TiDB 中文手册

PingCAP Inc.

20220825

目录

1 关于 TiDB

1.1 TiDB 简介

TiDB 是 PingCAP 公司设计的开源分布式 HTAP (Hybrid Transactional and Analytical Processing) 数据库，结合了传统的 RDBMS 和 NoSQL 的最佳特性。TiDB 兼容 MySQL，支持无限的水平扩展，具备强一致性和高可用性。TiDB 的目标是为 OLTP (Online Transactional Processing) 和 OLAP (Online Analytical Processing) 场景提供一站式的解决方案。

TiDB 具备如下特性：

- 高度兼容 MySQL

  大多数情况下，无需修改代码即可从 MySQL 轻松迁移至 TiDB，分库分表后的 MySQL 集群亦可通过 TiDB 工具进行实时迁移。

- 水平弹性扩展

  通过简单地增加新节点即可实现 TiDB 的水平扩展，按需扩展吞吐或存储，轻松应对高并发、海量数据场景。

- 分布式事务

  TiDB 100% 支持标准的 ACID 事务。

- 真正金融级高可用

  相比于传统主从 (M-S) 复制方案，基于 Raft 的多数派选举协议可以提供金融级的 100% 数据强一致性保证，且在不丢失大多数副本的前提下，可以实现故障的自动恢复 (auto-failover)，无需人工介入。

- 一站式 HTAP 解决方案

  TiDB 作为典型的 OLTP 行存数据库，同时兼具强大的 OLAP 性能，配合 TiSpark，可提供一站式 HTAP 解决方案，一份存储同时处理 OLTP & OLAP，无需传统繁琐的 ETL 过程。

- 云原生 SQL 数据库

  TiDB 是为云而设计的数据库，支持公有云、私有云和混合云，配合 TiDB Operator 项目 可实现自动化运维，使部署、配置和维护变得十分简单。

TiDB 的设计目标是 100% 的 OLTP 场景和 80% 的 OLAP 场景，更复杂的 OLAP 分析可以通过TiSpark 项目来完成。

TiDB 对业务没有任何侵入性，能优雅地替换传统的数据库中间件、数据库分库分表等 Sharding 方案。同时它也让开发运维人员不用关注数据库 Scale 的细节问题，专注于业务开发，极大地提升研发的生产力。

三篇文章了解 TiDB 技术内幕：

- 说存储
- 说计算
- 谈调度

1.1.1 部署方式

TiDB 可以部署在本地和云平台上，支持公有云、私有云和混合云。你可以根据实际场景或需求，选择相应的方式来部署 TiDB 集群：

- 使用 Ansible 部署：如果用于生产环境，推荐使用 TiDB Ansible 部署 TiDB 集群。
- 使用 Ansible 离线部署：如果部署环境无法访问网络，可使用 TiDB Ansible 进行离线部署。
- 使用 TiDB Operator 部署：使用 TiDB Operator 在 Kubernetes 集群上部署生产就绪的 TiDB 集群，支持部署到 AWS EKS、部署到谷歌云 GKE (beta)、部署到阿里云 ACK 等。
- 使用 Docker Compose 部署：如果你只是想测试 TiDB、体验 TiDB 的特性，或者用于开发环境，可以使用 Docker Compose 在本地快速部署 TiDB 集群。该部署方式不适用于生产环境。
- 使用 Docker 部署：你可以使用 Docker 部署 TiDB 集群，但该部署方式不适用于生产环境。
- 使用 TiDB Operator 部署到 Minikube：你可以使用 TiDB Operator 将 TiDB 集群部署到本地 Minikube 启动的 Kubernetes 集群中。该部署方式不适用于生产环境。
- 使用 TiDB Operator 部署到 kind：你可以使用 TiDB Operator 将 TiDB 集群部署到以 kind 方式启动的 Kubernetes 本地集群中。该部署方式不适用于生产环境。

1.1.2 项目源码

TiDB 集群所有组件的源码均可从 GitHub 上直接访问：

- TiDB
- TiKV
- PD
- TiSpark
- TiDB Operator

1.2 Benchmark 测试

1.2.1 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。

1.2.1.1 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
RAM	128GB
DISK	Intel Optane SSD P4800X 375G * 1
NIC	10Gb Ethernet

1.2.1.2 测试方案

1.2.1.2.1 TiDB 版本信息

组件	GitHash
TiDB	7a240818d19ae96e4165af9ea35df92466f59ce6
TiKV	e26ceadcdfe94fb6ff83b5abb614ea3115394bcd
PD	5e81548c3c1a1adab056d977e7767307a39ecb70

1.2.1.2.2 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

1.2.1.2.3 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
```

1.2.1.2.4 TiKV 配置

升高 TiKV 的日志级别同样有利于提高性能表现。

由于 TiKV 是以集群形式部署的，在 Raft 算法的作用下，能保证大多数节点已经写入数据。因此，除了对数据安全极端敏感的场景之外，raftstore 中的 `sync-log` 选项可以关闭。

TiKV 集群存在两个 Column Family（Default CF 和 Write CF），主要用于存储不同类型的数据。对于 Sysbench 测试，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"
[raftstore]
sync-log = false
[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"
[raftstore]
sync-log = false
[storage.block-cache]
capacity = "30GB"
```

更详细的 TiKV 参数调优请参考TiKV 性能参数调优。

1.2.1.3 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。

1.2.1.3.1 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
```

1.2.1.3.2 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
```

1.2.1.3.3 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;
```

1.2.1.3.4 Point select 测试命令

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

1.2.1.3.5 Update index 测试命令

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

1.2.1.3.6 Read-only 测试命令

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

1.2.1.4 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：

1.2.1.4.1 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

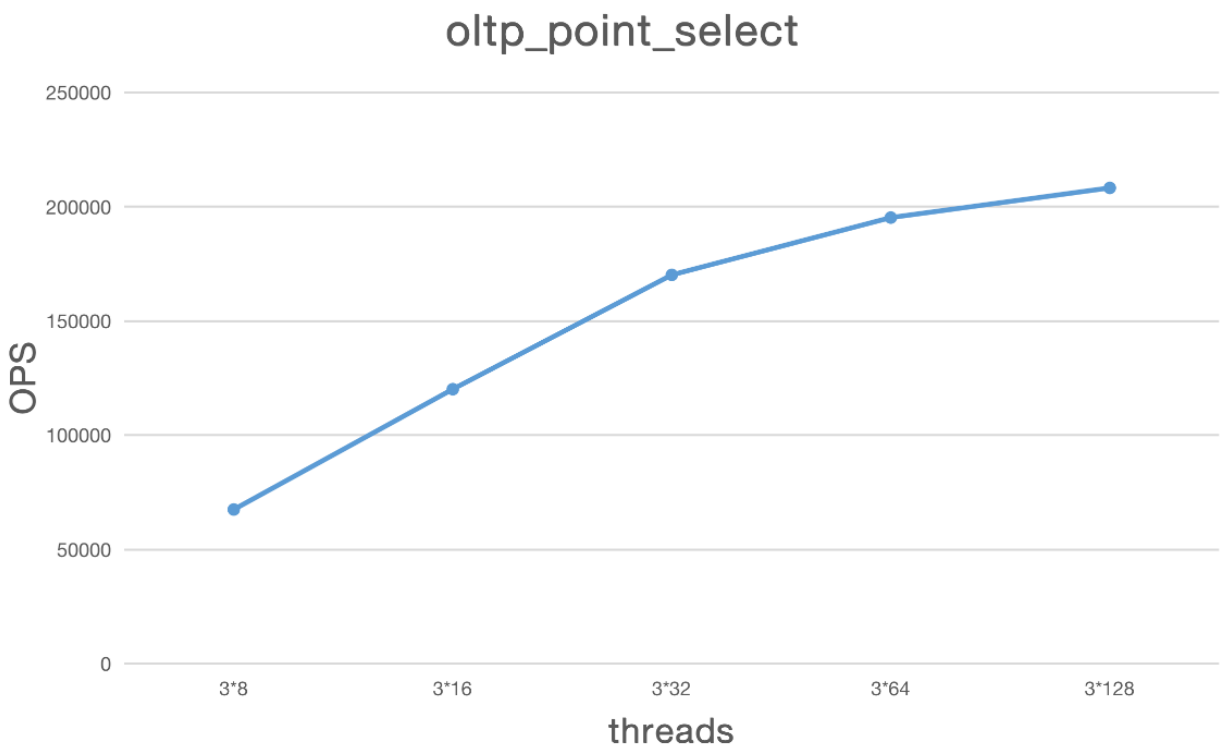


图 1: oltp_point_select

1.2.1.4.2 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

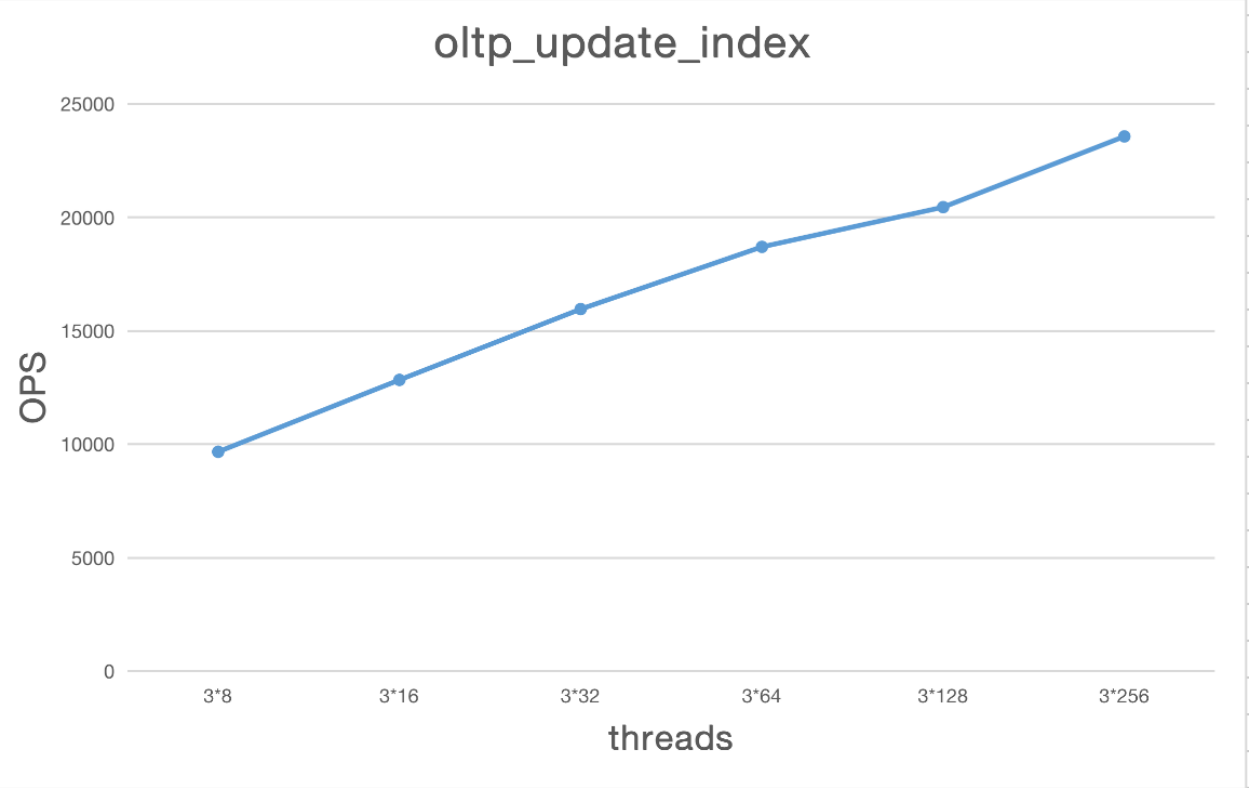


图 2: oltp_update_index

1.2.1.4.3 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

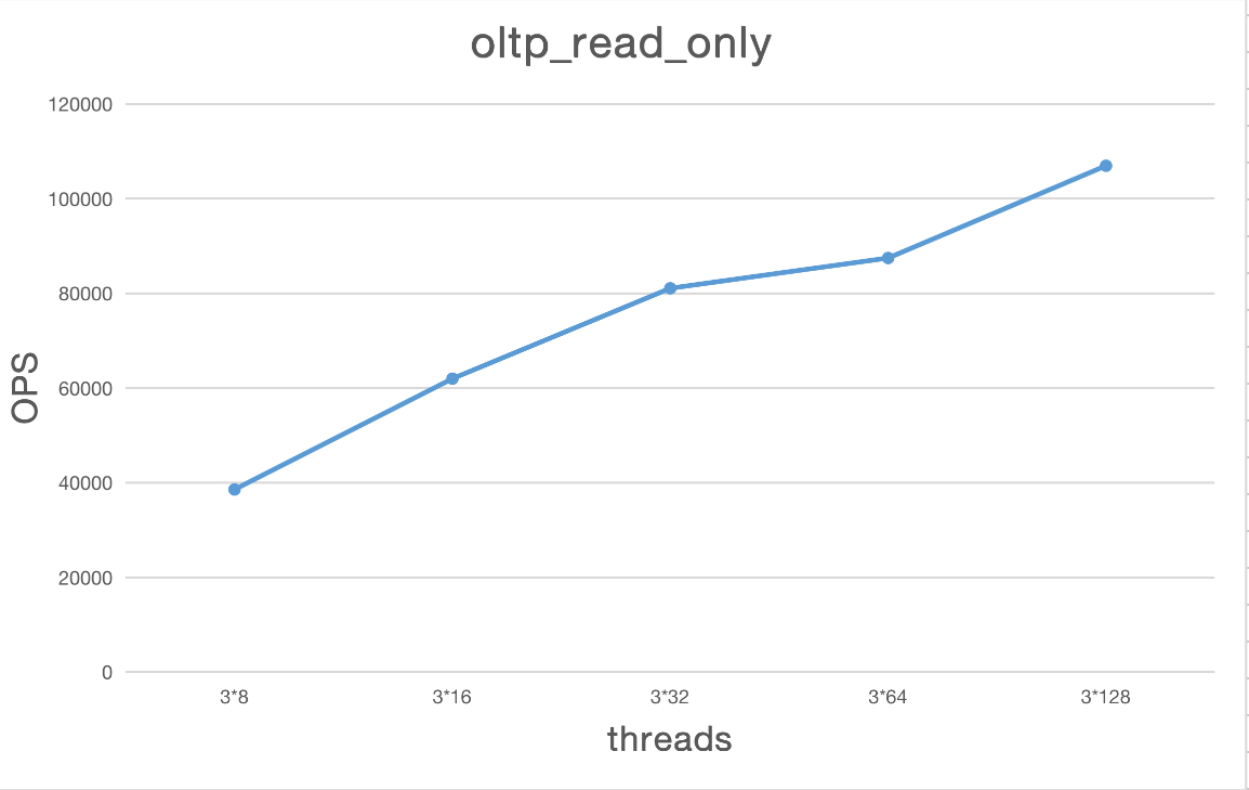


图 3: oltp_read_only

1.2.1.5 常见问题

1.2.1.5.1 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。

1.2.1.5.2 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。

1.2.1.5.3 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。

1.2.2 如何对 TiDB 进行 TPC-C 测试

本文介绍如何对 TiDB 进行 TPC-C 测试。

1.2.2.1 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
```

1.2.2.2 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。

1.2.2.3 配置调整

1.2.2.3.1 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
```

1.2.2.3.2 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 调优说明进行调整。

1.2.2.3.3 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
```

1.2.2.4 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
```

1.2.2.4.1 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。

1.2.2.4.2 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（以下简称 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 Lightning 导入数据，所以 csv 文件名最好符合 Lightning 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 Lightning 的要求。

生成 csv 文件

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

通过 Lightning 导入

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

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

启动

- 登录到部署 Lightning 和 Importer 的服务器
- 进入部署目录
- 在 Importer 目录下执行 `scripts/start_importer.sh`，启动 Importer
- 在 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 Lightning 的导入进度，或者通过日志查看导入是否结束。

1.2.2.4.3 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。

1.2.2.5 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。

1.2.3 TiDB Sysbench 性能对比测试报告 - v3.0 对比 v2.1

1.2.3.1 测试目的

对比 TiDB 3.0 版本和 2.1 版本在 OLTP 场景下的性能。

1.2.3.2 测试版本、时间、地点

TiDB 版本：v3.0.0 vs. v2.1.13

时间：2019 年 6 月

地点：北京

1.2.3.3 测试环境

测试在 AWS EC2 上进行，使用 CentOS-7.6.1810-Nitro (ami-028946f4cffc8b916) 镜像，各组件实例类型如下：

组件	实例类型
PD	r5d.xlarge
TiKV	c5d.4xlarge
TiDB	c5.4xlarge

Sysbench 版本：1.0.17

1.2.3.4 测试方案

使用 Sysbench 向集群导入 16 张表，每张数据 1000 万。起 3 个 sysbench 分别向 3 个 TiDB 发压，请求并发数逐步增加，单次测试时间 5 分钟。

准备数据命令：

```
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 \
    --mysql-password=password \
    prepare --tables=16 --table-size=10000000
```

执行测试命令：

```
sysbench $testname \
    --threads=$threads \
    --time=300 \
    --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 \
    --mysql-password=password \
    run --tables=16 --table-size=10000000
```

1.2.3.4.1 TiDB 版本信息

1.2.3.4.2 v3.0.0

组件	GitHash
TiDB	8efbe62313e2c1c42fd76d35c6f020087eef22c2
TiKV	a467f410d235fa9c5b3c355e3b620f81d3ac0e0c
PD	70aaa5eee830e21068f1ba2d4c9bae59153e5ca3

1.2.3.4.3 v2.1.13

组件	GitHash
TiDB	6b5b1a6802f9b8f5a22d8aab24ac80729331e1bc
TiKV	b3cf3c8d642534ea6fa93d475a46da285cc6acbf
PD	886362ebfb26ef0834935afc57bcee8a39c88e54

1.2.3.4.4 TiDB 参数配置

2.1 和 3.0 中开启 prepared plan cache (出于优化考虑，2.1 的 point select 与 read write 并未开启)：

```
[prepared-plan-cache]
enabled = true
```

并设置全局变量：

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

此外 3.0 还做了如下配置：

```
[tikv-client]
max-batch-wait-time = 2000000
```

1.2.3.4.5 TiKV 参数配置

2.1 和 3.0 使用如下配置：

```
log-level = "error"
[readpool.storage]
normal-concurrency = 10
[server]
grpc-concurrency = 6
[rocksdb.defaultcf]
block-cache-size = "14GB"
[rocksdb.writecf]
block-cache-size = "8GB"
[rocksdb.lockcf]
block-cache-size = "1GB"
```

3.0 还做了如下配置：

```
[raftstore]
apply-pool-size = 3
store-pool-size = 3
```

1.2.3.4.6 集群拓扑

机器 IP	部署实例
172.31.8.8	3 * Sysbench
172.31.7.80, 172.31.5.163, 172.31.11.123	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

1.2.3.5 测试结果

1.2.3.5.1 Point Select 测试

v2.1

Threads	QPS	95% latency(ms)
150	240304.06	1.61
300	276635.75	2.97

Threads	QPS	95% latency(ms)
600	307838.06	5.18
900	323667.93	7.30
1200	330925.73	9.39
1500	336250.38	11.65

v3.0

Threads	QPS	95% latency(ms)
150	334219.04	0.64
300	456444.86	1.10
600	512177.48	2.11
900	525945.13	3.13
1200	534577.36	4.18
1500	533944.64	5.28

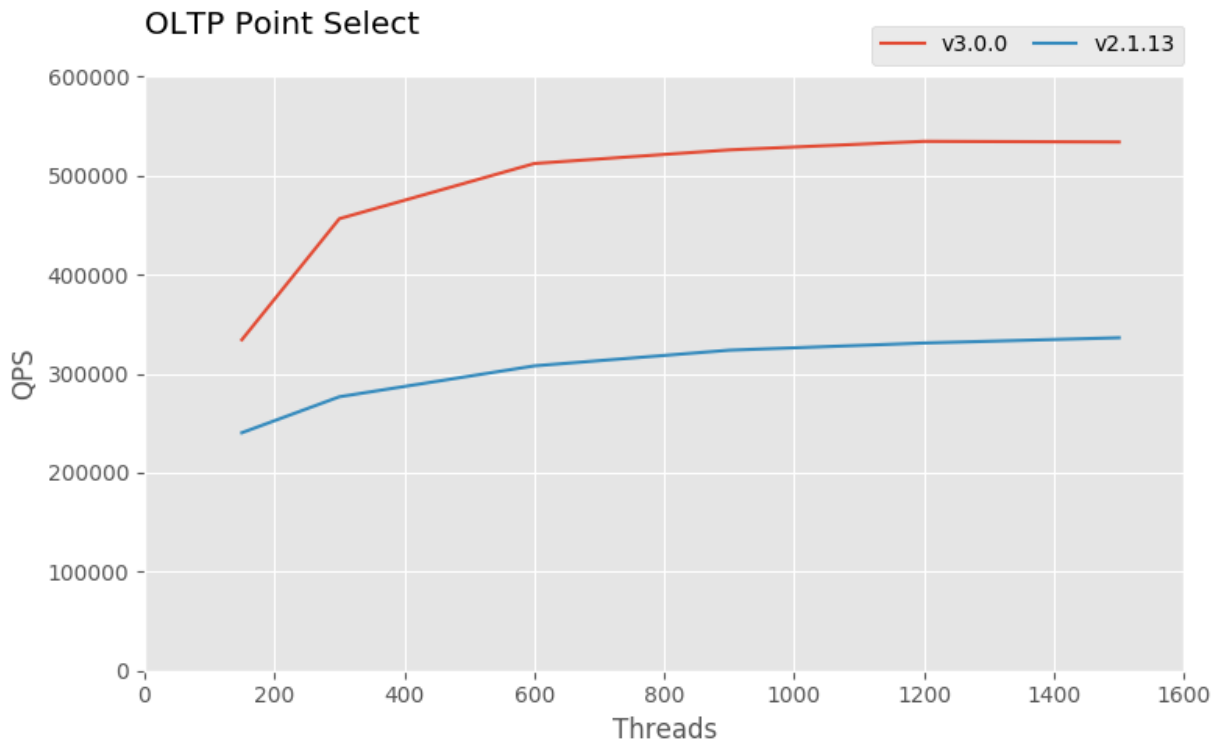


图 4: point select

1.2.3.5.2 Update Non-Index 测试

v2.1

threads	qps	95% latency(ms)
150	21785.37	8.58
300	28979.27	13.70
600	34629.72	24.83
900	36410.06	43.39
1200	37174.15	62.19
1500	37408.88	87.56

v3.0

threads	qps	95% latency(ms)
150	28045.75	6.67
300	39237.77	9.91
600	49536.56	16.71
900	55963.73	22.69
1200	59904.02	29.72
1500	62247.95	42.61

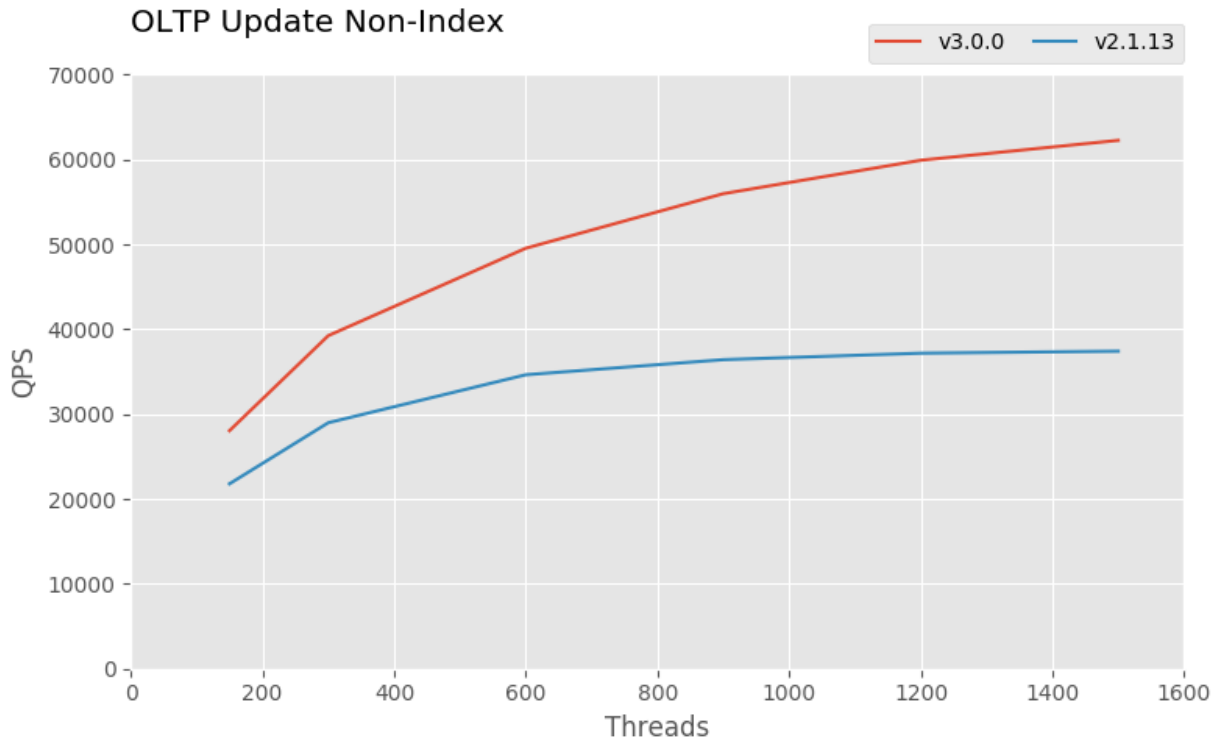


图 5: update non-index

1.2.3.5.3 Update Index 测试

v2.1

Threads	QPS	95% latency(ms)
150	14378.24	13.22
300	16916.43	24.38
600	17636.11	57.87
900	17740.92	95.81
1200	17929.24	130.13
1500	18012.80	161.51

v3.0

Threads	QPS	95% latency(ms)
150	19047.32	10.09
300	24467.64	16.71
600	28882.66	31.94
900	30298.41	57.87
1200	30419.40	92.42
1500	30643.55	125.52

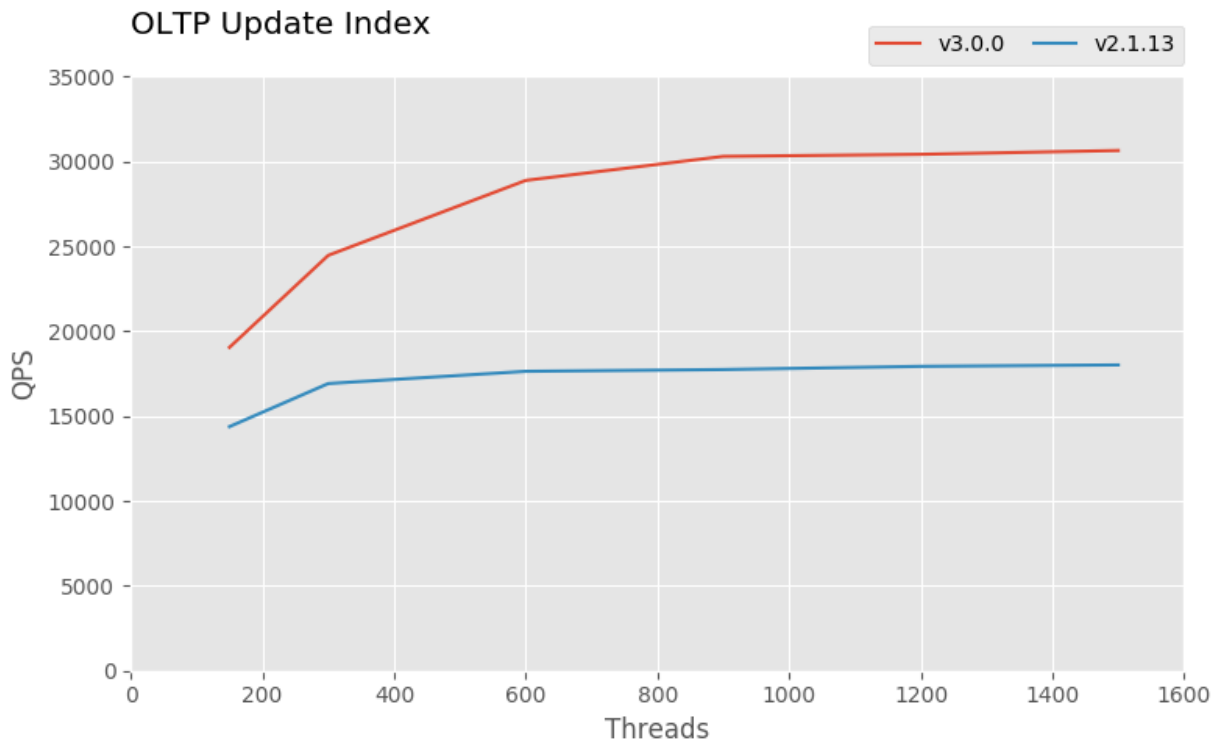


图 6: update index

1.2.3.5.4 Read Write 测试

v2.1

Threads	QPS	95% latency(ms)
150	85140.60	44.98
300	96773.01	82.96
600	105139.81	153.02
900	110041.83	215.44
1200	113242.70	277.21
1500	114542.19	337.94

v3.0

Threads	QPS	95% latency(ms)
150	105692.08	35.59
300	129769.69	58.92
600	141430.86	114.72
900	144371.76	170.48
1200	143344.37	223.34
1500	144567.91	277.21

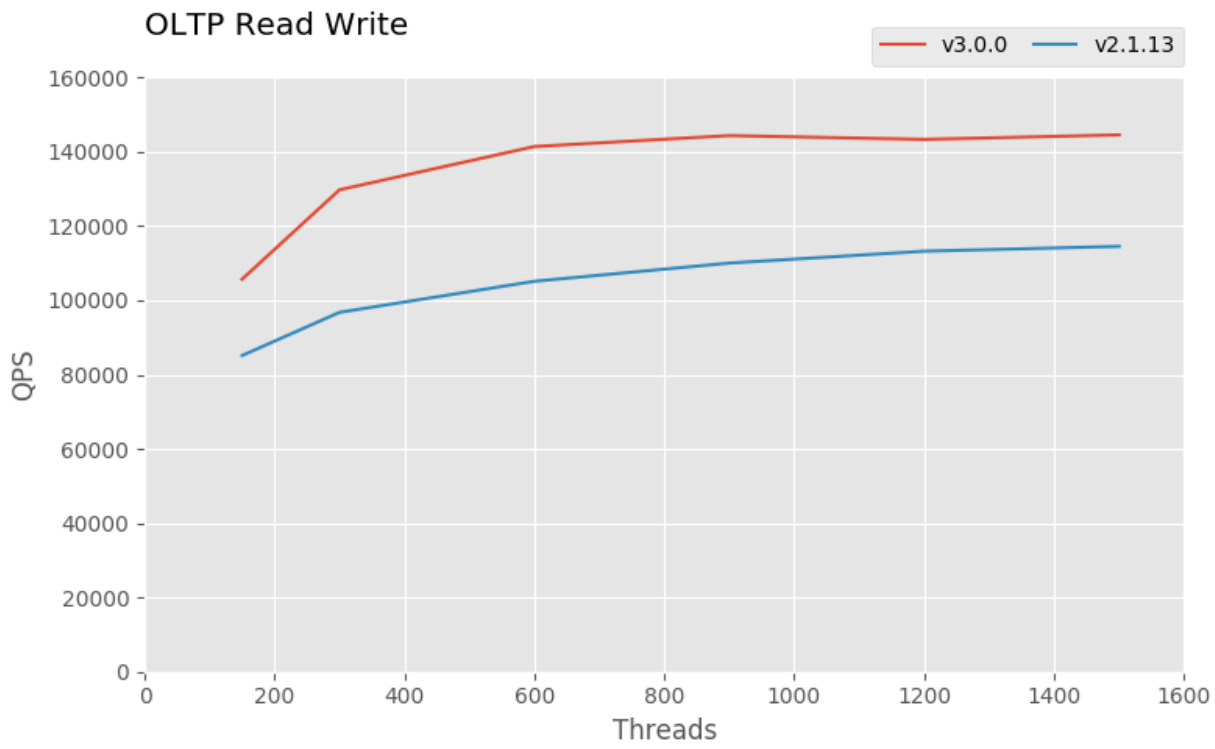


图 7: read write

1.2.4 TiDB TPC-C 性能对比测试报告 - v3.0 对比 v2.1

1.2.4.1 测试目的

对比 TiDB 3.0 版本和 2.1 版本的 TPC-C 性能表现。

1.2.4.2 测试版本、时间、地点

TiDB 版本：v3.0.0 vs. v2.1.13

时间：2019 年 6 月

地点：北京

1.2.4.3 测试环境

IDC 机器：

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	1.5TB SSD * 2

本文使用开源的 BenchmarkSQL 5.0 作为 TPC-C 测试工具并添加对 MySQL 协议支持，可以通过以下命令下载测试程序:

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

1.2.4.4 测试方案

使用 BenchmarkSQL 向集群导入 1000 warehouse 的数据。通过 HAProxy 代理，分别以递增并发数向集群发送请求，单次并发测试时间 10 分钟。

1.2.4.4.1 TiDB 版本信息

1.2.4.4.2 v3.0.0

组件	GitHash
TiDB	46c38e15eba43346fb3001280c5034385171ee20
TiKV	a467f410d235fa9c5b3c355e3b620f81d3ac0e0c
PD	70aaa5eee830e21068f1ba2d4c9bae59153e5ca3

1.2.4.4.3 v2.1.13

组件	GitHash
TiDB	6b5b1a6802f9b8f5a22d8aab24ac80729331e1bc
TiKV	b3cf3c8d642534ea6fa93d475a46da285cc6acbf
PD	886362ebfb26ef0834935afc57bcee8a39c88e54

1.2.4.4.4 TiDB 参数配置

```
[log]
level = "error"
[performance]
max-procs = 20
[prepared_plan_cache]
enabled = true
```

1.2.4.4.5 TiKV 参数配置

默认配置

1.2.4.4.6 集群拓扑

机器 IP	部署实例
172.16.4.75	2*TiDB 2*TiKV 1*pd
172.16.4.76	2*TiDB 2*TiKV 1*pd
172.16.4.77	2*TiDB 2*TiKV 1*pd

1.2.4.5 测试结果

版本	threads	tpmC
v3.0	128	44068.55
v3.0	256	47094.06
v3.0	512	48808.65
v2.1	128	10641.71
v2.1	256	10861.62
v2.1	512	10965.39

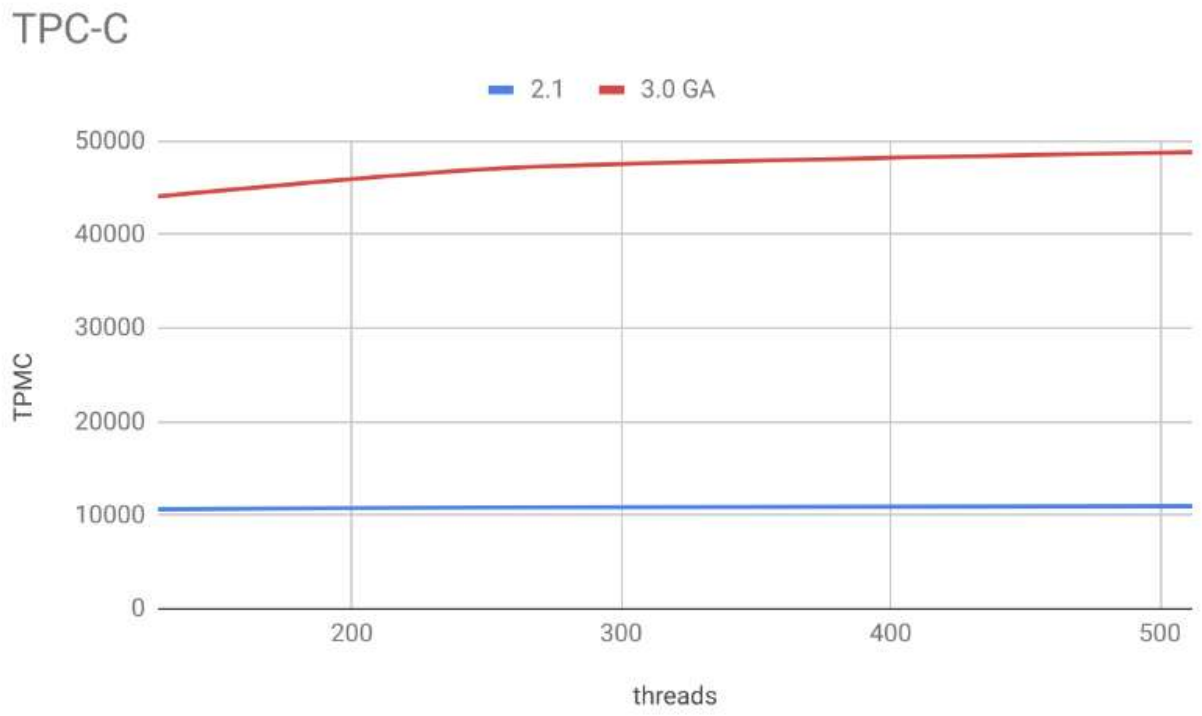


图 8: tpcc

v3.0 比 v2.1 在 TPC-C 性能上，提升了 450%。

1.2.5 线上负载与 ADD INDEX 相互影响测试

1.2.5.1 测试目的

测试 OLTP 场景下，ADD INDEX 与线上负载的相互影响。

1.2.5.2 测试版本、时间、地点

TiDB 版本：v3.0.1

时间：2019 年 7 月

地点：北京

1.2.5.3 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。

1.2.5.3.1 版本信息

组件	GitHash
TiDB	9e4e8da3c58c65123db5f26409759fe1847529f8
TiKV	4151dc8878985df191b47851d67ca21365396133
PD	811ce0b9a1335d1b2a049fd97ef9e186f1c9efc1

Sysbench 版本：1.0.17

1.2.5.3.2 TiDB 参数配置

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

1.2.5.3.3 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

1.2.5.3.4 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
```

1.2.5.4 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。

1.2.5.4.1 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

tidb_ddl_reorg_worker_cnt	add_index_durations(s)	sysbench TPS	sysbench QPS
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

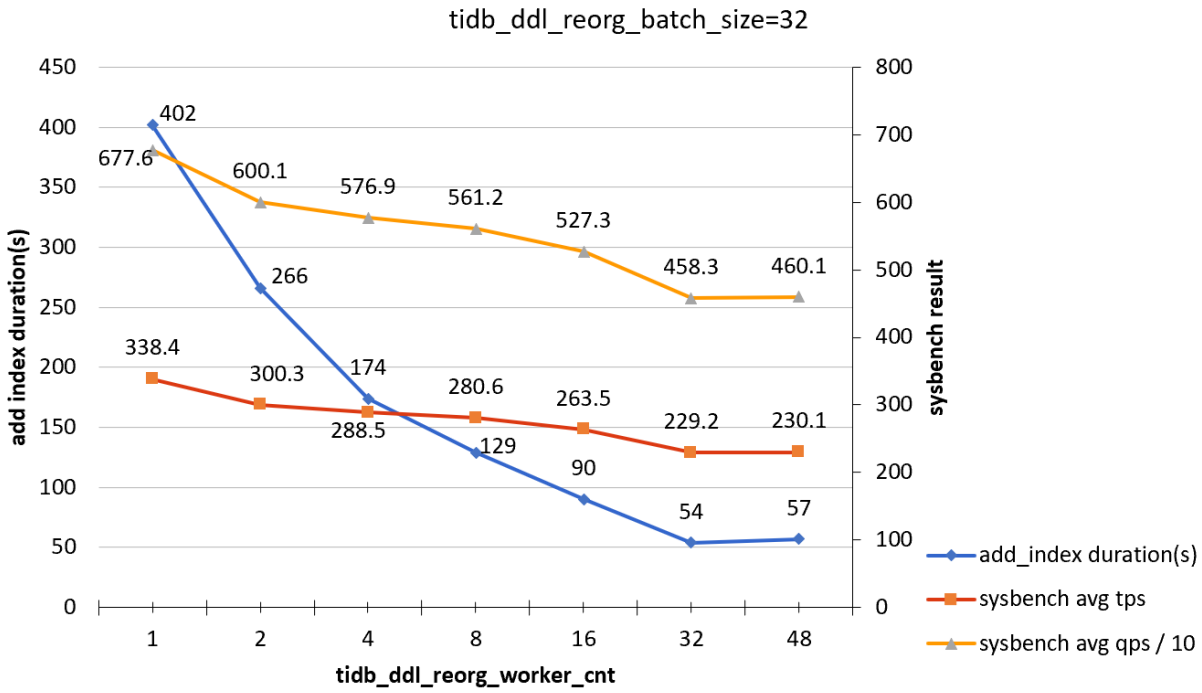


图 9: add-index-load-1-b32

```
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

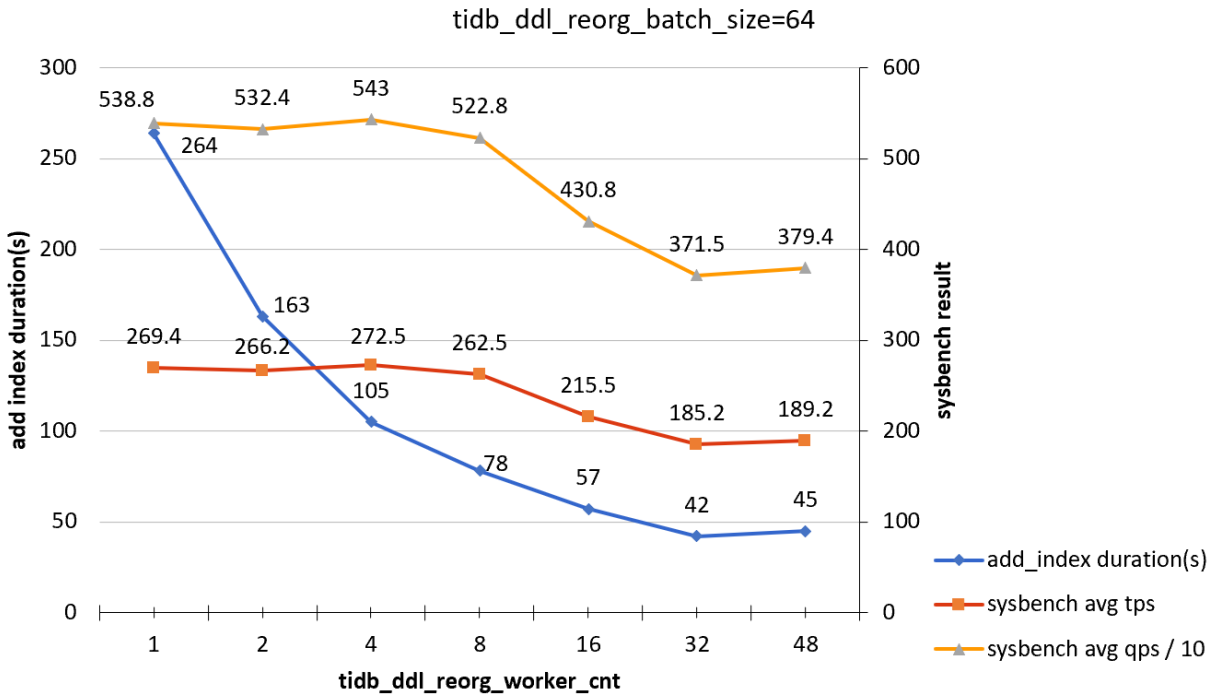


图 10: add-index-load-1-b64

`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

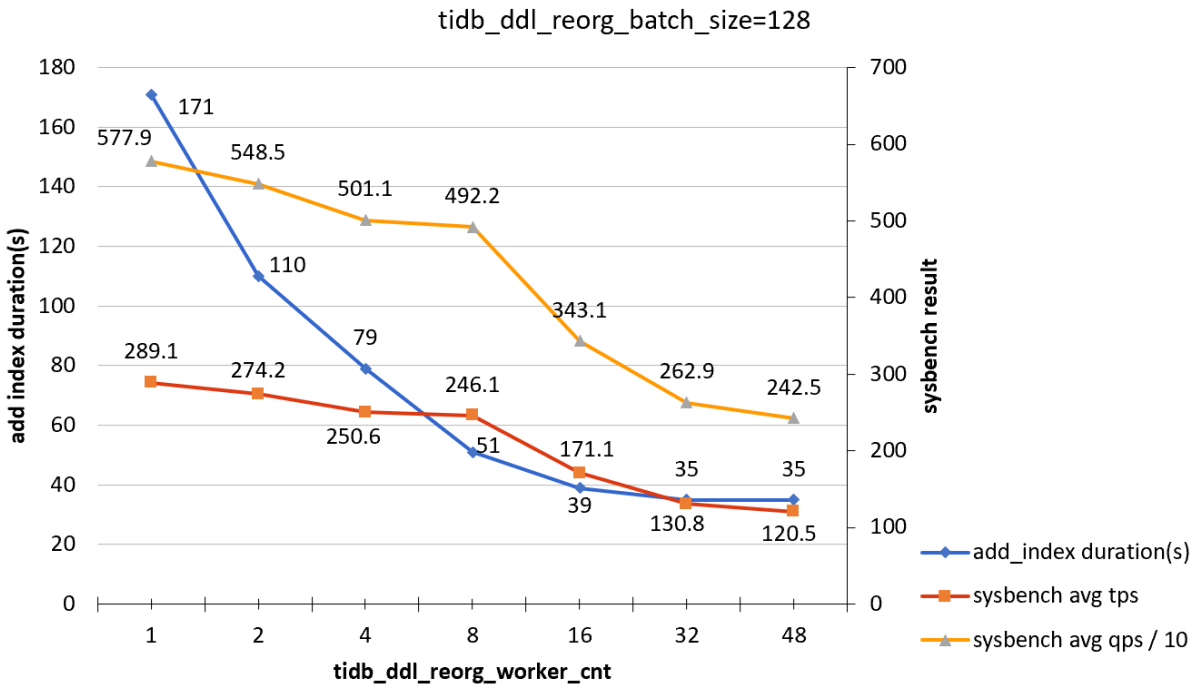


图 11: add-index-load-1-b128

```
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

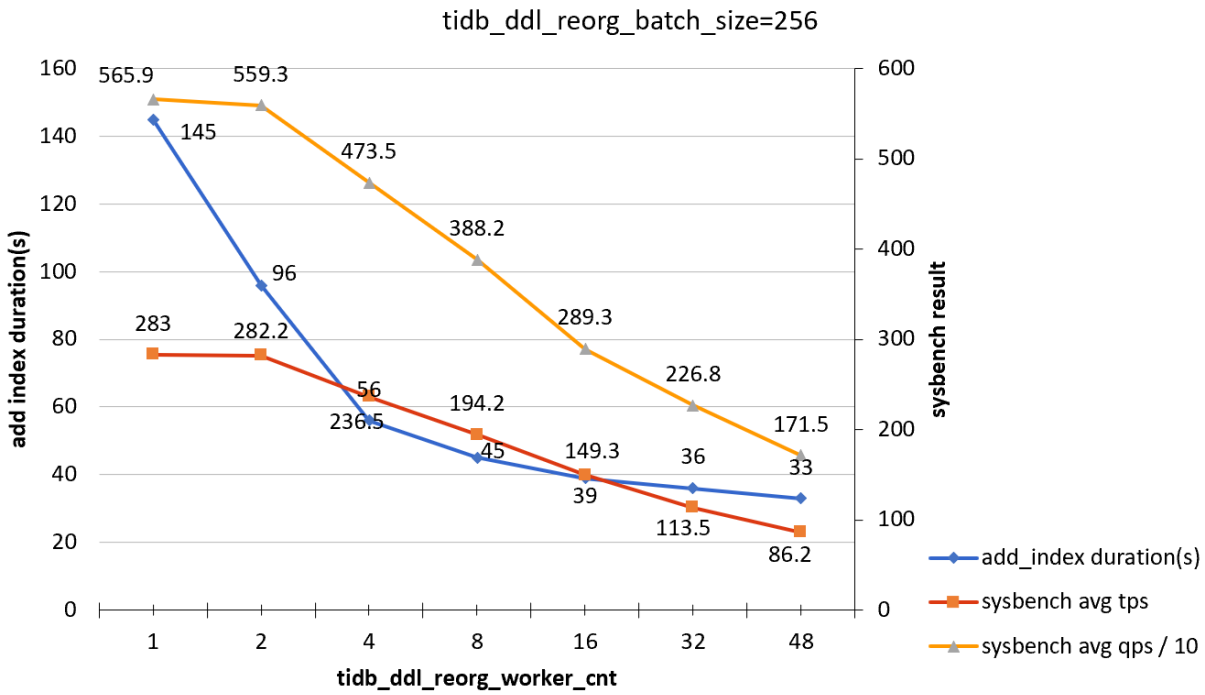


图 12: add-index-load-1-b256

`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

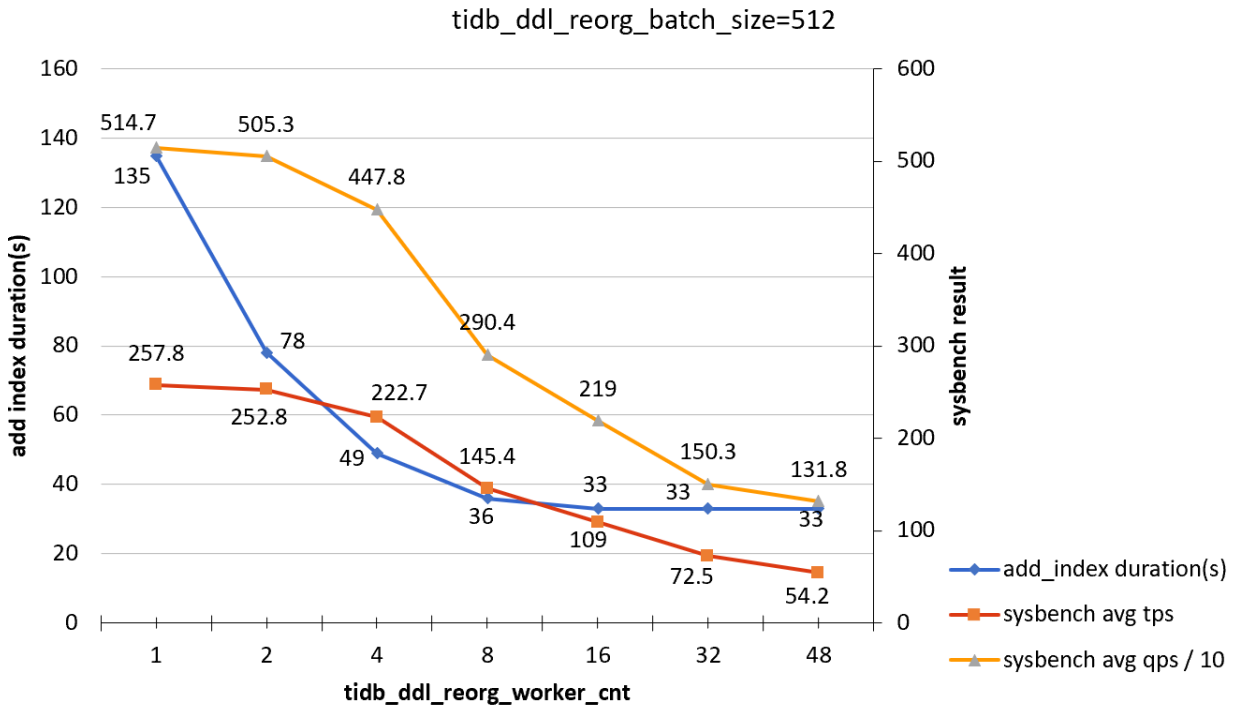


图 13: add-index-load-1-b512

`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

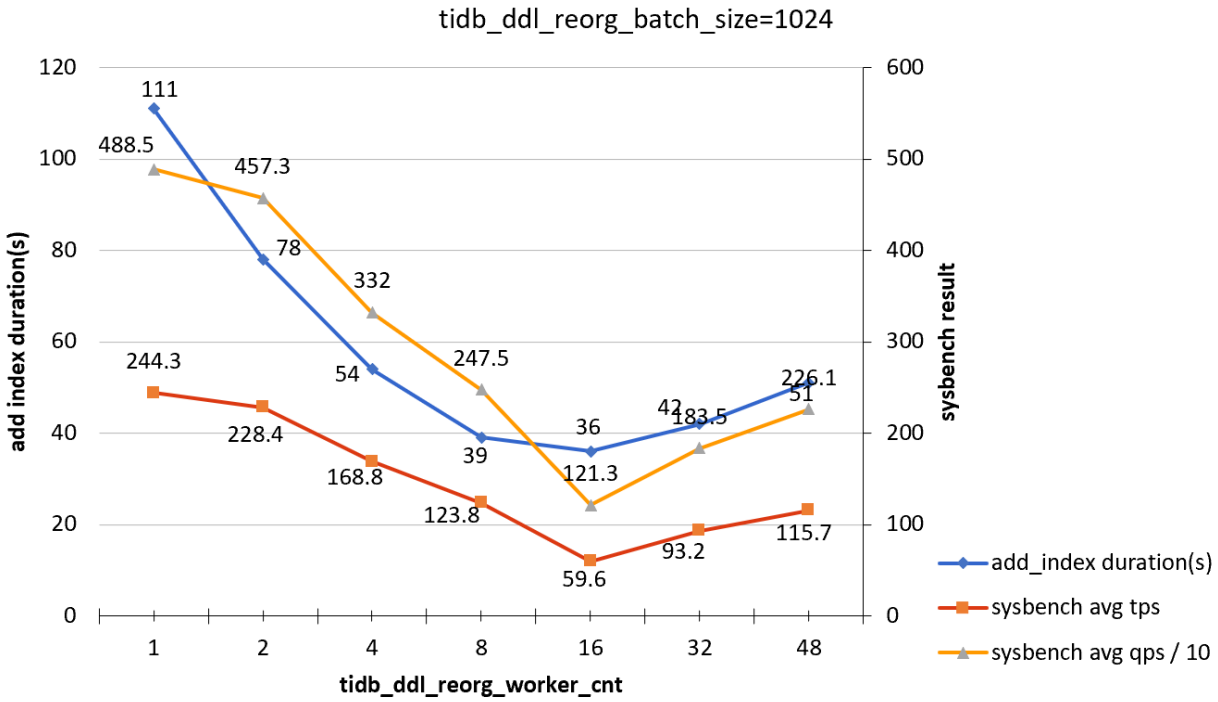


图 14: add-index-load-1-b1024

`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

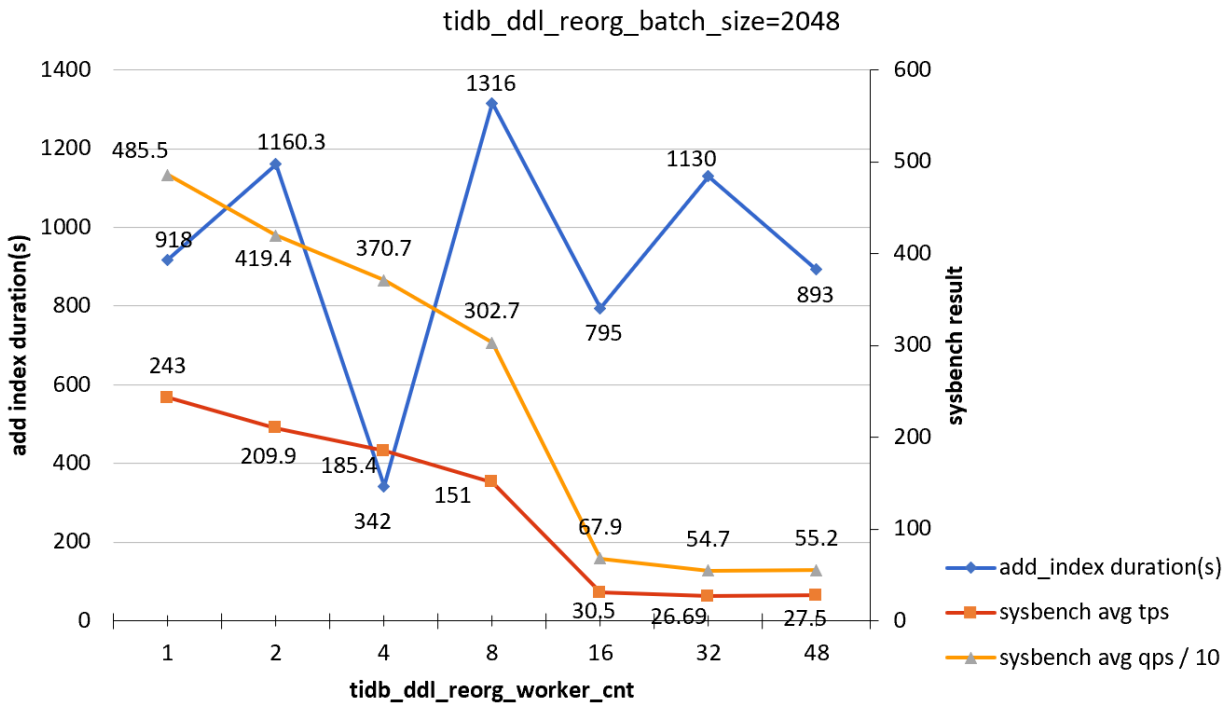


图 15: add-index-load-1-b2048

`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

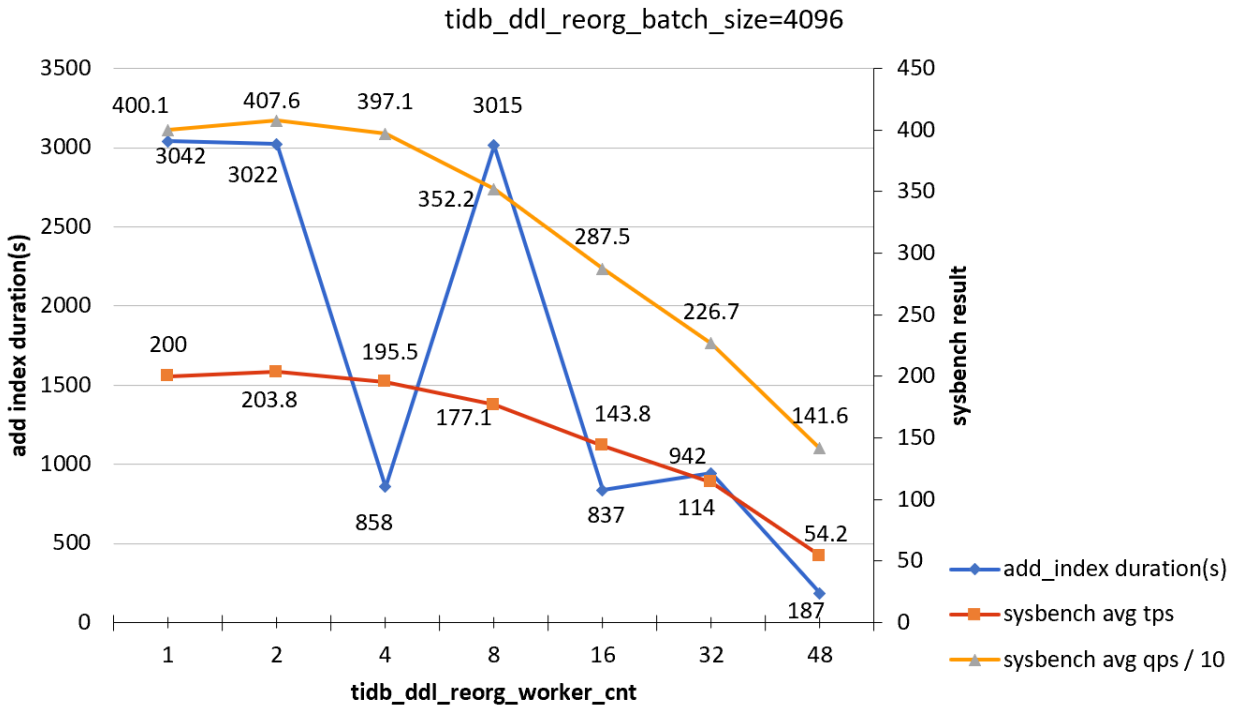


图 16: add-index-load-1-b4096

1.2.5.4.2 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。

1.2.5.5 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。

1.2.5.5.1 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

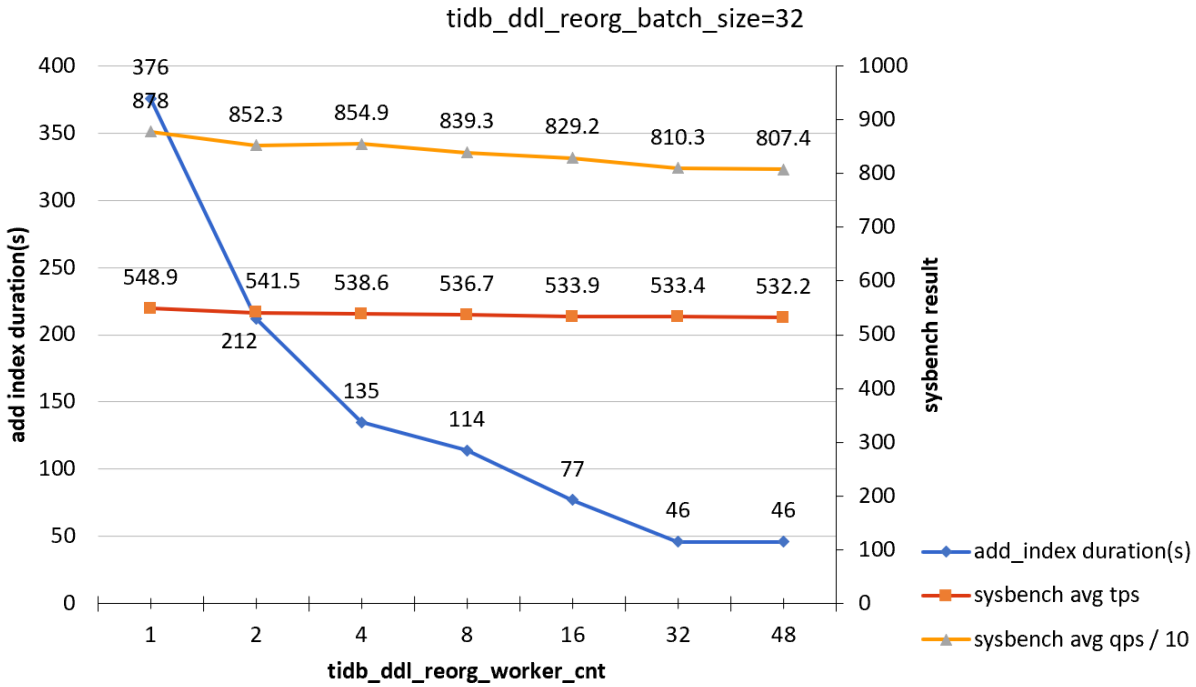


图 17: add-index-load-2-b32

`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

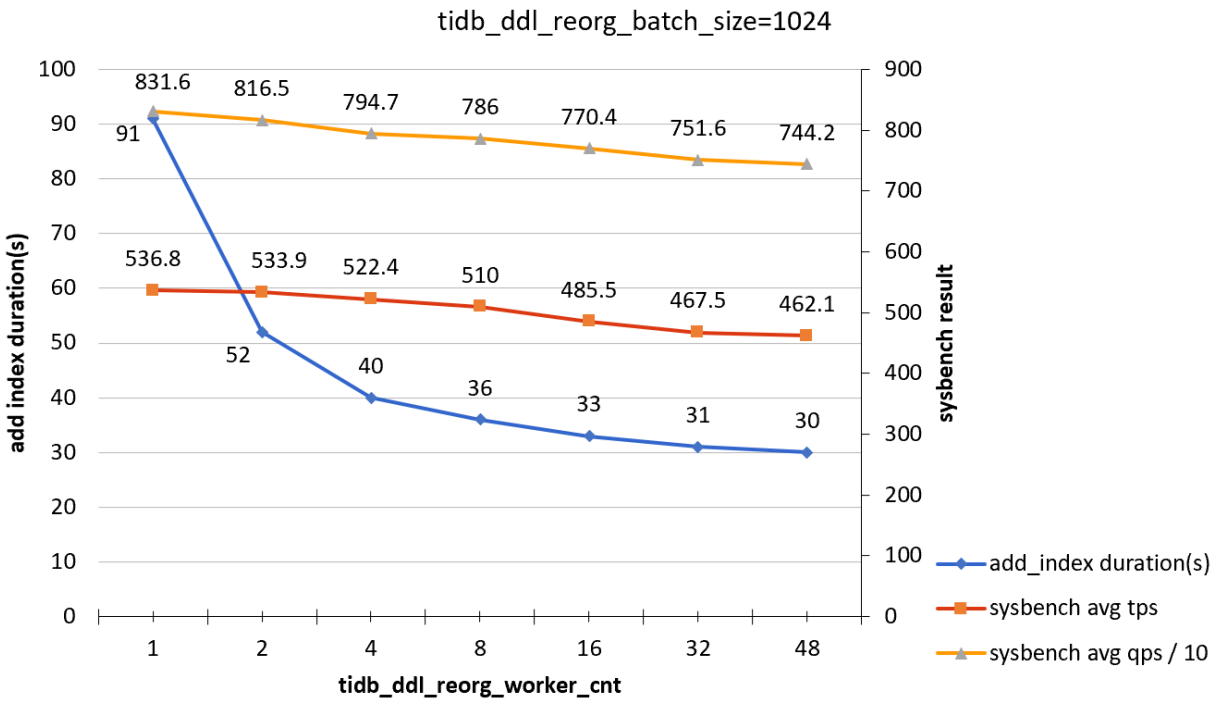


图 18: add-index-load-2-b1024

`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

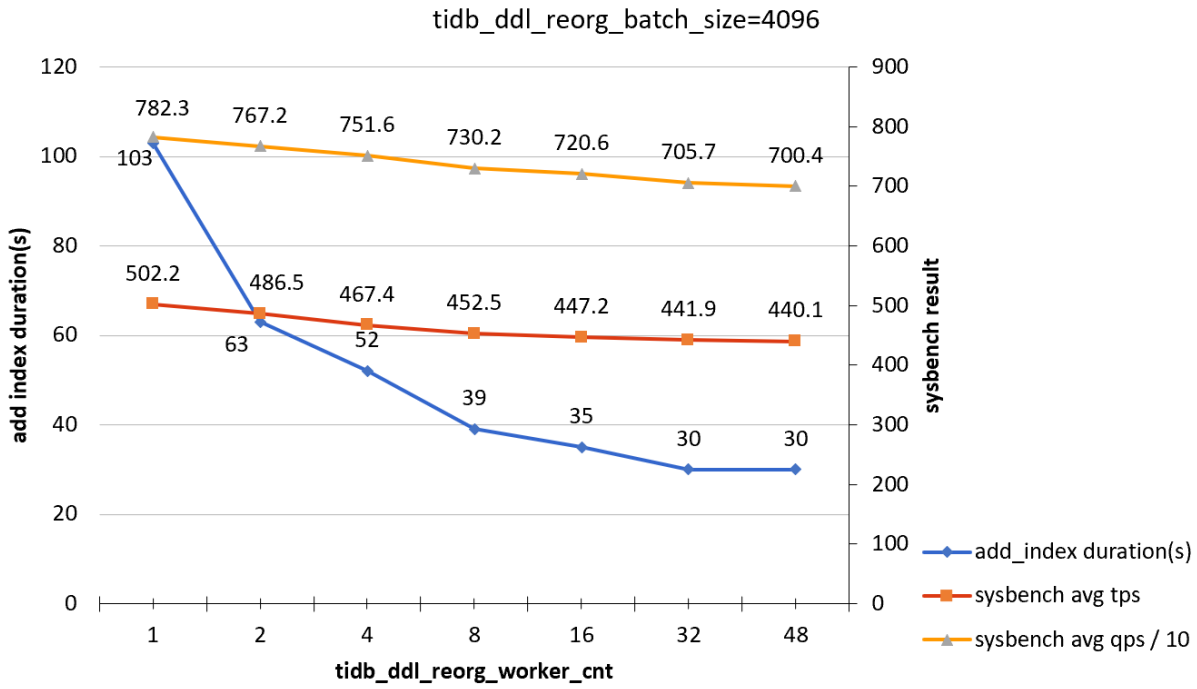


图 19: add-index-load-2-b4096

1.2.5.5.2 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。

1.2.5.6 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。

1.2.5.6.1 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

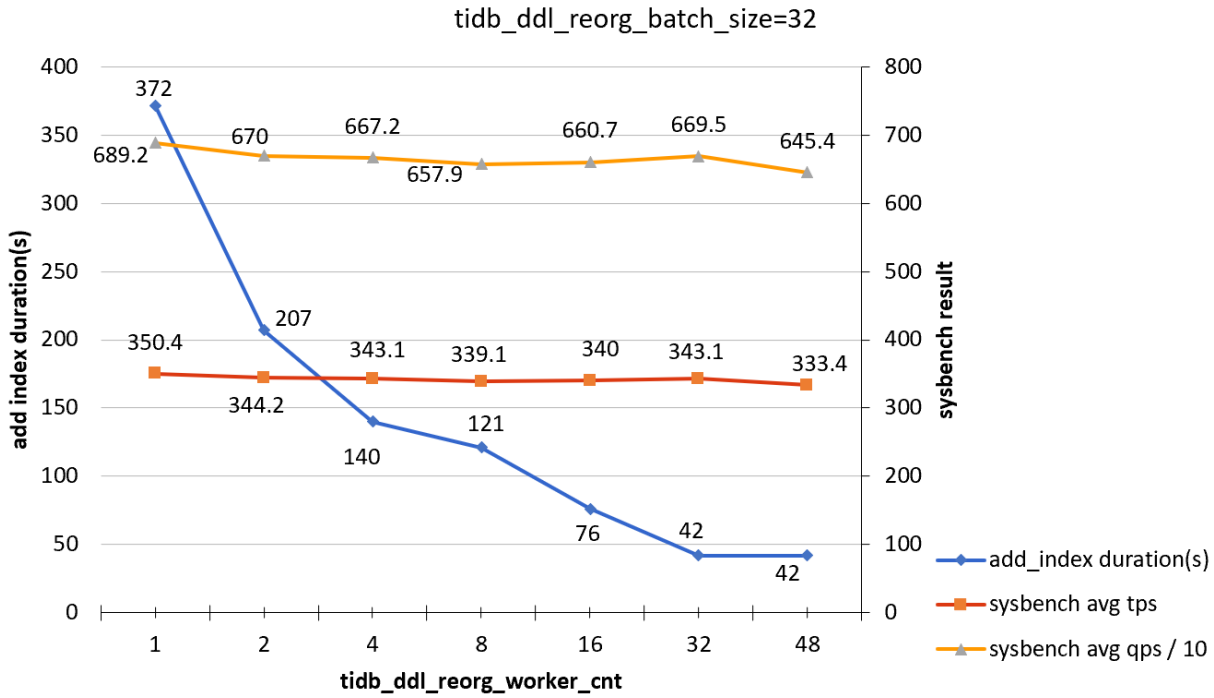


图 20: add-index-load-3-b32

```
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

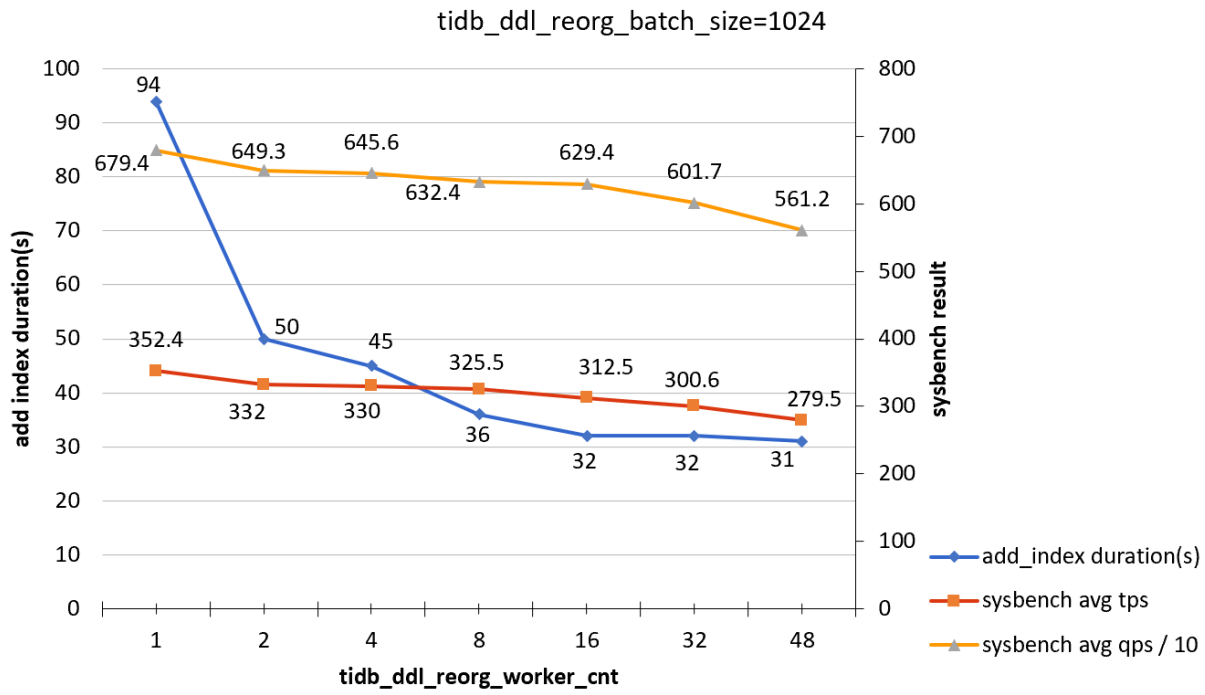


图 21: add-index-load-3-b1024

```
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

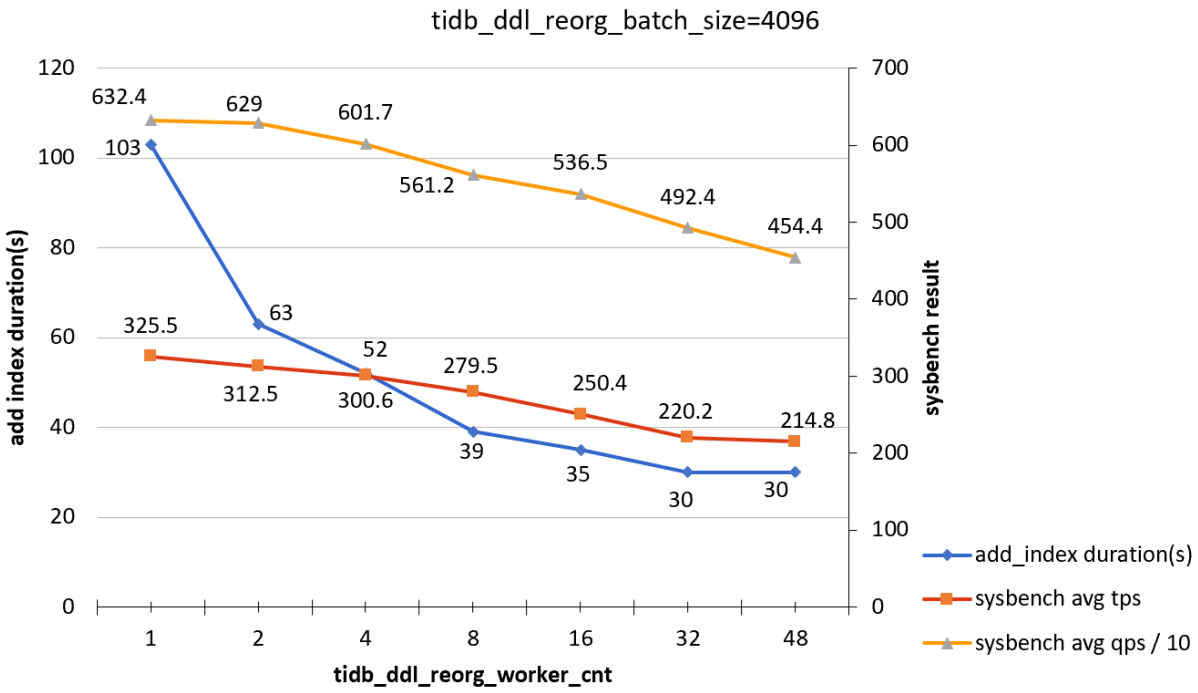


图 22: add-index-load-3-b4096

1.2.5.6.2 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。

1.2.5.7 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 主要概念

2.1 TiDB 整体架构

要深入了解 TiDB 的水平扩展和高可用特点，首先需要了解 TiDB 的整体架构。TiDB 集群主要包括三个核心组件：TiDB Server，PD Server 和 TiKV Server。此外，还有用于解决用户复杂 OLAP 需求的 TiSpark 组件和简化云上部署管理的 TiDB Operator 组件。

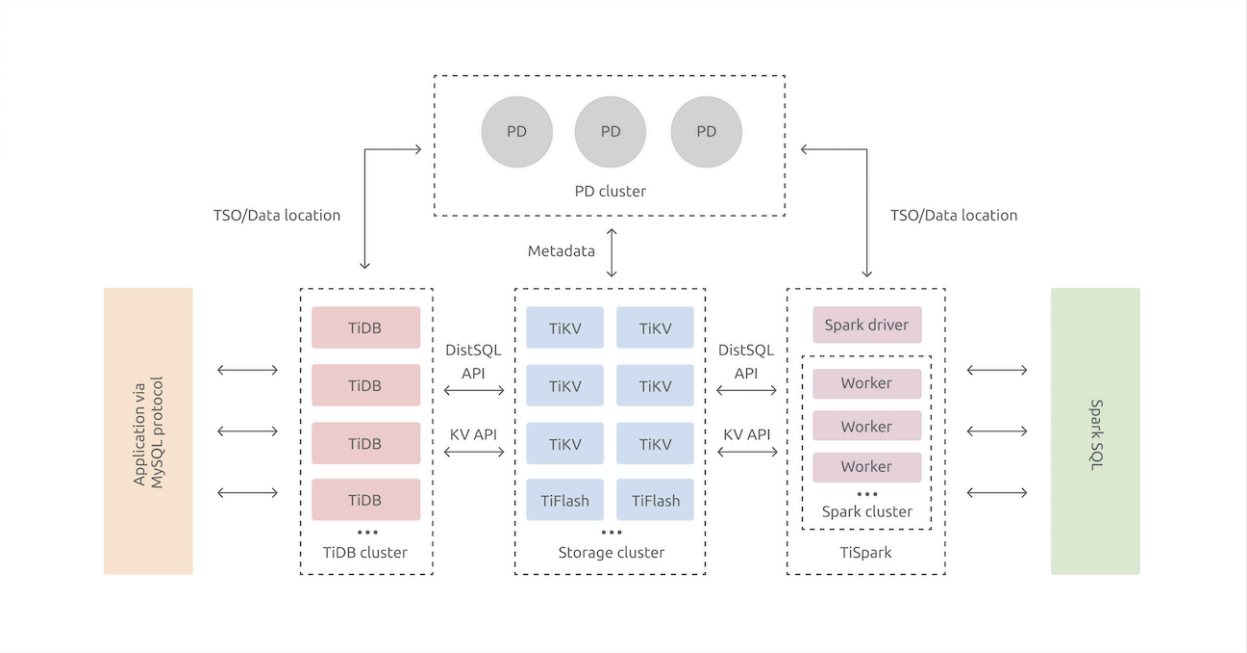


图 23: TiDB Architecture

2.1.1 TiDB Server

TiDB Server 负责接收 SQL 请求，处理 SQL 相关的逻辑，并通过 PD 找到存储计算所需数据的 TiKV 地址，与 TiKV 交互获取数据，最终返回结果。TiDB Server 是无状态的，其本身并不存储数据，只负责计算，可以无限水平扩展，可以通过负载均衡组件（如 LVS、HAProxy 或 F5）对外提供统一的接入地址。

2.1.2 PD Server

Placement Driver (简称 PD) 是整个集群的管理模块，其主要工作有三个：一是存储集群的元信息（某个 Key 存储在哪个 TiKV 节点）；二是对 TiKV 集群进行调度和负载均衡（如数据的迁移、Raft group leader 的迁移等）；三是分配全局唯一且递增的事务 ID。

PD 通过 Raft 协议保证数据的安全性。Raft 的 leader server 负责处理所有操作，其余的 PD server 仅用于保证高可用。建议部署奇数个 PD 节点。

2.1.3 TiKV Server

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 使用 Raft 协议做复制，保持数据的一致性和容灾。副本以 Region 为单位进行管理，不同节点上的多个 Region 构成一个 Raft Group，互为副本。数据在多个 TiKV 之间的负载均衡由 PD 调度，这里也是以 Region 为单位进行调度。

2.1.4 TiSpark

TiSpark 作为 TiDB 中解决用户复杂 OLAP 需求的主要组件，将 Spark SQL 直接运行在 TiDB 存储层上，同时融合 TiKV 分布式集群的优势，并融入大数据社区生态。至此，TiDB 可以通过一套系统，同时支持 OLTP 与 OLAP，免除用户数据同步的烦恼。

2.1.5 TiDB Operator

TiDB Operator 提供在主流云基础设施（Kubernetes）上部署管理 TiDB 集群的能力。它结合云原生社区的容器编排最佳实践与 TiDB 的专业运维知识，集成一键部署、多集群混部、自动运维、故障自愈等能力，极大地降低了用户使用和管理 TiDB 的门槛与成本。

2.2 核心特性

2.2.1 TiDB 核心特性

本文详细介绍 TiDB 的两大核心特性：水平扩展与高可用。

2.2.1.1 水平扩展

无限水平扩展是 TiDB 的一大特点，这里说的水平扩展包括两方面：计算能力和存储能力。TiDB Server 负责处理 SQL 请求，随着业务的增长，可以简单的添加 TiDB Server 节点，提高整体的处理能力，提供更高的吞吐。TiKV 负责存储数据，随着数据量的增长，可以部署更多的 TiKV Server 节点解决数据 Scale 的问题。PD 会在 TiKV 节点之间以 Region 为单位做调度，将部分数据迁移到新加的节点上。所以在业务的早期，可以只部署少量的服务实例（推荐至少部署 3 个 TiKV，3 个 PD，2 个 TiDB），随着业务量的增长，按照需求添加 TiKV 或者 TiDB 实例。

2.2.1.2 高可用

高可用是 TiDB 的另一大特点，TiDB/TiKV/PD 这三个组件都能容忍部分实例失效，不影响整个集群的可用性。下面分别说明这三个组件的可用性、单个实例失效后的后果以及如何恢复。

- TiDB

  TiDB 是无状态的，推荐至少部署两个实例，前端通过负载均衡组件对外提供服务。当单个实例失效时，会影响正在这个实例上进行的 Session，从应用的角度看，会出现单次请求失败的情况，重新连接后即可继续获得服务。单个实例失效后，可以重启这个实例或者部署一个新的实例。

- PD

  PD 是一个集群，通过 Raft 协议保持数据的一致性，单个实例失效时，如果这个实例不是 Raft 的 leader，那么服务完全不受影响；如果这个实例是 Raft 的 leader，会重新选出新的 Raft leader，自动恢复服务。PD 在选举的过程中无法对外提供服务，这个时间大约是 3 秒钟。推荐至少部署三个 PD 实例，单个实例失效后，重启这个实例或者添加新的实例。

- TiKV

  TiKV 是一个集群，通过 Raft 协议保持数据的一致性（副本数量可配置，默认保存三副本），并通过 PD 做负载均衡调度。单个节点失效时，会影响这个节点上存储的所有 Region。对于 Region 中的 Leader 节点，会中断服务，等待重新选举；对于 Region 中的 Follower 节点，不会影响服务。当某个 TiKV 节点失效，并且在一段时间内（默认 30 分钟）无法恢复，PD 会将其上的数据迁移到其他的 TiKV 节点上。

3 操作指南

3.1 快速上手

3.1.1 快速创建 TiDB 集群

如果想要快速上手体验 TiDB 数据库，参考 TiDB 数据库快速上手指南部署最新稳定版的 TiDB。该指南中的部署方式仅适用于快速上手体验，不适用于生产环境。

3.1.2 TiDB 中的基本 SQL 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。

本文介绍 CRUD 操作等基本的 SQL 语句。完整的 SQL 语句列表，参见 TiDB SQL 语法详解。

3.1.2.1 创建、查看和删除数据库

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;
```

使用 SHOW DATABASES 语句查看数据库：

```
SHOW DATABASES;
```

使用 DROP DATABASE 语句删除数据库，例如：

```
DROP DATABASE samp_db;
```

3.1.2.2 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：

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

如果表已存在，添加 IF NOT EXISTS 可防止发生错误：

```
CREATE TABLE IF NOT EXISTS person (
      number INT(11),
      name VARCHAR(255),
      birthday DATE
);
```

使用 SHOW CREATE 语句查看建表语句。例如：

```
SHOW CREATE table person;
```

使用 SHOW FULL COLUMNS 语句查看表的列。例如：

```
SHOW FULL COLUMNS FROM person;
```

使用 DROP TABLE 语句删除表。例如：

```
DROP TABLE person;
```

或者

```
DROP TABLE IF EXISTS person;
```

使用 SHOW TABLES 语句查看数据库中的所有表。例如：

```
SHOW TABLES FROM samp_db;
```

3.1.2.3 创建、查看和删除索引

对于值不唯一的列，可使用 CREATE INDEX 或 ALTER TABLE 语句。例如：

```
CREATE INDEX person_num ON person (number);
```

或者

```
ALTER TABLE person ADD INDEX person_num (number);
```

对于值唯一的列，可以创建唯一索引。例如：

```
CREATE UNIQUE INDEX person_num ON person (number);
```

或者

```
ALTER TABLE person ADD UNIQUE person_num (number);
```

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_num ON person;
```

```
ALTER TABLE person DROP INDEX person_num;
```

3.1.2.4 增删改查数据

使用 INSERT 语句向表内插入数据。例如：

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

使用 SELECT 语句检索表内数据。例如：

```
SELECT * FROM person;
```

```
+--------+------+------------+
| number | name | birthday   |
+--------+------+------------+
|      1 | tom  | 2017-09-12 |
+--------+------+------------+
```

使用 UPDATE 语句修改表内数据。例如：

```
UPDATE person SET birthday='20171010' WHERE name='tom';
```

```
SELECT * FROM person;
```

```
+--------+------+------------+
| number | name | birthday   |
+--------+------+------------+
|      1 | tom  | 2017-10-10 |
+--------+------+------------+
```

使用 DELETE 语句删除表内数据：

```
DELETE FROM person WHERE number=1;
```

```
SELECT * FROM person;
```

```
Empty set (0.00 sec)
```

3.1.2.5 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';
```

3.1.3 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。

3.1.3.1 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。

3.1.3.2 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。

3.1.3.3 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。

3.1.3.4 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。

3.1.4 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。

3.1.4.1 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。

3.1.4.2 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。

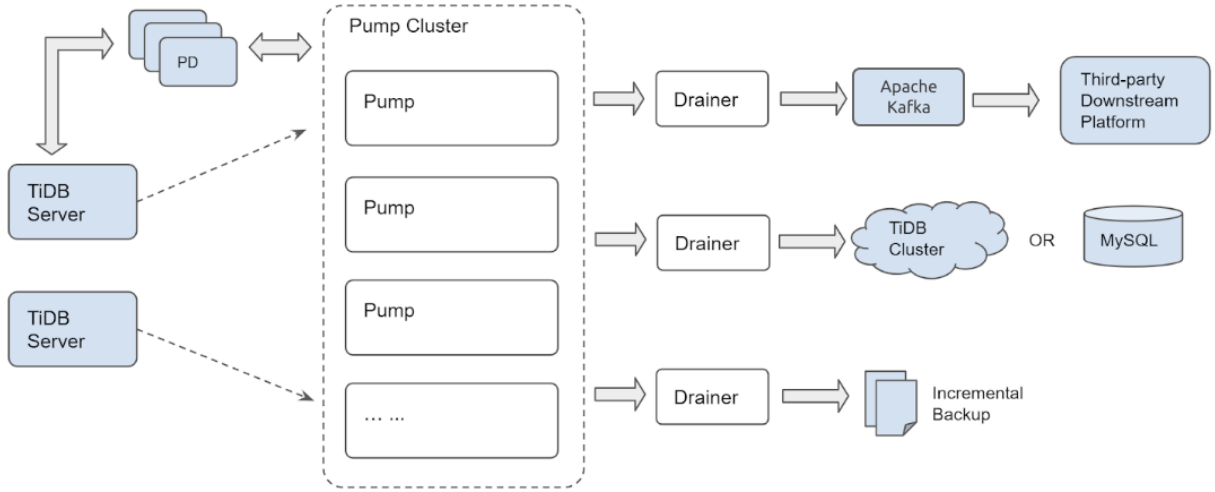


图 24: TiDB Binlog architecture

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。

3.1.4.3 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
```

预期输出：

```
curl -LO https://download.pingcap.org/tidb-v3.0-linux-amd64.tar.gz | tar xzf - &&
cd tidb-v3.0-linux-amd64/
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  279M  100  279M    0     0  4983k      0  0:00:57  0:00:57 --:--:-- 3638k
```

3.1.4.4 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
```

3.1.4.5 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。

3.1.4.6 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
Git Commit Hash: 60965b006877ca7234adaced7890d7b029ed1306
Git Branch: HEAD
UTC Build Time: 2019-06-28 12:14:07
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。

3.1.4.7 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
```

3.1.4.8 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。

3.1.4.9 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。

3.1.5 TiDB Data Migration 教程

3.1.6 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 Mydumper 或 CSV 输出格式的数据源。你可以在以下两种场景下使用 Lightning：

- 迅速导入大量新数据。
- 备份恢复所有数据。

TiDB Lightning 主要包含两个部分:

- `tidb-lightning`（“前端”）：主要完成适配工作，通过读取数据源，在下游 TiDB 集群建表、将数据转换成键/值对 (KV 对) 发送到 `tikv-importer`、检查数据完整性等。
- `tikv-importer`（“后端”）：主要完成将数据导入 TiKV 集群的工作，把 `tidb-lightning` 写入的 KV 对缓存、排序、切分并导入到 TiKV 集群。

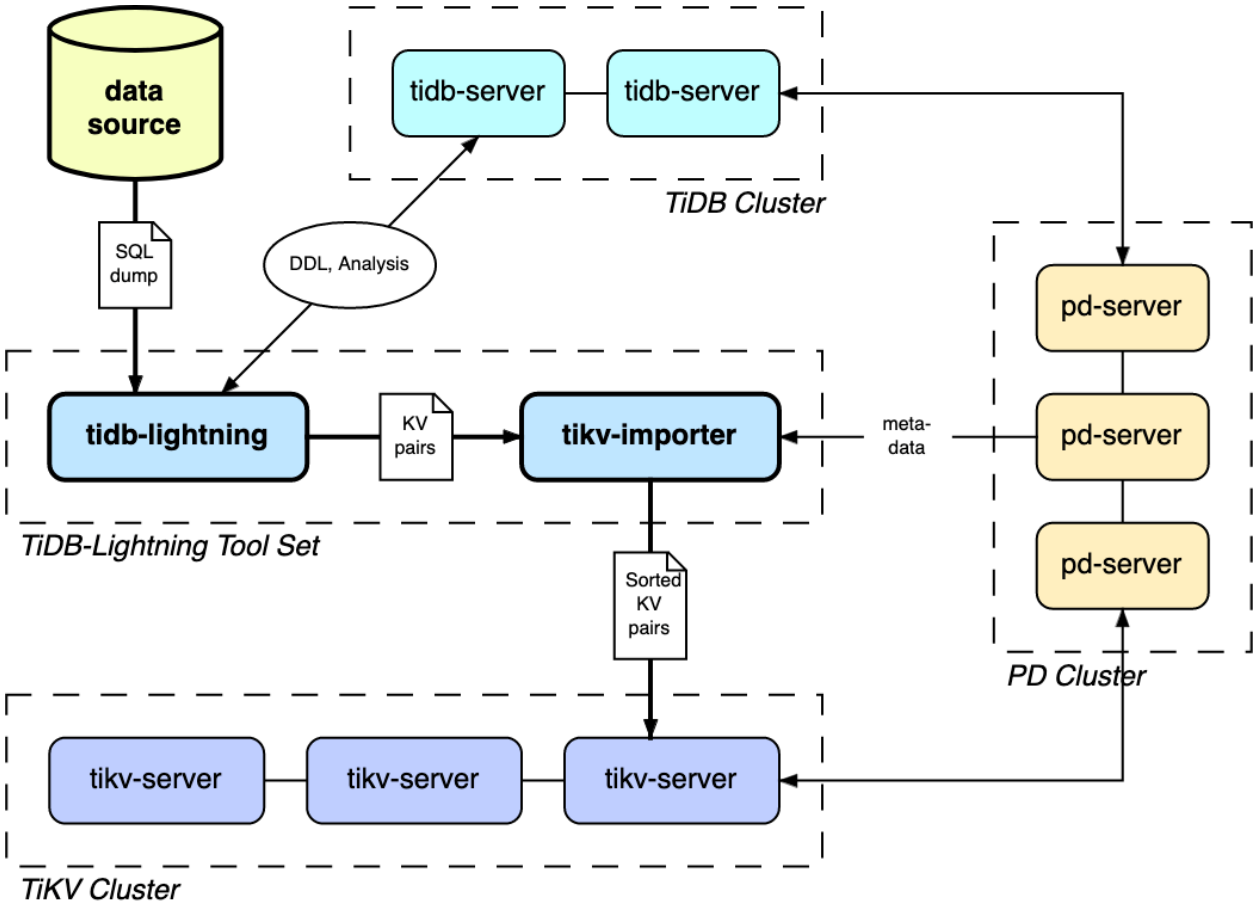


图 25: TiDB Lightning 整体架构

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 4 GB 以上的内存。

> **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。

3.1.6.1 准备全量备份数据

我们使用mydumper 从 MySQL 导出数据，如下：

```
./bin/mydumper -h 127.0.0.1 -P 3306 -u root -t 16 -F 256 -B test -T t1,t2 --skip-tz-utc -o /data/
    ↪ my_database/
```

其中：

- `-B test`：从 `test` 数据库导出。
- `-T t1,t2`：只导出 `t1` 和 `t2` 这两个表。
- `-t 16`：使用 16 个线程导出数据。
- `-F 256`：将每张表切分成多个文件，每个文件大小约为 256 MB。
- `--skip-tz-utc`：添加这个参数则会忽略掉 TiDB 与导数据的机器之间时区设置不一致的情况，禁止自动转换。

这样全量备份数据就导出到了 `/data/my_database` 目录中。

3.1.6.2 部署 TiDB Lightning

3.1.6.2.1 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 3.0.4 版本。部署方法可参考TiDB 快速入门指南。

3.1.6.2.2 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）：

- v3.0.4: tidb-toolkit-v3.0.4-linux-amd64.tar.gz

3.1.6.2.3 第 3 步：启动 `tikv-importer`

1. 将安装包里的 `bin/tikv-importer` 上传至部署 TiDB Lightning 的服务器。
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"

[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 = "192.168.20.10:8287"

[import]
# 存储引擎文档（engine file）的文件夹路径。
import-dir = "/mnt/ssd/data.import/"
```

3. 运行 `tikv-importer`。

```
nohup ./tikv-importer -C tikv-importer.toml > nohup.out &
```

3.1.6.2.4 第 4 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 \
            --importer 172.16.31.10:8287 \
            -d /data/my_database/ \
            --tidb-host 172.16.31.2 \
            --tidb-user root \
            --log-file tidb-lightning.log \
        > nohup.out &
```

3.1.6.2.5 第 5 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 错误排解。

3.1.6.3 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。

3.1.7 TiSpark 快速入门指南

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。

3.1.7.1 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/
```

3.1.7.2 环境准备

3.1.7.2.1 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)
```

3.1.7.2.2 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)
```

3.1.7.3 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

3.2 部署

3.2.1 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。

3.2.1.1 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
CentOS	7.3 及以上
Oracle Enterprise Linux	7.3 及以上
Ubuntu LTS	16.04 及以上

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 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、 XEN 主流虚拟化环境上。

3.2.1.2 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。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：

3.2.1.2.1 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+	无特殊要求	千兆网卡	1（可与 PD 同机器）

组件	CPU	内存	本地存储	网络	实例数量 (最低要求)
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

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 Docker Compose 部署方案单机进行测试。
- 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。

3.2.1.2.2 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 +	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
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。

3.2.1.3 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

组件	默认端口	说明
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
Pump	8250	Pump 通信端口
Drainer	8249	Drainer 通信端口
Prometheus	9090	Prometheus 服务通信端口
Pushgateway	9091	tikv-importer 聚合和上报端口
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
Grafana	8686	grafana_collector 通信端口，用于将 Dashboard 导出为 PDF 格式
Kafka_exporter	9308	Kafka_exporter 通信端口，用于监控 binlog kafka 集群

3.2.1.4 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。

3.2.2 集群部署方式

3.2.2.1 使用 TiDB Ansible 部署 TiDB 集群

3.2.2.1.1 概述

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

> **注意：**
>
> 对于生产环境，须使用 TiDB Ansible 部署 TiDB 集群。如果只是用于测试 TiDB 或体验 TiDB 的特性，建议使用 Docker Compose 在单机上快速部署 TiDB 集群。

3.2.2.1.2 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。
    - 推荐安装 CentOS 7.3 及以上版本 Linux 操作系统，x86_64 架构 (amd64)。
    - 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。

3.2.2.1.3 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
```

- 如果是中控机使用的是 Ubuntu 系统，执行以下命令：

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

3.2.2.1.4 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]-----+
```

3.2.2.1.5 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 3.0 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 `v3.0.2`。
> - 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。

3.2.2.1.6 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 release-2.0、release-2.1、release-3.0、release-3.1 以及最新开发版本兼容 Ansible 2.4 ~ 2.7.11 (2.4 ≤ 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
```

3.2.2.1.7 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。

3.2.2.1.8 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。

3.2.2.1.9 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
  ```

3.2.2.1.10 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)
```

3.2.2.1.11 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。

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
    ↪ 。
[tikv_servers]
TiKV1-1 ansible_host=172.16.10.4 deploy_dir=/data1/deploy tikv_port=20171 labels="host=tikv1"
TiKV1-2 ansible_host=172.16.10.4 deploy_dir=/data2/deploy tikv_port=20172 labels="host=tikv1"
TiKV2-1 ansible_host=172.16.10.5 deploy_dir=/data1/deploy tikv_port=20171 labels="host=tikv2"
TiKV2-2 ansible_host=172.16.10.5 deploy_dir=/data2/deploy tikv_port=20172 labels="host=tikv2"
TiKV3-1 ansible_host=172.16.10.6 deploy_dir=/data1/deploy tikv_port=20171 labels="host=tikv3"
TiKV3-2 ansible_host=172.16.10.6 deploy_dir=/data2/deploy tikv_port=20172 labels="host=tikv3"

#### 部署 3.0 版本的 TiDB 集群时，多实例场景需要额外配置 status 端口，示例如下：
#### 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：
> 推荐配置：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"。

3.2.2.1.12 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
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 主机组的 IP 设置为空。

变量	含义
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
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

3.2.2.1.13 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 到中控机。

```
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
```

3.2.2.1.14 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`

3.2.2.1.15 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
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

组件	端口变量	默认端口	说明
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 用户登录到中控机，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 `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"
```

3.2.2.2 离线 TiDB Ansible 部署方案

3.2.2.2.1 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。

3.2.2.2.2 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。

3.2.2.2.3 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。

3.2.2.2.4 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
```

3.2.2.2.5 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 TiDB Ansible 项目上下载 TiDB Ansible 3.0 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：
>
> - 将 $tag 替换为选定的 TAG 版本的值，例如 v3.0.2。
> - 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。

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。

3.2.2.2.6 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。

3.2.2.2.7 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。

3.2.2.2.8 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。

3.2.2.2.9 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。

3.2.2.2.10 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。

3.2.2.2.11 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。

3.2.2.2.12 测试集群

参考测试集群即可。

3.2.3 跨地域冗余

3.2.3.1 跨数据中心部署方案

作为 NewSQL 数据库，TiDB 兼顾了传统关系型数据库的优秀特性以及 NoSQL 数据库可扩展性，以及跨数据中心（下文简称“中心”）场景下的高可用。本文档旨在介绍跨数据中心部署的不同解决方案。

3.2.3.1.1 三中心部署方案

TiDB, TiKV, PD 分别分布在 3 个不同的中心，这是最常规，可用性最高的方案。

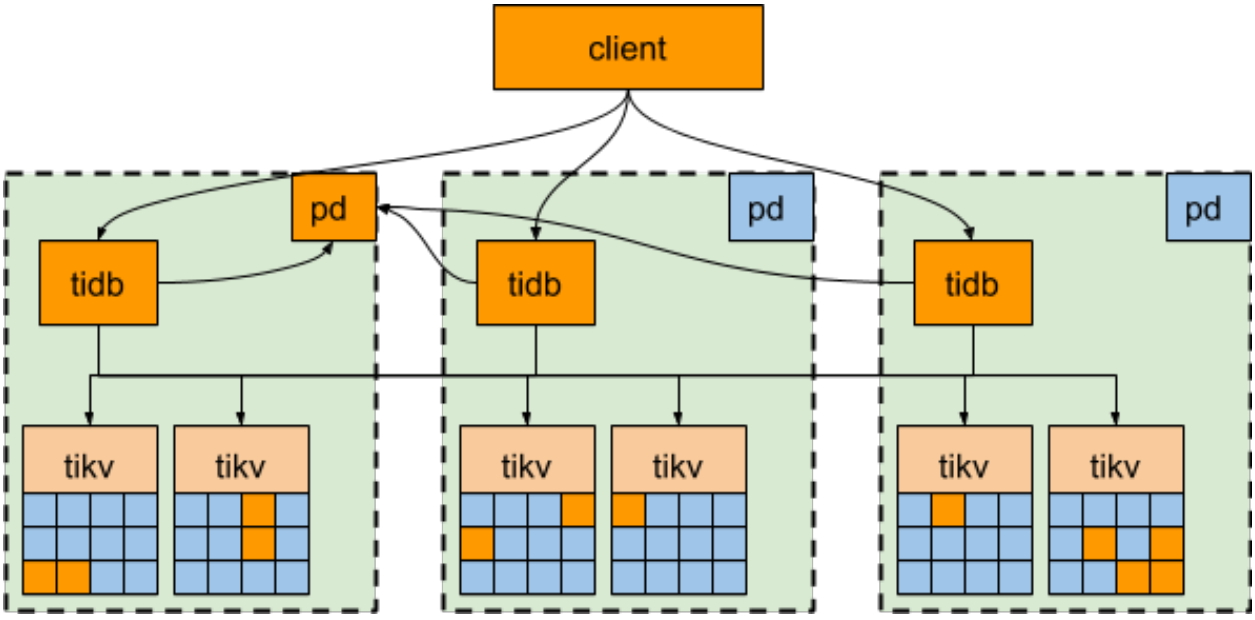


图 26: 三中心部署

优点

所有数据的副本分布在三个数据中心，任何一个数据中心失效后，另外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）恢复服务，并且不会产生数据丢失。

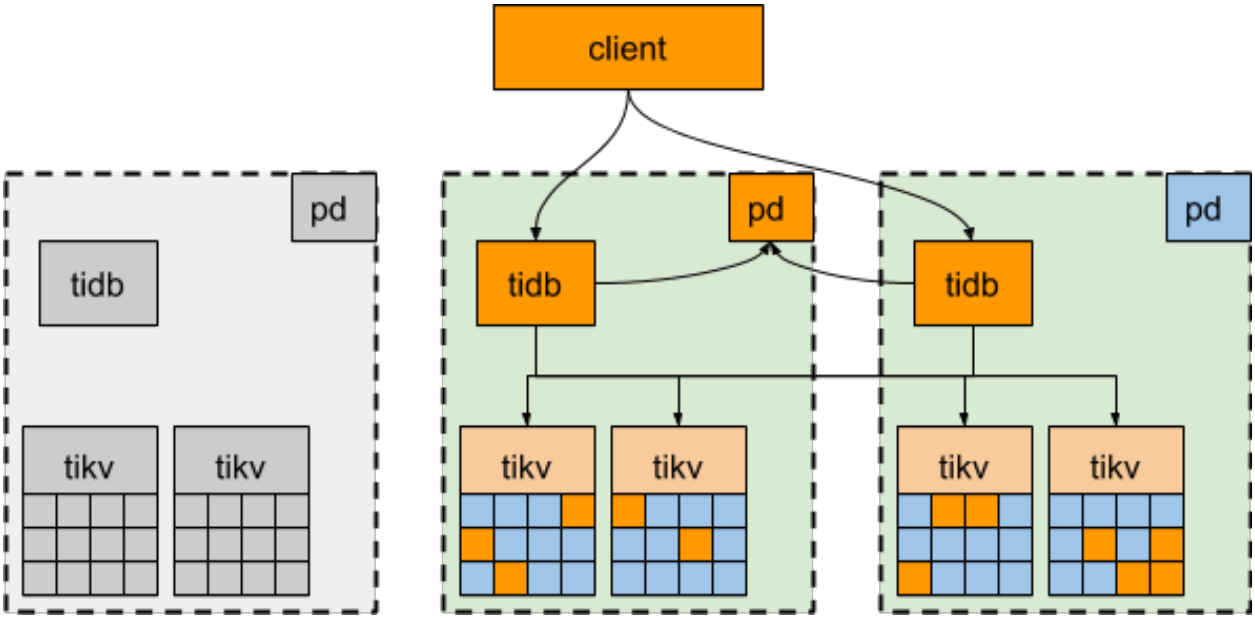


图 27: 三中心部署容灾

缺点

性能受网络延迟影响。

- 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。

读性能优化

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（我们在上文所述的测试中也做了这个优化）。这样一来，不管是从 PD 获取 TSO 还是读取 Region 都不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。

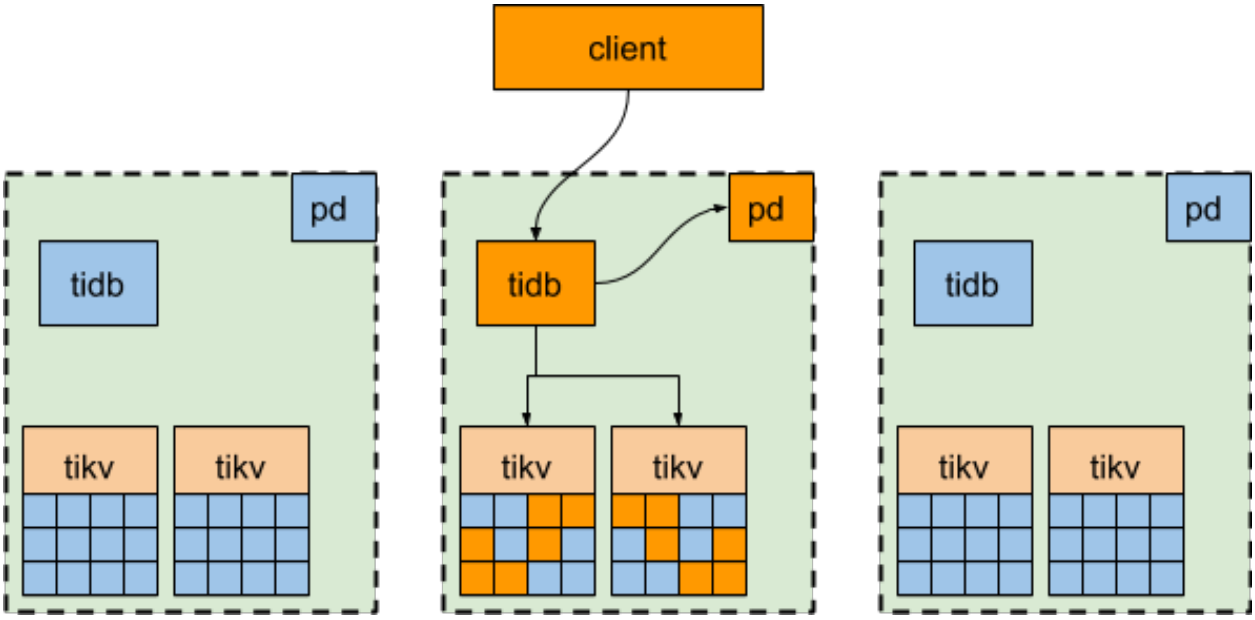


图 28: 三中心部署读性能优化

3.2.3.1.2 两地三中心部署方案

两地三中心的方案与三数据中心类似，算是三机房方案根据业务特点进行的优化，区别是其中有两个数据中心距离很近（通常在同一个城市），网络延迟相对很小。这种场景下，我们可以把业务流量同时派发到同城的两个数据中心，同时控制 Region leader 和 PD leader 也分布在同城的两个数据中心。

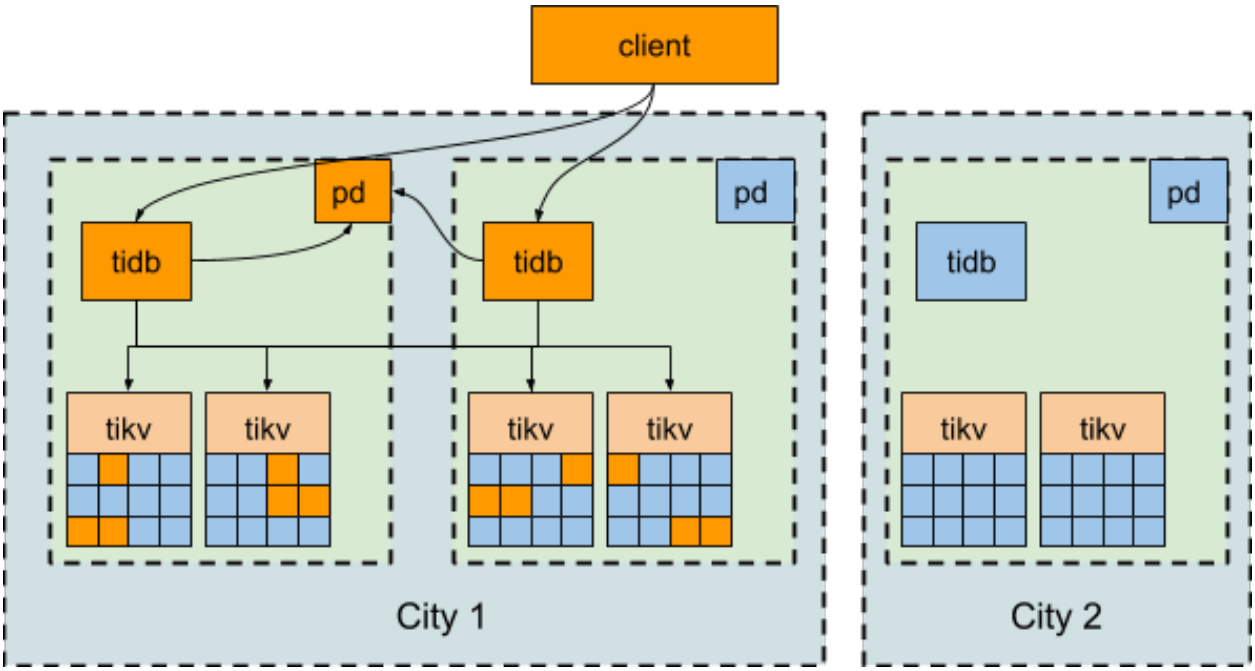


图 29: 两地三中心部署方案

与三数据中心方案相比，两地三中心有以下优势：

- 写入速度更优
- 两中心同时提供服务资源利用率更高
- 依然能保证任何一个数据中心失效后保持可用并且不发生数据丢失

但是，缺陷是如果同城的两个数据中心同时失效（理论上讲要高于异地三数据中心损失 2 个的概率），将会导致不可用以及部分数据丢失。

3.2.3.1.3 两数据中心 + binlog 同步方案

两数据中心 + binlog 同步类似于传统的 MySQL 中 master/slave 方案。两个数据中心分别部署一套完整的 TiDB 集群，我们称之为主集群和从集群。正常情况下所有的请求都在主集群，写入的数据通过 binlog 异步同步至从集群并写入。

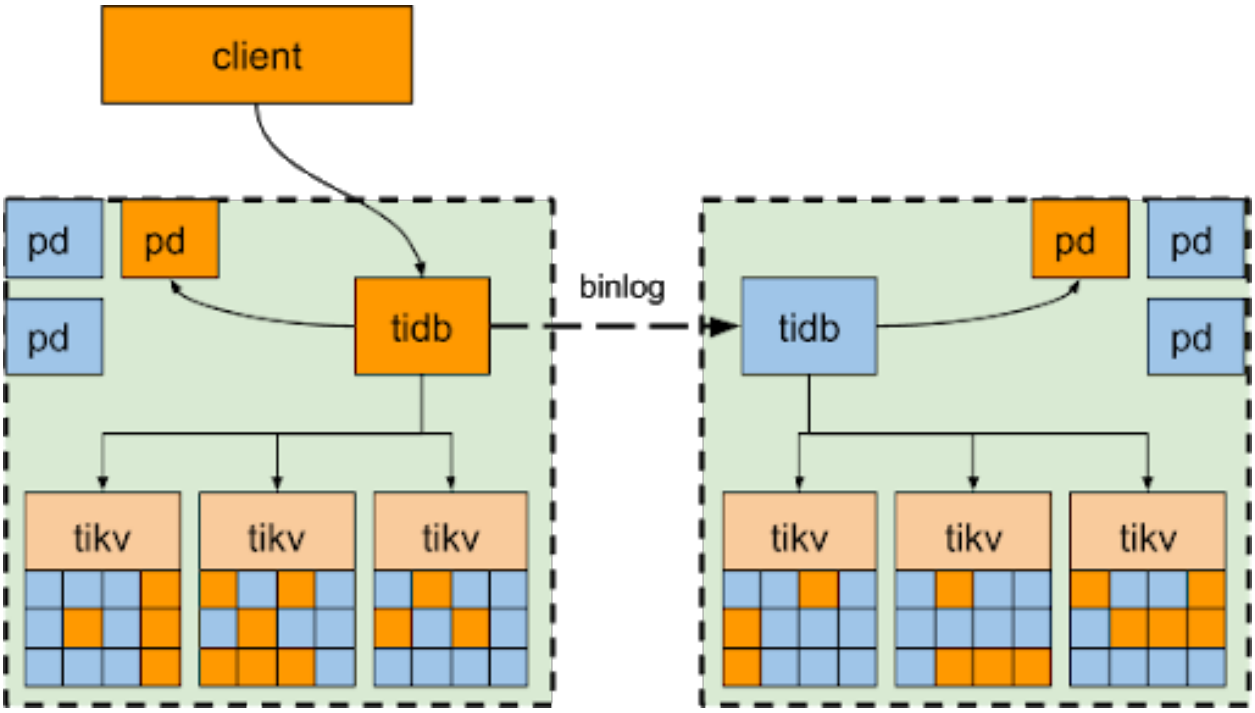


图 30: binlog 同步部署方案

当主集群整个数据中心失效后，业务可以切换至从集群，与 MySQL 类似，这种情况下会有一些数据缺失。对比 MySQL，这个方案的优势是数据中心内的 HA – 少部分节点故障时，通过重新选举 leader 自动恢复服务，不需要人工干预。

