstone 的进程被阻塞的问题 #2546
- 修复某些情况下从低版本升级后，PD 无法正常启动的问题 #2564

- TiFlash

  - 修正 proxy 遇到 region not found 时可能的 panic 的问题
  - 修正 schema 同步遇到 I/O exception 时可能无法继续同步的问题

14.2.16 TiDB 4.0.1 Release Notes

发版日期：2020 年 6 月 12 日

TiDB 版本：4.0.1

14.2.16.1 新功能

- TiKV

  - 添加 --advertise-status-addr 启动参数 #8046

- PD

  - 为内嵌的 TiDB Dashboard 添加内部代理的支持 #2511
  - 添加对 PD 客户端自定义超时的设置 #2509

- TiFlash

  - 支持 TiDB new collation framework 排序规则框架
  - 支持函数 If/BitAnd/BitOr/BitXor/BitNot/Json_length 下推到 TiFlash
  - 支持 TiFlash 中对于大事务的 Resolve Lock 逻辑

- Tools

  - Backup & Restore (BR)

    * 增加启动时集群版本检查，避免 BR 和 TiDB 集群不兼容的问题 #311

14.2.16.2 Bug 修复

- TiKV

  - 修复日志中 use-unified-pool 配置打印不正确的问题 #7946
  - 修复 tikv-ctl 不支持相对路径的问题 #7963
  - 修复点查监控指标不准确的问题 #8033
  - 修复过时副本在网络隔离消除后不能销毁的问题 #8006
  - 修复 read index 可能过时的问题 #8043
  - 改善备份恢复文件操作的可靠性 #7917

- PD
  - 防止某些场景下对 Placement Rules 的错误配置 #2516
  - 修复删除 Placement Rules 可能引发 panic 的问题 #2515
  - 修复当 Store 的已用空间为零时无法获取 Store 信息的 bug #2474
- TiFlash
  - 修复 TiFlash 中 Bit 类型列的 Default Value 解析不正确的问题
  - 修复 TiFlash 对于 1970-01-01 00:00:00 UTC 在部分时区下计算错误的问题

14.2.17 TiDB 4.0 GA Release Notes

发版日期：2020 年 5 月 28 日

TiDB 版本：4.0.0

14.2.17.1 兼容性变化

- TiDB
  - 优化事务过大时系统的报错信息，方便排查问题 #17219
- TiCDC
  - 优化 Changefeed 配置文件的结构，提升易用性 #588
  - 新增 `ignore-txn-start-ts` 配置项，过滤事务时条件由原来的 `commit_ts` 改为 `start_ts` #589

14.2.17.2 重点修复的 Bug

- TiKV
  - 修复 BR 备份时出现 `DefaultNotFound` 错误的问题 #7937
  - 修复 `ReadIndex` 因响应的数据包乱序而导致系统 panic 的问题 #7930
  - 修复 TiKV 重启后由于 snapshot 文件被错误删除导致系统 panic 的问题 #7927
- TiFlash
  - 修复因 Raft Admin Command 处理逻辑不正确，系统 panic 导致数据可能会丢失的问题

14.2.17.3 新功能

- TiDB
  - 新增 `committer-concurrency` 配置项，用于控制 retry commit 阶段的 goroutine 数量 #16849
  - 支持 `show table partition regions` 语法 #17294
  - 新增 `tmp-storage-quota` 配置项，用于限制 tidb-server 使用的临时磁盘空间 #15700
  - 创建和更改表时新增检查分区表是否使用唯一前缀索引的功能 #17213
  - 支持 `insert/replace into tbl_name partition(partition_name_list)` 语句 #17313

  - Distinct 函数支持检查 collations 的值 #17240
  - 哈希分区裁剪时支持 `is null` 过滤条件 #17310
  - 分区表中支持 `admin check index`、`admin cleanup index` 和 `admin recover index` #17392 #17405 #17317
  - 支持 `in` 表达式的范围分区裁剪 #17320

- TiFlash
  - Learner 读取数据时通过 Lock CF 的 `min commit ts` 值过滤出符合条件的 TSO 对应的数据
  - 若 Timestamp 类型的值小于 `1970-01-01 00:00:00`，系统显式报错以避免计算结果出错
  - Search log 的正则表达式支持使用 flag 参数

- TiKV
  - 支持 `ascii_bin` 和 `latin1_bin` 编码的排序规则 #7919

- PD
  - 支持为内置的 TiDB Dashboard 指定反向代理资源前缀 #2457
  - PD client Region 相关接口支持返回 `pending peer` 和 `down peer` 的信息 #2443
  - 添加 `Direction of hotspot move leader`、`Direction of hotspot move peer` 和 `Hot cache read entry number` 等监控 #2448

- Tools
  - Backup & Restore (BR)
    * 支持备份与恢复 `Sequence` 和 `View` #242
  - TiCDC
    * 创建 Changefeed 时新增检查 Sink URI 的合法性 #561
    * 系统启动时检查 PD 和 TiKV 版本是否符合系统要求 #570
    * 支持同一个调度任务生成周期内可调度多张表 #572
    * HTTP API 中增加节点角色的信息 #591

14.2.17.4 Bug 修复

- TiDB
  - 修复收发消息有不符合预期的超时，禁止合并发向 TiFlash 的消息 #17307
  - 修复分区裁剪时未正确区分有符号/无符号整数的错误，并提高了性能 #17230
  - 修复 3.1.1 升级到 4.0 时由于 `mysql.user` 表不兼容导致升级失败的问题 #17300
  - 修复 `update` 语句分区选择不正确的问题 #17305
  - 修复从 TiKV 收到未知错误消息时系统 panic 的问题 #17380
  - 修复创建按 key 分区的分区表由于处理逻辑不正确导致系统 panic 的问题 #17242
  - 修复某些情况下优化器处理逻辑不正确导致错误地选择 `Index Merge Join` 的问题 #17365
  - 修复 Grafana 中 `SELECT` 语句的 `duration` 的监控指标不准确的问题 #16561
  - 修复当系统发生错误时，GC 线程卡住的问题 #16915
  - 修复当列的类型是 `Boolean` 时，由于 `UNIQUE` 约束比较的逻辑不正确导致输出结果不正确的问题 #17306
  - 修复 `tidb_opt_agg_push_down` 开启且聚合函数下推分区表信息时，由于逻辑处理不正确导致系统 panic 的问题 #17328

  - 修复某些情况下会访问已经发生故障的 TiKV 的问题 #17342
  - 修复 `tidb.toml` 中的配置项 `isolation-read` 不生效时的问题 #17322
  - 修复通过 `hint` 强制执行流式聚合时由于逻辑处理不正确导致输出的结果顺序不正确的问题 #17347
  - 修复不同 `SQL_MODE` 时 `insert` 处理 `DIV` 的行为 #17314

- TiFlash

  - 修复 Search log 功能正则表达式匹配的行为与其他组件不一致的问题
  - 默认关闭 Raft Compact Log Command 延迟处理的优化，避免节点大量写入数据时重启时间过长的问题
  - 修复部分场景因 TiDB 执行 `DROP DATABASE` 处理逻辑不正确导致系统启动不成功的问题
  - 修复采集 `Server_info` 中 CPU 信息方式与其他组件不一样的问题
  - 修复开启 `batch coprocessor` 功能时，执行 `Query` 报错 `Too Many Pings` 的问题
  - 修复 Dashboard 因未上报 `deploy path` 信息导致相关信息显示不正确的问题

- TiKV

  - 修复 BR 备份时出现 `DefaultNotFound` 错误的问题 #7937
  - 修复 `ReadIndex` 因响应的数据包乱序时导致系统 panic 问题 #7930
  - 修复读请求回调函数没有被调用，导致返回非预期错误的问题 #7921
  - 修复 TiKV 重启后由于 snapshot 文件被错误删除导致系统 panic 问题 #7927
  - 修复存储加密中因处理逻辑不正确导致 master key 无法轮转的问题 #7898
  - 修复开启存储加密后 snapshot 的 lock cf 文件在接收后未被加密的问题 #7922

- PD

  - 修复使用 pd-ctl 删除 `evict-leader-scheduler` 或者 `grant-leader-scheduler` 时报 404 错误的问题 #2446
  - 修复当存在 TiFlash 副本的时候，可能会导致 `presplit` 功能无法正常使用的问题 #2447

- Tools

  - BR

    * 修复从云存储恢复数据时因网络原因导致恢复失败的问题 #298

  - TiCDC

    * 修复若干因数据争用 (data race) 导致系统 panic 的问题 #565 #566
    * 修复若干因处理逻辑不正确导致资源泄露或系统阻塞的问题 #574 #586
    * 修复 CLI 因连接不上 PD 导致命令行阻塞的问题 #579

14.2.18 TiDB 4.0 RC.2 Release Notes

发版日期： 2020 年 5 月 15 日

TiDB 版本： 4.0.0-rc.2

14.2.18.1 兼容性变化

- TiDB

  - 去掉了特别为开启 Binlog 时定义的事务容量上限（100 MB），现在事务的容量上限统一为 10 GB，但若开启 Binlog 且下游是 Kafka，由于 Kafka 消息大小的限制是 1 GB，请根据情况调整 `txn-total-size-limit` 配置参数 #16941
  - 查询 `CLUSTER_LOG` 表时，如果未指定时间范围，由默认时间范围变更为返回错误且用户必须指定时间范围 #17003
  - `CREATE TABLE` 创建分区表时指定未支持的 sub-partition 或 linear hash 选项，将会创建非分区普通表，而不是选项未生效的分区表 #17197

- TiKV

  - 将加密相关的配置移到 security 分类下，即调整配置项 `[encryption]` 为 `[security.encryption]` #7810

- Tools

  - TiDB Lightning

    * 导入数据时将 SQL Mode 由默认改成 `ONLY_FULL_GROUP_BY,NO_AUTO_CREATE_USER`，提高兼容性 #316
    * 在 tidb-backend 模式下禁止访问 PD 或者 TiKV 端口 #312
    * 日志信息默认输出到 tmp 文件且在启动时输出 tmp 文件的路径 #313

14.2.18.2 重点修复的 Bug

- TiDB

  - 修复当 `WHERE` 语句只有一个等值条件时错误选择分区表分区的问题 #17054
  - 修复当 `WHERE` 语句只包含字符串列时构造错误的 Index Range 导致结果错误的问题 #16660
  - 修复事务中执行 `DELETE` 之后再进行唯一索引点查语句 Panic 的问题 #16991
  - 修复 GC worker 在有错误发生时可能死锁的问题 #16915
  - 避免 TiKV 未宕机仅响应慢情况下的无故 RegionMiss 重试 #16956
  - 修改客户端 MySQL 协议握手阶段日志级别为 DEBUG，以解决干扰日志输出的问题 #16881
  - 修复 `TRUNCATE` 后未按照表定义的 `PRE_SPLIT_REGIONS` 信息进行预切分 Region 的问题 #16776
  - 修复两阶段提交中第二阶段遇到 TiKV 不可用重试导致的 Goroutine 暴涨的问题 #16876
  - 修复部分表达式不能下推可能导致语句执行 Panic 的问题 #16869
  - 修复 IndexMerge 在分区表上执行结果错误的问题 #17124
  - 修复因 Memory Tracker 锁竞争引起的宽表性能回退问题 #17234

- TiFlash

  - 修复库名、表名含特殊字符，系统升级后无法正常启动的问题

14.2.18.3 新功能

- TiDB

  - 新增 `BACKUP` 和 `RESTORE` 语句进行备份与恢复 #16960
  - 支持在提交前预检查单个 Region 提交数据量，并在超过阈值时预切分 Region 后再进行提交 #16959

  - 新增 Session 作用域变量 LAST_PLAN_FROM_CACHE，用于指示上一条语句是否命中 Plan Cache #16830
  - 支持在慢日志和 SLOW_LOG 表中记录 Cop_time 信息 #16904
  - 支持在 Grafana 中展示更多 Go Runtime 内存监控指标 #16928
  - 支持在 General Log 中输出 forUpdateTS 和 Read Consistency 隔离级别信息 #16946
  - 支持对 TiKV Region Resolve Lock 请求进行去重 #16925
  - 支持 SET CONFIG 语句进行 PD/TiKV 节点配置修改 #16853
  - 支持在 CREATE TABLE 语句中指定 auto_random 选项 #16813
  - 通过为 DistSQL 请求分配 TaskID 让 TiKV 更好地调度处理请求 #17155
  - 支持在 MySQL 客户端登录后显示 TiDB server 版本信息 #17187
  - 支持在 GROUP_CONCAT 中指定 ORDER BY 子句 #16990
  - 支持在 Slow Log 中展示 Plan_from_cache 信息，用于指示语句是否命中 Plan Cache #17121
  - Dashboard 支持显示 TiFlash 多盘部署容量信息功能
  - Dashboard 支持通过 SQL 查询 TiFlash 日志的功能

- TiKV

  - 加密存储适配 tikv-ctl，适配后开启加密存储后通过 tikv-ctl 操作管理集群 #7698
  - 新增加密码 Snapshot 中的 lock column famliy 的功能 #7712
  - 修改 Raftstore latency 在 Grafana 面板显示方式，采用 heatmap 方便诊断性能抖动问题 #7717
  - 支持配置 gRPC 消息大小的上限 #7824
  - Grafana 面板中添加了 encryption 相关的监控 #7827
  - TiKV 支持 ALPN 协议 #7825
  - 添加了更多的关于 Titan 的统计信息 #7818
  - 统一线程池支持用客户端提供的 task ID 来区分任务，以避免一个请求被来自同一个事务的另一个请求降低优先级 #7814
  - 提升了 batch insert 请求的性能 #7718

- PD

  - 下线节点时放开 Remove peer 的速度 #2372

- TiFlash

  - 调整 Grafana 中 Read Index 的 Count 图表的名称为 Ops
  - 优化系统负载较低时打开文件描述符的数据，降低系统资源占用量
  - 新增 capacity 配置参数用于限制存储数据容量

- Tools

  - TiDB Lightning

    * tidb-lightning-ctl 新增 fetch-mode 子命令，输出 TiKV 集群模式 #287

  - TiCDC

    * 支持通过 cdc cli 来管理同步任务 (changefeed) #546

  - Backup & Restore (BR)

    * 支持备份时候自动调整 GC 时间 #257
    * 优化恢复数据时的 PD 参数，以加速恢复 #198

14.2.18.4 Bug 修复

- TiDB
  - 完善多个算子中判断是否使用向量化进行表达式执行的逻辑 #16383
  - 修复 IndexMerge Hint 未能正确检查数据库名称的问题 #16932
  - 修复 Sequence 可以被 TRUNCATE 的问题 #17037
  - 修复 Sequence 可以被 INSERT/UPDATE/ANALYZE/DELETE 的问题 #16957
  - 修复启动阶段执行的内部 SQL 在 Statement Summary 表中未能正确记录为内部 SQL 的问题 #17062
  - 修复因 TiFlash 支持但 TiKV 不支持的过滤条件仅被下推到 IndexLookupJoin 算子之下导致的语句报错问题 #17036
  - 修复开启 Collation 后，LIKE 表达式可能出现的并发问题 #16997
  - 修复开启 Collation 后，LIKE 函数无法正确构造 Range 查询索引的问题 #16783
  - 修复触发填充 Plan Cache 语句后执行 @@LAST_PLAN_FROM_CACHE 返回值错误的问题 #16831
  - 修复为 IndexMerge 计算候选路径时漏掉 Index 上的 TableFilter 的问题 #16947
  - 修复使用 MergeJoin Hint 并存在 TableDual 算子时无法产生物理查询计划的问题 #17016
  - 修复 Statement Summary 表的 Stmt_Type 列值大小写错误的问题 #17018
  - 修复因不同用户使用相同的 tmp-storage-path 导致服务无法启动报 Permission Denied 错误问题 #16996
  - 修复返回结果类型由多个输入列决定的表达式（例如：CASE WHEN）结果类型 NotNullFlag 标识推导不正确的问题 #16995
  - 修复 Green GC 在有 Dirty Store 的情况下可能遗留锁的问题 #16949
  - 修复 Green GC 在遇到单个 key 有多个不同锁的情况下会遗留下锁的问题 #16948
  - 修复 INSERT VALUE 中子查询引用父查询列导致插入值错误的问题 #16952
  - 修复对 Float 值进行 AND 操作结果不正的问题 #16666
  - 修复 Expensive Log 日志中 WAIT_TIME 字段信息错误的问题 #16907
  - 修复悲观事务模式执行语句 SELECT FOR UPDATE 不能被记录到 Slow Log 的问题 #16897
  - 修复在 Enum 或 Set 类型列上执行 SELECT DISTINCT 时结果错误的问题 #16892
  - 修复 auto_random_base 在 SHOW CREATE TABLE 的显示问题 #16864
  - 修复 WHERE string_value 结果不正确的问题 #16559
  - 修复 GROUP BY Window Function 错误消息和 MySQL 不一致问题 #16165
  - 修复 FLASH TABLE 语句在数据库名有大写字母时执行失败的问题 #17167
  - 修复 Projection 执行器内存消耗记录不准确的问题 #17118
  - 修复 SLOW_QUERY 表在不同时区下时间过滤不正确的问题 #17164
  - 修复 IndexMerge 和虚拟生成列一起使用时 Panic 的问题 #17126
  - 修复 INSTR 和 LOCATE 函数大小写问题 #17068
  - 修复开启 tidb_allow_batch_cop 配置后频繁出现 tikv server timeout 错误的问题 #17161
  - 修复 Float 类型进行 XOR 操作结果和 MySQL 8.0 不一致的问题 #16978
  - 修复 ALTER TABLE REORGANIZE PARTITION 不支持但执行未报错的问题 #17178
  - 修复 EXPLAIN FORMAT="dot" FOR CONNECTION ID 可能遇到不支持展示的 Plan 发生报错的问题 #17160
  - 修复 Prepared Statement 在 Statement Summary 表中 EXEC_COUNT 列的记录问题 #17086
  - 修复设置 Statement Summary 系统变量时未检查值是否合法的问题 #17129
  - 修复启用 Plan Cache 时使用越界值查询 UNSIGNED BIGINT 主键报错的问题 #17120
  - 修复 Grafana TiDB Summary 面板基于机器实例和请求类型展示 QPS 不正确的问题 #17105

- TiKV
  - 修复 restore 后生成大量空 Region 的问题 #7632
  - 修复 Raftstore 收到乱序 read index 响应时会引发 panic 的问题 #7370
  - 修复启用统一线程池时，不会验证 storage 或 `coprocessor read pool` 配置是否无效的问题 #7513
  - 修复 TiKV server 关闭时，join 可能 panic 的问题 #7713
  - 修复通过诊断 API 搜索慢日志无返回结果的问题 #7776
  - 修复节点长时间运行时，系统会产生较多内存碎片的问题 #7556
  - 修复部分情况下因存储无效日期导致 SQL 执行失败的问题 #7268
  - 修复从 GCS 进行恢复数据不能正确运行的问题 #7739
  - 修复存储加密时未进行 KMS Key Id 验证的问题 #7719
  - 修复 Coprocessor 在不同架构编译器下潜在正确性问题 #7714 #7730
  - 修复启用加密时出现 `snapshot ingestion` 错误的问题 #7815
  - 修复当改写配置文件时发生 `Invalid cross-device link` 的问题 #7817
  - 修复将配置文件写入到空文件中时会出现错误的 toml 格式的问题 #7817
  - 修复 Raftstore 中已销毁的 Peer 仍然可能处理请求的问题 #7836
- PD
  - 修复 pd-ctl 中使用 `region key` 命令时发生 404 错误的问题 #2399
  - 修复 Grafana 面板中缺失关于 TSO 和 ID 分配的监控的问题 #2405
  - 修复 Docker 镜像中不包含 `pd-recover` 的问题 #2406
  - 将数据目录的路径解析为绝对路径以解决 TiDB Dashboard 可能不能正确显示 PD 信息的问题 #2420
  - 修复在 pd-ctl 中使用 `scheduler config shuffle-region-scheduler` 命令时没有默认输出的问题 #2416
- TiFlash
  - 修复部分场景错误上报已使用空间信息的问题
- Tools
  - TiDB Binlog
    * 修复当下游为 Kafka 时对 `mediumint` 类型数据未处理的问题 #962
    * 修复当 DDL 中库名为关键字时 reparo 解析失败的问题 #961
  - TiCDC
    * 修复当环境变量 TZ 未设置时使用错误时区的问题 #512
    * 修复 owner 因为部分错误没有正确处理导致 server 退出时没有清理资源的问题 #528
    * 修复重连 TiKV 可能导致 TiCDC 阻塞的问题 #531
    * 优化初始化表结构时的内存使用 #534
    * 使用 watch 模式监听同步状态变更并进行准实时更新，减少同步延迟 #481
  - Backup & Restore (BR)
    * 修复 BR 恢复带有 `auto_random` 属性的表之后，插入数据有一定概率触发 duplicate entry 错误的问题 #241

14.2.19 TiDB 4.0 RC.1 Release Notes

发版日期：2020 年 4 月 28 日

TiDB 版本：4.0.0-rc.1

14.2.19.1 兼容性变化

- TiKV
  - 默认关闭 hibernate region #7618
- TiDB Binlog
  - Drainer 增加对 Sequence DDL 的支持 #950

14.2.19.2 重点修复的 Bug

- TiDB
  - 修复由于未检查 `MemBuffer`，事务内执行 `INSERT ... ON DUPLICATE KEY UPDATE` 语句插入多行重复数据可能出错的问题 #16689
  - 修复 lock 多行重复 keys 时导致数据索引不一致的问题 #16769
  - 修复回收空闲 gRPC 连接导致 TiDB panic 的问题 #16303
- TiKV
  - 修复 TiDB 探活请求触发的死锁问题 #7540
  - 修复事务的 min commit ts 可能溢出、影响数据正确性的问题 #7642
- TiFlash
  - 修正多数据路径配置下进行 rename table 会导致数据丢失的问题
  - 修正当 Region 处于 merge 状态时读取产生的数据错误
  - 修正当 Region 处于非 normal 状态时读取产生的数据错误
  - 修正 TiFlash 中表名的映射方式以正确支持 `recover table/flashback table`
  - 修正数据存储路径以解决 `rename table` 时潜在的数据丢失问题
  - 修正 Super batch 开启后，有一定概率 TiDB panic 的问题
  - 修正在线更新时的读模型以优化读性能
- TiCDC
  - 修复 TiCDC 内部维护的 schema 信息对于读写操作时序问题没有正确处理导致同步失败问题 #438 #450 #478 #496
  - 修复 TiKV client 遇到部分 TiKV 异常没有正确维护内部资源的 bug #499 #492
  - 修复节点异常残留元数据信息没有正确清理的 bug #488 #504
  - 修复 TiKV client 没有正确处理 prewrite 重复发送的问题 #446
  - 修复 TiKV client 没有正确处理在初始化前接收到冗余 prewrite 的问题 #448

- Backup & Restore (BR)
  - 修复关闭 checksum 情况下，仍然执行 checksum 的问题 #223
  - 修复 TiDB 开启 auto-random 或 alter-pk 时，增量备份失败的问题 #230 #231

14.2.19.3 新功能

- TiDB
  - 支持发送 `batch coprocessor` 请求给 TiFlash #16226
  - 默认打开 Coprocessor cache #16710
  - 在 SQL 语句的特殊注释中，只有被注册了语句片段才能被 parser 正常解析，否则将被忽略 #16157
  - 支持使用 `SHOW CONFIG` 语法显示 PD 和 TiKV 的配置 #16475
- TiKV
  - 支持在备份到 S3 时使用用户提供的 KMS key 进行服务端加密 #7630
  - 支持基于负载的 Region split #7623
  - 支持 common name 验证 #7468
  - 通过检查文件锁避免多个 TiKV 实例绑定相同的地址 #7447
  - Encryption at rest 支持 AWS KMS #7465
- Placement Driver (PD)
  - 移除 `config manager` 以使其它组件自行控制它们的配置 #2349
- TiFlash
  - 添加 DeltaTree 引擎读写负载相关 metrics 上报
  - 缓存 `handle` 列和 `version` 列减小单次读请求的磁盘 I/O
  - 增加了 TiFlash 对于 `FromUnixTime` 和 `Date_format` 函数的支持
  - 根据第一块盘估算全局状态并上报
  - Grafana 添加 DeltaTree 引擎读写负载相关图表
  - 优化 TiFlash chunk encode decimal 的数据
  - 实现了 Diagnostics（SQL 诊断）的 gRPC API，以支持 `INFORMATION_SCHEMA.CLUSTER_INFO` 等系统表的查询
- TiCDC
  - 在 Kafka sink 模块提供发送批量消息支持 #426
  - 支持在 processor 内使用文件排序 #477
  - 增加自动 resolve lock 的支持 #459
  - 增加自动向 PD 设置 TiCDC 服务级别 safepoint 的功能 #487
  - 增加数据同步时的时区配置 #498
- Backup & Restore (BR)
  - 支持在存储的 URL 中配置 S3/GCS #246

14.2.19.4 Bug 修复

- TiDB
  - 修复系统表由于 unsigned 列定义导致无法正确显示负数的问题 #16004
  - 当使用 use_index_merge hint 包含不存在的索引时添加警告 #15960
  - 禁止多个 TiDB server 共享同一个临时目录 #16026
  - 修复打开 plan cache 时，执行 explain for connection 语句 panic 的问题 #16285
  - 修复显示 tidb_capture_plan_baselines 系统变量不正确的问题 #16048
  - 修复 prepare 语句中的 group by 语句解析错误的问题 #16377
  - 修复 analyze primary key 可能 panic 的问题 #16081
  - 修复 cluster_info 系统表中 TiFlash 节点信息错误的问题 #16024
  - 修复 index merge 可能 panic 的问题 #16360
  - 修复 index merge 遇到 generated column 时导致结果错误的问题 #16359
  - 修复 show create table 语句显示 sequence 默认值错误的问题 #16526
  - 修复主键使用 sequence 作为默认值时出现 not-null 的错误的问题 #16510
  - 修复当 TiKV 持续返回 StaleCommand 期间执行 SQL 卡住不报错的问题 #16530
  - 修复 CREATE DATABASE 时仅指定 COLLATE 时会报错的问题，同时在 SHOW CREATE DATABASE 结果中添加缺失的 COLLATE 部分 #16540
  - 修复打开 plan-cache 时，分区裁剪失败的问题 #16723
  - 修复点查在 handle 溢出时返回错误结果的问题 #16755
  - 修复等值时间查询 slow_query 系统表返回错误结果的问题 #16806
- TiKV
  - 解决 OpenSSL 的安全性问题：CVE-2020-1967 #7622
  - 避免将 BatchRollback 产生的 rollback 记录标为保护的记录以改善冲突较多场景下的性能 #7604
  - 修复锁竞争严重的场景下，不必要地唤醒事务导致多余的重试和性能下降的问题 #7551
  - 修复多次 merge 时，Region 可能卡住的问题 #7518
  - 修复删除 learner 时，learner 可能并未被删除的问题 #7518
  - 修复 follower read 可能使 raft-rs panic 的问题 #7408
  - 修复 SQL 可能因 “group by constant” 错误失败的问题 #7383
  - 修复当一个乐观锁对应的 primary lock 是悲观锁时，该乐观锁可能阻塞读的问题 #7328
- Placement Driver (PD)
  - 修复部分 API TLS 认证失败的问题 #2363
  - 修复 config API 不能识别带有前缀的配置项的问题 #2354
  - 修复找不到 scheduler 时会返回 500 错误的问题 #2328
  - 修复 scheduler config balance-hot-region-scheduler list 命令返回 404 错误的问题 #2321
- TiFlash
  - 禁用存储引擎的粗糙索引优化
  - 修正对 Region 进行 resolve lock 时遇到需要跳过的 lock 抛异常的问题
  - 修正 Coprocessor 统计信息收集的 NPE
  - 修正 Region meta 的检查以保证 Region Split/Region Merge 流程的正确性
  - 修正对 Coprocessor response 大小未做估算导致的消息大小超过 gRPC 限制的问题
  - 修正 TiFlash 对 AdminCmdType::Split 命令的处理

14.2.20 TiDB 4.0 RC Release Notes

发版日期：2020 年 4 月 8 日

TiDB 版本：4.0.0-rc

TiUP 版本：0.0.3

> **警告：**
>
> 该版本存在一些已知问题，已在新版本中修复，建议使用 4.0.x 的最新版本。

14.2.20.1 兼容性变化

- TiDB
  - 当 tidb-server 状态端口被占用时由原来打印一条告警日志改成拒绝启动 #15177
- TiKV
  - 悲观事务支持 pipelined 功能，TPC-C 性能提升 20%，风险：pipelined 功能可能会在执行阶段加锁不成功导致事务提交失败 #6984
  - 调整 unify-read-pool 配置项的方式，仅在新部署的集群时默认启用，旧集群保持原来的方式 #7059
- Tools
  - TiDB Binlog
    * 新增验证 Common Name 配置项目的功能 #934

14.2.20.2 重点修复的 Bug

- TiDB
  - 修复 DDL 采用 PREPARE 语句执行时，由于内部记录的 job query 不正确，导致上下游同步可能出错的问题 #15435
  - 修复 Read Committed 隔离级别下，子查询的输出结果可能不正确的问题 #15471
  - 修复 Inline Projection 优化所导致的结果错误问题 #15411
  - 修复某些情况下 SQL Hint `INL_MERGE_JOIN` 未正确执行的问题 #15515
  - 修复向 `AutoRandom` 列显式写入负数时，`AutoRandom` 列会 Rebase 的问题 #15397

14.2.20.3 新功能

- TiDB
  - 新增大小写不敏感的排序规则，用户可在新集群上启用 `utf8mb4_general_ci` 和 `utf8_general_ci` #33

  - 增强 RECOVER TABLE 语法，现在该语法支持恢复被 Truncate 的表 #15398
  - 当 tidb-server 状态端口被占用时由原来打印一条告警日志改成拒绝启动 #15177
  - 优化使用 Sequence 作为列的默认值时的写入性能 #15216
  - 新增 DDL Jobs 系统表，用于查询 DDL 任务详细信息 #14837
  - 优化 aggFuncSum 的性能 #14887
  - 优化 EXPLAIN 的输出结果 #15507

- TiKV

  - 悲观事务支持 pipelined 功能，TPC-C 性能提升 20%，风险：pipelined 功能可能会在执行阶段加锁不成功导致事务提交失败 #6984
  - HTTP 端口支持 TLS #5393
  - 调整 unify-read-pool 配置项的方式，仅在新部署的集群时默认启用，旧集群保持原来的方式 #7059

- PD

  - 新增通过 HTTP 接口获取 PD 默认配置信息功能 #2258

- Tools

  - TiDB Binlog

    * 新增验证 Common Name 配置项目的功能 #934

  - TiDB Lightning

    * 优化 TiDB Lightning 的性能 #281 #275

14.2.20.4 Bug 修复

- TiDB

  - 修复 DDL 采用 PREPARE 语句执行时，由于内部记录的 job query 不正确，导致上下游同步可能出错的问题 #15435
  - 修复 Read Committed 隔离级别下，子查询的输出结果可能不正确的问题 #15471
  - 修复 INSERT ... VALUES 指定 BIT(N) 类型数据时可能报错的问题 #15350
  - 修复 DDL Job 内部重试时，ErrorCount 的值没有被正确累加导致未完全达到重试预期的问题 #15373
  - 修复 TiDB 连接 TiFlash 时，垃圾回收可能工作不正常的问题 #15505
  - 修复 Inline Projection 优化所导致的结果错误问题 #15411
  - 修复某些情况下 SQL Hint INL_MERGE_JOIN 未正确执行的问题 #15515
  - 修复向 AutoRandom 列显式写入负数时，AutoRandom 列会 Rebase 的问题 #15397

- TiKV

  - 修复启用 Follower Read 功能，由于 transfer leader 导致系统 Panic 的问题 #7101

- Tools

  - TiDB Lightning

    * 修复 backend 是 TiDB 时由于字符转换错误导致数据错误的问题 #283

  - TiCDC
    * 修复 MySQL sink 执行 DDL 时，若下游没有 test 库系统报错的问题 #353
    * CDC cli 新增实时交互模式功能 #351
    * 同步数据时增加对上游表是否可同步的检查 #368
    * 新增异步写入 Kafka 的功能 #344

14.2.21 TiDB 4.0.0 Beta.2 Release Notes

发版日期：2020 年 3 月 18 日

TiDB 版本：4.0.0-beta.2

TiDB Ansible 版本：4.0.0-beta.2

14.2.21.1 兼容性变化

- Tools
  - TiDB Binlog
    * 修复 Drainer 配置 `disable-dispatch`、`disable-causality` 时系统直接报错并退出的问题 #915

14.2.21.2 新功能

- TiKV
  - 支持将动态修改配置的结果持久化存储到硬盘 #6684
- PD
  - 支持将动态修改配置的结果持久化存储到硬盘 #2153
- Tools
  - TiDB Binlog
    * 新增 TiDB 集群之间数据双向复制功能 #879 #903
  - TiDB Lightning
    * 新增配置 TLS 功能 #40 #270
  - 新增 TiCDC 工具，提供以下功能：
    * 捕捉 TiKV 变化的数据，同步到下游 Kafka、MySQL 协议的数据库
    * 确保数据最终一致性，若下游是 Kafka，也可确保行级别的有序
    * 提供进程级别的高可用能力
  - BR
    * 开启增量备份、支持将备份文件存储在 AWS S3 等实验性功能 #175
- TiDB Ansible
  - 新增将节点信息注册到 etcd 的功能 #1196
  - 新增支持在 ARM 平台上部署 TiDB 服务的功能 #1204

14.2.21.3 Bug 修复

- TiKV
  - 修复 backup 在遇到空的 short value 时可能 panic 的问题 #6718
  - 修复 Hibernate Region 在某些特殊条件下未被正确唤醒的问题 #6772 #6648 #6376
- PD
  - 修复因 rule checker 在给 Region 分配 store 失败导致系统 panic 的问题 #2160
  - 修复启用动态修改配置功能后，配置可能在切换 leader 时有同步延迟的问题 #2154
- Tools
  - BR
    * 修复因 PD 无法处理过大消息导致在数据规模较大时恢复失败的问题 #167
    * 修复因 BR 与 TiDB 版本不兼容导致 BR 运行失败的问题 #186
    * 修复因 BR 与 TiFlash 不兼容导致 BR 运行失败的问题 #194

14.2.22 TiDB 4.0.0 Beta.1 Release Notes

发版日期：2020 年 2 月 28 日

TiDB 版本：4.0.0-beta.1

TiDB Ansible 版本：4.0.0-beta.1

14.2.22.1 兼容性变化

- TiDB
  - 修改配置项 `log.enable-slow-log` 的类型，由整数型改为布尔类型 #14864
  - 调整修改系统表 `mysql.user` 中 `password` 列名为 `authentication_string`，与 MySQL 5.7 保持一致（该变动会导致升级后不能回退）#14598
  - `txn-total-size-limit` 配置项的默认值由 1GB 调整为 100MB #14522
  - 新增动态修改、更新配置项的功能，配置项由 PD 持久化存储 #14750 #14303 #14830
- TiKV
  - 新增 `readpool.unify-read-pool` 配置项，默认值为 `True`，用于控制点查是否共用 Coprocessor 的处理线程 #6375 #6401 #6534 #6582 #6585 #6593 #6597 #6677
- PD
  - 优化 HTTP API 兼容以新的配置项管理方式 #2080
- TiDB Lightning
  - 优化配置项，部分配置项在没有设置的时候使用默认配置 #255
- TiDB Ansible
  - 将 `theflash` 更名为 `tiflash` #1130
  - 优化 TiFlash 配置文件中的默认值及相关配置 #1138

14.2.22.2 新功能

- TiDB
  - 慢日志系统表 SLOW_QUERY / CLUSTER_SLOW_QUERY 支持查询任意时间段的日志#14840 #14878
  - 支持 SQL 性能诊断功能
    * #14843 #14810 #14835 #14801 #14743
    * #14718 #14721 #14670 #14663 #14668
    * #14896
  - 新增 Sequence 功能 #14731 #14589 #14674 #14442
  - 新增动态修改、更新配置项的功能，配置项由 PD 持久化存储 #14750 #14303 #14830
  - 新增系统自动根据负载均衡策略从不同角色上读取数据的功能，且新增 `leader-and-follower` 系统变量用于控制开启此功能 #14761
  - 新增 `Coercibility` 函数 #14739
  - 支持在分区表上建立 TiFlash 副本 #14735 #14713 #14644
  - 完善 SLOW_QUERY 表的权限检查 #14451
  - 新增当 `Join` 时若内存不足时系统自动将中间结果写磁盘文件的功能 #14708 #14279
  - 新增通过查询 `information_schema.PARTITIONS` 系统表查看 partition 详细信息的功能 #14347
  - 新增 `json_objectagg` 聚合函数 #11154
  - 新增审计日志记录用户登录失败的功能 #14594
  - 新增 `max-server-connections` 配置项，默认值为 `4096` ，用于控制单个服务器连接数 #14409
  - 支持隔离读在 Server 级别指定多个存储引擎 #14440
  - 优化 `Apply` 算子和 `Sort` 算子的代价估算模型，提升系统稳定性 #13550 #14708
- TiKV
  - 新增通过 HTTP API 从 status 端口获取配置项的功能 #6480
  - 提升 Coprocessor 的 Chunk Encoder 的性能 #6341
- PD
  - 新增通过 UI 访问集群热点数据分布功能 #2086
  - 新增收集集群组件的启动时间的功能 #2116
  - member API 返回信息新增部署路径和组件版本信息 #2130
  - pd-ctl 新增 `component` 命令用于修改、查看组件配置信息 (experimental) #2092
- TiDB Binlog
  - 同步链路新增 TLS 功能 #904 #894
  - Drainer 新增 `kafka-client-id` 配置项，支持连接 Kafka 客户端配置客户端 ID #902
  - Drainer 新增清理增量备份文件的功能 #885
- TiDB Ansible
  - 新增同一个集群中部署多个 Grafana/Prometheus/Alertmanager 的功能 #1142
  - TiFlash 配置文件新增 `metrics_port` 配置项，默认值为8234 #1145
  - TiFlash 配置文件新增 `flash_proxy_status_port` 配置项，默认值为 20292 #1141
  - 新增 TiFlash 监控 Dashboard #1147 #1151

14.2.22.3 Bug 修复

- TiDB
  - 修复创建视图时，列名超过 64 个字符时将报错的问题，报错改为重命名过长的列名 #14850
  - 修复因 `create or replace view` 语句处理逻辑不正确导致 `information_schema.views` 中有重复数据的问题 #14832
  - 修复开启 `plan cache` 之后，`BatchPointGet` 的获取到错误数据的问题 #14855
  - 修复按照时间分区的分区表，在修改时区后，因处理逻辑不正确导致数据插入到错误分区表的问题 #14370
  - 修复 `IsTrue` 函数的表达式因名称不正确，在执行外连化简利用非法函数名重建表达式导致系统 panic 的问题 #14515
  - 修复 `show binding` 语句权限检查不正确的问题 #14443
- TiKV
  - 修复 CAST 函数在 TiDB 和 TiKV 中行为不一致性的问题 #6463 #6461 #6459 #6474 #6492 #6569
- TiDB Lightning
  - 修复在非 server mode 模式下 web 界面无法打开的问题 #259

14.2.23 TiDB 4.0 Beta Release Notes

发版日期：2020 年 1 月 17 日

TiDB 版本：4.0.0-beta

TiDB Ansible 版本：4.0.0-beta

14.2.23.1 TiDB

- 当 `Insert/Replace/Delete/Update` 在执行过程中所使用的内存空间超过启动配置项 `MemQuotaQuery` 的限制时，输出日志或取消本次执行过程，具体行为取决于启动配置项 `OOMAction` #14179 #14289 #14299
- 估算 `Index Join` 的代价时由仅考虑驱动表的行数调整为考虑驱动表和被驱动表的行数，提升估算的准确性 #12085
- 新增 15 个 SQL hint，用于控制优化器行为，提升优化器稳定性
  - #11253 #11364 #11673 #11740 #11746
  - #11809 #11996 #12043 #12059 #12246
  - #12382
- 提升查询中所涉及到的列能被索引全覆盖时的性能 #12022
- 对表上的 OR 表达式过滤条件，使用多个索引组合进行表访问，提升查询性能 #10121 #10512 #11245 #12225 #12248 #12305 #12843
- 优化 Range 计算流程，缓存并去重索引计算的结果，减少 CPU 开销，提升 range 计算的性能 #12856
- Slow Log 日志的级别与普通日志的级别解耦 #12359
- 新增 `oom-use-tmp-storage` 参数，默认值为 `true`，用于控制当单条 SQL 执行过程中占用内存使用超过 `mem-quota-query` 且 SQL 中包含 `Hash Join` 时，系统会采用临时文件来缓存中间结果 #11832 #11937 #12116 #12067

- 支持使用 `create index/alter table add index` 语句创建表达式索引，使用 `drop index` 语句删除表达式索引 #14117
- `query-log-max-len` 参数默认值调大为 `4096`，减少输出被截断 SQL 的数量，此参数可通过 SQL 动态调整 #12491
- 支持在列属性上添加 `AutoRandom` 关键字，用于控制系统自动为主键分配随机整数，避免 `AutoIncrement` 自增主键带来的写入热点问题 #13127
- 支持表级锁 (Table Locks) #11038
- `ADMIN SHOW DDL JOBS` 支持 `LIKE` 或 `WHERE` 语法进行条件过滤 #12484
- `information_schema.tables` 表新增 `TIDB_ROW_ID_SHARDING_INFO` 列，输出列的 RowID 打散相关的信息 (例如：表 A 指定 `SHARD_ROW_ID_BITS`，该列的值为 `"SHARD_BITS={bit_number}"`) #13418
- 优化 SQL 错误信息的错误码，避免出现多个错误信息的错误码都是 `ERROR 1105 (HY000)` (即类型为 `Unknown Error`) 的情况
  - #14002 #13874 #13733 #13654 #13646
  - #13540 #13366 #13329 #13300 #13233
  - #13033 #12866 #14054
- 在估算行数时将离散类型的很窄的 range 转化为 `point set` 然后用 CM-Sketch 提升估算精度 #11524
- 支持普通 `Analyze` 得到的 CM-Sketch 维护 TopN 信息，将出现次数较多的值单独维护 #11409
- 支持动态调整 CM-Sketch 长宽和 TopN 数目 #11278
- 新增 SQL Binding 的自动捕获和自动演进功能 #13199 #12434
- 优化与 TiKV 之间通信息编码格式，采用 Chunk 格式编码，提升网络通信性能 #12023 #12536 #12613 #12621 #12899 #13060 #13349
- 支持新的行存储格式，提升宽表性能 #12634
- 优化 `Recover Binlog` 接口，确保等待当前正在提交的事务都提交完成再返回 #13740
- 新增通过 HTTP `info/all` 接口，查询集群中所有 TiDB server 开启 binlog 的状态 #13025
- 新增在事务模式是悲观事务时，支持使用 MySQL 兼容的 Read Committed 事务隔离级别 #14087
- 支持超大事务，事务大小的限制受限于物理内存大小
  - #11999 #11986 #11974 #11817 #11807
  - #12133 #12223 #12980 #13123 #13299
  - #13432 #13599
- 提升 `Kill` 稳定性 #10841
- `LOAD DATA` 支持十六进制和二进制表达式作为分隔符 #11029
- `IndexLookupJoin` 拆分为 `IndexHashJoin` 与 `IndexMergeJoin`，提升 `IndexLookupJoin` 的执行性能，减少执行过程中的内存消耗 #8861 #12139 #12349 #13238 #13451 #13714
- 修复 RBAC 若干问题 #13896 #13820 #13940 #14090 #13940 #13014
- 修复创建视图时，由于 `select` 语句包含 `union` 视图无法创建成功的问题 #12595
- 修复 `CAST` 函数若干问题
  - #12858 #11968 #11640 #11483 #11493
  - #11376 #11355 #11114 #14405 #14323
  - #13837 #13401 #13334 #12652 #12864
  - #12623 #11989
- Slow log 输出 TiKV RPC 的 `backoff` 具体信息，方便排查问题 #13770
- 优化统一 expensive log 中内存统计信息的格式 #12809

- 优化 `explain` 显式格式，支持输出算子占用内存和磁盘的信息 #13914 #13692 #13686 #11415 #13927 #13764 #13720
- 优化 `LOAD DATA` 重复值检查，按照事务粒度进行且事务大小可通过 `tidb_dml_batch_size` 配置 #11132
- 优化 `LOAD DATA` 性能，将数据读取处理和批量提交分离，且分派到不同的 Worker 处理 #11533 #11284