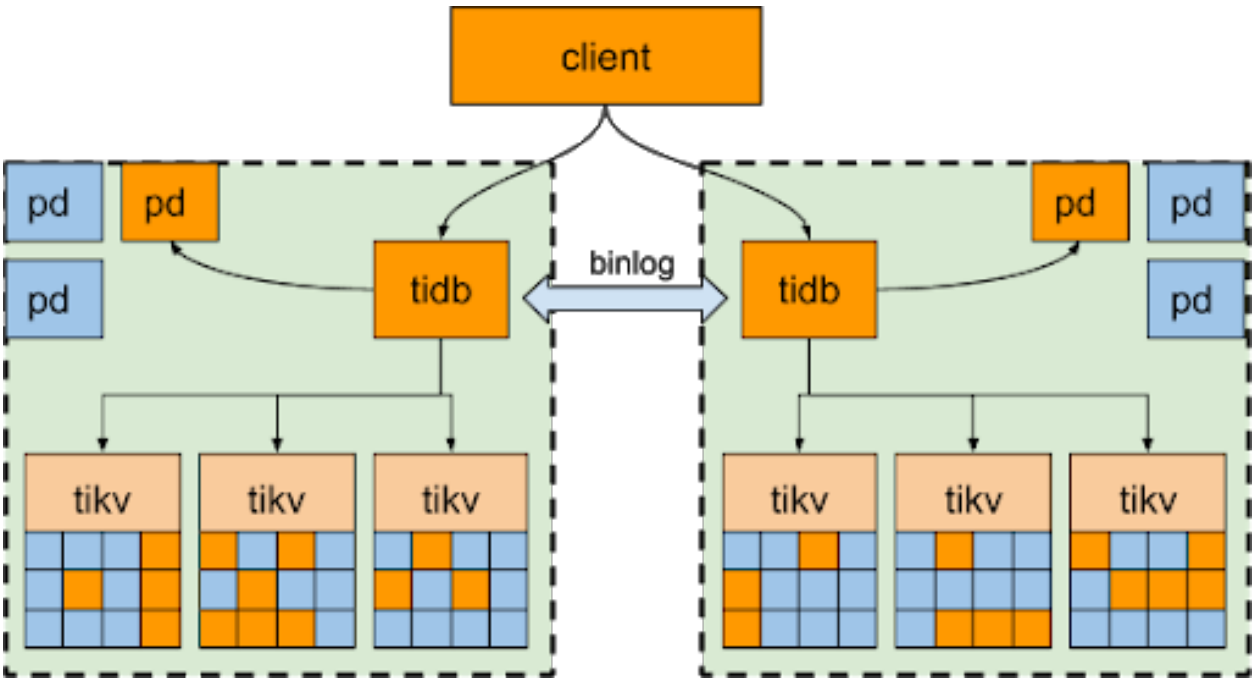


图 31: 两中心 binlog 相互备份方案

另外部分用户采用这种方式做双数据中心多活，两个数据中心各有一个集群，将业务分为两个库，每个库服务一部分数据，每个数据中心的业务只会访问一个库，两个集群之间通过 binlog 将本数据中心业务所涉及的库中的数据变更同步到对端机房，形成环状备份。

> 注意：
>
> 在两数据中心 + binlog 同步部署方案中，数据中心之间只有 binlog 异步复制。在数据中心间的延迟较高的情况下，从集群落后主集群的数据量会增大。当主集群故障后（DR），会造成数据丢失，丢失的数据量受网络延迟等因素影响。

3.2.3.1.4 高可用和容灾分析

对于三数据中心方案和两地三中心方案，我们能得到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。

对于两数据中心 + binlog 同步的方案，主集群内少量节点故障时也能自动恢复服务，不需要人工介入，并能保证数据一致性。当整个主集群故障时，需要人工切换至从集群，并可能发生一些数据丢失，数据丢失的数量取决于同步延迟，和网络条件有关。

3.2.3.2 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TiDB Ansible 部署方案。

3.2.3.2.1 根据集群拓扑配置 labels

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>"
```

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`，可以通过 PD 的配置文件进行配置。

```
[replication]
location-labels = ["zone", "rack", "host"]
```

当 PD 集群初始化完成后，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。

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"]
```

3.2.3.2.2 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。所以如果用户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。同时 TiDB 也提供了诸如 `isolation-level` 这样的强制隔离级别设置，以便更灵活地根据场景来控制对数据的拓扑隔离级别。

3.2.4 使用 Ansible 部署 DM 集群

3.3 配置

3.3.1 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。

3.3.2 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。

3.3.2.1 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	32 << 30
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;
```

3.3.3 Placement Rules 使用文档

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。

3.3.3.1 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：

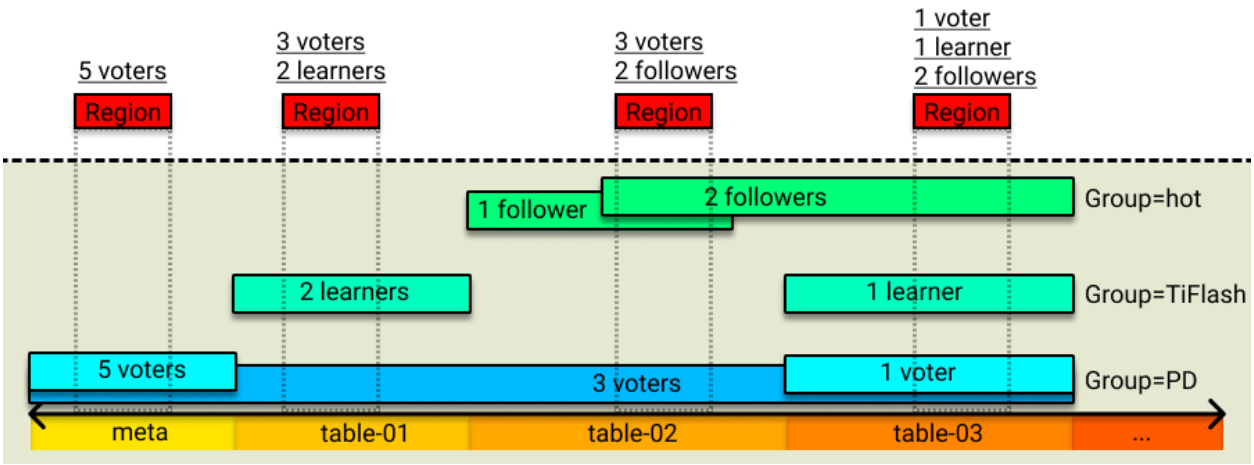


图 32: Placement rules overview

3.3.3.1.1 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。

3.3.3.2 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。

3.3.3.2.1 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。

3.3.3.2.2 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。

3.3.3.2.3 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。

3.3.3.2.4 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。

3.3.3.3 典型场景示例

本部分介绍 Placement Rules 的使用场景示例。

3.3.3.3.1 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"]
}
```

3.3.3.3.2 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_constraint` 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"]
    }
]
```

3.3.3.3.3 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_constraint` 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"]
}
```

3.3.3.3.4 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_constaint 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": "7480000000000000ff2d00000000000000f8",
  "end_key": "7480000000000000ff2e00000000000000f8",
  "role": "follower",
  "count": 2,
  "label_constraints": [
    {"key": "zone", "op": "in", "values": ["bj1", "bj2"]},
    {"key": "disk", "op": "notIn", "values": ["hdd"]}
  ],
  "location_labels": ["host"]
}
```

3.3.4 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。

3.3.4.1 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。

3.3.4.2 使用方法

Store Limit 相关的参数可以通过 pd-ctl 进行设置。

3.3.4.2.1 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。
```

3.3.4.2.2 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。
```

3.3.4.2.3 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。
```

3.3.4.2.4 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。
```

3.4 安全

3.4.1 安全传输层协议 (TLS)

3.4.1.1 使用加密连接

TiDB 服务端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。

3.4.1.1.1 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`。

3.4.1.1.2 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。

3.4.1.1.3 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。

3.4.1.1.4 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 509 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。

3.4.1.1.5 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
...
```

3.4.1.1.6 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。

支持的 TLS 版本

- TLS 1.0
- TLS 1.1
- TLS 1.2
- TLS 1.3

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

3.4.1.2 为 TiDB 组件间开启 TLS 和数据加密存储

本文档介绍 TiDB 集群如何开启 TLS 验证和数据加密存储。

3.4.1.2.1 开启 TLS 验证

本部分介绍 TiDB 集群如何开启 TLS 验证，TLS 验证支持：

- TiDB 组件之间的双向验证，包括 TiDB、TiKV、PD 相互之间，TiDB Control 与 TiDB、TiKV Control 与 TiKV、PD Control 与 PD 的双向认证，以及 TiKV peer 之间、PD peer 之间。一旦开启，所有组件之间均使用验证，不支持只开启某一部分的验证。
- MySQL Client 与 TiDB 之间的客户端对服务器身份的单向验证以及双向验证。

MySQL Client 与 TiDB 之间使用一套证书，TiDB 集群组件之间使用另外一套证书。

TiDB 集群组件间开启 TLS（双向认证）

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，而它们的各种客户端共用 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 `cfssl` 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"
```

此时 TiDB 集群各个组件间已开启双向验证。

> **注意：**
> 若 TiDB 集群各个组件间已开启 TLS，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/client-key.pem"
```

3. 配置校验调用者 Common Name。

   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（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）。推荐在生成证书时通过 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"]
```

4. 重加载证书。

   TiDB、PD 和 TiKV 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。

MySQL 与 TiDB 间开启 TLS

请参考使用加密连接。

3.4.1.2.2 开启数据加密存储

在 TiDB 集群中，用户的数据都存储在 TiKV 中，配置了 TiKV 数据加密存储功能，就代表 TiDB 集群已经加密存储了用户的数据。本部分主要介绍如何配置 TiKV 的加密存储功能。

操作流程

1. 生成 token 文件。

   token 文件存储的是密钥，用于对用户数据进行加密，以及对已加密的数据进行解密。

```
./tikv-ctl random-hex --len 256 > cipher-file-256
```

> 注意：
>
> TiKV 只接受 hex 格式的 token 文件，文件的长度必须是 2^n，并且小于等于 1024。

2. 配置 TiKV。

```
[security]
# Cipher file 的存储路径
cipher-file = "/path/to/cipher-file-256"
```

> 注意：
>
> 若使用Lightning 向集群导入数据，如果目标集群开启了加密功能，Lightning 生成的 sst 文件也必须是加密的格式。

使用限制

目前 TiKV 数据加密存储存在以下限制：

- 对之前没有开启加密存储的集群，不支持开启该功能。

- 已经开启加密功能的集群，不允许关闭加密存储功能。
- 同一集群内部，不允许部分 TiKV 实例开启该功能，部分 TiKV 实例不开启该功能。对于加密存储功能，所有 TiKV 实例要么都开启该功能，要么都不开启该功能。这是由于 TiKV 实例之间会有数据迁移，如果开启了加密存储功能，迁移过程中数据也是加密的。

3.4.2 生成自签名证书

3.4.2.1 概述

本文档提供使用 `cfssl` 生成自签名证书的示例。

假设实例集群拓扑如下：

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

3.4.2.2 下载 cfssl

假设使用 x86_64 Linux 主机：

```
mkdir ~/bin &&
curl -s -L -o ~/bin/cfssl https://pkg.cfssl.org/R1.2/cfssl_linux-amd64 &&
curl -s -L -o ~/bin/cfssljson https://pkg.cfssl.org/R1.2/cfssljson_linux-amd64 &&
chmod +x ~/bin/{cfssl,cfssljson} &&
export PATH=$PATH:~/bin
```

3.4.2.3 初始化证书颁发机构

生成 cfssl 的默认配置，以便于之后修改：

```
mkdir ~/cfssl &&
cd ~/cfssl &&
cfssl print-defaults config > ca-config.json &&
cfssl print-defaults csr > ca-csr.json
```

3.4.2.4 生成证书

3.4.2.4.1 证书介绍

- tidb-server certificate 由 TiDB 使用，为其他组件和客户端验证 TiDB 身份。

- tikv-server certificate 由 TiKV 使用，为其他组件和客户端验证 TiKV 身份。
- pd-server certificate 由 PD 使用，为其他组件和客户端验证 PD 身份。
- client certificate 用于通过 PD、 TiKV、 TiDB 验证客户端。例如 `pd-ctl`，`tikv-ctl`，`pd-recover`。

3.4.2.4.2 配置 CA 选项

根据实际需求修改 `ca-config.json`：

```
{
    "signing": {
        "default": {
            "expiry": "43800h"
        },
        "profiles": {
            "server": {
                "expiry": "43800h",
                "usages": [
                    "signing",
                    "key encipherment",
                    "server auth",
                    "client auth"
                ]
            },
            "client": {
                "expiry": "43800h",
                "usages": [
                    "signing",
                    "key encipherment",
                    "client auth"
                ]
            }
        }
    }
}
```

根据实际需求修改 `ca-csr.json`：

```
{
    "CN": "My own CA",
    "key": {
        "algo": "rsa",
        "size": 2048
    },
    "names": [
        {
            "C": "CN",
            "L": "Beijing",
```

```
            "O": "PingCAP",
            "ST": "Beijing"
        }
    ]
}
```

3.4.2.4.3 生成 CA 证书

```
cfssl gencert -initca ca-csr.json | cfssljson -bare ca -
```

将会生成以下几个文件：

```
ca-key.pem
ca.csr
ca.pem
```

3.4.2.4.4 生成服务器端证书

hostname 中为各组件的 IP 地址，以及 127.0.0.1

```
echo '{"CN":"tidb-server","hosts":[""],"key":{"algo":"rsa","size":2048}}' | cfssl gencert -ca=ca.
    ↪ pem -ca-key=ca-key.pem -config=ca-config.json -profile=server -hostname="
    ↪ 172.16.10.1,172.16.10.2,127.0.0.1" - | cfssljson -bare tidb-server &&

echo '{"CN":"tikv-server","hosts":[""],"key":{"algo":"rsa","size":2048}}' | cfssl gencert -ca=ca.
    ↪ pem -ca-key=ca-key.pem -config=ca-config.json -profile=server -hostname="
    ↪ 172.16.10.4,172.16.10.5,172.16.10.6,127.0.0.1" - | cfssljson -bare tikv-server &&

echo '{"CN":"pd-server","hosts":[""],"key":{"algo":"rsa","size":2048}}' | cfssl gencert -ca=ca.
    ↪ pem -ca-key=ca-key.pem -config=ca-config.json -profile=server -hostname="
    ↪ 172.16.10.1,172.16.10.2,172.16.10.3,127.0.0.1" - | cfssljson -bare pd-server
```

将会生成以下几个文件：

```
tidb-server-key.pem     tikv-server-key.pem     pd-server-key.pem
tidb-server.csr         tikv-server.csr         pd-server.csr
tidb-server.pem         tikv-server.pem         pd-server.pem
```

3.4.2.4.5 生成客户端证书

```
echo '{"CN":"client","hosts":[""],"key":{"algo":"rsa","size":2048}}' | cfssl gencert -ca=ca.pem -
    ↪ ca-key=ca-key.pem -config=ca-config.json -profile=client -hostname="" - | cfssljson -bare
    ↪  client
```

将会生成以下几个文件：

```
client-key.pem
client.csr
client.pem
```

3.5 监控

3.5.1 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。

3.5.1.1 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：

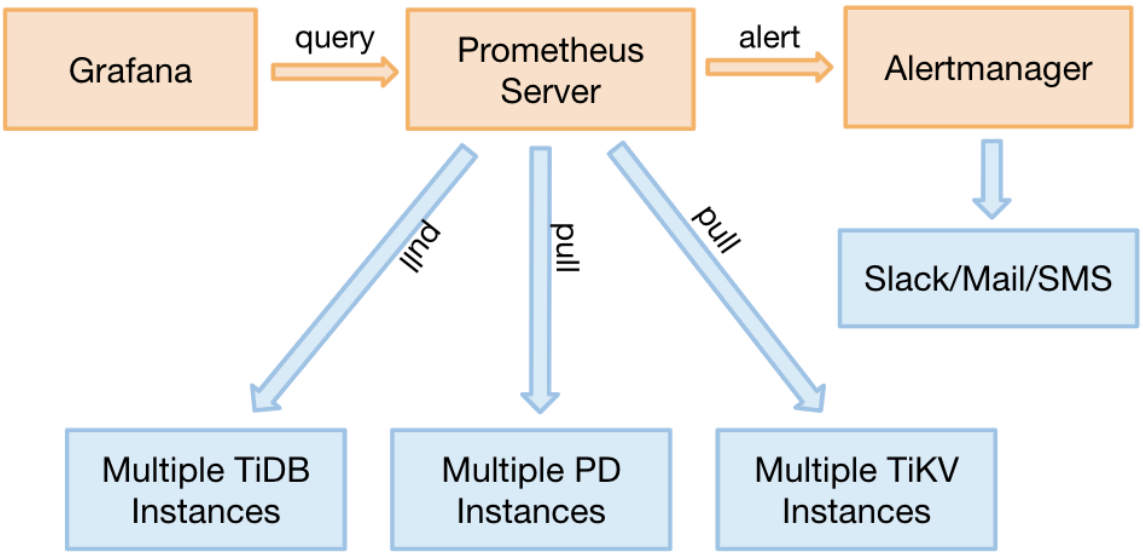


图 33: Prometheus in TiDB

3.5.1.2 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 的各项性能指标。如下图所示：

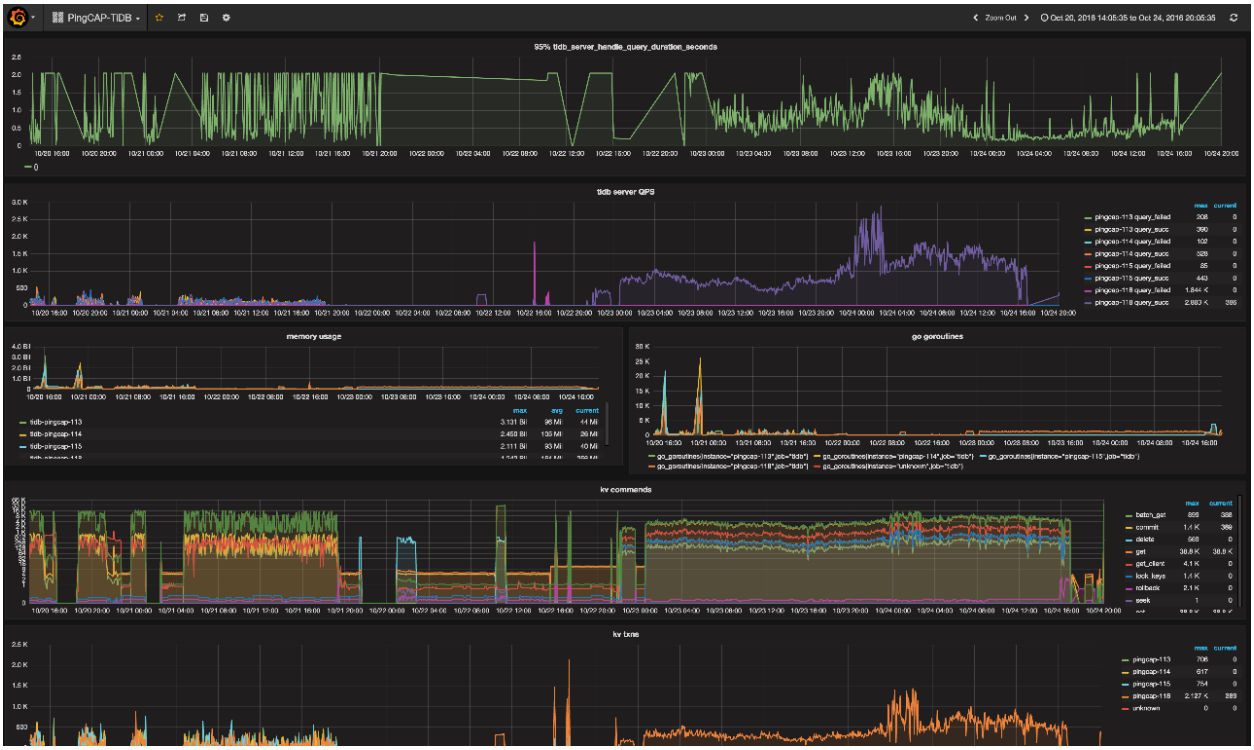


图 34: Grafana in TiDB

3.5.2 TiDB 集群监控

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。

3.5.2.1 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。

3.5.2.1.1 TiDB Server

- TiDB API 地址：`http://${host}:${port}`
- 默认端口：10080
- 各类 `api_name` 详细信息：参见 TiDB API 文档

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-v3.0.0-beta-250-g778c3f4a5",  # TiDB 版本号
    git_hash: "778c3f4a5a716880bcd1d71b257c8165685f0d70"  # TiDB 当前代码的 Git Hash
}
```

3.5.2.1.2 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": "3.0.0-beta",
        "state_name": "Up"
      },
      "status": {
        "capacity": "20 GiB",  # 存储总容量
        "available": "16 GiB",  # 存储剩余容量
        "leader_count": 17,
        "leader_weight": 1,
        "leader_score": 17,
        "leader_size": 17,
        "region_count": 17,
        "region_weight": 1,
        "region_score": 17,
        "region_size": 17,
        "start_ts": "2019-03-21T14:09:32+08:00",  # 启动时间
```

```
        "last_heartbeat_ts": "2019-03-21T14:14:22.961171958+08:00",  # 最后一次心跳的时间
        "uptime": "4m50.961171958s"
      }
    }
  ]
```

3.5.2.2 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。

3.5.2.2.1 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

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
```

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" &
```

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" &
```

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" &
```

3.5.2.2.2 **配置 Grafana**

本小节介绍如何配置 Grafana。

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。

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。

3.5.2.2.3 查看组件 metrics

在顶部菜单中，点击 New dashboard，选择要查看的面板。

图 35: 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

3.5.3 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。

3.5.3.1 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。

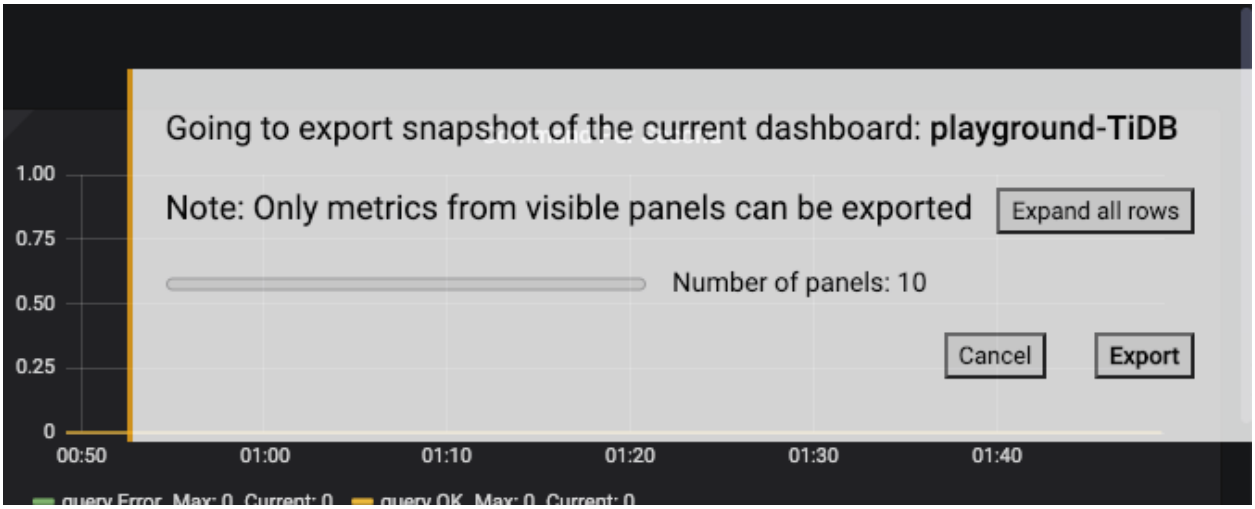


图 36: 运行用户脚本后的 MetricsTool Exporter 截图

- 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。

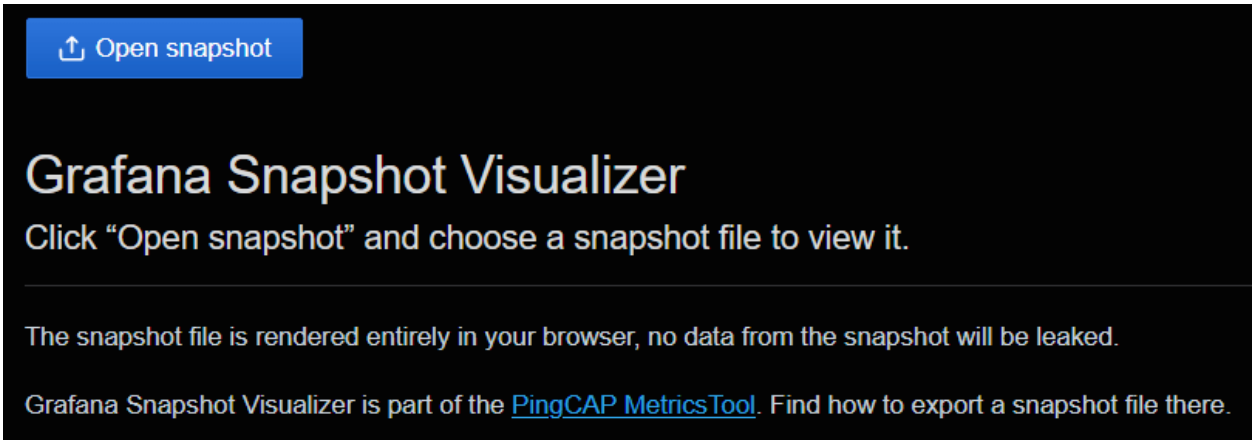


图 37: MetricsTool Visualizer 截图

- 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。

3.5.3.2 FAQs

3.5.3.2.1 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。

3.5.3.2.2 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。

3.5.3.2.3 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。

3.5.3.2.4 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。

3.5.3.2.5 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。

3.5.3.2.6 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。

3.6 迁移

3.6.1 TiDB 工具功能概览

本文档从生态工具的功能出发，介绍部分生态工具的功能以及它们之间的替代关系。

3.6.1.1 在 Kubernetes 上部署运维 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私有部署的 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

3.6.1.2 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

3.6.1.3 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，有两种模式：

- `importer` 模式：以 TiKV-importer 作为后端，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。
- `tidb` 模式：以 TiDB/MySQL 作为后端，这种模式相比 `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：

- 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。

3.6.1.4 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

3.6.1.5 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

3.6.1.6 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。

3.6.2 从 MySQL 迁移

3.6.2.1 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Mydumper 或者Dumpling 文档。

3.6.2.1.1 第 1 步：部署 TiDB Lightning

使用 Lightning 将数据导入 TiDB，Lightning 具体的部署方法见TiDB Lightning 部署。

> 注意：

- 如果选用 Importer-backend 来导入数据，除需要部署 TiDB Lightning 组件外，也需要部署 TiKV Importer 组件。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，只需要部署 TiDB Lightning 组件。导入期间集群可以正常提供服务。
- 二者的具体差别参见TiDB Lightning Backend。

3.6.2.1.2 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 = ""
```

其它配置参考TiDB Lightning 配置。

3.6.2.1.3 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。

3.6.2.2 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。

3.6.3 CSV 支持

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）。

3.6.3.1 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`。

文件扩展名必须为 `*.csv`，即使文件的内容并非逗号分隔。

3.6.3.2 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`。
- 首先在 TiDB 中直接创建空表，然后在 `tidb-lightning.toml` 中设置 `[mydumper] no-schema = true`。

3.6.3.3 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
```

3.6.3.3.1 `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。

3.6.3.3.2 `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。

3.6.3.3.3 `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。

3.6.3.3.4 `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。

3.6.3.3.5 `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。

3.6.3.3.6 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')。

3.6.3.3.7 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。

3.6.3.4 通用配置

3.6.3.4.1 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
```

3.6.3.4.2 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
```

3.6.3.4.3 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|
```

3.6.4 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Mydumper 或者Dumpling 文档。

3.6.4.1 第 1 步：部署 TiDB Lightning

使用 Lightning 将数据导入 TiDB，Lightning 具体的部署方法见TiDB Lightning 部署。

> 注意：
>
> - 如果选用 Importer-backend 来导入数据，除需要部署 TiDB Lightning 组件外，也需要部署 TiKV Importer 组件。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，只需要部署 TiDB Lightning 组件。导入期间集群可以正常提供服务。
> - 二者的具体差别参见TiDB Lightning Backend。

3.6.4.2 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 = ""
```

其它配置参考TiDB Lightning 配置。

3.6.4.3 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。

3.7 运维

3.7.1 TiDB Ansible 常见运维操作

3.7.1.1 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
```

3.7.1.2 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
```

3.7.1.3 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
```

3.7.1.4 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
```

3.7.2 备份与恢复

3.7.2.1 使用 BR 工具（推荐）

3.7.2.1.1 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` 和 `mydumper/loader`，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。

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。

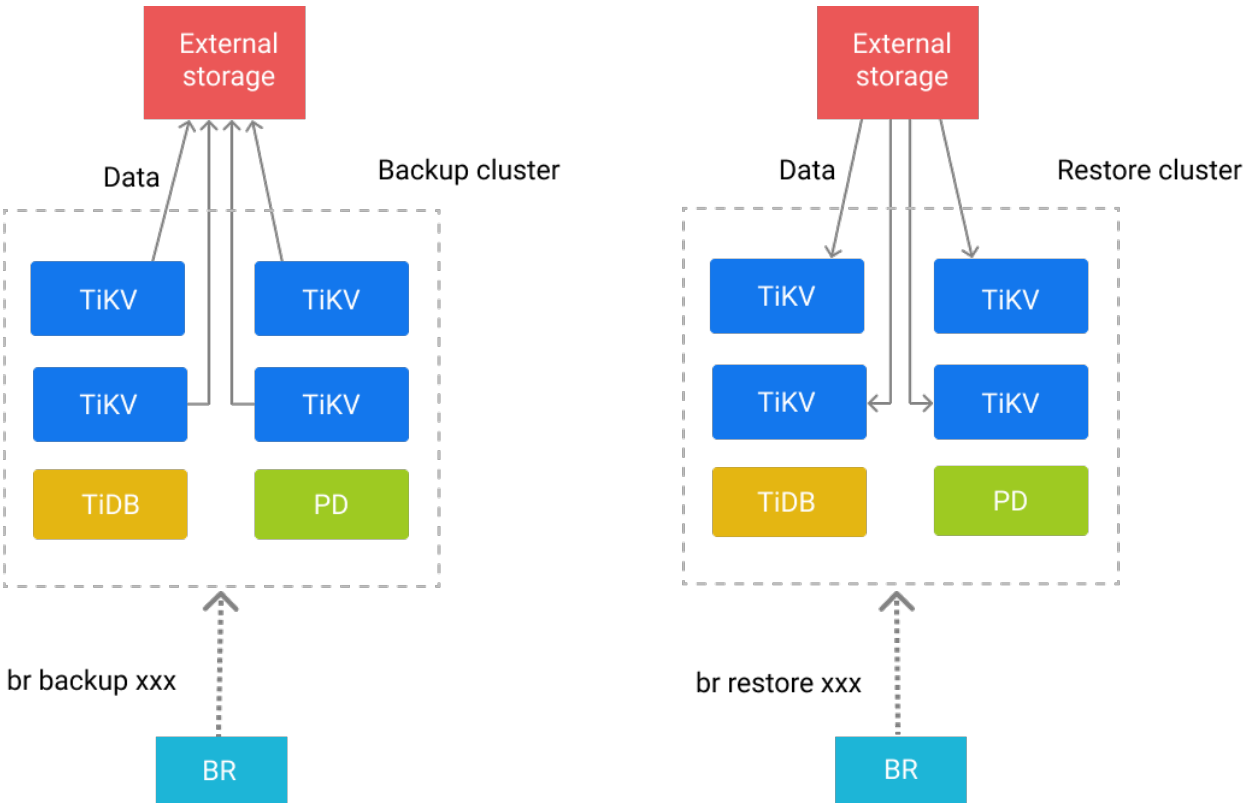


图 38: br-arch

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

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`）

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。

使用限制

下面是使用 BR 进行备份恢复的几条限制：

- BR 只支持在 TiDB v3.1 及以上版本使用。
- BR 恢复到 Drainer 的上游集群时，恢复数据无法由 Drainer 同步到下游。
- 在 v3.1 中，只支持在全新的集群上执行恢复操作。

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

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。

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

3.7.2.1.2 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。

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。

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。

如果备份时间可能超过设定的tikv_gc_life_time（默认 10m0s，即表示 10 分钟），则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'";
```

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 120 \
    --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 120 \
    --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 120 \
    --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 120 \
    --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 120 \
    --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。

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 \
    --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 \
    -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 \
    --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。

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" \
    --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" \
    --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。

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" \
    --storage "local:///tmp/backup" \
    --log-file restorefull.log
```

使用表库过滤功能恢复数据

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。

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}" \
    --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。

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 \
    --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" \
    --pd $PD_ADDR \
    --online
```

3.7.2.1.3 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

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。

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TiDB Ansible 部署 TiDB 集群，再下载TiDB Toolkit 获取 BR 应用。

集群版本

- TiDB: v3.1.2
- TiKV: v3.1.2
- PD: v3.1.2
- BR: v3.1.2

> 注意：
>
> v3.1.2 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

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。你只需要将 backupTS 注册到 PD 的 safePoint，保证 safePoint 在备份期间不会向前移动，即可避免手动设置 GC。

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 节点和 TiKV 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。

部署拓扑

部署拓扑如下图所示：

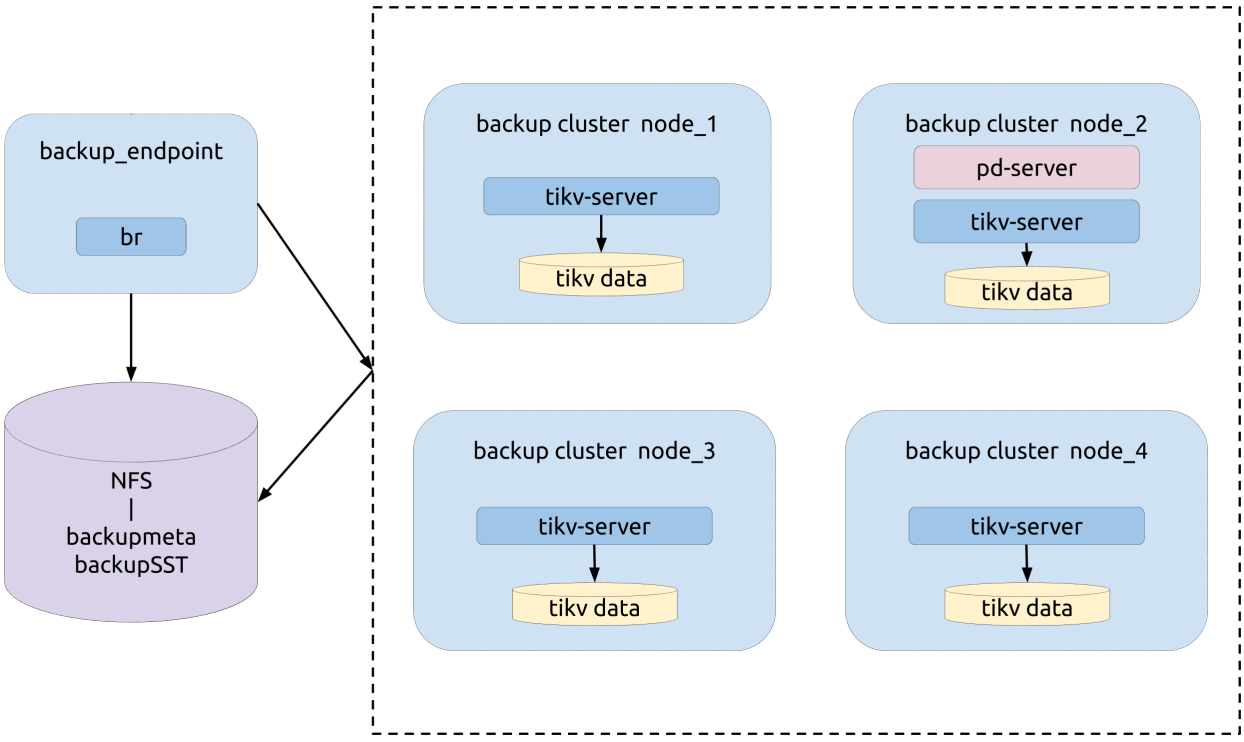


图 39: img

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 40: 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。

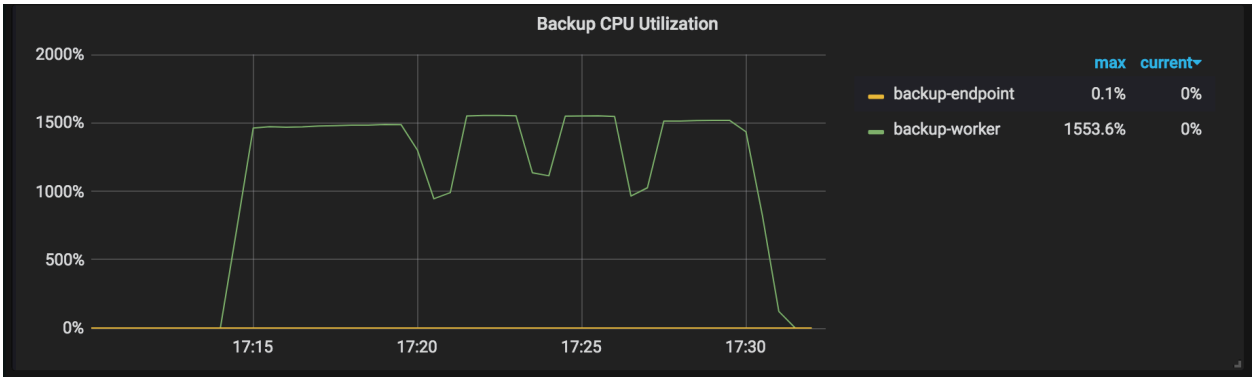


图 41: img

IO Utilization：参与备份的 TiKV 节点的 I/O 使用率。

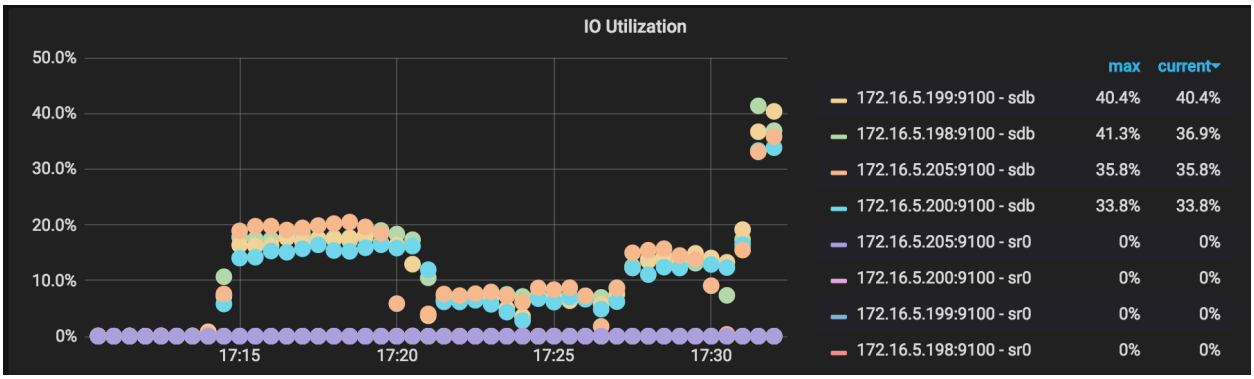


图 42: img

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。

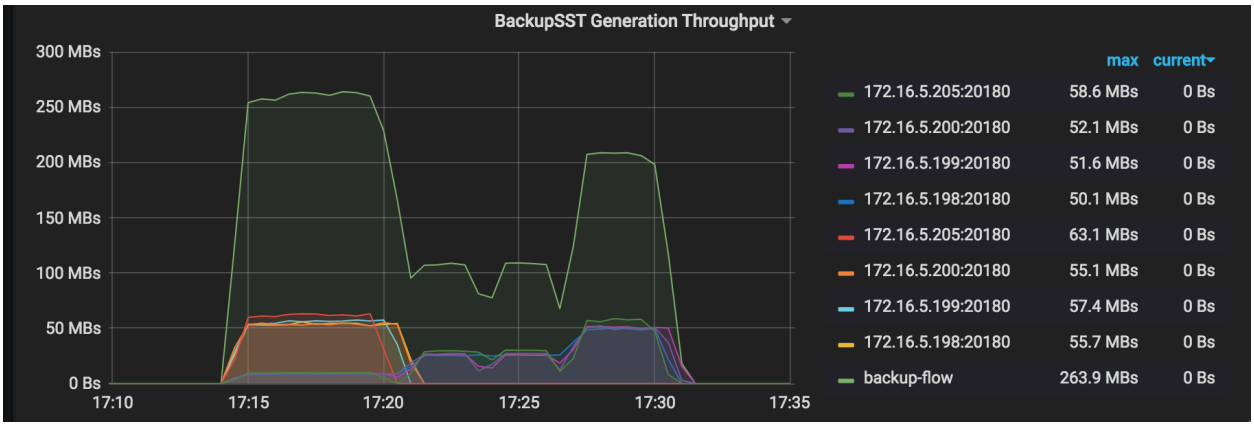


图 43: img

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）。

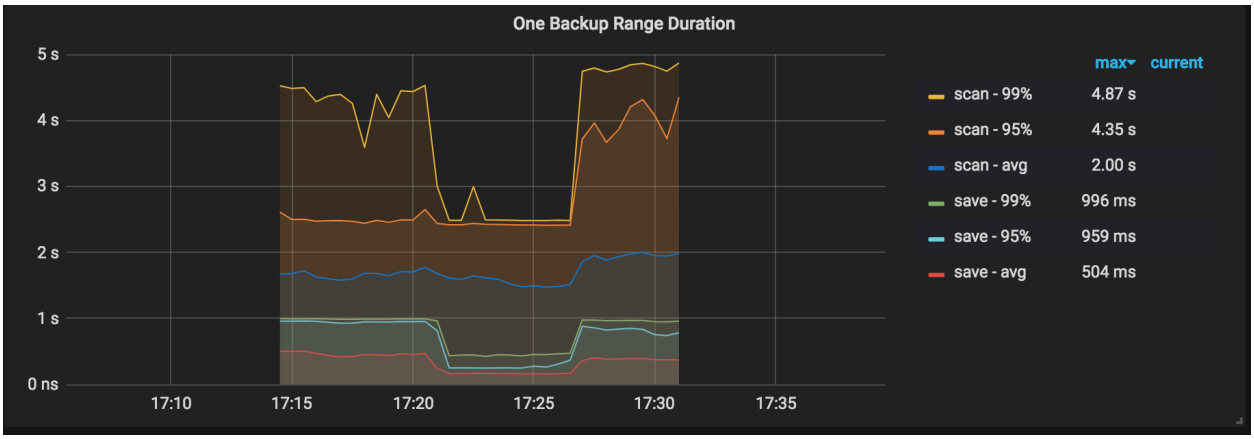


图 44: img

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。

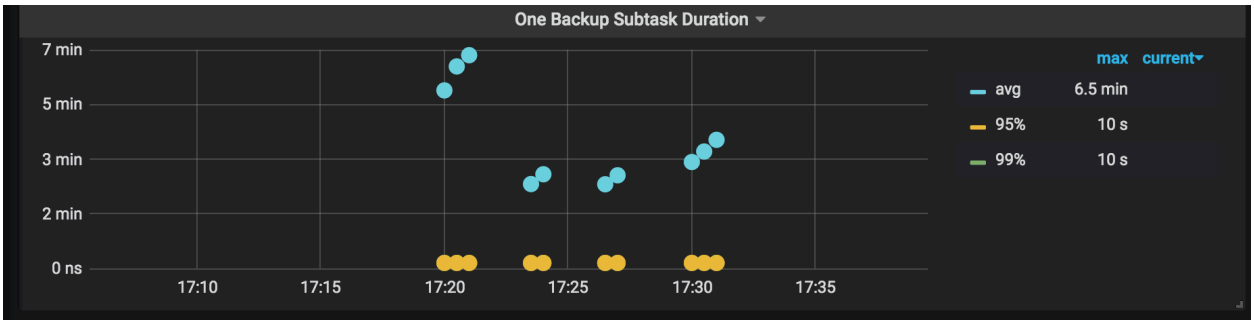


图 45: img

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。

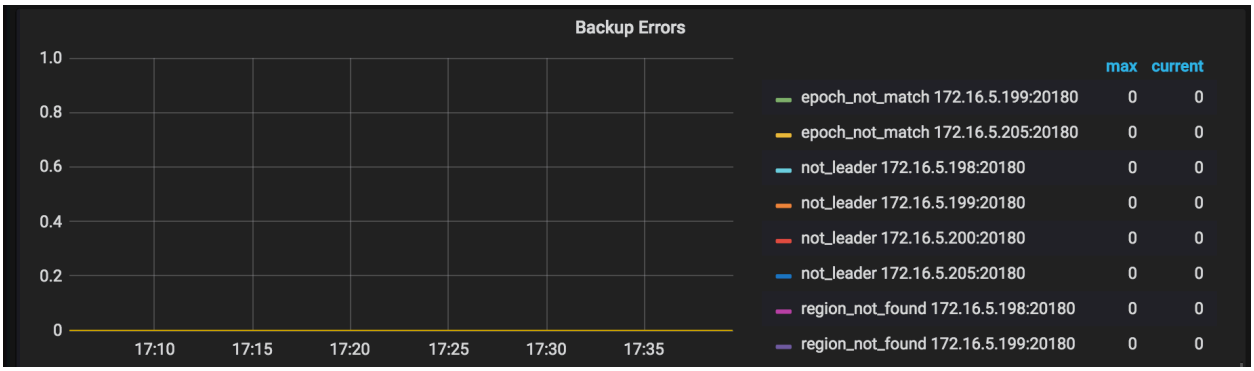


图 46: img

Checksum Request Duration：对备份集群执行 admin checksum 的耗时统计。

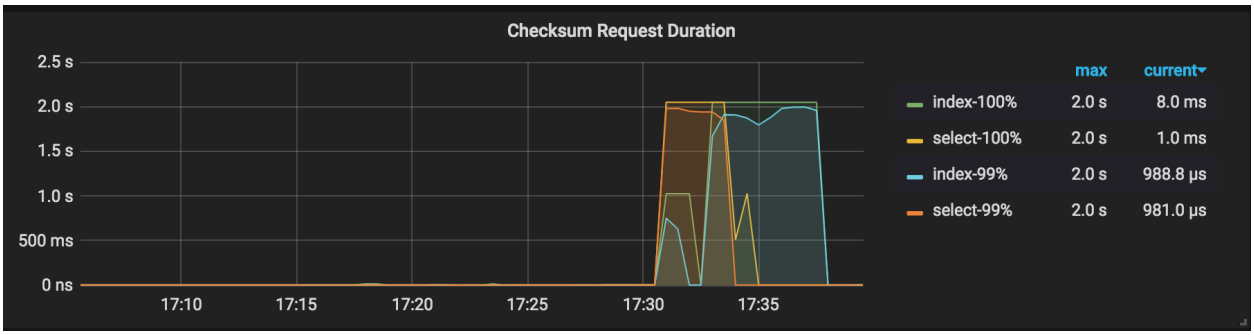


图 47: img

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): 986.43,
    total kv: 5659888624,
    total size(MB): 353227.18,
    avg speed(MB/s): 358.09"]
    ["backup total regions"=7196]
    ["backup checksum"=6m28.291772955s]
    ["backup fast checksum"=24.950298ms]
```

以上日志信息中包含以下内容：

- 备份耗时：`total take(s): 986.43`
- 数据大小：`total size(MB): 353227.18`
- 备份吞吐：`avg speed(MB/s): 358.09`
- 校验耗时：`take=6m28.29s`

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` = 89。

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` 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
```

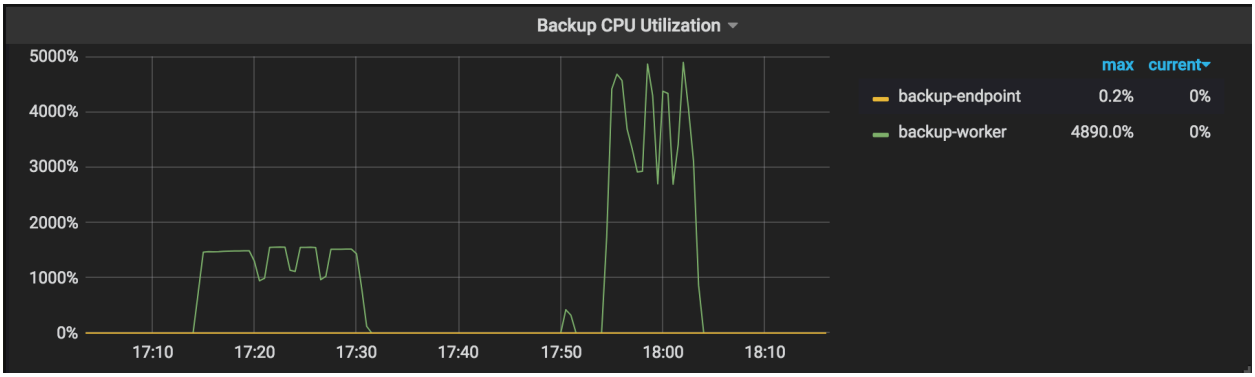


图 48: img

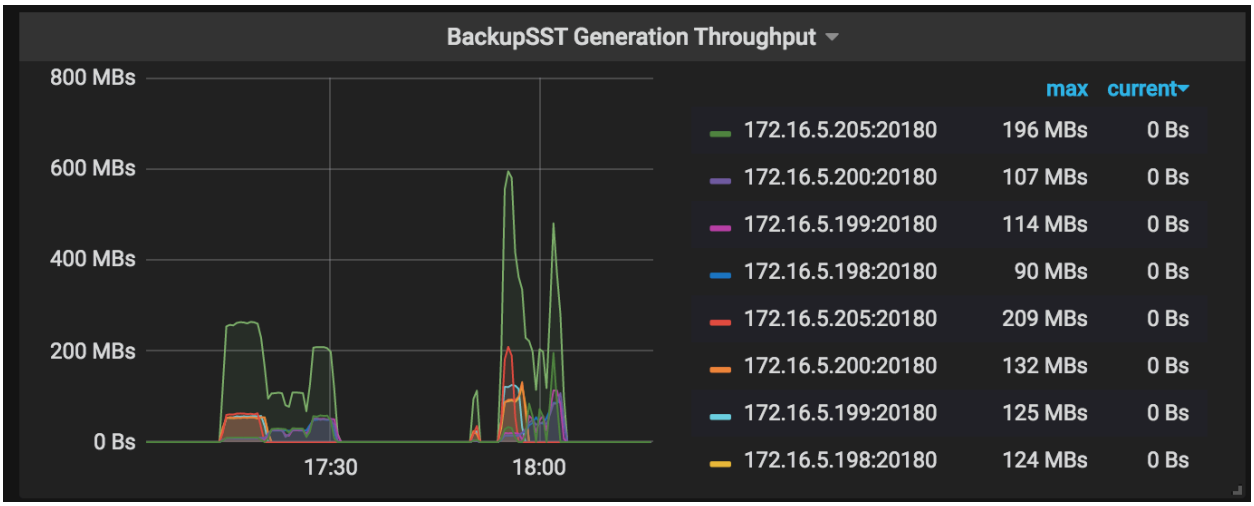


图 49: img

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：

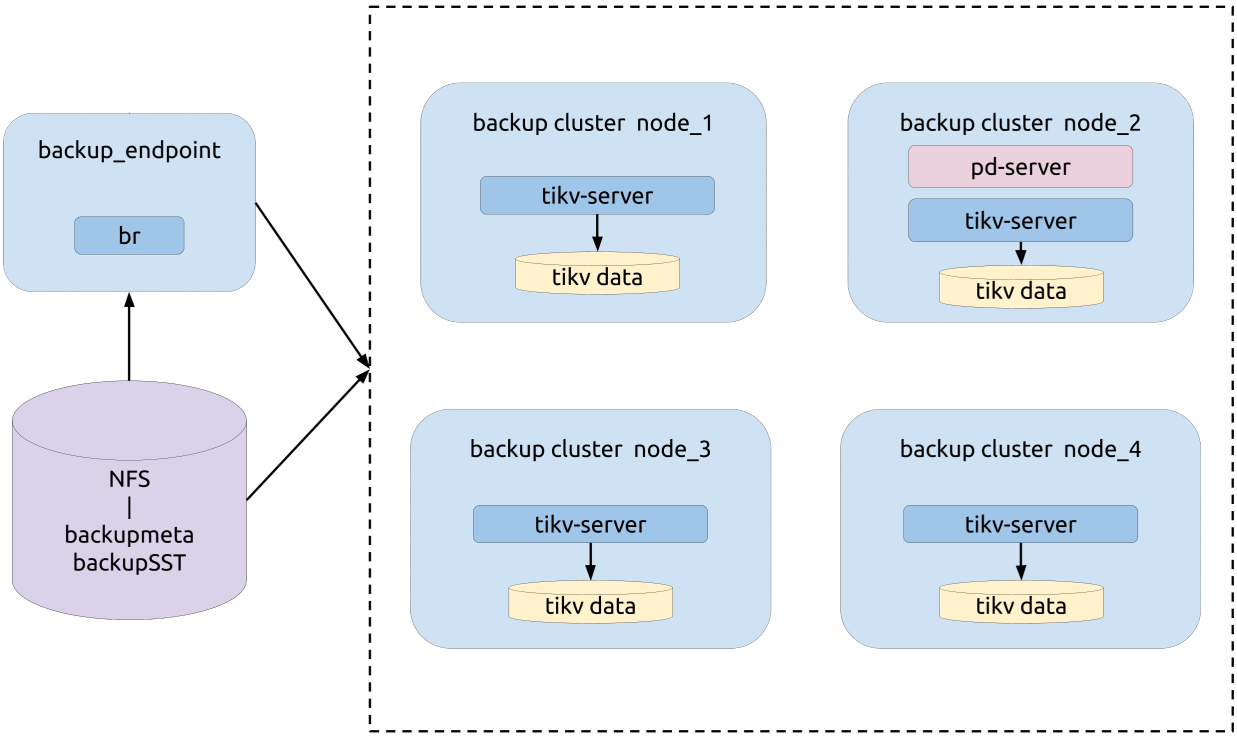


图 50: img

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。

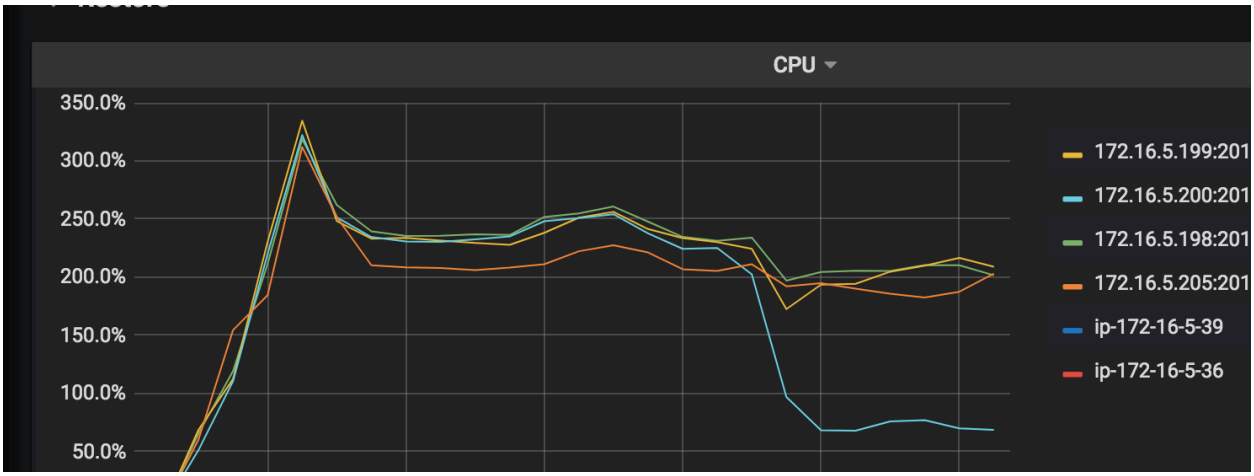


图 51: img

IO Utilization：参与恢复的 TiKV 节点的 I/O 使用率。

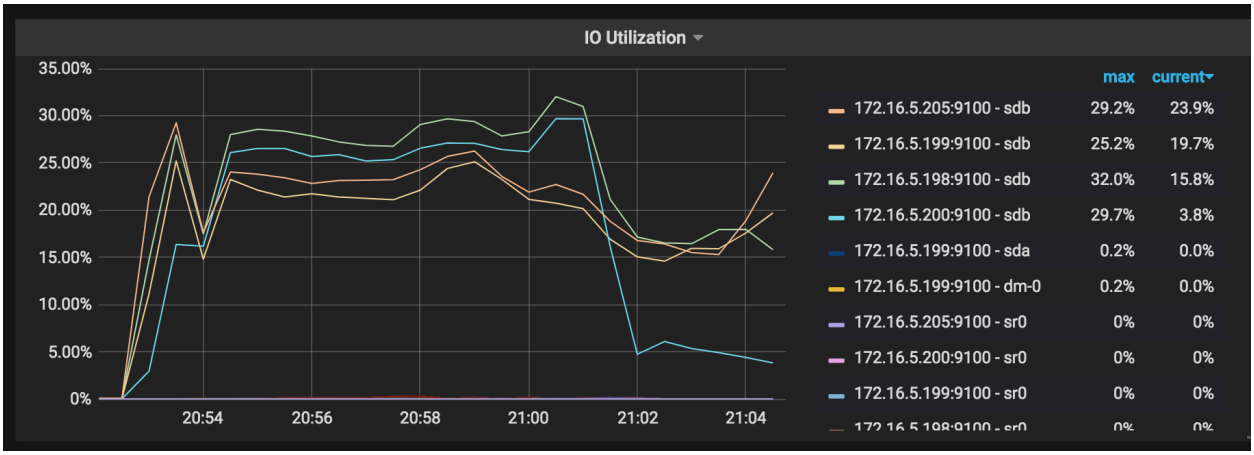


图 52: img

Region 分布：Region 分布越均匀，说明恢复资源利用越充分。

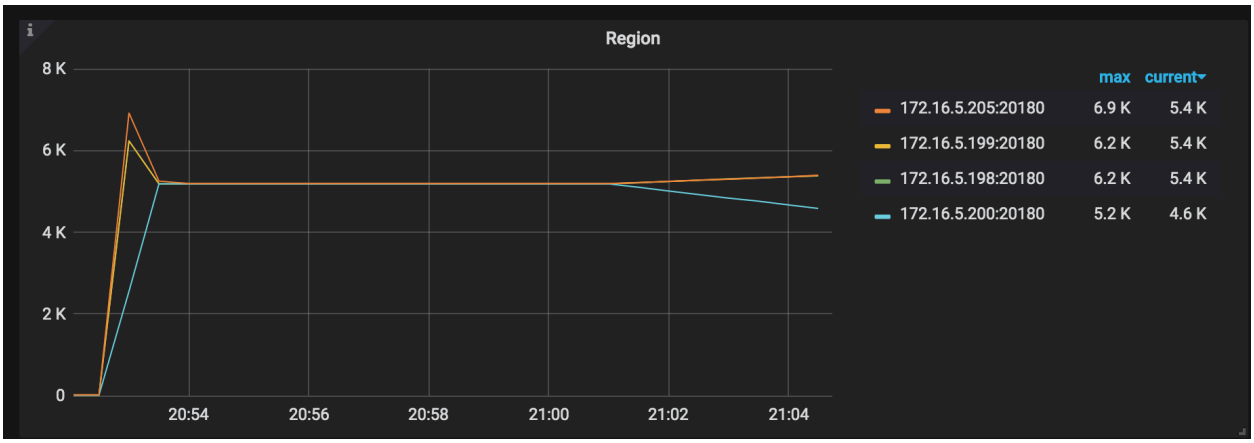


图 53: img

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。

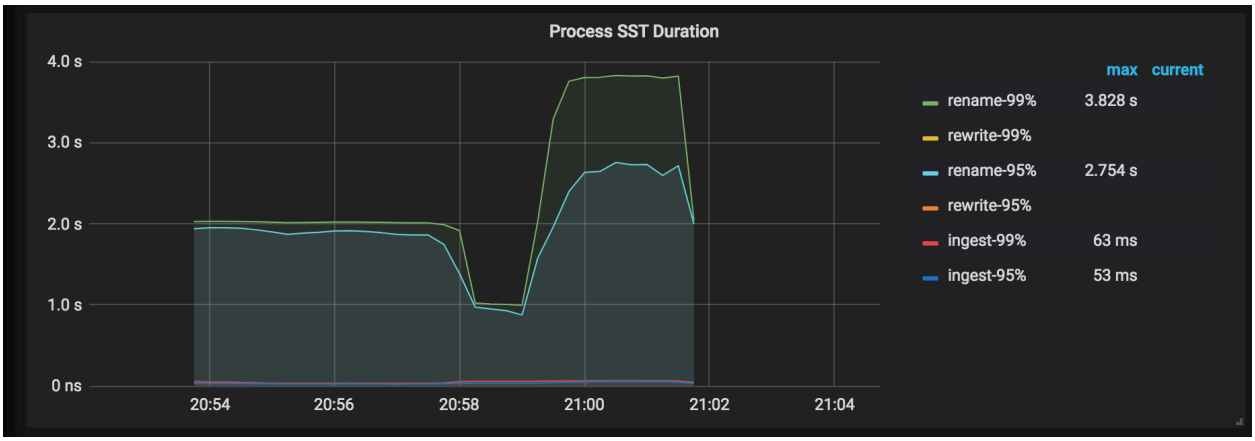


图 54: img

DownLoad SST Throughput：从 External Storage 下载 SST 文件的吞吐。

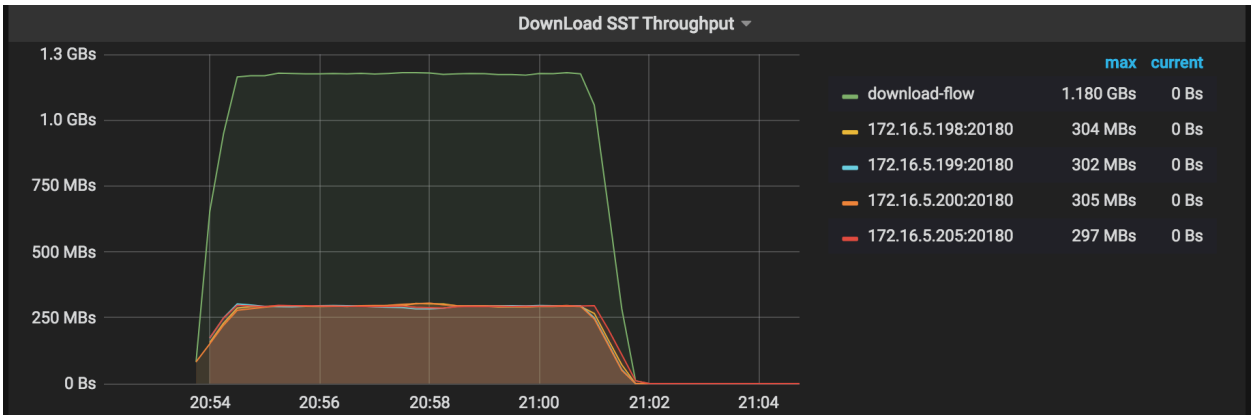


图 55: img

Restore Errors：恢复过程中的错误。

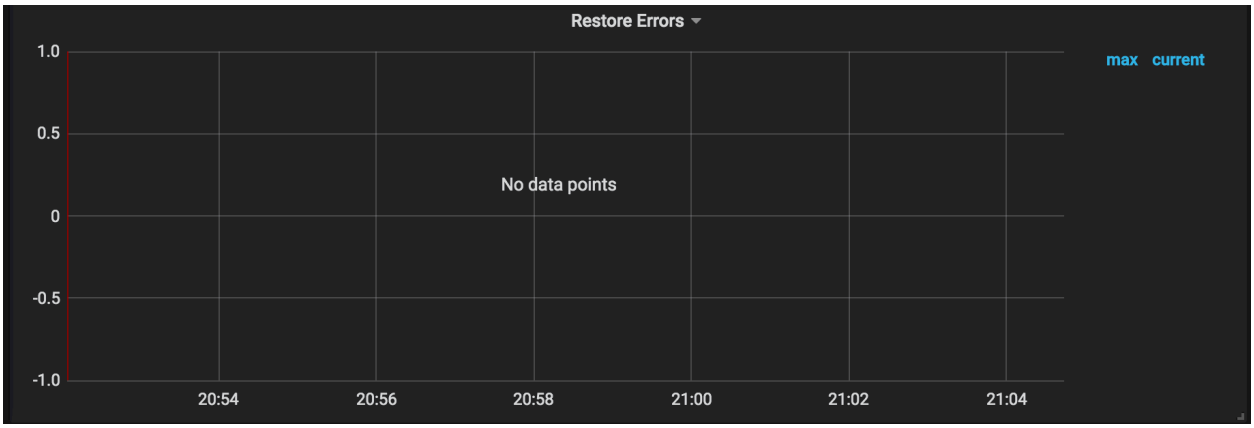


图 56: img

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。

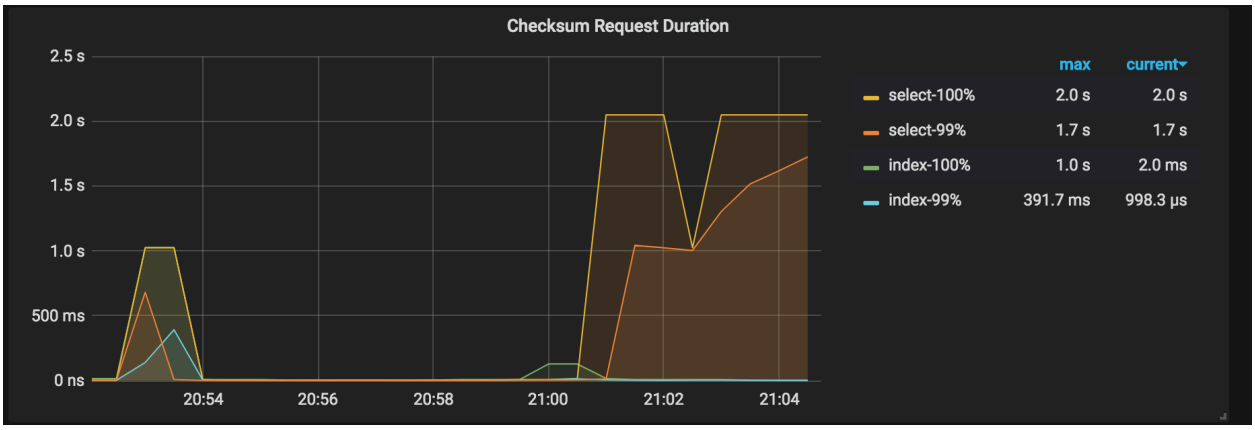


图 57: img

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(s): 961.37,
    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]
```

以上日志信息中包含以下内容：

- 恢复耗时：`total take(s):961.37`
- 数据大小：`total size(MB): 353227.18`
- 恢复吞吐：`avg speed(MB/s): 367.42`
- Region Split 耗时：`take=49.049182743s`
- 校验耗时：`take=6m34.879439498s`

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

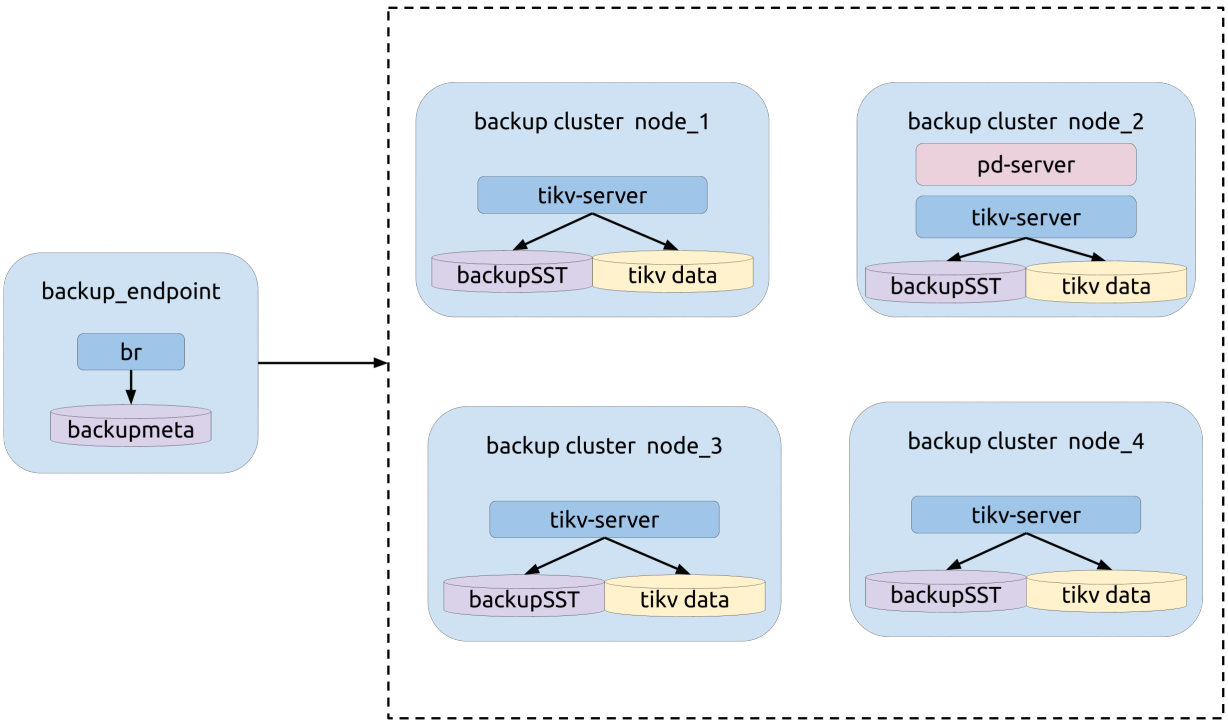


图 58: img

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 59: 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

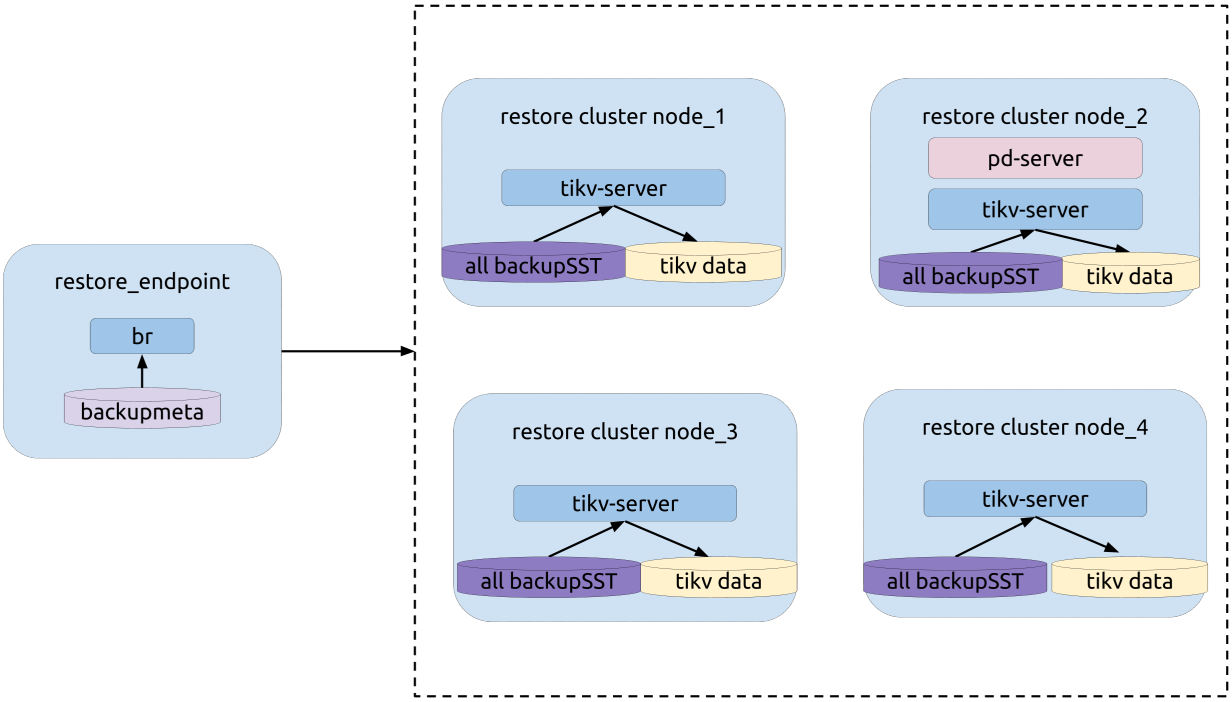


图 60: img

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

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。

3.7.2.2 使用 Dumpling 导出或备份 TiDB 数据

本文档介绍如何使用数据导出工具 Dumpling。该工具可以把存储在 TiDB 中的数据导出为 SQL 或者 CSV 格式，可以用于完成逻辑上的全量备份或者导出。

如果需要直接备份 SST 文件（KV 对）或者对延迟不敏感的增量备份，请参阅BR。

Dumpling 的更多具体用法可以使用 –help 指令查看，或者查看中文使用手册。

使用 Dumpling 时，需要在已经启动的集群上执行导出命令。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。

3.7.2.2.1 下载地址

最新版 Dumpling 的下载地址见下载链接。

3.7.2.2.2 从 TiDB 导出数据

导出到 sql 文件

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 \
  --threads 32 \
  -o /tmp/test \
  -F 256
```

上述命令中，-h、-P、-u 分别是地址，端口，用户。如果需要密码验证，可以用 -p $YOUR_SECRET_PASSWORD 传给 Dumpling。

导出到 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：
>
> 1. --sql 选项暂时仅仅可用于导出 csv 的场景。
> 2. 这里需要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。

筛选导出的数据

使用 --where 指令筛选数据

默认情况下，除了系统数据库中的表之外，Dumpling 会导出整个数据库的表。你可以使用 --where <SQL where expression> 来选定要导出的记录。

```
./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。

使用 --filter 指令筛选数据

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 \
  --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。

使用 -B 或 -T 指令筛选数据

Dumpling 也可以通过 -B 或 -T 参数导出特定的数据库/数据表。

> 注意：
>
> 1. --filter 参数与 -T 参数不可同时使用。
> 2. -T 参数只能接受完整的 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

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。常用参数如下：

- -o 用于选择存储导出文件的目录。
- -F 选项用于指定单个文件的最大大小，默认单位为 MiB。可以接受类似 5GiB 或 8KB 的输入。
- -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。

利用以上参数可以让 Dumpling 的并行度更高。

调整 Dumpling 的数据一致性选项

> 注意：
>
> 在大多数场景下，用户不需要调整 Dumpling 的默认数据一致性选项。

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 参数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 来保证一致性。
- 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。

一切完成之后，你应该可以在 /tmp/test 看到导出的文件了：

```
$ 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
```

另外，假如数据量非常大，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';
```

在操作结束之后，再将 GC 时间调回原样（默认是 10m）：

```
update mysql.tidb set VARIABLE_VALUE = '10m' where VARIABLE_NAME = 'tikv_gc_life_time';
```

最后，所有的这些导出数据都可以用Lightning 导入回 TiDB。

3.7.3 定位异常查询

3.7.3.1 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。

3.7.3.1.1 日志示例

```
#### Time: 2019-08-14T09:26:59.487776265+08:00
#### Txn_start_ts: 410450924122144769
#### User: root@127.0.0.1
#### Conn_ID: 3086
#### Query_time: 1.527627037
```

```
#### Parse_time: 0.000054933
#### Compile_time: 0.000129729
#### 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
#### Mem_max: 525211
#### Succ: true
#### Plan: tidb_decode_plan('
    ↪ ZJAwCTMyXzcJMAkyMAlkYXRhOlRhYmxlU2Nhbl82CjEJMTBfNgkxAR0AdAEY1Dp0LCByYW5nZTpbLWluZiwraW5mXSwga2VlcCBvcmRlc
    ↪ ==')
insert into t select * from t;
```

3.7.3.1.2 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。

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 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。

3.7.3.1.3 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Information Schema 中关于 SLOW_QUERY 表的介绍。

> **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。

3.7.3.1.4 查询 SLOW_QUERY 示例

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; |
+--------------+-----------------------------------------------------------------+
```

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

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 |
+-----------------------------+-------------+
```

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 |
+-----------------------------+-------------+---------------------------------+
```

3.7.3.1.5 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"
```

3.7.3.1.6 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
.
.
.
```

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。

3.7.3.1.7 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

3.7.3.2 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（默认值为 60s），或使用内存超过mem-quota-query（默认值为 32 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。

3.7.3.2.1 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"]
```

3.7.3.2.2 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。

3.8 扩容缩容

3.8.1 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。

> 注意：
>
> 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

3.8.1.1 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。

3.8.1.2 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
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。

3.8.1.3 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。

3.8.1.4 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
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。

3.8.1.5 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。

3.9 升级

3.9.1 TiDB 3.1 升级操作指南

本文档适用于从 TiDB 2.0、2.1、3.0 版本升级至 TiDB 3.1 版本以及 TiDB 3.1 低版本升级至 TiDB 3.1 高版本。目前，TiDB 3.1 版本兼容TiDB Binlog Cluster 版本。

3.9.1.1 升级兼容性说明

- 不支持在升级后回退至 2.1.x 或更旧版本
- 从 2.0.6 之前的版本升级到 3.1 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 3.1，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 3.1
  - 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 3.1 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。

3.9.1.2 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 2.4.2 及以上但不高于 2.7.11 的 Ansible 版本（`2.4.2 ≦ 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。

3.9.1.3 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 或 TiDB 3.1 低版本的 tidb-ansible 文件夹：

```
mv tidb-ansible tidb-ansible-bak
```

下载 TiDB 3.1 版本对应 tag 的 tidb-ansible 下载 TiDB Ansible，默认的文件夹名称为 `tidb-ansible`。

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

3.9.1.4 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。

3.9.1.4.1 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。

3.9.1.4.2 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。
> 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：
> 3.0 版本单机多 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。

3.9.1.5 下载 TiDB 3.1 binary 到中控机

确认 tidb-ansible/inventory.ini 文件中 tidb_version = v3.1.x，然后执行以下命令下载 TiDB 3.1 binary 到中控机。

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

3.9.1.6 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
```

3.9.1.7 滚动升级 TiDB 监控组件

```
ansible-playbook rolling_update_monitor.yml
```

3.10 故障诊断

3.10.1 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。

3.10.1.1 1. 服务不可用

3.10.1.1.1 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）。

3.10.1.1.2 1.2 PD 异常导致服务不可用

查看本文档5. PD 问题。

3.10.1.2 2. 延迟明显升高

3.10.1.2.1 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。

3.10.1.2.2 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。

3.10.1.3 3. TiDB 问题

3.10.1.3.1 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。

3.10.1.3.2 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 32GB，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。

3.10.1.3.3 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。

3.10.1.3.4 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。

3.10.1.4 4. TiKV 问题

3.10.1.4.1 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）。

3.10.1.4.2 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）。

3.10.1.4.3 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。

3.10.1.4.4 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。

3.10.1.4.5 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。

3.10.1.5 5. PD 问题

3.10.1.5.1 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。

3.10.1.5.2 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。详情请参考案例 case-947。
  - 其他原因，需报 bug。

3.10.1.5.3 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。

3.10.1.5.4 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。

3.10.1.6 6. 生态 Tools 问题

3.10.1.6.1 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;` 可解决次问题。
  - 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。

3.10.1.6.2 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。

3.10.1.6.3 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 Lightning 读到的参数是多少；
        * 如果 Lightning 与其他服务（如 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。
- 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。Lightning 或 Importer 之前发生了异常退出。为了防止数据意外损坏，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：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。

3.10.1.7 7. 常见日志分析

3.10.1.7.1 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;
```

3.10.1.7.2 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。

3.10.2 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。

3.10.2.1 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

3.10.2.2 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。

3.10.2.3 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。

3.10.2.4 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。

3.10.2.5 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`。

3.10.2.6 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）。

3.10.2.7 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。

3.10.2.8 TiDB panic

请提供 panic 的 log。

3.10.2.9 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

3.10.2.10 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。

3.10.2.11 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

3.10.3 TiDB Lightning 故障诊断

当 Lightning 遇到不可恢复的错误时便会异常退出，并在日志中记下错误原因。一般可在日志底部找到，也可以搜索 `[error]` 字符串找出中间发生的错误。本文主要描述一些常见的错误及其解决方法。

3.10.3.1 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 Lightning 读到的参数是多少。
2. 如果 Lightning 与其他服务（如 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），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 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。

原因 3：Lightning 版本太旧。

试试最新的版本吧！可能会有改善。

3.10.3.2 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 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。
    - 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 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。

3.10.3.3 Checkpoint for … has invalid status:（错误码）

原因：断点续传已启用。Lightning 或 Importer 之前发生了异常退出。为了防止数据意外损坏，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 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。

3.10.3.4 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 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
```

3.10.3.5 cannot guess encoding for input file, please convert to UTF-8 manually

原因：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。

3.10.3.6 [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 Lightning 与数据源时区一致。

    - 使用 TiDB Ansible 部署的话，修正 `[inventory.ini]` 下的 `timezone` 变量。

    ```
    # inventory.ini
    [all:vars]
    timezone = Asia/Shanghai
    ```

    - 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`。

3.10.3.7 `[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。

3.10.3.8 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。

4 参考手册

4.1 SQL

4.1.1 与 MySQL 兼容性对比

TiDB 支持 MySQL 传输协议及其绝大多数的语法。这意味着您现有的 MySQL 连接器和客户端都可以继续使用。大多数情况下您现有的应用都可以迁移至 TiDB，无需任何代码修改。

当前 TiDB 服务器官方支持的版本为 MySQL 5.7。大部分 MySQL 运维工具（如 PHPMyAdmin, Navicat, MySQL Workbench 等），以及备份恢复工具（如 mysqldump, Mydumper/myloader）等都可以直接使用。

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。

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

4.1.1.1 不支持的特性

- 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/空间函数与索引 #1793
- 非 `ascii/latin1/binary/utf8/utf8mb4` 的字符集
- BINARY 之外的排序规则
- 增加/删除主键
- 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

4.1.1.2 与 MySQL 有差异的特性

4.1.1.2.1 自增 ID

TiDB 中，自增列只保证自增且唯一，也能保证在单个 TiDB server 中自增，但不保证多个 TiDB server 中自增，不保证连续分配。TiDB 目前采用批量分配 ID 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。TiDB 可通过 tidb_allow_remove_auto_inc 系统变量开启或者关闭删除列的 AUTO_INCREMENT 属性。删除列属性的语法是：alter table modify 或 alter table change。

在集群中有多个 tidb-server 实例时，如果表结构中有自增 ID，建议不要混用缺省值和自定义值，否则在如下情况下会遇到问题。

假设有这样一个带有自增 ID 的表：

```
create table t(id int unique key AUTO_INCREMENT, c int);
```

TiDB 实现自增 ID 的原理是每个 tidb-server 实例缓存一段 ID 值用于分配（目前会缓存 30000 个 ID），用完这段值再去取下一段。

假设集群中有两个 tidb-server 实例 A 和 B（A 缓存 [1,30000] 的自增 ID，B 缓存 [30001,60000] 的自增 ID），依次执行如下操作：

1. 客户端向 B 插入一条将 id 设置为 1 的语句 insert into t values (1, 1)，并执行成功。
2. 客户端向 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，所以会分配 1 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 1 的数据，最终返回 Duplicated Error 错误。

另外，从 TiDB 3.0.4 版本开始，TiDB 将通过系统变量 @@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。移除后不可再恢复（因为 TiDB 不支持添加列的 AUTO_INCREMENT 属性）

> 注意：
>
> 在没有指定主键的情况下 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)
```

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。其中缓存大小可能会被自增列和 _tidb_rowid 共同消耗。此外如果在 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。

4.1.1.2.2 Performance schema

Performance schema 表在 TiDB 中返回结果为空。TiDB 使用Prometheus 和 Grafana 来监测性能指标。

从 TiDB 3.0.4 版本开始，TiDB 支持 events_statements_summary_by_digest，参见Statement Summary Table。

4.1.1.2.3 查询计划

TiDB 的查询计划（EXPLAIN/EXPLAIN FOR）输出格式与 MySQL 差别较大，同时 EXPLAIN FOR 的输出内容与权限设置与 MySQL 不一致，参见理解 TiDB 执行计划。

4.1.1.2.4 内建函数

TiDB 支持常用的 MySQL 内建函数，但是不是所有的函数都已经支持，具体请参考语法文档。

4.1.1.2.5 DDL

在 TiDB 中，运行的 DDL 操作不会影响对表的读取或写入。但是，目前 DDL 变更有如下一些限制：

- Add Index
  - 不支持同时创建多个索引
  - 不支持 VISIBLE/INVISIBLE 的索引
  - 不支持通过 ALTER TABLE 在生成列上添加索引
  - 其他类型的 Index Type (HASH/BTREE/RTREE) 只有语法支持，功能不支持
- Add Column
  - 不支持同时创建多个列
  - 不支持将新创建的列设为主键或唯一索引，也不支持将此列设成 AUTO_INCREMENT 属性

- Drop Column: 不支持删除主键列或索引列
- Change/Modify Column
  - 不支持有损变更，比如从 BIGINT 变为 INTEGER，或者从 VARCHAR(255) 变为 VARCHAR(10)，否则可能输出的错误信息 length %d is less than origin %d。
  - 不支持修改 DECIMAL 类型的精度
  - 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。
  - 不支持更改 UNSIGNED 属性
  - 只支持将 CHARACTER SET 属性从 utf8 更改为 utf8mb4
- Alter Database
  - 只支持将 CHARACTER SET 属性从 utf8 更改为 utf8mb4
- LOCK [=] {DEFAULT|NONE|SHARED|EXCLUSIVE}: TiDB 支持的语法，但是在 TiDB 中不会生效。所有支持的 DDL 变更都不会锁表。
- ALGORITHM [=] {DEFAULT|INSTANT|INPLACE|COPY}: TiDB 完全支持 ALGORITHM=INSTANT 和 ALGORITHM=INPLACE 语法，但运行过程与 MySQL 有所不同，因为 MySQL 中的一些 INPLACE 操作实际上是 TiDB 中的 INSTANT 操作。ALGORITHM=COPY 语法在 TiDB 中不会生效，会返回警告信息。
- 单个 ALTER TABLE 语句中无法完成多个操作。例如，不能用一个语句来添加多个列或多个索引。
- Table Option 不支持以下语法
  - WITH/WITHOUT VALIDATION
  - SECONDARY_LOAD/SECONDARY_UNLOAD
  - CHECK/DROP CHECK
  - STATS_AUTO_RECALC/STATS_SAMPLE_PAGES
  - SECONDARY_ENGINE
  - ENCRYPTION
- Table Partition 不支持以下语法
  - PARTITION BY LIST
  - PARTITION BY KEY
  - SUBPARTITION
  - {CHECK|EXCHANGE|TRUNCATE|OPTIMIZE|REPAIR|IMPORT|DISCARD|REBUILD|REORGANIZE} PARTITION

4.1.1.2.6 ANALYZE TABLE

- ANALYZE TABLE 语句在 TiDB 和 MySQL 中表现不同。在 MySQL/InnoDB 中，它是一个轻量级语句，执行过程较短；而在 TiDB 中，它会完全重构表的统计数据，语句执行过程较长。

4.1.1.2.7 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。而 TiDB 中该语法所返回的结果并不承诺任何顺序，与 MySQL 8.0 的行为一致。

4.1.1.2.8 视图

目前 TiDB 不支持对视图进行 UPDATE、 INSERT、 DELETE 等写入操作。

4.1.1.2.9 存储引擎

出于兼容性原因，TiDB 支持使用备用存储引擎创建表的语法。元数据命令将表描述为 InnoDB 存储引擎：

```
CREATE TABLE t1 (a INT) ENGINE=MyISAM;
```

```
Query OK, 0 rows affected (0.14 sec)
```

```
SHOW CREATE TABLE t1;
```

```
*************************** 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)
```

从架构上讲，TiDB 确实支持类似 MySQL 的存储引擎抽象，在启动 TiDB（通常是 tikv）时--store 选项指定的引擎中创建用户表。

4.1.1.2.10 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。

MySQL 中的 NO_DIR_IN_CREATE 和 NO_ENGINE_SUBSTITUTION 的 SQL 模式可用于解决兼容性问题，并不适用于 TiDB。

4.1.1.2.11 默认设置的区别

- 默认字符集与 MySQL 不同：
  - TiDB 中为 utf8mb4
  - MySQL 5.7 中为 latin1，MySQL 8.0 中修改为 utf8mb4
- 默认排序规则不同：
  - TiDB 中，utf8mb4 的默认排序规则为 utf8mb4_bin
  - MySQL 5.7 中，utf8mb4 的默认排序规则为 utf8mb4_general_ci，MySQL 8.0 中修改为 utf8mb4_0900_ai_ci
    ↪
  - 请使用SHOW CHARACTER SET 语句查看所有字符集的默认排序规则

- `foreign_key_checks` 的默认值不同：
  - TiDB 中该值默认为 OFF，并且目前 TiDB 只支持设置该值为 OFF。
  - MySQL 5.7 中该值默认为 ON。
- 默认 SQL mode 与 MySQL 5.7 相同，与 MySQL 8.0 不同：
  - TiDB 中为 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 中默认设置：
    - MySQL 5.7 的默认 SQL mode 与 TiDB 相同
    - MySQL 8.0 中为 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` 的默认值不同：
  - TiDB 中该值默认为 2，并且目前 TiDB 只支持设置该值为 2
  - MySQL 中默认设置：
    - 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` 的默认值不同：
  - TiDB 中该值默认为 ON，并且目前 TiDB 只支持设置该值为 ON
  - MySQL 中默认设置：
    - MySQL 5.7：OFF
    - MySQL 8.0：ON

4.1.1.2.12 日期时间处理的区别

时区

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。

TiDB 不需要导入时区表数据也能使用所有时区名称，采用系统当前安装的所有时区规则进行计算（一般为 `tzdata` 包），且无法通过导入时区表数据的形式修改计算规则。

> 注意：
>
> 能下推到 TiKV 的时间相关表达式会由 TiKV 进行计算。TiKV 总是采用 TiKV 内置时区规则计算，而不依赖于系统所安装的时区规则。若系统安装的时区规则与 TiKV 内置的时区规则版本不匹配，则在少数情况下可能发生能插入的时间数据无法再读出来的问题。例如，若系统上安装了 tzdata 2018a 时区规则，则在时区设置为 Asia/Shanghai 或时区设置为本地时区且本地时区为 Asia/Shanghai 的情况下，时间 1988-04-17 02:00:00 可以被正常插入 TiDB 3.0 RC.1，但该记录对于特定类型 SQL 则无法再读出来，原因是 TiKV 3.0 RC.1 依据的 tzdata 2018i 规则中该时间在 Asia/Shanghai 时区中不存在（夏令时时间后移一小时）。
>
> TiKV 各个版本内置的时区规则如下：

- 3.0.0 RC.1 及以后：tzdata 2018i
- 2.1.0 RC.1 及以后：tzdata 2018e

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，不建议将这两个模式设为禁用。尽管将这些模式设为禁用时 TiDB 仍可正常使用，但 TiKV coprocessor 会受到影响，具体表现为，执行特定类型的语句，将日期和时间处理函数下推到 TiKV 时可能会导致语句错误。

字符串类型行末空格的处理

目前 TiDB 在进行数据插入时，对于 VARCHAR 类型会保留行末空格，对于 CHAR 类型会插入截断空格后的数据。在没有索引的情况下，TiDB 和 MySQL 行为保持一致。如果 VARCHAR 类型上有 UNIQUE 索引，MySQL 在判断是否重复的时候，和处理 CHAR 类型一样，先截断 VARCHAR 数据末行空格再作判断；TiDB 则是按照保留空格的情况处理。

在做比较时，MySQL 会先截去常量和 Column 的末尾空格再作比较，而 TiDB 则是保留常量和 Column 的末尾空格来做精确比较。

4.1.1.2.13 类型系统的区别

以下的列类型 MySQL 支持，但 TiDB 不支持：

- FLOAT4/FLOAT8
- FIXED (alias for DECIMAL)
- SERIAL (alias for BIGINT UNSIGNED NOT NULL AUTO_INCREMENT UNIQUE)
- SQL_TSI_*（包括 SQL_TSI_YEAR、SQL_TSI_MONTH、SQL_TSI_WEEK、SQL_TSI_DAY、SQL_TSI_HOUR、SQL_TSI_MINUTE 和 SQL_TSI_SECOND）

4.1.1.2.14 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。

4.1.2 SQL 语言结构

4.1.2.1 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。

4.1.2.1.1 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。

4.1.2.1.2 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。

4.1.2.1.3 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。

4.1.2.1.4 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)
```

4.1.2.1.5 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)
```

4.1.2.1.6 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)
```

4.1.2.1.7 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。

4.1.2.2 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。

4.1.2.2.1 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。

4.1.2.3 关键字和保留字

关键字在 SQL 中有特殊的意义，例如 SELECT，UPDATE，DELETE，在作为表名跟函数名的时候，需要特殊对待，例如作为表名，保留字需要被反引号包住：

```
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)
```

BEGIN 和 END 是关键字，但不是保留字，所以不需要反引号：

```
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 中的关键字和保留字。保留字用 (R) 来标识。窗口函数的保留字用 (R-Window) 来标识：

A

- ACTION
- ADD (R)
- ADDDATE
- ADMIN
- AFTER

- ALL (R)
- ALTER (R)
- ALWAYS
- ANALYZE(R)
- AND (R)
- ANY
- AS (R)
- ASC (R)
- ASCII
- AUTO_INCREMENT
- AUTO_RANDOM
- AVG
- AVG_ROW_LENGTH

B

- BEGIN
- BETWEEN (R)
- BIGINT (R)
- BINARY (R)
- BINLOG
- BIT
- BIT_XOR
- BLOB (R)
- BOOL
- BOOLEAN
- BOTH (R)
- BTREE
- BY (R)
- BYTE

C

- CASCADE (R)
- CASE (R)
- CAST
- CHANGE (R)
- CHAR (R)
- CHARACTER (R)
- CHARSET
- CHECK (R)
- CHECKSUM
- COALESCE
- COLLATE (R)
- COLLATION

- COLUMN (R)
- COLUMNS
- COMMENT
- COMMIT
- COMMITTED
- COMPACT
- COMPRESSED
- COMPRESSION
- CONNECTION
- CONSISTENT
- CONSTRAINT (R)
- CONVERT (R)
- COUNT
- CREATE (R)
- CROSS (R)
- CUME_DIST (R-Window)
- CURRENT_DATE (R)
- CURRENT_TIME (R)
- CURRENT_TIMESTAMP (R)
- CURRENT_USER (R)
- CURTIME

D

- DATA
- DATABASE (R)
- DATABASES (R)
- DATE
- DATE_ADD
- DATE_SUB
- DATETIME
- DAY
- DAY_HOUR (R)
- DAY_MICROSECOND (R)
- DAY_MINUTE (R)
- DAY_SECOND (R)
- DDL
- DEALLOCATE
- DEC
- DECIMAL (R)
- DEFAULT (R)
- DELAY_KEY_WRITE
- DELAYED (R)
- DELETE (R)
- DENSE_RANK (R-Window)

- DESC (R)
- DESCRIBE (R)
- DISABLE
- DISTINCT (R)
- DISTINCTROW (R)
- DIV (R)
- DO
- DOUBLE (R)
- DROP (R)
- DUAL (R)
- DUPLICATE
- DYNAMIC

E

- ELSE (R)
- ENABLE
- ENCLOSED
- END
- ENGINE
- ENGINES
- ENUM
- ESCAPE
- ESCAPED
- EVENTS
- EXCLUSIVE
- EXECUTE
- EXISTS
- EXPLAIN (R)
- EXTRACT

F

- FALSE (R)
- FIELDS
- FIRST
- FIRST_VALUE (R-Window)
- FIXED
- FLOAT (R)
- FLUSH
- FOR (R)
- FORCE (R)
- FOREIGN (R)
- FORMAT
- FROM (R)

- FULL
- FULLTEXT (R)
- FUNCTION

G

- GENERATED (R)
- GET_FORMAT
- GLOBAL
- GRANT (R)
- GRANTS
- GROUP (R)
- GROUP_CONCAT
- GROUPS (R-Window)

H

- HASH
- HAVING (R)
- HIGH_PRIORITY (R)
- HOUR
- HOUR_MICROSECOND (R)
- HOUR_MINUTE (R)
- HOUR_SECOND (R)

I

- IDENTIFIED
- IF (R)
- IGNORE (R)
- IN (R)
- INDEX (R)
- INDEXES
- INFILE (R)
- INNER (R)
- INSERT (R)
- INT (R)
- INTEGER (R)
- INTERVAL (R)
- INTO (R)
- IS (R)
- ISOLATION

J

- JOBS
- JOIN (R)
- JSON

K

- KEY (R)
- KEY_BLOCK_SIZE
- KEYS (R)
- KILL (R)

L

- LAG (R-Window)
- LAST_VALUE (R-Window)
- LEAD (R-Window)
- LEADING (R)
- LEFT (R)
- LESS
- LEVEL
- LIKE (R)
- LIMIT (R)
- LINES (R)
- LOAD (R)
- LOCAL
- LOCALTIME (R)
- LOCALTIMESTAMP (R)
- LOCK (R)
- LONGBLOB (R)
- LONGTEXT (R)
- LOW_PRIORITY (R)

M

- MAX
- MAX_ROWS
- MAXVALUE (R)
- MEDIUMBLOB (R)
- MEDIUMINT (R)
- MEDIUMTEXT (R)
- MICROSECOND
- MIN
- MIN_ROWS
- MINUTE
- MINUTE_MICROSECOND (R)

- MINUTE_SECOND (R)
- MIN
- MIN_ROWS
- MINUTE
- MINUTE_MICROSECOND
- MINUTE_SECOND
- MOD (R)
- MODE
- MODIRY
- MONTH

N

- NAMES
- NATIONAL
- NATURAL (R)
- NO
- NO_WRITE_TO_BINLOG (R)
- NONE
- NOT (R)
- NOW
- NTH_VALUE (R-Window)
- NTILE (R-Window)
- NULL (R)
- NUMERIC (R)
- NVARCHAR (R)

O

- OFFSET
- ON (R)
- ONLY
- OPTION (R)
- OR (R)
- ORDER (R)
- OUTER (R)
- OVER (R-Window)

P

- PARTITION (R)
- PARTITIONS
- PASSWORD
- PERCENT_RANK (R-Window)
- PLUGINS

- POSITION
- PRECISION (R)
- PREPARE
- PRIMARY (R)
- PRIVILEGES
- PROCEDURE (R)
- PROCESS
- PROCESSLIST

Q

- QUARTER
- QUERY
- QUICK

R

- RANGE (R)
- RANK (R-Window)
- READ (R)
- REAL (R)
- REDUNDANT
- REFERENCES (R)
- REGEXP (R)
- RENAME (R)
- REPEAT (R)
- REPEATABLE
- REPLACE (R)
- RESTRICT (R)
- REVERSE
- REVOKE (R)
- RIGHT (R)
- RLIKE (R)
- ROLLBACK
- ROW
- ROW_COUNT
- ROW_FORMAT
- ROW_NUMBER (R-Window)
- ROWS (R-Window)

S

- SCHEMA
- SCHEMAS
- SECOND

- SECOND_MICROSECOND (R)
- SELECT (R)
- SERIALIZABLE
- SESSION
- SET (R)
- SHARE
- SHARED
- SHOW (R)
- SIGNED
- SMALLINT (R)
- SNAPSHOT
- SOME
- SQL_CACHE
- SQL_CALC_FOUND_ROWS (R)
- SQL_NO_CACHE
- START
- STARTING (R)
- STATS
- STATS_BUCKETS
- STATS_HISTOGRAMS
- STATS_META
- STATS_PERSISTENT
- STATUS
- STORED (R)
- SUBDATE
- SUBSTR
- SUBSTRING
- SUM
- SUPER

T

- TABLE (R)
- TABLES
- TERMINATED (R)
- TEXT
- THAN
- THEN (R)
- TIDB
- TIDB_INLJ
- TIDB_SMJ
- TIME
- TIMESTAMP
- TIMESTAMPADD
- TIMESTAMPDIFF

- TINYBLOB (R)
- TINYINT (R)
- TINYTEXT (R)
- TO (R)
- TRAILING (R)
- TRANSACTION
- TRIGGER (R)
- TRIGGERS
- TRIM
- TRUE (R)
- TRUNCATE

U

- UNCOMMITTED
- UNION (R)
- UNIQUE (R)
- UNKNOWN
- UNLOCK (R)
- UNSIGNED (R)
- UPDATE (R)
- USE (R)
- USER
- USING (R)
- UTC_DATE (R)
- UTC_TIME (R)
- UTC_TIMESTAMP (R)

V

- VALUE
- VALUES (R)
- VARBINARY (R)
- VARCHAR (R)
- VARIABLES
- VIEW
- VIRTUAL (R)

W

- WARNINGS
- WEEK
- WHEN (R)
- WHERE (R)
- WINDOW (R-Window)

- WITH (R)
- WRITE (R)

X

- XOR (R)

Y

- YEAR
- YEAR_MONTH (R)

Z

- ZEROFILL (R)

4.1.2.4 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。

4.1.2.4.1 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';
```

4.1.2.4.2 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。

4.1.2.5 表达式语法 (Expression Syntax)

在 TiDB 中，以下规则是表达式的语法，你可以在 parser/parser.y 中找到定义。TiDB 的语法解析是基于 yacc 的。

```
Expression:
      singleAtIdentifier assignmentEq Expression
    | Expression logOr Expression
    | Expression "XOR" Expression
    | Expression logAnd Expression
    | "NOT" Expression
    | Factor IsOrNotOp trueKwd
```

```
    | Factor IsOrNotOp falseKwd
    | Factor IsOrNotOp "UNKNOWN"
    | Factor

Factor:
      Factor IsOrNotOp "NULL"
    | Factor CompareOp PredicateExpr
    | Factor CompareOp singleAtIdentifier assignmentEq PredicateExpr
    | Factor CompareOp AnyOrAll SubSelect
    | PredicateExpr

PredicateExpr:
      PrimaryFactor InOrNotOp '(' ExpressionList ')'
    | PrimaryFactor InOrNotOp SubSelect
    | PrimaryFactor BetweenOrNotOp PrimaryFactor "AND" PredicateExpr
    | PrimaryFactor LikeOrNotOp PrimaryExpression LikeEscapeOpt
    | PrimaryFactor RegexpOrNotOp PrimaryExpression
    | PrimaryFactor

PrimaryFactor:
      PrimaryFactor '|' PrimaryFactor
    | PrimaryFactor '&' PrimaryFactor
    | PrimaryFactor "<<" PrimaryFactor
    | PrimaryFactor ">>" PrimaryFactor
    | PrimaryFactor '+' PrimaryFactor
    | PrimaryFactor '-' PrimaryFactor
    | PrimaryFactor '*' PrimaryFactor
    | PrimaryFactor '/' PrimaryFactor
    | PrimaryFactor '%' PrimaryFactor
    | PrimaryFactor "DIV" PrimaryFactor
    | PrimaryFactor "MOD" PrimaryFactor
    | PrimaryFactor '^' PrimaryFactor
    | PrimaryExpression

PrimaryExpression:
      Operand
    | FunctionCallKeyword
    | FunctionCallNonKeyword
    | FunctionCallAgg
    | FunctionCallGeneric
    | Identifier jss stringLit
    | Identifier juss stringLit
    | SubSelect
    | '!' PrimaryExpression
    | '~'  PrimaryExpression
```

```
    | '-' PrimaryExpression
    | '+' PrimaryExpression
    | "BINARY" PrimaryExpression
    | PrimaryExpression "COLLATE" StringName
```

4.1.2.6 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)
```

4.1.2.6.1 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。

4.1.2.6.2 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。

4.1.2.6.3 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**
>
> 由于 TiDB 可执行注释语法和优化器注释语法在 MySQL 客户端 5.7.7 之前的版本中，会被默认当成 comment 清除掉，如果需要在旧的客户端使用这两种语法，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`。

更多细节，请参考 MySQL 文档。

4.1.3 表属性和列属性

4.1.3.1 AUTO_RANDOM 从 v3.1.0 版本开始引入

> **警告：**
>
> 当前 AUTO_RANDOM 属性为实验功能，不建议在生产环境中使用。在后续版本中，AUTO_RANDOM 的语法或语义可能会变化。

使用 AUTO_RANDOM 功能前，须在 TiDB 配置文件 `experimental` 部分设置 `allow-auto-random = true`。该参数详情可参见`allow-auto-random`。

4.1.3.1.1 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 int 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。要避免这种性能下降，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 int PRIMARY KEY AUTO_RANDOM, b varchar(255))
```

或者

```
CREATE TABLE t (a int AUTO_RANDOM, b varchar(255), PRIMARY KEY (a))
```

此时再执行形如 INSERT INTO t(b)values... 的 INSERT 语句。

- 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。
- 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。

自动分配值的计算方式如下：

该行值在二进制形式下，除去最高位（无论是 unsigned 还是 signed）的次高五位（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 int 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: 268435455 |
+-------+------+----------------------------------------------------+
```

另外，含有 AUTO_RANDOM 属性的表在系统表 information_schema.tables 中 TIDB_ROW_ID_SHARDING_INFO 一列的值为 PK_AUTO_RANDOM_BITS=x，其中 x 为 shard bits 的数量。

要获取上一次 TiDB 隐式分配的 ID，可以使用 SELECT last_insert_id() 查看，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       |
+------------------+
```

4.1.3.1.2 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 int 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。

4.1.3.1.3 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。例外情况见关于 alter-primary-key 配置项的说明。
- 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。

关于 alter-primary-key 配置项的说明

- 当 alter-primary-key = true 时，即使是整型主键列，也不支持使用 AUTO_RANDOM。
- 配置文件中的 alter-primary-key 和 allow-auto-random 两个配置项的值不允许同时为 true。

4.1.4 数据类型

4.1.4.1 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。

4.1.4.2 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。

4.1.4.3 数值类型

4.1.4.3.1 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）

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 MySQL 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

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]
```

> 警告：
>
> 与在 MySQL 中一样，FLOAT 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。

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

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]
```

4.1.4.4 日期和时间类型

4.1.4.4.1 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）。

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）。

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
);
```

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)
```

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，时间部分被舍弃，因为 TIME 不包含任何时间信息。

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 |
+---------------------|----------------|--------------------+
```

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。

4.1.4.5 字符串类型

4.1.4.5.1 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。

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。

```
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。

```
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。

4.1.4.6 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。

4.1.5 函数与操作符

4.1.5.1 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。

4.1.5.2 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。

4.1.5.3 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

4.1.5.3.1 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.

4.1.5.3.2 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.

4.1.5.3.3 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.

4.1.5.3.4 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.

4.1.5.4 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

4.1.5.5 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。

4.1.5.5.1 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

4.1.5.5.2 不支持的函数

- LOAD_FILE()
- MATCH
- SOUNDEX()
- SOUNDS LIKE
- WEIGHT_STRING()

4.1.5.6 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。

4.1.5.6.1 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

4.1.5.6.2 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

4.1.5.7 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。

4.1.5.7.1 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

4.1.5.8 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

4.1.5.9 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。

4.1.5.9.1 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

4.1.5.10 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。

4.1.5.10.1 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_LENGTH()	返回字符串解压后的长度
CREATE_ASYMMETRIC_PRIV_KEY()	创建私钥
CREATE_ASYMMETRIC_PUB_KEY()	创建公钥
CREATE_DH_PARAMETERS()	创建 DH 共享密钥
CREATE_DIGEST()	从字符串创建摘要
ASYMMETRIC_DECRYPT()	使用公钥或私钥解密密文
ASYMMETRIC_DERIVE()	从非对称密钥导出对称密钥
ASYMMETRIC_ENCRYPT()	使用公钥或私钥加密明文

函数名	功能描述
ASYMMETRIC_SIGN ↪ ()	从摘要创建签名
ASYMMETRIC_VERIFY ↪ ()	验证签名字符串是否匹配摘要字符串

4.1.5.10.2 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。

4.1.5.11 信息函数

TiDB 支持使用 MySQL 5.7 中提供的大部分信息函数。

4.1.5.11.1 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

4.1.5.11.2 不支持的函数

- CHARSET()
- COERCIBILITY()
- COLLATION()

4.1.5.12 JSON 函数及语法糖

> **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）。

4.1.5.12.1 创建 JSON 值的函数

函数及语法糖	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

4.1.5.12.2 搜索 JSON 值的函数

函数及语法糖	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)` 的语法糖
->>	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))` 的语法糖
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

4.1.5.12.3 修改 JSON 值的函数

函数及语法糖	功能描述
JSON_INSERT(json_doc, path, val[, path, val] ···)	在 JSON 文档中在某一路径下插入子文档

函数及语法糖	功能描述
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
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

4.1.5.12.4 返回 JSON 值属性的函数

函数及语法糖	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

4.1.5.12.5 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_APPEND 及其别名 JSON_ARRAY_APPEND
- JSON_ARRAY_INSERT
- JSON_DEPTH
- JSON_MERGE_PATCH
- JSON_PRETTY
- JSON_SEARCH
- JSON_STORAGE_SIZE
- JSON_VALID

- JSON_ARRAYAGG
- JSON_OBJECTAGG

4.1.5.13 GROUP BY 聚合函数

本文将详细介绍 TiDB 支持的聚合函数。

4.1.5.13.1 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

> 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。

4.1.5.13.2 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。

4.1.5.13.3 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;
```

4.1.5.13.4 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。

- STD, STDDEV, STDDEV_POP
- STDDEV_SAMP
- VARIANCE, VAR_POP
- VAR_SAMP
- JSON_ARRAYAGG
- JSON_OBJECTAGG

4.1.5.14 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

函数名	功能描述
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

4.1.5.15 其他函数

TiDB 支持使用 MySQL 5.7 中提供的大部分其他函数。

4.1.5.15.1 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

4.1.5.15.2 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

4.1.5.16 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

4.1.5.16.1 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.

4.1.5.16.2 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.

4.1.5.16.3 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

4.1.5.16.4 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)
```

4.1.5.17 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。

4.1.5.17.1 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_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()

4.1.5.17.2 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。
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 (0.01 sec)

tidb> explain select * from t where a < 2 and a > 2;
```

```
+--
    ↪ ------------------+----------+------+-------------------------------------------------------------+
    ↪
| id                  | count    | task | operator info
    ↪                                                |
+--
    ↪ ------------------+----------+------+-------------------------------------------------------------+
    ↪
| TableReader_7       | 0.00     | root | data:Selection_6
    ↪                                             |
| └─Selection_6       | 0.00     | cop  | gt(test.t.a, 2), lt(test.t.a, 2)
    ↪                             |
|   └─TableScan_5     | 10000.00 | cop  | table:t, range:[-inf,+inf], keep order:false, stats:
    ↪ pseudo |
+--
    ↪ ------------------+----------+------+-------------------------------------------------------------+
    ↪
3 rows in set (0.00 sec)

tidb> insert into mysql.expr_pushdown_blacklist values('<'), ('>');
Query OK, 2 rows affected (0.00 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                  | count    | task | operator info
    ↪                                                |
+--
    ↪ ------------------+----------+------+-------------------------------------------------------------+
    ↪
| Selection_5         | 8000.00  | root | gt(test.t.a, 2), lt(test.t.a, 2)
    ↪                             |
| └─TableReader_7     | 10000.00 | root | data:TableScan_6
    ↪                                             |
|   └─TableScan_6     | 10000.00 | cop  | table:t, range:[-inf,+inf], 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 (0.00 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                    | count    | task | operator info
    ↪                                                 |
+--
    ↪ --------------------+----------+------+-------------------------------------------------------------+
    ↪
| Selection_5           | 2666.67  | root | lt(test.t.a, 2)
    ↪                                                |
| └─TableReader_8       | 3333.33  | root | data:Selection_7
    ↪                                               |
|   └─Selection_7       | 3333.33  | cop  | gt(test.t.a, 2)
    ↪                                                |
|     └─TableScan_6     | 10000.00 | cop  | table:t, range:[-inf,+inf], 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

运算符原始名称	运算符别名
>=	ge
=	eq
!=	ne
<>	ne
<=>	nulleq
|	bitor
&&	bitand
||	or
!	not
in	in
+	plus
-	minus
	mul
/	div
DIV	intdiv
IS NULL	isnull
IS TRUE	istrue
IS FALSE	isfalse

4.1.6 SQL 语句

4.1.6.1 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。

4.1.6.1.1 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'

ColumnKeywordOpt ::=
    'COLUMN'?