14.2.23.2 TiKV

- 升级 RocksDB 的版本到 6.4.6
- 系统启动时自动创建 2GB 大小的空文件，解决磁盘空间被写满时系统无法正常进行 Compaction 的问题 #6321
- 新增快速备份恢复功能
  - #6462 #6395 #6378 #6374 #6349
  - #6339 #6308 #6295 #6286 #6283
  - #6261 #6222 #6209 #6204 #6202
  - #6198 #6186 #6177 #6146 #6071
  - #6042 #5877 #5806 #5803 #5800
  - #5781 #5772 #5689 #5683
- 新增从 Follower 副本读取数据的功能
  - #5051 #5118 #5213 #5316 #5401
  - #5919 #5887 #6340 #6348 #6396
- 提升 TiDB 通过索引读取数据的性能 #5682
- 修复 CAST 函数在 TiDB 和 TiKV 中行为不一致性的问题
  - #6459 #6461 #6458 #6447 #6440
  - #6425 #6424 #6390 #5842 #5528
  - #5334 #5199 #5167 #5146 #5141
  - #4998 #5029 #5099 #5006 #5095
  - #5093 #5090 #4987 #5066 #5038
  - #4962 #4890 #4727 #6060 #5761
  - #5793 #5468 #5540 #5548 #5455
  - #5543 #5433 #5431 #5423 #5179
  - #5134 #4685 #4650 #6463

14.2.23.3 PD

- 新增根据存储节点负载信息优化热点调度的功能
  - #1870 #1982 #1998 #1843 #1750
- 新增 Placement Rules 功能，通过组合不同的调度规则，精细控制任意一段数据的副本的数量、存放位置、存储主机类型、角色等信息
  - #2051 #1999 #2042 #1917 #1904
  - #1897 #1894 #1865 #1855 #1834
- 支持插件功能 (experimental) #1799
- 新增调度器支持自定义配置功能，支持配置调度器的作用范围 (experimental) #1735 #1783 #1791
- 新增根据集群负载信息自动调整调度速度的功能（experimental，默认不打开）#1875 #1887 #1902

14.2.23.4 Tools

- TiDB Lightning
  - 命令行增加配置下游数据库密码的参数 #253

14.2.23.5 TiDB Ansible

- 下载包增加 Checksum 检查，防止下载到不完整的包 #1002
- 新增检查 systemd 版本功能，systemd 版本最低要求 `systemd-219-52` #1020 #1074
- 修复 TiDB Lightning 启动时未正确创建日志目录的问题 #1103
- 修复 TiDB Lightning 自定义端口不生效的问题 #1107
- 新增支持部署运维 TiFlash 的功能 #1119

14.3 v3.1

14.3.1 TiDB 3.1.2 Release Notes

发版日期：2020 年 6 月 4 日

TiDB 版本：3.1.2

14.3.1.1 Bug 修复

- TiKV
  - 修复 S3 和 GCS 备份恢复时的错误处理问题 #7965
  - 修复备份过程中的 `DefaultNotFound` 错误 #7838
- Tools
  - Backup & Restore (BR)
    * 提升备份恢复到 S3 和 GCS 存储的稳定性，在网络较差时会自动重试 #314 #7965
    * 修复恢复数据时因找不到 Region leader 出现的 `NotLeader` 错误，BR 会自动重试 #303
    * 修复恢复数据时 `rowID` 大于 `2^(63)` 的数据丢失问题 #323
    * 修复恢复数据时无法恢复空库空表的问题 #318
    * 增加备份恢复 S3 时的 AWS KMS 服务端加密 (SSE) 支持 #261

14.3.2 TiDB 3.1.1 Release Notes

发版日期：2020 年 4 月 30 日

TiDB 版本：3.1.1

TiDB Ansible 版本：3.1.1

14.3.2.1 新功能

- TiDB
  - 添加 `auto_rand_base` 的 table option #16812
  - 添加 `Feature ID` 注释：在 SQL 语句的特殊注释中，只有被注册了语句片段才能被 parser 正常解析，否则将被忽略 #16155
- TiFlash
  - 缓存 `handle` 列和 `version` 列减小单次读请求的磁盘 I/O
  - Grafana 添加 DeltaTree 引擎读写负载相关图表
  - 优化 TiFlash chunk encode decimal 数据的流程
  - TiFlash 低负载时，减少打开的文件描述符数量

14.3.2.2 Bug 修复

- TiDB
  - 修复实例级别的隔离读设置不生效的问题，以及 TiDB 升级后隔离读设置被不正确保留的问题 #16482 #16802
  - 修复 hash 分区表上面的分区选择语法，现在 `partition(P0)` 这样的语法不会报错 #16076
  - 修复若 update sql 中包含 view，但不会对 view 进行 update，update 语句仍然报错的问题#16789
  - 修复对查询最内层的 `not not` 消除而造成结果错误的问题 #16423
- TiFlash
  - 修复当 Region 处于非 normal 状态时读取产生的数据错误
  - 修复 TiFlash 中表名的映射方式以正确支持 `recover table/flashback table`
  - 修复数据存储路径以解决 `rename table` 时潜在的数据丢失问题
  - 修复在线更新时的读模型以优化读性能
  - 修复 database/table name 含特殊字符，升级后无法正常启动的问题
- Tools
  - Backup & Restore (BR)
    * 修复 BR 恢复带有 auto_random 属性的表之后，插入数据有一定概率触发 duplicate entry 错误的问题 #241

14.3.3 TiDB 3.1 GA Release Notes

发版日期：2020 年 4 月 16 日

TiDB 版本：3.1.0 GA

TiDB Ansible 版本：3.1.0 GA

14.3.3.1 兼容性变化

- TiDB
  - 支持 TiDB 在启动服务时，在开启 `report-status` 配置项情况下，如果发现 HTTP 监听端口不可用，则直接退出启动 #16291
- Tools
  - Backup & Restore (BR)
    * BR 不支持在 3.1 GA 版本之前的 TiKV 集群上进行恢复 #233

14.3.3.2 新功能

- TiDB
  - 支持在 `explain format = "dot"` 中展示 coprocessor 任务的信息 #16125
  - 通过 `disable-error-stack` 配置项减少日志的冗余 stack 信息 #16182
- Placement Driver (PD)
  - 优化热点 Region 调度 #2342
- TiFlash
  - 添加上报 DeltaTree 引擎读写负载相关 metrics 信息
  - 支持 `fromUnixTime` 和 `dateFormat` 函数下推
  - 默认禁用粗粒度索引过滤器
- TiDB Ansible
  - 新增 TiFlash 监控 #1253 #1257
  - 优化 TiFlash 配置参数 #1262 #1265 #1271
  - 优化 TiDB 启动脚本 #1268

14.3.3.3 Bug 修复

- TiDB
  - 修复 merge join 在某些场景下 panic 的问题 #15920
  - 修复在计算选择率时重复考虑某些表达式的问题 #16052
  - 修复极端情况下 load 统计信息可能出现的 panic 的问题 #15710
  - 修复 SQL query 中存在等价表达式在某些情况下无法识别导致报错的问题 #16015
  - 修复从一个数据库中查询另一个数据库的 `view` 时报错的问题 #15867
  - 修复 fast analyze handle 列时 panic 的问题 #16080
  - 修复 `current_role` 输出结果字符集不正确的问题 #16084
  - 完善 MySQL 连接握手错误相关日志 #15799
  - 修复加载审计插件后端口探测活动导致 panic 的问题 #16065

  - 修复因 TypeNull 类被错误识别为变长类型，导致 left join 上的 sort 算子 panic 的问题 #15739
  - 修复监控 session 重试错误计数不准确的问题 #16120
  - 修复在 ALLOW_INVALID_DATES 模式下，weekday 结果出错的问题 #16171
  - 修复在集群中存在 TiFlash 节点时，GC 可能不能正常工作的问题 #15761
  - 修复创建 hash 分区表时指定非常大的分区数量导致 TiDB OOM 的问题 #16219
  - 让 union 语句的行为和 select 语句保持相同，修复把 warnings 当 error 的问题 #16138
  - 修复 TopN 下推到 mocktikv 中的执行错误 #16200
  - 增大 chunk.column.nullBitMap 的初始化长度，以避免多余的 runtime.growslice 开销 #16142

- TiKV

  - 修复 replica read 导致 TiKV panic 的问题 #7418 #7369
  - 修复 restore 产生许多空 Region 的问题 #7419
  - 修复重复的 resolve lock 请求可能会破坏悲观事务原子性的问题 #7389

- TiFlash

  - 修复从 TiDB 同步 schema 时，进行 rename table 时潜在的问题
  - 修复多数据路径配置下进行 rename table 会导致数据丢失的问题
  - 修复某些场景下 TiFlash 存储空间上报错误的问题
  - 修复开启 Region Merge 情况下从 TiFlash 读取时潜在的问题

- Tools

  - TiDB Binlog

    * 修复因为 TiFlash 相关的 DDL job 导致 Drainer 同步中断的问题 #948 #942

  - BR

    * 修复关闭 checksum 情况下，仍然执行 checksum 的问题 #223
    * 修复 TiDB 开启 auto-random 或 alter-pk 时，增量备份失败的问题 #230 #231

14.3.4 TiDB 3.1 RC Release Notes

发版日期：2020 年 4 月 2 日

TiDB 版本：3.1.0-rc

TiDB Ansible 版本：3.1.0-rc

> **警告：**
>
> 该版本存在一些已知问题，已在新版本中修复，建议使用 3.1.x 的最新版本。

14.3.4.1 新功能

- TiDB
  - 采用的二分搜索实现分区裁剪，以来提升性能 #15678
  - 支持 RECOVER 语法恢复被 `truncate table` 删除的数据 #15460
  - 支持重用语句重试中已分配的 AUTO_RANDOM ID #15393
  - 支持 `recover table` 恢复 AUTO_RANDOM ID 分配器的状态 #15393
  - 支持 YEAR、 MONTH、 TO_DAY 函数作为 Hash partition table 的分区 key #15619
  - 只在读到数据，需要加锁的时候，才对表做 schema-change 的检查 #15708
  - 为 session 变量 `tidb_replica_read` 增加 `leader-and-follower` 值，实现读请求在 leader 和 follower 直接负载均衡 #15721
  - 支持 TiDB 在每次新建连接时动态更新 TLS 证书，实现不重启更新过期客户端证书 #15163
  - 通过更新 PD Client 支持每次新建连接是读取加载最新的证书 #15425
  - 如果配置了 `Cluster-SSL-*` 强制让 TiDB-PD 和 TiDB-TiDB 使用配置的证书进行 HTTPS 协议传输 #15430
  - 新增和 MySQL 兼容的 `--require-secure-transport` 启动项，配置时强制客户端使用 TLS #15442
  - 添加 `cluster-verify-cn` 配置，只有拥有特定 CN 属性值证书的访问者才能访问 TiDB Status Port 或建立 gRPC 连接 #15137
- TiKV
  - 支持通过 Raw KV API 备份数据 #7051
  - 状态服务支持 TLS #7142
  - KV server 支持 TLS #7305
  - 优化持有锁的时间以提升备份性能 #7202
- PD
  - `shuffle-region-scheduler` 支持调度 learner #2235
  - pd-ctl 增加配置 Placement Rules 的命令 #2306
- Tools
  - TiDB Binlog
    * 同步链路新增 TLS 功能 #931 #937 #939
    * Drainer 新增 `kafka-client-id` 配置项，支持连接 Kafka 客户端配置客户端 ID #929
  - TiDB Lightning
    * 优化 Lightning 的性能 #281 #275
    * 支持 TLS #270
  - BR
    * 优化日志输出信息，对用户更友好 #189
- TiDB Ansible
  - 优化 TiFlash 数据目录创建的方式 #1242
  - TiFlash 新增 `Write Amplification` 监控项 #1234
  - 优化 CPU epollexclusive 检查失败时提示信息，包括：通过升级内核版本解决，且提示支持的最小内核版本 #1243

14.3.4.2 Bug 修复

- TiDB
  - 修复由于 update tiflash replica 类型的 DDL 太频繁导致的 information schema changed 错误的问题 #14884
  - 修复在使用 AUTO_RANDOM 时，未正确生成的 `last_insert_id` 的问题 #15149
  - 修复更新 TiFlash replica 状态时可能导致 DDL 卡住的问题 #15161
  - 当存在谓词无法下推时，禁止聚合下推和 TopN 下推 #15141
  - 禁止相互嵌套地创建 `view` #15440
  - 修复 `set role all` 后执行 `select current_role` 报错的问题 #15570
  - 修复查询中指定列的 `view` 名时，报不识别 `view` 的问题 #15573
  - 修复预处理 DDL 语句在写 binlog 信息时可能出错的问题 #15444
  - 修复同时访问视图和分区表时导致 panic 的问题 #15560
  - 修复 `update duplicate key` 语句中 `bit(n)` 类型的 column 报错的问题 #15487
  - 修复 `max-execution-time` 部分场景下不生效的问题 #15616
  - 修复在生成 Index 计划时未判断当前的 ReadEngine 中是否包含 TiKV 的问题 #15773
- TiKV
  - 修复在关闭一致性检查参数时，事务中插入已存在的 Key 且立马删除导致冲突检测失效或数据索引不一致的问题 #7112
  - 修复 TopN 比较无符号整型时计算错误的问题 #7199
  - Raftstore 引入流控机制，解决没有流控可能导致追日志太慢可能导致集群卡住，以及事务大小太大会导致 TiKV 间连接频繁重连的问题 #7087 #7078
  - 修复发送到 replicas 的读请求可能被永久卡住的问题 #6543
  - 修复 replica read 会被 apply snapshot 阻塞的问题 #7249
  - 修复 read index 在 transfer leader 情况下可能导致 panic 的问题 #7240
  - 修复备份到 S3 时所有 SST 文件填充为零的问题 #6967
  - 修复备份时未记录 SST 文件大小的导致恢复后有很多空 Region 的问题 #6983
  - 备份支持 AWS IAM web identity #7297
- PD
  - 修复 PD 因处理 Region heartbeat 时的数据竞争导致 Region 信息不正确的问题 #2234
  - 修复 `random-merge-scheduler` 未遵守 location labels 和 Placement Rules 规则的问题 #2212
  - 修复 Placement Rule 被具有相同 `startKey` 和 endKey 的 Placement Rule 覆盖的问题 #2222
  - 修复 API 输出的版本号与 PD server 输出版本号不一致的问题 #2192
- Tools
  - TiDB Lightning
    - 修复 backend 是 TiDB 时由于字符转化错误导致数据错误的问题 #283
  - BR
    - 修复了在开启 TiFlash 集群中，无法使用 BR 恢复的问题 #194

14.3.5 TiDB 3.1 Beta.2 Release Notes

发版日期：2020 年 3 月 9 日

TiDB 版本：3.1.0-beta.2

TiDB Ansible 版本：3.1.0-beta.2

> **警告：**
>
> 该版本存在一些已知问题，已在新版本中修复，建议使用 3.1.x 的最新版本。

14.3.5.1 兼容性变化

- Tools
  - TiDB Lightning
    - 优化配置项，部分配置项在没有进行配置的时候使用TiDB Lightning 配置参数中的默认配置 #255
    - 新增 `--tidb-password` 命令行参数，用于设置 TiDB 的密码 #253

14.3.5.2 新功能

- TiDB
  - 支持在列属性上添加 `AutoRandom` 关键字，控制系统自动为主键分配随机整数，避免 `AUTO_INCREMENT` 自增主键带来的写入热点问题 #14555
  - 新增通过 DDL 语句为表创建、删除列存储副本的功能 #14537
  - 新增优化器可自主选择不同的存储引擎的功能 #14537
  - 新增 SQL Hint 支持不同的存储引擎的功能 #14537
  - 新增通过 `tidb_replic_read` 系统变量从 Follower 上读取数据的功能 #13464
- TiKV
  - Raftstore
    - 新增 `peer_address` 参数，为其他类型的服务提供通过不同端连接此 TiKV server 的能力 #6491
    - 新增 `read_index` 和 `read_index_resp` 监控项，用于监控 `ReadIndex` 请求数 #6610
  - PD Client
    - 新增将本地线程统计信息汇报给 PD 的功能 #6605
  - Backup
    - 用 Rust 的 `async-speed-limit` 流控库替代 `RocksIOLimiter` 流控库，避免备份时拷贝多次内存的问题 #6462
- PD
  - 新增 location label 的名字中允许使用斜杠 `/` 的功能 #2084
- TiFlash

- 初始版本

- TiDB Ansible
  - 新增同一个集群中部署多个 Grafana/Prometheus/Alertmanager 的功能 #1143
  - 新增部署 TiFlash 组件的功能 #1148
  - 新增 TiFlash 组件相关的监控指标 #1152

14.3.5.3 Bug 修复

- TiKV
  - Raftstore
    * 修复静默 Region 读数据处理不当导致无法处理读请求的问题 #6450
    * 修复 ReadIndex 在 leader 切换时可能导致系统 panic 的问题 #6613
    * 修复 Hibernate Region 在某些特殊条件下未被正确唤醒的问题 #6730 #6737 #6972
  - Backup
    * 修复备份数据时备份了多余的数据，导致恢复数据时数据索引不一致的问题 #6659
    * 修复备份时因处理已被删除的值逻辑不正确导致系统 panic 的问题 #6726
- PD
  - 修复因 rule checker 给 Region 分配 store 失败导致系统 panic 的问题 #2161
- Tools
  - TiDB Lightning
    * 修复在非 Server mode 模式下 web 界面无法打开的问题 #259
  - BR
    * 修复在恢复数据过程中遇到不可恢复的错误时，程序无法及时退出的问题 #152
- TiDB Ansible
  - 修复在某些场景下获取不到 PD Leader 导致滚动升级命令执行失败的问题 #1122

14.3.6 TiDB 3.1 Beta.1 Release Notes

发版日期：2020 年 1 月 10 日

TiDB 版本：3.1.0-beta.1

TiDB Ansible 版本：3.1.0-beta.1

14.3.6.1 TiKV

- backup
  - 备份文件的名称由 `start_key` 改为 `start_key` 的 hash 值，减少文件名的长度，方便阅读 #6198
  - 关闭 RocksDB `force_consistency_checks` 检查功能，避免一致性检查误报的问题 #6249
  - 新增增量备份功能 #6286

- sst_importer
  - 修复恢复后 SST 文件没有 MVCC Properties 的问题 #6378
  - 新增 `tikv_import_download_duration`、`tikv_import_download_bytes`、`tikv_import_ingest_duration` ↪ 、`tikv_import_ingest_bytes`、`tikv_import_error_counter` 等监控项，用于观察 Download SST 和 Ingest SST 的开销 #6404
- raftstore
  - 修复因 Follower Read 在 leader 变更时读到旧数据的问题，导致事务的隔离性被破坏的问题 #6343

14.3.6.2 Tools

- BR (Backup and Restore)
  - 修复备份进度信息不准确的问题 #127
  - 提升 split Region 的性能 #122
  - 新增备份恢复分区表的功能 #137
  - 新增自动调度 PD schedulers 功能 #123
  - 修复非 PKIsHandle 表恢复后数据覆盖的问题 #139

14.3.6.3 TiDB Ansible

- 新增初始化阶段自动关闭操作系统 THP 的功能 #1086
- 新增 BR 组件的 Grafana 监控 #1093
- 优化 TiDB Lightning 部署，自动创建相关目录 #1104

14.3.7 TiDB 3.1 Beta Release Notes

发版日期：2019 年 12 月 20 日

TiDB 版本：3.1.0-beta

TiDB Ansible 版本：3.1.0-beta

14.3.7.1 TiDB

- SQL 优化器
  - 丰富 SQL hint #12192
- 新功能
  - TiDB 支持 Follower Read 功能 #12535

14.3.7.2 TiKV

- 支持分布式备份恢复功能 #5532
- TiKV 支持 Follower Read 功能 #5562

14.3.7.3 PD

- 支持分布式备份恢复功能 #1896

14.4 v3.0

14.4.1 TiDB 3.0.20 Release Notes

发版日期：2020 年 12 月 25 日

TiDB 版本：3.0.20

14.4.1.1 兼容性更改

- TiDB
  - 废弃配置文件中的 `enable-streaming` 配置项 #21054

14.4.1.2 改进提升

- TiDB
  - 优化 `LOAD DATA` 语句执行 `PREPARE` 时的报错信息 #21222
- TiKV
  - 增加 `end_point_slow_log_threshold` 配置 #9145

14.4.1.3 Bug 修复

- TiDB
  - 修复错误缓存悲观事务提交状态的问题 #21706
  - 修复当查询 `INFORMATION_SCHEMA.TIDB_HOT_REGIONS` 时，统计信息不准确的问题 #21319
  - 修复了一处数据库名大小写处理不当，导致的 `DELETE` 未正确删除数据的问题 #21205
  - 修复创建递归的视图出现栈溢出的问题 #21000
  - 修复 TiKV 客户端 goroutine 泄漏的问题 #20863
  - 修复 `year` 类型默认值为 0 的问题 #20828
  - 修复 Index Lookup Join 的 goroutine 泄漏问题 #20791
  - 修复在悲观事务中执行 `INSERT SELECT FOR UPDATE` 后客户端收到 malformed packet 的问题 #20681
  - 修复 `'posixrules'` 错误时区的问题 #20605
  - 修复将无符号整型转换为 bit 类型时出现的错误 #20362
  - 修复 bit 列类型默认值错误的问题 #20339
  - 修复当等值条件中有 `Enum` 和 `Set` 类型时结果可能错误的问题 #20296
  - 修复 `!= any()` 时的错误问题 #20061
  - 修复类型转换在 `BETWEEN...AND...` 中会遇到结果错误的问题 #21503

- 修复 ADDDATE 函数兼容性的问题 #21008
- 为新增的 Enum 列设置正确的默认值 #20999
- 修复 `SELECT DATE_ADD('2007-03-28 22:08:28',INTERVAL "-2.-2" SECOND)` 这类 SQL 语句的结果问题，使之兼容 MySQL #20627
- 修复当修改列属性时，默认类型设置错误的问题 #20532
- 修复 `timestamp` 函数的参数为 `float` 和 `decimal` 时，结果错误的问题 #20469
- 修复统计信息可能会死锁的问题 #20424
- 修复溢出的 Float 类型数据被 `INSERT` 的问题 #20251

- TiKV

  - 修复当事务删除 key 时却报 key 已存在的问题 #8931

- PD

  - 修复当 stale Region 过多时，启动 PD 会打印过量日志的问题 #3064

14.4.2 TiDB 3.0.19 Release Notes

发版日期：2020 年 9 月 25 日

TiDB 版本：3.0.19

14.4.2.1 兼容性变化

- PD

  - 更改 PD 的导入路径 `pingcap/pd` 为 `tikv/pd` #2779
  - 更改 PD 的 copyright 信息 `PingCAP, Inc` 为 `TiKV Project Authors` #2777

14.4.2.2 提升改进

- TiDB

  - 缓解故障恢复对 QPS 的影响 #19764
  - 支持调整 union 运算符的并发数 #19885

- TiKV

  - 永久开启 `sync-log` #8636

- PD

  - 添加关于 PD 重启的告警规则 #2789

14.4.2.3 Bug 修复

- TiDB
  - 修复 `slow-log` 文件不存在导致查询出错的问题 #20050
  - 添加对 `SHOW STATS_META` 和 `SHOW STATS_BUCKET` 这两个命令的权限检查 #19759
  - 禁止将 Decimal 类型改成 Integer 类型 #19681
  - 修复更改 `ENUM/SET` 类型的列时没有检查限制的问题 #20045
  - 修复 tidb-server 在 panic 后没有释放 table lock 的问题 #20021
  - 修复 `OR` 运算符在 `WHERE` 子句中没有正确处理的问题 #19901
- TiKV
  - 修复 TiKV 的 status server 解析响应出错导致 panic 的问题 #8540
- Tools
  - TiDB Lightning
    * 修复了严格模式下 CSV 中遇到不合法 UTF 字符集没有及时退出进程的问题 #378

14.4.3 TiDB 3.0.18 Release Notes

发版日期：2020 年 8 月 21 日

TiDB 版本：3.0.18

14.4.3.1 提升改进

- Tools
  - TiDB Binlog
    * 支持更加细粒度的 Pump GC 时间 #996

14.4.3.2 Bug 修复

- TiDB
  - 修复 Hash 函数对 `Decimal` 类型的错误处理导致 HashJoin 结果错误的问题 #19185
  - 修复 Hash 函数对 `Set` 和 `Enum` 类型的错误处理导致 HashJoin 结果错误的问题 #19175
  - 修复 Duplicate Key 检测在悲观事务下失效的问题 #19236
  - 修复 `Apply` 算子和 `Union Scan` 算子执行导致结果错误的问题 #19297
  - 修复某些缓存的执行计划在事务中执行结果错误的问题 #19274
- TiKV
  - 将 GC 的失败日志从 Error 级别改成 Warning 级别 #8444

- Tools
  - TiDB Lightning
    * 修复命令行参数 `--log-file` 无法生效的问题 #345
    * 修复 TiDB-backend 遇到空的 binary/hex 报语法错误的问题 #357
    * 修复使用 TiDB backend 时非预期的 `switch-mode` 调用 #368

14.4.4 TiDB 3.0.17 Release Notes

发版日期：2020 年 8 月 3 日

TiDB 版本：3.0.17

14.4.4.1 Bug 修复

- TiDB
  - 当一个查询中含有 `IndexHashJoin` 或 `IndexMergeJoin` 算子，且该算子的子节点发生 panic 时，返回客户端 panic 的原因，而非返回空结果 #18498
  - 修复形如 `SELECT a FROM t HAVING t.a` 的查询返回 `UnknowColumn` 错误的问题 #18432
  - 当一张表没有主键，或其主键为整型时，禁止在这张表上执行添加主键 #18342
  - 对 `EXPLAIN FORMAT="dot" FOR CONNECTION` 始终返回空结果 #17157
  - 修复 `STR_TO_DATE` 函数处理 `'%r'` 和 `'%h'` 的行为 #18725
- TiKV
  - 修复在 Region 合并过程中可能导致读到旧数据的问题 #8111
  - 修复调度时可能产生内存泄漏的问题 #8355
- TiDB Lightning
  - 解决 `log-file` 参数不生效的问题 #345

14.4.4.2 优化

- TiDB
  - 将配置项 `query-feedback-limit` 默认值从 1024 修改为 512, 并优化统计信息反馈机制，降低其对集群的性能影响 #18770
  - 限制单次 split 请求中的 Region 个数 #18694
  - 加速 HTTP API `/tiflash/replica` 在集群中存在大量历史 DDL 记录时的访问速度 #18386
  - 提升索引等值条件下的行数估算准确率 #17609
  - 加快 `kill tidb conn_id` 的响应速度 #18506
- TiKV
  - 新增 `hibernate-timeout` 配置支持推后 Region 休眠时间，减少 Region 休眠对滚动升级的影响 #8207
- TiDB Lightning
  - 废弃 `[black-white-list]` 参数，新增一种更加简单易用的过滤规则 #332

14.4.5 TiDB 3.0.16 Release Notes

发版日期：2020 年 7 月 3 日

TiDB 版本：3.0.16

14.4.5.1 优化

- TiDB
  - 在 hash partition pruning 中支持 `is null` 过滤条件 #17308
  - 为每个 Region 设置单独的 `Backoffer` 避免多个 Region 同时失败引起等待时间过长 #17583
  - 添加新 partition 更新已有 partition 的分裂信息 #17668
  - 丢弃来自 `delete` / `update` 语句的 feedbacks #17841
  - 调整 `job.DecodeArgs` 中 `json.Unmarshal` 的使用以兼容新的 Go 版本 #17887
  - 移除 `slow log` 和 `statement summary` 中一些敏感信息 #18128
  - `Datetime` 解析的分隔符和 MySQL 兼容 #17499
  - 解析日期的 `%h` 时限定在 `1..12` 范围内 #17496
- TiKV
  - 避免在收到 snapshot 之后发送心跳给 PD 以提高稳定性 #8145
  - 优化了 PD client 的日志 #8091

14.4.5.2 Bug 修复

- TiDB
  - 修复当锁住的 primary key 在当前事务被插入/删除时可能造成的结果不一致问题 #18248
  - 修复因字段含义不一致导致日志中出现大量 `Got too many pings` gRPC 错误的问题 #17944
  - 修复当 HashJoin 返回 `Null` 类型列可能造成的 panic 问题 #17935
  - 修复访问被拒绝时的错误信息 #17722
  - 修复 JSON 数据中 `int` 和 `float` 类型比较的问题 #17715
  - 修复 Failpoint 测试造成的 data race 问题 #17710
  - 修复 Region 预分裂超时在创建表时可能不生效的问题 #17617
  - 修复 `BatchClient` 中因为失败可能导致的主动 panic #17378
  - 修复 `FLASHBACK TABLE` 在某些情况下可能失败的问题 #17165
  - 修复只有 string 列时 range 范围计算可能不准确的问题 #16658
  - 修复 `only_full_group_by` 模式下的错误 #16620
  - 修复 `case when` 函数返回字段长度不准确的问题 #16562
  - 修复 `count` 聚合函数对 `decimal` 类型推断的问题 #17702
- TiKV
  - 修复了潜在的 ingest file 导致的读取结果错误的问题 #8039
  - 修复了多次 merge 过程中被隔离的节点上的副本无法被正确移除的问题 #8005
- PD
  - 修复一些情况下使用 PD Control 查询 Region 报 404 错误的问题 #2577

14.4.6 TiDB 3.0.15 Release Notes

发版日期：2020 年 6 月 5 日

TiDB 版本：3.0.15

14.4.6.1 新功能

- TiDB
  - 禁止分区表上的查询使用 plan cache 功能 #16759
  - 分区表支持 `admin recover index`、`admin check index` 语句 #17315 #17390
  - Range 类型分区表支持按 `in` 查询条件进行分区裁剪 #17318
  - 优化 `SHOW CREATE TABLE` 的输出结果，在分区名称上添加了引号 #16315
  - `GROUP_CONCAT` 支持 `ORDER BY` 子句 #16988
  - 优化统计信息 `CMSketch` 的内存分配机制，减少垃圾回收导致的性能影响 #17543
- PD
  - 新增按照 Leader 个数调度的策略 #2479

14.4.6.2 Bug 修复

- TiDB
  - Hash 聚合函数中，采用深拷贝的方式拷贝 `enum` 和 `set` 类型数据，且修复一处正确性问题 #16890
  - 修复点查因整数溢出处理逻辑不正确导致输出结果不正确的问题 #16753
  - 修复 `CHAR()` 函数作为查询的谓词条件时因处理逻辑不正确导致输出的结果不正确的问题 #16557
  - 修复 `IsTrue` 和 `IsFalse` 函数存储层和计算层计算结果不一致的问题#16627
  - 修复部分表达式（例如 `case when`）中，`Not Null` 标记设置不正确的问题 #16993
  - 修复部分场景中优化器无法为 `TableDual` 找到物理计划的问题 #17014
  - 修复 Hash 分区表中分区选择的语法没有正确生效的问题 #17051
  - 修复 XOR 作用于浮点数时，结果与 MySQL 不一致的问题 #16976
  - 修复 prepare 方式执行 DDL 语句出错的问题 #17415
  - 修复 ID 分配器中计算 Batch 大小的逻辑处理不正确的问题 #17548
  - 修复 `MAX_EXEC_TIME` 的 SQL Hint 在超过 expensive 阈值后不生效的问题 #17534
- TiKV
  - 修复长时间运行后由于处理逻辑不正确导致碎片整理不再有效的问题 #7790
  - 修复系统意外重启后错误地删除 snapshot 文件导致系统 panic 的问题 #7925
  - 修复因消息包过大导致 gRPC 连接断开的问题 #7822

14.4.7 TiDB 3.0.14 Release Notes

发版日期：2020 年 5 月 9 日

TiDB 版本：3.0.14

14.4.7.1 兼容性变化

- TiDB
  - `performance_schema` 和 `metrics_schema` 由读写改为只读 #15417

14.4.7.2 重点修复的 Bug

- TiDB
  - 修复 join 条件在 handle 列上存在多个等值条件时，index join 查询结果错误的问题 #15734
  - 修复 fast analyze handle 列 panic 的问题 #16079
  - 修复通过 prepare 方式执行 DDL 语句时，DDL job 结构中 query 字段错误的问题，该问题可能导致使用 Binlog 同步时，上下游数据产生不一致 #15443
- TiKV
  - 修复重复清锁请求可能破坏事务原子性的问题 #7388

14.4.7.3 新功能

- TiDB
  - `admin show ddl jobs` 查询结果中添加库名和表名列 #16428
  - `RECOVER TABLE` 支持恢复被 `TRUNCATE` 的表 #15458
  - 新增 `SHOW GRANTS` 语句权限检查的功能 #16168
  - 新增 `LOAD DATA` 语句权限检查 #16736
  - 提升时间日期相关函数作为 partition key 时，分区裁剪的性能 #15618
  - `dispatch error` 的日志级别从 `WARN` 调整为 `ERROR` #16232
  - 新增支持 `require-secure-transport` 启动项，以强制要求客户端必须使用 TLS #15415
  - 支持内部组件间 http 通信使用 TLS #15419
  - `information_schema.processlist` 表中添加显示当前事务 `start_ts` 信息 #16160
  - 新增自动重加载集群间通讯 TLS 证书信息的功能 #15162
  - 通过重构分区裁剪的实现，提升分区表的读操作的性能 #15628
  - 新增当使用 `floor(unix_timestamp(a))` 作为 `range` 分区表的分区表达式时，支持分区裁剪功能 #16521
  - 修改 `update` 语句中包含 `view` 且不对该 `view` 进行 update 时的行为，由不允许执行改为正常执行 #16787
  - 禁止创建嵌套 `view` #15424
  - 禁止 truncate `view` #16420
  - 当列处于非 public 状态时，禁止用 `update` 语句显式的更新此列的值 #15576
  - 当 status 端口被占用时，禁止启动 TiDB #15466
  - `current_role` 函数的字符集由 binary 调整为 utf8mb4 #16083
  - 通过在处理完每个 Region 后增加检查 `max-execution-time` 是否符合条件，提升系统处理 `max-execution-time` 的响应灵敏度 #15615
  - 新增语法 `ALTER TABLE ... AUTO_ID_CACHE` 用于显式设置 `auto_id` 的缓存步长 #16287

- TiKV
  - 提升乐观事务存在大量冲突及 BatchRollback 存在时的性能 #7605
  - 提升悲观事务冲突严重的场景下悲观锁 waiter 被频繁唤醒导致性能下降的问题 #7584
- Tools
  - TiDB Lightning
    * tidb-lightning-ctl 新增 fetch-mode 子命令，输出 TiKV 集群模式 #287

14.4.7.4 Bug 修复

- TiDB
  - 修复 WEEKEND 函数在 SQL mode 为 ALLOW_INVALID_DATES 时结果与 MySQL 不兼容的问题 #16170
  - 修复当索引列上包含自增主键时，DROP INDEX 执行失败的问题 #16008
  - 修复 Statement Summary 中，TABLE_NAMES 列值有时会不正确的问题 #15231
  - 修复因 Plan Cache 启动后部分表达式计算结果错误的问题 #16184
  - 修复函数 not/istrue /isfalse 计算结果错误的问题 #15916
  - 修复带有冗余索引的表 MergeJoin 时 Panic 的问题 #15919
  - 修复谓词只跟外表有联接的情况下错误地化简外链接的问题 #16492
  - 修复 SET ROLE 导致的 CURRENT_ROLE 函数报错问题 #15569
  - 修复 LOAD DATA 在遇到 \ 时，处理结果与 MySQL 不兼容的问题 #16633
  - 修复数据库可见性与 MySQL 不兼容的问题 #14939
  - 修复 SET DEFAULT ROLE ALL 语句的权限检查不正确的问题 #15585
  - 修复 plan cache 导致的分区裁剪失效问题 #15818
  - 修复因事务未对相关表进行加锁，该表存在并发的 DDL 操作且有阻塞时导致事务提交时报 schema change 的问题 #15707
  - 修复 IF(not_int, *, *) 行为不正确的问题 #15356
  - 修复 CASE WHEN (not_int) 行为不正确的问题 #15359
  - 修复在使用非当前 schema 中的视图时报 Unknown column 错误的问题 #15866
  - 修复解析时间字符串的结果与 MySQL 不兼容的问题 #16242
  - 修复 left join 右孩子节点有 null 列可能会导致 join 上的排序算子 panic 的问题 #15798
  - 修复当 TiKV 持续返回 StaleCommand 错误期间，执行 SQL 的流程被阻塞且不报错的问题 #16528
  - 修复启用审计插件后端口探活可能会导致 panic 的问题 #16064
  - 修复 fast analyze 作用于 index 时导致 panic 的问题 #15967
  - 修复某些情况下 SELECT * FROM INFORMATION_SCHEMA.PROCESSLIST 语句 panic 的问题 #16309
  - 修复哈希分区表在建表时由于分配内存之前未及时检查分区数量导致当指定非常大的分区数量（例如 9999999999999）时，导致 TiDB OOM 的问题 #16218
  - 修复 information_schema.tidb_hot_table 对于分区表信息不准确的问题 #16726
  - 修复分区选择算法在哈希分区表上不生效的问题 #16070
  - 修复 mvcc 系列的 HTTP API 不支持分区表的问题 #16191
  - 保持 UNION 语句和 SELECT 语句对于错误处理的行为一致 #16137
  - 修复当 VALUES 函数参数类型为 bit(n) 时行为不正确的问题 #15486
  - 修复 view 列名过长时处理逻辑与 MySQL 不一致的问题，当列名过长时，系统自动生成一个短的列名 #14873

- 修复 `(not not col)` 被错误地优化为 `col` 的问题 #16094
- 修复 index join 构造内表 range 错误的问题 #15753
- 修复 `only_full_group_by` 对含括号的表达式检查错误的问题 #16012
- 修复 `select view_name.col_name from view_name` 报错的问题 #15572

- TiKV
  - 修复某些情况节点隔离恢复之后无法被正确删掉的问题 #7703
  - 修复网络隔离时 Region Merge 可能导致数据丢失的问题 #7679
  - 修复某些情况 learner 无法被正确移除的问题 #7598
  - 修复扫描 raw kv 时可能乱序的问题 #7597
  - 修复由于 Raft 消息 batch 过大时导致连接重连的问题 #7542
  - 修复 empty request 造成 gRPC 线程死锁的问题 #7538
  - 修复 merge 过程中 learner 重启的处理逻辑不正确的问题 #7457
  - 修复重复清锁请求可能破坏事务原子性的问题 #7388

14.4.8 TiDB 3.0.13 Release Notes

发版日期：2020 年 04 月 22 日

TiDB 版本：3.0.13

14.4.8.1 Bug 修复

- TiDB
  - 修复由于未检查 `MemBuffer`，事务内执行 `INSERT ... ON DUPLICATE KEY UPDATE` 语句插入多行重复数据可能出错的问题 #16690

- TiKV
  - 修复重复多次执行 `Region Merge` 导致系统被阻塞的问题，阻塞期间服务不可用 #7612

14.4.9 TiDB 3.0.12 Release Notes

发版日期：2020 年 3 月 16 日

TiDB 版本：3.0.12

TiDB Ansible 版本：3.0.12

> **警告：**
>
> 该版本存在一些已知问题，已在新版本中修复，建议使用 3.0.x 的最新版本。

14.4.9.1 兼容性变化

- TiDB
  - 修复慢日志中记录 prewrite binlog 的时间部分计时不准确问题。原本计时的字段名是 `Binlog_prewrite_time`，这次修正后，名称更改为 `Wait_prewrite_binlog_time`。#15276

14.4.9.2 新功能

- TiDB
  - 支持通过 `alter instance` 语句动态加载已被替换的证书文件 #15080 #15292
  - 添加 `cluster-verify-cn` 配置项，配置后必须是对应 CN 证书才使用 status 服务 #15164
  - 在每个 TiDB server 中添加对 DDL 请求的一个限流的功能，从而降低 DDL 请求冲突报错频率 #15148
  - 支持在 binlog 写入失败时，TiDB 退出 #15339
- Tools
  - TiDB Binlog
    * Drainer 新增 `kafka-client-id` 配置项，支持连接 Kafka 客户端配置客户端 ID #929

14.4.9.3 Bug 修复

- TiDB
  - 使 GRANT/REVOKE 在对多个用户修改时，保证原子性 #15092
  - 修复在分区表上面悲观锁的加锁未能锁住正确的行的问题 #15114
  - 建索引长度超过限制时，使报错信息根据配置中 `max-index-length` 的值显示 #15130
  - 修复 `FROM_UNIXTIME` 函数小数点位数不正确的问题 #15270
  - 修复一个事务中删除自己写的记录导致冲突检测失效或数据索引不一致问题 #15176
- TiKV
  - 修复一个在关闭一致性检查参数时，在事务中插入一个已存在的 Key 然后立马删除，导致冲突检测失效或数据索引不一致的问题 #7054
  - Raftstore 引入流控机制，解决没有流控可能导致追日志太慢可能导致集群卡住，以及事务大小太大会导致 TiKV 间连接频繁重连的问题 #7072 #6993
- PD
  - 修复 PD 因处理 Region heartbeat 时的数据竞争导致 Region 信息不正确的问题 #2233
- TiDB Ansible
  - 支持一个集群部署多个 Grafana/Prometheus/Alertmanager #1198

14.4.10 TiDB 3.0.11 Release Notes

发版日期：2020 年 3 月 4 日

TiDB 版本：3.0.11

TiDB Ansible 版本：3.0.11

> **警告：**
>
> 该版本存在一些已知问题，已在新版本中修复，建议使用 3.0.x 的最新版本。

14.4.10.1 兼容性变化

- TiDB
  - 新增 `max-index-length` 配置项，用于控制索引支持的最大长度，用户可自由选择兼容 v3.0.7 之前版本或者兼容 MySQL #15057

14.4.10.2 新功能

- TiDB
  - 新增在 `information_schema.PARTITIONS` 表中显示分区表的分区元信息的功能 #14849
- TiDB Binlog
  - 新增 TiDB 集群之间数据双向复制功能 #884 #909
- TiDB Lightning
  - 新增配置 TLS 功能 #44 #270
- TiDB Ansible
  - 优化 `create_user.yml` 的逻辑，中控机使用的用户不必和 `ansible_user` 一致 #1184

14.4.10.3 Bug 修复

- TiDB
  - 修复由于涉及 `Union` 的查询没有标记为只读，在乐观事务开启重试时会导致 Goroutine 泄露的问题 #15076
  - 修复执行 `SET SESSION tidb_snapshot = 'xxx';` 语句后，由于执行时未正确使用 `tidb_snapshot` 变量的值，导致 `SHOW TABLE STATUS` 未正确输出快照时刻表状态的问题 #14391
  - 修复 `Sort Merge Join` 与 `ORDER BY DESC` 在同一条 SQL 语句中时，输出结果不正确的问题 #14664
  - 修复创建分区表时，由于使用不支持的表达式，导致 TiDB server panic 的问题，修复后返回 `This ↪ partition function is not allowed` 错误信息 #14769

  - 修复执行 `select max()from subquery` 语句且 Subquery 包含 Union 的子查询时，输出结果不正确的问题 #14944
  - 修复执行 `DROP BINDING` 语句解除执行计划绑定后，执行 `SHOW BINDINGS` 语句系统返回错误信息的问题 #14865
  - 修复查询语句中别名长度大于 256 时，由于在查询结果中未按照 MySQL 协议对别名截断，导致连接被断开的问题 #14940
  - 修复字符串类型被用作 DIV 中时，查询结果可能不正确的问题，例如：`select 1 / '2007' div 1` 现在可以被正确地执行 #14098

- TiKV

  - 优化日志输出，删除部分不必要的日志 #6657
  - 修复 peer 在高负载情况下若被删除可能导致 panic 的问题 #6704
  - 修复 Hibernate Region 在某些特殊条件下未被正确唤醒的问题 #6732 #6738

- TiDB Ansible

  - 修复 `tidb-ansible` 中失效、过期的文档链接 #1169
  - 修复 `wait for region replication complete` task 可能出现未定义变量的问题 #1173

14.4.11 TiDB 3.0.10 Release Notes

发版日期：2020 年 2 月 20 日

TiDB 版本：3.0.10

TiDB Ansible 版本：3.0.10

> **警告：**
>
> 该版本存在一些已知问题，已在新版本中修复，建议使用 3.0.x 的最新版本。

14.4.11.1 TiDB

- 修复 IndexLookUpJoin 在利用 OtherCondition 构造 InnerRange 时出现错误 Join 结果 #14599
- 删除 `tidb_pprof_sql_cpu` 配置项，新增 Server 级别的 `tidb_pprof_sql_cpu` 变量 #14416
- 修复用户只在具有全局权限时才能查询所有数据库的问题 #14386
- 修复执行 point-get 时由于事务超时导致数据的可见性不符合预期的问题 #14480
- 将悲观事务激活的时机改为延迟激活，与乐观事务模型保持一致 #14474
- 修复 unixtimestamp 表达式在计算分区表分区的时区不正确的问题 #14476
- 新增`tidb_session_statement_deadlock_detect_duration_seconds` 监控项，用于监控死锁检测时间 #14484
- 修复 GC worker 由于部分逻辑不正确导致系统 panic 的问题 #14439
- 修复 IsTrue 函数的表达式名称不正确的问题 #14516
- 修复部分内存使用统计不准确的问题 #14533
- 修复统计信息 CM-Sketch 初始化时由于处理逻辑不正确导致系统 panic 的问题 #14470

- 修复查询分区表时分区裁剪（partition pruning）不准确的问题 #14546
- 修复 SQL 绑定中 SQL 语句默认数据库名设置不正确的问题 #14548
- 修复 json_key 与 MySQL 不兼容的问题 #14561
- 新增分区表自动更新统计信息的功能 #14566
- 修复执行 point-get 时 plan id 会变化的问题，正常情况 plan id 始终是 1 #14595
- 修复 SQL 绑定不完全匹配时处理逻辑不正确导致系统 panic 的问题 #14263
- 新增 `tidb_session_statement_pessimistic_retry_count` 监控项，用于监控悲观事务加锁失败后重试次数 #14619
- 修复 `show binding` 语句权限检查不正确的问题 #14618
- 修复由于 backoff 的逻辑里没有检查 killed 标记，导致 kill 无法正确执行的问题 #14614
- 通过减少持有内部锁的时间来提高 statement summary 的性能 #14627
- 修复 TiDB 从字符串解析成时间与 MySQL 不兼容的问题 #14570
- 新增审计日志记录用户登录失败的功能 #14620
- 新增 `tidb_session_ statement_lock_keys_count` 监控项，用于监控悲观事务的 lock keys 的数量 #14634
- 修复 json 对 & < > 等字符输出转义不正确的问题 #14637
- 修复 hash-join 在建 hash-table 时由于内存使用过多导致系统 panic 的问题 #14642
- 修复 SQL 绑定处理不合法记录时处理逻辑不正确导致 panic 的问题 #14645
- 修复 Decimal 除法计算与 MySQL 不兼容的问题，Decimal 除法计算中增加 Truncated 错误检测 #14673
- 修复给用户授权不存在的表执行成功的问题 #14611

14.4.11.2 TiKV

- Raftstore
  - 修复由于 Region merge 失败导致系统 Panic #6460 或者数据丢失 #598 的问题 #6481
  - 支持 yield 优化调度公平性，支持预迁移 leader 优化 leader 调度的稳定性 #6563

14.4.11.3 PD

- 当系统流量有变化时，系统自动更新 Region 缓存信息，解决缓存失效的问题 #2103
- 采用 leader 租约时间确定 TSO 的有效时间 #2117

14.4.11.4 Tools

- TiDB Binlog
  - Drainer 支持 relay log #893
- TiDB Lightning
  - 优化配置项，部分配置项在没有设置的时候使用默认配置 #255
  - 修复在非 server mode 模式下 web 界面无法打开的问题 #259

14.4.11.5 TiDB Ansible

- 修复某些场景获取不到 PD Leader 导致命令执行失败的问题 #1121
- TiDB Dashboard 新增 `Deadlock Detect Duration` 监控项 #1127
- TiDB Dashboard 新增 `Statement Lock Keys Count` 监控项 #1132
- TiDB Dashboard 新增 `Statement Pessimistic Retry Count` 监控项 #1133

14.4.12 TiDB 3.0.9 Release Notes

发版日期：2020 年 1 月 14 日

TiDB 版本：3.0.9

TiDB Ansible 版本：3.0.9

> **警告：**
>
> 该版本存在一些已知问题，已在新版本中修复，建议使用 3.0.x 的最新版本。

14.4.12.1 TiDB

- Executor
  - 修复聚合函数作用于枚举和集合列时结果不正确的问题 #14364
- Server
  - 支持系统变量 `auto_increment_increment` 和 `auto_increment_offset` #14396
  - 新增 `tidb_tikvclient_ttl_lifetime_reach_total` 监控项，监控悲观事务 TTL 达到 10 分钟的数量 #14300
  - 执行 SQL 过程中当发生 panic 时输出导致 panic 的 SQL 信息 #14322
  - statement summary 系统表新增 `plan` 和 `plan_digest` 字段，记录当前正在执行的 `plan` 和 `plan` 的签名 #14285
  - 配置项 `stmt-summary.max-stmt-count` 的默认值从 100 调整至 200 #14285
  - slow query 表新增 `plan_digest` 字段，记录 `plan` 的签名 #14292
- DDL
  - 修复 `alter table ... add index` 语句创建匿名索引行为与 MySQL 不一致的问题 #14310
  - 修复 `drop table` 错误删除视图的问题 #14052
- Planner
  - 提升类似 `select max(a), min(a)from t` 语句的性能。如果 a 列表上有索引，该语句会被优化为 `select * from (select a from t order by a desc limit 1)as t1, (select a from t order by a limit 1)as t2` 以避免全表扫 #14410

14.4.12.2 TiKV

- Raftstore
  - 提升 Raft 成员变更的速度 #6421
- Transaction
  - 新增 `tikv_lock_manager_waiter_lifetime_duration`、`tikv_lock_manager_detect_duration`、`tikv_lock_manager_detect_duration` 监控项，用于监控 `waiter` 的生命周期、死锁检测耗费时间、`wait table` 的状态 #6392
  - 通过优化配置项 `wait-for-lock-time` 默认值从 `3s` 调整到 `1s`、`wake-up-delay-duration` 默认值从 `100ms` 调整为 `20ms`，以降低极端场景下 Region Leader 切换、切换死锁检测的 leader 导致的事务执行延迟 #6429
  - 修复 Region Merge 过程中可能导致死锁检测器 leader 角色误判的问题 #6431

14.4.12.3 PD

- 新增 location label 的名字中允许使用斜杠 / 的功能 #2083
- 修复因为不正确地统计了 tombstone 的标签，导致该统计信息不准的问题 #2060

14.4.12.4 Tools

- TiDB Binlog
  - Drainer 输出的 binlog 协议中新增 unique key 信息 #862
  - Drainer 支持使用加密后的数据库连接密码 #868

14.4.12.5 TiDB Ansible

- 优化 Lightning 部署，自动创建相关目录 #1105

14.4.13 TiDB 3.0.8 Release Notes

发版日期：2019 年 12 月 31 日

TiDB 版本：3.0.8

TiDB Ansible 版本：3.0.8

14.4.13.1 TiDB

- SQL 优化器
  - 修复 SQL Binding 因为 cache 更新不及时，导致绑定计划错误的问题 #13891
  - 修复当 SQL 包含符号列表（类似于 “?, ?, ?” 这样的占位符）时，SQL Binding 可能失效的问题 #14004
  - 修复 SQL Binding 由于原 SQL 以 ; 结尾而不能创建/删除的问题 #14113

  - 修复 PhysicalUnionScan 算子没有正确设置统计信息，导致查询计划可能选错的问题 #14133
  - 移除 minAutoAnalyzeRatio 约束使自动 analyze 更及时 #14015

- SQL 执行引擎

  - 修复 INSERT/REPLACE/UPDATE ... SET ... = DEFAULT 语法会报错的问题，修复 DEFAULT 表达式与虚拟生成列配合使用会报错的问题 #13682
  - 修复 INSERT 语句在进行字符串类型到浮点类型转换时，可能会报错的问题 #14011
  - 修复 HashAgg Executor 并发值未被正确初始化，导致聚合操作执行在一些情况下效率低的问题 #13811
  - 修复 group by item 被括号包含时执行报错的问题 #13658
  - 修复 TiDB 没有正确计算 group by item，导致某些情况下 OUTER JOIN 执行会报错的问题 #14014
  - 修复向 Range 分区表写入超过 Range 外的数据时，报错信息不准确的问题 #14107
  - 鉴于 MySQL 8 即将废弃 PadCharToFullLength，revert PR #10124 并撤销 PadCharToFullLength 的效果，以避免一些特殊情况下查询结果不符合预期 #14157
  - 修复 ExplainExec 中没有保证 close() 的调用而导致 EXPLAIN ANALYZE 时造成 goroutine 泄露的问题 #14226

- DDL

  - 优化 “change column” / “modify column” 的输出的报错信息，让人更容易理解 #13796
  - 新增 SPLIT PARTITION TABLE 语法，支持分区表切分 Region 功能 #13929
  - 修复创建索引时，没有正确检查长度，导致索引长度超过 3072 字节没有报错的问题 #13779
  - 修复由于分区表添加索引时若花费时间过长，可能导致输出 GC life time is shorter than transaction duration 报错信息的问题 #14132
  - 修复在 DROP COLUMN/MODIFY COLUMN/CHANGE COLUMN 时没有检查外键导致执行 SELECT * FROM information_schema.KEY_COLUMN_USAGE 语句时发生 panic 的问题 #14105

- Server

  - Statement Summary 功能改进：
    * 新增大量的 SQL 指标字段，便于对 SQL 进行更详细的统计分析 #14151，#14168
    * 新增 stmt-summary.refresh-interval 参数用于控制定期将 events_statements_summary_by_digest 表中过期的数据移到 events_statements_summary_by_digest_history 表，默认间隔时间：30min #14161
    * 新增 events_statements_summary_by_digest_history 表，保存从 events_statements_summary_by_digest 中过期的数据 #14166
  - 修复执行 RBAC 相关的内部 SQL 时，错误输出 binlog 的问题 #13890
  - 新增 server-version 配置项来控制修改 TiDB server 版本的功能 #13906
  - 新增通过 HTTP 接口恢复 TiDB binlog 写入功能 #13892
  - 将 GRANT roles TO user 所需要的权限由 GrantPriv 修改为 ROLE_ADMIN 或 SUPER，以与 MySQL 保持一致 #13932
  - 当 GRANT 语句未指定 database 名时，TiDB 行为由使用当前 database 改为报错 No database selected，与 MySQL 保持兼容 #13784
  - 修改 REVOKE 语句执行权限从 SuperPriv 改成用户只需要有对应 Schema 的权限，就可以执行 REVOKE 语句，与 MySQL 保持一致 #13306
  - 修复 GRANT ALL 语法在没有 WITH GRANT OPTION 时，错误地将 GrantPriv 授权给目标用户的问题 #13943

- 修复 LoadDataInfo 中调用 addRecord 报错时，报错信息不包含导致 LOAD DATA 语句行为不正确信息的问题 #13980
- 修复因查询中多个 SQL 语句共用同一个 StartTime 导致输出错误的慢查询信息的问题 #13898
- 修复 batchClient 处理大事务时可能造成内存泄露的问题 #14032
- 修复 system_time_zone 固定显示为 CST 的问题，现在 TiDB 的 system_time_zone 会从 mysql.tidb 表中的 systemTZ 获取 #14086
- 修复 GRANT ALL 语法授予权限不完整（例如 Lock_tables_priv）的问题 #14092
- 修复 Priv_create_user 权限不能 CREATE ROLE 和 DROP ROLE的问题 #14088
- 将 ErrInvalidFieldSize 的错误码从 1105(Unknow Error) 改成 3013 #13737
- 新增 SHUTDOWN 命令用于停止 TiDB Server，并新增 ShutdownPriv 权限 #14104
- 修复 DROP ROLE 语句的原子性问题，避免语句执行失败时，一些 ROLE 仍然被非预期地删除 #14130
- 修复 3.0 以下版本升级到 3.0 时，tidb_enable_window_function 在 SHOW VARIABLE 语句的查询结果错误输出 1 的问题，修复后输出 0 #14131
- 修复 TiKV 节点下线时，由于 gcworker 持续重试导致可能出现 goroutine 泄露的问题 #14106
- 在慢日志中记录 Binlog 的 Prewrite 的时间，提升问题追查的易用性 #14138
- tidb_enable_table_partition 变量支持 GLOBAL SCOPE 作用域 #14091
- 修复新增权限时未正确将新增的权限赋予对应的用户导致用户权限可能缺失或者被误添加的问题 #14178
- 修复当 TiKV 链接断开时，由于 rpcClient 不会关闭而导致 CheckStreamTimeoutLoop goroutine 会泄露的问题 #14227
- 支持基于证书的身份验证（使用文档）#13955

- Transaction
  - 创建新集群时，tidb_txn_mode 变量的默认值由 "" 改为 "pessimistic" #14171
  - 修复悲观事务模式，事务重试时单条语句的等锁时间没有被重置导致等锁时间过长的问题 #13990
  - 修复悲观事务模式，因对没有修改的数据未加锁导致可能读到不正确数据的问题 #14050
  - 修复 mocktikv 中 prewrite 时，没有区分事务类型，导致重复的 insert value 约束检查 #14175
  - 修复 session.TxnState 状态为 Invalid 时，事务没有被正确处理导致 panic 的问题 #13988
  - 修复 mocktikv 中 ErrConfclit 结构未包含 ConflictCommitTS 的问题 #14080
  - 修复 TiDB 在 Resolve Lock 之后，没有正确处理锁超时检查导致事务卡住的问题 #14083
- Monitor
  - LockKeys 新增 pessimistic_lock_keys_duration 监控 #14194

14.4.13.2 TiKV

- Coprocessor
  - 修改 Coprocessor 遇到错误时输出日志的级别从 error 改成 warn #6051
  - 修改统计信息采样数据的更新行为从直接更行改成先删除再插入，更新行为与 tidb-server 保持一致 #6069
- Raftstore
  - 修复因重复向 peerfsm 发送 destory 消息，peerfsm 被多次销毁导致 panic 的问题 #6297
  - split-region-on-table 默认值由 true 改成 false，默认关闭按 table 切分 Region 的功能 #6253
- Engine

  - 修复极端条件下因 RocksDB 迭代器错误未正确处理导致可能返回空数据的问题 #6326

- 事务
  - 修复悲观锁因锁未被正确清理导致 Key 无法写入数据，且出现 GC 卡住的问题 #6354
  - 优化悲观锁等锁机制，提升锁冲突严重场景的性能 #6296
- 将内存分配库的默认值由 `tikv_alloc/default` 改成 `jemalloc` #6206

14.4.13.3 PD

- Client
  - 新增通过 `context` 创建新 client，创建新 client 时可设置超时时间 #1994
  - 新增创建 `KeepAlive` 连接功能 #2035
- 优化`/api/v1/regions` API 的性能 #1986
- 修复删除 `tombstone` 状态的 Store 可能会导致 panic 的隐患 #2038
- 修复从磁盘加载 Region 信息时错误的将范围有重叠的 Region 删除的问题 #2011，#2040
- 将 etcd 版本从 3.4.0 升级到 3.4.3 稳定版本，注意升级后只能通过 pd-recover 工具降级 #2058

14.4.13.4 Tools

- TiDB Binlog
  - 修复 Pump 由于没有收到 DDL 的 commit binlog 导致 binlog 被忽略的问题 #853

14.4.13.5 TiDB Ansible

- 回滚被精简的配置项 #1053
- 优化滚动升级时 TiDB 版本检查的逻辑 #1056
- TiSpark 版本升级到 2.1.8 #1061
- 修复 Grafana 监控上 PD 页面 Role 监控项显示不正确的问题 #1065
- 优化 Grafana 监控上 TiKV Detail 页面上 `Thread Voluntary Context Switches` 和 `Thread Nonvoluntary Context Switches` 监控项 #1071

14.4.14 TiDB 3.0.7 Release Notes

发版日期：2019 年 12 月 4 日

TiDB 版本：3.0.7

TiDB Ansible 版本：3.0.7

14.4.14.1 TiDB

- 修复 TiDB server 本地时间落后于 TSO 时间时，可能造成锁的 TTL 过大的问题 #13868
- 修复从字符串解析日期时，由于使用本地时区 (`gotime.Local`) 而导致解析结果的时区不正确的问题 #13793
- 修复 `builtinIntervalRealSig` 的实现中，`binSearch` 方法不会返回 error，导致最终结果可能不正确的问题 #13767
- 修复整型数据被转换为无符号浮点/Decimal 类型时，精度可能丢失造成数据错误的问题 #13755
- 修复 Natural Outer Join 和 Outer Join 使用 `USING` 语法时，`not null` 标记没有被重置导致结果错误的问题 #13739
- 修复更新统计信息时可能存在数据竞争，导致统计信息不准确的问题 #13687

14.4.14.2 TiKV

- 判断死锁检测服务的第一个 Region 时，加上 Region 合法检测，防止信息不完整的 Region 导致误判 #6110
- 修复潜在的内存泄漏问题 #6128

14.4.15 TiDB 3.0.6 Release Notes

发版日期：2019 年 11 月 28 日

TiDB 版本：3.0.6

TiDB Ansible 版本：3.0.6

14.4.15.1 TiDB

- SQL 优化器
  - 修复窗口函数 AST Restore SQL 文本后结果不正确问题，`over w` 不应被 restore 成 `over (w)` #12933
  - 修复 stream aggregation 下推给 double read 的问题 #12690
  - 修复 SQL bind 中引号处理不正确的问题 #13117
  - 优化 `select max(_tidb_rowid)from t` 场景，避免全表扫 #13095
  - 修复当查询语句中包含变量赋值表达式时查询结果不正确的问题 #13231
  - 修复 `UPDATE` 语句中同时包含子查询和 generated column 时结果错误的问题；修复 `UPDATE` 语句中包含不同数据库的两个表名相同的表时，`UPDATE` 执行报错的问题 #13350
  - 支持用 `_tidb_rowid` 做点查 #13416
  - 修复分区表统计信息使用错误导致生成执行计划不正确的问题 #13628
- SQL 执行引擎
  - 修复 year 类型对于无效值的处理时和 MySQL 不兼容问题 #12745
  - 在 `INSERT ON DUPLICATE UPDATE` 语句中复用 Chunk 以降低内存开销 #12998
  - 添加内置函数 `JSON_VALID` 的支持 #13133
  - 支持在分区表上执行 `ADMIN CHECK TABLE` #13140
  - 修复对空表进行 `FAST ANALYZE` 时 panic 的问题 #13343
  - 修复在包含多列索引的空表上执行 Fast Analyze 时 panic 的问题 #13394

- 修复当 WHERE 子句上有 UNIQUE KEY 的等值条件时，估算行数大于 1 的问题 #13382
- 修复当 TiDB 开启 Streaming 后返回数据有可能重复的问题 #13254
- 将 CMSketch 中出现次数最多的 N 个值抽取出来，提高估算准确度 #13429

- Server
  - 当 gRPC 请求超时时，提前让发往 TiKV 的请求失败 #12926
  - 添加以下虚拟表：#13009
    * performance_schema.tidb_profile_allocs
    * performance_schema.tidb_profile_block
    * performance_schema.tidb_profile_cpu
    * performance_schema.tidb_profile_goroutines
  - 修复 query 在等待悲观锁时，kill query 不生效的问题 #12989
  - 当悲观事务上锁失败，且事务只涉及一个 key 的修改时，不再异步回滚 #12707
  - 修复 split Region 请求的 response 为空时 panic 的问题 #13092
  - 悲观事务在其他事务先锁住导致上锁失败时，避免重复 backoff #13116
  - 修改 TiDB 检查配置时的行为，出现不能识别的配置选项时，打印警告日志 #13272
  - 支持通过 /info/all 接口获取所有 TiDB 节点的 binlog 状态 #13187
  - 修复 kill connection 时可能出现 goroutine 泄漏的问题 #13251
  - 让 innodb_lock_wait_timeout 参数在悲观事务中生效，用于控制悲观锁的等锁超时时间 #13165
  - 当悲观事务的 query 被 kill 后，停止更新悲观事务的 TTL，避免其他的事务做不必要的等待 #13046

- DDL
  - 修复 SHOW CREATE VIEW 结果与 MySQL 不一致的问题 #12912
  - 支持基于 UNION 创建 View，例如 create view v as select * from t1 union select * from t2 #12955
  - 给 slow_query 表添加更多事务相关的字段：#13072
    * Prewrite_time
    * Commit_time
    * Get_commit_ts_time
    * Commit_backoff_time
    * Backoff_types
    * Resolve_lock_time
    * Local_latch_wait_time
    * Write_key
    * Write_size
    * Prewrite_region
    * Txn_retry
  - 创建表时如果表包含 collate 则列使用表的 collate 而不是系统默认的字符集 #13174
  - 创建表时限制索引名字的长度 #13310
  - 修复 rename table 时未检查表名长度的问题 #13346
  - 新增 alter-primary-key 配置来支持 TiDB add/drop primary key，该配置默认关闭 #13522

14.4.15.2 TiKV

- 修复 acquire_pessimistic_lock 接口返回错误 txn_size 的问题 #5740

- 限制 GC worker 每秒写入量，降低对性能的影响 #5735
- 优化 lock manager 的准确度 #5845
- 悲观锁支持 innodb_lock_wait_timeout #5848
- 添加 Titan 相关配置检测 #5720
- 支持用 tikv-ctl 动态修改 GC 限流配置：`tikv-ctl --host=ip:port modify-tikv-config -m server -n gc.max_write_bytes_per_sec -v 10MB` #5957
- 减少无用的 clean up 请求，降低死锁检测器的压力 #5965
- 悲观事务 prewrite 时避免 TTL 被缩短 #6056
- 修复 Titan 可能发生 missing blob file 的问题 #5968
- 修复 Titan 可能导致 RocksDBOptions 不生效的问题 #6009

14.4.15.3 PD

- 为每个过滤器添加一个名为 ActOn 的新维度，以指示每个 scheduler 和 checker 受过滤器的影响。删除两个未使用的过滤器：disconnectFilter 和 rejectLeaderFilter #1911
- 当 PD 生成时间戳的时间超过 5 毫秒时，将打印一条 warning 日志 #1867
- 当存在 endpoint 不可用时，降低 client 日志级别 #1856
- 修复 region_syncer 同步时 gRPC 消息包可能过大的问题 #1952

14.4.15.4 Tools

- TiDB Binlog
  - Drainer 配置 initial-commit-ts 为 -1 时，从 PD 处获取初始同步时间戳 #788
  - Drainer checkpoint 存储与下游解耦，支持选择配置 checkpoint 保存到 MySQL 或者本地文件 #790
  - 修复 Drainer 在配置同步库表过滤使用空值会导致 Panic 的问题 #801
  - 修复 Drainer 因为向下游应用 Binlog 失败而 Panic 后进程没有退出而是进入死锁状态的问题 #807
  - 修复 Pump 下线因为 gRPC 的 GracefulStop 流程而 hang 住的问题 #817
  - 修复 Drainer 在 TiDB 执行 DROP COLUMN DDL 期间收到缺少一列的 binlog 而同步出错的问题（要求 TiDB 3.0.6 以上）#827
- TiDB Lightning
  - TiDB Backend 模式新增 max-allowed-packet 配置项，默认值为 64M #248

14.4.16 TiDB 3.0.5 Release Notes

发版日期：2019 年 10 月 25 日

TiDB 版本：3.0.5

TiDB Ansible 版本：3.0.5

14.4.16.1 TiDB

- SQL 优化器

  - 支持对 Window Functions 进行边界检查 #12404
  - 修复 partition 表上的 IndexJoin 返回错误结果的问题 #12712
  - 修复外连接 Apply 算子上层的 ifnull 函数返回错误结果的问题 #12694
  - 修复当 UPDATE 的 where 条件中包含子查询时更新失败的问题 #12597
  - 修复当查询条件中包含 cast 函数时 outer join 被错误转化为 inner join 的问题 #12790
  - 修复 AntiSemiJoin 的 join 条件中错误的表达式传递 #12799
  - 修复初始化统计信息时由于浅拷贝造成的统计信息出错问题 #12817
  - 修复 TiDB 中 str_to_date 函数在日期字符串和格式化字符串不匹配的情况下，返回结果与 MySQL 不一致的问题 #12725

- SQL 执行引擎

  - 修复在 from_unixtime 函数处理 null 时发生 panic 的问题 #12551
  - 修复 Admin Cancel DDL jobs 时报 invalid list index 错的问题 #12671
  - 修复使用 Window Functions 时发生数组越界的问题 #12660
  - 改进 AutoIncrement 列隐式分配时的行为，与 MySQL 自增锁的默认模式 (“consecutive” lock mode) 保持一致：对于单行 Insert 语句的多个自增 AutoIncrement ID 的隐式分配，TiDB 保证分配值的连续性。该改进保证 JDBC getGeneratedKeys() 方法在任意场景下都能得到正确的结果。#12602
  - 修复当 HashAgg 作为 Apply 子节点时查询 hang 住的问题 #12766
  - 修复逻辑表达式 AND 或 OR 在涉及类型转换时返回错误结果的问题 #12811

- Server

  - 实现修改事务 TTL 的接口函数，以助后续支持大事务 #12397
  - 支持将事务的 TTL 按需延长（最长可到 10min），用于支持悲观事务 #12579
  - 将 TiDB 缓存 schema 变更及相关表信息的次数从 100 调整为 1024，且支持通过 tidb_max_delta_schema_count 系统变量修改 #12502
  - 更新了 kvrpc.Cleanup 协议的行为，不再清理未超时事务的锁 #12417
  - 支持将 Partition 表信息记录到 information_schema.tables 表 #12631
  - 支持通过 region-cache-ttl 配置修改 Region Cache 的 TTL #12683
  - 支持在慢日志中打印执行计划压缩编码后的信息，此功能默认开启，可以通过 slow-log-plan 配置或者 tidb_record_plan_in_slow_log 变量进行开关控制。另外支持 tidb_decode_plan 函数将慢日志中的执行计划列编码信息解析成执行计划信息。#12808
  - 在 information_schema.processlist 表中支持显示内存使用信息 #12801
  - 修复 TiKV Client 判断连接空闲时可能出错并出现非预期的告警的问题 #12846
  - 修复 tikvSnapshot 没有正确对 BatchGet() 的 KV 结果进行缓存，导致 INSERT IGNORE 语句性能有所下降的问题 #12872
  - 修复了因建立到部分 KV 服务的连接较慢最终导致 TiDB 响应速度相对变慢的情况 #12814

- DDL

  - 修复 Create Table 操作对 Set 列不能正确设置 Int 类型默认值的问题 #12267
  - 支持 Create Table 语句中建唯一索引时带多个 Unique #12463
  - 修复使用 Alter Table 添加 Bit 类型列时，对存在的行填充此列的默认值可能出错的问题 #12489
  - 修复 Range 分区表以 Date 或 Datetime 类型列作为分区键时，添加分区失败的问题 #12815
  - 对于 Date 或 Datetime 类型列作为分区键的 Range 分区表，在建表或者添加分区时，支持检查分区类型与分区键类型的统一性 #12792
  - 在创建 Range 分区表时，添加对 Unique Key 列集合需大于等于分区列集合的检查 #12718

- Monitor

  - 添加统计 Commit 与 Rollback 操作的监控到 Transaction OPS 面板 #12505
  - 添加统计 Add Index 操作进度的监控 #12390

14.4.16.2 TiKV

- Storage
  - 悲观事务新特性：事务 Cleanup 接口支持只清理 TTL 已经过期的锁 #5589
  - 修复事务 Primary key 的 Rollback 被折叠的问题 #5646，#5671
  - 修复悲观锁下点查可能返回历史旧版本的问题 #5634
- Raftstore
  - 减少 Raftstore 消息的 flush 操作，以提升性能，减少 CPU 占用 #5617
  - 优化获取 Region 的大小和 key 个数估计值的开销，减少心跳的开销，降低 CPU 占用 #5620
  - 修复 Raftstore 取到非法数据时打印错误日志并 panic 的问题 #5643
- Engine
  - 打开 RocksDB `force_consistency_checks`，提高数据安全性 #5662
  - 修复 Titan 并发 flush 情况下有可能造成数据丢失的问题 #5672
  - 更新 rust-rocksdb 版本以避开 intra-L0 compaction 导致 TiKV 崩溃重启的问题 #5710

14.4.16.3 PD

- 提高 Region 占用空间的精度 #1782
- 修复 `--help` 命令输出内容 #1763
- 修复 TLS 开启后 http 请求重定向失败的问题 #1777
- 修复 pd-ctl 使用 `store shows limit` 命令 panic 的问题 #1808
- 提高 label 监控可读性以及当 leader 发生切换后重置原 leader 的监控数据，防止误报 #1815

14.4.16.4 Tools

- TiDB Binlog
  - 修复 `ALTER DATABASE` 相关 DDL 会导致 Drainer 异常退出的问题 #769
  - 支持对 Commit binlog 查询事务状态信息，提升同步效率 #757
  - 修复当 Drainer 的 `start_ts` 大于 Pump 中最大的 `commit_ts` 时，有可能引起 Pump panic 的问题 #758
- TiDB Lightning
  - 整合 Loader 全量逻辑导入功能，支持配置 backend 模式 #221

14.4.16.5 TiDB Ansible

- 增加 TiDB 添加索引速度的监控 #986
- 精简配置文件内容，移除不需要用户配置的参数 #1043c，#998
- 修复 performance read 和 performance write 监控表达式错误的问题 #e90e7
- 更新 raftstore CPU 使用率的监控显示方式以及 raftstore CPU 使用率的告警规则 #992
- 更新 Overview 监控面板中 TiKV 的 CPU 监控项，过滤掉多余的监控内容 #1001

14.4.17 TiDB 3.0.4 Release Notes

发版日期：2019 年 10 月 8 日

TiDB 版本：3.0.4

TiDB Ansible 版本：3.0.4

- 新特性
  - 新增系统表 `performance_schema.events_statements_summary_by_digest`，用于排查 SQL 级别的性能问题
  - TiDB 的 `SHOW TABLE REGIONS` 语法新增 `WHERE` 条件子句
  - Reparo 新增 `worker-count` 和 `txn-batch` 配置项，用于控制恢复速率
- 改进提升
  - TiKV 支持批量 `Split` 和空的 `Split` 命令，使得 Split 可以批量进行
  - TiKV 添加 RocksDB 双向链表支持，提升逆序扫性能
  - Ansible 新增 `iosnoop` 和 `funcslower` 两个 perf 工具，方便诊断集群状态
  - TiDB 优化慢日志输出内容，删除冗余字段
- 行为变更
  - TiDB 修改 `txn-local-latches.enable` 默认值为 `false`，默认不启用本地事务冲突检测
  - TiDB 添加全局作用域系统变量 `tidb_txn_mode`，允许配置使用悲观锁，请注意默认情况下，TiDB 仍然使用乐观锁
  - TiDB 慢日志中的 `Index_ids` 字段替换为 `Index_names` 字段，提升慢日志易用性
  - TiDB 配置文件中添加 `split-region-max-num` 参数，用于调整 `SPLIT TABLE` 语法允许的最大 Region 数量
  - TiDB 修改 SQL 超出内存限制后的行为，从断开链接修改为返回 `Out Of Memory Quota` 错误
  - 为避免误操作，TiDB 默认不再允许删除列的 `AUTO_INCREMENT` 属性，当确实需要删除时，请更改系统变量 `tidb_allow_remove_auto_inc`
- 问题修复
  - TiDB 修复特殊语法 `PRE_SPLIT_REGIONS` 没有使用注释的方式向下游同步的问题
  - TiDB 修复使用游标获取 `PREPARE + EXECUTE` 执行结果时，慢日志不正确的问题
  - PD 修复相邻小 Region 无法 Merge 的问题
  - TiKV 修复空闲集群中文件描述符泄漏导致长期运行可能会引起 TiKV 进程异常退出的问题
- 社区贡献者

  感谢以下社区贡献者参与本次发版：
  - sduzh
  - lizhenda

14.4.17.1 TiDB

- SQL 优化器
  - 修复 Feedback 切分查询范围出错的问题 #12170
  - 修改当 SHOW STATS_BUCKETS 结果中包含无效 Key 时的行为，将返回错误修改为使用 16 进制显示 #12094
  - 修复查询中包含 SLEEP 函数时（例如 `select 1 from (select sleep(1))t;`），由于列裁剪导致查询中的 `sleep(1)` 失效的问题 #11953
  - 当查询只关心表的行数而不关心表数据时，使用索引扫描降低 IO #12112
  - 当 `use index()` 中没有指定索引时不去使用任何索引，和 MySQL 兼容 (如 `explain select a from t use index();`) #12100
  - 严格限制统计信息 CMSketch 中 TopN 记录的数量，修复快速 analyze 因为超过事务大小限制而失败的问题 #11914
  - 修复 Update 语句包含子查询时，转换子查询出现的错误 #12483
  - 将 Limit 算子下推到 IndexLookUpReader 执行逻辑中优化 `select ... limit ... offset ...` 的执行性能 #12378
- SQL 执行引擎
  - PREPARED 语句执行错误时，在日志中打印 SQL 语句 #12191
  - 分区表使用 UNIX_TIMESTAMP 函数分区时，支持分区裁剪 #12169
  - 修复 AUTO_INCREMENT 分配 MAX int64 和 MAX uint64 没有报错的问题 #12162
  - SHOW TABLE ⋯ REGIONS 和 SHOW TABLE .. INDEX ⋯ REGIONS 语法新增 WHERE 条件子句 #12123
  - 修改 SQL 超出内存限制后的行为，从断开链接修改为返回 Out Of Memory Quota 错误 #12127
  - 修复 JSON_UNQUOTE 函数处理 JSON 文本结果不正确的问题 #11955
  - 修复 INSERT 语句中，第一行中为 AUTO_INCREMENT 列赋值，LAST INSERT ID 不正确的问题（例如 `insert into t (pk, c)values (1, 2), (NULL, 3)`）#12002
  - 修复 PREPARE 语句中，GroupBY 解析规则错误的问题 #12351
  - 修复点查中权限检查不正确的问题 #12340
  - 修复 PREPARE 语句类型没有记录在监控中的问题 #12331
  - 支持点查中表名使用别名（例如 `select * from t tmp where a = "aa"`）#12282
  - 修复向 BIT 类型列插入数值时，值没有作为无符号类型处理而导致插入负数报错的问题 #12423
  - 修复时间取整不正确的问题（例如 2019-09-11 11:17:47.999999666 应该被取整到 2019-09-11 11:17:48）#12258
  - 调整表达式黑名单系统表的用法（例如 < 与 lt 等价）#11975
  - 调整函数不存在的错误消息，添加数据库前缀（例如 `[expression:1305]FUNCTION test.std_samp does not exist`）#12111
- Server
  - 慢日志中添加 Prev_stmt 字段，用于最后一条语句是 COMMIT 时输出前一条语句 #12180
  - 优化慢日志输出内容，删除冗余字段 #12144
  - 修改 txn-local-latches.enable 默认值为 false，默认不启用本地事务冲突检测 #12095
  - 将慢日志中的 Index_ids 字段替换为 Index_names 字段，提升慢日志易用性 #12061
  - 添加全局作用域系统变量 tidb_txn_mode，允许配置使用悲观锁 #12049
  - 慢日志中添加 Backoff 字段，用来记录 2PC Commit 阶段的 Backoff 信息 #12335
  - 修复使用游标获取 PREPARE + EXECUTE 执行结果时，慢日志不正确的问题（例如 `PREPARE stmt1FROM SELECT * FROM t WHERE a > ?; EXECUTE stmt1 USING @variable`）#12392

  - 支持使用 tidb_enable_stmt_summary，开启后会对 SQL 语句进行统计，并可以使用系统表 performance_schema.events_statements_summary_by_digest 查询统计结果 #12308
  - 调整了 tikv-client 中部分日志级别（例如由于连接断开使得打印的 batchRecvLoop fails 日志级别由 ERROR 改为 INFO）#12383

- DDL

  - 新增变量 tidb_allow_remove_auto_inc，默认禁止删除列 AUTO_INCREMENT 属性 #12145
  - 修复 TiDB 特殊语法 PRE_SPLIT_REGIONS 没有使用注释的方式向下游同步，导致下游数据库报错的问题 #12120
  - 在配置文件中添加 split-region-max-num 参数，使得 SPLIT TABLE 语法允许的最大 Region 数量可调整，该参数默认值 10000 #12097
  - 支持将一个 Region 切分成多个 Region，并修复打散 Region 超时的问题 #12343
  - 修复当索引包含自增列，并且该自增列被两个索引引用时删除失败的问题 #12344

- Monitor

  - 增加监控指标 connection_transient_failure_count，用于统计 tikvclient 的 gRPC 连接错误数量 #12093

14.4.17.2 TiKV

- Raftstore

  - 修复 Raftstore 统计空 Region 中 key 个数不准确问题 #5414
  - 添加 RocksDB 双向链表支持，提升逆序扫性能 #5368
  - 支持 PD 批量 Split 和空的 Split 命令, 使得 Split 可以批量进行，提高 Split 效率 #5470

- Server

  - 修复查看版本命令的输出格式与 2.X 格式不一致的问题 #5501
  - 更新 Titan 至 3.0 分支最新版本 #5517
  - 更新 grpcio 至 v0.4.5 版本 #5523
  - 修复 gRPC coredump 问题，支持内存共享，以避免此处引起 OOM #5524
  - 修复空闲集群中文件描述符泄漏导致长期运行可能会引起 TiKV 进程异常退出的问题 #5567

- Storage

  - 支持悲观锁事务心跳检测 API，以使得 TiDB 的悲观锁行为与 MySQL 尽量一致 #5507
  - 修复部分情况下点查性能较低的问题 #5495 #5463

14.4.17.3 PD

- 修复相邻小 Region 无法 Merge 的问题 #1726
- 修复 pd-ctl 的 TLS 启用参数失效问题 #1738
- 修复可能导致 PD operator 被意外移除的线程安全问题 #1734
- Region syncer 支持 TLS #1739

14.4.17.4 Tools

- TiDB Binlog
  - Reparo 新增 `worker-count` 和 `txn-batch` 配置项，用于控制恢复速率 #746
  - Drainer 优化内存使用，提升同步执行效率 #737
- TiDB Lightning
  - 修复从 checkpoint 点重新导入可能会导致 TiDB Lightning 崩溃的问题 #237
  - 修改计算 `AUTO_INCREMENT` 的算法，降低溢出的风险 #227

14.4.17.5 TiDB Ansible

- 更新 TiSpark 版本至 2.2.0 #926
- 更新 TiDB 配置项 `pessimistic_txn` 的默认值为 `true` #933
- 新增更多系统级别监控到 `node_exporter` #938
- 新增 `iosnoop` 和 `funcslower` 两个 perf 工具，方便诊断集群状态 #946
- Ansible 的 Raw 模块更新成 Shell 模块，解决密码过期等场景发生的长时间等待问题 #949
- 更新 TiDB 配置项 `txn_local_latches` 的默认值为 `false`
- 优化 Grafana dashboard 监控项和告警规则 #962 #963 #969
- 新增配置文件检查功能，在部署和升级之前检查配置文件是否正确 #934 #972

14.4.18 TiDB 3.0.3 Release Notes

发版日期：2019 年 8 月 29 日

TiDB 版本：3.0.3

TiDB Ansible 版本：3.0.3

14.4.18.1 TiDB

- SQL 优化器
  - 添加 `opt_rule_blacklist` 表，用于禁用一些逻辑优化规则，比如 `aggregation_eliminate`，`column_prune` 等 #11658
  - 修复 Index join 的 join key 中使用前缀索引或者使用 unsigned 的索引列等于负数时结果不正确的问 #11759
  - 修复 `create ... binding ...` 的 Select 语句中带有 ” 或者 \ 时解析报错的问题 #11726
- SQL 执行引擎
  - 修复 Quote 函数处理 null 值的返回值类型出错的问题 #11619
  - 修复 Max 和 Min 在推导类型时没有去除 NotNullFlag 导致 `ifnull` 结果错误的问题 #11641
  - 修复对字符形式的 Bit 类型数据比较出错的问题 #11660
  - 减少需要顺序读取数据的并发度，以降低 OOM 出现概率 #11679
  - 修复对应含有多个参数的内置函数（如 `if`、`coalesce` 等），在多个参数都为 unsigned 时类型推导不正确的问题 #11621

  - 修复 Div 函数处理 unsigned 的 decimal 类型时与 MySQL 行为不兼容的问题 #11813
  - 修复执行修改 Pump/Drainer 状态的 SQL 时会报 panic 的问题 #11827
  - 修复在 Autocommit = 1 且没有 begin 的时，`select ... for update` 出现 panic 的问题 #11736
  - 修复执行 `set default role` 语句时权限检查出错的问题 #11777
  - 修复执行 `create user` 和`drop user` 语句出现权限检查错误的问题 #11814
  - 修复 `select ... for update` 在构建为 PointGetExecutor 时会重试的问题 #11718
  - 修复 Window function 处理 Partition 时边界出错的问题 #11825
  - 修复 `time` 函数在处理错误格式参数时直接断链接的问题 #11893
  - 修复 Window function 没有检查传入参数的问题 #11705
  - 修复 Explain 查看的 Plan 结果跟真实执行的 Plan 结果不一致的问题 #11186
  - 修复 Window function 内存重复引用导致崩溃或结果不正确的问题 #11823
  - 修复 Slow log 里面 Succ 字段信息错误的问题 #11887

- Server
  - 重命名 `tidb_back_off_weight` 变量为 `tidb_backoff_weight` #11665
  - 更新与当前 TiDB 兼容的最低版本的 TiKV 为 v3.0.0 的信息 #11618
  - 支持 `make testSuite` 来确保测试中的 Suite 都被正确使用 #11685

- DDL
  - 禁止不支持的 Partition 相关的 DDL 的执行，其中包括修改 Partition 类型，同时删除多个 Partition 等 #11373
  - 禁止 Generated Column 的位置在它依赖的列前 #11686
  - 修改添加索引操作中使用的 `tidb_ddl_reorg_worker_cnt` 和 `tidb_ddl_reorg_batch_size` 变量的默认值 #11874

- Monitor
  - Backoff 监控添加类型，且补充之前没有统计到的 Backoff，比如 commit 时遇到的 Backoff #11728

14.4.18.2 TiKV

- 修复 ReadIndex 请求可能由于重复 Context 而无法响应请求的问题 #5256
- 修复 `PutStore` 过早而引起一些调度造成抖动的问题 #5277
- 修复 Region Heartbeat 上报的时间戳不准的问题 #5296
- 剔除 share block cache 信息减少 coredump 文件大小 #5322
- 修复 Region merge 中会引起 TiKV panic 的问题 #5291
- 加快死锁检测器器的 leader 变更检查 #5317
- 使用 grpc env 创建 deadlock 的客户端 #5346
- 添加 `config-check` 检查配置是否正确 #5349
- 修复 ReadIndex 请求在没有 leader 情况下不返回的问题 #5351

14.4.18.3 PD

- `pdctl` 返回成功信息 #1685

14.4.18.4 Tools

- TiDB Binlog
  - 将 Drainer `defaultBinlogItemCount` 默认值从 65536 改为 512，减少 Drainer 启动时 OOM 的情况 #721
  - 优化 Pump server 下线处理逻辑，避免出现 Pump 下线阻塞的问题 #701
- TiDB Lightning
  - 导入时默认过滤系统库 `mysql`，`information_schema`，`performance_schema`，`sys` #225

14.4.18.5 TiDB Ansible

- 优化滚动升级 PD 的操作，提高稳定性 #894
- 移除当前 Grafana 版本不支持的 Grafana Collector 组件 #892
- 更新 TiKV 告警规则 #898
- 修复生成的 TiKV 配置遗漏 `pessimistic-txn` 参数的问题 #911
- 更新 Spark 版本为 2.4.3，同时更新 TiSpark 为兼容该 Spark 的 2.1.4 版本 #913 #918

14.4.19 TiDB 3.0.2 Release Notes

发版日期：2019 年 8 月 7 日

TiDB 版本：3.0.2

TiDB Ansible 版本：3.0.2

14.4.19.1 TiDB

- SQL 优化器
  - 修复当同一张表在查询里出现多次且逻辑上查询结果恒为空时报错“Can’t find column in schema”的问题 #11247
  - 修复了 `TIDB_INLJ` Hint 无法以指定表为 Inner 表构建 IndexJoin 时仍，会强制将其作为 Outer 表构建 IndexJoin，同时 Hint 可能会在不应生效的地方生效的错误，该错误是由于强制选取 IndexJoin 的判断逻辑有误，以及对表别名的处理有误导致的；该错误仅对包含 `TIDB_INLJ` 的查询产生影响 #11362
  - 修复某些情况下（例如 `SELECT IF(1,c,c)FROM t`），查询结果的列名称不正确的问题 #11379
  - 修复 `LIKE` 表达式某些情况下被隐式转换为 0，导致诸如 `SELECT 0 LIKE 'a string'` 返回结果为 `TRUE` 的问题 #11411
  - 支持在 `SHOW` 语句中使用子查询，现在可以支持诸如 `SHOW COLUMNS FROM tbl WHERE FIELDS IN (SELECT 'a')` 的写法 #11459
  - 修复 `outerJoinElimination` 优化规则没有正确处理列的别名，导致找不到聚合函数的相关列而查询报错的问题；改进了优化过程中对别名的解析，以使得优化可以覆盖更多类型的查询 #11377
  - 修复 Window Function 中多个违反语义约束（例如 `UNBOUNDED PRECEDING` 不允许在 Frame 定义的最后）时没有报错的问题 #11543
  - 修复 `ERROR 3593 (HY000): You cannot use the window function FUNCTION_NAME in this context` 报错信息中，`FUNCTION_NAME` 不为小写的问题，导致与 MySQL 不兼容 #11535
  - 修复 Window Function 中 `IGNORE NULLS` 语法尚未实现，但使用时没有报错的问题 #11593