ColumnDef ::=
    ColumnName ( Type | 'SERIAL' ) ColumnOptionListOpt

ColumnPosition ::=
    ( 'FIRST' | 'AFTER' ColumnName )?
```

4.1.6.1.2 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)
```

4.1.6.1.3 MySQL 兼容性

- 不支持同时添加多列。
- 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`。

4.1.6.1.4 另请参阅

- ADD INDEX
- CREATE TABLE

4.1.6.2 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。

4.1.6.2.1 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'

ColumnKeywordOpt ::=
    'COLUMN'?
```

```
ColumnDef ::=
    ColumnName ( Type | 'SERIAL' ) ColumnOptionListOpt

ColumnPosition ::=
    ( 'FIRST' | 'AFTER' ColumnName )?
```

4.1.6.2.2 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                  | count    | task | operator info
    ↪                                                |
+---------------------+----------+------+---------------------------------------------------------------+
    ↪
| TableReader_7       | 10.00    | root | data:Selection_6
    ↪                                                |
| └─Selection_6       | 10.00    | cop  | eq(test.t1.c1, 3)
    ↪                                               |
|   └─TableScan_5     | 10000.00 | cop  | table:t1, range:[-inf,+inf], 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               | count | task | operator info
    ↪                                                  |
+------------------+-------+------+-------------------------------------------------------------------+
    ↪
| IndexReader_6    | 10.00 | root | index:IndexScan_5
    ↪                                                 |
| └─IndexScan_5    | 10.00 | cop  | table:t1, index:c1, range:[3,3], keep order:false, stats:
    ↪ pseudo |
+------------------+-------+------+-------------------------------------------------------------------+
    ↪
2 rows in set (0.00 sec)
```

4.1.6.2.3 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）。
- 目前尚不支持同时添加多个索引。
- 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，详情参考：alter-primary-key。

4.1.6.2.4 另请参阅

- CREATE INDEX
- DROP INDEX
- RENAME INDEX
- ADD COLUMN
- CREATE TABLE
- EXPLAIN

4.1.6.3 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。示例如下。

```
ADMIN SHOW DDL;
```

ADMIN SHOW DDL 用于查看当前正在执行的 DDL 作业。

```
ADMIN SHOW DDL JOBS;
```

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] ...;
```

ADMIN SHOW DDL JOB QUERIES job_id [, job_id] ... 用于查看 job_id 对应的 DDL 任务的原始 SQL 语句。这个 job_id 只会搜索正在运行中的 DDL 作业以及 DDL 历史作业队列中最近的十条结果。

```
ADMIN CANCEL DDL JOBS job_id [, job_id] ...;
```

ADMIN CANCEL DDL JOBS job_id [, job_id] ... 用于取消当前正在运行的 job_id 的 DDL 作业，并返回对应作业是否取消成功。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 ID。
> - 如果希望取消的作业已经完成，则取消操作将会失败。

```
ADMIN CHECK TABLE tbl_name [, tbl_name] ...;
```

ADMIN CHECK TABLE tbl_name [, tbl_name] ... 用于对表 tbl_name 中的所有数据和对应索引进行一致性校验。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。

4.1.6.3.1 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' )
```

4.1.6.3.2 使用示例

```
admin show ddl jobs;
```

```
+--------+---------+------------+---------------+----------------------+-----------+----------+-----------+------
    ↪
| JOB_ID | DB_NAME | TABLE_NAME | JOB_TYPE      | SCHEMA_STATE         | SCHEMA_ID | TABLE_ID |
    ↪ ROW_COUNT | START_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  | rollback done |
| 43     | test    | t1          | add index      | public               | 32        | 37       | 6
    ↪          | 2019-01-10 12:35:13.66 +0800 CST  | synced        |
| 42     | test    | t1          | drop index     | none                 | 32        | 37       | 0
    ↪          | 2019-01-10 12:34:35.204 +0800 CST | synced        |
| 41     | test    | t1          | add index      | public               | 32        | 37       | 0
    ↪          | 2019-01-10 12:33:22.62 +0800 CST  | synced        |
| 40     | test    | t1          | drop column    | none                 | 32        | 37       | 0
    ↪          | 2019-01-10 12:33:08.212 +0800 CST | synced        |
| 39     | test    | t1          | add column     | public               | 32        | 37       | 0
    ↪          | 2019-01-10 12:32:55.42 +0800 CST  | synced        |
| 38     | test    | t1          | create table   | public               | 32        | 37       | 0
    ↪          | 2019-01-10 12:32:41.956 +0800 CST | synced        |
| 36     | test    |             | drop table     | none                 | 32        | 34       | 0
    ↪          | 2019-01-10 11:29:59.982 +0800 CST | synced        |
| 35     | test    |             | create table   | public               | 32        | 34       | 0
    ↪          | 2019-01-10 11:29:40.741 +0800 CST | synced        |
| 33     | test    |             | create schema  | public               | 32        | 0        | 0
    ↪          | 2019-01-10 11:29:22.813 +0800 CST | synced        |
+--------+---------+-------------+----------------+----------------------+-----------+----------+-----------+------
    ↪
```

- 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，在 Online, Asynchronous Schema Change in F1 论文中有详细说明，在此不再赘述。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。
- 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。

4.1.6.3.3 MySQL 兼容性

ADMIN 语句是 TiDB 对于 MySQL 语法的扩展。

4.1.6.4 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。

4.1.6.4.1 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 )*
```

4.1.6.4.2 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。

4.1.6.4.3 MySQL 兼容性

ADMIN CANCEL DDL 语句是 TiDB 对 MySQL 语法的扩展。

4.1.6.4.4 另请参阅

- ADMIN SHOW DDL [JOBS|QUERIES]

4.1.6.5 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。

4.1.6.5.1 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 )*
```

4.1.6.5.2 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)
```

4.1.6.5.3 MySQL 兼容性

`ADMIN CHECKSUM TABLE` 语句是 TiDB 对 MySQL 语法的扩展。

4.1.6.6 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。

4.1.6.6.1 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 )*
```

4.1.6.6.2 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。

4.1.6.6.3 MySQL 兼容性

ADMIN CHECK [TABLE|INDEX] 语句是 TiDB 对 MySQL 语法的扩展。

4.1.6.6.4 另请参阅

- ADMIN

4.1.6.7 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。

4.1.6.7.1 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?
```

4.1.6.7.2 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。

4.1.6.7.3 MySQL 兼容性

ADMIN SHOW DDL [JOBS|QUERIES] 语句是 TiDB 对 MySQL 语法的扩展。

4.1.6.7.4 另请参阅

- ADMIN CANCEL DDL

4.1.6.8 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。

4.1.6.8.1 示例

```
ALTER {DATABASE | SCHEMA} [db_name]
    alter_specification ...
alter_specification:
    [DEFAULT] CHARACTER SET [=] charset_name
  | [DEFAULT] COLLATE [=] collation_name
```

alter_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。

4.1.6.8.2 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。

4.1.6.8.3 另请参阅

- CREATE DATABASE
- SHOW DATABASES

4.1.6.9 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。

4.1.6.9.1 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。

4.1.6.9.2 语法图

```
AlterInstanceStmt ::=
    'ALTER' 'INSTANCE' InstanceOption
```

4.1.6.9.3 示例

```
ALTER INSTANCE RELOAD TLS;
```

4.1.6.9.4 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。

4.1.6.9.5 另请参阅

MySQL 客户端开启 TLS

4.1.6.10 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

4.1.6.10.1 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'
```

4.1.6.10.2 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                 | count    | task | operator info
    ↪                                                  |
+--------------------+----------+------+-------------------------------------------------------------------+
    ↪
| TableReader_7      | 10.00    | root | data:Selection_6
    ↪                                                 |
| └─Selection_6      | 10.00    | cop  | eq(test.t1.c1, 3)
    ↪                                                |
|   └─TableScan_5    | 10000.00 | cop  | table:t1, range:[-inf,+inf], 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               | count | task | operator info
    ↪                                                 |
+------------------+-------+------+-----------------------------------------------------------------+
    ↪
| IndexReader_6    | 10.00 | root | index:IndexScan_5
    ↪                                                |
| └─IndexScan_5    | 10.00 | cop  | table:t1, index:c1, range:[3,3], keep order:false, stats:
    ↪ pseudo |
+------------------+-------+------+-----------------------------------------------------------------+
    ↪
2 rows in set (0.00 sec)
```

4.1.6.10.3 MySQL 兼容性

- 支持除空间类型外的所有数据类型。
- 不支持 FULLTEXT，HASH 和 SPATIAL 索引。

4.1.6.10.4 另请参阅

- ADD COLUMN
- DROP COLUMN
- ADD INDEX
- DROP INDEX
- RENAME INDEX
- CREATE TABLE
- DROP TABLE
- SHOW CREATE TABLE

4.1.6.11 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。

4.1.6.11.1 语法图

```
AlterUserStmt ::=
    'ALTER' 'USER' IfExists (UserSpecList RequireClauseOpt ConnectionOptions
        ↪ PasswordOrLockOptions | 'USER' '(' ')' 'IDENTIFIED' 'BY' AuthString)

UserSpecList ::=
    UserSpec ( ',' UserSpec )*

UserSpec ::=
    Username AuthOption
```

4.1.6.11.2 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)
```

4.1.6.11.3 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。

4.1.6.11.4 另请参阅

- Security Compatibility with MySQL
- CREATE USER
- DROP USER
- SHOW CREATE USER

4.1.6.12 ANALYZE TABLE

ANALYZE TABLE 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 TABLE。

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。

4.1.6.12.1 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 )?
```

4.1.6.12.2 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               | count | task | operator info
    ↪                                                  |
+------------------+-------+------+-------------------------------------------------------------------+
    ↪
| IndexReader_6    | 10.00 | root | index:IndexScan_5
    ↪                                                 |
| └─IndexScan_5    | 10.00 | cop  | table:t1, index: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                | count | task | operator info                                 |
+-------------------+-------+------+-----------------------------------------------+
| IndexReader_6     | 1.00  | root | index:IndexScan_5                             |
| └─IndexScan_5     | 1.00  | cop  | table:t1, index:c1, range:[3,3], keep order:false |
+-------------------+-------+------+-----------------------------------------------+
2 rows in set (0.00 sec)
```

4.1.6.12.3 MySQL 兼容性

ANALYZE TABLE 在语法上与 MySQL 类似。但 ANALYZE TABLE 在 TiDB 上执行所需时间可能长得多，因为它的内部运行方式不同。

4.1.6.12.4 另请参阅

- EXPLAIN
- EXPLAIN ANALYZE

4.1.6.13 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。

4.1.6.13.1 语法图

BeginTransactionStmt:

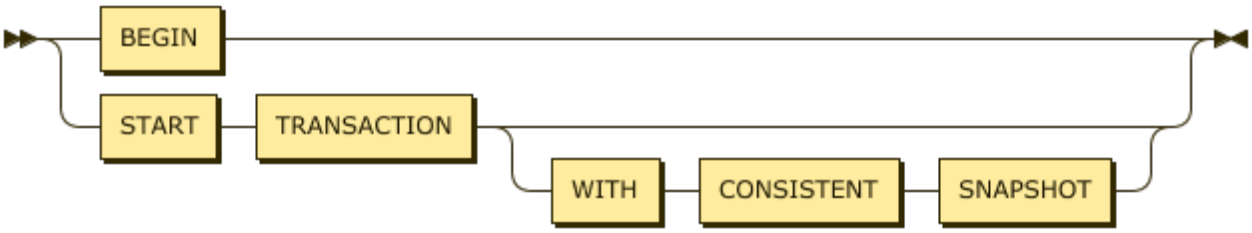


图 61: BeginTransactionStmt

4.1.6.13.2 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)
```

4.1.6.13.3 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。

4.1.6.13.4 另请参阅

- COMMIT
- ROLLBACK
- START TRANSACTION

4.1.6.14 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。

4.1.6.14.1 语法图

AlterTableStmt:

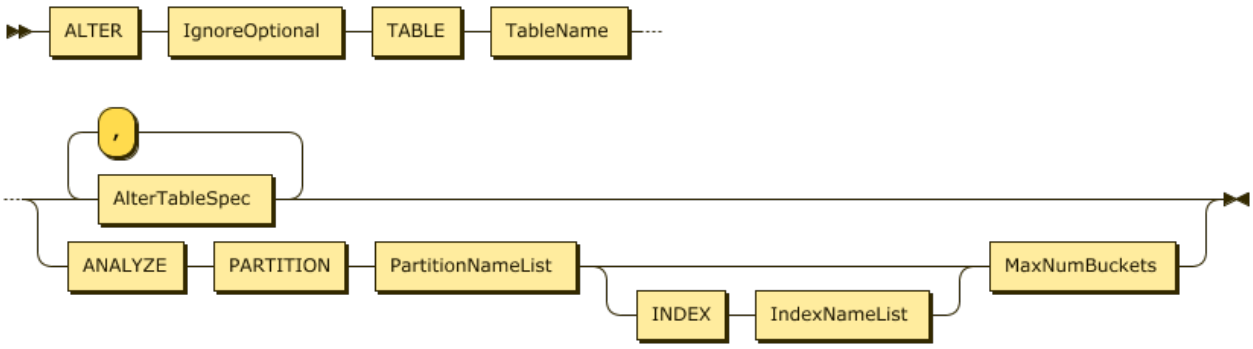


图 62: AlterTableStmt

AlterTableSpec:

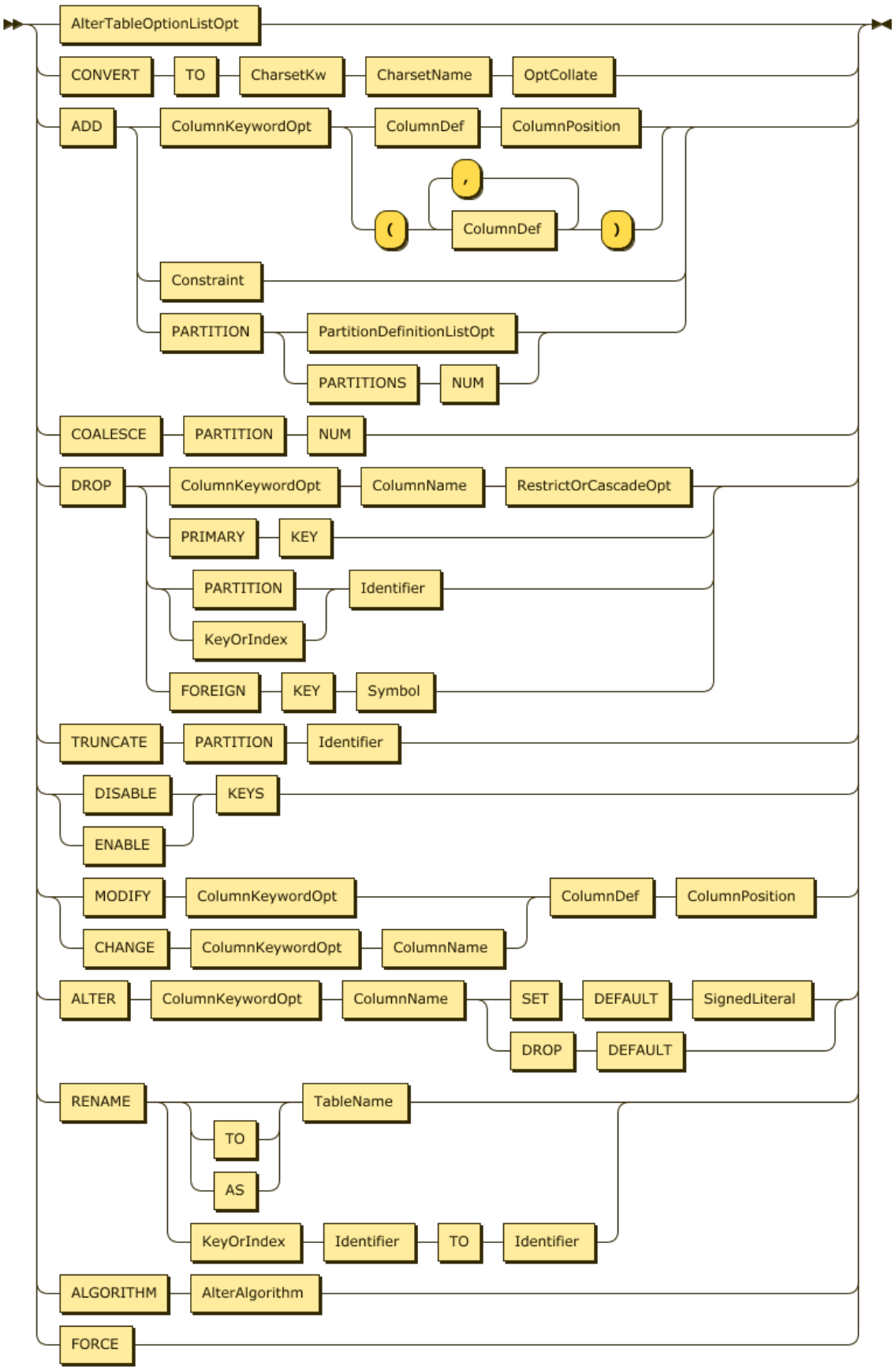


图 63: AlterTableSpec

ColumnKeywordOpt:

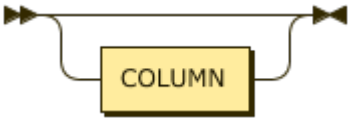


图 64: ColumnKeywordOpt

ColumnName:

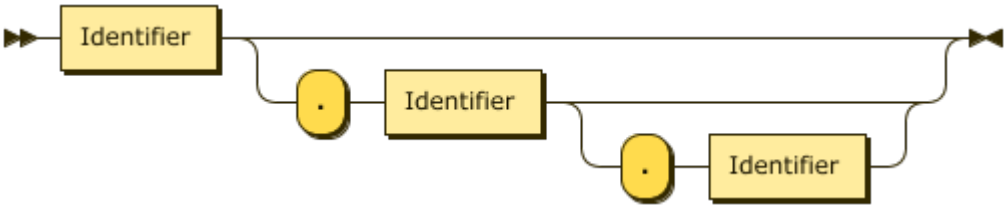


图 65: ColumnName

ColumnDef:

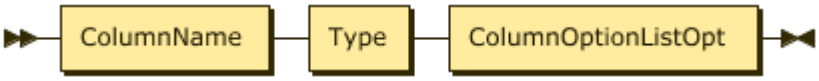


图 66: ColumnDef

ColumnPosition:

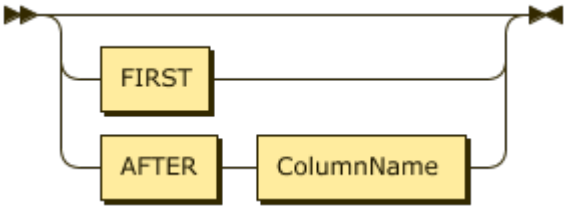


图 67: ColumnPosition

4.1.6.14.2 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
```

4.1.6.14.3 MySQL 兼容性

- 目前尚不支持在单个 ALTER TABLE 语句中进行多个更改。
- 仅支持特定的数据类型更改。例如，支持将 INTEGER 更改为 BIGINT，但不支持将 BIGINT 更改为 INTEGER。不支持从整数更改为字符串格式或 BLOB 类型。

4.1.6.14.4 另请参阅

- CREATE TABLE
- SHOW CREATE TABLE
- ADD COLUMN
- DROP COLUMN
- MODIFY COLUMN

4.1.6.15 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。

4.1.6.15.1 语法图

CommitStmt:

图 68: CommitStmt

4.1.6.15.2 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)
```

4.1.6.15.3 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被另一个事务修改因此可能导致 COMMIT 语句执行失败的情况。
- 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。

4.1.6.15.4 另请参阅

- START TRANSACTION
- ROLLBACK
- BEGIN
- 事务的惰性检查

4.1.6.16 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”。

4.1.6.16.1 语法图

CreateDatabaseStmt:

图 69: CreateDatabaseStmt

DatabaseSym:

图 70: DatabaseSym

IfNotExists:

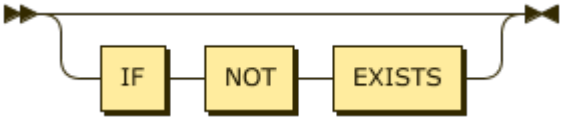


图 71: IfNotExists

DBName:

图 72: DBName

DatabaseOptionListOpt:

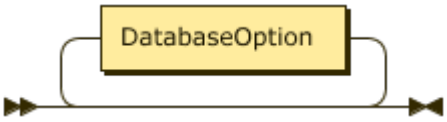


图 73: DatabaseOptionListOpt

4.1.6.16.2 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。

4.1.6.16.3 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)
```

4.1.6.16.4 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。

4.1.6.16.5 另请参阅

- USE
- ALTER DATABASE
- DROP DATABASE
- SHOW DATABASES

4.1.6.17 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。

4.1.6.17.1 语法图

CreateIndexStmt:

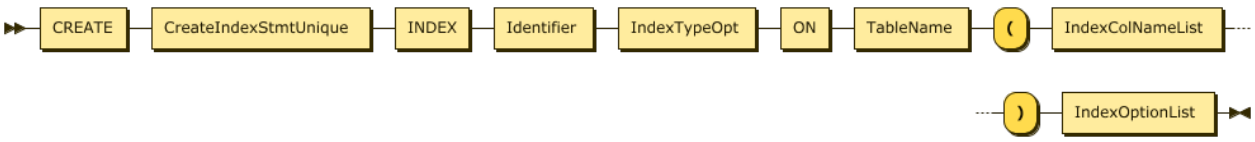


图 74: CreateIndexStmt

CreateIndexStmtUnique:

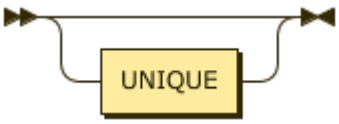


图 75: CreateIndexStmtUnique

Identifier:

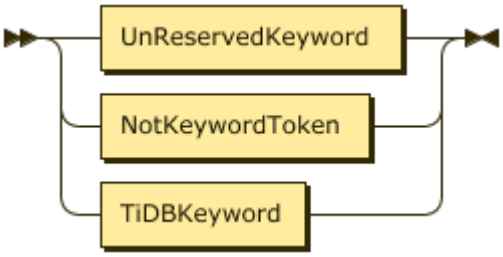


图 76: Identifier

IndexTypeOpt:

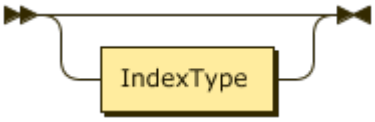


图 77: IndexTypeOpt

TableName:

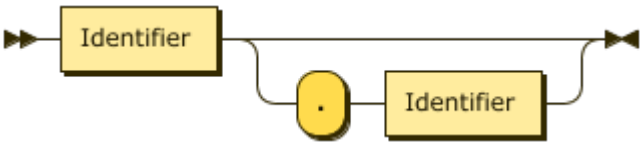


图 78: TableName

IndexColNameList:

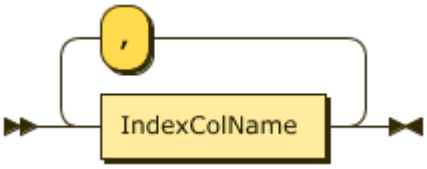


图 79: IndexColNameList

IndexOptionList:

图 80: IndexOptionList

IndexOption:

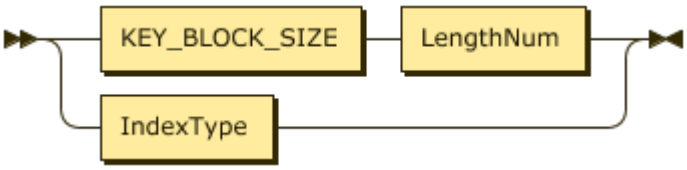


图 81: IndexOption

4.1.6.17.2 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                 | count    | task | operator info
    ↪                                                  |
+--------------------+----------+------+-------------------------------------------------------------------+
    ↪
| TableReader_7      | 10.00    | root | data:Selection_6
    ↪                                                 |
| └─Selection_6      | 10.00    | cop  | eq(test.t1.c1, 3)
    ↪                                                |
|   └─TableScan_5    | 10000.00 | cop  | table:t1, range:[-inf,+inf], 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               | count | task | operator info
    ↪                                                    |
+------------------+-------+------+-----------------------------------------------------------------------+
    ↪
| IndexReader_6    | 10.00 | root | index:IndexScan_5
    ↪                                                   |
| └─IndexScan_5    | 10.00 | cop  | table:t1, index: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)
```

4.1.6.17.3 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TiDB 特定系统变量。

4.1.6.17.4 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。

4.1.6.17.5 另请参阅

- ADD INDEX
- DROP INDEX
- RENAME INDEX
- ADD COLUMN
- CREATE TABLE
- EXPLAIN

4.1.6.18 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。

4.1.6.18.1 语法图

CreateRoleStmt:

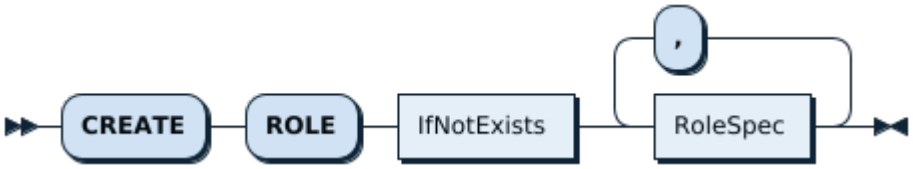


图 82: CreateRoleStmt

IfNotExists:

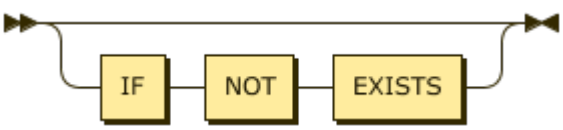


图 83: IfNotExists

RoleSpec:

图 84: RoleSpec

4.1.6.18.2 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)
```

4.1.6.18.3 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。

4.1.6.18.4 另请参阅

- DROP ROLE
- GRANT <role>
- REVOKE <role>
- SET ROLE
- SET DEFAULT ROLE
- 基于角色的访问控制

4.1.6.19 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。

4.1.6.19.1 语法图

CreateTableStmt:

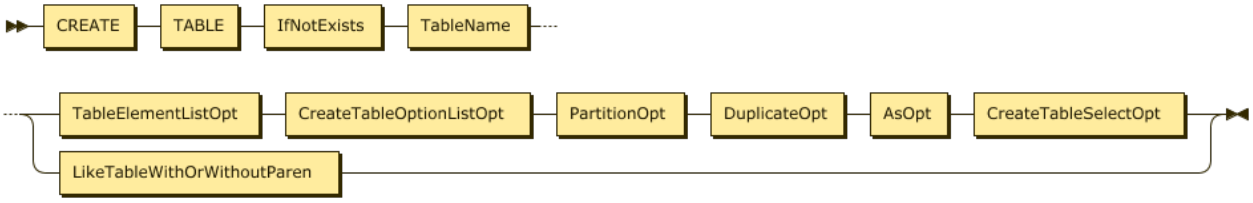


图 85: CreateTableStmt

LikeTableWithOrWithoutParen:

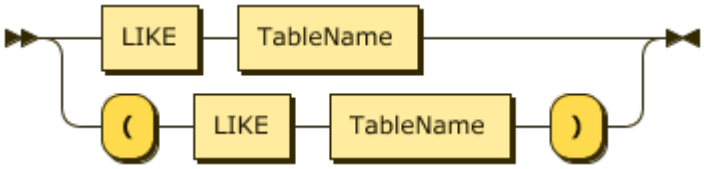


图 86: LikeTableWithOrWithoutParen

TableName:

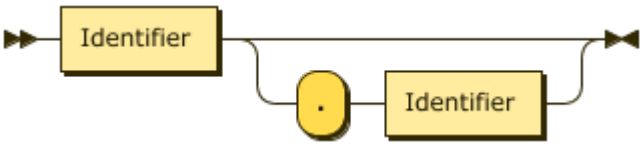


图 87: TableName

4.1.6.19.2 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)
```

4.1.6.19.3 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。

4.1.6.19.4 另请参阅

- CREATE TABLE
- SHOW CREATE TABLE

4.1.6.20 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。

4.1.6.20.1 语法图

CreateTableStmt:

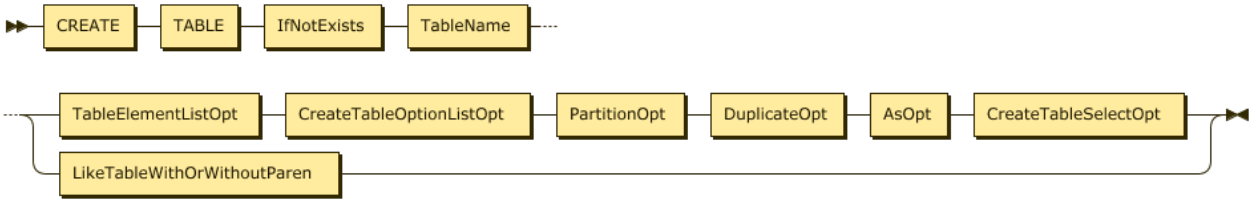


图 88: CreateTableStmt

IfNotExists:

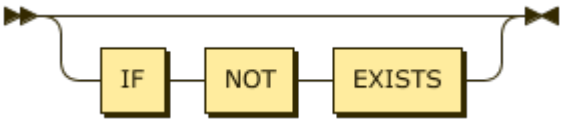


图 89: IfNotExists

TableName:

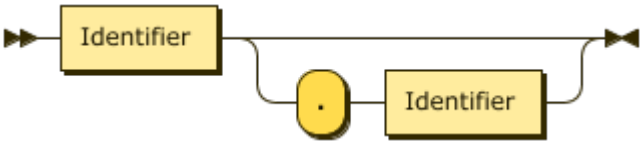


图 90: TableName

TableElementListOpt:

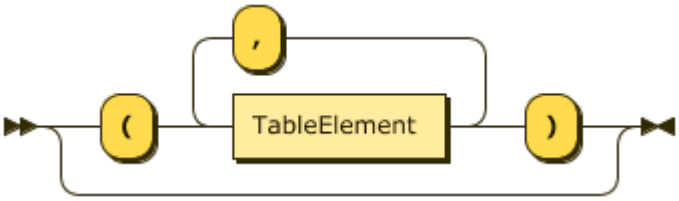


图 91: TableElementListOpt

TableElement:

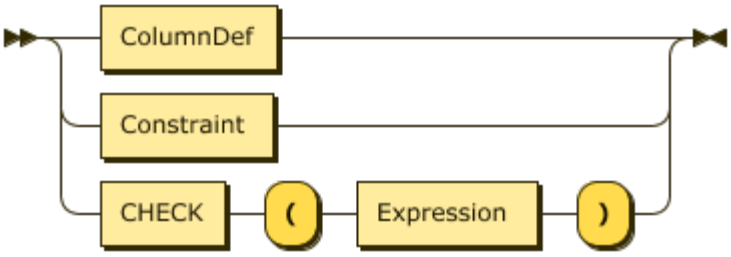


图 92: TableElement

PartitionOpt:

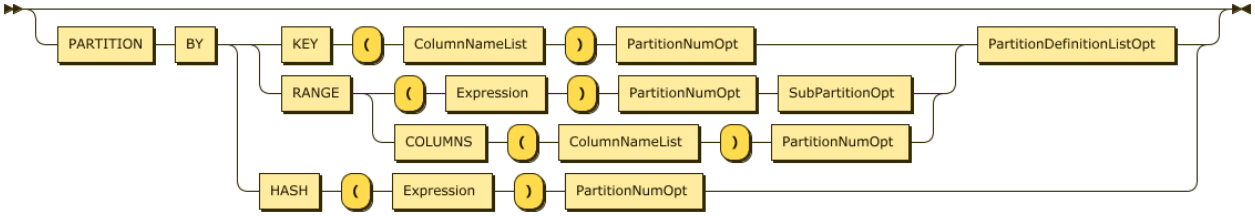


图 93: PartitionOpt

ColumnDef:

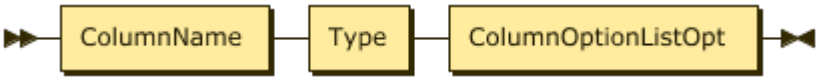


图 94: ColumnDef

ColumnName:

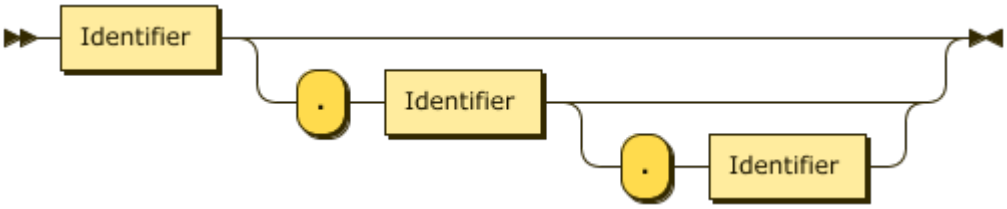


图 95: ColumnName

Type:

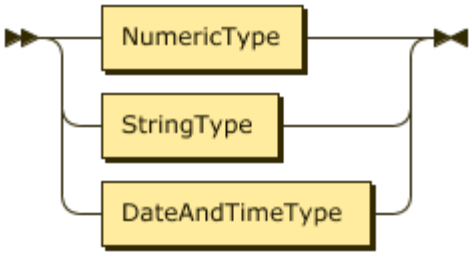


图 96: Type

ColumnOptionListOpt:

图 97: ColumnOptionListOpt

TableOptionListOpt:

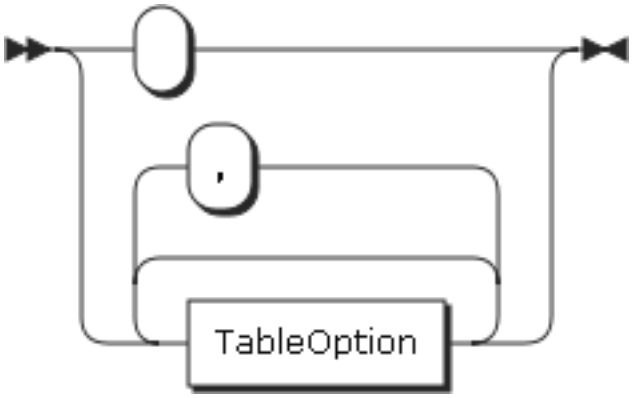


图 98: TableOptionListOpt

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
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
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’

split-table 配置项默认情况下会开启，在此配置项开启时，建表操作会为每个表建立单独的 Region。

4.1.6.20.2 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)
```

4.1.6.20.3 MySQL 兼容性

- TiDB 不支持 CREATE TEMPORARY TABLE 语法。
- 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。
- 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。

4.1.6.20.4 另请参阅

- 数据类型
- DROP TABLE
- CREATE TABLE LIKE
- SHOW CREATE TABLE

4.1.6.21 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。

4.1.6.21.1 语法图

CreateUserStmt:

图 99: CreateUserStmt

IfNotExists:

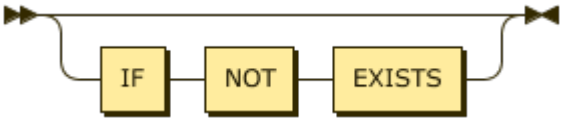


图 100: IfNotExists

UserSpecList:

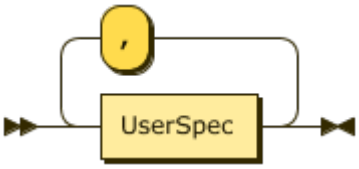


图 101: UserSpecList

UserSpec:

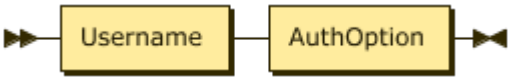


图 102: UserSpec

AuthOption:

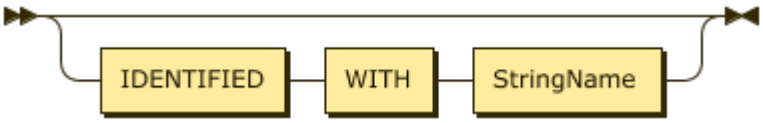


图 103: AuthOption

StringName:

图 104: StringName

4.1.6.21.2 示例

```
CREATE USER 'newuser' IDENTIFIED BY 'newuserpassword';
```

```
Query OK, 1 row affected (0.04 sec)
```

```
CREATE USER 'newuser2'@'192.168.1.1' IDENTIFIED BY 'newuserpassword';
```

```
Query OK, 1 row affected (0.02 sec)
```

4.1.6.21.3 MySQL 兼容性

- TiDB 尚不支持若干 CREATE 选项。这些选项可被解析，但会被忽略。

4.1.6.21.4 另请参阅

- Security Compatibility with MySQL
- DROP USER
- SHOW CREATE USER
- ALTER USER
- Privilege Management

4.1.6.22 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。

4.1.6.22.1 语法图

CreateViewStmt:

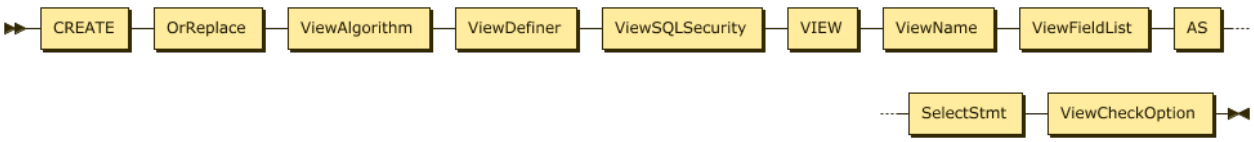


图 105: CreateViewStmt

OrReplace:

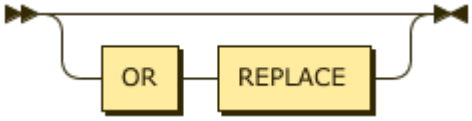


图 106: OrReplace

ViewAlgorithm:

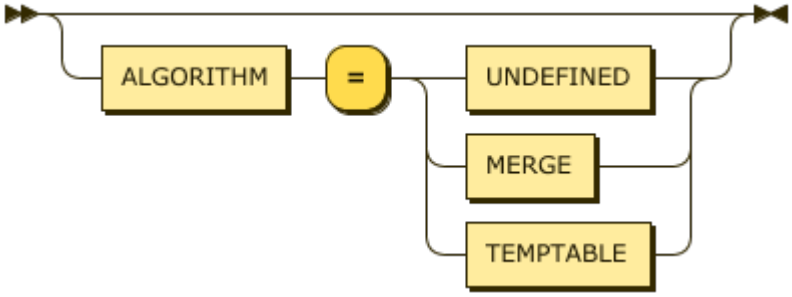


图 107: ViewAlgorithm

ViewDefiner:

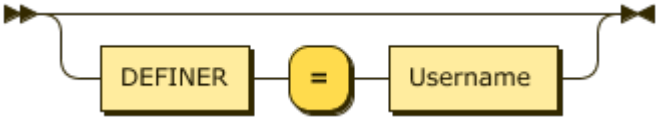


图 108: ViewDefiner

ViewSQLSecurity:

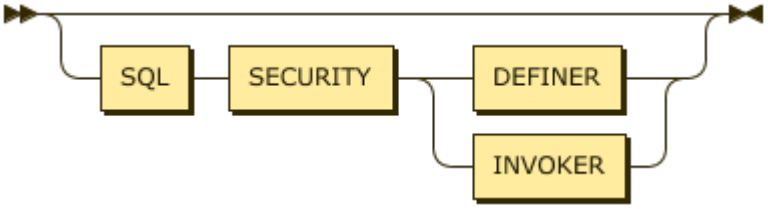


图 109: ViewSQLSecurity

ViewName:

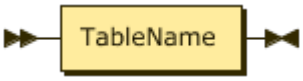


图 110: ViewName

ViewFieldList:

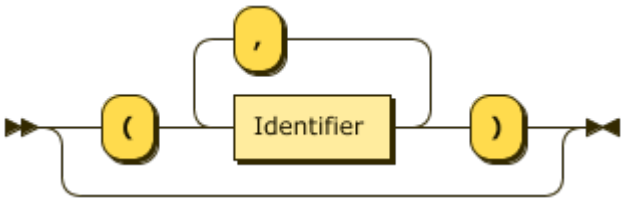


图 111: ViewFieldList

ViewCheckOption:

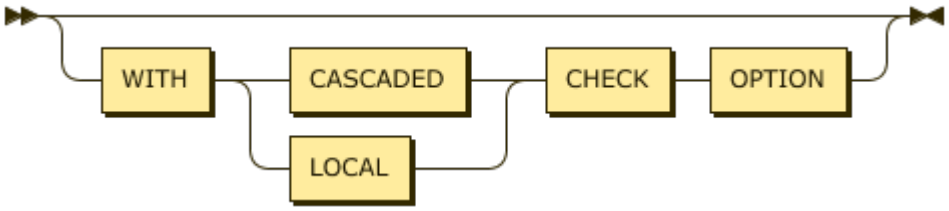


图 112: ViewCheckOption

4.1.6.22.2 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.
```

4.1.6.22.3 MySQL 兼容性

- 目前 TiDB 中的视图不可插入且不可更新。

4.1.6.22.4 另请参阅

- CREATE TABLE
- SHOW CREATE TABLE
- DROP TABLE

4.1.6.23 DEALLOCATE

DEALLOCATE 语句用于为服务器端预处理语句提供 SQL 接口。

4.1.6.23.1 语法图

DeallocateStmt:

图 113: DeallocateStmt

DeallocateSym:

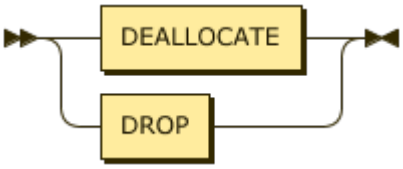


图 114: DeallocateSym

Identifier:

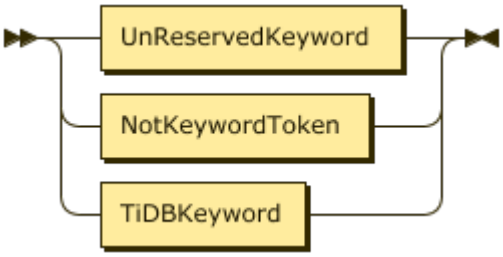


图 115: Identifier

4.1.6.23.2 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)
```

4.1.6.23.3 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。

4.1.6.23.4 另请参阅

- PREPARE
- EXECUTE

4.1.6.24 DELETE

DELETE 语句用于从指定的表中删除行。

4.1.6.24.1 语法图

DeleteFromStmt:

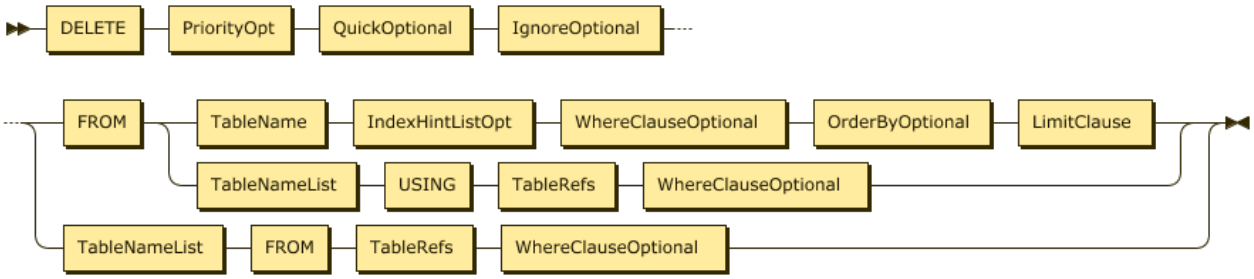


图 116: DeleteFromStmt

4.1.6.24.2 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)
```

4.1.6.24.3 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。

4.1.6.24.4 另请参阅

- INSERT
- SELECT
- UPDATE
- REPLACE

4.1.6.25 DESC

DESC 语句是EXPLAIN 的别名。包含该语句提供了 MySQL 兼容性。

4.1.6.26 DESCRIBE

DESCRIBE 语句为EXPLAIN 的别名。包含该语句提供了 MySQL 兼容性。

4.1.6.27 DO

DO 语句用于执行表达式，而不返回结果。在 MySQL 中，DO 的一个常见用例是执行存储的程序，而无需处理结果。但是 TiDB 不提供存储例程，因此该功能的使用较为受限。

4.1.6.27.1 语法图

DoStmt:

图 117: DoStmt

ExpressionList:

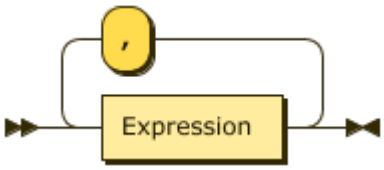


图 118: ExpressionList

Expression:

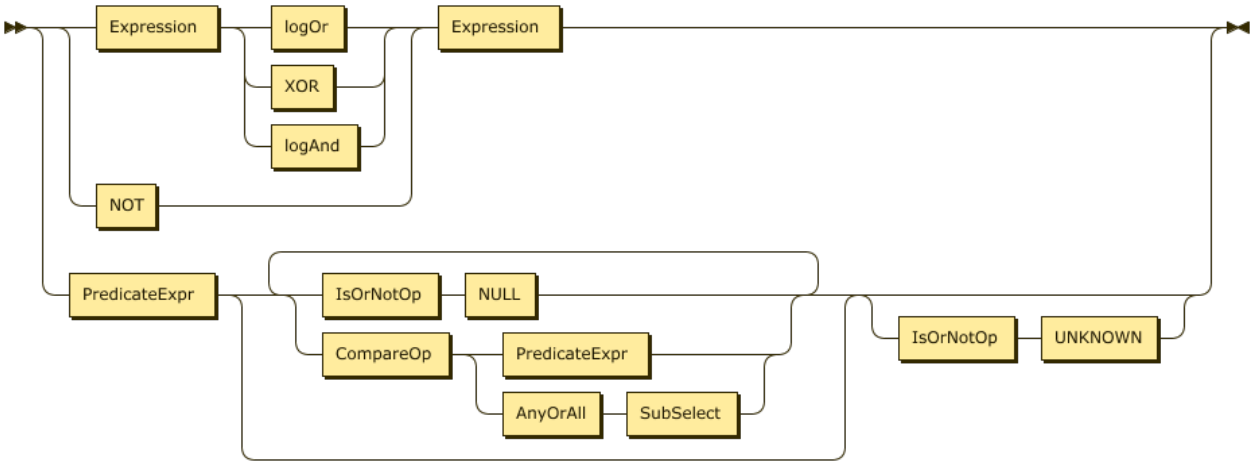


图 119: Expression

4.1.6.27.2 示例

```
SELECT SLEEP(5);
```

```
+----------+
| SLEEP(5) |
```

```
+----------+
|        0 |
+----------+
1 row in set (5.00 sec)
```

```
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)
```

4.1.6.27.3 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。

4.1.6.27.4 另请参阅

- SELECT

4.1.6.28 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。

4.1.6.28.1 语法图

AlterTableStmt:

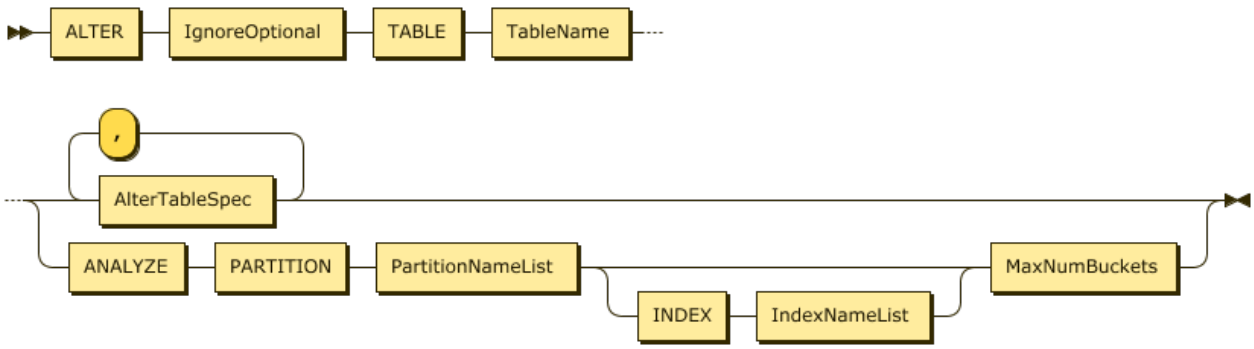


图 120: AlterTableStmt

AlterTableSpec:

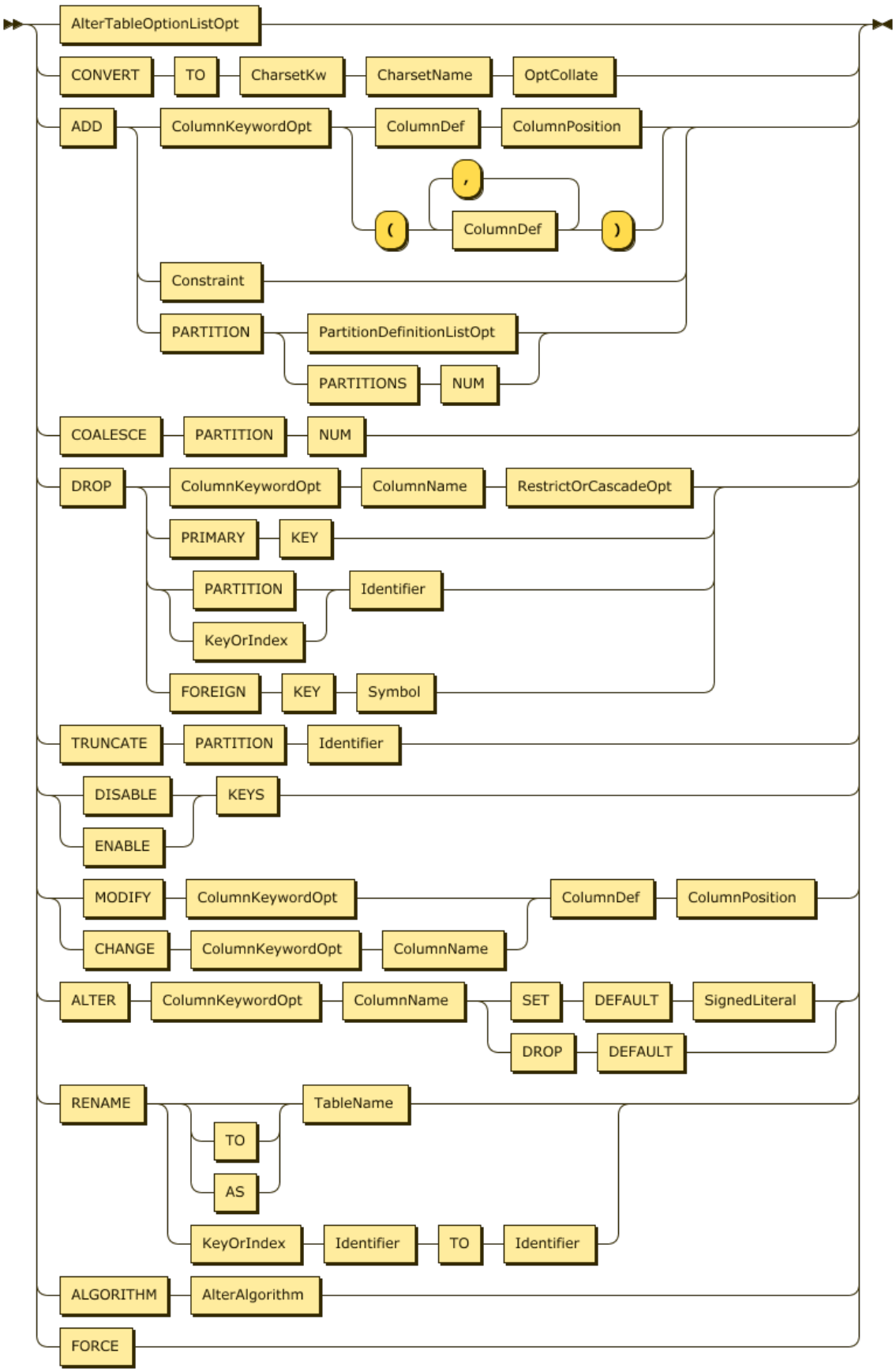


图 121: AlterTableSpec

ColumnKeywordOpt:

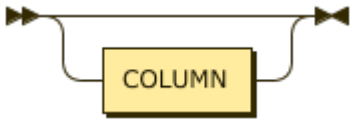


图 122: ColumnKeywordOpt

ColumnName:

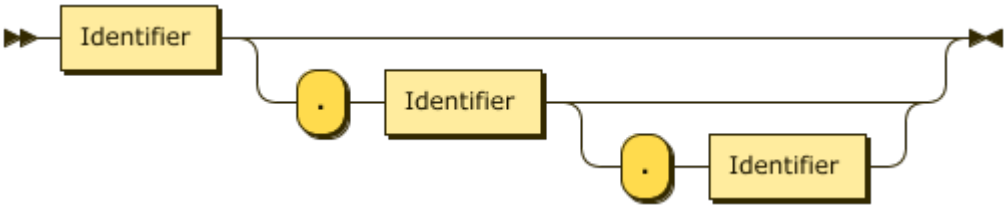


图 123: ColumnName

4.1.6.28.2 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)
```

4.1.6.28.3 MySQL 兼容性

- 不支持使用相同语句删除多个列。

4.1.6.28.4 另请参阅

- ADD COLUMN
- SHOW CREATE TABLE
- CREATE TABLE

4.1.6.29 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。

4.1.6.29.1 语法图

DropDatabaseStmt:

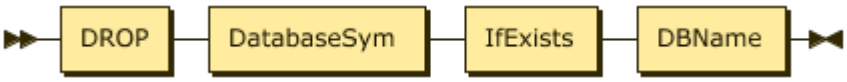


图 124: DropDatabaseStmt

DatabaseSym:

图 125: DatabaseSym

IfExists:

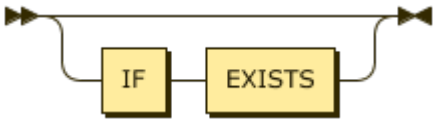


图 126: IfExists

DBName:

图 127: DBName

4.1.6.29.2 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)
```

4.1.6.29.3 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。

4.1.6.29.4 另请参阅

- CREATE DATABASE
- ALTER DATABASE

4.1.6.30 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。

4.1.6.30.1 语法图

AlterTableStmt:

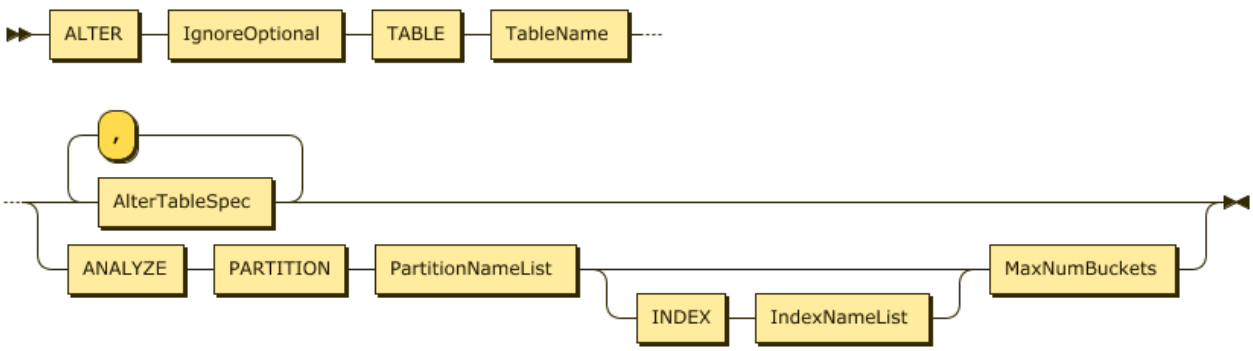


图 128: AlterTableStmt

AlterTableSpec:

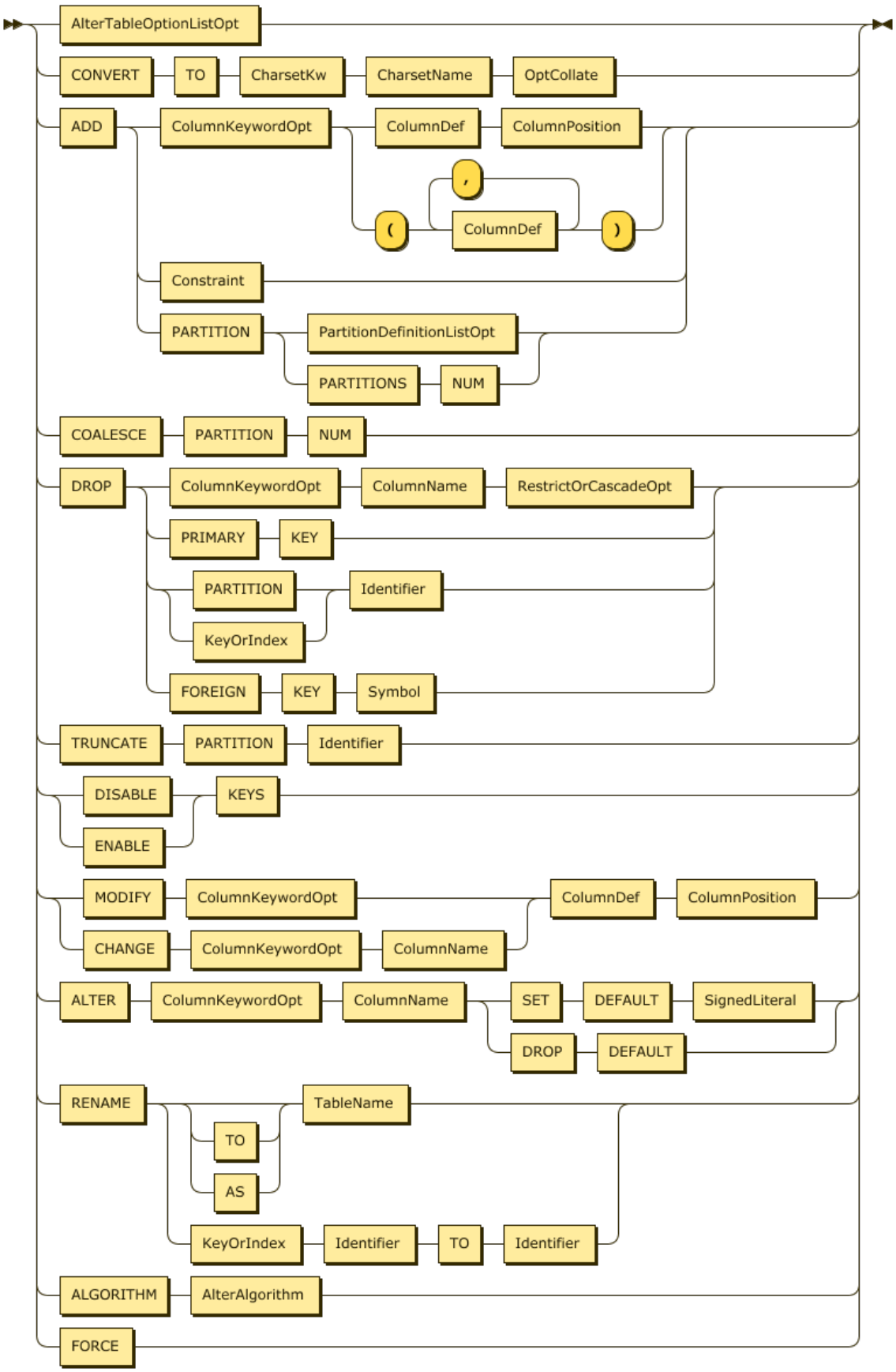


图 129: AlterTableSpec

KeyOrIndex:

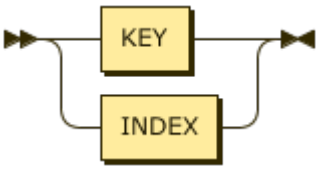


图 130: KeyOrIndex

Identifier:

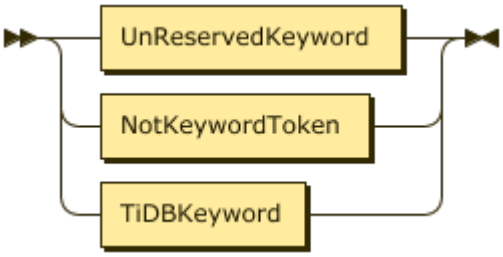


图 131: Identifier

4.1.6.30.2 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                  | count    | task | operator info
    ↪                                                  |
+---------------------+----------+------+-------------------------------------------------------------------+
    ↪
| TableReader_7       | 10.00    | root | data:Selection_6
    ↪                                                 |
| └─Selection_6       | 10.00    | cop  | eq(test.t1.c1, 3)
    ↪                                                |
|   └─TableScan_5     | 10000.00 | cop  | table:t1, range:[-inf,+inf], 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                | count | task | operator info
    ↪                                                 |
+-------------------+-------+------+-------------------------------------------------------------------+
    ↪
| IndexReader_6     | 10.00 | root | index:IndexScan_5
    ↪                                                |
| └─IndexScan_5     | 10.00 | cop  | table:t1, index: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)
```

4.1.6.30.3 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。

4.1.6.30.4 另请参阅

- SHOW INDEX
- CREATE INDEX
- ADD INDEX
- RENAME INDEX

4.1.6.31 DROP ROLE

使用 DROP ROLE 语句可删除已用 CREATE ROLE 语句创建的角色。

4.1.6.31.1 语法图

DropRoleStmt:

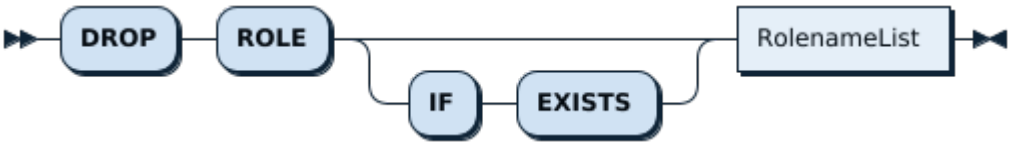


图 132: DropRoleStmt

RolenameList:

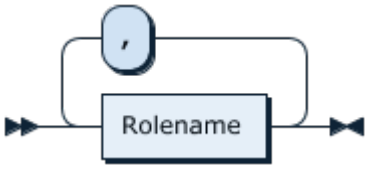


图 133: RolenameList

4.1.6.31.2 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@%
```

4.1.6.31.3 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。

4.1.6.31.4 另请参阅

- CREATE ROLE
- GRANT <role>
- REVOKE <role>
- SET ROLE
- SET DEFAULT ROLE
- 基于角色的访问控制

4.1.6.32 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。

按照设计，`DROP TABLE` 也会删除视图，因为视图与表共享相同的命名空间。

4.1.6.32.1 语法图

DropTableStmt:

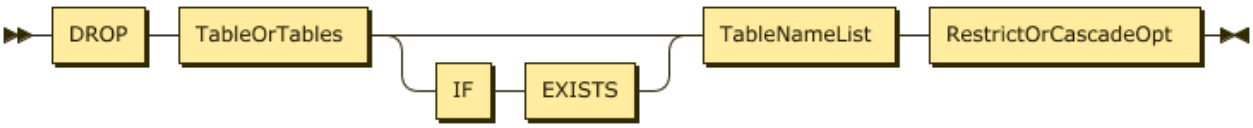


图 134: DropTableStmt

TableOrTables:

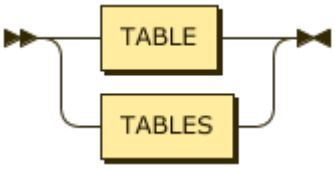


图 135: TableOrTables

TableNameList:

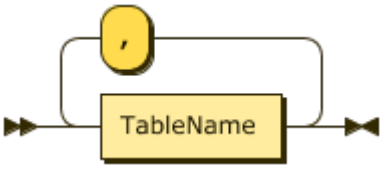


图 136: TableNameList

4.1.6.32.2 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)
```

4.1.6.32.3 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

4.1.6.32.4 另请参阅

- DROP VIEW
- CREATE TABLE
- SHOW CREATE TABLE
- SHOW TABLES

4.1.6.33 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。

4.1.6.33.1 语法图

DropUserStmt:

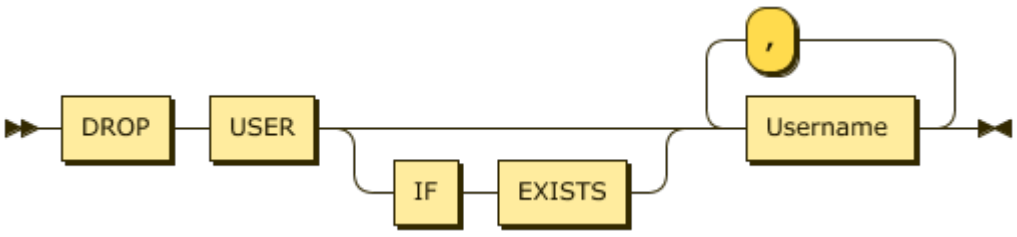


图 137: DropUserStmt

Username:

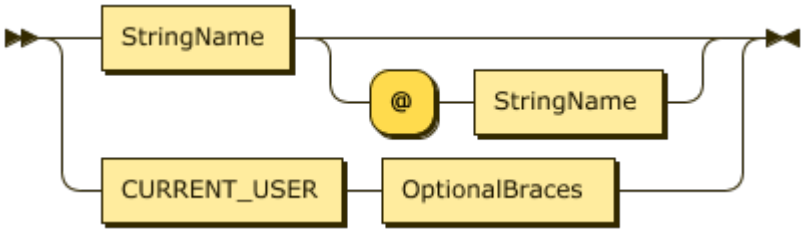


图 138: Username

4.1.6.33.2 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 '%'
```

4.1.6.33.3 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。

4.1.6.33.4 另请参阅

- CREATE USER
- ALTER USER
- SHOW CREATE USER
- Privilege Management

4.1.6.34 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。

4.1.6.34.1 语法图

DropViewStmt:

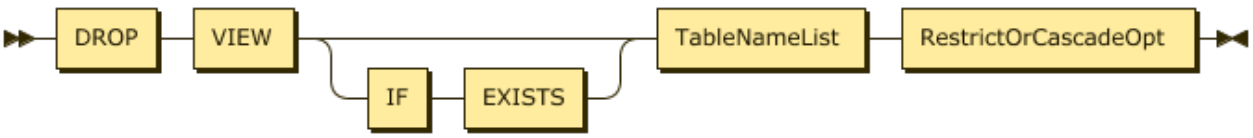


图 139: DropViewStmt

TableNameList:

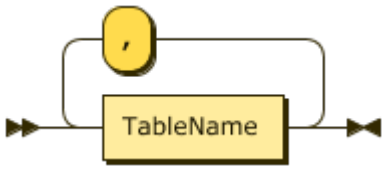


图 140: TableNameList

TableName:

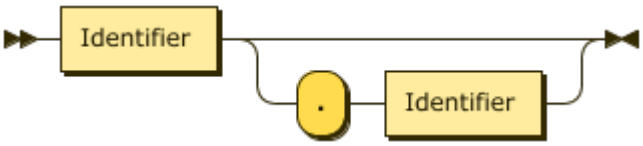


图 141: TableName

4.1.6.34.2 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)
```

4.1.6.34.3 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。

4.1.6.34.4 See also

- DROP TABLE
- CREATE VIEW

4.1.6.35 EXECUTE

`EXECUTE` 语句为服务器端预处理语句提供 SQL 接口。

4.1.6.35.1 语法图

ExecuteStmt:

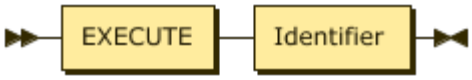


图 142: ExecuteStmt

Identifier:

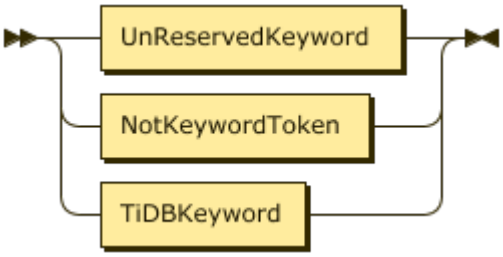


图 143: Identifier

4.1.6.35.2 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)
```

4.1.6.35.3 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。

4.1.6.35.4 另请参阅

- PREPARE
- DEALLOCATE

4.1.6.36 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`。

4.1.6.36.1 语法图

ExplainSym:

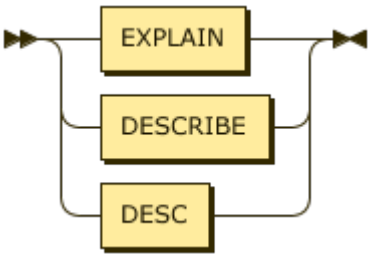


图 144: ExplainSym

ExplainStmt:

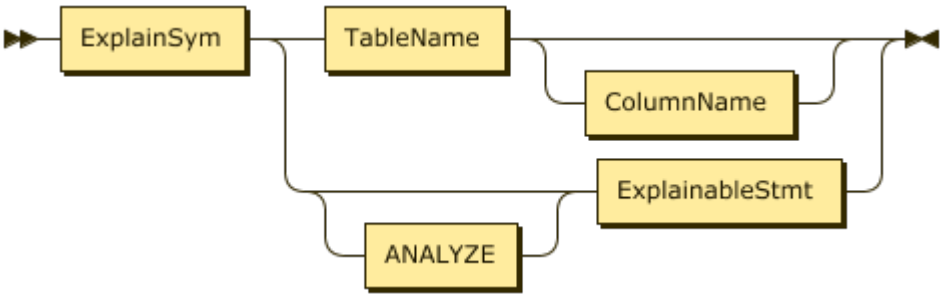


图 145: ExplainStmt

ExplainableStmt:

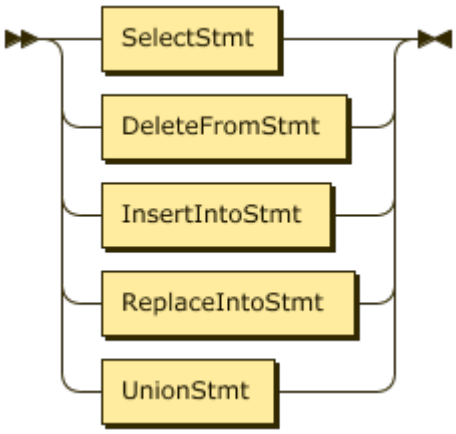


图 146: ExplainableStmt

4.1.6.36.2 示例

```
CREATE TABLE t1 (id INT NOT NULL PRIMARY KEY AUTO_INCREMENT, c1 INT NOT NULL);
```

```
Query OK, 0 rows affected (0.12 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          | count | task | operator info     | execution info            |
+-------------+-------+------+-------------------+---------------------------+
| Point_Get_1 | 1.00  | root | table:t1, handle:1 | time:0ns, loops:0, rows:0 |
+-------------+-------+------+-------------------+---------------------------+
1 row in set (0.01 sec)
```

```
EXPLAIN ANALYZE SELECT * FROM t1;
```

```
+------------------+----------+------+-------------------------------------------------------------------+------------
    ↪
| id               | count    | task | operator info
    ↪                                                | execution info                  |
+------------------+----------+------+-------------------------------------------------------------------+------------
    ↪
| TableReader_5    | 10000.00 | root | data:TableScan_4
    ↪                                                | time:931.759µs, loops:2, rows:3  |
```

```
| └─TableScan_4     | 10000.00 | cop  | table:t1, range:[-inf,+inf], keep order:false, stats:
    ↪ pseudo | time:0s, loops:0, rows:3         |
+-------------------+----------+------+-------------------------------------------------------------------+-----------
    ↪
2 rows in set (0.00 sec)
```

4.1.6.36.3 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。

4.1.6.36.4 另请参阅

- Understanding the Query Execution Plan
- EXPLAIN
- ANALYZE TABLE
- TRACE

4.1.6.37 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。

4.1.6.37.1 语法图

ExplainSym:

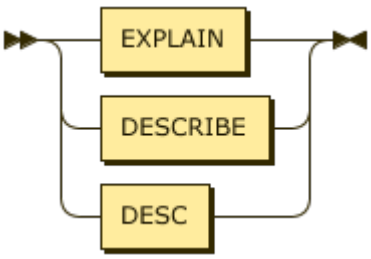


图 147: ExplainSym

ExplainStmt:

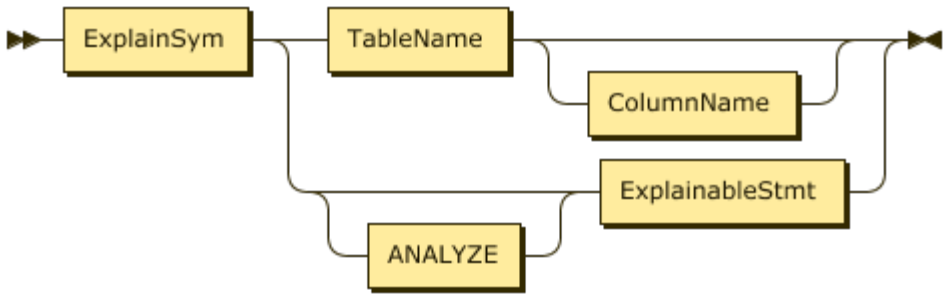


图 148: ExplainStmt

ExplainableStmt:

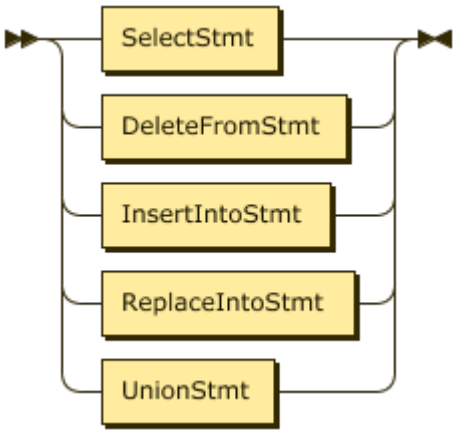


图 149: ExplainableStmt

4.1.6.37.2 示例

```
EXPLAIN SELECT 1;
```

```
+------------------+-------+------+---------------+
| id               | count | task | operator info |
+------------------+-------+------+---------------+
| Projection_3     | 1.00  | root | 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          | count | task | 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          | count | task | 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          | count | task | 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);
```

```
ERROR 1105 (HY000): Unsupported type *core.Insert
```

```
EXPLAIN UPDATE t1 SET c1=5 WHERE c1=3;
```

```
+---------------------+----------+------+-------------------------------------------------------------------+
    ↪
| id                  | count    | task | operator info
    ↪                                                  |
+---------------------+----------+------+-------------------------------------------------------------------+
    ↪
| TableReader_6       | 10.00    | root | data:Selection_5
    ↪                                                 |
```

```
| └─Selection_5        | 10.00    | cop  | eq(test.t1.c1, 3)
    ↪                                              |
|   └─TableScan_4      | 10000.00 | cop  | table:t1, range:[-inf,+inf], keep order:false, stats:
    ↪ pseudo |
+---------------------+----------+------+----------------------------------------------------------------+
    ↪
3 rows in set (0.00 sec)
```

```
EXPLAIN DELETE FROM t1 WHERE c1=3;
```

```
+---------------------+----------+------+----------------------------------------------------------------+
    ↪
| id                  | count    | task | operator info
    ↪                                                  |
+---------------------+----------+------+----------------------------------------------------------------+
    ↪
| TableReader_6       | 10.00    | root | data:Selection_5
    ↪                                               |
| └─Selection_5        | 10.00    | cop  | eq(test.t1.c1, 3)
    ↪                                              |
|   └─TableScan_4      | 10000.00 | cop  | table:t1, range:[-inf,+inf], keep order:false, stats:
    ↪ pseudo |
+---------------------+----------+------+----------------------------------------------------------------+
    ↪
3 rows in set (0.00 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 HashRightJoin_7 {
subgraph cluster7{
node [style=filled, color=lightgrey]
color=black
label = "root"
"HashRightJoin_7" -> "TableReader_10"
```

```
"HashRightJoin_7" -> "TableReader_12"
}
subgraph cluster9{
node [style=filled, color=lightgrey]
color=black
label = "cop"
"Selection_9" -> "TableScan_8"
}
subgraph cluster11{
node [style=filled, color=lightgrey]
color=black
label = "cop"
"TableScan_11"
}
"TableReader_10" -> "Selection_9"
"TableReader_12" -> "TableScan_11"
}
 |
+-------------------------------------------------------------------------------------------------------------------------------------------------------------------------
    ↪
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：

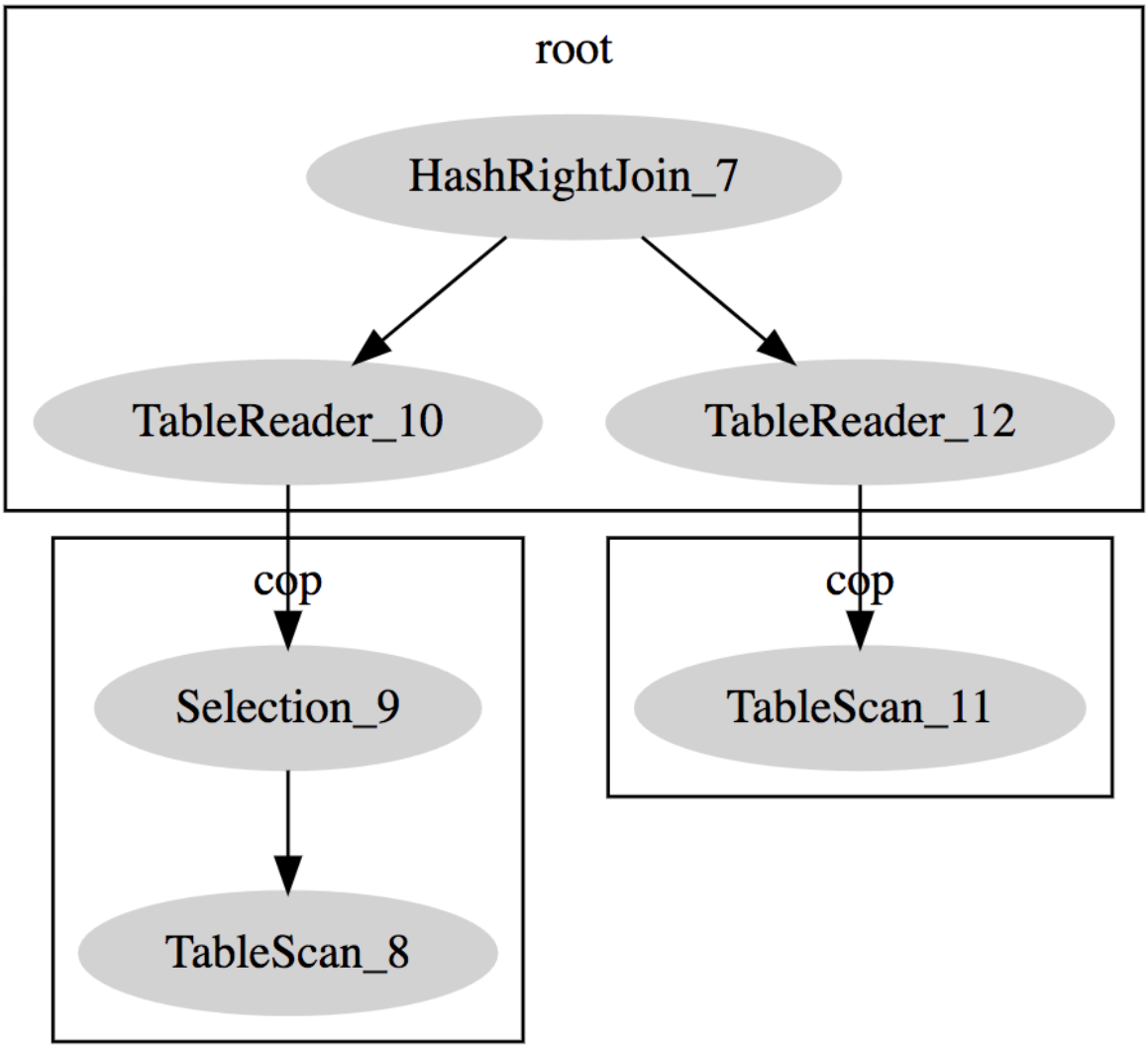


图 150: Explain Dot

4.1.6.37.3 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。

4.1.6.37.4 另请参阅

- Understanding the Query Execution Plan
- EXPLAIN ANALYZE
- ANALYZE TABLE

- TRACE

4.1.6.38 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。

4.1.6.38.1 语法图

FlushStmt:

图 151: FlushStmt

NoWriteToBinLogAliasOpt:

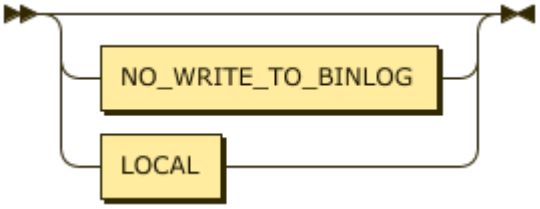


图 152: NoWriteToBinLogAliasOpt

FlushOption:

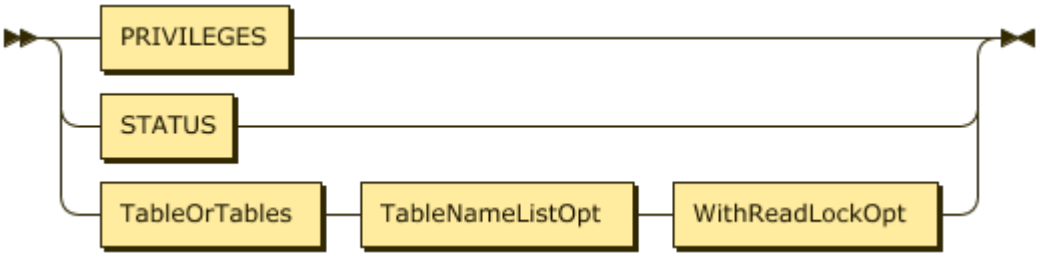


图 153: FlushOption

4.1.6.38.2 示例

```
FLUSH PRIVILEGES;
```

```
Query OK, 0 rows affected (0.01 sec)
```

4.1.6.38.3 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。

4.1.6.38.4 另请参阅

- `GRANT <privileges>`
- `REVOKE`
- Privilege Management

4.1.6.39 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。

4.1.6.39.1 语法图

FlushStmt:

图 154: FlushStmt

NoWriteToBinLogAliasOpt:

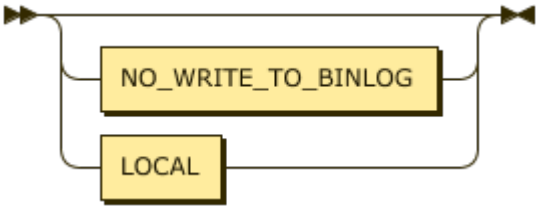


图 155: NoWriteToBinLogAliasOpt

FlushOption:

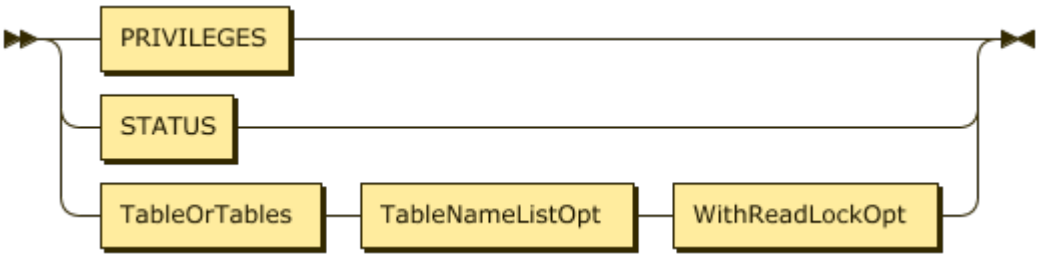


图 156: FlushOption

4.1.6.39.2 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)
```

4.1.6.39.3 MySQL 兼容性

- FLUSH STATUS 语句仅用于提供 MySQL 兼容性。

4.1.6.39.4 另请参阅

- SHOW [GLOBAL|SESSION] STATUS

4.1.6.40 FLUSH TABLES

FLUSH TABLES 语句用于提供 MySQL 兼容性，但在 TiDB 中并无有效用途。

4.1.6.40.1 语法图

FlushStmt:

图 157: FlushStmt

NoWriteToBinLogAliasOpt:

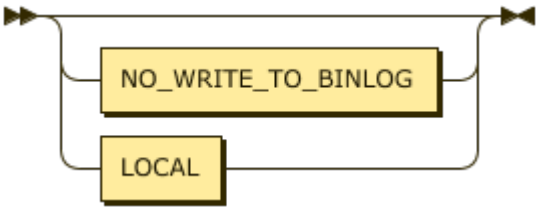


图 158: NoWriteToBinLogAliasOpt

FlushOption:

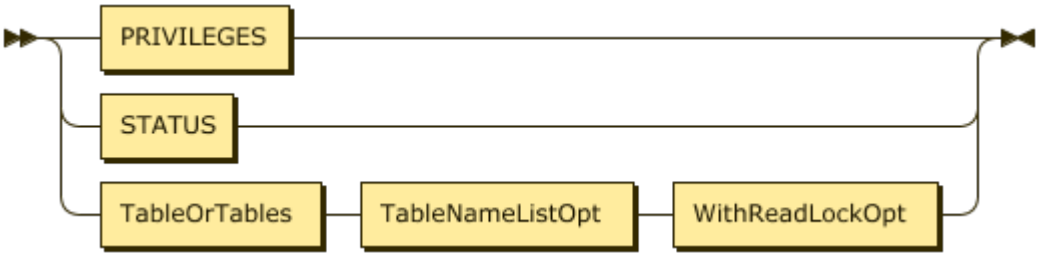


图 159: FlushOption

TableOrTables:

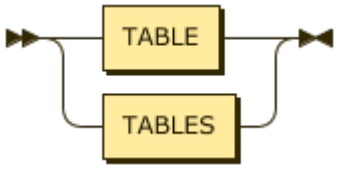


图 160: TableOrTables

TableNameListOpt:

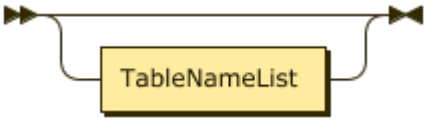


图 161: TableNameListOpt

WithReadLockOpt:

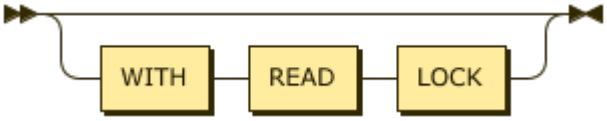


图 162: WithReadLockOpt

4.1.6.40.2 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
```

4.1.6.40.3 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。

4.1.6.40.4 另请参阅

- Read historical data

4.1.6.41 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。

4.1.6.41.1 语法图

GrantStmt:

图 163: GrantStmt

PrivElemList:

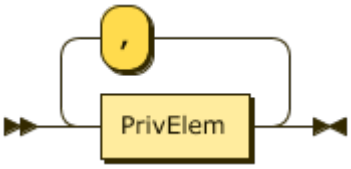


图 164: PrivElemList

PrivElem:

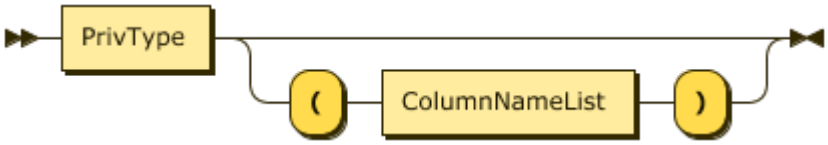


图 165: PrivElem

PrivType:

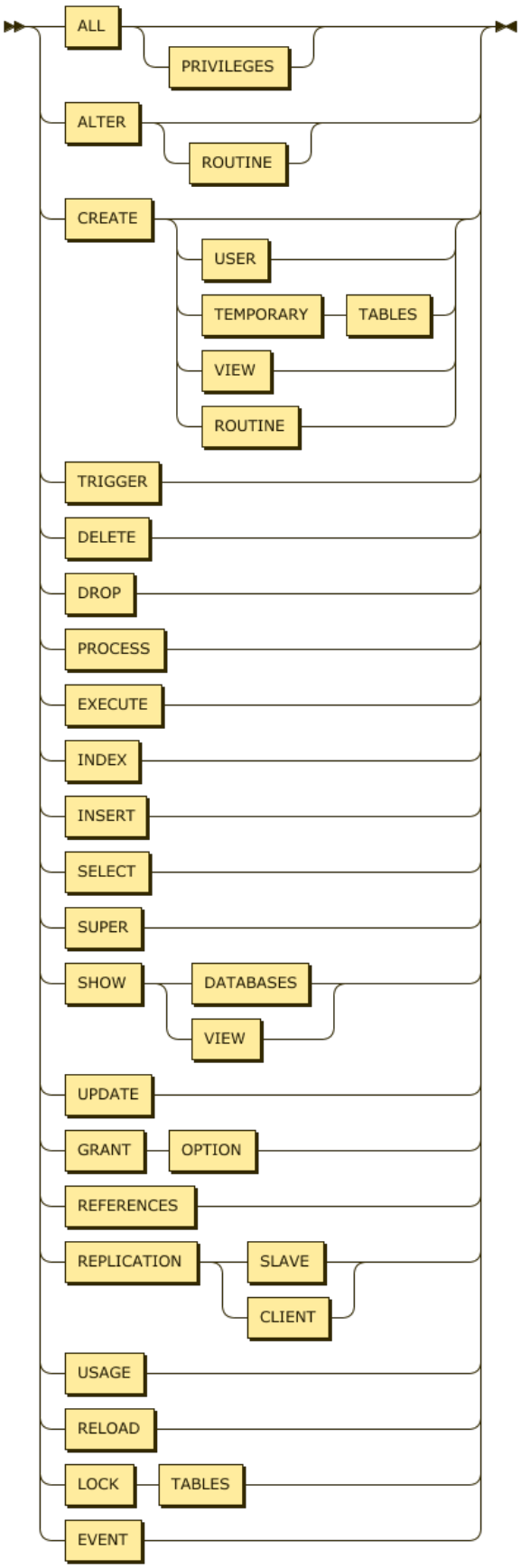


图 166: PrivType

ObjectType:

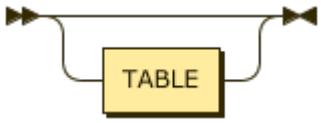


图 167: ObjectType

PrivLevel:

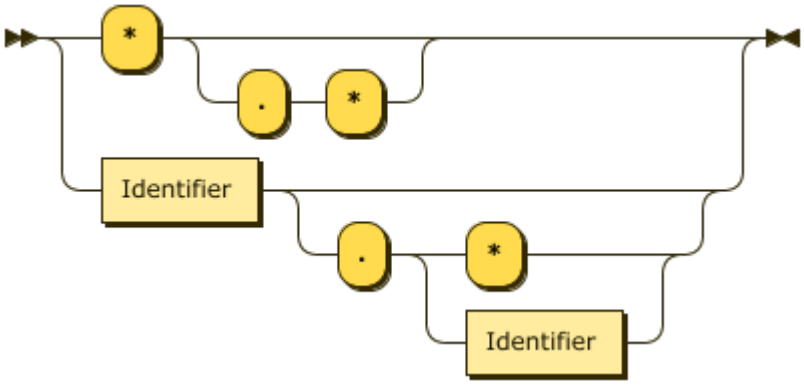


图 168: PrivLevel

UserSpecList:

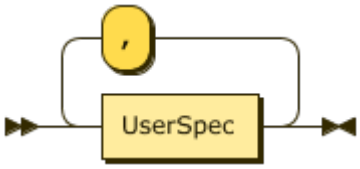


图 169: UserSpecList

4.1.6.41.2 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)
```

4.1.6.41.3 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。

4.1.6.41.4 另请参阅

- GRANT <role>
- REVOKE <privileges>
- SHOW GRANTS
- 权限管理

4.1.6.42 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。

4.1.6.42.1 语法图

GrantRoleStmt:

图 170: GrantRoleStmt

RolenameList:

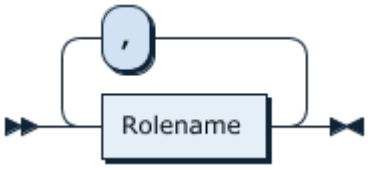


图 171: RolenameList

UsernameList:

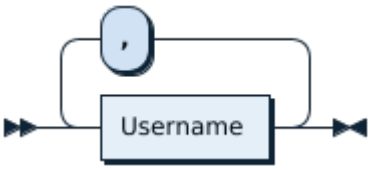


图 172: UsernameList

4.1.6.42.2 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)
```

4.1.6.42.3 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。

4.1.6.42.4 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

4.1.6.43 INSERT

使用 INSERT 语句在表中插入新行。

4.1.6.43.1 语法图

InsertIntoStmt:

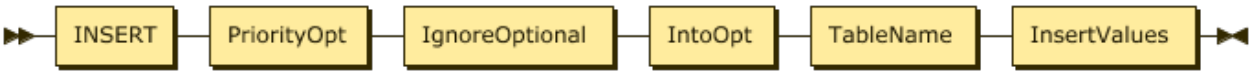


图 173: InsertIntoStmt

PriorityOpt:

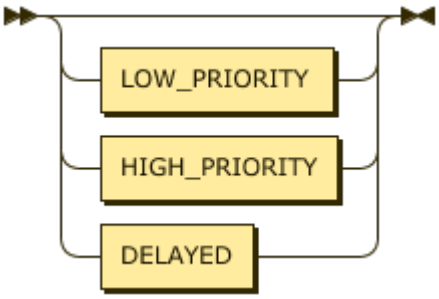


图 174: PriorityOpt

IgnoreOptional:

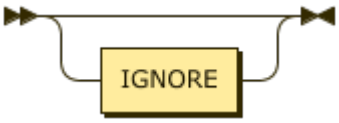


图 175: IgnoreOptional

IntoOpt:

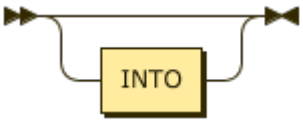


图 176: IntoOpt

TableName:

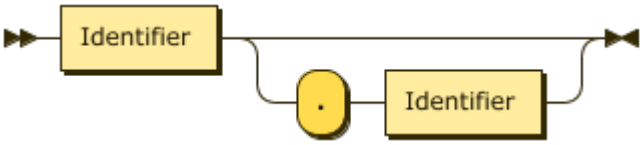


图 177: TableName

InsertValues:

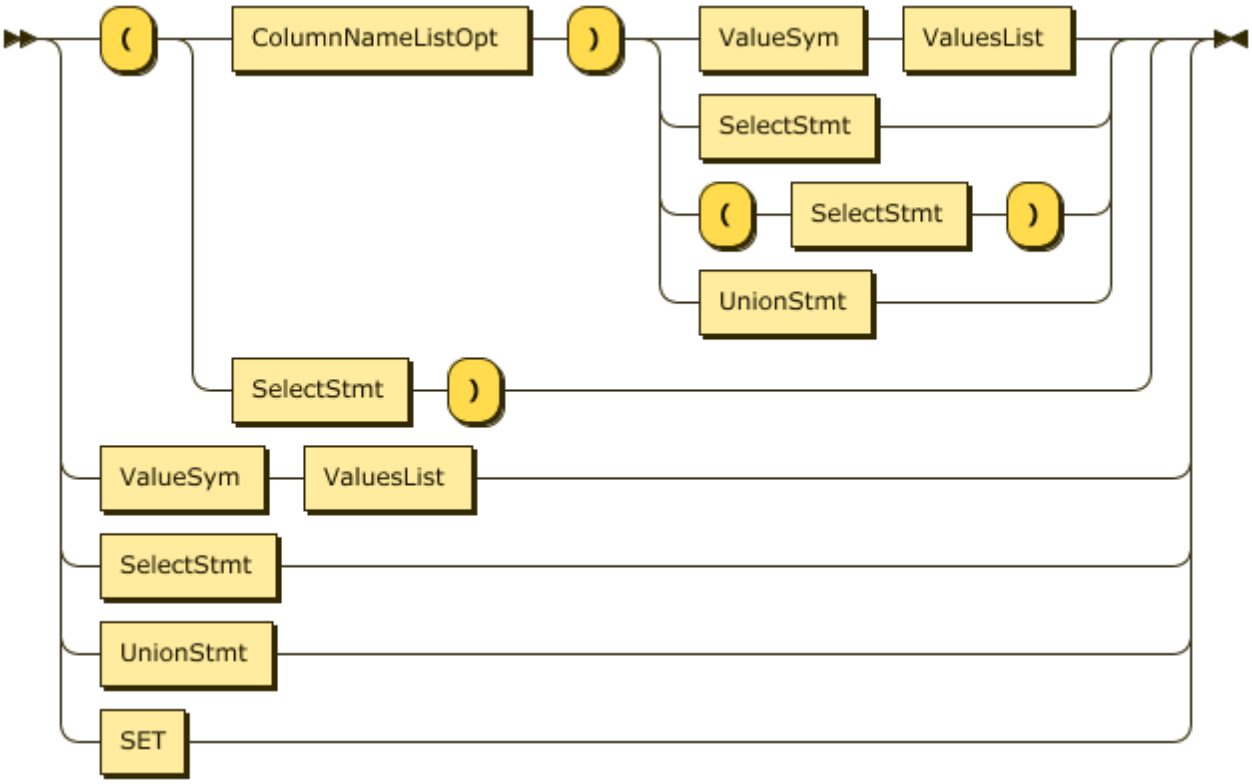


图 178: InsertValues

4.1.6.43.2 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)
```

4.1.6.43.3 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。

4.1.6.43.4 另请参阅

- DELETE
- SELECT
- UPDATE
- REPLACE

4.1.6.44 KILL TIDB

`KILL TIDB` 语句用于终止 TiDB 中的连接。

4.1.6.44.1 语法图

KillStmt:

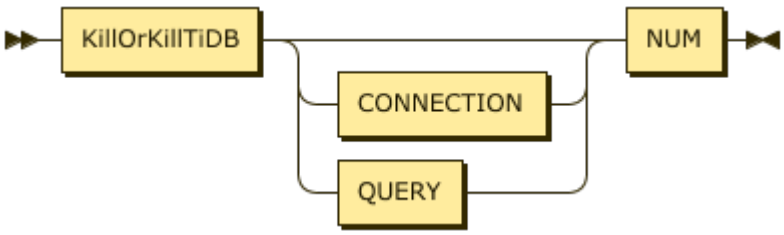


图 179: KillStmt

4.1.6.44.2 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)
```

4.1.6.44.3 MySQL 兼容性

- 按照设计，`KILL TIDB` 语句默认与 MySQL 不兼容。负载均衡器后方通常有多个 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`。

4.1.6.44.4 另请参阅

- `SHOW [FULL] PROCESSLIST`

4.1.6.45 LOAD DATA

`LOAD DATA` 语句用于将数据批量加载到 TiDB 表中。

4.1.6.45.1 语法图

LoadDataStmt:

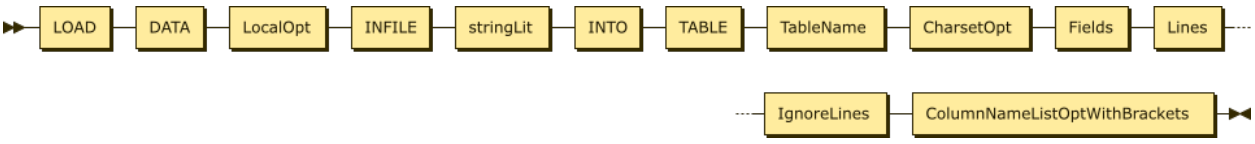


图 180: LoadDataStmt

4.1.6.45.2 参数说明

用户可以使用 `FIELDS` 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。

4.1.6.45.3 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
```

4.1.6.45.4 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。

4.1.6.45.5 另请参阅

- INSERT

4.1.6.46 LOAD STATS

`LOAD STATS` 语句用于将统计信息加载到 TiDB 中。

4.1.6.46.1 语法图

LoadStatsStmt:

图 181: LoadStatsStmt

4.1.6.46.2 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)
```

4.1.6.46.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

4.1.6.46.4 另请参阅

- 统计信息

4.1.6.47 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。

4.1.6.47.1 语法图

AlterTableStmt:

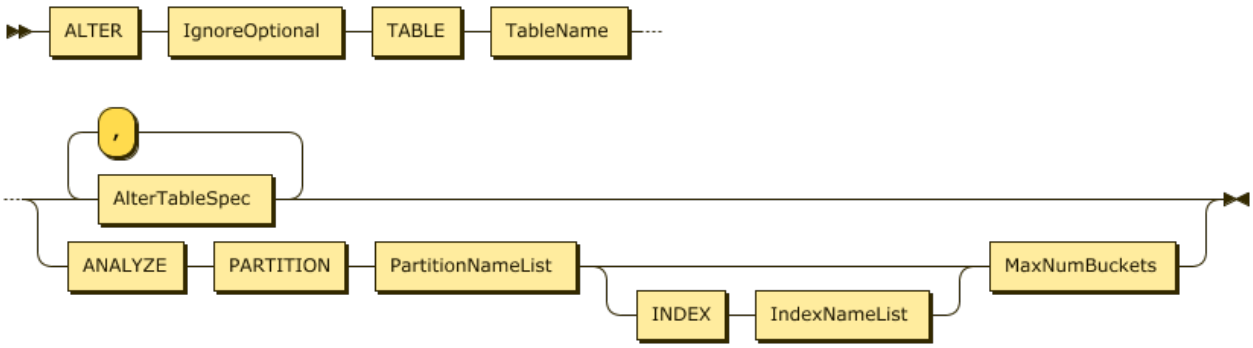


图 182: AlterTableStmt

AlterTableSpec:

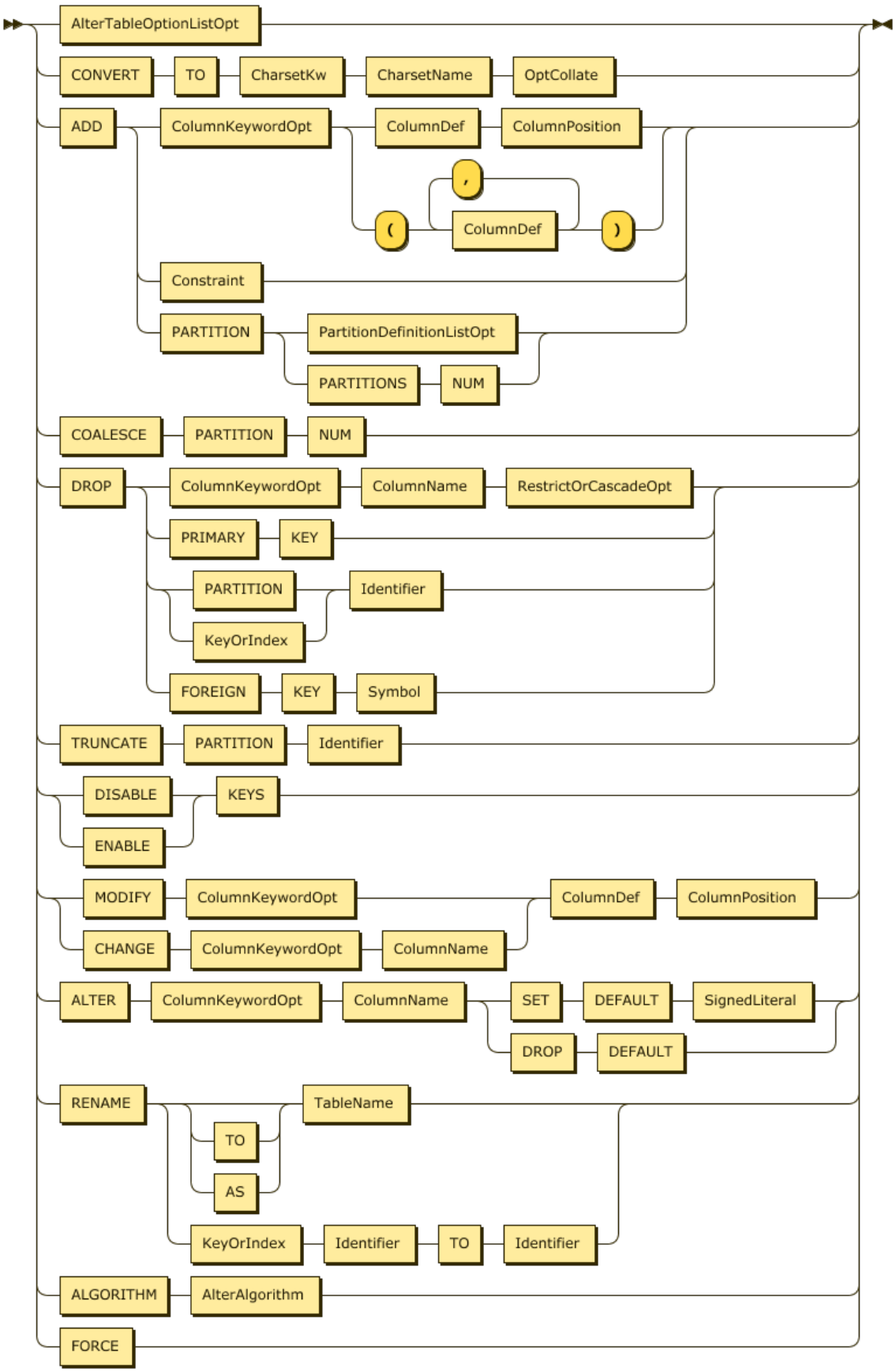


图 183: AlterTableSpec

ColumnKeywordOpt:

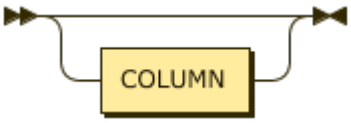


图 184: ColumnKeywordOpt

ColumnDef:

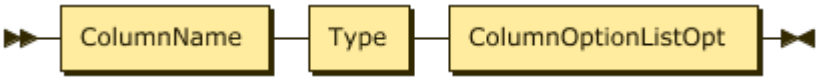


图 185: ColumnDef

ColumnPosition:

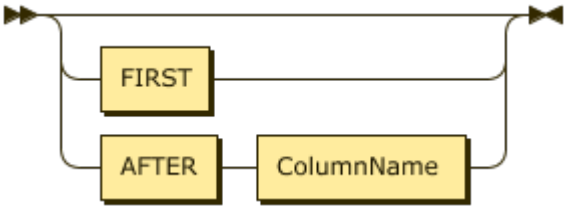


图 186: ColumnPosition

4.1.6.47.2 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;
```

```
*************************** 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)
```

```
ALTER TABLE t1 MODIFY col1 INT;
```

```
ERROR 1105 (HY000): unsupported modify column length 11 is less than origin 20
```

```
ALTER TABLE t1 MODIFY col1 BLOB;
```

```
ERROR 1105 (HY000): unsupported modify column type 252 not match origin 8
```

```
ALTER TABLE t1 MODIFY col1 BIGINT, MODIFY id BIGINT NOT NULL;
```

```
ERROR 1105 (HY000): can't run multi schema change
```

4.1.6.47.3 MySQL 兼容性

- 目前不支持使用单个 ALTER TABLE 语句进行多个修改。
- 仅支持特定类型的数据类型更改。例如，支持将 INTEGER 改为 BIGINT，但不支持将 BIGINT 改为 INTEGER。不支持将整数改为字符串格式或 BLOB 格式。
- 不支持修改 decimal 类型的精度。

4.1.6.47.4 另请参阅

- CREATE TABLE
- SHOW CREATE TABLE
- ADD COLUMN
- DROP COLUMN
- CHANGE COLUMN

4.1.6.48 PREPARE

PREPARE 语句为服务器端预处理语句提供 SQL 接口。

4.1.6.48.1 语法图

PreparedStmt:

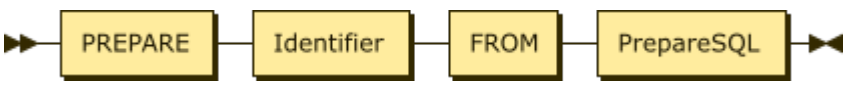


图 187: PreparedStmt

4.1.6.48.2 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)
```

4.1.6.48.3 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。

4.1.6.48.4 另请参阅

- EXECUTE
- DEALLOCATE

4.1.6.49 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。

4.1.6.49.1 语法

```
RECOVER TABLE table_name
```

```
RECOVER TABLE BY JOB ddl_job_id
```

4.1.6.49.2 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。

4.1.6.49.3 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。

4.1.6.49.4 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。

4.1.6.49.5 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

4.1.6.50 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。

4.1.6.50.1 语法图

AlterTableStmt:

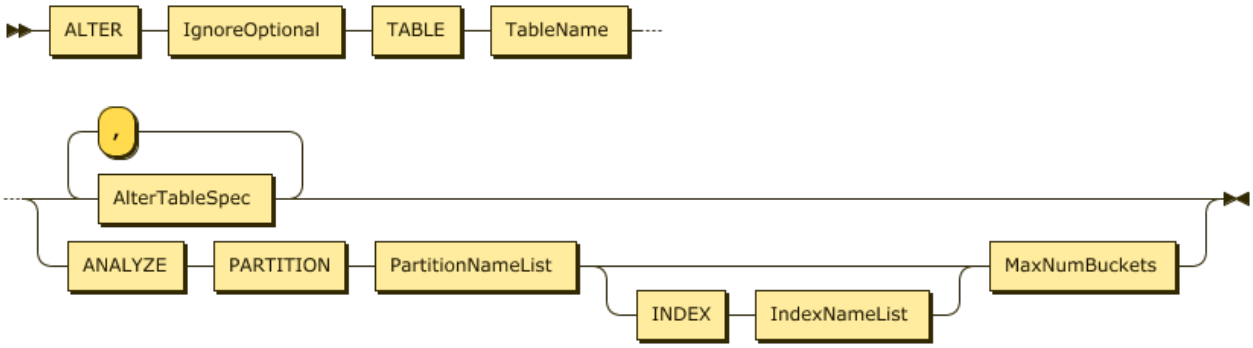


图 188: AlterTableStmt

KeyOrIndex:

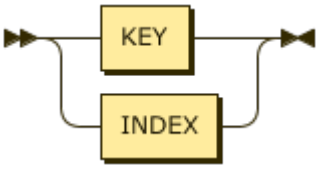


图 189: KeyOrIndex

4.1.6.50.2 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)
```

4.1.6.50.3 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。

4.1.6.50.4 另请参阅

- SHOW CREATE TABLE
- CREATE INDEX
- DROP INDEX
- SHOW INDEX

4.1.6.51 RENAME TABLE

`RENAME TABLE` 语句用于对已有表进行重命名。

4.1.6.51.1 语法图

RenameTableStmt:

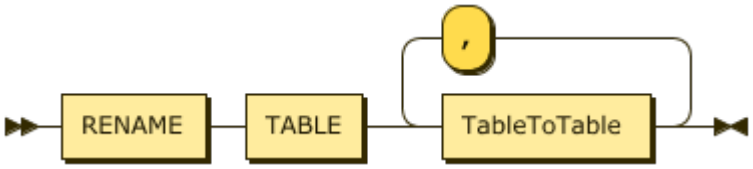


图 190: RenameTableStmt

TableToTable:

图 191: TableToTable

4.1.6.51.2 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)
```

4.1.6.51.3 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。

4.1.6.51.4 另请参阅

- CREATE TABLE
- SHOW TABLES
- ALTER TABLE

4.1.6.52 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。

4.1.6.52.1 语法图

ReplaceIntoStmt:

图 192: ReplaceIntoStmt

PriorityOpt:

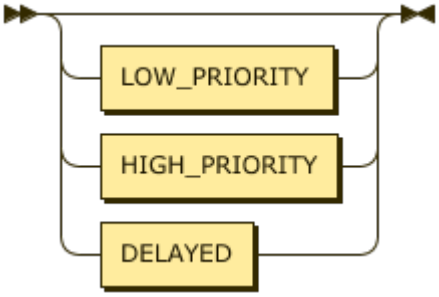


图 193: PriorityOpt

IntoOpt:

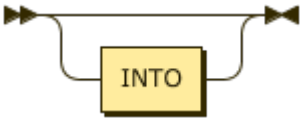


图 194: IntoOpt

TableName:

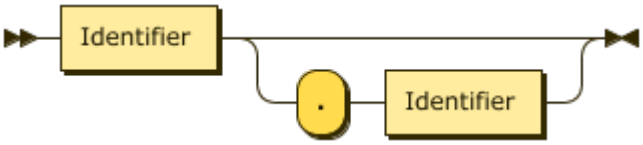


图 195: TableName

InsertValues:

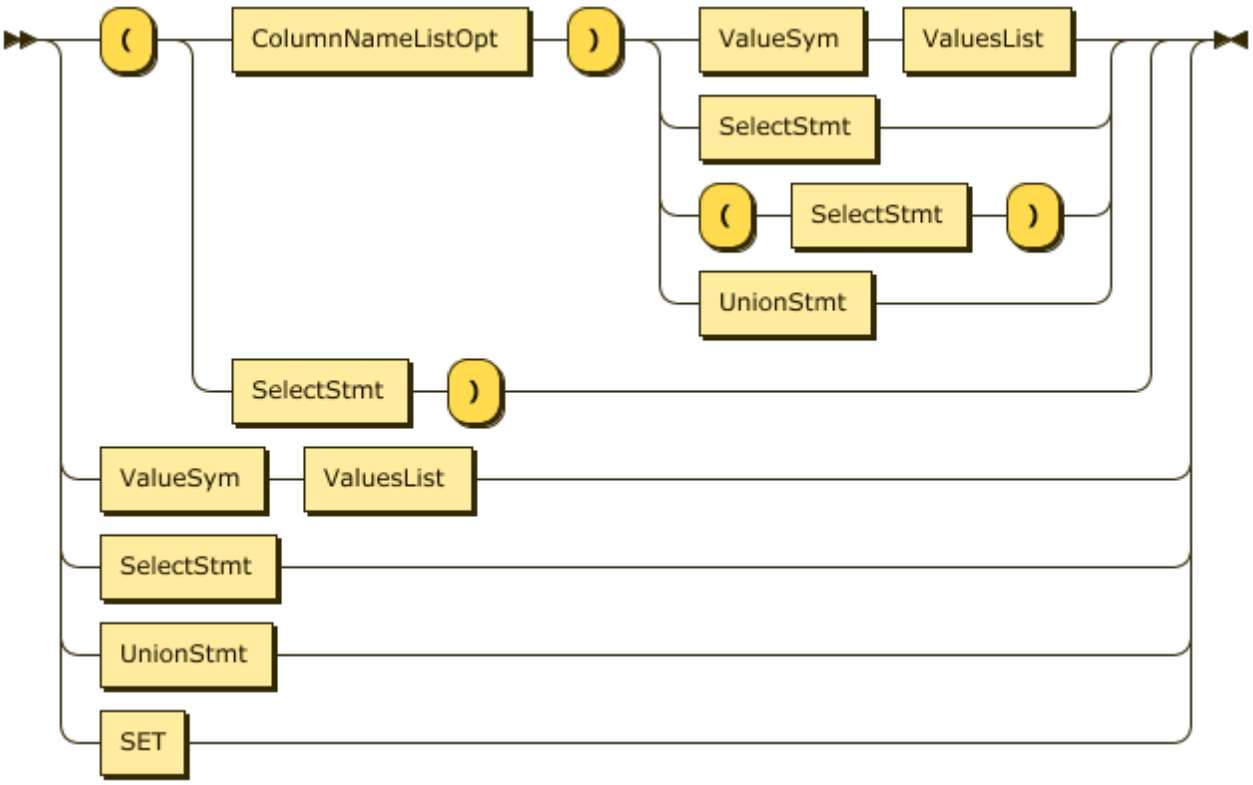


图 196: InsertValues

4.1.6.52.2 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)
```

4.1.6.52.3 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。

4.1.6.52.4 另请参阅

- DELETE
- INSERT
- SELECT
- UPDATE

4.1.6.53 REVOKE <privileges>

REVOKE <privileges> 语句用于删除已有用户的权限。

4.1.6.53.1 语法图

GrantStmt:

图 197: GrantStmt

PrivElemList:

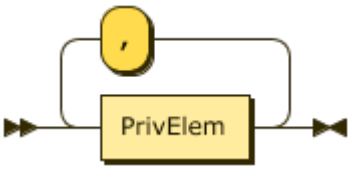


图 198: PrivElemList

PrivElem:

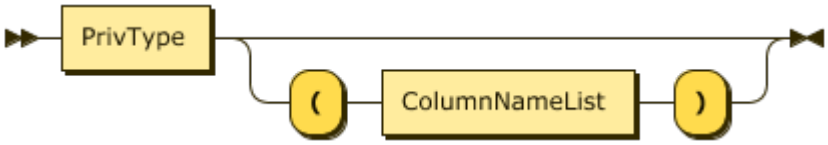


图 199: PrivElem

PrivType:

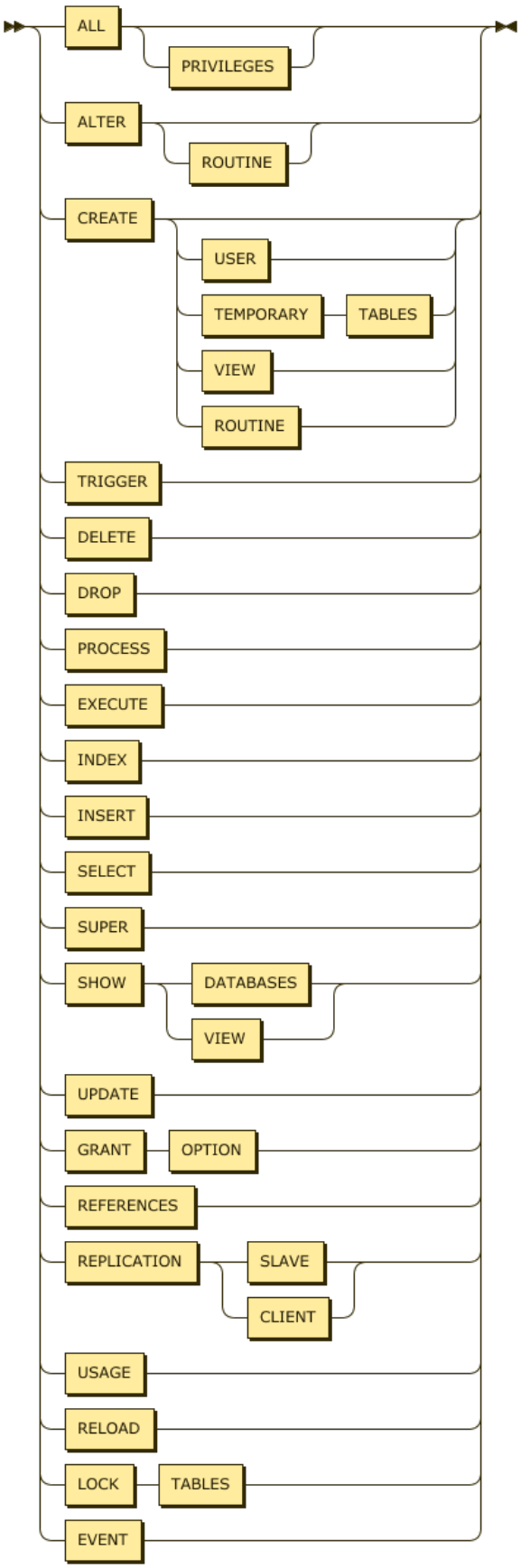


图 200: PrivType

ObjectType:

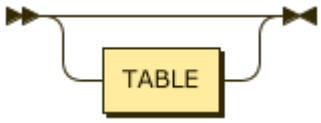


图 201: ObjectType

PrivLevel:

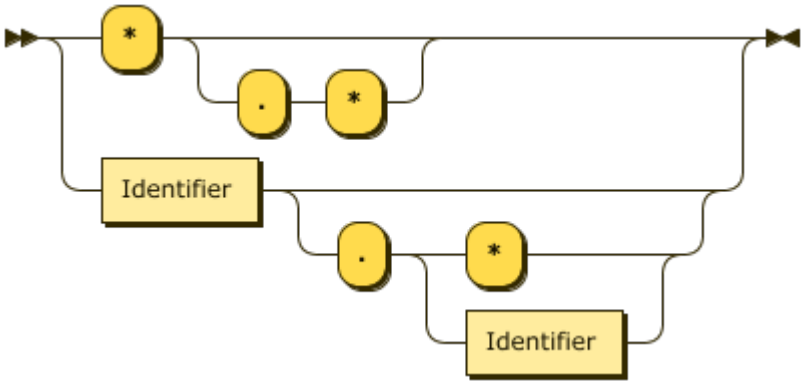


图 202: PrivLevel

UserSpecList:

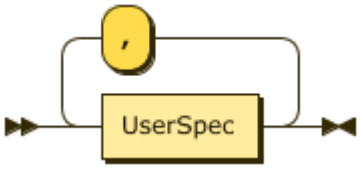


图 203: UserSpecList

4.1.6.53.2 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 '%'
```

4.1.6.53.3 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。

4.1.6.53.4 另请参阅

- GRANT <privileges>
- SHOW GRANTS
- Privilege Management

4.1.6.54 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。

4.1.6.54.1 语法图

RevokeRoleStmt:

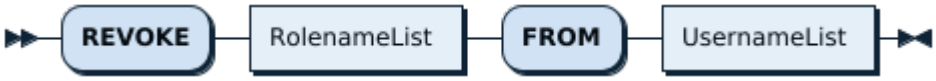


图 204: RevokeRoleStmt

RolenameList:

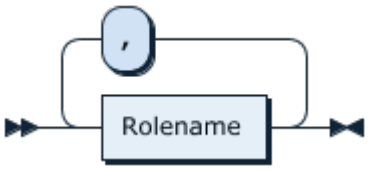


图 205: RolenameList

UsernameList:

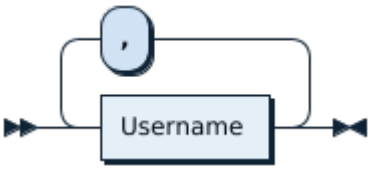


图 206: UsernameList

4.1.6.54.2 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)
```

4.1.6.54.3 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。

4.1.6.54.4 另请参阅

- CREATE ROLE
- DROP ROLE
- GRANT <role>
- SET ROLE
- SET DEFAULT ROLE
- 基于角色的访问控制

4.1.6.55 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。

4.1.6.55.1 语法图

Statement:

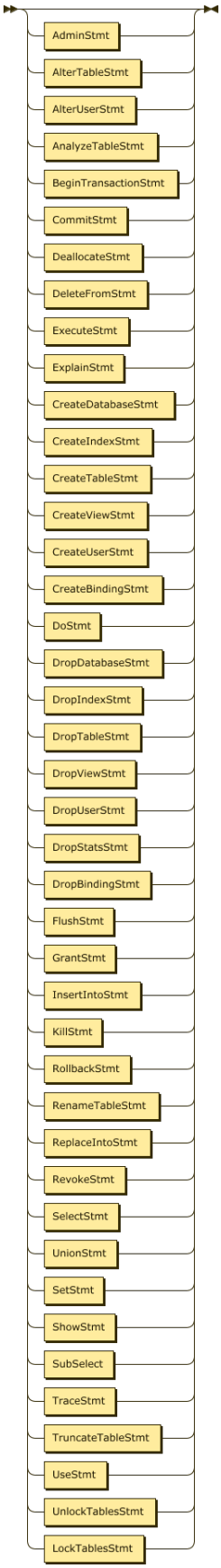


图 207: Statement

4.1.6.55.2 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)
```

4.1.6.55.3 MySQL 兼容性

TiDB 不支持 `ROLLBACK TO SAVEPOINT` 语句。如发现任何兼容性差异，请在 GitHub 上提交 issue。

4.1.6.55.4 另请参阅

- COMMIT
- BEGIN
- START TRANSACTION

4.1.6.56 SELECT

`SELECT` 语句用于从 TiDB 读取数据。

4.1.6.56.1 语法图

SelectStmt:

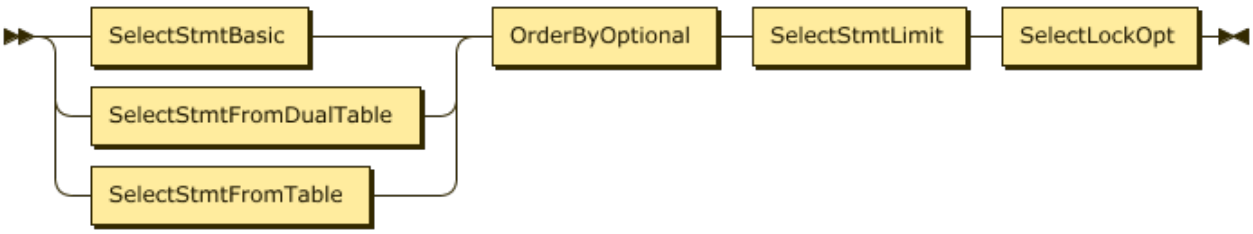


图 208: SelectStmt

FromDual:

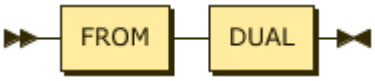


图 209: FromDual

WhereClauseOptional:

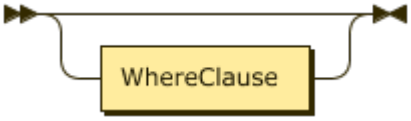


图 210: WhereClauseOptional

SelectStmtOpts:

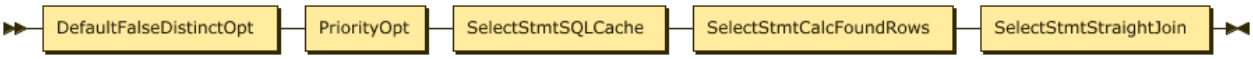


图 211: SelectStmtOpts

SelectStmtFieldList:

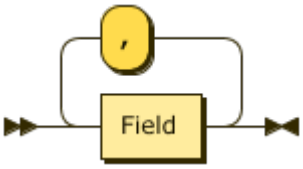


图 212: SelectStmtFieldList

TableRefsClause:

图 213: TableRefsClause

WhereClauseOptional:

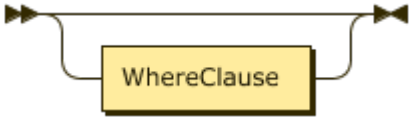


图 214: WhereClauseOptional

SelectStmtGroup:

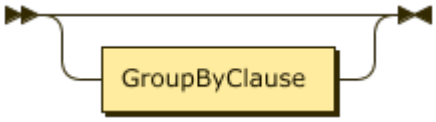


图 215: SelectStmtGroup

HavingClause:

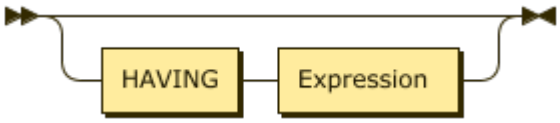


图 216: HavingClause

OrderByOptional:

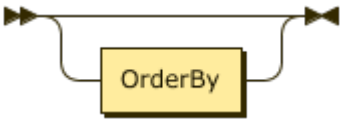


图 217: OrderByOptional

SelectStmtLimit:

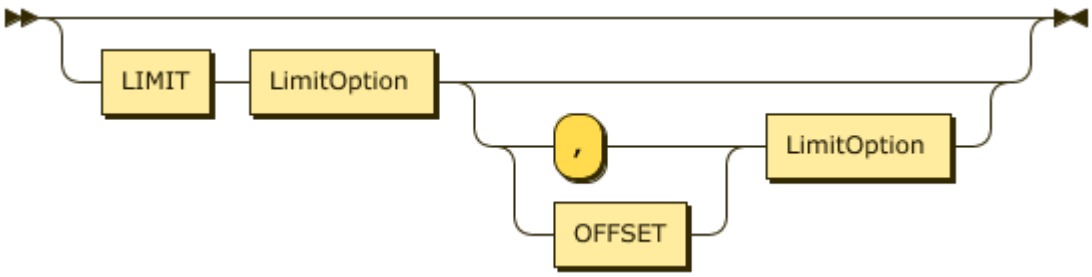


图 218: SelectStmtLimit

SelectLockOpt:

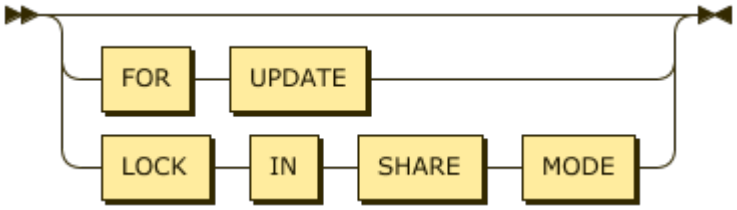


图 219: SelectLockOpt

4.1.6.56.2 语法元素说明

语法元素	说明
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。
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。

语法元素	说明
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

4.1.6.56.3 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)
```

4.1.6.56.4 MySQL 兼容性

SELECT 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。

4.1.6.56.5 另请参阅

- INSERT
- DELETE
- UPDATE
- REPLACE

4.1.6.57 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。

4.1.6.57.1 语法图

SetDefaultRoleStmt:

图 220: SetDefaultRoleStmt

SetDefaultRoleOpt:

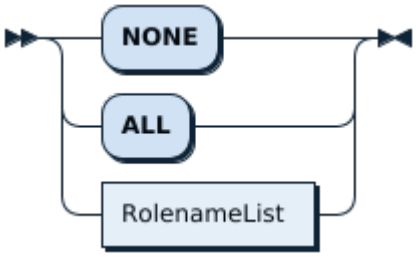


图 221: SetDefaultRoleOpt

RolenameList:

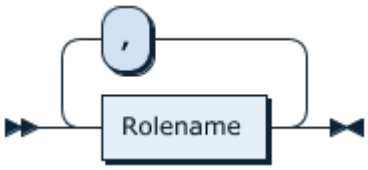


图 222: RolenameList

UsernameList:

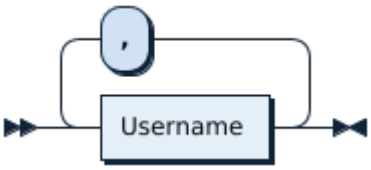


图 223: UsernameList

4.1.6.57.2 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@%
```

4.1.6.57.3 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。

4.1.6.57.4 另请参阅

- `CREATE ROLE`
- `DROP ROLE`
- `GRANT <role>`
- `REVOKE <role>`
- `SET ROLE`
- 基于角色的访问控制

4.1.6.58 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`。

4.1.6.58.1 语法图

SetStmt:

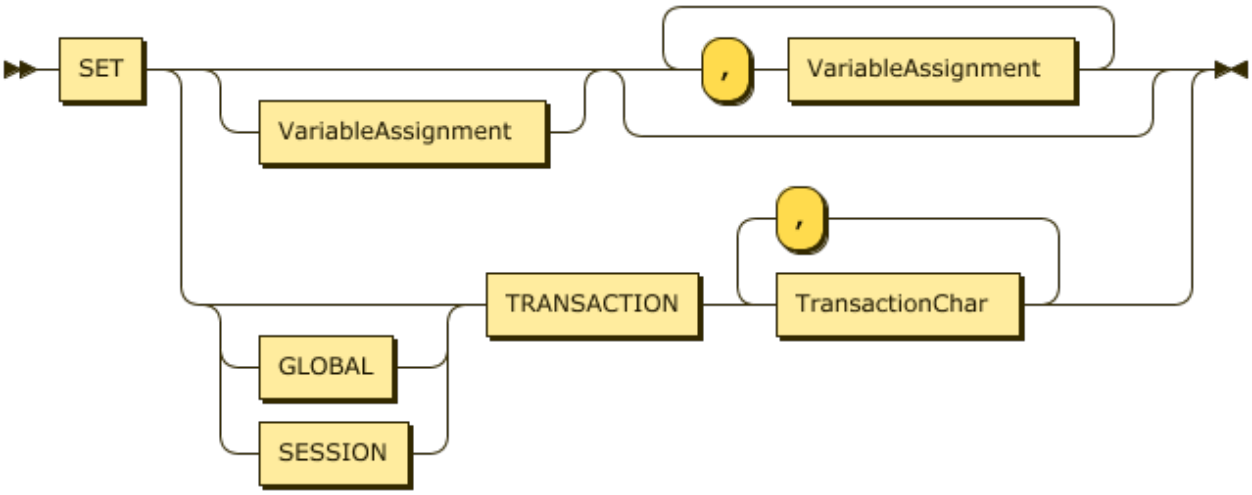


图 224: SetStmt

4.1.6.58.2 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)
```

4.1.6.58.3 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。

4.1.6.58.4 另请参阅

- `SHOW [GLOBAL|SESSION] VARIABLES`
- `SET <variable>`
- Character Set Support

4.1.6.59 SET PASSWORD

`SET PASSWORD` 语句用于更改 TiDB 系统数据库中的用户密码。

4.1.6.59.1 语法图

SetStmt:

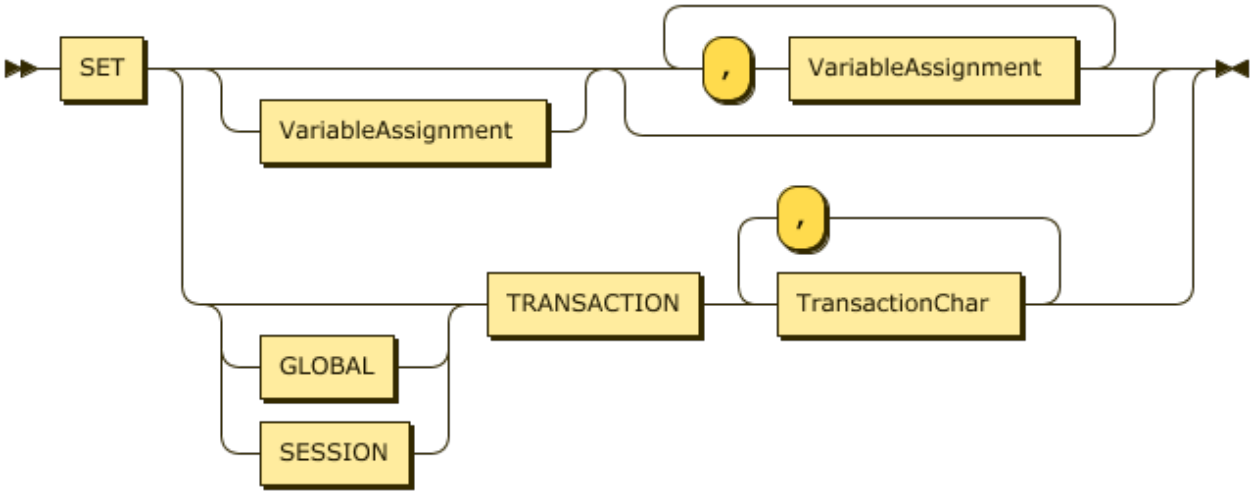


图 225: SetStmt

4.1.6.59.2 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)
```

4.1.6.59.3 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。

4.1.6.59.4 另请参阅

- CREATE USER
- Privilege Management

4.1.6.60 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。

4.1.6.60.1 语法图

SetRoleStmt:

图 226: SetRoleStmt

SetRoleOpt:

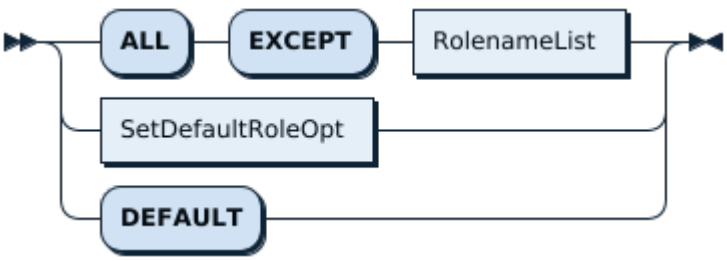


图 227: SetRoleOpt

SetDefaultRoleOpt:

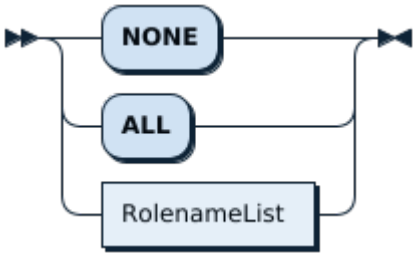


图 228: SetDefaultRoleOpt

4.1.6.60.2 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)
```

4.1.6.60.3 MySQL 兼容性

SET ROLE 语句与 MySQL 8.0 的角色功能完全兼容。如发现任何其他兼容性差异，请在 GitHub 上提交 issue。

4.1.6.60.4 另请参阅

- CREATE ROLE
- DROP ROLE
- GRANT <role>
- REVOKE <role>
- SET DEFAULT ROLE
- 基于角色的访问控制

4.1.6.61 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。

4.1.6.61.1 语法图

SetStmt:

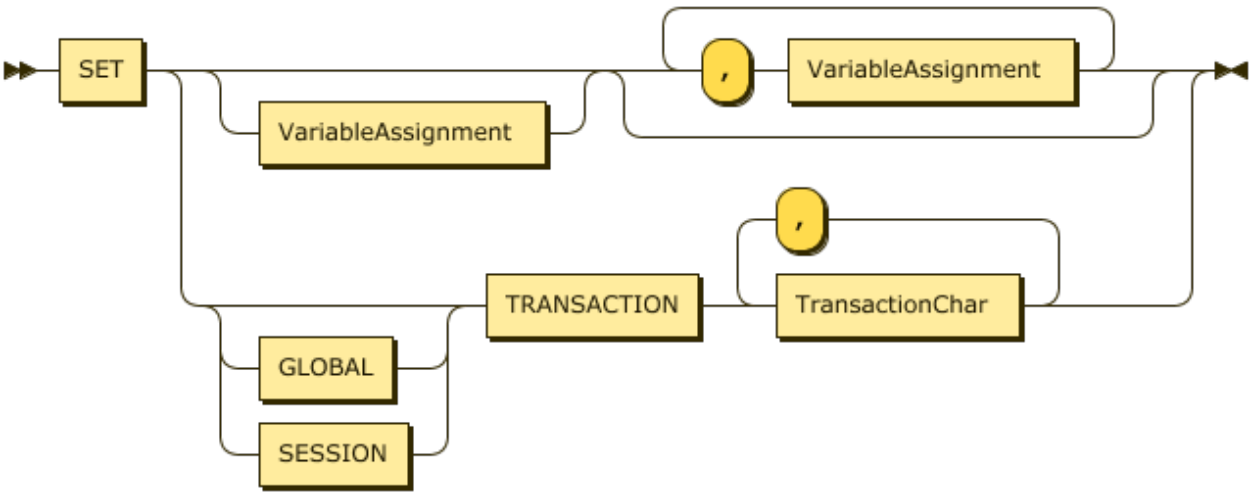


图 229: SetStmt

TransactionChar:

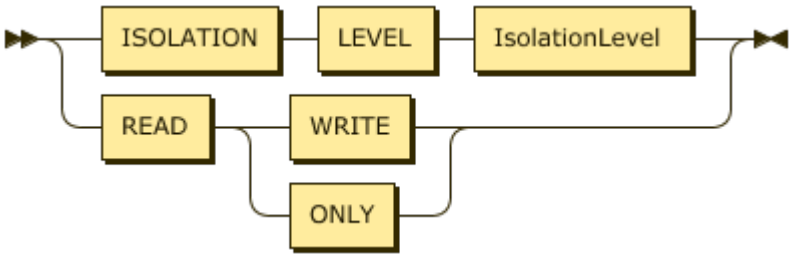


图 230: TransactionChar

IsolationLevel:

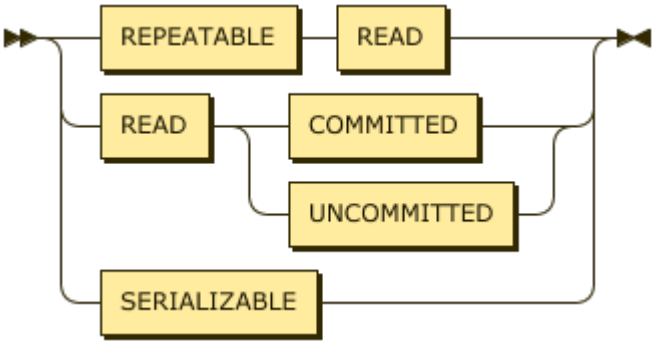


图 231: IsolationLevel

4.1.6.61.2 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)
```

4.1.6.61.3 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。
- 隔离级别 REPEATABLE-READ 在技术上属于快照隔离（Snapshot Isolation）。在 TiDB 中称为 REPEATABLE-READ 是为了和 MySQL 保持一致。

4.1.6.61.4 另请参阅

- SET [GLOBAL|SESSION] <variable>
- Isolation Levels

4.1.6.62 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。需注意，对 GLOBAL 变量的更改不适用于已有连接或本地连接，这与 MySQL 类似。只有新会话才会反映值的变化。

4.1.6.62.1 语法图

SetStmt:

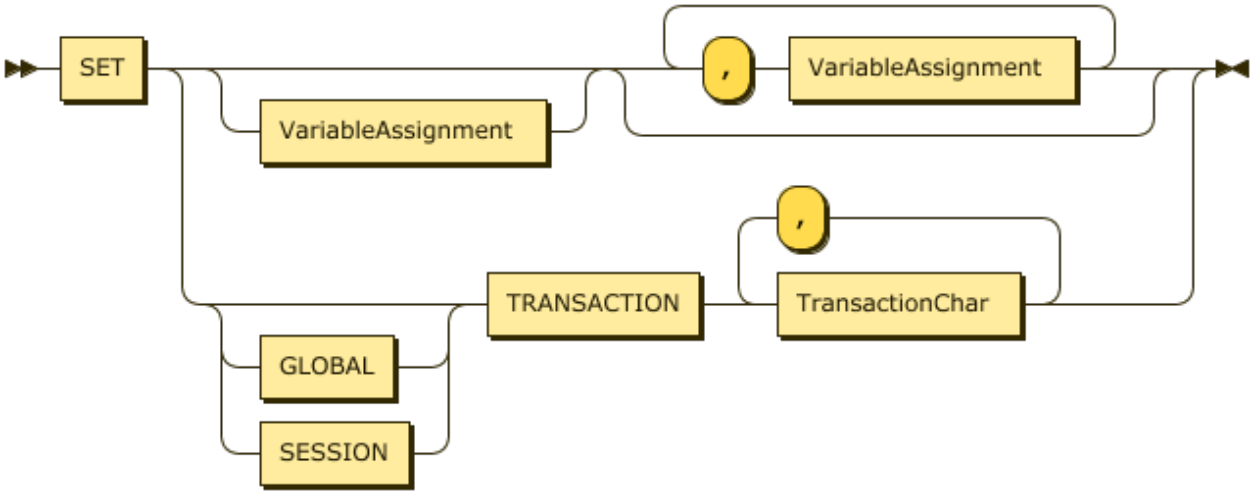


图 232: SetStmt

4.1.6.62.2 示例

```
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)
```

```
SET GLOBAL sql_mode = 'STRICT_TRANS_TABLES,NO_AUTO_CREATE_USER';
```

```
Query OK, 0 rows affected (0.03 sec)
```

```
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 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)
```

4.1.6.62.3 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`。

4.1.6.62.4 另请参阅

- `SHOW [GLOBAL|SESSION] VARIABLES`

4.1.6.63 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。

4.1.6.63.1 语法图

ShowStmt:

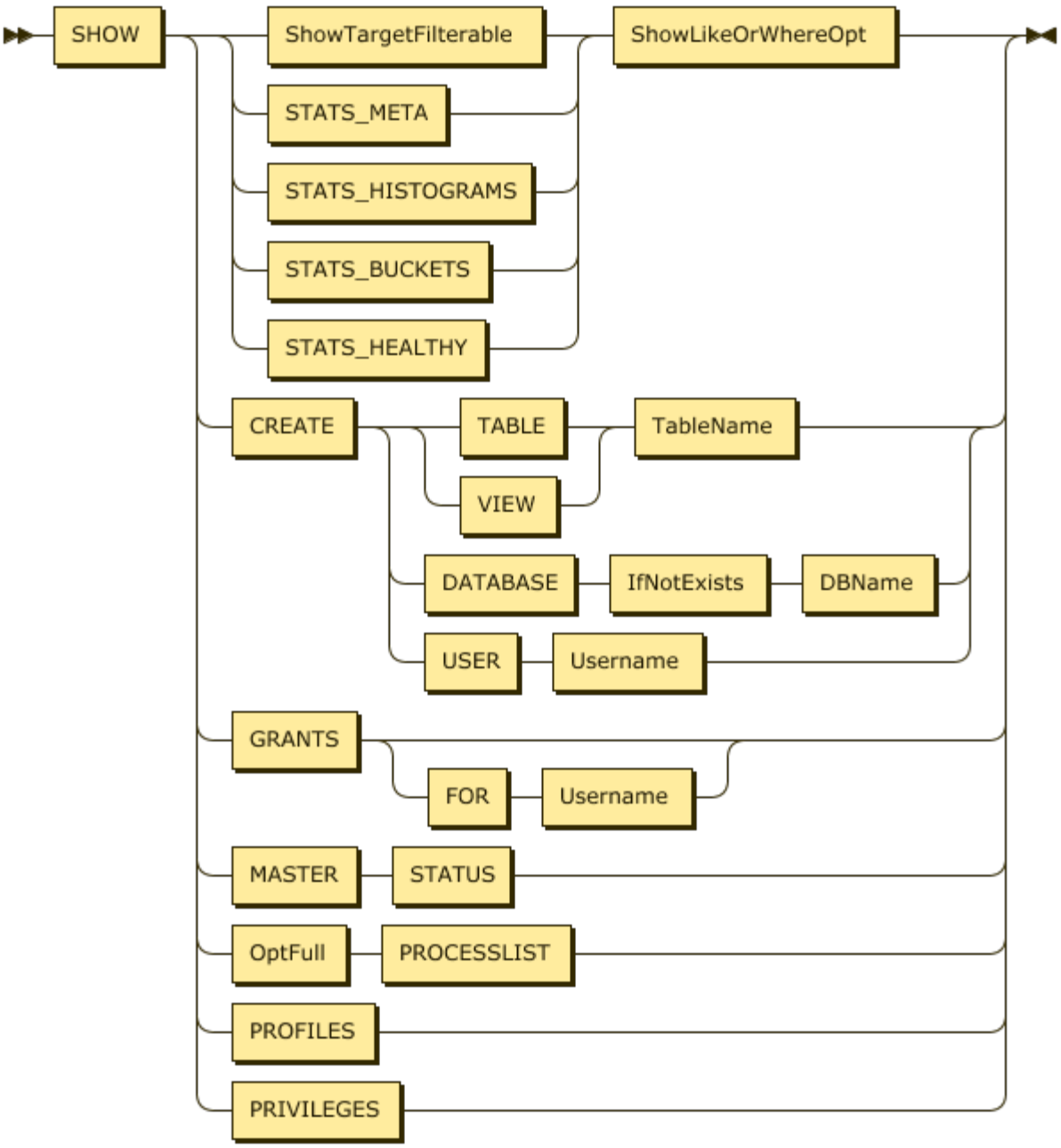


图 233: ShowStmt

ShowTargetFilterable:

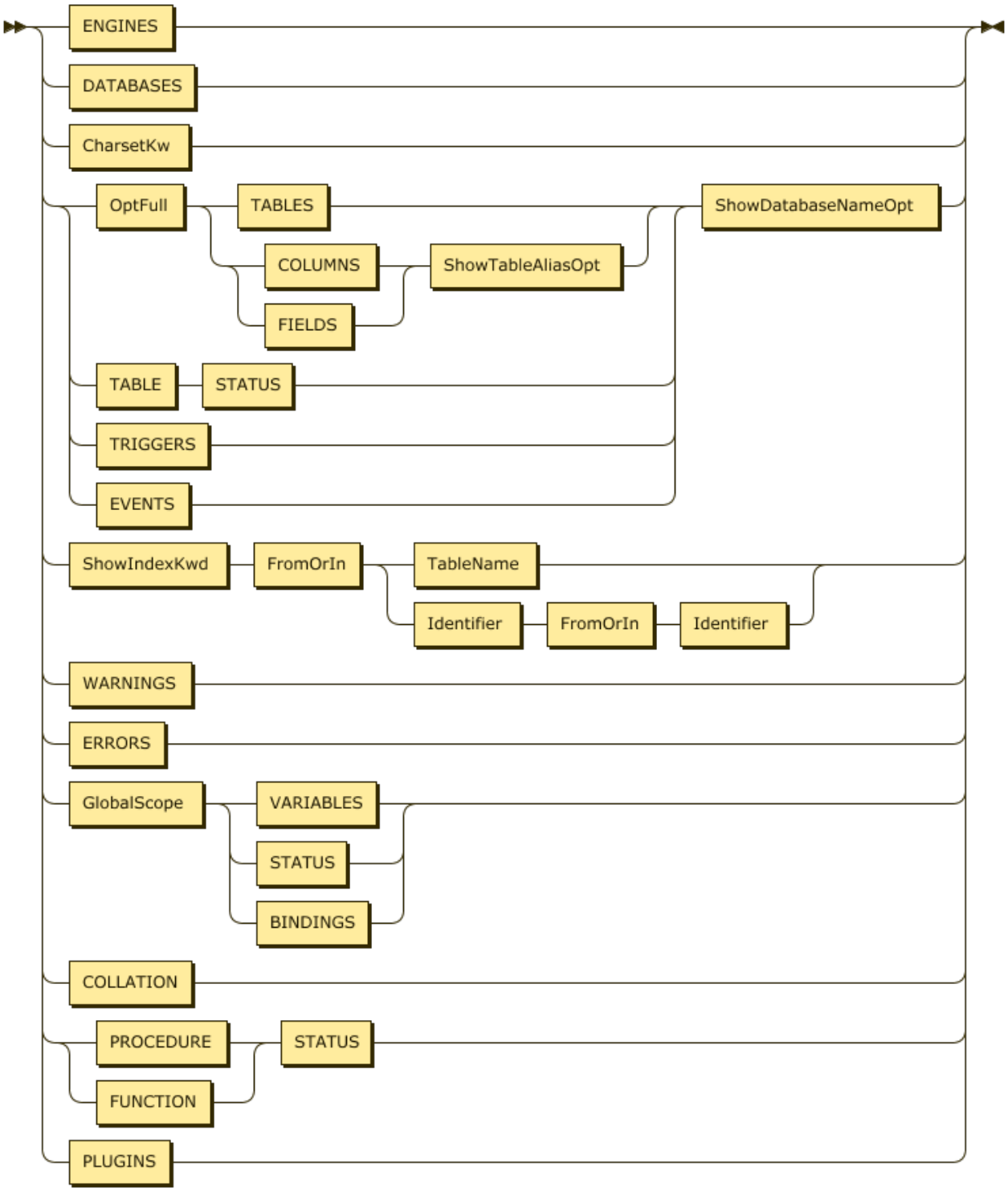


图 234: ShowTargetFilterable

4.1.6.63.2 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)
```

4.1.6.63.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

4.1.6.63.4 另请参阅

- ANALYZE_STATUS 表

4.1.6.64 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。

4.1.6.64.1 语法图

ShowStmt:

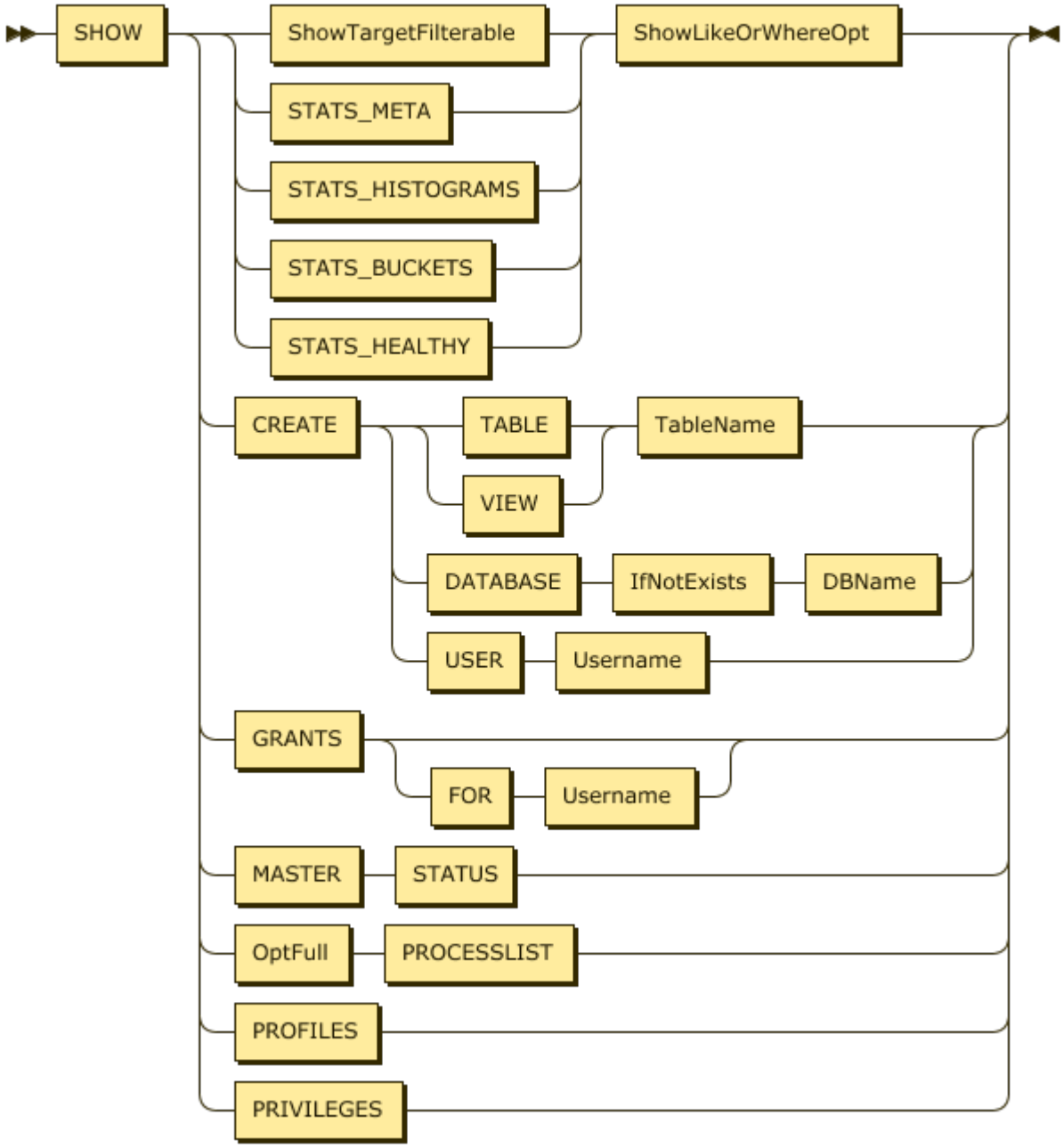


图 235: ShowStmt

ShowTargetFilterable:

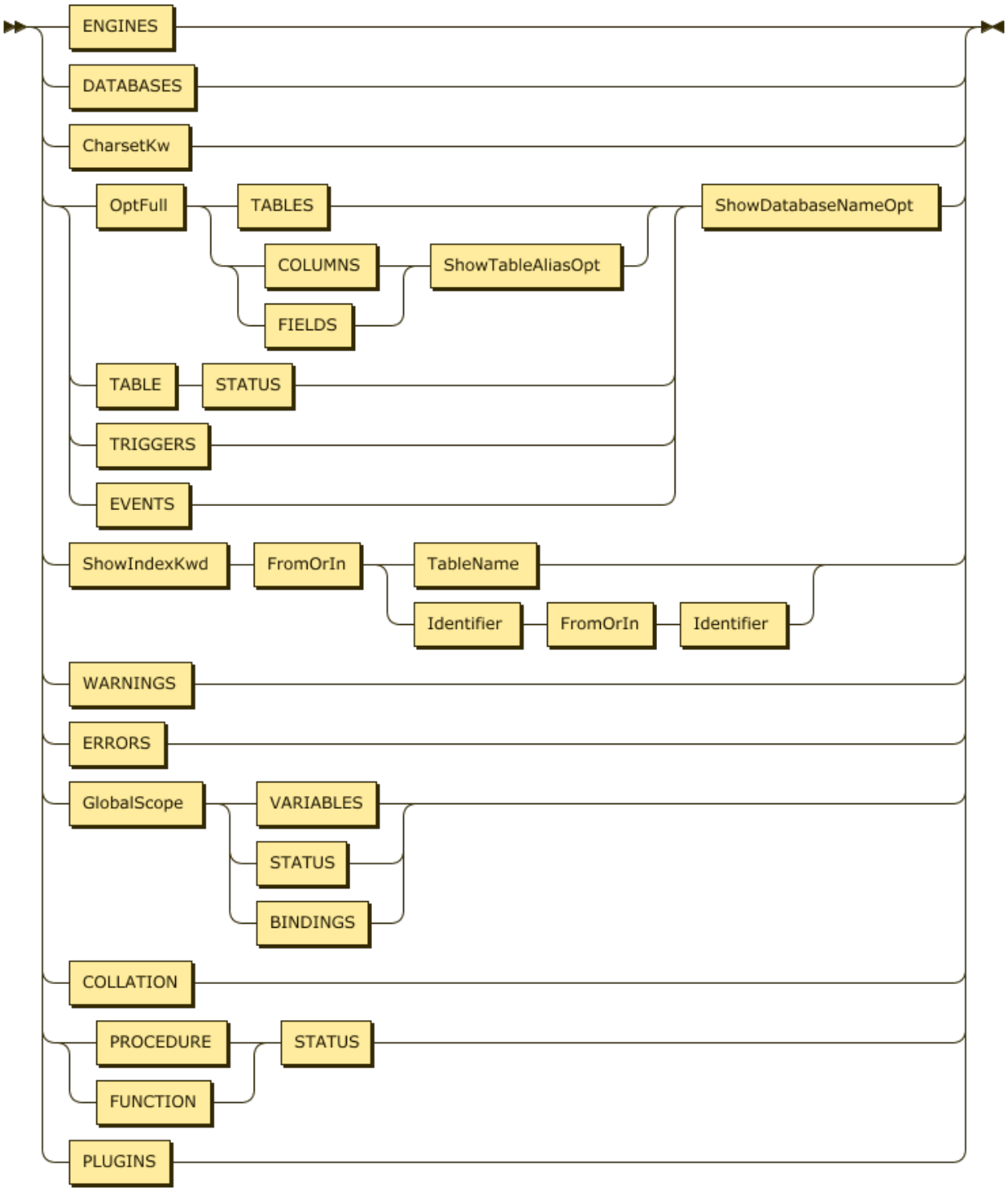


图 236: ShowTargetFilterable

CharsetKw:

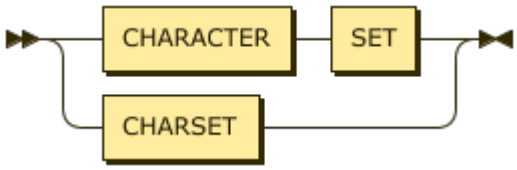


图 237: CharsetKw

4.1.6.64.2 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)
```

4.1.6.64.3 MySQL 兼容性

SHOW CHARACTER SET 语句与 MySQL 完全兼容。但是 TiDB 中字符集可能有与 MySQL 不同的默认排序规则，详情见与 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。

4.1.6.64.4 另请参阅

- SHOW COLLATION

4.1.6.65 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: TiDB 目前仅支持二进制排序规则。

4.1.6.65.1 语法图

ShowStmt:

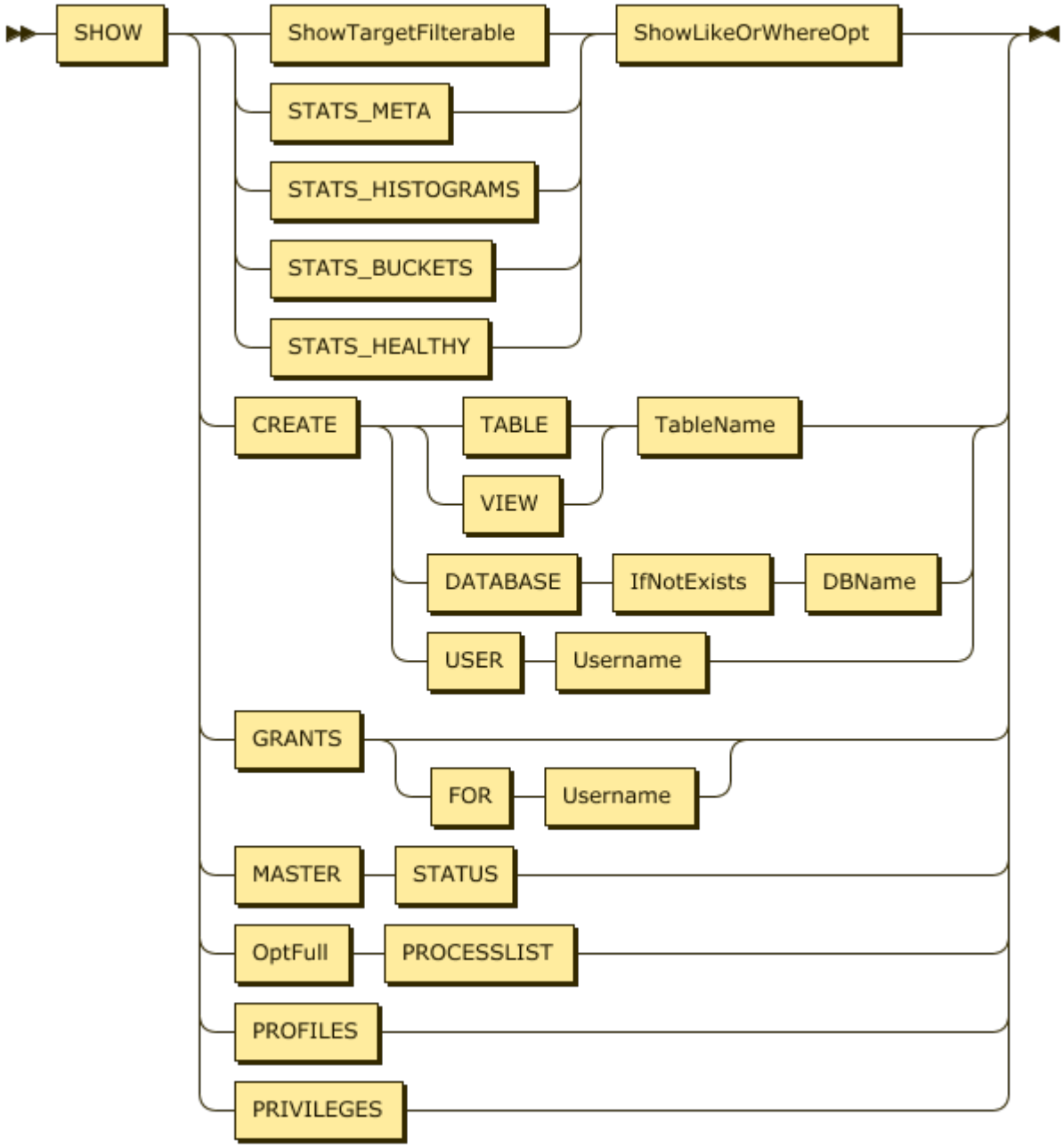


图 238: ShowStmt

ShowTargetFilterable:

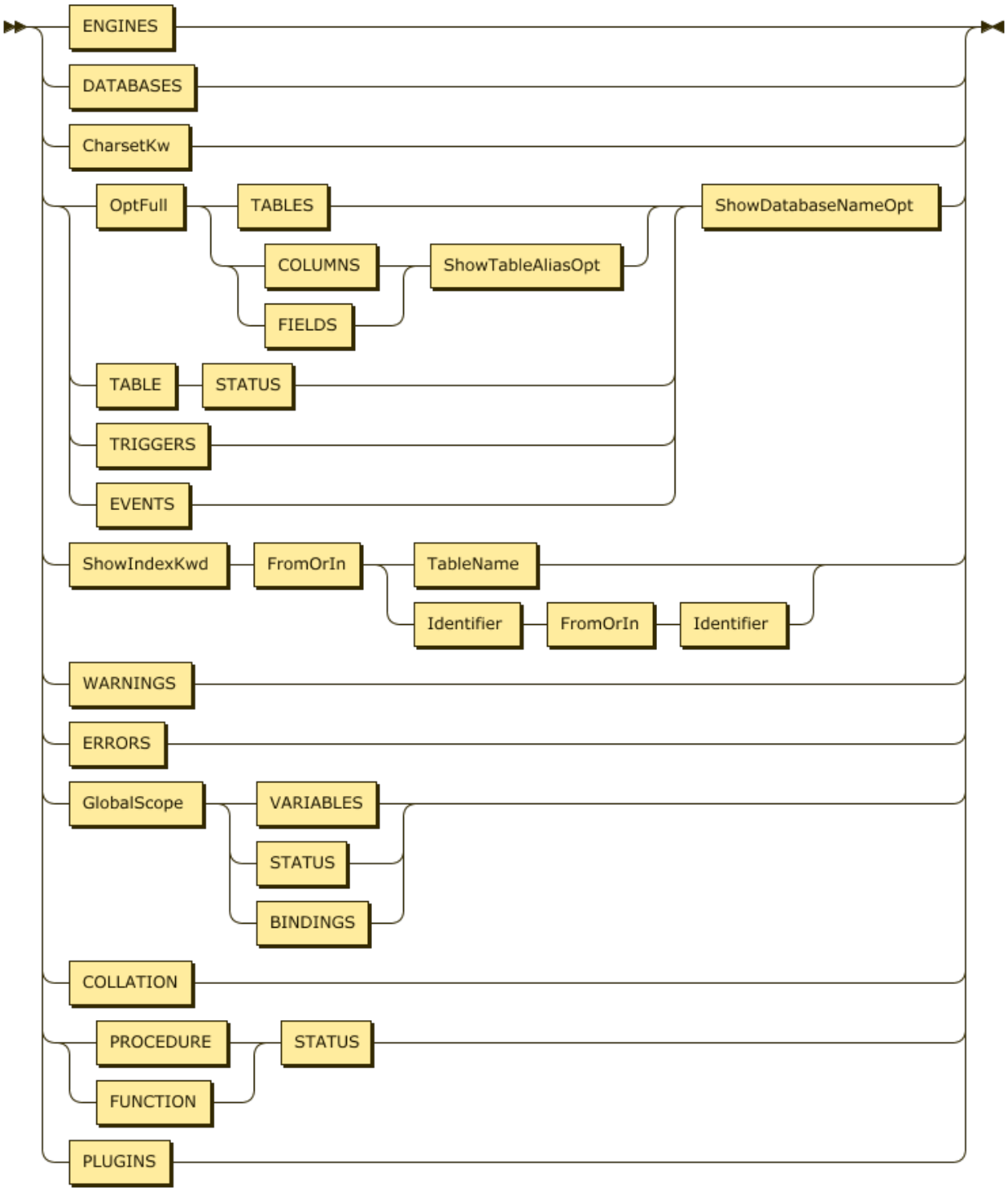


图 239: ShowTargetFilterable

4.1.6.65.2 示例

```
SHOW COLLATION;
```

```
+---------------------------+----------+------+---------+----------+---------+
| Collation                 | Charset  | Id   | Default | Compiled | Sortlen |
+---------------------------+----------+------+---------+----------+---------+
| big5_chinese_ci           | big5     |    1 | Yes     | Yes      |       1 |
| latin2_czech_cs           | latin2   |    2 |         | Yes      |       1 |
| dec8_swedish_ci           | dec8     |    3 | Yes     | Yes      |       1 |
| cp850_general_ci          | cp850    |    4 | Yes     | Yes      |       1 |
| latin1_german1_ci         | latin1   |    5 |         | Yes      |       1 |
| hp8_english_ci            | hp8      |    6 | Yes     | Yes      |       1 |
| koi8r_general_ci          | koi8r    |    7 | Yes     | Yes      |       1 |
| latin1_swedish_ci         | latin1   |    8 | Yes     | Yes      |       1 |
| latin2_general_ci         | latin2   |    9 | Yes     | Yes      |       1 |
| swe7_swedish_ci           | swe7     |   10 | Yes     | Yes      |       1 |
| ascii_general_ci          | ascii    |   11 | Yes     | Yes      |       1 |
| ujis_japanese_ci          | ujis     |   12 | Yes     | Yes      |       1 |
| sjis_japanese_ci          | sjis     |   13 | Yes     | Yes      |       1 |
| cp1251_bulgarian_ci       | cp1251   |   14 |         | Yes      |       1 |
| latin1_danish_ci          | latin1   |   15 |         | Yes      |       1 |
| hebrew_general_ci         | hebrew   |   16 | Yes     | Yes      |       1 |
| tis620_thai_ci            | tis620   |   18 | Yes     | Yes      |       1 |
| euckr_korean_ci           | euckr    |   19 | Yes     | Yes      |       1 |
| latin7_estonian_cs        | latin7   |   20 |         | Yes      |       1 |
| latin2_hungarian_ci       | latin2   |   21 |         | Yes      |       1 |
| koi8u_general_ci          | koi8u    |   22 | Yes     | Yes      |       1 |
| cp1251_ukrainian_ci       | cp1251   |   23 |         | Yes      |       1 |
| gb2312_chinese_ci         | gb2312   |   24 | Yes     | Yes      |       1 |
| greek_general_ci          | greek    |   25 | Yes     | Yes      |       1 |
| cp1250_general_ci         | cp1250   |   26 | Yes     | Yes      |       1 |
| latin2_croatian_ci        | latin2   |   27 |         | Yes      |       1 |
| gbk_chinese_ci            | gbk      |   28 | Yes     | Yes      |       1 |
| cp1257_lithuanian_ci      | cp1257   |   29 |         | Yes      |       1 |
| latin5_turkish_ci         | latin5   |   30 | Yes     | Yes      |       1 |
| latin1_german2_ci         | latin1   |   31 |         | Yes      |       1 |
| armscii8_general_ci       | armscii8 |   32 | Yes     | Yes      |       1 |
| utf8_general_ci           | utf8     |   33 | Yes     | Yes      |       1 |
| cp1250_czech_cs           | cp1250   |   34 |         | Yes      |       1 |
| ucs2_general_ci           | ucs2     |   35 | Yes     | Yes      |       1 |
| cp866_general_ci          | cp866    |   36 | Yes     | Yes      |       1 |
| keybcs2_general_ci        | keybcs2  |   37 | Yes     | Yes      |       1 |
| macce_general_ci          | macce    |   38 | Yes     | Yes      |       1 |
| macroman_general_ci       | macroman |   39 | Yes     | Yes      |       1 |
| cp852_general_ci          | cp852    |   40 | Yes     | Yes      |       1 |
| latin7_general_ci         | latin7   |   41 | Yes     | Yes      |       1 |
| latin7_general_cs         | latin7   |   42 |         | Yes      |       1 |
```

macce_bin	macce	43		Yes	1
cp1250_croatian_ci	cp1250	44		Yes	1
utf8mb4_general_ci	utf8mb4	45	Yes	Yes	1
utf8mb4_bin	utf8mb4	46		Yes	1
latin1_bin	latin1	47		Yes	1
latin1_general_ci	latin1	48		Yes	1
latin1_general_cs	latin1	49		Yes	1
cp1251_bin	cp1251	50		Yes	1
cp1251_general_ci	cp1251	51	Yes	Yes	1
cp1251_general_cs	cp1251	52		Yes	1
macroman_bin	macroman	53		Yes	1
utf16_general_ci	utf16	54	Yes	Yes	1
utf16_bin	utf16	55		Yes	1
utf16le_general_ci	utf16le	56	Yes	Yes	1
cp1256_general_ci	cp1256	57	Yes	Yes	1
cp1257_bin	cp1257	58		Yes	1
cp1257_general_ci	cp1257	59	Yes	Yes	1
utf32_general_ci	utf32	60	Yes	Yes	1
utf32_bin	utf32	61		Yes	1
utf16le_bin	utf16le	62		Yes	1
binary	binary	63	Yes	Yes	1
armscii8_bin	armscii8	64		Yes	1
ascii_bin	ascii	65		Yes	1
cp1250_bin	cp1250	66		Yes	1
cp1256_bin	cp1256	67		Yes	1
cp866_bin	cp866	68		Yes	1
dec8_bin	dec8	69		Yes	1
greek_bin	greek	70		Yes	1
hebrew_bin	hebrew	71		Yes	1
hp8_bin	hp8	72		Yes	1
keybcs2_bin	keybcs2	73		Yes	1
koi8r_bin	koi8r	74		Yes	1
koi8u_bin	koi8u	75		Yes	1
latin2_bin	latin2	77		Yes	1
latin5_bin	latin5	78		Yes	1
latin7_bin	latin7	79		Yes	1
cp850_bin	cp850	80		Yes	1
cp852_bin	cp852	81		Yes	1
swe7_bin	swe7	82		Yes	1
utf8_bin	utf8	83		Yes	1
big5_bin	big5	84		Yes	1
euckr_bin	euckr	85		Yes	1
gb2312_bin	gb2312	86		Yes	1
gbk_bin	gbk	87		Yes	1
sjis_bin	sjis	88		Yes	1

tis620_bin	tis620	89		Yes	1
ucs2_bin	ucs2	90		Yes	1
ujis_bin	ujis	91		Yes	1
geostd8_general_ci	geostd8	92	Yes	Yes	1
geostd8_bin	geostd8	93		Yes	1
latin1_spanish_ci	latin1	94		Yes	1
cp932_japanese_ci	cp932	95	Yes	Yes	1
cp932_bin	cp932	96		Yes	1
eucjpms_japanese_ci	eucjpms	97	Yes	Yes	1
eucjpms_bin	eucjpms	98		Yes	1
cp1250_polish_ci	cp1250	99		Yes	1
utf16_unicode_ci	utf16	101		Yes	1
utf16_icelandic_ci	utf16	102		Yes	1
utf16_latvian_ci	utf16	103		Yes	1
utf16_romanian_ci	utf16	104		Yes	1
utf16_slovenian_ci	utf16	105		Yes	1
utf16_polish_ci	utf16	106		Yes	1
utf16_estonian_ci	utf16	107		Yes	1
utf16_spanish_ci	utf16	108		Yes	1
utf16_swedish_ci	utf16	109		Yes	1
utf16_turkish_ci	utf16	110		Yes	1
utf16_czech_ci	utf16	111		Yes	1
utf16_danish_ci	utf16	112		Yes	1
utf16_lithuanian_ci	utf16	113		Yes	1
utf16_slovak_ci	utf16	114		Yes	1
utf16_spanish2_ci	utf16	115		Yes	1
utf16_roman_ci	utf16	116		Yes	1
utf16_persian_ci	utf16	117		Yes	1
utf16_esperanto_ci	utf16	118		Yes	1
utf16_hungarian_ci	utf16	119		Yes	1
utf16_sinhala_ci	utf16	120		Yes	1
utf16_german2_ci	utf16	121		Yes	1
utf16_croatian_ci	utf16	122		Yes	1
utf16_unicode_520_ci	utf16	123		Yes	1
utf16_vietnamese_ci	utf16	124		Yes	1
ucs2_unicode_ci	ucs2	128		Yes	1
ucs2_icelandic_ci	ucs2	129		Yes	1
ucs2_latvian_ci	ucs2	130		Yes	1
ucs2_romanian_ci	ucs2	131		Yes	1
ucs2_slovenian_ci	ucs2	132		Yes	1
ucs2_polish_ci	ucs2	133		Yes	1
ucs2_estonian_ci	ucs2	134		Yes	1
ucs2_spanish_ci	ucs2	135		Yes	1
ucs2_swedish_ci	ucs2	136		Yes	1
ucs2_turkish_ci	ucs2	137		Yes	1

```
| ucs2_czech_ci            | ucs2     |  138 |          | Yes      |       1 |
| ucs2_danish_ci           | ucs2     |  139 |          | Yes      |       1 |
| ucs2_lithuanian_ci       | ucs2     |  140 |          | Yes      |       1 |
| ucs2_slovak_ci           | ucs2     |  141 |          | Yes      |       1 |
| ucs2_spanish2_ci         | ucs2     |  142 |          | Yes      |       1 |
| ucs2_roman_ci            | ucs2     |  143 |          | Yes      |       1 |
| ucs2_persian_ci          | ucs2     |  144 |          | Yes      |       1 |
| ucs2_esperanto_ci        | ucs2     |  145 |          | Yes      |       1 |
| ucs2_hungarian_ci        | ucs2     |  146 |          | Yes      |       1 |
| ucs2_sinhala_ci          | ucs2     |  147 |          | Yes      |       1 |
| ucs2_german2_ci          | ucs2     |  148 |          | Yes      |       1 |
| ucs2_croatian_ci         | ucs2     |  149 |          | Yes      |       1 |
| ucs2_unicode_520_ci      | ucs2     |  150 |          | Yes      |       1 |
| ucs2_vietnamese_ci       | ucs2     |  151 |          | Yes      |       1 |
| ucs2_general_mysql500_ci | ucs2     |  159 |          | Yes      |       1 |
| utf32_unicode_ci         | utf32    |  160 |          | Yes      |       1 |
| utf32_icelandic_ci       | utf32    |  161 |          | Yes      |       1 |
| utf32_latvian_ci         | utf32    |  162 |          | Yes      |       1 |
| utf32_romanian_ci        | utf32    |  163 |          | Yes      |       1 |
| utf32_slovenian_ci       | utf32    |  164 |          | Yes      |       1 |
| utf32_polish_ci          | utf32    |  165 |          | Yes      |       1 |
| utf32_estonian_ci        | utf32    |  166 |          | Yes      |       1 |
| utf32_spanish_ci         | utf32    |  167 |          | Yes      |       1 |
| utf32_swedish_ci         | utf32    |  168 |          | Yes      |       1 |
| utf32_turkish_ci         | utf32    |  169 |          | Yes      |       1 |
| utf32_czech_ci           | utf32    |  170 |          | Yes      |       1 |
| utf32_danish_ci          | utf32    |  171 |          | Yes      |       1 |
| utf32_lithuanian_ci      | utf32    |  172 |          | Yes      |       1 |
| utf32_slovak_ci          | utf32    |  173 |          | Yes      |       1 |
| utf32_spanish2_ci        | utf32    |  174 |          | Yes      |       1 |
| utf32_roman_ci           | utf32    |  175 |          | Yes      |       1 |
| utf32_persian_ci         | utf32    |  176 |          | Yes      |       1 |
| utf32_esperanto_ci       | utf32    |  177 |          | Yes      |       1 |
| utf32_hungarian_ci       | utf32    |  178 |          | Yes      |       1 |
| utf32_sinhala_ci         | utf32    |  179 |          | Yes      |       1 |
| utf32_german2_ci         | utf32    |  180 |          | Yes      |       1 |
| utf32_croatian_ci        | utf32    |  181 |          | Yes      |       1 |
| utf32_unicode_520_ci     | utf32    |  182 |          | Yes      |       1 |
| utf32_vietnamese_ci      | utf32    |  183 |          | Yes      |       1 |
| utf8_unicode_ci          | utf8     |  192 |          | Yes      |       1 |
| utf8_icelandic_ci        | utf8     |  193 |          | Yes      |       1 |
| utf8_latvian_ci          | utf8     |  194 |          | Yes      |       1 |
| utf8_romanian_ci         | utf8     |  195 |          | Yes      |       1 |
| utf8_slovenian_ci        | utf8     |  196 |          | Yes      |       1 |
| utf8_polish_ci           | utf8     |  197 |          | Yes      |       1 |
```

```
| utf8_estonian_ci         | utf8     |  198 |         | Yes      |       1 |
| utf8_spanish_ci          | utf8     |  199 |         | Yes      |       1 |
| utf8_swedish_ci          | utf8     |  200 |         | Yes      |       1 |
| utf8_turkish_ci          | utf8     |  201 |         | Yes      |       1 |
| utf8_czech_ci            | utf8     |  202 |         | Yes      |       1 |
| utf8_danish_ci           | utf8     |  203 |         | Yes      |       1 |
| utf8_lithuanian_ci       | utf8     |  204 |         | Yes      |       1 |
| utf8_slovak_ci           | utf8     |  205 |         | Yes      |       1 |
| utf8_spanish2_ci         | utf8     |  206 |         | Yes      |       1 |
| utf8_roman_ci            | utf8     |  207 |         | Yes      |       1 |
| utf8_persian_ci          | utf8     |  208 |         | Yes      |       1 |
| utf8_esperanto_ci        | utf8     |  209 |         | Yes      |       1 |
| utf8_hungarian_ci        | utf8     |  210 |         | Yes      |       1 |
| utf8_sinhala_ci          | utf8     |  211 |         | Yes      |       1 |
| utf8_german2_ci          | utf8     |  212 |         | Yes      |       1 |
| utf8_croatian_ci         | utf8     |  213 |         | Yes      |       1 |
| utf8_unicode_520_ci      | utf8     |  214 |         | Yes      |       1 |
| utf8_vietnamese_ci       | utf8     |  215 |         | Yes      |       1 |
| utf8_general_mysql500_ci | utf8     |  223 |         | Yes      |       1 |
| utf8mb4_unicode_ci       | utf8mb4  |  224 |         | Yes      |       1 |
| utf8mb4_icelandic_ci     | utf8mb4  |  225 |         | Yes      |       1 |
| utf8mb4_latvian_ci       | utf8mb4  |  226 |         | Yes      |       1 |
| utf8mb4_romanian_ci      | utf8mb4  |  227 |         | Yes      |       1 |
| utf8mb4_slovenian_ci     | utf8mb4  |  228 |         | Yes      |       1 |
| utf8mb4_polish_ci        | utf8mb4  |  229 |         | Yes      |       1 |
| utf8mb4_estonian_ci      | utf8mb4  |  230 |         | Yes      |       1 |
| utf8mb4_spanish_ci       | utf8mb4  |  231 |         | Yes      |       1 |
| utf8mb4_swedish_ci       | utf8mb4  |  232 |         | Yes      |       1 |
| utf8mb4_turkish_ci       | utf8mb4  |  233 |         | Yes      |       1 |
| utf8mb4_czech_ci         | utf8mb4  |  234 |         | Yes      |       1 |
| utf8mb4_danish_ci        | utf8mb4  |  235 |         | Yes      |       1 |
| utf8mb4_lithuanian_ci    | utf8mb4  |  236 |         | Yes      |       1 |
| utf8mb4_slovak_ci        | utf8mb4  |  237 |         | Yes      |       1 |
| utf8mb4_spanish2_ci      | utf8mb4  |  238 |         | Yes      |       1 |
| utf8mb4_roman_ci         | utf8mb4  |  239 |         | Yes      |       1 |
| utf8mb4_persian_ci       | utf8mb4  |  240 |         | Yes      |       1 |
| utf8mb4_esperanto_ci     | utf8mb4  |  241 |         | Yes      |       1 |
| utf8mb4_hungarian_ci     | utf8mb4  |  242 |         | Yes      |       1 |
| utf8mb4_sinhala_ci       | utf8mb4  |  243 |         | Yes      |       1 |
| utf8mb4_german2_ci       | utf8mb4  |  244 |         | Yes      |       1 |
| utf8mb4_croatian_ci      | utf8mb4  |  245 |         | Yes      |       1 |
| utf8mb4_unicode_520_ci   | utf8mb4  |  246 |         | Yes      |       1 |
| utf8mb4_vietnamese_ci    | utf8mb4  |  247 |         | Yes      |       1 |
+--------------------------+----------+------+---------+----------+---------+
219 rows in set (0.00 sec)
```

4.1.6.65.3 MySQL 兼容性

SHOW COLLATION 语句与 MySQL 完全兼容。但是 TiDB 中字符集可能有与 MySQL 不同的默认排序规则，详情见与 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。

4.1.6.65.4 另请参阅

- SHOW CHARACTER SET

4.1.6.66 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。

4.1.6.66.1 语法图

ShowStmt:

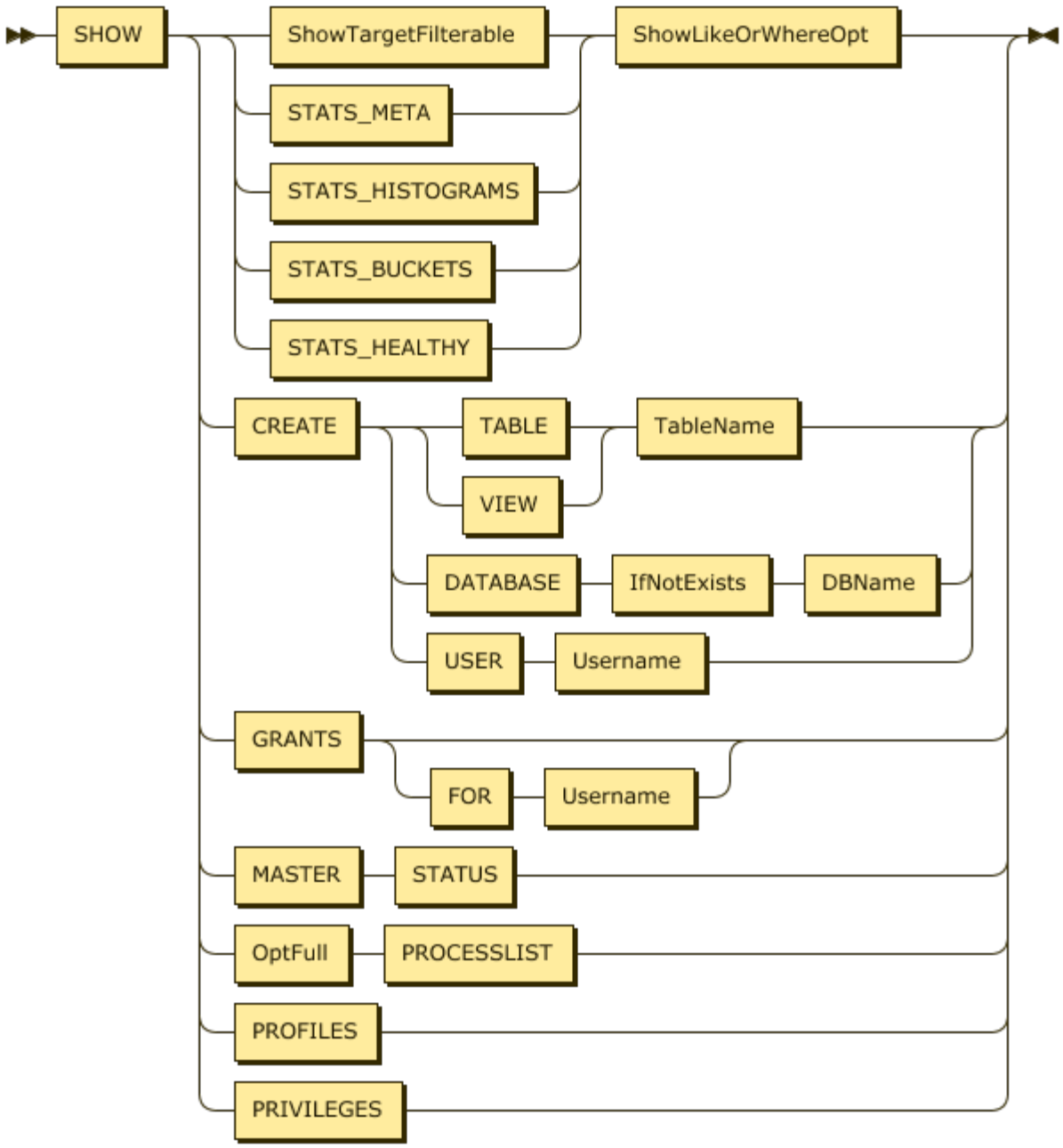


图 240: ShowStmt

ShowTargetFilterable:

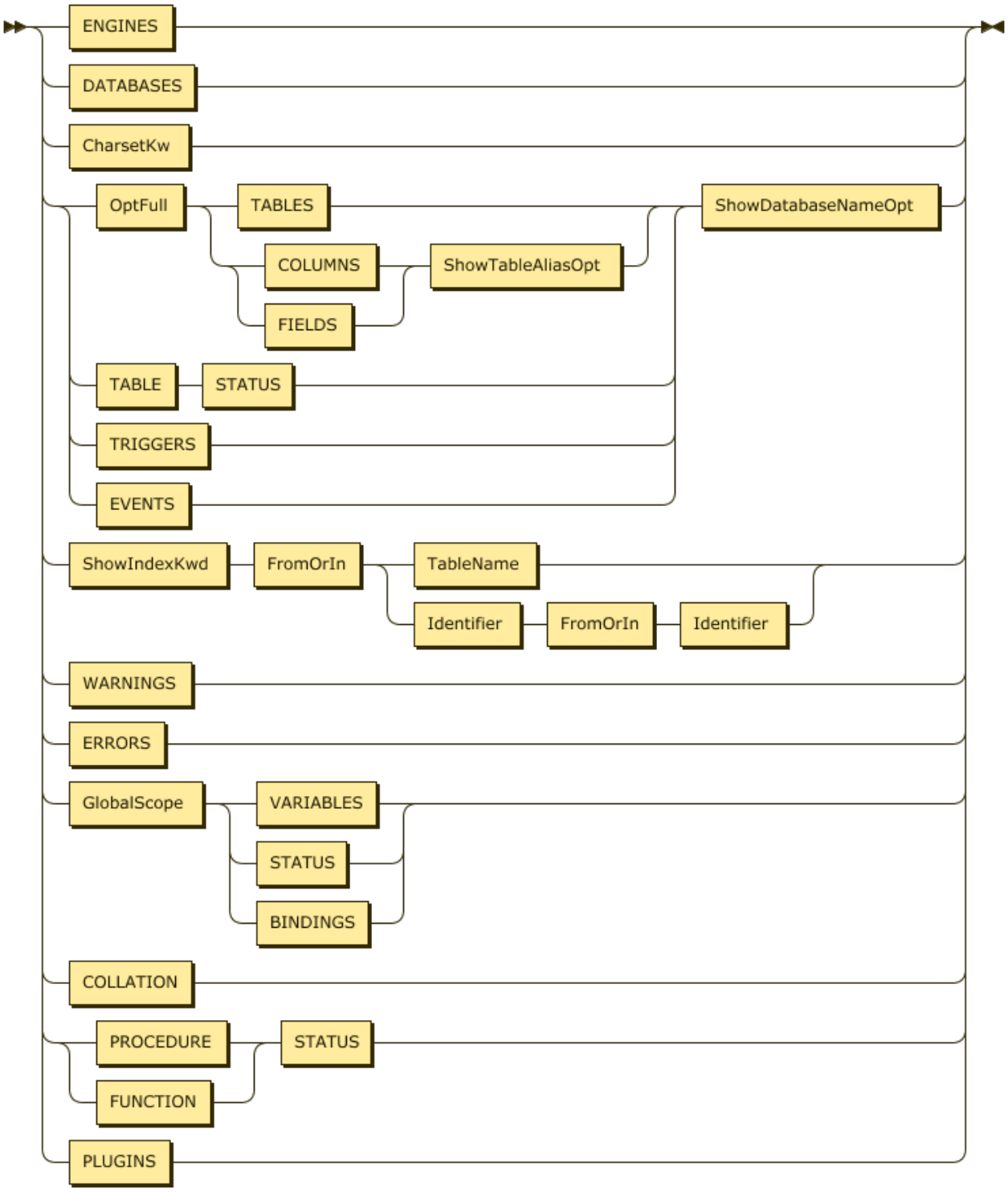


图 241: ShowTargetFilterable

OptFull:

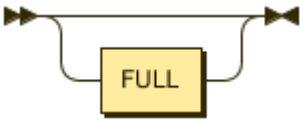


图 242: OptFull

4.1.6.66.2 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 |         |
| Password              | char(41)      | 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 |         |
+-----------------------+---------------+-------------+------+------+---------+-------+-------------------------
    ↪
29 rows in set (0.00 sec)
```

4.1.6.66.3 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。

4.1.6.66.4 另请参阅

- SHOW CREATE TABLE

4.1.6.67 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。

4.1.6.67.1 语法图

ShowStmt:

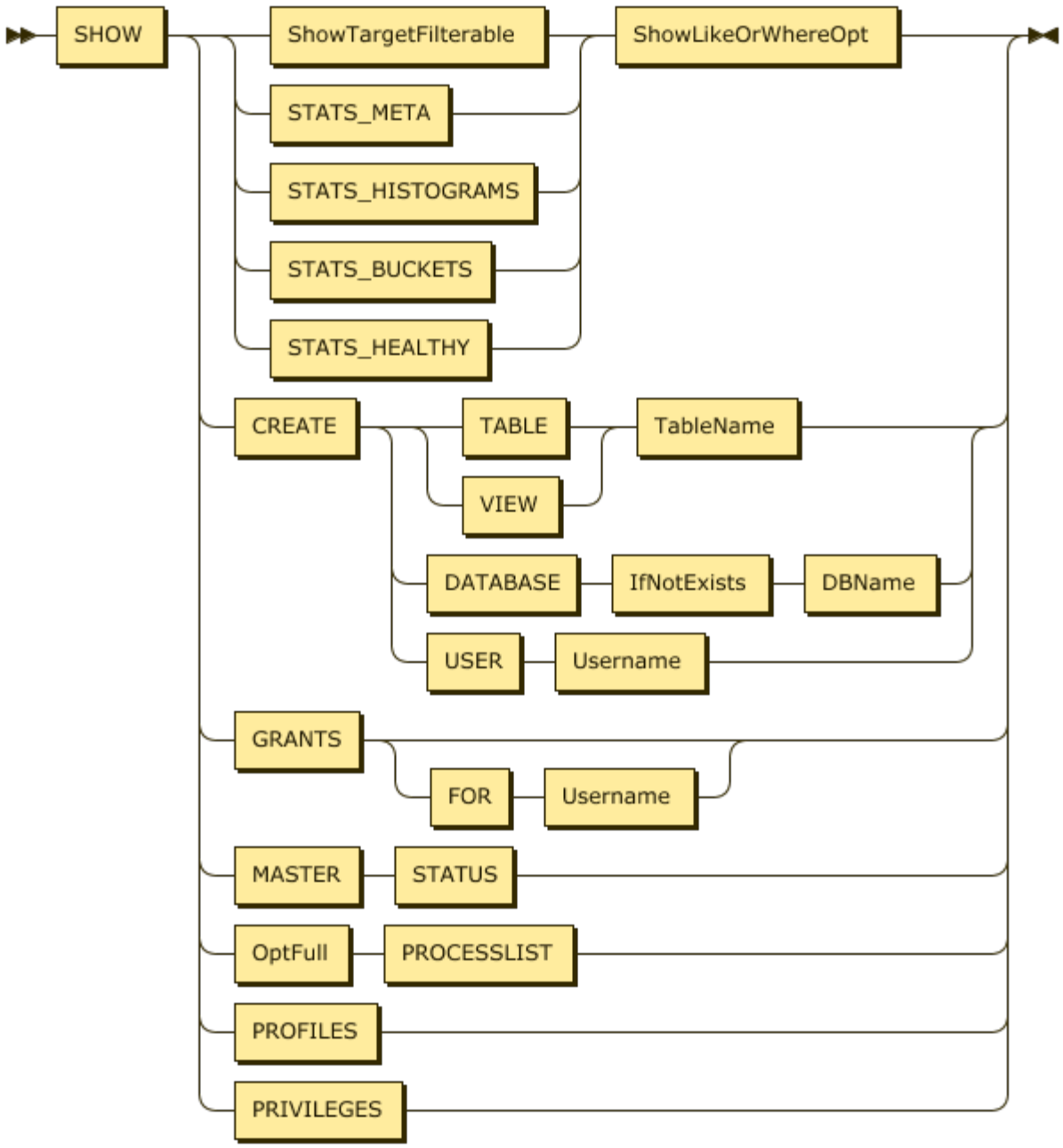


图 243: ShowStmt

TableName:

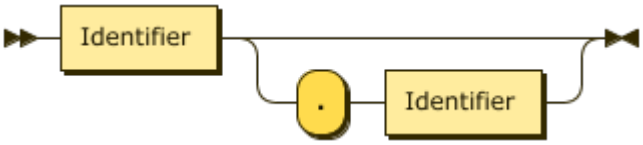


图 244: TableName

4.1.6.67.2 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)
```

4.1.6.67.3 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。

4.1.6.67.4 另请参阅

- CREATE TABLE
- DROP TABLE
- SHOW TABLES
- SHOW COLUMNS FROM

4.1.6.68 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。

4.1.6.68.1 语法图

ShowStmt:

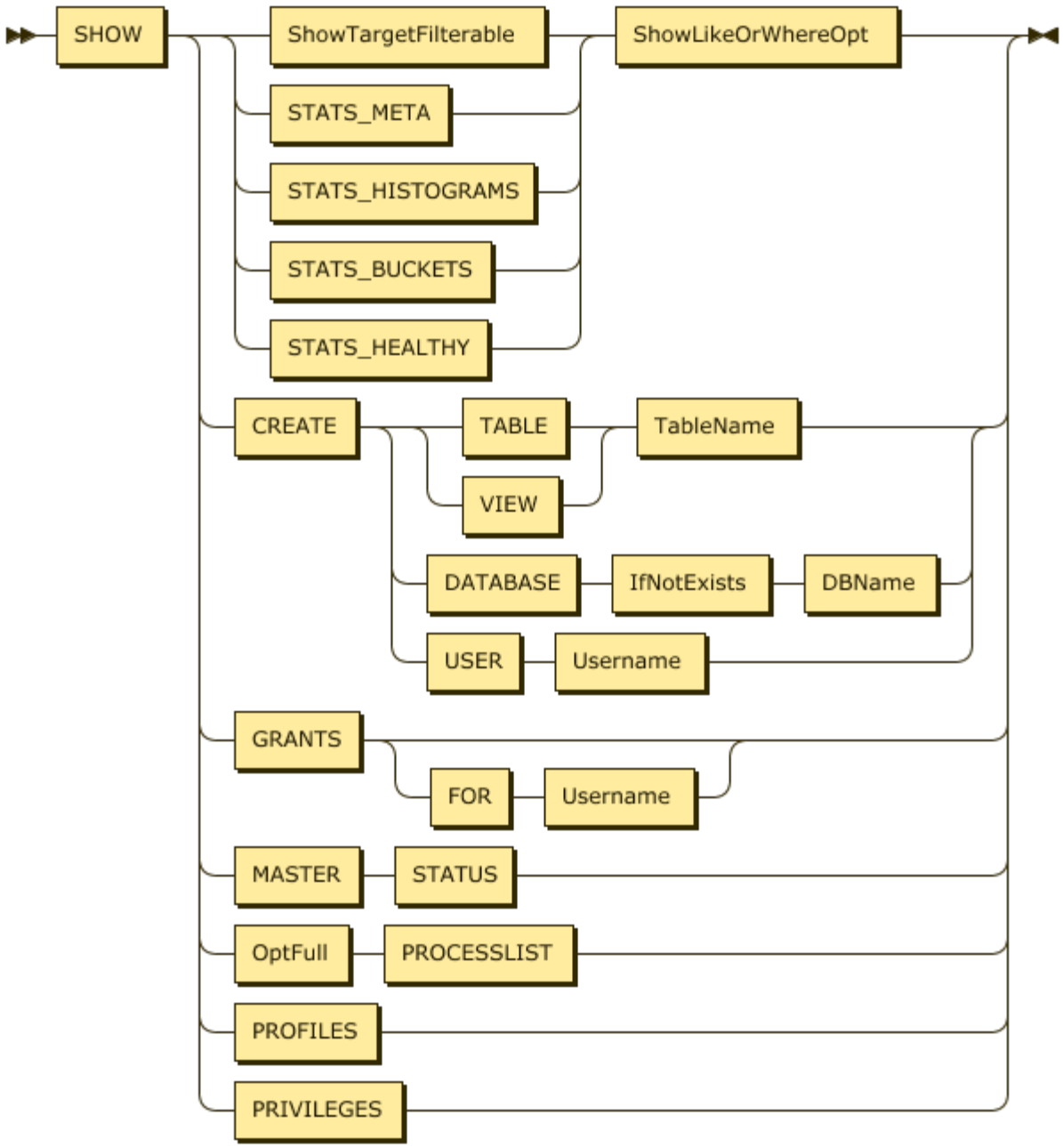


图 245: ShowStmt

Username:

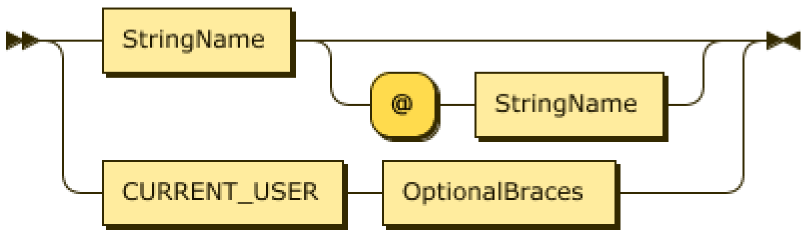


图 246: Username

4.1.6.68.2 示例

```sql
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)
```

4.1.6.68.3 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]。

4.1.6.68.4 另请参阅

- CREATE USER
- SHOW GRANTS
- DROP USER

4.1.6.69 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。

4.1.6.69.1 语法图

ShowStmt:

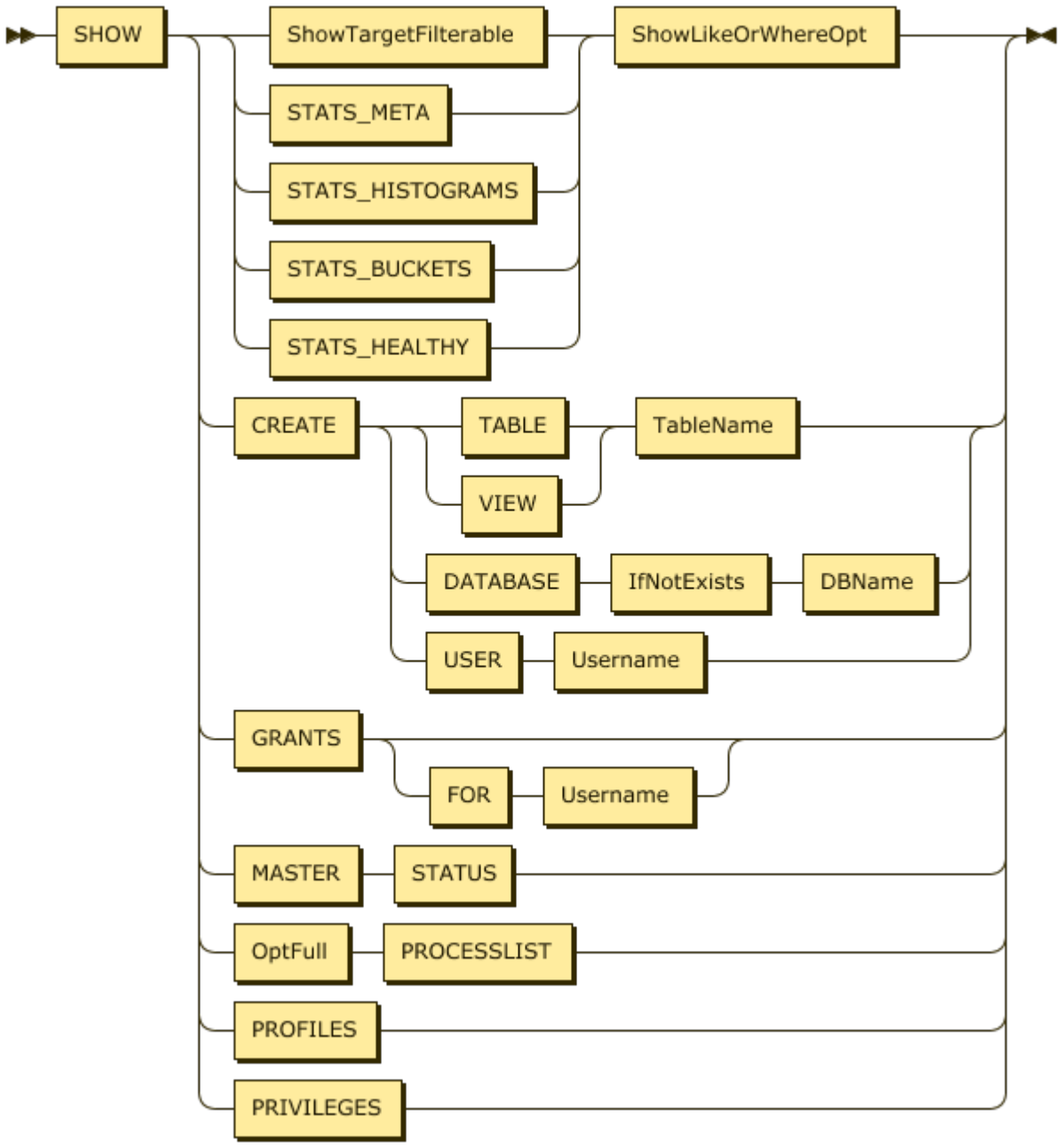


图 247: ShowStmt

ShowTargetFilterable:

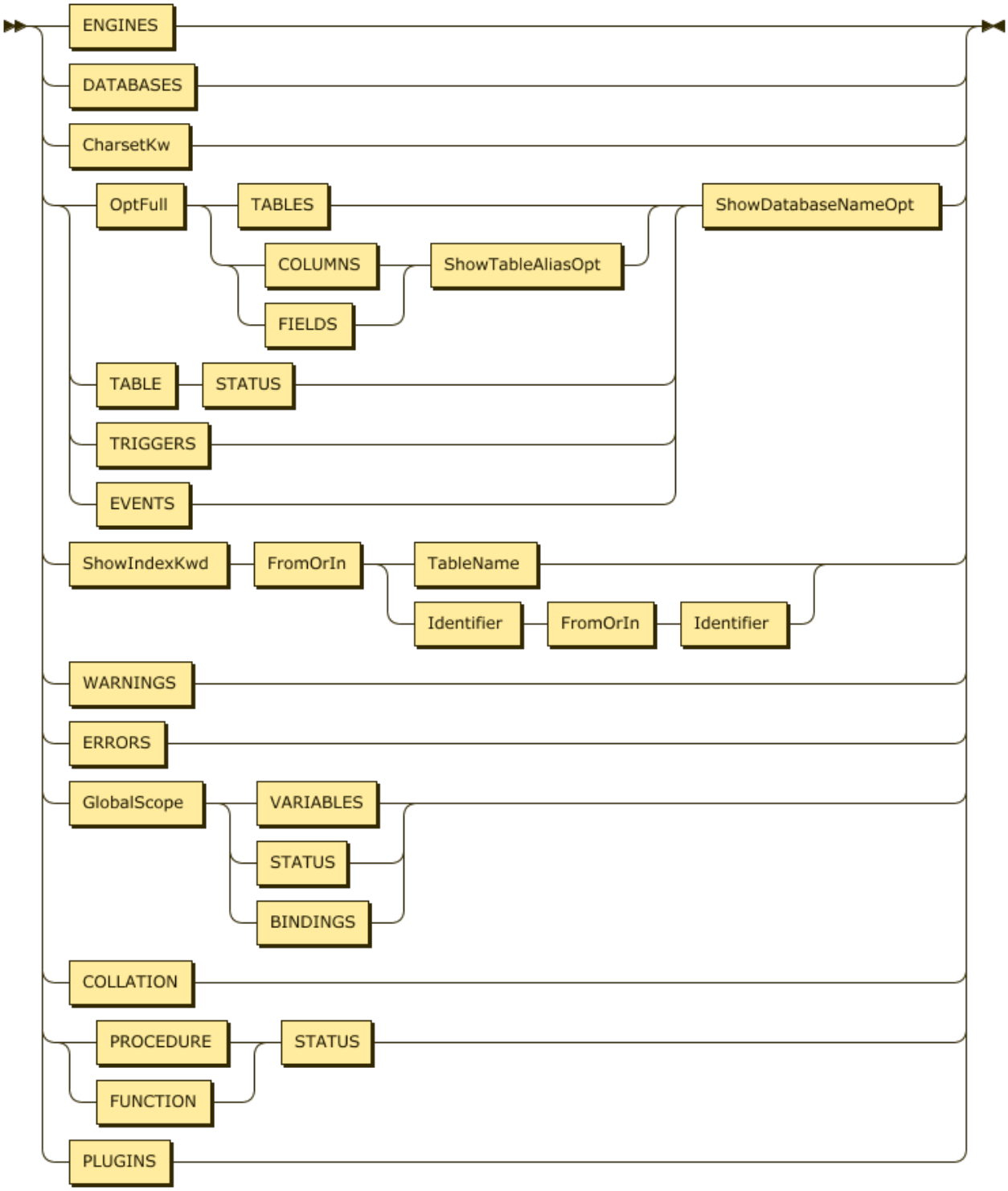


图 248: ShowTargetFilterable

4.1.6.69.2 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)
```

4.1.6.69.3 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。

4.1.6.69.4 另请参阅

- SHOW SCHEMAS
- DROP DATABASE
- CREATE DATABASE

4.1.6.70 SHOW ENGINES

`SHOW ENGINES` 语句仅提供 MySQL 兼容性。

4.1.6.70.1 语法图

```
SHOW ENGINES;
```

4.1.6.70.2 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)
```

4.1.6.70.3 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。

4.1.6.71 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。

4.1.6.71.1 语法图

ShowStmt:

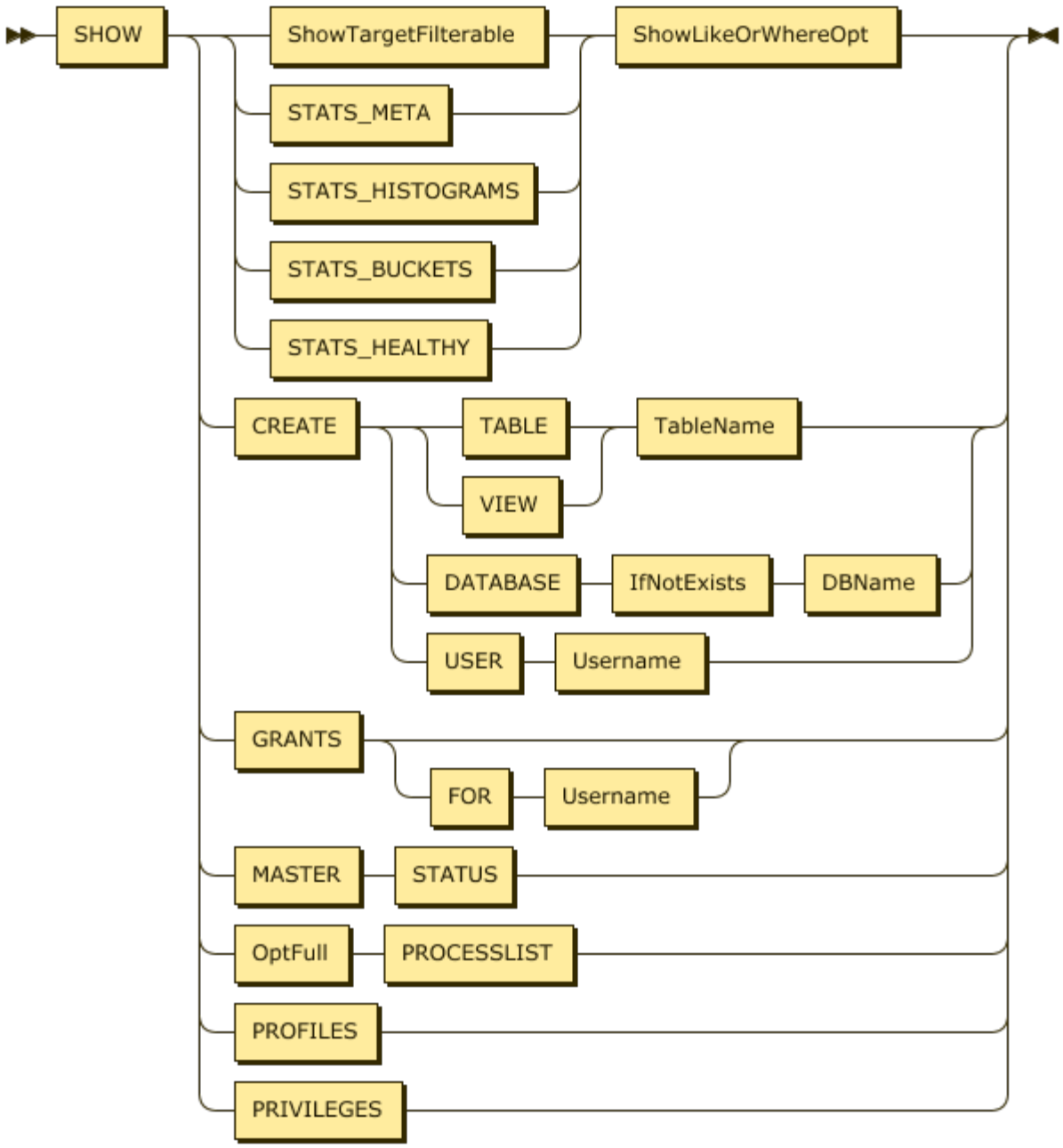


图 249: ShowStmt

ShowTargetFilterable:

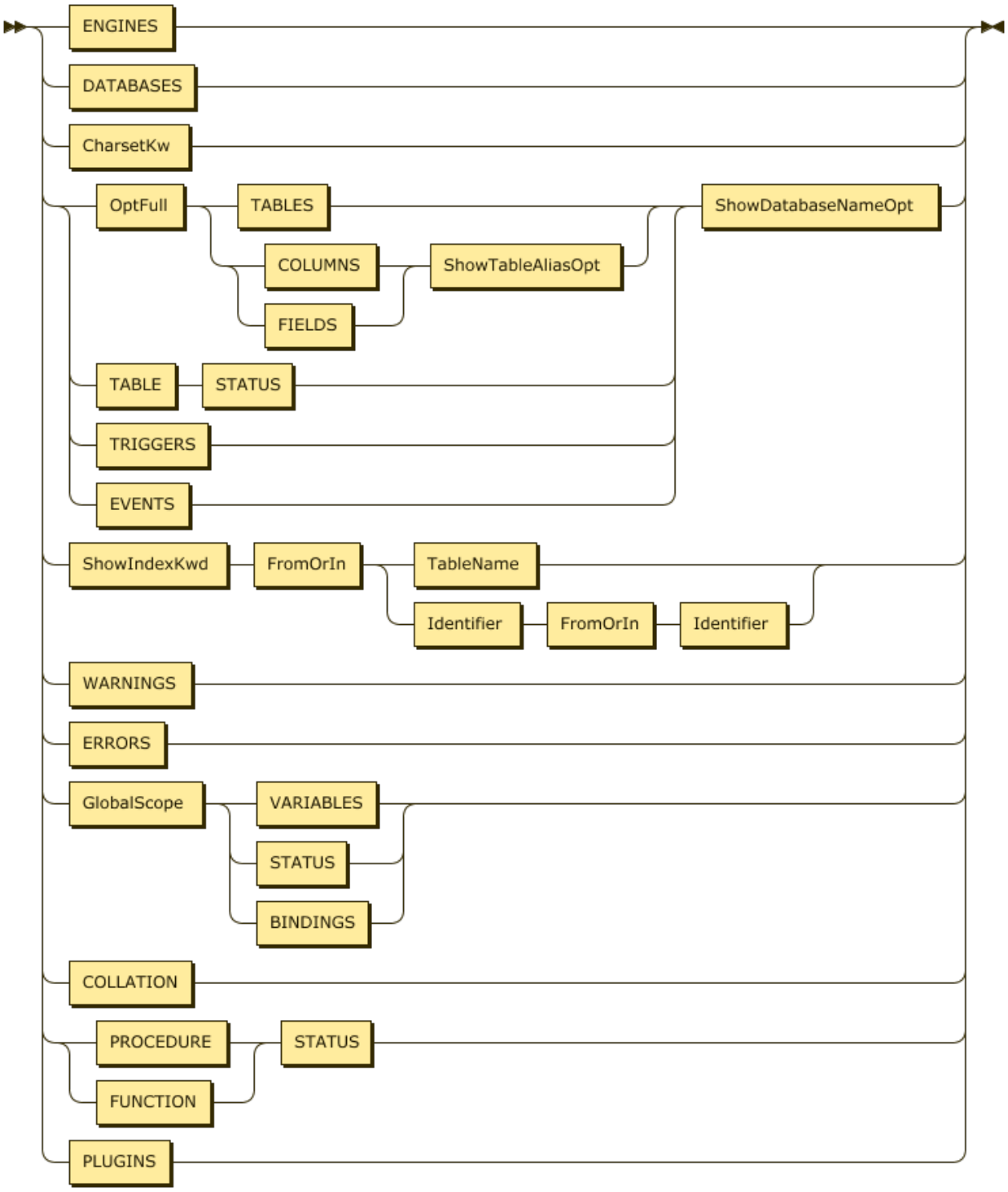


图 250: ShowTargetFilterable

4.1.6.71.2 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)
```

4.1.6.71.3 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。

4.1.6.71.4 另请参阅

- SHOW WARNINGS

4.1.6.72 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。

4.1.6.73 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。

4.1.6.73.1 语法图

ShowStmt:

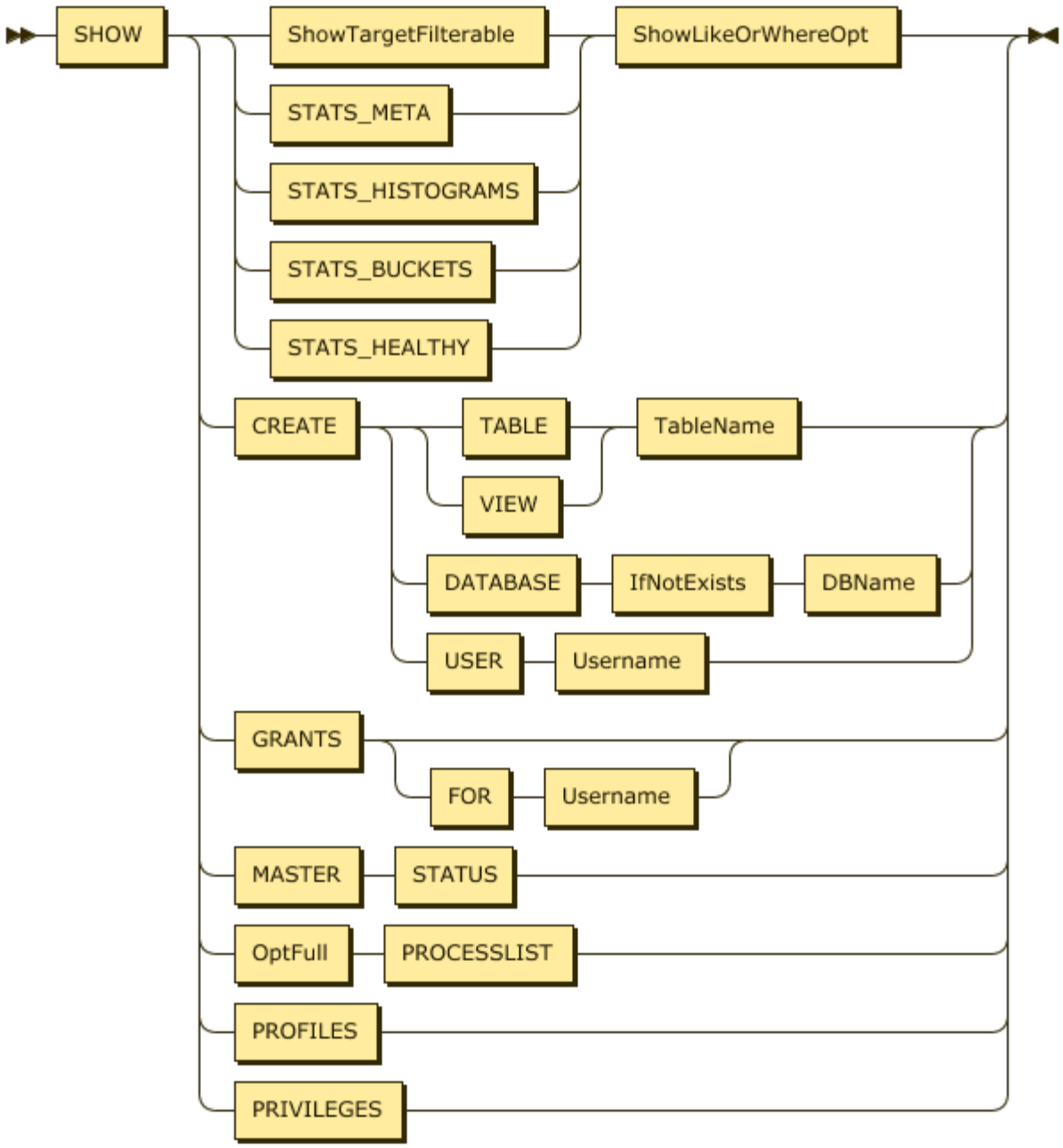


图 251: ShowStmt

Username:

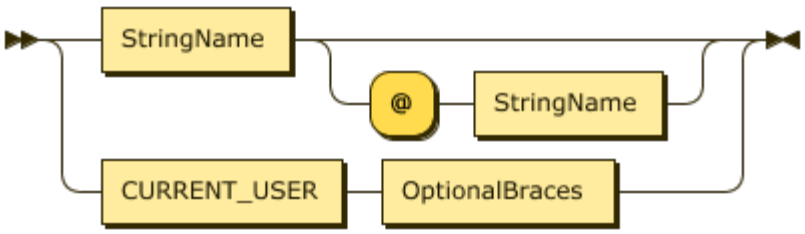


图 252: Username

4.1.6.73.2 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)
```

4.1.6.73.3 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。

4.1.6.73.4 另请参阅

- SHOW CREATE USER
- GRANT

4.1.6.74 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。

4.1.6.74.1 语法图

ShowStmt:

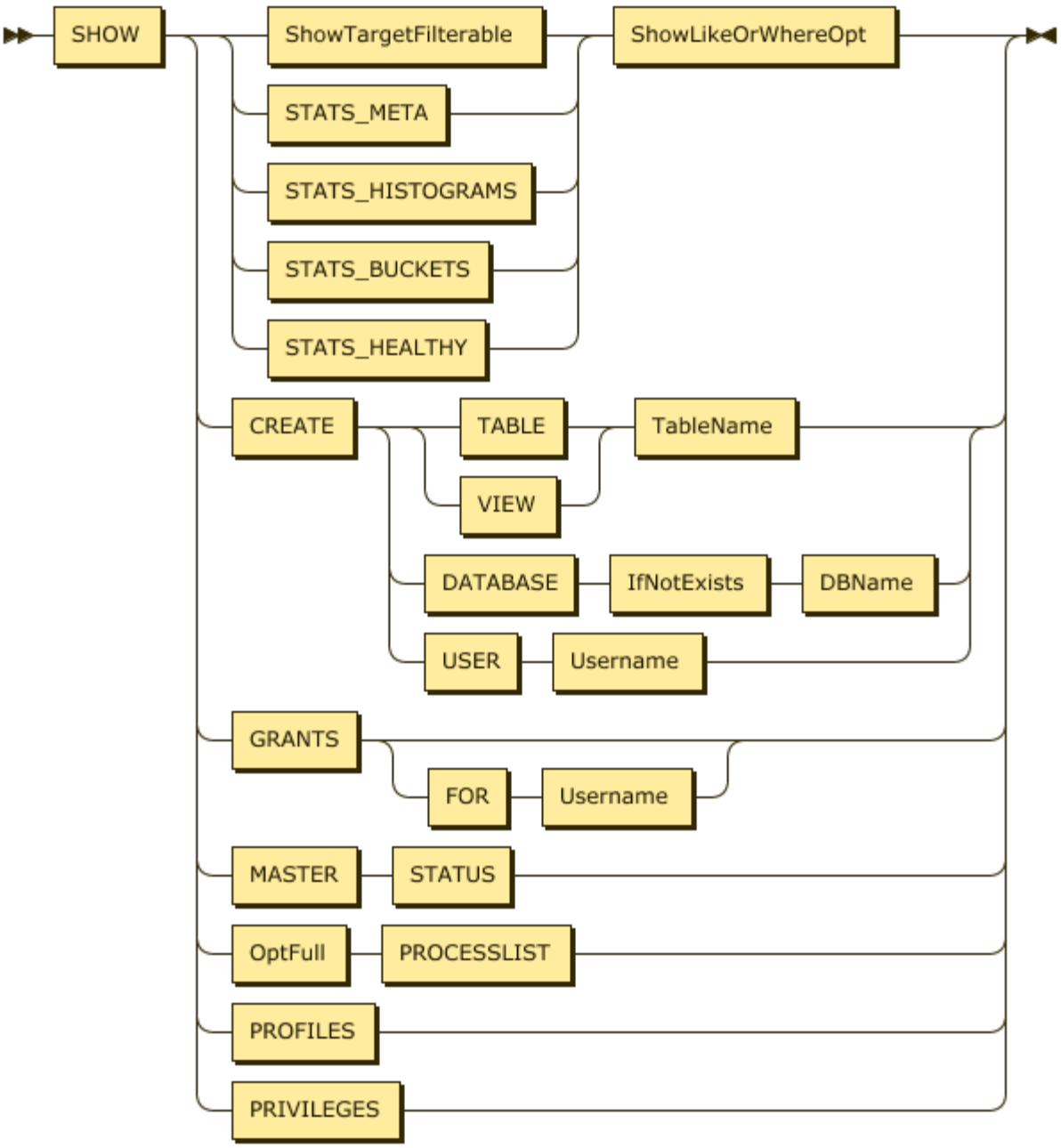


图 253: ShowStmt

ShowTargetFilterable:

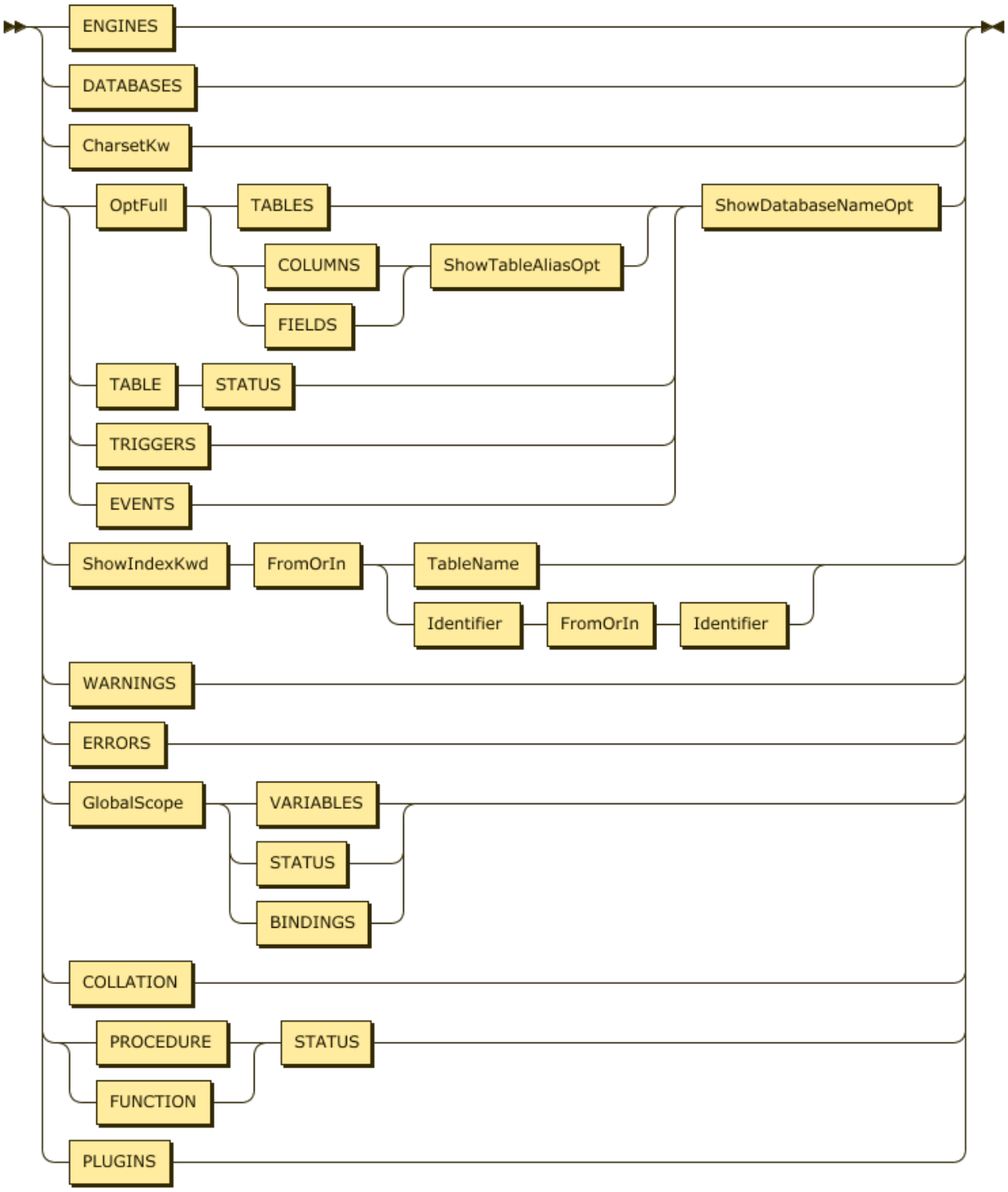


图 254: ShowTargetFilterable

ShowIndexKwd:

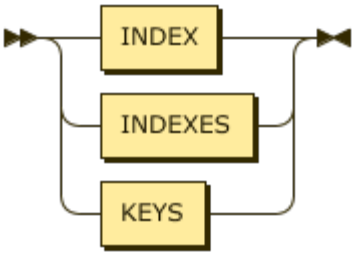


图 255: ShowIndexKwd

FromOrIn:

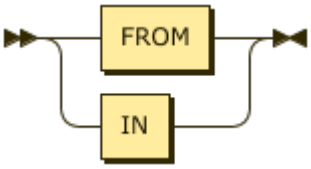


图 256: FromOrIn

TableName:

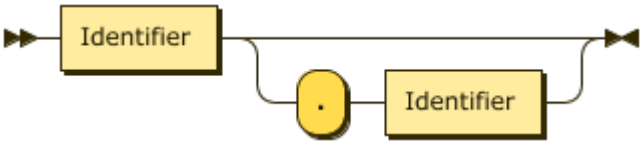


图 257: TableName

4.1.6.74.2 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)
```

4.1.6.74.3 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。

4.1.6.74.4 另请参阅

- SHOW CREATE TABLE
- DROP INDEX
- CREATE INDEX

4.1.6.75 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。

4.1.6.76 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。

4.1.6.77 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。

4.1.6.77.1 语法图

ShowStmt:

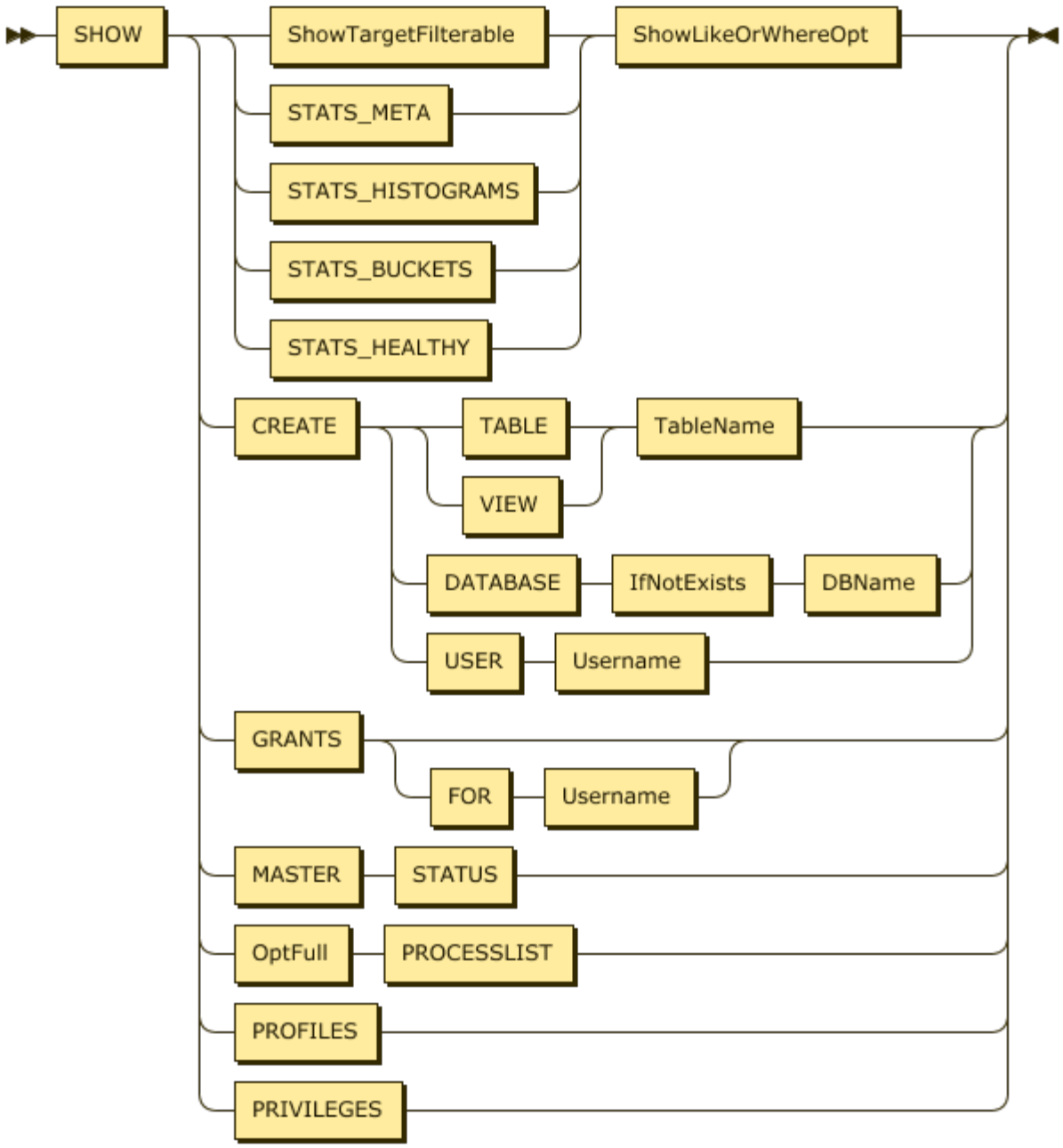


图 258: ShowStmt

4.1.6.77.2 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)
```

4.1.6.77.3 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。

4.1.6.77.4 另请参阅

- SHOW GRANTS
- GRANT <privileges>

4.1.6.78 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。

4.1.6.78.1 语法图

ShowStmt:

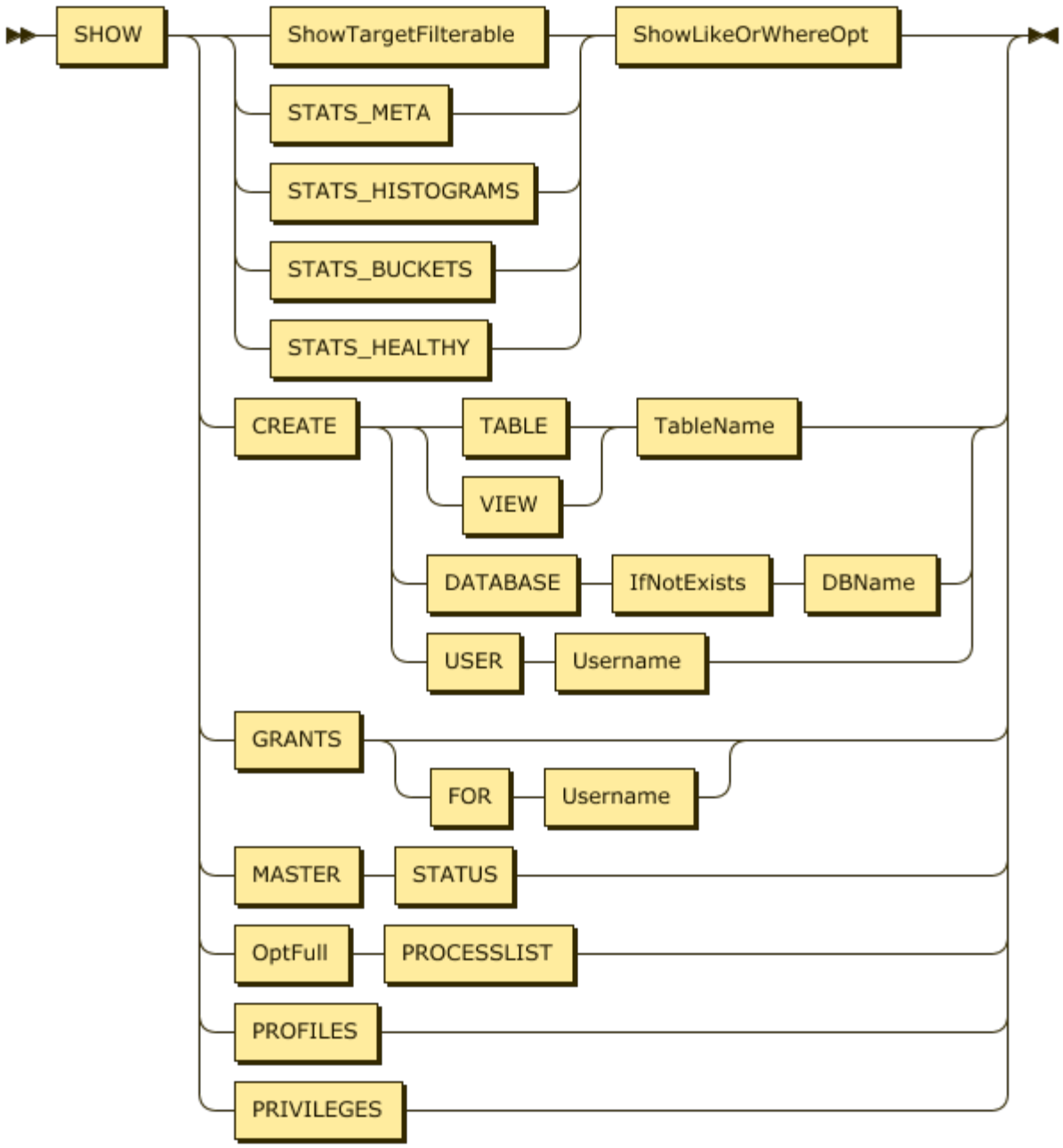


图 259: ShowStmt

OptFull:

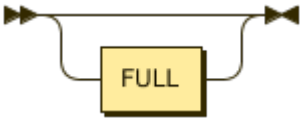


图 260: OptFull

4.1.6.78.2 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)
```

4.1.6.78.3 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。

4.1.6.78.4 另请参阅

- KILL [TIDB]

4.1.6.79 SHOW SCHEMAS

SHOW SCHEMAS 语句是SHOW DATABASES 的别名。包含该语句提供了 MySQL 兼容性。。

4.1.6.80 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。

4.1.6.80.1 语法图

ShowStmt:

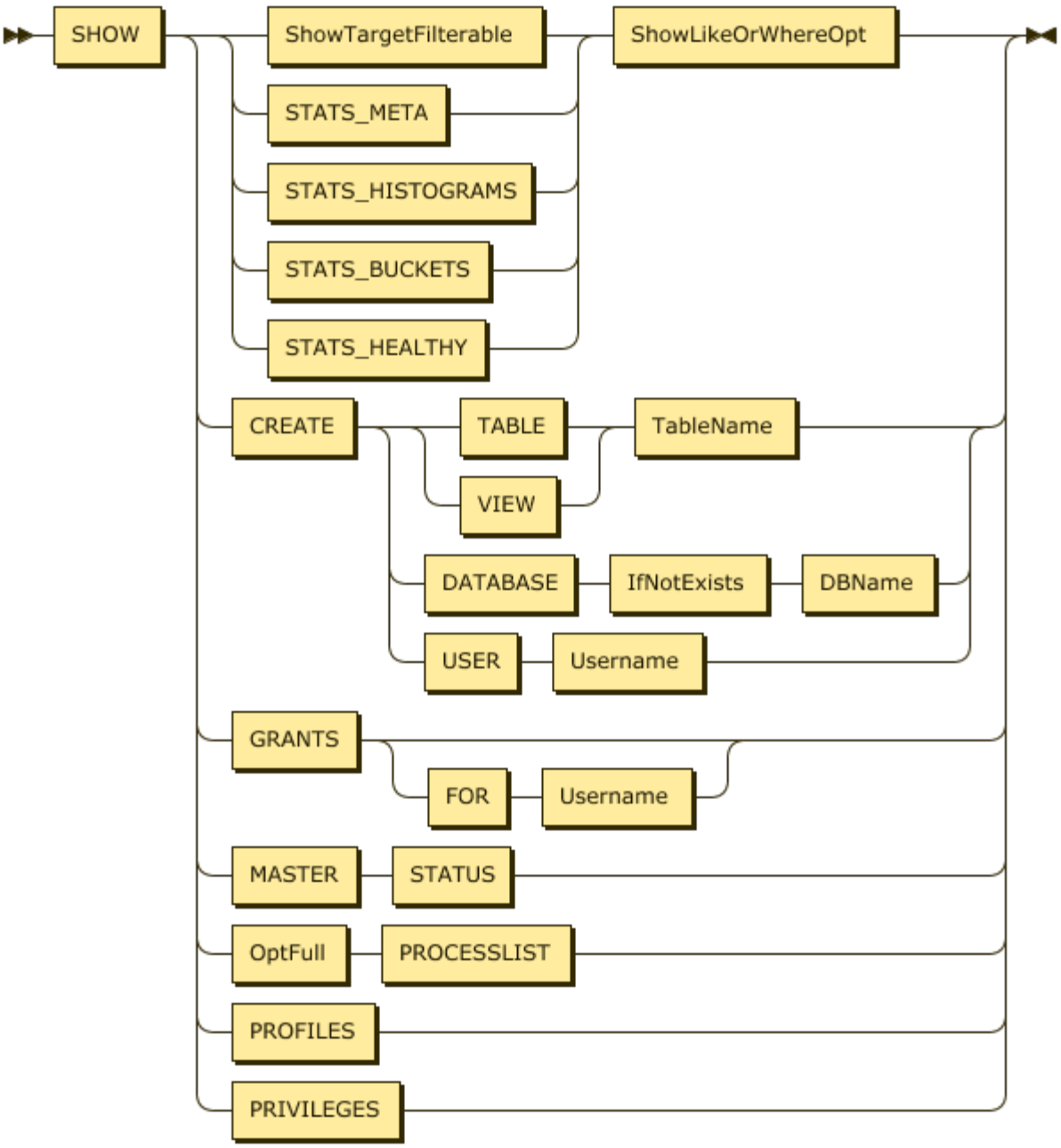


图 261: ShowStmt

ShowTargetFilterable:

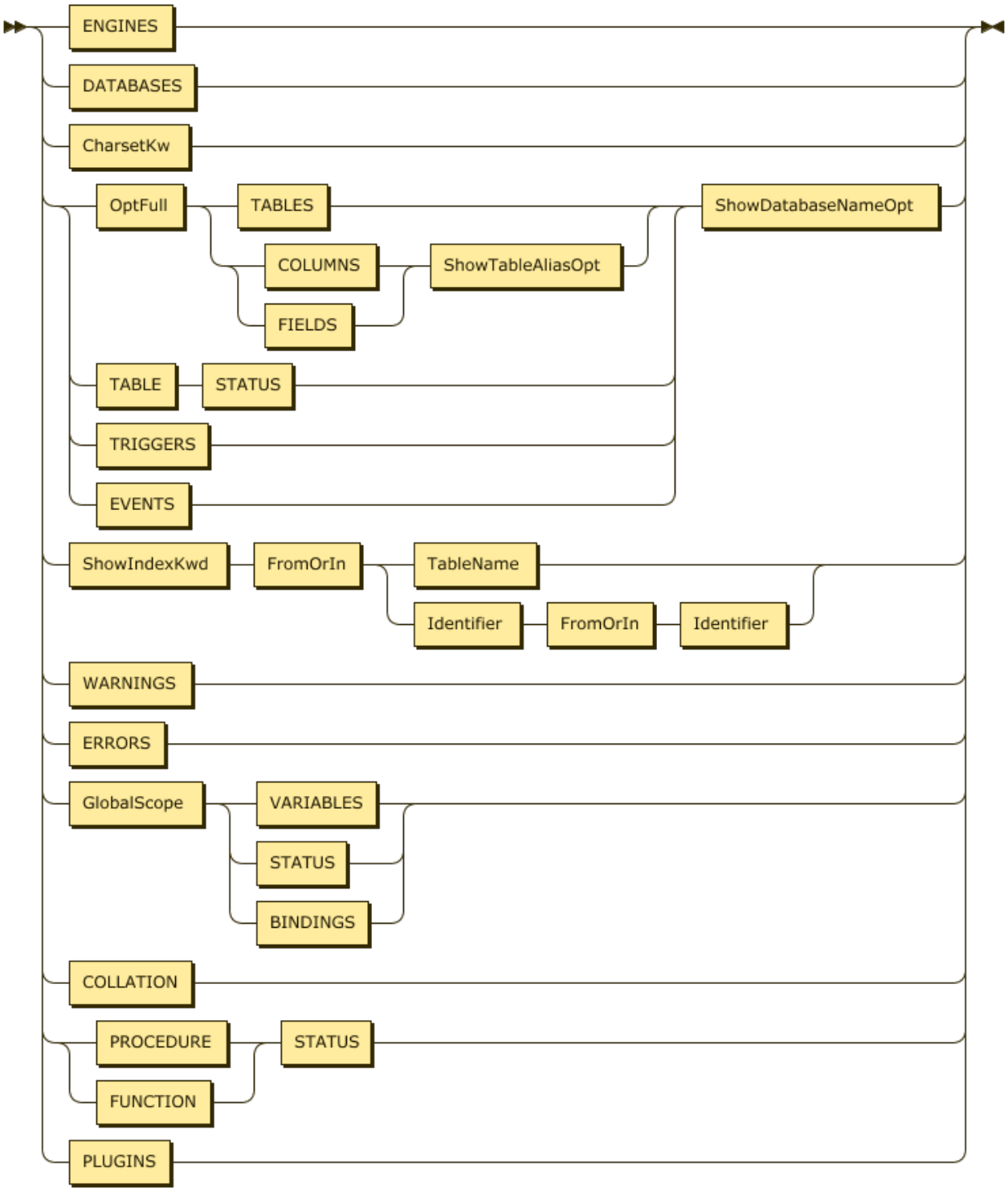


图 262: ShowTargetFilterable

GlobalScope:

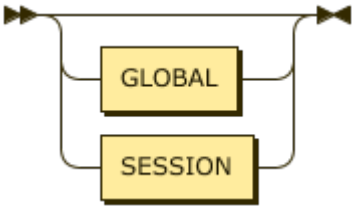


图 263: GlobalScope

4.1.6.80.2 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)
```

4.1.6.80.3 MySQL 兼容性

`SHOW [GLOBAL|SESSION] STATUS` 语句仅用于提供 MySQL 兼容性。

4.1.6.80.4 另请参阅

- FLUSH STATUS

4.1.6.81 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。

4.1.6.81.1 语法图

ShowStmt:

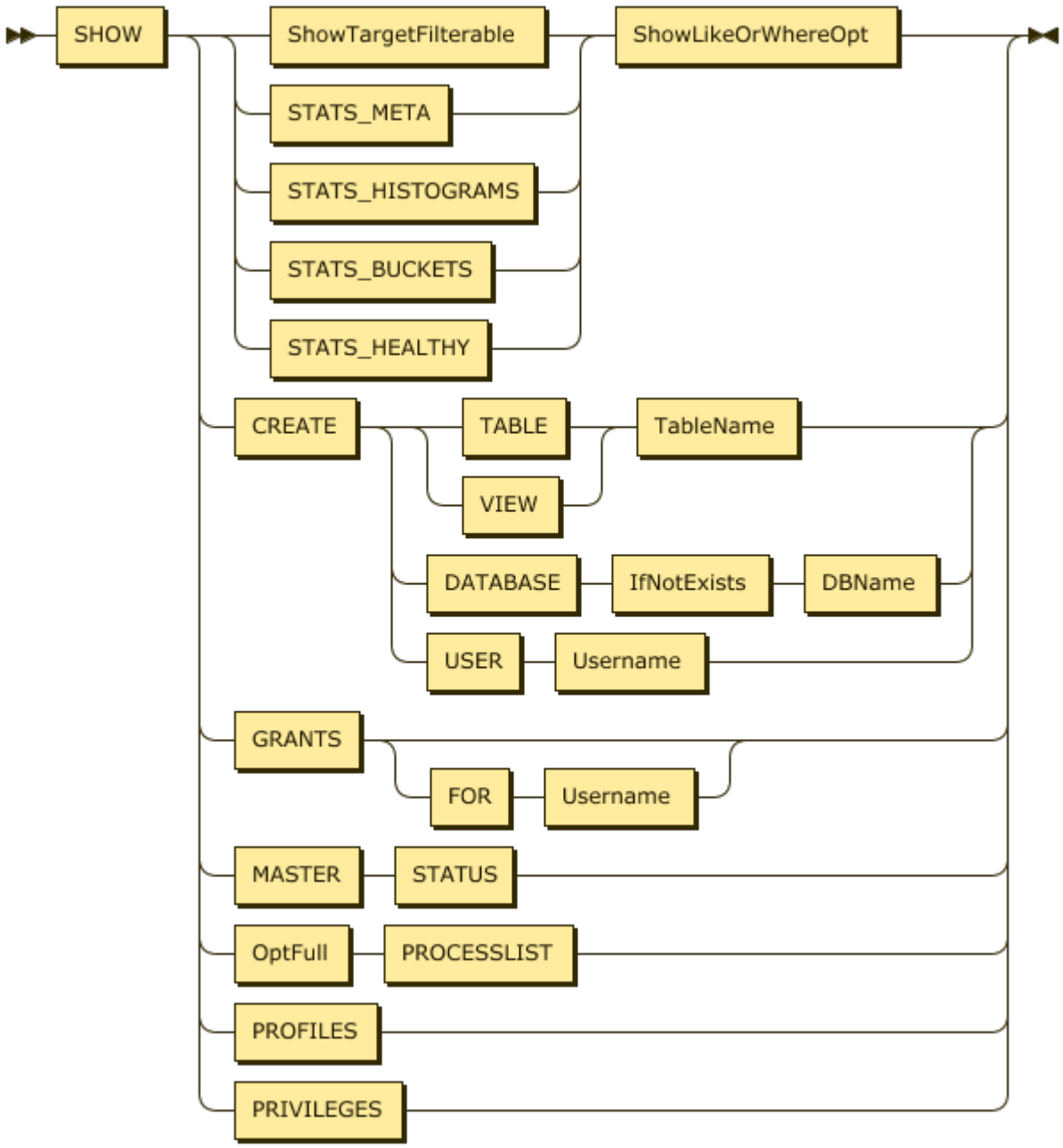


图 264: ShowStmt

ShowTargetFilterable:

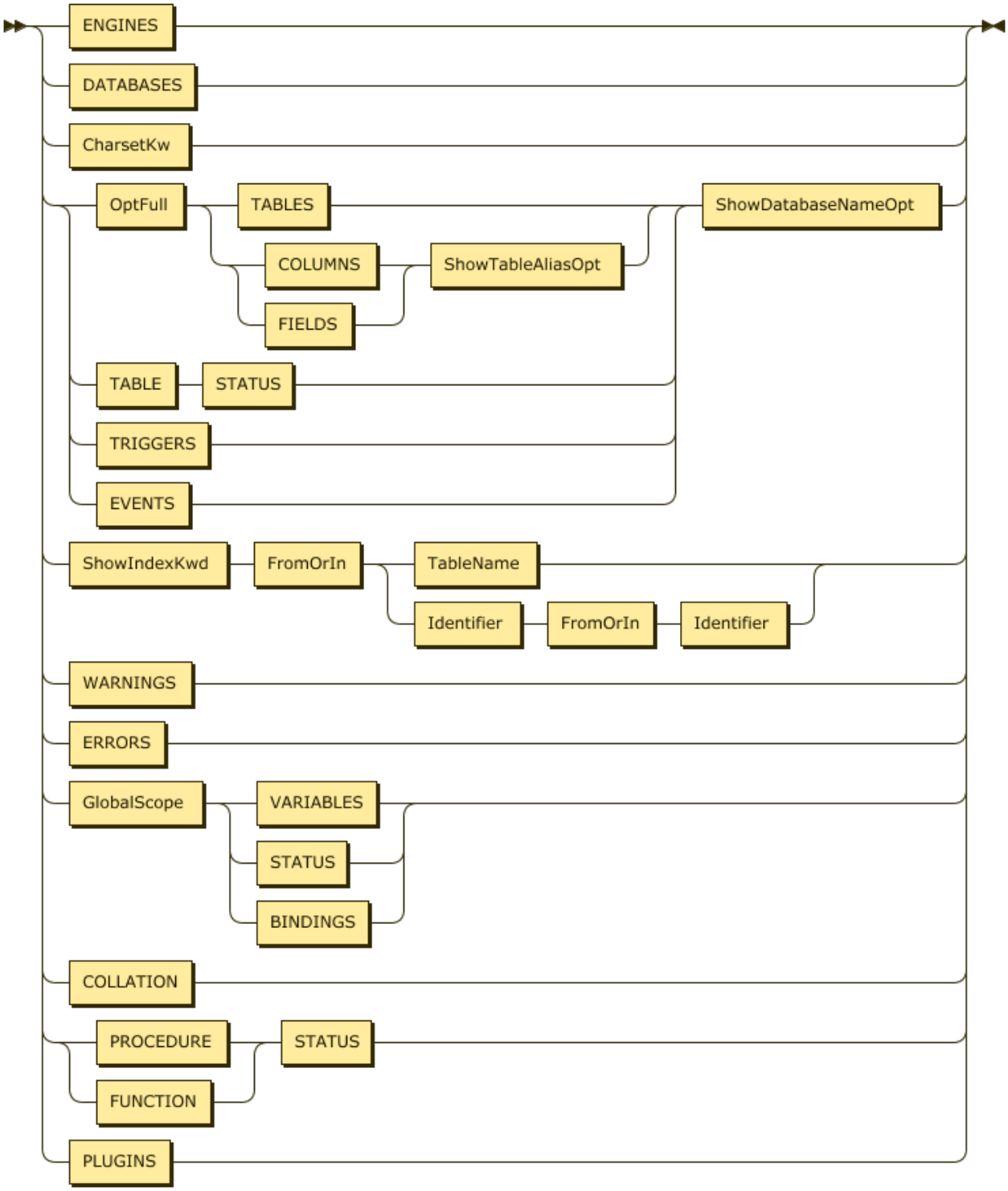


图 265: ShowTargetFilterable

ShowDatabaseNameOpt:

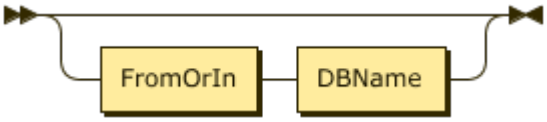


图 266: ShowDatabaseNameOpt

4.1.6.81.2 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        |
| gc_delete_range      |
| gc_delete_range_done |
| help_topic           |
| role_edges           |
```

```
| stats_buckets         |
| stats_feedback        |
| stats_histograms      |
| stats_meta            |
| tables_priv           |
| tidb                  |
| user                  |
+-----------------------+
16 rows in set (0.00 sec)
```

4.1.6.81.3 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。

4.1.6.81.4 另请参阅

- CREATE TABLE
- DROP TABLE
- SHOW CREATE TABLE

4.1.6.82 SHOW TABLE REGIONS

SHOW TABLE REGIONS 语句用于显示 TiDB 中某个表的 Region 信息。

4.1.6.82.1 语法图

```
SHOW TABLE [table_name] REGIONS [WhereClauseOptional];

SHOW TABLE [table_name] INDEX [index_name] REGIONS [WhereClauseOptional];
```

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。

4.1.6.82.2 示例

创建一个示例表，并在若干 Region 中填充足够的数据量：

```
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 (103, 109, 113, 2) 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
```

4.1.6.82.3 MySQL 兼容性

SHOW TABLE REGIONS 语句是 TiDB 对 MySQL 语法的扩展。

4.1.6.82.4 另请参阅

- SPLIT REGION
- CREATE TABLE

4.1.6.83 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。

4.1.6.83.1 语法图

ShowStmt:

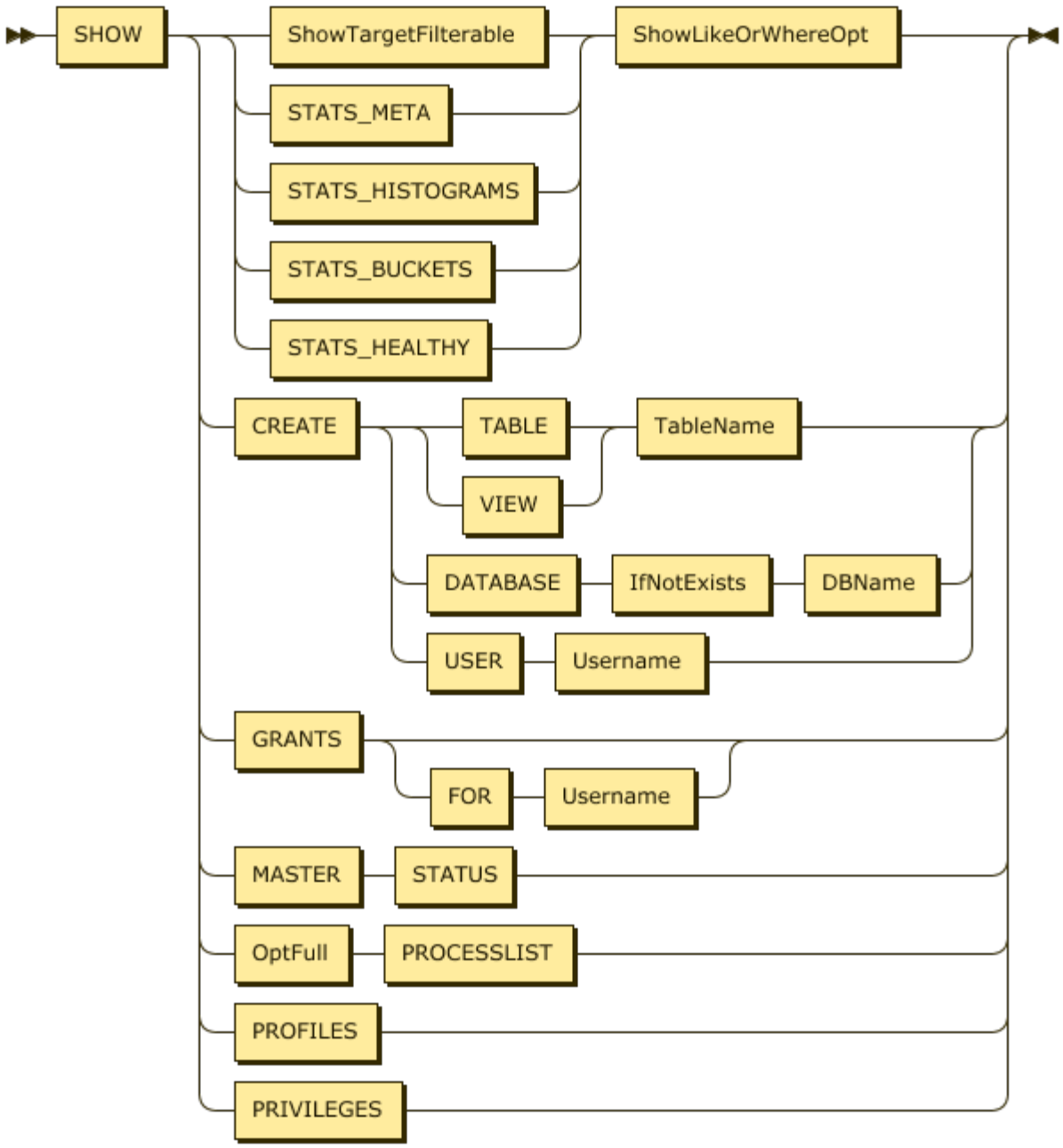


图 267: ShowStmt

ShowTargetFilterable:

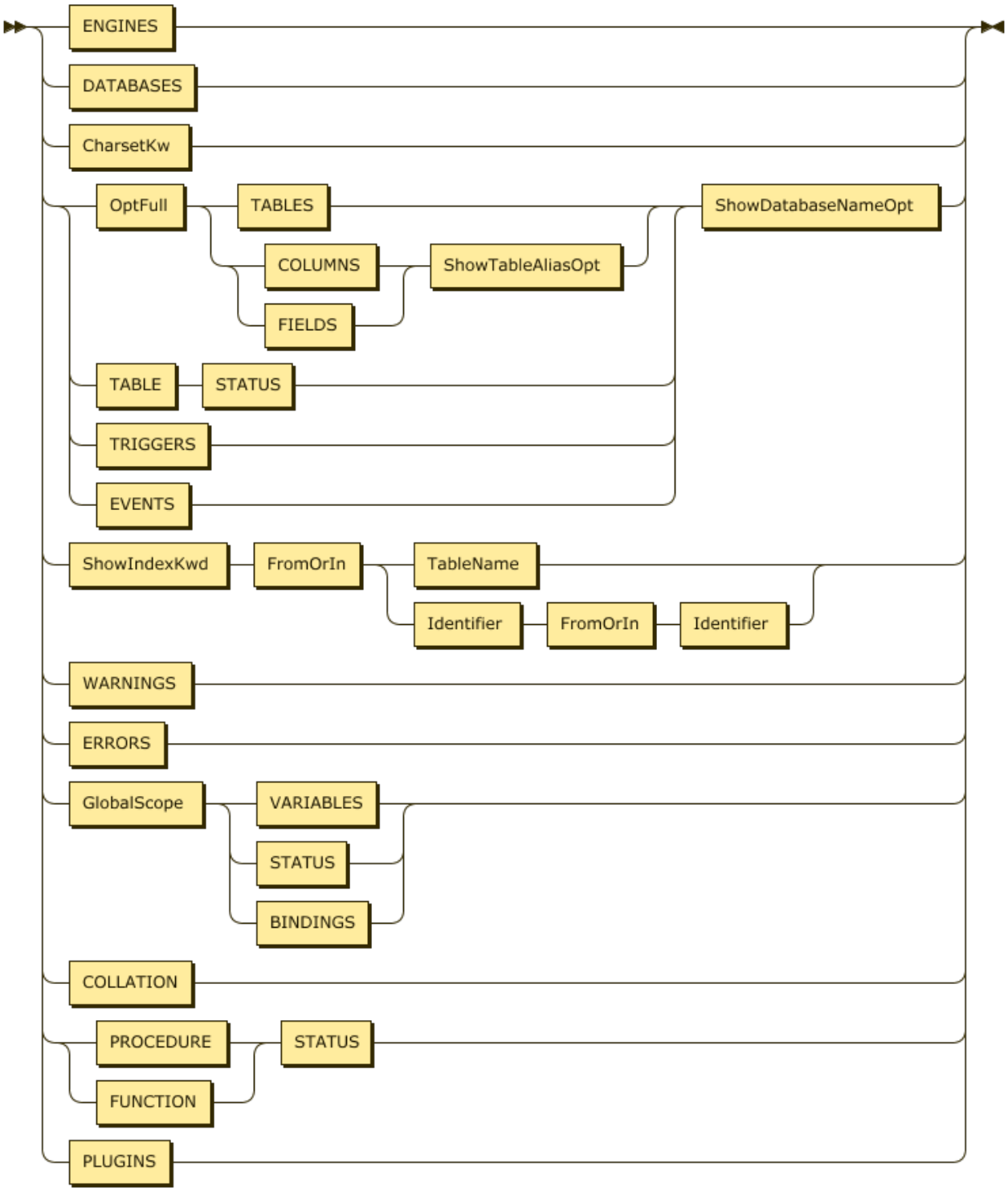


图 268: ShowTargetFilterable

ShowDatabaseNameOpt:

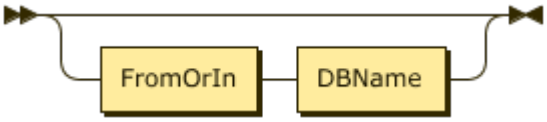


图 269: ShowDatabaseNameOpt

4.1.6.83.2 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)
```

4.1.6.83.3 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。

4.1.6.83.4 另请参阅

- SHOW TABLES
- CREATE TABLE
- DROP TABLE
- SHOW CREATE TABLE

4.1.6.84 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`。

4.1.6.84.1 语法图

ShowStmt:

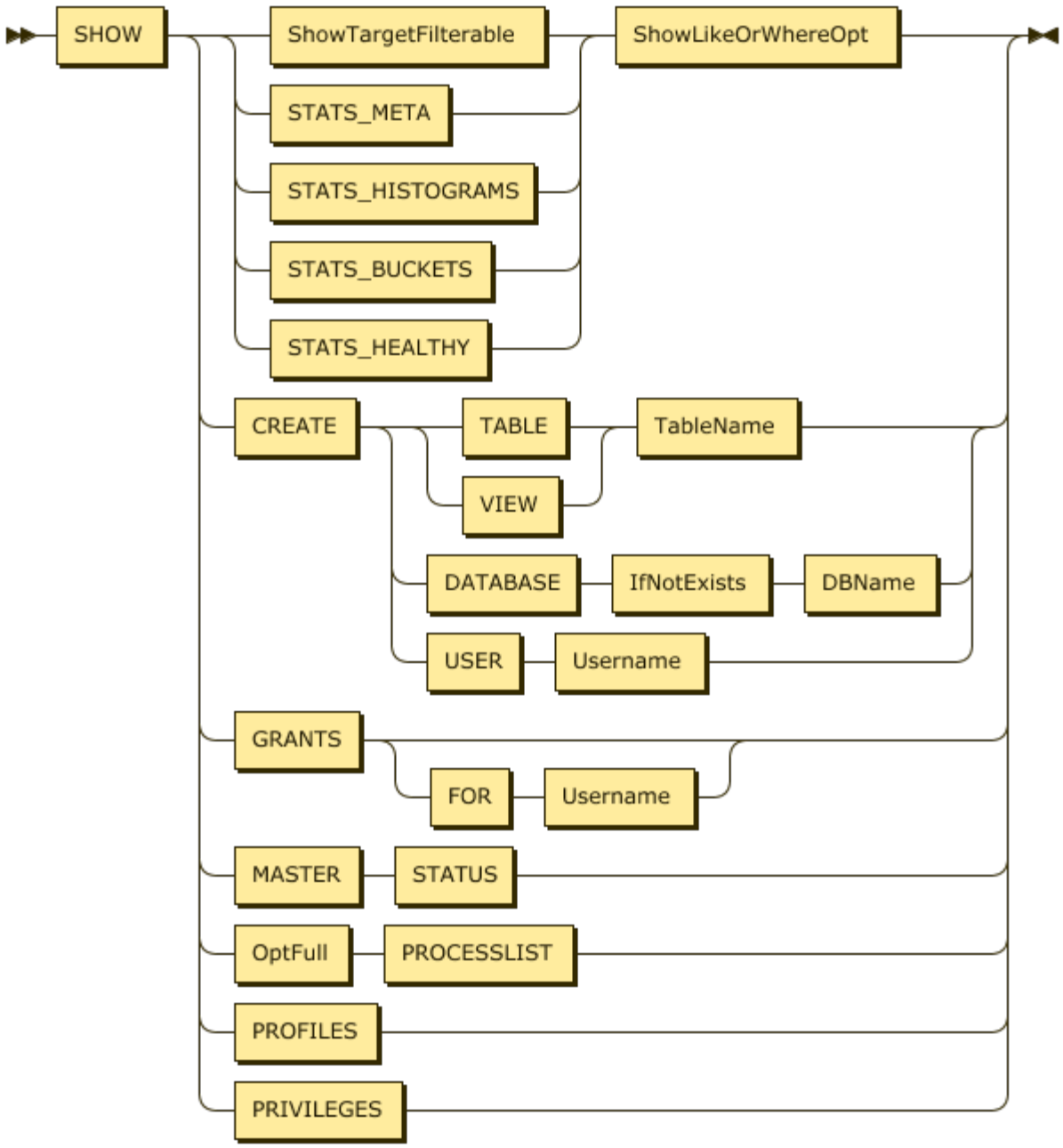


图 270: ShowStmt

ShowTargetFilterable:

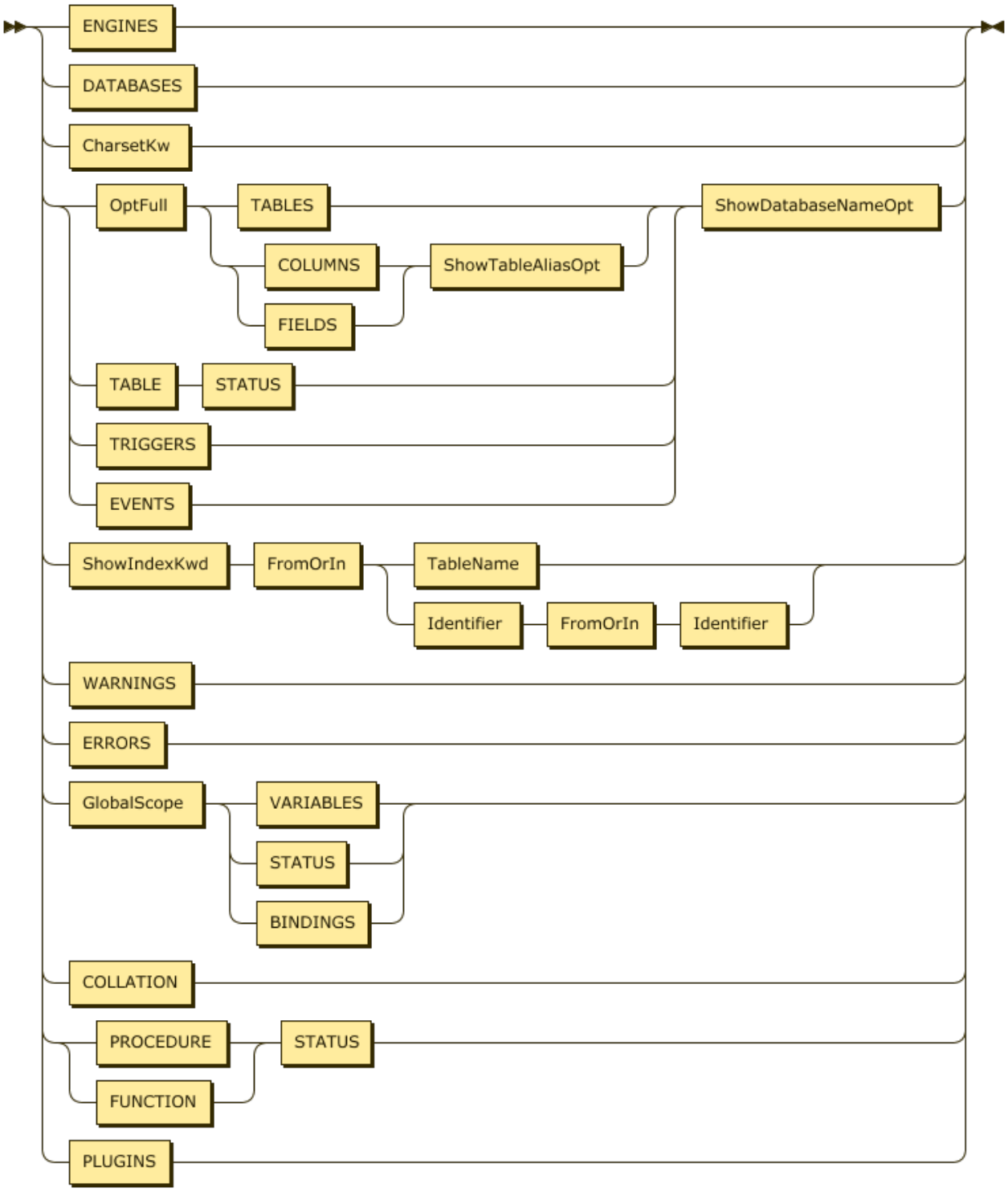


图 271: ShowTargetFilterable

GlobalScope:

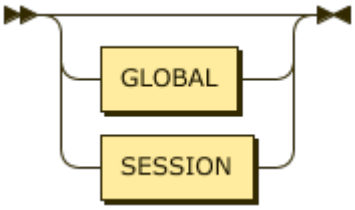


图 272: GlobalScope

4.1.6.84.2 示例

```
SHOW GLOBAL VARIABLES LIKE 'tidb%';
```

```
+-----------------------------------+---------------+
| Variable_name                     | Value         |
+-----------------------------------+---------------+
| tidb_retry_limit                  | 10            |
| tidb_enable_streaming             | 0             |
| tidb_hashagg_final_concurrency    | 4             |
| tidb_disable_txn_auto_retry       | 1             |
| tidb_mem_quota_query              | 34359738368   |
| tidb_skip_isolation_level_check   | 0             |
| tidb_ddl_reorg_batch_size         | 1024          |
| tidb_constraint_check_in_place    | 0             |
| tidb_current_ts                   | 0             |
| tidb_opt_insubq_to_join_and_agg   | 1             |
| tidb_enable_window_function       | 1             |
| tidb_ddl_error_count_limit        | 512           |
| tidb_checksum_table_concurrency   | 4             |
| tidb_config                       |               |
| tidb_batch_insert                 | 0             |
| tidb_opt_join_reorder_threshold   | 0             |
| tidb_auto_analyze_end_time        | 23:59 +0000   |
| tidb_index_lookup_size            | 20000         |
| tidb_projection_concurrency       | 4             |
| tidb_opt_correlation_threshold    | 0.9           |
| tidb_enable_table_partition       | auto          |
| tidb_wait_split_region_timeout    | 300           |
| tidb_skip_utf8_check              | 0             |
| tidb_dml_batch_size               | 20000         |
| tidb_backoff_weight               | 2             |
| tidb_txn_mode                     |               |
| tidb_mem_quota_indexlookupjoin    | 34359738368   |
| tidb_hashagg_partial_concurrency  | 4             |
| tidb_enable_radix_join            | 0             |
| tidb_mem_quota_topn               | 34359738368   |
```

```
| tidb_hash_join_concurrency          | 5             |
| tidb_max_chunk_size                 | 1024          |
| tidb_mem_quota_indexlookupreader    | 34359738368   |
| tidb_expensive_query_time_threshold | 60            |
| tidb_enable_cascades_planner        | 0             |
| tidb_mem_quota_mergejoin            | 34359738368   |
| tidb_auto_analyze_ratio             | 0.5           |
| tidb_index_lookup_concurrency       | 4             |
| tidb_force_priority                 | NO_PRIORITY   |
| tidb_ddl_reorg_priority             | PRIORITY_LOW  |
| tidb_index_lookup_join_concurrency  | 4             |
| tidb_index_join_batch_size          | 25000         |
| tidb_index_serial_scan_concurrency  | 1             |
| tidb_mem_quota_nestedloopapply      | 34359738368   |
| tidb_auto_analyze_start_time        | 00:00 +0000   |
| tidb_snapshot                       |               |
| tidb_low_resolution_tso             | 0             |
| tidb_batch_commit                   | 0             |
| tidb_init_chunk_size                | 32            |
| tidb_build_stats_concurrency        | 4             |
| tidb_backoff_lock_fast              | 100           |
| tidb_optimizer_selectivity_level    | 0             |
| tidb_batch_delete                   | 0             |
| tidb_distsql_scan_concurrency       | 15            |
| tidb_scatter_region                 | 0             |
| tidb_opt_correlation_exp_factor     | 1             |
| tidb_ddl_reorg_worker_cnt           | 16            |
| tidb_wait_split_region_finish       | 1             |
| tidb_mem_quota_sort                 | 34359738368   |
| tidb_general_log                    | 0             |
| tidb_opt_write_row_id               | 0             |
| tidb_check_mb4_value_in_utf8        | 1             |
| tidb_enable_fast_analyze            | 0             |
| tidb_slow_query_file                | tidb-slow.log |
| tidb_slow_log_threshold             | 300           |
| tidb_opt_agg_push_down              | 0             |
| tidb_mem_quota_hashjoin             | 34359738368   |
| tidb_query_log_max_len              | 2048          |
+-------------------------------------+---------------+
68 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)
```

4.1.6.84.3 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。

4.1.6.84.4 另请参阅

- `SET [GLOBAL|SESSION]`

4.1.6.85 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。

4.1.6.85.1 语法图

ShowStmt:

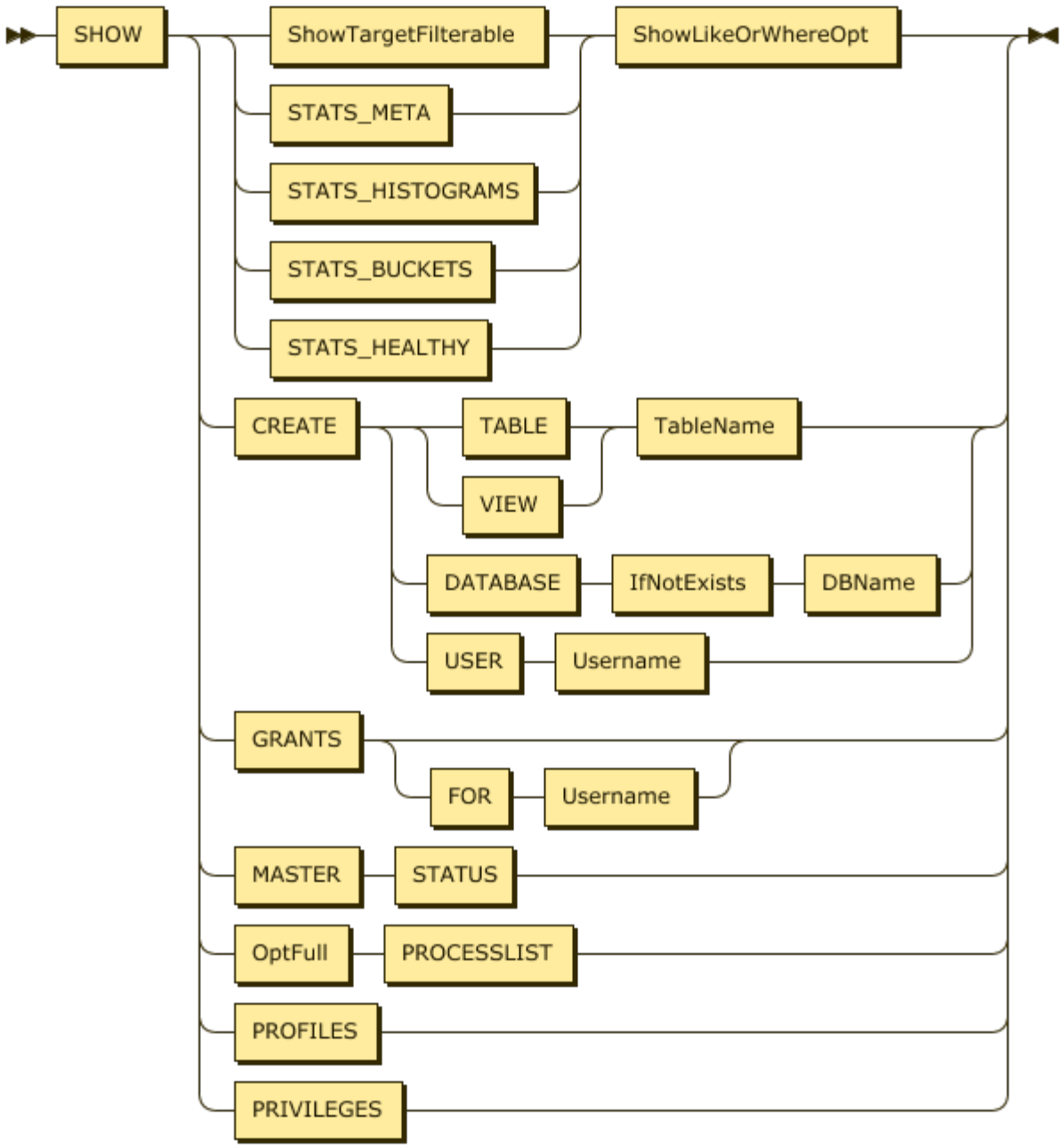


图 273: ShowStmt

ShowTargetFilterable:

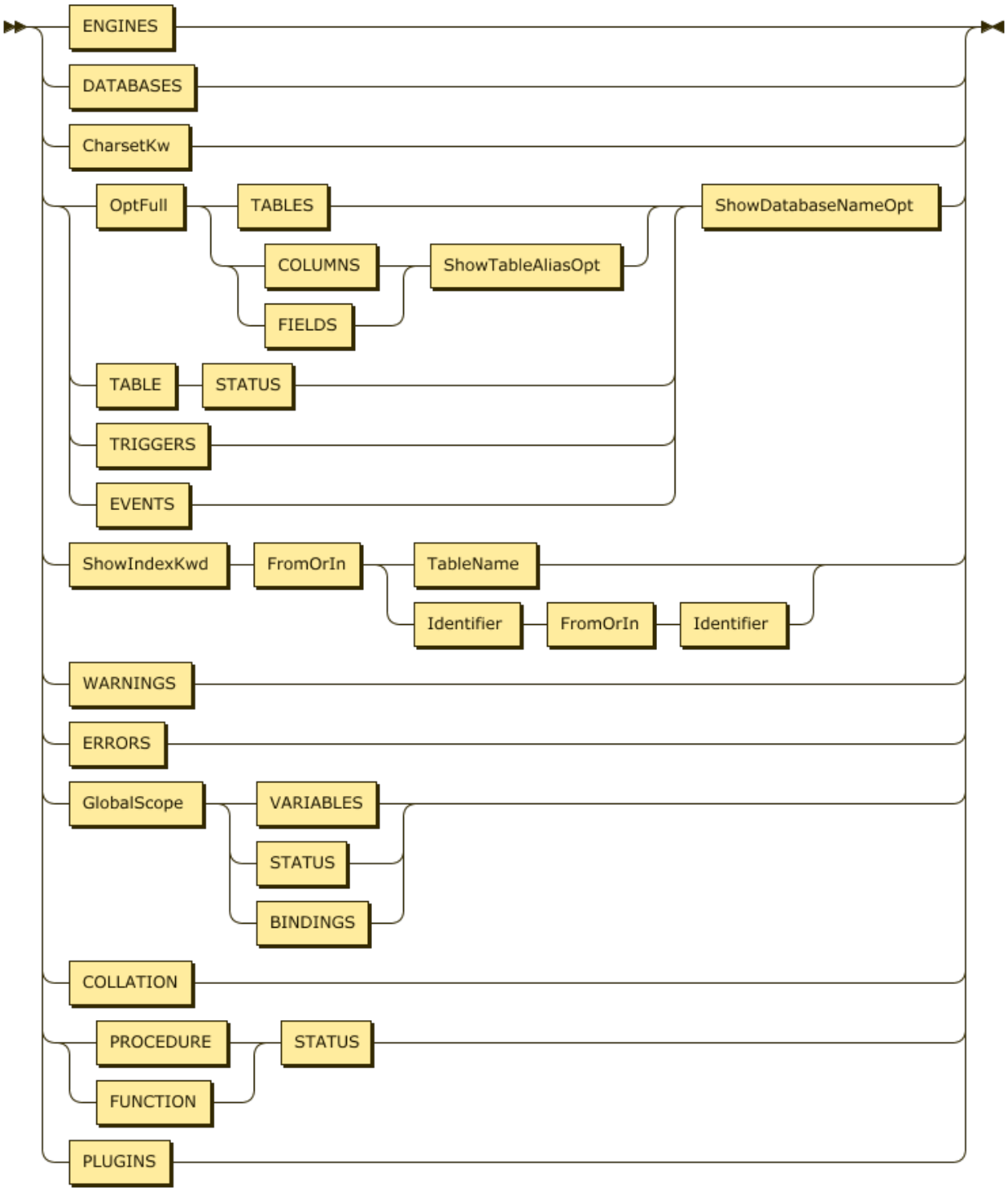


图 274: ShowTargetFilterable

4.1.6.85.2 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)
```

4.1.6.85.3 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。

4.1.6.85.4 另请参阅

- SHOW ERRORS

4.1.6.86 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。

4.1.6.86.1 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               )
```

4.1.6.86.2 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  )
```

4.1.6.86.3 MySQL 兼容性

该语句是 TiDB 对 MySQL 语法的扩展。

4.1.6.86.4 另请参阅

和 SPLIT REGION 语句相关的 session 变量有 tidb_scatter_region，tidb_wait_split_region_finish 和 tidb_wait_split_region_timeout，具体可参考TiDB 专用系统变量和语法。

4.1.6.87 START TRANSACTION

START TRANSACTION 语句用于在 TiDB 内部启动新事务。它类似于语句 BEGIN 和 SET autocommit = 0。

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。

4.1.6.87.1 语法图

BeginTransactionStmt:

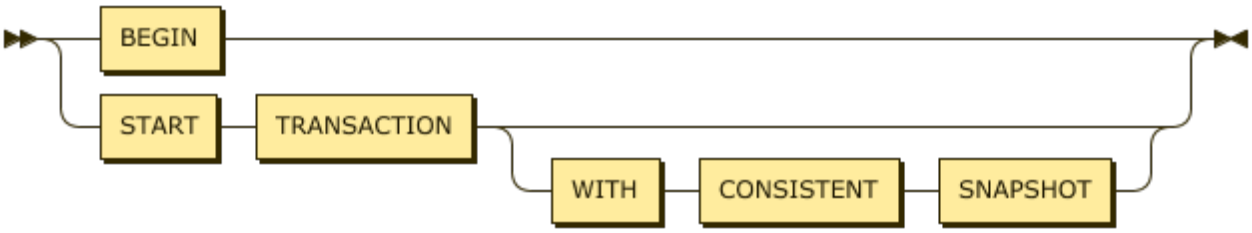


图 275: BeginTransactionStmt

4.1.6.87.2 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)
```

4.1.6.87.3 MySQL 兼容性

START TRANSACTION 语句与 MySQL 不完全兼容。

- START TRANSACTION 相当于 MySQL 的 START TRANSACTION WITH CONSISTENT SNAPSHOT，即 START TRANSACTION 后执行了一个从 InnoDB 任意表读数据的 SELECT 语句（非 SELECT FOR UPDATE）。
- READ ONLY 及其扩展选项都都只是语法兼容，其效果等同于 START TRANSACTION。

如发现任何其他兼容性差异，请在 GitHub 上提交 issue。

4.1.6.87.4 另请参阅

- COMMIT
- ROLLBACK
- BEGIN

4.1.6.88 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。

4.1.6.88.1 语法图

TraceStmt:

图 276: TraceStmt

TraceableStmt:

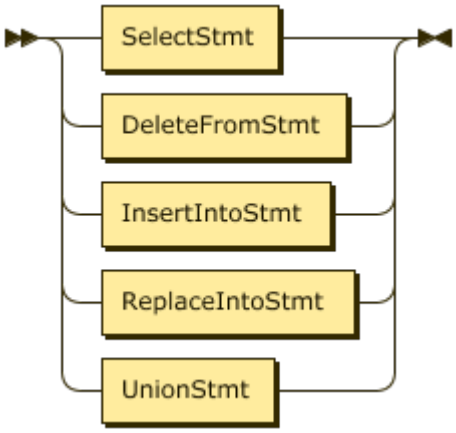


图 277: TraceableStmt

4.1.6.88.2 示例

```
trace format='row' select * from mysql.user;
```

```
+---------------------------+-----------------+------------+
| operation                 | startTS         | duration   |
+---------------------------+-----------------+------------+
| session.getTxnFuture      | 10:33:34.647148 | 3.847µs    |
|   ├─session.Execute       | 10:33:34.647146 | 536.233µs  |
|   ├─session.ParseSQL      | 10:33:34.647182 | 19.868µs   |
|   ├─executor.Compile      | 10:33:34.647219 | 295.688µs  |
|   ├─session.runStmt       | 10:33:34.647533 | 116.229µs  |
|   ├─session.CommitTxn     | 10:33:34.647631 | 5.44µs     |
|   ├─recordSet.Next        | 10:33:34.647707 | 833.103µs  |
|   ├─tableReader.Next      | 10:33:34.647709 | 806.783µs  |
|   ├─recordSet.Next        | 10:33:34.648572 | 19.367µs   |
|   └─tableReader.Next      | 10:33:34.648575 | 1.783µs    |
+---------------------------+-----------------+------------+
```

```
10 rows in set (0.00 sec)
```

```
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),(4),(5);
```

```
Query OK, 5 rows affected (0.02 sec)
Records: 5  Duplicates: 0  Warnings: 0
```

```
TRACE FORMAT='json' SELECT * FROM t1 WHERE id = 2;
```

```
operation: [
    {"ID":{"Trace":"60d20d005593de87","Span":"44e5b309242ffe2f","Parent":"79d146dac9a29a7e"},
        "Annotations":[
            {"Key":"Name","Value":"c2Vzc2lvbi5nZXRUeG5GdXR1cmU="},
            {"Key":"_schema:name","Value":null},
            {"Key":"Span.Start","Value":"MjAxOS0wNC0xN1QxMDozOToxMC45NDE2MTQ3ODYtMDY6MDA="},
            {"Key":"Span.End","Value":"MjAxOS0wNC0xN1QxMDozOToxMC45NDE2MjA0MDYtMDY6MDA="},
            {"Key":"_schema:Timespan","Value":null}
        ],
        "Sub":[
            {"ID":{"Trace":"60d20d005593de87","Span":"4dbf8f2ca373b4b0","Parent":"79
                ↪ d146dac9a29a7e"},
            "Annotations":[
                {"Key":"Name","Value":"c2Vzc2lvbi5QYXJzZVNRTA=="},
                {"Key":"_schema:name","Value":null},
                {"Key":"Span.Start","Value":"MjAxOS0wNC0xN1QxMDozOToxMC45NDE2NjE1MTQtMDY6MDA="},
                {"Key":"Span.End","Value":"MjAxOS0wNC0xN1QxMDozOToxMC45NDE3MDYxNjgtMDY6MDA="},
                {"Key":"_schema:Timespan","Value":null}
            ],
            "Sub":null},
            {"ID":{"Trace":"60d20d005593de87","Span":"6b6d6916df809604","Parent":"79
                ↪ d146dac9a29a7e"},
            "Annotations":[
                {"Key":"Name","Value":"ZXhlY3V0b3IuQ29tcGlsZQ=="},
                {"Key":"_schema:name","Value":null},
                {"Key":"Span.End","Value":"MjAxOS0wNC0xN1QxMDozOToxMC45NDE3NTcyODUtMDY6MDA="},
                {"Key":"Span.Start","Value":"MjAxOS0wNC0xN1QxMDozOToxMC45NDE3MzE0MjYtMDY6MDA="},
                {"Key":"_schema:Timespan","Value":null}
            ],
            "Sub":null},
            {"ID":{"Trace":"60d20d005593de87","Span":"3f1bcdd402a72911","Parent":"79
                ↪ d146dac9a29a7e"},
```

```
"Annotations":[
    {"Key":"Name","Value":"c2Vzc2lvbi5Db21taXRUeG4="},
    {"Key":"_schema:name","Value":null},
    {"Key":"Span.Start","Value":"MjAxOS0wNC0xN1QxMDozOToxMC45NDE3OTgyNjItMDY6MDA="},
    {"Key":"Span.End","Value":"MjAxOS0wNC0xN1QxMDozOToxMC45NDE4MDU1NzYtMDY6MDA="},
    {"Key":"_schema:Timespan","Value":null}
],
"Sub":null},
{"ID":{"Trace":"60d20d005593de87","Span":"58c1f7d66dc5afbc","Parent":"79
    ↪ d146dac9a29a7e"},
"Annotations":[
    {"Key":"Name","Value":"c2Vzc2lvbi5ydW5TdG10"},
    {"Key":"_schema:name","Value":null},
    {"Key":"Msg","Value":"eyJzcWwiOiJTRUxFQ1QgKiBGUk9NIHQxIFdIRVJFIGlkID0gMiJ9"},
    {"Key":"Time","Value":"MjAxOS0wNC0xN1QxMDozOToxMC45NDE3ODA1NjgtMDY6MDA="},
    {"Key":"_schema:log","Value":null},
    {"Key":"Span.End","Value":"MjAxOS0wNC0xN1QxMDozOToxMC45NDE4MTk5MzMtMDY6MDA="},
    {"Key":"Span.Start","Value":"MjAxOS0wNC0xN1QxMDozOToxMC45NDE3NzcyNDItMDY6MDA="},
    {"Key":"_schema:Timespan","Value":null}
],
"Sub":null},
{"ID":{"Trace":"60d20d005593de87","Span":"6bd8cc440fb31ed7","Parent":"79
    ↪ d146dac9a29a7e"},
"Annotations":[
    {"Key":"Name","Value":"c2Vzc2lvbi5FeGVjdXRl"},
    {"Key":"_schema:name","Value":null},
    {"Key":"Span.Start","Value":"MjAxOS0wNC0xN1QxMDozOToxMC45NDE2MTEwODktMDY6MDA="},
    {"Key":"Span.End","Value":"MjAxOS0wNC0xN1QxMDozOToxMC45NDE4NTU0My0wNjowMA=="},
    {"Key":"_schema:Timespan","Value":null}
],
"Sub":null},
{"ID":{"Trace":"60d20d005593de87","Span":"61d0b809f6cc018b","Parent":"79
    ↪ d146dac9a29a7e"},
"Annotations":[
    {"Key":"Name","Value":"cmVjb3JkU2V0Lk5leHQ="},
    {"Key":"_schema:name","Value":null},
    {"Key":"Span.Start","Value":"MjAxOS0wNC0xN1QxMDozOToxMC45NDE4NzQ1NTYtMDY6MDA="},
    {"Key":"Span.End","Value":"MjAxOS0wNC0xN1QxMDozOToxMC45NDIyOTg4NjYtMDY6MDA="},
    {"Key":"_schema:Timespan","Value":null}
],
"Sub":null},
{"ID":{"Trace":"60d20d005593de87","Span":"2bd2c3d47ccb1133","Parent":"79
    ↪ d146dac9a29a7e"},
"Annotations":[
    {"Key":"Name","Value":"cmVjb3JkU2V0Lk5leHQ="},
```

```
                {"Key":"_schema:name","Value":null},
                {"Key":"Span.Start","Value":"MjAxOS0wNC0xN1QxMDozOToxMC45NDIzMjY0ODgtMDY6MDA="},
                {"Key":"Span.End","Value":"MjAxOS0wNC0xN1QxMDozOToxMC45NDIzMjkwMDMtMDY6MDA="},
                {"Key":"_schema:Timespan","Value":null}
            ],
            "Sub":null}
        ]
    }
]
1 row in set (0.00 sec)
```

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：

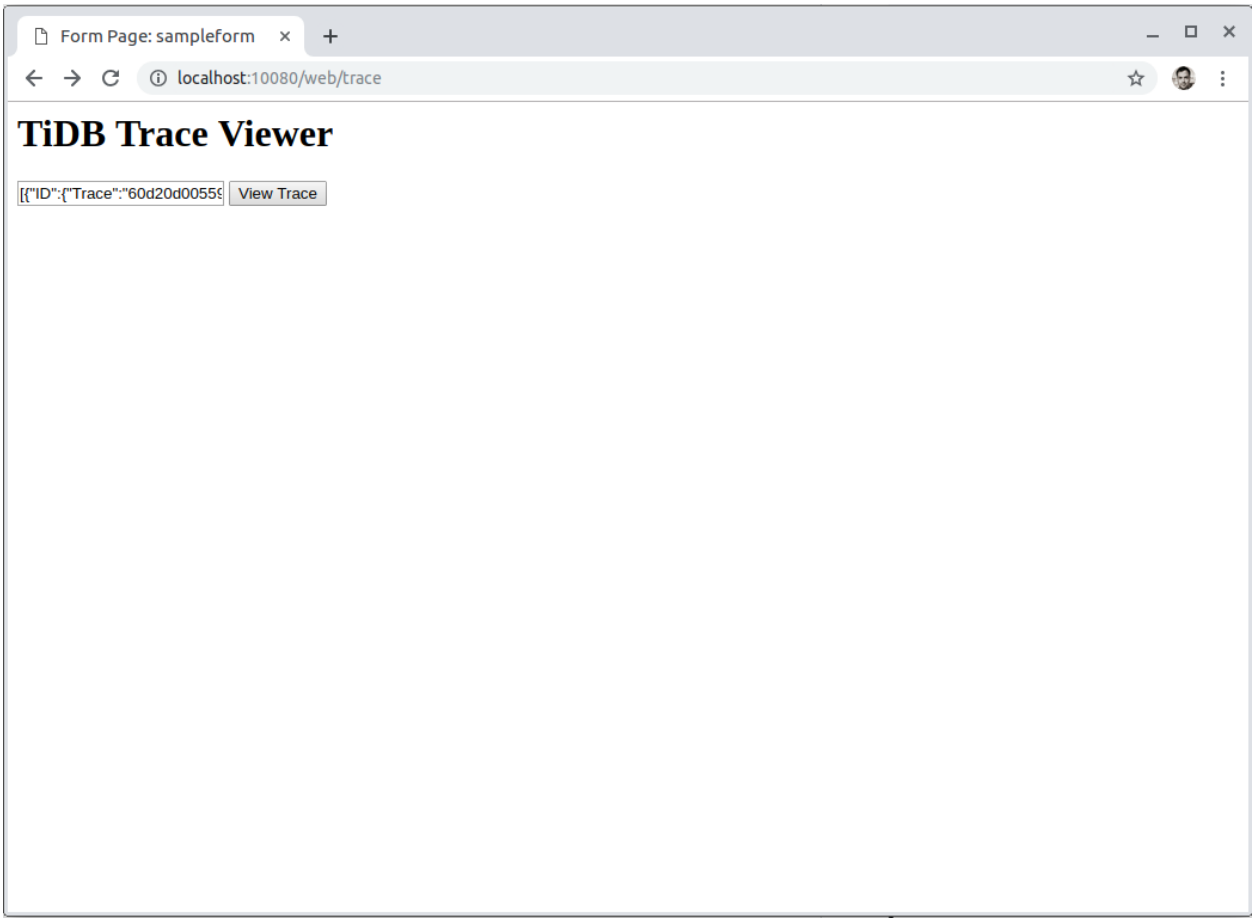


图 278: TiDB Trace Viewer-1

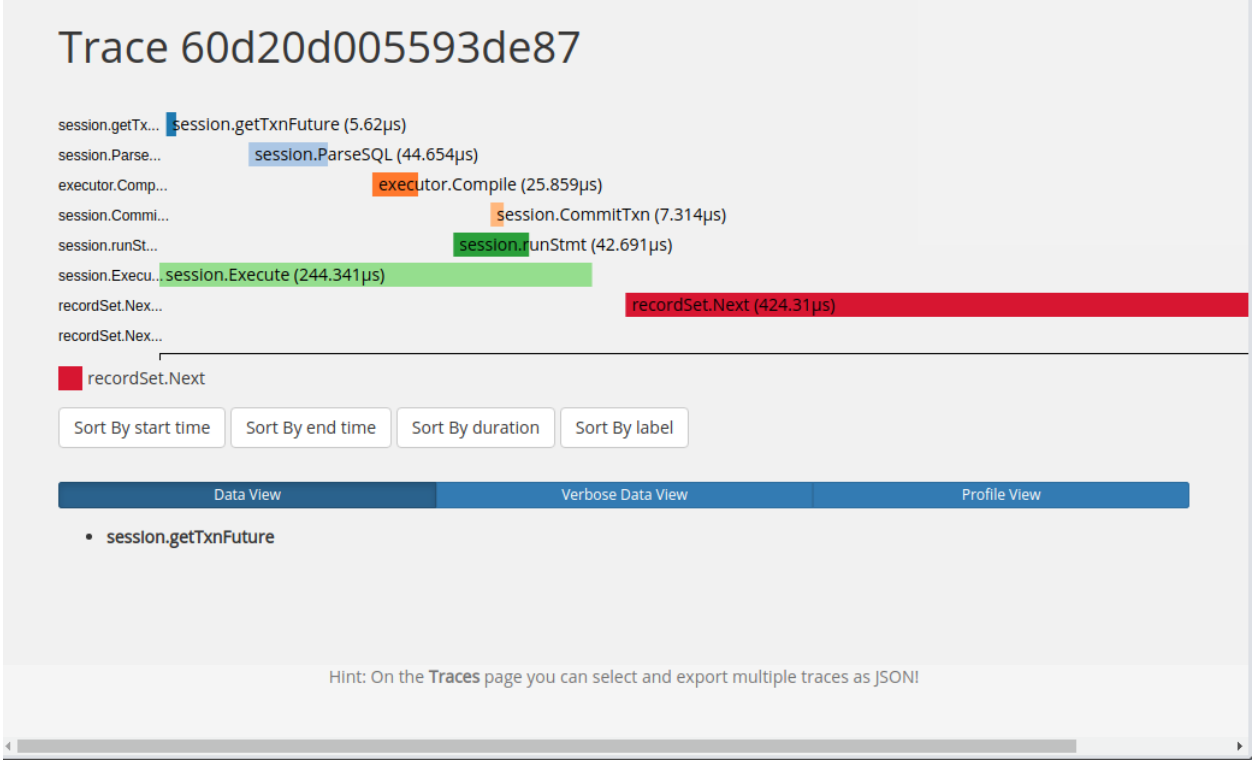


图 279: TiDB Trace Viewer-2

4.1.6.88.3 MySQL 兼容性

TRACE 语句是 TiDB 对 MySQL 语法的扩展。

4.1.6.88.4 另请参阅

- EXPLAIN ANALYZE

4.1.6.89 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。

4.1.6.89.1 语法图

TruncateTableStmt:

图 280: TruncateTableStmt

OptTable:

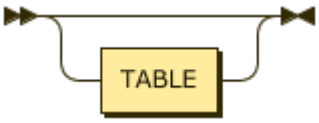


图 281: OptTable

TableName:

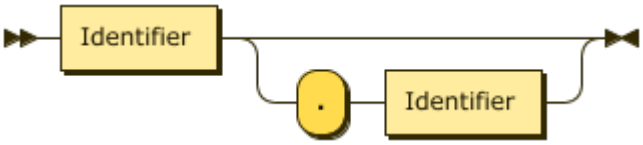


图 282: TableName

4.1.6.89.2 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)
```

4.1.6.89.3 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。

4.1.6.89.4 另请参阅

- DROP TABLE
- DELETE
- CREATE TABLE
- SHOW CREATE TABLE

4.1.6.90 UPDATE

UPDATE 语句用于修改指定表中的数据。

4.1.6.90.1 语法图

UpdateStmt:

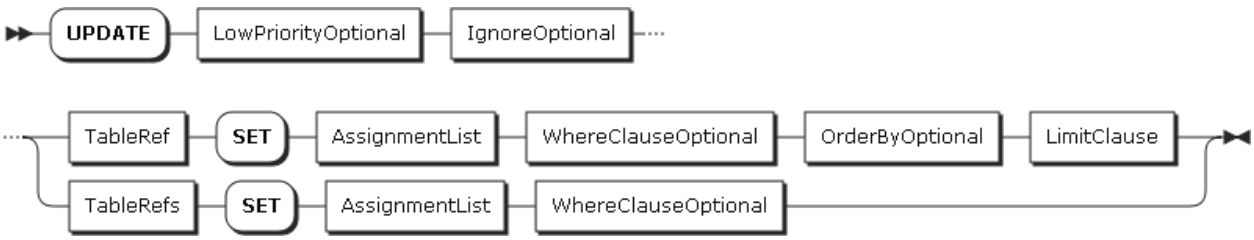


图 283: UpdateStmt

TableRef:

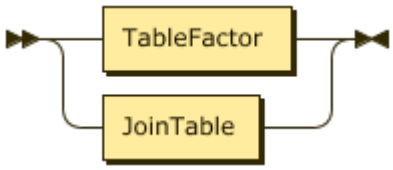


图 284: TableRef

TableRefs:

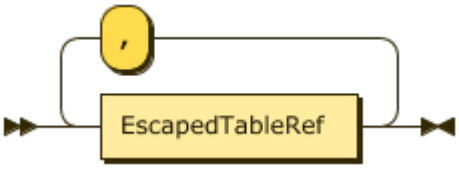


图 285: TableRefs

AssignmentList:

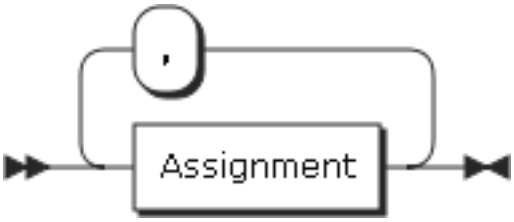


图 286: AssignmentList

WhereClauseOptional:

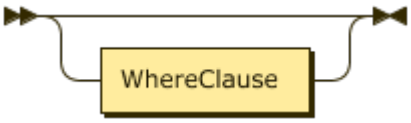


图 287: WhereClauseOptional

4.1.6.90.2 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)
```

4.1.6.90.3 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。

4.1.6.90.4 另请参阅

- INSERT
- SELECT
- DELETE
- REPLACE

4.1.6.91 USE

USE 语句可为用户会话选择当前数据库。

4.1.6.91.1 语法图

UseStmt:

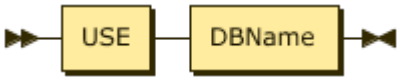


图 288: UseStmt

DBName:

图 289: DBName

4.1.6.91.2 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        |
| gc_delete_range      |
| gc_delete_range_done |
| help_topic           |
| role_edges           |
| stats_buckets        |
| stats_feedback       |
| stats_histograms     |
| stats_meta           |
| tables_priv          |
| tidb                 |
| user                 |
+----------------------+
```

```
16 rows in set (0.00 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)
```

4.1.6.91.3 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。

4.1.6.91.4 另请参阅

- CREATE DATABASE
- SHOW TABLES

4.1.7 约束

TiDB 支持的约束与 MySQL 的基本相同。

4.1.7.1 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。

4.1.7.2 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;
```

4.1.7.3 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 的值设置为 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。

4.1.7.4 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，如果开启该功能前所创建表带有整数类型的主键，即使开启添加/删除主键功能，也不能删除其主键约束。

4.1.7.5 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);
```

4.1.7.5.1 注意

- TiDB 支持外键是为了在将其他数据库的数据迁移到 TiDB 时，避免该语法导致的报错。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;
```

4.1.8 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。

为了在功能上兼容 MySQL 5.7，TiDB 支持生成列 (generated column)。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。

4.1.8.1 使用 generated column 对 JSON 建索引

MySQL 5.7 及 TiDB 都不能直接为 JSON 类型的列添加索引，即不支持如下表结构：

```
CREATE TABLE person (
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    address_info JSON,
    KEY (address_info)
);
```

为 JSON 列添加索引之前，首先必须抽取该列为 generated column。

以 `city` generated stored column 为例，你可以添加索引：

```
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'))) STORED,
    KEY (city)
);
```

该表中，`city` 列是一个 generated stored column。顾名思义，此列由该表的其他列生成，对此列进行插入或更新操作时，并不能对之赋值。此列按其定义的表达式生成，并存储在数据库中，这样在读取此列时，就可以

直接读取，不用再读取其依赖的 address_info 列后再计算得到。city 列的索引存储在数据库中，并使用和 varchar(64) 类的其他索引相同的结构。

可使用 generated stored column 的索引，以提高如下语句的执行速度：

```
SELECT name, id FROM person WHERE city = 'Beijing';
```

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 virtual column：

```
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'))) STORED NOT NULL,
    KEY (city)
);
```

INSERT 和 UPDATE 语句都会检查 virtual column 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
```

4.1.8.2 使用 generated virtual column

TiDB 也支持 generated virtual column，和 generated store column 不同的是，此列按需生成，并不存储在数据库中，也不占用内存空间，因而是虚拟的。

```
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'))) VIRTUAL
);
```

4.1.8.3 局限性

目前 JSON and generated column 有以下局限性：

- 不能通过 ALTER TABLE 增加 STORED 存储方式的 generated column；
- 不能通过 ALTER TABLE 将 generated stored column 转换为普通列，也不能将普通列转换成 generated stored column；
- 不能通过 ALTER TABLE 修改 generated stored column 的生成列表达式；
- 并未支持所有的JSON 函数。

4.1.9 分区表

本文介绍 TiDB 的分区表。

4.1.9.1 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。

4.1.9.1.1 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 操作进行定义。

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 NOT NULL DEFAULT '9999-12-31',
    job_code INT NOT NULL,
    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。你可以这样做：

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE NOT NULL DEFAULT '9999-12-31',
    job_code INT NOT NULL,
    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, 'Mitchell', 'Wilson', '1998-06-25', NULL, 13) 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 NOT NULL DEFAULT '9999-12-31',
    job_code INT NOT NULL,
    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 NOT NULL DEFAULT '9999-12-31',
    job_code INT NOT NULL,
    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 NOT NULL 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 值，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。

4.1.9.1.2 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 NOT NULL 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 NOT NULL 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，因为这种情况可以像 range 分区一样裁剪。

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。即使如此，`YEAR(date_col)` 也仍然是一个比较好的 hash 函数，因为它的结果是随着 `date_col` 的值的比例变化的。

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
```

4.1.9.1.3 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。

4.1.9.2 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。

4.1.9.2.1 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
```

4.1.9.2.2 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。

4.1.9.3 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)

4.1.9.3.1 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';
```

4.1.9.4 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。

4.1.9.5 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。

4.1.9.5.1 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。

4.1.9.5.2 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() (see CEILING() and FLOOR())
DATEDIFF()
DAY()
DAYOFMONTH()
DAYOFWEEK()
DAYOFYEAR()
EXTRACT() (see EXTRACT() function with WEEK specifier)
FLOOR() (see CEILING() and FLOOR())
HOUR()
MICROSECOND()
MINUTE()
MOD()
MONTH()
```

```
QUARTER()
SECOND()
TIME_TO_SEC()
TO_DAYS()
TO_SECONDS()
UNIX_TIMESTAMP() (with TIMESTAMP columns)
WEEKDAY()
YEAR()
YEARWEEK()
```

4.1.9.5.3 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。

INFORMATION_SCHEMA.PARTITION 表暂不支持。

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)
```

4.1.10 字符集支持

名词解释，下面的阐述中会交错使用中文或者英文，请互相对照：

- Character Set：字符集
- Collation：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)
```

> 注意：
>
> - 在 TiDB 中 utf8 被当做成了 utf8mb4 来处理。
> - 每种字符集都对应一个默认的 Collation，当前有且仅有一个。

对于字符集来说，至少会有一个 Collation（排序规则）与之对应。而大部分字符集实际上会有多个 Collation。利用以下的语句可以查看：

```
SHOW COLLATION WHERE Charset = 'latin1';
```

```
+-------------------|---------|------|---------|----------|---------+
| Collation         | Charset | Id   | Default | Compiled | Sortlen |
+-------------------|---------|------|---------|----------|---------+
| latin1_german1_ci | latin1  |    5 |         | Yes      |       1 |
| latin1_swedish_ci | latin1  |    8 | Yes     | Yes      |       1 |
| latin1_danish_ci  | latin1  |   15 |         | Yes      |       1 |
| latin1_german2_ci | latin1  |   31 |         | Yes      |       1 |
| latin1_bin        | latin1  |   47 |         | Yes      |       1 |
| latin1_general_ci | latin1  |   48 |         | Yes      |       1 |
| latin1_general_cs | latin1  |   49 |         | Yes      |       1 |
| latin1_spanish_ci | latin1  |   94 |         | Yes      |       1 |
+-------------------|---------|------|---------|----------|---------+
8 rows in set (0.00 sec)
```

latin1 Collation（排序规则）分别有以下含义：

Collation	含义
latin1_bin	latin1 编码的二进制表示
latin1_danish_ci	丹麦语/挪威语，不区分大小写
latin1_general_ci	多种语言的 (西欧)，不区分大小写
latin1_general_cs	多种语言的 (ISO 西欧)，区分大小写
latin1_german1_ci	德国 DIN-1 (字典序)，不区分大小写
latin1_german2_ci	德国 DIN-2，不区分大小写
latin1_spanish_ci	现代西班牙语，不区分大小写
latin1_swedish_ci	瑞典语/芬兰语，不区分大小写

每一个字符集，都有一个默认的 Collation，例如 utf8 的默认 Collation 就为 utf8_bin。

> **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 目前的 Collation 只支持区分大小写的比较排序规则。

4.1.10.1 Collation 命名规则

TiDB 的 Collation 遵循着如下的命名规则：

- Collation 的前缀是它相应的字符集，通常之后会跟着一个或者更多的后缀来表名其他的排序规则，例如：utf8_general_ci 和 lation1_swedish_ci 是 utf8 和 latin1 字符集的 Collation。但是 binary 字符集只有一个 Collation，就是 binary。
- 一个语言对应的 Collation 会包含语言的名字，例如 utf8_turkish_ci 和 utf8_hungarian_ci 是依据 Turkish(土耳其语) 和 Hungarian(匈牙利语) 的排序规则来排序。
- Collation 的后缀表示了 Collation 是否区分大小写和是否区分口音。下面的表展示了这些特性：

后缀	含义
_ai	口音不敏感（Accent insensitive）
_as	口音敏感（Accent sensitive）
_ci	大小写不敏感
_cs	大小写敏感

> **注意：**
>
> 目前为止 TiDB 只支持部分以上提到的 Collation。

4.1.10.2 集群 Character Set 和 Collation

暂不支持

4.1.10.3 数据库 Character Set 和 Collation

每个数据库都有相应的 Character Set 和 Collation，数据库的 Character Set 和 Collation 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 utf8 COLLATE utf8_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 |
+--------------------------|----------------------+
| utf8                     | utf8_general_ci      |
+--------------------------|----------------------+
1 row in set (0.00 sec)
```

```
CREATE SCHEMA test2 CHARACTER SET latin1 COLLATE latin1_general_ci;
```

```
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_general_ci    |
+--------------------------|----------------------+
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';
```

4.1.10.4 表的 Character Set 和 Collation

表的 Character Set 和 Collation 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 utf8 COLLATE utf8_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。

4.1.10.5 列的 Character Set 和 Collation

列的 Character Set 和 Collation 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。

4.1.10.6 字符串 Character Sets 和 Collation

每一字符串字符文字有一个字符集和一个比较排序规则，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 _latin1'string';
SELECT _latin1'string' COLLATE latin1_danish_ci;
```

规则，如下：

- 规则 1：指定 CHARACTER SET charset_name 和 COLLATE collation_name，则直接使用 CHARACTER SET charset_name 和 COLLATE collation_name。
- 规则 2: 指定 CHARACTER SET charset_name 且未指定 COLLATE collation_name，则使用 CHARACTER SET charset_name 和 CHARACTER SET charset_name 默认的排序比较规则。
- 规则 3: 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比较排序规则。

4.1.10.7 客户端连接的 Character Sets 和 Collations

- 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 utf8` 表示客户端会在接下来的请求中，都使用 utf8 字符集。服务端也会在之后返回结果的时候使用 utf8 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 默认的 Collation。

- `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 collation_connection = @@collation_database;
```

4.1.10.8 集群，服务器，数据库，表，列，字符串 Character Sets 和 Collation 优化级

字符串 => 列 => 表 => 数据库 => 服务器 => 集群

4.1.10.9 Character Sets 和 Collation 通用选择规则

- 规则 1：指定 CHARACTER SET charset_name 和 COLLATE collation_name，则直接使用 CHARACTER SET charset_name 和 COLLATE collation_name。
- 规则 2: 指定 CHARACTER SET charset_name 且未指定 COLLATE collation_name，则使用 CHARACTER SET charset_name 和 CHARACTER SET charset_name 默认的排序比较规则。
- 规则 3: CHARACTER SET charset_name 和 COLLATE collation_name 都未指定，则使用更高优化级给出的字符集和排序比较规则。

4.1.10.10 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。

更多细节，参考 Connection Character Sets and Collations。

4.1.11 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`。

4.1.11.1 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 的行为象“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。

4.1.11.2 SQL mode 列表

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_IN_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)

4.1.12 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。

4.1.12.1 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。

4.1.12.2 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。

4.1.12.2.1 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)
```

4.1.12.2.2 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)
```

4.1.12.2.3 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
}
```

4.1.12.3 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)
```

4.1.12.4 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]。

4.1.12.5 扩展阅读

- 创建视图
- 删除视图

4.1.13 tidb-server

4.1.13.1 系统变量

MySQL 系统变量 (System Variables) 是一些系统参数，用于调整数据库运行时的行为，根据变量的作用范围分为全局范围有效（Global Scope）以及会话级别有效（Session Scope）。TiDB 支持 MySQL5.7 的所有系统变量，大部分变量仅仅是为了兼容性而支持，不会影响运行时行为。

4.1.13.1.1 设置系统变量

通过SET 语句 可以修改系统变量的值。进行修改时，还要考虑变量可修改的范围，不是所有的变量都能在全局/会话范围内进行修改。具体的可修改范围参考 MySQL 动态变量文档。

全局范围值

- 在变量名前加 GLOBAL 关键词或者是使用 @@global. 作为修饰符:

```
SET GLOBAL autocommit = 1;
```

```
SET @@global.autocommit = 1;
```

> **注意：**
>
> 在分布式 TiDB 中，GLOBAL 变量的设置会持久化到存储层中，单个 TiDB 实例每 2 秒会主动进行一次全变量的获取并形成 gvc (global variables cache) 缓存，该缓存有效时间最多可持续 2 秒。在设置 GLOBAL 变量之后，为了保证新会话的有效性，请确保两个操作之间的间隔大于 2 秒。相关细节可以查看 Issue #14531。

会话范围值

- 在变量名前加 SESSION 关键词或者是使用 @@session. 作为修饰符，或者是不加任何修饰符:

```
SET SESSION autocommit = 1;
```

```
SET @@session.autocommit = 1;
```

```
SET @@autocommit = 1;
```

- LOCAL 以及 @@local. 是 SESSION 以及 @@session. 的同义词

系统变量的作用机制

- 会话范围的系统变量仅仅会在创建会话时才会根据全局范围系统变量初始化自己的值。更改全局范围的系统变量不会改变已经创建的会话正在使用的系统变量的值。

```
SELECT @@GLOBAL.autocommit;
```

```
+--------------------+
| @@GLOBAL.autocommit |
+--------------------+
| ON                 |
+--------------------+
1 row in set (0.00 sec)
```

```
SELECT @@SESSION.autocommit;
```

```
+----------------------+
| @@SESSION.autocommit |
+----------------------+
| ON                   |
+----------------------+
1 row in set (0.00 sec)
```

```
SET GLOBAL autocommit = OFF;
```

```
Query OK, 0 rows affected (0.01 sec)
```

会话范围的系统变量不会改变，会话中执行的事务依旧是以自动提交的形式来进行：

```
SELECT @@SESSION.autocommit;
```

```
+----------------------+
| @@SESSION.autocommit |
+----------------------+
| ON                   |
+----------------------+
1 row in set (0.00 sec)
```

```
SELECT @@GLOBAL.autocommit;
```

```
+---------------------+
| @@GLOBAL.autocommit |
+---------------------+
| OFF                 |
+---------------------+
1 row in set (0.00 sec)
```

```
exit
```

```
Bye
```

```
mysql -h 127.0.0.1 -P 4000 -u root -D test
```

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3
Server version: 5.7.25-TiDB-None MySQL Community Server (Apache License 2.0)

Copyright (c) 2000, 2019, 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.

mysql>
```

新建的会话会使用新的全局变量：

```
SELECT @@SESSION.autocommit;
```

```
+----------------------+
| @@SESSION.autocommit |
+----------------------+
| OFF                  |
+----------------------+
1 row in set (0.00 sec)
```

4.1.13.1.2 TiDB 支持的 MySQL 系统变量

下列系统变量是 TiDB 真正支持并且行为和 MySQL 一致：

变量名	作用域	说明
autocommit	GLOBAL | SESSION	是否自动 Commit 事务
sql_mode	GLOBAL | SESSION	支持部分 MySQL SQL mode，
time_zone	GLOBAL | SESSION	数据库所使用的时区
tx_isolation	GLOBAL | SESSION	事务隔离级别
max_execution_time	GLOBAL | SESSION	语句超时时间，单位为毫秒
innodb_lock_wait_timeout	GLOBAL | SESSION	悲观事务语句等锁时间，单位为秒
interactive_timeout	SESSION | GLOBAL	交互式用户会话的空闲超时，单位为秒

> 注意：
>
> `max_execution_time` 目前对所有类型的 `statement` 生效，并非只对 `SELECT` 语句生效。实际精度在 100ms 级别，而非更准确的毫秒级别。

4.1.13.1.3 TiDB 特有的系统变量

参见TiDB 专用系统变量。

4.1.13.2 TiDB 专用系统变量和语法

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。

4.1.13.2.1 系统变量

变量可以通过 SET 语句设置，例如

```
set @@tidb_distsql_scan_concurrency = 10;
```

如果需要设置全局变量，执行

```
set @@global.tidb_distsql_scan_concurrency = 10;
```

tidb_snapshot

作用域：SESSION

默认值：空字符串

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。

tidb_import_data

作用域：SESSION

默认值：0

这个变量用来表示当前状态是否为从 dump 文件中导入数据。当这个变量被设置为 1 时，唯一索引约束不被检查以加速导入速度。这个变量不对外用，只是给 lightning 使用，请用户不要自行修改。

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 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。

tidb_auto_analyze_ratio

作用域：GLOBAL

默认值：0.5

这个变量用来设置自动 ANALYZE 更新的阈值。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` 语句以自动更新该表的统计信息。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。

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 更新的开始时间。

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 更新的结束时间。

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。

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。

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。

tidb_config

作用域：SESSION

默认值：空字符串

这个变量是一个只读变量，用来获取当前 TiDB Server 的配置信息。

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。

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。

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。

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。

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。

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。

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。

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。

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。

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。

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。

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。

tidb_max_chunk_size

作用域：SESSION | GLOBAL

默认值：1024

这个变量用来设置执行过程中一个 chunk 最大的行数。

tidb_mem_quota_query

作用域：SESSION

默认值：32 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。

tidb_mem_quota_hashjoin

作用域：SESSION

默认值：32 GB

这个变量用来设置 HashJoin 算子的内存使用阈值。如果 HashJoin 算子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。

tidb_mem_quota_mergejoin

作用域：SESSION

默认值：32 GB

这个变量用来设置 MergeJoin 算子的内存使用阈值。如果 MergeJoin 算子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。

tidb_mem_quota_sort

作用域：SESSION

默认值：32 GB

这个变量用来设置 Sort 算子的内存使用阈值。如果 Sort 算子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。

tidb_mem_quota_topn

作用域：SESSION

默认值：32 GB

这个变量用来设置 TopN 算子的内存使用阈值。如果 TopN 算子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。

tidb_mem_quota_indexlookupreader

作用域：SESSION

默认值：32 GB

这个变量用来设置 IndexLookupReader 算子的内存使用阈值。

如果 IndexLookupReader 算子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。

tidb_mem_quota_indexlookupjoin

作用域：SESSION

默认值：32 GB

这个变量用来设置 IndexLookupJoin 算子的内存使用阈值。如果 IndexLookupJoin 算子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。

tidb_mem_quota_nestedloopapply

作用域：SESSION

默认值：32 GB

这个变量用来设置 NestedLoopApply 算子的内存使用阈值。如果 NestedLoopApply 算子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。

tidb_general_log

作用域：SERVER

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。

tidb_enable_streaming

作用域：SESSION

默认值：0

这个变量用来设置是否启用 Streaming。

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。

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。

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。

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）故障等时，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。

tidb_enable_table_partition

作用域：SESSION

默认值：“auto”

这个变量用来设置是否开启 TABLE PARTITION 特性。默认值 `auto` 表示开启 range partition 和 hash partion。`off` 表示关闭 TABLE PARTITION 的特性，此时语法还是会依旧兼容，只是建立的 partition table 实际上并不是真正的 partition table，而是和普通的 table 一样。on 表示开启，目前的作用和 `auto` 一样。

注意，目前 TiDB 只支持 range partition 和 hash partition。

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。

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。

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 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。

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。

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。

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。

tidb_force_priority

作用域：SESSION

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。

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。

SHARD_ROW_ID_BITS

对于 PK 非整数或没有 PK 的表，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;

tidb_slow_log_threshold

作用域：SESSION

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 300ms。

示例：

```
set tidb_slow_log_threshold = 200;
```

tidb_query_log_max_len

作用域：SESSION

默认值：2048 (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;
```

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。

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。当这个变量值设置为 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 的值设置为 0：

```
create table t (i int key);
insert into t values (1);
begin optimistic;
insert into t values (1);
```

```
Query OK, 1 row affected
```

```
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 的值设置为 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。

tidb_check_mb4_value_in_utf8

作用域：SERVER

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。

注意，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 以及之前）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。

tidb_opt_insubq_to_join_and_agg

作用域：SESSION | GLOBAL

默认值：1

这个用来设置是否开启优化规则：将子查询转成 join 和 aggregation。

示例：

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 join 如下：

```
select * 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 * from t, t1 where t.a=t1.a;
```

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。

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。

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。

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。

tidb_enable_fast_analyze

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。

tidb_expensive_query_time_threshold

作用域：SERVER

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。

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。

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。

tidb_scatter_region

作用域：GLOBAL

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。

tidb_allow_remove_auto_inc 从 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。默认为不允许。

tidb_enable_stmt_summary 从 v3.0.4 版本开始引入

作用域：SESSION | GLOBAL

默认值：0

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 performance_schema.events_statements_summary_by_digest 中，用于定位和排查 SQL 性能问题。

4.1.13.3 TiDB 命令行参数

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）。

4.1.13.3.1 --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

4.1.13.3.2 --binlog-socket

- TiDB 服务使用 unix socket file 方式接受内部连接，如 Pump 服务
- 默认：“”
- 例如，可以使用 “/tmp/pump.sock” 来接受 Pump unix socket file 通信

4.1.13.3.3 --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。

4.1.13.3.4 --cors

- 用于设置 TiDB HTTP 状态服务的 Access-Control-Allow-Origin
- 默认：“”

4.1.13.3.5 --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

4.1.13.3.6 -L

- Log 级别
- 默认：“info”
- 可选项为：debug、 info、 warn、 error、 fatal

4.1.13.3.7 --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

4.1.13.3.8 --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

4.1.13.3.9 --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

4.1.13.3.10 --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

4.1.13.3.11 -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

4.1.13.3.12 --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

4.1.13.3.13 --proxy-protocol-networks

- PROXY Protocol 允许的代理服务器地址列表。如需配置多个地址，用 , 分隔。
- 默认：“”
- 如果该参数为空，TiDB 会禁用 PROXY Protocol 功能。地址可以使用 IP 地址 (192.168.1.50) 或者 CIDR (192.168.1.0/24)，* 代表所有地址。

4.1.13.3.14 --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。

4.1.13.3.15 --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

4.1.13.3.16 --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

4.1.13.3.17 --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

4.1.13.3.18 --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

4.1.13.3.19 --status-host

- TiDB 服务状态监听 host
- 默认：“0.0.0.0”

4.1.13.3.20 --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）

4.1.13.3.21 --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

4.1.13.3.22 -V

- 输出 TiDB 的版本
- 默认：“”

4.1.13.4 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。

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。

oom-action

- 指定 TiDB 发生 out-of-memory 错误时的操作。
- 默认值：“log”
- 现在合法的选项是 [“log”，“cancel”]，如果为 “log”，仅仅是打印日志，不作实质处理。如果为 “cancel”，我们会取消执行这个操作，并且输出日志。

mem-quota-query

- 单条 SQL 语句可以占用的最大内存阈值。
- 默认值：34359738368
- 超过该值的请求会被 oom-action 定义的行为所处理。

enable-streaming

- 开启 coprocessor 的 streaming 获取数据模式。
- 默认值：false

`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）。

`lease`

- DDL 租约超时时间。
- 默认值：45s
- 单位：秒

`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`。

`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

`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

`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。

`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}`。

`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。

4.1.13.4.1 log

日志相关的配置项。

`format`

- 指定日志输出的格式，可选项为 [json, text, console]。
- 默认值：“text”

`disable-timestamp`

- 是否禁止在日志中输出时间戳。
- 默认值：false
- 如果设置为 true，那么日志里面将不会输出时间戳。

`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。

`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。

`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。

`query-log-max-len`

- 最长的 SQL 输出长度。
- 默认值：2048
- 当语句的长度大于 `query-log-max-len`，将会被截断输出。

4.1.13.4.2 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：300MB
- 最大设置上限为 4GB。

`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。

`log-rotate`

- 是否每日创建一个新的日志文件。
- 默认值：true
- 如果设置为 true，每天会新建新的日志文件，如果设置为 false，那么只会输出到一个日志文件。

4.1.13.4.3 security

安全相关配置。

`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。

`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。

`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。

`cluster-ssl-ca`

- CA 根证书，用于用 tls 连接 TiKV/PD
- 默认值：“”

`cluster-ssl-cert`

- ssl 证书文件路径，用于用 tls 连接 TiKV/PD
- 默认值：“”

`cluster-ssl-key`

- ssl 私钥文件路径，用于用 tls 连接 TiKV/PD

- 默认值： “”

`skip-grant-table`

- 是否跳过权限检查
- 默认值：false

4.1.13.4.4 performance

性能相关配置。

`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。

`max-memory`

- Prepare cache LRU 使用的最大内存限制，超过 performance.max-memory * (1 - prepared-plan-cache.memory-guard-ratio) 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。在 LRU 的 size 大于 prepared-plan-cache.capacity 的情况下，也会剔除 LRU 中的元素。

`stmt-count-limit`

- TiDB 一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。该限制只在乐观事务中生效，如果使用悲观事务，事务中的语句数将不受此限制。

`tcp-keep-alive`

- TiDB 在 TCP 层开启 keepalive。
- 默认值：true

`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

`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，analyze 以及 feedback 等操作都不会再进行。

`run-auto-analyze`

- TiDB 是否做自动的 Analyze。
- 默认值：true

`feedback-probability`

- TiDB 对查询收集统计信息反馈的概率。
- 默认值：0.05
- 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。

`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：1024

`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。

`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。

4.1.13.4.5 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。

`enabled`

- 开启 prepare 语句的 plan cache。
- 默认值：false

`capacity`

- 缓存语句的数量。
- 默认值：100

`memory-guard-ratio`

- 用于防止超过 performance.max-memory, 超过 max-proc * (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。

4.1.13.4.6 tikv-client

`grpc-connection-count`

- 跟每个 TiKV 之间建立的最大连接数。
- 默认值：16

`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

`grpc-keepalive-timeout`

- TiDB 与 TiKV 节点 rpc keepalive 检查的超时时间
- 默认值：3

- 单位：秒

`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。

`max-txn-time-use`

- 单个事务允许的最大执行时间。
- 默认值：590
- 单位：秒

`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

`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

`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。

`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

4.1.13.4.7 txn-local-latches

事务内存锁相关配置，当本地事务冲突比较多时建议开启。

`enable`

- 开启

- 默认值：false

`capacity`

- Hash 对应的 slot 数，会自动向上调整为 2 的指数倍。每个 slot 占 32 Bytes 内存。当写入数据的范围比较广时（如导数据），设置过小会导致变慢，性能下降。
- 默认值：2048000

4.1.13.4.8 binlog

TiDB Binlog 相关配置。

`enable`

- 开启 binlog 开关。
- 默认值：false

`write-timeout`

- 写 binlog 的超时时间，推荐不修改该值。
- 默认值：15s
- 单位：秒

`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。

`binlog-socket`

- binlog 输出网络地址。
- 默认值：“”

`strategy`

- binlog 输出时选择 pump 的策略，仅支持 hash ，range 方法。
- 默认值：“range”

4.1.13.4.9 status

TiDB 服务状态相关配置。

`report-status`

- 开启 HTTP API 服务的开关。
- 默认值：true

`record-db-qps`

- 输与 database 相关的 QPS metrics 到 Prometheus 的开关。
- 默认值：false

4.1.13.4.10 stmt-summary 从 v3.0.4 版本开始引入

系统表 `events_statement_summary_by_digest` 的相关配置。

max-stmt-count

- `events_statement_summary_by_digest` 表中保存的 SQL 种类的最大数量。
- 默认值：100

max-sql-length

- `events_statement_summary_by_digest` 表中`DIGEST_TEXT` 和 `QUERY_SAMPLE_TEXT` 列的最大显示长度。
- 默认值：4096

4.1.13.4.11 pessimistic-txn

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

max-retry-count

- 悲观事务中每个语句最大重试次数，超出该限制将会报错。
- 默认值：256

4.1.13.4.12 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。

`allow-auto-random` 从 v3.1.0 版本开始引入

- 用于控制是否允许使用 `AUTO_RANDOM`。
- 默认值：false
- 默认情况下，不支持使用 `AUTO_RANDOM`。当该值为 true 时，不允许同时设置 alter-primary-key 为 true。

4.1.14 pd-server

4.1.14.1 PD 配置参数

PD 可以通过命令行参数或环境变量配置。

4.1.14.1.1 --advertise-client-urls

- 对外客户端访问 URL 列表。
- 默认：${client-urls}
- 在某些情况下，譬如 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。

4.1.14.1.2 --advertise-peer-urls

- 对外其他 PD 节点访问 URL 列表。
- 默认：${peer-urls}
- 在某些情况下，譬如 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。

4.1.14.1.3 --client-urls

- 处理客户端请求监听 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。

4.1.14.1.4 --peer-urls

- 处理其他 PD 节点请求监听 URL 列表。
- default: 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。

4.1.14.1.5 --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。

4.1.14.1.6 --data-dir

- PD 存储数据路径。
- 默认：default.${name}

4.1.14.1.7 --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。

4.1.14.1.8 --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。

4.1.14.1.9 -L

- Log 级别。
- 默认：“info”
- 我们能选择 debug, info, warn, error 或者 fatal。

4.1.14.1.10 --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。

4.1.14.1.11 --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。

4.1.14.1.12 --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

4.1.14.1.13 --cacert

- CA 文件路径，用于开启 TLS。
- 默认：“”

4.1.14.1.14 --cert

- 包含 X509 证书的 PEM 文件路径，用户开启 TLS。
- 默认：“”

4.1.14.1.15 --key

- 包含 X509 key 的 PEM 文件路径，用于开启 TLS。
- 默认：“”

4.1.14.1.16 --metrics-addr

- 指定 Prometheus Pushgateway 的地址。
- 默认：“”
- 如果留空，则不开启 Prometheus Push。

4.1.14.1.17 --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。

4.1.14.1.18 -V, --version

- 输出版本信息并退出。

4.1.14.2 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。

`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

`tso-save-interval`

- TSO 分配的时间窗口, 实时持久存储。
- 默认：3s

`initial-cluster-state`

- 集群初始状态
- 默认：new

`enable-prevote`

- 开启 raft prevote 的开关。
- 默认：true

`quota-backend-bytes`

- 元信息数据库存储空间的大小，默认 8GiB。
- 默认：8589934592

`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

`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

`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

`tick-interval`

- etcd raft 的 tick 周期。
- 默认：100ms

`election-interval`

- etcd leader 选举的超时时间。
- 默认：3s

`use-region-storage`

- 开启独立的 region 存储。
- 默认：false

4.1.14.2.1 security

安全相关配置项。

`cacert-path`

- CA 文件路径
- 默认：“”

`cert-path`

- 包含 X509 证书的 PEM 文件路径
- 默认：“”

`key-path`

- 包含 X509 key 的 PEM 文件路径
- 默认：“”

4.1.14.2.2 log

日志相关的配置项。

`format`

- 日志格式，可指定为“text”，“json”，“console”。
- 默认：text

`disable-timestamp`

- 是否禁用日志中自动生成的时间戳。
- 默认：false

4.1.14.2.3 log.file

日志文件相关的配置项。

`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

`max-days`

- 日志保留的最长天数。
- 默认: 28
- 最小值为 1

`max-backups`

- 日志文件保留的最大个数。
- 默认: 7
- 最小值为 1

4.1.14.2.4 metric

监控相关的配置项。

`interval`

- 向 Prometheus 推送监控指标数据的间隔时间。
- 默认: 15s

4.1.14.2.5 schedule

调度相关的配置项。

`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

`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

`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

`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

`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

`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

`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

`leader-schedule-limit`

- 同时进行 leader 调度的任务个数。
- 默认：4

`region-schedule-limit`

- 同时进行 Region 调度的任务个数
- 默认：4

`replica-schedule-limit`

- 同时进行 replica 调度的任务个数。
- 默认：8

`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

`high-space-ratio`

- 设置 store 空间充裕的阈值。
- 默认：0.6
- 最小值：大于 0
- 最大值：小于 1

`low-space-ratio`

- 设置 store 空间不足的阈值。
- 默认：0.8
- 最小值：大于 0
- 最大值：小于 1

`tolerant-size-ratio`

- 控制 balance 缓冲区大小。
- 默认：5
- 最小值：0

`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

`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

`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

`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

`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

`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

4.1.14.2.6 replication

副本相关的配置项。

`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

`location-labels`

- TiKV 集群的拓扑信息。
- 默认：[]

4.1.14.2.7 label-property

标签相关的配置项。

key

- 拒绝 leader 的 store 带有的 label key。
- 默认：“”

`value`

- 拒绝 leader 的 store 带有的 label value。
- 默认：“”

4.1.15 tikv-server

4.1.15.1 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

4.1.15.1.1 `-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”

4.1.15.1.2 --advertise-addr

- TiKV 对外访问地址。
- 默认：${addr}
- 在某些情况下，譬如 docker，或者 NAT 网络环境，客户端并不能通过 TiKV 自己监听的地址来访问到 TiKV，这时候，你就可以设置 advertise addr 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 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

4.1.15.1.3 --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

4.1.15.1.4 -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

4.1.15.1.5 --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）

4.1.15.1.6 --data-dir

- TiKV 数据存储路径
- 默认：“/tmp/tikv/store”

4.1.15.1.7 -L, --log

- Log 级别
- 默认：“info”
- 我们能选择 trace, debug, info, warn, error, 或者 off

4.1.15.1.8 --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

4.1.15.1.9 --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

4.1.15.2 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。

4.1.15.2.1 server

服务器相关的配置项。

status-thread-pool-size

- Http API 服务的工作线程数量。
- 默认值：1
- 最小值：1

grpc-compression-type

- gRPC 消息的压缩算法，取值：none，deflate，gzip。
- 默认值：none

grpc-concurrency

- gRPC 工作线程的数量。
- 默认值：4
- 最小值：1

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

`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

`grpc-stream-initial-window-size`

- gRPC stream 的 window 大小。
- 默认值：2MB
- 单位：KB|MB|GB
- 最小值：1KB

`grpc-keepalive-time`

- gRPC 发送 keep alive ping 消息的间隔时长。
- 默认值：10s
- 最小值：1s

`grpc-keepalive-timeout`

- 关闭 gRPC 链接的超时时长。
- 默认值：3s
- 最小值：1s

`concurrent-send-snap-limit`

- 同时发送 snapshot 的最大个数，默认值：32
- 默认值：32
- 最小值：1

`concurrent-recv-snap-limit`

- 同时接受 snapshot 的最大个数，默认值：32
- 默认值：32
- 最小值：1

`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

`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

`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

`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

4.1.15.2.2 readpool.storage

存储线程池相关的配置项。

`high-concurrency`

- 处理高优先级读请求的线程池线程数量。
- 默认值：4
- 最小值：1

`normal-concurrency`

- 处理普通优先级读请求的线程池线程数量。
- 默认值：4
- 最小值：1

`low-concurrency`

- 处理低优先级读请求的线程池线程数量。
- 默认值：4
- 最小值：1

`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

`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

`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

`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

4.1.15.2.3 readpool.coprocessor

Coprocessor 线程池相关的配置项。

`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

`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

`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

`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

`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

`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

`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

4.1.15.2.4 storage

存储相关的配置项。

`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：2048000
- 最小值：1

`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。
- 默认值：4
- 最小值：1

`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

4.1.15.2.5 raftstore

raftstore 相关的配置项。

`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。

`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

`raftdb-path`

- raft 库的路径，默认存储在 storage.data-dir/raft 下。
- 默认值：“”

`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

`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

`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

`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

`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

`raft-max-size-per-message`

- 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

`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

`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

`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

`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

`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

`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

`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

`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

`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 `raftstore.peer-stale-state-check-interval` 调整 Leader 和 Follower 之间的心跳间隔。
- 默认值：false

`raftstore.peer-stale-state-check-interval`

- 修改对 Region 的状态检查间隔时间。
- 默认值：5 min

`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

`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

`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

`clean-stale-peer-delay`

- 延迟删除过期副本数据的时间。
- 默认值：10m
- 最小值：0

`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

`region-compact-min-tombstones`

- 触发 rocksdb compaction 需要的 tombstone 个数。
- 默认值：10000
- 最小值：0

`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

`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

`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

`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：5s
- 最小值：0

`snap-gc-timeout`

- snapshot 文件的最长保存时间。
- 默认值：4h
- 最小值：0

`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

`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

`notify-capacity`

- region 消息队列的最长长度。
- 默认值：40960
- 最小值：0

`messages-per-tick`

- 每轮处理的消息最大个数。
- 默认值：4096
- 最小值：0

`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

`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

`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

`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

`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：10
- 最小值：10

`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

`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

`raft-store-max-leader-lease`

- region 主可信任期的最长时间。
- 默认值：9s
- 最小值：0

`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

`allow-remove-leader`

- 允许删除主开关。
- 默认值：false

`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

`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

`use-delete-range`

- 开启 rocksdb delete_range 接口删除数据的开关。
- 默认值：false

> **警告：**
>
> 当前该功能为实验特性，不建议在生产环境中使用。

`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

`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

`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

`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

`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

`store-pool-size`

- 处理 raft 的线程池线程数。
- 默认值：2
- 最小值：大于 0

`future-poll-size`

- 驱动 future 的线程池线程数。
- 默认值：1
- 最小值：大于 0

4.1.15.2.6 coprocessor

Coprocessor 相关的配置项。

`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：true

`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

`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

`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

`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

`region-split-keys`

- 分裂后新 Region 的 key 的个数，此值属于估算值。
- 默认值：960000

4.1.15.2.7 rocksdb

rocksdb 相关的配置项。

`max-background-jobs`

- RocksDB 后台线程个数。
- 默认值：8
- 最小值：1

`max-sub-compactions`

- RocksDB 进行 subcompaction 的并发个数。
- 默认值：1
- 最小值：1

`max-open-files`

- RocksDB 可以打开的文件总数。
- 默认值：40960
- 最小值：-1

`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

`create-if-missing`

- 自动创建 DB 开关。
- 默认值：true

`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

`wal-dir`

- WAL 存储目录，默认：“tmp/tikv/store”。
- 默认值：/tmp/tikv/store

`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

`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

`enable-statistics`

- 开启 RocksDB 的统计信息。
- 默认值：true

`stats-dump-period`

- 开启或关闭 Pipelined Write。
- 默认值：true

`compaction-readahead-size`

- 异步 Sync 限速速率。
- 默认值：0
- 最小值：0
- 单位：B|KB|MB|GB

`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

`use-direct-io-for-flush-and-compaction`

- flush 或者 compaction 开启 DirectIO 的开关。
- 默认值：false

`rate-bytes-per-sec`

- Rate Limiter 限制速率。
- 默认值：0
- 最小值：0
- 单位：Bytes

`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

`auto-tuned`

- 开启自动优化 Rate LImiter 的配置的开关。
- 默认值：false

`enable-pipelined-write`

- 开启 Pipelined Write 的开关。
- 默认值：true

`bytes-per-sync`

- 异步 Sync 限速速率。

- 默认值：1MB
- 最小值：0
- 单位：B|KB|MB|GB

`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

`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

`info-log-roll-time`

- 日志截断间隔时间，如果为 0s 则不截断。
- 默认值：0s

`info-log-keep-log-file-num`

- 保留日志文件最大个数。
- 默认值：10
- 最小值：0

`info-log-dir`

- 日志存储目录。
- 默认值：“”

4.1.15.2.8 rocksdb.titan

Titan 相关的配置项。

> **警告：**
>
> 当前该功能为实验特性，不建议在严苛生产环境中使用。

`enabled`

- 开启或关闭 Titan。
- 默认值：false

`dirname`

- Titan Blob 文件存储目录。
- 默认值：titandb

`disable-gc`

- 控制是否关闭 Titan 对 Blob 文件的 GC。
- 默认值：false

`max-background-gc`

- Titan 后台 GC 的线程个数。
- 默认值：1
- 最小值：1

4.1.15.2.9 rocksdb.defaultcf

rocksdb defaultcf 相关的配置项。

`block-size`

- 设置 rocksdb block 大小。
- 默认值：64KB
- 最小值：1KB
- 单位：KB|MB|GB

`block-cache-size`

- 设置 rocksdb block 缓存大小。
- 默认值：机器总内存 / 4
- 最小值：0
- 单位：KB|MB|GB

`disable-block-cache`

- 开启或关闭 block cache。
- 默认值：false

`cache-index-and-filter-blocks`

- 开启或关闭缓存 index 和 filter。

- 默认值：true

`pin-l0-filter-and-index-blocks`

- 是否 pin 住 L0 的 index 和 filter。
- 默认值：true

`use-bloom-filter`

- 开启 bloom filter 的开关。
- 默认值：true

`optimize-filters-for-hits`

- 开启优化 filter 的命中率的开关。
- 默认值：true

`whole_key_filtering`

- 开启将整个 key 放到 bloom filter 中的开关。
- 默认值：true

`bloom-filter-bits-per-key`

bloom filter 为每个 key 预留的长度。

- 默认值：10
- 单位：字节

`block-based-bloom-filter`

- 开启每个 block 建立 bloom filter 的开关。
- 默认值：false

`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

`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”]

`write-buffer-size`

- memtable 大小。
- 默认值：128MB
- 最小值：0
- 单位：KB|MB|GB

`max-write-buffer-number`

- 最大 memtable 个数。
- 默认值：5
- 最小值：0

`min-write-buffer-number-to-merge`

- 触发 flush 的最小 memtable 个数。
- 默认值：1
- 最小值：0

`max-bytes-for-level-base`

- base level (L1) 最大字节数，一般设置为 memtable 大小 4 倍。
- 默认值：512MB
- 最小值：0
- 单位：KB|MB|GB

`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

`level0-file-num-compaction-trigger`

- 触发 compaction 的 L0 文件最大个数。
- 默认值：4
- 最小值：0

`level0-slowdown-writes-trigger`

- 触发 write stall 的 L0 文件最大个数。
- 默认值：20
- 最小值：0

`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

`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

`compaction-pri`

- Compaction 优先类型
- 可选择值：3（MinOverlappingRatio），0（ByCompensatedSize），1（OldestLargestSeqFirst），2（OldestSmallestSeq-First）。
- 默认值：3

`dynamic-level-bytes`

- 开启 dynamic level bytes 优化的开关。
- 默认值：true

`num-levels`

- RocksDB 文件最大层数。
- 默认值：7

`max-bytes-for-level-multiplier`

- 每一层的默认放大倍数。
- 默认值：10

`rocksdb.defaultcf.compaction-style`

- Compaction 方法，可选值为 level，universal。
- 默认值：level

`disable-auto-compactions`

- 开启自动 compaction 的开关。
- 默认值：false

`soft-pending-compaction-bytes-limit`

- pending compaction bytes 的软限制。
- 默认值：64GB
- 单位：KB|MB|GB

`hard-pending-compaction-bytes-limit`

- pending compaction bytes 的硬限制。
- 默认值：256GB
- 单位：KB|MB|GB

4.1.15.2.10 rocksdb.defaultcf.titan

rocksdb defaultcf titan 相关的配置项。

`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

`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

`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

`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

`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

`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

`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

`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

4.1.15.2.11 rocksdb.writecf

rocksdb writecf 相关的配置项。

`block-cache-size`

- block cache size。
- 默认值：机器总内存 * 15%
- 单位：MB|GB

`optimize-filters-for-hits`

- 开启优化 filter 的命中率的开关。
- 默认值：false

`whole-key-filtering`

- 开启将整个 key 放到 bloom filter 中的开关。
- 默认值：false

4.1.15.2.12 rocksdb.lockcf

rocksdb lockcf 相关配置项。

`block-cache-size`

- block cache size。
- 默认值：机器总内存 * 2%
- 单位：MB|GB

`optimize-filters-for-hits`

- 开启优化 filter 的命中率的开关。
- 默认值：false

`level0-file-num-compaction-trigger`

- 触发 compaction 的 L0 文件个数。
- 默认值：1

4.1.15.2.13 raftdb

raftdb 相关配置项。

`max-background-jobs`

- RocksDB 后台线程个数。
- 默认值：2
- 最小值：1

`max-sub-compactions`

- RocksDB 进行 subcompaction 的并发数。
- 默认值：2
- 最小值：1

`wal-dir`

- WAL 存储目录。
- 默认值：/tmp/tikv/store

4.1.15.2.14 security

安全相关配置项。

`ca-path`

- CA 文件路径

- 默认： “”

`cert-path`

- 包含 X509 证书的 PEM 文件路径
- 默认： “”

`key-path`

- 包含 X509 key 的 PEM 文件路径
- 默认： “”

4.1.15.2.15 import

用于 TiDB Lightning 导入及 BR 恢复相关的配置项。

`num-threads`

- 处理 RPC 请求线程数。
- 默认值： 8
- 最小值： 1

`num-import-jobs`

- 并发导入工作任务数。
- 默认值： 8
- 最小值： 1

4.1.15.2.16 backup

用于 BR 备份相关的配置项。

`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

4.1.15.2.17 pessimistic-txn

`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

`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，单位为毫秒。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：1000
- 最小值：1

`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` 毫秒之后被唤醒。
- 默认值：20

4.1.16 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陆方案。
  - MySQL 8.0 中，`mysql_native_password` 不再是默认/推荐的插件。若要使用 MySQL 8.0 的连接器连接到 TiDB，你必须显式地指定 `default-auth=mysql_native_password`。
- 不支持外部身份验证方式（如 LDAP）。
- 不支持列级别权限设置。
- 不支持使用证书验证身份。#9708
- 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
- 不支持透明数据加密（TDE）。

4.1.17 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。

4.1.17.1 权限相关操作

4.1.17.1.1 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。

```
mysql> 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 对于数据库或者表的授权，不检查数据库或表是否存在。

```
SELECT * FROM test.xxxx;
```

```
ERROR 1146 (42S02): Table 'test.xxxx' doesn't exist
```

```
GRANT ALL PRIVILEGES ON test.xxxx TO xxxx;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> SELECT user,host FROM mysql.tables_priv WHERE user='xxxx';
```

```
+------|------+
| user | host |
+------|------+
| xxxx | %    |
+------|------+
1 row in set (0.00 sec)
```

GRANT 还可以模糊匹配地授予用户数据库的权限：

```
mysql> 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。

4.1.17.1.2 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：

```
mysql> 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：

```
mysql> CREATE TABLE `select` (id int);
```

```
Query OK, 0 rows affected (0.27 sec)
```

4.1.17.1.3 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.%' |
+-------------------------------------------------------------------+
```

4.1.17.2 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
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

权限类型	权限变量名	权限简述
Show Databases	ShowDBPriv	显示 database 内的表情况

4.1.17.2.1 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。

4.1.17.2.2 CREATE DATABASE

需要拥有全局 CREATE 权限。

4.1.17.2.3 CREATE INDEX

需要对所操作的表拥有 INDEX 权限。

4.1.17.2.4 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。

4.1.17.2.5 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。

4.1.17.2.6 DROP DATABASE

需要对数据库拥有 DROP 权限。

4.1.17.2.7 DROP INDEX

需要对所操作的表拥有 `INDEX` 权限。

4.1.17.2.8 DROP TABLES

需要对所操作的表拥有 `DROP` 权限。

4.1.17.2.9 LOAD DATA

`LOAD DATA` 需要对所操作的表拥有 `INSERT` 权限。

4.1.17.2.10 TRUNCATE TABLE

需要对所操作的表拥有 `DROP` 权限。

4.1.17.2.11 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。

4.1.17.2.12 ANALYZE TABLE

需要对所操作的表拥有 `INSERT` 和 `SELECT` 权限。

4.1.17.2.13 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。

4.1.17.2.14 CREATE ROLE/USER

`CREATE ROLE` 需要 `CREATE ROLE` 权限。

`CREATE USER` 需要 `CREATE USER` 权限

4.1.17.2.15 DROP ROLE/USER

`DROP ROLE` 需要 `DROP ROLE` 权限。

`DROP USER` 需要 `CREATE USER` 权限

4.1.17.2.16 ALTER USER

`ALTER USER` 需要 `CREATE USER` 权限。

4.1.17.2.17 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。

4.1.17.2.18 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。

4.1.17.2.19 SET GLOBAL

使用 SET GLOBAL 设置全局变量需要拥有 SUPER 权限。

4.1.17.2.20 ADMIN

需要拥有 SUPER 权限。

4.1.17.2.21 SET DEFAULT ROLE

需要拥有 SUPER 权限。

4.1.17.2.22 KILL

使用 KILL 终止其他用户的会话需要拥有 SUPER 权限。

4.1.17.3 权限系统的实现

4.1.17.3.1 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';
```

4.1.17.3.2 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。

4.1.17.3.3 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。

4.1.17.3.4 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;
```

4.1.18 TiDB 用户账户管理

本文档主要介绍如何管理 TiDB 用户账户。

4.1.18.1 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
```

> **注意：**
>
> 若要使用 MySQL 8.0 的连接器连接到 TiDB，你必须显式地指定 `default-auth=mysql_native_password`，因为 `mysql_native_password` 不再是默认的插件。

4.1.18.2 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 [auth_spec] [, user [auth_spec]] ...
```

```
auth_spec: {
    IDENTIFIED BY 'auth_string'
  | IDENTIFIED BY PASSWORD 'hash_string'
}
```

- IDENTIFIED BY 'auth_string'：设置登录密码时，auth_string 会被 TiDB 经过加密存储在 mysql.user 表中。
- IDENTIFIED BY PASSWORD 'hash_string'：设置登录密码，hash_string 是一个类似于 *EBE2869D7542FCE37D1C9BBC724B97BD 的 41 位字符串，会被 TiDB 直接存储在 mysql.user 表中，该字符串可以通过 SELECT password('auth_string') 加密得到。

```
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 '';
```

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' |
+-------------------------------------------------------+
```