  - 修复优化器对时间类型数据的等值条件代价估算不准确的问题 #11512
  - 支持根据反馈信息对统计信息 Top-N 进行更新 #11507

- SQL 执行引擎

  - 修复 INSERT 函数在参数中包含 NULL 时，返回值不为 NULL 的问题 #11248
  - 修复 ADMIN CHECKSUM 语句在检查分区表时计算结果不正确的问题 #11266
  - 修复 INDEX JOIN 在使用前缀索引时可能结果不正确的问题 #11246
  - 修复 DATE_ADD 函数在进行涉及微秒的日期减法时，没有正确地对日期的小数位数进行对齐导致结果不正确的问题 #11288
  - 修复 DATE_ADD 函数没有正确地对 INTERVAL 中的负数部分处理导致结果不正确的问题 #11325
  - 修复 Mod(%)、Multiple(*) 和 Minus(-) 返回结果为 0 时，在小数位数较多（例如 `select 0.000 % 0.11234500000000000000` ）的情况下与 MySQL 位数不一致的问题 #11251
  - 修复 CONCAT 和 CONCAT_WS 函数在返回结果长度超过 max_allowed_packet 时，没有正确返回 NULL 和 Warning 的问题 #11275
  - 修复 SUBTIME 和 ADDTIME 函数在参数不合法时，没有正确返回 NULL 和 Warning 的问题 #11337
  - 修复 CONVERT_TZ 函数在参数不合法时，没有正确返回 NULL 的问题 #11359
  - EXPLAIN ANALYZE 结果中添加了 MEMORY 列，显示 QUERY 的内存使用 #11418
  - EXPLAIN 结果中，为笛卡尔积 Join 添加了 CARTESIAN 关键字 #11429
  - 修复类型为 FLOAT 和 DOUBLE 的自增列数据不正确的问题 #11385
  - 修复 Dump Pseudo Statistics 时，由于部分信息为 nil 导致 panic 的问题 #11460
  - 修复常量折叠优化导致 SELECT ⋯ CASE WHEN ⋯ ELSE NULL ⋯ 查询结果不正确的问题 #11441
  - 修复 floatStrToIntStr 对诸如 +999.9999e2 的输入没有正确解析的问题 #11473
  - 修复 DATE_ADD 和 DATE_SUB 函数结果超出合法范围时，某些情况下不会返回 NULL 的问题 #11476
  - 修复长字符串转换为整型时，若字符串包含不合法字符，转换结果与 MySQL 不一致的问题 #11469
  - 修复 REGEXP BINARY 函数对大小写敏感，导致与 MySQL 不兼容的问题 #11504
  - 修复 GRANT ROLE 语句在接受 CURRENT_ROLE 时报错的问题；修复 REVOKE ROLE 语句没有能够正确收回 mysql.default_role 权限的问题 #11356
  - 修复执行诸如 SELECT ADDDATE('2008-01-34', -1) 时，Incorrect datetime value Warning 信息的显示格式问题 #11447
  - 修复将 JSON 数据中的 Float 类型字段转为 Int 类型溢出时，报错信息中应当提示 constant ⋯ overflows bigint 而不应当为 constant ⋯ overflows float 的问题 #11534
  - 修复 DATE_ADD 函数接受 FLOAT、DOUBLE 和 DECIMAL 类型的列参数时，没有正确地进行类型转换而导致结果可能不正确的问题 #11527
  - 修复 DATE_ADD 函数中，没有正确处理 INTERVAL 小数部分的符号而导致结果不正确的问题 #11615
  - 修复 Ranger 没有正确处理前缀索引，导致 Index Lookup Join 中包含前缀索引时，查询结果不正确的问题 #11565
  - 修复 NAME_CONST 函数第二个参数为负数时执行会报 Incorrect arguments to NAME_CONST 的问题 #11268
  - 修复一条 SQL 语句在涉及当前时间计算时（例如 CURRENT_TIMSTAMP 或者 NOW），多次取当前时间值，结果与 MySQL 不兼容的问题：现在同一条 SQL 语句中取当前时间时，均使用相同值 #11394
  - 修复了父 Executor Close 出现错误时，没有对 ChildExecutor 调用 Close 的问题，该问题可能导致 KILL 语句失效时，子 ChildExecutor 没有关闭而导致 Goroutine 泄露 #11576

- Server

  - 修复 LOAD DATA 处理 CSV 文件中缺失的 TIMESTAMP 字段时，自动补充的值是 0 不是当前时间戳的问题 #11250

- 修复 SHOW CREATE USER 语句没有正确检查相关权限的问题，以及 SHOW CREATE USER CURRENT_USER() 结果中 USER、HOST 可能不正确的问题 #11229
- 修复在 JDBC 中使用 executeBatch 可能返回结果不正确的问题 #11290
- TiKV Server 在更换端口时，减少 Streaming Client 的报错信息的日志打印 #11370
- 优化 Streaming Client 在重新与 TiKV Server 连接时的逻辑：现在 Streaming Client 不会长时间被 Block #11372
- INFORMATION_SCHEMA.TIDB_HOT_REGIONS 中新增 REGION_ID#11350
- 取消了从 PD API 获取 Region 相关信息时的超时时间，保证在 Region 数量较大时，调用 TiDB API http://{TiDBIP}:10080/regions/hot 不会因为 PD 超时而获取失败 #11383
- 修复 HTTP API 中，与 Region 相关的请求没有返回分区表相关的 Region 问题 #11466
- 做以下改动以降低用户手动验证悲观锁时，操作较慢导致锁超时的概率 #11521：
  * 悲观锁的默认 TTL 时间由 30 秒提升为 40 秒
  * 最大允许的 TTL 时间由 60 秒提升为 120 秒
  * 悲观锁的持续时间改为从第一次 LockKeys 请求时开始计算
- 修改 TiKV Client 中的 SendRequest 函数逻辑：当连接无法建立时，由一直等待改为尽快尝试连接其他 Peer #11531
- 优化 Region Cache：当一个 Store 下线，同时另一个 Store 以同样的地址上线时，将已下线的 Store 标记为失效以尽快在 Cache 中更新 Store 的信息 #11567
- 为 http://{TiDB_ADDRESS:TIDB_IP}/mvcc/key/{db}/{table}/{handle} API 的返回结果添加 Region ID 信息 #11557
- 修复 Scatter Table API 没有对 Range Key 进行转义导致 Scatter Table 不生效的问题 #11298
- 优化 Region Cache：当 Region 所在的 Store 无法访问时，将对应的 Store 信息标记失效以避免对这些 Store 的访问造成查询性能下降 #11498
- 修复了多次 DROP 同名 DATABASE 后，DATABASE 内的表结构仍然能够通过 HTTP API 获取到的错误 #11585

- DDL
  - 修复在非字符串类型且长度为 0 的列建立索引时出错的问题 #11214
  - 禁止对带有外键约束和全文索引的列进行修改（注意：TiDB 仍然仅在语法上支持外键约束和全文索引）#11274
  - 修复并发使用 ALTER TABLE 语句更改的位置和列的默认值时，可能导致列的索引 Offset 出错的问题 #11346
  - 修复解析 JSON 文本的两个问题：
    * ConvertJSONToFloat 中使用 int64 作为 uint64 的中间解析结果，导致精度溢出的问题 #11433
    * ConvertJSONToInt 中使用 int64 作为 uint64 的中间解析结果，导致精度溢出的问题 #11551
  - 禁止 DROP 自增列索引，修复因为 DROP 自增列上的索引导致自增列结果可能出错的问题 #11399
  - 修复以下问题 #11492：
    * 修复显式指定列的排序规则但没有指定字符集时，列的字符集与排序规则不一致的问题
    * 修复 ALTER TABLE ⋯ MODIFY COLUMN 指定的字符集和排序规则冲突时，没有正确报错的问题
    * 修复 ALTER TABLE ⋯ MODIFY COLUMN 指定多次字符集和排序规则时，行为与 MySQL 不兼容的问题
  - 为 TRACE 语句的结果添加子查询的 trace 细节信息 #11458
  - 优化 ADMIN CHECK TABLE 执行性能，大幅降低了语句的执行耗时 #11547
  - 为 SPLIT TABLE ⋯ REGIONS/INDEX 添加了返回结果，结果包含 TOTAL_SPLIT_REGION 和 SCATTER_FINISH_RATIO 展示在超时时间内，切分成功的 Region 数量 #11484

  - 修复 `ON UPDATE CURRENT_TIMESTAMP` 作为列的属性且指定浮点精度时，`SHOW CREATE TABLE` 等语句显示精度不完整的问题 #11591
  - 修复一个虚拟生成列的表达式中含有另一个虚拟生成列时，该列的索引结果不能正确被计算的问题 #11475
  - 修复 `ALTER TABLE ··· ADD PARTITION ···` 语句中，`VALUE LESS THAN` 后不能出现负号的问题 #11581

- Monitor

  - 修复 `TiKVTxnCmdCounter` 监控指标没有注册导致数据没有被收集上报的问题 #11316
  - 为 Bind Info 添加了 `BindUsageCounter`、`BindTotalGauge` 和 `BindMemoryUsage` 监控指标 #11467

14.4.19.2 TiKV

- 修复由于 Raft Log 写入不及时可能导致 TiKV panic 的 bug #5160
- 修复 TiKV panic 后 panic 信息不会写入日志的 bug #5198
- 修复了悲观事务下 Insert 行为可能不正确的 bug #5203
- 降低一部分不需要人工干预的日志输出级别为 INFO #5193
- 提高存储引擎大小监控项的准确程度 #5200
- 提高 tikc-ctl 中 Region size 的准确程度 #5195
- 提高悲观锁死锁检测性能 #5192
- 提高 Titan 存储引擎 GC 性能 #5197

14.4.19.3 PD

- 修复 Scatter Region 调度器不能工作的 bug #1642
- 修复 pd-ctl 中不能进行 merge Region 操作的 bug #1653
- 修复 pd-ctl 中不能进行 remove-tombstone 操作的 bug #1651
- 修复 scan region 不能找到 key 范围相交的 Region 的问题 #1648
- 增加重试机制确保 PD 增加成员成功 #1643

14.4.19.4 Tools

TiDB Binlog

- 增加启动时配置项检查功能，遇到不合法配置项会退出运行并给出错误信息 #687
- Drainer 增加 `node-id` 配置，用于指定固定逻辑 Drainer #684

TiDB Lightning

- 修复 2 个 checksum 同时运行的情况下，`tikv_gc_life_time` 没有正常修改回原本值的问题 #218
- 增加启动时配置项检查功能，遇到不合法配置项会退出运行并给出错误信息 #217

14.4.19.5 TiDB Ansible

- 修复 Disk Performance 监控把 second 作为 ms 的单位错误的问题 #840
- Spark 新增 log4j 日志配置 #841
- 修复在开启了 Binlog 并且设置了 Kafka 或者 ZooKeeper 时导致生成的 Prometheus 配置文件格式错误的问题 #844
- 修复生成的 TiDB 配置文件中遗漏 `pessimistic-txn` 配置参数的问题 #850
- TiDB Dashboard 新增和优化 Metrics #853
- TiDB Dashboard 上每个监控项增加描述 #854
- 新增 TiDB Summary Dashboard，用于更好的查看集群状态和排查问题 #855
- TiKV Dashboard 更新 Allocator Stats 监控项 #857
- 修复 Node Exporter 的告警表达式单位错误的问题 #860
- 更新 tispark jar 包为 v2.1.2 版本 #862
- 更新 Ansible Task 功能描述 #867
- 兼容 TiDB 变更，TiDB Dashboard 更新 Local reader requests 监控项的表达式 #874
- Overview Dashboard 更新 TiKV Memory 监控项的表达式，修复监控显示错误的问题 #879
- 移除 Kafka 模式 Binlog 的支持 #878
- 修复执行 `rolling_update.yml` 操作时，切换 PD Leader 失效的 bug #887

14.4.20 TiDB 3.0.1 Release Notes

发版日期：2019 年 7 月 16 日

TiDB 版本：3.0.1

TiDB Ansible 版本：3.0.1

14.4.20.1 TiDB

- 新增对 `MAX_EXECUTION_TIME` 特性的支持 #11026
- 新增 `tidb_wait_split_region_finish_backoff` Session 变量，用于控制 Region 打散的 Backoff 时间 #11166
- 新增根据负载情况自动调整 Auto ID 分配的步长功能，步长自动调整范围最小 1000，最大 2000000 #11006
- 新增 `ADMIN PLUGINS ENABLE/ADMIN PLUGINS DISABLE` SQL 语句，管理 Plugin 的动态开启或关闭 #11157
- Audit Plugin 新增审记连接功能 #11013
- 修改 Region 打散时的默认行为为等待 PD 调度完成 #11166
- 禁止 Window Function 在 Prepare Plan Cache 中被缓存，避免某些情况下出现结果不正确的问题 #11048
- 禁止使用 Alter 语句修改 Stored Generated Column 的定义 #11068
- 禁止将 Virtual Generated Column 更改为 Stored Generated Column #11068
- 禁止改变带有索引的 Generated Column 的表达式 #11068
- 支持 TiDB 在 ARM64 架构下的编译 #11150
- 支持修改 Database/Table 的 Collate，条件限制为 Database/Table 字符集必须是 UTF8/UTF8MB4 #11086
- 修复 `UPDATE ⋯ SELECT` 语句中，`SELECT` 子查询没有解析到 `UPDATE` 表达式中的列而被误裁剪，导致报错的问题 #11252
- 修复点查时，某列被查询多次而且结果为 NULL 时会 Panic 的问题 #11226
- 修复 `RAND` 函数由于非线程安全的 `rand.Rand` 导致的 Data Race 问题 #11169

- 修复 oom-action="cancel" 时，某些情况下 SQL 内存使用超阈值没有被取消执行，返回结果不正确的问题 #11004
- 修复 MemTracker 未正确清理统计的内存使用值导致 SHOW PROCESSLIST 显示内存使用不为 0 的问题 #10970
- 修复某些情况下整数和非整数比较结果不正确的问题 #11194
- 修复在显式事务中查询对 Table Partition 的查询包含谓词时，查询结果不正确的问题 #11196
- 修复 DDL Job 由于 infoHandle 可能为 NULL 导致 Panic 的问题 #11022
- 修复嵌套聚合查询时，由于被查询列在子查询中没有引用而被误裁剪导致查询结果错误的问题 #11020
- 修复 Sleep 函数响应 Kill 命令不及时的问题 #11028
- 修复 SHOW PROCESSLIST 命令显示的 DB 和 INFO 列与 MySQL 不兼容的问题 #11003
- 修复 skip-grant-table=true 时，FLUSH PRIVILEGES 语句导致系统 Panic 的问题 #11027
- 修复表主键为 UNSIGNED 整数时，FAST ANALYZE 收集主键的统计信息不正确的问题 #11099
- 修复某些情况下 FAST ANALYZE 语句报 “invalid key” Error 的问题 #11098
- 修复 CURRENT_TIMESTAMP 作为列的默认值且指定浮点精度时，SHOW CREATE TABLE 等语句显示精度不完整的问题 #11088
- 修复窗口函数报错时函数名没有小写的问题，兼容 MySQL #11118
- 修复 TiKV Client Batch gRPC 的后台线程 panic 后导致 TiDB 无法正常连接 TiKV 进而无法提供服务的问题 #11101
- 修复 SetVar 方法由于字符串浅拷贝导致设置的变量不正确的问题 #11044
- 修复 INSERT ⋯ ON DUPLICATE 语句作用在 Table Partition 时执行失败报错的问题 #11231
- 悲观锁（实验性特性）
  - 修复悲观锁进行点查且数据为空时，由于行锁未生效导致结果不正确的问题 #10976
  - 修复使用悲观锁查询时由于没有使用 SELECT ⋯ FOR UPDATE 的 TSO 导致查询结果不正确的问题 #11015
  - 修改乐观锁与悲观锁同时使用时，乐观事务遇到悲观锁冲突时，检测行为由立即检测冲突修改为等待，防止锁冲突进一步恶化 #11051

14.4.20.2 TiKV

- 统计信息中新增对 Blob 文件大小的统计 #5060
- 修复由于进程退出未正确清理内存资源导致进程在退出时 core dump 问题 #5053
- 新增与 Titan 引擎相关的所有监控指标 #4772，#4836
- 统计打开文件句柄数量时，新增 Titan 引擎打开文件句柄数量，防止因文件句柄数统计不准确导致系统无文件句柄可用的问题 #5026
- 通过设置 blob_run_mode 来决定是否在某个 CF 上启动 Titan 引擎 #4991
- 修复读操作读不到悲观事务 commit 信息的问题 #5067
- 新增 blob-run-mode 配置参数控制 Titan 引擎的运行模式，取值：normal、 read-only、 fallback #4865
- 提升死锁检测的性能 #5089

14.4.20.3 PD

- 修复热点 Region 调度时，调度限制会自动调整为 0 的问题 #1552
- 新增 enable-grpc-gateway 的配置选项，用于开启 etcd 的 grpc gateway 功能 #1596
- 新增 store-balance-rate、 hot-region-schedule-limit 等与调度器配置相关的统计信息 #1601
- 优化热点 Region 调度策略，调度时跳过缺失副本的 Region，防止多个副本调度到同一个机房 #1609

- 优化 Region Merge 处理逻辑，优先 Merge Region Size 较小的 Region，提升 Region Merge 的速度 #1613
- 优化单次调度热点 Region 的限制值为 64，防止调度任务过多占用系统资源，影响性能 #1616
- 优化 Region 调度策略，新增优先调度 Pending 状态的 Region 功能 #1617
- 修复无法添加 `random-merge` 和 `admin-merge-region` operator 的问题 #1634
- 调整日志中输出 Region 中 Key 的格式为 16 进制，方便用户查看 #1639

14.4.20.4 Tools

TiDB Binlog

- 优化 Pump GC 策略，删除保证未被消费的 Binlog 不被清理的限制，确保资源不会长期占用 #646

TiDB Lightning

- 修正 SQL dump 指明的列名不是小写时导入错误的问题 #210

14.4.20.5 TiDB Ansible

- 新增 `ansible` 命令及其 `jmespath`、`Jinja2` 依赖包的预检查功能 #803，#813
- Pump 新增 `stop-write-at-available-space` 参数，控制当磁盘剩余空间小于该值（默认 10 GiB）时，Pump 停止写入 Binlog #806
- 更新 TiKV 监控中的 IO 监控项，兼容新版本监控组件 #820
- 更新 PD 监控信息，并修复 Disk Performance Dashboard 中 Disk Latency 显示为空的异常 #817
- TiKV Details Dashboard 新增 Titan 监控项 #824

14.4.21 TiDB 3.0 GA Release Notes

发版日期：2019 年 6 月 28 日

TiDB 版本：3.0.0

TiDB Ansible 版本：3.0.0

14.4.21.1 Overview

2019 年 6 月 28 日，TiDB 发布 3.0 GA 版本，对应的 TiDB Ansible 版本为 3.0.0。相比于 V2.1，V3.0.0 版本在以下方面有重要改进：

- 稳定性方面，显著提升了大规模集群的稳定性，集群支持 150+ 存储节点，300+TB 存储容量长期稳定运行。
- 易用性方面有显著的提升，降低用户运维成本，例如：标准化慢查询日志，制定日志文件输出规范，新增 `EXPLAIN ANALYZE`，SQL Trace 功能方便排查问题等。
- 性能方面，与 2.1 相比，TPC-C 性能提升约 4.5 倍，Sysbench 性能提升约 1.5 倍。因支持 View，TPC-H 50G Q15 可正常运行。
- 新功能方面增加了窗口函数、视图（实验特性）、分区表、插件系统、悲观锁（实验特性）、`SQL Plan Management` 等特性。

14.4.21.2 TiDB

- 新功能
  - 新增 Window Function，支持所有 MySQL 8.0 中的窗口函数，包括 NTILE，LEAD，LAG、PERCENT_RANK、NTH_VALUE、CUME_DIST、FIRST_VALUE、LAST_VALUE、RANK、DENSE_RANK、ROW_NUMBER 函数
  - 新增 View 功能（实验特性）
  - 完善 Table Partition 功能：
    * Range Partition
    * Hash Partition
  - 新增插件系统，官方提供 IP 白名单（企业版特性），审记日志（企业版特性）等插件
  - 新增 SQL Plan Management 功能，通过绑定 SQL 执行计划确保查询的稳定性（实验特性）
- SQL 优化器
  - 优化NOT EXISTS 子查询，转化为 Anti Semi Join 提升性能
  - 优化 Outer Join 常量传播，新增 Outer Join 消除优化规则，避免无效计算，提升性能
  - 优化 IN 子查询，先聚合后执行 Inner Join，提升性能
  - 优化 Index Join，适应更多的场景，提升性能
  - 优化 Range Partition 的 Partition Pruning 优化规则，提升性能
  - 优化 _tidb_rowid 查询逻辑，避免全表扫描，提升性能
  - 当过滤条件中包含相关列时，在抽取复合索引的访问条件时尽可能多地匹配索引的前缀列，提升性能
  - 利用列之间的顺序相关性，提升代价估算准确度
  - 基于统计信息的贪心算法及动态规划算法改进了 Join Order，提升多表关联的执行速度
  - 新增 Skyline Pruning，利用规则防止执行计划过于依赖统计信息，提升查询的稳定性
  - 提升单列索引上值为 NULL 时行数估算准确度
  - 新增 FAST ANALYZE，通过在各个 Region 中随机采样避免全表扫描的方式提升统计信息收集性能
  - 新增单调递增的索引列增量 Analyze 功能，提升统计信息收集性能
  - 支持 DO 语句中使用子查询
  - 支持在事务中使用 Index Join
  - 优化 prepare/execute，支持不带参数的 DDL 语句
  - 修改变量 stats-lease 值为 0 时系统的行为，使其自动加载统计
  - 新增导出历史统计信息功能
  - 新增导入导出列的关联性信息功能
- SQL 执行引擎
  - 优化日志输出，EXECUTE 语句输出用户变量，COMMIT 语句输出慢查询日志，方便排查问题
  - 新增 EXPLAIN ANALYZE 功能，提升 SQL 调优易用性
  - 新增 admin show next_row_id 功能，方便获取下一行 ID
  - 新增 JSON_QUOTE、JSON_ARRAY_APPEND、JSON_MERGE_PRESERVE、BENCHMARK、COALESCE、NAME_CONST 6 个内建函数
  - 优化 Chunk 大小控制逻辑，根据查询上下文件动态调整，降低 SQL 执行时间和资源消耗，提升性能
  - 新增 TableReader、IndexReader 和 IndexLookupReader 算子内存追踪控制
  - 优化 Merge Join 算子，使其支持空的 ON 条件
  - 优化单个表列较多时写入性能，提升数倍性能
  - 通过支持逆序扫数据提升 admin show ddl jobs 的性能

  - 新增 `split table region` 语句，手动分裂表的 Region，缓解热点问题
  - 新增 `split index region` 语句，手动分裂索引的 Region，缓解热点问题
  - 新增黑名单禁止下推表达式到 Coprocessor 功能
  - 优化 Expensive Query 日志，在日志中打印执行时间或者使用内存超过阈值的 SQL 查询

- DDL
  - 支持字符集从 `utf8` 转换到 `utf8mb4` 的功能
  - 修改默认字符集从 `utf8` 变为 `utf8mb4`
  - 新增 `alter schema` 语句修改数据库 charset 和 collation 功能
  - 新增 ALTER ALGORITHM INPLACE/INSTANT 功能
  - 新增 `SHOW CREATE VIEW` 功能
  - 新增 `SHOW CREATE USER` 功能
  - 新增快速恢复误删除的表功能
  - 新增动态调整 `ADD INDEX` 的并发数功能
  - 新增 pre_split_regions 选项，在 `CREATE TABLE` 时预先分配 Region，缓解建表后大量写入造成的写热点问题
  - 新增通过 SQL 语句指定表的索引及范围分裂 Region，缓解热点问题
  - 新增 `ddl_error_count_limit` 全局变量，控制 DDL 任务重次数
  - 新增列属性包含 AUTO_INCREMENT 时利用 SHARD_ROW_ID_BITS 打散行 ID 功能，缓解热点问题
  - 优化无效 DDL 元信息存活时间，使集群升级后一段时间 DDL 操作比较慢的情况变短

- 事务
  - 新增悲观事务模型（实验特性）
  - 优化事务处理逻辑，适应更多场景，具体如下：
    * `tidb_disable_txn_auto_retry` 的默认值为 on，即不会重试非自动提交的事务
    * 新增 `tidb_batch_commit` 系统变量控制将事务拆分成多个事务并发执行
    * 新增 `tidb_low_resolution_tso` 系统变量控制批量获取 `tso` 个数，减少事务获取 `tso` 的次数以适应某些数据一致性要求较低的场景
    * 新增 `tidb_skip_isolation_level_check` 变量控制事务检查隔离级别设置为 SERIALIZABLE 时是否报错
    * 修改 `tidb_disable_txn_auto_retry` 系统变量的行为，修改为影响所有的可重试错误

- 权限管理 - 对 `ANALYZE`、`USE`、`SET GLOBAL`、`SHOW PROCESSLIST` 语句进行权限检查 - 新增基于角色的权限访问控制功能 (RBAC)（实验特性）
- Server
  - 优化慢查询日志，具体包括：
    * 重构慢查询日志格式
    * 优化慢查询日志内容
    * 优化查询慢查询日志的方法，通过内存表 `INFORMATION_SCHEMA.SLOW_QUERY`，`ADMIN SHOW SLOW` 语句查询慢查询日志
  - 制定日志格式规范，重构日志系统，方便工具收集分析
  - 新增 SQL 语句管理 TiDB Binlog 服务功能，包括查询状态，开启 TiDB Binlog，维护发送 TiDB Binlog 策略
  - 新增通过 `unix_socket` 方式连接数据库
  - 新增 SQL 语句 `Trace` 功能
  - 新增 `/debug/zip` HTTP 接口，获取 TiDB 实例的信息，方便排查问题

  - 优化监控项，方便排查问题，如下：
    * 新增 high_error_rate_feedback_total 监控项，监控真实数据量与统计信息估算数据量之间的差距
    * 新增 Database 维度的 QPS 监控项
  - 优化系统初始化流程，仅允许 DDL Owner 执行初始化操作，缩短初始化或升级过程中的启动时间
  - 优化 kill query 语句执行逻辑，提升性能，确保资源正确释放
  - 新增启动选项 config-check 检查配置文件合法性
  - 新增 tidb_back_off_weight 系统变量，控制内部出错重试的退避时间
  - 新增 wait_timeout、interactive_timeout 系统变量，控制连接空闲超过变量的值，系统自动断开连接。
  - 新增连接 TiKV 的连接池，减少连接创建时间
- 兼容性
  - 支持 ALLOW_INVALID_DATES SQL mode
  - 支持 MySQL 320 握手协议
  - 支持将 unsigned bigint 列声明为自增列
  - 支持 SHOW CREATE DATABASE IF NOT EXISTS 语法
  - 优化 load data 对 CSV 文件的容错
  - 过滤条件中包含用户变量时谓词不下推，兼容 MySQL Window Function 中使用用户变量行为

14.4.21.3 PD

- 新增从单个节点重建集群的功能
- 将 Region 元信息从 etcd 移到 go-leveldb 存储引擎，解决大规模集群 etcd 存储瓶颈问题
- API
  - 新增 remove-tombstone 接口，用于清理 Tombstone Store
  - 新增 ScanRegions 接口，用于批量查询 Region 信息
  - 新增 GetOperator 接口，用于查询运行中的 Operator
  - 优化 GetStores 接口的性能
- 配置
  - 优化配置检查逻辑，防止配置项错误
  - 新增 enable-two-way-merge，用于控制 Region merge 的方向
  - 新增 hot-region-schedule-limit，用于控制热点 Region 调度速度
  - 新增 hot-region-cache-hits-threshold，连续命中阀值用于判断热点
  - 新增 store-balance-rate 配置，用于控制每分钟产生 balance Region Operator 数量的上限
- 调度器优化
  - 添加 Store Limit 机制限制调度速度，使得速度限制适用于不同规模的集群
  - 添加 waitingOperator 队列，用于优化不同调度器之间资源竞争的问题
  - 支持调度限速功能，主动向 TiKV 下发调度操作，限制单节点同时执行调度任务的个数，提升调度速度

  - Region Scatter 调度不再受 limit 机制限制，提升调度的速度
  - 新增 `shuffle-hot-region` 调度器，解决稳定性测试易用性问题

- 模拟器

  - 新增数据导入场景模拟
  - 新增为 Store 设置不同的心跳间隔的功能

- 其他

  - 升级 etcd，解决输出日志格式不一致，prevote 时选举不出 Leader，Lease 死锁等问题
  - 制定日志格式规范，重构日志系统，方便工具收集分析
  - 新增调度参数，集群 Label 信息，PD 处理 TSO 请求的耗时，Store ID 与地址信息等监控指标

14.4.21.4 TiKV

- 新增分布式 GC 以及并行 Resolve Lock 功能，提升 GC 的性能
- 新增逆向 `raw_scan` 和 `raw_batch_scan` 功能
- 新增多线程 Raftstore 和 Apply 功能，提升单节点内可扩展性，提升单节点内并发处理能力，提升单节点的资源利用率，降低延时，同等压力情况下性能提升 70%
- 新增批量接收和发送 Raft 消息功能，写入密集的场景 TPS 提升 7%
- 新增 Apply snapshot 之前检查 RocksDB level 0 文件的优化，避免产生 Write stall
- 新增 Titan 存储引擎插件，提升 Value 超过 1KiB 时系统的性能，一定程度上缓解写放大问题（实验特性）
- 新增悲观事务模型（实验特性）
- 新增通过 HTTP 方式获取监控信息功能
- 修改 Insert 语义，仅在 Key 不存在的时候 Prewrite 才成功
- 制定日志格式规范，重构日志系统，方便工具收集分析
- 新增配置信息，Key 越界相关的性能监控指标
- RawKV 使用 Local Reader，提升性能
- Engine

  - 优化内存管理，减少 `Iterator Key Bound Option` 的内存分配和拷贝，提升性能
  - 支持多个 column family 共享 block cache，提升资源的利用率

- Server

  - 优化 `batch commands` 的上下文切换开销，提升性能
  - 删除 txn scheduler
  - 新增 read index，GC worker 相关监控项

- RaftStore

  - 新增 hibernate Regions 功能，优化 RaftStore CPU 的消耗（实验特性）
  - 删除 local reader 线程

- Coprocessor

  - 重构计算框架，实现向量化算子、向量化表达式计算、向量化聚合，提升性能
  - 支持为 TiDB EXPLAIN ANALYZE 语句提供算子执行详情
  - 改用 work-stealing 线程池模型，减少上下文切换

14.4.21.5 Tools

- TiDB Lightning

  - 支持数据表重定向同步功能
  - 新增导入 CSV 文件功能
  - 提升 SQL 转 KV 对的性能
  - 单表支持批量导入功能，提升单表导入的性能
  - 支持将大表的数据和索引分别导入，提升 `TiKV-Importer` 导入数据性能
  - 支持对新增文件中缺少 Column 数据时使用 row id 或者列的默认值填充缺少的 column 数据
  - `TiKV-Importer` 支持对 upload SST 到 TiKV 限速功能

- TiDB Binlog

  - Drainer 新增 `advertise-addr` 配置，支持容器环境中使用桥接模式
  - Pump 使用 TiKV GetMvccByKey 接口加快事务状态查询
  - 新增组件之间通讯数据压缩功能，减少网络资源消耗
  - 新增 Arbiter 工具支持从 Kafka 读取 binlog 并同步到 MySQL 功能
  - Reparo 支持过滤不需要被同步的文件的功能
  - 新增同步 Generated column 功能
  - 新增 syncer.sql-mode 配置项，支持采用不同的 SQL mode 解析 DDL
  - 新增 syncer.ignore-table 配置项，过滤不需要被同步的表

- sync-diff-inspector

  - 新增 checkpoint 功能，支持从断点继续校验的功能
  - 新增 only-use-checksum 配置项，控制仅通过计算 checksum 校验数据的一致性
  - 新增采用 TiDB 统计信息以及使用多个 Column 划分 Chunk 的功能，适应更多的场景

14.4.21.6 TiDB Ansible

- 升级监控组件版本到安全的版本

  - Prometheus 从 2.2.1 升级到 2.8.1 版本
  - Pushgateway 从 0.4.0 升级到 0.7.0 版本
  - Node_exporter 从 0.15.2 升级到 0.17.0 版本
  - Alertmanager 从 0.14.0 升级到 0.17.0 版本
  - Grafana 从 4.6.3 升级到 6.1.6 版本
  - Ansible 从 2.5.14 升级到 2.7.11 版本

- 新增 TiKV summary 监控面板，方便查看集群状态
- 新增 TiKV trouble_shooting 监控面板，删除重复项，方便排查问题
- 新增 TiKV details 监控面板，方便调试排查问题
- 新增滚动升级并发检测版本是否一致功能，提升滚动升级性能
- 新增 lightning 部署运维功能
- 优化 `table-regions.py` 脚本，新增按表显示 leader 分布功能
- 优化 TiDB 监控，新增以 SQL 类别显示延迟的监控项
- 修改操作系统版本限制，仅支持 CentOS 7.0 及以上，Red Hat 7.0 及以上版本的操作系统
- 新增预测集群最大 QPS 的监控项，默认隐藏

14.4.22 TiDB 3.0.0-rc.3 Release Notes

发版日期：2019 年 6 月 21 日

TiDB 版本：3.0.0-rc.3

TiDB Ansible 版本：3.0.0-rc.3

14.4.22.1 Overview

2019 年 6 月 21 日，TiDB 发布 3.0.0-rc.3 版本，对应的 TiDB Ansible 版本为 3.0.0-rc.3。相比 3.0.0-rc.2 版本，该版本对系统稳定性、易用性、功能、优化器、统计信息以及执行引擎做了很多改进。

14.4.22.2 TiDB

- SQL 优化器
  - 删除收集虚拟生成列的统计信息功能 #10629
  - 修复点查时主键常量溢出的问题 #10699
  - 修复 `fast analyze` 因使用未初始化的信息导致 panic #10691
  - 修复 prepare `create view` 语句执行过程中因列信息错误导致执行失败的问题 #10713
  - 修复在处理 window function 时列信息未拷贝的问题 #10720
  - 修复 index join 中内表过滤条件在某些情况下的选择率估计错误的问题 #10854
  - 新增变量 `stats-lease` 值为 0 时系统自动加载统计数据功能 #10811
- 执行引擎
  - 修复在 `StreamAggExec` 调用 `Close` 函数资源未正确释放问题 #10636
  - 修复对分区表执行 `show create table` 结果中 `table_option` 与 `partition_options` 顺序不正确问题 #10689
  - 通过支持逆序扫数据提升 `admin show ddl jobs` 的性能 #10687
  - 修复 RBAC 中对 `show grants` 语句带 `current_user` 字段时结果与 MySQL 不兼容的问题 #10684
  - 修复 UUID 在多节点上可能生成重复值的问题 #10712
  - 修复 `explain` 没考虑 `show view` 权限的问题 #10635
  - 新增 `split table region` 语句，手动分裂表的 Region，缓解热点问题 #10765
  - 新增 `split index region` 语句，手动分裂索引的 region 缓解热点问题 #10764
  - 修复连续执行多个 `create user`、`grant` 或 `revoke` 等类似语句执行不正确的问题 #10737
  - 新增黑名单禁止下推表达式到 coprocessor 功能 #10791

  - 新增查询超出内存配置限制时打印 `expensive query` 日志的功能 #10849
  - 新增 `bind-info-lease` 配置项控制修改绑定执行计划的更新时间 #10727
  - 修复因持有 `execdetails.ExecDetails` 指针时 Coprocessor 的资源无法快速释放导致的在大并发场景下 OOM 的问题 #10832
  - 修复某些情况下 `kill` 语句导致的 panic 问题 #10876

- Server
  - 修复 GC 时可能发生的 goroutine 泄露问题 #10683
  - 支持 slow query 里面显示 `host` 信息 #10693
  - 支持循环利用与 TiKV 交互的空闲链接 #10632
  - 修复 RBAC 对开启 `skip-grant-table` 选项的支持问题 #10738
  - 修复 `pessimistic-txn` 配置失效的问题 #10825
  - 修复主动取消的 ticlient 请求还会被重试的问题 #10850
  - 提高在悲观事务和乐观事务冲突情况下的性能 #10881

- DDL
  - 修复在使用 `alter table` 修改 charset 时导致 `blob` 类型改变的问题 #10698
  - 新增列属性包含 `AUTO_INCREMENT` 时利用 `SHARD_ROW_ID_BITS` 打散行 ID 功能，缓解热点问题 #10794
  - 禁止通过 `alter table` 添加存储的生成列 #10808
  - 优化无效 DDL 元信息存活时间，使集群升级后一段时间 DDL 操作比较慢的情况变短 #10795

14.4.22.3 PD

- 新增 `enable-two-way-merge` 配置项，控制合并时仅允许单向合并 #1583
- 新增 `AddLightLearner` 和 `AddLightPeer` 的调度操作，Region Scatter 调度不受 limit 机制限 #1563
- 修复系统启动时因数据可能只进行一副本复制而导致可靠性不足的问题 #1581
- 优化配置检查逻辑，防止配置项错误 #1585
- 调整 `store-balance-rate` 配置的定义为每分钟产生 balance operator 数量的上限 #1591
- 修复 store 可能一直无法产生调度操作的问题 #1590

14.4.22.4 TiKV

- Engine
  - 修复因迭代器未检查状态导致系统生成残缺 snapshot 的问题 #4936
  - 修复在机器异常掉电时由于接收 snapshot 未及时将数据刷新到磁盘导致丢数据的问题 #4937

- Server
  - 新增检查 `block-size` 配置的有效性功能 #4928
  - 新增 read index 相关监控项 #4830
  - 新增 GC worker 相关监控项 #4922

- Raftstore
  - 修复 local reader 的 cache 没有正确清理的问题 #4778
  - 修复进行 transfer leader 和 conf change 时可能导致请求延迟增加的问题 #4734
  - 修复误报 stale command 的问题 #4682

  - 修复 command 可能一直 pending 的问题 #4810
  - 修复 snapshot 文件未及时落盘而导致掉电后文件损坏的问题 #4807，#4850

- Coprocessor
  - 向量计算支持 Top-N #4827
  - 向量计算支持 Stream 聚合 #4786
  - 向量计算中支持 AVG 聚合函 #4777
  - 向量计算中支持 First 聚合函数 #4771
  - 向量计算中支持 SUM 聚合函数 #4797
  - 向量计算中支持 MAX/MIN 聚合函数 #4837
  - 向量计算中支持 Like 表达式 #4747
  - 向量计算中支持 MultiplyDecimal 表达式 #4849
  - 向量计算中支持 BitAnd/BitOr/BitXor 表达式 #4724
  - 向量计算中支持 UnaryNot 表达式 #4808

- Transaction
  - 修复悲观事务中非悲观锁冲突导致出错的问题 #4801，#4883
  - 减少在开启悲观事务后乐观事务的无用计算，提高性能 #4813
  - 新增单语句 rollback，保证在当前语句发生死锁时不需要 rollback 整个事务 #4848
  - 新增悲观事务相关监控项 #4852
  - 支持 `ResolveLockLite` 命令用于轻量级清锁以优化在冲突严重时的性能 #4882

- tikv-ctl
  - 新增 `bad-regions` 命令支持检测更多的异常情况 #4862
  - `tombstone` 命令新增强制执行功能 #4862

- Misc
  - 新增 `dist_release` 编译命令 #4841

14.4.22.5 Tools

- TiDB Binlog
  - 修复 Pump 因写入失败时未检查返回值导致偏移量错误的问题 #640
  - Drainer 新增 `advertise-addr` 配置，支持容器环境中使用桥接模式 #634
  - Pump 新增 `GetMvccByEncodeKey` 函数，加快事务状态查询 #632

14.4.22.6 TiDB Ansible

- 新增预测集群最大 QPS 的监控项（默认隐藏）#f5cfa4d

14.4.23 TiDB 3.0.0-rc.2 Release Notes

发版日期：2019 年 5 月 28 日

TiDB 版本：3.0.0-rc.2

TiDB Ansible 版本：3.0.0-rc.2

14.4.23.1 Overview

2019 年 5 月 28 日，TiDB 发布 3.0.0-rc.2 版本，对应的 TiDB Ansible 版本为 3.0.0-rc.2。相比 3.0.0-rc.1 版本，该版本对系统稳定性、易用性、功能、优化器、统计信息以及执行引擎做了很多改进。

14.4.23.2 TiDB

- SQL 优化器
  - 在更多的场景中支持 Index Join #10540
  - 支持导出历史统计信息 #10291
  - 支持对单调递增的索引列增量 Analyze #10355
  - 忽略 `Order By` 子句中的 NULL 值 #10488
  - 修复精简列信息时逻辑算子 `UnionAll` 的 Schema 信息的计算不正确的问题 #10384
  - 下推 `Not` 操作符时避免修改原表达式 #10363
  - 支持导入导出列的关联性信息 #10573
- 执行引擎
  - 有唯一索引的虚拟生成列可以在 `replace on duplicate key update/insert on duplicate key update` 语句中被正确地处理 #10370
  - 修复 CHAR 列上的扫描范围计算 #10124
  - 修复 `PointGet` 处理负数不正确问题 #10113
  - 合并具有相同窗口名的窗口函数，提高执行效率 #9866
  - 窗口函数中 Range Frame 可以无需 `Order By` 子句 #10496
- Server
  - 修复 TiKV 故障时，TiDB 不断创建与 TiKV 的新连接的问题 #10301
  - `tidb_disable_txn_auto_retry` 不再只影响写入冲突错误，而是影响所有的可重试错误 #10339
  - 不带参数的 DDL 语句可以通过 `prepare/execute` 来执行 #10144
  - 新增 `tidb_back_off_weight` 变量，控制 TiDB 内部 back off 时间的长短 #10266
  - `tidb_disable_txn_auto_retry` 的默认值改为 on，即默认情况下，TiDB 不会重试非自动提交的事务 #10266
  - 修复 RBAC 中对 role 的数据库权限的判断不正确的问题 #10261
  - 支持悲观事务模型（实验性）#10297
  - 降低某些情况下处理锁冲突时的等待时间 #10006
  - 重构 Region cache，增加在 Region 故障时的轮询逻辑 #10256
  - 新增 `tidb_low_resolution_tso` 变量，控制批量获取 `tso` 个数，减少事务获取 `tso` 的次数，以适应某些数据一致性要求较低的场景 #10428
- DDL
  - 修复旧版本的 TiDB 存储的字符集名称大写的问题 #10272
  - 支持 table partition 预分裂 Region 功能，该选项可以在建表时预先分配 table Region，避免建表后大量写入造成的写热点 #10221
  - 修复某些情况下 TiDB 更新版本信息到 PD 不准确的问题 #10324
  - 支持通过 `alter schema` 语句修改数据库 charset 和 collation #10393
  - 支持通过语句按指定表的索引及范围分裂 Region，用于缓解热点问题 #10203
  - 禁止 `alter table` 语句修改 `decimal` 列的精度 #10433

  - 修复 hash partition 中对表达式和函数的约束 #10273
  - 修复某些情况下对含有 partition 的 table 添加索引时引发 TiDB panic 的问题 #10475
  - 添加对某些极端情况下导致 schema 出错的防护功能 #10464
  - 创建 range partition 若有单列或者创建 hash partition 时默认开启分区功能 #9936

14.4.23.3 PD

- 默认开启 Region storage 将 Region 元信息存储到 Region storage 中 #1524
- 修复热点调度受其他调度器抢占的问题 #1522
- 修复 Leader 优先级不生效的问题 #1533
- 新增 `ScanRegions` 的 gRPC 接口 #1535
- 主动下发 operator 加快调度速度 #1536
- 添加 store limit 机制，限制单个 store 的调度速度 #1474
- 修复 `config` 状态不一致的问题 #1476

14.4.23.4 TiKV

- Engine
  - 支持多个 column family 共享 block cache #4563
- Server
  - 移除 txn scheduler #4098
  - 支持悲观锁事务 #4698
- Raftstore
  - 新增 hibernate Regions 特性，减少 raftstore CPU 的消耗 #4591
  - 移除 local reader 线程 #4558
  - 修复 Leader 不回复 Learner `ReadIndex` 请求的问题 #4653
  - 修复在某些情况下 transfer leader 失败的问题 #4684
  - 修复在某些情况下可能发生的脏读问题 #4688
  - 修复在某些情况下 snapshot 少包含数据的问题 #4716
- Coprocessor
  - 新增更多的 RPN 函数
    * `LogicalOr` #4691
    * `LTReal` #4602
    * `LEReal` #4602
    * `GTReal` #4602
    * `GEReal` #4602
    * `NEReal` #4602
    * `EQReal` #4602
    * `IsNull` #4720
    * `IsTrue` #4720
    * `IsFalse` #4720

    * 支持 Int 比较运算 #4625
    * 支持 Decimal 比较运算 #4625
    * 支持 String 比较运算 #4625
    * 支持 Time 比较运算 #4625
    * 支持 Duration 比较运算 #4625
    * 支持 Json 比较运算 #4625
    * 支持 Int 加法运算 #4733
    * 支持 Real 加法运算 #4733
    * 支持 Decimal 加法运算 #4733
    * 支持 Int 求余函数 #4727
    * 支持 Real 求余函数 #4727
    * 支持 Decimal 求余函数 #4727
    * 支持 Int 减法运算 #4746
    * 支持 Real 减法运算 #4746
    * 支持 Decimal 减法运算 #4746