4.1.18.3 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。

4.1.18.4 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。

4.1.18.5 设置资源限制

暂不支持。

4.1.18.6 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 创建用户时可以通过 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';
  ```

4.1.18.7 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. 然后使用 root 登录后修改密码：

```
mysql -h 127.0.0.1 -P 4000 -u root
```

4.1.18.8 FLUSH PRIVILEGES

如果授权表已被直接修改，运行如下命令可使改动立即生效：

```
FLUSH PRIVILEGES;
```

详情参见权限管理。

4.1.19 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。

4.1.19.1 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。

4.1.19.1.1 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。

4.1.19.1.2 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';
```

4.1.19.1.3 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。

4.1.19.1.4 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。

4.1.19.1.5 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；用户在登陆时，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。

4.1.19.1.6 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，登陆 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。

4.1.19.1.7 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 登陆后：

```
SELECT CURRENT_ROLE();
```

```
+--------------------------------+
| CURRENT_ROLE()                 |
+--------------------------------+
| `app_read`@`%`,`app_write`@`%` |
+--------------------------------+
```

```
SET ROLE 'app_read'; SELECT CURRENT_ROLE();
```

```
+----------------+
| CURRENT_ROLE() |
+----------------+
| `app_read`@`%` |
+----------------+
```

4.1.19.1.8 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。

4.1.19.1.9 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。

4.1.19.1.10 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。

4.1.19.1.11 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。

4.1.19.1.12 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

4.1.20 TiDB 证书鉴权使用指南从 v3.0.8 版本开始引入

从 TiDB 3.0.8 版本开始，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
- 配置登陆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。

4.1.20.1 创建安全密钥和证书

4.1.20.1.1 安装 OpenSSL

目前推荐使用 OpenSSL 来生成密钥和证书。以 Debian 操作系统为例，先执行以下命令来安装 OpenSSL：

```
sudo apt-get install openssl
```

4.1.20.1.2 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。

4.1.20.1.3 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，用户登陆时 TiDB 将强制检查 `subject` 部分的信息是否一致。

4.1.20.1.4 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。

4.1.20.1.5 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
```

4.1.20.2 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。

4.1.20.2.1 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]
```

4.1.20.2.2 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。

4.1.20.3 配置登陆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。

4.1.20.3.1 获取用户证书信息

用户证书信息可由 `require subject`、`require issuer` 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 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';
  ```

4.1.20.3.2 配置用户证书信息

获取用户证书信息（`require subject`, `require issuer` 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`) 时配置登陆时需要校验的证书信息：

```
create user 'u1'@'%'  require issuer '<replaceable>' subject '<replaceable>' cipher '<
    ↪ replaceable>';
```

- 可以在赋予权限 (`grant`) 时配置登陆时需要校验的证书信息：

```
grant all on *.* to 'u1'@'%' require issuer '<replaceable>' subject '<replaceable>' cipher '
    ↪ <replaceable>';
```

- 还可以在修改已有用户 (alter user) 时配置登陆时需要校验的证书信息：

```
alter user 'u1'@'%' require issuer '<replaceable>' subject '<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

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)
```

4.1.20.4 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。

4.1.20.4.1 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。

4.1.20.4.2 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。

4.1.20.4.3 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。

4.2 事务

4.2.1 TiDB 事务概览

TiDB 支持完整的分布式事务，提供乐观事务与悲观事务（TiDB 3.0 中引入）两种事务模型。本文主要介绍涉及到事务的语句、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 以及tidb_retry_limit。

> 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。

4.2.1.1 常用事务语句

4.2.1.1.1 BEGIN 和 START TRANSACTION

语法：

```
BEGIN;
```

```
START TRANSACTION;
```

```
START TRANSACTION WITH CONSISTENT SNAPSHOT;
```

以上三条语句都用于开启事务，效果相同。执行开启事务语句可以显式地开启一个新的事务。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。

4.2.1.1.2 COMMIT

语法：

```
COMMIT;
```

该语句用于提交当前的事务，包括从 [BEGIN|START TRANSACTION] 到 COMMIT 之间的所有修改。

4.2.1.1.3 ROLLBACK

语法：

```
ROLLBACK;
```

该语句用于回滚当前事务，撤销从 [BEGIN|START TRANSACTION] 到 ROLLBACK 之间的所有修改。

4.2.1.2 自动提交

语法：

```
SET autocommit = {0 | 1}
```

当 autocommit = 1 时（默认），当前的 Session 为自动提交状态，即每条语句运行后，TiDB 会自动将修改提交到数据库中。设置 autocommit = 0 时更改当前 Session 更改为非自动提交状态，通过执行 COMMIT 语句来手动提交事务。

> 注意：
>
> 某些语句执行后会导致隐式提交。例如，执行 [BEGIN|START TRANCATION] 语句时，TiDB 会试图提交上一个事务，并开启一个新的事务。详情参见 implicit commit。

另外，autocommit 也是一个系统变量，你可以通过变量赋值语句修改当前 Session 或 Global 的值。

```
SET @@SESSION.autocommit = {0 | 1};
```

```
SET @@GLOBAL.autocommit = {0 | 1};
```

4.2.1.3 显式事务和隐式事务

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。

4.2.1.4 事务隔离级别

TiDB 只支持 SNAPSHOT ISOLATION，可以通过下面的语句将当前 Session 的隔离级别设置为 READ COMMITTED，这只是语法上的兼容，事务依旧是以 SNAPSHOT ISOLATION 来执行。

```
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

4.2.1.5 惰性检查

TiDB 中，对于普通的 INSERT 语句写入的值，会进行惰性检查。惰性检查的含义是，不在 INSERT 语句执行时进行唯一约束的检查，而在事务提交时进行唯一约束的检查。

举例：

```
CREATE TABLE T (I INT KEY);
INSERT INTO T VALUES (1);
BEGIN;
INSERT INTO T VALUES (1); -- MySQL 返回错误；TiDB 返回成功
INSERT INTO T VALUES (2);
COMMIT; -- MySQL 提交成功；TiDB 返回错误，事务回滚
SELECT * FROM T; -- MySQL 返回 1 2；TiDB 返回 1
```

惰性检查的意义在于，如果对事务中每个 INSERT 语句都立刻进行唯一性约束检查，将造成很高的网络开销。而在提交时进行一次批量检查，将会大幅提升性能。

> 注意：
>
> 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。

4.2.1.6 语句回滚

TiDB 支持语句执行的原子性回滚。在事务内部执行一个语句，遇到错误时，该语句整体不会生效。

```
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 (3);
commit;
```

上面的例子里面，第二条语句执行失败，但第一和第三条语句仍然能正常提交。

```
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 (3);
rollback;
```

以上例子中，第二条语句执行失败。由于调用了 ROLLBACK，因此事务不会将任何数据写入数据库。

4.2.1.7 事务大小

对于 TiDB 事务而言，事务太大或太小，都会影响事务的执行效率。

4.2.1.7.1 小事务

以如下 query 为例，当 `autocommit = 1` 时，下面三条语句各为一个事务：

```
UPDATE my_table SET a ='new_value' WHERE id = 1;
UPDATE my_table SET a ='newer_value' WHERE id = 2;
UPDATE my_table SET a ='newest_value' WHERE id = 3;
```

此时每一条语句都需要经过两阶段提交，频繁的网络交互致使延迟率高。为提升事务执行效率，可以选择使用显式事务，即在一个事务内执行三条语句。

优化后版本：

```
START TRANSACTION;
UPDATE my_table SET a ='new_value' WHERE id = 1;
UPDATE my_table SET a ='newer_value' WHERE id = 2;
UPDATE my_table SET a ='newest_value' WHERE id = 3;
COMMIT;
```

同理，执行 `INSERT` 语句时，建议使用显式事务。

> 注意：
>
> 由于 TiDB 中的资源是分布式的，TiDB 中单线程 workload 可能不会很好地利用分布式资源，因此性能相比于单实例部署的 MySQL 较低。这与 TiDB 中的事务延迟较高的情况类似。

4.2.1.7.2 大事务

由于 TiDB 两阶段提交的要求，修改数据的单个事务过大时会存在以下问题：

- 客户端在提交之前，数据都写在内存中，而数据量过多时易导致 OOM (Out of Memory) 错误。
- 在第一阶段写入数据耗时增加，与其他事务出现写冲突的概率会指数级增长。
- 最终导致事务完成提交的耗时增加。

因此，TiDB 对事务做了一些限制：

- 单个事务包含的 SQL 语句不超过 5000 条（默认）
- 每个键值对不超过 6 MB
- 键值对的总数不超过 300000
- 键值对的总大小不超过 100 MB

为了使性能达到最优，建议每 100 ~ 500 行写入一个事务。

4.2.2 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。关于该项功能对隔离级别的影响以及如何开启该项功能，请参考事务重试。

4.2.2.1 可重复读

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，那么该事务会回滚并启动自动重试。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;
commit;                         |
                                |               commit; -- 事务提交失败，回滚
```

4.2.2.1.1 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。

4.2.2.1.2 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快照隔离级别，MySQL 可重复读隔离级别的一致性要弱于快照隔离级别，也弱于 TiDB 的可重复读隔离级别。

4.2.2.2 更多阅读

- TiKV 的 MVCC (Multi-Version Concurrency Control) 机制

4.2.3 TiDB 乐观事务模型

本文介绍 TiDB 乐观事务的原理，以及相关特性。本文假定你对TiDB 的整体架构、Percolator 事务模型以及事务的ACID 特性都有一定了解。

TiDB 的乐观事务模型只有在两阶段事务提交时才会检测是否存在写写冲突。

> **注意：**
>
> 自 v3.0.8 开始，TiDB 默认使用悲观事务模型。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。

4.2.3.1 乐观事务原理

TiDB 中事务使用两阶段提交，流程如下：

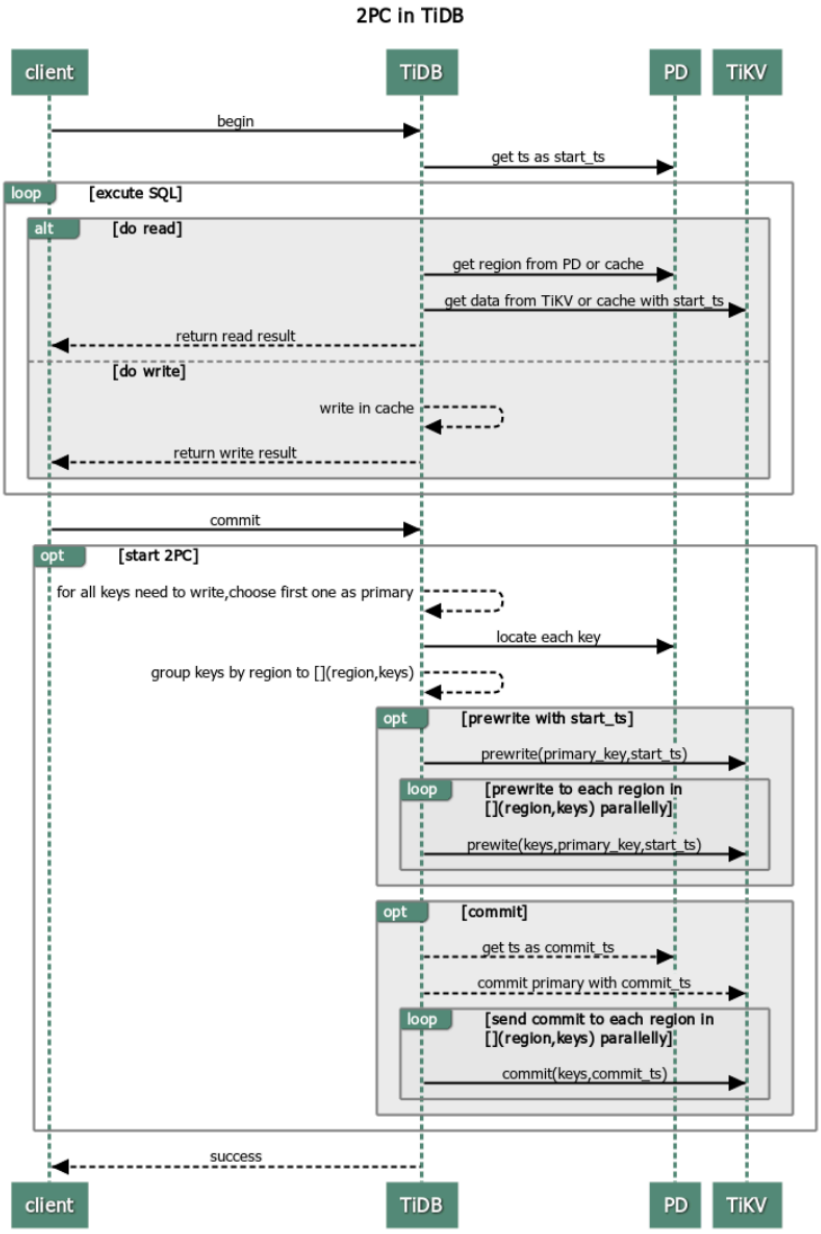


图 290: TiDB 中的两阶段提交

1. 客户端开始一个事务。

   TiDB 从 PD 获取一个全局唯一递增的版本号作为当前事务的开始版本号，这里定义为该事务的 `start_ts` 版本。

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` 版本下对应的数据信息。

3. 客户端发起写请求。

   TiDB 校验写入数据是否符合一致性约束（如数据类型是否正确、是否符合唯一索引约束等）。校验通过的数据将存放在内存里。

4. 客户端发起 commit。

5. TiDB 开始两阶段提交，保证分布式事务的原子性，让数据真正落盘。

   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。

4.2.3.2 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。
- 需要一个中心化的版本管理服务。
- 事务数据量过大时易导致内存暴涨。

实际应用中，你可以根据事务的大小进行针对性处理，以提高事务的执行效率。

4.2.3.3 事务的重试

使用乐观事务模型时，在高冲突率的场景中，事务很容易提交失败。而 MySQL 内部使用的是悲观事务模型，在执行 SQL 语句的过程中进行冲突检测，所以提交时很难出现异常。为了兼容 MySQL 的悲观事务行为，TiDB 提供了重试机制。

4.2.3.3.1 重试机制

当事务提交后，如果发现冲突，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 (`try again later`) 给应用层。

4.2.3.3.2 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。

4.2.3.4 冲突检测

乐观事务下，检测底层数据是否存在写写冲突是一个很重要的操作。具体而言，TiKV 在 prewrite 阶段就需要读取数据进行检测。为了优化这一块性能，TiDB 集群会在内存里面进行一次冲突预检测。

作为一个分布式系统，TiDB 在内存中的冲突检测主要在两个模块进行：

- TiDB 层。如果发现 TiDB 实例本身就存在写写冲突，那么第一个写入发出后，后面的写入已经清楚地知道自己冲突了，无需再往下层 TiKV 发送请求去检测冲突。
- TiKV 层。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。

其中 TiDB 层的冲突检测可以根据场景需要选择打开或关闭，具体配置项如下：

```
### 事务内存锁相关配置，当本地事务冲突比较多时建议开启。
[txn-local-latches]
### 是否开启内存锁，默认为 false，即不开启。
enabled = false
### Hash 对应的 slot 数，会自动向上调整为 2 的指数倍。
### 每个 slot 占 32 Bytes 内存。当写入数据的范围比较广时（如导数据），
### 设置过小会导致变慢，性能下降。（默认为 2048000）
capacity = 2048000
```

配置项 capacity 主要影响到冲突判断的正确性。在实现冲突检测时，不可能把所有的 Key 都存到内存里，所以真正存下来的是每个 Key 的 Hash 值。有 Hash 算法就有碰撞也就是误判的概率，这里可以通过配置 capacity 来控制 Hash 取模的值：

- capacity 值越小，占用内存小，误判概率越大。
- capacity 值越大，占用内存大，误判概率越小。

实际应用时，如果业务场景能够预判断写入不存在冲突（如导入数据操作），建议关闭冲突检测。

相应地，在 TiKV 层检测内存中是否存在冲突也有类似的机制。不同的是，TiKV 层的检测会更严格且不允许关闭，仅支持对 Hash 取模值进行配置：

```
### 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：

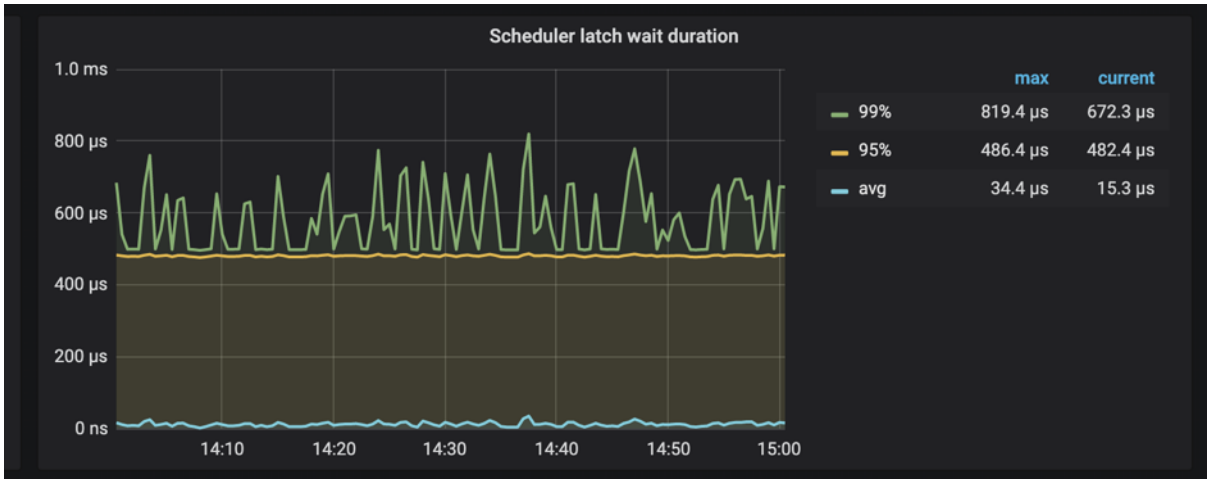


图 291: Scheduler latch wait duration

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。

4.2.3.5 更多阅读

- Percolator 和 TiDB 事务算法

4.2.4 TiDB 悲观事务模型

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。

4.2.4.1 悲观事务的使用方法

进入悲观事务模式有以下三种方式:

- 执行 `BEGIN PESSIMISTIC;` 语句开启的事务，会进入悲观事务模式。可以通过写成注释的形式 `BEGIN /*!90000 PESSIMISTIC */;` 来兼容 MySQL 语法。
- 执行 `set @@tidb_txn_mode = 'pessimistic';`，使这个 session 执行的所有显式事务（即非 autocommit 的事务）都会进入悲观事务模式。
- 执行 `set @@global.tidb_txn_mode = 'pessimistic';`，使之后整个集群所有新创建 session 执行的所有显示事务（即非 autocommit 的事务）都会进入悲观事务模式。

在配置了 `global.tidb_txn_mode` 为 `pessimistic` 之后，默认进入悲观事务模式，但是可以用以下三种方式使事务进入乐观事务模式：

- 执行 `BEGIN OPTIMISTIC;` 语句开启的事务，会进入乐观事务模式。可以通过写成注释的形式 `BEGIN /*!90000 OPTIMISTIC */;` 来兼容 MySQL 语法。

- 执行 `set @@tidb_txn_mode = 'optimistic';` 或 `set @@tidb_txn_mode = '';`，使当前的 session 执行的事务进入乐观事务模式。
- 执行 `set @@global.tidb_txn_mode = 'optimistic;'` 或 `set @@global.tidb_txn_mode = '';`，使之后整个集群所有新创建 session 执行的事务都进入乐观事务模式。

`BEGIN PESSIMISTIC;` 和 `BEGIN OPTIMISTIC;` 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。

如果想要禁用悲观事务特性，可以修改 TiDB 配置文件，在 `[pessimistic-txn]` 类别下添加 `enable = false`。

4.2.4.2 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）：

- `SELECT FOR UPDATE` 会读取已提交的最新数据，并对读取到的数据加悲观锁。
- `UPDATE`、`DELETE` 和 `INSERT` 语句都会读取已提交的最新的数据来执行，并对修改的数据加悲观锁。
- 当一行数据被加了悲观锁以后，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。
- 当一行数据被加了悲观锁以后，其他尝试读取这一行的事务不会被阻塞，可以读到已提交的数据。
- 事务提交或回滚的时候，会释放所有的锁。
- 当有多个事务同时等待同一个锁释放时，会尽可能按照事务 start ts 顺序获取锁，但不能严格保证。
- 如果并发事务出现死锁，会被死锁检测器检测到，随机终止掉其中一个事务并返回兼容 MySQL 的错误码 1213。
- 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。
- 通过设置 `innodb_lock_wait_timeout` 变量，设置等锁超时时间，等锁超时后返回兼容 MySQL 的错误码 1205。

4.2.4.3 和 MySQL InnoDB 的差异

1. TiDB 使用 range 作为 `WHERE` 条件，执行 DML 和 `SELECT FOR UPDATE` 语句时不会阻塞范围内并发的 `INSERT` 语句的执行。

   InnoDB 通过实现 gap lock，支持阻塞 range 内并发的 `INSERT` 语句的执行，其主要目的是为了支持 statement based binlog，因此有些业务会通过将隔离级别降低至 `READ COMMITTED` 来避免 gap lock 导致的并发性能问题。TiDB 不支持 gap lock，也就不需要付出相应的并发性能的代价。

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 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]`。

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 事务不支持悲观锁

   所有自动提交的语句都不会加悲观锁，该类语句在用户侧感知不到区别，因为悲观事务的本质是把整个事务的重试变成了单个 DML 的重试，autocommit 事务即使在 TiDB 关闭重试时也会自动重试，效果和悲观事务相同。

   自动提交的 select for update 语句也不会等锁。

4.2.4.4 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。

   除了有事务执行时间不能超出 tikv_gc_life_time 的限制外，悲观事务的 TTL 有 10 分钟上限，所以执行时间超过 10 分钟的悲观事务有可能提交失败。

4.3 系统数据库

4.3.1 TiDB 系统表

本文档主要介绍 TiDB 支持的系统表。

4.3.1.1 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

4.3.1.2 服务端帮助信息系统表

- help_topic 目前为空

4.3.1.3 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

4.3.1.4 GC Worker 相关系统表

- gc_delete_range

4.3.1.5 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

4.3.2 Information Schema

为了和 MySQL 保持兼容，TiDB 支持很多 INFORMATION_SCHEMA 表，其中有不少表都支持相应的 SHOW 命令。查询 INFORMATION_SCHEMA 表也为表的连接操作提供了可能。

4.3.2.1 ANALYZE_STATUS 表

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。

```
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
```

4.3.2.2 CHARACTER_SETS 表

CHARACTER_SETS 表提供字符集相关的信息。TiDB 目前仅支持部分字符集。

```
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)
```

4.3.2.3 COLLATIONS 表

COLLATIONS 表提供了 CHARACTER_SETS 表中字符集对应的排序规则列表。TiDB 当前仅支持二进制排序规则，包含该表仅为兼容 MySQL。

```
SELECT * FROM collations WHERE character_set_name='utf8mb4';
```

```
+-----------------------+--------------------+------+------------+-------------+---------+
| COLLATION_NAME        | CHARACTER_SET_NAME | ID   | IS_DEFAULT | IS_COMPILED | SORTLEN |
+-----------------------+--------------------+------+------------+-------------+---------+
| utf8mb4_general_ci    | utf8mb4            |   45 | Yes        | Yes         |       1 |
| utf8mb4_bin           | utf8mb4            |   46 |            | Yes         |       1 |
| utf8mb4_unicode_ci    | utf8mb4            |  224 |            | Yes         |       1 |
| utf8mb4_icelandic_ci  | utf8mb4            |  225 |            | Yes         |       1 |
| utf8mb4_latvian_ci    | utf8mb4            |  226 |            | Yes         |       1 |
| utf8mb4_romanian_ci   | utf8mb4            |  227 |            | Yes         |       1 |
| utf8mb4_slovenian_ci  | utf8mb4            |  228 |            | Yes         |       1 |
| utf8mb4_polish_ci     | utf8mb4            |  229 |            | Yes         |       1 |
| utf8mb4_estonian_ci   | utf8mb4            |  230 |            | Yes         |       1 |
| utf8mb4_spanish_ci    | utf8mb4            |  231 |            | Yes         |       1 |
| utf8mb4_swedish_ci    | utf8mb4            |  232 |            | Yes         |       1 |
| utf8mb4_turkish_ci    | utf8mb4            |  233 |            | Yes         |       1 |
| utf8mb4_czech_ci      | utf8mb4            |  234 |            | Yes         |       1 |
| utf8mb4_danish_ci     | utf8mb4            |  235 |            | Yes         |       1 |
```

```
| utf8mb4_lithuanian_ci  | utf8mb4            |  236 |            | Yes         |       1 |
| utf8mb4_slovak_ci      | utf8mb4            |  237 |            | Yes         |       1 |
| utf8mb4_spanish2_ci    | utf8mb4            |  238 |            | Yes         |       1 |
| utf8mb4_roman_ci       | utf8mb4            |  239 |            | Yes         |       1 |
| utf8mb4_persian_ci     | utf8mb4            |  240 |            | Yes         |       1 |
| utf8mb4_esperanto_ci   | utf8mb4            |  241 |            | Yes         |       1 |
| utf8mb4_hungarian_ci   | utf8mb4            |  242 |            | Yes         |       1 |
| utf8mb4_sinhala_ci     | utf8mb4            |  243 |            | Yes         |       1 |
| utf8mb4_german2_ci     | utf8mb4            |  244 |            | Yes         |       1 |
| utf8mb4_croatian_ci    | utf8mb4            |  245 |            | Yes         |       1 |
| utf8mb4_unicode_520_ci | utf8mb4            |  246 |            | Yes         |       1 |
| utf8mb4_vietnamese_ci  | utf8mb4            |  247 |            | Yes         |       1 |
+------------------------+--------------------+------+------------+-------------+---------+
26 rows in set (0.00 sec)
```

4.3.2.4 COLLATION_CHARACTER_SET_APPLICABILITY 表

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，包含此表也是为了兼容 MySQL。

```
SELECT * FROM collation_character_set_applicability WHERE character_set_name='utf8mb4';
```

```
+------------------------+--------------------+
| COLLATION_NAME         | CHARACTER_SET_NAME |
+------------------------+--------------------+
| utf8mb4_general_ci     | utf8mb4            |
| utf8mb4_bin            | utf8mb4            |
| utf8mb4_unicode_ci     | utf8mb4            |
| utf8mb4_icelandic_ci   | utf8mb4            |
| utf8mb4_latvian_ci     | utf8mb4            |
| utf8mb4_romanian_ci    | utf8mb4            |
| utf8mb4_slovenian_ci   | utf8mb4            |
| utf8mb4_polish_ci      | utf8mb4            |
| utf8mb4_estonian_ci    | utf8mb4            |
| utf8mb4_spanish_ci     | utf8mb4            |
| utf8mb4_swedish_ci     | utf8mb4            |
| utf8mb4_turkish_ci     | utf8mb4            |
| utf8mb4_czech_ci       | utf8mb4            |
| utf8mb4_danish_ci      | utf8mb4            |
| utf8mb4_lithuanian_ci  | utf8mb4            |
| utf8mb4_slovak_ci      | utf8mb4            |
| utf8mb4_spanish2_ci    | utf8mb4            |
| utf8mb4_roman_ci       | utf8mb4            |
| utf8mb4_persian_ci     | utf8mb4            |
| utf8mb4_esperanto_ci   | utf8mb4            |
```

```
| utf8mb4_hungarian_ci   | utf8mb4            |
| utf8mb4_sinhala_ci     | utf8mb4            |
| utf8mb4_german2_ci     | utf8mb4            |
| utf8mb4_croatian_ci    | utf8mb4            |
| utf8mb4_unicode_520_ci | utf8mb4            |
| utf8mb4_vietnamese_ci  | utf8mb4            |
+------------------------+--------------------+
26 rows in set (0.00 sec)
```

4.3.2.5 COLUMNS 表

COLUMNS 表提供了表的所有列的信息。

```
CREATE TABLE test.t1 (a int);
```

```
1 row in set (0.01 sec)
```

```
SELECT * FROM information_schema.columns WHERE table_schema='test' AND TABLE_NAME='t1';
```

```
*************************** 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 (0.01 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，则为 NO。
- DATA_TYPE：列数据类型。
- 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)
```

4.3.2.6 ENGINES 表

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。

```
SELECT * FROM engines;
```

```
*************************** 1. row ***************************
      ENGINE: InnoDB
     SUPPORT: DEFAULT
     COMMENT: Supports transactions, row-level locking, and foreign keys
TRANSACTIONS: YES
          XA: YES
  SAVEPOINTS: YES
1 row in set (0.00 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。

4.3.2.7 KEY_COLUMN_USAGE 表

KEY_COLUMN_USAGE 表描述了列的键约束，比如主键约束。

```
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。

4.3.2.8 PROCESSLIST 表

PROCESSLIST 和 show processlist 的功能一样，都是查看当前正在处理的请求。

PROCESSLIST 表会比 show processlist 多一个 MEM 列，MEM 是指正在处理的请求已使用的内存，单位是 byte。

```
+--
    ↪ --+------+------+--------------------+---------+------+-------+---------------------------+-----+
    ↪
| ID | USER | HOST | DB                 | COMMAND | TIME | STATE | INFO                      |
    ↪ MEM |
+--
    ↪ --+------+------+--------------------+---------+------+-------+---------------------------+-----+
    ↪
| 1  | root | ::1  | INFORMATION_SCHEMA | Query   | 0    | 2     | select * from PROCESSLIST | 0
    ↪   |
+--
    ↪ --+------+------+--------------------+---------+------+-------+---------------------------+-----+
    ↪
```

4.3.2.9 SCHEMATA 表

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。

```
SELECT * FROM schemata;
```

```
+--------------+--------------------+----------------------------+------------------------+----------+
    ↪
| CATALOG_NAME | SCHEMA_NAME        | DEFAULT_CHARACTER_SET_NAME | DEFAULT_COLLATION_NAME |
    ↪ SQL_PATH |
+--------------+--------------------+----------------------------+------------------------+----------+
    ↪
| def          | INFORMATION_SCHEMA | utf8mb4                    | utf8mb4_bin            | NULL
    ↪      |
| def          | mynewdb            | 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)
```

4.3.2.10 SESSION_VARIABLES 表

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。

```
SELECT * FROM session_variables LIMIT 10;
```

```
+---------------------------------+----------------------+
| VARIABLE_NAME                   | VARIABLE_VALUE       |
+---------------------------------+----------------------+
| max_write_lock_count            | 18446744073709551615 |
| server_id_bits                  | 32                   |
| net_read_timeout                | 30                   |
| innodb_online_alter_log_max_size | 134217728            |
| innodb_optimize_fulltext_only   | OFF                  |
| max_join_size                   | 18446744073709551615 |
| innodb_read_io_threads          | 4                    |
| session_track_gtids             | OFF                  |
| have_ssl                        | DISABLED             |
| max_binlog_cache_size           | 18446744073709547520 |
```

```
+----------------------------------+----------------------+
10 rows in set (0.00 sec)
```

4.3.2.11 SLOW_QUERY 表

SLOW_QUERY 表中提供了慢查询相关的信息，其内容通过解析 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。

```
mysql> desc information_schema.slow_query;
+---------------+---------------------+------+------+---------+-------+
| Field         | Type                | Null | Key  | Default | Extra |
+---------------+---------------------+------+------+---------+-------+
| Time          | timestamp unsigned  | 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 unsigned     | YES  |      | NULL    |       |
| Process_time  | double unsigned     | YES  |      | NULL    |       |
| Wait_time     | double unsigned     | YES  |      | NULL    |       |
| Backoff_time  | double unsigned     | 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_ids     | varchar(100)        | YES  |      | NULL    |       |
| Is_internal   | tinyint(1) unsigned | YES  |      | NULL    |       |
| Digest        | varchar(64)         | YES  |      | NULL    |       |
| Stats         | varchar(512)        | YES  |      | NULL    |       |
| Cop_proc_avg  | double unsigned     | YES  |      | NULL    |       |
| Cop_proc_p90  | double unsigned     | YES  |      | NULL    |       |
| Cop_proc_max  | double unsigned     | YES  |      | NULL    |       |
| Cop_proc_addr | varchar(64)         | YES  |      | NULL    |       |
| Cop_wait_avg  | double unsigned     | YES  |      | NULL    |       |
| Cop_wait_p90  | double unsigned     | YES  |      | NULL    |       |
| Cop_wait_max  | double unsigned     | YES  |      | NULL    |       |
| Cop_wait_addr | varchar(64)         | YES  |      | NULL    |       |
| Mem_max       | bigint(20) unsigned | YES  |      | NULL    |       |
| Succ          | tinyint(1) unsigned | YES  |      | NULL    |       |
| Query         | longblob unsigned   | YES  |      | NULL    |       |
+---------------+---------------------+------+------+---------+-------+
```

4.3.2.12 STATISTICS 表

STATISTICS 表提供了关于表索引的信息。

```
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) UNSIGNED  | YES  |      | NULL    |       |
| COLUMN_NAME   | varchar(21)         | YES  |      | NULL    |       |
| COLLATION     | varchar(1)          | YES  |      | NULL    |       |
| CARDINALITY   | bigint(21) UNSIGNED | YES  |      | NULL    |       |
| SUB_PART      | bigint(3) UNSIGNED  | 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    |       |
+---------------|---------------------|------|------|---------|-------+
```

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
```

4.3.2.13 TABLES 表

TABLES 表提供了数据库里面关于表的信息。

```
SELECT * FROM tables WHERE table_schema='mysql' AND table_name='user';
```

```
*************************** 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: 0
    CREATE_TIME: 2019-03-29 09:17:27
    UPDATE_TIME: NULL
     CHECK_TIME: NULL
TABLE_COLLATION: utf8mb4_bin
       CHECKSUM: NULL
 CREATE_OPTIONS:
  TABLE_COMMENT:
  TIDB_TABLE_ID: 5
1 row in set (0.00 sec)
```

以下操作是等价的：

```
SELECT table_name FROM INFORMATION_SCHEMA.TABLES
  WHERE table_schema = 'db_name'
  [AND table_name LIKE 'wild']
```

```
SHOW TABLES
  FROM db_name
  [LIKE 'wild']
```

4.3.2.14 TABLE_CONSTRAINTS 表

TABLE_CONSTRAINTS 表记录了表的约束信息。

```
SELECT * FROM table_constraints WHERE constraint_type='UNIQUE';
```

```
*************************** 1. row ***************************
CONSTRAINT_CATALOG: def
 CONSTRAINT_SCHEMA: mysql
   CONSTRAINT_NAME: name
      TABLE_SCHEMA: mysql
        TABLE_NAME: help_topic
   CONSTRAINT_TYPE: UNIQUE
*************************** 2. row ***************************
CONSTRAINT_CATALOG: def
 CONSTRAINT_SCHEMA: mysql
   CONSTRAINT_NAME: tbl
```

```
      TABLE_SCHEMA: mysql
        TABLE_NAME: stats_meta
   CONSTRAINT_TYPE: UNIQUE
*************************** 3. row ***************************
CONSTRAINT_CATALOG: def
 CONSTRAINT_SCHEMA: mysql
   CONSTRAINT_NAME: tbl
      TABLE_SCHEMA: mysql
        TABLE_NAME: stats_histograms
   CONSTRAINT_TYPE: UNIQUE
*************************** 4. row ***************************
CONSTRAINT_CATALOG: def
 CONSTRAINT_SCHEMA: mysql
   CONSTRAINT_NAME: tbl
      TABLE_SCHEMA: mysql
        TABLE_NAME: stats_buckets
   CONSTRAINT_TYPE: UNIQUE
*************************** 5. row ***************************
CONSTRAINT_CATALOG: def
 CONSTRAINT_SCHEMA: mysql
   CONSTRAINT_NAME: delete_range_index
      TABLE_SCHEMA: mysql
        TABLE_NAME: gc_delete_range
   CONSTRAINT_TYPE: UNIQUE
*************************** 6. row ***************************
CONSTRAINT_CATALOG: def
 CONSTRAINT_SCHEMA: mysql
   CONSTRAINT_NAME: delete_range_done_index
      TABLE_SCHEMA: mysql
        TABLE_NAME: gc_delete_range_done
   CONSTRAINT_TYPE: UNIQUE
6 rows in set (0.00 sec)
```

其中：

- CONSTRAINT_TYPE 的取值可以是 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。

4.3.2.15 TIDB_HOT_REGIONS 表

TIDB_HOT_REGIONS 表提供了关于热点 REGION 的相关信息。

```
desc TIDB_HOT_REGIONS;
```

```
+---------------+--------------------+------+-----+---------+-------+
| Field         | Type               | Null | Key | Default | Extra |
```

```
+----------------+---------------------+------+-----+---------+-------+
| TABLE_ID       | bigint(21) unsigned | YES  |     | <null>  |       |
| INDEX_ID       | bigint(21) unsigned | YES  |     | <null>  |       |
| DB_NAME        | varchar(64)         | YES  |     | <null>  |       |
| TABLE_NAME     | varchar(64)         | YES  |     | <null>  |       |
| INDEX_NAME     | varchar(64)         | YES  |     | <null>  |       |
| TYPE           | varchar(64)         | YES  |     | <null>  |       |
| MAX_HOT_DEGREE | bigint(21) unsigned | YES  |     | <null>  |       |
| REGION_COUNT   | bigint(21) unsigned | YES  |     | <null>  |       |
| FLOW_BYTES     | bigint(21) unsigned | YES  |     | <null>  |       |
+----------------+---------------------+------+-----+---------+-------+
```

4.3.2.16 TIDB_INDEXES 表

TIDB_INDEXES 记录了所有表中的 INDEX 信息。

```
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) unsigned | YES  |     | <null>  |       |
| KEY_NAME      | varchar(64)         | YES  |     | <null>  |       |
| SEQ_IN_INDEX  | bigint(21) unsigned | YES  |     | <null>  |       |
| COLUMN_NAME   | varchar(64)         | YES  |     | <null>  |       |
| SUB_PART      | bigint(21) unsigned | YES  |     | <null>  |       |
| INDEX_COMMENT | varchar(2048)       | YES  |     | <null>  |       |
| INDEX_ID      | bigint(21) unsigned | YES  |     | <null>  |       |
+---------------+---------------------+------+-----+---------+-------+
```

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。

4.3.2.17 TIKV_REGION_PEERS 表

TIKV_REGION_PEERS 表提供了所有 REGION 的 peer 信息。

```
desc TIKV_REGION_PEERS;
```

```
+--------------+---------------------+------+-----+---------+-------+
| Field        | Type                | Null | Key | Default | Extra |
+--------------+---------------------+------+-----+---------+-------+
| REGION_ID    | bigint(21) unsigned | YES  |     | <null>  |       |
| PEER_ID      | bigint(21) unsigned | YES  |     | <null>  |       |
| STORE_ID     | bigint(21) unsigned | YES  |     | <null>  |       |
| IS_LEARNER   | tinyint(1) unsigned | YES  |     | <null>  |       |
| IS_LEADER    | tinyint(1) unsigned | YES  |     | <null>  |       |
| STATUS       | varchar(10)         | YES  |     | <null>  |       |
| DOWN_SECONDS | bigint(21) unsigned | YES  |     | <null>  |       |
+--------------+---------------------+------+-----+---------+-------+
```

4.3.2.18 TIKV_REGION_STATUS 表

TIKV_REGION_STATUS 表提供了所有 REGION 的状态信息。

```
desc TIKV_REGION_STATUS;
```

```
+------------------+---------------------+------+-----+---------+-------+
| Field            | Type                | Null | Key | Default | Extra |
+------------------+---------------------+------+-----+---------+-------+
| REGION_ID        | bigint(21) unsigned | YES  |     | <null>  |       |
| START_KEY        | text                | YES  |     | <null>  |       |
| END_KEY          | text                | YES  |     | <null>  |       |
| EPOCH_CONF_VER   | bigint(21) unsigned | YES  |     | <null>  |       |
| EPOCH_VERSION    | bigint(21) unsigned | YES  |     | <null>  |       |
| WRITTEN_BYTES    | bigint(21) unsigned | YES  |     | <null>  |       |
| READ_BYTES       | bigint(21) unsigned | YES  |     | <null>  |       |
| APPROXIMATE_SIZE | bigint(21) unsigned | YES  |     | <null>  |       |
| APPROXIMATE_KEYS | bigint(21) unsigned | YES  |     | <null>  |       |
+------------------+---------------------+------+-----+---------+-------+
```

4.3.2.19 TIKV_STORE_STATUS 表

TIKV_STORE_STATUS 表提供了所有 TiKV Store 的状态信息。

```
desc TIKV_STORE_STATUS;
```

```
+------------------+---------------------+------+-----+---------+-------+
| Field             | Type                | Null | Key | Default | Extra |
```

```
+-------------------+---------------------+------+-----+---------+-------+
| STORE_ID          | bigint(21) unsigned | YES  |     | <null>  |       |
| ADDRESS           | varchar(64)         | YES  |     | <null>  |       |
| STORE_STATE       | bigint(21) unsigned | YES  |     | <null>  |       |
| STORE_STATE_NAME  | varchar(64)         | YES  |     | <null>  |       |
| LABEL             | json unsigned       | YES  |     | <null>  |       |
| VERSION           | varchar(64)         | YES  |     | <null>  |       |
| CAPACITY          | varchar(64)         | YES  |     | <null>  |       |
| AVAILABLE         | varchar(64)         | YES  |     | <null>  |       |
| LEADER_COUNT      | bigint(21) unsigned | YES  |     | <null>  |       |
| LEADER_WEIGHT     | bigint(21) unsigned | YES  |     | <null>  |       |
| LEADER_SCORE      | bigint(21) unsigned | YES  |     | <null>  |       |
| LEADER_SIZE       | bigint(21) unsigned | YES  |     | <null>  |       |
| REGION_COUNT      | bigint(21) unsigned | YES  |     | <null>  |       |
| REGION_WEIGHT     | bigint(21) unsigned | YES  |     | <null>  |       |
| REGION_SCORE      | bigint(21) unsigned | YES  |     | <null>  |       |
| REGION_SIZE       | bigint(21) unsigned | YES  |     | <null>  |       |
| START_TS          | datetime unsigned   | YES  |     | <null>  |       |
| LAST_HEARTBEAT_TS | datetime unsigned   | YES  |     | <null>  |       |
| UPTIME            | varchar(64)         | YES  |     | <null>  |       |
+-------------------+---------------------+------+-----+---------+-------+
```

4.3.2.20 USER_PRIVILEGES 表

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。

```
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)
```

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。

4.3.2.21 VIEWS 表

VIEWS 表提供了关于 SQL 视图的信息。

```
create view test.v1 as select 1;
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
select * from views;
```

```
*************************** 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: utf8
COLLATION_CONNECTION: utf8_general_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。

4.3.2.22 不支持的 Information Schema 表

TiDB 包含以下 INFORMATION_SCHEMA 表，但仅会返回空行：

- COLUMN_PRIVILEGES
- EVENTS
- FILES
- GLOBAL_STATUS
- GLOBAL_VARIABLES
- OPTIMIZER_TRACE

- PARAMETERS
- PARTITIONS
- PLUGINS
- PROFILING
- REFERENTIAL_CONSTRAINTS
- ROUTINES
- SCHEMA_PRIVILEGES
- SESSION_STATUS
- TABLESPACES
- TABLE_PRIVILEGES
- TRIGGERS

4.4 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。

4.4.1 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。另外还有一些特有的错误码：

错误码	说明
8001	请求使用的内存超过 TiDB 内存使用的阈值限制
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
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
8004	单个事务过大，原因及解决方法请参考这里
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
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
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
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
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
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
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
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

4.4.2 故障诊断

参见故障诊断文档以及FAQ。

4.5 连接器和 API

数据库连接器为客户端提供了连接数据库服务端的方式，APIs 提供了使用 MySQL 协议和资源的底层接口。无论是连接器还是 API，都可以用来在不同的语言和环境内连接服务器并执行 sql 语句，包括 odbc、java(jdbc)、Perl、Python、PHP、Ruby 和 C。

TiDB 兼容 MySQL(5.6、5.7) 的所有连接器和 API，包括：

- MySQL Connector/C++
- MySQL Connector/J
- MySQL Connector/Net
- MySQL Connector/ODBC
- MySQL Connector/Python
- MySQL C API
- MySQL PHP API
- MySQL Perl API
- MySQL Python API
- MySQL Ruby APIs
- MySQL Tcl API
- MySQL Eiffel Wrapper
- Mysql Go API

> 注意：
>
> 若要使用 MySQL 8.0 的连接器连接到 TiDB，你必须显式地指定 `default-auth=mysql_native_password`，因为 `mysql_native_password` 不再是默认的插件。

4.5.1 使用 MySQL 连接器连接 TiDB

Oracle 官方提供了以下 API，TiDB 可以兼容所有这些 API。

- MySQL Connector/C++：C++ 语言的客户端库
- MySQL Connector/J：Java 语言的客户端库，基于标准 JDBC 接口
- MySQL Connector/Net：.Net 语言的客户端库，MySQL for Visual Studio使用这个库，支持 Microsoft Visual Studio 2012，2013，2015 和 2017 版本
- MySQL Connector/ODBC：标准的 ODBC 接口，支持 Windows，Unix 和 OS X
- MySQL Connector/Python：Python 语言的客户端包，和 Python DB API version 2.0 一致

4.5.2 使用 MySQL C API 连接 TiDB

如果使用 C 语言程序直接连接 TiDB，可以直接链接 libmysqlclient 库，使用 MySQL 的 C API，这是最主要的一种 C 语言连接方式，被各种客户端和 API 广泛使用，包括 Connector/C。

4.5.3 使用 MySQL 第三方 API 连接 TiDB

第三方 API 非 Oracle 官方提供，下表列出了常用的第三方 API：

Environment	API	Type	Notes
Ada	GNU Ada MySQL Bindings	`libmysqlclient`	See MySQL Bindings for GNU Ada
C	C API	`libmysqlclient`	See MySQL C API.
C++	Connector/C++	`libmysqlclient`	See MySQL Connector/C++ Developer Guide.
	MySQL++	`libmysqlclient`	See MySQL++ Web site.
	MySQL wrapped	`libmysqlclient`	See MySQL wrapped.
Go	go-sql-driver	Native Driver	See Mysql Go API
Cocoa	MySQL-Cocoa	`libmysqlclient`	Compatible with the Objective-C Cocoa environment. See http://mysql-cocoa.sourceforge.net/
D	MySQL for D	`libmysqlclient`	See MySQL for D.
Eiffel	Eiffel MySQL	`libmysqlclient`	See Section 27.14, “MySQL Eiffel Wrapper”.
Erlang	`erlang-mysql-driver`	`libmysqlclient`	See `erlang-mysql-driver`.
Haskell	Haskell MySQL Bindings	Native Driver	See Brian O’Sullivan’s pure Haskell MySQL bindings.
	`hsql-mysql`	`libmysqlclient`	See MySQL driver for Haskell.
Java/JDBC	Connector/J	Native Driver	See MySQL Connector/J 5.1 Developer Guide.
Lua	LuaSQL	`libmysqlclient`	See LuaSQL.
.NET/Mono	Connector/Net	Native Driver	See MySQL Connector/Net Developer Guide.
Objective Caml	OBjective Caml MySQL Bindings	`libmysqlclient`	See MySQL Bindings for Objective Caml.
Octave	Database bindings for GNU Octave	`libmysqlclient`	See Database bindings for GNU Octave.
ODBC	Connector/ODBC	`libmysqlclient`	See MySQL Connector/ODBC Developer Guide.
Perl	`DBI/DBD::mysql`	`libmysqlclient`	See Section 27.10, “MySQL Perl API”.
	`Net::MySQL`	Native Driver	See `Net::MySQL` at CPAN
PHP	`mysql`, `ext/mysql`interface (deprecated)	`libmysqlclient`	See Original MySQL API.
	`mysqli`, `ext/mysqli`interface	`libmysqlclient`	See MySQL Improved Extension.
	PDO_MYSQL	`libmysqlclient`	See MySQL Functions (PDO_MYSQL).

Environment	API	Type	Notes
	PDO mysqlnd	Native Driver	
Python	Connector/Python	Native Driver	See MySQL Connector/Python Developer Guide.
Python	Connector/Python C Extension	`libmysqlclient`	See MySQL Connector/Python Developer Guide.
	MySQLdb	`libmysqlclient`	See Section 27.11, “MySQL Python API”.
Ruby	MySQL/Ruby	`libmysqlclient`	Uses `libmysqlclient`. See Section 27.12.1, “The MySQL/Ruby API”.
	Ruby/MySQL	Native Driver	See Section 27.12.2, “The Ruby/MySQL API”.
Scheme	`Myscsh`	`libmysqlclient`	See `Myscsh`.
SPL	`sql_mysql`	`libmysqlclient`	See `sql_mysql` for SPL.
Tcl	MySQLtcl	`libmysqlclient`	See Section 27.13, “MySQL Tcl API”.

4.5.4 TiDB 支持的连接器版本

Connector	Connector version
Connector/C	6.1.0 GA
Connector/C++	1.0.5 GA
Connector/J	5.1.8
Connector/Net	6.9.9 GA
Connector/Net	6.8.8 GA
Connector/ODBC	5.1
Connector/ODBC	3.51 (Unicode not supported)
Connector/Python	2.0
Connector/Python	1.2

4.6 垃圾回收 (GC)

4.6.1 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。

4.6.1.1 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，默认情况下每 10 分钟一次。每次 GC 时，首先，TiDB 会计算一个称为 safe point 的时间戳

（默认为当前时间减去 10 分钟），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。具体而言，分为以下三个步骤：

1. Resolve Locks
2. Delete Ranges
3. Do GC

4.6.1.1.1 Resolve Locks

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 会加上一个锁，其中一个锁会被设定为 Primary，其余的锁（Secondary）则会指向 Primary；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。如果替换这些 Secondary 锁的线程死掉了，锁就残留了下来。

Resolve Locks 这一步的任务即对 safe point 之前的锁进行回滚或提交，取决于其 Primary 是否被提交。如果一个 Primary 锁也残留了下来，那么该事务应当视为超时并进行回滚。这一步是必不可少的，因为如果其 Primary 的 Write 记录由于太老而被 GC 清除掉了，那么就再也无法知道该事务是否成功。如果该事务存在残留的 Secondary 锁，那么也无法知道它应当被回滚还是提交，也就无法保证一致性。

Resolve Locks 的执行方式是由 GC leader 对所有的 Region 发送请求进行处理。从 3.0 起，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。

4.6.1.2 Delete Ranges

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。

4.6.1.3 Do GC

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，但是会保留 safe point 前的最后一次写入（除非最后一次写入是删除）。

TiDB 2.1 及更早版本使用的 GC 方式是由 GC leader 向所有 Region 发送 GC 请求。从 3.0 起，GC leader 只需将 safe point 上传至 PD。每个 TiKV 节点都会各自从 PD 获取 safe point。当 TiKV 发现 safe point 发生更新时，便会对当前节点上所有作为 leader 的 Region 进行 GC。与此同时，GC leader 可以继续触发下一轮 GC。

> 注意：
>
> 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。

4.6.2 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;
```

```
+--------------------------+-----------------------------------------------------------------------------------------
    ↪
| VARIABLE_NAME            | VARIABLE_VALUE
    ↪                                                                                 |
+--------------------------+-----------------------------------------------------------------------------------------
    ↪
| bootstrapped             | True
    ↪
    ↪ |
| tidb_server_version      | 33
    ↪
    ↪ |
| system_tz                | UTC
    ↪
    ↪ |
| 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：

```
update mysql.tidb set VARIABLE_VALUE="24h" where VARIABLE_NAME="tikv_gc_life_time";
```

> 注意：
>
> mysql.tidb 系统表中除了下文将要列出的 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：上次 GC 运行时间
> - tikv_gc_safe_point：当前 GC 的 safe point

4.6.2.1 tikv_gc_enable

- 控制是否启用 GC。
- 默认值：true

4.6.2.2 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"

4.6.2.3 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：
>
> - tikv_gc_life_time 的值必须大于 TiDB 的配置文件中的max-txn-time-use 的值至少 10 秒，且不低于 10 分钟。
> - 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`）。

4.6.2.4 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。

4.6.2.5 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

4.6.2.6 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

4.6.2.7 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` `tikv_gc_auto_concurrency = false`	并行	并行
3.0 `tikv_gc_mode = centered` `tikv_gc_auto_concurrency = true`	自动并行	自动并行

版本/配置	Resolve Locks	Do GC
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。

4.6.2.8 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 -m server -n gc.max_write_bytes_per_sec -v 10MB
```

4.7 性能调优

4.7.1 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：
>
> 1. 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；
> 2. 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。

4.7.1.1 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。

4.7.1.1.1 60s 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。

1. uptime
2. dmesg | tail
3. vmstat 1
4. mpstat -P ALL 1
5. pidstat 1
6. iostat -xz 1
7. free -m
8. sar -n DEV 1
9. sar -n TCP,ETCP 1
10. top

具体用法可查询相应 man 手册。

4.7.1.1.2 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，性能监视单元）功能和软件功能（软件计数器，跟踪点）。详细用法请参考 perf Examples。

4.7.1.1.3 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)。

4.7.1.2 性能调优

性能调优将根据内核子系统进行分类描述。

4.7.1.2.1 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。命令为 `cpupower frequency-set --governor performance`。

4.7.1.2.2 处理器 ——中断亲和性

- 自动平衡：可通过 `irqbalance` 服务实现。
- 手动平衡：
  1. 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；
  2. 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。

4.7.1.2.3 NUMA 绑核

为尽可能的避免跨 NUMA 访问内存，我们可以通过设置线程的 CPU 亲和性来实现。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。

4.7.1.2.4 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。伴随使能 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。

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

4.7.1.2.5 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。

4.7.1.2.6 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。

4.7.1.2.7 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有许多要点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/interrrputs 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 10Gbps 足够。但若从 `ip -s link` 的输出中观察到 TX errors 值时，可尝试加倍：`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。

4.7.2 SQL 优化流程简介

在 TiDB 中，SQL 优化过程分为逻辑优化和物理优化两个阶段。

4.7.2.1 逻辑优化简介

逻辑优化是基于规则的优化，对输入的逻辑执行计划按顺序应用一些优化规则，从而使整个逻辑执行计划变得更好。这些优化规则包括：

- 列裁剪
- 投影消除
- 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

4.7.2.2 物理优化简介

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。

逻辑执行计划是一个树形结构，每个节点对应 SQL 中的一个逻辑算子。同样的，物理执行计划也是一个树形结构，每个节点对应 SQL 中的一个物理算子。逻辑算子只描述这个算子的功能，而物理算子则描述了完成这个功能的具体算法。对于同一个逻辑算子，可能有多个物理算子实现，比如 `LogicalAggregate`，它的实现可以是采用哈希算法的 `HashAggregate`，也可以是流式的 `StreamAggregate`。不同的物理算子具有不同的物理属性，也对其子节点有着不同的物理属性的要求。物理属性包括数据的顺序和分布等。TiDB 中现在只考虑了数据的顺序。

4.7.3 理解 TiDB 执行计划

TiDB 优化器会根据当前数据表的实际情况来选择最优的执行计划，执行计划由一系列的算子构成。本文将详细解释 TiDB 中 `EXPLAIN` 语句返回的执行计划信息。

4.7.3.1 使用 EXPLAIN 来优化 SQL 语句

`EXPLAIN` 语句的返回结果提供了 TiDB 执行 SQL 查询的详细信息：

- `EXPLAIN` 可以和 `SELECT`，`DELETE` 语句一起使用；
- 执行 `EXPLAIN`，TiDB 会返回被 `EXPLAIN` 的 SQL 语句经过优化器后的最终物理执行计划。也就是说，`EXPLAIN` 展示了 TiDB 执行该 SQL 语句的完整信息，比如以什么样的顺序，什么方式 JOIN 两个表，表达式树长什么样等等。详见EXPLAIN 输出格式；
- 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`。

通过观察 `EXPLAIN` 的结果，你可以知道如何给数据表添加索引使得执行计划使用索引从而加速 SQL 语句的执行速度；你也可以使用 `EXPLAIN` 来检查优化器是否选择了最优的顺序来 JOIN 数据表。

4.7.3.2 EXPLAIN 输出格式

目前 TiDB 的 EXPLAIN 会输出 4 列，分别是：id，count，task，operator info。执行计划中每个算子都由这 4 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，在整个执行计划中唯一的标识一个算子。在 TiDB 2.1 中，id 会格式化显示算子的树状结构。数据从 child 流向 parent，每个算子的 parent 有且仅有一个。
count	预计当前算子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。
task	当前这个算子属于什么 task。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，并行的在 TiKV 上执行。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 上的任务，每个 cop task 分散在 TiKV 集群中，由多个进程共同执行。
operator info	每个算子的详细信息。各个算子的 operator info 各有不同，详见Operator Info。

4.7.3.3 EXPLAIN ANALYZE 输出格式

作为 EXPLAIN 语句的扩展，EXPLAIN ANALYZE 语句执行查询并在 execution info 列中提供额外的执行统计信息。具体如下：

- time 显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。
- rows 是当前算子返回的行的总数。例如，可以将 count 列的精度和 execution_info 列中的 rows/loops 值进行对比，据此评定查询优化器估算的精确度。

4.7.3.3.1 用例

使用 bikeshare example database:

```
EXPLAIN SELECT count(*) FROM trips WHERE start_date BETWEEN '2017-07-01 00:00:00' AND '2017-07-01
    ↪  23:59:59';
```

```
+-------------------------+-------------+------+-----------------------------------------------------------
    ↪
| id                      | count       | task | operator info
    ↪
    ↪ |
+-------------------------+-------------+------+-----------------------------------------------------------
    ↪
| StreamAgg_20            | 1.00        | root | funcs:count(col_0)
    ↪
    ↪ |
| └─TableReader_21        | 1.00        | root | data:StreamAgg_9
    ↪
    ↪ |
|   └─StreamAgg_9         | 1.00        | cop  | funcs:count(1)
    ↪
    ↪ |
```

```
|     └─Selection_19       | 8166.73     | cop  | ge(bikeshare.trips.start_date, 2017-07-01
    ↪ 00:00:00.000000), le(bikeshare.trips.start_date, 2017-07-01 23:59:59.000000) |
|       └─TableScan_18     | 19117643.00 | cop  | table:trips, range:[-inf,+inf], keep order:
    ↪ false                                                                        |
+--------------------------+-------------+------+--------------------------------------------------------------------------------
    ↪
5 rows in set (0.00 sec)
```

在上面的例子中，coprocessor 上读取 `trips` 表上的数据（`TableScan_18`），寻找满足 `start_date BETWEEN '2017-07-01 00:00:00' AND '2017-07-01 23:59:59'` 条件的数据（`Selection_19`），然后计算满足条件的数据行数（`StreamAgg_9`），最后把结果返回给 TiDB。TiDB 汇总各个 coprocessor 返回的结果（`TableReader_21`），并进一步计算所有数据的行数（`StreamAgg_20`），最终把结果返回给客户端。在上面这个查询中，TiDB 根据 `trips` 表的统计信息估算出 `TableScan_18` 的输出结果行数为 19117643.00，满足条件 `start_date BETWEEN '2017-07-01 00:00:00' AND '2017-07-01 23:59:59'` 的有 8166.73 条，经过聚合运算后，只有 1 条结果。

上述查询中，虽然大部分计算逻辑都下推到了 TiKV 的 coprocessor 上，但是其执行效率还是不够高，可以添加适当的索引来消除 `TableScan_18` 对 `trips` 的全表扫，进一步加速查询的执行：

```
ALTER TABLE trips ADD INDEX (start_date);
```

```
EXPLAIN SELECT count(*) FROM trips WHERE start_date BETWEEN '2017-07-01 00:00:00' AND '2017-07-01
    ↪  23:59:59';
```

```
+------------------------+---------+------+-------------------------------------------------------------------------------
    ↪
| id                     | count   | task | operator info
    ↪                                                                                |
+------------------------+---------+------+-------------------------------------------------------------------------------
    ↪
| StreamAgg_25           | 1.00    | root | funcs:count(col_0)
    ↪                                                                                |
| └─IndexReader_26       | 1.00    | root | index:StreamAgg_9
    ↪                                                                                 |
|   └─StreamAgg_9        | 1.00    | cop  | funcs:count(1)
    ↪                                                                                |
|     └─IndexScan_24     | 8166.73 | cop  | table:trips, index:start_date, range:[2017-07-01
    ↪ 00:00:00,2017-07-01 23:59:59], keep order:false |
+------------------------+---------+------+-------------------------------------------------------------------------------
    ↪
4 rows in set (0.01 sec)
```

在添加完索引后的新执行计划中，使用 `IndexScan_24` 直接读取满足条件 `start_date BETWEEN '2017-07-01 00:00:00' AND '2017-07-01 23:59:59'` 的数据，可以看到，估算的要扫描的数据行数从之前的 19117643.00 降到了现在的 8166.73。在测试环境中显示，这个查询的执行时间从 50.41 秒降到了 0.01 秒！

4.7.3.4 EXPLAIN FOR CONNECTION

EXPLAIN FOR CONNECTION 用于获得一个连接中最后执行的查询的执行计划，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：

- MySQL 返回的是正在执行的查询计划，而 TiDB 返回的是最后执行的查询计划。
- 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。

4.7.3.5 概述

4.7.3.5.1 Task 简介

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。

4.7.3.5.2 表数据和索引数据

TiDB 的表数据是指一张表的原始数据，存放在 TiKV 中。对于每行表数据，它的 key 是一个 64 位整数，称为 Handle ID。如果一张表存在 int 类型的主键，TiDB 会把主键的值当作表数据的 Handle ID，否则由系统自动生成 Handle ID。表数据的 value 由这一行的所有数据编码而成。在读取表数据的时候，可以按照 Handle ID 递增的顺序返回。

TiDB 的索引数据和表数据一样，也存放在 TiKV 中。它的 key 是由索引列编码的有序 bytes，value 是这一行索引数据对应的 Handle ID，通过 Handle ID 可以读取这一行的非索引列。在读取索引数据的时候，TiKV 会按照索引列递增的顺序返回，如果有多个索引列，首先保证第 1 列递增，并且在第 i 列相等的情况下，保证第 i + 1 列递增。

4.7.3.5.3 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。

值得注意的是，TiDB 目前只支持比较符一端是列，另一端是常量，或可以计算成某一常量的情况，类似 `year(birth_day)< 1992` 的查询条件是不能利用索引的。还要注意应尽可能使用同一类型进行比较，以避免引入额外的 cast 操作而导致不能利用索引，如 `user_id = 123456`，如果 `user_id` 是字符串，需要将 123456 也写成字符串常量的形式。

针对同一列的范围查询条件使用 AND 和 OR 组合后，等于对范围求交集或者并集。对于多维组合索引，可以写多个列的条件。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。

4.7.3.6 Operator Info

4.7.3.6.1 TableReader 和 TableScan

TableScan 表示在 KV 端对表数据进行扫描，TableReader 表示在 TiDB 端从 TiKV 端读取，属于同一功能的两个算子。table 表示 SQL 语句中的表名，如果表名被重命名，则显示重命名。range 表示扫描的数据范围，如果在查询中不指定 WHERE/HAVING/ON 条件，则会选择全表扫描，如果在 int 类型的主键上有范围查询条件，会选择范围查询。keep order 表示 table scan 是否按顺序返回。

4.7.3.6.2 IndexReader 和 IndexLookUp

Index 在 TiDB 端的读取方式有两种：IndexReader 表示直接从索引中读取索引列，适用于 SQL 语句中仅引用了该索引相关的列或主键；IndexLookUp 表示从索引中过滤部分数据，仅返回这些数据的 Handle ID，通过 Handle ID 再次查找表数据，这种方式需要两次从 TiKV 获取数据。Index 的读取方式是由优化器自动选择的。

IndexScan 是 KV 端读取索引数据的算子，和 TableScan 功能类似。table 表示 SQL 语句中的表名，如果表名被重命名，则显示重命名。index 表示索引名。range 表示扫描的数据范围。out of order 表示 index scan 是否按照顺序返回。注意在 TiDB 中，多列或者非 int 列构成的主键是当作唯一索引处理的。

4.7.3.6.3 Selection

Selection 表示 SQL 语句中的选择条件，通常出现在 WHERE/HAVING/ON 子句中。

4.7.3.6.4 Projection

Projection 对应 SQL 语句中的 SELECT 列表，功能是将每一条输入数据映射成新的输出数据。

4.7.3.6.5 Aggregation

Aggregation 对应 SQL 语句中的 Group By 语句或者没有 Group By 语句但是存在聚合函数，例如 count 或 sum 函数等。TiDB 支持两种聚合算法：Hash Aggregation 以及 Stream Aggregation（待补充）。Hash Aggregation 是基于哈希的聚合算法，如果 Hash Aggregation 紧邻 Table 或者 Index 的读取算子，则聚合算子会在 TiKV 端进行预聚合，以提高计算的并行度和减少网络开销。

4.7.3.6.6 Join

TiDB 支持 Inner Join 以及 Left/Right Outer Join，并会自动将可以化简的外连接转换为 Inner Join。

TiDB 支持三种 Join 算法：Hash Join，Sort Merge Join 和 Index Look up Join。Hash Join 的原理是将参与连接的小表预先装载到内存中，读取大表的所有数据进行连接。Sort Merge Join 会利用输入数据的有序信息，同时读取两张表的数据并依次进行比较。Index Look Up Join 会读取外表的数据，并对内表进行主键或索引键查询。

4.7.3.6.7 Apply

Apply 是 TiDB 用来描述子查询的一种算子，行为类似于 Nested Loop，即每次从外表中取一条数据，带入到内表的关联列中，并执行，最后根据 Apply 内联的 Join 算法进行连接计算。

值得注意的是，Apply 一般会被查询优化器自动转换为 Join 操作。用户在编写 SQL 的过程中应尽量避免 Apply 算子的出现。

4.7.4 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。

4.7.4.1 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。

4.7.4.1.1 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
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

4.7.4.1.2 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;
```

4.7.4.2 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。

4.7.4.2.1 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()
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()

4.7.4.2.2 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：

```
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。

4.7.4.2.3 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。

4.7.4.2.4 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（参见使用 EXPLAIN 来优化 SQL 语句）。

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)
```

4.7.5 执行计划绑定

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，但有的时候需要在不修改 SQL 语句的情况下干预执行计划的选择。执行计划绑定提供了一系列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。

4.7.5.1 语法

4.7.5.1.1 创建绑定

```
CREATE [GLOBAL | SESSION] BINDING FOR SelectStmt USING SelectStmt;
```

该语句可以在 GLOBAL 或者 SESSION 作用域内为 SQL 绑定执行计划。在不指定作用域时，隐式作用域为 SESSION。被绑定的 SQL 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 和系统表中某个被绑定的 SQL 一致即可使用相应的优化器 Hint。

参数化：把 SQL 中的常量变成变量参数，并对 SQL 中的空格和换行符等做标准化处理。例如：

```
select * from t where a >    1
-- 参数化后：
select * from t where a > ?
```

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 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 t where a > ?`，而绑定 SQL 在经过处理后是 `select * from t where b > ?`。

4.7.5.1.2 删除绑定

```
DROP [GLOBAL | SESSION] BINDING FOR SelectStmt;
```

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。

4.7.5.1.3 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）和 invalid（无效）
create_time	创建时间
update_time	更新时间
charset	字符集
collation	排序规则

4.7.6 统计信息简介

TiDB 优化器会根据统计信息来选择最优的执行计划。统计信息收集了表级别和列级别的信息，表的统计信息包括总行数和修改的行数。列的统计信息包括不同值的数量、NULL 的数量、直方图、列上出现次数最多的值 TOPN 以及该列的 Count-Min Sketch 信息。

4.7.6.1 统计信息的收集

4.7.6.1.1 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];
```

WITH NUM BUCKETS 可以用来指定生成直方图的桶数量上限。

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];
```

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];
```

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];
```

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];
```

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];
```

4.7.6.1.2 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。

在查询语句执行时，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。可通过配置文件修改 feedback-probability，其默认值是 0.05。设置成 0.0 可以关闭这个功能。

> **注意：**
>
> 在配置文件中如果将 feedback-probability 设置为 0 会导致设置失败并报错。需要设置成 0.0 才可以关闭 feedback-probability。

4.7.6.1.3 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。

4.7.6.1.4 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

语法元素	说明
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）

4.7.6.2 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。

4.7.6.2.1 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：

```
SHOW STATS_META [ShowLikeOrWhere];
```

该语句会输出所有表的总行数以及修改行数等信息，你可以通过 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。

4.7.6.2.2 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`。

通过以下命令来查看表的统计信息健康度，你可以通过 `ShowLikeOrWhere` 来筛选需要的信息：

```
SHOW STATS_HEALTHY [ShowLikeOrWhere];
```

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
table_name	表名
partition_name	分区名
healthy	健康度

4.7.6.2.3 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

4.7.6.2.4 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];
```

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

语法元素	说明
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

4.7.6.3 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。

4.7.6.4 统计信息的导入导出

4.7.6.4.1 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}
```

4.7.6.4.2 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。

4.7.7 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。

4.7.7.1 示例

以下通过一些例子对 TopN 下推进行说明。

4.7.7.1.1 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。

4.7.7.1.2 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。

4.7.7.1.3 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。

4.7.7.1.4 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。

4.7.8 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。

4.7.8.1 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。

4.7.8.2 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。

4.7.8.2.1 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。

4.7.8.2.2 SM_JOIN(t1_name [, tl_name …])

`SM_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 /*+ SM_JOIN(t1, t2) */ * FROM t1, t2 WHERE t1.id = t2.id;
```

> 注意：
>
> `SM_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 `SM_JOIN`。

4.7.8.2.3 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`。

4.7.8.2.4 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：

```
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`。

4.7.8.2.5 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：

```
SELECT /*+ HASH_AGG() */ count(*) FROM t1, t2 WHERE t1.a > 10 GROUP BY t1.id;
```

4.7.8.2.6 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：

```
SELECT /*+ STREAM_AGG() */ count(*) FROM t1, t2 WHERE t1.a > 10 GROUP BY t1.id;
```

4.7.8.2.7 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。

4.7.8.2.8 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;
```

4.7.8.2.9 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;
```

4.7.8.2.10 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。例如：

```
SELECT /*+ READ_FROM_STORAGE(TIFLASH[t1], TIKV[t2]) */ t1.a FROM t t1, t t2 WHERE t1.a = t2.a;
```

4.7.8.2.11 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。

4.7.8.2.12 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。

4.7.8.2.13 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。

4.7.8.2.14 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。

4.7.9 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。

4.7.9.1 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，配合 TiDB Snapshot Isolation 事务隔离级别，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。

4.7.9.2 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。

4.7.9.3 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。

4.7.9.3.1 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。

4.7.9.3.2 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。

4.7.10 使用 SQL 语句检查 TiDB 集群状态

为了方便排查问题，TiDB 提供了一些 SQL 语句和系统表以查询一些有用的信息。

INFORMATION_SCHEMA 中提供了如下几个系统表，用于查询集群状态，诊断常见的集群问题。

- TABLES
- TIDB_INDEXES
- ANALYZE_STATUS
- TIDB_HOT_REGIONS
- TIKV_STORE_STATUS
- TIKV_REGION_STATUS
- TIKV_REGION_PEERS

除此之外，执行下列语句也可获得对排查问题或查询集群状态有用的信息：

- ADMIN SHOW DDL 可以获得是 DDL owner 角色的 TiDB 的 ID 及 IP:PORT 等具体信息。
- SHOW ANALYZE STATUS 和ANALYZE_STATUS 表的功能相同。
- 特殊的 EXPLAIN 语句：
  - EXPLAIN ANALYZE 语句可以获得一个 SQL 语句执行中的一些具体信息。
  - EXPLAIN FOR CONNECTION 可以获得一个连接中最后执行的查询的执行计划。可以配合 SHOW PROCESSLIST 使用。
  - 关于 EXPLAIN 相关的更具体的信息，参考文档理解 TiDB 执行计划。

4.7.11 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 3.1.0-beta 版本开始，TiDB 也提供系统表 events_statements_summary_by_digest，从 3.1.0-beta.1 开始提供系统表 events_statements_summary_by_digest_history。本文将详细介绍这两张表，以及如何利用它们来排查 SQL 性能问题。

4.7.11.1 events_statements_summary_by_digest

events_statements_summary_by_digest 是 performance_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;
```

规一化后都是：

```
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。

events_statements_summary_by_digest 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，events_statements_summary_by_digest 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。

因为 TiDB 中的很多概念不同于 MySQL，所以 TiDB 中 events_statements_summary_by_digest 的表结构与 MySQL 中的有很大区别。

以下为查询 events_statements_summary_by_digest 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_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)。

4.7.11.1.1 表的字段介绍

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

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_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

和 TiKV Coprocessor Task 相关的字段：

- COP_TASK_NUM：每条 SQL 发送的 Coprocessor 请求数量
- AVG_COP_PROCESS_TIME：cop-task 的平均处理时间
- 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
- AVG_COP_WAIT_TIME：cop-task 的平均等待时间
- 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

4.7.11.2 events_statements_summary_by_digest_history

events_statements_summary_by_digest_history 的表结构与 events_statements_summary_by_digest 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。

4.7.11.3 排查示例

下面用两个示例问题演示如何利用 statement summary 来排查。

4.7.11.3.1 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 performance_schema.events_statements_summary_by_digest
    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)
```

4.7.11.3.2 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 performance_schema.events_statements_summary_by_digest_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)
```

4.7.11.4 参数配置

statement summary 功能默认关闭，通过设置系统变量打开，例如：

```
set global tidb_enable_stmt_summary = true;
```

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。

还有两个控制 statement summary 的系统变量：

- `tidb_stmt_summary_refresh_interval`：`events_statements_summary_by_digest` 的清空周期，单位是秒 (s)，默认值是 1800。
- `tidb_stmt_summary_history_size`：`events_statements_summary_by_digest_history` 保存每种 SQL 的历史的数量，默认值是 24。

statement summary 配置示例如下：

```
set global tidb_stmt_summary_refresh_interval = 1800;
set global tidb_stmt_summary_history_size = 24;
```

以上配置生效后，events_statements_summary_by_digest 每 30 分钟清空一次，events_statements_summary_by_digest_history 保存最近 12 小时的历史数据。

以上两个系统变量都有 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

由于 statement summary tables 是内存表，为了防止内存问题，需要限制保存的 SQL 条数和 SQL 的最大显示长度。这两个参数都在 config.toml 的 [stmt-summary] 类别下配置：

- 通过 max-stmt-count 更改保存的 SQL 种类数量，默认 200 条。当 SQL 种类超过 max-stmt-count 时，会移除最近没有使用的 SQL。
- 通过 max-sql-length 更改 DIGEST_TEXT 和 QUERY_SAMPLE_TEXT 的最大显示长度，默认是 4096。

> 注意：
>
> tidb_stmt_summary_history_size、max-stmt-count、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。

4.7.11.5 目前的限制

Statement summary tables 现在还存在一些限制：

- 查询 statement summary tables 时，只会显示当前 TiDB server 的 statement summary，而不是整个集群的 statement summary。
- TiDB server 重启后 statement summary 会丢失。因为 statement summary tables 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。

4.7.12 TiKV 性能参数调优

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。

4.7.12.1 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 = 1

### 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"
```

4.7.12.2 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

4.7.12.3 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

4.7.13 TiDB 最佳实践

4.7.14 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。

4.8 监控指标

4.8.1 Overview 面板重要监控指标详解

使用 TiDB Ansible 部署 TiDB 集群时，一键部署监控系统 (Prometheus/Grafana)，监控架构请看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：

4.8.1.1 Services Port Status

- Services Online：各服务在线节点数量
- Services Offline：各服务 Down 掉节点数量

4.8.1.2 PD

- 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
- Number of Regions：当前集群的 Region 总量
- Leader Balance Ratio：Leader 数量最多和最少节点相差的百分比，一般小于 5%，节点重启时会有比较大的波动
- Region Balance Ratio：Region 数量最多和最少节点相差的百分比，一般小于 5%，新增/下线节点时相差比较大
- Store Status：集群 TiKV 节点的状态
  - Up Stores：正常运行的 TiKV 节点数量
  - Disconnect Stores：短时间内通信异常的 TiKV 节点数量
  - LowSpace Stores：剩余可用空间小于 20% 的 TiKV 节点数量
  - Down Stores：停止工作的 TiKV 节点数量，如果大于 0，说明有节点不正常
  - Offline Stores：正在下线的 TiKV 节点数量（正在下线的 TiKV 节点还在提供服务）
  - Tombstone Stores：下线成功的 TiKV 节点数量
- 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

4.8.1.3 TiDB

- Statement OPS：SQL 执行数量统计（包含 select、 insert、 update 等）
- Duration：SQL 执行的时间
- QPS By Instance：每个 TiDB 上的 QPS
- Failed Query OPM：失败 SQL 的统计，例如语法错误、主键冲突等
- Connection count：每个 TiDB 的连接数
- Heap Memory Usage：每个 TiDB 使用的堆内存大小
- Transaction OPS：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 从 PD 获取 TSO 的数量
- PD TSO Wait Duration：TiDB 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：事务冲突相关的数量
- 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（事务冲突等）

4.8.1.4 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：各个 TiKV 节点存储的数据量
- cf size：集群不同 CF 存储的数据量
- channel full：正常情况显示 No data，如果有了监控值，说明对应 TiKV 节点的消息处理不过来了
- server report failures：正常情况显示 No data，如果出现了 Unreachable，说明 TiKV 之间通信有问题
- scheduler pending commands：写入堆积的数量，偶尔出现峰值属于正常现象
- coprocessor pending requests：正常情况监控为 0 或者数量很少
- coprocessor executor count：不同类型的查询操作数量
- coprocessor request duration：TiKV 中查询消耗的时间
- 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：TiKV 查询线程的 CPU 使用率，和业务相关，复杂查询会使用大量的 CPU 资源

4.8.1.5 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：网络监控，TCP 相关信息统计
- IO Util：磁盘使用率，最高为 100%，一般到 80% - 90% 就需要考虑加节点

4.8.1.6 图例

图 292: overview

4.8.2 TiDB 重要监控指标详解

使用 TiDB Ansible 部署 TiDB 集群时，一键部署监控系统 (Prometheus/Grafana)，监控架构请看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 (其中 TiDB 面板包含 TiDB Summary 面板的内容)。

以下为 TiDB Dashboard 部分监控说明：

4.8.2.1 说明

- Query Summary
  - Duration：SQL 执行的时间
  - QPS：每个 TiDB 上的 SQL 执行结果按照 OK/Error 统计
  - Statement OPS：SQL 执行数量统计 (包含 SELECT、 INSERT、 UPDATE 等)
  - QPS By Instance：每个 TiDB 上的 QPS

  - Failed Query OPM：每个 TiDB 实例上，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）
  - 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）
  - 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：整个集群执行 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：TiDB 内部 SQL 语句执行数量统计

- Server
  - Uptime：TiDB 运行时间
  - Memory Usage：不同 TiDB 实例的内存使用统计
  - CPU Usage：不同 TiDB 实例的 CPU 使用统计
  - Connection Count：每个 TiDB 的连接数
  - Open FD Count：不同 TiDB 实例的打开的文件描述符统计
  - Goroutine Count：不同 TiDB 实例的 Goroutine 数量
  - Go GC Duration：不同 TiDB 实例的 GC 耗时统计
  - Go Threads：不同 TiDB 实例的线程数量
  - Go GC Count：不同 TiDB 实例的 GC 执行次数统计
  - Go GC CPU Usage：不同 TiDB 实例的 GC CPU 统计
  - Events OPM：统计关键事件，例如 start，close，graceful-shutdown，kill，hang 等
  - Keep Alive OPM：不同 TiDB 实例每分钟刷新监控的次数
  - Prepare Statement Count：不同 TiDB 实例执行 Prepare 语句数以及总数统计
  - Time Jump Back OPS：不同 TiDB 实例上每秒钟时间回跳的次数
  - Heap Memory Usage：每个 TiDB 使用的堆内存大小
  - Uncommon Error OPM：TiDB 非正常错误的统计，包括 panic，binlog 写失败等
  - Handshake Error OPS：不同 TiDB 实例每秒握手错误的次数统计
  - Get Token Duration：建立连接后获取 Token 耗时

- Transaction
  - Transaction OPS：事务执行数量统计
  - Duration：事务执行的时间
  - Transaction Retry Num：事务重试次数
  - Transaction Statement Num：一个事务中的 SQL 语句数量
  - Session Retry Error OPS：事务重试时遇到的错误数量
  - Local Latch Wait Duration：本地事务等待时间

- Executor
  - Parse Duration：SQL 语句解析耗时统计
  - Compile Duration：将 SQL AST 编译成执行计划耗时统计
  - Execution Duration：SQL 语句执行耗时统计
  - Expensive Executor OPS：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：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 Retry Duration：KV 重试请求的时间
  - 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（事务冲突等）
  - Lock Resolve OPS：事务冲突相关的数量
  - Other Errors OPS：其他类型的错误数量，包括清锁和更新 SafePoint
- KV Duration
  - KV Request Duration 999 by store：KV Request 执行时间，根据 TiKV 显示
  - KV Request Duration 999 by type：KV Request 执行时间，根据请求类型显示
  - KV Cmd Duration 99/999：KV 命令执行的时间
- KV Count
  - KV Cmd OPS：KV 命令执行数量统计
  - KV Txn OPS：启动事务的数量统计
  - Txn Regions Num 90：事务使用的 Region 数量统计
  - Txn Write Size Bytes 100：事务写入的字节数统计
  - Txn Write KV Num 100：事务写入的 KV 数量统计
  - Load SafePoint OPS：更新 SafePoint 的数量统计
- PD Client
  - PD Client CMD OPS：PD Client 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 执行命令失败统计
  - PD TSO OPS：TiDB 从 PD 获取 TSO 的数量
  - PD TSO Wait Duration：TiDB 从 PD 获取 TSO 的时间
  - PD TSO RPC Duration：TiDB 从调用 PD 获取 TSO gRPC 接口花费的时间
- Schema Load
  - Load Schema Duration：TiDB 从 TiKV 获取 Schema 的时间
  - Load Schema OPS：TiDB 从 TiKV 获取 Schema 的数量统计
  - Schema Lease Error OPM：Schema Lease 出错，包括 change 和 outdate 两种，出现 outdate 错误时会报警
- DDL
  - DDL Duration 95：DDL 语句处理时间统计
  - Batch Add Index Duration 100：创建索引时每个 Batch 所花费的时间统计
  - 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

  - 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

- 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
  - Update Stats OPS：利用 Feedback 更新统计信息的数量统计
  - Significant Feedback：重要的 Feedback 更新统计信息的数量统计

- 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
  - Meta Operations Duration 99：Meta 操作延迟

- GC

  - Worker Action OPM：GC 相关操作的数量统计，包括 run_job，resolve_lock，delete_range 等操作
  - Duration 99：GC 相关操作的耗时统计
  - GC Failure OPM：GC 相关操作失败数量统计
  - Action Result OPM：GC 相关操作结果数量统计
  - Too Many Locks Error OPM：GC 清锁过多错误的数量统计

- Batch Client

  - Pending Request Count by TiKV：等待处理的 Batch 消息数量统计
  - Wait Duration 95：等待处理的 Batch 消息延迟统计
  - Batch Client Unavailable Duration 95：Batch 客户端不可用的时间统计

4.8.3 PD 重要监控指标详解

使用 TiDB Ansible 部署 TiDB 集群时，一键部署监控系统 (Prometheus/Grafana)，监控架构请看TiDB 监控框架概述。

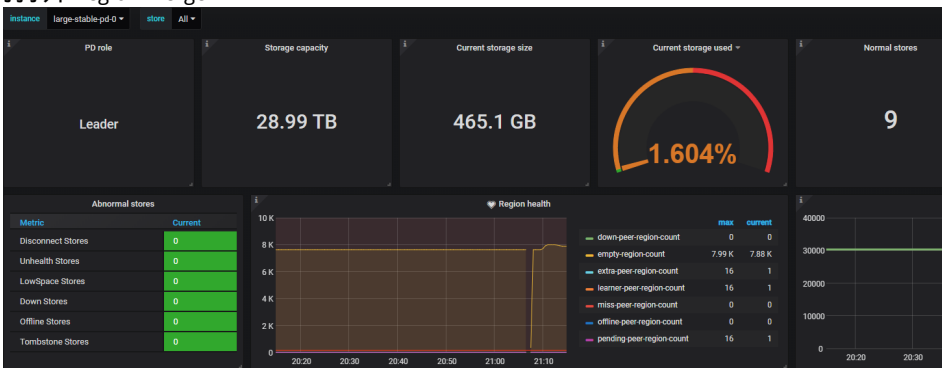
目前 Grafana Dashboard 整体分为 PD、 TiDB、 TiKV、 Node_exporter、 Overview 等。

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：

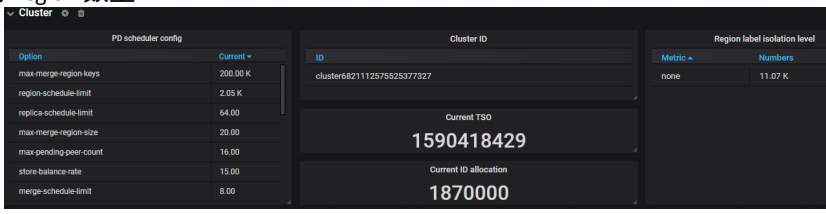
- PD role：当前 PD 的角色
- Storage capacity：TiDB 集群总可用数据库空间大小
- Current storage size：TiDB 集群目前已用数据库空间大小
- Current storage usage：TiDB 集群存储空间的使用率
- Normal stores：处于正常状态的节点数目
- Number of Regions：当前集群的 Region 总量

- 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



4.8.3.1 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



4.8.3.2 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

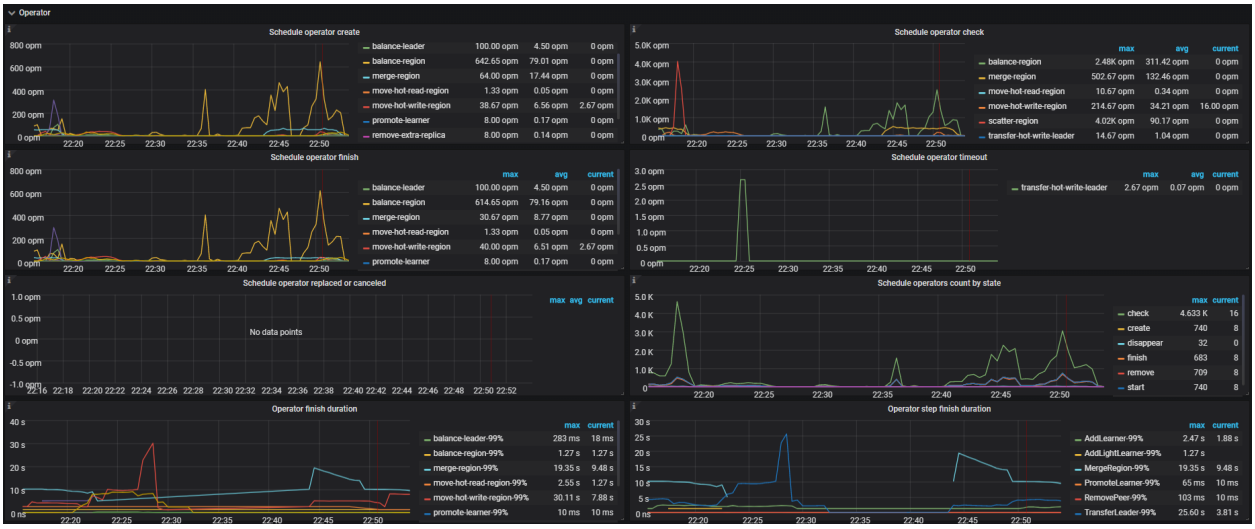


图 293: PD Dashboard - Operator metrics

4.8.3.3 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 294: PD Dashboard - Balance metrics

4.8.3.4 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 295: PD Dashboard - Hot write metrics

4.8.3.5 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 296: PD Dashboard - Hot read metrics

4.8.3.6 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

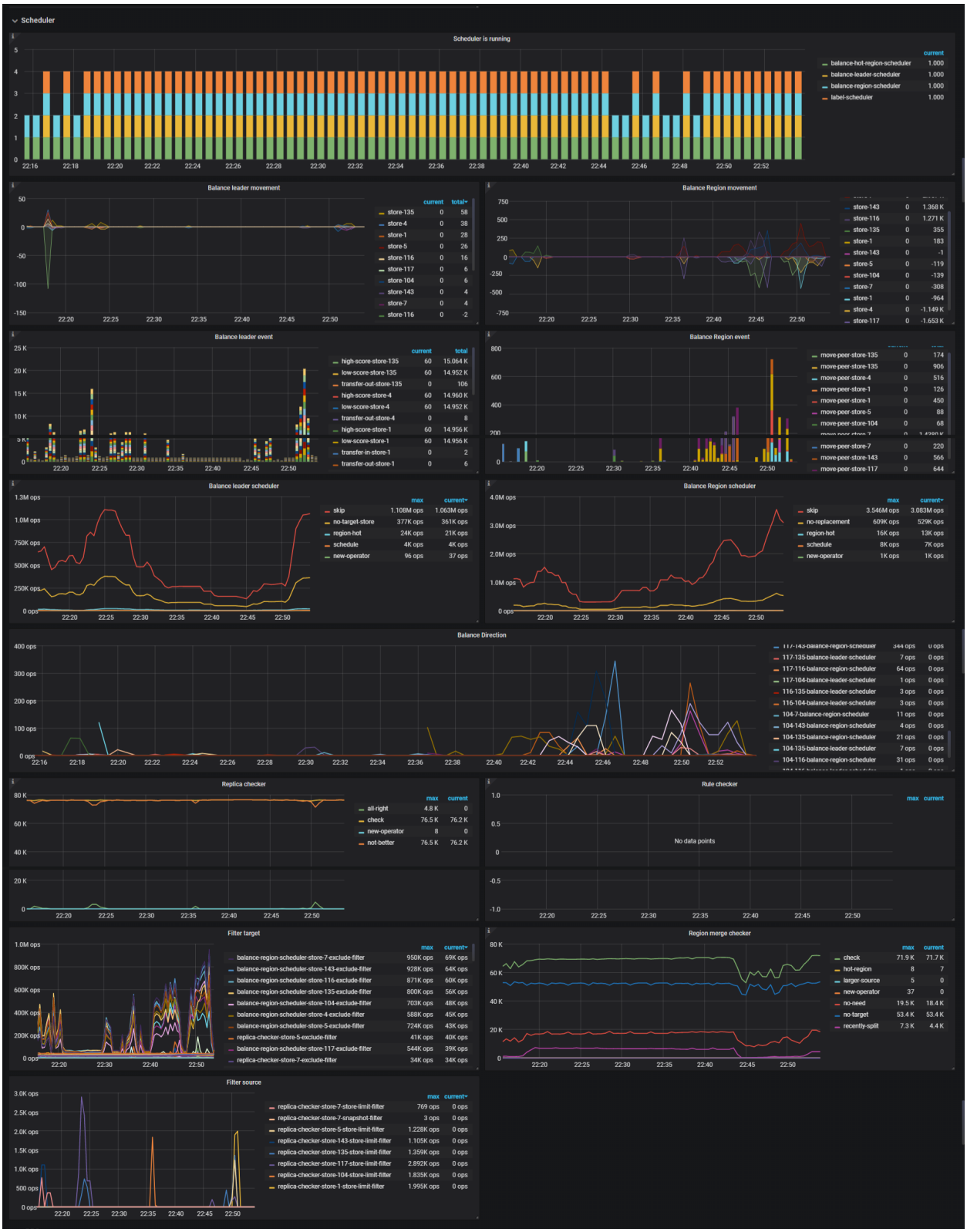


图 297: PD Dashboard - Scheduler metrics

4.8.3.7 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 298: PD Dashboard - gRPC metrics

4.8.3.8 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 299: PD Dashboard - etcd metrics

4.8.3.9 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

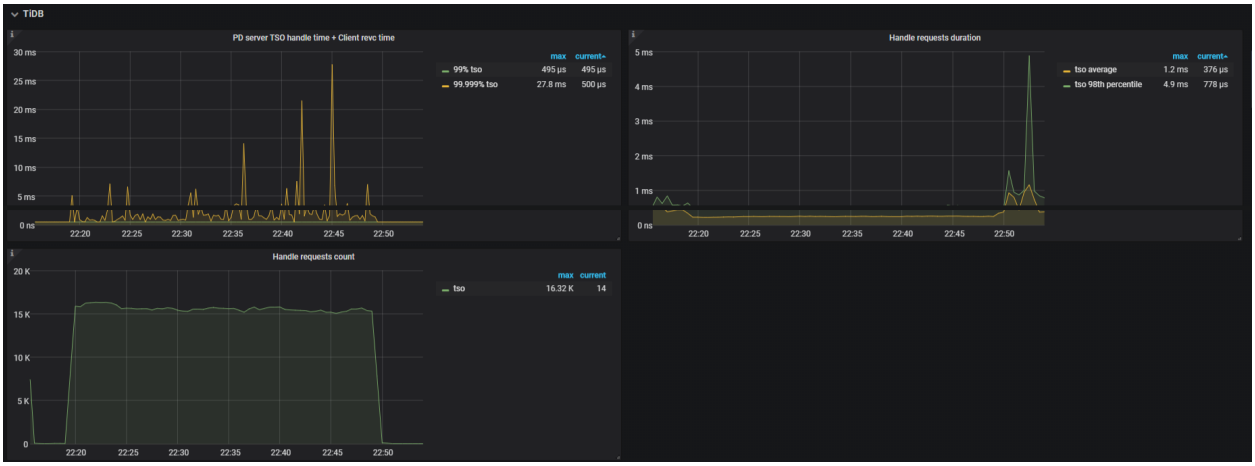


图 300: PD Dashboard - TiDB metrics

4.8.3.10 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

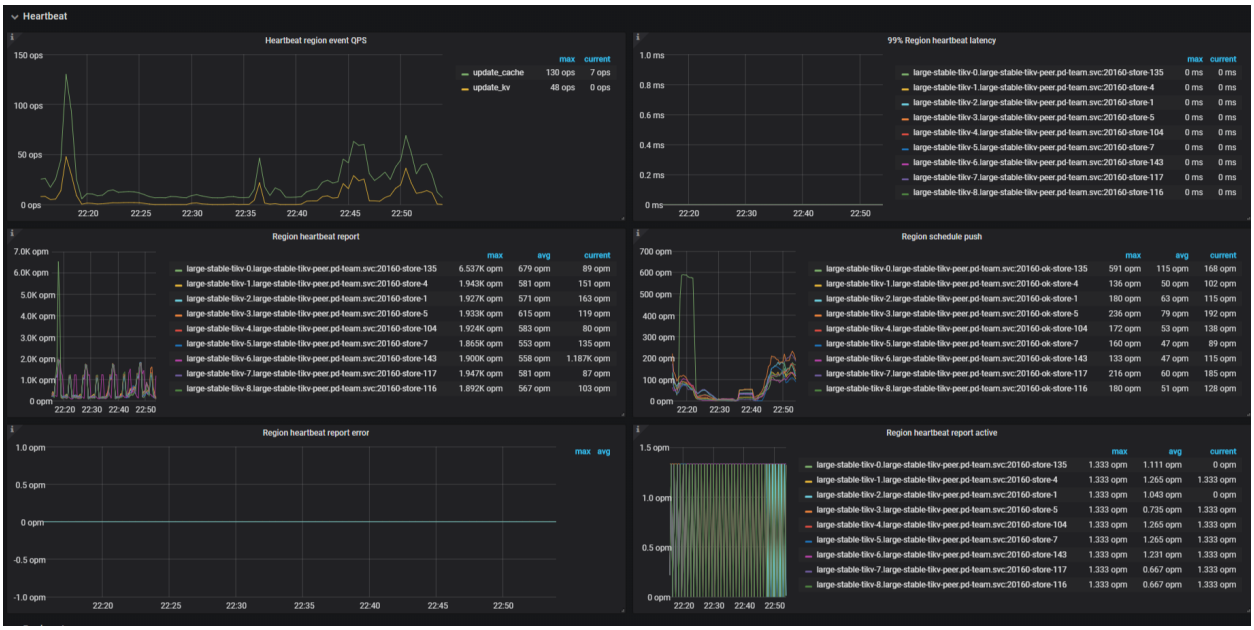


图 301: PD Dashboard - Heartbeat metrics

4.8.3.11 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 302: PD Dashboard - Region storage

4.8.4 TiKV 重要监控指标详解

使用 TiDB Ansible 部署 TiDB 集群时，一键部署监控系统 (Prometheus/Grafana)，监控架构请看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 面板上的 Metrics，可以了解 TiKV 当前的状态。

以下为 TiKV Dashboard 监控说明：

4.8.4.1 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 消息失败的个数
- leader：每个 TiKV 实例 leader 的个数
- Region：每个 TiKV 实例 Region 的个数

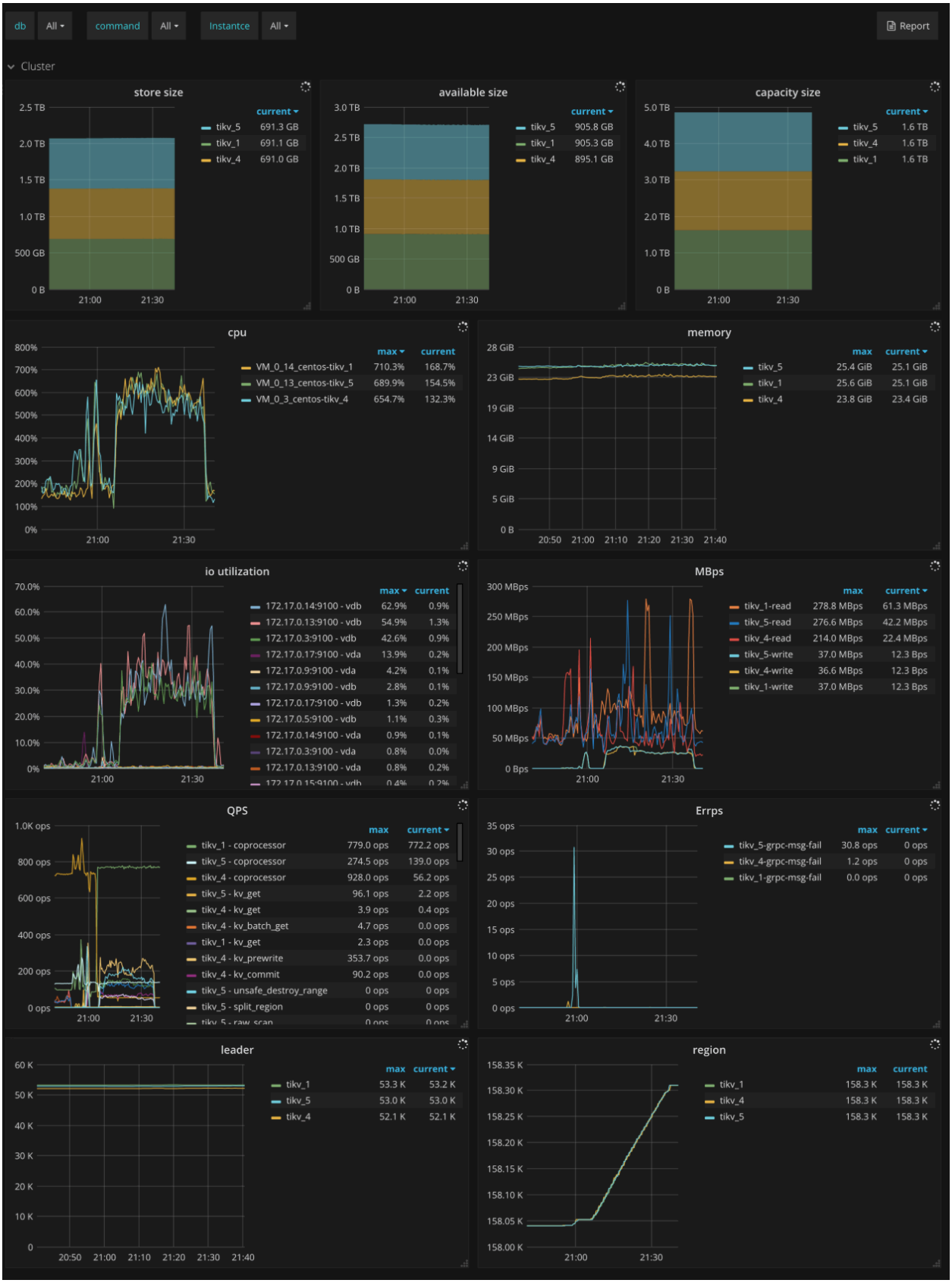


图 303: TiKV Dashboard - Cluster metrics

4.8.4.2 Errors

- Server is busy：各种会导致 server 繁忙的事件个数，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 304: TiKV Dashboard - Errors metrics

4.8.4.3 Server

- Leader：每个 TiKV 实例 leader 的个数
- Region：每个 TiKV 实例 Region 的个数
- CF size：每个 CF 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
- Server report failures：server 报错的消息个数，正常情况下应当为 0
- 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
- Active written leaders：每个 TiKV 实例上有效的 leader 个数
- Approximate Region size：每个 Region 近似的大小

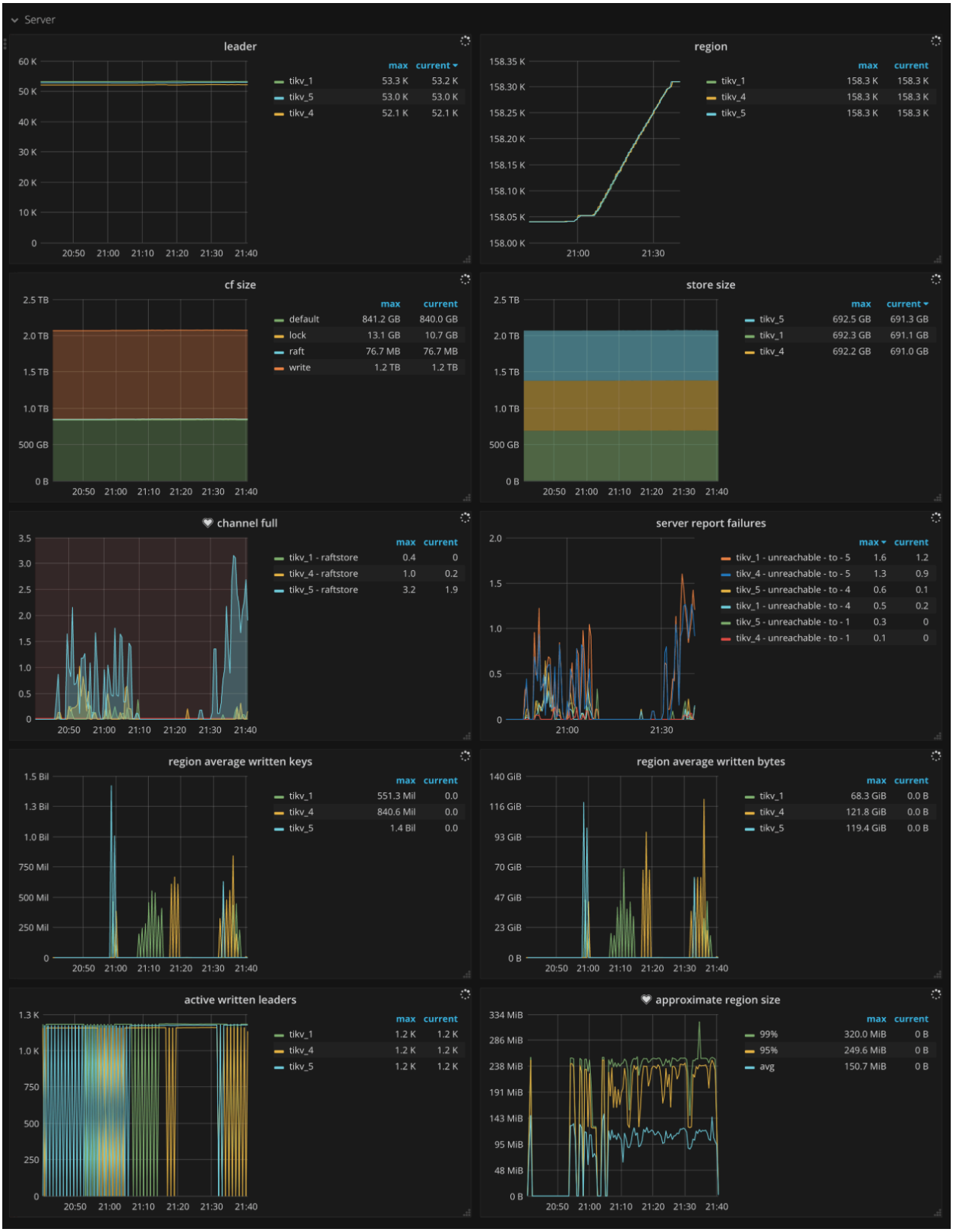


图 305: TiKV Dashboard - Server metrics

4.8.4.4 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

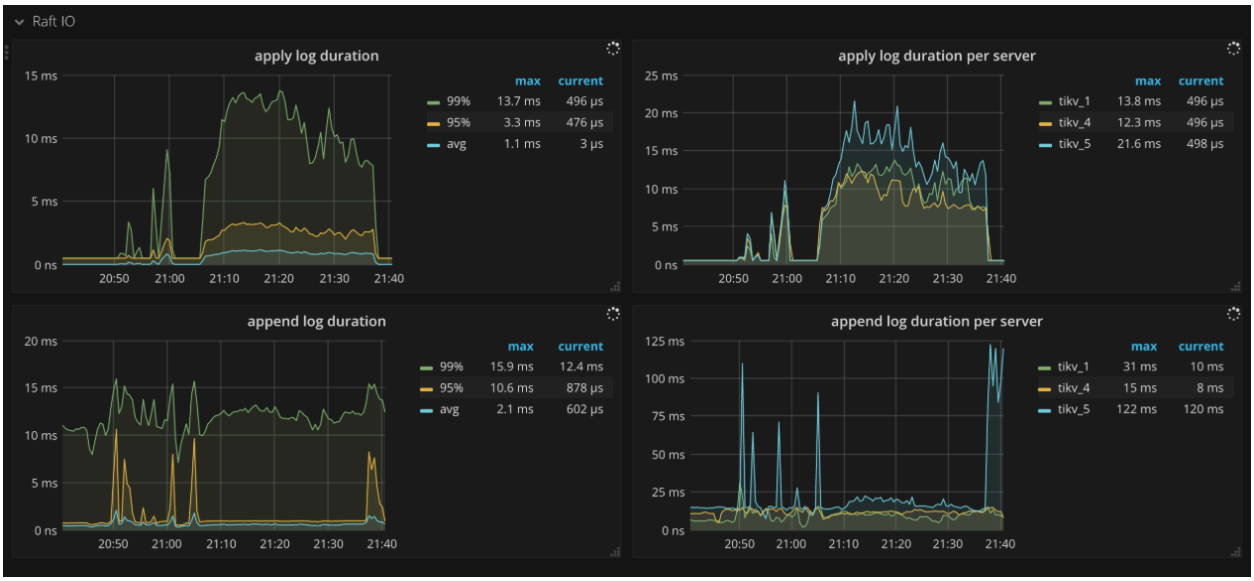


图 306: TiKV Dashboard - Raft IO metrics

4.8.4.5 Raft process

- Ready handled：Raft 中不同 ready 类型的个数
- 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
- Process tick duration per server：每个 TiKV 实例处理 tick 所花费的时间
- 0.99 Duration of Raft store events：99% 的 raftstore 事件所花费的时间

图 307: TiKV Dashboard - Raft process metrics

4.8.4.6 Raft message

- Sent messages per server：每个 TiKV 实例发送 Raft 消息的个数
- Flush messages per server：每个 TiKV 实例持久化 Raft 消息的个数
- Receive messages per server：每个 TiKV 实例接受 Raft 消息的个数
- Messages：发送不同类型的 Raft 消息的个数
- Vote：Raft 投票消息发送的个数
- Raft dropped messages：丢弃不同类型的 Raft 消息的个数

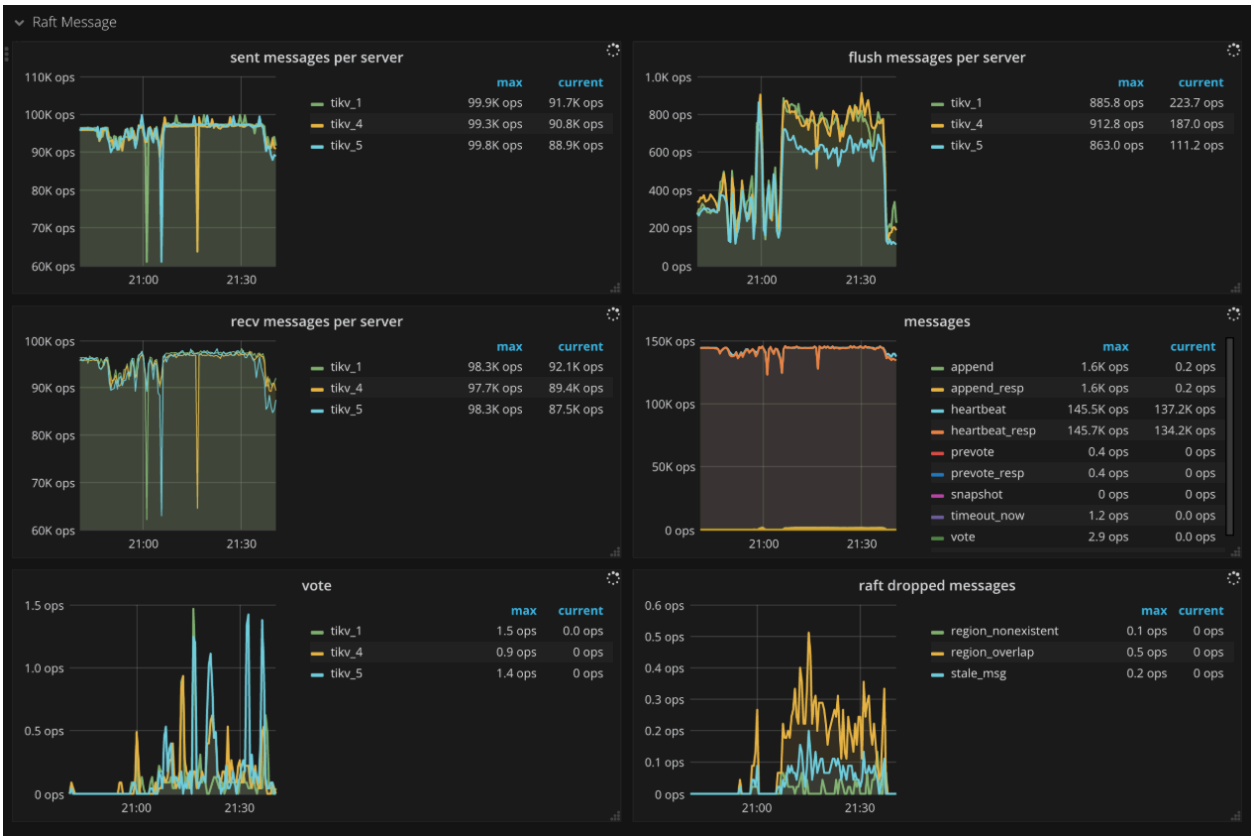


图 308: TiKV Dashboard - Raft message metrics

4.8.4.7 Raft propose

- Raft proposals per ready：在一个 mio tick 内，所有 Region proposal 的个数
- Raft read/write proposals：不同类型的 proposal 的个数
- Raft read proposals per server：每个 TiKV 实例发起读 proposal 的个数
- Raft write proposals per server：每个 TiKV 实例发起写 proposal 的个数
- Propose wait duration：每个 proposal 的等待时间
- Propose wait duration per server：每个 TiKV 实例上每个 proposal 的等待时间
- Raft log speed：peer propose 日志的速度

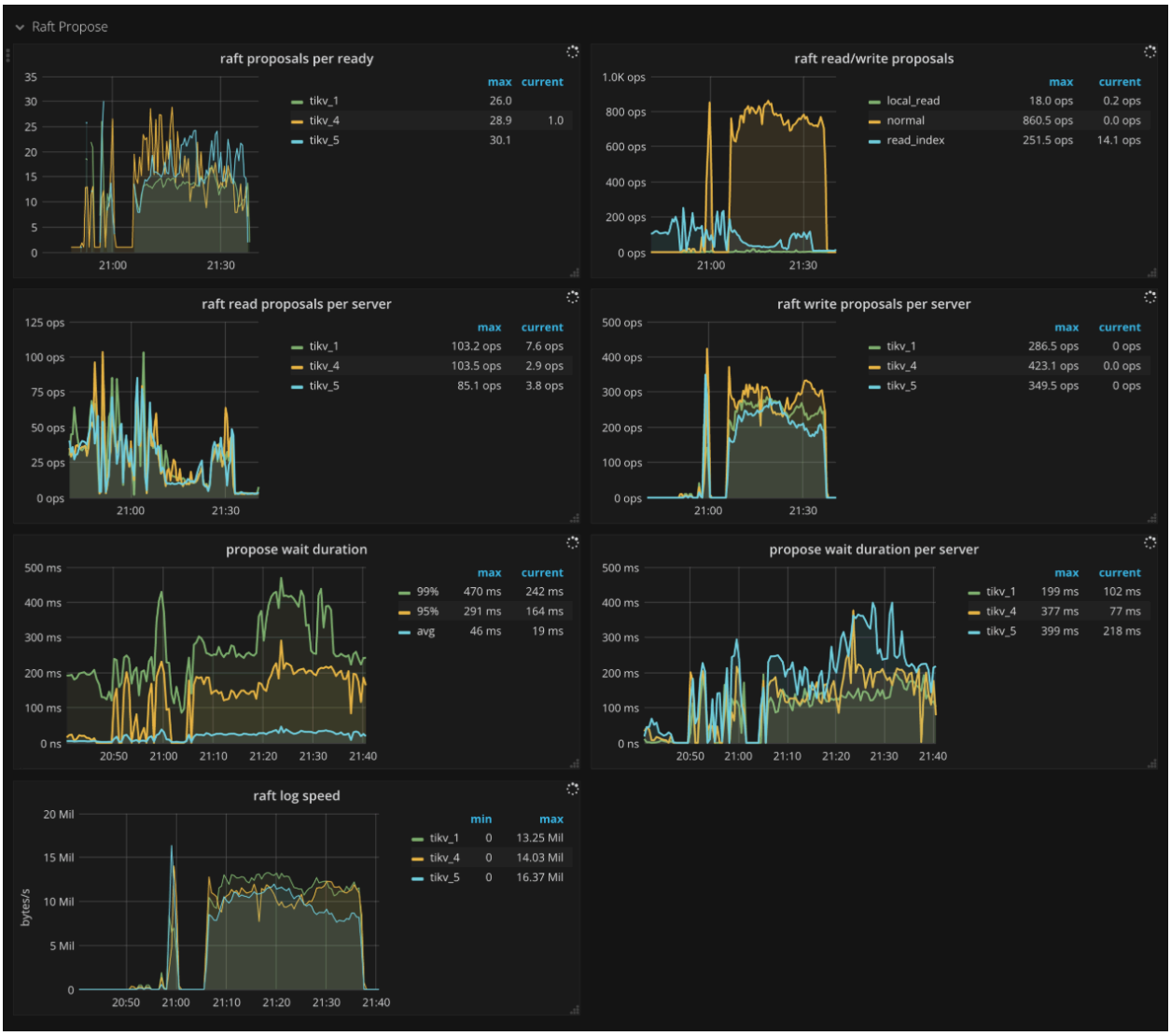


图 309: TiKV Dashboard - Raft propose metrics

4.8.4.8 Raft admin

- Admin proposals：admin proposal 的个数
- Admin apply：apply 命令的个数
- Check split：split check 命令的个数
- 99.99% Check split duration：99.99% 的情况下，split check 所需花费的时间

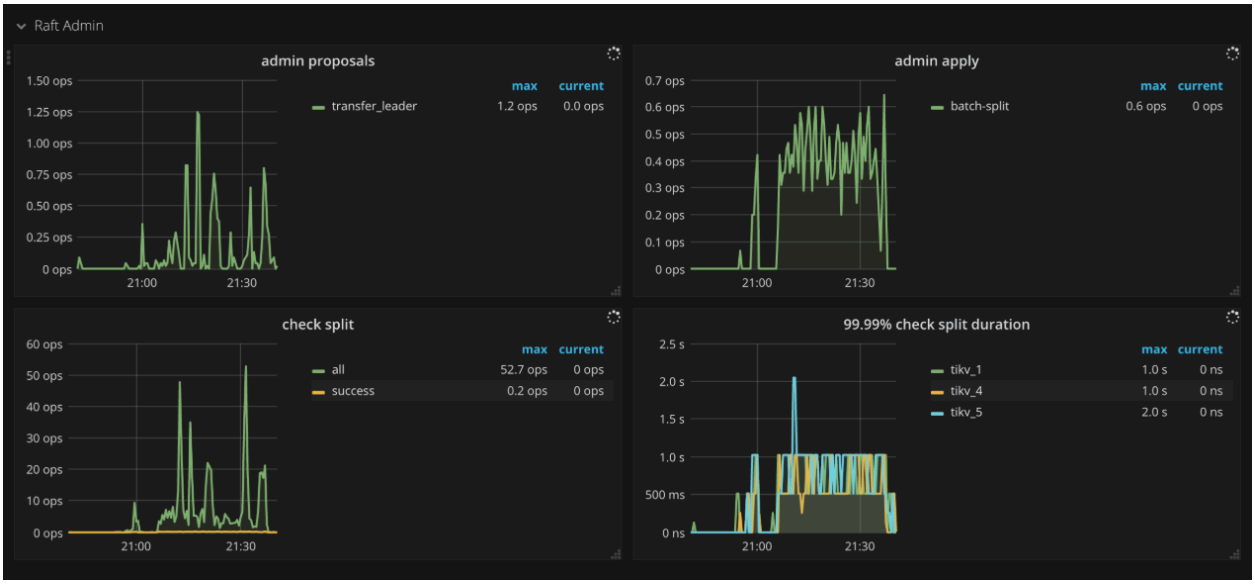


图 310: TiKV Dashboard - Raft admin metrics

4.8.4.9 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
- Local read requests duration：local read 请求的等待时间
- Local read requests batch size：local read 请求的批量大小

图 311: TiKV Dashboard - Local reader metrics

4.8.4.10 Storage

- Storage command total：收到不同命令的个数
- Storage async request error：异步请求出错的个数
- 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

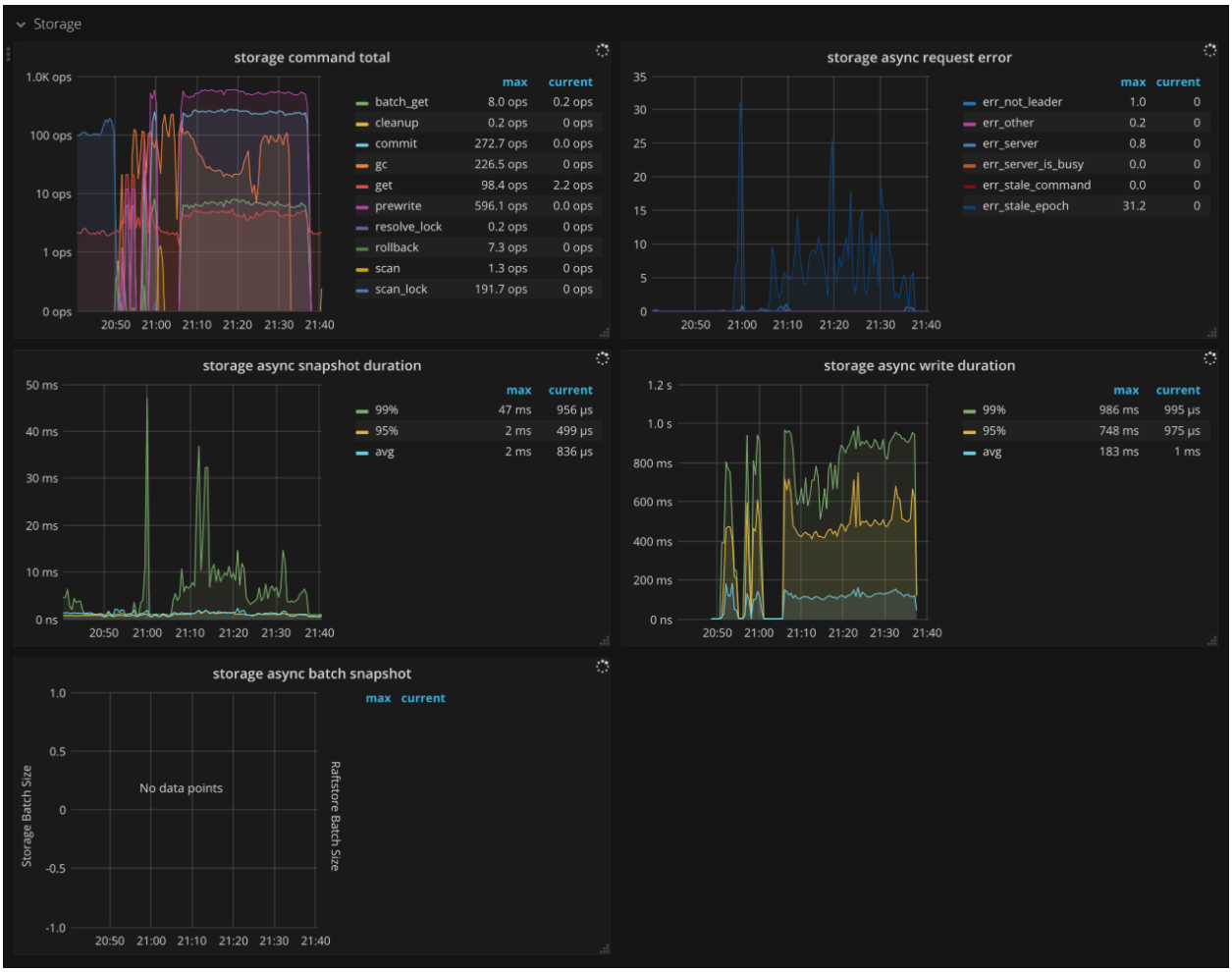


图 312: TiKV Dashboard - Storage metrics

4.8.4.11 Scheduler

- Scheduler stage total：每种命令不同阶段的个数，正常情况下，不会在短时间内出现大量的错误
- Scheduler priority commands：不同优先级命令的个数
- Scheduler pending commands：每个 TiKV 实例上 pending 命令的个数

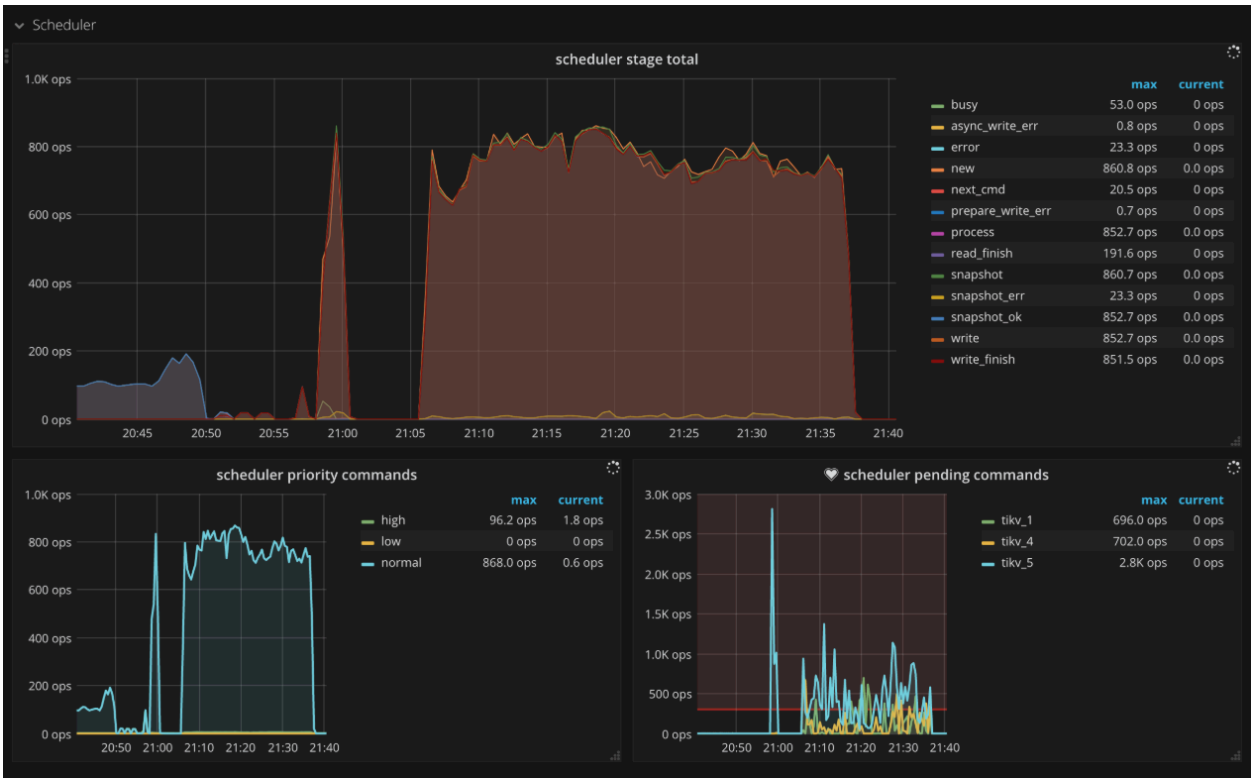


图 313: TiKV Dashboard - Scheduler metrics

4.8.4.12 Scheduler - batch_get

- Scheduler stage total：batch_get 中每个命令所处不同阶段的个数，正常情况下，不会在短时间内出现大量的错误
- Scheduler command duration：执行 batch_get 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：batch_get 命令读取 key 的个数
- Scheduler keys written：batch_get 命令写入 key 的个数
- Scheduler scan details：执行 batch_get 命令时，扫描每个 CF 中 key 的详细情况
- Scheduler scan details：执行 batch_get 命令时，扫描每个 lock CF 中 key 的详细情况
- Scheduler scan details [write]：执行 batch_get 命令时，扫描每个 write CF 中 key 的详细情况
- Scheduler scan details [default]：执行 batch_get 命令时，扫描每个 default CF 中 key 的详细情况

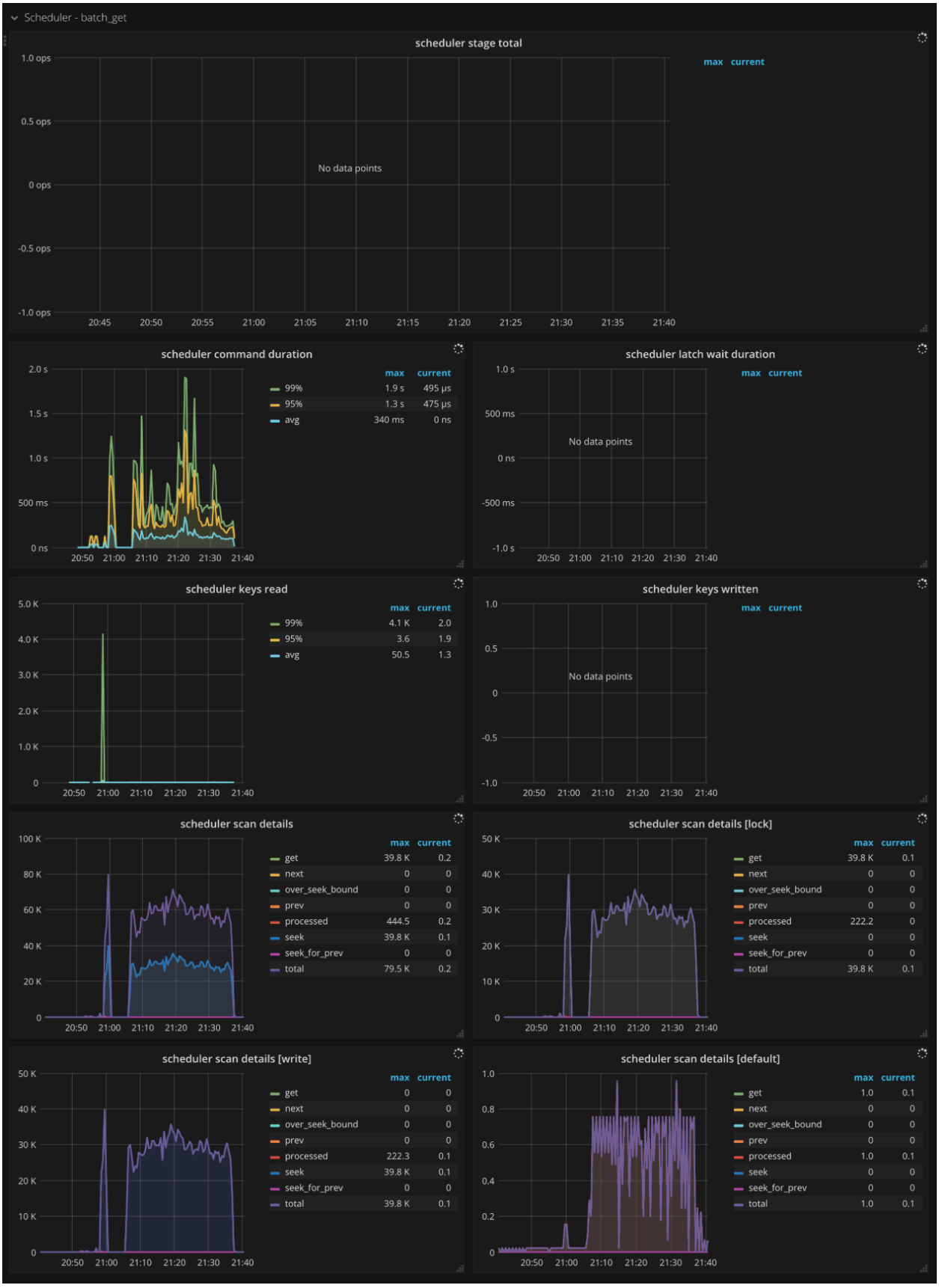


图 314: TiKV Dashboard - Scheduler - batch_get metrics

4.8.4.13 Scheduler - cleanup

- Scheduler stage total：cleanup 中每个命令所处不同阶段的个数，正常情况下，不会在短时间内出现大量的错误
- Scheduler command duration：执行 cleanup 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：cleanup 命令读取 key 的个数
- Scheduler keys written：cleanup 命令写入 key 的个数
- Scheduler scan details：执行 cleanup 命令时，扫描每个 CF 中 key 的详细情况
- Scheduler scan details [lock]：执行 cleanup 命令时，扫描每个 lock CF 中 key 的详细情况
- Scheduler scan details [write]：执行 cleanup 命令时，扫描每个 write CF 中 key 的详细情况
- Scheduler scan details [default]：执行 cleanup 命令时，扫描每个 default CF 中 key 的详细情况

图 315: TiKV Dashboard - Scheduler - cleanup metrics

4.8.4.14 Scheduler - commit

- Scheduler stage total：commit 中每个命令所处不同阶段的个数，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 316: TiKV Dashboard - Scheduler commit metrics

4.8.4.15 Scheduler - gc

- Scheduler stage total：gc 中每个命令所处不同阶段的个数，正常情况下，不会在短时间内出现大量的错误
- Scheduler command duration：执行 gc 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：gc 命令读取 key 的个数
- Scheduler keys written：gc 命令写入 key 的个数
- Scheduler scan details：执行 gc 命令时，扫描每个 CF 中 key 的详细情况

- Scheduler scan details [lock]：执行 gc 命令时，扫描每个 lock CF 中 key 的详细情况
- Scheduler scan details [write]：执行 gc 命令时，扫描每个 write CF 中 key 的详细情况
- Scheduler scan details [default]：执行 gc 命令时，扫描每个 default CF 中 key 的详细情况

4.8.4.16 Scheduler - get

- Scheduler stage total：get 中每个命令所处不同阶段的个数，正常情况下，不会在短时间内出现大量的错误
- Scheduler command duration：执行 get 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：get 命令读取 key 的个数
- Scheduler keys written：get 命令写入 key 的个数
- Scheduler scan details：执行 get 命令时，扫描每个 CF 中 key 的详细情况
- Scheduler scan details [lock]：执行 get 命令时，扫描每个 lock CF 中 key 的详细情况
- Scheduler scan details [write]：执行 get 命令时，扫描每个 write CF 中 key 的详细情况
- Scheduler scan details [default]：执行 get 命令时，扫描每个 default CF 中 key 的详细情况

4.8.4.17 Scheduler - key_mvcc

- Scheduler stage total：key_mvcc 中每个命令所处不同阶段的个数，正常情况下，不会在短时间内出现大量的错误
- Scheduler command duration：执行 key_mvcc 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：key_mvcc 命令读取 key 的个数
- Scheduler keys written：key_mvcc 命令写入 key 的个数
- Scheduler scan details：执行 key_mvcc 命令时，扫描每个 CF 中 key 的详细情况
- Scheduler scan details [lock]：执行 key_mvcc 命令时，扫描每个 lock CF 中 key 的详细情况
- Scheduler scan details [write]：执行 key_mvcc 命令时，扫描每个 write CF 中 key 的详细情况
- Scheduler scan details [default]：执行 key_mvcc 命令时，扫描每个 default CF 中 key 的详细情况

4.8.4.18 Scheduler - prewrite

- Scheduler stage total：prewrite 中每个命令所处不同阶段的个数，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

4.8.4.19 Scheduler - resolve_lock

- Scheduler stage total：resolve_lock 中每个命令所处不同阶段的个数，正常情况下，不会在短时间内出现大量的错误
- Scheduler command duration：执行 resolve_lock 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：resolve_lock 命令读取 key 的个数
- Scheduler keys written：resolve_lock 命令写入 key 的个数
- Scheduler scan details：执行 resolve_lock 命令时，扫描每个 CF 中 key 的详细情况
- Scheduler scan details [lock]：执行 resolve_lock 命令时，扫描每个 lock CF 中 key 的详细情况
- Scheduler scan details [write]：执行 resolve_lock 命令时，扫描每个 write CF 中 key 的详细情况
- Scheduler scan details [default]：执行 resolve_lock 命令时，扫描每个 default CF 中 key 的详细情况

4.8.4.20 Scheduler - scan

- Scheduler stage total：scan 中每个命令所处不同阶段的个数，正常情况下，不会在短时间内出现大量的错误
- Scheduler command duration：执行 scan 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：scan 命令读取 key 的个数
- Scheduler keys written：scan 命令写入 key 的个数
- Scheduler scan details：执行 scan 命令时，扫描每个 CF 中 key 的详细情况
- Scheduler scan details [lock]：执行 scan 命令时，扫描每个 lock CF 中 key 的详细情况
- Scheduler scan details [write]：执行 scan 命令时，扫描每个 write CF 中 key 的详细情况
- Scheduler scan details [default]：执行 scan 命令时，扫描每个 default CF 中 key 的详细情况

4.8.4.21 Scheduler - scan_lock

- Scheduler stage total：scan_lock 中每个命令所处不同阶段的个数，正常情况下，不会在短时间内出现大量的错误
- Scheduler command duration：执行 scan_lock 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：scan_lock 命令读取 key 的个数
- Scheduler keys written：scan_lock 命令写入 key 的个数
- Scheduler scan details：执行 scan_lock 命令时，扫描每个 CF 中 key 的详细情况
- Scheduler scan details [lock]：执行 scan_lock 命令时，扫描每个 lock CF 中 key 的详细情况
- Scheduler scan details [write]：执行 scan_lock 命令时，扫描每个 write CF 中 key 的详细情况
- Scheduler scan details [default]：执行 scan_lock 命令时，扫描每个 default CF 中 key 的详细情况

4.8.4.22 Scheduler - start_ts_mvcc

- Scheduler stage total：start_ts_mvcc 中每个命令所处不同阶段的个数，正常情况下，不会在短时间内出现大量的错误
- Scheduler command duration：执行 start_ts_mvcc 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：start_ts_mvcc 命令读取 key 的个数
- Scheduler keys written：start_ts_mvcc 命令写入 key 的个数
- Scheduler scan details：执行 start_ts_mvcc 命令时，扫描每个 CF 中 key 的详细情况
- Scheduler scan details [lock]：执行 start_ts_mvcc 命令时，扫描每个 lock CF 中 key 的详细情况
- Scheduler scan details [write]：执行 start_ts_mvcc 命令时，扫描每个 write CF 中 key 的详细情况
- Scheduler scan details [default]：执行 start_ts_mvcc 命令时，扫描每个 default CF 中 key 的详细情况

4.8.4.23 Scheduler - unsafe_destroy_range

- Scheduler stage total：unsafe_destroy_range 中每个命令所处不同阶段的个数，正常情况下，不会在短时间内出现大量的错误
- Scheduler command duration：执行 unsafe_destroy_range 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：unsafe_destroy_range 命令读取 key 的个数
- Scheduler keys written：unsafe_destroy_range 命令写入 key 的个数
- Scheduler scan details：执行 unsafe_destroy_range 命令时，扫描每个 CF 中 key 的详细情况
- Scheduler scan details [lock]：执行 unsafe_destroy_range 命令时，扫描每个 lock CF 中 key 的详细情况
- Scheduler scan details [write]：执行 unsafe_destroy_range 命令时，扫描每个 write CF 中 key 的详细情况
- Scheduler scan details [default]：执行 unsafe_destroy_range 命令时，扫描每个 default CF 中 key 的详细情况

4.8.4.24 Coprocessor

- Request duration：处理 coprocessor 读请求所花费的时间
- Wait duration：coprocessor 请求的等待时间，99.99% 的情况下，应该小于 10s
- Handle duration：处理 coprocessor 请求所花费的时间
- 95% Request duration by store：95% 的情况下，每个 TiKV 实例处理 coprocessor 读请求所花费的时间
- 95% Wait duration by store：95% 的情况下，每个 TiKV 实例上 coprocessor 请求的等待时间
- 95% Handle duration by store：95% 的情况下，每个 TiKV 实例处理 coprocessor 请求所花费的时间
- Request errors：下推的请求发生错误的个数，正常情况下，短时间内不应该有大量的错误
- DAG executors：DAG executor 的个数
- Scan keys：每个请求 scan key 的个数
- Scan details：scan 每个 CF 的详细情况
- Table Scan - Details by CF：table scan 针对每个 CF 的详细情况
- Index Scan - Details by CF：index scan 针对每个 CF 的详细情况
- Table Scan - Perf Statistics：执行 table sacn 的时候，根据 perf 统计的 RocksDB 内部 operation 的个数
- Index Scan - Perf Statistics：执行 index sacn 的时候，根据 perf 统计的 RocksDB 内部 operation 的个数

4.8.4.25 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 actions result：TiDB Region 层面的 GC 结果
- 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
- TiDB GC failure：TiDB GC job 失败的个数
- GC lifetime：TiDB 设置的 GC lifetime
- GC interval：TiDB 设置的 GC 间隔

4.8.4.26 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

4.8.4.27 Task

- Worker handled tasks：worker 处理的任务个数
- Worker pending tasks：当前 worker 中，pending 和 running 的任务个数，正常情况下，应该小于 1000
- FuturePool handled tasks：future pool 处理的任务个数
- FuturePool pending tasks：当前 future pool 中，pending 和 running 的任务个数

4.8.4.28 Thread CPU

- Raft store CPU：raftstore 线程的 CPU 使用率，通常应低于 80%
- Async apply CPU：async apply 线程的 CPU 使用率，通常应低于 90%
- Scheduler CPU：scheduler 线程的 CPU 使用率，通常应低于 80%
- Scheduler worker CPU：scheduler worker 线程的 CPU 使用率
- Storage ReadPool CPU：Readpool 线程的 CPU 使用率
- Coprocessor CPU：coprocessor 线程的 CPU 使用率
- Snapshot worker CPU：snapshot worker 线程的 CPU 使用率
- Split check CPU：split check 线程的 CPU 使用率
- RocksDB CPU：RocksDB 线程的 CPU 使用率
- gRPC poll CPU：gRPC 线程的 CPU 使用率，通常应低于 80%

4.8.4.29 RocksDB - kv

- Get operations：get 操作的个数
- Get duration：get 操作的耗时
- Seek operations：seek 操作的个数
- Seek duration：seek 操作的耗时
- Write operations：write 操作的个数

- Write duration：write 操作的耗时
- WAL sync operations：sync WAL 操作的个数
- WAL sync duration：sync WAL 操作的耗时
- Compaction operations：compaction 和 flush 操作的个数
- 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

4.8.4.30 RocksDB - raft

- Get operations：get 操作的个数
- Get duration：get 操作的耗时
- Seek operations：seek 操作的个数
- Seek duration：seek 操作的耗时
- Write operations：write 操作的个数
- Write duration：write 操作的耗时
- WAL sync operations：sync WAL 操作的个数
- WAL sync duration：sync WAL 操作的耗时
- Compaction operations：compaction 和 flush 操作的个数
- 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

4.8.4.31 gRPC

- gRPC message count：每种 gRPC 消息的个数
- gRPC message failed：失败的 gRPC 消息的个数
- 99% gRPC message duration：在 99% gRPC 消息的执行时间
- gRPC GC message count：gRPC GC 消息的个数
- 99% gRPC KV GC message duration：在 99% 情况下，gRPC GC 消息的执行时间

4.8.4.32 PD

- PD requests：TiKV 发送给 PD 的请求个数
- PD request duration (average)：TiKV 发送给 PD 的请求所需的平均时间
- PD heartbeats：发送给 PD 的心跳个数
- PD validate peers：通过 PD 验证 TiKV 的 peer 有效的个数

4.9 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

严重程度	说明
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

4.9.1 TiDB 报警规则

本节介绍了 TiDB 组件的报警项。

4.9.1.1 紧急级别报警项

4.9.1.1.1 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。

4.9.1.1.2 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。

4.9.1.1.3 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。

4.9.1.1.4 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` 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。

4.9.1.2 严重级别报警项

4.9.1.2.1 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。

4.9.1.3 警告级别报警项

4.9.1.3.1 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。

4.9.1.3.2 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。

4.9.1.3.3 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。

4.9.1.3.4 TiDB_tikvclient_backoff_total

- 报警规则：