14.4.23.5 Tools

- TiDB Binlog
  - Drainer 增加下游同步延迟监控项 checkpoint_delay #594
- TiDB Lightning
  - 支持数据库合并，数据表合并同步功能 #95
  - 新增 KV 写入失败重试机制 #176
  - 配置项 table-concurrency 默认值修改为 6 #175
  - 减少必要的配置项，tidb.port 和 tidb..pd-addr 支持自动获取 #173

14.4.24 TiDB 3.0.0-rc.1 Release Notes

发版日期：2019 年 5 月 10 日

TiDB 版本：3.0.0-rc.1

TiDB Ansible 版本：3.0.0-rc.1

14.4.24.1 Overview

2019 年 5 月 10 日，TiDB 发布 3.0.0-rc.1 版，对应的 TiDB Ansible 版本为 3.0.0-rc.1。相比 3.0.0-beta.1 版本，该版本对系统稳定性、易用性、功能、优化器、统计信息以及执行引擎做了很多改进。

14.4.24.2 TiDB

- SQL 优化器
  - 利用列之间的顺序相关性提升代价估算准确度，并提供启发式参数 tidb_opt_correlation_exp_factor 用于控制在相关性无法被直接用于估算的场景下对索引扫描的偏好程度。#9839

  - 当过滤条件中包含相关列时，在抽取复合索引的访问条件时尽可能多地匹配索引的前缀列。#10053
  - 用动态规划决定连接的执行顺序，当参与连接的表数量不多于 `tidb_opt_join_reorder_threshold` 时启用。#8816
  - 在构造 Index Join 的的内表中，以复合索引作为访问条件时，尽可能多地匹配索引的前缀列。#8471
  - 提升对单列索引上值为 NULL 的行数估算准确度。#9474
  - 在逻辑优化阶段消除聚合函数时特殊处理 GROUP_CONCAT ，防止产生错误的执行结果。#9967
  - 当过滤条件为常量时，正确地将它下推到连接算子的子节点上。#9848
  - 在逻辑优化阶段列剪裁时特殊处理一些函数，例如 RAND() ，防止产生和 MySQL 不兼容的执行结果。#10064
  - 支持 FAST ANALYZE，通过`tidb_enable_fast_analyze` 变量控制。该特性通过用对 Region 进行采样取代扫描整个 region 的方式加速统计信息收集。#10258
  - 支持 SQL PLAN MANAGEMENT。该特性通过对 SQL 进行执行计划绑定，以确保执行稳定性。该特性目前处于测试阶段，仅支持对 SELECT 语句使用绑定的执行计划，不建议在生产场景中直接使用。#10284

- 执行引擎

  - 支持对 TableReader、 IndexReader 和 IndexLookupReader 算子进行内存追踪控制。#10003
  - 在慢日志中展示更多 COPROCESSOR 端执行任务相关细节。如 COPROCESSOR 任务数，平均/最长/90% 执行/等待时间，执行/等待时间最长的 TiKV 地址等。#10165
  - 支持 PREPARE 不含占位符的 DDL 语句。#10144

- Server

  - TiDB 启动时，只允许 DDL owner 执行 bootstrap #10029
  - 新增 `tidb_skip_isolation_level_check` 变量控制检查隔离级别设置为 SERIALIZABLE 时不报错 #10065
  - 在慢日志中，将隐式提交的时间与 SQL 执行时间融合在一起 #10294
  - RBAC 权限管理
    * 支持 SHOW GRANT #10016
    * 支持 SET DEFAULT ROLE #9949
    * 支持 GRANT ROLE #9721
  - 修正了插件退出时导致 TiDB 退出的问题 #9889
  - 修正只读语句被错误地放到事务历史中的问题 #9723
  - kill 语句可以更快的结束 SQL 的执行，并快速释放资源 #9844
  - 增加启动选项 `config-check` 来检查配置文件的合法性 #9855
  - 修正非严格模式下对于写入 NULL 字段的合法性检查 #10161

- DDL

  - 为 CREATE TABLE 添加了 pre_split_regions 选项，该选项可以在建表时预先分配 Table Region，避免建表后大量写入造成的写热点 #10138
  - 优化了部分 DDL 语句的执行性能 #10170
  - FULLTEXT KEY 新增不支持全文索引的 warning #9821
  - 修正了旧版本 TiDB 中，UTF8 和 UTF8MB4 编码的兼容性问题 #9820
  - 修正了一个表的 shard_row_id_bits 的潜在 BUG #9868
  - 修正了 ALTER TABLE Charset 后，Column Charset 不会跟随变化的 BUG #9790
  - 修正了使用 BINARY/BIT 作为 Column Default Value 时，SHOW COLUMN 可能出错的 BUG #9897
  - 修正了 SHOW FULL COLUMNS 语句中，CHARSET / COLLATION 显示的兼容性问题 #10007

- 现在 SHOW COLLATIONS 语句只会列出 TiDB 所实际支持的 COLLATIONS #10186

14.4.24.3 PD

- 升级 ETCD 版本 #1452
  - 统一 etcd 的日志格式与 pd server 一致
  - 修复 prevote 可能无法选出 Leader 的问题
  - 快速 drop 掉会失败的 propose 和 read 请求，减少阻塞后面的请求时间
  - 修复 Lease 的死锁问题
- 修复 store 读热点的 keys 统计不正确问题 #1487
- 支持从单一 PD 节点强制重建 PD 集群 #1485
- 修复 Scatter Region 产生无效 Operator Step 的问题 #1482
- 修复 Region Merge Operator 超时时间过短的问题 #1495
- 热点调度使用高优先级 #1492
- 添加 PD server 端处理 TSO 请求的耗时 Metrics #1502
- 添加相对应的 Store ID 和 Address 到 store 相关的 Metrics #1506
- 支持 GetOperator 服务 #1477
- 修复 Heartbeat stream 下发送 error 找不到 store 的问题 #1521

14.4.24.4 TiKV

- Engine
  - 修复读流量统计不准确问题 #4436
  - 修复 prefix extractor panic 的问题 #4503
  - 优化内存管理，减少 `Iterator Key Bound Option` 的内存分配和拷贝 #4537
  - 修复 Merge Region 时未考虑 Learner log gap 造成的 panic 问题 #4559
  - 支持不同的 `column families` 共享 `block cache` #4612
- Server
  - 减少 `batch commands` 的上下文切换开销 #4473
  - 检查 seek iterator status 的合法性 #4470
- RaftStore
  - 可配置化 `properties index distance` #4517
- Coprocessor
  - 新增 batch index scan executor #4419
  - 新增向量化 evaluation 框架 #4322
  - 新增 batch 执行器统计框架 #4433
  - 构建 RPN expression 时检查 max column 以防止 evaluation 阶段 column offset 越界的问题 #4481
  - 实现 `BatchLimitExecutor` #4469
  - ReadPool 使用 `tokio-threadpool` 替换原本的 `futures-cpupool`，减少 context switch #4486
  - 新增 batch 聚合框架 #4533
  - 新增 `BatchSelectionExecutor` #4562

  - 实现 batch aggression function AVG #4570
  - 实现 RPN function `LogicalAnd` #4575
- Misc
  - 支持选用 tcmalloc 为内存分配器 #4370

14.4.24.5 Tools

- TiDB Binlog
  - 修复 unsigned int 类型的主键列的 binlog 数据为负数，造成同步出错中断的问题 #573
  - 删除下游是 pb 时的压缩选项，修改下游名字 pb 成 file #559
  - Pump 新增 storage.sync-log 配置项，支持 Pump 本地存储异步刷盘 #509
  - Pump 和 Drainer 之间通讯支持流量压缩 #495
  - Drainer 新增 syncer.sql-mode 配置项，支持使用不同 sql-mode 解析 DDL query #511
  - Drainer 新增 syncer.ignore-table 配置项，支持过滤不需要同步的表 #520
- Lightning
  - 使用 row id 或者列的默认值填充 dump 文件中缺少的 column 数据 #170
  - Importer 修复部分 SST 导入失败依然返回导入成功的 bug #4566
  - Importer 支持 upload SST 到 TiKV 限速 #4412
  - Lightning 优化导入表的顺序，按照表的数据大小顺序进行导入，减少导入过程中大表执行 checksum 和 Analyze 对集群的影响，并且提高 Checksum 和 Analyze 的成功率 #156
  - 提升 Lightning encode SQL 性能，性能提升 50%，直接解析数据源文件内容成 TiDB 的 types.Datum，省去 KV encoder 的多余解析工作 #145
  - 日志格式改为 Unified Log Format #162
  - 新增一些命令行选项，即使缺少配置文件也能使用。#157
- 数据同步对比工具 (sync-diff-inspector)
  - 支持 checkpoint，记录校验状态，重启后从上次进度继续校验 #224
  - 增加配置项 only-use-checksum，只通过计算 checksum 来检查数据是否一致 #215

14.4.24.6 TiDB Ansible

- TiKV 监控变更以及更新 Ansible、 Grafana、 Prometheus 版本 #727
  - summary 监控适用于用户查看集群状态
  - trouble_shooting 监控适用于 DBA 排查问题
  - details 监控适用于开发分析问题
- 修复下载 Kafka 版本 Binlog 失败的 BUG #730
- 修改操作系统版本限制，仅支持 CentOS 7.0 及以上，Red Hat 7.0 及以上版本的操作系统 #733
- 滚动升级时的版本检测改为多并发 #736
- 更新 README 中文档链接#740
- 移除重复的 TiKV 监控项，新增 trouble shooting 监控项 #735
- 优化 `table-regions.py` 脚本，按表显示 leader 分布 #739
- 更新 drainer 配置文件 #745
- 优化 TiDB 监控，新增以 SQL 类别显示延迟的监控项 #747
- 更新 Lightning 配置文件，新增 tidb_lightning_ctl 脚本 #1e946f8

14.4.25 TiDB 3.0.0 Beta.1 Release Notes

发版日期：2019 年 3 月 26 日

TiDB 版本：3.0.0-beta.1

TiDB Ansible 版本：3.0.0-beta.1

14.4.25.1 Overview

2019 年 03 月 26 日，TiDB 发布 3.0.0 Beta.1 版，对应的 TiDB Ansible 版本为 3.0.0 Beta.1。相比 3.0.0 Beta 版本，该版本对系统稳定性、易用性、功能、优化器、统计信息以及执行引擎做了很多改进。

14.4.25.2 TiDB

- SQL 优化器
  - 支持使用 Sort Merge Join 计算笛卡尔积 #9032
  - 支持 Skyline Pruning，用一些规则来防止执行计划过于依赖统计信息 #9337
  - 支持 Window Functions
    * NTILE #9682
    * LEAD 和 LAG #9672
    * PERCENT_RANK #9671
    * NTH_VALUE #9596
    * CUME_DIST #9619
    * FIRST_VALUE 和 LAST_VALUE #9560
    * RANK 和 DENSE_RANK #9500
    * RANGE FRAMED #9450
    * ROW FRAMED #9358
    * ROW NUMBER #9098
  - 增加了一类统计信息，表示列和 handle 列之间顺序的相关性 #9315
- SQL 执行引擎
  - 增加内建函数
    * JSON_QUOTE #7832
    * JSON_ARRAY_APPEND #9609
    * JSON_MERGE_PRESERVE #8931
    * BENCHMARK #9252
    * COALESCE #9087
    * NAME_CONST #9261
  - 根据查询上下文优化 Chunk 大小，降低 SQL 执行时间和集群的资源消耗 #6489
- 权限管理
  - 支持 SET ROLE 和 CURRENT_ROLE #9581
  - 支持 DROP ROLE #9616
  - 支持 CREATE ROLE #9461

- Server
  - 新增 `/debug/zip` HTTP 接口，获取当前 TiDB 实例的信息 #9651
  - 支持使用 `show pump status/show drainer status` 语句查看 Pump/Drainer 状态 #9456
  - 支持使用 SQL 语句在线修改 Pump/Drainer 状态 #9789
  - 支持给 SQL 文本加上 HASH 指纹，方便追查慢 SQL #9662
  - 新增 `log_bin` 系统变量，默认：`0`，管理 binlog 开启状态，当前仅支持查看状态 #9343
  - 支持通过配置文件管理发送 binlog 策略 #9864
  - 支持通过内存表 `INFORMATION_SCHEMA.SLOW_QUERY` 查询慢日志 #9290
  - 将 TiDB 显示的 MySQL Version 从 5.7.10 变更为 5.7.25 #9553
  - 统一日志格式规范，利于工具收集分析
  - 增加监控项 `high_error_rate_feedback_total`，记录实际数据量与统计信息估算数据量差距情况 #9209
  - 新增 Database 维度的 QPS 监控项, 可以通过配置项开启 #9151
- DDL
  - 增加`ddl_error_count_limit`全局变量，默认值：`512`，限制 DDL 任务重试次数，超过限制次数会取消出错的 DDL #9295
  - 支持 ALTER ALGORITHM INPLACE/INSTANT #8811
  - 支持 `SHOW CREATE VIEW` 语句 #9309
  - 支持 `SHOW CREATE USER` 语句 #9240

14.4.25.3 PD

- 统一日志格式规范，利于工具收集分析
- 模拟器
  - 支持不同 store 可采用不同的心跳间隔时间 #1418
  - 添加导入数据的场景 #1263
- 热点调度可配置化 #1412
- 增加 store 地址为维度的监控项，代替原有的 Store ID #1429
- 优化 `GetStores` 开销，加快 Region 巡检周期 #1410
- 新增删除 Tombstone Store 的接口 #1472

14.4.25.4 TiKV

- 优化 Coprocessor 计算执行框架，完成 TableScan 算子，单 TableScan 即扫表操作性能提升 5% ~ 30%
  - 实现行 `BatchRows` 和列 `BatchColumn` 的定义 #3660
  - 实现 `VectorLike` 使得编码和解码的数据能够用统一的方式访问 #4242
  - 定义 `BatchExecutor` 接口，实现将请求转化为 `BatchExecutor` 的方法 #4243
  - 实现将表达式树转化成 RPN 格式 #4329
  - TableScan 算子实现为 Batch 方式，通过向量化计算加速计算 #4351
- 统一日志格式规范，利于工具收集分析
- 支持 Raw Read 接口使用 Local Reader 进行读 #4222
- 新增配置信息的 Metrics #4206

- 新增 Key 越界的 Metrics #4255
- 新增碰到扫越界错误时 Panic 或者报错选项 #4254
- 增加 Insert 语义，只有在 Key 不存在的时候 Prewrite 才成功，消除 Batch Get #4085
- Batch System 使用更加公平的 batch 策略 #4200
- tikv-ctl 支持 Raw scan #3825

14.4.25.5 Tools

- TiDB Binlog
  - 新增 Arbiter 工具支持从 Kafka 读取 binlog 同步到 MySQL
  - Reparo 支持过滤不需要同步的文件
  - 支持同步 generated column
- Lightning
  - 支持禁用 TiKV periodic Level-1 compaction，当 TiKV 集群为 2.1.4 或更高时，在导入模式下会自动执行 Level-1 compaction #119，#4199
  - 根据 `table_concurrency` 配置项限制 import engines 数量，默认值：16，防止过多占用 importer 磁盘空间 #119
  - 支持保存中间状态的 SST 到磁盘，减少内存使用 #4369
  - 优化 TiKV-Importer 导入性能，支持将大表的数据和索引分离导入 #132
  - 支持 CSV 文件导入 #111
- 数据同步对比工具 (sync-diff-inspector)
  - 支持使用 TiDB 统计信息来划分对比的 chunk #197
  - 支持使用多个 column 来划分对比的 chunk #197

14.4.26 TiDB 3.0 Beta Release Notes

2019 年 1 月 19 日，TiDB 发布 3.0 Beta 版，TiDB Ansible 相应发布 3.0 Beta 版本。相比 2.1 版本，该版本对系统稳定性、优化器、统计信息以及执行引擎做了很多改进。

14.4.26.1 TiDB

- 新特性
  - 支持 View
  - 支持窗口函数
  - 支持 Range 分区
  - 支持 Hash 分区
- SQL 优化器
  - 重新支持聚合消除的优化规则 #7676
  - 优化 `NOT EXISTS` 子查询，将其转化为 Anti Semi Join #7842
  - 添加 `tidb_enable_cascades_planner` 变量以支持新的 Cascades 优化器。目前 Cascades 优化器尚未实现完全，默认关闭 #7879

  - 支持在事务中使用 Index Join #7877
  - 优化 Outer Join 上的常量传播，使得对 Join 结果里和 Outer 表相关的过滤条件能够下推过 Outer Join 到 Outer 表上，减少 Outer Join 的无用计算量，提升执行性能 #7794
  - 调整投影消除的优化规则到聚合消除之后，消除掉冗余的 `Project` 算子 #7909
  - 优化 `IFNULL` 函数，当输入参数具有非 NULL 的属性的时候，消除该函数 #7924
  - 支持对 `_tidb_rowid` 构造查询的 Range，避免全表扫，减轻集群压力 #8047
  - 优化 `IN` 子查询为先聚合后做 Inner Join 并，添加变量 `tidb_opt_insubq_to_join_and_agg` 以控制是否开启该优化规则并默认打开 #7531
  - 支持在 `DO` 语句中使用子查询 #8343
  - 添加 Outer Join 消除的优化规则，减少不必要的扫表和 Join 操作，提升执行性能 #8021
  - 修改 `TIDB_INLJ` 优化器 Hint 的行为，优化器将使用 Hint 中指定的表当做 Index Join 的 Inner 表 #8243
  - 更大范围的启用 `PointGet`，使得当 Prepare 语句的执行计划缓存生效时也能利用上它 #8108
  - 引入贪心的 Join Reorder 算法，优化多表 Join 时 Join 顺序选择的问题 #8394
  - 支持 View #8757
  - 支持 Window Function #8630
  - 当 `TIDB_INLJ` 未生效时，返回 warning 给客户端，增强易用性 #9037
  - 支持根据过滤条件和表的统计信息推导过滤后数据的统计信息的功能 #7921
  - 增强 Range Partition 的 Partition Pruning 优化规则 #8885

- SQL 执行引擎

  - 优化 Merge Join 算子，使其支持空的 `ON` 条件 #9037
  - 优化日志，打印执行 `EXECUTE` 语句时使用的用户变量 #7684
  - 优化日志，为 `COMMIT` 语句打印慢查询信息 #7951
  - 支持 `EXPLAIN ANALYZE` 功能，使得 SQL 调优过程更加简单 #7827
  - 优化列很多的宽表的写入性能 #7935
  - 支持 `admin show next_row_id` #8242
  - 添加变量 `tidb_init_chunk_size` 以控制执行引擎使用的初始 Chunk 大小 #8480
  - 完善 `shard_row_id_bits`，对自增 ID 做越界检查 #8936

- `Prepare` 语句

  - 对包含子查询的 `Prepare` 语句，禁止其添加到 `Prepare` 语句的执行计划缓存中，确保输入不同的用户变量时执行计划的正确性 #8064
  - 优化 `Prepare` 语句的执行计划缓存，使得当语句中包含非确定性函数的时候，该语句的执行计划也能被缓存 #8105
  - 优化 `Prepare` 语句的执行计划缓存，使得 `DELETE/UPDATE/INSERT` 的执行计划也能被缓存 #8107
  - 优化 `Prepare` 语句的执行计划缓存，当执行 `DEALLOCATE` 语句时从缓存中剔除对应的执行计划 #8332
  - 优化 `Prepare` 语句的执行计划缓存，通过控制其内存使用以避免缓存过多执行计划导致 TiDB OOM 的问题 #8339
  - 优化 `Prepare` 语句，使得 `ORDER BY/GROUP BY/LIMIT` 子句中可以使用 “?” 占位符 #8206

- 权限管理

  - 增加对 `ANALYZE` 语句的权限检查 #8486
  - 增加对 `USE` 语句的权限检查 #8414
  - 增加对 `SET GLOBAL` 语句的权限检查 #8837
  - 增加对 `SHOW PROCESSLIST` 语句的权限检查 #7858

- Server

  - 支持了对 SQL 语句的 Trace 功能 #9029
  - 支持了插件框架 #8788
  - 支持同时使用 unix_socket 和 TCP 两种方式连接数据库 #8836
  - 支持了系统变量 interactive_timeout #8573
  - 支持了系统变量 wait_timeout #8346
  - 提供了变量 tidb_batch_commit，可以按语句数将事务分解为多个事务 #8293
  - 支持 ADMIN SHOW SLOW 语句，方便查看慢日志 #7785
- 兼容性
  - 支持了 ALLOW_INVALID_DATES 这种 SQL mode #9027
  - 提升了 load data 对 CSV 文件的容错能力 #9005
  - 支持了 MySQL 320 握手协议 #8812
  - 支持将 unsigned bigint 列声明为自增列 #8181
  - 支持 SHOW CREATE DATABASE IF NOT EXISTS 语法 #8926
  - 当过滤条件中包含用户变量时不对其进行谓词下推的操作，更加兼容 MySQL 中使用用户变量模拟 Window Function 的行为 #8412
- DDL
  - 支持快速恢复误删除的表 #7937
  - 支持动态调整 ADD INDEX 的并发数 #8295
  - 支持更改表或者列的字符集到 utf8/utf8mb4 #8037
  - 默认字符集从 utf8 变为 utf8mb4 #7965
  - 支持 RANGE PARTITION #8011

14.4.26.2 Tools

- TiDB Lightning
  - 大幅优化 SQL 转 KV 的处理速度 #110
  - 对单表支持 batch 导入，提高导入性能和稳定性 #113

14.4.26.3 PD

- 增加 RegionStorage 单独存储 Region 元信息 #1237
- 增加 shuffle hot region 调度 #1361
- 增加调度参数相关 Metrics #1406
- 增加集群 Label 信息相关 Metrics #1402
- 增加导入数据场景模拟 #1263
- 修复 Leader 选举相关的 Watch 问题 #1396

14.4.26.4 TiKV

- 支持了分布式 GC #3179
- 在 Apply snapshot 之前检查 RocksDB level 0 文件，避免产生 Write stall #3606
- 支持了逆向 raw_scan 和 raw_batch_scan #3742

- 更好的夏令时支持 #3786
- 支持了使用 HTTP 方式获取监控信息 #3855
- 支持批量方式接收和发送 Raft 消息 #3931
- 引入了新的存储引擎 Titan #3985
- 升级 gRPC 到 v1.17.2 #4023
- 支持批量方式接收客户端请求和发送回复 #4043
- 多线程 Apply #4044
- 多线程 Raftstore #4066

14.5 v2.1

14.5.1 TiDB 2.1.19 Release Notes

发版日期：2019 年 12 月 27 日

TiDB 版本：2.1.19

TiDB Ansible 版本：2.1.19

14.5.1.1 TiDB

- SQL 优化器
  - 优化 `select max(_tidb_rowid)from t` 的场景，避免全表扫 #13294
  - 修复当查询语句中赋予用户变量错误的值且将谓词下推后导致错误的输出结果 #13230
  - 修复更新统计信息时可能存在数据竞争，导致统计信息不准确的问题 #13690
  - 修复 UPDATE 语句中同时包含子查询和 stored generated column 时结果错误的问题；修复 UPDATE 语句中包含不同数据库的两个表名相同时，UPDATE 执行报错的问题 #13357
  - 修复 PhysicalUnionScan 算子没有正确设置统计信息，导致查询计划可能选错的问题 #14134
  - 移除 `minAutoAnalyzeRatio` 约束使自动 ANALYZE 更及时 #14013
  - 当 WHERE 子句上有 UNIQUE KEY 的等值条件时，估算行数应该不大于 1 #13385
- SQL 执行引擎
  - 修复 ConvertJSONToInt 中使用 int64 作为 uint64 的中间解析结果，导致精度溢出的问题 #13036
  - 修复查询中包含 SLEEP 函数时（例如 `select 1 from (select sleep(1))t;`），由于列裁剪导致查询中的 `sleep(1)` 失效的问题 #13039
  - 通过实现在 INSERT ON DUPLICATE UPDATE 语句中复用 Chunk 来降低内存开销 #12999
  - 给 `slow_query` 表添加事务相关的信息段 #13129，如下：
    * Prewrite_time
    * Commit_time
    * Get_commit_ts_time
    * Commit_backoff_time
    * Backoff_types
    * Resolve_lock_time
    * Local_latch_wait_time
    * Write_key

    * Write_size
    * Prewrite_region
    * Txn_retry
  - 修复 UPDATE 语句中包含子查询时转换子查询出现的错误和当 UPDATE 的 WHERE 条件中包含子查询时更新失败的问题 #13120
  - 支持在分区表上执行 ADMIN CHECK TABLE #13143
  - 修复 ON UPDATE CURRENT_TIMESTAMP 作为列的属性且指定浮点精度时，SHOW CREATE TABLE 等语句显示精度不完整的问题 #12462
  - 修复在 DROP/MODIFY/CHANGE COLUMN 时没有检查外键导致执行 SELECT * FROM information_schema.KEY_COLUMN_USAGE 语句时发生 panic 的问题 #14162
  - 修复 TiDB 开启 Streaming 后返回数据可能重复的问题 #13255
  - 修复夏令时导致的 “无效时间格式” 问题 #13624
  - 修复整型数据被转换为无符号 Real/Decimal 类型时，精度可能丢失的问题 #13756
  - 修复 Quote 函数处理 null 值时返回值类型出错的问题 #13681
  - 修复从字符串解析日期时，由于使用 golang time.Local 本地时区导致解析结果的时区不正确的问题 #13792
  - 修复 builtinIntervalRealSig 的实现中，由于 binSearch 方法不会返回 error，导致最终结果可能不正确的问题 #13768
  - 修复 INSERT 语句在进行字符串类型到浮点类型转换时，可能会报错的问题 #14009
  - 修复 sum(distinct) 函数输出结果不正确的问题 #13041
  - 修复由于对 jsonUnquoteFunction 函数的返回类型长度赋值不正确的值，导致在 union 中同位置数据上进行 cast 转换时会截断数据的问题 #13645
  - 修复由于权限检查过于严格导致设置密码失败的问题 #13805
- Server
  - 修复 KILL CONNECTION 可能出现 goroutine 泄漏的问题 #13252
  - 新增通过 HTTP API 的 info/all 接口获取所有 TiDB 节点的 binlog 状态功能 #13188
  - 修复在 Windows 上 build TiDB 项目失败的问题 #13650
  - 新增 server-version 配置项来控制修改 TiDB server 版本的功能 #13904
  - 修复通过 Go1.13 版本编译的二进制程序 plugin 不能正常运行的问题 #13527
- DDL
  - 新增创建表时如果表包含 COLLATE 则列的 COLLATE 使用表的 COLLATE #13190
  - 新增创建表时限制索引名字的长度的功能 #13311
  - 修复 rename table 时未检查表名长度的问题 #13345
  - 新增 BIT 列的宽度范围检查的功能 #13511
  - 优化 change/modify column 的输出的错误信息，让人更容易理解 #13798
  - 修复执行 drop column 操作且下游 Drainer 还没有执行此 drop column 操作时，下游可能会收到不带此列的 DML 的问题 #13974

14.5.1.2 TiKV

- Raftstore
  - 修复 Region merge 和应用 Compact log 过程中系统若有重启，当重启时由于未正确设置 is_merging 的值导致系统 panic 的问题 #5884

- Importer
  - 取消 gRPC 的消息长度限制 #5809

14.5.1.3 PD

- 提升获取 Region 列表的 HTTP API 性能 #1988
- 升级 etcd，修复 etcd PreVote 无法选出 leader 的问题（升级后无法降级）#2052

14.5.1.4 Tools

- TiDB Binlog
  - 优化通过 binlogctl 输出的节点状态信息 #777
  - 修复当 Drainer 过滤配置为 `nil` 时 panic 的问题 #802
  - 优化 Pump 的 `Graceful` 退出方式 #825
  - 新增 Pump 写 binlog 数据时更详细的监控指标 #830
  - 优化 Drainer 在执行 DDL 后刷新表结构信息的逻辑 #836
  - 修复 Pump 在没有收到 DDL 的 commit binlog 时该 binlog 被忽略的问题 #855

14.5.1.5 TiDB Ansible

- TiDB 服务 `Uncommon Error OPM` 监控项更名为 `Write Binlog Error` 并增加对应的告警 #1038
- 升级 TiSpark 版本为 2.1.8 #1063

14.5.2 TiDB 2.1.18 Release Notes

发版日期：2019 年 11 月 4 日

TiDB 版本：2.1.18

TiDB Ansible 版本：2.1.18

14.5.2.1 TiDB

- SQL 优化器
  - 修复 Feedback 切分查询范围出错的问题 #12172
  - 修复点查中权限检查不正确的问题 #12341
  - 将 Limit 算子下推到 `IndexLookUpReader` 执行逻辑中，优化 `select ... limit ... offset ...` 的执行性能 #12380
  - 支持在 `ORDER BY`、`GROUP BY` 和 `LIMIT OFFSET` 中使用参数 #12514
  - 修复 partition 表上的 IndexJoin 返回错误结果的问题 #12713
  - 修复 TiDB 中 `str_to_date` 函数在日期字符串和格式化字符串不匹配的情况下，返回结果与 MySQL 不一致的问题 #12757
  - 修复当查询条件中包含 cast 函数时 outer join 被错误转化为 inner join 的问题 #12791

  - 修复 AntiSemiJoin 的 join 条件中错误的表达式传递 #12800

- SQL 执行引擎

  - 修复时间取整不正确的问题 (如 2019-09-11 11:17:47.999999666 应该被取整到 2019-09-11 11:17:48) #12259
  - 修复 PREPARE 语句类型没有记录在监控中的问题 #12329
  - 修复 FROM_UNIXTIME 在检查 NULL 值时 panic 的错误 #12572
  - 修复 YEAR 类型数据插入非法年份时，结果为 NULL 而不是 0000 的兼容性问题 #12744
  - 改进 AutoIncrement 列隐式分配时的行为，与 MySQL 自增锁的默认模式 (“consecutive” lock mode) 保持一致：对于单行 Insert 语句的多个自增 AutoIncrement ID 的隐式分配，TiDB 保证分配值的连续性。该改进保证 JDBC getGeneratedKeys() 方法在任意场景下都能得到正确的结果。#12619
  - 修复当 HashAgg 作为 Apply 子节点时查询 hang 住的问题 #12769
  - 修复逻辑表达式 AND / OR 在涉及类型转换时返回错误结果的问题 #12813

- Server

  - 修复 KILL TIDB QUERY 语法对 SLEEP() 语句无效的问题 #12159
  - 修复 AUTO INCREMENT 分配 MAX int64 和 MAX uint64 没有报错的问题 #12210
  - 修复日志级别设置为 ERROR 时，慢日志不会被记录的问题 #12373
  - 将缓存 100 个 Schema 变更相关的表信息调整成 1024 个，且支持通过 tidb_max_delta_schema_count 系统变量修改 #12515
  - 将 SQL 的统计方式开始时间由“开始执行”改为“开始编译”，使得 SQL 性能统计更加准确 #12638
  - 在 TiDB 日志中添加 set session autocommit 的记录 #12568
  - 将 SQL 的开始时间记录在 SessionVars 中，避免计划执行时，该时间被重置 #12676
  - 在 Order By/Group By/Limit Offset 字句中支持 ? 占位符 #12514
  - 慢日志中添加 Prev_stmt 字段，用于最后一条语句是 COMMIT 时输出前一条语句 #12724
  - 当一个显式提交的事务 COMMIT 时出错，在日志中记录 COMMIT 前一条语句 #12747
  - 优化在 TiDB Server 执行 SQL 时，对前一条语句的保存方式以提升性能 #12751
  - 修复 skip-grant-table=true 时，FLUSH PRIVILEGES 语句导致系统 Panic 的问题 #12816
  - 将 AutoID 的最小申请步长从 1000 增加为 30000，避免短时间大量写入时频繁请求 AutoID 造成性能瓶颈 #12891
  - 修复 Prepared 语句在 TiDB 发生 panic 时错误日志中未打印出错 SQL 的问题 #12954
  - 修复 COM_STMT_FETCH 慢日志时间记录和 MySQL 不一致问题 #12953
  - 当遇到写冲突时，在报错信息中添加错误码，以方便对冲突原因进行诊断 #12878

- DDL

  - 为避免误操作，TiDB 默认不再允许删除列的 AUTO INCREMENT 属性，当确实需要删除时，请更改系统变量 tidb_allow_remove_auto_inc；相关文档请见系统变量和语法 #12146
  - 支持 Create Table 语句中建唯一索引时带多个 Unique #12469
  - 修复 CreateTable 语句中指定外键约束时，外键表在没有指定 Database 时未能使用主表的 Database 导致报错的问题 #12678
  - 修复 ADMIN CANCEL DDL JOBS 时报 invalid list index 错的问题 #12681

- Monitor

  - Backoff 监控添加类型，且补充之前没有统计到的 Backoff，比如 commit 时遇到的 Backoff #12326
  - 添加统计 Add Index 操作进度的监控 #12389

14.5.2.2 PD

- 修复 pd-ctl `--help` 命令输出内容 #1772

14.5.2.3 Tools

- TiDB Binlog
  - 修复 `ALTER DATABASE` 相关 DDL 会导致 Drainer 异常退出的问题 #770
  - 支持对 Commit binlog 查询事务状态信息，提升同步效率 #761
  - 修复当 Drainer 的 `start_ts` 大于 Pump 中最大的 `commit_ts` 时候有可能引起 Pump panic 的问题 #759

14.5.2.4 TiDB Ansible

- TiDB Binlog 增加 queue size 和 query histogram 监控项 #952
- 更新 TiDB 告警表达式 #961
- 新增配置文件检查功能，部署或者更新前会检查配置是否合理 #973
- TiDB 新增加索引速度监控项 #987
- 更新 TiDB Binlog 监控 Dashboard，兼容 4.6.3 版本的 Grafana #993

14.5.3 TiDB 2.1.17 Release Notes

发版日期：2019 年 9 月 11 日

TiDB 版本：2.1.17

TiDB Ansible 版本：2.1.17

- 新特性
  - TiDB 的 `SHOW TABLE REGIONS` 语法新增 `WHERE` 条件子句
  - TiKV、PD 新增 `config-check` 功能，用于配置项检查
  - pd-ctl 新增 `remove-tombstone` 命令，支持清理 tombstone store 记录
  - Reparo 新增 `worker-count` 和 `txn-batch` 配置项，用于控制恢复速率
- 改进提升
  - PD 优化调度流程，支持主动下发调度
  - TiKV 优化启动流程，减少重启节点带来的抖动
- 行为变更
  - TiDB 慢日志中的 `start ts` 由最后一次重试的时间改为第一次执行的时间
  - TiDB 慢日志中的 `Index_ids` 字段替换为 `Index_names` 字段，提升慢日志易用性
  - TiDB 配置文件中添加 `split-region-max-num` 参数，用于调整 `SPLIT TABLE` 语法允许的最大 Region 数量，默认配置下，允许的数量由 1,000 增加至 10,000

14.5.3.1 TiDB

- SQL 优化器
  - 修复 EvalSubquery 在构建 Executor 出现错误时，错误信息没有被正确返回的问题 #11811
  - 修复 Index Lookup Join 中，外表的行数大于一个 batch 时，查询的结果可能不正确的问题；扩大 Index Lookup Join 的作用范围：可以使用 UnionScan 作为 IndexJoin 的子节点 #11843
  - 针对统计信息的反馈过程可能产生失效 Key 的情况，SHOW STAT_BUCKETS 语句现在增加了失效 Key 的显示，例如：invalid encoded key flag 252 #12098
- SQL 执行引擎
  - 修复 CAST 函数在进行数值类型转换时，首先将数值转换为 UINT 导致一些结果不正确的问题（例如，select cast(13835058000000000000 as double)）#11712
  - 修复 DIV 运算的被除数为 DECIMAL 类型且运算含有负数时，运算结果可能不正确的问题 #11812
  - 添加 ConvertStrToIntStrict 函数，修复执行 SELECT/EXPLAIN 语句时，一些字符串转换 INT 类型结果与 MySQL 不兼容的问题 #11892
  - 修复使用 EXPLAIN ... FOR CONNECTION 语法时，stmtCtx 没有正确设置导致 Explain 结果可能不正确的问题 #11978
  - 修复 unaryMinus 函数，当 Int 结果溢出时，返回结果类型没有为 Decimal 导致与 MySQL 不兼容的问题 #11990
  - 修复 LOAD DATA 语句执行时，计数顺序导致的 last_insert_id() 可能不正确的问题 #11994
  - 修复用户显式、隐式混合写入自增列数据时，last_insert_id() 可能不正确的问题 #12001
  - 修复一个 JSON_UNQUOTE 函数兼容性问题：只有在双引号（"）内的值需要 Unquote，例如 SELECT JSON_UNQUOTE("\\\\") 应当为 “\\”（不进行 Unquote）#12096
- Server
  - TiDB 事务重试时，记录在慢日志中的 start ts 由最后一次重试的时间改为第一次执行的时间 #11878
  - 在 LockResolver 中添加事务的 Key 数量：当 Key 数量较少时，可以避免对整个 Region 的 Scan 操作，减小清锁的代价 #11889
  - 修复慢日志中，succ 字段值可能不正确的问题 #11886
  - 将慢日志中的 Index_ids 字段替换为 Index_names 字段，提升慢日志易用性 #12063
  - 修复 Duration 内容中包含 - 时（例如 select time(‘--’)），TiDB 解析为 EOF Error 导致连接断开的错误 #11910
  - 改进 RegionCache：当一个 Region 失效时，它将会更快地从 RegionCache 中移除，减少向该 Region 发送请求的个数 #11931
  - 修复 oom-action = "cancel" 时，当 Insert Into ··· Select 语句发生 OOM，OOM Panic 没有被正确处理而导致连接断开的问题 #12126
- DDL
  - 为 tikvSnapshot 添加逆序扫描接口用于高效地查询 DDL History Job，使用该接口后 ADMIN SHOW DDL JOBS 的执行时间有明显降低 #11789
  - 改进 CREATE TABLE ... PRE_SPLIT_REGION 的语义：当指定 PRE_SPLIT_REGION = N 时，将预切分的 Region 个数由 2^(N-1) 改为 2^N #11797
  - 根据线上负载与 Add Index 相互影响测试，调小 Add Index 后台工作线程的默认参数以避免对线上负载造成较大影响 #11875

  - 改进 SPLIT TABLE 语法的行为：当使用 SPLIT TABLE ... REGIONS N 对 Region 切分时，会生成 N 个数据 Region 和 1 个索引 Region #11929
  - 在配置文件中添加 split-region-max-num 参数，使得 SPLIT TABLE 语法允许的最大 Region 数量可调整，该参数默认值 10000 #12080
  - 修复写 binlog 时，CREATE TABLE 语句中 PRE_SPLIT_REGIONS 部分没有被注释，导致语句不能被下游 MySQL 解析的问题 #12121
  - SHOW TABLE ··· REGIONS 和 SHOW TABLE .. INDEX ··· REGIONS 语法新增 WHERE 条件子句 #12124

- Monitor

  - 增加监控指标 connection_transient_failure_count，用于统计 tikvclient 的 gRPC 连接错误数量 #12092

14.5.3.2 TiKV

- 解决某些情况下 Region 内 key 个数统计不准的问题 #5415
- TiKV 新增 config-check 选项，用于检查 TiKV 配置项是否合法 #5391
- 优化启动流程，减少重启节点带来的抖动 #5277
- 优化某些情况下解锁的流程，加速事务解锁 #5339
- 优化 get_txn_commit_info 的流程，加速事务提交 #5062
- 简化 Raft 相关的 log #5425
- 解决在某些情况下 TiKV 异常退出的问题 #5441

14.5.3.3 PD

- PD 新增 config-check 选项，用于检查 PD 配置项是否合法 #1725
- pd-ctl 新增 remove-tombstone 命令，支持清理 tombstone store 记录 #1705
- 支持主动下发 Operator，加快调度速度 #1686

14.5.3.4 Tools

- TiDB Binlog

  - Reparo 新增 worker-count 和 txn-batch 配置项，用于控制恢复速率 #746
  - Drainer 优化内存使用，提升同步执行效率 #735
  - Pump 修复有时候无法正常下线的 bug #739
  - Pump 优化 LevelDB 处理逻辑，提升 GC 执行效率 #720

- TiDB Lightning

  - 修复从 checkpoint 点重新导入可能会导致 tidb-lightning 崩溃的 bug #239

14.5.3.5 TiDB Ansible

- 更新 Spark 版本为 2.4.3，同时更新 TiSpark 为兼容该 Spark 的 2.2.0 版本 #914，#919
- 修复了当远程机器密码过期时长时间连接等待的问题 #937，#948

14.5.4 TiDB 2.1.16 Release Notes

发版日期：2019 年 8 月 15 日

TiDB 版本：2.1.16

TiDB Ansible 版本：2.1.16

14.5.4.1 TiDB

- SQL 优化器
  - 修复时间列上的等值条件 Row Count 估算不准确的问题 #11526
  - 修复 `TIDB_INLJ` Hint 不生效或者对非指定的表生效的问题 #11361
  - 将查询中的 NOT EXISTS 由 OUTER JOIN 实现方式改为 ANTI JOIN，便于找到更优执行计划 #11291
  - 支持在 SHOW 语句中使用子查询，现在可以支持诸如 `SHOW COLUMNS FROM tbl WHERE FIELDS IN (SELECT 'a')` 的写法 #11461
  - 修复常量折叠优化导致 `SELECT ··· CASE WHEN ··· ELSE NULL ...` 查询结果不正确的问题 #11441
- SQL 执行引擎
  - 修复函数 DATE_ADD 在 INTERVAL 为负的情况下结果错误的问题 #11616
  - 修复 DATE_ADD 函数接受 FLOAT、DOUBLE 和 DECIMAL 类型的参数时，没有正确地进行类型转换而导致结果可能不正确的问题 #11628
  - 修复 CAST(JSON AS SIGNED) 出现 OVERFLOW 时错误信息不准确的问题 #11562
  - 修复在关闭 Executor 的过程中，子节点关闭返回错误时其他子节点未关闭的问题 #11598
  - 支持 SPLIT TABLE 语句返回切分成功的 REGION 数量，并且当部分 REGION SCATTER 在超时未完成调度时，不再返回错误，而是返回完成调度的比例 #11487
  - 修复 `REGEXP BINARY` 函数对大小写敏感，与 MySQL 不兼容的问题 #11505
  - 修复 DATE_ADD / DATE_SUB 结果中 YEAR 小于 0 或大于 65535 时溢出导致结果没有正确返回 NULL 值的问题 #11477
  - 慢查询表中添加用于表示是否执行成功的 Succ 字段 #11412
  - 修复一条 SQL 语句在涉及当前时间计算时（例如 CURRENT_TIMESTAMP 或者 NOW），多次取当前时间值，结果与 MySQL 不兼容的问题：现在同一条 SQL 语句中取当前时间时，均使用相同值 #11392
  - 修复 AUTO INCREMENT 列未处理 FLOAT / DOUBLE 的问题 #11389
  - 修复 CONVERT_TZ 函数在参数不合法时，没有正确返回 NULL 的问题 #11357
  - 修复 PARTITION BY LIST 报错的问题（仅添加语法支持，TiDB 执行时候会作为普通表创建并提供提示信息）#11236
  - 修复 `Mod(%)`、`Multiple(*)` 和 `Minus(-)` 返回结果为 0 时，在小数位数较多（例如 `select 0.000 % 0.11234500000000000000`）的情况下与 MySQL 位数不一致的问题 #11353
- Server
  - 修复插件在 OnInit 回调中获取 Domain 为 NULL 的问题 #11426
  - 修复当 Schema 删除后，依然可以通过 HTTP 接口获取该 Schema 中表信息的问题 #11586
- DDL
  - 禁止 DROP 自增列索引，修复因为 DROP 自增列上的索引导致自增列结果可能出错的问题 #11402
  - 修复列和表使用不同的 CHARSET 和 COLLATE 创建表和修改表时，列的字符集不正确的问题 #11423

  - 修复并行执行 “alter table … set default…” 和其他修改此列信息的 DDL，可能导致此列的结构出错的问题 #11374
  - 修复当 Generated column A 依赖 Generated column B 时，使用 A 创建索引，数据回填失败的问题 #11538
  - 提升 ADMIN CHECK TABLE 的速度 #11538

14.5.4.2 TiKV

- 访问正在关闭的 TiKV Region 时返回 Close 错误 #4820
- 支持逆向 `raw_scan` 和逆向 `raw_batch_scan` 接口 #5148

14.5.4.3 Tools

- TiDB Binlog
  - Drainer 增加 `ignore-txn-commit-ts` 配置项，用于跳过执行某些事务语句 #697
  - 增加启动时配置项检查功能，遇到不合法配置项会退出运行并给出错误信息 #708
  - Drainer 增加 `node-id` 配置，用于指定固定逻辑 Drainer #706
- TiDB Lightning
  - 修复 2 个 checksum 同时运行的情况下，`tikv_gc_life_time` 没有正常修改回原本值的问题 #224