```
increase(tidb_tikvclient_backoff_total[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。

4.9.1.3.5 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。

4.9.1.3.6 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。

4.9.2 PD 报警规则

本节介绍了 PD 组件的报警项。

4.9.2.1 紧急级别报警项

4.9.2.1.1 PD_cluster_offline_tikv_nums

- 报警规则：

  ```
  sum(pd_cluster_status{type="store_down_count"})> 0
  ```

- 规则描述：

  PD 长时间（默认配置是 30 分钟）没有收到 TiKV 心跳。

- 处理方法：

  - 检查 TiKV 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 无法恢复，可做下线处理。

4.9.2.2 严重级别报警项

4.9.2.2.1 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：

  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。

4.9.2.2.2 PD_miss_peer_region_count

- 报警规则：

```
sum(pd_regions_status{type="miss_peer_region_count"})> 100
```

- 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，下线 TiKV 节点也会导致少量 Region 缺副本（对于有 pending peer 的 Region 会走先减后加的流程）。

- 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。

4.9.2.3 警告级别报警项

4.9.2.3.1 PD_cluster_lost_connect_tikv_nums

- 报警规则：

```
sum(pd_cluster_status{type="store_disconnected_count"})> 0
```

- 规则描述：

  PD 在 20 秒之内未收到 TiKV 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。
  - 检查 TiKV 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 无法恢复，可做下线处理。
  - 如果确定 TiKV 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 被判定为无法恢复并开始搬移数据。

4.9.2.3.2 PD_cluster_low_space

- 报警规则：

```
sum(pd_cluster_status{type="store_low_space_count"})> 0
```

- 规则描述：

  表示 TiKV 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。

4.9.2.3.3 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。

4.9.2.3.4 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。

4.9.2.3.5 PD_down_peer_region_nums

- 报警规则：

```
sum(pd_regions_status{type="down_peer_region_count"})> 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。

4.9.2.3.6 PD_pending_peer_region_count

- 报警规则：

```
sum(pd_regions_status{type="pending_peer_region_count"})> 100
```

- 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。

4.9.2.3.7 PD_leader_change

- 报警规则：

```
count(changes(pd_server_tso{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。

4.9.2.3.8 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。

4.9.2.3.9 PD_system_time_slow

- 报警规则：

```
changes(pd_server_tso{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。

4.9.2.3.10 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。

4.9.3 TiKV 报警规则

本节介绍了 TiKV 组件的报警项。

4.9.3.1 紧急级别报警项

4.9.3.1.1 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。

4.9.3.1.2 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。

4.9.3.2 严重级别报警项

4.9.3.2.1 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。

4.9.3.2.2 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”`。

4.9.3.2.3 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。

4.9.3.2.4 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。

4.9.3.2.5 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。

4.9.3.2.6 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。

4.9.3.2.7 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。

4.9.3.2.8 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。

4.9.3.2.9 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。

4.9.3.2.10 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。

4.9.3.2.11 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。

4.9.3.2.12 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。

4.9.3.2.13 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。

4.9.3.3 警告级别报警项

4.9.3.3.1 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。

4.9.3.3.2 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。

4.9.3.3.3 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”。

4.9.3.3.4 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。

4.9.3.3.5 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。

4.9.3.3.6 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。

4.9.3.3.7 TiKV_coprocessor_request_error

- 报警规则：

  ```
  increase(tikv_coprocessor_request_error{reason!="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。

4.9.3.3.8 TiKV_coprocessor_request_lock_error

- 报警规则：

  ```
  increase(tikv_coprocessor_request_error{reason="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。

4.9.3.3.9 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。

4.9.3.3.10 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%。

4.9.3.3.11 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。

4.9.3.3.12 TiKV_low_space_and_add_region

- 报警规则：

```
count((sum(tikv_store_size_bytes{type="available"})by (instance)/ sum(tikv_store_size_bytes{
↪ type="capacity"})by (instance)< 0.2)and (sum(tikv_raftstore_snapshot_traffic_total{type="
↪ applying"})by (instance)> 0))> 0
```

4.9.3.3.13 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。

4.9.4 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。

4.9.5 Node_exporter 主机报警规则

本节介绍了 Node_exporter 主机的报警项。

4.9.5.1 紧急级别报警项

4.9.5.1.1 NODE_disk_used_more_than_80%

- 报警规则：

```
node_filesystem_avail{fstype=~"(ext.|xfs)", mountpoint!~"/boot"} / node_filesystem_size{
↪ 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。

4.9.5.1.2 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。

4.9.5.1.3 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。

4.9.5.2 严重级别报警项

4.9.5.2.1 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。

4.9.5.3 警告级别报警项

4.9.5.3.1 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。

4.9.5.3.2 NODE_cpu_used_more_than_80%

- 报警规则：

```
avg(irate(node_cpu{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。

4.9.5.3.3 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。

4.9.5.3.4 NODE_disk_read_latency_more_than_32ms

- 报警规则：

  ```
  ((rate(node_disk_read_time_ms{device=~".+"}[5m])/ rate(node_disk_reads_completed{device
  ↪ =~".+"}[5m]))or (irate(node_disk_read_time_ms{device=~".+"}[5m])/ irate(node_disk_reads_completed
  ↪ {device=~".+"}[5m])))> 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。

4.9.5.3.5 NODE_disk_write_latency_more_than_16ms

- 报警规则：

  ```
  ((rate(node_disk_write_time_ms{device=~".+"}[5m])/ rate(node_disk_writes_completed{device
  ↪ =~".+"}[5m]))or (irate(node_disk_write_time_ms{device=~".+"}[5m])/ irate(node_disk_writes_completed
  ↪ {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。

4.9.6 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。

4.9.6.1 紧急级别报警项

4.9.6.1.1 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。

4.9.6.1.2 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。

4.9.6.1.3 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。

4.9.6.1.4 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。

4.9.6.1.5 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。

4.9.6.1.6 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。

4.9.6.1.7 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。

4.9.6.1.8 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。

4.9.6.1.9 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。

4.9.6.1.10 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。

4.9.6.1.11 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。

4.9.6.2 警告级别报警项

4.9.6.2.1 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。

4.10 最佳实践

4.10.1 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。

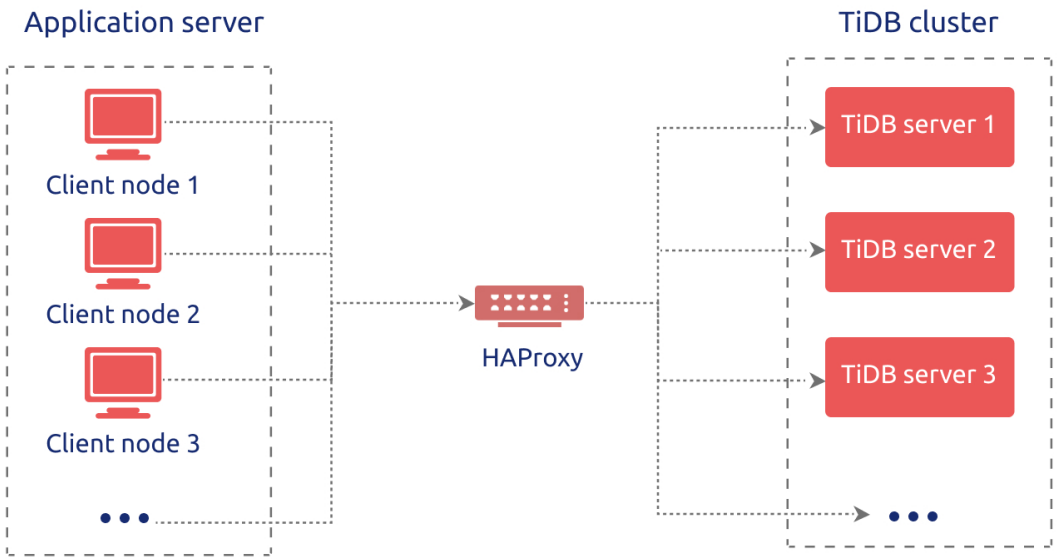


图 317: HAProxy 在 TiDB 中的最佳实践

4.10.1.1 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。

4.10.1.2 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。

4.10.1.3 准备环境

在部署 HAProxy 之前，需准备好以下环境。

4.10.1.3.1 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

4.10.1.3.2 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
```

4.10.1.4 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。

4.10.1.4.1 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。
-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。

参数	说明
-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)。
-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。

参数	说明
`-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。
`-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。

参数	说明
-S <bind >[,< bind_option >...]	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。

4.10.1.4.2 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
```

4.10.1.4.3 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
```

4.10.1.4.4 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
```

4.10.2 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。

4.10.2.1 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。

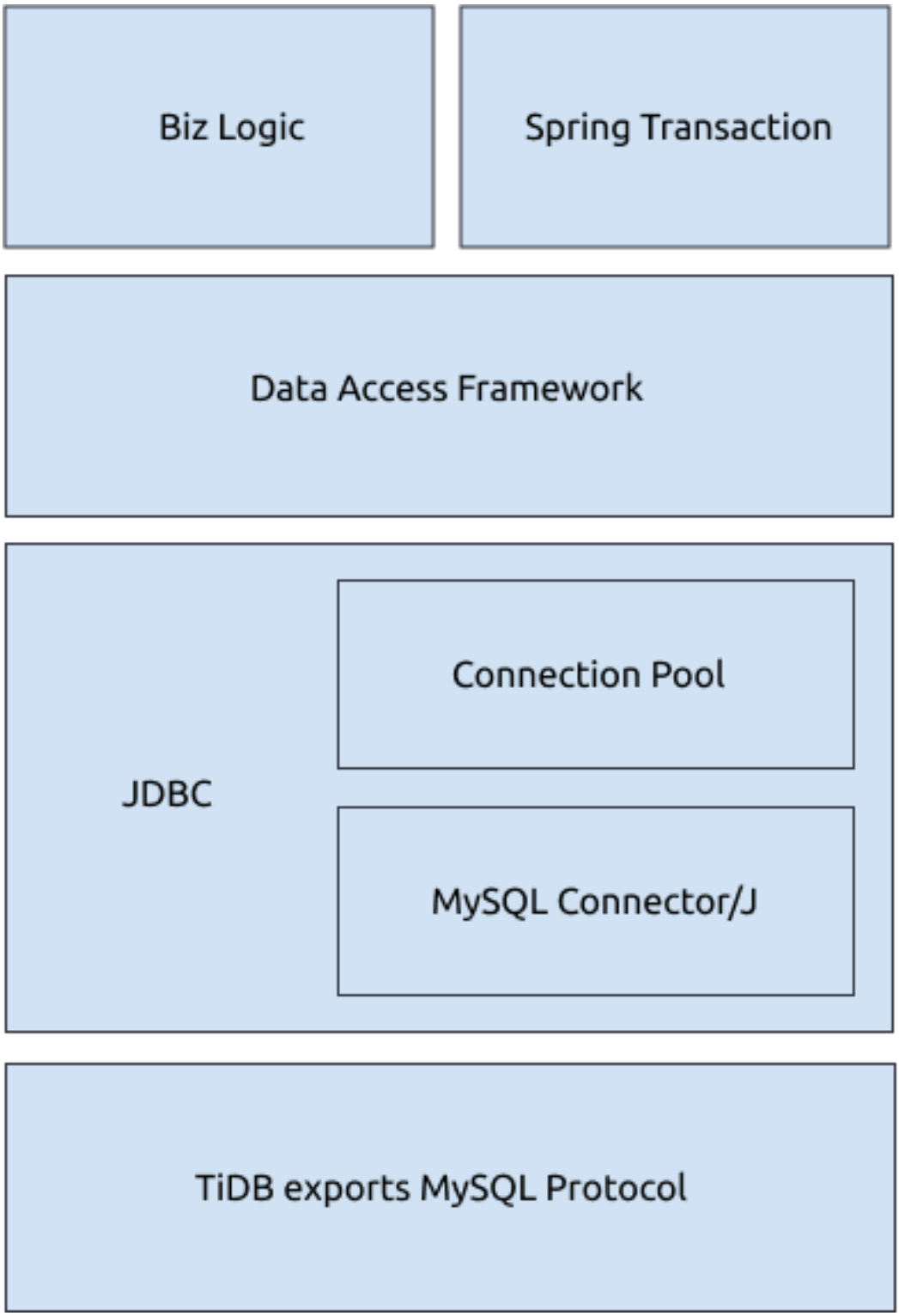


图 318: Java Component

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。

4.10.2.2 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。

4.10.2.2.1 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。

4.10.2.2.2 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。

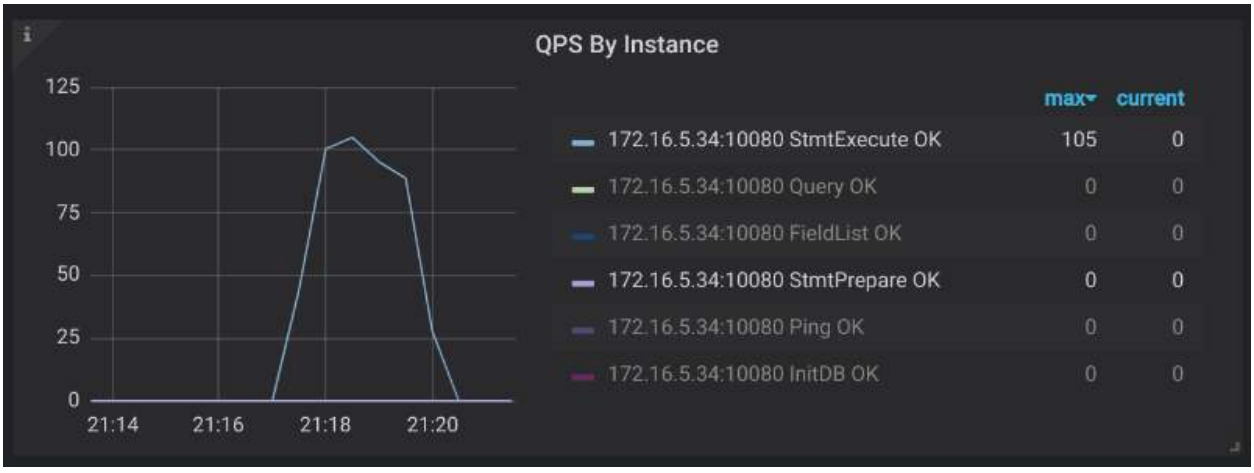


图 319: QPS By Instance

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。

4.10.2.3 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。

4.10.2.3.1 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。

4.10.2.3.2 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。

4.10.2.4 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。

4.10.2.4.1 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();
```

4.10.2.4.2 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。

4.10.2.5 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。

4.10.2.6 其他

4.10.2.6.1 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。

4.10.2.7 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。

4.10.3 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。

4.10.3.1 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。

4.10.3.2 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。

4.10.3.3 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。

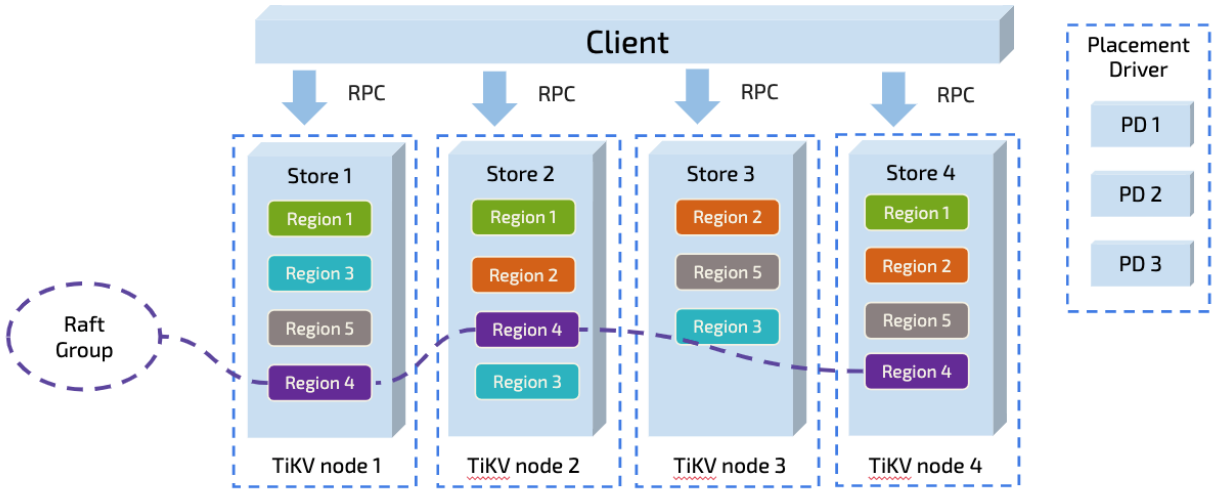


图 320: TiDB 数据概览

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。

4.10.3.4 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。

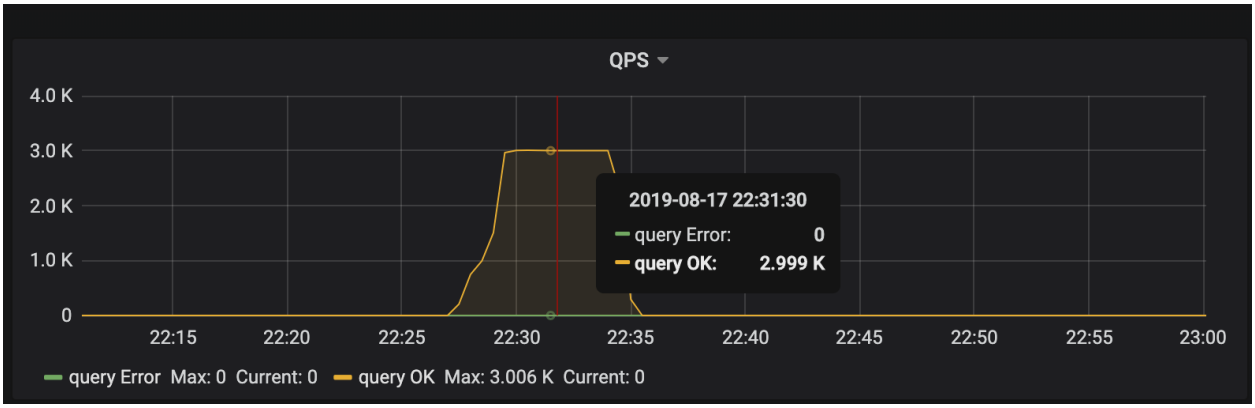


图 321: QPS1

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）：

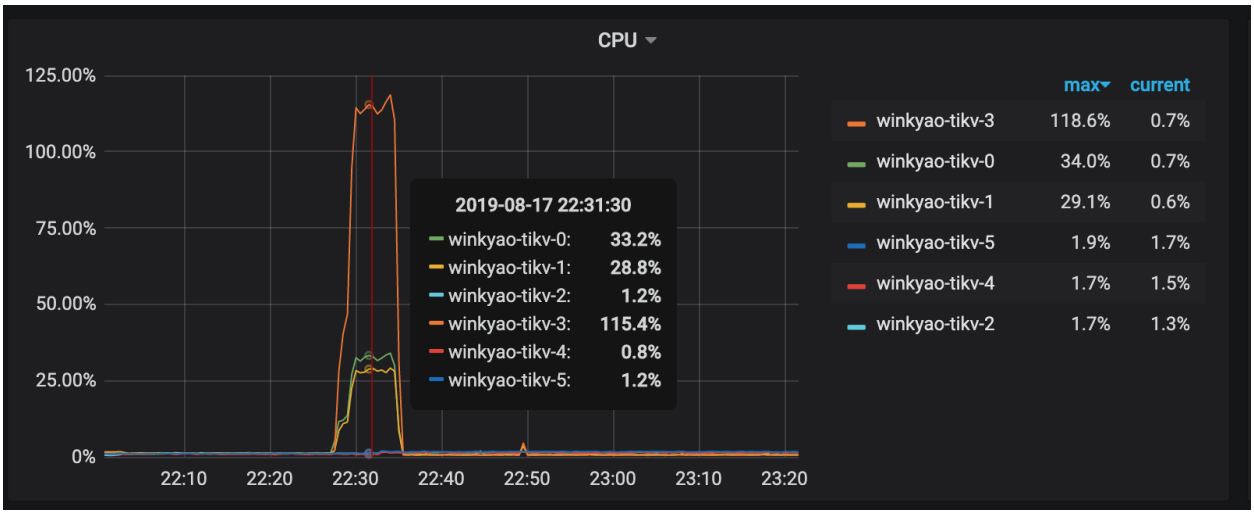


图 322: QPS2

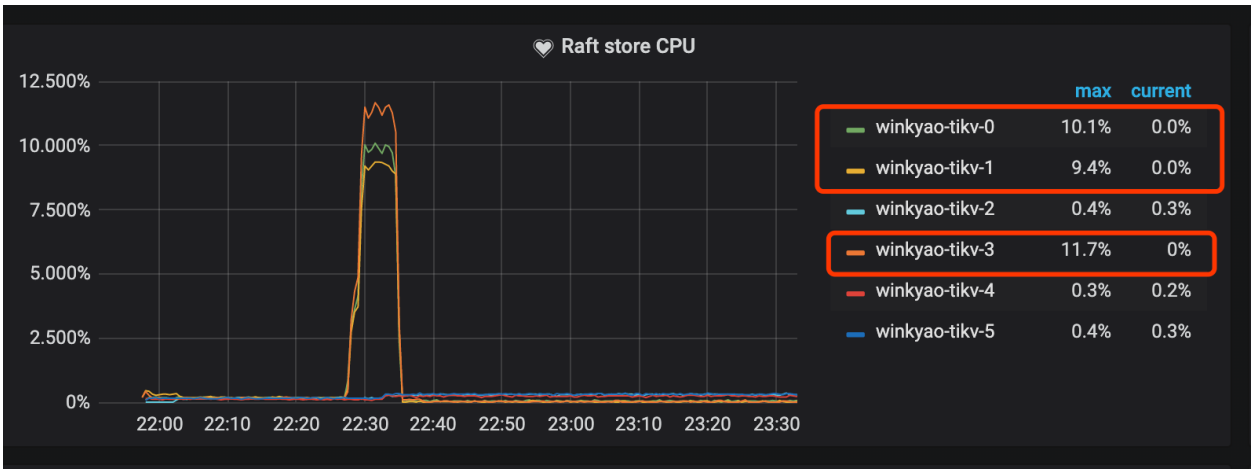


图 323: QPS3

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：

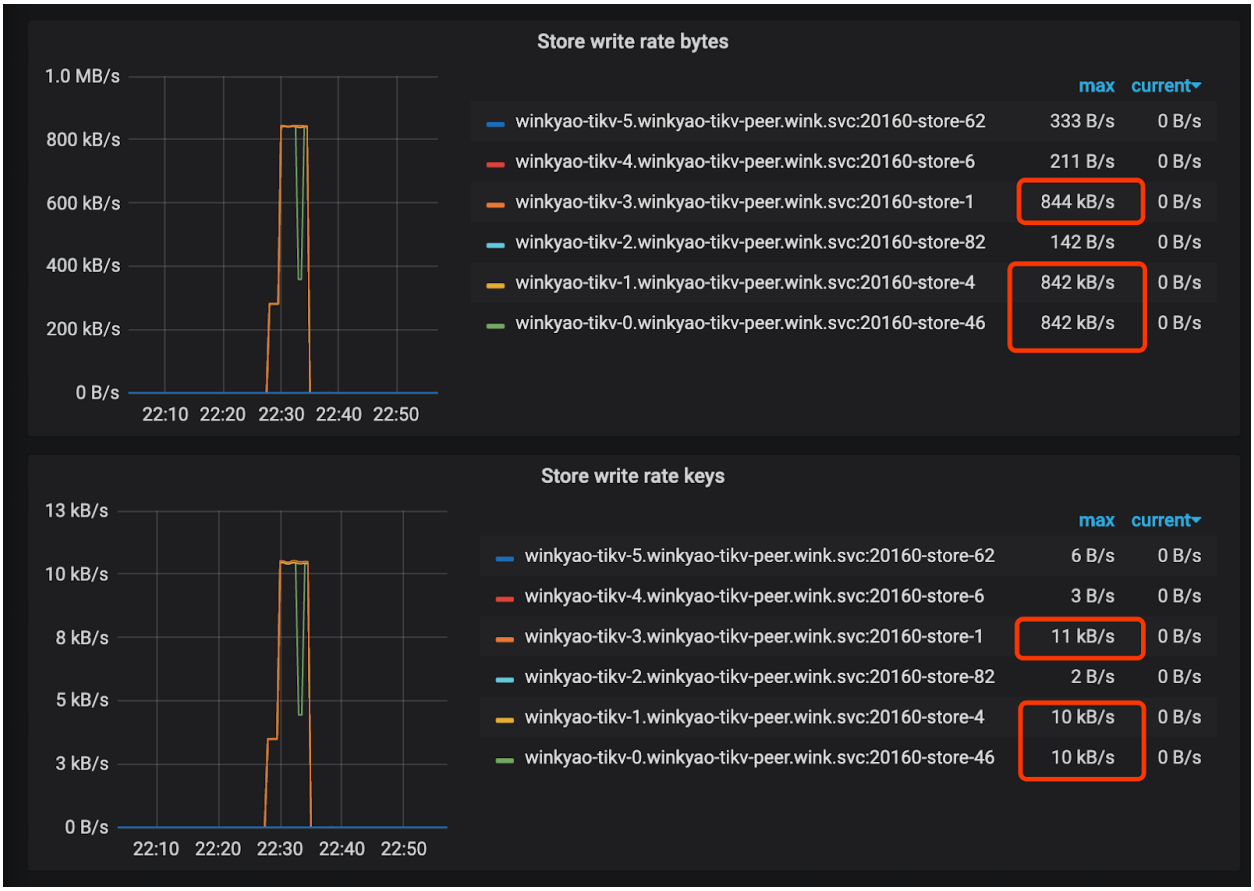


图 324: QPS4

4.10.3.5 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。

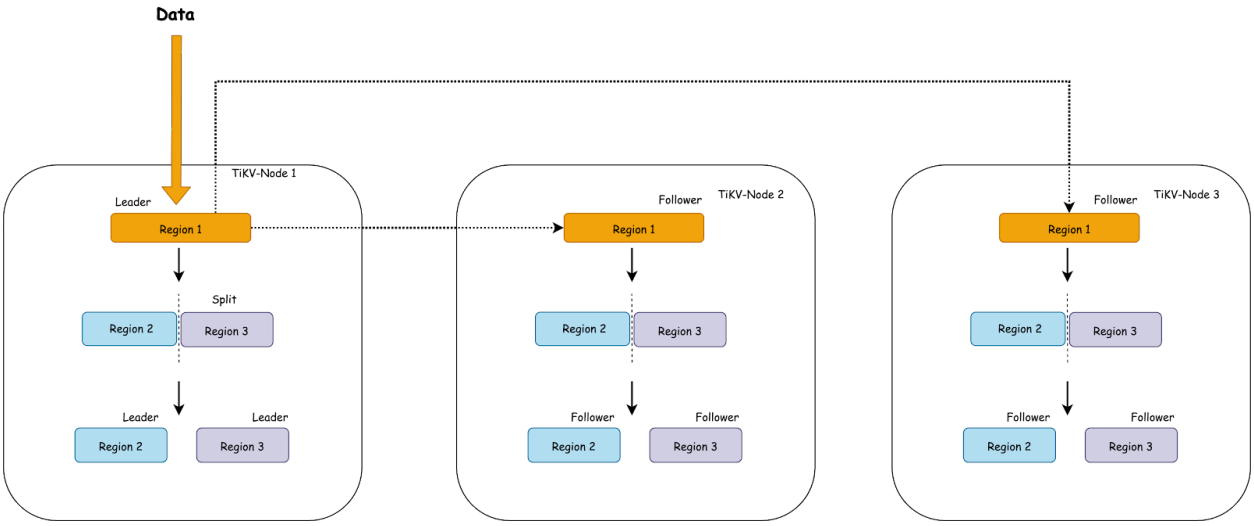


图 325: TiKV Region 分裂流程

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：

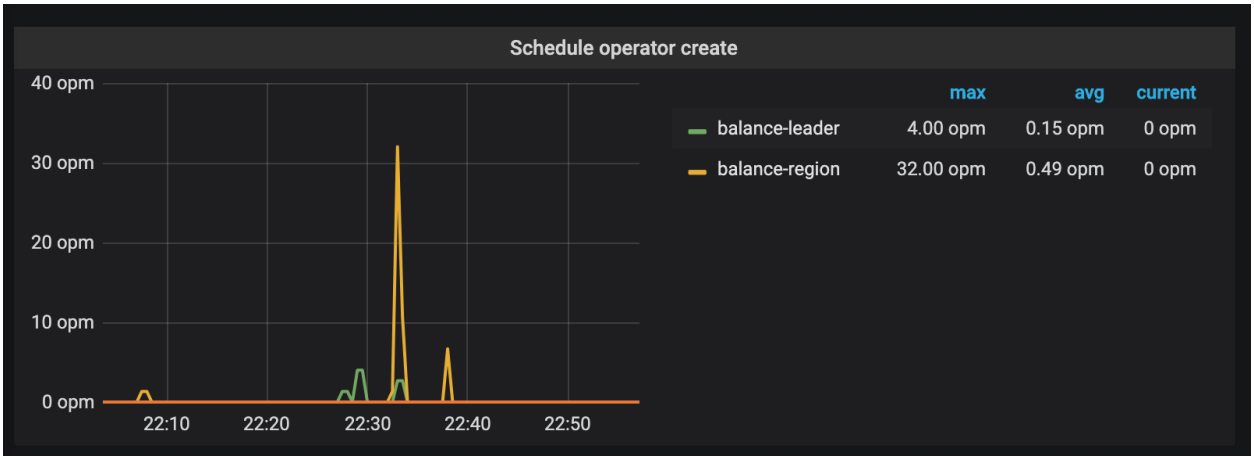


图 326: QPS5

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。

4.10.3.6 热点问题的规避方法

为了达到场景理论中的最佳性能，可以跳过这个预热阶段，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：

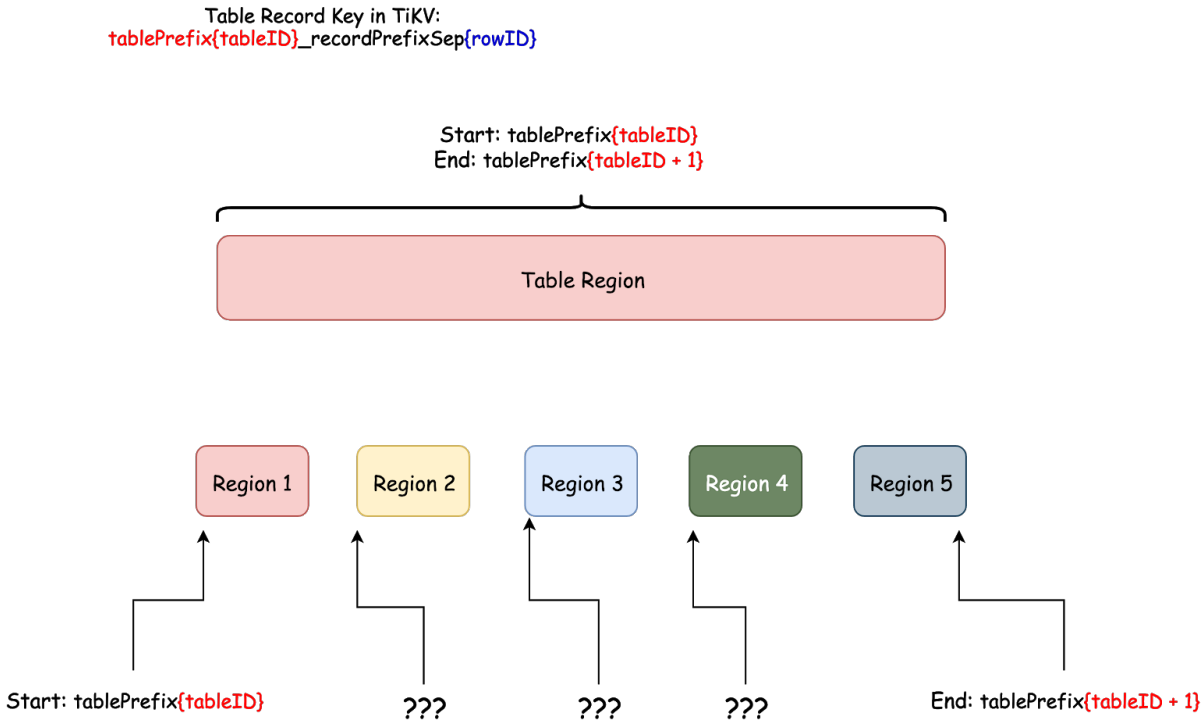


图 327: Table Region Range

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：

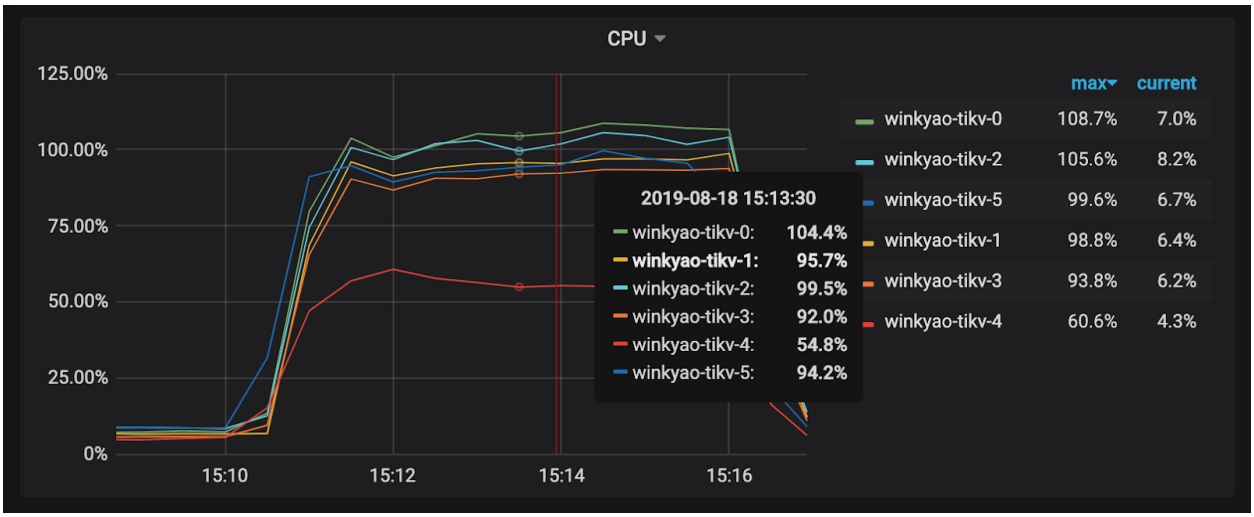


图 328: QPS6

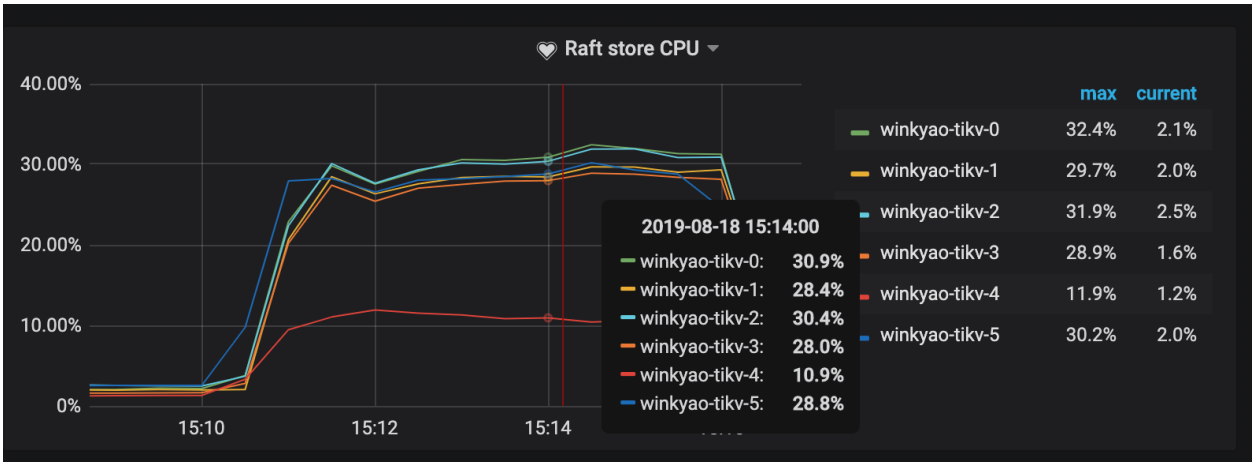


图 329: QPS7

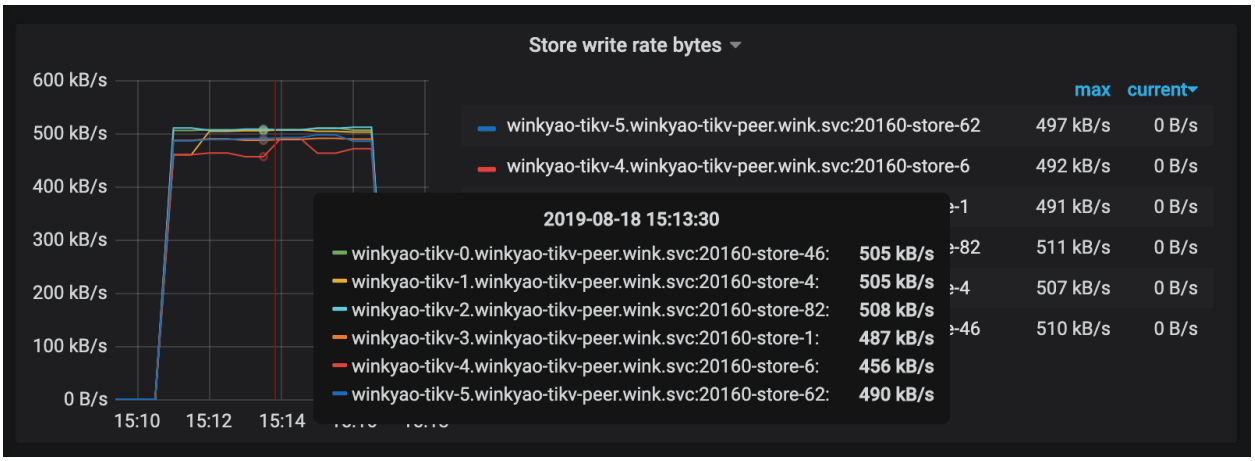


图 330: QPS8

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。

4.10.3.6.1 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。

4.10.3.7 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
```

4.10.4 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。

4.10.4.1 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。

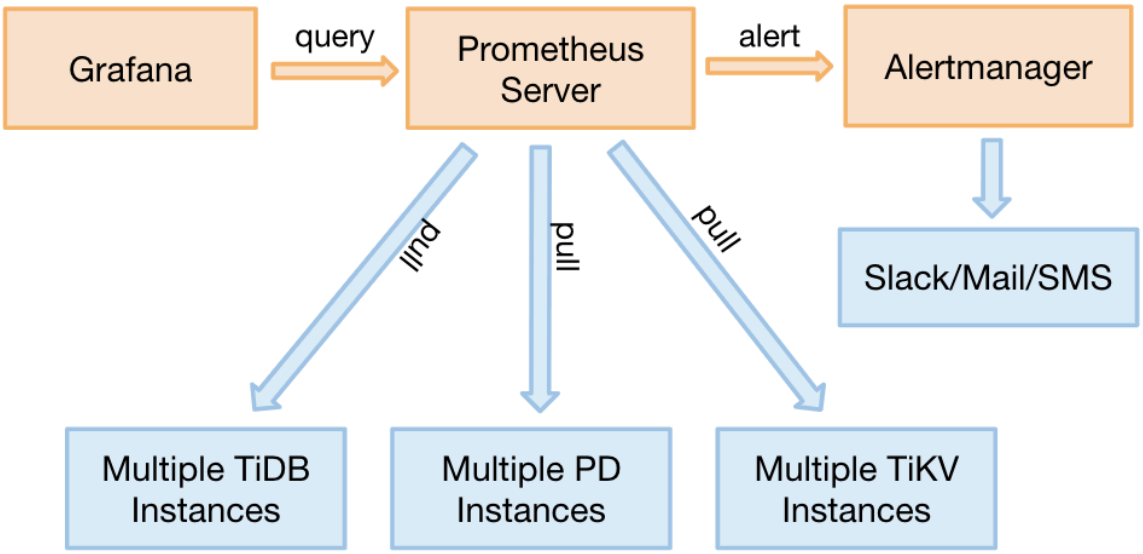


图 331: TiDB 监控整体架构

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。

4.10.4.2 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：

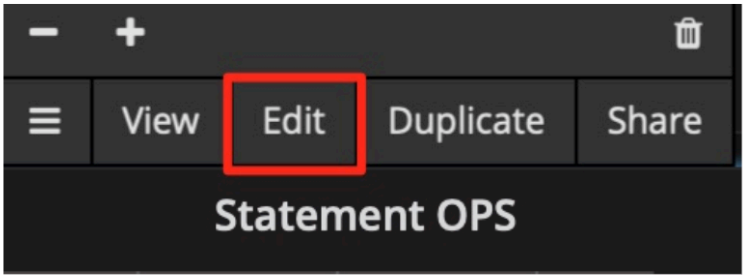


图 332: Metrics 面板的编辑入口

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 333: 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。

4.10.4.3 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。

4.10.4.3.1 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）：

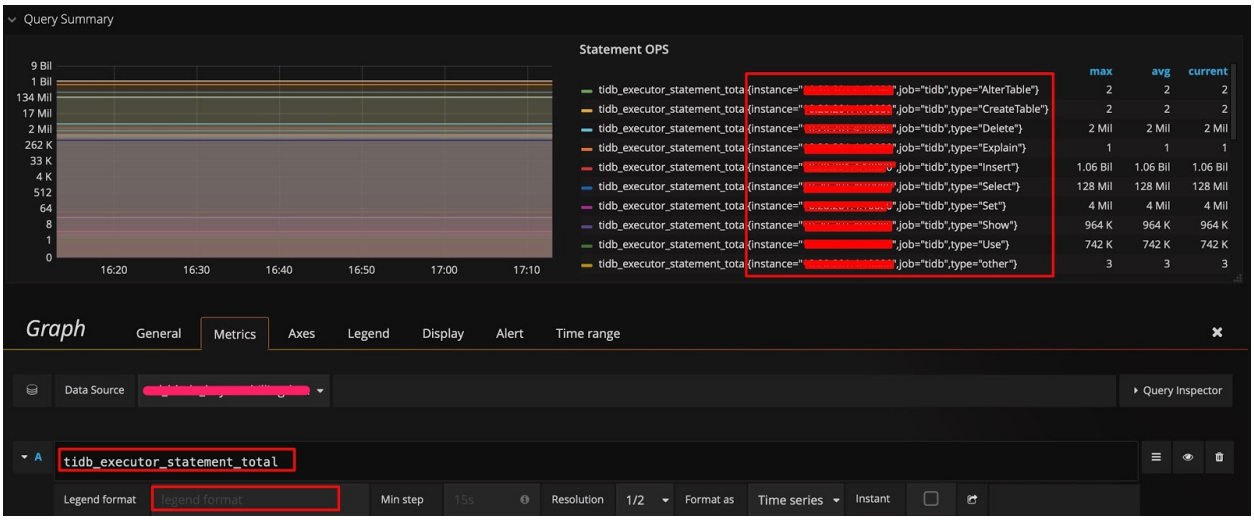


图 334: 编辑表达式并查看所有维度

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：

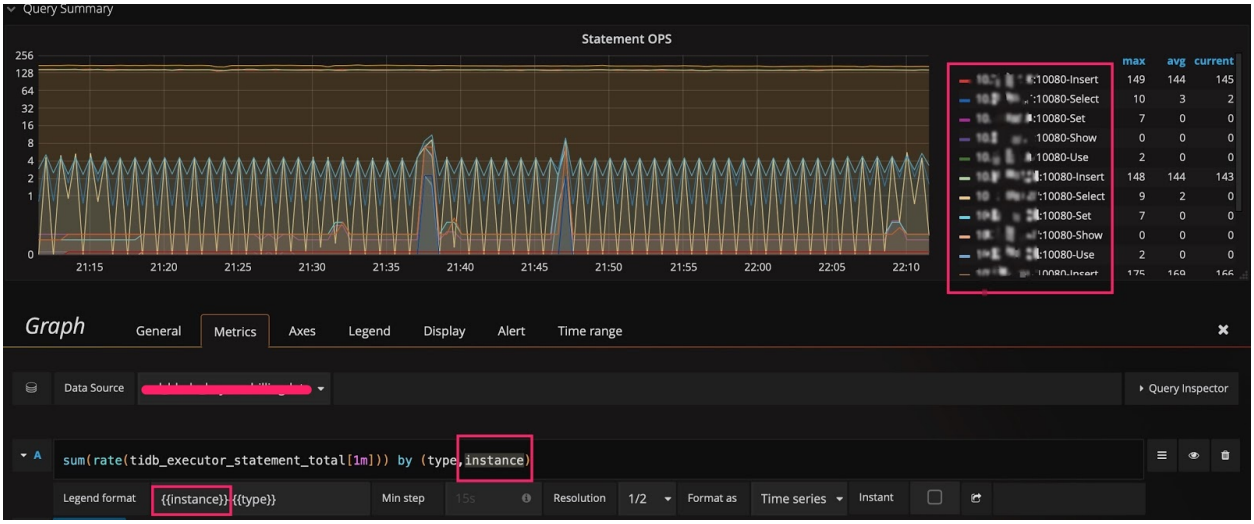


图 335: 给表达式增加一个 instance 维度

4.10.4.3.2 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 336: 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：

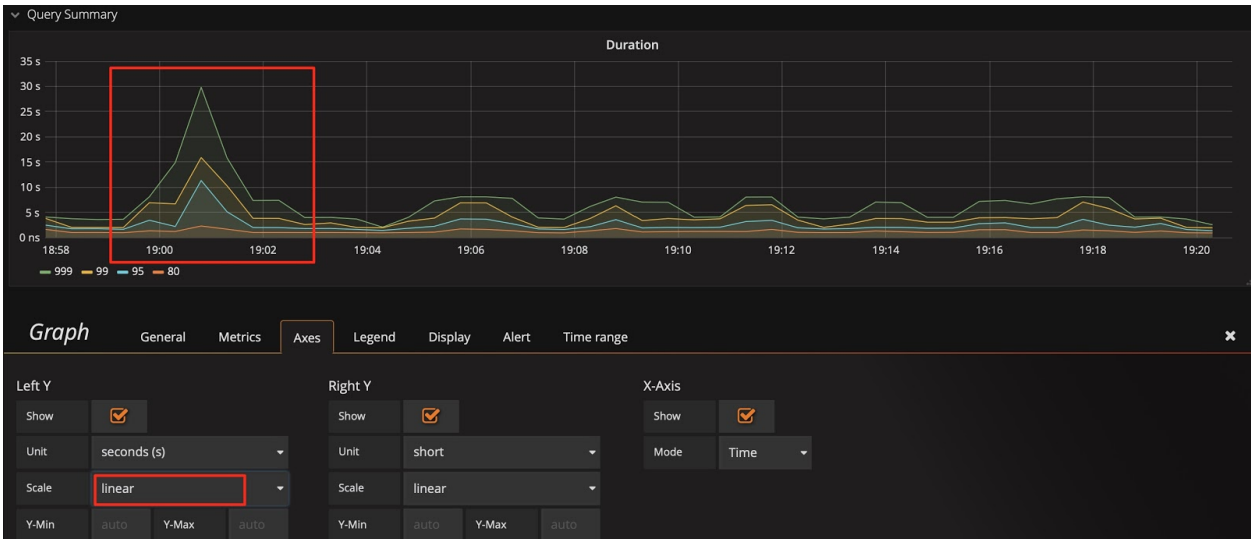


图 337: 调整标尺的比例尺为线性

> **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。

4.10.4.3.3 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：

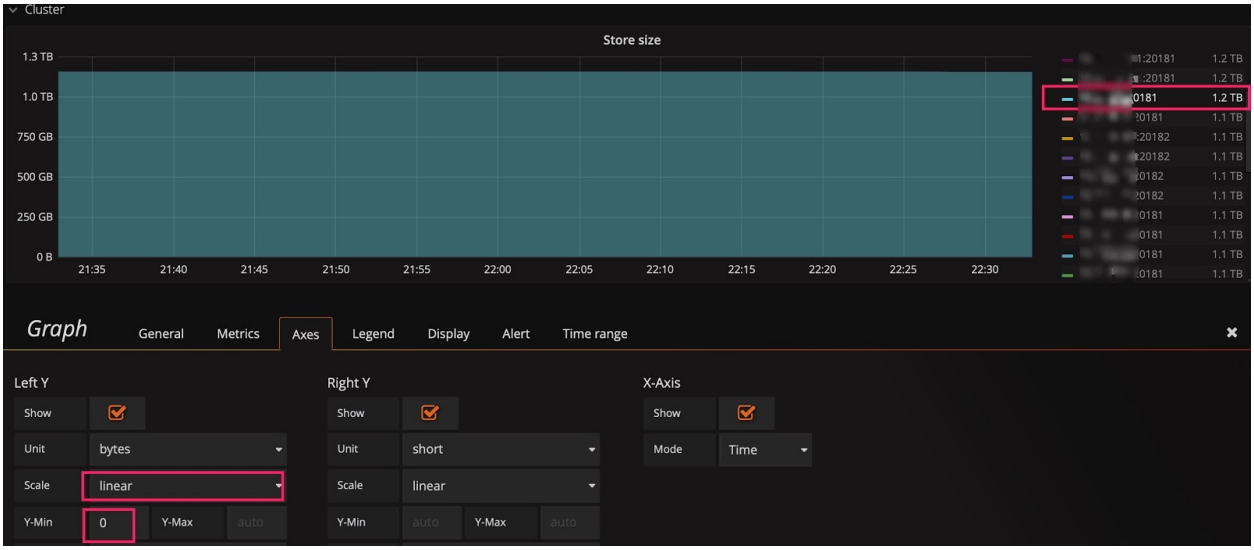


图 338: 基线默认为 0

将基线调整为 auto：

图 339: 调整基线为 auto

4.10.4.3.4 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。

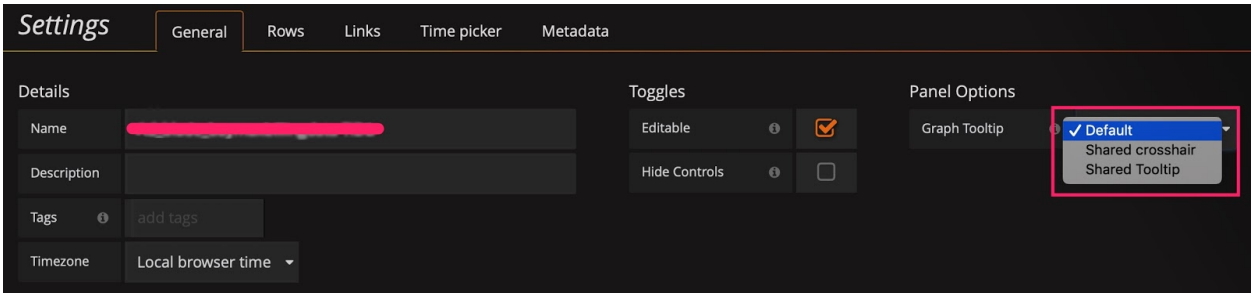


图 340: 图形展示工具

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：

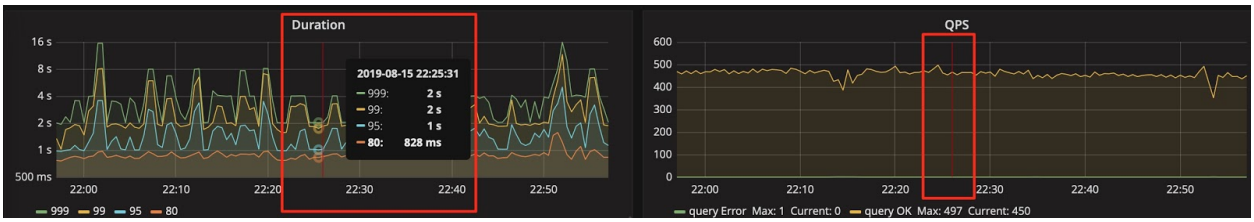


图 341: 调整图形展示工具为 Shared crosshair

将图形展示工具调整为 Shared Tooltip：

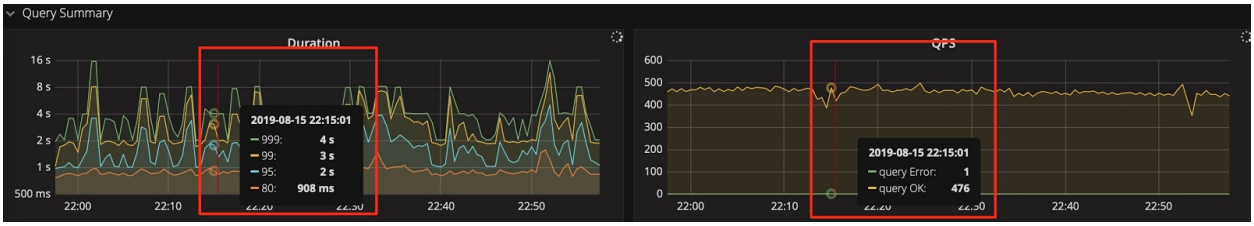


图 342: 调整图形展示工具为 Shared Tooltip

4.10.4.3.5 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。

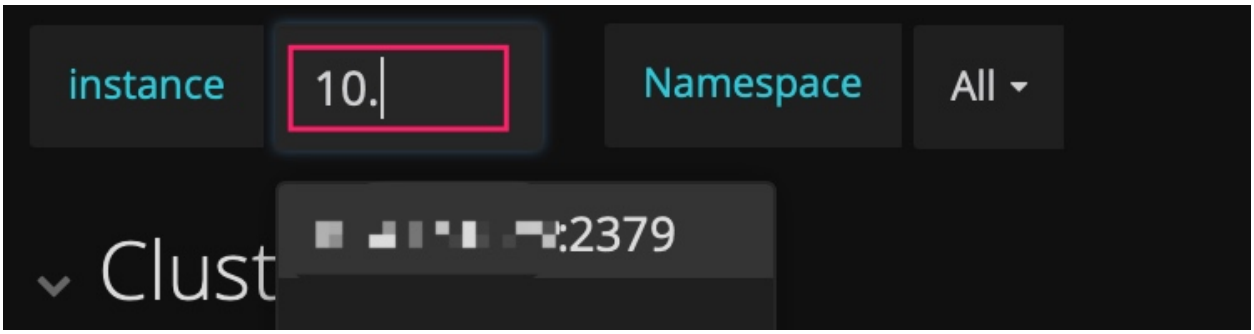


图 343: 查看历史 metric 信息

4.10.4.3.6 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：

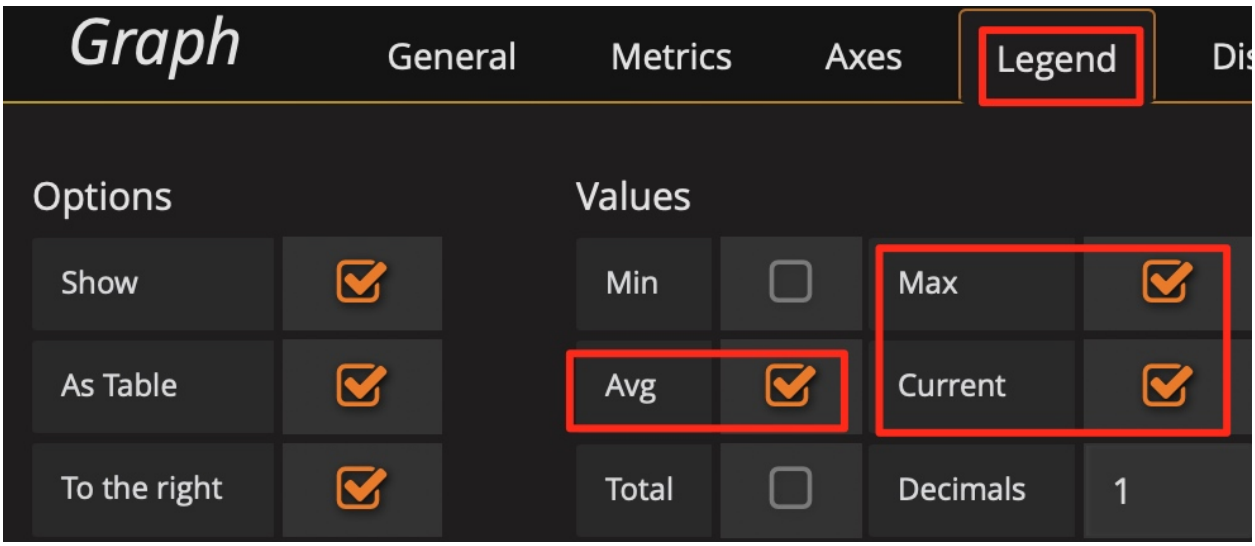


图 344: 增加 Avg 等汇总函数

然后查看整体趋势：

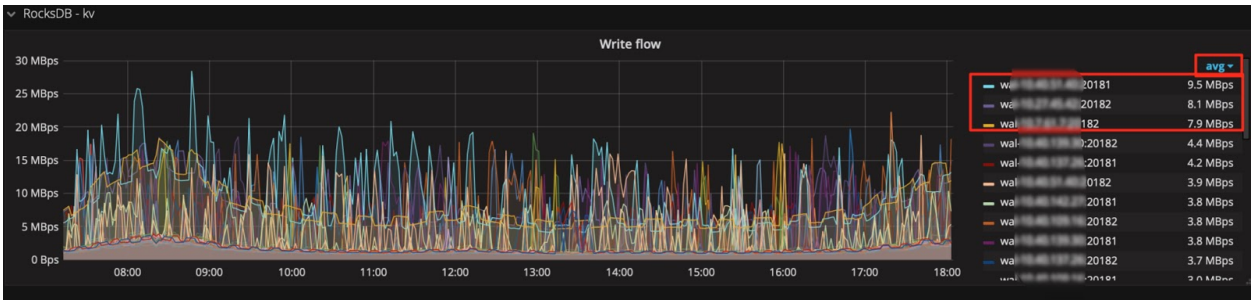


图 345: 增加 Avg 函数查看整体趋势

4.10.4.3.7 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：

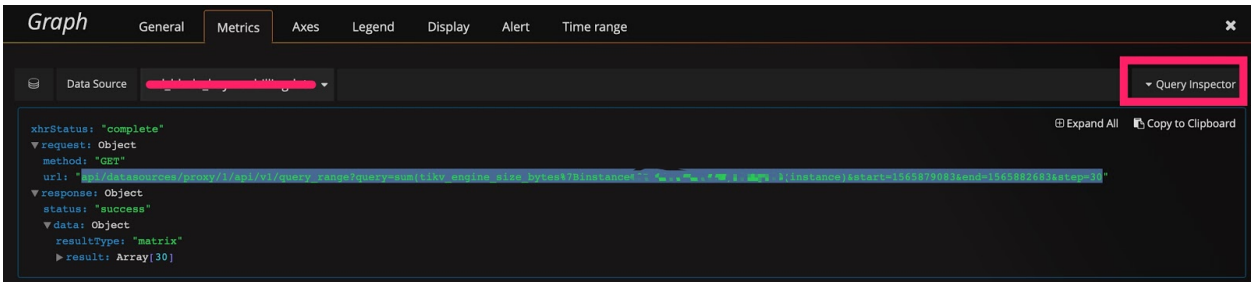


图 346: Prometheus 的 API 接口

```
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"
}
```

4.10.4.4 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。

4.10.5 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。

4.10.5.1 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。

4.10.5.1.1 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。

4.10.5.1.2 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。

4.10.5.1.3 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。

4.10.5.1.4 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。

4.10.5.1.5 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。

4.10.5.1.6 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。

4.10.5.2 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。

4.10.5.2.1 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

4.10.5.2.2 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。

4.10.5.2.3 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

4.10.5.2.4 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

4.10.5.3 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。

4.10.5.3.1 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

4.10.5.3.2 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

4.10.5.3.3 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
- `disable-replace-offline-replica`：停止处理节点下线的调度
- `disable-location-replacement`：停止处理调整 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

4.10.5.4 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。

4.10.5.4.1 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 `disable-replace-offline-replica = true` 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。

4.10.5.4.2 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。

4.10.5.4.3 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。

4.10.5.4.4 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。

4.10.5.4.5 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。

4.10.5.4.6 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。

4.10.6 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。

4.10.6.1 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。

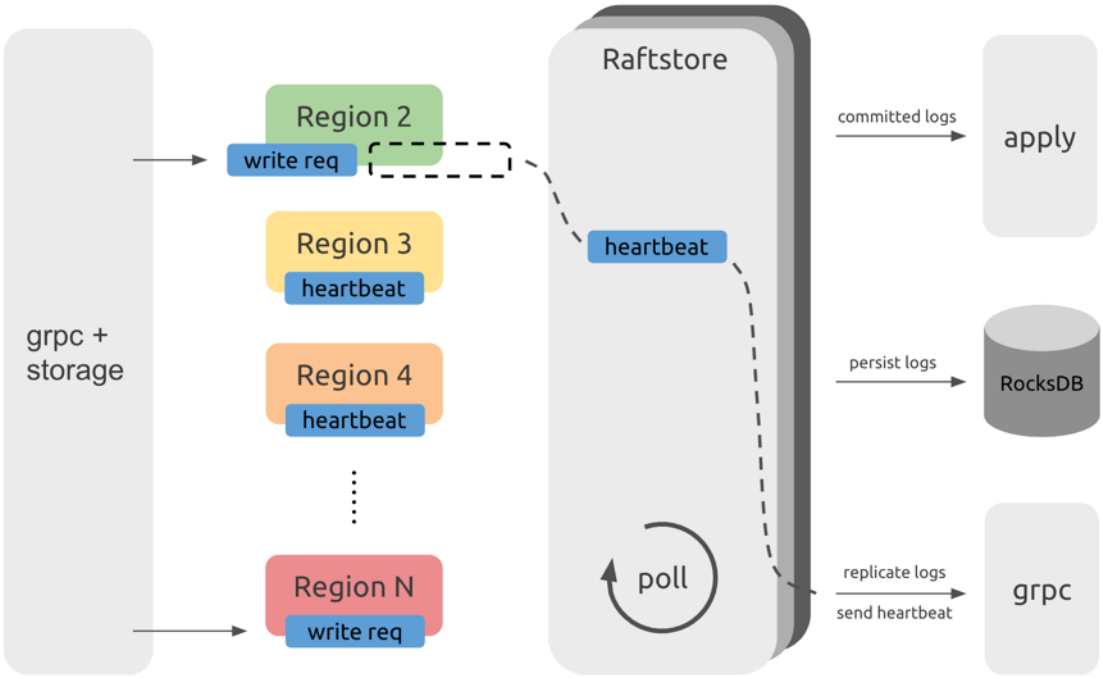


图 347: 图 1 Raftstore 处理流程示意图

> **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。

4.10.6.2 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%。

4.10.6.2.1 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%`。

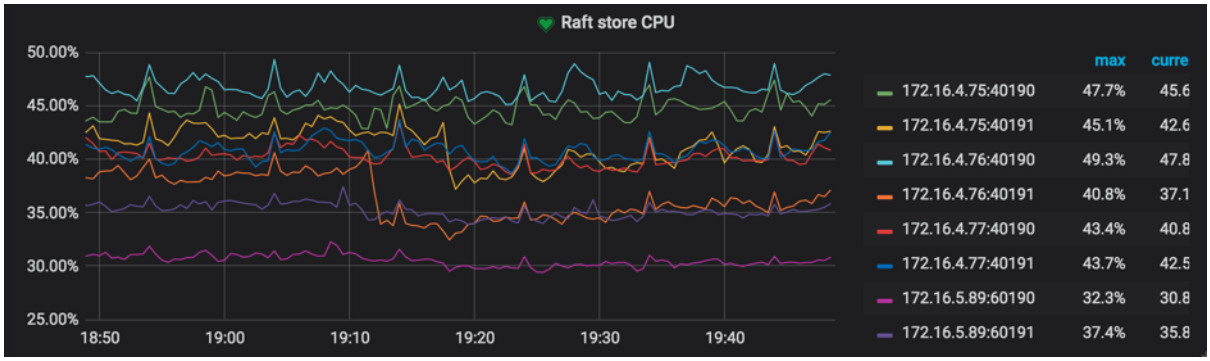


图 348: 图 2 查看 Raftstore CPU

- 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。

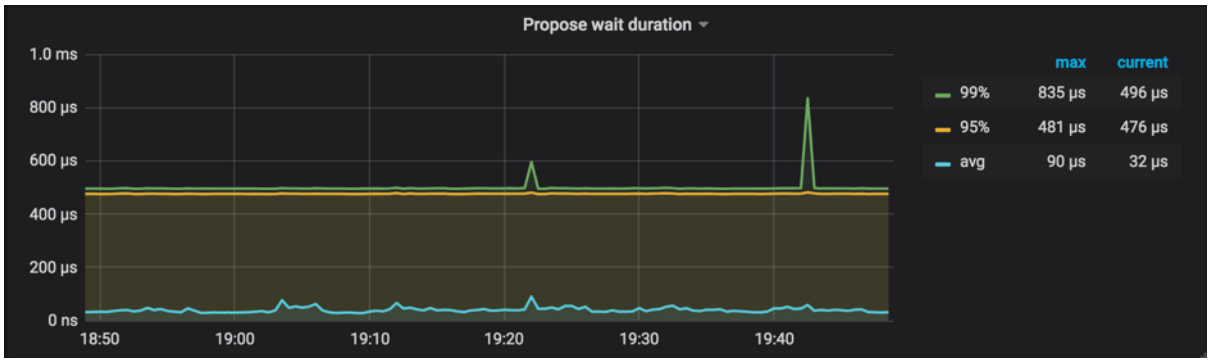


图 349: 图 3 查看 Propose wait duration

4.10.6.3 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

4.10.6.3.1 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。

4.10.6.3.2 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。

4.10.6.3.3 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。

4.10.6.3.4 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。

4.10.6.3.5 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。

4.10.6.4 其他问题和解决方案

4.10.6.4.1 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。

4.10.6.4.2 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。

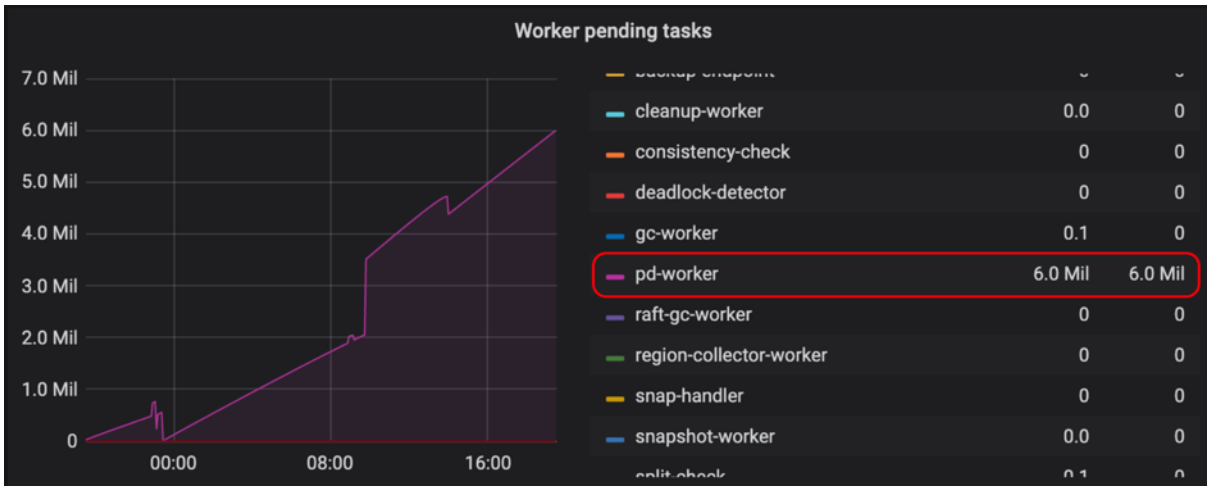


图 350: 图 4 查看 pd-worker

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。

4.10.6.4.3 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。

4.11 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。

4.11.1 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：

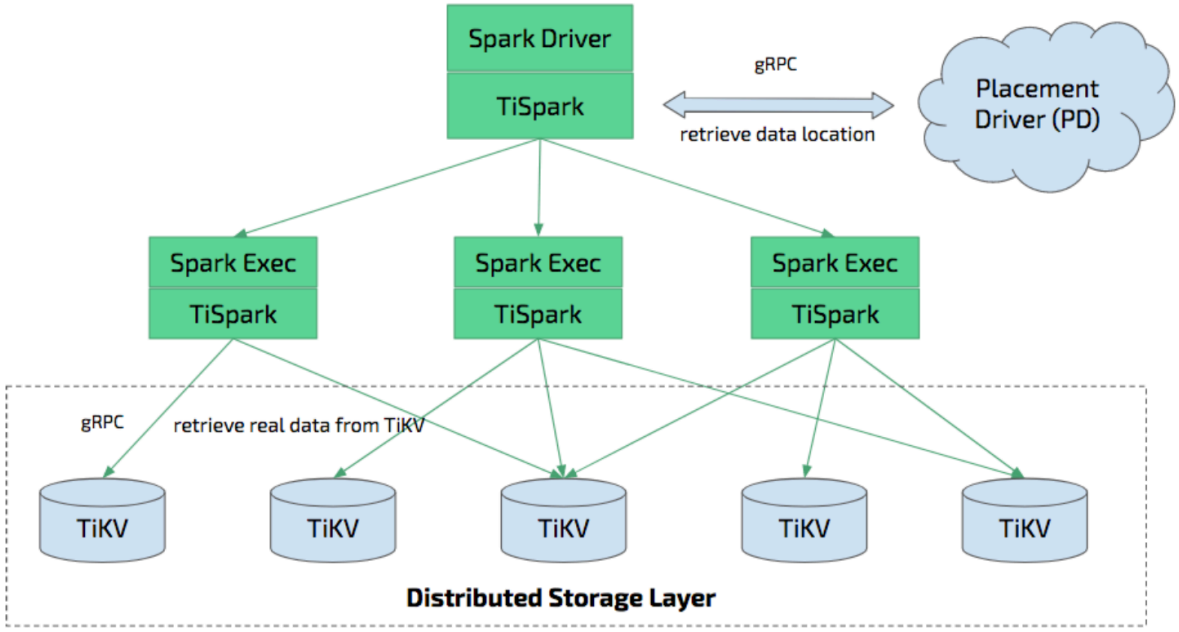


图 351: TiSpark Architecture

- 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。

4.11.2 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。

4.11.3 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。

4.11.3.1 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。

4.11.3.2 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`
`↪` 。

4.11.3.3 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。

4.11.4 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。

4.11.4.1 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
```

4.11.4.2 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。

4.11.4.2.1 下载安装包并安装

你可以在 Apache Spark™ 下载页面下载 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。

4.11.4.2.2 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。

4.11.4.2.3 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。

4.11.4.2.4 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。

4.11.5 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)
```

4.11.6 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
```

4.11.7 通过 JDBC 将 DataFrame 写入 TiDB

暂时 TiSpark 不支持直接将数据写入 TiDB 集群，但可以使用 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。

4.11.8 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。

统计信息将在 Spark Driver 进行缓存，请确定 Driver 内存足够缓存统计信息。可以在`spark-defaults.conf`中开启或关闭统计信息读取：

Property Name	Default	Description
spark.tispark.statistics.auto_load	true	是否默认进行统计信息读取

4.11.9 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。

4.12 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。

4.12.1 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。

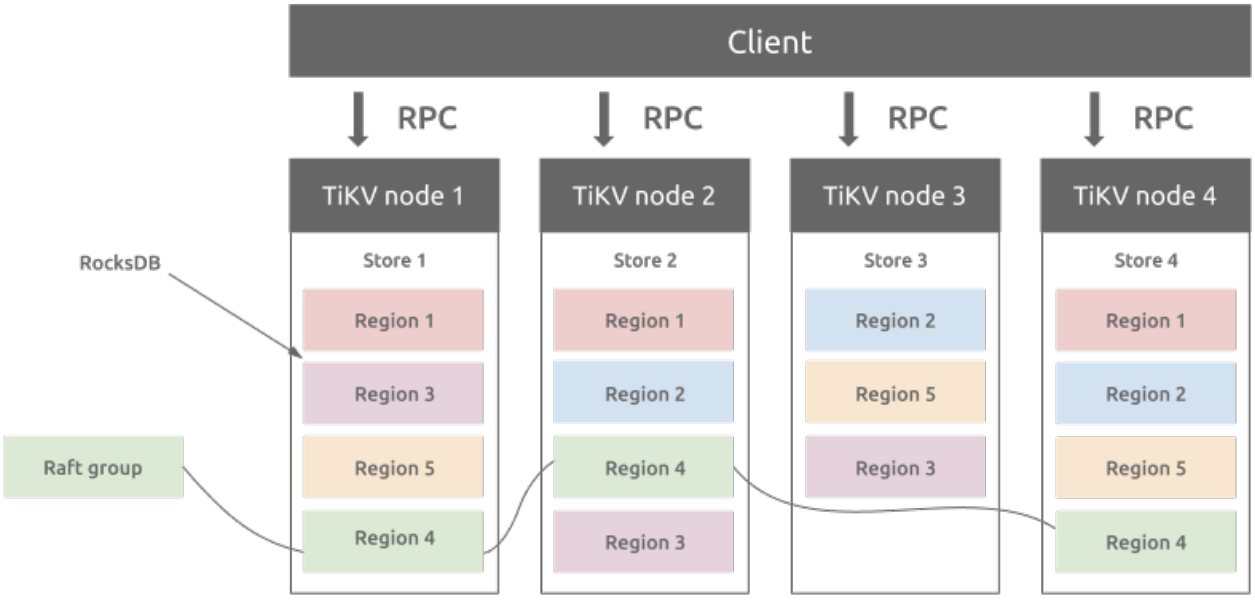


图 352: TiKV 架构

4.12.1.1 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。

4.12.1.2 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。

4.12.2 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。

4.12.3 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。

4.13 TiFlash

4.13.1 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。

4.13.1.1 整体架构

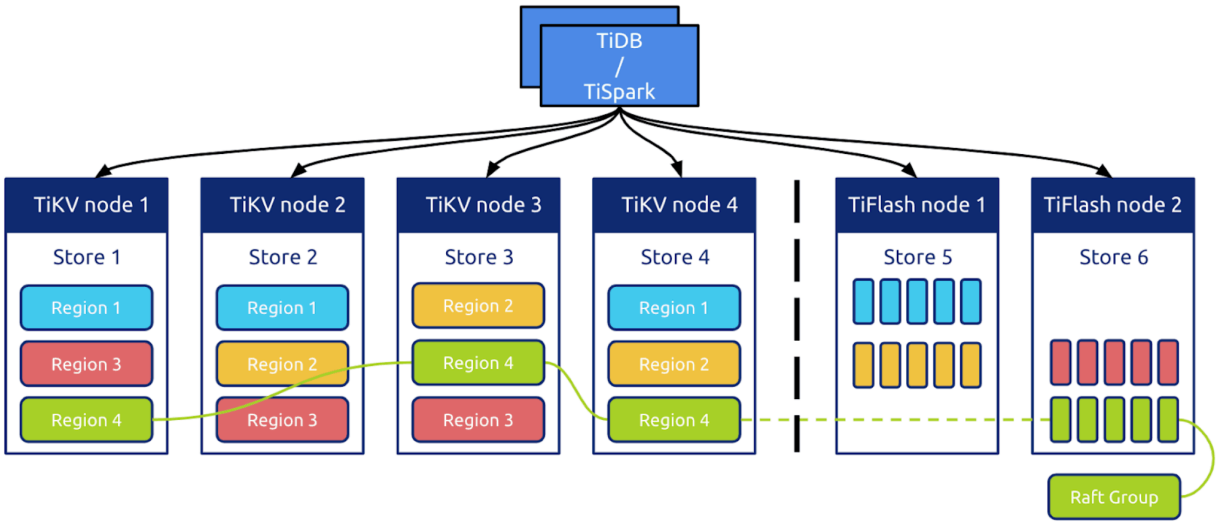


图 353: TiFlash 架构

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。

4.13.1.2 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。

4.13.1.2.1 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。

4.13.1.2.2 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。

4.13.1.2.3 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。

4.13.1.2.4 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。

4.13.2 部署 TiFlash 集群

> **注意：**
>
> 体验 TiFlash RC 版建议和 PingCAP 官方联系，以获得更多资讯和辅助。

本文介绍了部署 TiFlash 集群的环境要求以及不同场景下的部署方式。

4.13.2.1 推荐硬件配置

本节根据不同的 TiFlash 部署模式给出了硬件配置建议。

4.13.2.1.1 TiFlash 单独部署模式

- **最低配置：** 32 VCore, 64 GB RAM, 1 SSD + n HDD
- **推荐配置：** 48 VCore, 128 GB RAM, 1 NVMe SSD + n SSD

部署机器不限，最少一台即可。单台机器可以使用多盘，但不推荐单机多实例部署。

推荐用一个 SSD 盘来缓冲 TiKV 同步数据的实时写入，该盘性能不低于 TiKV 所使用的硬盘，建议是性能更好的 NVMe SSD。该 SSD 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。而其他硬盘，可以选择部署多块 HDD 或者普通 SSD，当然更好的硬盘会带来更好的性能。

TiFlash 支持多盘部署，所以无需使用 RAID。

4.13.2.1.2 TiFlash 和 TiKV 部署在相同节点模式

参考TiKV 节点的硬件配置，并且适当增加内存和 CPU 核数。

建议不要将 TiFlash 与 TiKV 同盘部署，以防互相干扰。

硬盘选择标准同TiFlash 单独部署模式。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 1TB、TiKV 副本数为 3、TiFlash 副本数为 2，则 TiFlash 的推荐总容量为 1024GB / 3 * 2。用户可以选择同步部分表数据而非全部。

4.13.2.2 针对 TiDB 的版本要求

目前 TiFlash 的测试是基于 TiDB 3.1 版本的相关组件（包括 TiDB、PD、TiKV、TiFlash）来进行的，TiDB 3.1 版本的下载方式参考以下安装部署步骤。

4.13.2.3 安装部署 TiFlash

本节介绍了在不同场景下如何安装部署 TiFlash，包括以下场景：

- 全新部署 TiFlash
- 在原有 TiDB 集群上新增 TiFlash 组件

> 注意：
>
> 1. 在开启 TiFlash 进程之前，必须确保 PD 的 Placement Rules 功能已开启（开启方法见在原有 TiDB 集群上新增 TiFlash 组件一节的第 2 步）。
> 2. 在 TiFlash 运行期间，必须确保 PD 的 Placement Rules 功能保持开启状态。

4.13.2.3.1 全新部署 TiFlash

目前对于全新部署 TiFlash 场景，推荐通过下载离线安装包来部署 TiFlash。步骤如下：

1. 下载 TiDB 3.1 版本对应 tag 的 TiDB Ansible：

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

2. 运行如下 local prepare 命令下载 TiDB、TiKV、PD 等组件的二进制文件：

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

3. 编辑 inventory.ini 配置文件，除了部署 TiDB 集群的配置，需要额外在 [tiflash_servers] 下配置 tiflash servers 所在的 ip (目前只支持 ip，不支持域名)。

   如果希望自定义部署目录，需要配置 data_dir 参数，不需要则不加。如果希望多盘部署，则以逗号分隔各部署目录（注意每个 data_dir 目录的上级目录需要赋予 tidb 用户写权限），例如：

```
[tiflash_servers]
192.168.1.1 data_dir=/data1/tiflash/data,/data2/tiflash/data
```

4. 按照 TiDB Ansible 部署流程完成集群部署的剩余步骤。

5. 验证 TiFlash 已部署成功的方式：通过pd-ctl（tidb-ansible 目录下的 resources/bin 包含对应的二进制文件）执行 pd-ctl store http://your-pd-address 命令，可以观测到所部署的 TiFlash 实例状态为 “Up”。

4.13.2.3.2 在原有 TiDB 集群上新增 TiFlash 组件

1. 首先确认当前 TiDB 的版本支持 TiFlash，否则需要先按照TiDB 升级操作指南升级 TiDB 集群至 3.1 rc 以上版本。

2. 在 pd-ctl（tidb-ansible 目录下的 resources/bin 包含对应的二进制文件）中输入 config set enable-placement-rules true 命令，以开启 PD 的 Placement Rules 功能。

3. 编辑 inventory.ini 配置文件，并在 [tiflash_servers] 下配置 tiflash servers 所在的 ip（目前只支持 ip，不支持域名）。

   如果希望自定义部署目录，需要配置 data_dir 参数，不需要则不加。如果希望多盘部署，则以逗号分隔各部署目录（注意每个 data_dir 目录的上级目录需要赋予 tidb 用户写权限），例如：

```
[tiflash_servers]
192.168.1.1 data_dir=/data1/tiflash/data,/data2/tiflash/data
```

> 注意：
> 即使 TiFlash 与 TiKV 同机部署，TiFlash 也会采用与 TiKV 不同的默认端口，默认 9000，无特殊需要可以不用指定，有需要也可在 inventory.ini 配置文件中新增一行 tcp_port=xxx 来指定。

4. 执行以下 ansible-playbook 命令部署 TiFlash：

```
ansible-playbook local_prepare.yml &&
ansible-playbook -t tiflash deploy.yml &&
ansible-playbook -t tiflash start.yml &&
ansible-playbook rolling_update_monitor.yml
```

5. 验证 TiFlash 已部署成功的方式：通过pd-ctl 执行 pd-ctl store http://your-pd-address 命令，可以观测到所部署的 TiFlash 实例状态为 “Up”。

4.13.3 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

4.13.3.1 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
  ```

- 目前版本里，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。
- 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。
- TiFlash 中保留了数据库 system，用户不能为 TiDB 中名字为 system 数据库下的表创建 TiFlash 副本。如果强行创建，结果行为未定义（暂时性限制）。

4.13.3.2 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。

4.13.3.3 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。

4.13.3.3.1 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。

4.13.3.3.2 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 和 tiflash，分别有 2 个配置级别：

1. 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，见 2。

2. TiDB 实例级别，即 INSTANCE 级别，和会话级别配置是交集关系。比如实例级别配置了 “tikv, tiflash”，而会话级别配置了 “tikv”，则只会读取 TiKV。

   在 TiDB 的配置文件添加如下配置项：

   ```
   [isolation-read]
   engines = ["tikv", "tiflash"]
   ```

   实例级别的默认配置为 ["tikv", "tiflash"]。

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。指定了 engine 后，对于查询中的表没有对应 engine 副本的情况（因为 TiKV 副本是必定存在的，因此只有配置了 engine 为 tiflash 而 TiFlash 副本不存在这一种情况），查询会报该表不存在该 engine 副本的错。

4.13.3.3.3 手工 Hint

手工 Hint 可以强制 TiDB 对于某张或某几张表使用 TiFlash 副本，其优先级低于 engine 隔离，如果 Hint 中指定的引擎不在 engine 列表中，会返回 warning，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;
```

更多关于该 Hint 语句的语法可以参考READ_FROM_STORAGE。

Engine 隔离的优先级高于 CBO 与 Hint，Hint 优先级高于代价估算，即代价估算仅会选取指定 engine 的副本。

> 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。

4.13.3.4 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 true 时，所有查询的表都会只读取 TiFlash 副本，设为 false 则只读取 TiKV 副本。设为 true 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

4.13.3.5 TiFlash 支持的计算下推

TiFlash 主要支持谓词、聚合下推计算，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主要是表连接和 DISTINCT COUNT，会在后续版本逐步优化。

目前 TiFlash 支持了有限的常用表达式下推，支持下推的表达式可参考该文件。

目前 TiFlash 不支持下推的情况包括：

- 所有包含 Time 类型和 JSON 类型的表达式均不能下推
- 在聚合函数或者 WHERE 条件中包含了不在该文件列表中的表达式，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。

4.13.4 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、下线 TiFlash 节点、TiFlash 故障处理等，以及 TiFlash 重要日志及系统表。

4.13.4.1 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
```

4.13.4.2 下线 TiFlash 节点

下线 TiFlash 节点与缩容 TiFlash 节点不同，下线 TiFlash 并不会在 TiDB Ansible 中删除这个节点，而仅仅是安全地关闭这个进程。

下线 TiFlash 节点的步骤如下：

> 注意：
>
> 如果下线该 TiFlash 节点后，TiFlash 集群剩余节点数大于等于所有数据表的最大副本数，直接进行下面第 3 步。

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）。
3. 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，查看该 TiFlash 节点对应的 store id。
4. 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。
5. 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。

> 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
```

4.13.4.3 TiFlash 故障处理

本节介绍了一些 TiFlash 常见问题、原因及解决办法。

4.13.4.3.1 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. 检查 PD 是否开启 Placement Rules 功能（开启方法见在原有 TiDB 集群上新增 TiFlash 组件的第 2 步）：

    ```
    echo 'config show replication' | /path/to/pd-ctl -u http://<pd-ip>:<pd-port>
    ```

    预期结果为 "enable-placement-rules": "true"。

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 PD 配置的 max-replicas 是否小于等于集群 TiKV 节点数。若 max-replicas 超过 TiKV 节点数，则 PD 不会向 TiFlash 同步数据；

    ```
    echo 'config show replication' | /path/to/pd-ctl -u http://<pd-ip>:<pd-port>
    ```

    再确认 “max-replicas” 参数值。

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。

4.13.4.3.2 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。

4.13.4.3.3 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。

4.13.4.3.4 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。

4.13.4.4 TiFlash 重要日志介绍

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

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。

4.13.4.5 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]

4.13.5 TiFlash 集群监控

本文介绍 TiFlash 集群的相关监控项及说明。

4.13.5.1 COPROCESSOR 相关监控

监控指标名称	监控指标说明
tiflash_coprocessor_request_count	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
tiflash_coprocessor_executor_count	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
tiflash_coprocessor_request_duration_seconds	每个 coprocessor request 总时间直方图，总时间为接收到该 coprocessor 请求至请求应答完毕的时间，其中 batch 是 batch 请求的总时间，cop 是直接通过 coprocessor 接口发送的 coprocessor 请求总时间
tiflash_coprocessor_request_error	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
tiflash_coprocessor_request_handle_seconds	每个 coprocessor 请求处理时间直方图，处理时间为该 coprocessor 请求开始执行到执行结束的时间，其中 batch 是 batch 请求的处理时间，cop 是直接通过 coprocessor 接口发送的 coprocessor 请求处理时间
tiflash_coprocessor_response_bytes	应答总字节数

4.13.5.2 DDL 相关监控

监控指标名称	监控指标说明
tiflash_schema_version	TiFlash 目前缓存的 schema 版本
tiflash_schema_apply_count	分为 diff apply、 full apply 和 failed apply：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
tiflash_schema_internal_ddl_count	TiFlash 内部进行的具体 DDL 操作的总数
tiflash_schema_apply_duration_seconds	单次 apply schema 消耗的时间

4.13.5.3 Raft 相关监控

监控指标名称	监控指标说明
tiflash_raft_read_index_count	coprocessor 触发 read_index 请求的次数，等于一个 coprocessor 触发的 Region 总数
tiflash_raft_read_index_duration_seconds	read_index 消耗的时间，主要消耗时间在于和 leader 的交互和重试时间
tiflash_raft_wait_index_duration_seconds	wait_index 消耗的时间，即拿到 read_index 请求后，等待 local index >= read_index 所花费的时间

4.13.6 TiFlash 集群扩缩容

本文介绍扩缩容 TiFlash 集群节点的步骤。

4.13.6.1 扩容 TiFlash 节点

以在节点 192.168.1.1 上部署 TiFlash 为例，扩容该 TiFlash 节点的步骤如下。

1. 编辑 `inventory.ini` 文件，添加该 TiFlash 节点信息（目前只支持 ip，不支持域名）：

```
[tiflash_servers]
192.168.1.1
```

2. 编辑 `hosts.ini` 文件，添加节点信息：

```
[servers]
192.168.1.1

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

3. 初始化新增节点：

    - 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 192.168.1.1 -u root -k
    ```

    - 在部署目标机器上安装 NTP 服务：

    ```
    ansible-playbook -i hosts.ini deploy_ntp.yml -u tidb -b
    ```

    - 在部署目标机器上初始化节点：

```
ansible-playbook bootstrap.yml -l 192.168.1.1
```

4. 部署新增节点：

```
ansible-playbook deploy.yml -l 192.168.1.1
```

5. 启动新节点服务：

```
ansible-playbook start.yml -l 192.168.1.1
```

6. 更新 Prometheus 配置并重启：

```
ansible-playbook rolling_update_monitor.yml --tags=prometheus
```

7. 打开浏览器访问监控平台，监控整个集群和新增节点的状态。

4.13.6.2 缩容 TiFlash 节点

以停止 192.168.1.1 节点的服务为例，缩容该 TiFlash 节点的步骤如下。

> 注意：
>
> 本节介绍的下线流程不会删除下线节点上的数据文件，如需再次上线，请先手动删除。

1. 首先参考下线 TiFlash 节点章节，对要进行缩容的 TiFlash 节点进行下线操作。
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）。
3. 等待 TiFlash 对应的 `store` 消失，或者 `state_name` 变成 `Tombstone` 后，执行如下命令关闭 TiFlash 进程：

```
ansible-playbook stop.yml -l 192.168.1.1
```

如果该节点仍有其他服务，只希望停止 TiFlash 则请注明 TiFlash 服务：

```
ansible-playbook stop.yml -t tiflash -l 192.168.1.1
```

4. 编辑 `inventory.ini` 和 `hosts.ini` 文件，移除节点信息。
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，监控整个集群的状态。

4.13.7 升级 TiFlash 节点

> **注意：**
>
> TiFlash Pre-RC 版本升级到更高版本请联系 PingCAP 官方，以获得更多资讯和帮助。

升级前请保证集群处于启动状态，升级 TiFlash 节点的步骤如下：

1. 备份 tidb-ansible 文件夹

```
mv tidb-ansible tidb-ansible-bak
```

2. 下载 TiDB 3.1 版本对应 tag 的 tidb-ansible

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

3. 下载 binary

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

4. 编辑 inventory.ini 文件
5. 滚动升级 TiFlash：