14.5.4.4 TiDB Ansible

- Spark 新增 log4j 日志配置 #842
- 更新 tispark jar 包为 v2.1.2 版本 #863
- 修复了 TiDB Binlog 使用 Kafka 或者 ZooKeeper 时导致生成的 Prometheus 配置文件格式错误的问题 #845
- 修复执行 `rolling_update.yml` 操作时，切换 PD Leader 失效的 Bug #888
- 优化滚动升级 PD 节点的逻辑，先升级 Follower 再升级 Leader，提高稳定性 #895

14.5.5 TiDB 2.1.15 Release Notes

发版日期：2019 年 7 月 18 日

TiDB 版本：2.1.15

TiDB Ansible 版本：2.1.15

14.5.5.1 TiDB

- 修复 DATE_ADD 函数处理微秒时由于没有对齐导致结果不正确的问题 #11289
- 修复 DELETE 语句中，字符串列中的空值与 FLOAT/INT 做比较时会报错的问题 #11279
- 修复 INSERT 函数参数有 NULL 值时，未正确返回 NULL 值的问题 #11249
- 修复在非字符串类型且长度为 0 的列建立索引时出错的问题 #11215
- 新增 SHOW TABLE REGIONS 的语句，支持通过 SQL 查询表的 Region 分布情况 #11238
- 修复 UPDATE … SELECT 语句因 SELECT 子查询中使用投影消除来优化规则所导致的报错 #11254

- 新增 ADMIN PLUGINS ENABLE/DISABLE SQL 语句，支持通过 SQL 动态开启/关闭 Plugin #11189
- Audit Plugin 新增审记连接功能 #11189
- 修复点查时，某列被查询多次而且结果为 NULL 时会 Panic 的问题 #11227
- 新增 tidb_scatter_region 配置项，控制创建表时是否开启自己打散 Record Region #11213
- 修复 RAND 函数由于非线程安全的 rand.Rand 导致的 Data Race 问题 #11170
- 修复某些情况下整数和非整数比较结果不正确的问题 #11191
- 支持修改 Database/Table 的 Collate，条件限制为 Database/Table 字符集必须是 UTF8/UTF8MB4 #11085
- 修复 CURRENT_TIMESTAMP 作为列的默认值且指定浮点精度时，SHOW CREATE TABLE 等语句显示精度不完整的问题 #11087

14.5.5.2 TiKV

- 统一日志格式 #5083
- 提高 region approximate size/key 在极端情况下的准确性，提升调度准确度 #5085

14.5.5.3 PD

- 统一日志格式 #1625

14.5.5.4 Tools

TiDB Binlog

- 优化 Pump GC 策略，去掉了未被在线 drainer 消费到的 binlog 保证不清理的限制 #663

TiDB Lightning

- 修复 SQL dump 指明的列名不是小写时导入错误的问题 #210

14.5.5.5 TiDB Ansible

- TiDB Dashboard 新增 parse duration 和 compile duration 监控项，用于监测 SQL 语句解析耗时和执行计划编译耗时 #815

14.5.6 TiDB 2.1.14 Release Notes

发版日期：2019 年 7 月 4 日

TiDB 版本：2.1.14

TiDB Ansible 版本：2.1.14

14.5.6.1 TiDB

- 修复某些情况下列裁剪导致查询结果不正确的问题 #11019
- 修复 `show processlist` 中 db 和 `info` 列信息显示有误的问题 #11000
- 修复 `MAX_EXECUTION_TIME` 作为 SQL hint 和全局变量在某些情况下不生效的问题 #10999
- 支持根据负载情况自动调整 Auto ID 分配的步长 #10997
- 修复 SQL 查询结束时 `MemTracker` 统计的 DistSQL 内存信息未正确清理的问题 #10971
- `information_schema.processlist` 表中新增 `MEM` 列用于描述 Query 的内存使用情况 #10896
- 新增全局系统变量 `max_execution_time`，用于控制查询的最大执行时间 10940
- 修复使用未支持的聚合函数导致 TiDB panic 的问题 #10911
- 新增 `load data` 语句失败后自动回滚最后一个事务功能 #10862
- 修复 TiDB 超过内存配额的行为配置为 CANCEL 时，某些情况下 TiDB 返回结果不正确的问题 #11016
- 禁用 TRACE 语句，避免 TiDB panic 问题 #11039
- 新增 `mysql.expr_pushdown_blacklist` 系统表，控制动态开启/关闭 TiDB 下推到 Coprocessor 的函数 #10998
- 修复 `ANY_VALUE` 函数在 `ONLY_FULL_GROUP_BY` 模式下不生效的问题 #10994
- 修复给字符串类型的用户量赋值时因未深度拷贝导致赋值不正确的问题 #11043

14.5.6.2 TiKV

- 优化 Raftstore 消息处理中对空回调的处理流程，避免发送不必要的消息 #4682

14.5.6.3 PD

- 调整当读取到无效配置项时日志信息输出的级别，由 Error 调整为 Warning #1577

14.5.6.4 Tools

TiDB Binlog

- Reparo
  - 新增 `safe-mode` 配置项，开启后支持导入重复的数据 #662
- Pump
  - 新增 `stop-write-at-available-space` 配置项，限制 Binlog 空间保留的大小 #659
  - 修复 LevelDB L0 文件个数为 0 时 GC 有时不生效的问题 #648
  - 优化 log 文件删除的算法，加速释放空间 #648
- Drainer
  - 修复下游 TiDB BIT 类型列更新失败的问题 #655

14.5.6.5 TiDB Ansible

- 新增 `ansible` 命令及其 `jmespath`、`Jinja2` 依赖包的预检查功能 #807
- Pump 新增 `stop-write-at-available-space` 参数，当磁盘剩余空间小于该值（默认 10 GiB）时，Pump 停止写入 Binlog #807

14.5.7 TiDB 2.1.13 Release Notes

发版日期：2019 年 6 月 21 日

TiDB 版本：2.1.13

TiDB Ansible 版本：2.1.13

14.5.7.1 TiDB

- 新增列属性包含 AUTO_INCREMENT 时利用 SHARD_ROW_ID_BITS 打散行 ID 功能，缓解热点问题 #10788
- 优化无效 DDL 元信息存活时间，缩短集群升级后恢复 DDL 操作正常执行所需的时间 #10789
- 修复因持有 `execdetails.ExecDetails` 指针时 Coprocessor 的资源无法快速释放导致的在大并发场景下 OOM 的问题 #10833
- 新增 `update-stats`配置项，控制是否更新统计信息 #10772
- 新增 3 个 TiDB 特有语法，支持预先切分 Region，解决热点问题：
  - 新增 Table Option PRE_SPLIT_REGIONS 选项 #10863
  - 新增 `SPLIT TABLE table_name INDEX index_name` 语法 #10865
  - 新增 `SPLIT TABLE [table_name] BETWEEN (min_value...)AND (max_value...)REGIONS [region_num ]` 语法 #10882
- 修复某些情况下 KILL 语句导致的 panic 问题 #10879
- 增强 ADD_DATE 在某些情况下跟 MySQL 的兼容性 #10718
- 修复 index join 中内表过滤条件在某些情况下的选择率估计错误的问题 #10856

14.5.7.2 TiKV

- 修复因迭代器未检查状态导致系统生成残缺 snapshot 的问题 #4940
- 新增检查 `block-size` 配置的有效性功能 #4930

14.5.7.3 Tools

TiDB Binlog

- 修复 Pump 因写入失败时未检查返回值导致偏移量错误问题 #640
- Drainer 新增 `advertise-addr` 配置，支持容器环境中使用桥接模式 #634

14.5.8 TiDB 2.1.12 Release Notes

发版日期：2019 年 6 月 13 日

TiDB 版本：2.1.12

TiDB Ansible 版本：2.1.12

14.5.8.1 TiDB

- 修复在使用 feedback 时由于类型不匹配导致进程 panic 的问题 #10755
- 修复某些情况下改变字符集导致 BLOB 类型变成 TEXT 类型的问题 #10745
- 修复某些情况下在事务中的 GRANT 操作误报 “Duplicate Entry” 错误的问题 #10739
- 提升以下功能跟 MySQL 的兼容性
  - DAYNAME 函数 #10732
  - MONTHNAME 函数 #10733
  - EXTRACT 函数在处理 MONTH 的时候支持零值 #10702
  - DECIMAL 类型转换成 TIMESTAMP 或者 DATETIME 类型 #10734
- 修改表的字符集时，同步修改列的字符集 #10714
- 修复某些情况下 DECIMAL 转换成浮点数的溢出问题 #10730
- 修复 TiDB 跟 TiKV 在 gRPC 最大封包设置不一致导致的某些超大封包报 “grpc: received message larger than max” 错误的问题 #10710
- 修复某些情况下 ORDER BY 没有过滤 NULL 值导致的 panic 问题 #10488
- 修复 UUID 函数的返回值，在多机器情况可能出现重复的问题 #10711
- CAST(-num as datetime) 的返回值由错误变更为 NULL 值 #10703
- 修复某些情况下 unsigned 列直方图遇到 signed 越界的问题 #10695
- 修复统计信息的 feedback 遇到 bigint unsigned 主键时处理不正确导致读数据时报错的问题 #10307
- 修复分区表某些情况下 Show Create Table 结果显示不正确的问题 #10690
- 修复在某些关联子查询上聚合函数 GROUP_CONCAT 计算不正确的问题 #10670
- 修复某些情况下 slow query 内存表在解析慢日志的时候导致的显示结果错乱的问题 #10776

14.5.8.2 PD

- 修复极端情况下 etcd Leader 选举阻塞的问题 #1576

14.5.8.3 TiKV

- 修复极端条件下 Leader 迁移过程中 Region 不可用的问题 #4799
- 修复在机器异常掉电时由于接收 snapshot 未及时将数据刷新到磁盘导致丢数据的问题 #4850

14.5.9 TiDB 2.1.11 Release Notes

发版日期：2019 年 6 月 03 日

TiDB 版本：2.1.11

TiDB Ansible 版本：2.1.11

14.5.9.1 TiDB

- 修复 delete 多表 join 的结果时使用错误 schema 的问题 #10595
- 修复 CONVERT() 函数返回错误的字段类型的问题 #10263

- 更新统计信息时合并不重叠的反馈信息 #10569
- 修复 unix_timestamp()-unix_timestamp(now()) 计算错误的问题 #10491
- 修复 period_diff 与 MySQL 8.0 不兼容的问题 #10501
- 收集统计信息的时候，忽略 Virtual Column，避免异常报错 #10628
- 支持 SHOW OPEN TABLES 语句 #10374
- 修复某些情况下导致的 goroutine 泄露问题 #10656
- 修复某些情况下设置 tidb_snapshot 变量时间格式解析出错的问题 #10637

14.5.9.2 PD

- 修复因为 balance-region 可能会导致热点 Region 没有机会调度的问题 #1551
- 将热点相关调度的优先级改为高优先级 #1551
- 新增配置项 hot-region-schedule-limit 控制同时进行热点调度任务的数量及新增 hot-region-cache-hits-threshold 控制判断是否为热点 Region #1551

14.5.9.3 TiKV

- 修复在仅有一个 leader，learner 时，learner 读到空 index 的问题 #4751
- 将 ScanLock 和 ResolveLock 放在高优先级线程池中处理，减少对普通优先级命令的影响 #4791
- 同步所有收到的 snapshot 的文件 #4811

14.5.9.4 Tools

- TiDB Binlog
  - 新增 GC 删数据限速功能，避免因为删除数据导致 QPS 降低的问题 #620

14.5.9.5 TiDB Ansible

- 新增 Drainer 参数 #760

14.5.10 TiDB 2.1.10 Release Notes

发版日期：2019 年 5 月 22 日

TiDB 版本：2.1.10

TiDB Ansible 版本：2.1.10

14.5.10.1 TiDB

- 修复在使用 tidb_snapshot 读取历史数据的时候，某些异常情况导致的表结构不正确 #10359
- 修复 NOT 函数在某些情况下导致的读取结果错误的问题 #10363
- 修复 Generated Column 在 Replace 或者 Insert on duplicate update 语句中的错误行为 #10385
- 修复 BETWEEN 函数在 DATE/DATETIME 类型比较的一个 bug #10407

- 修复使用 SLOW_QUERY 表查询慢日志时，单行慢日志长度过长导致的报错 #10412
- 修复某些情况下 DATETIME 和 INTERVAL 相加的结果跟 MySQL 不一致的问题 #10416，#10418
- 增加闰年二月的非法时间的检查 #10417
- 内部的初始化操作限制只在 DDL Owner 中执行，避免了初始化集群的时候出现的大量冲突报错 #10426
- 修复 DESC 在输出时间戳列的默认值为 `default current_timestamp on update current_timestamp` 时跟 MySQL 不兼容的问题 #10337
- 修复 Update 语句中权限检查出错的问题 #10439
- 修复 CHAR 类型的列在某些情况下 RANGE 计算错误导致的错误结果的问题 #10455
- 避免 ALTER SHARD_ROW_ID_BITS 缩小 shard bits 位数在极低概率下，可能导致的数据错误 #9868
- 修复 ORDER BY RAND() 不返回随机数字的问题 #10064
- 禁止 ALTER 语句修改 DECIMAL 的精度 #10458
- 修复 TIME_FORMAT 函数与 MySQL 的兼容问题 #10474
- 检查 PERIOD_ADD 中参数的合法性 #10430
- 修复非法的 YEAR 字符串在 TiDB 中的表现跟 MySQL 不兼容的问题 #10493
- 支持 ALTER DATABASE 语法 #10503
- 修复 SLOW_QUERY 内存表在慢语句没有；的情况下报错的问题 #10536
- 修复某些情况下 Partitioned Table 的表 Add index 操作没有办法取消的问题 #10533
- 修复在某些情况下无法抓住内存使用太多导致 OOM 的问题 #10545
- 增强 DDL 操作改写表元信息的安全性 #10547

14.5.10.2 PD

- 修复 Leader 优先级不生效的问题 #1533

14.5.10.3 TiKV

- 拒绝在最近发生过成员变更的 Region 上执行 transfer leader，防止迁移失败 #4684
- Coprocessor metrics 上添加 priority 标签 #4643
- 修复 transfer leader 中可能发生的脏读问题 #4724
- 修复某些情况下 CommitMerge 导致 TiKV 不能重启的问题 #4615
- 修复 unknown 的日志 #4730

14.5.10.4 Tools

- TiDB Lightning
  - 新增 TiDB Lightning 发送数据到 importer 失败时进行重试 #176
- TiDB Binlog
  - 优化 Pump storage 组件 log，以利于排查问题 #607

14.5.10.5 TiDB Ansible

- 更新 TiDB Lightning 配置文件，新增 `tidb_lightning_ctl` 脚本 #d3a4a368

14.5.11 TiDB 2.1.9 Release Notes

发版日期：2019 年 5 月 6 日

TiDB 版本：2.1.9

TiDB Ansible 版本：2.1.9

14.5.11.1 TiDB

- 修复 MAKETIME 函数在 unsigned 类型溢出时的兼容性 #10089
- 修复常量折叠在某些情况下导致的栈溢出 #10189
- 修复 Update 在某些有别名的情况下权限检查的问题 #10157 #10326
- 追踪以及控制 DistSQL 中的内存使用 #10197
- 支持指定 collation 为 utf8mb4_0900_ai_ci #10201
- 修复主键为 Unsigned 类型的时候，MAX 函数结果错误的问题 #10209
- 修复在非 Strict SQL Mode 下可以插入 NULL 值到 NOT NULL 列的问题 #10254
- 修复 COUNT 函数在 DISTINCT 有多列的情况下结果错误的问题 #10270
- 修复 LOAD DATA 解析不规则的 CSV 文件时候 Panic 的问题 #10269
- 忽略 Index Lookup Join 中内外 join key 类型不一致的时候出现的 overflow 错误 #10244
- 修复某些情况下错误判定语句为 point-get 的问题 #10299
- 修复某些情况下时间类型未转换时区导致的结果错误问题 #10345
- 修复 TiDB 字符集在某些情况下大小写比较不一致的问题 #10354
- 支持控制算子返回的行数 #9166
  - Selection & Projection #10110
  - StreamAgg & HashAgg #10133
  - TableReader & IndexReader & IndexLookup #10169
- 慢日志改进
  - 增加 SQL Digest 用于区分同类 SQL #10093
  - 增加慢语句使用的统计信息的版本信息 #10220
  - 输出语句内存使用量 #10246
  - 调整 Coprocessor 相关信息的输出格式，让其能被 pt-query-digest 解析 #10300
  - 修复慢语句中带有 # 字符的问题 #10275
  - 增加一些信息的列到慢查询的内存表 #10317
  - 将事务提交时间算入慢语句执行时间 #10310
  - 修复某些时间格式无法被 pt-query-digest 解析的问题 #10323

14.5.11.2 PD

- 支持 GetOperator 服务 #1514

14.5.11.3 TiKV

- 修复在 transfer leader 时非预期的 quorum 变化 #4604

14.5.11.4 Tools

- TiDB Binlog
  - 修复 unsigned int 类型的主键列的 binlog 数据为负数，造成同步出错中断的问题 #574
  - 删除下游是 pb 时的压缩选项，修改下游名字 pb 成 `file` #597
  - 修复 2.1.7 引入的 Reparo 生成错误 update 语句的 bug #576
- TiDB Lightning
  - 修复 parser 解析 bit 类型的 column 数据错误的 bug #164
  - 使用 row id 或者列的默认值填充 dump 文件中缺少的 column 数据 #174
  - Importer 修复部分 SST 导入失败依然返回导入成功的 bug #4566
  - Importer 支持 upload SST 到 TiKV 限速 #4607
  - 修改 Importer RocksDB SST 压缩方法为 `lz4`，减少 CPU 消耗 #4624
- sync-diff-inspector
  - 支持 checkpoint #227

14.5.11.5 TiDB Ansible

- 更新 tidb-ansible 中的文档链接，兼容重构之后的文档 #740，#741
- 移除 `inventory.ini` 中的 `enable_slow_query_log` 参数，默认即将 slow log 输出到单独的日志文件中 #742

14.5.12 TiDB 2.1.8 Release Notes

发版日期：2019 年 4 月 12 日

TiDB 版本：2.1.8

TiDB Ansible 版本：2.1.8

14.5.12.1 TiDB

- 修复 `GROUP_CONCAT` 函数在参数存在 NULL 值情况下与 MySQL 处理逻辑不兼容的问题 #9930
- 修复在 Distinct 模式下 decimal 类型值之间相等比较的问题 #9931
- 修复 `SHOW FULL COLUMNS` 语句在 date，datetime，timestamp 类型的 Collation 的兼容性问题
  - #9938
  - #10114
- 修复过滤条件存在关联列的时候统计信息估算行数不准确的问题 #9937
- 修复 `DATE_ADD` 跟 `DATE_SUB` 函数的兼容性问题
  - #9963
  - #9966
- `STR_TO_DATE` 函数支持格式 `%H`，提升兼容性 #9964
- 修复 `GROUP_CONCAT` 函数在 group by 唯一索引的情况下结果错误的问题 #9969
- 当 Optimizer Hints 存在不匹配的表名的时候返回 warning #9970

- 统一日志格式规范，利于工具收集分析 日志规范
- 修复大量 NULL 值导致统计信息估算不准确的问题 #9979
- 修复 TIMESTAMP 类型默认值为边界值的时候报错的问题 #9987
- 检查设置 `time_zone` 值的合法性 #10000
- 支持时间格式 `2019.01.01` #10001
- 修复某些情况下 EXPLAIN 结果中行数估计错误显示的问题 #10044
- 修复 `KILL TIDB [session id]` 某些情况下无法快速停止语句执行的问题 #9976
- 修复常量过滤条件在某些情况中谓词下推的问题 #10049
- 修复某些情况下 READ-ONLY 语句没有被当成 READ-ONLY 来处理的问题 #10048

14.5.12.2 PD

- 修复 Scatter Region 产生无效 Operator Step 的问题 #1482
- 修复 store 读热点的 key 统计不正确问题 #1487
- 修复 Region Merge Operator 超时时间过短的问题 #1495
- 添加 PD server 端处理 TSO 请求的耗时 metrics #1502

14.5.12.3 TiKV

- 修复读流量统计错误的问题 #4441
- 修复 Region 数过多的情况下 raftstore 的性能问题 #4484
- 调整当 level 0 SST 数量超过 `level_zero_slowdown_writes_trigger/2` 时不再继续 ingest file #4464

14.5.12.4 Tools

- Lightning 优化导入表的顺序，按照表的数据大小顺序进行导入，减少导入过程中大表执行 `Checksum` 和 `Analyze` 对集群的影响，并且提高 `Checksum` 和 `Analyze` 的成功率 #156
- 提升 Lightning encode SQL 性能，性能提升 50%，直接解析数据源文件内容成 TiDB 的 `types.Datum`，省去 KV encoder 的多余解析工作 #145
- TiDB Binlog Pump 新增 `storage.sync-log` 配置项，支持 Pump 本地存储异步刷盘 #529
- TiDB Binlog Pump 和 Drainer 之间通讯支持流量压缩 #530
- TiDB Binlog Drainer 新增 `syncer.sql-mode` 配置项，支持使用不同 `sql-mode` 解析 DDL query #513
- TiDB Binlog Drainer 新增 `syncer.ignore-table` 配置项，支持过滤不需要同步的表 #526

14.5.12.5 TiDB Ansible

- 修改操作系统版本限制，仅支持 CentOS 7.0 及以上，Red Hat 7.0 及以上版本的操作系统 #734
- 添加检测系统是否支持 `epollexclusive` #728
- 增加滚动升级版本限制，不允许从 2.0.1 及以下版本滚动升级到 2.1 及以上版本 #728

14.5.13 TiDB 2.1.7 Release Notes

发版日期：2019 年 3 月 28 日

TiDB 版本：2.1.7

TiDB Ansible 版本：2.1.7

14.5.13.1 TiDB

- 修复因 DDL 被取消导致升级程序时启动时间变长问题 #9768
- 修复配置项 `check-mb4-value-in-utf8` 在 `config.example.toml` 中位置错误的问题 #9852
- 提升内置函数 `str_to_date` 跟 MySQL 的兼容性 #9817
- 修复内置函数 `last_day` 的兼容性问题 #9750
- `infoschema.tables` 添加 `tidb_table_id` 列，方便通过 SQL 语句获取 `table_id`，新增 `tidb_indexes` 系统表管理 Table 与 Index 之间的关系 #9862
- 增加 Table Partition 定义为空的检查 #9663
- 将 `Truncate Table` 需要的权限由删除权限变为删表权限，与 MySQL 保持一致 #9876
- 支持在 DO 语句中使用子查询 #9877
- 修复变量 `default_week_format` 在 week 函数中不生效的问题 #9753
- 支持插件机制 #9880，#9888
- 支持使用系统变量 `log_bin` 查看 binlog 开启状况 #9634
- 支持使用 SQL 语句查看 Pump/Drainer 状态 #9896
- 修复升级时对 utf8 检查 mb4 字符的兼容性 #9887
- 修复某些情况下对 JSON 数据的聚合函数在计算过程中 Panic 的问题 #9927

14.5.13.2 PD

- 修改副本数为 1 时 balance-region 无法迁移 leader 问题 #1462

14.5.13.3 Tools

- 支持 binlog 同步 generated column #491

14.5.13.4 TiDB Ansible

- Prometheus 监控数据默认保留时间改成 30d

14.5.14 TiDB 2.1.6 Release Notes

2019 年 3 月 15 日，TiDB 发布 2.1.6 版，TiDB Ansible 相应发布 2.1.6 版本。相比 2.1.5 版本，该版本对系统稳定性、优化器、统计信息以及执行引擎做了很多改进。

14.5.14.1 TiDB

- 优化器/执行器
  - 当两个表在 `TIDB_INLJ` 的 Hint 中时，基于代价来选择外表 #9615
  - 修复在某些情况下，没有正确选择 IndexScan 的问题 #9587
  - 修复聚合函数在子查询里面的检查跟 MySQL 不兼容的行为 #9551
  - 使 `show stats_histograms` 语句只输出合法的列，避免 Panic #9502
- Server
  - 支持变量 `log_bin`，用于开启/关闭 Binlog #9634
  - 在事务中添加一个防御性检查，避免错误的事务提交 #9559
  - 修复设置变量导致的 Panic 的问题 #9539
- DDL
  - 修复 Create Table Like 语句在某些情况导致 Panic 的问题 #9652
  - 打开 etcd client 的 AutoSync 特性，防止某些情况下 TiDB 无法连接上 etcd 的问题 #9600

14.5.14.2 TiKV

- 修复在某些情况下解析 protobuf 失败导致 `StoreNotMatch` 错误的问题 #4303

14.5.14.3 Tools

- Lightning
  - importer 的默认的 region-split-size 变更为 512 MiB #4369
  - 保存原先在内存中的中间状态的 SST 到磁盘，减少内存使用 #4369
  - 限制 RocksDB 的内存使用 #4369
  - 修复 Region 还没有调度完成时进行 scatter 的问题 #4369
  - 将大表的数据和索引分离导入，在分批导入时能有效降低耗时 #132
  - 支援 CSV #111
  - 修复库名中含非英数字符时导入失败的错误 #9547

14.5.15 TiDB 2.1.5 Release Notes

2019 年 2 月 28 日，TiDB 发布 2.1.5 版，TiDB Ansible 相应发布 2.1.5 版本。相比 2.1.4 版本，该版本对系统稳定性、优化器、统计信息以及执行引擎做了很多改进。

14.5.15.1 TiDB

- 优化器/执行器
  - 当列的字符集信息和表的字符集信息相同时，`SHOW CREATE TABLE` 不再打印列的字符集信息，使其结果更加兼容 MySQL #9306

  - 将 Sort 算子中的表达计算抽取出来用一个 Project 算子完成，简化 Sort 算子的计算逻辑，修复某些情况下 Sort 算子结果不正确或者 panic 的问题 #9319
  - 移除 Sort 算子中的数值为常量的排序字段 #9335，#9440
  - 修复向无符号整数列插入数据时数据溢出的问题 #9339
  - 目标 binary 的长度超过 max_allowed_packet 时，将 cast_as_binary 设置为 NULL #9349
  - 优化 IF 和 IFNULL 的常量折叠过程 #9351
  - 使用 skyline pruning 优化 TiDB 的索引选择，增加简单查询的稳定性 #9356
  - 支持对析取范式计算选择率 #9405
  - 修复 !=ANY() 和 =ALL() 在某些情况下 SQL 查询结果不正确的问题 #9403
  - 修复执行 Merge Join 操作的两个表的 Join Key 类型不同时结果可能不正确或者 panic 的问题 #9438
  - 修复某些情况下 RAND() 函数结果和 MySQL 不兼容的问题 #9446
  - 重构 Semi Join 对 NULL 值和空结果集的处理逻辑，使其返回正确的结果，更加兼容 MySQL #9449
- Server
  - 增加系统变量 tidb_constraint_check_in_place，在 INSERT 语句执行时即检查数据的唯一性约束 #9401
  - 修复系统变量 tidb_force_priority 的值和配置文件中的设置的值不一致的问题 #9347
  - 在 general log 中增加 “current_db” 字段打印当前使用的数据库 #9346
  - 增加通过表 ID 来获取表信息的 HTTP API #9408
  - 修复 LOAD DATA 在某些情况下导入数据不正确的问题 #9414
  - 修复某些情况下，客户端建立连接慢的问题 #9451
- DDL
  - 修复撤销 DROP COLUMN 操作中的一些问题 #9352
  - 修复撤销 DROP/ADD 分区表操作中的一些问题 #9376
  - 修复某些情况下 ADMIN CHECK TABLE 误报数据索引不一致的问题 #9399
  - 修复 TIMESTAMP 类型的默认值在时区上的一些问题 #9108

14.5.15.2 PD

- GetAllStores 接口提供了 exclude_tombstone_stores 选项，将 Tombstone store 从返回结果中去除 #1444

14.5.15.3 TiKV

- 修复了某些情况下 Importer 导入失败的问题 #4223
- 修复了某些情况下 “key not in region” 错误 #4125
- 修复了某些情况下 Region merge 导致 panic 的问题 #4235
- 添加了详细的 StoreNotMatch 错误信息 #3885

14.5.15.4 Tools

- Lightning
  - 集群中有 Tombstone store 时 Lightning 不会再报错退出 #4223
- TiDB Binlog
  - 修正 DDL Binlog 同步方案，确保 DDL 同步的正确性 #9304

14.5.16 TiDB 2.1.4 Release Notes

2019 年 2 月 15 日，TiDB 发布 2.1.4 版，TiDB Ansible 相应发布 2.1.4 版本。相比 2.1.3 版本，该版本对系统稳定性、优化器、统计信息以及执行引擎做了很多改进。

14.5.16.1 TiDB

- 优化器/执行器
  - 修复 VALUES 函数未正确处理 FLOAT 类型的问题 #9223
  - 修复某些情况下 CAST 浮点数成字符串结果不正确的问题 #9227
  - 修复 FORMAT 函数在某些情况下结果不正确的问题 #9235
  - 修复某些情况下处理 Join 查询时 panic 的问题 #9264
  - 修复 VALUES 函数未正确处理 ENUM 类型的问题 #9280
  - 修复 DATE_ADD/DATE_SUB 在某些情况下结果不正确的问题 #9284
- Server
  - 优化 reload privilege success 日志，将其调整为 DEBUG 级别 #9274
- DDL
  - `tidb_ddl_reorg_worker_cnt` 和 `tidb_ddl_reorg_batch_size` 变成 GLOBAL 变量 #9134
  - 修复某些异常情况下，在 Generated column 增加索引导致的 Bug #9289

14.5.16.2 TiKV

- 修复在 TiKV 关闭时可能发生重复写的问题 #4146
- 修复某些情况下 event listener 结果处理异常的问题 #4132

14.5.16.3 Tools

- Lightning
  - 优化内存使用 #107，#108
  - 去掉 dump files 的 chunk 划分，减少对 dump files 的一次额外解析 #109
  - 限制读取 dump files 的 I/O 并发，避免过多的 cache miss 导致性能下降 #110
  - 对单个表实现 batch 导入，提高导入的稳定性 #110
  - TiKV 在 import 模式下开启 auto compactions #4199
  - 增加禁用 TiKV periodic Level-1 compaction 参数，因为当 TiKV 集群为 2.1.4 或更高版本时，在导入模式下会自动执行 Level-1 compaction #119
  - 限制 import engines 数量，避免过大占用 importer 磁盘空间 #119
- 数据同步对比统计 (sync-diff-inspector) 支持使用 TiDB 统计信息来划分 chunk #197

14.5.17 TiDB 2.1.3 Release Notes

2019 年 01 月 28 日，TiDB 发布 2.1.3 版，TiDB Ansible 相应发布 2.1.3 版本。相比 2.1.2 版本，该版本对系统稳定性、优化器、统计信息以及执行引擎做了很多改进。

14.5.17.1 TiDB

- 优化器/执行器
  - 修复某些情况下 Prepared Plan Cache panic 的问题 #8826
  - 修复在有前缀索引的某些情况下，Range 计算错误的问题 #8851
  - 当 SQL_MODE 不为 STRICT 时, CAST(str AS TIME(N)) 在 str 为非法的 TIME 格式的字符串时返回 NULL #8966
  - 修复某些情况下 Generated Column 在 Update 中 Panic 的问题 #8980
  - 修复统计信息直方图某些情况下上界溢出的问题 #8989
  - 支持对 _tidb_rowid 构造查询的 Range，避免全表扫，减轻集群压力 #9059
  - CAST(AS TIME) 在精度太大的情况下返回一个错误 #9058
  - 允许把 Sort Merge Join 用于笛卡尔积 #9037
  - 修复统计信息的 worker 在某些情况下 panic 之后无法恢复的问题 #9085
  - 修复某些情况下 Sort Merge Join 结果不正确的问题 #9046
  - 支持在 CASE 子句返回 JSON 类型 #8355
- Server
  - 当语句中有非 TiDB hint 的注释时返回警告，而不是错误 #8766
  - 验证设置的 TIMEZONE 的合法性 #8879
  - 优化 Metrics 项 QueryDurationHistogram，展示更多语句的类型 #8875
  - 修复 bigint 某些情况下下界溢出的问题 #8544
  - 支持 ALLOW_INVALID_DATES SQL mode #9110
- DDL
  - 修复一个 RENAME TABLE 的兼容性问题，保持行为跟 MySQL 一致 #8808
  - 支持 ADD INDEX 的并发修改即时生效 #8786
  - 修复在 ADD COLUMN 的过程中，某些情况 Update 语句 panic 的问题 #8906
  - 修复某些情况下并发创建 Table Partition 的问题 #8902
  - 支持把 utf8 字符集转换为 utf8mb4 字符集 #8951 #9152
  - 处理 Shard Bits 溢出的问题 #8976
  - 支持 SHOW CREATE TABLE 输出列的字符集 #9053
  - 修复 varchar 最大支持字符数在 utf8mb4 下限制的问题 #8818
  - 支持 ALTER TABLE TRUNCATE TABLE PARTITION #9093
  - 修复创建表的时候缺省字符集推算的问题 #9147

14.5.17.2 PD

- 修复 Leader 选举相关的 Watch 问题 #1396

14.5.17.3 TiKV

- 支持了使用 HTTP 方式获取监控信息 #3855
- 修复 data_format 遇到 NULL 时的问题 #4075
- 添加验证 Scan 请求的边界合法性 #4124

14.5.17.4 Tools

- TiDB Binlog
  - 修复在启动或者重启时 `no available pump` 的问题 #157
  - 开启 Pump client log 输出 #165
  - 修复表只有 unique key 没有 primary key 的情况下，unique key 包含 NULL 值导致数据更新不一致的问题

14.5.18 TiDB 2.1.2 Release Notes

2018 年 12 月 22 日，TiDB 发布 2.1.2 版，TiDB Ansible 相应发布 2.1.2 版本。该版本在 2.1.1 版的基础上，对系统兼容性、稳定性做出了改进。

14.5.18.1 TiDB

- 兼容 Kafka 版本的 TiDB Binlog #8747
- 完善滚动升级下 TiDB 的退出机制 #8707
- 修复在某些情况下为 generated column 增加索引 panic 的问题 #8676
- 修复在某些情况下语句有 `TIDB_SMJ Hint` 的时候优化器无法找到正确执行计划的问题 #8729
- 修复在某些情况下 `AntiSemiJoin` 返回错误结果的问题 #8730
- 增强 `utf8` 字符集的有效字符检查 #8754
- 修复事务中先写后读的情况下时间类型字段可能返回错误结果的问题 #8746

14.5.18.2 PD

- 修复 Region Merge 相关的 Region 信息更新问题 #1377

14.5.18.3 TiKV

- 支持以日 (d) 为时间单位的配置格式，并解决配置兼容性问题 #3931
- 修复 Approximate Size Split 可能会 panic 的问题 #3942
- 修复两个 Region merge 相关问题 #3822，#3873

14.5.18.4 Tools

- TiDB Lightning
  - 支持最小 TiDB 集群版本为 2.1.0
  - 修复解析包含 JSON 类型数据的文件内容出错 #144
  - 修复使用 checkpoint 重启后 `Too many open engines` 错误
- TiDB Binlog
  - 消除了 Drainer 往 Kafka 写数据的一些瓶颈点
  - TiDB 支持写 Kafka 版本的 TiDB Binlog

14.5.19 TiDB 2.1.1 Release Notes

2018 年 12 月 12 日，TiDB 发布 2.1.1 版。相比 2.1.0 版本，该版本对系统稳定性、优化器、统计信息以及执行引擎做了很多改进。

14.5.19.1 TiDB

- 优化器/执行器
  - 修复时间为负值时的四舍五入错误 #8574
  - 修复 `uncompress` 函数未检查数据长度的问题 #8606
  - 在执行 `execute` 命令后重置 `prepare` 语句绑定的变量 #8652
  - 支持对分区表自动收集统计信息 #8649
  - 修复在下推 `abs` 函数时设置错误的整数类型 #8628
  - 修复 JSON 列的数据竞争问题 #8660
- Server
  - 修复在 PD 故障时获取错误 TSO 的问题 #8567
  - 修复不规范的语句导致启动失败的问题 #8576
  - 修复在事务重试时使用了错误的参数 #8638
- DDL
  - 将表的默认字符集和排序规则改为 `utf8mb4` 和 `utf8mb4_bin` #8590
  - 增加变量 `ddl_reorg_batch_size` 来控制添加索引的速度 #8614
  - DDL 中的 character set 和 collation 选项内容不再大小写敏感 #8611
  - 修复对于生成列添加索引的问题 #8655

14.5.19.2 PD

- 修复一些配置项无法在配置文件中设置为 0 的问题 #1334
- 启动时检查未定义的配置 #1362
- 避免 transfer leader 至新创建的 Peer，优化可能产生的延迟增加问题 #1339
- 修复 RaftCluster 在退出时可能的死锁问题 #1370

14.5.19.3 TiKV

- 避免 transfer leader 至新创建的 Peer，优化可能产生的延迟增加问题 #3878

14.5.19.4 Tools

- Lightning
  - 优化对导入表的 `analyze` 机制，提升了导入速度
  - 支持 checkpoint 信息储存在本地文件
- TiDB Binlog
  - 修复 pb files 输出 bug，表只有主键列则无法产生 pb event

14.5.20 TiDB 2.1 GA Release Notes

2018 年 11 月 30 日，TiDB 发布 2.1 GA 版。相比 2.0 版本，该版本对系统稳定性、性能、兼容性、易用性做了大量改进。

14.5.20.1 TiDB

- SQL 优化器
  - 优化 `Index Join` 选择范围，提升执行性能
  - 优化 `Index Join` 外表选择，使用估算的行数较少的表作为外表
  - 扩大 Join Hint `TIDB_SMJ` 的作用范围，在没有合适索引可用的情况下也可使用 Merge Join
  - 加强 Join Hint `TIDB_INLJ` 的能力，可以指定 Join 中的内表
  - 优化关联子查询，包括下推 Filter 和扩大索引选择范围，部分查询的效率有数量级的提升
  - 支持在 `UPDATE` 和 `DELETE` 语句中使用 Index Hint 和 Join Hint
  - 支持更多函数下推：`ABS/CEIL/FLOOR/IS TRUE/IS FALSE`
  - 优化内建函数 `IF` 和 `IFNULL` 的常量折叠算法
  - 优化 `EXPLAIN` 语句输出格式，使用层级结构表示算子之间的上下游关系
- SQL 执行引擎
  - 重构所有聚合函数，提升 `Stream` 和 `Hash` 聚合算子的执行效率
  - 实现并行 `Hash Aggregate` 算子，部分场景下有 350% 的性能提升
  - 实现并行 `Project` 算子，部分场景有 74% 的性能提升
  - 并发地读取 `Hash Join` 的 `Inner` 表和 `Outer` 表的数据，提升执行性能
  - 优化 `REPLACE INTO` 语句的执行速度，性能提升 10x
  - 优化时间类型的内存占用，时间类型数据的内存使用降低为原来的一半
  - 优化点查的查询性能，Sysbench 点查效率提升 60%
  - TiDB 插入和更新宽表，性能提升接近 20 倍
  - 支持在配置文件中设置单个查询的内存使用上限
  - 优化 `Hash Join` 的执行过程，当 Join 类型为 `Inner Join` 或者 `Semi Join` 时，如果内表为空，不再读取外表数据，快速返回结果
  - 支持 `EXPLAIN ANALYZE` 语句，用于查看 Query 执行过程中各个算子的运行时间，返回结果行数等运行时统计信息
- 统计信息
  - 支持只在一天中的某个时间段开启统计信息自动 `ANALYZE` 的功能
  - 支持根据查询的反馈自动更新表的统计信息
  - 支持通过 `ANALYZE TABLE WITH BUCKETS` 语句配置直方图中桶的个数
  - 优化等值查询和范围查询混合的情况下使用直方图估算 Row Count 的算法
- 表达式
  - 支持内建函数：
    * `json_contains`
    * `json_contains_path`

    * encode/decode

- Server
    - 支持在单个 tidb-server 实例内部对冲突事务排队，优化事务间冲突频繁的场景下的性能
    - 支持 Server Side Cursor
    - 新增 HTTP 管理接口
    - 打散 table 的 Regions 在 TiKV 集群中的分布
    - 控制是否打开 `general log`
    - 在线修改日志级别
    - 查询 TiDB 集群信息
    - 添加 `auto_analyze_ratio` 系统变量控制自动 Analyze 的阈值
    - 添加 `tidb_retry_limit` 系统变量控制事务自动重试的次数
    - 添加 `tidb_disable_txn_auto_retry` 系统变量控制事务是否自动重试
    - 支持使用 `admin show slow` 语句来获取慢查询语句
    - 增加环境变量 `tidb_slow_log_threshold` 动态设置 slow log 的阈值
    - 增加环境变量 `tidb_query_log_max_len` 动态设置日志中被截断的原始 SQL 语句的长度
- DDL
    - 支持 Add Index 语句与其他 DDL 语句并行执行，避免耗时的 Add Index 操作阻塞其他操作
    - 优化 `Add Index` 的速度，在某些场景下速度大幅提升
    - 支持 `select tidb_is_ddl_owner()` 语句，方便判断 TiDB 是否为 `DDL Owner`
    - 支持 `ALTER TABLE FORCE` 语法
    - 支持 `ALTER TABLE RENAME KEY TO` 语法
    - `Admin Show DDL Jobs` 输出结果中添加表名、库名等信息
    - 支持使用 `ddl/owner/resign` HTTP 接口释放 DDL Owner 并开启新一轮 DDL Owner 选举
- 兼容性
    - 支持更多 MySQL 语法
    - BIT 聚合函数支持 `ALL` 参数
    - 支持 `SHOW PRIVILEGES` 语句
    - 支持 `LOAD DATA` 语句的 `CHARACTER SET` 语法
    - 支持 `CREATE USER` 语句的 `IDENTIFIED WITH` 语法
    - 支持 `LOAD DATA IGNORE LINES` 语句
    - `Show ProcessList` 语句返回更准确信息

14.5.20.2 PD (Placement Driver)

- 可用性优化
    - 引入 TiKV 版本控制机制，支持集群滚动兼容升级
    - PD 节点间开启 `Raft PreVote`，避免网络隔离后恢复时产生的重新选举
    - 开启 `raft learner` 功能，降低调度时出现宕机导致数据不可用的风险
    - TSO 分配不再受系统时间回退影响
    - 支持 `Region merge` 功能，减少元数据带来的开销

- 调度器优化
  - 优化 Down Store 的处理流程，加快发生宕机后补副本的速度
  - 优化热点调度器，在流量统计信息抖动时适应性更好
  - 优化 Coordinator 的启动，减少重启 PD 时带来的不必要调度
  - 优化 Balance Scheduler 频繁调度小 Region 的问题
  - 优化 Region merge，调度时考虑 Region 中数据的行数
  - 新增一些控制调度策略的开关
  - 完善调度模拟器，添加调度场景模拟
- API 及运维工具
  - 新增 `GetPrevRegion` 接口，用于支持 TiDB reverse scan 功能
  - 新增 `BatchSplitRegion` 接口，用于支持 TiKV 快速 Region 分裂
  - 新增 `GCSafePoint` 接口，用于支持 TiDB 并发分布式 GC
  - 新增 `GetAllStores` 接口，用于支持 TiDB 并发分布式 GC
  - pd-ctl 新增：
    * 使用统计信息进行 Region split
    * 调用 `jq` 来格式化 JSON 输出
    * 查询指定 store 的 Region 信息
    * 查询按 version 排序的 topN 的 Region 列表
    * 查询按 size 排序的 topN 的 Region 列表
    * 更精确的 TSO 解码
  - pd-recover 不再需要提供 max-replica 参数
- 监控
  - 增加 `Filter` 相关的监控
  - 新增 etcd Raft 状态机相关监控
- 性能优化
  - 优化处理 Region heartbeat 的性能，减少 heartbeat 带来的内存开销
  - 优化 Region tree 性能
  - 优化计算热点统计的性能问题

14.5.20.3 TiKV

- Coprocessor
  - 新增支持大量内建函数
  - 新增 Coprocessor ReadPool，提高请求处理并发度
  - 修复时间函数解析以及时区相关问题
  - 优化下推聚合计算的内存使用
- Transaction

  - 优化 MVCC 读取逻辑以及内存使用效率，提高扫描操作的性能，Count 全表性能比 2.0 版本提升 1 倍
  - 折叠 MVCC 中连续的 Rollback 记录，保证记录的读取性能
  - 新增 UnsafeDestroyRange API 用于在 drop table/index 的情况下快速回收空间
  - GC 模块独立出来，减少对正常写入的影响
  - kv_scan 命令支持设置 upper bound

- Raftstore

  - 优化 snapshot 文件写入流程避免导致 RocksDB stall
  - 增加 LocalReader 线程专门处理读请求，降低读请求的延迟
  - 支持 BatchSplit 避免大量写入导致产生特别大的 Region
  - 支持按照统计信息进行 Region Split，减少 IO 开销
  - 支持按照 Key 的数量进行 Region Split，提高索引扫描的并发度
  - 优化部分 Raft 消息处理流程，避免 Region Split 带来不必要的延迟
  - 启用 PreVote 功能，减少网络隔离对服务的影响

- 存储引擎

  - 修复 RocksDB CompactFiles 的 bug，可能影响 Lightning 导入的数据
  - 升级 RocksDB 到 v5.15，解决 snapshot 文件可能会被写坏的问题
  - 优化 IngestExternalFile，避免 flush 卡住写入的问题

- tikv-ctl

  - 新增 ldb 命令，方便排查 RocksDB 相关问题
  - compact 命令支持指定是否 compact bottommost 层的数据

14.5.20.4 Tools

- 全量数据快速导入工具 TiDB Lightning
- 支持新版本 TiDB Binlog

14.5.20.5 升级兼容性说明

- 由于新版本存储引擎更新，不支持在升级后回退至 2.0.x 或更旧版本
- 从 2.0.6 之前的版本升级到 2.1 之前，最好确认集群中是否存在正在运行中的 DDL 操作，特别是耗时的 Add Index 操作，等 DDL 操作完成后再执行升级操作
- 因为 2.1 版本启用了并行 DDL，对于早于 2.0.1 版本的集群，无法滚动升级到 2.1，可以选择下面两种方案：

  - 停机升级，直接从早于 2.0.1 的 TiDB 版本升级到 2.1
  - 先滚动升级到 2.0.1 或者之后的 2.0.x 版本，再滚动升级到 2.1 版本

14.5.21 TiDB 2.1 RC5 Release Notes

2018 年 11 月 12 日，TiDB 发布 2.1 RC5 版。相比 2.1 RC4 版本，该版本对系统稳定性、优化器、统计信息以及执行引擎做了很多改进。

14.5.21.1 TiDB

- SQL 优化器
  - 修复 IndexReader 在某些情况下读取的 handle 不正确的问题 #8132
  - 修复 IndexScan Prepared 语句在使用 Plan Cache 的时候的问题 #8055
  - 修复 Union 语句结果不稳定的问题 #8165
- 执行器
  - 提升 TiDB 插入和更新宽表的性能 #8024
  - 内建函数 Truncate 支持 unsigned int 参数 #8068
  - 修复转换 JSON 数据到 decimal 类型出错的问题 #8109
  - 修复 float 类型在 Update 时出错的问题 #8170
- 统计信息
  - 修复点查在某些情况下，统计信息出现错误的问题 #8035
  - 修复统计信息某些情况下在 primary key 的选择率的问题 #8149
  - 修复被删除的表的统计信息长时间没有清理的问题 #8182
- Server
  - 提升日志的可读性，完善日志信息
    * #8063
    * #8053
    * #8224
  - 修复获取 infoschema.profiling 表数据出错的问题 #8096
  - 替换 unix socket，使用 pumps client 来写 binlog #8098
  - 增加环境变量 tidb_slow_log_threshold 动态设置 slow log 的阈值 #8094
  - 增加环境变量 tidb_query_log_max_len 动态设置日志中被截断的原始 SQL 语句的长度 #8200
  - 增加环境变量 tidb_opt_write_row_id 来控制是否允许写入 _tidb_rowid #8218
  - ticlient Scan 命令增加边界，解决数据扫出边界的问题 #8081，#8247
- DDL
  - 修复在事务中某些情况下执行 DDL 语句出错的问题 #8056
  - 修复 partition 分区表执行 truncate table 没有生效的问题 #8103
  - 修复某些情况下 DDL 操作在被 cancel 之后没有正确回滚的问题 #8057
  - 增加命令 admin show next_row_id，返回下一个可用的行 ID #8268

14.5.21.2 PD

- 修复 pd-ctl 读取 Region key 的相关问题
  - #1298
  - #1299
  - #1308
- 修复 regions/check API 输出错误的问题 #1311
- 修复 PD join 失败后无法重新 join 的问题 #1279
- 修复某些情况下 watch leader 会丢失事件的问题 #1317

14.5.21.3 TiKV

- 优化 WriteConflict 报错信息 #3750
- 增加 panic 标记文件 #3746
- 降级 grpcio，避免新版本 gRPC 导致的 segment fault 问题 #3650
- 增加 kv_scan 接口扫描上界的限制 #3749

14.5.21.4 Tools

- TiDB 支持 TiDB Binlog cluster，不兼容旧版本 TiDB Binlog #8093，使用文档

14.5.22 TiDB 2.1 RC4 Release Notes

2018 年 10 月 23 日，TiDB 发布 2.1 RC4 版。相比 2.1 RC3 版本，该版本对系统稳定性、优化器、统计信息以及执行引擎做了很多改进。

14.5.22.1 TiDB

- SQL 优化器
  - 修复某些情况下 UnionAll 的列裁剪不正确的问题 #7941
  - 修复某些情况下 UnionAll 算子结果不正确的问题 #8007
- SQL 执行引擎
  - 修复 AVG 函数的精度问题 #7874
  - 支持通过 EXPLAIN ANALYZE 语句查看 Query 执行过程中各个算子的运行时间，返回结果行数等运行时统计信息 #7925
  - 修复多次引用同一列时 PointGet 算子 panic 的问题 #7943
  - 修复当 Limit 子句中的值太大时 panic 的问题 #8002
  - 修复某些情况下 AddDate/SubDate 执行过程中 panic 的问题 #8009
- 统计信息
  - 修复将组合索引的直方图下边界前缀判断为越界的问题 #7856
  - 修复统计信息收集引发的内存泄漏问题 #7873
  - 修复直方图为空时 panic 的问题 #7928
  - 修复加载统计信息时直方图边界越界的问题 #7944
  - 限制统计信息采样过程中数值的最大长度 #7982
- Server
  - 重构 Latch，避免事务冲突误判，提升并发事务的执行性能 #7711
  - 修复某些情况下收集 Slow Query 导致的 panic 问题 #7874
  - 修复 LOAD DATA 语句中，ESCAPED BY 为空字符串时 panic 的问题 #8005
  - 完善 “coprocessor error” 日志信息 #8006
- 兼容性
  - 当 Query 为空时，将 SHOW PROCESSLIST 结果中的 Command 字段设置为 “Sleep” #7839

- 表达式
  - 修复 SYSDATE 函数被常量折叠的问题 #7895
  - 修复 SUBSTRING_INDEX 在某些情况下 panic 的问题 #7897
- DDL
  - 修复抛出 “invalid ddl job type” 的错误时导致栈溢出的问题 #7958
  - 修复某些情况下 ADMIN CHECK TABLE 结果不正确的问题 #7975

14.5.22.2 PD

- 修复下线后的 TiKV 没有从 Grafana 面板中移除的问题 #1261
- 修复 grpc-go 设置 status 时的 data race 问题#1265
- 修复 etcd 启动失败导致的服务挂起问题 #1267
- 修复 leader 切换过程中可能产生的 data race #1273
- 修复下线 TiKV 时可能输出多余 warning 日志的问题 #1280

14.5.22.3 TiKV

- 优化 apply snapshot 导致的 RocksDB Write stall 的问题 #3606
- 增加 raftstore tick 相关 metrics #3657
- 升级 RocksDB，修复写入卡死及 IngestExternalFile 时可能写坏源文件的问题 #3661
- 升级 grpcio，修复 “too many pings” 误报的问题 #3650

14.5.23 TiDB 2.1 RC3 Release Notes

2018 年 9 月 29 日，TiDB 发布 2.1 RC3 版。相比 2.1 RC2 版本，该版本对系统稳定性、兼容性、优化器以及执行引擎做了很多改进。

14.5.23.1 TiDB

- SQL 优化器
  - 修复语句内包含内嵌的 LEFT OUTER JOIN 时，结果不正确的问题 #7689
  - 增强 JOIN 语句上的 predicate pushdown 优化规则 #7645
  - 修复 UnionScan 算子的 predicate pushdown 优化规则 #7695
  - 修复 Union 算子的 unique key 属性设置不正确的问题 #7680
  - 增强常量折叠的优化规则 #7696
  - 把常量传播后的 filter 是 null 的 data source 优化成 table dual #7756
- SQL 执行引擎
  - 优化事务内读请求的性能 #7717
  - 优化部分执行器 Chunk 内存分配的开销 #7540
  - 修复点查全部为 NULL 的列导致数组越界的问题 #7790
- Server

  - 修复配置文件里内存配额选项不生效的问题 #7729
  - 添加 tidb_force_priority 系统变量用来整体设置语句执行的优先级 #7694
  - 支持使用 `admin show slow` 语句来获取 SLOW QUERY LOG #7785
- 兼容性
  - 修复 `information_schema.schemata` 里 `charset/collation` 结果不正确的问题 #7751
  - 修复 `hostname` 系统变量的值为空的问题 #7750
- 表达式
  - 内建函数 `AES_ENCRYPT/AES_DECRYPT` 支持 `init_vecter` 参数 #7425
  - 修复部分表达式 `Format` 结果不正确的问题 #7770
  - 支持内建函数 `JSON_LENGTH` #7739
  - 修复 unsigned integer 类型 cast 为 decimal 类型结果不正确的问题 #7792
- DML
  - 修复 `INSERT ⋯ ON DUPLICATE KEY UPDATE` 语句在 unique key 更新时结果不正确的问题 #7675
- DDL
  - 修复在新建的 timestamp 类型的列上新建索引时，索引值没有做时区转换的问题 #7724
  - 支持 enum 类型 append 新的值 #7767
  - 快速新建 etcd session，使网络隔离后，集群更快恢复可用 #7774

14.5.23.2 PD

- 新特性
  - 添加获取按大小逆序排序的 Region 列表 API (/size) #1254
- 功能改进
  - Region API 会返回更详细的信息 #1252
- Bug 修复
  - 修复 PD 切换 leader 以后 `adjacent-region-scheduler` 可能会导致 crash 的问题 #1250

14.5.23.3 TiKV

- 性能优化
  - 优化函数下推的并发支持 #3515
- 新特性
  - 添加对 Log 函数的支持 #3603
  - 添加对 sha1 函数的支持 #3612
  - 添加 `truncate_int` 函数的支持 #3532
  - 添加 `year` 函数的支持 #3622
  - 添加 `truncate_real` 函数的支持 #3633
- Bug 修复
  - 修正时间函数相关的报错行为 #3487 #3615
  - 修复字符串解析成时间与 TiDB 不一致的问题 #3589

14.5.24 TiDB 2.1 RC2 Release Notes

2018 年 9 月 14 日，TiDB 发布 2.1 RC2 版。相比 2.1 RC1 版本，该版本对系统稳定性、优化器、统计信息以及执行引擎做了很多改进。

14.5.24.1 TiDB

- SQL 优化器
  - 新版 Planner 设计方案 #7543
  - 提升常量传播优化规则 #7276
  - 增强 Range 的计算逻辑使其能够同时处理多个 IN 或者等值条件 #7577
  - 修复当 Range 为空时，TableScan 的估算结果不正确的问题 #7583
  - 为 UPDATE 语句支持 PointGet 算子 #7586
  - 修复 FirstRow 聚合函数某些情况下在执行过程中 panic 的问题 #7624
- SQL 执行引擎
  - 解决 HashJoin 算子在遇到错误的情况下潜在的 DataRace 问题 #7554
  - HashJoin 算子同时读取内表数据和构建 Hash 表 #7544
  - 优化 Hash 聚合算子性能 #7541
  - 优化 Join 算子性能 #7493、 #7433
  - 修复 UPDATE JOIN 在 Join 顺序改变后结果不正确的问题 #7571
  - 提升 Chunk 迭代器的性能 #7585
- 统计信息
  - 解决重复自动 Analyze 统计信息的问题 #7550
  - 解决统计信息无变化时更新统计信息遇到错误的问题 #7530
  - Analyze 执行时使用低优先级以及 RC 隔离级别 #7496
  - 支持只在一天中的某个时间段开启统计信息自动更新的功能 #7570
  - 修复统计信息写日志时发生的 panic #7588
  - 支持通过 ANALYZE TABLE WITH BUCKETS 语句配置直方图中桶的个数 #7619
  - 修复更新空的直方图时 panic 的问题 #7640
  - 使用统计信息更新 information_schema.tables.data_length #7657
- Server
  - 增加 Trace 相关的依赖库 #7532
  - 开启 Golang 的 mutex profile 功能 #7512
  - Admin 语句需要 Super_priv 权限 #7486
  - 禁止用户 Drop 关键的系统表 #7471
  - 从 juju/errors 切换到 pkg/errors #7151
  - 完成 SQL Tracing 功能原型 #7016
  - 删除 goroutine pool #7564
  - 支持使用 USER1 信号来查看 goroutine 信息 #7587
  - 将 TiDB 启动时的内部 SQL 设置为高优先级 #7616
  - 在监控中用不同的标签区分内部 SQL 和用户 SQL #7631
  - 缓存最近一周内最慢的 30 条慢查询日志在 TiDB Server 上 #7646

  - TiDB 集群设置时区的方案 #7656
  - 丰富 GC life time is shorter than transaction duration 错误信息 #7658
  - 在 TiDB 集群启动时设置集群时区信息 #7638

- 兼容性
  - Year 类型字段增加 unsigned flag #7542
  - 修复在 Prepare/Execute 模式下，Year 类型结果长度设置问题 #7525
  - 修复 Prepare/Execute 模式下时间 0 值的处理问题 #7506
  - 解决整数类型除法实现中的错误处理问题 #7492
  - 解决 ComStmtSendLongData 处理过程中的兼容性问题 #7485
  - 解决字符串转为整数类型过程中的错误处理问题 #7483
  - 优化 information_schema.columns_in_table 表中的值精度 #7463
  - 修复使用 MariaDB 客户端对字符串类型数据的写入和更新的兼容性问题 #7573
  - 修复返回值别名的兼容性问题 #7600
  - 修复 information_schema.COLUMNS 表中浮点数的 NUMERIC_SCALE 值不正确的问题 #7602
  - 解决单行注释内容为空 Parser 报错的问题 #7612
- 表达式
  - 在 insert 函数中检查 max_allowed_packet 的值 #7528
  - 支持内建函数 json_contains #7443
  - 支持内建函数 json_contains_path #7596
  - 支持内建函数 encode/decode #7622
  - 修复一些时间相关的函数在某些情况下和 MySQL 行为不兼容的问题 #7636
  - 解决从字符串中解析时间类型数据的兼容性问题 #7654
  - 解决计算 DateTime 类型数据的默认值时没有考虑时区的问题 #7655
- DML
  - InsertOnDuplicateUpdate 语句设置正确的 last_insert_id #7534
  - 减少需要更新 auto_increment_id 计数器的情况 #7515
  - 优化 Duplicate Key 错误的报错信息 #7495
  - 修复 insert...select...on duplicate key update 问题 #7406
  - 支持 LOAD DATA IGNORE LINES 语句 #7576
- DDL
  - 在监控中增加 DDL Job 的类型和当前 Schema 版本的信息 #7472
  - 完成 Admin Restore Table 功能方案设计 #7383
  - 解决 Bit 类型的默认值超过 128 的问题 #7249
  - 解决 Bit 类型默认值不能为 NULL 的问题 #7604
  - 减少 DDL 队列中检查 CREATE TABLE/DATABASE 任务的时间间隔 #7608
  - 使用 ddl/owner/resign HTTP 接口释放 DDL Owner 并开启新一轮 Owner 选举 #7649
- TiKV Go Client
  - 支持 Seek 操作只获取 Key #7419
- Table Partition（实验性）
  - 解决无法使用 Bigint 类型列作为 Partition Key 的问题 #7520
  - 支持 Partitioned Table 添加索引过程中遇到问题回滚操作 #7437

14.5.24.2 PD

- 新特性
  - 支持 `GetAllStores`的接口 #1228
  - Simulator 添加评估调度的统计信息 #1218
- 功能改进
  - 优化 Down Store 的处理流程，尽快地补副本 #1222
  - 优化 Coordinator 的启动，减少重启 PD 带来的不必要调度 #1225
  - 优化内存使用，减少 heartbeat 带来的内存开销 #1195
  - 优化错误处理，完善日志信息 #1227
  - pd-ctl 支持查询指定 store 的 Region 信息 #1231
  - pd-ctl 支持查询按 version 比对的 topN 的 Region 信息 #1233
  - pd-ctl 支持更精确的 TSO 解码 #1242
- Bug 修复
  - 修复 pd-ctl 使用 hot store 命令错误退出的问题 #1244

14.5.24.3 TiKV

- 性能优化
  - 支持基于统计估算进行 Region split，减少 I/O 开销 #3511
  - 减少部分组件的内存拷贝 #3530
- 功能改进
  - 增加大量内建函数下推支持
  - 增加 `leader-transfer-max-log-lag` 配置解决特定场景 leader 调度失败的问题 #3507
  - 增加 `max-open-engines` 配置限制 `tikv-importer` 同时打开的 engine 个数 #3496
  - 限制垃圾数据的清理速度，减少对 `snapshot apply` 的影响 #3547
  - 对关键 Raft 消息广播 commit 信息，避免不必要的延迟 #3592
- Bug 修复
  - 修复新分裂 Region 的 PreVote 消息被丢弃导致的 leader 选举问题 #3557
  - 修复 Region merge 以后 follower 的相关统计信息 #3573
  - 修复 local reader 使用过期 Region 信息的问题 #3565

14.5.25 TiDB 2.1 RC1 Release Notes

2018 年 8 月 24 日，TiDB 发布 2.1 RC1 版。相比 2.1 Beta 版本，该版本对系统稳定性、优化器、统计信息以及执行引擎做了很多改进。

14.5.25.1 TiDB

- SQL 优化器
  - 修复某些情况下关联子查询去关联后结果不正确的问题 #6972
  - 优化 Explain 输出结果 #7011#7041
  - 优化 IndexJoin 驱动表选择策略#7019
  - 去掉非 PREPARE 语句的 Plan Cache #7040
  - 修复某些情况下 INSERT 语句无法正常解析执行的问题 #7068
  - 修复某些情况下 IndexJoin 结果不正确的问题 #7150
  - 修复某些情况下使用唯一索引不能查询到 NULL 值的问题 #7163
  - 修复 UTF-8 编码情况下前缀索引的范围计算不正确的问题 #7194
  - 修复某些情况下 Project 算子消除导致的结果不正确的问题 #7257
  - 修复主键为整数类型时无法使用 USE INDEX(PRIMARY) 的问题 #7316
  - 修复某些情况下使用关联列无法计算索引范围的问题 #7357
- SQL 执行引擎
  - 修复某些情况下夏令时时间计算结果不正确的问题 #6823
  - 重构聚合函数框架，提升 Stream 和 Hash 聚合算子的执行效率 #6852
  - 修复某些情况下 Hash 聚合算子不能正常退出的问题 #6982
  - 修复 BIT_AND/BIT_OR/BIT_XOR 没有正确处理非整型数据的问题 #6994
  - 优化 REPLACE INTO 语句的执行速度，性能提升近 10 倍 #7027
  - 优化时间类型的内存占用，时间类型数据的内存使用降低为原来的一半 #7043
  - 修复 UNION 语句整合有符号和无符号型整数结果时与 MySQL 不兼容的问题 #7112
  - 修复 LPAD/RPAD/TO_BASE64/FROM_BASE64/REPEAT 因为申请过多内存导致 TiDB panic 的问题 #7171 #7266 #7409 #7431
  - 修复 MergeJoin/IndexJoin 在处理 NULL 值时结果不正确的问题 #7255
  - 修复某些情况下 Outer Join 结果不正确的问题 #7288
  - 增强 Data Truncated 的报错信息，便于定位出错的数据和表中对应的字段 #7401
  - 修复某些情况下 Decimal 计算结果不正确的问题 #7001 #7113 #7202 #7208
  - 优化点查的查询性能 #6937
  - 禁用 Read Committed 隔离级别，避免潜在的问题 #7211
  - 修复某些情况下 LTRIM/RTRIM/TRIM 结果不正确的问题 #7291
  - 修复 MaxOneRow 算子无法保证返回结果不超过 1 行的问题 #7375
  - 拆分 range 个数过多的 Coprocessor 请求 #7454
- 统计信息
  - 优化统计信息动态收集机制 #6796
  - 解决数据频繁更新场景下 Auto Analyze 不工作的问题 #7022
  - 减少统计信息动态更新过程中的写入冲突 #7124
  - 优化统计信息不准确情况下的代价估算 #7175
  - 优化 AccessPath 的代价估算策略 #7233
- Server
  - 修复加载权限信息时的 bug #6976
  - 修复 Kill 命令对权限的检查过严问题 #6954

  - 解决 Binary 协议中某些数值类型移除的问题 #6922
  - 精简日志输出 #7029
  - 处理 mismatchClusterID 问题 #7053
  - 增加 advertise-address 配置项 #7078
  - 增加 GrpcKeepAlive 选项 #7100
  - 增加连接或者 Token 时间监控 #7110
  - 优化数据解码性能 #7149
  - INFORMMATION_SCHEMA 中增加 PROCESSLIST 表 #7236
  - 解决权限验证时多条规则可以命中情况下的顺序问题 #7211
  - 将部分编码相关的系统变量默认值改为 UTF-8 #7198
  - 慢查询日志显示更详细的信息 #7302
  - 支持在 PD 注册 tidb-server 的相关信息并通过 HTTP API 获取 #7082
- 兼容性
  - 支持 Session 变量 warning_count 和 error_count #6945
  - 读取系统变量时增加 Scope 检查 #6958
  - 支持 MAX_EXECUTION_TIME 语法 #7012
  - 支持更多的 SET 语法 #7020
  - Set 系统变量值过程中增加合法性校验 #7117
  - 增加 Prepare 语句中 PlaceHolder 数量的校验 #7162
  - 支持 set character_set_results = null #7353
  - 支持 flush status 语法 #7369
  - 修复 SET 和 ENUM 类型在 information_schema 里的 column size #7347
  - 支持建表语句里的 NATIONAL CHARACTER 语法 #7378
  - 支持 LOAD DATA 语句的 CHARACTER SET 语法 #7391
  - 修复 SET 和 ENUM 类型的 column info #7417
  - 支持 CREATE USER 语句的 IDENTIFIED WITH 语法 #7402
  - 修复 TIMESTAMP 类型计算过程中丢失精度的问题 #7418
  - 支持更多 SYSTEM 变量的合法性验证 #7196
  - 修复 CHAR_LENGTH 函数在计算 binary string 时结果不正确的问题 #7410
  - 修复在包含 GROUP BY 的语句里 CONCAT 结果不正确的问题 #7448
  - 修复 DECIMAL 类型 CAST 到 STRING 类型时，类型长度不准确的问题 #7451
- DML
  - 解决 Load Data 语句的稳定性 #6927
  - 解决一些 Batch 操作情况下的内存使用问题 #7086
  - 提升 Replace Into 语句的性能 #7027
  - 修复写入 CURRENT_TIMESTAMP 时，精度不一致的问题 #7355
- DDL
  - 改进 DDL 判断 Schema 是否已经同步的方法, 避免某些情况下的误判 #7319
  - 修复在 ADD INDEX 过程中的 SHOW CREATE TABLE 结果 #6993
  - 非严格 sql-mode 模式下, text/blob/json 的默认值可以为空 #7230
  - 修复某些特定场景下 ADD INDEX 的问题 #7142
  - 大幅度提升添加 UNIQUE-KEY 索引操作的速度 #7132
  - 修复 Prefix-index 在 UTF-8 字符集的场景下的截断问题 #7109

  - 增加环境变量 `tidb_ddl_reorg_priority` 来控制 add-index 操作的优先级 #7116
  - 修复 `information_schema.tables` 中 AUTO-INCREMENT 的显示问题 #7037
  - 支持 `admin show ddl jobs <number>` 命令, 支持输出 number 个 DDL jobs #7028
  - 支持并行 DDL 任务执行 #6955
- Table Partition（实验性）
  - 支持一级分区
  - 支持 `Range Partition`

14.5.25.2 PD

- 新特性
  - 引入版本控制机制，支持集群滚动兼容升级
  - 开启 `Region merge` 功能
  - 支持 `GetPrevRegion` 接口
  - 支持批量 `split Region`
  - 支持存储 GC safepoint
- 功能改进
  - 优化系统时间回退影响 TSO 分配的问题
  - 优化处理 Region heartbeat 的性能
  - 优化 Region tree 性能
  - 优化计算热点统计的性能问题
  - 优化 API 接口错误码返回
  - 新增一些控制调度策略的开关
  - 禁止在 `label` 中使用特殊字符
  - 完善调度模拟器
  - pd-ctl 支持使用统计信息进行 Region split
  - pd-ctl 支持调用 `jq` 来格式化 JSON 输出
  - 新增 etcd Raft 状态机相关 metrics
- Bug 修复
  - 修复 leader 切换后 namespace 未重新加载的问题
  - 修复 namespace 调度超出 schedule limit 配置的问题
  - 修复热点调度超出 schedule limit 的问题
  - 修复 PD client 关闭时输出一些错误日志的问题
  - 修复 Region 心跳延迟统计有误的问题

14.5.25.3 TiKV

- 新特性
  - 支持 `batch split`，防止热点 Region 写入产生超大 Region
  - 支持设置根据数据行数 split Region，提升 index scan 效率
- 性能优化

  - 使用 LocalReader 将 Read 操作从 raftstore 线程分离，减少 Read 延迟
  - 重构 MVCC 框架，优化 memory 使用，提升 scan read 性能
  - 支持基于统计估算进行 Region split，减少 I/O 开销
  - 优化连续写入 Rollback 记录后影响读性能的问题
  - 减少下推聚合计算的内存开销
- 功能改进
  - 增加大量内建函数下推支持，更完善的 charset 支持
  - 优化 GC 流程，提升 GC 速度并降低 GC 对系统的影响
  - 开启 prevote，加快网络异常时的恢复服务速度
  - 增加 RocksDB 日志文件相关的配置项
  - 调整 scheduler latch 默认配置
  - 使用 tikv-ctl 手动 compact 时可设定是否 compact RocksDB 最底层数据
  - 增加启动时的环境变量检查
  - 支持基于已有数据动态设置 dynamic_level_bytes 参数
  - 支持自定义日志格式
  - tikv-ctl 整合 tikv-fail 工具
  - 增加 threads IO metrics
- Bug 修复
  - 修复 decimal 相关问题
  - 修复 gRPC max_send_message_len 设置有误的问题
  - 修复 region_size 配置不当时产生的问题

14.5.26 TiDB 2.1 Beta Release Notes

2018 年 6 月 29 日，TiDB 发布 2.1 Beta 版。相比 2.0 版本，该版本对系统稳定性、优化器、统计信息以及执行引擎做了很多改进。

14.5.26.1 TiDB

- SQL 优化器
  - 优化 Index Join 选择范围，提升执行性能
  - 优化关联子查询，下推 Filter 和扩大索引选择范围，部分查询的效率有数量级的提升
  - 在 UPDATE、DELETE 语句中支持 Index Hint 和 Join Hint
  - 优化器 Hint TIDM_SMJ 在没有索引可用的情况下可生效
  - 支持更多函数下推：ABS/CEIL/FLOOR/IS TRUE/IS FALSE
  - 在常量折叠过程中特殊处理函数 IF 和 IFNULL
  - 优化 EXPLAIN 语句输出格式
- SQL 执行引擎
  - 实现并行 Hash Aggregate 算子，部分场景下能提高 Hash Aggregate 计算性能 350%
  - 实现并行 Project 算子，部分场景下性能提升达 74%
  - 并发地读取 Hash Join 的 Inner 表和 Outer 表的数据，提升执行性能
  - 修复部分场景下 INSERT ⋯ ON DUPLICATE KEY UPDATE ⋯ 结果不正确的问题

  - 修复 CONCAT_WS/FLOOR/CEIL/DIV 内建函数的结果不正确的问题

- Server
  - 添加 HTTP API 打散 table 的 Regions 在 TiKV 集群中的分布
  - 添加 auto_analyze_ratio 系统变量控制自动 analyze 的阈值
  - 添加 HTTP API 控制是否打开 general log
  - 添加 HTTP API 在线修改日志级别
  - 在 general log 和 slow query log 中添加 user 信息
  - 支持 Server side cursor
- 兼容性
  - 支持更多 MySQL 语法
  - BIT 聚合函数支持 ALL 参数
  - 支持 SHOW PRIVILEGES 语句
- DML
  - 减少 INSERT INTO SELECT 语句的内存占用
  - 修复 Plan Cache 的性能问题
  - 添加 tidb_retry_limit 系统变量控制事务自动重试的次数
  - 添加 tidb_disable_txn_auto_retry 系统变量控制事务是否自动重试
  - 修复写入 time 类型的数据精度问题
  - 支持本地冲突事务排队，优化冲突事务性能
  - 修复 UPDATE 语句的 Affected Rows
  - 优化 insert ignore on duplicate key update 语句性能
  - 优化 Create Table 语句的执行速度
  - 优化 Add index 的速度，在某些场景下速度大幅提升
  - 修复 Alter table add column 增加列超过表的列数限制的问题
  - 修复在某些异常情况下 DDL 任务重试导致 TiKV 压力增加的问题
  - 修复在某些异常情况下 TiDB 不断重载 Schema 信息的问题
- DDL
  - Show Create Table 不再输出外键相关的内容
  - 支持 select tidb_is_ddl_owner() 语句，方便判断 TiDB 是否为 DDL Owner
  - 修复某些场景下 YEAR 类型删除索引的问题
  - 修复并发执行场景下的 Rename table 的问题
  - 支持 ALTER TABLE FORCE 语法
  - 支持 ALTER TABLE RENAME KEY TO 语法
  - admin show ddl jobs 输出信息中添加表名、库名等信息

14.5.26.2 PD

- PD 节点间开启 Raft PreVote，避免网络隔离后恢复时产生的重新选举
- 优化 Balance Scheduler 频繁调度小 Region 的问题
- 优化热点调度器，在流量统计信息抖动时适应性更好
- region merge 调度时跳过数据行数较多的 Region
- 默认开启 raft learner 功能，降低调度时出现宕机导致数据不可用的风险

- pd-recover 移除 max-replica 参数
- 增加 Filter 相关的 metrics
- 修复 tikv-ctl unsafe recovery 之后 Region 信息没更新的问题
- 修复某些场景下副本迁移导致 TiKV 磁盘空间耗尽的问题
- 兼容性提示
  - 由于新版本存储引擎更新，不支持在升级后回退至 2.0.x 或更旧版本
  - 新版本默认开启 raft learner 功能，如果从 1.x 版本集群升级至 2.1 版本，须停机升级或者先滚动升级 TiKV，完成后再滚动升级 PD

14.5.26.3 TiKV

- 升级 Rust 到 nightly-2018-06-14 版本
- 开启 PreVote，避免网络隔离后恢复时产生的重新选举
- 添加 metric，显示 RocksDB 内部每层的文件数和 ingest 相关的信息
- GC 运行时打印版本太多的 key
- 使用 static metric 优化多 label metric 性能（YCSB raw get 提升 3%）
- 去掉多个模块的 box，使用范型提升运行时性能（YCSB raw get 提升 3%）
- 使用 asynchronous log 提升写日志性能
- 增加收集线程状态的 metric
- 通过减少程序中 box 的使用来减少内存拷贝的次数，提升性能

14.6 v2.0

14.6.1 TiDB 2.0.11 Release Notes

2019 年 1 月 3 日，TiDB 发布 2.0.11 版，TiDB Ansible 相应发布 2.0.11 版本。该版本在 2.0.10 版的基础上，对系统兼容性、稳定性做出了改进。

14.6.1.1 TiDB

- 修复 PD 发生异常的情况下，Error 没有被正确处理的问题 #8764
- 修复 Rename 相同表的行为，跟 MySQL 保持一致 #8809
- 修复 ADMIN CHECK TABLE 在 ADD INDEX 过程中误报的问题 #8750
- 修复前缀索引在某些情况下，开闭范围区间错误的问题 #8877
- 修复在某些添加列的情况下，UPDATE 语句 panic 的问题 #8904

14.6.1.2 TiKV

- 修复了两个 Region merge 相关的问题 #4003，#4004

14.6.2 TiDB 2.0.10 Release Notes

2018 年 12 月 18 日，TiDB 发布 2.0.10 版，TiDB Ansible 相应发布 2.0.10 版本。该版本在 2.0.9 版的基础上，对系统兼容性、稳定性做出了改进。

14.6.2.1 TiDB

- 修复取消 DDL 任务的时候可能导致的问题 #8513
- 修复 ORDER BY，UNION 语句无法引用带表名的列的问题 #8514
- 修复 UNCOMPRESS 函数没有判断错误输入长度的问题 #8607
- 修复 ANSI_QUOTES SQL_MODE 在 TiDB 升级的时候遇到的问题 #8575
- 修复某些情况下 select 返回结果错误的问题 #8570
- 修复 TiDB 在收到退出信号的时候可能无法退出的问题 #8501
- 修复某些情况下 IndexLookUpJoin 返回错误结果的问题 #8508
- 避免下推有 GetVar 或 SetVar 的 filter #8454
- 修复某些情况下 UNION 语句结果长度错误的问题 #8491
- 修复 PREPARE FROM @var_name 的问题 #8488
- 修复某些情况下导出统计信息 panic 的问题 #8464
- 修复统计信息某些情况下对点查估算的问题 #8493
- 修复某些情况下返回 Enum 默认值为字符串导致的 panic #8476
- 修复在宽表场景下，占用太多内存的问题 #8467
- 修复 Parser 对取模操作错误格式化导致的问题 #8431
- 修复某些情况下添加外键约束导致的 panic 问题 #8421，#8410
- 修复 YEAR 类型错误转换零值的问题 #8396
- 修复 VALUES 函数在参数不为列的时候 panic 的问题 #8404
- 存在子查询的语句禁用 Plan Cache #8395

14.6.2.2 PD

- 修复 RaftCluster 在退出时可能的死锁问题 #1370

14.6.2.3 TiKV

- 修复迁移 Leader 到新节点时造成请求延时问题 #3929
- 修复多余的 Region 心跳 #3930

14.6.3 TiDB 2.0.9 Release Notes

2018 年 11 月 19 日，TiDB 发布 2.0.9 版。该版本在 2.0.8 版的基础上，对系统兼容性、稳定性做出了改进。

14.6.3.1 TiDB

- 修复统计信息直方图为空的时候导致的问题 #7927
- 修复 UNION ALL 语句在某些情况下 panic 的问题 #7942
- 修复错误的 DDL JOB 情况下导致的递归溢出问题 #7959
- 为 Commit 操作加上慢操作日志 #7983
- 修复 Limit 值太大的情况下导致的 panic 问题 #8004
- 支持 USING 子句指定 utf8mb4 字符集 #8048
- 内建函数 TRUNCATE 支持类型为 unsigned int 的参数 #8069

- 修复统计信息模块在某些情况下主键选择率估算的问题 #8150
- 增加 Session 变量来控制是否允许写入 _tidb_rowid #8126
- 修复 PhysicalProjection 在某些情况下 panic 的问题 #8154
- 修复 Union 语句在某些情况下结果不稳定的问题 #8168
- 修复在非插入语句下 values 没有返回 NULL 的问题 #8179
- 修复某些情况下统计信息模块无法清除过期统计数据的问题 #8184
- 让事务允许的最长运行时间变成一个可配置项 #8209
- 修复 expression rewriter 某些情况下错误的比较逻辑 #8288
- 消除 UNION ORDER BY 语句生成的多余列的问题 #8307
- 支持 admin show next_row_id 语句 #8274
- 修复 Show Create Table 语句中特殊字符转义的问题 #8321
- 修复 UNION 语句在某些情况下遇到非预期错误的问题 #8318
- 修复某些情况下取消 DDL 任务导致的 Schema 没有回滚的问题 #8312
- 把变量 tidb_max_chunk_size 变成全局环境变量 #8333
- ticlient Scan 命令增加边界，解决数据扫出边界的问题 #8309 #8310

14.6.3.2 PD

- 修复 etcd 启动失败导致的服务挂起问题 #1267
- 修复 pd-ctl 读取 Region key 的相关问题 #1298 #1299 #1308
- 修复 regions/check API 输出错误的问题 #1311
- 修复 PD join 失败后无法重新 join 的问题 1279

14.6.3.3 TiKV

- 增加 kv_scan 接口扫描上界的限制 #3749
- 废弃配置 max-tasks-xxx 并新增 max-tasks-per-worker-xxx #3093
- 修复 RocksDB CompactFiles 的问题 #3789

14.6.4 TiDB 2.0.8 Release Notes

2018 年 10 月 16 日，TiDB 发布 2.0.8 版。该版本在 2.0.7 版的基础上，对系统兼容性、稳定性做出了改进。

14.6.4.1 TiDB

- 功能改进
  - 在 Update 没有更改相应 AUTO-INCREMENT 列情况下，防止 AUTO-ID 被消耗过快 #7846
- Bug 修复
  - 在 PD Leader 异常宕机的情况下，TiDB 快速创建 etcd Session 恢复服务 #7810
  - 修复 DateTime 类型使用默认值时候没有考虑时区的问题 #7672
  - 修复 duplicate key update 在某些情况下没有正确插入值的问题 #7685
  - 修复 UnionScan 中谓词条件没有下推的问题 #7726

- 修复增加 TIMESTAMP 索引没有正确处理时区的问题 #7812
- 修复某些情况下统计信息模块导致的内存泄露问题 #7864
- 修复在某些异常情况下，无法获得 ANALYZE 结果的问题 #7871
- 令 SYSDATE 不做表达式展开，以返回正确的结果 #7894
- 修复某些情况下，substring_index panic 的问题 #7896
- 修复某些情况下，错误将 OUTER JOIN 转为 INNER JOIN 的问题 #7899

14.6.4.2 TiKV

- Bug 修复
  - 修复节点宕机时 Raftstore EntryCache 占用内存持续上升的问题 #3529

14.6.5 TiDB 2.0.7 Release Notes

2018 年 9 月 7 日，TiDB 发布 2.0.7 版。该版本在 2.0.6 版的基础上，对系统兼容性、稳定性做出了改进。

14.6.5.1 TiDB

- New Feature
  - 在 information_schema 里添加 PROCESSLIST 表 #7286
- Improvements
  - 收集更多语句执行细节，并输出在 SLOW QUERY 日志里 #7364
  - SHOW CREATE TABLE 不再输出分区信息 #7388
  - 通过设置 RC 隔离级别和低优先级优化 ANALYZE 语句执行效率 #7500
  - 加速 ADD UNIQUE INDEX #7562
  - 增加控制 DDL 并发度的选项 #7563
- Bug Fixes
  - 修复 PRIMARY KEY 为整数的表，无法使用 USE INDEX(PRIMARY) 的问题 #7298
  - 修复 Merge Join 和 Index Join 在 inner row 为 NULL 时输出多余结果的问题 #7301
  - 修复 chunk size 设置过小时，Join 输出多余结果的问题 #7315
  - 修复建表语句中包含 range column 语法导致 panic 的问题 #7379
  - 修复 admin check table 对时间类型的列误报的问题 #7457
  - 修复以默认值 current_timestamp 插入的数据无法用 = 条件查询到的问题 #7467
  - 修复以 ComStmtSendLongData 命令插入空字符串参数被误解析为 NULL 的问题 #7508
  - 修复特定场景下 auto analyze 不断重复执行的问题 #7556
  - 修复 parser 无法解析以换行符结尾的单行注释的问题 #7635

14.6.5.2 TiKV

- Improvement
  - 空集群默认打开 dynamic-level-bytes 参数减少空间放大
- Bug Fix
  - 在 Region merge 之后更新 Region 的 approximate size 和 keys

14.6.6 TiDB 2.0.6 Release Notes

2018 年 8 月 6 日，TiDB 发布 2.0.6 版。该版本在 2.0.5 版的基础上，对系统兼容性、稳定性做出了改进。

14.6.6.1 TiDB

- Improvements
  - 精简 “set system variable” 日志的长度，减少日志文件体积 #7031
  - 在日志中记录 ADD INDEX 执行过程中的慢操作，便于定位问题 #7083
  - 减少更新统计信息操作中的事务冲突 #7138
  - 当待估算的值超过统计信息范围时，提高行数估计的准确度 #7185
  - 当使用 Index Join 时，选择行数估计较小的表作为驱动表，提高 Index Join 的执行效率 #7227
  - 为 ANALYZE TABLE 语句执行过程中发生的 panic 添加 recover 机制，避免收集统计信息过程中的异常行为导致 tidb-server 不可用 #7228
  - 当 RPAD/LPAD 的结果超过设置系统变量 max_allowed_packet 时，返回 NULL 和对应的 warning，兼容 MySQL #7244
  - 设置 PREPARE 语句中占位符数量上限为 65535，兼容 MySQL #7250
- Bug Fixes
  - 修复某些情况下，DROP USER 语句和 MySQL 行为不兼容的问题 #7014
  - 修复当 tidb_batch_insert 打开后，INSERT/LOAD DATA 等语句在某些场景下 OOM 的问题 #7092
  - 修复某个表的数据持续更新时，其统计信息自动更新失效的问题 #7093
  - 修复防火墙断掉不活跃的 gRPC 连接的问题 #7099
  - 修复某些场景下使用前缀索引结果不正确的问题 #7126
  - 修复某些场景下统计信息过时导致 panic 的问题 #7155
  - 修复某些场景下 ADD INDEX 后索引数据少一条的问题 #7156
  - 修复某些场景下查询唯一索引上的 NULL 值结果不正确的问题 #7172
  - 修复某些场景下 DECIMAL 的乘法结果出现乱码的问题 #7212
  - 修复某些场景下 DECIMAL 的取模运算结果不正确的问题 #7245
  - 修复某些特殊语句序列下在事务中执行 UPDATE/DELETE 语句后结果不正确的问题 #7219
  - 修复某些场景下 UNION ALL/UPDATE 语句在构造执行计划过程中 panic 的问题 #7225
  - 修复某些场景下前缀索引的索引范围计算错误的问题 #7231
  - 修复某些场景下 LOAD DATA 语句不写 binlog 的问题 #7242
  - 修复某些场景下在 ADD INDEX 过程中 SHOW CREATE TABLE 结果不正确的问题 #7243
  - 修复某些场景下 Index Join 因为没有初始化事务时间戳而 panic 的问题 #7246
  - 修复 ADMIN CHECK TABLE 因为误用 session 中的时区而导致误报的问题 #7258
  - 修复 ADMIN CLEANUP INDEX 在某些场景下索引没有清除干净的问题 #7265
  - 禁用 Read Committed 事务隔离级别 #7282

14.6.6.2 TiKV

- Improvements
  - 扩大默认 scheduler slots 值以减少假冲突现象
  - 减少回滚事务的连续标记以提升冲突极端严重下的读性能

  - 限制 RocksDB log 文件的大小和个数以减少长时间运行下不必要的磁盘占用

- Bug Fixes

  - 修复字符串转 Decimal 时出现的 crash

14.6.7 TiDB 2.0.5 Release Notes

2018 年 7 月 6 日，TiDB 发布 2.0.5 版。该版本在 2.0.4 版的基础上，对系统兼容性、稳定性做出了改进。

14.6.7.1 TiDB

- New Features

  - 增加一个系统变量 `tidb_disable_txn_auto_retry`，用于关闭事务自动重试 #6877

- Improvements

  - 调整计算 `Selection` 代价的方式，结果更准确 #6989
  - 查询条件能够完全匹配唯一索引或者主键时，直接选择作为查询路径 #6966
  - 启动服务失败时，做必要的清理工作 #6964
  - 在 `Load Data` 语句中，将 `\N` 处理为 NULL #6962
  - 优化 CBO 代码结构 #6953
  - 启动服务时，尽早上报监控数据 #6931
  - 对慢查询日志格式进行优化：去除 SQL 语句中的换行符，增加用户信息 #6920
  - 支持注释中存在多个星号的情况 #6858

- Bug Fixes

  - 修复 `KILL QUERY` 语句权限检查问题 #7003
  - 修复用户数量超过 1024 时可能造成无法登录的问题 #6986
  - 修复一个写入无符号类型 `float/double` 数据的问题 #6940
  - 修复 `COM_FIELD_LIST` 命令的兼容性，解决部分 MariaDB 客户端遇到 Panic 的问题 #6929
  - 修复 `CREATE TABLE IF NOT EXISTS LIKE` 行为 #6928
  - 修复一个 TopN 下推过程中的问题 #6923
  - 修复 `Add Index` 过程中遇到错误时当前处理的行 ID 记录问题 #6903

14.6.7.2 PD

- 修复某些场景下副本迁移导致 TiKV 磁盘空间耗尽的问题
- 修复 `AdjacentRegionScheduler` 导致的崩溃问题

14.6.7.3 TiKV

- 修复 decimal 运算中潜在的溢出问题
- 修复 merge 过程中可能发生的脏读问题

14.6.8 TiDB 2.0.4 Release Notes

2018 年 6 月 15 日，TiDB 发布 2.0.4 版。该版本在 2.0.3 版的基础上，对系统兼容性、稳定性做出了改进。

14.6.8.1 TiDB

- 支持 `ALTER TABLE t DROP COLUMN a CASCADE` 语法
- 支持设置 `tidb_snapshot` 变量的值为 TSO
- 优化监控项中语句类型展示
- 优化查询代价估计精度
- 设置 gRPC 的 `backoff max delay` 参数
- 支持通过配置文件设置单条语句的内存使用阈值
- 重构 Optimizer 的 error
- 解决 Cast Decimal 数据的副作用问题
- 解决特定场景下 `Merge Join` 算子结果错误的问题
- 解决转换 `Null` 对象到 String 的问题
- 解决 Cast JSON 数据为 JSON 类型的问题
- 解决 `Union` + `OrderBy` 情况下结果顺序和 MySQL 不一致的问题
- 解决 `Union` 语句中对 `Limit/OrderBy` 子句的合法性检查规则问题
- 解决 `Union All` 的结果兼容性问题
- 解决谓词下推中的一个 Bug
- 解决 `Union` 语句对 `For Update` 子句的兼容性问题
- 解决 `concat_ws` 函数对结果错误截断的问题