```
ansible-playbook rolling_update.yml --tags tiflash
```

6. 滚动升级 TiDB 监控组件：

```
ansible-playbook rolling_update_monitor.yml
```

4.13.8 TiFlash 配置参数

本文介绍了与部署使用 TiFlash 相关的配置参数。

4.13.8.1 PD 调度参数

可通过pd-ctl（tidb-ansible 目录下的 `resources/bin` 包含对应的二进制文件）调整参数：

- `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。

4.13.8.2 TiFlash 配置参数

4.13.8.2.1 配置文件 tiflash.toml

```
tmp_path = tiflash 临时文件存放路径
path = tiflash 数据存储路径      # 如果有多个目录，以英文逗号分隔
path_realtime_mode = false # 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，较旧的数据存放于其他目录。
listen_host = tiflash 服务监听 host # 一般配置成 0.0.0.0
tcp_port = tiflash tcp 服务端口
http_port = tiflash http 服务端口
```

```
[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，需要 3 个参数控制。

```
[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 = proxy 对外访问地址
    data-dir = proxy 数据存储路径
    config = proxy 配置文件路径
    log-file = proxy log 路径
```

```
[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]
    kvstore_path = kvstore 数据存储路径 # 默认为 "{path 的第一个目录}/kvstore"
    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
```

4.13.8.2.2 配置文件 tiflash-learner.toml

```
[server]
    engine-addr = tiflash coprocessor 服务监听地址
    status-addr = Prometheus 拉取 proxy metrics 信息的 ip + 端口
```

4.13.8.2.3 多盘部署

TiFlash 支持多盘部署，主要通过配置文件 `tiflash.toml` 中的 `path` 和 `path_realtime_mode` 这两个参数控制。

多个数据存储目录在 `path` 中以英文逗号分隔，比如 `/ssd_a/data/tiflash,/hdd_b/data/tiflash,/hdd_c/data/tiflash`。如果你的环境有多块磁盘，推荐一个数据存储目录对应一块磁盘，并且把性能最好的磁盘放在最前面，以发挥所有磁盘的全部性能。

`path_realtime_mode` 参数默认值为 false，表示数据会在所有的存储目录之间进行均衡。如果设为 true，且 `path` 配置了多个目录，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。

4.13.9 TiFlash 命令行参数

本文介绍了 TiFlash 的命令行启动参数。

4.13.9.1 `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

4.13.10 TiFlash 报警规则

本文介绍了 TiFlash 集群的报警规则。

4.13.10.1 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。

4.13.10.2 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。

4.13.10.3 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。

4.13.10.4 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。

4.13.11 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。

4.13.11.1 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。

4.13.11.2 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. 开启聚合推过 JOIN / UNION 等 TiDB 算子的优化：

```
set @@tidb_opt_agg_push_down = 1;
```

4.13.12 TiFlash 常见问题

本文介绍 TiFlash 使用过程中的常见问题及解决方案。

4.13.12.1 TiFlash 是否可直接写入？

TiFlash 暂时无法直接接受写入，只能通过写入 TiKV 再同步到 TiFlash。

4.13.12.2 如果想在已经存在的集群增加 TiFlash，怎么去估算存储资源？

可以衡量哪些表可能需要加速，这些表的单副本大小大致就是 TiFlash 两副本所需的空间，再算上计划的余量就行。

4.13.12.3 TiFlash 的数据如何做到高可用？

TiFlash 可以通过 TiKV 恢复数据。只要 TiKV 的对应 Region 是可用的，TiFlash 就可以从中恢复数据。

4.13.12.4 TiFlash 推荐设置多少个副本？

如果需要 TiFlash 服务本身高可用（并非数据高可用），那么推荐将 TiFlash 设置成 2 副本；如果可以允许 TiFlash 丢失节点的情况下通过 TiKV 副本继续服务，那么也可以使用单副本。

4.13.12.5 什么时候使用 TiSpark 进行查询？什么时候使用 TiDB server 进行查询？

如果查询以单表聚合和过滤为主，那么 TiDB server 比 TiSpark 在列存上拥有更好的性能；如果查询以表连接为主，那么推荐使用 TiSpark。

4.14 TiDB Binlog

4.14.1 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

4.14.1.1 TiDB Binlog 整体架构

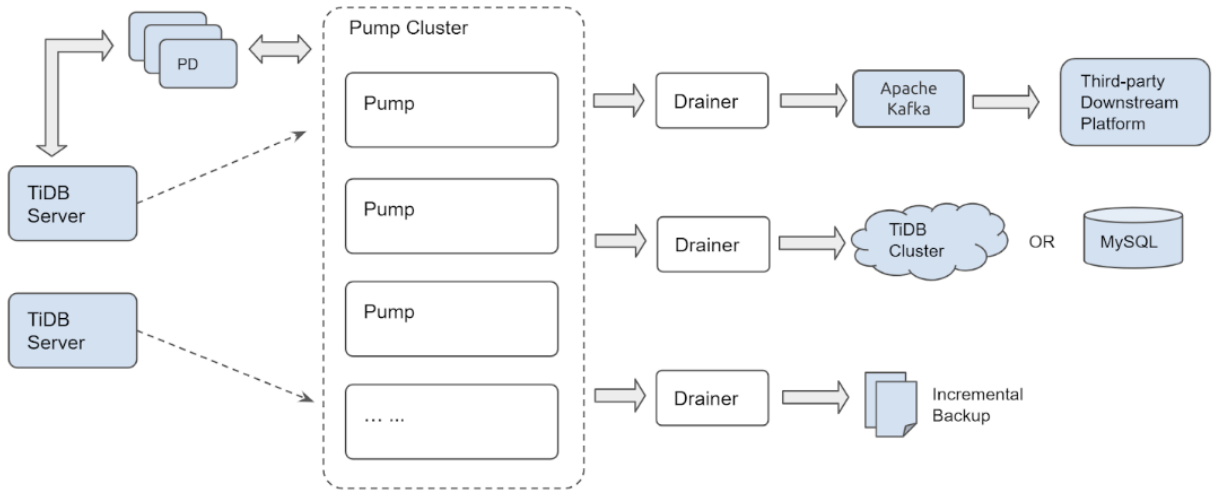


图 354: TiDB Binlog 架构

TiDB Binlog 集群主要分为 Pump 和 Drainer 两个组件，以及 binlogctl 工具：

4.14.1.1.1 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。

4.14.1.1.2 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。

4.14.1.1.3 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

4.14.1.2 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。

4.14.1.3 注意事项

- 需要使用 TiDB v2.0.8-binlog、v2.1.0-rc.5 及以上版本，否则不兼容该版本的 TiDB Binlog。
- 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。

4.14.2 TiDB Binlog 集群部署

4.14.2.1 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

4.14.2.2 使用 TiDB Ansible 部署 TiDB Binlog

4.14.2.2.1 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
release-3.0	3.0 版本	最新 3.0 稳定版本，可用于生产环境（建议）。

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 3.0 版本：

```
git clone -b release-3.0 https://github.com/pingcap/tidb-ansible.git
```

4.14.2.2.2 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 3.0.2 及之前版本中为 `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}
```

4.14.2.2.3 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。但是需要注意 v3.0.0，v3.0.1 的配置文件命名规则与其余版本略有不同，为 别名_drainer-cluster.toml。

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
```

4.14.2.3 使用 Binary 部署 TiDB Binlog

4.14.2.3.1 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
```

4.14.2.3.2 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，v3.0.1 后支持该功能
# 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-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
    v3.0.2 后支持该功能，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，v3.0.2 后支持该功能
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"
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

[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。

4.14.3 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。

4.14.3.1 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。

4.14.3.2 Pump/Drainer 的启动、退出流程

4.14.3.2.1 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。

4.14.3.2.2 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。

4.14.3.3 binlogctl 工具

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

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
```

对于 v2.1.0 GA 及以上版本，binlogctl 已经包含在 TiDB 的下载包中，其他版本需要单独下载 binlogctl:

```
wget https://download.pingcap.org/tidb-enterprise-tools-latest-linux-amd64.tar.gz && \
wget https://download.pingcap.org/tidb-enterprise-tools-latest-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
```

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 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。

4.14.3.4 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

```
show variables like "log_bin";
```

```
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| log_bin       |  ON   |
+---------------+-------+
```

值为 ON 时表示 TiDB 开启了 binlog。

- 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`。

4.14.4 TiDB Binlog 配置说明

本文档介绍 TiDB Binlog 的各项配置说明。

4.14.4.1 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。

4.14.4.1.1 addr

- HTTP API 的监听地址，格式为 `host:port`。
- 默认：`"127.0.0.1:8250"`

4.14.4.1.2 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。

4.14.4.1.3 socket

- HTTP API 监听的 Unix socket 地址。
- 默认：`""`

4.14.4.1.4 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"`

4.14.4.1.5 data-dir

- 本地存放 binlog 及其索引的目录。
- 默认：`"data.pump"`

4.14.4.1.6 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

4.14.4.1.7 gen-binlog-interval

- 指定写入 fake binlog 的间隔秒数。
- 默认：3

4.14.4.1.8 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

4.14.4.1.9 log-file

- 保存日志文件的路径。如果为空，日志不会被保存。
- 默认：`""`

4.14.4.1.10 log-level

- Log 等级。
- 默认：`"info"`

4.14.4.1.11 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`。

4.14.4.1.12 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：`""`

4.14.4.1.13 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。

4.14.4.2 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。

4.14.4.2.1 addr

- HTTP API 的监听的地址，格式为 `host:port`。
- 默认：`"127.0.0.1:8249"`

4.14.4.2.2 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

4.14.4.2.3 log-file

- 日志的文件保存路径。如果为空，日志不会被保存。
- 默认：`""`

4.14.4.2.4 log-level

- Log 等级。
- 默认：`"info"`

4.14.4.2.5 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`。

4.14.4.2.6 data-dir

- 用于存放 Drainer 运行中需要保存的文件的目录。
- 默认：`"data.drainer"`

4.14.4.2.7 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

4.14.4.2.8 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"`

4.14.4.2.9 initial-commit-ts

- 指定从哪个 commit timestamp 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。
- 默认：-1。Drainer 会从 PD 得到一个最新的 timestamp 作为初始时间。

4.14.4.2.10 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

4.14.4.2.11 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。

4.14.4.2.12 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：""

4.14.4.2.13 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

4.14.4.2.14 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`

4.14.4.2.15 syncer.to.checkpoint

以下是 `syncer.to.checkpoint` 相关的配置项。

4.14.4.2.16 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`

4.14.4.3 TiDB Binlog 配置说明

本文档介绍 TiDB Binlog 的各项配置说明。

4.14.4.3.1 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。

4.14.4.3.2 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 commit timestamp 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。
- 默认：-1。Drainer 会从 PD 得到一个最新的 timestamp 作为初始时间。

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`

4.14.5 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。

4.14.5.1 TiDB Ansible 部署

本节适用于使用 TiDB Ansible Playbook 部署的情况。

4.14.5.1.1 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

4.14.5.1.2 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

4.14.5.2 手动部署

4.14.5.2.1 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

4.14.5.2.2 升级 Drainer

1. 用新版本的 `drainer` 替换原来的文件
2. 重启 Drainer 进程

4.14.5.3 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	
Kafka	TiDB 1.0 ~ TiDB 2.1 RC5	TiDB 1.0 支持 local 版本和 Kafka 版本的 TiDB Bin-log。

TiDB Binlog 版本	TiDB 版本	说明
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。

4.14.5.3.1 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。

4.14.6 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。

4.14.6.1 监控指标

4.14.6.1.1 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
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

metric 名称	说明
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

4.14.6.1.2 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

4.14.6.2 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)。

4.14.6.2.1 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。

4.14.6.2.2 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。

4.14.6.2.3 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。

4.14.7 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

4.14.7.1 Reparo 使用

4.14.7.1.1 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）。
```

4.14.7.1.2 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 = ""
```

4.14.7.1.3 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。

4.14.8 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。

4.14.8.1 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"
```

4.14.8.2 自定义开发

4.14.8.2.1 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。

4.14.8.2.2 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。

4.14.9 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。

4.14.9.1 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":{}} |
+--------------------+------------------------------------------------------------------+
```

4.14.9.2 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。

4.14.9.2.1 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。

4.14.9.2.2 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。

4.14.9.3 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
```

4.14.10 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。

4.14.10.1 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：

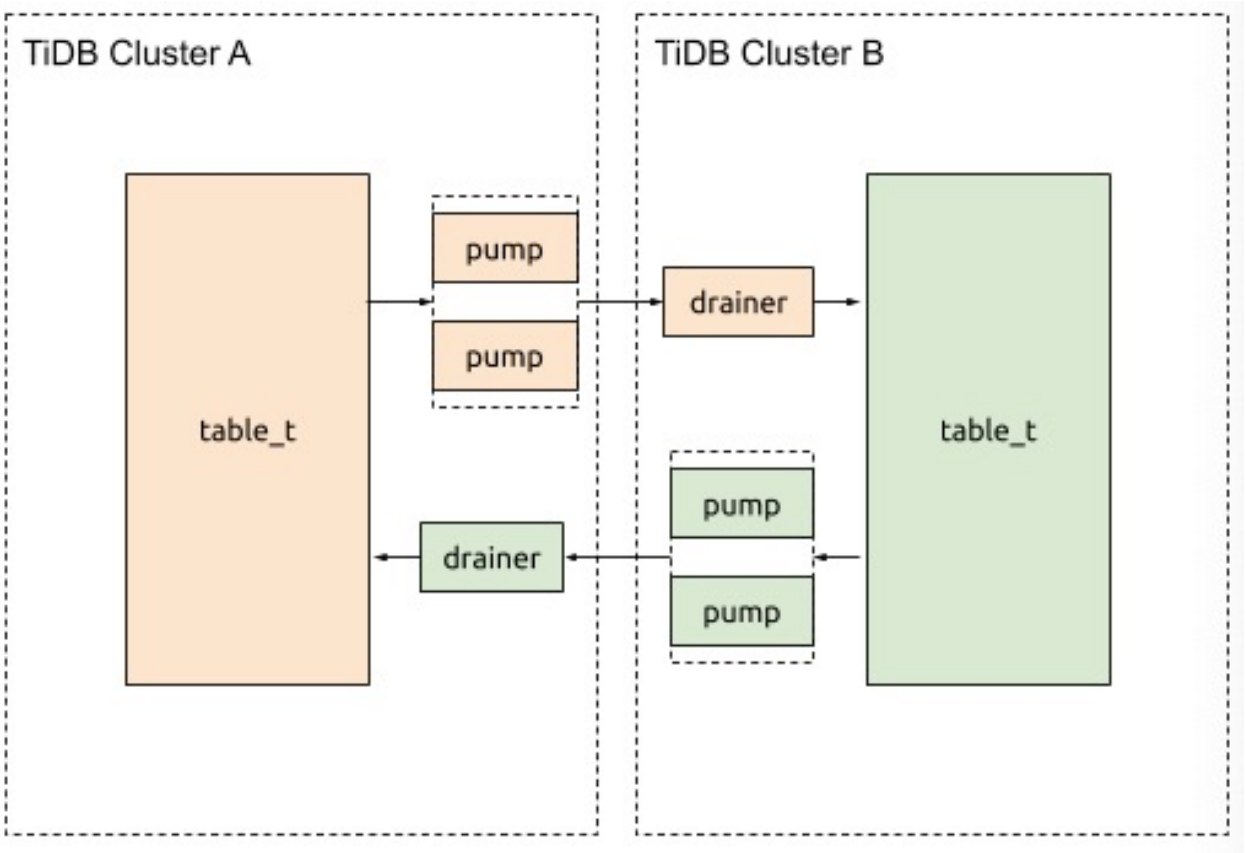


图 355: 使用场景示例图

4.14.10.2 实现原理

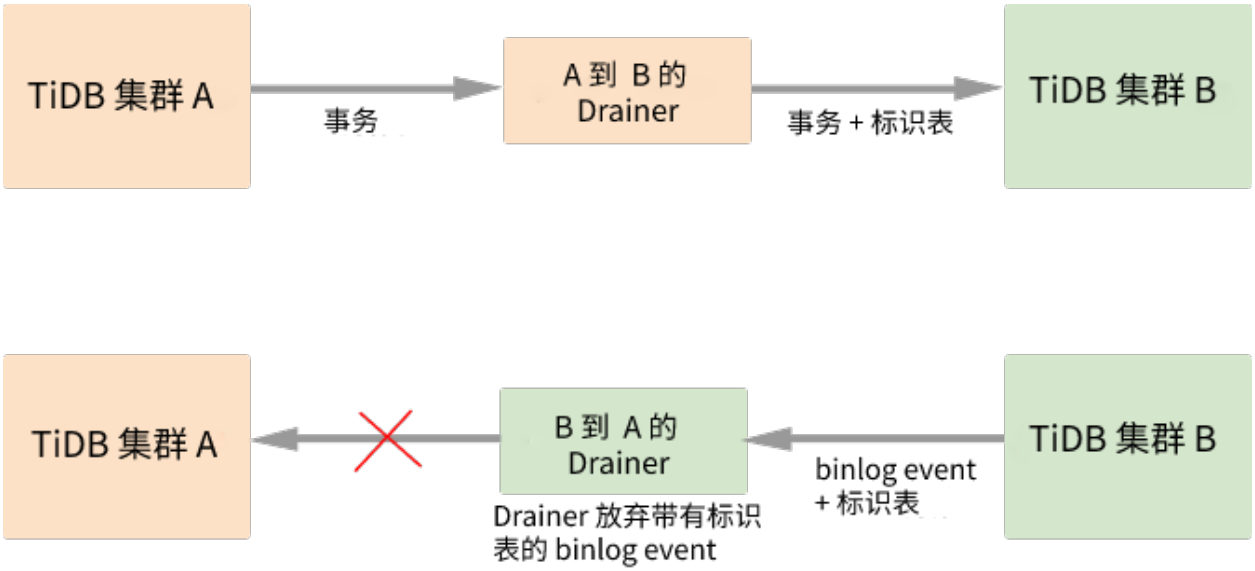


图 356: 原理示例图

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。

4.14.10.3 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 = ?;
```

4.14.10.4 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。

4.14.10.5 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"
```

4.14.11 TiDB Binlog 术语表

本文档介绍 TiDB Binlog 相关术语。

4.14.11.1 Binlog

在 TiDB Binlog 中，Binlog 通常 TiDB 写的 Binlog 数据，注意 TiDB Binlog 的格式跟 MySQL 不同，也指 Drainer 写到 Kafka 或者文件的 Binlog 数据，但他们的格式不一样。

4.14.11.2 Binlog event

TiDB 写的 DML Binlog 有 3 种 event, 分别是 Insert, Update, Delete, Drainer 监控面板可以看到对应同步数据不同 event 的个数。

4.14.11.3 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 接着从对应 commit-ts 同步数据到下游。

4.14.11.4 Safe mode

指增量复制过程中，用于支持在表结构中存在主键或唯一索引的条件下可重复导入 DML 的模式。

该模式的主要特点为将来自上游的 INSERT 改写为 REPLACE，将 UPDATE 改写为 DELETE 与 REPLACE 后再向下游执行。在 Drainer 启动的前 5 分钟会自动启动 Safe mode，另外也可以配置文件中通过 safe-mode 参数手动开启。

该配置仅对下游是 MySQL/TiDB 有效。

4.14.12 故障诊断

4.14.12.1 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。

4.14.12.2 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。

4.14.12.2.1 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
```

4.14.12.2.2 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。

4.14.12.2.3 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。

4.14.13 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。

4.14.13.1 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。

4.14.13.2 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。

4.14.13.3 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

4.14.13.4 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，版本 <= v3.0.0 的 pump 会保证非 `offline` 状态 Drainer 消费了数据才会 gc，如果有不再使用的 Drainer 需要使用 binlogctl 下线。

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。

4.14.13.5 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。

4.14.13.6 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。

4.14.13.7 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}。

4.14.13.8 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。

4.14.13.9 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`。

4.14.13.10 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。

4.14.13.11 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}。

4.14.13.12 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。

4.14.13.13 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。

4.14.13.14 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。

4.14.13.15 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。

4.14.13.16 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。

4.14.13.17 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。

4.14.13.18 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。

4.14.13.19 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。

4.14.13.20 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。

4.14.13.21 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。

4.14.13.22 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。

4.14.13.23 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。

4.15 工具

4.15.1 TiDB 工具适用场景

本文档从工具的适用场景出发，介绍部分常见场景下的工具选择。

4.15.1.1 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。

4.15.1.2 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。

4.15.1.3 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。

4.15.1.4 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。

4.15.1.5 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。

4.15.1.6 迁出数据到 MySQL/TiDB

当需要将 TiDB 集群的数据迁出到 MySQL 或其他 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 MySQL/TiDB。

如果还需要执行增量数据的迁移，则可使用TiDB Binlog。

4.15.1.7 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。

4.15.2 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。

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。

4.15.2.1 使用表库过滤

4.15.2.1.1 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/
#          ^~~~~~~~~~~~~~~~~~~~~~
```

- Lightning：

```
./tidb-lightning -f 'foo*.*' -f 'bar*.*' -d /tmp/data/ --backend tidb
#                ^~~~~~~~~~~~~~~~~~~~~~
```

4.15.2.1.2 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：

- Lightning：

```
[mydumper]
filter = ['foo*.*', 'bar*.*']
```

4.15.2.2 表库过滤语法

4.15.2.2.1 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。

4.15.2.2.2 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。

4.15.2.2.3 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。

4.15.2.2.4 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
```

4.15.2.2.5 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
```

4.15.2.2.6 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。

4.15.2.2.7 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*.*
```

4.15.2.2.8 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 `/`。

4.15.2.3 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）
employees.departments	√	√	√	规则 3（接受）
else.departments		√	√	规则 3（接受）

注意：

在 TiDB 生态工具中，无论表库过滤如何设置，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`

4.15.3 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私有部署的 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

4.15.4 Mydumper 使用文档

4.15.4.1 Mydumper 简介

Mydumper 是一个 fork 项目，可以用于对 MySQL 或者 TiDB 进行逻辑备份，并针对 TiDB 的特性进行了优化。

Mydumper 包含在 tidb-enterprise-tools 安装包中，可在此下载。

4.15.4.1.1 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。

4.15.4.2 基本用法

4.15.4.2.1 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"`。

4.15.4.2.2 需要的权限

- SELECT
- RELOAD
- LOCK TABLES
- REPLICATION CLIENT

4.15.4.2.3 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
```

4.15.4.3 表内并发 Dump

4.15.4.3.1 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。

4.15.4.3.2 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。

4.15.4.3.3 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
```

4.15.4.3.4 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

4.15.4.3.5 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。

4.15.4.4 FAQ

4.15.4.4.1 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

4.15.4.4.2 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
```

4.15.4.4.3 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。

4.15.4.4.4 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。

4.15.4.4.5 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）。

4.15.4.4.6 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。

4.15.4.4.7 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。

4.15.4.4.8 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。

4.15.4.4.9 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';
```

4.15.4.4.10 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。

4.15.4.4.11 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。

4.15.4.4.12 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。

4.15.4.4.13 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。

4.15.4.4.14 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。

4.15.5 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。

4.15.5.1 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。

4.15.5.2 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。

4.15.5.3 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

4.15.5.4 使用方法

4.15.5.4.1 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。

4.15.5.4.2 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"）
```

4.15.5.4.3 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"
```

4.15.5.4.4 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
```

4.15.5.5 FAQ

4.15.5.5.1 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"
```

4.15.5.5.2 全量导入过程中遇到报错 `packet for query is too large. Try adjusting the 'max_allowed_packet' variable`

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

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

4.15.6 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。

4.15.6.1 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。

4.15.6.2 Syncer 架构

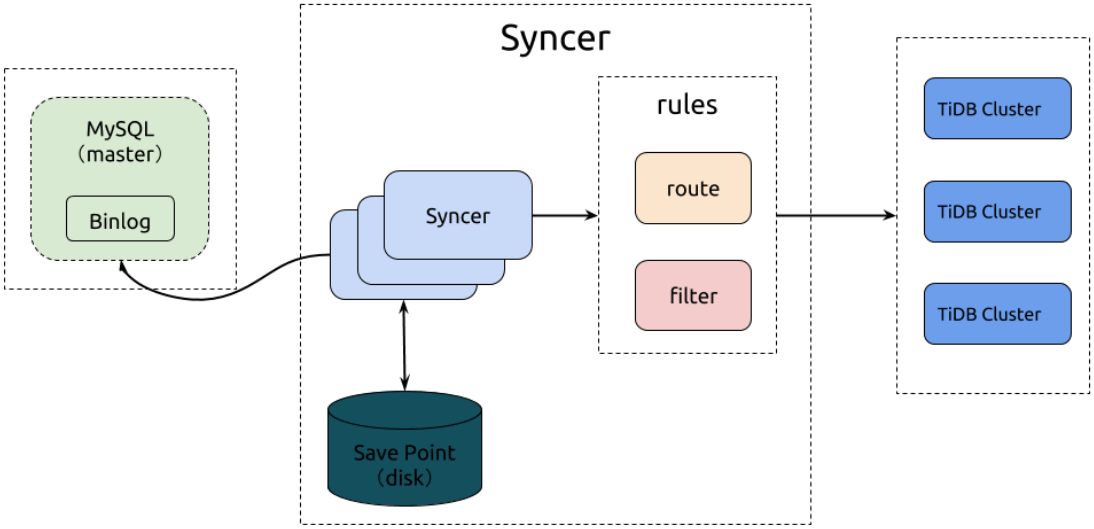


图 357: Syncer 架构

4.15.6.3 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。

4.15.6.4 Syncer 增量导入数据示例

使用前请详细阅读Syncer 同步前预检查。

4.15.6.4.1 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。

4.15.6.4.2 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)
```

4.15.6.4.3 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。

4.15.6.5 Syncer 配置说明

4.15.6.5.1 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_.*"
```

4.15.6.5.2 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。

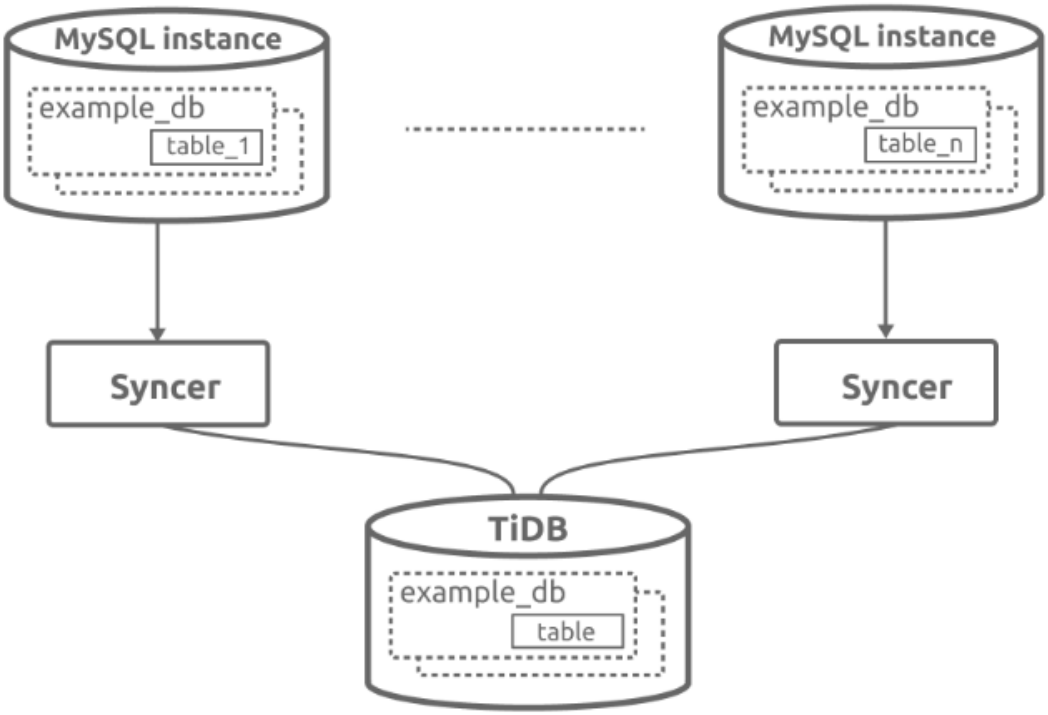


图 358: sharding

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"
```

4.15.6.5.3 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。

4.15.6.6 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：

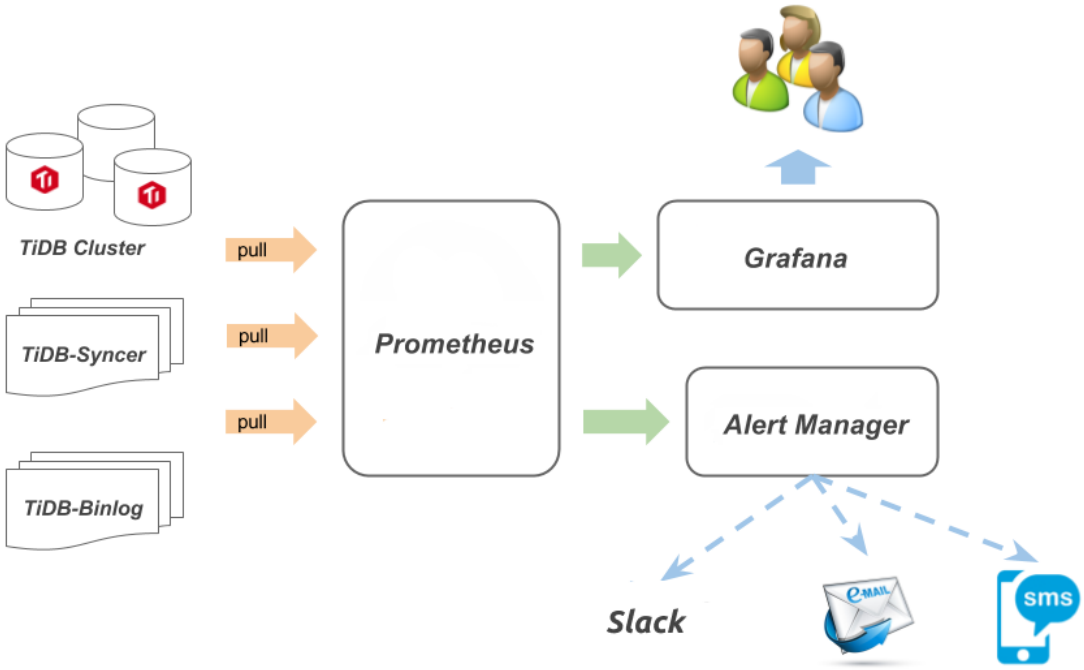


图 359: monitor_scheme

4.15.6.6.1 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 }}",
}
```

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

4.15.6.6.2 Grafana Syncer metrics 说明

title: binlog events

- metrics: `rate(syncer_binlog_event_count[1m])`
- info: 统计 Syncer 每秒已经收到的 binlog 个数

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

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

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

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

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

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

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

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

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

4.15.7 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。

4.15.7.1 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。

4.15.7.2 基本功能

本节介绍 DM 工具的核心功能模块。

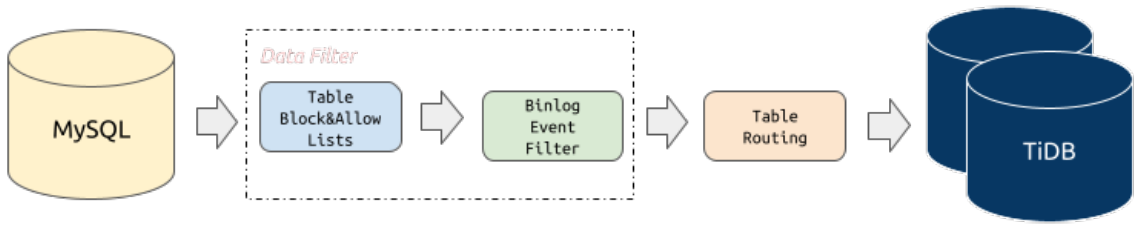


图 360: DM Core Features

4.15.7.2.1 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。

4.15.7.2.2 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。

4.15.7.2.3 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。

4.15.7.3 高级功能

4.15.7.3.1 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。

4.15.7.3.2 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。

4.15.7.3.3 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。

4.15.7.4 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。

4.15.8 TiDB Lightning

4.15.8.1 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：一是大量新数据的快速导入；二是全量备份数据的恢复。目前，Lightning 支持 Mydumper 或 CSV 输出格式的数据源。你可以在以下两种场景下使用 Lightning：

- 迅速导入大量新数据。
- 恢复所有备份数据。

4.15.8.1.1 TiDB Lightning 整体架构

TiDB Lightning 主要包含两个部分：

- `tidb-lightning`（“前端”）：主要完成适配工作，通过读取数据源，在下游 TiDB 集群建表、将数据转换成键值对（KV 对）发送到 `tikv-importer`、检查数据完整性等。
- `tikv-importer`（“后端”）：主要完成将数据导入 TiKV 集群的工作，对 `tidb-lightning` 写入的键值对进行缓存、排序、切分操作并导入到 TiKV 集群。

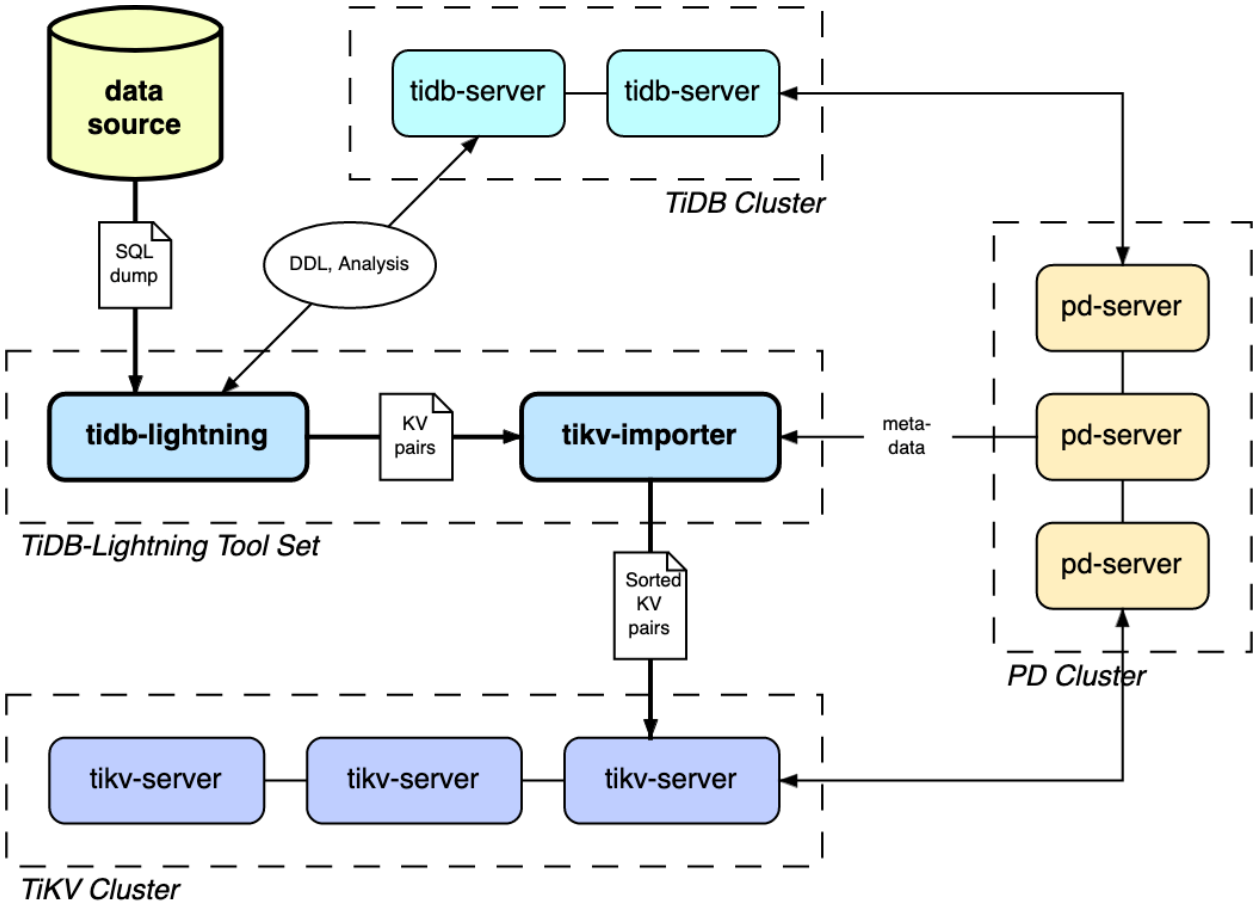


图 361: TiDB Lightning 整体架构

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导入。
4. `tidb-lightning` 会通过 gRPC 让 `tikv-importer` 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，然后发送到 `tikv-importer` 里对应的引擎文件。
5. 当一个引擎文件数据写入完毕时，`tikv-importer` 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。

TiDB Lightning 还支持使用 TiDB-backend 作为后端导入数据。和 Loader 类似，使用 TiDB-backend 时，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 TiDB-backend。

4.15.8.2 TiDB Lightning 部署与执行

本文主要介绍 TiDB Lightning 使用 Importer-backend（默认）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。

如果你不希望影响 TiDB 集群的对外服务，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。

4.15.8.2.1 注意事项

在使用 TiDB Lightning 前，需注意以下事项：

- TiDB Lightning 运行后，TiDB 集群将无法正常对外提供服务。
- 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。

4.15.8.2.2 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，建议单独部署。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。

> 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”。
> - tikv-importer 将中间数据存储缓存到内存上以加速导入过程。占用内存大小可以通过 (max-open-engines × write-buffer-size × 2) + (num-import-jobs × region-split-size × 2) 计算得来。如果磁盘写入速度慢，缓存可能会带来更大的内存占用。

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。

4.15.8.2.3 导出数据

使用mydumper 从 MySQL 导出数据，如下：

```
./bin/mydumper -h 127.0.0.1 -P 3306 -u root -t 16 -F 256 -B test -T t1,t2 --skip-tz-utc -o /data/
    ↪ my_database/
```

其中：

- `-B test`：从 test 数据库导出。
- `-T t1,t2`：只导出 t1 和 t2 这两个表。
- `-t 16`：使用 16 个线程导出数据。
- `-F 256`：将每张表切分成多个文件，每个文件大小约为 256 MB。
- `--skip-tz-utc`：添加这个参数则会忽略掉 TiDB 与导数据的机器之间时区设置不一致的情况，禁止自动转换。

如果数据源是 CSV 文件，请参考CSV 支持获取配置信息。

4.15.8.2.4 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。

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TiDB 快速入门指南。

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"

[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 = "192.168.20.10:8287"

[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。完整配置请参考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 &
```

4.15.8.2.5 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。

4.15.8.3 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。

4.15.8.3.1 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。

TiDB Lightning 全局配置

```
###### tidb-lightning 全局配置

[lightning]
#### 用于拉取 web 界面和 Prometheus 监控项的 HTTP 端口。设置为 0 时为禁用状态。
status-addr = ':8289'

#### 切换为服务器模式并使用 web 界面
#### 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
```

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

[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 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 “tidb “
#### backend = "importer"
#### 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"

[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 = 65536 # Byte (默认为 64 KB)

#### （源数据文件）单个导入区块大小的最小值。
#### 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，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 GB 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

#### mydumper 本地源数据目录。
data-source-dir = "/data/my_database"
#### 如果 no-shcema = 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 Lightning 会报错。
####  - gb18030：表结构文件必须使用 GB-18030 编码，否则 Lightning 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"

#### 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 = "STRICT_TRANS_TABLES,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

#### 数据导入完成后，tidb-lightning 可以自动执行 Checksum、Compact 和 Analyze 操作。
#### 在生产环境中，建议这将些参数都设为 true。
#### 执行的顺序为：Checksum -> Compact -> Analyze。
[post-restore]
#### 如果设置为 true，会对所有表逐个执行 `ADMIN CHECKSUM TABLE <table>` 操作
#### 来验证数据的完整性。
checksum = true
#### 如果设置为 true，会在导入每张表后执行一次 level-1 Compact。
#### 默认值为 false。
level-1-compact = false
#### 如果设置为 true，会在导入过程结束时对整个 TiKV 集群执行一次 full Compact。
#### 默认值为 false。
compact = false
#### 如果设置为 true，会对所有表逐个执行 `ANALYZE TABLE <table>` 操作。
analyze = true

#### 设置周期性后台操作。
#### 支持的单位：h（时）、m（分）、s（秒）。
[cron]
#### 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"
```

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"

[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 = "192.168.20.10:8287"
#### gRPC 服务器的线程池大小。
grpc-concurrency = 16

[metric]
#### 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"]

[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
```

4.15.8.3.2 命令行参数

`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	读取数据的目录	`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` 或`tidb`	`tikv-importer.backend`
–log-file file	日志文件路径	`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`

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”。

`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）
–import-engine uuid	将 TiKV Importer 上关闭的引擎文件导入到 TiKV 集群

参数	描述
–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。

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
–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

4.15.8.4 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，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。

4.15.8.4.1 断点续传的启用与配置

```
[checkpoint]
#### 启用断点续传。
#### 导入时，Lightning 会记录当前进度。
#### 若 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
```

4.15.8.4.2 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。

4.15.8.4.3 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。

4.15.8.5 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。

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。

4.15.8.5.1 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/
#          ^~~~~~~~~~~~~~~~~~~~~~
```

- 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：

- Lightning：

```
[mydumper]
filter = ['foo*.*', 'bar*.*']
```

4.15.8.5.2 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 /。

4.15.8.5.3 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）
employees.departments	√	√	√	规则 3（接受）
else.departments		√	√	规则 3（接受）

注意：

在 TiDB 生态工具中，无论表库过滤如何设置，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

4.15.8.6 CSV 支持

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）。

4.15.8.6.1 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。

文件扩展名必须为 *.csv，即使文件的内容并非逗号分隔。

4.15.8.6.2 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。
- 首先在 TiDB 中直接创建空表，然后在 tidb-lightning.toml 中设置 [mydumper] no-schema = true。

4.15.8.6.3 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。

4.15.8.6.4 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|
```

4.15.8.7 TiDB Lightning TiDB-Backend

TiDB Lightning 的后端决定 `tidb-lightning` 将如何把将数据导入到目标集群中。目前，TiDB Lightning 支持 Importer-backend（默认）和 TiDB-backend 两种后端，两者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。
- 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	Importer-backend	TiDB-backend
速度	快 (~300 GB/小时)	慢 (~50 GB/小时)
资源使用率	高	低
导入时是否满足 ACID	否	是
目标表	必须为空	可以不为空

4.15.8.7.1 部署 TiDB-backend

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`。

4.15.8.7.2 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 ...`

4.15.8.7.3 从 Loader 迁移到 TiDB Lightning TiDB-backend

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
log-level = "info"
log-file = "loader.log"
#### Prometheus
status-addr = ":8272"
#### 线程数
pool-size = 16
```

```
[lightning]
#### 日志
level = "info"
```

```
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
status-port = 10080  # <- 必须有的参数
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]]
#### schema-pattern = "shard_db_*"
#### table-pattern = "shard_table_*"
#### target-schema = "shard_db"
#### target-table = "shard_table"
```

4.15.8.8 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。

4.15.8.8.1 TiDB Lightning Web 首页

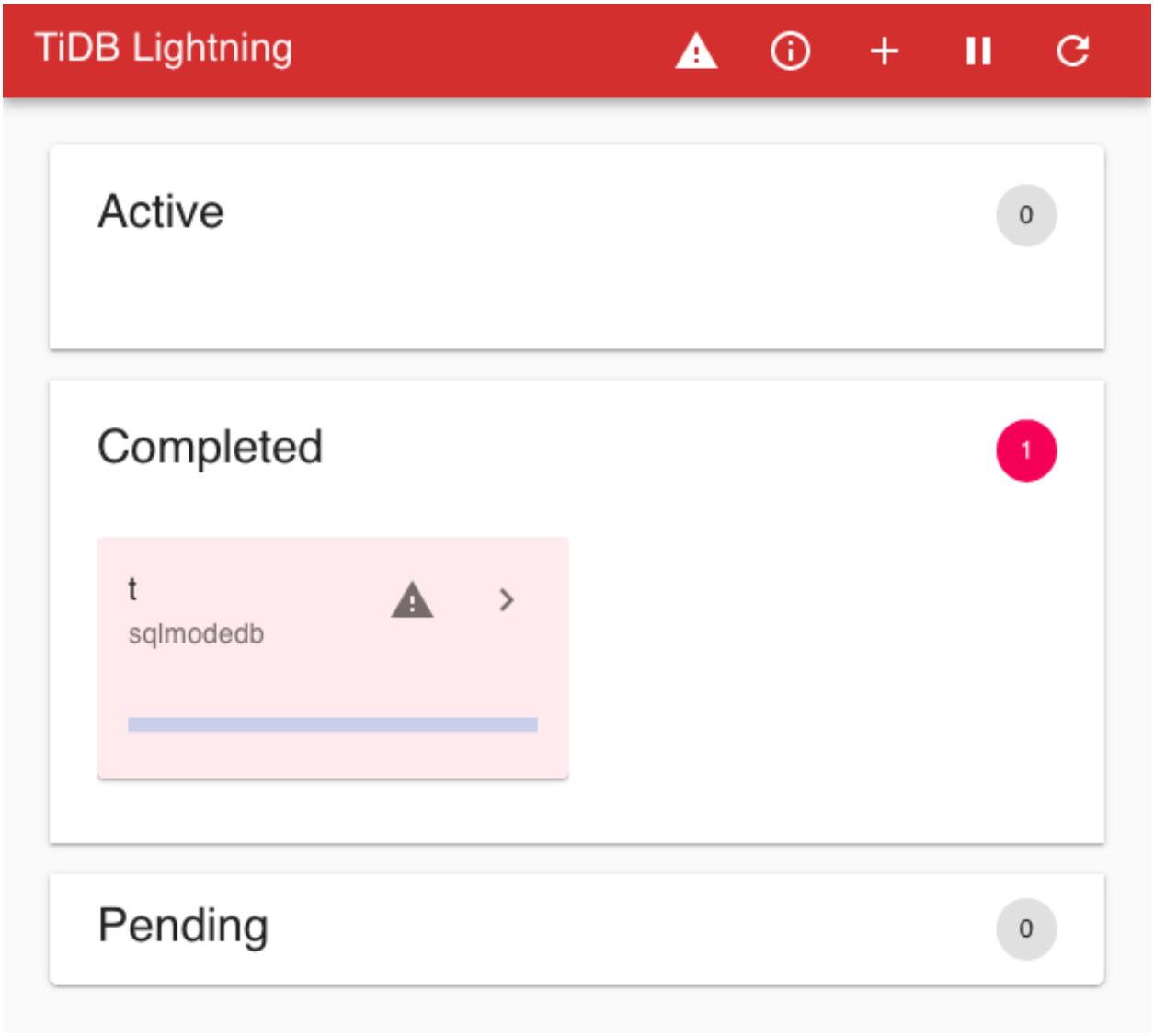


图 362: TiDB Lightning Web 首页

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。

4.15.8.8.2 提交任务

点击标题栏的 + 图标提交任务。

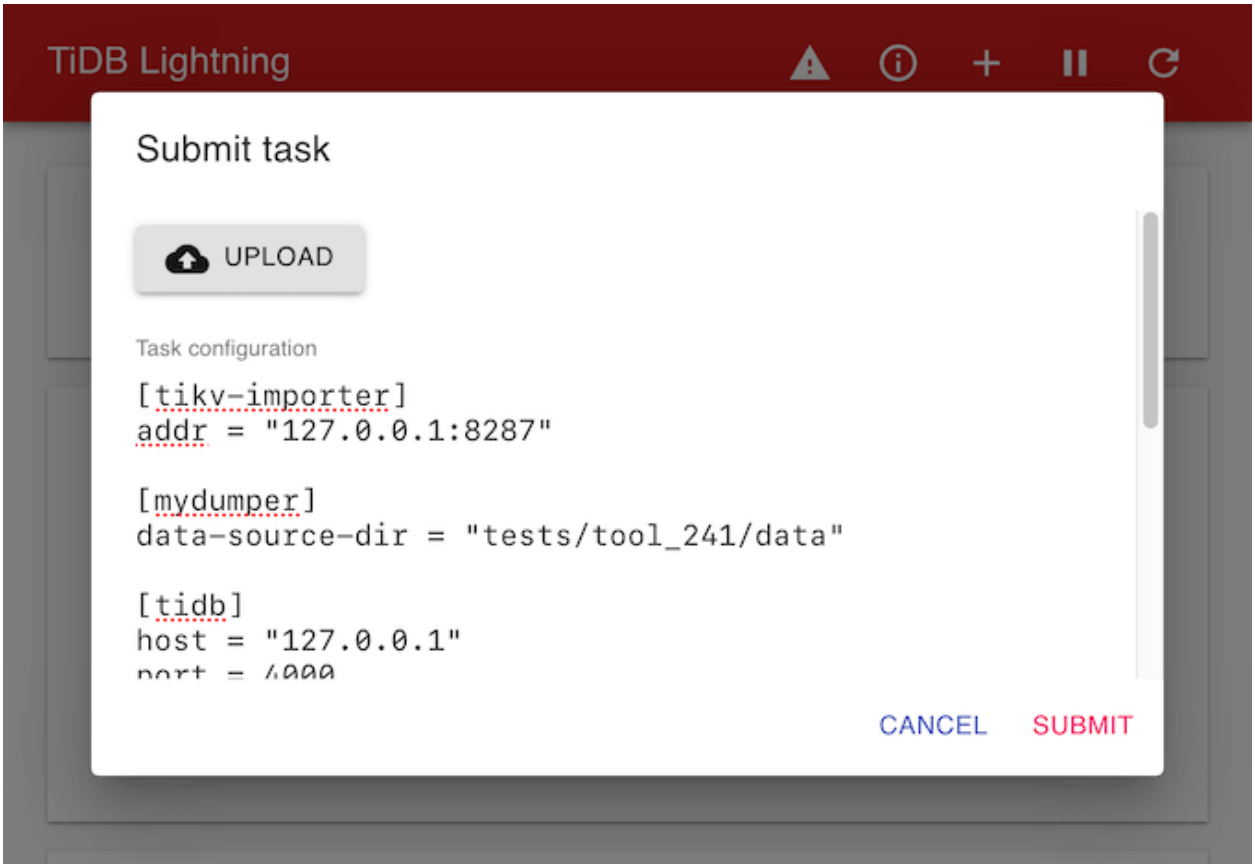


图 363: 提交任务对话框

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。

4.15.8.8.3 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 364: 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。

4.15.8.8.4 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 365: 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。

4.15.8.9 TiDB Lightning 监控告警

`tidb-lightning` 和 `tikv-importer` 都支持使用 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。

4.15.8.9.1 监控配置

- 如果是使用 TiDB Ansible 部署 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 Lightning，则参照以下步骤进行配置。

`tikv-importer`

`tikv-importer` v2.1 使用 Pushgateway 来推送监控指标。需要配置 `tikv-importer.toml` 来连接 Pushgateway：

```
[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 = ""
```

tidb-lightning

只要 Prometheus 能发现 tidb-lightning 的监控地址，就能收集监控指标。

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

...
```

要让 Prometheus 发现 Lightning，可以将地址直接写入其配置文件，例如：

```
...
scrape_configs:
  - job_name: 'tidb-lightning'
    static_configs:
      - targets: ['192.168.20.10:8289']
```

4.15.8.9.2 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。

第一行：速度面板

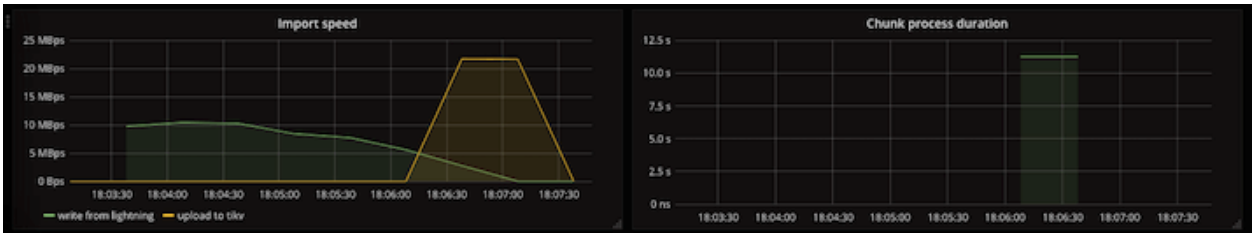


图 366: 第一行速度面板

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。

第二行：进度面板

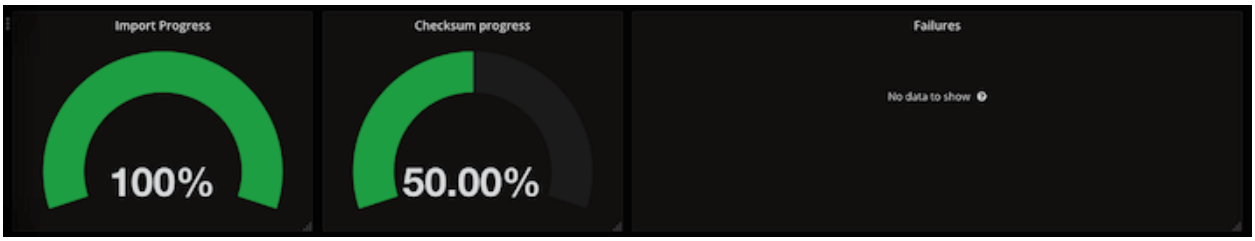


图 367: 第二行进度面板

面板名称	描述
Import progress	已编码的文件所占百分比

面板名称	描述
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

第三行：资源使用面板

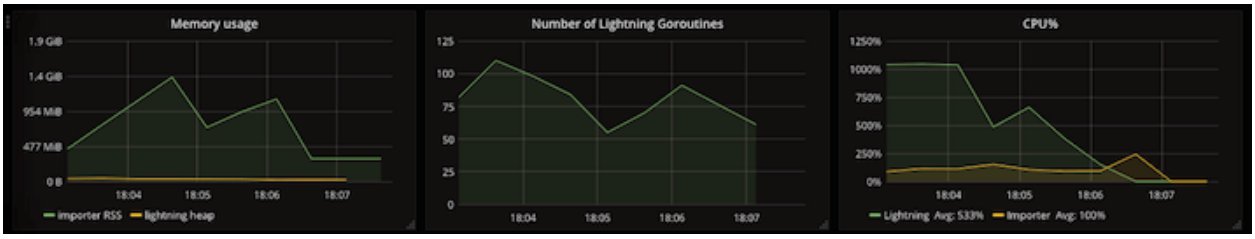


图 368: 第三行资源使用面板

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

第四行：配额使用面板

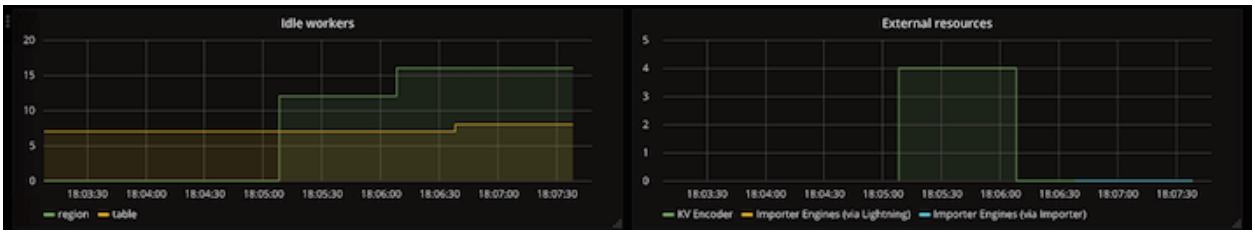


图 369: 第四行配额使用面板

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

第五行：读取速度面板

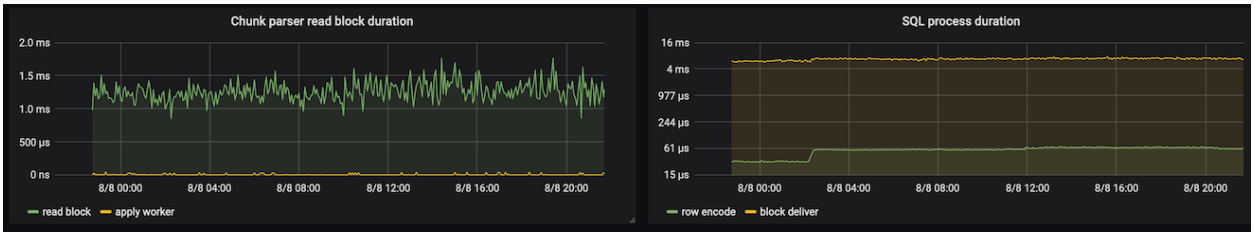


图 370: 第五行读取速度面板

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。

第六行：存储空间面板

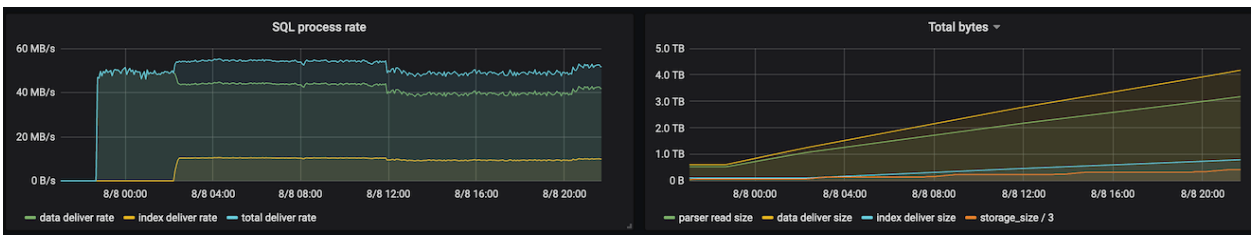


图 371: 第六行存储空间面板

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）

第七行：导入速度面板

图 372: 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

4.15.8.9.3 监控指标

本节将详细描述 `tikv-importer` 和 `tidb-lightning` 的监控指标。

`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 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。

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

4.15.8.10 TiDB Lightning 故障诊断

当 Lightning 遇到不可恢复的错误时便会异常退出，并在日志中记下错误原因。一般可在日志底部找到，也可以搜索 `[error]` 字符串找出中间发生的错误。本文主要描述一些常见的错误及其解决方法。

4.15.8.10.1 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 Lightning 读到的参数是多少。
2. 如果 Lightning 与其他服务（如 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），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 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。

原因 3：Lightning 版本太旧。

试试最新的版本吧！可能会有改善。

4.15.8.10.2 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 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。
    - 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 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。

4.15.8.10.3 Checkpoint for … has invalid status:（错误码）

原因：断点续传已启用。Lightning 或 Importer 之前发生了异常退出。为了防止数据意外损坏，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 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。

4.15.8.10.4 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 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
```

4.15.8.10.5 cannot guess encoding for input file, please convert to UTF-8 manually

原因：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。

4.15.8.10.6 [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 Lightning 与数据源时区一致。
    - 使用 TiDB Ansible 部署的话，修正 `[inventory.ini]` 下的 `timezone` 变量。

    ```
    # inventory.ini
    [all:vars]
    timezone = Asia/Shanghai
    ```

    - 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`。

4.15.8.10.7 `[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。

4.15.8.10.8 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。

4.15.8.11 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。

> 注意：
>
> 使用 TiDB Lightning 的过程中如遇错误，参考TiDB Lightning 故障诊断进行排查。

4.15.8.11.1 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。最低版本要求是 2.0.9，但建议使用最新的稳定版本 3.0。

4.15.8.11.2 TiDB Lightning 支持导入多个库吗？

支持。

4.15.8.11.3 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 TiBD Lightning 额外需要以下权限：

- INSERT
- DELETE

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。

4.15.8.11.4 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。

4.15.8.11.5 如何正确重启 TiDB Lightning？

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`。

4.15.8.11.6 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)
```

4.15.8.11.7 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. Mydumper 生成的 SQL dump
2. 储存在本地文件系统的CSV 文件

4.15.8.11.8 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。

4.15.8.11.9 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。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 = ""
...
```

4.15.8.11.10 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 Lightning 操作的表互不相同就可以。

4.15.8.11.11 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。

4.15.8.11.12 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。

4.15.8.11.13 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。

4.15.8.11.14 为什么用过 TiDB Lightning 之后，TiDB 集群变得又慢又耗 CPU？

如果 `tidb-lightning` 曾经异常退出，集群可能仍留在“导入模式”(import mode)，不适合在生产环境工作。此时需要强制切换回“普通模式”(normal mode)：

```
tidb-lightning-ctl --switch-mode=normal
```

4.15.8.11.15 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"
```

4.15.8.11.16 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

4.15.8.11.17 TiDB Lightning 使用过程中是否可以重启 TiKV Importer？

不能，Importer 会在内存中存储一些引擎文件，Importer 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 Importer 特有的信息丢失了。你可以在之后重启 Lightning。

4.15.8.11.18 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. 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。

4.15.8.12 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。

4.15.8.12.1 A

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。

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。

4.15.8.12.2 B

Backend

也称作 Back end（后端），用于接受 TiDB Lightning 解析结果。

详情参阅TiDB Lightning TiDB-backend。

4.15.8.12.3 C

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。

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，参考故障排查进行处理。

Chunk

指数据源中的单个文件。

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。

4.15.8.12.4 D

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）。

4.15.8.12.5 E

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。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。

4.15.8.12.6 F

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。

4.15.8.12.7 I

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。

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。

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。

4.15.8.12.8 K

KV pair

即 key-value pair（键值对）。

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。

4.15.8.12.9 L

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。

4.15.8.12.10 N

Normal mode

普通模式。未启用导入模式时的模式。

4.15.8.12.11 P

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。

4.15.8.12.12 R

Remote checksum

远程校验和。指导入 TiDB 后所计算的表的校验和。

4.15.8.12.13 S

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。

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。

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。

4.15.9 sync-diff-inspector

4.15.9.1 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

4.15.9.1.1 sync-diff-inspector 的使用

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。

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)

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-field = "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 = ""
```

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。

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。

4.15.9.2 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 = ""
```

4.15.9.3 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。

4.15.9.3.1 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：

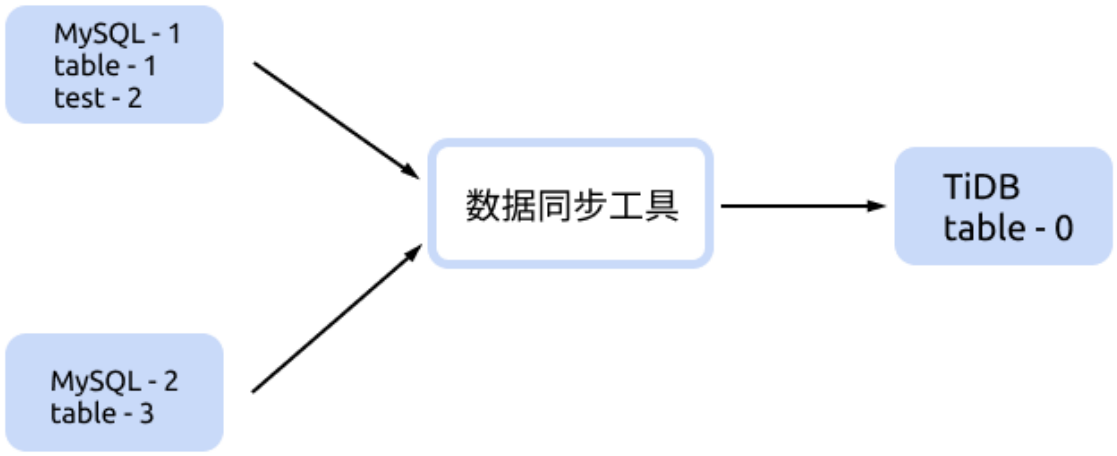


图 373: shard-table-sync-1

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"
```

4.15.9.3.2 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：

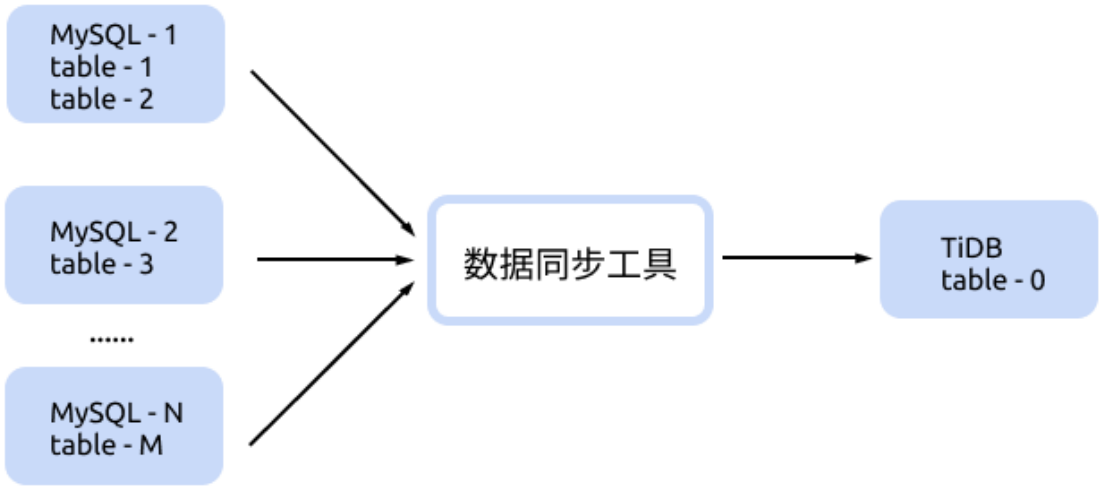


图 374: shard-table-sync-2

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"
```

4.15.9.4 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。

4.15.9.4.1 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。

4.15.9.4.2 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"
```

4.15.9.4.3 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。

4.15.10 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。

> **注意：**
>
> 建议使用的 Control 工具版本与集群版本保持一致。

4.15.10.1 源码编译

1. Go Version 1.13 以上
2. 在 PD 项目根目录使用 `make` 命令进行编译，生成 bin/pd-ctl

4.15.10.2 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 latest 版本的下载链接为 https://download.pingcap.org/tidb-latest-linux-amd64.tar.gz。也可以使用 latest 替代 {version} 来下载最新的未发布版本。

4.15.10.3 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"
```

4.15.10.4 命令行参数 (flags)

4.15.10.4.1 –cacert

- 指定 PEM 格式的受信任 CA 的证书文件路径
- 默认值: “”

4.15.10.4.2 –cert

- 指定 PEM 格式的 SSL 证书文件路径
- 默认值: “”

4.15.10.4.3 --detach,-d

- 使用单命令行模式（不进入 readline）
- 默认值：true

4.15.10.4.4 --interact,-i

- 使用交互模式（进入 readline）
- 默认值：false

4.15.10.4.5 –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: “”

4.15.10.4.6 --pd,-u

- 指定 PD 的地址
- 默认地址：`http://127.0.0.1:2379`
- 环境变量：`PD_ADDR`

4.15.10.4.7 –version,-V

- 打印版本信息并退出
- 默认值: false

4.15.10.5 命令 (command)

4.15.10.5.1 cluster

用于显示集群基本信息。

示例：

```
>> cluster
```

```
{
  "id": 6493707687106161130,
  "max_peer_count": 3
}
```

4.15.10.5.2 `config [show | set <option> <value>]`

用于显示或调整配置信息。示例如下。

显示 scheduler 的相关 config 信息：

```
>> config show
```

```
{
  "max-snapshot-count": 3,
  "max-pending-peer-count": 16,
  "max-merge-region-size": 50,
  "max-merge-region-keys": 200000,
  "split-merge-interval": "1h",
  "patrol-region-interval": "100ms",
  "max-store-down-time": "1h0m0s",
  "leader-schedule-limit": 4,
  "region-schedule-limit": 4,
  "replica-schedule-limit":8,
  "merge-schedule-limit": 8,
  "tolerant-size-ratio": 5,
  "low-space-ratio": 0.8,
  "high-space-ratio": 0.6,
  "disable-raft-learner": "false",
  "disable-remove-down-replica": "false",
  "disable-replace-offline-replica": "false",
  "disable-make-up-replica": "false",
  "disable-remove-extra-replica": "false",
  "disable-location-replacement": "false",
  "disable-namespace-relocation": "false",
  "schedulers-v2": [
    {
      "type": "balance-region",
      "args": null
    },
    {
      "type": "balance-leader",
      "args": null
    },
    {
      "type": "hot-region",
      "args": null
    }
  ]
}
```

显示所有的 config 信息：

```
>> config show all
```

显示名为 ts1 的 namespace 的相关 config 信息：

```
>> config show namespace ts1
```

```
{
  "leader-schedule-limit": 4,
  "region-schedule-limit": 4,
  "replica-schedule-limit": 8,
  "max-replicas": 3,
}
```

显示 replication 的相关 config 信息：

```
>> config show replication
```

```
{
  "max-replicas": 3,
  "location-labels": ""
}
```

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
```

```
"2.0.0"
```

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
```

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。这个值主要影响 Region balance 的速度，值越大调度得越快，设置为 0 则关闭调度。Region 调度的开销较大，所以这个值不宜调得太大。

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
```

以上对配置的修改是全局性的，还可以通过对不同 namespace 的配置，进行细化调整。当 namespace 未设置相应配置时，使用全局配置。注：namespace 的配置只支持对 leader-schedule-limit，region-schedule-limit，replica-schedule-limit，max-replicas 的调整，否则不生效。

设置名为 ts1 的 namespace 最多同时进行 4 个 leader 调度：

```
>> config set namespace ts1 leader-schedule-limit 4
```

设置名为 ts2 的 namespace 最多同时进行 2 个 Region 调度：

```
>> config set namespace ts2 region-schedule-limit 2
```

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
```

disable-raft-learner 用于关闭 raft learner 功能。默认配置下 PD 在添加副本时会使用 raft learner 来降低宕机或网络故障带来的不可用风险。

关闭 raft learner 功能：

```
config set disable-raft-learner true
```

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
```

disable-remove-down-replica 用于关闭自动删除 DownReplica 的特性。当设置为 true 时，PD 不会自动清理宕机状态的副本。

disable-replace-offline-replica 用于关闭迁移 OfflineReplica 的特性。当设置为 true 时，PD 不会迁移下线状态的副本。

disable-make-up-replica 用于关闭补充副本的特性。当设置为 true 时，PD 不会为副本数不足的 Region 补充副本。

disable-remove-extra-replica 用于关闭删除多余副本的特性。当设置为 true 时，PD 不会为副本数过多的 Region 删除多余副本。

`disable-location-replacement` 用于关闭隔离级别检查。当设置为 true 时，PD 不会通过调度来提升 Region 副本的隔离级别。

`disable-namespace-relocation` 用于关闭 Region 的 namespace 调度。当设置为 true 时，PD 不会把 Region 调度到它所属的 Store 上。

4.15.10.5.3 `config delete namespace <name> [<option>]`

用于删除 namespace 的配置信息。

示例：

在对 namespace 相关配置进行设置后，若想让该 namespace 继续使用全局配置，可删除该 namespace 的配置信息，之后便使用全局配置。

删除名为 ts1 的 namespace 的相关配置：

```
>> config delete namespace ts1
```

若只想让 namespace 中的某项配置使用全局配置而不影响其他配置，则可使用如下命令：

删除名为 ts2 的 namespace 的 region-schedule-limit 配置：

```
>> config delete namespace region-schedule-limit ts2
```

4.15.10.5.4 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
  }
  ......
]
```

4.15.10.5.5 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
```

4.15.10.5.6 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
```

4.15.10.5.7 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
```

```
{
  "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",
  "addr": "http://192.168.199.229:2379",
  "id": 9724873857558226554
}
```

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
```

```
......
```

4.15.10.5.8 operator [show | add | remove]

用于显示和控制调度操作，或者对 Region 进行分裂或合并。

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 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
```

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。

4.15.10.5.9 ping

用于显示ping PD 所需要花费的时间

示例：

```
>> ping
```

```
time: 43.12698ms
```

4.15.10.5.10 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
```

```
{
  "region": {
      "id": 2,
      ......
  }
  "leader": {
      ......
  }
}
```

4.15.10.5.11 region key [--format=raw|encode] <key>

用于查询某个 key 在哪个 Region 上，支持 raw 和 encoding 格式。使用 encoding 格式时，key 需要使用单引号。

Raw 格式（默认）示例：

```
>> region key 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\372'
```

```
{
  "region": {
    "id": 2,
    ......
  }
}
```

4.15.10.5.12 region sibling <region_id>

用于查询某个 Region 相邻的 Region。

示例：

```
>> region sibling 2
```

```
{
  "count": 2,
  "regions": [......],
}
```

4.15.10.5.13 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": [......],
}
```

4.15.10.5.14 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": [......],
}
```

4.15.10.5.15 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": [......],
}
```

4.15.10.5.16 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": [......],
}
```

4.15.10.5.17 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": [......],
}
```

4.15.10.5.18 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": [......],
}
```

4.15.10.5.19 region check [miss-peer | extra-peer | down-peer | pending-peer | incorrect-ns]

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
- incorrect-ns：有副本不符合 namespace 约束的 Region

示例：

```
>> region check miss-peer
```

```
{
  "count": 2,
  "regions": [......],
}
```

4.15.10.5.20 scheduler [show | add | remove]

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 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 scheduler 删掉
```

4.15.10.5.21 store [delete | label | weight] <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
```

4.15.10.5.22 table_ns [create | add | remove | set_store | rm_store | set_meta | rm_meta]

用于显示 table 的 namespace 的相关信息

示例：

```
>> table_ns add ts1 1            // 将 table id 为 1 的 table 添加到名为 ts1 的 namespace
>> table_ns create ts1           // 添加名为 ts1 的 namespace
>> table_ns remove ts1 1         // 将 table id 为 1 的 table 从名为 ts1 的 namespace 中移除
>> table_ns rm_meta ts1          // 将 meta 信息从名为 ts1 的 namespace 中移除
>> table_ns rm_store 1 ts1       // 将 store id 为 1 的 table 从名为 ts1 的 namespace 中移除
>> table_ns set_meta ts1         // 将 meta 信息添加到名为 ts1 的 namespace
>> table_ns set_store 1 ts1      // 将 store id 为 1 的 table 添加到名为 ts1 的 namespace
```

4.15.10.5.23 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
```

4.15.10.6 jq 格式化 json 输出示例

4.15.10.6.1 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"}
...
```

4.15.10.6.2 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"}
...
```

4.15.10.6.3 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]}
...
```

4.15.10.6.4 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]}
```

4.15.10.6.5 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]}
...
```

4.15.10.6.6 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]}
...
```

4.15.11 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。PD Recover 会随 tidb-ansible 一起下载，位于 resource/bin/pd-recover。

4.15.11.1 快速开始

4.15.11.1.1 获取 Cluster ID

一般在 PD、TiKV 或 TiDB 的日志中都可以获取 Cluster ID。可以从中控机使用 `ansible ad-hoc`，也可以直接去服务器上查看日志。

从 PD 日志获取 [info] Cluster ID（推荐）

使用以下命令，从 PD 日志中获取 [info] Cluster ID：

```
ansible -i inventory.ini pd_servers -m shell -a 'cat {{deploy_dir}}/log/pd.log | grep "init
    ↪ cluster id" | head -10'
```

```
10.0.1.13 | CHANGED | rc=0 >>
[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 [info] cluster id

使用以下命令，从 TiDB 日志中获取 [info] cluster id：

```
ansible -i inventory.ini tidb_servers -m shell -a 'cat {{deploy_dir}}/log/tidb*.log | grep "init
    ↪ cluster id" | head -10'
```

```
10.0.1.15 | CHANGED | rc=0 >>
2019/10/14 19:23:04.688 client.go:161: [info] [pd] init cluster id 6747551640615446306
......
```

从 TiKV 日志获取 [info] PD cluster

使用以下命令，从 TiKV 日志中获取 [info] PD cluster：

```
ansible -i inventory.ini tikv_servers -m shell -a 'cat {{deploy_dir}}/log/tikv* | grep "PD
    ↪ cluster" | head -10'
```

```
10.0.1.15 | CHANGED | rc=0 >>
[2019/10/14 07:06:35.278 +00:00] [INFO] [tikv-server.rs:464] ["connect to PD cluster
    ↪ 6747551640615446306"]
......
```

4.15.11.1.2 获取 Alloc ID (TiKV StoreID)

在指定 `alloc-id` 时需指定一个比当前最大的 `Alloc ID` 更大的值。可以从中控机使用 `ansible ad-hoc`，也可以直接去服务器上翻日志。

从 PD 日志获取 [info] allocates id（推荐）

使用以下命令，从 PD 日志中获取 [info] allocates id：

```
ansible -i inventory.ini pd_servers -m shell -a 'cat {{deploy_dir}}/log/pd* | grep "allocates" |
    ↪ head -10'
```

```
10.0.1.13 | CHANGED | rc=0 >>
[2019/10/15 03:15:05.824 +00:00] [INFO] [id.go:91] ["idAllocator allocates a new id"] [alloc-id
    ↪ =3000]
[2019/10/15 08:55:01.275 +00:00] [INFO] [id.go:91] ["idAllocator allocates a new id"] [alloc-id
    ↪ =4000]
......
```

或者也可以从 TiKV 的日志中获取。

从 TiKV 日志获取 [info] alloc store id

使用以下命令，从 TiKV 日志获取 [info] alloc store id：

```
ansible -i inventory.ini tikv_servers -m shell -a 'cat {{deploy_dir}}/log/tikv* | grep "alloc
    ↪ store" | head -10'
```

```
10.0.1.13 | CHANGED | rc=0 >>
[2019/10/14 07:06:35.516 +00:00] [INFO] [node.rs:229] ["alloc store id 4 "]

10.0.1.14 | CHANGED | rc=0 >>
[2019/10/14 07:06:35.734 +00:00] [INFO] [node.rs:229] ["alloc store id 5 "]

10.0.1.15 | CHANGED | rc=0 >>
[2019/10/14 07:06:35.418 +00:00] [INFO] [node.rs:229] ["alloc store id 1 "]

10.0.1.21 | CHANGED | rc=0 >>
[2019/10/15 03:15:05.826 +00:00] [INFO] [node.rs:229] ["alloc store id 2001 "]

10.0.1.20 | CHANGED | rc=0 >>
[2019/10/15 03:15:05.987 +00:00] [INFO] [node.rs:229] ["alloc store id 2002 "]
```

4.15.11.1.3 部署一套新的 PD 集群

```
ansible-playbook bootstrap.yml --tags=pd
```

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

```
ansible-playbook start.yml --tags=pd
```

旧集群可以通过删除 `data.pd` 目录后，重新启动 PD 服务。

4.15.11.1.4 使用 pd-recover

```
./pd-recover -endpoints http://10.0.1.13:2379 -cluster-id 6747551640615446306 -alloc-id 10000
```

4.15.11.1.5 重启 PD 集群

```
ansible-playbook rolling_update.yml --tags=pd
```

4.15.11.1.6 重启 TiDB 或 TiKV

```
ansible-playbook rolling_update.yml --tags=tidb,tikv
```

4.15.11.2 常见问题

4.15.11.2.1 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。

4.15.11.2.2 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。

4.15.12 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。

> **注意：**
>
> 建议使用的 Control 工具版本与集群版本保持一致。

TiUP 是晚于 `tidb-ansible` 推出的部署工具，使用方式更加简化，`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 tikv
```

```
Starting component `ctl`: ~/.tiup/components/ctl/v4.0.0-rc.2/ctl tikv
TiKV Control (tikv-ctl)
Release Version:   4.0.0-rc.2
Edition:           Community
Git Commit Hash:   2fdb2804bf8ffaab4b18c4996970e19906296497
Git Commit Branch: heads/refs/tags/v4.0.0-rc.2
UTC Build Time:    2020-05-15 11:58:49
Rust Version:      rustc 1.42.0-nightly (0de96d37f 2019-12-19)
Enable Features:   jemalloc portable sse protobuf-codec
Profile:           dist_release
```

你可以在 `tiup ctl tikv` 后面再接上相应的参数与子命令。

4.15.12.1 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。

4.15.12.2 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>`。

4.15.12.2.1 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})
```

4.15.12.2.2 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
```

4.15.12.2.3 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"
```

4.15.12.2.4 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。

4.15.12.2.5 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。

4.15.12.2.6 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。

4.15.12.2.7 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。

```
tikv-ctl --host 127.0.0.1:20160 compact -d kv
```

```
success!
```

4.15.12.2.8 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。

4.15.12.2.9 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。

4.15.12.2.10 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。

4.15.12.2.11 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。

4.15.12.2.12 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。

4.15.12.2.13 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
```

4.15.12.2.14 动态修改 TiKV 的 RocksDB 相关配置

使用 `modify-tikv-config` 命令可以动态修改配置参数，暂时仅支持对于 RocksDB 相关参数的动态更改。

- `-m` 用于指定要修改的模块，有 `storage`、`kvdb` 和 `raftdb` 三个值可以选择。
- `-n` 用于指定配置名。配置名可以参考 TiKV 配置模版中 `[storage]`、`[rocksdb]` 和 `[raftdb]` 下的参数，分别对应 `storage`、`kvdb` 和 `raftdb`。同时，还可以通过 `default|write|lock + . +` 参数名 的形式来指定的不同 CF 的配置。对于 `kvdb` 有 `default`、`write` 和 `lock` 可以选择，对于 `raftdb` 仅有 `default` 可以选择。
- `-v` 用于指定配置值。

设置 `shared block cache` 的大小：

```
tikv-ctl modify-tikv-config -m storage -n block_cache.capacity -v 10GB
```

```
success!
```

当禁用 `shared block cache` 时，为 `write` CF 设置 `block cache size`：

```
tikv-ctl modify-tikv-config -m kvdb -n write.block_cache_size -v 256MB
```

```
success!
```

```
tikv-ctl modify-tikv-config -m kvdb -n max_background_jobs -v 8
```

```
success!
```

```
tikv-ctl modify-tikv-config -m raftdb -n default.disable_auto_compactions -v true
```

```
success!
```

4.15.12.2.15 强制 Region 从多副本失败状态恢复服务

`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`。

```
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：

- 该命令只支持本地模式。在运行成功后，会打印 `success!`。
- 一般来说，您需要为指定 Region 的 peers 所在的每个 store 运行此命令。
- 如果使用 `--all-regions`，通常需要在集群剩余所有健康的 store 上执行此命令。

4.15.12.2.16 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。

4.15.12.2.17 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。

4.15.13 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。

4.15.13.1 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。

4.15.13.1.1 通过 TiDB Ansible 安装

对于使用TiDB Ansible 部署的 TiDB 集群，在 TiDB 的安装路径下可以找到 TiDB Control 的二进制程序 `tidb-ctl`。

4.15.13.1.2 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。

4.15.13.2 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

4.15.13.2.1 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。

4.15.13.2.2 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。

4.15.13.2.3 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。

4.15.13.2.4 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
```

4.15.13.2.5 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}
```

4.15.13.2.6 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"
```

4.15.13.2.7 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。

4.15.13.2.8 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: (74800000000000002f5f69, 74800000000000002f5f72)
    index c2: (74800000000000002f5f698000000000000001, 748000000000000002
        ↪ f5f698000000000000002)
    index c3: (74800000000000002f5f698000000000000002, 748000000000000002
        ↪ f5f698000000000000003)
    index c4: (74800000000000002f5f698000000000000003, 748000000000000002
        ↪ f5f698000000000000004)
  table rows: (74800000000000002f5f72, 748000000000000030)
```

4.15.14 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。

4.15.14.1 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。

4.15.14.2 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。

以下表格中也提供了 Kafka 版本的 TiDB Binlog 下载链接。

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 ↪

安装包	操作系统	架构	SHA256 校验和
https ↪ ://download ↪ .pingcap ↪ .org ↪ /tidb ↪ -binlog ↪ -kafka ↪ -linux ↪ -amd64 ↪ .tar ↪ .gz	Linux	amd64	https ↪ ://download ↪ .pingcap ↪ .org ↪ /tidb ↪ -binlog ↪ -kafka ↪ -linux ↪ -amd64 ↪ .sha256 ↪
(Kafka 版本的 TiDB Bin-log)			

注意：

下载链接中的 {version} 为 TiDB 的版本号。例如，v3.0.5 版本的下载链接为 https://download.pingcap.org/tidb-v3.0.5-linux-amd64.tar.gz。

4.15.14.3 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，v3.0.5 版本的下载链接为 https://download.pingcap.org/tidb-toolkit-v3.0.5-linux-amd64.tar.gz。

4.15.14.4 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，`v3.1.0-beta` 版本的下载链接为 `https://download.pingcap.org/tidb-toolkit-v3.1.0-beta-linux-amd64.tar.gz`。

4.15.14.5 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，v1.0.1 版本的下载链接为 https://download.pingcap.org/dm-v1.0.1-linux-amd64.tar.gz。可以通过 DM Release 查看当前已发布版本。

4.15.14.6 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

5 TiDB in Kubernetes

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私有部署的 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

6 常见问题 (FAQ)

6.1 FAQ

6.1.1 一、TiDB 介绍、架构、原理

6.1.1.1 1.1 TiDB 介绍及整体架构

6.1.1.1.1 1.1.1 TiDB 整体架构

TiDB 简介

6.1.1.1.2 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。

6.1.1.1.3 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。

6.1.1.1.4 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。

6.1.1.1.5 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。

6.1.1.1.6 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。

使用 MySQL 8.0 客户端时，如果遇到无法登陆的问题，可以尝试指定 `default-auth` 和 `default-character-set` 参数：

```
mysql -h 127.0.0.1 -u root -P 4000 --default-auth=mysql_native_password --default-character-set=
    ↪ utf8
```

无法登陆的原因是 MySQL 8.0 会更改了 MySQL 5.7 默认的密码加密方式，所以需要添加以上参数指定使用旧的加密方式。

6.1.1.1.7 1.1.7 TiDB 支持分布式事务吗？

TiDB 天然具备高可用特性，TiDB、TiKV、PD 这三个组件都能容忍部分实例失效，不影响整个集群的可用性。具体见TiDB 高可用性。

6.1.1.1.8 1.1.8 TiDB 数据是强一致的吗？

TiDB 实现了快照隔离 (Snapshot Isolation) 级别的一致性。为与 MySQL 保持一致，又称其为 “可重复读”。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。

6.1.1.1.9 1.1.9 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。

6.1.1.1.10 1.1.10 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。

6.1.1.1.11 1.1.11 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。

6.1.1.1.12 1.1.12 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。

6.1.1.1.13 1.1.13 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。

6.1.1.1.14 1.1.14 TiDB 对哪些 MySQL variables 兼容？

详细可参考系统变量。

6.1.1.1.15 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)
```

6.1.1.1.16 1.1.15 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。

6.1.1.1.17 1.1.16 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。

6.1.1.1.18 1.1.17 TiDB 用户名长度限制？

在 TiDB 中用户名最长为 32 字符。

6.1.1.1.19 1.1.18 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。

6.1.1.1.20 1.1.19 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。

6.1.1.1.21 1.1.20 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：

1）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。

2）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。

6.1.1.1.22 1.1.21 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。

6.1.1.1.23 1.1.22 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。

6.1.1.1.24 1.1.23 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。

6.1.1.1.25 1.1.23 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 TiDB Ansible 部署集群时，同样可以通过 conf/tidb.yml 配置文件中的 server-version 来设置合适的版本号，以避免出现安全漏洞扫描不通过的问题。

6.1.1.1.26 1.1.24 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：

第一步：客户端连接服务器；

第二步：服务器发送随机字符串 challenge 给客户端；

第三步：客户端发送 username + response 给服务器；

第四步：服务器验证 response。

6.1.1.2 1.2 TiDB 原理

6.1.1.2.1 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

6.1.1.2.2 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

6.1.1.2.3 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

6.1.2 二、安装部署升级

6.1.2.1 2.1 环境准备

6.1.2.1.1 2.1.1 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
CentOS	7.3 及以上
Oracle Enterprise Linux	7.3 及以上

2.1.1.1 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。

6.1.2.1.2 2.1.2 硬件要求

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：

2.1.2.1 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

2.1.2.2 线上环境

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
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

2.1.2.3 2 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。2 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。

2.1.2.4 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。

2.1.2.5 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 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。

6.1.2.2 2.2 安装部署

6.1.2.2.1 2.2.1 TiDB Ansible 部署方式（强烈推荐）

详细可参考使用 TiDB Ansible 部署 TiDB 集群。

2.2.1.1 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。

2.2.1.2 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。

2.2.1.3 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。

2.2.1.4 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

2.2.1.5 inventory.ini 变量参数解读

变量	含义
cluster_name	集群名称，可调整

变量	含义
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
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

变量	含义
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
enable_ntpd	检测部署目标机器 NTP 服务，默认为 True，请勿关闭

变量	含义
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
enable_binlog	是否部署 pump 并开启 bin-log，默认为 False，依赖 Kafka 集群，参见 zookeeper_addrs 变量
zookeeper_addrs	Kafka 集群的 zookeeper 地址

变量	含义
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
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。

6.1.2.2.2 2.2.2 TiDB 离线 Ansible 部署方案

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。

6.1.2.2.3 2.2.3 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。

6.1.2.2.4 2.2.4 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。

6.1.2.2.5 2.2.5 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。

6.1.2.2.6 2.2.6 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。

6.1.2.2.7 2.2.7 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
```

6.1.2.2.8 2.2.8 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。

6.1.2.3 2.3 升级

6.1.2.3.1 2.3.1 如何使用 TiDB Ansible 滚动升级？

滚动升级 TiKV 节点 (只升级单独服务)

```
ansible-playbook rolling_update.yml --tags=tikv
```

滚动升级所有服务

```
ansible-playbook rolling_update.yml
```

6.1.2.3.2 2.3.2 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/Drainer 服务，建议先停止 Drainer 后滚动升级 (升级 TiDB 时会升级 Pump)。

6.1.2.3.3 2.3.3 Binary 如何升级？

Binary 不是我们建议的安装方式，对升级支持也不友好，建议换成 TiDB Ansible 部署。

6.1.2.3.4 2.3.4 一般升级选择升级 TiKV 还是所有组件都升级？

常规需要一起升，因为整个版本都是一起测试的，单独升级只限当发生一个紧急故障时，需要单独对一个有问题的角色做升级。

6.1.2.3.5 2.3.5 启动集群或者升级集群过程中出现 “Timeout when waiting for search string 200 OK” 是什么原因？如何处理？

可能有以下几种原因：进程没有正常启动；端口被占用；进程没有正常停掉；停掉集群的情况下使用 rolling_update.yml 来升级集群 (操作错误)。

处理方式：登录到相应节点查看进程或者端口的状态；纠正错误的操作步骤。

6.1.3 三、集群管理

6.1.3.1 3.1 集群日常管理

6.1.3.1.1 3.1.1 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

6.1.3.1.2 3.1.2 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
```

6.1.3.1.3 3.1.3 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。

6.1.3.1.4 3.1.4 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。

6.1.3.1.5 3.1.5 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。

6.1.3.1.6 3.1.6 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。

6.1.3.1.7 3.1.7 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。

6.1.3.1.8 3.1.8 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。

6.1.3.1.9 3.1.9 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。

6.1.3.1.10 3.1.10 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`

6.1.3.1.11 3.1.11 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 TiDB Ansible 进行安装，TiDB Ansible 的版本也会随着 TiDB 的版本发布进行更新，因此建议用户在安装升级新版本的时候使用最新的 TiDB Ansible 安装包版本进行安装。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

6.1.3.1.12 3.1.12 有没有图形化部署 TiDB 的工具？

暂时没有。

6.1.3.1.13 3.1.13 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。

6.1.3.1.14 3.1.14 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。

6.1.3.1.15 3.1.15 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。

6.1.3.1.16 3.1.16 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)。

6.1.3.1.17 3.1.17 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。

6.1.3.2 3.2 PD 管理

6.1.3.2.1 3.2.1 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。

6.1.3.2.2 3.2.2 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。

6.1.3.2.3 3.2.3 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。

6.1.3.2.4 3.2.4 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。

6.1.3.2.5 3.2.5 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。

6.1.3.2.6 3.2.6 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。

6.1.3.2.7 3.2.7 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。

6.1.3.2.8 3.2.8 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。

6.1.3.2.9 3.2.9 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。

6.1.3.3 3.3 TiDB server 管理

6.1.3.3.1 3.3.1 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。

6.1.3.3.2 3.3.2 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。

6.1.3.3.3 3.3.3 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。

6.1.3.3.4 3.3.4 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。

6.1.3.3.5 3.3.5 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。

6.1.3.3.6 3.3.6 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。

6.1.3.3.7 3.3.7 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。

6.1.3.3.8 3.3.8 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。

6.1.3.3.9 3.3.9 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。

6.1.3.3.10 3.3.9 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。

6.1.3.3.11 3.3.10 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。

6.1.3.3.12 3.3.11 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。

6.1.3.3.13 3.3.12 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，比如Optimizer Hints。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;`

6.1.3.3.14 3.3.13 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（后两个报错原因与表无关）：

- 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（v3.0.5 版本之前此值为定值 100。v3.0.5 及之后版本默认值为 1024，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。

> 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。

6.1.3.3.15 3.3.14 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。

6.1.3.3.16 3.3.15 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。

6.1.3.4 3.4 TiKV 管理

6.1.3.4.1 3.4.1 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。

6.1.3.4.2 3.4.2 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。

6.1.3.4.3 3.4.3 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。

6.1.3.4.4 3.4.4 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。

6.1.3.4.5 3.4.5 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。

6.1.3.4.6 3.4.6 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。

6.1.3.4.7 3.4.7 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。

6.1.3.4.8 3.4.8 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。

6.1.3.4.9 3.4.9 TiKV 在分别在那些场景下占用大量 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。

6.1.3.4.10 3.4.10 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。

6.1.3.4.11 3.4.11 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。

6.1.3.4.12 3.4.12 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。

6.1.3.4.13 3.4.13 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。

6.1.3.4.14 3.4.14 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。

6.1.3.4.15 3.4.15 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。

6.1.3.4.16 3.4.16 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。

6.1.3.4.17 3.4.17 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。

6.1.3.4.18 3.4.18 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。

6.1.3.4.19 3.4.19 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。

6.1.3.4.20 3.4.20 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。

6.1.3.4.21 3.4.21 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。

6.1.3.4.22 3.4.22 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。

6.1.3.5 3.5 TiDB 测试

6.1.3.5.1 3.5.1 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。

6.1.3.5.2 3.5.2 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。

6.1.3.5.3 3.5.3 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。

6.1.3.6 3.6 TiDB 备份恢复

6.1.3.6.1 3.6.1 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BR，并且 `mysqldump` 备份和恢复大量数据的耗费更长。

6.1.4 四、数据、流量迁移

6.1.4.1 4.1 全量数据导出导入

6.1.4.1.1 4.1.1 Mydumper

参见Mydumper 使用文档。

6.1.4.1.2 4.1.2 Loader

参见Loader 使用文档。

6.1.4.1.3 4.1.3 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。

6.1.4.1.4 4.1.4 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。

6.1.4.1.5 4.1.5 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。

6.1.4.1.6 4.1.6 如何导出 TiDB 数据？

TiDB 目前暂时不支持 `select into outfile`，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。

6.1.4.1.7 4.1.7 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。

6.1.4.1.8 4.1.8 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，不过这个会导致内存增加。

6.1.4.1.9 4.1.9 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。

6.1.4.2 4.2 在线数据同步

6.1.4.2.1 4.2.1 Syncer 架构

详细参考解析 TiDB 在线数据同步工具 Syncer。

4.2.1.1 Syncer 使用文档

详细参考Syncer 使用文档。

4.2.1.2 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。

4.2.1.3 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。

4.2.1.4 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

4.2.1.5 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。

4.2.1.6 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：

1）把 syncer.meta 数据放到比较安全的磁盘上，例如磁盘做好 raid1；

2）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。

4.2.1.7 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。

6.1.4.3 4.3 业务流量迁入

6.1.4.3.1 4.3.1 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。

6.1.4.3.2 4.3.2 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。

6.1.4.3.3 4.3.3 Transaction too large 是什么原因，怎么解决？

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
- KV entry 的总条数不超过 30w
- KV entry 的总大小不超过 100MB

在 Google 的 Cloud Spanner 上面，也有类似的限制。

6.1.4.3.4 4.3.4 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。

6.1.4.3.5 4.3.5 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。

6.1.4.3.6 4.3.6 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。

6.1.4.3.7 4.3.7 TiDB 是否支持 replace into 语法？

支持，但是 load data 不支持 replace into 语法。

6.1.4.3.8 4.3.8 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。

6.1.4.3.9 4.3.9 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。

6.1.4.3.10 4.3.10 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。

6.1.4.3.11 4.3.11 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。

6.1.5 五、SQL 优化

6.1.5.1 5.1 TiDB 执行计划解读

详细解读理解 TiDB 执行计划。

6.1.5.1.1 5.1.1 统计信息收集

详细解读统计信息。

6.1.5.1.2 5.1.2 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。

6.1.5.1.3 5.1.3 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。

6.1.5.1.4 5.1.4 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。

6.1.5.1.5 5.1.5 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。

6.1.5.1.6 5.1.6 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。

6.1.5.1.7 5.1.7 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。

6.1.5.1.8 5.1.8 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。

6.1.6 六、数据库优化

6.1.6.1 6.1 TiDB

6.1.6.1.1 6.1.1 TiDB 参数及调整

详情参考TiDB 配置参数。

6.1.6.1.2 6.1.2 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 PK 非整数或没有 PK 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TiDB 专用系统变量和语法中 SHARD_ROW_ID_BITS 的介绍。

6.1.6.2 6.2 TiKV

6.1.6.2.1 6.2.1 TiKV 性能参数调优

详情参考TiKV 性能参数调优。

6.1.7 七、监控

6.1.7.1 7.1 Prometheus 监控框架

详细参考TiDB 监控框架概述。

6.1.7.2 7.2 监控指标解读

详细参考重要监控指标详解。

6.1.7.2.1 7.2.1 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。

6.1.7.2.2 7.2.2 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"
```

6.1.7.2.3 7.2.3 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。

6.1.7.2.4 7.2.4 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。

6.1.7.2.5 7.2.5 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。

6.1.8 八、云上部署

6.1.8.1 8.1 公有云

6.1.8.1.1 8.1.1 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。

6.1.9 九、故障排除

6.1.9.1 9.1 TiDB 自定义报错汇总

6.1.9.1.1 9.1.1 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。

6.1.9.1.2 9.1.2 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。

6.1.9.1.3 9.1.3 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。

6.1.9.1.4 9.1.4 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。

6.1.9.1.5 9.1.5 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。

6.1.9.1.6 9.1.6 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。

6.1.9.1.7 9.1.7 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。

6.1.9.1.8 9.1.8 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。

6.1.9.2 9.2 MySQL 原生报错汇总

6.1.9.2.1 9.2.1 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。

6.1.9.2.2 9.2.2 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。

6.1.9.2.3 9.2.3 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。

6.1.9.2.4 9.2.4 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。

6.1.9.3 9.3 TiDB 日志中的报错信息

6.1.9.3.1 9.3.1 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。

6.2 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。

> **注意：**
>
> 使用 TiDB Lightning 的过程中如遇错误，参考TiDB Lightning 故障诊断进行排查。

6.2.1 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。最低版本要求是 2.0.9，但建议使用最新的稳定版本 3.0。

6.2.2 TiDB Lightning 支持导入多个库吗？

支持。

6.2.3 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 TiBD Lightning 额外需要以下权限：

- INSERT
- DELETE

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。

6.2.4 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。

6.2.5 如何正确重启 TiDB Lightning？

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。

6.2.6 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)
```

6.2.7 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. Mydumper 生成的 SQL dump
2. 储存在本地文件系统的CSV 文件

6.2.8 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。

6.2.9 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。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 = ""
...
```

6.2.10 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 Lightning 操作的表互不相同就可以。

6.2.11 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。

6.2.12 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。

6.2.13 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`。

6.2.14 为什么用过 TiDB Lightning 之后，TiDB 集群变得又慢又耗 CPU？

如果 `tidb-lightning` 曾经异常退出，集群可能仍留在“导入模式”(import mode)，不适合在生产环境工作。此时需要强制切换回“普通模式”(normal mode)：

```
tidb-lightning-ctl --switch-mode=normal
```

6.2.15 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"
```

6.2.16 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

6.2.17 TiDB Lightning 使用过程中是否可以重启 TiKV Importer？

不能，Importer 会在内存中存储一些引擎文件，Importer 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 Importer 特有的信息丢失了。你可以在之后重启 Lightning。

6.2.18 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. 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。

6.3 升级后常见问题

本文列出了一些升级后可能会遇到的问题与解决办法。

6.3.1 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：

1. 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。
2. 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。

6.3.1.1 问题 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
            }
        }
    ],
    ...
}
```

6.3.1.2 问题 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;
```

6.3.1.3 问题 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。

6.4 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。

6.4.1 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。

6.4.2 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。

6.4.3 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。

6.4.4 BR 遇到 Permission denied 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。

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。

6.4.5 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。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。

6.4.6 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。

6.4.7 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。

6.4.8 BR 恢复到 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。
- BR 恢复时产生的 DDL jobs 还可能会让 Drainer 执行异常的 DDL。所以，如果一定要在 Drainer 的上游集群执行恢复，请将 BR 恢复的所有表加入 Drainer 的阻止名单。

可以通过`syncer.ignore-table` 加入阻止名单。

7 技术支持

7.1 支持资源

您可以通过以下任何一种方式找到我们的社区成员：

- Slack Channel: https://pingcap.com/tidbslack
- Stack Overflow: https://stackoverflow.com/questions/tagged/tidb
- Reddit: https://www.reddit.com/r/TiDB
- GitHub: https://github.com/pingcap/tidb/issues

有关企业支持合作的信息，请联系我们。

7.2 提交 Issue

我们正在努力使 TiDB 与 MySQL 5.7 兼容。如果您发现了任何未记录在文档中的表现差异、bug 或奇怪的性能特征，请在 GitHub 中提交新的 Issue。

在 GitHub 上提交的新 Issue 分为以下几种：

- 错误报告 (Bug Reports)
- 功能请求 (Feature Requests)
- 常规问题 (General Questions)

- 性能问题 (Performance Questions)

请确保您完全按照 Issue 模板进行填写，以便我们迅速定位问题并作出回应。

8 贡献

8.1 成为贡献者

8.1.1 成为 TiDB 的贡献者

我们推荐您先尝试解决带 help-wanted 标签的现有 Issue，这些问题非常适合新的贡献者。

一旦您提交给 TiDB/TiKV/TiSpark/PD/TiDB Operator/Docs/Docs-cn 项目的 PR (Pull Request) 被批准且合并，您即成为 TiDB 的贡献者。

在提交 PR 之前，请先签署 CLA。

如果您想实践一个影响范围相对较小的新想法，请遵循以下步骤：

1. 提交新 Issue，描述您对相关仓库的更改建议。
2. 相关仓库的负责人将及时回复您的 Issue。
3. 如果您的更改建议被接受，您需要先签署 CLA，便可以在您的克隆仓库 (fork) 里开始工作了。
4. 在测试过所做的更改后，您便可以提交包含更改的 PR 了。

我们欢迎并非常感谢您的贡献。有关提交补丁和贡献流程的详细信息，请参阅贡献者指南。

8.1.2 改进文档

我们欢迎更多贡献者来帮助改进文档！

TiDB 文档使用 Markdown 语言，并参考了 Google 开发者文档风格指南进行编写。如果这些概念对您来说比较陌生，请不要担心，我们很乐意为您提供指导。

如要对文档仓库进行贡献，请先 fork docs-cn 仓库，再提交您的 PR。

9 版本发布历史

9.1 TiDB 版本发布历史

TiDB 历史版本发布声明如下：

9.1.1 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

9.1.2 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

9.1.3 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

9.1.4 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

9.1.5 1.0

- 1.0
- Pre-GA
- RC4
- RC3
- RC2
- RC1

9.2 v3.1

9.2.1 TiDB 3.1.2 Release Notes

发版日期：2020 年 6 月 4 日

TiDB 版本：3.1.2

9.2.1.1 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

9.2.2 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

9.2.2.1 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

9.2.2.2 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

9.2.3 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

9.2.3.1 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

9.2.3.2 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

9.2.3.3 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

9.2.4 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。

9.2.4.1 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

9.2.4.2 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

9.2.5 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。

9.2.5.1 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

9.2.5.2 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

9.2.5.3 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

9.2.6 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

9.2.6.1 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

9.2.6.2 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

9.2.6.3 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

9.2.7 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

9.2.7.1 TiDB

- SQL 优化器
  - 丰富 SQL hint #12192
- 新功能
  - TiDB 支持 Follower Read 功能 #12535

9.2.7.2 TiKV

- 支持分布式备份恢复功能 #5532
- TiKV 支持 Follower Read 功能 #5562

9.2.7.3 PD

- 支持分布式备份恢复功能 #1896

9.3 v3.0

9.3.1 TiDB 3.0.20 Release Notes

发版日期：2020 年 12 月 25 日

TiDB 版本：3.0.20

9.3.1.1 兼容性更改

- TiDB
  - 废弃配置文件中的 `enable-streaming` 配置项 #21054

9.3.1.2 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

9.3.1.3 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

9.3.2 TiDB 3.0.19 Release Notes

发版日期：2020 年 9 月 25 日

TiDB 版本：3.0.19

9.3.2.1 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

9.3.2.2 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

9.3.2.3 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

9.3.3 TiDB 3.0.18 Release Notes

发版日期：2020 年 8 月 21 日

TiDB 版本：3.0.18

9.3.3.1 提升改进

- Tools
  - TiDB Binlog
    * 支持更加细粒度的 Pump GC 时间 #996

9.3.3.2 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

9.3.4 TiDB 3.0.17 Release Notes

发版日期：2020 年 8 月 3 日

TiDB 版本：3.0.17

9.3.4.1 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

9.3.4.2 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

9.3.5 TiDB 3.0.16 Release Notes

发版日期：2020 年 7 月 3 日

TiDB 版本：3.0.16

9.3.5.1 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

9.3.5.2 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

9.3.6 TiDB 3.0.15 Release Notes

发版日期：2020 年 6 月 5 日

TiDB 版本：3.0.15

9.3.6.1 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

9.3.6.2 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

9.3.7 TiDB 3.0.14 Release Notes

发版日期：2020 年 5 月 9 日

TiDB 版本：3.0.14

9.3.7.1 兼容性变化

- TiDB

  - performance_schema 和 metrics_schema 由读写改为只读 #15417

9.3.7.2 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

9.3.7.3 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

9.3.7.4 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

9.3.8 TiDB 3.0.13 Release Notes

发版日期：2020 年 04 月 22 日

TiDB 版本：3.0.13

9.3.8.1 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

9.3.9 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。

9.3.9.1 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

9.3.9.2 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

9.3.9.3 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

9.3.10 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。

9.3.10.1 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

9.3.10.2 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

9.3.10.3 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

9.3.11 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。

9.3.11.1 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

9.3.11.2 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

9.3.11.3 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

9.3.11.4 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

9.3.11.5 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

9.3.12 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。

9.3.12.1 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

9.3.12.2 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

9.3.12.3 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

9.3.12.4 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

9.3.12.5 TiDB Ansible

- 优化 Lightning 部署，自动创建相关目录 #1105

9.3.13 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

9.3.13.1 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

- 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

9.3.13.2 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

9.3.13.3 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

9.3.13.4 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

9.3.13.5 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

9.3.14 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

9.3.14.1 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

9.3.14.2 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

9.3.15 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

9.3.15.1 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

9.3.15.2 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

9.3.15.3 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

9.3.15.4 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

9.3.16 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

9.3.16.1 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

9.3.16.2 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

9.3.16.3 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

9.3.16.4 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

9.3.16.5 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

9.3.17 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

9.3.17.1 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

9.3.17.2 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

9.3.17.3 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

9.3.17.4 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

9.3.17.5 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

9.3.18 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

9.3.18.1 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题 #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

9.3.18.2 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

9.3.18.3 PD

- `pdctl` 返回成功信息 #1685

9.3.18.4 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

9.3.18.5 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

9.3.19 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

9.3.19.1 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

9.3.19.2 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

9.3.19.3 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

9.3.19.4 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

9.3.19.5 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

9.3.20 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

9.3.20.1 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

9.3.20.2 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

9.3.20.3 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

9.3.20.4 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

9.3.20.5 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

9.3.21 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

9.3.21.1 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。

9.3.21.2 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

9.3.21.3 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

9.3.21.4 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

9.3.21.5 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

9.3.21.6 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

9.3.22 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

9.3.22.1 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。

9.3.22.2 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

9.3.22.3 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

9.3.22.4 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

9.3.22.5 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

9.3.22.6 TiDB Ansible

- 新增预测集群最大 QPS 的监控项（默认隐藏）#f5cfa4d

9.3.23 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

9.3.23.1 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。

9.3.23.2 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

9.3.23.3 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

9.3.23.4 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

9.3.23.5 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

9.3.24 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

9.3.24.1 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。

9.3.24.2 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

9.3.24.3 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

9.3.24.4 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

9.3.24.5 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

9.3.24.6 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

9.3.25 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

9.3.25.1 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。

9.3.25.2 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

9.3.25.3 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

9.3.25.4 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

9.3.25.5 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

9.3.26 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。

9.3.26.1 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

9.3.26.2 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

9.3.26.3 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

9.3.26.4 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

9.4 v2.1

9.4.1 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

9.4.1.1 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

9.4.1.2 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

9.4.1.3 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

9.4.1.4 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

9.4.1.5 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

9.4.2 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

9.4.2.1 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：TiDB 专用系统变量和语法 #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

9.4.2.2 PD

- 修复 pd-ctl --help 命令输出内容 #1772

9.4.2.3 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

9.4.2.4 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

9.4.3 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

9.4.3.1 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

9.4.3.2 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

9.4.3.3 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

9.4.3.4 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

9.4.3.5 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

9.4.4 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

9.4.4.1 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

9.4.4.2 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

9.4.4.3 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

9.4.4.4 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

9.4.5 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

9.4.5.1 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

9.4.5.2 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

9.4.5.3 PD

- 统一日志格式 #1625

9.4.5.4 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

9.4.5.5 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

9.4.6 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

9.4.6.1 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

9.4.6.2 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

9.4.6.3 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

9.4.6.4 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

9.4.6.5 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

9.4.7 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

9.4.7.1 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

9.4.7.2 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

9.4.7.3 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

9.4.8 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

9.4.8.1 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

9.4.8.2 PD

- 修复极端情况下 etcd Leader 选举阻塞的问题 #1576

9.4.8.3 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

9.4.9 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

9.4.9.1 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

9.4.9.2 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

9.4.9.3 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

9.4.9.4 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

9.4.9.5 TiDB Ansible

- 新增 Drainer 参数 #760

9.4.10 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

9.4.10.1 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

9.4.10.2 PD

- 修复 Leader 优先级不生效的问题 #1533

9.4.10.3 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

9.4.10.4 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

9.4.10.5 TiDB Ansible

- 更新 TiDB Lightning 配置文件，新增 `tidb_lightning_ctl` 脚本 #d3a4a368

9.4.11 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

9.4.11.1 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

9.4.11.2 PD

- 支持 GetOperator 服务 #1514

9.4.11.3 TiKV

- 修复在 transfer leader 时非预期的 quorum 变化 #4604

9.4.11.4 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

9.4.11.5 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

9.4.12 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

9.4.12.1 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

9.4.12.2 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

9.4.12.3 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

9.4.12.4 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

9.4.12.5 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

9.4.13 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

9.4.13.1 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

9.4.13.2 PD

- 修改副本数为 1 时 balance-region 无法迁移 leader 问题 #1462

9.4.13.3 Tools

- 支持 binlog 同步 generated column #491

9.4.13.4 TiDB Ansible

- Prometheus 监控数据默认保留时间改成 30d

9.4.14 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。

9.4.14.1 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

9.4.14.2 TiKV

- 修复在某些情况下解析 protobuf 失败导致 `StoreNotMatch` 错误的问题 #4303

9.4.14.3 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

9.4.15 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。

9.4.15.1 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

9.4.15.2 PD

- GetAllStores 接口提供了 exclude_tombstone_stores 选项，将 Tombstone store 从返回结果中去除 #1444

9.4.15.3 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

9.4.15.4 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

9.4.16 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。

9.4.16.1 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

9.4.16.2 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

9.4.16.3 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

9.4.17 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。

9.4.17.1 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

9.4.17.2 PD

- 修复 Leader 选举相关的 Watch 问题 #1396

9.4.17.3 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

9.4.17.4 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

9.4.18 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。

9.4.18.1 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

9.4.18.2 PD

- 修复 Region Merge 相关的 Region 信息更新问题 #1377

9.4.18.3 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

9.4.18.4 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

9.4.19 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。

9.4.19.1 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

9.4.19.2 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

9.4.19.3 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

9.4.19.4 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

9.4.20 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。

9.4.20.1 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

9.4.20.2 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

9.4.20.3 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

9.4.20.4 Tools

- 全量数据快速导入工具 TiDB Lightning
- 支持新版本 TiDB Binlog

9.4.20.5 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

9.4.21 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。

9.4.21.1 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

9.4.21.2 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

9.4.21.3 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

9.4.21.4 Tools

- TiDB 支持 TiDB Binlog cluster，不兼容旧版本 TiDB Binlog #8093，使用文档

9.4.22 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。

9.4.22.1 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

9.4.22.2 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

9.4.22.3 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

9.4.23 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。

9.4.23.1 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

9.4.23.2 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

9.4.23.3 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

9.4.24 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。

9.4.24.1 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

9.4.24.2 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

9.4.24.3 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

9.4.25 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。

9.4.25.1 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`

9.4.25.2 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

9.4.25.3 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

9.4.26 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。

9.4.26.1 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

9.4.26.2 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

9.4.26.3 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

9.5 v2.0

9.5.1 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。

9.5.1.1 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

9.5.1.2 TiKV

- 修复了两个 Region merge 相关的问题 #4003，#4004

9.5.2 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。

9.5.2.1 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

9.5.2.2 PD

- 修复 RaftCluster 在退出时可能的死锁问题 #1370

9.5.2.3 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

9.5.3 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。

9.5.3.1 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

9.5.3.2 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

9.5.3.3 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

9.5.4 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。

9.5.4.1 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

9.5.4.2 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

9.5.5 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。

9.5.5.1 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

9.5.5.2 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

9.5.6 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。

9.5.6.1 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

9.5.6.2 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

9.5.7 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。

9.5.7.1 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

9.5.7.2 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

9.5.7.3 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

9.5.8 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。

9.5.8.1 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

9.5.8.2 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

9.5.8.3 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

9.5.9 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。

9.5.9.1 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

9.5.9.2 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

9.5.9.3 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

9.5.10 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。

9.5.10.1 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

9.5.10.2 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

9.5.10.3 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

9.5.11 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。

9.5.11.1 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

9.5.11.2 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

9.5.11.3 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

9.5.12 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。

9.5.12.1 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

9.5.12.2 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

9.5.12.3 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

9.5.12.4 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

9.5.13 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。

9.5.13.1 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

9.5.13.2 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

9.5.13.3 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

9.5.14 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。

9.5.14.1 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

9.5.14.2 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

9.5.14.3 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

9.5.15 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。

9.5.15.1 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

9.5.15.2 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

9.5.15.3 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

9.5.16 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。

9.5.16.1 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

9.5.16.2 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

9.5.16.3 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

9.5.17 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。

9.5.17.1 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

9.5.17.2 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

9.5.17.3 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

9.5.18 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。

9.5.18.1 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

9.5.18.2 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

9.5.18.3 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

9.6 v1.0

9.6.1 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。

9.6.1.1 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

9.6.1.2 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

9.6.1.3 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

9.6.1.4 TiSpark Beta Release

- 支持可配置框架
- 支持 ThriftSever/JDBC 和 Spark SQL 脚本入口

9.6.1.5 源码地址

源码地址

9.6.1.6 鸣谢

9.6.1.6.1 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

9.6.1.6.2 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

9.6.1.6.3 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

9.6.2 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。

9.6.2.1 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

9.6.2.2 PD

- 支持手动切换 PD 集群 Leader

9.6.2.3 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

9.6.2.4 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)

9.6.3 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。

9.6.3.1 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

9.6.3.2 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

9.6.3.3 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

9.6.3.4 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)

9.6.4 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。

9.6.4.1 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

9.6.4.2 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

9.6.4.3 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

9.6.5 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。

9.6.5.1 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

9.6.5.2 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

9.6.5.3 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

9.6.6 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。

9.6.6.1 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

9.6.6.2 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

9.6.6.3 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

9.6.6.4 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

10 术语表

10.1 A

10.1.1 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。

10.2 L

10.2.1 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。

10.3 O

10.3.1 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。

10.3.2 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

10.4 P

10.4.1 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。

10.5 R

10.5.1 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。

10.5.2 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。

10.5.3 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。

10.6 S

10.6.1 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）

10.6.2 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。