14.6.8.2 PD

- 改进 `max-pending-peer-count` 调度参数未设置时的行为，调整为不限制最大 `PendingPeer` 的数量

14.6.8.3 TiKV

- 新增 RocksDB `PerfContext` 接口用于调试
- 移除 `import-mode` 参数
- 为 `tikv-ctl` 添加 `region-properties` 命令
- 优化有大量 RocksDB tombstone 时 `reverse-seek` 过慢的问题
- 修复 `do_sub` 导致的崩溃问题
- 当 GC 遇到有太多版本的数据时记录日志

14.6.9 TiDB 2.0.3 Release Notes

2018 年 6 月 1 日，TiDB 发布 2.0.3 版。该版本在 2.0.2 版的基础上，对系统兼容性、稳定性做出了改进。

14.6.9.1 TiDB

- 支持在线更改日志级别
- 支持 COM_CHANGE_USER 命令
- 支持二进制协议情况下使用时间类型参数
- 优化带 BETWEEN 表达式的查询条件代价估算
- 在 SHOW CREATE TABLE 里不显示 FOREIGN KEY 信息
- 优化带 LIMIT 子句的查询代价估算
- 修复 YEAR 类型作为唯一索引的问题
- 修复在没有唯一索引的情况下 ON DUPLICATE KEY UPDATE 的问题
- 修复 CEIL 函数的兼容性问题
- 修复 DECIMAL 类型计算 DIV 的精度问题
- 修复 ADMIN CHECK TABLE 误报的问题
- 修复 MAX/MIN 在特定表达式参数下 panic 的问题
- 修复特殊情况下 JOIN 结果为空的问题
- 修复 IN 表达式构造查询 Range 的问题
- 修复使用 Prepare 方式进行查询且启用 Plan Cache 情况下的 Range 计算问题
- 修复异常情况下频繁加载 Schema 信息的问题

14.6.9.2 PD

- 修复在特定条件下收集 hot-cache metrics 会 panic 的问题
- 修复对旧的 Region 产生调度的问题

14.6.9.3 TiKV

- 修复 learner flag 错误上报给 PD 的 bug
- 在 do_div_mod 中 divisor/dividend 为 0 时返回错误

14.6.10 TiDB 2.0.2 Release Notes

2018 年 5 月 21 日，TiDB 发布 2.0.2 版。该版本在 2.0.1 版的基础上，对系统稳定性做出了改进。

14.6.10.1 TiDB

- 修复 Decimal 除法内置函数下推的问题
- 支持 Delete 语句中使用 USE INDEX 的语法
- 禁止在带有 Auto-Increment 的列中使用 shard_row_id_bits 特性
- 增加写入 Binlog 的超时机制

14.6.10.2 PD

- 使 balance leader scheduler 过滤失连节点

- 更改 transfer leader operator 的超时时间为 10 秒
- 修复 label scheduler 在集群 Regions 不健康状态下不调度的问题
- 修复 evict leader scheduler 调度不当的问题

14.6.10.3 TiKV

- 修复 Raft 日志没有打出来的问题
- 支持配置更多 gRPC 相关参数
- 支持配置选举超时的取值范围
- 修复过期 learner 没有删掉的问题
- 修复 snapshot 中间文件被误删的问题

14.6.11 TiDB 2.0.1 Release Notes

2018 年 5 月 16 日，TiDB 发布 2.0.1 版。该版本在 2.0.0 (GA) 版的基础上，对 MySQL 兼容性、系统稳定性做出了改进。

14.6.11.1 TiDB

- 实时更新 `Add Index` 的进度到 DDL 任务信息中
- 添加 Session 变量 `tidb_auto_analyze_ratio` 控制统计信息自动更新阈值
- 修复当事务提交失败时可能未清理所有的残留状态的问题
- 修复加索引在部分情况下的 Bug
- 修复 DDL 修改表面操作在某些并发场景下的正确性问题
- 修复某些情况下 `LIMIT` 结果不正确的问题
- 修复 `ADMIN CHECK INDEX` 语句索引名字区分大小写问题
- 修复 `UNION` 语句的兼容性问题
- 修复插入 `TIME` 类型数据的兼容性问题
- 修复某些情况下 `copIteratorTaskSender` 导致的 goroutine 泄漏问题
- 增加一个选项，用于设置 TiDB 在写 Binlog 失败的情况下的行为
- 优化 Coprocessor 慢请求日志格式，区分处理时间长与排队时间长的任务
- MySQL 协议握手阶段发生错误不打印日志，避免 KeepAlive 造成大量日志
- 优化 `Out of range value for column` 的错误信息
- 修复 `Update` 语句中遇到子查询导致结果错误的问题
- 调整 TiDB 进程处理 `SIGTERM` 的行为，不等待正在执行的 Query 完成

14.6.11.2 PD

- 添加 `Scatter Range` 调度，调度指定 Key Range 包含的 Region
- 优化 `Merge Region` 调度，使新分裂不久的 Region 不能被合并
- 添加 learner 相关的 metrics
- 修复重启误删 scheduler 的问题
- 修复解析配置文件出错问题
- 修复 etcd leader 和 PD leader 不同步的问题

- 修复关闭 learner 情况下还有 learner 出现的问题
- 修复读取包过大造成 load Regions 失败的问题

14.6.11.3 TiKV

- 修复 `SELECT FOR UPDATE` 阻止其他人读的问题
- 优化慢查询的日志
- 减少 `thread_yield` 的调用次数
- 修复生成 snapshot 会意外阻塞 raftstore 的 bug
- 修复特殊情况下开启 learner 无法选举成功的问题
- 修复极端情况下分裂可能导致的脏读问题
- 修正读线程池的配置默认值
- 修正删大数据表会影响写性能的问题

14.6.12 TiDB 2.0 Release Notes

2018 年 4 月 27 日，TiDB 发布 2.0 GA 版。相比 1.0 版本，该版本对 MySQL 兼容性、系统稳定性、优化器和执行器做了很多改进。

14.6.12.1 TiDB

- SQL 优化器
  - 精简统计信息数据结构，减小内存占用
  - 加快进程启动时加载统计信息速度
  - 支持统计信息动态更新experimental
  - 优化代价模型，对代价估算更精准
  - 使用 `Count-Min Sketch` 更精确地估算点查的代价
  - 支持分析更复杂的条件，尽可能充分的使用索引
  - 支持通过 `STRAIGHT_JOIN` 语法手动指定 Join 顺序
  - `GROUP BY`子句为空时使用 Stream Aggregation 算子，提升性能
  - 支持使用索引计算 `Max/Min` 函数
  - 优化关联子查询处理算法，支持将更多类型的关联子查询解关联并转化成 `Left Outer Join`
  - 扩大 `IndexLookupJoin` 的使用范围，索引前缀匹配的场景也可以使用该算法
- SQL 执行引擎
  - 使用 Chunk 结构重构所有执行器算子，提升分析型语句执行性能，减少内存占用，显著提升 TPC-H 结果
  - 支持 Streaming Aggregation 算子下推
  - 优化 `Insert Into Ignore` 语句性能，提升 10 倍以上
  - 优化 `Insert On Duplicate Key Update` 语句性能，提升 10 倍以上
  - 下推更多的数据类型和函数到 TiKV 计算
  - 优化 `Load Data` 性能，提升 10 倍以上
  - 支持对物理算子内存使用进行统计，通过配置文件以及系统变量指定超过阈值后的处理行为
  - 支持限制单条 SQL 语句使用内存的大小，减少程序 OOM 风险

  - 支持在 CRUD 操作中使用隐式的行 ID
  - 提升点查性能

- Server
  - 支持 Proxy Protocol
  - 添加大量监控项, 优化日志
  - 支持配置文件的合法性检测
  - 支持 HTTP API 获取 TiDB 参数信息
  - 使用 Batch 方式 Resolve Lock，提升垃圾回收速度
  - 支持多线程垃圾回收
  - 支持 TLS

- 兼容性
  - 支持更多 MySQL 语法
  - 支持配置文件修改 `lower_case_table_names` 系统变量，用于支持 OGG 数据同步工具
  - 提升对 Navicat 的兼容性
  - 在 `Information_Schema` 中支持显示建表时间
  - 修复部分函数/表达式返回类型和 MySQL 不同的问题
  - 提升对 JDBC 兼容性
  - 支持更多的 `SQL_MODE`

- DDL
  - 优化 `Add Index` 的执行速度，部分场景下速度大幅度提升
  - `Add Index` 操作变更为低优先级，降低对线上业务影响
  - `Admin Show DDL Jobs` 输出更详细的 DDL 任务状态信息
  - 支持 `Admin Show DDL Job Queries JobID` 查询当前正在运行的 DDL 任务的原始语句
  - 支持 `Admin Recover Index` 命令，用于灾难恢复情况下修复索引数据
  - 支持通过 `Alter` 语句修改 Table Options

14.6.12.2 PD

- 增加 `Region Merge` 支持，合并数据删除后产生的空 Region experimental
- 增加 `Raft Learner` 支持experimental
- 调度器优化
  - 调度器适应不同的 Region size
  - 提升 TiKV 宕机时数据恢复的优先级和恢复速度
  - 提升下线 TiKV 节点搬迁数据的速度
  - 优化 TiKV 节点空间不足时的调度策略，尽可能防止空间不足时磁盘被写满
  - 提升 balance-leader scheduler 的调度效率
  - 减少 balance-region scheduler 调度开销
  - 优化 hot-region scheduler 的执行效率
- 运维接口及配置
  - 增加 TLS 支持
  - 支持设置 PD leader 优先级

  - 支持基于 label 配置属性
  - 支持配置特定 label 的节点不调度 Region leader
  - 支持手动 Split Region，可用于处理单 Region 热点的问题
  - 支持打散指定 Region，用于某些情况下手动调整热点 Region 分布
  - 增加配置参数检查规则，完善配置项的合法性较验
- 调试接口
  - 增加 `Drop Region` 调试接口
  - 增加枚举各个 PD health 状态的接口
- 统计相关
  - 添加异常 Region 的统计
  - 添加 Region 隔离级别的统计
  - 添加调度相关 metrics
- 性能优化
  - PD leader 尽量与 etcd leader 保持同步，提升写入性能
  - 优化 Region heartbeat 性能，现可支持超过 100 万 Region

14.6.12.3 TiKV

- 功能
  - 保护关键配置，防止错误修改
  - 支持 `Region Merge` experimental
  - 添加 `Raw DeleteRange` API
  - 添加 `GetMetric` API
  - 添加 `Raw Batch Put`，`Raw Batch Get`，`Raw Batch Delete` 和 `Raw Batch Scan`
  - 给 Raw KV API 增加 Column Family 参数，能对特定 Column Family 进行操作
  - Coprocessor 支持 streaming 模式，支持 streaming 聚合
  - 支持配置 Coprocessor 请求的超时时间
  - 心跳包携带时间戳
  - 支持在线修改 RocksDB 的一些参数，包括 `block-cache-size` 大小等
  - 支持配置 Coprocessor 遇到某些错误时的行为
  - 支持以导数据模式启动，减少导数据过程中的写放大
  - 支持手动对 region 进行对半 split
  - 完善数据修复工具 tikv-ctl
  - Coprocessor 返回更多的统计信息，以便指导 TiDB 的行为
  - 支持 ImportSST API，可以用于 SST 文件导入experimental
  - 新增 TiKV Importer 二进制，与 TiDB Lightning 集成用于快速导入数据experimental
- 性能
  - 使用 ReadPool 优化读性能，`raw_get/get/batch_get` 提升 30%
  - 提升 metrics 的性能
  - Raft snapshot 处理完之后立即通知 PD，加快调度速度
  - 解决 RocksDB 刷盘导致性能抖动问题

  - 提升在数据删除之后的空间回收
  - 加速启动过程中的垃圾清理过程
  - 使用 DeleteFilesInRanges 减少副本迁移时 I/O 开销
- 稳定性
  - 解决在 PD leader 发送切换的情况下 gRPC call 不返回问题
  - 解决由于 snapshot 导致下线节点慢的问题
  - 限制搬移副本临时占用的空间大小
  - 如果有 Region 长时间没有 Leader，进行上报
  - 根据 compaction 事件及时更新统计的 Region size
  - 限制单次 scan lock 请求的扫描的数据量，防止超时
  - 限制接收 snapshot 过程中的内存占用，防止 OOM
  - 提升 CI test 的速度
  - 解决由于 snapshot 太多导致的 OOM 问题
  - 配置 gRPC 的 keepalive 参数
  - 修复 Region 增多容易 OOM 的问题

14.6.12.4 TiSpark

TiSpark 使用独立的版本号，现为 1.0 GA。TiSpark 1.0 版本组件提供了针对 TiDB 上的数据使用 Apache Spark 进行分布式计算的能力。

- 提供了针对 TiKV 读取的 gRPC 通信框架
- 提供了对 TiKV 组件数据的和通信协议部分的编码解码
- 提供了计算下推功能，包含：
  - 聚合下推
  - 谓词下推
  - TopN 下推
  - Limit 下推
- 提供了索引相关的支持
  - 谓词转化聚簇索引范围
  - 谓词转化次级索引
  - Index Only 查询优化
  - 运行时索引退化扫表优化
- 提供了基于代价的优化
  - 统计信息支持
  - 索引选择
  - 广播表代价估算
- 多种 Spark Interface 的支持
  - Spark Shell 支持
  - ThriftServer/JDBC 支持
  - Spark-SQL 交互支持
  - PySpark Shell 支持
  - SparkR 支持

14.6.13 TiDB 2.0 RC5 Release Notes

2018 年 4 月 17 日，TiDB 发布 2.0 RC5 版。该版本在 RC4 版的基础上，对 MySQL 兼容性、系统稳定性和优化器做了很多改进。

14.6.13.1 TiDB

- 修复应用 Top-N 下推规则的问题
- 修复对包含 NULL 值的列的行数估算
- 修复 Binary 类型的 0 值
- 修复事务内的 BatchGet 问题
- 回滚 Add Index 操作的时候，清除清除已写入的数据，减少空间占用
- 优化 insert on duplicate key update 语句性能，提升 10 倍以上
- 修复 UNIX_TIMESTAMP 函数返回结果类型问题返回结果类型问题
- 修复在添加 NOT NULL 列的过程中，插入 NULL 值的问题
- Show Process List 语句支持显示执行语句的内存占用
- 修复极端情况下 Alter Table Modify Column 出错问题
- 支持通过 Alter 语句设置 table comment

14.6.13.2 PD

- 添加 Raft Learner 支持
- 优化 Balance Region Scheduler，减少调度开销
- 调整默认 schedule-limit 配置
- 修复频繁分配 ID 问题
- 修复添加调度兼容性问题

14.6.13.3 TiKV

- tikv-ctl 支持 compact 指定的 Region
- Raw KV 支持 Batch Put、 Batch Get、 Batch Delete 和 Batch Scan
- 解决太多 snapshot 导致的 OOM 问题
- Coprocessor 返回更详细的错误信息
- 支持通过 tikv-ctl 动态修改 TiKV 的 block-cache-size
- 进一步完善 importer 功能
- 简化 ImportSST::Upload 接口
- 设置 gRPC 的 keepalive 属性
- tikv-importer 作为独立的 binary 从 TiKV 中分离出来
- 统计 Coprocessor 每个 scan range 命令扫描了多少行数据
- 解决在 macOS 系统上的编译问题
- 优化 metric 相关的内容
- 解决 snapshot 相关的一个潜在 bug
- 解决误用了一个 RocksDB metric 的问题
- Coprocessor 支持 overflow as warning 选项

14.6.14 TiDB 2.0 RC4 Release Notes

2018 年 3 月 30 日，TiDB 发布 2.0 RC4 版。该版本在 2.0 RC3 版的基础上，对 MySQL 兼容性、系统稳定性和优化器做了很多改进。

14.6.14.1 TiDB

- 支持 `SHOW GRANTS FOR CURRENT_USER();`
- 修复 `UnionScan` 里的 `Expression` 没有 Clone 的问题
- 支持 `SET TRANSACTION` 语法
- 修复 `copIterator` 中潜在的 goroutine 泄露问题
- 修复 `admin check table` 对包含 null 的 unique index 误判的问题
- 支持用科学计数法显示浮点数
- 修复 binary literal 计算时的类型推导
- 修复解析 `CREATE VIEW` 语句的问题
- 修复语句中同时包含 `ORDER BY` 和 `LIMIT 0` 时 panic 的问题
- 提升 `DecodeBytes` 执行性能
- 优化 `LIMIT 0` 为 `TableDual`，避免无用的执行计划构建

14.6.14.2 PD

- 支持手动 split Region，可用于处理单 Region 热点的问题
- 修复 `pdctl` 运行 `config show all` 不显示 label property 的问题
- metrics 及代码结构相关的优化

14.6.14.3 TiKV

- 限制接收 snapshot 时的内存使用，解决极端情况下的 OOM
- 可以配置 Coprocessor 在遇到 warnings 时的行为
- TiKV 支持导数据模式
- 支持 Region 从正中间分裂
- 提升 CI test 的速度
- 使用 `crossbeam channel`
- 改善 TiKV 在被隔离的情况下由于 leader missing 输出太多日志的问题

14.6.15 TiDB 2.0 RC3 Release Notes

2018 年 3 月 23 日，TiDB 发布 2.0 RC3 版。该版本在 2.0 RC2 版的基础上，对 MySQL 兼容性、系统稳定性和优化器做了很多改进。

14.6.15.1 TiDB

- 修复部分场景下 `MAX/MIN` 结果不正确的问题

- 修复部分场景下 Sort Merge Join 结果未按照 Join Key 有序的问题
- 修复边界条件下 uint 和 int 比较的错误
- 完善浮点数类型的长度和精度检查，提升 MySQL 兼容性
- 完善时间类型解析报错日志，添加更多错误信息
- 完善内存控制，新增对 IndexLookupExecutor 的内存统计
- 优化 ADD INDEX 的执行速度，部分场景下速度大幅度提升
- GROUP BY 子句为空时使用 Stream Aggregation 算子，提升速度
- 支持通过 STRAIGHT_JOIN 来关闭优化器的 Join Reorder 优化
- ADMIN SHOW DDL JOBS 输出更详细的 DDL 任务状态信息
- 支持 ADMIN SHOW DDL JOB QUERIES 查询当前正在运行的 DDL 任务的原始语句
- 支持 ADMIN RECOVER INDEX 命令，用于灾难恢复情况下修复索引数据
- ADD INDEX 操作变更为低优先级，降低对线上业务影响
- 支持参数为 JSON 类型的 SUM/AVG 等聚合函数
- 支持配置文件修改 lower_case_table_names 系统变量，用于支持 OGG 数据同步工具
- 提升对 Navicat 管理工具的兼容性
- 支持在 CRUD 操作中使用隐式的行 ID

14.6.15.2 PD

- 支持 Region Merge，合并数据删除后产生的空 Region 或小 Region
- 添加副本时忽略有大量 pending peer 的节点，提升恢复副本及下线的速度
- 优化有大量空 Region 时产生的频繁调度问题
- 优化不同 label 中资源不均衡的场景中 leader balance 调度的速度
- 添加更多异常 Region 的统计

14.6.15.3 TiKV

- 支持 Region Merge
- Raft snapshot 流程完成之后立刻通知 PD，加速调度
- 增加 Raw DeleteRange API
- 增加 GetMetric API
- 减缓 RocksDB sync 文件造成的 I/O 波动
- 优化了对 delete 掉数据的空间回收机制
- 完善数据恢复工具 tikv-ctl
- 解决了由于 snapshot 导致下线节点慢的问题
- Coprocessor 支持 streaming
- 支持 Readpool，raw_get/get/batch_get 性能提升 30%
- 支持配置 Coprocessor 请求超时时间
- Coprocessor 支持 streaming aggregation
- 上报 Region heartbeat 时携带时间信息
- 限制 snapshot 文件的空间使用，防止占用过多磁盘空间
- 对长时间不能选出 leader 的 Region 进行记录上报
- 加速启动阶段的垃圾清理工作
- 根据 compaction 事件及时更新对应 Region 的 size 信息

- 对 `scan lock` 的大小进行限制，防止请求超时
- 使用 `DeleteRange` 加速 Region 删除
- 支持在线修改 RocksDB 的参数

14.6.16 TiDB 2.0 RC1 Release Notes

2018 年 3 月 9 日，TiDB 发布 2.0 RC1 版。该版本在上一版的基础上，对 MySQL 兼容性、系统稳定性和优化器做了很多改进。

14.6.16.1 TiDB

- 支持限制单条 SQL 语句使用内存的大小，减少程序 OOM 风险
- 支持下推流式聚合算子到 TiKV
- 支持配置文件的合法性检测
- 支持 HTTP API 获取 TiDB 参数信息
- Parser 兼容更多 MySQL 语法
- 提升对 Navicat 的兼容性
- 优化器提升，提取多个 OR 条件的公共表达式，选取更优执行计划
- 优化器提升，在更多场景下将子查询转换成 Join 算子，选取更优查询计划
- 使用 Batch 方式 Resolve Lock，提升垃圾回收速度
- 修复 Boolean 类型的字段长度，提升兼容性
- 优化 Add Index 操作，所有的读写操作采用低优先级，减小对在线业务的影响

14.6.16.2 PD

- 优化检查 Region 状态的代码逻辑，提升程序性能
- 优化异常情况下日志信息输出，便于调试
- 修复监控中关于 TiKV 节点磁盘空间不足情况的统计
- 修复开启 TLS 时健康检查接口误报的问题
- 修复同时添加副本数量可能超过配置阈值的问题，提升程序稳定性

14.6.16.3 TiKV

- 修复 PD leader 切换，gRPC call 没被 cancel 的问题
- 对重要配置进行保护，第一次设置之后不允许变更
- 增加获取 metrics 的 gRPC API
- 启动时候，检查是否使用 SSD
- 使用 ReadPool 优化读性能，`raw get` 测试性能提升 30%
- 完善 metrics，优化 metrics 的使用

14.6.17 TiDB 1.1 Beta Release Notes

2018 年 2 月 24 日，TiDB 发布 1.1 Beta 版。该版本在 1.1 Alpha 版的基础上，对 MySQL 兼容性、系统稳定性做了很多改进。

14.6.17.1 TiDB

- 添加更多监控项, 优化日志
- 兼容更多 MySQL 语法
- 在 `information_schema` 中支持显示建表时间
- 提速包含 MaxOneRow 算子的查询
- 控制 Join 产生的中间结果集大小，进一步减少 Join 的内存使用
- 增加 `tidb_config` session 变量，输出当前 TiDB 配置
- 修复 Union 和 `Index Join` 算子中遇到的 panic 问题
- 修复 `Sort Merge Join` 算子在部分场景下结果错误的问题
- 修复 `Show Index` 语句显示正在添加过程中的索引的问题
- 修复 `Drop Stats` 语句失败的问题
- 优化 SQL 引擎查询性能，Sysbench 的 Select/OLTP 测试结果提升 10%
- 使用新的执行引擎提升优化器中的子查询计算速度；相比 1.0 版本，在 TPC-H 以及 TPC-DS 等测试中有显著提升

14.6.17.2 PD

- 增加 Drop Region 调试接口
- 支持设置 PD leader 优先级
- 支持配置特定 label 的节点不调度 Raft leader
- 增加枚举各个 PD health 状态的接口
- 添加更多 metrics
- PD leader 尽量与 etcd leader 保持同步
- 提高 TiKV 宕机时数据恢复优先级和恢复速度
- 完善 `data-dir` 配置项的合法性较验
- 优化 Region heartbeat 性能
- 修复热点调度破坏 label 约束的问题
- 其他稳定性问题修复

14.6.17.3 TiKV

- 使用 offset + limit 遍历 lock，消除潜在的 GC 问题
- 支持批量 resolve lock，提升 GC 速度
- 支持并行 GC，提升 GC 速度
- 使用 RocksDB compaction listener 更新 Region Size，让 PD 更精确的进行调度
- 使用 `DeleteFilesInRanges` 批量删除过期数据，提高 TiKV 启动速度
- 设置 Raft snapshot max size，防止遗留文件占用太多空间
- `tikv-ctl` 支持更多修复操作
- 优化有序流式聚合操作
- 完善 metrics，修复 bug

14.6.18 TiDB 1.1 Alpha Release Notes

2018 年 1 月 19 日，TiDB 发布 1.1 Alpha 版。该版本对 MySQL 兼容性、SQL 优化器、系统稳定性、性能做了大量的工作。

14.6.18.1 TiDB

- SQL parser
  - 兼容更多语法
- SQL 查询优化器
  - 统计信息减小内存占用
  - 优化统计信息启动时载入的时间
  - 更精确的代价估算
  - 使用 `Count-Min Sketch` 更精确地估算点查的代价
  - 支持更复杂的条件，更充分使用索引
- SQL 执行器
  - 使用 Chunk 结构重构所有执行器算子，提升分析型语句执行性能，减少内存占用
  - 优化 `INSERT IGNORE` 语句性能
  - 下推更多的类型和函数
  - 支持更多的 `SQL_MODE`
  - 优化 `Load Data` 性能，速度提升 10 倍
  - 优化 `Use Database` 性能
  - 支持对物理算子内存使用进行统计
- Server
  - 支持 PROXY protocol

14.6.18.2 PD

- 增加更多的 API
- 支持 TLS
- 给 Simulator 增加更多的 case
- 调度适应不同的 Region size
- Fix 了一些调度的 bug

14.6.18.3 TiKV

- 支持 Raft learner
- 优化 Raft Snapshot，减少 I/O 开销
- 支持 TLS
- 优化 RocksDB 配置，提升性能
- 优化 Coprocessor `count (*)` 和点查 unique index 的性能

- 增加更多的 Failpoint 以及稳定性测试 case
- 解决 PD 和 TiKV 之间重连的问题
- 增强数据恢复工具 `tikv-ctl` 的功能
- Region 支持按 table 进行分裂
- 支持 `Delete Range` 功能
- 支持设置 snapshot 导致的 I/O 上限
- 完善流控机制

14.7 v1.0

14.7.1 TiDB 1.0 Release Notes

2017 年 10 月 16 日，TiDB 发布 GA 版（TiDB 1.0）。该版本对 MySQL 兼容性、SQL 优化器、系统稳定性、性能做了大量的工作。

14.7.1.1 TiDB

- SQL 查询优化器
  - 调整代价模型
  - Analyze 下推
  - 函数签名下推
- 优化内部数据格式，减小中间结果大小
- 提升 MySQL 兼容性
- 支持 NO_SQL_CACHE 语法，控制存储引擎对缓存的使用
- 重构 Hash Aggregator 算子，降低内存使用
- 支持 Stream Aggragator 算子

14.7.1.2 PD

- 支持基于读流量的热点调度
- 支持设置 Store 权重，以及基于权重的调度

14.7.1.3 TiKV

- Coprocessor 支持更多下推函数
- 支持取样操作下推
- 支持手动触发数据 Compact，用于快速回收空间
- 提升性能和稳定性
- 增加 Debug API，方便调试

14.7.1.4 TiSpark Beta Release

- 支持可配置框架
- 支持 ThriftSever/JDBC 和 Spark SQL 脚本入口

14.7.1.5 源码地址

源码地址

14.7.1.6 鸣谢

14.7.1.6.1 特别感谢参与项目的企业和团队

- Archon
- Mobike
- SpeedyCloud
- UCloud
- 腾讯云
- 韩国三星研究院

14.7.1.6.2 感谢以下组织/个人提供出色的开源软件/服务

- Asta Xie
- CNCF
- CoreOS
- Databricks
- Docker
- Github
- Grafana
- gRPC
- Jepsen
- Kubernetes
- Namazu
- Prometheus
- RedHat
- RocksDB Team
- Rust Team

14.7.1.6.3 感谢社区个人贡献者 TiDB Contributor

- 8cbx
- Akihiro Suda
- aliyx
- alston111111
- andelf
- Andy Librian
- Arthur Yang
- astaxie
- Bai, Yang

- bailaohe
- Bin Liu
- Blame cosmos
- Breezewish
- Carlos Ferreira
- Ce Gao
- Changjian Zhang
- Cheng Lian
- Cholerae Hu
- Chu Chao
- coldwater
- Cole R Lawrence
- cuiqiu
- cuiyuan
- Cwen
- Dagang
- David Chen
- David Ding
- dawxy
- dcadevil
- Deshi Xiao
- Di Tang
- disksing
- dongxu
- dreamquster
- Drogon
- Du Chuan
- Dylan Wen
- eBoyy
- Eric Romano
- Ewan Chou
- Fiisio
- follitude
- Fred Wang
- follitude
- fud
- fudali
- gaoyangxiaozhu
- Gogs
- goroutine
- Gregory Ian
- Guanqun Lu
- Guilherme Hübner Franco
- Haibin Xie
- Han Fei

- Hiroaki Nakamura
- hiwjd
- Hongyuan Wang
- Hu Ming
- Hu Ziming
- Huachao Huang
- HuaiyuXu
- Huxley Hu
- iamxy
- Ian
- insion
- iroi44
- Ivan.Yang
- Jack Yu
- jacky liu
- Jan Mercl
- Jason W
- Jay
- Jay Lee
- Jianfei Wang
- Jiaxing Liang
- Jie Zhou
- jinhelin
- Jonathan Boulle
- Karl Ostendorf
- knarfeh
- Kuiba
- leixuechun
- li
- Li Shihai
- Liao Qiang
- Light
- lijian
- Lilian Lee
- Liqueur Librazy
- Liu Cong
- Liu Shaohui
- liubo0127
- liyanan
- lkk2003rty
- Louis
- louishust
- luckcolors
- Lynn
- Mae Huang

- maiyang
- maxwell
- mengshangqi
- Michael Belenchenko
- mo2zie
- morefreeze
- MQ
- mxlxm
- Neil Shen
- netroby
- ngaut
- Nicole Nie
- nolouch
- onlymellb
- overvenus
- PaladinTyrion
- paulg
- Priya Seth
- qgxiaozhan
- qhsong
- Qiannan
- qiuyesuifeng
- queenypingcap
- qupeng
- Rain Li
- ranxiaolong
- Ray
- Rick Yu
- shady
- ShawnLi
- Shen Li
- Sheng Tang
- Shirly
- Shuai Li
- ShuNing
- ShuYu Wang
- siddontang
- silenceper
- Simon J Mudd
- Simon Xia
- skimmilk6877
- sllt
- soup
- Sphinx
- Steffen

- sumBug
- sunhao2017
- Tao Meng
- Tao Zhou
- tennix
- tiancaiamao
- TianGuangyu
- Tristan Su
- ueizhou
- UncP
- Unknwon
- v01dstar
- Van
- WangXiangUSTC
- wangyisong1996
- weekface
- wegel
- Wei Fu
- Wenbin Xiao
- Wenting Li
- Wenxuan Shi
- winkyao
- woodpenker
- wuxuelian
- Xiang Li
- xiaojian cai
- Xuanjia Yang
- Xuanwo
- XuHuaiyu
- Yang Zhexuan
- Yann Autissier
- Yanzhe Chen
- Yiding Cui
- Yim
- youyouhu
- Yu Jun
- Yuwen Shen
- Zejun Li
- Zhang Yuning
- zhangjinpeng1987
- ZHAO Yijun
- ZhengQian
- ZhengQianFang
- zhengwanbo
- Zhe-xuan Yang

- ZhiFeng Hu
- Zhiyuan Zheng
- Zhou Tao
- Zhoubirdblue
- zhouningnan
- Ziyi Yan
- zs634134578
- zyguan
- zz-jason
- qiukeren
- hawkingrei
- wangyanjun
- zxylvlp

14.7.2 TiDB Pre-GA Release Notes

2017 年 8 月 30 日，TiDB 发布 Pre-GA 版。该版本对 MySQL 兼容性、SQL 优化器、系统稳定性、性能做了大量的工作。

14.7.2.1 TiDB

- SQL 查询优化器
  - 调整代价模型
  - 优化索引选择，支持不同类型字段比较的索引选择
  - 支持基于贪心算法的 Join Reorder
- 大量 MySQL 兼容性相关功能
- 支持 Natural Join
- 完成 JSON 类型支持 (Experimental)，包括对 JSON 中的字段查询、更新、建索引
- 裁剪无用数据，减小执行器内存消耗
- 支持在 SQL 语句中设置优先级，并根据查询类型自动设置部分语句的优先级
- 完成表达式重构，执行速度提升 30% 左右

14.7.2.2 PD

- 支持手动切换 PD 集群 Leader

14.7.2.3 TiKV

- Raft Log 使用独立的 RocksDB 实例
- 使用 DeleteRange 加快删除副本速度
- Coprocessor 支持更多运算符下推
- 提升性能，提升稳定性

14.7.2.4 TiSpark Beta Release

- 支持谓词下推
- 支持聚合下推
- 支持范围裁剪
- 通过 TPC-H 测试 (除去一个需要 View 的 Query)

14.7.3 TiDB RC4 Release Notes

2017 年 8 月 4 日，TiDB 正式发布 RC4 版。该版本对 MySQL 兼容性、SQL 优化器、系统稳定性、性能做了大量的工作。性能方面重点优化了写入速度，计算任务调度支持优先级，避免分析型大事务影响在线事务。SQL 优化器全新改版，查询代价估算更加准确，且能够自动选择 Join 物理算子。功能方面进一步 MySQL 兼容性。同时为了更好的支持 OLAP 业务，开源了 TiSpark 项目，可以通过 Spark 读取和分析 TiKV 中的数据。

14.7.3.1 TiDB

- SQL 查询优化器重构
  - 更好的支持 TopN 查询
  - 支持 Join 算子根据代价自动选择
  - 更完善的 Projection Elimination
- Schema 版本检查区分 Table，避免 DDL 干扰其他正在执行的事务
- 支持 BatchIndexJoin
- 完善 Explain 语句
- 提升 Index Scan 性能
- 大量 MySQL 兼容性相关功能
- 支持 Json 类型及其操作
- 支持查询优先级、隔离级别的设置

14.7.3.2 PD

- 支持通过 PD 设置 TiKV location labels
- 调度优化
  - 支持 PD 主动向 TiKV 下发调度命令
  - 加快 region heartbeat 响应速度
  - 优化 balance 算法
- 优化数据加载，加快 failover 速度

14.7.3.3 TiKV

- 支持查询优先级设置
- 支持 RC 隔离级别
- 完善 Jepsen，提升稳定性

- 支持 Document Store
- Coprocessor 支持更多下推函数
- 提升性能，提升稳定性

14.7.3.4 TiSpark Beta Release

- 支持谓词下推
- 支持聚合下推
- 支持范围裁剪
- 通过 TPC-H 测试 (除去一个需要 View 的 Query)

14.7.4 TiDB RC3 Release Notes

2017 年 6 月 16 日，TiDB 正式发布 RC3 版。该版本对 MySQL 兼容性、SQL 优化器、系统稳定性、性能做了大量的工作。性能方面重点优化了负载均衡调度策略和流程。功能方面进一步完善权限管理功能，用户可以按照 MySQL 的权限管理方式控制数据访问权限。另外 DDL 的速度也得到显著的提升。同时为了简化运维工作，开源了 TiDB Ansible 项目，可以一键部署/升级/启停 TiDB 集群。

14.7.4.1 TiDB

- SQL 查询优化器
  - 统计信息收集和使用
  - 关联子查询优化
  - 优化 CBO 框架
  - 通过 Unique Key 信息消除聚合
  - 重构 Expression
  - Distinct 转换为 GroupBy
  - 支持 topn 操作下推
- 支持基本权限管理
- 新增大量 MySQL 内建函数
- 完善 Alter Table 语句，支持修改表名、默认值、注释
- 支持 Create Table Like 语句
- 支持 Show Warnings 语句
- 支持 Rename Table 语句
- 限制单个事务大小，避免大事务阻塞整个集群
- Load Data 过程中对数据进行自动拆分
- 优化 AddIndex、 Delete 语句性能
- 支持 “ANSI_QUOTES” sql_mode
- 完善监控
- 修复 Bug
- 修复内存泄漏问题

14.7.4.2 PD

- 支持 Label 对副本进行 Location 调度
- 基于 region 数量的快速调度
- pd-ctl 支持更多功能
  - 添加、删除 PD
  - 通过 Key 获取 Region 信息
  - 添加、删除 scheduler 和 operator
  - 获取集群 label 信息

14.7.4.3 TiKV

- 支持 Async Apply 提升整体写入性能
- 使用 prefix seek 提升 Write CF 的读取性能
- 使用 memory hint prefix 提升 Raft CF 插入性能
- 优化单行读事务性能
- 支持更多下推功能
- 加入更多统计
- 修复 Bug

14.7.5 TiDB RC2 Release Notes

2017 年 3 月 1 日，TiDB 正式发布 RC2 版。该版本对 MySQL 兼容性、SQL 优化器、系统稳定性、性能做了大量的工作。对于 OLTP 场景，读取性能提升 60%，写入性能提升 30%。另外提供了权限管理功能，用户可以按照 MySQL 的权限管理方式控制数据访问权限。

14.7.5.1 TiDB

- SQL 查询优化器
  - 统计信息收集和使用
  - 关联子查询优化
  - 优化 CBO 框架
  - 通过 Unique Key 信息消除聚合
  - 重构 Expression
  - Distinct 转换为 GroupBy
  - 支持 topn 操作下推
- 支持基本权限管理
- 新增大量 MySQL 内建函数
- 完善 Alter Table 语句，支持修改表名、默认值、注释
- 支持 Create Table Like 语句
- 支持 Show Warnings 语句
- 支持 Rename Table 语句
- 限制单个事务大小，避免大事务阻塞整个集群

- Load Data 过程中对数据进行自动拆分
- 优化 AddIndex、 Delete 语句性能
- 支持 “ANSI_QUOTES” sql_mode
- 完善监控
- 修复 Bug
- 修复内存泄漏问题

14.7.5.2 PD

- 支持 Label 对副本进行 Location 调度
- 基于 region 数量的快速调度
- pd-ctl 支持更多功能
  - 添加、 删除 PD
  - 通过 Key 获取 Region 信息
  - 添加、 删除 scheduler 和 operator
  - 获取集群 label 信息

14.7.5.3 TiKV

- 支持 Async Apply 提升整体写入性能
- 使用 prefix seek 提升 Write CF 的读取性能
- 使用 memory hint prefix 提升 Raft CF 插入性能
- 优化单行读事务性能
- 支持更多下推功能
- 加入更多统计
- 修复 Bug

14.7.6 TiDB RC1 Release Notes

2016 年 12 月 23 日，分布式关系型数据库 TiDB 正式发布 RC1。

14.7.6.1 TiKV

- 提升写入速度
- 降低磁盘空间占用
- 支持百 TB 级别数据
- 提升稳定性，集群规模支持 200 个节点
- 提供 Raw KV API，以及 Golang client

14.7.6.2 PD

- PD 调度策略框架优化，策略更加灵活合理
- 添加 label 支持，支持跨 DC 调度
- 提供 PD Controler，方便操作 PD 集群

14.7.6.3 TiDB

- SQL 查询优化器
  - 支持 eager aggregate
  - 更详细的 explain 信息
  - union 算子并行化
  - 子查询性能优化
  - 条件下推优化
  - 优化 CBO 框架
- 重构 time 相关类型的实现，提升和 MySQL 的兼容性
- 支持更多的 MySQL 内建函数
- Add Index 语句提速
- 支持用 change column 语句修改列名；支持使用 Alter table 的 modify column 和 change column 完成部分列类型转换

14.7.6.4 工具

- Loader：兼容 Percona 的 Mydumper 数据格式，提供多线程导入、出错重试、断点续传等功能，并且针对 TiDB 有优化
- 开发完成一键部署工具

15 术语表

15.1 A

15.1.1 ACID

ACID 是指数据库管理系统在写入或更新资料的过程中，为保证事务是正确可靠的，所必须具备的四个特性：原子性 (atomicity)、 一致性 (consistency)、 隔离性（isolation）以及持久性（durability）。

- 原子性 (atomicity) 指一个事务中的所有操作，或者全部完成，或者全部不完成，不会结束在中间某个环节。TiDB 通过 Primary Key 所在Region 的原子性来保证分布式事务的原子性。
- 一致性 (consistency) 指在事务开始之前和结束以后，数据库的完整性没有被破坏。TiDB 在写入数据之前，会校验数据的一致性，校验通过才会写入内存并返回成功。
- 隔离性 (isolation) 指数据库允许多个并发事务同时对其数据进行读写和修改的能力。隔离性可以防止多个事务并发执行时由于交叉执行而导致数据的不一致，主要用于处理并发场景。TiDB 目前只支持一种隔离级别，即可重复读。
- 持久性 (durability) 指事务处理结束后，对数据的修改就是永久的，即便系统故障也不会丢失。在 TiDB 中，事务一旦提交成功，数据全部持久化存储到 TiKV，此时即使 TiDB 服务器宕机也不会出现数据丢失。

15.2 L

15.2.1 Leader/Follower/Learner

它们分别对应Peer 的三种角色。其中 Leader 负责响应客户端的读写请求；Follower 被动地从 Leader 同步数据，当 Leader 失效时会进行选举产生新的 Leader；Learner 是一种特殊的角色，它只参与同步 raft log 而不参与投票，在目前的实现中只短暂存在于添加副本的中间步骤。

15.3 O

15.3.1 Old value

Old value 特指在 TiCDC 输出的增量变更日志中的“原始值”。可以通过配置来指定 TiCDC 输出的增量变更日志是否包含“原始值”。

15.3.2 Operator

Operator 是应用于一个 Region 的，服务于某个调度目的的一系列操作的集合。例如“将 Region 2 的 Leader 迁移至 Store 5”，“将 Region 2 的副本迁移到 Store 1, 4, 5”等。

Operator 可以是由 Scheduler 通过计算生成的，也可以是由外部 API 创建的。

15.3.3 Operator Step

Operator Step 是 Operator 执行过程的一个步骤，一个 Operator 常常会包含多个 Operator Step。

目前 PD 可生成的 Step 包括：

- `TransferLeader`：将 Region Leader 迁移至指定 Peer
- `AddPeer`：在指定 Store 添加 Follower
- `RemovePeer`：删除一个 Region Peer
- `AddLearner`：在指定 Store 添加 Region Learner
- `PromoteLearner`：将指定 Learner 提升为 Follower
- `SplitRegion`：将指定 Region 一分为二

15.4 P

15.4.1 Pending/Down

Pending 和 Down 是 Peer 可能出现的两种特殊状态。其中 Pending 表示 Follower 或 Learner 的 raft log 与 Leader 有较大差距，Pending 状态的 Follower 无法被选举成 Leader。Down 是指 Leader 长时间没有收到对应 Peer 的消息，通常意味着对应节点发生了宕机或者网络隔离。

15.5 R

15.5.1 Region/Peer/Raft Group

每个 Region 负责维护集群的一段连续数据（默认配置下平均约 96 MiB），每份数据会在不同的 Store 存储多个副本（默认配置是 3 副本），每个副本称为 Peer。同一个 Region 的多个 Peer 通过 raft 协议进行数据同步，所以 Peer 也用来指代 raft 实例中的成员。TiKV 使用 multi-raft 模式来管理数据，即每个 Region 都对应一个独立运行的 raft 实例，我们也把这样的一个 raft 实例叫做一个 Raft Group。

15.5.2 Region Split

TiKV 集群中的 Region 不是一开始就划分好的，而是随着数据写入逐渐分裂生成的，分裂的过程被称为 Region Split。

其机制是集群初始化时构建一个初始 Region 覆盖整个 key space，随后在运行过程中每当 Region 数据达到一定量之后就通过 Split 产生新的 Region。

15.5.3 Restore

备份操作的逆过程，即利用保存的备份数据还原出原始数据的过程。

15.6 S

15.6.1 Scheduler

Scheduler（调度器）是 PD 中生成调度的组件。PD 中每个调度器是独立运行的，分别服务于不同的调度目的。常用的调度器及其调用目标有：

- `balance-leader-scheduler`：保持不同节点的 Leader 均衡。
- `balance-region-scheduler`：保持不同节点的 Peer 均衡。
- `hot-region-scheduler`：保持不同节点的读写热点 Region 均衡。
- `evict-leader-{store-id}`：驱逐某个节点的所有 Leader。（常用于滚动升级）

15.6.2 Store

PD 中的 Store 指的是集群中的存储节点，也就是 tikv-server 实例。Store 与 TiKV 实例是严格一一对应的，即使在同一主机甚至同一块磁盘部署多个 TiKV 实例，这些实例也对会对应不同的 Store。
